České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství
Obor: Aplikace softwarového inženýrství

Rozhraní Common client core pro Python

Common Client Core interface for Python

BAKALÁŘSKÁ PRÁCE

Vypracoval: Filip Vítek
Vedoucí práce: Ing. Antonín Květoň
Rok: 2022

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Děčíně dne ....

Filip Vítek

Poděkování

Tímto bych chtěl poděkovat Ing. Josefu Novému, Ph.D. za cenné rady, pomoc při práci na bakalářské práci a poskytnutí možnosti vycestovat za účely bakalářské práce do CERNu ve Švýcarsku.
Dále bych rád poděkoval Ing. Antonínu Květoňovi za spolupráci při programování rozhraní pro knihovnu Common client core.

Filip Vítek

Název práce:
Rozhraní Common client core pro Python

Autor: Filip Vítek

Studijní program: Aplikace přírodních věd
Obor: Aplikace softwarového inženýrství
Druh práce: Bakalářská práce

Vedoucí práce: Ing. Antonín Květoň
KFNT MFF UK, V Holešovičkách 2, 180 00 Praha 2
Konzultant: Ing. Josef Nový, Ph.D.
Katedra softwarového inženýrství,
Fakulta jaderná a fyzikálně inženýrská,
České vysoké učení technické v Praze

Abstrakt: Bakalářská práce je zaměřena na vytvoření rozhraní, které umožní ovládat dialogovou knihovnu Common client core pro systém sběru dat na experimentu COMPASS v CERNu ze skriptu psaném v programovacím jazyce Python. V první části práce jsou popsány základní pojmy a poté následuje popis systému pro sběr dat experimentu COMPASS a knihovny Common client core. Hlavní částí práce je návrh řešení a implementace rozhraní. V závěru je popsán výsledek testování v prostředí experimentu AMBER.
Klíčová slova: API, Python, C++, Sběr dat

Title:
Common Client Core interface for Python

Author: Filip Vítek

Abstract: This bachelors thesis is focused on creating an interface that allows user to control dialog library Common Client Core for data acquisition system in COMPASS project at CERN directly from a script written in Python. Basic concepts are explained in the first part a then follows description of COMPASS data acquisition system and Common Client Core library. The main part of the thesis is the solution and implementation of the interface. Conclusion contains results of testing in environment of AMBER project.
Key words: API, Python, C++, Data acquisition

Obsah

Úvod

Vzhledem k velkému množství dat, se kterým se v dnešní době pracuje, je potřeba mít vhodnou technologii k jejich správě. Jazyk Python byl vytvořen především pro účely zpracování velkého množství dat a byl navržen tak, aby byl zdrojový kód co nejlépe čitelný a pokud možno, zkrátil počet řádků potřebných k vyřešení problému na minimum. Ve srovnání s jazykem C++, ve kterém je psaná knihovna Common client core, je Python, právě díky své jednoduché syntaxi, snadnější na naučení. Přestože C++ je podstatně rychlejší, Python je díky své jednoduchosti populárnější a znalost tohoto jazyka je tedy rozšířenější.

Toto tvrzení je patrné z následující statistiky, kde je vidět, že 27.85% všech vyhledávání na Googlu, souvisejících s programovacím jazykem, se týká Pythonu, oproti 7% C++.

Worldwide, May 2022 compared to a year ago:

Rank	Language	Share
1	Python	27.85 %
2	Java	17.86 %
3	JavaScript	9.17 %
4	C#	7.62 %
5	C/C++	7.0 %

Obrázek 1: Statistika hledání informací k programovacímu jazyku [2]

V současné době mají možnost ovládat Common client core na úrovni kódu pouze ti se znalostí C++, pro které je již vytvořeno rozhraní. Cílem této bakalářské práce je vytvořit rozhraní, které promítne celou knihovnu do Pythonu a umožní importovat jednotlivé ovládací třídy knihovny. Tímto bude mnohem většímu počtu lidí umožněno knihovnu ovládat, aniž by se museli učit jazyk C++.

Implementace je prováděna pomocí tzv. *language binding*, neboli propojování jazyků. K promítnutí C++ do Pythonu je k dispozici několik knihoven, z nichž tři jsou popsané v kapitole 5 - Dostupné technologie. Rozhraní bude testováno v prostředí experimentu COMPASS, nicméně nasazeno bude až na nadcházejícím experimentu AMBER.

Práce mi byla nabídnuta Ing. Josefem Novým, Ph.D. Byl jsem od začátku studia rozhodnutý, že jako bakalářskou práci chci programovat a vzhledem k tomu, že C++ a Python patří mezi hlavní dva jazyky, kterým se aktivně věnuji, byl jsem rychle rozhodnutý a práci jsem přijal za svou bakalářskou.

V první kapitole se věnuji základním pojmům a CERNu. Dále systém pro sběr dat experimentu COMPASS a knihovna Common client core. Poté přijdou na řadu kapitoly návrhu řešení, rozebrání dostupných technologií a jejich kompilace. Tyto kapitoly plní roli rešerše. Poslední kapitolou je samotná implementace, kde popíši jak mé nezdařené pokusy, tak ten finální úspěšný, s popisem všech významných problémů, na které jsem narazil.

Kapitola 1

Základní pojmy

1.1 CERN

Zpracováno na základě [7, 8, 11, 10, 12, 13, 16]

Evropská organizace pro jaderný výzkum (původně francouzsky *Conseil Européen pour la recherche nucléaire*, CERN) je mezinárodní organizace se sídlem v Ženevě, zabývající se částicovou fyzikou. Byla zřízena roku 1954. Cílem organizace je spolupráce evropských států (mezi spolupracující státy se roku 1993 připojilo i Česko) v oblasti čistě vědeckého a základního výzkumu, jakož i výzkumu s ním do značné míry souvisejícího. Hlavní oblast CERNu se nachází ve vesnici Meyrin, na kraji Ženevy a nachází se v ní veliké výpočetní středisko, primárně užíváno pro ukládání a zpracování dat, společně se simulováním událostí atp. Organizace se nezabývá činností pro vojenské účely, výsledky jejích experimentálních a teoretických prací se zveřejňují nebo jinak zpřístupňují veřejnosti („*Open Science*“). V současnosti zde jsou prováděny různé pokusy a zkoumání částic menších než atom. Výzkumníci CERNu mají však také zásluhy z oblasti informatiky: ve zdejších laboratořích byl na přelomu 80. a 90. let 20. století vyvinut a spuštěn World Wide Web, jenž se brzo stal vůbec nejrozšířenější a nejužívanější službou na internetu.[7] Vědci v CERNu se zabývají otázkami jako „Co je hmota?, Jak vznikla?, Jak vznikají hvězdy a planety?“, čímž hrají důležitou roli v rozvoji technologie budoucnosti.

1.1.1 Výzkum

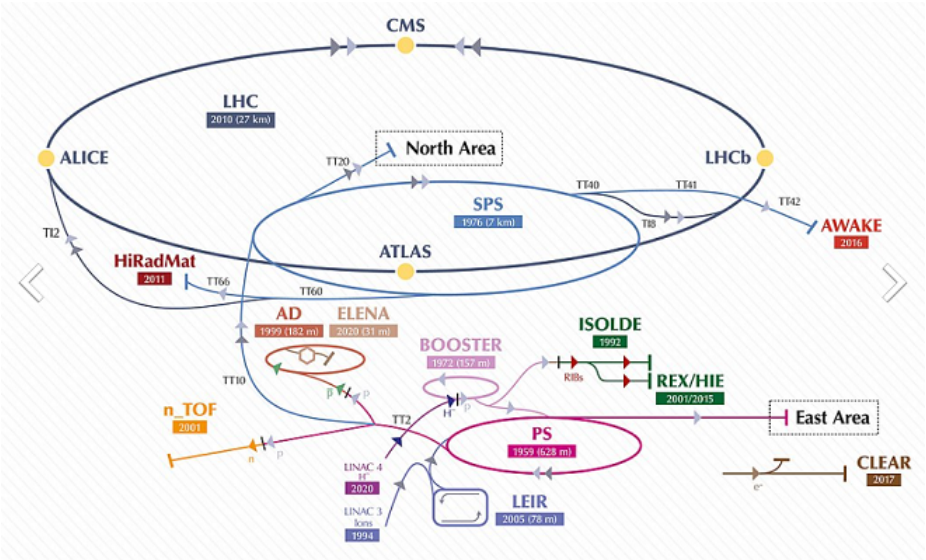
Úkolem laboratoře je porozumět tomu, z jakých součástí je hmota složena a jak tyto součásti spolu interagují. Jeho zařízení, urychlovače a detektory částic, patří mezi největší a nejsložitější vědecká zařízení na světě. [7]

Všechno ve vesmíru je vytvořeno z částic, nicméně částice samotná je z 99,99% prázdná, vědci se zabývají právě těmi hmotnými 0,01%. K vysvětlení hmoty stačí pouze čtyři druhy částic: *kvark u* a *kvark d*, tyto tvoří protony a neutrony, tudíž atomové jádro. Zbylé částice jsou *elektron*, ty společně s jádrem tvoří atom a *elektronové neutrino*. Z těchto částic jsou podle všeho tvořeny planety, slunce atp.

Teorie popisu částic hmoty a částic silových se nazývá standardní model. Ačkoli už po více než 20 let úspěšně prochází všemi experimentálními testy, není standardní model úplným popisem přírody. Fyzikové v CERNu přispívají svou prací k vytvoření dokonalejší představy, jak vesmír funguje. Standardní model zanechává stále příliš mnoho nezodpovězených otázek, aby mohl být konečnou teorií částic a sil. Proč mají částice hmotnost? Jsou různé síly v přírodě jenom jiným pohledem na stejnou věc? Skutečně není ve vesmíru žádná antihmota? To standardní model neříká. [7]

1.1.2 Urychlovače částic

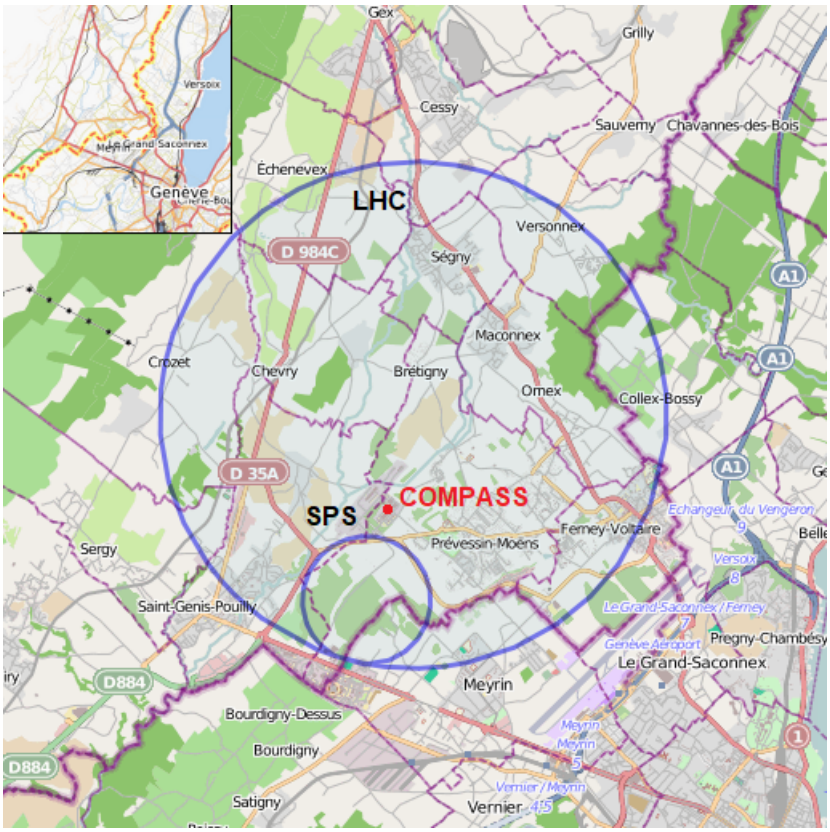
V CERNu je provozováno celkem sedm urychlovačů částic a dva zpomalovače. Tyto připravují částice pro vstřelení do jednotlivých experimentů, potažmo do jiného urychlovače. Částice se střílejí na terče a zkoumá se chování tohoto nárazu. Fyzikům tyto srážky pomáhají odhalit tajemství sil působících mezi částicemi. Existují urychlovače dvou typů: lineární a kruhové. Lineární urychlovače předávají energii svazku při jeho postupném pohybu po celé délce. Platí tedy, že čím delší zařízení, tím vyšší konečná energie. V kruhových urychlovačích částice létají znovu a znovu kolem dokola a hromadí tak energii s každým oběhem kruhu. V CERNu se můžeme setkat s oběma. Urychlovače používají silné elektrické pole, jehož prostřednictvím „nahustí“ energii do svazku částic. Magnetické pole svazek přesně zaostřuje a slouží i k udržení částic na kruhové dráze.[7] Na částice při vysoké rychlosti působí odstředivá síla, stejně jako na auto v zatáčce, proto musí být urychlovače tak veliké, aby magnety zvládaly usměrnit částice do požadovaného směru. Částice se v urychlovači srážejí nebo narážejí na terče uvnitř, přičemž vznikají nové částice. Tři hlavní urychlovače jsou PS, SPS a LHC, všechny kruhového typu.



Obrázek 1.1: Schéma urychlovačů částic v CERNu [8]

PS (*Proton Synchrotron*) je kruhový urychlovač, který byl dostaven roku 1959 a funguje jako podavač částic do většího SPS. Skládá se ze čtyř překrývajících se synchrotronních prstenců, které dostávají svazky záporných vodíkových iontů (H- = atom vodíku s elektronem navíc) z předběžného lineárního urychlovače *Linac4*. PS má obvod 628m.

SPS (*Super Proton Synchrotron*) je se svým obvodem 7km druhý největší urychlovač v CERNu. Uveden do provozu byl roku 1976. Částice z SPS jsou střílenы na terče do experimentů SHINE, COMPASS a NA62. Od roku 2008 (po dostavení LHC) jsou střílenы i do LHC k docílení ještě vyššího zrychlení. Výzkum pomocí svazků z SPS prozkoumal vnitřní strukturu protonů, preferenci přírody ke hmotě oproti antihmotě a zkoumal hmotu jako součást počátků vesmíru. Velký úspěch SPS zažil roku 1983, kdy objev částic W a Z znamenal výhru Nobelovy ceny.



Obrázek 1.2: LHC a SPS na mapě [8]

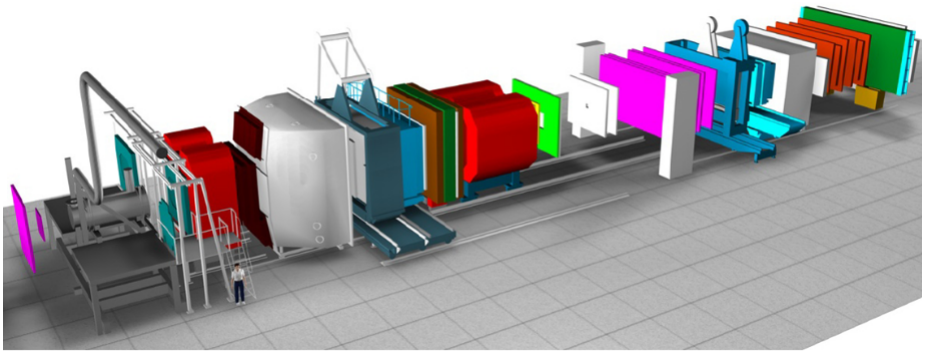
LHC (*Large Hadron Collider*) je doposud největším a nejsilnějším urychlovačem částic na světě, jeho obvod činí 27km a je umístěn 100m pod zemí. V jeho tunelu byl před LHC umístěn urychlovač LEP, který byl zastaven roku 2000. Podél LHC je umístěno celkem 8 experimentů (např.: ATLAS, ALICE), každý zaměřen na jiný výzkum. LHC generuje vysoké množství dat, které je posíláno do laboratoří po celém světě ke zpracování. Velká výzva pro inženýry bylo zajistit srážku dvou svazků, což by se dalo popsat jako vystřelení dvou jehel přes Atlantik, se snahou je srazit.

Poprvé se srážka podařila 30. března 2010 a to byl první krok k objevení Higgsova bosonu. Jde o hmotné skalární elementární částice ve standardním modelu částic a nelze ho pozorovat přímo, jelikož má velice krátkou dobu života. Můžeme jen pozorovat produkty jeho rozpadu. Hraje klíčovou roli ve vysvětlení původu hmotnosti ostatních elementárních částic, zejména rozdílu mezi nehmotným fotonem a velmi těžkými bosony W a Z. Bez existence Higgsova bosonu by vesmírem létaly všechny částice standardního částicového modelu světelnou rychlostí a nebylo by možné, aby utvořily atomy, předměty, planety, hvězdy apod. [9]

V současné době probíhá konstrukce vylepšení LHC na HL-LHC (*High Luminosity*), měl by být dokončen roku 2026 a zajistí řádově vyšší svítivost.

1.1.3 Experiment COMPASS

COMPASS je zkratka pro *COmmon Muon Proton Apparatus for Structure and Spectroscopy*. Experiment byl schválen roku 1997, projekt samotný začal až roku 2002. Oblast výzkumu se nachází v severní části CERNu, poblíž města Prévessin a ke svému výzkumu používá svazky střílené z urychlovače SPS. K výzkumu je potřeba střílet miony (mají vlastnosti jako těžký elektron) a piony na polarizovaný terč.



Obrázek 1.3: Spektrometr experimentu COMPASS [12]

Experiment sleduje komplexní způsoby, kterými spolu kvarky a gluony interagují k vytvoření částic, které dnes pozorujeme, od prostého protonu k širokému výběru exotických částic. Hlavní cíl je zkoumání vnitřní struktury protonů a neutronů a objevit více informací o vlastnosti zvané „spin“. Konkrétně jak moc „spinu“ přispívá pohyb kvarků a gluonů, které vše drží pohromadě. Gluony jsou elementární částice zprostředkující silnou interakci mezi kvarky. Důsledkem působení gluonů je možnost vzniku atomového jádra, neboť umožňuje vytvoření vazby mezi protonem a neutronem v atomovém jádře.[14] Dalším důležitým cílem je prošetřit hierarchii/spektrum částic, které mohou být zformovány kvarky a gluony (například hypotetická částice *„glueball“*, tvořená výhradně gluony[15]).

1.1.4 Experiment AMBER

Jde o budoucího nástupce experimentu COMPASS a jde o zkratku *Apparatus for Meson and Baryon Experimental Research*. AMBER bude stavět na základech COMPASSu a dotáhne je na vyšší úroveň. Proběhne vylepšení současných komponentů a přidají se nové detektory a terče, společně s nejnovější „read-out" technologií (vizuální reprezentace výstupu počítače či věděckého instrumentu).

1.2 API

Zpracováno na základě [3],[4]

API (zkratka pro Application Programming Interface) označuje v informatice rozhraní pro programování aplikací a jde o formu propojování programů, či počítačů. Je to způsob, jakým může programátor využít knihovny, soubory funkcí, třídy atp. API má jednoznačně určené, jakým způsobem se jednotlivé prvky ze zdrojového kódu volají, toto volaní musí být popsané ve *specifikaci API*. Pointa API je umožnit programátorovi použít programové celky, aniž by jejich funkčnost musel sám programovat.

Metaforicky by se API dalo popsat jako *číšník* v restauraci, který obdrží povel od *zákazníka*. Tento povel je vstup a pravděpodobně není ve tvaru, který očekává *kuchař*. *Číšník* tento povel zpracuje a předá *kuchaři* ve formě, kterou je schopen vykonat. Pokud naše API (*číšník*) funguje správně, to co se dostane zpět, neboli výstup procesu, bude námi objednaný pokrm.

Ve srovnání s uživatelským rozhraním, které spojuje počítač/program s člověkem, API spojuje části softwaru. Jeden z cílů API je schovat interní detaily funkčnosti systému a poskytovat pouze ty části programu, které budou užitečné programátorovi. Tyto části je potřeba udržet konzistentní i přes potencionální změny ve zdrojovém kódu.

Vzhledem k tomu, že API je od implementace oddělené, umožňuje nám volat funkce psané v jazyce A z jazyka B, což je cílem této bakalářské práce. Propojování jazyků je považováno za formu API.

1.2.1 API v objektově-orientovaném jazyce

V objektově orientovaných jazycích, se API tvoří popsáním souborů, definic tříd a jejich chování. Pro účely API je tedy potřeba popsat pravidla a chování jednotlivých veřejných metod, konstruktorů a parametrů třídy.Popsat je můžeme libovolným způsobem, důležité je, aby byli asociovány se správným prvkem třídy a bylo ošetřeno její chování.

1.2.2 Příklad z praxe

Jako příklad z praxe si dovolím uvést web, který shromažďuje informace o letech a umožňuje jejich rezervaci (např. *kiwi.com*). Taková stránka v reálném čase zpracovává data o letech ze všech leteckých společností. Celý proces funguje díky API, které tyto společnosti poskytují. Jejich funkce je kromě získání informací o letech také například zarezervování letu atp. Stránka tedy přes API každé společnosti stahuje v reálném čase informace o všech letech (reprezentované například souborem ve formátu *json*[5]) a zpracovává je pro účely svého uživatelského prostředí. Jelikož API umožňují i rezervace, je možné ze stránky let i přímo zarezervovat se všemi potřebnými parametry.

V příjemnějším případě se za volání API neplatí, nicméně například *Google maps* si v průměr účtují 10$ za 1000 volání API (dělí se na různé typy volání s rozdílným ceníkem [6]). Tyto volání můžou být typu *„Vrať mi nadmořskou výšku na těchto souřadnicích!“* nebo i složitější, které ukáží přímo *streetview* na určitě lokaci.

Toto byly stále ale jen příklady webového API. Velmi často pracujeme s Excelovými tabulkami a mnohdy spíše mimo samotný Excel, tj. z jiného programu/skriptu. V situaci, kdy tedy například pomocí Python skriptu pracujeme s Excelovou tabulkou a zpracováváme její data, používáme právě API Excelu. Pro tyto účely je především určený vestavěný programovací jazyk VBA od Microsoftu, který umožňuje skriptově pracovat s tabulkou a tímto způsobem dosáhneme výsledku nejrychleji, nicméně znalost VBA není tolik rozšířená. Zajímavá vlastnost VBA je, že se dá upravovat za běhu a tak efektivně ladit.

1.3 Propojování jazyků

Zpracováno na základě [17]

Propojování jazyků v programování znamená vytváření API, které poskytuje tzv.*„glue code“*, specificky vytvořený k umožnění programovacímu jazyku použít knihovny/systémy psané v jiném jazyce. *„Glue code“* je spustitelný kód který slouží pouze k adaptování různých částí kódu, které by jinak byly nekompatibilní, ale nijak nepřispívá k funkcionalitě ve smyslu splnění požadavků programu.[18]

Propojování obecně znamená namapování jedné věci na jinou. V kontextu softwarových knihoven, propojení je vytvoření „obalu“ knihovny, který funguje jako most mezi dvěma jazyky tak, že knihovna psaná v jazyku A, může být použita z jazyku B. Mnoho knihoven je psáno v systémových programovacích jazycích jako C nebo C++. Tyto knihovny můžeme potřebovat použít ve vyšším jazyku, jako je Python. Musí tedy být vytvořeno propojení do knihovny v jejím jazyce, přičemž vzniká šance na potřebu knihovnu překompilovat v závislosti na objemu potřebných změn.

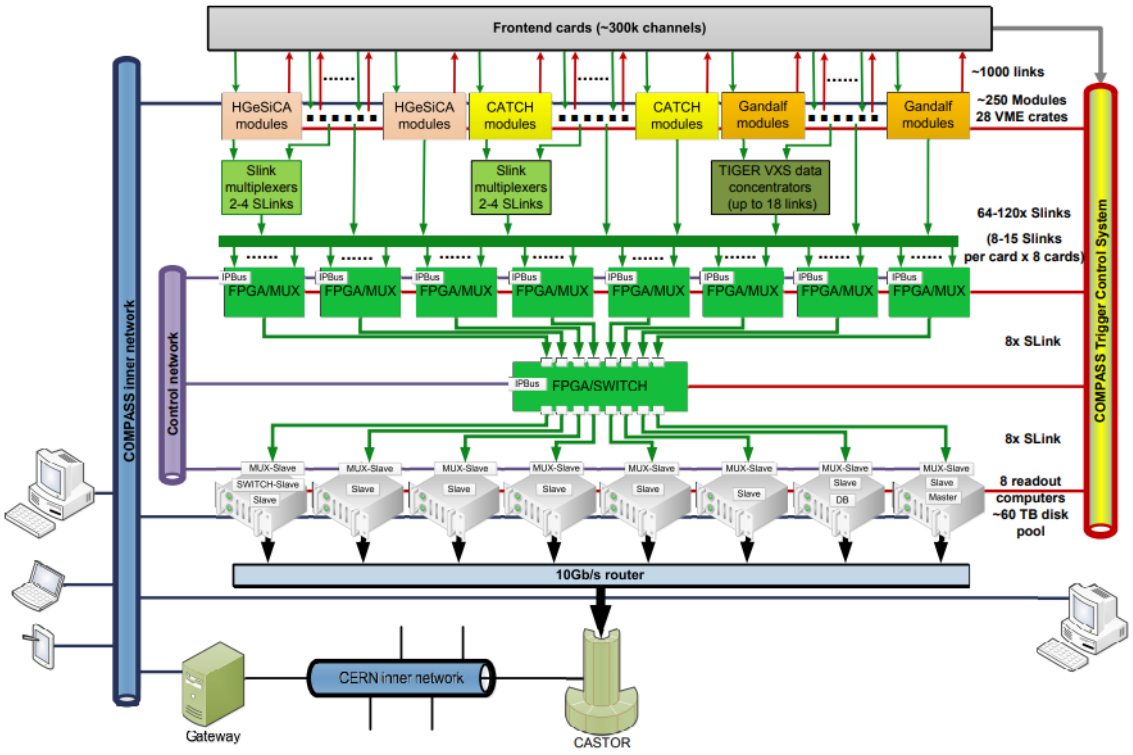
Hlavní motivace k propojování jazyků je za prvé opětovné použití softwaru a tudíž redukování opětovného implementování knihovny v ostatních jazycích. Za druhé, některé algoritmy mohou být hůře implementovatelné v některém z jazyků a tak ani jiná možnost, než propojení s jiným jazykem, neexistuje.

Kapitola 2

Systém pro sběr dat experimentu COMPASS

Zpracováno na základě [19, 20]

Současný systém pro sběr dat (DAQ) COMPASSu je používán od roku 2014, kdy nahradil starý systém. Ten byl ve službě od samotného počátku experimentu. Jeho účelem je čtení dat (*Raw Data*) týkajících se fyzikálních událostí z detektorů a jejich uložení. Tyto data jsou následně poslány do systému CASTOR (*CERN Advanced STORage manager*) - hierarchicky řízený systém ukládání ke správě fyzikálních datových souborů v jednotkách petabytů.



Obrázek 2.1: Struktura DAQ [19]

Vzhledem k nesmírně vysokému počtu fyzikálních událostí, které v urychlovači a ve spektrometru proběhnou, je nemožné všechny tyto data ukládat a následně analyzovat. Při tomto problému jsou zužitkovány tzv. *„Trigger Systems“* („Spouštěčové systémy“), neboli TS. TS se v reálném čase rozhodne, zda bude událost zobrazena, či vyřazena. Toto rozhodnutí je ovlivněno mnoha faktory, které zajišťují, že pouze data událostí relevantních k fyzikální analýze budou zachycena a tím se výrazně snižuje objem zachycených dat o několik řádů velikosti, čímž šetříme úložiště.

Samostatná fyzikální událost je téměř pokaždé popsaná daty z několika detektorů najednou. Celkem je 300 000 kanálů [2.1], ze kterých DAQ obdrží data. Frekvence spouště může být až 50 kHz s průměrnou velikostí jedné události 36kB. SPS funguje v cyklech, každý trvá přibližně 10 sekund, po které je svazek střílen do spektrometru. Následuje přibližně 40 sekund bez svazku. Skupinky kanálu jsou pomocí multiplexeru[1] převáděny do jediného kanálu. Tato fáze multiplexování má 3 vrstvy [2.1]. Jakmile je dokončeno, data jsou odeslány do FPGA přepínače [2.1], který událost sestaví. Každá událost je tak sjednocena do jednoho průběžného kanálu, přestože pocházeli z několika tisíc detektorů. Nakonec se data vypíšou v počítači pomocí *„Spillbufferu“* a uloží se. *„Spillbuffer“* je PCI Express karta s FPGA čipem, která se stará o připojení S-Link a RAM počítače. Nad celým procesem je „COMPASS Inner Network“, který ovládá a dohlíží na celý proces [2.1].

2.1 Profil uživatele

Existuje několik softwarových systémů, které ovládají a monitorují různé části experimentu COMPASS. Například systém CESAR (*CERN Experimental areas SoftwAre Renovation*) je používán k ovládání aspektů souvisejících se směrem letu svazku částic („Beam line“), čímž ho uživateli umožňuje v nějaké míře upravovat. Další příklad je DCS (*Detector Control System*), který poskytuje v reálném čase informace o jednotlivých detektorech.

Při procesu sbírání dat je nezbytné, aby operátoři experimentu byli přítomni v kontrolní místnosti a dohlíželi na celý proces a stav experimentu. Vzhledem k tomu, že během období sběru dat se data sbíraní 24 hodin denně 7 dní v týdnu, je potřeba, aby byl někdo neustále přítomen. Každý člen experimentu má povinnost strávit určitý čas v kontrolní místnosti a dohlížet na experiment.

Operátoři na experimentu, jsou často odborníci na pouze jednu věc a jejich znalost ostatních součásti může být nízká. Operátoři se často obměňují, přičemž nováček je zaučen jednou 8hodinovou směnou, která ale přirozeně nemůže stačit, většina se tak učí průběžně za běhu.

[1] Elektronická součástka s n ovládacími vstupy, 2^n datovými vstupy a jedním výstupem. Slouží k serializaci dat z několika kanálů.

2.2 Ovládací systém

DAQ Control System je softwarový systém určený k ovládání a monitorování posledních tří vrstev hardwaru (FPGA Multiplexery, FPGA Přepínač a vypisovací engine [2.1]). Systém se skládá z několika procesů:

- **Slave**[2] **readout** - Proces, běžící na počítačích, které čtou fyzikální data a stará se o jejich zpracování.
- **Slave control** - Proces, běžící na čtecích počítačích, který se stará o monitorování a konfiguraci FPGA karet, za použití přímé komunikace přes sběrnici.
- **Run control GUI** - Grafické uživatelské rozhraní určené k ovládání DAQ.
- **Master** - Proces běžící na konkrétním počítači. Tento proces slouží jako prostředník v komunikaci mezi procesy v DAQ. Jakoukoliv operaci musí *Master* schválit, stará se o chyby a celkově na vše dohlíží. Nemá nadvládu sám nad sebou, tj. nemůže sám sebe ukončit.
- **Message logger** - Proces sbírající zprávy ze *Slavů* a *Masterů* a ukládá je v databázi. Tyto zprávy se dělí na čtyři typy:

  1. Informace
  2. Varování
  3. Chyba
  4. Fatální chyba

- **Item browser** - Grafické rozhraní umožňující uživateli v reálném čase zobrazit jednotlivé zprávy z „Message loggeru“, včetně starších zpráv z databáze.

„Master“ a všechny „slave“ procesy mají připojené stavové automaty, které indikují povahu činnosti procesu v určitý čas.

DAQ Control system používá vlastní komunikační knihovnu k síťové komunikaci. Je implementována v C++ za použití Qt knihoven a nabízí C++ API. Je založena na principu služeb, příkazů a kontrolního serveru, který funguje jako prostředník počáteční síťové komunikace. Služby fungují jako prostředky jedné až n komunikací. Jakmile proces zaregistruje službu na kontrolním serveru, stane se *„Publisherem“* („podavatel“) a všechny data podané touto službou budou poslány přímo do všech procesů, které službu odebírají. Ostatní procesy mohou začít odebírat od této služby tak, že zažádají kontrolní server o síťové informace a hned po obdržení se připojí přímo k podavateli.

[2]Slave (otrok) je část plně ovládaná Masterem

2.3 FSM

Finite state machine, nebo FSM (v překladu „Konečný automat [27]“), je teoretický výpočetní model používaný v informatice pro studium formálních jazyků. Popisuje velice jednoduchý počítač, který může být v jednom z několika stavů, mezi kterými přechází na základě symbolů, které čte ze vstupu. Množina stavů je konečná (odtud název). Konečný automat nemá žádnou další paměť, kromě informace o aktuálním stavu. Konečný automat je velice jednoduchý výpočetní model, dokáže rozpoznávat pouze regulární jazyky. Formálně je konečný automat definován jako uspořádaná pětice hodnot [27]:

- S - konečná neprázdná množina stavů
- Σ - konečná neprázdná množina vstupních symbolů, nazývaná abeceda
- σ - přechodová funkce, popisující pravidla přechodů mezi stavy
- s - počáteční stav
- F - množina finálních stavů

Na počátku se automat nachází v definovaném počátečním stavu. Dále v každém kroku přečte jeden symbol ze vstupu a přejde do stavu, který je dán hodnotou, která v přechodové tabulce odpovídá aktuálnímu stavu a přečtenému symbolu. Poté pokračuje čtením dalšího symbolu ze vstupu, dalším přechodem podle přechodové tabulky atd. [27] Schéma tohoto modelu je vyobrazeno na obrázku 2.2.

DAQ COMPASSu běží na principu FSM. Jeho hlavní účel je poskytnout programátorům pomůcku k psaní aplikací řízených stavy. FSM může běžet jak synchronně[3], tak asynchronně[4]. Samotné FSM je také reprezentováno vlastní třídou *fsm_representation*. Přechod mezi stavem se uskutečňuje voláním metod:

```
FSM::increase_state()
FSM::decrease_state()
FSM::go_to_error_state()
```

Tyto metody jsou bezpečné pouze při synchronním běhu. Při asynchronním běhu je potřeba použít metody s parametry:

```
increase_state(state from, state to)
```

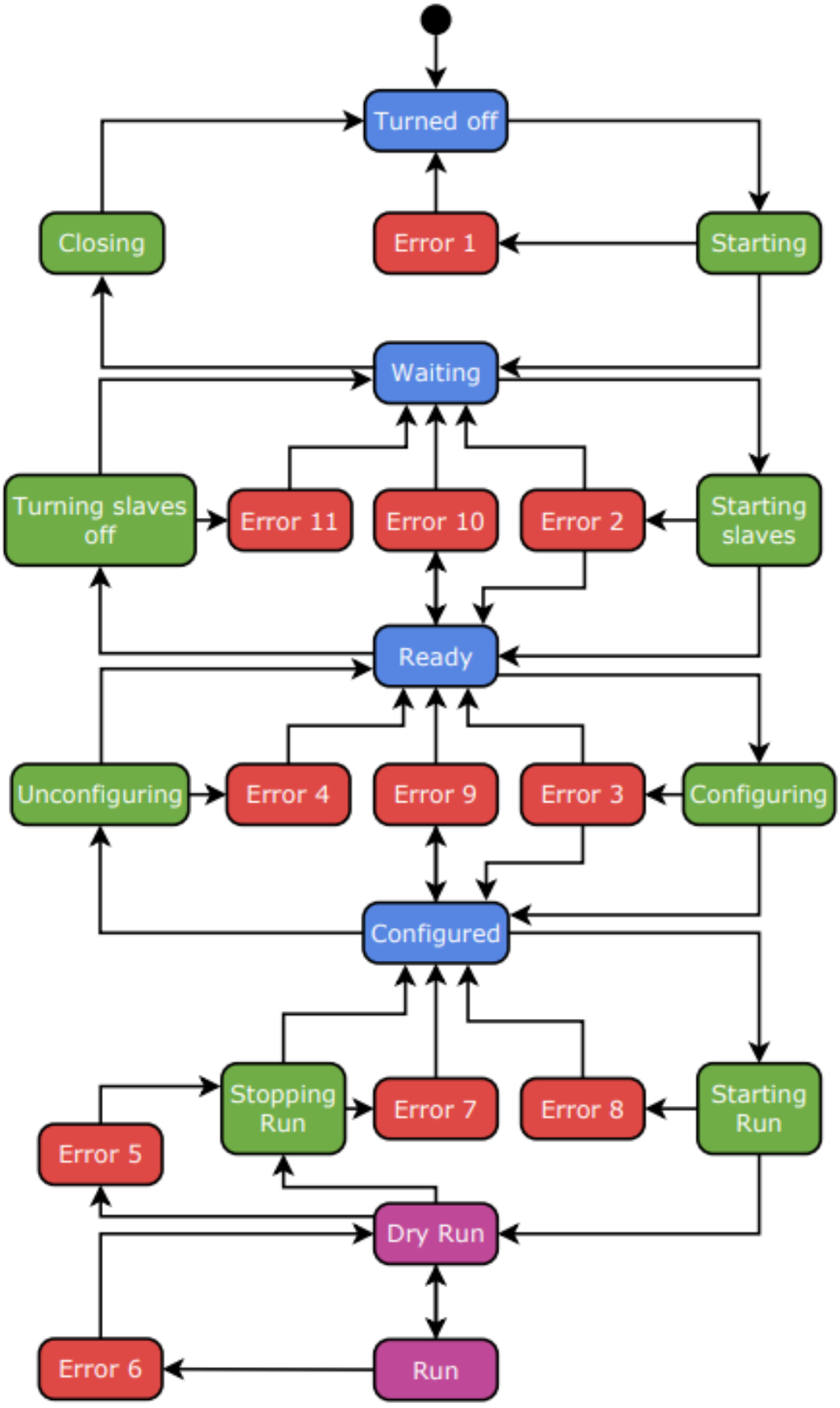
[3] systém vždy čeká na dokončení procesu

[4] je označení pro takovou podobu vstupu/výstupu, při které které mohou být během čekání na dokončení vstupního/výstupního datového přenosu souběžně prováděny jiné procesy [26]

FSM poskytuje *Qt*[5] signály a virtuální metody pro každý stav FSM. Například pro stav *ready* jsou poskytnuty následující metody [25]:

```
virtual bool can_enter_ready()
virtual bool can_leave_ready()
virtual void on_ready_increase()
virtual void on_ready_decrease()
virtual void on_ready_entered()
virtual void on_ready_entered_from_above()
virtual void on_ready_entered_from_below()
virtual void on_ready_left()
```

[5]CCC je postaveno na knihovnách *Qt*



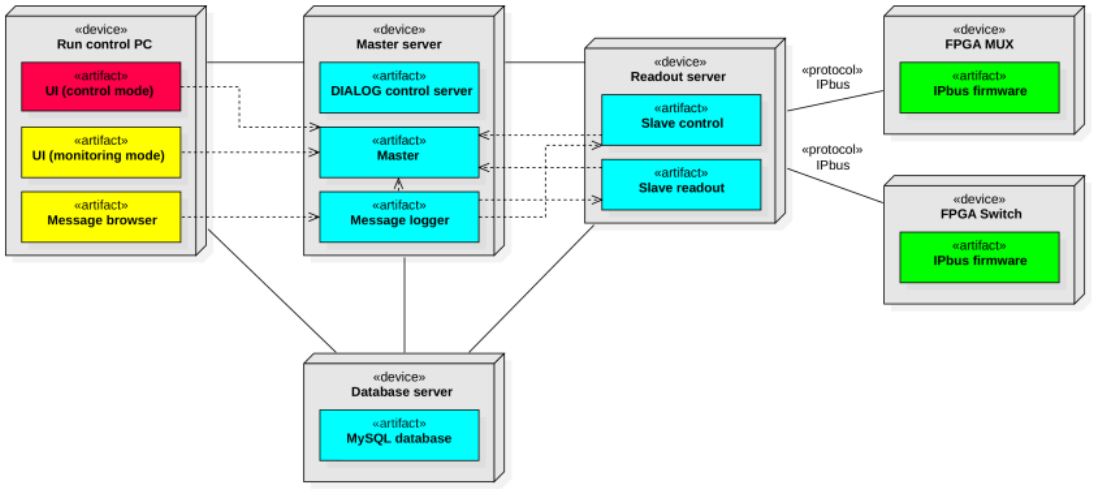
Obrázek 2.2: Diagram stavového automatu Master procesu [19]

Kapitola 3

Common Client Core

Zpracováno na základě [19, 21]

Několik rozhraní bylo nedílnou součástí frameworku od jeho počátku, zejména *Run Control GUI*. Během let 2015 a 2018 bylo implementováno API a rozhraní pro vzdálené ovládání systému, avšak byla náročná na údržbu, především kvůli nedostatečnému počtu softwarových inženýrů. Toto vedlo ke změně přístupu k rozhranní frameworku a návrhu nové architektury rozhraní.



Obrázek 3.1: Architektura softwarového frameworku iFDAQ [21]

3.1 Požadavky rozhraní

Typy potřebných rozhraní byli identifikovány jako:

- **GUI pro plochu počítače** - K použití v kontrolní místnosti. Důraz na UX[1], protože na tomto rozhraní se mají učit noví členi experimentu.
- **UI pro vzdálený přístup** - Určeno pro experty, k vyřešení problémů na dálku.
- **Programovací API** - Promítnutí ovládání a monitorování do ostatních systémů.
- **Skriptovací API** - Umožnění rychlé a jednoduché tvorby vlastních uživatelských pomůcek.

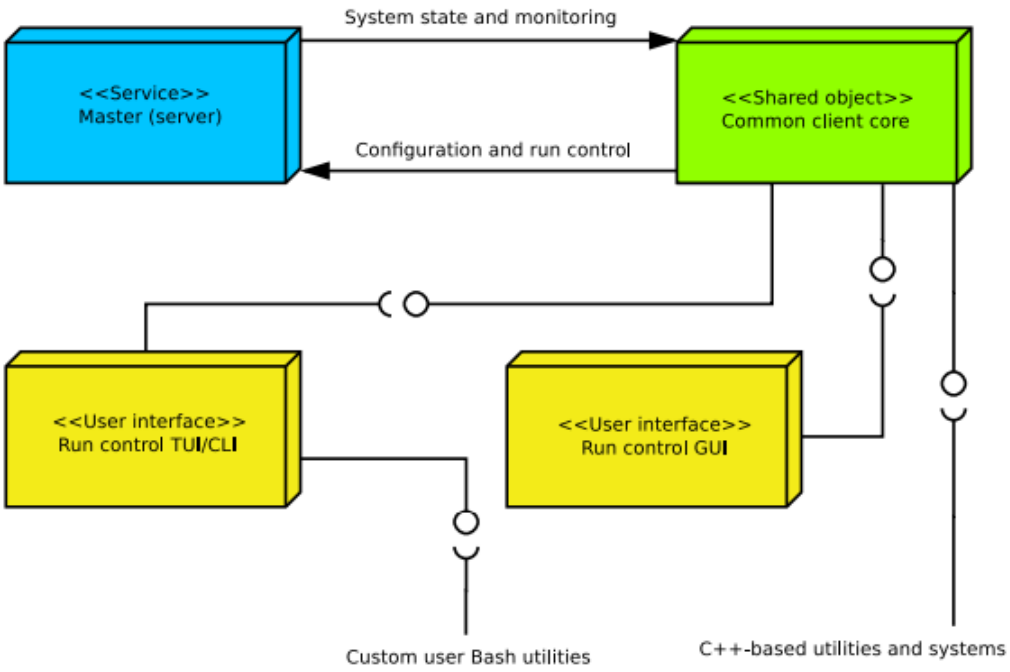
Z důvodu snahy dosáhnout maximální účinnosti pracovní síly, byl kladen důraz na možnost opětovného použití rozhraní pro rozdílné účely. Jak ve staré, tak v nové architektuře, *Master* procesy/služby fungují jako prostředník mezi rozhraními a zbytkem systému, zejména *Slave* procesy. Zatímco ve staré architektuře by klienti sdíleli jen malý komunikační protokol. Nová architektura představila koncept *„Common Client Core“*.

3.2 Implementace

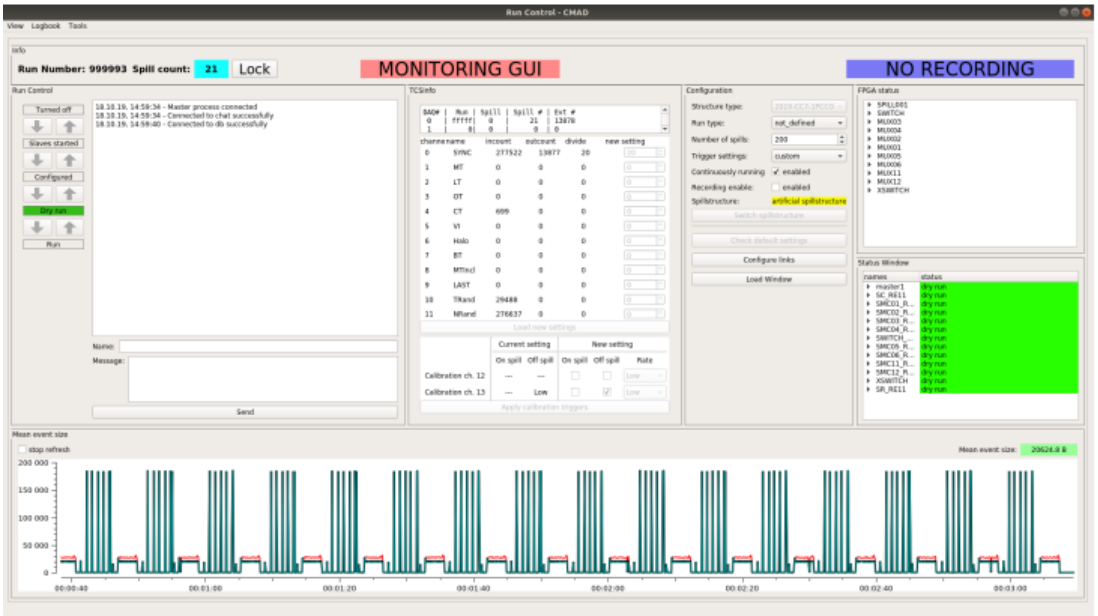
CCC, implementováno jako sdílená objektová knihovna (*Shared Object Library*) s C++ API, se stará o komunikace s *Master* službami. Při inicializaci začne odebírat *Masterovy* DIALOG služby a periodicky obdrží zprávy jak s informacemi týkající se stavu systému, tak monitorování. Logika funkčnosti je „parsovat“[2], uložit a zpřístupnit tyto informace uživateli ve formě C++ funkcí. Jakožto událostmi poháněné rozhraní, události jsou uživateli k dispozici jako Qt signály. CCC také sjednocuje komunikaci zasílanou *Master* službě, zejména konfigurace a *Run Control* příkazy. Díky tomu, že CCC obsahuje informace o stavu systému a poskytuje funkce konfigurace a ovládání za běhu, může být použit jako základ jakéhokoliv uživatelského rozhraní frameworku, stejně jako C++ API k iFDAQ systému, které může poté být použit cizím systémem. Nepodporuje ale komunikaci do jiné podsítě, API musí tedy běžet na stejné síti jako *Master*.

[1] User Experience - pohodlý uživatele při používání softwaru

[2] Rozebrat nějaký datový celek na formát použitelný programově



Obrázek 3.2: Architektura nového rozhraní iFDAQ [21]



Obrázek 3.3: Základní okno *Run Control GUI* [21]

3.3 Run Control GUI

Tento balíček zajišťuje pohodlné ovládání za běhu pro uživatele, určený pro kontrolní místnost COMPASSu. Zabere tři 1920x1080 obrazy v základním nastavení. Stav systému je reprezentován ve formě GUI elementů.

V základním okně [3.3] vidíme v levé části GUI pro stav systému, viz [2.2], s tlačítky pro přepínání mezi jednotlivými stavy a nad ním číslo běhu a *Spill*[3] *Count*. Vedle vidíme *chatbox*, kde mohou komunikovat jak operátoři pracující uvnitř spektrometru s kontrolní místnosti, tak přímo *Master* proces s uživatelem. Po pravě straně vidíme okénka zobrazující stavy multiplexerů a dalších prvků hardwaru DAQ. Ve spodní části je graf událostí a průměr jejich velikosti.

3.4 Run Control TUI[4]/CLI[5]

Velikost *Run Control GUI* ji dělá velmi nepraktickou pro vzdálený přístup. Vzdálený uživatel se musí spolehnout na *„X-Forwardování“*[6] přes několik SSH, což není uspokojivé řešení, jelikož výkon takto velikého GUI poskytovaného tímto způsobem je nesmírně slabý. Přestože existují způsoby, jak zlepšit tento výkon, rozhodlo se pro jiné řešení vzdáleného přístupu.

Jeden návrh byl webové rozhraní, ale to by bylo v konfliktu s novým přístupem, tj. důrazem na účinnost pracovní síly, protože by to znamenalo programování API v jazyku uzpůsobeném pro vývoj webu (PHP...), čímž by se zdvojnásobila zátěž CCC. Výhodnější řešení bylo použít dynamické terminálové textové rozhraní, jehož příkazy by mohli být volány i v módu příkazové řádky a tudíž vytvoření bash API.

3.4.1 Mód terminálu

Terminál tohoto rozhraní nabízí dynamickou vizualizaci stavu iFDAQ, včetně možnosti ovládat. Při zapnutí jsou příkazy, rozdělené do podnabídek, k dispozici na základě současného stavu systému. Monitorovací příkazy zužitkovávají *ANSI Escape Sequences*[7] k poskytnutí dynamického výstupu, ve kterém jsou hodnoty okamžitě obnoveny při aktualizování dat. Příkazy vyžadující vstup od uživatele jsou spuštěny jako *switche* a parametry. Pokud všechny parametry nejsou poskytnuty, jsou doplněny interaktivně, tj. uživatel bude požádán o poskytnutí nezbytných hodnot. Tento mód je zároveň kompatibilní s chytrými telefony.

[3] Svazek urychlených částic odebraných z prstencového SPS a vstřelených do spektrometru
[4] Textové uživatelské rozhraní
[5] Rozhraní příkazové řádky
[6] Způsob identifikování IP Adresy připojujícího se klienta skrz proxy server. [22]
[7] Standard pro sázení v textových terminálech

3.4.2 Mód příkazové řádky

Příkazová řádka umožňuje uživateli podat jeden nebo více příkazů do spustitelného binárního souboru a ty budou vykonány, stejně jako by byly v terminálu. Zatímco ovládací příkazy jsou identické s jejich protějšky v módu terminálu, monitorující příkazy poskytují strojem čitelný výstup (řetězce jsou nahrazeny jmenovanými číselnými hodnotami a výstup je vypsán podle předdefinovaného formátu). *ANSI escape sequences* zde nejsou použity.

Hodnoty proměnných monitorování nejsou ukládány do mezipaměti a proces se může opětovně připojovat k DIALOG serveru a čekat na aktualizaci, což způsobuje přibližně sekundové zdržení. Toto dělá mód příkazové řádky v současném stavu nevhodný pro vysokovýkonné aplikace.

3.5 Souhrn

Nová architektura rozhraní iFDAQ má za cíl významně redukovat z dlouhodobého hlediska zátěž vývojářů a zjednodušit přidání jednotných nových funkcí rozhraní. Jak TUI, tak CLI, mají využití mezi členy experimentu a *Run Control GUI* se ukázal jako přínosný přírůstek k iFDAQ softwarovému frameworku.

3.6 Použití

Zpracováno na základě [28]

Programátor musí nejprve inicializovat *daq_ client* a poté přístup k jeho obsahu, kdykoliv ho potřebuje. *Daq_ client* obsahuje *gettery* a *settery* k třídám *model* a *controller*. Třída *model* slouží k získání informací o stavu DAQ a *controller* je k ovládání DAQ. Obsah třídy *model* jsou automaticky aktualizovány na pozadí v reálném čase. Z důvodu asynchronnosti procesu je důležité zmínit několik bodů.

K některému obsahu může být přistoupeno pouze v určitých stavech. Proto mnoho metod hlásí výjimky, když je stav nevhodný pro volání dané metody. Je nezbytné tyto výjimky zachytit a vyřešit. Vždy, když uživatel přistoupí ke *cachovanému* obsahu v *daq_ clientu*, je vytvořena kopie. K přístupu k těmto kopiím je určena vlastní metoda.

Dva body je nutné zmínit ohledně třídy *controller*. K ovládání DAQ se musí uživatel nejprve uzamknout do ovládání pomocí metody:

```
controller::lock_control()
```

Pokud by program nebyl uzamknut do ovládání, nebo ve kterýkoli moment ztratí stav zamknutí, spuštěné metody budou bez efektu. Mnoho metod používá „defensivní parametry“, aby nedošlo k poskytnutí špatných argumentů. Toto znamená, že je nejdříve potřeba získat argumenty ze třídy *model*.

Takto by vypadal minimalistický příklad spuštění CCC [28].

```
#include <QCoreApplication>
#include "daq_client.h"
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    config_consts::get_instance().read_consts(app.arguments().at(1));
    daq_client::get_instance()
                .self_contained_start("DAQ_state_checker", &app);
    while(!daq_client::get_instance().get_model().state_initialized()){
    }
    while(true){
        QTextStream(stdout) <<
                 FSM::get_state_name(daq_client::get_instance()
                .get_model().get_master_fsm_copy()
                .get_current_state()) << "\n";
        sleep(1);
    }
    return app.exec();
}
```

3.7 Změny v CCC při přechodu z COMPASSu na AMBER

Základ zprovoznění COMPASS softwaru na AMBERu znamenalo pro CCC především odstranění všeho souvisejícího s *triggery*, viz kapitola 2. Další krok byl přepsání *master* procesu, kde byly provedeny velké změny v meziprocesové komunikaci. Veškerá serializace se začala provádět ve formátu *.json*. K tomuto všemu bylo potřeba adaptovat CCC, aniž by se samotné rozhraní změnilo. Následovalo předělání databázové knihovny a registrové knihovny. Nicméně ve výsledku je to z architekturálního a funkčního hlediska stejné rozhraní.

Kapitola 4

Návrh řešení

Řešení, ve své podstatě, je poměrně prosté, ale obnáší velké množství práce, jak je patrné z UML[1] diagramu knihovny v příloze. Je potřeba každou veřejnou metodu nebo parametr obalit do „schránky“ podle dokumentace používané technologie propojování jazyků, více k možnostem mezi technologiemi v kapitole 5.

4.1 Popis třídy UML diagramem

Na obrázku 4.1 můžeme vidět příklad jedné, té s nejvyšším počtem veřejných prvků, ze 23tříd knihovny CCC a všechny její metody a parametry. Parametry jsou zobrazeny v prostředním panelu pod názvem třídy (byli by zde zobrazeny například i enumerátory) a spodní panel obsahuje metody[2] třídy. Jednotlivé prvky mají před svým prvním znakem znaménko „+“ nebo „-“, toto jsou specifikátory přístupu. Znaménko „+“ znamená, že metoda/parametr jsou veřejné, přístupné odkudkoliv a tudíž všechny prvky třídy s tímto znaménkem budou promítnuty do Pythonu skrz rozhraní. Znaménko „-“ symbolizuje soukromé prvky, tedy prvky, u kterých je možný přístup/použití jen vně třídy. Pro tyto účely existuje ještě třetí parametr „#“, jež není v bloku použit, ten symbolizuje *protected* prvek. Kompletní diagram v příloze obsahuje pro účely reprezentace objemu práce pouze právě veřejné prvky se znaménkem „+“. Vytvoření kompletního diagramu včetně soukromých a *protected* metod by zabralo zbytečně mnoho místa v práci.

Rozdíl u psaní názvů prvků oproti zdrojovému kódu C++ je patrný, na prvním místě není datový typ parametru, potažmo datový typ výstupu metody. V UML diagramech je tato informace reprezentována až za dvojtečkou. Pokud je tedy za názvem metody a jejími argumenty „ : bool“, výstup té metody bude *boolean*(*True/False*) hodnota. Obdobně pro argumenty metody, zde je ale navíc používám specifikátor vstupu *in/out/inout*. Značka *in* znamená „vstupní hodnota, která nebude měněna“, argument s *out* bude po průběhu metody vrácen ve výstupu, tedy bude modifikován, a poslední *inout* má význam „vstup, který může být modifikován“.

[1] *Unified Modeling Language* - grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů [24]

[2] pozn.: UML standard nemá předepsaný formát pro specifickou reprezentaci *Virtual* metody

Obrázek 4.1: Blok UML diagramu třídy *Controller* z CCC [23]

4.2 Postup řešení

Pro účely rozhraní nás tedy zajímají pouze veřejné prvky. Knihovna CCC se skládá z hlavičkových souborů jednotlivých tříd[3] a souborů s implementací metod. Je nutné si jednotlivé hlavičky prohlédnout a zjistit, které prvky bude potřeba obalovat do modulu pro API. Všechny tyto prvky je poté potřeba v *.cpp* souboru obalit do modulu. Podrobnému popisu jednotlivých problémů a principů, jak obalit všechny případy, které nastávají v CCC, se budu věnovat v kapitole 7. Modul lze vytvořit jak přímo v *.cpp* soubory s definicemi jednotlivých metod, tak v samostatném souboru určeném konkrétně jen pro modul. Příslušná technologie poté při kompilování vyvolá makro, které vygeneruje *shared object* (*.so*) soubory a ty lze importovat do Pythonu. Po úspěšném zkompilování bude tedy v adresáři knihovny vygenerován *.so* soubor, který bude importovatelný do Pythonu. Je možné importovat i pod jiným názvem, než název souboru, ale pro přehlednost a pohodlnost použití jsou veškeré importy pojmenované jako zdroj. Detailnějšímu popisu vytváření modulů se budu věnovat v kapitole 5.

4.3 Výsledný produkt

CCC bude možné zapnout a použít přímo ze skriptu v Pythonu. Pro Python existují všechny potřebné *Qt* knihovny k zapnutí CCC, vše bude tedy provedeno jako v minimalistickém příkladu v kapitole 3. Ve výsledku je tedy možné volat jakýkoliv veřejný prvek ze zdroje z příslušného importu standardním způsobem pro balíčky v Pythonu:

```
import ImportovatelnyModul
Vystup = ImportovatelnyModul.metoda(argumenty,...)
```

Přestože konkrétní požadavky od expertů detektoru se týkali pouze určitých metod, a to poměrně malého počtu, bude rozhraní vytvořeno pro celou knihovnu. Nepočítá se tedy s příliš velikým využitím celkové šířky rozhraní, protože mnoho metod slouží spíše k použití software expertem a ten pravděpodobně bude umět s C++, takže nemá důvod používat Python API. Testována bude prioritně funkčnost stěžejních metod, tj. především metod z požadavků od expertů detektoru a několik dalších. Mezi hlavní požadavky patří metody umožňující:

- *Lock DAQ* - zamknutí za účelem ovládání
- Získání stavu FSM *Mastera*
- Nastavení parametrů struktury
- Nastavení struktury *Spillu*
- Nastavení parametrů běhu
- Nastavení monitorování
- Zapnutí/Vypnutí běhu

[3] 22/23 hlaviček obsahuje právě jednu třídu, hlavička *daqstructure.h* obsahuje více menších tříd, které byli ve starších verzích CCC také v samostatných hlavičkách

Kapitola 5

Dostupné technologie

Technologie, kterými by se tento úkol dal uskutečnit, je mnoho, vyjmenuji tedy postupně technologie, na které jsem narážel já při implementaci. Ve své podstatě všechny fungují na stejném principu, jen jejich moduly vypadají mírně odlišně a finální kompilace probíhá jinak.

5.1 Boost Python

Boost je velmi známý soubor knihoven se širokým pokrytím různých problémů v programování. Jde o knihovnu určenou striktně pro C++. Cílí na použitelnost v širokém spektru aplikací a je dostupná zdarma. Knihovny fungují na téměř všech operačních systémech. [29]

Mezi desítkami balíčků knihovny *Boost* je i baliček *Python*. Účel tohoto balíčku je právě promítnutí C++ to Pythonu skrz modul *BOOST_PYTHON_MODULE*. Balíček obsahuje podporu pro [30]:

- Reference a ukazatele
- Souvislosti mezi typy
- Automatické přetypování
- Účinné přetěžování funkcí
- Překlad výjimek z C++ do Pythonu
- Přednastavené argumenty
- Argumenty s klíčovým slovem
- Manipulování Python objekty v C++
- Export C++ iterátorů jako Python iterátory
- Dokumentační řetězce

5.1.1 Implementace

K promítnutí C++ do Pythonu balíčkem *Boost.Python* je potřeba použít tzv. *BOOST_PYTHON_MODULE*, kterým obalíme všechny funkce, metody a proměnné, které chceme přenášet. Tento proces ukážu na minimalistickém příkladu ze stránky dokumentace *Boost.Python*[30]. Máme jednoduchou funkci, která po zavolání pozdraví skrz příkazovou řádku:

```
char const* pozdrav()
{
        return "Ahoj, entito!"
}
```

Nyní je potřeba funkci obalit do modulu, to provedeme takto:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(robot)
{
    using namespace boost::python;
    def("pozdrav", pozdrav);
}
```

Argument (*robot*) modulu je právě ten název, pomocí kterého budeme v Pythonu později importovat. Funkci můžeme promítnout pod libovolným názvem, zde jsme zvolili pro přehlednost název stejný, jako má promítaná funkce, tj. „pozdrav“. Další krok je zkompilování *.cpp* souboru jako *Shared Object*. To vygeneruje soubor *.so*, standardně ve stejném adresáři jako zdrojový kód. Tento *.so* soubor můžeme poté importovat v Pythonu a funkci zavolat:

```
>>> import robot
>>> print robot.pozdrav()
"Ahoj, entito!"
```

Stěžejní krok přichází po napsání modulu, je potřeba správně zdrojový kód zkompilovat, aby byl umožněn import v Pythonu. *Boost.Python* k tomu poskytuje vlastní skript zvaný *bjam* [31]. Tento skript vyžaduje *jamfiles*, které obsahují informace o souborech, které chceme kompilovat. Pokud je *jamfile* v adresáři se skriptem *bjam* a všechny cesty k souborům, kompilátorům atp. jsou správně, po spuštění skriptu se vytvoří požadované *Shared Object* soubory a bude umožněno importovat. Toto je doporučená cesta, způsobů kompilování je mnoho. Příklad jednoduchého *jamfilu* k našemu předchozímu minimalistickému příkladu by vypadal takto:

```
project-root ;
import python ;

extension robot
:  robot.cpp
  cestaKBalickuBoost/libs/python/build/extension
  ;
#zde lze pridat Python skripty, ktere otestuji funkcnost
```

5.1.2 Výsledek

Boost.Python byl první volba, jak úkol vyřešit. Projekt se dostal pomocí této knihovny poměrně daleko. Celé CCC bylo obaleno tímto způsobem a na řadě byla kompilace. Zde tento první pokus ztroskotal. Knihovnu samotnou nebyl problém zkompilovat, v syntaxi modulů tedy problém nebyl. Při snaze kompilovat skrz *bjam*, ale nebyly *.so* soubory vygenerovány správně a nebylo možné importovat do Pythonu. Situaci zároveň příliš nepomáhala úroveň dokumentace k *Boost.Python* balíčku, která nebyla dostatečná, a tento způsob řešení také dle různých diskuzí na internetu není dnes ideální, a tak se od *Boost.Pythonu* odstoupilo.

Zde je příklad modulu[1] pro třídu *controller* vytvořeného pomocí *Boost.Python*:

```
#include "controller.h"

BOOST_PYTHON_MODULE(controller)
{
  using namespace boost::python;
  class_<controller,bases<QObject>,
           boost::noncopyable>("controller",no_init)
    .def("lock_control",&controller::lock_control)
    .def("is_control_locked",&controller::is_control_locked)
    .def("send_chat_message",&controller::send_chat_message)
    .def("runcontrol",&controller::runcontrol,
          return_value_policy<reference_existing_object>())
    .def("tcs_control",&controller::tcscontrol,
          return_value_policy<reference_existing_object>())
    .def("select_structure_type",&controller::select_structure_type)
    .def("select_run_type",&controller::select_run_type)
    .def("toggle_recording_enabled",
          &controller::toggle_recording_enabled)
    .def("toggle_continuously_running",
          &controller::toggle_continuously_running)
    .def("toggle_port",&controller::toggle_port)
    .def("mask_port_error",&controller::mask_port_error)
    .def("set_monitoring_prescaler",
          &controller::set_monitoring_prescaler)
    .def("set_udp_monitoring_source",
          &controller::set_udp_monitoring_source)
    .def("set_spill_structure",&controller::set_spill_structure)
    .def<void (controller::*)(const ifdaq_device*,one_register,
          quint32)>("set_registry_value",
          &controller::set_registry_value)
    .def<void (controller::*)(const ifdaq_device*,
          QString,quint32)>("set_registry_value",
          &controller::set_registry_value)
     ;
}
```

Jde o verzi třídy *controller* ze srpna 2021, která prošla změnami, proto se počet veřejných prvků liší od UML digramu v této práci.

[1] **include <boost/python.hpp>** se nachází v hlavičce

5.2 Cython

V rámci ztroskotání prvního pokusu se začaly hledat alternativy, mezi kterými se vyskytla knihovna *Cython*. Knihovna funguje oboustranně a poskytuje vlastní stejnojmenný programovací jazyk, který funguje jako obousměrný most mezi jazyky. Jazyk *Cython* je založen na Pythonu, lze tedy funkce promítnout pouze do *Cythonu* a použití bude téměř stejné[2].

Implementace je velice podobná předchozí variantě. Rozdílem je, že modul musí být ve vlastním souboru s příponou *.pxd*. Jednoduchá třídu z dokumentace *Cythonu* [32] Obdelnik.h s definicemi metod v Obdelnik.cpp:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Obdelnik {
    public:
        int x0, y0, x1, y1;
        Obdelnik(int x0, int y0, int x1, int y1);
        int vratPlochu();
    };


#endif
```

Cython modul by se psal do souboru *Obdelnik.pxd* a vypadal by takto:

```
cdef extern from "Obdelnik.cpp":
    pass

cdef extern from "Obdelnik.h":
    cdef cppclass Obdelnik:
        Obdelnik(int, int, int, int) except +
        int x0, y0, x1, y1
        int vratPlochu()
```

Této knihovně jsem se ale příliš dlouho nevěnoval a odstoupil jsem od ní.

5.3 Ctypes

S *Ctypes* jde o trochu jiný přístup, protože jde o „knihovnu pro cizí funkce“ (*„Foreign Function Library“*) jazyku python. Poskytuje datové typy kompatibilní s C++ a dovoluje volání funkcí z *.dll*. Rozhodně jde o cestu, kterou lze promítnout C++ to Pythonu, ale z důvodu komplexnosti datových typů, které jsou používány v CCC, jsem tento způsob také škrtl. Nebyl jsem si jistý, zda by to správně fungovalo.

[2] A implementace jednodušší

5.4 Pybind11

Zpracováno na základě [33].

PyBind11 je „lightweight“[3] čistě hlavičková[4] knihovna. Dokáže propojovat jak z C++ do Pythonu, tak naopak. Její cíle a syntaxe je podobná *Boost.Pythonu*. Hlavní problém s *Boost* je jeho enormní velikost a komplexní nabídka pomocných balíčků, které fungují téměř se všemi C++ kompilátory. Tato úroveň kompatibility má svou cenu, protože znamená použití různých triků a zkratek, z důvodu podpory starších a méně kvalitních kompilátorů. Tato vlastnost je s novými kompilátory spíše nazbyt. Proto vzniknul *Pybind11*, malá samostatná, avšak ne totožná, verze *Boost.Python*. Podporuje:

- Funkce přijímající/vracející vlastní datové struktury, reference, či ukazatele
- Instance metod a statické metody
- Přetížené funkce
- Instance atributů a statické atributy
- Libovolné výjimky
- Enumerace
- Zpětné volání
- Iterátory a intervaly
- Vlastní operátory
- Jednu či více dědění
- Datové struktury STL
- Chytré ukazatele s počítáním referencí
- Vnitřní reference s počítáním referencí
- C++ třídy s virtuálními metodami

Dokumentace k *PyBind11* je velice přehledná a kvalitní. Knihovna je navíc stále ve vývoji a každý týden přibývají nové commity v repositáři, který je veřejný na adrese [33].

[3] záměrně jí chybí funkce, aby poskytovala jednodušší API s méně vazbami na externí prostředky
[4] čistě hlavičkovou knihovnu je jednodušší přidat do cesty projektu

5.4.1 Implementace

Implementujeme opět pomocí modulu, zde se jmenuje *PYBIND11_MODULE*. Použití ukáži opět na minimalistickém příkladu, máme jednoduchou sčítací funkci v souboru *priklad.cpp* a tu obalíme do modulu:

```
int secti(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(priklad, m) {
    m.doc() = "scitani"; // dokumentace k balicku
    m.def("secti", &secti, "funkce, co secte dve cisla");
}
```

Makro *PYBIND11_MODULE* vytvoří funkci, která bude zavolána při importu v Pythonu. Prvni argument je „priklad“, jde o název modulu. Druhý argument „m“ definuje proměnou typu *py::module_*, což je hlavní rozhraní pro vytváření propojení. Metoda *module_::def* promítá funkci do Pythonu. Nyní stačí už jen zkompilovat. K možným způsobům více v kapitole 7.

Vzhledem k jednoduchosti, přehlednosti a velmi rozšířené znalosti této knihovny, jsem se rozhodl naprogramovat API CCC pro Python pravě v *PyBind11*.

Kapitola 6

Možnosti kompilace PyBind11

Zpracováno na základě [33, 35].

Možností, jak modul zkompilovat a vytvořit tak správně *.so* soubory k importu, je více. Popíši tři z nich, které jsem zvažoval k použití.

6.1 Manuální build

Jelikož *PyBind11* je čistě hlavičková knihovna, není potřeba ji propojovat s žádnými dalšími knihovnami a nejsou potřeba žádné speciální kroky. Na Linuxu se dá jednoduchý příklad ze sekce 5.4 zkompilovat tímto příkazem v terminálu, za předpokladu, že používáme *Python3*[1]:

```
$ c++ -O3 -Wall -shared -std=c++11 -fPIC $(python3 -m pybind11
  --includes) robot.cpp -o robot$(python3-config --extension-suffix)
```

První závorka poskytne cestu k hlavičkám *PyBind11* a Pythonu. Toto řešení předpokládá, že byl *PyBind11* nainstalován pomocí *pip* nebo *conda*. Pokud nebyl, můžete specifikovat cestu manuálně společně s cestou k Pythonu. Poté poskytnete příkazu soubor, který má kompilovat a parametr k vytvoření *.so* souboru. Jelikož v příkladu používáme *Python3*, můžeme použít „–extension-suffix“, ten ale nemusí být vždy dostupný. Jinak může být přípona definována přímo jako *.so*. Takový příkaz by vypadal takto:

```
$ c++ -O3 -Wall -shared -std=c++11 -fPIC $(python3 -m pybind11
  --includes) robot.cpp -o robot.so
```

Toto řešení slouží spíš k promítnutí jednoduchých modulů a není pro rozsáhlejší knihovnu praktické, proto je lepší v tomto případě použít nějakou z jiných dostupných možností.

[1] Jinak lze do příkazu napsat místo „python3“ „python/python2“

6.2 CMake

CMake je multiplatformní svobodný software pro automatizaci překladu programu v různých operačních systémech. Používá se pro vytvoření adresářové struktury a přípravu zdrojových souborů pro použití s konkrétními překladači. Například program *make* na Unixových strojích, Xcode firmy Apple, nebo Microsoft Visual Studio na systému Windows. CMake zvládá generovat výstup jak do stejné složky, ve které se nachází zdrojové soubory, tak do předem vybraného adresáře. Nastavuje se konfiguračním souborem *CMakeLists.txt*, ten se musí nacházet v kořenovém adresáři projektu.[34]

Pro C++ program s existujícím *CMake* kompilovacím systémem může být propojovací modul vytvořený jen pomocí pár řádků kódu[2]:

```
cmake_minimum_required(VERSION 3.4...3.18)
project(robot LANGUAGES CXX)
add_subdirectory(pybind11)
pybind11_add_module(robot robot.cpp)
```

Zde uvažujeme, že složka *pybind11* je v adresáři s *robot.cpp*. *CMake* příkaz *add_subdirectory* importuje *PyBind11*, který poskytne funkci *pybind11_add_module*. Ta se postará o všechny detaily potřebné ke zkompilování Python modulu. Samotná funkce očekává parametr *name* (*robot*), což bude název modulu postaveného z vypsaných zdrojů (*robot.cpp*). Funkce může dostat více parametrů, příklad je minimalistický. Modul *robot* může být poté manipulován jinými *CMake* příkazy. *Pybind11_add_module* přidá argumenty kompilátoru k zajištění maximální kvality generování kódu. Postará se o problémy související s použitím implementací z rozdílných verzí *PyBind11*. Povoluje se také LTO[3]. Bez LTO se jednotlivé soubory zdrojového kódu kompilují nezávisle a až poté se propojí, při LTO se zpracovávají a používají informace závislostí mezi vnitřními moduly všech zdrojových souborů, což vede ke zvýšení optimalizace. Pokud není složka s *pybind11* v adresáři, je možné ji importovat z jiné lokace pomocí funkce *find_package*:

```
cmake_minimum_required(VERSION 3.4...3.18)
project(robot LANGUAGES CXX)
find_package(pybind11 REQUIRED)
pybind11_add_module(robot robot.cpp)
```

Funkce bude fungovat správně pouze, pokud byl balíček *PyBind11* správně nainstalován, tj. po stažení či naklonování repositáře. Další kroky se nijak neliší, při detekování balíčku se vše importuje stejně jako při použití *add_subdirectory*.

V základním nastavení bude zkompilováno dle minimálních standardů požadovaných *PyBind11*, nebo dle minimálních standardů kompilátoru, na základě toho, který standard je vyšší. Standard se dá nastavit pomocí *CMAKE_CXX_STANDARD*.

[2] CXX = C++ v *CMake*
[3] *Link Time Optimization*

6.3 Cppimport

Cppimport je malé Python rozhraní k importování, které zjistí, zda existuje C++ zdrojový soubor, jehož jméno se shoduje s požadovaným modulem. Pokud soubor existuje, je zkompilován jako rozšíření pro Python za použití *PyBind11* a umístí příslušné *.so* soubory do adresáře s C++ souborem. Python je poté schopný modul najít a načíst.

Balíček lze nainstalovat příkazem *pip install cppimport*. Aplikovat na náš minimalistický příklad *priklad.cpp* ze sekce 5.4 by šel tímto způsobem:

```
// cppimport
#include <pybind11/pybind11.h>

int secti(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(priklad, m) {
    m.doc() = "scitani"; // dokumentace k k balicku
    m.def("secti", \&secti, "funkce, co secte dve cisla");
}

/*
<%
setup_pybind11(cfg)
%>
*/
```

Poté spustíme Python skript:

```
import cppimport.import_hook
import priklad #Zde se nachvilku kod pozastavi kvuli kompilaci
print(priklad.secti(1,2))

Output >>  3
```

Komentář na první řádku v souboru *priklad.cpp* není zbytečně, ale je důležitý pro vyhledávání souborů s *PyBind11* moduly. *Cppimport* prohledá všechny zdrojové soubory v kořenovém adresáři a všech podadresářích a hledá právě řetězec „cppimport“ na prvním řádku. Tímto způsobem se může zkrátit doba kompilace, v případě CCC toto nehrozí, jelikož je do Pythonu promítána celá knihovna.

Na konci souboru *priklad.cpp* je ale další komentář, který má také speciální roli, jde o konfigurační blok. Část uzavřená do *<% .....%>* je blok *Mako*[4] kódu. Tato část je vyhodnocena jako Python kód během kompilace a poskytuje *cppimportu* konfigurační informace o kompilátoru.

[4] *Mako* je šablonová knihovna psaná v Pythonu. Poskytuje známou syntaxi, která se překompilovává do Pythonu pro maximální výkonnost. Jeho syntaxe a API čerpá z nejlepších nápadů ostatních, jako *Django, Cheetah* a další. Jde v podstatě o vestavěný Python. [36]

Pro vydávání softwaru poskytuje *cppimport* jednoduchý příkaz, který předkompiluje všechny poskytnuté zdrojové soubory.

```
python -m cppimport build
```

Tento příkaz může být použit v *CI/CD pipeline*[5], ale není dělaný pro Windows. Prochází všechny adresáře a podadresáře a zkompiluje všechny *.c* a *.cpp* soubory, které jsou určeny požadavky k importování do Pythonu, tj. mají na prvním řádku řetězec „cppimport“. Na konec příkazu lze dopsat externí adresáře, které si také přejete zkompilovat, pokud ale vede cesta ke konkrétnímu souboru a ne k adresáři, je soubor automaticky považován za „určený pro import“ a řetězec „cppimport“ se nekontroluje.

Kompilaci lze libovolně konfigurovat, k tomu slouží právě již zmíněný *Mako* kód, do kterého můžeme vepsat parametry dostupné v dokumentaci [35]. Například k použití C++11 se dopíše do *Mako* bloku parametr:

```
cfg['extra_compile_args'] = ['-std=c++11']
```

Cppimport nepřekompilovává při každém spuštění, pouze při prvním spuštěním či pokud proběhnout nějaké změny. Můžeme přidat další závislosti do konfigurace, jako jsou hlavičkové soubory:

```
cfg['dependencies'] = ['hlavicka1.h', 'hlavicka2.h']
```

[5] *Continuous Integration/Continuous Deployment* - Série kroků, které musí být provedené před vydáním nové verze softwaru. Jde o různé testování zaměřené na konkrétní věci, analýzy atp. U velkých projektů je každý *Pull Request* ve verzovacím systému doprovázen tímto procesem a pouze při úspěšném splnění všech testů je povolen *merge*. Stávají se situace, kdy *CI* neprojde, ale změny v kódu nemohli nic v daném testu rozbít. V takovém případě může při důkladném zkontrolování administrátor přesto *merge* schválit a ignorovat upozornění ze *CI*.

Kapitola 7

Implementace

7.1 Obalení prvků pomocí PyBind11

Prvním krokem samotné implementace je vytvoření *PyBind11* modulů. Rozhodl jsem se vytvořit pouze jeden modul, který obsahuje všechny CCC. Modul jsem nazval *pythonapi* a nachází se kvůli cppimportu ve stejnojmenném souboru *pythonapi.cpp*. Tento soubor obsahuje pouze modul *PYBIND11_MODULE(pythonapi,m){...}*, veškeré příklady z této sekce jsou umístěny právě v tomto modulu, na pozici „tří teček".

Většinu metod CCC šlo obalit jednoduše dle postupu popsaném v sekci 5.4, vzniklo ale mnoho případů, kdy jsem se musel na dokumentaci [33] obrátit více do hloubky, či i hledat externí návody. Budu se v této sekci sedmé kapitoly soustředit na popis řešení všech problémů a případů, které při implementaci modulu vznikly. Samotná implementace jednotlivých metod není pro modul důležitá, jako zdroj budu tedy udávat pouze hlavičky tříd, obsahující definice metod a parametrů. Dovolím si v příkladech vynechat řádky s *#include* ve vrchní části hlavičkových souborů, protože pro demonstraci implementace nemají hodnotu.

Mnohokrát v předchozích kapitolách byla zmíněná třída *Controller*, jejíž veřejná část obsahuje pouze konstruktor a metody. Přestože jde o třídu s nejvyšším počtem veřejných prvků, byla implementace jejího obalu monotónní a bez problému. Část hlavičky *Controller.h*:

```
class Controller : public QObject
{
    Q_OBJECT

public:
    Controller(Model &model);
    void startSlaves();
    void stopSlaves();
    void configure();
    void unconfigure();
    ...
}
```

V sekci 5.4 jsme obalovali pouze funkci. Zde lze ukázat, jak obalit třídu, její konstruktor, metody a předat informaci o předkovy. Obal k této části třídy by vypadal takto:

```
pybind11::class_<Controller, QObject>(m,"Controller")
            .def(pybind11::init<Model &>())
            .def("startSlaves",&Controller::startSlaves)
            .def("stopSlaves", &Controller::stopSlaves)
            .def("configure", &Controller::configure)
            .def("unconfigure", &Controller::unconfigure)
            ...
        ;
```

Knihovna *PyBind11* obsahuje šablonu *class_*, která jako datový typ očekává třídu, kterou obalujeme a její předky. Jako argumenty očekává *py::module*, který jsme definovali jako *m* v *PYBIND11_MODULE*, a řetězec, kterým se tato třída bude používat v Pythonu. Pro pohodlnost použití je každá třída poskytována Pythonu pod stejným názvem, jaký má v C++. Takto vytvoříme třídy a můžeme začít obalovat její metody a parametry, k tomu slouží metody *.def()*, kterých je několik typů:

Základní

- *.def()* - Používané pro metody a funkce. Očekává řetězec, kterým se bude volat (v mé implementaci jsou všechny totožné s C++). Dále očekává metodu, kterou bude promítat a dobrovolný argument je popis této metody, tuto možnost jsem nevyužil, na funkci nemá ale žádný vliv. Alternativně, jako v případě konstruktoru, můžete do argumentu vložit příslušnou šablonu z *PyBind11*.
- *.def_static()* - Určené pro statické[1] metody.
- *.def_readwrite()* - Určené pro parametry, které chceme číst i měnit.
- *.def_readonly()* - Určené pro parametry s klíčovým slovem **const**. Parametr nebude možné měnit.
- *.def_property()* - Určené pro soukromé parametry.

Kombinace

- *.def_property_readonly()*
- *.def_readwrite_static()*
- *.def_readonly_static()*
- *.def_property_static()*
- *.def_property_readonly_static()*

[1] může být zavolána, přestože objekt není inicializován

Promítnutí konstruktoru provedeme pomocí šablony *init*, kterou poskytneme jako argument metodě *.def()*. Šablona očekává datové typy všech argumentů konstruktoru. Zbytek případů v této třídě jsou metody. Popis promítnutí metod je v předchozím výpisu *.def()* metod.

Třída *Controller* ale obsahuje i jeden speciální případ, jde o přetížení metody. Tento případ se nikde jinde nevyskytoval:

```
class Controller : public QObject
{
    Q_OBJECT
public:
    ...
    bool togglePort(QString moduleName, quint32 portId);
    ...
private:
    ...
    bool togglePort(QSharedPointer<Port> port, bool enable);
    ...
}
```

Zajímá nás pouze veřejná metoda, nicméně *PyBind11* potřebuje specifikovat, kterou přesně metodu máme na mysli. Je potřeba mu tedy poskytnout informaci o výstupním datovém typu a datových typech argumentů, tak jsme schopni přesně metodu identifikovat, protože přetěžování metod je právě na rozdílech v datových typech vstupů a výstupů založeno:

```
pybind11::class_<Controller, QObject>(m,"Controller")
            ...
            .def("togglePort", static_cast<bool (Controller::*)
                (QString, quint32)>(&Controller::togglePort))
            ...
        ;
```

Implementace obalu statické metody lze ukázat na třídě *DaqClient*:

```
class DaqClient : public QObject
{
    Q_OBJECT

public:
    static DaqClient& getInstance();
    void stop();
    void start();
    ...
}
```

Metoda funguje jako takový mezikrok inicializace objektu. Pokud ji zavoláme, vytvoří se instance třídy *DaqClient* a je vrácena ve výstupu metody. Metoda je obalena tímto způsobem:

```
pybind11::class_<DaqClient, QObject>(m, "DaqClient")
            .def_static("getInstance", &DaqClient::getInstance)
            ...
        ;
```

Stále jsme nenarazili na žádný případ, kdy by se promítal jen parametr, nicméně v celé knihovně ani žádný veřejný parametr neexistuje. Můžeme na ně narazit pouze v rámci „enumerátoru“ a uvnitř typu *struct*. Oba tyto případy pokrývá třída *Port*, kde se mezi veřejnými prvky nachází jak *enum*, tak *struct*:

```
class Port
{
public:
    enum class State
    {
        UP,
        DOWN,
        DISABLED,
        NULL_SOURCE_ID,
        INCORRECT_SOURCE_ID,
    };
    struct Error
    {
        bool masked = false;
        quint32 count = 0;
        QString description;
        QString registerName;
        QString registerPartName;
        Error(QString description_, QString registerName_,
              QString registerPartName_ = "") :
              description(description_), registerName(registerName_),
              registerPartName(registerPartName_){}
    };
    ...
}
```

V tomto případě je implementace mírně odlišná od předchozích. Šablona *class_* je určená i pro typ *struct*, ale potřebujeme jí dát informaci o tom, že je součástí jiné třídy. S *enum* je to podobné, nicméně v *PyBind11* existuje vlastní šablona *enum_*:

```
    pybind11::class_<Port>port(m,"Port");
    pybind11::enum_<Port::State>(port, "State")
            .value("UP", Port::State::UP)
            .value("DOWN", Port::State::DOWN)
            .value("DISABLED", Port::State::DISABLED)
            .value("NULL_SOURCE_ID", Port::State::NULL_SOURCE_ID)
            .value("INCORRECT_SOURCE_ID",
                    Port::State::INCORRECT_SOURCE_ID)
            .export_values()
            ;
            ...
```

Jelikož potřebujeme předat informaci o třídě, ve které je parametr definován, je nutné přiřadit šabloně *class_* konkrétní proměnnou, zde „port“. Poté použijeme šablonu *enum_*, ale na místo, kde bychom běžně použili typ *py::module*, použijeme právě proměnnou třídy *Port* a tím přesně říkáme, že tento *enum* je uvnitř třídy *Port*. Poté jen jednoduše obalíme jednotlivé hodnoty a metodou *.export_values()* exportujeme hodnoty do předka.

Pro *struct* je implementace obdobná, jen je potřeba při obalování parametrů používat metody *.def()*:

```
...
pybind11::class_<Port::Error>(port,"Error")
            .def(pybind11::init<QString,QString,QString>())
            .def_readwrite("masked", &Port::Error::masked)
            .def_readwrite("count", &Port::Error::count)
            .def_readwrite("description", &Port::Error::description)
            .def_readwrite("registerName", &Port::Error::registerName)
            .def_readwrite("registerPartName",
                           &Port::Error::registerPartName)
            ;  ...
```

Nejsložitější problém souvisí s abstraktními třídami. Třída *IfDaqDataDevice* je abstraktní:

```
class IfDaqDataDevice : public BaseDevice
{
public:
    IfDaqDataDevice(QSharedPointer<ControlSlave> slave,
                    QSharedPointer<IfDaqNode> node);
    virtual ~IfDaqDataDevice();
    quint32 getSliceNumber();
    quint32 getSpillNumber();
    quint32 getCurrentSourceId();
    quint32 getPreviousSliceNumber();
    quint32 getPreviousSpillNumber();
    virtual quint32 getCurrentSpillData() = 0;
    virtual quint32 getPreviousSpillData() = 0;
    virtual const QSharedPointer<Port> getPort(quint32 portId) = 0;
};
```

Pokud se tuto třídu pokusíme obalit standardním způsobem, dostaneme chybu „Allocating an object of abstract class type 'IfDaqDataDevice'". Problém je s konstruktorem. Pokud bychom ho zakomentovali, vše proběhne v pořádku, ale konstruktor nelze vynechat. Řešení je ve vytvoření tzv. „odrazové" třídy. Cíl je vytvořit třídu, která dědí z *IfDaqDataDevice*, ale modulu ji podáme naopak. „Odrazová" třída vypadá takto:

```
class IfDaqDataDeviceTrampoline : public IfDaqDataDevice{
public:
    using IfDaqDataDevice::IfDaqDataDevice;
    quint32 getCurrentSpillData() override{
        PYBIND11_OVERLOAD_PURE(quint32,IfDaqDataDevice,
                               getCurrentSpillData);
    }
    quint32 getPreviousSpillData() override{
        PYBIND11_OVERLOAD_PURE(quint32,IfDaqDataDevice,
                               getPreviousSpillData);
    }
    const QSharedPointer<Port> getPort(quint32 portId) override{
        PYBIND11_OVERLOAD_PURE(const QSharedPointer<Port>,
                               IfDaqDataDevice,getPort,portId);
    }};
```

Nyní máme vše připravené k obalení:

```
    pybind11::class_<IfDaqDataDevice, IfDaqDataDeviceTrampoline,
                                BaseDevice>(m, "IfDaqDataDevice")
            .def(pybind11::init<QSharedPointer<ControlSlave>,
                 QSharedPointer<IfDaqNode>>())
            .def("getSliceNumber", &IfDaqDataDevice::getSliceNumber)
            .def("getSpillNumber", &IfDaqDataDevice::getSpillNumber)
            .def("getCurrentSourceId",
                 &IfDaqDataDevice::getCurrentSourceId)
            .def("getPreviousSliceNumber",
                  &IfDaqDataDevice::getPreviousSliceNumber)
            .def("getPreviousSpillNumber",
                 &IfDaqDataDevice::getPreviousSpillNumber)
            .def("getCurrentSpillData",
                 &IfDaqDataDevice::getCurrentSpillData)
            .def("getPreviousSpillData",
                 &IfDaqDataDevice::getPreviousSpillData)
            .def("getPort", &IfDaqDataDevice::getPort)
            ;
```

Kromě tříd CCC je potřeba obalit i některé prvky z externích knihoven. Mezi tyto patří *QString* z důvodu, že některé metody CCC očekávají tento typ v argumentu, nicméně přetypování z Python *stringu* na *QString* není automatické, tudíž je potřeba ho také promítnout do Pythonu a patřičné argumenty zadávat jako instanci objektu *QString* v Pythonu.

Další externí třída, která je potřeba promítnout, je *ConfigConsts*[2]. Tato třída se stará o nahrání nastavení systému a potřebuji z ní pouze dvě metody.V modulu obě tyto třídy vypadají takto:

```
pybind11::class_<ConfigConsts, std::unique_ptr<ConfigConsts,
                  pybind11::nodelete>>(m,"ConfigConsts")
            .def_static("getInstance", &ConfigConsts::getInstance,
                pybind11::return_value_policy::reference)
            .def("readFile", &ConfigConsts::readFile)
            ;
pybind11::class_<QString>(m,"QString")
            .def(pybind11::init<const char *>());
```

Jako poslední potřebuji přidat enumerátor z knihovny *FSM*, pro zobrazování stavu stavového automatu. Zobrazení implementace si dovolím zkrátit, jde o 32 řádků principiálně stejného kódu:

```
    pybind11::enum_<FSM::State>(fsm,"State")
            .value("TURNED_OFF", FSM::State::TURNED_OFF)
            .value("STARTING_ERROR", FSM::State::STARTING_ERROR)
            ...
            .value("STOPPING_RUN", FSM::State::STOPPING_RUN)
            .value("RUN_ERROR", FSM::State::RUN_ERROR)
            .value("RUN", FSM::State::RUN);
```

[2]Jde o *Singleton*

7.2 Kompilace

7.2.1 Cppimport

První pokus o zkompilování *PyBind11* modulu a vygenerování importovatelného *.so* souboru jsem zkusil pomocí balíčku *Cppimport*, dostupného v *Python 3*. Princip použití je popsán v kapitole 5. Celá dokumentace zní velice jednoduše a jak je zvykem, když to zní příliš dobře, bude tam nějaký háček. Potřeboval jsem tedy napsat konfigurační *Mako* kód v souboru s *PyBind11* modulem, společně s komentářem *// cppimport* na prvním řádku souboru. *Mako* kód má za účelem poskytnout kompilátoru všechny potřebně informace, tj. cesty ke zdrojům *Common Client Core*, cesty k hlavičkovým souborům externích knihoven, předvolby pro kompilaci atp. Poté by mělo stačit soubor s modulem zkompilovat a ve stejném adresáři se vytvoří *.so* modulu.

Při psaní konfiguračního kódu se vyskytly problémy. Podařilo se mi nastavit několik parametrů pro kompilaci, přidal jsem cesty ke všem potřebným složkám s hlavičkovými soubory a přidal jsem jednotlivé hlavičky a zdrojové soubory *Common Client Core*. Po spuštění kompilace pomocí *cppimport* se knihovna úspěšně zkompilovala, nicméně importovat nebylo možné, protože modul obsahoval prvky z externích knihoven, které nebyly poskytnutý kompilátoru. Další problém byl s *moc* soubory *Qt* knihoven, pro zpracování *Qt* maker jsem nenašel řešení v dokumentaci *Cppimport*, tento způsob kompilace tedy nebyl úspěšný a odstoupil jsem od něj směrem k *CMake*.

Finální *Mako* konfigurační kód:

```
<%
cfg['extra_compile_args'] = ['-std=c++11']
cfg['include_dirs'] = ['../include',
'port',
'device',
'../include/port',
'../include/device',
'/usr/include/qt5/QtCore',
'/usr/include/qt5/QtNetwork',
'/usr/include/qt5/QtXml',
'/usr/include/qt5/QtSql',
'/usr/include/qt5/QtGui',
'/usr/include/qt5',
'/online/subsystems/daq/RCCARS/amber-rccars-daq/
common-client-core/include',
'/online/subsystems/daq/RCCARS/amber-rccars-hlt/common/include',
'/online/subsystems/daq/RCCARS/amber-rccars-daq/
DIALOGCommunication/include',
'/online/subsystems/daq/RCCARS/amber-rccars-daq/registrylib/include',
'/online/subsystems/daq/RCCARS/amber-rccars-daq/databaselib/include',
'/online/subsystems/daq/RCCARS/amber-rccars-daq/daq-structure/include',
'/online/subsystems/daq/RCCARS/amber-rccars-daq/fsmlib/include',
'/online/subsystems/daq/RCCARS/amber-rccars-daq/common/include',
'../build']
cfg['dependencies']=['../include/SafeStopLog.h',
'../include/Controller.h',
'../include/DaqClient.h',
'../include/ChatLog.h',
...
'../include/device/SpillbufferDevice.h',
'../include/device/SwitchDevice.h',
'../include/device/TcsControllerDevice.h']
cfg['sources']=['Controller.cpp',
'DaqClient.cpp',
'ChatLog.cpp',
'HardwareModel.cpp',
'Model.cpp',
'ProcedureCaller.cpp',
'SafeStopLog.cpp',
...
'device/SpillbufferDevice.cpp',
'device/SwitchDevice.cpp',
'device/TcsControllerDevice.cpp']
cfg['libraries'] = ['../build/libcommonClientCore.so']
setup_pybind11(cfg)
%>
```

Při zpětném čtení kódu vidím, že problém s propojením s externími knihovnami by pravděpodobně byl opraven přidáním jednotlivým cest k *.so* souborům knihoven ve spodní části konfiguračního kódu, do **cfg['libraries']**, avšak problém s *Qt* makry by stále přetrvával. Tento kód fungoval, kompilace proběhla, *.so* soubor modulu byl vytvořen, nicméně při importu modulu do Pythonu se objevovali chyby.

7.2.2 CMake

Celá podstata kompilace pomocí *CMake* je napsat konfigurační kód do souboru *CMakeLists.txt*. Jde principiálně o stejnou věc, jako v *Mako* kódu předchozí podkapitol, nicméně dokumentace *CMake* je podstatně kvalitnější a každý problém se řeší mnohem jednodušeji, také kvůli tomu, že podobné problémy přede mnou již řešilo mnoho lidí a existuje mnoho článku a příspěvků zaměřených na konkrétní problém.

Obecný plán, jak soubor napsat, byl v úvodu takový:

- Nastavit parametry pro kompilátor
- Najít instalované balíčky/knihovny
- Poskytnout cesty ke všem externím knihovnám, které CCC využívá
- Poskytnout cesty k hlavičkám a zdrojovým kódům CCC a vytvořit z nich knihovnu
- Vytvořit *PyBind11* modul
- Připojit k modulu veškeré, již zkompilované, externí knihovny

S nastavením parametrů žádné složité problémy nenastaly:

```
cmake_minimum_required(VERSION 3.4)
set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_CXX_STANDARD 11)
set(PYBIND11_PYTHON_VERSION 3.6)
set(CMAKE_POSITION_INDEPENDENT_CODE ON)
```

Nastavil jsem minimální verzi *CMake*, *PyBind11* vyžaduje minimální verzi *CMake* 3.4. Připojil jsem adresář, ve kterém se *CMakeLists.txt* nachází, nastavil verzi standardu C++, verzi Pythonu a *CMAKE_POSITION_INDEPENDENT_CODE ON*, což je parametr, který je potřeba k vytvoření *.so* souborů.

V dalším kroku jsem přidal instalované balíčky a knihovny pomocí vestavěné funkce *CMake* **find_package()**. Ta v systému vyhledá požadovaný balíček a importuje ho do *CMake*:

```
find_package(Python3 REQUIRED COMPONENTS Interpreter Development)
find_package(Qt5 REQUIRED COMPONENTS Sql)
find_package(Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt5 REQUIRED COMPONENTS Xml)
find_package(Qt5 REQUIRED COMPONENTS Network)
find_package(Qt5 REQUIRED COMPONENTS Core)
find_package(Qt5 REQUIRED COMPONENTS Gui)
find_package(pybind11 REQUIRED)
```

Dalším krokem bylo přidání všech cest k hlavičkám. Tento krok je důležitý kvůli kompilaci CCC. Použije se příkaz **include_directories("cesta")**. Tímto způsobem jsem musel přidat velké množství adresářů, uvedu tedy jen jeden konkrétní příklad:

```
include_directories("/online/subsystems/daq/RCCARS/
amber-rccars-daq/fsmlib/include")
```

Nyní mám k dispozici vše potřebné ke zkompilování CCC, vytvořím tedy knihovnu *commonClientCore* s parametrem *SHARED* a poskytnu jí všechny soubory CCC:

```
add_library(commonClientCore SHARED
"src/ChatLog.cpp"
"src/Controller.cpp"
"src/Model.cpp"
"src/DaqClient.cpp"
...
"src/device/SpillbufferDevice.cpp"
"src/device/SwitchDevice.cpp"
"src/device/TcsControllerDevice.cpp"
"include/ChatLog.h"
"include/Controller.h"
"include/Model.h"
"include/DaqClient.h"
...
"include/device/SpillbufferDevice.h"
"include/device/SwitchDevice.h"
"include/device/TcsControllerDevice.h")
```

Tato část kódu zajistí, že se celé CCC zkompiluje a vytvoří *.so* soubor CCC knihovny. Zde jsem byl dlouho zaseknutý, protože jsem byl v domnění, že již zde je potřeba propojit s externími *.so* soubory, tomu tak ale nebylo. Propojit bylo potřeba až finální *PyBind11* modul.

V CCC jsou ale použity Qt makra, které vytváří *moc* soubory. *CMake* je pro tyto účely vybavený a je možné mu nastavit parametry **AUTOMOC**, **AUTORCC** a **AUTOUIC**, které se postarají o automatické vygenerování *moc* souborů. Tyto parametry jsou určené pro *Qt*:

```
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)
...
set_target_properties(commonClientCore PROPERTIES AUTOMOC TRUE)
```

Knihovna na které toto provádím, v mém případě CCC, musí mít připojenou cestu ke hlavičkám, jinak CMake nenajde Qt makra a vygeneruje pouze *moc* soubor s komickou zprávou:

```
// This file is autogenerated. Changes will be overwritten.
// No files found that require moc or the moc files are included
enum some_compilers {need_more_than_nothing}
```

Po přidání všech potřebných cest se úspěšně vygenerují všechny potřebné *moc* soubory. Jde o soubory jako *moc_Model.cpp*,*moc_DaqClient.cpp* atp. Takto jsem se tedy dostal přes problém, který mě donutil odstoupit od balíčku *Cppimport* a celkově přes náležitosti *Qt*.

Nyní můžeme přidat modul pomocí funkce **pybind11_add_module**. Funkce očekává název modulu a soubor, ve kterém se nachází. Zde jsem také dlouho zaseknutý, protože jsem důležitost kroku podcenil. Po celou dobu jsem propojoval CCC s externími knihovnami a ne samotný modul, což byla podstata. K modulu je tedy potřeba připojit adresář s externími knihovnami, adresáře s hlavičkami externích knihoven a adresáře se zdrojovými soubory externích knihoven. Poté je potřeba se všemi knihovnami modul propojit, k tomu slouží funkce **target_link_libraries**. Finální přidání modulu vypadá takto:

```
pybind11_add_module(pythonapi src/pythonapi.cpp)
target_link_directories(pythonapi PUBLIC "/online/subsystems/daq/
RCCARS/amber-rccars-lib/lib/release")
target_include_directories(pythonapi PUBLIC
"../include" "../include/port" "../include/device"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/
DIALOGCommunication/include"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/registrylib/include"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/databaselib/include"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/daq-structure/include"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/fsmlib/include")
target_sources(pythonapi PUBLIC
"/online/subsystems/daq/RCCARS/amber-rccars-daq/
DIALOGCommunication/src"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/registrylib/src"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/databaselib/src"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/daq-structure/src"
"/online/subsystems/daq/RCCARS/amber-rccars-daq/fsmlib/src")
target_link_libraries(pythonapi
PRIVATE pybind11::module
commonClientCore
fsmlib
daqstructure
DIALOGCommunication
registryLib
registry
Database
Qt5::Sql
Qt5::Xml
Qt5::Widgets
Qt5::Network
Qt5::Core
Qt5::Gui
pybind11::embed
python3)
```

A to je vše. Psaní *CMakeLists.txt* zabralo delší dobu, protože jsem neměl žádné zkušenosti. V průběhu jsem musel vyřešit mnoho problémů s propojováním, s *Qt* atp., většinu jsem se pokusil popsat, ale rozhodně nejsou pokryty všechny problémy, které nastali. Konkrétní chyby, které hlásil Python při importování špatně zkompilovaného modulu nemám již k dispozici, nicméně zpravidla šlo o problémy s propojením s externími knihovnami. Zároveň jsem si při tomto procesu odhalil dvě chyby v CCC, kde v třídě *Port* byly deklarované dvě veřejné metody, které neměli definici v *port.cpp* a vzhledem k tomu, že byly veřejné, byly součástí *PyBind11* modulu.

Nicméně modul očekává ke každé obalované metodě její definici a zde neexistovala, musel jsem tyto metody tedy z modulu vyjmout a to byl poslední krok. Na některých částech souboru jsem spolupracoval s Ing. Antonínem Květoněm.

V tomto stavu je vše připraveno ke zkompilování:

```
cd build
cmake3 ..
make
```

Celé CCC je zkompilováno, vygeneruje se CCC *.so* soubor a poté se kompiluje *PyBind11* modul a zároveň propojí se všemi poskytnutými knihovnami. Výsledek je *pythonapi*.so*[3] nacházející se v adresáři *build*, nicméně nyní již funkční. Pokud otevřeme v adresáři *build* Python skript, můžeme konečně importovat *pythonapi* bez jediné chyby:

```
>>>import pythonapi
>>>from pythonapi import DaqClient
>>>from pythonapi import Model
```

7.3 Testování

Rozhraní je připravené k testování. Potřebujeme Qt knihovny, ty naštěstí existují i v Pythonu v balíčku *PyQt*, my používáme verzi 5. Dále bylo potřeba přesunout některé další řádky související s konfigurací do konstruktoru *DaqClient*, aby nebylo potřeba je všechny přidávat do modulu. Na funkčnost tento krok neměl absolutně žádný vliv.

V takovém stavu je vše připraveno k vytvoření třídy *DaqClient* a testování. Vytvořil jsem jednoduchý skript, který ukazuje funkčnost jednoduchých metod, které poskytují informace o stavu, tyto hodnoty jsem zároveň kontroloval s daty a všechny informace jsou správné. Skript je napsán tak, aby vypisoval stav stavového automatu, číslo běhu, číslo svazku, zda je *Model* inicializován, zda je *DaqClient* inicializován a zda je *Controller* uzamknut[4].

Nejdříve je vytvořena instance *QCoreApplication*, ve které běží *Event Loop*. Dále je potřeba nahrát nastavení a inicializovat *singleton DaqClient* metodou *start()*. Proces je asynchronní a *QCoreApplication* běží na jiném vlákně, proto připojuji *Timer* k funkci, protože musí být volána až po zapnutí *QCoreApplication*. Funkce *lock* zamkne ovládání mým uživatelským jménem se zprávou „testing“. Funkce *actions* funguje vypisuje informace v cyklu, každé dvě sekundy. Skript je napsán taky, aby zobrazil jednotlivé stádia inicializace, jak je patrné ve výpisu z terminálu později v této kapitole.

[3] * - pythonapi je pouze na začátku názvu souboru, samotný soubor obsahuje ještě jakousi kombinaci znaků

[4] Zamknutí *Controlleru* je podmínka měnění stavu

Výsledný skript vypadá takto:

```
from pythonapi import QString
import pythonapi as pa
from PyQt5.QtCore import QCoreApplication, QThread
import sys
from threading import Timer

def actions():
    print("*******************************")
    print("master state: " + str(client.getModel().getMasterState()))
    print("Run Number: " + str(client.getModel()
                                .getCurrentRunNumber()))
    print("Spill number: " + str(client.getModel()
                                .getCurrentSpillNumber()))
    print("Model initialized: " + str(client.getModel()
                                    .isInitialized()))
    print("Daq initialized: " + str(client.isInitialized()))
    print("Is Control locked: " + str(client.getController()
                                    .isControlLocked()))
    timer = Timer(2, actions)
    timer.start()

def lock(client):
    client.getController()
          .lockControl(QString("fivitek"),QString("testing"))

app = QCoreApplication(sys.argv)
client = pa.DaqClient.getInstance()
cfg = pa.ConfigConsts.getInstance().readFile(QString("/online/
subsystems/daq/RCCARS/amber-rccars-daq/rccars-configuration.ini"))
client.start()
timer = Timer(3, lock, [client])
timer.start()
actions()
sys.exit(app.exec_())
```

Ve výstupu v terminálu zobrazím výpisy ze tří cyklů funkce *actions*. Jednotlivé cykly jsou pro přehlednost odděleny řadou znaku „*“. V prvním cyklu vidíme, že je inicializovaný pouze *DaqClient* a *master* je vypnutý, to z důvodu, že je to první průběh *actions* a aplikace není ještě zapnutá. První průběh nastaví *Timer* a poté se vykonává funkce stále dokola, již při zapnuté aplikaci.

Při druhém cyklu již aplikace běží a *master* je ve stavu **WAITING**, *Model* je již inicializován.

Mezi druhým a třetím cyklem proběhne uzamknutí ovládání, což se také promítne ve výpisu. Stav, ve kterém je klient po třetím cyklu je připravený pro přechod do vyšší fáze stavového automatu, v tom případě bychom se výše dostali příkazem *Controlleru startSlaves()*. Ve vyšších stavech se zároveň začnou zvedat parametry *Run Number* a *Spill Number*.

Výstup v terminálu vypadá takto:

```
"DIALOG client is listening on (IP) port (port)"
Successfully connected to DIALOG server
**********
master state: State.TURNED_OFF
Run Number: 0
Spill number: 0
Model initialized: False
Daq initialized: True
Is Control locked: False
**********
master state: State.WAITING
Run Number: 0
Spill number: 0
Model initialized: True
Daq initialized: True
Is Control locked: False
**********
master state: State.WAITING
Run Number: 0
Spill number: 0
Model initialized: True
Daq initialized: True
Is Control locked: True
```

Obrázek 7.1: Projevení zamknutí ovládání v GUI na základě mého skriptu

Závěr

Prvním cílem bylo seznámit se se systémem pro sběr dat experimentů COMPASS a AMBER. Se systémem jsem se stručně seznámil na základě prací a článků od členů experimentu.

Druhým cílem práce bylo seznámení se samotnou knihovnou CCC, pro kterou jsem programoval rozhraní. Seznamovat jsem se začal postupně pomocí článků a prací od vývojářů rozhraní, to mi dalo teoretické porozumění. Poté během programování a testování samotného rozhraní jsem se seznamoval s funkčností na úrovni kódu, tj. jednotlivé třídy a jejich metody (především ty stěžejní).

Seznámení s principem propojování jazyků byl velice důležitý krok, pro mě nové odvětví, o kterém jsem do té doby nevěděl. Získal jsem znalosti o možnostech provedení propojování jazyků, o výhodách jednotlivých technologií a o jejich funkčnosti. Při studování implementací jsem se zároveň naučil různé nové techniky a možnosti, jak kód implementovat a nasazovat.

Znalosti získané během rešerže jsem využil při implementaci modulu a kompilaci rozhraní. Zde jsem si zlepšil schopnosti v souvislosti s knihovnami a především schopnosti v použití kompilátorů. Pracoval jsem s *Mako* kódem a *CMake* pro účely kompilování, přičemž jsem se lépe naučil s *CMake*, pomocí kterého jsem úspěšně rozhraní zkompiloval. Při propojování knihoven jsem lépe porozuměl principu *Shared Library File* (soubory *.so*), se kterými jsem permanentně pracoval.

Zároveň jsem se během celé práce seznamoval s Linuxovými systémy, na kterých běží systém v CERNu. Standardně na Linuxových systémech nepracuji. Rozhraní jsem testoval v prostředí doposud nezapočatého experimentu AMBER. Rozhraní jsem dovedl do funkčního použitelného stavu, kdy ho lze v Pythonu využívat a ovládat tak CCC. Během implementace byla vytipována i další, pokročilejší funkcionalita, která v současné verzi rozhraní ještě není plně funkční.

Literatura

[1] Abstrakt bakalářské práce | SeminarkyZa1. *Seminarkyza1.cz* [online]. 2016 [cit. 2022-05-16]. Dostupné z: https://www.seminarkyza1.cz/blog-item/abstrakt-bakalarskeprace/2300.

[2] PYPL PopularitY of Programming Language index. *Pypl.github.io* [online]. 2022 [cit. 2022-05-16]. Dostupné z: https://pypl.github.io/PYPL.html

[3] API — Wikipedie. *Wikipedia* [online]. 2001 [cit. 2022-05-17]. Dostupné z: https://cs.wikipedia.org/wiki/API

[4] YANOVER, Uri. API - Wikipedia. *Wikipedia* [online]. 2001 [cit. 2022-05-17]. Dostupné z: https://en.wikipedia.org/wiki/API

[5] CENGIZ, Tuğkan. Ryanair Scraper · Apify. *Apify* [online]. [cit. 2022-05-17]. Dostupné z: https://apify.com/tugkan/ryanair-scraper#ryanair-scraper

[6] Platform Pricing & API Costs - Google Maps Platform. *Mapsplatform.google* [online]. [cit. 2022-06-07]. Dostupné z: https://mapsplatform.google.com/pricing/

[7] Evropská organizace pro jaderný výzkum — Wikipedie. *Wikipedia* [online]. 2001, 12.února 2006 [cit. 2022-06-07]. Dostupné z: https://cs.wikipedia.org/wiki/Evropsk%C3%A1_organizace_pro_jadern%C3%BD_v%C3%BDzkum

[8] CERN - Wikipedia. *Wikipedia* [online]. 2001, 3. února 2002 [cit. 2022-06-07]. Dostupné z: https://en.wikipedia.org/wiki/CERN#Accelerators_under_construction

[9] Higgsův boson — Wikipedie. *Wikipedia* [online]. 2001, 7.listopadu 2006 [cit. 2022-06-07]. Dostupné z: https://cs.wikipedia.org/wiki/Higgs%C5%AFv_boson

[10] The Super Proton Synchrotron | CERN. *Home.cern* [online]. [cit. 2022-06-07]. Dostupné z: https://home.cern/science/accelerators/super-proton-synchrotron

[11] The Proton Synchrotron Booster | CERN. *Home.cern* [online]. [cit. 2022-06-07]. Dostupné z: https://home.cern/science/accelerators/proton-synchrotron-booster

[12] New COMPASS page. *Wwwcompass.cern* [online]. 30. 3. 2022 [cit. 2022-06-07]. Dostupné z: https://wwwcompass.cern.ch/

[13] COMPASS | CERN. *Home.cern* [online]. [cit. 2022-06-07]. Dostupné z: https://home.cern/science/experiments/compass

[14] Gluon — Wikipedie. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001, 31. 3. 2007 [cit. 2022-06-07]. Dostupné z: https://cs.wikipedia.org/wiki/Gluon

[15] Glueball - Wikipedia. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001, 20.července 2004 [cit. 2022-06-07]. Dostupné z: https://en.wikipedia.org/wiki/Glueball

[16] Meet AMBER | CERN. *Home.cern* [online]. 8. března 2021 [cit. 2022-06-07]. Dostupné z: https://home.cern/news/news/physics/meet-amber

[17] Language binding - Wikipedia. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001, 26. října 2005 [cit. 2022-06-07]. Dostupné z: https://en.wikipedia.org/wiki/Language_binding

[18] Glue code - Wikipedia. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001, 4. dubna 2006 [cit. 2022-06-07]. Dostupné z: https://en.wikipedia.org/wiki/Glue_code

[19] KVĚTOŇ, Ing. Antonín. *Remote user interface for the control system of the COMPASS experiment at CERN*. Prague, 2017, 101 s. Dostupné také z: https://dspace.cvut.cz/bitstream/handle/10467/70409/F3-DP-2017-Kveton-Antonin-Thesis.pdf?sequence=1&isAllowed=y. Softwarové inženýrství. ČVUT. Vedoucí práce Ing. Tomáš Černý, Ph.D.

[20] BAI, Y., M. BODLAK, V. FROLOV, et al. *The Communication Library DIALOG for iFDAQ of the COMPASS Experiment*. CERN, 2017. Softwarové inženýrství. -.

[21] KVĚTOŇ, Antonín, Martin BODLÁK, Vladimir FROLOV, Stefan HUBER, Vladimír JARÝ, Igor KONOROV, Dominik STEFFEN a Miroslav VIRIUS. A multi-purpose user interface for the iFDAQ of the COMPASS experiment. *EPJ Web Conf.* Adelaide, 2020, **24.**(245), 7. Dostupné z: doi:https://doi.org/10.1051/epjconf/202024505034

[22] X-Forwarded-For. *MDN Web Docs* [online]. -: -, 2005, 30. května 2022 [cit. 2022-06-07]. Dostupné z: X-Forwarded-For - HTTP | MDN

[23] KVĚTOŇ, Antonín a Martin ZEMKO. Files · master · AMBER_RCCARS / common-client-core · GitLab. *Gitlab* [online]. Ukrajina, 2011, 27. září 2019 [cit. 2022-06-09]. Dostupné z: https://gitlab.cern.ch/AMBER_RCCARS/common-client-core/-/commits/master

[24] Unified Modeling Language — Wikipedie. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 14. února 2005 [cit. 2022-06-09]. Dostupné z: https://cs.wikipedia.org/wiki/Unified_Modeling_Language

[25] KVĚTOŇ, Antonín. RCCARS common client core reference (DAQ API): Introduction. In: *rccars-common-client-core.web.cern* [online]. CERN: -, - [cit. 2022-06-10]. Dostupné z: https://rccars-common-client-core.web.cern.ch/rccars-common-client-core/ccc/index.html

[26] Asynchronní vstup/výstup — Wikipedie. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 11. Května 2020 [cit. 2022-06-10]. Dostupné z: https://cs.wikipedia.org/wiki/Asynchronn%C3%AD_vstup/v%C3%BDstup

[27] Konečný automat — Wikipedie. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 18. listopadu 2004 [cit. 2022-06-10]. Dostupné z: https://cs.wikipedia.org/wiki/Kone%C4%8Dn%C3%BD_automat

[28] KVĚTOŇ, Antonín. RCCARS common client core reference (DAQ API): Introduction. In: *Rccars-common-client-core.web.cern* [online]. CERN: -, - [cit. 2022-06-10]. Dostupné z: https://rccars-common-client-core.web.cern.ch/rccars-common-client-core/ccc/index.html

[29] DAWES, Beman a David ABRAHAMS. Boost C++ Libraries. *Boost* [online]. -: -, 1998, 1998 [cit. 2022-06-10]. Dostupné z: https://www.boost.org/

[30] ABRAHAMS, David a Steffan SEEFELD. Boost.Python - 1.79.0. *Boost* [online]. -: -, 1998, 2002 [cit. 2022-06-10]. Dostupné z: https://www.boost.org/doc/libs/1_79_0/libs/python/doc/html/index.html

[31] DE GUZMAN, Joel a David ABRAHAMS. Building Hello World - 1.79.0. *Boost* [online]. -: -, 1998, 2002 [cit. 2022-06-10]. Dostupné z: https://www.boost.org/doc/libs/1_79_0/libs/python/doc/html/tutorial/tutorial/hello.html

[32] BRADSHAW, Robert a Stefan BEHNEL. Cython: C-Extensions for Python. *Cython* [online]. -: -, 2007, 2022 [cit. 2022-06-10]. Dostupné z: https://cython.org/#community

[33] WENZEL, Jakob. GitHub - pybind/pybind11: Seamless operability between C++11 and Python. *Github* [online]. San Francisco: -, 2008, 9. Července 2015 [cit. 2022-06-10]. Dostupné z: https://github.com/pybind/pybind11

[34] STAREC, Vladimir. CMake — Wikipedie. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 22. Května 2013 [cit. 2022-06-13]. Dostupné z: https://cs.wikipedia.org/wiki/CMake

[35] THOMPSON, Ben. GitHub - tbenthompson/cppimport: Import C++ files directly from Python!. *Github* [online]. San Francisco: -, 2008, 12. Května 2016 [cit. 2022-06-13]. Dostupné z: https://github.com/tbenthompson/cppimport

[36] Mako · PyPI. *Pypi* [online]. -: -, - [cit. 2022-06-13]. Dostupné z: https://pypi.org/project/Mako/

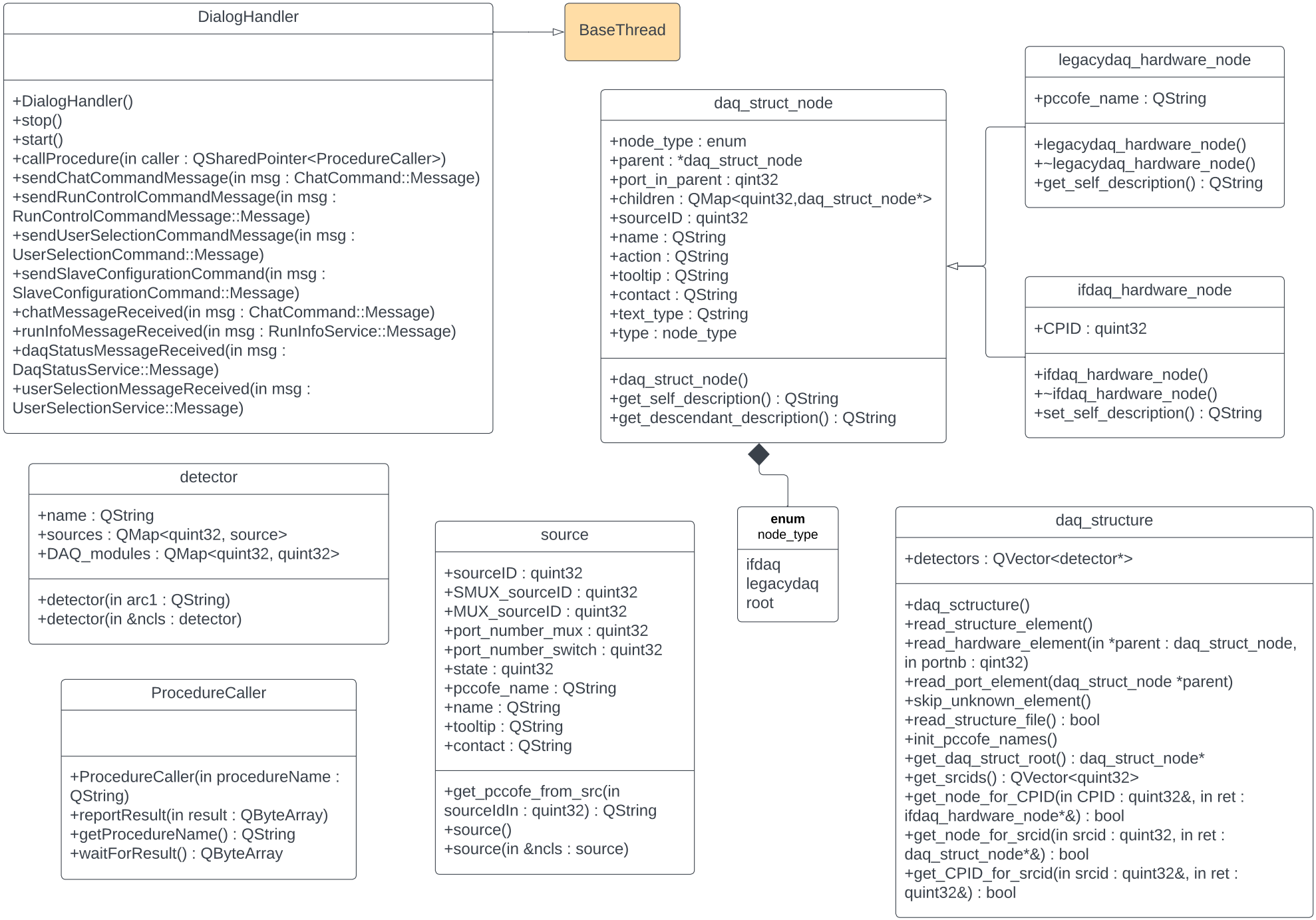
Příloha A

UML diagram veřejných metod a parametrů knihovny Common Client Core

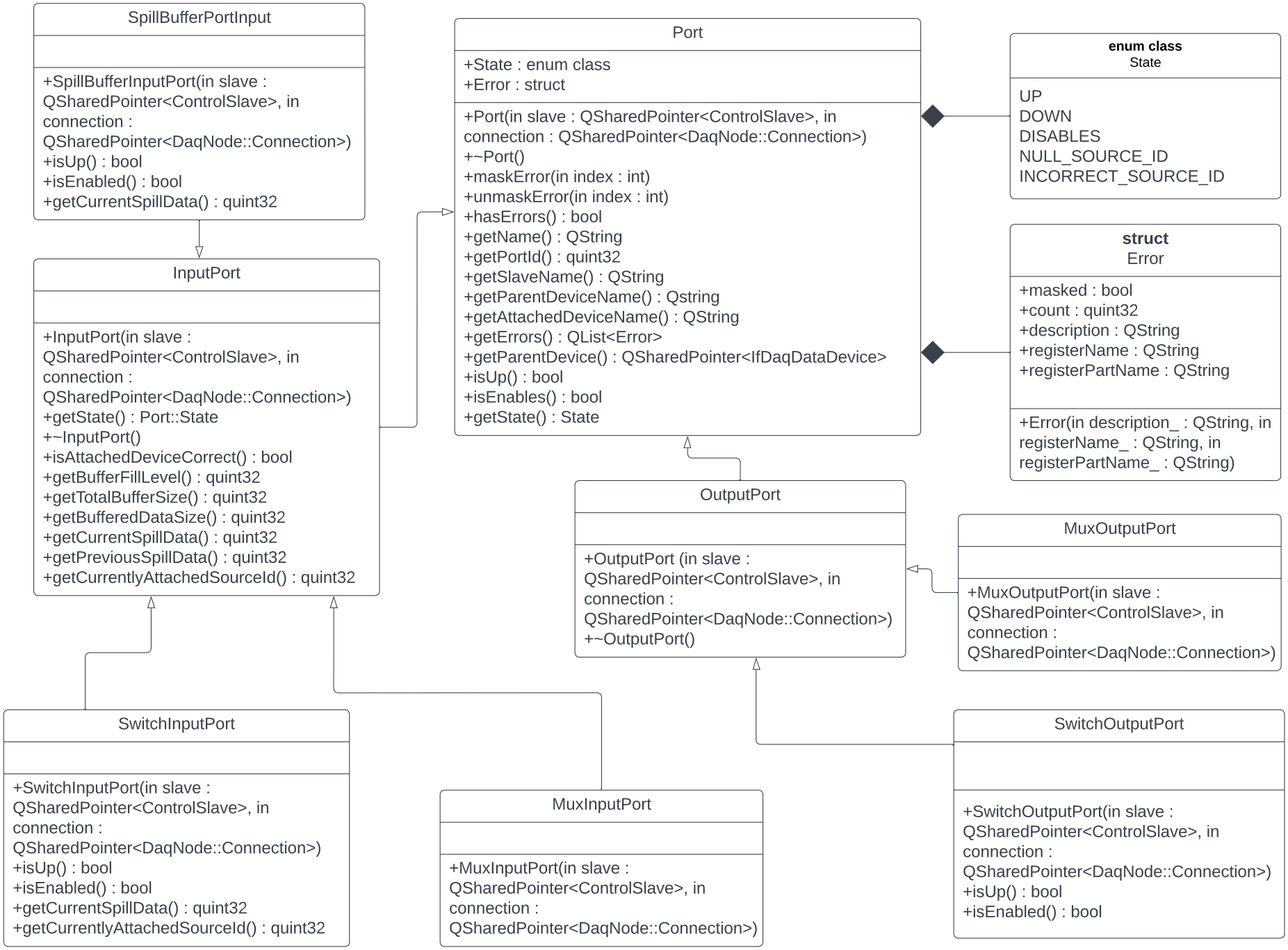
K příloze jsem vytvořil UML diagram zobrazující všechny veřejné parametry a metody CCC a tudíž přesně odráží to, co bylo dle zadání potřeba promítnout skrz API do Pythonu. Všechny třídy mají mnohem více metod, nicméně pro účely této práce by bylo kontraproduktivní je všechny vizualizovat do diagramu. Veškerá data jsou čerpána z [23].

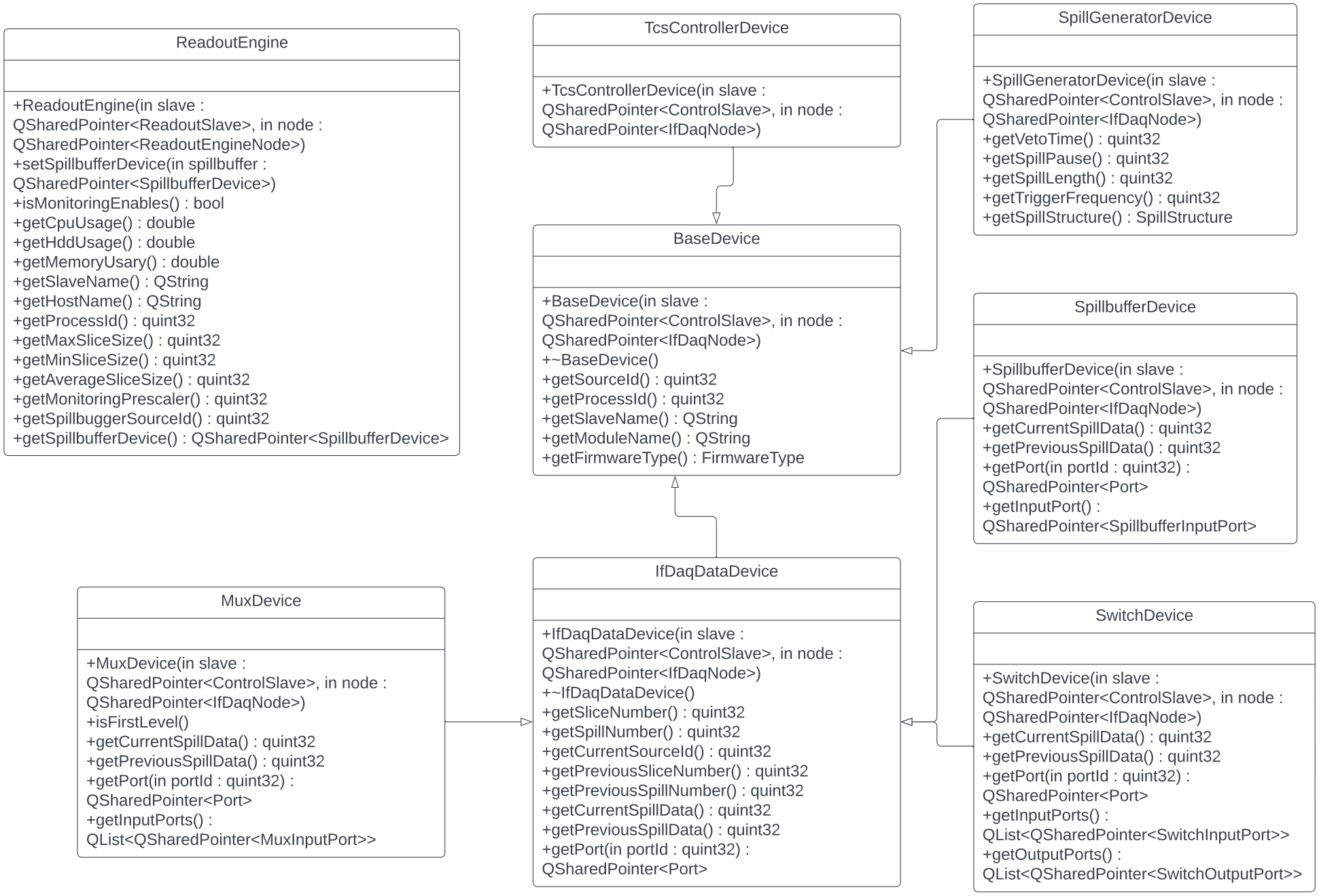
Většina promítnutých metod a parametrů slouží spíše k použití expertem. Metod, u kterých se očekává frekventované používání expertů detektoru (fyziků) je velmi málo a dali by se pravděpodobně počítat v řádech desítek.


Struct
Message
+user : QString
+text : QString
+hostName : QString
+timestamp : QDateTime
+type : ChatCommand::Messagetype
DaqClient
+getInstance() : DaqClient&
+stop()
+start()
+isInitialized() : bool
+getController() : Controller&
+getModel() : Model&
+stopDialog()
+startDialog()
Model
+Model()
+isInitialized() : bool
+getMasterState() : FSM::State
+getCurrentRunNumber() : quint32
+getCurrentSpillNumber() : quint32
+getChatLog() : ChatLog&
+SafeStopLog() : SafeStopLog&
+getRunTypes() : QList<RunType>
+getHardwareModel() : HardwareModel&
+getUserSelection() : UserSelection&
+getStructureTypes() : QList<StructureType>
+getLockMessage() : ChatCommand::Message&
+getSlaves() : QList<QSharedPointer<BaseSlave>>
+receiveRunInfoMessage(in msg :
RunInfoService::Message)
+receiveDaqStatusMessage(in msg :
DaqStatusService::Message)
+receiveUserSelectionMessage(in msg :
UserSelectionService::Message)
+callProcedure(in caller :
QSharedPointer<ProcedureCaller>)
+receiveChatMessage(in msg :
ChatCommand::Message)
+receiveSafeStopMessage(in msg :
SafeStopCommand::Message)
ChatLog
+Message
+ChatLog()
+setAutoClear(in maxMessageCount : int)
+clearMessages()
+getMessages() : QList<Message>
QObject
SafeStopLog
+SafeStopLog()
+clearErrors()
+setAutoClear(in maxErrorCount : int)
+getErrors() : QList<SafeStopCommand::StopType>
+getUnreadErrors() :
QList<SafeStopCommand::StopType>
+receiveSafeStopMessage(in msg :
SafeStopCommand::Message)
+safeStopMessageReceived(in msg :
SafeStopCommand::StopType)


DialogHandler
+DialogHandler()
+stop()
+start()
+callProcedure(in caller : QSharedPointer<ProcedureCaller>)
+sendChatCommandMessage(in msg : ChatCommand::Message)
+sendRunControlCommandMessage(in msg : RunControlCommandMessage::Message)
+sendUserSelectionCommandMessage(in msg : UserSelectionCommand::Message)
+sendSlaveConfigurationCommand(in msg : SlaveConfigurationCommand::Message)
+chatMessageReceived(in msg : ChatCommand::Message)
+runInfoMessageReceived(in msg : RunInfoService::Message)
+daqStatusMessageReceived(in msg : DaqStatusService::Message)
+userSelectionMessageReceived(in msg : UserSelectionService::Message)
BaseThread
daq_struct_node
+node_type : enum
+parent : *daq_struct_node
+port_in_parent : qint32
+children : QMap<quint32,daq_struct_node*>
+sourceID : quint32
+name : QString
+action : QString
+tooltip : QString
+contact : QString
+text_type : Qstring
+type : node_type
+daq_struct_node()
+get_self_description() : QString
+get_descendant_description() : QString
legacydaq_hardware_node
+pccofe_name : QString
+legacydaq_hardware_node()
+~legacydaq_hardware_node()
+get_self_description() : QString
ifdaq_hardware_node
+CPID : quint32
+ifdaq_hardware_node()
+~ifdaq_hardware_node()
+set_self_description() : QString
detector
+name : QString
+sources : QMap<quint32, source>
+DAQ_modules : QMap<quint32, quint32>
+detector(in arc1 : QString)
+detector(in &ncls : detector)
ProcedureCaller
+ProcedureCaller(in procedureName : QString)
+reportResult(in result : QByteArray)
+getProcedureName() : QString
+waitForResult() : QByteArray
source
+sourceID : quint32
+SMUX_sourceID : quint32
+MUX_sourceID : quint32
+port_number_mux : quint32
+port_number_switch : quint32
+state : quint32
+pccofe_name : QString
+name : QString
+tooltip : QString
+contact : QString
+get_pccofe_from_src(in sourceIdIn : quint32) : QString
+source()
+source(in &ncls : source)
enum
node_type
ifdaq
legacydaq
root
daq_structure
+detectors : QVector<detector*>
+daq_sctructure()
+read_structure_element()
+read_hardware_element(in *parent : daq_struct_node, in portnb : qint32)
+read_port_element(daq_struct_node *parent)
+skip_unknown_element()
+read_structure_file() : bool
+init_pccofe_names()
+get_daq_struct_root() : daq_struct_node*
+get_srcids() : QVector<quint32>
+get_node_for_CPID(in CPID : quint32&, in ret : ifdaq_hardware_node*&) : bool
+get_node_for_srcid(in srcid : quint32, in ret : daq_struct_node*&) : bool
+get_CPID_for_srcid(in srcid : quint32&, in ret : quint32&) : bool


QObject
HardwareModel
+HardwareModel()
+clear()
+isInitialized() : bool
+isSwitchIncluded() : bool
+isSpillGeneratorIncluded() : bool
+initialize(in slaves : QList<QSharedPointer<BaseSlave>>, in &structure : DaqStructure)
+getDeviceByName(in name : QString) : QSharedPointer<BaseDevice>
+getDeviceBySourceId(in sourceId : quint32) : QSharedPointer<BaseDevice>
+getDeviceByProccesId(in processId : quint32) : QSharedPointer<BaseDevice>
+getSwitch() : QSharedPointer<SwitchDevice>
+getReadoutEngineByName(in name : QString) : QSharedPointer<ReadoutEngine>
+getReadoutEngineByProcess(in processId : quint32) : QSharedPointer<ReadoutEngine>
+getSpillbufferBySourceId(in sourceId : quint32) : QSharedPointer<SpillbufferDevice>
+getSpillGeneratorDevice() : QSharedPointer<SpillGeneratorDevice>
+getMuxes() : QList<QSharedPointer<MuxDevice>>
+getDevices() : QList<QSharedPointer<BaseDevice>>
+getReadoutEngines() : QList<QSharedPointer<ReadoutEngine>>
+getSpillBuffers() : QList<QSharedPointer<SpillbufferDevice>>
Controller
+Controller(in &model : Model)
+startSlaves()
+stopSlaves()
+configure()
+unconfigure()
+startDryRun(in spillStructure : SpillStructure, in maxSpillNumber : quint32)
+stopDryRun()
+StartRun(in spillStructure : SpillStructure, in maxSpillNumber : quint32, in recording : bool, in continuouslyRunning : bool)
+stopRun()
+lockControl(in user : QString, in chat_message : QString)
+sendChatMessage(in user : QString, in chat_message : QString)
+selectRunType(in runTypeId : quint32)
+selectStructureType(in structureTypeId : quint32)
+toggleRecording(in recordingEnambled : bool)
+toggleContinuouslyRunning(in continuouslyRunning : bool)
+enablePort(in moduleName : QString, in portId : quint32) : bool
+togglePort(in moduleName : QString, in portId : quint32) : bool
+disablePort(in moduleName : QString, in portId : quint32) : bool
+setPrescaler(in engineName : QString, in prescaler : quint32) : bool
+isControlLocked() : bool
+toggleSpillStructure(in engineName : QString) : bool
+setSpillStructure(in settings : SpillGeneratorSettings) : bool
+sendChatCommand(in msg : ChatCommand::Message)
+sendRunControlCommand(in msg : RunControlCommand::Message)
+sendUserSelectionCommand(in msg : UserSelectionCommand::Message)
+sendSlaveConfigurationCommand(in msg : SlaveConfigurationCommand::Message)


SpillBufferPortInput
+SpillBufferInputPort(in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
+isUp() : bool
+isEnabled() : bool
+getCurrentSpillData() : quint32
Port
+State : enum class
+Error : struct
+Port(in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
+~Port()
+maskError(in index : int)
+unmaskError(in index : int)
+hasErrors() : bool
+getName() : QString
+getPortId() : quint32
+getSlaveName() : QString
+getParentDeviceName() : Qstring
+getAttachedDeviceName() : QString
+getErrors() : QList<Error>
+getParentDevice() : QSharedPointer<IfDaqDataDevice>
+isUp() : bool
+isEnables() : bool
+getState() : State
enum class
State
UP
DOWN
DISABLES
NULL_SOURCE_ID
INCORRECT_SOURCE_ID
struct
Error
+masked : bool
+count : quint32
+description : QString
+registerName : QString
+registerPartName : QString
+Error(in description_ : QString, in registerName_ : QString, in registerPartName_ : QString)
InputPort
+InputPort(in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
+getState() : Port::State
+~InputPort()
+isAttachedDeviceCorrect() : bool
+getBufferFillLevel() : quint32
+getTotalBufferSize() : quint32
+getBufferedDataSize() : quint32
+getCurrentSpillData() : quint32
+getPreviousSpillData() : quint32
+getCurrentlyAttachedSourceId() : quint32
OutputPort
+OutputPort (in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
+~OutputPort()
MuxOutputPort
+MuxOutputPort(in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
SwitchInputPort
+SwitchInputPort(in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
+isUp() : bool
+isEnabled() : bool
+getCurrentSpillData() : quint32
+getCurrentlyAttachedSourceId() : quint32
MuxInputPort
+MuxInputPort(in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
SwitchOutputPort
+SwitchOutputPort(in slave : QSharedPointer<ControlSlave>, in connection : QSharedPointer<DaqNode::Connection>)
+isUp() : bool
+isEnabled() : bool


ReadoutEngine
+ReadoutEngine(in slave : QSharedPointer<ReadoutSlave>, in node : QSharedPointer<ReadoutEngineNode>)
+setSpillbufferDevice(in spillbuffer : QSharedPointer<SpillbufferDevice>)
+isMonitoringEnables() : bool
+getCpuUsage() : double
+getHddUsage() : double
+getMemoryUsary() : double
+getSlaveName() : QString
+getHostName() : QString
+getProcessId() : quint32
+getMaxSliceSize() : quint32
+getMinSliceSize() : quint32
+getAverageSliceSize() : quint32
+getMonitoringPrescaler() : quint32
+getSpillbuggerSourceId() : quint32
+getSpillbufferDevice() : QSharedPointer<SpillbufferDevice>
TcsControllerDevice
+TcsControllerDevice(in slave : QSharedPointer<ControlSlave>, in node : QSharedPointer<IfDaqNode>)
SpillGeneratorDevice
+SpillGeneratorDevice(in slave : QSharedPointer<ControlSlave>, in node : QSharedPointer<IfDaqNode>)
+getVetoTime() : quint32
+getSpillPause() : quint32
+getSpillLength() : quint32
+getTriggerFrequency() : quint32
+getSpillStructure() : SpillStructure
BaseDevice
+BaseDevice(in slave : QSharedPointer<ControlSlave>, in node : QSharedPointer<IfDaqNode>)
+~BaseDevice()
+getSourceId() : quint32
+getProcessId() : quint32
+getSlaveName() : QString
+getModuleName() : QString
+getFirmwareType() : FirmwareType
SpillbufferDevice
+SpillbufferDevice(in slave : QSharedPointer<ControlSlave>, in node : QSharedPointer<IfDaqNode>)
+getCurrentSpillData() : quint32
+getPreviousSpillData() : quint32
+getPort(in portId : quint32) : QSharedPointer<Port>
+getInputPort() : QSharedPointer<SpillbufferInputPort>
MuxDevice
+MuxDevice(in slave : QSharedPointer<ControlSlave>, in node : QSharedPointer<IfDaqNode>)
+isFirstLevel()
+getCurrentSpillData() : quint32
+getPreviousSpillData() : quint32
+getPort(in portId : quint32) : QSharedPointer<Port>
+getInputPorts() : QList<QSharedPointer<MuxInputPort>>
IfDaqDataDevice
+IfDaqDataDevice(in slave : QSharedPointer<ControlSlave>, in node : QSharedPointer<IfDaqNode>)
+~IfDaqDataDevice()
+getSliceNumber() : quint32
+getSpillNumber() : quint32
+getCurrentSourceId() : quint32
+getPreviousSliceNumber() : quint32
+getPreviousSpillNumber() : quint32
+getCurrentSpillData() : quint32
+getPreviousSpillData() : quint32
+getPort(in portId : quint32) : QSharedPointer<Port>
SwitchDevice
+SwitchDevice(in slave : QSharedPointer<ControlSlave>, in node : QSharedPointer<IfDaqNode>)
+getCurrentSpillData() : quint32
+getPreviousSpillData() : quint32
+getPort(in portId : quint32) : QSharedPointer<Port>
+getInputPorts() : QList<QSharedPointer<SwitchInputPort>>
+getOutputPorts() : QList<QSharedPointer<SwitchOutputPort>>

Příloha B

Obsah CD

Přiložené CD obsahuje elektronickou podobu vytisknuté verze práce a zadání. Dále zdrojový kód *PyBind11* modulu *pythonapi.cpp*, *CMakeLists.txt*, kterým jsem vytvářel *.so* soubory a testovací Python kód *test.py*, jež je zároveň v práci. Navíc je ještě přidán vylepšený testovací skript, pomocí kterého lze přepnout stav *Mastera* do stavu *CONFIGURED*.