FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	ETCS - Aktualizace a nová architektura komponenty RBC
Student:	Ondřej Veselý
Vedoucí:	Ing. Jan Matoušek
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

ETCS (European Train Control System) je jednotný celoevropský vlakový zabezpečovací systém. V rámci ČVUT se ve spolupráci dvou fakult (Fakulta informačních technologií a Fakulta dopravní) vyvíjí simulátor tohoto systému.

RBC (Radio Block Centre) je klíčovou součástí ETCS. Je to centrální prvek systému, který má na starost řízení a monitorování pohybu vlaků.
Práce je součástí vetšího projektu simulátoru ETCS na ČVUT.

Cílem této práce je návrh nové architektury komponenty RBC simulátoru s ohledem na požadavky celého projektu a následné přepsání komponenty RBC simulátoru do této nové architektury.
Tato architektura bude navržena tak, aby ji mohly využít i ostatní komponenty simulátoru ETCS.
Dalším aspektem práce je vypracování komplexnějších scénářů jízdy, ve kterých lze zadat pokyny k nouzovému brzdění a zastavení vlaku.
S tím souvisí i podpora procedury post-trip, která následuje po nouzovém brzdění.

Pokyny pro vypracování
1) Analyzujte současný stav projektu (komponenta RBC a konfigurace scénářů jízdy).
2) Navrhněte novou architekturu, která bude více odpovídat požadavkům projektu.
3) Navrhněte nové scénáře jízdy, které pokrývají širší množinu požadovaných funkcionalit.
4) Reimplementujte všechny funkcionality stávající komponenty RBC simulátoru ETCS.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 3. února 2024 v Praze.

5) Implementujte nové scénáře jízdy a podporu nouzového brždení.
6) Vypracujte sady testů pro testování nové implementace komponenty RBC simulátoru ETCS.
7) Zdokumentujte novou architekturu a nové scénáře.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 3. února 2024 v Praze.

Bakalářská práce

ETCS – AKTUALIZACE A NOVÁ ARCHITEKTURA KOMPONENTY RBC

Ondřej Veselý

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jan Matoušek
16. května 2024

Odkaz na tuto práci: Veselý Ondřej. *ETCS – Aktualizace a nová architektura komponenty RBC*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych především poděkovat mému skvělému vedoucímu Ing. Janu Matouškovi za jeho trpělivost a vřelý přístup k celému projektu. Dále děkuji panu doc. Ing. Martinu Lesovi, Ph.D a panu Bc. Vítu Řezáčovi za jejich odborné rady a vřelé konzultace okolo tématu. V neposlední řadě bych rád poděkoval mým kolegům z týmu projektu ETCS hlavně tedy Martinu Čáslavskému a všem dalším, co se na tomto veledíle podíleli.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

Abstrakt

Tato bakalářská práce se zaměřuje na analýzu a reimplementaci klíčové komponenty RBC ze systému ETCS. ETCS je evropský vlakový zabezpečovací systém a tato práce je součástí simulátoru, který je vyvíjen ve spolupráci s Fakultou dopravní. Hlavní cíl je rozšiřitelnost a kvalita kódu. Je kladen důraz na analýzu stávající implementace a dodržení specifikací ERA-ERTMS. Výsledkem je kompletně přepsaná komponenta RBC v jazyce c++, sjednocená architektura mezi již stávajícími ostatními komponentami ETCS a definované společné zprávy v repositáři CEM. Tato práce byla klíčová pro posunutí projektu ETCS vpřed vzhledem k mnoha nesrovnalostem mezi jednotlivými komponentami a potřebě jejich vzájemné spolupráce. Díky této práci se komponenty staly mnohem rozšiřitelnějšími, a co je důležitější, běží na stejné architektuře, takže pochopení jejich fungování a následné rozšiřování je mnohem snazší.

Klíčová slova ETCS, reimplementace RBC, simulátor ETCS, vlakový zabezpečovač, C++

Abstract

This bachelor thesis focuses on the analysis and reimplementation of a key component of the RBC from the ETCS system. ETCS is the European Train Control System and this thesis is part of a simulator that is being developed in collaboration with the Faculty of Transport. The main objective is extensibility and code quality. The focus is on the analysis of the existing implementation and compliance with ERA-ERTMS specifications. The result is a completely rewritten RBC component in c++ language, a unified architecture between already existing other ETCS components and defined common messages in the CEM repository. This work was crucial to move the ETCS project forward due to the many inconsistencies between the components and the need for them to work together. This work has made the components much more extensible, and more importantly, they run on the same architecture, so understanding how they work and then extending them is much easier.

Keywords ETCS, reimplementation of RBC, ETCS simulator, train protection system, C++

Seznam zkratek

BTM Balise Transmission Module

CEM Common Enums and Messages

DMI Driver Machine Interface

EDA Event Driven Architecture
EoA End of Authority
EoM End of mission
ERTMS European Railway Traffic Management System
ETCS European Train Control System
EVC European Vital Computer

FS Full Supervision

GP Gradient Profile
GSM-R Global System for Mobile Communications – Railway

JRU Juridical Recording Unit

LPC Lektroské Pracoviště
LRBG Last relevant balise group

MA Movement Authority
MQTT Mosquitto

ODO Odometrie
OS On Sight

PT Post Trip

RBC Radio Block Center

SOA Service Oriented Architecture
SoM Start of mission
SR Staff Responsible
SSP Static Speed Profile
STM Specific Transmission Module

TAF Track Ahead Free
TIU Train Interface Unit
TR Trip

Úvod

V této kapitole bych rád uvedl systém ETCS, jeho simulátor a také mojí práci (jak práci na projektu tak i práci bakalářskou).

Úvod do tématu

European Train Control System (ETCS) je standardizovaný systém pro zabezpečení a řízení vlakové dopravy v Evropě. Jeho cílem je zvýšit bezpečnost a efektivitu železničního provozu tím, že umožňuje nepřetržitou komunikaci mezi vlaky a pozemními kontrolními centry. Pokud vlak překročí povolenou rychlost, nebo pokud existuje riziko kolize, má systém možnost automaticky zasáhnout. Buďto tím, že nařídí vlaku zpomalit nebo i zastavit. Je zároveň součástí širšího projektu European Railway Traffic Management System (ERTMS), který cílí nahradit různé národní systémy řízení a zabezpečení vlaků v Evropě.[1]

Historie projektu

Před několika lety vznikl na Fakultě dopravní projekt, který měl za cíl vytvořit trenažér vlakové dopravy se simulátorem ETCS. Na tento simulátor byli přizváni kolegové z Fakulty informačních technologií a vznikl tedy projekt simulátoru ETCS. Tento projekt je vyvíjen studenty v rámci předmětů SP1, SP2 a každý rok se předává novému týmu.

Moje práce na projektu

Já se k projektu dostal v rámci předmětu SP1 v zimním semestru roku 2023, kdy jsme s týmem do něj byli vrženi jako do jedoucího vlaku. V průběhu prvotní analýzy jsme jako tým zjistili, že stav komponenty RBC není vhodný pro další rozšíření a údržbu. Kód byl velmi nepřehledný, těžko rozšiřitelný a také plný chyb. Toto bylo částečně způsobeno nekorektním předáváním mezi týmy a také téměř neexistující dokumentací, minimem testů a nerozmyšleným návrhem. Po několika týdenním boji a konzultacemi, jak s týmem, tak s vedoucím projektu Ing. Janem Matouškem, bylo rozhodnuto, že je nutný kompletně nový návrh. Proto jsem část SP2 věnoval přípravám na tento rozsáhlý přepis a uskutečnil ho v rámci této bakalářské práce.

Cíle práce

Hlavním cílem práce je návrh nové architektury a přepis RBC do této nově navržené architektury. Klíčová pro tento návrh je jednoduchá rozšiřitelnost a hlavně kompatibilita s ostatními komponentami. Kromě toho je nutné zachovat všechny stávající funkcionality a hlavně rozšířit RBC o nové. Tyto nové funkcionality se týkají hlavně konfigurací, scénářů jízdy a možností lepší kontroly ze strany lektorského pracoviště. V neposlední řadě je důležitá i kvalita, čitelnost a přehlednost kódu. Tyto aspekty usnadní členům budoucích týmů se zorientovat a začít pracovat na projektu.

Kapitola 1
Analýza

Nejdůležitější částí jakéhokoliv projektu je důkladná analýza. I tento projekt není výjimkou a proto se zde zaměřím na analýzu dosavadního stavu projektu, zejména komponenty RBC. Toto mi poskytne nezbytný přehled o výchozím bodu vývoje, který je kritický pro následný návrh a implementaci. Samozřejmě je také třeba porovnat jednotlivé funkce systému s oficiální specifikaci ETCS, podle které se celý simulátor vyvíjí. Hlouběji se ponořím do funkcionalit, které jsou důležité pro porozumění, jak RBC funguje v tomto komplikovaném systému.

1.1 Analýza oficiální specifikace ETCS

V této části vysvětlím, co je systém ETCS, jaké jsou jeho nejdůležitější části a jak probíhá komunikace mezi nimi. Rád bych uvedl, že analyzuji specifikaci verze 2.3.0.

1.1.1 Popis ETCS a jeho částí

Hlavní výhodou ETCS je interoperabilita. ETCS je standardizovaný po celé Evropě, což umožňuje výměnu dat a spolupráci mezi různými železničními sítěmi a zeměmi. Toto značně usnadňuje mezinárodní vlakovou dopravu.[2]

Další důležitou výhodou je zvýšená bezpečnost. ETCS má spoustu pokročilých funkcí pro řízení rychlosti, monitorování polohy a automatické brzdění, což významně přispívá k bezpečnosti provozu vlaků. Díky všem těmto funkcím se minimalizuje riziko chyb, odchylek a dalších nehod.

Dále díky systému ETCS lze dosáhnout vyšší efektivity a kapacity. Systém může monitorovat a řídit vlaky s vysokou přesností a efektivitou. Tyto výhody vedou k lepšímu využití možností železniční sítě.

Kromě toho bych také vyzdvihl ještě jednu podstatnou výhodu. To je jednoduchost aktualizace a snížení nákladů na provoz. ETCS je digitální systém, který je navržen tak, aby byl přechod mezi novějšími verzemi co nejméně komplikovaný a hlavně bez potřeby změny fyzické infrastruktury. Zároveň díky možnosti automatizace tyto dvě vlastnosti přináší úspory, jak provozovatelům, tak cestujícím.

Systém ETCS se dělí do několika komponent, které spolu interagují. Tyto komponenty se následně dělí podle toho, kde se nacházejí. [3, kap. 2.4] Na obrázku 1.1 popisuji, která komponenta komunikuje s jakou a šipky značí směr jejích komunikace. Zaoblené jsou prvky na trati (vlak a balízy) a hranaté jsou komponenty systému ETCS.

1.1.1.1 Traťová část

Do traťové části (na obrázku 1.1 modře) se počítají:

GSM-R: Mezinárodní bezdrátový komunikační standard pro železniční aplikaci. Slouží pro oboustrannou komunikaci mezi vlakem a RBC. [3, kap. 2.5.1.4]

RBC: Komponenta traťové části ETCS. Má na starosti řízení a monitorování vlaků (v úrovni L2 a L3). Komunikuje prostřednictvím GSM-R a poskytuje povolení k jízdě, informace o trati a další důležité bezpečnostní informace. [3, kap. 2.5.1.5]

Balízy: Bodové prvky umístěné v kolejích. Automaticky předávají data vlakům, které nad nimi projíždí. [3, kap. 2.5.1.2] Tyto transpondery[1] mohou být použity pro přesné určení polohy vlaků a poskytování dat o stavu trati a rychlostních omezení. [4, kap. 3.4.3] Seskupují se do skupin nazývané „Balise Groups“. Tyto balízové skupiny jsou tvořeny 1-8 balízami a každá skupina má vlastní unikátní identifikátor. [4, kap. 3.4.1]

1.1.1.2 Vozidlová část

Do vozidlové části (na obrázku 1.1 oranžově) se počítají:

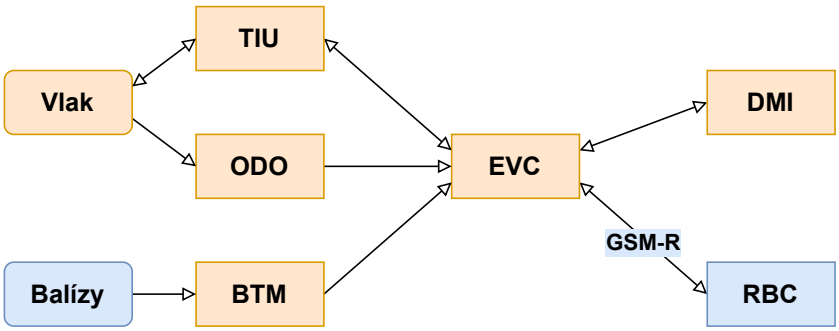
EVC: Hlavní část vozidlové části. Jeho úkolem je bezpečný dohled nad pohybem vozidla, který řídí strojvedoucí. V případě překročení mezních podmínek (rychlost, směr jízdy nebo místo povolení jizdy) zasáhne do jízdy vozidla omezením pohybu (snížení rychlosti nebo zastavením). [5]

DMI: Uživatelské rozhraní systému ETCS. Slouží jako primární komunikační kanál mezi strojvedoucím a systémem ETCS. Poskytuje vizuální a zvukové informace potřebné pro řízení vlaku. [5]

TIU: Poskytuje rozhraní mezi palubním vybavením vlaku (brzdy, trakční systémy a další řídící systémy vlaku) a systémem ETCS. Dále zajišťuje, že tyto příkazy budou spolehlivě přeneseny mezi těmito subsystémy. [5]

BTM: Umožňuje přenos informací z balíz do vlaku. BTM je nainstalováno na palubě vlaku, čte informace z balíz a posílá je systému ETCS ke zpracování. [5]

ODO: Systém na palubě vlaku, který měří jízdní informace o vlaku. ODO následně tyto informace periodicky posílá EVC ke zpracování. [5]



Obrázek 1.1 Popis interakcí komponent v systému ETCS

[1] Zařízení, které po přijetí signálu vysílá v reakci na něj jiný signál.

1.1.2 Komunikace mezi traťovou a vozidlovou částí

Komunikace mezi RBC a EVC je úzce určená ERTMS specifikací. Traťová část s vozidlovou spolu komunikují pomocí zpráv. Tyto zprávy mohou být rozšířeny o takzvané *pakety*, které slouží k připnutí dodatečných informací ke zprávě.

Každá zpráva/paket obsahuje pouze pevně specifikované proměnné. [6, kap. 8.6] Tyto proměnné mají svou vlastní velikost a uvedené hodnoty, kterých mohou nabývat. [7, kap. 7.5] Díky tomu manipulace s proměnnou je jednodušší a dobře definovaná.

Zprávy mají jednoznačný identifikátor *NID_MESSAGE*. Tato proměnná nabývá hodnot 1-99 pro zprávy od trati ke vlaku [6, kap. 8.5.3] a 100-199 pro zprávy od vlaku k trati. [6, kap. 8.5.2] Dále zprávy obsahují proměnnou *L_MESSAGE*, která nese délku zprávy v bytech. Tyto dvě proměnné se vyskytují v každé zprávě, ostatní proměnné se liší podle typu zprávy.

Pakety mají jednoznačný identifikátor *NID_PACKET*. Mají i proměnou *L_PACKET*, která — podobně jako je to u zpráv — informuje o bytové délce paketu. Existují zde i určitá pravidla, která diktují, jaký paket může být připojen k jaké zprávě. [6, kap. 8.4.4]

Zpráv, paketů a proměnných je příliš mnoho na to abych je zde všechny vypisoval. Pro jejich kompletní přehled zde poskytuji odkaz do oficiální specifikace. Několik ukázek uvádím dále u sekcí, které se jim věnují.

Zprávy: Subset-026-8 [6, kap. 8.6]

Pakety: Subset-026-7 [7, kap. 7.4]

Proměnné: Subset-026-7 [7, kap. 7.5]

1.1.3 Úrovně ETCS

Definice úrovní se v zásadě týkají použitého traťového zařízení, způsobu jakým se traťové informace dostávají k palubním jednotkám a toho, které funkce se zpracovávají v traťovém a které v palubním zařízení. [3, kap. 2.6.1]

Vlak vybavený palubním zařízením vždy spolupracuje s traťovým zařízením v definované úrovni ETCS. [3, kap. 2.6.2] Systém ETCS je realizován v těchto několika úrovních:

ETCS L0: Monitoruje maximální konstrukční rychlost vlaku a národní maximum (v ČR je to 100 km/h). Většinou se používá na úsecích kolejiště, kde není traťová část nainstalována. [3, kap. 2.6.3]

ETCS LSTM: Používá se v situaci, kdy vlak se systémem ETCS jede po trati, kde je jen nějaký národní zabezpečovací systém. ETCS s tímto systémem komunikuje pomocí dodatečného národně specifikovaného modulu STM. [3, kap. 2.6.4]

ETCS L1: Povolení k jízdě jsou generovány na trati a jsou komunikovány pouze prostřednictvím balíz. Vlaková část přebírá popis trati od balíz a pomoci něho určuje maximální povolenou rychlost. Pokud strojvedoucí nedodrží rychlostní omezení, může být vyslán příkaz k aplikování brzd. [3, kap. 2.6.5]

ETCS L2: V této úrovní většinu komunikace z trati zajišťuje RBC a síť GSM-R. Balízy slouží především už jen k určení polohy (mohou být využity k předání informací, jak kontaktovat RBC a dalších podmínkách při vstupu do oblasti). Každé RBC má na starosti určenou oblast, ve které spravuje vlaky. Vlaková část periodicky hlásí svojí polohu odpovědnému RBC. To monitoruje jejich pohyb a následně jim uděluje povolení k jízdě. [3, kap. 2.6.6]

ETCS L3: Zahrnuje všechny funkce z předešlých úrovní a navíc detekci volnosti kolejových úseků pouze na základě informací dostupných z vlaku. To umožňuje posouvání hranice vydání autority i na základě informace o poloze konce předešlého vlaku. Toto je koncept tzv. „moving block“, který by mohl dovolit vyšší kapacity na trati. [3, kap. 2.6.7]

1.1.4 Módy

Systém ETCS se vždy nachází v nějakém operačním módu. Tento mód určuje jaké možnosti systém má a co dovoluje strojvedoucímu. Zde jsou vypsány všechny módy ve kterých se ETCS může vyskytovat: [8, kap. 4.4]

IS (Isolation): V tomto módu je celý systém ETCS odpojen od vlaku. Nemůže nijak ovládat brzdy a na opuštění tohoto módu je potřeba speciální procedura (jelikož neexistuje přechod).

NP (No Power): Kdykoliv není zapnuté napájení vozidlové části ETCS. Při přechodu do tohoto módu dojde k vypuštění vzduchu z brzd a vlak je tudíž zabrzděn.

SF (System Failure): Do tohoto módu se ETCS systém přepne, pokud nastane nějaká chyba, která by mohla ovlivnit bezpečnost vlaku. Opět při přechodu do tohoto módu dojde k vypuštění brzd a tudíž k zabrzdění vlaku.

SL (Sleeping): Tento mód slouží jako „pasivní“ mód ETCS.

SB (Stand By): Výchozí mód ETCS.

SH (Shunting): Účel tohoto módu je aby umožnil možnost posunu vlaku. Je hlídána pouze maximální rychlost (40 km/h).

FS (Full Supervision): Přechod do tohoto módu si určuje EVC, kdy RBC vlaku dodává příslušná povolení a data. V tomto módu se vlak pohybuje s maximálním dohledem a strojvedoucí nemusí dbát vyšší opatrnosti a proto může „pouze“ udávat rychlost.

UN (Unfitted): Kdykoliv se vlak se systémem ETCS pohybuje po trati, kde není nainstalována traťová část ETCS nebo pokud není dostupný STM na komunikaci mezi místním zabezpečovačen. ETCS v tomto módu hlídá pouze maximální rychlost.

SR (Staff Responsible): Tento mód umožňuje strojvedoucímu řídit vlak na jeho vlastní zodpovědnost v oblasti s plně vybaveným ETCS systémem. Tento mód se používá, když nejsou dostupné všechny údaje (pozice vlaku, plán cesty atd.). Opět je hlídána pouze maximální rychlost (typicky 40km/h).

OS (On Sight): V tomto módu je umožněný vjezd do oblasti o které není jisté jestli není něčím blokována. Přechod do tohoto módu je možný, jen pokud je to nařízeno z traťové strany systému ETCS. Vlaková část opět hlídá pouze maximální rychlost (typicky 40km/h).

TR (Trip): Do tohoto módu se vlak dostane téměř vždy když je potřeba nějakého nouzového brzdění. Při přechodu do módu Trip systém ETCS vždy zavede nouzové brzdy. Pro odbrzdění je potřeba přechod do módu PT.

PT (Post Trip): Je mód do kterého se ETCS dostane po tom co strojvedoucí potvrdí TRIP a po úplném zastavení vlaku. Až zde dochází k odbrzdění vlaku a následnému řešení situace.

NL (Non Leading): Se používá pro situaci, kdy máme soupravu s více lokomotivami vybavenými systémem ETCS, ale nepropojené mezi sebou. Systém v tomto módu neprovádí žádné omezující manévry, pouze monitoruje rychlost, pozici a další jízdní data.

SE (STM European): Speciální mód který se používá jen na úrovni LSTM. V tomto módu je ETCS systém schopný pouze základních funkci související s omezováním rychlosti a brzděním.

SN (STM National): Tento mód je podobný módu SE, jen zde není žádná možnost supervize.

RV (Reversing): Je mód který se používá výhradně k couvání. Tento manévr je umožněn pouze v oblastech tomu předem určených. Používá se hlavně při nouzových situacích, aby se vlak dostal do „bezpečnější“ pozice.

1.1.4.1 Přechody mezi módy

Mezi módy lze přecházet, pokud je splněna alespoň jedna z podmínek. Tyto přechody definuje tabulka 1.1. Každé číslo reprezentuje jednu podmínku[2], která musí být pro přechod splněna. Dále má každý přechod svojí prioritu. Ta se využije tehdy, pokud je splněno více podmínek najednou, poté platí přechod s nižším číslem priority.

NP	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-	<29 -p2-		<29 -p2-	<29 -p2-	<29 -p2-
4> -p2-	SB	<19, 28 -p5-	<28 -p5-	<28, -p5-	<28, -p5-	<2, 3 -p4-	<28, 47 -p3-	<28, -p6-					<28 -p6-	<28 -p6-	<28 -p4-
	5,6, 50> -p7-	SH	<5,6, 50,51 -p6-	<5,6, 51 -p6-	<5,6 50,51 -p6-			<5,61 -p7-		<5,6, 50 -p4-			<61 -p7	<61 -p7	
	10> -p7-		FS	<31,32 -p6-	<31,32 -p6-			<25 -p7-		<31 -p4-			<25 -p7-	<25 -p7-	
	8,37> -p7-		37> -p6-	SR	<37 -p6-			<44 -p4-		<8,37 -p4-			<44 -p4-	<44 -p4-	
	15> -p7-		15,40> -p6-	40> -p6-	OS			<34 -p7-		<15 -p4-			<34 -p7-	<34 -p7-	
	14> -p5-					SL									
	46> -p6-	46> -p5-	46> -p6-	46> -p6-	46> -p6-		NL								
	60> -p7-		21> -p6-	21> -p6-	21> -p6-			UN	<62 -p3-				<21 -p7-	<21 -p7-	
	20> -p4-	49,52, 65> -p4-	12,16, 17,18, 20,41, 65,66> -p4-	18,20, 42,43, 36,54, 65> -p4-	12,16, 17,18, 20,41, 65,66> -p4-			67,39> -p5-	TR				<67,39 -p5-	<67,39 -p5-	
									7> -p4-	PT					
	13> -p3-	13> -p3-	13> -p3-	13> -p3-	13> -p3-			13> -p3-	13> -p3-	13> -p3-	SF		<13 -p3-	<13 -p3-	<13 -p3-
1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	1> -p1-	IS	<1 -p1-	<1 -p1-	<1 -p1-
	57> -p7-		55> -p6-	55> -p6-	55> -p6-			55> -p7-	64> -p4-				SE	<55 -p6-	
	58> -p7-		56> -p6-	56> -p6-	56> -p6-			56> -p7-	63> -p4-				56> -p7-	SN	
			59> -p6-		59> -p6-										RV

Tabulka 1.1 Tabulka přechodů mezi módy [8, kap. 4.6.2]

[2]Podmínek je příliš mnoho na kompletní vypsání, proto uvádím odkaz do oficiální specifikace, kde se nachází stejná tabulka se všemi podmínkami přechodů. [8, kap. 4.6.3]

1.1.5 Movement Authority

Koncept „Movement Authority (MA)“ je klíčovým principem celého fungování ETCS, který má důležitý význam při řízení vlaků. „Movement Authority“ definuje povolení k jízdě vlaku po trati. V rámci tohoto povolení se předávají údaje o maximální rychlosti, profilu trati a délce tohoto povolení.

Tyto informace se posílají ve zprávě 3: „Movement Authority“, která má strukturu popsanou v tabulce 1.2.

Název proměnné	Popis proměnné
NID_MESSAGE	Jednoznačný identifikátor zprávy
L_MESSAGE	Délka zprávy v bytech
T_TRAIN	Čas vlaku
M_ACK	Kvalifikátor pro žádost o potvrzení
NID_LRBG	Identifikátor poslední přejeté balízové skupiny (LRBG)
Packet 15	Paket obsahující informace o potvrzení k jízdě
Optional Packets	Další volitelné pakety

Tabulka 1.2 Struktura zprávy 3: Movement Authority

V této zprávě je také povinný paket 15: „Movement Authority“. V tabulce 1.3 popisuji jeho strukturu.

Název proměnné	Popis proměnné
NID_PACKET	Jednoznačný identifikátor paketu
Q_DIR	Kvalifikátor směru poslaných dat
L_PACKET	Délka zprávy v bytech
Q_SCALE	Měřítko vzdáleností
V_LOA	Dovolená rychlost na konci oprávnění jízdy
T_LOA	Jak dlouho platí V_LOA od poslaní zprávy (timeout)
N_ITER	Počet iterací datové sady následující po této proměnné
L_SECTION(n)	Délka úseku v MA
L_ENDSECTION	Délka posledního úseku v MA
Q_SECTIONTIMER	Zda se k danému úseku vztahuje časový limit úseku
Q_ENDTIMER	Zda existují pro konec MA informace o časovači.
Q_DANGERPOINT	Zde existuje popis bodu nebezpečí.
Q_OVERLAP	Zda se na konci MA je rezerva pro případ nedobrzdění.

Tabulka 1.3 Struktura paketu 15: Movement Authority

Tato zpráva se posílá z traťové části na vyžádání od vlakové části. Toto vyžádání je nejčastěji formou zprávy 132: „MA Request“, nebo zprávy 136: „Train position report“. Traťová část ETCS tyto zprávy přijme, zpracuje a po vyhodnocení situace na trati, sestaví zprávu o povolení k jízdě — nebo také o nepovolení — a pošle jí zpět části vlakové.

1.1.5.1 Gradient profile

„Gradient Profile (GP)“ popisuje změny ve sklonu trati podél určitého úseku železnice. Tento sklon je zcela zásadní pro výpočet brzdné dráhy vlaku a také se využívá při plánování trasy.

Tyto informace se vlaku posílají ve volitelném paketu 21: „Gradient Profile“. Strukturu toho paketu popisuje tabulka 1.4.

Název proměnné	Popis proměnné
NID_PACKET	Jednoznačný identifikátor paketu
Q_DIR	Kvalifikátor směru poslaných dat
L_PACKET	Délka zprávy v bytech
Q_SCALE	Měřítko vzdálenosti
D_GRADIENT	Vzdálenost k další změně gradientu
Q_GDIR	Zda je do kopce (1), nebo z kopce (0)
G_A	Stoupání v promilích
N_ITER	Počet iterací datové sady následující po této proměnné
D_GRADIENT(n)	**n** dalších D_GRADIENT
Q_GDIR(n)	**n** dalších Q_GDIR
G_A(n)	**n** dalších G_A

Tabulka 1.4 Struktura paketu 21: Gradient Profile

1.1.5.2 Static speed profile

Static Speed Profile (SSP) popisuje maximální povolené rychlosti na úsecích trati. Tento profil poskytuje vlaku informace o rychlostních omezeních, které vlak musí dodržovat v závislosti na aktuální poloze a směru jízdy.

Zase se jedná o volitelný paket 27: „Static Speed Profile“. Struktura je popsána v tabulce 1.5.

Název proměnné	Popis proměnné
NID_PACKET	Jednoznačný identifikátor paketu
Q_DIR	Kvalifikátor směru poslaných dat
L_PACKET	Délka zprávy v bytech
Q_SCALE	Měřítko vzádleností
D_STATIC	Vzdálenost k další změně omezení rychlosti
V_STATIC	Hodnota omezení rychlosti
Q_FRONT	Kvalifikátor zda omezení plati pro začátek/konec soupravy
N_ITER	Počet iterací datové sady následující po této proměnné
NC_DIFF	Používá se k určení speciálních tříd vlaků
V_DIFF	Diferenciální hodnota rychlostí k těmto speciálním třídám
N_ITER	Počet iterací datové sady následující po této proměnné
D_STATIC(n)	**n** dalších D_STATIC
V_STATIC(n)	**n** dalších V_STATIC
Q_FRONT(n)	**n** dalších Q_FRONT
N_ITER(n)	Počet iterací datové sady následující po této proměnné
NC_DIFF(n,k)	matice **n** x **k** dalších NC_DIFF
V_DIFF(n,k)	matice **n** x **k** dalších V_DIFF

Tabulka 1.5 Struktura paketu 27: Static Speed Profile

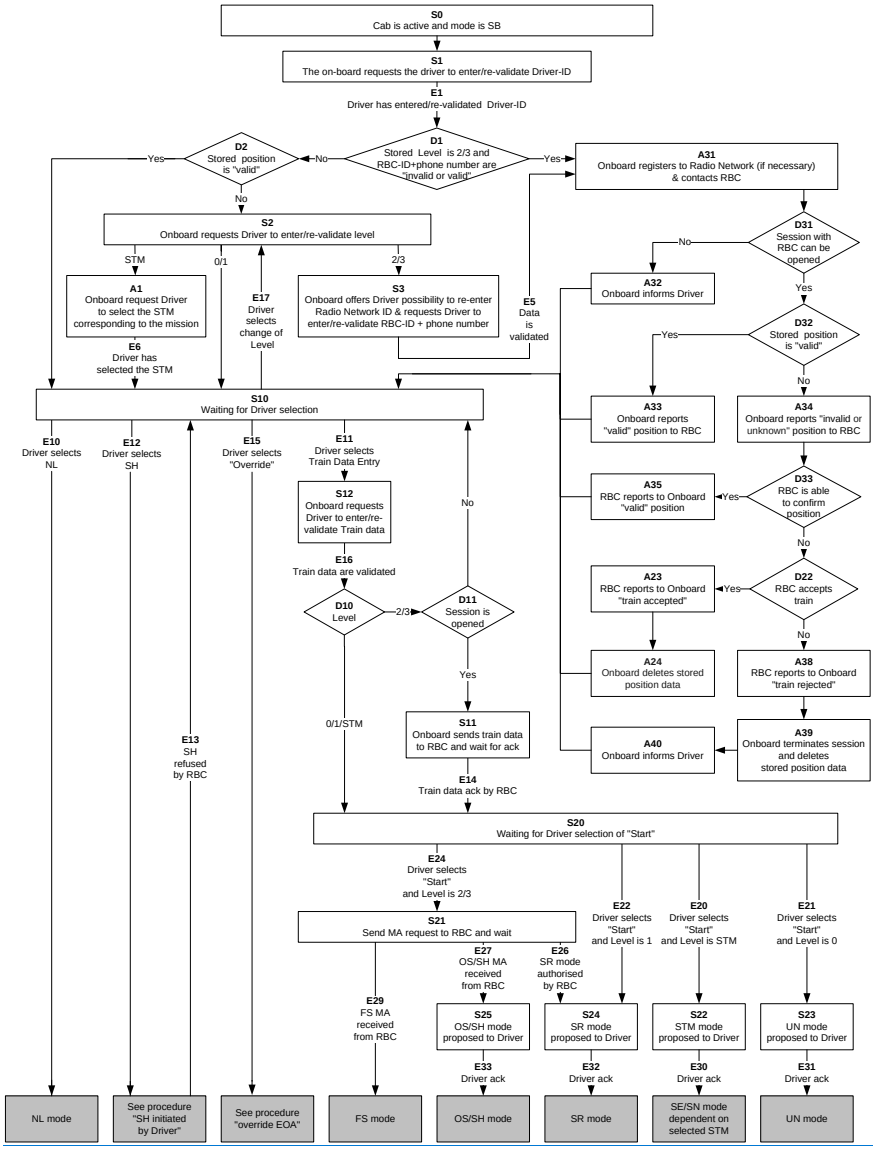
1.1.6 Procedury

Dále bych zde rád shrnul nejdůležitější procedury, které probíhají při provozu ETCS. Vzhledem k tomu že tato práce se věnuje komponentě RBC, zaměřím se na tyto procedury z jejího pohledu.

1.1.6.1 Start of mission

Tato procedura umožňuje vlaku začít novou operaci nebo misi pod dohledem ETCS. Zahrnuje řadu kroků, které musí strojvedoucí a systém provést, aby bylo zajištěno, že vlak je připraven a schopen bezpečně vykonat plánovanou jízdu podle pravidel ETCS. Cílem je zajistit, aby všechny systémy a komponenty byly ve správném stavu a aby byly splněny všechny podmínky pro bezpečný provoz. [9, kap. 5.4]

Tato procedura je popsána diagramem 1.2 převzatého z oficiální specifikace.



Obrázek 1.2 Kompletní diagram přechodů Start of mission [9, kap. 5.4.4]

Z pohledu RBC je zde několik důležitých částí:

Navázaní spojení s vlakem: Tato část SoM pokrývá sekci digaramu, kde vlaková část naváže komunikační spojení s traťovou.

A31: EVC v rámci zprávy 155: „Initiation of a communication session“ [6, kap. 8.6.13] pošle informace o vlaku.

D31: Pokud toto spojení lze vytvořit, RBC na tuto zprávu reaguje zprávou 32: „Configuration Determination“. [6, kap. 8.7.12] V ní pošle verzi, na které by chtělo komunikovat.

Pokud s touto konfigurací EVC souhlasí, tak odpoví zprávou 159: „Session established“. [6, kap. 8.6.17] Od této doby se spojení bere jako navázané (established).

Ověření pozice: Poté co se vlak a RBC dohodnou na verzi na které budou komunikovat, tak je další na řadě zjistit, kde se vlak nachází.

D32: EVC nejdříve zkontroluje, zda má nějakou polohu u sebe uloženou.

A34: Pokud ne, tak pošle zprávu 157: „SoM Position Report“. [6, kap. 8.6.15] Položku Q_STATUS při tom vyplní jako *unknown*. Tím dá vědět RBC, že neví, kde se nachází a je na něm, jak s tímto naloží.

D33/D22: Zde RBC zkontroluje, co mu přišlo od EVC. Pokud byla přijatá pozice označena jako *unknown*, nebo *invalid*, tak proces pokračuje dále.

A23: Pokud se RBC rozhodne, že je vše v pořádku a přijme ho i s neznámou pozicí, tak odpoví zprávou 41: „Train Accepted“. [6, kap. 8.7.19]

A24: EVC po přijetí smaže data, co mělo uložené a pokračuje s neznámou pozicí.

Tímto se obě části ETCS shodly na pozici a mohou pokračovat dále.

Ověření údajů o vlaku: Jakmile byla dohodnuta pozice, musejí se ještě části ETCS sesynchronizovat o technických specifikacích vlaku.

S12: EVC si od strojvedoucího vyžádá zadat/revalidovat data o vlaku.

S11: Tyto informace se předají jakmile je strojvedoucí potvrdí. EVC je pošle RBC ve zprávě 129: „Validated train data“. [6, kap. 8.6.1]

E14: Tyto informace si RBC uloží, zkontroluje a odpoví zprávou 8: „Acknowledgement of Train Data“. [6, kap. 8.7.4] Tímto potvrzuje přijetí dat a schvaluje jejich formát.

Vyžádání povolení k jízdě: Dále už zbývá jen získat povolení k jízdě.

E24: Toto se stane po té, co strojvedoucí zvolí tlačítko „Start“ na DMI.

S21: EVC hned na to pošle žádost o MA ve zprávě 132: „MA request“. [6, kap. 8.6.3] RBC tuto zprávu přijme, vyhodnotí všechny dostupné údaje a pošle odpověď podle situace na trati.

E29: Pokud jsou dostupné údaje o lokaci vlaku, tak odpovědí může být zpráva 3: „Movement Authority“. [6, kap. 8.7.2] Ta umožní vlaku se rozjet v plném rozsahu.

E27 : Pokud tyto informace byly neznámé (*unknown*), tak bude odpovědí zpráva 2: „SR Authorisation“. [6, kap. 8.7.1]

S25: Vzhledem k tomu, že nepřišlo kompletní povolení k jízdě, ale pouze autorizace k Staff Responsible (SR), tak strojvedoucí musí potvrdit přechod do módu SR.

E33: Jakmile řidič potvrdí přechod do módu SR, tak se vlak může rozjet.

1.1.6.2 End of mission

Procedura End of mission (EoM) referuje k situaci, kdy traťová část přestane autorizovat pohyb po trati. [9, kap. 5.5.1]. Je mnohem jednodušší než Start of mission. Jako End of mission se počítájí následující přechody mezi módy: [9, kap. 5.5.2]

Do Stand By (SB): Když ETCS přejde do SB z módů: FS, OS, UN, NL, SR nebo RV.

Do Sleeping (SL): Kdykoliv se přejde do režimu „Sleeping“.

Do Shunting (SH): Pokud přejde systém ETCS do SH z módů: FS, OS, SR, PT nebo UN.

Dále lze End of mission (EoM) dosáhnout následujícími kroky: [9, kap. 5.5.3]

1. MA, popis trati a data vlaku mohou být smazána.
2. EVC pošle RBC zprávu 150: „End of mission“. [6, kap. 8.6.10]
3. RBC vyžádá ukončení spojení.
4. EVC pošle zprávu 156: „Termination of communication session“ [6, kap. 8.6.14], kterou pro EVC mise končí.
5. RBC toto potvrdí zprávou 39: „Acknowledgement of termination of a communication session“ [6, kap. 8.7.17] a bere misi za ukončenou.

1.1.6.3 Train Trip

Procedura „Train Trip“ je bezpečnostní mechanismus, který je aktivovaný, když vlak poruší nebo je na cestě k porušení bezpečnostních podmínek diktovanými systémem ETCS. Tato procedura je navržená tak, aby převedla vlak do bezpečného stavu, ve kterém je jeho pohyb zastaven. Vlak takto setrvá, dokud se situace nevyřeší.

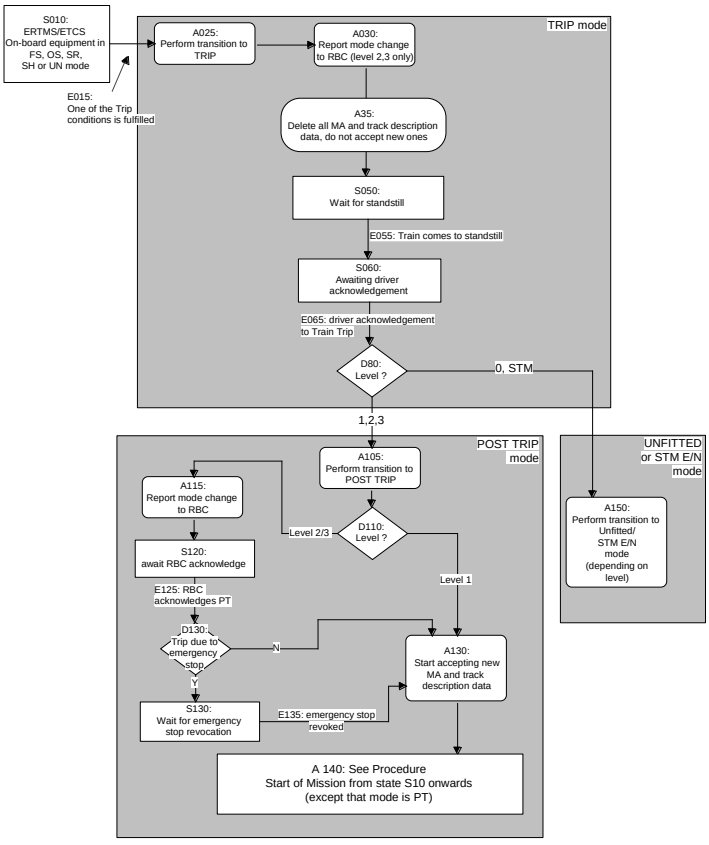
Podmínky pro přechod do módu Trip mohou být vidět v tabulce 1.1. Zde shrnu jen ty podmínky, které jsou relevantní pro mojí práci. (ostatní podmínky jsou pro traťovou část nebo pro funkcionality, které simulátor zatím neimplementuje)

16: Vlak přejede EoA a ETCS level je 2/3.

20: Přijde „Unconditional emergency stop“.

36: Vlak přejede balízu, která není v seznamu balíz, které má očekávat v módu SR.

Samotný proces „tripu“ je popsaný v diagramu 1.3. Lze zde vidět, že jakmile je splněna jedna z podmínek, tak vlak okamžitě přechází do tripu. Poté informuje RBC, že se tak stalo a smaže data o MA, popis trati a čeká na zastavení. Jakmile stojí, tak čeká na potvrzení od strojvedoucího. Následovně (vzhledem k tomu, že jsme v levelu 2) se přesune do módu Post Trip (PT). Zase ohlásí svou změnu módu RBC a čeká na jeho potvrzení od trati. Pokud bylo zastaveno na základně nouzového brzdění, tak se čeká na odbrzdění. Pokud ne tak se pokračuje z pozice *S10* v diagramu SoM 1.2.



Obrázek 1.3 Diagram procedury „Train Trip“ [9, kap. 5.11.2]

1.1.6.4 Emergency stop

Možnost nouzového zastavení je naprosto klíčová pro situace, kdy je nezbytné okamžitě zastavit, kvůli bezprostřední hrozbě nebo nebezpečné situaci. Vlak je možné zastavit pomocí zprávy o nouzovém zastavení s podmínkou („*Conditional emergency stop*“), nebo bez podmínky („*Unconditional emergency stop*“). Pokud vlak dostane zprávu 16: „Unconditional Emergency Stop“ [6, kap. 8.7.7], tak musí okamžitě zastavit. [4, kap. 3.10.2.1.3].

Struktura zprávy 16 je popsána tabulkou 1.6.

Název proměnné	Popis proměnné
NID_MESSAGE	Jednoznačný identifikátor zprávy
L_MESSAGE	Délka zprávy v bytech
T_TRAIN	Čas vlaku
M_ACK	Kvalifikátor pro žádost o potvrzení
NID_LRBG	Identifikátor poslední přejeté balise groupy (LRBG)
NID_EM	Jednoznačný identifikátor nouzového zastavení

Tabulka 1.6 Struktura zprávy 16: Unconditional Emergency Stop [6, kap. 8.7.7]

Pokud se jedná o zprávu 15: „Conditional Emergency Stop“ [6, kap. 8.7.6], tak vlak musí zkontrolovat, zda již nepřejel svojí přední částí začátek. pokud již přejel tak tuto zprávu ignoruje. Jinak musí tuto zprávu přijmout a upravit svojí jízdu podle informací ve zprávě. [4, kap. 3.10.2.1.2]

Struktura zprávy 15 je popsána tabulkou 1.7.

Název proměnné	Popis proměnné
NID_MESSAGE	Jednoznačný identifikátor zprávy
L_MESSAGE	Délka zprávy v bytech
T_TRAIN	Čas vlaku
M_ACK	Kvalifikátor pro žádost o potvrzení
NID_LRBG	Identifikátor poslední přejeté balise groupy (LRBG)
NID_EM	Jednoznačný identifikátor nouzového zastavení
Q_SCALE	Měřítko vzdáleností
Q_DIR	Kvalifikátor směru poslaných dat
D_EMERGENCYSTOP	Vzdálenost od LRBG k místu nouzového zastavení

Tabulka 1.7 Struktura zprávy 18: Conditional Emergency Stop [6, kap. 8.7.6]

1.1.6.5 Track ahead free

Koncept Track Ahead Free (TAF) zajišťuje, že trať před vlakem je volná a bezpečná pro průjezd do následujícího návěstidla. RBC si kdykoliv může vyžádat potvrzení, že trať je volná formou zprávy 34: „Track Ahead Free Request“. [6, kap. 8.7.14]. Struktura této zprávy je popsána v tabulce 1.8.

Název proměnné	Popis proměnné
NID_MESSAGE	Jednoznačný identifikátor zprávy
L_MESSAGE	Délka zprávy v bytech
T_TRAIN	Čas vlaku
M_ACK	Kvalifikátor pro žádost o potvrzení
NID_LRBG	Identifikátor poslední přejeté balise groupy (LRBG)
Q_SCALE	Měřítko vzádleností
Q_DIR	Kvalifikátor směru poslaných dat
D_TAFDISPLAY	Vzálenost od LRBG k místu zobrazení potvrzeni TAF
L_TAFDISPLAY	Délka po kterou má být toto potvzení zobrazeno

Tabulka 1.8 Struktura zprávy 34: TAF Request [6, kap. 8.7.14]

Vlaková část má za úkol na tuto zprávu reagovat zeptáním se strojvedoucího, zda je trať před vlakem volná. Ten má možnost toto potvrdit a v tu chvíli EVC odpoví RBC [4, kap. 3.15.5.4] s potvrzením zprávou 149: „Track Ahead Free Granted“ [6, kap. 8.6.9]. Tato zpráva je opět popsána v tabulce 1.9. Pokud ovšem strojvedoucí tuto výzvu nepotvrdí, ETCS nebude provádět žádné omezující nařízení (ale nijak neinformuje traťovou část). [4, kap. 3.15.5.5]

Název proměnné	Popis proměnné
NID_MESSAGE	Jednoznačný identifikátor zprávy
L_MESSAGE	Délka zprávy v bytech
T_TRAIN	Čas vlaku
NID_ENGINE	Identifikátor vlaku
Packet 0/1	Následuje paket o poziční zprávě

Tabulka 1.9 Struktura zprávy 149: Track Ahead Free Granted [6, kap. 8.6.9]

Pomocí této procedury je možně prodlužovat povolení k jízdě ze sekce trati, kde není jisté, že se mezi vlakem a koncem úseku nenachází žádná překážka (vlak). Tím, že strojvedoucí potvrdí TAF, tak bere odpovědnost na sebe. [10, kap. 4.6.10] Informace o tom, jak daleko posílat toto potvrzení od návěstidla by měla být v popisu trati, avšak nikdy to není více než 400 metrů.

Dále se požadavek na TAF posílá před povolením k přechodu do FS z SR nebo OS. Tento přechod může proběhnout i automaticky (tzv. ATAF) a to tak, že vlak pošle oznámení polohy těsně před návěstidlem.

1.2 Komponenta RBC v simulátoru ETCS

V této sekci bych se chtěl zaměřit na stávající stav komponenty RBC, scénáře jízdy a na traťovou databázi.

1.2.1 Historie vývoje komponenty RBC

Nejprve bych rád trochu osvětlil historii vývoje na této komponentě. Poslouží to pro částečné pochopení, proč stav komponenty je takový jaký je. Historicky byl na vždy na každou komponentu vyhrazen jeden tým. Náš tým byl první, který dostal všechny komponenty najednou s vidinou je sjednotit.

Už když projekt začínal, tak to s komponentou RBC nevypadalo moc dobře. První tým sice začínal „na zelené louce“, ale neprovedl dostatečně důkladnou analýzu a ani návrh. Výsledkem jejich práce byl proto spíš neúplný prototyp než nějaká komplexní komponenta. Další tým, co RBC dostal do rukou, se během semestru rozpadl a proto vývoj prakticky nepokračoval. Poslední tým před námi na komponentě pracoval jen jeden semestr během kterého se stav komponenty příliš nezlepšil. Spíše bylo pouze zavedeno více mysteriozních chyb a nesrovnalostí do již nepřehledného kódu.

Během všech iterací různých týmů běžely zároveň paralelně týmy ostatních komponent. Velkým nedostatkem práce těchto týmů byla velmi slabá komunikace. Toto je velký problém vzhledem k tomu, že tyto komponenty spolu musí v simulátoru ETCS velmi úzce spolupracovat.

1.2.2 Současný stav komponenty RBC

Podívám-li se na komponentů RBC z pohledu programátora, není na tom vůbec dobře. Celá komponenta je spíše několik na sobě závislých skriptů něž kód s jakoukoliv promyšlenou architekturou. Místo použití výhod objektově orientovaného přístupu jako je například polymorfismus nebo abstrakce, jsou veškeré funkce naimplementovány v jediném velkém *switch-case*. Tento jev snižuje čitelnost a prakticky eliminuje jakoukoliv rozšiřitelnost.

1.2.2.1 Dokumentace

Dokumentace k RBC prakticky neexistuje. Občas se vyskytne nějaký komentář v kódu, ale těch co opravdu něco dokumentují je minimum (viz 1.1). Nějaké informace se mi podařilo získat od studentů, co na něm v minulosti pracovali, ale těchto informací bylo také minimum. Nezbylo mi nic jiného, než si pročíst stávající kód a oficiální specifikaci.

```
//TODO: come up with an exception
return "nope";
```

Ukázka 1.1 Ukázka kvality kódu

1.2.2.2 Konfigurace

Pro simulátor je velmi důležitá konfigurace simulace. V současném RBC je několik proměnných, které se tváří, že ovlivňuji chovaní komponenty. Jejich upravovaní je ale komplikované a hlavně se vždy celé RBC musí překompilovat. Tento způsob konfigurace není vhodný a proto je potřeba přijít s lepším řešením.

1.2.2.3 Traťová databáze

RBC čerpá data o popisu trati z traťové databáze. Tato databáze je vytvořena studenty z Fakulty dopravní. Pro mojí práci jsou nejdůležitější prvky balízy, rychlostní omezení a stoupání trati.

Data o balízách jsou uložená ve formátu hexadecimálních znaků. Když se tyto znaky přepíšou do binární formy, zařadí za sebe a poté postupně čteme bit po bitu tak dostaneme datový obsah této balízy (nazývaný *telegram* viz 1.10). [6, kap. 8.4.2] Když chceme tyto data z telegramu interpretovat, je nutné je nejdříve převést do čitelné formy a následně s nimi nakládat.

Název proměnné	Délka (v bitech)	Popis proměnné
Q_UPDOWN	1	Určuje směr toku informací jakožto vlak na trať (Q_UPDOWN=0) nebo opačně (Q_UPDOWN=1).
M_VERSION	7	Verze komunikace ERTMS/ETCS.
Q_MEDIA	1	Definuje typ média: 0 = balíza
N_PIG	3	Určuje pozici balízy ve skupině
N_TOTAL	3	Celkový počet balíz ve skupině
M_DUP	2	Určuje, zda je informace o balíze duplikátem předcházející nebo následující.
M_MCOUNT	8	Umožňuje detekci změny zprávy během průchodu jednou skupinou balíz.
NID_C	10	Označuje identifikátor státu.
NID_BG	14	Označuje identifikátor balízové skupiny.
Q_LINK	1	Umožňuje označit balízovou skupinu jako linkovanou (Q_LINK=1) nebo nelinkovanou (Q_LINK=0).

Tabulka 1.10 Popis balízových dat v telegramu

Celá databáze obsahuje desítky tabulek. Pro nás je důležitá jen část. Tuto část jsem znázornil na diagramu databáze 1.4. Kompletní popis databáze naleznete v příloze A.

Obrázek 1.4 Výřez databáze pro mé potřeby

1.2.2.4 Testování

Testování je velmi důležitá část jakéhokoliv programu. Bohužel komponenta RBC má testy implementovány jen částečně, většina z nich nevypadá příliš smysluplně a nefungují úplně nejlépe. Vypadá to, jako by byly napsány pro starší část kódu a během práce na projektu nebyly aktualizovány.

K dobrému dojmu také nepřispívá nulová automatizace. Testy se nikde automaticky nespouští a ani se nekontroluje jejich výsledek. Bylo by tedy vhodné, kdyby se v budoucnu testy psaly během vývoje a udržovaly. Také by bylo dobré aby se testy automaticky spouštěly a kontroloval se jejich výsledek.

1.2.3 Scénáře jízdy

Scénář jízdy je jakýsi předpis pravidel, podle kterých se simulace musí chovat. Jednotlivá pravidla jsou očíslována svým identifikátorem „ruleId“, přičemž toto číslo musí být unikátní.

Momentálně jsou ve scénáři jízdy pouze sekce, po kterých se mají vydávat povolení k jízdě. Každá sekce má absolutní polohu začátku a konce. Scénáře se ukládají a posílají ve formátu *JSON* viz ukázka 1.2.

```
rules: [
    {
        ruleId: 0,
        event: {
            beginPosition: 0,
            endPosition: 2100
        }
    },
    {
        ruleId: 1,
        .
        .
        .
    },
    .
    .
    .
]
```

Ukázka 1.2 Ukázka stávajícího scénáře jízdy

1.3 Komunikace přes protokol MQTT

Ke komunikaci mezi komponentami je momentálně používán protokol Mosquitto (MQTT). Tento protokol je postaven na principu „publish-subscribe“, který funguje následovně. Účastníci komunikace se nejprve musí připojit k *mqtt brokerovi*[3]. Poté nějaký klient vystaví („publishne“) zprávu na daném tématu („topic[4]”). Jiný klient se může k tomuto tématu přihlásit („subscribe“) a tím pádem, bude zprávy, které byly na toto téma poslány, odposlouchávat a *mqtt broker* mu je doručí.

V našem simulátoru ETCS se například používá „topic“ ve tvaru *RBC/EVC*, který je určen na komunikaci od RBC do EVC. Každý tok zpráv má určený „topic“ viz tabulka 1.11.

„Topic“	Tok komunikace
RBC/EVC	RBC -> EVC
EVC/RBC	EVC -> RBC
LPC/RBC	LPC -> RBC
RBC/LPC	RBC -> LPC
RBC/JRU	RBC -> JRU

Tabulka 1.11 Popis všech různých „topics“

1.3.1 Implementace MQTT v RBC

Momentálně je v RBC několik komplikovaných adresářů se všemi možnými zprávami, viz popis adresáře 1.3. Zpráv je sice hodně (což ale musí být, protože jich opravdu tolik různých definuje specifikace), ale problém je, že jejich pojmenování je někdy nepřesné. To vede k tomu, že se těžce hledá, kterou zprávu ze specifikace implementují. Navíc chybí potřebné komentáře, což významně komplikuje identifikaci jednotlivých zpráv.

```
src
├── MessagesAndPackets ..... Tady je ještě navíc 8 různorodých souborů
│   ├── Messages ..... Zde se nachází 30 souborů
│   └── Packets ..... Zde je 38 souborů
│       ├── MAHelper ..... Několik pomocných tříd
│       └── ReportHelper ..... Opět několik pomocných tříd
└── MQTTClient ..... Tady je podivná implementace MQTT klienta
```

Ukázka 1.3 Popis struktury adresáře se zprávami a MQTT klientem

Dalším problémem je — co jsme jako tým během běhu předmětu SP1 objevili — že každá komponenta má zprávy implementované u sebe. Nejhorší na tom je, že tyto implementace se často liší. Ve stavu v jakém jsme jako tým tyto komponenty převzali, se spolu nedorozumí. Je to jako by mluvily odlišným jazykem. Proto by bylo vhodné tyto zprávy nějak sjednotit.

Samotná implementace MQTT klienta se nachází v adresáři MQTTClient. Najdeme zde třídu „MosquittoLib“, která obsahuje pouze volání knihovní funkce `mosquitto_lib_init();` a `mosquitto_lib_cleanup();`. Nic jiného tato třída nedělá. Poté je tady i třída „MQTTClient“, která pouze zaobaluje knihovní funkce z knihovny *mosquitto*.

[3] broker funguje jako jakýsi server, který distribuuje zprávy mezi klienty
[4] „topic“ se dá představit jako nějaký kanál na kterém komunikace probíhá

1.4 Funkční a nefunkční požadavky

V této sekci se zaměřím na funkční a nefunkční požadavky týkající se této práce. Toto je důležité pro pochopení co přesně RBC má a nemá dělat. To samé platí pro scénáře, aby bylo jasné co a jakou formou má v nich být uložené. Odpovídám tím na otázku: „Co má RBC dělat?“. Tyto požadavky jsem získal po konzultaci s panem docentem Lesem a zároveň, co jsem vypozoroval u stávající implementace simulátoru.

1.4.1 Funkční požadavky na novou komponentu RBC

Na tomto místě se nachází konkrétní požadované funkce nové komponenty RBC.

Generovaní povolení k jízdě RBC musí být schopno vydávat povolení k jízdě. Tyto povolení se generují podle toho, jak bylo specifikováno v pravidlech scénáře jízdy.

Konfigurační soubory ze strany LPC Předtím než začne simulace, LPC může poslat konfigurační soubory které ovlivní chovaní RBC během samotné simulace. Pokud zpráva s konfigurací nepřijde, tak jsou použity předem dané „defaultní“ hodnoty.

Reakce na lektorské pracoviště (start, stop, restart) LPC může kdykoliv poslat příkazy ke spuštění/restartovaní/zastavení simulace a RBC na to musí adekvátně reagovat.

Heartbeat Jakmile přijde zpráva od LPC k zahájení simulace, RBC začne periodicky (každou jednu sekundu) posílat zprávu, že žije. Tomuto říkáme „heartbeat“. Tato funkce je požadavkem ze strany projektu ETCS.

Správa nouzových brzdění Kdykoliv je vyžádáno nouzové brzdění (ať už scénářem nebo od LPC za běhu), tak RBC musí tyto nouzové brzdění hlídat a po uplynuté době zrušit, aby vlak mohl dále pokračovat v jízdě.

1.4.2 Funkční požadavky na scénáře jízdy

Zde bych rád shrnul funkční požadavky, které se týkají scénářů jízdy. Scénáře se používají pro konfiguraci simulace, takže je potřeba, aby byly jednoduše rozšiřitelné a čitelné. Také jsou zde požadavky na jejich formu a obsah.

Jednoduchá rozšiřitelnost Klíčové pro nové scénáře je jejich jednoduchá modifikace a hlavně rozšiřitelnost. Je potřeba proto zvolit, jak jednoduchý formát, tak i takový, ke kterému půjde snadno přidat další pravidla k simulaci.

Čitelnost S rozšiřitelností je úzce spjatá i čitelnost. Je důležité, aby tyto scénáře byly co nejednodušší a i lehce čitelné.

Movement Authority (MA) Tento požadavek je zcela stěžejní. Stávající scénáře toto umí, takže je jasné, že nové scénáře toto musí umět také. Je potřeba nést informaci o tom, kde sekce začíná a kde končí.

Nouzové brzdění V rámci scénářů je potřeba mít možnost určit místa kde se bude nouzově brzdit. Existují dva typy nouzového brzdění a pro obě jsou potřeba pravidla. Pro bez-kondiční nouzové brzdění stačí pozice, kde se má začít brzdit. Pro kondiční je potřeba pozice, kde má trať informovat vlak, že bude brzdit a pozice, kde se pak opravdu má začít brzdit.

Track Ahead Free (TAF) Tato část scénářů je asi nejméně důležitá, protože zbytek simulátoru ETCS tuto funkcionalitu nemá naimplementovanou. Tak či tak, tento typ pravidla ve scénáři musí nést informaci o tom, kde má systém strojvedoucímu tuto zprávu zobrazit a jak dlouho tuto zprávu zobrazovat.

1.4.3 Nefunkční požadavky

Kompletní přepis Z analýzy jasně vyplývá, že komponenta RBC je nepřehledná a nerozšiřitelná. Proto je potřeba kompletní přepis celé komponenty. Bude potřeba navrhnout novou architekturu, která vice vyhovuje požadavkům projektu.

Rozšiřitelnost Jeden z hlavních požadavků na výše zmíněnou architekturu je rozšiřitelnost. Projekt ETCS se neustále mění a rozrůstá a proto je potřeba navrhnout architekturu tak, aby jednoduše šlo přidávat nové zprávy, reagovat na tyto nové zprávy a adekvátně odpovídat.

Možnost paralelního zpracování Pro efektivní fungování je důležité využít více vláken pro obsluhování více požadavků najednou.

Sjednocení Protože je celý projekt simulátoru ETCS rozdělený do spoustu týmů a spoustu komponent, bylo by vhodné komponenty, které jsou spolu úzce spjaty, nějakým způsobem sjednotit. Toto významně pomůže členům budoucích týmů se zorientovat v kódu ostatních komponent.

Optimalizace testování Hlavní účel testování je odhalit chyby před tím, než způsobí nějaké komplikace. Na testování je důležité myslet už při návrhu. Špatně navržený kód se špatně testuje. Také je potřeba testy zautomatizovat a kontrolovat výsledek testování.

Kapitola 2
Návrh

Zde se bych rád představil můj návrh a vysvětlil proč, co a jak.

2.1 Návrh nové architektury

Z analýzy jasně vychází, že je potřeba celou komponentu RBC přepsat. K tomuto přepisu je důležité navrhnout architekturu, která vyhovuje požadavkům projektu.

Částečně budu čerpat z architektury, která byla navrhnuta minulým týmem komponenty EVC. Zde bych rád poděkoval Romanu Špankovi, který mi poskytl informace a myšlenky okolo této architektury. Ta má však spoustu nedostatků a není úplně dotažená do konce (z nedostatku času) a proto si přejímám hlavně koncept služeb a způsob zpracování zpráv.

2.1.1 Koncept nové architektury

Celý návrh architektury se silně inspiruje dvěma návrhovými vzory.

Event Driven Architecture (EDA): Tento vzor je velmi užitečný, pokud je potřeba měnit vnitřní stav na základě nějakých událostí („eventů“). [11] V mém případě tyto události budou přijetí různých zpráv. Celé to stojí na tzv. „Event handler“, což je třída, která je zavolána, pokud nastane jí přidělená událost. Pro mé použití bude třeba mírná modifikace. Pro každou zprávu vytvořím *message handler*, který bude zodpovědný za obsloužení dané zprávy. Například pro zprávu 136 („Position report“) vytvořím „PositionReportMessageHandler“, který kdykoliv přijde zpráva 136, tak bude zavolán a spracuje její obsah.

Service Oriented Architecture (SOA): Tento typ se zaměřuje na diskrétní služby (*services*), namísto monolitického návrhu.[12] Jedna služba je samostatná jednotka, která je jednoduše přístupná uvnitř programu. Hlavní výhodou tohoto typu architektury je velmi jednoduchá rozšiřitelnost. Dále tím, že je kód rozdělen do logických celků, tak je také mnohem lépe čitelný.

Nová architektura bude využívat konstruktorový *dependency injection* s výjimkou služeb (viz *IInitializable* 2.1.2.1).

2.1.2 Služby

Základní stavební jednotkou nové architektury jsou služby (*services*). Ty zaobalují více funkcí, které spolu souvisí, do jednoho logického celku. Mají veřejné metody pro volání těchto funkcím či pro ukládání a následné získávání dat.

Zde bych rád vyzdvihl nějaké služby, jejichž existence plyne přímo z analýzy (další určitě přibudou během fáze implementace (3.1.1)).

TrainRegistryService Tato služba má za úkol spravovat data o vlacích, které jsou aktivně připojené k RBC. Umožňuje tyto data zapisovat a následně číst.

DatabaseService Databázová služba je tu od toho, aby spravovala připojení k traťové databázi a následně se také do databáze dotazovala. Musí být schopna vytáhnout data o balízách, o sklonu trati a o rychlostních omezeních.

MqttListenerService *MqttListener* je služba, která neustále poslouchá na tématech komunikace MQTT, které byly specifikovány konfigurací. Kdykoliv přijde zpráva na tomto tématu, tak najde zodpovědný *topic worker* (viz 2.1.3) a přidá mu jí do fronty ke zpracování.

MqttPublisherService *MqttPublisher* je naopak v roli odesílatele zpráv přes MQTT. Umožňuje odeslat zprávu, která mu byla předem připravena a předána.

2.1.2.1 Polymorfismus u služeb

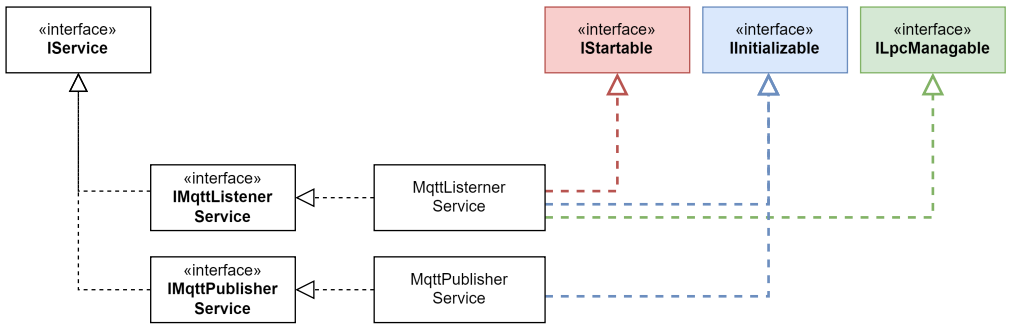
Každá služba dědí ze svého rozhraní. Toto rozhraní má všechny veřejné metody, které by tato služba ráda měla a vždy dědí ze třídy „IService“. Tímto je tato třída označena, že je to služba. Následně může každá libovolně dědit z těchto tří rozhraní:

IInitializable: Rozhraní *IInitializable* přidává možnost tuto službu inicializovat pomocí virtuální metody `Initialize()`. Tato metoda se využívá tehdy, pokud služba potřebuje ostatní služby pro nějakou funkcionalitu (platí prakticky pro všechny služby).

IStartable: Toto rozhraní označí třídu jako spustitelnou pomocí virtuální metody `Start()`. Zavoláním této metody se vytvoří nové vlákno a služba se v něm spustí.

ILpcManageable: Tímto je služba označena, že reaguje na zprávy od lektorského pracoviště. Přidává virtuální metody `LpcSaidStart()`, `LpcSaidRestart()` a `LpcSaidStop()`.

Na diagramu 2.1 je ukázka dědičnosti u služeb *MqttListenerService* a *MqttPublisherService*. Všechno více objasním v implementační části práce.



Obrázek 2.1 Digram dědičnosti služeb

2.1.2.2 Kontejner služeb

Na ukládaní služeb v nové architektuře bude sloužit kontejner služeb (*service container*). Jeho rolí je uchovávat instance služeb a následně poskytovat tyto služby pro zbytek programu k využití. Je zodpovědný se postarat o uvolnění paměti na konci běhu programu.

Každá služba se nejdříve do kontejneru přihlásí. Tím se uloží do vnitřní mapy, která je indexovaná typem služby (což je vlastní výčtový typ). Poté kontejner všechny služby iniciuje (zavolá metodu *Initialize()* na službách označených jako *IInitializable*) a jakmile jsou všechny služby inicializovány, tak ty které jsou označeny jako spustitelné (pomocí *IStartable*) spustí.

2.1.3 Topic workers

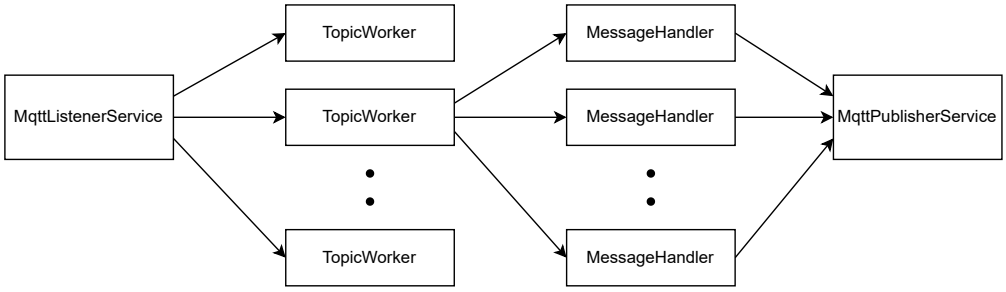
Topic worker je třída, jejíž instance běží každá na jednom vlákně. Každý jeden *worker* je zodpovědný za jedno téma (*topic*). Jak bylo výše popsáno, kdykoliv *MqttListener* dostane zprávu, tak najde zodpovědný *topic worker* a zprávu mu předá.

Tento *topic worker* pracuje tak, že má frontu všech zpráv, které mu *MqttListener* předá. Postupně je zprácovává tím, že první zkontroluje, zda zpráva obsahuje proměnnou *NID_MESSAGE*. Následně podle její hodnoty — protože je schopný podle ID jednoznačně identifikovat typ zprávy — pošle tuto zprávu dále ke zpracování přes konkrétní *message handler*.

2.1.4 Message handlers

Message handler hraje roli takovou, že kdykoliv se *topic worker* dostane ve frontě k nějaké zprávě, tak zavolá odpovědný *message handler* a předá mu zprávu ke zpracování.

Po dostání zprávy jí zkusí naparsovat[1]. Tímto zkontroluje formát zprávy a také, že nikde při přenosu nebo přijetí nedošlo k chybě. Poté, pokud má zpráva nějaké pakety, zkusí naparsovat i ty. Pokud vše bylo úspěšné tak následuje samotná logika zpracování zprávy, která je pro každou zprávu jedinečná.



Obrázek 2.2 Popis zpracování zprávy

2.1.5 Konfigurace

Další velmi důležitá vlastnost nové architektury je možnost konfigurace chování. Tato vlastnost vyplývá z analytické části této práce (viz 1.2.2.2). Řešením jsou konfigurační soubory, které budou za běhu načítány podle potřeby.

S tím přichází i nová služba: „ConfigurationService“. Tato služba je navržena tak, aby bylo možné za běhu načítat, případně generovat, konfigurační hodnoty pro ostatní služby.

[1] převézt data do strukturované formy

2.2 CEM (Common Enums & Messages)

Jak vyplývá z analýzy, velký problém při komunikaci přes MQTT je rozdíl implementací zpráv v různých komponentách. Řešením je nový repositář Common Enums and Messages (CEM). V CEMu budou všechny zprávy, společné výčtové typy, pakety a ostatní sdílené věci. Hlavní výhodou co CEM přinese, je sjednocení implementací a tím pádem jistota, že to, co bylo odesláno, je i to, co je očekáváno při doručení.

Tento repositář bude vložený jako „submodule“ ve všech komponentách a všechny komponenty z něj budou čerpat co potřebují. Struktura repositáře by mohla vypadat jako na ukázce 2.1. Struktura se určitě bude rozrůstat během implementační části, hlavně do ní bude také zasahovat tým ETCS. Zmiňuji to zde, protože je to důležitá část nové architektury.

```
cem
├── Enums ..... Zde budou společné výčtové typy
├── Messages
│   ├── Evc
│   │   └── Rbc ..... Zde budou zprávy od EVC do RBC
│   └── Rbc
│       └── Evc ..... Zde budou zprávy od RBC do EVC
├── Packets ..... Zde budou pakety
├── Rules ..... Zde se budou nacházet pravidla scénářů
└── Other ..... Zde budou ostatní pomocné třídy
```

Ukázka 2.1 Návrh struktury adresáře CEM

2.2.1 Výčtové typy

Existuje spoustu výčtových typů, které je nutné mít centralizované v CEMu. Krásný příklad je *NID_MESSAGE*. Jedná se o identifikátor zprávy, který je vnitřně reprezentován formou výčtového typu a je nutné ho mít stejný napříč komponentami. Těchto tříd bude určitě spousta. Bude potřeba mít výčtový typ na úroveň, ve které ETCS operuje (viz. 1.1.3). To samé bude potřeba pro mód, ve kterém se ETCS nachází (viz. 1.1.4) a určitě mnoho dalších.

2.2.2 Ostatní pomocné třídy

V CEMu jsou také potřeba další pomocné třídy na zaobalení dat, aby nakládání s nimi bylo jednodušší. Krásný příklad jsou třídy *Location* a *Distance*.

2.2.2.1 Location

Třída „Location“ se využije pro reprezentování relativní pozice na trati. Toto určování polohy se vztahuje k balízám. To znamená ve třídě „Location“ bude informace, ke které balízové skupině se vztahujeme (pomocí jejího identifikátoru *NID_LRBG*) a dále vzdálenost od ní.

2.2.2.2 Distance

Protože v systému ETCS se používají jednotky s různým měřítkem (*Q_SCALE*), je potřeba mít nějakou třídu, která nám operace se vzdálenostmi usnadní. Tato práce naštěstí už byla částečně odvedena mým týmem, takže se akorát již stávající implementace (třída „Distance“) musí vyladit a přesunout do repositáře CEM k použití pro všechny komponenty.

2.2.3 Zprávy

Veškeré zprávy v CEMu budou rozdělené podle toho, odkud kam směřují. Na diagramu 2.1 jsem již nastínil, že pokud zpráva je od EVC do RBC, tak se bude nacházet v *cem/Messages/Evc/Rbc*. Toto usnadní hledání zpráv, protože všechny zprávy co sdílí odesílatele a destinaci (topic) budou u sebe.

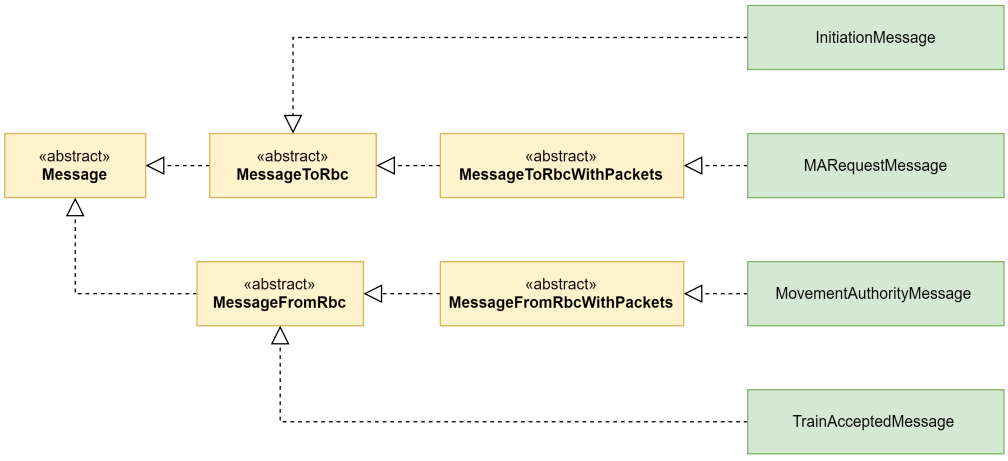
2.2.3.1 Hierarchie zpráv

Některé proměnné jsou společné pro více zpráv, ale každá zpráva obsahuje *NID_MESSAGE*. Proto vznikla abstraktní třída „Message“, která tuto proměnnou obsahuje.

Dále pro zprávy od RBC do EVC platí, že všechny zprávy mají společné tyto proměnné: *L_MESSAGE*, *T_TRAIN*, *M_ACK*, *NID_LRBG* a ty budou ve třídě „MessageFromRbc“. Dále tato zpráva může být rozšířena o pakety, proto jsem vytvořil třídu „MessageFromRbcWithPackets“.

Pro tok opačným směrem (EVC do RBC) platí, že společné proměnné jsou: *L_MESSAGE*, *T_TRAIN* a *NID_ENGINE*. Pro ně jsem vytvořil třídu „MessageToRbc“ a pokud má pakety tak „MessageToRbcWithPackets“.

Na diagramu 2.3 demonstruji tuto hierarchii na ukázce čtyř zpráv.„InitiationMessage“ je zpráva o navázaní spojení, je to jednoduchá zpráva, která dědí od „MessageToRbc“. „MARequestMessage“ je již trochu komplikovanější a hlavně má povinný paket o pozici vlaku. Proto dědí z „MessageToRbcWithPackets“. Její protějšek je „MovementAuthorityMessage“, která je odpověď od RBC s oprávněním jízdy. Tato zpráva má opět povinné pakety (viz. 1.1.5) a proto dědí z „MessageFromRbcWithPackets“. V neposlední řadě je zde „TrainAcceptedMessage“, která je opět velmi jednoduchá a jen dává vědět vlaku, že byl přijat. Díky tomu stačí, že dědí pouze z „MessageFromRbc“.



Obrázek 2.3 Ukázka návrhu zpráv na příkladu

Všechny tyto úrovně dědění se mohou zdát příliš komplikované ale jak uvidíte v implementační části, tak toto implementaci výrazně zjednoduší a přidávání nových zpráv (což je velmi častá a důležitá činnost při tvorbě tohoto simulátoru) bude velmi jednoduché.

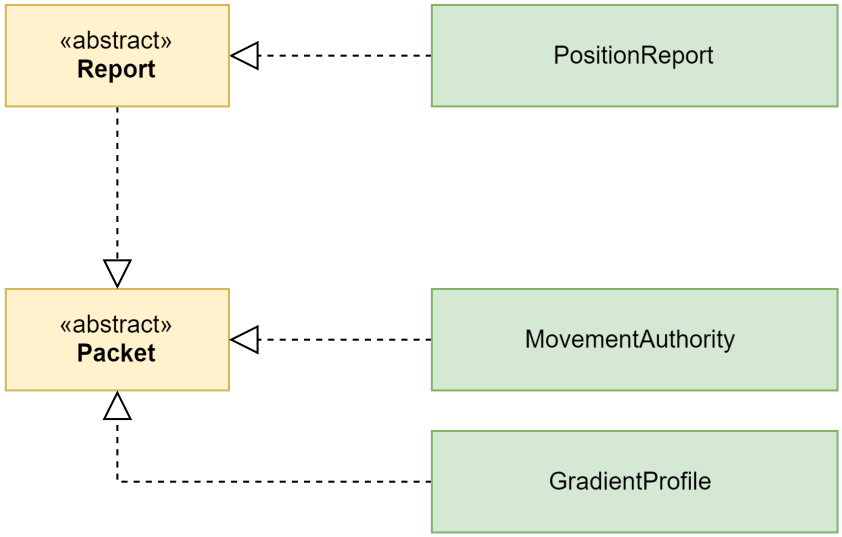
2.2.4 Pakety

Pakety se zatím vyskytují pouze ve zprávách mezi RBC a EVC. Proto je není nutné dělit tak detailně, jako zprávy. Samozřejmě pokud by v budoucnu přibyly další pakety a orientace by nebyla pohodlná, bude jednoduché je rozdělit.

2.2.4.1 Hierarchie paketů

Pro pakety jsem opět vytvořil hierarchii, tentokrát ale jednodušší. Pakety mají společné proměnné: *NID_PACKET* a *L_PACKET*. Tyto proměnné jsem zaobalil do třídy „Packet“. Dále existují dva pakety (0,1) které jsou typu „report“ a proto je zde i třída „Report“, která zaobaluje hodnoty společné pro ně.

Na diagramu 2.4 demonstruji tuto hierarchii na ukázce tří paketů. „PositionReport“ je paket o nahlášení pozice. Je typu „report“, tím pádem dědí ze třídy „Report“ , který dědí od „Packet“. „MovementAuthority“ je již trochu komplikovanější paket. Už není typu „Report“ a proto dědí přímo z „Packet“. Na závěr je zde „GradientProfile“, který obsahuje informace o stoupání trati (viz. 1.1.5.1) a ten také dědí přímo ze třídy „Packet“.



Obrázek 2.4 Návrh hierarchie paketů

2.2.5 Pravidla scénářů

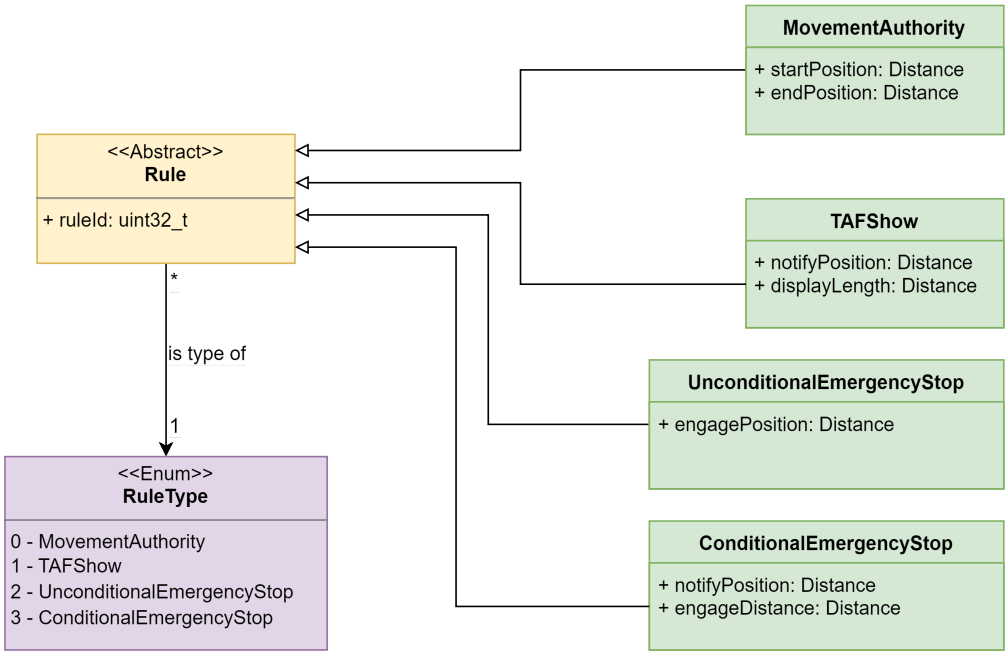
Scénáře jsou tvořeny pravidly (viz. 1.2.3) a tyto pravidla jsou definovaná na diagramu 2.5. Abstraktní třída „Rule“ obsahuje unikátní identifikátor a má vazbu na typ „RuleType“ (jeden z výčtových typů z CEMu). Tento typ se bude využívat při parsování (3.4.1) z formátu json.

Pravidlo „MovementAuthority“ se používá pro určení sekcí, kde má RBC rozdávat povolení k jízdě. Má dvě pozice, „startPosition“ a „endPosition“, které určují absolutní vzdálenost od začátku trati. Tyto dvě vzdálenosti určují začátek a konec úseku, kde platí dané povolení k jízdě.

„TAFShow“ je pravidlo, které udává, kde má být strojvedoucímu signalizováno potvrzení volné trati (*Track Ahead Free*). Údaj „notifyPosition“ udává, kde na trati být má toto potvrzení zobrazeno. Proměnná „displayLength“ je vzdálenost, která určuje jak dlouho má být zobrazeno. Po ujetí této vzdálenosti zase zmizí.

„UnconditionalEmergencyStop“ je pravidlo o bez-kondičním nouzovém brzdění (viz 1.1.6.4). Nese údaj „engagePosition“, který informuje o absolutní pozici, kde má být poslána zpráva k okamžitému nouzovému brzdění.

„ConditionalEmergencyStop“ je pravidlo o kondičním nouzovém brzdění (viz 1.1.6.4). Nese údaj „notifyPosition“, který informuje o absolutní pozici, kde má být poslána zpráva o oznámení následovného kondičního nouzového brzdění. Druhý údaj „engageDistance“ informuje o vzdálenosti od „notifyPosition“, kde se má začít brzdit.



Obrázek 2.5 Návrh nových scénářů

2.3 Návrh nových funkcionalit

V této sekci bych rád vysvětlil a popsal návrh funkcionalit, které přibudou s novým RBC. K popisu chování RBC jsem použil jazyk BPMN.

2.3.1 Heartbeat

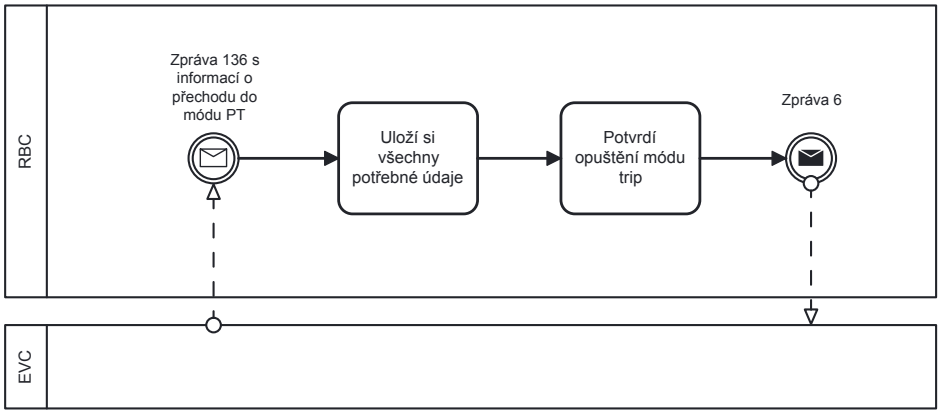
„Heartbeat“ by se měl posílat periodicky každou jednu vteřinu. Využije se zprávy definované v CEMu. Protože „heartbeat“ budou posílat všechny komponenty, tak tato zpráva nese informaci o kterou komponentu se jedná.

2.3.2 Reversing areas

V módu reversing je potřeba posílat vlaku oblasti, kde vlak může couvat, nebo-li „Reversing areas“. Tyto informace jsou posílány v packetu 138 „Packet 138: Reversing area infromation“. [7, kap. 7.4.2.34] Data o těchto oblastech by měly být uloženy v traťové databázi ale zatím nejsou. Tím pádem tato funkcionalita zatím nebude implementována.

2.3.3 Podpora módu post-trip

Jednou z dalších nových funkcí je podpora přechodu do módu Post Trip. Když EVC chce přejít do módu Post Trip, pošle po přechodu zprávu o pozici s módem Trip. Jakmile tuto zprávu RBC přijme, uloží si údaje o vlaku a změně módu a pošle zpět zprávu 6. Tento jednoduchý proces je popsán digramem 2.6.

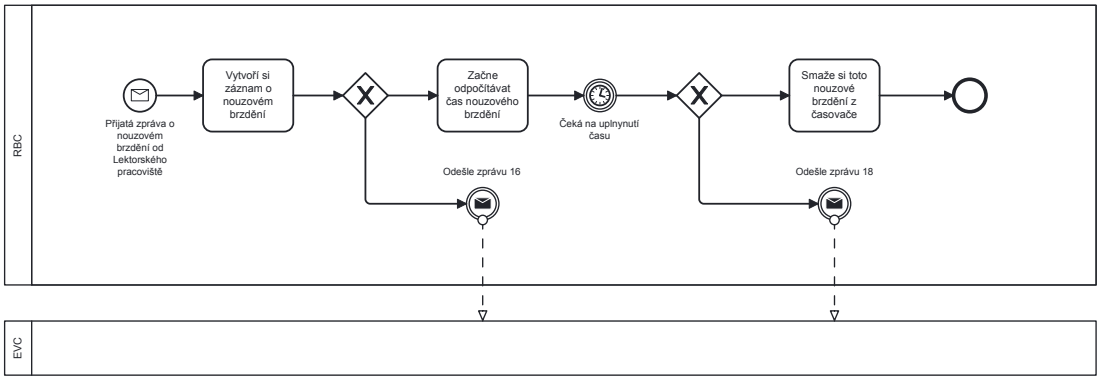


Obrázek 2.6 Popis průběhu přechodu do módu Post Trip (PT)

2.3.4 Správa nouzových zastavení

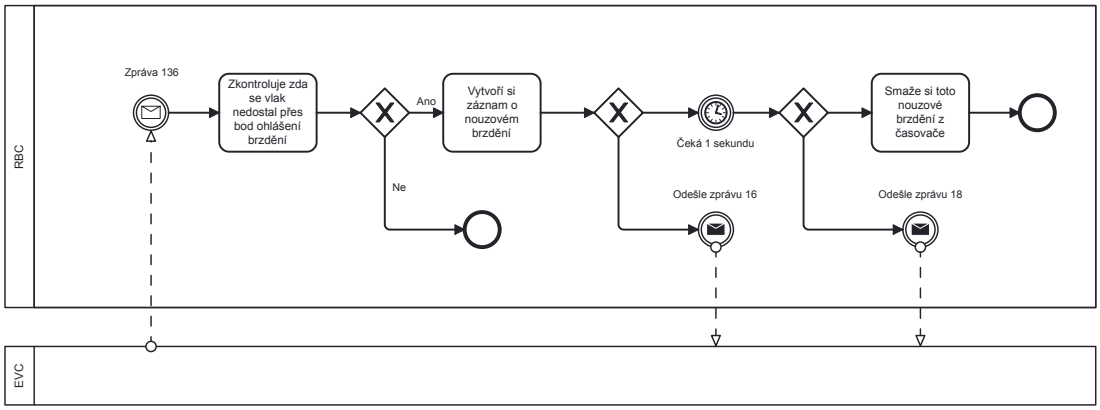
Nouzové zastavení může být iniciováno z mnoha stran. Ty, na které bych se chtěl zaměřit, jsou: ze strany lektorského pracoviště, pokud to určí pravidla simulace a z konzole (spíše pro testování).

Ze strany lektorského pracoviště může přijít zpráva o nouzovém zastavení. Tato zpráva v sobě nese informaci, kterého vlaku se týká a také jak dlouho se toto zastavení má držet. RBC po přijetí této zprávy musí okamžitě poslat zprávu 16. Poté začne počítat čas, jak dlouho má toto brzdění držet. Jakmile tento čas vyprší, tak pošle zprávu 18. Průběh tohoto je popsán níže v diagramu 2.7.



Obrázek 2.7 Popis průběhu nouzového zastavení iniciováno Lektorským pracovištěm

Pokud je nouzové zastavení vyvoláno pravidlem ve scénáři, proces je velmi podobný. RBC pokaždé když mu přijde zpráva o pozici vlaku, tak zkontroluje, jestli se vlak nedostal svojí novou polohou přes bod, kde má být ohlášeno nouzové brzdění. Pokud se tak stane, tak RBC okamžitě posílá zprávu o nouzovém brzdění. Průběh je opět popsán v BPMN diagramu 2.8.



Obrázek 2.8 Popis průběhu nouzového zastavení vyvoláno z pravidel scénáře

Kapitola 3
Implementace

Rád bych zde ujasnil, že vzhledem k roli této práce vůči projektu ETCS, bude tato část práce sloužit také jako dokumentace pro členy budoucích týmů. Proto zacházím trochu více do hloubky u každé služby/„message handleru“. Mám na mysli hlavně ukázky kódu a psaní o tom, co které metody dělají a proč. Věřím, že to budoucí tým ocení.

V této kapitole se zaměřím na východiska problémů, které byly shrnuty v analýze a na konkrétní implementaci navržených řešení. Rád bych upozornil, že všechny ukázky kódu jsou často zkrácené, nebo nějakým způsobem modifikované pro použití v textu. Obvykle, pokud nějaký kód vynechám, tak nebyl pro ukázku důležitý, nebo ho nahradím komentářem stylu `//komentář`.

Každý soubor, který jsem během implementace vytvořil je vybaven hlavičkou. Ta je sjednocená napříč celým projektem a jedna z informací v ní je list přispěvatelů, respektive jejich uživatelských jmen z fakulty. Tím je možno určit autorství. Všechen kód lze nalézt v příloze.

3.1 Jádro architektury

Pro vybudování jakékoliv aplikace je nejprve nutné položit základní kameny ve formě jádra architektury. Toto jádro se skládá s věcí jíž navržených v kapitole 2.1.1.

3.1.1 Služby

Z návrhu plyne, že každá služba má své rozhraní, které dědí z rozhraní *IService*. Implementace rozhraní je popsána v ukázce kódu 3.1. Je zde vidět, že má virtuální destruktor, který je důležitý přetížit v konkrétních službách, aby se vyhnulo únikům paměti. Zároveň se zde určuje typ této služby pomocí „static constexpr“. To umožňuje za kompilace předejít typovým nesrovnalostem. Tento typ je definován vlastním výčtovým typem *ServiceType*.

```
class IService {
public:
    virtual ~IService() = default;
    static constexpr ServiceType Type = ServiceType::None;
};
```

Ukázka 3.1 Kód rozhraní *IService*

Nepovinná rozšíření

Vytvořil jsem několik nepovinných rozhraní. Ty umožňují služby rozšířit o různé funkce. Více toto osvětlím později, kde se budu věnovat implementaci konkrétních služeb.

IInitializable Toto rozhraní přidává metodu `Initialize()`. Ta umožňuje již zmíněný „dependency injection“ tím, že v rámci této metody se předává kontejner služeb. To umožní službě, která tato rozhraní dědí, si během inicializace natahat všechny ostatní služby, které bude za běhu potřebovat.

K vyřešení cyklických závislostí mezi kontejnerem služeb a rozhraními se využije „forward deklarací“, které zajistí, že kód půjde zkompilovat a slinkovat. Dále je potřeba deklarovat kontejner služeb jako „friend class“, aby mohl volat metody tohoto rozhraní (nechceme aby byly veřejné).

```
class ServiceContainer; //toto je forward deklarace

class IInitializable {
protected:
    friend class ServiceContainer;

    virtual void Initialize(ServiceContainer& container) = 0;
};
```

Ukázka 3.2 Kód rozhraní *IInitializable*

IStartable Rozhraní *IStartable* označí službu jako spustitelnou. Tohoto docílí přidáním několika metod. `Start()` spustí tuto službu. `Wait()` je metoda kterou volá kontejner služeb, když čeká na doběhnutí všech vláken programu. Metoda `AppExit()` je volána opět z kontejneru a to se děje tehdy, pokud je program potřeba ukončit. Opět je zde „forward deklarace“ s kontejnerem služeb a take deklarovaná „friend class“ ze stejných důvodů jako u *IInitializable*.

```
class ServiceContainer; //zde opět forward deklarace

class IStartable {
protected:
    friend class ServiceContainer;

    virtual void Start(ServiceContainer& container) = 0;
    virtual void Wait() = 0;
    virtual void AppExit() = 0;
};
```

Ukázka 3.3 Kód rozhraní *IStartable*

ILpcManageable Toto rozhraní umožňuje označeným službám reagovat na zprávy od lektorského pracoviště. Metody `LpcSaidStart()`, `LpcSaidStop()` a `LpcSaidRestart()` se volají z kontejneru po tom co přijde start/stop/restart pokyn od lektorského pracoviště.

```
class ILpcManageable {
protected:
    friend class ServiceContainer;

    virtual bool LpcSaidStart() = 0;
    virtual bool LpcSaidStop() = 0;
    virtual bool LpcSaidRestart() = 0;
};
```

Ukázka 3.4 Kód rozhraní *ILpcManageable*

Služby jsem navrhnul a následně i implementoval tak, aby bylo co nejednodušší přidávat nové. Do přílohy B jsem přeložil kopii vývojářské příručky (originál je na fakultním GitLabu anglicky). V té se jedna z kapitol věnuje tomu, jak správně vytvořit a přidat novou službu.

3.1.2 Kontejner služeb

Nejprve bych rád vysvětlil, jak funguje přihlašování služeb do kontejneru služeb. Na výpisu kódu 3.5 lze vidět samotná metoda. Ta bere generický parametr služby, kterou se pokusí zkonstruovat a následně uložit do vnitřní mapy. Tato mapa je indexovaná typem služby a následně se používá k poskytování těchto služeb jinde v programu.

```
class ServiceContainer {
    template <class T>
    void RegisterService() {
        std::shared_ptr<T> service = std::make_shared<T>();
        if (!service)
            return;
        std::shared_ptr<IService> existing = FetchService<T>();
        if (existing)
            return;
        services[T::Type] = service;
    }

    std::map<ServiceType, std::shared_ptr<IService>> services;
}
```

Ukázka 3.5 Kód přihlášení služeb do kontejneru

Tato metoda je volána na úplném začátku programu v jádru aplikace, kdy jsou do kontejneru přihlášeny všechny potřebné služby. To zajistí, že již při inicializaci existují všechny možné služby a jsou tedy dostupné k „fetchnutí“.

Kontejner má několik veřejných metod. Generická metoda `FetchService()` vrací službu, jejíž typ se shoduje s typem, který byl předán generickým parametrem. Podle typu služby se poté kontejner dotazuje do vnitřní mapy (popsána výše) a vrací službu, kterou tam najde.

```
class ServiceContainer {
public:
    template <typename T>
    std::shared_ptr<T> FetchService() {
        return std::dynamic_pointer_cast<T>(services[T::Type]);
    }
    bool LpcSaidStart();
    bool LpcSaidStop();
    bool LpcSaidRestart();
}
```

Ukázka 3.6 Kód veřejných metod kontejneru služeb

Metody `LpcSaidStart()`, `LpcSaidStop()` a `LpcSaidRestart()` volají na vnitřně uložené služby, které dědí z rozhraní *ILpcManageable*, stejnojmenné metody. Je to jednodušší způsob, než někde v programu jednotlivě „fetchovat“ všechny služby, které toto rozhraní dědí, takhle se to provede automaticky jednou metodou v kontejneru služeb.

Dále je zde několik „protected“ metod, které jsou volány z jádra aplikace (k tomu slouží opět „friend class“ deklarace). Těmito metodami jsou `InitializeServices()`, která zavolá `Initialize()` na služby, které dědí z *IInitializable*. Metoda `StartServices()`, která zavolá `Start()` na služby, které dědí z *IStartable*. Dále je zde metoda `WaitForServices()`, díky které se počká na všechny běžící služby. A poslední metoda `AppExit()`, která na všechny spustitelné služby zavolá metodu `AppExit()`, aby se všechno správně ukončilo a nedošlo k úniku paměti.

```
class ServiceContainer {
protected:
    void InitializeServices();
    void StartServices();
    void WaitForServices();
    void AppExit();

    friend class Application; //aby na ně bylo vidět z jádra aplikace
}
```

Ukázka 3.7 Kód „protected“ metod kontejneru služeb

3.1.3 Message Handlers

„Message handler“ je třída plná kódu pro obsloužení jedné konkrétní zprávy. Existuje pro ně mateřská třída „IMessageHandler“, která je popsána na ukázce 3.8. Obsahuje virtuální metodu `HandleMessageBody()`, která bere jako parametr zprávu ke zpracování a uvnitř této metody jí následně zpracovává. Dědici této třídy mohou mít předefinovaný konstruktor¨, ve kterém přijmou kontejner služeb, diky kterému mohou získat ukazatel na služby k použití.

```
class IMessageHandler {
public:
    virtual void HandleMessageBody(const nlohmann::json& data) = 0;
}
```

Ukázka 3.8 Struktura třídy *IMessageHandler*

3.1.4 Topic workers

„Topic worker“ má na starosti delegovat zprávy na jednom tématu příslušným „message handlerům“. Má vnitřně uloženou mapu všech „message handlerů“, do které indexuje podle identifikátoru zprávy. Také má frontu všech zpráv ke zpracování, do které se dá přidávat metodou `AddToQueue()`. Tato fronta je vybavená synchronizačními zámky, aby se nestala nějaká vícevláknová neplecha.

```
class TopicWorker {
public:
    void AddToQueue(const nlohmann::json& data);
private:
    void ProcessMessage();
    AsyncQueue<nlohmann::json> messageQueue;
    std::map<MessageID, std::shared_ptr<IMessageHandler>> messageHandlers;
}
```

Ukázka 3.9 Kód práce „topic workera“

Samotné delegování zpráv funguje tak, že postupně prochází frontu přijatých zpráv. Poté podle identifikátoru zprávy najde k němu přidružený „message handler“ a zprávu mu předá. Na ukázce 3.10 je zjednodušená forma tohoto algoritmu.

```
void TopicWorker::ProcessMessage() {
    while(true){
        nlohmann::json j = messageQueue.Pop();
        MessageID messageId = j.at("NID_MESSAGE").get<MessageID>();
        std::shared_ptr<IMessageHandler> handler = messageHandlers.at(messageId);
        handler->HandleMessageBody(j);
    }
}
```

Ukázka 3.10 Kód práce „topic workera“

3.2 CEM (Common Enums & Messages)

Duležitá část mé práce na projektu bylo sjednocení komunikace do repozitáře CEM. Rád bych zde shrnul důležité části, které se týkají této práce.

```
cem
└── Other
    ├── Telegram
    ├── Balise
    ├── BaliseGroup
    ├── GradientSection
    └── SpeedSection
```

Ukázka 3.11 Struktura adresáře „Other“ v CEMu

3.2.1 Prvky pro popis trati

V této části jsou prvky, které se týkají popisu trati a traťové databáze.

Telegram Pomocná třída pro reprezentaci obsahu balízového telegramu (viz. 1.10). Na výpisu 3.12 je zkrácená ukázka kódu třídy *Telegram*.

```
class Telegram {
public:
    uint16_t Q_UPDOWN;
    uint16_t M_VERSION;
    uint16_t Q_MEDIA;
    uint16_t N_PIG;
    uint16_t N_TOTAL;
    uint16_t M_DUP;
    uint16_t M_MCOUNT;
    uint32_t NID_C;
    uint32_t NID_BG;
    uint16_t Q_LINK;
}
```

Ukázka 3.12 Zjednodušený kód třídy *Telegram*

Balise Reprezentuje jednu balízu na trati. Tato balíza má identifikátor typu `string`, nese v *telegram* a také má absolutní pozici na trati uloženou ve třídě *Distance* (viz 2.2.2.2). Uložená data ve třídě jsou vidět na ukázce 3.13.

```
class Balise {
private:
    std::string baliseID;
    Telegram telegram;
    Distance pos;
}
```

Ukázka 3.13 Zjednodušený kód třídy *Balise*

BaliseGroup Tato třída reprezentuje balízovou skupinu. Ta, jak již víme, může být tvořena více balízami. Proto má vnitřně uložený `vector` balíz, identifikátor skupiny a opět absolutní vzdálenost ve formě *Distance*. Znovu je zde ukázka 3.14 s implementací.

```
class BaliseGroup {
private:
    uint32_t baliseGroupId;
    std::vector<std::shared_ptr<Balise>> balises;
    Distance absPosition;
};
```

Ukázka 3.14 Zjednodušený kód třídy *BaliseGroup*

GradientSection Třída *GradientSection* má za úkol zaobalit data spojená s jednou sekcí stoupání na trati. K tomuto potřebuje nést informace, odkud kam se tato oblast táhne, a její stoupání. Vzdálenosti „startPosition“ a „endPosition“ jsou absolutní vzdálenosti začátku a konce sekce. „incline“ určuje míru stoupání v promilích a „dir“ určuje, zda tato míra je z kopce nebo do kopce. Na ukázce 3.15 je popis uložených dat ve třídě.

```
class GradientSection {
private:
    Distance startPosition;
    Distance endPosition;
    uint16_t incline;
    uint16_t dir;
};
```

Ukázka 3.15 Zjednodušený kód třídy *GradientSection*

SpeedSection Pro ukládání rychlostních omezení je zde třída *SpeedSection*. Ta má v sobě opět uloženou informaci, odkud kam platí tyto omezení a výši rychlostního omezení. Vzdálenosti „startPosition“ a „endPosition“ jsou absolutní vzdálenosti začátku a konce sekce. Proměnná „speed“ určuje výši tohoto omezení. Na ukázce 3.16 je popis dat uložených ve třídě.

```
class SpeedSection {
private:
    Distance startPosition;
    Distance endPosition;
    uint16_t speed;
};
```

Ukázka 3.16 Zjednodušený kód třídy *SpeedSection*

3.2.2 Zprávy

Všechny významy proměnných jsou shrnuty v návrhu, proto zde nebudu opakovat význam proměnných, ale pouze jejich implementační specifika. U tříd, které dědí nebudu opakovat kód a použiji symbol „. . . “ pro označení částí, které vynechávám kvůli podobnosti.

Všechny zprávy jsou ve společném repositáři viz diagram struktury adresáře 3.17.

```
cem
├── Enums
│   └── MessageID ..... Identifikátor zprávy
└── Messages
    ├── Message ..... Mateřská třída pro všechny zprávy
    ├── Evc
    │   └── Rbc
    │       ├── MessageToRbc ..... Abstraktní třída pro zprávy EVC->RBC
    │       ├── MessageToRbcWithPackets
    │       └── Zde jsou všechny zprávy od EVC do RBC
    └── Rbc
        ├── MessageFromRbc ..... Abstraktní třída pro zprávy RVC->EVC
        ├── MessageFromRbcWithPackets
        └── Evc
            └── Zde jsou všechny zprávy od RBC do EVC
```

Ukázka 3.17 Struktura zpráv v CEMu

Jak jsem již vysvětlil v návrhu (viz. 2.2.3), pro všechny zprávy platí, že jsou potomkem třídy *Message*. Hierarchie všeho tohoto dědění je popsaná v návrhové části v diagramu 2.3.

Kód třídy *Message* je vyobrazen na ukázce 3.18. Má virtuální destruktor, aby ho její potomci mohli přetížit a vyhnout se tak unikům paměti. Dále má operátor „==“ pro porovnávání, metody `to_json` a `from_json`, které umožňuji serializaci a deserializaci na formát json. Ty jsou opět virtuální, aby potomci této třídy mohli přidat proměnné k serializaci. V této třídě je taky jediná proměnná společná pro všechny zprávy a to „NID_MESSAGE“. Ta je typu *MessageID*, což je výčtový typ definovaný v repositáři CEM.

```
class Message {
public:
    virtual ~Message() = default;

    MessageID GetMessageID() const;

    bool operator==(const Message& rhs) const;
    virtual nlohmann::json to_json() const;
    virtual void from_json(const nlohmann::json& j);

protected:
    MessageID NID_MESSAGE;
};
```

Ukázka 3.18 Kód třídy *Message*

Z této třídy poté dědí následující dvě třídy:

MessageFromRbc Tato abstraktní třída je společná pro všechny zprávy směřované od RBC do EVC. Rozšiřuje třídu *Message* o několik dalších proměnných. Její zjednodušená implementace je na ukázce 3.19.

```
class MessageFromRbc : public Message {
public:
    ... //Zde jsou metody from/to_json, ==, destuktor a gettery
protected:
    uint32_t L_MESSAGE;
    uint32_t T_TRAIN;
    uint16_t M_ACK;
    uint32_t NID_LRBG;
};
```

Ukázka 3.19 Zjednodušený kód třídy *MessageFromRbc*

MessageToRbc Tato třída je pro směr opačný, EVC do RBC. Opět rozšiřuje *Message* o proměnné, které jsou společné pro tento směr toku zpráv. Zkrácenou implementaci lze vidět na ukázce 3.20.

```
class MessageToRBC : public Message {
public:
    ... //Zde jsou metody from/to_json, ==, destuktor a gettery
protected:
    uint32_t L_MESSAGE;
    uint32_t T_TRAIN;
    uint32_t NID_ENGINE;
};
```

Ukázka 3.20 Zjednodušený kód *MessageToRBC*

Dále z těchto zpráv dědí další dvě zprávy, které je rozšiřují o možnost mít pakety. Tyto zprávy jsou *MessageFromRbcWithPackets* a *MessageToRBCWithPackets*. Jejich implementace je téměř totožná. Dědí z výše uvedených dvou tříd a přidávají pouze vektor paketů ve formátu `std::vector<std::shared_ptr<Packet>> m_Packets;`.

Všechny zprávy pro přenos dat mezi RBC a EVC jsem přepsal do tohoto nového formátu. To velmi zjednodušuje komunikaci mezi těmito komponentami a zároveň tvoří pevný základ pro přidávání zpráv dalších. Při tomto jsem také v CEMu vytvořil několik pomocných tříd na ukládání dat, které se následně posílají ve zprávách. Všechny tyto třídy se nachází ve složce „Other“ uvnitř CEMu.

3.2.3 Pakety

Další důležitou části komunikace jsou pakety. Jdou ruku v ruce se zprávami a jsou si také dost podobné implementací. Všechny pakety se také nachází v CEMu (viz diagram adresáře 3.21).

```
cem
├── Enums
│   └── PacketID ..... Identifikátor paketu
└── Packets
    ├── Factory
    │   └── PacketFactory
    ├── Packet ..... Mateřská třída pro pakety
    └── Zde jsou všechny pakety
```

Ukázka 3.21 Struktura paketů v CEMu

Jejich implemetace se věrně drží návrhu (2.2.4). Diagram dědičnosti je v návrhu na obrázku 2.4. Existuje jedna mateřská třída *Paket*. Její implementace je na ukázce kódu 3.22. Je zde virtuální destruktor ze stejného důvodu jako je tomu u zpráv, to samé platí pro metody `to_json` a `from_json` a také pro operátor „==“. Obsahuje proměnnou „NID_PACKET“, která je typu *PacketID*. Tento typ je opět definovaný v CEMu.

```
class Packet {
public:
    virtual ~Packet() = default;

    PacketID GetNidPacket() const;
    uint16_t GetLPacket() const;

    bool operator==(const Packet& rhs) const;
    virtual nlohmann::json to_json() const;
    virtual void from_json(const nlohmann::json& j);

protected:
    PacketID NID_PACKET;
    uint16_t L_PACKET;
};
```

Ukázka 3.22 Kód třídy *Packet*

Další důležitou třídou je *PacketFactory*. Tato třída vznikla během implementace při serializaci a deserializaci. Její kód je na ukázce 3.23. Je totiž potřeba mít zkonstruovaný paket podle typu, aby se mohl naplnit daty. Tato třída umožňuje přesně toto. Má pouze jednu statickou metodu, která podle typu vytvoří příslušný prázdný paket.

```
class PacketFactory {
public:
    static std::shared_ptr<Packet> CreateEmptyPacketById(PacketID packetId);
};
```

Ukázka 3.23 Kód třídy *PacketFactory*

3.3 Reimplementace funkcionalit RBC

Prvním úkolem je zachovat všechny stávající funkce RBC. Proto se v této sekci budu věnovat reimplementaci dosavadních funkcionalit.

3.3.1 Interakce s traťovou databází

Databáze je typu „MySQL“ a proto použiji (stejně jako staré RBC) knihovnu „MySQLCpp-Connector“, která sice není nejkrásnější, ale jiná alternativa prakticky neexistuje. Na správu komunikace s databází vznikla služba *DatabaseService*.

```
src
└── Services
    ├── Database
    │   ├── Utils
    │   │   └── DatabaseQueries
    │   └── DatabaseService
    ├── TrackData
    │   └── TrackDataService
    └── TrainRegistry
        ├── Utils
        │   └── Train
        └── TrainRegistryService
```

Ukázka 3.24 Struktura adresáře služeb v RBC

Než se však budu věnovat implementaci samotné databázové služby, je nejdříve potřeba naimplemetovat pomocné služby pro ukládání dat o trati a vlacích. K tomuto slouží následující dvě služby:

TrackDataService Tato služba má za úkol uchovávat data o trati, které budou načteny z databáze. Balízové skupiny si vnitřně uchovává jako mapu indexovanou identifikátorem té balízové skupiny. Zbytek dat si uchovává v jednoduchém vektoru. Pro každý typ dat je zde „setter[1]“ a „getter[2]“, a také spousta dalších „getterů“ pro usnadnění manipulace s daty (např. pro získání balíz seřazených podle vzdálenosti) .

```
class TrackDataService : public ITrackDataService {
public:
    // Spoustu „getterů" a „setterů" na uložená data
private:
    std::map<uint32_t, std::shared_ptr<BaliseGroup>> baliseGroups;
    std::vector<std::shared_ptr<SpeedSection>> speedSections;
    std::vector<std::shared_ptr<GradientSection>> gradientSections;
    std::vector<std::shared_ptr<Rule>> rules;
}
```

Ukázka 3.25 Zjednodušený kód *TrackDataService*

[1] metoda pro nastavení hodnoty dat
[2] metoda pro získání hodnoty dat

TrainRegistryService V této službě se uchovávají data o vlacích. Je zde pomocná třída *Train*, která v sobě uchovává všechny informace o jednom vlaku. V rámci této třídy jsou zde dvě struktury:

TrainPosition Tato struktura ukládá poziční data o vlaku, které posílá EVC v rámci pravidelného oznamování polohy.

```
struct TrainPosition {
    uint32_t NID_LRBG;
    Distance D_LRBG;
    Direction Q_DIRLRBG;
    Direction Q_DLRBG;
    Distance L_DOUBTOVER;
    Distance L_DOUBTUNDER;
    QLength m_Length;
    Direction Q_DIRTRAIN;
};
```

Ukázka 3.26 Kód struktury *TrainPosition*

TrainData Druhá struktura je *TrainData*. Ta obsahuje ostatní (nepoziční) parametry vlaku. Tyto informace dodá EVC v rámci procedury SoM.

```
struct TrainData {
    uint32_t T_TRAIN;
    uint32_t NID_OPERATIONAL;
    uint16_t NC_TRAIN;
    uint32_t L_TRAIN;
    uint16_t V_MAXTRAIN;
    uint16_t M_LOADINGGAUGE;
    double M_AXLELOAD;
    bool M_AIRTIGHT;
    std::vector<uint16_t> M_TRACTIONs;
    std::vector<uint16_t> NID_STMs;
};
```

Ukázka 3.27 Kód struktury *TrainData*

Třída *Train* má svůj identifikátor, status připojení, který může být „ON_MISSION“, „ESTABLISHED“ nebo „NOT_ESTABLISHED“ a výše zmíněné dvě struktury.

```
class Train {
private:
    uint32_t id;
    ConnectionState state;
    TrainPosition trainPosition;
    TrainData trainData;
}
```

Ukázka 3.28 Zjednodušený kód třídy *Train*

TrainRegistryService má uloženou mapu těchto vlaků indexovanou jejich identifikátorem. Poskytuje spoustu veřejných metod pro různé operace s těmito vlaky. Nejdůležitejší operace

jsou: přidání vlaku pomocí metody `RegisterTrain()`, získání vlaku metodou `GetTrain()` a smazání vlaku metodou `RemoveTrain()`.

```
class TrainRegistryService : public ITrainRegistryService, public IInitializable {
public:
    void Initialize(ServiceContainer& container) override;
    bool RegisterTrain(uint32_t id) override;
    bool TrainRegistred(uint32_t id) override;
    Train& GetTrain(uint32_t id) override;
    bool RemoveTrain(uint32_t id) override;
private:
    std::map<uint32_t, Train> trains;
}
```

Ukázka 3.29 Zjednodušený kód třídy *TrainRegistryService*

Nyní jsem již vyjasnil všechny pomocné třídy a je tedy čas na databázovou službu. Tato služba je trochu komplexnější a proto zde popíšu jen základní myšlenky. V digramu 3.24 si můžeme všimnout, že je soubor „DatabaseQueries“. Tento soubor obsahuje SQL dotazy na databázi pro balízy, rychlostní omezení i stoupání trati. Třída „DatabaseService“ dědí jak z IInitializable, tak i z *ILpcManageable*. To je proto, protože načítat data z databáze chceme až poté, co nám lektorské pracoviště pošle všechny údaje k databázi. Proto se to děje až v reakci na zprávu „Start“ od LPC v rámci metody `LpcSaidStart()`.

Metoda `Connect()` se zkusí připojit k databázi. `LoadDataFromDatabase()` se pokusí data načíst voláním výše zmíněných SQL skriptů. V rámci inicializace této služby se musí načíst odkaz na třídu *TrackDataService* respektive na její rozhraní. Do ní se totiž data načtené z databáze uloží, aby se s nimi mohlo později operovat.

```
class DatabaseService : public IDatabaseService, public IInitializable,
                        public ILpcManageable {
public:
    bool LpcSaidStart() override {
        if(!Connect()) return false;
        if(!LoadDataFromDatabase()) return false;
        return true;
    }

private:
    bool Connect();
    bool LoadDataFromDatabase();

    ITrackDataService* trackDataService;
}
```

Ukázka 3.30 Velmi zjednodušený kód třídy *DatabaseService*

3.3.2 Konfigurace

Jak jsem dříve zmínil (viz. 2.1.5), jeden z důležitých požadavků je konfigurovatelnost. Proto jsem vytvořil službu „ConfigurationService“. Její metoda `FetchConfiguration()` bere jako argument typ konfigurace jakou má vrátit. Typy konfigurace jsou:

ListenerConfiguration Tato konfigurace obsahuje všechny témata („topics“), na kterých by měla poslouchat služba *MqttListenerService*.

NationalValuesConfiguration V této konfiguraci jsou sepsány všechny relevantní národní hodnoty. Nyní je zde jen jediná proměnná „D_NVSTFF“, která určuje vzdálenost, po kterou je povoleno jet v módu SR.

Uvnitř této metody se volá několik pomocných metod. `ConfigurationExists()` vrací, jestli konfigurace s daným typem existuje či nikoliv. `CreateConfiguration()` vytvoří konfiguraci s daným typem. `GetConfiguration()` získá existující konfiguraci. `GenerateDefaultConfiguraion()` vytvoří konfiguraci s výchozími parametry.

```
class ConfigurationService : public IConfigurationService {
public:
    std::shared_ptr<IConfiguration> FetchConfiguration(ConfigType type) override {
        // Pokud již existuje konfigurace, vrať tu
        // Pokud ne, vytvoří novou a vrátí jí
    }
private:
    std::shared_ptr<IConfiguration> GetConfiguration(ConfigType type);
    bool ConfigurationExists(ConfigType type);
    void CreateConfiguration(ConfigType type);
    std::shared_ptr<IConfiguration> GenerateDefaultConfiguration(ConfigType type);
};
```

Ukázka 3.31 Zjednodušený kód třídy *ConfigurationService*

Všechny konfigurace se ukládají ve formátu json. Ty mají vždy prioritu a kdykoliv se za běhu chce získat, tak se nejprve zkontroluje, zda tato json konfigurace existuje. Pokud ne, tak se vytvoří nová s předem definovanými hodnotami a uloží se i pro další použití.

Pro konfigurace opět existuje mateřská třída *IConfiguration*. Její kód je popsán na 3.32. Metoda `to_json` převede konfiguraci do json formátu. `from_json` udělá přesný opak, převede json na konfiguraci. Poslední metoda `Type()` vrací typ konfigurace, kde tato mateřská třída, vrací typ „None“. Má také virtuální destruktor, aby nedošlo k únikům paměti.

```
class IConfiguration {
public:
    virtual ~IConfiguration() = default;
    virtual void from_json(const nlohmann::json& j) = 0;
    virtual nlohmann::json to_json() const = 0;
    virtual ConfigType Type() { return ConfigType::None; }
};
```

Ukázka 3.32 Kód třídy *IConfiguraion*

3.3.3 Komunikace přes MQTT

Komunikacei přes MQTT jsem vyřešil dvěma službami. Jedna, která zprávy přijímá a druhá odesílá. Obě pro to využívají knihovnu „Eclipse Mosquitto“.

Díky nim (a také novým zprávám a paketům) je kompletně vyřešená komunikace mezi komponentami a celý proces je nyní velmi přímočarý. Není potřeba v každé komponentě definovat zprávy a pakety. I služby pro komunikaci jsou v ostatních komponentách (vzhledem k tomu, že využívají stejnou architekturu) téměř identické.

MqttListenerService Tato služba má na starosti poslouchat komunikaci a přijímat zprávy určené pro RBC. Vnitřně má mapu „topic workerů“ indexovanou „topicem“. Všechny zprávy, které odposlechne, přesměrovává do odpovědných „topic workerů“ skrze tuto mapu. Toto chování je naznačené v ukázce implemetace 3.33 v metodě `MqttCallback()`.

Kód je v reálné implementaci poněkud složitý (kvůli implementaci MQTT knihovny), proto jsem zde jen popsal co nejdůležitější metody krok po kroku dělají. V rámci inicializace si služba natahá ostatní služby, které potřebuje, hlavně konfigurační službu. Z té totiž následně při přijetí zprávy „start“ od LPC potřebuje získat „topicy“, na kterých má poslouchat.

```
class MqttListenerService : public IMqttListenerService, public IStartable,
                            public IInitializable, public ILpcManageable {
public:
    void Initialize(ServiceContainer& container) override {
        // Natáhne si z kontejneru všechny potřebné služby
        // Připojí se k MQTT brokerovi
        // Začne poslouchat na topicu LPC/RBC
        // Zaregistruje do mapy topic workera na topic LPC/RBC
    }
    bool LpcSaidStart() override {
        // Načte z konfigurace všechny topicy na kterých má poslouchat
        // Začne na nich poslouchat
        // Zaregistruje do mapy ke každému topicu nový topic worker
    }
    void MqttCallback(struct mqtt, struct mqtt_message) {
        // Naparsuje zprávu z „mqtt_message" na json
        // Podle topicu najde v mapě odpovědného „topic workera"
        // Přida tuto zprávu tomuto „workerovi" do fronty ke zpracování
    }

private:
    std::map<Topic, std::shared_ptr<TopicWorker>> topicWorkers;
}
```

Ukázka 3.33 Velmi zjednodušený kód třídy *MqttListenerService*

MqttPublisherService Tato služba naopak zprávy publikuje. V inicializaci si opět načte potřebné služby a připojí se k MQTT brokerovi. Následně metodě `Publish()` bere jako argument ukazatel na námi vytvořenou zprávu. Tuto zprávu převede do formátu json a odešle jí přes MQTT. Tyto procesy jsou opět v reálné implementaci mírně složitější a proto v ukázce 3.34 znovu jen zjednodušeně popisuji kroky, které metody dělají po zavolaní.

```
class MqttPublisherService : public IMqttPublisherService, public IInitializable {
public:
    void Initialize(ServiceContainer& container) override {
        // Natáhne si z kontejneru všechny potřebné služby
        // Připojí se k MQTT brokerovi
    }
    void Publish(std::shared_ptr<Message> msg) override {
        // Převede si zprávu na json
        // Odešle zprávu přes MQTT
    }
}
```

Ukázka 3.34 Velmi zjednodušený kód třídy *MqttPublisherService*

3.3.4 Udělování povolení k jízdě

Pro funkci vytváření povolení k jízdě vznikla služba „MAGeneratorService“. Tato služba umí vygenerovat dva typy povolení k jízdě. Plnohodnotné a poté autorizaci k pohybu v módu SR.

Generování autorizace v SR je jednodušší a proto jí shrnu první. RBC nejdříve musí vypočítat, kde přesně se vlak nachází (tato poloha může být neznámá). Poté, pokud zná polohu, může (ale nemusí) najít balízy, na které očekává, že vlak v módu SR narazí. Tyto balízy případně přibalí do paketu 63: „SRAuthBalises“. Poté si z konfigurace vytáhne hodnotu „D_NVSTFF“, která určuje po jakou vzdálenost může vlak jet v SR (viz 3.3.2). Následně vytvoří zprávu 2: „SRAuthorizationMessage“, ve které naplní proměnnou „D_SR“ hodnotou z konfigurace. Pokud je poloha neznámá, nastaví „D_SR“ na maximální hodnotu.

Tvorba kompletního povolení k jízdě je komplikovanější a proto jí rozdělím do několika kroků.

1. **Načtení dat o vlaku:** RBC si musí zjistit všechny informace, co jsou potřeba k vytvoření plnohodnotného povolení k jízdě. Nejdůležitější je určitě poloha vlaku. Je potřeba znát poslední balízu kterou vlak přejel („NID_LRBG“) ale také vzdálenost od ní („D_LRBG“).

2. **Vytvoření paketů:** Následně přichází na řadu vygenerování všech potřebných paketů.

   a. Povinný paket 15: „MovementAuthority“ se vytvoří docela jednoduše. Podle pravidel simulace (dané od LPC) se najde nejzazší pravidlo, jejichž úsek stále odpovídá lokaci vlaku a použije se vzdálenost do konce tohoto pravidla. Tato vzdálenost se přepočítá aby se odpovídala od poslední přejeté balízy (LRBG) a následně se zapíše do proměnné „L_ENDSECTION“ v paketu a tím je paket hotov.

   b. Další je na řadě paket 27: „Static speed profile“. Při tvorbě tohoto paketu je nutné koukat do dat načtených z databáze, konkrétně na traťová rychlostní omezení. Tyto omezení se jedno po druhém uloží do paketu a odešlou se v rámci něj.

   c. Poslední paket 21: „Gradient profile“ je opět záludný tím, že se musí koukat do dat z databáze, konkrétně na profil trati. Proto, podobně jako s předešlým paketem, se postupně musí informace uložit do nového paketu a odeslat v něm.

3. **Vytvoření zprávy:** Poslední je na řadě zabalení všech paketů a dat do jedné zprávy. Vytvoří se proto nová zpráva a přilepí se jí na konec všechny vytvořené pakety. Povinný je jen paket 15 ale v naší simulaci posíláme vždy všechny 3.

V ukázce kódu 3.35 lze vidět zjednodušená implementace služby, která tuto funkcionalitu zaobaluje. Služba *MAGeneratorService* opět dědí z rozhraní *IInitializable*, protože pro generování povolení potřebuje jak „TrainRegistryService“ pro načítání dat o vlacích, tak „TrackDataService“ pro informace o rychlostních omezeních na trati, o balízách a o profilu trati. Metoda `GenerateSRMessage` (jak název napovídá) generuje již zmíněnou autorizaci jízdy v módu SR. Má i přepínač „withSRAuthBalises“, kterým lze určit, zda je žádoucí generovat seznam balíz, na které má vlak narazit. Dále metoda `GenerateMAMessage` již vytváří to plnohodnotné povolení k jízdě. Provolává pomocné metody `Generate(MA/SSP/GP)Packet` a nakonec vytvoří zprávu o povolení k jízdě.

```
class MAGeneratorService : public IMAGeneratorService, public IInitializable {
public:
    void Initialize(ServiceContainer& container) override {
        // Natáhne si z kontejneru všechny potřebné služby
    }
    std::shared_ptr<Message> GenerateMAMessage(uint32_t trainId) override {
        // Získá informace o vlaku podle argumentu „trainId"
        // Podle těchto informací zavolá metody Generate(MA/SSP/GP)Packet
        // Seskupí tyto data do zprávy 3: „MovementAuthorityMessage"
        // Vrátí jako návratovou hodnotu tuto zprávu uloženou v Message
    }

    std::shared_ptr<Message> GenerateSRMessage(uint32_t trainID,
                                               bool withBalises) override {
        // Získá pozici vlaku podle argumentu „trainId"
        // Pokud je potřeba vytvoří seznam očkávaných balíz
        // Všechno obalí do zprávy 2: „SRAuthorizationMessage"
    }
private:
    std::shared_ptr<Packet> GenerateMAPacket(const Train& train) {
        // Vytvoří podle dostupných dat paket 15: „Movement Authority"
    }
    std::shared_ptr<Packet> GenerateSSPPacket(const Train& train) {
        // Vytvoří podle dostupných dat paket 27: „Static speed profile"
    }
    std::shared_ptr<Packet> GenerateGPPacket(const Train& train) {
        // Vytvoří podle dostupných dat paket 21: „Gradient profile"
    }
}
```

Ukázka 3.35 Velmi zjednodušený kód třídy *MAGeneratorService*

3.3.5 Procedura SoM

Implemetace této procedury je poměrně přímočará, jelikož vychází přímo z analýzy (viz 1.1.6.1). Pro každou zprávu, která je očekávaná od EVC, je nový „message handler“.

Zpráva 155: „Initiation of communication session“ Pro tuto zprávu vznikla třída *InitiationOfCommunicationMessageHandler*. Ta zkontroluje obsah zprávy a také jestli vlak se stejným identifikátorem není již připojený. Následně ho přihlásí a odpoví zprávou 32: „Configuration Determination“. Nastíněná impementace je na ukázce 3.36.

```
class InitiationOfCommunicationMessageHandler : public IMessageHandler {
public:
    InitiationOfCommunicationMessageHandler(ServiceContainer& container) {
        // Natahá si potřebné služby
    }
    void HandleMessageBody(const nlohmann::json& data) override {
        // Zkontroluje tvar zprávy
        // Ujistí se že vlak se stejným ID ještě není registrovaný
        // Zaregistruje nový vlak do TrainRegistryService
        // Odpoví zprávou 32: „Configuration Determination"
    }
};
```

Ukázka 3.36 Zjednodušený kód třídy *InitiationOfCommunicationMessageHandler*

Zpráva 159: „Session established“ Na tuto zprávu reaguje „SessionEstablishedMessageHandler“. Ten po přijetí nastaví status vlaku jako *ESTABLISHED*. Opět zjednodušená implementace je na ukázce 3.37.

```
class SessionEstablishedMessageHandler : public IMessageHandler {
public:
    SessionEstablishedMessageHandler(ServiceContainer& container) {
        // Natahá si potřebné služby
    }
    void HandleMessageBody(const nlohmann::json& data) override {
        // Zkontroluje tvar zprávy
        // Nastaví status vlaku jako ESTABLISHED
    }
};
```

Ukázka 3.37 Zjednodušený kód třídy *SessionEstablishedMessageHandler*

Zpráva 157: „SoM Position Report“ Pro reakci na tuto zprávu vznikl *SoMPositionReportMessageHandler*. Ten nejprvě opět zkontoluje obsah zprávy. Následně se podívá na proměnnou *Q_STATUS* a pokud je *INVALID* nebo *UNKNOWN*, tak přijme vlak. To udělá tak, že si nejprve poziční data o něm uloží a poté odpoví zprávou 41: „Train Accepted“. Tím se sjednotí data o pozici vlaku jak na straně RBC tak i EVC. Jeho zjednodušená implementace je na ukázce 3.38.

```
class SoMPositionReportMessageHandler : public IMessageHandler {
public:
    SoMPositionReportMessageHandler(ServiceContainer& container) {
        // Natahá si potřebné služby
    }
    void HandleMessageBody(const nlohmann::json& data) override {
        // Zkontroluje tvar zprávy
        // Zkontroluje Q_STATUS (validitu dat)
        // Pokud jsou INVALID nebo UNKNOWN tak se rozhodne přijmout vlak
        // Nejdříve si uloží poziční data o vlaku do TrainRegistry
        // Následně pošle zprávu 41: „Train Accepted"
    }
};
```

Ukázka 3.38 Zjednodušený kód třídy *SoMPositionReportMessageHandler*

Zpráva 129: „Validated Train Data“ Pro tuto zprávu je zde *ValidatedTrainDataMessageHandler*. Ten po přijetí zprávy zkontroluje její obsah, data uloží a potvrdí přijetí dat zprávou 8: „Ack Train Data“. Zjednodušený kód je na ukázce 3.39.

```
class ValidatedTrainDataMessageHandler : public IMessageHandler {
public:
    ValidatedTrainDataMessageHandler(ServiceContainer& container) {
        // Natahá si potřebné služby
    }
    void HandleMessageBody(const nlohmann::json& data) override {
        // Zkontroluje tvar zprávy
        // Uloží si všechna data do TrainRegitry
        // Odpoví zprávou 8: „Ack Train Data"
    }
};
```

Ukázka 3.39 Zjednodušený kód třídy *ValidatedTrainDataMessageHandler*

3.4 Nové funkcionality

V této sekci se zaměřím na funkce, které vznikly nově v rámci této práce. Většina z nich vyplývá z požadavků v analytické části práce (viz 1.4).

3.4.1 Nová pravidla scénářů jízdy

V rámci návrhu (viz. 2.2.5) jsem představil nové řešení scénářů a pravidel jízdy. Tento návrh jsem naimplementoval a zde tuto implementaci popíši. Celá implementace se nachází v repositáři CEM (viz 3.40).

Simulátor ETCS se každým rokem rozrůstá a proto jsem i nová pravidla navrhnul s myšlenkou jednoduché rozšiřitelnosti. V příloze B je vývojářská příručka, ve které jednou z jejích částí je i postup, jak přidat nové pravidla scénářů jízdy.

```
cem
├── Enums
│   └── RuleType ..... Výčtový typ určující typ pravidla
└── Rules
    ├── Rule ..... Mateřská třída pro pravidla
    ├── MovementAuthorityRule
    ├── TAFShowRule
    ├── ConditonalEmergencyStopRule
    └── UnconditionalEmergencyStopRule
```

Ukázka 3.40 Scénáře a pravidla v CEMu

Nejdříve je zde výčtový typ *RuleType*. Ten má pro každý typ pravidla jednu hodnotu. Používá se při „parsování“ pravidel, když přijdou od LPC ve formátu json.

Jak jsem popsal v návrhu, je zde mateřská třída *Rule*. Ta obsahuje již zmíněný *RuleType* a také jednoznačný identifikátor pravidla. Podobně jako je to u zpráv a paketů, tak je i zde přetížený operátor == pro porovnání. Také tu jsou metody `to_json` a `from_json`, které stejně jako u zpráv převedou data z/do formátu json. Tyto metody jsou virtuální, protože následní dědici této třídy si mohou přidat nějaké proměnné a poté bude potřeba je přetížit. Má také virtuální destruktor ke správnému hospodaření s pamětí.

```
class Rule {
public:
    virtual ~Rule() = default;

    uint32_t GetRuleID() const;
    RuleType GetRuleType() const;

    bool operator==(const Rule& rhs) const;
    virtual nlohmann::json to_json() const;
    virtual void from_json(const nlohmann::json& j);
protected:
    uint32_t ruleID;
    RuleType ruleType;
};
```

Ukázka 3.41 Kód třídy *Rule*

Protože kód pravidel jízdy je velmi podobný, všechno ukáži na pravidle *MovementAuthorityRule*. U zbytku uvedu jen v čem se liší. Použiji symbol „...“ pro označení částí, které vynechávám kvůli podobnosti.

Všechny dědici této třídy přetěžují její destruktor. Dále také metody `to_json` a `from_json` a operátor ==. Význam všech proměnných je lépe popsán v kapitole o návrhu těchto pravidel (viz 2.2.5).

Movement authority

První z pravidel určuje sekce pro udělování oprávnění k jízdě. Přibyly zde dvě proměnné „startPosition“ a „endPosition“, které jsou obě typu *Distance* (viz. 2.2.2.2). Proměnná „startPosition“ indikuje začátek sekce a „endPosition“ její konec.

```
class MovementAuthorityRule : public Rule {
public:
    MovementAuthorityRule() = default;
    MovementAuthorityRule(uint32_t id, Distance startPos, Distance endPos);
    ~MovementAuthorityRule() override = default;

    Distance GetStartPosition() const;
    Distance GetEndPosition() const;

    nlohmann::json to_json() const override;
    void from_json(const nlohmann::json& j) override;
    bool operator==(const MovementAuthorityRule& rhs) const;

private:
    Distance startPosition;
    Distance endPosition;
};
```

Ukázka 3.42 Kód třídy *MovementAuthorityRule*

Unconditional emergency stop

Toto pravidlo určuje místo pro odeslání zprávy o bez-kondičním nouzovém brzdění. Má proměnnou „engagePosition“, která je typu *Distance*. Ta určuje pozici, kde se tato zpráva odešle.

```
class UnconditionalEmergencyStopRule : public Rule {
public:
    ...
private:
    Distance engagePosition;
};
```

Ukázka 3.43 Zkrácený kód třídy *UnconditionalEmergencyStopRule*

Conditional emergency stop

Toto pravidlo je velmi podobné jako minulé. Navíc přibyla proměnná „notifyPosition“, která je také opět typu *Distance* a určuje pozici odeslání zprávy o kondičním brzdění. A místo „engagePosition“ je „engageDistance“, protože již není absolutní ale relativní k „notifyPosition“. Ta určuje vzádlenost od notifikace k místu brdění.

```
class ConditionalEmergencyStopRule : public Rule {
public:
    ...
private:
    Distance notifyPosition;
    Distance engageDistance;
};
```

Ukázka 3.44 Zkrácený kód třídy *ConditionalEmergencyStopRule*

Track ahead free

Poslední pravidlo určuje oblast, kde se má strojvedoucímu vyobrazit okno s Track Ahead Free (TAF). Proměnné „notifyPosition“ a „displayLength“ jsou opět typu *Distance*. Ty určují pozici zobrazení toho okna a vzdálenost, po kterou má být zobrazeno.

```
class TAFShowRule : public Rule {
public:
    ...
private:
    Distance notifyPosition;
    Distance displayLength;
};
```

Ukázka 3.45 Zkrácený kód třídy *TAFShowRule*

3.4.2 Parsování a kontrolování pravidel

Při přijímaní pravidel od LPC hraje hlavní roli *TrackRulesMessageHandler*. Ten v rámci své metody `HandleMessageBody()` (kterou má díky 3.1.3) pravidla uloží do *TrackDataService* (3.3.1).

Tyto pravidla se v rámci *TrackDataService* ukládají schovaná v jejich společném rozhraní *Rule*. Kdykoliv je ale potřeba dostat konkrétní pravidla, na příklad *MovementAuthorityRule*, tak *TrackDataService* musí tyto pravidla projet všechny a vrátit jen ty, kterých *RuleType* odpovídá požadovanému typu. Tento problém se vyskytuje i u zpráv a paketů, ale tady je trochu více očividné a proto jsem to zde vyzdvihl.

Pro správné fungování simulace je potřeba periodicky kontrolovat zda se vlak nedostal do sekce, kde se mění nějaká pravidla simulace. Nejlepší příklad je asi pravidlo pro bez-kondiční nouzové zastavení. Kdykoliv vlak ohlásí polohu, tak proběhne kontrola, jestli se tímto pohybem vlak nedostal přes bod, kde má být poslán příkaz k zastavení.

K tomuto jsem vytvořil novou službu *RuleCheckingService*. Její zjednodušená implementace se nachází na ukázce 3.46. Metoda `CheckRules()` se volá ze třídy *TrainPositionReportMessageHandler*, kdykoliv přijde od vlaku ohlášení o poloze.

```
class RuleCheckingService : public IRuleCheckingService, public IInitializable {
public:
    void Initialize(ServiceContainer& container) override {
        // Natahá si potřebné služby
    }
    void CheckRules(uint32_t trainId) const override {
        // Načte si polohu vlaku
        // Porovná tuto polohu se všemi pravidly simulace
        // Popřípadě vytvoří nouzové brzdění
        // Adekvátně informuje EVC o těchto okolnostech
    }
};
```

Ukázka 3.46 Zjednodušený kód třídy *RuleCheckingService*

V rámci metody `CheckRules()` (pokud je nutné vykonat nouzové zastavení) se zavolají metody ze třídy *EmergencyStopService*, která je popsaná o odstavec níže. Ty zajistí jak vytvoření tak i odvolání nových nouzových brzdění.

Příkaz k nouzovému brzdění nemusí pocházet jen z pravidel simulace. Z lektorského pracoviště může kdykoliv přijít zpráva o okamžitém nouzovém brzdění. Pro tento příkaz je v CEMu ve složce *cem/Messages/Lpc/Rbc* zpráva *LpcEmergencyStopMessage*, ve které je identifikátor vlaku, kterého se brzdění týká a čas po který se mají brzdy držet. V tomto případě v *LpcEmergencyStopMessageHandler*u zavolá jen metoda *CreateUnconditionalEmergencyStop()* s parametry ze zprávy a zbytek je automaticky vyřešen.

3.4.3 Spravovaní nouzových zastavení

Nejdříve jsem vytvořil třídu pro ukládání informace o tom, že nouzové brzdění bylo vytvořeno. Pojmenoval jsem jí *EmergencyStop* a její kód lze vidět na ukázce 3.47

```
class EmergencyStop {
public:
    EmergencyStop(uint32_t _id, uint32_t _time);
    bool Tick() {
        if(remainingTime-- > 0) {
            return true;
        }
        return false;
    }
private:
    uint32_t id;
    uint32_t remainingTime;
};
```

Ukázka 3.47 Zjednodušený kód třídy *EmergencyStop*

Pro správu těchto nouzových zastavení jsem vytvořil novou službu *EmergencyStopService*. Tato služba má za úkol pravidelně kontrolovat a popřípadě rušit nouzová zastavení po uplynutí stanovené doby. Její zjednodušený kód lze vidět na ukázce 3.48.

Má dvě veřejné metody pro generování nových nouzových brzdění. Tyto metody automaticky pošlou zprávu do EVC o brzdění a zároveň ho přidají do mapy, aby se později mohlo odvolat. V rámci monitorování těchto brzdění se každou sekundu volá `Tick()` na každém jednom *EmergencyStop*. A pokud tato metoda vrátí *false* tak je na čase toto brzdění zrušit. To se dělá pomocí zprávy 18: „Emergency stop revocation“, kde se „NID_EM“ vyplní identifikátorem brzdění. Tento postup je více popsán v návrhu (viz. 2.3.4).

```
class EmergencyStopService : public IEmergencyStopService,
                             public IInitializable, public IStartable {
public:
    void Initialize(ServiceContainer& container) override {
        // Natahá si potřebné služby
    }
    void Start(ServiceContainer& container) override {
        // Spustí monitorování nouzových brzdění
    }
    void CreateUnconditionalEmergencyStop(uint32_t trainId,
                                          uint32_t time) override;
    void CreateConditionalEmergencyStop(uint32_t trainId,
                                        Distance engageDistance) override;
    ...
private:
    std::map<uint32_t, std::list<EmergencyStop>> emergencyStops;
    ...
};
```

Ukázka 3.48 Zjednodušený kód třídy *EmergencyStopService*

3.4.4 Heartbeat

Další důležitá třída je *HeartbeatService*. Tato služba periodicky posílá zprávu, že komponenta žije. Toto se děje ve smyčce, která je regulovaná zprávami od LPC. Její zjednodušená implementace je na ukázce 3.49, kde jsem opět využil „...“ pro vynechání nedůležitého kódu na úkor ukázky.

Metoda `Start()` vytvoří nové vlákno pro tuto službu, které ještě neposílá žádné zprávy, jen čeká na příkaz od LPC. Jakmile přijde pokyn „start“ od LPC, tak začne každou jednu sekundu odesílat „heartbeat“. Pokud přijde zase příkaz „stop“, tak se vlákno nezastaví, jen přestane posílat zprávy. Když je potřeba pracovní vlákno úplně zastavit, slouží k tomu metoda `AppExit()`, která je poděděná díky rozhraní `IStartable`.

```
class HeartbeatService : public IHeartbeatService, public IInitializable,
                         public ILpcManageable, public IStartable {
public:
    void Initialize(ServiceContainer& container) override {
        // Natahá si potřebné služby
    }
    void Start(ServiceContainer& container) override {
        // Vytvoří pracovní vlákno
    }
    bool LpcSaidStart() override {
        // Nastaví spuštěnému vláknu aby začlo posílat zprávu
    }
    bool LpcSaidStop() override {
        // Nastaví spuštěnému vláknu aby přestalo posílat zprávu
    }
    void AppExit() override {
        // Zastaví kompletně pracovní vlákno
    }
    ...
private:
    ...
};
```

Ukázka 3.49 Zjednodušený kód *HeartbeatService*

Kapitola 4
Testování

Testování je velmi důležitou součástí jakéhokoli softwarového díla, ale v C++ má spoustu problémů. Jeho hlavním je chybějící reflexe a s ní spojená úroveň kontroly nad volanými metodami či generovanými objekty. GoogleTest část těchto problému eliminuje, ale není to dostatečné a proto je občas nutné část věcí obejít.

Pro testování základních funkcí použiji sady unit testů. Kde bude potřeba testovat komplikovanější chování, využiji mockování.

4.1 GoogleTest

GoogleTest je testovací framework pro psaní automatických testů. Poskytuje širokou škálu asercí[1] pro ověření očekávaného chování kódu. Jsou zde dva základní typy asercí. Verze `ASSERT_*` při selhání generují fatální selhání a přeruší aktuální funkci. Naopak `EXPECT_*` generují nefatální selhání, která nepřerušují aktuální funkci. Obvykle se dává přednost `EXPECT_*`, protože umožňují nahlásit více než jedno selhání v testu. `ASSERT_*` by se však mělo použít, pokud nemá smysl pokračovat, když dané tvrzení selže. [13]

Samotné testy pak používají více těchto maker k otestování kódu a podle jejich výsledků se odvíjí úspěch celého testu. Jestli všechny projdou, tak test se vyhodnotí jako úspěšný. Pokud alespoň jedna aserce selže, tak celý test je vyhodnocen jako neúspěšný.

Nyní bych rád shrnul nějaké makra, které poskytuje GoogleTest pro psaní testů.

`EXPECT_TRUE(podmínka)` Ověří, zda je `podmínka` pravdivá.

`EXPECT_FALSE(podmínka)` Ověří, zda je `podmínka` nepravdivá.

`EXPECT_THROW(příkaz, typ_výjimky)` Ověří, zda `příkaz` vyhodí výjimku typu `typ_výjimky`.

`EXPECT_NO_THROW(příkaz)` Ověří, zda `příkaz` nevyhodí žádnou výjimku.

`EXPECT_EQ(hodnota1, hodnota2)` Ověří, zda se `hodnota1` a `hodnota2` rovnají.

`EXPECT_NE(hodnota1, hodnota2)` Ověří, zda se `hodnota1` a `hodnota2` liší.

`EXPECT_GT(hodnota1, hodnota2)` Ověří, zda je `hodnota1` vetší než `hodnota2`.

`EXPECT_LT(hodnota1, hodnota2)` Ověří, zda je `hodnota1` menší než `hodnota2`.

[1] Aserce je tvrzení, které má nějakou pravdivostní hodnotu.

4.1.1 Fixtures

„Fixtures“ je koncept v google testu, který umožňuje sjednotit více testů „pod jednu střechu“. Zároveň umožní napsat kód, který je zavolán před/po každém testu. Má formu třídy, která dědí z `testing::Test`, což je třída od google test frameworku. Na ukázce 4.1 je vidět použití při testování třídy *TrainRegistryService*. Všechny proměnné ze třídy jsou dostupné pak v jednotlivých scénářích. Je zde metoda `SetUp()`, která se volá před scénářem a metoda `TearDown()`, která se volá po. Při provádění samotných scénářů se tedy nejdříve vytvoří tato „fixture“ třída. Zavolá se metoda `SetUp()`. Provede se kód testu a nakonec se zavolá metoda `TearDown()`. Pro každý test je tedy celá „fixture“ třída vlastní a testy se navzájem nemohou ovlivnit. [13]

```
#include "gmock/gmock.h"

class TrainRegistryServiceTest : public testing::Test {
protected:
    void SetUp() override {
        trainRegistryService = new TrainRegistryService();
        trainRegistryService->RegisterTrain(0);
    }
    void TearDown() override {
        delete trainRegistryService;
    }
    TrainRegistryService* trainRegistryService;
};
```

Ukázka 4.1 Kód mocku třídy *ConfigurationService*

4.1.2 Google Mock

Pro simulování chování tříd, na kterých je závislá třída, která se testuje, využiji rozšíření gMock. To přídává možnost tzv. mockování tříd. Umožňuje vytvořit „falešnou“ třídu, která navenek vypadá totožně, ale má předem definované chování. Díky tomu eliminuje možnost chyby v jedné ze závislostí a je jistota, že test testuje pouze funkce třídy, která je předmětem testu. [14]

Mockování tříd je dosaženo opět pomocí maker. Nejdříve je nutné třídu definovat pomocí makra `MOCK_METHOD()`. Toto makro bere strukturu metody jako parametr. Na ukázce kódu 4.2 lze vidět jeho použití k mocknutí třídy *ConfigurationService*.

```
#include "gmock/gmock.h"

class MockConfigurationService: public IConfigurationService {
public:
    //----------Návratová hodnota----------------Jméno metody-------
    MOCK_METHOD(std::shared_ptr<IConfiguration>, FetchConfiguration,
                (ConfigType), (override));
    //-----------Parametry-----Přetížení--
};
```

Ukázka 4.2 Kód mocku třídy *ConfigurationService*

Po lze instanci mocknuté třídy nastavit její chování. To se dělá makrem `ON_CALL()`. Toto makro bere jako argumenty třídu jejíž chování chceme nastavit a její metodu. Následně pomocí `.WillByDefault(Return(proměnná)` se nastaví návratová hodnota po zavolání této metody. Toto lze ještě posilnit tím, že lze nastavit konkrétní parametr očekávaný ve volání. Na ukázce kódu 4.3 lze všechno toto vidět.

```
mockConfigService = new MockConfigurationService();
//-------Třída--------------Metoda-------------Její parametr---------
ON_CALL(*mockConfigService, FetchConfiguration(ConfigType::Listener))
    .WillByDefault(Return(std::make_shared<ListenerConfiguration>()));
//---Po zavolání---Vrátí--------------------Nějakou hodnotu-----------
```

Ukázka 4.3 Kód nastavení chování mocku

Následně lze toto chování odchytit a testovat, zda byla metoda zavolána. K tomu slouží makro `EXPECT_CALL()`, které má velmi podobnou strukturu. Nejdříve je třeba určit třídu a metodu, které opět lze specifikovat parametr (`_` značí libovolný jeden parametr). Na to navazuje počet volání pomocí `.Times(počet)`. Do argumentu `počet` lze dát číslo nebo nějakou další funkci z google testu. V ukázce 4.4 je použita funkce `AtLeast(1)`, která znamená alespoň jedno zavolání.

```
//-----------Třída--------------Metoda-----------------Počet volání----
EXPECT_CALL(*mockConfigService, FetchConfiguration(_)).Times(AtLeast(1);
```

Ukázka 4.4 Kód odchycení volání mocku

4.2 Testování zpráv

Testování zpráv je již zpracováno týmem SP1/2. Proto jsem pro mnou nově vytvořené zprávy jen rozšířil jejich sadu testů. Zprávy jsou testovány třídou *MessagesTest* ve které se každá zpráva porovná s referenční zprávou. Testují se metody `to_json`, `from_json`, a operátor „==“. To samé platí i pro pakety a třídy ve složce „other“.

4.3 Testování pravidel scénářů

Co je kompletně nové pro testování, jsou pravidla scénářů. Pro jejich testy, jsem přidal do CEMu složku „Rules“ uvnitř adresáře s testy (viz 4.5). V ní má každé pravidlo svůj soubor se všemi testovacími scénáři. Pro testování metod `to_json` a `from_json` jsem využil již existující konstrukci, funkci `testAgainstJson()`, která byla napsána týmem SP1/2.

```
cem
└── test
    └── Rules
        ├── ConditionalEmergencyStopRuleTest.cpp
        ├── MovementAuthorityRuleTest.cpp
        ├── TAFShowRuleTest.cpp
        └── UnconditionalEmergencyStopRuleTest.cpp
```

Ukázka 4.5 Testy pravidel scénáře uvnitř CEMu

4.4 Testování služeb

Testování služeb je již více komplikované. Kvůli jejich provázanosti to není jen několik jednotkových testů, ale často jsem musel služby, na kterých jsou ostatní závislé, namockovat.

4.4.1 Mock kontejneru služeb

Pro testování služeb bylo důležité namockovat i kontejner služeb. Ten potřeboval zviditelnit (jsou `protected` a nebyly by možné volat z testů) metody `RegisterService()` a `Initialize()`. Dále jsem přidal metodu `SetupMockService`, která umožní dostat mockované služby dovnitř, aby byly dostupné pro testovanou službu a vypadaly stejně jako reálná implementace. Také bylo potřeba namockovat ostatní veřejné metody kontejneru služeb, `LpcSaidStart/Stop/Restart()`.

```
class MockServiceContainer: public ServiceContainer {
public:
    template <typename T>
    void SetupMockService(T* ptr) {
        services[T::Type] = std::shared_ptr<T>(ptr);
    }
    template <typename T>
    void RegisterService() {
        ServiceContainer::RegisterService<T>();
    }
    void InitializeServices() {
        ServiceContainer::InitializeServices();
    }

    MOCK_METHOD(bool, LpcSaidStart, (), (override));
    MOCK_METHOD(bool, LpcSaidStop, (), (override));
    MOCK_METHOD(bool, LpcSaidRestart, (), (override));
};
```

Ukázka 4.6 Kód mocku kontejneru služeb

Všechny mocknuté třídy (včetně kontejneru služeb) se nachází ve složce „MockClasses“ v testovacím adresáři (viz popis adresáře 4.7).

```
rbc
└── test
    ├── MessageHandlers ..... Testy message handlerů
    ├── MockClasses ..... Mock implementace tříd
    ├── Services ..... Testy služeb
    └── Utils ..... Testy pomocných tříd
```

Ukázka 4.7 Popis adresáře testů v komponentě RBC

4.4.2 Ukázka na *NationalValuesRegistry*

Nejjednodušší příklad pro ukázku testování služeb je *NationalValuesRegistry*. Ta je závislá jen na jedné službě, *ConfigurationService* a ta je již namockována, viz ukázka 4.2.

Pro začátek popíšu její „fixture“ třídu, která je vidět na ukázce 4.8. Nejdříve se vytvoří mock služby *ConfigurationService*, nastaví se jí očekávané chování pomocí makra `ON_CALL`. Poté je vytvořen mock kontejneru služeb a do ní přihlášen mock konfigurační služby. Nakonec se zaregistruje *NationalValuesRegistryService*, která se bude testovat.

```
class NationalValuesRegistryServiceTest : public testing::Test {
protected:
    void SetUp() override {
        cfgService = new MockConfigService();
        ON_CALL(*cfgService, FetchConfiguration(ConfigType::NationalValues))
                .WillByDefault(Return(NationalValuesConfiguration>());

        serviceContainer = new MockServiceContainer();
        serviceContainer->SetupMockService<MockConfigService>(cfgService);
        serviceContainer->RegisterService<NationalValuesRegistryService>();
    }
    void TearDown() override {
        delete serviceContainer;
    }
    MockConfigService* cfgService;
    MockServiceContainer* serviceContainer;
};
```

Ukázka 4.8 Ukázka kódu „fixture“ třídy *NationalValuesRegistryServiceTest*

Po definici „fixture“ třídy následují samotné testovaní scénáře. Ty jsou deklarovány pomocí makra `TEST_F()`. Jak je vidět na ukázce 4.9, první argument makra je „fixture“, do které test patří a druhý je jméno testu. Následuje samotné tělo testu, kde se nejdříve nastaví očekávané chování a následně se zavolá metoda `GetDNvStff()` a otestuje její výsledek.

```
TEST_F(NationalValuesRegistryServiceTest, GetDNvStff) {
    EXPECT_CALL(*cfgService, FetchConfiguration).Times(AtLeast(1));
    serviceContainer->InitializeServices();

    INationalValuesRegistryService* service = serviceContainer->
                        FetchService<INationalValuesRegistryService>().get();

    EXPECT_EQ(service->GetDNvStff().GetMeters(), 300);
    EXPECT_EQ(service->GetDNvStff().GetCm(), 30000);
}
```

Ukázka 4.9 Ukázka jednoho testovacího scénáře

4.5 Testování message handlerů

Proces testování message handlerů je velmi podobný procesu testování služeb. Opět se nejdříve vytvoří „fixture“ třída, která vydefinuje všechny závislosti daného message handleru. Následně se vytvoří nový handler. V samotných testech se pak vytvoří falešná zpráva, která se handleru pošle ke zpracování a odchytává se jeho chování.

4.6 Výsledky testování

Během testování jsem odhalil několik chyb v přepočítávání jednotek. Časté byly chyby ve vzdálenostech, které vyřešila třída *Distance*. Dále jsem odhalil spoustu chybějících kontrol stavu či ostatních podmínek u message handlerů, které by měly kontrolovat stav připojení vlaku nebo jeho pozici. Toto jsem vyřešil přidáním kontrol před začátkem zpracovávání zprávy.

Zde bych rád vyzdvihl několik zajímavějších chyb, které mě při testování potrápily.

- Chybějící virtuální destruktor u konfigurací.
  Tato chyba vedla k únikům paměti.
- Obrácené pořadí argumentů u metod *EmergencyStopService*.
  Prohozené ID a čas trvání vedlo k nesmyslnému chování.
- Zbytečné „gettery“ u databázové služby.
  Zbytečný kód navíc, který se nikdy nepoužíval.
- Přebývající konstruktor u *IInitializable*
  Nedává smysl mít konstruktor u rozhraní.
- Nekonečná spousta překliků u pojmenování metod.
  Asi nejčastější programátorská chyba. Neovlivní to chování programu, ale může to vzbudit podivné pohledy od budoucích kolegů.

Všechny tyto chyby (a mnoho dalších) jsem stihl vyladit a předávám komponentu RBC otestovanou a se všemi odhalenými chybami vyladěnými.

4.6.1 Spouštění testů

V příloze je kromě spustitelné binárky jménem „RBC_run“ také binárka „RBC_test“, která spustí komponentu v testovacím režimu, kde se automaticky spustí všechny sady vytvořených testů. Do konzole se poté vypíše výsledek testování a případné chyby (pokud nějaký test neuspěje).

Kapitola 5

Integrace s ETCS projektem

Vzhledem k tomu, že role komponenty RBC je jen částí celého projektu simulátoru ETCS, rád bych v této sekci shrnul procesy nutné fungování v rámci projektu.

5.1 Gitlab

Klíčem veškerých možností fungování většího projektu jsou komunikace, koordinace a hlavně kooperace. Nástroj díky, kterému je všechno toto možné je GitLab. Projekt ETCS a všechny jeho části jsou uložené na fakultní distribuci GitLabu, kam mají všichni členové projektu přístup.

Komponenta RBC má zde svůj vlastní repositář, ve kterém se nachází veškerý její kód. Tento repositář je dále vložen jako „submodule“ ve větším repositáři „ETCS_TOGETHER“ pro integraci s projektem. V tomto repositáři se koordinují aktivity týmů a dalších tvůrců bakalářských i všech ostatních prací.

Součástí repozitáře RBC (ale i spousty dalších) je již zmíněný CEM. Díky němu je zajištěno, že zprávy, které se posílají, vypadají stejně na straně odesílatele i příjemce. To zamezuje vzniku problému při komunikaci, kdy např. RBC pošle nějakou zprávu a EVC ji očekává v jiném formátu. Tento problém byl v minulosti častý a velmi nepříjemný. Vznik CEMu tento problém eliminoval a vše je nyní sjednocené a centralizované.

Pipeline

„GitLab Pipeline“ je nástroj, který umožňuje automatizovat procesy spojené s vývojem softwaru, jako je sestavení, testování a nasazení aplikací.

V repositáři RBC se také nachází automatizovaná „pipeline“, ve které se celý kód kompiluje a automaticky otestuje. Toto dílo ovšem ale vzniklo paralelně k mé bakalářské práci během běhu předmětů SP1/2 a nepovažuji ho za mé, pouze ho vděčně využívám. Tímto tedy děkuji zbytku mého týmu, že tuto krásnou „pipeline“ napsali a že ji teď mohu využívat.

5.2 Docker

Docker je nástroj, který umožňuje snadnější vytváření, spouštění a sdílení aplikací tím, že je zabalí do izolovaných balíčků nazývaných kontejnery. To zajišťuje, že aplikace bude mít konzistentní prostředí bez ohledu na to, kde bude spuštěna a její nasazení bude jednodušší. [15]

Podobný scénář jako s „pipeline“ platí pro integraci dockeru. RBC ke svému fungování potřebuje databázi, která ne vždy běží někde externě. Pro vývojářské potřeby vznikl dockerový kontejner ve kterém běží databáze naplněná daty. Tento kontejner lze vytvořit a spustit díky souboru „docker-compose.yml“, díky němu se spoustí jak RBC spolu s MQTT brokerem tak i již zmíněná databáze. K ní se po spuštění může RBC připojit a pracovat s ní.

Všechny tyto procesy slouží spíše pro vývoj a debuggování a byly vyvinuty během běhu předmětů SP1/SP2. Zmiňuji se o nich protože existují a jsou součástí repositáře. Pro produkční prostředí se komponenta spouští samotná a všechno je řešeno ze strany týmu ETCS.

5.3 JRU logger

V rámci projektu vznikl požadavek na centralizované logovaní. Tento problém vyřešila komponenta JRU, která byla vyvinuta v rámci bakalářské práce Martina Čáslavského. Pro přesnější informací o této funkcionalitě doporučuji přečíst právě jeho práci. Vše tam je popsáno více do detailu, zde to vysvětlím jen velmi stroze. [16]

Pro účely RBC byla od Martina Čáslavského dodána nová služba *JRULoggerService*, která má několik veřejných metod pro logování. Lze určit závažnost logu pomocí výčtového typu *MessageType*. Dále lze určit zda vypisovat do konzole nebo jen poslat do komponenty JRU. Na výpisu kódu 5.1 je několik ukázek použití této služby.

```
jruLoggerService->Log(true, MessageType::Error, "Train data not valid.");

jruLoggerService->Log(true, MessageType::Fatal,
                      "MqttListener could not connect to MQTT broker.");

jruLoggerService->Log(MessageType::Info,
                      "Received normal position report");

jruLoggerService->Log(true, MessageType::Warning,
                      "Tried to remove non registered engine. id:%id%",
                      msg.GetNidEngine());
```

Ukázka 5.1 Několik ukázek použití *JRULoggerService*

V argumentu metody `Log()` je první nepovinný argument typu *bool*. Pokud je zde „true“, znamená to, že log má být i vypsán v konzoli (viz 1. a 2. log na ukázce 5.1). Pokud zde není nic, tak se pouze pošle do JRU (viz 3. log na 5.1). Druhý argument je typ logu. Ten určuje závažnost, podle které se i určuje barva výpisu do konzole. Následuje samotná zpráva logu do které se pomocí notace „%názevProměnné%“ dá vložit proměnná. Ta musí následovat hned za zprávou logu. Tento příklad je na ukázce 5.1 úplně poslední.

Kapitola 6
Závěr

Teoretická část této práce popisuje fungování systému ETCS, jeho součástí a vysvětlení jeho přínosů pro vlakovou dopravu. Dále objasňuje procedury, kterých se přímo účastní komponenta RBC. Byl popsán stávající stav implementace komponenty RBC v simulátoru ETCS a také její nedostatky.

Praktická část byla rozdělena na dvě části. Část návrhová, kde pomocí metod softwarového inženýrství vznikl návrh nové architektury a funkčností požadované simulátorem ETCS a část implementační, kde se věnuji konkrétní implementaci navržených funkcionalit. Popsány jsou nejen základní stavební kameny nové architektury, ale i jednotlivá řešení požadavků práce.

Cílem této práce bylo navržení nové architektury, reimplementace stávajícího RBC a rozšíření o nové požadavky simulátoru. Vznikla kompletně nová architektura, která splňuje požadavky týmového projektu. Všechny stávající funkcionality byly úspěšně reimplementovány. Vznikly nové scénáře jízdy a podpora procedur s nimi spjatými. Přepsané RBC se povedlo nasadit a již spolupracuje se zbytkem projektu simulátoru ETCS.

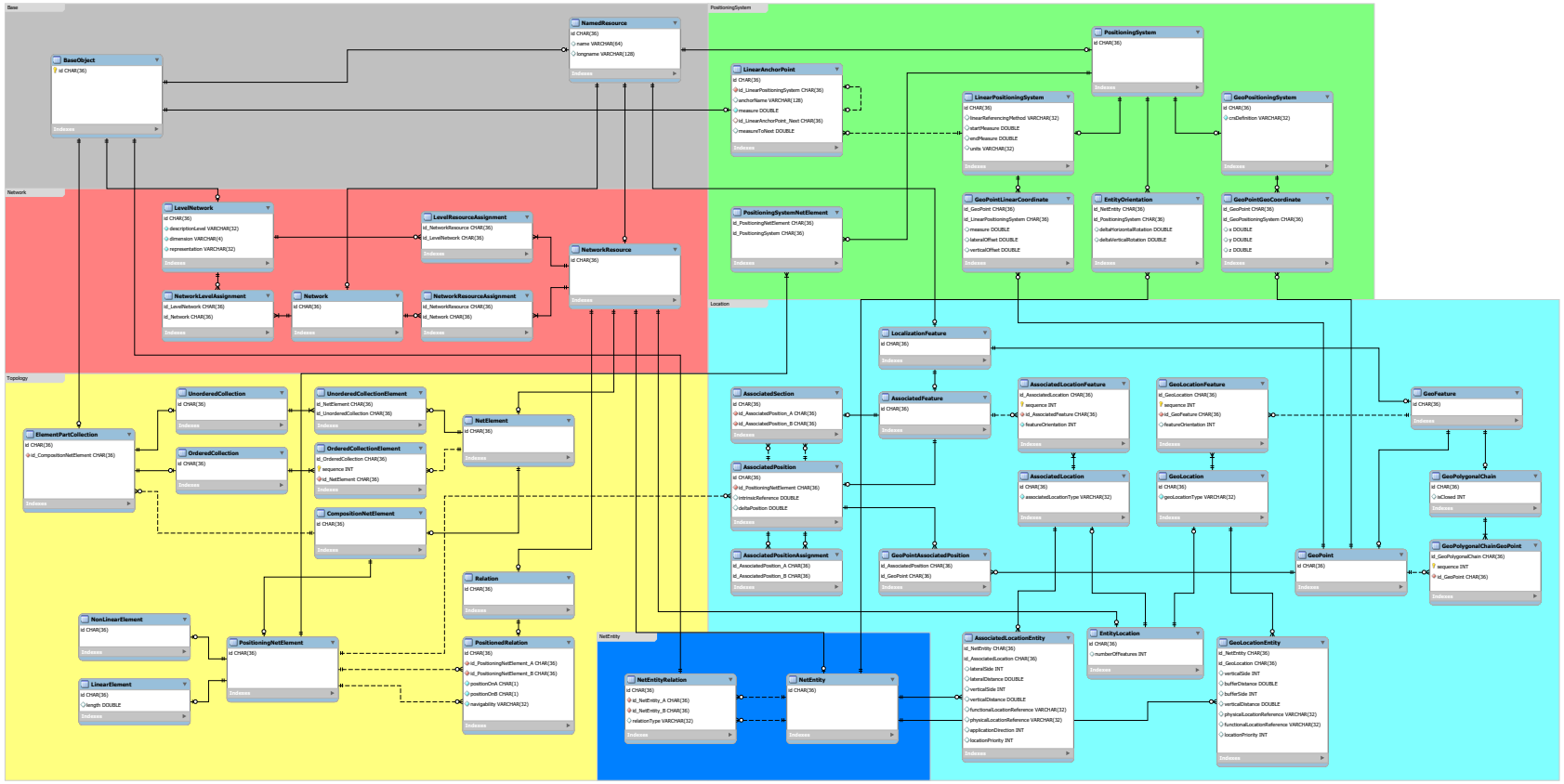
Dále do cílů spadá také zdokumentování a otestování nově vzniklého kódu. Kód je vybaven rozsáhlými testy. Tyto sady testů pokrývají nejen standardní situace, ale i mezní případy. Díky nim, kvalitní struktuře kódu, dokumentaci a množství doplňujících komentářů v kódu, lze považovat kód nově přepsané komponenty RBC za srozumitelný a hlavně rozšiřitelný pro členy budoucích týmů. Kód komponenty je zdokumentován nástrojem *Doxygen*. Dále jako dokumentace slouží i implementační část této práce, kde je většina komplikovanějších částí kódu detailně popsána. Také jsem do repositáře „ETCS_TOGETHER“ přidal na wiki v angličtině text (přeložená kopie v příloze B), který funguje jako vývojářská příručka. Cíle práce se tedy povedlo úspěšně splnit.

V budoucnu je možné rozšířit komponentu pro monitorování více vlaků najednou. V rámci praktické části na toto bylo myšleno, aby tato funkcionalita nebyla příliš komplikovaná na přidání. Také se v rámci projektu mluvilo o cestování v čase pomocí změny stavových proměnných. Tato funkce je již komplikovanější na provedení, ale opět jsem se na to snažil myslet při návrhu a implementaci a díky tomu by to nemělo být nadmíru obtížné.

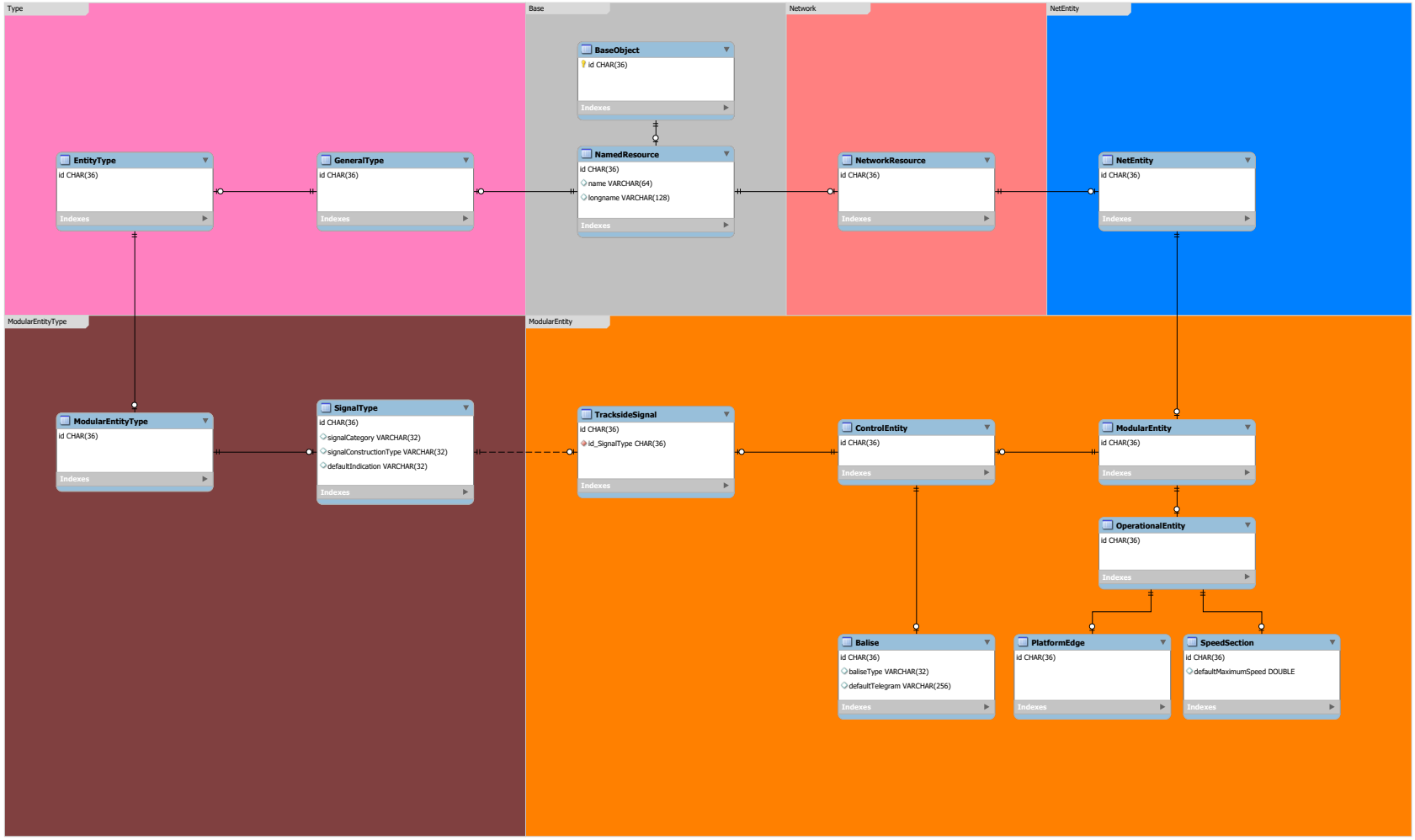
Příloha A

Popis traťové databáze

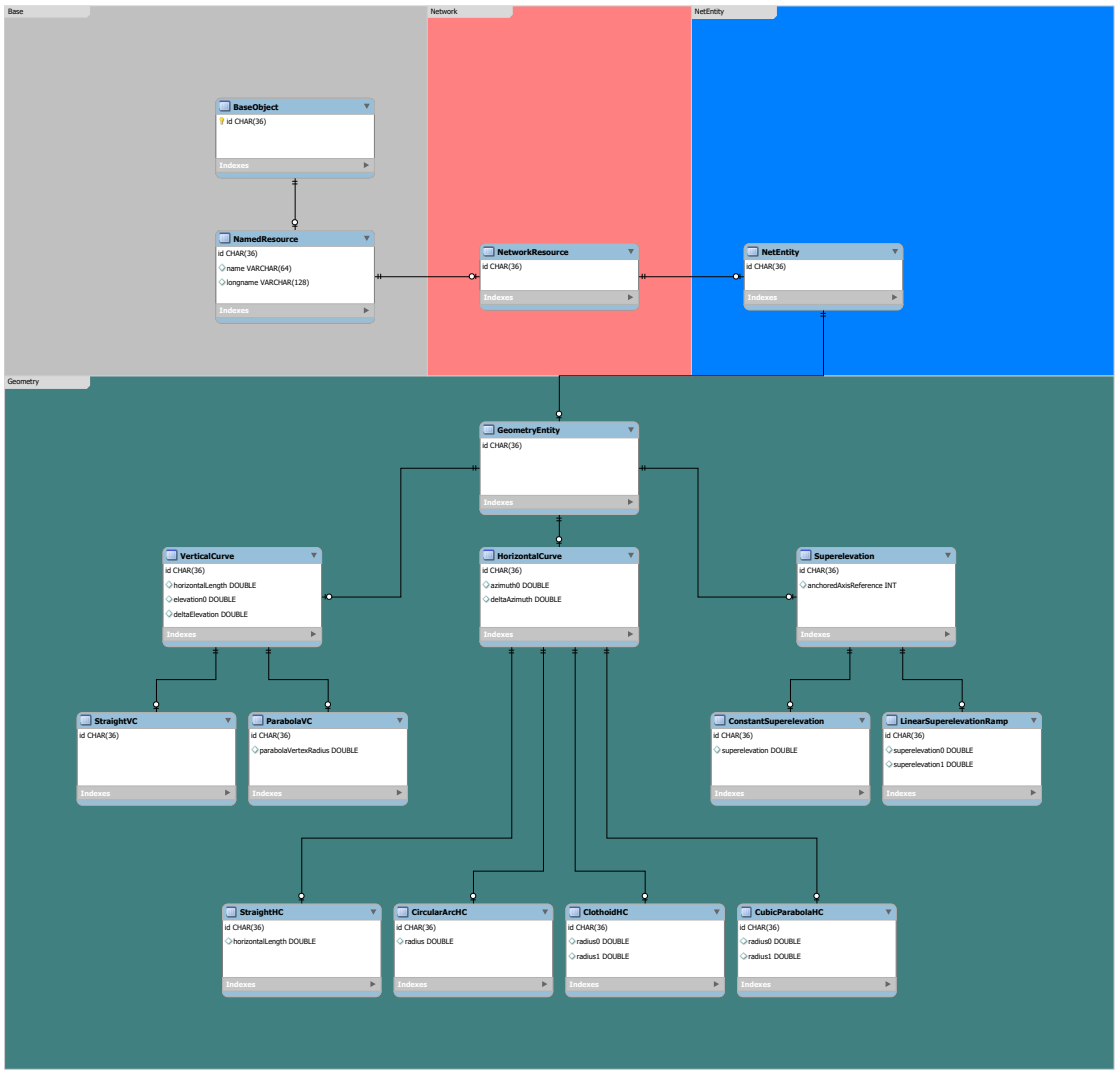
Na dalších stránkách přikládám popis databáze, který jsem převzal od studentů Fakulty dopravní ČVUT.



Obrázek A.1 Kompletní základ databáze



Obrázek A.2 Rozšíření pro *modular entity*



Obrázek A.3 Rozšíření pro geometrii okolo *vertical curve*

Příloha B

Vývojářská příručka

Architektura

Tuto architekturu lze rozdělit do několika částí.

Služby

Hlavní funkční část kódu. Každá služba musí mít deklarovaný typ služby a každá služba musí mít jiný typ. Některé z důležitějších služeb:

- **MqttListenerService** – slouží k přijímání zpráv zasílaných přes MQTT a jejich distribuci podle `topic` příslušným *topic workers*.
- **MqttPublisherService** – slouží k publikování nových zpráv prostřednictvím MQTT.
- **JRULoggerService** – slouží k zasílání informačních zpráv o stavu libovolné komponenty do JRU.

Kontejner služeb

Třída, díky které je možný tzv. „inversion of control“.
Obsahuje všechny instance služeb, které lze načíst pomocí metody `.FetchService<>()`.
Předává se při každé inicializaci služby, aby mohla načítat další služby.
Všechny služby musí být zaregistrovány v `Application.Initialize()`, aby byly k dispozici pro načítání. Kontejner služeb je zodpovědný za uvolnění paměti alokovaných služeb, samotné služby si uchovávají pouze „raw pointer“ na ostatní služby.

Message handlers

Část kódu, která je volána z pracovníků témat při přijetí nějaké zprávy. Obvykle se skládá z několika částí.

1. Parsování příchozí zprávy, která má být později zpracována.
2. Kontrola obsahu zprávy, aby odpovídala očekávanému obsahu.
3. Zpracování zprávy
4. (*nepovinné*) Parsování paketů
5. (*nepovinné*) Zpracování paketů
6. Generování odpovědi
7. Odeslání odpovědi

Topic Worker

Pracovní vlákno, které přijímá zprávu od služby *MqttListenerService* a je zodpovědné za provedení práce. Každé z nich pracuje na jednom *topicu* a provolává odpovědné message handlery.

Konfigurace

Je zde služba *ConfigurationService*, která umí načítat konfigurace ze souborů JSON, popřípadě generovat nové.

Děláme to tak, že máme výchozí konfiguraci deklarovanou v souboru *SomeConfiguration.hpp*. Kdykoli se pokusíme konfiguraci načíst, nejprve se podíváme, zda již nějaká nexistuje a není uložená v JSONu. Pokud ne, vytvoříme novu s hodnotami ze třídy *SomeConfiguration.hpp*, vezmeme je jako výchozí a použijeme.

Všechny konfigurační JSONy jsou uloženy odděleně ve složce *configurations* vedle binárky. A tato složka není v gitu verzována, neverzujeme ji, abychom zabránili tomu, že někde budou uloženy jiné než výchozí/testovací hodnoty. JSONy jsou automaticky generovány, jak je popsáno v odstavci výše.

Ukázka použití

Výše zmíněná *ListenerConfiguration* má pole vyplněná v konstruktoru.

```
class ListenerConfiguration : public IConfiguration {
public:
    // Ukázka z RBC
    ListenerConfiguration() {
        topics.push_back(Topic::EVCtoRBC);
        topics.push_back(Topic::TEST);
    }
    std::vector<Topic> topics;
    .
    . //tady by byly metody to_json a from_json, které jsem odebral kvuli ukázce
    .
}
```

Vygenerovaný JSON vypadá následovně:

```
{
"topics": [
    "evc/rbc",
    "test"
    ]
}
```

Příklad ze služby *MqttListenerService* v metodě `LpcSaidStart()`.

```
// Zde nejprve vytvoříme výchozí konfiguraci, abychom mohli volat metodu from_json
// Hodnoty uvnitř budou přepsány načtenou konfigurací
std::shared_ptr<ListenerConfiguration> config = std::make_shared<ListenerConfiguration>();
config->from_json(configurationService->FetchConfiguration(ConfigType::Listener)->to_json());

for (Topic t : config->topics) {
    topicWorkers.at(t)->Stop();
    mosquitto_unsubscribe(komar, NULL, ConvertTopicToString(t).c_str());
    topicWorkers.erase(t);
}
// A používáme ji normálně
```

Údržba

Nové služby

Když vytváříte novou službu, inspirujte se u již vytvořených služeb. Každá služba má zde deklarován svůj typ v výčtovém typu *ServiceType.hpp*. Má také své rozhraní, které má deklarovanou proměnou typu `constexpr` a dědí z *IService.hpp*. Samotná implementace služby může dědit z vybraných rozhraní.

- *IInitializable.hpp*
- *IStartable.hpp*
- *ILpcManagable.hpp*

Mělo by to vypadat následovně:

ISomethingService.hpp

- Dědí z *IService*.
- Má `constexpr` *ServiceType*.
- Deklaruje všechny `public` metody.
- Má všechny metody „pure virtual“.

SomethingService.hpp

- Dědí z *ISomethingService*.
- Typicky dědí z *IInitializable*.
- Volitelně může také dědit z *IStartable*.
- Deklaruje všechny `private` členské proměnné.
- „Overriduje“ všechny `public` metody popsané ve svém rozhraní.
- Volitelně může deklarovat některé pomocné `private` metody.

SomethingService.cpp

- Implementuje všechno v *SomethingService.hpp*.

Získávání služeb

Mnoho částí kódu potřebuje ke svému fungování služby. Tyto služby je třeba načíst z kontejneru služeb. Služby se načítají ve fázi inicializace služby a jsou uloženy jako soukromé proměnné ve formě „raw pointeru“.

Možná vás zajímá, proč používáme „raw pointery“. Vysvětlení najdete v části *Proč používáme raw pointery*.

Vždy ukládejte a načítejte **rozhraní** služby, kterou chcete použít. (*výjimkou jsou „templatované“ služby, jak je uvedeno níže*)

```
// Kód nějaké služby v .hpp
...
private:
    IMAGeneratorService* maGeneratorService;
...
// Kód nějaké služby v .cpp
...
SomeService::Initialize(ServiceContainer &container) {
    maGeneratorService = container.FetchService<IMAGeneratorService>().get();
}
...
```

V **MessageHandlers** je proces načítání velmi podobný, jen s tím rozdílem, že namísto načítání v `.Initialize()` se načítá v konstruktoru.

Pravidla scénářů jízdy

Když je potřeba vytvořit nové pravidlo pro scénáře je potřeba nejdříve přidat typ pravidla do výčtového typu *RuleType*. Následně vytvořit nové pravidlo, které dědí z *Rule* a přetížit jeho virtuální metody.

Zde nailustruji na vymyšleném pravidle, které signalizuje, kde má strojvedoucí zamávat dětem na mostě.

Nejdříve přidat typ do *RuleType*:

```
enum class RuleType {
    MovementAuthority = 0,
    TAFShow = 1,
    UnconditionalEmergencyStop = 2,
    ConditionalEmergencyStop = 3,
    Wave = 4
};
```

Poté definovat hlavičkový .hpp soubor:

```
// SomeRule.hpp
class WaveRule : public Rule {
public:
    WaveRule() = default;
    WaveRule(uint32_t id, Distance wavePos);
    ~WaveRule() override = default;

    Distance GetWavePosition() const;

    nlohmann::json to_json() const override;
    void from_json(const nlohmann::json& j) override;
    bool operator==(const WaveRule& rhs) const;

private:
    Distance wavePosition;
};
```

A nakonec implementace:

```
// SomeRule.cpp
#include "WaveRule.hpp"

WaveRule::WaveRule(uint32_t id, Distance wavePos) {
    ruleType = RuleType::Wave;
    ruleID = id;
    wavePosition = wavePos;
}

Distance WaveRule::GetWavePosition() const {
    return wavePos;
}

nlohmann::json WaveRule::to_json() const {
    nlohmann::json j = Rule::to_json();
    j["wavePosition"] = wavePos.GetMeters();
    return j;
}

void WaveRule::from_json(const nlohmann::json& j) {
    Rule::from_json(j);
    wavePosition.SetDistanceFromMeters(j.at("wavePosition").get<uint32_t>());
}

bool WaveRule::operator==(const WaveRule& rhs) const {
    return Rule::operator==(rhs) &&
           wavePosition == rhs.wavePosition;
}
```

Proč používáme raw pointery

Instance služby je uložena v kontejneru služeb, který je také zodpovědný za uvolnění paměti na konci. Tu uchováváme ve „smart pointeru“, abychom nemuseli explicitně volat destruktory.

Ve službách a „message handlerech“ získáváme pouze „raw pointer“ pro použití. Pokud bychom použili vždy `shared_ptr`, narazili bychom na problém, když by se navzájem potřebovalo více služeb, tak by si navzájem udržovaly `shared_ptr` a na konci aplikace by počty odkazů uvnitř nikdy nedosáhly nuly a aplikace by celá „leakovala“ paměť.

Proto si uchováváme pouze „raw pointer“ a **NEVOLÁME NA NĚJ DELETE**. Za to odpovídá kontejner služeb, my si pouze „půjčujeme“ ukazatel na instanci uloženou v kontejneru služeb.

Bibliografie

1. ERTMS.NET. *ERTMS in brief* [online]. 2024. [cit. 2024-02-27]. Dostupné z: https://www.ertms.net/about-ertms/ertms-in-brief/.
2. SPRÁVA ŽELEZNIC. *Proč ETCS* [online]. [B.r.]. [cit. 2024-03-03]. Dostupné z: https://www.spravazeleznic.cz/digitalizace/etcs/proc-etcs.
3. EUROPEAN UNION AGENCY FOR RAILWAYS. *SUBSET-026 System Requirements Specification Chapter 2 Basic System Description* [online]. 2006. Ver. 2.3.0 [cit. 2024-03-05]. Dostupné z: https://www.era.europa.eu/era-folder/set-specifications-1-etcs-b2-gsm-r-b1.
4. EUROPEAN UNION AGENCY FOR RAILWAYS. *SUBSET-026 System Requirements Specification Chapter 3 Principles* [online]. 2006. Ver. 2.3.0 [cit. 2024-03-05]. Dostupné z: https://www.era.europa.eu/era-folder/set-specifications-1-etcs-b2-gsm-r-b1.
5. EVROPSKÁ UNIE. *Subsystems and Constituents of the ERTMS - European Commission* [online]. 2023. [cit. 2024-03-03]. Dostupné z: https://transport.ec.europa.eu/transport-modes/rail/ertms/what-ertms-and-how-does-it-work/subsystems-and-constituents-ertms_en.
6. EUROPEAN UNION AGENCY FOR RAILWAYS. *SUBSET-026 System Requirements Specification Chapter 8 Message* [online]. 2006. Ver. 2.3.0 [cit. 2024-03-05]. Dostupné z: https://www.era.europa.eu/era-folder/set-specifications-1-etcs-b2-gsm-r-b1.
7. EUROPEAN UNION AGENCY FOR RAILWAYS. *SUBSET-026 System Requirements Specification Chapter 7 ERTMS/ETCS language* [online]. 2006. Ver. 2.3.0 [cit. 2024-03-05]. Dostupné z: https://www.era.europa.eu/era-folder/set-specifications-1-etcs-b2-gsm-r-b1.
8. EUROPEAN UNION AGENCY FOR RAILWAYS. *SUBSET-026 System Requirements Specification Chapter 4 Modes and Transitions* [online]. 2006. Ver. 2.3.0 [cit. 2024-03-05]. Dostupné z: https://www.era.europa.eu/era-folder/set-specifications-1-etcs-b2-gsm-r-b1.
9. EUROPEAN UNION AGENCY FOR RAILWAYS. *SUBSET-026 System Requirements Specification Chapter 5 Procedures* [online]. 2006. Ver. 2.3.0 [cit. 2024-03-05]. Dostupné z: https://www.era.europa.eu/era-folder/set-specifications-1-etcs-b2-gsm-r-b1.
10. SPRÁVA ŽELEZNIC. *SŽ Z8* [online]. 2022. [cit. 2024-03-14]. Dostupné z: https://provoz.spravazeleznic.cz/Portal/.
11. CAROL, Sliwa. *Event-driven architecture poised for wide adoption* [online]. 2003. [cit. 2024-04-23]. Dostupné z: https://www.computerworld.com/article/1726743/event-driven-architecture-poised-for-wide-adoption.html.

12. BOB, Sutor; IBM. *The pathway to a service-oriented architecture* [online]. 2003. [cit. 2024-04-23]. Dostupné z: https://www.computerworld.com/article/1325749/the-pathway-to-a-service-oriented-architecture.html.
13. GOOGLE. *GoogleTest* [online]. 2024. [cit. 2024-04-16]. Dostupné z: https://google.github.io/googletest/primer.html.
14. GOOGLE. *GoogleMock* [online]. 2024. [cit. 2024-04-22]. Dostupné z: https://google.github.io/googletest/gmock_for_dummies.html.
15. DOCKER. *Docker dokumentace* [online]. 2024. [cit. 2024-04-28]. Dostupné z: https://docs.docker.com/.
16. ČÁSLAVSKÝ, Martin. *ETCS – Studie řízení projektu a kontroly kvality výstupů.* 2024.

Obsah příloh

- `readme.txt` ..... stručný popis obsahu média
- `app` ..... adresář se spustitelnou formou implementace
  - `win` ..... pro windows
    - `RBC_run` ..... spustitelná binárka komponenty RBC
    - `RBC_test` ..... spustitelná binárka s testovacím režimem RBC
  - `ubuntu` ..... pro unbuntu
    - `RBC_run` ..... spustitelná binárka komponenty RBC
    - `RBC_test` ..... spustitelná binárka s testovacím režimem RBC
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF