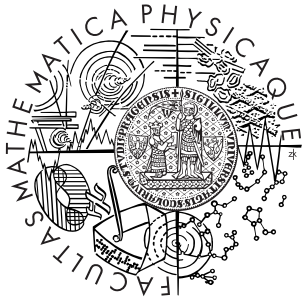
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jakub Marek

Lokalizace robota ve venkovním prostředí

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. David Obdržálek
Studijní program: Informatika, obecná informatika

2009

Děkuji panu RNDr. Davidu Obdržálkovi za vedení této práce, za trpělivost a cenné rady. Také děkuji své rodině za pomoc a podporu.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 5. 8. 2009 Jakub Marek

Obsah

Název práce: Lokalizace robota ve venkovním prostředí
Autor: Jakub Marek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. David Obdržálek
e-mail vedoucího: David.Obdrzalek@mff.cuni.cz

Abstrakt: V této práci je popsán návrh a implementace frameworku pro tvorbu řídících systémů pro mobilní roboty. Jde o jednoduchý modulární systém usnadňující tvorbu řídících systémů a pokusy s jejich jednotlivými moduly. Součástí frameworku jsou také některé nástroje pro lokalizaci ve venkovním prostředí.
Klíčová slova: lokalizace, robotika, mobilní robotika, venkovní robot, framework

Title: Outdoor robot localization
Author: Jakub Marek
Department: Department of Software Engineering
Supervisor: RNDr. David Obdržálek
Supervisor's e-mail address: David.Obdrzalek@mff.cuni.cz

Abstract:The work describes the design and implementation of a framework for building control systems for mobile robots. It is a simple modular system simplifying creation of control systems and experiments with the modules. The framework contains also several tools for localization in outdoor environment.
Keywords: localization, robotics, mobile robotics, outdoor robot, framework

Kapitola 1

Úvod

Při vytváření softwaru pro autonomního robota je zapotřebí řešit mnoho různých problémů. Většina z těchto problémů, jako je například lokalizace, nebo komunikace se senzory, je ale u různých robotů podobná, bez ohledu na konkrétní hardware.

Cílem této práce bylo vytvořit systém pro lokalizaci a řízení autonomního robota ve venkovním prostředí, tj. zpracování dat ze senzorů, jejich vyhodnocování a využití pro řízení robota. Jako součást této práce vznikl framework RCT (Robot Controling Thingie), který tyto problémy řeší a usnaňuje tvorbu řídícího systému pro mobilní roboty.

V následujícím textu se věnuji návrhu a implementaci tohoto frameworku. Druhá kapitola popisuje technologie a algoritmy, použité v této práci. Třetí kapitola obsahuje analýzu problému. Ve čtvrté kapitole je popsaný návrh RCT a pátá kapitola obsahuje popis jeho implementace.

Kapitola 2

Použité technologie

Tato kapitola obsahuje stručný přehled technologií použitých v této práci.

2.1 Lokalizace

Lokalizace mobilního robota je proces, při kterém se zjišťuje pozice robota v prostředí. Vstupem pro lokalizaci jsou data naměřená senzory. Protože ale senzory často neposkytují dostatečně přesná měření, jsou tato data zatížena chybami ([3]).

Problém lokalizace je možné rozdělit na udržování odhadu pozice, kdy je k dispozici počáteční odhad a cílem je pouze tento odhad upravovat a korigovat chyby odometrie, a globální lokalizaci, při které musí lokalizační algoritmus počáteční pozici nejprve zjistit. Kromě těchto dvou základních problémů existuje velké množství dalších variant lokalizace, například aktivní lokalizace, kdy lokalizační algoritmus může přímo ovlivňovat řízení robota pro dosažení lepší přesnosti. Mezi další problémy souvisejících s lokalizací patří například také řešení takzvaného problému uneseného robota Při řešení tohoto problému se robot, který je již lokalizovaný a pouze udržuje odhad pozice, přesune bez upozornění na neznámé místo ([2]).

2.1.1 Bayesovské filtry

Bayesovské filtry slouží k pravděpodobnostnímu odhadu stavu systému na základě nepřesných pozorování. Stav – u lokalizace je to pozice, orientace atd. – je reprezentován jako náhodná proměnná a rozdělení pravděpodobnosti popisuje nejistotu odhadu. Tento odhad se pak aktualizuje pokaždé, když se robot pohybuje, nebo jsou dostupná nová data ze senzorů.

Aby bylo možné tyto úpravy efektivně vypočítat, předpokládá se, že se jedná o Markovův systém, tedy že v každém okamžiku měření ze senzorů závisí pouze na současném stavu robota a že předchozí stavy neposkytují žádnou

další informaci. S tímto předpokladem je možné předchozí měření zapomínat a upravovat stav pouze na základě posledních informací ([3]).

Bayesovské filtry pracují ve dvou fázích. V první fázi – predikční – se odhad upraví na základě pohybového modelu robota, předpoví se nový stav ze stavu předchozího. Následně se v korekční fázi tento odhad upraví podle informací ze senzorů ([3, 4]).

Existuje velké množství variant Bayesovských filtrů, které se liší reprezentací rozdělení pravděpodobnosti odhadu pozice.

2.1.2 Monte Carlo lokalizace

Jednou z variant Bayesovských filtrů jsou částicové filtry, mezi které patří i Monte Carlo lokalizace (MCL). Tento druh lokalizačního algoritmu reprezentuje rozdělení pravděpodobnosti odhadu pomocí množiny vzorků náhodně vybraných z tohoto rozdělení. Každý vzorek odpovídá jedné pozici ze stavového prostoru robota a má přiřazenou váhu, která určuje pravděpodobnost dané pozice.

Algoritmus MCL probíhá v iteracích, každá iterace se skládá z následujících čtyř fází ([6]):

1. *Predikce*: Vzorky se přesunou podle údajů z odometrie a podle modelu chyby těchto údajů.
2. *Korekce*: V následujícím kroku se změní váhy vzorků na základě měření z ostatních senzorů.
3. *Normalizace*: Váhy vzorků se normalizují tak, aby jejich součet byl 1.
4. *Převzorkování*: Viz níže.

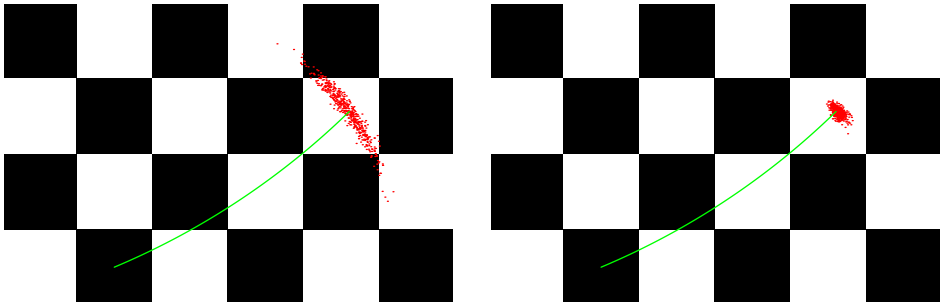
MCL je schopná globální lokalizace (v počátečním odhadu jsou vzorky rovnoměrně rozložené v celém stavovém prostoru a mají stejné váhy) a je poměrně odolná proti chybám při lokalizaci. Mezi další výhody tohoto algoritmu patří také jednoduchost implementace. Monte Carlo lokalizace může také podle potřeby adaptivně měnit velikost množiny vzorků a tím šetřit výpočetní výkon v době kdy není potřeba plný počet vzorků ([4]).

Na obrázku 2.1 jsou dvě ukázky sledování pozice na šachovnici za pomoci tohoto algoritmu. První ukázka využívá pouze odometrie a je na ní vidět reprezentaci odhadu pozice množinou vzorků. Ve druhé ukázce se robot lokalizuje i podle barvy podlahy a odhad je výrazně přesnější.

Více detailů o MCL je v [3, 4, 6].

Modely senzorů

Aby bylo možné upravovat pozice vzorků v predikční fázi, nebo váhy vzorků podle měření senzorů v korekční fázi, je nutné znát model pohybu robota, a model dostupných senzorů.



Obrázek 2.1: Monte Carlo lokalizace na šachovnici

Převzorkování

Částicové filtry obecně by bez použití převzorkování trpěly degenerovaností množiny vzorků, tedy že malé množství vzorků by mělo velkou váhu a váha většiny ostatních vzorků by byla blízká k nule. Převzorkování tento problém řeší tak, že z rozložení pravděpodobnosti reprezentovaného vzorky vygeneruje novou množinu vzorků, která má u všech vzorků stejné váhy. Toho se dosáhne tím, že na místě původních vzorků s vysokou váhou vytvoří větší množství vzorků a některé vzorky s nízkou váhou odstraní.

Problémem převzorkování ale je, že množina vzorků po převzorkování neodpovídá přesně stejnému rozložení pravděpodobnosti jako množina vzorků před převzorkováním, a tedy se některé vzorky ztrácí. Protože ale převzorkování slouží pouze pro odstranění degenerovanosti množiny odhadů, nemusí se provádět v každém kroku. Místo toho se počítá efektivní počet vzorků a převzorkuje se pouze pokud tento počet klesne pod zadanou hranici ([6]).

Pro převzorkování slouží například algoritmus systematického převzorkování (algoritmus 2 z [6]).

2.2 Senzory

2.2.1 Odometrie

Lokalizace je využívání informací z motorů, nebo enkodérů robota pro lokalizaci. Odometrie může sloužit jako model pohybu v predikční části bayesovské lokalizace ([3]), ale může být využívána i jinak, například dead reconing ([1]) využívá odometrii jako jediný zdroj lokalizačních informací.

V této práci se zabývám pouze odometrií robota s diferenciálním pohonem.

Diferenciální pohon

Diferenciální pohon má dvě nezávisle poháněná kola a rozdíl jejich rychlostí určuje rychlost zatáčení.

Model chyby diferenciálního pohonu podle [17] a [18] předpokládá, že vzdálenost skutečně ujetá kolem na nějakém úseku se řídí normálním rozdělením $N(s + k_1 \cdot s, k_2 \cdot s)$, kde s je vzdálenost naměřená senzory, k_1 je konstanta korigující systematickou chybu měření a k_2 je konstanta popisující varianci (nesystematickou chybu). Dále se předpokládá, že vzdálenosti ujeté levým a pravým kolem jsou na sobě nezávislé.

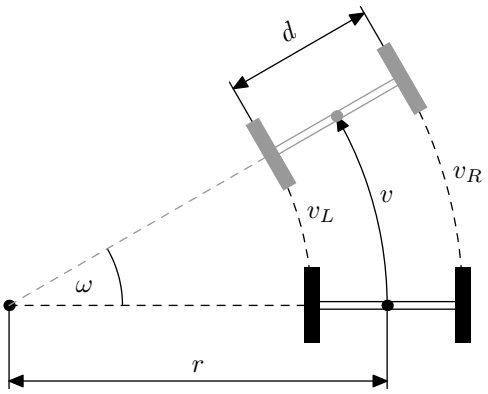
Rovnice pro řízení diferenciálního pohonu Pro řízení robota není vhodné dávat příkazy přímo ve tvaru požadovaných rychlostí pro kola. Vhodnější je tyto příkazy vytvářet ve tvaru požadované dopředné rychlosti a požadované rychlosti zatáčení. Pro převádění mezi těmito reprezentacemi slouží následující rovnice:

$$v_L = v - \frac{\omega \cdot d}{2} \qquad v_R = v + \frac{\omega \cdot d}{2}$$

$$v = \frac{v_L + v_R}{2} \qquad \omega = \frac{v_R - v_L}{d}$$

v_L a v_R jsou rychlosti levého, respektive pravého kola, v je celková dopředná rychlost robota, ω je úhlová rychlost zatáčení a d je rozchod kol (viz obrázek 2.2).

Pro úpravu vzorků se používají rovnice 1.3 - 1.7 z [1], které zjednodušují pohyb na otočení a poté přímý pohyb.



Obrázek 2.2: Diferenciální pohon při projetí kruhového úseku

2.2.2 GPS

NAVSTAR-GPS je celosvětový satelitní navigační systém, vytvořený americkou armádou. GPS umožňuje zjišťování pozice, rychlosti směru a času na základě doby letu signálu od satelitu. Pro jednoduché mobilní roboty se nejčastěji využívají levné GPS přijímače, které s počítačem komunikují pomocí nějakého druhu sériového rozhraní (RS232, Bluetooth, USB, ...).

Nejčastěji využívaným protokolem pro tuto sériovou komunikaci je protokol NMEA ([19]), ale některé přijímače komunikují i jinými protokoly. Další možností pro komunikaci s GPS je například použití démona GPSD ([20]).

Model chyby

V této práci používám model chyby GPS, který pochází z [5]. Podle tohoto modelu se chyba měření GPS řídí podle Rayleighova rozložení:

$$p(chyba \leq x|HDOP) = 1 - e^{-x/\sqrt{(3.04 \cdot HDOP)^2 + 3.57^2}}$$

Projekce GPS souřadnic

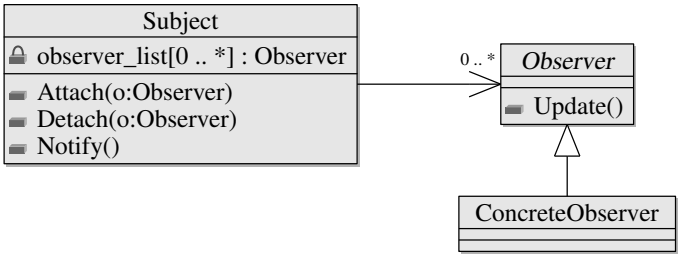
Výstupem z GPS je zeměpisná šířka a délka odhadované pozice ve formátu WGS84 ([19]). Pro použití této hodnoty v lokalizaci je ale nutné ji převést do kartézského souřadného systému. K tomu využívám ortografickou projekci ([21]). Tato projekce kolmo promítá bod zadaný zeměpisnou délkou a šířkou do roviny položené tečně na povrch země ve zvoleném referenční bodě.

Projekce funguje správně pouze pro body v blízkosti referenčního bodu, což je ale pro lokalizaci malých venkovních robotů dostatečné. Problémy by mohl nastat pouze v případě, že by robot cestoval řádově stovky kilometrů od referenčního bodu.

2.3 Návrhový vzor Observer

Observer je strukturní návrhový vzor, který řeší sdílení informací mezi objekty. Vytváří vazbu jednoho objektu zveřejňujícího informace k několika přijímajícím objektům. Přijímající objekty se mohou přihlašovat a odhlašovat k odběru informací za běhu a může jich být neomezený počet ([15]).

Na obrázku 2.3 jsou zobrazeny vztahy mezi třídami používajícími návrhový vzor Observer.



Obrázek 2.3: Návrhový vzor Observer

Tento návrhový vzor používá třídu *Subject* (někdy nazývanou také *Observable*), a rozhraní nazvané *Observer*. *Subject* obsahuje seznam přihlášených

objektů, splňujících rozhraní *Observer*. S tímto seznamem se manipuluje pomocí metod *Attach()* a *Detach()*.

Když u *Subject* nastane nová událost, tak jeho metoda *Notify()* projde jeho seznam přihlášených objektů a u každého z nich zavolá metodu *Update()*. Pokud je u notifikace nutné přenést i nějaká data, je možné je přidat například jako parametr *Notify()*.

Kapitola 3

Analýza problému

V této kapitole se věnuji analýze přístupů k tvorbě řídícího systém pro mobilní roboty a požadavků na něj, a to zejména z pohledu na řídící systém jako na prostředí propojující jednotlivé části programu robota.

3.1 Komponentový řídící systém

Existuje velké množství různých přístupů k řízení robotů, které se liší úrovní, na jaké chápou řízení robota, architekturou řídícího systému atd.

Jedním z možných přístupů je použití komponentového softwarového inženýrství. Takový přístup rozděluje řídící systém do komponent, řešících jednotlivé problémy, jako je například výběr akcí, lokalizace, nebo plánování trasy.

Každá komponenta reprezentuje jednu logicky oddělenou část systému. Komponenty jsou vzájemně oddělené a komunikují pouze pomocí sady přesně definovaných rozhraní. Komponenta v sobě zapouzdřuje data a algoritmy, potřebné pro svůj provoz ([13, 14]).

[13] navíc přidává další požadavek, a to že komponenty jsou uzavřené binární balíky a jejich uživatel nemusí při jejich používání pracovat s jejich zdrojovým kódem.

Přísné oddělení jednotlivých komponent zjednodušuje ladění řídícího systému. Navíc řídící systém vyvinutý pomocí komponentového návrhu je téměř automaticky modulární a tím pádem umožňuje opakovaně využívat jednotlivé komponenty, nebo používat hotové knihovny komponent ([7, 13]).

3.2 Požadavky na řídící systém

Na framework pro tvorbu robotického řídícího systému lze klást celou řadu požadavků ([11, 9]). V následujících odstavcích uvedu některé z nejvýraznějších požadavků:

Jednoduchost Jednoduché by mělo být používání frameworku, ale také jeho implementace.

Minimální režie Části řídícího systému, které propojují jednotlivé komponenty by měly mít pokud možno minimální nároky na výpočetní výkon a paměť Propojení by na jednotlivých komponentách nemělo vynucovat více práce, než je nezbytně nutné (například na konverzi formátů atd.).

Obecnost Framework by měl mít minimální požadavky na jednotlivé komponenty, na návrh robota, druhy senzorů, použité algoritmy a podobně.

Mělo by být možné v hotovém řídícím systému nahradit některou z komponent jinou podobnou komponentou bez nutnosti měnit zbytek systému.

Robustnost Robustní řídící systém je schopen vyrovnat se s chybnými vstupními daty, nebo se selháním některé z částí řídícího systému.

Nezávislost na platformě Výhodná je možnost přeložit řídící program pro různé platformy.

Nástroje pro ladění Ladění řídícího softwaru může být složitější než u ostatních druhů programů, zejména kvůli obtížné replikaci stavu fyzického robota. Framework pro řídící systémy by měl poskytovat nástroje pro zjednodušení ladění. Vhodná je také možnost připojení na simulátor a možnost jednoduchého použití logů.

Knihovna komponent Framework by měl obsahovat často používané algoritmy a nástroje, které může uživatel frameworku použít bez toho, že by musel znát jejich konkrétní implementaci.

3.3 Knihovna nástrojů

Podle [9] by jednotlivé frameworky měly být schopné používat společné knihovny algoritmů a nástrojů. Důvodem je, že zatímco potřebné vlastnosti frameworku závisí na přesných požadavcích na konkrétní řídící systém a jeho použití, jsou požadované vlastnosti komponent poměrně přesně určitelné a je možné vytvořit jednu správnou a obecně použitelnou implementaci.

Knihovna, která patří pouze ke konkrétnímu frameworku, se ale na druhou stranu nemusí omezovat snahou o co nejobecnější návrh. Může využívat speciálních vlastností daného frameworku a být díky jejich využívání jednodušší, nebo výkonnější. Podle mého názoru je proto vhodné, aby řídící systém umožňoval používat jak komponenty sdílené s jinými frameworky, tak speciální komponenty.

3.4 Řídící systém pro venkovní a vnitřní prostředí

Pokud se řídící systém chápe jako knihovna propojující jednotlivé komponenty, není mezi řízením venkovního nebo vnitřního robota prakticky žádný rozdíl. Tento rozdíl se projeví až u jednotlivých komponent, ze kterých se bude řídící software skládat, nebo z nastavení parametrů u některých z těchto komponent.

Například venkovní robot může používat GPS a magnetický kompas a jeho odometrie bude zatížena větší chybou, zatímco roboti pracující uvnitř budovy se mohou orientovat pomocí laserových dálkoměrů a na hladké podlaze může jejich odometrie pracovat velmi přesně.

3.5 Existující knihovny pro programování robotů

Je dostupných mnoho knihoven pro programování robotů, které se liší přístupy k návrhu robota, rozsahem a specializací, přenositelností atd. Tyto knihovny jsou většinou navzájem nekompatibilní a často používané jen malým počtem uživatelů ([22]).

Dvě z rozšířenějších knihoven a současně knihovny, které nejvíce ovlivnily tvorbu této práce, jsou Player a Orca.

3.5.1 Player

Player je síťový server pro řízení robotů. Poskytuje rozhraní pro komunikaci s hardwarem, které může být sdíleno mezi libovolný počet klientů. Kromě samotných hardwarových rozhraní umožňuje tato knihovna používat také virtuální hardwarová rozhraní poskytující algoritmy jako například lokalizace pomocí Kalmanových filtrů, nebo metodou Monte Carlo. Díky nim lze také Player částečně chápat jako komponentový systém ([13]).

Součástí projektu Player jsou také simulátory Stage a Gazebo, které je možné připojit ke klientovi místo skutečného hardwaru.

Více informací o tomto frameworku lze získat v [11] a [12].

3.5.2 Orca

Framework Orca je navržen jako přísně komponentový systém.

Na rozdíl od knihovny Player nevynucuje Orca žádnou konkrétní architekturu řídícího systému. Komponenty mohou plnit libovolný účel, a mohou být propojené v libovolné konfiguraci (vrstvená architektura, centrální komponenta, propojení všech komponent navzájem atd.).

Komunikaci mezi jednotlivými komponentami zajišťuje middleware knihovna ICE, díky níž může každá komponenta běžet na jiném počítači.

K frameworku Orca patří i knihovna Hydro, která obsahuje nástroje specifické pro framework Orca a knihovna Gearbox, která obsahuje nástroje nezávislé na použitém prostředí.

Detailnější informace o frameworku Orca se dají získat v [9] a [10].

Kapitola 4

Návrh frameworku

Následující kapitola popisuje a přístup k tvorbě řídícího systému zvolený pro framework RCT, rozdělení frameworku na jádro a komponenty a jejich návrh.

Hlavní motivací pro vytvoření frameworku RCT byla potřeba propojení různých algoritmů v řídícím systému a sdílení dat mezi nimi. RCT má za cíl usnadňovat tvorbu komponent a jejich propojování do řídícího systému. Umožňuje použití událostmi řízeného programování pro vytváření komponent a jejich snadné propojování do jednoho celku. Důraz se klade na postupný vývoj a testování řídícího systému a preferuje se jednoduchost i za cenu mírného snížení výkonnosti, nebo vynechání některé méně důležité funkčnosti.

4.1 Základní rozdělení

RCT dělí řídící systém do dvou základních částí: jádra a modulů. Jádro zajišťuje komunikaci mezi moduly a další podpůrné funkce, moduly reprezentují jednotlivé komponenty řídícího systému. Modul řeší některou z úloh nutných pro řízení robota a komunikuje s ostatními moduly v řídícím systému pomocí zasílání zpráv.

4.2 Jádro

Jádro RCT může pracovat s libovolnými, libovolně propojenými moduly. Řídící systém může používat jakoukoliv architekturu, od jednoduché sense-plan-act smyčky až po složité vrstvené systémy.

Snaha o jednoduchost se projevuje mimo jiné i v tom, že řídící systém je při použití RCT tvořený pouze jedním spustitelným souborem a běží pouze na jednom počítači. To je rozdíl oproti jiným frameworkům. Toto rozhodnutí na jednu stranu mírně omezuje použitelnost frameworku a porušuje požadavek na komponentový systém z [13], ale výhodou je značné zjednodušení jádra RCT, které nemusí řešit síťovou komunikaci, synchronizaci, atd. Toto omezení se

ale týká pouze této implementace jádra RCT, modulům nic nebrání v tom, aby komunikovaly po síti. RCT by bylo možné rozšířit na distribuovaný řídící systém pomocí proxy modulů a to bez nutnosti měnit cokoliv v jádře. Podobně by bylo možné přepracovat jádro a umožnit síťovou komunikaci bez změny API pro moduly.

Kromě propojování modulů poskytuje jádro RCT několik pomocných v funkcí, které jsou popsány v následujících odstavcích:

4.2.1 Nahrávání komunikace

RCT umožňuje nahrávat veškeré zprávy zasílané mezi moduly do souboru a později těmito nahrávkami nahrazovat výstup zvolených modulů. To umožňuje například replikovat stav řídícího systému v okamžiku kdy nastala chyba nebo testovat nové moduly bez nutnosti používat skutečný hardware, nebo simulátor.

4.2.2 Konfigurační soubory

Řídící systém robota může používat dva konfigurační soubory. První z nich je pouze pro čtení a obsahuje uživatelské nastavení, jako například fyzické rozměry robota, použité komunikační porty a podobné. Druhý konfigurační soubor (soubor stavu) slouží pro uložení parametrů, které robot získává za provozu. Může jít třeba o kalibrační údaje senzorů, hodnoty zadané z uživatelského rozhraní a podobně.

Hodnoty jsou v těchto souborech uložené jako dvojice řetězcového klíče a hodnoty libovolného typu. Při vyhledávání klíče se pak přednostně vyhledává v souboru stavu a pokud se v něm požadovaný klíč nenachází, pak vyhledávání pokračuje v konfiguračním souboru.

Každý modul má navíc vlastní jmenný prostor pro uložení svých nastavení.

4.2.3 Logování

Další pomocnou funkčností jádra RCT je zaznamenávání zpráv. Zprávy se zapisují na standardní výstup, do souboru, nebo se odesílají systémovému loggeru (syslog).

4.3 Zprávy

Zprávy jsou datové struktury, pomocí kterých probíhá komunikace ve frameworku RCT. Každá zpráva má zadaný typ, obsahuje uživatelská data a je schopná tato data uložit nebo načíst (viz 4.2.1). Příkladem typu zprávy může být měření dálkoměru, příkaz pro motory, nebo informace o novém odhadu z lokalizace. Zprávy mohou využívat dědičnost a polymorfismus.

4.4 Moduly

Modul má definovanou sadu vstupů a sadu výstupů, každý z nich s určeným typem zprávy. Dva moduly je možné propojit, pokud typ zprávy u přijímajícího modulu je předkem typu zprávy u odesílajícího modulu (zprávu je možné bezpečně přetypovat z odesílaného typu na přijímaný typ). Na výstup modulu se může připojit libovolné množství modulů přijímajících zprávy odpovídajícího typu, zpráva odeslaná modulem je doručena všem modulům propojeným s odesílajícím modulem. Zprávy pak každý příjemce zpracovává v obslužné rutině.

Modul může buď pouze reagovat na přicházející zprávy, nebo může mít navíc spuštěná vlastní vlákna a v nich provádět jakoukoliv další činnost. RCT umožňuje spouštět a ukončovat tato pracovní vlákna všech modulů najednou.

Přijetí zprávy by mělo být co nejrychlejší, protože uživatelské obslužné rutiny se volají ve vlákně odesílajícím zprávu a to postupně pro jednotlivé přijímající moduly. Pokud modul potřebuje na přijetí zprávy více času, měl by si vytvořit vlastní vlákno a neblokovat odesílanou zprávu.

Přijatá zpráva musí být platná pouze po dobu, po kterou probíhá její odesílání. Pokud některý modul potřebuje data z přijaté zprávy i po opuštění obsluhy přijetí zprávy, pak si musí vytvořit její kopii.

4.5 Knihovna modulů

Součástí RCT je několik modulů provádějících běžné operace, které jsou popsány níže. Kromě těchto modulů ale je také možné jako vstup, nebo výstup řídícího systému použít Player klienta.

4.5.1 Společné vlastnosti modulů

Kromě samotných modulů a jejich rozhraní je vhodné specifikovat ještě několik dalších vlastností, které sice nejsou explicitní součástí rozhraní, ale usnadní návrh modulů a práci s nimi.

Jednotky, numerické typy

Je nutné určit jednotky a datové typy používané při komunikaci mezi moduly.

RCT pracuje s čísly v plovoucí řádové čárce, typu `double` a používá jednotky SI. Úhlové jednotky používané v rozhraních modulů jsou radiány, ale hodnoty přístupné z uživatelského rozhraní (konfigurační soubory) jsou ve stupních. Pro reprezentaci času se využívá typ `boost::posix_time::ptime`.

Výhodou tohoto řešení je jednodušší práce s hodnotami probíhajícími mezi moduly a intuitivnější rozhraní modulů.

Souřadné systémy

Pro souřadnice v mapě se využívá souřadný systém, který má počátek v libovolném bodě (při použití GPS viz 5.2.1), kladný směr osy X směřující na východ a kladný směr osy Y na sever.

V souřadném systému robota směřuje osa X vpravo a osa Y dopředu.

Pozice senzoru na robotovi

Moduly, které jsou součástí RCT, podporují libovolnou polohu senzoru na robotovi. Tyto parametry každý modul načítá z konfiguračních souborů.

Rozdělení zodpovědnosti modulů

Další rozhodnutí, které ovlivní návrh knihovny modulů je zamýšlené rozdělení zodpovědnosti mezi moduly.

Jedním extrémem by byl jediný modul, který obsahuje celý řídící systém a nevyžaduje žádné další doplnění. Očividná nevýhoda tohoto řešení je to, že prakticky ignoruje veškeré výhody přinášené komponentovým systémem. Je například vhodné mít možnost nahrávat data vstupující do řídícího systému předtím, než projdou jakýmkoliv zpracováním, protože to umožňuje využít nahrávku při ladění větší části řídícího systému.

Druhý extrém je rozdrobení veškeré funkčnosti do velmi jednoduchých modulů. To už neposkytuje žádné zjednodušení řídícího systému a může trpět zvýšenou režií, způsobenou přenášením zbytečného množství informací skrz framework.

Moduly v RCT se snaží rozdělit funkčnost tak, aby tato režie byla co nejnižší a aby současně umožnily co nejlepší opakované využívání modulů a co nejpružnější využití vlastností frameworku.

4.5.2 Monte Carlo lokalizace

Modul Monte Carlo lokalizace je nejdůležitějším modulem frameworku RCT. Zajišťuje lokalizaci metodou Monte Carlo, a to obecně, bez ohledu na použité senzory nebo reprezentaci stavu robota.

4.5.3 Sériový port

Součástí RCT je také modul pro obousměrnou sériovou komunikaci.

Tento modul je primárně určený pro komunikaci přes sériový port, ale je možné jej použít i s jinými rozhraními.

4.5.4 GPS

Modul GPS zpracovává NMEA data z GPS přijímače na sériovém portu. Data modul získává z modulu pro sériový port. Tento modul je možné připojit jako vstup lokalizace.

4.6 Knihovna pomocných nástrojů

4.6.1 Náhodné proměnné

V RCT jsou funkce pro generování pseudonáhodných hodnot z rovnoměrného a z normálního rozložení. Tyto funkce používají pro získání pseudonáhodných hodnot funkci `random()` ze standardních knihoven, hodnoty z normálního rozložení se generují pomocí polární formy Box-Mullerovy transformace ([16]).

Pro normální rozložení je k dispozici také funkce, která počítá hodnotu hustoty v zadaném bodě.

4.6.2 Diferenciální pohon

Pro zjednodušení práce s řízením robota s diferenciálním pohonem je k dispozici přepočítávání dopředné rychlosti a úhlové rychlosti na rychlosti kol a naopak.

Kapitola 5

Popis implementace

Tato kapitola popisuje implementaci jádra a modulů RCT a věnuje se některým implementačním rozhodnutím, které vznikly při tvorbě frameworku.

RCT je implementované v C++, používá knihovny boost. RCT je programováno s ohledem na přenositelnost, momentálně však bylo testováno pouze na POSIXových systémech.

Pro podrobný popis API použijte doxygenem vygenerovanou dokumentaci v adresáři `rct/doc/generated-dev`.

5.1 Jádro

5.1.1 Struktura jádra

Jádro RCT se z pohledu uživatele skládá z třídy `Master` a šablonových tříd `Receives<T>` a `Sends<T>`, které slouží jako základní třídy pro odvozování modulů.

Master je centrální prvek pro funkce, které jsou společné pro celý řídící systém. Udržuje seznam modulů, pouze moduly přidané ke stejnému masteru spolu mohou komunikovat.

Moduly mají jednoznačné řetězcové identifikátory, které se používají při nahrávání a přehrávání zpráv a pro vypisování v logu.

5.1.2 Rozhraní modulů a zasílání zpráv

Vhodným řešením propojení modulů by se mohl zdát návrhový vzor Observer. Ten řeší zasílání zpráv a propojení jednoho zdrojového objektu s několika cílovými.

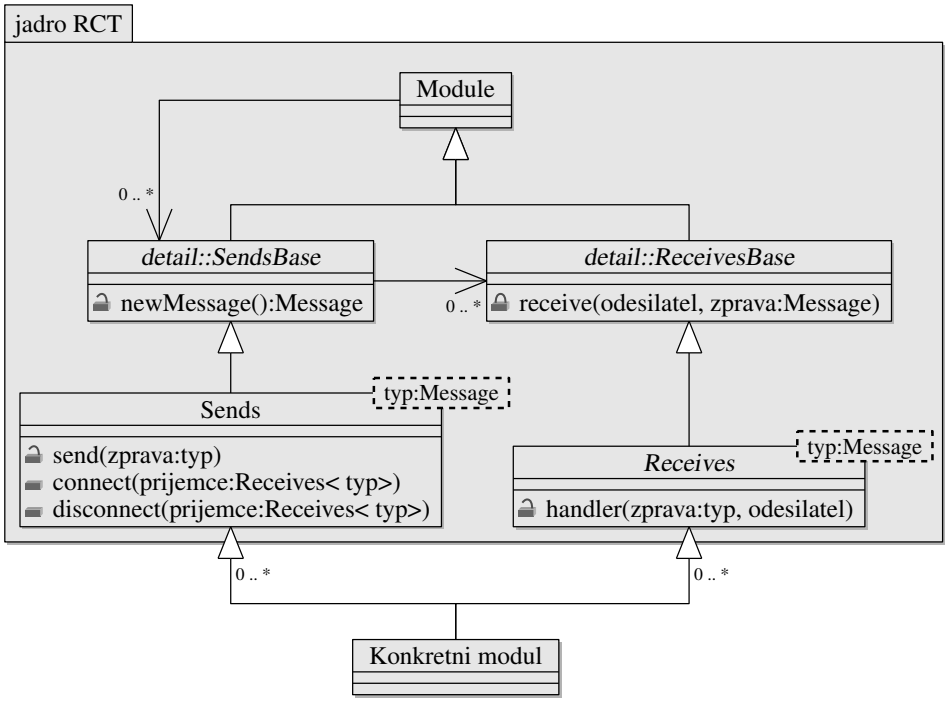
Přímočarému použití tohoto návrhového vzoru ale brání požadavek, aby modul mohl odesílat i přijímat neomezený počet typů zpráv. Dalším problémem je, že ke každému modulu a odesílanému typu zprávy mohou být v RCT připojeny moduly s různými typy přijímaných zpráv (viz 4.4). Poslední komplikací je potřeba zaznamenávat komunikaci probíhající v systému a později ji přehrávat.

Proto používám řešení odvozené od návrhového vzoru Observer založené na vícenásobné dědičnosti.

Observer je v tomto řešení nahrazený šablonovou třídou `Receives<T>` a *Subject* třídou `Sends<T>`. Parametrem šablony je v obou případech typ zprávy se kterou se pracuje. Každý modul tedy vlastně používá několikanásobný Observer a jednotlivé instance tohoto návrhového vzoru se odlišují typem zprávy.

Když modul odešle zprávu (pomocí metody `Sends<T>::send()`), použije se seznam příjemců ze třídy `detail::SendsBase` (to je předek `Sends<T>`). V tomto seznamu jsou uložené ukazatele na `detail::ReceivesBase`, které odpovídají připojeným modulům. Zpráva se odešle zavoláním metody `receive()` u každého objektu ze seznamu příjemců. Tato metoda je virtuální, nadefinovaná v `Receives<T>` a dokáže tedy přetypovat zprávu na požadovaný přijímaný typ a nakonec zavolat uživatelskou obslužnou rutinu. Protože propojit je možné pouze moduly s kompatibilními typy zpráv, mohou být tato přetypování statická a budou vždy korektní.

Na obrázku 5.1 je diagram této části jádra.



Obrázek 5.1: Třídy v jádře RCT

Propojování modulů

Propojování modulů zajišťuje metoda `Sends<T>::connect()`, která má jako parametr přijímající modul. Při propojování modulů se cílový modul přidá do seznamu příjemců ve třídě `detail::SendsBase` odesílajícího modulu.

Protože ale syntaxe pro volání této metody (s vícenásobnou dědičností) je poněkud komplikovaná a nepřehledná (v příkladu 5.1 se propojuje `modul1` a `modul2`, s odesílaným typem `TypA` a přijímaným typem `TypB`), jsou v masteru definovány obalovací metody pro připojení a odpojení modulů (příklad 5.2).

```
modul1->Sends<TypA>::connect<TypB>(modul2);
```

Příklad 5.1: Propojení modulů – přímá varianta

```
master.connect<TypA, TypB>(modul1, modul2);
```

Příklad 5.2: Propojení modulů – varianta používající master

5.1.3 Nahrávání zpráv

Pro nahrávání zpráv je nutné mít nástroj pro serializaci zprávy do souboru. Není ale nutné podporovat ukládání komplikovaných struktur, jako jsou cyklické grafy ukazatelů apod. Z tohoto důvodu jsem se rozhodl zvolit vlastní řešení místo hotové serializační knihovny, jako je například boost serialize.

Abstraktní třída `Message`, která je předkem všech zpráv zasílaných v řídícím systému má čistě virtuální metodu `save()`, ve které zpráva ukládá svoje parametry do souboru. Tuto metodu musí nadefinovat uživatel u každé nové třídy, kterou vytvoří a uživatel je také zodpovědný za uložení všech datových položek a předků zprávy v této metodě. Zápis do souboru zajišťuje třída `RecordingW`. Hodnoty se konvertují na řetězec pomocí `boost::lexical_cast` a zapisují do souboru. Ukládání hodnot je společné s ukládáním konfiguračních souborů (viz 5.1.5). Každá zpráva má navíc definovaný jednoznačný název.

Při odesílání zprávy se u masteru zkontroluje, jestli je nahrávání aktivní a pokud ano, tak se uloží čas uplynulý od poslední odeslané zprávy, identifikátor odesílajícího modulu, jméno zprávy a zavolá metodu `save()` zprávy.

Aby bylo možné zaznamenané zprávy přeskakovat bez čtení, vkládá se za každou zprávu oddělovací sekvence (oddělovací sekvence je jednoznačná, není ji možné vytvořit uložením nějakých dat).

Záznam je možné za běhu komprimovat algoritmem gzip. K tomu se využívá knihovna boost iostreams a její třída `filtering_ostream`, která umožňuje vytvářet streamy pomocí skládání filtrů. Pokud tedy uživatel požaduje kompresi záznamu, přidá se k výstupnímu streamu filtr pro gzip.

5.1.4 Přehrávání zpráv

Přehrání zprávy uložené v souboru probíhá následovně:

1. Pokud se to při spouštění přehrávání požaduje, čeká se po dobu zaznamenanou v souboru, aby přehrané zprávy odcházely se stejným časováním jako při nahrávání.

2. Zjistí se modul, který odeslal zprávu (podle uloženého identifikátoru) a přečte se název zprávy.

3. V seznamu všech odesílaných typů zpráv tohoto modulu se vyhledá `detail::SendsBase`, který odesílá zprávu s načteným jménem. (Tento seznam je v třídě `Module` a obsahuje všechny odesílané typy spolu s ukazateli na odpovídající `detail::SendsBase`.)

4. Metodou `new_message()` takto nalezeného `detail::SendsBase` se vytvoří nová instance zprávy požadovaného typu.

5. Obsah zprávy se načte pomocí metody `Message::load()` (obdobná jako `Message::save()`).

6. Zpráva se doručí stejným způsobem jako při běžném odesílání.

Toto řešení má výhodu v tom, že při načtení každé zprávy dokáže její odesílající modul vytvořit instanci této zprávy podle jména a načíst ji. Vyhýbá se tak nutnosti registrovat nové typy zpráv, na druhou stranu ale modul, který odesílal zprávu, musí být přidaný k masteru i při přehrávání.

Pokud selže vyhledávání modulu nebo typu zprávy, pak se zpráva přeskočí a do logu se vypíše varování.

Moduly mají příznak přehrávání a načtená zpráva se odešle pouze v případě, že modul označený jako odesílatel má tento příznak nastavený na `true` (tento příznak uživatel nastaví u modulů, jejichž výstupy chce přehrávat).

Otevírání komprimovaných záznamů je transparentní – RCT se pokusí otevřít soubor s dekompresí, a pokud toto selže, pokusí se jej otevřít přímo.

Jestli je soubor korektním záznam RCT se zjistí z magického řetězce, který je vždy první řádkou v nahrávce.

5.1.5 Konfigurační soubory

Konfigurační soubory ukládají dvojice řetězcového klíče a libovolné hodnoty. V paměti je konfigurační soubor reprezentovaný jako `std::map` řetězcových klíčů a řetězcových hodnot a při každém přístupu k některé hodnotě se tato přetypuje z požadovaného typu (při zápisu), nebo na požadovaný typ (při čtení) pomocí `boost::lexical_cast`.

Každý modul přidaný k masteru má k dispozici vlastní jmenný prostor v konfiguračních souborech. Tento jmenný prostor vznikne přidáním identifikátoru modulu před vyhledávaný klíč. Moduly ale mají možnost zasahovat i do jiných jmenných prostorů a RCT tyto přístupy nijak nekontroluje.

Formát souboru

Do souboru na disku se obsah mapy konfiguračního souboru zapisuje jako dvojtečkami oddělené dvojice řetězců a každá dvojice je ukončená znakem nové řádky. Speciální znaky včetně dvojteček, znaku nového řádku a podobně se v souboru zapisují v uvozovkách, nebo pomocí escape sekvencí podobných jako používá bash nebo C. Funkce pro zápis řetězců do souboru a jejich čtení jsou soustředěné ve třídě `detail::ConfigFileBase` a používají se i pro záznam a přehrávání zpráv.

5.1.6 Logování

Logovací zprávy v RCT mají nastavenou úroveň důležitosti. Tyto úrovně odpovídají prioritám v syslog(3). Pro každý modul je nastavená jeho minimální úroveň, zprávy s nižší úrovní se ignorují.

Za zmínku stojí ještě makra zapisující hlášení do logu. Tato makra nejprve otestují, jestli je zadaná úroveň zprávy dost vysoká. Pokud je, tak se vytvoří nový `std::stringstream` a zadaný výraz se do něj operátorem `<<` zapíše. Na výstup se pak posílá až obsah tohoto stringstreamu. Díky tomu je možné zjistit konec zprávy a správně zamykat logovací soubor.

5.1.7 Vícevláknový běh

RCT předpokládá, že řídící systém bude fungovat v několika vláknech. Části jádra, které mohou být využívány z modulů (logování, konfigurační soubory, nahrávání zpráv, atd.), jsou zabezpečené zamykáním.

Pro hromadné spouštění pracovních vláken modulů definuje třída `Module` virtuální metody `start()` a `stop()`. Pokud modul používá vlákna, pak by tyto metody měly být potlačené a spouštět respektive zastavovat všechna vlákna modulu.

5.2 Knihovna modulů

5.2.1 Lokalizace

Modul lokalizace v RCT používá algoritmus Monte Carlo lokalizace.

Aby byl mohl pracovat co nejobecněji, používá celý tento modul šablonu s typem vzorku. Vzorek popisuje jednu možnou pozici robota, v závislosti na tomto parametru může modul lokalizovat robota v libovolně složitých stavových prostorech.

Rozhraní

Výstupem tohoto modulu je zpráva s odhadem pozice, reprezentovaná třídou `MCL::PositionEstimate<T>`. Jejím parametrem šablony je typ vzorku a pomocí explicitní specializace šablon se vytváří odhady pro různé typy vzorků.

Vstupem lokalizace jsou dva druhy zpráv, první z nich reprezentuje predikční data vstupující do lokalizace a druhá korekční data. V tomto místě se využije schopnost RCT používat u zpráv polymorfismus. Lokalizace definuje dvě abstraktní třídy (opět závisející na typu vzorku) pro zprávy sloužící jako vstup do lokalizace.

`MCL::Prediction<T>` je předkem zpráv obsahujících predikční data a má čistě virtuální metodu `predict()`, která dostane jako parametr vzorek a upraví jej podle nových dat.

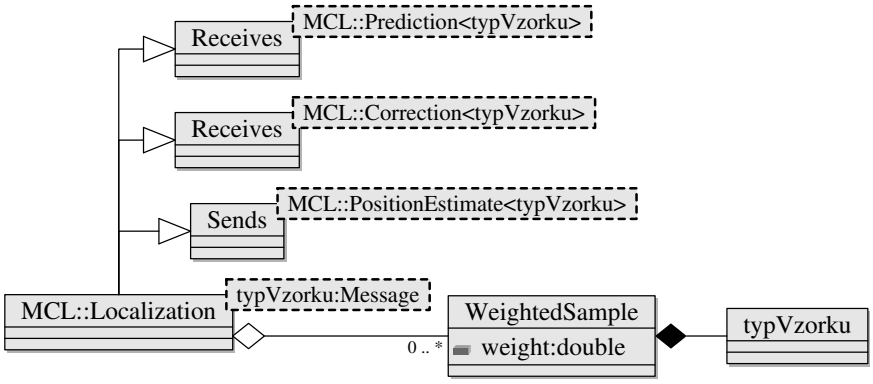
`MCL::Correction<T>` slouží pro zprávy obsahující korekční data. Její metoda `correct()` dostane jako vstup vzorek spolu s jeho váhou a vrací novou upravenou váhu.

Zprávy, které slouží jako vstup do lokalizace, jsou odvozené od těchto tříd namísto od `Message`.

Obě třídy pro vstupy do lokalizace mají metodu `prepare()`, která se zavolá předtím než se zpráva použije pro úpravu vzorků. Tato metoda umožňuje předpočítat mezivýsledky společné pro jednu iteraci lokalizačního algoritmu. Například zpráva s GPS odhadem v této metodě počítá konstanty závislé na odhadu horizontální chyby, tyto hodnoty se pak používají pro všechny vzorky.

Všechny tři metody u zpráv vstupujících do lokalizace mají jako parametr ještě ukazatel na aktuální mapu používanou lokalizací.

Závislosti na typu vzorku jsou zobrazeny na obrázku 5.2



Obrázek 5.2: Diagram tříd sloužících pro lokalizaci

Mapa K tomu, aby lokalizace mohla správně pracovat, musí znát mapu prostředí, ve kterém se robot lokalizuje. Mapa je definovaná pouze rozhraním a je opět závislá na typu vzorku. Její konkrétní implementace závisí na uživateli.

Mapa není modul – nekomunikuje s lokalizací zasíláním zpráv, lokalizace pouze volá její metody.

Mapa musí umožňovat vygenerovat náhodný vzorek pro novou inicializaci globální lokalizace.

Pro dálkoměry je v rozhraní mapy metoda pro změření vzdálenosti ke stěně. Tato metoda bere jako parametr bod a směr a vrací vzdálenost k nejbližší stěně v zadaném směru.

Pro použití GPS je v rozhraní mapy metoda pro získání GPS souřadnic bodu, který má v mapě souřadnice $(0, 0)$.

Jednotné rozhraní mapy bez ohledu na použité senzory je poměrně omezující. Pokud totiž budou používané senzory požadovat, složitější rozhraní mapy, je nutné ji explicitně přetypovávat a tím se v případě statického přetypování ztrácí typová bezpečnost, nebo v případě použití `dynamic_cast` výkon. Řešením tohoto problému by bylo, aby každý druh vstupu do lokalizace mohl vyžadovat vlastní rozhraní mapy. To je ale v C++ poněkud obtížné a zatím to nebylo implementováno.

Vnitřní implementace

Algoritmus Monte Carlo lokalizace funguje po iteracích. Iterace začíná predikcí nového stavu robota, tomu odpovídá přijetí predikční zprávy. Přijetí této zprávy ukončí předchozí iteraci (vygeneruje nový výstup, převzorkuje, ... Viz dále).

Samotná predikční fáze probíhá tak, že se pro všechny vzorky zavolá metoda `predict()` přijaté predikční zprávy.

Po přijetí korekční zprávy se projdou vzorky a jejich váhy se upraví pomocí metody `correct()` přijaté korekční zprávy.

Možina vzorků lokalizace je reprezentovaná jako `std::list` struktur, které obsahují vzorek spolu s jeho váhou.

Odhadu pozice generuje tento modul pomocí třídy `MCL::PositionEstimate<T>`. Ta je odvozená od zprávy s odhadem pozice `PositionEstimate<T>` a dokáže odhad vypočítávat ze seznamu vzorků interně používaných modulem Monte Carlo lokalizace. Pro odhady s různými typy vzorků se opět používá explicitní specializace šablony.

Převzorkování (viz 2.1.2) se provádí pomocí algoritmu 2 z [6] v případě, že efektivní počet vzorků klesne pod nastavenou hranici (výchozí hodnota je 75% ze všech vzorků).

Lokalizace ve dvourozměrném prostoru

V RCT je implementovaná pouze lokalizace ve dvourozměrném prostoru. Typ `MCL::Sample2D` používaný jako vzorek obsahuje X a Y souřadnici a natočení robota.

Odhad pozice pro dvourozměrný prostor je reprezentovaný specializovanou třídou `PositionEstimate<MCL::Sample2D>`, která obsahuje střední hodnotu

souřadnic a orientace a kovarianční matici těchto hodnot.

Lokalizace ve složitějších prostorech

Pro použití modulu Monte Carlo lokalizace se složitějšími stavovými prostory by bylo nutné vytvořit strukturu reprezentující vzorek a třídy `PositionEstimate<T>` a `MCL::PositionEstimate<T>` explicitně specializované na tento typ vzorku.

Také by bylo nutné vytvořit nové zprávy se vstupy do lokalizace, protože stávající zprávy pracují pouze s 2D vzorky.

5.2.2 Sériová komunikace

Modul pro sériovou komunikaci má vytvořené vlákno (viz 5.1.7), ve kterém blokovaně čte data z portu. Přečtená data pak odesílá jako zprávy. Protože sériový port odesílá pouze samostatné znaky, obsahuje standardně každá odcházející zpráva pouze jeden přečtený znak. Existuje ale možnost seskupovat celé řádky ukončené zadaným znakem. V takovém případě modul neodešle zprávu s přečtenými daty, dokud nepřečte zvolený znak konce řádku.

Tento modul zajišťuje obousměrnou komunikaci, to znamená, že data z přijatých zpráv se zapisují do otevřeného portu.

5.2.3 GPS

Modul pro GPS má jako vstup data z modulu pro sériovou komunikaci. Čte data ve protokolu NMEA, v současné verzi zpracovává pouze věty GPGGA, což je dostatečné pro zjištění polohy.

5.2.4 Ostatní senzory

Kromě GPS umožňuje RCT využívat ještě kompas a dálkoměry, ale protože pro tyto senzory není k dispozici standardizované hardwarové rozhraní, nejsou pro ně v RCT moduly. Protože je ale model chyby senzoru obsažený ve zprávě, může autor modulu tyto zprávu jednoduše použít bez jakéhokoliv dalšího programování.

5.2.5 Player klient

Moduly připojující RCT k Player serveru jsou implementovány jako obal kolem tříd z knihovny libplayerc++.

Pro každé rozhraní Playeru je potřeba vytvořit nový modul, který obsahuje proxy třídu z libplayerc++.

Při zavolání metody `Module::start()` takového modulu se spustí čtecí vlákno pro `PlayerClient`, pokud ještě neběželo a přihlásí se odběr oznámení o nových datech player klienta.

Pro zjednodušení je v RCT třída `detail::PlayerProxyWrapper<T>`, která slouží jako základ pro obalovací třídy a řeší tyto problémy.

5.3 Knihovna pomocných nástrojů

5.3.1 Náhodné proměnné

Pro generování hodnot z normálního rozdělení pravděpodobnosti se využívá polární forma Box-Mullerovy transformace ([16]). Protože tato metoda vytváří dvojice hodnot, jsou vytvořena dvě rozhraní pro práci s nimi. Jednoduché rozhraní vrací pouze jednu vygenerovanou hodnotu a druhou z nich zahazuje. To je vhodné pro generování malého množství náhodných hodnot. Pro generování většího počtu hodnot z tohoto rozložení se dá využít funkce, která vrací obě vygenerovaná čísla.

Pro generování hodnot z rovnoměrného rozdělení se využívá generátor pseudonáhodných čísel z knihovny C (funkce `rand()`).

Kapitola 6

Závěr

Cílem této práce bylo vytvořit řídící systém pro mobilního robota. Framework RCT, který při řešení této práce vznikl, je jednoduchý modulární systém, usnadňující tvorbu řídících systémů a pokusy s jejich jednotlivými moduly.

V porovnání s jinými existujícími knihovnami je vidět zjednodušující přístup RCT, ale zejména v kombinaci s moduly Player klienta může RCT sloužit k sestavení plnohodnotného řídícího systému, pokud nejsou vyžadovány pokročilé vlastnosti, jako je například rozdělení řídícího systému na více počítačů. V RCT je mnoho prostoru pro rozšiřování a vylepšování, a to zejména v knihovně modulů, která je v současné verzi spíše ilustrací možností frameworku, než plnohodnotným nástrojem. V jádře je to například přidání zpráv s možností odpovědi, v knihovně modulů by to mohlo být zvýšení výkonosti modulu lokalizace, nebo důslednější zpracování dat z GPS.

Literatura

[1] J Borenstein, HR Everett, L Feng: *Where am I? Sensors and Methods for Mobile Robot Positioning*, University of Michigan, 1996

[2] Sebastian Thrun, Dieter Fox, Wolfram Burgardz, Frank Dellaert: *Robust Monte Carlo Localization for Mobile Robots* Artificial Intelligence, Summer 2001

[3] D Fox, J Hightower, L Liao, D Schulz, G Borriello: *Bayesian filtering for location estimation*, IEEE Pervasive Computing, 2003

[4] D Fox, W Burgardy, F Dellaert, S Thrun: *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots*, Proc. of the National Conference on Artificial Intelligence, 1999

[5] WILSON, David L.: *HDOP and GPS Horizontal Position Errors* `http://users.erols.com/dlwilson/gpshdop.htm`

[6] Sanjeev Arulampalam, et al.: *A tutorial on particle filters for online nonlinear/non-GaussianBayesian tracking*, IEEE Transactions on signal processing, 2002

[7] A. Makarenko, A.Brooks, T.Kaupp: *On the Benefits of Making Robotic Software Frameworks Thin*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007). Workshop on Evaluation of Middleware and Architectures, 2007

[8] Erwin Prassler: *Comparative Evaluation of Robotic Software Integration Systems: A Case Study*, Intelligent Robots and Systems, 2007

[9] Alexei Makarenko, Alex Brooks, Tobias Kaupp: *Orca: Components for Robotics* EEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006). Workshop on Robotic Standardization, 2006

[10] *Webové stránky frameworku Orca* `http://orca-robotics.sourceforge.net/orca/index.html`.

[11] Toby H. J. Collett, Bruce A. MacDonald, Brian Gerkey: *Player 2.0: Toward a Practical Robot Programming Framework* Proceedings of the Australasian Conference on Robotics and Automation, 2005

[12] *Webové stránky frameworku Player*
http://playerstage.sourceforge.net

[13] Alex Brooks, et al.: *Towards component-based robotics* Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on, 2005

[14] G. T. Heineman, W. T. Councill: *Component-based software engineering: putting the pieces together*. Addison-Wesley, Boston, 2001

[15] Benneth Christiansson, et al.: *GoF Design Patterns – with examples using Java and UML2*
http://www.scribd.com/doc/9973578/Design-Patterns-Explained-With-Java-and-Uml2-2008

[16] Everett F. Carter, Jr: *Generating Gaussian Random Numbers*
http://web.archive.org/web/20061110124742/http://www.taygeta.com/random/gaussian.html

[17] Agostino Martinelli and Roland Siegwart: *Estimating the Odometry Error of a Mobile Robot during Navigation*

[18] Kok Seng Chong, Lindsay Kleeman: *Accurate odometry and error modelling for a mobile robot* Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on

[19] Dale DePriest: *NMEA data*
http://www.gpsinformation.org/dale/nmea.htm

[20] *Webové stránky projektu GPSD*
http://gpsd.berlios.de/

[21] Eric W. Weisstein: *Orthographic Projection* From MathWorld – A Wolfram Web Resource.
http://mathworld.wolfram.com/OrthographicProjection.html

[22] Michael Somby: *Updated review of robotics software platforms.*
http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Updated-review-of-robotics-software-platforms/ (2008)

Příloha A

Obsah přiloženého CD

- `bc.pdf` Tento text.
- `rct/` Kořenový adresář frameworku RCT.
- `rct/demo/` Ukázkový program.
- `rct/doc/` Dokumentace frameworku.
- `rct/doc/generated-dev/` Vygenerovaná programátorská dokumentace.
- `rct/doc/generated-user/` Vygenerovaná uživatelská dokumentace.
- `rct/tests/` Testy frameworku.

Příloha B

Uživatelská dokumentace

V této části dokumentace jsou popsány puze nejdůležitější části API. Pro detailní popis použijte doxygenem vygenerovanou dokumentaci v adresáři `rct/doc/generated-user/`.

B.1 Instalace

RCT využívá systém řízení překladu Cmake, knihovny boost a zlib. V případě, že je při konfiguraci nastavena instalace modulů pro Player klienta, vyžaduje RCT také knihovnu playerc++.

Pro překlad RCT použijte následující příkazy:

```
mkdir build
cd build
ccmake ../
make install
```

Tím se vytvoří adresář pro překlad, spustí se uživatelské rozhraní pro konfiguraci překladu a následně se RCT v tomto adresáři přeloží a nainstaluje se.

Při prvním spuštění konfiguračního nástroje je nutné klávesou `c` spustit konfiguraci. Následně je možné upravit parametry překladu (zejména `RCT_*_WITH_*` upravující které části frameworku se přeloží, a `CMAKE_INSTALL_PREFIX` určující instalační prefix). Poté opět klávesou `c` se tyto hodnoty zpracují a klávesou `g` se vygenerují soubory Makefile a ukončí se editor.

RCT se instaluje do adresářů `prefix/include/rct/`, `prefix/lib/` a vytváří i soubory pro pkgconfig v `prefix/lib/pkgconfig/`.

Výsledkem instalace jsou knihovny `rctcore`, obsahující pouze jádro frameworku, a `rctmodules`, obsahující moduly a zprávy.

Deklarace RCT se nacházejí v hlavičkovém souboru RCT.h. Všechny části knihovny jsou ve jmenném prostoru RCT.

B.2 Moduly

Modul v RCT je potomkem třídy `Module` a je označený jednoznačným řetězcovým identifikátorem, který předává jako parametr konstruktoru třídy `Module`. Pokud modul přijímá nebo odesílá jakékoliv zprávy, pak by neměl být odvozený přímo od třídy `Module`, ale měl by využívat tříd `Sends<T>` a `Receives<T>` (viz níže).

`Sends<T>` je předek pro třídy, které odesílají zprávy, parametr šablony `T` určuje typ zprávy.

Pro moduly, které zprávy přijímají je určena třída `Receives<T>`, parametr `T` určuje opět typ zprávy. Uživatel pak musí nadefinovat metodu `handler()`, která zpracovává přijatou zprávu.

Handler se volá z vlákna odesílajícího modulu, a proto by měl zprávu zpracovat co nejrychleji, nebo si pro její zpracování spustit vlastní vlákno.

To že modul je odvozený od těchto tříd umožňuje využít polymorfismu a typové kontroly v C++ pro zajištění typové korektnosti odesílaných a přijímaných zpráv.

V příkladu B.1 je ukázka modulu, který přijímá zprávy typu `SomeMessage` a odesílá zprávy typu `SerialData`.

```
class FunnySerial: public Sends<SerialData>,
    public Receives<SomeMessage>{
public:
    FunnySerial(const char *name) : Module(name) {}

    void handler(SomeMessage *m, Module *source){
        SerialData *d = new SerialData();
        d->data = "Hello World!";
        send<SerialData>(this, d);
        delete d;
    }
};
```

Příklad B.1: Modul, který přijímá zprávy typu `SomeMessage` a odesílá zprávy typu `SerialData`

Moduly se identifikují pomocí jména zadaného v konstruktoru třídy Module. Jméno musí být neprázdné, složené pouze z alfanumerických znaků a podtržítka. Jména všech modulů použitých s jedním masterem musí být unikátní. Tento požadavek se kontroluje při přidání modulu.

B.3 Zprávy

Zprávy jsou odvozené od třídy `Message`. Každá zpráva musí mít definované metody `save()` a `load()` a statickou položku `const char *name` udávající jméno

typu zprávy (toto jméno musí být opět unikátní, jinak nebude správně fungovat načítání nahrávek).

Metoda `save()` ukládá zprávu do záznamu. Parametr této metody je ukazatel na objekt typu `RecordingW`, jeho metoda `save_value()` ukládá jednotlivé hodnoty do záznamu.

Metoda `load()` funguje obdobně, používá `RecordingR::load_value()` pro načítání hodnot.

Vytváření zpráv ilustruje příklad B.2.

```
class SomeMessage: public Message{
public:
    void save(RecordingW *r) const{
        r->save_value(a);
    }

    void load(RecordingR *r){
        r->load_value(a);
    }

    int a;

    static const char *name;
};
```

Příklad B.2: Zpráva `SomeMessage`

`save_value()` a `load_value()` používají `boost::lexical_cast` a dokážou tedy pracovat s typy, které mají definované operátory `>>` a `<<`. Pro ostatní typy musí uložení definovat uživatel.

Pokud zpráva dědí od jiné zprávy, musí jako první zavolat metodu `save()` svého předka (příklad B.3).

B.4 Odesílání zpráv a propojování modulů

Zprávy se odesílají pomocí metody masteru `send<T>()`. Parametrem šablony je typ odesílané zprávy, parametry funkce jsou ukazatel `Sends<T>`, který zprávu odesílá a ukazatel na zprávu (typu `T`). Alternativní možností odesílání je metoda `Sends<T>::send()`, jejíž jediný parametr je ukazatel na odesílanou zprávu.

Takto odeslanou zprávu přijmou všechny moduly, které jsou připojené k odesílajícímu modulu pro zprávy typu T.

Moduly se propojují pomocí metody masteru `connect<T, U>()`. Parametry šablony jsou odesílaný a přijímaný typ zprávy, parametry metody jsou ukazatele na odesílající a přijímající modul (viz příklad B.4). Musí platit, že typ U je předek typu T. Pokud je zadaný pouze jeden typ, použije se přijímaný typ stejný

```
class AnotherMessage: public SomeMessage{
public:
    void save(RecordingW *r) const{
        SomeMessage::save(r);
        r.save_value(s);
    }

    void load(RecordingR *r){
        SomeMessage::load(r);
        r.load_value(s);
    }

    string s;

    static const char *name;
};

const char *AnotherMessage::name = "AnotherMessage";
```

Příklad B.3: Zpráva `AnotherMessage` odvozená od `SomeMessage`

jako odesílaný typ. Pro odpojování modulů slouží obdobná metoda masteru `disconnect<T, U>()`.

V současné verzi RCT existuje omezení které zabraňuje vytvořit modul, který odesílá nebo přijímá několik na sobě nezávislých zpráv stejného typu. To by mohl být problém například pokud by modul reprezentoval skupinu sériových portů. Pak by nebylo možné připojit každý sériový port nezávisle na ostatních. Tento problém lze vyřešit například vytvořením několika obalovacích typů pro zprávu.

```
Master master;
SerialPort serial("modul_Serioveho_portu");
GPS gps("modul_GPS");
MCL::Localization mcl("modul_MCL");

// ...
// jednoduche propojeni:
master.connect<SerialData>(&serial, &gps);

// propojeni s ruznym prijimanym a odesilanym typem:
master.connect<
    GPSFix,
    MCL::Correction<MCL::Sample2D>
    >(&gps, &mcl);
// ...
```

Příklad B.4: Propojování modulů

B.5 Vícevláknový běh

Pro řízení robota může být velmi užitečné, aby některé moduly běžely ve vlastních vláknech (např. vestavěný modul SerialPort čte ve vlastním vlákně data ze sériového portu – viz B.10).

Kritické části jádra RCT (logování, nahrávání, ...) jsou vytvořené s ohledem na bezpečnost při použití s vlákny.

Pokud modul používá vlákna, měl by nadefinovat metody `start()` a `stop()`, které spustí a zastaví všechna vlákna nutná pro činnost modulu. Tyto metody by měly ignorovat pokusy o opakované zastavení, nebo opakované spuštění vláken.

Voláním metod `start_threads()` a `stop_threads()` masteru se spustí a zastaví vlákna všech připojených modulů.

Moduly se nesmí přidávat k masteru ani odebírat, pokud mají některé z ostatních modulů připojených k tomuto masteru spuštěná svá vlákna.

B.6 Nahrávání

Nahrávání se spouští pomocí metody `Master::start_recording()`, parametry jsou název souboru a volitelný požadavek na kompresi souboru se záznamem. Zastavit nahrávání je možné metodou `Master::stop_recording()`, jestli nahrávání stále probíhá zjišťuje metoda `Master::is_recording()`.

Přehrávání záznamu je o něco složitější než jeho načítání. Spouští se metodou `Master::start_replaying()`. Ta otevře soubor (automaticky detekuje jestli je soubor komprimovaný) a zruší příznak přehrávání všech modulů, ale nečte ze souboru žádné zprávy. Potom musí uživatel nastavit příznaky přehrávání u těch modulů u kterých mají být výstupy nahrazeny načtenými zprávami.

Následně je nutné číst zaznamenané zprávy. To je možné pomocí metody `Master::continue_replaying()`, která přehrává všechny zprávy až do konce záznamu, nebo pomocí metody `Master::step_replaying()`, která přehraje jednu zprávu. Obě tyto metody mají volitelný parametr, který určuje, jestli se má před přečtením zprávy čekat po dobu zadanou v souboru nahrávky (výchozí nastavení je že se čekat má). Stav přehrávání lze zjistit metodou `Master::is_replaying()`, metoda `Master::stop_recording()` zastavuje probíhající přehrávání a zavírá soubor nahrávky.

Pokud se při přehrávání čeká, pak doba čekání je vždy vztažená k poslední přehrané zprávě (pokud například zpráva odešla 10 sekund po předchozí zprávě, ale program mezitím už 3 sekundy čekal, tak se bude před přečtením čekat 7 sekund navíc). Pokud doba určená k čekání uplynula před přečtení zprávy, nebude se čekat, ale následující zpráva bude opět časovaná podle poslední zprávy (pokud tři zprávy A, B a C byly odeslány v desetisekundových intervalech, a mezi přehráním zpráv A a B uplynulo 15 sekund pak mez přehráními B a C bude interval stále 10 sekund).

Spuštění nahrávání nebo přehrávání zastaví jakékoliv probíhající nahrávání nebo přehrávání.

B.6.1 Formát nahrávky

Nahrávka začíná komentářem s řetězcem `RCT recording` (magic string) a komentářem s datem a časem pořízení nahrávky. Po těchto dvou řádcích následují už jednotlivé zprávy.

První řádek zprávy je čas uplynutý od odeslání předchozí zprávy ve formátu hh:mm:ss.ssssss. Dále identifikátor odesílajícího modulu, jméno zprávy a data zprávy (uložená metodou `save()` zprávy). Záznam zprávy je ukončený separátorem – řádkem na kterém je pouze `\END`.

Ukázka nahrávky komunikace je v příkladu B.5.

Všechny položky kromě separátoru jsou uzavřené ve dvojitých uvozovkách. Mezi položkami záznamu mohou být komentáře. (Formát je odvozený z formátu pro konfigurační soubory. Pro detailnější popis viz sekce B.7.4.)

```
# RCT recording
# 2009-05-12 11:35:43
"00:00:00"
"port"
"SerialData"
"$GPGSA,A,3,04,13,02,20,23,,,,,,,,4.1,2.5,3.3*37\r\n"
\END

"00:00:00.113262"
"port"
"SerialData"
"$GPRMC,093543.000,A,5005.26"
\END
```

Příklad B.5: Nahrávka se dvěma zprávami

B.7 Konfigurační soubory

B.7.1 Načítání konfiguračních souborů

Nejjednodušším způsobem pro otevření konfiguračního souboru je použít metodu `Master::load_config_file2()`, která otevře konfigurační soubor a pokud v něm najde klíč `state_file`, pak otevře také soubor stavu. Pokud hodnota klíče `state_file` začíná lomítkem, pak je interpretována jako cesta relativní k cestě ke konfiguračnímu souboru, jinak se použije jako absolutní cesta. Jako poslední krok znovu nakonfiguruje všechny moduly. Pokud kterýkoliv krok

(otevírání souborů, hledání hodnoty) selže, pak tato metoda vrátí `false` a oba soubory budou zavřené.

Otevřít pouze konfigurační soubor bez otevírání souboru stavu je možné pomocí metody `Master::load_config_file()`, samotný soubor se stavem lze otevřít pomocí metody `Master::load_state_file()`.

Všechny tyto tři metody mají jako parametr název souboru který se má načíst a vrací `true` pokud se načítání zdařilo, jinak `false`.

B.7.2 Ukládání souboru stavu

Soubor stavu je možné uložit pomocí metody `Master::save_state_file()`. Nepovinným parametrem této metody je název souboru (pokud je název vynechaný, nebo je `NULL`, pak se použije poslední název souboru). Pokud se použitý konfigurační soubor pro čtení a zápis, pak se jeho obsah uloží na disk v destruktoru masteru.

B.7.3 Přístup k hodnotám

Metody `Master::get_config()`, `Module::get_config()` a `Module::get_cofig2()` slouží pro čtení z konfiguračního souboru. Tyto metody jsou šablonové, parametrem šablony je typ přečtené proměnné (provede se konverze z řetězcového zápisu v souboru na daný typ pomocí `boost::lexical_cast`). První parametr je řetězcový klíč pod kterým se má hodnota vyhledat. Pokud zadaný klíč není v souboru stavu, hledá se v konfiguračním souboru a pokud klíč není ani v tomto souboru, pak tyto metody vrací buď objekt zkonstruovaný bezparametrickým konstruktorem daného typu, nebo výchozí hodnotu zadanou volitelným druhým parametrem.

Pro přístup ke konfiguračním hodnotám z modulu je nutné, aby byl modul přidaný k masteru. `Master::get_config()` je základní varianta, vrací přímo hodnotu odpovídající zadanému klíči, mimo jmenné prostory. Metoda modulu `Module::get_config()` vrací hodnotu zadaného klíče ve jmenném prostoru modulu, `Module::get_config2()` vrací hodnotu klíče mimo jmenné prostory, tedy stejnou jako metoda `get_config()` odpovídajícího masteru. Příklad použití těchto metod je v příkladu B.6.

Obdobně jako metody pro čtení hodnot fungují metody `Master::set_config()`, `Module::set_config()` a `Module::set_config2()`. Jejich parametry jsou klíč a hodnota, typ hodnoty je parametrem šablony. Zapisuje se vždy pouze do souboru stavu.

Je také možné bez čtení jeho hodnoty ověřit, jestli je klíč reprezentovaný v souboru stavu nebo v konfiguračním souboru. K tomu slouží metody `Master::check_config()`, `Module::check_config()` a `Module::check_config2()`. Tyto metody vyhledávají klíč obdobně jako metody pro čtení.

```
Master master();
SomeModule module("ID");

master.load_config_file2("/etc/robot.cfg");

master.add(&module);

// konfiguracni parametr s vychozi hodnotou:
double a = module.get_config<double>("klic", -1.0);

// zapsani hodnoty
module.set_config("klic", 2.71828);

// zkontrolovani hodnoty
bool b = module.check_config("klic");

// nasledujici volani jsou ekvivalentni:
double c = module.get_config<double>("klic");
double d = module.get_config2<double>("ID.klic");
double e = master.get_config<double>("ID.klic");
```

Příklad B.6: Načítání konfiguračních souborů

B.7.4 Formát konfiguračních souborů

Formát konfiguračních souborů je textový, obsahuje dvojice klíč-hodnota a komentáře.

Klíč je od hodnoty oddělený dvojtečkou, dvojice je ukončená znakem newline.

Klíč a hodnota mohou obsahovat části ve dvojitých uvozovkách. V nich ztrácí dvojtečka a newline svůj význam. Dalším významným znakem je zpětné lomítko, které odstraní speciální význam následujícího speciálního znaku (včetně newline). Pokud za zpětným lomítkem následuje n, r, t, nebo x a hexadecimální číslo, pak to znamená newline, carriage return, tabulátor, nebo znak se zadaným kódem.

Poznámky začínají znakem `#` a pokračují do konce řádky. Prázdné řádky, nebo řádky na kterých je pouze poznámka se ignorují.

Bílé znaky před a za klíčem nebo hodnotou se ignorují, pokud nejsou uzavřené v uvozovkách, nebo pokud před nimi není zpětné lomítko (tedy `" "abc \ ` je stejné jako `" abc "`).

Uložené hodnoty (v souboru stavu a v nahrávce) jsou celé uzavřené v uvozovkách a mají všechny výskyty znaků newline, tabulátoru a carriage return nahrazeny odpovídající escape sekvencí. Navíc všechny netisknutelné znaky jsou nahrazené za `\x`*kód*.

Ukázka možných syntaxí v konfiguračním souboru je v příkladu B.7.

```
klic: hodnota
klic2: viceradkova \
hodnota                                  # komentar
# prazdne radky

mezery v klici: mezery v hodnote
"dvojtecka:v klici": hodnota
"viceradkovy
klic": escape sekvence \x123 \t\n\r\x40
mezery: "na konci hodnoty             "
```

Příklad B.7: Ukázka možné syntaxe v konfiguračním souboru

B.7.5 Konfigurace modulů

Virtuální metoda `Module::configure()` slouží pro načtení konfigurace z konfiguračních souborů. Tato metoda je automaticky zavolána po přidání modulu k masteru, a v metodě `Master::load_config_file2()`, ale může být ručně zavolána kdykoliv pro načtení parametrů.

B.8 Logování

Pro logování slouží makra `RCT_LOG` a `RCT_LOG_MODULE`. Parametry `RCT_LOG` jsou reference na master, úroveň zprávy (viz níže) a výraz který se má zaznamenat. `RCT_LOG_MODULE` funguje pouze v metodách modulu a k textu připojuje i identifikátor modulu. Namísto reference na master používá ukazatel `this` a metodu `Module::get_current_master()`. Zaznamenaný výraz obsahuje data, která se mají pomocí operátoru `<<` zapsat do streamu (viz příklad B.8). Pokud je úroveň závažnosti nižší než je nastavená hranice, pak se výraz k logování nevyhodnocuje.

Úrovně zpráv určují jak je zpráva závažná a jestli se má zaznamenávat. Úrovně (vzestupně podle závažnosti) jsou:

- `LogLevel::debug` pro ladící výpisy
- `LogLevel::info` pro informační zprávy
- `LogLevel::notice` pro normální, ale důležité zprávy
- `LogLevel::warning` pro varování
- `LogLevel::err` pro chyby
- `LogLevel::crit` pro kritické chyby
- `LogLevel::alert` pro chyby které musí být okamžitě vyřešeny
- `LogLevel::emerg` pro nejvážnější chyby

Hranice pro zaznamenávání se buď nastavuje pro master a všechny jeho připojené moduly společně pomocí `Master::set_log_threshold_all()`, nebo je ji možné nastavit pro každý modul metodou `Module::set_log_threshold()` a

pro master metodou `Master::set_log_threshold()`. Parametrem těchto metod je nejnižší úroveň zprávy, která se ještě bude zaznamenávat. Základní nastavení hranice je `LogLevel::notice`. Hranice zaznamenávání modulu se při přidání k masteru nastaví na stejnou hodnotu jakou má aktuálně master.

Logované zprávy které mají dostatečnou úroveň se posílají na všechny otevřené výstupy. Při inicializaci masteru se otvírá logování na standardní chybový výstup.

Otevřít logovací výstup lze pomocí metody `Master::open_log_cil()` kde *cil* je *stderr* pro standardní chybový výstup, *file* pro soubor, nebo *syslog*. Zavřít výstup je možné pomocí `Master::close_log_cil()`.

```
class AnotherModule : public Module{
// ...
};

void AnotherModule::method(){
    RTC_LOG_MODULE(LogLevel::emerg, "foo" << 321);
}

// ...

Master master();
AnotherModule module("ID")


// veskere zpavy pujdou pouze na syslog a do souboru
master.close_log_stderr();
master.open_log_syslog();
master.open_log_file("soubor.log");

// zaznamenavat pouze varovani a dulezitejsi,
// ale od modulu module zaznamenavat vse
master.set_log_threshold_all(LogLevel::warning);
module.set_log_threshold(LogLevel::debug);

RCT_LOG(master, LogLevel::err, "foo" << 123 << "bar");
```

Příklad B.8: Logování

B.9 Master

Kromě funkcí popsaných výše má master ještě dvě zajímavé metody. První z nich je `get_module_by_name()`, která vyhledává v připojených modulech a vrací ukazatel na modul podle zadaného jména.

`dump_connection_graph()`. Parametrem této metody je reference na ostream. Do tohoto streamu vypíše graf propojení modulů ve formátu pro graphviz.

B.10 Modul `SerialPort`

Modul `SerialPort` slouží pro jednoduchou komunikaci po sériové lince, například jako zdroj dat pro hardwarové rozhraní, pro GPS, atd.

`SerialPort` odesílá a přijímá zprávy typu `SerialData`. Data přečtená z portu se odesílají v závislosti na nastavení parametru `eol_char`. Pokud je tento parametr záporný, pak se pro každý znak přečtený z portu pošle nová zpráva. Jinak se hodnota tohoto parametru interpretuje znak ukončující řádky a ve zprávách se odesílají celé přijaté řádky.

Data přijatá ve zprávách se zapisují do otevřeného portu.

B.10.1 Zprávy typu `SerialData`

Zprávy typu `SerialData` reprezentují data ze sériového portu. Hodnota je uložená ve veřejné proměnné třídy `data` typu `std::string`.

B.10.2 Konfigurace portu

- `port` Povinný parametr. Určuje cestu k portu.
- `params` Povinný parametr. Nastavení portu ve tvaru <přenosová rychlost> <délka slova><parita><počet stop bitů>. Například `9600 8N1`.
- `eol_char` Nepovinný parametr. Nastavuje znak konce řádky. Výchozí hodnota je -1, tedy odesílání každého přijatého znaku zvlášť.

B.11 Lokalizace

Modul `MCL::Localize<T>` řeší lokalizaci metodou MCL.

Celá lokalizace je postavena kolem typu vzorku (ten je parametrem šablony modulu). Vzorek popisuje jednu možnou pozici robota, v závislosti na tomto parametru může modul lokalizovat robota v libovolně složitých stavových prostorech.

Výstupem tohoto modulu je zpráva s odhadem pozice, reprezentovaná třídou `MCL::PositionEstimate<T>`. Jejím parametrem šablony je typ vzorku a pomocí specializace šablon se vytváří odhady pro různé typy vzorků.

Vstupem lokalizace jsou dva druhy zpráv, první z nich reprezentuje predikční data vstupující do lokalizace a druhá korekční data. V tomto místě se využije schopnost RCT používat u zpráv polymorfismus. Lokalizace definuje dvě abstraktní třídy (opět závisející na typu vzorku) pro zprávy sloužící jako vstup do lokalizace.

`MCL::Prediction<T>` je předkem zpráv obsahujících predikční data a má čistě virtuální metodu `predict()`, která dostane jako parametr vzorek upraví jej podle nových dat.

`MCL::Correction<T>` slouží pro zprávy obsahující korekční data a její metoda `correct()` dostane jako vstup vzorek spolu s jeho váhou a vrací novou upravenou váhu.

Zprávy, které slouží jako vstup do lokalizace jsou odvozené od těchto tříd namísto od `Message`.

Obě třídy pro vstupy do lokalizace mají metodu `prepare()`, která se zavolá před tím než se zpráva použije pro úpravu vzorků. Tato metoda umožňuje předpočítat mezivýsledky společné pro jednu iteraci lokalizačního algoritmu. Například zpráva s GPS odhadem v této metodě počítá konstanty závislé na odhadu horizontální chyby, tyto hodnoty se pak používají pro všechny vzorky.

Všechny tři metody u zpráv vstupujících do lokalizace mají jako parametr ještě ukazatel na aktuální mapu používanou lokalizací.

Tento modul přijímá zprávy typu `MCL::Prediction<T>` a `MCL::Correction<T>`. Tyto zprávy slouží jako předci pro predikční a korekční vstupy lokalizace, jejich parametrem šablony je typ vzorku.

B.11.1 Mapa

Mapa pro MCL je definovaná pouze rozhraním a je opět závislá na typu vzorku.

Mapa musí umožňovat vygenerovat náhodný vzorek pro novou inicializaci globální lokalizace.

Pro dálkoměry je v rozhraní mapy metoda pro změření vzdálenosti ke stěně. Tato metoda bere jako parametr bod a směr a vrací vzdálenost k nejbližší stěně v zadaném směru.

Pro použití GPS je v rozhraní mapy metoda pro získání GPS souřadnic bodu, který má v mapě souřadnice $(0, 0)$. Části RCT využívající GPS pak ještě předpokládají, že sever je na mapě v kladném směru osy Y a východ v kladném směru osy X.

B.11.2 Konfigurace

- `samples` Nepovinný parametr. Určuje Počet vzorků použitých při lokalizaci. Výchozí hodnota je 500.
- `resampling_threshold` Nepovinný parametr. Nastavuje hranici efektivního počtu vzorků pod kterou se provede převzorkování. Výchozí nastavení je 0,75.

B.12 GPS

Modul `GPS` je jednoduché rozhraní pro GPS používající NMEA protokol. Modul přijímá zprávy typu `SerialData` a odesílá zprávy `GPSFix`, které jsou odvozené od `MCL::Correction`.

V současné verzi podporuje tento modul pouze zprávy typu *GPGGA*, protože tato zpráva dostačuje pro lokalizaci ve dvourozměrném prostoru.

B.13 Player klient

RCT umožňuje využívat Player klienta jako vstupy a výstupy z lokalizace.

K tomu se používají moduly, které jsou obalovacími třídami Proxy tříd z knihovny libplayerc++.

Tyto moduly lze vytvářet odvozováním od `detail::PlayerClientWrapper<T>`. Parametrem šablony této třídy je typ proxy třídy z libplayerc++, například `PlayerCc::Position2dProxy`. `detail::PlayerClientWrapper<T>` obsahuje proxy třídu playeru a zajišťuje spouštění a zastavování vláken a zjišťování nových dat. Čistě virtuální metoda `new_proxy_data()` se volá pokaždé když proxy nahlásí nová data.

Parametry konstruktoru této třídy jsou podobné jako konstruktor proxy třídy, pouze místo ukazatele na `PlayerCc::PlayerClient` je použitý ukazatel na jeho obalovací třídu `PlayerClientWrapper`.

B.14 Pomocné nástroje

B.14.1 Náhodné proměnné

Ve jmenných prostorech `UniformDistribution` a `NormalDistribution` se nacházejí funkce umožňující pracovat s náhodnými hodnotami z rovnoměrného a normálního rozdělení.

Rovnoměrné rozložení

`UniformDistribution::value()` má dva parametry a vrací hodnotu z rovnoměrného rozdělení. Hodnota je mezi zadanými parametry. Tato funkce má také variantu bez parametrů, která vrací hodnoty mezi 0 a 1.

Jmenný prostor `NormalDistribution` obsahuje funkce pro práci s normálním rozdělením. funkce `value()` vrací hodnotu se zadanou střední hodnotou a standardní odchylkou. Opět existuje varianta bez parametrů, ta vrací hodnoty se střední hodnotou 0 a standardní odchylkou 1. Kvůli tomu, že generování normálně rozložených náhodných čísel používá Box-Mullerovu transformaci, která generuje dvojice hodnot, existuje i funkce `value2()` pro získání této dvojice. Tato funkce má za parametry střední hodnoty a standardní odchylky požadovaných hodnot (pro každou zvlášť) a reference na dvě proměnné pro uložení výsledku. Je výhodnější používat `value2()`, než dvakrát volat `value()`.

Funkce `NormalDistribution::density()` vrací hodnotu hustoty normálního rozdělení v zadaném bodě.

B.14.2 Diferenciální pohon

RCT obsahuje funkce pro přepočty rychlostí pro diferenciální pohon. Ty se nacházejí ve jmenném prostoru `DifferentialDrive`. Pro přepočítávání dopředné rychlosti a úhlové rychlosti zatáčení na rychlosti levého a pravého kola slouží funkce `turn2diff`. Jejími parametry jsou rozchod kol, dopředná rychlost, úhlová rychlost zatáčení (kladné hodnoty značí zatáčení vlevo) a reference na vypočítané hodnoty. Pro opačný převod slouží funkce `diff2turn()`.