Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Automatyki i Informatyki Stosowanej

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Systemy Informacyjno-Decyzyjne

Implementacja systemu wnioskującegoo w systemie operacyjnym Android

Grzegorz Kuduk
Numer albumu 277320

promotor
dr inż. Mariusz Kamola

WARSZAWA 2020

Streszczenie

Tematem pracy jest „Implementacja systemu wnioskującego w systemie operacyjnym Android”. Celem było stworzenie aplikacji Android implementującej system wnioskujący. Początkowym założeniem była implementacja na podstawie istniejącej aplikacji iOS, lecz ostatecznie powstała ona od podstaw. Szczególnemu uwzględnieniu podległa interoperacyjność z jej odpowiednikiem na iOS, którego autorem jest inż. Jędrzej Blak. We współpracy z nim opracowano standardy wykorzystane w tworzeniu systemów wnioskujących obu prac.

Aby system wnioskujący miał na czym wnioskować, należy dostarczyć do niego dane, które w tym przypadku pochodzą z urządzeń (sensorów) połączonych z telefonem Android poprzez Bluetooth Low Energy oraz z innych telefonów przesyłających już wywnioskowane po swojej stronie dane.

Zastosowano silnik wnioskujący HyLAR (Hybrid Location-Agnostic incremental Reasoner). Jest to framework open source napisany w języku JavaScript i służy do tworzenia aplikacji sieci semantycznych. Opiera się on na danych RDF (Resource Description Framework), ontologiach OWL (Web Ontology Language) i zawartych w nich „trójkach”.

Pierwszą kluczową częścią systemu jest aplikacja Android. Została ona napisana w języku Kotlin i odpowiada za zbieranie danych z sensorów i sieci Bluetooth Low Energy, przesyłanie ich do lokalnego serwera wnioskującego oraz odbieranie odpowiedzi z wywnioskowanymi faktami, na podstawie których wyświetla powiadomienia i wysyła informacje z powrotem do lokalnej sieci urządzeń Android i iOS.

Drugą kluczową częścią systemu jest serwer wnioskujący. Jego zadaniem jest komunikacja z aplikacją poprzez protokół HTTP oraz wykonywanie odpowiednich operacji silnika wnioskującego HyLAR na modelu danych.

Słowa kluczowe:

- System wnioskujący
- HyLAR
- RDF
- OWL
- Bluetooth Low Energy
- Android
- Kotlin

Abstract

The topic of this thesis is „Implementation of a reasoning system on the Android operating system". It implies that its goal is to create an Android application that implements a reasoning system. The initial assumption was that it would be an implementation basing on an existing iOS application, but ultimately it was created from scratch. Taken into special consideration was interoperability with its iOS counterpart made by Eng. Jędrzej Blak.

For the reasoning system to have something to reason on, it needs to have data delivered to it. Data which in this case comes from devices (sensors) connected to the Android phone via Bluetooth Low Energy and from other phones sending data that has been already reasoned on their side.

The reasoning engine that has been used is HyLAR (Hybrid Location-Agnostic incremental Reasoner). It's an open source framework written in JavaScript and it's used for making semantic web applications. It relies on RDF (Resource Description Framework) data, OWL (Web Ontology Language) ontologies and „triples", which are included in both.

The first key part of the system is the Android application. It was written in Kotlin and is responsible for gathering data from sensors and the Bluetooth Low Energy network, sending it to the local reasoning server and receiving responses with reasoned facts, based on which it shows notifications and sends information back into the local network of Android and iOS devices.

The second key part of the reasoning system is the reasoning server. Its tasks are to communicate with the application via the HTTP protocol and executing the right HyLAR reasoning engine operations on the data model.

Thesis keywords:

- Reasoning system
- HyLAR
- RDF
- OWL
- Bluetooth Low Energy
- Android
- Kotlin

załącznik do zarządzenia nr 28/2016 r.
Rektora PW

Politechnika Warszawska

„załącznik nr 3 do zarządzenia nr 24/2016 Rektora PW

Warszawa, 30.01.2020
miejscowość i data

Grzegorz Kudelski
imię i nazwisko studenta
277320
numer albumu
Informatyka
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

Grzegorz Kudelski
czytelny podpis studenta"

Spis treści

Wstęp

Na początek należy przybliżyć tematykę oraz cele tematu „Implementacja systemu wnioskującego w systemie operacyjnym Android”. Ogólnie mówiąc, system wnioskujący to system, który emuluje proces podejmowania decyzji. Najpierw powinno się określić na podstawie jakich danych mają być one podejmowane oraz jak je wykorzystać. Platformą docelową były urządzenia mobilne, a jak wiadomo dzisiejsze smartfony posiadają wiele wbudowanych sensorów, więc mają również dużą ilość danych źródłowych, które nadają się do wnioskowania. Dodatkowo powszechnie stosowane są opaski mierzące aktywność i inne urządzenia przenośnie z sensorami badającymi stan użytkownika lub otoczenia, które łączą się ze smartfonem. Mając takie źródła informacji, system wnioskujący może dostawać cyfrowe dane od świata zewnętrznego, a mając te dane jest w stanie przeprowadzić proces wnioskowania. Celem wnioskowania jest powiadomienie o wywnioskowanych faktach, na przykład w formie ostrzeżenia lub przydatnej informacji, lecz może ono również inicjować bardziej złożone operacje.

W dzisiejszym świecie prawie każdy nosi w kieszeni smartfon, który sam w sobie jest w stanie „opisać” elementy swojego otoczenia, ale nie jest w stanie ich połączyć w spójną całość. Zadaniem systemu wnioskującego jest stworzenie właśnie tej spójnej całości, aby powstały nowe fakty, które można wykorzystać w różnych celach. Jednym z celów jest wywołanie pewnych zachowań u ludzi. Przykładowo starsza osoba z czujnikiem bicia serca oraz smartfonem w kieszeni mogłaby dostać zawału serca i upaść lub mogłaby zostać zatrzymana akcja jej serca. W takiej sytuacji odpowiedni system wnioskujący byłby w stanie wykryć upadek / zatrzymanie akcji serca / obie sytuacje naraz na podstawie gwałtownego wzrostu odczytywanych z akcelerometru wartości / spadku pulsu poniżej normy i powiadomić o nim osoby znajdujące się w pobliżu, mając nadzieję, że wywoła u nich chęć sprawdzenia czy dana osoba rzeczywiście potrzebuje pomocy.

Stworzony w ramach tej pracy system to dowód koncepcji, że już w niedalekiej przyszłości systemy wnioskujące będą w stanie wspierać ludzi na porządku dziennym. Porusza on także kwestię przyszłościowego edge computing, ponieważ już dwa telefony znajdujące się w sieci tworzonej przez połączenia pomiędzy nimi stosują paradygmat rozproszonego przetwarzania danych. Warto również wspomnieć, że pewne ustalenia dotyczące rozwiązania powstały we współpracy z inż. Jędrzejem Blakiem – autorem odpowiednika opisywanego systemu wnioskującego na systemie operacyjnym iOS – w celu zapewnienia interoperacyjności obu aplikacji.

Zanim przedstawiona zostanie implementacja systemu wnioskującego, należy przedstawić i wytłumaczyć główne koncepty i pojęcia z nią związane. Rozdział 1. ma na celu przybliżenie tych zagadnień. W Rozdziale 2. wyjaśniono kwestie potrzebne do zrozumienia zastosowanego w rozwiązaniu Bluetooth Low Energy jako środku komunikacji bezprzewodowej telefonu z innymi urządzeniami wspierającymi BLE. Natomiast w Rozdziale 3. pracy przedstawiono implementację systemu wnioskującego. Początkowo jej celem była aplikacja, która miała powstać na podstawie istniejącej aplikacji iOS, jednak ostatecznie powstała aplikacja została stworzona od podstaw. W dalszej części tego rozdziału opisano również test systemu wnioskującego za pomocą symulacji pomiaru wysokiej temperatury powietrza.

Pierwszą z dwóch najbardziej problematycznych kwestii pisania tej pracy była implementacja łączności Bluetooth Low Energy. Wiele razy pewna rzecz nie działała przez błędy wewnętrzne, poza kontrolą z poziomu kodu, a po krótkiej chwili / restarcie aplikacji / restarcie telefonu zaczynała działać. Drugą taką kwestią była mała dostępność silników wnioskujących działających natywnie na systemie operacyjnym Android, a te działające zazwyczaj były przestarzałe, przez co stworzono niestandardowe rozwiązanie.

Do testowania połączeń z urządzeniami Bluetooth Low Energy wykorzystano urządzenie Texas Instruments SimpleLink™ SensorTag CC2650STK. Wersje systemów Android na testowanych telefonach: 7.0 i 9.0. Do niektórych testów, między innymi funkcjonalności serwera wnioskującego, skorzystano z emulatora Android 10.0 w środowisku Android Studio oraz node.js w wersji v10.15.1.

Szczególne podziękowania należą się promotorowi dr inż. Mariuszowi Kamoli za opiekę nad pracą.

Rozdział I.
Wnioskowanie

1.1. Systemy wnioskujące

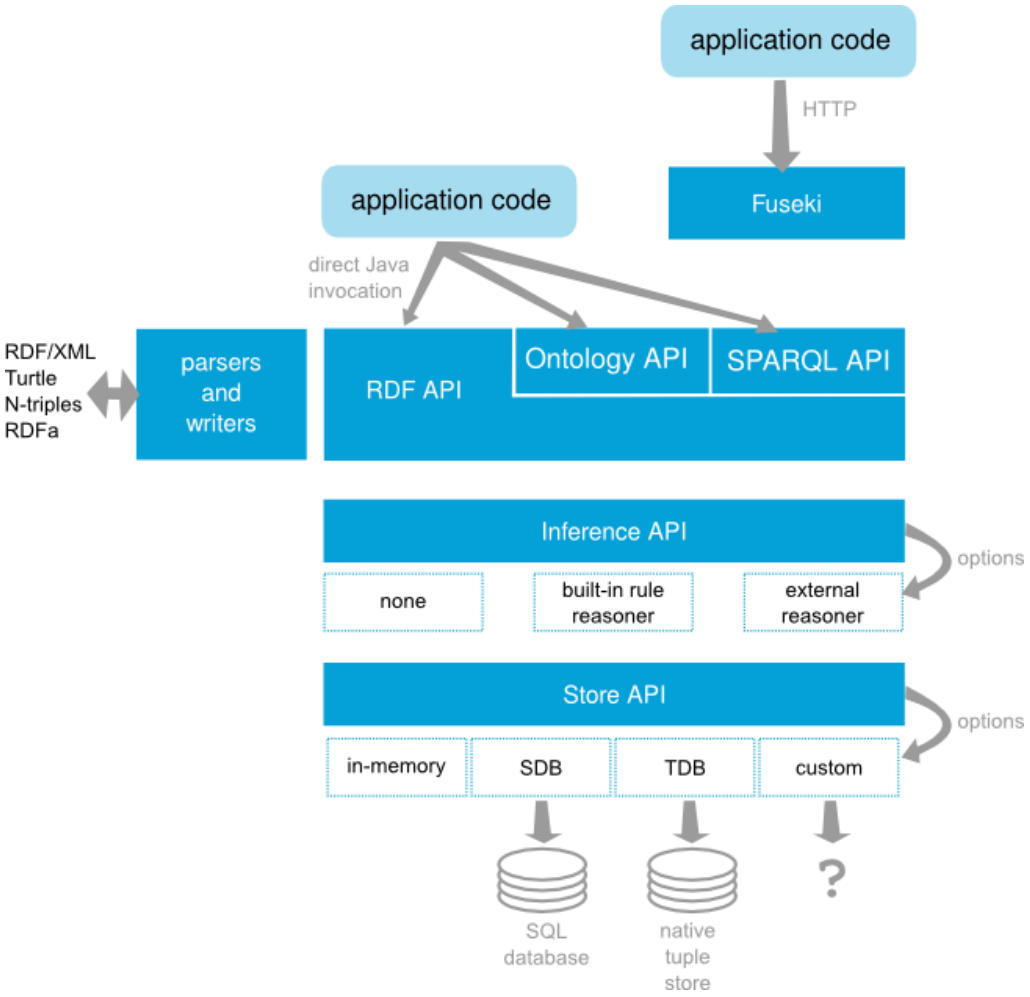
System wnioskujący, zwany również systemem ekspertowym, to system, który emuluje proces podejmowania decyzji. Rozwiązuje on złożone problemy na podstawie analizy baz wiedzy i składa się z dwóch głównych części:

- silnika wnioskującego – programu szukającego odpowiedzi na zadane pytania
- bazy danych – zawierającej fakty, które są analizowane przez silnik wnioskujący w celu znalezienia odpowiedzi.

W ramach wprowadzenia do tematu systemów wnioskujących przeczytano artykuł naukowy „Semantic Reasoning for Context-Aware Internet of Things Applications”[1]. Jest on świetną lekturą, która przybliża rozwiązania stosowane w IoT i dzięki której można zapoznać się z głównymi konceptami systemów wnioskujących. Kolejnym artykułem naukowym wartym wspomnienia, który naprowadził na potencjalne, jednak już przestarzałe rozwiązania na system operacyjny Android, jest publikacja „Android Went Semantic: Time for Evaluation”[2].

Początkowe prace postępowały z założeniem użycia **Apache Jena** lub jej adaptacji na system Android **Jena Android**. Apache Jena to framework open source zawierający różne API współdziałające ze sobą w celu przetwarzania danych RDF. Całość została napisana w języku Java, ale aby framework był używalny na systemie Android, wymagane jest wykorzystanie pośrednich rozwiązań. Problemy wynikają z różnic pomiędzy Java Virtual Machine i Dalvik Virtual Machine, z którego korzysta Android.

Apache Jena oferuje wiele programistycznych dogodności, takich jak bezpośredni zapis bazy RDF do trzymanych w pamięci obiektów Java oraz automatyczne wnioskowanie po wykonaniu operacji dodania / usunięcia zasobów z modelu danych. Na Rysunku 1. pokazano architekturę framework’a.



Rysunek 1: Architektura Apache Jena.
Źródło: jena.apache.org

Niestety, ze względu na chęć zachowania jak najlepszej interoperacyjności z odpowiednikiem aplikacji działającej na systemie iOS, nastąpiła potrzeba odrzucenia rozwiązania wykorzystującego framework Apache Jena i zastosowania bardziej niestandardowej metody – połączenia aplikacji z silnikiem HyLAR.

Rozwiązanie wykorzystujące silnik wnioskujący HyLAR jest niestandardowe ze względu na język, w którym został on napisany – JavaScript. JavaScript nie jest językiem natywnym systemu Android i nie może zostać skompilowany na ten system w napisanej aplikacji, co jednak nie wyklucza możliwości jego integracji z aplikacją Android. Sposób implementacji silnika wnioskującego opisano w Rozdziale 3.

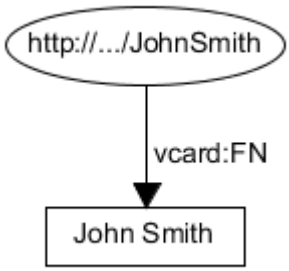
1.1.1. RDF i OWL

Resource Description Framework (RDF) to zestaw specyfikacji World Wide Web Consortium (W3C) oryginalnie zaprojektowany jako model metadanych. Z upływem czasu RDF stał się ogólną metodą konceptualnego opisu i modelowania zasobów, korzystając z różnych formatów serializacji danych, składni i notacji. Bardzo powszechnym zastosowaniem RDF są właśnie systemy wnioskujące.

Celem wprowadzenia RDF było utworzenie ogólnoświatowego standardu zapisu metadanych, który nie pozwalałby na dużą dowolność. Umożliwiłoby to automatyczne, maszynowe

przetwarzanie abstrakcyjnych opisów zasobów i tym samym wyszukiwanie lub śledzenie informacji na dany temat.

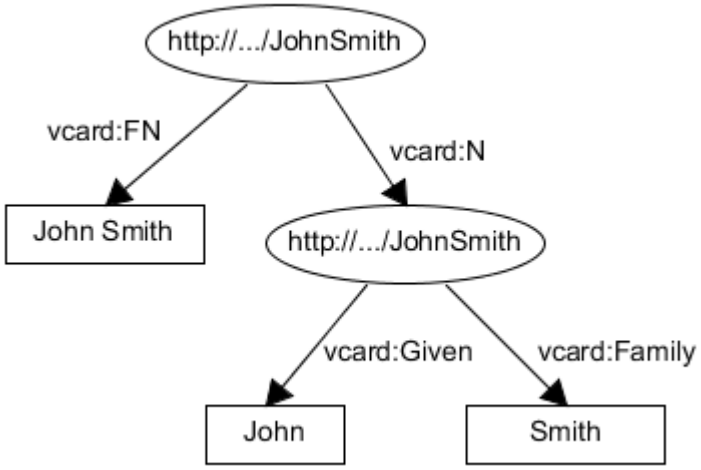
Bazy faktów RDF to nazywane, kierunkowe grafy. Ich podstawowym budulcem jest tzw. **trójka** (ang. **triple**). Trójka to wyrażenie składające się z trzech elementów: podmiotu, predykatu (własności) i obiektu (wartości) – powszechnie znanych w języku angielskim jako: *subject*, *predicate*, *object*.



Rysunek 2: Przykład trójki.
Źródło: jena.apache.org

Zaczynając od prostego przykładu pojedynczej trójki widocznej na Rysunku 2., pokazany wewnątrz eklipsy zasób – John Smith – jest identyfikowany przez ujednolicony identyfikator zasobów **URI** (ang. Uniform Resource Identifier) http://.../JohnSmith. URI pozwala na wymianę trójek pomiędzy bazami faktów bez ich ponownej konfiguracji. Zasoby mają własności, w tym przypadku jest to pełne imię danej osoby „John Smith". Własność jest reprezentowana przez strzałkę, oznaczoną nazwą własności, od podmiotu do obiektu. Nazwy własności również mają postać URI, ale w tym przypadku zostało ono skrócone przez lokalny dla bazy faktów prefiks *vcard* z danej przestrzeni nazw. Obiektem może być inny zasób, pusty zasób lub wartość (w tym przypadku wartość tekstowa *String*).

Przykład z dodatkowym podziałem imienia zasobu na imię i nazwisko został przedstawiony na Rysunku 3.



Rysunek 3: Przykład bazy faktów.
Źródło: jena.apache.org

Ten sam przykład zapisany w bazie RDF ze składnią RDF/XML wygląda następująco:

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
>
  <rdf:Description rdf:about='http://somewhere/JohnSmith'>
    <vcard:FN>John Smith</vcard:FN>
    <vcard:N rdf:nodeID="A0"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A0">
    <vcard:Given>John</vcard:Given>
    <vcard:Family>Smith</vcard:Family>
  </rdf:Description>
</rdf:RDF>
```

Dokument zaczyna się od deklaracji RDF oraz definicji przestrzeni nazw *rdf* oraz *vcard*. Następnie opisany jest zasób http://somewhere/JohnSmith poprzez atrybuty *vcard:FN* i *vcard:N*, z których pierwszy ma wartość tekstową, a drugi sam w sobie jest zasobem bez wartości. Pusty zasób jest opisany przez atrybuty *vcard:Given* i *vcard:Family*, które mają odpowiednie wartości tekstowe.

Formaty serializacji danych takie jak JSON-LD, N3 i Turtle mają na celu uproszczenie zapisywania danych RDF, jednocześnie dostarczając dodatkową funkcjonalność. Na przykład trójka z Rysunku 2. zapisana w składni Turtle może wyglądać następująco:

```
@base <http://somewhere/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

<#JohnSmith>
   vcard:FN "John Smith" ;
   vcard:N _:name .

_:name vcard:Given "John" .
_:name vcard:Family "Smith" .
```

Jak widać, dane zapisane w tej składni są bardziej czytelne i w dodatku można ustawić na przykład prefiks podstawowy za pomocą *@base*, który zostanie zastosowany automatycznie, jeśli dany podmiot nie korzysta z innego.

W tej części zaprezentowane zostały jedynie bardzo proste przykłady w celu pokazania podstaw budowania baz danych RDF. W przypadku rzeczywistego, trochę bardziej rozbudowanego, systemu wnioskującego, liczba trójek dochodzi do dziesiątek tysięcy.

Web Ontology Language (OWL) to rozszerzenie RDF. Opierając się na stronie W3C: OWL to język Semantic Web zaprojektowany do reprezentacji bogatej i złożonej wiedzy o rzeczach, grupach rzeczy oraz relacji pomiędzy nimi. OWL to oparty na logice język obliczeniowy, umożliwiający wykorzystanie reprezentowanej w nim wiedzy przez programy komputerowe, np. w celu weryfikacji spójności tej wiedzy lub ujawnienia wiedzy domniemanej[3].

Analogicznie do RDF również jest to język ze składnią opartą na trójkach, a semantyką na logice opisowej. Służy do zapisywania danych w postaci **ontologii** – opisów relacji semantycznych pomiędzy zasobami. OWL pozwala na wskazanie powiązań pomiędzy zasobami nie znajdujących się w tych samych trójkach, czyli na przykład może pomóc

programowi w zrozumieniu, że jeśli „A jest ojcem B” oraz „B jest ojcem C”, to „A jest dziadkiem C”.

1.1.2. Reguły wnioskowania

Reguły wnioskowania to wytyczne, według których silnik przeprowadza proces wnioskowania.

Składnia reguł może różnić się pomiędzy silnikami wnioskowania ze względu na różne możliwości tych silników oraz różne zastosowane algorytmy parsowania tych reguł. Wspomniana na początku tego rozdziału Apache Jena wspiera zarówno wnioskowanie w przód, jak również wnioskowanie wstecz i wnioskowanie hybrydowe (w przód i wstecz połączone).

Wnioskowanie w przód polega na tworzeniu nowych faktów na podstawie istniejącej bazy faktów, podczas gdy wnioskowanie wstecz, jak można się domyślić, działa odwrotnie – algorytm dostaje cel, który powinien zostać przez niego udowodniony jako prawdziwy i idzie od celu do znanych, znajdujących się w bazie, faktów. Zależnie od rozmiaru bazy faktów, złożoności jej modelu, reguł wnioskowania i zastosowanego algorytmu – jeden tryb wnioskowania może okazać się bardziej skuteczny i szybszy od drugiego.

Przykładowa prosta reguła napisana składnią wykorzystywaną przez Apache Jena wygląda następująco:

```
[same1: (?A ?P ?B), (?A ?P ?C) -> (?B owl:sameAs ?C) ]
```

Dla Jeny znane są ogólne prefiksy, takie jak *rdf* i *owl*, również w regułach. Dana reguła zawiera się w kwadratowych nawiasach i ma nazwę *same1*. Następnie w nawiasach znajdują się trójki oddzielone przecinkiem i jedna trójka po strzałce w prawo ->, która mówi silnikowi, że do danej reguły powinno zostać zastosowane wnioskowanie w przód. Trójki po lewej stronie strzałki to przesłanki, które jeśli zostaną spełnione, oznaczą że trójka po prawej stronie strzałki jest prawdziwa i może zostać dopisana do bazy faktów. Całą regułę można wyrazić wyrażeniem:

„Jeśli spełnione są warunki (?A ?P ?B) i (?A ?P ?C) to znaczy, że (?B owl:sameAs ?C).”

Nazwy za znakami zapytania jak *?A* oznaczają zmienne, do których silnik dopasowuje kolejne fakty. Predykat *owl:sameAs* oznacza, że podmiot i obiekt są takie same. Znając zasady zapisywania reguł, reguła *same1* może zostać przetłumaczona na zdanie:

„Jeśli istnieje taka trójka, gdzie A jest podmiotem, P predykatem i B obiektem oraz taka trójka, gdzie A jest podmiotem, P predykatem i C obiektem, to B i C są jednakowe.”

1.1.3. SPARQL

SPARQL to język zapytań (ang. query language) i protokół służący do wykonywania operacji na bazach RDF zaprojektowany przez W3C RDF Data Access Working Group. Służy on do formułowania zapytań skierowanych do bazy RDF, które z kolei mogą posłużyć zarówno do wyszukiwania i odczytywania zasobów, jak i do wykonywania operacji na bazie faktów[4]. SPARQL jest dla baz RDF tym, czym SQL jest dla baz MySQL, Oracle, itp.

Składnia zapytań jest podobna do zaprezentowanej w poprzedniej części składni reguł – tak samo oznaczane są zmienne. Niektórymi z podstawowych operacji, do wykonania których zdolny jest SPARQL, są operacje:

- SELECT – zwraca zasoby spełniające podane warunki
- INSERT DATA – wstawia podane trójki do bazy
- DELETE DATA – usuwa podane trójki z bazy, jeśli je w niej znajdzie
- DELETE / INSERT – odpowiednio usuwa / wstawia trójki z / do bazy bazując na podanym w klauzuli WHERE wzorze
- PREFIX – możliwość zdefiniowania prefiksu.

Podstawowym zapytaniem, które zwróci wszystkie trójki danej bazy RDF jest prosty SELECT:

```
SELECT * { ?s ?p ?o }
```

Opierając się na przykładowej bazie z Rysunku 3., zapytanie SPARQL:

```
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?o WHERE { ?s vcard:FN ?o }
```

zwróci pojedynczą wartość:

```
„John Smith”
```

Zaprezentowana w tej części funkcjonalność SPARQL stanowi małą część dużego potencjału tego języka, a pomimo to umożliwia już wykonanie najważniejszych operacji na bazie RDF. W przypadku prezentowanego w tej pracy systemu wnioskującego znajomość tych kilku operacji w zupełności wystarczy do zrozumienia jego implementacji.

1.2. HyLAR

HyLAR (Hybrid Location-Agnostic incremental Reasoner) to silnik wnioskujący napisany w języku JavaScript. Korzysta z bibliotek rdfstore-js, sparqljs i rdf-ext, które służą do wykonywania operacji na bazach RDF. Może on zostać wykorzystany lokalnie jako moduł npm (node package manager) lub globalnie jako serwer. Dzięki wykorzystaniu rdfstore-js HyLAR wspiera serializacje JSON-LD, N3 i Turtle. Wydajność silnika oraz jego możliwości opisano w publikacji autorstwa jego twórców[5].

Silnik wnioskujący HyLAR może zostać wykorzystany zarówno w postaci biblioteki JavaScript, jak i również w postaci serwera instalowanego poprzez **npm** (Node Package Manager). Podczas gdy wersja serwerowa posiada funkcjonalność wystarczającą do efektywnego wnioskowania, w przypadku zaimplementowanej aplikacji komunikacja z tym serwerem zawierałaby wiele sekwencji kolejnych zapytań HTTP, co nie byłoby efektywne. W związku z tym wybrano opcję stworzenia serwera wnioskującego wykorzystującego bibliotekę HyLAR z implementacją dopasowaną do potrzeb aplikacji. Szczegółowy opis znajduje się w rozdziale 3. pracy.

Najważniejszymi funkcjami biblioteki, które zostały wykorzystane w pracy, są:

- *load(ontology, mimeType)* – ładuje ontologię podaną w pierwszym argumencie do modelu danych korzystając z jednej ze wspieranych serializacji podanych w drugim argumencie, np. „text/turtle” dla formatu Turtle

- *query(q)* – wykonuje zapytanie SPARQL *q* w postaci *String* na załadowanym modelu danych
- *parseAndAddRule(rule, name)* – dodaje regułę *rule* o nazwie *name* do zestawu reguł, na podstawie których przeprowadzane jest wnioskowanie
- *classify()* – klasyfikuje załadowany model danych w oparciu o dostępne reguły.

Oprócz możliwości definicji własnych reguł, HyLAR oferuje wiele predefiniowanych reguł pochodzących z RDFS (RDF Schema) i OWL 2 RL (Web Ontology Rule Language). Poszerza to możliwości silnika bez potrzeby definiowania zestawu poszczególnych reguł przy każdym uruchomieniu programu.

Rozdział II.
Komunikacja bezprzewodowa

2.1. Bluetooth Low Energy

Klasyczny protokół Bluetooth został zaprojektowany do bezprzewodowej komunikacji pomiędzy urządzeniami wymagającymi ciągłego połączenia i do dziś pełni taką funkcję. Wykorzystywany jest na przykład do przesyłania danych audio pomiędzy smartfonem a bezprzewodowymi słuchawkami czy zestawami głośnomówiącymi. Niestety takie połączenia cechują się wysokim zużyciem energii, a gdy celem jest zapewnienie bezprzewodowej komunikacji z urządzeniami z małymi bateriami, takimi jak na przykład sensory lub opaski monitorujące aktywność, klasyczny Bluetooth nie jest dobrym rozwiązaniem.

Bluetooth Low Energy (BLE) to technologia spokrewniona z klasycznym Bluetooth, która również została zaprojektowana przez Bluetooth Special Interest Group (Bluetooth SIG). Technologia ta, jak jej nazwa może podpowiadać, ma na celu znaczące zredukowanie poboru energii w porównaniu do klasycznego Bluetooth przy zapewnieniu podobnego zasięgu, dzięki czemu świetnie nadaje się do zastosowania w urządzeniach z baterią o niewielkiej pojemności[6].

W porównaniu do klasycznego Bluetooth, BLE cechuje się mniejszą przepustowością danych. Zazwyczaj jednak nie wpływa to na jego efektywność podczas stosowania w sensorach i rozwiązaniach IoT, gdzie przesyłane pakiety są bardzo małe w porównaniu do pakietów audio lub plików.

Urządzenia korzystające z Bluetooth Low Energy można podzielić na cztery role:

- Broadcaster – urządzenie rozgłasza dane
- Peripheral – urządzenie rozgłasza dane, ale może odebrać połączenie od innych urządzeń
- Observer – urządzenie skanujące w celu znalezienia danych rozgłaszanych przez inne urządzenia
- Central – urządzenie, które może zarówno skanować w celu znalezienia danych wystawionych przez inne urządzenia jak i łączyć się z nimi.

Po prezentacji standardu Bluetooth Low Energy, pierwsze dwie role były powszechne dla urządzeń typu beacon i sensorów, takich jak termometry, a rola trzecia i czwarta dla telefonów i komputerów. Od tamtej pory jednak technologia ta rozwinęła się i wiele urządzeń wspiera wszystkie cztery role i może działać w więcej niż jednej roli naraz. Android zawiera wbudowane wsparcie dla BLE w roli centralnej od 18 poziomu API (Android 4.3). Opisywana w tej pracy aplikacja na system Android opiera się właśnie na takiej funkcjonalności.

2.2. Komunikacja z urządzeniami Bluetooth Low Energy

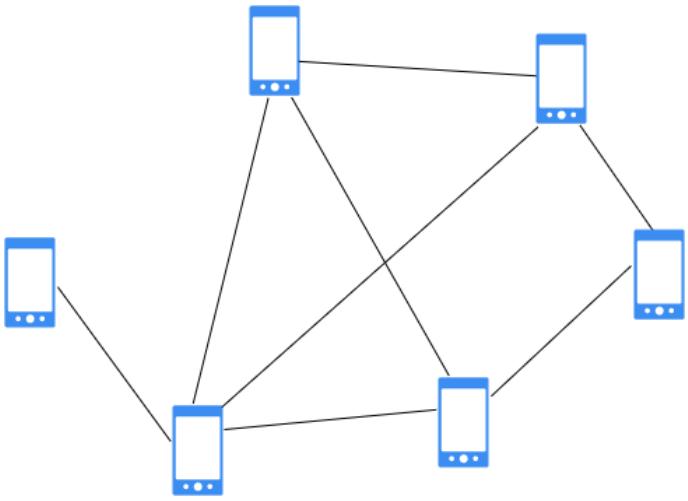
Zamysłem pracy jest umożliwienie wnioskowania w sieci złożonej z wielu urządzeń, tak więc jedną z niezbędnych funkcjonalności systemu jest możliwość komunikacji z innymi systemami wnioskującymi, które działają w tym samym czasie na innych smartfonach (Android i iOS).

Początkowe prace postępowały z założeniem użycia standardu **Bluetooth Mesh** do zapewnienia funkcjonalności komunikacji poprzez Bluetooth Low Energy z innymi telefonami, na których w tym samym czasie działałaby ta sama aplikacja systemu wnioskującego (lub jej odpowiednik na systemie iOS).

Bluetooth Mesh stosuje topologię siatki – sieci, w której węzły (urządzenia) mogą wysyłać i odbierać informacje do i od innych węzłów, nawet gdy nie mają z nimi bezpośredniego połączenia. Oferuje także między innymi możliwość tworzenia podsieci i jednocześnie również zapewnia bezpieczeństwo na wielu poziomach, między innymi poprzez klucze *AppKeys*. Wszystkie węzły w sieci posiadają przynajmniej jeden klucz sieci (*NetKey*), z których każdy odpowiada podsieci, do których należy dany węzeł. Posiadanie *NetKey* pozwala węzłowi na deszyfrację i uwierzytelnienie na poziomie sieci, co udostępnia podstawowe funkcje Bluetooth Mesh, takie jak przekazywanie wiadomości, ale nie pozwala na odczytywanie informacji w nich zawartych. Każdy węzeł ma również swój unikatowy klucz *DeviceKey* używany do procesu zaopatrzenia (ang. provisioning) i konfiguracji węzła. W przypadku usunięcia danego węzła z sieci aplikacja zaopatrująca dodaje go do czarnej listy i inicjowana jest procedura odświeżenia kluczy – wszystkie węzły oprócz tych, które znajdują się na czarnej liście otrzymują nowe klucze. Zapobiega to nieautoryzowanemu użyciu odłączonego węzła w przyszłości[7].

W przypadku opisywanej aplikacji protokół taki jak Bluetooth Mesh świetnie nadałby się do realizacji funkcjonalności przekazywania wywnioskowanych faktów pomiędzy urządzeniami poprzez swoje zaawansowane mechanizmy. Jednocześnie zapewniałby bezpieczeństwo sieci oraz możliwość tworzenia podsieci – pozwoliłoby to na funkcjonowanie mniejszych grup systemów wnioskujących, które wciąż mogłyby wymieniać się niektórymi faktami. Niestety jednak jak okazało się po przeprowadzeniu dogłębnego badania środowiska – Android w chwili pisania tej pracy nie ma (i możliwe, że nigdy nie będzie miał) wsparcia dla Bluetooth Mesh w technologicznym stacku Bluetooth. Obecnie jest to rozwiązanie dostępne jedynie dla urządzeń IoT z bibliotekami implementującymi Bluetooth Mesh napisanymi przez firmy trzecie z ograniczeniami możliwości użycia jedynie na urządzeniach tych firm.

Jednak koncept topologii siatki oraz przekazywania informacji pomiędzy urządzeniami i tak został zaimplementowany w aplikacji, aczkolwiek w prostszej postaci. Szczegóły implementacji opisano w Rozdziale 3.



Rysunek 4: Topologia siatki.

Przed implementacją komunikacji poprzez Bluetooth Low Energy w aplikacji, należało najpierw zapoznać się z powiązanymi kluczowymi pojęciami:

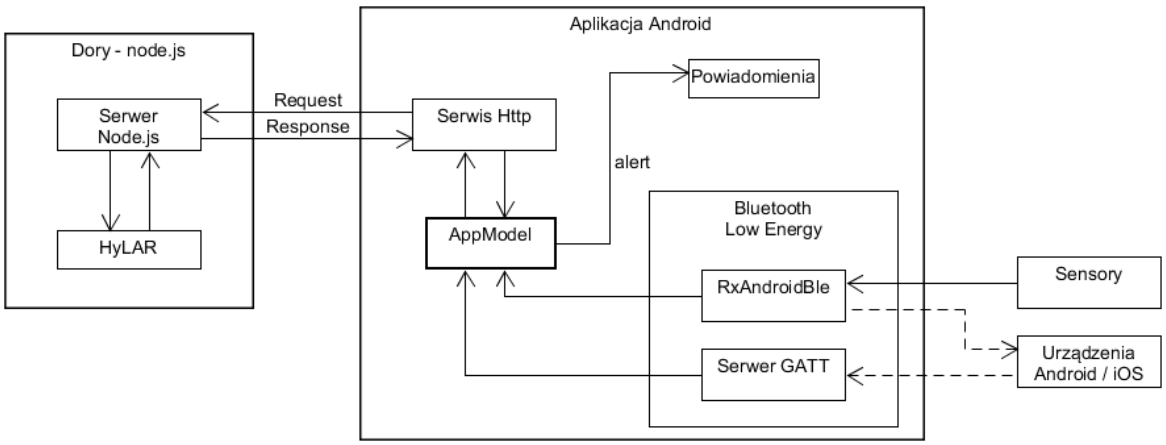
- Generic Attribute Profile (GATT) – Ogólna specyfikacja wysyłania i odbierania krótkich danych zwanych atrybutami poprzez połączenie BLE. GATT jest zbudowany na ATT.
- Attribute Protocol (ATT) – Protokół zoptymalizowany dla urządzeń BLE. Każdy atrybut jest identyfikowany poprzez unikatowy UUID. Atrybuty transportowane przez ATT są formatowane jako charakterystyki i serwisy.
- Universally Unique Identifier (UUID) – ustandaryzowany, unikatowy 128-bitowy format wykorzystywany do identyfikacji informacji.
- Charakterystyka – Typ / Klasa atrybutu zawierająca pojedynczą wartość i 0-n deskryptorów opisujących daną wartość.
- Deskryptor – Zdefiniowane atrybuty opisujące wartość charakterystyki. Opisuje na przykład zakres akceptowalnych wartości czy jednostki pomiaru.
- Serwis – Kolekcja charakterystyk. Również identyfikowany przez UUID.
- Maximum Transmission Unit (MTU) – maksymalna wielkość pakietu ATT. Domyślna wynosi 23 bajty, w tym 3 bajty zarezerwowane, a maksymalna zależy od ograniczeń sprzętowych oraz systemowych.

Rozdział III.
Implementacja systemu wnioskującego

W tym rozdziale przedstawiono stworzony w ramach tej pracy system wnioskujący i poszczególne elementy jego implementacji oraz rozwiązania problemów z nią związanych. Składa się on z dwóch części – aplikacji na system Android oraz serwera *node.js* z silnikiem wnioskującym. Na początek zostanie omówiona architektura w celu przekazania ogólnego zarysu działania, a następnie główne podzespoły.

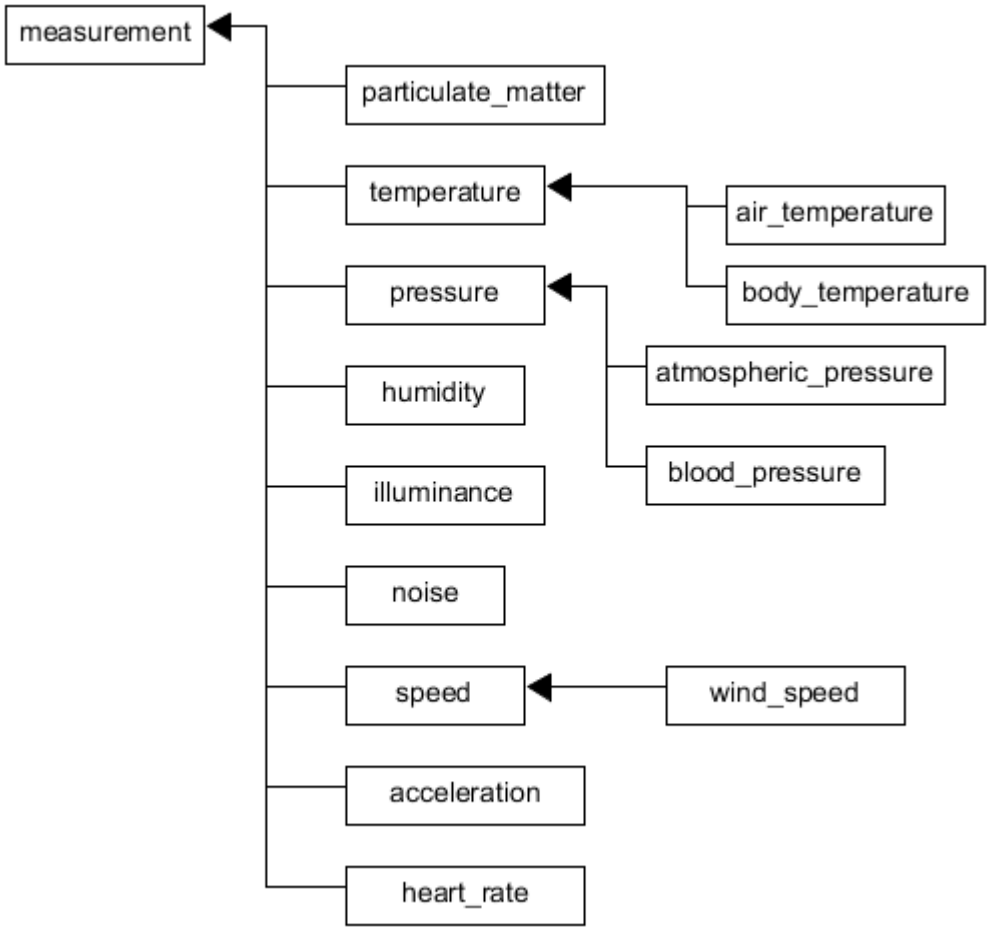
3.1. Architektura systemu wnioskującego

Ze względu na potrzebę wykorzystania aplikacji *Dory – node.js* system wnioskujący został podzielony na dwie części. Pierwsza część to aplikacja Android, która zajmuje się komunikacją z innymi telefonami i sensorami poprzez Bluetooth Low Energy, pobieraniem istotnych dla systemu informacji (np. dane z akcelerometru telefonu), przechowywaniem odpowiednich struktur danych, przesyłaniem danych do serwera wnioskującego oraz wyświetlaniem powiadomień w przypadku wywnioskowania odpowiednich faktów przez silnik wnioskujący. Druga część systemu to uruchomiony w aplikacji Dory serwer node.js z silnikiem wnioskującym HyLAR. Komunikacja pomiędzy aplikacją i serwerem node.js odbywa się poprzez zapytania i odpowiedzi HTTP. Na Rysunku 5. widoczny jest schemat architektury systemu.



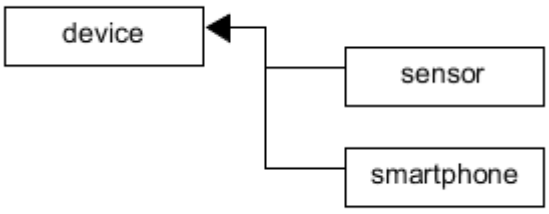
Rysunek 5: Architektura systemu wnioskującego.

Aby wykorzystać pełny potencjał wnioskowania na bazach RDF zaprojektowano ontologię pomiarów. Wygenerowano ją w programie **Protégé** za pomocą OWL 2 Api. Zastosowana ontologia dzieli się na trzy części: pomiary, urządzenia i jednostki. Pierwsza, główna część, zawiera podstawową klasę *measurement* oraz jej podklasy, które specyfikują typ pomiaru. Ważne jest aby każdy pomiar miał określony typ:



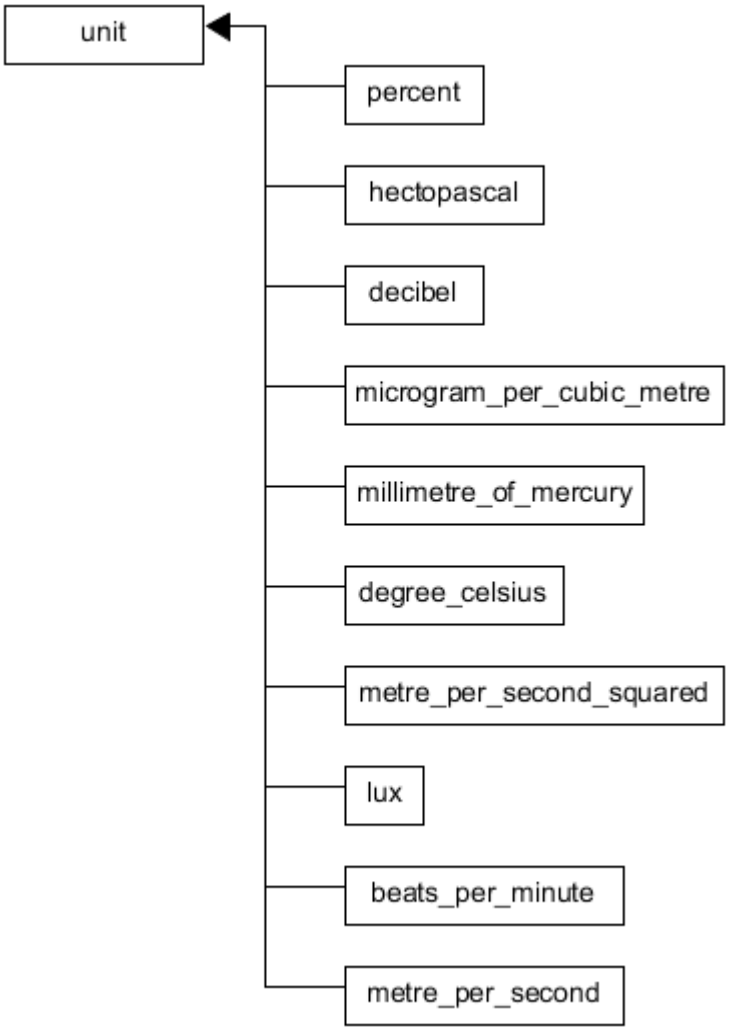
Rysunek 6: Ontologia pomiarów.

Drugą częścią są urządzenia. Każdy zasób pomiaru może mieć także określony typ urządzenia, z którego pochodzi, czyli sensor lub smartfon:



Rysunek 7: Ontologia urządzeń.

Trzecią częścią całej ontologii są jednostki przeprowadzonego pomiaru. Do każdego pomiaru ze zdefiniowanym typem po przeprowadzeniu klasyfikacji / wnioskowania zostanie dołączona odpowiednia jednostka:



Rysunek 8: Ontologia jednostek pomiarów.

Zaprezentowaną ontologię zastosowano w celu wprowadzenia pewnego stopnia ustandaryzowania zapisywanych pomiarów.

3.2. Aplikacja Android

3.2.1. Architektura aplikacji

Pierwszym kluczowym elementem systemu wnioskującego jest aplikacja Android. Została ona napisana w środowisku Android Studio w języku **Kotlin**. Kotlin to język przetwarzany na kod bajtowy (ang. bytecode) Javy, przez co pozwala na korzystanie z bibliotek napisanych w języku Java przeważnie bez problemów, a dodatkowo oferuje mechanizmy pomocne w programowaniu, jak na przykład *null-pointer safety*, dotyczący odwołania do zmiennej, która ma wartość *null*. W przypadku aplikacji mobilnych jedno takie odwołanie może spowodować, że program przestanie działać prawidłowo i aplikacja zostanie zamknięta.

Podstawowym budulcem interfejsu użytkownika w aplikacji Android jest tzw. aktywność (ang. activity). Aktywność to pojedyncza, konkretna rzecz, którą użytkownik może wykonać. Możliwe jest wykonanie całego interfejsu użytkownika na samych aktywnościach, co było tradycyjną

metodą. Zalecanym sposobem wykonania interfejsu jest jednak implementacja tzw. fragmentów – na nich właśnie powstała część aplikacji, z którą użytkownik może oddziaływać.

Fragment reprezentuje zachowanie lub część interfejsu w aktywności. W porównaniu do aktywności, Android API zapewnia wiele udogodnień w korzystaniu z fragmentów. Chcąc przejść do innego fragmentu aplikacji należy wywołać metodę *addToBackStack()* z *FragmentManager*, co również pozwala na bardzo intuicyjną obsługę przycisku wstecz. W dodatku, aby przekazać dane do następnego fragmentu, wystarczy umieścić je w dołączanej do fragmentu paczce *Bundle*. Działanie tego mechanizmu można zaobserwować na przykładzie przejścia z *HomeFragment* do *RuleSetFragment*:

```
val fragment = RuleSetFragment()

val bundle = Bundle()
bundle.putString(EXTRA_ADDRESS_STRING ,name)
bundle.putInt(EXTRA_ADDRESS_INT, position)
fragment.arguments = bundle

activity?.supportFragmentManager?.beginTransaction()?.replace(
   R.id.fragment_container, fragment)?.addToBackStack("tagName")?.commit()
```

Natomiast w docelowym fragmencie, w tym przypadku *RuleSetFragment*, wystarczyło nadpisać metodę *onCreateView* oraz odczytać przekazane wartości, podając odpowiednie klucze:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
   savedInstanceState: Bundle?): View? {

    ruleSetName = arguments?.getString(HomeFragment.EXTRA_ADDRESS_STRING)
    rsIndex = arguments?.getInt(HomeFragment.EXTRA_ADDRESS_INT)

    return inflater.inflate(R.layout.fragment_rule, container, false)
}
```

Tradycyjny interfejs aplikacji Android jest zbudowany na aktywnościach, a zmiana ekranów odbywa się poprzez zmianę aktywności. Jest to starsze rozwiązanie, które niekoniecznie jest gorsze, ale, nie wnikając w szczegóły, na potrzeby tej aplikacji fragmenty wydają się bardziej intuicyjną opcją.

Nie wszystkie dane mogą lub przynajmniej nie powinny być przekazywane od fragmentu do fragmentu. Niektóre obiekty wymagają przechowywania przez cały żywot aplikacji. Przykładem takiego obiektu jest **RxBleClient**. Szczegóły jego implementacji opisano w dalszej części pracy.

Jako mechanizm pozwalający na zapewnienie takiej funkcjonalności została wykorzystana klasa *AppModel* dziedzicząca po klasie *Application* – jest to podstawowa klasa do zarządzania globalnym stanem aplikacji, a utworzenie klasy dziedziczącej pozwala na implementację własnej funkcjonalności. Aby móc z niej skorzystać należało najpierw zadeklarować nazwę klasy w manifeście AndroidManifest.xml:

```
<application
    android:name=".AppModel"
    …
>
```

Następnie potrzebna była prosta implementacja:

```
import android.app.Application

class AppModel : Application() {

   companion object {
      var reasoner: Reasoner = Reasoner()
      var bluetooth: MyBluetooth = MyBluetooth()
   }
}
```

Klasa AppModel jest klasą statyczną, dostępną od włączenia aplikacji. Zawiera ona *companion object*, wewnątrz którego znajdują się ogólnodostępne z całej aplikacji obiekty klas *Reasoner* oraz *MyBluetooth* – te z kolei są strukturami odpowiadającymi za procesowanie i manipulowanie danymi związanymi odpowiednio z wnioskowaniem oraz komunikacją przez Bluetooth Low Energy.

Aby aplikacja mogła w pełni funkcjonować, potrzebuje nadania dostępu do kluczowych elementów w manifeście *AndroidManifest.xml*. Dodatkowo został dodany tag <*uses-feature*> do Bluetooth Low Energy. Tag ten oznacza jedynie, że aplikacja nie pokazywałaby się jako kompatybilna na urządzeniach bez BLE:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.INTERNET"/>

<uses-feature
    android:name="android.hardware.bluetooth_le"
    android:required="true"/>
```

Ważnym elementem aplikacji jest powiązana z czwartą linijką powyższego wycinku manifestu komunikacja HTTP z serwerem. Do jej implementacji wybrano bibliotekę **Retrofit**. Pozwala ona na stworzenie interfejsu odpowiadającego za definicję metod, do których przypisywane są adres zapytania HTTP, jego typ oraz pozostałe elementy jak treść.

Z funkcjonalności biblioteki skorzystano poprzez zbudowanie obiektu z ustawionymi odpowiednimi opcjami, czyli podstawowym adresem na *http://localhost:3000*, pod którym udostępniono serwer wnioskujący opisany w kolejnej części pracy, oraz fabrykami konwertującymi odpowiednie obiekty na treść zapytania HTTP i vice-versa. Następnie utworzono obiekt *service* korzystający z funkcjonalności *retrofit* zgodnie z interfejsem *ReasonerServer* (zawartym w Załączniku 1.) zawierającym metody do komunikacji HTTP:

```
val retrofit: Retrofit = Retrofit.Builder()
   .baseUrl("http://localhost:3000")
   .addConverterFactory(ScalarsConverterFactory.create())
   .addConverterFactory(GsonConverterFactory.create())
   .build()

val service: ReasonerServer = retrofit.create(ReasonerServer::class.java)
```

Aby skomunikować się z serwerem, zaimplementowano metody przygotowujące i odbierające dane do i z zapytań HTTP. Przykładowa funkcja, odpowiadająca za wysyłanie zapytań SPARQL, wygląda następująco:

```
fun query(query: String) {
   val queryReq = QueryRequest(query)
   val req: Call<String> = service.query(queryReq)

   req.enqueue(object: Callback<String> {
      override fun onResponse(call: Call<String>, response: Response<String>) {
         Log.i("Http", "Response received: ${response.body()}")
      }

      override fun onFailure(call: Call<String>, t: Throwable) {
         Log.e("Http", "Server communication error $t")
      }
   })
}
```

Budowany jest obiekt typu *Call* z typem odpowiedzi *String*. Oznacza to, że otrzymana odpowiedź zostanie przekonwertowana na obiekt *String*. Następnie wywołano metodę *enqueue()*, która odpowiada za asynchroniczną komunikację HTTP. W przypadku opisanej implementacji, powyższy blok kodu jest wykonywany z głównego wątku aplikacji, a co za tym idzie zapytanie HTTP musi być asynchroniczne, ponieważ w systemie Android zabronione jest „usypianie" tego wątku – co ma miejsce w synchronicznym odpowiedniku *enqueue()* – *execute()*. Metoda *enqueue()* jako argument przyjmuje obiekt klasy *Callback* z typem obiektu zwrotnego, w tym przypadku *String* i wymaga nadpisania metod *onResponse()* oraz *onFailure()*. Klasa *QueryRequest* to prosta klasa przechowująca obiekt *String query* z konstruktorem na niego. Wykorzystanie takiej klasy jest możliwe dzięki konwersji obiektów na JSON przez wspomnianą wcześniej fabrykę *GsonCoverterFactory*, podczas gdy *ScalarsConverterFactory* odpowiada za konwersję z / do *String*.

```
class QueryRequest {
   var query: String

   constructor(query: String) {
      this.query = query
   }
}
```

Obiekt tej klasy potrzebny jest do wywołania metody zadeklarowanej w interfejsie *ReasonerServer*:

```
@POST("/query")
fun query(@Body query: QueryRequest): Call<String>
```

Co prawda do przesłania zapytania SPARQL wystarczyłby zwykły obiekt *String query*, ale klasą *QueryRequest* pokazano zasadę budowania zapytań HTTP z użyciem biblioteki Retrofit.

Ostatnim istotnym elementem procesu komunikacji z serwerem jest sam proces sprawdzania i przysyłania bufora danych przeznaczonych do silnika wnioskującego. Wykorzystano w tym celu metodę *postDelayed()* z klasy *Handler*, która uruchamia się co wyżej określony czas (w trakcie implementacji systemu czas ten wynosił 5 sekund). W metodzie sprawdzany jest stan bufora i jeśli nie jest pusty zawartość zostaje przesłana do serwera. Niestety, wspomniana metoda *postDelayed()* nie działała poprawnie na starszej wersji systemu Android, podczas gdy problemy nie występowały na nowszej wersji, pomimo że w dokumentacji znaleziono wpis o dostępności tej funkcjonalności od pierwszej wersji Android API.

3.2.2. Komunikacja BLE z sensorami

Aby porozumieć się z urządzeniem poprzez Bluetooth Low Energy, aplikacja musi skorzystać z GATT. Zbieranie danych z urządzeń BLE przez aplikację odbywa się za pomocą powiadomień o zmianach wartości charakterystyk. W tej części zostanie opisana implementacja tej funkcjonalności.

Tradycyjny sposób implementacji łączności BLE w aplikacji Android wiąże się z dużą ilością nadpisywania metod w określonych klasach, więc do komunikacji z sensorami została użyta biblioteka **RxAndroidBle**. Biblioteka ta implementuje metody z **RxJava** do celów łączności BLE, co pozwala na prostsze oprogramowanie funkcjonalności BLE z wykorzystaniem konceptów z **ReactiveX**, między innymi Observables.

Aby klient biblioteki RxAndroidBle mógł być wykorzystywany z dowolnego miejsca w aplikacji, stworzono i przechowano instancję obiektu klasy RxBleClient w kontekście aplikacji w klasie AppModel:

```
AppModel.bluetooth.rxBleClient = RxBleClient.create(applicationContext)
```

Aby następnie połączyć się z urządzeniem BLE o znanym adresie MAC utworzono obiekt **RxBleDevice**, oraz wykonano operacje na obiekcie **CompositeDisposable** w metodzie *connect()*. Aby zakończyć połączenie, wywołana zostaje metoda *disconnect()*:

```
val device: RxBleDevice = AppModel.bluetooth.rxBleClient!!.getBleDevice(mac)
lateinit var connectionObservable: Observable<RxBleConnection>
val charConnectionDisposable = CompositeDisposable()
val disconnectTriggerSubject = PublishSubject.create<Unit>()

fun connect() {
   connectionObservable = device
      .establishConnection(false)
      .takeUntil(disconnectTriggerSubject)
      .compose(ReplayingShare.instance())

   connectionObservable
      .flatMapSingle { it.discoverServices() }
      .observeOn(AndroidSchedulers.mainThread())
      .doOnSubscribe { Log.d(TAG, "BLE: Connecting") }
      .subscribe(
         { characteristic ->
            Log.d(TAG, "BLE: Connection established")
         },
         { Log.d(TAG, "BLE: Connection failed") },
         { Log.d(TAG, "BLE: On connection complete") }
       )
      .let { charConnectionDisposable.add(it) }
}
fun disconnect() {
   if (device.isConnected) {
      disconnectTriggerSubject.onNext(Unit)
   }
}
```

Powyższy kod umożliwia rozpoczęcie i zakończenie połączenia z urządzeniem BLE poprzez wykonywanie operacji na obiekcie *connectionObservable*, który reprezentuje stan połączenia z urządzeniem oraz jest ogólnym kanałem komunikacji z nim. Następnym krokiem jest

rejestracja powiadomień o zmianach wartości wybranych charakterystyk – po nawiązaniu połączenia możliwe jest wykonanie następujących operacji spełniających tę funkcjonalność w metodzie *registerNotifications()*:

```
fun registerNotifications() {
   if (device.isConnected) {
      connectionObservable
         .flatMap { it.setupNotification(BUTTON_STATE_UUID) }
         .doOnNext {
            Log.d("BLE", "Notification has been set up")
            watchedCharacteristics.add(BUTTON_STATE_UUID)
         }
         .flatMap { it }
         .observeOn(AndroidSchedulers.mainThread())
         .subscribe({ onNotificationReceived(it) }, { onNotificationFailure(it) })
         .let { charConnectionDisposable.add(it) }
   }
}

fun onNotificationReceived(bytes: ByteArray) {
   watchedCharacteristicsValues.add(watchedCharacteristics[0].toString() + ":
${bytes.toHex()}")
   Log.d(TAG, watchedCharacteristics[0].toString() + ": ${bytes.toHex()}")
}
fun onNotificationFailure(throwable: Throwable) {
   Log.d(TAG, "BtDevice '$mac' onNotificationFailure: $throwable")
}
```

Wewnątrz metody wykonywane są kolejne operacje na obiekcie *connectionObservable*. W pierwszej operacji *flatMap* rejestrowane są powiadomienia na zmianę stanu wciśnięcia przycisków urządzenia za pomocą metody *setupNotification()* obiektu *connectionObservable*. Poprzez wykonanie kolejnych operacji oraz *subscribe()* wskazywane są metody wykonywane w przypadku otrzymania powiadomienia.

3.2.3. Komunikacja BLE z innymi telefonami

Do komunikacji z innymi urządzeniami Android / iOS kluczowa jest wspomniana w poprzednim rozdziale funkcjonalność Bluetooth Low Energy – rozgłaszanie (ang. advertisement). Urządzenie z uruchomioną aplikacją odbierające informacje od innego urządzenia pełni rolę urządzenia peryferyjnego i oczekuje na nadchodzące połączenie.

Do komunikacji z innymi telefonami ustawiony i skonfigurowany został serwer GATT w metodzie *initGatt* klasy *MyBluetooth*. Zostaje ona wywołana z przekazaniem kontekstu aplikacji *applicationContext* przy starcie aplikacji w głównej aktywności:

```
fun initGatt(c: Context) {
   val gattServerCallback = GattServerCallback()
   gattServer = btManager?.openGattServer(c, gattServerCallback)
   setupGatt()
}

// MainActivity.kt
AppModel.bluetooth.initGatt(applicationContext)
```

W pokazanym wyżej kodzie wykorzystana została instancja klasy *BluetoothManager* oraz *GattServerCallback* dziedzicząca po *BluetoothGattServerCallback*. Jej implementacja została

dołączona w Załączniku 2. i zostanie omówiona w dalszej części tego podrozdziału. W funkcji *setupGatt()* do serwera GATT dodany zostaje serwis zawierający charakterystykę, oba z unikatowymi UUID. Charakterystyka pełni rolę „docelowego adresu", na który przesyłane są później informacje z innych urządzeń BLE. Wymagane jest określenie typu charakterystyki i nadania odpowiedniego pozwolenia, aby mogła ona otrzymywać dane.

```
private fun setupGatt() {
   service = BluetoothGattService(MY_SERVICE_UUID,
BluetoothGattService.SERVICE_TYPE_PRIMARY)
   writeChar = BluetoothGattCharacteristic(
      MY_CHAR_UUID,
      BluetoothGattCharacteristic.PROPERTY_WRITE,
      BluetoothGattCharacteristic.PERMISSION_WRITE)
   service?.addCharacteristic(writeChar)
   gattServer!!.addService(service)
}
```

Aby urządzenie peryferyjne było widoczne podczas procesu skanowania urządzeń BLE przez urządzenie centralne, wymagane jest rozpoczęcie rozgłaszania z danymi *AdvertiseData* oraz konfiguracją *AdvertiseSettings*:

```
val P_UUID: ParcelUuid = ParcelUuid(MY_SERVICE_UUID)

val adSettings: AdvertiseSettings = AdvertiseSettings.Builder()
   .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
   .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
   .setConnectable(true)
   .build()

btAdvertiser = bluetoothAdapter?.bluetoothLeAdvertiser

val adData = AdvertiseData.Builder()
   .setIncludeDeviceName(false)
   .addServiceUuid(P_UUID)
   .build()

fun advertise() {
   btAdvertiser?.startAdvertising(adSettings, adData, advertisingCallback)
}
```

Metoda *Builder()* klasy *AdvertiseSettings* pozwala na skonfigurowanie procesu rozgłaszania. Od trybu oraz mocy zależy częstotliwość i zasięg rozgłaszanego pakietu – tryb ustawiono na zbalansowany, a moc na średnią, co pozwala na zachowanie stosunkowo dużej wykrywalności przy mniejszym zużyciu energii w porównaniu do trybu „low latency" i wysokiej mocy. Wywołano metodę *setConnectable(true)*, aby umożliwić połączenie się z rozgłaszającym urządzeniem przez urządzenie, które odbierze rozgłaszany pakiet. Metodą *Builder()* klasy *AdvertiseData* do pakietu dodano jedynie UUID serwisu opakowane w obiekt *ParcelUuid*. Istotnym elementem rozgłoszenia jest to, że rozgłaszany pakiet może zawierać jedynie do 31 bajtów informacji. Jeśli obiekt *AdvertiseData* podany do metody *startAdvertising()* będzie większy niż 31 bajtów – wystąpi błąd i żadne dane nie zostaną rozgłoszone. Z tego powodu ważna jest minimalizacja ich wielkości. Dlatego również w implementacji nie została zawarta nawet nazwa urządzenia. Trzecim argumentem *startAdvertising()* jest obiekt klasy *AdvertisingCallback* – nadpisuje ona klasę *AdvertiseCallback* jedynie po to, aby wypisać logi o sukcesie lub porażce (wraz z kodem błędu) próby startu rozgłaszania.

Sukces próby startu rozgłaszania oznacza, że urządzenie jest gotowe na odbieranie wiadomości. Jednak aby miało co odbierać, po drugiej stronie procesu komunikacji urządzenie nadające musi w jakiś sposób wysłać wiadomość. Proces nadawania pakietów BLE zaczyna się od skanu rozgłaszających urządzeń BLE w zasięgu:

```
val subscription = rxBleClient?.scanBleDevices(
   ScanSettings.Builder()
      .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
      .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
      .build(),
   ScanFilter.Builder()
      .setServiceUuid(P_UUID)
      .build()
)!!
```

Skan zostaje wywołany z ustawionym filtrem na UUID serwisu zawartego w rozgłaszanym pakiecie. Dzięki temu filtrowi nadające urządzenie nie próbuje wysyłać zbędnych pakietów do urządzeń, które nie są częścią sieci wnioskującej. Tryb skanu ustawiono na tryb „low latency" ze względu na potencjalną częstotliwość potrzeby przesłania nowo wywnioskowanych faktów. Biorąc pod uwagę możliwie dużą liczbę urządzeń wykrytych przez skan oraz bliżej nieprzewidywalną naturę Bluetooth, uznano że proces skanu i przesłania wiadomości powinien trwać jak najkrócej. Oczywistą wadą tego trybu jest zwiększone zużycie baterii, jednak możliwe, że lepiej jest skanować i wysyłać krótko i bardziej intensywnie, niż długo i oszczędnie.

Proces skanu przekazuje wszystkie wyniki przechodzące przez filtr do kolejnej części strumienia operacji, którą jest przesłanie danych:

```
.forEach { scanResult ->
   scanResult.bleDevice.establishConnection(false)
      .flatMap { rxBleConnection -> rxBleConnection.createNewLongWriteBuilder()
         .setCharacteristicUuid(MY_CHAR_UUID)
         .setBytes("$messageId;$messageTtl;$message".toByteArray())
         .build()
      }
```

Do tego celu wykorzystano operację typu *long write* zapewnioną przez bibliotekę RxAndroidBle. Ze względu ograniczenia wielkości pakietu BLE do 20 bajtów, wysłanie dłuższych wiadomości wymaga wykonania operacji *write* wiele razy z rzędu na tym samym połączeniu. Alternatywą jest ustawienie większego MTU i następnie wysyłanie dłuższych wiadomości, jednak nie wszystkie smartfony dobrze wspierają tę funkcjonalność, więc zdecydowano się na wykorzystanie *long write* na podstawowej wielkości MTU. Format wiadomości ustalono na kolejne wartości oddzielone średnikami:

- identyfikator – złożony z adresu MAC Bluetooth urządzenia wysyłającego wraz z liczbą wylosowaną z przedziału 100000-999999
- time-to-live – liczba przekazań wiadomości, zmniejszana o jeden przy każdym odebraniu wiadomości
- wiadomość – dane w postaci zasobów RDF.

Po drugiej stronie połączenia, odbiorca ma za zadanie odebrać pełną wiadomość ze zrozumieniem. Odbywa się to poprzez kolejno odbieranie kolejnych 20 bajtowych pakietów, konwertowanie ich z ByteArray do *String* z kodowaniem UTF-8, dodawanie do jednej zmiennej

typu *String* i w końcu parsowanie całej wiadomości, gdy połączenie zostanie zakończone, na podstawie oddzielających wartości średników:

```
val messageId = bleMsgBuffer.substringBefore(";")
val messageTtl = bleMsgBuffer.substringAfter(";").substringBefore(";")
val message = bleMsgBuffer.substringAfter(";").substringAfter(";")
```

Następnie zmniejszana jest wartość TTL i jeśli wciąż jest ona większa od 0 wiadomość zostaje przesłana dalej tym samym sposobem. Treść wiadomości zostaje wstawiona do kolejki do przesłania do serwera wnioskującego, tym samym kończąc proces przesłania.

Zaimplementowany system komunikacji z innymi urządzeniami Android / iOS poprzez Bluetooth Low Energy jest prosty w porównaniu do rozbudowanego protokołu Bluetooth Mesh, jednak zapewnia podstawową funkcjonalność przesyłania wiadomości. Podczas gdy prostota jest zaletą tego rozwiązania, ma ono swoje wady. Nie zapewniane jest jakiekolwiek bezpieczeństwo – zarówno na poziomie transportowym BLE jak i na poziomie aplikacyjnym. System operacyjny Android udostępnia pakiety otrzymywane przez Bluetooth Low Energy wszystkim aplikacjom z odpowiednimi uprawnieniami. Z pewnością przydałaby się implementacja szyfrowania / deszyfrowania pakietów na poziomie aplikacji, jeśli ta miałaby być faktycznie używana, jednak zapewnienie bezpieczeństwa komunikacji przez BLE nie było celem tej pracy. Kolejną wadą jest brak oczywistego rozwiązania umożliwiającego tworzenie podsieci systemów wnioskujących. Namiastką takiej funkcjonalności jest korzystanie z różnych UUID serwisu, który jest rozgłaszany i wyszukiwany podczas skanowania (w kodzie oznaczony jako MY_SERVICE_UUID). Umożliwiłoby to funkcjonowanie różnych „podsieci" systemów wnioskujących na tym samym obszarze, ale nie byłyby to prawdziwe podsieci, ponieważ aplikacje z różnymi UUID rozgłaszanego serwisu nie byłyby dla siebie nawzajem widoczne podczas skanowania o ile nadal miałoby być stosowane ograniczenie na wysyłanie wiadomości jedynie do urządzeń rozgłaszających dany, znany UUID.

3.3. Serwer wnioskujący

3.3.1. Architektura serwera wnioskującego

Serwer wnioskujący jest drugim kluczowym elementem zaimplementowanego systemu wnioskującego. Odpowiada on za odbieranie danych przesyłanych z aplikacji, wykonywanie odpowiednich operacji z silnikiem wnioskującym HyLAR i odsyłanie odpowiedzi. Do tych celów wykorzystano aplikację *Dory – node.js / javascript / git / ssh server.* Udostępnia ona funkcjonalność niektórych narzędzi natywnie niewspieranych na mobilnych systemach operacyjnych. Przede wszystkim pozwoliła ona na uruchomienie programu napisanego w JavaScript na lokalnym – dla telefonu – serwerze node.js, dzięki czemu możliwe stało się wykorzystanie silnika HyLAR w systemie wnioskującym.

Aplikacja *Dory* znacząco ułatwia proces kompilacji programów JavaScript oraz hostowania serwerów node.js w tle, w porównaniu do rozważanych alternatyw jak **AndroidJS** czy **Node.js for Mobile Apps**.

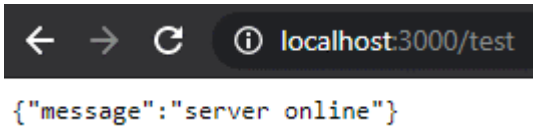
W przypadku wspomnianych alternatyw trzeba stworzyć osobną aplikację, do której wymagana jest wcześniejsza konfiguracja środowiska, a następnie dochodzi problem implementacji ciągłości funkcjonowania serwera jako serwis działający w tle. Proces uruchomienia serwera wnioskującego przy użyciu aplikacji Dory jest zdecydowanie mniej skomplikowany – wystarczy wybrać odpowiedni plik z kodem JavaScript i wcisnąć *Start*.

Udostępniane i zachowywane są także logi z serwera, co bardzo pomogło w procesie tworzenia systemu.

Do implementacji komunikacji HTTP po stronie serwera zastosowane zostały biblioteki **express** i **body-parser** (w wersji express 4.16.0 body-parser został do niego dołączony). Express pozwala na uruchomienie serwera node.js w kilku prostych krokach, a body-parser „wyciąga" część body zapytania HTTP i udostępnia ją pod *request.body*. Każda funkcja podpięta do danej ścieżki i typu zapytania przyjmuje argumenty zapytania *req* oraz odpowiedzi *res*. W celu prostego testu funkcjonalności napisany został przykładowy program:

```
var express = require('express');
var app = express();
var port = 3000;
var bodyParser = require('body-parser');
app.use(bodyParser.json());

var routes = function (app) {
    var ctrl = {
        onlineTest : function (req, res) {
            var response = {
                message : 'server online'
            };
            res.json(response);
        }
    }
    app.route('/test')
    .get(ctrl.onlineTest)
}

routes(app);
app.listen(port, function () {
    console.log('Reasoner server started on: ' + port);
});
```

Po uruchomieniu aplikacji na node.js i wejściu w przeglądarce na adres *http:///localhost:3000/test* wyświetlony zostanie obiekt JSON widoczny na Rysunku 9.



Rysunek 9: Test serwera node.js.

Analogicznie do powyższego przykładu zaimplementowano endpointy służące do obsługi wnioskowania:

```
app.route('/ontology')
    .get(ctrl.getOntology)
    .post(ctrl.setOntology)

    app.route('/rules')
    .get(ctrl.getRules)
    .post(ctrl.setRules)

    app.route('/rules/add')
    .post(ctrl.addRules)
```

```
    app.route('/query')
    .post(ctrl.query)

    app.route('/reason')
    .post(ctrl.reason)
```

Główne funkcje programu opisano w następnej części.

3.3.2. Implementacja silnika wnioskującego

HyLAR dostosowano na potrzeby systemu wnioskującego, integrując go z funkcjonalnością biblioteki express. W pokazanych fragmentach kodu obiekt *h* nawiązuje do instancji biblioteki *hylar*, zainicjowanej poprzez:

```
const Hylar = require('hylar');
const h = new Hylar();
```

Do podstawowej funkcjonalności serwera wnioskującego należą: ładowanie nowej ontologii do modelu danych, dodawanie reguł wnioskowania i wykonywanie zapytań SPARQL. Do ładowania ontologii wykorzystano opisaną w Rozdziale 1. prostą metodę *load()*, do dodawania reguł *parseAndAddRule()* w pętli, a zapytania SPARQL są przekazywane do metody *query()*.

Gdy reguły zostaną już załadowane do silnika, a ontologia do modelu danych i przeprowadzona zostanie pierwsza klasyfikacja (automatycznie podczas wykonywania metody *load()*), serwer staje się gotowy na odbiór nowych danych.

Najważniejszą funkcjonalnością serwera wnioskującego jest funkcja dostępna pod ścieżką *'/reason'*. To ona jest odpowiedzialna za przeprowadzenie kolejnej iteracji wnioskowania z uwzględnieniem nowo dostarczonych danych. Wykonane zostają w niej synchronicznie operacje dodania danych wyciągniętych z zapytania HTTP do załadowanego modelu, przeprowadzenia wnioskowania na podstawie załadowanych reguł (funkcja *classify()*) oraz wyszukania i zwrócenia istotnych informacji na temat zasobów z modelu spełniających wyrażenie:

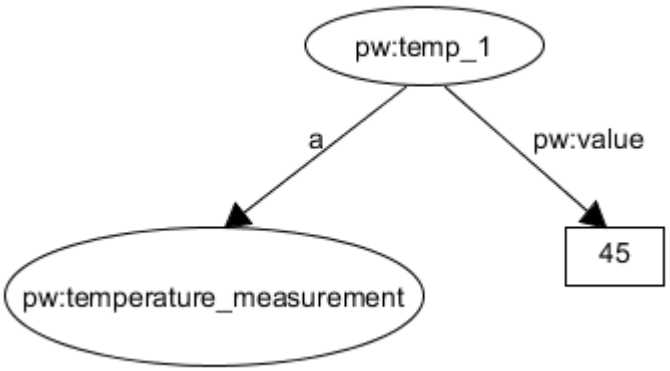
```
await h.query(
    "PREFIX pw: <http://pw.edu.pl/onto/main-ontology.owl#> INSERT DATA { " +
    req.body.buffer +
    " }"
)

await h.classify();

await h.query(
    "PREFIX pw: http://pw.edu.pl/onto/main-ontology.owl# " +
    "SELECT * WHERE { " +
    "?x pw:reasoned ?y . ?x pw:value ?val . ?x pw:reasoned_type ?type }"
)
```

Zapytanie *SELECT* * zwraca wszystkie zmienne oznaczone znakiem zapytania w regule *WHERE*. Dzięki tej operacji aplikacja dostaje informacje zwrotne, z których dowiaduje się (odpowiednio *?y*, *?val* i *?type*) czy zostało coś wywnioskowane, a jeśli tak to na jakiej podstawie została wywnioskowana, jaką miała wartość oraz jakiego typu była ta informacja.

3.4. Przykład zastosowania aplikacji

W tej części opisano przykład implementacji funkcjonalności powiadamiania użytkownika o wysokiej temperaturze. Do testu wykorzystano dwa telefony – jeden z systemem Android 7.0 oraz jeden z systemem Android 9.0. Pierwszy wykorzystano do symulacji urządzenia BLE – ze względu na wersję systemu operacyjnego i powiązane z nią problemy z metodą *postDelayed()* – które dokonało pomiaru temperatury powietrza o wysokości 45°C. Informację o pomiarze przesłano do drugiego telefonu, na którym przeprowadzono wnioskowanie i wyświetlono ostrzeżenie o wysokiej temperaturze.

Schemat danych zasymulowanego pomiaru przedstawiono na Rysunku 10. Wykorzystano ontologię opisaną na początku tego rozdziału. Predykat *a* jest skrótem od URI *http://www.w3.org/1999/02/22-rdf-syntax-ns#type* pochodzącego z przestrzeni nazw RDF.



Rysunek 10: Schemat przykładowego pomiaru.

Wnioskowanie przeprowadzono z wykorzystaniem pojedynczej reguły o nazwie *hot_alert*. Pełne URI zastąpiono prefiksowanymi wartościami dla lepszej czytelności – metoda parsowania reguł w silniku HyLAR, w przeciwieństwie do parsera serializacji, nie rozumie prefiksów:

```
(hot_alert,
(?x ?y pw:temperature_measurement) ^ (?x pw:value ?val) ^ (?val > 40) -> (?x
pw:reasoned pw:alert) ^ (?x pw:reasoned_type pw:high_temperature) )
```

Po odbiorze informacji zwrotnej od serwera o nowych wnioskach aplikacja sprawdza jej zawartość – jeśli wystąpił *pw:alert* wyświetlone zostaje odpowiednie dla danego typu ostrzeżenia powiadomienie. Dla danych testowych wykryto ostrzeżenie typu *pw:high_temperature*, więc w wyniku przeprowadzonego testu telefon Android 9.0 wyświetlił powiadomienie ostrzegające o wysokiej temperaturze widoczne na Rysunku 11.

Rysunek 11: Powiadomienie o wysokiej temperaturze.

Logi z serwera wnioskującego dołączono w Załączniku 3.

Przykładowa implementacja zastosowania aplikacji na pierwszy rzut oka zadziała jak należy, jednak po kolejnych iteracjach klasyfikacji na załadowanej ontologii silnik wnioskujący HyLAR zwalnia, a czas przeprowadzania wnioskowania znacznie wzrasta. Po zaledwie kilku iteracjach pojedyncza klasyfikacja wydłuża się z ułamka sekundy do kilku sekund. Podejrzewanym powodem takiego zachowania jest duża ilość pustych węzłów / zasobów (ang. blank nodes) w wygenerowanej ontologii, ponieważ wypisując model danych po każdym kolejnym procesie wnioskowania zauważono wzrost ich liczby z ok. 50 do ok. 150, potem 300 itd. Oznacza to, że silnik HyLAR może nie być zoptymalizowany do obsługi pustych zasobów. Problem coraz dłuższego wnioskowania można jednak wyeliminować refaktoryzując zapis ontologii w serializacji Turtle pozbywając się pustych zasobów, ponieważ podczas testów na małych modelach danych, w których nie było żadnych takich zasobów, silnik sprawdzał się bardzo dobrze.

Docelowo system wnioskujący może zostać zastosowany na wiele sposobów i w różnych obszarach – zarówno w celu ostrzegania o zagrożeniu, jak i w celach rekreacyjnych, na przykład jako system:

- System wczesnego ostrzegania przed atakiem / pożarem / wypadkiem. Jego reguły mogłyby także określać bezpieczny kierunek sprawnej ewakuacji.
- System wczesnego wykrywania objawów chorób – w przypadku zaraźliwych chorób nawet zwykłe monitorowanie temperatury ciała użytkownika mogłoby ograniczyć prawdopodobieństwo zarażenia kolejnych osób.
- System informujący osoby znajdujące się w otoczeniu o pogorszeniu stanu zdrowia danego użytkownika, np. osoby chorej na cukrzycę lub alergika.
- System drogowy, który informuje kierowców o warunkach jazdy.
- System dla imprez masowych, który przedstawia atrakcyjne miejsca, pomaga unikać kolejek oraz pomaga w ewakuacji.
- System wsparcia dla zawodników i trenerów drużyn w dyscyplinach zespołowych i grach terenowych, który monitoruje stan zawodników i powiadamia o nim trenerów.

Zakończenie

Podsumowując pracę o tytule „Implementacja systemu wnioskującego w systemie operacyjnym Android”, jej celem było stworzenie systemu wnioskującego, który jest dowodem koncepcji, że już w niedalekiej przyszłości systemy wnioskujące będą w stanie wspierać ludzi na porządku dziennym.

W pierwszym rozdziale pracy przybliżono pojęcia i koncepty związane z systemami wnioskującymi i zasadami ich działania. Pokazano również architekturę rozbudowanego frameworka Apache Jena, którego ostatecznie nie wykorzystano w rozwiązaniu oraz przedstawiono lekką (w porównaniu do Jeny) alternatywę silnika wnioskującego – HyLAR.

W drugim rozdziale wytłumaczono podstawowe pojęcia Bluetooth i nowszego, spokrewnionego z nim, Bluetooth Low Energy oraz opisano różnice między nimi i omówiono funkcjonalności BLE wykorzystane w pracy.

Na początku trzeciego rozdziału przedstawiono ogólną architekturę systemu wnioskującego i zastosowaną w nim ontologię. Następnie wdano się w szczegóły implementacji zarówno aplikacji Android, jak i również serwera wnioskującego. Na koniec tego rozdziału omówiono przykład, w którym zasymulowano pobranie pomiaru o wysokiej temperaturze na jednym telefonie i na jego podstawie system wnioskujący klasyfikując pomiar oznaczył go jako pomiar wysokiej temperatury i wysłał ostrzegające powiadomienie. Przeprowadzony test dowiódł, że rozwiązanie nie jest doskonałe, ale rzeczywiście działa i po odpowiedniej optymalizacji może stać się skutecznym narzędziem.

Praca z pewnością mogłaby zostać rozwinięta w niektórych aspektach. Na początek warto wspomnieć o komunikacji Bluetooth Low Energy – zaimplementowany protokół komunikacyjny jest spełnia swoje zadanie i w porównaniu do przytoczonego rozwiązania Bluetooth Mesh jest o wiele prostszy, ale za to niezbyt bezpieczny. BLE samo w sobie nie posiada zabezpieczeń, więc zastosowany protokół jest podatny na ataki, takie jak odbieranie złośliwych wiadomości czy przechwytywanie informacji przez niepożądane urządzenie. Kolejnym aspektem do poprawy może być wnioskowanie – potencjalnym rozwiązaniem problemów występujących z silnikiem HyLAR mogłoby być zastosowanie innego silnika lub optymalizacja ontologii.

Dziedzina systemów wnioskujących nie jest znana od wczoraj, lecz wydaje się, że niewiele się dzieje w kategorii jej rozwoju. Owszem, tworzone są kolejne ontologie, powstają kolejne modele danych i aplikacje badawcze, jednak brakuje „namacalnych” rozwiązań, które byłyby powszechnie znane i stosowane, obecne w codziennym życiu. Napisaną pracę można interpretować jako próbę „popchnięcia” dziedziny w kierunku rozwoju. W świecie, w którym coraz więcej aspektów jest monitorowanych przez wszechobecne sensory, coraz bardziej realnym rozwiązaniem wydaje się bieżąca analiza danych o otoczeniu w celu wspierania w podejmowaniu decyzji lub ostrzeżeniu na przykład o niebezpieczeństwie lub niesprzyjających dla zdrowia warunkach. Wraz z rozwojem smartfonów, technologii sensorów i rozwiązań IoT wzrasta również możliwość kierowania się w kierunku *edge reasoning – edge computing* dla systemów wnioskujących. Ideą *edge reasoning* są małe urządzenia (w tej chwili smartfony), które nie tylko wnioskują na podstawie danych z bliskiego środowiska, ale także przesyłają ważne informacje do pozostałych urządzeń, tworząc tym samym jeden większy, wydajny system wnioskujący zdolny na wiele, ze względu na rozproszenie wszelkiego wnioskowania i umieszczenie go jak najbliżej źródeł danych.

Na koniec podsumowania, jeszcze raz należą się podziękowania dla promotora dr inż. Mariusza Kamoli za inspirację i pomoc w realizacji pracy inżynierskiej oraz dla mojej rodziny za wsparcie i motywację do pracy.

Bibliografia

[1] A. Maarala, X. Su, J. Riekki. *Semantic reasoning for context-aware Internet of Things applications*. IEEE Internet of Things Journal. 2017.

[2] C. Bobed, F. Bobillo, R. Yus, G. Esteban, E. Mena. *Android Went Semantic: Time for Evaluation*. 2014.

[3] OWL Working Group. *Web Ontology Language (OWL) Overview*. 11 grudzień 2012. https://www.w3.org/OWL.

[4] A. Passant, A. Polleres, P. Gearon. *SPARQL 1.1 Update*. 21 marzec 2013. W3C. https://www.w3.org/TR/sparql11-update.

[5] M. Terdjimi, L. Médini, M. Mrissa. *HyLAR+: Improving Hybrid Location-Agnostic Reasoning with Incremental Rule-based Update*. kwiecień 2016. https://hal.archives-ouvertes.fr/hal-01276558.

[6] Android Developers. *Bluetooth low energy overview*. https://developer.android.com/guide/topics/connectivity/bluetooth-le.

[7] K. Ren, M. Woolley. *Bluetooth Mesh Security Overview*. 11 wrzesień 2017. https://www.bluetooth.com/blog/bluetooth-mesh-security-overview.

Spis ilustracji

Załączniki

Załącznik 1.
Interfejs do komunikacji HTTP z serwerem w aplikacji

```
interface ReasonerServer {
   @GET("/test")
   fun testGet(): Call<String>

   @POST("/test")
   fun testPost(@Body value: QueryRequest): Call<String>

   @POST("/query")
   fun query(@Body query: QueryRequest): Call<String>

   @GET("/ontology")
   fun getOntology(): Call<OntologyRequest>

   @POST("/ontology")
   fun setOntology(@Body ontology: OntologyRequest): Call<String>

   @GET("/rules")
   fun getRules(): Call<RulesRequest>

   @POST("/rules")
   fun setRules(@Body rules: RulesRequest): Call<String>

   @POST("/rules/add")
   fun addRules(@Body rules: RulesRequest): Call<String>

   @POST("/reason")
   fun reason(@Body buffer: ReasoningRequestWithBuffer): Call<String>
}
```

Załącznik 2.
Implementacja GattServerCallback

```
class GattServerCallback : BluetoothGattServerCallback() {

   override fun onConnectionStateChange(device: BluetoothDevice?, status: Int,
newState: Int) {
      super.onConnectionStateChange(device, status, newState)
      if (newState == BluetoothProfile.STATE_CONNECTED) {
         connectedBleClients.add(device!!)
         Log.d("BLE", "Connected to device: ${device.address}")
      } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
         connectedBleClients.remove(device)
         AppModel.bluetooth.parseBleBuffer()
         Log.d("BLE", "Disconnected from device: ${device?.address}")
      }
   }

   override fun onServiceAdded(status: Int, service: BluetoothGattService?) {
      super.onServiceAdded(status, service)
      Log.d("BLE", "Added service $service with status $status")
   }

   override fun onCharacteristicWriteRequest(
      device: BluetoothDevice?,
      requestId: Int,
      characteristic: BluetoothGattCharacteristic?,
      preparedWrite: Boolean,
      responseNeeded: Boolean,
      offset: Int,
      value: ByteArray?
   ) {
      super.onCharacteristicWriteRequest(
         device,
         requestId,
         characteristic,
         preparedWrite,
         responseNeeded,
         offset,
         value
      )
      if (characteristic?.uuid!! == MY_UUID3) {
         gattServer?.sendResponse(device, requestId, BluetoothGatt.GATT_SUCCESS, 0,
            "success".toByteArray())
         bleMsgBuffer += value?.toString(Charset.forName("UTF-8"))
         Log.d("BLE", "Received write on characteristic ${characteristic.uuid},
            value: ${value?.toString(Charset.forName("UTF-8"))}")
         return
      }
      Log.d("BLE", "Received write for unknown characteristic
${characteristic.uuid}")
   }
}
```

Załącznik 3.
Logi serwera wnioskującego z przykładu 3.4.

```
Reasoner server started on: 3000

// Dodanie reguły 'hot_alert'
Current set of rules:
[ { name: 'prp-dom ',
    rule: '(?p domain ?c) ^ (?x ?p ?y) -> (?x type ?c)' },
  { name: 'prp-rng ',
    rule: '(?p range ?c) ^ (?x ?p ?y) -> (?y type ?c)' },
……
……
  { name: 'hot_alert',
    rule: '(?x ?y temperature_measurement) ^ (?x value ?val) ^ (?val > "40")
           -> (?x reasoned alert)E(?x reasoned_type high_temperature)' } ]

// Załadowanie ontologii
[HyLAR] Classification started.
[HyLAR] Registering derivations to dictionary...
[HyLAR] Registered successfully.
[HyLAR] Classification succeeded.
Ontology load successful

// Dodanie danych o pomiarze z przykładu
Reasoned facts:
[ { x:
     { token: 'uri',
       value: 'http://pw.edu.pl/onto/main-ontology.owl#temp_1' },
    y:
     { token: 'uri',
       value: 'http://pw.edu.pl/onto/main-ontology.owl#alert' },
    val:
     { token: 'literal',
       value: '45',
       type: 'http://www.w3.org/2001/XMLSchema#integer' },
    type:
     { token: 'uri',
       value: 'http://pw.edu.pl/onto/main-ontology.owl#high_temperature' } } ]
Responding with reasoned facts
```