VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATICKÉ TESTOVÁNÍ SOFTWARE

AUTOMATIC TESTING OF SOFTWARE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MATEJ MRÁZIK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. PAVOL KORČEK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Mrázik Matej**
Program: Informační technologie
Název: **Automatické testování software**
Automatic Testing of Software
Kategorie: Softwarové inženýrství

Zadání:

1. Seznamte se s existujícími systémy, které umožňují automatické testování software a vybraný systém podrobněji nastudujte.
2. Za účelem testování nastudujte doporučený open-source software, a to zejména jeho rozhraní a funkce.
3. Pro daný software navrhněte několik testů, pokud možno po konzultaci s autory daného software.
4. Implementujte vybrané testovací případy a integrujte je do prostředí pro automatické testování.
5. Diskutujte dosažené výsledky a zhodnoťte další možnosti rozšíření práce.

Literatura:

- Dle pokynu 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: **Korček Pavol, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 25. října 2019

Abstrakt

Hlavným cieľom bakalárskej práce je automatizácia testovania softvéru IQRF Gateway Daemon. Praktickým výstupom je nástroj schopný testovať IQRF Gateway Daemon prostredníctvom emulovanej virtuálnej siete inteligentných zariadení. V teoretickej časti práca načtrne čitateľovi problematiku testovania softvéru. Uvedené princípy sú následne aplikované pri testovaní IQRF GW Daemon. Čitateľ sa tak zoznámi s výsledným nástrojom a jeho funkcionalitou, ktorú bude môcť v prípade potreby ďalej rozširovať.

Abstract

The main goal of the bachelor thesis is to automate the testing of IQRF Gateway Daemon software. The practical output is a tool capable of testing the IQRF Gateway Daemon through an emulated virtual network of intelligent devices. In the theoretical part, the work outlines the issues of software testing to the reader. These principles are then applied in testing the IQRF GW Daemon. The reader will get acquainted with the resulting tool and its functionality, which will be able to further expand if necessary.

Kľúčové slová

Testovanie, verifikácia, validácia, chyba, defekt, princípy testovania, proces testovania, úrovne testovania, Tavern, IoT, IQRF, IQRF OS, DPA, IQRF GW Deamon.

Keywords

Testing, verification, validation, error, defect, testing principles, testing process, testing levels, Tavern, IoT, IQRF, IQRF OS, DPA, IQRF GW Deamon.

Citácia

MRÁZIK, Matej. *Automatické testování software*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Pavol Korček, Ph.D.

Automatické testování software

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Pavla Korčeka, Ph.D. Ďalšie informácie mi poskytli Ing. František Mikulu, Ing. Rostislav Špinar, Roman Ondráček a Karel Hanák. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Matej Mrázik
3. júna 2020

Poďakovanie

Ďakujem vedúcemu bakalárskej práce Ing. Pavlovi Korčekovi, Ph.D. za účinnú metodickú, pedagogickú a odbornú pomoc pri spracovaní mojej bakalárskej práce. Moje poďakovanie taktiež patrí externým konzultantom z firmy IQRF Tech s.r.o a to konkrétne Ing. Františkovi Mikule a Ing. Rostislavovi Špinarovi za odbornú pomoc a za dodanie potrebnej technológie. Veľká vďaka patrí aj Romanovi Ondráčkovi za jeho technické rady a za dodanie potrebných dát. Nakoniec sa chcem poďakovať Karelovi Hanákovi za konzultácie.

Obsah

Kapitola 1

Úvod

Rozvoj informačných technológií a konkurencia na trhu prispeli k zlepšeniu kvality riešení. Je to z dôvodu, že sme vo fáze, keď má zákazník na výber medzi viacerými produktmi, ktoré spĺňajú jeho požiadavky. Preto dodávanie funkčného softvéru zákazníkom je rozhodujúce pre úspech produktu. Ak softvér nefunguje správne, je pravdepodobné, že si ho zákazník nekúpi. Taktiež náklady spojené s nápravou škôd vzniknutých v dôsledku nedostatočného testovania vo fáze vývoja často dosahujú súm ako náklady na bezprostredný vývoj. Z týchto dôvodov je testovanie softvéru čoraz častejšie súčasťou vývojového cyklu softvérov.

Testovanie je časovo náročná disciplína a veľa z úloh môže byť vykonaných pomocou automatických nástrojov. Moja bakalárska práca si dáva za cieľ zautomatizovať testovací proces softvéru IQRF GW Daemon. Taktiež je cieľom umožniť, aby tento proces mohol byť vykonaný bez potreby pripojenia fyzickej siete inteligentných zariadení. Vývojári by mali možnosť si jednoducho zostaviť virtuálnu sieť podľa ich požiadaviek. Dôležité je, aby implementácia bola zrozumiteľná pre možné rozšírenia v budúcnosti. Vývojári, ktorí sa podieľajú na vývoji softvéru IQRF GW Daemon, robili toto testovanie ručne a nepravidelne. Taktiež nemali stanovené ani pravidlá testovania, takže nebolo jasné, ktorá funkcionalita je momentálne otestovaná. Preto vznikla požiadavka na nástroj, ktorá by testovanie sprehľadnila a zautomatizovala.

Túto prácu som si vybral s cieľom zvýšiť povedomie o tom, aké výhody má zavedenie správnych testovacích postupov.

Nasledujúca kapitola 2 obsahuje úvod do problematiky testovania softvéru. Vysvetľuje základné pojmy, ktoré je potrebné pochopiť, ak sa chceme baviť o testovaní softvéru. Nakoniec sa podrobnejšie zameriava na automatické testovanie. V kapitole 3 sú popísané testovacie nástroje. Najskôr sú popísané ich všeobecné výhody a nevýhody. Kapitola zároveň opisuje ich možné využitie v mojej bakalárskej práce. Kapitola 4 zahŕňa krátky úvod do internetu vecí a venuje sa softvéru IQRF GW Daemon, ktorého testovanie je cieľom mojej práce. Kapitola 5 upresňuje požiadavky na výsledné riešenie a navrhuje postup, ako by mala byť práca implementovaná. Kapitola 6 vysvetľuje, akým spôsobom je realizovaná celková implementácia a následne opisuje spôsob implementácie jednotlivých častí. Posledná kapitola 7 zhŕňa dosiahnuté výsledky a poukazuje, aké vylepšenia sú možné v budúcnosti.

Kapitola 2

Testovanie

Táto kapitola obsahuje úvod do problematiky testovania softvéru. Postupne popisuje základné princípy testovania, typy testov a testovacie prostredia.

V dnešnej dobe, keď používame aplikácie každý deň, sa veľa z nás stretlo s nejakou chybou v aplikácií. Aj keď si to neuvedomujeme, každý z nás denne niečo testuje. Možno vám to príde zvláštne, že ako môžete testovať softvér. Na to je jednoduchá odpoveď. Pri používaní aplikácie vykonávate nejakú činnosť a očakávate, že softvér vám odpovie podľa vašich očakávaní. V skratke sa dá povedať, že sa uisťujeme a vyhodnocujeme, či je vec v poriadku alebo nie. Tento popis je veľmi laický a v prípade, keď pojednávame o testovaní softvéru, proces je oveľa náročnejší a komplexnejší.

Softvérové testovanie je proces vykonávania alebo vyhodnocovania systému manuálnymi alebo automatickými prostriedkami. Cieľom je overiť, či spĺňa stanovené požiadavky, alebo identifikovanie rozdielov medzi skutočnými a očakávanými výsledkami [6].

2.1 Verifikácia, validácia

Testovanie sa často spája s pojmami validácia a verifikácia. Oba pojmy v terminológii testovania znamenajú niečo úplne odlišné. I keď rozdiel medzi nimi môže byť pomerne nejasný. V rôznych literatúrach sú tieto pojmy definované inak. Ja som vybral definície, s ktorými sa najviac stotožňujem.

Verifikácia je overenie/preskúmanie pravosti a správnosti produktu. Na verifikáciu si môžeme položiť otázku, či vytvárame produkt správne. Validácia je zase overenie správnosti produktu alebo služby vzhľadom k požiadavkám. Validácia poskytuje odpoveď na otázku, či vytvárame správny produkt [46]. Inými slovami, tester kontroluje, či výstupy vývoja sú relevantné alebo definované pre danú úroveň vývoja. Ako rozdeľujeme testovací proces, bude rozobrané v kapitole 2.4. Čím viac sa blížime ku koncu vývoja aplikácie, tým sa dáva väčšie úsilie na zistenie validácie produktu. Je to z dôvodu, že keď ukončujeme vývoj aplikácie, tak sa nemusíme už zameriavať toľko na otázku, či vytvárame produkt správne. Naša pozornosť sa sústreďuje na odpoveď, či vytvárame správny produkt [3].

2.2 Chyby, defekty, zlyhania

V tejto kapitole si povieme rozdiel medzi pojmami chyba, defekt a zlyhanie. Chybu môže urobiť človek, a tým zapríčiniť vznik defektu v softvérovom kóde alebo v nejakom inom súvisiacom pracovnom produkte. Chyba, ktorá spôsobí zanesenie defektu do jedného pra-

covného produktu, môže spôsobiť inú chybu. Tá má zas za následok zanesenie iného defektu do súvisiaceho produktu.

Ak dôjde ku spusteniu defektu v kóde, ten môže spôsobiť zlyhanie, ale aj nemusí. Môže sa v produkte objaviť defekt, ktorý ale aby spôsobil zlyhanie, musí mať na vstupe špecifické parametre, s ktorými sa systém tak často nestretne.

Príčin, prečo vznikajú defekty, je mnoho. Asi najlogickejšou príčinou je omyl človeka, keďže ľudia sú omylní. V posledných rokoch sa do popredia ale dostáva aj ďalšia príčina a tou je komplexnosť a robustnosť dnešných softvérov. V dnešnej dobe nám jeden systém ponúka skoro neobmedzené možnosti, čo s ním môžeme robiť a tam vzniká problém. Už veľmi dlho nie je možné kompletné testovanie všetkých kombinácií vstupných a výstupných parametrov. Preto sa zaviedli princípy testovania, ktoré budú opísané v ďalšej kapitole.

2.3 Základné princípy testovania

Aby testovanie bolo čo najúčinnejšie, vznikli základné princípy testovania, ktorých sa každý tester snaží držať [9].

Testovanie ukazuje prítomnosť defektov

Testovanie ukazuje prítomnosť defektov, ale nie ich neprítomnosť. Testovanie nemôže dokázať, že v softvéri nie sú žiadne defekty, len znižuje pravdepodobnosť neobjavených defektov. Treba si zapamätať, že nájdenie defektu nie je dôkazom bezchybnosti. Preto ak očakávame, že program funguje správne, tak je väčšia pravdepodobnosť, že nejakú chybu prehliadneme. Na druhej strane, ak očakávame nejaké chyby, tak ich nájdenie je viac pravdepodobné [8].

Vyčerpávajúce testovanie je nemožné

Kompletné testovanie nie je možné. Preto je veľmi potrebné spraviť si analýzu rizík, stanoviť si priority a podľa toho zamerať naše testovacie úsilie.

Včasné testovanie

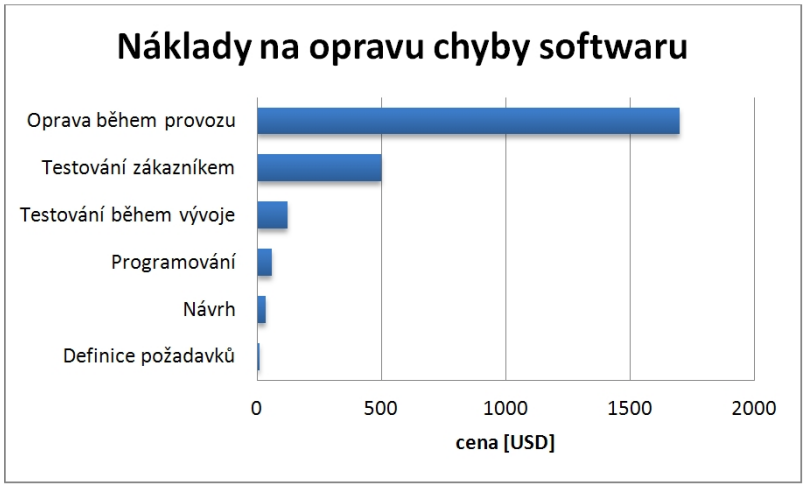
Včasné testovanie šetrí čas a peniaze. Preto je dôležité, aby testovacie aktivity boli zapojené čím skôr do procesu vývoja softvéru. Čím skôr sa chyba opraví, tým jej oprava bude menej finančne náročná. Toto tvrdenie je demonštrované aj na obrázku 2.1.

Zhlukovanie defektov

Je princíp, ktorý nám hovorí, že kde nájdeme jeden defekt, tam je vyššia pravdepodobnosť, že nájdeme aj ďalšie. V softvéri väčšinou malé množstvo modulov obsahuje väčšinu všetkých defektov.

Pesticídny paradox

Ak opakujeme rovnaké testy stále dookola, po nejakom čase tieto testy už nenájdu žiadny nový defekt. Tento názov vznikol preto, lebo testy už nie sú efektívne pri hľadaní defektov. Podobne ako aj pesticídy nie sú po istom čase efektívne pri hubení hmyzu. Aby sme predišli pesticídnemu efektu, tak musíme pravidelne revidovať testy a písať nové.



Obr. 2.1: Náklady na opravu chyby v software
[32]

Testovanie je závislé na kontexte

Súvisí s tým, že rôzny softvér môže mať rôzne využitie. Aplikácia, ktorá je určená na chytré hodinky sa bude testovať inak, ako aplikácia určená na mobil. Napríklad testovanie internetového bankovníctva bude viac zamerané na bezpečnosť ako e-shop, ktorý sa bude viac testovať na prívetivosť k užívateľovi.

Neprítomnosť chyb je klam

Nájdenie a opravenie defektov nepomôže, pokiaľ vytvorený softvér je nepoužiteľný a nespĺňa potreby a očakávania užívateľov. Úspešný systém musí spĺňať požiadavky zákazníka a očakávania užívateľov.

2.4 Úrovne testovania

Základné úrovne testov sú definované ako súbor testov, ktorými je softvér testovaný na danom stupni podrobnosti. Podľa toho, v akej fáze a s akým časovým odstupom od písania kódu sa testovanie prevádza, ho delíme do štyroch základných úrovní [13]. Pre každú úroveň testov si definujeme všeobecné ciele, objekt testovania, typické defekty a zlyhania.

Testovanie jednotiek

Pre túto úroveň testovania sa používa aj pomenovanie testovanie komponent alebo *unit testing*. Zameriava sa na komponenty (jednotky), ktoré sú samostatne testovateľné. U objektovo orientovaného programovania sa jedná o testovanie jednotlivých tried a metód. Testy týchto jednotiek sa zapisujú vo forme programového kódu, a preto ich spravidla obsluhujú vývojári. Testovanie jednotiek je veľmi dôležité. Je potrebné ho zaviesť hneď na začiatku vývoja softvéru, lebo sa veľmi ťažko aplikujú už v zabehnutých projektoch. Zistené chyby sú opravené okamžite, takže nie je potrebný žiadny *defect tracking* (zaznamenanie defektu).

Integračné testovanie

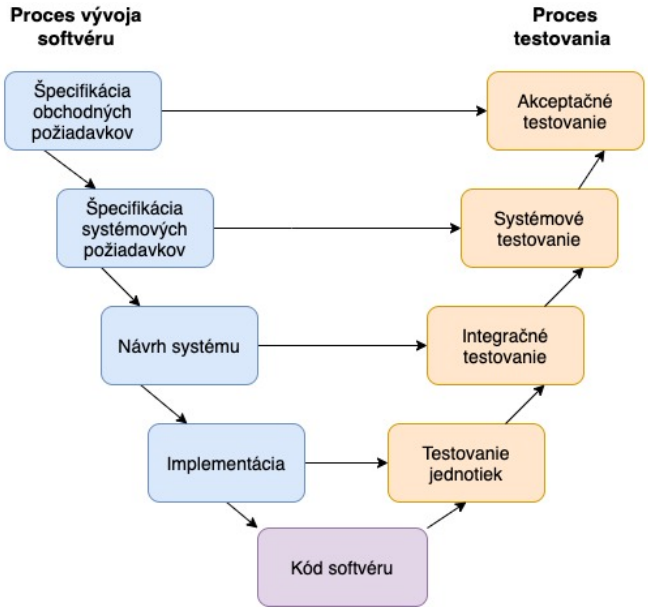
V tejto fáze už nastupuje testovací tím. Ten overuje bezchybnú komunikáciu a integráciu medzi jednotlivými komponentami v systéme alebo medzi systémami. Integračné testovanie má dve úrovne. Integračné testovanie komponentov sa odohráva po testovaní komponentov a je všeobecne automatizované. Druhá úroveň je systémové integračné testovanie, ktoré sa zameriava na rozhrania medzi systémami, súbormi programov a mikroslužbami. Pri menších projektoch sa môžu integračné testy vynechať. Na defekty, na ktoré by prišli integračné testy, sa príde v ďalších fázach testovania. Ale nesmieme zabúdať, že stále platí, čím skôr objavíme defekt, tým menej úsilia a peňazí nás jeho oprava bude stáť.

Systémové testovanie

Keď overíme, že integrita medzi komponentami je v poriadku, prejdeme na testovanie celého systému alebo produktu. Overujeme funkčnosť systému ako celku podľa pripravených scenárov simulujúcich situácie, ktoré v praxi môžu nastať. Systémové testovanie by sa malo sústrediť na celkové *end-to-end* správanie systému, a to funkcionálne aj nefunkcionálne. Testovanie by malo prebiehať v prostredí, ktoré prinajlepšom bude rovnaké ako produkčné prostredie aplikácie. Keď sa v tejto fáze objaví chyba, je opravená a opätovne otestovaná. Preto sa aj tu využívajú automatizované testy na scenáre, ktoré budú najbežnejšie. Výsledkom systémových testov je výstupná kontrola softvéru, ktorá pomáha pri rozhodovaní, či môžeme odovzdať systém zákazníkovi. Ten na ňom prevedie poslednú fázu testov.

Akceptačné testovanie

Je posledná úroveň testov na strane zákazníka. Testovanie prebieha v produkčnom prostredí, kde ho testujú testeri od zákazníka. Počas testovania sa môžu objaviť nejaké defekty, ktoré je dôležité, čo najrýchlejšie opraviť. Avšak cieľom tohto testovania nemusí byť vždy nájdenie defektov. Často môže tiež spĺňať právne alebo regulačné požiadavky, štandardy. Z vlastnej skúsenosti môžem povedať, že je veľmi dôležité si dopredu určiť, akou formou budú oznamované chyby od zákazníka a aká chybovosť softvéru je prípustná.



Obr. 2.2: Úrovne testovania

2.5 Rozdelenie testov

Funkcionálne testovanie

Funkcionálne testovanie je typ softvérového testovania, kde je systém testovaný proti funkcionálnym požiadavkám/špecifikáciám. Testujú sa funkcie tak, že do nich vložíme vstup a preskumávame výstup. Tento typ testovania sa nezameriava na spracovanie, ale skôr na výsledok spracovania. Simulujeme ním skutočné využitie systému, ale neprináša nám žiadne predpoklady správnosti štruktúry systému. Keďže sa uvažuje o správaní softvéru, tak z tohto dôvodu môžu byť využité postupy testovania čiernej skrinky. Tie budú bližšie vysvetlené neskôr v tejto kapitole. Funkcionálne testovanie sa najviac využíva na úrovni systémového a akceptačného testovania.

Aby sme dosiahli najvyššiu efektivitu funkcionálneho testovania, musíme podmienky testu vytvoriť priamo z požiadaviek používateľov daného softvéru alebo od zákazníka. Ak by sme vytvorili požiadavky zo systémovej dokumentácie, neprišli by sme na chyby v dokumentácii, a to by mohlo mať vplyv na kvalitu softvéru [10].

Nefunkcionálne testovanie

Je typ testovania, ktorý je definovaný ako softvérové testovanie, ktoré kontroluje nefunkcionálne aspekty aplikácie. Jednoduchšie povedané, je to testovanie, kde sa zisťuje, ako dobre sa systém správa. Nefunkcionálnymi aspektami môžeme rozumieť bezpečnosť, spoľahlivosť, dostupnosť, použiteľnosť, škálovateľnosť, efektívnosť aplikácie. Je odporúčané, aby sa nefunkcionálne testovanie vykonávalo na všetkých úrovniach testovania. Malo by sa realizovať čo najskôr, lebo neskorá identifikácia nefunkcionálnych defektov môže byť často extrémne nebezpečná pre úspešné dokončenie projektu.

Návrh nefunkcionálneho testovania a samotné nefunkcionálne testovanie môže zahŕňať špeciálne zručnosti alebo aj znalosti. Medzi tieto znalosti môžu patriť znalosti o zraniteľnosti spojenia s konkrétnymi programovacími jazykmi.

Pre lepšie pochopenie tohto typu testov si môžeme predstaviť test na aplikácii, kde zisťujeme, aký počet ľudí sa môže naraz prihlásiť do systému [34].

Testovanie bielej skrinky

Testovanie bielej skrinky je tiež známe ako *white box, glass box, open box* alebo štrukturálne testovanie. Testovanie je založené na vnútornej štruktúre alebo implementácii systému. Vnútornou štruktúrou rozumieme kód, architektúru alebo pracovné toky. Pri tomto type testovania má tester prístup k zdrojovému kódu a k príslušnej dokumentácii. Testovanie bielej skrinky prebieha buď automatizovane alebo ručne. V praxi sa však veľmi často tieto dva spôsoby kombinujú. Návrh a vykonávanie môže zahŕňať špeciálne zručnosti alebo vedomosti. Napríklad, akým spôsobom je kód zostavený, ako sú dáta uložené a ako používať nástroje pokrytia a správne interpretovať ich výsledky.

Využitie white box testovania je v ranných fázach vývojového cyklu, kde môže tester a vývojár spolupracovať na odhaľovaní chýb. Pri tomto type testovania sa často nájde nejaký nežiaduci kód alebo bezpečnostná chyba. Nájdenie nežiadúceho kódu sa považuje za najväčšiu výhodu testovania bielej skrinky. Aj keď je v aplikácii nežiadúci kód, aplikácia môže pracovať správne. Na druhej strane výsledkom jeho činnosti môže byť neoprávnený prístup do aplikácie, prevod finančných prostriedkov a veľa ďalšieho. Do aplikácie sa môže dostať neúmyselne pri ladení, napr. keď si vývojár nechal zapisovať niečo na výstup a zabudol túto funkcionalitu odstrániť z kódu [49].

Testovanie čiernej skrinky

Black box, closed box sú ďalšie názvy pre testovanie čiernej skrinky. Je to funkčné testovanie realizované bez znalosti vnútornej, dátovej a programovej štruktúry. To znamená, že oproti testovaniu bielej skrinky, tester nemá prístup k dokumentácií a k zdrojovému kódu aplikácie. Pri tomto testovaní sa využívajú testovacie scenáre, ktoré buď vytvorí tester sám, alebo sú mu poskytnuté. Taktiež sa využíva kombinácia automatizovaného a manuálneho testovania. Testovanie čiernej skrinky sa využíva tam, kde sú presne definované vstupné a výstupné parametre. Keď tester zadá konkrétnu hodnotu na vstupe, tak vie, akú hodnotu má očakávať na výstupe. Keď výstupná hodnota sa nezhoduje s očakávanou, predpokladáme, že v aplikácií je chyba.

Black box testovanie demonštruje chybu, ktorá bola objavená testovaním bielej skrinky. K testovaniu stačí mať k dispozícii len daný program. Ak poznáme jeho umiestnenie, môžu byť testy vykonané aj na diaľku po sieti. Výhodou tohto testovania je jednoduchosť jeho prevedenia. Tester vôbec nemusí mať znalosť programovacieho jazyka a v krátkom období je možné otestovať aj rozsiahle aplikácie. Zase na druhej strane, keď na výstupoch sa nám zobrazujú očakávané hodnoty, tak to neznamená, že aplikácia je napísaná správne a je efektívna. Tiež nevieme odhaliť, či aplikácia nevykonáva iné akcie, ktoré nie sú očakávané a nie sú špecifikované [47].

Testovanie šedej skrinky

Je testovanie, ktoré predpokladá obmedzenú znalosť vnútornej štruktúry a dátového toku aplikácie. Ako sa dá predpokladať z názvu, ide o kombináciu testovania čiernej skrinky a bielej skrinky. Nejde o testovanie čiernej skrinky, lebo tester má vedomosti o nejakých vnútorných štruktúrach. Ale nejedná sa ani o testovanie bielej skrinky, lebo znalosti vnútornej štruktúry nie sú dostatočné. Testovanie prebieha zvonku ako u testovania čiernej skrinky.

Tester ale vie, ako produkt funguje a vďaka tomu ho môže lepšie otestovať. Dokáže lepšie zostaviť testovacie scenáre.

Jeho hlavné využitie je pri testovaní rozhrania, ktoré je medzi dvoma modulmi od rôznych užívateľov. Ďalšie typické použitie je v podobných prípadoch ako pri testovaní čiernej skrinky, kde zadávame vstupné hodnoty a očakávame adekvátne výstupy. Navyše sa však pozeráme na hodnoty, ktoré sa ukladajú do databázy aplikácie [48].

Statické testovanie

Statické testovanie je testovanie, kam patria techniky, ktoré sú založené na manuálnom preskúmaní pracovných produktoch, alebo na hodnotení kódu, alebo iných pracovných produktov pomocou nástrojov. Statické testy nevyžadujú beh softvéru. Preto sa hlavne využívajú v počiatočných fázach životného cyklu softvéru, keď ešte nie je vytvorený ani funkčný prototyp aplikácie. Statické testovanie taktiež možno použiť pred začiatkom písania kódu, a to na kontrolu požiadaviek a na analýzu zdrojového kódu.

Analýza kódu, inak aj statická analýza, je dôležitá hlavne pre bezpečnostné počítačové systémy z odvetví letectva, medicíny, armády. Statická analýza býva súčasťou systémov na automatické zostavenie aplikácií. Primárne využitie je počas procesu agilného vývoja, pri ktorom sa používa distribuovaný systém riadenia revízií.

Hlavnou výhodou statického testovania je, že odhaľuje chyby už na začiatku životného cyklu vývoja. Tým pádom odhaľuje chyby ešte pred vykonaním dynamického testovania. Preto je oprava defektov, ktoré sa odhalia statickým testovaním, lacnejšia. V posledných rokoch firmy kladú čím ďalej väčší dôraz na statické testovanie. Zistili, že keď porovnajú úsilie na vykonanie týchto testov a cenu opravy, tak sa im to finančne oplatí. Statické testovanie taktiež dokáže odhaliť defekty, ktoré nevyvolávajú zlyhanie softvéru.

Dynamické testovanie

Dynamické testovanie je presný opak statického testovania, takže vyžaduje spustenie softvéru. Aj keď sú to dva rozdielne typy testov, ich ciele môžu byť rovnaké. Pre vykonávanie testov je potrebný aspoň spustiteľný prototyp softvéru. Kvôli tomu sa dynamické testovanie nasadzuje v neskorších fázach vývoja a je zamerané na prevádzku softvéru. Cieľom dynamického testovania je nájdenie zlyhania. Typy defektov, ktoré najčastejšie detekuje, sú chyby vo funkcii a zlé kvalitatívne charakteristiky. Zameriava sa na navonok viditeľné správanie. To kontroluje, či sa zhoduje so správaním, ktoré je popísané v špecifikácii.

Keďže sa dynamické testovanie zapája neskôr v životnom cykle softvéru, odstránenie chýb je drahšie. Na druhej strane vďaka neskoršiemu testovaniu máme komplexnejší pohľad na softvér.

2.6 Proces testovania

Plánovanie testov

Začať testovací proces bez plánu nie je dobrý nápad, ak chceme dosiahnuť cieľ. Preto prvý a najdôležitejší proces v softvérovom testovaní je plánovanie testov. Dobré testovanie je založené na dobrom testovacom pláne, ktorý si pripravíme. V tejto fáze sa definuje stratégia testovania, rozsah, testovacie nástroje, rozpočet. Plán testov sa môže prepracovať na základe spätnej väzby [39].

Analýza a návrh testov

Keď je testovací plán kompletný, tak ešte pred implementáciou je potrebná analýza a návrh testov. Analýza určuje, čo sa bude testovať. Návrh testov má na starosti, ako sa bude softvér testovať. Inými slovami sa vytvárajú testovacie podmienky a testovacie prípady. V tomto procese sa vykonávajú hlavne úlohy, ako je preskúmanie základu testovania, identifikácia testovacích podmienok na základe analýzy testovaných položiek, písanie testovacích prípadov [2].

Implementácia a vykonávanie testov

Väčšina ľudí si pod pojmom testovanie softvéru predstaví fázu vykonávania testov. Ako môžeme ale vidieť, tak pred vykonávaním testov sú dve inicializačné fázy a ďalšie dve ukončovacie fázy [40]. Implementácia má na starosti, či máme všetko potrebné, aby sme mohli začať testovať. Tým môžeme rozumieť dokončenie prípravy testovacieho prostredia potrebného pre vykonávanie testov.

Vyhodnotenie testovania a výstupných kritérií

Pri ukončení testovania sa hodnotia výstupné kritériá. Tie sa líšia od projektu ku projektu. Poznáme viac kritérií ukončenia testovania. Medzi tieto kritériá patrí miera chybovosti, či v systéme nie sú žiadne kritické problémy a na záver percentuálna úspešnosť testovacích prípadov. Na základe týchto kritérií sa rozhodne, či sú potrebné dodatočné testy. Posledným krokom je spísanie výsledkov.

Ukončenie testovania

Je posledná fáza v testovacom procese. Spisuje sa záverečná správa o incidentoch, kontroluje sa správnosť všetkej dokumentácie, archivácia testovacieho softvéru, testovacieho prostredia a infraštruktúry. Aktivity ukončenia testovania sa vyskytujú v projektových míľnikoch. Ako napríklad, keď sa vydá systém, dokončí sa projekt testovania, dokončí sa iterácia agilného projektu.

Obr. 2.3: Proces testovania
[39]

2.7 Automatické testovanie

V poslednej dobe sa automatické testovanie stáva čím ďalej populárnejším. A to preto, že prináša možnosť zrýchliť otestovanie základných funkcionalít v priebehu celého vývoja. Ďalším dôvodom je, že pri opakovanom testovaní funkcionalít je potrebné zakaždým vynaložiť

rovnaké úsilie, nároky na zdroje. Automatické testy sú napriek tomu znovupoužiteľné a je možné ich opakované spúšťanie.

Pred zavedením automatizovaného testovania na nejaký projekt treba rátať s tým, že počiatočné náklady na zavedenie sú dosť vysoké. Avšak po ich vytvorení je už beh automatických testov finančne málo náročný. Nesmieme však zabudnúť, že automatizáciu nemôžeme použiť na úplne celý projekt. Vždy sa nájdu nejaké prípady, kde sa viac oplatí použiť manuálne testovanie. Takým príkladom, kde sa oplatí manuálne testovanie, môže byť testovanie zložitého grafického rozhrania, kde jeho automatizácia by bola finančne veľmi náročná.

Hlavným cieľom automatizovaného testovania je práve časová úspora a taktiež zefektívnenie a zjednodušenie procesu vykonávania testovania softvéru. Môže sa zdať, že práca testera nie je vôbec potrebná. Stačí spustiť pripravené testy a testovacie nástroje všetko vykonajú samé. A to dokonca rýchlejšie a efektívnejšie. Ako všetko aj automatické testovanie má svoje nevýhody.

Automatizovať testy je vhodné predovšetkým regresné testy a *smoke testy*. To sú prevažne testy, ktoré spúšťame veľakrát a sú nemenné. Tieto testy zároveň ešte pokrývajú časť aplikácie, ktorá sa často mení.

Ako pri realizácii manuálnych testov, aj pri realizácii automatizovaných testov, je nutné zaistiť niektoré podmienky v postupe testovania softvéru. Logickou a najjednoznačnejšou podmienku je, že na existujúci program, ktorý už je v prevádzke, je veľmi náročné aplikovať automatizované testy. Preto je najlepšie tieto testy vytvárať už pri vývoji softvéru.

Vždy sa snažíme pokryť testami čo najväčšiu časť aplikácie/kódu. Testovací proces neautomatizujeme pri aplikáciach, ktoré majú krátky životný cyklus vývoja. Oplatí sa to u aplikácií, ktorých životnosť bude minimálne aspoň rok a budú mať viac verzií. U automatizovaných testov sa pokrýva hlavne časť programu, kde hrozí existencia závažných chýb. Pri posudzovaní výhod a nevýhod použitia automatizovaných testov je vždy nutné zahrnúť vlastnosti daného projektu, kde majú byť použité. To preto, lebo každý projekt má svoje špecifiká a tie môžu hrať zásadnú rolu pri možnostiach využitia automatizácie.

Nevýhodou je, že automatizované testy treba neustále udržovať aktuálne. Pokiaľ sa v softvéri niečo zmení, okamžite musíme adekvátne zmeniť aj testy. Ak by sme tak nespravili, môže to mať hrozný dopad. Ak by sme neodhalili chyby, ktoré sa tam mohli pri zmene zaniesť, tie by sa vyskytli až v reálnej prevádzke aplikácie. Tam by bola oprava veľmi drahá.

Pre napísané výhody a nevýhody je dobré, keď sa väčšina testovacieho procesu automatizuje. Automatizácia nikdy úplne nenahradí manuálnu prácu, ako môžeme vidieť v množstve odvetví. To isté platí aj tu. Preto by sme automatizované testovanie brali skôr ako pomôcku v práci testerov, aby mali čas na riešenie zložitejších a komplexnejších problémov.

Kapitola 3

Testovacie nástroje

V tejto kapitole sa budem zaoberať testovacími nástrojmi a ich výhodami a nevýhodami pre konkrétne použitie v mojej bakalárskej práci. Keďže každý testovací nástroj je v niečom odlišný, je potrebné si zvoliť správny nástroj hneď na začiatku. Mojimi hlavnými kritériami pri vyberaní testovacieho nástroja boli skutočnosti, aby testovací nástroj podporoval MQTT protokol (prenos vzdialených meraní pomocou front správ). Bol voľne dostupný a aby jeho integrácia do Gitlab CI bola čo najjednoduchšia.

Testovacie nástroje sú inak povedané aplikácie, ktoré nám uľahčujú prácu pri testovaní softvéru. Prácu testera dokážu zrýchliť, zefektívniť alebo niekedy dokonca vôbec umožniť. Testovacie nástroje vieme rozdeliť do štyroch základných skupín.

Prvou skupinou sú nástroje, ktoré sa starajú o *bug tracking* (sledovanie chýb). Sú to nástroje, ktoré sa využívajú na evidenciu a správu nájdených chýb. Bugtracking je základ testovania a mal by byť vždy nejako riešený. Najprimitívnejší spôsob je využitie zdieľaných dokumentov. V praxi sa skôr ale využívajú špecializované nástroje, ako sú napríklad *Jira*[1], *Bugzilla*[2], *Mantis*[3] a mnoho ďalších.

Ďalšou skupinou sú nástroje, ktoré umožňujú vytváranie testovacích prípadov, ich spájanie s testovacími dátami a aj ich samostatné spúšťanie. A to buď priamo s použitím nástroja automatizovaného testovania alebo len v podobe otvorenia dokumentu, kde sú uložené kroky [33]. Medzi známe systémy patria TestLink[4], TestRail[5].

Rozsiahlou skupinou nástrojov sú nástroje pre automatizáciu testovania. V ďalších častiach tejto kapitoly budem bližšie popisovať niektoré z nich. Nástroje som vyberal na základe svojich osobných skúseností a na základe, čo musel daný testovací nástroj spĺňať, aby bol použiteľný aj v praktickej časti tejto práce.

Na záver nesmieme zabudnúť aj na podporné nástroje. Veľmi často sa na tieto nástroje zabúda, ale tester ich pri svojej práci často využíva. Sem patria napríklad databázové nástroje, editory. Tester pri svojej práci pracuje so súbormi v najrôznejších formátov. Preto potrebuje využívať rôzne editory, aby si mohol ich obsah zobraziť a prípadne upraviť.

Ako už bolo povedané, testovacích nástrojov je veľké množstvo a ja som sa rozhodol vybrať len niektoré. Inšpiroval som sa zoznamom desiatich najlepších automatizovaných nástrojov [7]. Z tohto zoznamu som vybral 7 testovacích nástrojov. Na konci kapitoly je

[1] www.atlassian.com/software/jira
[2] www.bugzilla.org
[3] www.mantisbt.org
[4] www.testlink.org
[5] www.gurock.com/testrail

opísaný testovací nástroj Tavern, ktorý som vybral ako najvhodnejší, a preto bude použitý aj v praktickej časti práce.

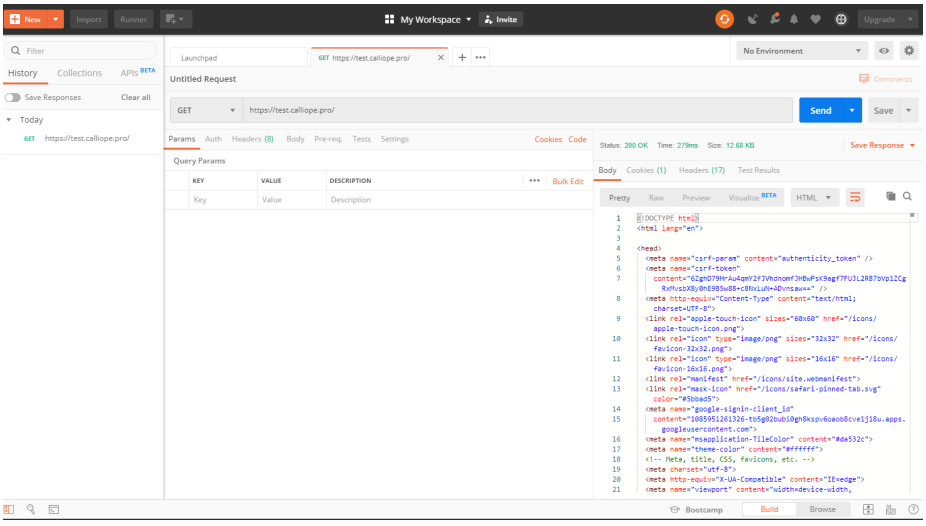
3.1 Postman

Je veľmi populárny automatizačný nástroj určený pre testovanie *Application programming interface* (rozhranie pre programovanie aplikácii) ďalej len API. Je možné si ho nainštalovať ako rozšírenie do prehliadača alebo na pracovnú plochu. Je podporovaný vo všetkých systémových prostrediach ako je MacOS, Windows alebo Linux. Je populárny nielen medzi testermi, ale aj medzi vývojármi, ktorí tento nástroj používajú k vývoju a testovaniu rozhrania API. Na to, aby sme mohli využívať Postman, sa stačí len zaregistrovať na ich webovej stránke a nemusíme nič zaplatiť, čo je veľká výhoda. Taktiež podporuje integráciu s GitHub alebo aj s Gitlab.

Jeho hlavnou výhodou je jednoduché posielanie žiadostí a prijímanie odpovedí pomocou *HTTP*[6] (hypertextový prenosový protokol). Dáva na výber veľké množstvo metód ako sú GET, POST, DELETE, UPDATE..., a spracovanie ich odpovedí. U jednotlivých metód môžeme testovať hlavičky v odpovedi, čas odpovedí, správu v odpovedi. Môžeme taktiež vykonávať pozitívne i negatívne testy s rôznymi dátami. Testovanie sa umožňuje predovšetkým skriptami, ktoré sú písané v JavaScripte.

Postman má nevýhodu, pokiaľ ide o monitorovanie testovacích prípadov. Veľa testerov sa sťažuje, že sú často limitovaní voľnou verziou a potrebovali si zaplatiť rozšírenú verziu. Či toto môže byť aj náš prípad, je ťažké povedať dopredu. Na to, či budeme limitovaní voľnou verziou, prídeme až po nejakom čase používania.

Ako sa môže zdať Postman je premyslený nástroj na testovanie. Jeho hlavnou nevýhodou pre moju bakalársku prácu je, že nepodporuje MQTT protokol, takže testovanie aplikácie by s týmto nástrojom nebolo možné.



Obr. 3.1: Užívateľské rozhranie Postman
[36]

[6] https://devdocs.io/http/

3.2 Insomnia

Insomnia[7] je intuitívny multiplatformový *REST API Client*. Je to výkonný klient so správou súborov cookie, s premennými pre prostredia, s generovaním kódu a autentifikáciou pre systémy Mac, Windows, Linux. Premenné prostredia si môžeme predstaviť ako nejaké hodnoty, ktoré sa vyžadujú pri každej žiadosti na danú API. Pre takéto prípady má Insomnia premenné prostredia, ktoré sa veľmi ľahko zadávajú a sú ľahko rozšíriteľné. Jeho hlavné využitie je pri testovaní rozhrania API, kde posielame žiadosti pomocou HTTP protokolu.

Insomnia je zadarmo a rozhranie je veľmi prívetivé a prehľadné. Určite odporúčam pre ľudí, ktorí práve len začínajú s testovaním a nemajú skúsenosti s programovaním a snažia sa len pochopiť ako HTTP protokol funguje. Taktiež komunita okolo Insomnie je veľmi silná, takže na internete nájdeme veľa návodov, ktoré nám pomôžu sa zorientovať. Pre tých pokročilejších sú na výber rozšírenia, ktoré sa dajú nainštalovať a rozširujú funkčnosť Insomnie. Dokumentácia, ktorú nájdeme na ich oficiálnych stránkach, je nadštandardná s porovnaním dokumentácií iných testovacích nástrojov, ktoré sú tiež zadarmo.

Ako bolo uvedené, základná verzia Insomnie je bez poplatkov. Ďalej však ponúkajú verziu Plus, ktorá je vhodná pre ľudí, ktorí chcú pracovať na jednej veci na viacerých počítačoch. Táto verzia stojí 50$ ročne. Pre spolupracovníkov, ktorí chcú kolaborovať na projektoch, je tu verzia Teams, ktorá im to uľahčí. Táto verzia však už stojí 80$ pre jedného užívateľa.

3.3 Selenium

Selenium[8] je považovaný za priemyselný štandard pre testovanie automatizácie používateľských rozhraní webových aplikácií. Takmer deväť z desiatich testerov používa alebo niekedy používali Selenium vo svojich projektoch [38]. Za popularitou stojí to, že je zadarmo a zároveň je aj open source (otvorený zdrojový kód). Na testovanie svojich webových rozhraní ho používajú aj veľké firmy ako Facebook alebo Google. Po jeho úvodnom, celkom zložitom nastavení, poskytuje efektívny spôsob generovania testovacích skriptov, overenie funkčnosti a opätovné použitie týchto skriptov.

Je určený pre developerov a testerov, ktorí majú skúsenosti s programovaním a skriptovaním. Užívatelia môžu písať testovacie skripty vo veľkom množstve v rôznych jazykoch ako napríklad Java, Groovy, Python, C#, PHP, RUBY, a Perl, ktoré bežia vo viacerých systémových prostrediach ako sú Windows, Mac, Linux a prehliadačoch Chrome, Firefox, IE.

Vďaka svojim mnohým výhodám nachádza Selenium široké využitie pri testovaní. Umožňuje rýchly vývoj testov a je aj veľmi populárny v prípade agilného vývoja. Tiež je obľúbený u zamestnancov IT, ktorí automatizujú opakujúce sa administratívne úlohy na webe.

Na druhej strane má Selenium pár nevýhod. Jednou z nich je, že nemá management na správu testov. Testovacie skripty sa ukladajú len ako jednoduché súbory bez atribútov. Tiež nepodporuje zdieľanie testov a výsledkov iba manuálnym spôsobom. Používateľ musí mať pokročilé znalosti programovania, aby dokázal vybudovať automatizačné rámce a knižnice potrebné na automatizáciu. Ďalšie informácie nájdeme na [37].

[7] www.insomnia.rest
[8] www.selenium.dev

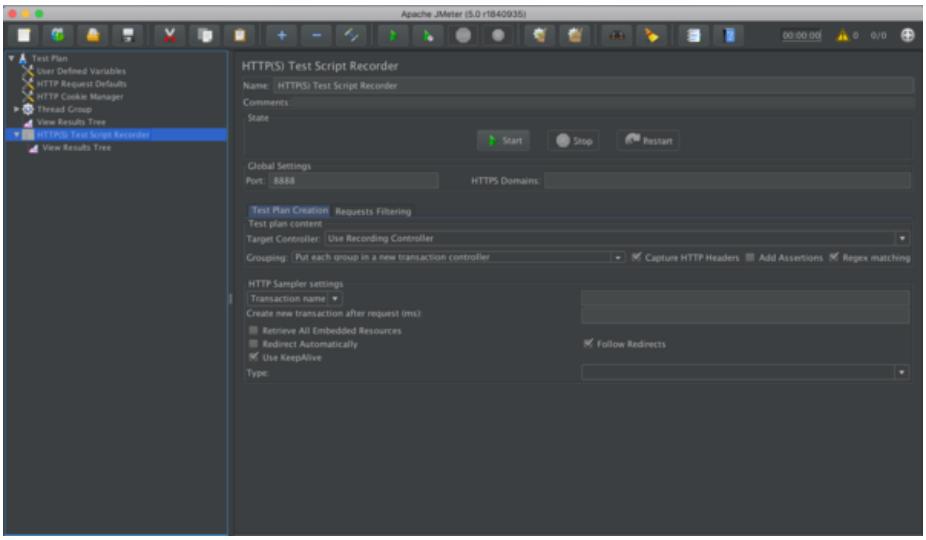
3.4 Apache JMeter

Apache JMeter[9] je softvér, ktorý je možno použiť na vykonávanie testov výkonu, testov záťaže a testov funkčnosti webových aplikácií. Jmeter tiež môže simulovať veľké zaťaženie servera vytvorením toku virtuálnych súbežných používateľov na webový server. Je to nástroj s otvoreným zdrojovým kódom, ktorý bol vyvinutý spoločnosťou Apache Software Foundation. Testovací nástroj Apache JMeter je napísaný v jazyku Java a je prevažne určený na testovanie funkčného správania a testovania výkonu. Pôvodne bol navrhnutý na testovanie webových aplikácií, ale od tej doby sa rozšíril a stále sa rozširuje o ďalšie testovacie funkcie.

Apache JMeter má schopnosť testovať veľa rôznych typov aplikácií, serverov a protokolov ako sú webové služby HTTP, HTTPS, SOAP, REST ďalej FTP, databáza cez JDBC, email SMTP, POP3, IMAP. Obsahuje tiež modul, ktorý umožňuje rýchle nahrávanie testovacieho plánu z prehliadačov alebo natívnych aplikácií. Režim príkazového riadku umožňuje načítanie testu z ľubovoľného operačného systému, ktorý je kompatibilný s Java. Sem patria Linux, Windows, MacOS a mnoho ďalších. Toto je len základ zo schopností, ktoré Apache JMeter má. Pre viac podrobných informácií o schopnostiach tohto testovacieho nástroja môžeme navštíviť oficiálnu webovú stránku [25].

Medzi hlavné výhody patrí otvorený zdrojový kód, ľahké použitie, nezávislosť platformy, robustné podávanie reportov. Veľa testovacích nástrojov má rovnakú slabinu a to je reportovanie. V tomto je JMeter veľmi silný. Výsledok dokáže vizualizovať do rôznych typov grafov a stromových štruktúr. Taktiež podporuje rôzne výstupné formáty ako obyčajný text, XML, HTML alebo aj JSON [30].

Na druhej strane, ako každý softvér, aj JMeter má svoje nevýhody. Ako už bolo uvedené skôr, je to dobrý nástroj na testovanie webových aplikácií, ale nie je vhodným nástrojom na testovanie desktopových aplikácií. Simulácia veľkého zaťaženia a vizualizácia reportov spotrebuje veľa operačnej pamäte. Keďže JMeter nie je prehliadač, nedokáže pustiť JavaScript vo webovej aplikácií. Má obmedzenú podporu pre spracovanie JavaScriptu alebo Ajaxu, čo môže ovplyvniť presnosť simulácie.



Obr. 3.2: Uživateľské rozhranie Apache JMeter [26]

[9] www.jmeter.apache.org

3.5 SoapUI

SoapUI[10] nie je nástroj na automatizáciu testovania a na testovanie webových alebo mobilných aplikácií, ale môže to byť nástroj voľby na testovanie API a webových služieb. Je to veľmi výkonný nástroj používaný na testovanie webových služieb (SOAP a REST). Obsahuje niekoľko funkcií ako napríklad: kontrola webových služieb, vývoj, testovanie funkčnosti, zaťaženie a testovanie zhody. Je postavený na platforme Java a používa Swing pre užívateľské rozhranie. Ide teda o multiplatformový nástroj, ktorý podporuje nástroj IntelliJ IDEA, Eclipse a NetBeands.

Existujú dve verzie a to verzia *open source* alebo *Pro*. Vydanie *Pro* má užívateľsky prívetivé rozhranie a niekoľko pokročilých funkcií navyše. Medzi pokročilé funkcie možno zaradiť generovanie testov pomocou drag and drop, point-and-click. Môže to znieť všetko pekne, ale verzia Pro začína na čiastke 640$ za rok, čo nie je najmenej, ak by sme chceli túto verziu pre viacerých testerov.

Medzi hlavné nevýhody SoapUI je problém pri porovnaní odpovedí vo forme *JSON (JavaScript Object Notation)*[11], keď žiadosť obsahuje niektoré špeciálne znaky. Je pravda, že implementácia je veľmi jednoduchá, ale testovanie záťaže a pracovného toku nie je vôbec jednoduchá úloha ani pre pokročilejších testerov. Taktiež integrácia do Gitlab CI, čo je pre nás veľmi dôležité, nie je vôbec jednoduchá. V mnohých prípadoch nie je ani vôbec možná.

Viac detailných informácií môžeme nájsť na ich oficiálnych stránkach [41].

3.6 Katalon Studio

Katalon studio[12] je bezplatný nástroj na automatizáciu testov na webové stránky, mobilné aplikácie a na webové služby. Obsahuje nahrávanie a prehrávanie a má manuálny režim. Vďaka manuálnemu režimu je vhodný aj pre ľudí, ktorí nemajú skúsenosti s programovaním a pomocou Katalon Studio sú schopní vytvárať prípady automatizačného testovania. Na druhej strane pre používateľov so skúsenosťami v programovaní obsahuje skriptový režim, pomocou ktorého môžu používatelia písať testovacie skripty v aplikácii Groovy [29]. Nahrávanie testov je inteligentné, čo znamená, že generuje pružné a udržovateľné lokátory. To umožňuje, aby skripty fungovali aj pri zmene testovacej aplikácie, čo veľa iných testovacích nástrojov neponúka.

Obsahuje bohatú sadu funkcií a je podporovaný na platforme Windows, MacOS a Linux. Taktiež je na výber veľa rozšírení, ktoré ešte obohacujú jeho funkcionalitu. S využitím Selenia a Appia vytvára prostredie pre testy, ktoré majú problémy s integráciou. Katalon Studio tiež umožňuje jednoduchú integráciu do GIT (distribuovaný systém riadenia revízií).

Aplikácia je zadarmo, ale naneštastie je vytváranie skriptov obmedzené iba na Javu a na Groovy. Ďalšou nevýhodou môže byť malá komunita, keďže nástroj bol vyvinutý v roku 2015. Zatiaľ ho nepoužíva dostatok ľudí na to, aby sa na internete vždy našlo riešenie nášho problému. Vtedy sa treba spoľahnúť na ich podporný tím, kde si na odpoveď chvíľu počkáme [28]. Viac podrobnosti o tomto nástroji môžeme nájsť v [27].

[10] www.soapui.org
[11] https://www.json.org/json-en.html
[12] www.katalon.com

3.7 Ranorex

Ranorex[13] je komerčný testovací nástroj, ktorý poskytuje testovanie desktopových, webových a mobilných aplikácií. Ranorex nepoužíva žiadny špecifický skriptovací jazyk. Je postavený na Microsoft's .NET platforme. Tým pádom podporuje štandardné programovacie jazyky ako sú C# a VB.NET, ktoré sa môžu využívať na úpravu nahrávok alebo na vytváranie vlastných testov. Ranorex je postavený na XPath (XML Path Language), ktorý umožňuje jednoduchšie a efektívnejšie vyhľadávanie komponentov vo webovej aplikácií. Je to určite dobrá voľba pre testerov, ktorí nemajú skoro žiadne skúsenosti s programovaním. Podobne ako Katalon Studio zjednodušuje automatizáciu testovania pomocou priateľského a intuitívneho používateľského rozhrania pomocou záznamu a prehrávania. Dá sa využiť aj generovanie skriptov [1].

S Ranorexom je možné spúšťať automatické testy webových, desktopových a aj mobilných aplikácií, pričom konkurenčné nástroje sa často zameriavajú len na jednu kategóriu z troch. Má silné centrum pomoci, kde je veľa návodov, ktoré veľmi uľahčujú prvé kroky. Taktiež má zákaznícku podporu, ktorá nám pomôže, ak nastane nejaký problém. Pomoci sa dočkáme celkom rýchlo.

Prvým a hlavným negatívom je cena. Základná verzia stojí 690€, ale verzia Ranorex Pro stojí až 2890€. Keďže Ranorex je postavený na .NET, nie sme schopní si ho spojazdniť na operačnom systéme MacOS. Ako bolo napísané v tejto kapitole skôr, centrum pomoci je silné, ale komunita okolo tohto produktu je veľmi malá. Takže často sa nám môže stať, že návod na vyriešenie nášho problému nenájdeme na internete. Budeme odkázaní na ich podporu, ktorá je dobrá, ale odpovie nám za dva dni. Poslednou a dosť veľkou nevýhodou sú nestabilné verzie. Preto je odporúčané, že ak budeme raz pracovať s Ranorexom, nesťahujme hneď novú verziu. Počkajme pár dní pokiaľ doladia chyby, ktoré sú tam. Viacej informácií môžeme nájsť na ich oficiálnych stránkach [11].

3.8 TestComplete

TestComplete[14] je automatizovaný nástroj na testovanie používateľského rozhrania vyvinutý spoločnosťou SmartBear Software. Obsahuje rozsiahly počet výkonných a komplexných funkcií pre testovanie webových, mobilných a desktopových aplikácií. Taktiež je nazývaný buď Script alebo Scriptless nástroj a to z dôvodu, že ak chceme skriptovať, môžeme tak urobiť v siedmich rôznych jazykoch. Je to JavaScript, Python, VBScript, Jscript, DelphiScript, C#, C++. Ak nemáme skúsenosti s programovaním, môžeme využiť funkciu nahrávania a prehrávania. Obsahuje veľmi užitočný nástroj na detekovanie prvkov dynamického užívateľského rozhrania. Tento nástroj je zvlášť užitočný v aplikáciach, ktoré majú dynamické a často sa meniace užívateľské rozhranie.

Ako produkt firmy SmartBerat, dokáže byť jednoducho integrovaný s ďalšími produktami od tejto firmy. V roku 2019 pridali integráciu do Jenkins (populárny open source CI server) [31]. Je možná aj integrácia s nástrojmi ako Azure DevOps a TeamCity. Pre nás je ale dôležitá integrácia s Gitlab CI, a tá nie je vôbec jednoduchá.

Cena za tento testovací nástroj sa pohybuje okolo 9$ za užívateľa na rok, kde je ale možnosť 30 dní na vyskúšanie zdarma. Bohužiaľ TestComplete má podporu len na platforme Windows, čo je dosť obmedzujúce. Viac informácií nájdeme na ich oficiálnych stránkach [45].

[13] www.ranorex.com
[14] www.smartbear.com

3.9 Tavern

Tavern[15] nie je testovací nástroj sám o sebe, ale je to plugin, nástroj v príkazovom riadku a Python knižnica pre automatické testovanie API. Jeho syntax je veľmi jednoduchá a je založená na formáte YAML (Ain't Markup Language) [16]. Tavern podporuje testovanie API založených na MQTT a testovanie RESTful API.

Veľa testovacích nástrojov spomenutých hore umožňuje testovanie len HTTP API. Tavern sa ale môže použiť na testovanie MQTT, alebo sa môžu kombinovať príkazy MQTT priamo s testami HTTP, ktoré sa využíva na testovanie zložitejších systémov. Je pravda, že Tavern nedokáže veľa vecí, ktoré vie Postman alebo Insomnia. Nenájdeme tu žiadne užívateľské rozhranie a ani monitorovanie API. Na druhej strane Tavern je zadarmo a open-source a je viac výkonný nástroj pre vývojárov na automatické testovanie.

Yaml syntax je jednoduchá na písanie, na čítanie a na pochopenie. Dokážeme s ním testovať čokoľvek. Od jednoduchých až po zložité testy je Tavern stále čitateľný a jeho vývojári sa snažia, aby bol aj intuitívny. Jeho integrácia do GITLAB CI nie je príliš náročná.

Prikladáme krátku ukážku, ako taký test v Tavern vyzerá.

```yaml
test_name: Get some fake data from the JSON placeholder API

stages:
  - name: Make sure we have the right ID
    request:
      url: https://jsonplaceholder.typicode.com/posts/1
      method: GET
    response:
      status_code: 200
      json:
        id: 1
      save:
        json:
          returned_id: id
```

Obr. 3.3: Príklad testu písaného pomocou Tavern [44]

[15] www.github.com/taverntesting/tavern
[16] https://yaml.org/spec/1.2/spec.html

Kapitola 4

IQRF GW Deamon

V tejto kapitole bude popísaný IQRF Gateway Deamon. Je to open-source softvér, pre ktorý budem v praktickej časti navrhovať a implementovať testovacie prípady a testovacie prostredie. Ešte pred popisom daného softvéru bude krátka podkapitola, ktorá sa bude zaoberať Internetom vecí. Je to z dôvodu, že Gateway Deamon patrí do ekosystému IQRF a ten je komplexným riešením v odbore Internet vecí. Po úvode do IoT bude nasledovať druhá podkapitola, ktorá bude rozdelená do viacerých častí. Na začiatku bude popísaný už spomenutý softvér. V ďalších častiach bude popísaný celý ekosystém IQRF a jeho najdôležitejšie časti.

4.1 IoT

Internet vecí (Internet of Things, IoT) je celkom nový trend, ktorý vďačí za svoj vznik technológii bezdrôtovej komunikácie. Ide o kontrolu a komunikáciu predmetov bežného využitia medzi sebou alebo s človekom [14]. Internet vecí bude súčasťou nášho bežného života a budeme sa s ním stretávať pravidelne. Prepojením fyzického sveta s virtuálnym svetom získame veľa nových možností. Všetky zariadenia, ktoré budú pripojené ku internetu, môžu na základe nazbieraných dát rozhodovať autonómne bez zásahu človeka a vykonávať činnosti.

História

Samotná myšlienka internetu vecí existovala oveľa rokov skôr ako vzniklo samostatné pomenovanie. Už v roku 1926 vynálezca Nikola Tesla v rozhovore pre Collier's Magazine povedal:*„Až bude bezdrôtová technológia perfektne adaptovaná, celý svet sa premení na jeden obrovský mozog a všetky časti reality sa spoja do jedného rytmického celku. Budeme schopní komunikovať okamžite medzi sebou bez ohľadu na vzdialenosť. A nielen to, budeme sa môcť navzájom počuť a vidieť tak dobre, ako keby stojíme priamo oproti sebe. I keď nás budú deliť tisíce kilometrov. Zariadenia, ktoré budeme používať na komunikáciu, budú v porovnaní so súčasnými telefónmi úžasne jednoduché. Človek ich bude môcť dokonca nosiť vo vrecku svojej vesty“* [14].

Samotný vznik termínu internet vecí sa datuje na rok 1999 a jeho autorom je Kevin Ashton. Podľa svojich vlastných slov tento výraz použil v jednej zo svojich prezentácií. *„Môžem sa mýliť, ale som si celkom istý, že fráza internet vecí vznikla ako titulok jednej mojej prezentácie, ktorú som vytvoril vo firme Procer & Gamble v roku 1999“* [4].

Medzi rokmi 2008 a 2009 sa stal dôležitý míľnik z pohľadu IoT. V tomto období podľa spoločnosti Cisco prekročil počet zariadení pripojených k internetu počet svetovej populácie [16]. Ako prvé boli do internetu pripojené počítače. Po nich nasledovali chytré telefóny a tablety. V roku 2020 sa odhaduje, že do internetu bude pripojených 50 miliard zariadení [35].

Definícia

Internet vecí má veľa rozdielnych definícií, a preto je niekedy obtiažne to presne definovať. Ja som vybral definíciu, ktorú vytvorila IERC (European Research Cluster on the Internet of Things). Ich definícia znie *„Internet vecí je globálna sieťová infraštruktúra s vlastnými konfiguračnými schopnosťami založenými na štandartných a interoperabilných komunikačných protokoloch, kde fyzické a virtuálne „veci“ majú identitu, fyzické atribúty a virtuálne osobnosti a používajú inteligentné rozhrania a sú bezproblémovo integrované do systému informačných sietí“* [15].

Požiadavky na IoT

Architektúra IoT musí umožniť zber dát, uloženie dát, analýzu dát a zdieľanie výsledkov. Toto sú hlavné požiadavky na IoT. Nesmieme však zabudnúť na bezpečnosť. Keďže má v rámci siete komunikovať veľa zariadení, je dôležitá aj prepojiteľnosť, aby sa dalo s rozhraniami jednoducho komunikovať. Rýchlosť je jednou z ďalších dôležitých premenných, ktoré vstupujú do fungovania ekosystému. Jednotlivé zariadenia potrebujú reagovať rýchlo na okolité podnety a situácie [5].

Smery vývoja IoT

V rámci IoT máme dva hlavné smery. Priemyselný internet vecí a spotrebiteľský internet vecí. Priemyselný internet vecí sa používa v priemyselných odvetviach ako sú priemyselná automatizácia, dopravný priemysel, energetický priemysel a zdravotníctvo. Hlavným zameraním tohto segmentu je lepšie využívanie zdrojov, zvýšenie pracovnej produktivity, zníženie nákladov. Predpokladá sa, že tento segment IoT bude prevládajúci [35].

Spotrebiteľský internet vecí sa zameriava na ľudí, spotrebiteľov. Patrí sem elektronika, ktorá sa využíva v domácnosti. Práčky, televízie, chladničky, osvetlenie. Hlavným zameraním je spraviť z obyčajných zariadení v domácnosti chytré, aby sa zvýšil užívateľský zážitok.

4.2 IQRF GW Daemon

IQRF GW Deamon, celým názvom IQRF Gateway Deamon, je open source softvér[1], ktorý umožňuje jednoducho vytvoriť bránu IQRF. Zabezpečuje spojenie systému Linux s internetom alebo s cloud. Tým pádom dokáže sprístupniť zariadenia komunikujúce na bezdrôtovej technológii IQRF do Internetu. S IQRF Gateway Webapp ponúkajú komplexné riešenie aj s užívateľským rozhraním. Deamon spravidla beží na jednočipovom počítači ako Raspberry PI, UP Board, Linux PlugPC, BeagleBone a ďalšie [19]. Ako je vidieť na priloženom obrázku, s koordinátorom 4.2 môže komunikovať cez SPI, UART alebo USB CDC. Z druhej strany môžeme komunikovať s Deamon cez IQRF IDE, ktoré bude opísané neskôr v tejto

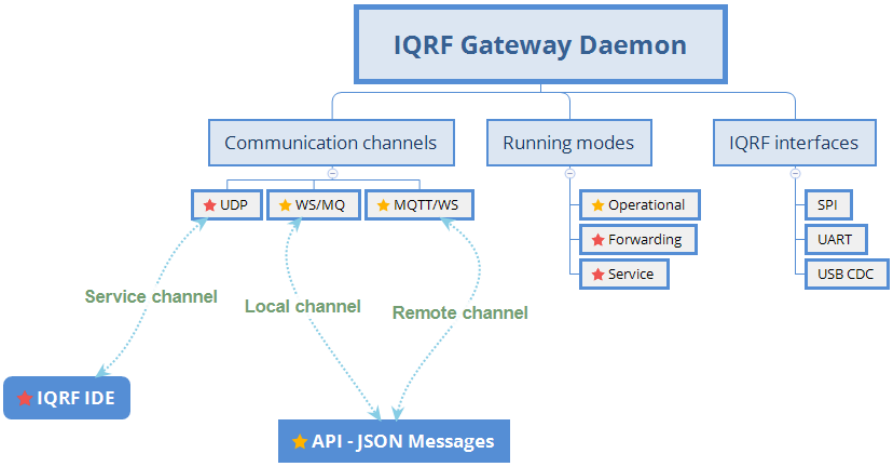
[1] https://gitlab.iqrf.org/open-source/iqrf-gateway-daemon

kapitole. Komunikácia prebieha cez servisný kanál pomocou UDP protokolu. Tento kanál sa používa na servis brány.

U certifikovaných zariadení zaistí všetky potrebné informácie. Medzi tieto informácie patrí aj spôsob komunikácie. Vďaka Daemon nemusíme riešiť ako treba poskladať správu, pomocou ktorej chceme ovládať zariadenia. Nemusíme ani poznať, ako je správa rozdelená a čo znamenajú jednotlivé časti správy. O všetko sa postarajú drivery, ktoré sa jednoducho zaintegrujú do riadiacej aplikácie.

Aby sa umožnila implementácia vlastných aplikácií nad IQRF, tak Deamon taktiež podporuje komunikáciu cez API-JSON správy. Tento typ správ je v dnešnej dobe populárny. Dajú sa posielať cez protokol MQTT (MQ Telemetry Transport) alebo cez WS (websocket). Keď takúto správu dostane Daemon, rozpársuje ju a vytvorí správu, ktorá je v súlade s protokolom DPA. Túto správu pošle do siete a očakáva odpoveď, ktorá je tiež v protokole DPA následne spracovaná v Daemon. Deamon nám už potom vracia správu vo formáte API-JSON.

Aplikácia IQRF Daemon má ešte nadstavbu s webovým rozhraním na konfiguráciu a riadenie siete IQMESH priamo z webového prehliadača. Táto aplikácia je navrhnutá ako open-source. Touto aplikáciu sa rozumie už vyššie spomenutá IQRF Gateway Webapp.



Obr. 4.1: IQRF GW Deamon
[20]

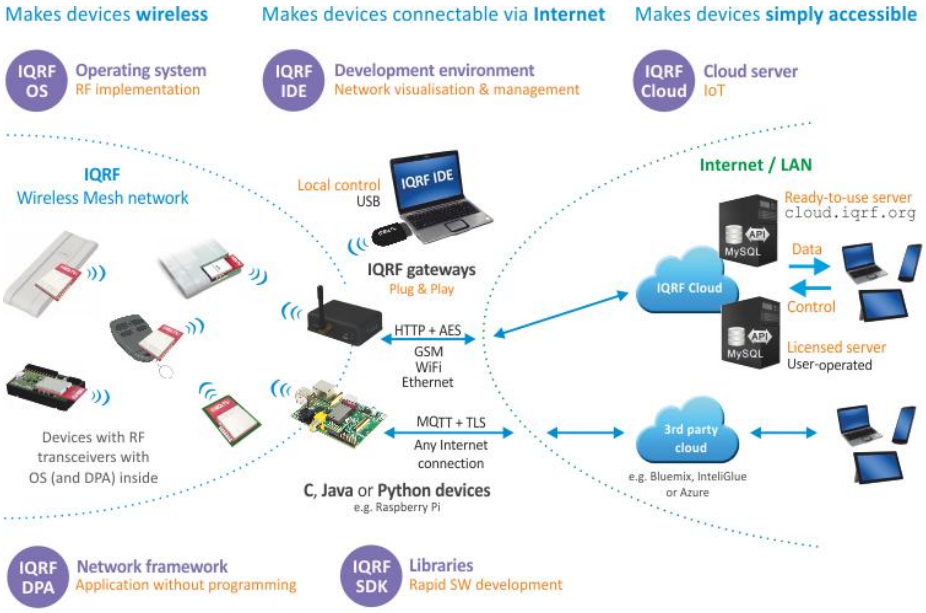
IQRF

IQRF je platforma českého výrobcu určená pre bezdrôtovú konektivitu s malým výkonom, nízkou rýchlosťou a nízkym objemom dát. Bola predstavená v roku 2004 českou firmou MICRORISC s.r.o a zahŕňa aj softvér aj hardvér. V roku 2017 sa z pôvodnej firmy oddelila časť firmy a vytvorili firmu IQRF Tech s.r.o, ktorá sa stará o všetky záležitosti spojené s IQRF technológiou.

Táto technológia je vhodná hlavne pre aplikácie v oblasti IoT. Prvky IQRF môžu byť použité s ľubovoľným elektronickým zariadením, kde je potrebný bezdrôtový prenos. Je to komplexný ekosystém od jednej značky vrátane hardvéru. Takéto elektronické zariadenie pripojíme do siete prostredníctvom štandartných rozhraní typu SPI (sériové periférne

rozhranie) alebo UART (univerzálny asynchrónny prijímač/vysielač). Takto pripojené zariadenie je potom možné bezdrôtovo ovládať alebo z neho získavať potrebné dáta.

Pre bezdrôtovú komunikáciu využíva bezlicenčné pásma 868 MHz alebo 433 MHz. Konfiguráciu je možné prispôsobiť požiadavkám niektorých štátov na vysielanie v pásme 916 MHz. V pásme 868 MHz je možné využívať až 62 kanálov, ktorých šírka je 100 kHz [18].



Obr. 4.2: IQRF ekosystém
[22]

Sieťová typológia

Každá sieť je riadená hlavným zariadením, ktoré sa volá koordinátor. Ten komunikuje s ostatnými zariadeniami, ktoré voláme nody. Komunikácia môže prebiehať v dvoch módoch a to peer-to-peer a v móde mesh. O móde mesh bude viac napísané v samostatnej podkapitole. Peer-to-peer je určený na komunikáciu s jedným alebo viacerými nody, ktoré musia byť v priamom dosahu. Je užitočný hlavne pri správe siete, napríklad pri lokalizácií konkrétneho zariadenia či údržbe [43].

IQMESH

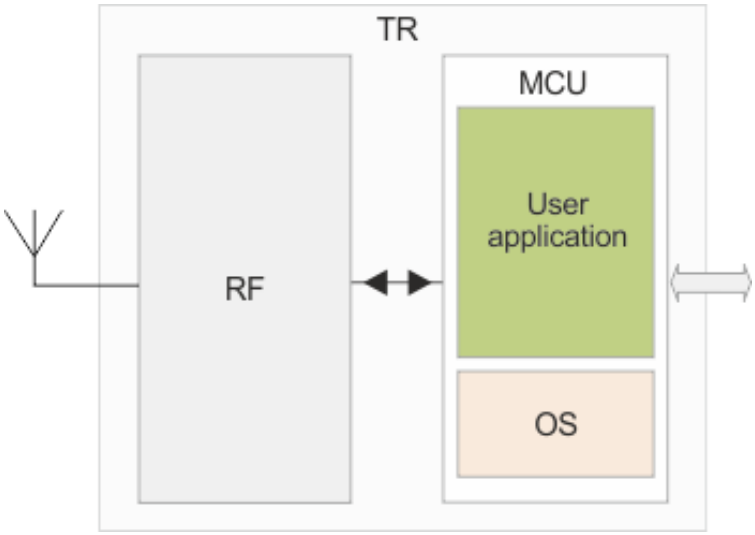
Smerovanie v IQRF sieti je pomocou smerovacieho protokolu IQMESH (IQRF Mesh protokol). Na rozdiel od iných smerovacích technológií je založený na smerovom synchronizovanom zaplavení. To znamená, že všetky zariadenia v sieti, ktorým sa to konfiguračne nezakáže, opakujú správu. Preto je vysoká pravdepodobnosť, že správa dorazí aj do najvzdialenejšej časti siete.

Takáto sieť môže mať až 240 zariadení, kde jedno zariadenie je koordinátor a zvyšné sú uzly. Každé zariadenie môže byť koordinátor alebo uzol, záleží len na nahranom hardvérovom profile a konfigurácii. Keď sa uzol pridáva do siete, priradíme mu virtuálne routovacie číslo. Uzly, ktoré sú bližšie ku koordinátorovi, majú toto číslo nižšie. Čím nižšie číslo, tým

toto zariadenie v menších intervaloch vysiela signál v sieti. Cieľom je, aby uzly, ktoré sú na konci siete a sú koncovými zariadeniami, zbytočne nevysielali signály.

IQRF OS

Je dvojvrstvová architektúra, ktorá obsahuje operačný systém a užívateľskú aplikáciu napísanú v jazyku C. Má veľa preddefinovaných funkcií, ktoré môžeme nájsť na tomto odkaze [17]. Tieto funkcie sú rozdelené do viacerých kategórií. Každá funkcia je typu *void* a všetky parametre sú typu *uns8*. *Uns8* je celočíselný typ bez znamienka, ktorý má 8 bitov. Rozsah platnosti je od 0 po 255. Táto dvojvrstvová architektúra sa dá použiť len pre nesieťové aplikácie. Pre sieťové aplikácie sa používa trojvrstvová architektúra DPA, ktorá bude viac opísaná neskôr v tejto kapitole.



Obr. 4.3: IQRF OS
[24]

IQRF IDE

Je to integrované vývojové a servisné prostredie vyvinuté firmou IQRF. Obsahuje nástroje pre návrh, programovanie, ladenie, konfiguráciu, aktualizáciu operačného systému, správu paketov a ďalšie. Kompletný zoznam nástrojov môžeme nájsť na stránke [23]. Sú podporované všetky vývojové kity a brány s USB rozhraním. Podporuje transceivery len rady TR-xB a vyššie. Pre nižšie rady treba použiť IQRF IDE 2.

Existujú dva typy tohto prostredia. Jedno je s grafickým rozhraním a druhý typ sa ovláda cez príkazový riadok. Využitie verzie s príkazovým riadkom je rýchlejšie a efektívnejšie, ale je odporúčané len pre skúsenejších programátorov.

DPA

DPA(Direct Peripheral Access) je jednoduchý bajtovo orientovaný protokol využívaný pre riadenie služieb a periférií. Koordinátor pošle nejakú požiadavku na uzol a pokiaľ nie je výpadok niekde na sieti, tak koordinátor obdrží potvrdzovaciu správu. DPA je trojvrstvová architektúra, ktorá sa skladá z operačného systému, DPA vrstvy a *Custom DPA Handler*.

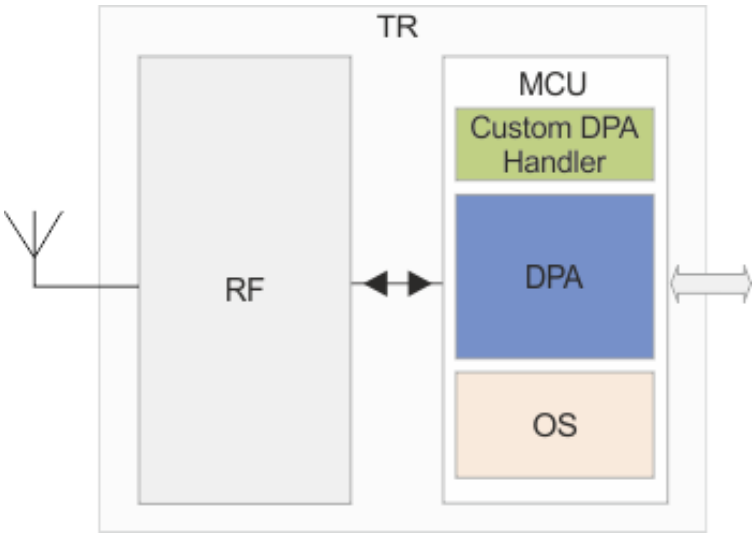
Základom je operačný systém, ktorý je v transceiveroch nahratý už od výroby a je možné ho jednoducho aktualizovať zo strany užívateľa. Nad operačným systémom je vrstva DPA, pomocou ktorej vieme zvoliť hardverový profil a tým urobíme z transceiveru buď koordinátor, alebo uzol. V tretej a poslednej vrstve môže byť Custom DPA Handler. Ten sa používa na rozšírenie DPA funkcionality. Je to komplikovaný program napísaný v jazyku C, pomocou ktorého ide prispôsobiť chovanie transceiveru. Ak chceme použiť Custom DPA Handler, tak ho musíme povoliť v konfigurácii transceiveru.

DPA umožňuje aplikovať bezdrôtovú sieť veľmi jednoduchým spôsobom. Sieť IQMESH je riadená protokolom DPA z riadiaceho systému pripojeného ku koordinátorovi prostredníctvom buď rozhrania SPI alebo UART.

Každá správa, ktorá je poslaná protokolom DPA, vždy obsahuje štyri povinné parametre NADR, PNUM, PCMD a HWPID. Správa má tiež voliteľnú časť, ktorá môže obsahovať ďalšie dáta. Táto časť sa volá PDATA.

DPA paket:

- NADR je sieťová adresa, ktorú zariadenie dostalo po pridaní do siete. Koordinátor má vždy adresu 0 a uzly majú adresy 1-239. Ďalšie adresy sú už rezervované. Jedna z nich je napríklad rezervovaná pre broadcast. Veľkosť tohto poľa je 2B.
- PNUM je číslo periférie, s ktorou chceme komunikovať. Rezervovaná veľkosť tohto poľa je 1B.
- PCMD obsahuje kód príkazu, ktorý má vykonať zvolená periféria. Jeho hodnoty závisia na zvolenej perifériií. Veľkosť je 1B.
- HWPID jednoznačne určuje funkčnosť zariadenia. Akcia sa vykoná len na zariadení, ktoré má rovnaké HWPID, ako je v DPA správe. Ide o identifikáciu hardverového profilu zariadenia. Veľkosť je 2B.
- PDATA je pole obsahujúce voliteľné dáta, ktoré môžu byť až vo veľkosti 56B [42].



Obr. 4.4: IQRF DPA
[21]

Kapitola 5

Návrh riešenia

Táto kapitola navrhuje, akým spôsobom bude výsledok tejto práce naimplementovaný. Najskôr sú v nej detailne špecifikované požiadavky na výsledný softvér. S týmito informáciami sa pozriem na testovacie nástroje, ktoré sú opísané v kapitole 3 a zhodnotím, ktorý z nich je najvhodnejší. Na konci sa táto kapitola venuje návrhu jednotlivých častí výsledného riešenia.

5.1 Požiadavky na výsledné riešenie práce

Požiadavka na nástroj vznikla vo firme IQRF, v skupine ľudí, ktorá sa venuje vývoju softvéru IQRF GW Daemon. Nástroj je určený pre vývojárov z firmy IQRF, ktorý by bol schopný zautomatizovať a uľahčiť testovací proces. Výsledný nástroj by mal ponúknuť možnosť testovať daný softvér bez potreby fyzicky pripojenej siete inteligentných zariadení. To znamená, že musí dokázať emulovať takúto sieť, lebo Daemon bez nej nefunguje. Toto testovanie by malo prebiehať automaticky v Gitlab CI po každom commite. Veľmi dôležité je aj, aby vývojári dokázali pridávať alebo editovať testy podľa potreby. Ďalej bola požiadavka na to, aby testovací nástroj bolo možné v budúcnosti rozšíriť o ďalšiu funkcionalitu.

Vhodným príkladom využitia nástroja sú situácie, ktoré podľa slov vývojárov z IQRF už nastali viackrát. Ide o situácie, keď im chyby na produkte hlásia vývojári, ktorí implementujú aplikácie komunikujúce s Daemon. Tieto chyby je ale potrebné odchytiť hneď po commite do Gitlab tak, aby ich oprava bola čo najrýchlejšia. Doteraz testovanie prebiehalo nepravidelne a ručne. Vývojár na testovanie potreboval mať pripojenú reálnu sieť inteligentných zariadení, kde posielal požiadavky z API a kontroloval, či naspäť dostal validné odpovede. Nebolo definované, ani ktorá funkcionalita sa má testovať. Kvôli týmto nedostatkom vznikla požiadavka na nástroj, ktorý by testovanie sprehľadnil a zautomatizoval.

Stručne zhrnuté požiadavky na nástroj:

- emulovať sieť inteligentných zariadení, ktorá komunikuje z Daemon pomocou protokolu DPA,
- zautomatizovať testovací proces v Gitlab CI,
- vybrať testovací nástroj pre posielanie správ cez MQTT protokol.

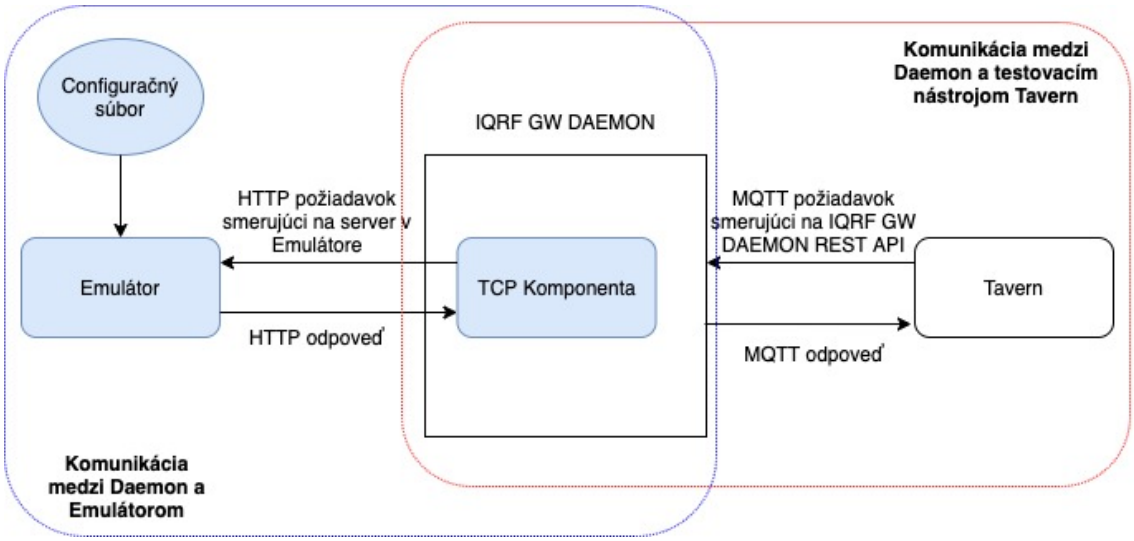
Toto sú zhrnuté všetky základne požiadavky na výsledok tejto práce. Má sa jednať o nástroj, ktorý dokáže testovať Daemon z obidvoch strán. To znamená, že z jednej strany bude

posielať YAML súbory a na druhej strane bude odchytávať DPA správy, na základe ktorých bude simulované chovanie v sieti. S ohľadom na stav siete a danú požiadavku následne vygenerujeme príslušnú odpoveď vo formáte DPA. Táto správa bude poslaná Daemonu. Ten ju spracuje a pošle nám naspäť YAML súbor, ktorý skontrolujeme, či je validný. Takto by malo byť pokryté, či je správanie testovaného softvéru korektné.

5.2 Návrh koncepcie riešenia

V tejto sekcii je popísaný koncept, ako by malo vyzerať návrhové riešenie práce. Obrázok 5.1, zobrazuje celkový návrh tak, aby bolo docielené komplexné testovanie, ako bolo popísané. To zahŕňa server, konfiguračný súbor, validačný súbor a TCP komponentu.

V budúcnosti by mala byť rozšírená časť, ktorá emuluje sieť. Je veľká pravdepodobnosť, že nejaké rozšírenia budú robiť vývojári z IQRF. Z toho dôvodu bol vybraný programovací jazyk Python. Uvažovalo sa sa aj nad jazykom Java, ale po konzultácii s ľuďmi z IQRF, som sa rozhodol pre Python.



Obr. 5.1: Návrh koncepcie riešenia

Obrázok 5.1 zobrazuje návrh výslednej práce. Obrázok je rozdelený na dve časti: komunikácia medzi Daemon a Taver a komunikácia medzi Daemon a emulátorom. Výsledok emulácie siete je publikovaný do Daemon, ktorý spracováva a odpovedá na požiadavky odoslané z Tavern. Modré časti sú súčasťou implementácie tejto práce.

V ďalších odsekoch sú postupne popísané jednotlivé časti návrhu.

- **Tavern**[1] - Tento testovací nástroj som sa rozhodol využiť z dôvodu, že podporuje MQTT. Taktiež je postavený na Python a emulátor, ktorý bude opísaný neskôr v tejto kapitole, bude programovaný v jazyku Python. Jeho integrácia do Gitlab CI testovanej aplikácie nebude komplikovaná a nebudú potrebné veľké úpravy. Posledná výhoda je, že vývojári pracujúci na aplikácií, ktorá má byť testovaná, už majú nejaké skúsenosti s Tavernom.

[1] www.github.com/taverntesting/tavern

- **Konfiguračný súbor** - Je súbor vo formáte JSON[2]. Tento súbor bude nahratý pred vytvorením komunikácie medzi emulátorom a Daemon. Obsahuje všetky údaje, ktoré sú potrebné na vytvorenie virtuálnej siete inteligentných zariadení. Jeho presný formát a možnosti budú opísané v kapitole 6.2.

- **TCP Komponenta** - Je komponenta, ktorá je naimplementovaná na strane IQRF GW Daemon. Je potrebná preto, aby bola možná komunikácia medzi mojim emulátorom a Daemon. Pri návrhu tejto komponenty som sa inšpiroval komponentami, ktoré sú už implementované. Hlavnou inšpiráciou bola komponenta UdpMessaging[3]. Na strane Daemon sa TCP komponenta chová ako TCP klient. TCP server je naimplementovaný v Emulátore. Myslím si, že toto môže byť tiež prínosom pre IQRF komunitu a vývojárov.

- **Emulátor** - Je hlavnou a najväčšou časťou mojej bakalárskej práce. Je písaný v programovacom jazyku Python. Hlavnou úlohou na emulátor bola požiadavka, aby nahradil fyzickú sieť, ktorá musela byť pripojené ku Daemon. Inak Daemon nefungoval a nedal sa testovať. Ako je možné vidieť na obrázku 6.1, vykonáva viac funkcií. Jeho prvou funkcionalitou je vykonávanie funkcie TCP servera tak, aby bolo možné komunikovať s Daemon. Spolu s pridanou TCP komponentou v Daemon tvoria spolu funkčnú komunikáciu, cez ktorú sa posielajú správy v protokole DPA. Jeho ďalšou funkciou je spracovať konfiguračný súbor a na základe neho vytvoriť príslušnú emulovanú sieť. Na tejto sieti sa vykoná emulácia na základe dát, ktoré boli prijaté na TCP serveri. Po simulácii sa poskladajú dáta, ktoré sú poslané naspäť klientovi. V mojom prípade bude klientom vždy Daemon.

[2] www.json.org/json-en.html
[3] https://gitlab.iqrf.org/open-source/iqrf-gateway-daemon/-/tree/master/src/UdpMessaging

Kapitola 6

Implementácia a testovanie

Táto kapitola popisuje vlastnú implementáciu ako celok a neskôr sa venuje jednotlivým častiam implementácie.

6.1 Popis celkovej implementácie

Celková implementácia, zobrazená na obrázku 6.1 sa skladá z časti *modul zostavenia siete, hlavný modul, TCP komponenta a Tavern s testami*. Všetky časti na strane emulátora (konfiguračný súbor, validačná schéma, overenie platnosti konfiguračného súboru, zostavenie siete, TCP server, spracovanie vstupných dát, emulácia v sieti, zostavenie výstupných dát) sú naimplementované v jednom Docker[1] kontajneri. Je to z dôvodu, že aj keď emulátor je rozdelený na viaceré časti, tak tieto časti fungujú ako celok a sú na seba úzko naviazané. TCP komponenta je implementovaná priamo v Daemon. Tavern beží v samostatnom kontajneri a začne automaticky posielať požiadavky hneď po úspešnom zostavení Daemon a emulátora.

Emulátor je vyvíjaný v jazyku Python 3.7 a skladá sa z konfiguračného súboru a validačnej schémy, ktoré sú vo formáte JSON, hlavného modulu a modulu zostavenia siete. TCP komponenta je implementovaná v jazyku C++ a je tam tiež *makelist*[2] vo formáte txt[3]. Konfiguračné súbory testov na strane Tavernu, ako aj samostatné testy, sú vo formáte YAML.

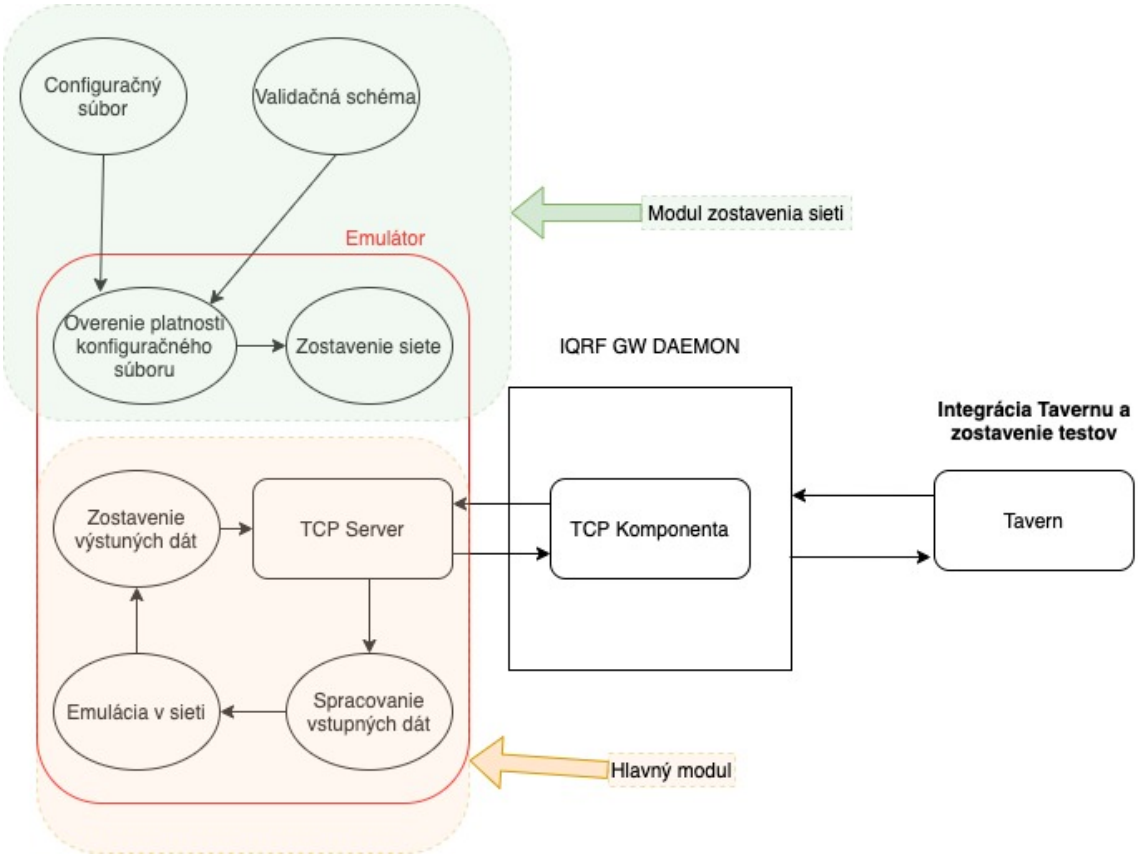
Celá implementácia funguje cez docker compose[4]. Docker compose je nástroj na definovanie a spúšťanie aplikácií s viacerými kontajnermi. Na konfiguráciu je použitý YAML súbor. Následne sa pomocou jediného príkazu vytvoria a spustia všetky služby. Moja implementácia je rozdelená do 4 kontajnerov. Jeden kontajner je pre MQTT broker, ktorý umožňuje komunikáciu s Daemon cez API. Daemon aj emulátor bežia ako služby v samostatných kontajneroch. Posledný a štvrtý kontajner je pre Tavern.

Taktiež bolo potrebné upraviť Dockerfile pre Daemon. Bol využitý Dockerfile, ktorý už využívali vývojári z IQRF a následne sa upravil na potreby mojej bakalárskej práce. V konfigurácií sú vypnuté komponenty Uart a Spi a na druhej strane je povolená TCP komponenta. Táto úprava sa uskutočnila z dôvodu, aby Daemon komunikoval so sieťou

[1] www.docker.com
[2] https://cmake.org/cmake/help/latest/guide/tutorial/index.html
[3] https://en.wikipedia.org/wiki/Text_file
[4] https://docs.docker.com/compose/

cez správny kanál. Posledným krokom je konfigurácia MQTT na strane Daemon. Tam sa nastavuje adresa MQTT brokera, meno, heslo a ďalšie potrebné parametre[5].

Emulátor a Tavern s testami sú popísané v nasledujúcich sekciách.



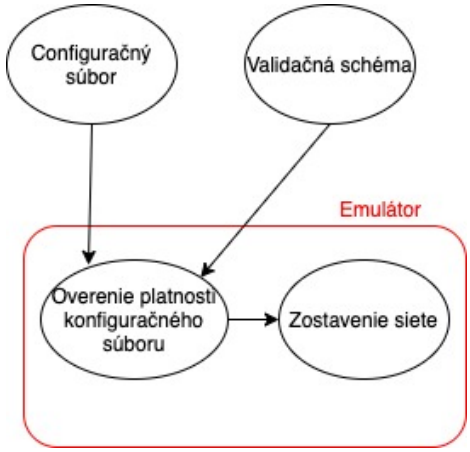
Obr. 6.1: Celková schéma implementácie

Na obrázku 6.1 je celková schéma implementácie, rozdelená na modul emulátor, TCP komponenta a Tavern s testami. Emulátor je rozdelený na hlavný modul a na modul zostavenia siete.

6.2 Implementácia modulu zostavenia siete

Jedná sa o modul v emulátore, ktorý sa stará o zostavenie siete. Tento modul je automatický spúšťaný hneď na začiatku životného cyklu emulátora ešte pred hlavným modulom. Implementáciu tohto modulu bolo možné rozčleniť na menšie celky, ktoré budú samostatne opísané v tejto kapitole.

[5] https://gitlab.iqrf.org/open-source/iqrf-gateway-daemon-utils/api-testing/-/blob/master/Daemon/configuration/iqrf__MqttMessaging.json



Obr. 6.2: Schéma modulu zostavenia siete

Konfiguračný súbor

Konfiguračný súbor je vo formáte JSON. Jeho výhodou je veľká popularita v komunite vývojárov. To bol dôvod pre jeho výber. Vývojári z IQRF si tam budú vedieť nakonfigurovať vlastnú sieť. Ďalšou výhodou je jeho prehľadnosť a ľahká čitateľnosť.

Konfiguračný súbor sa skladá z povinného objektu *coordinator* a z voliteľných objektov, ktorých počet môže byť premenlivý. V mojom príklade mám tento objekt jeden a je pomenovaný ako *node*. Objekt *coordinator* má povinné položky *nadr, dpaval, hwpid, mid, did, configuration, rfpqm, undocomented*. *Node* obsahuje všetky položky, ktoré aj *coordinator*, okrem *did*, ale navyše obsahuje položky *bonded* a *ibk*. Objekt *per* môže obsahovať aj *coordinator* v rovnakom tvare ako je vidno pri *node*. Príklad formátu konfiguračného súboru môžete vidieť neskôr v kapitole 6.1.

- **coordinator:**
  - **nadr** je sieťová adresa, ktorá pri coordinator musí byť vždy nastavená na nulu,
  - **dpaval** je hodnota, ktorá nadobúda vždy kladné hodnoty,
  - **hwpid** špecifikuje funkčnosť zariadenia, jeho naimplementované užívateľské periférie. Je potreba kontrolovať hodnotu len pri štandardných periférií ach. Viac o štandartných periférií ach nájdete 6.3,
  - **mid** je identifikátor, ktorý je pridelený každému modulu od výroby. Hodnota mid musí byť unikátna v celom konfiguračnom súbore,
  - **did** je discovery identifikátor siete,
  - **configuration** je parameter, ktorý sa využíva pri TR konfigurácií[6],
  - **rfpqm** na hodnote pri tomto parametre v emulovanej sieti moc nezáleží. Využíva sa pri TR konfigurácii, kde sa len vracia na Daemon,
  - **undocomented** má rovnaké využitie ako parameter rfpqm.
- **node:**

[6] https://static.iqrf.org/Tech_Guide_DPA-Framework-413_200227.pdf strana 61

  - **bonded** určuje, či je zariadenie v sieti nabondované. To znamená, či je dostupné. Hodnota nula znamená, že zariadenie nie je dostupné, hodnota jedna, že je. Iné hodnoty sú nevalidné,
  - **ibk** je *individual bonding key*. Hodnota je nastavená od výrobcu. Pre účely emulácie si môžete túto hodnotu vymyslieť, no musí byť unikátna. Veľkosť tohto čísla je 16 bajtov.

- **per** je skratka pre periférium. Je to objekt, v ktorom sú definované periférie pre jednotlivé zariadenia. Zariadenie môže mať zadefinovaných viac periférií. Periférie môžu byť pamäte, teplomery, ledky, senzory...
  - **pnum** je hodnota, ktorú má zadefinovanú každá periféria. Konkrétne hodnoty pre periférie sú opísané v tabuľke 6.2,
  - okrem *pnum* sú pre niektoré periférie definované aj iné hodnoty. Ako môžete vidieť v príklade, pre periférium 10 je pridaná hodnota *temp*. V tabuľke 6.2 je opísané, pre ktorú konkrétnu periférium sú tieto pridané hodnoty povinné.

periféria	pnum	variabilná hodnota
EEPROM	3	x
EEEPROM	4	x
RAM	5	x
LEDR	6	x
LEDG	7	x
SPI	8	x
IO	9	data
Thermometer	10	temp
UART	12	x
Sensor	94	sensors
Binary output	75	imputs
Light	113	x

Tabuľka 6.1: Tabuľka popisuje periférie, ktoré sa môžu definovať pre jednotlivé zariadenia

```
{
    "coordinator": {
        "nadr": "0",
        "dpaval": 0,
        "hwpid": 0,
        "mid": 0,
        "did": 50,
        "configuration" : [128, 42, 0, 7, 7, 6, 6, 0, 0, 0, 0, 0, 52, 2, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0],
        "rfpgm": 131,
        "undocomented" : 32,

  },
```

```
    "node": {
        "nadr": "1",
        "dpaval": 75,
        "hwpid": 0,
        "mid": 1,
        "bonded": 1,
        "ibk": [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],
        "configuration" : [128, 42, 0, 7, 7, 6, 6, 0, 0, 0, 0, 0, 52, 2, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0],
        "rfpgm": 131,
        "undocomented" : 32,

        "per": [
            {
                "pnum": "6"
            },
            {
                "pnum": "7"
            },
            {
                "pnum": "10",
                "temp": 30.6875
            }
        ]
    }
}
```

Výpis 6.1: Príklad konfiguračného súboru

Validačná schéma

Je súbor, ktorý sa využíva na validáciu konfiguračného súboru. Sú v ňom zahrnuté podmienky ako napríklad, ktorá hodnota je povinná alebo v akom rozsahu môže nadobúdať hodnoty. Je tiež vo formáte JSON. Na jeho vytvorenie som použil online generátor [7]. Po vygenerovaní validačnej schémy bolo nutné spraviť manuálne úpravy a pridanie podmienok, ktoré tieto generátory nevedia vyčítať z JSON súboru. Tento súbor sa načíta do emulátora hneď, ako prebehne načítanie konfiguračného súboru. Samozrejme za podmienky, že načítanie konfiguračného súboru prebehlo bez problémov.

Overenie platnosti konfiguračného súboru

Je proces, pri ktorom sa kontroluje, či konfiguračný súbor spĺňa všetky podmienky, ktoré sú špecifikované vo validačnej schéme. Na tento proces som využil knižnicu *jsonschema 3.2.0*[8]. Z knižnice som presne využil funkciu *Draft7Validator*. Ak nastali nejaké chyby pri validácií, tak je upravený výpis týchto chýb. Je to z dôvodu, pretože základný výpis knižnice *jsonschema* nebol prehľadný. Ak validácia prebehne bez problémov, prejde sa na ďalší proces, čo je zostavenie siete.

[7] https://www.jsonschemavalidator.net/
[8] https://pypi.org/project/jsonschema/

Zostavenie siete

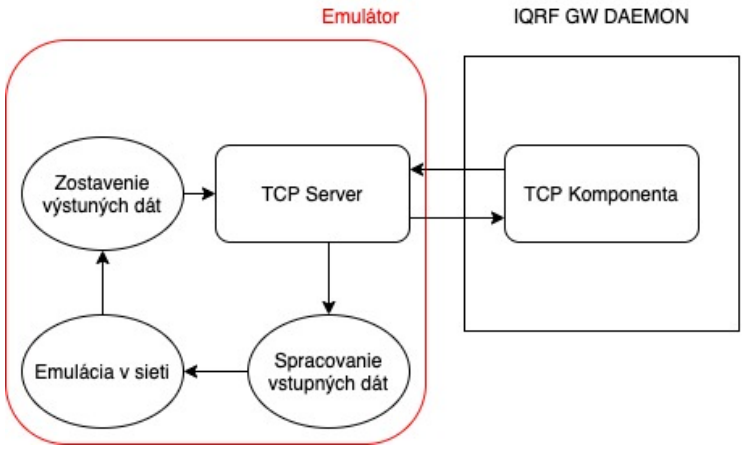
Je poslednou a finálnou časťou, ktorú vykonaná modul pre zostavenie siete. Celá sieť je uložená ako dátový typ *slovník*[9]. Ako index sa využíva *nadr* hodnota z konfiguračného súboru. Každá položka v slovníku je typu *device*. Každý *device* obsahuje hodnoty *dpval, hwpid, ibk, mid*. Okrem týchto hodnôt obsahuje slovník periférií. Tieto periférie sú indexované podľa *pnum*. V slovníku periférií je na prvom mieste vždy buď periféria *coordinátor* alebo *node*. Na druhom mieste sa vždy nachádza periféria operačný systém. *Device* taktiež obsahuje slovník štandartných periférií. Medzi štandardné periférie radíme *sensor, binary output, light* a *dali*.

V emulátore je každá periféria definovaná ako samostatná trieda. Je to z dôvodu, že každá periféria má naimplementovanú inú funkcionalitu podľa príslušnej dokumentácie[10] a k tomu svoje špecifické premenné. Všetky periférie ale obsahujú premenné *pert, perte, par1, par2*, kde *pert* nesie hodnotu typu periférie. *Perte* je rozšírená charakteristika periférie. *Par1* a *Par2* sú nepovinné špecifické informácie.

Modul zostavenia siete je ukončený, keď tento proces prejde úspešne. Tým pádom je emulátor pripravený prejsť do ďalšej fázy, v ktorej sa už naviaže komunikácia s Daemon.

6.3 Implementácia hlavného modulu a TCP komponenty

V tejto kapitole bude opísaná funkcionalita a implementácia TCP komponenty a hlavného modulu emulátora. Konkrétne bude popisovať, ako funguje komunikácia medzi emulátorom a Daemon. Taktiež ako emulátor simuluje virtuálnu sieť, aby dokázal vytvoriť odpoveď, ktorá by prišla aj z reálnej siete.



Obr. 6.3: Schéma hlavného modulu

TCP komponenta

Je komponenta, ktorá je naimplementovaná v Jazyku C++. Táto komponenta je zasadená priamo v Daemon. Aby sa lepšie pochopilo, čo táto komponenta robí, mohol by jej byť priradený názov TCP klient. Najskôr sa na začiatku vytvorí socket. Potom sa pokúsi pripojiť

[9] https://docs.python.org/3/tutorial/datastructures.html
[10] https://static.iqrf.org/Tech_Guide_DPA-Framework-413_200227.pdf

Daemon na TCP server v emulátore. Ak sa pripojenie podarí, začnú si medzi sebou posielať dáta. Táto komponenta sa dá v nastaveniach Daemon aktivovať ako aj deaktivovať. Toto ale pre moju bakalársku prácu nie je podstatné. Táto funkcionalita bola pridaná, pretože bola vyžadovaná zo strany vývojárov IQRF. Pretože je to súčasť inej práce [12], tak viac informácií o tejto komponente je možné nájsť tam.

TCP server

TCP server je už implementovaný ako súčasť emulátora. Keďže je súčasťou emulátora, je implementovaný v jazyku Python 3.7. Základná funkcionalita, ako bola už spomenutá validácia konfiguračného súboru a aj TCP server, sú implementované v jednom súbore, ktorého meno je *server.py*. Server beží na IP adrese 0.0.0.0 a na porte 10000. Port bol vybraný po konzultácii s vývojármi z IQRF, aby nenastali nejaké konflikty.

Pri implementácii som využil knižnicu *socket* [11]. Na začiatku sa vytvorí TCP socket. Potom sa využije funkcia *setsockopt*, aby sa dala použiť jedna adresa viackrát. To je z dôvodu, aby nenastal problém pri znovupoužití rovnakej adresy. Funkcia bind vykoná naviazanie socket na definovanú adresu a port. Po týchto krokoch je už server pripravený začať nadväzovať spojenie. Hneď po pripojení Daemon na emulátor je potrebné poslať asynchrónne paket do Daemon. Tento paket nesie hodnoty konfigurácie a informácie o koordinátore, keďže v reálnej sieti všetka komunikácia ide cez koordinátor na ostatné zariadenia. Nastavuje sa tam verzia operačného systému alebo verzia DPA protokolu. Bez tohto Daemon nie je schopný komunikovať so sieťou. Ak by sa aj tieto dáta poslali, ale mali by zlé hodnoty, tak Daemon sa bude správať neočakávane a testovanie nebude možné. Daemon obsahuje rôzne konfigurácie a pri ich nastavovaní treba byť opatrný. Tieto konfigurácie sú z dôvodu, že pripojená reálna sieť môže používať ešte staršiu verziu DPA protokolu.

Všetky dáta, ktoré sa posielajú medzi Daemon a emulátorom, sú v bajtoch. Ako spracovať takéto dáta a ako ich zostaviť pred odosielaním bude popísané ďalej v tejto kapitole.

Spracovanie vstupných dát

Ako už bolo povedané, dáta prichádzajú v bajtoch, čo nie je veľmi prívetivý typ na prácu s nimi. Preto je naimplementovaná funkcia, ktorá tieto dáta rozpársuje. Na začiatku sa skontroluje, či dĺžka týchto dát nie je menšia ako 6. Ak je tak, prišli nevalidné dáta. Táto hodnota je zistená z dokumentácie o DPA frameworku. Po tejto validácii sú dáta rozdelené na pole bajtov, lebo na vstup prišli ako reťazec bajtov. Pomocou knižnice *struct*[12] sú bajtové hodnoty prevedené na príslušné číselné hodnoty. Z takto rozpársovaných hodnôt sa dostáva *nadr, pnum, pcmd, hwpid, pdata*.

Emulácia v sieti

Po spracovaní dát je na rade emulácia v sieti. Zo vstupných dát máme rozpársované všetky potrebné údaje ako *nadr, pnum*. Tieto dva údaje sa využívajú na nájdenie konkrétneho zariadenia v sieti a nájdenie presnej periférie na tomto zariadení. Ak nájdenie prebehlo úspešne, volaním sa prechádza do zdrojového kódu periféria. Každá periféria má vlastný implementačný súbor, ktorý nesie meno podľa názvu implementovanej periférie.

Každá periféria má funkciu *HandleRequest*, ktorá má na starosti vybrať konkrétnu funkciu, ktorú je potrebné simulovať. Toto je možné vďaka rozpársovanej hodnote *pcmd*. Ak

[11] https://docs.python.org/3.7/library/socket.html
[12] https://docs.python.org/3/library/struct.html

je táto hodnota nevalidná, tak funkcia periféria nič nevykonáva a vracia chybovú hodnotu 2 a prázdne pole, ktoré reprezentuje dáta. Na druhej strane, ak je hodnota validná, zavolá sa príslušná funkcia v periférii. Každá funkcia má počet parametrov podľa potreby. Ak sú dáta žiadané, tak sa predajú ako parametre funkcie.

Pomenovanie samotnej funkcie je inšpirované z dokumentácie DPA. Ak funkcia prebehla bez chyby, vracia sa chybový kód 0 a dáta, ktoré očakáva Daemon. Nie vždy ale Daemon nejaké dáta očakáva. Po ukončení vykonania funkcionality na periférii je už na rade posledný krok a to je zostavenie výstupných dát.

V mojej bakalárskej práci sú naimplementované periférie *eeprom, ledg, light, ram, spi, node*. Periférie *coordinator* a *os* sú periférie s rozsiahlou funkcionalitou, a preto ich čiastočná implementácia je obsahom práce [12].

Zostavenie výstupných dát

Je presný opak toho, čo je naimplementované v prípade spracovania vstupných dát. Na túto činnosť sa taktiež využíva knižnica *struct* a jej príslušné funkcie. Aby tieto dáta boli podľa očakávania Daemon, je spravené, že ku hodnote *pcmd* sa pripočíta číslo 128. Dáta po prevedení na bajty sú spojené dokopy a poslané naspäť do Daemon pomocou TCP komunikácie. Ich obsahom je *nadr, pnum, pcmd, hwpid, errn, dpval, data*. Jediná nová hodnota, ktorá ešte doteraz nebola vysvetlená, je *errn*. Táto hodnota znamená chybový kód. Ak je jej hodnota 0, tak všetko prešlo bez komplikácií. Chybové hodnoty v emulovanej sieti sú totožné s hodnotami, ktoré sú špecifikované v DPA framework dokumentácii. Bez korekcie hodnôt s dokumentáciu by nebolo možné dôveryhodne testovať Daemon a jeho funkcionalitu.

6.4 Integrácia Tavernu a zostavenie testov

Táto podkapitola bude opisovať, ako je naimplmenetovaná integrácia Tavernu do navrhnutého ekosystému. Ďalej bude popísané, ako prebieha spúšťanie testov a aká funkcionalita Daemona je testovaná.

Integrácia Tavernu

Tavern beží ako samostatná aplikácia v Docker Compose. Na jeho integráciu bolo potrebné napísať Dockerfile. S Daemon je schopný komunikovať vďaka *MQTT broker*[13]. V mojej bakalárskej práce som použil konkrétne *mosquitto*[14]. *Mosquitto* umožňuje komunikáciu medzi Tavernon a Daemon cez MQTT správy. Aby komunikácia fungovala správne, tak je potrebné do *.tavern.yaml* súboroch, kde sa spúšťajú jednotlivé testy, pridať hlavičku. Názov hlavičky je *paho-mqtt*. Vďaka tomuto názvu Tavern vie, že komunikácia bude prebiehať cez MQTT správy. Ďalšími parametrami sú port, host, typ pripojenia a identifikátor klienta.

Spúšťanie testov

Spúšťanie testov má na starosti súbor *run.py*. Na túto funkcionalitu sa využíva knižnica tavern.core a funkcia run. V tejto funkcii sa ako prvý parameter zadáva súbor *.taver.yaml*. Ďalšie použité parametre sú *-x*. Tento parameter udáva, že keď jeden test je neúspešný,

[13] http://mqtt.org/
[14] https://mosquitto.org/

tak je ukončený celý testovací proces. Parameter *–tb=short* upravuje chybový výstup. Je použitý z dôvodu, že neupravený výstup Tavernu je neprehľadný a zdĺhavý.

Každá periféria má vlastný *.taver.yaml* súbor. Tento súbor obsahuje hlavičku, ktorá už bola opísaná vyššie. Telom tohto súboru je *stages*. Jeden súbor môže obsahovať aj viac týchto *stages*. Jednotlivé testy sú pomenované a obsahujú *mqtt_publish* a *mqtt_response*. V prvej časti je cez príkaz *json* importovaný už konkrétny súbor pre žiadosť. V druhej časti je cez rovnaký príkaz importovaný súbor, ktorý už obsahuje očakávanú odpoveď.

Zostavenie samotných testov

Každá periféria má vlastný priečinok, kde má zostavené všetky testy týkajúce sa jednej periférie. Názov jednotlivých súborov obsahuje vždy reťazec *request* alebo *response*. Je to definované podľa toho, či je to súbor, ktorý definuje žiadosť na Daemon alebo súbor, ktorý definuje očakávanú odpoveď.

Aj žiadosť aj súbor pre odpoveď obsahujú na začiatku položku mType. Táto položka obsahuje názov funkcie, ktorá sa má vykonať. Tento reťazec je jasný identifikátor funkcie, ktorá sa má vykonať na sieti. Z jej hodnoty Daemon zostaví hodnotu *pcmd*. Druhou a poslednou položkou je *data*. Je to objekt, ktorý obsahuje už všetky ostatné položky, ktoré sú premenlivé. V prílohe B môžete nájsť postup, ako vytvárať nové testovacie prípady.

Výsledky testovania

Počas implementácie emulátora a zostavovania testov prebiehalo súbežne aj samotné testovanie Daemon. Toto testovanie obsahuje celkovo 79 testov. Zoznam všetkých naimplementovaných testov je spísaný v prehľadnej tabuľke 6.2.

Ukázalo sa, že testovanie je účinné. Počas tvorenia bakalárskej práce sa odhalili dve podstatné chyby na strane Daemon, vďaka ktorým nefungovali dve funkcie. Prvá chyba bola objavená pri funkcii *Set_MID* kde funkcionalita nebola dostupná kvôli nezhode v menách zdrojových súboroch. Druhá chyba nastala pri testovaní funkcii *WriteCfg* na periférií *OS*. Táto funkcionalita bola taktiež nedostupná. Odhalilo sa, že Daemon načítaval starý *driver*. Tieto chyby boli nahlásené a následne opravené. Taktiež sa našli nedostatky v dokumentácii, na ktoré boli upozornení vývojári z IQRF.

názov periférie	**názov funkcie**	**počet testov**
EEPROM	Read	3
	Write	3
	GetPerInfo	2
Explore	Enumerate	2
	GetMorePerInfo	2
Ledg	Set	2
	Pulse	2
	Flash	2
	GetPerInfo	2
Light	SetPower	3
	IncrementPower	2
	DecrementPower	2
	EnumerateLights	1

Ram	Read	3
	Write	3
	GetPerInfo	2
Spi	WriteRead	4
	GetPerInfo	2
Node	RemoveBond	4
	BackupN	2
	RestoreN	2
	GetPerInfo	2
Coordinator	Backup	2
	Restore	2
	ClearBonds	2
	BondNode	2
	SmartConnect	2
	AddrInfo	3
	SetDPA	2
	SetMID	2
OS	Sleep	2
	ReadCfg	3
	GetPerInfo	2
	SetSecurity	2
	WriteCfg	1
názov periférie	**názov funkcie**	**počet testov**

Tabuľka 6.2: Tabuľka obsahuje zoznam otestovaných funkcií a počet ich testov

Kapitola 7

Záver

Cieľom tejto práce bolo vytvoriť testovací softvér, ktorý umožní testovať open source softvér IQRF GW Daemon, ktorý je vyvíjaný ľuďmi z IQRF bez potreby pripojenia fyzickej siete inteligentných zariadení. Ďalšími požiadavkami bolo vytvorenie testov overujúcich funkčné správanie Daemon a taktiež integrácia vytvoreného testovacieho systému do Gitlab CI.

V teoretickej časti mojej práce som sa najskôr oboznámil s teóriou v oblasti testovania. Ďalej som vykonal prieskum dostupných testovacích nástrojov. Zozbieral som informácie o testovacích nástrojoch a podľa požiadaviek vybral najvhodnejší. Ďalej som sa oboznámil so samotným softvérom Daemon. Konkrétne som sa zameral na jeho pozíciu a funkčnosť v typológii IQRF. Nevyhnutnou súčasťou bolo dôkladné naštudovanie protokolov, pomocou ktorých daný softvér komunikuje. V závere teoretickej časti som analyzoval funkčnosť jednotlivých zariadení v sieti a komunikáciu medzi nimi.

Po nadobudnutí teoretických znalostí som v návrhu popísal vybraný testovací softvér, ktorý bol využitý v mojej bakalárskej práce. Ďalej som zhrnul požiadavky na testovací systém, ktoré vyplývali z charakteristiky softvéru Daemon. V poslednej časti návrhu systému som s ohľadom na požiadavky a na nadobudnuté teoretické znalosti vytvoril návrh systému.

Podľa vytvoreného návrhu som realizoval implementáciu testovacieho systému a testov. Vzhľadom na to, že jednou z požiadaviek na systém bola rozšíriteľnosť, je implementácia prevedená spôsobom, aby sa pri ďalšom rozširovaní nezasahovalo do funkcií, ktoré sú základným kameňom celej práce. Spolu s testovacím nástrojom a emulátorom siete bola vytvorená aj kolekcia testov. Počas implementácie mojej práce bol Daemon priebežne testovaný. Vďaka tomu testovací systém už počas fázy vývoja dokázal odhaliť chyby na strane Daemon.

V priebehu práce som sa naučil lepšie komunikovať s ľuďmi a organizovať si čas. Práca mi pomohla prehĺbiť znalosti v oblasti internetu vecí, hlavne pri technológii IQRF.

Možným pokračovaním práce je rozšírenie funkcionality emulácie siete. Pri tomto rozširovaní bude potrebné vytvárať aj nové testy. Prípadne bude nutná ďalšia obmena existujúcich testov, napríklad v prípade zmeny vo funkcionalite testovaného softvéru.

Literatúra

[1] Altexsoft. *The Good and the Bad of Ranorex GUI Test Automation Tool* [online]. altexsoft.com, 2018 [cit. 2020-31-03]. Dostupné z: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-ranorex-gui-test-automation-tool/.

[2] *What is Fundamental of Test Process in Software Testing?* [online]. softwaretestingbooks.com [cit. 2020-25-01]. Dostupné z: http://softwaretestingbooks.com/what-is-fundamental-of-test-process-in-software-testing.

[3] Andreas Spillner, H. S. *Software Testing Foundations*. 4. vyd. Rocky Nook, Inc, 2014 [cit. 2020-05-05]. ISBN 978-1-937538-42-2.

[4] Ashton, K. *That 'Internet of Things' Thing* [online]. rfidjournal.com, 1999 [cit. 2020-25-01]. Dostupné z: https://www.rfidjournal.com/that-internet-of-things-thing.

[5] Beran, P. *Představení a analýza IOTA protokolu - 2. část - Internet věcí* [online]. medium.com, 2019 [cit. 2020-25-01]. Dostupné z: https://medium.com/@pavelberan/p%C5%99edstaven%C3%AD-a-anal%C3%BDza-iota-protokolu-2-%C4%8D%C3%A1st-internet-v%C4%9Bc%C3%AD-d55dd00abb63.

[6] Boehm, B. W. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*. [online]. 1984, roč. 1, č. 1, [cit. 2020-16-01], s. 75–88.

[7] Brian. *Best Automation Testing Tools for 2020 (Top 10 reviews)* [online]. medium.com, 2017 [cit. 2020-20-03]. Dostupné z: https://medium.com/@briananderson2209/best-automation-testing-tools-for-2018-top-10-reviews-8a4a19f664d2.

[8] Cem Kaner, H. Q. N. *Wiley Computer Publishing*. 2. vyd. Robert Ipsen, 1999 [cit. 2020-08-05]. ISBN 0-471-35846-0.

[9] Dankanin, R. *Testovanie softvérových systémov, súčasné trendy a výzvy* [online]. kpi.fei.tuke.sk, 2017 [cit. 2020-16-01]. Dostupné z: https://kpi.fei.tuke.sk/sites/www2.kpi.fei.tuke.sk/files/aktuality/testovanie_softverovych_systemov_sucasne_trendy_a_vyzvy-dankanin-2017.pdf.

[10] *Functional Testing* [online]. softwaretestingfundamentals.com [cit. 2020-23-01]. Dostupné z: http://softwaretestingfundamentals.com/functional-testing/.

[11] GmbH, R. *Ranorex* [online]. 2020 [cit. 2020-31-03]. Dostupné z: https://www.ranorex.com/.

[12] Hanák, K. *Automatické testování software*. 2020. Bakalářské práce. Vysoké učení technické v Brně, Fakulta informačních technologií. [cit. 2020-25-05].

[13] Hlava, T. *Fáze a úrovně provádění testů* [online]. testovanisoftwaru.cz, 2011 [cit. 2020-16-01]. Dostupné z: http://testovanisoftwaru.cz/category/druhy-typy-a-kategorie-testu/.

[14] *Internet of Things (IoT) History* [online]. iot-portal.cz [cit. 2020-25-01]. Dostupné z: https://www.iot-portal.cz/co-je-iot/.

[15] *Internet of Things* [online]. internet-of-things-research.eu, 2016 [cit. 2020-25-01]. Dostupné z: http://www.internet-of-things-research.eu/about_iot.htm.

[16] *Internet vecí* [online]. Pavel Pohanka, 2020 [cit. 2020-25-01]. Dostupné z: http://www.pavelpohanka.cz/internet-of-things/.

[17] *IQRF OS Reference guide* [online]. doc.iqrf.org, 2019 [cit. 2020-04-04]. Dostupné z: https://doc.iqrf.org/IQRF-OS-Reference-guide/index.html?page=table-of-os-functions.html.

[18] *What is IQRF?* [online]. iqrf.org, 2020 [cit. 2020-04-04]. Dostupné z: https://www.iqrf.org/what-is-iqrf.

[19] *GW Daemon* [online]. IQRF Tech s.r.o, 2020 [cit. 2020-04-04]. Dostupné z: https://www.iqrf.org/technology/gw-daemon.

[20] *IQRF Gateway Documentation* [online]. IQRF Tech s.r.o, 2020 [cit. 2020-04-04]. Dostupné z: https://docs.iqrf.org/iqrf-gateway/introduction.html.

[21] *DPA* [online]. IQRF Tech s.r.o, 2020 [cit. 2020-04-04]. Dostupné z: https://www.iqrf.org/technology/dpa.

[22] *IQRF* [online]. iot-portal.cz [cit. 2020-04-04]. Dostupné z: https://www.iot-portal.cz/2017/11/27/iqrf/.

[23] *IDE* [online]. IQRF Tech s.r.o, 2020 [cit. 2020-04-04]. Dostupné z: https://www.iqrf.org/technology/ide.

[24] *Operating system* [online]. IQRF Tech s.r.o, 2020 [cit. 2020-04-04]. Dostupné z: https://www.iqrf.org/technology/os.

[25] *Apache Jmeter* [online]. 2020 [cit. 2020-30-03]. Dostupné z: https://jmeter.apache.org/.

[26] *Apache JMeter* [online]. en.wikipedia.org, 2020 [cit. 2020-20-03]. Dostupné z: https://en.wikipedia.org/wiki/Apache_JMeter.

[27] *Katalon* [online]. [cit. 2020-29-03]. Dostupné z: https://www.katalon.com/.

[28] *The Good and the Bad of Katalon Studio Automation Testing Tool* [online]. altexsoft.com, 2019 [cit. 2020-30-03]. Dostupné z: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-katalon-studio-automation-testing-tool/.

[29] *What is Katalon Studio?* [online]. artoftesting.com, 2019 [cit. 2020-29-03]. Dostupné z: https://artoftesting.com/katalon-studio-features-advantages-and-disadvantages.

[30] Limited, E. A. P. *What is JMeter? Why use JMeter? Advantages and disadvantages* [online]. linkedin.com, 2017 [cit. 2020-30-03]. Dostupné z: https://www.linkedin.com/pulse/what-jmeter-why-use-advantages-disadvantages-private-limited/.

[31] Mahajan, A. *Selenium vs Testcomplete* [online]. knowledgehut.com, 2019 [cit. 2020-30-03]. Dostupné z: https://www.knowledgehut.com/blog/software-testing/selenium-vs-testcomplete.

[32] *Diagnostika a testování elektronických systémů* [online]. umel.feec.vutbr.cz, 2012 [cit. 2020-16-01]. Dostupné z: http://www.umel.feec.vutbr.cz/bdts/index.php/diagnosticke-testy/dulezitost-testovani1cia.

[33] *Základy automatizace testování* [online]. test.swtestovani.cz [cit. 2020-29-03]. Dostupné z: http://test.swtestovani.cz/index.php?option=com_content&view=article&id=40:zaklady-automatizace-testovani&catid=3:zaklady&Itemid=11.

[34] *What is Non Functional Testing? Types with Example* [online]. guru99.com [cit. 2020-23-01]. Dostupné z: https://www.guru99.com/non-functional-testing.html.

[35] Pohanka, P. *Internet věcí* [online]. pavelpohanka.cz, 2017 [cit. 2020-25-01]. Dostupné z: http://www.pavelpohanka.cz/internet-of-things/.

[36] *Postman API testing* [online]. docs.calliope.pro, 2020 [cit. 2020-20-03]. Dostupné z: https://docs.calliope.pro/supported-tools/postman-tools/.

[37] *Selenium* [online]. 2020 [cit. 2020-29-03]. Dostupné z: https://www.selenium.dev/.

[38] *[Infographic] Challenges in Test Automation: Survey Key Results* [online]. katalon.com, 2018 [cit. 2020-29-03]. Dostupné z: https://www.katalon.com/resources-center/blog/infographic-challenges-test-automation/.

[39] Shobika. *Fundamental Test Process* [online]. letzdotesting.com, 2016 [cit. 2020-25-01]. Dostupné z: http://letzdotesting.com/fundamental-test-process/.

[40] Skokanová, B. M. *Metodika pre podporu riadenia testovania softvéru* [online]. 2018 [cit. 2020-23-01]. Dostupné z: https://theses.cz/id/he5z83/.

[41] *SoapUI* [online]. 2020 [cit. 2020-29-03]. Dostupné z: https://www.soapui.org/.

[42] Spurná, I. *Jak na IQRF: tři ukázky, jak pomocí protokolu DPA komunikovat s IoT zařízeními* [online]. root.cz, 2019 [cit. 2020-05-04]. Dostupné z: https://www.root.cz/clanky/jak-na-iqrf-tri-ukazky-jak-pomoci-protokolu-dpa-komunikovat-s-iot-zarizenimi/.

[43] Spurná, I. *Správa sítě IQRF s DPA Peer-to-Peer.* [online]. root.cz, 2020 [cit. 2020-04-04]. Dostupné z: https://www.root.cz/clanky/sprava-site-iqrf-s-dpa-peer-to-peer/.

[44] *Basic Concepts* [online]. tavern.readthedocs.io, 2020 [cit. 2020-20-03]. Dostupné z: https://tavern.readthedocs.io/en/latest/basics.html.

[45] *SmartBear* [online]. 2020 [cit. 2020-30-03]. Dostupné z: https://smartbear.com/.

[46] *Rozdiel medzi verifikáciu a validáciou* [online]. Kvizi.eu, 2008 [cit. 2020-16-01]. Dostupné z: https://www.kvizy.eu/slovnik-rozdielov/rozdiel/valid%C3%A1cia/verifik%C3%A1cia.

[47] Čermák, M. *Black box test* [online]. cleverandsmart.cz, 2010 [cit. 2020-21-01]. Dostupné z: https://www.cleverandsmart.cz/black-box-test/.

[48] Čermák, M. *Grey box test* [online]. cleverandsmart.cz, 2010 [cit. 2020-21-01]. Dostupné z: https://www.cleverandsmart.cz/grey-box-test/.

[49] Čermák, M. *White box test* [online]. cleverandsmart.cz, 2010 [cit. 2020-21-01]. Dostupné z: https://www.cleverandsmart.cz/white-box-test/.

Príloha A

Obsah pamäťového média

- **Daemon -** Obsahuje súbory pre správnu konfiguráciu Daemon
- **Doc -** Latexové zdrojové súbory a bakalárska práca v pdf
- **Emulator -** Zdrojový kód práce
- **test_realna_siet -** Obsahuje testy, ktoré sú určené na reálnu sieť
- **testcases -** Obsahuje testy, ktoré sú spúšťané na emulátore
- **.gitignore -** Obsahuje zoznam položiek, ktoré sú ignorované pri Git commit
- **.gitlab-ci.yml -** Súbor definujúci Gitlab CI
- **docker-compose.yml -** Docker compose súbor
- **Emulator.Dockerfile -** Docker file, ktorý má na starosti emulátor
- **README.md -** Obsahuje návod na spustenie
- **requirements.txt -** Obsahuje zoznam knižníc, ktoré sa nainštalujú pri zostavení
- **run.py -** Súbor obsahujúci kolekcie testov, ktoré sa majú spustiť
- **run_realna_siet.py -** Spúšťa testy na reálnej sieti
- **Testing.Dockerfile -** Docker file, ktorý má na starosti spustenie testov

Príloha B

Postup vytvárania nového testovacieho prípadu

Pri vytváraní nového testovacieho prípadu je doporučené sa držať opísaného postupu:

- Do priečinka *testcase* je potrebné vytvoriť súbor s príponou *.tavern.yaml*. V tomto súbore treba definovať konfiguráciu, ktorá bude rovnaká s ostatnými súbormi *.tavern.yaml*.
- Na základe mena periférie, ktorú chcete testovať, vytvoríte dva súbory s koncovkami *request.yaml* a *response.yaml*. Tieto dva súbory vložíte do priečinka, ktorého názov sa zhoduje s názvom testovanej periférie.
- V daných dvoch súboroch zadefinujete testovacie dáta vo formáte YAML.
- Po vytvorení testovacích dát je potrebné tieto dva súbory zadefinovať v súbore, ktorý bol vytvorený v prvom kroku.
- Posledným krokom je zadefinovať súbor *.tavern.yaml* v súbore *run.py*. Po tejto definícii je už všetko potrebné pripravené na spustenie vašich testov.