JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 9 (2018) 350-358
Wysłane: 2018-09-25
Przyjęte: 2018-10-09

Badanie funkcjonalności aplikacji wykonanej w technologii .NET Core na platformie Raspberry Pi II

Tomasz Piotr Sajnóg*, Dariusz Czerwiński
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule przedstawiono wyniki analizy funkcjonalności aplikacji wykonanej w technologii .NET Core firmy Microsoft na platformie Raspberry Pi 2. Badania zostały zrealizowane poprzez implementację testowych aplikacji wykorzystujących biblioteki firmy Microsoft oraz opracowane przez społeczność zgromadzoną wokół technologii .NET Core. Skupiono się na zastosowaniu niniejszej technologii w zastosowaniach Internetu Rzeczy. Przy porównaniu brano pod uwagę możliwość wykorzystania interfejsów Raspberry Pi w technologii .NET Core oraz oficjalnie wspieranego języka Python i popularnej technologii Node.js. Porównano również wydajność tych technologii w badanym środowisku sprzętowym.

Słowa kluczowe: .NET Core; Raspberry Pi; Internet Rzeczy

*Autor do korespondencji.

Adres e-mail: tomeksajnog@wp.pl

Testing the functionality of the application made in .NET Core technology on the Raspberry Pi II platform

Tomasz Piotr Sajnóg*, Dariusz Czerwiński
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The paper presents the results of functionality analysis of application made in Microsoft .NET Core technology on Raspberry Pi 2 platform. The research was done by implementing applications using libraries provided by Microsoft and developed by .NET Core community. The focus was placed on usage of this technology in the Internet of Things applications. The comparison of the possibility of using Raspberry Pi interfaces in .NET Core, officially supported Python language and popular Node.js technology was made. The performance of these technologies was also compared.

Keywords: .NET Core; Raspberry Pi; Internet of Things

Corresponding author.

E-mail address: tomeksajnog@wp.pl

1. Wstęp

W dzisiejszych czasach coraz częściej jest mowa o tzw. Internecie Rzeczy. Jest to koncepcja gromadzenia, przesyłania oraz przetwarzania informacji przez różnego typu urządzenia. Najczęściej są to rozmaite czujniki zbierające informacje o środowisku, w którym się znajdują. Obecnie najczęściej używa się technologii Internetu Rzeczy do monitoringu środowiska, zwłaszcza jakości powietrza, a także w automatyce domowej [1].

Jedną z najpopularniejszych platform IoT jest komputer jednopłytkowy Raspberry Pi. Początkowo Raspberry Pi opracowano do nauki informatyki w szkołach, lecz znalazł wiele innych zastosowań. Najczęściej stosowany jest jako prosty serwer, który zużywa niewiele energii. Platforma ta bazuje na układach SoC firmy Broadcom i używa architektury ARM [2].

Jako systemy operacyjne instalowane na platformach typu Raspberry wykorzystuje się systemy oparte na jądrze Linux, m.in. Raspbian, ale istnieje specjalna edycja Windows 10 IoT Core, która jest darmowa [2]. Niestety różni się znacząco od systemu Windows znanego z komputerów i telefonów i jest jedynie platformą do uruchamiania aplikacji. Posiada wiele ograniczeń związanych z dostępnością sterowników do urządzeń peryferyjnych podłączanych do Raspberry Pi [3].

Popularność Raspberry Pi została osiągnięta dzięki obecności złącza GPIO, które udostępnia wiele interfejsów komunikacyjnych. Pozwala to na bezpośrednie podłączenie do układu SoC różnych czujników i elementów wykonawczych bez potrzeby stosowania konwerterów. Zwykły komputer zazwyczaj nie pozwala na to bez zastosowania konwerterów podłączanych najczęściej przez złącze USB. Złącze zapewnia także zasilanie peryferiów, choć nie powinny być to urządzenia energochłonne.

Oficjalnym językiem programowania na platformę Raspberry Pi jest Python i najwięcej przykładowych programów jest właśnie w tym języku. Również człon Pi w nazwie pochodzi od zmodyfikowanej nazwy tego języka i początkowo to Python miał być jedynym wspieranym językiem [4]. Jest on uznawany za prosty do nauki i polecany początkującym programistom. Jednak jego składnia znacząco

różni się od składni języka C i pochodnych, tj. C++, C#, Java i innych popularnych języków [5].

Kolejną popularną technologią wykorzystywaną do tworzenia aplikacji na platformie Raspberry Pi jest Node.js. Jest to środowisko do uruchamiania aplikacji napisanych w języku JavaScript. Najpopularniejszym zastosowaniem są serwery WWW i aplikacje internetowe. Podstawą jego działania jest silnik V8 opracowany przez firmę Google dla przeglądarki Chrome oraz biblioteka libUV. Technologia została stworzona w celu ułatwienia wprowadzania powiadomień push na stronach internetowych [6].

Od niedawna dla Raspberry Pi jest dostępna technologia .NET Core, która jest oficjalną implementacją .NET na otwartej licencji. Technologia .NET jest wieloplatformowa, więc może działać na innych systemach niż Windows, z którym od zawsze kojarzono .NET. Inną implementacją jest znacznie starsze Mono, lecz firma Xamarin została przejęta przez Microsoft i rozwój Mono nie jest tak dynamiczny jak .NET Core. Opracowanie Mono było możliwe dzięki standaryzacji specyfikacji .NET oraz obietnicy braku roszczeń patentowych ze strony firmy Microsoft. Jest to odmienna sytuacja niż w przypadku Javy, co pokazuje przykład procesów sądowych wytaczanych firmie Google przez Oracle za użycie patentów związanych z Javą w systemie Android [7,8].

Dla celów badawczych opracowano aplikacje w następujących językach: Python, JavaScript i C#. Ostatni z języków jest w pełni obiektowym językiem stworzonym na potrzeby .NET przez zespół Andersa Hejlsberga w latach 1998-2001 [9,10]. Jako platformę do analizy wybrano Raspberry Pi 2, ponieważ technologia .NET Core wymaga nowszego procesora, który jest obecny w Raspberry Pi 2 oraz 3. W drugim przypadku można wykorzystać 64-bitowe środowisko uruchomieniowe pod warunkiem uruchomienia 64-bitowego systemu operacyjnego, ale standardowy system Raspbian jest jedynie 32-bitowy.

Celem badawczym, który został w artykule przedstawiony jest analiza funkcjonalności oraz wydajności aplikacji .NET Core na platformie Raspberry Pi 2 w porównaniu do aplikacji powstałych w środowiskach Python i Node.js.

Na początku badań założono, że technologia .NET Core jest w stanie obsługiwać interfejsy oraz komponenty podłączone do platformy Raspberry Pi, a wykonanie tych procedur jest wydajniejsze niż w konkurencyjnych technologiach.

2. Dostępne biblioteki .NET dla Raspberry Pi

W trakcie badań literaturowych znaleziono jedynie dwie biblioteki, które były dedykowane Raspberry Pi: Unosquare.RaspberryIO oraz Raspberry# IO.

Biblioteka Unosquare.RaspberryIO jest oparta na natywnej bibliotece WiringPi i może być uruchomiona w środowiskach Mono oraz .NET Core. W aplikacjach zainstalowano ją przy pomocy narzędzia NuGet [11].

Drugą biblioteką jest Raspberry# IO, która wymagała pobrania kodu źródłowego z repozytorium i niezbędnych modyfikacji w postaci wygenerowania nowych plików .csproj, ponieważ biblioteka nie była przystosowana do .NET Core. Zmodyfikowano również kod samej biblioteki, ponieważ z powodu błędu w systemie Raspbian, nie można było uruchomić aplikacji wymagających rozpoznania typu procesora, który był identyfikowany jako BCM2835. Jest to układ z pierwszych edycji Raspberry Pi, które nie są wspierane przez .NET Core [12].

3. Analiza funkcjonalności i wydajności

W celu przeprowadzenia badań porównawczych opracowano aplikacje dedykowane dla platformy Raspberry Pi 2. Aplikacje wykonano w językach Python, JavaScript i C#.

3.1. Metoda badawcza

Wykonane aplikacje sprawdzały działanie podstawowych operacji, czyli wyświetlania tekstu i operacji matematycznych, a także działania komponentów sprzętowych, które zostały podłączone do Raspberry Pi przez interfejsy: SPI, I2C, GPIO i UART.

Za kryterium oceny funkcjonalności przyjęto poprawne działanie aplikacji w technologii .NET Core w stosunku do innych technologii. Wyniki, które uzyskiwane były przez aplikacje .NET powinny być zbliżone do tych, które były uzyskiwane przez aplikacje napisane w językach Python i Node.js.

Za kryterium oceny wydajności przyjęto czas wykonania danej operacji, który zmierzono funkcjami wbudowanymi w daną technologię. Każda operacja została wykonana 10 razy, ale z wyjątkiem inicjalizacji, która została zmierzona tylko raz w niektórych przypadkach. Wyznaczono czas średni, maksymalny oraz minimalny. Wartości czasu podano w milisekundach i zaokrąglono do 5 miejsc po przecinku. W przypadku analizy wydajności aplikacji internetowej wykorzystano program ApacheBench, który wchodzi w skład serwera Apache.

3.2. Stanowisko badawcze

Jako sprzęt badawczy wykorzystano Raspberry Pi 2 Model B V1.1. Nośnikiem dla systemu operacyjnego i aplikacji była karta microSDHC marki SanDisk Ultra o pojemności 16 GB i klasie szybkości 10.

Zainstalowano system operacyjny Raspbian Stretch Lite wydany 27 czerwca 2018 r. Wybrano wersję Lite, ponieważ nie posiada ona środowiska graficznego, które nie było użyte, a mogłoby mieć wpływ na wyniki. Po instalacji systemu uruchomiono usługę SSH i wszystkie potrzebne interfejsy komunikacyjne, a także zaktualizowano pakiety. Systemem zarządzano przez połączenie SSH przy pomocy programu PuTTY. W systemie znajdował się domyślnie zainstalowany interpreter języka Python 3.5.3. Zainstalowano z pakietów binarnych dostarczonych przez NodeSource najnowszą wersję Node.js w wersji 8.11.4, ponieważ wersja 10 nie chciała

działać z jedną z bibliotek dla Node.js, a w repozytorium systemu nie była dostępna żadna nowsza wersja. Według instrukcji ze strony biblioteki Unosquare.RaspberryIO [11] zainstalowano środowisko .NET Core. Zainstalowane SDK było w wersji 2.1.401, a środowisko uruchomieniowe w wersji 2.1.3.

Komponenty sprzętowe podłączano przez taśmę 40-pin znaną z napędów standardu IDE, do której podłączano krótkie przewody ze złączami męskimi, a ich końce do płytki stykowej, na której umieszczano elementy. Podłączono zasilacz z wyjściem MicroUSB o napięciu 5V i wydajności 2A. Podłączono również przewód sieci Ethernet, a komputer sterujący był podłączony przez sieć bezprzewodową Wi-Fi 802.11g.

Aplikacje dla .NET Core skompilowano jeszcze na komputerze przy pomocy polecenia *dotnet publish -c Release*, ponieważ kompilacja ich na Raspberry Pi mogłaby zająć więcej czasu oraz znacznie zużyć kartę pamięci. Z tego samego powodu nie zdecydowano się na kompilację aplikacji do aplikacji typu Self Contained App, czyli posiadających wbudowane środowisko uruchomieniowe, ponieważ ich rozmiar wynosi ok. 50 MB i znajduje się w nich bardzo dużo plików. Zdecydowano się uruchamiać aplikacje w sposób zbliżony do konkurencyjnych technologii, czyli podając polecenie *dotnet* i po nim nazwę pliku DLL z kodem pośrednim danej aplikacji.

3.3. Opracowane aplikacje do testowania

Opracowano następujące aplikacje w trzech technologiach:

1) wyświetlanie napisu *Hello World!* na ekranie konsoli
2) wykonanie prostych operacji matematycznych
3) prosta aplikacja internetowa wyświetlająca napis *Hello World!* po otwarciu w przeglądarce
4) wysyłanie napisu *TEST* za pomocą portu szeregowego
5) włączanie i wyłączanie diody elektroluminescencyjnej podłączonej do GPIO 17
6) odczyt temperatury z czujnika Dallas DS18B20
7) odczyt temperatury i wilgotności z czujnika DHT22
8) odczyt natężenia światła z czujnika BH1750
9) odczyt wartości z przetwornika analogowo-cyfrowego MCP3008 z podłączonym czujnikiem temperatury LM35 i konwersja na napięcie i temperaturę
10) wyświetlanie napisu *TEST* na wyświetlaczu OLED ze sterownikiem SSD1306
11) odczyt identyfikatora tagu Mifare przy pomocy czytnika MFRC522.

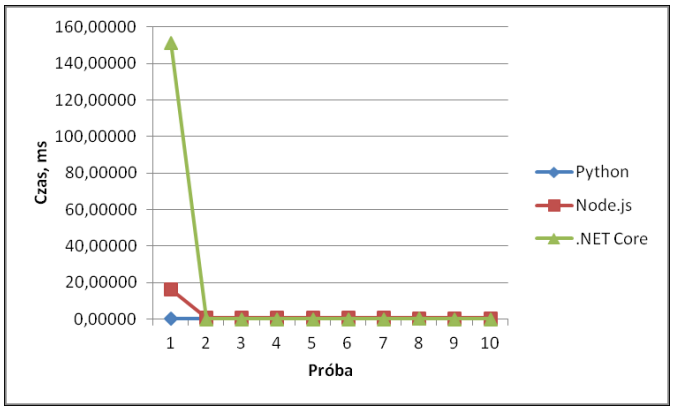
Aplikacja migająca diodą podłączoną do GPIO została opracowana w dwóch wersjach w celu porównania obu bibliotek dla .NET, ponieważ jedna z nich jest jedynie nakładką na WiringPi, a druga bezpośrednio operuje na rejestrach procesora z poziomu języka C#.

4. Wyniki badań

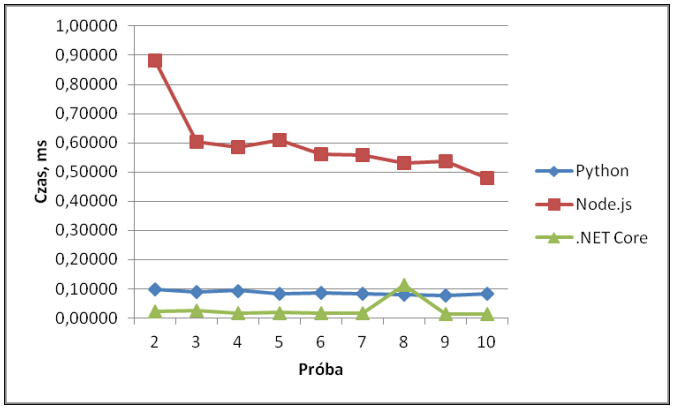
Wszystkie aplikacje udało się uruchomić bez problemów, choć w jednym przypadku aplikacja .NET nie była w stanie odczytać wskazań czujnika DHT22 z powodu implementacji biblioteki i konieczności zachowania sztywnych reguł czasowych przy obsłudze czujnika.

4.1. Wyświetlanie napisu na ekranie

Opracowane aplikacje zadziałały prawidłowo i wyświetliły napis *Hello World!* na konsoli. Na rysunkach 1 i 2 przedstawiono wykresy czasu wykonania. Zdecydowano się na dwa wykresy z powodu dłuższego czasu dla pierwszej próby, który powoduje nieczytelność wykresu dla pozostałych prób.



Rys. 1. Wykres czasu wykonania wyświetlenia napisu dla wszystkich prób



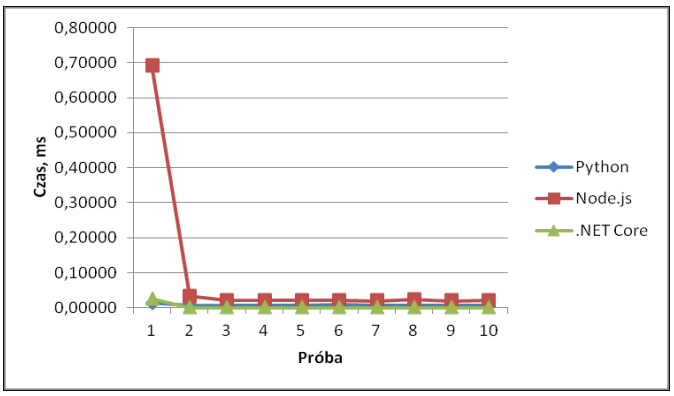
Rys. 2. Wykres czasu wykonania wyświetlenia napisu bez pierwszej próby

Pierwsze wykonanie jest znacznie dłuższe dla .NET Core, ale przy kolejnych próbach .NET Core jest najwydajniejszą technologią. W tym przypadku Node.js znacznie dłużej wykonywał operację wyświetlania napisu niż pozostałe technologie.

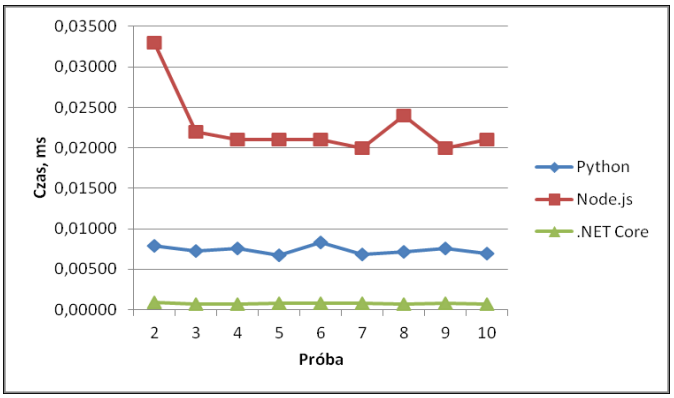
4.2. Podstawowe operacje matematyczne

Opracowane aplikacje wyświetlały jedynie czas wykonania operacji bez prezentowania wyników. Sprawdzono operacje dodawania, odejmowania, mnożenia i dzielenia dwóch liczb: x=0,5 i y=0,25. Na rysunkach 3-7 przedstawiono wykresy dla każdej z operacji.

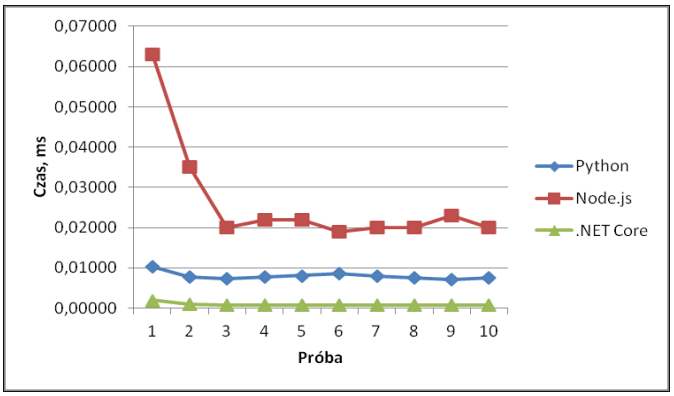
W tym przypadku pierwsze wykonanie jest znacznie dłuższe dla Node.js. Natomiast w każdej sytuacji .NET Core był najwydajniejszą technologią.



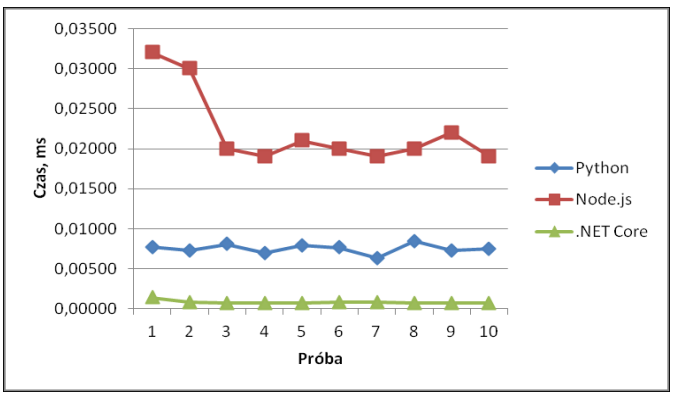
Rys. 3. Wykres czasu wykonania dla wszystkich prób operacji dodawania



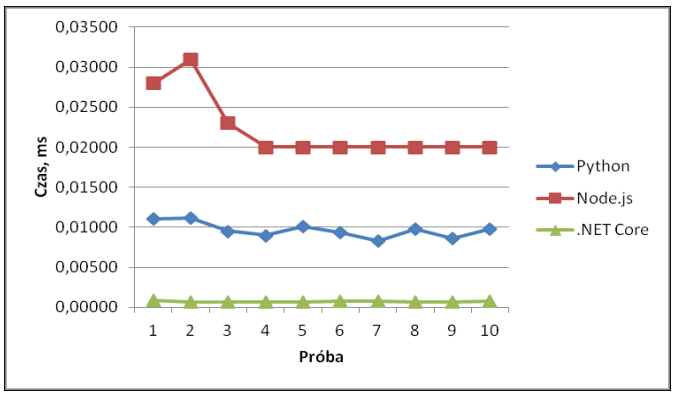
Rys. 4. Wykres czasu wykonania dla operacji dodawania bez pierwszej próby



Rys. 5. Wykres czasu wykonania dla operacji odejmowania



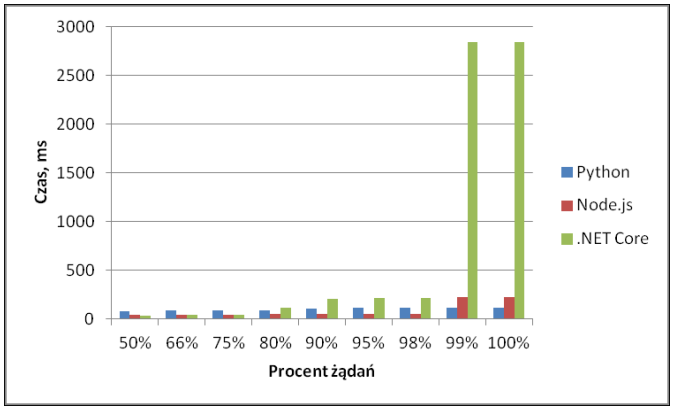
Rys. 6. Wykres czasu wykonania dla operacji mnożenia



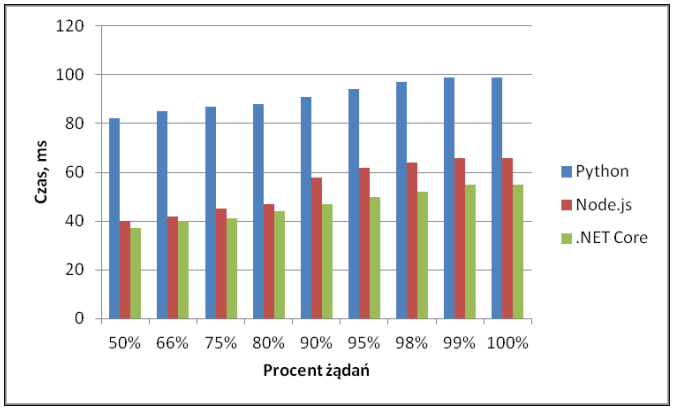
Rys. 7. Wykres czasu wykonania dla operacji dzielenia

4.3. Aplikacja internetowa

Opracowano aplikacje w ten sposób, aby każda nasłuchiwała na porcie TCP 5000. W języku Python napisano aplikację z wykorzystaniem szkieletu programistycznego Flask, w Node.js użyto Express.js, a w .NET Core wykorzystano ASP.NET Core. Ostatnia z technologii powodowała wyświetlanie się tekstu w przeglądarce inaczej niż w pozostałych, ponieważ nie był wysyłany nagłówek *Content-Type* z typem MIME danych, który w pozostałych technologiach był ustawiony na *text/html*.



Rys. 8. Czasy realizacji żądań dla pierwszej próby



Rys. 9. Czasy realizacji żądań dla drugiej próby

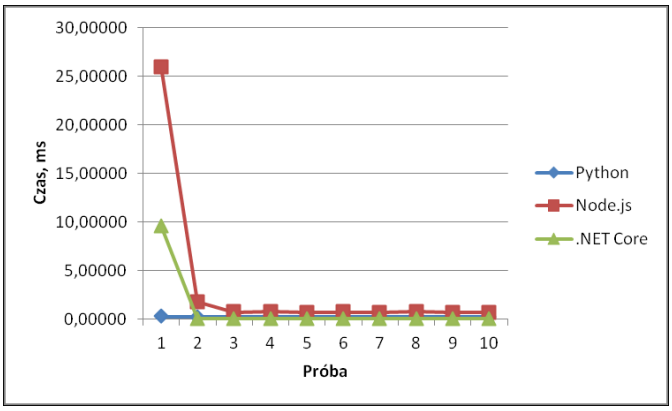
Aplikacje przetestowano programem ApacheBench, który został ustawiony do wysłania 100 żądań w tym 10 powinno

odbyć się jednocześnie. Wykonano dwie próby, aby zaobserwować zachowanie się technologii przy drugiej próbie. Na rysunkach 8 i 9 przedstawiono wykresy z danych generowanych przez program, które pokazują procentowy udział konkretnych czasów obsługi żądań.

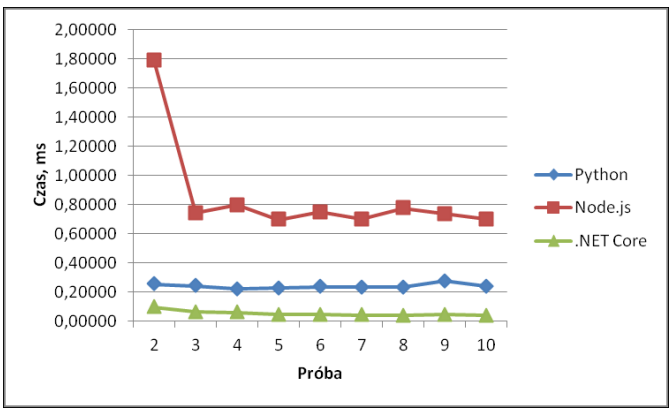
Pierwsza próba dla .NET Core była znacząco mniej wydajna, ale przy drugiej próbie okazała się to najwydajniejsza technologia, a język Python z kolei najmniej wydajny. Jednocześnie z raportu generowanego przez program wynikło, że przy drugiej próbie .NET Core obsłużył ponad 250 żądań/s.

4.4. Wysyłanie napisu przez port szeregowy

Opracowane aplikacje wysyłały napis TEST przez port szeregowy Raspberry Pi, do którego podłączono konwerter USB-UART oparty na układzie PL2303, a wyniki działania aplikacji obserwowano w drugim oknie PuTTY. W tym przypadku użyto biblioteki *NetCoreSerial*, która została pobrana z NuGet, ponieważ oficjalna biblioteka Microsoftu *System.IO.Ports* wspiera obecnie tylko system Windows. Wraz z napisem wysyłano sekwencję CRLF, aby napis wyświetlał się w nowej linii na systemie Windows. Aplikacje działały poprawnie i na ekranie ukazywał się zdefiniowany napis. Na rysunkach 10 i 11 przedstawiono wykresy dla wszystkich prób oraz wariant bez pierwszej próby.



Rys. 10. Wykres czasu wykonania wysłania napisu dla wszystkich prób

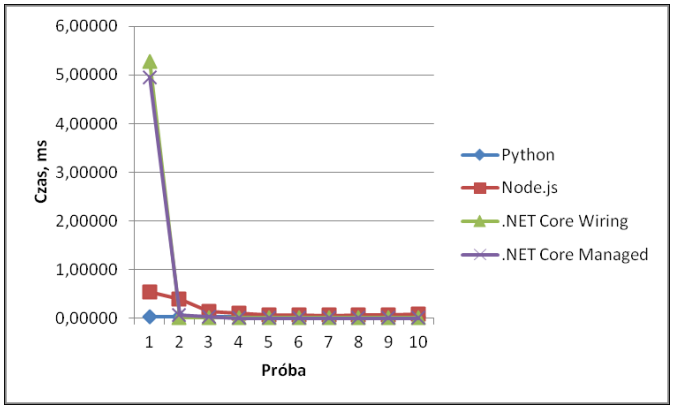


Rys. 11. Wykres czasu wykonania wysłania napisu bez pierwszej próby

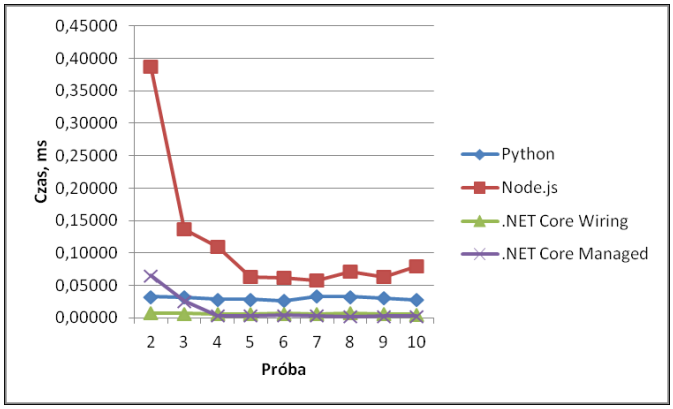
Technologia .NET Core w tym przypadku była najwydajniejsza z wyjątkiem pierwszego wykonania, które i tak było znacznie szybsze niż w przypadku Node.js, a ta technologia była najmniej wydajna.

4.5. Włączanie i wyłączanie diody

Aplikacje inicjalizowały pin GPIO 17 jako wyjście, a potem cyklicznie włączały i wyłączały diodę czerwoną podłączoną do GPIO 17 przez rezystor 200 Ω. Zmierzono również czas inicjalizacji dla każdej z technologii. Czas ten był najkrótszy dla języka Python i wynosił 0,14958 ms, dla Node.js 9,83300 ms, a dla .NET Core 693,73150 ms dla aplikacji korzystającej z biblioteki Unosquare.RaspberryIO oraz 260,63710 ms dla aplikacji używającej biblioteki Raspberry# IO. Pierwsza z bibliotek wykorzystuje WiringPi, a druga bezpośrednio operuje na rejestrach układu Broadcom BCM2836. Na rysunkach 12 i 13 przedstawiono wykresy czasu wykonania. Pierwszą aplikację oznaczono jako .NET Core Wiring, a drugą jako .NET Core Managed, ponieważ używa kodu zarządzanego.



Rys. 12. Wykres czasu wykonania przełączania diody dla wszystkich prób



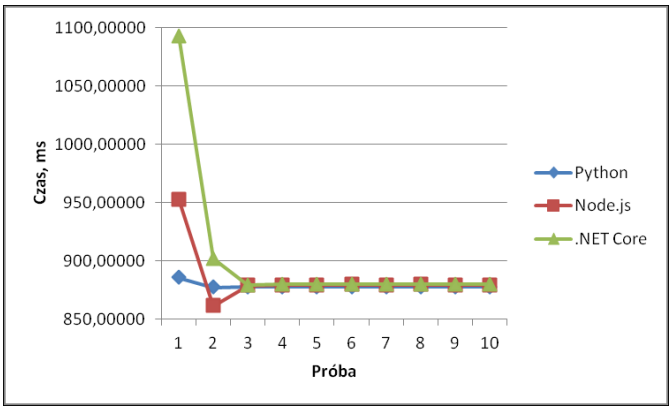
Rys. 13. Wykres czasu wykonania przełączania diody bez pierwszej próby

W przypadku aplikacji .NET Core pierwsze wykonanie było najdłuższe i to w przypadku obu metod, choć pierwsza aplikacja była nieznacznie wolniejsza. Dla kolejnych prób .NET był najwydajniejszy przy obu metodach, ale nieznaczną przewagę miała aplikacja wykorzystująca operacje na rejestrach procesora. Node.js z kolei był najmniej wydajny. Wykazuje to, że .NET Core jest bardzo wydajny przy

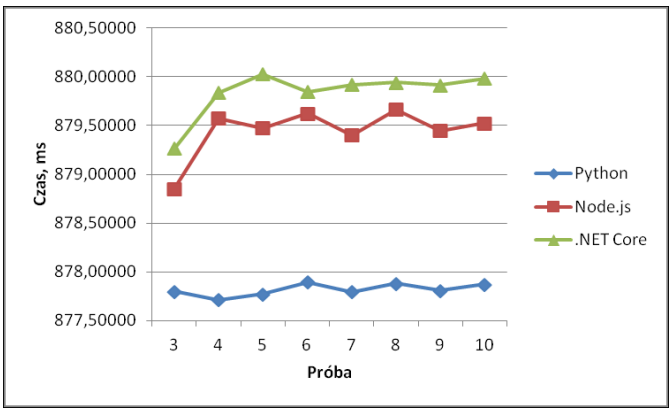
operacjach realizowanych w jak największym stopniu w języku C#, a mniej przy operacjach związanych z wykonaniem kodu z bibliotek natywnych.

4.6. Odczyt temperatury z czujnika Dallas DS18B20

Odczyt temperatury realizowany jest bez jakichkolwiek bibliotek poza standardową biblioteką dla każdej technologii, ponieważ jest to prosty odczyt i przetworzenie zawartości pliku udostępnionego przez moduł jądra systemu operacyjnego. Unikalny identyfikator czujnika został wpisany do każdej aplikacji w kodzie. Zmierzone wartości temperatury były zbliżone do siebie. Na rysunkach 14 i 15 przedstawiono wykresy czasu wykonania.



Rys. 14. Wykres czasu wykonania odczytu temperatury dla wszystkich prób

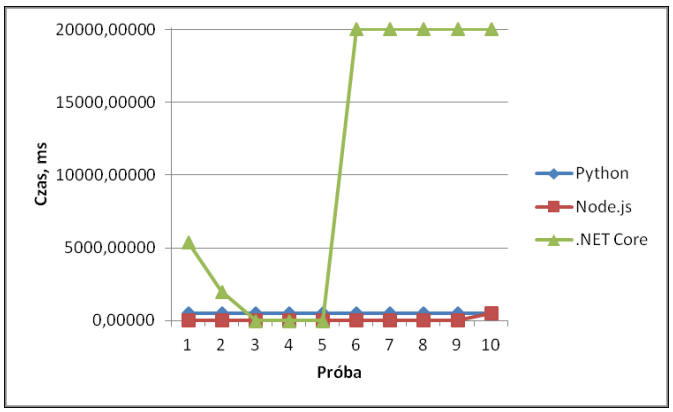


Rys. 15. Wykres czasu wykonania odczytu temperatury bez pierwszej próby

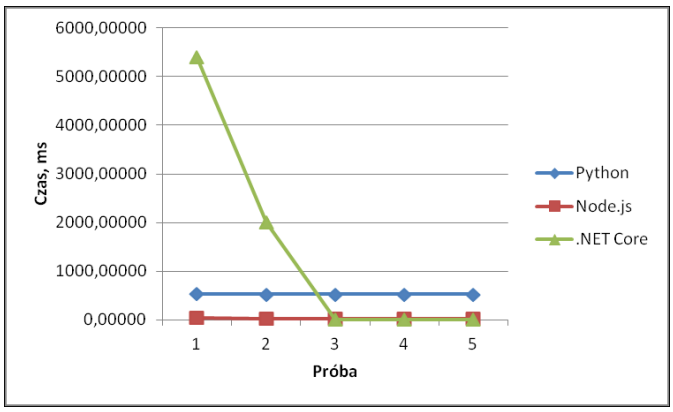
W tym przypadku .NET Core był najwolniejszy. Nieznacznie szybszy był Node.js, a najwydajniejszy okazał się Python. Czasy te jednak mogą być zaburzone przez konieczność komunikacji z czujnikiem przez jądro systemowe, a sam czujnik nie jest zbyt szybki.

4.7. Odczyt temperatury i wilgotności z czujnika DHT22

Jest to jedyny test, który się nie powiódł dla technologii .NET Core. Jedynie pierwsze wykonanie dało pozytywny rezultat i zbliżone wyniki. Kolejne cztery próby dawały nieprawdziwe wartości temperatury i wilgotności, a ostatnie 5 prób w ogóle nie powiodło się, ponieważ wystąpił błąd odczytu. Czujnik ten wymaga bardzo sztywnych zależności czasowych na linii GPIO. Wykorzystano do tego bibliotekę Raspberry# IO. Na rysunkach 16 i 17 przedstawiono wykresy dla wszystkich oraz pierwszych pięciu prób.



Rys. 16. Wykres czasu wykonania odczytu temperatury i wilgotności dla wszystkich prób



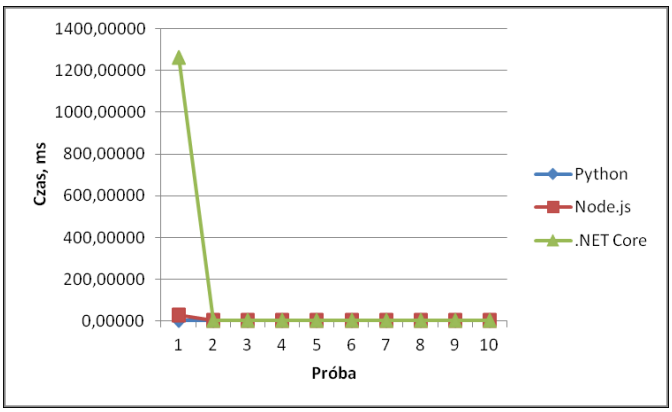
Rys. 17. Wykres czasu wykonania odczytu temperatury i wilgotności dla pięciu pierwszych prób

W tym przypadku technologia .NET była najmniej użyteczna pomimo szybkich odczytów wartości nieprawidłowych. Odczyt wartości prawidłowej był najdłuższy i trwał ponad 5 sekund. Konkurencyjne technologie posiadały biblioteki korzystające z natywnych bibliotek, a .NET Core korzystał jedynie z techniki opisanej wcześniej przy obsłudze GPIO.

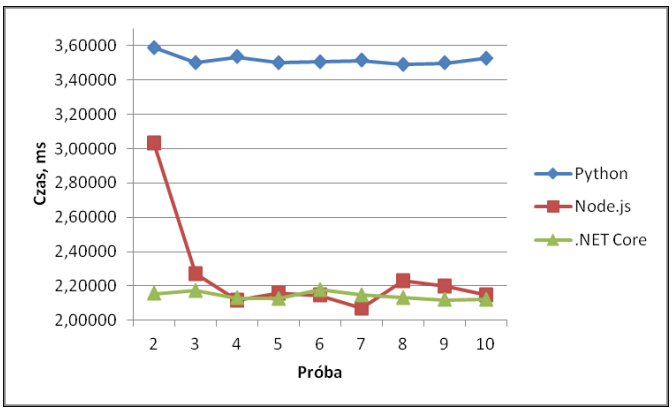
4.8. Odczyt natężenia światła z czujnika BH1750

Ten czujnik podłączono przez sprzętowy interfejs I2C. Na rysunkach 18 i 19 przedstawiono wykresy czasu wykonania. Uzyskiwane wartości pomiaru były zbliżone do siebie.

Pierwsze wykonanie jest najdłuższe dla .NET Core, ale już kolejne wykonania odczytu są zbliżone do Node.js i znacznie wydajniejsze niż w przypadku języka Python. Aplikacja .NET Core wykorzystywała bibliotekę Raspberry# IO, która I2C obsługuje bezpośrednio przez rejestry procesora, a pozostałe technologie korzystają z funkcji systemowych.



Rys. 18. Wykres czasu wykonania odczytu natężenia światła dla wszystkich prób



Rys. 19. Wykres czasu wykonania odczytu natężenia światła bez pierwszej próby

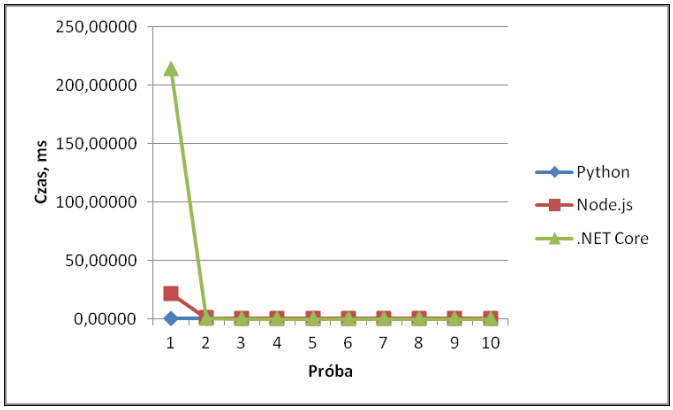
4.9. Odczyt wartości z przetwornika A/C MCP3008

Wymagane było stworzenie własnej klasy obsługującej przetwornik w bibliotece Raspberry# IO, aby obsługiwała sprzętowe SPI, ponieważ domyślna implementacja zakładała tylko SPI programowe. Biblioteka wspiera oba rodzaje implementacji SPI, więc to wykorzystano. Własną klasę opracowano na podstawie oryginalnej klasy, ale zmieniono procedury komunikacji przez interfejs SPI. Do kanału 0 przetwornika podłączono czujnik analogowy LM35, który wytwarza na wyjściu napięcie zmieniające się o 10 mV na każdy stopień Celsjusza. Uzyskiwano wartości zbliżone do siebie, choć mogły one się różnić w zależności od zakłóceń, a także dokładności samego czujnika i przetwornika. Wpływ mogła mieć również dokładność napięcia 3,3V. Na rysunkach 20 i 21 przedstawiono wykresy czasu wykonania.

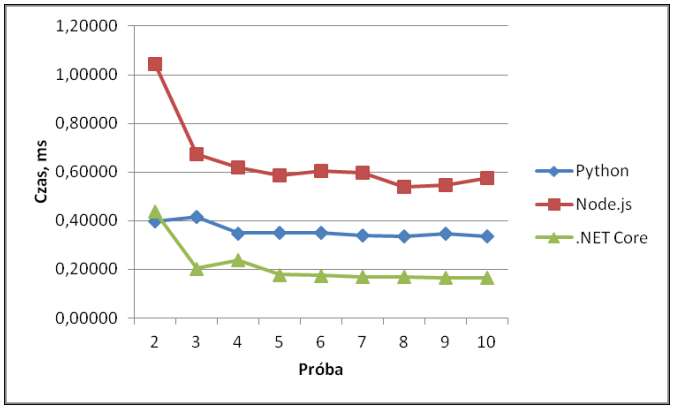
Również i tym razem pierwsze wykonanie dla .NET Core było najdłuższe, ale już kolejne były wydajniejsze, a z kolei Node.js był najmniej wydajny.

4.10. Wyświetlanie napisu na wyświetlaczu OLED

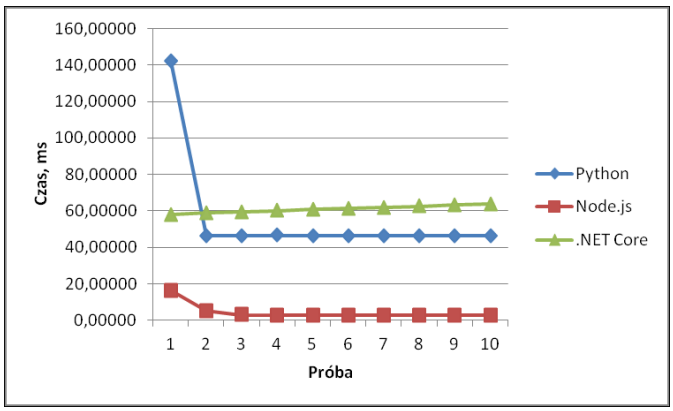
Aplikacje wyświetlały napis TEST na wyświetlaczu OLED 128x64 opartym na sterowniku SSD1306. Znów zaszła konieczność modyfikacji biblioteki Raspberry# IO, ponieważ domyślna implementacja wykorzystywała interfejs I2C. Wyświetlacz natomiast posiadał jedynie SPI, choć istnieje możliwość zmiany wykorzystywanego interfejsu przez wlutowanie rezystorów stanowiących zworki w odpowiednim miejscu. Wykorzystano SPI sprzętowe jak w poprzednim przypadku, ponieważ pozostałe aplikacje również wykorzystywały sprzętowy interfejs SPI. Na rysunkach 22 i 23 przedstawiono wykresy czasu wykonania.



Rys. 20. Wykres czasu wykonania odczytu wartości dla wszystkich prób

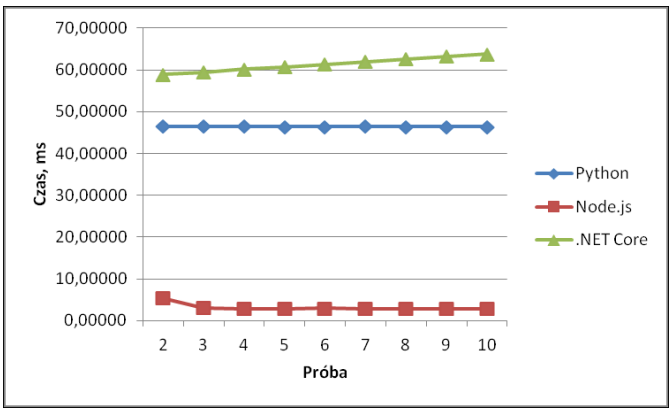


Rys. 21. Wykres czasu wykonania odczytu wartości bez pierwszej próby



Rys. 22. Wykres czasu wykonania wyświetlenia napisu dla wszystkich prób

W tym przypadku najwydajniejsza okazała się technologia Node.js, a najmniej wydajny był .NET Core, choć pierwsze wykonanie było znacznie dłuższe w przypadku języka Python. Również czas inicjalizacji dla Pythona wyniósł 13,97596 ms, dla Node.js 51,00600 ms, a dla .NET Core 473,25510 ms. Zaobserwowano również niewielkie wydłużanie się czasu w przypadku .NET Core w kolejnych próbach.

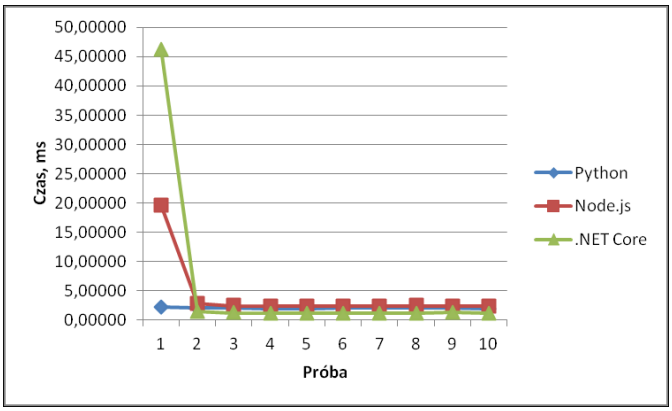


Rys. 23. Wykres czasu wykonania wyświetlenia napisu bez pierwszej próby

Biblioteka w przypadku języka Python w ogóle nie przewiduje wyświetlania tekstu, ale jest to realizowane przez inne biblioteki, które wytwarzają obraz gotowy do wysłania przez bibliotekę obsługującą wyświetlacz. Pozostałe technologie posiadają wbudowaną w biblioteki tablicę zawierającą wartości do wysłania dla każdego znaku ASCII.

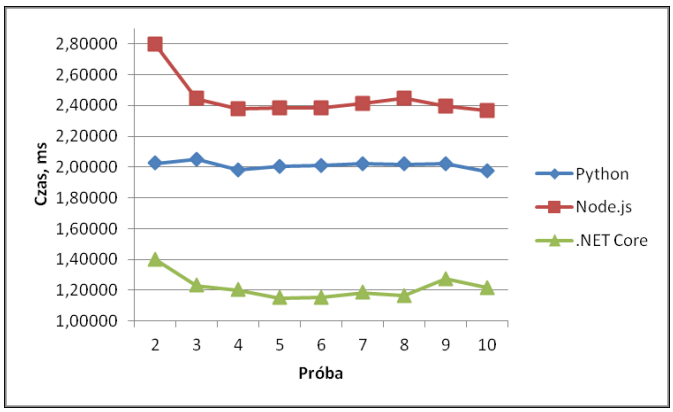
4.11. Odczyt identyfikatora tagu Mifare przy pomocy czytnika MFRC522

Po przyłożeniu breloczka zbliżeniowego załączonego wraz z modułem czytnika odczytywany był 10 razy unikalny identyfikator tagu. Aplikacje oczekiwały na zbliżenie znacznika i dopiero wtedy odczytywany był identyfikator. Wykorzystano tym razem bibliotekę Unosquare.RaspberryIO. Na rysunkach 24 i 25 przedstawiono wykresy czasu wykonania.



Rys. 24. Wykres czasu wykonania odczytu identyfikatora dla wszystkich prób

Z wyjątkiem pierwszego wykonania .NET Core był najwydajniejszy i wyraźnie szybszy od języka Python i Node.js. Najwolniejszy okazał się Node.js, a nieznacznie od niego był wolniejszy Python. Nie sprawdzono sytuacji w przypadku zbliżenia w tym samym czasie wielu znaczników Mifare.



Rys. 25. Wykres czasu wykonania odczytu identyfikatora bez pierwszej próby

5. Wnioski

Uzyskane wyniki świadczą o tym, że technologia .NET Core w obecnym stadium rozwoju jest obiecującą technologią do zastosowania w urządzeniach o niewielkiej mocy obliczeniowej, a do takich należy Raspberry Pi 2. Z wyjątkiem dość długich pierwszych wywołań była to w większości technologia najwydajniejsza. Świadczy to o dobrej optymalizacji środowiska uruchomieniowego. Optymalizacji powinien jednak zostać poddany kompilator JIT, aby szybciej kompilował kod pośredni do kodu maszynowego, choć na wydajność ma wpływ modułowa budowa .NET Core, ponieważ załadowanie jednego pliku wykonywalnego pociąga za sobą załadowanie oraz kompilację w locie plików będących zależnościami. Można jednak skompilować samodzielnie pliki do kodu maszynowego lub wykorzystać CoreRT, którego wynikiem jest niewielki plik niezależny od .NET i zawierający kod maszynowy. Druga technika jednak może prowadzić do problemów w przypadku wykorzystywania mechanizmów refleksji. Technologia ASP.NET Core posiada bardzo wydajny serwer WWW napisany w języku C#.

Dostępne biblioteki bardzo dobrze obsługują interfejsy oraz komponenty. Wyjątek stanowi jedynie czujnik DHT22, choć prawdopodobnie w przypadku próby napisania aplikacji w pozostałych technologiach bez użycia natywnego modułu efekt mógłby być podobny, co w przypadku .NET Core. Niektóre aplikacje wymagały modyfikacji kodu jednej z bibliotek, aby w ogóle mogły działać lub było możliwe porównanie wydajności obsługi interfejsu sprzętowego.

Technologia .NET Core rozwija się bardzo szybko. Zawdzięcza to udostępnieniu kodu źródłowego na licencji open source, co umożliwia jej rozwój przez firmy i osoby prywatne, a nie tylko przez Microsoft. Do niedawna .NET kojarzono jedynie z zamkniętą technologią, która mogła być uruchomiona jedynie pod systemem Windows.

Literatura

[1] https://pl.wikipedia.org/wiki/Internet_rzeczy [12.09.2018]

[2] https://pl.wikipedia.org/wiki/Raspberry_Pi [12.09.2018]

[3] Singh K.J., Kapoor D.S. Create Your Own Internet of Things: A survey of IoT platforms. IEEE Consumer Electronics Magazine, 6(2), (2017), 57-68

[4] Upton E., Halfacree G., Raspberry Pi. Przewodnik Użytkownika, Helion, 2013.

[5] https://pl.wikipedia.org/wiki/Python [12.09.2018]

[6] https://pl.wikipedia.org/wiki/Node.js [12.09.2018]

[7] https://en.wikipedia.org/wiki/.NET_Core [12.09.2018]

[8] https://pl.wikipedia.org/wiki/Mono_(projekt) [12.09.2018]

[9] Pańczyk B., Badurowicz M., Programowanie obiektowe - Język C#, Politechnika Lubelska, 2013.

[10] https://pl.wikipedia.org/wiki/C_Sharp [12.09.2018]

[11] https://github.com/unosquare/raspberryio [12.09.2018]

[12] https://github.com/raspberry-sharp/raspberry-sharp-io [12.09.2018]