VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

ROZŠÍŘENÍ FRAMEWORKU PATRIOT O MODELOVÁNÍ NÁHODNÝCH UDÁLOSTÍ

THESIS TITLE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MAREK ŠŤASTNÝ**
AUTHOR

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

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

157343

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Šťastný Marek, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Vestavěné systémy
Název:	**Rozšíření frameworku PatrIoT o modelování náhodných událostí**
Kategorie:	Modelování a simulace
Akademický rok:	2023/24

Zadání:

1. Seznamte se s frameworkem PatrIoT, kdy se zaměřte především na modul patriot-data-generator. Popište architekturu tohoto modulu.
2. Na základě poznatků z bodu 1 identifikujte možnosti rozšíření modulu patriot-data-generator o rozhraní pro přidání modelů umožňujících simulaci náhodných událostí.
3. Zabývejte se problematikou simulování rozsáhlých událostí (např. fyzikální simulace šíření požárů). Najděte vhodný a dostatečně obecný teoretický model pro libovolné geometrické rozmístění IoT zařízení a simulaci dějů v tomto prostoru, se kterými daná zařízení interagují.
4. Navrhněte úpravu rozhraní stávajícího modulu patriot-data-generator tak, aby umožňoval synchronizaci stavu generátorů na základě informace o jejich umístění v prostoru a čase.
5. Navrhněte manažer zodpovědný za synchronizaci a propagování rozsáhlé simulované události napříč vícero simulovanými zařízeními (šíření požáru, změna počasí na základě přesunu teplé fronty apod.).
6. Implementuje synchronizační manažer a rozhraní do modulu patriot-data-generator.
7. Funkčnost řešení demonstrujte rozšířením projektu virtual-smart-home-plus o vhodné simulace.
8. Zhodnoťte dosažené výsledky a navrhněte další možné vylepšení, případně další směry vývoje.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:
Splnění bodů 1 až 4 zadání.

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

Vedoucí práce:	**Šimek Václav, Ing.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2023
Termín pro odevzdání:	31.7.2024
Datum schválení:	30.10.2023

Abstrakt

Práce se zabývá rozšířením frameworku pro integrační a end-to-end testování IoT systémů. Specificky se zaměřuje na vytvoření simulace virtuálního prostředí, které ovlivňuje data generovaná senzory a aktuátory, jež tvoří testovací prostředí. Práce se věnuje formalismům pro popis prostorů a metodám pro modelování širokého spektra dějů. Přichází s modulárním návrhem obsahujícím třídy pro reprezentaci prostoru, času a dílčích simulací. Jádrem práce je simulátor založený na modelu komunikace publish-subscribe, který propojuje dílčí simulace a umožňuje synchronizaci simulačního času s reálným.

Abstract

This thesis deals with the extension of a framework for integration and end-to-end testing of IoT systems. Specifically, it focuses on creating a simulation of a virtual environment that influences data generated by sensors and actuators, which constitute the testing environment. The thesis addresses formalisms for describing spaces and methods for modeling a wide range of phenomena. It presents a modular design containing classes for the representation of space, time, and partial simulations. The core of the work is a simulator based on a publish-subscribe communication model that connects the partial simulations and also allows for the synchronization of simulation time with real time.

Klíčová slova

framework PatrIoT, IoT, integrační testování, end-to-end testování, sběrnice událostí, simulace, Java, souřadnicový systém, hybridní simulace

Keywords

framework PatrIoT, IoT, integration testing, end-to-end testing, event bus, simulation, Java, coordinate system, hybrid simulation

Citace

ŠŤASTNÝ, Marek. *Rozšíření frameworku PatrIoT o modelování náhodných událostí.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Václav Šimek,

Rozšíření frameworku PatrIoT o modelování náhodných událostí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Václava Šimka. Další informace mi poskytl Bc. Mirek Jaroš. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Marek Šťastný
31. července 2024

Poděkování

Tímto bych rád poděkoval vedoucímu práce panu Ing. Václavu Šimkovi za konzultace při realizaci této diplomové práce. Dále velké poděkování patří technickému konzultantovi Bc. Miroslavovi Jarošovi za odborné vedení, nadstandardní pomoc a motivaci. Díky patří i přítelkyni MDDr. Ivě Kašparové, rodině a přátelům Mgr. Štěpánu Hudečkovi, Ing. Michaelu Kincovi, Bc. Vojtěchu Krejčíkovi, Ing. Arthuru Nácarovi, Ing. Vojtěchu Mimochodkovi a Ing. Petru Vrtalovi, kteří mi poskytovali obrovskou podporu.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Testování je nezbytná část procesu vývoje softwaru. Pomáhá nám u projektu řídit riziko, kontrolovat kvalitu, zvyšuje bezpečnost produktu a zlepšuje uživatelskou zkušenost. V průběhu let vznikla spousta nástrojů, které napomáhají s vytvářením testovacího prostředí, implementací testů a s jejich spouštěním a vyhodnocováním. S rozvojem internetu věcí (IoT) vznikly i specifické požadavky na testování těchto systémů. PatrIoT framework je nástroj, který nabízí řešení pro některé typy testování IoT systémů. V současné době, jeden z modulů toho frameworku, je knihovna pro vytváření virtuálních IoT zařízení. Tato zařízení ovšem umí generovat pouze lokální data založená na vnitřním stavu modelovaného zařízení. V současném stavu tedy chybí podpora po generování dat, které by dokázaly reflektovat externí událost ovlivňující více zařízení. Tato práce se zabývá rozšířením frameworku PatrIoT tak, aby podporoval simulace událostí, které ovlivňují data generovaná chytrými zařízeními.

Struktura textu

Tato práce je členěna do devíti kapitol, které popisují průběh práce na tomto projektu od seznámení se s projektem PatrIoT přes analýzu problému, návrh řešení, implementaci, až po demonstraci výsledného řešení.

Druhá kapitola podrobněji stanovuje cíle práce. Nejprve se snaží uvést čtenáře do aktuálního stavu frameworku a seznámit jej s jeho nedostatky. Dále jsou popsány implementační cíle této práce, které vychází z nedostatků současného stavu.

Ve třetí kapitole tato práce popisuje účel frameworku PatrIoT a na jakých principech stojí jeho návrh. Dále zevrubně popisuje jeho strukturu a následně se detailněji zaměřuje na modul *PatrIoT Data Generator*. Tento modul je předmětem úprav v rámci této práce. Konečně jsou ve třetí kapitole popsány hlavní komponenty tohoto modulu, jejich funkce a jak spolu interagují.

Ve čtvrté kapitole jsou popsány metody pro modelování různých systémů, které simulují jevy ovlivňující data generovaná IoT zařízeními. Obzvláště je kladen důraz na matematické modelování, celulární automaty a agentní systémy. Na tyto modely se bude práce odkazovat v kapitole o návrhu řešení 5. Dále je podroben diskusi vztah mezi těmito systémy, jejich členění a možnosti interoperability. Na konci této kapitoly jsou představeny dostupné nástroje, které zajišťují interoperabilitu různých simulačních modelů a umožňující rozsáhlé simulace.

Pátá kapitola pojednává o návrhu rozšíření. Nejprve jsou formálně stanoveny požadavky na daný systém. Poté se definují základní rozhraní. Nakonec jsou navrhnuty konkrétní třídy, které implementují základní rozhraní a poskytují řešení pro specifické způsoby použití.

Šestá kapitola čerpá z kapitoly 4 při výběru vhodných knihoven a nástrojů pro implementaci. Následně popisuje kroky potřebné proNásledně popisuje implementaci základních tříd navržených v kapitole 5.

V sedmé kapitole je čtenář seznámen s výsledným řešením. Jsou zde shrnuty základní kroky, které je potřeba provést pro úspěšné spuštění knihovny a následné využití její funkcionality. Dále je detailněji popsán vznik dvou demonstračních projektů využívajících modulu popsaného v této práci. Tato kapitola může sloužit jako ukázka pro případné uživatele nebo jako zdroj informací, jak daný modul využívat.

Osmá kapitola pojednává o dosažených výsledcích, dalších potřebných rozšířeních a nastiňuje možnosti, jakými dalšími směry by se modul `PartIoT Data Generator` mohl vyvíjet.

Kapitola 2

Cíle práce

Hlavním cílem této práce je rozšíření frameworku PatrIoT. Konkrétně se jedná o rozšíření modulu `patriot-data-generator`[1], který je zodpovědný za vytváření virtuálních IoT zařízení, které generují data pro testování IoT systému.

V původní verzi jsou zařízení schopna generovat data pouze na základě jednoduchých stochastických funkcí. Tento způsob neumožňuje vytvářet vazby mezi daty generovanými z jednotlivých zařízení. Nelze tak například vytvořit sadu vlhkoměrů, které by generovaly přibližně stejnou vlhkost, která by byla definovaná jedním zdrojem.

Rozšíření by mělo umožnit nadefinovat prostor s odpovídajícími souřadnicemi, ve kterém se zařízení nachází. Definice souřadnicového systému by měla být dostatečně obecná, aby uživateli dala svobodu použít vhodné souřadnicové systémy pro specifické potřeby. Framework PatrIoT by měl minimálně umožnit použití kartézského souřadnicového systému, či použití grafu.

Rozšíření by mělo do modulu `patriot-data-generator` také zavést pojem času. Jelikož framework PatrIoT umožňuje vytváření hybridního testovacího prostředí[2], je potřeba, aby bylo možné zavedený virtuální čas synchronizovat s reálným.

Framework PatrIoT by po dokončení této práce měl umožnit v simulovaném prostoru nadefinovat události, která se mohou (náhodně) stát za uživatelem definovaných podmínek. Tyto vzniklé události následně mohou ovlivnit data generovaná zařízeními, které spadají do prostoru vzniklé události a daná událost na ně má vliv.

Po implementaci výše zmíněných požadavků je potřeba dosažené výsledky prezentovat funkční demonstrací. Tato demonstrace bude sloužit jako ukázka schopností modulu a zároveň jako návod pro uživatele k zavedení náhodných událostí.

Modul `virtual-smart-home-plus`[3] slouží k demonstraci použití virtuálních zařízení. Vytváří REST rozhraní domu, na které je možné posílat požadavky a zjišťovat informace ohledně chytrých zařízení v domě. Součástí této práce by tedy mělo být využití tohoto modulu pro simulace o vhodně zvolených událostí, které budou ovlivňovat data generovaná zařízeními v tomto domě.

[1] https://github.com/PatrIoT-Framework

[2] Testovací prostředí je tvořeno reálnými zařízeními a virtuálními prvky.

[3] https://github.com/PatrIoT-Framework/virtual-smart-home-plus

Příkladem takovéto události může být vznik požáru, který vypukne v 11 hodin v kuchyni virtuálního domu. Tato událost ovlivní teplotní senzory pouze v kuchyni a to tak, že postupně začínají hlásit vyšší teploty, až nakonec přestanou hlásit cokoliv (poškození zařízení).

Na konci práce je potřeba zhodnotit do jaké míry byly splněny cíle této práce. Dále je třeba popsat úskalí či přednosti implementovaného řešení a konečně je třeba popsat, jakými dalšími směry by se mohl ubírat vývoj rozšiřovaného modulu `patriot-data-generator` a jaká jsou další možná vylepšení.

Kapitola 3

Úvod k frameworku PatrIoT

V této kapitole bude popsán framework Patriot[1] [2], jeho účel, základní paradigmata a moduly, ze kterých je sestaven. Dále bude zevrubně popsán účel každého modulu, jeho implementace a interakce se zbytkem frameworku. Konečně budou detailněji popsány komponenty *test runner*, *testbed hub*, *network simulation manager* a *device emulator*, které úzce souvisí s touto prací. Informace popsané v této kapitole jsou založeny na článcích [6], [7], na zdrojových kódech projektu a na oficiální dokumentaci [3].

3.1 Účel frameworku PatrIoT

S tím, jak se stále častěji vyvíjejí nové systémy, které spadají do kategorie IoT[3], vyvstává potřeba testování těchto systémů. Framework PatrIoT si klade za cíl poskytovat podporu pro integrační testování, end-to-end testování a testování interoperability těchto systémů. Pro sestavení testovacího prostředí navíc nabízí podporou hybridního přístupu, který kombinuje výhody testování s využitím simulací a výhody testování na nasazených systémech, nebo prototypech. Při návrhu tohoto frameworku bylo dbáno na tyto návrhové principy:

- vývoj pod open-source licencí Apache 2.0,
- využívání dobře etablovaných open-source nástrojů pro jednoduchost použití a integraci s dalšími testovacími infrastrukturami,
- škálovatelnost pro potřeby různých testování (testování API[4], rozsáhlé E2E[5] testy),
- umožnění integrace - jak fyzických, tak simulovaných komponent,
- modulární přístup pro simulaci zařízení, kde je každé zařízení konfigurovatelný modul, a zároveň poskytnutí předdefinovaných šablon zařízení,
- podpora připojení datových vstupů z nástrojů Model Based Testing (MBT) prostřednictvím otevřených rozhraní,

[1] Oficiální stránka projektu: https://patriot-framework.io/

[2] Zdrojové kódy jsou dostupné na stránce: https://github.com/PatrIoT-Framework

[3] IoT – Internet of Things (internet věcí – systém internetem propojených inteligentních zařízení, která si mohou vyměňovat data a stát se tak aktivními účastníky procesu)

[4] API – Aplication Proramming Interface (aplikační programové rozhraní)

[5] End-To-End testing (technika testování softwaru, kdy se testuje celý systém pomocí simulování využití softwaru scénaři z reálného provozu.)

- poskytnutí podpory k vytvoření dobře strukturovaných automatizovaných testů interoperability a integrace,
- modulární architektura – sestavení projektu z opakovaně použitelných modulů propojených přes otevřená rozhraní.

3.2 Architektura frameworku PatrIoT

Projekt Patriot byl při návrhu rozdělen do komponent, kdy každá komponenta zodpovídá za část funkcionality frameworku. Pro testování IoT systému pak uživatel typicky použije většinu těchto komponent pro vytvoření testovacího prostředí a spuštění testů. Původně navržené komponenty, zmíněné v článku PatrIoT: IoT Automated Interoperability and Integration Testing Framework [7], byly tyto:

- test runner
- testbed hub
- network simution manager
- device emulator
- collector
- provisioner
- reporter

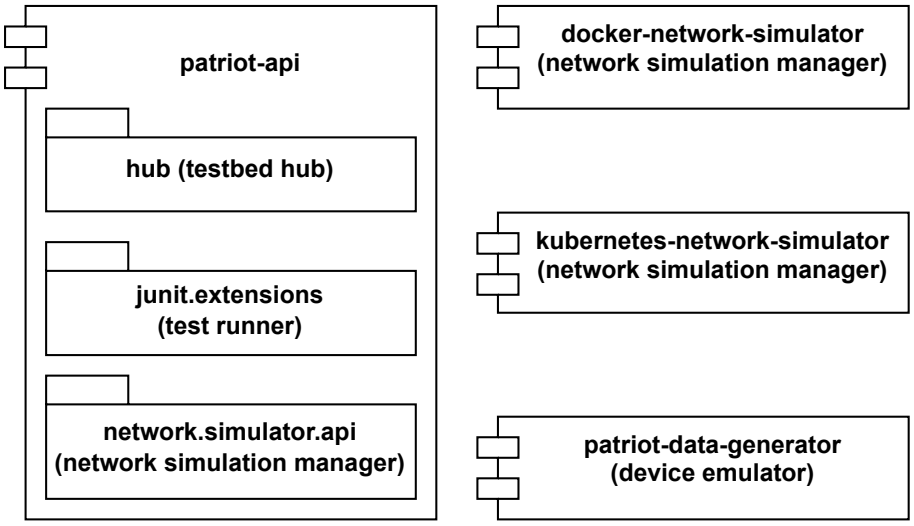
Toto navržené členění ovšem přesně neodpovídá implementovaným repozitářům a java balíčkům frameworku v aktuálním stavu. Aktuálně je PatrIoT implementován pomocí modulů `patriot-api`, `patriot-data-generator`, `docker-network-simulator` a `kubernetes-network-simulator`. Aktuální moduly a vztah k původním komponentám je vyobrazen na obrázku 3.1.

3.3 Patriot-api

Tento modul vytváří hlavní rozhraní, které využívá další moduly frameworku a obsahuje následující tři klíčové balíčky: `junit.extensions`, `hub` a `network.simulator.api`.

3.3.1 Balíček junit.extensions

Tento balíček rozšiřuje testovací framework JUnit, který je navržen pro jednotkové testování. Pro použití JUnit ve frameworku PatrIoT, který je určen pro integrační testování, bylo potřeba JUnit rozšířit o nastavení testovacího prostředí před spuštěním testů. Dále bylo potřeba pro integrační testování umožnit podmíněné spuštění některých testů v závislosti na úspěšném ukončení testů jiných (integrační testy na sobě mohou mít závislosti). Nastavení testovacího prostředí se provádí pomocí rozšíření abstraktní třídy SetupExtension a podmíněné spuštění testu je zajištěno třídou ConditionalDisableExtension.



Obrázek 3.1: Přehled klíčových modulů frameworku PatrIoT. U modulu patriot-api jsou zobrazeny tři hlavní balíčky, které obsahuje. V závorkách jsou uvedeny názvy komponent dle původního návrhu [7], které daný modul nebo balíček implementuje.

3.3.2 Balíček network.simulator.api

Základní balíček, který poskytuje třídy pro interakci frameworku s uživatelem. Patrně nejdůležitější třída je TopologyBuilder, která umožňuje vytvořit model virtuální sítě. Vytvoření modelu sítě je dosaženo posloupností čtyř základních kroků, které jsou:

1. Přidání routerů – zde uživatel definuje identifikační jméno.
2. Přidání podsítí – u sítě je třeba natavit jméno, IP adresu a masku.
3. Definování cest mezi podsítěmi – zde je potřeba definovat podsítě, ke kterým patří daná cesta, přes jaký router daná cesta vede a za jakou cenu.
4. Nasazení vytvořeného modelu sítě pomocí network-simulatoru – zde je potřeba zvolit, zdali má být síť simulovaná pomocí modulu docker-network-simulator, nebo kubernetes-network-simulator. Podrobněji je tato volba popsána v sekci 3.4.

3.3.3 Balíček hub

Tento balíček poskytuje třídu `PatriotHub` zodpovědnou za stav a řízení testovacího prostředí. Tato třída uchovává informace o virtuální síti, nasazených aplikacích v rámci virtuální sítě a všech virtuálních zařízeních (blíže popsaných v sekci 3.5). Skrze tento objekt probíhá pomocí metody `deployTopology` vytvoření simulované sítě zmíněné v podsekci 3.3.2. Díky metodám `getApplication` a `getManager` lze získat singleton objekty Application a Manager, které jsou zodpovědné za správu aplikací a virtuální sítě.

3.4 Network simulator

V podsekci 3.3.2 byly popsány kroky potřebné pro vytvoření modelu virtuální sítě. V podsekci 3.3.3 byla popsána třída `PatriotHub`, která pomocí metody `deployTopology` vytvoří na základě modelu simulaci této sítě. Pro vytvoření simulace je ovšem potřeba simulátor, ve kterém daná simulace probíhá. Framework PatrIoT poskytuje hned dva takovéto simulátory, a to `docker-network-simulator` a `kubernetes-network-simulator`.

Funkcionalita výše zmíněných simulátorů je stejná, ale liší se technologiemi, na kterých jsou postaveny. Jak název napovídá `docker-network-simulator` využívá nástroj *Docker*[6] a `kubernetes-network-simulator` využívá *Kubernetes*[7].

3.5 Data Generator

Modul `Komponenta data-generator` je implementací původně navržené komponenty device emulator zmíněné v sekci 3.2. Data generator byl navržen jako knihovna pro podporu simulování IoT zařízení (aktuátorů a senzorů). Za tímto účelem vytváří čtyři hlavní komponenty, které jsou rozděleny definovanými rozhraními. Informace v této podkapitole jsou čerpány z diplomové práce pojednávající o tvorbě tohoto modulu [26] a z vlastních pozorování zdrojového kódu.

Pro vytvoření modelu IoT zařízení vytváří komponentu Device. Ta uživateli poskytuje možnost použít předdefinované modely (DHT11, teploměr a další), nebo rozšířit základní třídy (AbstractDevice, AbstractActuator, AbstractSensor) o vlastní implementaci, a namodelovat si tak libovolné potřebné zařízení.

Další komponenta je DataFeed. Instance tohoto rozhraní předané modelovaným zařízením jsou zodpovědné za generování dat těmto zařízením. Stejně jako u komponenty Device, i zde si uživatel může vybrat předdefinovaný generátor, nebo si může implementovat vlastní. Data jsou u předdefinovaných generátorů generována na základě stochastických funkcí.

Komponenta NetworkAdapter vytváří pro zařízení síťové rozhraní. Po jeho přidání k zařízení jsou vygenerovaná data zasílána po síti. V současném stavu tato komponenta podporuje protokoly HTTP, MQTT a CoAP.

Na vytvořené virtualizované zařízení v PatrIoT framworku je kladen požadavek, aby bylo každé zařízení schopné být konfigurovatelné po síti. Tohoto je docíleno pomocí kompenenty Controller, která využívá protokol CoAP.

3.5.1 Komponenta Device

Tato komponenta vytváří dvě základní rozhraní. Unit a Device. Unit definuje základní metody pro identifikaci zařízení. Device rozšiřuje rozhraní Unit o metody, které umožňují nastavení síťové konfigurace (viz. 3.5.4), zasílání dat po síti (viz. 3.5.3) a nastavení generátoru dat (viz. 3.5.2).

Chytrá IoT zařízení dělíme na aktuátory a senzory. Senzory umožňují pomocí měření získávat informace o svém okolí. Aktuátory jsou pak schopny své okolí ovlivňovat způsobem, ke kterému byly navrženy. Oba typy zařízení interpretují data o svém stavu a okolním

[6] Docker je nástroj pro nasazování a správu kontejnerů, které vytváří virtuální prostředí pro aplikace, což umožňuje konzistentní nasazení napříč různými prostředími. Pomocí Dockeru lze také vytvořit virtuální síť, ve které jsou následně kontejnery spuštěny.

[7] Kubernetes je platforma pro automatizaci nasazení, škálování a správu kontejnerizovaných aplikací. Umožňuje efektivně spravovat clustery kontejnerů, zajišťuje vysokou dostupnost a také umožňuje nasazení kontejnerů v rámci virtuální sítě.

prostředí, a skrze komunikační rozhraní je dále distribuují dalším prvkům v síti. Toto dělení je zohledněno i v návrhu předdefinovaných zařízeních určených pro uživatele.

Pro rozhraní Device je implementována třída AbstractDevice, ze které jsou odvozeny třídy AbstractSensor a AbstractActuator, které modelují základní třídy pro senzory a akuátory. Předdefinované zařízení pro použití uživatelem jsou pak rozšířením těchto tříd. V současné situaci jsou pro uživatele implementovány tyto senzory:

- Default,
- DHT11,
- Hygrometer a
- Thermometer.

Dále jsou předdefinovány tyto aktuátory:

- BasicActuator,
- BinaryActuator,
- LinearActuator a
- RotaryActuator.

3.5.2 Komponenta DataFeed

Tato komponenta definuje rozhraní DataFeed a poskytuje třídu, již danému rozhraní poskytuje implementaci. Rozhraní DataFeed definuje především metodu `getNextValue(Object... params)`, která vrací objekt Data. Jak název této metody napovídá, při každém volání by měla vrátit další hodnotu ze série, kterou implementovaný objekt reprezentuje. Pro zvýšení flexibility této metody přijímá libovolné množství parametrů. Data je třída, která slouží jako kontejner pro hodnoty, který zajišťuje typovou bezpečnost. Pro uživatele je vytvořeno šest předdefinovaných tříd, které implementují rozhraní DataFeed. Jedná se o třídy:

- ConstantDataFeed,
- LinearDataFeed,
- ExpressionDataFeed,
- ExponentialDistDataFeed,
- NormalDistVariateDataFeed a
- DayTemperatureDataFeed.

Každá z těchto tříd při volání funkce getNextValue vrací objekt typu Data obsahující hodnotu typu double.

ConstantDataFeed

Jedná se o generátor, který při každém volání funkce getNextValue vrací konstantu, která byla této instanci předána při volání konstruktoru.

LinearDataFeed

Tento generátor implementuje lineární funkci s počátkem v nule. Při vytváření instance uživatel předává směrnici, která udává rozdíl mezi dvěma po sobě jdoucími hodnotami.

ExpressionDataFeed

Tento generátor generuje hodnoty na základě matematických výrazů. Tyto výrazy jsou závislé na jedné proměnné. U této třídy je využito vlastnosti, že funkce `getNextValue` přijímá libovolné množství parametrů. Tato třída pro svoji implementaci využívá knihovny exp4j[8], která umožňuje definovat výrazy pomocí řetězcových literálů.

ExponentialDistDataFeed

Generování dat na základě stochastických funkcí. Data stochastické funkce jsou generovány pomocí knihovny Stochastic Simulation in Java (SSJ)[9]. Tato třída modeluje Exponenciální rozložení hustoty pravděpodobnosti. Při vytváření se předává konstruktoru hodnota střední hodnoty četnosti výskytu (lambda) a při volání funkce getNextValue se předává jako parametr hodnota, pro kterou je distribuční funkce vyhodnocena.

NormalDistVariateDataFeed

Tato třída reprezentuje generování hodnot s normálním rozložením pravděpodobnosti. Při vytváření instance této třídy je potřeba zadat střední hodnotu a směrodatnou odchylku. Tyto dva parametry definují graf hustoty pravděpodobnosti generovaných hodnot. Generování hodnot v této třídě je stejně jako u ExponentialDistDataFeed založeno na využití knihovy SSJ.

DayTemperatureDataFeed

Generátor pro generování teploty v průběhu dne. Algoritmus implementovaný v této třídě vypočítává hodnotu na základě čtyř parametrů. Času, kdy nastává nejnižší teplota, času, kdy nastává nejvyšší teplota, minimální teploty během dne a maximální teploty během dne. První dvě hodnoty jsou zafixované na pátou a šestnáctou hodinu. Uživateli tedy zbývá při konstrukci instacne definovat minimální a maximální hodnotu během dne. Získávání další hodnoty ze sekvence dat je pak závislé na parametru času.

3.5.3 Komponenta NetworkAdapter

Tato komponenta je zodpovědná za distribuci dat pomocí různých síťových komunikačních protokolů. Pro interakci se zařízeními implementujícími rozhraní Device bylo vytvořeno rozhraní NetworkAdapter. To definuje tři funkce, které musí splňovat každá implementace tohoto rozhraní.

- `void send(List<Data> data)` – zpracuje data pomocí objektu DataWraper a pošle je.
- `void setDataWrapper(DataWrapper dataWrapper)` – nastaví pro danou instanci NetworkAdapter DataWraper.

[8] https://www.objecthunter.net/exp4j/
[9] http://simul.iro.umontreal.ca/ssj/

- `DataWrapper getDataWrapper()` – Vrátí DataWraper, který je nastavený pro danou instanci NetworkAdapter

DataWraper

Účel objektů tohoto typu je serializace předaných do formátu vhodného pro přenos dat. V době psaní tohoto textu modul PatrIoT Data Generator implemntuje dvě takovéto třídy:

- JSONWrapper – Serializace do formátu JSON.
- XMLWrapper – Serializace do formátu XML.

MQTT implementace rozhraní NetworkAdapter

Třída, jež je implementací rozhraní `NetworkAdapter`, zasílá data pomocí protokolu MQTT verze 3. Při vytváření instance je potřeba konstruktoru předat URI serveru. Instance poté vytvoří asynchronní spojení. Zdá se, že tato třída není ve skutečnosti kompletní, jelikož metody a `send`, `setDataWrapper` mají prázdné tělo a metoda `getDataWrapper` vrací za všech okolností `null`.

Rest implementace rozhraní NetworkAdapter

Tato implementace NetworkAdapter zasílá data na REST API koncové body pomocí síťového protokolu HTTP. Při vytváření instance je potřeba konstruktoru předat URL koncového bodu, na který se mají zasílat data, a dále je třeba předat DataWraper, který předaná data upraví do formátu, který je zadaný endpoint schopný zpracovat.

3.5.4 Komponenta Controller

Tato komponenta byla navržena k tomu, aby umožnila konfiguraci a ovládání modelovaných zařízení pomocí síťové komunikace. Tato komponenta umožňuje nad vytvořeným testovacím prostředím provádět externí změny a tím pádem umožňuje realizovat různorodé testovací scénáře (například je možné v průběhu měření vybrat jeden z teploměrů a změnit jeho generátor dat na takový, který simuluje poruchu zařízení). Tato komponenta je tvořena několika základními třídami popsanými níže. Implemetace CoAP komunukace je založena na knihovně Californium[10].

Transfer dat je implementován pomocí protokolu CoAP. Tento protokol je založen na modelu Klient-server. Tato komponenta řeší konfiguraci zařízení tím, že vytváří singleton instanci serveru, na který lze zasílat požadavky a ten nadále tyto požadavky zpracovává nad zdroji, které k němu byly zaregistrovány.

Identifikace jednotlivých koncových zařízení je v protokolu CoAP dosažena podobně jako u protokolu HTTP pomocí strukturovaného URI. Jednotlivé obsluhy pro jednotlivé koncové body jsou implementovány pomocí instancí třídy Resource.

Obsluha CoAP zdrojů třídou Resource

Každý z CoAP dotazů na libovolný zdroj na serveru obsahuje identifikátor, o jakou dotazovací metodu se jedná. Dotaz může být jedna z následujících metod:

- GET,

[10] https://eclipse.dev/californium/

- POST,
- PUT,
- DELETE,
- FETCH,
- PATCH a
- iPATCH,

přičemž každá z těchto metod má definovanou sémantiku. [24]

Pokud má koncový bod některé z těchto metod obsloužit, je třeba aby dědil od třídy `Resource` z knihovny Californium a implementoval některé z metod vypsaných výše. Knihovna PatrIoT Data Generator implementuje pět takovýchto zdrojů, které jsou popsány v tabulce 3.1.

název třídy	**metoda**	**sématika**
ActuatorRootResource	–	definuje část URI pro aktuátor
ActuatorResource	GET	poskytuje aktuální stav aktuátoru
	POST	přepínání mezi stavy aktuátoru (ON, OFF)
SensorRootResource	–	definuje část URI pro senzor
SensorResource	GET	poskytuje základní informace o senzoru
	POST	přepínání mezi stavy aktuátoru (aktivní, neaktivní)
DataFeedResource	GET	poskytne základní informace o generátoru dat
	POST	V těle zprávy je očekávána definice data generátoru ve formátu JSON. Tento generátor je přidán k danému zařízení.

Tabulka 3.1: Přehled předdefinovaných Tříd rozšiřující Resource. V tabulce jsou vypsány metody, které jednotlivé třídy implementují a také je popsána jejich sémantika.

CoapControlServer

Knihovna Californium implementuje CoAP server. Tato třída se stará o správnou konfiguraci tohoto serveru pro účely použití PatrIoT Data Generátoru.

Tento modul je navržen tak, aby obsahoval pouze jeden konfigurační server. Tento požadavek byl při návrhu této třídy podpořen návrhovým vzorem singleton.

Aby bylo zařízení konfigurovatelné pomocí tohoto serveru, je potřeba jej do serveru zaregistrovat. Registrace, respektive odregistrace, lze docílit využitím metody `add` respektive `remove`.

CoapController

Tato třída představuje pro uživatele jednoduchý způsob nastavení externí konfigurace pro zařízení. Každému zařízení, které má být přístupné pomocí protokolu CoAP, stačí vytvořit instanci implementující rozhraní CoapController, předat zařízení konstruktorovi a poté nad instancí zavolat metodu `registerDevice`.

Třida implementující rozhraní CoapController má za úkol vytvořit odpovídající instance Resource a registraci těchto zdrojů do CoapControlServeru. V době psaní tohoto textu jsou v této komponentě dostupné dvě implementace tohoto rozhraní - ActuatorCoapController a SensorCoapController.

Kapitola 4

Modely pro simulace komplexních systémů

Tato diplomová práce si klade za cíl rozšíření stávajícího frameworku o podporu pro simulaci náhodných událostí, což je jedna z klíčových vlastností pro efektivní testování IoT systémů. IoT zahrnuje širokou škálu chytrých zařízení, z nichž jsou sestaveny systémy, které nacházejí uplatnění v mnoha oblastech lidské činnosti. Jako příklad jsou uvedeny následující [16]:

- **Průmysl 4.0** – průmysl využívající propojení výrobních linek skrze technologie IoT, pro optimalizaci výrobních procesů a efektivní sledování zdrojů.
- **Smart Home(Chytrá domácnost)** – využití chytrých komponent pro vzdálené a automatizované ovládání domu, které umožňuje zvýšení komfortu a zároveň nižší energetickou náročnost.
- **Smart City (Chytré město)** – do této kategorie spadá inteligentní osvětlení veřejných prostorů, inteligentní řízení dopravy, monitorování kvality prostředí a další.
- **Doprava** – autonomní řízení, komunikace mezi vozidly (v2v), vzdálená diagnostika.
- **Zemědělství** – monitorování a automatizace zavlažování, sledování zdraví rostlin, automatizace sběru.

Vysoká diverzita IoT systémů způsobuje, že události, které má rozšířený modul PatrIoT Data Generator podporovat, nelze popsat jediným jednoduchým a univerzálním modelem. V následujících sekcích budou popsány techniky modelování, které umožňují vytváření simulací různých dějů a popisování vazeb mezi nimi. Vazby definují, jak se modely navzájem ovlivňují.

4.1 Popis různých prostorů a systémů souřadnic

Popisem vlastností různých prostorů a vztahů mezi objekty, které se v nich nachází, se zabývá Geometrie. Existují různé popisy prostorů jako například axiomatický popis, topologické prostory a variety, ale tyto popisy jsou pro praktické modelování příliš abstraktní. Tato sekce popisuje různé souřadnicové systémy, které byly zvoleny jako vhodné pro reprezentaci modelů spojených se simulováním IoT v reálném světě. Zastřešujícím pojmem, pod který spadají všechny zvolené popisované prostory, je metrický prostor popsaný na začátku

této sekce 1. Metrický prostor byl zvolen z důvodu, že se během studia různých konkrétních popisů prostoru ukázala možnost měřit vzdálenost jednotlivých aktérů jako klíčová.

Souřadnice v této sekci byly rozděleny na diskrétní a spojité pro různé potřeby simulací.

4.1.1 Historie systémů souřadnic

V dobách antiky se pro popis prostoru používaly axiomy. Například Eukleides stanovil prostor, ve kterém se zabýval matematikou pomocí následujících axiomů:

1. Lze vytvořit úsečku, která spojuje dva dané body.
2. Danou úsečku lze na obou stranách libovolně prodloužit.
3. Lze vytvořit kruh o daném středu, na jehož obvodě leží daný bod.
4. Všechny pravé úhly jsou si rovny.
5. Jestliže úsečka protíná dvě úsečky tak, že na jedné straně je součet vnitřních přilehlých úhlů menší než dva pravé úhly, pak lze na této straně úsečky prodloužit tak, aby se tato jejich prodloužení protala.

Tento systém sice vytvářel aparát pro formální důkazy, avšak selhával na poli vytváření kvantitativních predikcí a číselného popisu objektů. Z tohoto důvodu je naprosto nepraktické tento systém využívat k softwarovému modelování.

Zvrat nastal s příchodem matematika René Descarta a jeho popisem prostoru pomocí kartézského systému souřadnic, což vedlo ke vzniku dnešní analytické geometrie.

Souřadnicové systémy umožňují převádět geometrické problémy do reprezentace, která je vhodná pro použití technik matematické analýzy. Nad různými prostory lze vytvářet nespočet systémů souřadnic. V závislosti na souřadnicovém systému zvoleného pro daný prostor je možné definovat různé operace s prvky tohoto prostoru. Náročnost implementace těchto operací a intuitivnost jejich použití se může značně lišit v závislosti na vybraném souřadnicovém systému (viz definice rotace pro 2D kartézské souřadnice 4.2 a rotace v polárních souřadnicích 4.3). Z tohoto důvodu je důležité pro reprezentaci bodů a dalších objektů v prostoru vždy zvolit vhodný souřadnicový systém v závislosti na tom, jaký je zadaný problém.

Souřadnicemi v této sekci bude myšlena libovolná reprezentace bodů v prostoru pomocí prvků vhodné množiny, které se zobrazují do bodů odpovídajícího prostoru.

4.1.2 Metrický systém souřadnic

Výchozím bodem je definice metrického prostoru, která je překladem definice metrického prostoru z knihy Introduction to Metric and Topological Spaces[27].

Definice 1 (Metrický prostor) *Metrický prostor je tvořen neprázdnou množinou X společně s funkcí $d : X \times X \to \mathbb{R}$ která splňuje následující vlastnosti.*

1. *(nenulová vzdálenost různých prvků)* $\forall x, y \in X : d(x, y) \geq 0$ *a* $d(x, y) = 0 \Leftrightarrow x = y$
2. *(symetrie)* $\forall x, y \in X : d(x, y) = d(y, x)$
3. *(trojúhelníková nerovnost)* $\forall x, y, z \in X : d(x, z) \leq d(x, y) + d(y, z)$

Předchozí definice jinými slovy říká, že metrický prostor je množina prvků pro kterou je definována funkce, která pro libovolné dva prvky z množiny určí jejich vzdálenost. Funkce vzdálenosti může být jakákoli funkce, která splňuje tři vlastnosti, a to že nulovou vzdálenost má pouze prvek sám od sebe, dva prvky mají stejnou vzdálenost od sebe nehledě na to, jestli měříme vzdálenost od prvního prvku k druhému, nebo od druhého prvku k prvnímu a nakonec, že vzdálenost mezi dvěma body je vždy menší nebo rovna vzdálenosti cesty mezi těmito body přes body jiné.

Velice jednoduchým příkladem metrického prostoru je množina reálných čísel $\mathbb{R}$ společně s operací odečítaní v absolutní hodnotě $|-| : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Tento prostor, byť je velice jednoduchý, už může najít uplatnění i v simulaci, například pro reprezentaci času, nebo v simulaci, kde stačí zaznamenat skalární veličiny. Konkrétním příkladem může být parkovací senzor, který podle vzdálenosti předmětu vydává různé signalizační zvuky.

Nad libovolným prostorem, který splňuje definici výše definovaného metrického prostoru 1, lze zavést metrický systém souřadnic.

Definice 2 (Metrický systém souřadnic) *Nechť (M, d) je metrický prostor, a nechť platí $X \subset M$. Metrická souřadnicová množina pro X je množina prvků $C \subset M$, pro kterou platí, že pro všechny $x, y \in X$ kde $x \neq y$, existuje nějaký prvek c, $c \in C$, pro který platí $d(x, c) \neq d(y, c)$.*

N-tici (M, d, X, C) pak nazýváme ***metrický systém souřadnic****.*

Tím pádem, každý prvek $x \in X$ je v metrickém souřadnicovém systému (M, d, X, C) reprezentován C-ticí reálných čísel.

$$x_C = (x_c)_{c \in C} \in \mathbb{R}^C$$

kde

$$x_c := d(x, c).$$

x_c je c-tá metrická souřadnice prvku x.

[8]

Tato definice říká, že je v metrickém prostoru možné zvolit množinu prvků C (bázi metrických souřadnic), díky které je možné za použití funkce vzdálenosti určit každý další bod v prostoru. Pro příklad s prostorem reálných čísel a funkcí vzdálenosti zadanou rozdílem v absolutní hodnotě nastíněným výše, je možné zvolit $C = (0, 1)$ a $X = \mathbb{R}$, pak pro určení libovolného bodu x lze dopočítat jeho souřadnice jako $x_C = (|x - 0|, |x - 1|)$.

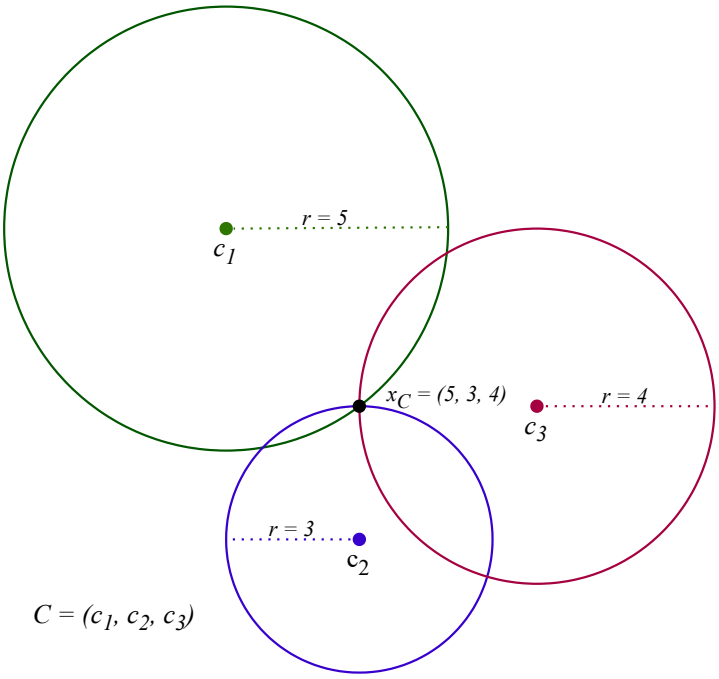
Určení bodu ze souřadnic probíhá postupem, kdy pro každou souřadnici určíme množinu bodů v prostoru, která je ve vzdálenosti, kterou udává souřadnice od referenčního bodu pro danou souřadnici. Průnikem těchto množin pak dostaneme právě jeden bod v prostoru. Grafické znázornění tohoto postupu je vyobrazeno na obrázku 4.1.

4.1.3 Diskrétní souřadnicové systémy

V této práci se na diskrétní prostor nahlíží jako na graf. Diskrétní souřadnicové systémy jsou definovány jako zobrazení souřadnic do množiny vrcholů.

Grafy společně s metrikou definovanou níže 7 spadají do kategorie metrických prostorů 1. Následující definice jsou převzaty z kapitoly 3 knihy *Kapitoly z diskrétní matematiky* [20].

Definice 3 *Graf G je uspořádaná dvojice (V,E), kde V je nějaká neprázdná množina a E je množina dvouprvkových podmnožin množiny V. Prvky množiny V se jmenují vrcholy grafu G a prvky množiny E hrany grafu G.*



Obrázek 4.1: Ukázka určení bodu pomocí metrických souřadnic. Bod x má v metrických souřadnicích vůči bázi $C = (c_1, c_2, c_3)$ souřadnice $x_C = (5, 4, 3)$

Následuje definice cesty v grafu 4, která je dále využita pro definici souvislého grafu 5.

Definice 4 *Cesta v grafu je posloupnost*

$$(v_0, e_1, v_1, \ldots, e_t, v_t),$$

kde $v_0, v_1, \ldots, v_t$ jsou navzájem různé vrcholy grafu G, a pro každé $i = 1, 2, \ldots, t$ je $e_i = \{v_{i-1}, v_i\} \in E(G)$. Obšírněji se cesta $(v_0, e_1, v_1, \ldots, e_t, v_t)$ nazývá cesta z v_0 do v_t délky t.

Definice 5 *Řekneme, že graf je souvislý, jestliže pro každé dva jeho vrcholy x a y v něm existuje cesta z x do y*

Definice 6 *Orientovaný graf G je dvojice (V, E), kde E je podmnožina kartézského součinu $V \times V$. Prvky E nazýváme šipky (nebo orientované hrany). Tedy šipka e má tvar (x, y). Říkáme, že tato šipka vychází z x a končí v y.*

Definice 7 *Nechť $G = (V, E)$ je souvislý graf. Pro vrcholy v, v' definujeme číslo $d_G(v, v')$ jako délku nejkratší cesty z v do v' v grafu G. Číslo $d_G(v, v')$ se nazývá vzdálenost vrcholů v a v' v grafu G.*

Následující věta je přímým důsledkem definice.

Věta 1 *Funkce $d_G : V \times V \to \mathbb{R}$, kterou nazýváme metrika grafu G, má následující vlastnosti (tj. (G, d_G) je metrický prostor dle definice 1)*

1. $(d_G(v, v') \geq 0)$, *a* $(d_G(v, v') = 0)$ *právě když* $(v = v')$;
2. $(symetrie)(d_G(v, v') = d_G(v', v))$ *pro každou dvojici vrcholů* (v, v');
3. *(trojúhelníková nerovnost)* $(d_G(v, v'') \leq d_G(v, v') + d_G(v', v''))$ *pro každou trojici* $(v, v'$, $v'')$ *vrcholů z (V).*

Výše definovaná funkce d_G *má navíc ještě následující speciální vlastnosti:*

4. $d_G(v, v')$ *je nezáporné celé číslo pro každou dvojici* v, v';
5. *Jestliže* $d_G(v, v'') > 1$, *potom existuje* v', $v \neq v' \neq v''$ *tak, že* $d_G(v, v') + d_G(v', v'') = d_G(v, v'')$.

Typickým algoritmem pro implementaci metriky je Dijkstrův algoritmus. Tento algoritmus lze použít, jak pro grafy (každá hrana má hodnotu implicitně hodnotu 1), tak pro grafy s ohodnocenými hranami.

Definice 8 ***Vstup:*** *graf* $G = (V, E)$, *ohodnocení hran* $w(e)$, $e \in E$, *startovní vrchol* s. ***Proměnné:*** *čísla* $d(v)$, $v \in V$, *číslo* δ, *množiny* $A, N \subseteq V$. ***Výstup:*** *po skončení algoritmu udává* $d(v)$ *vzdálenost* v *od* s, *pro každé* $v \in V$.

1. ***Inicializace***
   $d(s) := 0$; $d(x) := \infty$ *pro* $x \in V \setminus \{s\}$; $A := V$.
2. ***Test ukončení***
   Jestliže pro všechna $x \in A$ *platí* $d(x) = \infty$, *algoritmus končí, jinak se pokračuje krokem 3.*
3. ***Volba množiny*** N
   $\delta := \min\{d(y); y \in A\}$
   $N := \{v \in A; d(v) = \delta\}$
   $A := A \setminus N$
4. ***Aktualizace*** $d(y)$ ***pro sousedy*** N
   Pro každou hranu $e = \{v, y\}$, *kde* $v \in N$ *a* $y \in A$, *proveď příkaz*
   $d(y) := \min(d(y), d(v) + w(e))$.
   Po vyčerpání všech takových hran e pokračuj krokem 2.

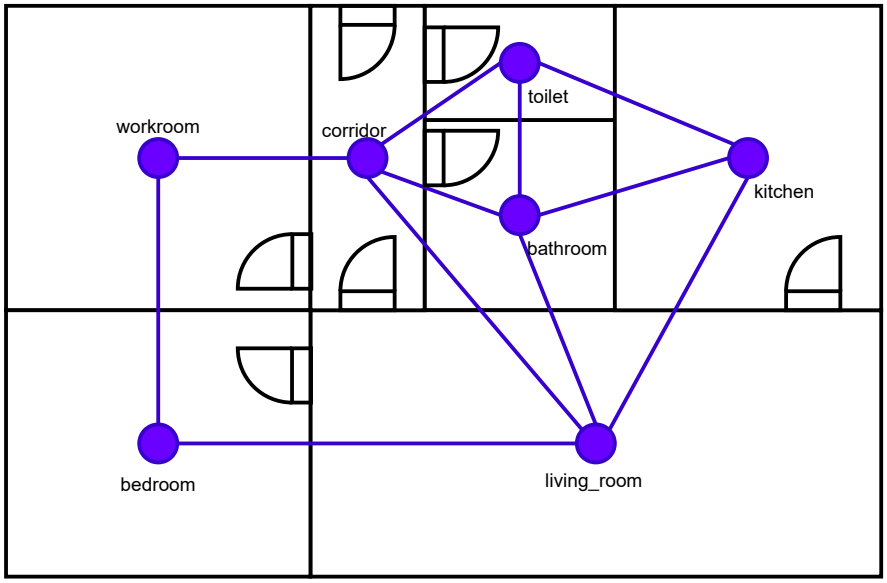
Grafové souřadnicové systémy jsou vhodné pro reprezentaci datových struktur skládajících se z uzlů a hran, kde topologie hraje důležitější roli než geometrická poloha. Umožňují zaznamenat vztahy mezi prvky systému. Následuje popis vybraných souřadnicových systémů nad grafy.

Jednoduchý grafový souřadnicový systém

Patrně nejjednodušším souřadnicovým systémem, který lze nad grafem zkonstruovat je určení polohy samotným vrcholem. Formální definice je následující:

Definice 9 *Jednoduchý grafový souřadnicový systém je dvojice* (V, φ), *kde* V *je množina vrcholů grafu a* φ *je funkce* $\varphi : V \to V$ *definovaná jako* $\varphi(v) = v$.

Tyto souřadnice lze snadno rozšířit přejmenováním vrcholů.



Obrázek 4.2: Ukázka využití grafového souřadnicového systému s přejmenováním. Nad plánem domu je promítnut graf, který vytváří model domu zaznamenávající sousednost místností.

Definice 10 *Grafový souřadnicový systém s přejmenováním je trojice* (V, X, φ)*, kde* V *je množina vrcholů grafu a* φ *je funkce* $\varphi : X \to V$*, která je bijektivní.*

V praxi je tento systém souřadnic vhodný zejména pro pohodlí uživatele, kdy lze vrcholy pojmenovat intuitivními názvy a s těmito názvy pracovat i v situacích, kdy jsou samotné vrcholy komplexní struktury. Na obrázku 4.2 je zobrazen plán domu, do kterého je promítnut graf reprezentující pokoje a jejich sousedící stěny. Díky pojmenování vrcholů odpovídají souřadnice vrcholů modelované doméně.

Mřížkový grafový souřadnicový systém

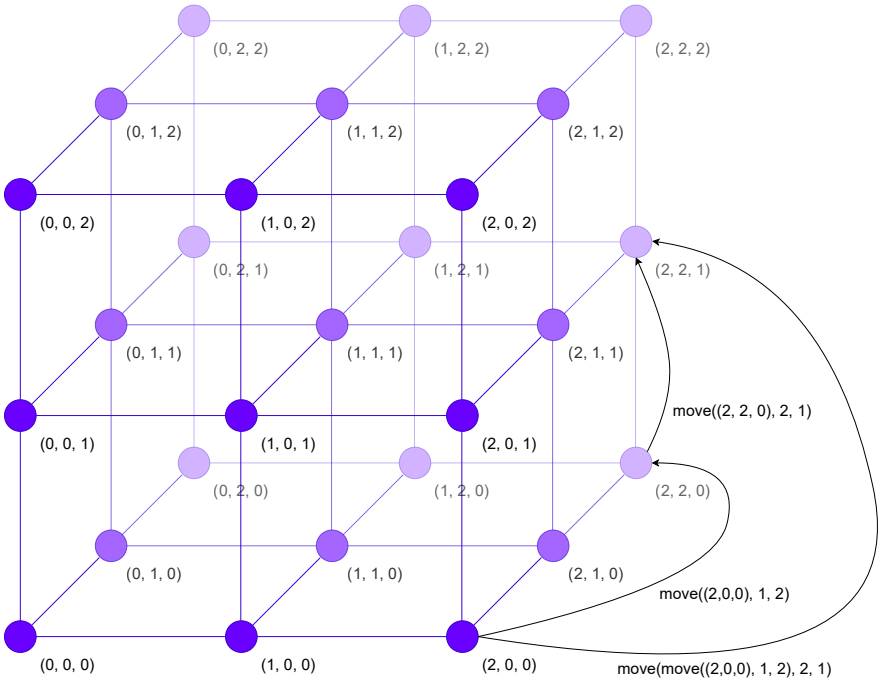
Nechť je Z_n množina prvních $n-1$ přirozených čísel včetně nuly, tj. $Z_n = \{0, \ldots, n-1\}$.

Speciálním typem grafu je tzv. mříž, tj. graf na množině $(Z_n)^k$, kde $k, n \in \mathbb{N}$ a množina hran je definována následovně: $\{\{(x_1, \ldots, x_i, \ldots, x_k), (x_1, \ldots, x_i+1, \ldots, x_k)\} | i \in \{1, \ldots, k\}, x_1, \ldots, x_k \in Z_n,\ x_i \neq n-1\}$

Tento graf vytváří pravidelnou strukturu, ve které jsou vrcholy grafu uspořádány do mřížky o hraně délky n v k dimenzích. Viz obrázek 4.3

Pro mříž jsou přirozené souřadnice právě množina vrcholů tohoto grafu. V těchto souřadnicích lze snadno vypočítat vzdálenost a to pomocí následujícího vzorce.

$$d_{Z_n^k}((x_1, \ldots, x_k), (y_1, \ldots, y_k)) = \sum_{i=1}^{k} |x_i - y_i|$$



Obrázek 4.3: Vizualizace souřadnic na mříži pro $n = 2$ a $k = 3$. V pravé části je zobrazena operace `move((2, 0, 0), 1, 2)`, která vypočítá souřadnici jež je v 1. dimenzi posunuta o vzdálenost 2 od vrcholu $(2, 0, 0)$. Dále je zobrazena operace `move((2, 2, 0), 2, 1)`, která vypočítá souřadnici jež je v 2. dimenzi posunuta o vzdálenost 1 od vrcholu $(2, 2, 0)$. Třetí zobrazená operace je `move(move((2,0,0), 1, 2), 2, 1)`, která je složením předchozích dvou operací.

Nad mřížkovými souřadnicemi lze snadno definovat operace jako:

- Posun o vzdálenost l v d-té dimenzi od souřadnice c (`move(c, l, d)`).
- Získání všech vrcholů blíže než l od souřadnice c (`neighbors(c, l)`).

Operace `move(c, l, d)` je rovněž zobrazena v obrázku 4.3

Tytou souřadnice jsou využívány například v modelech celulárních automatů, které jsou popsány v sekci 4.2.3.

4.1.4 Spojité souřadnicové systémy

Pro formální definici spojitého systému souřadnic z pohledu geometrie je potřeba mít nadefinované pojmy topologická varieta, otevřená podmnožina a homeomorfismus. Tyto pojmy jsou nad rámec této práce, ale zvídavý čtenář si je může dohledat například v knize *Introduction to Smooth Manifolds* [18].

Pro účely v této práci postačí zjednodušená definice která definuje spojité souřadnice jako n-tice čísel společně se zobrazením do prostoru $\mathbb{R}^n$.

Definice 11 *Spojitý souřadnicový systém nad množinou $\mathbb{R}^n$, které říkáme spojitý prostor, je uspořádaná trojice $(\mathbb{R}^n, V, \varphi)$, která využívá uspořádané n-tice reálných čísel (souřadnice) pro určení prvků prostoru. φ je zobrazení $\varphi : V \to \mathbb{R}^n$, kde $V \subseteq \mathbb{R}^n$.*

Všechny následující souřadnicové systémy předpokládají, že vzdálenost dvou bodů v $\mathbb{R}^n$ se měří pomocí tzv. *euklidovské metriky*, tj.

$$d((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{4.1}$$

a mezi dvěma vektory $(x_1, \ldots, x_n)$ a $(y_1, \ldots, y_n)$ je dán úhel pomocí vztahu

$$\alpha = \cos^{-1}\left(\frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{n}(x_i)^2 \cdot \sum_{i=1}^{n}(y_i)^2}}\right)$$

Díky tomu, že je pro tyto souřadnicové systémy poskytnuta metrika, lze na ně nahlížet jako na spojitý metrický prostor. Společně s definicí úhlu lze tento prostor považovat za euklidovský [5].

Kartézský souřadnicový systém

Jedná se patrně o nejpoužívanější systém souřadnic. Byl zaveden francouzským matematikem René Descartem v 17. století. Tento systém je založen na množině ortogonálních os, které se protínají v počátku. Bod v prostoru je pak určen jako průnik rovin, kdy každá rovina má jako normálu osu ke které patří a protíná ji v dané vzdálenosti od počátku. Pro n dimenzionální prostor s n souřadnicovými osami jsou souřadnice zaznamenány jako uspořádaná n-tice, která zaznamenává vzdálenosti rovin od počátku. Při použití těchto souřadnic je potřeba stanovit k jednotlivým položkám n-tice odpovídající osy.

Ve smyslu definice výše (4.1.4) se efektivně jedná o identické zobrazení, tj. $V = \mathbb{R}^n$ a $\varphi(x) = x$.

Užitečnost dvou a tří dimenzionálních kartézských souřadnic spočívá v tom, že popisují reálný svět pomocí skalárních veličin. Našel využití v mnoha odvětvích lidské činnosti včetně:

- fyziky – klasická mechanika, popis pohybu vesmírných těles v astronomii,
- inženýrství – výpočty a plánování konstrukce budov, dopravních komunikací a strojů,
- počítačová grafiky – tvorba 3D modelů a jejich projekce do 2D obrazu,
- robotiky – navigace a manipulace robotů v reálném prostředí,

a mnoho dalších.

Nad tímto prostorem lze mnoho matematických operací (např. rotace či škálování) modelovat pomocí maticového násobení.

2D kartézský souřadnicový systém

Jedná se o specifický případ kartézského prostoru. Je vhodný pro reprezentaci objektů v rovině. Konkrétním příkladem může být zaznamenání aproximace polohy na povrchu tělesa (vytvoření geografické mapy). Bývá též používán jako nástroj pro zobrazení funkcí jedné proměnné, komplexních čísel, nebo jako prostor pro 2D analytickou geometrii. Osy se běžně označuj písmeny x a y.

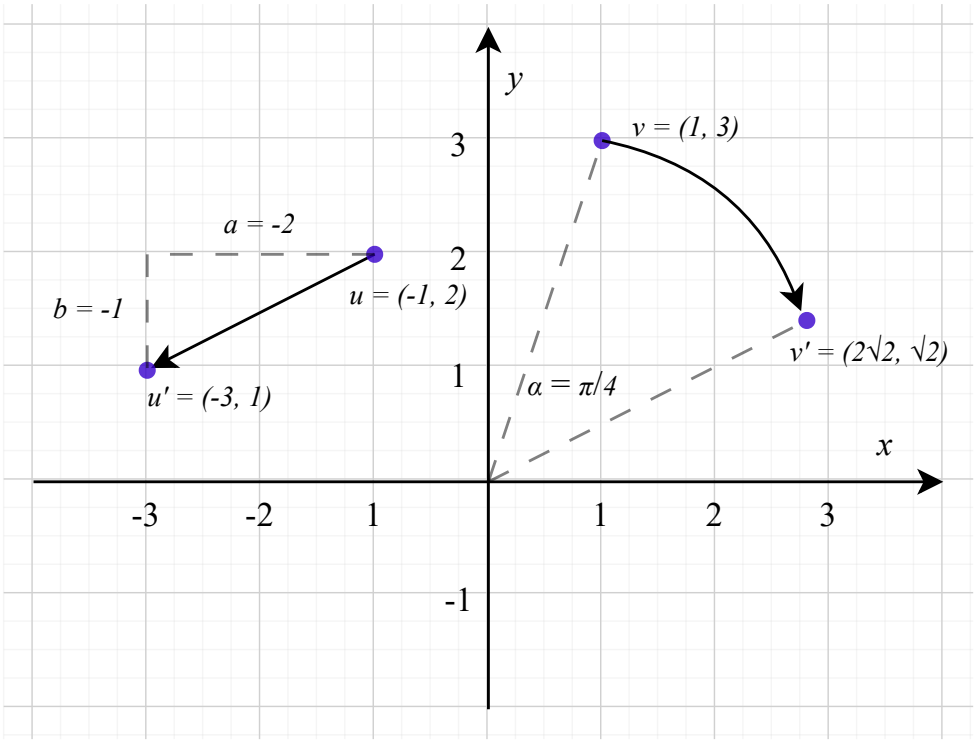
Rovnice, které definují rotaci bodu o úhel α okolo počátku proti směru hodinových ručiček lze pomocí maticové operace vyjádřit následovně:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.2}$$

Translace o vektor (a, b) je definována následovně:

$$x' = x + a$$
$$y' = y + b$$

Z definic je patrné, že je v těchto souřadnicích výpočet translace jednoduší než rotace. Obě tyto operace jsou zobrazeny na obrázku 4.4.



Obrázek 4.4: Ukázka operací translace a rotace v dvourozměrném kartézském prostoru. V levém horním kvadrantu je zobrazena translace bodu $u = (-1, 2)$ o -2 v ose x a o -1 v ose y. V pravém horním kvadrantu je zobrazena rotace bodu $v = (1, 3)$ o úhel $\pi/4$

Polární Souřadnice

Polární souřadnicový systém určuje polohu bodu v $\mathbb{R}^2$ a to pomocí dvojice (ρ, θ).

- $\rho \in \langle 0, \infty)$ – udává vzdálenost bodu od počátku.
- $\theta \in \langle -\pi, \pi \rangle$ – úhel svíraný mezi kladnou poloosou x a spojnicí bodu a počátku

Tyto souřadnice je výhodné použít v situacích kdy je přirozené pracovat s úhly a vzdálenostmi. Tyto souřadnice přirozeně vznikají při používání radaru, který nám udává směr objektu a vzdálenost.

Převod polárních souřadnic na kartézské je dán vztahy:

$$x = \rho \cos(\theta)$$
$$y = \rho \sin(\theta)$$

Převod z kartézských souřadnic na polární pak vypadá takto:

$$\rho = \sqrt{x^2 + y^2}$$
$$\theta = \operatorname{sgn}(y) \cos^{-1}\left(\frac{x}{\rho}\right)$$

Rotace o úhel α je dána vztahy:

$$\rho' = \rho \tag{4.3}$$
$$\theta' = \theta + \alpha$$

Translace o vektor (a, b), kde $b > 0$ vypadá následovně

$$r' = \sqrt{r^2 + a^2 + b^2 + 2r\sqrt{a^2 + b^2}\cos(\cos^{-1}(a/\sqrt{a^2 + b^2}) - \theta)}$$
$$\theta' = \arccos\left(\frac{r\cos(\theta) + a}{r'}\right)$$

Na těchto dvou operacích je patrné, že výpočet rotace je v polárních souřadnicích výrazně jednodušší než translace. Náročnost těchto operací je opačná než jak je tomu u kartézských souřadnic. Tento fakt demonstruje důležitost výběru vhodného souřadnicového systému.

Sférické Souřadnice

Sférické souřadnice zobecňují polární souřadnice do třetího rozměru. Polohu bodu určujeme pomocí trojice čísel, (ρ, ϕ, θ).

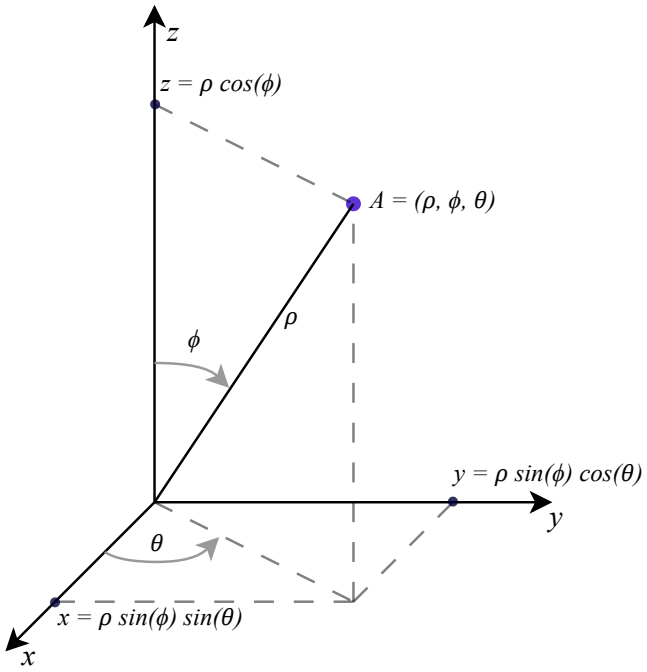
- $\rho \in \langle 0, \infty)$ – udává vzdálenost bodu od počátku.
- $\phi \in \langle 0, \pi\rangle$ – úhel svíraný mezi kladnou poloosou z a spojnicí počátku a bodu.
- $\theta \in \langle -\pi, \pi\rangle$ – úhel mezi kladnou poloosou x a projekcí bodu do roviny x-y.

Všechny tyto parametry jsou zobrazeny na obrázku 4.5.

Tento systém je vhodný pro určování pozice vesmírných těles vůči zemi, nebo pro určování polohy na zemi – v tomto případě je možné aproximovat tvar geoidu pomocí koule a určit fixní poloměr (6378 km), pak lze polohu zadávat pouze pomocí ϕ (zeměpisná šířka) a θ (zeměpisná délka), což odpovídá souřadnicím GPS.

Převod mezi kartézskými a sférickými souřadnicemi je dán vztahy:

$$x = \rho \sin(\phi) \cos(\theta)$$
$$y = \rho \sin(\phi) \sin(\theta)$$
$$z = \rho \cos(\phi)$$



Obrázek 4.5: Ukázka bodu A, který má ve sférických souřadnicích polohu (ρ, ϕ, θ). Obrázek dále ukazuje, jak tento bod převést ze sférických souřadnic do kartézských ($\mathbb{R}^3$).

Nazpět je pak možné souřadnice převés pomocí rovnic:

$$\rho = x^2 + y^2 + z^2$$
$$\phi = \cos^{-1}(\frac{z}{\rho})$$
$$\theta = \mathrm{sgn}(y)\cos^{-1}\left(\frac{x}{\rho}\right)$$

4.2 Základní modely

Tato sekce se zaměřuje na různé modelovací přístupy, které jsou použitelné pro simulaci událostí, které ovlivňují IoT systémy. Kromě tradičního matematického modelování pomocí rovnic se tato sekce zabývá i dalšími metodami jako jsou agentní systémy a celulární automaty.

4.2.1 Matematické modelování

Matematické modely jsou tvořeny rovnicemi, které popisují vztahy mezi entitami z reálného světa. Jedná se patrně o nejstarší techniku pro modelování. Proces modelování je založen na vyjádření stavu systému pomocí stavových proměnných a následném vyjádření vztahů mezi nimi pomocí rovnic.

Patrně nejznámějším příkladem matematického modelování, který použil Leonardo Fibonacci ve své knize Liber Abaci [25], je výpočet králičí populace. Tento model vychází z následujících (nerealistických) předpokladů:

- V prvním měsíci je v modelu pouze jeden pár králíků.
- Nový pár králíků zplodí další pár králíků až ve druhém měsíci svého života.
- Ve chvíli, kdy pár začne plodit další páry, plodí je každý další měsíc.
- Králíci jsou nesmrtelní.

Pokud si stavovou proměnnou počet králíků v měsíci t vyjádříme jako X_t, pak výše zmíněné vztahy lze vyjádřit rekurzivní rovnicí:

$$X_t = X_{t-1} + X_{t-2}, \text{ kde počáteční hodnoty jsou } X_0 = 0 \text{ a } X_1 = 1$$

Dostáváme tak předpis generující známou Fibonacciho posloupnost. Tato diferenční rovnice vytváří velice jednoduchý matematický model s diskrétním časem.

4.2.2 Matematické modely založené na diferenciálních rovnicích

V praxi se velice často používají modely se spojitým časem založené na diferenciálních rovnicích. Jednoduchý systém závaží kmitajícího na pružině zobrazený na obrázku 4.6 je příkladem systému, který lze popsat diferenciální rovnicí. Obyčejná diferenciální rovnice modelující tento systém pak vypadá následovně:

$$y'' = -g - \frac{k}{m}y$$

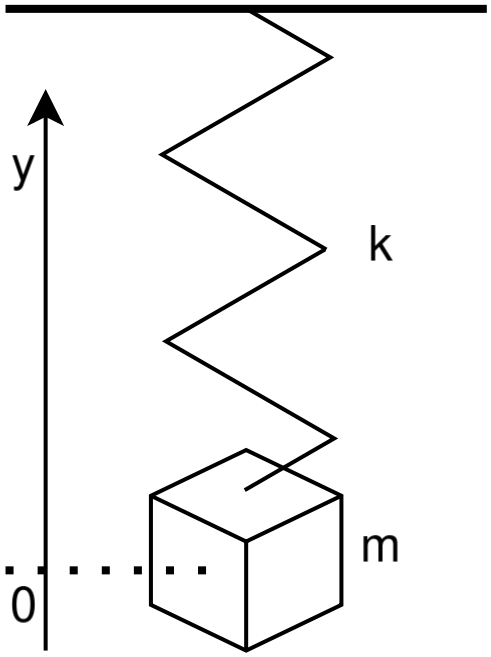
kde:

- y: vzdálenost od rovnovážné polohy, $[m]$
- g: tíhové zrychlení $[m/s^2]$
- k: tuhost pružiny $[N \cdot m^{-1}]$
- m: hmotnost závaží $[kg]$

Pro jedinečnost řešení je ještě potřeba určit počáteční podmínky. Rovnice uvedená výše je pak již natolik triviální, že ji lze řešit analyticky, avšak u složitějších modelů se lze snadno dostat do situace, kdy analytické řešení není známo, nebo jeho nalezení je příliš komplikovaný proces. Proto se v simulacích matematických modelů založených na diferenciálních rovnicích většinou využívají numerické metody. Ty sice neposkytují přesný výsledek, avšak pro účely simulací bývá jejich přesnost většinou dostatečná.

Pro řešení obyčejných diferenciálních rovnic prvního řádu se nejčastěji používá Eulerova metoda. Začíná v počátečních podmínkách a na základě zadané velikosti kroku, se kterým se má postupovat, postupně odhaduje průběh řešení.

Pro větší přesnost výsledků se používají metody Runge-Kutta, avšak za cenu náročnějšího výpočtu. [21]



Obrázek 4.6: Konceptuální model závaží na pružině. k značí tuhost pružiny a m značí hmotnost.

4.2.3 Využití agentních systémů pro modelovnání IoT systémů

Agentní systémy je koncept hojně používaný v informatice, který ovšem nemá jednu exaktní definici. V této sekci se jsou agentní systémy popisovány ve shodě s popisem z knihy An Introduction to Multi Agent Systems [30].

Základní koncept určuje, že existuje prostředí, ve kterém se vyskytují agenti, kteří získávají informace o prostředí pomocí vjemů. Každý agent má v daném prostředí svůj cíl, kterého se snaží dosáhnout. Na základě získaných informací o prostředí agenti vykonávají akce, které mění stav prostředí.

Tohoto formalismu se využívá i u modelování založeném na agentech, které je vhodné pro zkoumání komplexních systémů, které se skládají z prvků, které se navzájem ovlivňují, a pro systémy, kde nás zajímá emergentní chování. Modelování založené na agentech našlo uplatnění i při modelování IoT systémů, jelikož chytré zařízení mají mnoho podobností s definicí agenta, a tak k využití tohoto formalismu přímo vybízí. Konkrétním případem je například vytvoření modelu IoT systému pro řízení dopravy [15].

Agent

Abstraktní model agenta se skládá z různých subsystémů. Tyto subsystémy určují, do jaké kategorie daný agent spadá. Základní subsytémy, které se vyskytují ve všech agentech jsou *senzory*, *aktuátory* a *agentní funkce*.

Prostřednictvím senzorů získává agent informace o prostředí. Tyto získané informace se nazývají *vjemy*. Posloupnosti všech takto získaných informací se nazývá *sekvence vjemů*.

Aktuátory jsou pak prostředky skrze které agent ovlivňuje prostředí. Každý aktuátor má v daném okamžiku množinu proveditelných akcí (ne každá akce je proveditelná v každém okamžiku).

Agent má také agentní funkci, ta definuje chování agenta. Jedná se o funkci, která přiřazuje sekvenci vjemů nějakou akci.

Typy prostředí

Prostředí, ve kterém se vyskytují agenti lze dělit v několika osách. Prostředí se dělí například na plně a částečně pozorovatelné. Dále jde dělit podle počtu agentů na prostředí s jedním agentem, nebo na multiagentní prostředí, na deterministické, nebo stochastické a další.

Celulární automaty

Automat je teoretický stroj, který mění svůj vnitřní stav na základě aktuálních vstupů a vnitřního stavu. Množina možných vnitřních stavů bývá diskrétní a konečná.

Tyto automaty modelují děje, které jsou vázány na prostředí. Vyznačují se tím, že za pomoci jednoduchých pravidel, které jsou lokálního charakteru, jsou schopny vytvářet dynamické systémy s velice komplexním chováním. Celulární automaty (CA) prokázali svoji užitečnost v řadě oblastí – od ekologie, biologie přes ekonomii až po novou klasifikaci komplexity systémů navrženou Stephenem Wolframem [29].

Následuje definice jedno-dimenzionálního celulárního automatu založená na článku z časopisu Nature *Cellular automata as models of complexity* [28].

Jedno-dimenzionální automat se skládá z nekonečné sekvence buněk, kde každá buňka nese hodnotu v rozmezí $0, \ldots, k-1$. Hodnota buňky a_i je na všech pozicích i v diskrétních časových krocích aktualizována podle předem daných neměnných pravidel, která závisí na okolí buňky:

$$a_i^{(t+1)} = \phi[a_{i-r}^{(t)}, a_{i-r+1}^{(t)}, \ldots, a_{i+r}^{(t)}] \tag{4.4}$$

kde:

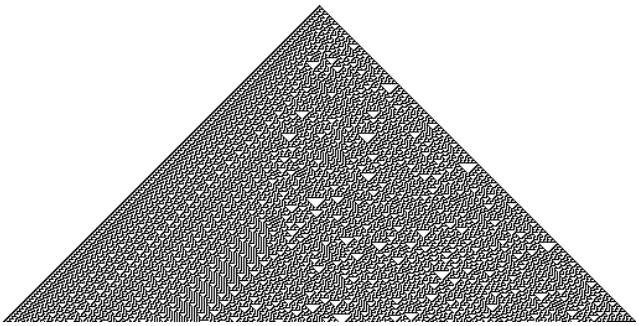
- $a_i^{(t)}$: hodnota buňky a_i v čase t
- ϕ: funkce počítající hodnotu buňky v dalším kroku pomocí jejího r-okoli

Už pro hodnoty $k = 2$ a $r = 1$ celkové chování automatu může být velice komplexní. Příkladem je pravidlo 30, které vytváří aperiodické chaotické vzory, které lze nalézt v přírodě například na ulitě homolice síťkované. Toto pravidlo je popsáno v tabulce 4.1. Vzory, které toto pravidlo generují, jsou zobrazeny v obrázku 4.7.

aktuální vzor	111	110	101	100	011	010	001	000
následující hodnota buňky	0	0	1	1	1	1	0	0

Tabulka 4.1: Zobrazení pravidel pro výpočet nových hodnot pro buňky. V prvním řádku jsou v políčkách aktuální hodnoty v buňce předcházející, v buňce pro kterou se počítá nová hodnota a v buňce následující. V druhém řádku je zobrazena hodnota, na kterou se má buňka přepsat. Hodnoty v této tabulce jsou známy jako pravidlo 30.

Druhů celulárních automatů existuje velké množství. Lišit se mohou například prostorem, ve kterém jsou buňky definované. Výše popsaný CA měl buňky definované v jednorozměrném poli, avšak tyto automaty mohou využívat výše definovanou mříž 4.1.3, mřížky



(a)

(b)

Obrázek 4.7: Dva obrázky zobrazující stejný vzor. Tento vzor je složen z řádků. Generování vzoru započalo výchozím horním řádkem a každý další řádek vznikl vygenerováním buněk pomocí pravidla 30 na základě hodnot z předchozího řádku.
(a) Vzor vygenerovaný celulárním automatem s pravidlem 30. Autor: Zhiming Wang, dostupné z: https://raw.githubusercontent.com/zmwangx/rule30/master/images/rule30.png
(b) Ulita homolice síťkované. Autor: Richard Ling, dostupné z: https://commons.wikimedia.org/w/index.php?curid=293495

které nejsou čtvercové, nebo existují automaty, které jsou definované nad obecnými grafy 3. Další rozdíly mezi automaty jsou dané tím, jaké využívají buňky okolí pro výpočet svého stavu. Toto okolí se může dokonce v průběhu času měnit. Dále může být do pravidel začleněna pravděpodobnost a stavy mohou nabývat spojitých hodnot.

Kapitola 5

Návrh řešení

5.1 Analýza požadavků

Vzhledem k různorodosti reálných systémů je důležité, aby framework nejen pokrýval široké spektrum standardních situací, ale aby také nabízel rozhraní, které umožní uživatelům integraci vlastních, specificky navržených modelů do testovacího prostředí. To umožní pokrýt i ty nejvíce specifické a unikátní požadavky na testování, které standardní nástroje a modely nemusí plně adresovat.

Tato práce se tedy zaměřuje na vytvoření robustního a flexibilního řešení, které poskytne uživatelům efektivní nástroje pro simulaci a testování širokého spektra IoT systémů, a to i v případech, kdy je nutné simulovat náhodné události a specifické scénáře.

5.1.1 Funkční požadavky

Centrální řízení simulace

Testovací prostředí je pomocí Patriotu typicky vytvořeno jako systém sestávající z více procesů. Popis událostí, které ovlivňují generovaná data, je ale potřeba definovat pouze na jednom místě, a to v definici testu. Dále by mělo být uživateli umožněno v průběhu testu s vytvořenou simulací interagovat (zasílat řídící signály, posouvat se v čase).

Schopnost distribuce dat k zařízením ve virtuální síti

Jak bylo popsáno v kapitole 3, framework PatrIoT poskytuje podporu pro vytváření virtuální sítě a nasazení chytrých zařízení v rámci této sítě jako konteinerizované aplikace. Je potřeba vytvořit mechanismus, který umožní propagovat informaci o změně generovaných dat do kontejnerů s chytrými zařízeními.

Základní řízení času

Uživatel by měl mít možnost po vytvoření simulačního modelu a následném spuštění možnost řídit simulační čas, a to ve dvou módech.

První mód je posouvání simulace do uživatelem stanovených časů. Po uplynutí simulace do jednoho ze stanovených časů by mělo dojít k předání běhu programu zpět do testovací funkce, která v požadovaném čase může otestovat stav testovaného systému.

Patriot framework podporuje i hybridní testování. Může tedy nastat případ, kdy bude testovaný systém, nebo testovací prostředí využívat fyzické zařízení. Druhý mód řízení času

je pro ty případy, kdy je potřeba, aby byl průběh simulace synchronizovaný s reálným časem. V tomto módu by mělo jít řídit čas minimálně tím způsobem, že bude umožněno nastavit výchozí čas, spustit simulaci, manuálně zastavit simulaci a nastavit délku simulace, po kterém bude automaticky ukončena.

Určení pozice zařízení

Framework by měl umožňovat definovat virtuální prostor, a to jak diskrétní, tak spojitý. Tento prostor bude sloužit pro simulace událostí ovlivňujících chytrá zařízení. Řešení taktéž musí poskytnout mechanismus umožňující zařízením generovat data, která budou spjata s konkrétní pozicí ve virtuálním prostoru.

Definice interakce mezi různými událostmi

V reálném světě může nastat více náhodných událostí souběžně. Souběžně mohou mít vzniklé události mezi sebou vztah, který ovlivňuje jejich chování. Příkladem může být lesní požár, který změní rychlost šíření, pokud přijde déšť. Tyto skutečnosti je třeba promítnout do výsledného řešení a poskytnout uživateli kromě možnosti definice různých událostí i možnost změny chování simulací události při vzniku jiných.

Determinismus

Data generovaná za pomoci tohoto rozšíření jsou využívána v testech. Z toho plyne, že stejně jako testy i generovaná data musí být deterministická, aby bylo možné reprodukovat případně nalezené chyby.

Ovlivnění simulace aktuátorem

Chytrá zařízení jsou ve frameworku PatrIoT dělena na aktuátory a senzory. Potřeba synchronizace a modulace generovaných dat různými senzory již byla v této práci popsána. V tomto případě jsou informace propagovány ve směru simulace – chytré zařízení. Zbývá adresovat potřebu propagace ve směru chytré zařízení – simulace. Konkrétně se jedná o přenos informace od aktuátoru do simulace. Tento přenos je potřeba, jelikož simulace vytváří virtuální prostředí kolem chytrých zařízení a aktuátory jsou dle definice zařízení, která své prostředí ovlivňují. V příkladu s ohněm může být aktuátor stabilní hasící zařízení. Při vzniklé události požáru se stabilní hasící zařízení aktivuje a požár uhasí. Tuto informaci je v potřeba propagovat do simulace, která stav požáru uchovává.

5.1.2 Nefunkční požadavky

Do nefunkčních požadavků spadají požadavky na výsledný systém, které souvisí s jeho technickou realizací a nejsou definovány pomocí funkčních požadavků. Tyto požadavky kladou omezení na vyvíjený produkt. Různé organizace definují vlastní požadavky na svůj software různě a přes některé snahy o vytvoření standardu na popis nefunkčních požadavků, neexistuje žádný obecně přijímaný.[22].

Následuje popis nefunkčních požadavků, které byly identifikovány jako klíčové pro tuto práci. Při analýze nefunkčních požadavků byla většina přejata z požadavků kladených na framework PatrIoT. Tyto požadavky jsou zmíněny v podkapitole 3.1. K těmto převzatým požadavkům byly následně přidány požadavky specifické pro toto rozšíření.

Kompatibilita se zbytkem projektu

Nejnovější vydaná verze frameworku PatrIoT je v době psaní této práce kompletně implementována v jazyce Java 8. Rozšíření stávající implementace si vynucuje, že i nově přidaný kód musí být v jazyce Java, avšak verze 8 je v dnešní době považována za zastaralou.

Další vydání frameworku by již mělo být kompatibilní s verzí jazyka Java 17. Tento fakt by měla respektovat i tato práce a nově přidaný kód by měl odpovídat požadované verzi. Tento fakt umožní při vývoji využívat benefity novější verze jako je použití sealed modifikátoru, automatické odvození lokálního datového typu, rozšíření výrazu switch, zvýšení výkonu aplikace a další [2].

Aby se během vývoje předešlo problémům s kompilací nekompatibilních modulů v různých verzích jazyka, bude potřeba využívat lokálně sestavené balíčky, které jsou založeny na doposud nevydané vývojářské verzi kódu.

Rozšířitelnost

V rámci návrhu a implementace softwarových systémů je jedním z klíčových požadavků zajištění rozšiřitelnosti, znovupoužitelnosti a obecně dodržování zásad čistého kódu. Tyto vlastnosti jsou zásadní pro dlouhodobou udržitelnost a evoluci softwaru, zejména v kontextu tvorby frameworku, kde se přímo očekává, že si část uživatelů bude základní třídy rozšiřovat pro specifické potřeby.

Aby bylo dosaženo těchto cílů, je vhodné se řídit osvědčenými zásadami pro návrh softwaru. Návrhové principy, které se staly pro objektově orientované jazyky de facto standardem, jsou známé pod akronymem SOLID. Jedná se o těchto pět principů, které byly popsány Robertem C. Martinem v knize Design Principles and Design Patterns [19]:

1. S - Single Responsibility (Princip jedné odpovědnosti): Každá třída by měla mít jedinou odpovědnost nebo důvod ke změně. Dodržování tohoto principu zajišťuje, že změny v jedné části systému nebudou mít nečekané důsledky v jiných částech.

2. O - Open-Closed Principle (Princip otevřenosti a uzavřenosti): Třídy by měly být otevřené pro rozšíření, ale uzavřené pro změny. To znamená, že chování třídy lze rozšířit bez nutnosti měnit její stávající kód. Tento princip podporuje rozšiřitelnost a umožňuje přidávání nových funkcionalit. Tím pádem je pro tuto práci klíčový.

3. L - Liskov Substitution (Liskovové princip zastupitelnosti): Objekty by měly být zaměnitelné za instance jejich podtříd tak, aby požadavky kladené na rozhraní původní třídy byly splněny i pro podtřídy. To znamená, že podtřídy musí být schopny plně zastoupit své nadtřídy. Dodržování tohoto principu zajišťuje, že nové třídy mohou být snadno integrovány do stávajících systémů.

4. I - Interface Segregation (Princip segregace rozhraní): Místo jednoho univerzálního rozhraní by měla být vytvořena specifická rozhraní pro každou třídu. Tímto způsobem jsou klienti nuceni implementovat pouze metody, které skutečně potřebují. Tento princip zlepšuje znovupoužitelnost a snižuje závislosti mezi částmi systému.

5. D - Dependency Inversion (Princip obrácení závislosti): Tento princip doporučuje závislost na abstrakcích, nikoli na konkrétních implementacích. To znamená, že vyšší vrstvy systému by neměly záviset na nižších vrstvách, ale obě by měly záviset na abstrakcích. Tím se zvyšuje flexibilita a umožňuje snadná výměna komponent.

V praxi se tyto principy aplikují tak, že pokud programátor přesně neví jakou architekturu pro daný program zvolit, napíše naivní implementaci. Následně iterativně kontroluje, jestli navržené třídy splňují všechny návrhové principy. Pokud některá třída porušuje jeden z principů, aplikuje obecné pravidlo pro odstranění konkrétního porušení a přejde na další iteraci.

Vývoj pod open-source licencí Apache 2.0

Apache License 2.0 (ALv2) je permisivní open-source licence, která umožňuje s minimálními omezeními kopírovat, modifikovat a dále distribuovat software pod jinou licencí. Při vývoji pod touto licencí je třeba dbát na to, aby byl v kořenovém adresáři umístěn kompletní text ALv2, a na začátku každého zdrojového kódu by mělo být oznámení o licenci daného kódu.

Licence pro open-source software se v základu dělí na dvě kategorie – permisivní a copyleft. Permisivní licence jsou benevolentní. Umožňují aby byl kód kopírován, modifikován, využit pro komereční využití a dále šířen pod restriktivnějšími licencemi. ALv2 je licence, která bývá označována jako permisivní licence. [17]

Copyleft licence typicky vyžadují, aby projekt, který využívá dílo pod copyleft licencí, byl vydán pod stejnou copyleft licencí. Jelikož je potřeba zachovat u frameworku PatrIoT původní ALv2 licencí, není možné využívat knihovny a nástroje pod copyleft licencemi.

Po analýze běžně používaných licencí se ukázalo užití softwaru pod následujícími licencemi jako kompatibilní s projektem PatrIoT:

- ALv2
- LGPL-2.1
- LGPL-3.0
- MIT
- BSD 3-clause "New"or "Revised"license
- Public Domain

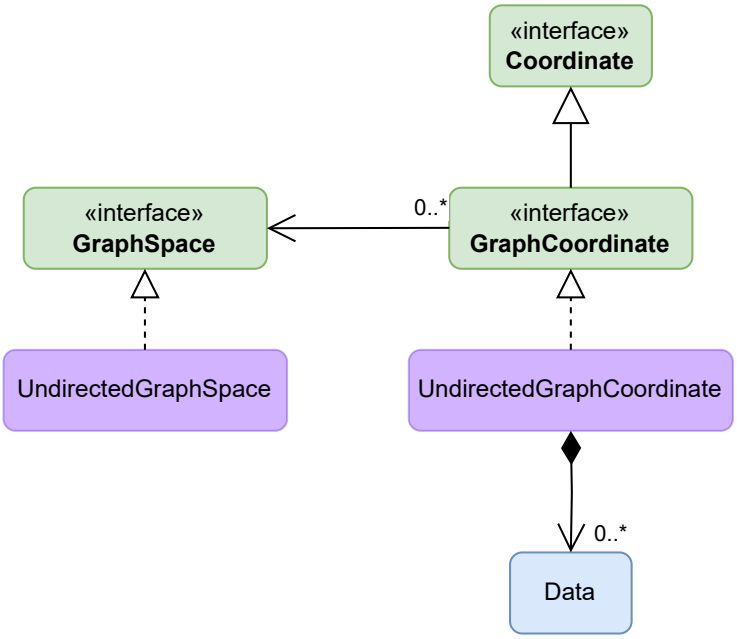
open-source

U projektu pod open-source licencí je třeba dbát na dodržení závazku, že zdrojový kód je veřejně dostupný. Tohoto požadavku může být docíleno nahráním zdrojového kódu do veřejně přístupného repozitáře na platformě GitHub[1], který bude obsahovat změny v rámci této práce. Následně mohou být tyto změny sloučeny s hlavní vývojovou větví projektu projektu PatrIoT[2].

5.2 Návrh souřadnic

Pro reprezentaci objektů v prostoru je v rámci této práce navržena sada tříd reprezentující souřadnice v různých prostorech. Hierarchie tříd odpovídá hierarchii souřadnicových systémů popsaných v sekci 4.1. Všechny navržené třídy implementují rozhraní `Coordinates`, které určuje že se jedná o metrické souřadnice – tedy že implementuje metodu distance.

[1] https://github.com/
[2] https://github.com/PatrIoT-Framework



Obrázek 5.1: Diagram tříd zobrazující vztahy mezi třídami implementující diskrétní grafové souřadnice.

Jako reprezentace spojitého prostoru byl zvolen kartézský prostor. Tím pádem bylo možné pro tento prostor určit společné vlastnosti pro všechny třídy reprezentující souřadnice nad tímto prostorem a zároveň zaručit, že různé souřadnice nad stejným kartézským prostorem jsou mezi sebou převoditelné. Pro zaručení těchto vlastností bylo vytvořeno rozhraní `CartesianCoordinate`. Každá třída, která má reprezentovat souřadnici kartézského souřadnicového systému, by ve frameworku Patriot měla toto rozhraní implementovat. Příkladem různých souřadnic nad kartézským prostorem můžou být standardní kartézské souřadnice (vektory čísel), polární souřadnice, sférické souřadnice a další.

5.2.1 Diskrétní grafové souřadnice

Jako reprezentant diskrétních souřadnic byl zvolen neorientovaný graf. Na rozdíl od kartézských souřadnic není topologie prostoru implicitně určena zvolenými souřadnicemi, proto je potřeba kromě objektů reprezentujících souřadnice navrhnout i objekt, který definuje prostor (v tomto případě neorientovaný graf). Pro určení vzdálenosti a dalších operací nad grafovým prostorem je potřeba, aby souřadnice reprezentující bod v prostoru (vrchol grafu) měly referenci na grafový prostor. Vztahy mezi těmito třídami jsou zobrazeny v diagramu tříd 5.1.

Třída UndirectedGraphSpace

Tato třída slouží jako wrapper kolem neorientovaného grafu, který vytváří rozhraní pro práci se souřadnicemi. Tato třída by měla poskytnout návrhový vzor stavitel pro usnadnění

sestavení prostoru, dále by měla poskytnout metodu pro výpočet vzdálenosti dvou souřadnic a také metody pro získání instancí souřadnic.

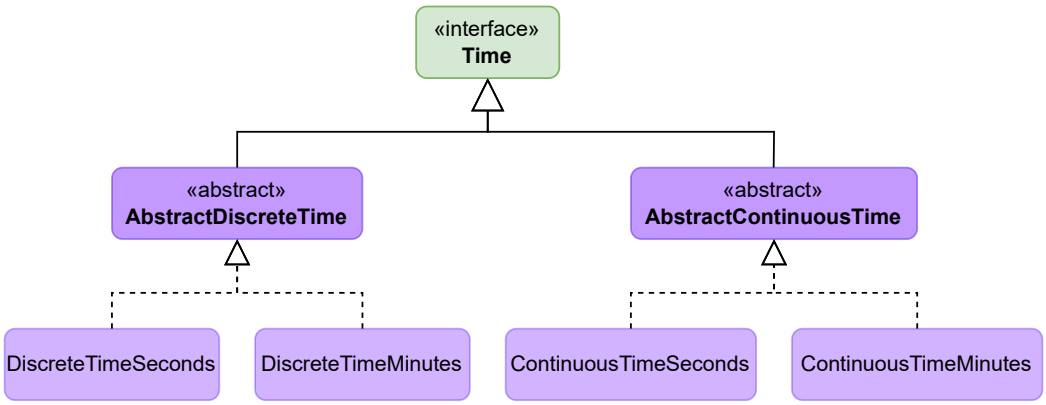
Začlenění metody pro získání vzdálenosti dvou vrcholů umožní tento prostor považovat za metrický. Vzdáleností je myšlena délka nejkratší cesty mezi dvěma vrcholy. Tato metoda vyžaduje, aby třída `UndirectedGraphSpace` obsahovala pouze souvislé grafy.

Třída UndirectedGraphCoordinate

Třída reprezentující souřadnici pro instanci grafového prostoru. Tyto souřadnice zároveň obsahují data, která jsou v daném vrcholu obsažena.

5.3 Návrh Tříd reprezentujících čas

Pro práci s časem byl navržen balíček `Time`, který obsahuje hierarchicky rozdělenou sadu tříd pro reprezentaci času. Hierarchie těchto tříd je zobrazena v diagramu tříd5.2 Nejmenší simulační krok byl zvolen jedna milisekunda. Tento fakt bylo potřeba promítnout i do objektů reprezentujících rozhraní, která musí implementovat. Každá reprezentace času tak obsahuje metodu `long getMillis()`, která vrací čas převedený do milisekund. Převod veškerých časů na společnou reprezentaci umožňuje operace jako sčítání, odečítání a porovnávání mezi různými reprezentacemi času. Rovnost mezi dvěma časy je stanovena na základě rovnosti času v milisekundách.



Obrázek 5.2: Diagram tříd pro reprezentaci času.

Hlavní dělení času nastává u abstraktních tříd `AbstractDiscreteTime` a `AbstractContinuousTime`, které jsou základem pro třídy reprezentující diskrétní a spojitý čas. Výhoda spojitého času je v tom, že ve chvíli, kdy se během simulace ukáže, že zvolená časová jednotka je příliš velká (sekunda), je možné časové kroky dělit až k nejmenšímu simulačnímu kroku (tisícina sekundy). Tyto třídy si uchovávají informace o tom, jakou jednotku času reprezentují a kolik milisekund daná jednotka trvá. U spojitého času jsou prováděné operace udržovány ve spojité doméně a převod do diskrétních milisekund je proveden až na vyžádání.

Pro přímé využití byly navrženy dvě třídy rozšiřující výše popsané abstraktní třídy. Jak pro diskrétní čas, tak pro spojitý byly vytvořeny třídy reprezentující čas v sekundách a v minutách. V případě zájmu si uživatel může dodefinovat další. Pro vytvoření nové třídy

stačí rozšířit abstraktní třídu a implementovat dvě metody. První vrací počet milisekund v jednotce a druhá vrací řetězcový literál, který udává značku nebo popis jednotky.

5.4 Návrh úpravy CoAP serveru

Jak bylo popsáno v sekci 3.5.1, způsob, jak v současné situaci framework vytváření model reálného světa, se kterým interagují chytré zařízení je skrze objekt `DataFeed` pro sensory a `StateMachine` pro Aktuátory.

Vytvořená simulace bude též vytvářet okolí zařízení a skrze objekty `DataFeed` půjde informace o stavu simulovaného prostředí skrze senzor do testovacího prostředí. Naopak aktuátor jakožto zařízení ovlivňující simulované prostředí musí informovat skrze stavový automat o svých akcích simulaci.

Simulace, ve které se budou odehrávat různé události ovlivňující zařízení, potřebuje mechanismus, kterým bude možné zaměnit `DataFeed` u senzoru a zjistit změny stavů u `StateMachine`.

Při analýze v podsekci 3.5.4 byl identifikován způsob, jakým je v dosavadním stavu frameworku možné interagovat s chytrými zařízeními nasazenými ve virtuální síti. Interakce s těmito zařízeními je možná skrze servisní CoAP server. Tento server aktuálně neumožňuje výměnu data-feedu a získání akcí provedených aktuátorem, avšak má vhodnou architekturu, která poskytuje solidní základ. Pro implementaci požadované funkcionality postačí rozšíření bez dramatických zásahů do původních struktur.

Na obrázku 5.3 kroky 1–3 zobrazují proces výměny data-feedu. Simulace, která při vzniklé události potřebuje změnit data generovaná senzorem, skrze CoAP klienta zašle požadavek na CoAP server, který je spuštěn společně se senzorem v java virtual machine v kontejneru virtuální sítě. Tento požadavek obsahuje v těle zprávy serializovaný data-feed. CoAP server následně spustí obslužnou rutinu, deserializuje přijatý data-feed a vymění u příslušného senzoru původní data-feed za nový.

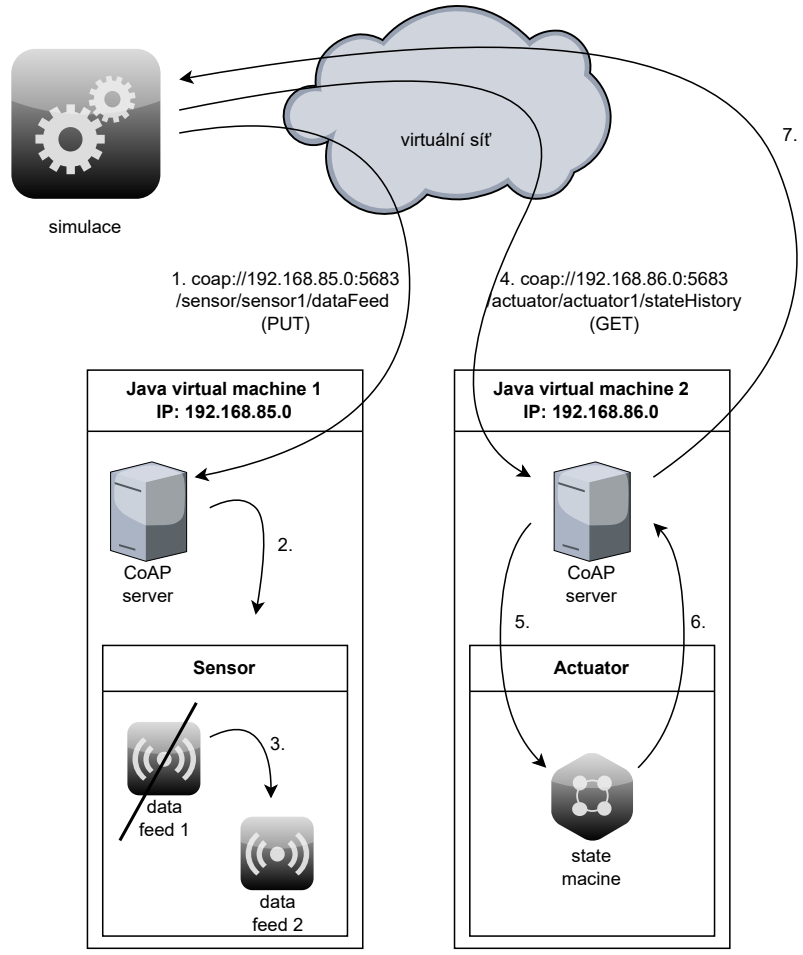
Kroky 4–7 pak ukazují proces získání akcí provedených aktuátorem. Kroky jsou podobné jako u procesu výměny data-feedu, avšak při požadavku je tělo zprávy prázdné, obslužná rutina serveru získá akce ze stavového automatu požadovaného aktuátoru a tyto akce jsou poté poslány v těle CoAP zprávy nazpět.

5.4.1 Úprava rozhraní servisního serveru

Aby byl výše popsaný mechanismus provedítelný, je potřeba rozšířit stávající rozhraní CoAP serveru o nové koncové body. Schéma CoAP rozhraní a nové navržené koncové body jsou zobrazeny na obrázku 5.4

Zdroj DataFeed

Tento zdroj by měl být dostupný pod URI `/sensor/{sensorLabel}/dataFeed` a vytváří jeden segment logicky členěné cesty pro identifikaci konkrétních data-feedů a jejich dalších zdrojů. Tento zdroj implementuje metodu GET, která vrací seznam všech dostupných datafeedů pro daný senzor. Z tohoto seznamu je též možné získat seznam identifikátorů data-feedů, které jsou potřeba pro vytvoření požadavku na konkrétní datafeed senzoru. Tato metoda je zde přidána především pro ulehčení práce s CoAP serverem, jelikož by bylo technicky možné získat seznam dostupných data-feedů i pomocí service discovery procedury, která je v CoAP serverech implementována [24].



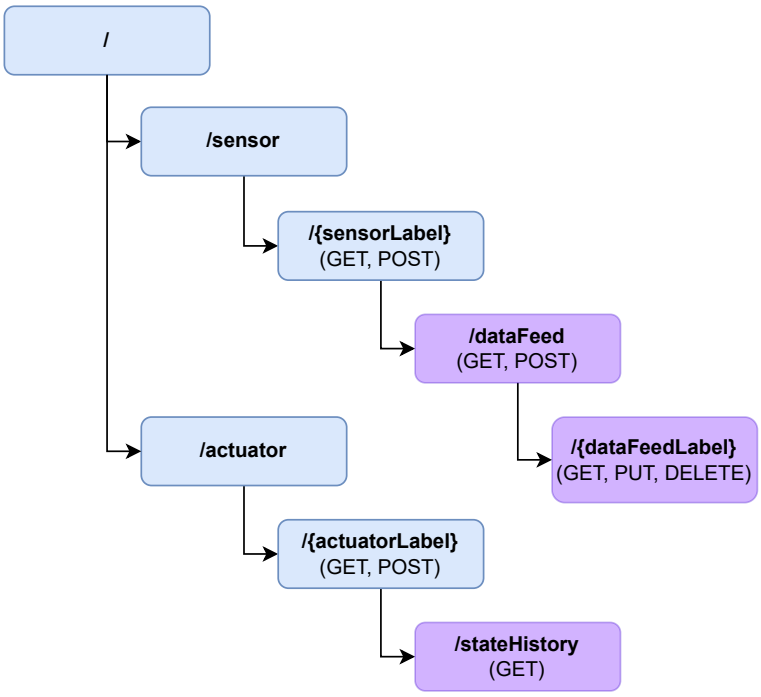
Obrázek 5.3: Nastínění mechanismu výměny data-feedu a získání akcí provedených aktuátorem.

Zdroj dataFeed/label

Zdroj dostupný pod URI-cestou `/sensor/{sensorLabel}/dataFeed/{dataFeedLabel}`, který pomocí metod GET, POST, PUT a DELETE, zajišťuje získání, vytvoření, úpravu a smazání konkrétních data-feedů senzoru.

Zdroj StateHistory

Zdroj dostupný pod URI-cestou `/actuator/{actuatorLabel}/stateHistory`, který pomocí metody GET poskytuje seznam řetězcových literálů. Řetězcové literály seznamu reprezentují stavy, ve kterých se daný aktuátor nacházel od posledního dotazu až po aktuální stav. Tato metoda není bezpečná ani idempotentní a návrh této metody vychází z předpokladu, že existuje pouze jeden klient, který má za daný aktuátor zodpovědnost.



Obrázek 5.4: Schématický přehled koncových koncových bodů CoAP servisního serveru. Modré panely reprezentují původní koncové body. Fialové panely reprezentují návrh na přidání nových koncových bodů. U koncových bodů jsou též zaznamenány metody, které obsluhují.

5.4.2 Návrh tříd implementujících rozhraní servisního serveru

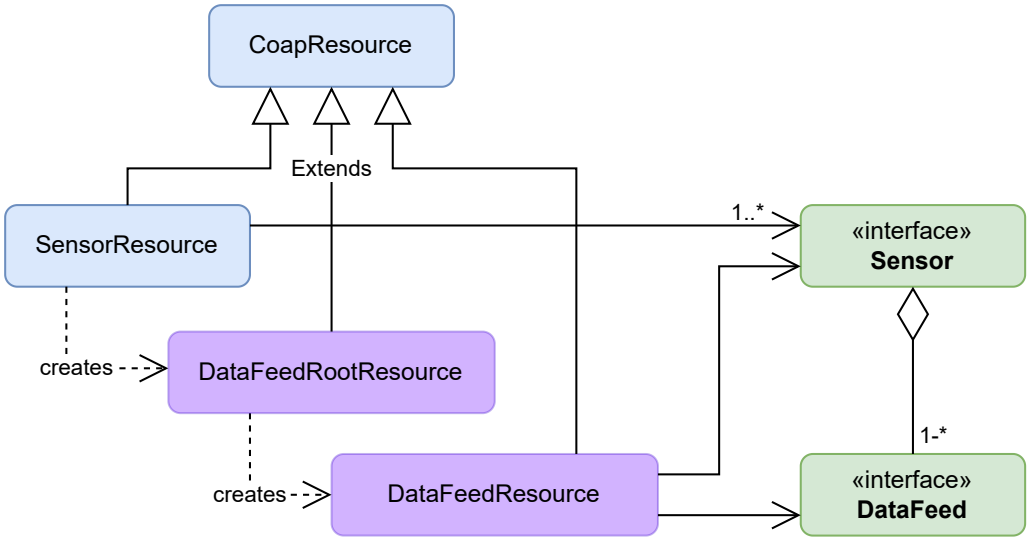
Coap URI schéma zobrazené na obrázku 5.4 vytváří stromovou strukturu. Třídy obsluhující jednotlivé koncové body jsou navrženy tak, aby danou strukturu reflektovaly a umožňovaly její snadné vytvoření.

Instance tříd rozšiřující třídu CoapResource (dále už jen zdroje) jsou zodpovědné za zpracování CoAP požadavků na URI adrese, pro kterou byly vytvořeny. Každý zdroj je dále zodpovědný za vytvoření a registraci do CoAP serveru všech dalších zdrojů, které pod něj hierarchicky spadají.

Ze zvolené implementace CoAP serveru vyplývá, že všechny třídy vytvářející zdroj serveru, musí dědit od třídy CoapResource knihovny Californium [12].

Třída DataFeedRootResouce

Tato třída je závislá na instanci Třídy Sensor. Aby byl splněn požadavek na dodržení principů SOLID 5.1.2, konkrétně pak princip obrácení závislostí, je potřeba instanci implementující rozhraní Sensor předat v konstruktoru. Tento vzor vytváří závislost na rozhraní místo konkrétní třídy a je známý jako "dependency injection"[23]. Vztah této třídy k rozhraní Sensor a dalším třídám je zobrazen v diagramu tříd 5.5.



Obrázek 5.5: Diagram tříd zobrazující vztahy mezi nově přidanými třídami (DataFeedRootResource a DataFeedResource) a původním řešením.

Třída DataFeedResouce

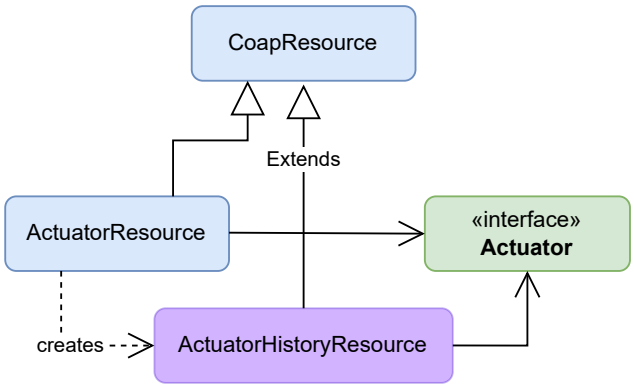
Tento zdroj je zodpovědný za obsluhu všech URI, které identifikují konkrétní data-feedy pomocí unikátního identifikátoru. Tato třída obdobně jako DataFeedRootResouce vytváří závislost na Instanci Sensor, avšak dále potřebuje i instanci DataFeed, pomocí které identifikuje, který jeden z více data-feedů senzoru daný zdroj obsluhuje. Instance třídy DataFeed je též poskytnuta v konstruktoru. Tato skutečnost je zobrazena v diagramu tříd 5.5. V tomto diagramu je též patrný proces vytváření instancí třídy DataFeedResource, který je zaznamenán vazbami "creates". SensorResource (reprezentující celý senzor) je zodpovědný za vytvoření třídy DataFeedRootResource, a ta je dále zodpovědná za vytvoření všech instancí třídy DafaFeedResource.

Třída ActuatorHistoryResource

Tento zdroj je zodpovědný za poskytování historie Aktuátoru. Tento zdroj je vytvořen zdrojem ActuatorResource. Je závislý na instanci Actutator a stejně jako u předchozích zdrojů je Actuator poskytnut konstruktorem. Při návrhu bylo zvažováno, zdali má být tento zdroj závislý na instanci Actuator, nebo přímo instanci StateMachine, která ve realizuje pro Actuator změny stavu. Třída Actuator se ukázala jako správná volba, jelikož mezi Actuator a StateMachie není pevná kompoziční vazba, a tudíž není obecně zaručeno, že StateMachine po celou doubu běhu bude odpovídat stavu Aktuátoru. Začlenění tohoto zdroje je zobrazeno v diagramu tříd 5.6

5.5 Návrh simulátoru

Pro samotný simulátor byl zvolen návrhový vzor "Data Bus". Tento návrhový vzor je vhodný v situacích, kdy je potřeba sdílet data napříč různými entitami bez potřeby explicitního pá-



Obrázek 5.6: Diagram tříd zobrazující vztahy mezi nově přidanou třídou ActuatorHistoryResource a původním řešením.

rování mezi těmito entitami. Data Bus vytváří komunikační rozhraní, které umožňuje komunikaci mezi účastníky, kteří v době svého vývoje nemuseli mít o druhém žádné informace [11].

Hlavní myšlenka tohoto vzoru je, že existuje objekt reprezentující sběrnici, po které všichni aktéři simulace komunikují. Komunikace probíhá na principu *publish-subscribe*, tedy poskytování dat pod určitými tématy a asynchronní přijímání všemi objekty, kteří si k danému tématu přihlásili odběr.

5.5.1 Stadard High Level Architecture

Na návrhovém vzoru Data Bus je založena i sada standardů *High Level Architecture* (HLA). Standardy HLA popisují základní komponenty systému pro distribuovanou simulaci komplexních systémů. Vývojáři mají díky těmto standardům možnost strukturovat svůj simulační model a vytvořit k němu rozhraní tak, aby byl znovupoužitelný, interoperabilní a použitelný ve větší distribuované simulaci [13]. Následující podsekce je založena na informacích ze standaru HLA [4].

Základní logické prvky HLA simulace

HLA přichází s vlastními pojmy pro popis architektury distribuované simulace.

Federát je simulační systém, který má v rámci HLA standardu definované rozhraní. Chování tohoto systému musí být definováno dokumentem, který splňuje standard Object Model Template (OMT). OMT je klíčový prvek, který zajišťuje výše zmiňovanou znovupoužitelnost a interoperabilitu.

Runtime Infrastructure (RTI) – Software, který tvoří komunikační kanál mezi jednotlivými federáty. Standard HLA definuje sadu služeb, které musí RTI implementovat. Propojením více federátů pomocí RTI vzniká Federace. Komunikace skrze RTI je založena na návrhovém vzoru publish-subscribe. Základní dva prvky, ke kterým mohou federáty přihlásit odběr nebo je sami publikovat, jsou atributy objektů a interakce. RTI rozhraní také podporuje posun v čase.

Federate Object Model (FOM) – Specifikace informací sdílených pomocí RTI. FOM obsahuje například definice tříd s jejich jejich atributy a interakcí s jejich atributy.

Federace je vytvořena ve chvíli, kdy ji jeden z federátů pomocí RTI vytvoří. Federáty se od této chvíle mohou k dané federaci připojit a začít mezi sebou interagovat.

High level architecture jako základ simulace

Začlenění standardů HLA se při návrhu jevilo jako způsob, jak zajistit možnost využití již existujících simulačních modelů. Zároveň tento standard nabízel využití některé z implementací jako simulátoru pro navrhované rozšíření frameworku, které by navíc nabízelo možnost distribuované a výpočetně náročné simulace. Bohužel se v průběhu této práce ukázalo, že jediná dostupná implementace (Portico[3]), jejíž licence by splňovala podmínky kladené na projekt PatrIoT (5.1.2), není aktuálně v technickém stavu, kdy by se dala použít. Dále se HLA standardy ukázaly jako natolik komplikované, že by interakce uživatele, který má zájem vytvořit testy pro IoT systém, s tímto modulem byla neúměrně náročná.

5.5.2 Vlastní implementace simulátoru

V této práci bude na klienty sběrnice pohlíženo jako na různé simulační modely, jež reprezentují určitou část komplexní simulace tvořené dílčími simulacemi. Po sběrnici budou posílána data o událostech, které dílčí simulace způsobily. Na tyto vzniklé události mohou reagovat jak ostatní dílčí simulace, které si k daným událostem přihlásily odběr, tak přímo zařízení (o mechanismu distribuce událostí k zařízením pojednává podsekce **??**). Systém sběrnice a jejich klientů je založen na třech rozhraních – `EventBus`, `EventDistributorService` a `EventProcessor`, které budou dále popsány. Definice dílčích simulačních modelů a zpráv posílaných po sběrnici je ponechána na uživateli, avšak tento návrh počítá s tím, že se vytvoří balíčky s předdefinovanými simulačními modely pro specifické domény, které budou moct uživatelé využít.

Rozhraní EventDistributorService

Zodpovědnost EventDistributorService je poskytování služby pro dílčí simulace pro asynchronní zasílání zpráv pod určitým tématem od producenta ke všem účastníkům, kteří si k danému tématu přihlásili odběr. Očekává se, že toto rozhraní bude používáno pro synchronizaci dílčích simulaci. Jednou z důležitých částí simulace je posouvání času. Rozhraní EventDistributorService tuto skutečnost zohledňuje poskytnutím funkcí pro zasílání zpráv, u kterých jde specifikovat čas doručení, a poskytnutím funkce pro zjištění simulačního času. Stěžejní funkce rozhraní EventDistributorService jsou následující:

- `void subscribe(Simulation simulation, String topic)`: Touto funkcí se dílčí simulace přihlásí k odběru zpráv pod daným tématem.
- `void publish(Data message, String topic)`: Funkce k poslání zprávy všem registrovaným příjemcům v nejbližším dalším simulačním kroku.
- `void registerAwake(Simulation simulation, Time time)`: Dílčí simulace se pomocí této funkce registrují k provedení akce ve specifikovaném simulačním čase.
- `Time getTime()`: Funkce, pro zjištění aktuálního simulačního času.

[3] https://github.com/openlvc/portico

Rozhraní obsahuje i další funkce, ty ovšem nepřidávají novou funkcionalitu. Stejného efektu jde docílit kombinací použití základních, výše popsaných funkcí. Například funkce `void registerRecurringAwake(Simulation simulation, Time interval)` jde nahradit voláním funkce `registerAwake(Simulation simulation, Time time)` při každém probuzení. Tyto funkce jsou do rozhraní zahrnuty pouze pro usnadnění práce uživatele, který interaguje s tímto rozhraním.

Rozhraní EventDistributorClient

Toto rozhraní je určeno klientním třídám, které chtějí využívat `EventDistributorService`. Třídy implementující toto rozhraní v rámci projektu PatrIoT reprezentují dílčí simulace, které je potřeba synchronizovat s ostatními. Rozhraní obsahuje tři funkce, a to:

- `void init()`: Tato Funkce je volána před spuštěním simulace, v jejím těle se může dílčí simulace přihlásit k odběru témat a naplánovat si první probuzení.
- `void receive(Data message, String topic)`: Tato funkce je zvolána v moment, kdy některá z dílčích simulací poslala zprávu, ke které má instance implementující tuto funkci přihlášen odběr. V těle této zprávy by mělo dojít ke zpracování zprávy. Volání metody `Publish` nad `EventDistributorService` je v této metodě spjato s rizikem vytvoření nekonečného cyklu reakcí, proto je doporučováno, aby byla volaná pouze z metody `void awake()`.
- `void awake()`: Tato funkce je volána v simulačním čase, ve kterém se klient zaregistroval k probuzení. V těle této funkce dílčí simulace mohou reagovat na zprávy které byly přijaty od posledního probuzení, například posláním nových zpráv. V těle této funkce se též mohou registrovat další probuzení.

Rozhraní EventBus

Konceptuální model pro `EventBus` si interně udržuje informaci o simulačním čase. Každá poslaná zpráva má určené, v jakém čase se má doručit. Invariantem pro tuto třídu je, že se posouvá v simulačním čase stále k vyšším hodnotám. Nová hodnota času je určena jako nejnižší hodnota simulačního času z naplánovaných událostí, která je vyšší než aktuální simulační čas. Jedná se o kalendář řízený událostmi. Posloupnost akcí při zpracovávání událostí v simulačním čase je následující:

1. Probuzení všech dílčích simulací, které se přihlásily k pravidelnému buzení. Aktuální čas odpovídá jejich budícímu intervalu.
2. Probuzení všech dílčích simulací, které se přihlásili k buzení v aktuálním čase.
3. Doručení všech zpráv pro aktuální čas.

Provedení těchto akcí bude dále v této práci označováno jako simulační krok.

Rozhraní `EventBus` rozšiřuje `EventDistributorService` o funkce pro řízení kroků. Umožňuje provést jeden krok, kroky do určitého času a všechny kroky. Následuje výčet funkcí a popis jejich chování.

- `Time getTime()`: Vrací simulační čas na sběrnici. V případě pozastavené sběrnice vrací poslední zpracovaný simulační čas.

- `Time getNextStepTime()`: Vrací simulační čas, ve kterém budou zpracovávány následující události.
- `tick()`: Posune simulační čas a zpracuje v něm všechny události (provede krok simulace).
- `run()`: Zpracuje veškeré události pro všechny simulační časy, ve kterých byla naplánovaná alespoň jedna událost.
- `boolean runUntil(Time until)` Zpracuje veškeré události do zadaného času (včetně).

Sekvenční diagram 5.7 zobrazuje volání mezi dílčími simulacemi a sběrnicí. Tento diagram zobrazuje scénář, kdy se k EventBusu zaregistrují dva klienti. První se přihlásí k probuzení ve třetí vteřině a zaregistruje si odběr zpráv s tématem "topic1". Druhý si registruje probuzení v čase 0. Následně, když Conductor zažádá EventBus o zpracování dalšího okamžiku (volání metody `tick()`) je v čase 0 probuzen druhý klient, který pošle zprávu s tématem "topic1". Ta je následně doručena prvnímu klientovi. Po druhém posunu času na sběrnici je probuzen první klient v čase 3.

5.5.3 Conductor

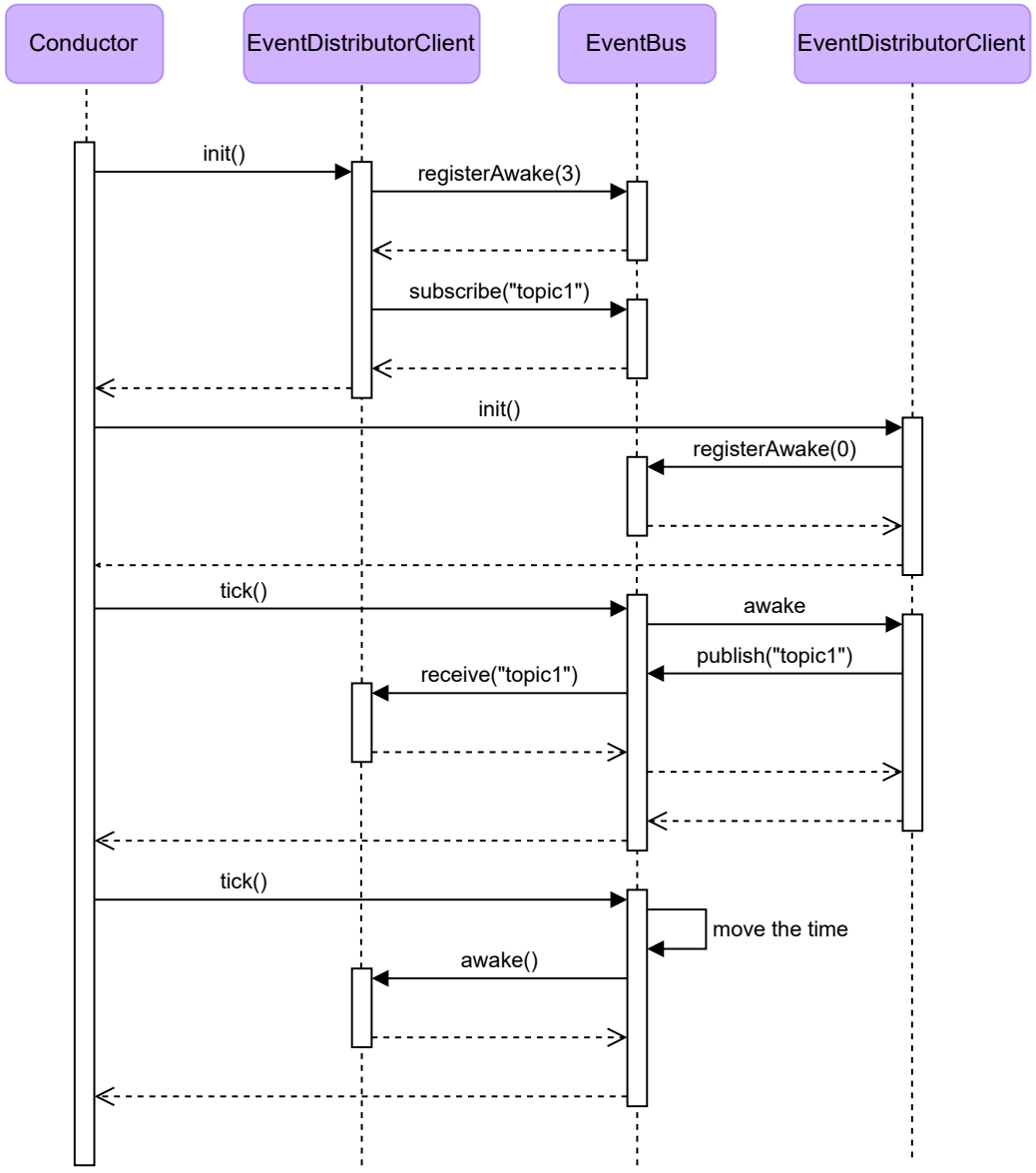
V předchozích podsekcích byly popsány třídy, které svým spojením vytváří simulační model. Pro spojení dílčích simulací je potřeba dílčím simulacím poskytnout stejnou instanci třídy `EventDistributorServer`, která slouží jako služba pro komunikaci. Dále je před spuštěním potřeba dílčí simulace inicializovat vůči dané sběrnici. Třída `Conductor` popsaná v této podsekci slouží jako rozhraní pro uživatele, skrze které je možné uživatelsky přívětivým způsobem simulaci sestavit a následně ji řídit. Řízením je myšleno spuštění sestavené simulace, pozastavení a posouvání času. K tomu poskytuje tato třída následující tři funkce:

- `void run()`: Pokračuje v simulaci od aktuálního simulačního času, dokud v kalendáři existují naplánované události.
- `void runFor(Time duration)`: Pokračuje v simulaci od aktuálního simulačního času po časový úsek, který je předaný v parametru.
- `void runUntil(Time endTime)`: Pokračuje v simulaci od aktuálního času po čas předaný v parametru (včetně).

Doba běhu těchto funkcí v reálném čase bude co nejkratší. Doba záleží na výkonnosti systému a náročnosti simulace.

Třída `Conductor` rovněž reflektuje požadavek na synchronizaci simulačního času s časem reálným, jenž je popsaný v podsekci 5.1.1. `Conductor` Pro následující funkce platí, že intervaly mezi událostmi v simulačním čase odpovídají intervalům mezi událostmi v reálném čase.

- `void runRealTime()`: Pokračuje v simulaci od aktuálního simulačního času, dokud v kalendáři existují naplánované události.
- `void runRealTimeFor(Time duration)`: Pokračuje v simulaci od aktuálního simulačního času po časový úsek, který je předaný v parametru.
- `void runRealTimeUntil(Time endTime)`: Pokračuje v simulaci od aktuálního času po čas předaný v parametru (včetně).



Obrázek 5.7: Sekvenční diagram zobrazující průběh simulace během prvních dvou časových oken.

Ve spojitosti s funkcemi pro real-time simulaci jsou k dispozici ještě další dvě, které dále rozšiřují možnosti uživatele:

- `void pause()`: Po dokončení zpracovávání aktuálního času pozastaví simulaci. Po přerušení je možné v simulaci pokračovat voláním libovolné z předchozích funkcí.
- `boolean isRunning()`: Funkce pro zjištění, zdali je simulace právě v běhu.

Může nastat, že reálná doba výpočtu všech událostí v některém čase přesáhne reálnou dobu vyčleněnou pro daný časový okamžik, v ten moment se simulační čas začne opožďovat za reálným a `Conductor` o vzniklé situaci vypíše varování.

Kapitola 6

Implementace

Tato kapitola se zaměřuje na praktický aspekt rozšíření frameworku PatrIoT. Nejprve popisuje, jaké kroky bylo potřeba učinit před samotným zahájením implementace nové funkcionality a identifikuje, které verze kódu byly použity jako základ pro rozšíření. Hlavní část této kapitoly se zabývá popisem použitých knihoven, implementací navržených tříd a konkrétními příklady kódu, které řeší zásadní funkcionalitu.

6.1 Praktické aspekty implementace

Verzování probíhalo pomocí nástroje Git [9]. Zdrojový kód frameworku je dostupný na platformě GitHub[1] v několika repozitářích. Tato práce zasahuje do třech stávajících repozitářů a jeden další vytváří. Aby bylo možné změny provedené v jednom modulu propagovat do dalších, byly vytvořeny privátní kopie repozitářů, které byly sestavovány aplikací pro sestavování a verzování balíčků Maven [10]. Nově vzniklé balíčky byly označeny novou verzí `5.0.0-SNAPSHOT-DP`.

Repozitáře obsahují větev `master`, jejíž poslední commit odpovídá poslední vydané verzi frameworku. Dále repozitáře obsahují větev `devel`, ve které se provádí změny, které se při dalším vydání přidají do hlavní větve. Vzhledem k požadavku na vývoj v jazyce Java verze 17 bylo třeba změny přidávat do větve, která je postavena na vývojové větvi a přidává změny, které posouvají verzi Javy v projektu z verze 8 na verzi 17.

6.2 Rozšíření servisního CoAP serveru.

Pro vytvoření CoAP serveru je použita knihovna *Eclipse Californium* Tato knihovna poskytuje open-source knihovnu pro iplementaci serveru splňující standard RFC7252 pro CoAP Protokol [14].

Pro vytvoření nových URI cest pro obsluhu DataFeedů a Aktuátorů navržených v sekci 5.4, bylo třeba vytvořit nové třídy reprezentující zdroje na serveru. U třídy, která má reprezentovat tyto zdroje je nezbytné, aby dědila od třídy `org.eclipse.californium.core.server.resources.Resource` a v konstruktoru volala rodičovský konstruktor, kterému je předána cesta, pod kterou má být daný zdroj identifikován. Základní chování takto vzniklé třídy je, že při požadavku na danou cestu vrací návratový kód `4.05` – Method Not Allowed. Pro umožnění zpracování jednotlivých metod bylo potřeba přepsat chování metod, které mají na starost jednotlivé CoAP metody na daných koncových bodech. Pro

[1] https://github.com/PatrIoT-Framework

metodu `GET` bylo potřeba přepsat `public void handleGET(CoapExchange exchange)` pro metodu `POST` přepsat `public void handlePOST(CoapExchange exchange)` a podobně pro metody `PUT` a `DELETE`.

6.2.1 Implementace třídy EventBusImpl

Tato třída implementuje rozhraní `EventBus`. Stav této třídy je uložen v datové struktuře `TreeMap`, která umožňuje vyhledávání prvků podle klíče a zároveň nad těmito klíči udržuje uspořádání. Viz výpis 6.1. V tomto případě udržuje dvojice klíč–hodnota, kde klíč reprezentuje časový okamžik (třída implementující rozhraní `Time`), pro který jsou registrovány nějaké události a hodnota je objekt `TimeActions`, který udržuje veškeré události, které se mají v daném čase provést.

```
1 private TreeMap<Time, TimeActions> actionsQueue = new TreeMap<>();
```

Výpis 6.1: Ukázka vytvoření struktury pro uchování a řazení všechn akcí.

Třída TimeActions

Tato třída uchovává veškeré akce, které se mají v čase, pro který je přiřazena, udát. Z výpisu 6.2 je patrné, že uchovává tři typy událostí.

Události uložené v množině `recurringAwakeApplicants` jsou uložení klienti, kteří si zažádali o periodické buzení a časový interval, s jakým se mají probouzet.

V množině `awakeApplicants` jsou uložení klienti, kteří si zažádali o probuzení v čase, který je spojen s touto instancí `TimeActions`

Množina `events` ukládá všechny zprávy, které mají být doručeny v čase, který je spojen s touto instancí `TimeActions`.

```
1    private class TimeActions {
2        public Set<ImmutablePair<Time, EventDistributorClient>>
             recurringAwakeApplicants = new LinkedHashSet<>();
3        public Set<EventDistributorClient> awakeApplicants = new
             LinkedHashSet<>();
4        public Set<Event> events = new LinkedHashSet<>();
5    }
```

Výpis 6.2: Třída uchovávající veškeré akce pro stanovený časový okamžik.

Stěžejní metodou pro běh simulace je metoda `tick()`, jejíž definice je zobrazena ve výpisu 6.3. Tato metoda je zodpovědná za provedení jednoho simulačního kroku a vrací booleovskou hodnotu, která značí zdali jsou k dispozici další kroky. Její implementace využívá uspořádanosti hodnot podle klíče ve slovníku `actionsQueue`. Z tohoto slovníku si bere akce s nejnižším možným časem (řádek 2 fragmentu kódu 6.3) a provede všechny akce v daném čase.

```
1    public boolean tick() {
2        Map.Entry<Time, TimeActions> entry = actionsQueue.firstEntry();
3        if(entry == null) {
4            return false;
5        }
6        currentTime = entry.getKey();
7        processTimeStep(entry.getValue());
8        actionsQueue.pollFirstEntry();
9        return !actionsQueue.isEmpty();
10   }
```

Výpis 6.3: Deffinice metody tick.

6.3 Implementace třídy Conductor

Jednou z hlavních zodpovědností této třídy je synchronizace běhu sběrnice s reálným časem. Pro měření reálného času byla zvolena služba `ScheduledExecutorService`, která slouží k plánování příkazů po určitém zpoždění, nebo k jejich periodickému spouštění.[1]. Vzhledem k tomu, že události mohou na sběrnici nastat v libovolných simulačních časech, není možnost využít periodického spouštění a je potřeba dynamicky dopočítat interval k dalšímu spuštění. Simulace synchronizovaná s reálným časem je založená na algoritmu 6.1 zobrazeným níže.

1 Poznamenej si výchozí reálný čas (r_t_start).
2 **Opakuj** dokud jsou naplánované další události.
3 Proveď krok simulace
4 $r\_t\_dalsi = r\_t\_start + simulacni\_cas\_dalsiho\_kroku - vychozi\_simulacni\_cas$
5 $interval = r\_t\_dalsi - aktualni\_realny\_cas$
6 **Pokud** $interval < 0$:
7 Varuj uživatele
8 $interval = 0$
9 naplánuj další probuzení za dobu $interval$

Algoritmus 6.1: Algoritmus plánování běhu simulace v reálném čase.

Výhoda tohoto algoritmu je, že počítá dobu do další události z reálného času počátku simulace, aktuálního reálného času a simulačního času další události. Z těchto hodnot je schopen dopočítat hodnotu reálného času spuštění dalšího kroku. Z rozdílu aktuálního reálného času a reálného času dalšího kroku pak vyjde doba za kterou je třeba naplánovat spuštění dalšího kroku. Díky tomuto výpočtu se nebude akumulovat chyba, která je daná posunem reálného času během běhu programu od zjištění reálného času k zaregistrování k dalšímu spuštění.

Tento algoritmus je implementován ve funkci `public void runRealTime()` jejíž fragment kódu je zobrazen ve výpisu 6.4. V této ukázce lze vidět na řádku 2 poznačení si reálného času pomocí metody `System.currentTimeMillis()`, která poskytuje systémový čas. Spouštění jednotlivých kroků simulace se děje pomocí vláken, jejichž čas spuštění je plánován pomocí služby `ScheduledExecutorService`, jejíž instanciace je zobrazena na řádku 3. Na řádku 8 začíná definice rutiny, která se spouští v naplánovaných časech a zajišťuje spuš-

tění simulačního kroku a následně naplánování dalšího spuštění. Na řádcích 10–12 probíhá samotné spuštění dalšího kroku voláním metody `tick()`, pokud vrátí návratovou hodnotu `false`, není již naplánována další událost, tím pádem není třeba plánovat další krok a rutina se předčasně ukončí. Řádky 15–17 implementují výpočet časového intervalu do dalšího kroku. Řádky 19–22 zpracovávají situaci, kdy výpočet simulačního kroku trval delší dobu než interval od času spuštění po čas spuštění dalšího kroku. Na řádku 23 dochází k plánování dalšího spuštění. Na řádku 27 je inicializační krok, ve kterém dojde k naplánování prvního spuštění rutiny, čímž dojde k cyklickému plánování kroků.

```
1       public void runRealTime() {
2       startRealTime = System.currentTimeMillis();
3       long startSimulationTime = eventBus.getTime().getMillis();
4       scheduler = Executors.newScheduledThreadPool(1);
5       running = true;
6       Runnable task = new Runnable() {
7
8           public void run() {
9               synchronized (shutdownLock) {
10                  if( ! eventBus.tick()) {
11                      pause();
12                      return;
13                  }
14              }
15              Time nextStepSimulationTime = eventBus.getNextStepTime();
16              long nextStepRealTime = startRealTime +
                    nextStepSimulationTime.getMillis() - startSimulationTime;
17              long delay = nextStepRealTime - System.currentTimeMillis();
18
19              if(delay < 0) {
20                  delay = 0;
21                  LOGGER.warn("The simulation step takes longer than the
                        allotted time");
22              }
23              scheduler.schedule(this, delay, TimeUnit.MILLISECONDS);
24          }
25      };
26
27      scheduler.schedule(task, 0, TimeUnit.MILLISECONDS); // start the
            event bus now
28  }
```

Výpis 6.4: Fragment kódu, který implementuje běh simulace v reálném čase.

Zastavení běhu simulace je docíleno funkcí `pause()`, která nechá dokončit zpracovávání simulačního kroku, jeli právě aktivní a zabrání spuštění dalšího. Tohoto cíle je dosaženo zrušením plánovače, který je zodpovědný za spuštění dalšího kroku. Tento příkaz je zobrazen na řádku 6 ukázky kódu 6.5. U přerušení bylo třeba počítat se synchronizačními problémy. Kdyby došlo k ukončení vlákna zpracovávajícího krok simulace, sběrnice by se po dalším spuštění dostala do nedefinovaného stavu. Řešením bylo vytvoření dvou výlučných sekcí se

společným zámkem. Jedna výlučná sekce je na řádcích 94–104 a druhá je v metodě `public void runRealTime()` v ukázce 6.4 na řádcích 97–72. Tato sekce zaručuje, že buď dojde k přerušení ve chvíli kdy sběrnice neprovádí výpočet, nebo se do výlučné sekce metoda `pause()` dostane až ve chvíli, kdy dojde k dokončení simulačního kroku.

```
1   public void pause() {
2       synchronized (shutdownLock) {
3           running = false;
4           scheduler.shutdownNow();
5           try {
6               if(! scheduler.awaitTermination(1, TimeUnit.SECONDS)) {
7                   LOGGER.warn("Event bus did not pause correctly");
8               }
9           } catch (InterruptedException e) {
10              throw new RuntimeException(e);
11          }
12      }
13  }
```

Výpis 6.5: Fragment kódu, zobrazující metodu pro zastavení běhu simulace v reálném čase.

Kapitola 7

Demonstrace využití balíčku eventSimulator pro vytváření testovacího prostředí

V rámci této práce byl vytvořen projekt1, jehož účelem je demonstrovat na konkrétních případech schopnost frameworku PatrIoT a napomáhat při vytváření testovacího prostředí pro itegrační tesování a end-to-end testování.

V tomto projektu byly vytvořeny dva konkrétní testy využívající objekty vytvořené v této práci. První test demonstruje vytvoření jednoduché simulace požáru a jeho vliv na emulované teploměry. Druhý test demonstruje simulaci průjezdu auta kolem automatických dveří.

Na začátku této kapitoly jsou popsány prerekvizity potřebné pro spuštění frameworku PatrIoT, následuje obecný popis kroků pro vytvoření simulace náhodných událostí a na konci jsou dvě konkrétní ukázky.

7.1 Prerekvizity

Před vytvářením samotných testovacích prostředí je nejprve třeba nainstalovat veškeré prerekvizity a balíčky frameworku PatrIoT. Podrobný přehled kroků potřebných pro úspěšnou instalaci je dostupný na oficiálních stránkách frameworku PatrIoT[1]. V době psaní tohoto textu ovšem není vydaná nová verze frameworku, která by již zahrnovala rozšíření z této práce. Tento fakt je třeba zohlednit při instalačním procesu v několika krocích, které se liší od postupu popsaného na stránkách projektu.

1. Na systému je třeba mít nainstalovaný Java Development Kit ve verzi 17 místo verze 8

2. V souboru `pom.xml` vytvořeného Maven projektu, je třeba změnit závislost na PatrIoT balíčcích z poslední vydané verze na verzi této diplomové práce. Tedy nahradit `<patriot.framework.version>3.0.0</patriot.framework.version>` za `<patriot.framework.version>5.0.0-SNAPSHOT-DP</patriot.framework.version>`

Pokud má testovací prostředí využívat virtuální síť frameworku (což demonstrační projekt využívá) je potřeba doinstalovat další prerekvizity. Návod je opět dostupný na strán-

[1] https://patriot-framework.io/latest/docs/getting-started.html

kách projektu[2]. Pro demonstrační ukázku je z tohoto návodu nutný pouze krok instalace stažení docker image.

```
$ docker pull patriotframework/patriot-router:latest
```

Tento image je rovněž dostupný v přiložených souborech.

Konečně je třeba stáhnout a nainstalovat zdrojové kódy modulů frameworku. Tyto moduly jsou dostupné na platformě `GitHub`[3][4][5][6][7] a v přiložených souborech.

Tyto moduly je možné instalovat pomocí příkazu v odpovídajících adresářích:

```
$ mvn clean install
```

Demonstrační testy je pak možné spustit pomocí příkazů spuštěných v domovském adresáři modulu `smart-home-plus-integration-tests`:

```
$  mvn clean test -Dtest=FireSimulation
$  mvn clean test -Dtest=GarageTest#carFast
$  mvn clean test -Dtest=GarageTest#carSlow
```

7.2 Workflow vytvoření testovacího prostředí s podporou simulací událostí

V této sekci je popsán obecný postup pro vytvoření simulačního modelu, propojení simulace s emulovanými zařízeními a řízení simulace během testu.

Vytvoření virtuální sítě dle dokumentace PatrIoTu

Nejprve je třeba definovat virtuální síť, jež obsahuje emulovaná zařízení. Simulace dějů bude následně těmto zařízením modifikovat generovaná data. Postup pro vytvoření virtuální sítě je detailně popsán na stránkách frameworku [8].

Definice dílčích simulačních modelů

Celkový simulační model se skládá z dílčích simulací, které jsou propojeny skrze sběrnici. Uživatel tedy nejprve musí vytvořit tyto dílčí simulace a jejich vzájemné interakce. K tomuto účelu byla vytvořena abstraktní třída `EventBusClientBase`, od které by měly dědit uživatelem definované dílčí simulace. Nová třída musí definovat tři abstraktní metody z rodičovské třídy `EventBusClientBase`. Jedná se o metody `init`, `awake` a `recieve`, které byly popsány u rozhraní `EventDistributorClient` v podsekci 5.5.2.

Pro definici výše zmíněných metod třída `EventBusClientBase` poskytuje funkce pro interakce s dalšími dílčími simulacemi. Jedná se o metody:

[2] https://patriot-framework.io/latest/docs/network-sim.html
[3] https://github.com/mastastny/patriot-data-generator/tree/diploma-thesis-tag
[4] https://github.com/mastastny/patriot-api/tree/diploma-thesis-tag
[5] https://github.com/mastastny/docker-network-simulator/tree/diploma-thesis-tag
[6] https://github.com/mastastny/virtual-smart-home-plus/tree/diploma-thesis-tag
[7] https://github.com/PatrIoT-Framework/smart-home-plus-integration-tests/tree/diploma-thesis-tag
[8] https://patriot-framework.io/latest/docs/network-sim.html

- `subscribe(String topic)`: Přihlášení se k odběru zpráv pod daným tématem.
- `registerAwake(Time time)`: Registrace k probuzení v konkrétním čase.
- `registerRecurringAwake(Time interval)`: Registrace k pravidelnému buzení s daným intervalem.
- `registerRecurringAwake(Time interval, Time startTime)`: Registrace k pravidelnému buzení s daným intervalem se stanoveným časem prvního probuzení (startTime).
- `unregisterRecurringAwake()`: Zrušení pravidelného buzení.
- `publish(Data message, String topic)`: Poslání zprávy všem registrovaným zájemcům k danému tématu.
- `publishOnTime(Data message, String topic, Time time)`: Poslání zprávy všem registrovaným zájemcům k danému tématu s doručením ve stanoveném čase (time)

Pro reprezentaci objektů v prostoru byly vytvořeny Třídy reprezentující souřadnice v různých souřadnicových systémech. Tyto třídy jsou dostupné v balíčku `io.patriot_framework.generator.eventSimulator.coordinates`. Souřadnice mohou být posílány jako součást těla zpráv.

Definice adaptérů

Adaptér je specifická implementace klienta sběrnice, jehož zodpovědností je provádět komunikaci stavu simulace mezi simulačním modelem a koncovým zařízením, za které je daný adaptér zodpovědný.

Jako podpora uživateli pro vytvoření adaptéru jsou k dispozici dvě abstraktní třídy – `SensorAdapterBase` a `ActuatorAdapterBase`. První z nich slouží k překladu stavu simulace do dat generovaných senzory. Druhá slouží k propagaci změny stavu aktuátoru do simulace.

Pro vytvoření nového adaptéru k aktuátoru je potřeba dědit od třídy `ActuatorAdapterBase`. Konstruktoru této třídy by měl být předán parametr s jakým intervalem bude probíhat kontrola změny stavu aktuátoru a instance třídy `ActuatorMessenger`. Dále je třeba definovat chování abstraktní metody `processUpdates`. Tato metoda by měla zpracovat změnu stavu aktuátoru posláním událostí do sběrnice. Kromě metod z třídy `EventBusClientBase` má k dispozici navíc tyto metody:

- `String getNext()`: Vrací další doposud nezpracovanou změnu stavu.
- `String peekLast()`: Vrací poslední změnu stavu.
- `boolean hasChanged()`: Vrací `true`, pokud se změnil stav od minulého volání `processUpdates`.
- `int stateUpdatesCount()`: Vrací počet nezpracovaných změn stavu.

Pro vytvoření nového adaptéru k senzoru je potřeba dědit od `SensorAdapterBase`. Konstrutoru této třídy by měla být předána instance třídy `DataFeedMessenger`. Chování třídy by mělo být jako v třídě `EventBusClientBase` popsáno v metodách `init`, `awake` a `recieve`. Uživatel má pro definici chování k dispozici opět metody z třídy `EventBusClientBase`. Navíc má k dispozici tyto metody:

- `boolean changeDataFeed(DataFeed dataFeed)`: Vymění data-feed u senzoru za který má adaptér zodpovědnost.
- `boolean updateData(Double data)`: Nahraje konstantu do senzoru.

Sestavení simulace

K sestavení, řízení a spuštění simulace slouží třída `Conductor`. Pro vytváření instance třídy `Conductor` jsou k dispozici dva konstruktory. První konstruktor `Conductor (EventBus eventBus)` očekává implementaci rozhraní `EventBus`. Druhý konstruktor očekává implementaci rozhraní `Time`, které určuje od jakého simulačního času bude spuštěna simulace. Druhý konstruktor si vytváří vlastní výchozí sběrnici z třídy `EventBusImpl`.

Před spuštěním simulace je třeba vytvořit instance dílčích simulací a adaptérů. U každého páru adaptéru je navíc nutné předat instanci třídy, která zajišťuje komunikaci mezi adaptérem a koncovým zařízením (messenger). Každý messenger obsahuje adresu zařízení.

Všechny dílčí simulace a adaptéry je potřeba zaregistrovat pomocí metody `addSimulation` nad instancí třídy `Conductor`. Zamýšlený způsob, jak přidávat dílčí simulace do hlavní simulace, je před prvním spuštěním simulace, avšak implementace sběrnice a třídy `Conductor` nevylučuje ani přidávání v průběhu simulace.

Řízení běhu simulace

K řízení běhu simulace je dispozici sada metod implementovaných v objektu třídy `Conductor`, která byla instanciována v předchozím kroku.

Pro běh simulace v módu „co nejrychleji“ jsou dostupné funkce `run()`, `runFor(Time duration)` a `runUntil(Time endTime)`. Tyto funkce umožňují co nejrychlejší posun do kýženého simulačního časového momentu. Pro mód „v reálném čase“ lze použít funkce `runRealTime()`, `runRealTimeFor(Time duration)` a `runRealTimeUntil(Time endTime)`, které synchronizují simulaci s reálným časem. Detailnější popis výše zmíněných funkcí je zmíněn v podsekci 5.5.3.

7.3 Ukázka použití – simulace požáru

Jedná se o simulaci požáru v domě, která využívá diskrétní souřadnice implementované pomocí třídy `UndirectedGraphSpace` a `UndirectedGraphCoordinate`. Každá souřadnice (vrchol grafu)) reprezentuje jednu místnost v bytě. Každá místnost obsahuje data o teplotě (*temperature*), množství paliva (*fuel*) a příznak, zdali je zapálená (*is-on-fire*). Topologie sestaveného grafu reprezentuje rozmístění místností v domu zobrazeného na obrázku 4.2 z podsekce 4.1.3. Názvy pokojů jsou použity jako souřadnice pro vrcholy grafu.

Části simulačního modelu použité v této simulaci jsou:

- `TemperatureDiffuser`: Třída zodpovědná za šíření tepla mezi místnostmi. Je implementována pomocí celulárního automatu (4.2.3), který do dané místnosti uloží vážený průměr teplot z dané místnosti a jejich sousedů.
- `Fire`: Simulace požáru. V každé místnosti, která hoří, zvýší teplotu výměnou za množství paliva v místnosti. Pokud daná místnost nehoří, zkontroluje zdali místnost nesplňuje iniciální podmínky pro vznik požáru.

- `ChildWithMatches`: Zakládá požár v náhodné místnosti. V každém kroku s pravděpodobností 1/3 založí požár a zanikne nebo (s pravděpodobností 2/3) se přemístí do sousední místnosti.

Dále tato simulace obsahuje tři teploměry umístěné v různých místnostech *livingRoom*, *workRoom* a *corridor*. Rovněž obsahuje jejich adaptéry, které propagují teploty do koncových zařízení – dvou teploměrů.

Tato simulace se spouští v reálném čase a demonstruje použití diskrétních souřadnic a celulárního automatu nad těmito souřadnicemi. Rovněž demonstruje použití adaptérů pro propagaci teploty do senzorů.

7.4 Ukázka použití – simulace sledování pohybu

Jedná se o simulaci otevírání automatických garážových vrat na základě vzdálenosti auta od garáže. Tato simulace využívá spojité souřadnice implementované pomocí třídy `CartesianCoordinateBase` Dílčí simulace jsou:

- `LinearMotion`: Modeluje pohyb auta pomocí jednoduché rovnice rovnoměrného přímočarého pohybu. Konstruktor přijímá implementaci potomka `AbstractContinuousTime`, který reprezentuje spojitý čas, tím pádem je možné spustit simulaci s téměř libovolnou časovou granularitou.

- `DoorSimulation`: Kontroluje jestli auto nenabouralo do garáže. V pravidelných intervalech porovnává vzdálenost auta od garáže a pokud je auto moc blízko a zároveň není garáž otevřena, tak vyvolá událost *CRASH*. Ta zastaví auto.

Dále tato simulace obsahuje garážová vrata, které interně obsahují vzdálenostní senzor (měřící vzdálenost auta od garáže) a píst (aktuátor otevírající vrata). Také obsahuje jejich adaptéry. Tato simulace se spouští v módu „v reálném čase“ a demonstruje použití spojitých souřadnic a spojitého času v souvislosti se simulováním pomocí matematických rovnic. Rovněž demonstruje použití adaptéru aktuátoru.

Simulace je napsána ve dvou různých verzích. Liší se od sebe různou počáteční rychlostí automobilu. V první se dveře nestihnou otevřít včas a dojde k havárii. Ve druhé jede auto pomaleji a vrata se stihnou otevřít včas – auto jí projede bez kolize.

Kapitola 8

Závěr

V tomto textu byla zdokumentována práce, která se zabývala rozšířením frameworku pro testování IoT systémů. V sedmi kapitolách byly detailněji popsány požadavky na řešení, čtenář byl seznámen s účelem a strukturou dosavadního stavu frameworku. Dále byly popsány základní simulační modely, matematické definice pro různé prostory a k nim náležící souřadnicové systémy. V návrhové části byly zhodnoceny poznatky z předchozích kapitol a byl navržen systém tříd založený na návrhovém vzoru data bus. Ten umožňuje definovat dílčí simulace událostí, které jsou spolu schopné mezi sebou interagovat a vytvářet tak rozsáhlou simulaci, která je schopna ovlivňovat data generovaná virtuálními zařízeními v rámci celé virtuální sítě. V implementační části byly popsány klíčové části kódu a na demonstračním projektu byla ukázána, jak funkčnost řešení, tak popsán postup vytvoření simulace za využití různých souřadnic.

Z výše zmíněného vyplývá, že práce naplnila zadání, avšak její potenciál sahá mnohem dále.

Směr dalšího vývoje nejspíše ukážou až zkušenosti z praktického využití, nicméně některé náměty lze nastínit už nyní. Jedním je vývoj jádra samotného simulátoru. Vzhledem k tomu, že se simulace využívá k testovacím účelům, nepředpokládá se u ní, že by měla mít masivní nároky na výpočetní výkon, tudíž u ní nebyl brán ohled na paralelizaci a distribuci výpočtu skrze síťovou komunikaci. Přesto by paralelní implementace mohla najít uplatnění ve specifických případech.

Aktuální technický stav modulu `patriot-api` neumožňuje spolehlivě navázat komunikaci z jednotlivých virtuálních podsítí k serveru spuštěném v rámci testu. Po budoucí úpravě zmiňovaného modulu bude možné distribuovat data ze simulace v pull modelu komunikace a z aktuátoru do simulace naopak v modelu komunikace push. Toto řešení je efektivnější a snižuje síťový provoz.

Další možný směr vývoje spočívá ve zvýšení uživatelské přívětivosti. Obdobně jako v rámci této práce vznikl balíček pro simulaci požáru v domě, framework by mohl poskytovat další sady dílčích simulátorů, které by byly zaměřeny na specifické domény. Příkladem budiž sada dílčích simulátorů, která umožní zařízením pohybovat se po křivkách na rotačním elipsoidu, která by umožňovala senzorům poskytovat polohu GPS. Pro uživatele by též mohlo být zajímavé implementovat další třídy pro reprezentaci dalších různých druhů souřadnic.

Během vývoje se ukázalo že knihovna `Portico` implementující infrastrkuturu pro běh HLA simulací, která slibovala zvýšení interoperability a znovupoužitelnosti, se ukázala téměř nepoužitelná. Tato knihovna je v současné době velice špatně udržovaná, s neaktuální dokumentací a nevyřešenými chybami kódu. Po usilovné snaze tyto překážky překonat bylo nakonec od využití této knihovny upuštěno.

Literatura

[1] *Interface ScheduledExecutorService* [online]. Oracle [cit. 2024-08-20]. Dostupné z: https://docs.oracle.com/en%2Fjava%2Fjavase%2F17%2Fdocs%2Fapi%2F%2F/java.base/java/util/concurrent/ScheduledExecutorService.html.

[2] *Java Language Changes* [online]. Oracle [cit. 2024-05-15]. Dostupné z: https://docs.oracle.com/en/java/javase/17/language/java-language-changes.html.

[3] *PatrIoT Framework Documentation* [online]. [cit. 2024-01-14]. Dostupné z: https://patriot-framework.io/latest/welcome/index.html.

[4] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*. Aug 2010, s. 1–38. DOI: 10.1109/IEEESTD.2010.5553440.

[5] Berger, M. *Geometry*. Springer-Verlag, 1987. Geometry I [-II]. ISBN 9783540116585.

[6] Bureš, M. Framework for Integration Testing of IoT Solutions. In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2017, s. 1838–1839. ISBN 9781538626528.

[7] Bureš, M., Ahmed, B. S., Rechtberger, V., Klima, M., Trnka, M. et al. PatrIoT: IoT Automated Interoperability and Integration Testing Framework. In: *IEEE International Conference on Software Testing Verification and Validation*. 2021, s. 454.

[8] Calcaterra, C., Boldt, A., Green, M. a Bleecker, D. Metric Coordinate Systems. 2002.

[9] Chacon, S. a Straub, B. *Pro Git (Second Edition)*. Second edition. Berkeley, CA: Apress, 2014. ISBN 9781484200773.

[10] Company, S. *Maven : the definitive guide*. 1st ed. Sebastopol: O'Reilly, 2008. ISBN 978-0-596-51733-5.

[11] Douglass, B. P. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison Wesley, 2002. ISBN 0-201-69956-7.

[12] Eclipse Foundation. *Class CoapResource* [https://javadoc.io/doc/org.eclipse.californium/californium-core/latest/org/eclipse/californium/core/CoapResource.html]. [cit. 2024-06-16].

[13] Falcone, A., Garro, A., Anagnostou, A. a Taylor, S. J. An introduction to developing federations with the High Level Architecture (HLA). In: *2017 Winter Simulation Conference (WSC)*. 2017, s. 617–631. DOI: 10.1109/WSC.2017.8247820.

[14] Foundation, E. *Eclipse Californium* [online]. [cit. 2024-08-15]. Dostupné z: https://eclipse.dev/californium/.

[15] Kaminski, N., Murphy, M. a Marchetti, N. Agent-based modeling of an IoT network. In:. říjen 2016, s. 1–7. DOI: 10.1109/SysEng.2016.7753151.

[16] Khan, M. A. K. A. S. S., ed. *Internet of things (IoT) : concepts and applications*. Cham: Springer, 2020. ISBN 978-3-030-37467-9.

[17] Kropáček, P. a Balážová, M. *Jak vyzrát na open source software - 2. část*. Sep 2016. Dostupné z: https://www.epravo.cz/top/clanky/jak-vyzrat-na-open-source-software-2-cast-102708.html.

[18] Lee, J. M. *Introduction to smooth manifolds*. Second edition. New York: Springer Science+Business Media, 2013. Graduate text in mathematics (Springer). ISBN 978-1-4419-9981-8.

[19] Martin, R. C. *Design Principles and Design Patterns*. Www.objectmentor.com, 2000. Dostupné z: https://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf.

[20] Matoušek, J. *Kapitoly z diskrétní matematiky*. 2. upr. vyd. Praha: Karolinum, 2000. ISBN 80-246-0084-6.

[21] Pelánek, R. *Modelování a simulace komplexních systémů : jak lépe porozumět světu*. Vyd. 1. Brno: Masarykova univerzita, 2011. ISBN 978-80-210-5318-2.

[22] Richards, M. W. M. *Fundamentals of software architecture : an engineering approach*. First edition. Beijing ; Boston ; Farnham ; Sebastopol ; Tokyo: O'Reilly, 2020. ISBN 978-1-4920-4345-4.

[23] Robillard, M. P. *Introduction to Software Design with Java*. Second edition. Cham: Springer International Publishing AG, 2022. ISBN 3030978982.

[24] Shelby, Z. *The Constrained Application Protocol (CoAP)* [online]. [cit. 2023-11-9]. Dostupné z: https://www.rfc-editor.org/rfc/rfc7252.html.

[25] Sigler, L. *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation*. Springer New York, 2003. Sources and Studies in the History of Mathematics and Physical Sciences. ISBN 9780387407371.

[26] Smolár, J. *Data generator for IoT testing [online]*. 2020 [cit. 2024-02-01]. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. SUPERVISOR : Adam Rambousek. Dostupné z: https://is.muni.cz/th/omtep/.

[27] Sutherland, W. A. *Introduction to Metric and Topological Spaces*. 2. vyd. Oxford University Press, 2009. [Oxford Mathematics]. ISBN 019956308X,9780199563081.

[28] Wolfram, S. Cellular automata as models of complexity. *Nature*. 1984, sv. 311, s. 419–424.

[29] Wolfram, S. *A new kind of science.* Champaign, IL: Wolfram Media, 2002. ISBN 1-57955-008-8.

[30] Wooldridge, M. J. *An introduction to multiagent systems.* 2nd ed. Chichester: Wiley, 2009. ISBN 978-0-470-51946-2.

Příloha A

Obsah přiložené SD karty

```
/
├── README.md ..... popis obsahu média
├── diplomová_práce.pdf ..... technická zpráva
├── diplomová_práce_zdroj ..... adresář se zdrojovými soubory technické zprávy
├── zdrojový_kód ..... adresář se zdrojovými kódy
└── soubory_změn_zdrojového_kódu ..... úpravy kódu zaznamenané nástrojem git
```