VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

BACKEND PRO KOLABORATIVNÍ PROGRAMOVÁNÍ V ROZŠÍŘENÉ REALITĚ

BACKEND FOR COLLABORATIVE PROGRAMMING IN AUGMENTED REALITY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. TOMÁŠ WILLASCHEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ZDENĚK MATERNA, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Willaschek Tomáš, Bc.**
Program: Informační technologie
Obor: Inteligentní systémy
Název: **Backend pro kolaborativní programování v rozšířené realitě**
Backend for Collaborative Programming in Augmented Reality
Kategorie: Softwarové inženýrství
Zadání:

1. Seznamte se s postupy a technikami využívanými pro kolaborativní aplikace obecně a speciálně pak s možnostmi pro zamykání sdílených prostředků apod. v jazyce Python (asyncio).
2. Seznamte se se systémem pro programování v rozšířené realitě ARCOR2.
3. Navrhněte úpravu serverové části řešení tak, aby umožňovala různé typy spolupráce více uživatelů současně.
4. Návrh implementujte.
5. Navrhněte a implementujte sadu testů.
6. Řešení zdokumentujte.

Literatura:

- Dle doporučení vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1-3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Materna Zdeněk, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

Abstrakt
Tato práce se zabývá implementací a aplikací výlučného přístupu pro systém ARCOR2, který slouží ke kolaborativnímu programování robotů v rozšířené realitě. Cílem práce je analýza výchozího stavu, návrh a implementace výlučného přístupu pro tento systém. Implementace je rozsáhlá a umožňuje řadu pracovních scénářů, které vyžadují aplikaci výlučného přístupu. Tyto scénáře jsou odhaleny analýzou systému. Na tomto základu je vytvořen návrh, který řeší problematické scénáře. Nedostatek systému je vyřešen vytvořením globálního manažera zámků, který je aplikován. V rámci práce jsou definovány vzory, jak manažera zámků použít. Přínosem práce je efektivní a nekonfliktní kolaborativní programování.

Abstract
This thesis deals with the implementation and application of exclusive access for the ARCOR2 system, which is used for collaborative programming of robots, using augmented reality. The goal of this thesis is an analysis of default state, proposal, and implementation of an exclusive access solution for this system. The implementation is extensive and allows for a number of work scenarios, which require the usage of exclusive access. Scenarios are revealed by the system analysis. Based on the analysis proposal of a solution is created. The problem is resolved by creating a global lock manager, which is applied. Patterns of how the manager should be used, are defined in the work. The benefits of this work are effective and easygoing collaborative programming.

Klíčová slova
výlučný přístup, zamykání v AsyncIO, konkurentní programování, rozšířená realita

Keywords
exclusive access, AsyncIO lock, concurrent programming, augmented reality

Citace
WILLASCHEK, Tomáš. *Backend pro kolaborativní programování v rozšířené realitě*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zdeněk Materna, Ph.D.

Backend pro kolaborativní programování v rozšířené realitě

Prohlášení

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

....
Tomáš Willaschek
18. května 2021

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce, Ing. Zdeňkovi Maternovi Ph.D., za jeho ochotu, věnovaný čas a odborné rady, které mě vždy nasměrovaly správným směrem.

Dále bych chtěl poděkovat celému ARCOR2 týmu za včasnou implementaci souvisejících změn a pomoc s jejich testováním.

Obsah

1 Úvod **2**

2 Kolaborativní aplikace **3**
2.1 Principy vytváření kolaborativního softwaru 4

3 ARCOR2 **5**
3.1 Části systému 5
3.1.1 Front-End – AREditor 6
3.1.2 Back-End – Server 10

4 Návrh zamykání **13**
4.1 Vzájemné vyloučení přístupu ke sdíleným zdrojům 13
4.2 Knihovna Asynchronous I/O 14
4.2.1 Zámek v AsyncIO 15
4.3 Požadavky pro návrh výlučného přístupu 16
4.3.1 Demonstrace současných problémů v kódu 17
4.4 Návrh výlučného přístupu 18
4.4.1 Příklady zamykání 21

5 Implementace **23**
5.1 Zámek a jeho části 23
5.1.1 Zamykání a RPC 27
5.1.2 Vylepšení implementace 29
5.2 Registrace uživatelů 30
5.2.1 Notifikace uživatelů 30
5.3 Možná vylepšení 32
5.4 Dokumentace řešení 34
5.5 Zhodnocení 34

6 Testování **36**

7 Závěr **39**

Literatura **40**

A Obsah CD **42**

Kapitola 1

Úvod

Kolaborativní aplikace jsou dnes velmi rozšířené. Mnoho z nich na první pohled nevypadá jako kolaborativní, ať už to je díky důmyslnému zpracování dané aplikace nebo jen úzkým spektrem možností, které přímo zasahují do tohoto tématu. Základem takovýchto aplikací je kvalitně navržené jádro, které zvládá obsloužit více uživatelů současně, s důrazem na rychlost zpracování paralelních požadavků. Jádro musí za každé situace zprostředkovávat validní a hlavně konzistentní informace. Z tohoto důvodu je také kladen důraz na zapracování výlučného přístupu ke zdrojům, se kterými uživatelé pracují.

Cílem této práce je analýza, návrh a implementace výlučného přístupu pro serverovou část systému ARCOR2. Tento sytém je tvořen serverem a kolaborativní aplikací a umožňuje programování v rozšířené realitě. Uživatel systému definuje chování robota skrze aplikaci, avšak programován může být jakýkoliv jiný objekt. Cílem uživatele je vytvořit si scénu se speciálními body, pomocí kterých následně definuje chování robota (např. přesun na pozici, uchopení objektu, aj.).

V průběhu analýzy jsou odhaleny části, které jsou ve zmíněném systému náchylné na chybu a jsou definovány požadavky na implementaci. Dále je definován návrh řešení, který je demonstrován na reálných problémech a tento návrh je následně implementován spolu se sérií testů, které zajistí jeho funkčnost. Celé řešení je doplněno o dokumentaci.

Obsah práce je rozdělen na části, z nichž kapitola 2 je věnována kolaborativním aplikacím obecně, k čemu slouží a jaké na ni mohou být požadavky. Dále zde jsou popsány principy, které by každá kolaborativní aplikace měla splňovat. V kapitole 3 je popsán systém ARCOR2 a uživatelské rozhraní, doplněné o reálné snímky, aby byl čtenář srozuměn s účelem aplikace a její funkcionalitou. Tato část se dále zabývá funkcionalitou serveru a komunikací v rámci celého systému. Kapitola 4 je věnována technologiím, na kterých je vystavěn server systému ARCOR2. Druhá část této kapitoly se zabývá návrhem implementace výlučného přístupu pro serverovou část, kde je tento návrh představen na reálných problémech. Implementaci a řešení souvisejících problémů je věnována kapitola 5. Kapitola 6 se zabývá validaci a otestováním implementace. Nachází se zde popis testů, které jsou za tímto účelem vytvořeny a reálné případy použití, které je potřeba validovat.

Kapitola 2

Kolaborativní aplikace

Kolaborativní systémy jsou systémy, které umožňují skupině uživatelů vidět a editovat stejný typ dokumentu, média, objektu, ve stejný čas nezávisle na geografickém umístění uživatele, pomocí komunikační sítě, v reálném čase [10]. V dnešní době existuje celá řada takovýchto systémů, jako příklad lze uvést sdílený dokument. Jelikož uživatelé pracují ve sdíleném prostředí se stejnými prostředky, existují požadavky, které by dané prostředí mělo splňovat. Podle [17] mohou být požadavky následující:

- **Kolaborativní prohlížení** – všem uživatelům se sdílené prostředky zobrazují stejně, nezávisle na zařízení, jež používají.
- **Kolaborativní manipulace** – uživatelé manipulující s prostředkem jsou omezováni pravidly, které zabraňují vznikům konfliktů.

První z těchto požadavků lze vyřešit zasíláním informací v jednotném formátu, tedy informace o změně bude obsahovat stanovené parametry, se kterými umí konkrétní zařízení pracovat a tyto informace budou ve stejném formátu (stejné jednotky, souřadnicový systém). Zprávy o změně by měly být zasílány okamžitě po provedení změny, aby všichni uživatelé měli přístup k aktuálním datům Druhý z požadavků lze splnit dvěma způsoby:

- **Algoritmy pro odstranění konfliktů** – v tomto případě musí kolaborativní systém disponovat mechanismem, který dokáže detekovat konflikty a v ideálním případě je i sám odstranit. Takovýto mechanismus popisuje například [7].
- **Zamykáním** – sdílený prostředek upravuje vždy omezený počet uživatelů. Pro ostatní se jeví jako nedostupný. Tento stav je uživatelům nejčastěji signalizován zařízením, které zprostředkovává informace o daných prostředcích. Zařízení nejčastěji disponuje seznamem prostředků u nichž si značí, zda jdou odemčené či uzamčené, jako tomu je například v [5].

Velmi častým a jednodušším z řešení je právě aplikace zamykání, která umožňuje editaci pouze uživateli, který získá výlučný přístup k danému prostředku. Pro toto řešení není zapotřebí detekce konfliktů, protože prostředek edituje nejčastěji výhradně jeden uživatel.

2.1 Principy vytváření kolaborativního softwaru

Pro většinu typů aplikací existují principy, jak je správně vytvářet a požadavky, které musí výsledné řešení splňovat (např. informační systém musí uchovávat informace o uživatelích, apod.). Jelikož kolaborativní aplikace umožňují současnou práci více uživatelů na jednom projektu, je více než potřebné definovat, nebo se držet již definovaných požadavků na takovouto aplikaci.
Požadavky pro návrh kolaborativního softwaru se dají shrnout do čtyř bodů [6]:

1. **Paměť** – Kolaborativní aplikace musí uchovávat data, která vytvořil uživatel. Tyto data pak následně správně zobrazovat ostatním uživatelům, kteří spolupracují na jednom projektu.

2. **Uživatelé** – Mnoho takovýchto aplikací potřebuje ke své funkčnosti rozdělit uživatele do skupin, kde každá takováto skupina má přístup k určitým prostředkům. Za tímto účelem je potřeba uchovávat určitá data o uživatelích, pro jejich pozdější identifikaci.

3. **Výlučný přístup** – Jelikož kolaborativní aplikace je vytvořená pro to, aby více lidí mohlo společně vyvíjet jednu věc, musí tato aplikace zajistit, že vyvíjená část zůstane vždy v konzistentním stavu pro všechny uživatele.

4. **Sezení** – Systém by měl být schopen zjistit, ve kterém sezení uživatel pracuje.

Kapitola 3

ARCOR2

ARCOR2 [19] (Augmented Reality Collaborative Robot) je systém pro programování kolaborativních robotů, který umožňuje vytvářet scény (viz 3.1.1), což je vlastně anotování pracovního prostoru, kdy se určuje rozmístění konkrétních objektů. Na základě vytvořené scény lze vytvořit projekt, ve kterém uživatelé programují chování objektů, nejčastěji právě robota. Toto programování probíhá pomocí vkládání speciálních bodů, jež jsou prostorovými značkami, které nejčastěji definují trajektorii pohybu robota. Systém dokáže s robotem pracovat na více úrovních, a proto mohou uživatelé definovat kromě trajektorie také rychlost pohybu či nastavení jednotlivých robotických kloubů. Systém slouží primárně k programování robotů, avšak je vytvořen natolik obecně, že dovoluje definování vlastního objektu s následným programováním jeho chování.

Celé toto programování probíhá v rozšířené realitě. Podle [1] je rozšířená realita scéna reálného světa, která je upravená přidáním virtuálních, počítačově generovaných informací v reálném čase. Virtuálně vytvořenou scénou se rozumí například model robota, který je umístěn například na stůl v reálném prostředí. Tato kombinace virtuální scény a reálného prostředí je následně zobrazena například na mobilním zařízení nebo speciálními brýlemi (například HoloLens[1]). V případě mobilního zařízení (telefonu, tabletu, aj.) je nezbytné, aby toto zařízení disponovalo potřebným software, který umožňuje zobrazovat virtuální objekty v reálném prostředí. Je důležité zmínit, že v případě rozšířené reality se nejedná pouze o 2D scénu, nýbrž o 3D. Z toho vyplývá, že aplikace podporující rozšířenou realitu musí být schopna prostorově ukotvit objekty.

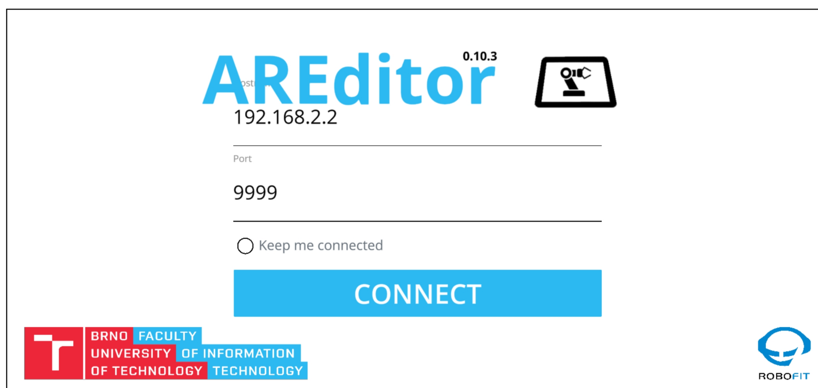
3.1 Části systému

ARCOR2 je distribuovaný systém, postavený na architektuře klient-server [14], která se vyznačuje tím, že klient zasílá požadavky, server je zpracovává a následně na ně odpovídá. Dalším specifikem je způsob ukládání dat, kde server disponuje databází, ve které si ukládá data a následně je posílá klientovi, který je zobrazuje. Klientem je kolaborativní aplikace AREditor (viz 3.1.1), skrze kterou uživatel definuje rozmístění objektů ve scéně a následně programuje jejich chování. Druhou část tvoří server (viz 3.1.2), ke kterému se klient musí připojit. Server dále validuje požadavky zaslané uživatelem AREditoru, odpovídá na ně a informuje ostatní přihlášené uživatele o změnách.

[1] **HoloLens** – https://www.microsoft.com/en-us/hololens

3.1.1 Front-End – AREditor

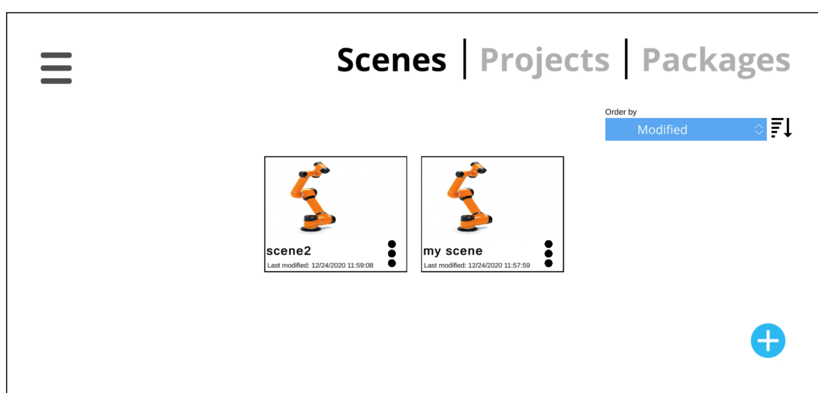
AREditor je kolaborativní aplikací. Princip užívání této aplikace je takový, že každý klient se připojí k serveru (viz 3.1) na kterém si vytváří nebo kolaborativně vyvíjí již existující projekt. Po přihlášení jsou uživateli zpřístupněny záložky pro zobrazení scén, projektů a ba-



Obrázek 3.1: Formulář pro přihlášení k serveru

líčků (obrázek 3.2). Záložka `Scenes` přehledně zobrazí již vytvořené scény a také tlačítko pro vytvoření nové scény. S každou zobrazenou scénou je možné dělat následující operace:

- **Přidat hvězdičku** – tato možnost slouží pro označení důležitých scén. Díky tomuto označení lze scény předřadit před ostatní.
- **Editovat** – tato možnost zahrnuje akce jako přejmenování scény, její smazání, či změny obrázku. Všechny tyto změny se týkají pouze záložky `Scenes`.
- **Kopírovat** – vytváří duplikát vybrané scény.
- **Vazba na projekty** – je možné vytvořit projekt přímo z existující scény nebo si přehledně zobrazit, které projekty používají konkrétní scénu.
- **Exportovat** – exportuje scénu z aplikace.



Obrázek 3.2: Hlavní menu aplikace

Obdobné možnosti nabízejí zbylé dvě záložky `Projects` a `Packages`, každá pro svůj obsah, tedy projekty a exekuční balíčky (viz 3.1.2).

Je důležité říci, že server je navržený tak, že podporuje pouze jedno aktivní sezení. To znamená, že pokud se první přihlášený uživatel rozhodne něco měnit ve scéně, kterou si otevře, nově přihlášení uživatelé jsou považováni za spolupracovníky a je jim automaticky otevřena tatáž scéna. Stejně tak v případě projektů.

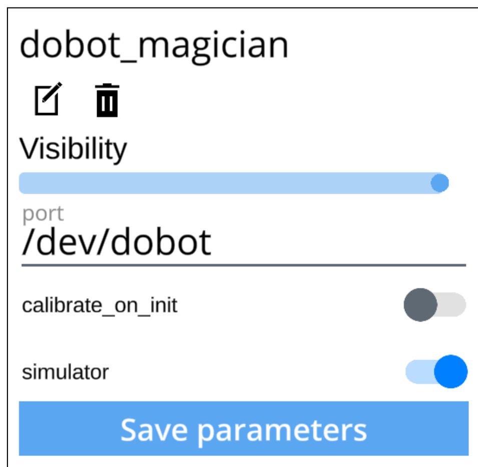
Scéna

Základním stavebním kamenem při vytváření projektu za použití AREditoru je scéna. Zde uživatel definuje, kde budou umístěny objekty (příklad jednoduchého rozmístění objektů je zachycen na obrázku 3.3). Objekt je buď předem naprogramován s určitými vlastnostmi či přepínači, nebo může být uživatelem definován přímo ve scéně. Co se týče předprogramovaných objektů, ty je potřeba nejprve nahrát na server, aby byly následně dostupné ve scéně. Při definici objektu ve scéně se jedná o jednoduché objekty, například různé tvary (kostka, koule), jejichž vzhled jde upravit texturou a tento objekt se uživatelům v rozšířené realitě zobrazuje jako konkrétní předmět, se kterým pracují. Tyto nově vytvořené nebo nahrané objekty je také možné smazat.

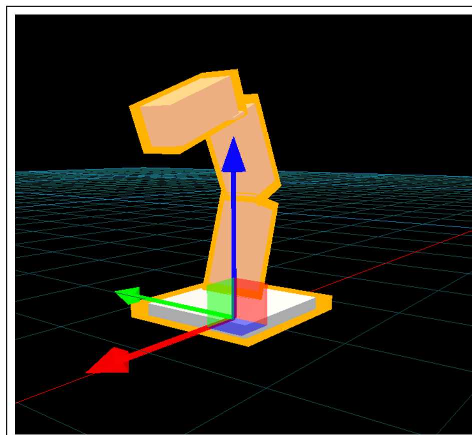
Objekt lze do scény umístit prakticky kamkoliv, pokud toto umístění dává logický smysl. Umístění není fixní, s objektem je možné i nadále pohybovat (viz obrázek 3.4b), otáčet jej či smazat. Každý objekt má dále své menu (viz obrázek 3.4a), v němž je možné měnit některé z předprogramovaných parametrů (např. adresu a port, na kterém komunikuje robot, pro umožnění pohybu s robotem z editoru). Dále je možné změnit například: jméno, viditelnost objektu ve scéně a další parametry, které jsou pro daný objekt dostupné. Je důležité zmínit,

Obrázek 3.3: Příklad jednoduché scény s robotem a dvěma objekty

že scéna je jediným prostředkem jak upravovat pozici a orientaci objektů. V módu projektu se pouze programuje chování a pozice objektů je již pevná.



(a) Menu objektu s jeho jménem, dostupnými akcemi, viditelností a volitelnými parametry



(b) Posuv objektu ve scéně

Obrázek 3.4: Příklad dostupných akcí ve scéně pro jednotlivé objekty

Projekt

Jakmile má uživatel definovaný vzhled scény, může přistoupit k programování chování jednotlivých objektů. Toto programování se provádí za pomocí akčních bodů[2] v prostředí projektu.
Pokud je ve scéně robot, akčním bodům je možné přiřadit například tyto akce:

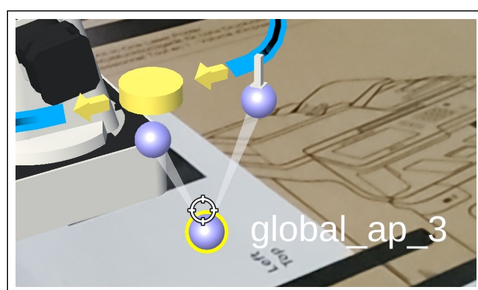
- **Přisaj** – pokud má robot přísavku, tato akce zajistí aktivaci této přísavky. Robot tedy uchopí daný předmět přisátím.
- **Uvolni** – robot deaktivuje svou „uchopovací“ část (např. přísavku) a tím uvolní předmět, který drží.
- **Hýbej se** – robot změní svou orientaci tak, že se natočí směrem k akčnímu bodu a přesune zde své rameno.

Výše zmíněné akce jsou pouhou ilustrací. Tyto akce jsou specifické pro každého robota či objekt, jehož autor může naprogramovat prakticky jakoukoliv smysluplnou akci.

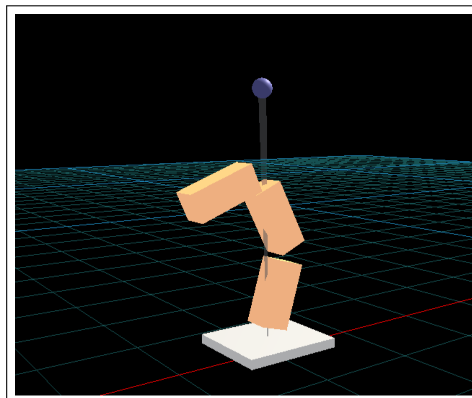
Pro vytváření akcí je nejprve potřeba vložit akční bod, jež je základním stavebním kamenem této programovací logiky. Akční body se vytvářejí vždy globální, ale lze jim definovat předky, čímž vzniká stromová struktura. Těmto bodům se obvykle přiřazuje orientace, která značí, v jakém směru přistaví robot svůj efektor na pozici akčního bodu. K vytvořenému bodu lze pomocí menu následně přidat akci či vícero akcí. Součástí vytvoření je i definice, který aktivní objekt (nejčastěji robot) bude tuto akci vykonávat. Vytvořenou akci lze vidět na obrázku 3.5a.

Jakmile má uživatel vytvořené potřebné akce, může je začít propojovat v logickém pořadí, jak jdou po sobě. Tímto propojením uživatel vytváří program, který následně řídí robota (například nastavení robota na místo A, přisátí předmětu, nastavení na místo B a uvolnění předmětu). Aby bylo jednoznačně identifikovatelné, kde program začíná a kde

[2] **Akční bod** – objekt ve scéně, ke kterému je možné přiřadit akci, kde akce se primárně vztahuje k aktivním objektům (např. robot).



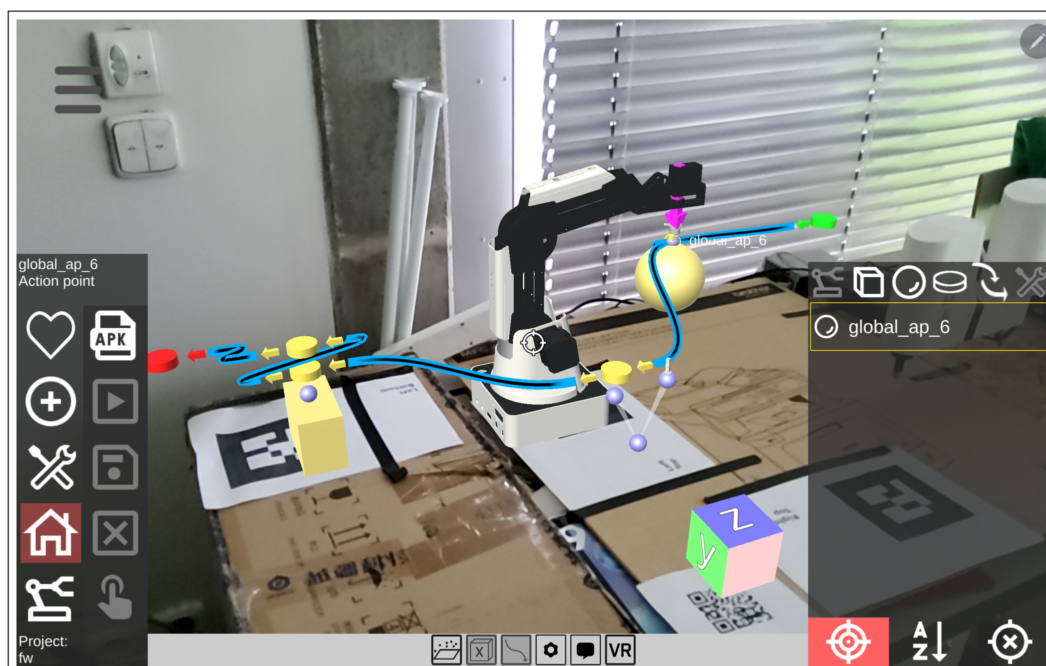
(a) Globální akční bod jako rodič bodů, kde levý má přiřazenou akci a pravý orientaci



(b) Akční bod v hierarchii, jehož předek je objekt robota

Obrázek 3.5: Příklad objektů a jejich hierarchie v projektu

končí, tak se v prostředí nachází dvě značky. První z nich definuje začátek propojení a druhá konec. V převodu do programovacího jazyku tyto značky znamenají začátek funkce `main` a `return`. Ilustrace propojených akcí v projektu je zachycen na obrázku 3.6.



Obrázek 3.6: Jednoduchý příklad projektu s propojenými akcemi

Akční body dovolují mimo přidání orientace a akcí také nastavení jednotlivých kloubů robota, jež pomáhají definovat pohyby robota, které v některých případech musí být jasně definované, protože velký robot se nemůže hýbat naprosto libovolně v malém prostoru nebo nekoordinovaně otáčet předmět, který drží.

Prostředí projektu sice neumožňuje hýbat s objekty, které jsou definované ve scéně, zde je však možné pohybovat akčními body a jim přiřazenými orientacemi. Toto umožňuje detailně definovat, jak se má robot chovat. Se změnou polohy akčního bodu souvisí i výše

zmíněná hierarchie (obrázky 3.5a a 3.5b). Jednak když uživatel aktualizuje polohu akčního bodu, tak se spolu s ním změní polohy všech jeho potomků a jednak když se uživatel pokusí smazat akční bod, který má potomky, tak mu to není dovoleno. Při změně polohy akčního bodu je logicky zneplatněno některé chování, například definice stavu kloubů robota, protože při změně pozice akčního bodu se logicky robot neumí již nastavit do požadované pozice.

3.1.2 Back-End – Server

Zmíněné body, které má splňovat kolaborativní aplikace (viz 2.1) platí také pro AREditor a jeho serverovou část. V tomto případě se zmíněné požadavky týkají právě back-endové části. Server v aktuální podobě splňuje 3 ze 4 bodů:

- **Paměť** – server ukládá data, která vytvoří uživatelé. Veškeré změny se prakticky okamžitě projevují na všech spuštěných editorech. Jelikož u rychle postřehnutelných změn se jedná převážně o změnu polohy objektu, je potřeba zajistit správné zobrazení této změny. Tuto správnost zajišťují 3D souřadnice, které jsou součástí každého objektu.

- **Uživatelé** – aplikace aktuálně sice nedělí uživatele do přístupových skupin, protože se jedná o kolaborativní programování, kde této funkcionality není zapotřebí. I přes to, že zde nejsou přístupové skupiny, je potřeba jednoznačně identifikovat uživatele a to minimálně z důvodu implementace výlučného přístupu. Pro tuto implementaci je vhodné znát, kdo je vlastníkem zámku na určitý objekt, aby byla zajištěna integrita dat. Této jednoznačné identifikace je server schopen.

- **Sezení** – celkový návrh AREditoru nepočítá s tím, že bude aktivních více sezení, což ovšem neznamená, že tento bod není splněn. Aplikace umožňuje otevřený vždy výlučně buď jeden projekt, nebo jednu scénu, která se zobrazuje všem aktivním i nově přihlášeným uživatelům. Toto je z důvodu, že se předpokládá samostatná instance serveru pro každé pracoviště, jelikož uživatelé programují v reálném čase robota, kterých je na pracovišti omezené množství. Sezení pro každého uživatele je tedy jednoznačně identifikovatelné.

Poslední bod **výlučný přístup** řeší tato práce a je detailněji popsán v následujících kapitolách.

Server se skládá z více služeb, které je potřeba spustit. Pro ilustraci jeho funkčnosti však dostačuje popsat následující:

- **ARServer** – centrální bod pro uživatelské rozhraní. Skrze tuto část probíhá veškerá komunikace mezi UI a serverem. Editor zde odesílá své požadavky, ARServer je dále zpracuje a následně na ně odpovídá.

- **Execution** – s touto službou udržuje ARServer perzistentní spojení. Jsou zde vyřizovány požadavky typu registrace/od-registrace zařízení k serveru či nahrávání, správa a spouštění exekučních balíčků[3].

- **Build** – vytváří exekuční balíčky.

- **Služba pro scénu a projekt** – ve službě pro projekt jsou uložené scény a projekty. Služba pro scénu spravuje kolizní objekty. Obě tyto služby vyvíjí firma Kinali[4]. Jelikož

[3]**Exekuční balíček** – kompilace scény a projektu do Pythonu
[4]https://www.kinali.cz/cs/

nejsou veřejně dostupné, nacházejí se zde ve formě Mocků. Mock je takový objekt (část systému), který poskytuje reálné výsledky a je využíván převážně pro testování [2].

Komunikace

Komunikace mezi editorem a serverem nebo mezi službami serveru je realizována pomocí RPC (Remote Procedure Call), což znamená spuštění procedury na vzdáleném zařízení pomocí síťového připojení [4]. Tento typ komunikace lze dále rozdělit do více kategorií, ze kterých server používá následující – REST API a WebSocket API.
REST API (Representational State Transfer) je architektura rozhraní, navržená pro distribuované prostředí. Implementuje čtyři základní metody (GET, POST, UPDATE a DELETE) [11]. Tento způsob je využíván výhradně na straně serveru a slouží pro komunikaci mezi službami (například mezi službami Execution a Build). Specifikum REST API je takové, že jedna strana komunikace zasílá požadavky a druhá na ně výhradně odpovídá (nikdy nezasílá požadavky).
WebSocket API – je technologie, která umožňuje vytvořit obousměrné komunikační prostředí mezi klientem a serverem [12]. Jak vyplývá z předchozí věty, tento způsob je využíván právě pro komunikaci mezi editorem a serverem, ale také mezi službami serveru.
Každé RPC má jasně definovanou strukturu, která může vypadat následovně:

```
@dataclass
class IdArgs(JsonSchemaMixin):
    id: str

class DeleteObjectType(RPC):
    @dataclass
    class Request(RPC.Request):
        args: IdArgs
        dry_run: bool = False

    @dataclass
    class Response(RPC.Response):
        pass
```

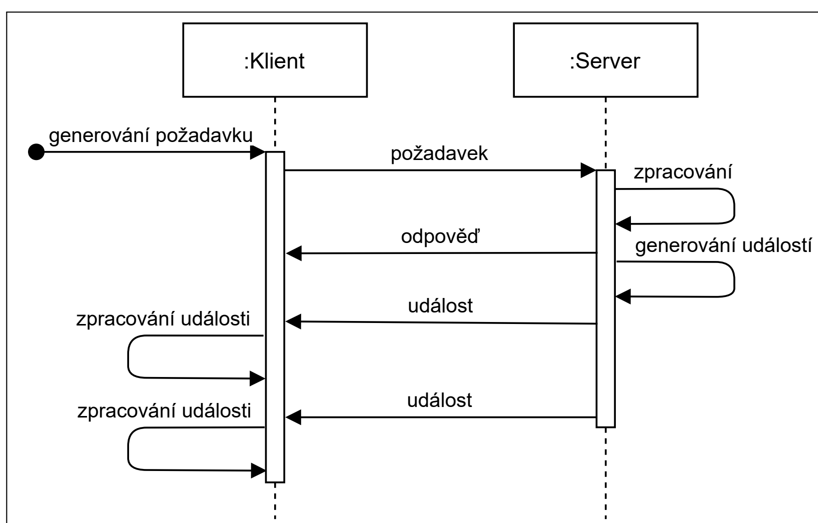
Kód popisuje strukturu RPC pro smazání objektu. Struktura je třídou `DeleteObjectType`. Třída `RPC`, jež je předkem, definuje základní vlastnosti RPC, tedy požadavek a odpověď. Tyto vlastnosti jsou rozšířeny o potřebné atributy, relevantní danému RPC. V tomto případě se jedná o atributy `args` a `dry_run`. V případě dodatečných argumentů požadavku není další popis potřeba. Atribut `dry_run` je ovšem zajímavější, protože se vyskytuje ve více typech RPC a jeho význam je stále stejný. Při jeho použití se jedná pouze o test, zda lze dané RPC úspěšně provést a žádná data nejsou modifikována. Význam má v tom, že editor na základě tohoto „testovacího“ výsledku (de)aktivuje některá tlačítka, čímž je používání editoru uživatelsky přívětivé.

Takováto definice struktury slouží primárně ke dvěma účelům:

1. **Typová kontrola** – v rámci zdrojových kódu projektu ARCOR2 je používána typová kontrola, která je validována pomocí knihovny Mypy[5]. Díky definici datových typů v deklaraci funkce a explicitně u proměnných, jejichž hodnota není jednoduchý datový typ (`int`, `string`, ...), je IDE schopno programátorovi napovídat všechny dostupné atributy, v případě práce s objektem a zároveň validovat datový typ, při práci s proměnnými. Tuto typovou kontrolu je možné pozorovat například za dodatečnou definicí atributů požadavku, kde `args` je typu `IdArgs`, tedy tento parametr musí být instance třídy `IdArgs`.

2. **Generování RPC pro AREditor** – díky definici struktury a požadavku ve třídě `RPC`, kde tyto třídy využívají dědičnosti speciální třídy (možno pozorovat i u třídy `IdArgs`), je možné vygenerovat Open API dokument. Open API je takový standard, kde dokumenty vytvořené podle této definice jsou strojově a lidsky čitelné a popisují požadavky zdroje bez nutnosti nahlížení do kódu [13]. Vzhledem k tomu, že se jedná standard, lze vygenerovanou definici datových tříd použít téměř pro libovolný jazyk.

Komunikace neprobíhá pouze stylem požadavek-odpověď, ale server dále generuje notifikace. Účelem těchto zpráv je informovat editor, popřípadě více editorů o změně, která nastala. V případě, že uživatel změní polohu například akčního bodu, vytváří se požadavek na server. Ten jej zpracuje a odpovídá, zda byl úspěšný. Server dále vytváří událost, popřípadě více událostí v závislosti na RPC, kterou odesílá všem editorům. Odesílání a počet notifikací je závislé na typu RPC. Diagram příkladové komunikace je zobrazen v 3.7.



Obrázek 3.7: Diagram komunikace mezi serverem a editorem.

[5]**Mypy** – https://mypy.readthedocs.io/en/stable/.

Kapitola 4

Návrh zamykání

Tato kapitola se zabývá problémy a prostředky pro řešení výlučného přístupu, který musí být řešen spolu s návrhem chybějící implementace. Toto téma je více rozvedeno v sekci 4.1. Sekce 4.2 pojednává o významné technologii, která tvoří jádro celého serveru. Následně, když jsou známy obecné problémy a technologie, v které mají být řešeny, je možné sumarizovat požadavky, které musí výsledné řešení splňovat (sekce 4.3). Poslední část této kapitoly je věnována návrhu řešení, které bude splňovat všechny požadavky, viz 4.4.

4.1 Vzájemné vyloučení přístupu ke sdíleným zdrojům

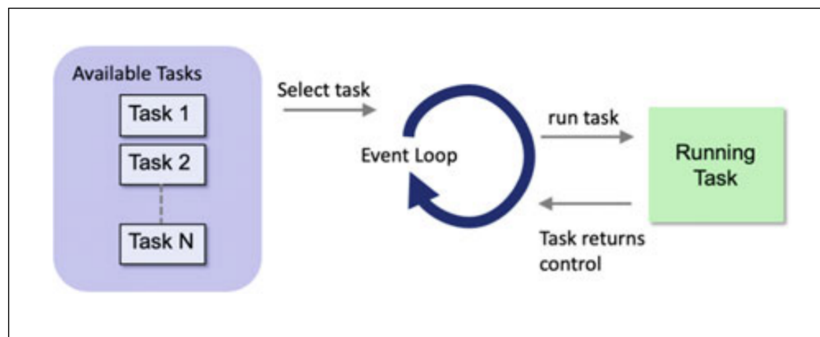
Z důvodu rychlosti by měla kolaborativní aplikace dovolit konkurentní zpracování požadavků. Za tímto účelem je nutné zavádět zamykání zdrojů. Toto zamykání zajistí, že zdroj může v jednu chvíli používat výlučně jedna korutina, která zpracovává požadovaný kód. Pokud je dodržen tento předpoklad, nemůže nastat nekonzistentní stav sdíleného zdroje. Pro zajištění výlučného přístupu může být využito více prostředků. Volba prostředku závisí na daném problému. Dostupnými prostředky mohou dle [8] být:

- **Zámek** – objekt, který může v jednu chvíli vlastnit pouze jeden proces. Užití zámku je vhodné v takovém případě, potřebujeme-li zajistit, že daný zdroj bude využívat vždy právě jeden proces. Zámek je nezbytně nutné použít například při paralelním aktualizování hodnoty jedné proměnné.
- **Semafor** – objekt, který může v jednu chvíli vlastnit více než jeden proces. Užití semaforu je vhodné například pro problém večeřejících filozofů [3].
- **Podmínka** – objekt, který slouží k synchronizaci více procesů. Jeden proces čeká na podmínku, kterou aktivuje jiný. Užití podmínky je vhodné například u problému „producent-konzument“.
- A jiné.

4.2 Knihovna Asynchronous I/O

Tato sekce čerpá z [8, 15]. Asynchronous I/O neboli AsyncIO je knihovna, která umožňuje vytvářet konkurentní aplikace v Pythonu. Problémy řešené pomocí této knihovny jsou rozděleny do úkolů, které se řeší asynchronně. Je potřeba zmínit, že řešení není vystavěno na základě více-vláknové nebo více-procesové teorie, ale jedná se o myšlenku kooperativního multitaskingu. Díky tomuto řešení je možno se vyhnout složitým a časově drahým zámkům, které je nutno používat u více-vláknových aplikací, protože celý výpočet řeší jediný proces. Další výhodou AsyncIO je, že k přepnutí kontextu mezi úkoly dochází pouze v definovaných místech (viz dále), zatímco u vláken může docházet k přepnutí kontextu kdykoliv. I přes to, že složitější více-vláknové zámky nejsou potřeba, nelze je vynechat úplně. Z důvodu asynchronního řešení úkolů je stále potřeba zamykat sdílené zdroje, avšak v rámci jednoho procesu se jedná již o jednodušší i časově levnější řešení.

Zpracování vytvořených úkolů je prováděno pomocí smyčky událostí. Základní princip fungování této smyčky je takový, že smyčka vybere úkol, který je připraven ke zpracování a spustí jej. Spuštěný úkol má v rámci programu plnou kontrolu nad procesorem, dokud se jej sám nevzdá (např. dokončením úkolu nebo čekáním na data). Jakmile má řídící smyčka opět kontrolu nad procesorem, poznamená si stav úkolu a vybírá následující ke spuštění. Tento proces je zachycen na obrázku 4.1.



Obrázek 4.1: Řídící smyčka v AsyncIO [8]

AsyncIO dále zavádí klíčové slova `async` a `await`, jejichž použití je podmíněno spuštěním výše zmíněné smyčky. Použití `await` znamená, že se daná funkce vzdává procesoru a čeká na dokončení nějakého úkolu. Zatímco `async` je klíčové slovo použité před definicí funkce a znamená, že daná funkce je spuštěna jako úkol. Volání takovéto funkce je podmíněno tím, že před voláním musí být použito klíčové slovo `await`. Tyto úkoly se nikdy nevytvářejí dopředu, jejich vytvoření je podmíněno použitím `await` nebo explicitním spuštěním jednoho či více úkolů.

4.2.1 Zámek v AsyncIO

AsyncIO potřebuje občas ke své činnosti zámky k zajištění výlučného přístupu. Tato potřeba je odvozena již z asynchronního prostředí. AsyncIO disponuje řadou prostředku, k zajištění tohoto požadavku. Pro ilustraci, implementuje všechny prostředky zmíněné v sekci 4.1, ale také další[1].

Kdy a kde použít výlučný přístup? Synchronizační prostředky je potřeba používat kdykoliv, když je možné volat jednu funkci vícekrát nebo více funkcí přistupuje k jedné proměnné a zároveň součástí funkce je nějaké asynchronní volání. Taková funkce může vypadat následovně:

```
import asyncio

async def foo():
    ...  # modifikace sdíleného prostředku

    # volání funkce, která vyžaduje aktuální modifikaci sdíleného
    #  prostředku
    await bar()
```

V tomto případě se uvažuje, že na funkci `foo` může čekat více úkolů. Při této implementaci ovšem může docházet k vzájemnému přepisování sdíleného prostředku mezi úkoly. Správné řešení by mohlo vypadat například následovně:

```
import asyncio

lock = asyncio.Lock()

async def foo():
    async with lock:
        ...  # modifikace sdíleného prostředku

        # volání funkce, která vyžaduje aktuální modifikaci sdíleného
        # prostředku
        await bar()
```

Tento kód zajistí, že proměnná bude vždy obsahovat správné informace. Jedná se však pouze o ilustrativní příklad, správných řešení je více.

[1]Synchronizační prostředky v AsyncIO – https://docs.python.org/3/library/asyncio-sync.html

4.3 Požadavky pro návrh výlučného přístupu

Jelikož se jedná o rozsáhlý kód, je nutné jej nejprve nastudovat do hloubky a vyhledat problémy, které by mohly nastávat. Tímto studováním byly problémy roztříděny do dvou základních bodů:

Bod 1. **Zamykání objektů v rámci projektu a scény** – Jak scéna tak projekt slouží k práci s uživatelem definovanými prvky. V případě scény jsou prvky veškeré vložené objekty (například robot). V případě projektu jsou prvky převzaty jednak ze scény a jednak jsou vytvářeny další (například akční body). U aktualizace těchto prvků je potřeba, aby jeden objekt vždy aktualizoval výhradně jeden uživatel.

Bod 2. **Kompletní uzamčení v rámci práce se scénou či projektem** – Je potřeba umožnit kompletní uzamčení editoru. V tomto případě se musí jednat o atomickou operaci, která slouží například pro uložení či uzavření scény/projektu. V případě ukládání nemůže být umožněna žádná práce s objekty, neboť by uložení nemuselo proběhnout úspěšně. Obdobně při uzavírání.

Bod 1 je dále příliš obecný, proto je vhodné jej dále rozdělit na následující části:

Část 1. **Zamykání v rámci jednoho požadavku** – Editor zasílá RPC (např. aktualizaci parametru objektu), které lze zpracovat v rámci funkce.

Část 2. **Zamykání v rámci více požadavků** – Editor zasílá několik RPC (např. pohyb s objektem a spojené kontroly), které vyžadují stálé uzamčení jednoho objektu.

Toto jsou základní problémy, které musí výsledný návrh zohledňovat a v ideálním případě i řešit. Základní ale neznamená všechny. Co když uživatel vyvolá některé z RPC, zamkne jím objekt ale následně už neprovede akci, která jej odemkne? Například z důvodu, že mu aplikace nebo OS přestane pracovat. Nebo zařízení odloží na stůl a odejde? Ostatní uživatelé jsou omezováni a nemohou pracovat se zamčenými objekty. Z tohoto důvodu by měly být zámky časově omezené.

Dalším problémem mohou být zmíněné hierarchie objektů. V případě, že uživatel chce pohybovat celým stromem akčních bodů, nedává smysl, aby jiný uživatel pohyboval částí tohoto stromu. Návrh tedy musí umožňovat uzamčení celé hierarchie od kořenových objektů až po listové.

4.3.1 Demonstrace současných problémů v kódu

V následující části demonstruji a popíši výše zmíněné požadavky s odkazem do kódu, ve kterém mohou vzniknout problémy, které bude řešit konkrétní požadavek. Jedná se pouze o výňatek kódu, tudíž nejsou řešeny importy a související funkce.

Kompletní uzamčení v rámci práce se scénou či projektem (Bod 2):

```
async def close_project_cb(req: srpc.p.CloseProject.Request, ui: WsClient
                           ) -> None:
    can_modify_scene()

    assert glob.PROJECT

    if not req.args.force and glob.PROJECT.has_changes:
        raise Arcor2Exception("Project has unsaved changes.")

    if req.dry_run:
        return None

    await close_project()
    return None
```

Tento kód je vykonáván při zavírání projektu. Kritickou sekcí je asynchronní volání funkce `close_project`. V případě, že kterýkoliv z uživatelů aktualizuje v tuto nevhodnou chvíli projekt, tak jeho provedené změny nemusí být uloženy. Tato chyba může nastat z důvodu, že uživatel již není schopen zajistit přerušení RPC, jež uzavírá projekt a projekt se tedy uzavře bez uložení této změny.

Zamykání v rámci jednoho požadavku (Bod 1. část 1.):

```
async def rename_project_cb(
        req: srpc.p.RenameProject.Request, ui: WsClient
) -> None:

    unique_name(req.args.new_name, (await project_names()))

    if req.dry_run:
        return None

    async with managed_project(req.args.project_id) as project:

        project.name = req.args.new_name
        project.update_modified()

        ...  # notify UI about project rename
```

Tento kód je vykonáván při přejmenování projektu. Problém zde může nastat při samotné realizaci přejmenování. Dva uživatelé v jednu chvíli mění jména dvou projektů a oba současně použijí stejný název. Nejprve se asynchronně zjistí všechna jména existujících projektů, následně se spustí tatáž kontrola pro druhého uživatele. Obě kontroly proběhly úspěšně, tedy název neexistuje. Avšak při samotné realizaci přejmenování vzniknou dva projekty se stejným jménem, což samozřejmě není dovoleno.

Zamykání v rámci více požadavků (Bod 1. část 2.):

```
async def update_action_point_position_cb(
        req: srpc.p.UpdateActionPointPosition.Request, ui: WsClient
) -> None:
    assert glob.PROJECT

    ap = glob.PROJECT.bare_action_point(req.args.action_point_id)
    if req.dry_run:
        return

    await update_ap_position(ap, req.args.new_position)
```

Tento kód je vykonáván při aktualizaci polohy akčního bodu. První uživatel označí bod a začne s ním pohybovat. Všechno funguje bez problému. Občas se vyšle RPC, které provádí kontroly, zda je aktuální poloha bodu v pořádku, ale server odpovídá, že ano. V jednu chvíli však druhý uživatel bod smaže, protože jej považuje za nepotřebný. Prvnímu uživateli najednou označený bod zmizí. Obdobný problém může nastat například při přejmenovávání.

4.4 Návrh výlučného přístupu

V tuto chvíli jsou známy veškeré požadavky, které musí být vyřešeny. Vzhledem k tomu, že objekt v UI je reprezentován třídou a hierarchie objektů znamená uspořádání těchto objektů do stromové struktury, kde každý potomek zná svého rodiče, je možné elegantně vyřešit zamykání celé stromové struktury. Jednoduchá řešení často bývají velmi efektivní a umožňují jen malé množství chyb. Z tohoto důvodu jsem zvolil umístit zámek celého stromu do jeho kořene, tedy do třídy, která neodkazuje na žádného předka. Následně je potřeba vyřešit zamykání jednotlivých objektů. Toho lze dosáhnout dvěma způsoby, kde každý z nich má určité výhody, ale také nevýhody:

1. **Zamykat jednotlivé objekty** – Každý objekt obsahuje proměnnou/objekt, jejíž obsah definuje, zda je zamčený a jakým způsobem.
    + Správa zámku je dostupná v rámci objektu.
    + Potřeba definovat takovýto parametr pro každý typ objektu. Jednoduchá implementace v rámci dědičnosti.
    – Zámky v podobě objektu mohou zabírat více paměti.
    – Potřeba do-implementovat kompletní stromovou strukturu s vkládáním a odebíráním prvků. Součástí struktury budou tedy i vazby na potomky, aby bylo možno zjistit, zda lze zamknout celý strom.

- Zjištění, jestli lze zamknout celý strom, vede na exponenciální časovou složitost. Je potřeba projít N uzlů pro zjištění rodiče a následně prohledat celý strom, jestli některý z objektů není zamčený.

2. **Vytvořit centrální „databázi" pro správu zamčených objektů** – Na míru navržená databáze může mít velmi rychlou odezvu, protože nemusí prohledávat celý strom. Příklad takové databáze je popsán v sekci 4.4.
   + Databáze je na jednom místě.
   + Zjištění, zda lze zamknout celý strom se složitostí $\mathcal{O}(N)$ – nutnost projít N uzlů pro zjištění rodiče a následně v konstantním čase zjistit, zde obsahuje nějaké zamčené objekty.
   – Databáze vyžaduje výlučný přístup pro modifikaci. Může zpomalovat aplikaci při více požadavcích.
   – Potřeba implementovat celou databázi i s kritickou sekcí.

Pro implementaci prvního bodu se zdá být vhodná knihovna aiorwlock[2]. V tomto případě by každý objekt obsahoval jedinou instanci zámku z této knihovny a parametr, zda je zamčený celý strom. Implementace druhého bodu, jak již bylo zmíněno, začíná od základu.

Složitost implementace nehraje roli, protože výše zmíněné způsoby vyžadují velmi podobnou složitost zásahu. Z tohoto důvodu lze způsoby porovnávat z pohledu času. První způsob může vést až na exponenciální složitost, zatímco druhý je vždy lineárně složitý. Tento fakt má asi největší váhu v rozhodování, proč vlastně zvolit druhý způsob.

Struktura pro centrální správu zámků

Základem ukládání dat o zamčených objektech je pevně definovaná struktura, která obsahuje vše potřebné. Následující struktura, v podobě třídy, reprezentuje vždy jeden kořenový objekt. Z této struktury je čitelné, jestli je kořenový objekt zamčený sám, jako celý strom, nebo jsou zamčeny objekty v tomto stromu a jakým způsobem.

```
class LockStruct:
    class LockData:
        owners: List = []
        timestamp: str = None

    # Slovník uzamčených objektů se strukturou <id objketu: data>
    read: Optional[Dict[str: LockData]] = None
    write: Optional[Dict[str: LockData]] = None
    tree: bool = False
```

Jedna tato struktura je uložena jako hodnota slovníku. Pro její okamžité nalezení je použito ID kořenového objektu, jako klíč. Pokud je potřeba zamknout celý strom, stačí nastavit hodnotu `tree` na `True`. V případě zámku pro čtení nebo pro zápis stačí vložit ID objektu do příslušného slovníku jako klíč a jako hodnotu vyplnit strukturu `lock_data`. Ta vyžaduje přidání vlastníka do pole a nastavení aktuálního času. Pole vlastníků je opravdovým polem

[2] https://github.com/aio-libs/aiorwlock

pouze u zámku pro čtení, který umožňuje víc vlastníků. V případě zámku pro zápis je v poli vždy výhradně jeden prvek. Z délky tohoto pole lze dále vyčíst i počet vlastníků, který může být použit například při signalizaci, proč nelze daný objekt zamknout pro zápis. Způsob uložení vlastníka zámku pro zápis by mohl být zjednodušen pouze na textovou hodnotu, protože zámek neumožňuje rekurzivní uzamčení. Toto řešení by ovšem vedlo na druhou, téměř duplicitní strukturu.

Pro zajištění výlučného přístupu je potřeba, aby tato databáze (slovník ve kterém jsou uchovávány tyto data), byla opatřena zámkem. Kritická sekce této databáze bude řešena vytvořením veřejných metod pro uzamčení a odemčení objektu. Samotná databáze pak bude neveřejná. V ideálním případě je potřeba vytvořit jak metody pro uzamčení, tak funkce, které tyto metody budou používat. V případě funkcí se bude jednat o kontextový manažer, který zajistí uzamčení, vykonání kritické sekce a následné odemčení.
Metoda pro uzamčení objektu, jak pro čtení, tak pro zápis, by měla v pořadí vykonávat následující kroky:

1. Najdi kořenový objekt stromu, ve kterém se nachází zamykaný identifikátor.
2. Požádej o výlučný přístup k databázi.
3. Zjisti, zda je možné objekt zamknout.
4. Pokud ne, signalizuj chybu. Jinak pokračuj.
5. Vlož záznam o zámku a signalizuj úspěch.

Tímto dojde k uložení informace o zamčeném objektu a je možné s ním manipulovat dle potřeby a typu zámku. Dále byl zmíněn kontextový manažer. Tato funkcionalita umožňuje spustit dvě související operace jako pár a navíc vykonat jinou část kódu mezi těmito operacemi [9]. K jeho užití slouží známé klíčové slovo `with`. Kontextový manažer je záhodno použít ve chvíli, kdy je potřeba vykonat akce s jedním objektem v rámci jednoho RPC (viz 4.3.1) a měl by vykonávat následující kroky:

1. Postupně zamykej všechny vstupní objekty za pomocí zamykací metody.
2. Pokud zamykání selže, uvolni zamčené zdroje a signalizuj chybu. Jinak pokračuj.
3. Předej řízení uživatelskému kódu (klíčové slovo `yield`).
4. Uvolni uzamčené prostředky bez ohledu na výsledek kroku 3.

Co se týče uzamčení objektu přes více RPC, tak zde je nutné vytvořit speciální RPC, které bude potřeba spustit vždy před prací s požadovaným objektem. Pro vynucení tohoto volání je vhodné kontrolovat, zda je objekt zamčený dříve, než se s ním začne pracovat.

V poslední řadě je potřeba zajistit životnost zámků. K tomuto účelu je vhodné při spuštění serveru vytvořit úkol, který bude například určitou dobu spát a po probuzení zkontroluje všechny zámky, případně některé uvolní. Na konci naplánuje další spuštění sama sebe. Součástí uvolnění musí být i notifikace editoru, že byl zámek uvolněn, aby se zamezilo případným chybám.

4.4.1 Příklady zamykání

Tato sekce ukazuje, jak metody implementované podle výše specifikovaných požadavků použít na zmíněných problémech.

Příklad, kdy je potřeba zamknout celou databázi:

```
async def close_project_cb(req: srpc.p.CloseProject.Request, ui: WsClient
                           ) -> None:
    async with get_db_lock():
        can_modify_scene()

        assert glob.PROJECT

        # Kontrola, zda není nic zamčeno
        if get_all_locked_objects():
            raise LockingException("Something locked")

        if not req.args.force and glob.PROJECT.has_changes:
            raise Arcor2Exception("Project has unsaved changes.")

        if req.dry_run:
            return None

        await close_project()
```

V tomto konkrétním případě nevadí, že se uvnitř zámku nachází asynchronní volání, protože je uzamčen veškerý přístup k databázi. Dále byl přidán řádek, který kontroluje, zda je možné projekt uzavřít. Nedává smysl zavírat projekt, pokud v něm některý z aktivních uživatelů pracuje.

Příklad, kdy je potřeba, aby byl objekt uzamčen ještě před vykonání akce s ním:

```
async def update_action_point_position_cb(
        req: srpc.p.UpdateActionPointPosition.Request, ui: WsClient
) -> None:

    assert glob.PROJECT

    # Kontrola, zda je objekt uzamčen editorem reprezentovaným 'ui'
    await ensure_locked(req.args.action_point_id, ui)

    ap = glob.PROJECT.bare_action_point(req.args.action_point_id)
    if req.dry_run:
        return

    await update_ap_position(ap, req.args.new_position)
```

V tomto případě je kontrola jednoduchá, stačí přidat podmínku s kontrolou, zda je objekt skutečně uzamčen. V případě, že není, je vyhozena výjimka. Tento kousek kódu reprezentovaný asynchronní funkcí `ensure_locked` zajistí, že volání akce nad nezamčeným objektem skončí chybou, kterou by měl vhodně vypsat editor.

Příklad, kdy je potřeba zamknout objekt, v rámci jednoho požadavku:

```
async def rename_project_cb(
        req: srpc.p.RenameProject.Request, ui: WsClient
) -> None:

    # uzamčení ID scény editorem 'ui'
    async with context_write_lock(req.args.scene_id, ui):
        unique_name(req.args.new_name, (await project_names()))
        if req.dry_run:
            return None

        async with managed_project(req.args.project_id) as project:
            project.name = req.args.new_name
            project.update_modified()
            evt = sevts.p.ProjectChanged(project.bare)
            evt.change_type = Event.Type.UPDATE_BASE
            asyncio.ensure_future(notif.broadcast_event(evt))
    return None
```

V tomto konkrétním případě také nevadí, že se uvnitř zámku nachází další asynchronní volání, protože je zamčená celá scéna. V případě, že se jinému uživateli podaří otevřít tuto scénu a vložit do ní nový objekt, nebude mu umožněno ji dále uložit.

Kapitola 5

Implementace

Na základě návrhu z kapitoly 4.4 jsem vytvořil prvotní objekt zámku, který umožňoval základní uzamykání objektů, tedy pro čtení a pro zápis. Tento objekt a postupné vylepšování o metody, které jsou nezbytné pro správné fungování výlučného přístupu na celém serveru, popisuje sekce 5.1. Sekce 5.1.1 popisuje obecně způsoby, jakými byl zajištěn výlučný přístup pro všechny dostupné RPC a problémy, které jsem při této implementaci řešil. Následně představím vylepšení, která jsem implementoval v rámci vývoje a která zjednodušují práci uživatelům AREditoru (viz 5.1.2).

V sekci 5.2 představím databázi uživatelů, kterou jsem implementoval spolu se zámkem. Tato databáze slouží primárně k tomu, aby byli vlastníci zámků pojmenovaní a jejich jméno bylo možné zobrazit v UI, pro lepší přehled uživatelů. Se správou uživatelů úzce souvisí i jejich notifikování o případných změnách spojených se zámkem, které popisuji v sekci 5.2.1.

Po představení celé implementace a problémů, které jsem řešil, vyhodnotím použitelnost a validitu implementace na testováních, které proběhly (viz 5.5). Implementace byla sice úspěšně dokončena, avšak i v tomto případě je prostor pro zlepšení, viz 5.3. V poslední sekci popíši, jak jsem vytvořené kódy zdokumentoval, aby se v nich dokázal orientovat kdokoliv, kdo bude potřebovat funkcionalitu zámku vylepšit.

5.1 Zámek a jeho části

Samotný balíček zámku se skládá z několika modulů. Balíček je složka, která obsahuje soubor `__init__.py` a množinu modulů, kde modul je `.py` soubor [16]. Adresářová struktura tohoto balíčku je následující:

```
lock/
├── __init__.py
├── exceptions.py
├── lock.py
├── notifications.py
└── structures.py
```

Strukturu jsem vytvářel primárně z důvodu, že programování pouze v jednom souboru je chaotické a v tomto souboru je obtížné se orientovat. V tomto případě jsem kód logicky

rozdělil na modul výjimek, kódu pro notifikace, struktury zámku a nakonec samotný modul zámku, kde poslední dva zmíněné moduly jsou klíčové pro tuto sekci.

Modul `structures.py` obsahuje dvě struktury. První z nich se týká notifikací a bude představena později. Druhá přímo souvisí se zámkem a její základ již byl představen v 4.4. Ve finální verzi (viz obrázek 5.1) jsou úpravy pouze nepatrné. Mimo jiného pojmenování zmizela zanořená struktura `LockData` a vlastníci zámků, respektive vlastník zámku v případě práva pro zápis, byli přesunuti jako hodnota slovníku `read` či `write`. Tyto změny byly vyvozeny na základě pozdějšího vyjasnění, jak se má chovat časové omezení zámků. Po odstranění časového razítka v původní struktuře zbývala jediná položka a udržovat pro ni zvláštní objekt je redundantní.

LockedObject

read: dict

write: dict

tree: bool

read_lock()

read_unlock()

write_lock()

write_unlock()

check_upgrade()

check_downgrade()

Obrázek 5.1: Hlavní struktura pro ukládání zámků

Struktura `LockedObject` (`LockStruct` v návrhu) má tedy až na změnu uložení vlastníka zámku totožný formát. Pro zjednodušení struktury modulu `lock.py`, jež se volá z více míst serveru, zde byly dále doplněny metody pro uzamčení a uvolnění zámku a to jak pro čtení, tak pro zápis. Všechny tyto metody obsahují primárně kontroly, zda může být získán požadovaný typ zámku, tedy zda zámek již není uzamčen nebo zda případně podporuje vícero uzamčení. V případě odemykání jsou kontroly primárně směřovány na to, zda je objekt opravdu uzamčen či jestli jej odemyká jeho skutečný vlastník. Při uvolnění zámku jsou následně odstraněny všechny data ze struktury, aby nedocházelo k falešně pozitivním identifikacím zamčených objektů. Pro zrychlení přístupu do struktury jsem definoval sloty pomocí `__slots__`. Podle [16] využívá tato vlastnost jinou konstrukci atributů objektu v Pythonu, jež neumožňuje následné rozšiřování instance, což pro tento případ ani není nutné. Výhodou této konstrukce je nižší paměťová náročnost a rychlejší přístup k atributům.
Dále jsem zde implementoval metody pro kontrolu, zda může být zámek upraven. Úpravy jsou v této chvíli možné dvě:

1. **Povýšení** – v případě, že je uzamčen pouze jediný objekt, lze zámek povýšit na zámek celé stromové struktury, ve které se uzamčený objekt nachází. Metoda analogicky obsahuje kontroly, zda toto povýšení může provést uživatel, jež vyvolat RPC.

2. **Ponížení** – v případě, že je uzamčena celá stromová struktura objektů, lze zámek snížit pouze na zámek jediného objektu. Kontroly jsou opět analogické.

Tyto kontroly byly implementovány později, jako vylepšení, protože bez funkcionality aktualizace zámku musel uživatel objekt odemykat a znovu zamykat při změně scénáře práce s objektem[1].

Další modul `lock.py`, je klíčový pro funkcionalitu výlučného přístupu k objektům rozšířené reality. Tento modul je inicializován při spuštění serveru a jeho aktivní instance se udržuje po celou dobu. Tato instance shromažďuje veškeré informace o zámcích (jaký objekt je uzamčen, jakým způsobem, kdo je vlastníkem zámku), umožňuje tyto informace rozšiřovat, vhodným způsobem je interpretovat a také mazat. Veškerá tato funkcionalita

[1]**Scénář práce s objektem** – práce s objektem je možná ve více scénářích např. posouvání v prostoru, práce v menu objektu, přidávání logických akcí, aj.

je dostupná pomocí veřejných metod. Nejdůležitější, či nejčastěji používané metody jsou následující:

- `read_(un)lock` a `write_(un)lock` – Metody jsou používány bez výjimek v rámci každého RPC, které vyžaduje výlučný přístup k objektu, ať už je objekt zamčen jen v rámci serveru či požadavkem z editoru. Hlavním cílem těchto metod je zajistit, že při zamykání či odemykání nevznikne žádná nekonzistentní situace, která by způsobila v budoucnu problém. Jinými slovy zámek může vzniknout jen v případě, pokud daný identifikátor již není označen za uzamčený, jedině vlastník může odstranit zámek a po každém uvolnění nezůstane v databázi zámků žádný pozůstatek (prázdná množina, aj.).

- `is_write_locked` – Metoda kontroluje, zda-li je objekt zamčen pro zápis konkrétním uživatelem. Využití našla v případech, kdy je objekt zamčen již z editoru, ale vykonávaná akce stále nebyla dokončena (např. přejmenování, kde editor zamkne objekt a zasílá požadavky na kontrolu duplicity jména). Kód této metody nekontroluje pouze přímé uzamčení objektu, ale také zda-li se objekt nenachází v již zamčeném stromu.

- `get_lock` – V tomto jediném případě se nejedná o metodu, ale o kontextového manažera. Vykonávaný kód uzamkne celou databázi zámků výlučně pro jediného vlastníka, tudíž z ní nemůže nikdo ani číst a předá řízení funkci, ze které je volán. Po dokončení činnosti funkce celou databázi opět odemkne. Tento kontextový manažer je vhodný zejména pro RPC, která nějakým způsobem manipulují s projektem či scénou a při této manipulaci nesmí být provedena žádná změna (např. uložení či zavření projektu), z čehož vyplývá, že výlučný přístup nelze získat, pokud existuje nějaký aktivní zámek pro zápis.

- `get_write_locks_count` – Tato metoda je především využita s předchozím kontextovým manažerem. Kód této metody přistupuje do databáze zámků a počítá, kolik objektů je zamčeno pro zápis. V příkladech z minulého bodu nedává smysl, aby uživatelé nějakým způsobem aktualizovali stav prvků v projektu, pokud je již naplánováno uzavření projektu. V případě, kdy tato metoda vrací nenulový počet výsledků nemůže být předchozí kontextový manažer získán, což je signalizováno výjimkou.

- `get_root_id` – Snad nejdůležitější metoda celého zámku je právě tato. Metoda obstarává korektní nalezení kořenového objektu stromu, ve kterém se nachází prvek, jež chce uživatel zamknout. Jak jsem zmínil v návrhu a budu popisovat v 5.1.1, do databáze zámků se vkládá právě tento kořen a uzamčený objekt, z důvodu rychlosti prohledávání. Metoda je nezbytná, protože objekty scény a projektu nemají totožnou strukturu a pouze některý z těchto objektů obsahuje informaci o svém předku.

- `update_write_lock` – Tato metoda zajišťuje změnu kořenového objektu v databázi zámků pro již uzamčený objekt. Vhodná je zejména v případě, kdy se mění rodič akčního bodu. Vzhledem k tomu, že akční bod musí být před touto manipulací uzamčen, tak bez tohoto kódu by následovala výjimka, že daný objekt nelze odemknout, protože není nalezen v databázi. Tato chyba by úzce souvisela se strukturou databáze, tedy že objekt je uzamčen skrze svého nejstaršího předka.

- `update_lock` – Metoda má velmi podobný název s předchozí, avšak provádí úplně jinou činnost. V tomto případě se jedná o povýšení či ponížení zámku, které jsem zmiňoval dříve. Tento obecný název jsem volil z toho důvodu, že aktuální modul

obsahuje již mnoho metod, tudíž jsem nechtěl vytvářet jednu pro povýšení a druhou pro ponížení.

V případě zmíněných metod se jedná pouze o nejvíce používané nebo jiným způsobem zajímavé. Modul `lock.py` disponuje mnohem větším počtem metod, kde přibližně polovina z nich je privátních. Tyto privátní metody neobsahují žádnou speciální funkcionalitu a jedná se především o metody pojmenované podobně s výše zmíněnými. Tato nezanedbatelná redundance má prostý důvod. Veřejná metoda slouží nejčastěji pro asynchronní získání dat, které jsou nezbytné pro následný, privátní, kód a dále pro získání výlučného přístupu k databázi zámků. Privátní metoda následně reprezentuje kritickou sekci, kde by se nemělo nacházet žádné asynchronní volání a to ze dvou důvodů:

1. **Uváznutí** – v případě, že kritická sekce asynchronně zavolá veřejnou část zámku či jiný kód, který vyžaduje výlučný přístup k databázi zámků, celý server se dostává do stavu uváznutí a nemůže dále pokračovat. V tomto případě jsem neimplementoval žádnou detekci uváznutí, neboť se nejedná o složitý kód. Za účelem předcházení tomuto problému jsem vytvořil manuál, jak přistupovat do kritické sekce, který se nachází v souboru `README.md`, který je součástí balíčku zámku.

2. **Ztráta výkonu** – v případě, že programátor použije asynchronní volání uvnitř kritické sekce a shodou okolností se mu podaří server nezablokovat, dostává se nejčastěji do situace, kdy je výkon serveru značně degradován. V této situaci nemůže být obslouženo téměř žádné RPC (protože většina vyžaduje zamykání) a celý server čeká na dokončení právě této asynchronní operace.

Modul zámku obsahuje i několik vlastností (angl. property), z nichž nejdůležitější jsou `scene` a `project`. Jak již vyplývá z názvu, jedná se o instanci aktivní scény v případě, kdy je otevřená scéna, či scény a projektu v případě otevřeného projektu. Obě tyto vlastnosti byly původně globální proměnné a implementací zámku, který je rovněž globální proměnnou, mohly být přesunuty zde. Přesun mohl být realizování z důvodu, že zámek obě tyto vlastnosti často využívá pro nalezení kořene (viz metoda `get_root_id`) a také proto, že instance zámku je aktivní po celou dobu běhu serveru.

5.1.1 Zamykání a RPC

Z výše představených metod a struktur nemusí být na první pohled jasné, jak samotný zámek funguje. Stěžejní logiku, kterou jsem navrhl v 4.4, jsem použil beze změn. Struktura tedy obsahuje kořenové objekty, skrze které lze zjistit, zda je nějaký z potomků uzamčen. Klíčem k rychlosti prohledávání této struktury jsou Pythonovské objekty datového typu slovník, které umožňují velmi rychlý přístup k hodnotě skrze klíč. Časová složitost přístupu do slovníku je nejčastěji $\mathcal{O}(1)$ a v nejhorším případě $\mathcal{O}(N)$ [18].

V případě vkládání (uzamykání) jediného objektu je potřeba dodržet obecné principy zámků, tedy že daný objekt nesmí být uzamčen pro zápis v případě zámku pro čtení, či je zcela odemčen v případě zámku pro zápis. Pokud jsou tyto podmínky splněny, je do databáze vložen slovník s identifikátorem kořenového objektu, jehož hodnotou je struktura, která uchovává zamčené potomky a informaci, jakým stylem jsou uzamčeny. V případě uzamykání celého stromu se dále kontroluje, zda již není uzamčen kterýkoliv z potomků kořenového objektu. Tato kontrola je již jednoduchá, protože se stačí podívat, zda identifikátor kořenového objektu existuje v databázi či nikoliv. V případě odemykání postačí kontrola vlastníka zámku s následným uvolněním záznamu z databáze.

Zamykání editorem

Pro účely uzamykání objektu z editoru jsem vytvořil čtyři RPC. Pro uzamčení a odemčení, každé ve dvou variantách pro čtení a zápis. Každé s těchto RPC obsahuje parametr pro identifikování objektu, který má být uzamčen. RPC pro uzamčení pro zápis obsahuje navíc parametr, zda má být uzamčen celý strom. Tato možnost není dostupná v případě zámku pro čtení, protože v aktuálním prostředí nedává smysl. V případě odemykání parametr na odemčení stromu také chybí. Tato informace není pro odemykání potřebná, protože když je uzamčen celý strom, tak se v databázi nachází pouze identifikátor kořenového objektu, uzamčeného objektu a informace, že je uzamčen celý strom. Dále jsem vytvořil RPC pro aktualizaci zámku, které opět obsahuje identifikátor objektu a dále typ aktualizace. Typy jsou v tuto chvíli dva, viz 5.1.

Kontextový manažer

Dále jsem implementoval kontextový manažer, který podporuje uzamčení jednoho či více objektů. Funkce volá standardní metody zámku pro uzamykání a odemykání objektů, tudíž přístup k databázi zámků je pouze z jednoho místa. Funkce je klíčová v případech, kdy server vykonává kód RPC, jež pracuje s objektem, který není uzamčen z editoru. Ve všech případech se v kódu RPC nachází asynchronní volání, tudíž je potřeba využít kontextového manažeru a dotyčný objekt uzamknout. Nemůže se ovšem jednat o klasický kontextový manažer, který uzamkne objekty, vykoná kód a objekty odemkne. Důvodem je asynchronní zpracování RPC. V případě, že dvě RPC pracují se stejným objektem a jsou zavolány v přibližně stejnou chvíli, první z nich si objekt uzamkne a druhé na tomto uzamykání selže. Z tohoto důvodu je potřeba opakovat pokus o uzamčení po krátkém časovém intervalu. Tento interval musí být dostatečně dlouhý na to, aby první z funkcí úspěšně skončila, ale ne příliš dlouhý, aby uživatel nemusel zbytečně čekat.

Jako řešení tohoto problému jsem navrhl dekorátor, který pokus o uzamčení opakuje. Počet pokusů jsem volil 13 a po každém neúspěšném se funkce následující pokus odloží o 0.15 s. Tyto parametry jsem odhadl na základě pozorování odezvy na RPC, ale určitě se nemusí jednat o finální nastavení. Konstanty jsou umístěny na začátku třídy zámku, takže

se dají lehce upravit. Aktuálně bude uživatel čekat v nejhorším případě dvě vteřiny. Finální kód kontextového manažeru s opakováním zamykání v případě neúspěchu je následující:

```
@asynccontextmanager
async def ctx_write_lock(
    obj_ids: Union[str, Iterable[str]],
    owner: str,
    auto_unlock: bool = True,
    dry_run: bool = False
) -> AsyncGenerator[None, None]:
    @retry(exc=CannotLock,
           tries=glob.LOCK.LOCK_RETRIES,
           delay=glob.LOCK.RETRY_WAIT)
    async def lock():
        if not await glob.LOCK.write_lock(obj_ids, owner):
            raise CannotLock(glob.LOCK.ErrMessages.LOCK_FAIL.value)

    await lock()
    try:
        yield
    except Arcor2Exception:
        if not dry_run and not auto_unlock:
            await glob.LOCK.write_unlock(obj_ids, owner)
        raise
    finally:
        if dry_run or auto_unlock:
            await glob.LOCK.write_unlock(obj_ids, owner)
```

Základními parametry dekorátoru jsou seznam objektů pro uzamčení a uživatel, jež se je pokouší uzamknout. Následuje parametrizování dekorátoru a odekorování funkce, která se pokouší o uzamčení objektů. V tomto případě se musí jednat o funkci, protože samotný kontextový manažer nepodporuje dekorátory opakující pokusy. Po získání zámků, kontextový manažer předává řízení funkci, ze které je volán. Zbylé dva parametry této funkce slouží pro definování, co se má stát se zámky, jakmile je dokončen kód funkce, která používá tento kontextový manažer. V případě nastavení `auto_unlock` na hodnotu `False` nechává kontextový manažer zámky uzamčené. Parametr `dry_run` jsem přidal pouze pro správné chování vybraných RPC, jež využívají deaktivovaný argument `auto_unlock`. Tento parametr je hojně využíván i u většiny RPC a značí, že se provádí pouze testovací spuštění RPC. V případě tohoto testu nesmí objekty zůstávat uzamčeny, neboť by se staly nedostupnými. Tyto dva dodatečné argumenty jsem primárně použil v případech, kdy server po úspěšném dokončení RPC automaticky objekty odemyká. Jedná se například o přejmenování objektu. Funkcionalita ušetří jen zlomek času, avšak editor spouští paralelně mnoho RPC, tudíž každá taková optimalizace zlepšuje odezvu editoru.

Speciální případ použití

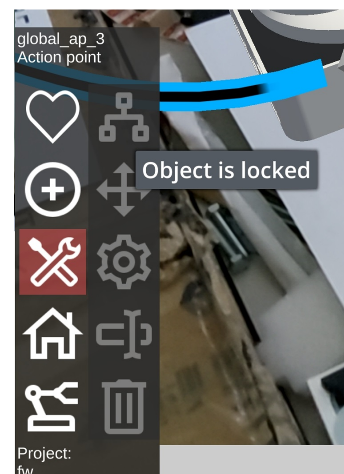
Parametr `auto_unlock` jsem dále použil v kódech RPC, které spouštějí důležité funkce pomocí `ensure_future` či `create_task`, jež jsou funkcemi knihovny `asyncio`. Obě tyto

funkce fungují velmi podobně, naplánují asynchronní spuštění jiné funkce a již nečekají na její výsledek. Jedná se například o kalibraci robota, jež vyžaduje jiný objekt. Při spuštění této kalibrace je robot uzamčen, avšak druhý potřebný objekt nikoliv. Ten se zamyká až na serveru a následně se odemyká po dokončení naplánované funkce. Pro tento účel bylo dále nutné modifikovat asynchronní, naplánovanou funkci tak, aby celý její kód byl obalen pomocí `try` a `finaly`. Blok `finally` obsahuje odemčení objektu, který zamyká server, zatímco kód této funkce je vykonáván v bloku `try`. Tato modifikace je důležitá v případě, že kód funkce selže a objekt uzamčený serverem by se mohl dále stát nedostupným.

Dalším místem, kde bylo vhodné použít tuto funkcionalitu, je hlavní menu. Zde je potřeba zamykat existující projekty a scény. Kontextový manažer, který automaticky neodemyká objekty jsem použil například pro mazání scény či projektu. V tomto případě vyvstává otázka, proč vlastně objekt není uzamčen editorem před započetím této editace. Je to z toho důvodu, že projekty a scény nejsou uzamčeny pro čtení ani pro zápis při otevření jejich menu a zamykat je explicitně pro zápis například před smazáním by bylo zbytečné RPC, protože by po něm automaticky následoval požadavek pro smazání. Zamykání při otevření menu není aplikováno z důvodu velkého omezení hlavního menu při používání více uživateli. V případě, že si dva uživatelé otevřou menu stejného projektu, není možné s projektem dále manipulovat, protože ani jeden z nich nezíská zámek pro zápis. Z tohoto důvodu jsem využil až zamykání na straně serveru, tudíž práce v hlavním menu zůstává také zabezpečena zámky a není nijak omezující. V případě zmíněného mazání projektu, je použit kontextový manažer s vypnutým `auto_unlock` z toho důvodu, že po smazání již není identifikátor projektu znám a tudíž jej nelze z databáze odstranit standardním způsobem. V tomto případě jsem aplikoval odemykání objektu těsně před jeho samotným smazáním, tudíž veškeré kontroly jsou validní, objekt má uzamčen výlučně jediný uživatel a v databázi zámků nezůstane žádný neexistující identifikátor.

5.1.2 Vylepšení implementace

V rámci implementace a testování jsem navrhl také několik vylepšení jak pro editor, tak pro server. Co se týká editoru, tak jsem navrhl omezení počtu odeslaných RPC, protože některé odpovědi byly dopředu známé. Toto se týká například levého menu v aktivním projektu či scéně (viz 5.2), kde se editor s každou změnou dotazoval, zda je akce dostupná. Tyto dotazy byly redukovány do stavu, kdy některá tlačítka menu jsou s notifikací o zamčení objektu deaktivovány a jejich obnovení se provádí až s notifikací o odemčení libovolného objektu.



Obrázek 5.2: Menu akcí s hláškou signalizující důvod nedostupnosti tlačítka

Další změnou, která byla aplikována pro editor je animace načítání v případě kalibrace. Zde jsem navrhl zobrazit tuto animaci, protože kalibrace trvá krátce a značně to zjednoduší zpracování RPC na serveru. Kalibraci lze spustit z menu objektu, tudíž objekt je při spuštění uzamčen. V případě, že by tato animace nebyla zobrazena, server by musel zamykat speciální konstantu, aby kód kalibrování nebyl závislý na zamykání objektu, které je řízeno z editoru, protože při kalibrování musí být objekt uzamčen.

Poslední vylepšení se týká získání výlučného přístupu k databázi zámků na delší dobu, než v rámci jedné metody. Tuto funkcionalitu jsem implemen-

toval opět jako kontextový manažer a i zde jsem použil dekorátor pro opakování uzamčení. Problém nastal ve chvíli, kdy si některý z uživatelů zamkl objekt z editoru. V tomto případě nelze získat konkrétní typ výlučného přístupu z důvodu aktivního zámku pro zápis a operace se tedy opakuje. Jedná se ovšem o zámek z editoru, tedy uživatel provádí časově náročnější akci (např. mění pozici objektu). Odpověď na dané RPC tedy trvá nejčastěji přibližně dvě vteřiny, což je maximální délka daná opakováním pokusů zamčení. Tento problém jsem řešil pomocí databáze zámků objektů v UI. Jedná se o jinou, jednodušší databázi, která obsahuje pouze objekty, jejichž uzamčení je signalizováno v editoru. Hlavní smysl této databáze a její strukturu popíši v 5.2.1. Zmíněný problém jsem tedy řešil tak, že v případě uzamčení některého z objektů z editoru, se daný objekt nachází ve zmíněné databázi a tudíž funkce neopakuje pokus o uzamčení, ale okamžitě vrací výjimku.

5.2 Registrace uživatelů

Každý editor, který se připojí k serveru, lze jednoznačně identifikovat identifikátorem instance websocketu[2], skrze který se připojil. Tento identifikátor je však prosté číslo, které uživatelům moc neřekne. Z tohoto důvodu jsem navrhl přidání textového vstupu k přihlašovacímu formuláři 3.2, kde si uživatel následně vyplní své jméno. Za účelem registrace jsem vytvořil RPC, skrze které se UI registruje. Dále jsem vytvořil objekt pro správu přihlášených uživatelů `Users`. Instance tohoto objektu je aktivní opět po celou dobu běhu serveru a přístup k ní je umožněn přes globální proměnnou. Je známým faktem, že globální proměnné nejsou nejlepší řešení, avšak v tomto případě žádná nepřibyla. Správa uživatelů souvisí totiž se všemi aktivními websockety, které byly do té doby uložené v jiné globální proměnné. Nové řešení umožňuje správu websocketů pomocí metod. Celé přihlášení bylo tedy nově rozšířeno o registraci uživatelského jména. Z pohledu funkčnosti editor nejprve vytváří spojení a skrze tohle spojení nově zasílá RPC pro registraci. V případě registrace již obsazeného uživatelského jména celé připojení končí chybovou hláškou.

5.2.1 Notifikace uživatelů

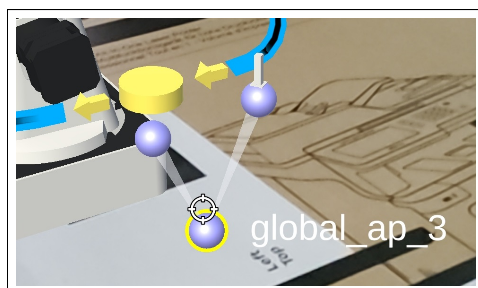
Po registraci se jméno propojí s instancí websocketu a v komunikaci dále nefiguruje. Toto jméno slouží primárně pro notifikace editorů, který z uživatelů vlastní zámek konkrétního objektu. Pro účely těchto notifikací jsem vytvořil dva objekty. Pro uzamčení a pro odemčení. Oba tyto objekty mají totožnou strukturu, která obsahuje seznam identifikátorů a jméno vlastníka. Pro rozeznání uzamčených objektů jsem navrhl změnu barvy, čímž je na první pohled zřejmé, že objekt či celá struktura objektů jsou uzamčeny. Pro tento případ se následně užilo zašednutí (viz obrázky 5.3a a 5.3b), což v editoru značí obecně nedostupný objekt. Pokud si uživatel následně objekt zaměří, zobrazí se nad ním jméno vlastníka zámku.

Editor nepotřebuje nijak složitě procházet stromovou strukturu při uzamčení celého stromu. Pro tyto účely jsem implementoval metody jak pro zjištění všech rodičů, tak pro zjištění všech potomků. V notifikaci se tedy nachází všechny objekty jednoho stromu.

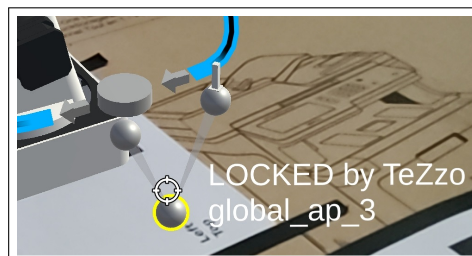
Správa uzamčených objektů

Zmíněné zašednutí uzamčených objektů se neděje při každém uzamčení. Tato funkcionalita se vykonává pouze pro déle trvající zámky, což jsou v tomto případě požadavky na uzamčení

[2] Websockets je knihovna, kterou server používá pro komunikaci s UI a naopak. Viz https://pypi.org/project/websockets/.



(a) Odemčená stromová struktura



(b) Uzamčená stromová struktura se zobrazením vlastníka zámku

Obrázek 5.3: Příklad objektů a jejich hierarchie v projektu

odeslané z editoru. V případě notifikování o každé změně spojené se zamykáním by nastával nepříjemný jev, že by objekty blikaly a v návaznosti na tyto notifikace by editor posílal nespočet zbytečných testovacích RPC. Tento jev je úzce spojený s RPC, ve kterých server užívá kontextového manažeru pro uzamykání.

V případě připojení nového editoru, je zapotřebí jej informovat o objektech, jež mají být uzamčené, respektive jsou zašedlé. Pro tyto účely jsem vytvořil druhou databázi, pro správu objektů uzamčených v UI. Struktura této databáze je již pouze slovník, jehož klíčem je jméno uživatele a hodnotou je seznam uzamčených objektů. Při připojení editoru pouze transformuji obsah této databáze do notifikací, kde pro každého uživatele vzniká samostatná notifikace. Tyto zprávy následně zasílám pouze nově připojenému editoru.

Fronta notifikací

S notifikacemi o zamykání je potřeba řešit další problém, v podobě pořadí těchto zpráv. Většinu z těchto informací server zpracovává tak, že naplánuje spuštění funkce, která tuto notifikaci odešle. V případě plánování není zajištěno pořadí, tudíž může nastat situace, kdy se připojí uživatel, vytvoří se objekty notifikací s tím, že se odešlou později a v tu chvíli jiný uživatel uvolní zámek, na který již je vytvořena notifikace. Toto uvolnění vytváří automaticky další notifikaci. V jakém pořadí editor přijme vytvořené zprávy? Pro vyřešení tohoto problému jsem vytvořil FIFO frontu, do které se vkládají notifikace. Fronta je zpracovávána asynchronně, avšak pořadí zpráv je již zajištěno. Kód zpracování této fronty je následující:

```
async def run_lock_notification_worker() -> None:
    while True:
        notif_data = await glob.LOCK.notifications_q.get()
        obj_ids = notif_data.obj_ids
        data = sevts.lk.LockData(obj_ids, notif_data.owner)
        evt = sevts.lk.ObjectsLocked(data) if notif_data.lock \
            else sevts.lk.ObjectsUnlocked(data)

        if notif_data.client:
            await notif.event(notif_data.client, evt)
        else:
            await notif.broadcast_event(evt)
```

Tato funkce je spuštěna se startem serveru. Jedná se o nekonečenou smyčku, avšak díky čekání na notifikace pomocí `await` nevytěžuje procesor prázdnými operacemi. Celý kód spočívá v tom, že je vyňata notifikace z fronty, následně se vytvoří objekt notifikace ve formátu, který rozpozná editor a notifikace se odešle. Typ a parametrizace objektu notifikace vychází z parametrů prvku vyňatého z fronty. V případě notifikace jsem vytvořil následující strukturu:

```
class LockEventData:
    __slots__ = "obj_ids", "owner", "lock", "client"

    def __init__(
        self, obj_ids, owner, lock=False, client=None
    ):
        self.obj_ids = list(obj_ids)
        self.owner = owner
        self.lock = lock
        self.client = client
```

Struktura je použita primárně z důvodu typové kontroly, jež nelze jednoduše provádět například u slovníku. Obsahuje dva povinné parametry v podobě seznamu objektů a vlastníka daného zámku. Parametr `lock` signalizuje, zda má být vytvořena notifikace pro uzamčení či odemčení a v případě parametru `client` se jedná o adresáta. V případě nevyplnění je notifikace odeslána všem. Tohoto parametru lze využít například při přihlašování nového uživatele a následném oznamování, které objekty jsou uzamčeny.

Životnost zámku

V případě, že editor graficky znázorňuje uzamčený objekt, nabízí se otázka, co se stane s tímto zámkem v případě, že se jeho majitel odpojí. V implementaci zámku jsem řešil i tento případ. Pokud se uživatel odpojí (úmyslně, uspí zařízení, či editor selže), daný websocket se odregistrovává od serveru. Tuto funkcionalitu jsem rozšířil jednak o smazání uživatele z databáze aktivních uživatelů a dále jsem spustil funkci, která se asynchronně uspí po dobu časového intervalu a následně uvolní všechny zámky odpojeného uživatele. Při uvolnění smažu zámky z databáze zámků, ale také z databáze zámků v UI, o jejichž uvolnění následně zpravuji všechny editory.

V případě, že se uživatel opět připojí, je funkce pro uvolnění zrušena a uživateli přicházejí notifikace o všech jeho zámcích. Jednalo-li se o odpojení v podobě uspání zařízení, editor toto detekuje a uživateli zámky zůstávají. V případě násilného ukončení editor zasílá automaticky RPC pro uvolnění všech svých zámků. Toto uvolnění je nezbytné, jelikož editor postrádá informace o tom, jakou činnost uživatel vykonával před násilným ukončením.

5.3 Možná vylepšení

Každá práce lze nějakým způsobem vylepšit a toto pravidlo se týká i této práce. V průběhu implementace a testování jsem odhalil tři problémy, které by bylo vhodné upravit. Tyto nedostatky nejsou výrazně limitující a nejedná se o hrubý nedostatek, který by byl až nepříjemný. Jedná se pouze o vylepšení, které dokáže zpříjemnit práci v AREditoru.

Zamykání

V případě zamykání se dá vylepšit zamykání stromu. Aktuální implementace zamyká vždy celý strom. Toto vylepšení by umožňovalo uzamknout jen podstrom, jehož kořenem bude uzamykaný objekt. Řešení vyžaduje dodatečné kontroly v případě uzamykání objektů ve stejném stromě a modifikaci struktury uložení jednotlivých zámků. V případě kontroly již zámek disponuje potřebnými metodami, které stačí jen vhodně použít. V případě struktury jsou vyžadovány modifikace, které ovlivní nemalé množství metod zámku, což zvyšuje náročnost případné implementace. Tato funkcionalita umožňuje práci více uživatelů na jednom stromu objektů, tudíž může nepatrně urychlit vývoj projektu. Na druhou stranu složitější kontroly mohou zvýšit prodlevu při čekání na odpověď RPC. Případné posouzení a porovnání rychlosti již nechávám na případném autorovi tohoto vylepšení.

Rychlejší zpracování RPC

Toto vylepšení vyžaduje velmi málo programátorského času. Jedná se o implementovaný dekorátor opakující zamykání. V tomto případě se nepodařilo nasbírat potřebné množství dat pro optimalizaci doby čekání. Náročnost tohoto vylepšení spočívá v tom, že autor musí shromáždit data z používání serveru v rámci alespoň několika hodin. Server musí dále používat více uživatelů zároveň, protože v případě jednoho uživatele je délka čekání nejčastěji nulová. Za účelem sběru těchto dat jsem přidal výpis, který zaznamenává celkovou dobu čekání do logu. Tyto data je nutné extrahovat, zpracovat a na základě získaných dat upravit dobu čekání.

Granularita zámků

Rozdělení zamykání objektů na jednotlivé části, které mohou být uzamčeny může opět zrychlit práci na projektu. V tomto případě se může jednat o umožnění uzamčení pro pohyb, pro menu či dokonce pro jednotlivé položky tohoto menu (podobně pro hlavní menu). Díky této změně by bylo možné nepoužívat notifikace o zámcích a případně jen vypínat jednotlivé nedostupné položky se zobrazením patřičné hlášky. Toto vylepšení je velmi náročné na implementaci jak na straně editoru, tak na straně serveru. Dle mého názoru dává smysl pro projekty, na kterých budou pracovat desítky uživatelů najednou. V opačném případě se jedná o drahé vylepšení.

5.4 Dokumentace řešení

V případě dokumentování kódu zastávám názor, že vytváření dlouhých wiki stránek je nepraktické a u projektu v aktivním vývoji téměř nepoužitelné. Tyto stránky většinou obsahují větší množství informací a s ohledem na toto množství nezbývá čas na jejich údržbu, tudíž rychle stárnou a obsahují zastaralé informace. Z tohoto důvodu jsem zvolil dokumentování kódu pomocí docstring[3], což je rozšířený formát, ze kterého lze automatizovaně generovat manuálové stránky či jiný formát dokumentace. V tomto formátu jsem okomentoval všechny funkce balíčku zámku. Vytvořený komentář může vypadat například následovně:

```
async def write_lock(
    self, obj_ids: Union[Iterable[str], str], owner: str,
    lock_tree: bool = False, notify: bool = False
) -> bool:
    """Lock object or list of objects for writing, possible to lock
    whole tree where object(s) is located.

    :param obj_ids: object or objects to be locked
    :param owner: name of lock owner
    :param lock_tree: boolean whether lock whole tree
    :param notify: if set, broadcast information about locked
    objects and store those objects in user lock database
    :return: boolean whether lock was successful
    """
```

Součástí komentáře je stručný a výstižný popis funkce, jejich parametrů a návratové hodnoty. Formát umožňuje popis datových typů, ovšem toto již není zapotřebí, neboť typy argumentů se nacházejí již v deklaraci funkce. Tuto typovou kontrolu lze také považovat za část dokumentace, protože usnadňuje použití dané funkce v IDE tím, že zobrazuje relevantní nápovědu. Výpis nejčastěji obsahuje kombinaci dokumentačních způsobů, nejčastěji datový typ a textový popis převzatý z docstring popisu.

Z důvodu, že zámek je Pythonovský balíček a některé použití nemusí být z dokumentace zřejmé, vytvořil jsem také soubor `README.md`, jež je součástí balíčku. Součástí tohoto popisu je stručný návod, jak používat zámek a jeho struktury. Dále jsem doplnil případy aplikace zamykání v situacích, které nejsou na první pohled zřejmé (například plánování funkce, při jejímž spuštění musí být objekt stále uzamčen).

V kódu jsem dále vytvořil několik `TODO` komentářů, které odkazují na možné vylepšení nebo na funkcionalitu celého serveru která je plánovaná, ale zatím neimplementovaná.

5.5 Zhodnocení

V případě implementace výlučného přístupu lze očekávat zpomalení aplikace. Z tohoto důvodu jsem se rozhodl porovnat výslednou rychlost před a po implementaci zamykání. Za tímto účelem jsem vytvořil skript, který simuluje připojení tří zařízení na server a spouští jediné RPC, které aktualizuje pozici akčního bodu, pořád dokola. Tento testovací příklad jsem spouštěl v devíti verzích. Nejprve případ, kdy se nezamyká vůbec, což simuluje nevalidní,

[3]Python Docstring konvence – `https://www.python.org/dev/peps/pep-0257/`.

ale původní řešení. Následně jsem testoval rychlost stejného RPC s aplikací kontextového zámku a nakonec s uzamykáním a odemykáním z editoru. Toto testování jsem prováděl ve třech případech užití. V prvním případě jsem odesílal RPC cyklicky za sebou a v ostatních jsem mezi jednotlivými kroky cyklu aplikoval čekání 0.01 s a 0.05 s. V případě této pauzy se jedná o náhodnou hodnotu v intervalu $< 0; X >$, kde X reprezentuje délku pauzy. Pro simulaci chování více uživatelů jsem vygeneroval pole těchto náhodných hodnot o délce 5000 záznamů, které jsem promíchal při přihlášení každého uživatele, pro simulaci náhodnosti. Díky této vlastnosti čekali všichni simulovaní uživatelé stejnou dobu. Výsledky délky běhu pro všechny varianty jsou zobrazeny v tabulce 5.1.

Z běhu bez čekání mezi RPC lze vidět, že zpracování požadavku doplněného o uzamčení pomocí kontextového manažera je prodlouženo o zanedbatelnou hodnotu. Toto však nelze konstatovat v případě uzamykání pomocí RPC, kde délka běhu je přibližně dvojnásobná. Tyto výsledky jsou dále potvrzeny i simulací s čekáním mezi odesíláním jednotlivých testovacích RPC. Jedná se ovšem o zpomalení v řádech jednotek tisícin vteřiny v nejhorších případech, tudíž tato doba je stále zanedbatelná. Reálná odezva serveru se bude pohybovat mezi těmito hodnotami, protože server využívá jednak oba testované způsoby zamykání, ale také jejich kombinaci či částečnou aplikaci.

	Čekání mezi RPC		
	0.0 s	0.01 s	0.05 s
Celkové čekání	0.0 s	25.024 s	126.406 s
Před zamykáním	7.543 s	35.583 s	141.756 s
Po zamykání Kontextovým zámkem	8.448 s	36.427 s	140.070 s
Rozdíl na RPC	0.000181 s	0.0001688 s	0.0003372 s
Po zamykání pomocí RPC	16.456 s	43.912 s	154.874 s
Rozdíl na RPC	0.0017826 s	0.0016658 s	0.0026236 s

Tabulka 5.1: Porovnání délky požadavku aktualizace polohy akčního bodu pro různé doby čekání před a po implementaci zamykání kontextovým zámkem a zamykání z editoru.

Kapitola 6

Testování

Pro zajištění správnosti implementace jsem implementoval sadu testů pokrytí kódu, regresních a integračních testů. Všechny tyto testy kontrolují validitu celého balíčku zámku tedy zamykání jednotlivých objektů, uvolnění zámků po odhlášení uživatele nebo také odeslání notifikací o zámcích. Funkční testy jsou také podmínkou pro zařazení změn do gitu. V opačném případě nelze provést `merge`.

Regresní testy a testy pokrytí

Pro verifikaci, zda nově přidaná funkcionalita nerozbije funkční kód [8], jsem vytvořil regresní testy. Hlavním cílem těchto testů je prověřit, zda metody zámku dodržují očekávané chování. Příkladem testovacích případů může být:

- Kontrola, zda zámek ukládá data do databáze korektně.
- Možnost uzamčení více objektů v jednom stromu.
- Nemožnost uzamčení části stromu v případě, že je uzamčen celý strom.
- Kontrola odstranění všech pozůstatků zámku při odemčení.
- Rozsáhlejší testování funkce `get_root_id`, která je stěžejní pro celý zámek. V hlavním menu musí najít existující scény a projekty, při spuštěné scéně všechny objekty dané scény a analogicky pro spuštěný projekt.

Jedná se o pytesty[1], kde každý z testů je zaměřen na jinou část zámku. Tyto testy nevyžadují kompletní spuštění serveru, nebo více jeho částí (viz 3.1.2), ale inicializují instanci zámku, kterou dále testují.

Samostatné regresní testy dosahovaly již značného pokrytí kódu, tudíž jsem je rozšířil o některé další případy. Pokrytí kódu jednotlivých modulů je zobrazeno na obrázku 6.1. Výsledné pokrytí kódu je 96 %. Zbylé nepokryté řádky jsou výjimky v krajních případech, které nepovažuji za kritickou část zámku a tudíž jsem je netestoval. Modul notifikací má nejnižší pokrytí z důvodu, že se zde nachází funkce pro odesílání notifikací a tuto funkcionalitu není možné ověřit tímto typem testu. Validitu tohoto kódu testuji v integračním testu.

Při dotváření testů na pokrytí kódu jsem odhalil několik částí mrtvého kódu a také několik chyb. Oba tyto nedostatky jsem odstranil. Z tohoto poznatku lze konstatovat, že pravidlo „testování má smysl“ obecně platí.

[1] pytest – https://docs.pytest.org/en/6.2.x/.

Coverage report: 96%

Module ↑	statements	missing	excluded	coverage
src/python/arcor2_arserver/lock/__init__.py	3	0	0	100%
src/python/arcor2_arserver/lock/exceptions.py	7	0	0	100%
src/python/arcor2_arserver/lock/lock.py	354	9	0	97%
src/python/arcor2_arserver/lock/notifications.py	11	7	0	36%
src/python/arcor2_arserver/lock/structures.py	84	1	0	99%
Total	**459**	**17**	**0**	**96%**

Obrázek 6.1: Celkové pokrytí kódu zámku testy

Integrační testy

Integrační testy pro jednotlivé části serveru již v nějaké podobě existovaly před implementací zámku. Jedná se primárně o testy, které spustí server se všemi potřebnými částmi. Díky těmto testům lze reálně zasílat RPC na server a kontrolovat všechny očekávané odpovědi jak na RPC, tak v podobě notifikací.

Pro účely testování zámku jsem vytvořil fixturu[2] pro projekt, která inicializuje potřebnou scénu, vytvoří projekt a do projektu přidá základní objekty. Dále jsem vytvořil test, ve kterém jsem se snažil maximálně otestovat notifikace spojené se zamykáním. Pro tento testovací případ jsem připojil hned dva websockety a kontroloval, zda oba dostávají veškeré zprávy, které dostávat mají. Tento testovací případ zahrnoval i odhlášení a opětovnou registraci jednoho z těchto testovacích uživatelů.

Testování implementace

Implementace zamykání probíhala tak, že jsem nejprve vytvořil zámek s patřičnými metodami a postupně aplikoval zamykání do kódu jednotlivých RPC. Při této aplikaci jsem komunikoval se členy ARCOR2 týmu a diskutoval, jak zamykat RPC, u nichž nebylo jasné chování. Po této vlastní implementaci byla vydána nová verze editoru, která byla z velké části přepracovaná, což rozbilo část mé implementace. Toto považuji za komunikační nedostatek. Problém se podařilo rychle odstranit díky tomu, že byla zahájena aplikace zámků v editoru ze strany ARCOR2 týmu. Při této implementaci jsme kooperativně hledali nedostatky jak v mé implementaci, tak v editoru a opravovali je.

Jakmile byly obě implementace v použitelném stavu s pouze malým počtem nedostatků, proběhlo rozsáhlejší testování, kdy na projektu pracovalo pět uživatelů současně. Cílem této akce bylo primárně vyzkoušení editoru se zámky. Část scénáře vypadala následovně:

- **Vytváření a pozicování objektů scény** – zde si každý vytvořil svůj objekt, pojmenoval jej a umístil kdekoliv do scény. V tomto kroku byla objevena limitace v podobě vytváření objektů, kde tato akce vyžadovala kompletní přístup k zámku.
- **Vytváření a pozicování objektů projektu** – zde si každý vytvořil nějaké akční body a dále programoval logiku.
- **Přístup k uzamčeným objektům** – všichni jsme se přesunuli k jedné stromové struktuře a po komunikaci jsme zamykali různé části stromu a zkoušeli editovat uzamčené objekty.

[2] https://docs.pytest.org/en/6.2.x/fixture.html

- **Testování notifikací zámku** – uživatelé mezi sebou komunikovali a sledovali, zda objekty správně zašedávají, při uzamčení a naopak, při odemčení.

Při tomto testování bylo odhaleno několik málo chyb, kde většina souvisela s editorem. Chyby týkající se této práce spolu s odhaleným nedostatkem při vytváření objektů jsem odstranil a aktuálně nejsou známy žádné další chyby.

Kapitola 7

Závěr

Cílem této práce bylo analyzovat systém pro programování v rozšířené realitě ARCOR2 a na základě této analýzy navrhnout způsob řešení výlučného přístupu. Dále bylo potřeba tento návrh implementovat, vytvořit sadu testů, které zajistí funkčnost řešení a vytvořit dokumentaci k této implementaci. Tyto cíle byly úspěšně splněny.

Při analýze systému ARCOR2 jsem se seznámil primárně s knihovnou AsyncIO, ale také s jinými knihovnami, které implementují nějaký způsob výlučného přístupu. Toto seznámení bylo pozitivní, protože do této doby jsem o takovém elegantním řešení konkurenčního programování nevěděl.

Po prozkoumání zdrojů, na kterých je server vystavěn, jsem definoval místa, která jsou náchylná na chybu v asynchronním prostředí. Na základě získaných znalostí jsem navrhl řešení, které nalezené nedostatky pokrývá a zabezpečuje tím používání celého systému v kolaborativním prostředí s více uživateli.

Dle návrhu jsem implementoval výlučný přístup ke zdrojům, u nichž hrozil stav inkonzistence při použití v asynchronním prostředí. Při implementaci jsem kladl důraz na rychlost, protože z analýzy vyplynulo, že aplikace výlučného přístupu bude velmi častá. Tohoto cíle bylo dosaženo uspořádáním kódu podle nejpoužívanějších možností, v případě hledání hodnoty, snaha o používání synchronního zpracování, či upřednostnění komunikace v rámci serveru před ostatními službami. V průběhu realizace návrhu jsem dále odhalil drobné nedostatky, se kterými návrh nepočítal. Tyto nedostatky byly odstraněny modifikací návrhu. Výsledné řešení umožňuje velmi jednoduchou aplikaci výlučného přístupu a dle výsledků testování lze konstatovat, že funguje.

Validace řešení je zajištěna v podobně implementované sady testů, které kontrolují základní i složitější případy použití částí implementace, ale také použití implementace jako celku.

Při analýze výsledného řešení bylo zjištěno, že aplikace výlučného přístupu zpomalí rychlost komunikace se serverem přibližně na polovinu původní rychlosti. Jedná se ovšem o nejhorší možný případ, protože testovaný scénář trojnásobně zvyšuje komunikaci se serverem, což je při reálném použití téměř vyloučeno.

V budoucnu by v této implementaci bylo vhodné řešit granularitu zámků a také omezení v podobně uzamykání celé stromové struktury. Tyto nedostatky mohou být limitující v případech, kdy se na řešení podílí více uživatelů, protože nedovolují práci s jednotlivými prvky objektů či práci více uživatelů v jedné stromové struktuře.

Literatura

[1] *Handbook of Augmented Reality.* 1. vyd. New York, NY: Springer New York, 2011. ISBN 9781461400639.

[2] Agile Aliance. *What are Mock Objects?* [online]. 2015 [cit. 2020-12-28]. Dostupné z: https://www.agilealliance.org/glossary/mocks/.

[3] Biswas, S. *Dining Philosopher Problem Using Semaphores* [online]. 2019 [cit. 2020-12-28]. Dostupné z: https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/.

[4] Christensson, P. *RPC Definition.* 2006 [cit. 2020-12-27]. Dostupné z: https://techterms.com/definition/rpc.

[5] Deng, L., Yang, Z., Du, P. a Song, Y. A cloud platform for space science mission concurrent design. *Concurrent Engineering.* Srpen 2017, sv. 26, s. 1063293X1772484. DOI: 10.1177/1063293X17724848.

[6] Guerrero, L. A. a Fuller, D. A. A pattern system for the development of collaborative applications. *Information and Software Technology.* 2001, sv. 43, č. 7, s. 457 – 467. DOI: https://doi.org/10.1016/S0950-5849(01)00154-9. ISSN 0950-5849. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0950584901001549.

[7] Hou, J., Su, C., Tang, L., LidaZhu a Wang, W. Conflict resolution for collaborative design. In: *2008 IEEE International Conference on Automation and Logistics.* 2008, s. 875–880. DOI: 10.1109/ICAL.2008.4636273.

[8] Hunt, J. *Advanced Guide to Python 3 Programming.* Cham: Springer International Publishing, 2019. Undergraduate Topics in Computer Science. ISBN 978-3-030-25942-6.

[9] Khalid, M. Y. U. *Context Managers* [online]. 2017 [cit. 2020-12-31]. Dostupné z: https://book.pythontips.com/en/latest/context_managers.html.

[10] Li, R., Yu, G., Lu, Z. a Song, W. P2P-based Locking in Real-Time Collaborative Editing Systems. In: *2007 11th International Conference on Computer Supported Cooperative Work in Design.* 2007, s. 24–29. DOI: 10.1109/CSCWD.2007.4281404.

[11] Malý, M. *REST: architektura pro webové API* [online]. 2009 [cit. 2020-12-28]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api//.

[12] MDN contributors. *The WebSocket API (WebSockets)* [online]. 2020 [cit. 2021-01-10]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[13] Miller, D. and Whitlock, J. and Gardiner, M. and Ralphson, M. and Ratovsky, R. and Sarid, U. and Harmon, J. and Tam, T.. *OpenAPI Specification* [online]. 2020 [cit. 2021-01-10]. Dostupné z: http://spec.openapis.org/oas/v3.0.3.

[14] Oluwatosin, H. S. Client-Server Model. *IOSR Journal of Computer Engineering*. 2014, sv. 16, s. 67 – 71. ISSN 2278-8727. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1083.8741&rep=rep1&type=pdf.

[15] Python Software Foundation. *AsyncIO — Asynchronous I/O* [online]. 2020 [cit. 2020-12-29]. Dostupné z: https://docs.python.org/3/library/asyncio.html.

[16] Summerfield, M. *Programming in Python 3 : a complete introduction to the Python language*. Upper Saddle River, New Jersey: Addison-Wesley, 2010. ISBN 978-0-321-68056-3.

[17] Tan, A., Pham, B., Zhang, J. a Brown, R. A Collaborative Framework for Simultaneous and Seamless 3D Graphics Manipulation. In: *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*. New York, NY, USA: Association for Computing Machinery, 2008, s. 206–210. MoMM '08. DOI: 10.1145/1497185.1497229. ISBN 9781605582696. Dostupné z: https://doi.org/10.1145/1497185.1497229.

[18] Tas, S. *Faster Lookups In Python* [online]. 2020 [cit. 2021-05-12]. Dostupné z: https://towardsdatascience.com/faster-lookups-in-python-1d7503e9cd38.

[19] Vysoké Učení Technické v Brně. *Test-it-off: robotizované offline testování produktů* [online]. 2020 [cit. 2020-12-23]. Dostupné z: https://www.fit.vut.cz/research/project/1308/.cs.

Příloha A

Obsah CD

Vzhledem k tomu, že se jedná o projekt[1] ve stádiu vývoje a tato práce se týká více částí, nelze některé moduly považovat za mou tvorbu. Z tohoto důvodu přikládám spolu se soubory zámku také soubor `git.diff`, který obsahuje všechny mnou provedené změny.

Adresářová struktura CD

```
/
├── lock/
│   ├── README.md
│   ├── __init__.py
│   ├── exceptions.py
│   ├── lock.py
│   ├── notifications.py
│   └── structures.py
├── rpc/
│   ├── lock.py
│   └── user.py
├── rpc_data/
│   ├── lock.py
│   └── user.py
├── decorators.py
├── test_lock.py
└── git.diff
```

[1] https://github.com/robofit/arcor2