Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Ovládání robota NAO v přímém režimu

Controlling of the NAO robot in direct mode

Bakalářská práce

Vypracoval: Daniel Štrobl

Vedoucí práce: Mgr. Václav Šimandl, Ph.D.

České Budějovice 2021

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury. Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 22.04.2021

Daniel Štrobl

Abstrakt/Anotace

Práce se zabývá programováním a prací s robotem NAO od společnosti Softbankrobotics. Popisuje robotův hardware i software. Cílem této práce bylo vytvoření aplikace, která přímo ovládá robota. Aplikace byla vytvořena v jazyku Python 2.7 pro operační systém Windows. Při tvorbě této aplikace bylo použito SDK dodané výrobcem. Aplikace využívá robotův framework naoqi a používá jeho API. Aplikace je schopna okamžitě posílat příslušné příkazy, nebo tyto příkazy skládat do sekvencí a vytvářet z nich programy. Tyto programy nejsou samostatně spustitelné, ale je potřeba je nahrát do aplikace a odtud je spouštět. Ve své práci popisuji tvorbu této aplikace, její vlastnosti a problémy na které jsem při vývoji této aplikace narazil. Vytvořenou aplikaci jsem porovnal s oficiálním programem Choreographe. Z testování vyplynulo, že mnou vytvořená aplikace je rychlejší jak z hlediska tvorby sekvence příkazů, tak z hlediska rychlosti posílání příkazů robotovi.

Klíčová slova

Robot, NAO, Přímý režim, Python 2.7, SDK, Aplikace pro Windows, Choreographe

Abstract

The thesis deals with programming and work with the NAO robot from the company Softbankrobotics. It describes the robot's hardware and software. The aim of this work was to create an application that directly controls the robot. The application was created in python 2.7 for the Windows operating system. The SDK supplied by the manufacturer was used to create this application. The application uses the robot's naoqi framework and uses its API. The application is able to send the appropriate commands immediately, or compose these commands into sequences and create programs from them. These programs are not self-executing, but it's needed to load them into the application and run them from there. In my work, I describe the creation of this application, its features, and problems that I encountered during the development of this application. I compared the created application with the official Choreographe program. The testing showed, that the application I created is faster both in terms of creating a sequence of commands and in terms of the speed of sending commands to the robot.

Keywords

Robot, NAO, Direct mode, Python 2.7, SDK, Application for Windows, Choreographe

Poděkování

Rád bych poděkoval Mgr. Václavu Šimandlovi, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování bakalářské práce.

Vymezení důležitých pojmů

API – (Application Programming Interface) označuje v informatice rozhraní pro programování aplikací. Jde o sbírku procedur, funkcí, tříd či protokolů nějaké knihovny, které může programátor využívat.

Framework – je softwarová struktura, která slouží jako podpora při programování, vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovny API, podporu pro návrhové vzory nebo doporučené postupy při vývoji.

GUI – Grafické uživatelské rozhraní, které umožňuje ovládat aplikaci pomocí interaktivních ovládacích prvků.

Hardware – Označení veškerého fyzicky existujícího vybavení zařízení

IDE – Vývojové prostředí, které usnadňuje programátorům práci

SDK – (Software Developer Kit) typická sada vývojových nástrojů umožňující vytváření aplikací pro určité softwarové balíčky, frameworky, počítačové systémy, herní konzole, operační systémy nebo podobné platformy

Software – (neboli programové vybavení) je v informatice souhrnný název pro všechny výpočetní programy používané v počítači, nebo stroji, které provádějí nějakou činnost

Widget – je element GUI, který zobrazuje informace, nebo poskytuje specifickou cestu, jak může uživatel interagovat s aplikací

Obsah

Abstrakt/Anotace 3
Klíčová slova 3
Abstract 4
Keywords 4
Poděkování 5
Vymezení důležitých pojmů 6
Obsah 7
1 Úvod 10
2 Cíle práce 10
2.1 Metoda práce 11
3 Teoretická část 12
3.1 Robot Nao 12
3.2 Hardware 12
3.2.1 Možnosti připojení 13
3.2.2 Zvuková a vizuální technika 13
3.2.3 Tlakové senzory 15
4 Software 16
4.1 Software uvnitř robota 16
4.1.1 Naoqi OS 16
4.1.2 NAOqi 16
4.1.3 NAOqi framework 17
4.2 Síťová komunikace 18
4.2.1 Broker 18
4.2.2 Proxy 19
4.2.3 Moduly 20
4.2.4 Blokované a neblokované volání funkcí 21

4.3 Software pro uživatele 22

4.3.1 Choreographe 22

4.3.2 Vysvětlení API a SDK 23

4.3.3 Dostupná API 24

5 Vývoj aplikace 25

5.1 Požadavky na aplikaci: 25

5.2 Tvorba aplikace pro Android 26

5.3 Python 26

5.4 Konfigurace Pythonu, SDK 27

5.5 Použité API a moduly z knihovny naoqi 28

5.5.1 Pohyb 28

5.5.2 Zvuk a hlas 29

5.5.3 Vytvoření proxy 30

5.6 Návrh a vývoj aplikace 30

5.6.1 Modul prikazy.py 30

5.6.2 Okamžité spuštění příkazu 34

5.6.3 Editor 36

5.6.4 Grafická podoba 39

6 Dílčí problémy řešené při vývoji aplikace 41

6.1 Významné programové konstrukce 41

6.1.1 Import knihoven 41

6.1.2 Kombinování funkcí 42

6.1.3 Získání a zpracování vstupů od uživatele 42

6.1.4 Reagování na stav aplikace 43

6.1.5 Vykreslování zadaných příkazů 44

6.2 Vytvoření souboru .exe 45

7 Testování aplikace 47

7.1 Porovnání s aplikací Choreographe ... 47
7.1.1 Testování časové náročnosti ... 47
7.1.2 Testování rychlosti odpovědi ... 48
7.1.3 Limity mé aplikace ... 49
7.2 Hodnocení aplikace ... 49
8 Závěr ... 50
9 Seznam obrázků ... 51
10 Seznam zdrojových kódů ... 52
11 Zdroje ... 53

1 Úvod

Pro vypracování této práce jsem se rozhodl z důvodů mého zájmu o robotiku a programování, konkrétně mě zaujal robot Nao. Dále jsem si chtěl prohloubit znalosti v oblastech programování, vývoj aplikací a síťové komunikace. Hlavní motivací pro vznik této bakalářské práce byla touha robota rychle a snadno ovládat mimo oficiální software, který je velice robustní a pomalý pro provádění jednoduchých úkonů. Během různých popularizačních akcích, na kterých jsem viděl interakci publika a robota Nao, jsem si všiml, že pokud robotovi skončí program, který má uložený v paměti, nebo se ho někdo na něco zeptá, tak prezentující nemá rychlý způsob, jak na tyto neočekávané situace reagovat. Výsledek mé práce by tedy měla být aplikace, která dokáže rychle a snadno interagovat s publikem.

2 Cíle práce

Bakalářská práce je rozdělena na dvě části, na teoretickou část a praktickou část.

Cílem teoretické části práce bylo popsat, jak fungují jednotlivé části robota, jaké jsou jejich vlastnosti a jak se dají individuálně programovat. Dále jsem zjistil, jak lze s robotem navázat spojení, jak následně probíhá jeho komunikace s uživatelem a jaké možnosti programování a ovládání robota již existují. Uvedl jsem, jaké má úskalí vývoj podobné aplikace pro různé operační systémy a za jakých podmínek je možné ji realizovat.

Cílem praktické části práce bylo vytvořit aplikaci, prostřednictvím které by bylo možno robota NAO ovládat přímými příkazy, tj. bez nutnosti vytvářet program uložený v paměti robota. Popsal jsem framework NAOqi a jeho významné API, které jsem používal. Dále jsem vytvořil návrh výsledné aplikace a popsal její důležité části. Tato aplikace měla být následně otestována před skupinou dětí, ale vzhledem k situaci, která nastala kvůli Covid-19 jsem od toho upustil.

Aplikace obsahuje příkazy: postav se, sedni si, lehni si na záda, lehni si na břicho, otoč se o x°, jdi dopředu o x, řekni větu, přečti text ze souboru.

Tyto příkazy se dají skládat do řady a vytvářet z nich jednotlivé programy. Tyto programy jsme schopni jednak ukládat do souborů, i načítat z nich.

2.1 Metoda práce

Ze začátku jsem se seznámil s robotovým manuálem a jeho dokumentací. Zjistil jsem jeho základní funkce a principy. Popsal jsem, jak funguje a jakými způsoby dokáže přijímat instrukce. Následně jsem vyzkoušel, jakým způsobem se dá s robotem komunikovat přes síť. Začal jsem u PC a komunikaci jsem prováděl přes oficiální software Choreographe. Zjistil jsem, že nejsem schopen vytvořit tuto aplikaci pro systém Android. Jelikož nejsem schopen vytvořit tuto aplikaci pro Android, vybral jsem po dohodě s vedoucím práce jazyk Python, pro který výrobce poskytuje SDK a vytvořil jsem aplikaci pro systém Windows.

Při navrhování aplikace jsem postupoval podle vodopádového modelu.

Požadavky – Návrh – Implementace – Verifikace

Nejdříve jsem si stanovil požadavky na aplikaci. Podle požadavků jsem vytvořil návrh dané aplikace, a nakonec jsem vyzkoušel funkčnost výsledné aplikace.

3 Teoretická část

3.1 Robot Nao

Robot Nao, je robot vyrobený francouzskou společností SoftBank Robotics v roce 2006. Jedná se o autonomního, programovatelného humanoidního robota. Robot je vybaven velkým množstvím senzorů a společně s ostatními funkcemi je primárně určený k tomu, aby dokázal mluvit, chodit, tančit a rozpoznávat obličeje, zkrátka aby dokázal imitovat činnosti člověka. Od svého vzniku byl tento robot neustále inovován a zdokonalován až do aktuální šesté generace. Tento robot, který je jeden obrovský programovatelný nástroj a má obrovský potenciál jak ve vzdělávání, tak i ve výzkumu [1].

Obrázek 1 Robot Nao [1]

3.2 Hardware

Robot je vysoký necelých 58 cm a je vyroben z několika typů plastů, konkrétně ABS-PC, PA-66 a XCF-30 a váží úctyhodných 5,4 Kg. Robot je ovládán operačním systémem NAOqi OS, který běží na čtyř jádrovém procesoru Atom E3845, tento procesor je doplněný operační pamětí typu DDR3 o velikosti 4 GB [2].

Pro zjišťování dat polohy robota vůči zemi například, aby nespadl nebo dokázal udržet rovnováhu má na starosti inerciální jednotka složená z tří osového gyroskopu s úhlovou rychlostí ~500°/s a tří osového akcelerometru se zrychlením až 2 g, kde g je tíhové zrychlení zhruba 9,8 $m \cdot s^{-2}$.

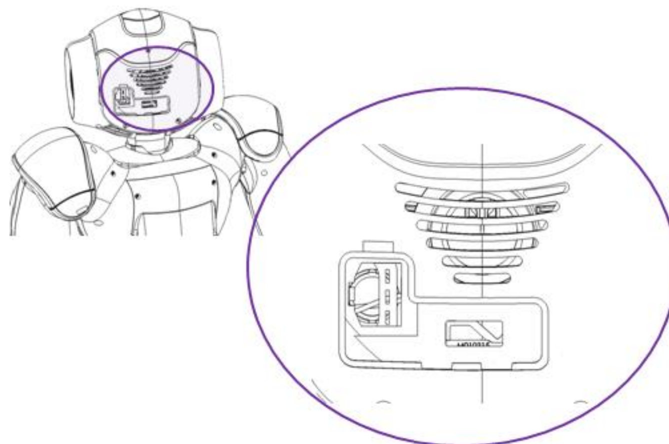
Další vnitřní senzory a čidla mají za úkol zjišťovat například polohu jednotlivých motorů nebo teplotu procesoru, aby nedošlo k mechanickému poškození. Na základě dat z inerciální jednotky, ze senzorů a ostatních čidel, řídí operační systém jednotlivé motory tak, aby jednak uspokojit požadavky uživatele, ale hlavně aby se pohyboval bezpečně a prováděl úkony, které nijak neohrozí robotovy součásti. Celkem robot disponuje 25 motory o 5různých parametrech a desítkami různých čidel a senzorů, včetně sonaru [1].

3.2.1 Možnosti připojení

Robot nabízí několik možných způsobů, jak s ním komunikovat, Hlavní komunikace probíhá přes bezdrátové připojení WiFi. Robot ve verzi 6 podporuje protokol IEEE 802.11 a/b/g/n

Pokud víme, že se robot nebude pohybovat, tak se nabízí způsob připojení přes ethernetový port. Tento ethernetový port je umístěný na zadní straně hlavy robota a používá se hlavně při aktualizaci robota, nebo obecně když je potřeba přenést velké množství dat.

Případné periferie jako jsou například USB disky nebo třeba Arduino lze připojit přes USB port také umístěný na zadní straně hlavy robota [2].

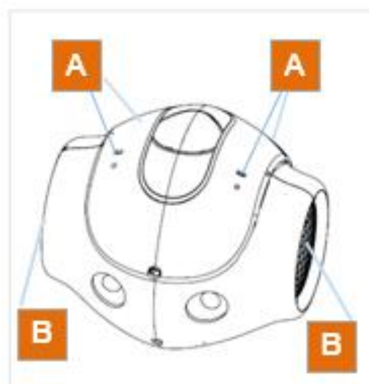


Obrázek 2 Umístění portů [2]

3.2.2 Zvuková a vizuální technika

Robot Nao je vybaven dvěma stereo reproduktory (B), které jsou umístěny na jeho hlavě. Tyto reproduktory bude robot v mé aplikaci převážně využívat na přehrávání mluveného textu v češtině.

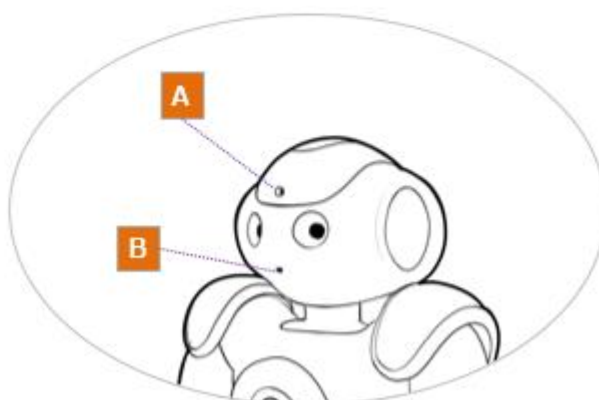
Na hlavě jsou také umístěny 4 všesměrové mikrofony (A), pro které zatím nemám využití, ale v budoucnu by bylo možné vytvořit v aplikaci widget na získávání zvukového vstupu.



Obrázek 3 Umístění mikrofonů a reproduktorů [2]

Robot také disponuje sadou několika LED diod, které se nachází v očních důlcích a v oblasti uší na okraji reproduktorů [1].

Robot disponuje dvěma kamerami s rozlišením až 2560x1920 v rychlosti jeden snímek za sekundu, tento režim je zejména vhodný k pořizování fotografií. Ke snímání videozáznamu a práci s ním, robot používá rozlišení 640x480 v rychlosti 30 snímků za sekundu [2].



Obrázek 4 Umístění kamer [2]

3.2.3 Tlakové senzory

Na robotovi se nachází několik tlakových senzorů (tlačítek), které jsou určeny pro interakci s uživatelem. Hlavní tlačítko se nachází na hrudi, přes toto tlačítko dokážeme robota vypnout/zapnout nebo po krátkém stisknutí tohoto tlačítka nám robot je schopen ohlásit informace o své IP adrese, nebo podrobnosti o chybě, pokud nějaká nastala [1].

Na hlavě jsou umístěna hned troje tlačítka, takže lze snímat například pohlazení. Na každé ruce se nachází další tři tlačítka sloužící třeba k vedení robota za ruku a na každém chodidle nalezneme mechanické tlačítko tj „Bumper“ sloužící například k detekci překážek [1].

4 Software

4.1 Software uvnitř robota

4.1.1 Naoqi OS

Naoqi OS je operační systém, který běží na robotovi. Je to GNU/Linux operační systém založen na distribuci Gentoo, který je speciálně upraven pro vývoj robotů od Softbankelectronics. Naoqi OS vychází z původního operačního systému OpenNAO [1].

Hlavní uživatel je „nao“ a jako u každého operačního systému Linux je zde i superuživatel „root“. Z továrního nastavení jsou hesla k účtům stejná, jako jsou jejich názvy [1].

Naoqi OS spravuje a spouští několik programů, mezi ty zásadní patří:

- conman: síťový manager
- NAOqi: software, který mimo jiné dovoluje robotovi se hýbat

4.1.2 NAOqi

NAOqi je název hlavního software, který běží na robotovi a kontroluje robotův pohyb. Tento software může běžet jak na robotově operačním systému Naoqi OS, nebo přímo na počítači, kde se dá použít k otestování kódu na simulovaném robotovi třeba v prostředí Webots. NAOqi se automaticky zapíná ihned po spuštění robota [2].

Robotův start se řídí skriptem, který je umístěn /etc/init.d/naoqi a je zde defaultně nastaveno:

- Rozsviť oči
- Spusť NAOqi

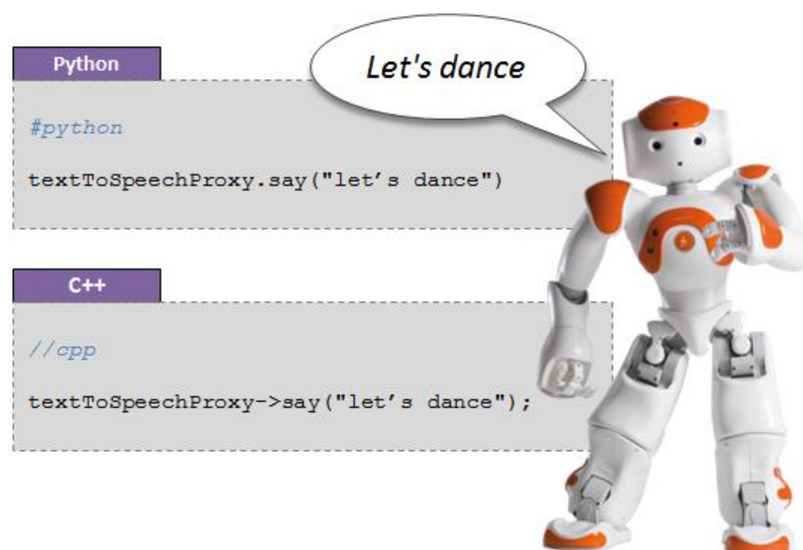
NAOqi řeší nejen robotovu logiku, jak bude postupovat po přijmutí zadané instrukce, ale i řídí veškeré robotovi interakční procesy včetně autonomního chování robota. To znamená, že když zrovna nemá robot žádné zadané instrukce a tím pádem by měl být v naprostém klidu a neměl by se nijak hýbat, tak i přesto bude provádět zdánlivě náhodné pohyby, které jsou velice podobné chování člověka.

4.1.3 NAOqi framework

Naoqi framework je softwarová struktura, která slouží jako podpora při programování, vývoji a organizaci jiných softwarových projektů. Obsahuje podpůrné programy, knihovny API apod. Zprostředkovává běžné potřeby při robotizaci, jako je paralelismus, správa prostředků, synchronizace, zpracovávání událostí. Tento framework nám dovoluje navázat homogenní komunikaci mezi jednotlivými moduly (pohyb, audio, video), homogenní programování a sdílení informací [2].

NAOqi framework je cross-platformní, což znamená, že je možné tento framework používat na všech standardních platformách jako je Windows, Mac nebo Linux. Ve všech případech volání programové metody je naprosto totožné. Software pro NAOqi framework může být vyvíjen v jazycích C++ a Python Všechny dostupné API mohou být nezávisle volány pomocí jakéhokoli podporovaného jazyka [2].

Dostupné API jsou zobrazeny ve webovém prohlížeči. Stačí zadat robotovu IP adresu a port 9559 – například *http://127.0.0.1:9559*. Robot zobrazí své seznamy modulů, seznam metod, parametry metod, popis a příklady [2].



Obrázek 5 Stejné API v C++ a v Pythonu [4]

Při tvorbě mé aplikace jsem postupoval podle starší dokumentace dostupné k zvolenému SDK, proto používám starší framework NAOqi a ne novější framework qi. Nové vlastnosti a funkce novějšího frameworku jsou, že podporuje mapy, listy, objekty a struktury. Tento framework již nepotřebuje modul, aby dokázal reagovat na události a dovoluje asynchronní volání s návratovou hodnotou [4].

Tento novější framework jsem objevil až poté, co jsem vytvořil svoji aplikaci ve starším plně funkčním frameworku NAOqi. Novější framework je o něco složitější a nesnadno by se na něj zpětně přecházelo, a hlavně k tomu není žádný důvod.

API a moduly, které nás budou z frameworku NAOqi zajímat jsou:

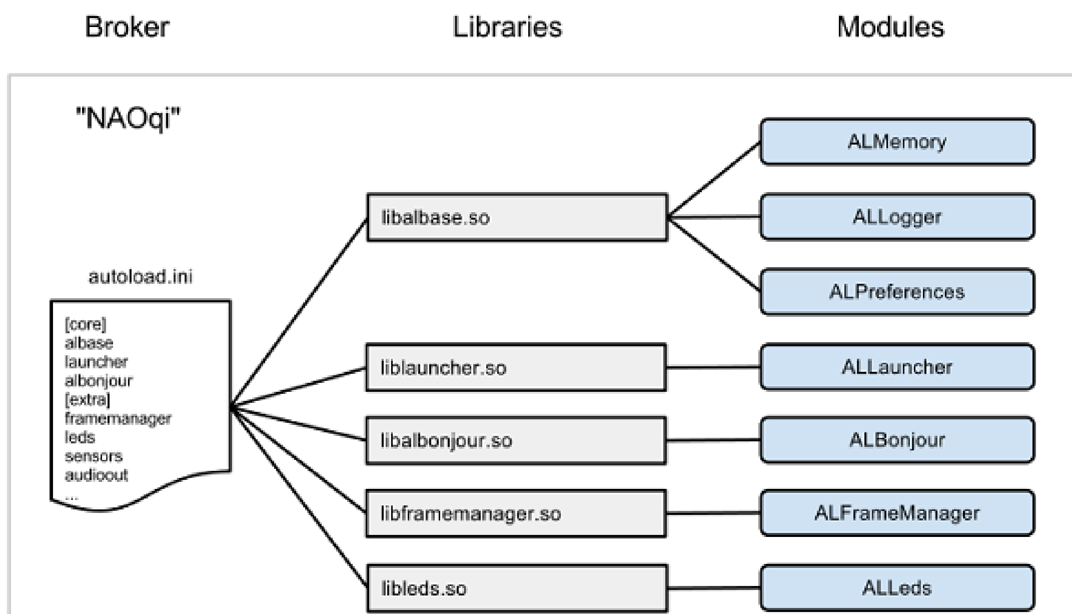
- ALProxy
- ALMotion
- ALTextToSpeech

ALProxy vytvoří objekt Proxy pro dané API (více v kapitole o proxy). ALMotion poskytuje metody, které pomáhají robotovi s pohybem. Obsahuje příkazy, které nám dovolují nastavovat tuhost kloubů, úhel kloubů a vyšší stupeň API dovolující nám kontrolovat robotovu chůzi. ALTextToSpeech použijeme k převodu českého textu na řeč [4].

4.2 Síťová komunikace

4.2.1 Broker

Broker je NAOqi spustitelný soubor, který, když se spustí, tak načte soubor autoload.ini, podle kterého systém načte preferované knihovny. Každá knihovna obsahuje jeden, nebo více modulů, které používají broker k nabízení svých metod a funkcí napříč sítí [4].



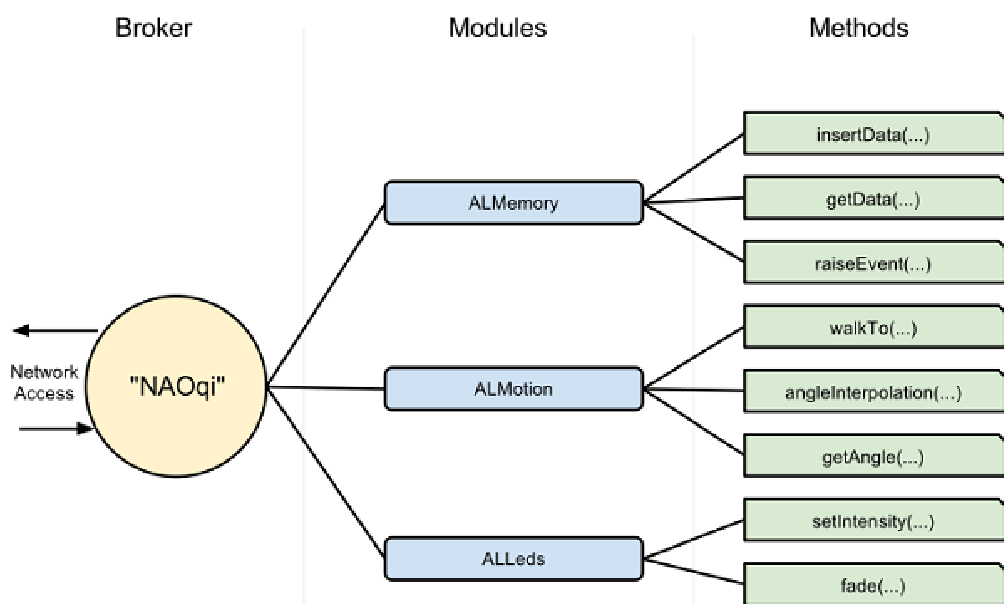
Obrázek 6 Soubor Autoload.ini [4]

Broker poskytuje vyhledávací služby, takže dokáže nalézt jakýkoli modul ve stromu, nebo napříč sítí je schopen nalézt metody, které jsou nabízeny. Načítání modulů vytváří strom metod, které jsou připnuté k jednotlivým modulům a tyto moduly jsou připnuté k brokeru [4].

Broker poskytuje vyhledávací službu. Každý modul a jeho metody jsou uspořádány do stromové struktury, proto je lze snadno nalézt i napříč sítí, pokud byli zavolány [4].

Ve většině případů nemusíme o brokerech vůbec vědět, protože dělají svou práci, aniž by to uživatel věděl. Brokery nám dovolují psát kód, který bude stejný jak pro volání lokálních modulů (ve stejném procesu) tak pro volání externích modulů (v jiném procesu, nebo na jiném zařízení) [4].

V mojí aplikaci broker reprezentuje software uvnitř robota, který řeší veškerou síťovou komunikaci. Například vyhledávání a volání jednotlivých modulů.



Obrázek 7 Broker – Moduly – Metody

4.2.2 Proxy

Proxy je síťový objekt, který se bude chovat jako modul, který reprezentuje. Například pokud vytvoříme proxy modulu ALMotion. Dostaneme objekt, který bude obsahovat všechny metody z modulu ALMotion [4].

K vytvoření proxy k nějakému modulu, abychom dokázali volat jeho metody, máme 2 možnosti:

- Jednoduše použijeme název modulu. V tomto případě kód, který posíláme a modul, ke kterému se chceme připojit, musí být pod stejným brokerem. Tento princip se nazývá lokální volání (local call). Tento způsob použijeme v případě, že už jsme připojeni a pouze voláme další metodu z jiného modulu pod stejným brokerem [4].
- Použijeme název modulu, ale k tomu i IP adresu a port daného Brokeru. Zvolený modul se musí nacházet v daném brokeru [4]. Tento způsob volíme například, když voláme daný modul z počítače.

4.2.3 Moduly

Typicky každý modul je třída uvnitř knihovny. Když je knihovna nahrána ze souboru autoload.ini, automaticky vytvoří instanci třídy modulu. Moduly se rozdělují na lokální a externí.

Lokální moduly jsou takové, které jsou spuštěny v tom samém procesu a jejich komunikace probíhá přes právě jeden broker. Protože jsou lokální moduly spouštěny jedním procesem, mohou mezi sebou sdílet proměnné a volat metody ostatních lokálních modulů, aniž by došlo k serializaci, nebo k použití sítě. Komunikace mezi lokálními moduly je rychlejší než komunikace modulů externích [4].

Externí moduly jsou moduly, které komunikují prostřednictvím sítě. Externí modul potřebuje broker, nebo proxy, aby mohl mluvit s ostatními moduly. Broker je zodpovědný za veškerou komunikaci. Externí moduly pracují se sítí za pomoci protokolu SOAP (Simple Object Access Protocol). Externí modul se nesmí používat jako rychlý přístup k datům (vyžádat si přímo data z paměti) [4].

Připojení mezi externími moduly:

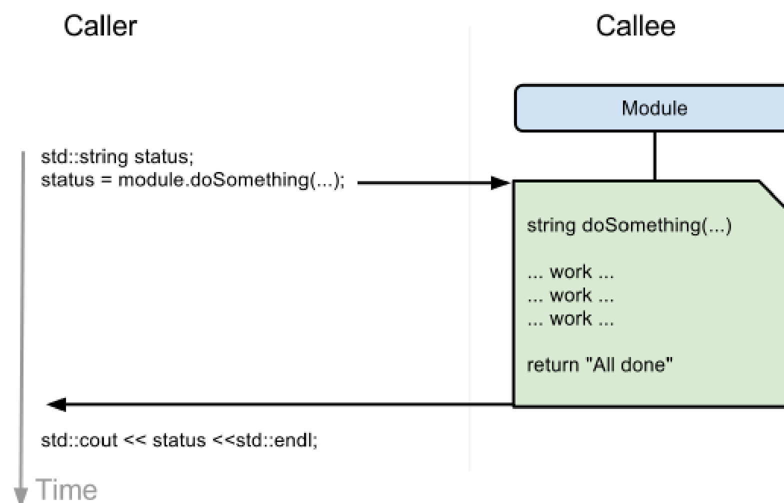
- Připojení Broker – Broker, které naváže obousměrnou komunikaci
- Připojení Proxy – Broker, které naváže jednosměrnou komunikaci

Pokud bychom chtěli nahrát do robota náš vlastní modul, nebo bychom chtěli na robotovi používat modul, který není standardně nahráván do paměti při spouštění NAOqi, musíme robotovi explicitně říci. K tomuto účelu nám slouží právě soubor autoload.ini. Tento soubor na adrese /home/nao/naoqi/preferences/autoload.ini obsahuje veškeré knihovny, které se nahrají do paměti při spuštění softwaru NAOqi [4].

Při tvorbě mojí aplikaci vytvářím připojení proxy – broker a navazuji jednosměrnou komunikaci mezi mojí aplikací a robotovým brokerem skrz který vytvářím jednotlivé objekty proxy pro jednotlivé API a volám tak jednotlivé metody.

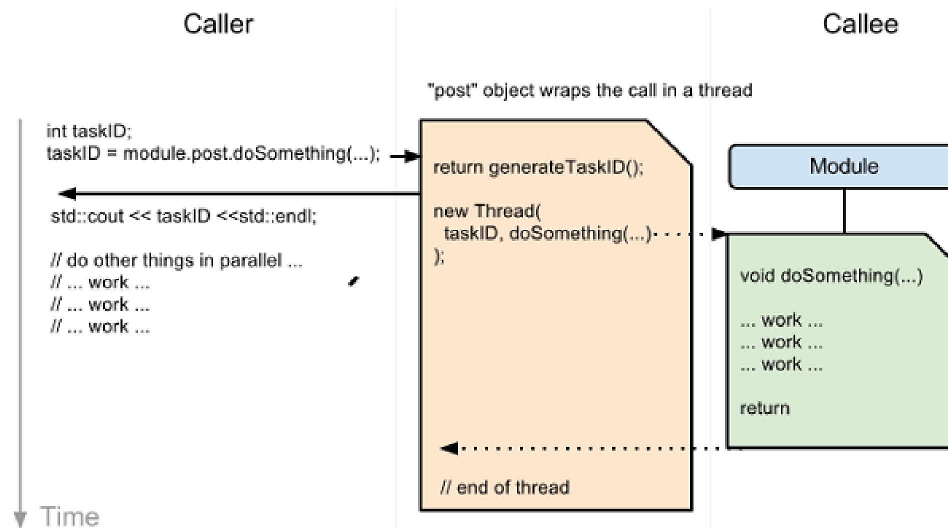
4.2.4 Blokované a neblokované volání funkcí

Framework NAOqi nabízí dva způsoby volání funkcí a metod. Blokované volání funkcí, každá další instrukce bude zpracována hned poté co předchozí skončí. Veškeré volání mohou způsobit výjimku a měli by být umístěny do bloku try-catch. Volání funkcí může mít návratovou hodnotu [4].



Obrázek 8 Příklad blokovaného volání metody [4]

Neblokované volání funkcí, přes užití objektu proxy „post“ bude příkaz vytvořen v paralelním vláknu. Toto nám dovoluje dělat několik věcí najednou (mluvit a chodit zároveň). Každé takovéto volání generuje unikátní „task id“. Podle tohoto id můžeme zjistit, zdali daný příkaz běží, nebo čeká, než skončí [4].



Obrázek 9 Příklad neblokovaného volání metody přes objekt "post" [4]

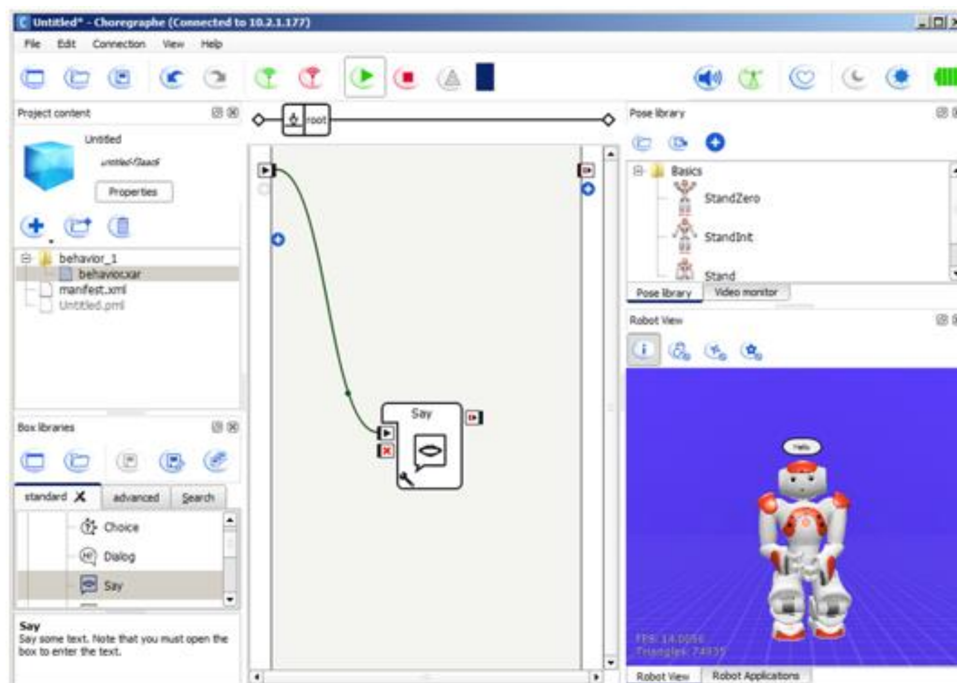
4.3 Software pro uživatele

4.3.1 Choreographe

K robotovi se v základu využívá jeho oficiální software Choreographe, tento způsob je nejjednodušší, ale pro nového uživatele je ze začátku nepřehledný a těžko stravitelný. Tento software umožňuje vytvářet jednotlivé programy a ovládat robota.

Programování a ovládání probíhá přes předdefinované bloky, které se následně spojují do sekvencí a vytvářejí komplexní program. Uživatel může vytvářet od jednoduchých programů, jako jsou třeba jednotlivé demonstrace funkcí programovatelných částí rukou, nohou, reproduktorů až po složité programy na rozpoznávání obličeje, rozpoznání mluveného slova a vedení konverzace. Jakmile máme námi vytvořený program hotový, tak se tento program nahraje do paměti robota, většinou přes bezdrátové připojení, konkrétně přes protokol TCP nebo přes USB kabel.

Následně se dá takovýto program na robotovi spustit přes předdefinovanou událost například pohlazení hlavy. Tento program poté běží na vnitřním systému robota (NAOqi) a robot nemusí být připojený k počítači.



Obrázek 10 Program Choreographe [7]

Z mého pohledu největší výhodou programu Choreographe je, že nabízí pohled na stav robota v reálném čase. Přímo v aplikaci je tedy vidět, zdali je robot připojený, jak moc je robot nabitý, v jaké se zrovna nachází pozici nebo co aktuálně snímají jeho dvě kamery. Hlavní nevýhodou programu Choreographe dle mého názoru je, že jak pro začínajícího uživatele, tak pro pokročilého je hodně obsáhlý, a proto vytváření jednoduchých úloh a příkazů trvá značnou dobu.

4.3.2 Vysvětlení API a SDK

API je označení jednoduchého prostředí, které nám dovoluje komunikovat s dalším softwarem, nebo jeho funkcemi. Tento název je odvozen od anglického spojení Applicacion Programing Interface. O API můžeme přemýšlet tak, že máme dva rozdílné stroje, které mluví svými jazyky a mají své vlastní sady instrukcí. API zastává funkci překladače, aby se mezi sebou dokázali domluvit. Řečeno jednoduše, API je sada instrukcí, které dosáhnou požadovaného výsledku. Jednoduchý příklad používání API je kopírování textu ve Windows. Přes zkratku Ctrl + C aktivujeme API, které uloží požadovaný text do paměti RAM, tyto data stejné API přenese do jiné aplikace. Pokud budeme chtít vložit uložený obsah, zavolá se další API, které data vyrendruje a vloží do zvolené aplikace [8].

SDK neboli Software Developer Kit je sada vývojářských nástrojů. SDK si můžeme představit na příkladu lepíme model auta nebo letadla. Když chceme slepit model letadla, potřebujeme k tomu několik nástrojů, samotné díly letadla, lepidlo, návod a mnoho dalšího. Takto můžeme přemýšlet i o SDK je to sada nástrojů, které nám slouží vytvářet softwarové aplikace na specifické platformě [8].

Pokud jsou API stavební bloky, poté SDK je celá dílna včetně těchto bloků, která slouží k tvorbě software na dané platformě [8]. Dostupné SDK pro framework NAOqi jsou pouze pro jazyky Python 2.7, C++ a Javascript [3].

Programming Languages	Bindings running on		Choregraphe support	
	Computer	Robot	Build Apps	Edit code
Python	✔	✔	✔	✔
C++	✔	✔	⃠	⃠
JavaScript	✔	✔	✔	⃠
ROS	✔	⃠	⃠	⃠

✔	OK
⃠	Not available

Obrázek 11 Podporované programovací jazyky [3]

4.3.3 Dostupná API

Jelikož knihovna naoqi používá celý framework NAOqi, tak existuje veliká spousta jeho API. Tyto API dokážeme rozdělit do několika skupin, podle jejich účelu a co přesně zajišťují. Rozdělují se na: Core, Emotion, Interaction engines, Motion, Audio, Vision, People perception, Sensors & LEDs.

Z těchto API mě převážně zajímali API ovládající pohyb (Motion) a zvuk (Audio). Nejvíce zásadní však byl modul ALProxy, který na určité IP adrese a portu vytváří objekt proxy z daného API.

5 Vývoj aplikace

5.1 Požadavky na aplikaci:

Mezi první požadavky na aplikaci bude požadavek Připojit se k robotovi“. Tento požadavek nevyjadřuje způsob připojení (USB kabel, Wi-Fi, …), ale způsob identifikace robota v síti. Když jsem přemýšlel nad realizací toho požadavku, tak mě napadl přístup, že by bylo možné skenovat veškeré IP adresy v rozsahu a následně dohledat IP adresu robota. Nakonec jsem se ale rozhodl, využít toho, že robot mi sám dokáže říci svou IP adresu, po stisknutí jeho tlakového senzoru na hrudi. Tudíž tuto IP adresu nemusím hledat. Ve své aplikaci proto používám přímé zadání robotovi IP adresy.

Další požadavek, pokud již znám robotovu IP adresu, bude s robotem nějak komunikovat. Komunikovat ve smyslu získávání a posílání dat robotovi skrze nějaký sítový protokol. S ohledem na bezpečnost nebudu chtít používat nešifrované komunikační protokoly jako je například protokol Telnet. Příhodné je, že knihovna naoqi obsahuje metodu ALProxy, která dokáže posílat data skrze protokol SOAP. SOAP je síťový protokol určený pro výměnu zpráv založených na XML, hlavně pomocí HTTP. Formát SOAP tvoří základní vrstvu komunikace mezi webovými službami a poskytuje prostředí pro tvorbu složitější komunikace.

Když už dokážu robotovi poslat nějaká data, tak další požadavek na aplikaci bude, aby posílala robotovi určité příkazy a dokázala zjistit, zda byly korektně provedeny. Příkazy jsem ze všech možných vybral ty, které nepracují s výsledky předchozího robotova příkazu a dají se spouštět nezávisle na ostatním dění. Mezi mé příkazy patří příkazy ovládající robotův pohyb a jeho hlasový projev. Myslím si, že jsem vybral stěžejní příkazy, které vzhledem k humanoidnímu vzhledu robota by mohli diváka, popřípadě žáka zaujmout. Konkrétně se jedná o příkazy: Stoupni si, Sedni si, Lehni na břicho, Lehni na záda, Otoč se doprava/doleva, Chůze na určitou vzdálenost, Převést zadaný český text na řeč, Přečíst český text ze souboru.

Další požadavek na aplikaci bude z těchto příkazů v jednoduchém grafickém editoru tvořit sekvence a tvořit z nich soubory (dále jen programy), které půjdou ukládat a načítat. Tyto programy bude moje aplikace schopna opakovaně pouštět a také zjistí, zda byly korektně provedeny. Grafický editor bude tyto programy vykreslovat jako bloky jednotlivých příkazů

5.2 Tvorba aplikace pro Android

Původně jsem zamýšlel aplikaci vyvíjet pro operační systém Android, ale po dlouhém zkoumání jsem od toho byl nucen upustit. Hlavní překážkou, na kterou jsem narazil, bylo chybějící SDK a také to, že všechny API umožňující ovládání robota, jsou podporované pouze pro jazyky C++ a Python 2.7. Neexistuje tedy žádné oficiální SDK, umožňující programování aplikaci v Javě, či jiných programovacích jazycích, které se používají k vývoji aplikací pro operační systém Android.

Při mém výzkumu jsem našel dvě funkční aplikace pro operační systém Android. První byla vytvořena tureckým vývojářem a byla to webová aplikace, která používala robotův framework a přímo ovládala robotovi funkce, ale nedokázal jsem zjistit, jak přesně funguje, ale pravděpodobně používá Javascript.

Druhá aplikace byla vytvořena americkou univerzitou státu Alabama v roce 2013. Při zkoumání jejich práce jsem narazil na NAOqi knihovnu psanou v jazyku Java nazvanou jNAOqi.jar. Tato knihovna by byla schopna volat NAOqi API. Bohužel je tato knihovna jednak zastaralá, a nebyla vytvořena proto, aby fungovala na virtuálních strojích, které běží na zařízení Android. [9]

Výslednou aplikaci vytvořili hodně hackerským způsobem, použili oficiální software Choreographe a wireshark, skrze který odposlouchávali síťovou komunikaci mezi počítačem a robotem. Bohužel jsem nedokázal dohledat, jak následně data dekódovali. Když jsme se s vedoucím práce pokusili jejich přístup zopakovat, tak zachycená data byla rozsáhlá a v binární podobě, tudíž pro nás nepoužitelná.

Poté, co jsem nenašel žádný funkční způsob, který by nebyl složitější než odposlouchání a dekódování síťové komunikace, jsme se s vedoucím práce domluvili, že výslednou aplikaci vytvořím pro operační systém Windows v jazyku Python

5.3 Python

Při tvorbě aplikace v jazyku Python, bylo důležité použít správnou verzi jazyka. Jazyk Python byl vyvíjen ve dvou hlavních větvích, ve verzích 2.x a 3.x. V dnešní době se nejvíce využívá verze 3.x a verze 2.x ke konci roku 2020 skončila býti podporovaná [13].

Bohužel Softbank robotics používá ve svém SDK verzi pythonu 2.7. Tato verze i když již není aktuální, pořád funguje, ale oproti své novější verzi používá trochu jinou syntaxi. Velikou překážkou pro mě bylo hledání dokumentace jednotlivých knihoven a funkcí. Například když jsem hledal dokumentaci ke knihovně Tkinter (přes kterou tvořím veškeré GUI aplikace), tak většina zdrojů byla psána pro verzi 3.x a bylo časově náročné nalézt podobnou dokumentaci pro starší verzi.

Další nevýhodou této verze je, že proměnné typu string jsou ukládány v kódování ASCII, a ne v UNICODE, jako je to u pythonu 3.x. Tento problém jsem částečně odstranil nadepsáním hlavičky souboru, kde jsem definoval defaultní kódování na UTF-8. Bohužel tato metoda nebyla všemocná a objevil se mi problém s kódováním, když jsem předával string jako parametr jiné funkci (získávání vstupů z klávesnice). Daná proměnná se i nadále ukládala v ASCII kódování, i když byla na začátku kódu nadepsána správná hlavička. Tento problém jsem vyřešil explicitním zadáním kódování dané proměnné při jejím zápisu.

5.4 Konfigurace Pythonu, SDK

Python 2.7 –32bit jsem stáhnul z oficiálních stránek http://Python.org/download/. Při programování používám vývojové prostředí IDLE, které je přímo součástí jazyku Python. Toto IDE nenabízí žádné pokročilé funkce, jako je doplňování příkazů, proměnných apod. Toto IDE jsem zvolil kvůli jeho jednoduchosti. Ještě jsem přemýšlel o Microsoft Visual Studio Code, ale protože jsem v tomto prostředí už v minulosti pracoval, vím, že není jednoduché se v tomto IDE vyznat a moje práce by se míchala s ostatními projekty v jiných jazycích.

Python SDK jsem stáhnul od výrobce z jeho oficiálních stránek pro vývojáře [3]. Je to SDK pro Python verzi 2.7 – 32bits.

Operační systém	Verze
Linux	Ubuntu 16.04 Xenial Xerus – 64bit
Windows	Microsoft Windows 10 64bit
Mac	Mac OS X 10.12 Sierra

Tabulka 1 Podporované operační systémy [10]

Stažený soubor jsem rozbalil a poznamenal si jeho cestu. Nejdůležitější při instalaci SDK do našeho pythonu je potřeba správně nastavit systémovou proměnnou, aby Python věděl, kde má dané SDK hledat. K tomu slouží systémové proměnné. Seznam těchto proměnných u operačního systému Windows jsem našel v systémovém nastavení. Do seznamu systémových proměnných jsem přidal proměnnou s názvem PYTHONPATH a jako její hodnotu jsem uložil cestu k používanému SDK [3].

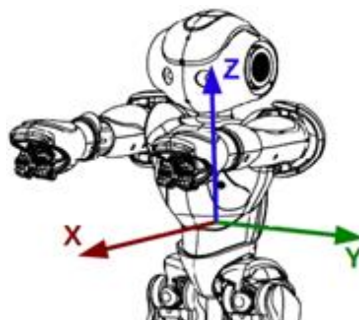
Díky této proměnné jsem schopen importovat knihovny z používaného SDK, jako kdyby to byly knihovny pythonu. Z SDK používám knihovnu naoqi, která používá framework NAOqi. Import je jednoduchý, pouze na začátku souboru je potřeba uvést buď `import naoqi`, nebo `from naoqi import modul`.

5.5 Použité API a moduly z knihovny naoqi

5.5.1 Pohyb

Ze skupiny API, které souvisí s robotovým pohybem, jsem použil právě dvě, a to ALRobotPosture a ALMotion.

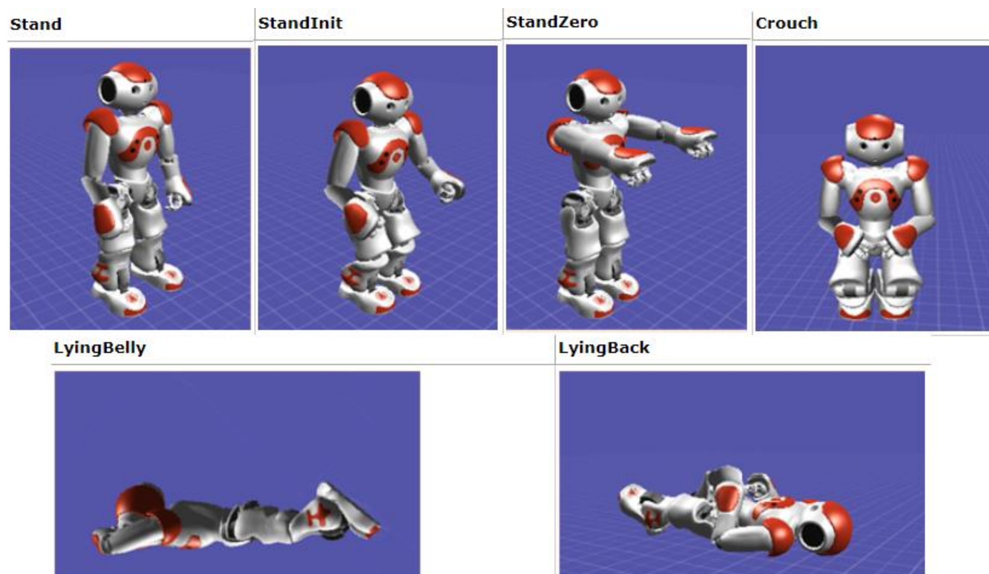
ALMotion, je obecné API pro ovládání veškerého pohybu, včetně ovládání motorů, tuhosti a pozice kloubů. Toto API má také na starosti chůzi. Toto API má v sobě zabudované ochranné prvky jako je například ochrana před srážkou při pohybu s vlastními částmi těla a ochranu motorů. Toto API používám, když chci, aby se robot otočil, nebo šel dopředu na určitou vzdálenost. Z tohoto obsáhlého API používám příkazy `setStiffnesses(část těla, hodnota)`, tento příkaz uvolní robotovy svaly, aby se mohl začít pohybovat. Část robotova těla používám „Body“, což je souhrnné označení celého těla. Při uvolnění kloubů musí být robot ve stabilní poloze a na tento příkaz musí navazovat další funkce, která bude určovat, jak se robot bude hýbat, jinak může hrozit robotův pád [10].



Obrázek 12 Zobrazení Os [10]

Funkce, která u mé aplikace navazuje na povolení kloubů je `moveTo(vzdálenost X, vzdálenost Y, otočení podle osy Z)`. První parametr určuje vzdálenost, o kterou robot půjde dopředu. Druhý parametr určuje, jak moc bude robot při svém pohybu dopředu zatáčet. U prvních dvou parametrů jsou jednotky udány v metrech. Poslední parametr určuje, úhel, o který se robot na místě otočí po směru hodinových ručiček podle osy Z. Základní jednotkou je radián. Pokud po této funkci chceme, aby robot šel dopředu o jeden metr, dáme této funkci parametry (1,0,0) [10].

ALRobotPosture jak napovídá název, ovládá robotovu polohu, toto API je nadstavba API ALMotion, kde se již přímo neovládají jednotlivé motory, ale pouze se pošle příkaz, do jaké předdefinované pozice se má robot dostat, a o ostatní se postará samo API [11]. Konkrétně z tohoto API použiji funkci `goToPosture(pozice, rychlost)`. Pozice jsou předefinovány a mají jednoznačný název viz obrázek 15.



Obrázek 13 Předdefinované polohy [11]

5.5.2 Zvuk a hlas

Z Naových Audiovizuálních zařízení budu potřebovat pouze reproduktory, ty budu chtít ovládat skrze API, které dokáže zpracovat psaný text do audio podoby. K tomuto účelu slouží API ALTextToSpeech, defaultně má toto API nastaveno za používaný jazyk angličtinu (Pokud není nakonfigurován jinak), ale podporuje spoustu světových jazyků včetně češtiny [12]. Použití tohoto API ukazuji na funkci `mluv()` v kapitole 5.6.1

5.5.3 Vytvoření proxy

Ve svém kódu pokaždé, když potřebuji používat nějaké API, tak veškerou komunikaci probíhá přes modul ALProxy, který s robotem naváže spojení typu Proxy – Broker. Tento modul vytváří proxy ze zadaného API, které je udáno jeho unikátním jménem.

Pokud tedy ve svém kódu mám něco takového jako: `motion = ALProxy('ALMotion', ip, port)`. Poté motion je objekt typu proxy, který v sobě má veškeré API z modulu ALMotion a má nastaveno IP a port kam bude odesílat zadané instrukce.

5.6 Návrh a vývoj aplikace

Aplikace je rozdělena na dvě části, na spouštěč jednotlivých příkazů a editor. Ze začátku jsem tyto části vytvářel odděleně, až poté, co jsem si byl jistý jejich funkčností, jsem je spojil do jedné. Obě dvě části využívají stejné příkazy, tak jsem si zjednodušil práci tím, že jsem vytvořil modul prikazy.py, který realizuje jednotlivé příkazy. Tento soubor jsem vytvořil zvlášť a z výsledné aplikace pouze volám jeho funkce. Hlavní výhodou tohoto přístupu je, že takto mohu upravovat, nebo vytvářet další funkce a nemusím přímo zasahovat do zdrojového kódu vlastní aplikace a také to napomáhá přehlednosti kódu. V tomto souboru již pracuji s SDK a jeho API, které používám k vykonávání jednotlivých příkazů.

5.6.1 Modul prikazy.py

Touto částí mé práce jsem začínal. Nejdříve jsem testoval, zdali funguje výrobcem poskytnuté SDK a dokáži robotovi poslat nějaké instrukce. Ze začátku moje aplikace neměla vůbec žádné GUI, byli to pouze mnou vytvořené scripty, které když jsem spustil, tak robot provedl nějakou akci. Až posléze, když jsem měl představu, jak se s daným SDK a jeho API pracuje, jsem z těchto scriptů sestavil jednotlivé funkce. Vytvořil jsem tak podpůrný soubor procedur a nazval ho prikazy.py.

Tento soubor je psán v pythonu 2.7 a používá SDK a jeho API. V tomto souboru řeším veškerou interakci s robotem. Program pracuje s robotem Nao prostřednictvím knihovny naoqi z importovaného SDK. Program má v sobě přesně 6 funkcí, z toho 5 z nich přímo pracuje s knihovnou naoqi a vytvářejí třídy proxy z jednotlivých API.

Jednotlivé funkce dostávají celkem 3 parametry, které obsahují informace o IP adrese, portu a požadovaném úkonu.

K vytváření objektů proxy používám z knihovny naoqi modul ALProxy, který potřebuje 3 parametry, první je název API, ze kterého vytvářím proxy, druhý a třetí jsou informace o IP adrese a portu. Nejlépe to budu demonstrovat na mojí funkci `pozice()` viz zdrojový kód 1.

V tomto kuse kódu nejdříve importuji z knihovny naoqi, definuji funkci `pozice()` se třemi parametry. Protože modul ALProxy bere IP adresu jako string a port jako integer, definuji, že proměnná port je opravdu integer (číslo). Dále se pokouším vytvořit objekt proxy na zadané IP adrese a portu. Tento objekt, který jsem si nazval `posture`. vytvářím přes modul ALProxy a jako první parametr zadávám jméno modulu, které chci použít. Teď, když mám vytvořená objekt proxy na API ALRobotPosture, mohu použít API z tohoto modulu. Konkrétně používám API goToPosture(pozice, rychlost). Defaultně jsem udal 80% rychlost kvůli bezpečnosti

```
from naoqi import ALProxy
def pozice(ip, port,pozice):
  port=int(port)
  try:
    posture = ALProxy("ALRobotPosture", ip, port)
  except Exception, e:
    k=str(e)
    tkMessageBox.showerror("Chyba", "Nedokázal jsem se
    připojit \n" + k)
    return 1try:
    posture.goToPosture(pozice, 0.8)
  except Exception,e:
    x=str(e)
    tkMessageBox.showerror("Chyba", "Z nějakého důvodu
    nemohu provést požadovanou akci \n" + x)
    return 2
  return 0
```

Zdrojový kód 1 Funkce pozice()

Postupně jsem takto stejným způsobem vytvořil ostatní funkce. Druhá takto vytvořená funkce byla funkce `mluv(ip, port, text)`, která pracuje s API ALTextTo-Speech. Tato funkce nejdříve z daného API vytváří objekt proxy, který jsem nazval `tts`. Následně z tohoto API volám příkaz `setLanguage("Czech")`, kterým nastavuji český jazyk a následným příkazem `say(text)`, posílám robotovi text, který má přečíst.

```
def mluv(ip, port, text):
    port=int(port)
    try:
      tts=ALProxy("ALTextToSpeech", ip, port)
      tts.setLanguage("Czech")
      tts.say(text)
    except Exception,e:
      k=str(e)
      tkMessageBox.showerror("Chyba", "Nedokázal jsem se připojit:\n" + k)
      return 1
  return 0
```

Zdrojový kód 2 Funkce mluv()

Další funkci, kterou jsem vytvořil byla funkce `jdi(ip, port, vzdalenost)` a `otoc(ip, port, hodnota ve stupních)`. Tyto funkce jsou kromě menších detailů totožné. Obě používají stejné API ALRobotPosture a ALMotion. Jediný rozdíl mezi těmito dvěma funkcemi je jejich zpracování třetího parametru, který funkce dostává. Ve funkci `jdi(ip, port, vzdalenost)` se vzdálenost, která je zadána od uživatele v centimetrech, převádí do metrů a hodnota se vkládá do prvního parametru API `moveTo(vzdalenost,0,0)`. Funkce `otoc(ip, port, hodnota)` zadaný vstup převádí ze stupňů do jednotek radiánu a hodnotu vkládá do třetího parametru API `moveTo(0,0,hodnota)`. Jelikož jsou si tyto dvě funkce tolik podobné, uvádím zde pouze zdrojový kód funkce `otoc(ip, port, hodnota)`.

Nejdříve jsem importoval knihovnu math, která obsahuje matematickou hodnotu čísla pí. Číslo pí potřebuji k převedení zadané hodnoty ve stupních na radiány. Po definování funkce `otoc(ip, port, hodnota)` upravuji vstupy tak, aby port byl integer, hodnota byla float a dalo se s ní počítat v reálných číslech. Na dalším řádku je již samotné převádění stupňů do radiánů. Poté, co mám ošetřené vstupy, vytvářím dva objekty typu proxy. První objekt tohoto typu bude používat modul ALMotion a druhý bude požívat modul ALRobotPosture.

Díky funkci `getPostureFamily()`, dokáži zjistit, v jaké pozici se robot nachází. To znamená, že pokud již robot stojí a je v připravené pozici “StandInit“, může se pohyb provádět okamžitě. V opačném případě je potřeba robota nejdříve do této pozice dostat. Aby se robot mohl hýbat, je nejdříve potřeba povolit jeho zafixované klouby příkazem `setStiffnesses("Body", 1.0)`, kde první parametr udává část těla a druhý, pokud je různý od nuly, povolí klouby. Poslední volám funkci `moveTo()`.

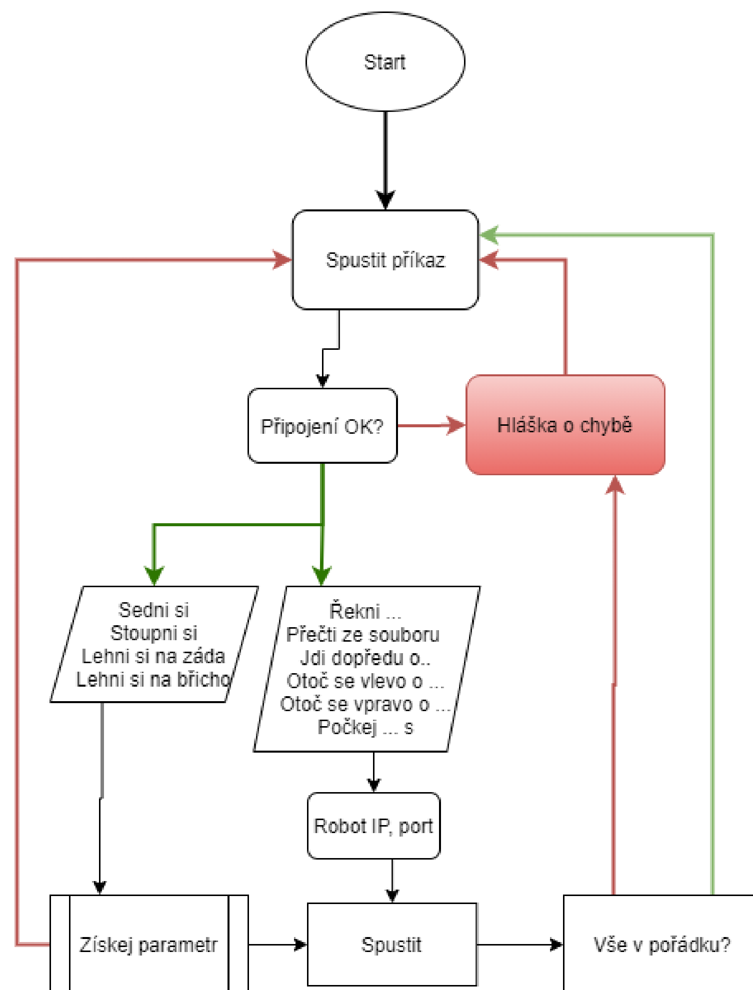
```
import math
def otoc(ip, port, hodnota):
  port=int(port)
  hodnota=float(hodnota)
  hodnota= hodnota*math.pi/180
  try:
    motion=ALProxy("ALMotion", ip, port)
    posture=ALProxy("ALRobotPosture", ip, port)
  except, Exception,e:
    k=str(e)
    tkMessageBox.showerror("Chyba","Nedokázal jsem se
připojit \n" + k)
    return 1
  try:
    if posture.getPostureFamily()=="StandInit":
      motion.setStiffnesses("Body", 1.0)
      motion.moveTo(0,0,hodnota)
    else:
      posture.goToPosture("StandInit", 0.8)
      motion.setStiffnesses("Body", 1.0)
      motion.moveTo(0,0,hodnota)
  except Exception, e:
    x=str(e)
    tkMessageBox.showerror("Chyba", "Z nějakého důvodu
nemohu provést požadovanou akci \n" + x)
    return 2
  return 0
```

Zdrojový kód 3 Funkce otoc()

5.6.2 Okamžité spuštění příkazu

Následující část mé aplikace byla tvorba základního uživatelského rozhraní (GUI) pro přímou interakci se souborem prikazy.py. Nejdříve jsem si vytvořil návrh, jak budu chtít, aby moje aplikace vypadala a jak bude zpracovávat jednotlivé funkce.

Schéma jsem navrhnul, aby nebylo možné používat jednotlivé příkazy, pokud nejsou správně zadané údaje o IP adrese a nebylo vyzkoušené připojení. Pokud jakýkoli příkaz skončí chybou, tak se vzniklá chyba vypíše. Pokud vznikla chyba, není možné zadat další příkaz a aplikace zablokuje veškerá tlačítka. K jejich aktivování je potřeba se znovu k robotovi připojit.

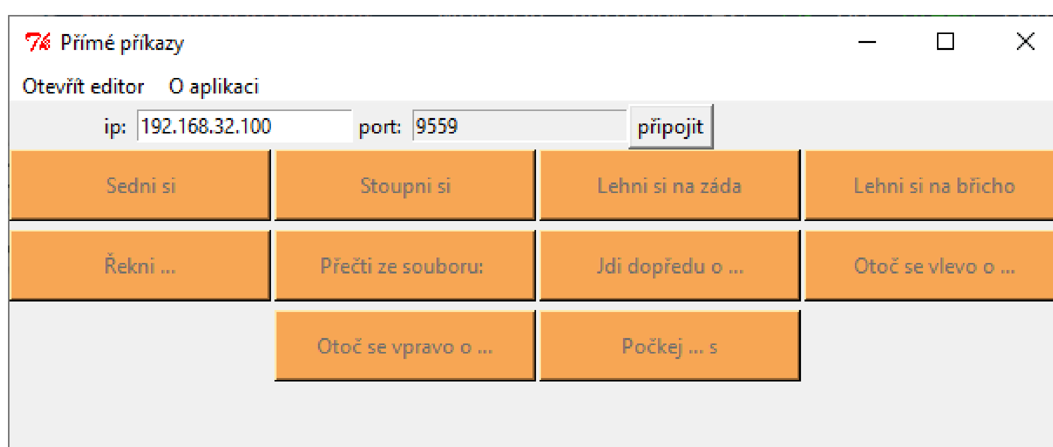


Obrázek 14 Diagram spustit příkaz

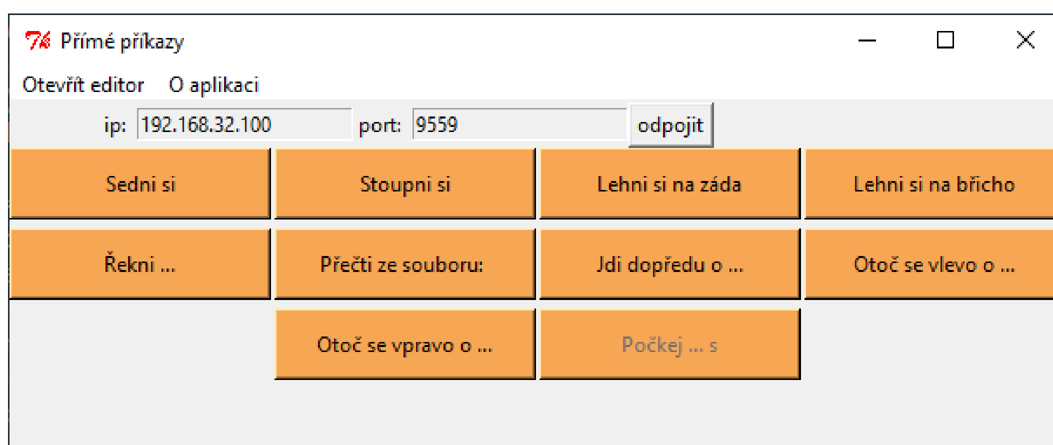
Veškeré GUI jsem tvořil pomocí knihovny Tkinter. Nahoře hned pod názvem aplikace je jednoduché menu. Toto menu obsahuje nabídku: Otevřít editor a O aplikaci. Pod tímto menu se nacházejí údaje o IP adrese, portu a tlačítko připojit/odpojit. Tlačítko připojit/odpojit se ukazuje podle aktuálního stavu připojení. Další na řadě je tabulka, do které jsou vloženy jednotlivé příkazy ve formě bloků.

Každý blok se chová jako tlačítko, po jehož stisknutí se provede požadovaná akce. Tlačítka jsou aktivní pouze tehdy, pokud bylo navázáno spojení s robotem.

V aplikaci pracuji s 2 typy příkazů, ty, které potřebují parametr a ty, které ne. Příkazy, které nepotřebují parametr jsou ty, které robota dostávají do předdefinované pozice, například příkaz „Vstaň". Daleko obtížnější bylo získávat parametr od uživatele a předávat ho správné funkci. Tento problém jsem vyřešil přes další dialogové okno z prostředí Tkinter. Při ukládání parametru jsem narazil, na již výše popisovaný problém s kódováním a musel jsem funkcí, která zpracovávala vstup jasně říci, že data bude ukládat v kódování UTF-8.



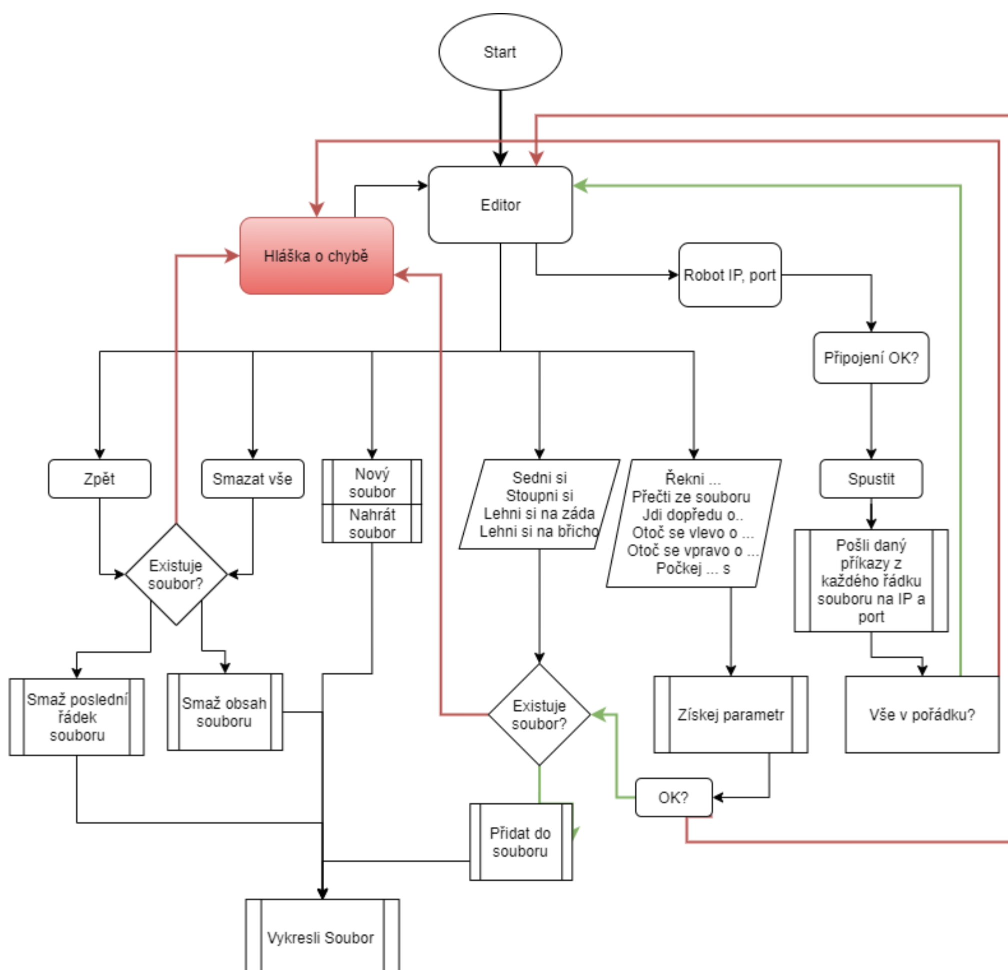
Obrázek 15 Grafická podoba části aplikace "Přímé příkazy" – nepřipojeno



Obrázek 16 Grafická podoba části aplikace "Přímé příkazy" – připojeno

5.6.3 Editor

Další část mé aplikace je editor, který dokáže tvořit, nebo nahrávat jednotlivé programy. Programy se vytváří z grafických bloků, které reprezentují jednotlivé příkazy například: řekni, sedni si apod.



Obrázek 17 Diagram Editor

Návrh, který jsem vytvořil (viz obrázek 17), byl navržen tak, aby nebyla nutnost býti připojen k robotovi při tvorbě programu. Dokud tedy aplikace nenaváže s robotem spojení, je deaktivované pouze tlačítko „Spustit“, a nelze odeslat žádná data. Můj návrh jsem tvořil s ohledem na přehlednost uživatelského prostředí a veškeré příkazy jsou ve stejné tabulce jako v předešlé části aplikace. Pokud máme nahraný soubor a na nějaký příkaz klikneme, tak místo toho, aby se daný příkaz spustil, se tento příkaz uloží do souboru. Pokaždé, když jakkoli upravuji soubor, například když ukládám další příkaz, volá se funkce, která změny vykreslí na obrazovku.

Ve schéma jsem také myslel na ošetření vstupů, jakmile při nějaké akci nastane chyba, zobrazí se chybová hláška. Chybová hláška se nezobrazuje pouze tehdy, pokud nebylo úspěšné získání parametru od uživatele. V tomto případě není potřeba nic vypisovat.

Editor obsahuje menu, ve které jsou jednotlivé položky:

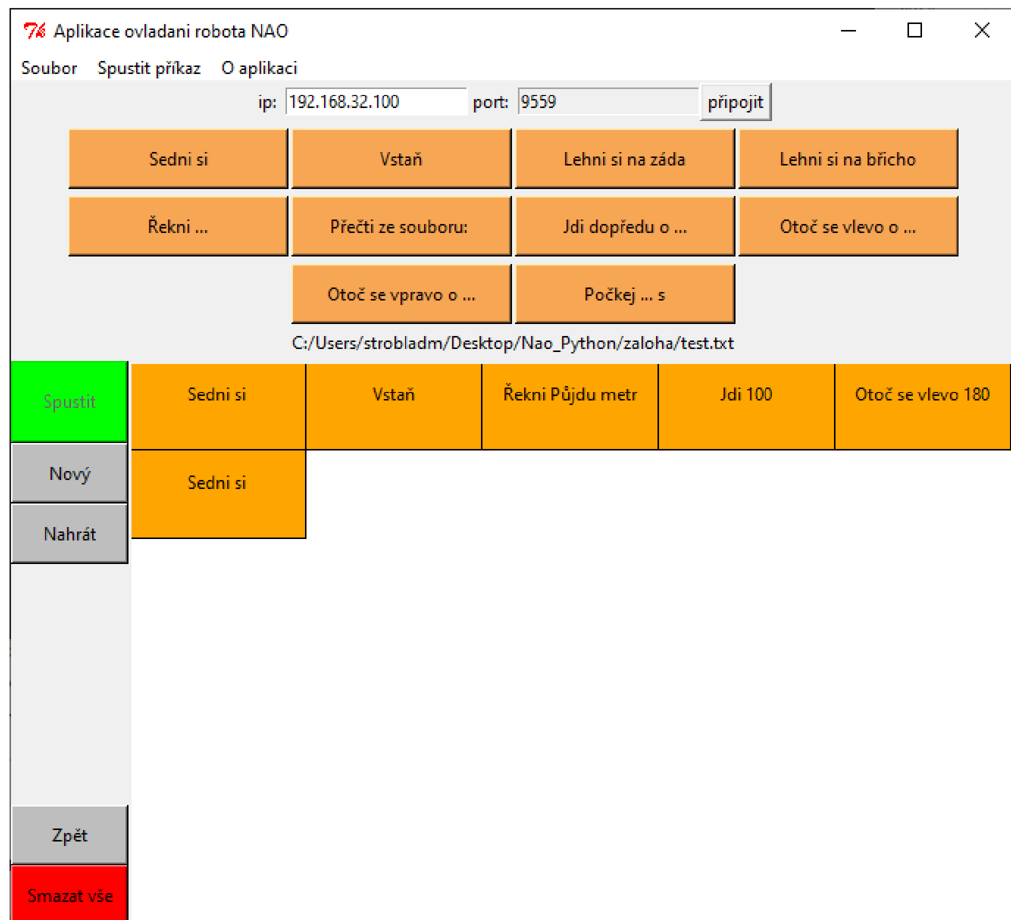
- Soubor
  - Nový – vytvoří nový soubor
  - Načíst – načte soubor
- Spustit příkaz – přesměruje do druhé části aplikace
- O aplikaci

Pod touto lištou se nachází jednotlivé bloky s příkazy. Pokud jsme již vytvořili, nebo nahráli soubor, tak po kliknutí na libovolný blok se daný příkaz zapíše do souboru a vykreslí se jako blok do zobrazovacího pole. V zobrazovacím poli jsou chronologicky zobrazeny jednotlivé příkazy daného programu. Vedle tohoto pole se nachází ovládací tlačítka, která umí smazat poslední blok, smazat všechno, načíst soubor, vytvořit nový soubor nebo daný soubor spustit.

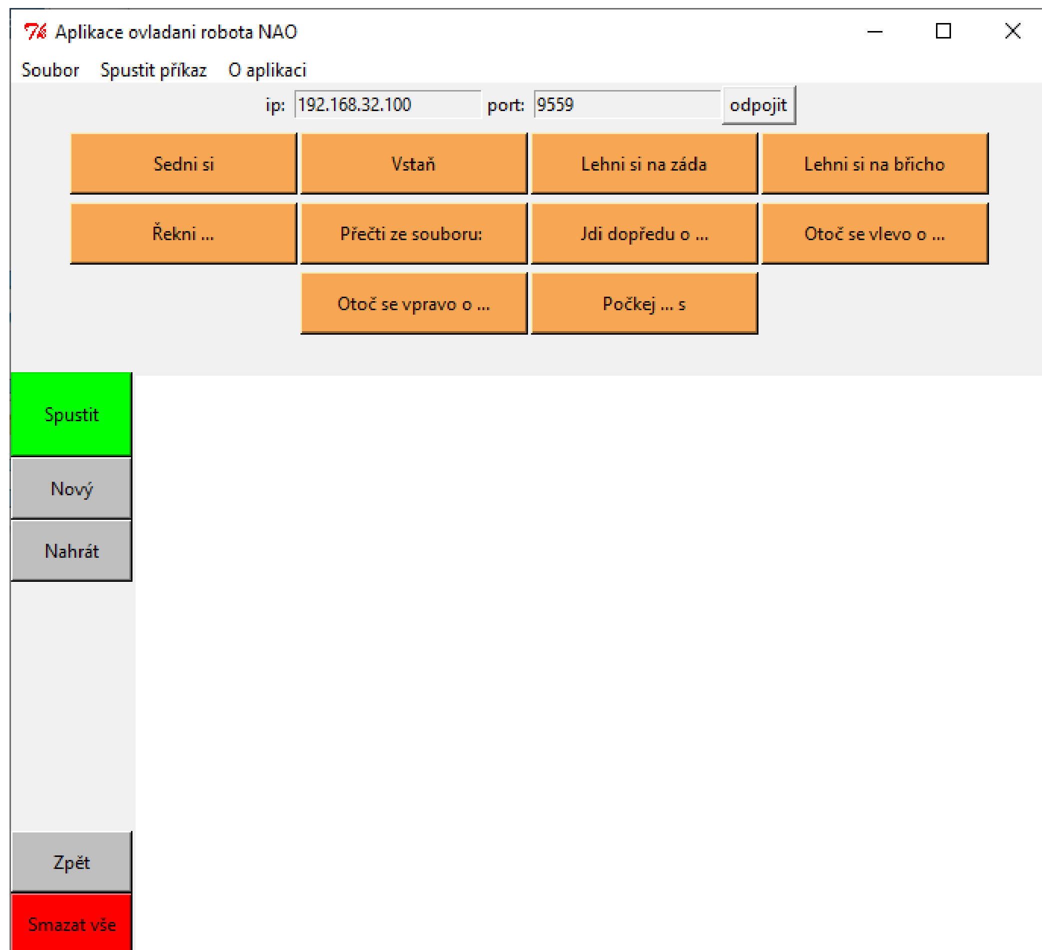
Hlavní stavební kámen při vytváření této části aplikace je práce se souborem. Do souboru se zapisují jednotlivé příkazy a také se z něj čtou. Pro jednoduchost jsem typ souboru zvolil .txt. Aplikace obsahuje dva typy příkazů, s parametrem, a bez parametru. Příkazy s parametrem musejí získat od uživatele nějaká data, zcela typicky kolik cm půjde robot dopředu, co řekne apod.

Jednotlivé příkazy se nacházejí na novém řádku a mezi příkazem a parametrem je umístěn oddělovač. Důležité je, že soubor není otevřený po celou dobu běhu aplikace. V paměti aplikace se pouze nachází cesta k danému souboru. Pokud přijde podnět k zápisu, nebo ke čtení, daný soubor se otevře, provede se požadovaná akce a zase se zavře. Toto má bezespornou výhodu při ochraně dat. Pokud by nám program spadl, nebo by byl násilně ukončen, tak nepřijdeme o žádná data. Jak jsem již zmínil, veškeré příkazy v části aplikace „Editor“, se ukládají do souboru, proto jsem také musel vyřešit ochranu vstupů, aby nebylo možné cokoli spouštět, zapisovat a smazat, pokud by nebyl vybrán žádný soubor, nebo pokud by nebyl zadán žádný parametr.

Data ze souboru čteme hned při několika funkcích. Pokaždé když přidáme, nebo odebereme příkaz, tak program pomocí parsovací funkce prohledá soubor, nalezne příkazy a jejich parametry a ty následně pomocí další funkce vykreslí do bloků v zobrazovacím poli aplikace. Při spouštění hotového programu se nejdřív do paměti načte cílová IP adresa a port, tyto informace se dále předávají funkcím ze souboru prikazy.py. Poté se načtený soubor prohledá pomocí stejné parsovací funkce, kterou jsem použil při vykreslování, ale s tím rozdílem, že pro každý řádek s nalezeným příkazem se zavolá příslušná funkce z mého souboru prikazy.py s parametry IP adresa, port, parametr (pokud existuje).



Obrázek 18 Grafická podoba části aplikace "Editor" – odpojeno



Obrázek 19 Grafická podoba části aplikace "Editor" – připojeno

5.6.4 Grafická podoba

Jak jsem již v minulých kapitolách zmínil, tak veškeré grafické prostředí jsem tvořil pomocí knihovny Tkinter. Knihovna Tkinter má v sobě několik widgetů, ze kterých jsem následně vytvořil celou aplikaci.

Nejvíce jsem používal tyto widgety:

- Frame – vytvoří rámeček
- Label – vytváří textový blok určený pro čtení
- Entry – vytváří textové vstupní pole, určené k zápisu
- Button – vytváří tlačítko

Každý tento widget má definované vlastnosti například: výšku, šířku, barvu pozadí a speciální vlastnosti, které interagují s uživatelem. Pokud vytvoříme nějaký widget, automaticky to neznamená, že se zobrazí. K umístění těchto widgetů musíme zavolat nějakou zobrazovací metodu. Na výběr jsou dvě hlavní metody. První metoda je metoda `pack()`. Pomocí této metody vytváříme layouty „balením" widgetů do rodičovských widgetů tím, že jsou považovány za čtvercové oblasti umisťované do rámečků. Druhou metodou je metoda `grid()`. Metoda `grid()` nám umožňuje vytvářet layouty podobné tabulkám, umisťováním widgetů do dvourozměrné mřížky. Ve své aplikaci používám kombinaci obou těchto metod. Metodu `grid()` jsem používal na tabulku příkazů, metodu `pack()` poté na nabídku ovládacích tlačítek po levé straně zobrazovacího pole aplikace.

6 Dílčí problémy řešené při vývoji aplikace

6.1 Významné programové konstrukce

V této kapitole se budu zabývat důležitými funkcemi, které jsem vytvořil. Nebudu zde popisovat všechny, ale pouze ty, které mi přijdou zajímavé.

6.1.1 Import knihoven

Nejdříve bych začal hlavičkou hlavního souboru a modulu prikazy.py, zde importuji veškeré knihovny, které používám. Importuji knihovny, Tkinter, naoqi, sys, os, os.path, tkMessageBox, ScrolledText, tkFileDialog, subprocess, math a pár dalších. Také do hlavního souboru importuji pomocný modul prikazy.py.

```
# -*- coding: utf-8 -*-
from Tkinter import *
from naoqi import ALProxy
import prikazy
import sys,os
from os import path
import tkMessageBox
import Tkinter, ScrolledText
import tkFileDialog
#import subprocess – již nepotřebný
import time
```

Zdrojový kód 4 Import knihoven v hlavním souboru

```
# -*- coding: utf-8 -*-
import sys
import argparse
from Tkinter import *
import tkMessageBox
from naoqi import ALProxy
import math
```

Zdrojový kód 5 Import knihoven v modulu prikazy.py

6.1.2 Kombinování funkcí

Důležitá funkce, kterou jsem často používal, byla funkce `combine_funcs`. Tato funkce bere jako parametr několik různých funkcí a všechny je zavolá. Takto jsem mohl z jednoho tlačítka zároveň zavolat funkci na přidání příkazu do souboru a zároveň funkci, kterou jsem zavřel dialogové okno.

To za normálních okolností nelze, proto mi tato funkce velice usnadnila práci. Pokud bych tuto funkci nepoužíval, musel bych pro každé tlačítko vytvořit funkcí, která by následně volala ostatní funkce. Tuto funkci jsem nevytvořil, ale našel jsem ji na pomocném webu pro vývojáře stackoverflow [14].

```
def combine_funcs(*funcs):
  def combined_func(*args,**kwargs):
    for f in funcs:
      f(*args, **kwargs)
  return combined_func
```

Zdrojový kód 6 Funkce combine_funcs()

6.1.3 Získání a zpracování vstupů od uživatele

Nejdůležitější funkce, které jsem vytvořil, byli z mého pohledu funkce `getparametr()` a `sendto()`. Tyto dvě funkce zpracovávají parametry, které zadává uživatel pro určité příkazy. Funkce `getparametr()` dostává kromě informací o IP adrese a portu parametr, který označuje, ke kterému příkazu se data od uživatele vztahují.

Nejdříve jsem si definoval nové okno a nazval jsem ho „w“. Název tohoto okna je „Zadejte parametr:“, obsahuje textové pole, do kterého uživatel zadá příslušná data a tlačítko, které tyto data předá funkci `sendto()`.

```
  def getparametr(ip, port, cislo):
    w=Toplevel(root_R)
    wlabel=Label(w, text="Zadejte parametr:")
    wentry= Entry(w)
    wbutton= Button(w, text="Ok", bg="green", command= com-
bine_funcs(lambda: sendto(ip,port,cislo, wentry.get().en-
code('utf-8')), lambda: w.destroy()))
    wlabel.pack()
    wentry.pack()
    wentry.focus_set()
    wbutton.pack()
```

Zdrojový kód 7 Funkce getparametr()

Funkce, které se data od uživatele předávají, se jmenuje `sendto`. Tuto funkci používám v obou částech aplikace s tím rozdílem, že v jedné části příslušná data ukládám do souboru a v druhé přímo volám dané funkce z modulu prikazy.py. V ukázkách zdrojových kódů se jedná o funkce z modulu prikazy.py, které přímo odesílají robotovi instrukce. Návratové hodnoty se následně zpracovávají funkcí `stav()`.

```
def sendto(ip, port, cislo, parametr):
  vysledek=0
  port=int(port)
  cislo=int(cislo)
  if cislo==0:
    vysledek=prikazy.mluv(ip,port,parametr)
  elif cislo==1:
    vysledek=prikazy.cti(ip,port,parametr)
  elif cislo==2:
    vysledek=prikazy.jdi(ip,port,parametr)
  elif cislo==3:
    vysledek=prikazy.otoc(ip,port,parametr)
  else:
    parametr=float(parametr) * -1
    vysledek=prikazy.otoc(ip,port,parametr)
  stav(vysledek)
```

Zdrojový kód 8 Funkce sendto()

6.1.4 Reagování na stav aplikace

Tato funkce zjišťuje, zdali je aplikace připojená k robotovi, či nikoli. Pokud je aplikace připojena, tak tato funkce aktivuje ovládací prvky. Při odpojení nebo pokud nastane jakákoli chyba při odesílání příkazů, se tyto ovládací prvky deaktivují. Ovládacími prvky se v části „Editor“ myslí tlačítko „Spustit“. V druhé části se deaktivují veškerá tlačítka volající jednotlivé funkce.

Funkce jako svůj parametr dostává hodnoty 0, nebo 1,2. Tyto hodnoty získává z návratových hodnot jednotlivých funkcí. V závislosti na této hodnotě dává volaná funkce najevo, zdali skončila úspěšně 0, nebo s chybou. Pokud tato návratová hodnota nabývá hodnotu 1, poté volaná funkce skončila chybou kvůli nenavázanému spojení s robotem. Pokud je hodnota 2, nastala jiná chyba.

```
def stav(x)
  global pripojeni
  pripojeni=x
  if x==0:
    button0["state"]="normal"
    button1["state"]="normal"
    ...
    Button9["state"]="normal"
    ipentry["state"]="read-only"
    test_connect.grid_forget()
    test_disconnect.grid(row=0,column=4)
  else:
    button0["state"]="disabled"
    button1["state"]="disabled"
    ...
    Button9["state"]="disabled"
    ipentry["state"]="normal"
    test_connect.grid(row=0,column=4)
    test_disconnect.grid_forget()
```

Zdrojový kód 9 Funkce stav()

6.1.5 Vykreslování zadaných příkazů

Další zajímavou funkcí, kterou jsem vytvořil, je funkce, která vykresluje bloky příkazů, které jsou uloženy v souboru. Tato funkce pracuje s knihovnou Tkinter, konkrétně s widgetem Canvas. Tuto funkci jsem nazval `update`, protože ji pokaždé volám, když se změní soubor, do kterého ukládám.

Funkce nejdříve na svém plátnu všechno smaže, poté otevře soubor, kam se ukládají data. Funkce nastaví počáteční pozici X(x1, y1) a vzdálenosti xdistance a ydistance, tyto hodnoty udávají, jakou mají vytvořené bloky velikost. Proměnné maxx a maxy udávají maximální výšku a šířku plátna. Podle hodnot v proměnných můžeme snadno zjistit, že maximální počet bloků je 5 na šířku a 6 na výšku. Dohromady tedy program dokáže vykreslit 30 po sobě jdoucích příkazů. Soubor poté funkce nahrává do paměti, a prochází ho parsovacím cyklem. Pro každý řádek funkce vykresluje blok daného příkazu, který řádně umisťuje na plátno widgetu Canvas. Text tohoto bloku se rovná příkazu a jeho parametru zjištěnému na daném řádku. Po skončení procházení souboru, se soubor zavře a odstraní z paměti.

```
def update():
    programvstup.delete("all")
    file1=open(filepath,'r')
    x1=0
    y1=0
    xdistance=120
    ydistance=60
    maxx=600
    maxy=360
    for line in file1:
        if "Čti" in line:
            k=line.replace('#', '')
            x=k.split('/', -1)
            line="Čti z \n" + x[-1]
        else:
            line=line.replace('#', '')
        if x1<maxx:
            x2=x1+xdistance
            y2=y1+ydistance
        else:
            x1=0
            x2=x1+xdistance
            y1=y1+ydistance
            y2=y1+ydistance
        programvstup.create_rectan-
gle(x1,y1,x2,y2,fill="#FFA500")
        programvstup.create_text((x2-60),(y2-30), text=line)
        x1=x2
    file1.close()
```

Zdrojový kód 10 Funkce update()

6.2 Vytvoření souboru .exe

Vytvoření spustitelného souboru byl náročný úkol. Největší problém byl s verzí pythonu. Našel jsem celkem 2 způsoby, jak vytvořit spustitelný soubor ze scriptu Python. První způsob, na který jsem narazil, byl projekt auto-py-to-exe. Tento projekt přestal podporovat jazyk Python ve verzi 2.7. Další způsob, který jsem vyzkoušel byl pyinstaller. Tento balíček pracuje s příkazovým řádkem a nemá žádné GUI. Tento balíček jsem instaloval přes příkazový řádek, konktrétně skrze správce balíčků pro Python pip. Pip se automaticky nainstaloval při instalaci pythonu, ale bohužel byl ve špatné verzi. Pip jsem přeinstaloval na verzi 20.3.4, která podporuje jazyk Python ve verzi 2.7. Balíček pyinstaller jsem poté nainstalovat prostřednictvím příkazu `pip install pyinstaller`.

Bohužel k mé smůle jsem i přes správnou verzi programu pip nainstaloval nejnovější verzi daného balíčku. Po dlouhém hledání jsem, našel verzi 2.1, která podporuje tvorbu .exe ze skriptů pythonu 2.7. Číslo této verze jsem přidal jako podmínku při instalaci balíčku pyinstaller. Příkaz na získání správné verze tedy vypadá takto: `pip install pyinstaller==2.1`.

Po nainstalování balíčku pyinstaller ve verzi 2.1 jsem se pokoušel pomocí skriptu `pyinstaller --onefile program.py` vytvořit spustitelný soubor. Všechno vypadalo, že funguje do doby, než se skript pokusil nahrát knihovnu naoqi z SDK. K této knihovně skript nenašel cestu. Následně jsem zjistil, že proces vytváření spustitelného souboru nebere v potaz systémové proměnné, mezi kterými se nachází moje proměnná PYTHONPATH, která odkazuje na umístění knihovny naoqi. K vyřešení tohoto problému jsem musel nastavit cestu PATH, aby vzala v potaz systémové proměnné.

Finální sekvence příkazů v příkazovém řádku pro vytvoření spustitelného souboru .exe by tedy byla následující:

- pip --version
    - pokud nemáme verzi, která nepodporuje Python 2.7 musíme nainstalovat jinou verzi
- pip install pyinstaller==2.1
- set PATH=%PATH%;C:\Windows\System32\downlevel
- pyinstaller --windowed --onefile aplikace.py

7 Testování aplikace

Dosavadní testování mé aplikace mi pomohlo odstranit pár technických chyb. Chyby převážně nebyli spjaté s ovládáním robota, ale spíše s logikou aplikace, a proto jsem musel řádně ošetřit vstupy. Jediný problém, který jsem si vytvořil sám, bylo nesprávné použití metody `post()`, která vytvářela paralelní proces (viz kapitola 4.2.4) a nedosahovala požadovaného výsledku. Aplikace se nyní nachází ve stavu, kdy v editoru již není možné zadávat příkaz, pokud není otevřen nějaký soubor. Z obou částí aplikace není možné posílat jakýkoli příkaz, pokud nebylo předtím otestované připojení k robotovi. Pokud zadáme prázdný parametr funkci s parametrem, nebo zavřeme dialogové okno pro získání parametru, tak se tento zbytečný příkaz neprovede, ani neuloží. Pokud nezvolíme nový soubor, který se přes dialogové okno nahraje, zůstane otevřený ten původní. Podobně jsem odstranil i další menší chyby aplikace. Ve své aplikaci, která je nyní ve své finální verzi, jsem nenašel žádné další logické problémy.

7.1 Porovnání s aplikací Choreographe

Mnou vytvořenou aplikaci jsem následně testoval a porovnával s oficiální aplikací Choreographe. Při testování jsem se snažil vytvořit stejné podmínky pro obě aplikace. Obě dvě aplikace jsem testoval až v době, kdy obě byli připravené k práci. To znamená, že jsem netestoval rychlost načítání aplikace do paměti počítače, ani dobu trvání, než aplikace navázala spojení s robotem. Aplikace jsem testoval na základě dvou kritérií. První kritérium se zabývalo spotřebovaným časem, který zkušený operátor potřebuje na to, aby splnil jednotlivé fáze testování. Druhé kritérium testovalo rychlost robotovy odpovědi po odeslání příkazu.

7.1.1 Testování časové náročnosti

Časové údaje získané při tomto testování je potřeba brát s rezervou. Moje aplikace není tak hardwarově náročná jako Choreographe, proto mohlo dojít ke zkreslení časových údajů, kvůli výkonu zařízení, na kterém jsem testování prováděl.

Aplikace jsem podle prvního kritéria testoval ve třech fázích. První fází bylo vytvoření a spuštění jednoduchých příkazů. Příkazy jsem vybral následující: „Postav se“, „Řekni ‘Ahoj’ “. Zjistil jsem, že průměrná doba vytvoření daného programu s jednoduchým příkazem v aplikaci Choreographe trvá průměrně minutu a půl.

V mé aplikaci vytvoření stejného programu v editoru trvalo zhruba 20 sekund. Pokud bych daný příkaz spouštěl pomocí spuštění příkazu, příkaz by se provedl skoro okamžitě (více v následující kapitole).

Druhá fáze byla tvorba složitější sekvence příkazů. Robot se nejdříve posadí, následně vstane, řekne „Půjdu metr dopředu“, půjde dopředu o jeden metr, udělá čelem vzad a posadí se. Vytvoření takového programu v aplikaci Choreographe mi trvalo zhruba 3 minuty a 40 sekund. Díky tomu, že v mé aplikaci jsou tyto příkazy přímo připravené, nebyla potřeba je v rozsáhlých seznamech hledat, a proto tvorba tohoto programu v mé aplikaci trvala 36 sekund.

Třetí fází testování podle prvního kritéria byla editace již vytvořeného programu z druhé fáze, konkrétně úprava posledních dvou příkazů. Poslední dva příkazy udávali robotovi, aby se otočil čelem vzad a poté se posadil. Tyto instrukce byli vyměněny za příkazy, po kterých robot řekne „To jsem se unavil“ a lehne si na břicho. V aplikaci Choreographe tato změna trvala necelé dvě minuty, přesně 1minutu a 58 sekund. V mé aplikaci jsem tuto úpravu byl schopen udělat za 40 sekund.

7.1.2 Testování rychlosti odpovědi

Při testování podle druhého kritéria jsem u aplikací testoval rychlost jejich odpovědi. Konkrétně jsem zjišťoval, za jak dlouhou dobu je robot schopen vykonat požadovanou činnost po odeslání příkazu. Toto testování jsem prováděl pomocí předpřipraveného programu, kde byla jednoduchá instrukce: „Řekni Ahoj já jsem Nao“. Čas jsem stopoval od doby, kdy jsem tento program spustil a zastavil jsem ho poté, co robot začal mluvit. Toto testování jsem prováděl celkem 10x. U tohoto testování jsem pozoroval největší rozdíl mezi mojí aplikací a aplikací Choreographe. Zatímco u aplikace Choreographe byla průměrná doba odpovědi 4,2 sekundy, u mojí aplikace jsem nebyl schopen efektivně měřit, protože naměřené časové údaje byli příliš malé. Doba odezvy mé aplikace byla menší než jedna sekunda.

7.1.3 Limity mé aplikace

Moje aplikace má oproti oficiálnímu programu Choreographe několik omezení. Aplikace má maximální počet zobrazených příkazů v jednom programu (30). Příkaz řekni, nedokáže v editoru zobrazit delší text, proto je lepší používat příkaz čti z externího souboru. Uživatel musí vědět co dělá, pokud robotovi zadá nějaký příkaz, nelze robota nijak zastavit, než dokončí zadaný příkaz.

7.2 Hodnocení aplikace

Pokud nebudu brát v potaz limity mojí aplikace, tak rozdíl mezi mojí aplikací a oficiální aplikací Choreographe, byl nejvíce patrný při mém testování rychlosti odpovědi. Zatímco po zadání příkazu pomocí mojí aplikace trvalo robotovi provedení příkazu necelou sekundu, oficiální aplikaci Choreographe to trvalo déle než 4 sekundy. Tento rozdíl je daný principem, jak aplikace spouští programy. Zatímco moje aplikace zpracovává program na počítači a robotovi posílá až určité příkazy, v programu Choreographe se takto vytvořený program nahrává do robotovi paměti a až odtud se spouští. Proto tedy v programu Choreographe doba nahrávání závisí na velikosti programu. Kromě rychlosti jsem mezi aplikacemi nepostřehl žádný jiný rozdíl a obě vykonávali svoji činnost stejně plynule.

8 Závěr

Současná desktopová aplikace je plně funkční a podle mého testování splnila hlavní cíle práce. Moje aplikace dokáže rychle a efektivně ovládat robota v přímém čase a pro jednoduché úkony je jednodušší, rychlejší a přehlednější než oficiální aplikace Choregraphe. Moje aplikace se svými vlastnostmi skvěle hodí jako doplněk ke komplexnějšímu programu, který by běžel na robotovi a pouze by doplňoval jeho případné mezery. Choreographe má bezespornou výhodu svým komplexním zpracováním, ale hodí se spíše pro tvorbu složitých autonomních programů, jako je například vedení konverzace, procvičování učiva, nebo procházení labyrintem. Bohužel, kvůli epidemiologické situaci (Covid-19) jsem nemohl aplikaci řádně otestovat při popularizační akci, jak bylo stanoveno v zadání práce.

Aplikace používá již starý framework naoqi a nepodporovaný jazyk Python verze 2.7. Dokud bude robot podporovat framework naoqi, tak volba programovacího jazyka na funkčnosti aplikace nebude mít žádný vliv. Další možné pokračování mé práce by se nabízela tvorba aplikace pro Android. Nedávno jsem četl na oficiálních stránkách výrobce, že vytváří nové SDK pro Android pro robota Pepper. Pokud by dané SDK vzniklo i pro vývoj aplikace pro robota NAO, bylo by možné vytvořit aplikaci podobné mojí, pro operační systém Android bez větších obtíží. Pokud by dané SDK nebylo podporované pro robota NAO, nabízela by se další možnost, jak vytvořit aplikaci pro Android. V této práci by mnou již vytvořená aplikace pro Windows měla funkci serveru, kam by se aplikace pro Android připojila a moje upravená aplikace by pouze interpretovala přijaté příkazy a přeposílala je robotovi.

9 Seznam obrázků

Obrázek 1 Robot Nao [1] ... 12
Obrázek 2 Umístění portů [2] ... 13
Obrázek 3 Umístění mikrofonů a reproduktorů [2] ... 14
Obrázek 4 Umístění kamer [2] ... 14
Obrázek 5 Stejné API v C++ a v Pythonu [4] ... 17
Obrázek 6 Soubor Autoload.ini [4] ... 18
Obrázek 7 Broker – Moduly – Metody ... 19
Obrázek 8 Příklad blokovaného volání metody [4] ... 21
Obrázek 9 Příklad neblokovaného volání metody přes objekt "post" [4] ... 22
Obrázek 10 Program Choreographe [7] ... 23
Obrázek 11 Podporované programovací jazyky [3] ... 24
Obrázek 12 Zobrazení Os [10] ... 28
Obrázek 13 Předdefinované polohy [11] ... 29
Obrázek 14 Diagram spustit příkaz ... 34
Obrázek 15 Grafická podoba části aplikace "Přímé příkazy“ – připojeno ... 35
Obrázek 16 Grafická podoba části aplikace "Přímé příkazy“ – nepřipojeno ... 35
Obrázek 17 Diagram Editor ... 36
Obrázek 18 Grafická podoba části aplikace "Editor" – odpojeno ... 38
Obrázek 19 Grafická podoba části aplikace "Editor" – připojeno ... 39

10 Seznam zdrojových kódů

Zdrojový kód 1 Funkce pozice() 31
Zdrojový kód 2 Funkce mluv() 32
Zdrojový kód 3 Funkce otoc() 33
Zdrojový kód 4 Import v hlavním souboru 41
Zdrojový kód 5 Import v modulu prikazy.py 41
Zdrojový kód 6 Funkce combine_funcs() 42
Zdrojový kód 7 Funkce getparametr() 42
Zdrojový kód 8 Funkce sendto() 43
Zdrojový kód 9 Funkce stav() 44
Zdrojový kód 10 Funkce update() 45

11 Zdroje

[1] *NAO Documentation* [online]. Paris: SoftBank Robotics Europe, 2019 [cit. 27.8.2020]. Dostupné z: https://developer.softbankrobotics.com/nao6/nao-documentation/nao-developer-guide

[2] *NAO Documentation* [online]. Paris: SoftBank Robotics Europe, 2016 [cit. 3.3.2020]. Dostupné z: http://doc.aldebaran.com/2-8/home_nao.html

[3] *NAOqi developer guide* [online]. Paris: SoftBank Robotics Europe, 2019 [cit. 18.8.2020]. Dostupné z: https://developer.softbankrobotics.com/nao6/naoqi-developer-guide

[4] *NAO (NAOqi 2.1)* [online]. Paris: Softbank Robotics Europe, 2019 [cit. 8.4.2021]. Dostupné z: https://developer.softbankrobotics.com/nao-naoqi-2-1/naoqi-developer-guide/naoqi-framework/key-concepts

[6] *Naoqi APIs* [online]. Paris: SoftBank Robotics Europe, 2017 [cit. 26.8.2020]. Dostupné z: http://doc.aldebaran.com/2-8/naoqi/index.html

[7] *NAO V6 Documentation* [online]. Paris: SoftBank Robotics Europe, 2017 [cit. 27.8.2020]. Dostupné z: https://developer.softbankrobotics.com/nao6/naoqi-developer-guide/sdks

[8] SANDOVAL, Kristopher. *What is the difference between an API and an SDK* [online]. Stockholm: Nordic APIs AB, 2016 [cit. 2021-04-09]. Dostupné z: https://nordicapis.com/what-is-the-difference-between-an-api-and-an-sdk/

[9] BROWN, Roslyn, Heather L. HELTON, Allison C. WILLIAMS, Michael T. SHROVE, Mladen MILOŠEVIĆ, Emil JOVANOV, David COE a Jeff KULICK. Android Control Application for Nao Humanoid Robot. In: *The 2013 International Conference on Frontiers in Education: Computer Science and Computer Engineering* [online]. Las Vegas, Nevada, USA, 2013.[cit. 2021-04-10]. Dostupné z: http://worldcomp-proceedings.com/proc/p2013/FEC4165.pdf

[10] *Naoqi Python 2.7 SDK* [online]. Paris: Softbank Robotics Europe, [cit. 11.4.2021]. Dostupné z: https://www.softbankrobotics.com/emea/en/support/nao-6/downloads-softwares

[11] *NAO Software Motion API documentation* [online]. Paris: SoftBank Robotics Europe, [cit. 12.4.2020]. Dostupné z: http://doc.aldebaran.com/1-14/naoqi/motion/index.html

[12] *NAO Software Audio API documentation* [online]. Paris: SoftBank Robotics Europe, [cit. 12.4.2020]. Dostupné z: http://doc.aldebaran.com/1-14/naoqi/audio/index.html

[13] *Sunsetting Python 2* [online]. USA: Python Software Foundation, 2014 [cit. 2021-04-17]. Dostupné z: https://www.Python.org/doc/sunset-Python-2/

[14] Have multiple commands when button is pressed. In: *Stackoverflow* [online]. USA: Stackoverflow, 2012 [cit. 2021-04-18]. Dostupné z: https://stackoverflow.com/questions/13865009/have-multiple-commands-when-button-is-pressed/13865150#13865150