Zeszyty Naukowe WWSI, No 19, Vol. 12, 2018, s. 51-77
DOI: 10.26348/znwwsi.19.51

Porównanie czasów wykonywania funkcji natywnych w aplikacjach mobilnych zaimplementowanych w technologiach natywnej i hybrydowej

Dariusz Bzdyk, Robert Janowski*

Warszawska Wyższa Szkoła Informatyki

Abstrakt

W pracy przedstawiono zagadnienie oceny wydajności aplikacji tworzonych w modelach natywnym oraz hybrydowym. Analizę porównawczą przeprowadzono przyjmując jako kryterium czas wykonywania funkcji natywnych, takich jak np. dostęp do sprzętu, dostęp do sieci, zapis i odczyt danych z wykorzystaniem mechanizmów specyficznych dla konkretnego systemu operacyjnego. Pomiary wykonano przygotowując dwie jednakowe pod względem funkcjonalnym aplikacje dla systemu operacyjnego Android, jedną w języku Java (metodologia natywna), drugą w języku JavaScript i HTML z wykorzystaniem mostu PhoneGap (metodologia hybrydowa), w których wywoływano określone funkcje natywne i mierzono czas ich zakończenia. Badania wykonano dla kilku wersji systemu operacyjnego Android w celu uzyskania szerszego poglądu na analizowane zagadnienie.

Słowa kluczowe **–** Aplikacje hybrydowe, aplikacje natywne, Android OS, PhoneGap

* E-mail: rjanowski@poczta.wwsi.edu.pl

Zgłoszono do druku 5 listopada 2018 r.

1. Wstęp

Obecnie rynek smartfonów zdominowany jest przez dwa systemy operacyjne: Android i iOS. Według raportu Gartnera [1] stanowiły one w skali globalnej odpowiednio 81,7% i 17,9% sprzedaży smartfonów w czwartym kwartale 2016 roku. System Windows jest zmarginalizowany i jego udział w sprzedaży wynosi zaledwie 0,3%.

Jednak 10 lat temu, gdy zaczęły pojawiać się pierwsze smartfony, sytuacja wyglądała zupełnie inaczej. Pod względem systemów operacyjnych rynek smartfonów był bardziej zróżnicowany. W roku 2007 oprócz systemów Android OS i iOS w użyciu były systemy Bada OS (firmy Samsung), BlackBerry oraz Windows Phone (wcześniej Windows Mobile). Systemy operacyjne dla urządzeń mobilnych (smartfonów, tabletów) różnią się architekturą, wykorzystywanymi językami programowania, a także narzędziami do tworzenia aplikacji. Na przykład aplikacje dla systemu operacyjnego Android OS tworzone są w języku Java [2], a dla systemu iOS w języku Objective-C [3] lub Swift [4].

Nabycie odpowiednich kompetencji związanych z tworzeniem aplikacji dla urządzeń mobilnych wymaga długotrwałej nauki odpowiedniego języka programowania, znajomości interfejsu programistycznego (API) specyficznego dla danego systemu operacyjnego oraz opanowania narzędzi wykorzystywanych w procesie tworzenia aplikacji, np. obsługi środowiska IDE (Integrated Development Environment).

Wymienione aspekty powodują, że programiści zajmujący się wytwarzaniem aplikacji na urządzenia mobilne specjalizują się w konkretnym systemie operacyjnym. To z kolei powoduje, że podmioty gospodarcze, chcąc dotrzeć ze swoim produktem software'owym do klientów używających np. 3 różnych systemów operacyjnych, muszą stworzyć tą samą aplikację 3 razy, osobno dla każdego systemu operacyjnego. W efekcie prowadzi to do zwiększenia kosztów wytwarzania, a także utrzymania (np. poprawiania, uaktualniania, rozbudowywania) aplikacji. Im większa dywersyfikacja systemów operacyjnych wśród użytkowników urządzeń mobilnych, tym problem ten staje się coraz bardziej dotkliwy.

Z tego powodu około 10 lat temu pojawiły się koncepcje pozwalające na jednokrotne wytworzenie aplikacji, a następnie jej kompilację i wygenerowanie programów wykonywalnych mogących działać na różnych systemach operacyjnych. Jednym z rozwiązań było tworzenie aplikacji w sposób podobny do rozwoju aplikacji

Webowych, czyli z użyciem języka JavaScript do kodowania logiki oraz języka HTML i stylów CSS do wykonania interfejsu graficznego. Należy przy tym zaznaczyć, że w przeciwieństwie do aplikacji webowych, tak stworzone aplikacje miały również umożliwiać korzystanie z funkcji natywnych urządzeń mobilnych, tzn. sprzętu lub specyficznych struktur przechowania danych, czyli ze wszystkich funkcji, do których jest dostęp z tzw. aplikacji natywnych, tj. tworzonych z wykorzystaniem dedykowanego języka oraz API specyficznego dla danego systemu operacyjnego.

Ze względu na łączenie cech aplikacji natywnych (dostęp do wszystkich funkcji systemu operacyjnego poprzez specjalizowane API) oraz cech aplikacji webowych (tworzenie w języku JavaScript i HTML z użyciem stylów CSS oraz uniwersalność w sensie możliwości działania w różnych systemach operacyjnych), aplikacje tworzone w ten sposób określano mianem aplikacji hybrydowych. W założeniu aplikacje hybrydowe miały stanowić optymalne rozwiązanie kwestii tworzenia aplikacji w heterogenicznym środowisku systemów operacyjnych, minimalizując czas i koszty rozwoju aplikacji przy utrzymaniu pełnej funkcjonalności tj. dostępu do wszystkich funkcji oferowanych przez system operacyjny. W realizacji okazuje się jednak, że są podstawy do tego, aby twierdzić, że aplikacje hybrydowe ustępują aplikacjom natywnym pod względem wydajności.

W dalszej części opracowania, w rozdziale 2, została przedstawiona geneza problemu poruszanego w pracy z odwołaniem do architektury aplikacji natywnych i hybrydowych, a także przegląd literatury związanej z badaniem wydajności aplikacji obu typów. W rozdziale 3 przedstawiono metodę oraz wyniki porównania wydajności poprzez pomiar czasu wykonania tych samych funkcji natywnych w aplikacjach hybrydowych i natywnych. Rozdział 4 podsumowuje pracę i zawiera wnioski wyciągnięte z analizy porównawczej udokumentowanej w rozdziale 3.

2. Geneza problemu porównywania wydajności aplikacji natywnych i hybrydowych

Celem koncepcji rozwoju aplikacji w modelu hybrydowym było uniknięcie wielokrotnego tworzenia kodu tej samej aplikacji, ze względu na specyfikę systemów operacyjnych, które udostępniają różne zestawy funkcji (API) oraz wymagają użycia

innych języków programowania. W realizacji ta uniwersalność podejścia hybrydowego pozwalającego na napisanie kodu w języku skryptowym, np. Java Script, a interfejsu w języku HTML, z możliwością wytworzenia aplikacji na dowolny system operacyjny, powoduje pewne skomplikowanie architektury rozwiązania.

2.1. Architektura aplikacji natywnych kontra hybrydowych

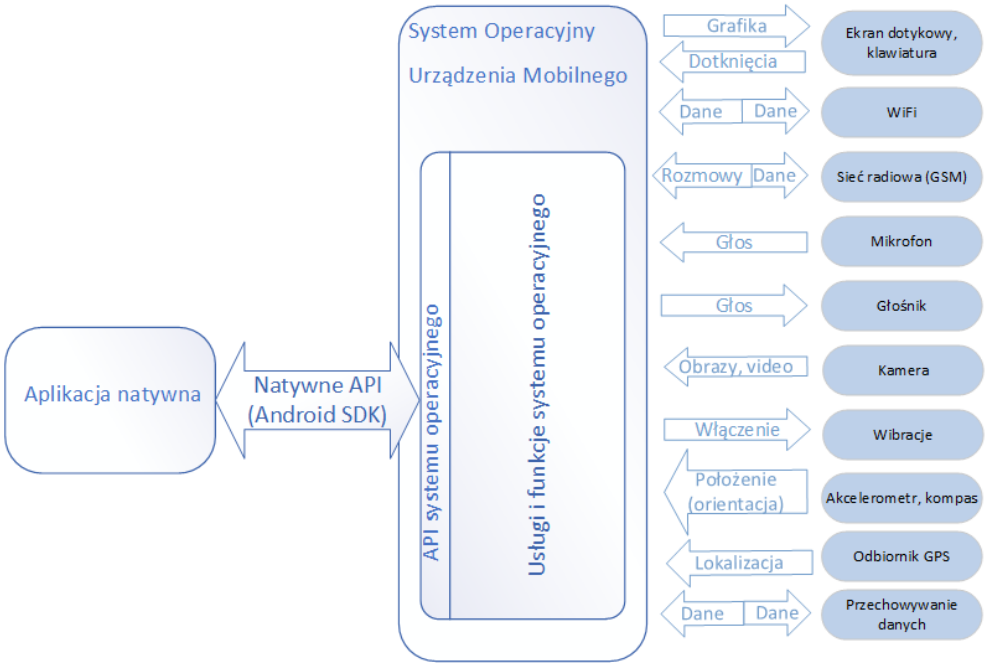
Aplikacje natywne mają bezpośredni dostęp do wszystkich funkcji oferowanych przez system operacyjnym za pomocą API udostępnionego przez producenta. Często też aplikacje natywne mają dostęp do unikalnych cech lub funkcji specyficznych dla danego systemu operacyjnego, np. wyglądu elementów GUI (Graphical User Interface).

Stworzenie aplikacji natywnej wymaga napisania kodu w określonym dla danego systemu operacyjnego języku programowania, a także dodania wymaganych przez dany system operacyjny specyficznych dla niego plików z deklaracjami, np. AndroidManifest.xml dla systemu Android. Kod aplikacji jest kompilowany do postaci wykonywalnej za pomocą narzędzi również specyficznych dla danego systemu operacyjnego, określanych mianem SDK (Software Development Kit). Uruchomiona aplikacja korzysta z możliwości oferowanych przez system operacyjny poprzez wywoływanie funkcji z API. Schemat opisanej interakcji pomiędzy aplikacją a systemem operacyjnym pokazano na rysunku 1.

Aplikacje hybrydowe tworzone są w podobny sposób, jak aplikacje webowe, tj. ich kod powstaje w języku skryptowym, np. JavaScript, a graficzny interfejs użytkownika w języku HTML. W odróżnieniu jednak od aplikacji webowych, aplikacje hybrydowe mają pełny dostęp do funkcji oferowanych przez system operacyjny urządzenia mobilnego. Jest to możliwe dzięki specjalnej architekturze tych aplikacji.

Aplikacje hybrydowe są kompilowane do kodu bajtowego, a w wyniku otrzymywany jest plik gotowy do instalacji w wybranym systemie operacyjnym, np. plik z rozszerzeniem apk dla systemu Android. Jest to możliwe dlatego, że kod pisany w języku JavaScript korzysta z funkcji udostępnianych przez odpowiedni most, np. Cordova [5] lub PhoneGap [6], który wywołuje funkcje z natywnego API odwołujące

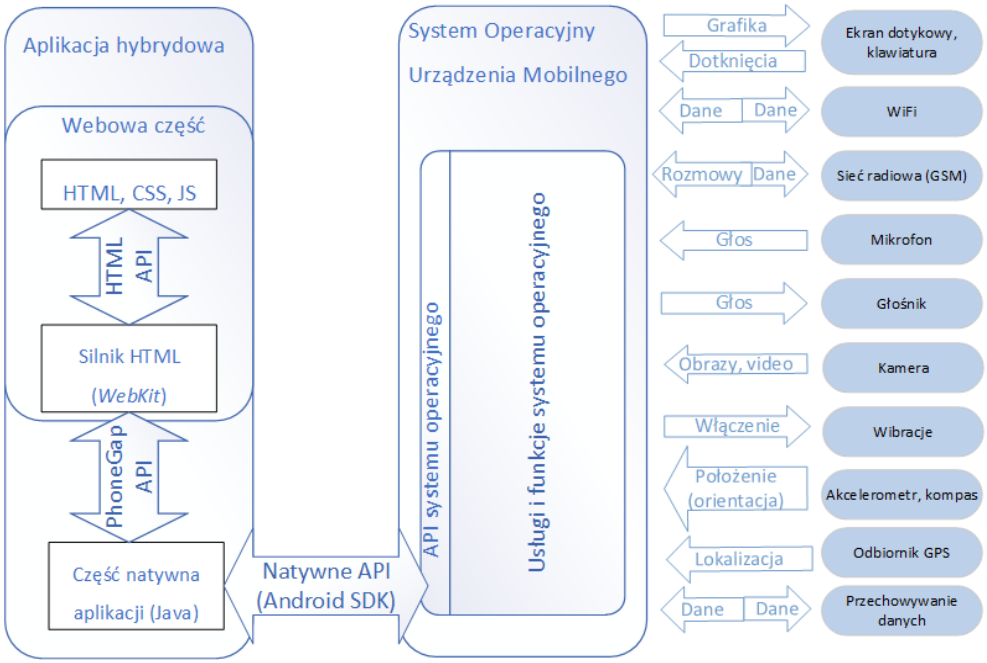
się bezpośrednio do systemu operacyjnego. Oznacza to, że most, np. Cordova lub PhoneGap, oferuje API w języku JavaScript opakowujące funkcje natywne i w trakcie wyboru docelowej platformy dla tworzonej aplikacji, dokonuje auto-generowania kodu natywnego, który ostatecznie podlega kompilacji.



Rysunek 1. Schemat interakcji aplikacji natywnej z systemem operacyjnym (na podstawie [7])

Graficzny interfejs użytkownika implementowany w języku HTML i CSS, jest uruchamiany w natywnym kontenerze (w systemie Android jest to *WebView*), który interpretuje kod HTML za pomocą wewnętrznej przeglądarki (*Rendering Engine*), a działania inicjowane przez wybory użytkownika w GUI, które wymagają odwołania się do funkcji natywnych, przekazuje na poziom kodu natywnego w celu ich realizacji.

Schemat opisanej interakcja aplikacji hybrydowej z systemem operacyjnym został zilustrowany na rysunku 2.



Rysunek 2. Schemat interakcji aplikacji hybrydowej z systemem operacyjnym (na podstawie [7])

2.2. Przegląd prac dotyczących porównania wydajności aplikacji natywnych i hybrydowych

Porównanie architektury aplikacji natywnych i hybrydowych opisanych w rozdziale 2.1 szybko nasuwa spostrzeżenie dotyczące wydajności ich działania. Ze względu na fakt, że aplikacje hybrydowe implementują GUI w postaci stron HTML, każde odwołanie z GUI do funkcji natywnych wymaga przeniesienia sterowania z wewnętrznej przeglądarki (*Rendering Engine*), wywołania odpowiedniej funkcji opakowującej w języku JavaScript, wywołania właściwej funkcji natywnej, wykonania tej funkcji, a następnie przekazania wyniku kolejno do kodu JavaScript poprzez zwrotne wywołanie funkcji opakowującej, a z niej przekazania sterowania do GUI. Można spodziewać się, że taki schemat działania będzie skutkować wydłużonym

czasem wykonywania funkcji natywnych z kodu aplikacji hybrydowych. Z tego powodu wkrótce po udostępnieniu narzędzi do tworzenia aplikacji w modelu hybrydowym temat oceny wydajności tych aplikacji stał się jednym z przedmiotów prac badawczych związanych z architekturą aplikacji hybrydowych.

W pracy [8] autorzy dokonali oceny wydajności działania aplikacji napisanych w modelach natywnym i hybrydowym dla dwóch systemów operacyjnych: Android i Windows Phone. W aplikacjach pisanych w modelu hybrydowym wykorzystywali oprogramowanie PhoneGap [6] jako most pomiędzy kodem pisanym w języku JavaScript, a funkcjami natywnymi systemów operacyjnych (Android i Windows Phone). Jako kryterium porównania, autorzy przyjęli 3 miary związane z aplikacjami, tj. czas uruchomienia, rozmiar zajmowanej pamięci RAM oraz rozmiar aplikacji na dysku. Dla systemu operacyjnego Android korzystniejsze wyniki zostały otrzymane dla aplikacji hybrydowej (poprawa o ok. 30%, 20% i 70% w stosunku do aplikacji natywnej odpowiednio dla czasu uruchamiania aplikacji, rozmiaru zajmowanej pamięci RAM oraz rozmiaru aplikacji na dysku). Wyniki dla systemu Windows Phone były przeciwne, tj. wykazały pogorszenie o ok. 100%, 100% i 390% w stosunku do aplikacji natywnej odpowiednio dla czasu uruchamiania aplikacji, rozmiaru zajmowanej pamięci RAM oraz rozmiaru aplikacji na dysku.

Takie same dwa kryteria (czas startu aplikacji oraz wielkość zajmowanej pamięci RAM) zostały przyjęte przez autorów publikacji [9] podczas porównywaniu wydajności aplikacji tworzonych w modelach natywnym i hybrydowym z użyciem platformy *WorkLight* dla systemu operacyjnego Android. Aplikacja hybrydowa została napisana z wykorzystaniem *frameworku jQuery* stworzonego z myślą o efektywnym tworzeniu interfejsów GUI, który jest standardową częścią platformy *WorkLight*.

Pomimo, że charakter przeprowadzanych badań był podobny do wcześniej przeprowadzanych ewaluacji przez innych badaczy, to jednak autorzy pracy [9] przekonywali w niej o ich unikalności z powodu specyficznych warunków wynikających z budowy aplikacji hybrydowej przy użyciu *frameworku jQuery*. Udokumentowane wyniki potwierdziły znaczącą przewagę aplikacji natywnych w sensie przyjętych miar wydajnościowych, a także wykazały dużą zależność otrzymywanych wyników od liczby elementów interfejsu GUI (np. elementów listy) w aplikacji hybrydowej.

Inne kryteria oceny przy porównywaniu aplikacji tworzonych w modelach natywnym i hybrydowym przyjęli autorzy pracy [10]. W swoim opracowaniu skupili

się na ocenie czasu wykonywania funkcji natywnych urządzeń mobilnych działających również jak w poprzednich publikacjach pod kontrolą bardzo rozpowszechnionego systemu operacyjnego Android. Badaniom zostały poddane funkcje dostępu do akcelerometru, wygenerowania powiadomienia dźwiękowego, wygenerowania wibracji, dostępu do danych z czujnika GPS, dostępu do informacji o sieci, zapisu i odczytu pliku, odczytu danych z listy kontaktów. Przygotowane aplikacje pozwalające na wielokrotne wywołanie wybranej funkcji i pomiar czasu jej wykonania zostały uruchomione w systemie operacyjnym Android w wersji 2.2 na telefonie HTC Nexus One. Wyniki zostały zaprezentowane w formie średniej geometrycznej czasu wykonania poszczególnych funkcji. Analiza udokumentowanych w tej pracy wyników wskazuje, że wykonywanie funkcji natywnych w aplikacjach hybrydowych jest wolniejsze niż w aplikacjach natywnych. Jedynym wyjątkiem okazała się funkcja generowania dźwięku powiadomienia, która była wykonywana o ok. 35% szybciej w aplikacji hybrydowej niż natywnej.

W naszym opracowaniu staraliśmy się zweryfikować ogólne przekonanie, że funkcje natywne wywoływane z aplikacji hybrydowych działają wolniej niż te same funkcje wywoływane z aplikacji natywnych. W związku z tym, podobnie jak w pracy [10], badaliśmy czas wykonywania poszczególnych funkcji natywnych. Nasze opracowanie nie jest tylko powtórzeniem badań opublikowanych we wcześniejszych pracach, ale przede wszystkim próbuje pokazać, jak różnice w wydajności wykonywania funkcji natywnych dla różnych modeli tworzenia aplikacji zmieniły się wraz z rozwojem platform mobilnych, tj. wykorzystywania bardziej wydajnego sprzętu i nowych wersji systemu operacyjnego Android.

3. Porównanie wydajności wykonania funkcji w obu modelach programistycznych

W zależności od realizowanego zadania, funkcje natywne są implementowane w różny sposób. Niektóre funkcje jedynie odczytują dane z czujników sprzętowych, inne wywołują określony efekt sprzętowy np. wygenerowanie dźwięku lub wibracji, jeszcze inne odwołują się do danych w sposób bezpośredni, np. zapis lub odczyt pliku lub za pomocą mechanizmów pośrednich np. dostawcy treści (*Content Provider*) w przypadku listy kontaktów w systemie operacyjnym Android. Chcąc przeprowadzić

miarodajną ewaluację wydajności wykonywania funkcji natywnych, zdecydowaliśmy się wybrać do analizy przynajmniej jedną funkcję z każdego typu.

W celu umożliwienia wykonania pomiarów zostały stworzone dwie, funkcjonalnie identyczne, aplikacje dla systemu operacyjnego Android. Różniły się one sposobem implementacji wynikającym z przyjętego modelu programistycznego, tj. jedna aplikacja, tzw. natywna, była pisana w języku Java z wykorzystaniem narzędzi przeznaczonych dla systemu operacyjnego Android, czyli Android Studio z dedykowanym SDK, druga aplikacja tzw. hybrydowa była pisana w języku Java Script z wykorzystaniem języka HTML, stylów CSS oraz oprogramowania PhoneGap stanowiącego most pomiędzy kodem JavaScript a funkcjami natywnymi. Każda z dwóch wersji aplikacji pozwalała na jednokrotne (kontrolne) lub wielokrotne – zgodne z podaną liczbą powtórzeń – wykonanie określonej funkcji natywnej oraz zapisanie czasu wykonania w pliku do późniejszej obróbki statystycznej.

3.1. Implementacja aplikacji natywnej

Dostęp do funkcji natywnych w aplikacji pisanej dla określonego systemu operacyjnego z wykorzystaniem dedykowanego języka programowania i udostępnionego przez producenta SDK polega na wywoływaniu funkcji API, które są bezpośrednio wykonywane jako funkcje systemowe. W systemie Android dostępne są klasy, a wewnątrz nich metody, które pozwalają na dostęp do wszystkich funkcji oferowanych przez terminal mobilny. W aplikacji wykorzystywano funkcje dostępu do czujników: akcelerometru i magnetometru, lokalizacji odczytanej z odbiornika GPS, uruchamiania wibracji, generowania dźwięków systemowych np. dźwięku powiadomienia, zapisu i odczytu danych, dodawania nowych kontaktów i wyszukiwania istniejących, uzyskiwania informacji o statusie sieci.

Dostęp do czujników uzyskiwany jest za pomocą klasy *SensorManager*. Odbiór informacji z czujnika reprezentowanego w kodzie programu przez klasę *SensorManager*, odbywa się poprzez wykorzystanie interfejsu *SensorEventListener*, w którym należy zaimplementować dwie metody:

- *onAccuracyChanged* – wywoływaną, gdy dokładność czujnika została zmieniona oraz

- *onSensorChanged* – wywoływaną, gdy pojawia się nowe zdarzenie wygenerowane przez czujnik. Wygenerowanie nowego zdarzenia oznacza zmianę wartości odczytywanej przez czujnik lub upłynięcie pewnego granicznego czasu (*timeout*) pomimo braku zmiany mierzonej wartości.

Z kolei dostęp do informacji o lokalizacji wymaga wykorzystania klasy *LocationManager* oraz interfejsu *LocationListener*. Implementacja metody *onLocationChanged* interfejsu *LocationListener* pozwala na odczytywanie informacji o położeniu z obiektu *Location*, który jest przekazywany do tej metody, a który zawiera między innymi dane o szerokości i długości geograficznej, wysokości oraz prędkości.

Generowanie wibracji umożliwia klasa *Vibrator*, której instancję uzyskujemy wywołując metodę *getSystemService* z argumentem *VIBRATOR_SERVICE*. Metoda *vibrate(int)* tej klasy uruchamia wibracje na czas określony w przekazanym argumencie. Wygenerowanie dźwięku, np. dzwonka powiadomienia, jest możliwe dzięki klasom *RingtoneManager*, *Ringtone* oraz metodzie *play()*.

Zapis danych do pliku wykonywany jest za pomocą klas *FileOutputStream* oraz *OutputStreamWriter*. Ta ostatnia klasa, opakowująca klasę *FileOutputStream*, służy do skonwertowania strumienia znaków do postaci bajtów. Analogicznie odczyt danych z pliku wykorzystuje klasy *InputStreamReader* do skonwertowania strumienia bajtów do postaci znaków oraz *BufferedReader*, opakowującą klasę *InputStreamReader*, do podniesienia efektywności procesu odczytywania danych z pliku poprzez ich buforowanie, co pozwala na wczytywanie danych większymi porcjami zamiast pojedynczymi bajtami.

Sprawdzenie statusu połączenia sieciowego umożliwiają klasy *ConnectivityManager*, *NetworkInfo* oraz *TelephonyManager*. Klasa *ConnectivityManager* udostępnia metody pozwalajace poznać status połączenia transmisji danych, np. WiFi, GPRS, UMTS. Klasa *NetworkInfo* posiada metody pozwalające na sprawdzenie statusu interfejsu sieciowego. Natomiast klasa *TelephonyManager* zapewnia dostęp do informacji o stanie sieci w odniesieniu do usług głosowych. W sieciach 2G i 3G, a także w sieci 4G, która nie używa techniki VoLTE (*Voice over LTE*), stany te nie są tożsame.

Operacje przeprowadzane na liście kontaktów – dodawanie lub wyszukiwanie kontaktu – różnią się od innych funkcji natywnych opisanych powyżej, ze względu na korzystanie z dodatkowego mechanizmu pośredniczącego w dostępie do danych. Tym pośrednikiem jest *ContentProvider* – klasa oferująca dostęp do modelu danych za pomocą obsługi wywołań kierowanych pod odpowiednie adresy URI.

```
public void onClick(View view) {
        int i=0;
        TimingLogger timings = new TimingLogger("COMP2", "methodA");
        while (Integer.valueOf(wibraIlosc.getText().toString())>i)
        {
           ((Vibrator) getSystemService(VIBRATOR_SERVICE)).vibrate(1000);
           timings.addSplit("Test");
           timings.dumpToLog();
           try {
              Thread.sleep(2000);
        } catch (InterruptedException e) {
              e.printStackTrace();
        }
        timings.reset();
        i++;
     }
   }
});
```

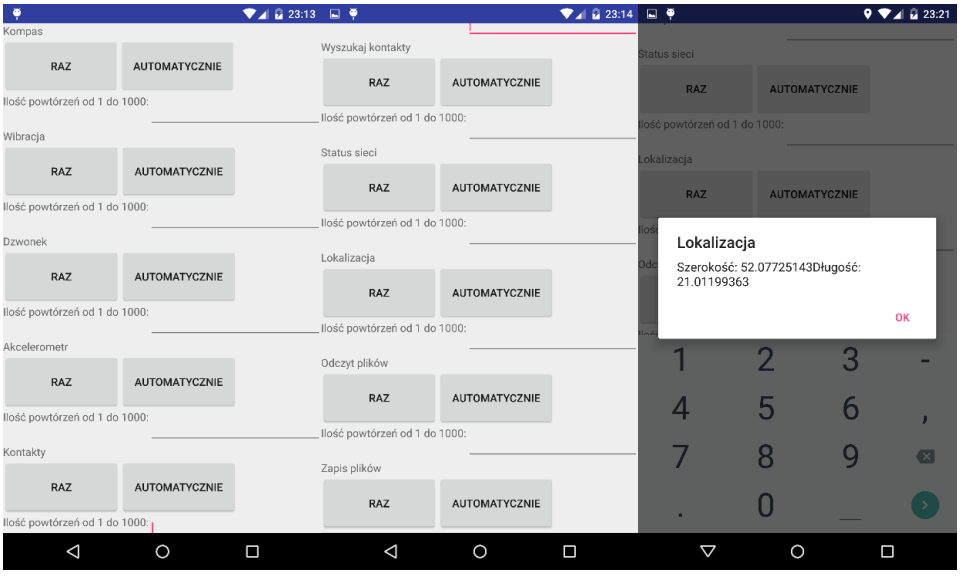
Rysunek 3. Kod źródłowy implementujący wywołanie wibracji

```
public void Lokalizacja(){
   if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M && checkSelfPermission
      Manifest.permission.ACCESS_FINE_LOCATION) !=
      PackageManager.PERMISSION_GRANTED) {requestPermissions(new String[]
         {Manifest.permission.ACCESS_FINE_LOCATION},
            PERMISSIONS_REQUEST_ACCESS_FINE_LOCATION);
   }
   else {
      LocationManager locationManager = (LocationManager)
         getSystemService(LOCATION_SERVICE);
   if (ActivityCompat.checkSelfPermission(MainActivity.this,
            Manifest.permission.ACCESS_FINE_LOCATION) !=
            PackageManager.PERMISSION_GRANTED
            && ActivityCompat.checkSelfPermission(MainActivity.this,
            Manifest.permission.ACCESS_COARSE_LOCATION) !=
            PackageManager.PERMISSION_GRANTED) {
      return;
   }
   locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
   30000, 0, mLocationListener);
   Criteria criteria = new Criteria();
   String bestProvider = locationManager.getBestProvider(criteria, true);
   Location location = locationManager.getLastKnownLocation(bestProvider);
   if (location == null) {
      Toast.makeText(getApplicationContext(), "Brak sygnalu GPS",
      Toast.LENGTH_SHORT).show();
   }
   if (location != null) {
      latitude = location.getLatitude();
      longitude = location.getLongitude();
      onLocationChanged(location);
   }
 }
}
```

Rysunek 4. Kod źródłowy implementujący odczyt lokalizacji z odbiornika GPS

Takie podejście pozwala na uniknięcie ryzyka ewentualnego zaburzenia struktury przechowywanych danych, gdyż daje możliwość jedynie inicjowania określonych działań bez bezpośredniej ingerencji. Jest to także sposób na ograniczenie zakresu dostępu do danych i ich zmiany. Przykładowy kod z implementacją funkcji został przedstawiony na rysunkach 3 i 4.

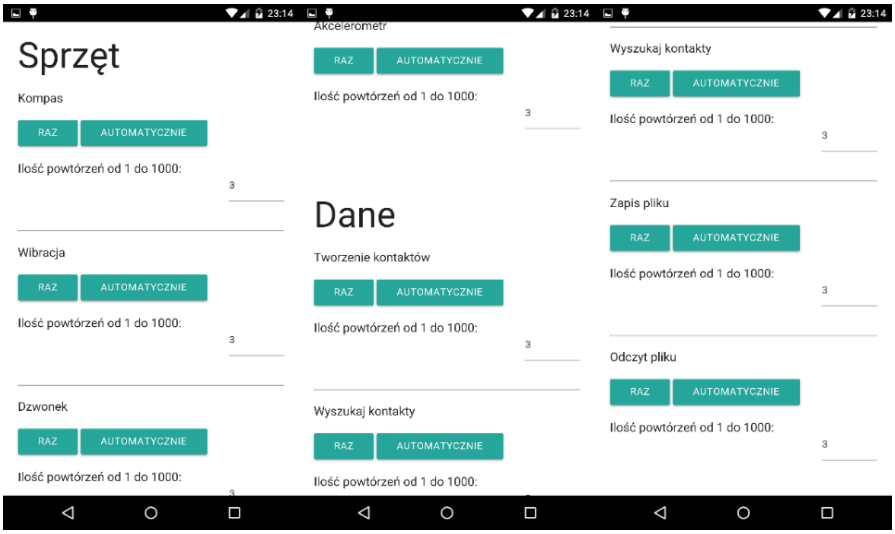
Graficzny interfejs użytkownika (GUI) aplikacji natywnej został, zgodnie z metodologią tworzenia aplikacji natywnej dla systemu operacyjnego Android, przygotowany w języku XML. Interfejs GUI pozwala na pomiar czasu wykonywania wybranej funkcji natywnej. Pomiar może być otrzymany dla jednokrotnego wykonania lub wielokrotnego z taką liczbą powtórzeń jaka została określona przez użytkownika. Widok interfejsu GUI z możliwością wyboru funkcji, której czas wykonania ma zostać zmierzony a także wskazania liczby powtórzeń, został przedstawiony na rysunku 5.

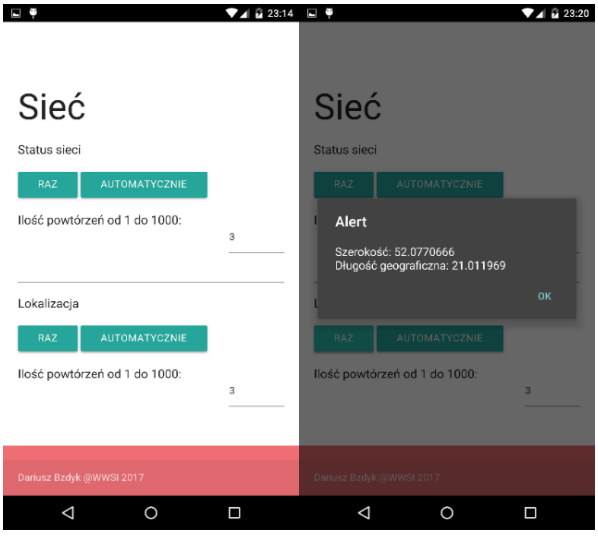


Rysunek 5. Interfejs GUI aplikacji natywnej dla systemu operacyjnego Android z przykładowym wynikiem czasu wykonania funkcji odczytującej położenie terminala

3.2. Implementacja aplikacji hybrydowej

Graficzny Interfejs użytkownika (GUI) aplikacji hybrydowej jest bardzo podobny do interfejsu GUI aplikacji natywnej (rysunek 6).

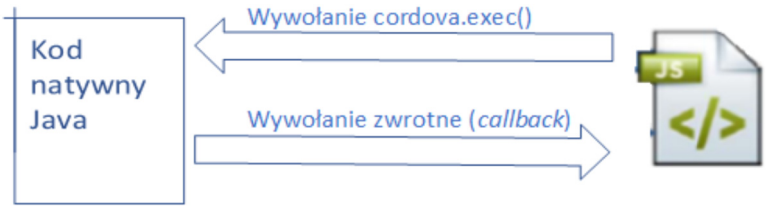




Rysunek 6. Interfejs GUI aplikacji hybrydowej dla systemu operacyjnego Android z przykładowym wynikiem czasu wykonania funkcji odczytującej położenie terminala

Aplikacja hybrydowa oferuje ten sam zakres funkcjonalny oraz opcje wykonywania pomiarów co aplikacja natywna, dzięki czemu możliwe jest bezpośrednie porównanie wydajności wywoływania funkcji natywnych zaprogramowanych dwoma różnymi sposobami.

Dostęp do funkcji natywnych z poziomu kodu w języku JavaScript zapewniony jest poprzez użycie dodatkowego oprogramowania PhoneGap (w wersji 8.0.0), które pełni funkcję mostu. Umożliwienie wywołania funkcji natywnej z poziomu języka JavaScript wymaga odpowiedniego przygotowania kodu JavaScript i powiązanego z nim kodu w Javie. Funkcjonalność tego powiązania udostępniona z poziomu języka JavaScript określana jest terminem „wtyczka" (*plugin*). Część wtyczek jest już zaimplementowanych w oprogramowaniu mostu, inne można dodać odpowiednio wytwarzając kod w językach JavaScript i Java. Budowa i ogólna zasada działania wtyczek zostały zilustrowane na rysunku 7.



Rysunek 7. Budowa i ogólna zasada działania wtyczek PhoneGap

Wtyczka posiada klasy i metody zaimplementowane w języku JavaScript, których można używać w kodzie JavaScript aplikacji. Każde odwołanie z klasy lub metody JavaScript do funkcji natywnej wymaga implementacji metody *cordova.exec*, która przyjmuje cztery argumenty formalne (rysunek 8):

- pierwszy argument – określa nazwę funkcji (wraz z listą argumentów) wywoływanej zwrotnie po udanym wykonaniu funkcji *cordova.exec*. W naszym przykładzie jest to funkcja o nazwie *function(winParam)*,
- drugi argument określa nazwę funkcji (wraz z listą argumentów) wywoływanej zwrotnie po nieudanym wykonaniu funkcji *cordova.exec*. W naszym przykładzie jest to funkcja *function(error)*. Argumentem tej funkcji jest obiekt opisujący błąd związany z nieudanym wykonaniem funkcji *cordova.exec*,
- trzeci argument (*service*) – określa usługę natywną, z której chcemy skorzystać. Usługa ta jest wskazana poprzez podanie klasy natywnej, zdefiniowanej bezpośrednio w języku Java,

– czwarty argument (*action*) – określa właściwą funkcję natywną, którą wywołujemy. Funkcja ta musi być składnikiem klasy podanej w trzecim argumencie,
– piąty argument (*arguments*) – określa listę argumentów wywołania funkcji natywnej podanej w czwartym argumencie. Argumenty są przekazywane w postaci listy.

```
cordova.exec(function(winParam) {},
    function(error) {},
    "service",
    "action",
    ["firstArgument", "secondArgument", 42, false]);
```

Rysunek 8. Składnia wywołania funkcji *cordova.exec*, służącej do komunikacji poziomu JavaScript z funkcjami natywnymi

Metoda *cordova.exec* musi być osobno zdefiniowana, poprzez definicje wszystkich jej argumentów, dla każdej platformy natywnej, dla której ma być dostępna aplikacja tworzona w modelu hybrydowym. Wywołanie funkcji *cordova.exec* z kodu JavaScript, powiązanego ze stroną *html* wyświetlaną dzięki komponentowi natywnemu *WebView*, powoduje przekazanie sterowania do metody *execute()* klasy natywnej wskazanej jako trzeci argument tego wywołania.

W metodzie *execute()* następuje rozpoznanie wartości czwartego argumentu wywołania czyli sprawdzenie, która metoda klasy natywnej powinna zostać wykonana. Klasa implementująca część natywną wtyczki musi rozszerzać klasę *CordovaPlugin*. Przykładowy kod implementujący wtyczkę został pokazany na rysunku 9 (kod w języku JavaScript) oraz na rysunku 10 (kod w języku Java).

```
NetworkConnection.prototype.getInfo=function(successCallback,errorCallback) {
 exec(successCallback,errorCallback,'NetworkStatus','getConnectionInfo', []);
};
```

Rysunek 9. Kod JavaScript implementujący wtyczkę *cordova-plugin-network-information*

```
public boolean execute(String action,JSONArray args, CallbackContext call-
   backContext) {
   if (action.equals("getConnectionInfo")) {
      this.connectionCallbackContext = callbackContext;
      NetworkInfo info = sockMan.getActiveNetworkInfo();
      String connectionType = "";
      try {
         connectionType = this.getConnectionInfo(info).get("type").toString();
      } catch (JSONException e) {
         LOG.d(LOG_TAG, e.getLocalizedMessage());
      }

      PluginResult pluginResult = new PluginResult(PluginResult.Status.OK,
      connectionType);
      pluginResult.setKeepCallback(true);
      callbackContext.sendPluginResult(pluginResult);
      return true;
   }
   return false;
}
```

Rysunek 10. Kod Java implementujący wtyczkę *cordova-plugin-network-information* – część natywna

Oprogramowanie PhoneGap zawiera wiele zaimplementowanych i gotowych do użycia wtyczek do obsługi praktycznie wszystkich funkcji natywnych. W aplikacji hybrydowej zostały wykorzystane gotowe wtyczki, aczkolwiek sposób ich implementacji ma wpływ na ostateczne wyniki czasu wykonywania funkcji natywnych.

3.3. Implementacja pomiarów

Celem wykonywania pomiarów jest zmierzenie czasu działania poszczególnych funkcji natywnych gdy są wywoływane bezpośrednio z kodu w języku Java oraz gdy są wywoływane z kodu napisanego w języku JavaScript. Mając na uwadze, że spodziewane czasy wykonywania funkcji będą rzędu dziesiątek lub nawet pojedynczych milisekund, w aplikacjach starano się zadbać, aby zmierzony czas był dokładnym czasem wykonania samej funkcji. Z tego powodu zdecydowano o zaimplementowaniu pomiarów wewnątrz kodu aplikacji, dodając odpowiednie znaczniki czasowe bezpośrednio przed wywołaniem interesujących nas funkcji i bezpośrednio po nim. Zapisanie różnicy wartości tych znaczników w pliku już po zakończeniu działania funkcji pozwala na późniejszą obróbkę statystyczną i nie wpływa na wynik samego pomiaru.

W aplikacji hybrydowej pomiar został zrealizowany z wykorzystaniem metody *performance.now()*, dostępnej w Web API. Metoda ta zwraca znacznik czasowy (*TimeStamp*) w milisekundach. Rysunek 11 zawiera fragment kodu ilustrującego realizację pomiaru czasu wykonywania funkcji natywnej.

```
function polozenia() {
   var width = 0;
   var id=setInterval(action,5000);
   var ile=document.getElementById("licznikPolozenia").value;
   function action() {
      if (width == ile) {
         clearInterval(id);
         alert('Koniec');
      } else {
         width++;
         function onSuccess(acceleration) {

         }
         function onError() {
            alert('onError!');
         }
            var startTime = performance.now();
         navigator.accelerometer.getCurrentAcceleration(onSuccess, onError);
         var delta = performance.now()-startTime;
         save(delta);
         window.CacheClear(success, error);
      }
   }
}
```

Rysunek 11. Kod realizujący pomiar czasu wykonania funkcji natywnej wywołanej w aplikacji hybrydowej

W aplikacji natywnej idea pomiaru jest taka sama, jednak ze względu na różnice w API jest ona realizowana za pomocą innej metody. Do pomiaru wykorzystany został obiekt *TimingLogger*, posiadający możliwość zmierzenia czasu potrzebnego na wywołanie metody. Wynik podawany w milisekundach jest przedstawiany z dokładnością do 1ms. Wyniki są zapisywane do pliku tekstowego w pamięci telefonu, a także są dostępne z poziomu narzędzia Android Monitor w środowisku Android Studio.

Rysunek 12 zawiera fragment kodu ilustrującego realizację pomiaru czasu wykonania funkcji natywnej w aplikacji natywnej.

```
akcelAuto.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
         TimingLogger timings = new TimingLogger("COMP2", "methodA");
         For (int i=0;i<(Integer.valueOf(akcelIlosc.getText().toString()));i++) {
            Accelerometer ac = new Accelerometer(MainActivity.this);
            timings.addSplit("Test "+i);
            double log = ac.ax;
            timings.dumpToLog();
            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            timings.reset();
         }
      }
   });
```

Rysunek 12. Kod realizujący pomiar czasu wykonania funkcji natywnej wywołanej z aplikacji natywnej

Pomiar czasów wykonywania funkcji natywnych nie jest procesem deterministycznym, lecz ma charakter losowy. W trakcie wykonywania pomiarów system operacyjny urządzenia mobilnego cały czas obsługuje pewną liczbę procesów, które muszą być wykonywane, np. procesy systemowe. To obciążenie może być zmienne w czasie i stanowić powód różnic w czasie wykonania tych samych funkcji natywnych w kolejnych wywołaniach.

Z tego powodu każdy pomiar został powtórzony kilkanaście razy, aby zebraną serię próbek poddać obróbce statystycznej, pozwalającej na uwzględnienie zmienności poprzez uśrednienie wartości poszczególnych próbek i wyznaczenie przedziałów ufności, w których zawiera się prawdziwa, szukana przez nas wartość. Chcąc mieć wysokie przekonanie, że w wyznaczonym przedziale znajduje się szukana przez nas wartość prawdziwego czasu wykonania funkcji natywnej, przyjęliśmy poziom ufności równy 95%. Wszystkie pomiary, z wyjątkiem tych wymagających dostępu do sieci, zostały wykonane w trybie samolotowym.

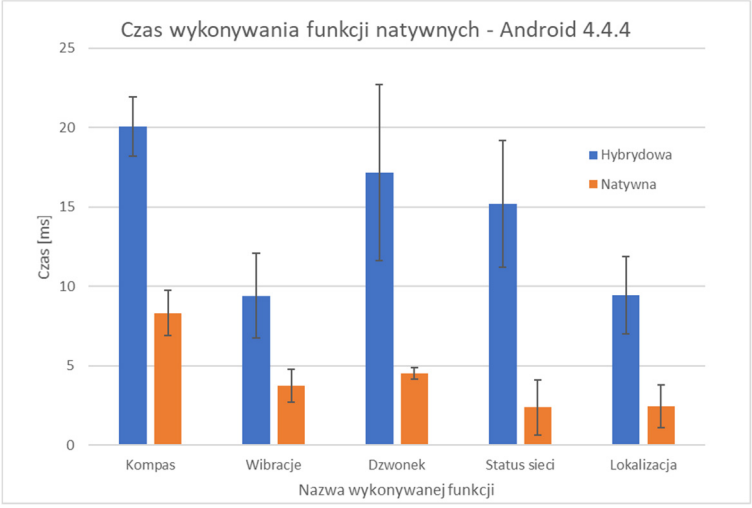
3.4. Porównanie wyników

Przypomnijmy, że naszym zadeklarowanym na początku tego opracowania (w rozdziale 2.2) celem jest porównanie wydajności wykonywania funkcji natywnych

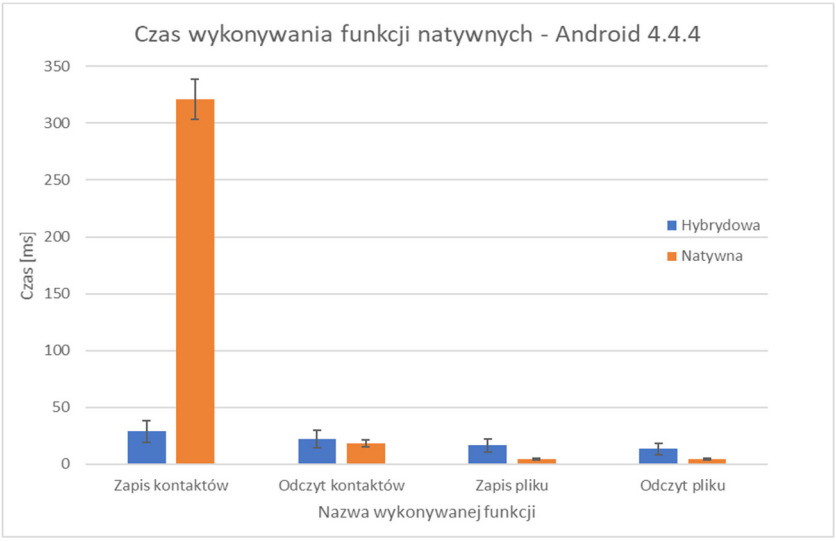
w aplikacjach pisanych w modelach natywnym i hybrydowym, również w zależności od wydajności urządzenia mobilnego oraz nowych wersji systemu operacyjnego Android. Interesuje nas nie tylko, czy istnieje różnica w czasach wykonywania funkcji natywnych, ale również to, jak ta różnica zmienia się w funkcji postępu technicznego, czyli dla nowych, bardziej wydajnych urządzeń mobilnych i nowych wersji Android OS. W ten sposób chcemy ocenić, czy wnioski przedstawione w pracy [10], a dotyczące sprzętu HTC Nexus One i Android OS w wersji 2.2, są nadal aktualne dla obecnie produkowanych i rozpowszechnianych urządzeń Nexus 5 lub Samsung S8 i systemu operacyjnego Android w wersjach 4.4.5, 6.0.1 lub 8.0.0.

Na poniższych wykresach zostały porównane czasy wykonywania funkcji natywnych wywoływanych z aplikacji stworzonych w modelach hybrydowym oraz natywnym dla trzech wersji systemu operacyjnego Android. Ze względu na rodzaj wykonywanych funkcji (dostęp do sprzętu lub operacje na danych), a także z uwagi na czytelność prezentowanych wartości (różne zakresy), wyniki pomiarów dziesięciu wybranych funkcji zostały rozdzielone w następujący sposób:

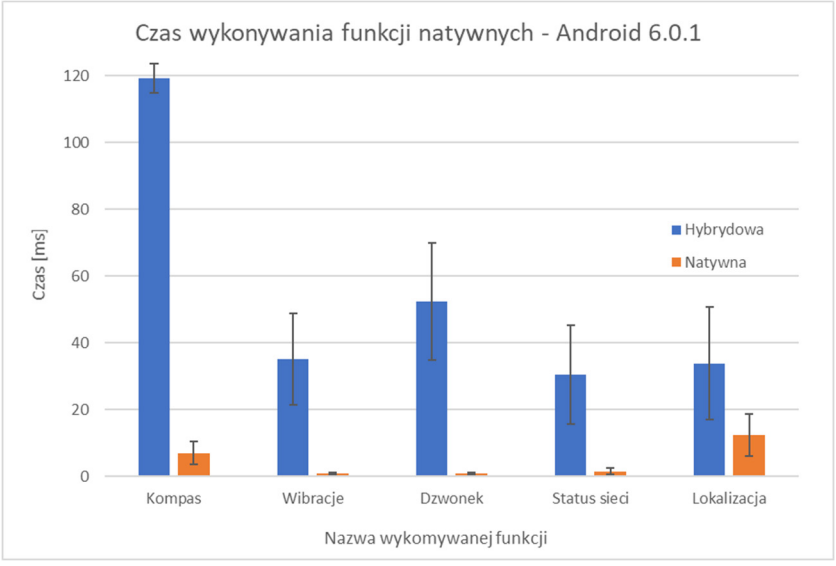
- funkcje związane ze sprzętem: kompas, wibracje, dzwonek, status sieci, lokalizacja GPS,
- funkcje związane z operacjami na danych: zapis kontaktu, odczyt kontaktu, zapis danych do pliku, odczyt danych z pliku.



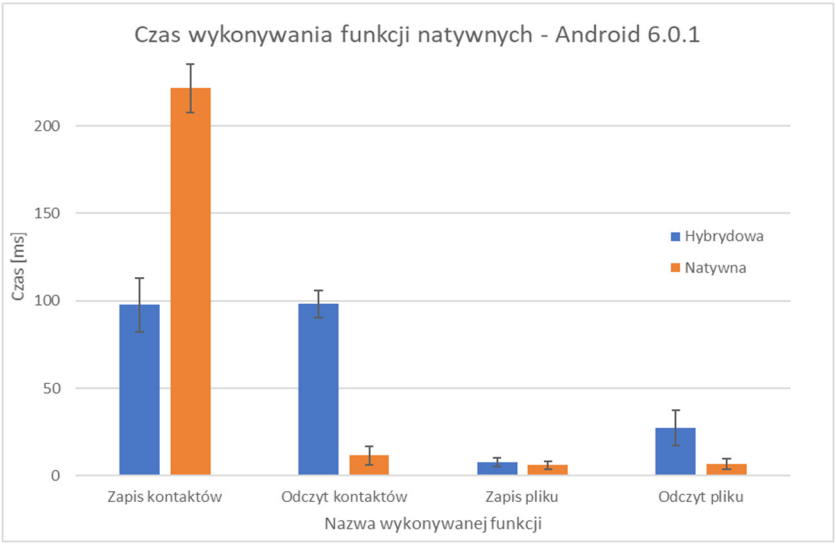
Rysunek 13. Czas wykonywania funkcji natywnych związanych ze sprzętem dla systemu operacyjnego Android w wersji 4.4.4 (Nexus 5)



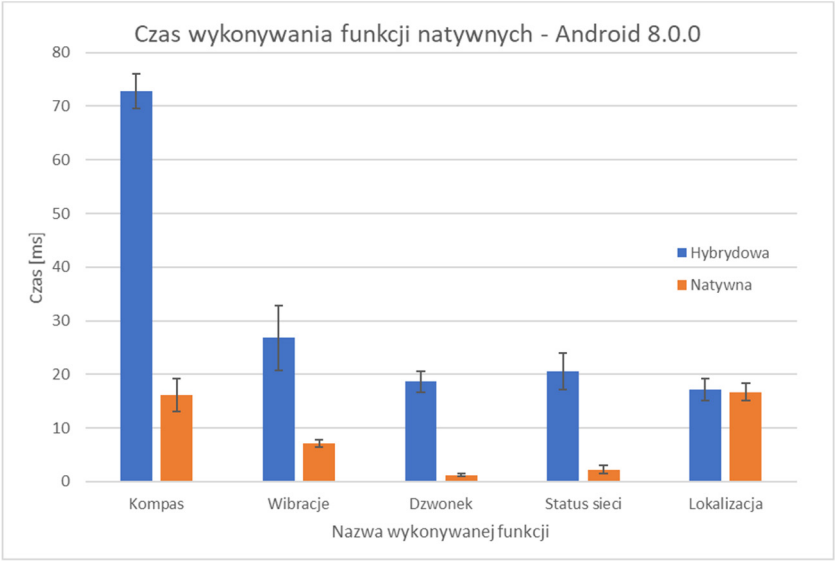
Rysunek 14. Czas wykonywania funkcji natywnych związanych z operacjami na danych dla systemu operacyjnego Android w wersji 4.4.4 (Nexus 5)



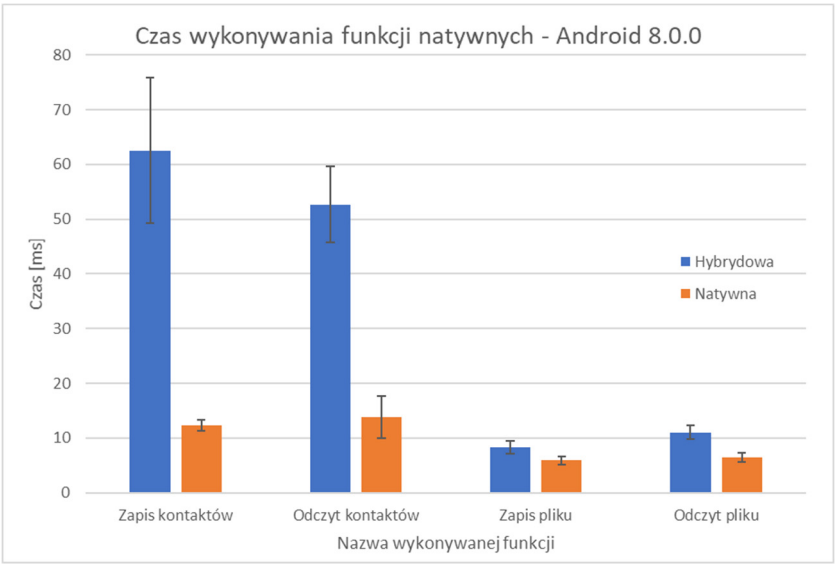
Rysunek 15. Czas wykonywania funkcji natywnych związanych ze sprzętem dla systemu operacyjnego Android w wersji 6.0.1 (Nexus 5)



Rysunek 16. Czas wykonywania funkcji natywnych związanych z operacjami na danych dla systemu operacyjnego Android w wersji 6.0.1 (Nexus 5)



Rysunek 17. Czas wykonywania funkcji natywnych związanych ze sprzętem dla systemu operacyjnego Android w wersji 8.0.0 (Samsung S8)



Rysunek 18. Czas wykonywania funkcji natywnych związanych z operacjami na danych dla systemu operacyjnego Android w wersji 8.0.0 (Samsung S8)

Przedstawione wyniki reprezentują wartości uśrednione z 14 pomiarów, z uwzględnieniem przedziałów ufności wyznaczonych dla poziomu ufności 95%.

Większość otrzymanych wyników jest zgodna z oczekiwaniami, według których czasy wykonywania funkcji natywnych w aplikacjach zbudowanych zgodnie z modelem hybrydowym powinien być dłuższy od czasu wykonywania tych samych funkcji w aplikacjach natywnych. Statystycznie dłuższe czasy wykonywania funkcji natywnych, na poziomie ufności równym 95%, otrzymano dla funkcji związanych ze sprzętem, wykonywanych w systemie operacyjnym Android w wersji 4.4.4 (rysunek 13), wersji 6.0.1 (rysunek 15) oraz wersji 8.0.0 (rysunek 17), w obu przypadkach jednak z wyjątkiem funkcji odczytu lokalizacji z czujnika GPS.

W podobny sposób można podsumować wyniki dotyczące zapisu danych w plikach i odczytu danych z nich (rysunek 14 i 18), z wyjątkiem wyników dla systemu operacyjnego Android w wersji 6.0.1 (rysunek 16), gdzie czasy zapisu danych w pliku nie różnią się w sposób statystycznie istotny.

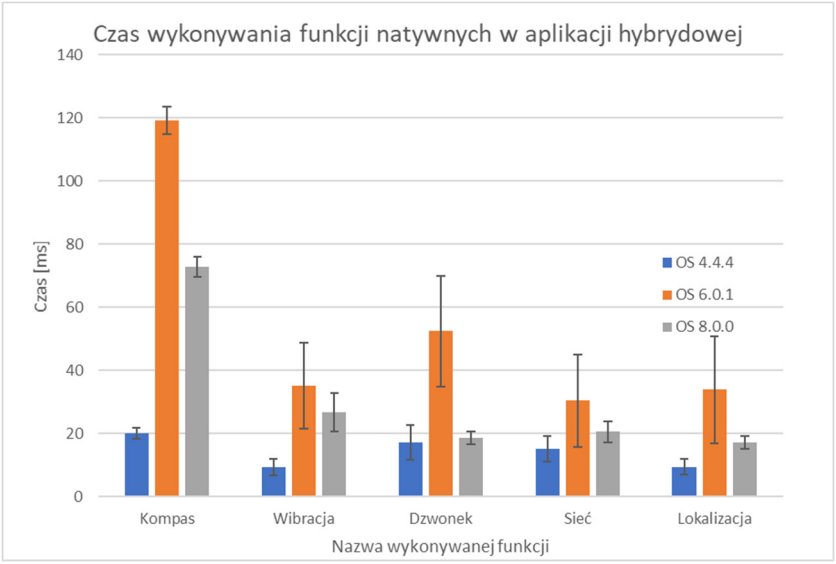
Natomiast zaskoczeniem są wyniki dla funkcji związanych z zapisywaniem kontaktów. W tym przypadku czasy wykonywania funkcji natywnej dla systemu operacyjnego Android w wersjach 4.4.4 i 6.0.1 okazały się dłuższe dla aplikacji natywnej (rysunki 14 i 16) . Zapis kontaktów jest operacją specyficzną w tym sensie, że do jej obsługi wykorzystywany jest w aplikacji natywnej mechanizm dostawcy treści (*ContentProvider*). Natomiast zaskakujące jest to, że aplikacja hybrydowa używająca wtyczki PhoneGap, która również wykorzystuje ten sam mechanizm dostępu do listy kontaktów, działa szybciej niż aplikacja natywna. Wiedza o konstrukcji wewnętrznej wtyczki do obsługi kontaktów wskazuje, że czasy wykonywania tej funkcji w aplikacji hybrydowej powinny być nie mniejsze, niż czasy wykonywania analogicznych operacji w aplikacji natywnej.

Jedynie w przypadku systemu operacyjnego Android w wersji 8.0.0 pracującego na telefonie Samsung S8, wszystkie czasy związane z wykonywania operacji na danych (kontaktach lub plikach) były statystycznie istotnie krótsze dla aplikacji natywnej w porównaniu z aplikacją hybrydową. W sensie bezwzględnym uzyskane czasy przyjmowały wartości od kilku do nawet kilkuset milisekund. Krótsze czasy obserwowano dla funkcji związanych ze sprzętem, natomiast najdłuższe dla funkcji związanych z zapisem kontaktów, niezależnie od typu aplikacji.

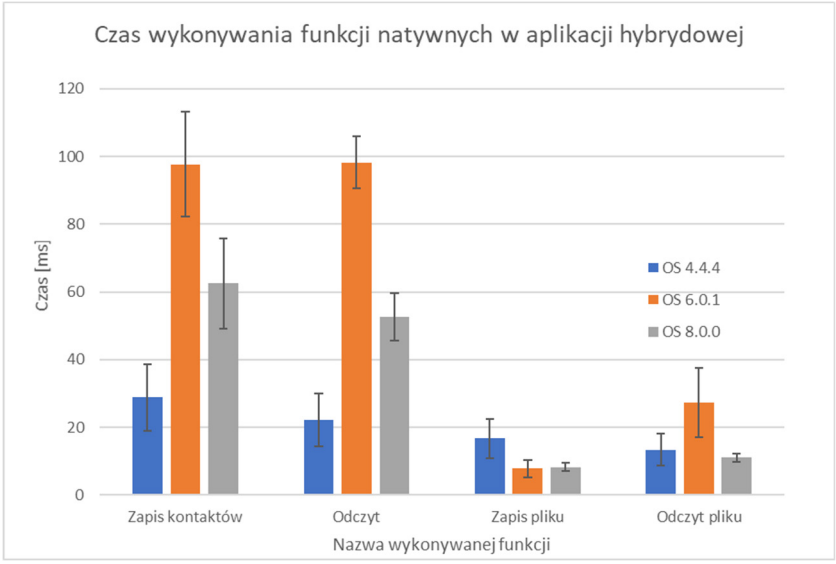
Porównanie wykonane w funkcji postępu technicznego tj. dla kolejnych wersji systemu operacyjnego Android i nowszych platform sprzętowych (Nexus 5 i Samsung S8) nie pozwoliło na udowodnienie istotnego statystycznie wpływu większej wydajności (procesor Octa-core 4x2,3 GHz w telefonie Samsung kontra procesor Quad-core 2,3 GHz w telefonie Nexus 5) na uzyskane wyniki.

Wydaje się, że wydajność platformy sprzętowej nie przesądza o wynikach dlatego, że istotny wpływ ma również obciążenie wynikające z liczby procesów działających w systemie operacyjnym, co może być silnie zależne od jego wersji.

Taką konkluzję może nasunąć analiza wykresów z rysunków 19 i 20, na których widać wydłużenie czasu wykonywania funkcji dla systemu operacyjnego Android w wersji 6.0.1 w porównaniu z wersją 4.4.4, a z drugiej strony skrócenie tych czasów dla systemu operacyjnego Android w wersji 8.0.0, działającego na urządzeniu Samsung S8, nie więcej jednak niż do wartości podobnych, jak dla wersji 4.4.4 działającej na telefonie Nexus 5.



Rysunek 19. Czas wykonywania funkcji natywnych związanych ze sprzętem dla różnych wersji systemu operacyjnego Android



Rysunek 20. Czas wykonywania funkcji natywnych związanych z operacjami na danych dla różnych wersji systemu operacyjnego Android

4. Wnioski

W większości przeprowadzonych testów otrzymano wyniki zgodne, w sensie ogólnym, z doniesieniami z wcześniejszych opracowań, np. [10], tzn. dłuższe czasy wykonywania funkcji natywnych inicjowanych z aplikacji hybrydowych, niż z aplikacji natywnych. Wydaje się to w pełni uzasadnione architekturą aplikacji hybrydowych, której zarys został przedstawiony w rozdziale 2.1.

Zaobserwowano jednak przypadki, w których otrzymane wyniki są przeciwne do spodziewanych, tzn. czas wykonania funkcji natywnej zapisu kontaktów wywołanej z aplikacji hybrydowej był krótszy, na poziomie istotnym statystycznie, niż czas wykonania tej samej funkcji wywołanej z aplikacji natywnej. Podobne, „zaskakujące" wyniki otrzymali autorzy pracy [10], aczkolwiek w odniesieniu do innej funkcji natywnej – generowanie dźwięku powiadomienia.

Niestety w tym opracowaniu, podobnie jak w [10], nie przedstawiono logicznego wyjaśnienia przyczyn niespodziewanych wyników. Ustalenie prawdziwych przyczyn wymaga zaimplementowania dodatkowych pomiarów cząstkowych na poziomie poszczególnych składowych biorących udział w realizacji danej funkcji wywoływanej z poziomu PhoneGap API.

Również porównanie czasów wykonywania funkcji natywnych w różnych wersjach systemu operacyjnego Android nie pozwoliło na wyciągniecie konkretnych wniosków. Otrzymane w tym przypadku wyniki nie są bowiem jednoznaczne. Dla aplikacji hybrydowej działającej w systemie operacyjnym Android w wersji 6.0.1 otrzymano na tym samym telefonie dłuższe czasy niż dla wersji 4.4.4, która jest starsza o dwie generacje. Jednocześnie dla wersji Android 8.0.0 otrzymano czasy krótsze niż dla wersji 6.0.1 i zbliżone do czasów dla wersji 4.4.4, aczkolwiek na nowszym sprzęcie, tj. telefonie Samsung S8.

Podobnie jak wyjaśnienie różnic w wynikach dla aplikacji hybrydowych i natywnych wykonujących te same funkcje natywne, również wyjaśnienie różnic w wynikach dla różnych wersji systemów operacyjnych Android wymaga implementacji dodatkowych pomiarów cząstkowych, które pokażą wpływ poszczególnych składowych biorących udział w realizacji danej funkcji wywoływanej z poziomu PhoneGap API.

Literatura

[1] Gartner, *Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016*, http://www.gartner.com/newsroom/id/3609817 [20.10.2018]

[2] S. Komatineni, D. MacLean, S. Hashimi, *Android 3: tworzenie aplikacji*, Helion, 2012.

[3] S. Kochan, *Objective-C. Praktyczny podręcznik tworzenia aplikacji na systemy iOS i Mac OS X! Vademecum profesjonalisty*, Helion, 2012.

[4] M. Mathias, J. Gallagher, *Programowanie w języku Swift. Big Nerd Ranch Guide*, Helion, 2017.

[5] Apache Cordova, https://cordova.apache.org.

[6] Adobe PhoneGap framework, https://phonegap.com.

[7] E. Pimpler, *Anatomy of a Hybrid Mobile GIS Application*, Geospatial Training Services, March 5 2012, http://geospatialtraining.com/anatomy-of-a-hybrid-mobile-gis-application/.

[8] V. Ahti, S. Hyrynsalmi, O. Nevalainen, *An Evaluation Framework for Cross-Platform Mobile App Development Tools: A case analysis of Adobe PhoneGap framework*, CompSysTech ‘16 Proceedings of the 17th International Conference on Computer Systems and Technologies 2016, DOI: 10.1145/2983468.2983484.

[9] F. Rösler, A. Nitze, A. Schmietendorf, *Towards a Mobile Application Performance Benchmark*, The Ninth International Conference on Internet and Web Applications and Services, July 2014.

[10] L. Corral, A. Sillitti, G. Succi, *Mobile Multiplatform Development: An Experiment for Performance Analysis*, “Procedia Computer Science” Vol. 10, 2012, https://doi.org/10.1016/j.procs.2012.06.094.

The Comparison of the Native Function Execution Times for Mobile Application Implemented Using Native and Hybrid Approaches

Abstract

This paper presents the performance evaluation of the mobile native and hybrid applications. The comparison of application performance was carried out assuming a native function execution time (e.g. an access to the hardware, an access to the network, writing or reading files or contacts) as a main criteria. The measurements were conducted by preparing two functionally identical applications for Android OS, one written in Java language (native methodology) the other written in JavaScript and HTML languages with the aid of PhoneGap bridge (hybrid methodology), that were later used to call selected native functions and measure their execution time. The evaluation was performed for three versions of Android OS in order to have a broader perspective on the analysed issue.

Keywords – Hybrid applications, native applications, Android OS, PhoneGap