Politechnika Krakowska im. Tadeusza Kościuszki

Wydział Inżynierii Elektrycznej i Komputerowej

Praca inżynierska

**Porównanie klasycznych stosów programistycznych z nowoczesnym podejściem do budowy oprogramowania.**

*Patryk Żmigrodzki*

*Promotor*

prof. dr hab. inż. Volodymyr Samotyy

Kraków, 2016

**Spis treści**

**1 Wstęp 4**

**2 Opis technologii 5**

2.1 Java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 JavaScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2.1 Node.js . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.3 Elixir i Erlang . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.3.1 Erlang . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.3.2 Elixir . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Architektura systemu informatycznego 8**

3.1 Współczesne trendy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.1.1 Architektura mikroserwisowa . . . . . . . . . . . . . . . . . . 9

3.2 Java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.2.1 Architektura wielowarstwowa . . . . . . . . . . . . . . . . . . 10

3.3 JavaScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.3.1 libuv . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.3.2 Wzorzec Reaktor . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.4 Elixir . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.4.1 Open Telecom Platform . . . . . . . . . . . . . . . . . . . . . 15

3.4.2 Model aktorowy . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.5 Dyskusja . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**4 Wydajność 19**

4.1 Duża liczba zapytań . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4.1.1 | Java . . . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 20 |
| 4.1.2 | JavaScript | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 24 |
| 4.1.3 | Elixir . . . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 29 |
| Czasochłonne obliczenia . . . . . . . . . . . . . . . . . . . . . . . . . 33 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4.2.1 | Java . . . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 33 |
| 4.2.2 | JavaScript | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 38 |
| 4.2.3 | Elixir . . . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 43 |
| Operacje na zbiorach danych . . . . . . . . . . . . . . . . . . . . . . . 48 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4.3.1 | Java . . . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 48 |
| 4.3.2 | JavaScript | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 53 |
| 4.3.3 | Elixir . . . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 58 |
| Ograniczenia wejścia/wyjścia . . . . . . . . . . . . . . . . . . . . . . . 63 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4.4.1 | Java . . . . . . . . . . | | | | | | | | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 63 |
| 4.4.2 | JavaScript . . . . . . . | | | | | | | | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 68 |

4.2

4.3

4.4

4.4.3 Elixir . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

4.5 Podsumowanie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

**5 Skalowalność 82**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5.1 | Skalowalność pionowa . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 82 |
| 5.2 | Skalowalność pozioma | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 83 |
| 5.3 | Dyskusja . . . . . . . . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | 84 |

**6 Produktywność 85**

6.1 Kod . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

6.2 Biblioteki . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

**7 Uwagi końcowe 88**

**8 Możliwości rozwoju 89**

**1 Wstęp**

Rozwój jest koniecznym i nieodłącznym elementem każdej branży. W tak dy- namicznie rozwijającej się dziedzinie jaką jest informatyka szczególnie ważne jest, aby nie dopuścić do stagnacji, zachować konkurencyjność. Innowacyjność na polu programistycznym może okazać się kartą przetargową do uzyskania przewagi w określonym sektorze. Z drugiej strony optymistyczne i pochopne podejście do nowości, zbyt wczesna adaptacja niedojrzałych lub niestabilnych technologii może doprowadzić do poważnych komplikacji.

Przedmiotem tej pracy jest zbadanie wyłaniających się technologii w zastosowa- niach biznesowych. Wydzielono dwie grupy w celu ich porównania oraz sprawdzenia przydatności i potencjału technologicznego. Do pierwszej grupy należą środowiska cieszące się dobrą reputacją, z ustaloną pozycją na rynku, zaś do drugiej rozwią- zania nowe, promujące odmienne modele programowania. Przedstawiono je na płaszczyźnie współczesnych problemów systemów informatycznych. Poruszono tematy architektury takich systemów jako istotnego czynnika w tworzeniu udanych projektów programistycznych. Równie znaczącym zagadnieniem jest efektywne wy- korzystanie dostępnego sprzętu komputerowego poprzez przetwarzanie wieloproce- sorowe. Z tego względu przyjrzano się różnym podejściom i modelom współbież- ności, wykorzystywanym przez testowane technologie. Jednakże, możliwości rów- noczesnego przetwarzania nie kończą się na pojedynczej maszynie. Zastosowanie architektur rozproszonych jest równie ważną cechą współczesnych systemów infor- matycznych. Poza prześledzeniem aspektów skalowalności przetestowano wydajność wybranych technologii w typowych dla współczesnych systemów informatycznych scenariuszach użytkowania. Poddano je również analizie pod względem efektyw- ności pracy programistycznej, będącej kluczowym czynnikiem w procesie tworzenia oprogramowania.

**2 Opis technologii**

Współczesne języki programowania to nie tylko składania i kompilatory. To także ustalone modele i wzorce programistyczne, standardy oraz społeczności.

Typowym przedstawicielem kategorii klasycznych technologii jest Java, ze względu na powszechność, długą historię oraz wyraźne podłoże biznesowe. Do drugiej, inno- wacyjnej, grupy wybrano dwa języki programowania: skupiający ogromną społecz- ność programistów JavaScript wraz z Node.js oraz promujący funkcyjną metodykę programowania Elixir.

**2.1 Java**

Java jest platformą programistyczną, której kluczowymi elementami są: obiek- towy język programowania Java oraz Java Virtual Machine (ang. wirtualna ma- szyna Javy). Została stworzona w 1995 przez firmę Sun Microsystems do budowa- nia przenośnego oprogramowania. Obecnie ona jest szyk Oracle Corporation. Java należy do jednych z najpopularniejszych języków programowania na świecie[1]. Dzieli się ona na kilka wydań różniących się funkcjonalnością i przeznaczeniem. Naj- bardziej powszechne z nich to *Java Standard Edition* przeznaczone do zastosowań ogólnych. Stanowi ona rdzeń języka wzbogacony o często wykorzystywane biblioteki jak dostęp do bazy danych czy łączność sieciową oraz narzędzia, w skład których wchodzą wirtualna maszyna i aplikacje deweloperskie.

Kolejnym wydaniem jest *Java Micro Edition* cechująca się małymi wymaganiami sprzętowymi, przeznaczona do programowania systemów wbudowanych. Rozbu- dowaną bibliotekę edycji standardowej zastąpiono mechanizmami niezbędnymi do interakcji z warstwą sprzętową. Ostatnim z nich jest *Java Enterprise Edition*. Ba- zuje ono na Javie SE, rozszerzając ją o interfejsy przeznaczone dla rozbudowanych, skalowalnych systemów sieciowych do zastosowań biznesowych.[2]

**2.2 JavaScript**

JavaScript jest językiem programowania zapoczątkowanym w 1995 przez Net- scape Communications Corporation do wykonywania skryptów na stronach inter- netowych. Język ten bardzo się rozwinął i wyrósł ponad pierwotne skryptowe za- stosowania. Bywa nazywany językiem Internetu, gdyż jest implementowany przez większość przeglądarek internetowych stanowiąc warstwę logiczną stron www, obok HTML tworzącego treść i CSS specyfikującego warstwę prezentacji.[3]

**2.2.1 Node.js**

Nie wszystkie implementacje JavaScriptu są częścią przeglądarek internetowych. JavaScript można również zastosować do tworzenia oprogramowania serwerowego przy użyciu środowiska Node.js. Dzięki temu programiści znający język z two- rzenia kodu klienckiego mogą wykorzystać swoje umiejętności do programowania logiki serwerowej. Node.js bazuje na silniku JavaScriptowym przeglądarki Google Chrome nazywanym V8. Do implementacji silnika oraz Node.js wykorzystano języki C oraz C++. Wybór ten podyktowany był możliwością ograniczenia zapotrzebo- wania na pamięć operacyjną przy zachowaniu wysokiej wydajności. W przeciwień- stwie do większości współczesnych środowisk programistycznych, Node do obsługi współbieżnego przetwarzania logiki biznesowej **nie wykorzystuje** wielowątkowości, a **modelu** opartego o asynchronicznie przetwarzane zdarzeń. Takie połączenie miało na celu stworzenie platformy, która umożliwiałaby w prosty sposób budowanie lekkich i wydajnych systemów informatycznych. Dzięki powszechności JavaSciptu w przeglądarkach internetowych oraz powstania możliwości zastosowania go po stronie serwerowej, wokół języka zgromadziła się ogromna społeczność. Repozytorium *npm* przechowuje ponad 230 tysięcy publicznie dostępnych bibliotek oraz odnotowuje prawie 140 milionów pobranych plików dziennie.[4–6]

**2.3 Elixir i Erlang**

Elixir to język programowania stworzony na podstawie języka *Erlang*. Korzenie te są na tyle wyraźne i znaczące, że nie sposób go przedstawić bez wcześniejszego zapoznania się z Erlangiem.

**2.3.1 Erlang**

Erlang jest funkcyjnym językiem programowania stworzonym w latach osiem- dziesiątych przez szwedzką firmę telekomunikacyjną Ericsson. Przeznaczeniem tej technologii było budowanie skalowalnych i niezawodnych systemów.

Pomimo swoich początków w systemach telekomunikacyjnych nie jest wyspecjalizo- wany w tej domenie, a sprawdza się wszędzie tam gdzie zachodzi potrzeba współ- bieżności, rozproszonej komunikacji i odporności na błędy.[7] W dobie internetu są to cechy bardzo pożądane. Aby spełnić postawione założenia stworzono wirtualną maszynę zwaną BEAM (Bogdan/Björn’s Erlang Abstract Machine)[8]. Programy pisane w Erlangu są wysoce współbieżne ze względu na fakt, że poszczególne funk- cjonalności są wykonywane w ramach lekkich procesów (aktorów)[zob 3.4.2]. Ich cyklem życia zarządza wirtualna maszyna, rozdzielając dostępne zasoby sprzętowe. Aby zapewnić niezawodność, procesy są od siebie odseparowane i niezależne, awaria jednego z nich nie prowadzi do eskalacji problemu. Zbiór takich jednostek może

tworzyć złożone, skalowalne systemy.[9]

**2.3.2 Elixir**

*Elixir* jest młodym językiem programowania działającym na maszynie wirtual- nej *Erlanga*. W przeciwieństwie do *Erlanga*, nie jest to produkt konkretnej firmy, a otwarty projekt, rozwijany przez społeczność entuzjastów.

Celem José Valima, twórcy Elixira, było stworzenie rozszerzalnego oraz przyjaznego programistom języka programowania [10]. Pomimo tego, że Elixir jest bardzo zbli- żony do Erlanga, jest przystępniejszy dla użytkownika. Uporządkowano standar- dową bibliotekę pozbywając się zbędnych, zduplikowanych elementów oraz wpro- wadzono powszechne konwencje ułatwiające tworzenie i utrzymanie kodu. Ponadto *Elixir* wprowadza mechanizmy pozwalające rozszerzać język przy użyciu metapro- gramowania i polimorfizmu.

Dzięki wykorzystaniu BEAM, Elixir może korzystać z narzędzi oraz bibliotek stwo- rzonych na potrzeby Erlanga kodu bez uszczerbku na wydajności[11, 12].

**3 Architektura systemu informatycznego**

Złożoność systemów informatycznych stale rośnie. Wraz ze złożonością zwięk- sza się ich zapotrzebowanie na moc obliczeniową. Biorąc pod uwagę spowalniające prawo Moore’a, sprostanie potrzebom zwiększenia mocy jest znacznie utrudnione. Rozwiązaniem tego problemu jest stosowanie architektur równoległych, procesorów wielordzeniowych i układów wieloprocesorowych. Wzrost zainteresowania kompute- rami wielordzeniowymi niesie dodatkową pracę programistyczną, wymagającą two- rzenia oprogramowania współbieżnego. Taki kod jest znacznie trudniejszy do ana- lizy, ponieważ poza kontekstem aktualnie wykonywanej sekwencji kodu należy wziąć pod uwagę synchronizację z innymi wątkami. Skomplikowanie zbioru częściowo upo- rządkowanych operacji wykonywanych współbieżnie jest znaczenie wyższe niż pro- gramu sekwencyjnego. Dodatkową przeszkodą jest problem pamięci współdzielonej. Dominującym modelem programowania współbieżnego jest wykorzystanie pamięci dzielonej pomiędzy wiele, działających paralelnie wątków. Jeżeli co najmniej dwa z nich jednocześnie próbują uzyskać dostęp jednego obszaru pamięci pojawia się zja- wisko *wyścigu*, które może prowadzić do niespójności i uszkodzenia danych. Istnieją metody zapobiegania takim sytuacjom, najprostsza z nich to stosowanie blokad przy dostępie do współdzielonego obszaru pamięci. Pomimo prostoty koncepcji, stosowa- nie blokad w skomplikowanych systemach może prowadzić do dużej liczby błędów, ponieważ nie jest to podejście naturalne dla deweloperów. Blokady wprowadzają nowe klasy problemów jak *deadlocki* i *livelocki*. Prowadzi to do powstawania opro- gramowania nieodpornego na błędy, które trudno skalować. Ciągle wytwarzane są narzędzia oraz nowe modele współbieżności, które mają ułatwić pracę programi- stom, pomagające rozwiązać wymienione problemy z blokadami [13, 14].

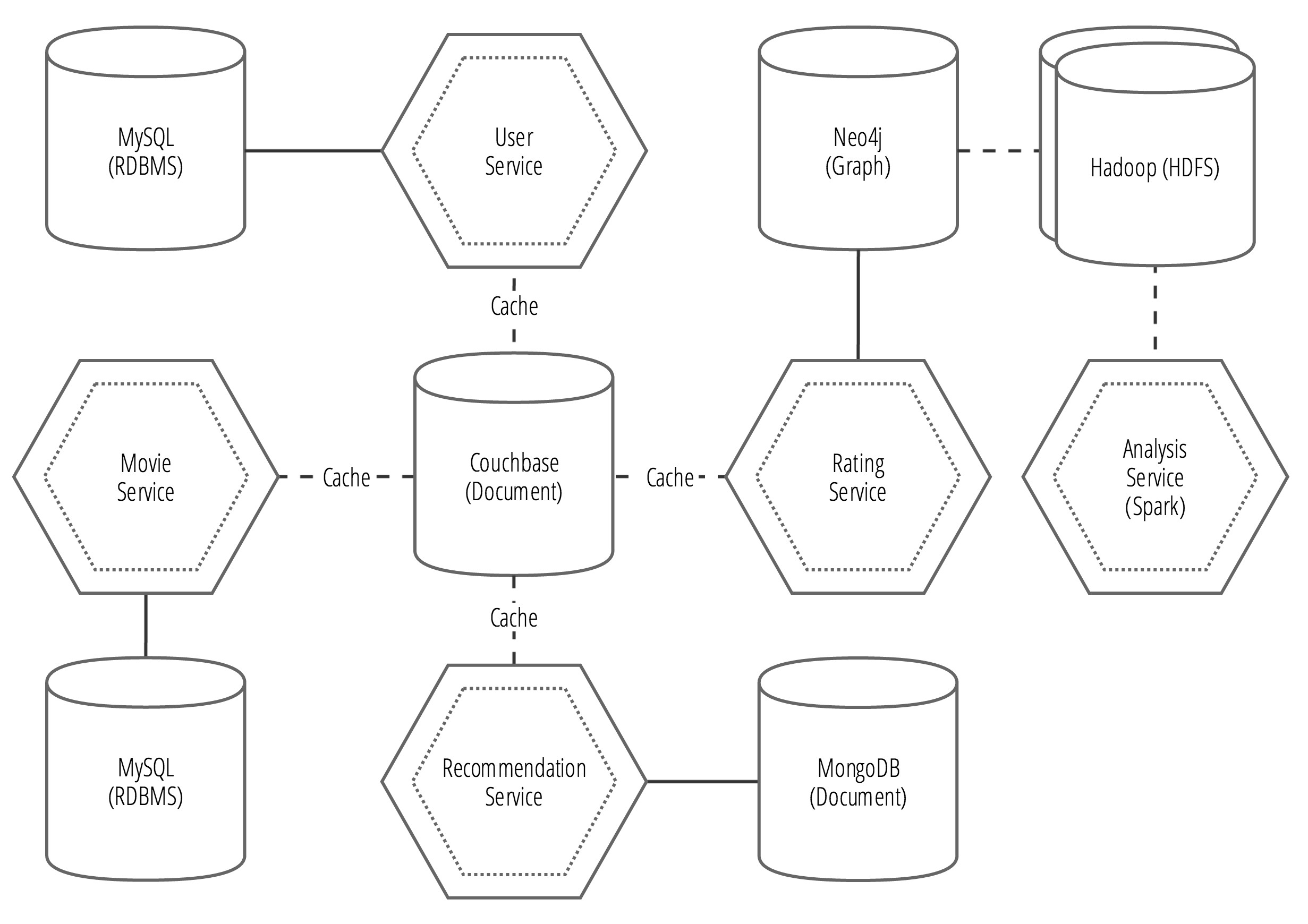
Problem rosnącego zapotrzebowania na moc obliczeniową można rozwiązać również poprzez zastosowanie systemów rozproszonych. Za tym podejściem przemawia wiele czynników, między innymi: geograficzne rozdystrybuowanie źródeł i ujść informacji, wymagania niższego czasu rekcji, niższa cena komputerów, zwiększona niezawod- ność. Z drugiej strony tworzenie systemów rozproszonych niesie ze sobą te same ograniczenia i problemy dostępu do współdzielonych zasobów jak wykorzystanie pamięci współdzielonej. Dodatkową komplikacją w tym przypadku jest tworzenie algorytmów korzystających z wielu maszyn jednocześnie. Klasycznym podejściem jest wykorzystanie czasu jako podstawy opracowywania algorytmów synchronizacji. Wprowadzenie dodatkowego czynnika jakim jest czas, wynika z opóźnień powstałych w na etapie komunikacji pomiędzy procesami działającymi na odrębnych maszynach, jednakże dodatkowa zmienna znacznie zwiększa złożoność i nakład pracy [15].

**3.1 Współczesne trendy**

**3.1.1 Architektura mikroserwisowa**

Wiele uwagi poświęca się w ostatnim czasie tematowi zastosowania architektury mikroserwisowej w projektach systemów informatycznych.

Mikroserwisowy styl architektoniczny reprezentuje podejście do wytwarzania opro- gramowania jako zboru małych usług, każda działająca jako osobny proces, po- rozumiewające się za pomocą lekkich protokołów. Usługi te są zbudowane wokół funkcjonalności biznesowych i każda z nich jest niezależnie wdrażana przez zauto- matyzowane systemy wdrożeniowe. Założeniem jest, aby usługi w jak najmniejszym stopniu wykorzystywały narzędzia scentralizowanego zarządzania. Implementacja każdej z usług, jak również stosowane narzędzia są na tyle niezależne od siebie, że mogą wykorzystywać różne języki programowania i technologie przy założeniu jednolitego interfejsu.



Rysunek 1: Przykład projektu systemu w architekturze mikroserwisowej[16]

Nie ma formalnej definicji architektury mikroserwisowej, lecz są pewne cechy cha- rakterystyczne, które często pojawiają się w tego typu systemach. Usługi często są wykorzystywane jako reużywalne komponenty. Każdy z nich może być uruchomiony niezależnie i wprowadzanie w nich zmian nie prowadzi do ponownego wdrożenia ca- łości aplikacji, a jedynie jednej z jej części. Nie tylko technologia programowania jest niezależna od architektury, to samo tyczy się trwałych składowisk danych. Usługi

nie powinny dzielić jednej bazy danych, tak aby każda z nich była niezależna. Nic nie stoi na przeszkodzie, aby wykorzystać różne technologie przechowywania danych, przykładowo bazę danych relacyjną oraz NoSQL w jednym systemie informatycz- nym.

Taki podział systemu wymaga mechanizmów zewnętrznej komunikacji pomiędzy usługami, których koszt jest dużo wyższy od wywołań wewnątrz aplikacji. W związku z tym funkcjonalności i odpowiedzialności muszą być odpowiednio zapro- jektowane i wydzielone. Celem tego jest minimalizacja koniecznej komunikacji po- między komponentami, co jednakże wiąże się z dodatkowymi nakładami na projek- towanie.

Do komunikacji w systemach rozproszonych wykorzystuje się różne technologie i po- dejścia, których odpowiedni dobór ma istotne znaczenie dla działania całości. Na przestrzeni lat stosowano wiele podejść, przykładowo Korporacyjną Szynę Usług (ang. *Enterprise Service Bus*), która skupia szereg skomplikowanych mechanizmów zarządzania usługami, trasowania wiadomości i transformacji danych. Społeczność wspierająca mikroserwisy proponuje wykorzystywanie prostych, lekkich rozwiązań komunikacyjnych. W tym celu często jest stosowany protokół HTTP przy zastoso- waniu interfejsów REST. [17–19]

**3.2 Java**

Architektura złożonych systemów pisanych w Javie jest silnie związana z *Javą Enterprise Edition*, gdyż definiuje ona szereg standardów dla tworzenia logiki aplika- cyjnej. Proces, w którym ustalane są owe standardy, jest przeprowadzany w ramach *Java Community Process*. Członkami komitetu, jak również osobami udzialającymi się przy tworzeniu propozycji, jest społeczność użytkowników Javy i specjaliści w tej dziedzinie. Wielu z nich to przedstawiciele przedsiębiorstw aktywnie wykorzystują- cych te technologie, np. Credit Suisse, Ericsson, Fujitsu, IBM czy Intel[20]. Poprzez wkład ich oraz wielu innych firm, Java Enterprise Edition jest powszechnie wykorzy- stywana między innymi w branży finansowej czy IT do tworzenia oprogramowania biznesowego.

**3.2.1 Architektura wielowarstwowa**

Java EE definiuje architekturę tworzonych usług jako wielowarstwową aplikację, mającą zapewnić skalowalność, dostępność oraz łatwość zarządzania niezbędną dla systemów biznesowych. Przyjmując takie założenia autorzy Javy EE podzielili lo- gikę aplikacji na komponenty w zależności od ich funkcji. Dzięki temu podziałowi możliwe jest rozdystrybuowanie systemu poprzez umieszczenie każdej z warstw na osobnej maszynie.

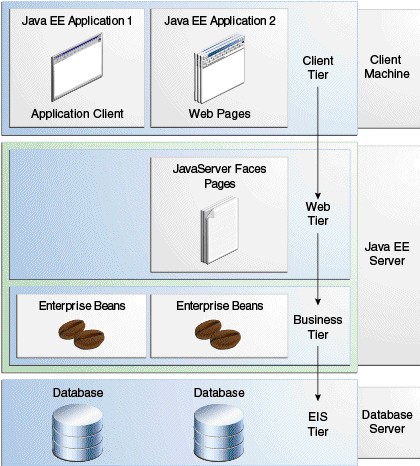
Przewidziano następujące komponenty:

*•* warstwa kliencka na komputerze użytkownika,

*•* warstwa webowa na serwerze Java EE,

*•* warstwa biznesowa na serwerze Java EE,

*•* baza danych na serwerze bazodanowym.



Rysunek 2: Architektura wielowarstwowa [2]

Model ten jest powszechnie stosowany nie tylko dla aplikacji w Javie, ale również przy użyciu wielu innych technologii. [2, 21]

**3.3 JavaScript**

Biorąc pod uwagę fakt, że JavaScript został stworzony do zastosowań w apli- kacjach klienckich, środowisko to nie posiada ustandaryzowanych wzorców projek- towania i tworzenia systemów informatycznych. Ze względu na dużą i różnorodną społeczność wielu programistów o odmiennym podłożu zawodowym, nieustannie po- wstaje szereg bibliotek oraz wiele modeli aplikacyjnych, lecz jak dotąd nie wyłonił się z nich żaden dominujący paradygmat. Jedynym wspólnym ich elementem jest Node.js, lecz biblioteka ta nie dyktuje wzorców architektonicznych, poza posługiwa- niem się zdarzeniami i odwołaniami obsługi tychże.

Programowanie w JavaScript w dużej mierze opiera się na obsłudze zdarzeń. Łatwo to zauważyć na przykładzie aplikacji przeglądarkowych, gdzie zdarzenia są wywo- ływane przez interakcje użytkownika z interfejsem, na przykład kliknięcia myszy

czy przeciąganie obiektów. Z perspektywy projektu języka, w JavaScript wprowa- dzono ułatwienia dla programowania sterowanego zdarzeniami (ang. *event-driven programming*). Jednym z bardziej istotnych jest wprowadzenie funkcji wyższego rzędu. Dzięki nim możliwe jest przekazywanie jako parametrów funkcji innych funk- cji jako odwołania do wykonanej akcji.

1 **f u n c t i o n** h a n d l e C l i c k ( ) {

2 a l e r t ( " B u t t o n c l i c k e d ! " ) ;

3 } ;

4

5 **v a r** b u t t o n = d o c u m e n t . g e t E l e m e n t B y I d ( " b u t t o n " ) ;

6 b u t t o n . a d d E v e n t L i s t e n e r ( " c l i c k " , h a n d l e C l i c k ) ;

Listing 1: Obsługa zdarzeń kliknięcia

W linijce 6 powyższego listingu do obiektu button została dodana obsługa zda- rzenia. Jako pierwszy parametr metody *addEventListener* przekazano typ akcji - kliknięcie (*‘click’* ) oraz drugi parametr będący nazwą funkcji, która zostanie wyko- nana przy wykonaniu zdefiniowanej akcji.

**3.3.1 libuv**

libuv jest to natywna biblioteka stanowiącą warstwę abstrakcji nad różnymi urzą- dzeniami wejścia/wyjścia do wykonywania na nich asynchronicznych operacji. Zo- stała zaprojektowana z myślą o Node.js, lecz znalazła zastosowanie w innych pro- jektach.

W skład jej możliwości wchodzą:

*•* asynchroniczna obsługa połączeń sieciowych poprzez TCP oraz UDP,

*•* asynchroniczna obsługa plików oraz operacji na systemie plików,

*•* zdarzenia systemu plików,

*•* komunikacja między procesowa,

*•* procesy pomne,

*•* pula wątków,

*•* obsługa przerwań,

*•* zegar wysokiej rozdzielczości,

*•* obsługa wątków i ich synchronizacja.

libuv udostępnia użytkownikom 2 interfejsy do pracy z wejściem/wyjściem: uchwyt

(ang. *handle*) oraz zapytanie (ang. *request*).

Uchwyty stanowią obiekty o długim czasie życia, zdolne do przeprowadzania pew- nych operacji w czasie aktywacji, przykładowo uchwyt może implementować przyj- mowanie połączeń na serwerze TCP i być aktywowany za każdym razem gdy pojawia się nowe połączenie.

Zapytanie reprezentuje operacje o krótkim czasie życia. Mogą być wywoływane w cyklu obsługi uchwytu lub samodzielnie. Powyższe abstrakcje służą użytkownikom do interakcji z pętlą zdarzeń (ang. *event loop*).

Pętla wejścia/wyjścia lub też pętla zdarzeń jest kluczową częścią libuv. Odpowiada ona za przetwarzanie wszystkich operacji związanych z wejściem/wyjściem, uży- wając niecodziennego, jednowątkowego i asynchronicznego modelu obsługi tychże operacji. Wszystkie działania sieciowe wykonywane są w jednym wątku posługując się nieblokującymi gniazdami (ang. *socket*), które są cyklicznie odpytywane.

W przeciwieństwie do sieciowego wejścia/wyjścia, obsługa plików jest bazowana na blokującym (synchronicznym) dostępie wykorzystującym pulę wątków. Każdy wą- tek z puli może niezależnie przetwarzać operacje na pliku.

Takie podejście do równoległości jest przykładem wzorca *Reaktor*.

**3.3.2 Wzorzec Reaktor**

Wzorzec *Reaktor* jest zaprojektowany do obsługi zapytań przychodzących rów- nolegle do systemu od jednego lub więcej klientów. Każda funkcjonalność systemu jest reprezentowana przez osobne jednostki przetwarzania, odpowiedzialne za obsługę jedynie zapytań przeznaczonych dla nich. Za podział zapytań pomiędzy jednostki przetwarzania odpowiedzialny jest synchroniczny demultiplekser.

Kluczowymi elementami wzorca Reaktor są: uchwyty, demultiplekser zdarzeń (ang. *event demultiplexer* ), dyspozytor wejściowy (ang. *initiation dispatcher* ) oraz jed- nostki obsługi zdarzeń (ang. *event handler* ).

Uchwyty są zasobami zarządzanymi przez system operacyjny. Wśród nich znajdują się między innymi połączenia sieciowe czy otwarte pliki.

Synchroniczny demultiplekser zdarzeń blokuje nadchodzące zdarzenia w oczekiwa- niu na uchwyty i zwalnia blokadę, kiedy operacja może zostać przeprowadzona na uchwycie bez potrzeby blokowania.

Dyspozytor wejściowy definiuje interfejs do rejestracji, derejestracji i dyspozycji jed- nostek obsługi zdarzeń. Jest on informowany o nowych zdarzeniach w systemie, w wyniku czego wybiera odpowiednią jednostkę obsługi zdarzenia do otrzymanej akcji. Jednostka obsługi zdarzeń implementuje logikę przetwarzania przychodzących zda- rzeń. System rejestruje takie jednostki w dyspozytorze wejściowym dla konkretnych typów zdarzeń. Kiedy jedno z nich zostanie odebrane, dyspozytor rozwiązuje odpo- wiednią jednostkę obsługi zdarzeń i wywołuje jej kod obsługi.

Po zarejestrowaniu wszystkich jednostek obsługi zdarzeń startowana jest pętla obsługi zdarzeń (ang. *event loop*) dyspozytora. Uchwyty zostają powiązane z za- rejestrowanymi z jednostkami obsługi, a demultiplekser czeka na zdarzenia przy- chodzące na uchwytach. Po nadejściu jednego z nich demultiplekser zawiadamia dyspozytora, że uchwyt jest gotowy do rozpoczęcia przetwarzania danych. Ten z kolei, używając typu źródła zdarzenia, wybiera jednostkę obsługi i wywołuje jej

Aplikacja Dyspozytor J. przetwarzania Uchwyt

inicjalizuj Dyspozytora zarejestruj J. przetwarzania

obsłuż zdarzenie

podaj uchwyt wybierz

obsłuż zdarzenie

Rysunek 3: Sekwencja działania Reaktora (źródło: praca własna)

kod.

Takie podejście pociąga za sobą szereg zalet jak i wad. Przede wszystkim zapew- nia łatwą kontrolę nad współbieżnością. Reaktor kolejkuje zdarzenia na poziomie demultipleksowania i delegacji pracy do jednostek obsługi. Jest to zwykle wystar- czające, aby wyeliminować potrzebę stosowania bardziej skomplikowanych metod synchronizacji i blokad w aplikacji. Dodatkowo, wzorzec ten ułatwia rozdzielenie odpowiedzialności pomiędzy komponenty systemu. Demultipleksowanie oraz dele- gacja jest niezależna od aplikacji i może być reużywana w różnych projektach. Część funkcjonalna systemu jest rozdzielona pomiędzy jednostki obsługi zdarzeń i każda z nich jest odpowiedzialna za obsługę konkretnych typów zapytań.

Jednakże ceną przetwarzania jednowątkowego jest brak możliwości wywłaszczenia wątku, jednostki obsługi zdarzeń wykonują pracę nieprzerwanie. Z tego wynika, że żadne z nich nie powinno wykonywać blokujących operacji, ponieważ w takim przy- padku jeden zablokowałby cały proces, ograniczając responsywność systemu. Co więcej, aplikacje napisane z wykorzystaniem wzorca Rektor mogą być trudniejsze do analizy i rozwiązywania błędów, ze względu na odwrócony proces przepływu ste- rowania. Implementacja tego wzorca jest ograniczona do możliwości systemu opera- cyjnego, który musi wspierać nieblokujące operacje. Ten problem można obejść wy- korzystując wiele wątków, w których przetwarzanie odbywa się w sposób blokujący, jednak wyzbywając się korzyści wynikających z braku przełączania kontekstu.[22]

**3.4 Elixir**

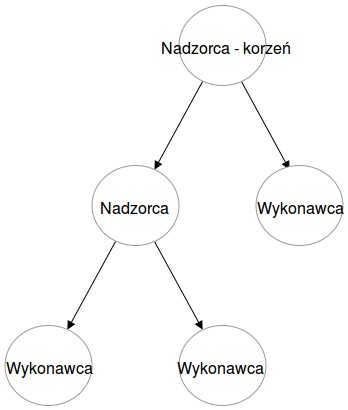
Erlang, a zatem i Elixir, to nie tylko języki programowania, ale również zestaw narzędzi i wzorców programistycznych zwanych *Open Telecom Platform* (OTP).

**3.4.1 Open Telecom Platform**

Mimo swojej nazwy, OTP jest zbiorem bibliotek ogólnego zastosowania, wpro- wadzający standardowe rozwiązania dla powszechnych problemów. Dodatkowo za- wiera narzędzia do rozproszonej komunikacji, wykrywania błędów czy przełado- wywania kodu działającego systemu. OTP wprowadza struktury generyczne dla różnych projektów informatycznych, które mają ułatwić pracę programistom, przy- kładowo ustandaryzowanej struktury plików projektowych, standardową bibliotekę powszechnie stosowanych funkcjonalności oraz zbiór tzw. *behaviour* (ang. *zacho- wanie*) będących wzorcami projektowymi dla różnorodnych systemów. Dzięki temu nowy pracownik zespołu może rozpoznać znane wzorce, co ułatwia zapoznanie się z istniejącą bazą kodu.

Podejście do współbieżności w Erlangu/OTP znacznie różni się od powszechnie sto- sowanego modelu dzielonej pamięci z blokadami. W tym przypadku wykorzystywane są lekkie procesy, projektowane na *model aktorowy*, zarządzane przez wirtualną ma- szynę. Procesy te są od siebie całkowicie odizolowane, nie jest wykorzystana dzielona pamięć. Ze względu na taką separację, komunikacja pomiędzy procesami odbywa się poprzez przekazywanie wiadomości między sobą (ang. *message passing*). Dzie- lone informacje są kopiowane i przesyłane do innego procesu. Dzięki temu nie jest możliwa wielodostępowa modyfikacja danych, co zapobiega powstawaniu anomalii. Jednakże takie podejście nie jest bez wad, gdyż powielanie dużych bloków pamięci może być kosztowne. W celu zapewnienia wysokiej niezawodności i odporności na błędy wprowadzone zostało jedno z ważniejszych *zachowań* OTP, *Supervisor* (ang. *nadzorca*). Jest on odpowiedzialny za tworzenie i monitorowanie stanu procesów wy- konawczych. W sytuacji gdy proces wykonawczy, w wyniku błędu, przerwie pracę, wtedy nadzorca jest w stanie przywrócić poprawne funkcjonowanie systemu odtwa- rzając wykonawcę w normalnym stanie. Nadzorcy, będąc jedynie typem zachowania, mogą być traktowani jako zwykłe procesy wykonawcze, a zatem ich stan może być monitorowany przez innego nadzorcę tworząc w drzewa nadzorcze. W ten sposób system nie posiada jednego punktu odpowiedzialnego za stabilność, a jest on po- dzielony na sekcje.

Równie ważnym elementem OTP jest protokół komunikacji rozproszonej Erlanga. Podobnie jak lekkie procesy wirtualnej maszyny, komunikacja rozproszona bazuje na modelu aktorowym. Poza izolacją procesów na różnych węzłach komunikacji, istotną cechą protokołu jest jego transparentność. Wymiana wiadomości odbywa się w taki sam sposób jak pomiędzy procesami na jednej maszynie. Każdy z rozproszo- nych węzłów jest świadomy istnienia innych węzłów i może się z nimi komunikować tak jakby tworzyły jeden spójny system.[9]



Rysunek 4: Schemat drzewa nadzorczego

**3.4.2 Model aktorowy**

Model aktorowy jest modelem programowania, w którym przetwarzanie jest wy- konywane z natury współbieżnie. Podstawową jednostką wykonawczą tego modelu jest *aktor*.

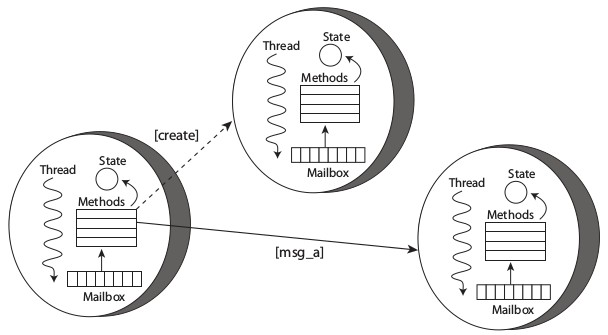
Aktor jest jednostką wykonawczą, która odwzorowuje każdą przychodzącą wia- domość na krotkę składającą się z:

*•* skończonego zbioru komunikatów przesłanych do innego aktora;

*•* nowego zachowania, które wpłynie na odpowiedź następnego przetwarzanego komunikatu;

*•* skończonego zbioru nowo utworzonych aktorów.

Aktorzy, w odróżnieniu od modelu współdzielonych zmiennych, nie dzielą mię- dzy sobą wspólnych obszarów pamięci. Informacje w obliczeniach aktorów mogą być przekazywane, tylko i wyłącznie, poprzez wiadomości. Model ze współdzie- loną pamięcią nie dostarcza żadnych mechanizmów abstrakcji i ukrywania informa- cji. Aby stwierdzić, czy inny obiekt otrzymał dostęp lub zmodyfikował wykorzysty- wane zmienne, wymagane jest zdefiniowanie odpowiedniego protokołu. Co więcej, nie można stwierdzić czy na danych nie zostały wykonane niewłaściwe lub wręcz niepożądane operacje. Jednym ze sposobów radzenia sobie z sytuacjami tego typu jest wykorzystywanie blokad i synchronizacji. Model aktorowy zakłada, że komu- nikacja pomiędzy aktorami nie jest synchroniczna, a akcje stanowią częściowy po-



Rysunek 5: Akcje w modelu aktorowym [23]

rządek. Nadchodzące wiadomości trafiają do skrzynki odbiorczej, gdzie czekają na przetworzenie. Wszystko to ma służyć zapobieganiu blokowania i przetrzymywania zasobów, co może doprowadzić do zakleszczeń (ang. *deadlock* ). Podstawową formą wiedzy zawartą w wiadomości jest istnienie innego *aktora*. Jest to spowodowane tym, że *aktor* A może skomunikować się z *aktorem* B jedynie znając jego *nazwę*. Tę wiedzę może posiąść jeśli został z nią utworzony lub poznał w wyniku przetwarzania nadchodzących wiadomości. Co więcej, komunikacja jest transparentna. Pomimo “świadomości” istnienia innego aktora, nie jest znane jego położenie. Umożliwia to utworzenie systemu aktorów fizycznie rozproszonych pomiędzy wiele maszyn połą- czonych w sieć oraz dynamiczną rekonfigurację topologii.[23–25]

**3.5 Dyskusja**

Poprzez wiele lat rozwoju i kolejne aktualizacje liczba standardów zawartych w pakiecie Java Enterprise Edition zwiększyła się z 10 do 34. Serwery aplikacyjne, które są podstawowym składnikiem budulcowym aplikacji napisanych wykorzystu- jących Javę EE, dążą do pełnej implementacji standardowej specyfikacji oraz wzbo- gacają ją o własne rozwiązania. Niewiele z tych technologii wprowadza korzystne dla architektury mikroserwisowej rozwiązania. Standard Java EE nigdy nie był projektowany z myślą o systemach rozproszonych, poza wydzieleniem warstwy ba- zodanowej i klienckiej. W związku z powyższym, na starcie utrzymujemy w systemie dużą bazę kodu, który nie zostanie wykorzystany. Stosowanie serwerów aplikacyj- nych zakłada wdrażanie na jednym z nich wielu usług jednocześnie. Biorąc po uwagę fakt, że taki serwer pracuje w jednym procesie wirtualnej maszyny Javy, aplikacje

działające pod jego nadzorem mogą zakłócać działanie sobie nawzajem, a w naj- gorszym przypadku jedna z nich może doprowadzić do awarii wszystkich poprzez przerwanie działania samego serwera.

Tworzenie aplikacji monolitycznych jest możliwe przy wykorzystaniu Node.js, lecz zastosowanie jednowątkowego wzorca Rektor nie sprzyja takiemu podejściu. Z tego względu lepszym rozwiązaniem wydaje się wydzielenie poszczególnych funk- cjonalności na osobne programy, dzięki czemu mogą wykorzystać zasoby systemów wieloprocesorowych bez wzajemnej ingerencji w działanie. Z tego powodu, pomimo braku standardowych rozwiązań, zastosowanie architektury mikroserwisowej jest po- pularnym wyborem w środowisku JavaScript i wokół tego modelu rozwijanych jest szereg narzędzi i bibliotek.

Podobnie jak w przypadku języka JavaScript, w Elixirze nie istnieją standardowe wzorce architektoniczne systemów informatycznych. OTP sugeruje jedynie kształt podstawowych elementów budulcowych aplikacji, nie narzucając projektu. Monolityczne oprogramowanie stworzone w Elixirze można podzielić na mniejsze usługi wydzielając niezależne grupy procesów (aktorów) jako działające niezależne programy. Do komunikacji między procesami w architekturze mikroserwisowej popularnym wyborem jest HTTP. Jednakże, przy użyciu Elixira, usługi mogą porozumiewać się przy pomocy protokołu rozproszonej komunikacji wirtualnej maszyny Erlanga. Jest to także ułatwienie dla programistów, gdyż wykorzystanie jego jest transparentne z punktu widzenia użytkownika.[patrz 5] OTP ze względu na pochodzenie ze środowiska Erlanga jest technologią dojrzałą, stabilną oraz aktywnie rozwijaną przez społeczność Elixira. [26]

**4 Wydajność**

Wydajność technologii można mierzyć pod wieloma różnymi aspektami, a na otrzymane wyniki wpływa duża liczba czynników. Przedstawiane środowiska pro- gramistyczne zostały przetestowane pod kątem wydajności, w kilku odmiennych scenariuszach.

Wybrano 4 przypadki w 3 kategoriach:

*•* Duża liczba zapytań:

**–** zapytania nie wymagające dużej mocy obliczeniowej

*•* Czasochłonne obliczenia:

**–** obliczenia na liczbach całkowitych

**–** operacje macierzowe

*•* Ograniczenia wejścia/wyjścia

**–** operacje na systemie plików

Wyżej wymienione sytuacje są powszechnie spotykane w współczesnych syste- mach informatycznych.

Badania przeprowadzono na sprzęcie o następujących parametrach: Maszyna testowa:

*•* Procesor: Intel(R) Core(TM)2 Duo CPU L9400 @ 1.86GHz

*•* Pamięć: 4GB DDR2 800MHz

*•* System operacyjny: GNU/Linux 4.2.5-1-ARCH Maszyna testująca:

*•* Procesor: Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz

*•* Pamięć: 4GB DDR2 800MHz

*•* System operacyjny: GNU/Linux 4.2.5-1-ARCH

Na maszynę testującą celowo wybrano komputer o większej mocy, aby zapew- nić ciągłość testów i uniknąć komplikacji wynikających z niewystarczającej mocy do analizy danych. Komputery podłączono bezpośrednio w sieć o przepustowości

1Gb/s.

Oprogramowanie wykorzystane do implementacji oraz przeprowadzenia testów wy- brano na podstawie powszechności zastosowania. Parametr ten określono w oparciu statystyki dostępne w publicznych repozytoriach [4, 27, 28].

Rozwiązanie w języku Java stworzono na bazie serwera aplikacyjnego WildFly, bę- dącego otwartą dla społeczności dystrybucją serwera JBoss Enterprise Application

Platform firmy RedHat. Wspiera on najnowszą dostępną wersję standardu Java EE

7.

W implementacji kodu JavaScript i Node.js wykorzystano bibliotekę Express.js udo- stępniającą podstawowe mechanizmy wymagane do stworzenia aplikacji interneto- wej, przy zapewnieniu stabilności dla rozwiązań produkcyjnych. Rozwiązanie w języku Elixir korzysta z pakietu Phoenix Framework, gromadzącego dojrzałe tech- nologie, przeznaczone do zastosowań w usługach internetowych.

Do przeprowadzenia badań wykorzystano oprogramowanie Gatling. Udostępnia ono, bazujący na języku Scala, język definiowania testów obciążeniowych.

Wersje oprogramowania:

*•* Java: 1.8.0\_66

*•* Wildfly: 9.0.1.Final

*•* Node.js: v5.3.0

*•* Express.js: 4.13.1

*•* Elixir: 1.1.1

*•* Phoenix Framework: 1.0.3

*•* Scala: 2.11.7

*•* Gatling: 2.1.7

**4.1 Duża liczba zapytań**

Wiele aplikacji używanych na co dzień korzysta z łączności sieciowej. Obserwuje się w ostatnich latach znaczny wzrost na rynku usług zdalnych, a także zwiększenie liczby użytkowników owych usług. Z tego względu na współczesne systemy informatyczne muszą być w stanie obsłużyć znaczne liczby jednoczesnych połączeń i zapytań.

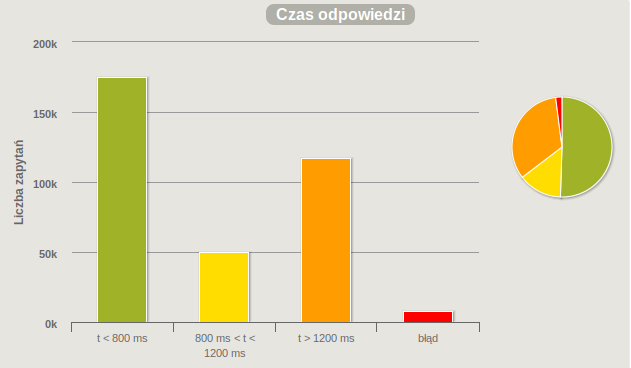
Test polega na wykonaniu metody HTTP GET na serwerze zwracającym pro- sty łańcuch tekstowy. Zasymulowano 350000 użytkowników wykonujących żądanie niezależnie, rozłożonych na przestrzeni 100 sekund.

Implementacje we wszystkich trzech technologiach są trywialne, polegają na pro- stym umieszczeniu łańcucha znaków w odpowiedzi, więc zostały pominięte.

**4.1.1 Java**

Na połowę zapytań otrzymano odpowiedź w czasie poniżej 800 milisekund, a nieznaczna cześć z nich została oznaczona jako błędne.

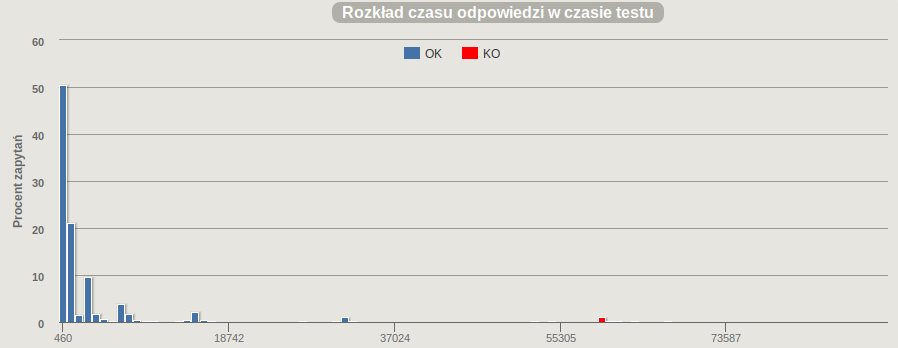
*Liczba aktywnych użytkowników* przedstawia zapytania oczekujące na odpowiedź w danej chwili czasu. Liczba aktywnych użytkowników rośnie w przybliżeniu jedno- stajnie. Odpowiedź na ostatnie żądanie użytkownik otrzymał po 30 sekundach od jego wysłania.



Rysunek 6: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)

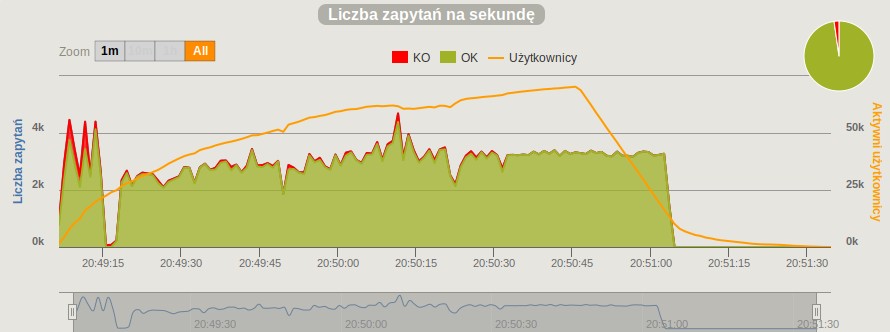


Rysunek 7: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)



Rysunek 8: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca własna)

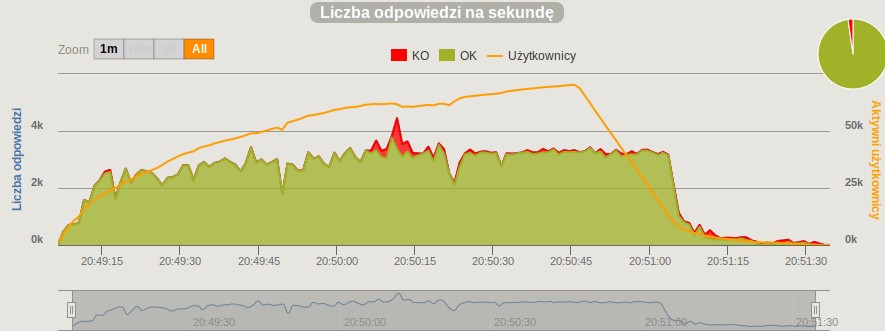
Większość czasów odpowiedzi plasuje się na lewym krańcu wykresu. Występują pojedyncze wyjątki, niewielka część przekroczyła maksymalny czas oczekiwania 60 sekund.



Rysunek 9: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna) Wirtualna maszyna Javy potrzebuje czasu na, tak zwane, *rozgrzanie się*. Wtedy

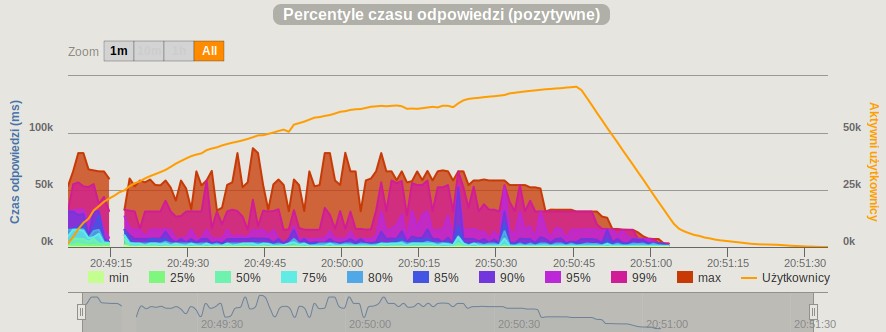
dokonuje wczytania niezbędnych komponentów i automatycznej optymalizacji kodu bajtowego. Zjawisko to można zaobserwować na wykresie 9. Na początku testu wiele żądań zostało oznaczonych jako błędnych, lecz po dokonaniu poprawek przez maszynę wirtualną liczba przyjmowanych zapytań na sekundę ustabilizowała się.

Wykres 10 również odzwierciedla proces optymalizacji. W granicy 60 sekundy testu zauważono zwiększoną liczbę błędnych odpowiedzi. Część zapytań przyjętych przed optymalizacją nie została poprawnie przetworzona.



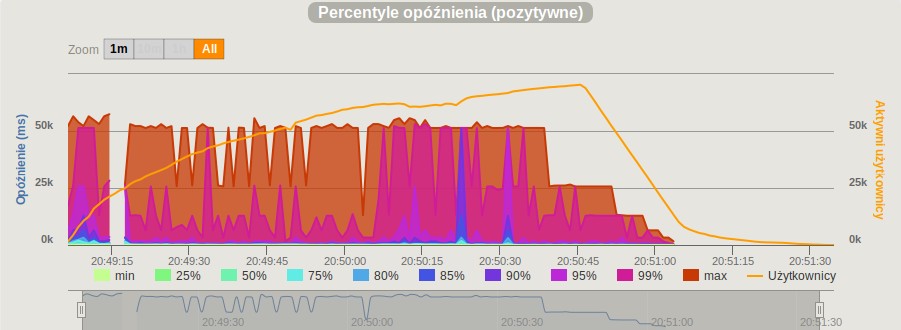
Rysunek 10: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 11: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 12: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Rysunki 11,12 przedstawiają dane jedynie dla poprawnie przetworzonych zapy- tań. *Czas odpowiedzi* jest różnicą czasu otrzymania pełnej odpowiedzi od czasu wy- słania zapytania. *Opóźnienie* stanowi czas, który upłynął od wysłania zapytania do otrzymania pierwszego bajta odpowiedzi. W granicach 20:50:10 zauważono zmianę zarówno w czasie odpowiedzi jak i opóźnieniu. Została ona poprzedzona utratą czę- ści przetwarzanych zapytań, wywołaną uruchomieniem odśmiecacza pamięci (ang. *garbage col lector* ). Po tym zabiegu czasy poprawiły się i zostały zakwalifikowane do niższej kategorii na wykresach percentyli.

Tablica 1: Statystyki Java w teście prostego zapytania

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 350000 | 342147 | 7853 |
| Średnia l./s | 2361,052 | 2308,077 | 52,975 |
| Min | 3 | 3 | 9 |
| 50 percentyl | 602 | 337 | 60032 |
| 75 percentyl | 3026 | 3019 | 61503 |
| 95 percentyl | 25211 | 15071 | 67298 |
| 99 percentyl | 61040 | 32740 | 91059 |
| Max | 91412 | 86686 | 91412 |
| Średnia | 4155 | 2894 | 59091 |
| Odchylenie standardowe | 11121 | 7160 | 13856 |

**4.1.2 JavaScript**

83% zapytań zostało obsłużonych w czasie większym niż 1200 milisekund oraz

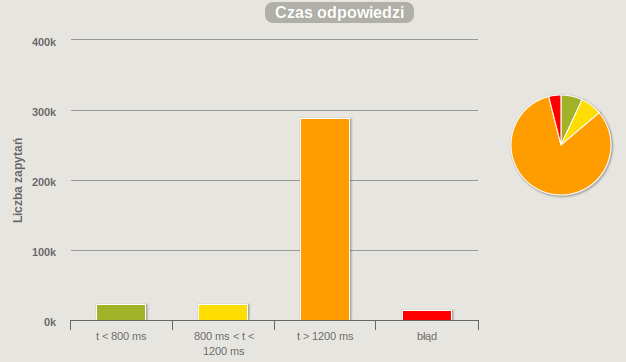
4% z całości oznaczono jako błędne. 24

Liczba aktywnych użytkowników rośnie liniowo i po otrzymaniu ostatniego żą- dania również liniowo spada obsługując oczekujących użytkowników.

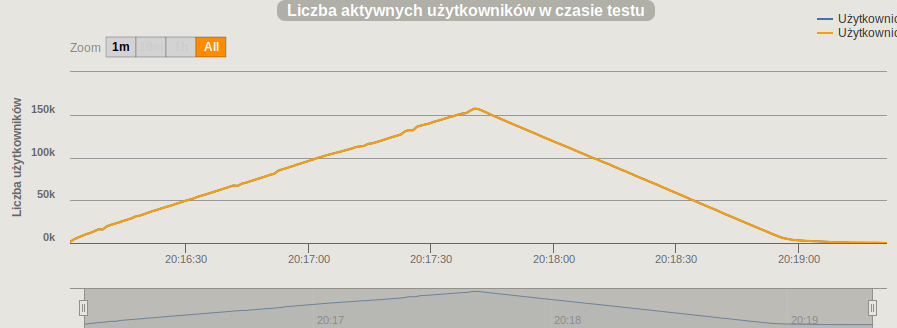
Liczba przyjmowanych zapytań na sekundę utrzymuje stały poziom w czasie trwania testu. Zauważono małe zaburzenia stanowiące nieznaczną część całości te- stu.

Podobnie jak przyjmowanie, przetwarzanie i czas odpowiedzi na żądania został utrzymany na stałym poziomie. Jednakże liczba zapytań przekroczyła możliwości

obsługi, w związku z czym po 60 sekundach zauważono wzrost w ilości błędów.

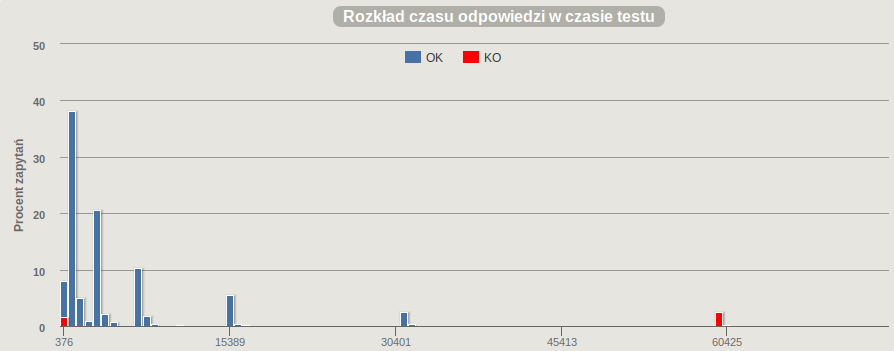


Rysunek 13: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)

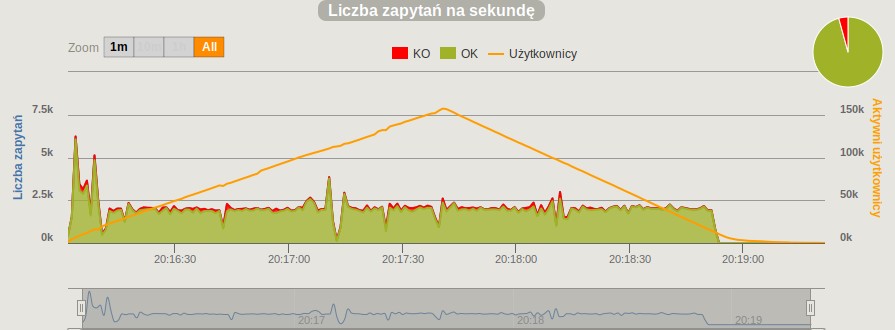


Rysunek 14: Wykres aktywnych użytkowników w całym czasie trwania testu (źródło:

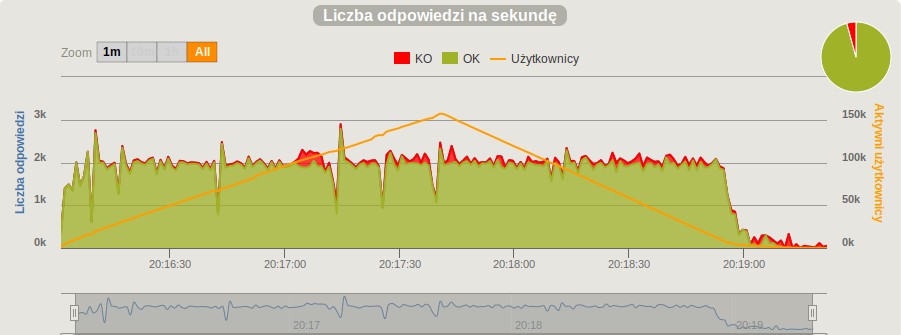
praca własna)



Rysunek 15: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

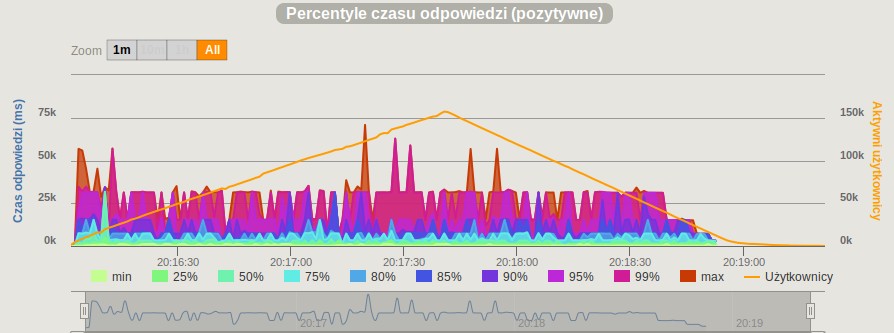


Rysunek 16: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



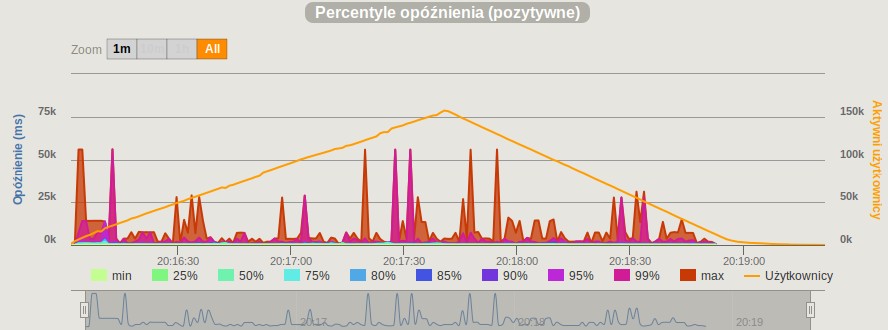
Rysunek 17: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 18: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 19: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 2: Statystyki JavaScript w teście prostego zapy- tania

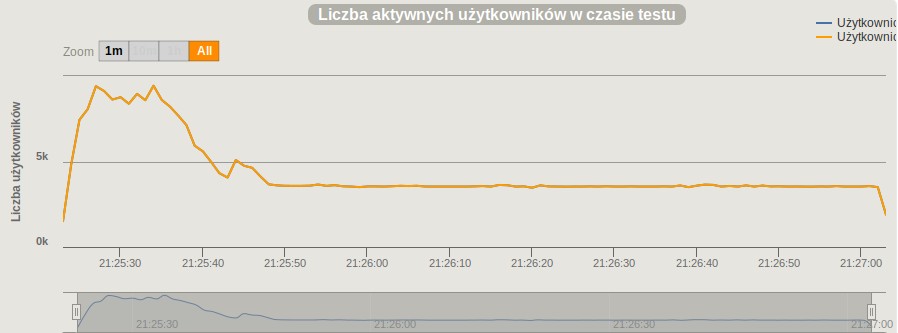
|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 350000 | 335176 | 14824 |
| Średnia l./s | 1741,328 | 1667,575 | 73,753 |
| Min | 1 | 1 | 8 |
| 50 percentyl | 2039 | 1932 | 60006 |
| 75 percentyl | 4826 | 4011 | 60011 |
| 95 percentyl | 31244 | 15422 | 60041 |
| 99 percentyl | 60012 | 31353 | 60613 |
| Max | 75062 | 70845 | 75062 |
| Średnia | 5847 | 4462 | 37180 |
| Odchylenie standardowe | 10805 | 6237 | 29181 |

**4.1.3 Elixir**



Rysunek 20: Wykres czasu odpowiedzi na zapytania (źródło: praca własna) Spośród wszystkich zapytań, na 93% otrzymano odpowiedzi w czasie mniejszym

niż 800ms oraz nie wykazano żadnych błędów.

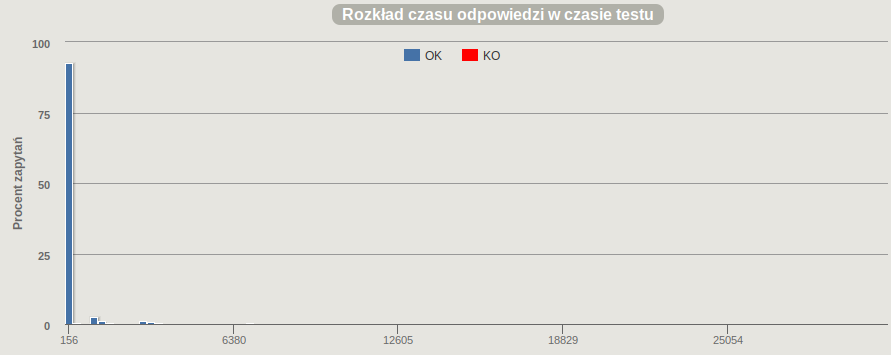


Rysunek 21: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)

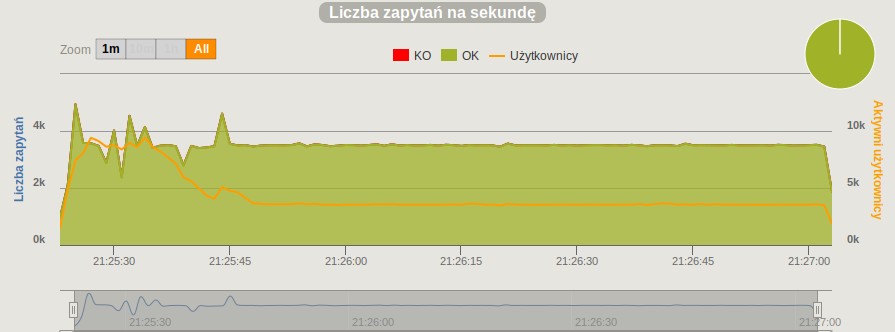
Liczba aktywnych użytkowników stabilizuje się po 25 sekundach od rozpoczęcia testu.

Na wykresach 25 oraz 26 wyraźnie widać czas stabilizacji aplikacji. Po nim aplikacja zaczęła przyjmować zapytania na stałym poziomie.

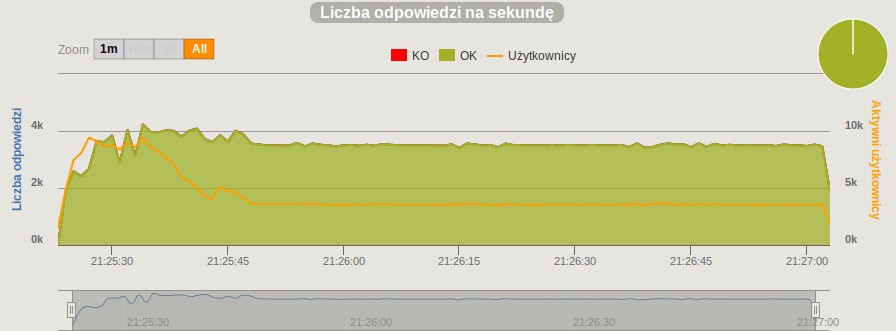
Wykres liczby odpowiedzi odpowiada wykresowi liczby zapytań, stała liczba od- powiedzi na sekundę i brak błędów.



Rysunek 22: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

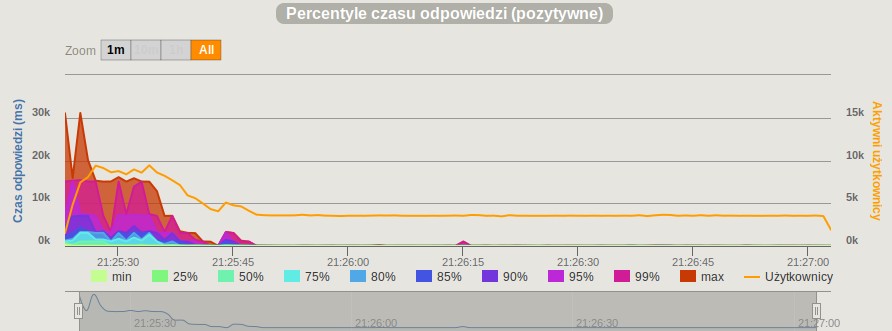


Rysunek 23: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



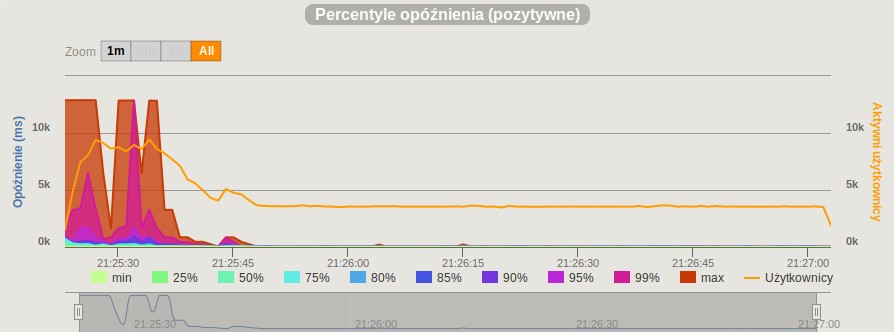
Rysunek 24: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 25: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 26: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 3: Statystyki Elixir w teście prostego zapytania

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 350000 | 350000 | 0 |
| Średnia l./s | 3498.74 | 3498.74 | - |
| Min | 0 | 0 | - |
| 50 percentyl | 9 | 9 | - |
| 75 percentyl | 26 | 26 | - |
| 95 percentyl | 1071 | 1071 | - |
| 99 percentyl | 4270 | 4270 | - |
| Max | 31123 | 31123 | - |
| Średnia | 223 | 223 | - |
| Odchylenie standardowe | 1107 | 1107 | - |

**4.2 Czasochłonne obliczenia**

Współczesne systemy informatyczne wykonują wiele skomplikowanych operacji. Ważna jest nie tylko możliwość obsługi dużej liczby zapytań, ale także wykorzysta- nie mocy obliczeniowej sprzętu do wykonywania operacji.

Celem tego testu jest sprawdzenie wydajności technologii przy równoczesnym do- stępie wielu użytkowników, jednocześnie testując wsparcie dla dużych liczb.

Test polega na wykonaniu metody HTTP GET na serwerze zwracającym 100000 element ciągu Fibonacciego. Zasymulowano 1000 użytkowników wykonujących za- pytanie niezależnie, rozłożonych na przestrzeni 100 sekund.

**4.2.1 Java**

1 **i m p o r t** j a v a . m a t h . B i g D e c i m a l ;

2

3 **p u b l i c c l a s s** F i b o n a c c i {

4 **p u b l i c s t a t i c** B i g D e c i m a l c a l c u l a t e N t h ( **l o n g** n ) {

5 B i g D e c i m a l r e s u l t = **n e w** B i g D e c i m a l ( n ) ;

6 **i f** ( n < 2 ) {

7 **r e t u r n** r e s u l t ;

8 }

9

10 B i g D e c i m a l n 1 = **n e w** B i g D e c i m a l ( 0 ) ;

11 B i g D e c i m a l n 2 = **n e w** B i g D e c i m a l ( 1 ) ;

12 n - -;

13 **w h i l e** ( n > 0 ) {

14 r e s u l t = n 1 . a d d ( n 2 ) ;

15 n 1 = n 2 ;

16 n 2 = r e s u l t ;

17 n - -;

18 }

19

20 **r e t u r n** r e s u l t ;

21 }

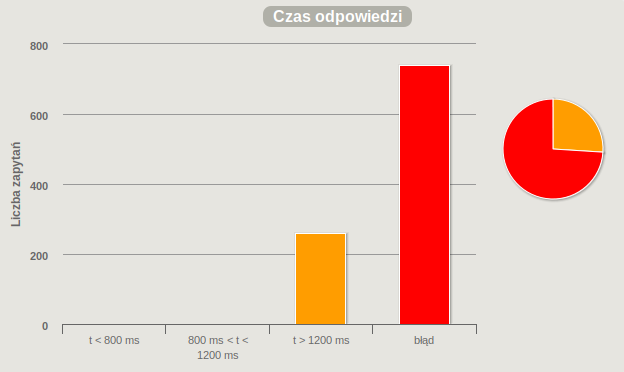
22 }

Listing 2: Java - obliczanie n-tego elementu ciągu Fibonacciego

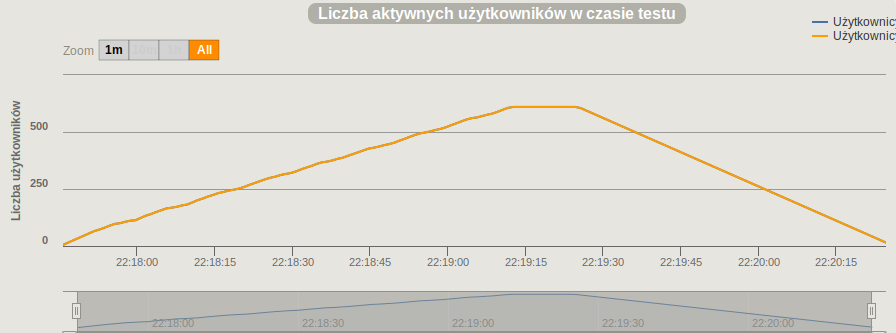
W implementacji zastosowano iteracyjny algorytm wyznaczania n-tego elementu ciągu Fibonacciego. Wybrano tę wersję algorytmu ze względu na możliwość wy- stąpienia przepełnienia stosu przy wykorzystaniu jej rekurencyjnego odpowiednika. Do przechowywania dużych wartości całkowitych wykorzystano klasę BigDecimal, należącą do standardowej biblioteki Javy.

74% zapytań zostało oznaczonych jako błędne z powodu przekroczenia limitu oczekiwania wynoszącego 60 sekund.

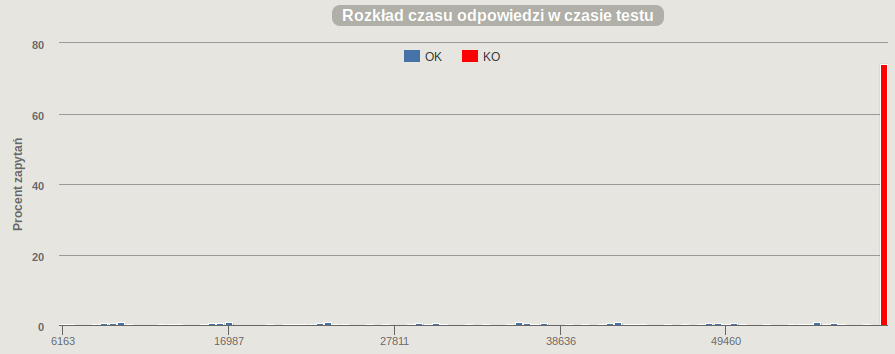
Po upłynięciu 1/3 testu serwer przestał przyjmować zapytania. Wynika to z



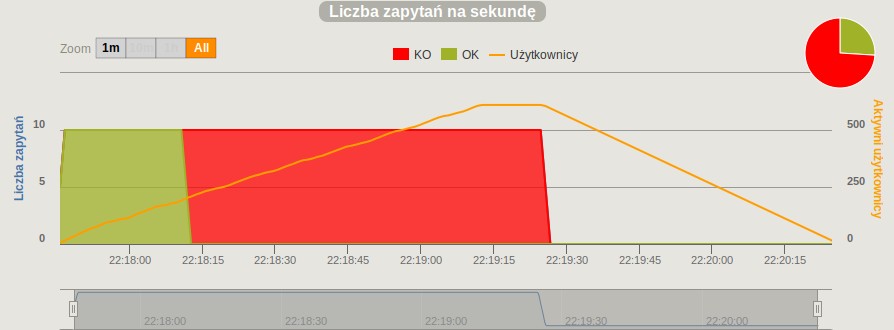
Rysunek 27: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)



Rysunek 28: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)

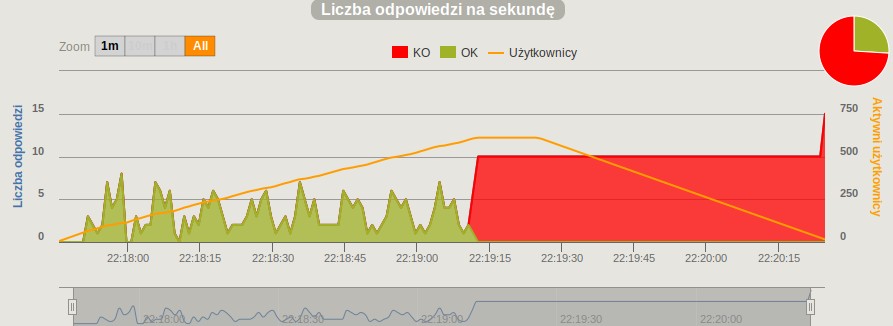


Rysunek 29: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)



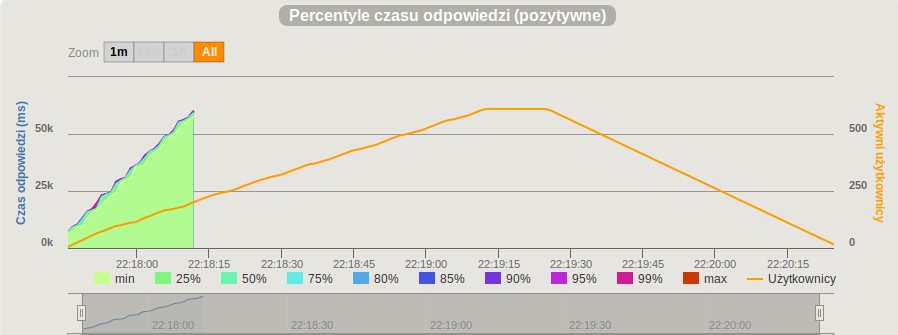
Rysunek 30: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)

przyjętego modelu współbieżności. Każde nadchodzące zapytanie jest obsługiwane w nowym wątku lub **jest** reużywany wątek używany uprzednio. Przy takim natęże- niu wymagających czasowo żądań, osiągnięto limit wykorzystywanych wątków. W związku z tym kolejne nadchodzące zapytania nie mogły zostać przetworzone.



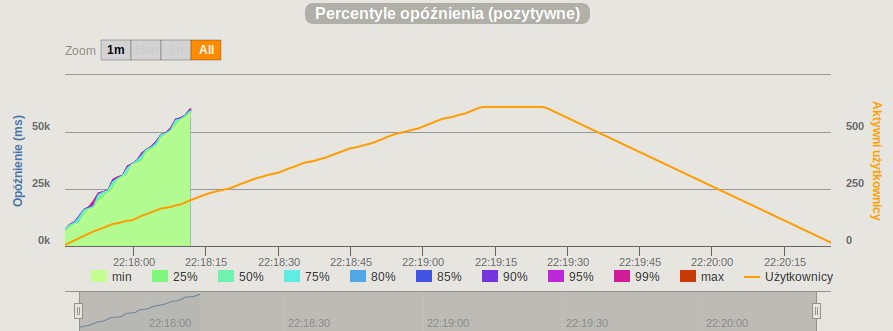
Rysunek 31: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 32: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 33: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 4: Statystyki Java w teście z wykorzystaniem czasochłonnych obliczeń

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 1000 | 260 | 740 |
| Średnia l./s | 6,253 | 1,626 | 4,627 |
| Min | 5892 | 5892 | 60000 |
| 50 percentyl | 60011 | 32751 | 60011 |
| 75 percentyl | 60011 | 45996 | 60011 |
| 95 percentyl | 60011 | 56091 | 60011 |
| 99 percentyl | 60011 | 58727 | 60012 |
| Max | 60014 | 59941 | 60014 |
| Średnia | 52915 | 32721 | 60010 |
| Odchylenie standardowe | 14326 | 15440 | 1 |

**4.2.2 JavaScript**

1 **v a r** B i g N u m = r e q u i r e ( ’ b i g n u m ’ ) ;

2

3 **f u n c t i o n** c a l c u l a t e N t h ( n ) {

4 **v a r** r e s u l t = B i g N u m ( n ) ;

5 **i f** ( n < 2 ) {

6 **r e t u r n** r e s u l t ;

7 }

8

9 **v a r** n 1 = B i g N u m ( 0 ) ;

10 **v a r** n 2 = B i g N u m ( 1 ) ;

11 n -= 1 ;

12 **w h i l e** ( n > 0 ) {

13 r e s u l t = n 1 . a d d ( n 2 ) ;

14 n 1 = n 2 ;

15 n 2 = r e s u l t ;

16 n -= 1 ;

17 }

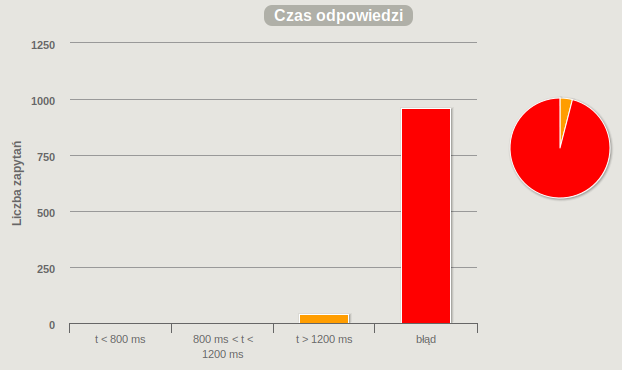
18

19 **r e t u r n** r e s u l t ;

20 }

Listing 3: JavaScript - obliczanie n-tego elementu ciągu Fibonacciego

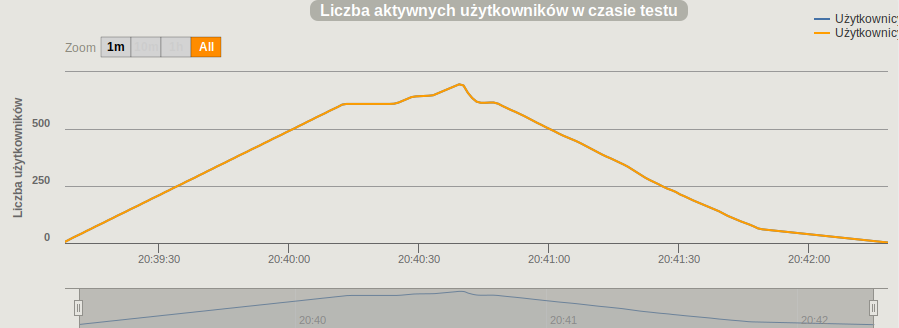
Powyższa implementacja jest translacją wersji w języku Java. JavaScript nie po- siada w standardzie wsparcia dla dużych liczb całkowitych. Do ich przechowywania wykorzystano bibliotekę *bignum*.



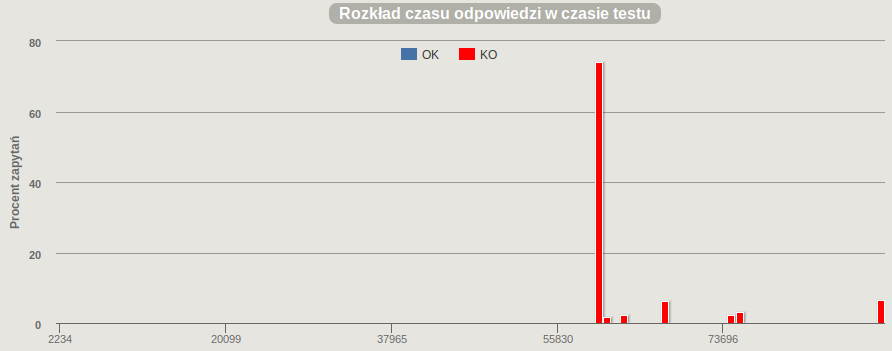
Rysunek 34: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)

96% zapytań zostało oznaczonych jako błędnych błędne ze względu na przekroczenie czasu żądania wynoszącego 60 sekund. Wykorzystana biblioteka *bignum* jest im-

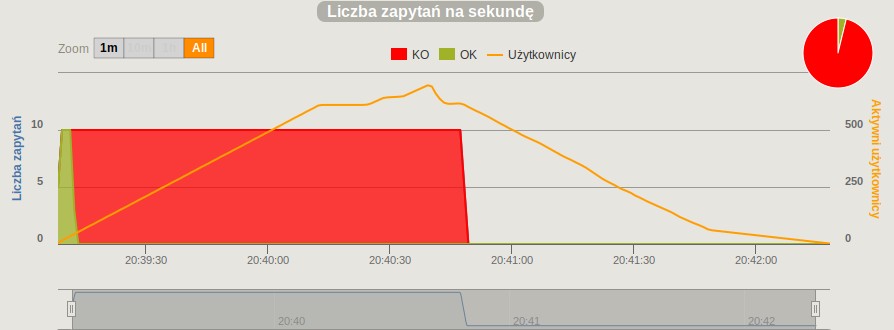
plementacją natywną. Ze względu na charakterystykę zadania, wykonywanie jest wiele operacji na dużych liczbach. Z tego powodu silnik V8 języka JavaScript jest zmuszony do wykonywania ciągłych odwołań do natywnego kodu, co znaczenie spo- walnia pracę aplikacji.



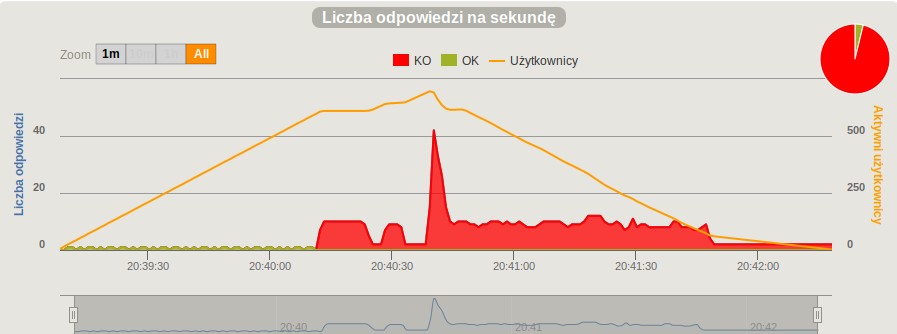
Rysunek 35: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)



Rysunek 36: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

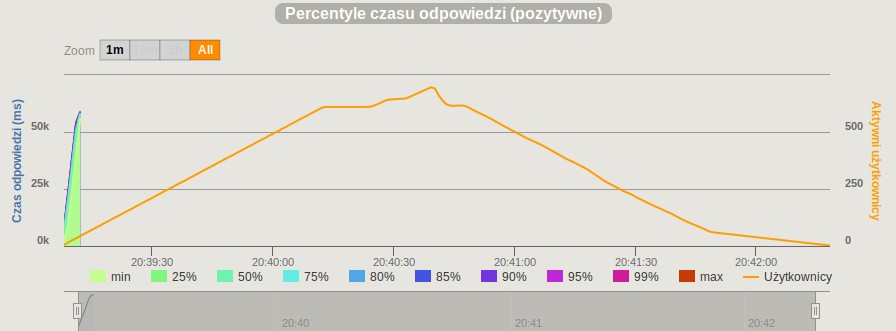


Rysunek 37: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



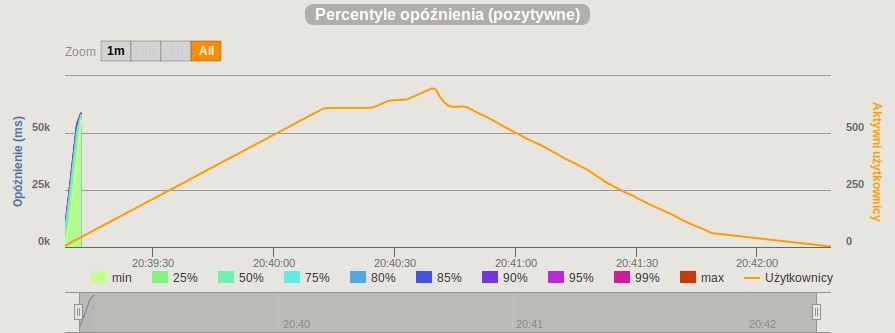
Rysunek 38: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 39: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 40: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 5: Statystyki JavaScript w teście z wykorzysta- niem czasochłonnych obliczeń

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 1000 | 38 | 962 |
| Średnia l./s | 5,249 | 0,199 | 5,05 |
| Min | 1787 | 1787 | 60000 |
| 50 percentyl | 60005 | 30198 | 60004 |
| 75 percentyl | 60006 | 44441 | 60006 |
| 95 percentyl | 91075 | 55782 | 91075 |
| 99 percentyl | 91104 | 58082 | 91104 |
| Max | 91115 | 58662 | 91115 |
| Średnia | 62260 | 30188 | 63527 |
| Odchylenie standardowe | 10900 | 16884 | 8367 |

**4.2.3 Elixir**

1 **d e f m o d u l e** T e s t . C a l c u l a t i o n . F i b o n a c c i **d o**

2 **d e f** c a l c u l a t e \_ n t h ( 0 ) , **d o** : 0

3 **d e f** c a l c u l a t e \_ n t h ( 1 ) , **d o** : 1

4 **d e f** c a l c u l a t e \_ n t h ( n ) , **d o** : f i b (0 , 1 , n - 2 )

5

6 **d e f p** f i b ( \_ , prv , -1) , **d o** : p r v

7 **d e f p** f i b ( p r v p r v , prv , n ) **d o**

8 n e x t = p r v + p r v p r v

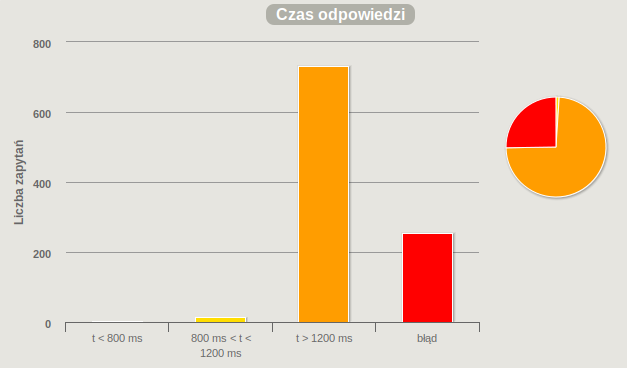
9 f i b ( prv , n e x t , n - 1 )

10 **e n d**

11 **e n d**

Listing 4: Elixir - obliczanie n-tego elementu ciągu Fibonacciego

W języku Elixir, z racji jego funkcyjnego charakteru, nie istnieje pojęcie pę- tli. Z tego względu wykorzystano rekurencyjną wersję algorytmu. Użyto *rekurencji ogonowej* (*rekurencji prawostronnej*, ang. *tail cal l* ), która umożliwia automatyczną optymalizację kodu rekurencyjnego do wersji iteracyjnej. Zabieg ten zwiększa wy- dajność i eliminuje zagrożenie przepełnienia stosu.



Rysunek 41: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)

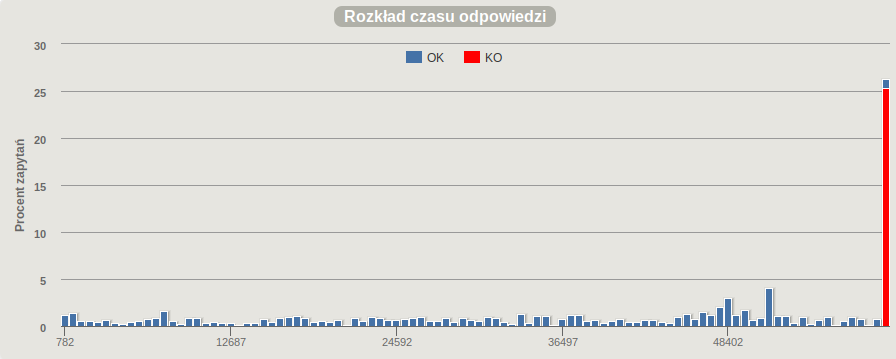
1/3 zapytań została oznaczona jako błędne ze względu na przekroczenie maksy- malnego czasu żądania wynoszącego 60 sekund. Poprawne odpowiedzi otrzymano w czasie powyżej 1200 milisekund. Jedynie nieznaczną część z nich otrzymano poniżej

1200 milisekund.

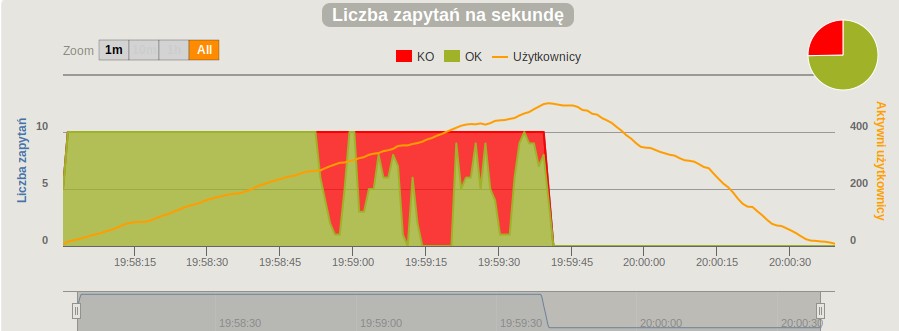
W połowie testu obciążony system ograniczył przyjmowanie nadchodzących za- pytań czego rezultatem są zwrócone błędy.



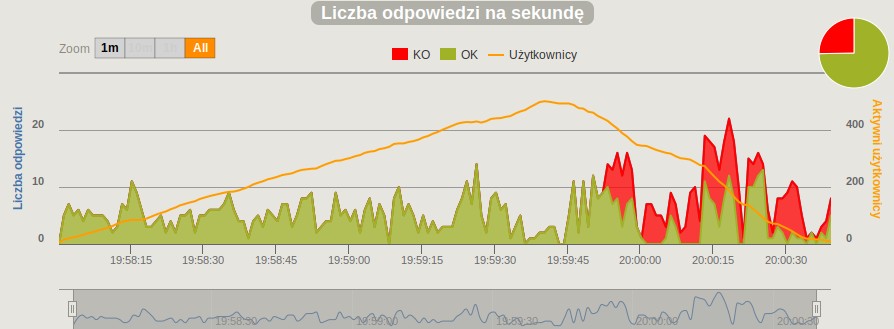
Rysunek 42: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)



Rysunek 43: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

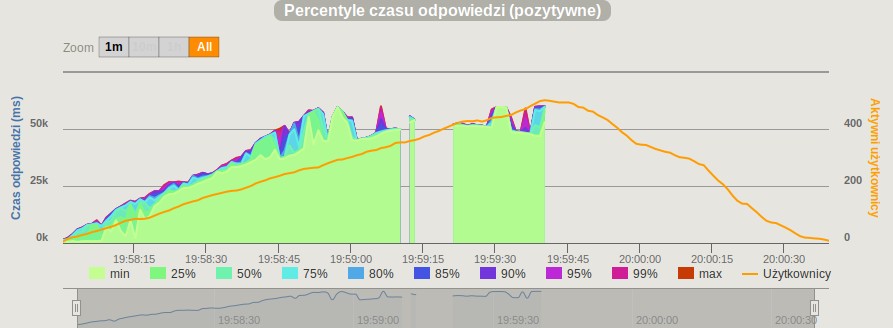


Rysunek 44: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



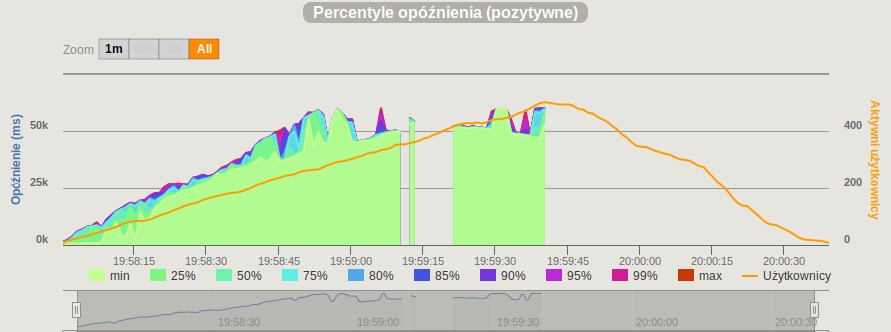
Rysunek 45: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 46: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 47: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 6: Statystyki Elixir w teście z wykorzystaniem czasochłonnych obliczeń

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 1 000,00 | 747,00 | 253,00 |
| Średnia l./s | 6,26 | 4,68 | 1,585 |
| Min | 484 | 484 | 60008 |
| 50 percentyl | 46684 | 35390 | 60009 |
| 75 percentyl | 60008 | 48516 | 60009 |
| 95 percentyl | 60008 | 56588 | 60009 |
| 99 percentyl | 60009 | 59529 | 60010 |
| Max | 60010 | 59975 | 60010 |
| Średnia | 40075 | 33324 | 60008 |
| Odchylenie standardowe | 18823 | 17152 | 0 |

**4.3 Operacje na zbiorach danych**

Zadaniem testu 4.2 jest przetestowanie dużej liczby wykonywanych operacji na każde otrzymane żądanie. Celem tego testu jest porównanie każdej z wybranych technologii w kategorii możliwości manipulowania danymi pod obciążeniem.

Test polega na wykonaniu metody HTTP POST, w ciele której umieszczono wygenerowaną macierz liczb całkowitych o rozmiarze 100 na 100. Zadaniem ser- wera jest zwrócenie w odpowiedzi transponowanej macierzy. Zasymulowano 10000 użytkowników wykonujących zapytanie niezależnie, rozłożonych na przestrzeni 100 sekund.

**4.3.1 Java**

1 **p u b l i c c l a s s** M a t r i x {

2 **p u b l i c s t a t i c i n t** [ ] [ ] t r a n s p o s e ( **i n t** [ ] [ ] m a t r i x ) {

3 **i n t** x D i m e n s i o n = m a t r i x . l e n g t h ;

4 **i n t** y D i m e n s i o n = m a t r i x [ 0 ] . l e n g t h ;

5

6 **i n t** [ ] [ ] r e s u l t = **n e w i n t** [ y D i m e n s i o n ] [ x D i m e n s i o n ] ;

7 **f o r** ( **i n t** r o w = 0 ; r o w < x D i m e n s i o n ; r o w + + ) {

8 **f o r** ( **i n t** c o l = 0 ; c o l < y D i m e n s i o n ; c o l + + ) {

9 r e s u l t [ c o l ] [ r o w ] = m a t r i x [ r o w ] [ c o l ] ;

10 }

11 }

12

13 **r e t u r n** r e s u l t ;

14 }

15 }

Listing 5: Java - transpozycja macierzy

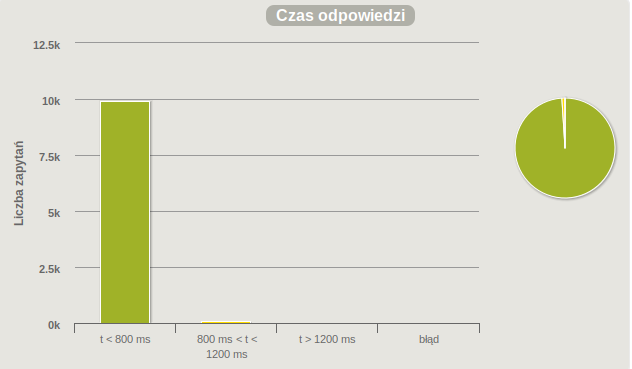
Powyższy kod pobiera wymiary *x, y* przekazanej macierzy *a*, aby utworzyć nową macierz *b* o wymiarach *y, x*. Następnie, wiersz po wierszu i element po elemencie z każdego wiersza, wartości macierzy *a* są przepisywane do macierzy *b*, tak aby *axy* = *byx*

Korzystając z implementacji w języku Java żadne z zapytań nie zwróciło błędu.

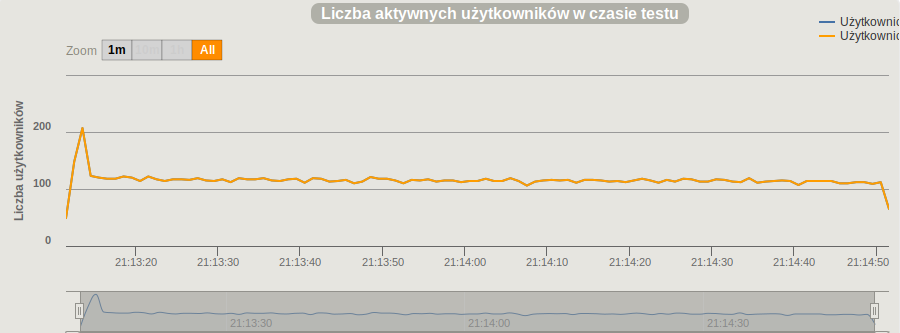
Na 99% żądań odpowiedź otrzymano poniżej 800 milisekund, a jedynie 83 z nich przetworzono powyżej tej granicy.

Zauważono ubytek, a zaraz po nim skok liczby odpowiedzi na sekundę w począt- kowej fazie testu. Po wstępnej inicjalizacji aplikacji, każdy nowy użytkownik został obsłużony na bieżąco.

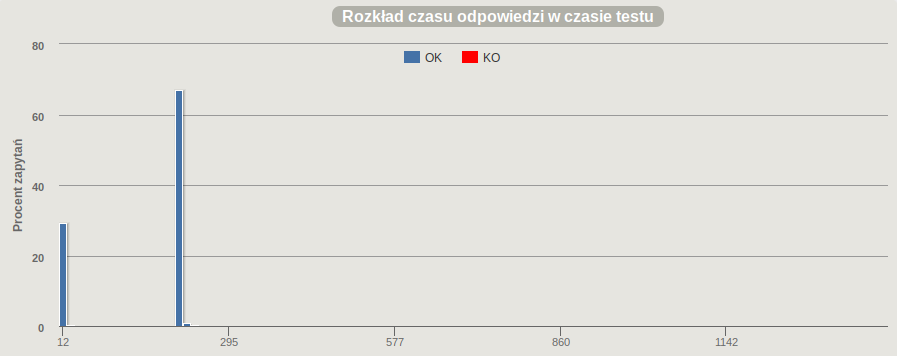
Wystąpiła niewielka różnica w czasach odpowiedzi i opóźnienia pomiędzy po- szczególnymi zapytaniami w czasie trwania testu.



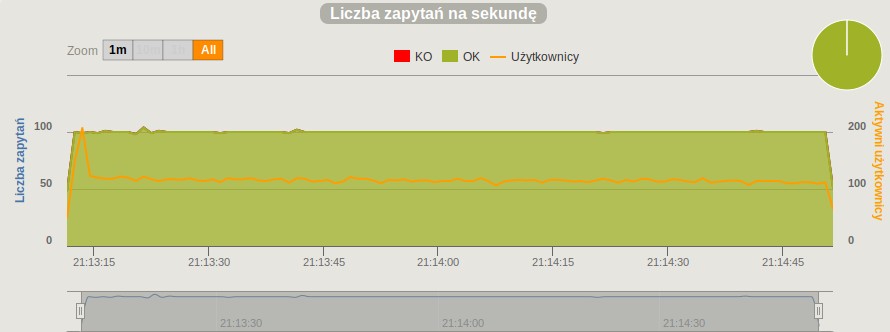
Rysunek 48: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)



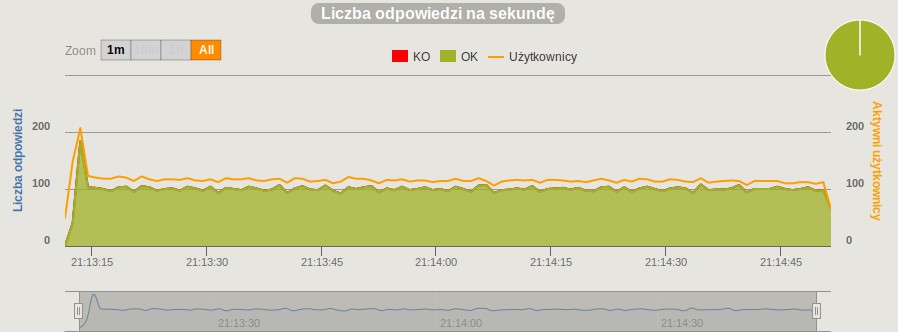
Rysunek 49: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)



Rysunek 50: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

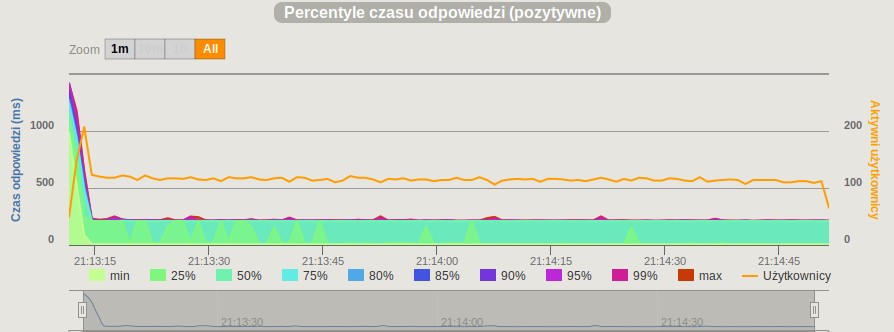


Rysunek 51: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



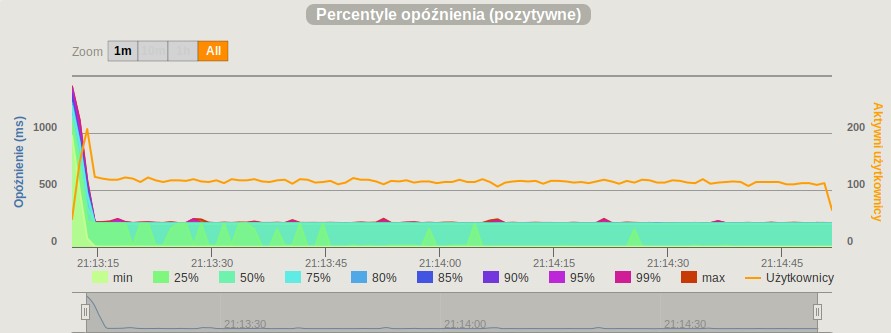
Rysunek 52: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 53: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 54: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 7: Statystyki Java w teście z wykorzystaniem zbiorów danych

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 10000 | 10000 | 0 |
| Średnia l./s | 99,785 | 99,785 | - |
| Min | 5 | 5 | - |
| 50 percentyl | 209 | 209 | - |
| 75 percentyl | 210 | 210 | - |
| 95 percentyl | 214 | 214 | - |
| 99 percentyl | 815 | 815 | - |
| Max | 1418 | 1418 | - |
| Średnia | 162 | 162 | - |
| Odchylenie standardowe | 137 | 137 | - |

**4.3.2 JavaScript**

1 **f u n c t i o n** t r a n s p o s e ( m a t r i x ) {

2 **v a r** x D i m e n s i o n = m a t r i x . l e n g t h ;

3 **v a r** y D i m e n s i o n = m a t r i x [ 0 ] . l e n g t h ;

4

5 **v a r** r e s u l t = [ ] ;

6 f o r ( **v a r** r o w = 0 ; r o w < x D i m e n s i o n ; r o w + + ) {

7 **v a r** n e w \_ r o w = [ ] ;

8 f o r ( **v a r** c o l = 0 ; c o l < y D i m e n s i o n ; c o l + + ) {

9 n e w \_ r o w . p u s h ( m a t r i x [ c o l ] [ r o w ] ) ;

10 }

11 r e s u l t . p u s h ( n e w \_ r o w ) ;

12 }

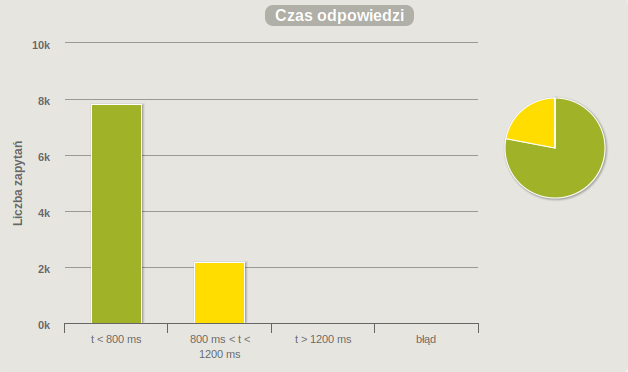
13

14 **r e t u r n** r e s u l t ;

15 }

Listing 6: JavaScript - transpozycja macierzy

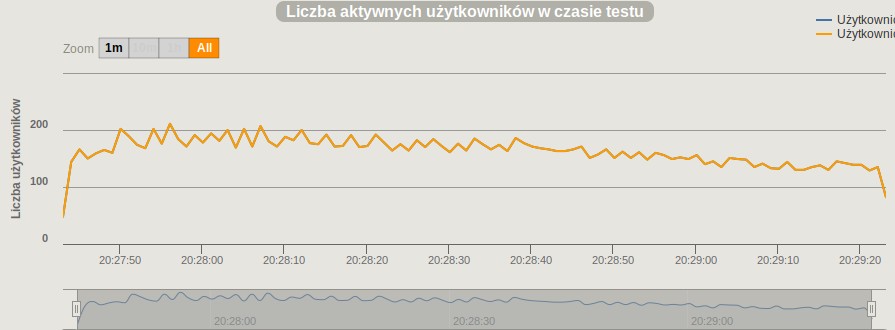
Implementacja ta jest translacją z języka Java, jednak każdy z wierszy jest two- rzony i uzupełniany niezależnie, a następnie dołączany na koniec macierzy wyniko- wej.



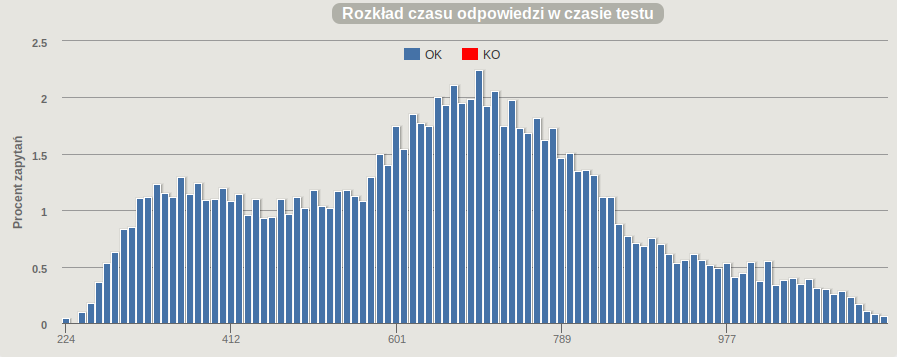
Rysunek 55: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)

78% ze wszystkich zapytań zostało obsłużonych poniżej 800 milisekund oraz żadne nie przekroczyło granicy 1200 milisekund czasu odpowiedzi.

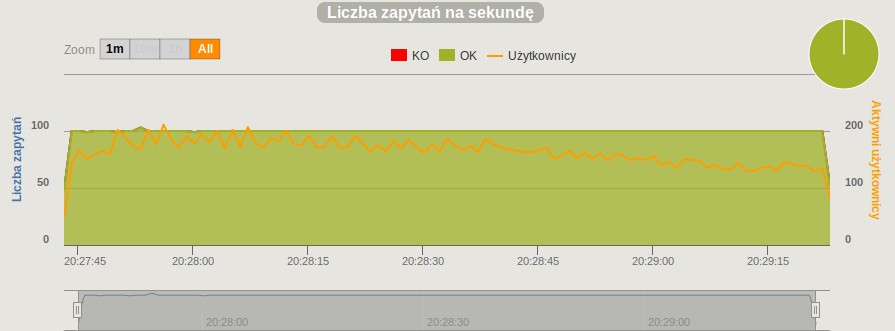
Wszystkie żądania przyjęto niezwłocznie do obsługi, zaś wykres czasu odpowiedzi przedstawia niemalże cykliczne skoki. Node.js operuje jednym wątkiem w jednym procesie. W czasie testu system operacyjny starał się zbalansować pracę obydwu dostępnych rdzeni procesora przenosząc obciążenie pomiędzy nimi.



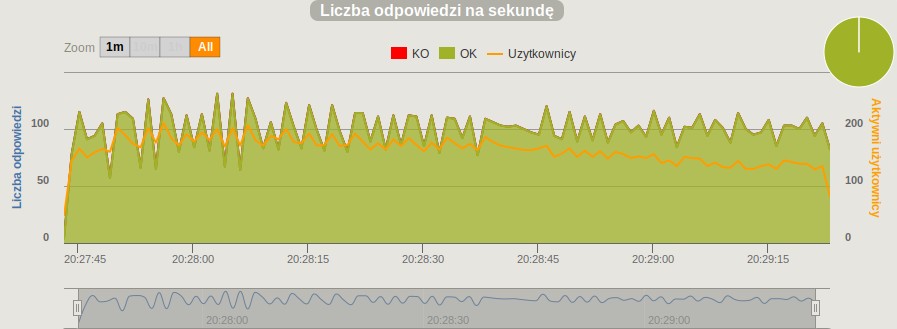
Rysunek 56: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)



Rysunek 57: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

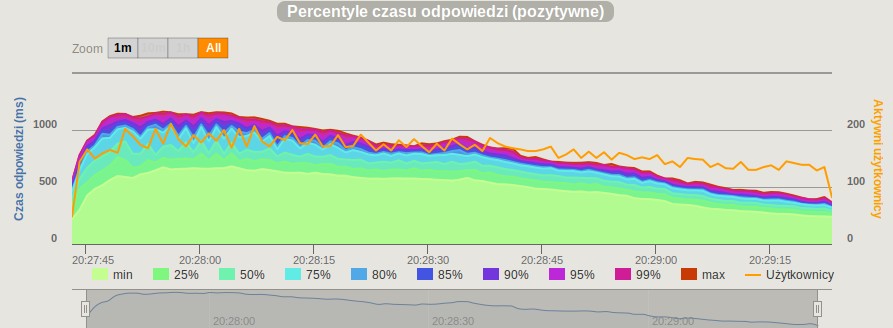


Rysunek 58: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



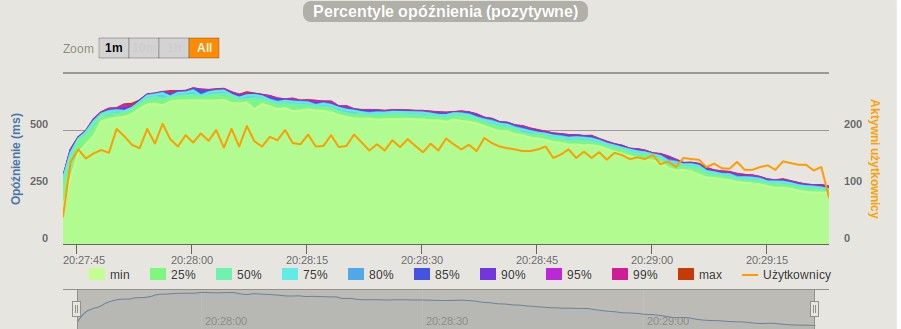
Rysunek 59: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 60: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 61: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Na podstawie wykresów 60 oraz 61 można zaobserwować wpływ optymalizacji przetwarzania zapytań przez silnik V8. Czas odpowiedzi, jak i opóźnienie wyraźnie spada podczas trwania testu.

Tablica 8: Statystyki JavaScript w teście z wykorzysta- niem zbiorów danych

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 10000 | 10000 | 0 |
| Średnia l./s | 99,743 | 99,743 | - |
| Min | 219 | 219 | - |
| 50 percentyl | 655 | 655 | - |
| 75 percentyl | 779 | 779 | - |
| 95 percentyl | 1002 | 1001 | - |
| 99 percentyl | 1102 | 1102 | - |
| Max | 1161 | 1161 | - |
| Średnia | 644 | 644 | - |
| Odchylenie standardowe | 203 | 203 | - |

**4.3.3 Elixir**

1 **d e f m o d u l e** T e s t . C a l c u l a t i o n . M a t r i x **d o**

2 **d e f** t r a n s p o s e ( m a t r i x ) **d o**

3 m a t r i x | >

4 L i s t . z i p | >

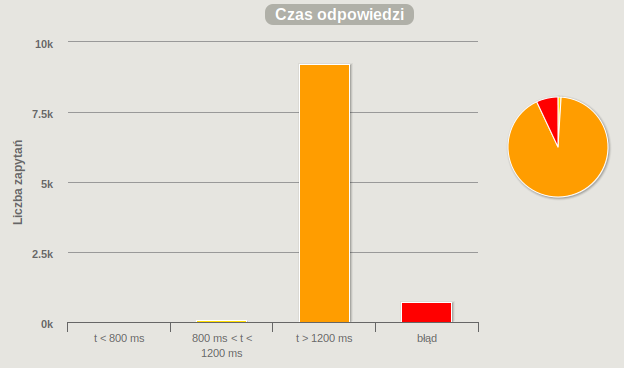
5 E n u m . m a p ( & T u p l e . t o \_ l i s t ( & 1 ) )

6 **e n d**

7 **e n d**

Listing 7: Elixir - transpozycja macierzy

Transpozycja macierzy w języku Elixir korzysta z jego funkcyjnych możliwości. Dzięki użyciu funkcji dostępnych w standardowej bibliotece kod jest bardziej zwięzły od pozostałych implementacji. Funkcja *List.zip* łączy w krotki elementy z każdego wiersza macierzy o tej samej pozycji. Następnie, używając funkcji *Enum.map*, na każdej z utworzonych krotek wykonywana jest funkcja *Tuple.to\_list*, w celu prze- kształcenia ich do jednorodnej macierzy.

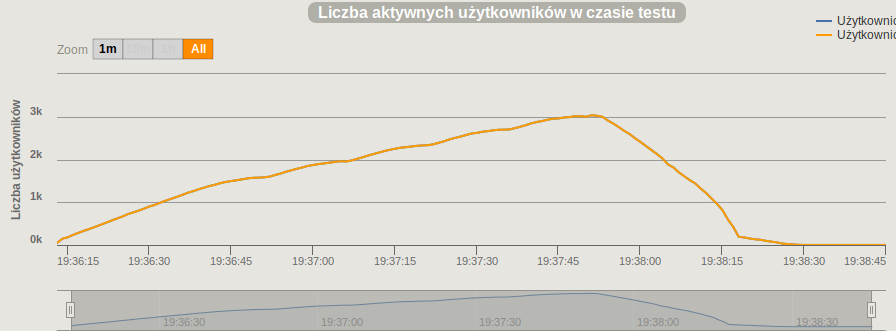


Rysunek 62: Wykres czasu odpowiedzi na zapytania (źródło: praca własna) Błędne zapytania stanowią 7% całego testu, pozostałe zaś zostały przetworzone

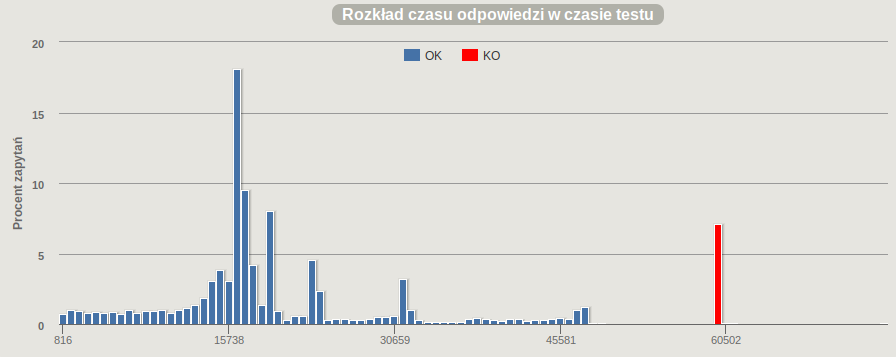
w czasie powyżej 1200 milisekund.

Liczba przyjmowanych żądań na sekundę utrzymywała się na stałym poziomie. Jednak, część zapytań, ze względu na obciążenie została odrzucona, a odpowiedzi na nie przekroczyły maksymalny czas żądania wynoszący 60 sekund.

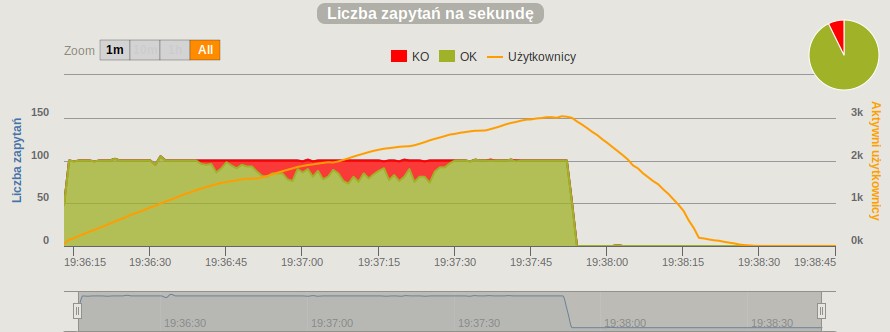
Pomimo utraty części zapytań w granicach 19:37:25 czas odpowiedzi oraz opóź- nienie uległy poprawie.



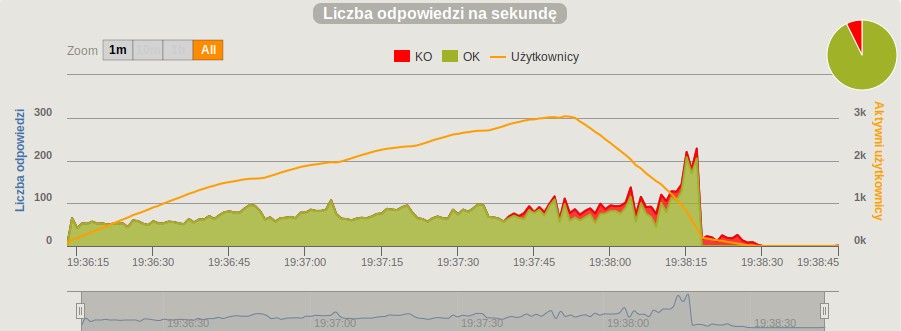
Rysunek 63: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)



Rysunek 64: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

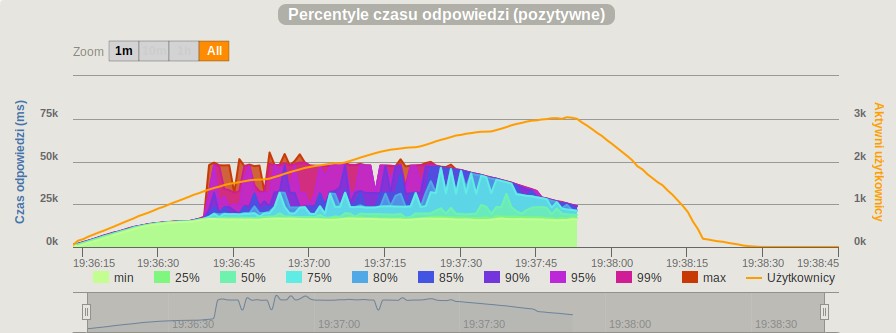


Rysunek 65: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



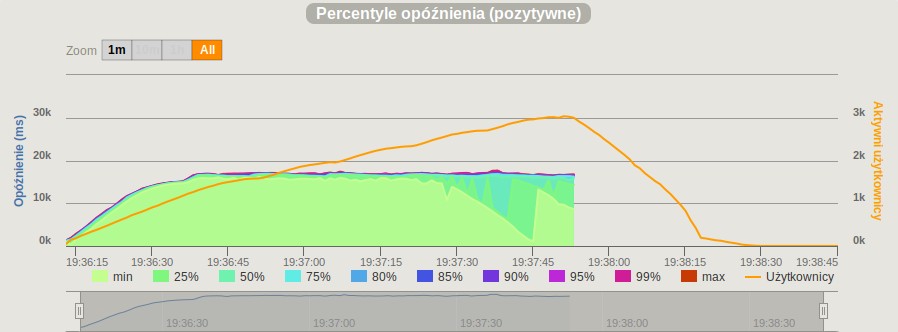
Rysunek 66: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 67: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



Rysunek 68: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 9: Statystyki Elixir w teście z wykorzystaniem zbiorów danych

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 10000 | 9274 | 726 |
| Średnia l./s | 65,691 | 60,922 | 4,769 |
| Min | 443 | 443 | 38129 |
| 50 percentyl | 17306 | 16997 | 60004 |
| 75 percentyl | 23553 | 21548 | 60005 |
| 95 percentyl | 60004 | 41425 | 60006 |
| 99 percentyl | 60005 | 47648 | 62513 |
| Max | 75051 | 55303 | 75051 |
| Średnia | 22038 | 19058 | 60103 |
| Odchylenie standardowe | 14021 | 9461 | 1518 |

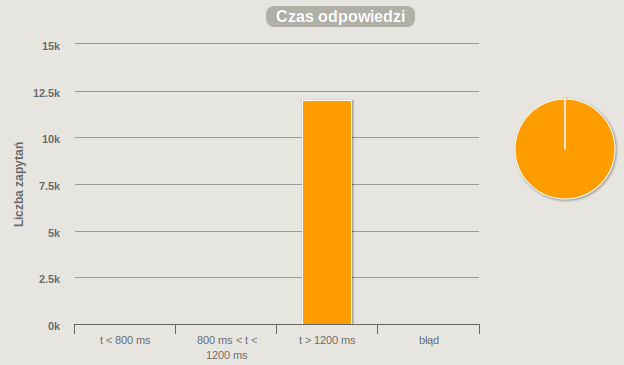
**4.4 Ograniczenia wejścia/wyjścia**

Większość systemów informatycznych korzysta z pewnego rodzaju urządzeń wej- ścia/wyjścia. Nie licząc urządzenia sieciowego, używanymi interfejsami mogą być system plików czy system zarządzania bazą danych. W tym teście wykorzystano system plików, gdyż jest obsługiwany przez standardową bibliotekę każdej **w** porów- nywanych technologii, w przeciwieństwie do komunikacji z bazą danych.

Test polega na wykonaniu metody HTTP GET na serwerze zwracającym plik tekstowy ze znakami ASCI o rozmiarze 1MB. Zasymulowano 12000 użytkowników wykonujących zapytanie niezależnie, rozłożonych na przestrzeni 100 sekund.

Implementacje we wszystkich trzech technologiach są trywialne, korzystają ze standardowej biblioteki do umieszczenia pliku w ciele odpowiedzi, więc zostały po- minięte.

**4.4.1 Java**



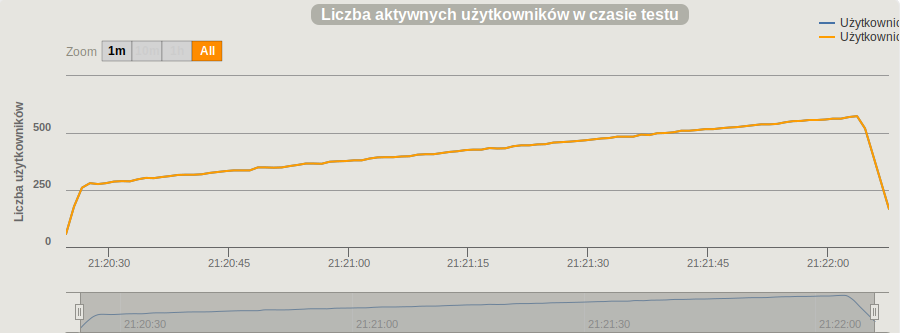
Rysunek 69: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)

Nie utracono żadnego żądania i żadna odpowiedź nie została oznaczona jako błędna, lecz 100% z nich otrzymano w czasie większym niż 1200 milisekund.

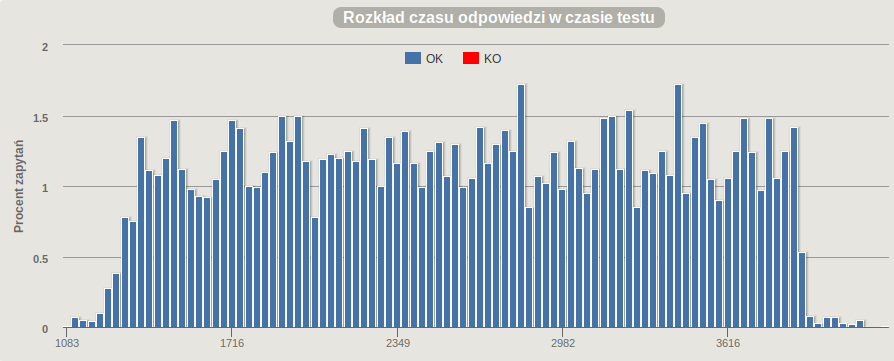
Liczba aktywnych użytkowników rośnie jednostajnie ze względu na oczekiwanie na obsługę poprzedzających zapytań.

Serwer utrzymał stały poziom obsługi przychodzących połączeń oraz odpowiedzi. Czas odpowiedzi oraz opóźnienie pokrywają się z liczbą aktywnych użytkowni- ków. Wraz ze wzrostem liczby oczekujących zapytań wzrasta czas odpowiedzi oraz opóźnienie. Wystąpiło nieznaczne rozwarstwienie w czasach odpowiedzi w danej

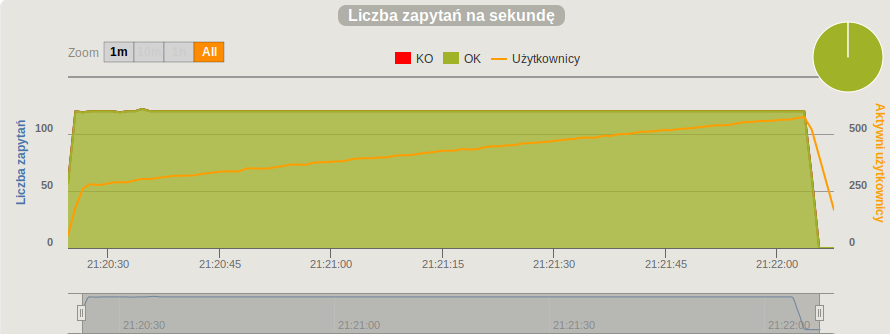
chwili czasu testu.



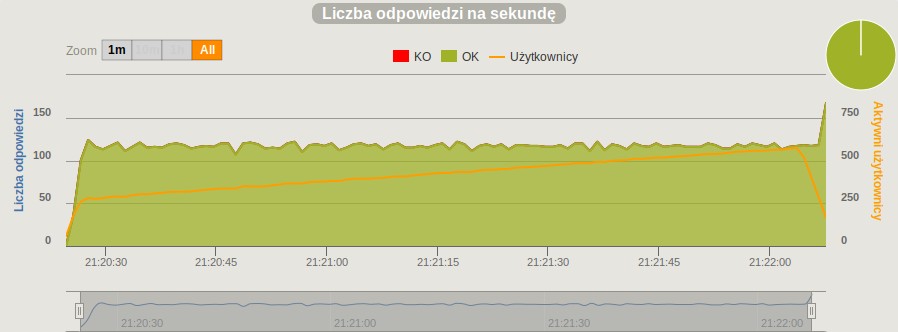
Rysunek 70: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)



Rysunek 71: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

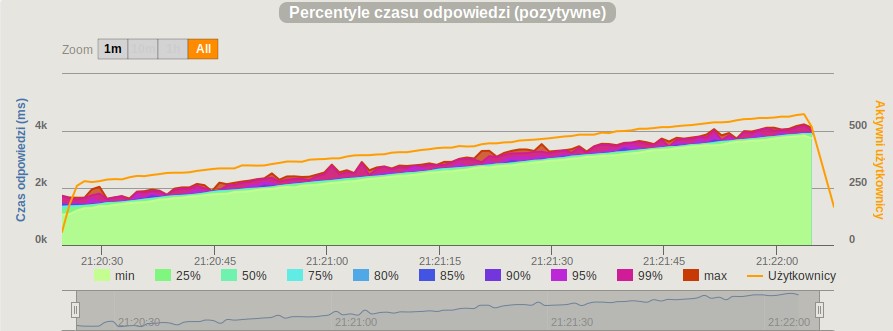


Rysunek 72: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



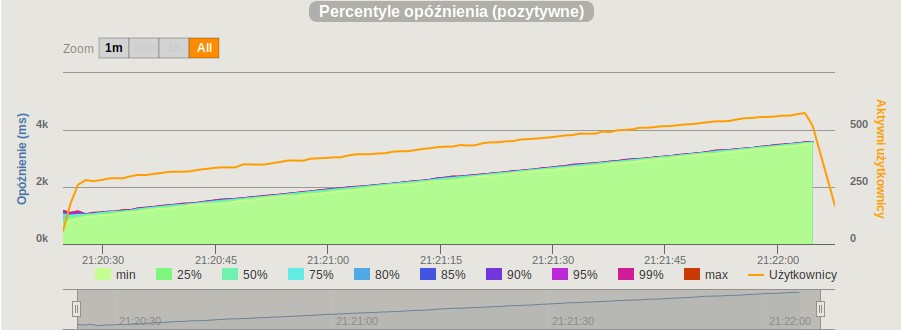
Rysunek 73: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 74: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)

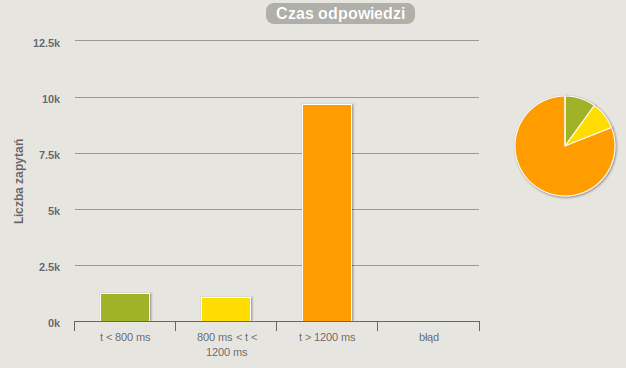


Rysunek 75: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Tablica 10: Statystyki Java w teście z ograniczeniami wejścia/wyjścia

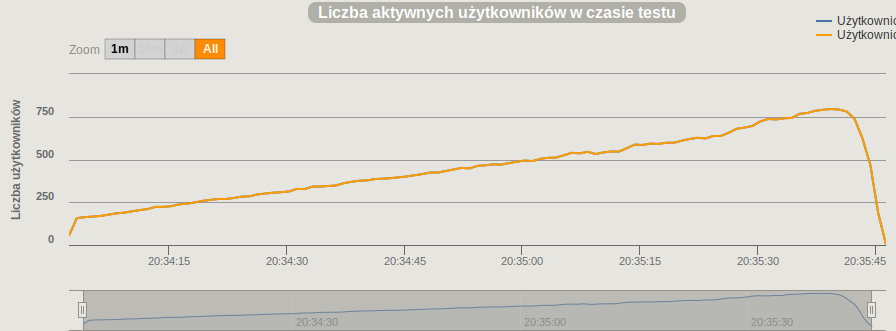
|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 12000 | 12000 | 0 |
| Średnia l./s | 115,657 | 115,657 | - |
| Min | 1067 | 1067 | - |
| 50 percentyl | 2611 | 2611 | - |
| 75 percentyl | 3252 | 3252 | - |
| 95 percentyl | 3782 | 3782 | - |
| 99 percentyl | 3882 | 3882 | - |
| Max | 4233 | 4233 | - |
| Średnia | 2606 | 2606 | - |
| Odchylenie standardowe | 759 | 759 | - |

**4.4.2 JavaScript**



Rysunek 76: Wykres czasu odpowiedzi na zapytania (źródło: praca własna) Serwer na 10% zapytań wysłał odpowiedź **( szyk** ) w czasie poniżej 800 milisekund, na

kolejnych 9% w czasie pomiędzy 800 a 1200 milisekund, zaś pozostałe powyżej 1200 milisekund.

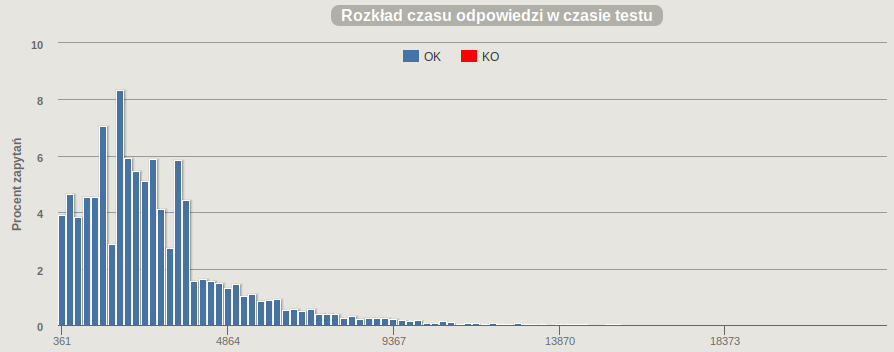


Rysunek 77: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)

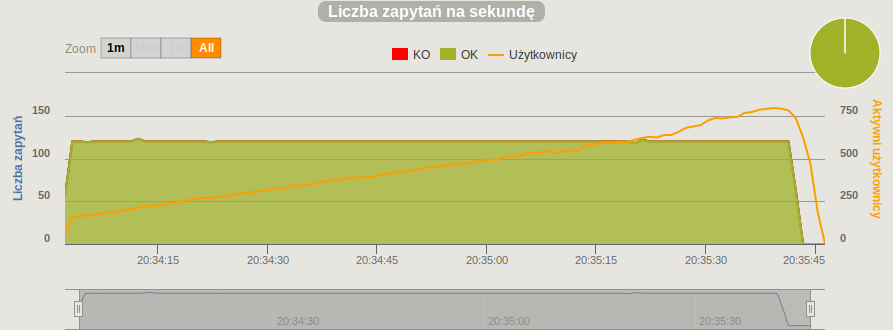
Liczba aktywnych użytkowników rośnie jednostajnie ze względu na oczekiwanie na obsługę poprzedzających zapytań.

Zarówno liczbę zapytań jak i odpowiedzi utrzymano na stałym poziomie.

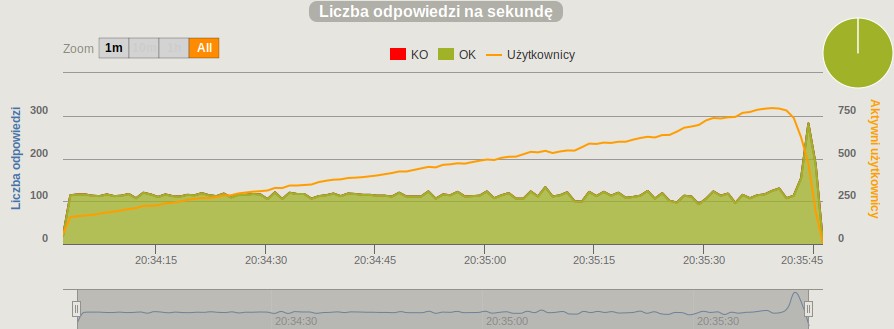
Czas odpowiedzi oraz opóźnienie rosną wraz z wzrastającą liczbą aktywnych użytkowników, oczekujących na odpowiedź. Opóźnienia oraz czasy odpowiedzi w



Rysunek 78: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

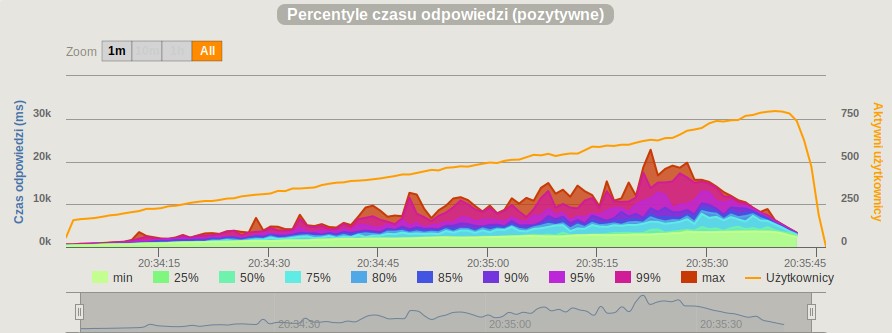


Rysunek 79: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



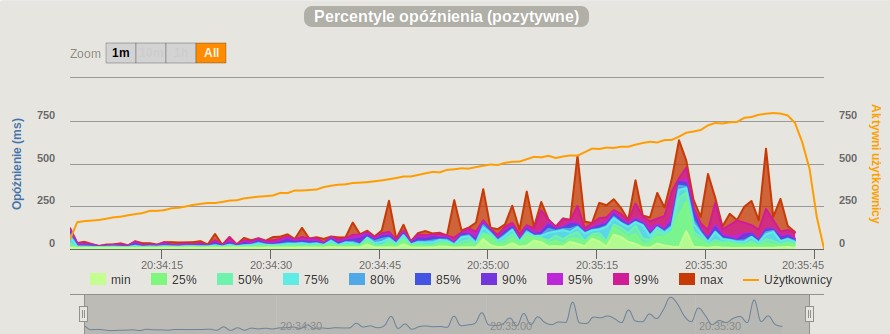
Rysunek 80: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 81: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



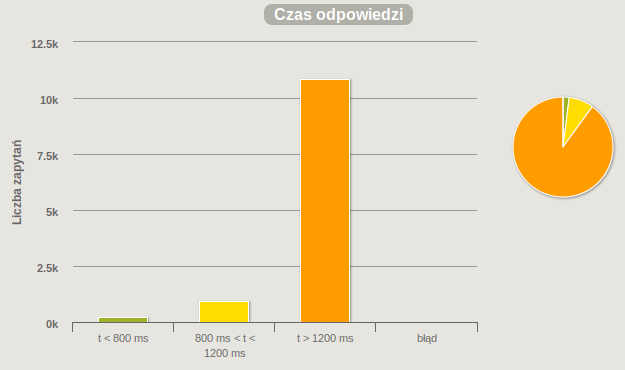
Rysunek 82: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

danej chwili czasu są niejednorodne, rozwarstwienie rośnie w miarę przybywania aktywnych użytkowników.

Tablica 11: Statystyki JavaScript w teście z ogranicze- niami wejścia/wyjścia

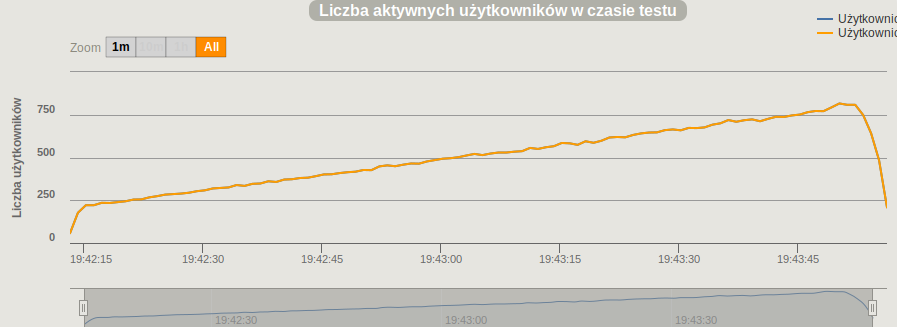
|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 12000 | 12000 | 0 |
| Średnia l./s | 114,718 | 114,718 | - |
| Min | 248 | 248 | - |
| 50 percentyl | 2461 | 2461 | - |
| 75 percentyl | 3632 | 3631 | - |
| 95 percentyl | 7101 | 7101 | - |
| 99 percentyl | 10849 | 10849 | - |
| Max | 22763 | 22763 | - |
| Średnia | 2921 | 2921 | - |
| Odchylenie standardowe | 2162 | 2162 | - |

**4.4.3 Elixir**



Rysunek 83: Wykres czasu odpowiedzi na zapytania (źródło: praca własna)

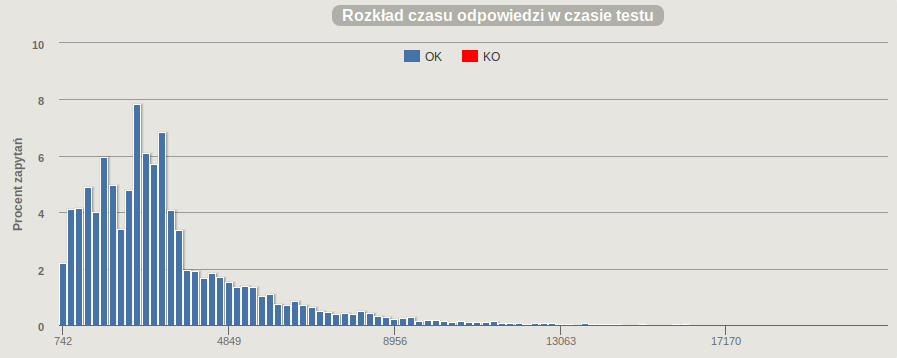
Zaledwie 2% zapytań obsłużono w czasie poniżej 800 milisekund, 90% całości powyżej 1200 milisekund, pozostałe żądania trafiły pomiędzy te dwie grupy. Żadna z odpowiedzi nie została oznaczona błędem.



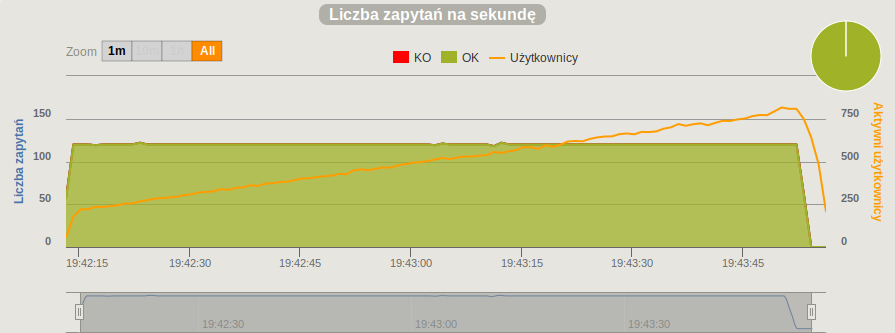
Rysunek 84: Wykres aktywnych użytkowników w czasie trwania testu (źródło: praca własna)

Liczba aktywnych użytkowników rośnie jednostajnie ze względu na oczekiwanie na obsługę poprzedzających zapytań.

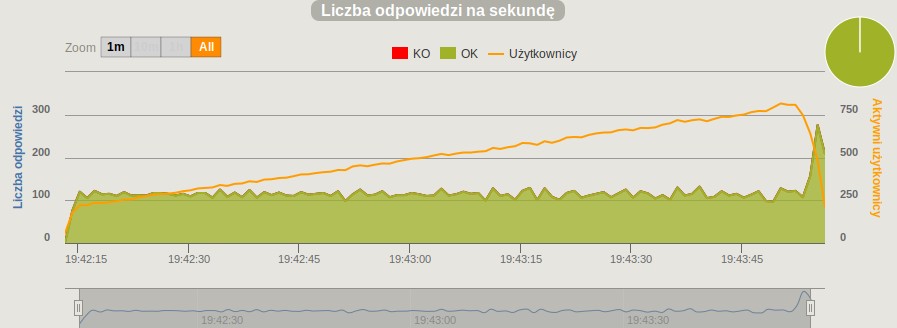
Utrzymano stały poziom obsługi połączeń przychodzących oraz przetwarzania odpowiedzi.



Rysunek 85: Wykres rozkładu czasu odpowiedzi w czasie testu (źródło: praca wła- sna)

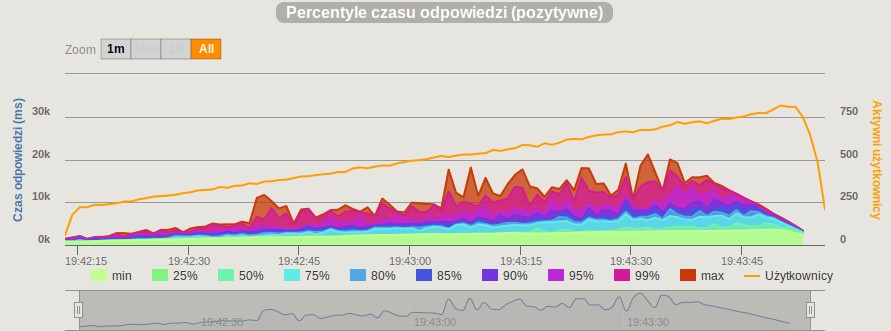


Rysunek 86: Wykres liczby zapytań na sekundę w czasie testu (źródło: praca własna)



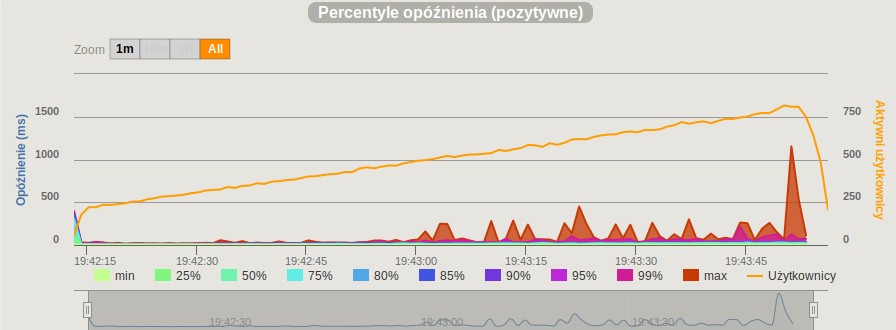
Rysunek 87: Wykres liczby odpowiedzi na sekundę w czasie trwania testu (źródło:

praca własna)



Rysunek 88: Wykres percentyli czasu odpowiedzi w czasie trwania testu (źródło:

praca własna)



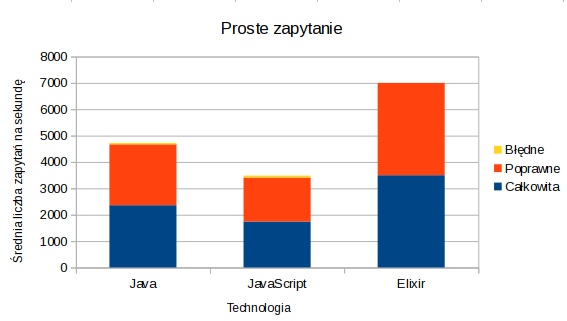
Rysunek 89: Wykres percentyli opóźnienia w czasie trwania testu (źródło: praca własna)

Czas odpowiedzi oraz opóźnienie rosną wraz z wzrastającą liczbą aktywnych użytkowników, oczekujących na odpowiedź. Opóźnienia oraz czasy odpowiedzi w danej chwili czasu są niejednorodne, rozwarstwienie rośnie w miarę przybywania aktywnych użytkowników.

Tablica 12: Statystyki Elixir w teście z ograniczeniami wejścia/wyjścia

|  |  |  |  |
| --- | --- | --- | --- |
|  | W sumie | OK | KO |
| Zapytania | 12000 | 12000 | 0 |
| Średnia l./s | 116,457 | 116,457 | - |
| Min | 639 | 639 | - |
| 50 percentyl | 2797 | 2797 | - |
| 75 percentyl | 3988 | 3988 | - |
| 95 percentyl | 7900 | 7900 | - |
| 99 percentyl | 11616 | 11616 | - |
| Max | 21174 | 21174 | - |
| Średnia | 3342 | 3342 | - |
| Odchylenie standardowe | 2239 | 2239 | - |

**4.5 Podsumowanie**

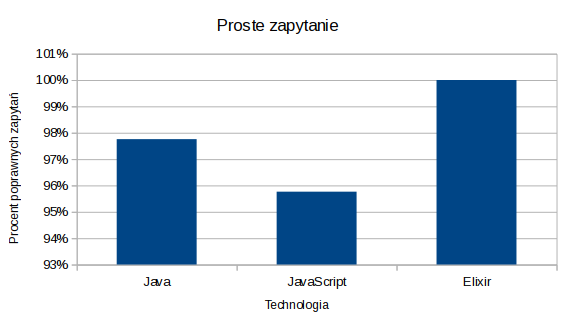


Rysunek 90: Wykres średniej liczby zapytań dla prostych zapytań

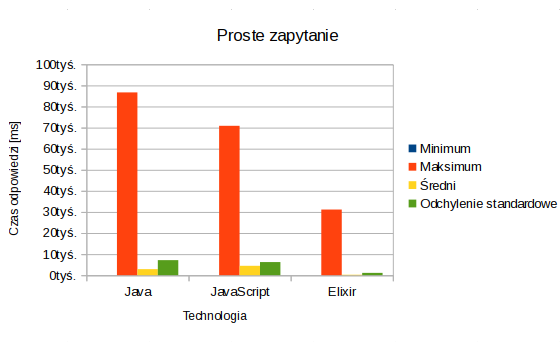
W przypadku testu dużej liczby jednoczesnych zapytań, najlepszy wynik otrzymał Elixir. Osiągnął 3498,74 żądań na sekundę, a żadne z nich nie otrzymało błędnej odpowiedzi. Zastosowanie modelu aktorowego do obsługi zapytań sprowadza się do oddelegowania każdego przychodzącego żądania do nowego aktora. Strategia ta wypadła lepiej od odpowiednika w Javie, zakładającego wykorzystywanie wątków systemu operacyjnego do przetwarzania zapytań. Najgorszy wynik otrzymano przy użyciu JavaScript. Średnia liczba otrzymanych błędnych odpowiedzi niewiele różni się pomiędzy Javą i JavaScriptem, lecz przy 74% wydajności JavaScriptu względem Javy.

W tym przypadku znaczący wpływ wywarła optymalizacja w czasie wykonywa- nia programu. Czasy odpowiedzi pierwszych 20 sekund przetwarzania w Elixirze, chociaż niższe od ekwiwalentnych w Javie i JavaScripcie, są nieporównywalnie wyż- sze od kolejnych. Wirtualna maszyna Erlanga wykryła powtarzający się wzorzec i stworzyła optymalny kod dla tego przypadku. Powtórne testy wykazały takie same wyniki w każdej z prób. Średni czas odpowiedzi dla Javy i JavaScriptu jest porów- nywalny, z nieznaczącą przewagą pierwszej technologii. Rezultat ten odbił się na odchyleniu standardowym ze względu na wyższy czas maksymalny w Javie.

W grupie czasochłonnych obliczeń, dla przypadku obliczania liczby Fibonacciego, pomimo zbliżonej średniej liczbie zapytań na sekundę pomiędzy Javą i Elixirem, Elixir osiągnął lepszy wynik pod względem średniej liczby zapytań na sekundę oraz odsetka błędnych odpowiedzi. Procent poprawnie przetworzonych zapytań Javy i JavaScriptu, odpowiednio 26% i 4%, jest nieporównywalny z 75% Elixira.



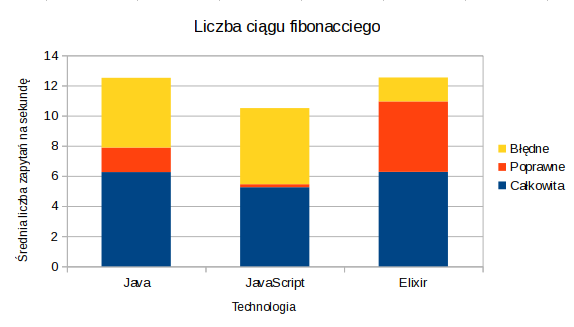
Rysunek 91: Wykres poprawnej wymiany zapytań i odpowiedzi dla prostych zapytań



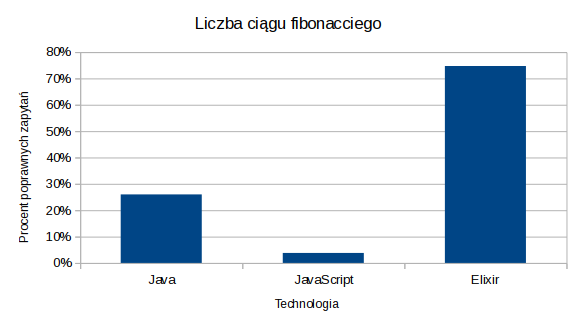
Rysunek 92: Wykres czasu odpowiedzi dla prostych zapytań

Czasy odpowiedzi dla każdej z trzech testowanych technologii plasują się na po- dobnym poziomie, z różnicami jedynie w czasach minimalnych. Należy jednak wziąć pod uwagę fakt, że rysunek 95 przedstawia wyniki jedynie dla odpowiedzi popraw- nych. W przypadku technologii Java i JavaScript wyniki mogą stanowić źródło nieprawdy, gdyż niewielka cześć całości żądań testu została przez nie przetworzona poprawnie.

W przypadku transpozycji macierzy Java oraz JavaScript uzyskały niemalże



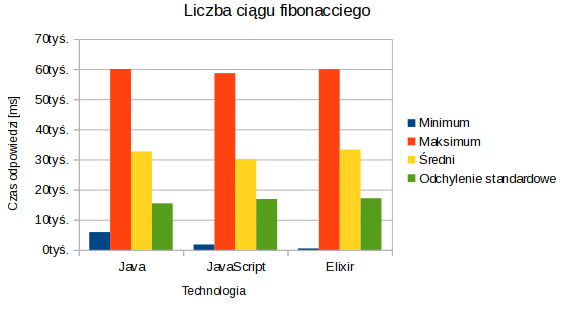
Rysunek 93: Wykres średniej liczby zapytań dla wyliczenia liczby ciągu Fibonacciego



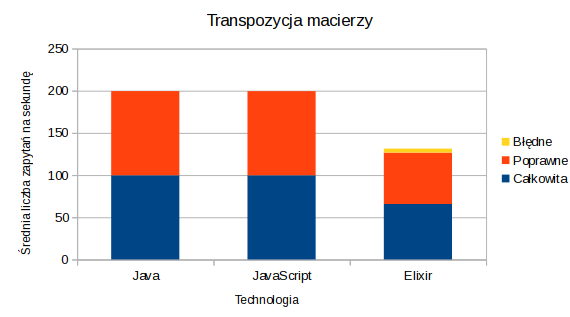
Rysunek 94: Wykres poprawnej wymiany zapytań i odpowiedzi dla wyliczenia liczby ciągu Fibonacciego

identyczne wyniki. Rozwiązania z wykorzystaniem obu technologii obsłużyły po- prawnie wszystkie zapytania z wynikiem około 100 zapytań na sekundę.

W kategorii czasu odpowiedzi Java wypadła lepiej od JavaScript ze średnim cza- sem 162 milisekund przeciwko 644 milisekundom. W tym teście wyniki Elixira są znaczenie gorsze od dwóch pozostałych technologii osiągając średni czas odpowiedzi równy 22038 milisekund oraz tracąc 7% zapytań. Tak pokaźne pogorszenie w sto-



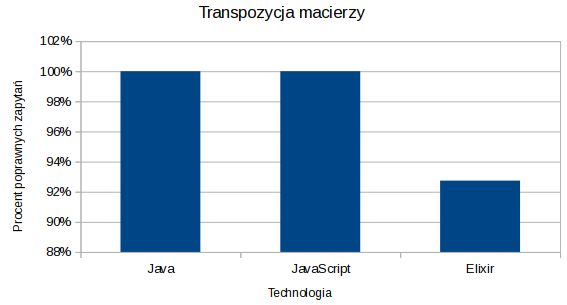
Rysunek 95: Wykres czasu odpowiedzi dla wyliczenia liczby ciągu Fibonacciego



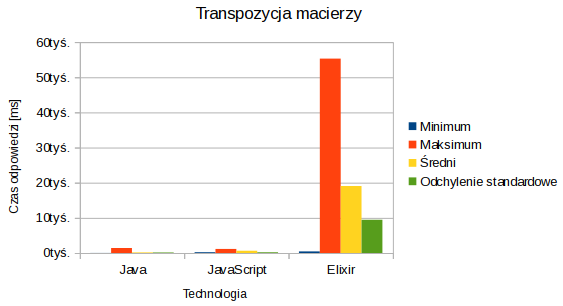
Rysunek 96: Wykres średniej liczby zapytań dla transpozycji macierzy

sunku do poprzednich testów wynika z przyjętego modelu obliczeń. Model aktorowy, który uprzednio stanowił atut, jest przyczyną zwiększonych czasów odpowiedzi. Ze względu na fakt, że lekkie procesy w wirtualnej maszynie Erlanga nie współdzielą stanu, wszystkie dane pomiędzy nimi są kopiowane powodując opóźnienia. Do- datkowo, proces ( szyk ) transpozycji macierzy, polegający na przekształceniach struktur, wymaga wykonywania kolejnych kopii danych, gdyż wszelkie struktury danych w języku Elixir są niezmienne.

W przypadku odczytu pliku wszystkie zapytania i odpowiedzi zostały obsłużone w 100%.



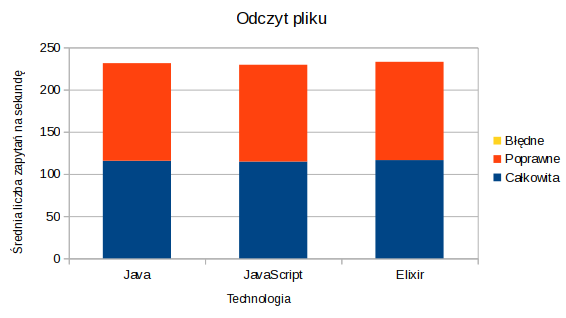
Rysunek 97: Wykres poprawnej wymiany zapytań i odpowiedzi dla transpozycji macierzy



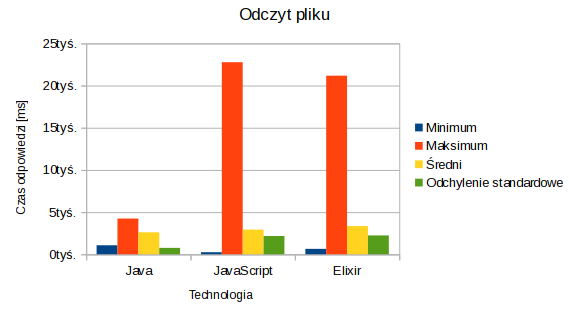
Rysunek 98: Wykres czasu odpowiedzi dla transpozycji macierzy

Java uzyskała bardzo stabilne wyniki, wydajnościowo test nie stanowił problemu, lecz ze względu na to, że dostęp do plików w Javie jest w znacznej części blokujący, obsługa wielu z zapytań została opóźniona przez oczekiwanie na urządzenie wejścia-

/wyjścia. Najniższy czas odpowiedzi, 248 milisekund, osiągnął JavaScript, a tuż za nim, z wynikiem 639 milisekund Elixir.



Rysunek 99: Wykres średniej liczby zapytań dla odczytu pliku



Rysunek 100: Wykres czasu odpowiedzi dla odczytu pliku

**5 Skalowalność**

*Skalowalność*, w kontekście systemów informatycznych, jest zdolnością systemu do efektywnego wykorzystania zwiększonej puli zasobów. Innymi słowy zwiększając możliwości obliczeniowe sprzętu, w ramach którego oprogramowanie jest urucho- mione, wydajność powinna się zwiększyć. Można wyróżnić dwa wymiary skalowal- ności[29]:

*•* pionowa

*•* pozioma

**5.1 Skalowalność pionowa**

Mówimy o *skalowalności pionowej* jeśli zwiększamy zasoby jednego węzła sys- temu komputerowego, rdzenie procesora lub pamięć do jednego komputera.[29]

Wirtualna maszyna, z której korzysta Java, jest w stanie użyć wszystkich do- stępnych rdzeni procesora. Ich wykorzystanie zależy od zastosowania mechanizmów współbieżności dostępnych w języku, np. wątków. Wielowątkowość JVM jest zaim- plementowana z użyciem natywnych wątków systemów operacyjnych. Dodatkowo Java udostępnia pewne abstrakcje, np. wykonawców (ang. *executor* ), mające za zadanie ułatwić programistom prace w wielowątkowym środowisku. Wirtualna ma- szyna Javy oferuje zbiór parametrów konfiguracyjnych pozwalających na strojenie jej pracy. Dzięki nim można dostosować, między innymi, wykorzystanie pamięci operacyjnej, aby lepiej wykorzystać dostępne zasoby.

Założeniem JavaScriptu, a raczej Node.js, jest użytkowanie jednowątkowego mo- delu Reaktor. W związku z tym każdy proces aplikacji może wykorzystać tylko jeden wątek systemu operacyjnego. Standardowa biblioteka Node’a zawiera moduł *clu- ster*. Umożliwia on rozwidlenie aplikacji przy użyciu komendy *fork* na zasadzie *ma- ster/slave*. Proces *master* jest punktem wejściowym programu i jest odpowiedzialny za uruchomienie procesów potomnych. Tworzone procesy, zwane *worker*, odpowia- dają za logikę aplikacji. Mogą one współdzielić nasłuchiwany port, wtedy zapytania zostaną automatycznie zbalansowane pomiędzy wszystkie procesy potomne. Zamy- słem modułu była proste, niskopoziomowe wsparcie dla wielu procesów, programista jest odpowiedzialny za zarządzanie nimi i obsługę błędów.

Wirtualna maszyna Erlanga, BEAM, zarządza lekkimi procesami automatycz- nie, zajmuje się rozdzielaniem dostępnych zasobów pomiędzy jednostki obliczeniowe i harmonogramuje zadania. Jedyną abstrakcją ponad wątkami są lekkie procesy wir- tualnej maszyny, modelowane na aktorów. W wypadku, gdy potrzebna jest większa kontrola nad sprzętem, niezbędne jest wykorzystanie natywnych wywołań. Podob- nie jak wirtualna maszyna Javy, BEAM udostępnia szereg parametrów konfigura- cyjnych, jednak w przeciwieństwie do JVM umożliwia kontrolę nad wykorzystaniem

procesorów.

**5.2 Skalowalność pozioma**

Mówimy o *skalowalności poziomej* jeśli dodajemy kolejne węzły do rozproszonego systemu komputerowego.[29]

Aplikacje, które nie były projektowane z myślą o przetwarzaniu rozproszonym, mogą być skalowane z użyciem uniwersalnych technik. Dzięki nim monolityczne systemy, do pewnego stopnia, mogą czerpać korzyść z używania wielu maszyn jed- nocześnie. Podstawową techniką jest uruchamianie wielu instancji aplikacji oraz zrównoważenie obciążenia (ang. *loadbalancing*) pomiędzy nimi. Nie jest to jednak technika wykorzystywana jedynie w przypadku systemów monolitycznych, można ją wykorzystać również dla architektur mikroserwisowych. Powszechną praktyką jest powielanie wybranych usług w czasie dużego ich obciążenia, często przeprowadzane automatycznie.

Standard Java EE definiuje obiekty EJB (ang. *Enterprise Java Beans*). Jedną z nich funkcjonalności jest udostępnienie metod do zdalnego wywoływania. Mogą one być wykorzystane do komunikacji pomiędzy różnymi maszynami w rozproszo- nej architekturze. Jednakże używanie ich wymaga wcześniejszej konfiguracji i od- powiedniego oprogramowania usług. Na przestrzeni lat technologia ta znacząco się rozwinęła, lecz również straciła na popularności kosztem interfejsów HTTP.

Poza modułem *cluseter*, Node.js nie oferuje narzędzi do komunikacji międzypro- cesowej. Wspomniana biblioteka nie posiada również funkcji umożliwiających ko- munikację zdalną, tworzenie systemów rozproszonych. W tym przypadku pozostaje użycie zewnętrznych narzędzi i technik ogólnego zastosowania, jak *loadbalancing*.

Model aktorowy zakłada transparentność komunikacji pomiędzy aktorami, za- równo lokalnie jak i w systemie rozproszonym. Jedynym wymogiem jest znajomość *nazwy* odbiorcy, czy przekazana w chwili utworzenia czy dostarczona przy użyciu wiadomości, Procesy w BEAM są w stanie komunikować się ze w sieci wielu ma- szyn taki sam sposób jakby to robiły w obrębie jednej wirtualnej maszyny, jednakże wciąż jest wymagana znajomość identyfikator procesu. Biblioteka Erlanga udostęp- nia funkcje umożliwiające rejestrację identyfikatora w globalnym rejestrze, który można później odzyskać posługując się wybraną nazwą. Jednakże zanim komuni- kacja pomiędzy wirtualnymi maszynami **jest** możliwa muszą one nawiązać ze sobą połączenie, co wiąże się ze znajomością adresu zdalnego hosta. Poza komunikacją poprzez przekazywanie wiadomości, standardowa biblioteka udostępnia moduł rpc (ang. *remote procedure cal l* ), który umożliwia zdalne wykonywanie procedur na określonym węźle w sieci.

**5.3 Dyskusja**

Skalowalność pionowa z perspektywy Elixira jest *wbudowana* w język. Jego wir- tualna maszyna była projektowana z myślą o systemach wieloprocesorowych i wielor- dzeniowych. Ma to ułatwić programistom prace na skomplikowanych architekturach sprzętowych, umożliwić unikanie błędów powszechnych przy użyciu tradycyjnych metod. Z drugiej strony, cechy te nie przychodzą bez kompromisów. Programowanie w Elixirze uniemożliwia nam dostęp do niskopoziomowych abstraktów, jak wątki, co z kolei odbiera twórcom systemów informatycznych kontrolę nad komputerem. Douglas C. Schmidt stworzył wzorzec Reaktor z pobudek podobnych do tych, któ- rymi kierowali się projektanci BEAM, ograniczenie liczby defektów spowodowanych błędami manualnego zarządzania przetwarzaniem w środowisku wielowątkowym. Chociaż powody są identyczne, zastosowano znacznie różniące się podejścia. Po- mimo możliwości wykorzystania wielu procesów jednocześnie, głównym założeniem Node.js jest wykorzystanie jednowątkowego demultipleksera. Problemy wielowątko- wości zostały w znacznej mierze rozwiązane, lecz skalowalność wciąż jest źródłem komplikacji. Java jest najbardziej elastyczna z tej trójki. Pojęcie wątków wciąż istnieje i może być wykorzystywane, jeśli weźmie się pod uwagę ryzyko. Udostęp- nione narzędzia usprawniające pracę z nimi, jak *executor* lub *fork-join pool*, pomimo zmniejszenia złożoności procesu, nie eliminują możliwości powstawania zakleszczeń czy problemu zagłodzenia. Dzięki uniwersalnym narzędziom istnieje możliwość, do pewnego stopnia, skalowania każdej stworzonej aplikacji sieciowej. Niestety jest to podejście bardzo ogólne, gdy potrzebna jest większa kontrola nad systemem takie zabiegi są niewystarczające. W przypadku architektur monolitycznych otrzymuje się zbędną redundancję, gdyż powielane są nie tylko elementy krytyczne z punktu wi- dzenia wydajności, a komplet funkcjonalności. Java udostępnia możliwość sieciowej komunikacji komponentów poprzez zdalne wykonywanie metod w EJB. Statyczną konfigurację, wymaganą przez tę technologię, można uzupełnić za pomocą zewnętrz- nych narzędzi. Podstawowym dodatkiem są serwery DNS, które umożliwiają posłu- giwanie się nazwami zamiast adresami usług, dzięki czemu można przekonfigurować architekturę systemu bez konieczności ingerencji w działanie aplikacji. Kod w Eli- xirze nie wymaga dodatkowych konstrukcji językowych do komunikacji międzypro- cesowej przy użyciu wiadomości, gdyż potrzebny jest jedynie identyfikator procesu. Jednakże, należy pamiętać o nawiązaniu połączenie z siecią wirtualnych maszyn. W tym przypadku, podobnie jak przy użyciu EJB w Javie, można skorzystać z ser- werów DNS. Tego rodzaju narzędzia umożliwiają również rozwiązywanie adresów węzłów aplikacji przy korzystaniu z modułu rpc. Node.js nie udostępnia abstrakcji komunikacji rozproszonej.

**6 Produktywność**

**Produktywność** - wielkość efektu produkcyjnego uzyskanego z danych nakła- dów. [30]

**6.1 Kod**

Mierzenie produktywności nie jest prostym zadaniem, gdyż trudno określić obiek- tywne kryterium. Na przestrzeni lat opracowano wiele metod wyznaczania produk- tywności pracy programistycznej. Najprostszą z nich jest liczba napisanych linii kodu. Na potrzeby tej rozprawy wybrano właśnie ją, gdyż nie jest mierzona wielkość wykonanej pracy, a konfrontacja wykonania tego samego zadania w trzech różnych technologiach.

Tablica 13: Liczba linii kodu i konfiguracji w podziale na technologie

|  |  |  |  |
| --- | --- | --- | --- |
|  | Java | JavaScript | Elixir |
| Kod Logika | 36 | 35 | 19 |
| Serwer | 55 | 31 | 98 |
| Suma | 91 | 66 | 117 |
| Konfiguracja Serwer | 421 | 88 | 22 |
| Aplikacja | 103 | 19 | 24 |
| Suma | 524 | 107 | 46 |
| ———— | —— | ———— | ——– |
| Suma | 615 | 173 | 163 |

Powyższa tabela przedstawia liczbę linii kodu i konfiguracji dla implementacji, na bazie których przeprowadzono testy w rozdziale 4. Przedstawiony stan liczbowy nie zawiera komentarzy ani pustych linijek. Najbardziej wymierną kategorią jest suma linii kodu logiki aplikacyjnej, gdyż jest on niezależny od doboru bibliotek. Nie zawiera on kodu odpowiedzialnego za obsługę żądań, jedynie przeprowadzane operacje i obliczenia, jak transpozycja macierzy czy wczytanie danych z pliku. Po- zostałe wartości nie są bez znaczenia, ponieważ charakteryzują środowisko pracy z daną technologią. Liczba linii kodu w przypadku Javy i JavaScriptu jest prawie identyczna. Wynika to z faktu, że składnia obu języków wywodzi się z języka C oraz implementacje w JavaScripcie są translacjami tych z Javy. Różnice polegają głów- nie **w** definicji klas, obsłudze modułów i eksportowaniu funkcji. Kod w języku Java może wydawać się rozwlekły w porównaniu do jego ekwiwalentu w JavaScripcie, ze względu na statyczne typowanie oraz konwencje nazewnictwa. Liczba linii kodu w

Elixirze odstaje od dwóch poprzednich. Języki funkcyjne należą do typu deklara- tywnego. W przeciwieństwie do języków imperatywnych, jak Java lub JavaScript, operują na wyższym poziome abstrakcji. Operacje są wyrażeniami na zbiorach, nie sekwencją kroków prowadzących do rozwiązania. Najlepszym przykładem jest porównanie funkcji transpozycji macierzy.

1 **p u b l i c s t a t i c i n t** [ ] [ ] t r a n s p o s e ( **i n t** [ ] [ ] m a t r i x ) {

2 **i n t** x D i m e n s i o n = m a t r i x . l e n g t h ;

3 **i n t** y D i m e n s i o n = m a t r i x [ 0 ] . l e n g t h ;

4

5 **i n t** [ ] [ ] r e s u l t = **n e w i n t** [ y D i m e n s i o n ] [ x D i m e n s i o n ] ;

6 **f o r** ( **i n t** r o w = 0 ; r o w < x D i m e n s i o n ; r o w + + ) {

7 **f o r** ( **i n t** c o l = 0 ; c o l < y D i m e n s i o n ; c o l + + ) {

8 r e s u l t [ c o l ] [ r o w ] = m a t r i x [ r o w ] [ c o l ] ;

9 }

10 }

11

12 **r e t u r n** r e s u l t ;

13 }

Listing 8: Java - funkcja transpozycji macierzy

1 **d e f** t r a n s p o s e ( m a t r i x ) **d o**

2 m a t r i x | >

3 L i s t . z i p | >

4 E n u m . m a p ( & T u p l e . t o \_ l i s t ( & 1 ) )

5 **e n d**

Listing 9: Elixir - funkcja transpozycji macierzy

Sama pętla for w Javie, przepisująca wartości pomiędzy macierzami, zajmuje 3 linijki (pomijając klamry). W przypadku Elixira, cała funkcja transpozycji macierzy została zapisana w 3 linijkach dla większej czytelności, ten sam kod można zapisać w 1. W drugim z poniższych listingów widać deklaratywność styl, posługiwanie się funkcjami bez definiowania kroków przetwarzania.

Kod serwera odpowiada za odbieranie zapytań, wywołanie funkcji obliczeń i zwró- cenie odpowiedzi. W większym stopniu jest zależny od doboru narzędzi, niż charak- terystyki języka programowania. Zapis w Javie opiera się w całości na standardach Java EE. W przypadku JavaScriptu, do obsługi kodu sieciowego, wybrano bibliotekę Express.js. Ma ona być mała i elastyczna, wprowadzając jedynie niewielkie udogod- nienia w stosunku do programowania w Node.js. Opis ten jest odzwierciedlony w otrzymanych wynikach. Jak można zauważyć JavaScript posiada najmniej kodu serwerowego. W przeciwieństwie do Express.js, Phoenix Framework jest biblioteką obszerną, lecz jedyną określaną jako stabilną i odpowiednią do zastosowań produk- cyjnych jaka istnieje dla języka Elixir na tę chwilę. Pomimo największej liczby linii kodu w tej kategorii, znaczna jego część jest generowana przy tworzeniu projektu.

Jednak mimo braku konieczności pisania, programista powinien go znać, aby do- stosować do potrzeb systemu, dlatego też został on wliczony do sumy z powyższej tabeli. Konfiguracja serwera aplikacyjnego w Javie to obszerny plik XML. Znajdują się w nim ustawienia wszystkich komponentów dostępnych na serwerze Wildfly, de- finiowanych przez standardy Java EE. Serwer w JavaScripcie jest konfigurowany z użyciem samego języka programowania. Jak wspomniano, Express.js jest biblioteką minimalistyczną, opcje są zdefiniowane jedynie dla ograniczonej liczby używanych komponentów. Podobnie jak w przypadku JavaScriptu, w Elixirze ustawienia są zapisane przy użyciu języka programowania. Twórcy Phoenixa przyjęli zasadę *kon- wencja ponad konfigurację*, w związku z czym opcje ograniczono do minimum.

Pliki konfiguracyjne aplikacji wszystkich trzech technologii zawierają informacje o projekcie oraz zależnościach. Podobnie jak inne ustawienia, Java używa formatu XML, a Elixir języka programowania. W przypadku JavaScriptu używany jest JSON (JavaScript Object Notation).

**6.2 Biblioteki**

Istotnym elementem każdej technologii deweloperskiej jest dostępność bibliotek. Znaczną część pracy można wykonać bez napisania ani jednej linijki kodu, z wykorzystaniem dostępnych publicznie bibliotek. Dzięki temu zyskujemy przetestowaną funkcjonalność, gotową do użycia. Największym repozytorium bibliotek bazujących na wirtualnej maszynie Javy jest Central Repository[31]. Istniejąca od 2002 roku baza skupia około 130 tysięcy projektów, które w sumie udostępniły prawie 1,2 miliona artefaktów. Niestety nie są dostępne aktualne statystyki pobrań.[32] Na dorobek Javy składa się praca wielu organizacji patronujących otwartym projektom. Wśród nich znajdują się fundacje jak Apache Software Foundation[33] skupiająca ponad 350 projektów Open Source oraz inicjatyw obejmujących szerokie spektrum technologii, czy Eclipse Foundation[34], będąca członkiem komitetu Java Commu- nity Process. Na rozwój Javy mają również wpływ przedsiębiorstwa w różnych branżach. [zob. 3.2]

npm, repozytorium modułów Node.js, gromadzi ich prawie 230 tysięcy, licząc ponad

140 milionów pobrań dziennie.[4] Powszechność JavaScriptu sprzyja powstawaniu nowych projektów, lecz bez wsparcia zorganizowanych podmiotów, tak jak w przy- padku Javy, znaczna ich część pozostaje we wczesnej fazie rozwoju lub jest porzu- cana.

Ze względu na fakt, że Elixir jest młodym językiem, zbiór dostępnych bibliotek nie jest tak imponujący jak w przypadku dwóch pozostałych ekosystemów. Projekt hex, tworzący manager pakietów w języku Elixir, gromadzi 1384 modułów w 6097 wersjach, które pobrano ponad 100 tysięcy razy dziennie.[28]

**7 Uwagi końcowe**

Trudno zaprzeczyć, że Java jest obecnie dominującą technologią w branży. Pozy- cja ta może utrzymać się jeszcze przez wiele lat, a to ze względu na ogromną liczbę systemów informatycznych stworzonych z jej wykorzystaniem. Wszystkie te projekty wymagają utrzymania póki nie zostaną zastąpione nowymi. Jednakże Java ciągle się rozwija i ewoluuje. Powstają nowe specyfikacje i standardy. Nie tylko starsze systemy wykorzystują tę technologię, jest atrakcyjna również dla nowo powstają- cych projektów. Zarówno język jak i wirtualna maszyna jest dojrzała i stabilna. Cechuje się wysoką wydajnością w ogólnym przypadku oraz sprawdza się w wielu różnorodnych wyspecjalizowanych scenariuszach. Trudności w konfiguracji i obsłu- dze wielowątkowości rekompensuje bogatym zapleczem bibliotek.

JavaScript i Node.js osiągnął zaskakująco dobre wyniki w przeprowadzonych testach. Wyróżnia się w przypadkach, gdzie wąskim gardłem są operacje wejścia/wyjścia, jak strumieniowanie danych. Chociaż wzorzec Reaktor sprawdza się doskonale w takich przypadkach, to w przypadku gdy wystąpi potrzeba przeprowadzenia intensywnych obliczeń czy przetworzenia złożonej logiki biznesowej jednowątkowy model może nie być wystarczający. Pomimo skupienia wokół siebie ogromnej społeczności, brakuje w niej wkładu przedsiębiorstw, tak jak to ma się w przypadku Javy.

Cechy Elixira czynią go dobry kandydatem do tworzenia nowych projektów infor- matycznych, jednak wciąż jest bardzo wczesny etap w rozwoju tego języka. Wybór wirtualnej maszyny Erlanga jako jego podstawy z pewnością działa na jego korzyść. Jednak sam Erlang nigdy nie zdobył popularności, czego przyczyną może być długi okres, w którym był zamkniętym językiem używanym przez firmę Ericsson. Być może, na fali popularności funkcyjnego paradygmatu programowania, języki te po- większą grono użytkowników. Jednakże, podobnie jak w przypadku JavaScriptu, potrzebują one wsparcia biznesu. Niewątpliwie Elixir wykazuje cechy niezbędne do tworzenia systemów o wysokiej odporności na błędy. Model aktorowy dobrze spełnia swoją rolę w zastosowaniach architektur wieloprocesorowych. Chociaż odmienne od klasycznych modele współbieżności mogą pomóc w redukcji ilości defektów związa- nych ze współbieżnym kodem, to wciąż tworzenie oprogramowania wykorzystującego przetwarzanie wieloprocesorowe jest niełatwym przedsięwzięciem. Asynchroniczny charakter modelu aktorowego lub wzorca Reaktor mogą być trudniejsze do ana- lizy i rozwiązywania błędów, szczególnie dla programistów zaznajomionych jedynie z klasycznymi metodami. Elixir nie jest również językiem uniwersalnym. Bardzo trudnym zadaniem może być zamodelowanie procesów numerycznych w sposób wy- dajny z użyciem Elixira.

**8 Możliwości rozwoju**

Niewątpliwie temat tej pracy jest bardzo obszerny i pozostawia szerokie spektrum rozwoju. Chociaż w testach wydajnościowych wzięto pod uwagę różnorodne scena- riusze, nie skorzystano ze zdolności testowanych technologii do tworzenia systemów rozproszonych. Kolejnym elementem wartym włączenia do testów jest wykorzystanie sprzętu serwerowego zamiast komputerów osobistych. Ponadto zwiększenie czasu symulacji może przynieść interesujące wyniki. Już przy zaledwie 100 sekundach zauważono zmiany w dynamice stworzonych systemów. Zaobserwowanie możliwo- ści optymalizacji przez środowiska jest także interesującym zjawiskiem. Z drugiej strony, w tego typu aplikacjach problemy mogą uwidocznić się po godzinach lub nawet dniach ciągłej pracy pod obciążeniem. W pracy przedstawiono, w znaczny stopniu, rozwiązania standardowe, dostępne dla danych technologii. Jednakże przy tak rozwiniętej społeczności jak w przypadku Javy, stworzono wiele bibliotek odblo- kowujących nowe możliwości. Istnieją projekty implementujące model aktorowy lub wzorzec Reaktor dla wirtualnej maszyny Javy. W kontekście współbieżności można wziąć pod uwagę inne opracowane modele, jak pamięć transakcyjna. Tematyka architektury systemów informatycznych również jest rozległa, a koncepcje architek- tury mikroserwisowej ewoluują z dnia na dzień. Innowacje technologiczne dotyczą nie tylko języków programowania i ich ekosystemów. Nieodłączny element każdego systemu, bazy danych, także ewoluują i wyłaniają się nowe nisze, jak bazy NoSQL czy grafowe. Połączenie innowacyjnego języka z odpowiednimi systemami zarzą- dzania bazą danych jest interesującym kierunkiem badań. Technologie pojawiają się i znikają, ale podstawy teoretyczne, które za nimi stoją wciąż mogą inspirować kolejne, nawet jak kiedy aktualne zostaną zapomniane.

**Bibliografia**

[1] *TIOBE Index for December 2015*. Grud. 2015. url: [http://www.tiobe.com/](http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html)

[index.php/content/paperinfo/tpci/index.html](http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html) (term. wiz. 13. 12. 2015).

[2] E. Jendrock i in. *The Java EE 7 Tutorial: Volume 2*. 5th. Addison-Wesley

Professional, 2014. isbn: 0321980085, 9780321980083.

[3] D. Flanagan. *JavaScript: The Definitive Guide*. Ó’Reilly Media, Inc.", 2006. [4] *npm*. url: <https://www.npmjs.com/>(term. wiz. 16. 12. 2015).

[5] S. Tilkov i S. Vinoski. „Node.Js: Using JavaScript to Build High-Performance Network Programs”. W: *IEEE Internet Computing* 14.6 (list. 2010), s. 80–83. issn: 1089-7801. doi: [10.1109/MIC.2010.145](http://dx.doi.org/10.1109/MIC.2010.145). url: [http://dx.doi.org/](http://dx.doi.org/10.1109/MIC.2010.145)

[10.1109/MIC.2010.145](http://dx.doi.org/10.1109/MIC.2010.145).

[6] R. M. Lerner. „At the Forge: Node.JS”. W: *Linux Journal* 2011.205 (maj

2011). issn: 1075-3583.

[7] J. Armstrong. „Erlang - A survey of the language and its industrial applica- tions”. W: *ICFP ’97 Proceedings of the second ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 1996. isbn: 0897919181. doi: [10.1145/258948.258967](http://dx.doi.org/10.1145/258948.258967).

[8] F. Hebert. *Learn You Some Erlang for Great Good!: A Beginner’s Guide*. San

Francisco, CA, USA: No Starch Press, 2013. isbn: 1593274351, 9781593274351.

[9] M. Logan, E. Merritt i R. Carlsson. *Erlang and OTP in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010. isbn: 1933988789, 9781933988788.

[10] J. Valim. *Elixir Design Goals*. Sierp. 2013. url: [http://elixir-lang.org/](http://elixir-lang.org/blog/2013/08/08/elixir-design-goals/)

[blog/2013/08/08/elixir-design-goals/](http://elixir-lang.org/blog/2013/08/08/elixir-design-goals/) (term. wiz. 03. 11. 2015).

[11] D. Thomas. *Programming Elixir: Functional , Concurrent , Pragmatic , Fun*.

1st. Pragmatic Bookshelf, 2014. isbn: 1937785580, 9781937785581.

[12] S. St. Laurent i J. D. Eisenberg. *Introducing Elixir: Getting Started in Functio-*

*nal Programming*. 1st. O’Reilly Media, Inc., 2014. isbn: 1449369995, 9781449369996.

[13] H. Sutter i J. Larus. „Software and the Concurrency Revolution”. W: *Queue*

3.7 (2005), s. 54–62. issn: 1542-7730. doi: [10.1145/1095408.1095421](http://dx.doi.org/10.1145/1095408.1095421).

[14] E. A. Lee. „The Problem with Threads”. W: *Computer* 39.5 (2006), s. 33–42.

issn: 0018-9162. doi: [10.1109/MC.2006.180](http://dx.doi.org/10.1109/MC.2006.180).

[15] L. Lamport. „Time, clocks, and the ordering of events in a distributed system”.

W: *Communications of the ACM* 21.7 (1978), s. 558–565.

[16] K. Bastani. *Spring Cloud Example Project*. url: [https : / / github . com /](https://github.com/kbastani/spring-cloud-microservice-example)

[kbastani/spring-cloud-microservice-example](https://github.com/kbastani/spring-cloud-microservice-example) (term. wiz. 20. 12. 2015).

[17] M. Fowler i J. Lewis. *Microservices*. Mar. 2014. url: [http://martinfowler. com/articles/microservices.html](http://martinfowler.com/articles/microservices.html) (term. wiz. 03. 11. 2015).

[18] M. Fowler. *MicroservicePremium*. Maj 2015. url: [http : / / martinfowler . com/bliki/MicroservicePremium.html](http://martinfowler.com/bliki/MicroservicePremium.html) (term. wiz. 20. 12. 2015).

[19] S. Newman. *Building Microservices*. O’Reilly Media, Incorporated, 2015. isbn:

9781491950357.

[20] *Java Community Process EC Elections*. List. 2015. url: [https://www.jcp. org/en/whatsnew/elections](https://www.jcp.org/en/whatsnew/elections) (term. wiz. 13. 12. 2015).

[21] M. Eisele. *Modern Java EE Design Patterns*. 1st. O’Reilly Media, Inc., 2015. [22] D. C. Schmidt. „Pattern Languages of Program Design”. W: wyed. J. O. Co-

plien i D. C. Schmidt. New York, NY, USA: ACM Press/Addison-Wesley Pu-

blishing Co., 1995. Rozd. Reactor: An Object Behavioral Pattern for Concur- rent Event Demultiplexing and Event Handler Dispatching, s. 529–545. isbn:

0201607344. url: <http://dl.acm.org/citation.cfm?id=218662.218705>.

[23] R. K. Karmani, A. Shali i G. Agha. „Actor Frameworks for the JVM Platform: A Comparative Analysis”. W: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. PPPJ ’09. Calgary, Al- berta, Canada: ACM, 2009. isbn: 9781605585987. doi: [10 . 1145 / 1596655 .](http://dx.doi.org/10.1145/1596655.1596658)

[1596658](http://dx.doi.org/10.1145/1596655.1596658).

[24] C. Hewitt i H. G. Baker. „Laws for Communicating Parallel Processes”. W: *Information Processing 77, Proceedings of IFIP Congress 77*. North Holland, Amsterdam, sierp. 1977, s. 987–992.

[25] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*.

Cambridge, MA, USA: MIT Press, 1986. isbn: 0262010925.

[26] J. Valim. *Elixir in times of microservices*. Czer. 2015. url: [http : / / blog . plataformatec.com.br/2015/06/elixir- in- times- of- microservices/](http://blog.plataformatec.com.br/2015/06/elixir-in-times-of-microservices/) (term. wiz. 20. 12. 2015).

[27] *MVNRepository*. url: <http://mvnrepository.com/>(term. wiz. 16. 12. 2015). [28] *hex*. url: <https://hex.pm/>(term. wiz. 16. 12. 2015).

[29] H. El-Rewini i M. Abd-El-Barr. *Advanced Computer Architecture and Paral lel*

*Processing*. Wiley-Interscience, 2005. isbn: 0471467405.

[30] E. Sobol, red. *Słownik języka polskiego PWN*. Warszawa: Wydawnictwo Na- ukowe PWN, 2015.

[31] S. Inc. *The Central Repository*. url: <http://central.sonatype.org/>(term. wiz. 30. 12. 2015).

[32] S. Inc. *Quick Statistics for The Central Repository*. url: [http : / / search . maven.org/#stats](http://search.maven.org/#stats) (term. wiz. 30. 12. 2015).

[33] *The Apache Software Foundation*. url: <http://www.apache.org/>(term. wiz.

30. 12. 2015).

[34] *Eclipse Foundation*. url: [https :/ /eclipse .org / org /foundation/](https://eclipse.org/org/foundation/) (term. wiz. 30. 12. 2015).