VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

MOBILNÍ APLIKACE PRO PODPORU VZDÁLENÉHO MĚŘENÍ S VYUŽITÍM KOMUNIKAČNÍ SÍTĚ SIGFOX

MOBILE APPLICATION FOR REMOTE MEASUREMENT SUPPORT USING SIGFOX COMMUNICATION NETWORK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **DAVID BULAWA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ŠIMEK VÁCLAV**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Bulawa David**
Program: Informační technologie
Název: **Mobilní aplikace pro podporu vzdáleného měření s využitím komunikační sítě Sigfox**
Mobile Application for Remote Measurement Support Using Sigfox Communication Network
Kategorie: Uživatelská rozhraní

Zadání:

1. Seznamte se se sítí Sigfox a rozhraním, které Sigfox nabízí pro zobrazování dat odeslaných zařízeními v rámci této sítě.
2. Navrhněte aplikaci pro OS Android, která umožní vyčítání dat přes API sítě Sigfox a vhodným způsobem zobrazí uživateli data odeslaná ze vzdálených zařízení.
3. Vytvořený návrh aplikace doplňte o možnost načítání zobrazovaných dat ve Vámi zvoleném formátu i lokálně přes rozhraní BLE (Bluetooth Low-Energy).
4. Implementujte vámi navrženou aplikaci dle bodů 2) a 3) zadání. Ponechte uživateli možnost vybrat různou interpretaci dat pro každé zařízení.
5. V aplikace rovněž implementujte vhodně zvolené statistické funkce nad daty získanými ze Sigfox zařízení.
6. Důkladně otestujte vámi implementovanou aplikaci v reálných podmínkách.
7. Zhodnoťte dosažené výsledky a pokuste se navrhnout případná další vylepšení či rozšíření.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Šimek Václav, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 25. října 2019

Abstrakt

Cílem této práce je implementovat mobilní aplikaci pro operační systém Android. Aplikace podporuje vzdálené měření přes síť Sigfox a lokální měření přes rozhraní Bluetooth Low Energy. Implementoval jsem ji nativně v programovacím jazyce Kotlin s použitím návrhového vzoru Model-View-Viewmodel. Získávání naměřených hodnot probíhá vzdáleně pomocí Sigfox REST API a lokálně s pomocí Android knihovny pro Bluetooth . Sesbíraná data jsou ukládána v lokální databázi na mobilním zařízení. Dále jsou data interpretována a vizualizována do grafů, které si uživatel může přizpůsobit. Vytvořené řešení poskytuje uživateli možnost získat, sledovat a vizualizovat data ze senzorů včetně jejich historických hodnot. Aplikace má jednoduché použití, současně však umožňuje konfiguraci interpretace dat a podporuje několik různých formátů příchozích zpráv.

Abstract

The purpose of this project is the implementation of a mobile application for the operation system Android. The application supports remote metering via the Sigfox network and local metering via Bluetooth Low Energy. I implemented it natively in the Kotlin programming language using the Model-View-ViewModel design pattern. Measured values are collected remotely using the Sigfox REST API and locally using the Android Bluetooth library. The collected data are persisted in the local database on the mobile device. Then, the data is interpreted and visualized into graphs that the user can customize. The created solution provides a possibility to acquire, monitor and visualize data from sensors, including their historical values. The application is easy to use, but also allows configuring data interpretation and supports several different incoming message formats.

Klíčová slova

Internet věcí, IoT, Sigfox, Bluetooth Low Energy, inteligentní měření, chytrá domácnost, Android

Keywords

Internet of Things, IoT, Sigfox, Bluetooth Low Energy, smart metering, smart home, Android

Citace

BULAWA, David. *Mobilní aplikace pro podporu vzdáleného měření s využitím komunikační sítě Sigfox*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Šimek Václav

Mobilní aplikace pro podporu vzdáleného měření s využitím komunikační sítě Sigfox

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Václava Šimka. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Bulawa
30. července 2020

Poděkování

Chtěl bych poděkovat panu Ing. Václavovi Šimkovi za vedení práce a odborné konzultace. Dále panu Ing. Janu Nevoralovi a panu Ing. Viktorovi Obrovi za technické konzultace.

Obsah

Kapitola 1

Úvod

Tato práce popisuje tvorbu aplikace pro mobilní telefon s operačním systémem Android určenou pro lokální i vzdálené chytré měření dat. V běžném životě neustále sledujeme a měříme nějaké hodnoty. Ať už se jedná o hodnoty z vodoměru, plynoměru, elektroměru či venkovní teplotu, je šikovné mít všechny tyto údaje na jednom místě s možností si je odkudkoli přehledně zobrazit. Práce nastíní aktuální situaci ve světě chytrého měření, které se pomalu stává součástí nejen komerčních prostor, ale i běžných rodinných domů nebo bytů. Přeci jen zase lidstvu o něco usnadňuje rutinní činnosti. Také díky němu můžeme lehce najít nepožadované výkyvy a zmírnit tak spotřebu energií.

Smyslem práce není tvořit něco co již existuje, ale inspirovat se a posunout toto téma, alespoň o trochu dál a třeba tím i přispět ke stále více popularizovanému internetu věcí. Sám jsem se tímto odvětvím informačních technologií v praxi již zabýval. Vidím v něm potenciál a věřím, že se zanedlouho stane běžnou součástí života každého z nás. Další mou motivací byla tvorba mobilní aplikace, kterou jsem chtěl vždy vytvořit, ale neměl jsem na to dostatek prostoru.

Cílem práce je navrhnout, vytvořit a otestovat mobilní aplikaci. Ta by měla být jednoduchá a přehledná i pro uživatele, kteří se tímto tématem až tolik nezabývají. Na druhou stranu se nepředpokládá, že by aplikaci používali lidé s minimálními technickými znalostmi a i na to jsem při vývoji myslel. Přehlednost a jednoduchost však nesmí zastínit obecnost řešení, která je při práci s velkým množstvím různých veličin velmi důležitá. Samozřejmě, že nasbírané hodnoty je nutné uložit na bezpečné místo, aby si je uživatel mohl kdykoli znovu zobrazit.

V následující kapitole vás čeká několik informací o chytrém měření, internetu věcí a popis aktuálního stavu, ve kterém se tyto odvětví nachází. Také popis používaných komunikačních protokolů a technologií. Především technologií Bluetooth Low Energy a Sigfox, které tvoří komunikační kanál mezi senzorem a mobilním zařízením v této aplikaci. V kapitole 3 se nachází informace o operačním systému Android a vývoji aplikací pro něj. Jsou zde zmíněny aktuální možnosti vývoje, architektura operačního systému i nejčastěji používaná architektura aplikací. V kapitole 4 se podíváme, jak probíhal návrh aplikace, probereme existující řešení a rozebereme si použité nástroje a knihovny. Kapitola 5 se zabývá popisem implementačních detailů a konkrétních použitých postupů, včetně ukázek ze zdrojových kódů. Také se zde nachází návrhy možných rozšíření. V předposlední kapitole 6 je zmíněno testování aplikace automatickými testy i akceptační testování v reálných podmínkách. To vše je zakončeno závěrem.

Kapitola 2

Aktuální situace ve světě internetu věcí

Díky rychlému technologickému vývoji v oblasti komunikačních technologií se stal internet součástí běžného života téměř každého z nás. Začal pronikat do světa běžných zařízení, které se bez přístupu ke globálnímu komunikačnímu kanálu zatím obešly. Zvláště pak za poslední roky se stal pojem internet věcí velice často používán nejen v odborných publikacích. To vzbudilo zájem široké veřejnosti o toto téma. Jak to tak už bývá, kde je vysoká poptávka, přibývá i nabídka. Spousty firem, které se často do té doby až tolik nezabývali oborem informačních a komunikačních technologií se nyní začali o toto téma zajímat.

Termín internet věcí poprvé použil v roce 1999 Kevin Ashton. Na světlo světa se tento termín dostal, až v období kolem roku 2009. Od tohoto roku stále roste na popularitě. V roce 2017 bylo k internetu připojeno něco kolem 25 miliard zařízení, což znamená více než 3 zařízení každodenně komunikující přes internet na 1 osobu. [11]

Mnoho lidí si myslí, že se jedná o nějaké ovládání světel, žaluzií pomocí chytrého telefonu. V zásadě mají pravdu, ale je to jen malá množina služeb, které internet věcí pokrývá. Jedná se o velké množství vzájemně komunikujících senzorů a aktivních článků mezi sebou a směrem k uživateli. Díky tomu mohou moderní prostory vybavené "chytrými" prvky sami řídit klimatizaci, vytápění, zabezpečení pomocí inteligentních zámku a mnoho dalšího.

2.1 Oblast vzdáleného měření

Oblast s názvem chytré měření či inteligentní měření můžeme brát jako podmnožinu internetu věcí. Je to způsob měření, zaznamenávání a uchovávání dat s využitím technických komunikačních kanálů. Díky tomu si můžeme zaznamenané hodnoty zobrazit na jednom místě odkudkoli na světě. Pojem bývá nejčastěji využíván v souvislosti s měřením spotřeby energií. Není však pravidlem, že by nutně musel být spjat pouze s energiemi. Díky chytrému měření je možné analyzovat spotřebu energií a tím získat informace, podle kterých můžeme eliminovat plýtvání a zmírnit spotřebu.

Méně často je vidět, že by se do tohoto tématu zařazovalo například měření teploty, vlhkosti vzduchu a podobných hodnot. Ovšem stále platí, že tento typ měření bývá nedílnou součástí internetu věcí a je po něm velká poptávka. V oblasti automatizace budov se měření teploty často využívá ve spojení s termostatem. V kancelářských objektech se rozšiřuje používání senzorů pro monitorování hodnot oxidu uhličitého. Hojně se též využívá měření intenzity osvětlení.

2.2 Architektura přenosu dat

K internetu se začalo připojovat stále více menších a menších zařízení, na které bylo nutné implementovat sadu protokolů TCP/IP. Bylo jasné, že tyto zařízení nebudou využívat většinu protokolů, které se běžně používají při komunikaci osobních počítačů. Propojení dalších prvků s internetem tedy otevřelo brány novým možnostem použití tohoto média. Začali se na světlo světa dostávat nové komunikační vzory a protokoly. Často byli vymyšleny už před popularizací internetu věcí, ale nenašli na aktuálním trhu takové využití. Až po rozmachu chytrých technologií se dočkali svého. Ukážeme si několik komunikačních vzorů i protokolů používaných v tomto oboru, které jsou důležité pro pochopení komunikačních rozhraní této aplikace. Kromě slovního popisu je architektura i graficky popsána obrázkem 2.1.

Device-to-Device vzor komunikace

Velmi často potřebuje zařízení informovat jiné zařízení o změně svého stavu. Informované zařízení na to může zareagovat. Mějme příklad, kdy teplotní senzor při poklesu teploty vody v bazénu pod 24°C notifikuje zařízení řídící ohřev vody, aby začalo vodu ohřívat. Takových scénářů je spousta, proto neexistuje jeden osvědčený postup. Můžeme se, ale podívat na dva nejpoužívanější scénáře.

- Komunikace probíhá na přímo mezi zařízeními.
- V síti se nachází řídící prvek, který přeposílá zprávy zařízením.

Přímá komunikace mezi zařízeními méně zatěžuje komunikační kanál, což je při stále rostoucím počtu chytrých prvků více než vhodné [16]. Dále také vyšší rychlost přenosu, která může být v některých systémech velmi klíčová. V síti s větším počtem prvků nastává problém s orchestrací komunikace. Ke správné orchestraci napomůže do provozu sítě zařadit již zmíněný řídící prvek, který řídí provoz sítě uvnitř mezi zařízeními, ale i směrem ven do internetu. Většinou se můžete setkat s pojmem **gateway** či **base station**.

Device-to-Cloud vzor komunikace

Tento vzor ukazuje komunikaci chytrých prvků s cloudovým serverem. Zařízení odesílají na server naměřené hodnoty. Ten hodnoty uchovává pro jejich další použití. V podstatě je to takové "sběrné místo" dat. Někdy nad daty umožňuje provádět výpočty a statistické operace. Při komunikaci směrem k zařízením se pak cloudový server může chovat jako řídící prvek, přes který zprávy prochází.

Mějme příklad, kde potřebujeme sepnout nějaký aktivní prvek, například rozsvítit osvětlení. Pošleme tedy na cloudový server HTTP[1] požadavek s identifikátorem zařízení, které řídí přepínání osvětlení. Cloudový server požadavek zpracuje a odesílá zprávu na zařízení. Pokud zpráva úspěšně doputovala, tak se provede příslušný úkon (rozsvícení osvětlení). Většinou po vykonání úkonu zařízení ještě notifikuje cloudový server, že úkon byl vykonán.

Cloudové platformy často nabízí i aplikaci, přes kterou má uživatel snadný přístup k datům. Pokud ne, mají k dispozici alespoň **webové API**[2], které přístup umožňuje. Webové API se dá velmi dobře využít při integraci vlastní aplikace na již existující cloudové řešení. Typickým formátem pro přenos dat bývá JSON. Jak může vypadat komunikace s API, je ukázáno v následujícím výpise 2.1.

[1] Hypertext Transfer Protocol - internetový protokol určený pro komunikaci s webovými servery.
[2] Application Programming Interface - rozhraní pro komunikaci s aplikací

```
POST /api/1/devices/messages
Host: dashboard.hologram.io

{
  "deviceids": [
    1234,
    1236
  ],
  "protocol": "TCP",
  "port": 80,
  "data": "Hello world!",
  "base64data": "SGVsbG8gd29ybGQhCg=="
}
```

Výpis 2.1: HTTP požadavek pro zapsání hodnoty Hello world! na 2 chytré zařízení s využitím cloudového serveru Hologram.io.[4]

Často organizace spravující server také prodává inteligentní prvky, v takovém případě se celá komunikace stává interní a nevzniká potřeba interoperability. Na trhu se objevují i organizace jež nabízí integrační platformu, která je schopná komunikovat s prvky třetích stran. [10]

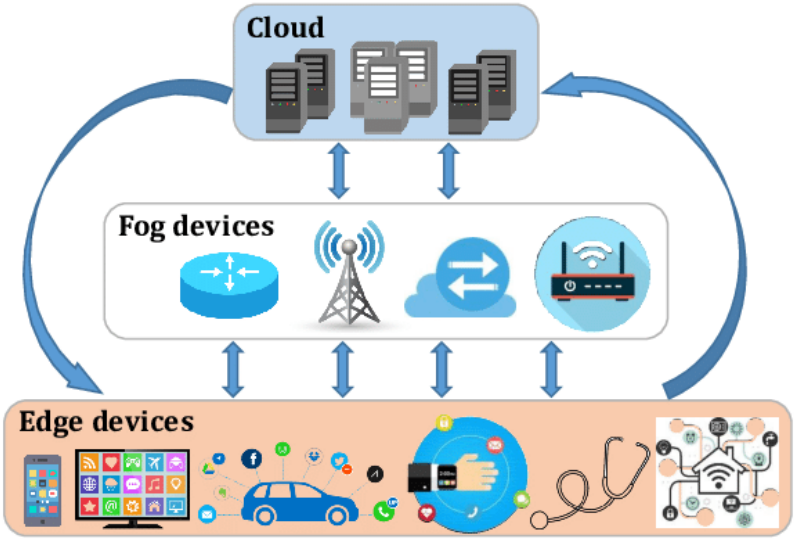
Device-to-Gateway vzor komunikace

Poslední vzor ukazuje způsob komunikace mezi chytrými prvky a bránou, která se obvykle nachází ve stejné lokální síti. V odborných publikacích se o takových zařízeních píše jako o **Fog devices**. Bývají velmi často od stejného výrobce jako senzory či aktivní články, které řídí. Přesto se jich najde několik, které umí integrovat komunikační technologie více výrobců. Tento komunikační vzor lze často nalézt u nasazení inteligentních objektů v systémech, které vyžadují možnosti vzdálené konfigurace a interakce v reálném čase[10].

Pod pojmem brána si někteří představí router. Většinou i zde se jedná o fyzické zařízení routeru podobné. Není to však pravidlem. Může plnit roli brány například mobilní telefon, který zároveň slouží jako klient pro čtení nebo zapisování hodnot. V tomto případě mobilní telefon neslouží jako řídící prvek, ale pouze jako prostředník mezi zařízeními a cloudem. Hlavní výhodou mobilního telefonu je jednoduchý způsob aktualizace. Systém, ale zcela postrádá možnost orchestrace jednotlivých zařízení. Ve většině rozsáhlejších a automatizovanějších systémů je proto nutné přistoupit k sofistikovanějšímu řešení pomocí fyzického zařízení, které data synchronizuje s cloudem automaticky a zároveň funguje i jako interní řídící prvek při posílání zpráv mezi zařízeními.

Brána komunikuje s koncovými "chytrými" prvky pomocí bezdrátových i drátových technologií. Využívá se například WiFi(IEEE 802.11), která bývá součástí prostor většinou ještě před nasazením chytrých technologií, což značně ulehčí jejich montáž. Někdy se využívá Bluetooth Low Energy, zvláště pak při použití mobilního telefonu jako klienta pro čtení a zápis hodnot nebo jej využíváme jako bránu. Mezi další známé bezdrátové technologie patří ZigBe nebo české IQRF. I mezi drátovými technologiemi je spousta technologií, využívá se Ethernet. Dále technologie ModBus nebo M-Bus, které nabízí drátové i bezdrátové varianty, se používají spíše v průmyslových a rozsáhlých implementacích. M-Bus se specializuje na komunikaci v oblasti chytrého měření. Pracuje na principu sériového asynchronního přenosu po dvouvodičové sběrnici.

[4] Převzato z hologram.io



Obrázek 2.1: **Vrstvy komunikace v prostředí internetu věcí.** [5]

V praxi se využívá kombinace těchto scénářů. Asi úplně nejčastějším případem užití bývá spojení zařízení s bránou, která zpracovává a přeposílá požadavky jednak interní v rámci lokální sítě mezi zařízeními, ale také řeší synchronizaci s cloudovým serverem.

2.3 Neznámější komunikační protokoly

V této sekci si nastíníme ty nejznámější aplikační protokoly, které se v internetu věcí používají.

MQTT

Message Queue Telemetry Transport je asi nejznámější prokol v souvislosti s internetem věcí. Je to protokol aplikační vrstvy určený pro omezená zařízení, vyvinutý společností IBM a standardizovaný v OASIS[14]. Využívá principu **publish/subscribe** 2.2.

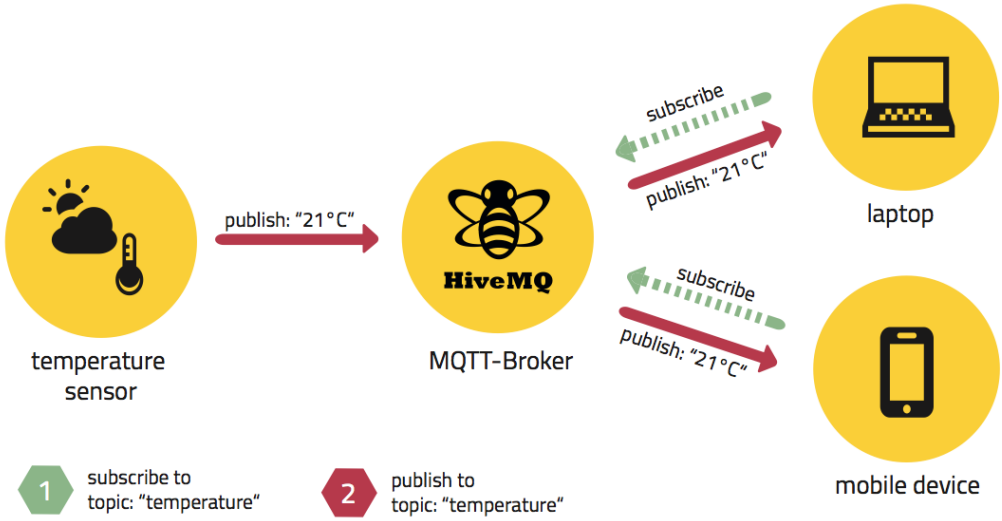
V síti, kde probíhá komunikace pomocí protokolu MQTT vždy figuruje zařízení zvané **broker**. Tento broker řídí přenášení zpráv mezi jednotlivými zařízeními. V takové síti se kromě brokeru nachází 2 typy klientů.

- Odběratel(subscriber)
- Vydavatel(publisher)

Odběratel si u brokeru zaregistruje **témata(topics)**, které chce odebírat. Po registraci bude odběratel dostávat všechny zprávy se zaregistrovanými tématy, které na broker přijdou. Vydavatel pod vybraným tématem posílá zprávy na broker. Broker poté přeposílá zprávy dál, všem odběratelům tohoto tématu. Jedno zařízení může figurovat zároveň jako vydavatel i odběratel.

Mějme příklad, z podsekce o komunikačním vzoru Device-to-Device 2.2 o teplotním senzoru a ohřevu vody. Zde by například teplotní senzor hrál roli vydavatele a zařízení řídící ohřev vody roli odběratele. Při poklesu teploty by teplotní senzor poslal zprávu na

broker pod tématem například `pool_temperature_drop`. Broker by zprávu přeposlal řídící jednotce ohřevu vody, která toto téma odebírá.



Obrázek 2.2: **Ukázka komunikace typu publish/subscribe v MQTT-Brokeru.** [8]

Další klíčovou vlastností pro MQTT je QoS[5]. Prokol definuje tři úrovně kvality.

- **QoS level 0 (nejvýše jednou)** - Při použití této úrovně není garantované, že zpráva opravdu dorazila kam měla. Příjemce nepotvrzuje přijetí zprávy. Tato úroveň je také známa jako "fire and forget". Po odeslání se na zprávu zapomene, jelikož ji odesílatel nijak neuchovává pro další použití. Výhodou je nízká režie. [14]
- **QoS level 1 (alespoň jednou)** - V tomto případě je garantované, alespoň jedno úspěšné doručení zprávy. Odesílatel si zprávu po prvním odeslání uchová a opakuje odesílání, dokud mu nedorazí potvrzení od příjemce, že zpráva byla doručena. [14]
- **QoS level 2 (právě jednou)**- Jedná se o nejvyšší úroveň zajištění spolehlivosti přenosu. Zde je garantované, že k příjemci zpráva doputuje právě jednou. Vhodný například pro platební systémy, kde si nemůžeme dovolit ztrátu zprávy. [14]

Povšimněte si, že se zde nabízí možnost použít broker i jako bránu. I to je jeden z důvodů, proč je tento protokol v oboru internetu věcí tak populární.

CoAp

Constrained Application Protocol byl vyvinut společností IETF Constrained RESTful Environments (CoRE)[14]. Je založen na principu **dotaz/odpověď**, který přebral od protokolu HTTP. Na rozdíl od protokolu HTTP pracuje na UDP. Jelikož je oproštěn od režie, kterou způsobuje transportní přenos přes TCP, je více vhodný pro použití v zařízeních s omezeným výkonem a pamětí. Protože je UDP přenos nespolehlivý, protokol používá mechanismy na aplikační vrstvě pro zajištění spolehlivosti.

[5]Quality of Service - zajištění spolehlivosti přenosu

Typy zpráv v protokolu CoAp:

- **Confirmable (CON)** - Typ zprávy, který vyžaduje potvrzení (ACK). Odpověď na tuto zprávu může být poslána společně s potvrzením, ale nemusí. [14]
- **Non-Confirmable (NON)** - Zpráva s tímto typem nevyžaduje potvrzení[14].
- **Acknowledgement (ACK)** - Potvrzení zprávy typu Confirmable[14].
- **Reset (RST)** - Posílá se jako odpověď na zprávu, která nemůže být zpracována[14].

AMQP

Advanced Message Queuing Protocol je otevřený standard určený pro message-oriented middleware[6]. Pracuje také na principu **publish/subscribe**. Jeho použití nalezneme spíše při koordinaci modulů v cloudu.

2.4 Sigfox

Sigfox je francouzská společnost založená v roce 2010[2]. Zabývá se stavěním bezdrátových sítí globálních rozměrů pro nízkoenergetická zařízení, která přenáší malé množství dat. Často se název Sigfox používá v souvislosti s technologií, na které jsou bezdrátové sítě postaveny. V České Republice sítě s touto technologií buduje společnost SimpleCell Networks ve spolupráci s T-mobile.

2.4.1 Pokrytí signálem

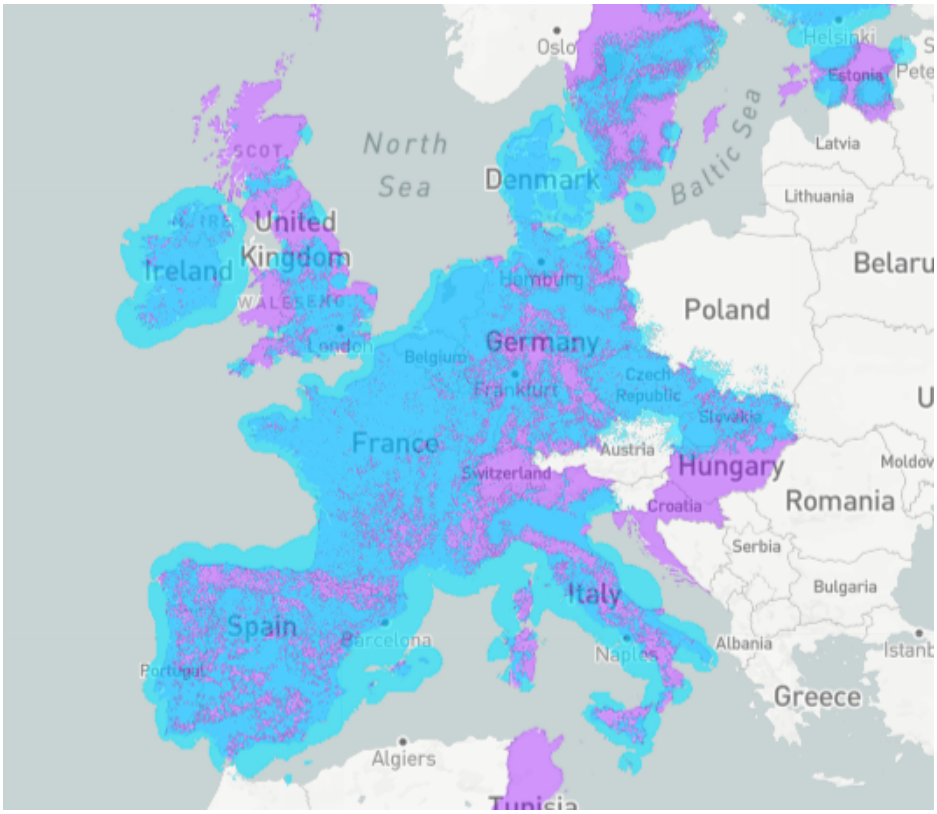
Technologie Sigfox se pomalu dostává do celého světa. Od roku 2010 je k jejich síti připojeno již **70 zemí světa**. Společnost se tak stává dobrým kandidátem na postavení největší globální sítě pro internet věcí[2]. Je partnerem pro celostátní operátory i malé startupy a díky tomu dokáže velmi rychle pronikat do dalších zemí. Podpora od operátorů zajišťuje široké pokrytí. Síť momentálně pokrývá **96 % české populace** a zasahuje i do míst bez dosahu GSM[7] signálu mobilních operátorů[15]. Co se týče ostatních států, má Sigfox největší pokrytí v západní Evropě 2.3. Dobré pokrytí najdeme také například v Japonsku, Malajsii či v Jihoafrické republice. V Severní Americe je pokrytí pouze ve velkých městech.

Společnost sama vysílače nestaví, ale využívá pokrytí operátorů v daných zemích. Jejich vysílače poté používá jako tzv. **base stations(základny)**. Jedna taková stanice dokáže pokrýt okruh až 50 kilometrů ve volné krajině a 10 kilometrů v městské zástavbě. Dokážete si tedy představit, že lze s jedním dobře umístěným vysílačem pokrýt celé větší město. Například v Belgii dokázali pokrýt všech 30 500 kilometrů čtverečních pouze sedmi stanicemi [17]. Musíme samozřejmě počítat s tím, že pokud budou zařízení připojené k nejbližšímu vysílači na hraně signálu, může dojít k problémům při přenosu.

2.4.2 Technologie v pozadí sítě

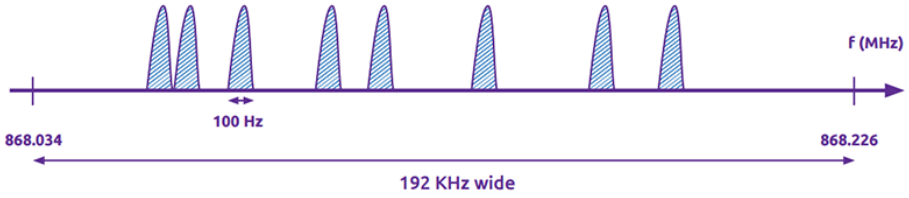
Pracuje na rozshlasovém pásmu ISM[8]. Vysílá na frekvenci **868 MHz v Evropě a 902 MHz v USA**. V pásmu sice běží nelicencované krátkodosahové technologie, ale Sigfox se jim vyhýbá. Nezasahuje do pásem nejpoužívanějších technologií jako Wifi či Bluetooth.

[6] Prostředí, jež používá posílání zpráv jako komunikaci v distribuovaných systémech.
[7] Groupe Spécial Mobile - telekomunikační standard pro digitální mobilní sítě
[8] ISM (industrial, scientific and medical) - bezlicenční pásmo pro rádiové vysílání.



Obrázek 2.3: **Pokrytí sítě Sigfox v západní Evropě.** [17]

K přenosu Sigfox komunikace se využívá **velmi úzké pásmo** pro vyslání jen krátkého pulsu dat s vysílacím výkonem omezeným na 100 mW a modulací pracující v 200kHz veřejném pásmu, jak je znázorněno v obrázku 2.4. Každá vyslaná zpráva pak zabírá šířku pásma 100 Hz a je šířena rychlostí 100 nebo 600 bitů/s (záleží na regionu). Vysílaná zpráva využívá tzv. DBPSK modulaci, které stačí k dosažení rychlosti přenosu 1 bit/s pásmo o frekvenci 1 Hz. Díky velmi úzkému pásmu a modulaci typu DBPSK, která využívá klíčování frekvenčním posuvem, dosahuje síť vysoké přenosové kapacity a přenos je spolehlivý i na velké vzdálenosti. [18].



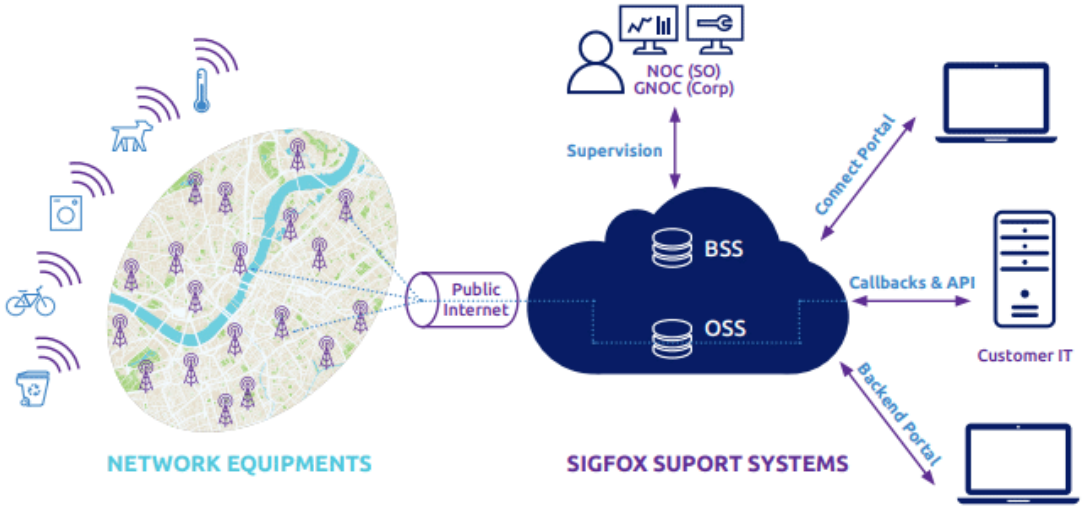
Obrázek 2.4: **Ukázka velmi úzkých pulsů při vysílání.** [7]

Sigfox umí komunikovat v obou směrech, přesto se specializuje spíše na uplink komunikaci, což je komunikace směřem od zařízení ke cloudu.

2.4.3 Architektura sítě

Samotná architektura by se dala rozdělit na 2 vrstvy. V následující sekci je popsána jak slovně, tak obrázkem 2.5.

- **Vrstva síťového vybavení** - Skládá se z base stations a zařízení k nim připojovaných. Úkolem base station je přijímat zprávy od zařízení, převést je do digitální podoby a přeposílat je vrstvě systému podpory.
- **Systém podpory Sigfox** - Přijímá zprávy, zpracovává je a uchovává v cloudovém úložišti. Případně pomocí callbacku přeposílá zprávy do zákaznických systémů třetích stran. Vystavuje webové aplikační rozhraní pro snadné interakce se systémem. [4]



Obrázek 2.5: **Kompletní vizualizace architektury Sigfox.** [4]

Vrstva síťového vybavení

Architektura sítě je založena na **topologii hvězda**. Jednotlivé zařízení odesílají zprávy přímo k base station. Ta tedy tvoří jakýsi centrální prvek této topologie. Pro odeslaní se využívají technologie zmíněné v předchozí sekci. Zprávy, které base station přijme jsou zpracovány a za pomoci demodulace převedeny zpět do digitální podoby. Stanice využívají připojení místních operátorů k internetu. Většinou používají DSL[9] v kombinaci s 3G nebo 4G jako zálohu. Díky tomu je možné zprávu dále přeposlat do systému podpory Sigfox přes TCP/IP internetovou komunikaci.

Limity pro komunikaci v síti Sigfox:

- Uplink komunikace je omezena maximální velikostí odeslané hodnoty na 12 bytů za 1 zprávu.

[9]Digital Subscriber Line - komunikační médium používané k přenosu digitálních signálů přes telefonní linky

- Downlink komunikace je stejným způsobem omezena maximální velikostí 8 bytů na 1 zprávu.
- Maximálně lze odeslat přes síť 144 zpráv za 1 den na 1 zařízení.

Systém podpory Sigfox

Má za úkol shromažďovat data a poskytnout k nim jednoduchý přístup. Stává se, že stejná zpráva je systému předána vícekrát, proto při vstupu do systému prochází zprávy filtrem, který odstraní repliky, aby mohl následně uložit jen 1 kopii dané zprávy [4]. Zpráva je poté distribuována pomocí callbacků směrem k zákaznickým aplikacím. Systém, ale zprostředkovává mnohem více, než jen rozhraní k přijatým zprávám. Nabízí jednotný přístup ke službám a webovým aplikacím v rámci celé platformy nejen pro zákazníky.

Nejpoužívanější moduly platformy.

- Sigfox partner - Nabízí zařízení či kompletní řešení od partnerů. Také popisuje postup, jak se vaše společnost může stát partnerem.
- Buy Sigfox - Předplacení a aktivace konektivity.
- Sigfox portal - Portál pro registrované uživatele. Správa zařízení, uživatelů či možnost vygenerování přístupových údaju k API.
- Sigfox support - Návody a dokumentace.
- Sigfox build - Technická specifikace. Postupy jak integrovat technologii Sigfox.
- Ask Sigfox - Komunitní fórum.

2.4.4 Přístup k datům

Všechny zprávy, které od zařízení přicházejí na platformu Sigfox, jsou ukládány do cloudového úložiště. K uloženým zprávám je možné se dostat za pomoci REST[10] API, callbacků nebo přímo přes webové grafické rozhraní, které je k dispozici v Sigfox portálu. Pro integrování aplikace třetí strany na Sigfox cloud je grafické rozhraní nepoužitelné, zbývají nám tedy 2 možnosti.

REST API

Za pomoci HTTP volání je možné ovládat většinu věcí, které platforma nabízí. Podporuje 4 základní typy HTTP požadavků (POST, GET, DELETE, PUT). Pro vstupní požadavky i odpovědi používá typ média `application/json`, který je lehce zpracovatelný v jakémkoli moderním programovacím jazyce. Volání je zabezpečeno **basic autentizací**. Přístupové údaje k API je nutné vygenerovat na Sigfox portále. Jedná se o dvojici login a heslo. Tyto údaje je nutné vložit do HTTP Authorization hlavičky ve formátu "login:heslo" a zakódovat je pomocí Base64 kódování. Jinak vám bude přístup k API odmítnut a na vaše volání dostanete odpověď ve formě HTTP statusu 401 Unauthorized.

[10] Representational state transfer - architektura webového rozhraní pro snadné čtení, vytváření, editování nebo mazání dat ze serveru za pomoci HTTP.

Co se týče dotazů na zprávy ze zařízení, tak jste skrze REST API schopen získat zprávy staré nejvýše 3 dny, starší jsou k dispozici pouze přes webové GUI. Pokud tedy chcete ve své aplikaci pracovat s historickými hodnotami vašich senzorů, je nutné zprávy ukládat ve vlastní databázi a přes API získávat pouze aktuální data. Seznam všech HTTP požadavků, které je možné na API směrovat je dostupný v oficiální dokumentaci. Naleznete zde i OpenAPI[11] specifikaci v JSON[12] struktuře.

Callback

Druhá možnost jak získat data ze Sigfox cloudu do svojí aplikace. Jedná se o HTTP požadavek od Sigfox cloudu na předem zvolený endpoint. V těle požadavku se nachází údaje, podle typu callbacku. Callback je spuštěn ve chvíli, kdy na cloudové úložiště dostaneme novou zprávu od zařízení, polohu zařízení nebo byla detekována ztráta komunikace, to vše záleží na zvoleném typu[1]. Je možné nadefinovat si vlastní callback nebo využít již definované callbacky na jednu z 5 platforem, které Sigfox integruje[1].

- AWS IoT
- AWS Kinesis
- Microsoft Azure Event HUB
- Microsoft Azure IoT HUB
- IBM Watson Iot

Při definování vlastního callbacku je možné využít tyto typy:

- **Data** - Přeposílá zprávy přijaté od zařízení. Podporuje i downlink komunikaci pro potvrzení přijetí zprávy systémem, na který je požadavek odeslán.
- **Error** - Přeposílá chybové zprávy při závadě v komunikaci.
- **Service** - Posílá dodatečné informace. Například stav baterie, geolokaci nebo potvrzení od zařízení o přijetí downlink zprávy.

2.5 Bluetooth Low Energy

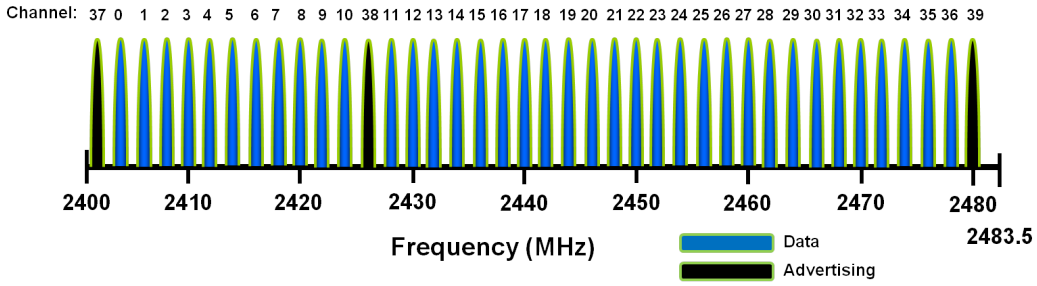
Bluetooth na přelomu 20. a 21. století přišel na trh jako náhrada za stávající kabelové řešení. Stal se velice populární především ve světě mobilních a periferních zařízení. Později s příchodem verze 4.0, která vyšla v roce 2010, se objevila nová technologie nesoucí název Bluetooth Smart. Měla se specializovat na zařízení, pro které je kritická spotřeba baterie a přenášejí malé množství dat[6]. Bylo tedy jasné, že nebude sloužit k většině účelů, ke kterým sloužilo klasické Bluetooth. Avšak mělo velký potenciál se uchytit s nástupem internetu věcí. Později bylo přejmenováno na Bluetooth Low Energy(dále jen "BLE").

Dnes tedy rozlišujeme technologie Bluetooth classic a BLE. Technologie jsou navzdory stejnému výrobci k sobě nekompatibilní. Zařízení pracující s jednou technologií přímo nepřipojíme k zařízení pracující na druhé. Jedinou možností jak propojení uskutečnit, je použít jiné zařízení, které ovládá obě technologie, jako "prostředníka". Takových zařízení je dnes již spousta. Zejména toto umožňují počítače nebo mobilní telefony.

[11] OpenAPI - formát popisu REST API
[12] JavaScript Object Notation - způsob zápisu dat určených pro přenos

2.5.1 Technologické detaily protokolu

Pracuje v bezlicenčním ISM pásmu na **frekvenci 2.400 - 2.4835 GHz**. Frekvenční rozsah se může lišit v některých regionech. Frekvenční pásmo, na kterém BLE operuje, je rozděleno do 40 kanálu o šířce pásma 2 MHz. Rozdělení je možné pozorovat na následujícím obrázku 2.6.



Obrázek 2.6: **Ukázka rozdělení kanálů v BLE**.[13]

Spotřeba enegie se nedá přesně vyjádřit, jelikož se hodně liší podle zařízení a softwaru, který na něm běží. Špičková spotřeba v době přenosu běžně nepřesahuje hodnotu 15 mA[6].

Dosah přenosu se také nedá přesně specifikovat, protože je zde příliš mnoho proměnných faktorů, jako je prostředí pro přenos nebo mód, ve kterém komunikace probíhá. V oficiálních publikacích se uvádí i více než 100m. Častokrát, ale zjistíte, že i 15m může být velký problém.

S příchodem Bluetooth verze 5.0 byly představeny **2 módy** fyzické vrstvy protokolu. Módy umožňují komunikaci lépe přizpůsobit specifickým podmínkám, které vyžadujeme pro rychlost a dosah. V předchozích verzích byla maximální rychlost přenosu vždy 1 Mb/s.

- **2M** - Tento mód umožňuje komunikaci s rychlostí až 2 Mb/s, což je dvakrát větší rychlost než byla dosavadní maximální. Toto má pozitivní vliv na spotřebu baterie, jelikož je možné přenést větší objem dat za kratší dobu. To vše za cenu kratšího dosahu přenosu.
- **Coded PHY** - Někdy také označován jako "long-range" mode. Umožňuje komunikaci na vzdálenost více než 800m v ideálních podmínkách. Na úkor toho je, ale omezen rychlostí komunikace na 125 nebo 500 kb/s. Záleží na počtu symbolů, které jsou vyžadovány pro zakódování 1 bitu. Kvůli pomalejší rychlosti je vyšší i spotřeba baterie.

2.5.2 Architektura

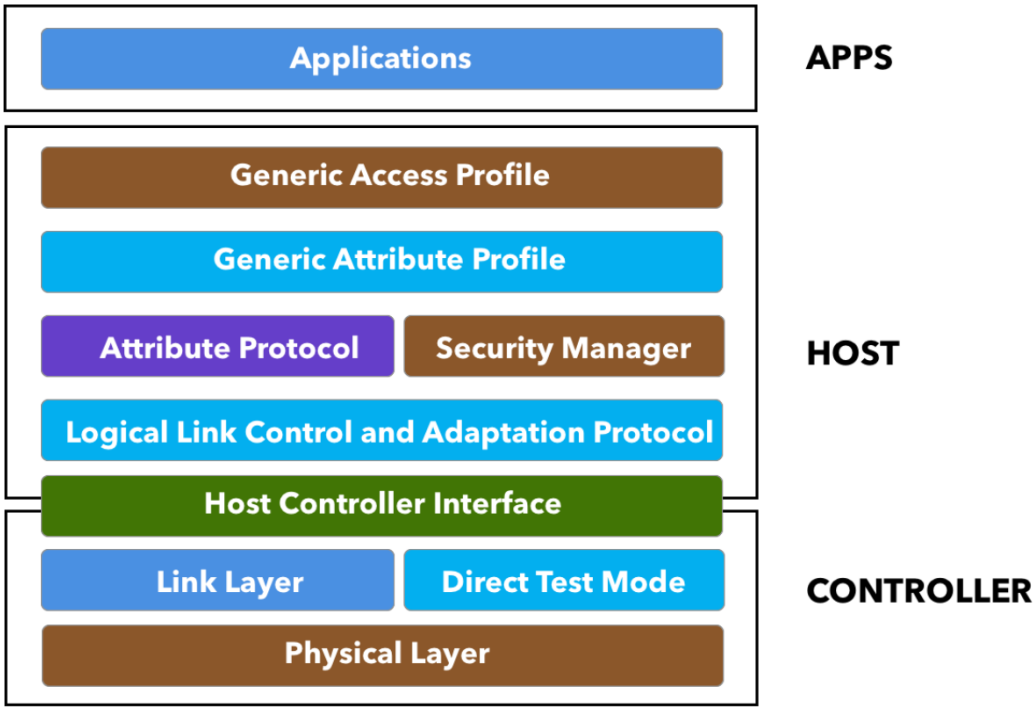
Architektura je rozdělena do 3 hlavních částí, které můžeme vidět na obrázku 2.7. Můžeme se také setkat s názvem "protocol stack", který je možná výstižnější.

- **Applications** - Jedná se o konrétní implementaci aplikace postavené na protokolu BLE. Řekněme, že vyjadřuje způsob, jak aplikace pracuje s přijatými a odesílanými daty. Některé zdroje tuto vrstvu do architektury vůbec nezařazují.
- **Host** - Nachází se pod vrstvou aplikace, se kterou komunikuje skrze aplikační rozhraní. Skládá se z několika síťových protokolů, které značně usnadňují komunikaci s

[13] Převzato z microchipdeveloper.com.

dalšími zařízeními. Při implementaci BLE aplikace nás bude zajímat nejvíce. Postupně si ukážeme ty nejdůležitější protokoly této části.

- **Contoller** - Snaží se odstínit od detailů rádiové komunikace. Kontroluje příjem a vysílání paketů a garantuje spolehlivost přenosu. Principy fyzické vrstvy jsou popsány v sekci 2.5.1.



Obrázek 2.7: **Architektura Bluetooth Low Energy**. [6]

Link Layer

Poskytuje vyšším vrstvám abstraktní rozhraní pro práci s fyzickou vrstvou. Řeší přepínání mezi stavy, ve kterých zařízení operují.

Seznam stavů, ve kterých se BLE zařízení může nacházet:

- **Standby** - Výchozí stav. Zařízení v tomto stavu nepřijímá ani neodesílá žádná data[6].
- **Advertising** - V tomto stavu zařízení posílá tzv. "advertising" pakety a tím se nabízí dalším zařízením, které jej mohou objevit a pracovat s ním.
- **Scanning** - Zařízení objevuje jiné další zařízení v okolí, které se nachází ve stavu Advertising.
- **Initiating** - V tomto stavu se nachází zařízení, které se rozhodne iniciovat připojení.
- **Connected** - Zařízení jsou spojeny a vzájemně si vyměňují data. Tento stav se vždy týká obou propojených uzlů. Zařízení, které připojení iniciovalo se označuje jako **Master** a zařízení, které vysílalo "advertising" pakety se označuje **Slave**.

Aby bylo možné identifikovat každé zařízení, je nutné je adresovat. Bluetooth využívá adresu podobnou MAC o délce 48 bitů. Může být předem definovaná a registrovaná v IEEE[14] nebo náhodně vygenerovaná.

Host Controller Interface (HCI) Layer

Obstarává komunikaci mezi částmi **Host** a **Controller**. Převádí příkazy vyšších vrstev a notifikuje je o změnách stavu. To vše pro zachování nezávislosti obou částí. Díky této vrstvě můžeme každou část spustit na jiném chipsetu a přesto spolu budou moci komunikovat[6].

Logical Link Control and Adaptation Protocol (L2CAP) Layer

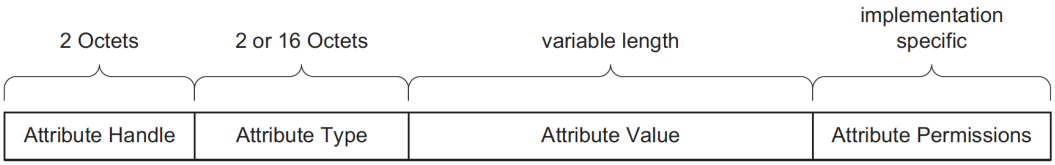
Zapouzdřuje protokoly vyšších vrstev do standardního BLE paketu[6]. Dále má na starost fragmentaci a zpětnou defragmentaci paketů. Obsah BLE zprávy, tzv. payload, nesmí totiž obsahovat více než 27 bytů.

Attribute Protocol

Definuje jakým způsobem server vystavuje data pro klienta a jak jsou strukturována[6].

- **Klient** - Inicializuje spojení na server za účelem číst nebo zapisovat hodnoty. Od serveru akceptuje notifikace a může na ně reagovat. Typicky to bývá mobilní telefon nebo počítač, který se připojuje na zařízení, jako je třeba libovolný senzor, chytré hodinky či jiné zařízení poskytující BLE služby. Není to, ale pravidlem i mobilní telefon či počítač může fungovat v roli serveru.
- **Server** - Vystavuje služby pro získání dat, jež má k dispozici. Případně ovládací prvky, do kterých lze zapisovat hodnoty a tím spouštět akce nebo měnit chování zařízení v roli serveru. Přijímá požadavky od klienta a vhodným způsobem na ně reaguje. Server také může notifikovat klienta o změně stavu.

Serverem poskytovaná data jsou strukturovaná do tzv. **atributů**. Definují pevnou strukturu pro vystavování dat. Skládají se z několika částí. Jak jdou jednotlivé části po sobě je znázorněno v následujícím obrázku 2.8.



Obrázek 2.8: **Složení BLE atributu**. [6]

- **Typ atributu (Attribute type)** - Jedná se o 16 bitové číslo zapsané v hexadecimálním formátu v případě, že použijeme některý ze standartizovaných atributů. Nestandartizované, někdy pojmenované jako "vendor specific", má délku 128 bitů. To proto, že u standartizovaných atributů je zbylých 112 bitů vždy stejných.

[14]IEEE - Mezinárodní institut usilující o vzestup v oblasti elektrotechniky

Identifikuje typ dat a tím je dokáže převést na informace. Například 0x2A1C je standartizovaný typ pro měření teploty. Díky tomu klient ví, že hodnota tohoto atributu bude teplota a může se podle toho zachovat.

- **Attribute Handle** - 16 bitové číslo identifikující atribut. Je přidělené serverem. Ten garantuje jeho unikátnost v rámci aktivního spojení s klientem. [6]
- **Oprávnění atributu (Attribute Permissions)** - Definuje práva, které má klient v souvislosti s daným atributem. Jedná se o práva na čtení, zápis a nastavení notifikací. Délka této části záleží na konkrétní implementaci.
- **Hodnota atributu (Attribute Value)** - Data, které může klient číst nebo je měnit. Délka je proměnná.

Generic Attribute Profile (GATT)

Dodává atributům strukturu a hierarchii. Rozlišuje atributy zvané Services (služby) a Characteristics (charakteristiky).

- **Services (služby)** - Sdružují jeden nebo více atributů. Většinou se jedná o charakteristiky, ale mohou to být i jiné doprovodné atributy. Jako příklad můžeme mít službu "Temperature Service", která obsahuje charakteristiky "Temperature Level" a "Temperature Sensor Location". Standartizované služby nalezneme na officialních webových stránkách Bluetooth SIG[15].
- **Characteristics (charakteristiky)** - Vždy jsou součástí nějaké služby. Obvykle prezentují hodnoty klientovi s oprávněním je číst nebo měnit. Dále mohou obsahovat seznam operací, které může klient s daty provádět a také seznam tzv. Descriptorů, které slouží k popsání charakteristiky. Podobně jako standartizované služby i charakteristiky jsou zveřejněny na officiálních stránkách.

Služby bývají organizovány do tzv. **profilů** (GATT profiles). Ty definují chování jak serveru, tak klienta. Specifikují vyžadované služby a charakteristiky a popisují jakým způsobem by měli být používány. Obvykle popisují i vyžadované zabezpečení. Bluetooth SIG zveřejňuje seznam standartizovaných profilů včetně specifikací na svých webových stránkách. Služby jsou poté lépe implementovatelné pro klienty. Například pro zařízení, které měří krevní tlak můžeme využít profilu "Blood Pressure Profile" a tím standartizovat chování takového senzoru.

Generic Access Profile(GAP) klientovi

Tato vrstva řídí interakci s ostatními BLE zařízeními. Má na startost především:

- Správu rolí a interakcí mezi nimi
- Spuštění skenování okolních zařízení nebo vysílání "advertising" paketů
- Vytvoření spojení - Inicializace a akceptování pokusů o připojení
- Bezpečnost

[15] Bluetooth Special Interest Group - organizace vyvíjející Bluetooth standardy

Mezi používané role patří:

- **Peripheral** - Rozesílá “advertising“ pakety a očekává připojení jiného zařízení (v roli Central).
- **Broadcaster** - Opět rozesílá “advertising“ pakety, ale narozdíl od Peripheral role neakceptuje připojení od jiného zařízení v roli Central. Použitelné například pro vnitřní lokalizační systémy.
- **Observer** - Pouze hledá zařízení, které vysílají “advertising“ pakety. Nepřipojuje se k nim.
- **Central** - Funguje podobně jako Observer, ale dokáže se na zařízení připojit a komunikovat s ním.

“**Advertising state**“ je stav, kdy zařízení vysílá “advertising“ pakety a tím se stává viditelné pro ostatní zařízení. To vše v intervalech začínajících na 20 milisekundách a zvyšujících se po 625 mikrosekundách až k 10 vteřinám. V sekci o technologii BLE bylo zmíněno 40 kanálů, na kterých BLE pracuje. Pro “advertising“ pakety jsou primárně vyhrazeny kanály 37, 38 a 39, jak je znázorněno na obrázku 2.6. Pokud nestačí tyto 3 kanály, mohou být využity i ostatní volné.

Při **skenování** musí zařízení poslouchat na stejném kanálu jako jiné zařízení vysílá “advertising“ pakety, jinak zařízení neuvidí. Řešením je velmi rychlé přepínání mezi třemi primárními kanály pro “advertising“ pakety.

Kapitola 3

Vývoj aplikací pro operační systém Android

Mobilní telefon se stal nedílnou součástí života většiny populace. Přístroj, který byl ze začátku vytvořen jako přenosná verze klasického telefonu, postupně začal nabývat na popularitě a nahrazovat další zařízení jako např. kalkulačka, fotoaparát, diktafon či dokonce osobní počítač. S tím přicházeli i vyšší nároky na hardware a software.

V roce 2005 společnost Google Inc. odkoupila malý startup Android Inc. za 50 milionů dolarů a vytvořil z ní svojí dceřinnou firmu[12]. Po odkoupení začal tým vývojářů vyvíjet platformu pro mobilní telefony, která používala Linuxové jádro. Tímto se začal Google dostávat do světa mobilních technologií. V roce 2008 přišel první telefon s operačním systémem Android. Od té doby Android stoupal raketovou rychlostí vzhůru. O dva roky později se stal nejpoužívanějším operačním systémem pro mobilní zařízení. A v roce 2012 používalo Android až 60% mobilních telefonů.

Jedna z vlastností, která zapříčinila tak rychlý vzestup tohoto operačního systému byla otevřenost software. Také systém stahování aplikačních balíčků a poměrně jednoduchý vývoj aplikací, pro který Google poskytuje spoustu příruček a dokumentací přímo na oficiálních stránkách. Vývoj aplikací značně ulehčuje velká spousta knihoven a aplikačních rozhraní pro práci s komponenty operačního systému nebo přímo s hardware.

3.1 Architektura operačního systému

Oficiální dokumentace rozděluje architekturu na šest základních modulů[3]. Tyto moduly jsou v pořadí, ve kterém jsou rozepsány, vrstveny na sebe.

- **Linux kernel** - Základní kámen tvoří jádro Linuxu. Určitě, ale Android nepatří ke klasickým linuxovým distribucím, jelikož nevyužívá všechny prvky linuxového jádra. Pro výrobce hardware je mnohem snaží vyvíjet ovladač pro tak známé jádro.
- **Hardware Abstraction Layer (HAL)** - Jak již název napovídá, tato vrstva se snaží vystavit sadu abstraktních rozhraní pro interakci s hardwarovými komponenty. S tímto rozhraním poté komunikují vyšší vrstvy architektury, především Java API Framework.
- **Android runtime** - Běhové prostředí Android aplikací spolu s knihovnami jádra.

- **Nativní C/C++ knihovny** - Spousta komponent nižších vrstev je napsána v nativním jazyce, který potřebuje knihovny jazyka C/C++ ke svému běhu.
- **Java API Framework** - Aplikační rozhraní v jazyce Java. Zapouzdřuje funkcionalitu operačního systému do snadno použitelných komponent. Při programování aplikací budete velmi často využívat funkce tohoto frameworku. Díky této vrstvě můžeme pohodlně využívat prvky operačního systému.
- **Systemové aplikace** - Sada zabudovaných aplikací, které se nachází na zařízení s tímto operačním systémem. Např. Kalendář, Email, SMS zprávy, Hovory, Kontakty a spousty dalších. Liší se podle typu zařízení.

3.2 Vývoj mobilních aplikací

V dnešní době si lze při vývoji mobilní aplikace vybrat z více technologií. Určitě je dobré si prostudovat výhody a nevýhody jednotlivých technologií a vybrat si tu nejvhodnější pro vaší aplikaci. Vhodné je si technologii ujasnit před samotným začátkem vývoje, protože se tím ovlivní průběh implementace celé aplikace. Každá technologie používá jiný programovací jazyk a architekturu. Také definuje operační systémy, na kterých aplikace poběží. Krátce si ukážeme nejpoužívanější technologie a frameworky pro vývoj aplikací pro zařízení s operačním systémem Android.

Java/Kotlin

Jedná se o nativní přístup k vývoji aplikace. Google pro vývoj poskytuje spoustu dokumentací a metodik. Největší výhodou je široká podpora knihoven, které velmi usnadňují práci s operačním systémem a hardware. Zároveň aplikace psané tímto způsobem jsou lépe optimalizovány. Nevýhodou je však skutečnost, že aplikace není kompatibilní s jakýmkoli jiným operačním systémem a také lehce zdlouhavější vývoj.

React Native

Multiplatformní framework vyvinutý společností Facebook, Inc. Umožňuje interpretovat JavaScript kód přímo na zařízení. Na rozdíl od knihovny "React", která byla vyvinuta dříve stejnou společností, nepracuje s technologiemi HTML ani CSS, ale umožňuje s určitou mírou abstrakce komunikovat s nativními prvky UI. Největší výhodou je vývoj jedné aplikace, která se dá použít na více platformách, což značně ušetří vývojářům práci. Mezi hlavní nevýhody patří pomalejší běh aplikací v porovnání s nativními technologiemi na jednotlivých platformách. Také menší dostupnost knihoven na jednotlivých platformách a horší dokumentace.

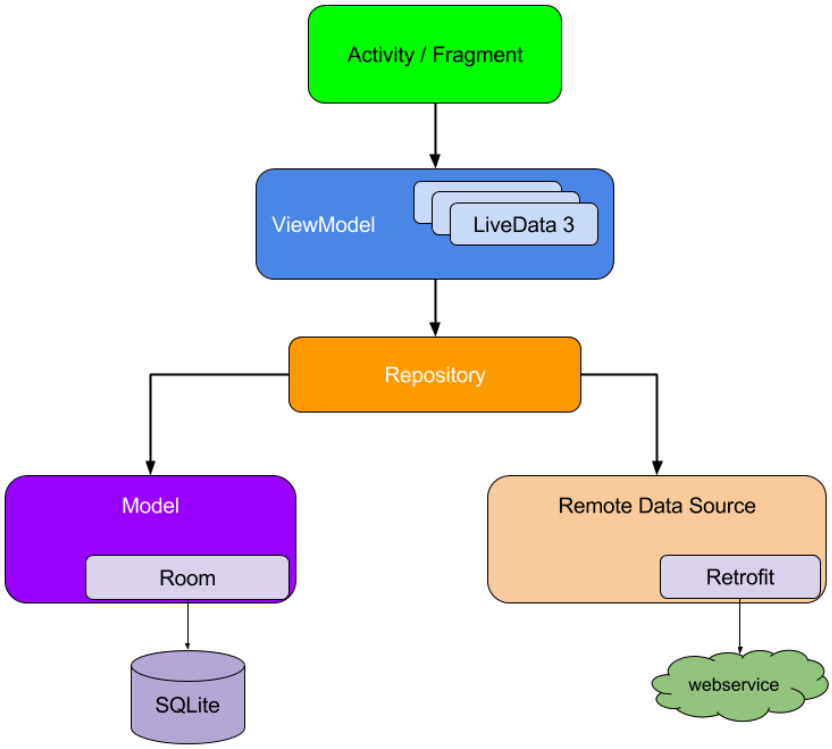
Flutter

Poměrně nový multiplatformní framework od společnosti Google, Inc. Byl vyvinut jako odpověď na trend multiplatformního vývoje. Jelikož je framework poměrně nový, není tolik rozšířený jako např. React Native. S tím souvisí i málo materiálů ke studiu a slabší dokumentace. Framework využívá programovací jazyk Dart, který také není tolik rozšířený. V testech výkonu dosahuje o něco lepších výsledků než React Native.

3.3 Architektura nativních aplikací

Android dlouhá léta nedoporučoval žádnou konkrétní architekturu. Bylo tak na každém, aby si architekturu zvolil jakoukoli chtěl. Mohl použít známé návrhové vzory jako model-view-presenter (MVP), model-view-controller (MVC), model–view–viewmodel (MVVM) či si mohl vytvořit jakoukoli vlastní architekturu. Každá použitá architektura vyžadovala napsání vlastního kódu nebo použití neoficiálních knihoven. [9]

Až v roce 2017 Android představil oficiální architekturu pro aplikace. Je založena na návrhových vzorech MVVM a Repository. Pro její vysvětlení si rozeberme jednotlivé komponenty architektury. A to textově i graficky diagramem 3.1.



Obrázek 3.1: **Doporučovaná architektura Android aplikací.**[1]

- **Activity/Fragment** - Reprezentují vrstvu View. Dynamicky ovládají jednotlivé UI komponenty. Pracují s daty, které jim poskytuje ViewModel a podle nich upravují UI. Nikdy neřeší komplexní operace a nepřistupují přímo k datům, ať už jsou dostupná lokálně nebo vzdáleně.
- **ViewModel** - Na požadavky View vrstvy přeposílá data obousměrně mezi View a Repository vrstvou. Provádí nad daty potřebné operace a předává je View vrstvě nejčastěji jako tzv. LiveData 3.3.
- **Repository** - Vystavuje jednoznačné rozhraní s určitou mírou abstrakce pro práci s daty. Typicky obsahuje alespoň funkce zprostředkovávající základní operace jako:

[1] Převzato z https://developer.android.com/jetpack/.

Create, Read, Update a Delete. Dále některé specializovanější operace, které závisí na návrhu aplikace a struktuře dat. Snaží se vrchní vrstvy odstínit od nedůležitých detailů. Ať už jsou data uložena lokálně na zařízení či na vzdáleném serveru, tak by Repository vrstva měla zajistit stejná rozhraní pro práci s daty.

Určitě je dobré si uvědomit, že tato architektura je pouze doporučená a rozhodně to není tak, že by ji každý vývojář při tvorbě mobilní aplikace musel využít se všemi jejími komponenty. Každá aplikace je odlišná a není možné stanovit nejlepší architekturu takto obecně. Ovšem prostudovat si ji je určitě dobré a věřím, že si z ní spousta vývojářů něco vezme a do svojí aplikace alespoň část implementuje. Usnadní tak práci nejen sobě, ale hlavně dalším vývojářům, kteří budou kód číst a upravovat. Já sám jsem se ve své aplikaci snažil co nejvíce využít tyto komponenty a i když to ze začátku byl trochu problém všem komponentám architektury porozumět, tak by jsem se sám později ve svém vlastním kódu ztratil.

Aktivita

Tvoří **základní kámen** Android aplikace. Dalo by říct, že jedna aktivita se rovná jedné aktuálně otevřené obrazovce. Každá aplikace s touto architekturou by měla obsahovat základní aktivitu, která se spouští při startu aplikace. Ta je definována v souboru `AndroidManifest.xml`, který obsahuje základní konfiguraci aplikace. Umožňuje poté spouštět další aktivity, podle interakce uživatele s uživatelským rozhraním.

U aktivit je důležitý jejich životní cyklus. Ten začíná při prvním spuštění a končí jejím zničením. Třída `Activity` vystavuje metody, které knihovna pro práci s fragmenty zavolá ve chvíli, kdy se změní stav životního cyklu. Přetížením těchto metod můžeme implementovat chování UI, které si přejeme při změnách stavů životního cyklu provést.

- onCreate() - Prvotní vytvoření aktivity.
- onStart() - Obnovení viditelnosti aktivity.
- onResume() - Přenesení aktivity z pozadí na popředí.
- onPause() - Pozastavení aktivity či částečné překrytí.
- onStop() - Přechod aktivity na pozadí.
- onRestart() - Při přechodu z pozadí na popředí, pokud se předtím aktivita nacházela ve stavu onStop.
- onDestroy() - Zničení aktivity.

Fragment

Je komponent, který je aktivitě podobný, ale na rozdíl od ní není jeho přítomnost v architektuře tohoto typu nutná. Aktivita může mít 0..N fragmentů a jeden fragment může být spuštěn více aktivitami. Formálně by se dal tento vztah definovat jako 1..N ku 0..N. Komponent vznikl především kvůli zjednodušení návrhu responzivního UI. Díky tomu může být na velkém displayi zároveň zobrazeno více fragmentů a vyplní tak lépe plochu displaye. Pro tablety se používá většinou zobrazení dvou fragmentů vedle sebe. Takovému zobrazení se říká **two pane**.

Životní cyklus je velice podobný cyklu aktivity. Obsahuje o něco více základních metod kvůli závislosti na aktivitě. Velmi často se využívá konceptu **Single Activity** . Taková aplikace je řízena jedinou aktivitou, jednotlivá okna jsou potom vyobrazeny ve fragmentech.

LiveData

Je třída z knihovny AndroidX, která umožňuje držet libovolný objekt, který zároveň "pozoruje". V podstatě se jedná o obálku, která zapouzdřuje libovolný objekt. Pokud je tento objekt změněn, automaticky je pozorovatel notifikován a může na změnu reagovat. Využívá se zde principů návrhového vzoru Observer. Třída je narozdíl od běžného Observeru tzv. lifecycle-aware, což znamená, že respektuje životní cyklus komponent, ve kterých je inicializována. Velmi vhodné je použití v kombinaci s asynchronním programováním. Jedná se o opravdu jednoduchý princip, jak data, prezentované uživateli, udržet aktuální a v závislosti na změně dat provést i aktualizaci UI.

3.4 Programovací jazyk Kotlin

Je moderní, multiplatformní, staticky typovaný jazyk vytvořený společností JetBrains s.r.o. Syntaxí je velmi podobný moderním objektově orientovaným jazykům jako např. Java, C, Swift, Scala nebo Groovy. Snaží se poučit z jejich chyb a také co nejvíce minimalizovat množství kódu. Podívejme se tedy, jak dlouhou deklaraci třídy v jazyce Java 3.1, můžeme v jazyce Kotlin shrnout na pár řádků 3.2. Definuje struktury, které v běžných jazycích nenajdete. Také prosazuje používání prvků funkcionálního programování.

Důležitou vlastností, především při psaní aplikací pro Android, je jeho interoperabilita s programovacím jazykem Java. Můžeme tedy při programování využít knihovny psané v Javě. Kotlin totiž běží nad JVM[2] stejně jako Java.

```
1  public abstract class Device {
2      @NotNull
3      private String id;
4      private String name;
5
6      public Device(String id, String name) {
7          this.id = id;
8          this.name = name;
9      }
10
11     public String getId() {
12         return id;
13     }
14
15     public void setId(String id) {
16         this.id = id;
17     }
18
19     public String getName() {
20         return name;
21     }
22
23     public void setName(String name) {
24         this.name = name;
25     }
```

[2] Java Virtual Machine - Virtuální stroj, který je schopen zpracovat tzv. Java bytecode.

```
26 }
```

Výpis 3.1: Ukázka definice třídy Device v Javě.

```
1 abstract class Device (
2     val id: String,
3     val name: String?
4 )
```

Výpis 3.2: Pro srovnání definice stejné třídy v jazyce Kotlin.

Kotlin, díky kvalitě kódu a bezpečnosti, vyzdvihl programování mobilních aplikací zase o něco výš. Oficiálně byl pro Android představen Googlem v roce 2017, ale už v tu dobu jazyk existoval několik let.[13]

Kapitola 4

Návrh aplikace

V této kapitole bych rád popsal jak probíhal návrh a analýza požadavků před samotnou implementací aplikace. V první sekci najdete také analýzu existujících řešení.

4.1 Existující řešení

Před modelováním jsem se seznámil se způsobem vyčítání hodnot ze Sigfox API a přes rozhraní Bluetooth Low Energy, abych měl lepší představu, jak by mohla aplikace vypadat. Také jsem se snažil dohledat co nejvíce již existujících řešení.

Řešení specializovaných přímo pro Sigfox není mnoho. Většina aplikací bývá součástí nějaké platformy pro IoT, jež využívá Sigfox síť a synchronizuje data z jejich cloudu do svého. Tyto data následně poskytuje uživatelům. Uživatelé ocení možnost pracovat se senzory odlišných technologií v jednotném prostředí. Platformy jsou velice často zpoplatněné měsíčním tarifem. Sám Sigfox nemá vlastní klientskou aplikaci ani jako showcase. Pokud tedy najdeme aplikaci, jež umí komunikovat se Sigfox API, bývá často součástí nějakého většího celku. Tyto aplikace nalezneme přímo na stránkách Sigfox Partner Network. Většinou bývají zpoplatněné a možnost dema bývá na požádání. Nebylo tedy snadné si nějaké z nich projít.

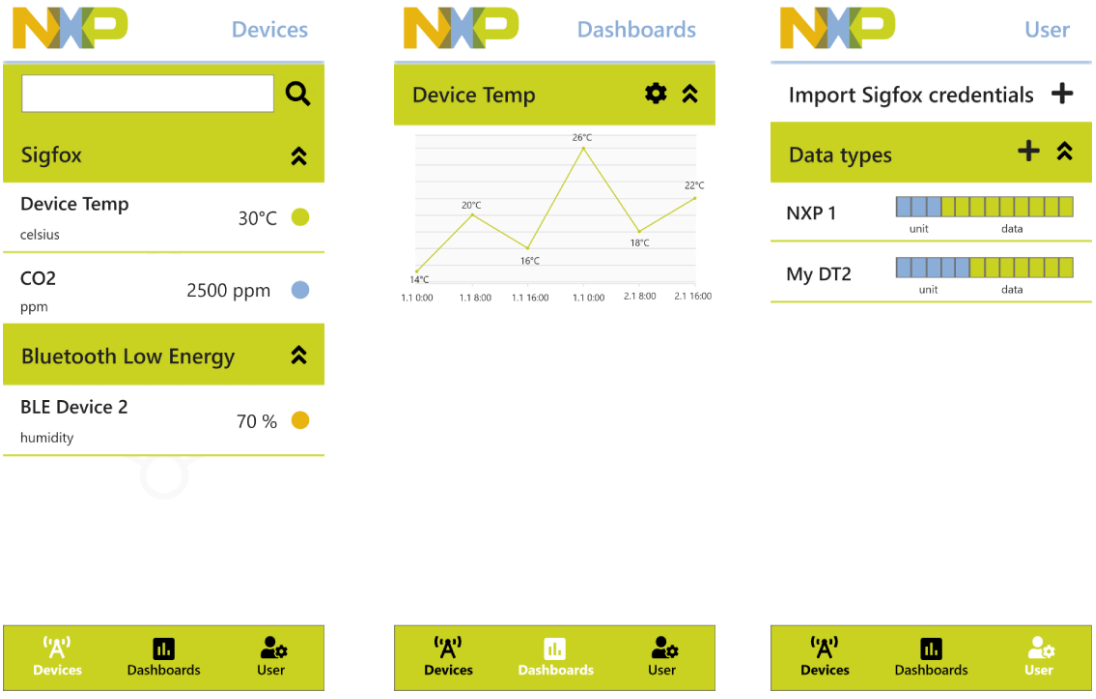
Osobně jsem se nechal inspirovat například českou společností MyMight a jejich mobilní aplikací Myjordomus, která nabízí i průvod aplikací skrze demo účet. Také mě zaujal projekt CleverFarm, který se věnuje především IoT v oblasti zemědělství. Myslím, že v této oblasti má technologie Sigfox velkou budoucnost a dokážu si představit širokou škálu využití. Věřím, že v budoucnu bude automatizace v zemědělství běžná a díky tomu budeme schopni zajistit větší kvalitu i kvantitu českých surovin. K přístupu do jejich platformy jsou však nutné údaje z portálu farmáře, který provozuje ministerstvo zemědělství. Takto jsem prošel více platforem a zaměřil jsem se hlavně na jejich mobilní aplikace. Vesměs je většina aplikací dosti podobná. Data se uživateli prezentují nejčastěji ve formě lineárních grafů naměřených hodnot za určité období, ukazatelů poslední naměřené hodnoty či nabízí jednoduché statistické zpracování dat.

V případě Bluetooth Low Energy jsou aplikace často vázány na vyčítání hodnot pouze ze zařízení jednoho výrobce. Ten často prodává senzory a čidla, ke kterým poskytuje vlastní aplikaci. Komunikační kanál aplikace s BLE zařízeními se často nedrží Bluetooth standardů či data mezi aplikací a senzorem nějakým způsobem modifikuje nebo šifruje. Bývá tedy problém obsluhovat takové zařízení bez znalosti těchto informací.

Protože vlastním senzor teploty a vlhosti od výrobce Xiaomi, měl jsem možnost si dobře vyzkoušet jejich aplikaci Xiaomi Home. Při prvním spuštění mě upoutala designová stránka aplikace, která je velmi přívětivá. Do aplikace je možné přidat senzory vyrobené společností Xiaomi a jejími partnery. To provedete velmi jednoduše a rychle. Zkrátka si své zařízení v aplikaci najdete podle názvu a zkusíte se s ním spojit. Pokud vše proběhne v pořádku, zařízení uvidíte v záložce “Mi Home“. Dále se mi líbila možnost seskupovat si senzory do “pokojů“ pro lepší přehlednost v případě, že máte senzorů více. Největším pozitivním překvapením je možnost nastavení komunikace mezi zařízeními bez interakce uživatele a to za pomoci tzv. scén. Například si můžeme nastavit, že při klesnutí hodnoty intenzity osvětlení se rozsvítí světla a zároveň o tom notifikujeme všechny uživatele. Konfigurace těchto "scén"probíhá pomocí definování podmínek typu If...Then...And... Vše je velmi přehledné a proto si myslím, že by takovou konfiguraci hravě zvládl i člověk, který není až tak oddán informačním technologiím.

4.2 Proces tvorby Mockupu

Před samotným vývojem jsem se sešel s vedoucím mé práce a zaměstnanci pobočky NXP, kde jsme společně vytvořili představu, co by měla aplikace splňovat a jak by měla vypadat. Navrhl jsem, že vytvořím mockup, abychom si lépe ujasnili představu.



Obrázek 4.1: **Návrh ve formě mockupu**.

Začal jsem mockup tvořit v aplikaci **Adobe XD**. Na tento nástroj jsem narazil zhruba před rokem a tvořil jsem v něm již mockupy k jiným projektům. To byl hlavní důvod, proč jsem si ho zvolil. Také se mi líbilo, že nabízí mobilní verzi, která se dá využít pro prezentaci mockupu. Jelikož se v Adobe XD, kromě statických prvků dají modelovat i dynamické přechody, můžeme tedy vizualizovat také akce po interakci uživatele a tím si lépe představit dynamický design aplikace.

Výsledný mockup 4.1 jsem odeslal a předvedl panu Nevoralovi a panu Obrovi z NXP, kteří moji práci zastřešovali po technické stránce. Převážně byli spokojeni a tak jsem doladil detaily a mohl jsem začít s implementací. Také jsem mockup předvedl panu Ing. Šimkovi při obhajobě semestrálního projektu. Byl taktéž spokojen.

4.3 Struktura Mockupu

Po projití několika již existujících řešení jsem načerpal dostatek inspirace, abych dokázal navrhnout uživatelské rozhraní.

Chtěl jsem, aby bylo přehledné a jednoduché. Samozřejmě by také mělo na první pohled zaujmout líbivým grafickým zpracováním. Tento "Wow efect" je důležitý hlavně u vývoje pro širší veřejnost a u showcase aplikací. Pokud uživatele aplikace při prvním spuštění neoslní nebo se mu dokonce nelíbí, často ji vůbec nezačne používat. A to hovořím jen o uživatelích, kteří ji nezavrhli hned podle screenshotů v obchodě Google Play. Zkrátka co není designově přívětivé se velmi špatně prodává.

Na začátku vytváření jsem si zvolil barvy, které budu při modelování používat. Na internetu je mnoho stránek, které vám usnadní výběr této "barevné palety" pro váš mockup. Já se rozhodl, že těchto nástrojů nevyužiji. Řekl jsem si, aby aplikace v uživateli evokovala pocit, že se jedná produkt od firmy NXP, tak použiji barvy, které NXP používá ve svém logu a jiných svých produktech. Prokonzultoval jsem to s panem Nevoralem a panem Obrem, zda-li to z jejich strany nečiní nějaký problém. Nápad se jim líbil, takže jsem měl barvy vybrané.

Dále jsem měl představu, že v aplikaci bude umístěno navigační menu vespod obrazovky (tzv. Bottom navigation bar). Tento způsob navigace mezi položkami se používá stále častěji v moderních mobilních aplikacích. Mě samotnému je toto ovládání velice pohodlné. Takové menu by však nemělo obsahovat více než pět položek, protože pokud by byli jednotlivé položky příliš malé, bylo by pro uživatele obtížné se na ně prstem trefit a to především u menších displayů. Při první fázi návrhu jsem menu rozdělil na položky **Devices**, **Dashboards**, **User**. To se v průběhu vývoje ještě lehce pozměnilo. Nakonec menu obsahuje čtyři tyto položky:

- **Devices** - Karta zobrazuje uživateli zařízení, ke kterým se může připojit a později z nich vyčíst hodnoty.
- **Sensors** - Karta zobrazuje uživateli jednotlivé zdroje dat (senzory), ze kterých už může přímo po stisknutí tlačítka získat hodnoty, za předpokladu, že se na zařízení, kde se senzor nachází, dokáže připojit. Do seznamu senzorů si uživatel vybere pouze ty zařízení a jejich senzory, které chce vidět. To učiní na kartě "Devices".
- **Dashboard** - Karta zobrazuje uživateli graficky vizualizovaná data. Pro zobrazení dat si uživatel přidá nový "widget", kde si zvolí typ zobrazení a zdroj dat (Jeden či více senzorů). Tento widget poté uživatel uvidí v seznamu widgetů. Může ho libovolně editovat či smazat.
- **User** - Karta zobrazuje uživateli nastavení aplikace. Přidávají se zde údaje pro přístup k Sigfox API. Tyto údaje uživatel nalezne po přihlášení na **??**Sigfox portál. Do aplikace je možné si naimportovat více Sigfox API účtů.

4.4 Použité nástroje a knihovny

Samozřejmě jsem se rozhodl určité části a komponenty aplikace neřešit sám a využít dostupné knihovny. Ušetříte si tím spoustu času a vyhnete se zbytečným chybám. Určité množství času je však nutné věnovat jejich studiu.

Android API

V tomto případě se nejedná přímo o knihovnu, ale spíše rozhraní pro přístup k utilitám operačního systému. V případě, že budete tvořit nativní Android aplikaci se můžete spolehnout, že Android API použijete. Já jsem využil hlavně komponenty pro tvorbu uživatelského rozhraní, předávání informací při přepínání mezi fragmenty, metody pro práci s více vlákny a operace s Bluetooth. Sám jsem byl překvapen, ale pro práci s Bluetooth Low Energy jsou v Android API velmi dobře nachystané třídy a metody.

Androidx

Jedná se o pomocnou sadu knihoven, kterou vyvíjí sám Google. Narozdíl od Android API je potřeba ji explicitně vyjádřit včetně verze v souboru `build.gradle`. Dalo by se říci, že jedná o "nadstavbu" Android API. Nejvíce jsem ji použil při práci s fragmenty a strukturou LiveData. Tyto prvky jsou popsány v sekci 3.3.

Room

Je součástí Androidx, ale kvůli větší komplexnosti ho uvádím zvlášť. Jedná se o ORM[1] framework pro práci s lokální SQLite databází. Umožňuje vytvoření tzv. DAO[2] rozhraní, ve kterých probíhá definice databázových dotazů pomocí anotací. Příklad takové definice je uveden ve zdrojovém kódu 4.1. Framework poté z anotovaných rozhraní dokáže na pozadí vytvořit konkrétní objekty. Také je nutné vytvořit třídy, na které nám framework bude databázové entity mapovat. Jedná se o jednoduché třídy jejichž atributy budou mapovány na databázové atributy. Takové třídy musí být označeny anotací `@Entity`. Room umožňuje i verzování databáze, za pomoci migrací. Migrační skripty je nutné definovat v jazyce SQL přímo v kódu nebo v externím SQL souboru, který potom importujeme.

S frameworkem se mi pracovalo dobře a mohu ho pouze doporučit. Není tak komplexní jako jiné ORM frameworky, se kterými jsem pracoval (Hibernate, Spring Data JPA), ale přesto nabízí pohodlný a dostačující manévrovací prostor i pro složitější dotazy.

```
1  @Dao
2  interface SensorDao{
3      @Query("SELECT * from SENSORS")
4      fun getAll(): LiveData<List<Sensor>>
5
6      @Insert(onConflict = OnConflictStrategy.REPLACE)
7      fun insert(sensor: Sensor)
8
9      @Query("SELECT * from SENSORS WHERE id=:id ")
10     fun getOne(id : String) : Sensor
11
12     @Query("DELETE FROM SENSORS")
```

[1] Object Relational Mapping - technika, která převádí databázové entity na objekty v objektově orientovaném jazyce

[2] Database Access Object - objekt, který zprostředkovává jednotný a transparentní přístup k datům

```
13      suspend fun deleteAll()
14
15      @Query("DELETE FROM SENSORS WHERE id=:id")
16      fun delete(id: String)
17
18      @Query("SELECT * from SENSORS")
19      fun getAllNoLive(): List<Sensor>
20
21      @Query("SELECT * FROM SENSORS WHERE id in (:ids)")
22      fun getAllByIds(ids : List<String>)  : List<Sensor>
23
24      @Update
25      fun update(sensor: Sensor)
26  }
```

Výpis 4.1: Ukázka definice rozhraní DAO v frameworku Room (Kotlin)

Material Design

Je sada pokynů a doporučení pro tvorbu grafického designu webových, mobilních i desktopových aplikací. Někdy se takovým příručkám říká designový jazyk (design language). Ve své podstatě popisuje, jak má daný ovládací prvek vypadat, co je pro uživatele přívětivé a čemu se vyvarovat. Také na svých stránkách poskytuje zdarma ke stažení velké množství ikon, které ve své aplikaci využívám.

Kromě této teoretické části Material Design nabízí i vlastní knihovnu pro Android. Ta definuje ovládací prvky, které dědí od těch defaultních a mění jejich design, tak aby korespondoval s doporučeními v příručce. Díky dědičnosti můžeme tyto "rozšířené" komponenty použít stejně jako ty standardní. Tím si ulehčíme práci při tvorbě designu. Obzvláště to ocení méně kreativní jedinci. Avšak je dobré mít na paměti, že ne všechny Android komponenty mají i svoji Material design variantu. V zásadě se jedná o základní komponenty typu: Button, EditText, DropDown list atd.

Retrofit

Je typově bezpečný HTTP klient. Jedná se o nejpoužívanějšího klienta v Android aplikacích. Velmi snadno se s ním pracuje. Podobně jako v případě Room frameworku si vytvoříme rozhraní, které nám bude poskytovat jednotný přístup k datům. I zde konfigurace jednotlivých metod rozhraní probíhá pomocí anotací. Jen namísto SQL dotazu, nastavujeme URL adresu a parametry HTTP dotazu. Opět musíme mít vytvořené třídy, jejichž instance budeme přes klienta posílat a přijímat. Často se jim říká DTO[3]. K vytvoření takových tříd nám opět pomůžou anotace. U atributů musíme specifikovat jejich název v serializovaném formátu. Pokud přichází serializovaná zpráva s více atributy než má DTO, tak se ty nespecifikované ignorují.

Odesílání požadavků a čekání na odpověď probíhá zpravidla asynchronně pomocí rozhraní `Callback`, které specifikuje 2 metody: `onResponse` a `onFailure`. Pro každý asynchronní dotaz vytvoříme objekt, jež implementuje toto rozhraní a v metodách si definujeme chování, které chceme. Knihovna podporuje i synchronní požadavky, ale kvůli optimalizaci je nedoporučuji používat, pokud to není vyloženě nutné.

[3]Data Transfer Object - objekty, které přenáší data mezi službami

Příklad serializované zprávy ve formátu JSON 4.2 a DTO, do kterého framework zprávu uloží 4.3.

```
{
    "device":{"id": "24E59E"},
    "time": 1588523654000,
    "data": "fefe000a0200",
    "seqNumber": 41,
    "rinfos":[{"baseStation":{"id": "367D" }, "delay": 1.0169999599456787, "lat": "49.0",],
    "satInfos":[],
    "nbFrames": 3,
    "operator": "SIGFOX_Czech_Rep_Simplecell",
    "country": "CZE",
    "computedLocation":[],
    "lqi": 0
}
```

Výpis 4.2: Jedna zpráva ze zařízení 24E59E z Sigofox API (JSON)

```
data class SigfoxDeviceMessage (
    @Expose
    @SerializedName("device")
    val deviceDto: SigfoxDeviceDto,

    @Expose
    @SerializedName("time")
    val timeStamp: Long,

    @Expose
    @SerializedName("data")
    val data: String,

    @Expose
    @SerializedName("lqi")
    val linkQuality: Int,

    @Expose
    @SerializedName("seqNumber")
    val sequenceNumber: Int
)
```

Výpis 4.3: Příklad Retrofit DTO nachystané pro přijímání (Kotlin)

MPAndroidChart

Knihovna pro kreslení grafů. Umožňuje vizualizovat od základních grafů až po složitější grafické reprezentace. Definuje komponenty, které přidáme do XML souboru, ve kterém specifikujeme design. Většinu konfigurace provádíme přímo v kódu. Hlavně pro zajištění lepší dynamičnosti, jelikož často pracujeme s předem neznámou množinou dat. Nastavujeme vzhled jednak pro celý graf, tak i pro jednotlivé datové sety.

Jelikož je konfigurace vzhledu grafu přímo v kódu, je nutné aplikaci vždy spustit, abychom se podívali, jak se úprava vzhledu dotkla celkového designu našeho grafu. Proto poměrně dlouho trvá než doladíme vzhled grafu našim představám. Nicméně je to asi nejlepší knihovna pro vytváření grafů, jež je zdarma k použití. Všechny souřadnice datových bodů musíme převádět na datový typ float. Což může způsobovat optimalizační problém, při vykreslování grafů s velkým počtem vstupních dat, které nejsou typu float.

Roboeletric

Framework pro testování Android aplikací, který umožňuje spouštět testy ve vývojovém prostředí bez nutnosti emulátoru a zároveň využívat komponenty Android API. Testy pro Android se dají rozdělit na 2 typy:

- **Tests** - Spouští se ve vývojovém prostředí. Nemohou využívat Android komponenty. Klasické JUnit testy.
- **Android Tests** - Testy, které se spouští přímo na připojeném Android zařízení nebo na emulátoru.

Emulátor je pomalý a ne vždy při spouštění testů máme připojené mobilní zařízení. Díky tomuto frameworku dokážeme spustit testy, které by musely spadat do kategorie Android Tests jako normální testy. Framework tedy dokáže simulovat Android API. Simulace se může chovat lehce odlišně, jelikož není možné přesně simulovat některé jeho části. Hlavně část API přistupující k hardwarovým komponentám. Tyto části jsou mockovány[4] a jejich chování je specifikováno často zjednodušeně, aby bylo možné jej v testech využít.

Mockito

Framework pro tvorbu testovacích mocků. Původně vyvíjený pro jazyk Java. Později doplněný i pro jazyk Kotlin. Umožňuje jednoduchým způsobem pomocí anotací tvořit mocky a definovat jejich chování.

Využití jsem našel především při testování volání REST API přes knihovnu Retrofit 4.4. Jelikož v testech není možné pracovat s reálnými daty ze Sigfox API.

[4]Mock - Objekt, který simuluje chování jiného reálného objektu

Kapitola 5

Implementace

Před samým začátkem implementace jsem analyzoval možné technologie vývoje zmíněné v sekci 3.2. Od začátku jsem byl nakloněn spíše k nativnímu vývoji v programovacím jazyce Java. Po zkoumání ostatních technologií se moje rozhodnutí lehce změnilo. Přístup k vývoji zůstal stejný, ale rozhodl jsem se pro programovací jazyk Kotlin. Jazyk Java znám poměrně dobře a měl jsem chuť se přiučit něco nového. Líbil se mi Kotlin také proto, že v porovnání s jazykem Java nabízí to stejné a ještě něco navíc. Také byl ve většině článků k tvorbě moderních mobilních aplikací doporučován.

Nativní implementace Android aplikace není nejrychlejší cesta, ale výsledný produkt poté dosahuje lepší robustnost a rychlost než multiplatformně vyvíjené mobilní aplikace.

Rozhodně nechci popisovat podrobné kroky mojí implementace. Chtěl bych v následující kapitole pouze nastítnit nejdůležitější části mé práce a některým zajímavým detailům se věnovat podrobněji.

5.1 Android studio

Implementace probíhala v nejznámějším IDE[1] pro vývoj Android aplikací. Líbil se mi fakt, že je postavené na jádru IntelliJ. V tomto IDE často pracuji při programování v jazyce Java. Takže jsem neměl problém se adaptovat přesto, že jsem tento nástroj nikdy předtím nepoužil. Studio nabízí spousty vychytávek, které jako vývojář mobilních aplikací oceníte. Jedná se např. o zabudovaný Android emulátor, editor designu, klient pro SQLite databázi a mnoho dalšího. Velmi kladně hodnotím i integraci s Git a Gradle. Pokud by vám něco chybělo, je dobré se porozhlédnout v dostupných pluginech, jestli třeba někdo nevytvořil pro váš problém už nějaký plugin. Je jich opravdu spousty i já jsem při implementaci nějaké využil.

Celkově se mi ve studiu pracovalo dobře, mohu vytknout jen pomalejší spuštění, které může na slabším počítači trvat i několik minut.

5.2 Git

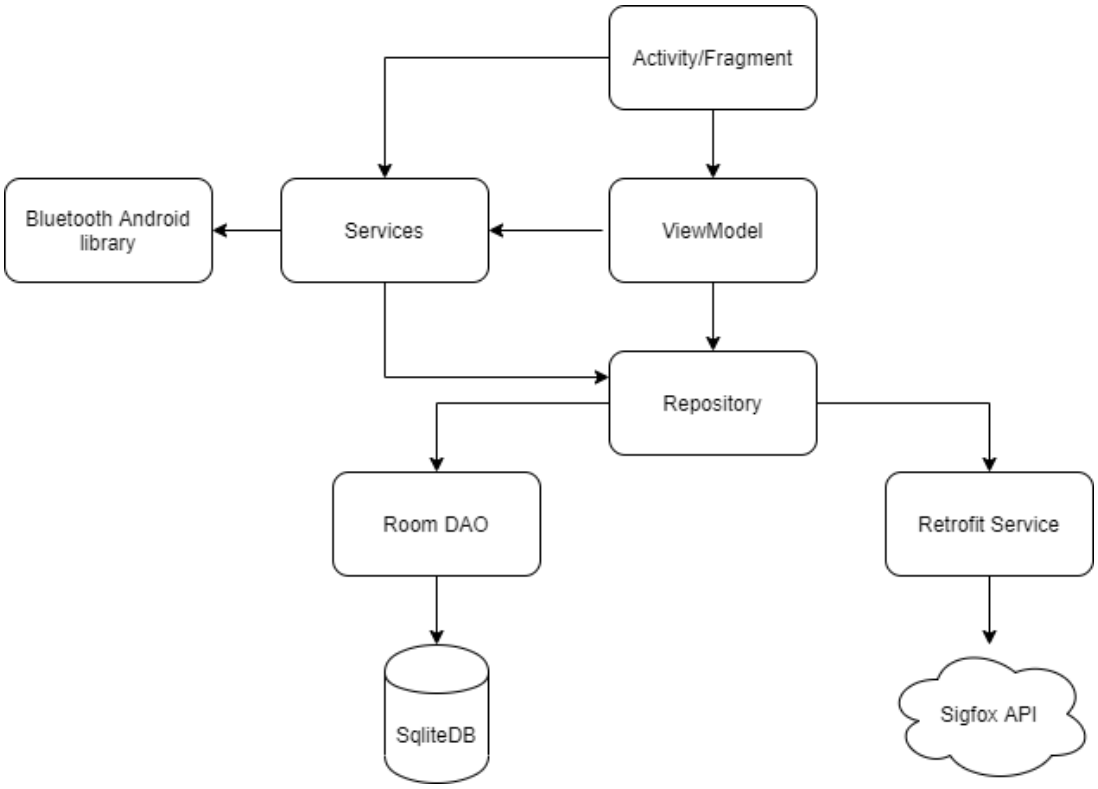
S nástrojem se setkal asi každý, kdo se v oboru vývoje softwaru pohybuje. Proto jej asi není potřeba nijak popisovat. Už v prvním ročníku jsem ho využíval pro školní projekty. Bylo

[1] Integrated development environment - software, který obsahuje co nejvíce funkcí, které vývojář potřebuje pro vývoj v daném jazce

tedy jasné, že opět nástroj využiji pro verzování kódu. K uchovávání zdrojového kódu na vzdáleném serveru jsem využil cloudového úložiště GitHub, který je pro studenty zdarma.

5.3 Architektura této implementace

V tomto případě jsem se zdržel kreativity a postupoval jsem podle doporučení, které mi nabízelo vývojové IDE a oficiální dokumentace. To je aplikace s arhitekturou postavnou na návrhovém vzoru MVVM 3.3. Přesto se v architektuře aplikace nachází nějaké prvky, které v té oficiální nejsou. Jedná se především o služby(Services). Architektura je popsaná následujícím diagramem 5.1.



Obrázek 5.1: **Architektura této aplikace**.

Aktivity

Aplikaci jsem začal vyvíjet jako podle konceptu "single activity", který je popsán v sekci 3.3. Přepínání mezi fragmenty jsem řešil následujícími způsoby:

- **automaticky** - S pomocí `androidx.navigation.ui.NavigationUI`, jež dokáže automaticky nastavit navigační menu. Definujete si statický vzhled i dynamické chování navigačního menu pomocí XML souborů. Při inicializaci aktivity poté zavoláte funkci `setupWithNavController`, které předáte příslušné View a ona provede konfiguraci přepínání fragmentů za vás. To vám ušetří psaní a udržování funkcionality, která by toto zajišťovala. Dosáhnete tím stabilního navigačního menu.

- **ručně** - Pomocí fragment manageru, kterému předáte příslušné View a fragment. Po zavolání metody `commit()` vám aktuální fragment bude nahrazen za nový.

Nicméně později během vývoje jsem zjistil, že na některé fragmenty bude lepší si definovat nové aktivity, takže jsem u konceptu "single activity" nezůstal. Avšak věřím, že může být pro jednoduché aplikace vhodný. Vytvořil jsem si tři specializované aktivity:

- **DashboardConfigurationActivity** - Aktivita pro konfiguraci widgetu v dashboardu. Po inicializaci spouští `DashboardConfigurationFragment`, který obstarává veškeré řízení uživatelského rozhraní.
- **SensorConfigurationActivity** - Aktivita pro konfiguraci senzoru. Taktéž spouští po inicializaci fragment `SensorConfigurationFragment`.
- **SigfoxPrefixConfigurationActivity** - Aktivita pro konfiguraci prefixovaných zpráv přes Sigfox síť. Také spouští po inicializaci fragment `SigfoxPrefixConfigurationFragment`.

Využil jsem principů dědičnosti, abych zbytečně neopakoval kód. Definoval jsem si společného předka pro všechny aktivity v aplikaci. Do předka jsem se snažil přesunout co nejvíce metod, které by se dali napříč aktivitami použít. Jako je např. nastavení navigačního menu či inicializace toolbaru na vrchní části obrazovky.

Fragmenty

Fragmenty jsem využíval pro veškerou inicializaci UI. Pro každou obrazovku jsem si vytvořil speciální fragment. Nevyužil jsem žádného společného předka jako u aktivit, jelikož je každý fragment velice specializovaný a pracuje pokaždé jinými s UI komponenty, nebylo to ani nutné. Všechny fragmenty v aplikaci tedy dědí z AndroidX třídy `Fragment`.

LiveData

Tento princip jsem se snažil využívat co nejvíce to šlo. Použití návrhového vzoru MVVM k tomu přímo vybízí. Třídu `androidx.lifecycle.LiveData` jsem využil v dotazech do lokální databáze, při stahování dat ze Sigfox API přes Retrofit knihovnu a skenování okolních Bluetooth zařízení. Všechny tyto části pracují asynchronně, ovšem lže je provést i synchronně, zbytečně se tím však aplikace zpomaluje. Inicializaci třídy `Observer` pro LiveData jsem prováděl vždy při vytvoření fragmentu. Třída obsahuje pouze 1 metodu `onChanged`, ve které jsem dynamicky upravoval UI podle změny získaných dat.

Repository

Ačkoli se v Android aplikacích často tato vrstva opomíjí, já jsem ji využil hlavně z důvodu, že jsem byl zvyklý na tento návrhový vzor z předchozích projektů. Líbí se mi koncept jednotného přístupu k datům, ať už se jedná o přístup do databáze nebo k API. Definuji si zde metody, které zajišťují čtení, modifikaci a mazání dat podle potřeby aplikace.

Services

Jedná se o doprovodné třídy řešící složitější operace jako např. skenování okolních bluetooth zařízení, zpracovávání přijatých dat, šifrování a dešifrování přístupových údajů k API a

další. Nevyužil jsem konceptu Services podle oficiální Android specifikace. Ta totiž hovoří o službách jako o objektech, které se využívají pro dlouho běžící procesy. Já jsem je pojal spíše jako pomocné služby s možností přístupu k Repository vrstvě. Přístupu se využívá pouze ve třídách `SigfoxService` a `CredentialsService`. V tomto konceptu tvoří prostředníka mezi ViewModel a Repository vrstvou. Kvůli větší složitosti při získávání dat ze Sigfox API či databáze totiž nemají ViewModely přímý přístup.

5.4 Skenování dostupných zařízení

Liší se v závislosti na typu zařízení. Všechna úspěšně naskenovaná zařízení jsou ukládána do tzv. holderu `InternalDeviceHolder`. To je Singleton třída, která obsahuje seznam všech aktuálně naskenovaných zařízení pro BLE i Sigfox. Při opakovaném spuštění skenování je tento seznam promazán a znovu naplněn novými zařízeními. Je tím zajištěný snadný přístup k naskenovaným zařízením v rámci celé aplikace.

BLE zařízení

Pro skenování okolních BLE zařízení jsem si definoval třídu `BleDeviceScanner`, která má 2 hlavní metody `startScan()` a `stopScan()`. Tyto metody zapouzdrují volání metod objektu `android.bluetooth.le.BluetoothLeScanner`. Je nutné si definovat vlastní `ScanCallback`, který obsahuje 2 metody `onScanResult` a `onScanFailed`. V těchto metodách reagujeme na úspěšné či neúspěšné skenování.

Sigfox zařízení

Skenování Sigfox zařízení probíhá jednoduše HTTP GET požadavkem na Sigfox API. V odpovědi dostaneme seznam zařízení, ke kterým máme oprávnění, ve formátu JSON.

Uživatel si vybere zařízení, ze kterého chce vyčítat hodnoty a přidá si ho do senzorů. Před samotným přidáním je ještě vyzván k zadání prefixů, pomocí kterých může rozřadit data z jednoho zařízení do více senzorů. Je tedy možné ze zařízení odesílat do sítě Sigfox hodnoty různých čidel, které budou vázány k jednomu zařízení z pohledu Sigfox API. Uživatel si tedy vybere jestli chce takto rozřazovat přijatá data. Pokud ano, je vyzván k vybrání počtu prefixů, které bude aplikace rozlišovat. U každého prefixu si zvolí název a bytovou hodnotu, která ho bude uvnitř zprávy reprezentovat. Prefix je vždy první byte v odeslané hodnotě. Uživatel má tedy na výběr 0 až 255. Aby bylo přidávání prefixů pro uživatele přehledné, počet prefixů je omezen na 5. Pokud se prefixy rozhodne nevyužít, jsou veškerá data ze zařízení vztahována k jednomu senzoru.

Do databáze se uloží pro každý prefix jeden senzor s informací o identifikátoru zařízení, bytovou hodnotou prefixu a uživatelském jméně definujícím, ke kterému účtu je v rámci Sigfox platformy zařízení navázáno. Do aplikace si totiž můžeme vložit více uživatelských účtů a kvůli optimalizaci je dobré znát, které přihlašovací údaje do API máme později použít při vyčítaní hodnot.

5.5 Připojení přes Bluetooth Low Energy k zařízení

O připojení k zařízením přes BLE se stará třída `BleConnector`, která už při inicializaci očekává tzv. gatt callback. To je třída s předkem `android.bluetooth.BluetoothGattCallback`, ve které definujeme chování po připojení k zařízení, po zjištění dostupných BLE services,

po vyčtení hodnoty z BLE charakteristiky a další možné reakce. `BleConnector` obsahuje jednu hlavní metodu `connect`, která se dokáže připojit k zařízení na základě jeho adresy. Ta se předává do metody parametrem. Pokud zařízení není již připojené, zavolá funkci z knihovny `android.bluetooth`, která se pokusí k zařízení připojit.

Uživateli je po přípojení zobrazen seznam služeb a charakteristik daného BLE zařízení. Vybere si, které charakteristiky chce přidat do aplikace jako senzory. Ty se uloží do lokální databáze s informací o adrese zařízení, identifikátorem služby a charakteristiky.

5.6 Vyčítání naměřených hodnot

Vyčítání dat iniciuje třída `SynchronizeValueListener`, která podle typu zařízení spustí mechanizmus pro vyčtení hodnoty.

BLE zařízení

V případě, že se jedná o BLE zařízení, je zavolána metoda bluetooth knihovny `readCharacteristic`, která zajistí přečtení hodnoty charakteristiky na vybraném zařízení. Až je hodnota vyčtena, knihovna zavolá metodu `onCharacteristicRead` v gatt callbacku, ve které s vyčtenou hodnotou dále pracujeme.

Sigfox

Vyčítaní naměřených hodnot v případě, že se jedná o Sigfox zařízení probíhá opět voláním REST API. Tentokrát s požadavkem na získání všech zpráv k zařízení, které jsou na cloudu uloženy a jsou novější než poslední synchronizovaná zpráva. Díky tomu, že API nabízí možnost filtrování podle časové známky, je možné stahovat z API pouze nové zprávy. Tím se značné zrychlí zpracování příchozích dat. Do požadavku je nutné zakomponovat identifikátor zařízení pomocí tzv. path variable. Ten je uložen v lokální databázi u senzoru. V odpovědi dostaneme naměřená data, čas měření, sekvenční číslo měření a další informace.

Získané hodnoty jsou ukládány do lokální databáze v binární podobě spolu s časovým razítkem údávajícím čas měření. A to stejným způsobem pro oba typy zařízení. V případě prefixovaných zpráv jsou získané hodnoty filtrovány dle prefixu a uloženy ke konkrétnímu zařízení z daným prefixem. Uložené hodnoty mohou být dále vizualizovány do grafů.

5.7 Zpracování přijatých dat ze senzorů

Sám bych řekl, že se jedná o nejdůležitější a zároveň nejzajímavější část celé práce. Můj hlavní cíl bylo nastavit zpracování tak, aby bylo možné číst hodnoty pohodlně z co největšího počtu zařízení.

Data jsou od senzorů přijímána v binární podobě. V této podobě jsou i uchovávána v databázi. Interpretace takových dat se poté liší v závislosti na nastavení konfigurace senzoru. Především závisí na datovém typu a jednotce, která je u senzoru nastavena.

Datový typ

Standardně aplikace podporuje prezentaci třech základních datových typů. Bytová konstanta, která datový typ reprezentuje, je inspirována dle BLE specifikace typů.

Hodnota	Jednotka	Zkratka
0x00	Celsius	°C
0x01	Procento	%
0x02	Watt	W
0x03	Sekunda	s
0x04	Minuta	m
0x05	Hodina	h
0x06	Den	day(s)
0x07	Amper	A
0x08	Volt	V
0x09	Ohm	Ω
0x10	Lux	lx
0x11	Gauss	G
0x12	Litr	l
0x13	Metr	m
0x13	Kilometr	km
0x13	Centimetr	cm
0x12	Metr krychlový	m^2
0x13	No unit	

Tabulka 5.1: Jednotky, které aplikace podporuje.

- **Integer - 0x10** - Data jsou prezentována jako celé číslo. Bytová hodnota je před zobrazením uživateli převedena na příslušný datový typ v jazyce Kotlin podle délky. Jedná se o typy Byte(1B), Short(2B) a Integer(4B).
- **Float - 0x14** - Data jsou prezentována jako desetinné číslo. Bytová hodnota je vždy převedena na datový typ Float.
- **String 0x19** - Data jsou prezentována jako řetězec znaků. Dle ASCII tabulky jsou jednotlivé byty převedeny na znaky.

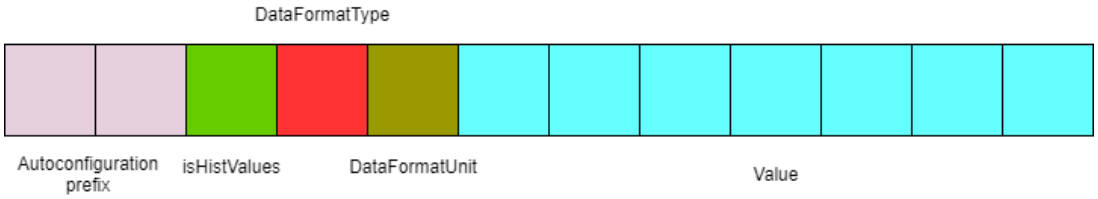
Jednotka

Aplikace podporuje u senzorů nastavení různých fyzikálních jednotek. Díky tomu může být hodnota přehledně vizualizována uživateli a on lépe pochopí k čemu se hodnota může vztahovat. Seznam všech dostupných jednotek je uveden v tabulce 5.1.

Autokonfigurační formát zpráv

Chtěl jsem, aby aplikace podporovala nějaký způsob autokonfigurace. Zkrátka, aby uživatel nemusel v aplikaci nastavovat nějaké konfigurace a senzor se mu nastavil sám na příslušný datový typ a jednotku. Vymyslel jsem formát zpráv, který obsahuje informace o datovém typu zprávy, jednotce ve které byla hodnota změřena a samotnou hodnotu nebo pole historicky naměřených hodnot. Formát můžeme pozorovat na následujícím obrázku B.1. Později jsou jednotlivé části rozepsány v tabulce 5.2.

Pokud taková zpráva přijde na zařízení z nějakého senzoru, automaticky změní u senzoru datový typ a jednotku podle přijaté zprávy.



Obrázek 5.2: Formát autokonfigurační zprávy.

Pole	Popis	Délka
Autoconfiguration prefix	Tento prefix indikuje, zda se jedná o autokonfigurační zprávu. Musí mít hodnotu **0xFEFE**, jinak nebude zpráva parsována jako autokonfigurační.	2B
isHistValues	Jedná se o příznak, který značí, zda-li se v části Value nachází pole historických hodnot či pouze jedna hodnota. **0x01** pro historické hodnoty, **0x00** pro jednu hodnotu.	1B
DataFormatType	Hodnota značí v jakém datovém typu je zpráva poslána. viz 5.7	1B
DataFormatUnit	Hodnota značí v jaké jednotce byla hodnota naměřena. viz 5.7	1B
Value	Naměřená hodnota	n

Tabulka 5.2: Jednotlivé části autokonfiguračního formátu a jejich význam.

V případě, že se jedná o zprávu, která obsahuje pole historických hodnot je parsována dle konečného automatu, který je sestaven podle schématu 5.3.
Jednotlivé znaky jsou reprezentovány příslušnými binárními hodnotami dle tabulky ASCII. Je tedy nutné dodržet následující formát pro historické hodnoty:

```
[{*hodnota*,*časová známka*},{*hodnota*,*časová známka*},...]
```

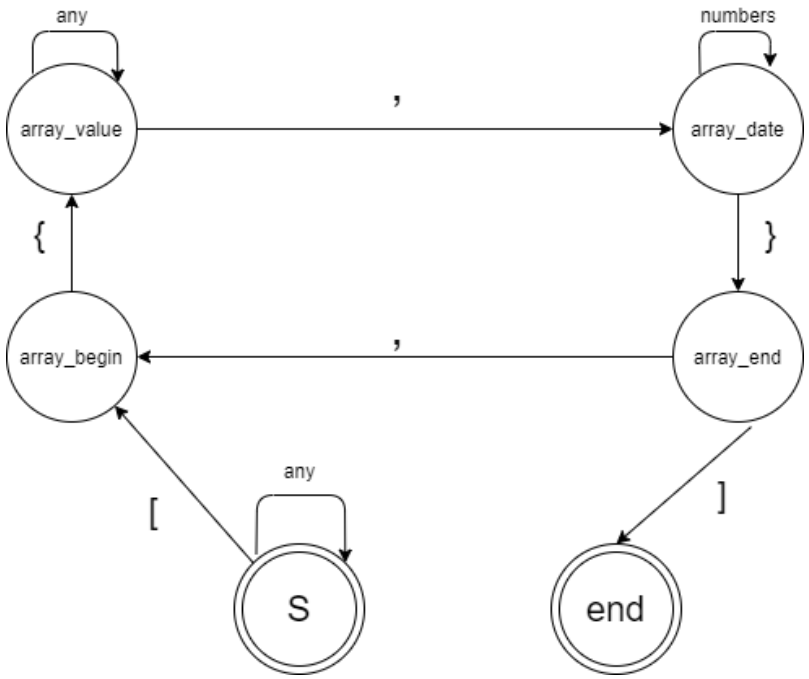
Hodnota má binární podobu, stejně jako v ostatních typech zpráv. Časová známka má tvar tzv. UNIX timestamp.

5.8 Ukládání Sigfox API přihlašovacích údajů

K přístupu do Sigfox API je nutné, aby aplikace měla k dispozici přihlašovací údaje do API. Tyto údaje uživatel přidává na záložce "User" a jsou tvořeny uživatelským jménem a heslem. Přidat jich může libovolný počet. Při skenování zařízení se v aplikaci zobrazí dostupná zařízení ze všech naimportovaných API účtů.

Jelikož jsou tyto údaje citlivé, je nutné je uchovávat v šifrované podobě. V případě, že by se někdo šikovný dostal k datům uloženým v databázi vašeho telefonu, měl by kompletní přístup pro volání Sigfox API pod vaším účtem. Mohl by získat informce o vašem profilu, seznam zařízení, zprávy ze zařízení a mnoho dalšího.

Třída `CredentialsService` obstarává ukládání přihlašovacích údajů do databáze a jejich čtení z databáze. Uživatelské jméno je uloženo v nešifrované podobě zatímco heslo se při



Obrázek 5.3: Konečný automat pro parsování historických hodnot.

uložení šifruje s využitím pomocné třídy pro šifrování `KeyHelper`. Při získávání uložených údajů se heslo dešifruje.

Šifrování probíhá pomocí symetrické blokové šifry AES-256[2]. Jedná se o blokovou šifru s klíčem o délce 256 bitů, který je při prvním šifrování vygenerován pomocí objektu `javax.crypto.KeyGenerator` 5.1. Při každém dalším šifrování je použit již vygenerovaný klíč. Tímto klíčem jsou hesla při čtení dešifrována.

```
fun generateKey(): Key {
    if (!keyStore.containsAlias(KEY_ALIAS)) {
        val keyGenerator: KeyGenerator = KeyGenerator.getInstance(KeyProperties.
    KEY_ALGORITHM_AES, AndroidKeyStore)
        keyGenerator.init(
            KeyGenParameterSpec.Builder(
                KEY_ALIAS,
                KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT
            )
                .setBlockModes(KeyProperties.BLOCK_MODE_GCM).setEncryptionPaddings(
    KeyProperties.ENCRYPTION_PADDING_NONE)
                .setKeySize(KEY_SIZE)
                .build()
        )
        return keyGenerator.generateKey()
    }
    return keyStore.getKey(KEY_ALIAS, null)
}
```

Výpis 5.1: Generování klíče pro šifrování (Kotlin)

[2]Advanced Encryption Standard - standartizovaný algoritmus pro symtrické šifrování

5.9 Návrh možných rozšíření implementace

Aplikace je momentálně ve stavu, ve kterém dokáže elegantně vyčítat hodnoty ze zařízení s protokolem BLE a ze Sigfox API. Umožňuje si uložit senzory a čidla pro pohodlnější další používání. Získané hodnoty perzistuje v lokální databázi. Také k nim poskytuje možnost přehledného zobrazení v grafech, které si sám uživatel může nastavit. Tvořit grafy může z dat získaných z jednoho či více senzorů. Také si může zvolit statistickou funkci a tu zobrazit v daném intervalu za pomoci sloupcového grafu. Podporuje několik formátů přijímaných zpráv, popsaných v sekci 5.7. Může si vybrat časový rozsah grafů pomocí výběru přesného data nebo sledovat hodnoty za několik posledních hodin/dní. Přesto všechno není dokonalé a najde se zde spousta nápadů na vylepšení.

Už v průběhu vývoje jsem si shromažďoval nápady, které bych rád implementoval. Některé z nich jsem již nestihl, jiné jsem už od začátku zavrhl a věděl jsem, že se na ně nedostanu. V této sekci nastíním nejzajímavější tipy k vylepšení aplikace.

Synchronizace dat do vlastního cloudu

Jelikož naměřené hodnoty zůstávají dostupné skrze Sigfox API pouze po dobu třech dnů. Je nutné je vyčíst a uložit do lokální SQL databáze dříve než zmizí. Myšlenka byla taková, že by mezi Sigfox API a mobilním telefonem byl ještě takový "prostředník", který by automaticky pomocí callbacků, které jsou popsány v sekci 2.4.4, synchronizoval data s Sigfox API. Mobilní klient by poté komunikoval pouze s cloudem.

V případě měření přes rozhraní BLE by bylo možné využít mobilního telefonu jako klienta, který by naměřené hodnoty odesílal do cloudu.

Tímto by se mobilní klient zbavil velké části aktuální SQLite databáze a většinu dat si stahoval z cloudu. Takové řešení by veškeré senzory a jejich hodnoty vázalo k uživatelským účtům, které by si uživatelé museli založit. Díky tomu by mohli synchronizovat hodnoty ze senzorů do více zařízení např. tablet i mobilní telefon.

Autentizace při vstupu do aplikace

Přestože aplikace nepracuje s citlivými daty, přeci jen nechcete aby si někdo jiný, kdo se dostane k vašemu telefonu, mohl zjistit např. průměrnou spotřebu elektrické energie. Aplikace by tedy mohla vyžadovat autentizaci pomocí zadání hesla nebo otisku palce. Tím by byla zabezpečena proti nedovolenému přístupu. Heslo i otisk by si uživatel zvolil při prvním spuštění.

Synchronizace hodnot ze Sigfox API na pozadí

Je poměrně nešikovné, že musíme data z API synchronizovat ručně. Obzvláště, protože jsou data dostupná pouze tři dny. Aplikace by mohla data několikrát za den automaticky synchronizovat, bez jakékoli interakce uživatele. Uživateli, by tedy byli k dispozici stále "aktuální data". To jak moc jsou aktuální, by záleželo na frekvenci vyčítání na pozadí. Tento problém se synchronizací řeší návrh o vlastním cloudu 5.9.

To vše by mohlo být podmíněno nastavením, kde by si uživatel vybral zda-li chce data synchronizovat pouze ručně nebo i na pozadí.

Kapitola 6

Testování mobilní aplikace

Velmi často se testování při vývoji produktu dostává do pozadí a nezbývá na něj tolik času. Zkrátka firma chce raději vydávat nové funkcionality, které může prodávat zákazníkům. Nicméně nedostatečné testování se může časem vymstít. I tohoto faktu jsem si byl vědom a tak jsem se snažil testování nepodcenit.

6.1 Automatické testování

Android primárně nabízí 2 druhy testování, které jsou zmíněné v kapitole o frameworku Roboelectric 4.4. Testům, jež se spouští přímo na Android zařízení či emulátoru, jsem se snažil vyhnout. Chtěl jsem, aby bylo možné testy spustit na jakémkoli zařízení a v budoucnu je například spouštět i automatizovaně na serveru např. po změně zdrojových kódů v repozitáři.

Pro spouštění testů jsem použil `RobolectricTestRunner`, který automaticky mockuje některé části z Android API. Vytváří tzv. shadow objekty. To jsou objekty, které implmentují rozhraní reálných tříd. Díky tomu je "shadow" objekt typově shodný s reálným objektem a můžeme s ním tak i pracovat. Navíc často definuje nějaké pomocné metody, které nám usnadní testování ve složitě testovatelných částech. Nejvíce jsem tento princip uplatnil při testování funkcionalit okolo Bluetooth.

Dále jsem využil několika metod pro testování fragmentů definovaných v rámci knihovny AndroidX. Pomocí funkce `launchFragmentInContainer` je možné inicializovat fragment bez grafického rozhraní v kontejneru, který je uzavřený a na ničem jiném nezávislý. V inicializovaném fragmentu můžeme testovat chování fragmentu pomocí `fragmentScenario.onFragment{}`.

- **Testování perzistence** - V testovacím balíku `roomTests` se nachází testy, které ověřují čtení a zápis dat do databáze. Aby bylo dosaženou co největšího pokrytí kódy, probíhá testování na co nejvyšší úrovní. Typicky to bývá voláním metod vrstvy ViewModel či Service 5.1. Před každým testem si do databáze ukládám testovací data se kterými následně mohu pracovat.

- **Testování fragmentů** - V testovacím balíku `fragmentTests` se nachází testy, které ověřují chování jednotlivých fragmentů. Jedná se o skenování BLE i Sigfox zařízení, připojení k BLE zařízením, zpracování naměřených hodnot a jejich ukládání. Díky mockům a "shadow" objektům je možné otestovat části systému, které by se jen stěží dali testovat. Samozřejmě i zde občas potřebuji vložit do databáze nějaká testovací

data, ovšem primárně se tomuto věnuje testování perzistence. Pro simulaci odpovědí ze Sigfox API používám framework Mockito 4.4, kde si můžu definovat návratové hodnoty z metod v rozhraní `SigfoxDevicesContract`.

6.2 Testování s pomocí vývojové desky FRDM-KW41Z

Od společnosti NXP mi byli zapůjčeny dvě vývojové desky FRDM-KW41Z a jeden Sigfox modul OM2385/SF001. Díky tomuto hardwaru jsem mohl simulovat chování reálného měřícího zařízení a tím otestovat aplikaci jako celek.

FRDM-KW14Z

K této desce NXP nabízí zdarma ke stažení SDK[1]. Kit obsahuje spoustu příkladů jednoduchých a funkčních programů pro desku. Kromě jiného je zde i několik příkladů, ve kterých je deska v roli serveru a nabízí služby dostupné přes komunikační protokol BLE. Vycházel jsem z příkladu, který měl simulovat měření srdečního tepu. Hodnoty, které generuje vystavuje na GATT server, který je popsán v teoretické části 2.5.2.

Velmi jednoduše jsem program upravil, aby simuloval měření teploty. Nyní periodicky generuje náhodné hodnoty v rozmezí 0 až 100, které ihned vystavuje na GATT server pod vlastní službou a charakteristikou. Tyto hodnoty jsem zkoušel mobilní aplikací číst a tím aplikaci ladit. Samozřejmě jsem generované hodnoty postupně měnil, abych dokázal odladit všechny příslušné formáty dat, které aplikace umožňuje zpracovat.

OM2385/SF001

NXP bohužel nenabízí SDK k tomuto modulu na svých stránkách. Ovšem pan Nevoral ze společnosti NXP byl tak ochotný a zaslal mi program, který umí přes Sigfox síť odeslat libovolnou hodnotu. Tuto hodnotu poté mohu číst ze Sigfox API.

Opět jsem jednoduše program upravil, tak aby opět generoval nějaké náhodné hodnoty. Ty jsem postupně odesílal do sítě Sigfox a pomocí aplikace je načítal. To ve všech formátech, které aplikace dokáže u Sigfox zařízeni zpracovat. Takto jsem odladil postupně i načítání hodnot ze Sigfox API.

6.3 Testování v reálných podmínkách

Ve spolupráci s NXP jsem se domluvil, že mi spustí zařízení, které bude měřit reálná data a publikovat je do sítě Sigfox. Publikace všech těchto hodnot je vázána k jednomu zařízení. K rozlišení naměřených hodnot se využívají prefixy, jak jsou popsány v sekci 5.4. Jedná se o měření teploty, osvětlení a zrychlení s prefixy(1B prefix + 4B hodnota) popsánými v následující tabulce 6.1.

Zařízení běží 24 hodin denně na Brněnské pobočce NXP, kde periodicky měří a odesílá všechny zmíněné hodnoty každých 30 minut do sítě Sigfox. Kratší interval bohužel není možný jelikož maximální počet přenesených zpráv na jedno zařízení je 144 za 24 hodin. Pokud tedy odesíláme každých 30 minut tři zprávy dosáhneme přesně denního limitu zpráv. Získané hodnoty intenzity osvětlení a teploty můžeme vidět na obrázku 6.1. Je zde vyobrazen lineární graf vývoje intenzity osvětlení během dne 26.07. a sloupcový graf zobrazující

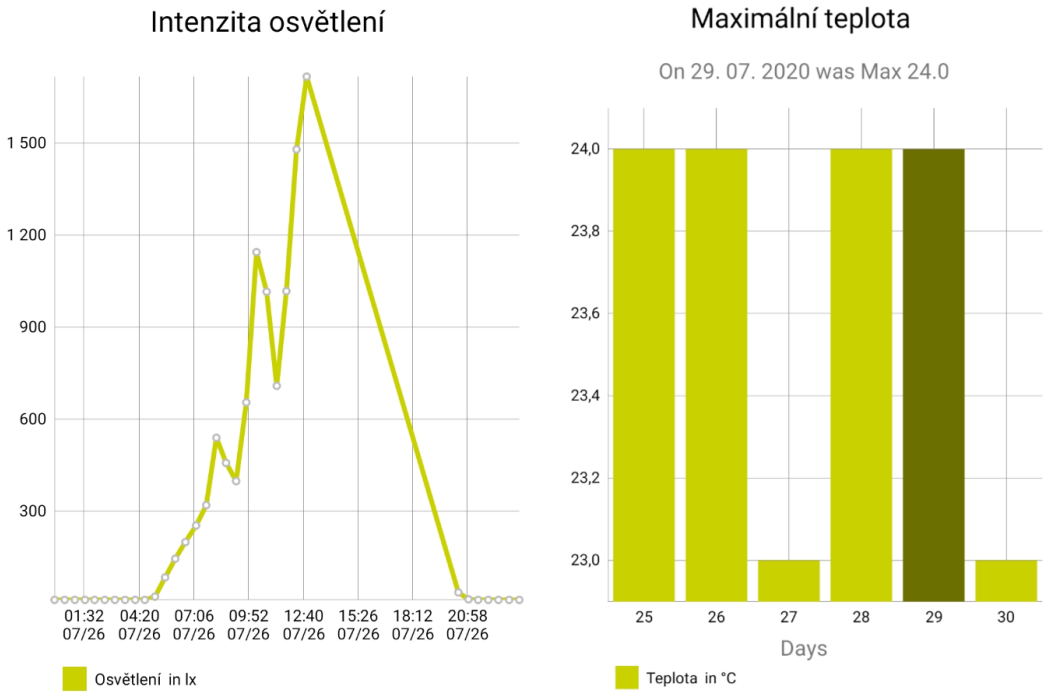
[1]Software Development Kit - Balíček nástrojů určených k vývoji softwaru na daném zařízení.

Hodnota prefixu	Název	Příklad hex hodnoty	Jednotka
1	teplota	0100000017	°C
2	zrychlení	020000000	mg
3	osvětlení	0300000836	lx

Tabulka 6.1: Výpis měřených veličin.

maximální teplotu za každý den v intervalu posledních 5 dní. Jelikož je teplota měřena ve vnitřních prostorech, nemůžeme pozorovat mezi jednotlivými dny velké rozdíly.

V rámci testování BLE zařízení v reálných podmínkách jsem chtěl využít teplotní senzor od výrobce Xiaomi, který jsem si k tomuto účelu zakoupil. Nicméně posílané údaje o teplotě jsou modifikovány a dokumentace k čidlu neobsahuje návod k jejich čtení z aplikací třetích stran. Poskytují svoji vlastní aplikaci pomocí které mohou hodnoty uživatelé vyčítat a vizualizovat. Tudíž jsem údaje o teplotě bohužel nemohl získat v čitelném formátu. Avšak údaje o stavu baterie se mi v čitelném formátu získat povedlo. Vyzkoušel jsem si tedy, že čtení ze zařízení funguje spolehlivě a není problém hodnoty z takových zařízení vyčítat. Vše záleží jen na použitém formátu.



Obrázek 6.1: Reálné naměřené hodnoty pomocí aplikace

Kapitola 7

Závěr

Cílem práce bylo vytvořit mobilní aplikaci pro operační systém Android, která umí získat hodnoty ze zařízení přes síť Sigfox a protokol Bluetooth Low Energy. Získané hodnoty má interpretovat uživateli dle jeho požadavků s možností zpracování ve statistických funkcích. Záměrem implementace bylo ukázat vhodnost použitých technologií v oblasti chytrého měření a sestrojit aplikaci pro firmu NXP. Ta by posléze měla fungovat jako show case pro ukázku vyčítání hodnot z jejich hardware. Jelikož existujících řešení, zejména pro vyčítání hodnot ze Sigfox API, není mnoho, mohla by si tato aplikace najít své místo jak pro technologické nadšence či běžné uživatele, kteří si chtějí zjednodušit život pomocí "chytrých" technologií.

Všechny výše uvedené požadavky se podařilo úspěšně splnit, implementovat a ověřit testováním. Aplikace bude vydána pod open-source licencí, aby nezůstala zapomenutá, ale mohla být dále rozvíjena.

Vždy jsem chtěl vyvinout aplikaci na mobilní telefon. Bohužel jsem v době studia na to neměl dostatek prostoru. Jsem rád, že jsem si mohl v rámci mojí práce konečně vývoj vyzkoušet. Chtěl bych i nadále aplikaci rozvíjet a vylepšovat, protože odevzdáním nepovažuji aplikaci za navždy uzavřenou. Je zde spousta možných vylepšení, které bych velmi rád implementoval.

Naučil jsem se novým technologiím, které se mi určitě budou v budoucnu velice hodit. Vývoj Android aplikací bude ještě dlouhou dobu žhavé téma. Jsem rád, že jsem si vybral větší výzvu a pustil se do implementace v programovacím jazyce Kotlin, protože tento jazyk začíná nabírat na popularitě a jeho znalost se mi bude více než hodit.

Literatura

[1] *Callback API* [online]. Sigfox S.A. Dostupné z: https://backend.sigfox.com/apidocs/callback.

[2] *OUR STORY* [online]. [cit. 2020-04-06]. Dostupné z: https://sigfox.cz/cs/o-nas.

[3] *Platform Architecture* [online]. Google Developers. Dostupné z: https://developer.android.com/guide/platform.

[4] *Sigfox Technical Overview* [online]. Sigfox S.A. Dostupné z: https://www.disk91.com/wp-content/uploads/2017/05/4967675830228422064.pdf.

[5] Adegbija, T., Lysecky, R. a Kumar, V. Right-Provisioned IoT Edge Computing: An Overview. In:. Květen 2019, s. 531–536. DOI: 10.1145/3299874.3319338.

[6] Afaneh, M. *Intro to Bluetooth Low Energy* [online]. Novel Bits, 2018. ISBN 978-1790198153. Dostupné z: https://www.novelbits.io/introduction-to-bluetooth-low-energy-book/.

[7] element14, F. *Wireless solutions 5 SigFox - Fulfilling an IoT communications need* [online]. Farnell, únor 2018 [cit. 2020-04-06]. Dostupné z: https://cz.farnell.com/fulfilling-an-iot-communications-need.

[8] Götz, C. MQTT 101 – How to Get Started with the lightweight IoT Protocol. [online]. 2014. Dostupné z: https://www.eclipse.org/community/eclipse_newsletter/2014/october/article2.php.

[9] Hossain, M. T. Android Application Architecture. [online]. Srpen 2017. Dostupné z: https://medium.com/oceanize-geeks/android-application-architecture-189b4721c7c5.

[10] H.Tschofenig, D. T. D. M. *Architectural Considerations in Smart Object Networking* [Internet Requests for Comments]. RFC 7452. RFC Editor, květen 2015. Dostupné z: https://tools.ietf.org/html/rfc7452.

[11] Jordi Salazar, S. S. *Internet věcí*. České vysoké učení technické v Praze Fakulta elektrotechnická, edition =.

[12] Manjoo, F. A Murky Road Ahead for Android, Despite Market Dominance. [online]. Květen 2015. Dostupné z: https://www.nytimes.com/2015/05/28/technology/personaltech/a-murky-road-ahead-for-android-despite-market-dominance.html.

[13] Marcin Moskala, I. W. *Android Development with Kotlin* [online]. Packt Publishing Ltd., 2017. ISBN 978-1-78712-368-7. Dostupné z: https://www.shabakeh-mag.com/sites/default/files/files/attachment/1397/04/1530550032.pdf.

[14] Martí, M., Garcia Rubio, C. a Campo, C. Performance Evaluation of CoAP and MQTT SN in an IoT Environment. *Proceedings*. MDPI AG. 2019, sv. 31, č. 1, s. 49. ISSN 2504-3900.

[15] noviny, H. Chytrá síť Sigfox je v Česku hotová. [online]. 2019. Dostupné z: https://ictrevue.ihned.cz/c3-66631940-0ICT00_d-66631940-chytra-sit-sigfox-je-v-cesku-hotova.

[16] Pawar, P. a Trivedi, A. Device-to-Device Communication Based IoT System: Benefits and Challenges. *IETE Technical Review*. Taylor Francis. 2019, sv. 36, č. 4, s. 362–374. ISSN 0256-4602. Dostupné z: http://www.tandfonline.com/doi/abs/10.1080/02564602.2018.1476191.

[17] Romeika, C. *Reinvented connectivity! Pangea's latest addition called Sigfox* [online]. Pangea, květen 2018 [cit. 2020-04-06]. Dostupné z: https://pangea-group.net/2018/03/28/reinvented-connectivity-pangeas-latest-addition-called-sigfox/.

[18] Vojáček, A. SIGFOX - princip, struktura, protokol, použití. [online]. Květen 2017. Dostupné z: https://vyvoj.hw.cz/sigfox-princip-struktura-protokol-pouziti.html.

Příloha A

Obsah paměťového média

- `/android-sources` - Zdrojové kódy Android aplikace
- `/debug-sources/Ble` - Zdrojové kódy ladícího programu pro desku FRDM-KW41Z pro odesílání dat přes protokol Bluetooth Low Energy
- `/debug-sources/Sigfox` - Zdrojové kódy ladícího programu pro desku FRDM-KW41Z s modulem OM2385/SF001 pro odesílání dat do sítě Sigfox
- `/thesis-sources` - Zdrojové kódy technické zprávy
- `/android-binaries` - Spustitelné soubory mobilní aplikace
- `/debug-binaries` - Spustitelné soubory ladících programů
- `/BachelorThesis.pdf` - PDF formát tohoto dokumentu
- `/Readme.md` - Návod na instalaci

Příloha B

Schéma lokální SQLite databáze