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 — **KAREL HANÁK**
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: **Hanák Karel**
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

Tato práce se zabývá testováním open source softwaru a automatizací testování. Cílem práce je navrhnout testy, implementovat testovací případy a integrovat je do prostředí vývoje softwaru pro správu sítí Internetu věcí (IoT). Software byl rozšířen o komunikační komponentu využívající protokol TCP, která slouží k navázání spojení s emulátorem IoT sítě. Pro testování a automatizaci byl zvolen nástroj Tavern. Celé řešení je rozděleno do obrazů nástroje Docker. Výsledné řešení je snadno rozšiřitelné o možnou budoucí funkcionalitu. Na základě testování provedeného s emulátorem sítě se podařilo objevit několik chyb v dokumentaci softwaru i samotném softwaru. Přínosem této práce je identifikace chyb a usnadnění testování v podobě emulátoru sítě, který umožňuje provádět automatizované testování bez nutnosti využití skutečných IoT zařízení.

Abstract

This thesis deals with open source software testing and automation of testing. The goal was to design tests, implement test cases and integrate them into the development environment of a software used to manage Internet of Things (IoT) networks. A new communication component using the TCP protocol was implemented to establish a connection with an IoT network emulator. A tool called Tavern was chosen for testing and automation. The entire solution is split into Docker images. The result can be easily extended with possible future functionality. As a result of testing with the network emulator, a handful of errors were found in both software documentation and the software itself. The main contribution of this thesis is the identification of errors as well as a way to simplify testing in the form of an IoT network emulator, allowing for automation of testing without the need for real IoT devices.

Klíčová slova

testování, automatizace, internet věcí, IoT, IQRF Tech s.r.o, Tavern, Docker, CI/CD

Keywords

testing, automation, internet of things, IoT, IQRF Tech s.r.o, Tavern, Docker, CI/CD

Citace

HANÁK, Karel. *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

Prohlášení

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

....
Karel Hanák
4. června 2020

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Pavolovi Korčekovi, Ph.D. za vedení a rady pro vypracování teoretické a praktické části této práce. Také bych rád poděkoval Matěji Mrázikovi za konzultace. Dále bych rád poděkoval Romanovi Ondráčkovi, Rostislavovi Špinarovi a Františkovi Mikulu z firmy IQRF Tech s.r.o za odborné konzultace.

Obsah

1 Úvod 2

2 Úvod do testování softwaru 3
2.1 Základní pojmy 3
2.2 Metody testování 5
2.3 Úrovně testování 6
2.4 Automatizované testování 9
2.5 Nástroje pro automatizaci procesu vývoje software 11
2.6 Nástroje pro testování aplikačního rozhraní 13

3 Testovaný software 16
3.1 IQRF Gateway Daemon 16
3.2 Technologie IQRF 18
3.3 Protokol IQRF DPA 19
3.4 Aplikační rozhraní IQRF JSON 21
3.5 Přehled poskytované funkcionality 22

4 Návrh řešení 24
4.1 Analýza zadání a požadavků na řešení 24
4.2 Zvolené nástroje 24
4.3 Komunikace Daemonu s emulovanou sítí 25
4.4 Emulace sítě 26

5 Implementace a testování 28
5.1 Komunikační komponenta IqrfTcp 28
5.2 Emulátor sítě 29
5.3 Testování 32
5.4 Zařazení do vývojového procesu 35

6 Závěr 36

Literatura 37

A Obsah přiloženého CD 40

B Tabulky testovacích sad 41

C Ukázka výstupu z CI/CD pipeline 44

Kapitola 1

Úvod

Software se v dnešní době vyskytuje prakticky všude. Najdeme jej ve stolních počítačích, v mobilních telefonech, chytrých náramcích a hodinkách, v běžných domácích spotřebičích jako je třeba pračka nebo mikrovlnná trouba ale také v dopravě, průmyslu, medicíně aj. Software je nedílnou součástí našeho života a často na něj spoléháme, aniž si to uvědomujeme. Takový software by měl fungovat správně. K odhalení chyb softwaru slouží testování softwaru, bez kterého se vývoj žádného softwaru neobejde. Ke zefektivnění testování se využívá automatizace testování, která spočívá v automatickém provádění manuálních testovacích případů pokaždé, když je software pozměněn.

Cílem této práce je seznámit se s rozhraním a funkcemi open source softwaru, navrhnout testy pro aplikačního rozhraní, implementovat tyto testovací případy a integrovat je do vhodně zvoleného prostředí pro automatické testování softwaru.

Práce je rozdělena na 4 větší kapitoly (až na úvod a závěr). Kapitola 2 je věnována problematice testování softwaru obecně, vysvětlení základních pojmů a principů, metod a úrovní testování softwaru. Dále jsou v této kapitole čtenáři představeny principy automatizace testování softwaru, nástroje umožňující automatizaci a také nástroje pro samotné testování.

Kapitola 3 seznámí čtenáře s open source softwarem zvoleným pro testování. Kapitola se zabývá účelem, funkcionalitou a rozhraním tohoto softwaru, a s ním spojenou technologií.

V kapitole 4 se čtenář dočte o aktuálním stavu testování softwaru, zadání práce je společně s požadavky na řešení analyzováno a celý problém je rozdělen na podproblémy. Tyto podproblémy jsou dále diskutovány a pro každý z nich je pak uveden návrh řešení.

Kapitola 5 je věnována implementačním detailům a testování. Je zde popsána implementace jednotlivých částí řešení a sestavování testovacích sad. Dále je popsáno jakým způsobem proběhlo testování a výsledky testování jsou diskutovány. Nakonec je popsána automatizace testování a integrace do prostředí vývoje softwaru tak, aby řešení autoři softwaru mohli používat a v případě potřeby dále rozšiřovat.

Kapitola 2

Úvod do testování softwaru

Cílem této kapitoly je čtenáře uvést do problematiky testování softwaru od základních pojmů, přes vybrané přístupy k testování, úrovně testování a různé testovací techniky. Kromě toho je část kapitoly věnována i automatizaci testování, nástrojům pro automatizaci a testování, a jejich využití při vývoji softwaru.

2.1 Základní pojmy

Na začátek je vhodné uvést základní pojmy z oblasti testování softwar. Přestože se jedná o poměrně známé pojmy, často si pod nimi lidé mohou představit něco jiného, než co skutečně znamenají.

Softwarová chyba

Ron Patton ve své knize o testování softwaru [22] definuje pojem softwarová chyba následovně:

- *Software nedělá něco, co by podle specifikace produktu dělat měl.*
- *Software dělá něco, co by podle údajů specifikace produktu dělat neměl.*
- *Software dělá něco o čem se produktová specifikace nezmiňuje.*
- *Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.*
- *Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo — podle názora testera softwaru — jej koncový uživatel nebude považovat za správný.*

Na základě řady studií provedených nad projekty různé velikosti byl vyvozen závěr že příčinou vzniku chyb je (v poměru od nejčastější po nejméně častou) specifikace, návrh, samotný programový kód nebo jiné příčiny. V případě specifikace je několik možností. Specifikace buď vůbec neexistuje, nebo existuje ale není dostatečně podrobná nebo se její obsah stále mění. Je také možné, že se jí někteří členové vývojového týmu dostatečně nevěnovali. Co se návrhu týče, chyby mohou vznikat protože je buď uspěchaný, často se mění, nebo nebyl dostatečně probrán. Chyby v programovém kódu mohou být zapříčeněněny složitostí softwaru, nedostatečná dokumentace, nedostatek času a nebo jen chyby z nepozornosti.

Mezi jiné příčiny spadají situace, které ostatní výše vysvětlené příčiny nezahrnují. Náklady na opravu závisí na tom, v jaké fázi vývoje softwaru jsou odhaleny. Je zřejmé, že náklady na opravu chyby odhalené během vytváření specifikace software jsou podstatně menší, než kdyby šlo o chybu odhalenou až po uvedení softwaru na trh [22].

Validace

Následující pojmy byly převzaty a přeloženy z knihy Paula Ammanna a Jeffa Offutta [1].

Validace je proces vyhodnocení zda software na konci vývoje je v souladu se zamýšleným využitím.

U validace je důležitá znalost oblasti, ve které se vyvíjený software bude využívat. Prostě řečeno, u validace jde o to, zda software splňuje, o co zákazník žádá. V tomto případě nejsou důležité detaily toho jak to software dělá, ale zda dělá to co od něj očekáváme.

Verifikace

Verifikace je proces rozhodování, zda produkty dané fáze procesu vývoje software splňují požadavky stanovené v předcházející fázi.

Verifikace je aktivita spíše technické povahy, při které se opíráme o znalosti softwarové specifikace a požadavků kladných na software. V případě verifikace už nám naopak jde o detaily implementace, které pro zákazníka nejsou důležité.

Další pojmy z oblasti testování testování

- **Testování** je vyhodnocování softwaru pozorováním jeho běhu.

  Pod testováním softwaru si také můžeme představit hledání chyb softwaru za účelem jejich opravení.

- **Selhání testu** je běh software, který skončí selháním.

  Test selže, například pokud se výstup testované funkce neshoduje s očekávaným výstupem.

- **Ladění** je proces hledání chyby spojené se selháním.

  Příkladem může být špatný operátor u matematické funkce, který způsobí selhání testu vyprodukováním špatného výstupu.

- **Hodnoty testovacího případu** jsou vstupní hodnoty potřebné k dokončení běhu testovaného softwaru.

- **Očekávaný výsledek** je výsledek, který bude vyprodukován při běhu testu jen a pouze pokud se program chová tak jak je zamýšleno.

- **Prefixové hodnoty** jsou vstupy potřebné k uvedení softwaru do stavu vhodného pro přijetí hodnot testovacího případu.

- **Postfixové hodnoty** jsou vstupy, které je potřeba předat softwaru poté co byly předány hodnoty testovacího případu.

- **Testovací případ** se skládá z hodnot testovacího případu, očekávaného výsledku, prefixových a postfixových hodnot potřebných pro dokončení běhu a vyhodnocení testovaného softwaru.
- **Testovací sada** je množina testovacích případů.

2.2 Metody testování

Statické a dynamické testování

Statické testování je pojem, pod kterým si můžeme představit testování softwaru, který nepracuje. Jde tedy pouze o pozorování a kontrolu. Pod pojmem **dynamické testování** si naopak můžeme představit testování běžícího programu [22]. S těmito pojmy je také spojené testování tzv. skříňky, skříňkou je v tomto kontextu právě testovaný software.

Testování černé skříňky

Testování černé skříňky spočívá v testování softwaru bez přístupu ke zdrojovým kódům, strukturám nebo architektuře. Tester v tomto případě ví jen co testovaný software má dělat, ale už neví jak to má dělat. Odtud tedy název černá skříňka, je to skříňka do které není vidět. Metody statického a dynamického testování je možné aplikovat na testování skříněk.

Statické testování černé skříňky představuje testování specifikace produktu. Tester má při tomto testování za úkol revidovat specifikaci, to znamená hledat v ní zásádní chyby či omyly, přehlédnuté, opomenuté a vynechané části. Patří sem také revize z pohledu zákazníka, kdy se tester v roli zákazníka snaží kontrolovat, zda specifikace produktu je v souladu s jeho potřebami. Dále je možné se zaměřit na obecné standardy a zásady, ať už firemní, oborové, vládní nebo hardwarové.

Dynamické testování černé skříňky je pak z dřívě vysvětlených pojmů testování bežícího softwaru, jehož zdrojový kód nemáme k dispozici a nevíme tedy jak pracuje, jen co dělá. Existuje množství principů dynamického testování černé skříňky. Jedním z nich jsou tzv. testy splněním a testy selháním. Testování splněním slouží k zjištění, zda je software vůbec schopen běhu a vykonávání toho co podle specifikace má dělat a to za normálních podmínek. Pokud software skutečně dělá to co má, pak přichází na řadu testování selháním a spočívá v prozkoumání chování v krajních případech za účelem odhalení chyby. Jinými slovy se snažíme software „rozbít“. Jedním z dalších principů je, že netestujeme funkci se všemi možnými vstupy. Naopak se snažíme množinu testovacích případů redukovat na menší ale stále efektivní. Vybereme tedy jeden nebo dva standardní případy a pak krajní či potenciálně problematické případy. Můžeme také testovat různé hodnoty, formát hodnot, kódování, výchozí hodnoty, nedefinované hodnoty a jiné, a sledovat jak se software zachová. Takových principů samozřejmě existuje mnohem více, ale pro ilustraci jsou zmíněné jen některé z nich. Více informací viz publikace Rona Pattona [22].

Testování bílé skříňky

Narozdíl od černé skříňky do bílé skříňky jako tester vidíme. Někdy se také označuje jak průhledná skříňka. Tester má zdrojový kód programu k dispozici a může jej studovat pro potřeby testování. Důležité je však dávat pozor na to, aby nevznikaly testovací sady, které

sedí na kód. S testováním bílé skříňky je pak spojený také pojem pokrytí kódu, kde se navrženými testy snažíme otestovat určité množství kódu. Pokrytí veškerého kódu nějaké funkce například s větvením by pak znamenalo vytvoření testovací sady, která postupně otestuje jednotlivé větve v rámci funkce a „pokryje“ tak její kód.

Statické testování bílé skříňky spočívá ve zkoumání a kontrole zdrojového kódu, ve kterém hledáme chyby bez jeho spuštění. V kódu nehledáme jen chyby, ale je důležité se zaměřit i na to co by v kódu mohlo chybět. Dalším aspektem je kontrola dodržování standardů a zásad pro kódování. Kód může sice fungovat správně, ale nemusí splňovat pravidla standardů a zásad programování v jednotlivých programovacích jazycích či obecně. Dodržováním standardů a zásad docílíme spolehlivějšího kódu, který je méně náchylný na chyby, je čitelný a snadno udržovatelný, a také je přenositelný, což je důležité při běhu programu na jiném hardwaru a kompilaci jinými kompilátory. Při zkoumání kódu je možné odhalit chyby jako neinicializované proměnné, špatnou indexaci ať už špatné syntaktické použití indexů, přístup mimo indexovatelný rozsah nebo přiřazení hodnoty špatného typu do proměnné. Dále jsou to chyby u deklarace proměnných jako špatné datové typy, délka proměnné, snadno zaměnitelné a podobné názvy proměnných, nevyužité proměnné či využívané proměnné, které nejsou deklarované. Mohou to být také chyby ve výpočtech jako nedefinované hodnoty funkcí a operátorů, špatné pořadí vyhodnocení částí výrazů kvůli prioritě operátorů, neošetření nekompatibility při kombinování různých datových typů. Častými chybami je také zaměnění pořadí znamének u porovnání, špatné znaménko, přiřazení do proměnné místo porovnání, nekonečný cyklus, předčasné ukončení cyklu, špatné větvení a mnoho dalších možných chyb.

Dynamické testování bílé skříňky je testování při kterém testujeme běžící program s tím, že máme k dispozici jeho zdrojový kód. Můžeme se rozhodnout co budeme testovat, co testovat nebudeme a jak k testování budeme přistupovat. Toto testování zahrnuje přímo testování funkcí, procedur, podprogramů a knihoven na nízké úrovni. Tedy testujeme aplikační programové rozhraní. Dále testování softwaru jako kompletního programu s tím že můžeme testy přizpůsobit podle toho co jsme zjistili ze zdrojového kódu. Za běhu programu můžeme kontrolovat hodnoty proměnných a stav softwaru. Je vhodné testovat software zároveň s jeho vývojem, otestováním menších částí předtím než jsou spojeny do většího celku se předejde složitým opravám na nejnižších úrovních programu. Nejprve se provádí testování jednotek a modulů, poté se testuje interakce jednotlivých jednotek a modulů, až se postupně na nejvyšší úrovni systémovým testováním otestuje celý program, nebo jeho podstatná část. Více informací viz publikace Rona Pattona [22].

Testování šedé skříňky

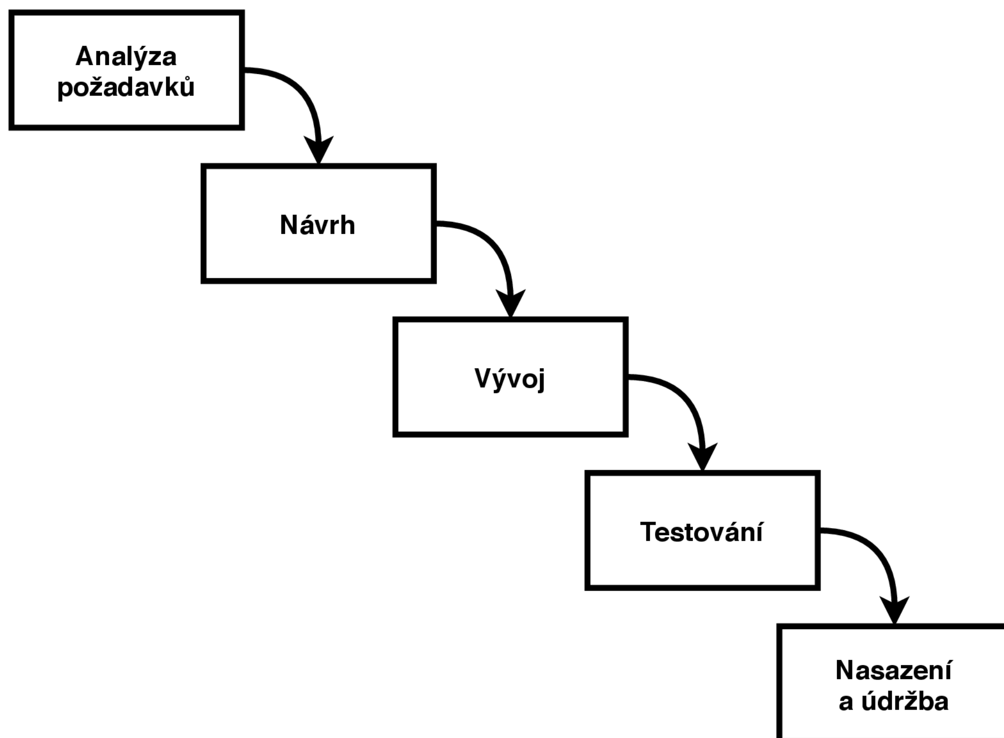
Testování šedé skříňky je metoda, která kombinuje testování černé skříňky a testování bílé skříňky. U testování šedé skříňky máme k dispozici datové struktury a používané algoritmy, ale nemáme k dispozici zdrojový kód. Případně může jít také o situaci, kdy tester ví jak pracují samostatné jednotky či moduly, ale neví jaká je mezi nimi interakce a testuje se pouze poskytované rozhraní [30].

2.3 Úrovně testování

Úrovně testování se vztahují k jednotlivým etapám životního cyklu vývoje softwaru. Už se nejedná o testovací metody, ale o aplikaci těchto principů při samotném testování softwaru. Životní cyklus vývoje softwaru je postup vývoje kvalitního softwaru od původního nápadu

a specifikace produktu až po jeho nasazení. Těchto postupů existuje více a jsou definovány modely životního cyklu vývoje softwaru.

Jedním ze základních modelů je **vodopádový model**. Projekt vyvíjený v modelu vodopádu sestupuje v posloupnosti kroků od původní myšlenky až k dokončenému produktu pro nasazení. Schéma vodopádového modelu je ukázáno na obrázku 2.1. V každém kroce je zhodnoceno, zda bylo dosaženo všeho potřebného a je možno přejít k dalšímu kroku. Jednotlivé kroky se nepřekrývají a jak již název „vodopádový“ model naznačuje, postupujeme pouze k nižším krokům a zpět se nevracíme. Kroky tohoto modelu jsou analýza požavků, návrh, vývoj, testování a nasazení výsledného produktu a jeho následná údržba. Velkou nevýhodou tohoto modelu je, že čím později je chyba odhalena, tím nákladnější je její opravení. Více informací viz publikace Rona Pattona [22].



Obrázek 2.1: Vodopádový model vývoje softwaru, převzato z [22] a upraveno.

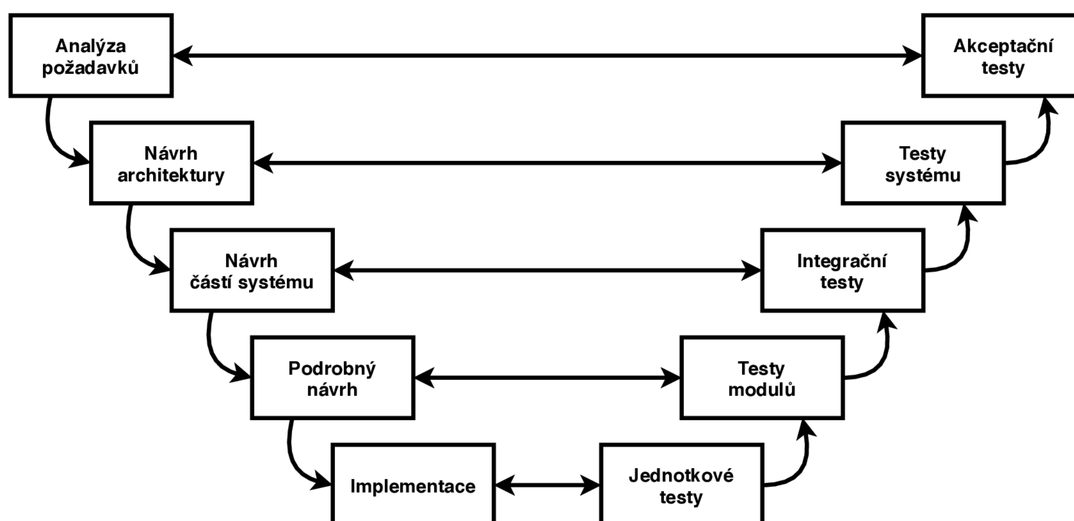
Model V

Model V vychází z vodopádového modelu, ale narozdíl od něj ke každé fázi vývoje náleží odpovídající úroveň testování. V praxi to znamená, že se provádí testování na různých úrovních zároveň s vývojem softwaru. Schéma modelu V je ukázáno na obrázku 2.2. Návrh testů pro jednotlivé fáze vývoje probíhá v rámci dané fáze nehledě na to, že software bude spustitelný nejdříve v implementační fázi. Model definuje následující fáze vývoje a odpovídající úrovně testování [1]:

- Analýza požadavků – Akceptační testy
- Návrh architektury – Testy systému
- Návrh částí systému – Integrační testy

- Podrobný návrh – Testy modulů
- Implementace – Jednotkové testy

Tyto fáze vývoje a s nimi související úrovně testování jsou vysvětleny v následujících podkapitolách.



Obrázek 2.2: Model V vývoje softwaru, převzato z [1].

Jednotkové testy

Jednotkové testy se týkají fáze implementace. V této fázi už vzniká kód a je tedy možné provádět testování na nejnižší úrovni. Testy se nazývají jednotkové, protože testují jednotlivé jednotky celého systému. Jednotkové testy vznikají zároveň s implementací jednotek. Jednotky mají mnoho názvů a většinou záleží na programovacím jazyce. V jazycích C/C++ se jim říká funkce, v Ada se jedná o procedury či funkce a v Javě metody. Testy může navrhovat a vytvářet tester, nebo v mnohých případech i samotný programátor [1]. Příkladem takového testu může být volání funkce pro výpočet součtu dvou čísel. Většinou se jedná o volání metody s hodnotami testovacího případu v rámci funkce assert, které se také dá očekávaný výstup. Tato funkce poté porovná výsledek volané metody s očekávaným výstupem a rozhodne o úspěchu či selhání testu.

Testy modulů

Fáze podrobného návrhu určuje strukturu a chování jednotlivých modulů. V této fázi také probíhá plánování testování modulů. Samotné testy pro moduly jsou však tvořeny až po implementaci. Modul je kolekce příbuzných jednotek, které jsou seskupeny ve zdrojovém souboru, balíčku či třídě. V C by se jednalo o soubor, v Ada o balíček a v C++/Java pak o třídu. Testování modulů slouží ke zhodnocení jednotlivých modulů, například vzájemná interakce jednotek v daném modelu a s jejich datovými strukturami. Testování modulů se provádí až po jednotkovém testování a mohou jej často provádět programátoři [1].

Integrační testy

Fáze návrhu částí systému specifikuje strukturu a chování podsystémů. Úkolem každého z těchto podsystémů je vyhovět nějakému požadavku na funkcionalitu v architektuře systému. Podsystémy často bývají adaptace dříve vyvinutého softwaru. V této fázi probíhá plánování integračního testování. Cílem integračního testování je posoudit, zda rozhraní mezi moduly podsystému správně komunikují a předpokládá se že samotné moduly pracují správně. Integrační testování provádí vývojáři a testeři [1].

Testy systému

Fáze návrhu architektury softwaru definuje komponenty a jejich vazby, které spolu tvoří systém, jehož specifikace má vyhovovat požadavkům zákazníka definovaným ve fázi analýzy požadavků. V této fázi probíhá plánování testování systému. Testy systému slouží k určení, zda sestavený systém odpovídá jeho specifikaci. Cílem tohoto testování je odhalit chyby v návrhu a ve specifikaci. Testování je většinou prováděno testovacím týmem, nikoliv programátory [1].

Akceptační testy

Fáze analýzy požadavků se zabývá potřebami cílového zákazníka. Jejím cílem je zjistit co od systému zákazník požaduje. Smyslem je neformální požadavky zákazníka zanalyzovat a definovat je formálním a strukturovaným způsobem, čímž se v ideálním případě předejde situaci, kdy zákazník po dodání produktu zjistí, že vlastně nedělá to co od něj očekával. Také se plánuje akceptační testování [17]. Akceptační testování slouží ke zjištění, zda software splňuje požadavky zákazníka. Toto testování vyžaduje přítomnost uživatelů či jiných individuí s vhodnou znalostí domény, ve které se software bude používat [1].

2.4 Automatizované testování

Testování je proces, který se neprovádí jen jednou. V průběhu vývoje softwaru probíhá testování mnohokrát, třeba při vytvoření nové funkce je nutné napsat nové testy a tuto funkci otestovat. Kontroluje se nejen to, že funkce se chová správně, ale také zda-li nová funkcionalita s sebou nepřinesla nové chyby v již existující implemetaci. Dále je nutné testovat existující funkcionalitu po odhalení a opravě zjištěných chyb, protože nemůžeme zaručit že jsme opravením dříve odhalených chyb nezavedli do systému jiné chyby. Tento proces opakovaného provádění testů se nazývá regresní testování. Vzhledem k tomu, že testovacích případů mohou být desítky, ale také tisíce, může být opakované manuální provádění testování časově náročné, únavné nebo dokonce i nemožné. Z tohoto důvodu využíváme testovací nástroje a automatizaci testování.

Testovací nástroje a automatizace testování přináší mnoho výhod. Jednou z nich je šetření času a peněz. Jelikož není nutné aby tester prováděl testy manuálně, provádění testovacích případů může být i tisícinásobně rychlejší. Pokud za nás testování provádí nástroje, můžeme se věnovat jiné práci. Takto získaný čas může být použit například pro plánování nových testovacích případů ať už pro stávající funkcionalitu, nebo nově připravovanou. Další výhodou je, že testovací nástroj se neunaví, nezačne se nudit a nedopustí se tak chyb z únavy nebo nepozornosti [22].

Pipeline

Pipe, nebo také roura je způsob přesměrování využívaný v unixových operačních systémech a jim podobných, a slouží k přesunutí výstupu jednoho programu na vstup jiného programu za účelem dalšího zpracování. Zpravidla se jedná o přesměrování standardního výstupu k nějakému cíli. Tímto cílem může být jiný program, soubor, nebo také výstupní periferní zařízení jako je tiskárna. V operačních systémech roury představují sekvence alespoň dvou příkazů oddělených svislou čarou v příkazové řádce. Toto přímé propojení mezi programy umožňuje přesunovat data mezi jednotlivými programy bez nutnosti vytvářnení dočasných souborů a jejich simultánní práci.

Modifikace tohoto principu, tzv. "CI/CD pipeline" (continuous integration/continuous delivery pipeline) se používá při vývoji softwaru a umožňuje automatizovat jednotlivé kroky celého procesu vývoje software od integrace nového kódu, testování a sjednocení větví repozitáře přes sestavení a otestování až po nasazení do produkce [18], [26].

Continuous Integration

Continuous integration, nebo také kontinuální integrace je prvním stadiem automatizovaného procesu vývoje softwaru. Umožňuje zefektivnit vývoj softwaru pro více vývojářů pracujících na stejném projektu zároveň. Vývojáři pracující na různých funkcionalitách v jednotlivých větvích repozitářů musí řešit potenciální konflikty vzniklé zařezením jejich kódu do již funkční aplikace a je možné že tak celá aplikace přestane fungovat. Některé organizace mají vyhrazené dny pro sjednocení změn všech vedlejších větví do hlavní větve repozitáře. Toto je možné zjednodušit právě pomocí kontinuální integrace, díky tomu, že po sjednodení změn v aplikace s předchozí verzí se spustí automatizované testy většinou na jednotkové a integrační úrovni. V případě konfliktů či chyb kontinuální integrace umožňuje opravit vzniklé chyby rychle. V tomto stadiu se sestaví kód s novými změnami, software se otestuje a pokud testy skončí úspěchem, může se nový kód sjednotit s ostatním kódem [26].

Continuous Delivery

Continuous delivery, nebo také kontinuální dodávka je druhým stadiem automatizovaného procesu vývoje software. V tomto stadiu jsou změny vývojáře již úspěšně otestovány a nahrány do společného repozitáře. Účelem tohoto stadia je mít vždy k dispozici validovaný kód, připravený pro nasazení do produkčního prostředí. Každý krok tohoto stadia je doprovázen automatizovaným testováním a uvolnením kódu [26].

Continuous Deployment

Continuous deployment, nebo také kontinuální nasazení je posledním stadiem automatizovaného procesu vývoje software. Toto stadium má za úkol automatizovat nasazení softwaru do produkce a oproti kontinuální dodávce také zajišťuje, že provedené změny, které prošly všemi stadii produkční pipeline se automaticky dostanou až ke koncovým uživatelům. Toto může být otázka i několika minut po napsání kódu a záleží na konfiguraci produkční pipeline jestli je nová verze poskytnuta každý týden, nebo jakmile je k dispozici. Díky tomu může být nasazení aplikace méně riskantní a nákladné, protože místo dlouhého čekání na vydání nové verze aplikace s více velkými změnami je možné uvolňovat menší změny a dostávat tak zpětnou vazbu od koncových uživatelů častěji a co nejdříve, a případné chyby nebo nedostatky vyřešit rychle [26].

2.5 Nástroje pro automatizaci procesu vývoje software

Tato podkapitola poskytuje čtenáři informace o vybraných, populárních nástrojích používaných pro automatizaci procesu vývoje software.

Docker

Tato podkapitola čerpá ze článku na webu InfoWorld [32]. Docker[1] je open source nástroj pro tvorbu kontejnerů, vývoj aplikací založených na kontejnerech a poskytuje nástroje pro sestavování, testování, dodávání a údržbu softwaru. Je dostupný pro operační systémy Linux, Windows i MacOS.

Kontejner je menší, odlehčené prostředí sdílející výpočetní prostředky s operačním systémem, ale jeho běh je od operačního systému izolován. Kontejnery vznikly za účelem izolování běhu aplikací na stejném zařízení od sebe bez nutnosti využití virtuálních strojů, které mohou být poměrně objemné. Od virtuálního stroje se liší tím, že nepotřebuje vlastní operační systém, využívá mnohanásobně méně prostředků, které mu jsou přiděleny, poskytuje jednoduchý a efektivní způsob kombinování softwarových komponent pro běh a jejich aktualizace a údržba je snadná.

Způsob, kterým se kontejner vytváří je také zcela odlišný od virtuálního stroje, kde se po spuštění operačního systému nainstalují vyžadované aplikace a následně se využívá dle potřeby. Kontejnery se sestavují na základě souboru nazývaného Dockerfile. Dockerfile je textový soubor využívající jednoduchou syntaxi pro specifikování instrukcí k sestavení tzv. obrazu Dockeru. V Dockerfile se určí operační systém na kterém kontejner poběží spolu s proměnnými prostředí, umístění souborů, nastavení sítě, další kompontenty potřebné pro běh a hlavně úkoly, které bude tento kontejner po spuštění vykonávat.

Poté co je Dockerfile napsán je možné pomocí utility Dockeru `build` vytvořit obraz Dockeru. Jedná se o přenositelný soubor, který obsahuje instrukce pro spuštění a běh softwarových komponent kontejneru. Po spuštění kontejner představuje instanci daného obrazu a takových instancí může běžet více zároveň. Kontejnery je také možné pozastavit a znovu spustit ve stejném stavu. Docker také umožňuje definovat vzájemnou interakci více kontejnerů pomocí nástroje Docker Compose. Takto je možné vyvíjet a testovat vícekontejnerové aplikace.

Jenkins

Jenkins[2] je open source automatizační server použitelný pro automatizaci procesů spojených s vývojem software. Je možné automatizovat sestavení, testování, dodávku i nasazení jako pipeline definované kódem. Tento server umožňuje napojení na git repozitáře populárních služeb jako je GitLab[3], GitHub[4], BitBucket[5] a jiné. Vzhledem k tomu, že je napsán v jazyce Java, je možné jej využít pro většinu platforem a navíc podporuje docker. Jenkins pipeline je složena ze sady pluginů[6] (zásuvné moduly rozšiřující funkcionalitu), které umožňují do Jenkins integrovat CI/CD pipeliny. Jenkins Pipeline je rozšiřitelná o další pluginy pro různé

[1] `https://www.docker.com/`
[2] `https://www.jenkins.io/`
[3] `https://about.gitlab.com/`
[4] `https://github.com/`
[5] `https://bitbucket.org/`
[6] Pluginy pro Jenkins – `https://plugins.jenkins.io/`

učely, včetně podpory pro jiné programovací jazyky. Jedná o poměrně populární a často používaný nástroj, díky čemuž takových pluginů existuje velké množství.

Jenkins pipeliny jsou definovány uživatelem v tzv. Jenkinsfile. Jedná se o textový soubor s konfigurací celého procesu sestavení, otestování a následného dodání aplikace. Pipeline se skládá z bloků node představující jednotlivé stroje v prostředí Jenkins schopné běhu pipeline. Node block pak umožňuje definovat jednotlivá stadia celého procesu v blocích stage. Nakonec, bloky stage se skládájí z jednotlivých kroků, neboli jednotlivých příkazů k vykonání. Je také možné nakonfigurovat prostředí, ve kterém bude pipeline spuštěna včetně proměnných tohoto prostředí. Výsledky testování je možné agregovat a vizualizovat v různých formátech. Pokud je potřeba, lze nadefinovat i kroky, které musí být obsloužené člověkem, třeba v případě kontroly. Je možné také definovat čístící a obslužné rutiny po dokončení stadií. Více informací viz uživatelská dokumentace Jenkins [16].

Travis CI

Travis CI[7] je nástroj umožňující sestavování, testování a nasazení softwaru. Jedná se o cloudovou aplikaci, kterou lze použít pouze pro repozitáře na GitHub nebo Bitbucket. Díky tomu, že se jedná o cloudovou službu, je možné provádět testování v různých prostředích, operačních systémech a na různých strojích. Travis CI je napsán v Ruby, ale podporuje většinu populárních programovacích jazyků. Jedná se o placenou službu, s tím že je zdarma pouze pro open source projekty, nikoliv komerční.

Repozitář na Github nebo Bitbucket při každém commitu aktivuje Travis CI, ten naklonuje repozitář do nového virutálního prostředí a provede všechny úkoly definované uživatelem v konfiguračním souboru. Jedná se o konfigurační soubor ve formátu yaml[8]). Travis CI poskytuje možnost provedení různých úkonů v jednotlivých fázích úkolu, např.: před, během a po instalaci závislostí, provedení skriptu a nasazení. Také umožňuje definovat čistící a obslužné rutiny v případě úspěšného i neúspěšného běhu.

Oproti Jenkins je Travis CI jednodušší k použití, ale nabízí méně možností pro přizpůsobení specifickým požadavkům. Konfigurace závisí na předem definovaném konfiguračním YAML souboru, kdežto Jenkins umožňuje kompletní přizpůsobení konfigurace dle potřeb. Více informací v dokumentaci Travis CI [3] a na webu guru99 [6].

Gitlab CI/CD

GitLab CI/CD[9] je nástroj zabudovaný do GitLab, který umožňuje využití principů kontinuální integrace, dodávky a nasazení bez nutnosti využití jiných nástrojů. Ačkoliv Gitlab je placená služba, základní funkcionality git repozitáře spolu se zabudovanými nástroji pro CI/CD jsou k dispozici zadarmo. Jelikož je nástroj přímo zabudovaný do Gitlab, je vždy k dispozici pro každý repozitář a je pouze otázkou konfigurace. Stejně jako předchozí nástroje poskytuje služby jako pipeline. Nabízí prostředky pro automatizaci sestavování a testování aplikací, analýzu zdrojového kódu, využití dockeru, různých prostředí včetně proměnných daných prostředí.

Podobně jako u Travis CI, GitLab CI/CD používá konfigurační soubor ve formátu yaml. V konfiguračním souboru se specifikuje tzv. runner (instance virtuálního stroje, která spouští předem definované rutiny) jako obraz dockeru, který vykoná příkazy skriptu. Dále

[7] `https://travis-ci.org/`

[8] YAML - YAML Ain't Markup Language - `https://yaml.org/`

[9] `https://docs.gitlab.com/ee/ci/`

je možné definovat samotné příkazy skriptu v jednotlivých stadiích, včetně toho co se má provést před a po vykonání skriptu. Je také možné specifikovat promínky pro vykonání či nevykonání jednotlivých úkolů, závislostí či událostí při kterých se jednotlivé úkoly mají provést. Při každém nahraném commitu do vzdáleného repozitáře se pak spustí celá pipeline definovaná v tomto konfiguračním souboru a je možné sledovat její průběh. Na konci každého běhu jsou pak vygenerované logy a zprávy. Více informací viz dokumentace GitLab CI/CD [5].

2.6 Nástroje pro testování aplikačního rozhraní

Tato podkapitola poskytuje informace o vybraných nástrojích pro testování a automatizaci, o tom kde se používají a za jakým účelem.

Postman

Postman[10] je kolaborační platforma pro vývoj API. Jedná se o placené nástroje, které umožňují odesílat požadavky na servery a přijímat od nich odpovědi, automatizovat manuální testy a integrovat je do CI/CD pipeline, navrhovat API a simulovat koncové body bez nutnosti existence serveru, který by požadavky zpracovával. Dále také poskytuje nástroje pro generování dokumentací, monitorování výkonu a doby čekání na odpověď. Jedná se však pouze o HTTP protokol.

Jedním z hlavních účelů je návrh a vývoj API za pomocí schémat definovatelných v několika formátech s možností jejich následné úpravy a verzování. Simulování koncových bodů umožňuje vývojářím lépe odhadnout jak jejich API bude fungovat ještě než začne samotný vývoj. Je možné takto simulovat komunikaci s existujícími aplikacemi a službami. Kromě toho Postman také umožňuje psát testovací sady pro jednotlivé navrhované požadavky v JavaScriptu, ale lze je také sestavit bez něj pomocí specifikování očekávaných hodnot. Testy je také možné parametrizovat tak, že se použijí proměnné, do kterých se vloží hodnoty z datových souborů nebo z proměnných prostředí. Testovací sady je pak možné spouštět s runnerem v rámci Postmanu, pozorovat průběh testů a procházet logy. Existuje také Newman, který plní stejný účel, ale narozdíl od Postmanu nemá neposkytuje grafické uživatelské rozhraní a podporuje jen příkazovou řádku. Díky tomu je možné jej použít pro integraci do CI/CD pipeline. Více informací viz webové stránky nástroje Postman [23], [24], [25].

Tavern

Tavern[11] je framework pro testování API pro protokoly HTTP a MQTT. Je postavený na frameworku pytest[12], který umožňuje psát testy v jazyce Python. Je jednoduchý, snadný k použití a vzhledem k tomu, že je postaven na Python/pytest rozšiřitelný pro specifické potřeby. Jedná se o nástroj, který se používá pomocí příkazové řádky a je možné jej integrovat CI/CD pipeline.

Testy jsou definované v YAMLu, každý test se skládá z názvu testu, jednoho nebo více stadií s identifikátory, požadavkem a odpovědí. Po spuštění testovacího případu je požadavek odeslán na server a odpověď představuje to co očekáváme že dostaneme. Po přijetí odpovědi je tato odpověď porovnána s očekávanou a test je vyhodnocen jako úspěšný

[10] `https://www.postman.com/`
[11] `https://taverntesting.github.io/`
[12] Pytest - `https://docs.pytest.org/en/latest/`

nebo neúspěšný. V rámci testovacích případů je také možné ukládat hodnoty z odpovědí, které mohou být použity v budoucích požadavcích. V požadavcích je také možné vytvořit proměnné, do kterých budou pak jen vložené specifické hodnoty, aby se nemusely hodnoty do každého testu vkládat "natvrdo". Kromě těchto specifických proměnných je možné definovat také proměnné prostředí. Co se samotných odpovědí týče, Tavern má zabudované validátory, ale je také možné naprogramovat funkci, která vezme položku z odpovědi a zkontroluje ji, tato funkce se pak automaticky zavolá při přijetí odpovědi pokud je specifikovaná v testovacím případě. Pokud není nutná až tak striktní kontrola vrácených hodnot, Tavern také nabízí klíčová slova, které porovnají vrácené hodnoty na datové typy. Stejně tak je možné volat funkce pro generování hodnot pro požadavky, nebo předat JSON soubor s položkami a hodnotami pro požadavek. V neposlední řadě poskytuje Tavern prostředky pro logování a debuggování testů pokud je využit plugin pytest. Více informací viz dokumentace nástroje Tavern [2].

Patriot

Patriot[13] je framework pro testování IoT aplikací. Jedná se o rozšíření frameworku jUnit5 napsané v Javě. Nabízí možnost integrace do CI/CD nástrojů či integrovaných vývojových prostředí. Umožňuje vývojářům simulovat síťová spojení mezi jednotlivými komponentami testovaného systému. Kromě toho také dokáže emulovat různé typy fyzickch zařízení a generovat data potřebná k emulování realistického chování zařízení, které napodobují.

Skládá se z modulů pro aplikační rozhraní, síť a generování dat. Patriot api je rozšířením jUnit frameworku a poskytuje potřebné prostředky pro efektivní testování IoT aplikací a vůbec umožňuje integrační testování. Rozšíření zahrnuje metody pro konfiguraci před a po provedení testů, správu simulovaného prostředí a kontrolu topologie sítě. Síť je simulovaná pomocí dockeru a základní komponentou je router. Router představuje zařízení, které propojuje 2 nebo více sítí na síťové vrstvě a posílá mezi nimi data. K tomu používá směrovací tabulky. Síťový modul poskytuje potřebné metody a třídy pro definování routerů, sítí, topologie sítě a směrovacích tabulek. Modul pro generování dat pak poskytuje metody pro generování různých dat ze simulovaných senzorů nebo aktuátorů jako jsou například teploměr, vlhkoměr a jiné. Více informací viz web nástroje Patriot [21] a jeho dokumentace [20].

UFT

UFT[14] je sada nástrojů pro automatizaci funkcionálního testování desktopových, webových a mobilních aplikací. Také umožňuje automatizaci testování API a robotických procesů. Primárně se využívá pro funkcionální a regresní testování. Podporovaný operační systém je pouze Microsoft Windows a jediným podporovaným jazykem je VBScript. Jedná se o placený nástroj.

UFT umožňuje automatizaci interakce uživatele s webovou či klientskou desktopovou aplikací. K tomu poskytuje možnost nahrávání uživatelem provedených akcí, které zaznamenává jako posloupnost příkazů v jazyce VBScript. Je samozřejmě možné skripty také psát. Tyto zaznamenané posloupnosti lze pak přehrávat a použít je k automatizaci testování aplikací. V základu nepodporuje všechny možné technologie, ale je možné nainstalovat pluginy, které podporu pro různé technologie doplní. Příkladem mohou být pluginy pro

[13] https://patriot-framework.io/

[14] https://www.microfocus.com/en-us/products/uft-one/overview

více populární technologie jako je .NET, Java, aplikace od Oracle, emulátory terminálů, webových služeb a jiné. Více informací viz web LearnQTP [15] a guru99 [7].

TestComplete

TestComplete[15] je nástroj pro automatizované funkcionální testování uživatelského rozhraní. Jedná se o populární, ale placený nástroj. Poskytuje prostředky pro testování uživatelských rozhraní desktopových, webových i mobilních aplikací. Podporované platformy jsou Microsoft Windows, Android a iOS. Nabízí možnost integrace do CI systémů.

Nabízí možnost testování mezi různými prohlížeči, automatické testování lze nahrát na jednom prohlížeči a pak jej přehrát a spustit na jiných prohlížečích nebo v cloudu. Kromě nahrávání a přehrávání testů je také možné testy vytvořit pomocí klíčových slov, nebo pomocí skriptů napsaných v JavaScriptu, Pythonu nebo VBScriptu. Nabízí také příkazovou řádku a API, se kterými je možné automatizované testy integrovat do CI systémů, verzovacích systémů jako Git i nástrojů pro správu testů a sledování bugů. Testy je možné pouštět paralelně na různých zařízeních. Dříve zmíněná možnost nahrávání testů zahrnuje také gesta pro mobilní aplikace. Více informací viz web nástroje TestComplete [27], [28], [29].

Selenium

Selenium[16] je open source sada nástrojů a knihoven, která umožňuje automatizaci webových prohlížečů. Poskytuje rozšíření pro emulování interakce uživatele s webovým prohlížečem. Je tak možné naprogramovat sady různých scénářů a testovat webové aplikace. Mezi podporované jazyky patří Java, Python, C#, Ruby, JavaScript, PHP a další populární jazyky. Oficiálně podporované webové prohlížeče jsou Chrome, Firefox, Internet Explorer, Opera a Safari. Narozdíl od UFT je zcela zdarma.

Selenium nabízí 3 nástroje. Prvním z nich je WebDriver. WebDriver se používá k automatizaci webových prohlížečů. Každému prohlížeči náleží specifická implementace WebDriver a zprostředkovává komunikaci mezi Selenium a prohlížečem. Vývojáři poskytuje objektově orientované API. Dalším nástrojem je IDE (integrované vývojové prostředí). Jedná se o snadno použitelné rozšíření pro prohlížeče Chrome a Firefox a využívá se k vytváření testovacích případů. Rozšíření zaznamenává akce, které uživatel provádí a na základě nich sestaví Selenium posloupnost příkazů s parametry jednotlivých prvků webových stránek. Posledním nástrojem je Selenium Grid. Jedná se o nástroj, který umožňuje spouštění testovacích případů na různých zařízeních a platformách. Nabízí plnou kontrolu nad podmínkami a událostmi, které musí nastat pro spuštění testů. Grid vytváří síť serverů, kde jeden ze serverů rozesílá příkazy na vzdálené instance webových prohlížečů, neboli uzly sítě. Dále umožňuje spravovat verze a konfigurace jednotlivých prohlížečů centrálně. Více informací viz dokumentace nástroje Selenium [31].

[15] `https://smartbear.com/product/testcomplete/overview/`
[16] `https://www.selenium.dev/`

Kapitola 3

Testovaný software

Pro testování byl vybrán open source software IQRF Gateway Daemon[1], který je součástí IQRF ekosystému a kterému je věnována kapitola 3.1. IQRF ekosystém může být chápán jako podmnožina Internetu věcí (z anglického Internet of Things, dále jen "IoT"), zahrnující koncová zařízení, brány, cloudové či další služby a aplikace pro vybudování IoT sítě.

IoT je koncepce připojení libovolného zařízení k Internetu a dále propojení s ostatními připojenými zařízeními. Tato zařízení je možné monitorovat, využít ke sběru velkého množství dat o sobě a okolním prostředí. Kromě toho je také možné tato zařízení ovládat. Obecně se jedná o zařízení s vestavěnými senzory připojená k nějaké IoT platformě, která získává data z různých senzorů a na základě analýzy těchto dat je možné učinit informovaná rozhodnutí, detekovat problémy a v některých případech problémům i předcházet, zefektivnit procesy nebo dokonce automatizovat určité úlohy. Zařízení data přijímají a vysílají pomocí bezdrátových sítí. Často je také nazýváme chytrá zařízení [4].

Tato zařízení lze využívat například k ovládání světel pomocí desktopové nebo mobilní aplikace. Některá zařízení umožňují dokonce i ovládání hlasem. Stejně tak je možné nastavit termostat pomocí mobilní aplikace po cestě domů. Příkladem automatizace pak může být chytrý termostat, který na základě mé pozice získané z mého chytrého telefonu zjistí, že jsem na cestě domů a začne dům vytápět dopředu.

3.1 IQRF Gateway Daemon

IQRF Gateway Daemon (dále jen Daemon) je open-source software, který umožňuje snadno vytvořit bránu IQRF disponující připojením k Internetu a cloudovým službám z libovolného linuxového stroje nebo do existující brány zakomponovat IQRF konektivitu. Takto vytvořená brána poté řídí síť IQMESH, které je společně s technologií IQRF věnována kapitola 3.2. Software poskytuje aplikační rozhraní pro správu sítí a jím naležící zařízení. Aplikační rozhraní je popsáno v kapitole 3.4. Bránu vytvořenou pomocí Daemonu je možné spravovat pohodlným a zabezpečeným způsobem pomocí webové aplikace IQRF Gateway Webapp nebo jiné, například cloudové aplikace [10].

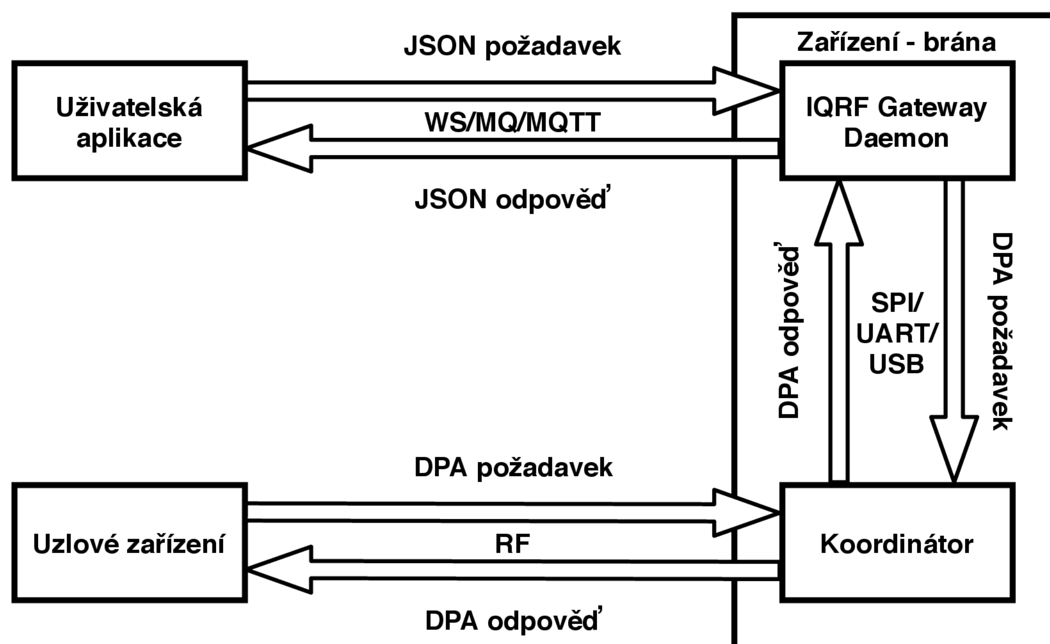
Daemon slouží jako prostředník mezi spravovanou sítí a například uživatelem, který danou síť spravuje a monitoruje. Aplikační rozhraní, které Daemon poskytuje využívá serializační formát JSON[2] pro přenos zpráv. Zprávy mezi uživatelskou aplikací a Daemonem

[1] `https://www.iqrf.org/technology/gw-daemon`
[2] JSON - JavaScript Object Notation - `https://www.json.org/json-en.html`

jsou doručovány pomocí kanálu WebSocket[3], Message Queue[4] nebo MQTT[5]. Mezi Daemonem a zařízeními v síti se pak vyměňují zprávy ve speciálním protokolu, kterému je věnována kapitola 3.3. Pro jejich přenos se využívají rozhraní SPI (Serial Peripheral Interface), UART (Universal asynchronous receive-transmitter) nebo USB (universal serial bus).

Příklad zpracování požadavku

Příkladem komunikace může být požadavek na zjištění aktuální teploty v místnosti. Uživatel pomocí webové aplikace IQRF Gateway Webapp zvolí požadavek na zjištění teploty, dále zvolí cílové uzlové zařízení, ze kterého chce teplotu vyčíst a požadavek odešle. Tento požadavek je odeslán jako zpráva ve formátu JSON pomocí kanálu WebSocket na adresu a port stroje, kde běží Daemon, ten zprávu přijme, a formát JSON zpracuje. Zpracovaná data transformuje na paket DPA, který předá koordinátoru (řídícímu zařízení) v síti pomocí dříve zmíněných rozhraní SPI, UART nebo USB. Následně je paket pomocí skoků přes jednotlivá zařízení v síti doručen na cílové zařízení s teploměřem, které paket s požadavkem přijme, sestaví odpověď a paket obsahující odpověď s teplotou odešle zpět na koordinátor stejným způsobem. Tento paket je předán Daemonu, který jej zpracuje a transformuje data do formátu JSON a opět pomocí kanálu WebSocket odešle odpověď do webové aplikace, ze které původní požadavek přišel. Celý proces je také znázorněn na obrázku 3.1.



Obrázek 3.1: Schéma komunikace mezi webovou aplikací, Daemonem a zařízeními při zpracování požadavku.

[3]https://tools.ietf.org/html/rfc6455
[4]https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html
[5]http://mqtt.org/

3.2 Technologie IQRF

Technologie IQRF umožňuje připojit zařízení k IoT pomocí bezdrátové smíšené síťové topologie IQMESH a byla původně vyvíjena společností MICRORISC[6], v dnešní době má vývoj na starosti odvozená společnost IQRF Tech s.r.o[7]. Technologie IQRF poskytuje pro výrobce elektronických produktů bezdrátovou konektivitu včetně připojení do sítě Internetu a definuje standard, jehož implementace umožňuje snadnou integraci a interoperabilitu zařízení od různých výrobců v jedné síti.

Jedná se o platformu pro bezdrátové připojení s nízkým výkonem, nízkou rychlostí, malým množstvím přenášených dat a nízkou spotřebou energie. V jedné síti může existovat až 240 zařízení a délka skoku od jednoho zařízení k druhému je několik desítek metrů uvnitř a několik set metrů venku. Data se přenášejí v paketech a maximální velikost paketu přenášených dat je 64B. Využívají se radiofrekvenční pásma ISM[8], jejichž použití není zpoplatněno. Konkrétně jde o pásma 868 MHz pro Evropu, 916 MHz pro Ameriku a 433 MHz pro celosvětové užití [11].

Základní komunikační komponentou IQRF je tzv. transciever. Název vznikl spojením slov transmitter (vysílač) a receiver (přijímač). Jedná se o malou inteligentní elektronickou desku s veškerou potřebnou obvodovou výbavou pro implementaci bezdrátového vysokofrekvenčního připojení. Mikrokontrolér obsahuje operační systém IQRF, který implementuje všechny standardní funkce těchto transcieverů. V případě potřeby je možné další funkcionalitu doprogramovat v programovacím jazyce C. Specifikace transcieverů IQRF je k dispozici na webu IQRF[9] [12].

IQMESH

IQMESH je protokol popisující výměnu dat mezi zařízeními komunikujícími v síti IQRF. Jedná se topologii sítě, kde jednotlivé uzly mohou být propojeny mezi sebou a ne jen propojeny s centrálním zařízením. Síť může být typu STD, která podporuje pouze zařízení ve standardním režimu, nebo STD+LP, která podporuje i zařízení v režimu nízké spotřeby, taková síť je pak o něco pomalejší oproti standardní síti.

V jediné síti může existovat až 240 zařízení a jsou to zařízení dvou typů. Prvním z nich je koordinátor, který slouží jako řídící prvek v síti. Tento typ zařízení se v síti vyskytuje pouze jednou. Druhým typem zařízení je uzel, typicky se jedná o zařízení, kterému jsou zasílány příkazy s nastavením pokud se například jedná o zařízení, které ovládá osvětlení. Z uzlových zařízení se také dají získávat data, například pokud se jedná o nějaký senzor. Důležité je, že každé zařízení v síti je schopné přeposílat pakety přicházející od koordinátoru.

Díky možnosti propojení jednotlivých uzlů můžeme vytvořit síť, ve které jsou jednotlivé uzly dosažitelné více cestami, taková síť je pak robustnější a spolehlivější narozdíl od topologie hvězda, kde koncová zařízení jsou připojena k jednomu centrálnímu zařízení. [13] Příkladem takové sítě může být síť spravující veřejné osvětlení na ulici, koordinátor je zde bránou a uzly jsou jednotlivá světla této ulice.

[6] `https://www.microrisc.com/cs`

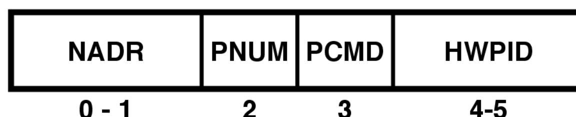
[7] `https://www.iqrf.tech/`

[8] ISM - Industrial, scientific and medicinal (Průmyslové, vědecké a zdravotnické)

[9] `https://www.iqrf.org/products/transceivers`

3.3 Protokol IQRF DPA

Direct Peripheral Access (DPA) je jednoduchý protokol pro správu služeb a periférií zařízení sítě IQMESH. Protokol využívá strukturované zprávy pro výměnu informací. Data jsou zakódována do posloupnosti znaků v hexadecimální soustavě. Každá zpráva obsahuje 4 povinné složky dle obrázku 3.2.



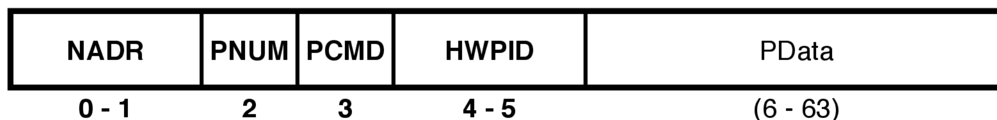
Obrázek 3.2: Povinné složky DPA zprávy s označením pozic bajtů.

- `NADR` – Adresa zařízení v rámci sítě IQMESH. Pro tuto položku jsou vyhrazeny 2 bajty, ale reálně se vyšší byte zatím nepoužívá. Příklad adresy: 0x0002 – uzel na adrese 2.
- `PNUM` – Číslo periferie cílového zařízení. Tato položka specifikuje, se kterou periferií se bude pracovat a využívá pouze 1 bajt. Příklad zařízení: 0x06 – červená led dioda.
- `PCMD` – Číslo příkazu pro zvolenou periferii. Opět se zde využívá pouze 1 bajt. Příklad příkazu: 0x04 – červená led dioda má začít blikat.
- `HWPID` – Profilové ID hardware, které specifikuje typ zařízení a periferie, které implementuje, jeho chování etc. Využívá 2 bajty. Požadavek je obsloužen pouze pokud `HWPID` v přijaté zprávě odpovídá `HWPID` cílového zařízení. Pokud je jeho hodnota 0xFFFF, pak se HWPID nekontroluje a zařízení požadavek vykoná.

Kromě povinných parametrů je pak vyhrazeno zbývajících 58 bajtů pro návratové hodnoty. Protokol DPA definuje 4 typy zpráv: požadavek, potvrzení, oznámení a odpověď. Popis možných hodnot těchto parametrů je k dispozici v dokumentaci DPA.

DPA Request

Zpráva typu požadavek. Skládá se z výše vysvětlené čtveřice paramterů a může navíc může obsahovat data v sekci `PData` pokud jsou součástí konkrétního požadavku. Velikost části PData záleží na typu požadavku. Schéma požadavku na obrázku 3.3 níže.



Obrázek 3.3: Složky DPA zprávy typu požadavek s označením pozic bajtů.

DPA Confirmation

Zpráva typu potvrzení. Slouží pro potvrzení přijetí požadavku DPA. Odesílá jej uzlové zařízení zpět na koordinátor. Kromě čtveřice povinných parametrů obsahuje speciální bit

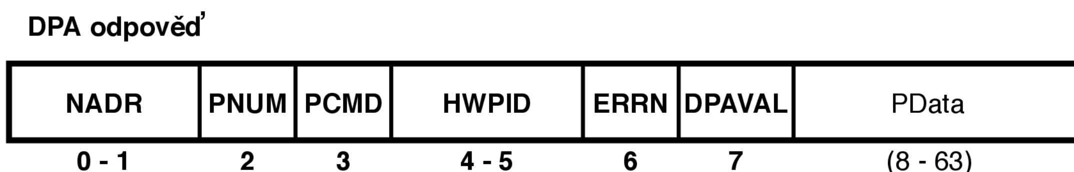
značící potvrzovací zprávu, DPA hodnotu zařízení (stav a chování v síti), počet skoků v síti pro doručení požadavku adresovanému zařízení, doba předání zprávy a počet skoků v síti pro doručení odpovědi na koordinátor. Tato zpráva se odesílá pouze pokud je požadavek poslán na vzdálené zařízení.

DPA Notification

Zpráva typu oznámení. Používá se, pokud na uzlovém zařízení byl zpracován požadavek, který není read-only, tedy pokud se mění konfigurace cílového zařízení. Obsahuje stejnou čtveřici jako přijatý požadavek, s tím, že parametr `NADR` je nastaven na adresu odesílatele. Do `HWPID` pak zařízení vloží své vlastní `HWPID` a zprávu odešle na koordinátor.

DPA Response

Zpráva typu odpověď. Slouží pro předání odpovědi na DPA požadavek. Schéma odpovědi je k dispozici na obrázku 3.4. Odpověď se skládá ze stejné čtveřice jako původní požadavek, s tím, že nejvyšší bit parametru `PCMD` je nastaven tak aby specifikoval odpověď. Stejně jako u zprávy typu oznámení zařízení, které zpět odesílá odpověď nahradí původní hodnotu `HWPID` v požadavku svou vlastní hodnotou `HWPID`. Dále pak obsahuje 1 bajt pro kód odpovědi `ERRN`, další bajt pro DPA hodnotu zařízení `DPAVAL` a samotná data odpovědi, pokud požadavek v odpovědi očekává data z periferie. Velikost části PData záleží na typu požadavku.



Obrázek 3.4: Složky DPA zprávy typu odpověď s označením pozic bajtů.

Příklad komunikace

Jako příklad zde uvedu ukázky z dokumentace DPA. Jedná se o zapnutí zelené led diody na vzdáleném uzlu na adrese 0x0A:

- DPA Request
  NADR = 0x000A, PNUM = 0x07, PCMD = 0x01, HWPID = 0xFFFF
- DPA Confirmation
  NADR = 0x000A, PNUM = 0x07, PCMD = 0x01, HWID = 0xFFFF, PData = 0xFF, 0x07, 0x06, 0x04, 0x06
- DPA Notification
  NADR = 0x0000, PNUM = 0x07, PCMD = 0x01, HWPID = 0xABCD
- DPA Response
  NADR = 0x000A, PNUM = 0x07, PCMD = 0x81, HWPID = 0xABCD, PData = 0x00, 0x06

Více informací o protokolu DPA je k dispozici v dokumentaci DPA [9].

3.4 Aplikační rozhraní IQRF JSON

Zprávy na úrovni aplikačního rozhraní jsou ve formátu JSON. Zde existují pouze 2 typy zpráv, a sice požadavek a odpověď. Příklady zpráv typu požadavek a odpověď jsou uvedeny níže ve výpisech 3.1 a 3.2. Každá zpráva typu požadavek má následující povinné parametry:

- `mType`
  Řetězec představující typ odesílané zprávy, Daemon na základě tohoto parametru zpracuje zbytek JSON zprávy odpovídajícím způsobem. Příklad: iqrfEmbedLedg_Set slouží k rozsvícení či zhasnutí zelené led diody na zařízení.
- `data`
  Objekt, který nese data zprávy, obsahuje identifikátor požadavku, informace pro zvolení cílového zařízení a parametry pro nastavení periferií daného zařízení.
  - `msgId`
    Řetězec pro párování požadavaku s odpovědí.
  - `req`
    Představuje objekt nesoucí data pro zpracování a ze kterých se pak sestaví požadavek ve formátu DPA.

Existují 3 typy požadavků. Prvním z nich je tzv. `raw`, který využívá typ zprávy `iqrfRaw`, kde se v objektu `req` odesílá požadavek přímo ve formátu DPA v řetězci `rData`. Takto je možné odeslat jakýkoliv požadavek. Druhý je požadavek typu `rawHdp`, který narozdíl od `raw` umožňuje specifikovat požadavek pomocí čísel v dekadické soustavě. V JSON formátu zprávy se vyplní položky pro adresu zařízení, číslo periferie, číslo, příkazu a pole hodnot představující volitelná data části PData. Poslední typ požadavků už je specifický požadavek pro každou periferii a zprávu. Konkrétní typy zpráv pak mají vlastní parametry.

Odpověď ve formátu JSON obsahuje identický `mType`, odpovídající `msgId` a data odpovědi. Odpověď vždy obsahuje objekt `raw`, který nese řetězce `request` s požadavkem ve formátu DPA, `confirmation` s potvrzením v DPA formátu a `response` s odpovědí v DPA formátu. Ke každé z těchto položek také náleží časová známka. V případě odpovědi na požadavek `iqrfRaw`, je vrácena odpověď ve formátu DPA v objektu `rsp`, řetězci `rData`. Pokud se však nejedná o raw message, odpověď je Daemonem zpracována a rozdělena na jednotlivé pojmenované položky v objektu `rsp`. Nakonec má také každá odpověď řetězcovou i celočíselnou reprezentaci návratové hodnoty.

```
{
  "mType": "iqrfRaw",
  "data": {
    "msgId": "testRaw",
    "timeout": 1000,
    "req": {
      "rData": "00.00.06.03.FF.FF"
    },
    "returnVerbose": true
  }
}
```

Výpis 3.1: JSON požadavek typu raw message

```
{
    "mType": "iqrfRaw",
    "data": {
        "msgId": "testRaw",
        "timeout": 1000,
        "rsp": {
            "rData": "00.00.06.83.00.00.00.44"
        },
        "raw": {
            "request": "00.00.06.03.ff.ff",
            "requestTs": "2018-08-31T12:48:35.887733",
            "confirmation": "",
            "confirmationTs": "",
            "response": "00.00.06.83.00.00.00.44",
            "responseTs": "2018-08-31T12:48:35.915579"
        },
        "insId": "iqrfgd2-1",
        "statusStr": "ok",
        "status": 0
    }
}
```

Výpis 3.2: JSON odpověď na požadavek typu raw message

Více informací je k dispozici na stránkách dokumentace aplikačního rozhraní [14].

3.5 Přehled poskytované funkcionality

Tato část poskytuje přehled periferií zařízení v síti IQMESH a funkcí k obsluze periferií, které Daemon nabízí.

Funkce standardu IQRF

Standard IQRF[10] v rámci interoperability zařízení od různých výrobců definuje funkce pro správu binárního výstupu, světel, různých senzorů, kterými zařízení disponují. Kromě toho také umožňuje zasílat zprávy pro rozhraní DALI[11].

- Binární výstup – výčet implementovaných výstupů, nastavení hodnoty binárního výstupu a vrácení předchozí hodnoty.
- Světla – výčet implementovaných světel, nastavení intenzity světla, zvýšení a snížení intenzity světla.
- Senzory – výčet implementovaných senzorů, čtení dat ze senzorů popřípadě čtení dat ze senzorů včetně typu senzoru.

[10] Standard IQRF - https://www.iqrfalliance.org/iqrf-interoperability/

[11] DALI - Digital Addressable Lighting Interface - https://www.digitalilluminationinterface.org/dali/

- Rozhraní DALI – odesílání synchronních a asynchronních zpráv do sběrnice DALI a získávání odpovědí.

Funkce protokolu DPA

Protokol DPA definuje zprávy pro správu periferií jednotlivých zařízení sítě IQMESH. Mezi standardní periferie popisované protokolem PDA patří koordinátor sítě a uzly, operační systém transcieverů, interní paměti mikrokontroléru EEPROM, externí sériové paměti EEEPROM, interní paměti RAM, LED diody, teploměr, vstup a výstup, rozhraní SPI a UART a modul Fast Response Command (FRC). Kromě toho také popisuje funkce pro průzkum sítě a zjištění informací o zařízení [8].

- Průzkum sítě – výčet zařízení a v síti a jejich periferií, které jsou implementované a dále aktivní.
- Koordinátor – zjištění výskytu, počtu a adres uzlů propojených s koordinátorem, odebrání všech uzlů, přidání či odebrání konkrétního uzlu, nastavení počtu skoků pro přenost požadavku a odpovědi, záloha a obnovení konfigurace koordinátoru, přidělení specifického identifikátoru modulu a jiné.
- Uzel — čtení informací z uzlu a jeho konfirgurace, restartování uzlu, záloha a obnova konfigurace uzlu, validace propojení s koordinátorem.
- Operační systém -– zjištění informací o zařízení, vyresetování, restartování a uspání transcieveru, čtení a zápis konfigurace transcieveru, nastavení zabezpečení, provedení několika DPA požadavků najednou, aktualizace firmware a tovární nastavení.
- Paměti EEPROM, EEEPROM a RAM — čtení a zápis.
- LED diody — zapnutí a vypnutí LED, puls nebo blikání, zprávy zvlášť pro červenou a zelenou diodu transcieveru.
- Teploměr — čtení teploty ve stupních celsia.
- Vstup a výstup — nastavení portů na vstup nebo na výstup, přečtení hodnoty ze vstupního portu, zápis hodnoty na výstupní port.
- Rozhraní SPI — čtení a zápis.
- Rozhraní UART — nastavení přenosové rychlosti, čtení a zápis.
- Modul Fast Response Command — odeslání FRC požadavků jednoho typu za účelem sběru stejného typu informace z více uzlů v síti.

Kapitola 4

Návrh řešení

Cílem této kapitoly je čtenáři přiblížit současný stav testování a představit mu problém s testováním spojený, který tato práce má vyřešit. Součástí kapitoly je analýza požadavků, návrhu řešení komunikace testovaného softwaru s testovacím nástrojem, automatizace a integrace celého řešení do CI/CD prostředí.

4.1 Analýza zadání a požadavků na řešení

V současnosti pro software IQRF Gateway Daemon existují pouze základní testy pro některé komponenty. Jedním z cílů této práce a zároveň požadavkem na řešení je otestování aplikačního rozhraní a funkcí softwaru pro zpracování zpráv JSON , zpráv DPA, jejich konverzi a správnou interpretaci. V minulosti autoři softwaru o toto testování usilovali, ale jednalo se pouze o manuální provádění testovacích případů vůči skutečné síti se zařízením typu koordinátor a několika uzlovými zařízeními součástí této sítě. Toto řešení je nevhodné, protože se jedná o manuální provádění testování a také proto, že síť používaná pro testování nemusí být vždy dostupná třeba z neočekávaných technických důvodů. To nás přivádí k dalšímu cíli práce, a sice automatizace tohoto testování a následná integrace do již automatizovaného procesu sestavení softwaru, vytvoření balíčků a dodání koncovým uživatelům. Kromě toho by také bylo lepší, kdyby automatizované testování nebylo závislé na existenci skutečné sítě se všemi možnými druhy zařízení. Řešení by také mělo být snadno rozšířitelné v případě potřeby testování nových typů zařízení nebo periferií. Celý problém jsem tedy rozdělil na následující podproblémy:

- Který nástroj pro testování je nejvhodnější?
- Které prostředí pro automatizaci je nejvhodnější?
- Jak se zbavit závislosti na existenci skutečných zařízení a sítě?
- Pokud nebude použito skutečné zařízení, jak bude probíhat komunikace se softwarem?
- Jak celé řešení integrovat do automatizovaného procesu vývoje?

4.2 Zvolené nástroje

Jako testovací nástroj jsme zvolili Tavern, který je popsán v kapitole 2.6 a to z několika důvodů. Komunikace mezi cloudovými, webovými aplikacemi a Daemonem je možna pomocí

kanálů WebSocket, message queue a MQTT. Tavern podporuje MQTT komunikaci a nabízí se tak jako vhodný kandidát. Mimo jiné je to snadno použitelný a rozšiřitelný framework v Pythonu, který je jedním z nejpoužívanějších a nejvíce podporovaných jazyků v dnešní době. Jeho jednoduchost a použití v příkazové řádce umožňuje integraci do CI/CD pipeline. Tavern byl zvolen také proto, že po diskuzi s autory softwaru jsme zjistili, že už s Tavernem mají alespoň nějaké dřívější zkušenosti, a tak si myslím, že v něm viděli potenciál. Testovací případy budou definované ve formátu YAML. Tavern jednotlivé testy pak transformuje do formátu JSON, aby je Daemon mohl zpracovat.

Komunikace mezi Tavernem a Daemonem bude možná pouze pomocí protokolu MQTT a bude potřeba použít MQTT server, který bude přijímat a doručovat zprávy. Při diskuzi s autory softwaru jsme také zjistili, že při pokusu o testování byl použit MQTT server Eclipse Mosquitto[1] a tak se tedy nabízí jako vhodná volba.

Vzhledem k tomu, že samotný repozitář IQRF Gateway Daemon a IQRF Gateway Webapp jsou na GitLab, tak se to jeví jako jasná volba. GitLab má vlastní zabudovaný CI/CD systém a podporuje Docker, který bude potřeba pro běh Daemonu, Tavernu a MQTT serveru pro přijímání a přeposílání zpráv do Daemonu a zpět.

Nejlepším způsobem integrace řešení do CI/CD pipeline bude využití Dockeru. Celé řešení bude rozděleno do nasledujících čtyř kontejnerů Dockeru:

- kontejner Eclipse Mosquitto - tento kontejner bude v Dockeru sloužit jako server pro výměnu zpráv mezi Daemonem a Tavernem
- kontejner Emulátor - tento kontejner ponese instanci emulátoru sítě, ke kterému se Daemon připojí za účelem výměny DPA zpráv
- kontejner IQRF Gateway Daemon - tento kontejner ponese instanci předem sestaveného Daemonu, který se připojí ke kontejnerům MQTT serveru a emulátoru sítě
- kontejner Tavern - tento kontejner se spustí až jako poslední a bude sloužit k postupnému provádění testovacích případů a následnému vyhodnocení

Zde má smysl použít nástroj Docker Compose, jelikož je jedná o několik kontejnerů, které spolu při běhu mají komunikovat. Schéma komunikace v rámci Docker Compose je zobrazeno na obrázku 4.1. Docker Compose postupně spustí všechny kontejnery a proběhne testování. Po dokončení testování běh kontejneru Tavern skončí s návratovou hodnotou podle toho jestli testy dopadly úspěšně nebo selhaly. Po skončení běhu kontejneru Tavern budou ukončeny i ostatní kontejnery a Docker Compose skončí s návratovou hodnotu kontejneru Tavern. Pokud se bude jednat o nechybovou návratovou hodnotu, může pipeline pokračovat dalšími úkoly. V opačném případě se celá pipeline přeruší.

4.3 Komunikace Daemonu s emulovanou sítí

Jak již bylo zmíněno v kapitole 3.1, Daemon komunikuje s koordinátorem v síti pomocí rozhraní SPI, UART nebo USB. Protože bude skutečná síť nahrazena programem, který ji bude emulovat, tak využití těchto existujících rozhraní není možné. Jednotlivá rozhraní jsou pro Daemon implementována jako komunikační komponenty. Po diskuzi s autory softwaru jsem se tedy rozhodl rozšířit Daemon o komunikační komponentu využívající protokol TCP[2] (transmission control protocol), který zajišťuje doručení všech paketů a ve správném

[1] https://mosquitto.org/
[2] https://tools.ietf.org/html/rfc793

pořadí. Pro potřeby testování se po spuštění Daemonu aktivuje komponenta TCP, připojí se k emulované síti a toto spojení bude udržováno po dobu testování. Možnost komunikace s Daemonem pomocí protokolu TCP by potenciálně mohl využít jeden z partnerů firmy.

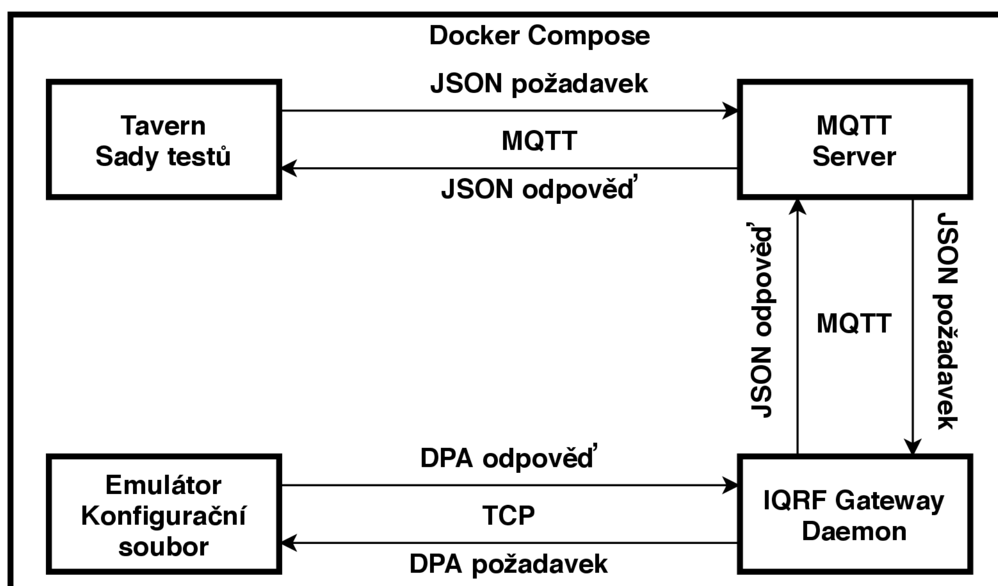
4.4 Emulace sítě

Místo využití skutečných zařízení a sítě jsme se rozhodli síť emulovat a testování provádět vůči této emulované síti. Nástroj Patriot se v tomto případě nabízí jako zajímavá možnost, ale pro naše potřeby je zbytečně složitý. Z toho důvodu jsme se rozhodli napsat program, který bude emulovat síť a zařízení v ní, a se kterým bude Daemon komunikovat při provádění testovacích případů. Hlavním přínosem emulátoru bude to, že se autoři softwaru zbavují nutnosti udržování skutečné sítě a využívání konkrétních zařízení. Emulátor bude také možno použít pro testování webové aplikace IQRF Gateway Webapp za účelem generování odpovědí. Poskytovaná funkcionalita bude na úrovni verze 4.13 protokolu DPA.

Architektura emulátoru

Na nejvyšší úrovni abstrakce lze emulátor rozdělit na 2 části. První z nich je server, který čeká na klienta. Klient se připojí a odešle požadavky na server. V tomto případě je klientem právě Daemon, který zasílá DPA požadavky a čeká na DPA odpovědi. Druhá část je správce sítě. Správce sítě má za úkol přijaté požadavky zpracovat, aplikovat požadované změny, případně shromáždit data ze sítě a sestavit odpověď, kterou předá zpět serveru pro odeslání Daemonovi. Vzhledem k tomu, že emulátor poběží primárně jako server v prostředí CI/CD pipeline a komunikace ním bude probíhat skrze jiné nástroje či aplikace, grafické rozhraní zde nemá moc smysl.

Jak již bylo zmíněno v kapitole 3.2, v síti se může vyskytovat až 240 zařízení a může se jednat o různá zařízení implementující různé periferie či standardy. Z toho důvodu jsme se rozhodli vytvořit konfigurační soubor ve formátu JSON, ve kterém se definuje emulovaná síť. Každé zařízení v této síti bude jedním objektem v tomto konfiguračním souboru a každé zařízení také bude mít definováno pole objektů představující periferie, které dané zařízení implementuje. Emulátor tedy po spuštění tento konfigurační soubor přečte a interně vystaví síť, se kterou poté bude správce sítě pracovat. Zařízení budou v interní síti emulátoru jako objekty třídy zařízení a jejich periferie analogicky. Pro každou periferii a standard bude vytvořena třída s vlastnostmi a metodami, které periferie implementuje dle dokumentace DPA [9].



Obrázek 4.1: Schéma komunikace mezi kontejnery v Docker Compose.

Kapitola 5

Implementace a testování

V této kapitole budou vysvětleny detaily implementace komunikační komponenty rozšiřující software IQRF Gateway Daemon, a emulátoru sítě. Dále se kapitola věnuje sestavování testovacích případů, způsobu testování a integraci do CI/CD pipeline. Nakonec jsou diskutovány výsledky testování.

5.1 Komunikační komponenta IqrfTcp

Jednotlivé komponenty softwaru IQRF Gateway Daemon jsou vybudovány na C++ frameworku shape[1]. Implementace je rozdělena do funkcí šablony komponent softwaru. Jedná se o funkce `activate` a `deactivate`, které jsou povinné pro každou komponentu. Dále jsou využity funkce `send` a `listen` pro odesílání a přijímání dat.

Komponenta IqrfTcp implementuje komunikaci pomocí protokolu TCP s Daemonem v roli klienta. Spojení je navázáno za pomocí BSD schránek, které slouží jako koncové body. Při implementaci byly využity datové struktury, funkce a konstanty definované v následujících hlavičkových souborech:

- `sys/socket.h`[2] – datové struktury BSD schránek a funkce pro jejich obsluhu
- `netinet/in.h`[3] – datové struktury nesoucí ip adresy, porty a další informace potřebné k navázání spojení
- `arpa/inet.h`[4] – funkce pro konverzi ip adres a portů mezi reprezentací s tečkami a reprezentací používanou při navázání spojení, doplňující struktury k `netinet/in.h`
- `netdb.h`[5] – funkce potřebné pro překlad doménového jména na ip adresy

Při aktivaci komponenty se z jí příslušného konfiguračního souboru načtou informace o zařízení, ke kterému se má připojit. Na základě těchto informací se získá jedna nebo více internetových adres, které lze použít pro vytvoření spojení. Pro každou adresu se komponenta pokusí vytvořit schránku pro odpovídající verzi ip adresy. Následuje pokus o navázání spojení. Pokud se nepodaří navázat spojení, vyzkouší se následující nabízená

[1] https://github.com/logimic/shape
[2] https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_socket.h.html
[3] https://pubs.opengroup.org/onlinepubs/009695399/basedefs/netinet/in.h.html
[4] https://pubs.opengroup.org/onlinepubs/009695399/basedefs/arpa/inet.h.html
[5] https://pubs.opengroup.org/onlinepubs/009695399/basedefs/netdb.h.html

adresa. Tento proces se opakuje dokud nedojde k úspěšnému navázání spojení, nebo dokud nedojdou použitelné adresy a komponenta se deaktivuje. Po navázání spojení může začít probíhat výměna zpráv.

5.2 Emulátor sítě

Emulátor sítě byl naprogramován mnou a Matějem Mrázikem. Na základě konzultací s vývojáři z firmy IQRF Tech s.r.o jsme se rozhodli, že emulátor bude implementován v programovacím jazyce Python verze 3. Během konzultací jsme se dohodli na způsobu implementace části, která bude pro oba společná. Do této části spadá konfigurace sítě a její vystavění v emulátoru, a obecné zpracování požadavků. Implementace jednotlivých zařízení, periferií a jejich následné testování pak bylo individuální. Implementace a testování provedené Matějem Mrázikem je možné nalézt v jeho práci [19]. Fáze běhu emulátoru jsou zobrazeny na obrázku 5.1 na konci této podkapitoly.

Konfigurační schéma a vystavění sítě

Konfigurační schéma je textový soubor ve formátu JSON, který nese objekty představující jednotlivá zařízení v síti. V síti musí být právě jedno zařízení typu koordinátor a dále v ní může být 0 – 239 zařízení typu uzel. Každý objekt má povinné vlastnosti, které je nutné v konfiguračním souboru specifikovat. Kromě vlastností má také každé zařízení seznam periferií, které implementuje.

Emulátor po spuštění otevře soubor předaný ve spouštěcích argumentech zpracovaných pomocí knihovny `argparse`[6] a pomocí knihovny `json`[7] přečte jeho obsah. V pozdější fázi implementace jsem se rozhodl ještě vytvořit validační schéma JSON Schema[8] pro konfigurační soubor. Validační schéma není předáváno jako argument při spuštění emulátoru, ale emulátor si jej sám načte. Cesta k validačnímu schématu je vyhodnocena pomocí knihovny `pathlib`[9]. Validační schéma obsahuje pravidla, která popisují jak mají jednotlivé objekty a položky souboru JSON vypadat. Data z konfiguračního souboru jsou pak vůči tomuto schématu validována pomocí knihovny `jsonschema`[10] a v případě porušení pravidel jsou nalezené chyby vypsány na standardní chybový výstup a běh programu skončí.

Validní data se následně předají funkci `buildNetwork`, která z načtených dat interně vystaví emulovanou síť a předá ji zpět hlavnímu modulu emulátoru. Pro zařízení i periferie byly napsány třídy, a v síti jsou existují jako objekty, tedy instance těchto tříd. Emulovaná síť je uložena v asociativním poli, kde jako klíč slouží síťová adresa zařízení. Každý objekt nese vlastní informace a také další asociativné pole, ve kterém jsou uložené periferie, které implementuje. Jako klíč zde slouží číslo periferie dle dokumentace DPA. Analogicky je řešena implementace standardů interoperability, s tím rozdílem, že pro ně je na zařízení vytvořené další asociativní pole.

[6] `https://docs.python.org/3.7/library/argparse.html`
[7] `https://docs.python.org/3.7/library/json.html`
[8] `https://json-schema.org/`
[9] `https://docs.python.org/3.7/library/pathlib.html`
[10] `https://github.com/Julian/jsonschema`

Komunikace s Daemonem

Emulátor má při komunikaci s Daemonem roli serveru. Po vystavění sítě je pomocí funkcí, maker a konstant z knihovny `socket`[11] vytvořen koncový bod na straně serveru, který akceptuje příchozí připojení. Tato knihovna poskytuje stejné funkce jako hlavičkové soubory pro práci se schránkami BSD zmíněnými v kapitole 5.1 a je tak kompatibilní s implementací komponenty na straně Daemonu. V případě výskytu chyby při vytváření či nastavování schránky se na standardní chybový výstup vypíše chybová hláška a běh programu skončí.

Po nastavení schránky začne emulátor naslouchat a čekat na připojení. Příchozí připojení od Daemonu je akceptováno a asynchronně je po malé časové prodlevě Daemonu odeslána konfigurace koordinátoru v síti. Na základě této konfigurace Daemon ví, která verze protokolu DPA je použita a jak má sestavovat požadavky a zpracovávat odpovědi. Poté emulátor pouze čeká na požadavky. Když dojde požadavek, je předán manažeru sítě aby jej obsloužil a sestavil odpověď. Odpověď je pak odeslána zpět Daemonu a celý proces se opakuje dokud je Daemon připojen.

Zpracování požadavků DPA

Přijatá odpověď je předána manažeru sítě funkcí `processRequest`. Zde probíhá zpracování příchozích požadavků DPA. Požadavek je přijat jako pole bajtů a je třeba jej tedy prvně konvertovat na datový typ, se kterým se pracuje lépe. Požadavek je nejprve předán funkci `parseRequest`, kde se překonvertuje z řetězce bajtů v hexadecimální soustavě na dekadickou reprezentaci a rozdělí se na položky dle kapitoly 3.3.

Překonvertovaná a rozdělená data požadavku jsou předána zpět a kontroluje se, zda se adresa zařízení, na které je požadavek zasílán, nachází v asociativním poli zařízení v síti. Pokud se v ní nenachází, je vytvořena odpověď s hodnotou značící špatnou adresu a odeslána zpět do Daemonu ke zpracování. Pokud se cílové zařízení v síti nachází, zkontroluje se, zda zařízení implementuje periferii, pro kterou požadavek je. Analogicky jako u zařízení se vyhledá číslo periferie v asociativním poli periferií. Pokud se v ní nenachází, je vytvořena a odeslána odpověď s odpovídající chybovou hodnotou.

Pokud zařízení periferii implementuje, data z požadavku jsou předána odpovídající periferii či standardu. Každá periferie či standard je implementována jako třída s atributy potřebnými pro emulaci její funkcionality. Kromě toho každá z těchto tříd implementuje metodu `handleRequest`, které jsou data požadavku předána. Třídy také implementují jednotlivé funkce popsané v dokumentaci protokolu DPA a případně pomocné funkce. Podle čísla příkazu z požadavku se zvolí odpovídající funkce k vykonání. V případě neplatného čísla příkazu pro danou periferii je vytvořena odpověď s odpovídající chybovou hodnotou. Funkce určená příkazem provede požadované změny v síti nebo prosté čtení dat a vrací zpět návratovou hodnotu a případná data k odeslání zpět Daemonu. Pokud je adresa cílového zařízení v síti hodnota `255`, jedná se o požadavek typu broadcast a požadavek zpracuje každé zařízení, které periferii v požadavku implementuje.

Data navrácená z periferie jsou společně se změněnou hodnotou profilového ID hardwaru a DPA hodnotou zařízení předána funkci `buildResponse`, kde jsou data konvertována zpět na řetězec bajtů v hexadecimální soustavě, aby mohla být schránkou na straně serveru odeslána zpět. Pro konverzi jsem použil knihovnu `struct`[12], která poskytuje funkce pro převod mezi hodnotami užívanými v Pythonu a strukturami v jazyce C. Sestavená odpověď

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

je následně předána zpět serverové části emulátoru a ten ji odešle zpět. Návratové hodnoty odpovědí a jejich význam je možné nalézt v dokumentaci protokolu DPA [9].

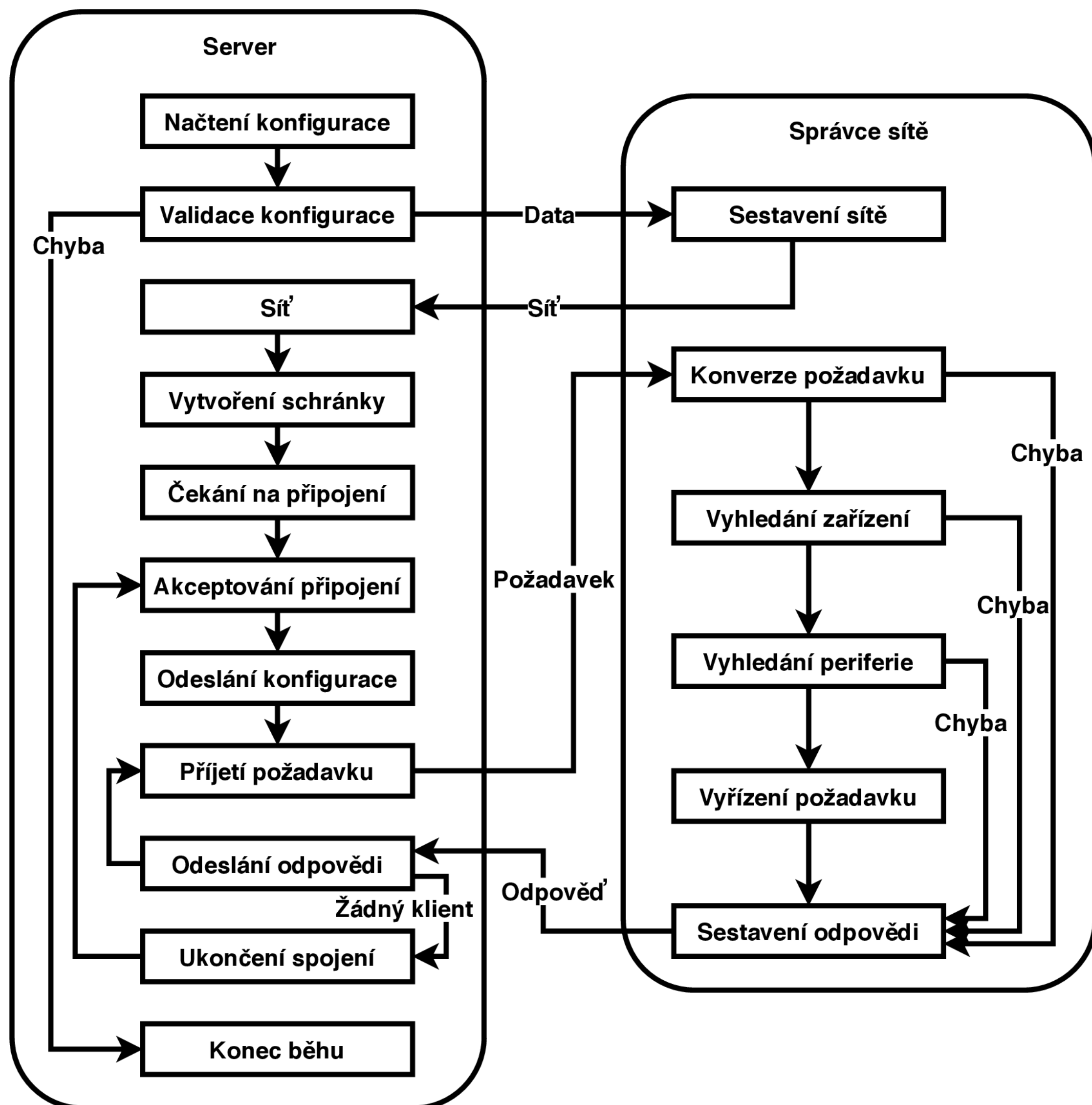
Implementace periferií

V rámci této práce jsem implementoval standardy a periferie binární výstup, senzor, paměť EEEPROM, IO, červená LED dioda, teploměr a rozhraní UART. Periferie FRC byla implementována jen částečně a sice funkce `Send`, `Extra result` a `Set FRC Params`. Funkce `Send` je navíc komplexní a představuje implementaci mnoha příkazů pro různé periferie. Dále jsem implementoval funkce pro průzkum zařízení (explore). Jelikož periferie koordinátor a operační systém jsou rozsáhle, a jsou důležité pro fungování sítě jako celku, po konzultaci s Matějem Mrázikem jsme každý implementoval část těchto periferií. V případě periferie koordinátor jsem implementoval funkce `Get discovered nodes`, `Get bonded nodes`, `Clear all bonds`, `Bond node`, `Remove bonded node` a `Discovery`. Funkce `Authorize bond` nebyla implementována, protože se jedná o funkcionalitu poskytovanou speciálním integrovaným vývojovým prostředím IQRF IDE[13]. Pro periferii OS jsem implementoval funkce `Read`, `Reset`, `Restart`, `Run RFPGM`, `Test RF Signal`, `Factory Settings`, `Indicate` a `Batch`. Funkce `LoadCode` nebyla implementována, protože se jedná o funkci, která do samotných zařízení nahrává jinou verzi DPA.

Jelikož se snažíme emulovat hardware na softwarové úrovni, není vždy možné funkcionalitu plně replikovat a pokud ano, bylo by to pravděpodobně velmi složité nebo nevhodné. Příkladem periferie, kde je možné funkcionalitu replikovat plně je LED dioda. LED diodu je možné rozsvítit, zhasnout, nechat blikat a nebo přimět bliknout pouze jednou. K tomu stačí jediná proměnná. Naopak standard binární výstup je na pro plnohodnotnou implementaci nepraktický. Příkladem je požadavek nastavení stavu až 32 binárních výstupů. Je možné výstupy vypnout či zapnout a to až na 127 minut. Tohoto by bylo možné docílit s využitím vláken nebo časových známek při přijetí požadavků od Daemonu. Pro potřeby testování je však nepraktické čekat až 127 minut jen proto, abychom mohli zkontrolovat zda byly výstupy uvedeny do stavu vypnuto po požadované době. V tomto případě se na straně emulátoru provede kontrola, zdali se jedná o platnou hodnotu určující dobu, ale samotný stav binárního výstupu je nastaven na vypnuto jakoby doba po kterou má být zapnut už uplynula.

Periferie IO poskytuje funkce nastavování pinů transcieverů a čtení dat z připojených zařízení. Implementace nastavování pinů je obecná, neboť pro každý typ transcieveru platí, že některé piny není možné pomocí těchto funkcí ovlivnit z důvodu využití pro specifický účel.

[13] `https://www.iqrf.org/technology/ide`



Obrázek 5.1: Schéma běhu emulátoru.

5.3 Testování

Jak již bylo zmíněno v kapitole 4.2, k testování byl použit nástroj Tavern. Jednotlivé testovací sady byly sestaveny ve formátu YAML, se kterým se oproti formátu JSON pracovalo lépe. Pro spuštění testování nástrojem Tavern bylo potřeba vytvořit Python skript, ve kterém se z knihovny `tavern`[14] importuje nástroj funkce `run`, pomocí které se postupně spouštějí jednotlivé sady testů specifikované v argumentech. Tavern požaduje, aby formát názvu souboru s testy byl `název_testu.tavern.yaml`, pokud tomu tak není, běh končí s chybovou hláškou. Samotná komunikace mezi Tavernem a Daemonem je zprostředkována knihovnou `paho-mqtt`[15], která je potřebná pro fungování Tavernu. Na začátku souboru s testovací sadou je nejprve nutné specifikovat název sady testů, typ spojení pro předání dat,

[14] `https://github.com/taverntesting/tavern`
[15] `https://pypi.org/project/paho-mqtt/`

identifikátor klienta a informace o MQTT serveru. Ukázka použité konfigurace testů je k dispozici ve výpise 5.1.

```
paho-mqtt:
  client:
    transport: tcp
    client_id: tavern-test-runner
  connect:
    host: mosquitto
    port: 1883
    timeout: 5
    keepalive: 30
```

Výpis 5.1: Ukázka konfigurace použité pro testování s nástrojem Tavern.

Sestavování testovacích sad

Jednotlivé testovací případy se v souboru sad definují v objektu `stages`, každý testovací případ má jméno, data s hodnotami testovacího případu k odeslání a očekávaný výsledek. V našem případě se jedná o požadavek a odpověď. Požadavky a odpovědi jsou přiložené k testům jako odkazy na soubory. Ukázka definice testů je k dispozici ve výpise 5.2. Přehled testovacích sad pro běh s emulátorem je k dispozici v tabulce B.1 přílohy B. Sady testů jsem navrhnul tak, aby se otestovaly všechny funkce testované periferie a zároveň aby jejich posloupnost byla pro testování smysluplná.

Jako příklad zde uvedu sadu testů pro periferii UART. Periferie UART poskytuje funkce pro otevření rozhraní s určitou rychlostí přenosu, uzavření rozhraní, zápis dat do přijímacího bufferu, čtení dat z vysílacího bufferu a zápis do přijímacího bufferu po jeho vyprázdnění. Kromě toho každá periferie implementuje funkci pro zjištění informací o periferii pro její identifikaci a správné zpracování odpovědí. Navržená testovací sada tedy vypadá takto:

1. Zjištění informací o periferii - platný požadavek.
2. Otevření rozhraní s nedefinovanou hodnotou rychlosti přenosu - neplatný požadavek.
3. Otevření rozhraní s definovanou hodnotou rychlosti přenosu - platný požadavek.
4. Pouze zápis do přijímacího bufferu - platný požadavek.
5. Čtení z vysílacího bufferu a zápis do přijímacího bufferu - platný požadavek, kontrola zda data z předchozího zápisu byla uložena a přepsání části dat bufferu.
6. Čtení z vysílacího bufferu a zápis do přijímacího bufferu po vyprázdnění - platný požadavek, kontrola zda byly skutečně předchozím požadavkem přepsány hodnoty bufferu.
7. Čtení z vysílacího bufferu - platný požadavek, kontrola zda skutečně došlo k vyprázdnění bufferu před přijetí dat.
8. Uzavření rozhraní - platný požadavek.
9. Čtení a zápis - neplatný požadavek, rozhraní není otevřeno.

10. Čtení a zápis po vyprázdnění přijímacího bufferu - neplatný požadavek, rozhraní není otevřeno.

```
stages:
  - name: uart_perinfo
    mqtt_publish:
      topic: Iqrf/DpaRequest
      json: !include uart/uart_perinfo_rq.yaml
    mqtt_response:
      topic: Iqrf/DpaResponse
      json: !include uart/uart_perinfo_rsp.yaml
      timeout: 5
```

Výpis 5.2: Ukázka definice testovacího případu pro nástroj Tavern.

Testování proti skutečné síti

Kromě testování s emulátorem sítě bylo také provedeno testování proti skutečné síti za účelem odhalení chyb implementace emulátoru, testovacích případů a další chyby softwaru, které při prvotním testování odhalené nebyly. Síť se skládala z 5 zařízení. Každé zařízení disponovalo periferiemi koordinátor nebo uzel, OS, RAM, EEPROM, EEEPROM, LED diody, IO, teploměr a FRC. Kromě toho na každém uzlovém zařízení byl implementován alespoň jeden ze standardů binární výstup, světla, senzor a DALI. Některé periferie nebylo možné na této síti otestovat, neboť je zařízení neimplementovala. Verze protokolu DPA na testované síti je 4.10, což je starší verze, než pro kterou byl implementován emulátor, ale liší se pouze přítomností nových funkcí a opravou menších chyb.

Testovací sady pro skutečnou síť nebudou součástí automatizovaného testování. Přehled těchto sad je k dispozici v tabulce B.2 přílohy B. Kromě této sítě jsem také provedl některé testy na alternativní síti s Romanem Ondráčkem z firmy IQRF Tech s.r.o. Jednalo se o případy, kdy byl potřeba lidský zásah k uskutečnění či dokončení testovacího případu, což v případě použití emulátoru nutné nebylo. Také zde byly otestovány funkce představené ve verzi protokolu DPA 4.13 a opravené chyby. Na alternativní síti bylo otestováno čtení hodnoty z teploměru poté, co byl vystaven teplotě pod 0°C. Dále bylo na alternativní síti otestováno vytváření a rušení vazeb mezi koordinátorem a uzly. Funkce `Bond node` pro vytvoření vazby vyžaduje potvrzení stisknutím tlačítka na zařízení. Funkce `Indicate` není součástí protokolu DPA 4.10, a tak byla také otestována na alternativní síti.

Výsledky testování

Pro testování s emulátorem sítě jsem vytvořil 10 testovacích sad, které dohromady obsahují 90 různých testovacích případů. Jedná se o sady testů pro periferie implementované v rámci této práce kromě funkcí průzkumu sítě a zařízení. Mezi těmito sadami byly i periferie koordinátor a operační systém, které jsem implementoval jen částečně a tak testovací sady obsahují testy pouze pro mnou implementovanou funkcionalitu.

Dále bylo vytvořeno dalších 9 testovacích sad s 66 testovacími případy pro ověření správnosti implementace periferií emulátoru a testovacích případů určených k automatizaci. Součástí testovacích sad nejsou případy vůči alternativní reálné síti zmíněné v předchozí části. Při sestavování testů a samotném testování jsem odhalil následující chyby:

- Špatné zpracování odečtených hodnot z periferie teploměr. Při přečtení záporné hodnoty nebyla prováděna korekce a Daemon interpretoval zápornou hodnotu $-20.5°C$ jako $234.5°C$ což je mimo měřitelné hodnoty. Chyba byla již opravena.
- Různé velikosti paměti použitelné koordinátorem v dokumentaci DPA a dokumentaci operačního systému IQRF. Chyba bude opravena v nové verzi dokumentace.
- Funkce `Set Output` periferie BinaryOutput (binární výstup) nese bitovou masku ovlivněných výstupů a poté jejich nové stavy, v případě rozdílného počtu bajtů stavů a nastavených bitů v masce je vrácena jiná chybová hodnota, než je uvedena v dokumentaci standardu. Jedná se o chybu standardu a bude opravena v další verzi.
- Dle dokumentace DPA se při přijetí DPA požavadku na čtení paměti kontroluje zda adresa není mimo platný rozsah. Tato kontrola se neprováděla. V nové verzi protokolu DPA se adresa kontrolovat bude.
- U periferie EEEPROM bylo možné zadat zápornou hodnotu adresy a požadavek byl Daemonem zpracován jako platný. Sice tato chyba byla objevena při testování periferie EEEPROM, ale jednalo se o chybějící kontrolu na zápornou hodnotu v případě vícebajtových dat. Konverze z dekadické do hexadecimální soustavy provádí jedna funkce pro více příkazů a periferií. Chyba byla opravena a zamezilo se tak průchodu záporných hodnot vícebajtových položek požadavků.

5.4 Zařazení do vývojového procesu

Obrazy Dockeru s emulátorem sítě a Tavernem s testovacími sadami jsou nahrávány na Docker hub. Tyto obrazy jsou poté staženy a jsou z nich v rámci CI/CD pipeline repozitáře softwaru IQRF Gateway Daemon vytvořeny kontejnery. Docker Compose je spuštěn jako samostatné stadium po proběhnutí jednotkových testů pro IQRF Gateway Daemon. Pokud Docker Compose skončí s nechybovým návratovým kódem, tedy testování skončí úspěšně, pokračuje CI/CD pipeline další fází.

Emulátor je v případě potřeby možné rozšířit o další vlastnosti zařízení, implementace nových periferií či nové funkcionality některých z periferií. Nové vlastnosti zařízení je potřeba přidat do konfiguračního souboru, vytvořit pro ně v třídě `device` pro zařízení nové proměnné a ve funkci `buildNetwork` souboru `networkhandler.py` se nová vlastnost přiřadí objektům třídy.

V případě implementace nové periferie je potřeba pro ni vytvořit novou třídu a importovat ji do `networkhandler.py`, definovat vlastnosti a obdobně jako u vlastností zařízení ve funkci `buildNetwork` přidat nový typ periferie k vytvoření objektů, které ji na zařízení reprezentují. Pro periferie lze pak implementovat nové funkce v rámci třídy, která ji reprezentuje a do funkce `handleRequest` přidat nové číslo příkazu, na základě kterého se funkce zavolá.

Je samozřejmě možné navrhnout a implementovat další testovací případy či sady. Ke každému testovacímu případu je potřeba napsat dva soubory, jeden z nich ponese data požadavku a druhý data očekávané odpovědi. Testovací případ se poté přidá do souboru testovací sady jak je ukázáno ve výpise 5.2. Pokud se jedná o zcela novou testovací sadu, je nutné tuto sadu přidat do souboru `run.py` ke spuštění.

Kapitola 6

Závěr

Cílem této práce bylo se seznámit s open source softwarem IQRF Gateway Daemon, jeho rozhraním a funkcemi. Dalším cílem bylo navrhnout testy pro tento software, implementovat testovací případy a integrovat je do prostředí vývoje softwaru za účelem automatizace testování. Výsledné řešení bylo integrováno do CI/CD pipeline repozitáře softwaru IQRF Gateway Daemon v podobě obrazů Dockeru, jejichž instance jsou spuštěny po nahrání nového kódu či změn v existujícím kódu do repozitáře.

Oproti původnímu zadání práce byl do řešení zařazen emulátor IoT sítě, aby bylo testování možné lépe automatizovat a eliminovat tak závislost na existenci skutečné sítě, vůči které by bylo testování prováděno. V rámci této práce pro emulátor byly implementované některé periferie. Emulovaná síť je vystavěna z konfiguračního souboru, který je možné kdykoliv změnit pro potřeby testování a změnit tak testovanou síť. Emulátor byl navrhžen pro podporu aktuální verze protokolu DPA 4.13 a implementace periferií zařízení je obecná dle dokumentace tohoto protokolu. Oproti původnímu zadání byl také software IQRF Gateway Daemon rozšířen o komunikační komponentu IqrfTcp, která se využívá pro komunikaci s emulátorem a mohla by být v budoucnu využívána jedním z partnerů firmy IQRF Tech s.r.o.

Pro zařízení a periferie, které software IQRF Gateway Daemon spravuje byly navrhženy a implementovány testovací sady, které pak byly použity pro otestování softwaru IQRF Gateway Daemon za použití emulátoru sítě. Dále bylo také provedeno testování vůči reálné síti za účelem ověření správnosti implementace emulátoru. Na základě testování bylo odhaleno 5 chyb v dokumentaci protokolu DPA a v samotném softwaru IQRF Gateway Daemon.

V budoucnu je možné v práci pokračovat například zakomponováním času. Emulátor oproti reálné síti nemusí řešit zpoždění vzniklé přenosem dat mezi jednotlivými zařízeními. Další možností je rozšířit emulátor o chování periferií, které se liší mezi jednotlivými modely zařízení.

Literatura

[1] AMMANN, P. a OFFUTT, J. *Introduction to Software Testing.* 1. vyd. New York: Cambridge University Press, 2008. 346 s. ISBN 978-0-521-88038-1.

[2] BOULTON, M. *Tavern Documentation: Release 0.34.0* [online]. Leden 2020 [cit. 3. dubna 2020]. Dostupné z: https://tavern.readthedocs.io/_/downloads/en/latest/pdf/.

[3] CI, T. *Travis CI User Documentation* [online]. 2020 [cit. 1. dubna 2020]. Dostupné z: https://docs.travis-ci.com/.

[4] CLARK, J. What is the Internet of Things, and how does it work? *Internet of Things blog* [online], 17. listopadu 2016. Aktualizováno 19. 9. 2017 [cit. 19. ledna 2020]. Dostupné z: https://www.ibm.com/blogs/internet-of-things/what-is-the-iot/.

[5] GITLAB. *Gitlab CI/CD | GitLab* [online], 26. srpna 2015. Aktualizováno 27. 3. 2020 [cit. 2. dubna 2020]. Dostupné z: https://docs.gitlab.com/ee/ci/.

[6] GURU99. Jenkins vs Travis-CI: What is the difference? *Guru99* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://www.guru99.com/jenkins-vs-travis.html.

[7] GURU99. What is QTP/UFT Automation Testing. *Guru99* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://www.guru99.com/uft-qtp-automation-testing.html.

[8] IQRF TECH S.R.O.. *DPA Framework Technical Guide Version v4.11* [online]. Prosinec 2019 [cit. 21. ledna 2020]. Dostupné z: https://static.iqrf.org/Tech_Guide_DPA-Framework-411_191211.pdf.

[9] IQRF TECH S.R.O.. *DPA Framework Technical Guide Version v4.13* [online]. únor 2020 [cit. 23. března 2020]. Dostupné z: https://static.iqrf.org/Tech_Guide_DPA-Framework-413_200227.pdf.

[10] IQRF TECH S.R.O. IQRF GW Daemon. *IQRF - Technology for wireless* [online]. 2020 [cit. 20. ledna 2020]. Dostupné z: https://www.iqrf.org/technology/iqrf-gw-daemon.

[11] IQRF TECH S.R.O. IQRF About. *IQRF - Technology for wireless* [online]. 2020 [cit. 19. ledna 2020]. Dostupné z: https://www.iqrf.org/iqrfabout.

[12] IQRF TECH S.R.O. Technology. *IQRF - Technology for wireless* [online]. 2020 [cit. 20. ledna 2020]. Dostupné z: https://www.iqrf.org/technology.

[13] IQRF Tech s.r.o. IQMESH. *IQRF - Technology for wireless* [online]. 2020 [cit. 20. ledna 2020]. Dostupné z: https://www.iqrf.org/technology/iqmesh.

[14] IQRF Tech s.r.o.. *IQRF Gateway User v2.2.0-rc documentation* [online]. 2020. Aktualizováno 25. 3. 2020 [cit. 31. března 2020]. Dostupné z: https://docs.iqrf.org/iqrf-gateway/index.html.

[15] Jain, A. What is UFT (QTP) ? (Micro Focus Unified Functional Testing). *LearnQTP: A Complete Portal on Micro Focus UFT (formerly QTP)* [online], 27. července 2012. Aktualizováno 22. 9. 2019 [cit. 3. dubna 2020]. Dostupné z: https://www.learnqtp.com/what-is-qtp/.

[16] Jenkins. *Jenkins User Documentation* [online], 29. února 2016. Aktualizováno 31. 3. 2020 [cit. 1. dubna 2020]. Dostupné z: https://jenkins.io/doc/.

[17] Křena, B. a Kočí, R. *Úvod do softwarového inženýrství IUS: Studijní opora.* Božetěchova 2, 612 00 Brno: FIT VUT v Brně, prosinec 2010.

[18] LINFO. Pipes: A brief introduciton. *The Linux Information Project* [online], 29. dubna 2004. Aktualizováno 23. 8. 2006 [cit. 1. dubna 2020]. Dostupné z: http://www.linfo.org/pipe.html.

[19] Mrázik, M. *Automatické testování software.* Brno, 2020. [cit. 24. května 2020]. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Pavol Korček, Ph.D.

[20] *Patriot - IoT Testing Framework* [online]. 2018 [cit. 3. dubna 2020]. Dostupné z: https://patriot-framework.io/.

[21] Patriot Framework Documentation. *Patriot - IoT Testing Framework* [online]. 2018 [cit. 3. dubna 2020]. Dostupné z: https://patriot-framework.io/latest/welcome/index.html.

[22] Patton, R. *Testování softwaru.* 1. vyd. Přeložil D. Krásenský. Praha: Computer Press, 2002. 314 s. ISBN 80-7226-636-5. Přeloženo z: Software Testing.

[23] Postman, Inc. *Postman | The Collaboration Platform for API Development* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://www.postman.com/.

[24] Postman, Inc. How to Use Postman for Application Development. *Postman | The Collaboration Platform for API Development* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://www.postman.com/use-cases/application-development.

[25] Postman, Inc. How to Use Postman for API Testing Automation. *Postman | The Collaboration Platform for API Development* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://www.postman.com/use-cases/api-testing-automation.

[26] Red Hat, Inc. What is CI/CD? *Red Hat* [online], 31. ledna 2018 [cit. 1. dubna 2020]. Dostupné z: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[27] SmartBear Software. Automated UI Testing Tools | TestComplete. *SmartBear* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://smartbear.com/product/testcomplete/overview/.

[28] SmartBear Software. TestComplete Use Cases | SmartBear. *SmartBear* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://smartbear.com/product/testcomplete/use-cases/.

[29] SmartBear Software. Automated Mobile Testing Tool | TestComplete. *SmartBear* [online]. 2020. [cit. 3. dubna 2020]. Dostupné z: https://smartbear.com/product/testcomplete/mobile-testing/.

[30] Gray Box Testing. *Software Testing Fundamentals* [online], 19. prosince 2010. Aktualizováno 3. 3. 2018 [cit. 22. ledna 2020]. Dostupné z: http://softwaretestingfundamentals.com/gray-box-testing/.

[31] *The Selenium Browser Automation Project :: Documentation for Selenium* [online]. 2020 [cit. 3. dubna 2020]. Dostupné z: https://www.selenium.dev/documentation/en/.

[32] Yegulalp, S. What is Docker? The spark for the container revolution. *InfoWorld* [online], 19. dubna 2019 [cit. 19. května 2020]. Dostupné z: https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html.

Příloha A

Obsah přiloženého CD

Struktura adresářů na přiloženém CD:

- /doc – adresář s textem práce a zdrojovými soubory textu práce.
- /IqrfTcp – adresář se zdrojovými kódy komunikační komponenty IqrfTcp.
- /testing – adresář se zdrojovými kódy emulátoru IoT sítě, testovacími sadami a konfiguračními soubory pro emulátor, sestavení obrazů Dockeru a spuštění kontejnerů.
- readme.md – návod pro instalaci a spuštění řešení včetně ukázek.
- xhanak34-bp.zip – archiv obsahující výše zmíněné soubory.

Příloha B

Tabulky testovacích sad

Tabulka B.1: Tabulka testovacích sad jednotlivých periferií integrovaných do vývojového prostředí softwaru IQRF Gateway Daemon.

Periferie	**Funkce**	**Počet testovacích případů**
Coordinator	Peripheral Information	1
	Get discovered nodes	4
	Get bonded nodes	2
	Clear all bonds	1
	Bond node	4
	Remove bonded node	2
	Discovery	2
OS	Peripheral Information	1
	Read	2
	Reset	1
	Restart	1
	Run RFPGM	1
	Batch	1
	Test RF Signal	2
	Factory Settings	2 + 1 (po factory settings)
	Indicate	2
EEEPROM	Peripheral Information	1
	Extended Read	3
	Extended Write	3
FRC	Peripheral Information	1
	Send	2
	Set FRC Param	3
IO	Peripheral Information	1
	Direction	3
	Set	2
	Get	1
LEDR	Peripheral Information	1
	Set	2
	Pulse	1
	Flash	1

Thermometer	Peripheral Information	1
	Read	3
UART	Peripheral Information	1
	Open	2
	Close	1
	Write & Read	4
	Clear & Write & Read	2
Standard BinaryOutput	Set Output	4
	Enumerate	1
Standard Sensor	Read Sensors	–
	Read Sensors with Types	13
	Enumerate Sensors	2

Tabulka B.2: Tabulka testovacích sad jednotlivých periferií použitých při testování proti skutečné síti.

Periferie	**Funkce**	**Počet testovacích případů**
Coordinator	Peripheral Information	2
	Get discovered nodes	2
	Get bonded nodes	2
	Clear all bonds	1
	Remove bonded node	1
	Discovery	2
	SmartConnect (oprava sítě)	4
OS	Peripheral Information	1
	Read	2
	Reset	1
	Restart	1
	Batch	1
	Test RF Signal	2
	Factory Settings	2 + 2 (po factory settings)
	SmartConnect (oprava sítě)	1
EEEPROM	Peripheral Information	1
	Extended Read	3
	Extended Write	3
FRC	Peripheral Information	1
	Send	2
	Set FRC Param	3
IO	Peripheral Information	1
	Direction	3
	Set	2
	Get	1
LEDR	Peripheral Information	1
	Set	2
	Pulse	1
	Flash	1
Thermometer	Peripheral Information	1
	Read	2
Standard BinaryOutput	Set Output	6
	Enumerate	1
Standard Sensor	Read Sensors	–
	Read Sensors with Types	3
	Enumerate Sensors	2

Příloha C

Ukázka výstupu z CI/CD pipeline

```
tavern_test | ============================ test session starts
    ==============================
tavern_test | platform linux -- Python 3.8.3, pytest-5.4.2, py-1.8.1,
    pluggy-0.13.1
tavern_test | rootdir: /app
tavern_test | plugins: pspec-0.0.3, tavern-1.2.2
tavern_test | collected 1 item
tavern_test |
mosquitto-api-test | 1591105106: New connection from 172.18.0.5 on port
    1883.
mosquitto-api-test | 1591105106: New client connected from 172.18.0.5 as
    tavern-test-runner (c1, k30).
network_emulator | Request received.
network_emulator | ['0x1', '0x0', '0xa', '0x3f', '0xff', '0xff']
network_emulator | Response sent.
network_emulator | ['0x1', '0x0', '0xa', '0xbf', '0x0', '0x0', '0x0', '0x4b
    ', '0x1', '0xb', '0x0', '0x0']
network_emulator | -----------------------------------------
network_emulator | Request received.
network_emulator | ['0x1', '0x0', '0xa', '0x0', '0xff', '0xff']
network_emulator | Response sent.
network_emulator | ['0x1', '0x0', '0xa', '0x80', '0x0', '0x0', '0x0', '0x4b
    ', '0x1e', '0xeb', '0x1']
network_emulator | -----------------------------------------
network_emulator | Request received.
network_emulator | ['0x2', '0x0', '0xa', '0x0', '0xff', '0xff']
network_emulator | Response sent.
network_emulator | ['0x2', '0x0', '0xa', '0x80', '0x0', '0x0', '0x0', '0x4b
    ', '0xeb', '0xb8', '0xe']
network_emulator | -----------------------------------------
network_emulator | Request received.
network_emulator | ['0x3', '0x0', '0xa', '0x0', '0xff', '0xff']
network_emulator | Response sent.
```

```
network_emulator | ['0x3', '0x0', '0xa', '0x80', '0x0', '0x0', '0x0', '0x3f
    ', '0x80', '0x0', '0x0']
network_emulator | ----------------------------------------
mosquitto-api-test | 1591105108: Client tavern-test-runner disconnected.
tavern_test | testcases/test_thermometer.tavern.yaml
tavern_test | thermometer.tavern.yaml
tavern_test | embed thermometer peripheral:
tavern_test | 1: thermometer perinfo
tavern_test | 2: thermometer read positive
tavern_test | 3: thermometer read negative
tavern_test | 4: thermometer read not installed
tavern_test | [100%]
tavern_test |
tavern_test | ======================== 1 passed, 3 warnings in 1.84s
    =========================
```

Výpis C.1: Ukázka výstupu z probíhajícího testování v CI/CD pipeline.