POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Tomasz Gajger
Nr albumu: 143218
Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność/profil: -

Imię i nazwisko studenta: ANDRZEJ PODGÓRSKI
Nr albumu: 143321
Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność/profil: -

Imię i nazwisko studenta: BARTOSZ POLLOK
Nr albumu: 143322
Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność/profil: -

PROJEKT DYPLOMOWY INŻYNIERSKI

Tytuł projektu w języku polskim: Podsystem do zarządzania zużyciem energii dla heterogenicznego środowiska obliczeń wysokiej wydajności

Tytuł projektu w języku angielskim: A module for management of energy consumption for a heterogeneous HPC system

Potwierdzenie przyjęcia projektu	
Opiekun projektu *podpis*	Kierownik Katedry/Zakładu *podpis*
dr hab. inż. Paweł Czarnul	

Data oddania projektu do dziekanatu:

STRESZCZENIE

Przedmiotem niniejszej pracy jest zaprojektowanie, a następnie dokonanie implementacji systemu, pozwalającego na zarządzanie w jednolity i scentralizowany sposób zużyciem energii w heterogenicznym (składającym się z wielu urządzeń różnego typu) i rozproszonym środowisku obliczeniowym. Problemem, który należało rozwiązać, są znaczne różnice w konstrukcji bibliotek dostarczanych przez producentów urządzeń i idąca za tym konieczność wchodzenia w interakcję z każdą z nich w specyficzny dla konkretnej biblioteki sposób. Wykonany w ramach pracy rozproszony system umożliwia zarządzanie zużyciem energii w sposób niezależny od urządzenia i scentralizowany, a jego architektura została przemyślana tak, by jego dalsze rozszerzanie było możliwie jak najłatwiejsze. Korzystając z doświadczeń wyniesionych z fazy implementacji, dokonano również analizy porównawczej bibliotek: NVML, MPSS i NMPRK, wskazując na podobieństwa i różnice między nimi oraz ich mocne i słabe strony. W wyniku tej analizy jednoznacznie stwierdzono, że na najlepszą ocenę zasługuje biblioteka NVML, a na najgorszą – NMPRK. Przeprowadzono również szereg testów wytworzonego systemu w środowisku, w którego skład wchodzą akceleratory Intel Xeon Phi oraz Nvidia Tesla. Wszystkie testy zakończyły się sukcesem i potwierdziły prawidłowe działanie systemu w praktyce.

Tomasz Gajger wykonał modelowanie biznesowe systemu, zaprojektował ogólną architekturę całego rozwiązania, zaimplementował agenta i przeprowadził analizę porównawczą bibliotek służących do zarządzania zużyciem energii. Andrzej Podgórski miał udział w projektowaniu architektury systemu, przygotował środowisko deweloperskie, zaimplementował większość serwera, część interfejsu użytkownika oraz napisał całościowe testy aplikacji (ang. *end to end*). Bartosz Pollok zaimplementował większość interfejsu użytkownika, a także część serwera, nadzorował wspólne uruchamianie wszystkich podsystemów w środowisku przedprodukcyjnym i przeprowadził w nim testy integracyjne. Testy ukończonego systemu w środowisku docelowym zostały wykonane wspólnym wysiłkiem wszystkich trzech autorów.

Tomasz Gajger – udział w rozdziałach: 1, 6, 9, indywidualnie rozdziały: 3, 7 oraz podrozdziały: 2.1, 2.2, 4.1, 4.2, 4.4, 5.1, 6.1, 8.3, 8.8 i indywidualnie dodatki A i B. Andrzej Podgórski – udział w rozdziałach: 1, 6, 9 oraz indywidualnie podrozdziały: 2.3, 4.5, 5.2, 8.2, 8.4, 8.7. Bartosz Pollok – udział w rozdziałach: 1, 6, 9 oraz indywidualnie podrozdziały: 2.4, 4.3, 4.6, 5.3, 8.1, 8.5, 8.6.

Słowa kluczowe: zużycie energii, akceleratory obliczeniowe, heterogeniczne środowiska obliczeniowe, obliczenia wysokiej wydajności, NVML, MPSS, NMPRK.

Dziedzina nauki i techniki, zgodnie z wymogami OECD: nauki inżynieryjne i techniczne, inżynieria informatyczna.

ABSTRACT

The purpose of this study is to design and then implement a system which would allow to manage the energy consumption of a heterogeneous (consisting of many devices of various types) and distributed computation environment in a centralized and uniform manner. The main problem that had to be solved were the differences in design and APIs of power management libraries supplied by the manufacturers of computational devices and thus the need to interact with each of them in that library's specific way. The system created as a part of this thesis provides a device independent and centralized way to manage the power consumption and furthermore, its architecture was designed with the goal to be as much extensible as possible. Based on the experiences from the implementation phase, comparison of following libraries: NVML, MPSS and NMPRK was made, in which similarities and differences as well as pros and cons of each of them were pointed out. The conclusion of this analysis is that from aforementioned libraries, NVML deserves the highest grade and the NMPRK has proven to be the worst one. Finally, several test scenarios in an environment consisting of Intel Xeon Phi accelerators and Nvidia Tesla GPUs were executed, every single of them completed successfully and thus the correct functioning of implemented system was confirmed.

Keywords: energy consumption, accelerators, heterogeneous computational environments, high performance computing, NVML, MPSS, NMPRK.

SPIS TREŚCI

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

API – Application Programming Interface
CPU – Central Processing Unit
FLOPS – Floating Point Operations Per Second
GCC – GNU Compiler Collection
GPGPU – General-Purpose computing on GPU
GPU – Graphics Processing Unit
HTML – Hypertext Markup Language
HTTP – Hypertext Transfer Protocol
JS – JavaScript
JSON – JavaScript Object Notation
KASK – Katedra Architektury Systemów Komputerowych
MPSS – Manycore Platform Software Stack
NMPRK – Node Manager Programmer's Reference Kit
NVML – NVIDIA Management Library
PCI – Peripheral Component Interconnect
PEP – Python Enhancement Proposal
PU – Przypadek użycia
RAM – Random Access Memory
REST – Representational State Transfer
SDK – Software Development Kit
SPA – Single Page Application
SSH – Secure Shell
UI – User Interface
UUID – Universally Unique Identifier
VPS – Virtual Private Server

1. WSTĘP I CEL PRACY

Kilkanaście lat temu przy tworzeniu superkomputerów zwracano uwagę jedynie na moc obliczeniową, a zużycie energii nie odgrywało większej roli. Wybór rozwiązań był również mocno ograniczony, nie istniały układy w rodzaju Intel Xeon Phi, a paradygmat GPGPU[1] dopiero zaczynał się rozwijać [Kirk D., Hwu W., 2010], co jak się później okaże mocno ograniczało możliwości poprawy wydajności energetycznej. Jednak już wtedy można było zauważyć wzrastającą popularność klastrów z węzłami wyposażonymi w urządzenia wielordzeniowe [Czarnul P., 2006].

Obecnie akceleratory różnego rodzaju, charakteryzujące się znacznie większą wydajnością energetyczną niż procesory [Li B. i inni, 2014], zdobywają coraz większą popularność i często stają się podstawowym elementem składowym we współczesnych rozwiązaniach przeznaczonych do wykonywania obliczeń zarówno w kręgach akademickich, rządowych, jak i przemysłowych.

Dzieje się to pod wpływem wielu czynników, do których można przykładowo zaliczyć:

- przesłanki ekonomiczne – zmniejszanie wydatków na energię elektryczną;
- przesłanki ideologiczne – ograniczanie zużycia energii by być „przyjaznym" dla środowiska;
- zużycie energii nabierające coraz bardziej na znaczeniu jako czynnik ograniczający możliwości rozbudowywania systemów obliczeniowych;
- dynamiczny rozwój technologii chmurowych, wymagający tworzenia coraz większych farm serwerów.

Trend ten można bez wątpienia uznać za coś bardzo pozytywnego, jednak niesie on za sobą pewną oczywistą konsekwencję – środowisko z homogenicznego, w którym do obliczeń wykorzystywane były jedynie procesory, staje się heterogeniczne, a więc wyposażone zarówno w procesory, jak i akceleratory [Czarnul P., Rościszewski P., 2014]. Skoro zaczęto interesować się zużyciem energii podczas wykonywanych obliczeń i pojawiła się chęć optymalizowania kosztów, potrzebne są rozwiązania, które to umożliwią. Producenci poszczególnych urządzeń, wychodząc naprzeciw oczekiwaniom swoich klientów, udostępniają im biblioteki pozwalające na monitorowanie i zarządzanie zużyciem energii tychże urządzeń.

Problem z zarządzaniem w środowisku heterogenicznym jest bardzo prosty w swojej naturze. Różne urządzenia oznaczają różne standardy zarządzania, a co za tym idzie konieczność obsługi kilku interfejsów na raz. Nie jest to ani wygodne, ani pożądane, ale często jedyne rozwiązanie. Wprawdzie istnieją systemy umożliwiające monitorowanie takich środowisk (np. Ganglia), ale nie zapewniają one pewnego istotnego elementu – automatycznego reagowania na zmiany, jak również nie dają użytkownikowi możliwości swobodnego wprowadzania zautomatyzowanej kontroli zużycia energii w poszczególnych elementach środowiska. Drugim istotnym czynnikiem jest tendencja do tworzenia rozproszonych środowisk

[1] GPGPU (ang. *general-purpose computing on graphics processing units* - obliczenia ogólnego przeznaczenia na układach GPU) – technologia, dzięki której procesor graficzny umożliwia wykonywanie obliczeń ogólnego przeznaczenia, które zwykle wykonuje procesor główny [Wilk A., 2009].

obliczeniowych, składających się z dużej liczby węzłów, które mogą być umiejscowione w różnych lokalizacjach. Pożądana byłaby więc możliwość wprowadzenia scentralizowanego zarządzania takimi środowiskami.

Celem niniejszej pracy jest dostarczenie aplikacji umożliwiającej zarządzanie w jednolity i scentralizowany sposób zużyciem energii na różnych rodzajach urządzeń w środowisku rozproszonym. Można wyróżnić cztery główne zadania realizowane przez dostarczony system:

1. nałożenie warstwy abstrakcji na różnice w poszczególnych bibliotekach i udostępnienie użytkownikowi jednego, spójnego interfejsu;
2. umożliwienie użytkownikowi określania reguł zarządzania energią tak, by zapewnić jak największą elastyczność w modelowaniu zużycia energii;
3. prezentowanie użytkownikowi statystyk zużycia energii, aby mógł zweryfikować skuteczność zdefiniowanych przez siebie reguł i poprawność działania samego systemu;
4. udostępnienie centralnego punktu umożliwiającego zarządzanie całym środowiskiem.

Motywacją do podjęcia niniejszej pracy była również możliwość integracji w przyszłości wytworzonej w jej ramach aplikacji z realizowanym na Katedrze Architektury Systemów Komputerowych systemem Kernel Hive[2], gdzie aplikacja stanowiłaby podsystem umożliwiający zarządzanie zużyciem energii. Sama integracja nie jest przedmiotem tej pracy, ale stanowiła istotne źródło wymagań, które należało uwzględnić.

Prezentowana praca nie ma na celu analizy czy dostarczenia gotowych algorytmów lub rozwiązań pozwalających na zarządzanie energią w sposób optymalny. Zamiast tego nacisk kładziony jest na stworzenie programu, który umożliwi dokonywanie takich analiz w przyszłości, zdejmując z użytkownika ciężar obsługi różniących się od siebie bibliotek. Istotnym elementem pracy jest natomiast opis i analiza porównawcza tychże bibliotek, która bazuje na doświadczeniach zebranych podczas implementacji samego rozwiązania oraz na wynikach testów praktycznych przeprowadzonych z wykorzystaniem już w pełni sprawnego systemu.

W rozdziale 1. przestawione zostało krótkie wprowadzenie do zagadnień poruszanych w pracy, motywacja do jej podjęcia oraz cel, który chciano osiągnąć. Rozdział 2. zawiera krótkie porównanie urządzeń obliczeniowych. Proces modelowania biznesowego systemu został opisany w rozdziale 3. Rozdział 4. ujmuje w sposób szczegółowy architekturę całego rozwiązania, a proces implementacji opisany został w rozdziale 5. Wyniki testów przeprowadzonych z wykorzystaniem zaimplementowanego systemu zostały zawarte w rozdziale 6. W rozdziale 7., korzystając z wniosków wyciągniętych na etapie implementowania rozwiązania, porównano biblioteki służące do zarządzania zużyciem energii. Proces realizacji projektu udokumentowano w rozdziale 8. Rozdział 9. zawiera podsumowanie całej pracy i propozycję jej kontynuacji.

[2] http://eti.pg.edu.pl/katedra-architektury-systemow-komputerowych/kernelhive

2. PRZEGLĄD WYBRANYCH URZĄDZEŃ OBLICZENIOWYCH

2.1. Wprowadzenie

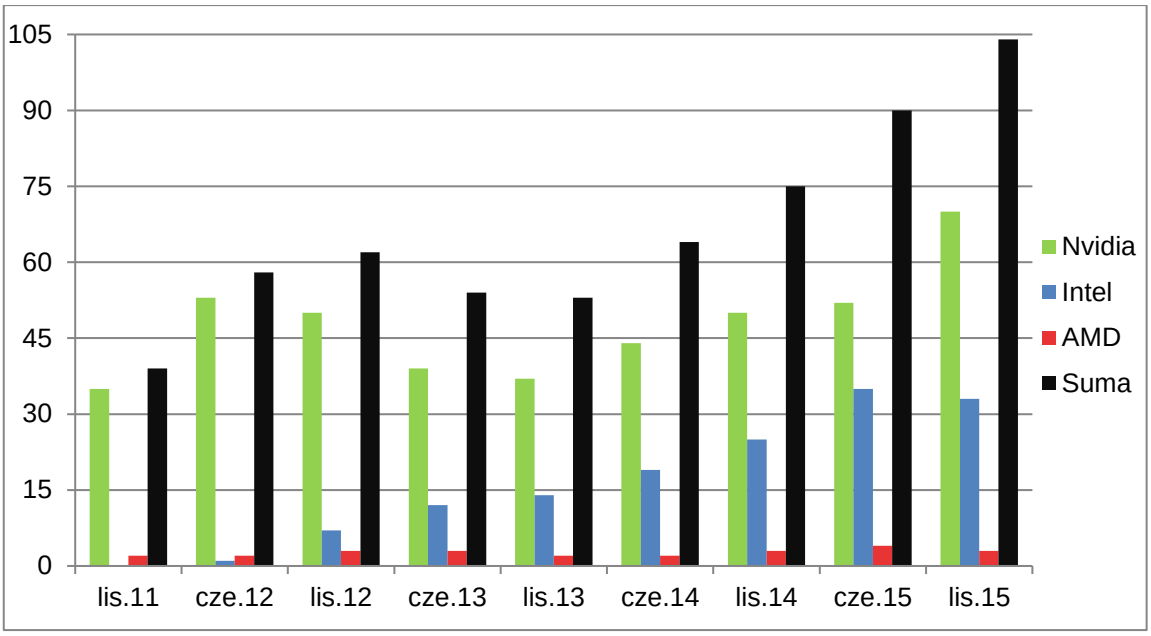
Urządzenia obliczeniowe można podzielić na trzy główne grupy, za kryterium podziału przyjmując ich ogólną charakterystykę, ale w szczególności ilość, możliwości i stopień skomplikowania rdzeni wchodzących w skład tychże urządzeń.

W pierwszej kolejności mamy procesory główne. Posiadają one małą ilość bardzo złożonych rdzeni o ogromnych możliwościach i taktowanych zegarami rzędu 2, 3 czy nawet 4 GHz. Rdzenie te wyposażone są w wielopoziomowe pamięci podręczne o relatywnie dużej pojemności. Mają zaawansowane układy przewidywania rozgałęzień, rozbudowane potoki czy wreszcie mechanizmy wykonywania instrukcji poza kolejnością.

Kolejną grupą są koprocesory, które można uznać za pewnego rodzaju ogniwo pośrednie pomiędzy procesorami głównymi a kartami graficznymi. Należy zaznaczyć, że nie mamy tutaj na myśli koprocesorów, które wchodzą w skład procesorów głównych, a samodzielne urządzenia podłączane przez złącze PCIe[3], podobnie jak karty graficzne. Rdzenie tych urządzeń są znacznie prostsze w swojej konstrukcji. Taktowane są zegarami o mniejszej częstotliwości, ale jest ich zdecydowanie więcej – dziesiątki w jednym urządzeniu, a jeżeli wziąć pod uwagę ich zwielokrotnienie wynikające z wykorzystania zmodyfikowanej technologii Intel *Hyper-Threading*[4], to osiągamy wartości rzędu 260 rdzeni. Rdzenie zostają też wyposażone w rozbudowane jednostki wektorowe, mające na celu dalsze zwiększenie mocy przez umożliwienie jeszcze efektywniejszego zrównoleglenia wykonywanych instrukcji. Należy również zauważyć, że w urządzeniach tych hierarchia pamięci poszerza się o kolejną pozycję. Do pamięci podręcznej (najszybszej, o małej pojemności) i pamięci operacyjnej komputera (wolniejszej, o dużej pojemności) dołącza pamięć urządzenia (szybka, o średniej pojemności).

Ostatnią grupą są karty graficzne, dla których liczby rdzeni obliczeniowych podaje się w setkach, a nawet w tysiącach. Oczywistą konsekwencją jest prostota konstrukcji i małe możliwości pojedynczego rdzenia oraz dodatkowe wymagania wymuszane przez model przetwarzania, które należy brać pod uwagę, aby w pełni wykorzystać możliwości obliczeniowe tych układów (na przykład takie jak: charakterystykę odwołań do pamięci czy rozbiegania się ścieżek wykonania w skutek występowania instrukcji warunkowych) [Czarnul P., 2014]. Oczywiście obostrzenia takie występują również w dwóch wcześniej wymienionych typach urządzeń, jednak w znacznie mniejszym zakresie.

[3] PCIe (ang. *Peripheral Component Interconnect Express*) – połączenie typu punkt-punkt, służące do instalacji kart rozszerzeń na płycie głównej. Zastąpiło magistralę PCI.
[4] *Hyper-Threading* (hiperwątkowość) – technologia pozwalająca na wykonywanie przez procesor obliczeń z kilku wątków jednocześnie przez zduplikowanie pewnych jego elementów [Jędruch A., 2013].



Rys. 2.1. Liczba superkomputerów na liście TOP500 [8] wykorzystujących akceleratory

Poszczególne pozycje mogą się nie sumować do wartości podanej jako suma, ponieważ jeden system może zwierać akceleratory różnego typu, np. Nvidia Tesla i Intel Xeon Phi. W zestawieniu pojawiają się akceleratory od innych producentów: IBM i Pezy, nie zostały jednak one uwzględnione na wykresie, aby nie pogarszać jego czytelności.

2.2. Karty graficzne

Karty graficzne można podzielić na kilka segmentów w zależności od przeznaczenia. Można wyróżnić:

- karty do komputerów osobistych – przeznaczone do renderowania grafiki, ale mogące również służyć do wykonywania innych obliczeń: Nvidia GeForce, AMD (ang. *Advanced Micro Devices*) Radeon;
- karty do urządzeń mobilnych – charakteryzujące się znacznie mniejszym poborem energii, co w oczywisty sposób przekłada się na ich wydajność: Nvidia Tegra;
- karty do profesjonalnej obróbki grafiki czy aplikacji CAD (ang. *computer aided design* - projektowanie wspomagane komputerowo*)* – Nvidia Quadro, AMD FirePro seria W;
- karty klasy serwerowej – przeznaczone do środowisk obliczeń wysokiej wydajności: Nvidia Tesla, AMD FirePro seria S, wyposażone w pamięć ECC (ang. *Error Checking and Correction* – sprawdzanie i korekcja błędów).

Początkowo urządzenia te były wykorzystywane jedynie do obliczeń związanych z renderowaniem grafiki, jednak z czasem zauważono, że ich potencjał jest znacznie większy, co skutkowało dynamicznym rozwojem paradygmatu GPGPU [Sanders J., Kandrot E., 2011]. Rozwój ten doprowadził do powstania dedykowanych rozwiązań znacznie ułatwiających stosowanie kart graficznych do ogólnie rozumianych obliczeń, takich jak Nvidia CUDA (ang. *Compute Unified Device Architecture* – technologia typu GPGPU autorstwa Nvidii), OpenACC (ang. *Open Accelerators* – otwarty standard programowania heterogenicznych środowisk

złożonych z procesorów ogólnego przeznaczenia i graficznych) czy OpenCL (ang *Open Computing Language* – podobnie jak OpenACC).

Karty graficzne oprócz wysokiej mocy obliczeniowej, charakteryzują się bardzo dużą wydajnością energetyczną [Ge R. i inni, 2013], co dobrze widać w tabeli 2.1. Wydajność ta dwojako przekłada się na zmniejszenie kosztów ich eksploatacji. Po pierwsze oznacza ona mniejszy pobór prądu, a po drugie zmniejszoną emisję ciepła, a więc mniejsze nakłady na schłodzenie całego systemu. W związku z tym karty graficzne są coraz powszechniej wykorzystywane jako elementy współczesnych superkomputerów, co widać na rys. 2.1. i wchodzą w skład tych najwydajniejszych pod względem zużycia energii, co pokazuje tabela 2.2. Wysoka wydajność energetyczna jest również bardzo pożądana w segmencie urządzeń mobilnych, gdzie przekłada się ona bezpośrednio na czas pracy baterii, a więc kulturę pracy urządzenia i komfort jego użytkowania.

Przedmiotem naszego zainteresowania będzie ostatnia grupa z prezentowanej wcześniej klasyfikacji, a więc urządzenia klasy serwerowej, a ze względu na dostępność sprzętu w laboratorium katedralnym ograniczymy się do kart z serii Nvidia Tesla. Monitorowanie i zarządzanie zużyciem energii tychże kart możliwe jest za pomocą biblioteki NVML (ang. *Nvidia Management Library* – biblioteka zarządzająca Nvidia). Urządzenia z serii GeForce i Quadro również współpracują z NVML, jednak nie wspierają one kluczowych z punktu widzenia niniejszej pracy funkcji, tak więc nie będziemy się nimi zajmować.

2.3. Koprocesory

Koprocesorem można nazwać właściwie dowolny układ wspomagający procesor główny. Również karty graficzne oraz dźwiękowe (ze sprzętowym przetwarzaniem dźwięku) można uznać za koprocesory, termin ten jednak nie przyjął się w stosunku do nich.

Najczęściej koprocesorem nazywany jest układ FPU (ang. *Floating Point Unit* – jednostka zmiennoprzecinkowa). Obecnie jest on zintegrowany z CPU (ang. *Central Processing Unit* – centralna jednostka obliczeniowa) w jednym kawałku krzemu, jednak dawniej był to osobny układ umieszczany w specjalnej podstawce na płycie głównej. Wspomaga on procesor główny w obliczeniach na liczbach zmiennoprzecinkowych [Jędruch A., 2013]. Jest on bardzo ważny z punktu widzenia obliczeń wysokiej wydajności, jednak z powodu, że nie jest on już odrębnym urządzeniem, nie był przedmiotem zainteresowania pracy.

Innym przykładem koprocesora może być PhysX P1 zaprezentowany w 2006 roku przez firmę Ageia. Nazywany jest on układem PPU (ang. *Physics Processor Unit* – jednostka procesora fizyki). Miał on służyć do wspomagania obliczeń fizycznych w grach komputerowych oraz do wydajnego modelowania zjawisk fizycznych. Z tego względu mogłoby to być ciekawe urządzenie dla obliczeń wysokiej wydajności [Stolarczyk S., 2006]. Karta jednak nie cieszyła się dużą popularnością, a firma Ageia w 2008 roku została przejęta przez Nvidię. Model programowy PhysX został zaadaptowany do działania na GPU wspierających Nvidia CUDA, a wytwarzanie koprocesorów zostało porzucone.

Obecnie jednymi z najbardziej interesujących koprocesorów z punktu widzenia wysokowydajnych obliczeń są Intel Xeon Phi, wyposażone w zależności od modelu w od 57 do 61 rdzeni taktowanych zegarem od 1,053 do 1,238 GHz oraz posiadające od 6 do 16 GB pamięci RAM. Każdy z rdzeni ma architekturę zbliżoną do pierwszych procesorów Pentium, jednak ze wsparciem dla trybu 64-bitowego, 512-bitowych instrukcji SIMD[5] oraz czterech sprzętowych wątków (*Hyper-Threading*). Pamięć podręczna jest koherentna, to znaczy spójna między rdzeniami. Karty mają osobny system operacyjny – specjalnie przygotowaną dystrybucję Linuksa i mogą być zarządzane przy użyciu znanych z Linuksa poleceń. Producent zapewnia o bardzo dobrym stosunku wydajności do poboru energii (co po części jest prawdą, jak widać w tabeli 2.1.) oraz o dużych możliwościach zarządzania zużyciem energii [Intel, 2015]. Z tego powodu oraz ze względu na dość dużą popularność oraz obiecującą architekturę urządzeń, wybrano je jako jedno z wspieranych przez wytwarzany w ramach tej pracy system. Szczegółowy opis sposobu zarządzania zużyciem energii w urządzeniach Intel Xeon Phi znajduje się w rozdziale 7.

2.4. Procesory

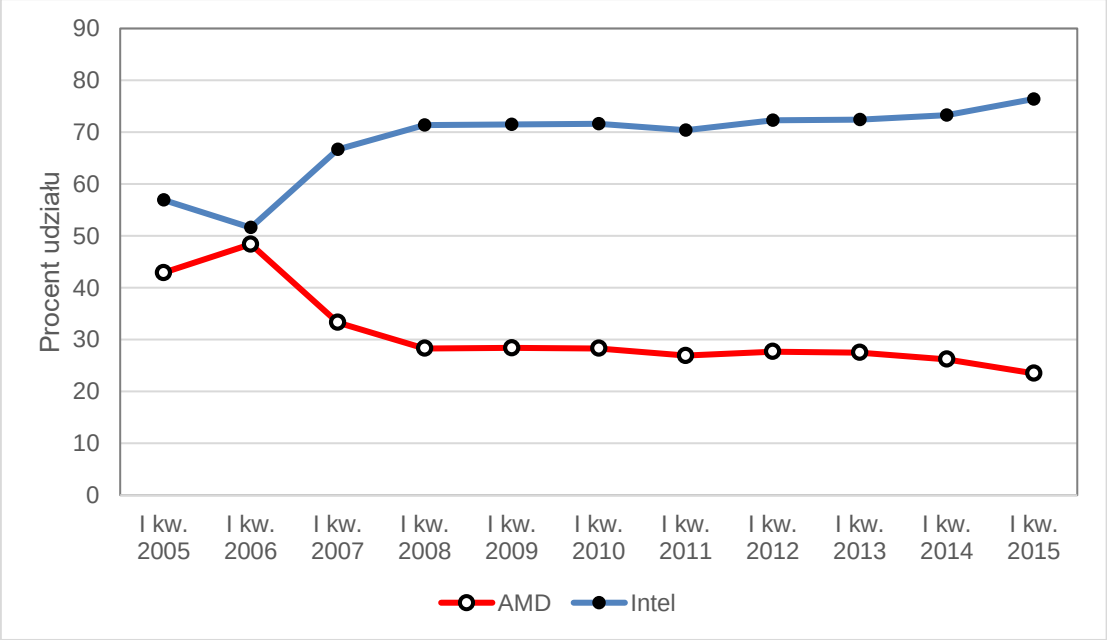
Procesory są nieodzownym elementem każdego systemu komputerowego. W zależności od zastosowań można podzielić je na następujące kategorie:

- procesory do urządzeń przenośnych – najczęściej architektura ARM (ang. *Advanced RISC[6] Machine* – zaawansowana maszyna RISC), niskie zużycie energii, np. Cortex-A17 (ARMv7-A);
- procesory do komputerów klasy PC (ang. *Personal Computer* – komputer osobisty) – szeroki wachlarz modeli (od energooszczędnych po wysokowydajne), architektura x86/x64, niewielka ilość rdzeni (od 1 do kilku), np. Intel Pentium, Intel Core, AMD Phenom, VIA C7;
- procesory serwerowe – często klastrowane, większa liczba rdzeni, wysokowydajne (jak najwyższa wydajność przy jak najniższym zużyciu energii), wyposażone w szereg rozwiązań specjalnego przeznaczenia (m.in. z zakresu kryptografii, wirtualizacji, szybkiego dostępu do pamięci, kontroli błędów, zarządzania energią), np. AMD Opteron, Intel Xeon.

Pomimo ciągłego zwiększania wydajności procesorów przeznaczonych dla komputerów osobistych, ustępują one procesorom serwerowym na wielu polach, gdyż przyrost ich wydajności skupia się na obsłudze aplikacji przeznaczonych do zapewniania rozrywki, jak np. gry komputerowe. Procesory serwerowe mają nad nimi przewagę ze względu na łatwość łączenia ich w większe zespoły, obsługę technologii takich jak ECC oraz nacisk na wykonywanie specyficznych zadań, w tym obliczeń. Rynek procesorów zdominowany jest obecnie przez dwie firmy – AMD oraz Intel [13]. Ich udział w rynku obrazuje wykres na rys. 2.2.

[5] SIMD (ang. *Single Instuction, Multiple Data* – jedna instrukcja, wiele danych) – rodzaj architektury komputerowej obejmujący systemy, w których przetwarzanych jest wiele strumieni danych w oparciu o pojedynczy strumień rozkazów [Jędruch A., 2013].
[6] RISC (ang. *Reduced Instruction Set Computing* – obliczenia na zredukowanym zbiorze instrukcji) – rodzaj architektury komputerów o małej liczbie rozkazów i dużej liczbie rejestrów [Jędruch A., 2013].

Za warte uwzględnienia w ramach niniejszej pracy uznano procesory klasy serwerowej. Konkretnymi modelami, których obsługę zaimplementowano, są procesory Intel Xeon, ze względu na dostępność takowych w katedrze, a również ze względu na istniejący interfejs programistyczny i biblioteki do zarządzania zużyciem energii, tj. NMPRK (ang. *Node Manager Programmer's Reference Kit*), która została opisana w rozdziale 7.



Rys. 2.2. Udział firm Intel i AMD w rynku procesorów x86/x64

Procesory Intel Xeon, będące przedmiotem rozważań tej pracy, wykorzystywane są również w czołówce najwydajniejszych pod względem energetycznym superkomputerów, zwanej GREEN500 [14], co obrazuje tabela 2.2.

Tabela 2.1. Teoretyczna wydajność energetyczna wybranych urządzeń obliczeniowych dla obliczeń pojedynczej precyzji.

Urządzenie	Wydajność [GFLOPS/wat]
Nvdia Tesla K80	29,12
Nvdia Tesla K40	18,26
Nvdia Tesla K20	16,75
Nvdia Tesla K20x	15,66
Intel Xeon Phi 5110P	8,99
Intel Xeon Phi 7120P	8,06
Intel Xeon E5-2698v3	4,36
Intel Xeon E5-2697v3	4,02
Intel Xeon E5-2680v3	4,00
Intel Xeon E5-2697v2	3,99
Intel Xeon E5-2670	2,89

Sposób obliczania wartości przedstawionych w tabeli 2.1 został dokładnie omówiony w dodatku A. Urządzenia zostały dobrane tak, aby stanowiły jak najbardziej reprezentatywną grupę, a więc należały do różnych segmentów wydajnościowych oraz były obecne w zestawieniu TOP500.

Tabela 2.2. Najwydajniejsze pod względem energetycznym superkomputery na świecie

Pozycja	Liczba rdzeni procesorów	Liczba rdzeni akceleratorów	Procesory	Akceleratory	MFLOPS/wat
1	1536	786432	Intel Xeon E5-2618Lv3	PEZY-SC	7031,57
2	1024	262144	Intel Xeon E5-2618Lv3	PEZY-SC	6842,32
3	640	262144	Intel Xeon E5-2660v2	PEZY-SC	6217,04
4	1120	9856	Intel Xeon E5-2690v2	AMD FirePro S9150	5271,81
5	528	2464	Intel Xeon E5-2620v2	Nvidia K20x	4257,88
6	1300	13520	Intel Xeon E5-2680v2	Nvidia K80	4112,10
7	440	2640	Intel Xeon E5-2660v2	Nvidia K40m	3962,73
8	1536	3328	Intel Xeon E5-2630v2	Nvidia K20	3631,69
9	1536	1664	Intel Xeon E5-2680v3	Nvidia K80	3614,70
10	2050	2268	Intel Xeon E5-2680v2	Nvidia K20x	3543,31
11	1280	3584	Intel Xeon E5-2680v2	Nvidia K20x	3517,83
12	9780	732	Intel Xeon E5-2680v3	Intel Xeon Phi 7120P	3222,80
13	42176	73808	Intel Xeon E5-2670	Nvidia K20x	3185,90
14	2080	3640	Intel Xeon E5-2650v2	Nvidia K20x	3131,06
15	6912	8640	Intel Xeon E5-2680v3	Nvidia K40	3045,04
16	2000	4875	Intel Xeon E5-2650	Nvidia K20m	3019,72
17	16896	59136	Intel Xeon X5670	Nvidia K20x	2951,95
18	960	3120	Intel Xeon E5-2620v2	Nvidia K20	2877,92
19	1620	2268	Intel Xeon E5-2680v2	Nvidia K20x	2629,41
20	1620	2268	Intel Xeon E5-2680v2	Nvidia K20x	2629,41

3. MODELOWANIE BIZNESOWE SYSTEMU

3.1. Cele systemu

3.1.1. Biznesowe

Poniżej przedstawiono cztery główne oczekiwania względem systemu.

- Możliwość kontrolowania na bieżąco ilości energii zużywanej podczas wykonywania obliczeń oraz sprawdzania, ile energii zostało zużyte w danym okresie czasu.
- Ułatwienie analizowania opłacalności wykonywanych obliczeń i wydajności energetycznej systemu.
- Ograniczenie ilości energii zużywanej podczas obliczeń.
- Umożliwienie scentralizowanego zarządzania rozproszonym środowiskiem.

3.1.2. Funkcjonalne

Na podstawie celów biznesowych należy określić cele funkcjonalne, które w bardziej bezpośredni sposób przekładają się na projekt systemu, zostały one przestawione poniżej.

- Możliwość dodawania i usuwania węzłów obliczeniowych wchodzących w skład środowiska obliczeniowego.
- Ciągły nadzór nad węzłami obliczeniowymi wchodzącymi w skład środowiska obliczeniowego.
- Nałożenie warstwy abstrakcji na różnice w poszczególnych bibliotekach i udostępnienie użytkownikowi jednego, spójnego interfejsu.
- Umożliwienie określania reguł zarządzania energią tak, by zapewnić jak największą elastyczność w modelowaniu zużycia energii.
- Kontrolowanie zużycia energii w sposób automatyczny, zgodnie z określonymi przez użytkownika regułami.
- Zbieranie i przechowywanie statystyk zużycia energii.
- Prezentowanie użytkownikowi statystyk zużycia energii, aby mógł zweryfikować skuteczność zdefiniowanych przez siebie reguł i ilość energii zużytej w danym przedziale czasu.

3.2. Przewidywane komponenty programowe

W projektowanym systemie planowane jest wprowadzenie trzech komponentów programowych, każdy z nich będzie odpowiedzialny za inny aspekt jego funkcjonowania.

- Agent[7] – komponent który będzie umieszczony na węzłach obliczeniowych i będzie wchodził w interakcję z urządzeniami obliczeniowymi. Będzie zbierał dane i w odpowiednim formacie przekazywał je do serwera oraz wykonywał polecenia wysłane przez serwer.
- Serwer – będzie stanowił rdzeń systemu odpowiedzialny za przyjmowanie poleceń od interfejsu użytkownika, przechowywanie danych trwałych i obsługę logiki związanej z zarządzaniem energią. Będzie też wysyłał polecenia do agentów.
- Interfejs użytkownika – jego zadaniem będzie prezentowanie danych oraz przekazywanie akcji użytkownika do serwera.

3.3. Specyfikacja wymagań funkcjonalnych

Wymagania zostały pogrupowane w podpunkty ze względu na to, których aktorów dotyczą. Podpunkt 3.3.1. przedstawia wymagania opisujące zbiór funkcji, który powinien realizować agent aby mógł on w pełni wykonywać zadania znajdujące się w jego obszarze odpowiedzialności. Podobnie w przypadku wymagań stawianych wobec serwera w podpunkcie 3.3.2. Podpunkt 3.3.3. opisuje wymagania funkcjonalne, których bezpośrednim źródłem jest użytkownik systemu.

3.3.1. Agent

Agent jest odpowiedzialny za:

- wykrywanie dostępnych urządzeń i odczytywanie istotnych informacji o maszynie, na której jest uruchomiony;
- odczytywanie aktualnie obowiązującego dla danego urządzenia limitu zużycia energii;
- odczytywanie ilości energii zużywanej w danej chwili przez urządzenie;
- ustawianie limitu zużycia energii i możliwych do ustawienia wartości tego limitu na danym urządzeniu;
- komunikowanie się z serwerem.

[7] Należy w tym miejscu zaznaczyć, że przyjęta nazwa nie odnosi się do systemów agentowych, gdyż funkcjonalność tego komponentu spełnia jedynie część definicji agenta, w tym sensie, że jest on zdolny do postrzegania stanu środowiska, w którym się znajduje i wpływania na nie, ale nie będzie modyfikował swoich zachowań na podstawie nabytej wiedzy [Matuszek M., 2015]. Przy wyborze tej nazwy kierowano się nazewnictwem stosowanym dla komponentów programowych uruchamianych na monitorowanych maszynach w systemach/protokołach monitorujących rozproszone środowiska, przykładami których mogą być: SNMP, RMON, Ruxit czy New Relic APM.

3.3.2. Serwer

Funkcje, które musi realizować serwer, aby mógł wypełniać przypisane mu obowiązki, są następujące:

- przechowywanie i umożliwienie modyfikowania zawartości listy węzłów obliczeniowych;
- nadzorowanie stanu węzłów obliczeniowych;
- komunikowanie się z agentem i interfejsem użytkownika;
- udostępnianie gotowych szablonów dla reguł zużycia energii;
- przechowywanie utworzonych na podstawie szablonów reguł zarządzania zużyciem energii;
- kontrolowanie, czy każde z urządzeń podłączonych do systemu pracuje zgodnie ze zdefiniowanymi dla niego regułami zużycia energii;
- systematyczne gromadzenie danych dotyczących zużycia energii.

3.3.3. Użytkownik

System powinien umożliwiać:

- dodawanie i usuwania węzłów obliczeniowych;
- wyświetlanie informacji dotyczących węzłów dodanych do systemu i listy urządzeń dostępnych na każdym z nich wraz z informacjami o tych urządzeniach;
- wyświetlanie dostępnych szablonów reguł i tworzenie na tej podstawie ich konkretnych instancji przypisywanych do urządzeń oraz usuwanie niepotrzebnych reguł;
- wyświetlanie danych statystycznych dotyczących zużycia energii i limitów zużycia energii, które obowiązywały w danym czasie.

3.4. Specyfikacja wymagań pozafunkcjonalnych

Poniższe wymagania zamieszczone są w formacie: nazwa (priorytet) opis, gdzie 1 to priorytet najwyższy, a 4 najniższy. Priorytet oznacza istotność danego wymagania, a więc przekłada się na stopień, w jakim powinno być ono brane pod uwagę podczas projektowania i implementacji systemu.

Wydajność (2) – agent powinien w jak najmniejszym stopniu obciążać zasoby węzła obliczeniowego, na którym pracuje. Serwer ma mniejsze wymagania wydajnościowe, powinien być w stanie obsłużyć komunikację z wieloma agentami w tym samym czasie.

Niezawodność (2) – system powinien pozostawać dostępny i działać w sposób niezawodny, podczas gdy wykonywane są obliczenia. Wymaganie to jest szczególnie istotne, jeżeli chodzi o agentów, którzy mogą być umieszczeni na dużej liczbie węzłów obliczeniowych, przez co trudno będzie diagnozować i naprawiać ich awarie. Należy wziąć również pod uwagę niezawodne działanie serwera, gdyż jest on głównym węzłem komunikacyjnym. Wysoką niezawodnością musi też cechować się baza danych przechowująca wszystkie niezbędne informacje o konfiguracji systemu (lista węzłów obliczeniowych, statystyki, reguły).

Dostępność (3) – system powinien być dostępny na czas przeprowadzania obliczeń, gdyż wtedy potrzebne jest zarządzanie zużyciem energii. Tym samym wymagania co do dostępności nie są wygórowane i można założyć, że jeżeli zajdzie taka konieczność, system będzie mógł być serwisowany oraz nie trzeba podejmować specjalnych działań mających zapewnić jego nieprzerwane działanie, aczkolwiek byłoby to mile widziane.

Ochrona (3) – komunikacja pomiędzy poszczególnymi komponentami mogłaby być zabezpieczona, jednak nie jest to bardzo istotny element i może być traktowany jako potencjalne rozszerzenie funkcjonalności systemu. Przez sieć nie są przesyłane bardzo wrażliwe dane, a w wypadku ataku potencjalną konsekwencją będzie jedynie destabilizacja systemu. Potencjalnym celem ataku jest też agent, który posiadałby uprawnienia pozwalające na dostęp do wrażliwych ustawień urządzeń obliczeniowych (np. taktowania zegarów lub napięcia), jednak w systemie nie istnieje możliwość ingerowania w nie w bezpośredni sposób, a więc problem dotyczy raczej sfery teoretycznej niż praktycznej.

Bezpieczeństwo (4) – system nie kontroluje działania urządzeń, których awaria może zagrozić życiu lub zdrowiu użytkowników lub osób postronnych.

Przenośność (3) – agent z założenia ma wspierać tylko jedną platformę, gdyż środowiska wysokiej wydajności są w znakomitej większości oparte o system Linux. Sprawa ma się podobnie w sytuacji serwera, jednak w jego wypadku wieloplatformowość byłaby mile widziana.

Elastyczność (1) – system powinien uwzględniać możliwość łatwej rozbudowy o obsługę nowych typów urządzeń i rozszerzanie szablonów reguł zarządzania energią o nowe pozycje. Rozbudowa interfejsu komunikacyjnego wystawianego przez serwer również nie powinna sprawiać trudności i w jak największym stopniu uwzględniać jego przyszłą rozbudowę o nowe elementy.

Konfigurowalność (1) – możliwość dostosowania pracy systemu do potrzeb użytkownika powinna być traktowana w sposób priorytetowy. Zrealizować można to poprzez udostępnienie dużego zbioru szablonów reguł zarządzania energią oraz możliwie jak największe usprawnienie procesu dodawania nowych węzłów i definiowania nowych reguł na podstawie istniejących szablonów.

Interfejs użytkownika (1) – API (ang. *Application Programming Interface* - interfejs programistyczny aplikacji, czyli sposób w jaki systemy i podsystemy komunikują się ze sobą) służące do komunikacji z interfejsem powinno być uniwersalne, elastyczne i możliwie jak najprostsze, tak aby umożliwić komunikowanie się z systemem za pomocą różnych rozwiązań końcowych (np. przeglądarka, cURL[8]). Dostarczany interfejs użytkownika w postaci aplikacji działającej w środowisku przeglądarki internetowej powinien być prosty, czytelny i intuicyjny oraz pozwalać na pełne wykorzystanie możliwości oferowanych przez system.

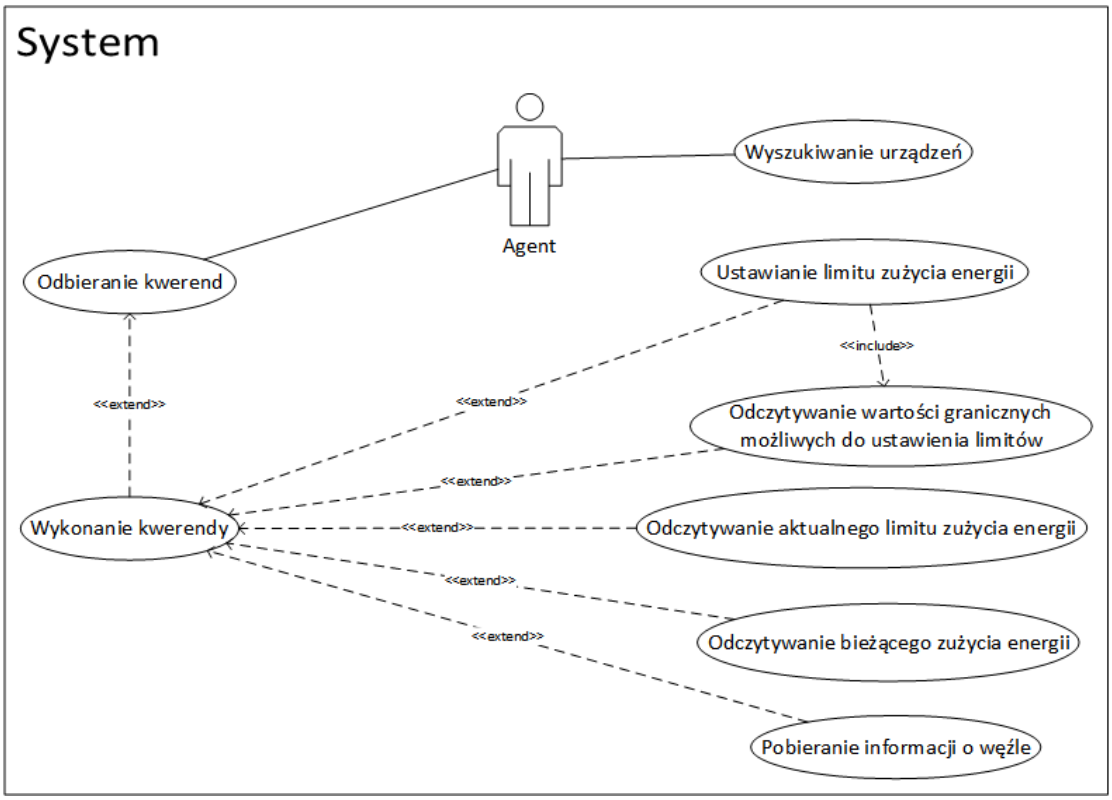
[8] cURL - sieciowa biblioteka programistyczna działająca po stronie klienta, najczęściej używana jako linuksowe polecenie *curl.*

3.5. Przypadki użycia

3.5.1. Opis aktorów

- Agent – aplikacja uruchomiona na węźle obliczeniowym.
- Serwer – aplikacja nadzorująca węzły obliczeniowe i komunikująca się ze znajdującymi się na nich agentami. Jej pracą sterują komunikaty otrzymywane od interfejsu użytkownika.
- Użytkownik – osoba korzystająca z aplikacji pracującej w środowisku przeglądarki internetowej (pełniącej rolę interfejsu użytkownika) lub moduł innego systemu, który komunikuje się z serwerem. W tej analizie odnosimy się do użytkownika jako do osoby, aby wymagania można było sformułować w prosty i spójny sposób. Gdybyśmy rozważali je pod kątem modułu innego systemu, będącego użytkownikiem, zmianie uległby jedynie sposób formułowania wymagań, ale nie ich zasadnicza treść.

3.5.2. Diagram przypadków użycia agenta

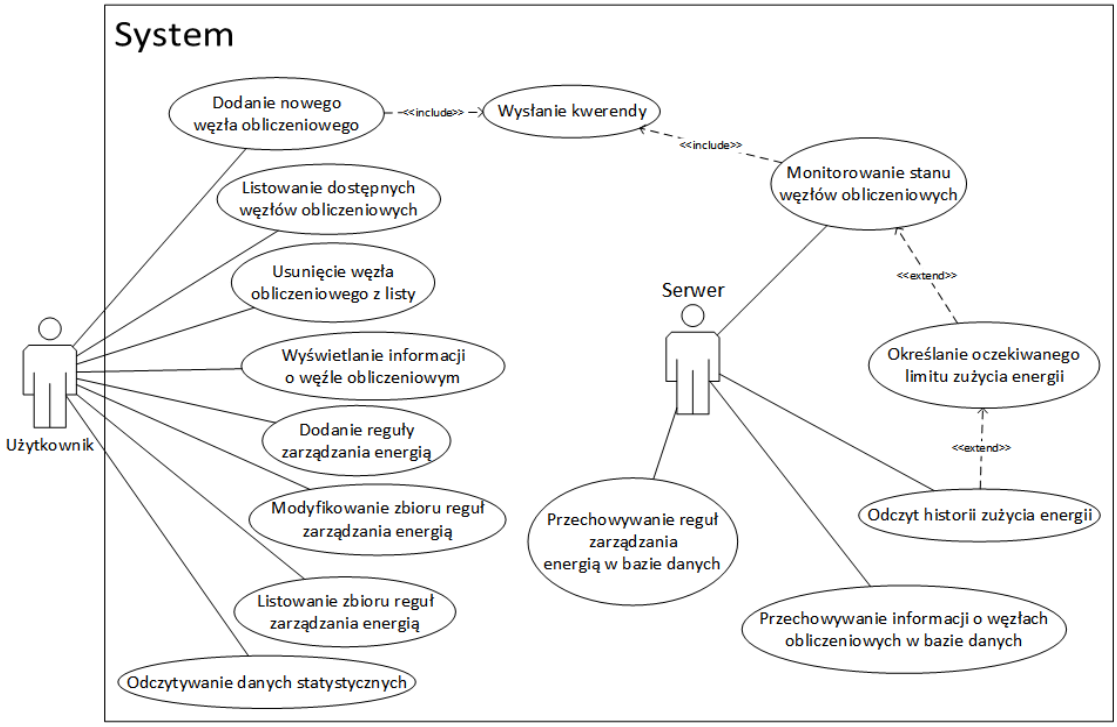


Rys. 3.1. Diagram przypadków użycia agenta

Poniżej znajduje się skrócony opis przypadków użycia zilustrowanych na rys. 3.1., opis szczegółowy został umieszczony w dodatku B.

- Wyszukiwanie urządzeń - w systemie operacyjnym wyszukiwane są wspierane przez agenta urządzenia obliczeniowe.
- Ustawianie limitu zużycia energii - na wybranym urządzeniu ustawiany jest nowy limit zużycia energii.
- Odczytywanie wartości granicznych możliwych do ustawienia limitów - dla danego urządzenia odczytywany jest minimalny oraz maksymalny możliwy do ustawienia limit zużycia energii.
- Odczytywanie aktualnego limitu zużycia energii – z urządzenia odczytywany jest aktualnie ustawiony dla niego limit zużycia energii.
- Odczytywanie bieżącego zużycia energii – odczytywane jest bieżące zużycie energii dla zadanego urządzenia.
- Pobieranie informacji o węźle – pobierane są informacje o węźle, między innymi: nazwa maszyny, wersja systemu, dostępne urządzenia obliczeniowe.
- Wykonanie kwerendy – otrzymana od serwera kwerenda jest parsowana i, o ile jest poprawna, zostaje wykonana.
- Odbieranie kwerend – agent prowadzi czynny nasłuch w oczekiwaniu na kwerendy, w przypadku odebrania kwerendy o rozpoznanym typie zostaje ona wykonana, a rezultat wykonania zostaje zwrócony zlecającemu.

3.5.3. Diagram przypadków użycia serwera



Rys. 3.2. Diagram przypadków użycia serwera

Poniżej znajduję się skrócony opis przypadków użycia zilustrowanych na rys. 3.2., opis szczegółowy został umieszczony w dodatku B.

- Wysyłanie kwerendy – serwer wysyła kwerendę do agenta na wybranym węźle obliczeniowym.
- Monitorowanie stanu węzłów obliczeniowych – wszystkie zarejestrowane węzły obliczeniowe są okresowo odpytywane o limit zużycia energii i chwilowe zużycie energii. W razie konieczności wysyłana jest korekta limitu zużycia energii.
- Określanie oczekiwanego limitu zużycia energii – określany jest limit zużycia energii, który powinien być ustawiony na określonym urządzeniu na określonym węźle.
- Odczyt historii zużycia energii – historia zużycia energii dotycząca określonego urządzenia dla danego przedziału czasowego jest odczytywana z bazy danych.
- Przechowywanie informacji o węzłach obliczeniowych w bazie danych – informacja o węzłach obliczeniowych jest ładowana z bazy danych do pamięci lub informacja znajdująca się w pamięci jest zapisywana do bazy danych.
- Przechowywanie reguł zarządzania energią w bazie danych – informacja o regułach zarządzania energią jest ładowana z bazy danych do pamięci lub informacja znajdująca się w pamięci jest zapisywana do bazy danych.
- Dodanie nowego węzła obliczeniowego – do listy węzłów obliczeniowych dodawany jest nowy węzeł, o ile istnieje i jest na nim uruchomiony agent.
- Listowanie dostępnych węzłów obliczeniowych – lista dostępnych węzłów obliczeniowych zostaje wyświetlona użytkownikowi.
- Usunięcie węzła obliczeniowego z listy – węzeł obliczeniowy jest usuwany z listy węzłów.
- Wyświetlanie informacji o węźle – użytkownik z listy węzłów wybiera jeden węzeł, dla którego wyświetlone zostaje okno ze szczegółowymi informacjami, między innymi z listą dostępnych urządzeń.
- Dodanie reguły zarządzania energią – użytkownik dodaje nową regułę zarządzania energią.
- Modyfikowanie zbioru reguł zarządzania energią – użytkownik modyfikuje jedną lub wiele reguł zarządzania energią. Modyfikacją może być usunięcie, zmiana kolejności lub zmiana parametrów danej reguły.
- Listowanie zbioru reguł zarządzania energią – zbiór reguł zarządzania energią zostaje wyświetlony użytkownikowi.
- Odczytywanie danych statystycznych – dane dla danego okresu i urządzenia o tym, jak zmieniało się chwilowe zużycie energii i jak zmieniany był limit zarządzania energią są odczytywane z bazy danych i wyświetlane użytkownikowi.

4. ARCHITEKTURA SYSTEMU

4.1. Podział systemu na komponenty

System podzielony jest na trzy komponenty, realizujące rozłączne podzbiory funkcjonalności całego systemu. Taka dekompozycja pozwoliła w jasny sposób określić zakres odpowiedzialności każdego z komponentów, ułatwiła implementację (jasny podział pracy pomiędzy członków zespołu) oraz znacznie ułatwi wprowadzanie modyfikacji, jeżeli zajdzie taka potrzeba. Przykładowo: wprowadzenie redundancji w przypadku serwera, tak by w razie awarii jednego z nich system dalej funkcjonował lub wymiana komponentu będącego interfejsem użytkownika (np. na moduł innego systemu wchodzący w bezpośrednią interakcję z serwerem). Rys. 4.1. przestawia wysokopoziomową architekturę systemu.

Agent – uruchamiany jest na węzłach obliczeniowych, odpowiedzialny jest za interakcję z bibliotekami służącymi do zarządzania zużyciem energii na poszczególnych urządzeniach obliczeniowych. Nasłuchuje na żądania wysyłane przez serwer.

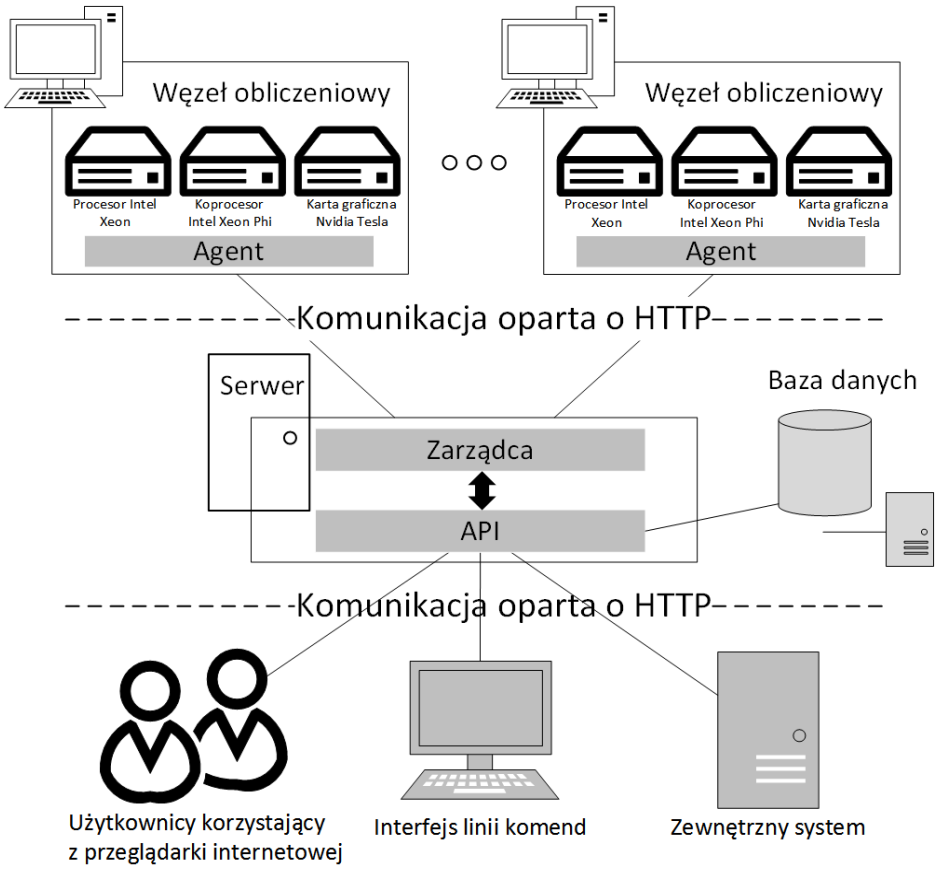
Serwer – rdzeń systemu, uruchamiany na wybranej maszynie, która może, ale nie musi być węzłem obliczeniowym. Odpowiedzialny jest za monitorowanie stanu węzłów obliczeniowych, przechowywanie zbioru reguł zarządzania energią i gromadzenie statystyk. Zapisuje i odczytuje informacje z bazy danych, nasłuchuje na żądania wysyłane przez interfejs użytkownika.

Interfejs użytkownika – odpowiedzialny za prezentowanie i umożliwienie modyfikacji elementów takich jak lista dostępnych węzłów obliczeniowych czy lista reguł zarządzania energią oraz wyświetlanie statystyk.

Komunikacja pomiędzy poszczególnymi komponentami odbywa się za pomocą protokołu HTTP (ang. *HyperText Transport Protocol* – protokół przesyłania dokumentów hipertekstowych) z wykorzystaniem architektury REST[9]. Wybór ten motywowany był kilkoma czynnikami. Pierwszym z nich jest prostota implementacji – wystarczy po stronie odbiorczej udostępnić serwer HTTP nasłuchujący na żądania, a po stronie nadawczej klienta, który żądania te będzie wysyłał. Kolejną zaletą jest popularność tego rozwiązania, dzięki czemu żadnego problemu nie stanowiło skomunikowanie ze sobą bardzo odmiennych technologii: JavaScript po stronie interfejsu użytkownika, Python po stronie serwera i C++ po stronie agenta. Dla komunikacji pomiędzy serwerem, a interfejsem użytkownika istotną kwestią było umożliwienie bezproblemowej wymiany elementu pełniącego rolę interfejsu użytkownika. W proponowanym przez nas rozwiązaniu jego rolę pełni aplikacja działająca w środowisku przeglądarki internetowej, ale z powodzeniem może ona zostać zastąpiona przez aplikację konsolową (np. cURL wykorzystywany w ramach skryptu) lub komponent innego systemu odpowiedzialny za komunikację.

[9] REST (ang. *Representational State Transfer* - zmiana stanu poprzez reprezentacje) - styl architektury oprogramowania zakładający m.in. bezstanową komunikację, najczęściej implementowany przy użyciu HTTP.

Wybrana metoda wprowadza też pewne ograniczenia, które szczęśliwie nie mają przełożenia na istotne problemy. Niedogodność może stanowić przesyłanie zarówno zapytań, jak i odpowiedzi tekstem jawnym, co nie jest optymalną formą wymiany komunikatów, jeżeli wziąć pod uwagę obciążenie zasobów sieciowych. Jednak w przypadku stosunkowo małej intensywności komunikacji w systemie, wprowadzanie mechanizmu serializacji danych byłoby niepotrzebną komplikacją i nie wydaje się konieczne. Drugą z rzeczy, które należało mieć na uwadze, było zastosowanie odpowiednich kodów odpowiedzi HTTP, co wymuszało poświęcenie większej uwagi temu zagadnieniu. Problem ten byłby nieobecny w przypadku wykorzystywania własnego standardu wymiany wiadomości. Wreszcie, odpowiedzi przesyłane są z wykorzystaniem formatu JSON (ang. *JavaScript Object Notation* – notacja obiektu JavaScript), co również wymaga dostosowania przesyłanych danych do jego struktury, jednak jego prostota i elastyczność pozwoliła na wykonanie tego bez większych problemów. Dzięki małemu, dodatkowemu nakładowi pracy po stronie nadawczej zaoszczędzono większego nakładu pracy po stronie odbiorczej. Ostatnie z wymagań dotyczy modelu komunikacji, który aby być zgodnym z architekturą REST, powinien odbywać się na zasadzie „żądanie – odpowiedź". Przy projekcie aplikacji z góry założono, że komunikacja będzie odbywać się w sposób synchroniczny i zgodnie z przytoczoną zasadą, więc siłą rzeczy nie stanowiło to żadnego problemu.



Rys. 4.1. Wizualizacja podziału systemu na komponenty

4.2. Zapytania REST obsługiwane przez serwer

Każde z zapytań może zawierać jedynie wskazane w tabeli 4.1. parametry. Odpowiedzią serwera na zapytania skierowane do wymienionych zasobów jest kod statusu i JSON bądź kod błędu odpowiadający kodowi błędu HTTP.

Możliwe kody statusu i błędów:

- 200 – powodzenie operacji GET/DELETE;
- 201 – powodzenie operacji PUT;
- 404 – nie znaleziono obiektu zapytania (węzła obliczeniowego lub urządzenia);
- 406 – niepowodzenie operacji.

Tabela 4.1. Składnia zapytań serwera

Metoda	Zasób	Składnia pojedynczego parametru
GET	/status	Brak
GET	/nodes/computation_nodes	Brak
GET	/nodes/computation_node/{nazwa}	{nazwa} – nazwa węzła
GET	/nodes/computation_node/{nazwa}/{ident}/power_limit	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
GET	/nodes/computation_node/{nazwa}/{ident}/statistics_interval	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
GET	/nodes/computation_node/{nazwa}/{ident}/statistics_data/{data}	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia {data} – data i godzina
GET	/rule_types	Brak
GET	/nodes/computation_node/{nazwa}/{ident}/rule	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
GET	/nodes/computation_node/{nazwa}/{ident}/statistics_data	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
PUT	/nodes/computation_node/{nazwa}	{nazwa} – nazwa węzła
PUT	/nodes/computation_node/{nazwa}/{ident}/power_limit	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
PUT	/nodes/computation_node/{nazwa}/{ident}/statistics_interval	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
PUT	/nodes/computation_node/{nazwa}/{ident}/statistics_data/{data}	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia {data} – data i godzina
PUT	/nodes/computation_node/{nazwa}/{ident}/rule	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
DELETE	/nodes/computation_node/{nazwa}	{nazwa} – nazwa węzła
DELETE	/nodes/computation_node/{nazwa}/{ident}/power_limit	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
DELETE	/nodes/computation_node/{nazwa}/{ident}/statistics_interval	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia
DELETE	/nodes/computation_node/{nazwa}/{ident}/statistics_data/{data}	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia {data} – data i godzina
DELETE	/nodes/computation_node/{nazwa}/{ident}/rule	{nazwa} – nazwa węzła {ident} – identyfikator urządzenia

4.3. Zapytania REST obsługiwane przez agenta

Tabela 4.2. Składnia zapytań agenta

Metoda	Zasób	Składnia pojedynczego parametru
GET	/power_usage	{typ},{identyfikator}
GET	/power_limit	{typ},{identyfikator}
GET	/power_limit_percentage	{typ},{identyfikator}
GET	/power_limit_constraints	{typ},{identyfikator}
GET	/node_information	Brak
PUT	/power_limit	{typ},{identyfikator}={wartość}
PUT	/power_limit_percentage	{typ},{identyfikator}={wartość}
DELETE	/power_limit	{typ},{identyfikator}

Gdzie:

- {typ} – typ urządzenia, może przyjmować wartości: IntelXeon, IntelXeonPhi lub NvidiaTesla;
- {identyfikator} – identyfikator urządzenia, musi być zgodny z formatem identyfikatora dla danego typu urządzenia;
- {wartość} – wartość parametru do ustawienia.

Dla każdego z wymienionych zasobów, za wyjątkiem */node_information*, część URI (ang. *Uniform Resource Identifier* – Ujednolicony Identyfikator Zasobów), będąca zapytaniem, może zawierać wiele następujących po sobie parametrów. Odpowiedzią na każde z zapytań jest obiekt JSON zawierający żądane dane lub komunikat o błędzie, jeżeli takowy wystąpił, podczas odpytywania konkretnego urządzenia o zadaną wartość. Zwracana odpowiedź jest obiektem zawierającym obiekty związane z konkretnymi urządzeniami, przykład znajduje się na listingu 4.1.

Zwracane kody odpowiedzi HTTP:

- 200 – zapytanie zostało poprawnie obsłużone, nie oznacza to jednak, że przy odpytywaniu któregoś z urządzeń nie wystąpił błąd. Informację o powodzeniu wykonania zawiera wynikowy obiekt będący rezultatem przypisanym do konkretnego urządzenia;
- 400 – nie powiodło się walidowanie składni zapytania;
- 404 – wysłano zapytanie do nieistniejącego zasobu;
- 405 – zasób, do którego wysłano zapytanie, istnieje, ale nie obsługuje wybranej metody;
- 500 – wewnętrzny błąd agenta, treść odpowiedzi może, ale nie musi zawierać opisu zaistniałego błędu.

Patrząc na tabelę 4.2. można zauważyć dość rażące odstępstwo od architektury REST, a mianowicie sposób odwoływania się do konkretnych urządzeń, zilustrujmy to na przykładzie:

GET http://agent/power_limit?IntelXeon,1&NvidiaTesla,1

Zamiast:

GET http://agent/power_limit/IntelXeon/1

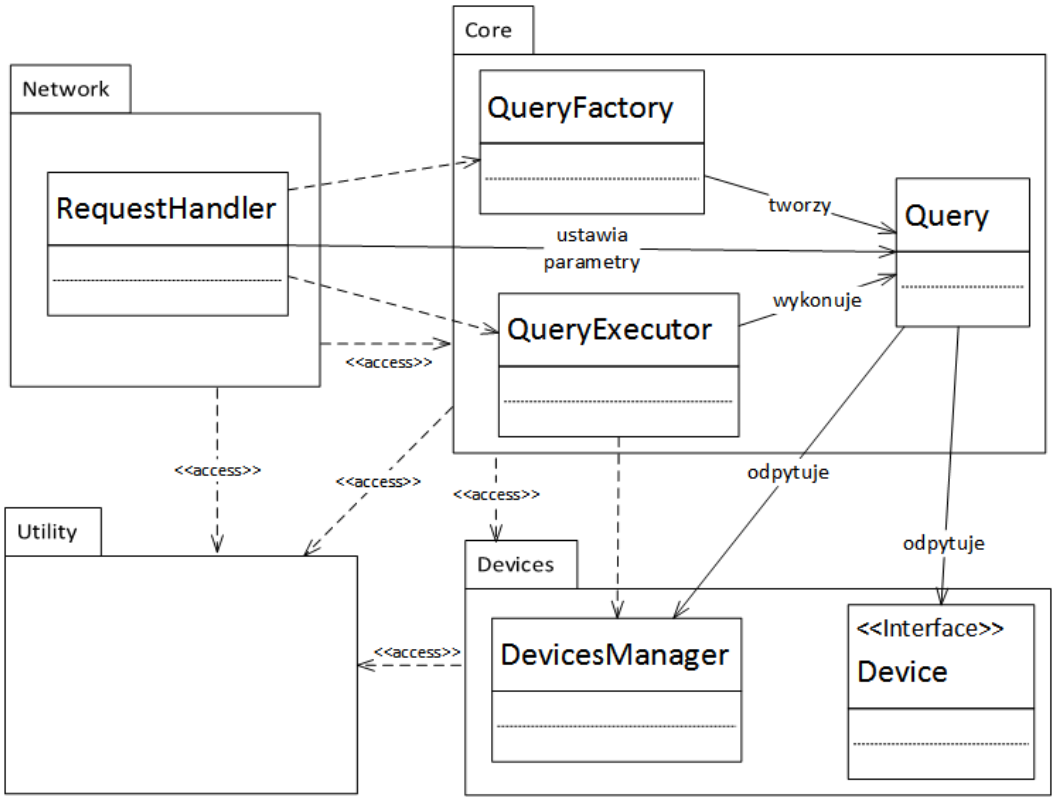
GET http://agent/power_limit/NvidiaTesla/1

Wyboru tego dokonano świadomie po wzięciu pod uwagę, w jaki sposób serwer będzie wysyłał komunikaty do agenta. Gdyby zawsze odpytywał o pojedyncze lub wszystkie urządzenia, można by pozostać w pełni zgodnym z architekturą REST, implementując odwołanie do wszystkich dostępnych urządzeń jako do kolekcji. Jednak serwer w naszym wypadku może wysyłać zapytania o kilka urządzeń różnego typu, wybranych z całej listy, a w takiej sytuacji trzeba by wysyłać wiele pojedynczych zapytań, jak zobrazowano powyżej. Po wzięciu pod uwagę faktu, że komunikacja ta odbywa się wewnątrz systemu, decyzja o dostosowaniu wykorzystywanej architektury do własnych potrzeb wydaje się jak najbardziej zasadna.

Listing 4.1. Rezultat wywołania polecenia odczytującego aktualne zużycie energii dla dwóch urządzeń

```
[
    {
        "PowerUsage": 26,
        "Type": "NvidiaTesla",
        "id": "GPU-7cf39d4a-359b-5922-79a9-506fddfc61e3"
    },
    {
        "PowerUsage": 25,
        "Type": "NvidiaTesla",
        "id": "GPU-041fac5f-eb49-3563-cecd-a3e4df89f1a8"
    }
]
```

4.4. Architektura agenta



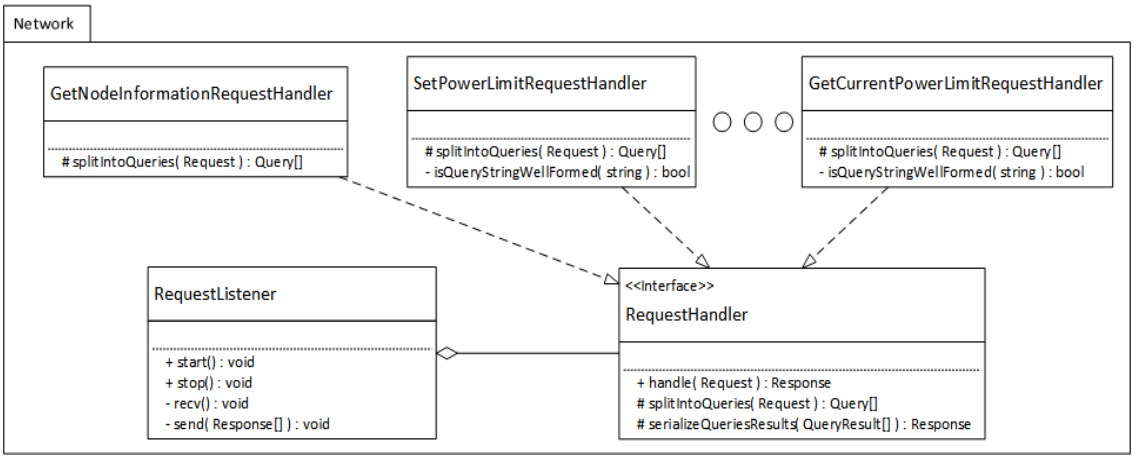
Rys. 4.2. Diagram modułów uwzględniający istotniejsze klasy

Bazując na rys. 4.2. poddajmy pod analizę architekturę agenta. Można w niej wydzielić trzy główne moduły z jasno zdefiniowanymi: punktami styku, zakresem odpowiedzialności i sposobem, w jaki odbywa się wzajemna komunikacja. Występuje również jeden moduł pomocniczy.

Moduł Network odbiera, waliduje i rozbija na elementy składowe (*Query*) otrzymane z sieci żądania, formuje i wysyła odpowiedzi, definiuje obsługiwane typy żądań i ich składnię, tworzy *Query* i przekazuje je do wykonania do modułu *Core*. Moduł *Core* odbiera *Query* wytworzone w module Network, definiuje obsługiwane typy *Query*, definiuje format i sposób serializacji dla rezultatu każdego z *Query*, wywołuje funkcje z modułu Device i przetwarza rezultaty tych wywołań. Moduł *Devices* wchodzi w interakcję z API służącymi do zarządzania poszczególnymi urządzeniami, ładuje i inicjalizuje niezbędne biblioteki, nadzoruje stan urządzeń, zapewnia jednolity interfejs dostępowy do wszystkich urządzeń dla modułu *Core*. Moduł *Utility* zawiera klasy i funkcje użyteczne w obrębie całej aplikacji, komunikują się z nim wszystkie trzy główne moduły.

Teraz przejdźmy do bardziej szczegółowego opisu poszczególnych modułów, których wewnętrzna architektura została zobrazowana na rysunkach: 4.3, 4.4, 4.5 i 4.6.

4.4.1. Opis modułów



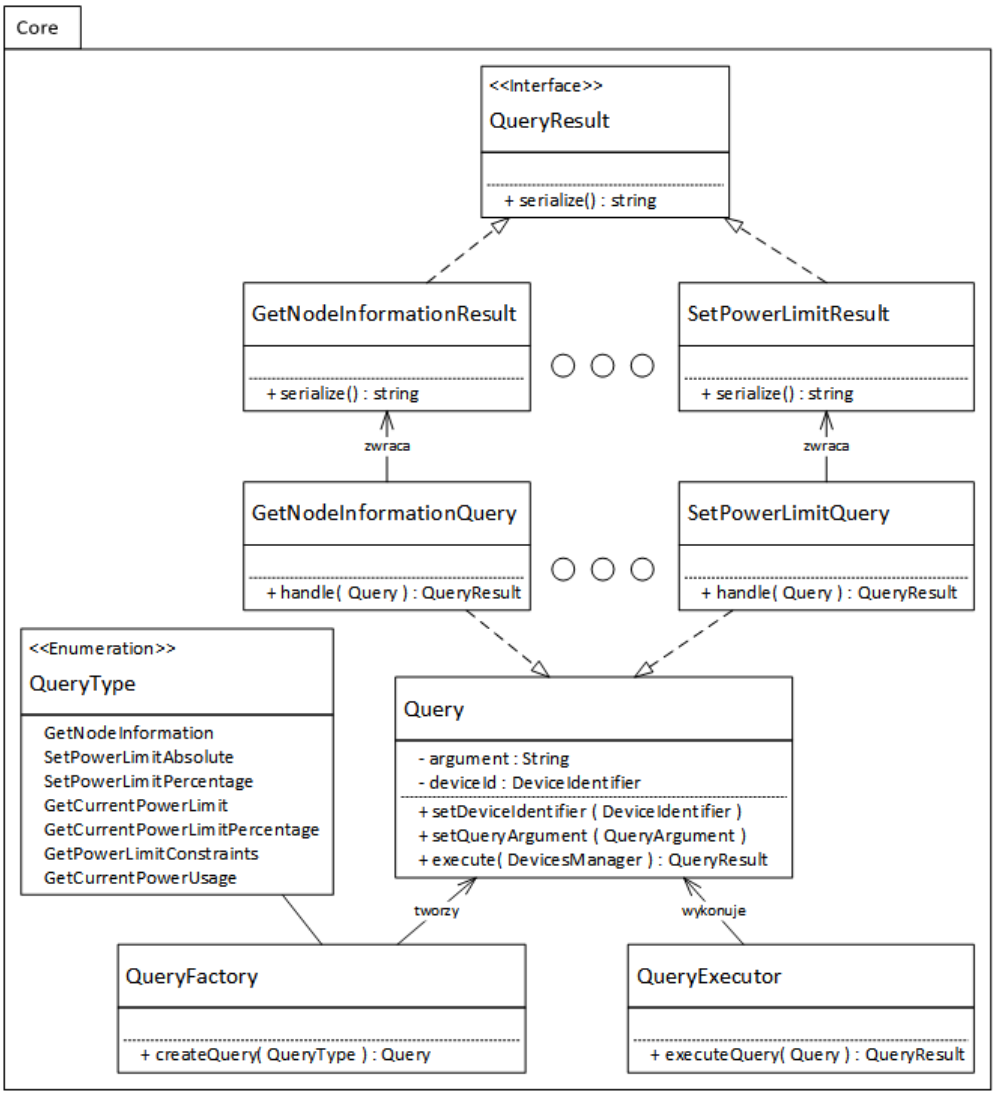
Rys. 4.3. Diagram klas modułu *Network*

Na samym początku zostanie omówiony moduł *Network*. *RequestListener* może być uznany za swojego rodzaju punkt wejściowy do aplikacji, to przez niego przechodzi cała komunikacja przychodząca z sieci i wychodząca do niej. W tej klasie obsługiwany jest serwer WWW udostępniany przez bibliotekę CPPREST, o której więcej informacji znajduje się w podrozdziale 5.1.4. Również w niej, z wykorzystaniem wzorca projektowego Strategia, rejestrowane są klasy implementujące interfejs *RequestHandler*. Klasy te odpowiadają za obsługę konkretnych typów żądań. Metoda *handle* interfejsu *RequestHandler* wykorzystuje wzorzec Metoda Szablonowa ze względu na to, że ogólny przebieg algorytmu obsługi jest niezależny od typu żądania. Metody *splitIntoQueries* i *serializeQueriesResults* są elementami algorytmu zależnymi od typu, a więc zadeklarowano je jako metody czysto wirtualne, które implementowane są w każdej z klas odpowiedzialnej za obsługę konkretnych żądań. Na listingu

4.2 można zauważyć, że *Query*, odpowiadające typowi zapytania, są formowane w funkcji *splitIntoQueries*, a następnie kolejno przekazywane do modułu *Core* w celu ich wykonania.

Listing 4.2. Pseudokod algorytmu obsługi żądania

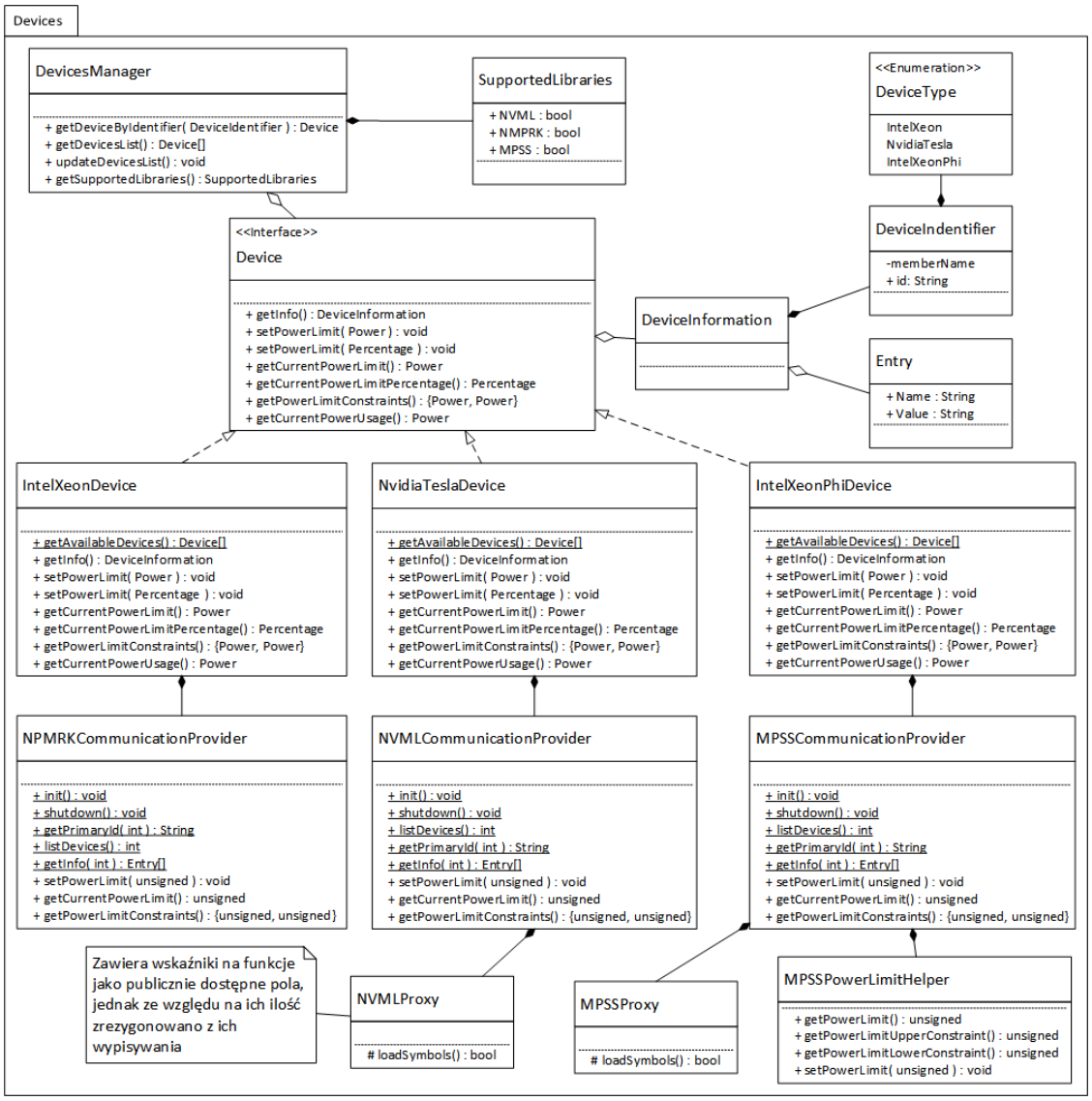
```
def obsłuż( Żądanie ) : Odpowiedź
   try
      Wynik[] wyniki
      zapytania = splitIntoQueries( Żądanie )
      for each zapytanie in zapytania
         wyniki.dodaj( queryExecutor.execute( zapytanie ) )
      odpowiedź = serializeQueriesResults( wyniki )
      odpowiedź.ustawKodHTTP( 200 )
      return odpowiedź
   catch
      odpowiedź.ustawKodHTTP ( wyjątek.kod() )
```



Rys. 4.4. Diagram klas modułu *Core*

Kolejnym z modułów jest *Core*. Metoda *createQuery* z klasy *QueryFactory* jest realizacją wzorca Metoda Fabrykująca i służy do tworzenia (w zależności od zadanego typu) nowych obiektów *Query* (realizujących wzorzec Polecenie). Tak utworzone obiekty są następnie przekazywane do klasy *QueryExecutor* w celu wykonania. Poszczególne typy implementujące

interfejs *Query* odpowiedzialne są za komunikację z modułem *Devices*, a dokładniej z jego klasami *DevicesManager* i klasami implementującymi interfejs *Device*. Do każdego z tych obiektów przypisany jest specyficzny dla niego rezultat wykonania, czyli klasa implementująca interfejs *QueryResult*, która posiada metodę pozwalająca na serializację jej zawartości do łańcucha znaków. Obiekt *QueryResult* jest zwracany jako rezultat działania metody *execute*, poprzez *QueryExecutora* do modułu *Network*. Dokładny przebieg obsługi zapytania został zobrazowany na rys 4.7.

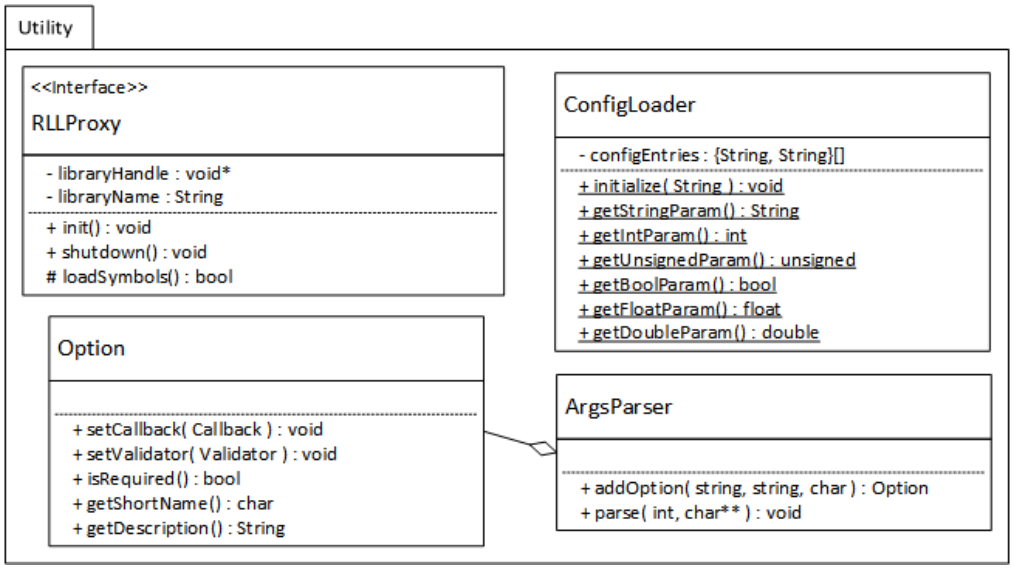


Rys. 4.5. Diagram klas modułu *Devices*

Trzecim modułem jest *Devices*. *DevicesManager* odpowiedzialny jest za pozyskanie i przechowywanie listy urządzeń wykrytych na danej maszynie. Poszczególne typy urządzeń są klasami implementującymi interfejs *Device*, który ujednolica odwołania i separuje wywołującego od detali implementacji każdej z bibliotek. W skład każdej klasy implementującej interfejs *Device* wchodzi klasa *DeviceInformation,* zawierająca informacje o urządzeniu, na które składają się: *DeviceIdentifier*, czyli typ urządzenia wraz z jego unikalnym identyfikatorem (identyfikator musi być unikalny w obrębie typu urządzeń, ale nie musi być unikalny w obrębie całego systemu)

oraz lista parametrów urządzenia w postaci „klucz – wartość” (parametrem może być przykładowo taktowanie zegara czy nazwa urządzenia). Unikalny identyfikator urządzenia jest też jednym z jego parametrów, jako identyfikator dla karty Nvidia i koprocesorów Intel Xeon Phi zostały wybrane ich UUID (ang. *Universally Unique Identifier* – uniwersalnie unikalny identyfikator), natomiast w przypadku procesorów Xeon identyfikatorem jest liczba heksadecymalna.

Na najgłębszym szczeblu hierarchii wywołań znajdują się klasy *CommunicationProvider*, które nie implementują co prawda wspólnego interfejsu, mimo że mogą stwarzać takie wrażenie, ponieważ nie dawało to żadnych realnych korzyści i było problematyczne ze względu na różniący się sposób identyfikowania urządzeń przez poszczególne biblioteki. Klasy te mogą wydawać się niepotrzebne – komunikacja z bibliotekami mogłaby odbywać się bezpośrednio w klasach implementujących interfejs *Device*, ale zdecydowano się na zastosowanie w tym miejscu wzorca projektowego Most, żeby wyraźnie oddzielić abstrakcję w formie jednolitego interfejsu dla każdego urządzenia od implementacji w postaci kodu wchodzącego w interakcję z wywołaniami bibliotecznymi. Dzięki temu w klasach *Device* można było zawrzeć wysokopoziomową logikę, taką jak weryfikowanie poprawności i wstępne przetwarzanie danych wejściowych, a w klasach *CommunicationProvider* obsłużyć, czasem dość kłopotliwe i mało eleganckie, wywołania biblioteczne.



Rys. 4.6. Diagram klas modułu *Utility*

Ostatnim i zarazem pomocniczym modułem jest *Utility*. Klasa *RLLProxy* służy do obsługi logiki związanej z ładowaniem w trakcie działania programu bibliotek dynamicznych. Zagadnienie to zostanie poruszone w dalszej części rozdziału. Klasa *ConfigLoader* jest Singletonem do którego można się odwołać z dowolnego miejsca aplikacji w celu pobrania parametrów konfiguracyjnych, po wcześniejszym wczytaniu przez niego pliku zawierającego te parametry. Klasa *ArgsParser* wraz z pomocniczą klasą *Option* pozwala na wygodne zdefiniowanie parametrów wywołania programu i przetworzenie linii komend w ich poszukiwaniu.

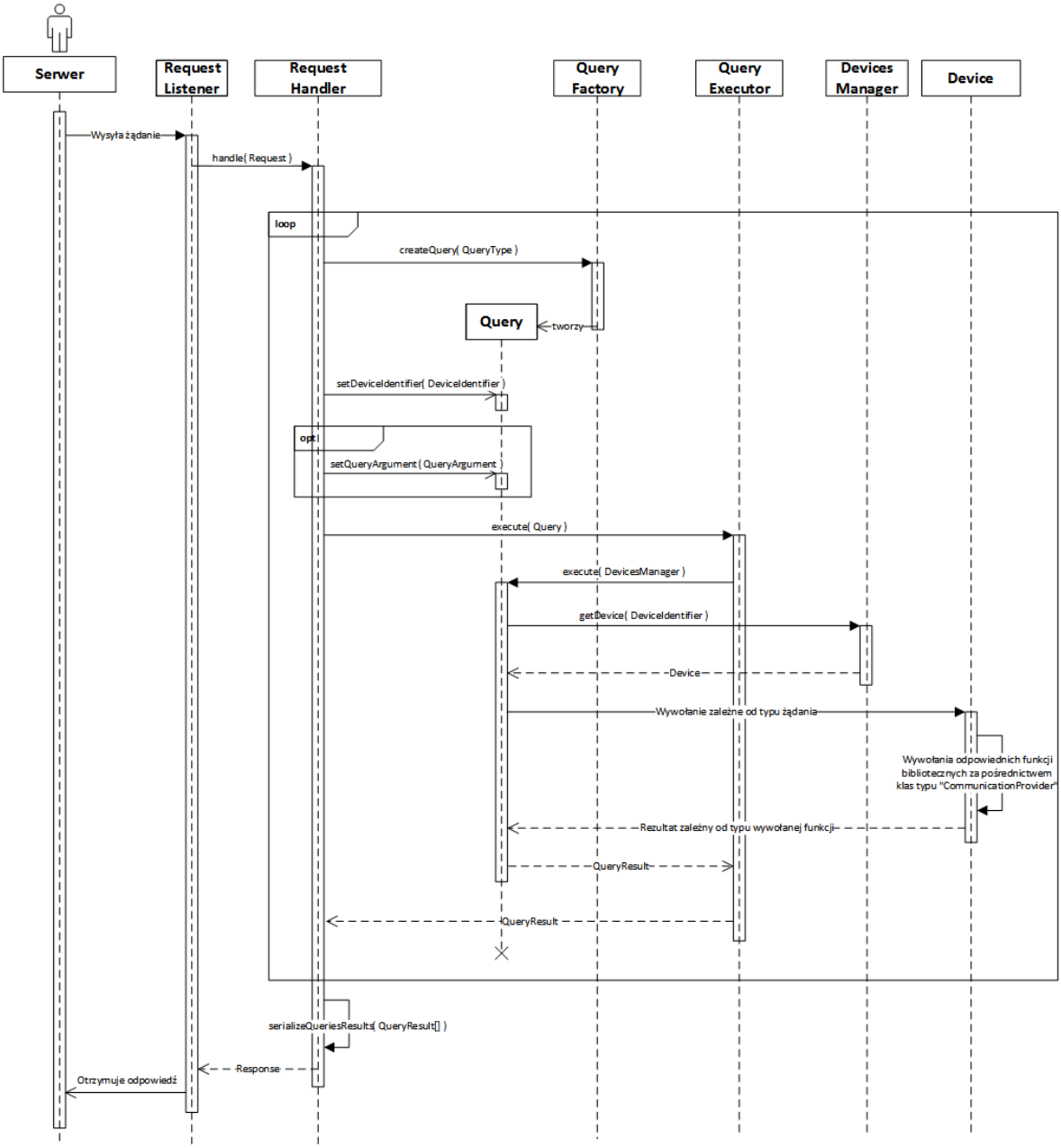
Wymieńmy jeszcze raz wszystkie wzorce projektowe, które zostały wykorzystane przy projektowaniu architektury agenta [Freeman E. i inni, 2011]:

- Strategia – przypisanie właściwych typów klas obsługujących zapytanie do typów zapytań;
- Metoda Szablonowa – algorytm obsługi zapytań, którego elementy są zależne od typu zapytania;
- Pełnomocnik – ładowanie i pośredniczenie w dostępie do bibliotek dynamicznych;
- Singleton – globalny dostęp do wartości zapisanych w pliku konfiguracyjnym;
- Most – oddzielenie interfejsu urządzenia od jego implementacji komunikującej się z biblioteką służącą do zarządzania zużyciem energii;
- RAII (ang. *Resource Acquisition Is Initialization* – inicjowanie przy pozyskaniu zasobu) – zachowanie pełnej kontroli nad zwalnianiem zasobów nawet w sytuacji, gdy zostanie rzucony wyjątek;
- Metoda fabrykująca – tworzenie obiektów poleceń;
- Polecenie – ukrycie detali związanych z wykonaniem konkretnego zapytania.

4.4.2. Proces obsługi żądań

Diagram sekwencji widoczny na rys. 4.7. przedstawia obsługę żądania skierowanego bezpośrednio do urządzenia (wszystkie za wyjątkiem *GetNodeInformation*). Zakładamy, że podczas obsługi żądania nie wystąpiły błędy. Prześledźmy jego przebieg.

1. Serwer wysyła żądanie.
2. *RequestListener* nasłuchujący na żądania odbiera je i przekazuje odpowiedniemu Handlerowi do wykonania.
3. Handler weryfikuje poprawność żądania i dzieli je na zapytania do poszczególnych urządzeń, które są tworzone przez klasę *QueryFactory*.
4. Następnie ustawiany jest identyfikator pozwalający na identyfikację urządzenia docelowego i opcjonalnie dla zapytań PUT argument.
5. Stworzone zapytanie jest przekazywane do klasy *QueryExecutor*, która inicjuje jego wykonanie.
6. Obiekt zapytania odwołuje się do klasy *DevicesManager* w celu pobrania urządzenia zgodnego z ustawionym wcześniej identyfikatorem.
7. Wywoływana jest metoda klasy *Device* zależna od typu zapytania.
8. Konkretny obiekt klasy implementującej interfejs *Device* wchodzi w interakcję z biblioteką zarządzającą danym urządzeniem.
9. Rezultat jest propagowany aż dotrze do obiektu klasy implementującej interfejs *RequestHandler*.
10. Po wykonaniu wszystkich zapytań ich rezultaty są serializowane do obiektu JSON.
11. Kod HTTP odpowiedzi jest ustawiany na 200 i zostaje ona wysłana do serwera.



Rys. 4.7. Diagram sekwencji obrazujący przebieg procesu obsługi żądania

4.4.3. Dynamiczne ładowanie bibliotek

Istotną kwestią, która musiała zostać uwzględniona na etapie projektowania rozwiązania, była potencjalna różnorodność maszyn, na których będzie uruchamiany agent. Konkretniej rzecz ujmując, chodzi o występowanie na nich bibliotek służących do zarządzania urządzeniami obliczeniowymi. O ile w wypadku NMPRK problem ten nie występuje, bo jest ona linkowana statycznie, to już w wypadku NVML i MPSS (ang. *Manycore Platform Software Stack* – biblioteka zarządcza dla urządzeń Intel Xeon Phi) kwestia ta nabiera na znaczeniu, gdyż biblioteki te dołączane są dynamicznie. Rozważmy sytuację, w której na danym węźle nie jest zainstalowana któraś z dwóch wcześniej wymienionych bibliotek. Odpalenie aplikacji nie powiedzie się, gdyż próba załadowania biblioteki dynamicznej zostanie podjęta niezależnie od tego, czy biblioteka ta jest w systemie, czy nie. Rozwiązaniem może więc wydawać się

dostarczanie bibliotek razem z plikami aplikacji, jednak niosłoby to za sobą dwa problemy: zbędny rozrost i zamieszanie wprowadzane w paczce dystrybucyjnej (co jeszcze dałoby się zaakceptować) oraz drugi, nie do zaakceptowania, możliwa niezgodność wersji dostarczanej biblioteki z modułami jądra systemu odpowiedzialnymi za obsługę komunikacji z poszczególnymi urządzeniami. Problem ten jest na tyle istotny, że wymusił znalezienie rozwiązania alternatywnego.

Zdecydowano się więc na zrezygnowanie z łączenia do bibliotek dynamicznych na etapie kompilowania aplikacji i wprowadzenia klasy pomocniczej, która będzie służyła do wczytywania ich już po jej odpaleniu, na żądanie i mając wtedy pewność, że biblioteka ta jest obecna w systemie plików. Do tego celu służy klasa abstrakcyjna *RLLProxy,* realizująca wzorzec Pełnomocnika. Jej konkretne implementacje (*NVMLProxy* i *MPSSProxy)* są odpowiedzialne za udostępnienie wskaźników na funkcje w załadowanej dynamicznie bibliotece i ustawienie ich wartości. Wszystkie odwołania do funkcji bibliotecznych są potem realizowane poprzez te wskaźniki, żaden z obiektów *CommunicationProvider* nie wywołuje funkcji bibliotecznych bezpośrednio.

4.4.4. Opis algorytmu wykorzystanego dla Intel Xeon Phi

MPSS, w przeciwieństwie do NVML, nie daje możliwości zdefiniowania twardego limitu dla maksymalnego zużycia energii, w konsekwencji czego należało znaleźć inny sposób, niezaburzający koncepcji jednolitego API do zarządzania energią, które jest niezależne od specyfiki danej biblioteki. Na szczęście udostępniane nam są dwa miękkie limity (PL0 PL1) i możliwość odczytu aktualnego zużycia energii, które pozwalają w mniej lub bardziej dokładny sposób zasymulować twardy limit. Całość wymagała jedynie wprowadzenia klasy pomocniczej pośredniczącej w dostępie do danych związanych z zużyciem energii pomiędzy *IntelXeonDevice*, a *IntelXeonCommunicationProvider*. W klasie tej znajduje się dodatkowy wątek, który na bieżąco monitoruje zużycie energii. Na listingu 4.3. przedstawiono pseudokod algorytmu wykonującego tę operację, prezentowana tam funkcja „ustawPL0iPL1” ustawia odpowiednie (różniące się od siebie) wartości limitów, jednak podawanie szczegółowych wartości nie jest konieczne w kontekście ogólnego spojrzenia na proponowany algorytm. Warto również zwrócić uwagę na stałą „DŁUGOŚĆ_CZASU_STABILIZACJI” i zmienną „licznikCzasuStabliziacji”, które potrzebne są aby korekta nie była wykonywana zaraz po ustawieniu nowej wartości na skutek znajdowania się w buforze pomiarów wykonanych kiedy obowiązywał jeszcze wcześniejszy limit.

Listing 4.3. Pseudokod algorytmu kontroli limitu zużycia energii dla Intel Xeon Phi

```
while uruchomiony
   if licznikCzasuStabilizacji == 0
      buforKołowy.dodajPróbkę( pobierzAktualneZużycieEnergii() )
      średniaKrocząca ZużyciaEnergii = buforKołowy.pobierzŚrednią()

      if średniaKrocząca ZużyciaEnergii - TOLERANCJA > limitZużyciaEnergii
         if limitZużyciaEnergii - KROK_DOSTOSOWANIA >= dolnaGranicaLimitu
            ustawPL0iPL1( limitZużyciaEnergii - KROK_DOSTOSOWANIA )
            licznikCzasuStabilizacji = DŁUGOŚĆ_CZASU_STABILIZACJI
      else if średniaKrocząca ZużyciaEnergii + TOLERANCJA < limitZużyciaEnergii
         if limitZużyciaEnergii + KROK_DOSTOSOWANIA <= górnaGranicaLimitu
            ustawPL0iPL1( limitZużyciaEnergii + KROK_DOSTOSOWANIA )
            licznikCzasuStabilizacji = DŁUGOŚĆ_CZASU_STABILIZACJI
   else
      licznikCzasuStabilizacji -= 1

   czekaj( CZAS_POMIĘDZY_ODCZYTAMI )
```

4.4.5. Metoda odczytywania zużycia energii

W celu zapewnienia jak najwyższej dokładności odczytywanych wartości chwilowego zużycia energii, zdecydowano się na zastosowanie mechanizmu bardziej złożonego niż prosty odczyt za pomocą wywołania funkcji bibliotecznej. Dla każdego z urządzeń tworzony jest osobny wątek, który cyklicznie odpytuje urządzenie o jego zużycie energii, czyniąc to w ściśle określonych odstępach czasu, które mogą być definiowane na poziomie pliku konfiguracyjnego. Pomiary te są następnie umieszczane w buforze cyklicznym, którego rozmiar również jest określany w konfiguracji, a gdy serwer wyśle zapytanie o chwilowe zużycie energii na danym urządzeniu, zostaje zwrócona średnia arytmetyczna pomiarów znajdujących się w buforze. Uodparnia to aplikację na możliwe chwilowe wahania w zużyciu energii, które mogłyby być przyczyną zafałszowanych pomiarów. Należy pamiętać o odpowiednim dobraniu rozmiaru bufora, gdyż uśrednianie pomiarów ze zbyt długiego odcinka czasu również negatywnie wpłynie na ich dokładność.

4.4.6. Inne istotne element architektury

Tabela 4.3. Parametry linii komend agenta

Parametr	Wymagany	Przyjmowane wartości	Przeznaczenie
config	Tak	Każda poprawna ścieżka do pliku.	Określenie lokalizacji głównego pliku konfiguracyjnego.
withNVML		„true” lub „false”.	Określenie czy dana biblioteka ma być obsługiwana przez aplikację.
withNMPRK			
withMPSS			

Wszystkie parametry przedstawione w tabeli 4.3. muszą być poprzedzone dwoma myślnikami, a ich wartość podawana jest po znaku równości. W przypadku NVML i MPSS przekazanie wartości „true” będzie skutkowało załadowaniem odpowiednich bibliotek dynamicznych przez aplikację, o czym wcześniej wspominano. Opcja ta była konieczna również w przypadku NMPRK, gdyż wywołanie funkcji z tej biblioteki w przypadku gdy nie jest ona obecna w systemie operacyjnym, może mieć niepożądane efekty, np. w postaci nagłego zatrzymania się aplikacji.

Aplikacja korzysta z dwóch plików konfiguracyjnych. Pierwszy z nich zawiera wszystkie parametry potrzebne aplikacji do uruchomienia (np. adres i port, na którym ma nasłuchiwać, lokalizację pliku z konfiguracją logowania itd.), jest on zapisany w postaci: „klucz wartość".

Można w nim również umieszczać komentarze, poprzedzając je znakiem #. Ze względu na małą ilość potrzebnych parametrów, nie wymaga on skomplikowanej, rozbudowanej struktury, a w związku z tym do jego parsowania nie jest wykorzystywane żadne gotowe rozwiązanie. Cała logika związana z obsługą pliku konfiguracyjnego została zawarta w jednej specjalnie do tego przeznaczonej klasie, a więc w razie konieczności można bez trudu wymienić je na takie, które będzie oferowało większe możliwości.

Zawartość drugiego z plików konfiguracyjnych jest przekazywana bezpośrednio do biblioteki odpowiedzialnej za prowadzenie logów, a więc ma format specyficzny dla niej.

W istotnych miejscach programu informacje o przebiegu jego działania wypisywane są, z wykorzystaniem przeznaczonej do tego biblioteki, do logu. Każdy komunikat posiada przypisany poziom istotności, co pozwala na ich łatwe odfiltrowywanie i wartościowanie, wyliczając od najbardziej istotnych do najmniej:

- *FATAL* – krytyczny błąd, którego skutkiem będzie przerwanie działania aplikacji;
- *ERROR* – wystąpił błąd, który prawdopodobnie nie będzie skutkował przerwaniem działania aplikacji, ale może ona nie być w pełni funkcjonalna;
- *WARNING* – wystąpiło niepożądane zdarzenie, które nie wpłynie negatywnie na dalsze działanie aplikacji;
- *INFO* – standardowa informacja o przebiegu wykonania aplikacji;
- *TRACE* – jak wyżej, przy czym poziom istotności jest mniejszy;
- *DEBUG* – najmniej istotne informacje, przydatne przy szczegółowym badaniu działania aplikacji, dostępne jedynie, jeżeli typ konfiguracji budowania to „debug".

Istnieje tutaj duża swoboda konfiguracji całego procesu: zapis może odbywać się do pliku lub na standardowe wyjście, wypisywane mogą być wiadomości począwszy od danego poziomu istotności czy wreszcie można definiować strukturę pojedynczego wpisu. Wszystkie te parametry są w drugim z plików konfiguracyjnych.

4.5. Architektura serwera

Serwer składa się z dwóch aplikacji. Pierwsza, zwana API, jest punktem wejścia do systemu – to z nią użytkownik wchodzi w interakcję przez interfejs w postaci strony internetowej lub samodzielnie, wysyłając zapytania HTTP. Komunikuje się ona z bazą danych i agentem. Druga, zwana Zarządcą, uruchamia co określony czas zadanie, które polega na sprawdzeniu, czy ustawione są odpowiednie limity zużycia na wszystkich urządzeniach, pobraniu aktualnych statystyk używanej mocy z urządzeń oraz wykonaniu zadań zdefiniowanych przez reguły.

4.5.1. API

Wszystkie opisane poniżej moduły znajdują się w module *hpcpm.api*. Można go traktować jak odpowiednik przestrzeni nazw z innych języków programowania.

- *main* → Punktem wejścia do aplikacji jest moduł *main*, w którym znajdują się funkcje odpowiedzialne za wczytanie konfiguracji z pliku, parsowanie parametrów wiersza poleceń oraz skonfigurowanie *loggera* (ang. obiekt, którego zadaniem jest kierowanie tekstowych informacji – logów – w odpowiednie miejsce – logowanie). Przepływ sterowania w tym module jest stały, na końcu wywoływana jest funkcja inicjalizującą moduł *app* na podstawie konfiguracji wczytanej z pliku, a następnie zostaje do niego przekazane sterowanie.
- *app* → Moduł *app* posiada wcześniej wspomnianą funkcję *initialize*, która wywołuje funkcję konfigurującą moduł odpowiedzialny za komunikację z bazą danych oraz tworzącą obiekt klasy *ApiSpec*, definiujący istniejące zasoby REST API. Rejestruje ona utworzony obiekt w obiekcie *Flask*. Funkcja *run* odpowiada za utworzenie instancji serwera HTTP *Tornado* z kontenerem WSGI[10] utworzonym na podstawie obiektu *Flask* oraz uruchomienie go. Pozwala na zmianę liczby uruchomionych procesów obsługujących zapytania. Funkcje *start_request* i *end_request* wywoływane są odpowiednio przed i po obsłużeniu każdego zapytania HTTP przez serwer. Służą one logowaniu informacji oraz dodaniu odpowiednich nagłówków.
- *version* → Moduł version zawiera jedynie zmienną z oznaczeniem wersji aplikacji. Jest ona potrzebna w celu zbudowania paczki *wheel*.
- *Resources* → Jest to podmoduł modułu *hpcpm.api,* będący przestrzenią nazw dla modułów opisanych niżej.
- *ApiSpec* → Moduł ten zawiera klasę o tej samej nazwie. Posiada ona tylko konstruktor, który odpowiada za zdefiniowanie rutingu oraz zainicjalizowanie biblioteki *flask-restful-swagger*.
- *Endpoints* → Jest podmodułem modułu *hpcpm.api.resources,* zawierającym moduły z definicją udostępnianych przez API zasobów, które opisane są poniżej. Wszystkie one zawierają po jednej klasie nazywającej się tak samo, jak moduł.
- *Status* → Obsługuje zasób */status*. Udostępnia metodę GET, zwracającą zawsze kod 200 i treść „OK”. Pozwala na proste sprawdzenie czy aplikacja działa.
- *RuleTypes* → Udostępnia jako metodę GET zasobu */rule_types* listę obsługiwanych przez klas reguł zarządzania zużyciem energii. Jest ona zdefiniowana jako stała w pliku *Constants*.
- *nodes.computation_node* → Jest to podmoduł modułu hpcpm.api.resources, zawierający moduły i klasy określające jakie zasoby odpowiedzialne za operacje na węzłach obliczeniowych są udostępniane przez API, zostały one opisane poniżej.

[10] WSGI (ang. *Web Server Gateway Interface* – Interfejs bramy serwera sieciowego) – interfejs między serwerem aplikacji a aplikacją, stworzony dla języka Python.

- *AllComputationNodes* → Udostępnia metodę GET dla zasobu */nodes/computation_nodes*. Zwraca ona podstawowe informacje o wszystkich węzłach obliczeniowych zarejestrowanych w systemie: nazwę, adres oraz numer portu na którym działa agent.
- *ComputationNode* → Obsługuje metody GET, PUT i DELETE odwołujące się do zasobu */nodes/computation_node/<string:name>*. Dla metody GET zwraca wszystkie posiadane w bazie danych informacje o węźle, w tym m.in. informacje o zainstalowanych urządzeniach. Metoda PUT dodaje nowy węzeł do systemu na podstawie przekazanego adresu i portu. Odpytuje ona uruchomionego na nich agenta, w celu pobrania inicjalnych informacji. DELETE powoduje usunięcie wszystkich informacji o węźle z systemu.
- *PowerLimit* → Udostępnia metody GET, PUT i DELETE dla zasobu */nodes/computation_node/<string:name>/<string:device_id>/power_limit,* pozwalające na pobranie obecnie ustawionego limitu zużycia energii dla urządzenia, jego ustawienie bądź usunięcie. Żądanie ustawienia lub usunięcia zużycia powoduje natychmiastowe ustawienie go przez komunikację z agentem. Metoda GET zwraca jedynie wartość zapisaną w bazie danych. O jej poprawność dba Zarządca.
- *Rule* → Zapewnia obsługę metod GET, PUT i DELETE dla zasobu */nodes/computation_node/<string:name>/<string:device_id>/rule*. Pozwala to na pobranie, ustawienie oraz usunięcie rodzaju reguły zastosowanej dla danego urządzenia oraz jej parametrów, które są różne dla różnych reguł. Podczas ustawiania reguły sprawdzana jest jej obecność w liście obsługiwanych reguł.
- *StatisticsInterval* → Również udostępnia metody GET, PUT i DELETE. Obsługiwany zasób to */nodes/computation_node/<string:name>/<string:device_id>/statistics_interval*. Jedynym parametrem metody PUT jest interwał (w minutach), co który zbierane będą statystyki z danego urządzenia. Metoda DELETE usuwa interwał, co powoduje zaprzestanie zbierania statystyk.
- *StatisticsData* → Pozwala na odwołanie się do zasobu */nodes/computation_node/ <string:name>/<string:device_id>/statistics_data/<string:date_time>*. Dostępne metody to: GET, PUT i DELETE. Pozwala na operacje na danych statystycznych dla jednej konkretnej chwili. Wykorzystywana głównie przez Zarządcę. Metoda PUT dodatkowo aktualizuje w bazie danych datę i czas kolejnego zbierania statystyk na urządzeniu.
- *StatisticsDataWithInterval* → Udostępnia dla metody GET odwołującej się do zasobu */nodes/computation_node/<string:name>/<string:device_id>/statistics_data*
  posortowaną rosnąco według czasu listę chwilowego zużycia energii dla wszystkich punktów pomiarowych pomiędzy początkiem a końcem przedziału przekazanym jako parametry. Wymaga to dość skomplikowanych operacji na listach i słownikach pobranych z bazy danych.
- *Helpers* → Jest to podmoduł modułu *hpcpm.api*, w którym znajdują się moduły funkcji i stałych pomocniczych opisane poniżej.

- *Constants* → Zawiera definicje stałych używanych w aplikacji – głównie opisów parametrów i wartości zwracanych dla *flask-restful-swagger*.
- *Database* → Zawiera klasę *Database*, która jest zgodna z wzorcem singleton i zawiera metody dostępu do bazy danych. Izoluje logikę aplikacji od bazy danych. Dostęp do bazy w systemie odbywa się zawsze za pośrednictwem tego modułu.
- *Requests* → Zawiera funkcje wywołujące żądania HTTP do agenta. Pozwala on na łatwe dostosowanie serwera do zmian w agencie. API komunikuje się z agentem zawsze przez ten moduł.
- *Utils* → Moduł zawiera użyteczne w całej aplikacji funkcje, np. odrzucające zapytanie, gdy parametr nie jest poprawny.

4.5.2. Zarządca

Wszystkie opisane poniżej moduły znajdują się w module *hpcpm.management*. Zarządca jest aplikacją jednowątkową, jednoprocesową. W przypadku problemów z wydajnością, jest możliwość przeprojektowania go tak, aby używał biblioteki *Celery* pozwalającej zrównoleglić wykonywane przez niego operacje.

- *Main* → Analogicznie do odpowiedniego modułu z API, jest to punkt wejścia do aplikacji, który po ustawieniu konfiguracji przekazuje sterowanie do modułu *app*.
- *Rules* → Zawiera definicję klas reguł zarządzania zużyciem. Jest to klasa bazowa *Rule* oraz klasy po niej dziedziczące. W tej chwili jest to jedynie klasa *TimeBased*, która obsługuje regułę, którą można opisać językiem naturalnym, jako zbiór podreguł typu: „od daty i godziny X do daty i godziny Y ustaw limit zużycia na Z” oraz klasa *HardLimit* dla reguł typu „ustaw limit zużycia na Z”.
- *App* → Zawiera funkcję *run*, która definiuje przy pomocy biblioteki *APScheduler* wykonanie co określony, zdefiniowany w pliku konfiguracyjnym, czas wywołanie funkcji *do_work*. Funkcja *do_work* wywołuje trzy kolejne funkcje.
  1. Pierwsza z nich odpowiada za zebranie statystyk zużycia energii z urządzeń. W tym celu pobierana jest przez API lista węzłów obliczeniowych. Dla każdego węzła pobierana jest lista urządzeń. Dla każdego urządzenia pobierany jest ustawiony interwał pobierania statystyk wraz z datą i czasem kolejnego pobrania. Jeśli czas ten nadszedł, to agent jest odpytywany o statystykę dla urządzenia i jest ona wysyłana do API.
  2. Druga jest odpowiedzialna za sprawdzenie, czy na urządzeniach są ustawione właściwe limity zużycia energii. Działa ona podobnie jak poprzednia, ale dla każdego urządzenia sprawdza, czy w systemie ustalony jest limit zużycia energii. Jeśli tak, to odpytuje agenta o limit ustawiony fizycznie na urządzeniu. Jeśli jest on różny od zdefiniowanego w bazie, to Zarządca nakazuje agentowi ustawienie go we właściwy sposób.
  3. Trzecia natomiast służy do obsługi reguł zarządzania zużyciem. Dla każdego urządzenia sprawdza, czy zdefiniowana nazwa typu reguły jest zgodna z nazwą

którejś z klas dziedziczących po *Rule*. Jeśli tak, to tworzy obiekt tego typu, przekazując jako parametry nazwę węzła i ID urządzenia, a następnie wywołuje metodę *proceed* tego obiektu z parametrami, które otrzymała z API wraz z typem reguły. Nie przeprowadza walidacji tych parametrów.

- *Version* → Podobnie jak w API przechowuje zmienną z wersją aplikacji potrzebną do zbudowania paczki *wheel*.
- *api_requests* → Zawiera klasę posiadającą metody dostępu do części API serwera. Komunikacja z API zawsze odbywa się z użyciem tego modułu.
- *backend_requests* → Zawiera funkcje dostępu do agenta. Komunikacja z agentem zawsze odbywa się z użyciem tego modułu.
- *Utils* → Zawiera funkcje ogólnego przeznaczenia. W tej chwili jest to tylko funkcja *find*, pozwalająca na wyszukiwanie elementu w kolekcji (wraz z indeksem) na podstawie predykatu.

4.5.3. Schemat bazy danych

Zastosowana baza danych – MongoDB – jako baza dokumentowa nie posiada schematu. Przytoczone zostaną jednak przykładowe dokumenty dla każdej z kolekcji (kolekcja jest pewnego rodzaju odpowiednikiem tabeli z baz SQL). Wszystkie obiekty zawierają dodatkowo pole *_id*, które jest definiowane przez samo MongoDB i nie jest używane w systemie.

Kolekcja *statistics_intervals* przedstawiona na listingu 4.4. zawiera obiekty zawierające nazwę węzła, ID urządzenia, ustawiony interwał (w minutach) pobierania danych na temat zużycia oraz datę i czas następnego pobrania.

Listing 4.4. Przykładowy obiekt z *statistics_intervals*

```
{
  "interval": 1,
  "device_id": "GPU-7cf39d4a-359b-5922-79a9-506fddfc61e3",
  "name": "node1",
  "next_measurement": "2015-11-20T21:27"
}
```

Kolekcja *statistics_data* przedstawiona na listingu 4.5. zawiera obiekty z danymi na temat poboru mocy urządzeń. Jeden obiekt zawiera wszystkie statystyki danego urządzenia oraz jego ID i nazwę węzła. Ze względu na to, że wszystko jest przechowywane jako słownik, a więc nieposortowane względem czasu, wymagane są dość złożone operacje w serwerze, aby zwrócić posortowane dane dla danego okresu. Nie należy się jednak obawiać o wydajność – operacje na kolekcjach przy użyciu wbudowanych w Pythona funkcji są bardzo szybkie.

Listing 4.5. Przykładowy obiekt z kolekcji *statistics_data*

```
{
   "name": "node1",
   "device_id": "GPU-7cf39d4a-359b-5922-79a9-506fddfc61e3",
   "2015-11-20T12:58": 15,
   "2015-11-20T13:29": 120,
   "2015-11-20T13:30": 46
}
```

Kolekcja *rules* przedstawiona na listingu 4.6. zawiera obiekty, z których każdy zawiera nazwę węzła i ID urządzenia, dla którego zdefiniowana jest reguła, rodzaj reguły oraz jej parametry, które są dowolnym obiektem i są interpretowane przez kod reguły w module Zarządcy.

Listing 4.6. Przykładowy obiekt z kolekcji *rules*

```
{
   "name": "node1",
   "device_id": "GPU-7cf39d4a-359b-5922-79a9-506fddfc61e3",
   "rule_typ": "TimeBased",
   "rule_params": "{
      "klucz": "wartość"
   }"
}
```

Kolekcja *computation_nodes* przedstawiona jest na listingu 4.7. Każdy jej obiekt reprezentuje jeden węzeł obliczeniowy. W obiekcie znajdują się: adres i port, na którym nasłuchuje agent, nazwa zdefiniowana w systemie oraz obiekt *backend_info,* zawierający wszystkie dane zwrócone przez agenta, w tym listę z danymi na temat wszystkich obsługiwanych urządzeń.

Listing 4.7. Przykładowy obiekt z kolekcji *computation_nodes*

```
{
"address": "hostaddress",
"backend_info": {
   "nodename": "hostname",
   "supportsNVML": true,
   "supportsMPSS": true,
   "devices": [
   {
      "id": "GPU-7cf39d4a-359b-5922-79a9-506fddfc61e3",
      "Type": "NvidiaTesla",
      "info": {
         "PersistenceMode": "Disabled",
         "SerialNumber": "0324513074390",
         "PciBusId": "0000:00:00.0",
         "PcieLinkGeneration": "2",
         "InforomImageVersion": "2081.0208.01.09",
         "CurrentECCMode": "Enabled",
         "CurrentGpuOperationMode": "Compute",
         "UUID": "GPU-7cf39d4a-359b-5922-79a9-506fddfc61e3",
         "MaxMemoryClock": "2600",
         "ComputeMode": "Default",
         "Name": "Tesla K20m",
         "PowerManagementCapable": "true",
         "VBiosVersion": "80.10.39.00.04",
         "Index": "0",
         "DefaultPowerManagementLimit": "190"
      }
   }],
   "release": "2.6.32-042stab094.7",
   "version": "#1 SMP Wed Oct 22 12:43:21 MSK 2014",
   "supportsNMPRK": true,
   "sysname": "Linux",
   "machine": "x86_64"
},
"name": "node1",
"port": "40000"
}
```

Kolekcja *power_limits* przedstawiona została na listingu 4.8. Każdy z jej obiektów reprezentuje limit zużycia energii zdefiniowany dla danego urządzenia w węźle obliczeniowym. Zawiera on jedynie nazwę węzła, ID urządzenia i ustawiony limit zużycia.

Listing 4.8. Przykładowy obiekt z kolekcji *power_limits*

```
{
  "name": "node1",
  "device_id": "GPU-7cf39d4a-359b-5922-79a9-506fddfc61e3",
  "power_limit": "150"
}
```

4.5.4. Reguły

W systemie zaimplementowane zostały dwie klasy reguł zarządzania zużyciem energii. Możliwa jest jednak łatwa rozbudowa o nowe rodzaje. W tym celu w pliku *rules.py* Zarządcy należy utworzyć klasę dziedziczącą po klasie *Rule* z metodą *proceed*, która obsłuży regułę. Metoda ta będzie wywoływana dla każdego urządzenia z ustawioną regułą tego typu, co ustalony interwał. Metoda jako parametr otrzymuje obiekt *rule_params*, który jest obiektem parametrów reguły opisanym poniżej. Walidacja poprawności parametru spoczywa na twórcy reguły. Dostępne są również zmienne *self.node_name* i *self.device_id*, które zawierają odpowiednio nazwę węzła obliczeniowego i ID urządzenia, dla którego reguła jest obsługiwana oraz obiekt *self.api_request* dający dostęp do części API serwera. Parametr reguły jest definiowany przy jej ustawianiu poprzez odwołanie do zasobu */nodes/computation_node/<string:name>/<string:device_id>/rule* i może być dowolnym obiektem w formacie JSON.

Zaimplementowane zostały następujące klasy reguł:

- *HardLimit* – prosta reguła, która ustawia limit zużycia energii dla urządzenia na stałą wartość;
- *TimeBased* – reguła która pozwala ustawić wiele podreguł, z których każda definiuje początek i koniec zakresu czasu oraz limit zużycia energii, jaki ma obowiązywać w tym czasie; obiekt parametrów jest listą takich podreguł; przy ustawianiu reguły przez API należy zadbać o to, aby lista była posortowana względem czasu.

4.6. Architektura interfejsu użytkownika

Interfejs użytkownika jest aplikacją internetową opartą o pojedynczą stronę (SPA, ang. *Single Page Application*) napisaną przy użyciu HTML (ang. *HyperText Markup Language* – Hipertekstowy język znaczników) i JavaScript. Oznacza to, że wszystkie źródła, kod i skrypty pobierane są przy ładowaniu strony, a podczas interakcji z użytkownikiem przełączane są jedynie widoki. Dodatkowe dane pobierane są dynamicznie, co wyklucza konieczność przeładowywania strony w celu odświeżenia jej zawartości.

Strona aplikacji wyświetlana użytkownikowi opiera się o wyżej wspomniane widoki, które obsługiwane są przez odpowiednie kontrolery. Stanowi to pewne odwzorowanie modelu „model-widok-kontroler”. Kontrolery wykonują odpowiednie akcje, m.in. pobieranie danych,

pakowanie danych w struktury. Poza tym istnieją niezależne serwisy, dające dostęp do wybranych zasobów i funkcji z poziomu kodu całej aplikacji.

Wykorzystaną platformą programistyczną, który dostarcza te funkcjonalności, jest AngularJS. Oprócz wyżej wymienionych, pozwala ona na tworzenie prostych, dwustronnych powiązań pomiędzy zmiennymi w widoku i kontrolerze bez dodatkowego nakładu pracy, co zostało licznie wykorzystane w projekcie.

4.6.1. Serwisy

W projekcie interfejsu występują dwa serwisy, służące do uzyskiwania dostępu do danych globalnie:

1. *EndpointsService*;
2. *DataService*.

Serwis *EndpointsService* przechowuje wewnątrz siebie adresy do zasobów udostępnianych przez serwer, jak również adres bazowy, będący punktem wejściowym dla wszystkich zasobów, do których przekazywane będą wszelkie żądania REST.

Serwis *DataService* służy do właściwej komunikacji REST z serwerem. Dostarcza on metody przyjmujące odpowiednie parametry i opakowujące właściwe wywołania biblioteczne służące do komunikacji z zasobami serwera przez protokół HTTP.

4.6.2. Kontrolery

Kontrolery w interfejsie użytkownika obsługują całą logikę aplikacji. Komunikują się z serwerem i przygotowują dane dla widoków. W tej aplikacji dokładnie jeden kontroler odpowiada za obsługę dokładnie jednego widoku lub jego części.

Oto kolejne dziesięć zaimplementowanych kontrolerów:

1. *BodyController;*
2. *DeviceStatsShowController*;
3. *DeviceStatsIntervalModalController*;
4. *NodesController*;
5. *NewNodeController*;
6. *NodeDetailsController*;
7. *NodeDetailsModalController*;
8. *PowerLimitModalController*;
9. *RemoveNodeController*;
10. *RemovePowerLimitModalController*.

Kontroler *BodyContoller* odpowiada za obsługę ciała strony głównej aplikacji, a także za obsługę okna dodawania nowego węzła obliczeniowego do bazy danych. Jest też kontrolerem nadrzędnym dla kontrolera *NodesController*.

Kontroler *DeviceStatsShowController* służy do uzyskania danych wejściowych od użytkownika i pobierania danych dotyczących statystyk zużycia energii z serwera dla poszczególnych urządzeń z węzłów obliczeniowych. Na podstawie uzyskanych statystyk z bazy, tworzy on model wykresu, który zostaje wyświetlony w skojarzonym widoku, ewentualnie – w przypadku niedostępności danych dla wybranego przedziału czasowego – przesyła stosowny komunikat o błędzie do widoku.

Kontroler *DeviceStatsIntervalModalController* służy do ustawiania pobranego od użytkownika interwału, określającego czas (w minutach), co który zostaną pobrane dane statystyczne i wysłania go do serwera, który wykorzysta go do uzyskania tych danych z urządzeń przez agenta. Pozwala również na usunięcie tej wartości.

Kontroler *NodesController* jest właścicielem modelu tabeli, w której znajdują się podstawowe dane na temat każdego z dodanych do bazy węzłów obliczeniowych. Dane są cyklicznie odświeżane przez kontroler. Umożliwia on również przejście do kontrolera pozwalającego na usuwanie węzłów obliczeniowych z bazy danych.

Kontroler *NewNodeController* odpowiada za przyjęcie danych opisujących nowy węzeł obliczeniowy i dodanie ich w odpowiedniej formie do bazy. Po dodaniu nowego węzła, wysyła komunikat dotyczący odświeżenia tabeli do węzła *NodesController*.

Kontroler *NodeDetailsController* nadzoruje widok szczegółów węzła obliczeniowego. Umieszcza w modelu tabeli informacje pobrane z serwera, w tym listę obsługiwanych urządzeń, które znajdują się w węźle, a ponadto pobiera i ustawia zmienne, które opisują szczegółowo węzeł: jego nazwę w bazie, adres, port, na którym działa agent, nazwę hosta, nazwę systemu, wersję, architekturę i wydanie (np. jądra). Odpowiada również za przekazanie odpowiednich parametrów do innych kontrolerów odpowiedzialnych za powiązane z urządzeniami akcje.

Kontroler *NodeDetailsModalController* to kontroler odpowiadający za wyświetlenie danych z *NodeDetailsController* w całości i w „surowej” formie, dostarczając również możliwość ich skopiowania do schowka.

Kontroler *PowerLimitModalController* służy pobraniu od użytkownika danych dotyczących limitu zużycia energii, jaki ma zostać ustawiony (w postaci określonej reguły, o których mowa jest w podrozdziale 4.5.4.) i przesłania go do serwera, w celu zastosowania jego, a także przekazaniu do widoku obecnego limitu (reguły).

Kontroler *RemoveNodeController* obsługuje żądanie użytkownika dotyczące usunięcia wybranego węzła obliczeniowego.

Kontroler *RemovePowerLimitController* umożliwia realizację żądania użytkownika polegającego na usunięciu zastosowanej wcześniej reguły limitu zużycia energii z konkretnego urządzenia w węźle obliczeniowym, a także pobiera i przekazuje do widoku obecną regułę.

4.6.3. Widoki

Widoki w aplikacji to szablony zawarte w pliku HTML. Zmieniane są zależnie od wyborów użytkownika i powiązane są ze swoimi kontrolerami poprzez powiązania zmiennych, jak i wywołania funkcji. Widoki zmieniają się na podstawie przejść pomiędzy adresami, które obsługiwane są przez odpowiednio skonfigurowany trasownik widoków. W wypadku zmiany widoku wykonywana jest operacja na drzewie DOM (ang. *Document Object Model* – obiektowy model dokumentu), polegająca na podmianie węzła wewnątrz określonego i wyróżnionego elementu DIV na wybrany szablon, co oznacza brak konieczności przeładowania strony, gdyż zgodnie z założeniem SPA, wszystkie szablony widoków zostają pobrane przy pierwszym załadowaniu strony aplikacji. Taka zmiana widoku jest natychmiastowa.

W aplikacji występuje dziesięć widoków, z czego jeden jest widokiem bazowym, wewnątrz którego umieszczane są inne. Te widoki to:

- Widok bazowy;
- *NodeDetails.html*;
- *DeviceStatsShow.html*;
- *DeviceStatsIntervalModal.html*;
- *NodeDetailsModal.html*;
- *PowerLimitModal.html*;
- *RemovePowerLimitModal.html*;
- *Nodes.html*;
- *NewNode.html;*
- *RemoveNode.html*.

Widoki prezentują rezultaty działania kontrolerów o tej samej nazwie – wyświetlają przygotowane tabele, udostępniają pola tekstowe do wpisywania danych wejściowych, umożliwiają wykonanie odpowiednich akcji (dodanie, usunięcie, skopiowanie), a także wyświetlają komunikaty dotyczące powodzenia lub niepowodzenia wykonanych przez kontrolery akcji. Nazwy widoków zawierają rozszerzenie „.html”, jest to jednak tylko identyfikator – nie oznacza to, że zawierają się w osobnych plikach, choć jest to możliwe do wykonania.

5. OPIS IMPLEMENTACJI

5.1. Agent

5.1.1. Język programowania

Agent został w całości napisany w języku C++, korzystając z możliwości oferowanych przez C++11 [Stroustrup B., 2013]. Wprowadza to co prawda wymaganie, aby kompilator był zgodny z tym standardem, co było źródłem pewnych problemów (zostały one opisane w podrozdziale 8.8.), jednak było to konieczne ze względu na bibliotekę CPPREST, która wykorzystuje C++11. Dodatkowo wybór ten ma wiele zalet, gdyż pozwoliło to na wykorzystanie oferowanych przez ten standard możliwości, które usprawniają proces pisania kodu i czynią wynikowy kod bardziej eleganckim i zrozumiałym, między innymi:

- wyrażenia lambda;
- silnie typowane wyliczenia;
- automatyczne określanie typu – specyfikator auto;
- jednolite inicjowanie;
- pętla for oparta o zakres;
- static assert;
- nullptr.

5.1.2. Wspierane platformy

Mając na uwadze fakt, że środowiskiem docelowym, na którym ma działać agent, jest Linux, jedynie ta platforma jest wspierana w jego aktualnej wersji. Wiąże to się głównie z faktem, że znakomita większość środowisk, w których wykonywane są obliczenia, to właśnie te, na których jest zainstalowany ten system operacyjny. W razie ewentualnej konieczności rozszerzenia wsparcia na inne systemy operacyjne konieczne będzie zastąpienie pewnego zestawu wywołań systemowych, z których korzysta aplikacja, w szczególności ładowanie bibliotek dynamicznych i pobieranie statystyk systemowych.

5.1.3. Sposób kompilacji

Mimo iż nie jest to aplikacja wieloplatformowa i można by ograniczyć się do specyficznego dla platformy rozwiązania (np. GNU Make), to jako metodę kompilacji wybrano generator systemów budowania w postaci CMake. Jego niewątpliwą zaletą jest prostota, znacznie większa niż w przypadku GNU Make, a dodatkową zaletą jest uniwersalność – w razie konieczności dodania wsparcia dla innej platformy wymagane będą jedynie drobne zmiany w pliku konfiguracyjnym *CMakeLists.txt*. CMake nie jest sam w sobie systemem budowania, a generatorem takowych, więc równie łatwo, jak pliki GNU Make można wytworzyć solucję do Visual Studio. Proces kompilacji jest dwufazowy. W pierwszej fazie generowane są pliki dla wybranego systemu budowania, a w drugiej fazie system ten zostaje uruchomiony. Stworzony na potrzeby agenta plik CMake pozwala na:

- wybór typu budowania: „debug” (odpowiedni do debugowania), *release* (do wdrożenia), lub *test* (służący do wykonywania testów jednostkowych);
- określenie, czy aplikacja ma korzystać z rzeczywistych klas komunikujących się z bibliotekami do zarządzania energią, czy z ich atrap (opisanych w podrozdziale 8.3.3.);
- automatyczne wykrycie bibliotek niezbędnych do kompilacji, o ile znajdują się w którejś z dobrze znanych lokalizacji w systemie plików;
- automatyczne ustawienie flag kompilacji w zależności od wybranego typu konfiguracji budowania.

5.1.4. Wykorzystane biblioteki

Dołączane statycznie:

- Easylogging++ – prosta, wygodna i zawierająca się w pojedynczym pliku nagłówkowym biblioteka służąca do prowadzenia logów;
- C++ REST SDK – udostępniania przez Microsoft, jest jednym z najistotniejszych elementów aplikacji, gdyż za jej pomocą odbywa się cała komunikacja sieciowa;
- NMPRK – biblioteka do zarządzana zużyciem energii przeznaczona dla procesorów Intel Xeon;
- Boost – ogromny zbiór bibliotek o szerokiej gamie zastosowań będący bardzo istotnym elementem języka C++, stanowi on swoisty przedsionek, z którego poszczególne biblioteki są włączane do standardu tego języka; w aplikacji wykorzystano biblioteki: Xpressive (do obsługi wyrażeń regularnych) i Tokenizer (do podziału łańcuchów znaków na elementy składowe); mimo że w standardzie C++11 znajduje się biblioteka obsługująca wyrażenia regularne to jej implementacja w gcc[11], z którego korzystano (wersja 4.8) jest niepełna i nie spełniała naszych wymagań, dlatego zdecydowano się na skorzystanie z Boost.Xpressive; Boost.Tokenizer można w bardzo łatwy sposób zastąpić własną implementacją, więc jeżeli zajdzie konieczność usunięcia zależności od Boost, nie powinno stanowić to problemu;
- Googletest i wchodzący w jego skład googlemock – pozwalające na pisanie i uruchamiane testów jednostkowych, wykorzystywane jedynie przy typie konfiguracji budowania „test”.

Dołączane dynamicznie:

- libdl – służąca do ładowania w czasie działania aplikacji bibliotek dynamicznych.

Ładowane w trakcie działania aplikacji:

- NVML – służy do zarządzania kartami Nvidia, a w szczególności sterowania zużyciem energii na oferujących taką możliwość urządzeniach;
- MPSS – podobnie jak powyższa, jednak w tym przypadku obsługiwane są urządzenia Intel Xeon Phi.

[11] gcc (ang. *GNU compiler collection*) – zestaw kompilatorów rozwijanych przez GNU i dostępnych dla systemu operacyjnego Linux, w szczególności kompilator języka C++.

Decyzja, aby większą część bibliotek dołączać statycznie, była podyktowana problemami z kompilacją w środowisku testowym. Docelowo takie rozwiązanie nie jest korzystne ze względu na bardzo duży rozrost pliku wykonywalnego, ale powrót do linkowania dynamicznego większości bibliotek nie stanowiłby żadnego problemu – wymaga to jedynie prostych zmian w pliku konfiguracyjnym CMake.

5.2. Serwer

5.2.1. Język programowania

Serwer jest modułem aplikacji, w którym znajduje się największa część logiki systemu. Miał on być łatwy do edycji i rozbudowania, a jego działanie miało być pewne i stabilne. Z tego względu zdecydowano, że głównymi kryteriami wyboru języka programowania będą: łatwość i szybkość pisania kodu, jego czytelność oraz modularność. Zdecydowano się na wybór języka Python w wersji 3.4. Za tym wyborem przemawiał również fakt, że dwóch z trzech członków zespołu ma doświadczenie w wytwarzaniu aplikacji tego typu w Pythonie.

Python jest językiem skryptowym ogólnego przeznaczenia o dynamicznym typowaniu. Posiada on bardzo rozbudowany zasób bibliotek w repozytorium PyPI (ang. *the Python Package Index* – Indeks paczek Pythona) instalowanych w łatwy sposób przy użyciu narzędzia PIP, które od wersji 3.4 Pythona stało się jego oficjalnym menadżerem pakietów. Python pozwala na odseparowanie środowiska uruchamianej aplikacji wraz z jej zależnościami od innych przez wirtualne środowiska (przy użyciu narzędzia virtualenv). Środowisko takie posiada własny interpreter Pythona, a biblioteki instalowane są lokalnie, zamiast w środowisku głównego interpretera [Summerfield M., 2010]. Język ten teoretycznie nie cechuję się wysoką szybkością działania, jednak ze względu na to, że wiele potrzebnych algorytmów i funkcji jest obecnych w bibliotece standardowej lub bibliotekach osób trzecich, często są one zaimplementowane w sposób na tyle optymalny, ze różnica w stosunku do innych języków zaciera się. Należy pamiętać o używaniu gotowych funkcji do zadań takich jak operacje na kolekcjach, zamiast implementować je samodzielnie [Gorelick M., Ozsvald I., 2014].

5.2.2. Wspierane platformy

Aplikacja działa w dowolnym środowisku w którym jest obecny interpreter Pythona 3.4. Jednak twórcy jednej z kluczowych bibliotek, Tornado, zalecają, aby środowiska Windows i OS X używać tylko do celów deweloperskich, a dla zastosowań produkcyjnych wybrać Linuksa lub BSD [Tornado Web Server]. Jako że cały system z założenia miał działać pod kontrolą Linuksa, nie jest to ograniczeniem. Dodatkowo przygotowane skrypty do uruchamiania testów i budowania projektu znajdują się w pliku *Makefile*, do użycia z popularnym na Linuksie programem GNU Make.

5.2.3. Sposób budowania projektu

Zasadniczo, projektów w Pythonie nie trzeba w żaden sposób budować. Wystarczy nakazać interpreterowi, który ma zainstalowane odpowiednie biblioteki potrzebne dla aplikacji, wykonać wejściowy plik .py (najczęściej nazywa się on „*main.py*”). Jednak dla wygody często zbiera się wszystkie pliki źródłowe wraz z dodatkowymi metadanymi w paczki *.whl* (*wheel*), które pozwalają na zainstalowanie aplikacji w środowisku interpretera przy użyciu pip, pobranie zależności z repozytorium PyPI, a następnie uruchomienie jej jak pliku wykonywalnego. W celu utworzenia paczki *.whl* zostały utworzone odpowiednie wpisy w pliku *Makefile*. Znajdują się tam również wpisy pozwalające na wykonanie statycznej analizy kodu, testów jednostkowych, czy utworzenia wirtualnego środowiska i pobrania do niego zależności oraz zainstalowanie zależności systemowych, na przykład Pythona 3.4.

5.2.4. Wykorzystane biblioteki

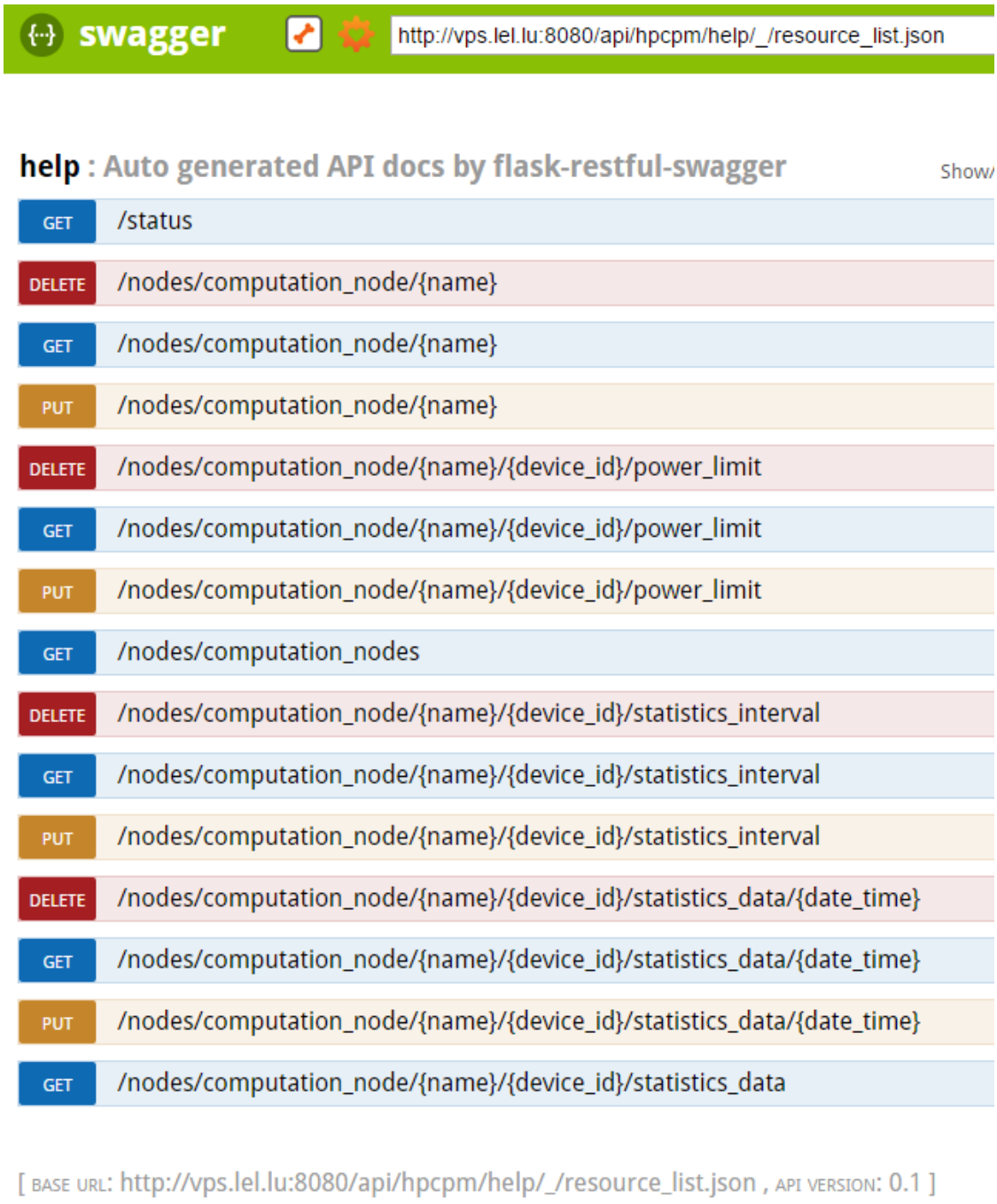
Wykorzystane biblioteki języka Python można podzielić na dwie kategorie: konieczne do działania aplikacji oraz potrzebne do jej testowania i budowania artefaktów.
Biblioteki konieczne do działania aplikacji:

- podstawową biblioteką, potrzebną do działania części API serwera, jest *Flask*. Jest to *microframework* (lekkie, minimalistyczne środowisko przeznaczone do wytwarzania aplikacji internetowych, w przeciwieństwie do na przykład ASP.NET). Pozwala on na definicję ścieżek HTTP przypisanych do obsługujących je funkcji przez dekorator *@app.route(<<ścieżka>>).* Flask zajmuje się również uruchomieniem kontenera WSGI, co pozwala na uruchomienie wielu procesów obsługujących zapytania przy użyciu gunicorn, UWSGI czy Tornado (opisanego poniżej). Wydawany jest on na licencji BSD;
- kolejną przydatną biblioteką jest *Flask-RESTful*. Jest to rozszerzenie Flaska, pozwalające na łatwe tworzenie REST API. Każdy punkt końcowy API jest w kodzie reprezentowany przez klasę dziedziczącą po *flask_restful.Resource*, która implementując metody o nazwach takich jak metody HTTP (np. GET, POST, DELETE), udostępnia je dla danego punktu końcowego (ang. *endpoint*). Ułatwia ona też dostęp do różnego rodzaju parametrów zapytania. Był on początkowo tworzony jako projekt na wewnętrzny użytek firmy Twillo, ale został udostępniony do publicznego użytku i modyfikacji. Jedynym obostrzeniem licencji jest zakaz używania nazwy Twillo do promowania produktów powstających na bazie biblioteki;
- przydatną (choć nie niezbędną) biblioteką jest *flask-restful-swagger*. Ułatwia ona dokumentowanie i testowanie REST API przez udostępnienie prostego interfejsu (rys. 5.1) do jego obsługi, co bardzo ułatwia wytwarzanie aplikacji. Użycie jej wymaga napisania zaledwie kilku wierszy kodu dokumentującego dla każdej metody HTTP zasobu oraz współpracę z narzędziem Swagger tworzonym przez firmę SmartBear. Niestety w październiku 2015 roku projekt przestał być rozwijany i wspierany przez twórcę, konieczne może więc być zaprzestanie jego używania w przyszłości. Udostępniany na licencji MIT;

- tornado jest bardzo rozbudowanym asynchronicznym serwerem HTTP i środowiskiem programistycznym dla Pythona. W projekcie jednak jest używana tylko jedna z jego funkcji – uruchomienie wielu instancji kontenera WSGI utworzonego przez Flaska. Udostępniany na licencji Apache 2.0;
- *PyMongo* to tworzona przez twórców MongoDB biblioteka dla Pythona do komunikacji z tą bazą danych. Wyjątkowo prosta w użyciu. Udostępnia wszystkie możliwe operacje jako metody obiektu kolekcji. Utworzenie tego obiektu, jak i połączenia z bazą, jest wyjątkowo łatwe. Wydawana na licencji Apache 2.0;
- dodatkowo używana jest biblioteka *Requests*, która ułatwia wykonywanie zapytań HTTP do agenta oraz z Zarządcy do API. Licencja to Apache 2.0;
- w module Zarządcy używana jest biblioteka *APScheduler*, która pozwala odroczone oraz powtarzalne wykonywanie zadań. Wydawana na licencji MIT.

Biblioteki potrzebne do testowania lub budowania aplikacji:

- *PEP8, PyLint* i *PyFlakes* to narzędzia do statycznej analizy kodu napisanego w Pythonie. Ich działanie jest szczegółowo opisane w podrozdziale 8.4.1. Nie są one używane jako biblioteki w aplikacji, a uruchamiane jak pliki wykonywalne. Są udostępniane na licencjach odpowiednio: MIT (nazywanej przez twórców „Expat”), GNU GPL oraz MIT;
- narzędziem używanym do testowania jednostkowego kodu w Pythonie jest *pytest*. Jest to najpopularniejszy tego typu program. Wydawany na licencji MIT;
- *Flask-Testing* udostępnia kilka funkcji ułatwiających testowanie jednostkowe aplikacji pisanych przy użyciu Flaska. Wydany na zmodyfikowanej licencji BSD;
- *Coverage* oraz *pytest-cov* użyte razem służą do zbierania danych na temat pokrycia kodu aplikacji testami jednostkowymi. Generują pliki HTML obrazujące pokrycie. Są one na licencji odpowiednio Apache 2.0 i MIT;
- *Wheel* służy do tworzenia opisanych wcześniej paczek *.whl,* które są obecnie standardem w Pythonie. Wydany na zmodyfikowanej licencji BSD.



Rys. 5.1. Interfejs flask-restful-swagger

5.2.5. Baza danych

Systemem bazodanowym wykorzystanym do przechowywania danych przez serwer jest MongoDB. MongoDB jest wieloplatformowym, zorientowanym na dokumenty systemem zarządzania bazami danych. Zakwalifikowano go jako NoSQL, ponieważ unika relacyjności zbudowanej na tabelach, na rzecz tak zwanych dokumentów w formacie obiektów JSON z dynamiczną strukturą. Dokonaliśmy takiego wyboru ze względu na to, że format przechowywania danych przez MongoDB jest naturalnym formatem danych używanym przez język, w którym powstaje serwer, tj. Python. Bardzo łatwo przejść z obiektu JSON do słownika w Pythonie, dzięki czemu konieczność konwersji zwracanych danych ograniczona jest do minimum. Również dynamiczna struktura jest niezwykle przydatnym elementem tej bazy danych. Ze względu na to, iż różne urządzenia i węzły obliczeniowe mogą dostarczać dane w różnych formatach i w różnej ilości, to dokumenty w postaci JSON są doskonale dopasowane do tych wymagań, czego nie mogłyby zapewnić tabele klasycznych, relacyjnych baz danych, posiadające sztywny, z góry zdefiniowany format (kolumny). Jako przedstawiciel NoSQL, MongoDB wykazuje przewagę w szybkości przetwarzania danych [Nayak A., 2014].

Dodatkowym i istotnym powodem, dla którego wykorzystujemy MongoDB jest jego użycie w projekcie Kernel Hive na katedrze ASK, z którym docelowo nasz projekt ma zostać zintegrowany. Nie jest to w żadnym stopniu konieczne wymaganie, lecz postawiliśmy na spójność rozwiązań.

Do komunikacji pomiędzy serwerem a bazą MongoDB wykorzystywany jest interfejs programistyczny PyMongo. Dostarcza on komplety zestaw metod i dokumentację, dzięki którym możliwe było zaimplementowanie obsługi bazy w serwerze.

Baza danych użyta w projekcie składa się z kolekcji, które przechowują opisane wyżej dokumenty JSON. Istnieje pięć kolekcji: kolekcja węzłów obliczeniowych, kolekcja limitów zużycia energii, kolekcja statystyk, kolekcja z interwałami zbierania statystyk i kolekcja reguł.

Zapytania kierowane do bazy danych wywoływane są za pomocą odpowiednich metod dostarczonych przez bibliotekę PyMongo. W większości przypadków, jako parametry tych zapytań używane są obiekty JSON, np. wybieranie elementu z kolekcji, jego aktualizacja bądź usunięcie. Rezultatem wywołania takiej funkcji w wielu przypadkach są również obiekty JSON, np. w przypadku usunięcia dokumentu z kolekcji, zostaje on zwrócony, jako potwierdzenie pomyślnego wykonania takiej operacji.

5.3. Interfejs użytkownika

5.3.1. Język programowania

Interfejs użytkownika to połączenie głównie dwóch technologii dominujących obecnie w aplikacjach internetowych, tj. HTML i JavaScript. Aplikacja interfejsu została wykonana w koncepcji SPA, czyli aplikacji opartej o pojedynczą stronę. Interfejs zawiera dokładnie jeden plik HTML – *index.html*, wewnątrz którego znajdują się wszystkie szablony-widoki, a także załączenia skryptów i stylów. Wybraliśmy HTML jako język, który jest powszechnie nam znany

w dobrym stopniu. Możliwe były wybory innych języków, kompilowanych na końcu do HTML-a, jednakże wymagałoby to opanowania ich w dość dobrym stopniu, co przyczyniło się do pozostania przy sprawdzonym rozwiązaniu.

Za szatę graficzną interfejsu użytkownika odpowiada w całości platforma Bootstrap, gwarantująca spójny wygląd elementów graficznych, a także dostosowanie wyglądu strony do różnych urządzeń o różnych rozdzielczościach ekranów, w oparciu o system podziału strony na kolumny. Ograniczyła ona konieczność definiowania własnych stylów, umożliwiając wykorzystanie gotowych [Bhaumik S., 2013].

Strona skryptowa aplikacji to kod źródłowy w całości napisany w języku JavaScript – kontrolery oraz serwisy opisane w podrozdziale 4.6. Jest to obecnie najszerzej stosowana technologia programowania logiki interfejsu po stronie użytkownika. Jest to również język wspierany przez wszystkie najpopularniejsze przeglądarki internetowe, dzięki czemu nie ma problemu z jego nieobsługiwaniem po stronie klienta – wyjątkiem jest użytkownik, który świadomie wyłącza obsługę JavaScript w swojej przeglądarce. Istnieje wiele alternatyw dla JavaScript, w tym języki dostarczające szereg unowocześnień i usprawnień, jak obiektowość czy silne typowanie, jednakże są to zazwyczaj języki ostatecznie kompilowane do czystego JavaScriptu i, podobnie jak w przypadku alternatyw dla HTML-a, wymagałyby one zdobycia wiedzy na ich temat i praktyki w pisaniu w nich kodu.

5.3.2. Wspierane platformy

Platformami wspieranymi przez aplikację interfejsu są wszystkie platformy, na których działają przeglądarki internetowe z obsługą JS. W przypadku przeglądarki Internet Explorer, wymagana jest wersja przynajmniej 9, ze względu na to, że wykorzystywana biblioteka AngularJS nie wspiera niższych wersji i ze względu na problemy z niektórymi rozwiązaniami CSS (ang. *Cascading Style Sheets* – kaskadowe arkusze stylów).

Środowisko deweloperskie jest możliwe do uruchomienia na wszystkich platformach pozwalających na uruchomienie programów takich jak Node.js, gdyż to na nim się ono głównie opiera. Kwestię platform wspieranych przez przygotowany serwer HTTP poruszono w podrozdziale 5.3.5.

5.3.3. Sposób budowania projektu

Wykorzystywane są dwa rodzaje zależności, koniecznych do działania, konfigurowania i budowania projektu:

- zależności serwerowo-deweloperskie;
- zależności klienckie.

Za zależności serwerowo-deweloperskie odpowiada program NPM, będący składnikiem platformy programistycznej Node.js. Zależności te to biblioteka serwera HTTP – Express i biblioteki dla niej pomocnicze oraz po stronie deweloperskiej: Grunt – służący do automatyzacji zadań w czasie tworzenia projektu, wtyczki do Grunta oraz Karma i Jasmine – służące do uruchamiania testów jednostkowych. Znajdują się one wszystkie w pliku *package.json*, co

umożliwia prostą instalację wszystkich zależności na raz. Po instalacji umieszczane są one w folderze node_modules.

Zależności od strony klienta zarządzane są przez program Bower. Jest to głównie biblioteka AngularJS, będąca trzonem aplikacji. Pozostałe zależności to moduły wspomagające, służące do prezentacji określonych rzeczy: wykresów, plików JSON, tabeli, dat oraz komunikacji z serwerem przez HTTP. Znajdują się one, podobnie jak w przypadku poprzednich zależności, w pliku bower.json, a po instalacji dostępne są w folderze bower_components.

Projekt interfejsu nie wymaga sam w sobie zbudowania go w jakikolwiek sposób, ponieważ działa on na lokalnych komputerach ze względu na to, że wszystkie potrzebne biblioteki są dołączane z plików lokalnych. Jednakże, aby uprościć strukturę gotowej aplikacji, został przygotowany schemat minimalizacji (czyli usuwania białych znaków i skracania nazw funkcji i zmiennych) i łączenia skryptów JS i plików CSS, w celu zmniejszenia ich rozmiaru, co przekłada się na szybsze ładowanie aplikacji, a także w celu uporządkowania załączania ich w kodzie HTML. Tym procesem zajmuje się wyżej wspomniany Grunt, który pobiera reguły i kolejność zadań z pliku o nazwie *Gruntfile.js*. Wykonywane są kolejno takie zadania: analiza statyczna kodu, testowanie jednostkowe, kopiowanie plików HTML i konfiguracyjnych, łączenie dołączonych bibliotek, łączenie skryptów zawierających kontrolery i serwisy w jeden plik, łączenie stylów, minimalizacja tych plików. Wynikowe pliki przenoszone są do katalogu *dist*, który wykorzystywany jest przez QuickBuild jako katalog źródłowy artefaktów wynikowych. Grunt pozwala również na uruchamianie przygotowanego serwera HTTP i przebudowywanie projektu, gdy nastąpi zapisanie zmian w którymkolwiek z jego plików.

Dodatkowo został przygotowany plik *Makefile*, umożliwiający zainstalowanie zależności systemowych, uruchamianie serwera i budowanie projektu przy użyciu polecenia *make*, popularnego w środowiskach linuksowych.

5.3.4. Wykorzystane biblioteki

Wszystkie biblioteki wykorzystane po stronie klienta zostały pobrane z repozytorium aplikacji Bower, która, jak zostało wspomniane w poprzednim rozdziale, zarządza nimi.

Główną biblioteką, która jest zarazem platformą programistyczną, na której opiera się interfejs w tym projekcie jest *AngularJS*. Jest to otwartoźródłowy projekt (licencja MIT), ułatwiający tworzenie dynamicznych aplikacji internetowych. Istotną cechą AngularJS są wiązania zmiennych pomiędzy kontrolerami a widokami. Nie wymagają one od programisty jakichkolwiek operacji na drzewie DOM strony HTML, ponieważ każda zmienna w zakresie kontrolera jest dostępna w widoku poprzez umieszczenie wewnątrz znaczników instrukcji {{nazwa_zmiennej}}. Powiązania między kontrolerami a widokami są obustronne – modyfikacja zmiennej w jednym powoduje odwzorowanie tej zmiany w drugim i na odwrót. Znanym problemem jest wyraźny spadek wydajności aplikacji przy bardzo dużej liczbie takich powiązań, co na szczęście nie stanowiło problemu w tym projekcie, ze względu na umiarkowaną liczbę tych rozwiązań [Kalbarczyk A., Kalbarczyk D., 2015].

Kolejną istotną wykorzystywaną biblioteką jest *jQuery* (licencja MIT). Wykorzystywana jest ona przez AngularJS, jednakże nie jest konieczna.

Biblioteką, która odpowiada za komunikację RESTful z serwerem jest *Restangular*. Jest to otwarta biblioteka (licencja MIT), oparta na obietnicach, czyli asynchronicznych operacjach wykonywanych wtedy, kiedy nadejdzie odpowiedź serwera na zadane wcześniej zapytanie. W ten sposób unika się konieczności cyklicznego sprawdzania, czy odpowiedź serwera już nadeszła. Dostarcza ona również bardzo wygodną i czytelną składnię zapytań, co przekłada się na minimalizację kodu potrzebnego do obsługi komunikacji.

AngularUI Router (licencja MIT) służy trasowaniu widoków na podstawie przejść pomiędzy adresami URL, dzięki czemu nie występują żadne przeładowania strony przy przechodzeniu do innego widoku w aplikacji.

Kolejną ważną biblioteką jest biblioteka *ngTable* (uproszczona licencja BSD). Służy ona do budowania dynamicznych tabel przy wykorzystaniu dyrektyw AngularJS. Pozwala na filtrowanie i stronicowanie danych i dynamiczne ich odświeżanie, przy zapewnieniu odpowiedniej wydajności.

Biblioteki użyte do prezentacji lub pobierania określonych danych to:

- *Chart.js* – prezentacja wykresów, licencja MIT;
- *Combodate* – pobieranie przedziału czasowego na podstawie wyboru użytkownika, licencja MIT;
- *AngularJS-Toaster* – wyświetlanie powiadomień-dymków, licencja MIT;
- *JSON Formatter* – przedstawianie dokumentu JSON jako interaktywnego, rozwijanego drzewa, licencja Apache 2.0.

Dodatkowo używane są biblioteki, które służą jako zależności wyżej wymienionych bibliotek i musiały zostać dołączone do pliku HTML.

5.3.5. Serwer HTTP

Aplikacja interfejsu użytkownika do działania wymaga odpowiednio skonfigurowanego serwera HTTP. Może być to dowolny serwer HTTP, jednakże w ramach projektu został przygotowany dedykowany jemu serwer. Jest to serwer napisany w oparciu o biblioteki Node.js oraz Express. Możliwe jest jego uruchomienie na każdym systemie, w którym da się uruchomić aplikację Node.js. Dostarcza on podstawowe trasowanie – serwuje plik *index.html* przy wejściu na adres strony głównej, a także serwuje pliki skryptów i bibliotek pod odpowiednimi ścieżkami. Loguje także wszystkie kierowane do niego żądania w czytelnej formie. Można uruchomić go ręcznie, przez program Node.js, przez wywołanie go z poziomu Grunta lub też z poziomu pliku *Makefile*. Jest on przenośny, przechowuje swoje ustawienia dotyczące ścieżek do komponentów i portu, na którym ma działać w pojedynczym pliku o nazwie *config.js*.

6. OPIS I WYNIKI PRZEPROWADZONYCH TESTÓW

6.1. Test uruchomieniowy systemu w środowisku docelowym

Pierwszy z testów polegał na zweryfikowaniu, czy wytworzona aplikacja będzie funkcjonowała po połączeniu ze sobą poszczególnych komponentów, ale w odróżnieniu od testów integracyjnych, zamiast atrap klas komunikujących się z urządzeniami obliczeniowymi, wykorzystane będą te faktycznie wchodzące w interakcję z bibliotekami służącymi do zarządzania zużyciem energii, a agenty uruchomione będą na węzłach udostępnianych przez KASK:

- *apl11.eti.pg.gda.pl*, zwany dalej apl11;
- *apl12.eti.pg.gda.pl*, zwany dalej apl12.

Podczas realizacji tego testu wystąpił dość istotny problem, który należało rozwiązać: wyżej wymienione maszyny znajdują się za firewallem, a więc standardowe tunelowanie SSH[12] się nie powiodło. Rozwiązaniem tego problemu było utworzenie odwrotnego tunelu SSH z obu maszyn docelowych na maszynę z publicznym adresem IP, na której uruchomiony był serwer. Po wykonaniu tego kroku serwer mógł nawiązać komunikację z agentami i wszystko przebiegło bez dalszych trudności. Lista urządzeń dostępnych na każdym z węzłów została pobrana przez serwer i wyświetlona na interfejsie użytkownika.

Test ten został wykonany na samym początku, gdyż weryfikacja, czy aplikacja poprawnie uruchamia się na środowisku docelowym, była kluczowa dla przeprowadzania dalszych, bardziej zaawansowanych testów, które weryfikują bardziej złożone aspekty jej funkcjonowania.

6.2. Reagowanie na zmiany limitów zużycia energii dokonywane spoza aplikacji

Test zakończył się sukcesem, reakcja serwera na modyfikację limitu była dokładnie taka jak, opisana w poniższym scenariuszu.

1. Uruchomienie agenta na węźle apl11.
2. Dodanie węzła do listy węzłów za pośrednictwem UI.
3. Ustawienie limitu zużycia energii na pierwszej z kart Nvidia Tesla K20x na 200W:
   a. limit zostaje wprowadzony na UI i przekazany do serwera;
   b. serwer wysyła do agenta żądanie ustawienia limitu dla wybranego urządzenia;
   c. agent ustawia limit.
4. Zmiana limitu zużycia energii na 210W za pomocą polecenia *nvidia-smi*.
5. Serwer kontrolujący limity za pomocą okresowego odpytywania agenta o aktualnie obowiązujący limit zauważa niezgodność.
6. Serwer wysyła do agenta żądanie ustawienie ponownie poprawnego limitu (200W):
   a. agent ustawia limit.

[12] SSH (ang. *Secure Shell* – bezpieczna powłoka) – protokół umożliwiający nawiązywanie bezpiecznej, szyfrowanej komunikacji pomiędzy dwoma urządzeniami przez niezabezpieczoną sieć.

6.3. Kontrola zużycia energii na podstawie zdefiniowanych reguł

Test został wykonany z wykorzystaniem reguły pozwalającej na zdefiniowanie limitu zużycia energii dla danego przedziału czasowego, dokładny przebieg był następujący:

- Uruchomienie agenta na węźle apl11;
- Dodanie węzła do listy węzłów za pośrednictwem UI;
- Stworzenie nowej reguły o następujących parametrach:
  - 28.11.2015 od 17:04 do 17:06 – 200W;
  - 28.11.2015 od 17:07 do 17:08 – 220W.
- Zatwierdzenie nowej reguły.

Rezultat powyższych działań powinien być następujący (w nawiasach podana została wartość limitu zużycia energii, który powinien obowiązywać po wykonaniu danego kroku).

- Początkowo nie jest ustawiony limit zarządzania energią (225W).
- 17:04 serwer zauważa, że limit na urządzeniu nie zgadza się z oczekiwanym.
- Do agenta zostaje wysłane polecenie ustawienia nowego limitu na urządzeniu, agent wykonuje polecenie (200W).
- 17:06 serwer zauważa, że limit przestał obowiązywać, a więc zostaje wysłane polecenie usunięcia limitu (225W).
- 17:07 zaczyna obowiązywać druga część reguły, co zostaje zauważone przez serwer, który wysyła do agenta polecenie ustawienia limitu (220W).
- 17:08 limit przestaje obowiązywać, więc ponownie zostaje wysłane polecenie usunięcia limitu (225W).

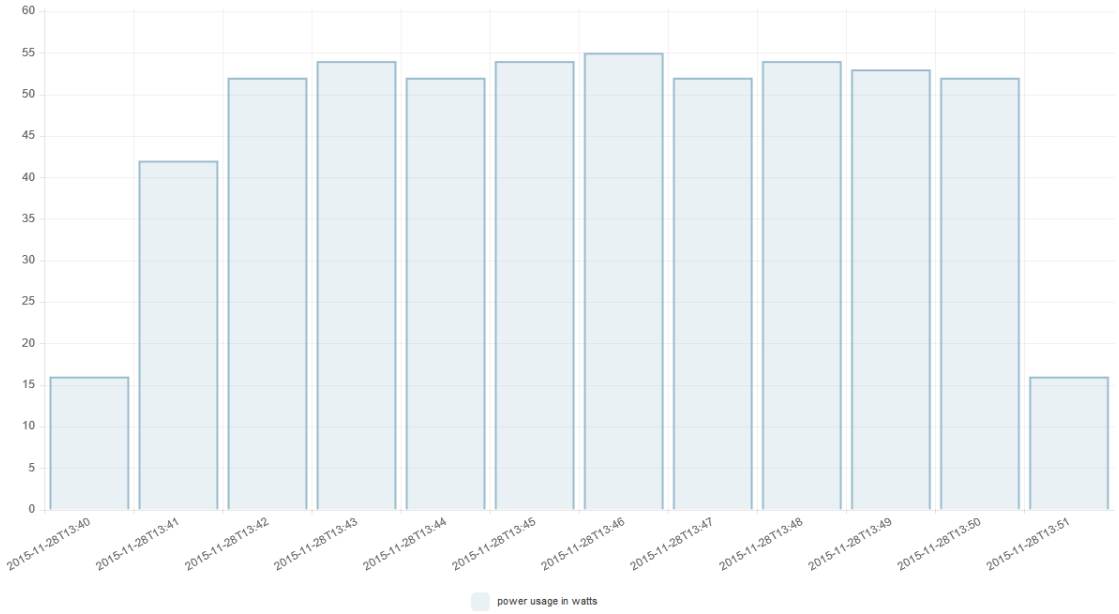
Wszystko przebiegło zgodnie z oczekiwaniami, co potwierdzają logi agenta przedstawione na listingu 6.1 (identyfikator urządzenia został skrócony dla poprawienia czytelności).

Listing 6.1. Zawartość logów agenta uruchomionego na maszynie apl11

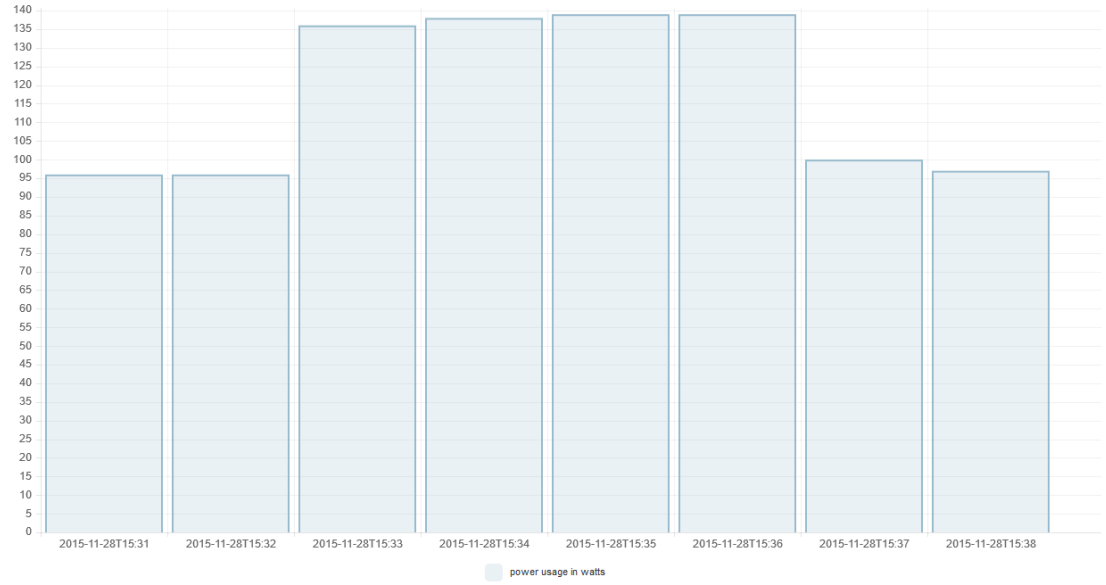
```
2015-11-28 16:52:02 INFO : Handling GET/node_information
2015-11-28 17:04:29 INFO : Handling GET/power_limit?NvidiaTesla,GPU-...
2015-11-28 17:04:29 INFO : Handling PUT/power_limit?NvidiaTesla,GPU-...=200
[STDOUT] Power limit for GPU 0000:05:00.0 was set to 200.00 W from 225.00 W.
2015-11-28 17:04:59 INFO : Handling GET/power_limit?NvidiaTesla,GPU-...
2015-11-28 17:05:29 INFO : Handling GET/power_limit?NvidiaTesla,GPU-...
2015-11-28 17:06:29 INFO : Handling GET/power_limit?NvidiaTesla,GPU-...
2015-11-28 17:06:29 INFO : Handling DELETE/power_limit?NvidiaTesla,GPU-...
[STDOUT] Power limit for GPU 0000:05:00.0 was set to 225.00 W from 200.00 W.
2015-11-28 17:07:29 INFO : Handling GET/power_limit?NvidiaTesla,GPU-...
2015-11-28 17:07:29 INFO : Handling PUT/power_limit?NvidiaTesla,GPU-...=220
[STDOUT] Power limit for GPU 0000:05:00.0 was set to 220.00 W from 225.00 W.
2015-11-28 17:07:59 INFO : Handling GET/power_limit?NvidiaTesla,GPU-...
2015-11-28 17:08:29 INFO : Handling GET/power_limit?NvidiaTesla,GPU-...
2015-11-28 17:08:29 INFO : Handling DELETE/power_limit?NvidiaTesla,GPU-...
[STDOUT] Power limit for GPU 0000:05:00.0 was set to 225.00 W from 220.00 W.
```

6.4. Zbieranie danych o ilości energii zużywanej podczas wykonywania obliczeń

Test polegał na uruchomieniu zbierania statystyk na urządzeniach Nvidia Tesla K20x i Intel Xeon Phi 5110P, a następnie obciążenia ich pewnymi obliczeniami, aby zweryfikować, czy podwyższone zużycie energii faktycznie zostanie zarejestrowane. Jak można zauważyć na rys. 6.1. i 6.2. przedstawiających zmianę zużycia energii w czasie, test przebiegł pomyślnie.



Rys. 6.1. Dane odczytane z urządzenia Nvidia Tesla K20x



Rys. 6.2. Dane odczytane z urządzenia Intel Xeon Phi 5110P

Podczas powyższego testu dokonano ciekawej obserwacji dotyczącej zużycia energii w stanie gotowości (czyli gdy urządzenie nie wykonuje żadnych obliczeń, ale jest okresowo odpytywane przez aplikację agenta – wymusza to pewną aktywność ze strony urządzenia), jak widać na urządzeniu Intel Xeon Phi jest ono kilkukrotnie wyższe. Podczas wykonywania testu zauważono również, że przy pierwszym odpytaniu urządzenia Intel Xeon Phi , jeżeli znajdowało się ono wcześniej w stanie bezczynności, raportowane zużycie energii wynosiło w przybliżeniu 39W, a więc było dużo niższe. Na tej podstawie można stwierdzić, że warto rozważyć wprowadzenie modyfikacji do agenta powodującej, że urządzenie pomiędzy kolejnymi wysyłanymi przez agenta komunikatami będzie mogło pozostawać w stanie bezczynności, co byłoby zachowaniem oczekiwanym w sytuacji, gdy nie są wykonywane żadne obliczenia.

Zużycia energii podczas wykonywania obliczeń nie można bezpośrednio porównać ze względu na różne problemy obliczeniowe, które były rozwiązywane przez każde z urządzeń. W przypadku kart Nvidia było to mnożenie macierzy, a koprocesor Intel Xeon Phi wykonywał całkowanie adaptacyjne. W ramach rozwinięcia pracy warto byłoby wykonać porównanie z wykorzystaniem tego samego problemu obliczeniowego zadanego obu urządzeniom do rozwiązania, np. mnożenia macierzy. Nie jest to jednak zagadnienie trywialne, gdyż należałoby tak dobrać parametry, żeby nie faworyzować żadnego z urządzeń. Co więcej, trzeba napisać dwie wersje kodu wybranego algorytmu, uwzględniając wszystkie szczegóły związane z architekturą danego układu tak, by w pełni wykorzystać jego możliwości. Wykracza to poza zakres niniejszej pracy, więc zdecydowano się na niewykonywanie takiego porównania.

7. ANALIZA MOŻLIWOŚCI ZARZĄDZANIA ZUŻYCIEM ENERGII NA POSZCZEGÓLNYCH TYPACH URZĄDZEŃ

W rozdziale drugim przedstawiono wybrane do porównania urządzenia, jednak to nie one są najistotniejszym elementem wpływającym na kształt niniejszej pracy. Są nim natomiast biblioteki udostępniane przez producentów tych urządzeń, służące do zarządzania zużyciem energii, które zostaną przeanalizowane w tym rozdziale:

- Nvidia Management Library (NVML) – dla kart Nvidia;
- Intel Manycore Platform Software Stack (MPSS) – dla koprocesorów Intel Xeon Phi;
- Intel Node Manager Programmer's Reference Kit (NMPRK) – dla procesorów Intel Xeon.

7.1. Podobieństwa i różnice architektoniczne

Tabela 7.1. Zestawienie bibliotek

Biblioteka	Sposób informowania o błędach w wywołaniu funkcji bibliotecznych	Sposób linkowania	Wartości graniczne dla limitu zużycia energii	Odczyt limitu zużycia energii	Ustawianie twardego limitu zużycia energii	Odczyt aktualnego zużycia energii
NVML	Zwracanie kodu błędu	Dynamiczne	Tak	Tak	Tak	Tak
MPSS	Zwracanie kodu błędu	Dynamiczne	Tak	Tak	Nie	Tak
NMPRK	Rzucenie wyjątku lub zwrócenie kodu błędu	Statyczne	Nie?	Tak?	?	Tak?

W przypadku NMPRK wartości podane w tabeli 7.1. są oznaczone znakiem zapytania, gdyż prawdopodobnie ze względu na brak pełnego wsparcia dla Intel Node Manager na maszynach testowych nie udało się uruchomić wywołań bibliotecznych odpowiedzialnych za odczyt tych wartości. Powiodło się jedynie odczytanie i to w bardzo ograniczonym zakresie informacji o samym węźle. Omówmy pokrótce wszystkie kategorie:

- sposób informowania o błędach w wywołaniu funkcji bibliotecznych – wygodniejsze i mniej problematyczne jest zwracanie kodu błędu, brak obsłużonego kodu błędu ma znacznie mniejsze konsekwencje dla działania aplikacji niż nieprzechwycony wyjątek;
- sposób linkowania – linkowanie dynamiczne ma zarówno zalety (zmniejszony rozmiar pliku wynikowego), jak i wady (wymagało wprowadzenia możliwości ładowania bibliotek w trakcie działania aplikacji), co omówiono w podrozdziale 4.4.3.;
- wartości graniczne dla limitu zużycia energii:
    - NVML podaje je w jawny sposób i umożliwia ich odczyt, są one różne dla różnych serii urządzeń;
    - MPSS nie umożliwia odczytu wartości granicznych, jednak w dokumentacji znajduje się zapis określający, z jakiego przedziału wartości są dopuszczalne, siłą rzeczy te same limity dotyczą różnych serii urządzeń;

    - NMPRK nie ma zapisu w dokumentacji określającego limity, nie ma też funkcji która pozwalałby na ich odczyt, weryfikacja w praktyce była niemożliwa ze względu na brak sprzętu umożliwiającego testy,
- odczyt limitu zużycia energii - każda z bibliotek, pomimo różnego rozumienia limitu, pozwala na jego odczyt;
- ustawianie twardego limitu zużycia energii:
    - NVML umożliwia ustawienie twardego limitu, który powoduje, że sterownik karty będzie stosował algorytm tak ograniczający zużycie energii, aby nie nastąpiło przekroczenie zadanego limitu,
    - MPSS nie operuje pojęciem twardego limitu, udostępnia dwa „miękkie limity", których przekroczenie powoduje podjęcie określonych akcji, mających na celu zmniejszenie ilości energii zużywanej przez koprocesor; w podrozdziale 4.4.4. przedstawiona została propozycja algorytmu mającego za zadanie przekształcenie limitów miękkich w twardy;
    - W przypadku NMPRK dokumentacja nie określa, czy możliwy do ustawienia limit jest twardy, czy miękki, nie można było tego również zweryfikować w praktyce;
- odczyt aktualnego zużycia energii – każda z bibliotek pozwala na odczyt aktualnego zużycia energii.

Zdecydowanie najistotniejszą z występujących pomiędzy poszczególnymi bibliotekami rozbieżności jest różne rozumienie przez nie limitu zużycia energii, co wymaga zastosowania odpowiednich algorytmów i rozwiązań po stronie agenta, tak aby ukryć tę niespójność przed serwerem. Niestety ze względu na ograniczenia sprzętowe i administracyjne (uprawnienia dla Xeon Phi) niemożliwa była praktyczna weryfikacja rozwiązania proponowanego dla kart Intel Xeon Phi, a w przypadku NMPRK jedyne, co udało się osiągnąć, to skuteczne odczytanie informacji o urządzeniu, bo platforma najprawdopodobniej nie wspierała zarządzania zużyciem energii. Mogło to również wynikać z błędu popełnionego podczas implementacji komunikacji z biblioteką NMPRK, jednak nie udało się w pełni zweryfikować żadnej z tych hipotez.

7.2. Łatwość korzystania z poszczególnych bibliotek

Z trzech przedstawionych bibliotek zdecydowanie wyróżnia się NVML. Dostępna dla niej jest kompletna, wyczerpująca i zrozumiała dokumentacja, której znalezienie nie stanowiło najmniejszego problemu. Niestety w przypadku MPSS i NMPRK dokumentacja jest raczej zdawkowa i odszukanie jej nie było tak proste, jak w wypadku NVML. Zdecydowanie najgorzej na tle innych bibliotek wypada NMPRK. Jej interfejs jest skomplikowany, nie posiada dobrej dokumentacji, a co więcej, stawia ona dość duże wymagania odnośnie wsparcia ze strony sprzętu.

W przypadku NVML i MPSS zależni jesteśmy jedynie bezpośrednio od wsparcia ze strony konkretnego urządzenia obliczeniowego, jeżeli chodzi o NMPRK – wsparcie musi być również zapewnione przez chipset płyty głównej i jej firmware, co stanowi pewne ograniczenie.

Analogiczna sytuacja ma miejsce w przypadku interfejsu udostępnianego przez biblioteki, jeżeli pominąć różnice w rozumieniu limitu zużycia energii, to NVML i MPSS są podobne i równie proste w obsłudze. Prawdopodobnie, gdyby dokumentacja NMPRK była wyższej jakości, to można by wysnuć o nim podobny wniosek, gdyż nie sprawia on wrażenia zanadto skomplikowanego. Brak dokumentacji wpływa jednak negatywnie na możliwość zrozumienia interfejsu, tak więc wnioskiem końcowym może być stwierdzenie, że NMPRK wypada gorzej na tle dwóch pozostałych bibliotek.

Zapewnienie w MPSS jasno zdefiniowanych wartości granicznych limitu zużycia energii możliwych do oczytania za pomocą konkretnego wywołania funkcji bibliotecznej byłoby bardzo korzystne dla tego rozwiązania. Podobnie umożliwienie ustawiania twardego limitu z odpowiednim algorytmem kontrolującym, aby nie został on przekroczony, tak jak ma to miejsce w NVML. W przypadku NMPRK problemem był też brak jasnej definicji urządzenia. Autorzy niniejszej pracy do tej pory nie są pewni, co do jego znaczenia i zakładają, że odnosi się ono do modułu Node Managera, gdyż znalezienie informacji potwierdzającej lub obalającej tę tezę było niemożliwe, a w świetle braku konkretnego zapisu w dokumentacji, pozostają jedynie przypuszczenia. Urządzeniem może być równie dobrze konkretny procesor, jednak, co w wypadku, gdy na danym węźle znajdują się dwa lub więcej procesorów, co nie jest sytuacją rzadką, pytanie to pozostaje otwarte. Reasumując, wnioski płynące z implementowania aplikacji z wykorzystaniem trzech wyżej wymienionych bibliotek są następujące:

- można zauważyć pewne podobieństwa pomiędzy poszczególnymi rozwiązaniami;
- różnice są na tyle istotne, że dobrze przemyślana warstwa abstrakcji izolująca warstwy wyższe od specyfiki danej biblioteki jest konieczna, a jej stworzenie i wdrożenie nie jest zadaniem trywialnym;
- biblioteki znacznie różnią się pomiędzy sobą; jeżeli chodzi o jakość dokumentacji;
- występują znaczące różnice w wymaganiach sprzętowych;
- szeregując od najłatwiejszych w zastosowaniu do najtrudniejszych: NVML, MPSS, NMPRK.

7.3. API bibliotek

Tabela 7.2. Zestawienie wywołań bibliotecznych

Zastosowanie	NVML	MPSS	NMPRK
Inicjalizacja biblioteki	*nvmlInit*	Brak.	*swSubSystemSetup* *initSystemForLocal*
Zakończenie korzystania z biblioteki	*nvmlShutdown*	Brak.	*swSubSystemSetup*
Ustalanie unikalnego identyfikatora urządzenia	*nvmlDeviceGetIndex* *nvmlDeviceGetSerial* *nvmlDeviceGetUUID* *nvmlDeviceGetPciInfo*	*mic_get_uuid*	*getDeviceId*
Pobieranie i zwalnianie uchwytu do urządzenia	*nvmlDeviceGetCount* *nvmlDeviceGetHandleByIndex* *nvmlDeviceGetHandleBySerial* *nvmlDeviceGetHandleByUUID* *nvmlDeviceGetHandleByPciBusId*	*mic_get_devices* *mic_get_ndevices* *mic_get_device_at_index* *mic_open_device* *mic_close_device*	Stworzenie obiektu *device* z typem ustawionym na *device_nm* i adresem na *local.* *connectDevice* *disconnectDevice*
Weryfikacja czy urządzenie wspiera zarządzanie zużyciem energii	*nvmlDeviceGetPowerManagementMode*	Każde z urządzeń Intel Xeon Phi wspiera zarządzanie zużyciem energii.	*getCapabilities*
Odczyt aktualnego zużycia energii	*nvmlDeviceGetPowerUsage*	*mic_get_power_utilization_info* *mic_get_inst_power_readings*	*getSample* *resetStatistics*
Odczyt górnego i dolnego ograniczenia na limit zużycia energii	*nvmlDeviceGetPowerManagementLimitConstraints*	Brak funkcji pozwalającej na odczyt, ale dokumentacja określa zakres wartości jako od 50 do 400.	Nie udało się ustalić ze względu na niemożność przeprowadzenia testów praktycznych.
Odczyt limitu zużycia energii	*nvmlDeviceGetPowerManagementLimit*	*mic_get_power_limit* *mic_get_time_window0* *mic_get_time_window1* *mic_get_power_lmrk* *mic_get_power_hmrk*	*getPolicy*
Ustawianie limitu zużycia energii	*nvmlDeviceSetPowerManagementLimit*	*mic_set_power_limit0* *mic_set_power_limit1*	*setPolicy*
Inne warte uwagi funkcje	*nvmlSystemGetNVMLVersion* *nvmlSystemGetDriverVersion* *nvmlDeviceGetPersistenceMode* *nvmlDeviceSetPersistenceMode* *nvmlDeviceGetName*	*mic_get_version_info* *mic_get_flash_version* *mic_get_uos_version*	*getNMVersion*

Tabela 7.2. nie zawiera opisu argumentów poszczególnych funkcji oraz przykładów ich zastosowania. Informacje te są dostępne w dokumentacji bibliotek [24-26], przykłady użycia znajdują się również bezpośrednio w kodzie źródłowym agenta. W przypadku MPSS dokumentacja wywołań bibliotecznych powinna być dostępna po wydaniu polecenia *man libmicmgmt* – po wcześniejszym zainstalowaniu MPSS w systemie Linux.

8. OPIS REALIZACJI PROJEKTU

8.1. Opis organizacji i podziału pracy

Na początku planowania projektu podzieliliśmy się w zespole obowiązkami. Jako że przyjęliśmy, iż nasz projekt będzie się składał z trzech modułów, naturalnym podziałem stał się podział w postaci jednego modułu na jedną osobę: Tomasz zajął się zaprojektowaniem i implementacją agenta, Andrzej – serwera, a Bartosz – interfejsu użytkownika. Dodatkowo Andrzej i Bartosz wzajemnie wprowadzali zmiany do swoich modułów, z racji znajomości technologii i odmiennego spojrzenia na temat, co niejednokrotnie przyczyniło się do poprawy działania lub ulepszenia funkcji modułów.

Poza podziałem obowiązków przy tworzeniu samego projektu, rozdzieliliśmy między sobą dodatkowe role, okołoprojektowe, niezbędne do sprawnego wytwarzania modułów.

Tomasz zajął się zaprojektowaniem głównej architektury systemu, przygotował wstępne diagramy klas, zdefiniował początkowe zadania, a także zorganizował uzyskanie uprawnień dostępów do zasobów katedralnych (maszyny apl09-12), niezbędnych do wykonania testów na fizycznych urządzeniach, znajdujących się w laboratorium.

Andrzej podjął się skonfigurowania i utrzymywania środowiska programistycznego – zarządzania maszyną wirtualną, na której znajdowało się to środowisko, zainstalowania programów Quickbuild (do budowania projektów) i Gerrit (do zarządzania repozytorium), skonfigurowania bazy MySQL i tworzenia kopii zapasowych.

Bartosz zajął się stworzeniem maszyny wirtualnej ze środowiskiem przedprodukcyjnym. Służyła ona do testowania modułów poza lokalnym komputerem, dawała możliwość pokazywania efektów pracy na zewnątrz. Utrzymywał również domenę, pod którą były dostępne wszystkie serwisy deweloperskie i przedprodukcyjna maszyna.

8.2. Opis środowiska deweloperskiego

W celu ułatwienia współpracy w zespole podczas wytwarzania projektu oraz aby zagwarantować wysoką jakość wytwarzanego oprogramowania, utworzone zostało współdzielone środowisko wyposażone w narzędzia do wspomagania kooperacji w zespole oraz pozwalające na testowanie aplikacji. Również oprogramowanie używane przez członków zespołu na ich komputerach zostało ustalone w sposób jednolity.

Środowisko zostało stworzone na dwóch maszynach wirtualnych typu VPS, dostarczanych przez VPSDime – oferta „High RAM VPS”. Każda z nich jest wyposażona w 6GB pamięci RAM, współdzielony dostęp do czterech rdzeni procesora Intel Xeon E5-2630 i 30GB przestrzeni na dysku SSD. Duża ilość pamięci była konieczna, ponieważ jedna z maszyn miała za zadanie nie tylko uruchamiać potrzebne narzędzia, ale również budować wytwarzane aplikacje. Zwłaszcza kompilacja agenta jest procesem mającym duże zapotrzebowanie na pamięć i moc obliczeniową procesora. Oczywistym jest więc, że większość dostępnych za darmo dla studentów rozwiązań, wyposażonych co najwyżej w 512MB pamięci operacyjnej, nie nadawała się do tego zastosowania. Obie maszyny działały pod kontrolą systemu operacyjnego

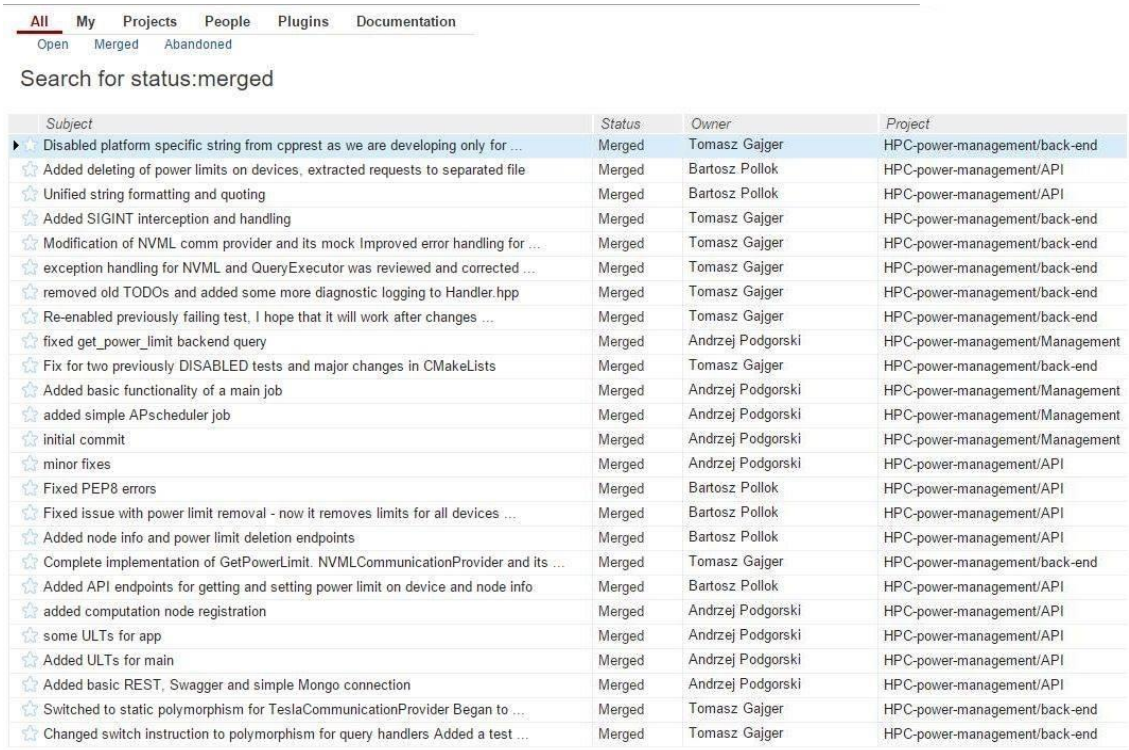
Ubuntu 14.04 LTS, a wbudowane programy były na bieżąco automatycznie aktualizowane przy użyciu APT.

Jedna z maszyn służyła jako środowisko przedprodukcyjne. Uruchamiane były w nim na stałe kolejne wersje wytwarzanych aplikacji w celu ułatwienia programowania pozostałych modułów tak, aby interfejs między nimi był zgodny. Uruchomiona była tam także instancja bazy danych MongoDB. Środowisko używane było również do prostego testowania integracji systemu oraz działania jego jako całości, a także do prezentowania jego działania na zewnątrz.

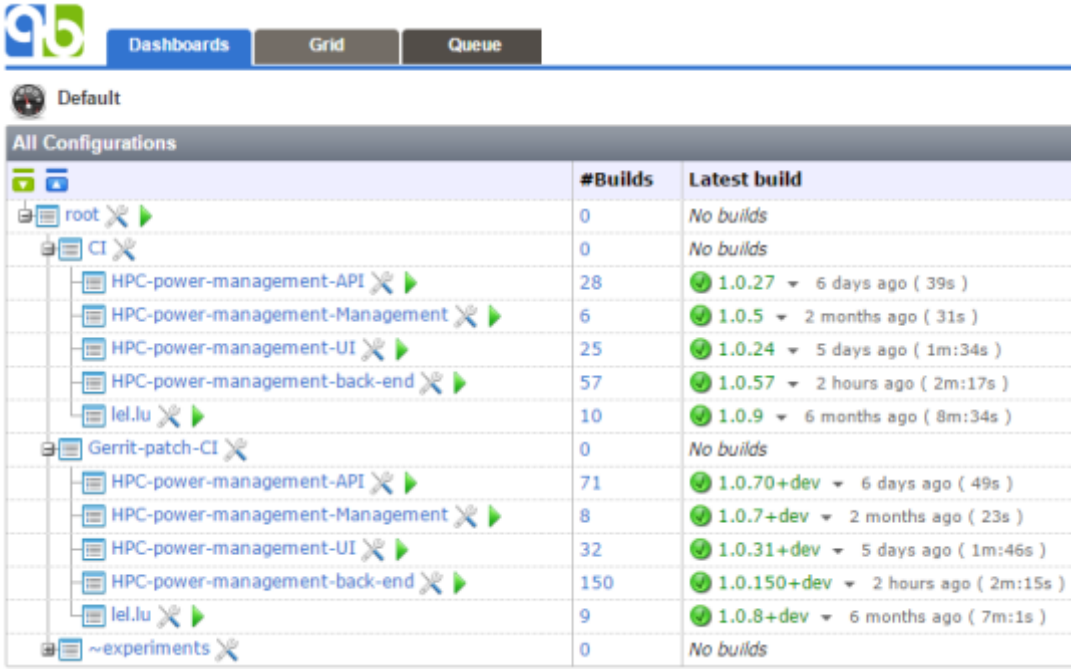
Druga natomiast była środowiskiem, na którym działały dwa programy: QuickBuild oraz Gerrit, a także baza danych MySQL, używana przez pierwszy z nich. Jak wspomniane zostało wyżej, maszyna ta zajmowała się również budowaniem wytwarzanych aplikacji. Dodatkowo wykonywała ona statyczna analizę kodu, testy jednostkowe oraz gromadziła i udostępniała do pobrania wytworzone artefakty. Proces wytwarzania oprogramowania, jego testowania oraz ciągłej integracji opisany jest w dalszej części podrozdziału. Codziennie tworzone były kopie zapasowe danych z wymienionych aplikacji. Przenoszone były one na usługę Dropbox, a stamtąd dystrybuowane dodatkowo na komputery jednego z członków zespołu.

Gerrit jest tworzonym przez Google darmowym narzędziem do zarządzania repozytoriami Git oraz inspekcji kodu. Posiada ono interfejs graficzny w postaci strony WWW (widoczny na rys. 8.1.). W wygodny sposób pokazuje ono historię zmian oraz pozwala na łatwe dokonywanie inspekcji, dzięki wygodnemu ekranowi porównania wersji plików. Daje możliwość zapobieżenia dołączeniu kodu do gałęzi głównej bez pozytywnej oceny innych członków zespołu. Pozwala również na automatyczną weryfikację, np. testy jednostkowe, przez zewnętrzne aplikacje dzięki udostępnianemu dla nich API [Milanesio L., 2013].

QuickBuild jest komercyjnym produktem firmy PMEase służącym do budowania, testowania oraz automatycznego wdrażania aplikacji. Wyposażony jest w interfejs w postaci strony WWW widoczny na rys. 8.2. Posiada bardzo zaawansowane możliwości konfiguracji, co pozwala używać go również dla bardzo dużych systemów. Jest z powodzeniem używany przez wiele znanych na świecie firm. Posiada wsparcie dla wielu systemów kontroli wersji, m.in. Git, Subversion czy Perforce. Współpracuje również z Gerrit – pozwala na pobieranie zmian przekazanych przez programistę do oceny oraz zwrócenie do Gerrit wyniku wcześniej zdefiniowanych operacji, najczęściej budowania, statycznej analizy kodu i testów jednostkowych. Posiada architekturę opartą o serwer i agenty budujące, co jest przydatne dla dużych projektów. Dla potrzeb projektu inżynierskiego wystarczająca byłą darmowa wersja „*community*”, pozwalająca na zdefiniowanie do 16 konfiguracji uruchomiona bez osobnych agentów budujących.



Rys. 8.1. Zrzut ekranu z interfejsu programu Gerrit



Rys. 8.2. Interfejs programu QuickBuild

Pozostałe narzędzia deweloperskie używane w zespole to zintegrowane środowiska programistyczne.

- QT Creator – do tworzenia kodu agenta w C++. Agent na tę chwilę wspiera jedynie Linuksa, logicznym krokiem było więc użycie środowiska działającego w tym systemie operacyjnym.
- PyCharm dla języka Python – do tworzenia kodu serwera. Bardzo rozbudowane i popularne środowisko do Pythona. Działa zarówno pod kontrolą Windowsa, jak i Linuksa. Pozwala na zdalne uruchamianie i debugowanie tworzonej aplikacji na innym systemie, przykładowo na programowanie przy użyciu systemu Windows, a uruchamianie na Linuksie zainstalowanym na innej maszynie. Wersja Professional dla studentów dostępna jest za darmo.
- Dla kodu UI, wytwarzanego w HTML i JavaScripcie przez niektórych członków zespołu wykorzystywany był również PyCharm. Inne stosowane narzędzie to Atom. Do debugowania kodu JavaScript używany był panel narzędzi deweloperskich przeglądarki Google Chrome.
- Dodatkowo konieczne było używanie klientów Git oraz skryptu git-review w celu publikowania zmian.

Proces publikowania zmian od momentu napisania kodu i podjęcia przez programistę decyzji o jego publikacji przedstawiał się następująco.

1. Utworzenie commita, najlepiej bazującego na głowie gałęzi głównej, na lokalnej gałęzi repozytorium Git.
2. Uruchomienie skryptu git-review w celu opublikowania zmian w Gerrit.
3. QuickBuild co ustalony czas (około 30 sekund) wykonuje zapytanie do Gerrit o nowe zmiany w danym repozytorium. Jeśli zmiany się pojawiły, pobiera je.
4. Następnie budowana jest aplikacja i wykonywane są testy. Dzieje się to w różny sposób w zależności od modułu, opisane jest to w rozdziale 5. oraz podrozdziale 8.3.
5. QuickBuild wysyła do Gerrit informację o statusie weryfikacji. Bez pozytywnej weryfikacji nie ma możliwości włączenia kodu do głównej gałęzi.
6. Jeśli weryfikacja jest negatywna, programista wprowadza odpowiednie poprawki i powtarza proces od punktu 2.
7. Inny członek zespołu wykonuje inspekcję kodu, sugeruje ewentualne zmiany.
8. Jeśli inspekcja jest negatywna, programista wprowadza odpowiednie poprawki i powtarza proces od punktu 2.
9. Po pozytywnej inspekcji, dowolny członek zespołu klika przycisk „Submit”, który powoduje dołączenie zmiany do gałęzi głównej repozytorium.
10. QuickBuild pobiera zmiany z gałęzi głównej, wykonuje testy, aby upewnić się, że aplikacja nie ucierpiała w wyniku scalania z kodem gałęzi głównej, buduje aplikację w konfiguracji gotowej do wydania i publikuje zbudowane artefakty. Są one dostępne do pobrania z poziomu interfejsu.

8.3. Metodologia testowania agenta

8.3.1. Testy jednostkowe

W trakcie implementowania poprawność działania poszczególnych elementów aplikacji była weryfikowana na bieżąco za pomocą testów jednostkowych. Jest ich 60 i pokrywają wszystkie istotniejsze fragmenty kodu. Były one pisane z wykorzystaniem biblioteki googletest, a w celu osiągnięcia możliwie jak największej separacji testowanych elementów stosowano atrapy poszczególnych obiektów, do których tworzenia wykorzystano bibliotekę googlemock. Podejście to sprawdziło się doskonale i pozwoliło wykryć wiele błędów wprowadzonych nieumyślnie podczas dokonywania zmian w kodzie, a możliwość odpalania testów bezpośrednio z poziomu IDE (QTCreator) i bardzo krótki czas ich trwania (rzędu kilkuset milisekund), przełożyły się na ich bardzo częste uruchamianie, w praktyce po każdej zmianie.

8.3.2. Testy funkcjonalne

Agent udostępnia interfejs w postaci listy zasobów, do których można odwoływać się wysyłając odpowiednie żądania protokołu HTTP, został on dokładnie opisany w podrozdziale 4.3. Dzięki temu można w bardzo łatwy sposób, bez konieczności implementacji specjalizowanego klienta, testować jego działanie. Oczywiście mówimy tu o sytuacji, gdy chcemy przeprowadzać izolowane testy samego agenta, gdy nie jest on podłączony do serwera. Do testów wykorzystano program *curl*, w ten sposób na bieżąco w trakcie wytwarzania weryfikowano, czy agent w poprawny sposób odpowiada na wysłane do niego żądania. Testy te były przeprowadzane zarówno dla wersji kompilowanych z wykorzystaniem atrap, których idea zostanie wyjaśniona dalej, jak i rzeczywistych klas komunikujących się z bibliotekami do zarządzania energią. Testy te były przeprowadzane ręcznie, co było rozwiązaniem wygodniejszym ze względu na dość duża zmienność aplikacji w trakcie wytwarzania, ale dobrym pomysłem byłaby ich automatyzacja w postaci skryptu dla interpretera *Bash*.

8.3.3. Atrapy

Możliwość uruchamiania i testowania działania agenta jest mocno ograniczona koniecznością interakcji z bibliotekami służącymi do zarządzania energią. Aby nie stanowiło to przeszkody w wytwarzaniu, każda z klas *CommunicationProvider* posiada swoją atrapę, która w pewnym ograniczonym zakresie symuluje jej działania i uniezależnia całą aplikację od rzeczywistych bibliotek. Rozwiązanie zostało zaprojektowane tak, aby nie wymagało żadnej ingerencji w kod aplikacji w razie chęci przełączenia się z rzeczywistych klas na atrapy. To, której klasy chcemy używać, jest określane przez opcję *USE_MOCKS* przekazywaną do CMake podczas generowania plików systemu budowania. Przełączanie się pomiędzy klasami zostało zrealizowane z wykorzystaniem statycznego polimorfizmu opartego o szablony, pokażmy to na przykładzie klasy NvidiaTeslaDevice, mechanizm działania jest taki sam dla pozostałych klas. Klasa ta jest szablonem z parametrem będącym klasą odpowiedzialną za komunikację, w zależności, czy w miejsce tego parametru podana zostanie klasa

NVMLCommunicationProvider czy *MockNVMLCommunicationProvider,* będziemy posługiwali się klasą rzeczywistą lub jej atrapą. O tym, która klasa ma zostać przekazana jako parametr, decyduje makro preprocesora *USE_COMM_PROVIDER_MOCKS*, które jest ustawiane przez CMake w zależności od wartości wspomnianego wcześniej parametru *USE_MOCKS*.

8.4. Metodologia testowania serwera

8.4.1. Statyczna analiza kodu

W celu zadbania o to, aby kod serwera był jednolity niezależnie od tego, kto go pisał, zgodny z standardami i zaleceniami oraz pozbawiony tzw. „brzydkich zapachów", zastosowano narzędzia do statycznej analizy kodu. Ich użycie było częścią procesu CI (ang. *Continuous Integration* – ciągła integracja) opisanego w podrozdziale 8.2.

Pierwsze narzędzie to PEP8. Służy on do sprawdzania zgodności formatowania z zaleceniem PEP8 (ang. *Python Enhancement Proposal* - Propozycje poprawek Pythona). Włącza się w to na przykład stosowanie czterech spacji jako indentacji nowego wiersza na końcu pliku czy dwóch wierszy odstępu między definicjami klas.

Kolejne narzędzie to PyFlakes. Sprawdza ono drzewo syntaktyczne każdego pliku źródłowego oddzielnie w poszukiwaniu błędów. Pozwala wykryć niektóre nieprawidłowe użycia zmiennych, na przykład próbę wywołania nieistniejącej metody obiektu. Niestety ze względu na dynamiczne typowanie, w Pythonie jest to zadanie trudne i nie zawsze się udaje. Narzędzie informuje o nieprawidłowościach tylko wtedy, gdy jest pewne, że w kodzie jest błąd – praktycznie brak fałszywych alarmów. Z tego względu nie jest konieczna zmiana konfiguracji.

Ostatnim wykorzystanym narzędziem tego typu jest PyLint. Jest to bardzo popularne, rozbudowane narzędzie służące do statycznej analizy kodu w Pythonie. Analizuje on projekt jako całość. Sprawdza między innymi, czy nazwy zmiennych i metod nie są zbyt długie, czy zaimportowane moduły są używane, czy klasy nie są zbyt duże lub zbyt małe. Poszukuje także duplikatów kodu, sugerując wyodrębnienie go do osobnej funkcji. Narzędzie to dość często generuje fałszywe alarmy, jednak można je konfigurować.

8.4.2. Testy jednostkowe

Testy jednostkowe obu aplikacji serwera przeprowadzane zostały przy użyciu modułu *pytest*. Wraz z wtyczką *pytest-cov* pozwala on na wygenerowanie raportu HTML obrazującego pokrycie kodu testami (rys. 8.3. i 8.4.). W chwili obecnej, ze względu na małą ilość czasu, poziom pokrycia kodu testami jednostkowymi nie jest zadowalający. Przed wprowadzeniem projektu do użytku należałoby napisać dodatkowe testy. Nie jest to trudne zadanie, bo testy jednostkowe w Pythonie pisze się łatwo, ze względu na obecny w bibliotece standardowej moduł *mock*, który służy do tworzenia atrap modułów, klas, i funkcji [Sale D., 2014].

Module	statements	missing	excluded	branches	partial	coverage
hpcpm/__init__.py	0	0	0	0	0	100%
hpcpm/api/__init__.py	2	0	0	0	0	100%
hpcpm/api/app.py	32	9	0	0	0	72%
hpcpm/api/helpers/__init__.py	0	0	0	0	0	100%
hpcpm/api/helpers/constants.py	29	0	0	0	0	100%
hpcpm/api/helpers/database.py	78	43	0	10	0	40%
hpcpm/api/helpers/requests.py	10	6	0	0	0	40%
hpcpm/api/helpers/utils.py	28	21	0	7	0	20%
hpcpm/api/main.py	55	1	0	2	1	96%
hpcpm/api/resources/ApiSpec.py	25	11	0	0	0	56%
hpcpm/api/resources/__init__.py	0	0	0	0	0	100%
hpcpm/api/resources/endpoints/RuleTypes.py	6	1	0	0	0	83%
hpcpm/api/resources/endpoints/Status.py	5	1	0	0	0	80%
hpcpm/api/resources/endpoints/__init__.py	0	0	0	0	0	100%
hpcpm/api/resources/endpoints/nodes/__init__.py	0	0	0	0	0	100%
hpcpm/api/resources/endpoints/nodes/computation_node/AllComputationNodes.py	24	15	0	6	0	30%
hpcpm/api/resources/endpoints/nodes/computation_node/ComputationNode.py	59	46	0	12	0	18%
hpcpm/api/resources/endpoints/nodes/computation_node/PowerLimit.py	53	41	0	11	0	19%
hpcpm/api/resources/endpoints/nodes/computation_node/Rule.py	32	22	0	10	0	24%

Rys. 8.3. Zbiorczy raport obrazujący pokrycie kodu testami jednostkowymi

```
15 def initialize(config):
16     database.configure(config["database"])
17     api_spec = ApiSpec(config["host"])
18     flask_app.register_blueprint(api_spec.blueprint, url_prefix='/api/hpcpm')
19
20
21 def run(port):
22     http_server = HTTPServer(WSGIContainer(flask_app))
23     http_server.bind(port, "0.0.0.0")
24     log.debug("HTTP server starting")
25     http_server.start(1)
26     IOLoop.instance().start()
27
28
29 @flask_app.before_request
30 def start_request(*_):
31     request.real_ip = request.headers.get('X-Real-Ip', request.remote_addr)
32     request.start_time = time.time()
33
34     log.info("REQUEST STARTED: %s %s %s", request.real_ip, request.method, request.url)
35
```

Rys. 8.4. Szczegółowy raport obrazujący pokrycie pojedynczego pliku testami jednostkowymi

8.4.3. Testy funkcjonalne

Ze względu na to, że funkcje agenta, których używa serwer, powstawały szybciej niż zależny od nich kod serwera, testy funkcjonalne mogły być prowadzone w integracji z agentem. Do tego celu korzystano z zestawionego środowiska przedprodukcyjnego. Istniała możliwość testowania całego środowiska przedprodukcyjnego lub użycia tylko agenta działającego na nim w integracji z serwerem tworzonym na lokalnej maszynie. Pozwalało to na prowadzenie testów podczas tworzenia aplikacji. Używaną bazą danych zawsze była instancja w środowisku przedprodukcyjnym. Ze względu na małą liczebność zespołu nie zachodziła obawa o wzajemne przeszkadzanie sobie w zakresie korzystania z bazy danych.

Testy przeprowadzano ręcznie, wywołując odpowiednie zapytania HTTP przy użyciu interfejsu *flask-restful-swagger* i sprawdzając otrzymane odpowiedzi oraz logi z aplikacji. Wydaje się, że nie zachodzi potrzeba tworzenia zautomatyzowanych testów funkcjonalnych samego serwera. Testy jednostkowe w przypadku REST API pozwalają sprawdzić właściwie całą logikę, a testy integracji systemu jako całości lepiej przeprowadzać z poziomu interfejsu użytkownika, co opisane jest w podrozdziale 8.6.

8.5. Metodologia testowania UI

8.5.1. Statyczna analiza kodu

Kod napisany w języku JavaScript był poddawany analizie statycznej za pomocą narzędzia JSHint w domyślnej konfiguracji, w celu wykrycia ewentualnych błędów w samym kodzie, a także aby ujednolicić go, gdyż JavaScript w wielu sytuacjach przyzwala na pewną swobodę w pisaniu kodu, np. pomijanie średników.

8.5.2. Testy jednostkowe

Poprawność implementowanych funkcji interfejsu użytkownika była sprawdzana przez uruchamianie testów jednostkowych podczas budowania projektu w serwisie QuickBuild, a także lokalnie na żądanie. Została wykorzystana biblioteka Jasmine, umożliwiająca pisanie testów sterowanych zachowaniem, a także narzędzie Karma, uruchamiające serwer i środowisko, tj. przeglądarkę internetową lub program udający przeglądarkę bez wyświetlania jej okna (np. PhantomJS), a następnie wykonujące testy w przygotowanym przez siebie otoczeniu. Testy wykorzystywały licznie atrapy dostarczane przez producenta biblioteki AngularJS, tj. Angular Mocks, gdyż były one niezbędne do symulowania zachowań elementów graficznych tudzież komunikacji HTTP. Liczba testów ze względu na presję czasową jest niewielka, lecz przewiduje się pełne pokrycie napisanego kodu [Ragonha P., 2015].

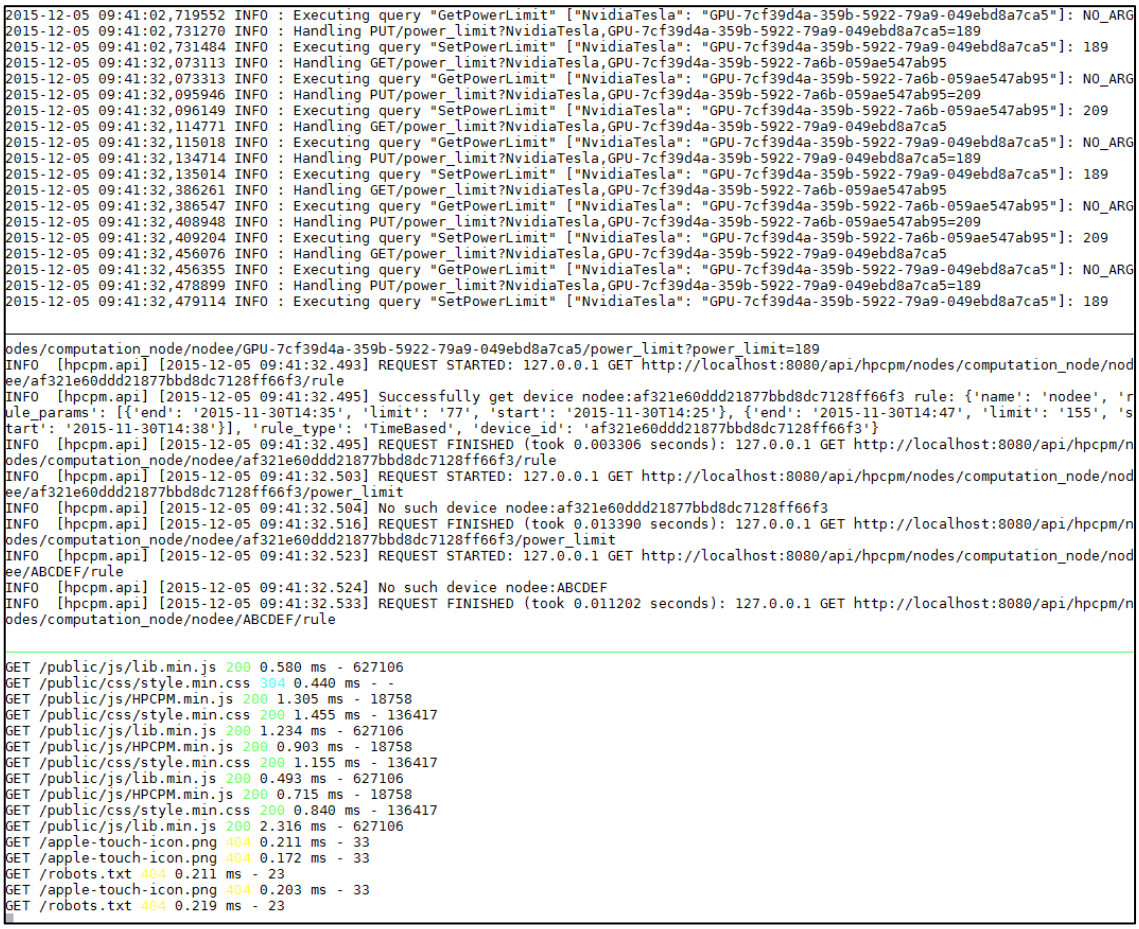
8.5.3. Testy funkcjonalne

Testowanie funkcjonalne zostało przeprowadzone przez uruchomienie interfejsu użytkownika lokalnie lub na maszynie przedprodukcyjnej i sprawdzenie działania otwartej strony aplikacji w przeglądarce internetowej. Następnie wszystkie dostarczone funkcje zostawały ręcznie przetestowane przez interakcję z nimi z poziomu strony w przeglądarce i wykonywanie odpowiednich scenariuszów.

8.6. Testy integracyjne

Testy integracyjne były przeprowadzane poprzez skonfigurowanie i uruchomienie wszystkich modułów – agenta, serwera i interfejsu użytkownika w środowisku przedprodukcyjnym. W początkowych etapach prac było to zadanie czasochłonne, jednak później proces instalacji i konfiguracji został opanowany w dobrym stopniu oraz zwiększono jego automatyzację. Testowana była głównie komunikacja pomiędzy modułami: pomiędzy agentem i serwerem, a także pomiędzy interfejsem użytkownika i serwerem. Testowanie agenta

opierało się na wykorzystaniu atrap, gdyż maszyna, na której był on uruchamiany, nie posiadała żadnego ze wspieranych urządzeń. Testy przeprowadzane były ręcznie, wykorzystując zarówno interfejs użytkownika, do kierowania określonych zapytań przy użyciu jego funkcjonalności, jak również interfejs *flask-restful-swagger*, dostarczany przez serwer, gdy rozwijane były nowe funkcje interfejsu. Wyniki wykonanych akcji były analizowane poprzez czytanie logów serwera i agenta (rys. 8.5) oraz sprawdzanie odpowiednich pól w bazie danych.



Rys. 8.5. Zrzut ekranu z konsoli maszyny przedprodukcyjnej.
Uruchomione od góry: agent, serwer, interfejs użytkownika

8.7. Testy end to end

Dla projektu utworzono również automatyczne testy *end to end* (ang. całościowe, od końca do końca). W tym celu skorzystano z platformy Protractor, która służy do testowania serwisów internetowych tak, jak zrobiłby to człowiek. Scenariusze testowe dla niej pisane są jako kod JS, w którym dostępne są akcje takie, jak odnajdywanie elementów wg. atrybutów czy zawartości, najeżdżanie myszą i klikanie, a także pobieranie zawartości i atrybutów w celu porównania z wzorcowymi. Istnieje również możliwość zlecenia Protractorowi wykonania zrzutu ekranu po każdym przypadku testowym (wykonują się one po kolei). Testy takie wykonują się przy użyciu wskazanej przeglądarki (na przykład Chrome, Firefox lub PhantomJS). Korzystając z opisanych technik utworzono scenariusz testowy pokrywający większość funkcji systemu w tym:

1. wyświetlenie strony głównej z listą węzłów;
2. dodanie nowego węzła;
3. wyświetlenie ekranu szczegółów węzła;
4. ustawienie interwału pobierania statystyk, a następnie sprawdzenie poprawności ich wyświetlania;
5. ustawienie reguły zużycia energii.

8.8. Problemy

Kompilacja programu na maszynach udostępnianych przez KASK okazała się dość problematyczna z dwóch względów. Po pierwsze zainstalowane tam gcc w wersji 4.4 nie wspiera C++11. Po drugie – zainstalowane tam środowisko jest dość stare, glibc w wersji 2.12 zniweczyło próbę uruchomienia tam aplikacji skompilowanej na innym systemie. Postanowiono więc umieścić tam własne gcc wraz z niezbędnymi bibliotekami. Pierwszym krokiem była kompilacja gcc, która przebiegła bez większych problemów, a następnie skompilowano Boost, który również wymagał nowszej niż dostępna tam wersji. Ostatnim krokiem była kompilacja C++ REST SDK, która sprawiła drobne problemy, gdyż trzeba było dokonać ręcznej edycji konfiguracji CMake dla tej biblioteki. Zwieńczeniem całego procesu było zmodyfikowanie konfiguracji CMake agenta tak, aby korzystał on z nowej wersji gcc i nowych bibliotek. Po wykonaniu tych wszystkich operacji kompilacja i uruchomienie aplikacji przebiegły pomyślnie.

9. PODSUMOWANIE

9.1. Zdobyte doświadczenie

Tomasz Gajger podczas realizacji niniejszej pracy zapoznał się z podstawami obsługi CMake, nauczył się wykonywać dynamiczne ładowanie bibliotek z poziomu kodu C++ na systemie Linux, poznał podstawy architektury REST, zdobył doświadczenie w korzystaniu z bibliotek: NVML, MPSS, NMPRK i CPPREST, odświeżył informacje związane z GoogleTest oraz udoskonalił umiejętność projektowania architektury aplikacji z wykorzystaniem wzorców projektowych. Cennym doświadczeniem było dla niego również kompilowanie gcc wraz z bibliotekami w docelowym środowisku, w którym miał działać system.

Andrzej Podgórski rozwinął swoje umiejętności w zakresie wytwarzania aplikacji internetowych opartych o REST w języku Python. Poznał tajniki wytwarzania aplikacji od podstaw na platformie AngularJS. Zapoznał się ze sposobem zarządzania zużyciem energii na obsługiwanych w systemie urządzeniach. Nauczył się też, w jaki sposób konfigurować środowisko deweloperskie. Zebrał dodatkowe doświadczenie dotyczące współpracy w zespole.

Bartosz Pollok nabył umiejętności tworzenia aplikacji internetowych opartych o język JavaScript i różnorakie napisane w nim biblioteki, jak np. AngularJS i jQuery, a także wykorzystywania różnych technik tworzenia stron internetowych w HTML i Bootstrap. Zapoznał się z wytwarzaniem aplikacji serwerowych przy użyciu platformy Node.js. Nauczył się automatyzowania testowania aplikacji i uruchamiania ich w środowisku przedprodukcyjnym. Nabył wiedzę z zakresu tworzenia aplikacji opartych o architekturę REST w języku Python.

9.2. W jakim stopniu cel pracy został zrealizowany

Głównym celem pracy było stworzenie aplikacji pozwalającej na zarządzanie zużyciem energii w sposób jednolity na różnych rodzajach urządzeń obliczeniowych. Cel ten został w dużej mierze zrealizowany. Drobną przesadą byłoby twierdzenie, że został on zrealizowany w całości, gdyż ze względu na brak wsparcia dla biblioteki NMPRK na sprzęcie testowym, musiano ograniczyć się do implementacji wspierającej MPSS i NVML (dla NMPRK wsparcie jest jedynie teoretyczne, niepoparte testami w rzeczywistym środowisku). Wytworzona aplikacja rzeczywiście pozwala na obserwowanie zużycia energii i nakładanie na nie limitów w prosty sposób z wykorzystaniem wygodnego interfejsu użytkownika w postaci aplikacji działającej w środowisku przeglądarki internetowej. Udało się osiągnąć zakładaną elastyczność i modyfikowalność wytworzonego systemu. Format reguł pozwala na łatwe wprowadzenie nowych typów, co opisane zostało w podrozdziale 4.5.4. Interfejs komunikacyjny udostępniany przez serwer jest prosty i został wdrożony z wykorzystaniem popularnej architektury REST, co umożliwia zastąpienie obecnej aplikacji pełniącej rolę interfejsu użytkownika dowolnym innym komponentem, co ma istotne znaczenie dla kwestii potencjalnej integracji z Kernel Hive.

Drugi z istotnych celów pracy, czyli porównanie bibliotek służących do zarządzania zużyciem energii, również został zrealizowany w stopniu zadowalającym, w skutek wystąpienia wspomnianych wcześniej problemów z NMPRK, porównanie ograniczyło się głównie do MPSS i NVML, a analiza NMPRK została zrealizowana częściowo na tyle, na ile było to możliwe.

9.3. Propozycje usprawnień i dalszych prac

W obecnym stanie, każdy użytkownik, który zna adres i składnie poleceń API serwera lub agenta może wykonywać dowolne polecenia. Nie ma przed tym zabezpieczenia. Najprostszym rozwiązaniem tego problemu jest wysyłanie skrótów predefiniowanych haseł lub nazwy użytkowników i haseł zapisanych w bazie danych. W celu zapobieżenie przechwycenia tego rodzaju hasła należy zabezpieczyć ruch HTTP przy użyciu SSL/TLS[13]. Operacja taka jest możliwa na przykład przez użycie serwera WWW Apache jako pośrednika zarówno na węzłach obliczeniowych, jak i na serwerze. Proces taki jest stosunkowo prosty i został dokładnie opisany w książce *Webmin Administrator's Cookbook* [Karzyński M., 2014].

Istotnym usprawnieniem, które powinno zostać wprowadzone, jest modyfikacja sposobu komunikacji agenta z serwerem tak, aby to agent inicjował połączenie w postaci sesji TCP[14] i utrzymywał je aktywne. Obecnie wymiana informacji odbywa się na zasadzie żądanie – odpowiedź, z wykorzystaniem protokołu HTTP. Konsekwencją powyższego jest fakt, że bez dodatkowych operacji, komunikacja może być efektywnie realizowana jedynie na poziomie sieci lokalnej. Wystarczy, aby na drodze serwera stanęło urządzenie sieciowe wykonujące translację adresów lub zapora ogniowa, a wysyłanie komunikatów do agenta stanie się niemożliwe bez wprowadzania modyfikacji w konfiguracji powyższych (np. ustawienie przekierowania portu) lub stosowania dedykowanych do tego metod (np. odwrotne tunelowanie SSH, z którego skorzystano podczas testowania aplikacji – podrozdział 6.1.). Taka modyfikacja mogłaby iść w parze z poruszonym akapit wcześniej zabezpieczeniem komunikacji za pomocą TLS.

Warte rozważenia jest również wprowadzenie Google Protocol Buffers (GPB) jako nośnika informacji pomiędzy agentem a serwerem. Pozwoliłoby to na zminimalizowanie narzutu komunikacyjnego, który obecnie jest znaczący – wynika to z wykorzystania protokołu HTTP, a w zestawieniu z GPB można określić go nawet jako bardzo duży, ponieważ w drugim przypadku wykonywana jest bardzo efektywna serializacja danych, tak aby wolumen ruchu przesyłanego przez sieć był jak najmniejszy. GPB oferuje wsparcie zarówno dla C++, jak i dla Pythona, a dodatkowo jeden z autorów niniejszej pracy korzystał już z tego rozwiązania i zna je na tyle, że jego wprowadzenie nie stanowiłoby dla niego żadnego problemu.

[13] SSL (ang. *Secure Socket Layer* – warstwa bezpiecznego gniazda) oraz TLS (ang. Transport Layer Security – bezpieczeństwo warstwy transportowej) – protokoły zapewniające poufność i integralność transmisji w sieci.
[14] TCP (ang. *Transmission Control Protocol* – protokół kontroli transmisji) – protokół służący do wymiany informacji między procesami na różnych maszynach. Zakłada model połączeniowy.

Pożądaną funkcjonalnością, której agent obecnie nie posiada, jest automatyczne wykrywanie bibliotek takich, jak NVML czy MPSS podczas startu aplikacji, tak aby użytkownik nie był zmuszony za każdym razem specyfikować za pomocą argumentów linii poleceń, które z nich mają zostać załadowane. Powinien również zostać stworzony pakiet instalacyjny dla agenta, zawierający skrypt pozwalający na odpalanie go jako demona za pomocą polecenia *service*.

Serwer może zostać rozszerzony o dodatkowe szablony dla bardziej złożonych reguł zarządzania energią, gdyż obecnie istnieją tylko dwa, dość proste rodzaje reguł.

W module API należałoby zadbać o zwracanie, w odpowiedzi na żądania HTTP, bardziej zróżnicowanych statusów wraz z opisem błędu, aby ułatwić śledzenie problemów i ewentualne wyświetlanie ich w interfejsie.

Moduł Zarządcy w wypadku problemów z jego wydajnością można przerobić tak, aby korzystał z biblioteki *Celery*, która pozwala na zrównoleglenie działań przez niego wykonywanych.

Interfejs użytkownika powinien zostać ulepszony wizualnie, w celu uzyskania większej spójności stylów, a także wzbogacony o proste animacje, np. ładowania zasobów z serwera. Sprawdzanie poprawności podanych przez użytkownika danych również powinno zostać usprawnione poprzez wyraźniejsze zaznaczanie błędów i informowanie, w jaki sposób należy je poprawić. Przydatnym elementem byłoby opracowanie generycznego wyświetlania i dodawania reguł, co ułatwiłoby rozszerzanie kontrolerów i widoków o kolejne typy reguł. Część z kontrolek HTML, odpowiadających np. za wprowadzanie daty i godziny, powinna zostać zmieniona na przejrzystsze i przyjaźniejsze użytkownikowi. Dodatkowym usprawnieniem byłoby umożliwienie użytkownikowi edytowania już stworzonej reguły, zamiast musieć tworzyć ją od nowa. Wykres wyświetlania zużycia energii powinien zostać wzbogacony o wyświetlanie na nim obecnego limitu zużycia energii na urządzeniu.

W ramach kontynuacji części teoretycznej może zostać przeprowadzone porównanie wydajności energetycznej urządzeń Nvidia Tesla i Intel Xeon Phi tak, jak zasugerowano to w podrozdziale 6.4.

Wreszcie kontynuacją niniejszej pracy może być wprowadzenie jej jako podsystemu do zarządzania zużyciem energii do Kernel Hive, co było jedną z głównych motywacji do jej podjęcia. Zależy to jednak w dużym stopniu od oceny jej przydatności przez Opiekuna.

9.4. Wnioski

Agent jest nieświadomy istnienia innych elementów systemu (serwer, inne agenty), a komunikacja z nim odbywa się za pomocą odwoływania się do udostępnianych przez niego zasobów REST, oznacza to możliwość wykorzystania agenta, jako samodzielnej aplikacji bez angażowania reszty systemu, np. wysyłając żądania z wykorzystaniem *cURL*.

Konieczne było wykorzystanie ładowania bibliotek dynamicznych podczas działania aplikacji, by agent mógł być bezproblemowo uruchamiany na różniących się od siebie maszynach, np. z zainstalowanym jedynie NVML.

Pomiędzy wykorzystywanymi bibliotekami służącymi do zarządzania zużyciem energii występują pewne podobieństwa, wszystkie z nich pozwalają na odczytanie aktualnego zużycia oraz odczytanie i ustawienie limitów. Można też zauważyć istotne różnice, głównie jeżeli chodzi o definiowanie granicznych wartości limitów oraz sposobu ustawiania samego ograniczenia (miękkie i twarde limity). Brak wsparcia sprzętowego dla NMPRK uniemożliwił przetestowanie tej biblioteki w praktyce, poza tym najpoważniejszym problemem, którego rozwiązanie zaproponowano w podrozdziale 4.4.4., jest brak możliwości ustawienia twardego limitu zużycia energii za pomocą MPSS.

Warta podkreślenia jest wysoka jakość dokumentacji technicznej udostępnianej przez firmę Nvidia, na tle której ta stworzona przez Intel wypada miernie.

Obliczenia przeprowadzone w ramach rozdziału 2. i wyjaśnione dokładnie w dodatku A wyraźnie wskazują na większą wydajność energetyczną, przynajmniej w teorii, kart graficznych względem innych urządzeń.

W trakcie jednego z testów praktycznych zauważono ciekawą zależność związaną ze zużyciem energii w stanie bezczynności. Koprocesor Intel Xeon Phi pobierał jej ponad pięciokrotnie więcej niż karta Nvidia Tesla, zostało to dokładnie opisane w podrozdziale 6.4.

Udało się osiągnąć zakładaną elastyczność systemu:

- możliwa jest łatwa wymiana komponentu pełniącego rolę interfejsu użytkownika;
- dodawanie nowych szablonów reguł nie powinno sprawiać żadnych trudności;
- architektura agenta pozwala na bezproblemowe rozszerzenie go o wsparcie dla nowych typów urządzeń (bibliotek).

Z procesu implementacji serwera można zauważyć, że tworzenie aplikacji internetowych, które wystawiają określone zasoby API REST jest bardzo proste przy użyciu języka, bibliotek i narzędzi opisanych w podrozdziale 5.2. Podobne wnioski można wyciągnąć z implementacji interfejsu użytkownika, gdzie dobór bibliotek był bardzo szeroki, lecz zostały wyselekcjonowane tylko te, które pozwalały osiągnąć zamierzony efekt w jak najprostszy sposób (podrozdział 5.3.).

Ponadto wartym zauważenia jest fakt, iż odpowiednie przygotowanie środowiska wytwarzania oprogramowania, w tym edytorów, narzędzi automatyzacji pracy, a także automatyzacji testowania, pozwala ograniczyć czas potrzebny na wykonywanie zadań pobocznych, jak uruchamianie testów, ręczne kompilowanie kodu lub jego transfer do środowiska uruchomieniowego. Jednocześnie zwiększa ilość czasu, którą można przeznaczyć na faktyczne implementowanie funkcjonalności i właściwe ich testowanie. Proces ciągłej integracji, opisany w podrozdziale 8.2., w dużej mierze przyczynił się do eliminacji błędów na początkowych etapach implementacji modułów.

WYKAZ LITERATURY

1. Kirk D., Hwu W.: *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers, 2010.
2. Czarnul P.: *Wydajne obliczenia na klastrach i gridach*, w: Aplikacje rozproszone i systemy internetowe: praca zbiorowa Katedry Architektury Systemów Komputerowych, Politechnika Gdańska, 2006.
3. Li B. i inni: *The Power-Performance Tradeoffs of the Intel Xeon Phi on HPC Applications*, In Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW '14), IEEE Computer Society, Stany Zjednoczone, 2014.
4. Czarnul P., Rościszewski P.: *Optimization of Execution Time under Power Consumption Constraints in a Heterogeneous Parallel System with GPUs and CPUs*, w: Distributed Computing and Networking, Springer, 2014.
5. Wilk A.: *Komputery bez procesora?*, Burda Communications, 25 września 2009, http://www.chip.pl/artykuly/trendy/2009/09/komputery-bez-procesora (data dostępu 26 listopada 2015).
6. Czarnul P.: *Materiały wykładowe do przedmiotu Przetwarzanie równoległe CUDA,* WETI PG, 2014.
7. Jędruch A.: *Materiały wykładowe do przedmiotu Architektura Komputerów*, WETI PG, 2013.
8. *TOP500*, http://www.top500.org/ (data dostępu 26 listopada 2015).
9. Sanders J., Kandrot E.: Cuda *by example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley, 2011.
10. Ge R. i inni: *Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU*, w: 'ICPP', IEEE, Stany Zjednoczone 2013.
11. Stolarczyk S.: *ASUS PhysX P1 - sprzętowy akcelerator fizyki*, Grupa Onet.pl, 15 maja 2006, http://pclab.pl/art19719-2.html (data dostępu 24 listopada 2015).
12. *Intel Xeon Phi™ Coprocessor x100 Product Family Datasheet*, Intel, kwiecień 2015, http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html (data dostępu 24 listopada 2015).
13. *AMD vs Intel Market Share*, PassMark Software, listopad 2015, https://www.cpubenchmark.net/market_share.html (data dostępu 26 listopada 2015).
14. *GREEN500*, http://www.green500.org/ (data dostępu 25 listopada 2015).
15. Matuszek M.: *Materiały wykładowe do przedmiotu Systemy Agentowe*, WETI PG, 2015.
16. Freeman E. i inni: *Wzorce projektowe. Rusz głową!*, Helion, Gliwice 2011.
17. Stroustrup B.: *A Tour of C++*, Addison-Wesley, 2013.
18. Summerfield M.: *Python 3. Kompletne wprowadzenie do programowania, wydanie II*, Helion, Gliwice 2010.
19. Gorelick M., Ozsvald I.: *High Performance Python*, O'Reilly Media, Sebastopol, Stany Zjednoczone 2014.
20. *Tornado Web Server*, http://www.tornadoweb.org (data dostępu 24 listopada 2015).
21. Nayak A.: *MongoDB Cookbook*, Packt Publishing, Birmingham, Wielka Brytania 2014.
22. Bhaumik S.: *Bootstrap Essentials*, O'Reilly Media, Sebastopol, Stany Zjednoczone 2013.
23. Kalbarczyk A., Kalbarczyk D.: *AngularJS pierwsze kroki*, Helion, Gliwice 2015.
24. *NVML API Reference Guide*, Nvidia, maj 2015, http://docs.nvidia.com/deploy/nvml-api/index.html (data dostępu 24 listopada 2015).
25. *Intel Manycore Platform Software Stack User's Guide*, Intel, 2014, http://registrationcenter.intel.com/irc_nas/7689/mpss_users_guide.pdf (data dostępu 24 listopada 2015).

26. *Intel Node Manager Programmer's Reference Kit Implementation Guide*, Intel, 2012, https://01.org/sites/default/files/documentation/implementationguide-nmprk.pdf (data dostępu 24 lIistopada 2015).
27. Milanesio L.: *Learning Gerrit Code Review*, Packt Publishing, Birmingham, Wielka Brytania 2013.
28. Sale D.: *Testing Python: Applying Unit Testing, TDD, BDD and Acceptance Testing*, John Wiley & Sons, Nowy Jork, Stany Zjednoczone 2014.
29. Ragonha P.: *Jasmine JavaScript Testing - Second Edition*, Packt Publishing, Birmingham, Wielka Brytania 2015.
30. Karzyński M.: *Webmin Administrator's Cookbook,* rodział 8, Packt Publishing, Birmingham, Wielka Brytania 2014.
31. Fernandez M. R., *Nodes, Sockets, Cores and FLOPS, Oh, My*, listopad 2011, http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2329 (data dostępu 26 listopada 2015).
32. Bobkowska A.: *Materiały wykładowe do przedmiotu Inżynieria Oprogramowania*, WETI PG, 2014.
33. O'Higgins N.: *MongoDB and Python*, O'Reilly Media, Sebastopol, Stany Zjednoczone 2011.

WYKAZ LISTINGÓW

WYKAZ RYSUNKÓW

WYKAZ TABEL

DODATEK A: METODA OBLICZANIA TEORETYCZNEJ WYDAJNOŚCI ENERGETYCZNEJ

W celu obliczenia teoretycznej wydajności energetycznej urządzenia, wyrażanej we FLOPSach [Fernandez M. R., 2011] na wat, potrzebujemy kilku podstawowych informacji o nim.

- TDP (ang. *Thermal Design Power*) – określa maksymalną ilość ciepła, które może wydzielić urządzenie podczas pracy, a tym samym maksymalną ilość energii pobieraną przez nie. Należy zauważyć, że rzeczywiste zużycie energii może przekroczyć TDP jednak przyjęło się, żeby stosować je jako wartość referencyjną przy wykonywaniu różnego rodzaju obliczeń.
- Częstotliwość cyklu zegarowego – czyli ilość cykli na sekundę, które wykonuje każdy z rdzeni procesora.
- Liczba fizycznych rdzeni – nie uwzględniamy technologii w rodzaju *Hyper-Threading*, gdyż polegają one na zwielokrotnieniu tylko pewnych elementów składowych rdzenia i w efekcie nie pozwalają na faktyczne wykonywanie dwóch (lub czterech) instrukcji w pojedynczym cyklu.
- Liczba instrukcji zmiennoprzecinkowych wykonywanych w pojedynczym cyklu – tu pojawia się istotna różnica pomiędzy poszczególnymi urządzeniami, zostanie ona omówiona dalej.

Wykonując teoretyczne oszacowanie mocy obliczeniowej danego urządzenia należy wziąć pod uwagę scenariusz, w którym jego możliwości są w pełni wykorzystywane. Oznacza to, że należy uwzględnić posiadanie przez dany układ sprzętowy jednostek wektorowych, wykonujących obliczenia na wielu operandach na raz. Współczesne procesory firmy Intel posiadają jednostki wektorowe o długości 256 bitów, a koprocesory Intel Xeon Phi 512 bitów, można więc w prosty sposób obliczyć ilość operandów, na których zostanie wykonana instrukcja w pojedynczym cyklu: 256 / 32 = 8, 512 / 32 = 16. Dzielnik o wartości 32 wynika z faktu, że omawiamy przypadek, gdy obliczenia są wykonywane na liczbach zmiennoprzecinkowych pojedynczej precyzji, a więc 32 bitowych.

Istotnym elementem jest również typ instrukcji, która może być wykonywana w pojedynczym cyklu. Wszystkie urządzenia, które przedstawiono na początku pracy w tabeli 2.1. mogą wykorzystywać instrukcje MAC (ang. *multiply-accumulate*) lub FMAC (ang. *fused multiply-accumulate*), które pozwalają na wykonanie w pojedynczym cyklu dodawania i mnożenia, a więc dwóch operacji zmiennoprzecinkowych. Różnica pomiędzy MAC i FMAC polega na sposobie zaokrąglania podczas obliczeń. Pierwsze z nich zaokrąglają zarówno wynik operacji mnożenia, jak i dodawania, natomiast drugie wykonują mnożenie i dodawanie dokładnie, a dopiero wynik końcowy zostaje zaokrąglony.

Należy mieć na uwadze, że prezentowana metoda nie bierze pod uwagę innych istotnych czynników wpływających na rzeczywistą wydajność obliczeniową urządzenia, taki jak: przepustowość szyny pamięci, rozmiar pamięci podręcznej, możliwość ciągłej pracy pod maksymalnym obciążeniem i inne, a więc jest ona jedynie sposobem na wyliczenie „surowej”

mocy obliczeniowej danego układu i należy oczekiwać, że faktyczna wydajność będzie mniejsza i zależna od typu wykonywanych obliczeń.

Po obliczeniu wydajności danego urządzenia wyrażanej we FLOPSach należy otrzymany wynik podzielić przez ilość zużywanej energii, aby w rezultacie otrzymać wydajność energetyczną, mamy więc ogólny wzór:

$$Eff = \frac{f \times cores \times IPC}{TDP} \tag{1.1}$$

gdzie:

Eff – wydajność energetyczna urządzenia [FLOPS/W];

f – częstotliwość cyklu zegarowego [Hz];

cores – liczba fizycznych rdzeni;

IPC – liczba operacji zmiennoprzecinkowych wykonywanych w jednym cyklu po uwzględnieniu jednostek wektorowych i instrukcji MAC;

TDP – ilość energii zużywanej przez urządzenie [W].

Przeprowadźmy teraz przykładowe obliczenia dla trzech urządzeń przedstawionych w tabeli 2.1. z wykorzystaniem wzoru 1.1. Dane techniczne (częstotliwość cyklu zegarowego, liczba rdzeni, itd.) pozyskane zostały ze stron producentów.

Nvidia Tesla K80: $0{,}875GHz \times 4992 \times 2 \,/\, 300W \;= 29{,}12\ [GFLOPS/W]$

Intel Xeon Phi 7120P: $1{,}238GHz \times 61 \times (512 \,/\, 32 \times 2) \,/\, 300W \;= 8{,}06\ [GFLOPS/W]$

Intel Xeon E5-2697v3: $2{,}6GHz \times 14 \times (256 \,/\, 32 \times 2) \,/\, 145W \;= 4{,}02\ [GFLOPS/W]$

DODATEK B: SZCZEGÓŁOWE OPISY PRZYPADKÓW UŻYCIA

Nazwa	Wyszukiwanie urządzeń.
Warunki początkowe	• Lista urządzeń jest pusta.
Przebieg	1. Dla każdej ze wspieranych technologii wywoływana jest funkcja zwracająca listę dostępnych urządzeń. 2. Jeżeli zwrócona lista nie jest pusta, to dla każdego urządzenia z listy: a. odczytaj informacje dotyczące urządzenia; b. dodaj odczytane informacje do wpisu na liście dotyczącego tego urządzenia. 3. Otrzymana lista jest dopisywana do listy głównej.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Na liście urządzeń znajdują się wszystkie wykryte urządzenia.

Nazwa	Ustawianie limitu zużycia energii.
Warunki początkowe	• Wybrane urządzenie. • Podany limit jako wartość całkowita lub procentowa.
Przebieg	1. Wywoływany jest PU „Odczytywanie wartości granicznych możliwych do ustawienia limitów”. 2. Sprawdzane jest, czy nowy limit mieści się w dopuszczalnych granicach. 3. Jeżeli tak: nowy limit zostaje ustawiony. 4. Jeżeli nie: zwracany jest komunikat o błędzie.
Przebiegi alternatywne	1. Podczas odwoływania się do urządzenia wystąpił wyjątek: a. operacja jest przerywana i zwracany jest komunikat o błędzie.
Warunki końcowe	• Limit został ustawiony lub zwrócono komunikat o błędzie.

Nazwa	Odczytywanie wartości granicznych możliwych do ustawienia limitów.
Warunki początkowe	• Wybrane urządzenie.
Przebieg	1. Odczytywane są wartości graniczne możliwych do ustawienia limitów.
Przebiegi alternatywne	1. Podczas odwoływania się do urządzenia wystąpił wyjątek: a. operacja jest przerywana i zwracany jest komunikat o błędzie.
Warunki końcowe	• Odczytane wartości zostały przekazane wywołującemu.

Nazwa	Odczytywanie aktualnego limitu zużycia energii.
Warunki początkowe	• Wybrane urządzenie.
Przebieg	1. Odczytywany jest limit zużycia energii.
Przebiegi alternatywne	1. Podczas odwoływania się do urządzenia wystąpił wyjątek: a. operacja jest przerywana i zwracany jest komunikat o błędzie.
Warunki końcowe	• Odczytana wartość została przekazana wywołującemu.

Nazwa	Odczytywanie bieżącego zużycia energii.
Warunki początkowe	• Wybrane urządzenie.
Przebieg	1. Odczytywane jest zużycie energii.
Przebiegi alternatywne	1. Podczas odwoływania się do urządzenia wystąpił wyjątek: a. operacja jest przerywana i zwracany jest komunikat o błędzie.
Warunki końcowe	• Odczytana wartość została przekazana wywołującemu.

Nazwa	Pobieranie informacji o węźle.
Warunki początkowe	• Obsługiwana jest kwerenda dotycząca pobierania informacji o węźle.
Przebieg	1. Odczytywane są informacje o systemie operacyjnym. 2. Odczytywane są informacje dla każdego urządzenia znajdującego się na liście urządzeń. 3. Odczytane informacje zostały umieszczone w strukturze będącej odpowiedzią na kwerendę.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Została przygotowana odpowiedź na kwerendę zawierająca odczytane informacje.

Nazwa	Wykonanie kwerendy.
Warunki początkowe	• Odebrano kwerendę. • Podany typ kwerendy.
Przebieg	1. Odczytywana jest treść kwerendy. 2. Treść kwerendy sprawdzana jest pod kątem poprawności składniowej. 3. Jeżeli składnia jest niepoprawna: a. Wykonanie kwerendy jest przerywane i zostaje zwrócony komunikat o błędzie. 4. Dla każdego elementu kwerendy (elementem jest pojedyncze polecenie wchodzące w skład kwerendy): a. stwórz „Obiekt Polecenie” zawierający informacje niezbędne do wykonania elementu kwerendy; b. przekaż „Polecenie” do warstwy komunikacji z urządzeniami w celu jego wykonania; c. warstwa komunikacji z urządzeniami wykonuje Polecenie i zwraca rezultat. 5. Umieść rezultaty wykonania poszczególnych elementów we wspólnej odpowiedzi. 6. Przekaż przygotowaną odpowiedź do wywołującego.
Przebiegi alternatywne	1. Podczas wykonywania Polecenia wystąpił wyjątek: a. operacja jest przerywana i zwracany jest komunikat o błędzie.
Warunki końcowe	• Odpowiedź przekazana do wywołującego lub zwrócony komunikat o błędzie.

Nazwa	Odbieranie kwerend
Warunki początkowe	• Podana lista rozpoznawanych typów kwerend wraz z obiektami służącymi do ich obsługi.
Przebieg	1. Obiekty obsługujące komendy są rejestrowane w systemie. 2. Rozpoczyna się nasłuch. 3. Zostaje odebrana kwerenda. 4. Jeżeli typ kwerendy jest znany: a. wykonaj PU „Wykonanie kwerendy”. 5. Jeżeli typ kwerendy nie jest znany: a. W odpowiedzi umieść kod błędu. 6. Wyślij odpowiedź na kwerendę. 7. Jeżeli nie otrzymano polecenia zakończenia wykonania, to kontynuuj od pkt. 2.
Przebiegi alternatywne	1. Podczas wykonywania kwerendy wystąpił wyjątek: a. operacja jest przerywana i zwracany jest kod błędu oraz komunikat o błędzie.
Warunki końcowe	• Brak.

Nazwa	Wysyłanie kwerendy.
Warunki początkowe	• Wybrany węzeł obliczeniowy. • Przygotowana kwerenda.
Przebieg	1. Kwerenda zostaje wysłana. 2. Rozpoczyna się oczekiwanie na odpowiedź. 3. Jeżeli upłynie czas oczekiwania na odpowiedź: a. przerwij i zwróć komunikat o błędzie. 4. Jeżeli odpowiedź nadeszła: a. przekaż treść odpowiedzi do wywołującego.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Treść odpowiedzi przekazana do wywołującego bądź zwrócony komunikat o błędzie.

Nazwa	Monitorowanie stanu węzłów obliczeniowych.
Warunki początkowe	• Dana lista węzłów obliczeniowych.
Przebieg	1. Dla każdego węzła na liście: a. wyślij kwerendę zawierającą zapytanie o aktualne zużycie energii każdego ze znajdujących się na nim urządzeń (PU Wysyłanie kwerendy); b. wpisz otrzymane chwilowe wartości zużycia energii do bazy danych; c. wykonaj dla urządzenia PU „Określanie oczekiwanego limitu zużycia energii”; d. odpytaj urządzenia o ustawione limity zużycia energii; e. jeżeli ustawiony limit nie zgadza się z oczekiwanym: i. wyślij kwerendę nakazującą zmianę limitu zużycia energii na oczekiwaną wartość. 2. Jeżeli nie otrzymano polecenia zakończenia wykonania, to kontynuuj od pkt. 1.
Przebiegi alternatywne	1. Podczas wykonywania kwerendy wystąpił wyjątek: a. wykonanie operacji dla danego urządzenia jest przerywane, wykonanie jest kontynuowane od następnego w kolejności urządzenia.
Warunki końcowe	• Chwilowe zużycie energii wpisane do historii. • Limity zużycia energii na wszystkich urządzeniach mają odpowiednią wartość.

Nazwa	Określanie oczekiwanego limitu zużycia energii.
Warunki początkowe	• Wybrany węzeł obliczeniowy. • Wybrane urządzenie. • Dana lista reguł zarządzania energią dla wybranego urządzenia.
Przebieg	1. Określ początkowy limit zużycia energii. 2. Reguły na liście są przetwarzane sekwencyjnie, dla każdej z nich: a. Jeżeli istnieje taka konieczność, odczytaj odpowiedni fragment historii zużycia energii (PU Odczyt historii zużycia energii); b. zmodyfikuj początkowy limit zużycia energii zgodnie z oczekiwaniami. 3. Zwróć wartość limitu zużycia energii otrzymaną po przetworzeniu wszystkich reguł.
Przebiegi alternatywne	1. Podczas przetwarzania reguły wystąpił błąd: a. przerwij operację i zwróć komunikat o błędzie.
Warunki końcowe	• Oczekiwana wartość zużycia energii została zwrócona wywołującemu.

Nazwa	Odczyt historii zużycia energii.
Warunki początkowe	• Wybrane urządzenie. • Wybrany przedział czasowy.
Przebieg	1. Stwórz zapytanie do bazy danych na podstawie zadanego urządzenia i przedziału czasowego. 2. Wyślij zapytanie. 3. Przekaż wynik zapytania do wywołującego.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Historia zużycia energii przekazana do wywołującego.

Nazwa	Przechowywanie informacji o węzłach obliczeniowych w bazie danych.
Warunki początkowe	• Dana lista węzłów obliczeniowych (z możliwością jej edycji).
Przebieg	1. W przypadku zapisu: a. wykonaj serializację danych; b. wykonaj zapis. 2. W przypadku odczytu: a. wykonaj odczyt; b. wykonaj deserializację danych.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Dane zostały odczytane z bazy lub zapisane w niej.

Nazwa	Przechowywanie reguł zarządzania energią w bazie danych.
Warunki początkowe	• Dana lista reguł zarządzania energią(z możliwością jej edycji).
Przebieg	1. W przypadku zapisu: a. wykonaj serializację danych. b. wykonaj zapis. 2. W przypadku odczytu: a. wykonaj odczyt; b. wykonaj deserializację danych.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Dane zostały odczytane z bazy lub zapisane w niej.

Nazwa	Dodanie nowego węzła obliczeniowego.
Warunki początkowe	Brak.
Przebieg	1. Użytkownikowi zostaje wyświetlone okno służące dodawaniu nowego węzła. 2. Użytkownik wprowadza adres i port, na którym nasłuchuje agent. 3. Użytkownik potwierdza wprowadzone dane. 4. Do serwera zostaje wysłane polecenie dodania nowego węzła. 5. Serwer wysyła do węzła zapytanie o jego informacje (PU Wysyłanie kwerendy). 6. Jeżeli węzeł odpowiedział: a. węzeł zostaje dodany do listy; b. serwer wysyła wiadomość o sukcesie do interfejsu użytkownika; c. użytkownikowi zostaje wyświetlona informacje o udanym dodaniu węzła. 7. Jeżeli węzeł nie odpowiedział: a. serwer wysyła wiadomość o porażce do interfejsu użytkownika; b. użytkownikowi zostaje wyświetlona informacja o nieudanej próbie dodania węzła.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Węzeł został dodany lub wyświetlony komunikat o błędzie.

Nazwa	Listowanie dostępnych węzłów obliczeniowych.
Warunki początkowe	• Brak.
Przebieg	1. Do serwera wysyłane jest zapytanie o listę węzłów obliczeniowych. 2. Jeżeli serwer poinformował o sukcesie: a. lista zostaje wyświetlona użytkownikowi. 3. Jeżeli wystąpił błąd: a. wyświetl komunikat o błędzie.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Wyświetlona lista węzłów obliczeniowych lub wyświetlony komunikat o błędzie.

Nazwa	Usunięcie węzła obliczeniowego z listy.
Warunki początkowe	• Lista węzłów widoczna dla użytkownika.
Przebieg	1. Użytkownik wybiera, który węzeł chce usunąć. 2. Polecenie usunięcia węzła z listy zostaje wysłane do serwera. 3. Jeżeli serwer poinformował o sukcesie: a. wyświetl komunikat o udanym usunięciu węzła. 4. Jeżeli wystąpił błąd: a. wyświetl komunikat o błędzie.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Węzeł usunięty lub wyświetlony komunikat o błędzie.

Nazwa	Wyświetlanie informacji o węźle.
Warunki początkowe	• Lista węzłów widoczna dla użytkownika.
Przebieg	1. Użytkownik wybiera węzeł, dla którego chce wyświetlić okno z informacjami. 2. Do serwera zostaje wysłane zapytanie o informacje dotyczące węzła (zawierające listę dostępnych urządzeń). 3. Jeżeli serwer poinformował o sukcesie: a. wyświetl okno z informacjami o węźle. 4. Jeżeli wystąpił błąd: a. wyświetl komunikat o błędzie.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Okno z informacjami o węźle wyświetlone użytkownikowi lub wyświetlony komunikat o błędzie.

Nazwa	Dodanie reguły zarządzania energią.
Warunki początkowe	• Wybrany węzeł.
Przebieg	1. Użytkownik wybiera opcję dodania nowej reguły zarządzania energią. 2. Wyświetlana jest lista prototypów reguł. 3. Użytkownik wybiera jeden z prototypów z listy. 4. Użytkownik podaje parametry wymagana do utworzenia reguły o danym prototypie. 5. Użytkownik potwierdza wprowadzone wartości. 6. Do serwera zostaje wysłane polecenie dodania nowej reguły. 7. Serwer przetwarza otrzymane polecenie. 8. Jeżeli serwer poinformował o sukcesie: a. wyświetl informację o udanym dodaniu reguły. 9. Jeżeli wystąpił błąd: a. wyświetl komunikat o błędzie.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Dodana nowa reguła lub wyświetlony komunikat o błędzie.

Nazwa	Modyfikowanie zbioru reguł zarządzania energią.
Warunki początkowe	• Wybrana reguła.
Przebieg	1. Użytkownik określa akcję którą chce wykonać. 2. Jeżeli usunięcie: a. wyślij polecenie usunięcia reguły do serwera. 3. Jeżeli zmiana kolejności: a. odczytaj nową pozycję reguły w sekwencji reguł; b. wyślij polecenie zmiany sekwencji reguł do serwera. 4. Jeżeli zmiana parametrów: a. użytkownikowi zostaje wyświetlone okno z aktualnymi parametrami; b. użytkownik wprowadza i potwierdza modyfikacje; c. polecenie zmiany parametrów reguły zostaje wysłane do serwera. 5. Jeżeli serwer poinformował o sukcesie: a. wyświetl informację o udanej modyfikacji lub usunięciu reguły. 6. Jeżeli wystąpił błąd: a. wyświetl komunikat o błędzie.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Modyfikacja przebiegła pomyślnie lub wyświetlony komunikat o błędzie.

Nazwa	Listowanie zbioru reguł zarządzania energią.
Warunki początkowe	• Wybrany węzeł.
Przebieg	1. Użytkownik wybiera urządzenie, dla którego reguły mają zostać odczytane. 2. Reguły zarządzania energią są odczytywane z bazy danych. 3. Reguły zostają przekazane do interfejsu użytkownika.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Użytkownik widzi wyświetlone reguły.

Nazwa	Odczytywanie danych statystycznych.
Warunki początkowe	• Wybrany węzeł.
Przebieg	1. Użytkownik określa przedział czasu i wybiera urządzenie, dla którego dane mają zostać wyświetlone. 2. Do serwera wysyłane jest polecenie o podanie danych. 3. Dane są odczytywane z bazy danych. 4. Jeżeli serwer poinformował o sukcesie: a. dane zostają wyświetlone użytkownikowi. 5. Jeżeli wystąpił błąd: a. wyświetl komunikat o błędzie.
Przebiegi alternatywne	Brak.
Warunki końcowe	• Dane zostały wyświetlone użytkownikowi lub został wypisany komunikat o błędzie.