UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2017 Bc. Jan Hetych

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Návrh a implementace ladícího nástroje pro objekty replikačního serveru

Bc. Jan Hetych

Diplomová práce

2017

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2016/2017**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan Hetych**
Osobní číslo:	**I15205**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Návrh a implementace ladícího nástroje pro objekty replikačního serveru**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V teoretické části práce: a) Budou popsány principy replikace s důrazem na řešení konzistence objektů, topologie sítí, 32 a 64 bitovou architekturu. b) Bude proveden přehled vybraných replikačních řešení. V praktické části se diplomat zaměří na návrh a implementaci softwarového ladícího nástroje pro objekty replikačního portu. Toto softwarové řešení bude na základě analýzy objektů pro replikační port schopné generovat grafické rozhraní, které dále uživateli umožní s tímto objektem operovat. Pro analýzu objektů pro replikaci bude využit nástroj Doxygen, který umožňuje konvertovat zdrojové sobory do XML standardu.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

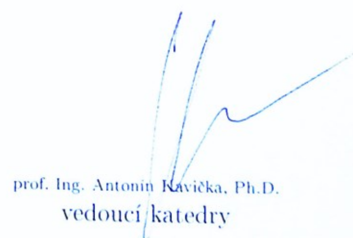
***YUZWA, Erik. Game programming in C++. Hingham, MA: Charles River Media, c2006. ISBN 1584504323.**
***RISCHPATER, Ray. Application development with Qt creator a fast-paced guide for building cross-platform applications using Qt and Qt Quick. New Edition. Birmingham, UK: Packt Pub, 2013. ISBN 1783282312.**

Vedoucí diplomové práce: **Ing. Jan Fikejz, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2016**
Termín odevzdání diplomové práce: **17. května 2017**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.



prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2016

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 16. 5. 2017

podpis autora

Bc. Jan Hetych

PODĚKOVÁNÍ

Chtěl bych poděkovat vedoucímu mé bakalářské práce Ing. Janu Fikejzovi Ph.D. za trpělivost a velmi rychlý a vstřícný přístup pro konzultace mé diplomové práce. Dále bych chtěl poděkovat Ing. Markétě Kohoutkové za poskytnutí několika bezstarostných víkendů pro vytváření textové části diplomové práce.

ANOTACE

Tato diplomová práce se zabývá návrhem a implementací ladícího nástroje, který umožní za běhu vytvořit grafické uživatelské rozhraní. Aplikace vytvoří grafické rozhraní pro konkrétní objekt replikačního serveru na základě jeho třídy. Toto grafické rozhraní umožní uživateli vybrat si konkrétní objekt z replikačního serveru a volat jeho metody včetně parametrů.

KLÍČOVÁ SLOVA

GUI, replikační server, ladící nástroj, objekt

TITLE

Design and implementation of an objects debugging tool for a replication server

ANNOTATION

This diploma thesis deals with the design and implementation of a debugging tool, which will allow users to create a graphical user interface at runtime. The application creates a graphical interface for a particular replication server object based on its class. This graphical interface allows the user to select a specific object from the replication server and call its methods including parameters.

KEYWORDS

GUI, replication server, debugging tool, object

OBSAH

Úvod....13
1 Replikace 15
1.1 Principy 15
1.1.1 Architektura 15
1.1.2 Funkce....17
1.1.3 Řešení problémů 19
1.2 Topologie 22
1.2.1 Statická a dynamická 24
1.2.2 Šíření změn 24
1.2.3 Segmenty 28
1.3 Dostupná řešení....30
1.3.1 ReplicaNet 30
1.3.2 Unreal Engine 34
2 Frameworky uživatelského rozhraní....38
2.1 wxWidgets....38
2.1.1 Historie....38
2.1.2 Vlastnosti 38
2.1.3 Vývoj a použití....39
2.2 GTK+ 40
2.2.1 Historie....40
2.2.2 Vlastnosti 41
2.3 QT 43
2.3.1 Historie....43
2.3.2 Vlastnosti 43
2.4 Výběr GUI frameworku pro diplomovou práci 45
3 Syntaktická analýza jazyka C++....47

3.1 CastXML 47

3.2 Clang 47

3.3 Doxygen 48

3.4 Vlastní analýza 50

3.5 Výběr C++ parseru pro diplomovou práci 50

4 Návrh a implementace aplikace RPL Debugger 51

4.1 Koncepce aplikace 51

4.1.1 Získání objektů z replikačního serveru 52

4.1.2 Vytvoření pluginu pro podporu GUI u metod nového replikovaného objektu...55

4.2 Rozhraní pro práci s replikačním serverem 62

4.3 Struktura aplikace 64

4.3.1 Konfigurační soubor 66

4.3.2 Zdrojové kódy 67

4.3.3 Formuláře 67

4.3.4 Prostředky – zdrojová data aplikace 69

4.4 Vzhled aplikace 69

4.5 Funkce aplikace 71

4.5.1 Filtrování zobrazených objektů 71

4.5.2 Zobrazování logu při kompilaci pluginů 72

4.5.3 Zobrazení více oken 73

4.6 Testování aplikace v praxi 74

ZÁVĚR 75

Použitá literatura 77

Přílohy 79

SEZNAM ILUSTRACÍ A TABULEK

Obrázek 1 - Klientské aplikace a replikační server na dvou počítačích 16
Obrázek 2 - Způsob uložení čísla 0x4A3B2C1D na adresu 100 pomocí little-endian.............21
Obrázek 3 - Způsob uložení čísla 0x4A3B2C1D na adresu 100 pomocí big-endian 21
Obrázek 4 - Možná topologie v rámci jednoho počítače 22
Obrázek 5 - Každý klient/aplikace a jeho UUID...23
Obrázek 6 - Plug & play spojení replikačních serverů 24
Obrázek 7 – Zakázaný cyklus v topologii replikačního řešení firmy RETIA, a.s. 25
Obrázek 8 - Princip šíření objektu a konstrukce v jednotlivých uzlech replikační sítě...........26
Obrázek 9 - Příklad odpojení části sítě..27
Obrázek 10 - Segmenty v topologii replikační sítě...29
Obrázek 11 - Vývojové prostředí Unreal Engine 4 34
Obrázek 12 - Autoritativní klient/server architektura...35
Obrázek 13 - Uživatelské rozhraní TrueCrpyt vytvořené ve wxWidgets..................................39
Obrázek 14 - IDE wxFormBuilder 40
Obrázek 15 - Graf počtu řádků zdrojového kódu knihovny GTK+ napříč lety........................41
Obrázek 16 - Grafický rastrový editor GIMP využívající knihovnu GTK+.............................42
Obrázek 17 - IDE Glade pro GTK...42
Obrázek 18 - QT Creator 44
Obrázek 19 - VLC Media Player využívající knihovnu QT pro grafické rozhraní 45
Obrázek 20 - Předklad zdrojového kódu, front-end a back-end překladačem...........................47
Obrázek 21 - Koncepce přidání podpory pro nový objekt replikačního serveru......................55
Obrázek 22 - Diagram tříd pro uložení hierarchie třídy na základě XML souboru..................59
Obrázek 23 - Diagram tříd pro hlavní okno aplikace MainWindow 65
Obrázek 24 - Struktura projektu celé aplikace...66
Obrázek 25 - Formulář hlavního okna z prostředí QT Creator..68
Obrázek 26 - Ukázka hlavního okna aplikace RplDebugger...70
Obrázek 27 - Ukázky filtrování seznamu replikovaných objektů 72
Obrázek 28 - Průběh kompilace pluginů na základě hlavičkových souborů 73
Obrázek 29 - Podpora více oken pro editací vícero objektů zároveň 74

Tabulka 1 - Rozsah zdrojového kódu GTK+ v počátcích vývoje...41

SEZNAM ZKRATEK A ZNAČEK

GUI	Graphical User Interface (grafické uživatelské rozhraní)
UUID	Universally Unique IDentifier (univerzální unikátní identifikátor)
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
ROL	Replica Object Language
RPC	Remote Procedure Call (vzdáleného volání procedur)
IDE	Integrated Development Environment (integrované vývojové prostředí)
AST	Abstract Syntax Tree (abstraktní syntaktický strom)

ÚVOD

Informační technologie v posledních letech pronikly téměř do všech oblastí lidské činnosti. Mnoho firem a podniků by dnes neexistovalo nebo neexistovalo na takové úrovni nebýt rozmachu informačních technologií. Společnosti, pro které je stěžejní vývoj vlastního softwaru, vynakládají vysoké prostředky na kvalifikovanou pracovní sílu, které je stále nedostatek. Vzniká tedy potřeba, jak vývoj co nejvíce zefektivnit a kvalifikované pracovní síle vývoj ulehčit. V některých případech podniku pomůže pouze lepší organizace při vývoji software, jako je vypracování vývojového plánu nebo rozdělení práce na více fází či rozdělení práce mezi více lidí. V některých případech je nutné vyvinout speciální software, který není výsledným produktem určeným pro zákazníka, ale pouze pomáhá při vývoji výsledného produktu.

V určitých softwarových řešeních je nutné v reálném čase rozdistribuovat stejné informace napříč sítí do více počítačů. Obecně tento druh distribuce můžeme nazvat replikací, kde pro skupinu klientů šíříme replikovaný objekt tak, aby byl stejný na kterémkoliv z nich. Velice známým a populárním případem jsou počítačové hry, konkrétně hry s podporou více hráčů. V této oblasti je například nutné napříč klienty (hráče) rozdistribuovat informace o herním světě. Z programátorského hlediska jsou to konkrétní entity, jako může v tomto případě být objekt herního světa, který může obsahovat nespočet atributů (např. aktuální stav zničitelných objektů v herní mapě – zničeno/nezničeno), které ovlivňují výslednou grafickou reprezentaci, jež uživatel vidí na monitoru.

Tento druh distribuce objektů napříč klienty není výsadou pouze zábavního průmyslu, ale také jiných oblastí jako je například vojenství nebo letectví, kde je nutné, abychom pro skupinu uživatelů dokázali zobrazit stejná data u každého z nich tak, aby s těmito daty mohlo pracovat více lidí zároveň, např. skupina letových dispečerů nebo armádní jednotka obsluhující radar a palebné zbraně. Každý z těchto uživatelů může dokonce na svém počítači používat jiný software, který pouze využívá stejná rozdistribuovaná data a tímto uživatele specializovat na konkrétní úkol v dané skupině.

Při vývoji tohoto druhu softwaru je nutné otestovat chování replikovaných objektů na změny a pozorovat tak, zda se změna rozdistribuuje po celé síti nebo zda změna měla kýžený efekt na objekt. Změna objektu je obvykle vyvolána zavoláním některé z dostupných veřejných metod objektu. Aby bylo možné otestovat všechny metody objektu, je nutné nejdříve tento objekt použít v nějaké aplikaci a zavolat nad ním všechny tyto metody. Tento druh testování má však svá úskalí. Prvním z nich je pevně definované pořadí volání metod. Pro změnu pořadí by bylo

nutné v kódu aplikace tato volání zaměnit a aplikaci znovu sestavit. Úplně stejný problém by nastal, pokud by měla být otestována metoda s jinou hodnotou parametru. Elegantnějším a pohodlnějším řešením by byla aplikace s grafickým uživatelským rozhraním (GUI), která by umožňovala zadávání parametrů a volání metod pomocí grafických prvků tohoto rozhraní. Ani toto řešení není bez úskalí. Největší problém spočívá v nutnosti vytvářet GUI aplikaci šitou na míru danému objektu. Pro jiný objekt, který má jiné metody s jiným počtem parametrů je nutné vytvořit GUI aplikaci novou. Pokud je objekt stále ve vývoji, je potřeba změny pokaždé promítnout i do GUI aplikace.

Ve firmě RETIA, a.s. vznikla potřeba vyvinout speciální aplikaci, která by dokázala odstranit i tato úskalí. Vnikla potřeba na aplikaci, která by umožňovala vytvořit GUI rozhraní univerzálně pro jakýkoliv replikovaný objekt a otestovat ho v rámci svého replikačního řešení.

Cílem této diplomová práce bude popsat problematiku replikace a následně zanalyzovat dostupná řešení pro vývoj GUI aplikací. Na závěr teoretické části se bude diplomová práce zabývat C++ parsery, jež umožní analyzovat třídu včetně atributů a metod. Na teoretickou část naváže praktická část, ve které budou popsány přístupy a řešení návrhu aplikace.

1 REPLIKACE

Replikace je proces kopírování objektů napříč celým systémem, přičemž hlavním cílem je držet objekt ve všech uzlech systému v konzistentním stavu. Tato diplomová práce se bude zabývat replikačním serverem vytvořeným v programovacím jazyce C++ Ing. Miroslavem Kubíkem mladším ve firmě RETIA, a.s. v Pardubicích. Ladící nástroj bude s tímto serverem úzce spolupracovat a bude mít za úkol nad jeho objekty vytvořit GUI rozhraní, které umožní volání všech dostupných metod objektu. V závěru této kapitoly budou popsána veřejně dostupná alternativní řešení a především jejich odlišnosti v porovnání s řešením použitým ve firmě RETIA, a.s.

1.1 Principy

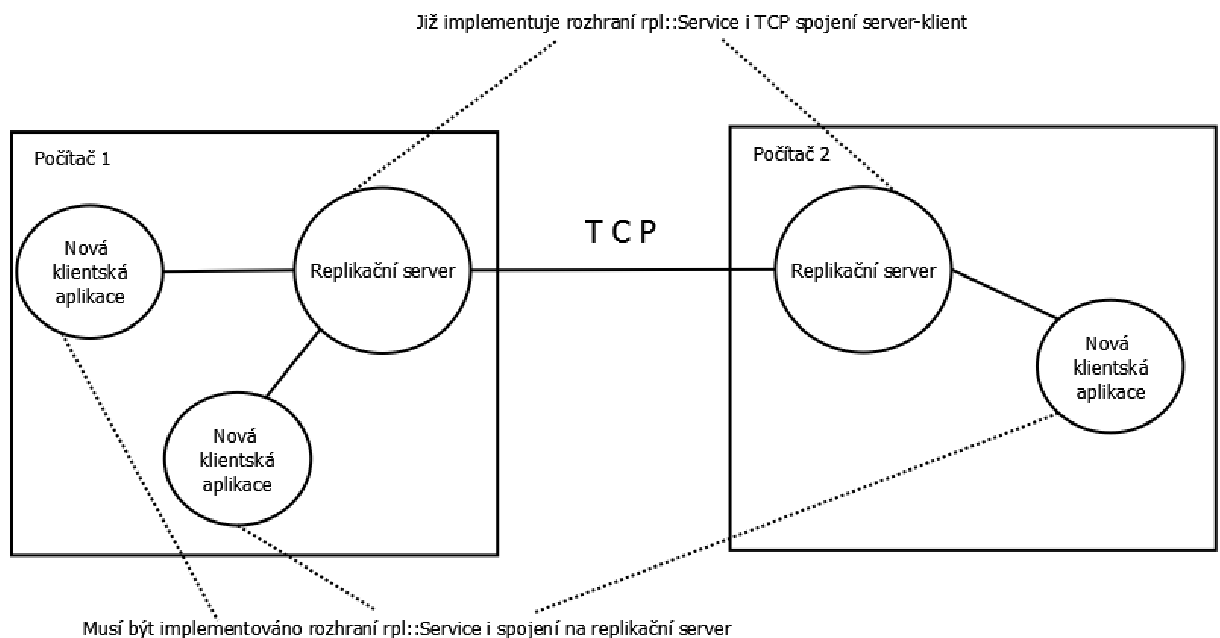
Pro vývoj ladícího nástroje je téměř nutností znát princip jakým se replikované objekty šíří, jak replikační server funguje a jak tyto objekty získat. V této diplomové práci budou popsány obecné principy fungování replikačního serveru firmy RETIA, a.s. Konkrétní zdrojové kódy replikačního serveru jsou chráněným know-how firmy RETIA, a.s., a proto zde nebudou zveřejněny. V kapitole 4 však bude se svolením firmy RETIA, a.s. zveřejněno rozhraní, se kterým ladící nástroj pracuje a ukázka replikovaného objektu.

1.1.1 Architektura

Každý uzel replikační sítě musí implementovat replikační službu - rozhraní `rpl::Service`, které poskytuje službu samotné replikace. Jedná se o bázovou třídu, poskytující několik virtuálních i čistě virtuálních metod. Tato služba se stará o veškeré dění replikace, ale hlavně o sdílení a přijímání objektů (typu `rpl::Object`) od jiných uzlů. Cílem této služby však není zprávy či data dopravit do dalšího uzlu, ale služba pouze definuje protokol, v jakém formátu se mají odeslat do dalšího uzlu a jakým způsobem se mají přijatá data zpracovat. Konkrétní rozhraní této bázové třídy `rpl::Service` bude popsáno v kapitole 4. Toto rozhraní tedy neimplementuje konkrétní způsob komunikace mezi uzly sítě a nechává tím tedy volnou ruku programátorovi, jak tuto komunikaci zajistí – např. TCP knihovnou třetích stran.

Aby součástí nové aplikace využívající rozhraní `rpl::Service` nemusela být implementace komunikace mezi počítači (typicky server-klient), firma RETIA, a.s. má již hotovou aplikaci replikačního serveru (`rpl::Server`), která plní roli prostředníka mezi klientskou aplikací a jiným počítačem. `Rpl::Service` tedy implementuje protokol a chování samotné replikace,

ale `rpl::Server` je již konkrétní implementace tohoto rozhraní `rpl::Service` včetně konkrétního řešení způsobu komunikace pomocí TCP spojení klient-server mezi počítači (Obrázek 1).



Obrázek 1 - Klientské aplikace a replikační server na dvou počítačích

zdroj: vlastní

Pokud je požadavkem například replikovat uživatelův objekt na dvou počítačích, tak se v praxi nejdříve ustaví spojení mezi dvěma počítači pomocí replikačního serveru spuštěném na každém z počítačů. Typicky stačí replikační servery nakonfigurovat pomocí konfiguračního souboru obsahujícího mimo jiné i IP adresy. Jakmile spojení mezi počítači funguje, uživatel vytvoří novou klientskou aplikaci, která implementuje rozhraní `rpl::Service` a která se připojí na replikační server například pomocí protokolu TCP. Objekt z klientské aplikace stačí nasdílet pomocí metody replikační služby a na druhém počítači dá replikační služba klientské aplikaci vědět (pomocí mechanismu handlerů – obsluhy událostí), že se byl přidán nový objekt a klientská aplikace se podle toho může zachovat. Jak tento proces šíření objektů nebo jejich změn funguje, bude popsáno v oddílu 1.2.

Mezi uzly replikační sítě jsou šířeny objekty pouze daného formátu. Tento formát je definovaný ve třídě `rpl::Objekt`, který musí být pro jakýkoliv šířený objekt veřejně děděný z této třídy.

```
RETIA, a.s. - librpl: rpl_testobject.h

namespace rpl
{

class TestObject : public Object
{
    ................
    ................
}
```

1.1.2 Funkce

Replikační řešení firmy RETIA, a.s. poskytuje mnoho funkcí a mezi ty stěžejní patří serializace a deserializace objektů. Každý objekt uchovává stav v podobě hodnot svých atributů a právě tyto informace je nutné dopravit do všech uzlů sítě. Přenos těchto atributů probíhá binární formou. Obecně řečeno jsou tyto hodnoty atributů převedeny do binární podoby a poté zřetězeny za sebe a tím vznikne proud jedniček a nul. Deserializací jsou tyto atributy z binární podoby v opačném pořadí přečteny a nastaveny do aktuálního objektu.

Další významnou funkcí je funkce `print`, která je v podstatě obdobou `toString` v konvencích programovacího jazyka Java. Metoda `print` musí dodržovat jednotný formát napříč všemi objekty.

RETIA, a.s. - librpl: rpl_testobject.h

```
    .................
    .................

private:

    bool boolean_;
    int integer_; // serialized as int32_t
    std::string string_;
    Pointer<const TestObject> pointer_;
    std::vector<uint8_t> stack_;

    // not replicated
    int local_;
    std::string localStr_;

    .................
    .................

    virtual void serialize(OutStream &stream) const
    {
        Object::serialize(stream);
        stream << boolean_ << (int32_t)integer_;
        stream << string_ << pointer_   << stack_;
    }

    virtual void deserialize(InStream &stream)
    {
        Object::deserialize(stream);
        stream >> boolean_;
        stream.read<int32_t>(integer_);
        stream >> string_ >> pointer_ >> stack_;
    }

    virtual void print(std::ostream &stream) const
    {
        Object::print(stream);
        stream << " Boolean(" << (boolean_ ? "true" : "false") << ")

    .................
    .................
    }
    .................
    .................
```

Každý replikovaný objekt je definován pouze hlavičkovým souborem, kde je kompletně popsána celá třída. Aby však bylo možné v jiném uzlu sítě nový objekt zkonstruovat, třída nestačí. Je potřeba z nové třídy `rpl:Objektu` vytvořil plugin - knihovnu `.so`, kterou lze stejně jako v operačním systému Windows (soubory typu `.dll`) načítat do aplikace dynamicky až po spuštění aplikace. Tento plugin lze pak přenášet do jiných uzlů sítě a zajistit tak podporu daného objektu napříč replikační sítí.

Replikační služba pro komunikaci s tímto pluginem potřebuje rozhraní. Je tedy nutné definovat metody, které budou stejné napříč všemi pluginy. Tyto metody v podstatě vytvoří metody pro zjišťování informací nebo pro vytvoření instance objektu u aktuálně načteného pluginu replikační službou. Aby nemusel uživatel, který vytváří svůj nový objekt, tyto metody vytvářet sám a dbát na jejich konzistenci napříč jinými `rpl::Objekty`, stačí použít makro `RPL_PUBLIC` v části public třídy nového objektu.

RETIA, a.s. - librpl: rpl_testobject.h

```
    .................
    .................

public:
    RPL_PUBLIC(TestObject)
    .................
    .................
```

Aby bylo možné výslednou třídu replikovaného objektu zkompilovat jako plugin `.so`, je nutné vytvořit nový .cpp soubor s makrem, které automaticky definuje rozhraní toho pluginu na základě jeho názvu a to opět pomocí makra.

RETIA, a.s. - librpl: rpl_testobject.cpp

```
#include "rpl/rpl_testobject.h"
#include "rpl/rpl_plugin.h"

RPL_PLUGIN(rpl::TestObject)
```

1.1.3 Řešení problémů

Replikační řešení formy RETIA, a.s. musí být nezávislé na platformě. Je proto nutné zajistit kompatibilitu, aby se replikační služba chovala konzistentně. Hlavní problémy způsobí především použití různých platforem.

Velikost všech základních datových typů není striktně určena standardem jazyka C/C++. Pokud tedy bude aplikace zkompilována překladačem pro 32 bitový operační systém nebo pro 64 bitový systém, tak se velikost proměnné v paměti může lišit (Bílek, 2003).

Velikost jednotlivých datových typů se liší především na použitém operačním systému a na tom, zda systém umí adresovat 16, 32 nebo 64 bitové adresy. Aktuálně se lze nejčastěji setkat se 4 skupinami, mezi kterými se velikost jednotlivých datových typů liší:

1. LP32 (int má 16bitů, long a ukazatel má 32 bitů)
   - Win16 API – např. Windows 3.1
2. ILP32 (int, long a ukazatel mají 32 bitů)
   - Win32 API – např. Windows 9x/XP
   - Unix – např. Linux, Max OS X
3. LLP64 (int a long jsou 32 bitové a ukazatel má 64 bitů)
   - Win64 API – např. Windows x64
4. LP64 (int má 32 bitů, long a ukazatel má 64 bitů)
   - Unix – např Linux x64, Mac OS X x64

Kompletní přehled velikostí datových typů na různých platformách v příloze A (cppreference, 2017).

Problém tohoto typu se v replikačním řešení firmy RETIA, a.s. týká hlavně serializace a deserializace atributů objektu. Pokud bude například serializován atribut objektu datového typu long, tak velikost proměnné bude po serializaci na systému Linux x32 přesně 32 bitů. Jakmile by takto serializovaný atribut přišel do replikační služby na systému Linux x64, byl by tento atribut přečten jako 64 bitový, takže spolu s dalšími serializovanými atributy, které po této proměnné long následují, by bylo deserializování posunuto o 32 bitů doprava a hodnoty dalších atributů by byly později načteny chybně.

Replikační řešení firmy RETIA, a.s. tento problém v serializaci a deserializaci řeší pomocí proměnných int s fixní délkou, která byla uvedena v C++ standardu C99. Celé číslo lze tak například natvrdo serializovat jako 32 bitové pomocí datového typu int32_t.

RETIA, a.s. - librpl: rpl_testobject.h

```
virtual void serialize(OutStream &stream) const
{
    Object::serialize(stream);
    stream << boolean_ << (int32_t)integer_;
    stream << string_ << pointer_ << stack_;
}
```

Dalším významným problémem je endianita. Endianita určuje způsob uložení čísel v operační paměti. Definuje pořadí bajtů, v jakém jsou čísla ukládána. Nejpoužívanějším způsobem

endianity je little-endian (např. architektura x86), u kterého je 16 hexadecimální číslo uloženo tak, že každá číslice (bajt) je v paměti uložena v opačném pořadí, než se zapisuje (Obrázek 2).

	100	101	102	103	
...	1D	2C	3B	4A	...

Obrázek 2 - Způsob uložení čísla 0x4A3B2C1D na adresu 100 pomocí little-endian

zdroj: http://referaty-seminarky.cz/endianita/

Naproti tomu méně používaný big-endian ukládá tyto bajty v pořadí, v jakém jsou zapsány (Obrázek 3).

	100	101	102	103	
...	4A	3B	2C	1D	...

Obrázek 3 - Způsob uložení čísla 0x4A3B2C1D na adresu 100 pomocí big-endian

zdroj: http://referaty-seminarky.cz/endianita/

Endianita je jeden ze základních zdrojů nekompatibility při výměně dat v digitální podobě. Týká se přenášení binárních souborů nebo při síťové komunikaci mezi různými platformami (anonymous, 2008). Proto je nutné tuto nekompatibilitu ošetřit i v replikačním řešení firmy RETIA, a.s. Problém endianity je zde ustálen na verzi little-endian pomocé tříd `rpl::InStream` a `rpl::OutStream`. Při serializaci a deserializaci je tak nutné ošetřit, zda replikační služba běží na platformě little-endian nebo big-endian a v případě, že je to právě druhá možnost, pak je nutné otočit pořadí bajtů.

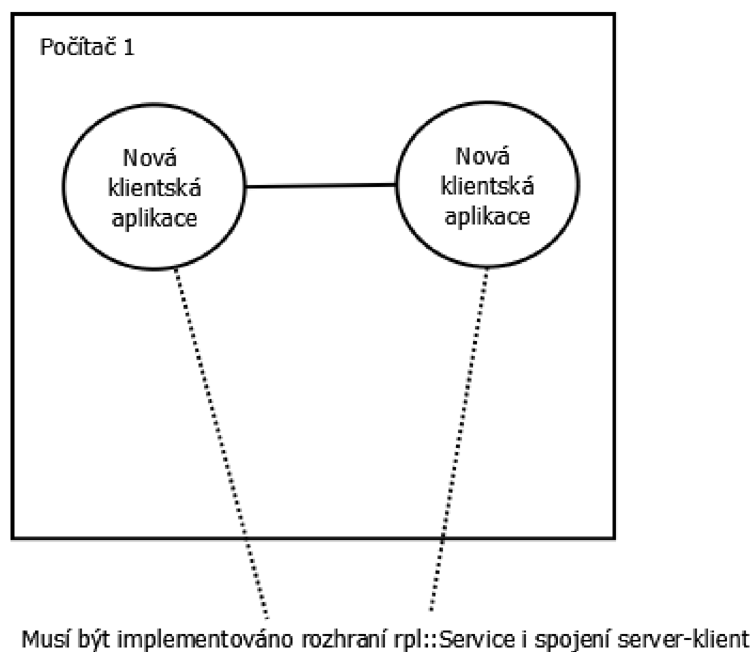
Posledním výrazným problémem v replikačním řešení firmy RETIA, a.s. je verzování replikovaných objektů. Postupem času může nastat situace, že bude nutné existující `rpl::Objekt` upravit. Aby nedocházelo k nekonzistenci a k chybám v důsledku různých verzí tohoto objektu napříč replikační sítí, je potřeba objekty verzovat. Způsob verzování je velmi jednoduchý, stačí použít makro `RPL_VERSION` s celým číslem. Pokud bude nějaký replikovaný objekt pozměněn, například přibude v něm atribut, který bude replikován, tak zvýšením čísla verze v tomto makru docílíme, že pracovat s tímto objektem může pouze replikační služba se stejnou verzí pluginu.

RETIA, a.s. - librpl: rpl_testobject.h

```
.................
.................
public:

 RPL_VERSION(3)

.................
.................
```

1.2 Topologie

V oddíle architektura byly zmíněny ty nejdůležitější stavební kameny pro fungování celého replikačního řešení firmy RETIA, a.s. Velmi významným rysem architektury je způsob, jakým je celá replikační síť vybudována. Replikační služba – `rpl::Service` neimplementuje žádné rozhraní pro komunikaci mezi uzly sítě a je zcela na programátorovi, jak tuto komunikaci zajistí (například pomocí fronty zpráv, roury, soketů, atd.). Řešení firmy RETIA, a.s. tedy umožňuje vybudovat replikační síť i v rámci jednoho počítače (Obrázek 4).

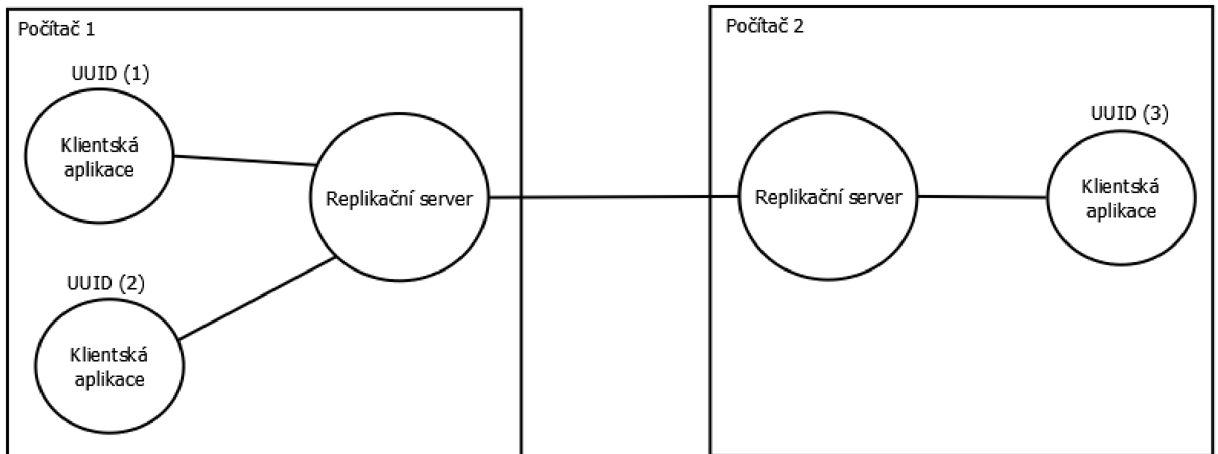


Obrázek 4 - Možná topologie v rámci jednoho počítače

zdroj: vlastní

Další velmi důležitou vlastností replikační sítě je zajištění oprávnění ke změnám replikovaného objektu. Každý šířený objekt má uveden svého vlastníka. Vlastníkem se stává ten, kdo objekt poprvé nasdílí do replikační služby. Replikační služba tomuto objektu přiřadí UUID

(universally unique identifier – univerzální unikátní identifikátor), který implementačně skrývá unikátně vygenerovaný řetězec z dostupných informací jako je číslo procesu, MAC adresa síťového rozhraní atd. V rámci celé sítě lze tedy jednoznačně určit konkrétního vlastníka daného objektu (Obrázek 5).



Obrázek 5 - Každý klient/aplikace a jeho UUID

zdroj: vlastní

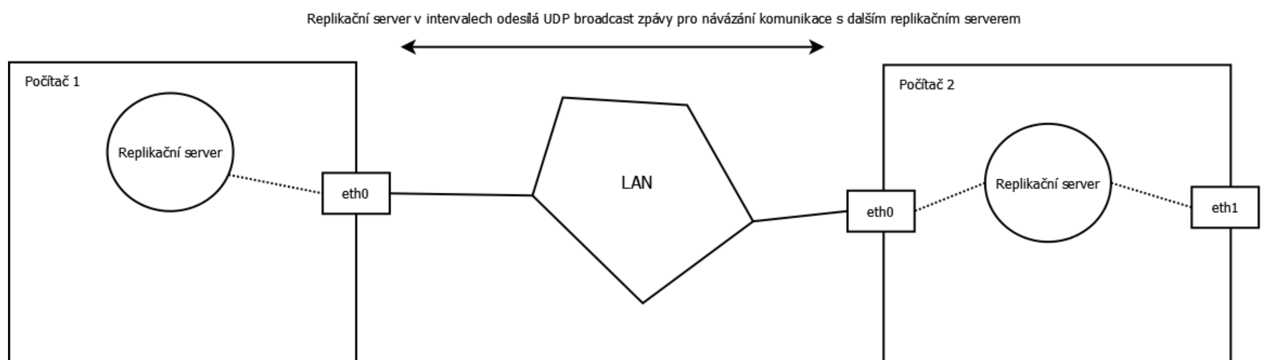
Tento unikátní identifikátor má největší význam při kontrole oprávnění na změnu tohoto objektu. U některých typů objektů je vyžadováno, aby změnu mohl provést pouze vlastník (klient/aplikace) objektu a tímto unikátním identifikátorem lze zkontrolovat, zda má daná klientská aplikace oprávnění na jeho změnu.

RETIA, a.s. - librpl: rpl_testobject.h

```
.................
.................

private:

      std::vector<uint8_t> stack_;
.................
.................
public:
      void pushByte(uint8_t byte)
      {
        if (owned()) // Kontrola vlastnictví objektu přes UUID

              {
                  stack_.push_back(byte);
                  .................
              }
                  .................
       }
```

1.2.1 Statická a dynamická

Velmi důležitou aplikací zajišťující komunikaci klient-server mezi počítači implementující rozhraní `rpl::Service` je aplikace `rpl::Server`. Tato aplikace může spojovat segmenty replikační sítě a to dvěma způsoby. První způsob je statický a byl již naznačen (Obrázek 1) a to pomocí přímého spojení. V praxi to znamená, že replikační servery propojíme na základě pevně zadaných IP adres v konfiguračním souboru replikačního serveru. Druhý způsob je dynamický a funguje na principu plug & play a to znamená, že kdykoliv můžeme na síťové rozhraní připojit nový uzel s replikačním serverem a spojení bude automaticky navázáno bez předem známé znalosti IP adres (Obrázek 6). Každý replikační server totiž v pravidelných intervalech odesílá UDP (User Datagram Protocol) broadcast zprávy, kterým replikační server na druhé straně rozumí a je tedy možno ustavit spojení po výměně „představovacích" zpráv. Konfigurační soubor tedy na rozdíl od statického případu neobsahuje konkrétní IP adresy, ale názvy síťových rozhraní, které se mají prohledávat (ve firmě RETIA, a.s. se tato funkce nazývá discovery).



Obrázek 6 - Plug & play spojení replikačních serverů

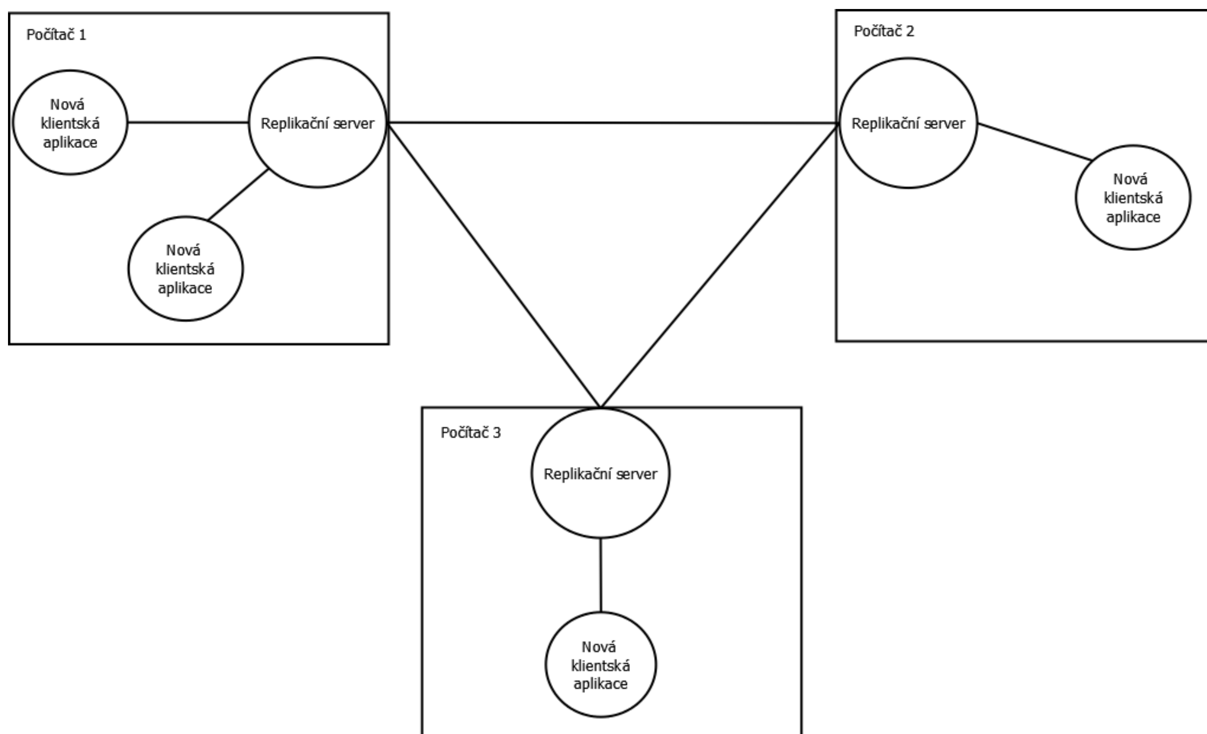
zdroj: vlastní

1.2.2 Šíření změn

V předchozích oddílech a pododdílech byly zmíněny ty nejdůležitější prvky replikačního řešení firmy RETIA, a.s. avšak hlavní účel spočívá v samotném principu fungování šíření informací v replikační síti. Každý uzel replikační sítě využívá replikační službu `rpl::Service`, která zhmotňuje implementaci fungování replikace jako takové.

Šíření jakýchkoli změn je provedeno na principu záplavy. Každý uzel replikační sítě přečte a přepošle změnu dál a tímto se změna rozšíří do všech uzlů sítě. Změnu uzel nepřeposílá zpět odkud přišla, ale pouze do dalších uzlů tam odkud změna nepřišla. Pokud replikační síť

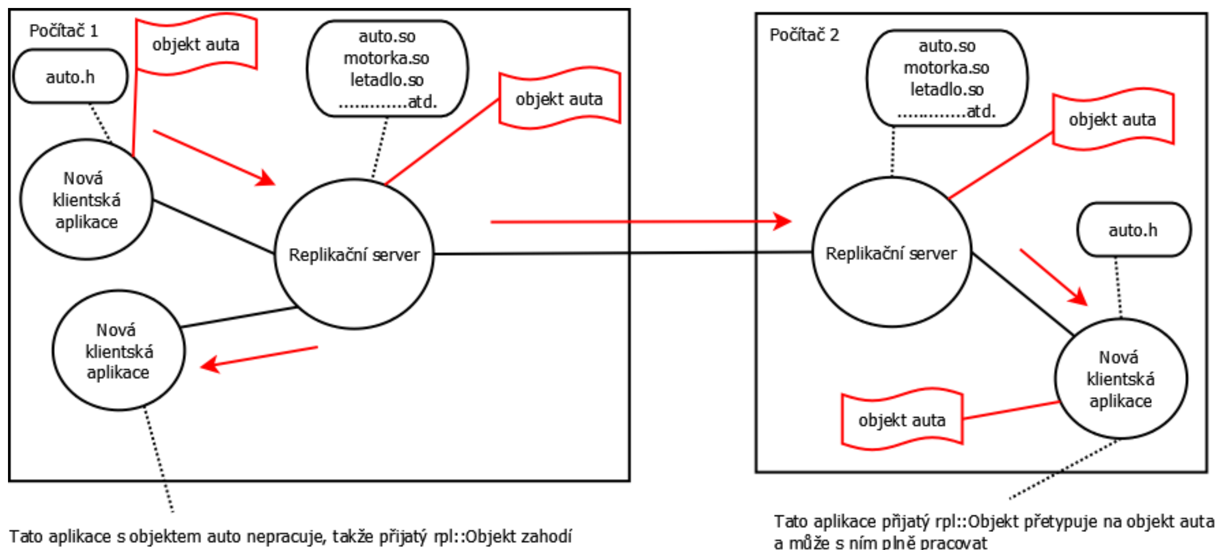
obsahuje cyklus, šíření změn by pokračovalo donekonečna, a proto je tento typ topologie v replikačním řešení firmy RETIA, a.s. zakázán (Obrázek 7).



Obrázek 7 – Zakázaný cyklus v topologii replikačního řešení firmy RETIA, a.s.

zdroj: vlastní

Prvním případem šíření změny v replikační síti je nasdílení nového objektu. Pakliže jeden uzel sítě – např. klientská aplikace nasdílí nový objekt pomocí rozhraní replikační služby, tak se tento objekt záplavově rozšíří do všech uzlů sítě (Obrázek 8). Aby se tento objekt mohl rozšířit do všech uzlů této sítě, musí být splněno několik podmínek. Replikovaný objekt musí být podporován všemi uzly sítě. To znamená, že plugin replikovaného objektu musí být přítomen ve všech replikačních serverech sítě. Replikovaný objekt je šířen jako binární zpráva serializovaných atributů objektu a v každém uzlu sítě je tato zpráva deserializována a z atributů je opět vytvořen objekt pomocí konstruktoru v příslušného pluginu. Tento princip pluginů umožňuje dynamicky přidávat další podporované objekty, aniž by bylo nutné, překompilovat samotný replikační server.



Obrázek 8 - Princip šíření objektu a konstrukce v jednotlivých uzlech replikační sítě

zdroj: vlastní

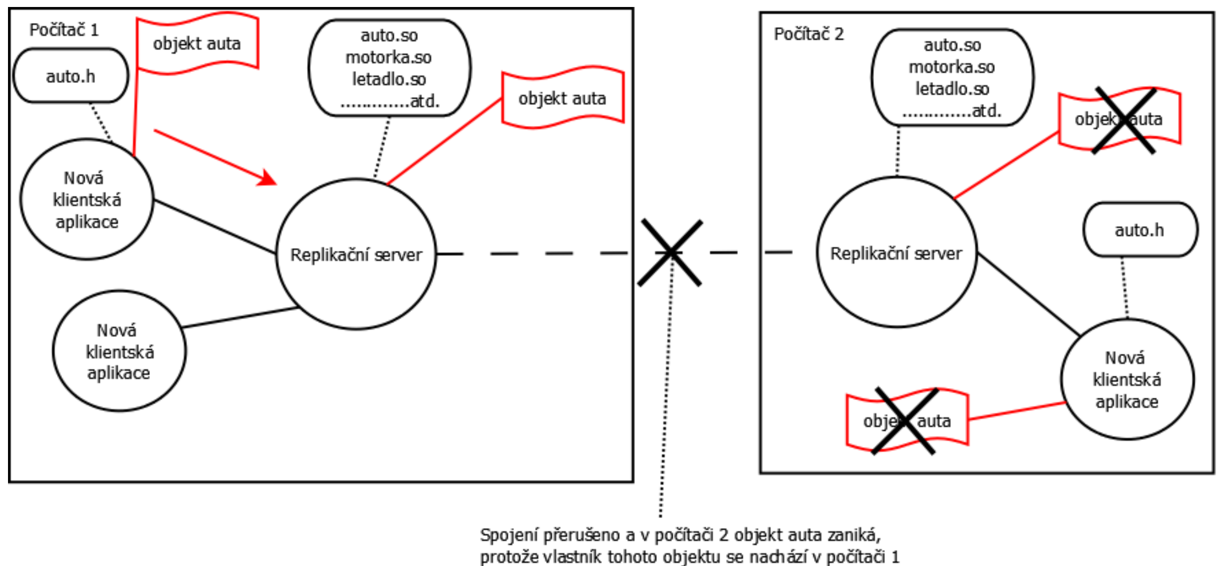
Další podmínkou pro rozšíření replikovaného objektu do všech uzlů sítě je stejná verze pluginu replikovaného objektu. Pokud některý replikační server narazí na nepodporovanou verzi objektu, tak tuto zprávu zahodí a dál tento objekt nešíří.

Velmi důležitou vlastností replikačního řešení firmy RETIA, a.s. je způsob, jakým lze určit vlastníka daného objektu. UUID slouží pouze jako unikátní identifikátor, zda klientská aplikace je nebo není vlastníkem replikovaného objektu. UUID však nic neříká v jaké místě replikační sítě se vlastník nachází. Z tohoto důvodu si replikační řešení umí v každém uzlu zapamatovat, odkud nový objekt přišel (z jakého spojení). Pokud je potřeba vlastníka objektu kontaktovat, tak stačí pozpátku replikační sítí projít uzly a v každém z nich je známo, z jakého spojení tento objekt přišel a právě do tohoto uzlu je nutné se vydat a takto rekurzivně pokračovat, dokud nebude nalezen uzel se shodným UUID.

Další změnou v replikační síti může být připojení nové klientské aplikace na replikační server. V tomto případě se po vzájemné komunikaci klientské aplikace řízené protokolem replikační služby a replikačního serveru pošlou všechny replikované objekty do klientské aplikace, o které klientská aplikace požádá.

Odpojení libovolného uzlu (replikační server nebo klientská aplikace) od replikační sítě způsobí zánik všech objektů vlastněných klientskými aplikacemi za tímto uzlem (Obrázek 9). Každý uzel sítě využívá replikační službu, která obsahuje smyčku událostí. Mezi takové události patří i pravidelné „keepalive“ zprávy na své sousední uzly. Pokud několik takových zpráv za sebou nedorazí do sousedního uzlu, je tento uzel považován za odpojený a všechny objekty vlastněné

klientskými aplikacemi za tímto uzlem jsou smazány v celé replikační síti. Takto lze detekovat i hardwarové odpojení síťového kabelu a jiné nekorektní formy odpojení od části sítě.



Obrázek 9 - Příklad odpojení části sítě

zdroj: vlastní

Další změnou v replikační síti může být modifikace objektu. Pokud je replikovaný objekt změněn, je tato změna záplavově šířena stejně jako u nasdílení nového objektu. Možností, jak udělat změnu je hned několik. Daná metoda objektu může povolit změnu provedenou pouze vlastníkem pomocí funkce `owned()`, viz oddíl 1.2. Pokud tedy tuto metodu zavolá klientská aplikace, která není vlastníkem objektu, nic se nestane. Druhou možností, jak změnit objekt je pomocí RPC (vzdáleného volání procedur). Pokud nemůže klientská aplikace uplatnit změnu z toho důvodu, že není vlastníkem objektu, lze tuto změnu zavolat vzdáleně od vlastníka pomocí metody `call()`. Replikační sítí se tedy k vlastníkovi objektu dostane žádost o vzdálené volání metody a poté je tato metoda zavolána. Pokud chceme změnu atributu rozšířit do replikační sítě, musí být zavolána metoda `notify()`. Jestliže není tato metoda zavolána, tak se o změně objektu v replikační síti nikdo nedozví. Programátor si tedy může vybrat, zda metodu `notify` zavolá nebo ne, v závislosti na tom, zda má být daný atribut replikovaný do všech uzlů sítě nebo ne.

RETIA, a.s. - librpl: rpl_testobject.h

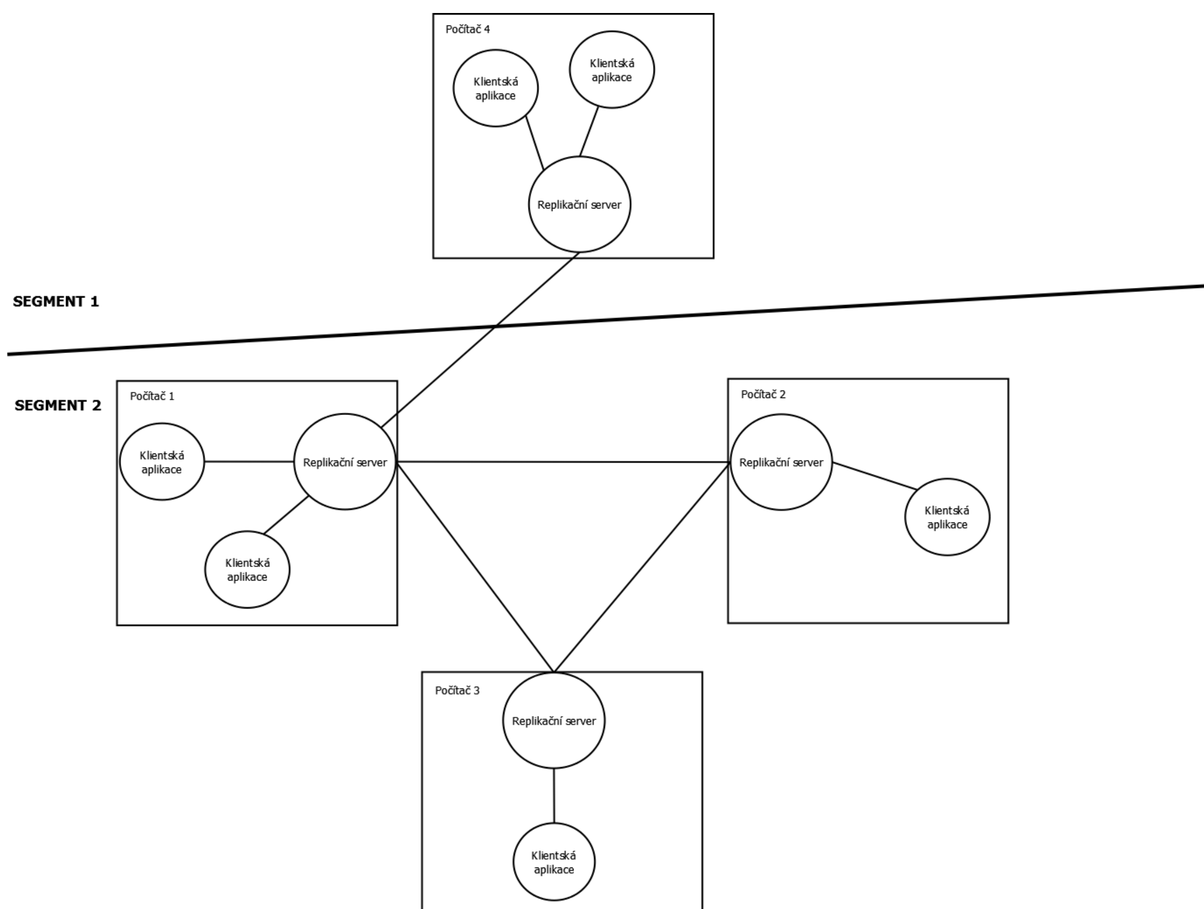
```
.................
.................

private:

      std::vector<uint8_t> stack_;
.................
.................
public:

void pushByte(uint8_t byte)
    {
        if (owned()) //True pokud je aplikace vlastník tohoto objektu
        {
            // Vykonání změny objektu
            stack_.push_back(byte);

            // Rozeslání do replikační sítě zprávu o změně
            notify(AStack, AInsert) << byte;

        }
        else
        {
           /* Pokud aplikace není vlastník objektu, zavolej metodu
      vzdáleně přes vlastníka */
            call(MPushByte) << byte;
        }
    }
```

Další speciální možností, jak objekt změnit, je možnost atomické modifikace objektu. Při každém volání metody objektu se pomocí volání `notify()` vygeneruje zpráva, která se rozšíří celou replikační sítí. Pokud je však potřeba, změnit více atributů naráz, např. letadlo změní polohu a rychlost, tak je nutno tuto informaci distribuovat najednou – atomicky, protože letadlo nezměnilo rychlost a až potom svou polohu. Makro `RPL_ATOMIC()` zajistí atomičnost operace v rámci programového bloku {}, ve kterém seskupí všechna volání `notify()` do jedné zprávy. Do všech uzlů sítě poté dorazí pouze jedna zpráva, která modifikuje více atributů naráz.

1.2.3 Segmenty

V pododdílu 1.2.2 je ukázka zakázaného cyklu jakožto topologii v replikačním řešení firmy RETIA, a.s. Při šíření změny každý uzel sítě záplavově rozešle změnu dál a v případě uzavřeného cyklu v topologii replikační sítě by změna kolovala donekonečna. Z tohoto důvodu byla implementována funkce segmentů, které rozdělují oblast na segmenty. Každý uzel sítě má

definovaný segment. V rámci segmentu se změny příchozí ze stejného segmentu nešíří. Takže z obrázku 10 je zřejmé, že počítač 4 může odeslat zprávu o změně atributu počítači 1, počítač 1 tuto změnu přepošle počítači 2 i 3 jelikož zpráva dorazila ze segmentu 1, ale počítač 2 ani 3 již tuto změnu nepřeposílá, jelikož byla zpráva přeposlána ze stejného segmentu, ve kterém se nachází. Tímto způsobem, je možné v topologii vytvořit cyklus.



Obrázek 10 - Segmenty v topologii replikační sítě

zdroj: vlastní

I přes funkce segmentů zde můžou vniknout nekonzistentní stavy a hrozí zacyklení. Například některému uzlu dojde zpráva o změně dvakrát. To se může stát především při použití paralelního spojení mezi segmenty, například v ilustraci Obrázek 10 kdyby byl počítač 4 propojen ještě s počítačem 2. V replikačním řešení je paralelní spojení mezi segmenty považováno za nepodporovanou konfiguraci. Záleží tedy především na uživateli, jak segmenty rozvrhne. Ve výchozím nastavení jsou segmenty nastaveny tak, že za každým síťovým rozhraním počítače je vždy jiný segment.

1.3 Dostupná řešení

V tomto oddílu budou zmíněny jiné dostupné řešení replikace a budou popsány ty nejdůležitější vlastnosti a některé odlišnosti v porovnání s replikačním řešením ve firmě RETIA, a.s.

1.3.1 ReplicaNet

Tato multiplatformní síťová knihovna je veřejně dostupná a stejně jako řešení ve firmě RETIA, a.s. je i knihovna ReplicaNet implementována v programovacím jazyce C++. Prvním podstatným rozdílem je licencování. Řešení šité na míru ve firmě RETIA, a.s. není veřejně dostupné a tudíž, zde není ustavena žádná licenční politika. Replikační řešení ReplicaNet je zdarma pouze pro nekomerční použití a je dodáno bez zdrojových kódů knihoven. V opačném případě licence stojí 6500 $ a je dostupná včetně zdrojových kódů knihoven i včetně ladících nástrojů a lze ji použít bez omezení pro komerční použití (ReplicaSoftware, 2017).

ReplicaNet je multiplatformní řešení a lze jej použít na operačních systémech Windows, Linux i Mac OS. Naproti tomu je replikační řešení ve firmě RETIA, a.s. použito pouze na linuxové distribuci, a to především na distribuci OpenSuse. Dalším významný rozdílem je způsob komunikace mezi uzly replikační sítě. Ve firmě RETIA, a.s. používají replikační servery pouze spolehlivý způsob zasílání zpráv přes protokol TCP. Jelikož je toto replikační řešení nasazeno vždy na lokální síti, tak zde zpravidla nevznikají situace ztráty zpráv a tudíž vyšší režie protokolu TCP není omezením. Naproti tomu řešení ReplicaNet může být použito například v nějakém komerčním herním titulu pro více hráčů, ve kterém se hráči připojují přes internet, a rychlá odezva zde může být prioritou před spolehlivostí (například pro zasílání zpráv o pozici uživatele), a proto je zde i podpora zasílání zpráv přes protokol UDP (ReplicaSoftware, 2017).

V praxi jsou zde ještě dva podstatné rozdíly, které je nutné zmínit. V replikačním řešení firmy RETIA, a.s. je především na programátorovi, aby replikovaná třída měla všechny náležitosti, všechna potřebná makra a všechna data byla správně serializována a deserializována. Naproti tomu ReplicaNet používá svůj vlastní jazyk ROL (Replica Object Language) na popis replikovaných atributů. Druhým podstatným rozdílem je množství funkcí, které řešení ReplicaNet poskytuje. Lze tak nastavit například migraci objektů k jinému vlastníkovi v případě chyby připojení některého z klientů. V řešení firmy RETIA, a.s. jsou vždy objekty vlastníka v případě ztráty spojení vymazány z celé replikační sítě (ReplicaSoftware, 2017).

Obě řešení mají však i mnoho společných rysů, a to především způsob jakým fungují. Vždy je zde replikovaná třída zděděná z jiné třídy, jejíž objekt je ve všech uzlech replikační sítě zkonstruován se shodnou hodnotou atributů. V případě ReplicaNet stačí třídu vydědit pouze ze

třídy `_RO_DO_PUBLIC_RO`. `BaseObjekt` je v tomto případě vlastní třída, která pouze sjednocuje rozhraní pro objekty podobného typu a vůbec zde být nemusí.

ReplicaNet tutorial: player.h

```
#ifndef _PLAYER_H_
#define _PLAYER_H_

#include "BaseObject.h"
#include "_RO_Player.h"

class Player : _RO_DO_PUBLIC_RO(Player) , public BaseObject
{
public:
    Player();
    virtual ~Player();

    void Init(void);
    void Tick(float timeDelta);
    void Render(void);

    void PostObjectCreate(void)
    {
        Init();
    }

    int mPlayerColour;
};
```

Pro takto definovanou třídu v hlavičkovém souboru stačí implementovat metody v souboru player.cpp dle libosti (například pouze textovými výpisy, nic konkrétního zde totiž pro funkčnost replikace jako v řešení RETIA, a.s. být nemusí). ReplicaNet používá pro popis replikovaných atributů svůj vlastní jazyk ROL a zápis v tomto jazyce je velmi intuitivní.

ReplicaNet tutorial: _RO_Player.rol

```
object Player
{
    datablock NData;

    networking
    {
        NData(mPlayerColour)
        {
            Reliable;
        }
    };
}
```

Takto definovaný ROL soubor lze přeložit na `.cpp` a `.h` soubor pomocí přiloženého kompilátoru, který vyžaduje pouze jeden povinný parametr a to je cesta k `.rol` souboru. Ostatní dva parametry jsou nepovinné (ReplicaSoftware, 2017).

RNROLCompiler.exe <input.rol> <output.cpp> <output.h>

Takto se ale ROL soubory nemusí kompilovat pokaždé ručně, ale lze ve vývojovém prostředí tuto komplikaci nastavit a zautomatizovat pro překlad celého projektu včetně těchto ROL souborů.

Ve výše zmíněném `.rol` souboru je definována replikace atributu `mPlayerColour` třídy `Player`. Tento atribut je označen parametrem `Reliable` – tzn., že jde o spolehlivý přenos. ReplicaNet rozlišuje celkem 4 druhy přenosu dat a v závislosti na něm je v podpalubí knihoven použit protokol TCP, UDP nebo UDP s potvrzováním (ReplicaSoftware, 2017).

1. Reliable – Spolehlivý: Zaručuje přenos atributů a doručení ve stejném pořadí, v jakém byly vyslány.
2. Certain – Jistý: Zaručuje doručení, ale ne pořadí v jakém budou data doručeny.
3. Unreliable – Nespolehlivý: Nezaručuje doručení ani pořadí v jakém budou data doručeny.
4. Unreliable ordered – Nespolehlivý uspořádaný: Nezaručuje doručení dat, ale zaručuje správné pořadí, v jakém budou doručeny.

(ReplicaSoftware, 2017)

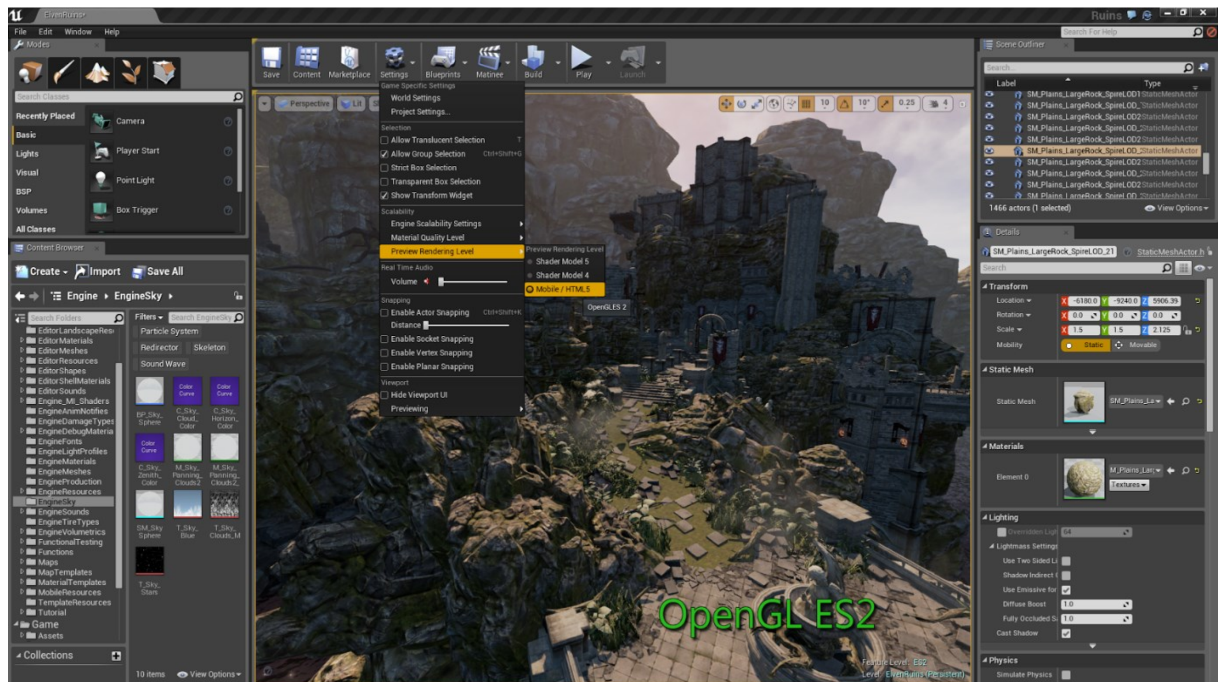
S hotovým replikovaným objektem lze již pracovat spolu s funkcemi replikační knihovny a tím tak tento objekt sdílet napříč klienty. V příloze B je celý zdrojový kód. Tato ukázka obsahuje pouze ty nejpodstatnější volání.

ReplicaNet tutorial: main.cpp

```
.................
.................

int main(int argc,char **argv)
{
    RNReplicaNet::ReplicaNet *myNetwork = new RNReplicaNet::ReplicaNet;
    .................
    .................
    // Hledání sousedních session
    myNetwork->SessionFind();
    .................
    .................
    do
    {
    // Výsledek vyhledávání sousedních session
        found = myNetwork->SessionEnumerateFound();
    .................
    .................
    } while (found != "");

    // Připoj se k existující session pokud existuje nebo vytvoř novou
    if (firstFound == "")
    {
        myNetwork->SessionCreate("Tutorial1");
    }
    else
    {
        myNetwork->SessionJoin(firstFound);
    }
    // Alokace replikovaného objektu Player
    RNReplicaNet::ReplicaObject *myPlayer = new Player;
    myPlayer->PostObjectCreate();
    // Nasdílení objektu do replikační sítě
    myPlayer->Publish();
    .................
    .................
    // Smyčka událostí
    bool doGameLoop = true;
    while(doGameLoop)
    {
        RNReplicaNet::ReplicaObject *iterator;
        myNetwork->ObjectListBeginIterate();
        .................
        .................
        iterator = myNetwork->ObjectListIterate();
        while (iterator)
        {
            BaseObject *baseObject =
            (BaseObject *)iterator->GetOpaquePointer();
            baseObject->Tick(myNetwork->GetLocalTime());
            iterator = myNetwork->ObjectListIterate();
        }
        myNetwork->ObjectListFinishIterate();
        RNReplicaNet::CurrentThread::Sleep(1000);
    }

    return 0;
}
```

1.3.2 Unreal Engine

Tento herní engine zahrnuje spoustu možností pro vytváření počítačových her obdobně jako Unity Engine. Unreal Engine je vytvořený v jazyce C++ a vývojové prostředí stejně jako Unity Engine podporuje vysokoúrovňovou práci nad celou aplikací (Obrázek 11).

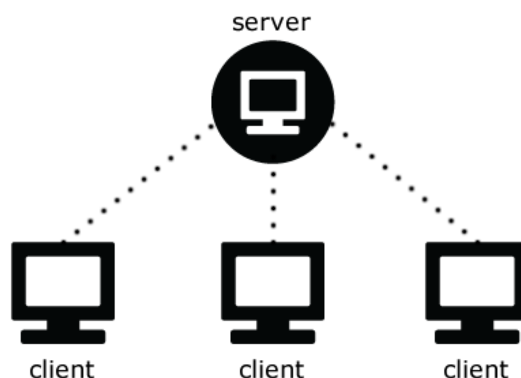


Obrázek 11 - Vývojové prostředí Unreal Engine 4

zdroj: https://forums.unrealengine.com/showthread.php?53230-Unreal-Engine-4-6-Released!

Stejně jako u komerčního produktu ReplicaNet, tak je i Unreal Engine vázán licenčními poplatky. Obdobně jako Unity je Unreal Engine zdarma pro hry, které budou šířeny nekomerčně. Pokud se bude jednat o komerční produkt, je nutné zaplatit 5 % z částky, která za roční čtvrtletí přesáhne částku 3000 $.

Replikace je v tomto enginu na vysoké úrovni, ale je především zaměřena na herní odvětví. Replikační síť ve firmě RETIA, a.s. je decentralizovaná a nelze tedy zrušením jednoho uzlu sítě dosáhnout nefunkčnosti celé sítě. V tomto směru se toto řešení chová stejně jako síť Internet - například narušením jednoho uzlu sítě může být způsoben maximálně výpadek lokálně, nikoliv globálně. Topologie Unreal Engine je především architektura autoritativního server/klient modelu. Znamená to tedy, že veškerá komunikace proudí na autoritativní server, který je zodpovědný za validaci zpráv a následnou replikaci dat pro své klienty (Obrázek 12).



Obrázek 12 - Autoritativní klient/server architektura

zdroj: https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074

Tím čím se replikace v Unreal Engine odlišuje od řešení firmy RETIA, a.s. ale i ReplicaNet je specializací na herní průmysl. Například pohyb objektů ve virtuálním světě je dílem pravidelně odesílaných zpráv klienty obsahující změnu pozice. Obvykle je zde použit nespolehlivý protokol UDP, kde v praxi může dojít k výkyvům v pořadí došlých zpráv. Aby takovýto pohyb napříč sítí internet byl pro všechny klienty/hráče plynulý, jsou tyto zprávy na serveru opravovány a mohou zahrnovat jistou míru predikce. Také zde můžou být validována data, které mohou naznačovat podvádění (naměřena větší rychlost, než je možné). Následně jsou tyto zprávy rozreplikovány na všechny klienty (UnrealEngine, 2017).

Unreal Engine podporuje celkem čtyři síťové role objektů:

1. ROLE_None – Objekt nemá žádnou síťovou roli a není replikován.
2. ROLE_SimulatedProxy – Objekt lokálně kopíruje objekt na serveru, avšak nemá žádné pravomoce cokoliv změnit ani vzdáleně volat.
3. ROLE_AutonomousProxy – Objekt lokálně kopíruje objekt na serveru a může objekt měnit pomocí vzdálených volání.
4. ROLE_Authority – Tento objekt je replikován na klienty a je v podstatě originálem, nad kterým jsou volána vzdálená volání od klientů v roli ROLE_AutonomousProxy.

(UnrealEngine, 2017)

Implementačně je na tom Unreal Engine velice podobně jako u ostatních řešení. Replikovaná třída musí dědit z nějaké nadřazené a za použití makra `UPROPERTY` je zde specifikováno, které atributy mají být replikovány.

Unreal Engine Replication: ReplicatedActor.h

```
#pragma once
#include "Core.h"
#include "AReplicatedActor.generated.h"

UCLASS()
class AReplicatedActor : public AActor
{
    GENERATED_UCLASS_BODY()
public:
    /** A Replicated Boolean Flag */
    UPROPERTY(Replicated)
    uint32 bFlag:1;

    /** A Replicated Array Of Integers */
    UPROPERTY(Replicated)
    TArray<uint32> IntegerArray;

    UFUNCTION(Server, Reliable, WithValidation)
    void Server_ReliableFunctionCallThatRunsOnServer();

    UFUNCTION(Client, Reliable)
    void Client_ReliableFunctionCallThatRunsOnOwningClientOnly();

    UFUNCTION(NetMulticast, Unreliable)
    void Client_UnreliableFunctionCallThatRunsOnAllClients();

private:
    UFUNCTION()
    void OnRep_Flag();
};
```

Unreal Engine Replication: ReplicatedActor.cpp

```
#include "ReplicatedActor.h"
#include "UnrealNetwork.h"
AReplicatedActor::AReplicatedActor(const class
FPostConstructInitializeProperties& PCIP)
: Super(PCIP)
{
    bReplicates = true;
}
void AReplicatedActor::ServerSetFlag()
{
    if (HasAuthority() && !bFlag) // Ensure Role == ROLE_Authority
    {
        bFlag = true;
        OnRep_Flag(); // Run locally since we are the server this won't
be called automatically.
    }
}
void AReplicatedActor::OnRep_Flag()
{
    // When this is called, bFlag already contains the new value. This
    // just notifies you when it changes.
}
.................
.................
```

Stejně jako replikační řešení firmy RETIA, a.s. tak i Unreal Engine podporuje vzdálená volání. Před danou metodu stačí napsat makro/anotaci UFUNCTION(). Unreal Engine podporuje dokonce 3 druhy vzdálených volání:

1. Server – Funkce se vykoná pouze nad objektem serveru (role objektu musí být ROLE_Authority nebo ROLE_AutonomousProxy).
2. Client – Funkce se vykoná nad objektem klienta (role objektu musí být ROLE_AutonomousProxy)
3. NetMulticast – Funkce se vykoná na všech klientech (role objektu musí být ROLE_Authority).

2 FRAMEWORKY UŽIVATELSKÉHO ROZHRANÍ

Ladící nástroj pro replikační řešení firmy RETIA, a.s. bude aplikací, která za běhu vygeneruje grafické uživatelské rozhraní (GUI) pro replikovaný objekt. Musí umožnit uživateli volat jeho metody přes prvky grafického rozhraní, a proto volba správného GUI frameworku je důležitá.

Bez GUI frameworku by bylo nutné vykreslovat a implementovat logiku každého prvku uživatelského rozhraní jako je checkbox nebo tlačítko zvlášť. GUI frameworků existuje celá řada a určitě mezi velmi používané patří i Windows Forms, WPF, AWT, Swing nebo JavaFX avšak u většiny z nich je problém s podporou pro operační systém OpenSuse, který je ve firmě RETIA, a.s. nejvíce používaným systémem.

Zdrojové kódy jsou napříč těmito frameworky velmi podobné a ani ve wxWidgets, QT, nebo GTK+ se nepoužívají žádné speciální techniky, kterými by se tyto frameworky odlišovaly. Jde o velmi podobně techniky (např. obslužné metody událostí) provázání grafických prvků a programových částí stejně jako například v jazyce Java v prostředí Netbeans nebo C# v prostředí Microsoft Visual Studio.

2.1 wxWidgets

Tento framework obsahuje sadu multiplatformních knihoven umožňující tvorbu multiplatformních GUI aplikací. Hlavní výhoda spočívá ve svobodné licenci, která nemá vliv u komerčního použití.

2.1.1 Historie

WxWidgets byla původně vyvíjena Jiluanem Smartem na Univerzitě Edinburgh (Velká Británie) pro interní účely. Veřeně byla uvedena v roce 1992. V roce 1999 byla značně vylepšena a uvedena jako verze 2. Posledním velkým updatem bylo vydání verze 3 v roce 2013. V současnosti je wxWidgets vyvíjeno stále Julianem Smartem a mnoha dalšími. Mezi hlavní vývojáře, kterého zmiňuje i oficiální stránka wxWidgets patří i Čech Václav Slavík (wxWidgets, 2017).

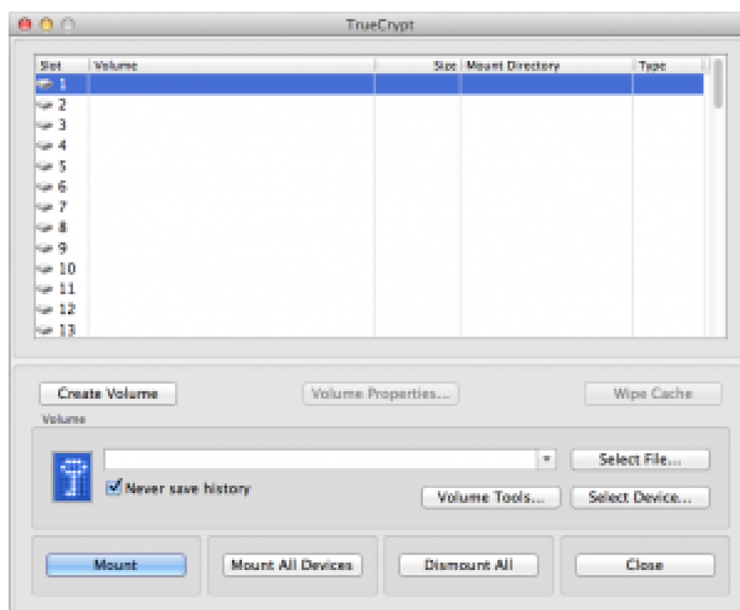
2.1.2 Vlastnosti

Tato knihovna je zdarma včetně zdrojových kódů a zároveň je multiplatformní takže zajišťuje podporu na operačních systémech Windows, Linux nebo Mac OS. Jako jediná z těchto třech knihoven je zdarma i pro komerční použití bez omezení (dáno licencí LGPL s dovětkem), což se například o knihovně QT říci nedá (Zachar, 2008).

Hlavním programovacím jazykem je C++, ale je zde podpora i pro jazyky Python, Perl, Ruby i Java. WxWidgets není pouze knihovna na tvorbu uživatelského rozhraní, ale stejně jako knihovna QT umožňuje wxWidgets spravovat přístup k souborům, síťová spojení, práci s databází a dokonce je zde podpora i pro vícevláknové aplikace. Knihovna pro vykreslování komponent využívá SDK operačního systému, takže výsledná aplikace vypadá pro daný operační systém přirozeně.

2.1.3 Vývoj a použití

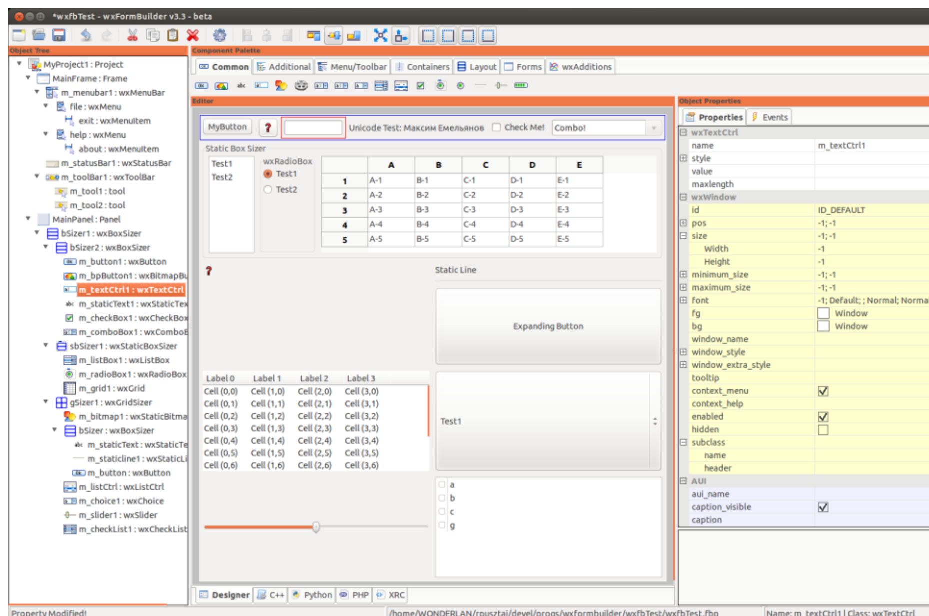
Velice známými programy vytvořené ve wxWidgets je program Code::Blocks nebo TrueCrypt (Obrázek 13). WxWidgets byl dále použit i v aplikaci VLC Media Player, ale od verze 0.9 z roku 2008 vývojáři přešli na knihovnu QT (Zachar, 2008).



Obrázek 13 - Uživatelské rozhraní TrueCrpyt vytvořené ve wxWidgets

zdroj: http://www.nerdenmeister.org/2013/08/16/build-truecrypt-on-os-x-64-bit-with-hardware-acceleration/

Pro wxWidget je hned několik vývojových prostředí (IDE) podporujících grafickou editaci prvků uživatelského rozhraní. Některá vývojová prostředí podporují wxWidgets až po doinstalaci pluginu. Mezi podporované prostředí patří wxFormBuilder (Obrázek 14), wxDev-C++ nebo Code::Blocks s pluginem wxSmith.



Obrázek 14 - IDE wxFormBuilder

zdroj: https://sourceforge.net/projects/wxformbuilder/

2.2 GTK+

Tento framework podporuje tvorbu prvků uživatelského rozhraní. V současnosti je tato knihovna využívána jako hlavní v desktopovém prostředí GNOME. Na rozdíl od knihoven QT a wxWidgets, které podporují mnoho užitečných knihoven (např. pro síťovou komunikaci či práci se soubory), umožňuje framework GTK+ tvorbu pouze GUI elementů.

2.2.1 Historie

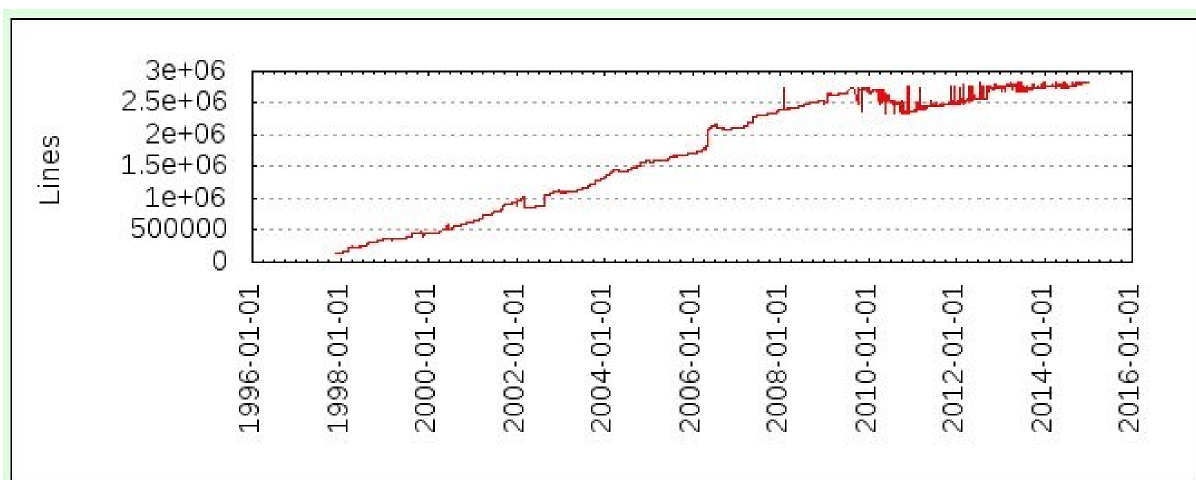
Knihovna jako taková vznikla již v roce 1996 a byla vytvořena původně pouze pro grafický program GIMP (GTK = GIMP ToolKit), který je dnes volně šiřitelnou alternativou (Obrázek 16) k rastrovému editoru Adobe Photoshop. GTK byla v začátcích přepsána na objektovou verzi s názvem GTK+ pomocí knihoven GLib (jelikož jde o knihovnu napsanou v jazyce C, který objekty nepodporuje). První verze byla veřejně uvedena v roce 1998 a už ve verzi 1.2 vydanou v roce 1999 poskytovala knihovna dostatek grafických prvků, aby mohla být použita univerzálně na jakoukoliv aplikaci. Později se knihovna GTK+ rozpadla na několik nezávislých podknihoven jako je například Pango pro rendering textu. Knihovna se od verze 2.0 ujala u desktopového grafického rozhraní GNOME a Xface a současně se stále rozvíjí. Rapidní vývoj v počátcích vývoje popisuje Tabulka 1 (Redhat, 2017).

Tabulka 1 - Rozsah zdrojového kódu GTK+ v počátcích vývoje

Verze	Rok	Řádků kódu
1.0	duben 98	93000
1.2	únor 99	160000
2.0	březen 02	460000
2.2	prosinec 02	488000
2.4	březen 04	558000

zdroj: http://people.redhat.com/mclasen/Usenix04/notes/x29.html

V současnosti knihovna zahrnuje celkem přes 2,8 milionů řádků kódu (Obrázek 15).



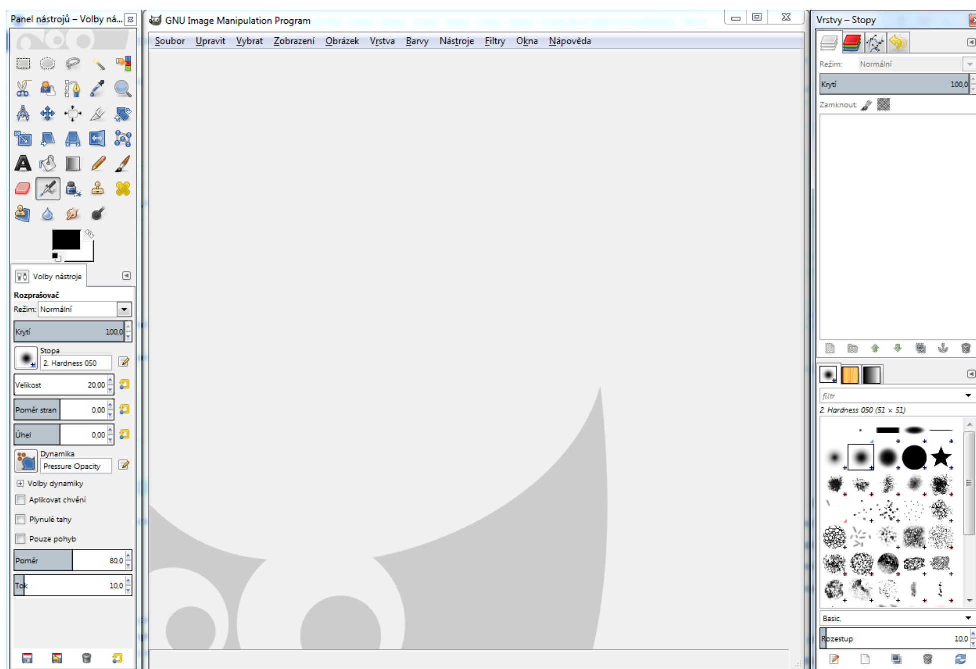
Obrázek 15 - Graf počtu řádků zdrojového kódu knihovny GTK+ napříč lety

zdroj: http://www.phoronix.com/scan.php?page=news_item&px=MTg3ODQ

2.2.2 Vlastnosti

GTK+ patří dnes mezi nepoužívanější knihovny na linuxových distribucích pro tvorbu uživatelského rozhraní, ale na rozdíl od knihovny QT nebo wxWidget je knihovna GTK+ zaměřená pouze na tvorbu GUI. Dalším významným rozdílem je použití programovacího jazyka C. Pro programování v jazyce C++ je nutné použít nadstavbu gtkmm (Karas, 2006).

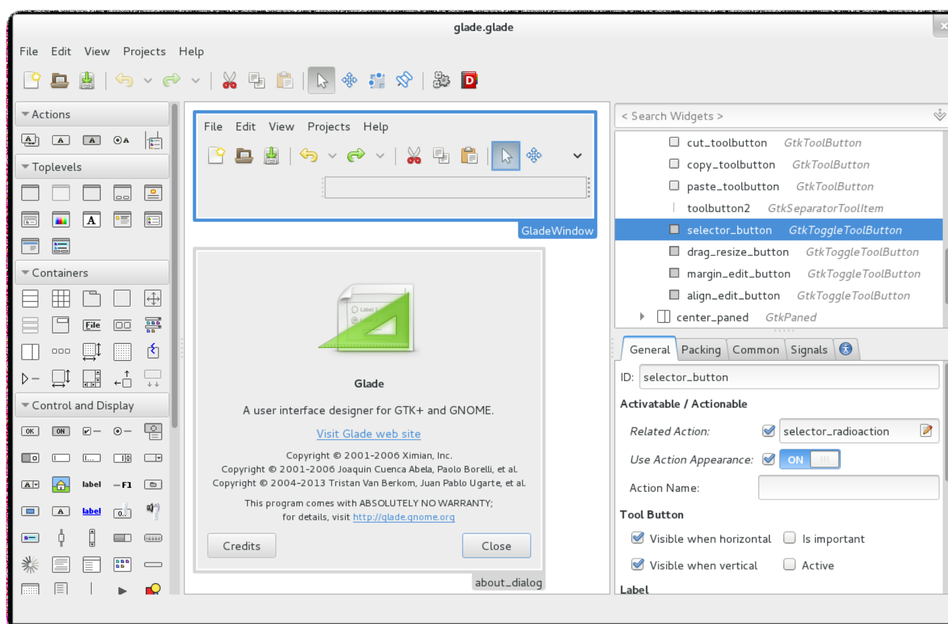
Tuto knihovnu využívá kromě grafického programu GIMP (Obrázek 16) i jiná spousta programů jako je Inkscape (nejpopulárnější Linuxový vektorový editor), XChat (IRC klient) a další.



Obrázek 16 - Grafický rastrový editor GIMP využívající knihovnu GTK+

zdroj: vlastní

Mezi jediným IDE pro GTK+ podporujícím grafický designer pro tvorbu GUI a s podporou operačního systému Windows je Glade (Obrázek 17). Dalším prostředím, avšak podporujícím pouze Linux je Anjuta Devstudio (Karas, 2006).



Obrázek 17 - IDE Glade pro GTK

zdroj: https://glade.gnome.org/

2.3 QT

Sada knihoven QT umožňuje multiplatformní vývoj GUI aplikací. QT obsahuje širokou škálu knihoven, které usnadňují vývoj kompletní aplikace. Některé knihovny obalují nízko úrovňové API operačního systému, jako např. knihovna pro vytváření síťových spojení, ve které je umožněno vytvořit TCP spojení jednoduše a zároveň objektově. Na rozdíl od knihoven wxWidgets a GTK+ rozvíjí QT vlastní IDE, které umožňuje navrhovat GUI aplikace přímo ve vestavěném designeru.

2.3.1 Historie

Samotná historie vzniku grafické knihovny QT sahá do podobného období jako předchozí dvě knihovny. Tentokrát se o vznik postarala norská dvojice Havard Nord a Eirik Chambe-Eng, kteří v roce 1990 pracovali na aplikaci uchovávající snímky z ultrazvuku. Zároveň potřebovali, aby aplikace fungovala jako GUI aplikace na operačním systému Unix, Macintosh i Windows. Oba se tedy rozhodli, že je potřeba vymyslet grafickou objektově orientovanou knihovnu, která bude multiplatformní. V roce 1991 byly napsány první třídy a zároveň vymyšlen systém signálů a slotů, který byl přejat mnoha dalšími grafickými knihovnami. O tři roky později, v roce 1994 založili společnost a rozhodli se jít s tímto produktem na trh. Dnes se tato společnost jmenuje Trolltech. Název QT vznikl z písmene Q, které se Haavardu Nordovi líbilo ve fontu Emacs a je výhradně použito jako prefix ve všech knihovnách náležící QT. Písmeno T označuje slovo toolkit (Shamendra, 2014).

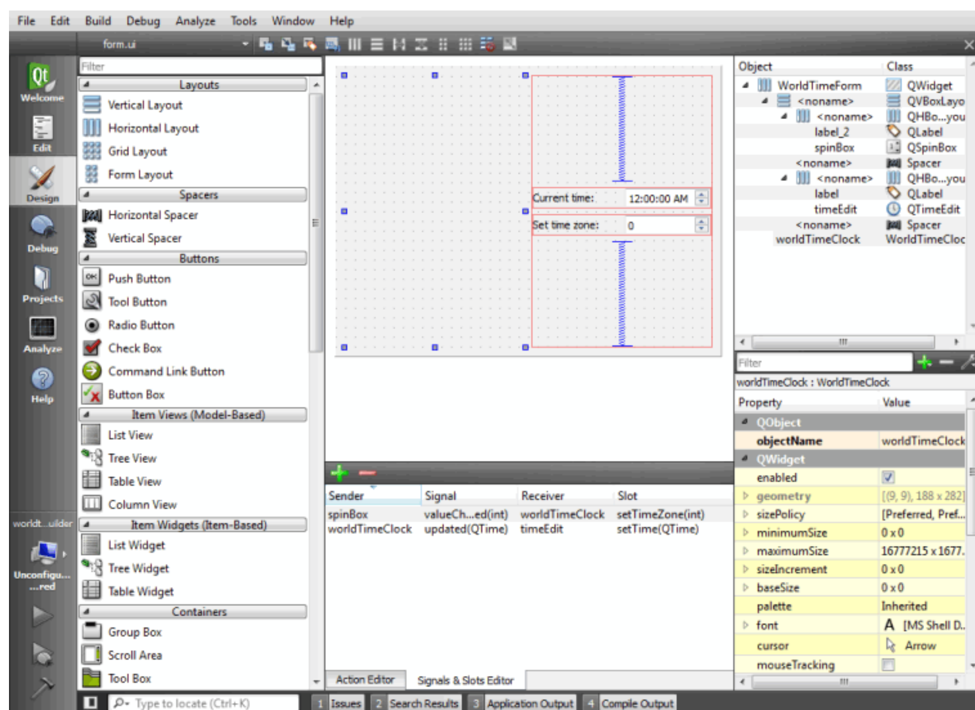
V roce 1995 získali oba zakladatelé první kontrakt u společnosti Metis, kde potřebovali využít knihovnu QT pro vývoj jejich vlastního softwaru. V pořadí druhým zákazníkem se v roce 1996 stala Evropská vesmírná agentura, která nakoupila hned 10 licencí. V následujících měsících byli do vývoje začleněni další programátoři. V roce 1997 vnikal projekt KDE, jakožto nové desktopové prostředí a volba GUI frameworku padla právě na QT. QT se tak stalo standardem na vývoj GUI aplikací v prostředí oken KDE. V roce 1999 vychází i první opensource verze, která má podporovat vývoj právě v prostředí KDE. V roce 2008 je QT (Trolltech) odkoupeno společností Nokia. Poslední verze 5 podporuje modul QT Quick, který zcela odděluje uživatelské rozhraní od programové části podobně jako technologie WPF.

2.3.2 Vlastnosti

Použití pro nekomerční účely je u knihovny QT zdarma. U komerčního použití jsou ceny nastaveny od 79 $ měsíčně, ale to je pouze v ideálním případě. Obchodní model zahrnuje škálování, co se týče licencování a to znamená, že ceny nejsou stanoveny fixně pro určitou

platformu a je nutné cenu pro konkrétní využití na konkrétní platformě poptat přímo na stránkách QT (QT-Company, 2017).

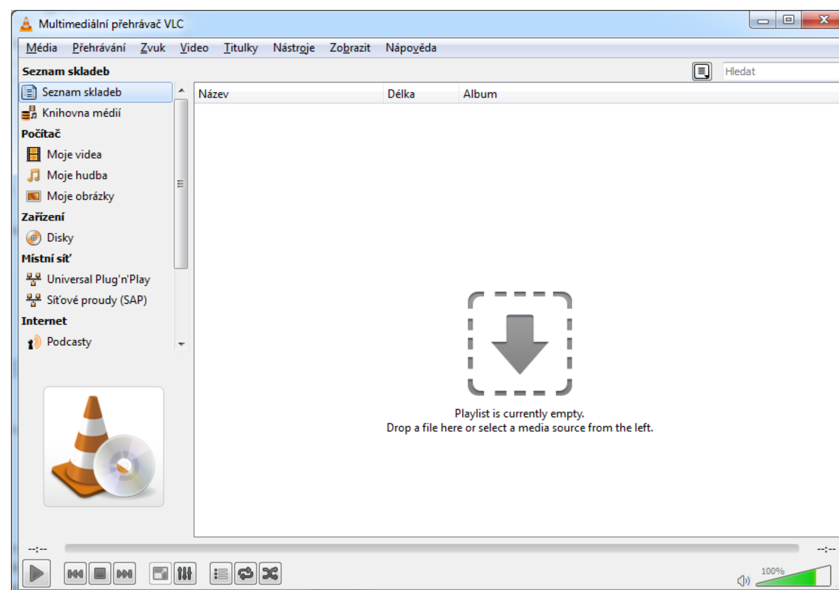
QT na rozdíl od ostatních frameworků vyvíjí i vlastní IDE nástroj – QT Creator (Obrázek 18), který je také multiplatformní a funguje tedy na operačních systémech Windows, Linux i MacOS. S tímto nástrojem je možné vyvíjet i univerzální Windows aplikace UWP, aplikace na iOS 7 a vyšší, a dokonce i na Android (QT-Company, 2017).



Obrázek 18 - QT Creator

zdroj: http://doc.qt.io/qtcreator/creator-using-qt-designer.html

Knihovna QT je dnes poměrně populárním multiplatformním prostředkem pro vytváření GUI aplikací jako je Skype, nebo VLC Media Player (Obrázek 19) případně VirtualBox.



Obrázek 19 - VLC Media Player využívající knihovnu QT pro grafické rozhraní

zdroj: vlastní

QT Creator umožňuje GUI vytvářet přes vestavěný designer, který vytváří `.ui` soubory. Zvláštností tohoto frameworku je speciální MOC (meta object compiler), který tyto `.ui` soubory založené na XML, převede na klasické `.cpp` a `.h` C++ soubory, ve kterých jsou konkrétní grafické prvky vytvářeny pomocí zdrojového kódu.

QT se nezaměřuje pouze na vytváření GUI, ale obsahuje spoustu pomocných tříd od práce se soubory po vyváření vícevláknových aplikací.

2.4 Výběr GUI frameworku pro diplomovou práci

Za hlavní faktory při výběru správného GUI frameworku stojí určitě výhodnost použití. Jakožto ladící nástroj, který bude využit programátory při návrhu a testování nového replikovaného objektu, musí být zohledněna právě kompatibilita. Samotné replikační řešení firmy RETIA, a.s. není multiplatformní. Není to z toho důvodu, že by to nešlo, ale především z toho důvodu, že to není potřeba. Výběr GUI frameworku proto může být zjednodušen na operační systém Linux.

Velmi důležitým faktorem pro nasazení ladicího nástroje napříč programátory je podpora GUI frameworků na současných počítačích. Na linuxově založených systémech je velmi časté šířit aplikaci v podobně zdrojového kódu a na konkrétní distribuci je poté aplikace zkompilována až přímo u uživatele. Tento fakt umožní aplikaci šířit a zajistit její funkčnost například na

distribuci Ubuntu, tak i na distribuci OpenSuse. Při kompilaci aplikace jsou použity dostupné knihovny a hlavičkové soubory ze systému, ale pokud chybí, program nepůjde zkompilovat.

Jelikož ladící nástroj není první GUI aplikací ve firmě RETIA, a.s., bude v mé diplomové práci využit framework QT, který je zde hojně rozšířen napříč programátory a překlad aplikace závislé na knihovnách QT, tak bude o moc jednodušší.

3 SYNTAKTICKÁ ANALÝZA JAZYKA C++

Druhým významným pilířem pro vytvoření ladícího nástroje je správná analýza třídy. Aby bylo možné volat metodu konkrétního objektu, je nutné znát názvy metod včetně parametrů. Jediným spolehlivým zdrojem těchto informací je třída objektu samotná. V replikačním řešení firmy RETIA, a.s. je replikovaný objekt definován jako třída implementovaná kompletně v jednom hlavičkovém souboru. Ladicí nástroj musí nějakým způsobem zjistit názvy metod a názvy parametrů těchto metod, aby mohlo být vygenerováno příslušné GUI.

3.1 CastXML

Tato knihovna umožňuje informace o zdrojovém C++ souboru vyexportovat do XML souboru. Knihovna je implementována jako rozšíření kompilátoru GCC. Původně byla vyvíjena pod názvem GCC_XML jako součást projektu ITK (Insight Segmentation and Registration Toolkit), takže požadavky na tento projekt byly kladeny výhradně touto firmou. Proto zde chybí například podpora pro parsování některých C++ konstrukcí jako je šablonování (template) (King, 2017).

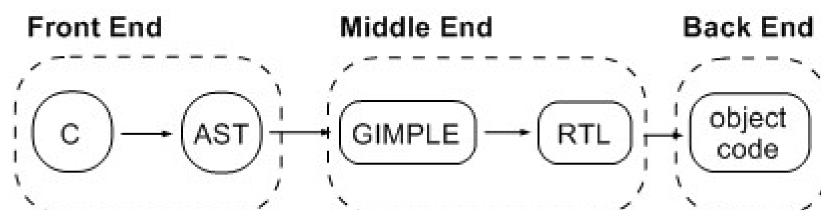
Knihovna podporuje standard C++11 a pro instalaci je vyžadován překladač LLVM/Clang. Pro převod zdrojového kódu na XML soubor stačí použít příkaz:

/castxml rpl_testobject.h –o výsledek.xml

Bohužel pro vygenerování XML souboru je nutné, aby vybraný soubor šel z daného umístění zkompilovat. Každý replikovaný objekt dědí z nějaké nadtřídy a ta dědí z dalších tříd atd. Pokud tedy nejde soubor zkompilovat, žádný XML výsledek bohužel vygenerován nebude a převod skončí chybou.

3.2 Clang

Je to front-end překladač, který má za úkol zdrojový kód převést na abstraktní syntaktický strom (AST) a ten poté převést na RTL (Register Transfer Language) a ten je pak předán back-end překladači např. LLVM, který vytvoří binární kód pro konkrétní platformu (Obrázek 20).



Obrázek 20 - Předklad zdrojového kódu, front-end a back-end překladačem

zdroj: https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture_4_1

Clang tedy umožňuje parsovat zdrojový kód, takže by mohl posloužit pro analýzu třídy replikovaného objektu. Bohužel výstup z tohoto parseru v podobě abstraktního syntaktického stromu je poněkud syrový a těžko čitelný.

Ukázka AST - http://clang.llvm.org/docs/IntroductionToTheClangAST.html

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to
pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int'
<LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar
0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10
'result' 'int'
```

3.3 Doxygen

Doxygen je nástroj, který dokáže na základě speciální struktury komentářů dodržované programátorem vygenerovat dokumentaci. Tímto způsobem lze popsat všechny funkce, třídy atd. v komentářích přímo ve zdrojovém kódu. Dokumentaci poté stačí vygenerovat. Doxygen je velmi rozšířený a funguje na operačních systémech Windows, Linux i Mac OS. Na Linuxových distribucích bývá často implicitně předinstalovaný.

Doxygen neumí generovat pouze HTML, RTF či PDF dokumentaci, ale umí tímto způsobem vygenerovat i XML soubor. Doxygen při generování dokumentace neshlukuje pouze

komentáře, nýbrž musí rozumět i zdrojovému kódu, jelikož výsledně vygenerovaná dokumentace obsahuje i seznam parametrů, názvy funkcí z daného okomentovaného řádku kódu.

Doxygen umožňuje výstup ve formátu XML ovlivnit velkým množstvím parametrů, které obvykle mají vliv na rozsah výstupu v XML souboru. Velkou výhodou je také vlastnost, že vstupem pro Doxygen může být samotný soubor bez vazeb na další závislosti a lze tak vygenerovat XML výstup i ze zdrojového C++ souboru, který je samostatně nezkompilovatelný. Následující ukázka obsahuje výstup ve formátu XML funkce setBoolean replikovatelného objektu.

RETIA, a.s. - librpl: rpl_testobject.cpp

```
// Funkce setBoolean z rpl_testobject.h
    void setBoolean(bool boolean)
    {
        if (boolean_ != boolean)
        {
            boolean_ = boolean;
            notify(ABoolean, ASet) << boolean_;
        }
    }

// Výstup XML z programu Doxygen
<memberdef kind="function"
id="classrpl_1_1TestObject_1af6565e680076be131c4b5e84b2c33a33"
prot="public" static="no" const="no" explicit="no" inline="yes"
virt="non-virtual">
        <type>void</type>
        <definition>void rpl::TestObject::setBoolean</definition>
        <argsstring>(bool boolean)</argsstring>
        <name>setBoolean</name>
        <param>
          <type>bool</type>
          <declname>boolean</declname>
        </param>
</memberdef>
```

Výstup ve formátu XML obsahuje i modifikátory funkcí, tzn., zda jsou veřejné, chráněné či neveřejné. U parametrů funkce může být také uvedeno, zda je parametr povinný či nikoliv. V celém XML výstupu jsou odkazy i na ostatní případně vygenerované XML soubory a lze tedy využít závislosti tak, aby bylo možné najít třeba metody podporované nadtřídou současné třídy.

3.4 Vlastní analýza

Parsování C++ kódu třídy není triviální záležitostí. Pokud však bude vstupem vždy třída v předepsaném formátu a budou se lišit pouze názvy metod, parametrů a počty, tak lze vytvořit vlastní parser, který tuto třídu zpracuje. Velký problém však nastává u složitějších konstrukcí, které mohou nastat, jako jsou například makra. Bylo by tedy nutné implementovat i preprocesor. Dále může třída obsahovat i vnitřní třídu nebo strukturu. Samotné metody mohou být také šablonované (template).

Dalším významný problém spočívá v tom, že jazyk C++ je kontextově závislý. Pokud při parsování přijde na řadu řádek `AA BB(CC);` není jednoznačné, zda je zakládán objekt typu `AA` s parametrem `CC`, anebo jde o prototyp funkce s návratovou hodnotou `AA` a názvem `BB`. Je tedy nutné vyhodnotit i kontext v jakém se daný řádek nachází (Kreinin, 2009).

3.5 Výběr C++ parseru pro diplomovou práci

Kvůli široké podpoře programu Doxygen napříč Linuxovými distribucemi byl pro vývoj ladícího nástroje zvolen právě tento parser C++ kódu. S ohledem na velkou popularitu a rozšíření tohoto programu lze očekávat podporu nových standardů C++ a zajistit tak dlouhou živostnost i ladícímu nástroji. Na rozdíl od řešení CastXML je zde určitá jistota podpory nových C++ standardů a především CastXML neumožňuje provést výstup do formátu XML ze zdrojového C++ kódu, který nelze bez závislostí samostatně zkompilovat.

4 NÁVRH A IMPLEMENTACE APLIKACE RPL DEBUGGER

Aplikaci jsem se rozhodl vytvořit v oficiálním vývojovém prostředí QT Creator, které poskytuje pohodlný vývoj GUI aplikací založených právě na knihovnách QT. Vývoj probíhal na operačním systému OpenSuse 13.2, který je ve firmě RETIA, a.s. hojně používaný. Instalace programu je vytvořena pro nástroj Autotools, který při instalaci dokáže zkontrolovat přítomnost nutných knihoven nezávisle na použité linuxové distribuci. Dále dokáže spravovat nespočet parametrů ovlivňující samotnou instalaci a jedním z nich je vlastně vytvořený přepínač --with-libqt4, který vynutí použití starší verze knihovny QT4 (aktuální verze je QT5). Ladící nástroj (odteď RplDebugger) je tedy kompatibilní i se staršími knihovnami QT, které se stále vyskytují na značné části počítačů ve firmě RETIA, a.s.

Ladící nástroj – rpl_debugger (Autotools – konfigurační soubor): configure.ac

```
# libqt4
AC_ARG_WITH([libqt4], [  --with-libqt4       compile against qt4 ],
[ cfg_libqt4="--with-libqt4=$withval"; USEQT4 =  $withval], [])
AM_CONDITIONAL([RPL_LIBQT4], test "$cfg_libqt4" = "--with-libqt4=yes")

AC_DEFINE([QMAKE_VERSION],[0],[QMAKE neexistuje])
AC_CHECK_PROG([QMAKEVERZE5],[qmake-qt5],[yes],[no])
AC_CHECK_PROG([QMAKEVERZE4],[qmake],[yes],[no])

if test x"$QMAKEVERZE4" == x"no" && test x"$QMAKEVERZE5" == x"no" ;
then
    AC_MSG_ERROR([QMAKE and QMAKE-QT5 are missing!])
fi

if test x"$QMAKEVERZE4" == x"yes" ; then
    AC_DEFINE([QMAKE_VERSION],[4],[QMAKE 4])
fi


if test x"$QMAKEVERZE5" == x"yes" ; then
AM_COND_IF([RPL_LIBQT4],[], AC_DEFINE(QMAKE_VERSION, 5,[QMAKE 5]) )

fi
```

4.1 Koncepce aplikace

RplDebugger má za úkol plnit celkem dvě podstatné úlohy. Prvním úkolem RplDebuggeru je zobrazení objektů replikačního serveru tak, aby uživatel/programátor s nimi mohl přes rozhraní GUI pracovat. Druhým úkolem ladícího nástroje je vytvoření pluginu pro přidání podpory zobrazování GUI u metod pro nový objekt.

4.1.1 Získání objektů z replikačního serveru

Po spuštění nového okna aplikace je inicializován objekt replikační služby, který implementuje všechny metody jako `onAdd`, `onChange`, `onRemove` apod. Dále je nad replikační službou zavolána metoda `registerClass`, ve které lze specifikovat filtr objektů. U každé klientské aplikace lze specifikovat, které objekty chce z replikační sítě přijímat. Pokud je vyplněna hvězdička „*“, znamená to, že replikační služba bude z replikačního serveru zasílat všechny dostupné objekty.

Ladící nástroj – rpl_debugger: deb_mainwindow.cpp

```
    .................
    .................
MainWindow::MainWindow(bool verbose, QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
   .................
   .................
 //Rpl service
 rplService_ = new DebRplService();
 rplService_->registerClass("*");

//List of objects
objectListWidget_ = new DebObjectListWidget(ui->treeWidgetObjectList,
rplService_, ui->lineEditFilter);
objectListWidget_->setAutoUpdate(ui->checkBoxAutoUpdate->isChecked());

   .................
   .................
```

Zároveň je vytvořen objekt typu `DebObjectListWidget`, který obsluhuje celou GUI komponentu `QTreeWidget`, ve které je zobrazován aktuální seznam všech dostupným objektů v replikační síti.

Knihovna QT umožňuje použít konstrukci signálů a slotů. Každá třída obsahující makro `Q_OBJECT` umožňuje vytvářet tyto dvě speciální metody. První druh metody se nazývá `slot`. Tato metoda obsahuje tělo a je zavolána při výskytu konkrétního signálu. Druhým typem metody je `signal`. Je to metoda bez těla, kterou lze zavolat klíčovým slovem `emit`. Vždy je nutné svázat konkrétní metodu `signal` s konkrétní metodou slot. Zároveň musí obsahovat stejný počet a typ parametrů, který lze předávat. Tímto způsobem lze svázat nový objekt `rplService_` a `objectListWidget`. Pokud objekt replikační služby detekuje přidání či odebrání objektu v replikační síti, dá vědět objektu `objectListWidget` a ten aktualizuje svůj seznam objektů v GUI.

Ladící nástroj – rpl_debugger: deb_mainwindow.cpp

```
    .................
    .................
//Refresh list of objects when object added or removed
QObject::connect(rplService_, SIGNAL(addSignal(rpl::Object *)),
objectListWidget_, SLOT(refresh(rpl::Object *)));

QObject::connect(rplService_, SIGNAL(removeSignal(rpl::Object *)),
objectListWidget_, SLOT(refresh(rpl::Object *)));

   .................
   .................
```

Z výše uvedeného kódu je zřejmé, že jakmile objekt `rplService_` emituje uvnitř některé ze svých metod signál s názvem `addSignal` nebo `removeSignal` je předán objekt `rpl::Objekt` do metody `refresh` u objektu `objectListWidget_`, který spravuje seznam objektů v GUI.

V oddílu 4.2 je popsán způsob, jakým je RplDebugger propojen s replikační službou firmy RETIA, a.s. Aby mohla replikační služba komunikovat s okolními uzly, musí být implementován komunikační kanál, který bude na tuto replikační službu navázán opět pomocí signálů a slotů. Po stisknutí tlačítka „Connect“ je vyvolána metoda `connectToRplServer`, která tuto akci spustí.

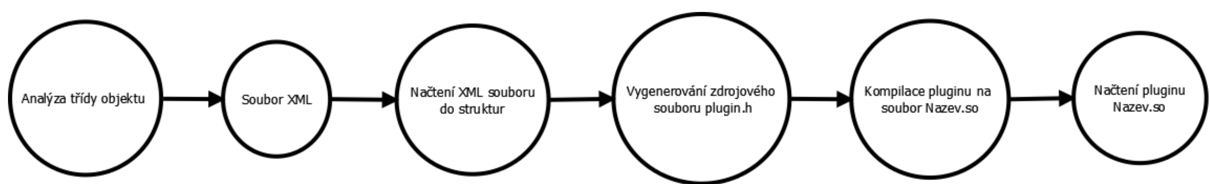
Ladící nástroj – rpl_debugger: deb_mainwindow.cpp

```
.................
.................
void MainWindow::connectToRplServer(bool)
{

.................
.................

debClient_ = new Client("Klient", lineEditIP->text(),
lineEditPort->text().toInt());

//Link TCP debClient with RplService
QObject::connect(&(debClient_->getClient()), SIGNAL(connected()),
rplService_, SLOT(connectSlot()));

QObject::connect(&(debClient_->getClient()), SIGNAL(disconnected()),
rplService_, SLOT(disconnectSlot()));

QObject::connect(debClient_, SIGNAL(readSignal(QByteArray)), rplService_,
SLOT(recieveSlot(QByteArray)));

QObject::connect(rplService_, SIGNAL(sendSignal(const uint8_t *,
uint32_t)), debClient_, SLOT(sendSlot(const uint8_t *, uint32_t)));

.................
.................

//TCP client signal slots for GUI events
QObject::connect(debClient_, SIGNAL(connectedSig(Client *)), this,
SLOT(connectedToRplServer(Client *)));

QObject::connect(debClient_, SIGNAL(disconnectedSig(Client *)), this,
SLOT(disconnectedFromRplServer(Client *)));

QObject::connect(debClient_, SIGNAL(connectErrorSig(Client *)), this,
SLOT(connectErrorRplServer(Client *)));

connectButton->setEnabled(false);
//Start TCP debClient
debClient_->start();

}
   .................
   .................
```

Pro připojení na replikační server je použita IP adresa z formuláře GUI v hlavním okně `lineEditIP->text()`. Pokud je potřeba připojit se na spuštěný server, který se nachází na stejném počítači, jednoduše lze použít adresu 127.0.0.1.

4.1.2 Vytvoření pluginu pro podporu GUI u metod nového replikovaného objektu

Jakmile funguje propojení s replikační službou a připojení k replikačnímu serveru, nic nebrání tomu, aby RplDebugger přijímal nové objekty a zobrazoval je v seznamu pro uživatele. Hlavní cíl této diplomové práce spočívá v zobrazování prvků GUI u metod jednotlivých objektů. Aby bylo možné prvky GUI vygenerovat, je nutné nejdříve třídu analyzovat, poté vytvořit zdrojový kód, ve kterém budou GUI prvky provázány s voláním metod objektu a poté tento zdrojový kód zkompilovat do podoby dynamicky načítaného pluginu (Obrázek 21). Jakmile uživatel vybere nějaký konkrétní objekt ze seznamu, je načten pouze příslušný plugin, který vytváří samotné GUI.



Obrázek 21 - Koncepce přidání podpory pro nový objekt replikačního serveru

zdroj: vlastní

Pro načtení zdrojového souboru třídy je využita pomocná třída `QFileDialog` a její metoda `getOpenFileNames`, kterou je možné omezit pouze na soubory s danou příponou. V případě RplDebuggeru jsou to soubory typu `.h`, `.cpp`, `.hpp` či `.c`. Souborů lze vybrat více najednou. RplDebugger pro efektivní práci podporuje více vláken. Pakliže uživatel vybere více souborů najednou, ty mohou být zpracovány paralelně pomocí modulu `QtConcurrent`, který spustí více úloh najednou na ideální počet vláken (tento počet vrací metoda `QThread::idealThreadCount()`). V praxi bude spuštěno na 8 jádrovém procesoru maximálně 8 vláken a může být zpracováno maximálně 8 souborů v jeden okamžik. Při testování aplikace se tato metoda jeví jako velmi efektivní, jelikož bez této metody by mohla vzniknout situace, kdy kompilátor GCC je spuštěn několikrát a kompilace 5 pluginů najednou na dvoujádrovém procesoru naprosto zahltí prostředky počítače.

Ladící nástroj – rpl_debugger: deb_mainwindow.cpp

```
    .................
    .................
void MainWindow::buildPluginDialog()
{

    .................
    .................
  // Files dialog
  QStringList files = QFileDialog::getOpenFileNames(
                           this,
                           "Select one or more files to open",
                           headersDirectory.absolutePath(),
                           "C++ source files (*.h *.cpp *.hpp *.c)");

  if (files.size() > 0)
   {
    .................
    .................
    // Objekt pro vytváření vláken
    QFutureWatcher<void> watcher;
    .................
    .................
     // For every file in files list will be created thread in
    buildPlugin method
    watcher.setFuture(QtConcurrent::map(files,
    std::bind(&MainWindow::buildPlugin,
    this,
    std::placeholders::_1)));

    buildingProgressDialog_->exec();

     // Wait for finishing all threads
    watcher.waitForFinished();

   }
    .................
    .................

}
   .................
   .................
```

Pro každé vlákno je spuštěna metoda `buildPlugin`, ve které je založen pro každý zdrojový soubor třídy nový objekt typu `DebHeader`, který se postará o generování XML souborů pomocí Doxygenu.

Ladící nástroj – rpl_debugger: deb_mainwindow.cpp

```
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
 void MainWindow::buildPlugin(QString file)
{
    DebHeader *newHeader = new DebHeader(file);

    //Build all possible plugins based on header file
    newHeader->createPlugins(QString(PROJECT_DIR)
    + "/etc/deb_Doxyfile", qmake_);
  . . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . . .
}
. . . . . . . . . . . . . . . . .
```

Mezi klíčovou metodou třídy `DebHeader` patří metoda `generateXmlFiles`. V této metodě je pomocí knihovny QT vytvořen objekt `QProcess`, jež dokáže spustit proces zadaný v podobě řetězce stejně, jako by tomu bylo možné v příkazové řádce. U každého zdrojového souboru třídy je pro program Dogygen upraven konfigurační soubor, ve kterém je přidán parametr `INPUT` (cesta k souboru třídy) a `XML_OUTPUT` ve kterém je cesta do dočasného adresáře v systémové složce /tmp. Výstup XML bude uložen právě do dočasné složky a po zkompilování pluginu budou tyto dočasné soubory nakonec smazány.

Ladící nástroj – rpl_debugger: deb_header.cpp

```
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
void DebHeader::generateXmlFiles(QString doxyConfigPath)
{
    //Generate XML files based on the source header file
    QProcess *process = new QProcess();

    xmlDirPath_ = tmpDirForXML_ + ""
    + QFileInfo(QFile(sourceHeader_).fileName()).fileName() + "_"
    + uniqueId_;

    xmlDirPath_.replace(".", "_");

    //Process command
    QString command = " ( cat " + doxyConfigPath + " ; echo \"INPUT="
    + sourceHeader_ + "\" ; echo  \"XML_OUTPUT = " + xmlDirPath_
    + "\")  | doxygen -";

    process->start("bash", QStringList() << "-c" << command);
    process->waitForFinished(-1);

. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .

}
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
```

Ukázka výstupu XML ze třídy `rpl_testobject` je vidět v oddílu 3.3.

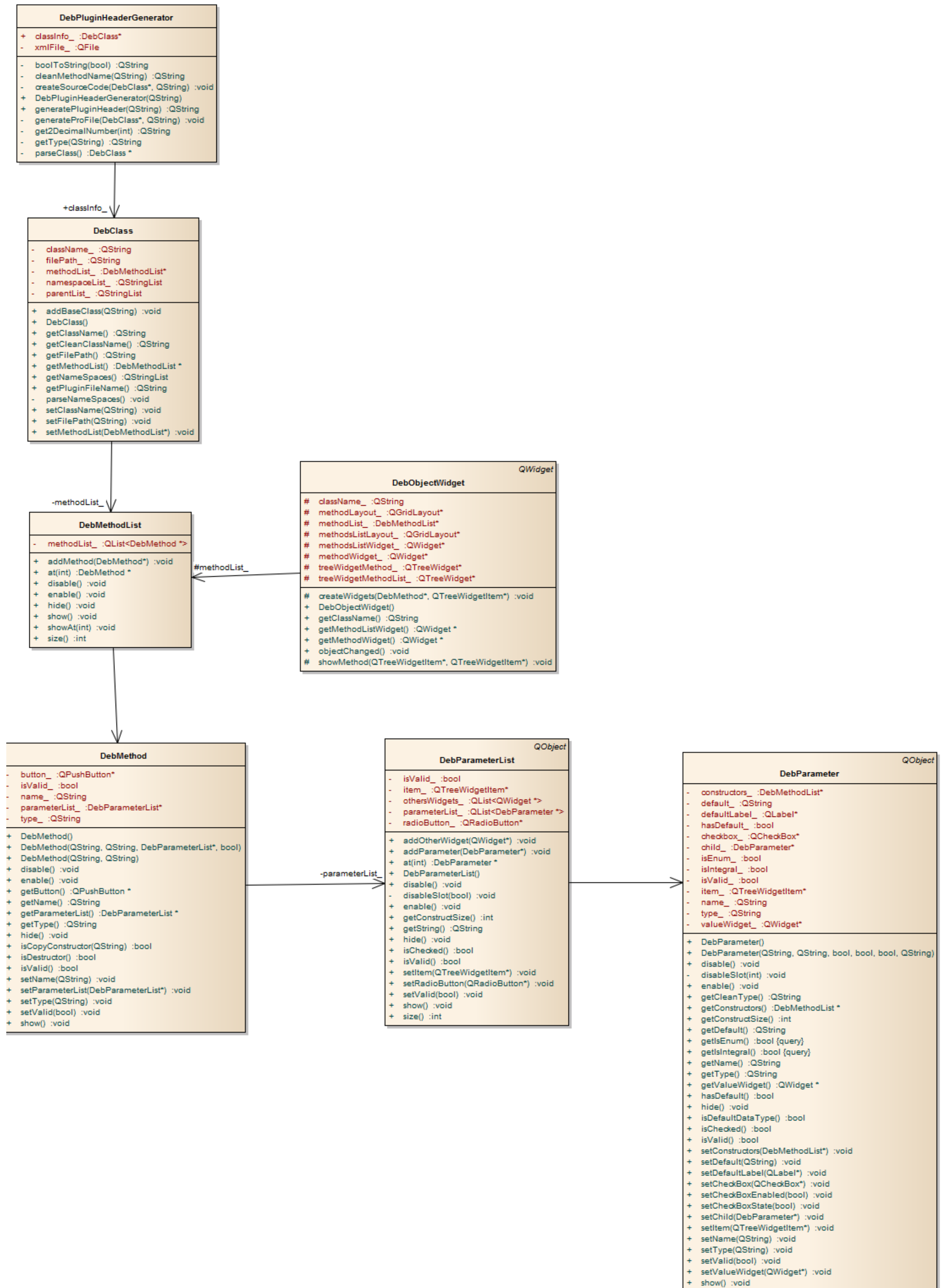
Jakmile jsou vytvořeny všechny adresáře pro každou ze tříd, které obsahují XML soubory (pokud jeden hlavičkový soubor obsahuje více tříd, je ve složce více XML souborů), je možné na základě těchto souborů vygenerovat zdrojové soubory pluginu. Ve třídě `DebHeader` v metodě `generatePlugin()` je pro každý XML soubor vytvořen objekt typu `DebPluginGenerator`, který má za úkol načítat již hotové XML soubory a vygenerovat z nich třídu do souboru `plugin.h`.

Pro čtení XML souborů slouží ve třídě `DebPluginHeaderGenerator` metoda `parseClass`, která vytváří objekt třídy `DebClass`. Objekt `DebClass` obsahuje seznam metod. Každá metoda pak obsahuje seznam parametrů. Celá tato hierarchie je vidět v diagramu tříd (Obrázek 22). Celá struktura v paměti pak v podstatě obsahuje vše podstatné, co popisuje vygenerovaný XML soubor. Na základě této struktury lze nyní vygenerovat zdrojový soubor pluginu.

Stěžejní metoda celého RplDebuggeru se nazývá `createSourceCode` ve třídě DebPluginHeaderGenerator. Tato metoda v podstatě prochází celou strukturu objektu `DebClass` založenou na informacích z XML souboru a na základě ní vygeneruje C++ třídu (Příloha D).

Ladící nástroj – rpl_debugger: deb_plugin_header_generator.cpp

```
.................
//Create plugin.h header of plugin based on class structure
void DebPluginHeaderGenerator::createSourceCode(DebClass *newClass,
QString tmpDir)
{
  .................
  QTextStream stream(&file);

  stream << getFileContent(":/gencontent/gen_content/01includes.txt");
  stream << endl << "#include \"rpl/" << newClass->getFilePath()<<"\"";

//Generate namespaces
  QStringList namespaceList = newClass->getNameSpaces();
  for (int i = 0; i < namespaceList.size(); ++i)
  {
  stream << endl << "namespace " << namespaceList.at(i);
  stream << endl << "{";
  }

 //Create plugin class
  stream << endl << "class ObjectWidget : public DebObjectWidget,public "
  << newClass->getClassName();
  stream << endl << "{";
  .................
```



Obrázek 22 - Diagram tříd pro uložení hierarchie třídy na základě XML souboru

zdroj: vlastní

Metoda `createSourceCode` ve skutečnosti vygeneruje dohromady 2 třídy do souboru `plugin.h`. První třída s názvem `ObjectWidget` vytváří samotné GUI prvky jako tlačítka, textboxy apod. a zabírá téměř celý soubor `plugin.h`. Druhá třída s názvem `Plugin1` implementuje rozhraní třídy `DebPluginInterface` a vrací objekt třídy `ObjectWidget`. Toto řešení dovoluje vytvářet GUI prvky metod vícekrát. RplDebugger podporuje editaci replikovaných objektů v samostatných oknech, kterých si lze otevřít více najednou. Pro každé okno je potřeba vytvořit opět stejné GUI prvky, a jelikož lze stejný plugin načíst do aplikace pouze jednou, je toto řešení opakovaného vytváření objektu `ObjectWidget` nejvhodnější.

V případě načítání pluginu RplDebuggerem je nutné znát typ objektu, který dynamická knihovna vytvoří, dopředu. Z tohoto důvodu je tu rozhraní `DebPluginInterface`.

Ladící nástroj – rpl_debugger: deb_plugin_interface.h

```
.................
.................
class DebPluginInterface  : public QObject
{

    Q_OBJECT

public:
    virtual ~DebPluginInterface() { }
    virtual DebObjectWidget *getObjectWidget(void *object) = 0;
    virtual DebMethodList *getConstructorParametersList() = 0;
    virtual void *Constructor(DebParameter *methodTree) = 0;

};


#define DebPluginInterface_iid "org.qt-
project.Qt.Retia.DebPluginInterface"

Q_DECLARE_INTERFACE(DebPluginInterface, DebPluginInterface_iid)

  .................
  .................
```

Tento interface obsahuje celkem 3 metody. Metoda `getObjectWidget` je stěžejní a vrací objekt, ve kterém jsou všechny GUI prvky napojené na replikovaný objekt. Tento celek stačí vložit do RplDebuggeru jako samostatnou GUI komponentu. Druhá metoda `getConstructorParameterList` vrací seznam metod a parametrů nutných pro konstrukci replikovaného objektu. Tzn., pokud má objekt konstruktor pouze jeden, je v seznamu metod pouze jediná metoda. Metoda `Constructor` vrací již zkonstruovaný objekt třídy replikovaného objektu. Tyto poslední dvě metody slouží pro konstrukci objektů. Pokud se

u replikovaného objektu vyskytne metoda, která vyžaduje parametr objektu nezákladního datového typu (např. objekt typu NovyObjekt), tak RplDebugger se na základě názvu „NovyObjekt“ tohoto neznámého objektu snaží nalézt existující plugin a když ho najde, vyžádá si seznam parametrů pro konstrukci. Tímto způsobem se lze rekurzivně zanořovat do té doby, dokud RplDebugger nenarazí na základní datový typ.

Jakmile je plugin hotov, může být zkompilován. Specialitou knihovny QT je metakompilátor. Pro zkompilování projektu QT metakompilátorem není potřeba Makefile, ale soubor s příponou .pro (viz oddíl 4.3), kde jsou uloženy všechny přepínače, knihovny apod.

V praxi pro kompilaci stačí zdrojový soubor pluginu - plugin.h a konfigurační soubor např. gen_plugin.pro. Tyto soubory se zkompilují nejdříve metakompilátorem pomocí příkazu qmake a poté se zkompilují např. GCC kompilátorem pomocí příkazu make. RplDebugger tuto činnost provádí v metodě buildPluginHeaders třídy DebHeader.

Ladící nástroj – rpl_debugger: deb_header.cpp

```
void DebHeader::buildPluginHeaders(QString qmake)
{

    for (int i = 0; i < pluginList_.size(); ++i)
    {
        QProcess *process = new QProcess();

        process->setWorkingDirectory(
        pluginList_.at(i)->pluginHeaderDir);
      //Make commands
        process->start("bash", QStringList() << "-c"
        << QString(qmake + " 2>&1 ; make 2>&1"));

        process->waitForFinished(-1);
        pluginList_.at(i)->makeExitCode = process->exitCode();
        pluginList_.at(i)->makeOutput = process->readAll();
    }

}
```

Jelikož je příkaz qmake různý u QT verze 4 („qmake“) než u QT verze 5 („qmake-qt5“) je do metody buildPluginHeaders posílán tento název jako parametr.

Jakmile je plugin zkompilován a uživatel klikne na objekt např. s názvem „NovyObjekt“ je dle tohoto názvu dohledán plugin obsahující toto jméno. Pro načtení pluginu do RplDebuggeru slouží funkce loadPluginObject ze souboru deb_plugin_enums.h (kde jsou pomocné funkce využívané ve více třídách). Pro samotné načtení pluginu je využita QT třída QPluginLoader.

Ladící nástroj – rpl_debugger: deb_plugin_enums.

```
inline DebPluginInterface *loadPluginObject(QString fileName)
{
    DebPluginInterface *returnObject = NULL;
    QFile *qfile = new QFile(fileName);
    if (qfile->exists())
    {
        //Object to load plugin
        QPluginLoader loader(fileName);

        //Create plugin object instance by its constructor
        QObject *plugin = loader.instance();

        if (plugin)
        {
           //Cast plugin object to interface object
            returnObject = qobject_cast<DebPluginInterface *>(plugin);
        }
    }
    return returnObject;
}
```

4.2 Rozhraní pro práci s replikačním serverem

V první části implementace RplDebuggeru je potřeba, připojit se na existující replikační server, který běží na stejném počítači. Pro komunikační protokol zpráv s tímto serverem slouží replikační služba `rpl::Service`, která se stará o logiku samotné replikační sítě. Třída `rpl::Service` poskytuje rozhraní (Příloha C), ve kterém se nacházejí metody, jež reprezentují události v replikační síti. Z toho rozhraní je tedy nutné vydědit vlastní třídu, ve které budou obslouženy události přicházející z replikační sítě.

Mezi metody rozhraní `rpl::Service`, které implementuje RplDebugger, patří především metoda `onAdd`, která je volána replikační službou v případě, že se v replikační síti vyskytl nový objekt. RplDebugger si tento nový objekt (ukazatel) uloží do seznamu, aby s tímto objektem mohl kdykoliv později pracovat.

Ladící nástroj – rpl_debugger: deb_rpl_service.h

```
    .................
    .................
#include <rpl/rpl_service.h>
#include "rpl/rpl_object.h"

class DebRplService : public QObject, public rpl::Service
{
    Q_OBJECT

private:

    QMap<rpl::Object *, QList<QStringList>*> objectList;
    .................
    .................
public:
    DebRplService();
    virtual bool onAdd(rpl::Object *object);
    virtual bool onChange(rpl::Object *object, rpl::AttributeUInt att);
    virtual bool onRemove(rpl::Object *object);
    virtual void send(const uint8_t *data, uint32_t bytes, void *clientPtr
= NULL);
   QMap<rpl::Object *, QList<QStringList>*> getObjectList();
   int getObjectListSize();
    .................
    .................
```

V případě, že bude objekt v replikační síti změněn, je vyvolána obslužná metoda `onChange`, ve které RplDebugger může zareagovat na změnu. Samozřejmě je objekt v té době již změněn a ukazatel, který byl uložen do seznamu při volání `onAdd` odkazuje vždy na aktuální stav objektu. Je to dáno tím, že objekt vytvořila replikační služba `rpl::Service`. Není tedy nutné měnit objekt ručně, pouze jsme informováni, že byl objekt změněn.

Poslední z událostí, jež RplDebugger obsluhuje, je metoda `onRemove`. Tato metoda je volána vždy, když je nějaký objekt z replikační sítě smazán, případě bylo ztraceno spojení s vlastníkem tohoto objektu. RplDebugger tento objekt jednoduše smaže ze svého seznamu `objectList`.

V případě, že nasdílíme vlastní objekt do replikační služby, je potřeba implementovat metodu `send`. RplDebugger pomocí této metody přijatá data touto metodou odešle pomocí vlastního spojení do replikačního serveru.

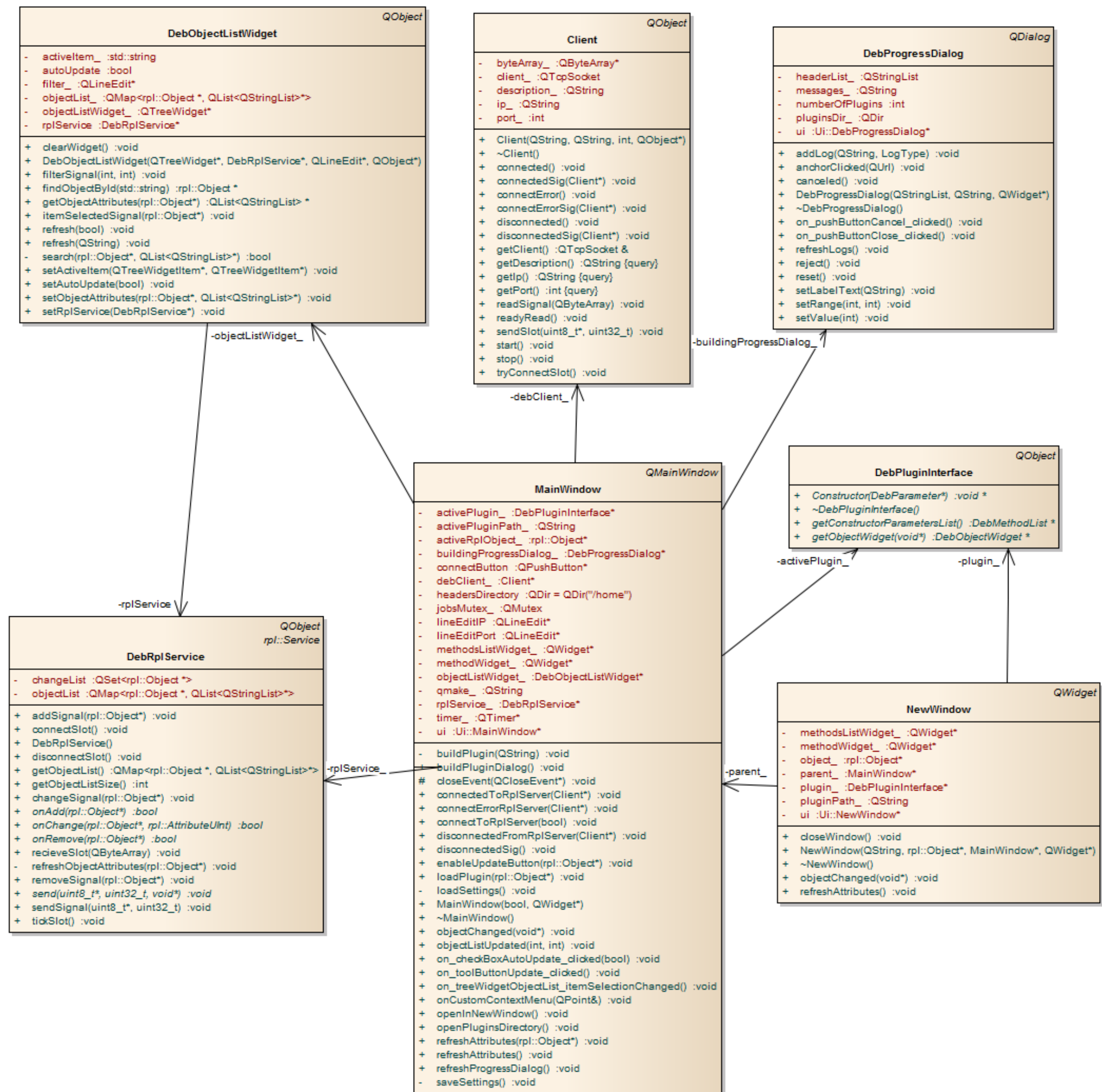
Rozhraní služby `rpl::Service` (Příloha C) obsahuje mnohem více metod, které je možné implementovat. V rámci aplikace ladícího nástroje však stačí kontrolovat tok objektů v replikační síti a schopnost odesílat změny do této sítě.

4.3 Struktura aplikace

Celou aplikaci tvoří několik tříd. Mezi ty nejdůležitější patří třída hlavního okna `MainWindow`, které vzniká ihned při spuštění aplikace v souboru `main.cpp`. Hlavní okno MainWindow má životnost po celou dobu běhu aplikace a obsahuje ty nejdůležitější struktury (Obrázek 23).

Ladící nástroj – rpl_debugger: main.cpp

```
.................
.................
int main(int argc, char *argv[])
{
   .................
   .................
    QApplication a(argc, argv);
    MainWindow w(verbose);
    w.show();
    return a.exec();
}
```

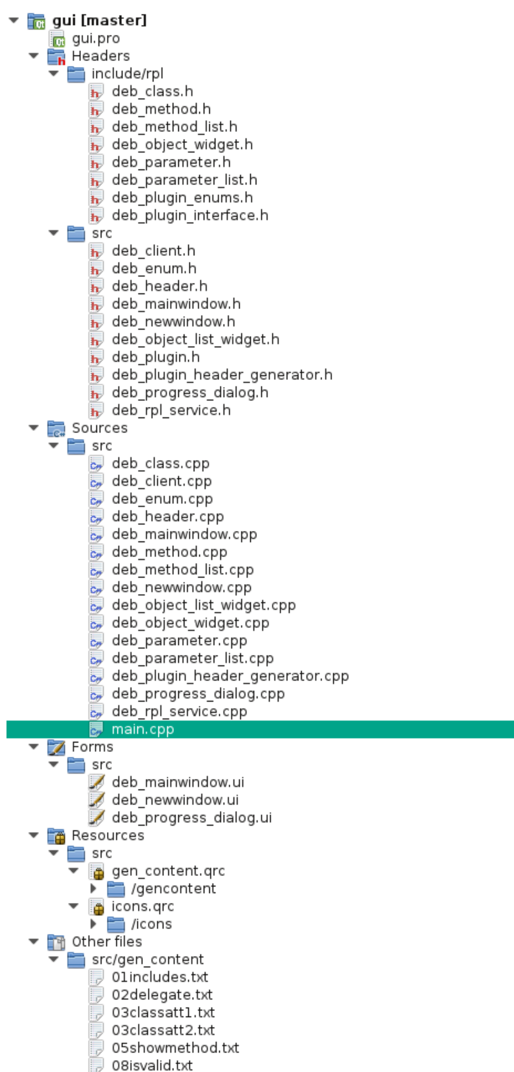


Obrázek 23 - Diagram tříd pro hlavní okno aplikace MainWindow

zdroj: vlastní

Zde je nutné podotknout, že diagram tříd u projektu založeném na knihovnách QT nezachycuje různá propojení objektů mezi sebou pomocí signálů a slotů. Těchto propojení je nesčetně a obvykle jde o reakci na nějakou událost, která se vyskytla. Obecně řečeno hlavní třída `MainWindow` se stará o celý životní cyklus objektu replikační služby `DebRplService`, dále se stará o síťové TCP spojení (`DebClient`) na replikační server, také spravuje aktuální seznam objektů (`DebObjectListWidget`) a umožňuje vytvářet nová okna (`NewWindow`) pro práci s více replikovanými objekty najednou. Při kompilaci pluginů se hlavní okno `MainWindow` stará o zobrazování indikátoru průběhu kompilace (`DebProgressDialog`).

Celý projekt v prostředí IDE QT Creator je rozdělen na několik částí (Obrázek 24). Celkem se projekt skládá z konfiguračního souboru `.pro`, zdrojových souborů `.h` a `.cpp`, dále se skládá z formulářových souborů typu `.ui` a ze zdrojových dat typu `.qrc`.



Obrázek 24 - Struktura projektu celé aplikace

zdroj: vlastní

4.3.1 Konfigurační soubor

Soubor typu `.pro` je konfiguranční soubor, přes který lze v QT Creatoru otevřít projekt. V podstatě obsahuje informace nutné pro kompilaci programu stejně jako u souboru typu `Makefile`. Metakompilátor QT na základě těchto informací vytvoří skutečný `Makefile`, který lze pak použít pro kompilaci např. kompilátorem GCC. Následující ukázka se týká konfiguračního souboru pluginu. Pro dynamicky načítané knihovny musí být vždy uveden přepínač `rdynamic`.

Ladící nástroj – rpl_debugger: gen _plugin.pro

```
QT        += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

QMAKE_CXXFLAGS += -std=c++11
CONFIG += plugin
TARGET = plugin dynamic
TEMPLATE = lib
DEFINES += TEST
QMAKE_LFLAGS+=-pie -rdynamic
SOURCES +=


LIBS += -L/home/user/Retia/build/lib64 -lrpl
INCLUDEPATH += /home/user/Retia/build/include
TARGET  = "class_rpl-TestObject"#$$qtLibraryTarget(plugin1)
DESTDIR = /home/user/Retia/build/lib64/rpl/deb
HEADERS += \
   plugin.h
 # install
 target.path = /home/user/Retia/build/lib64/rpl/deb
 INSTALLS += target
```

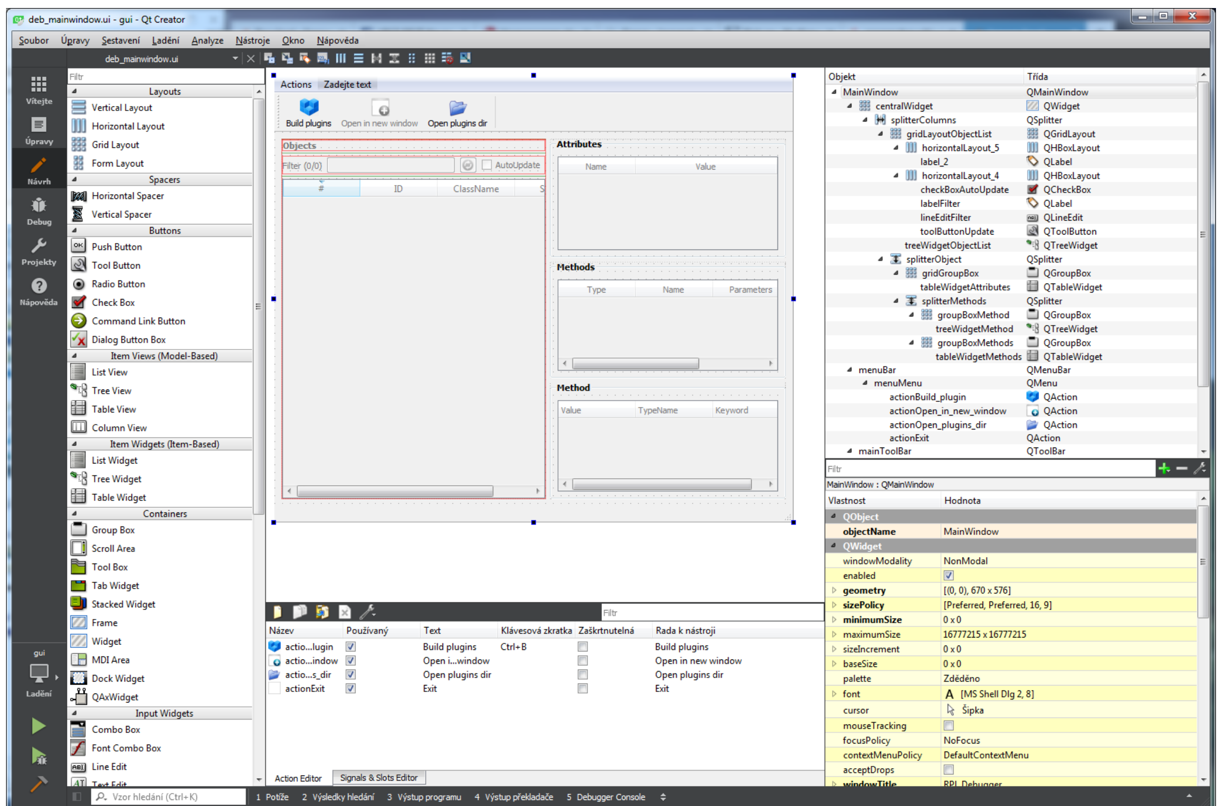
4.3.2 Zdrojové kódy

Druhým typem souborů jsou hlavičky a zdroje. V podstatě se jedná o hlavičkové soubory typu `.h` a zdrojové soubory typu `.cpp`. Lze si všimnou (Obrázek 24), že část hlavičkových souborů se nachází v podadresáři `src` a část v podadresáři `include/rpl`. Tyto hlavičkové soubory jsou takto rozděleny záměrně, jelikož při instalaci programu přes utilitu Autotools na linuxových distribucích jsou tyto hlavičkové soubory v podadresáři `include/rpl` nainstalovány do standardního adresáře v systému, aby je mohly využít i jiné programy. V tomto případě jsou tyto hlavičkové soubory potřeba pro kompilaci samotného pluginu pomocí RplDebuggeru.

4.3.3 Formuláře

RplDebugger obsahuje celkem 3 formuláře:

1. Hlavní okno (Obrázek 25)
2. Nové okno
3. Okno indikace kompilace



Obrázek 25 - Formulář hlavního okna z prostředí QT Creator

zdroj: vlastní

IDE QT Creator umožňuje spravovat veškeré GUI vestavěným designerem. Napravo je vždy vidět hierarchická struktura grafických komponent. Tyto `.ui` soubory jsou ukládány jako XML soubory, kde jsou specifikovány všechny parametry. Tento soubor je při kompilaci metakompilátorem knihovny QT převeden na zdrojové soubory `.h` a `.cpp`, kde je vidět konstrukce jednotlivých komponent.

Ladící nástroj – rpl_debugger: deb_mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  .................
  .................
  <property name="windowTitle">
   <string>RPL Debugger</string>
  </property>
  <property name="styleSheet">
   <string notr="true"/>
  </property>
  <widget class="QWidget" name="centralWidget">
   <layout class="QGridLayout" name="gridLayout_4">
    <item row="3" column="0">
    .................
```

4.3.4 Prostředky – zdrojová data aplikace

V případě, že je nutné do projektu zavést externí data jako text, obrázek nebo jiný druh dat, je možné vytvořit soubor typu `.qrc`, ve kterém se uveden seznam komponent, které mají být součástí výsledného binárního souboru. Výsledná aplikace se pak nemusí odkazovat na externí soubory, ale aplikace již tyto soubory obsahuje ve spustitelném binárním souboru. Tato vlastnost se může hodit v případě, že je potřeba znemožnit cizím osobám v modifikaci těchto dat. Na druhou stranu roste celková velikost spustitelného souboru. V praxi se u velkých aplikací (typicky her) používají externí binární soubory, jejichž formátu rozumí pouze samotná aplikace.

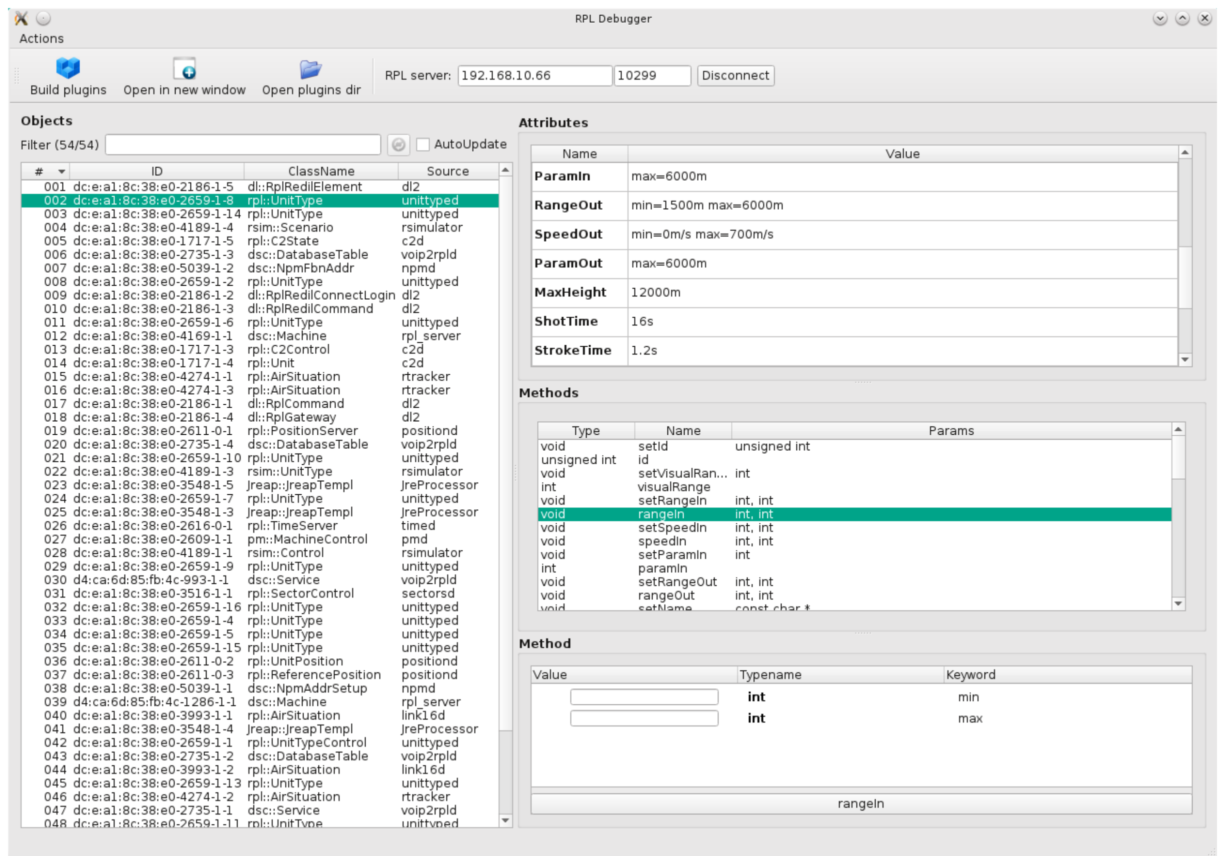
Spustitelný soubor RplDebuggeru obsahuje zkompilovaná data ikon a textových souborů. Textové soubory v RplDebuggeru obsahují často používané části zdrojového kódu pro generování pluginů (např. seznam `#include` notací).

Ladící nástroj – rpl_debugger: icons.qrc

```
<RCC>
    <qresource prefix="/icons">
        <file>icons/create.png</file>
        <file>icons/new-window.png</file>
        <file>icons/open_folder.png</file>
        <file>icons/failure.png</file>
        <file>icons/success.png</file>
        <file>icons/warning.png</file>
        <file>icons/update.png</file>
    </qresource>
</RCC>


//Načtení souboru ve zdrojovém souboru např.:
//getFileContent(":/gencontent/gen_content/01includes.txt");
```

4.4 Vzhled aplikace

RplDebugger je navržen jako ladící utilita, kde není kladen důraz na vzhled nýbrž na praktičnost. Hlavní okno aplikace je rozděleno na dvě poloviny (Obrázek 26). V levé části je zobrazen seznam objektů z replikačního serveru, kterým se implementačně zabýval pododdíl 4.1.1. V pravé polovině je zobrazena tabulka atributů, která je vyparsována z metody `print()` (obdoba `toString()`) u každého replikovaného objektu. Tato metoda je totiž pro všechny objekty povinná a formát je jednotný.



Obrázek 26 - Ukázka hlavního okna aplikace RplDebugger

zdroj: vlastní

Tabulkový seznam metod (Methods) je dílem načteného pluginu, který byl předtím vytvořen RplDebuggerem. Při vybrání konkrétní metody se v dolní části okna zobrazí formulář pro vybranou metodu včetně tlačítka pro zavolání příslušné metody.

V horní části aplikace jsou tlačítka pro vytváření pluginů, pro otevření nového okna, pro otevření složky s vytvořenými pluginy a formulář pro připojení k replikačnímu serveru.

Celé hlavní okno obsahuje celkem čtyři rozdělovače (splitter), které je možné libovolně zvětšovat a zmenšovat na úkor jiných grafických komponent. Aby se mohl uživatel pohodlně vrátit do stavu v jakým byla aplikace před posledním zavřením, je nastavení před každým ukončením aplikace uloženo do `.ini` souboru. Při spuštění je toto nastavení opět vyvoláno. Ukládány jsou především informace o stavu rozdělovníků, velikosti celého okna a také hodnota IP adresy a portu pro připojení na replikační server. Samotné uložení není třeba dělat ručně, je na to již připravená třída z knihovny QT s názvem `QSettings`.

Ladící nástroj – rpl_debugger: deb_mainwindow.cpp

```
<void MainWindow::saveSettings()
{
 QSettings settings("settings.ini", QSettings::IniFormat);
 settings.setValue("headersDirectory", headersDirectory.absolutePath());
 settings.setValue("mainWindowGeometry", saveGeometry());
 settings.setValue("splittlerColumns", ui->splitterColumns->saveState());
 settings.setValue("splittlerObject", ui->splitterObject->saveState());
 settings.setValue("splittlerMethods", ui->splitterMethods->saveState());
 settings.setValue("objectList", ui->treeWidgetObjectList->header()
->saveState());
 settings.setValue("rplserverip", lineEditIP->text());
 settings.setValue("rplserverport", lineEditPort->text());
}

void MainWindow::loadSettings()
{
 QSettings settings("settings.ini", QSettings::IniFormat);
 restoreGeometry(settings.value("mainWindowGeometry").toByteArray());
 headersDirectory.setPath(settings.value("headersDirectory").toString());
 ui->splitterColumns->restoreState(
 settings.value("splittlerColumns").toByteArray());
 ui->splitterObject->restoreState(
 settings.value("splittlerObject").toByteArray());
 ui->splitterMethods->restoreState(
 settings.value("splittlerMethods").toByteArray());
 ui->treeWidgetObjectList->header()->restoreState(
 settings.value("objectList").toByteArray());
 lineEditIP->setText(settings.value("rplserverip").toString());
 lineEditPort->setText(settings.value("rplserverport").toString());
}
```

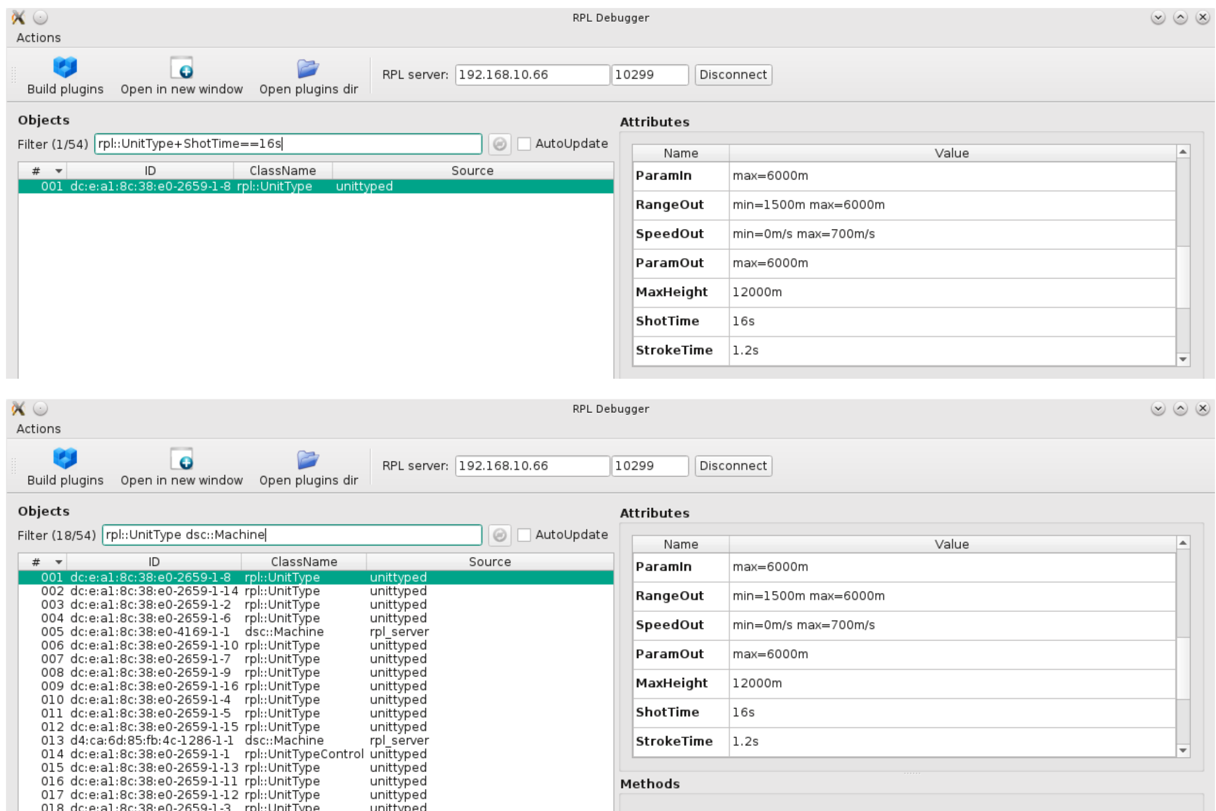
4.5 Funkce aplikace

Kromě zobrazování objektů z replikačního serveru a přidávání podpory pro další objekty obsahuje RplDebugger další užitečné funkce, které zpříjemní programátorovi práci s tímto programem.

4.5.1 Filtrování zobrazených objektů

RplDebugger podporuje filtrování seznamu objektů (Obrázek 27). Jelikož replikační síť může obsahovat stovky entit, programátor zúžení výběru jistě ocení. Filtrování objektů zahrnuje několik operátorů:

1. „ “ (mezera) – sjednocení hledaných hodnot
2. + průnik hledaných hodnot
3. == rovná se hodnotě atributu objektu
4. != nerovná se hodnotě atributu objektu
5. ~= vyskytuje se v hodnotě atributu objektu



Obrázek 27 - Ukázky filtrování seznamu replikovaných objektů

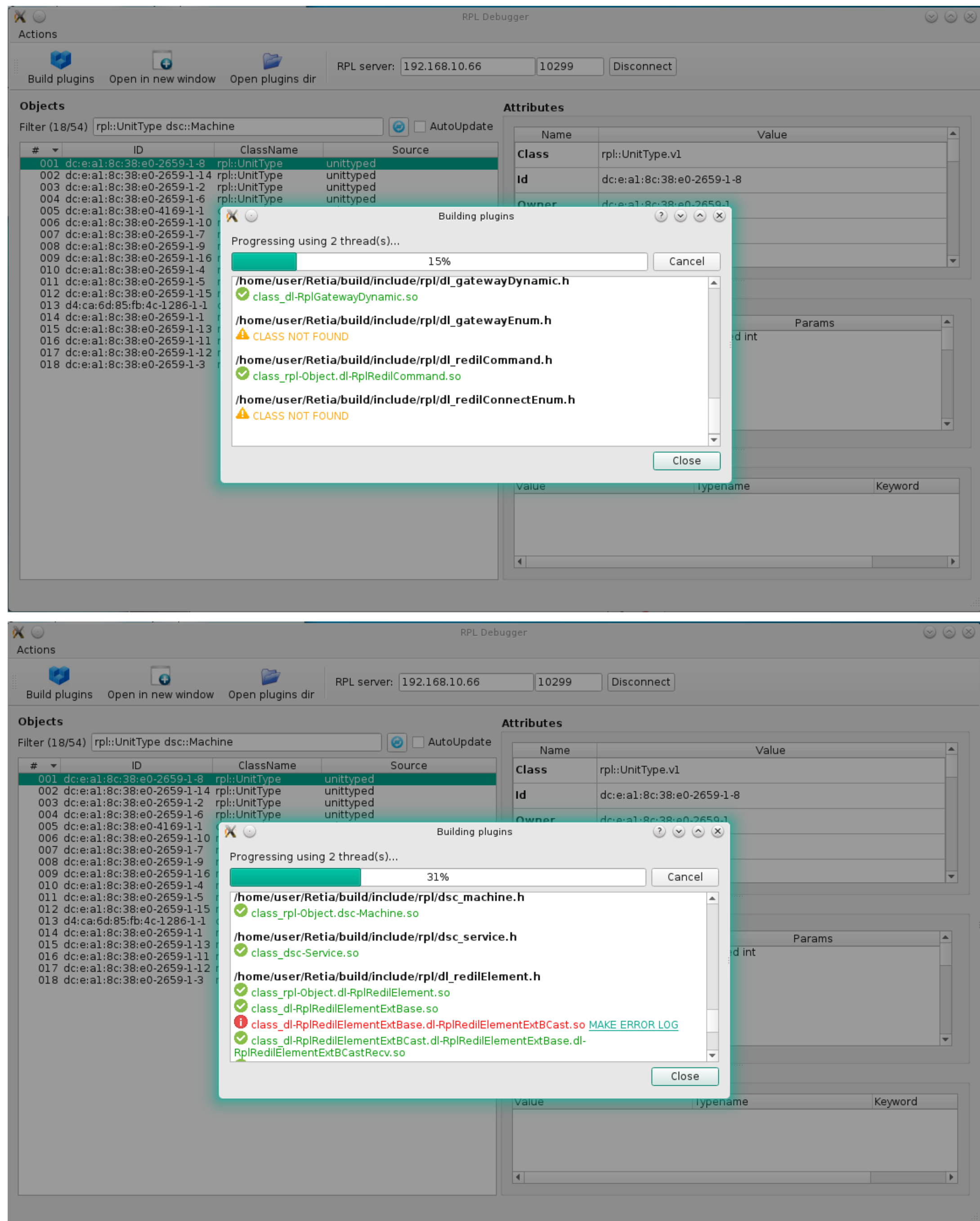
zdroj: vlastní

Filtrování umožňuje prohledávat i atributy objektů a hledat tak konkrétní atribut. V prvním přikladu (Obrázek 27) je hledaný objekt typu `rpl::UnitType` a zároveň musí hodnota atributu být „16s“. Zřetězit lze neomezené množství těchto řetězců v hledání. Tímto způsobem lze filtr o jednom textovém poli využít pro nespočet omezení výsledků hledání.

4.5.2 Zobrazování logu při kompilaci pluginů

V případě, že programátor vybere velké množství tříd replikovaných objektů, může kompilace pluginů trvat delší dobu. Jakmile běží kompilace pluginů, nelze s oknem RplDebuggeru pracovat. Aby uživatel viděl průběh kompilace, je spolu s indikátorem stavu (progressbar) zobrazen i textový log, který zobrazuje úspěšně a neúspěšně vytvořené pluginy. Zelenou barvou jsou barveny názvy pluginů, které se podařilo vytvořit. Žlutou barvou je zobrazena zpráva, pokud se v daném hlavičkovém souboru nepodařila najít třída. Červenou barvou jsou zobrazeny pluginy, které se nepodařilo vytvořit z důvodu chyby při kompilaci vygenerovaného pluginu. U tohoto typu logovací zprávy je odkaz „MAKE ERROR LOG“, který po kliknutí otevře textový soubor, ve kterém je zobrazení chybového výstupu kompilátoru.

Tuto funkcionalitu jsem přidal pro rychlé odladění aplikace v případě, že nově vytvářený objekt zaměstnancem firmy RETIA, a.s. nepůjde zkompilovat. Zaměstnanec firmy RETIA, a.s. mě může s tímto logem kontaktovat a já mohu najít zdroj chyby rychleji.



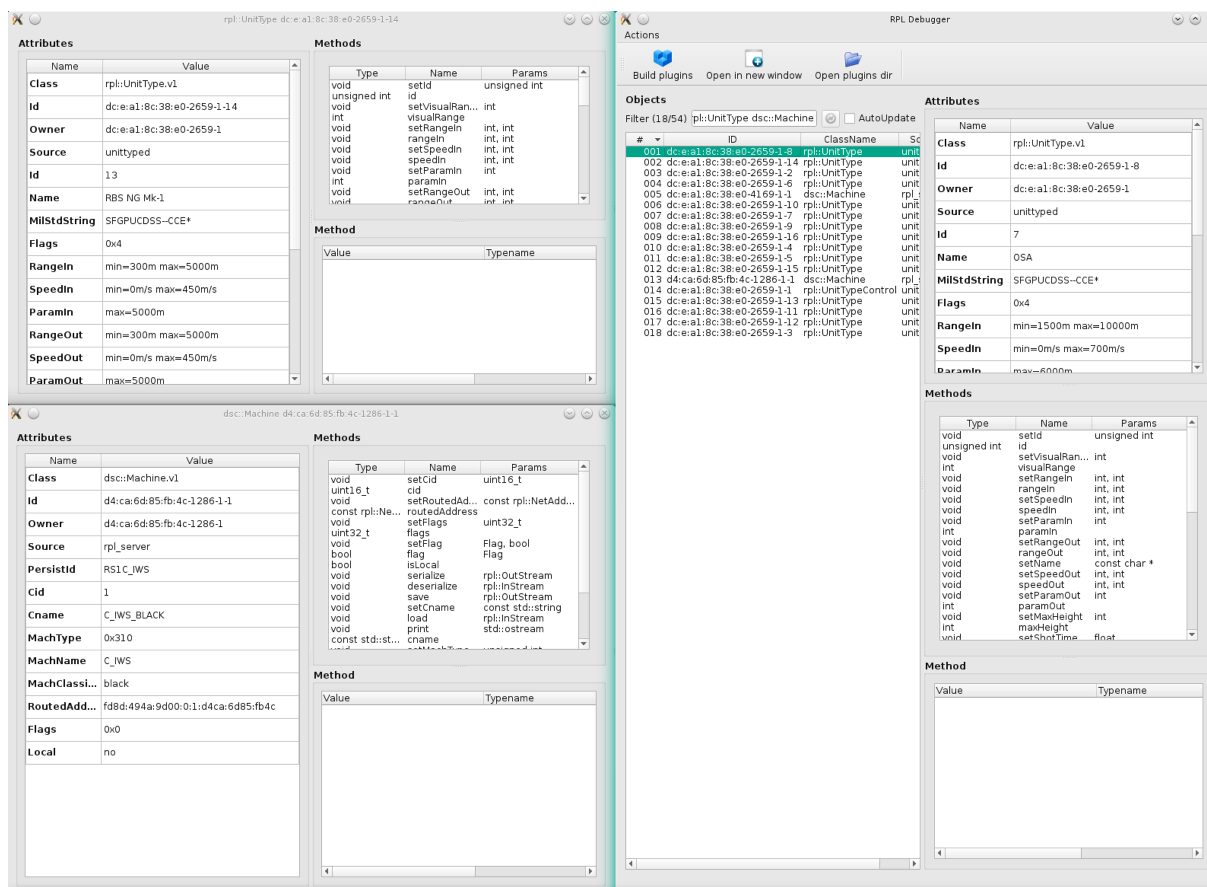
Obrázek 28 - Průběh kompilace pluginů na základě hlavičkových souborů

zdroj: vlastní

4.5.3 Zobrazení více oken

Při testování chování objektů se v praxi můžou objekty vzájemně ovlivňovat. Pokud by aplikace RplDebugger obsahovala pouze jedno okno, bylo by možné editovat pouze jeden replikovaný

objekt v daném čase a nebylo by možné sledovat změny v jiných objektech. Z tohoto důvodu byla do aplikace přidána podpora pro více oken. Otevření nového okna je možné kliknutím pravého tlačítka na konkrétní objekt v seznamu objektů nebo stačí daný objekt vybrat levým tlačítkem a stisknout tlačítko s ikonou v horní nabídce programu „Open in new window“ (Obrázek 29).



Obrázek 29 - Podpora více oken pro editací vícero objektů zároveň

zdroj: vlastní

4.6 Testování aplikace v praxi

Aplikce RplDebugger byla úspěšně otestována ve firmě RETIA, a.s. na kompilaci všech dostupných replikovatelných objektů, kterých je přibližně 70 (některé hlavičkové soubory obsahují více tříd). RplDebugger dokázal úspěšně vytvořit plugin pro 95% z nich. Příčinou neúspěchu kompilace některých pluginů pramení z faktu, že je velmi zřídka některý z replikovaných objektu vyděděný z nějaké další třídy, která je uvedena v jednom hlavičkovém souboru.

Podpora děděných replikovatelných objektů bude součástí budoucího rozvoje aplikace RplDebugger.

ZÁVĚR

Tato diplomová práce se zabývá návrhem ladícího nástroje pro replikační server. Ve firmě RETIA, a.s. vznikla potřeba testovat objekty určené pro jejich replikační řešení. Byl dán požadavek, aby aplikace uměla volat metody objektu a pozorovat, zda se požadované změny rozšířily napříč replikační sítí. Pro volání metod tohoto objektu dosud bylo vždy nutné vytvářet aplikaci konzolovou či s grafickým rozraním, která bude volat tyto metody s danými parametry. Ladicí nástroj, který je cílem této diplomové práce dokáže vytvářet grafické rozhraní univerzálně pro jakýkoliv objekt určený pro replikační server.

Při vývoji ladícího nástroje bylo nutné se nejdříve seznámit s problematikou replikace. V tomto tématu bylo důležité pochopit způsob, jakým jsou replikované objekty šířeny napříč počítačovou sítí a jak jsou změny objektů rozšířeny, aby na každém klientovi byl objekt ve shodném stavu jako na ostatních klientech. V této kapitole bylo klíčové, poznat strukturu objektu, který má být replikován.

Před samotným vývojem ladícího nástroje bylo potřeba vybrat vhodný GUI framework, který bude vyhovovat požadavkům na kompatibilitu s operačním systémem OpenSuse. Jako nejvhodnějším řešením se ukázal framework QT, který poskytuje i vlastní vývojové prostředí QT Creator a především již mají tyto knihovny připravené zázemí na počítačích ve firmě RETIA, a.s.

Druhým nejdůležitějším úkolem byl výběr vhodného C++ parseru, který by byl využit pro analýzu třídy pro názvy a parametry metod objektu. Nejvhodnějším řešením pro tento účel poskytl software pro tvorbu dokumentace – Doxygen. Největším argumentem pro výběr tohoto nástroje byla skutečnost, že Doxygen je dnes standardem na tvorbu dokumentace a je tak rozšířen na mnoha distribucích. Zároveň tento nástroj dokáže vyprodukovat informace o třídě do souboru XML, i když samotný zdrojový soubor třídy není zkompilovatelný.

V poslední části se tato diplomová práce zabývá konkrétním způsobem implementace tohoto ladícího nástroje. Samotná aplikace musela být nejdříve napojena na replikační službu, která tvoří interface, kterým proudí objekty a jejich změny. Poté bylo nutné analyzovat zdrojový kód C++ třídy replikovaného objektu pomocí aplikace Doxygen. Celý tento proces je řízen čistě ladícím nástrojem a není nutné používat Doxygen samostatně. V poslední řadě ladící nástroj přečte XML soubor a vytvoří na základě těchto znalostí o třídě zkompilovatelný plugin, který vytváří GUI rozhraní pro každou metodu objektu.

Ladící nástroj funguje spolehlivě na téměř všechny replikované objekty ve firmě RETIA, a.s. Výjimku tvoří specificky vytvořené objekty, které v rámci jednoho hlavičkového souboru využívají dědičnosti a RPLDebugger zatím tyto závislosti nedokáže detekovat, takže vygenerovaný kód pluginu nelze sestavit. Pro volání metod je zde podpora pro zadávání parametrů základních datových typů jako je `int`, `double`, `char` apod. Je zde podpora i pro neznámé objekty, o jejichž konstrukci se plugin snaží zjistit informace od dalších pluginů vyhovující názvu neznámého objektu. Takto lze generovat GUI rekurzivně, dokud ladící nástroj nenarazí na základní datový typ, pro který lze prvek GUI vytvořit.

Ladící nástroj nyní umožňuje měnit parametry objektu pomocí forumulářových prvků. Při vývoji nového objektu určeného pro replikační server programátor pouze nechá ladícím nástrojem zanalyzovat třídu objektu, na základě které zajistí podporu GUI pro volání metod. Tento proces zabere celkově méně než jednu minutu. Do této chvíle vývoj aplikace pro otestování nového objektu pro replikační síť trvala řádově jednotky hodin, takže v tomto ohledu se jedná o významné snížení finančních a časových nákladů.

Ladicí nástroj jakožto aplikace má ambice být nadále rozvíjen. Dalším možným rozšířením by mohla být podpora pro zobrazení návratové hodnoty metody po jejím zavolání.

Program RplDebugger vytvořený v rámci této diplomové práce se stal v letošním roce součástí instalačního balíku replikační kihovny librpl ve firmě RETIA, a.s.

POUŽITÁ LITERATURA

anonymous. 2008. Referáty-seminárky.cz. *Endianita.* [Online] 12. 8 2008. [Citace: 8. 4 2017.] http://referaty-seminarky.cz/endianita/.

Bílek, Petr. 2003. sallyx.org. *Datové typy, bloky, deklarace a definice.* [Online] 29. 8 2003. [Citace: 2. 4 2017.] http://www.sallyx.org/sally/c/c05.php.

cppreference. 2017. cppreference.com . *Fundamental types.* [Online] 23. 2 2017. [Citace: 2. 4 2017.] http://en.cppreference.com/w/cpp/language/types.

Karas, Michal. 2006. ABC Linuxu. *GTK+.* [Online] 11. 8 2006. [Citace: 17. 4 2017.] http://www.abclinuxu.cz/software/programovani/knihovny/gtkp.

King, Brad. 2017. github. *CastXML.* [Online] 2017. [Citace: 22. 4 2017.] https://github.com/CastXML/CastXML#readme.

Kreinin, Yossi. 2009. yosefk. *Defective C++.* [Online] 17. 10 2009. [Citace: 22. 4 2017.] http://www.yosefk.com/c++fqa/defective.html#defect-2.

QT-Company. 2017. QT. *Identify the right license for your project.* [Online] 2017. [Citace: 22. 4 2017.] https://www.qt.io.

Redhat. 2017. Redhat. *GTK+ History.* [Online] 2017. [Citace: 22. 4 2017.] http://people.redhat.com/mclasen/Usenix04/notes/x29.html.

ReplicaSoftware. 2017. http://replicanet.com/. *ReplicaNet multiplayer networking.* [Online] 2017. [Citace: 12. 4 2017.] http://replicanet.com/.

Shamendra, Udaya. 2014. masteringqt. *History of Cute QT.* [Online] 2014. [Citace: 22. 4 2017.] http://www.masteringqt.com/2013/06/history-of-cute-qt.html.

TAUFER, Ivan, KOTYK, Josef a JAVŮREK, Milan. 2009. *Jak psát a obhajovat závěrečnou práci bakalářskou, diplomovou, rigorózní, disertační, habilitační.* Pardubice : Univerzita Pardubice, 2009. ISBN 978-80-7395-157-3.

UnrealEngine. 2017. Unreal Engine. *Replication.* [Online] 9. 4 2017. [Citace: 14. 4 2017.] https://wiki.unrealengine.com/Replication.

wxWidgets. 2017. wxWidgets. *Introduction.* [Online] 2017. [Citace: 22. 4 2017.] http://docs.wxwidgets.org/trunk/page_introduction.html.

Zachar, Jiří. 2008. zaachi.com. *WxWidgets.* [Online] 3. 12 2008. [Citace: 17. 4 2017.] http://zaachi.com/2008/12/03/wxwidgets.html.

PŘÍLOHY

Příloha A – *Velikosti datových typů na různých platformách* 80

Příloha B – *Zdrojový kód souboru main.cpp z tutorialu ReplicaNet*..81

Příloha C – *Zdrojový kód rozhraní rpl_service.h z knihoven RETIA, a.s.* 84

Příloha D – *Zdrojový kód pluginu pro třídu rpl_testobject.h RETIA, a.s.* 86

Příloha A – *Velikosti datových typů na různých platformách*

Název datového typu	**Ekvivalent**	**Velikost v bitech**				
		C++ standard	**LP32**	**ILP32**	**LLP64**	**LP64**
short	short int	alespoň	**16**	**16**	**16**	**16**
short int		**16**				
signed short						
signed short int						
unsigned short	unsigned short int					
unsigned short int						
int	int	alespoň	**16**	**32**	**32**	**32**
signed		**16**				
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	alespoň	**32**	**32**	**32**	**64**
long int		**32**				
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int	alespoň	**64**	**64**	**64**	**64**
long long int	(C++11)	**64**				
signed long long						
signed long long int						
unsigned long long	unsigned long long int					
unsigned long long int	(C++11)					

Příloha B – *Zdrojový kód souboru main.cpp z tutorialu ReplicaNet*

```
#include <stdio.h>
#include <stdlib.h>
#include <string>

#include "RNReplicaNet/Inc/ReplicaNet.h"
#include "RNPlatform/Inc/ThreadClass.h"

#include "Player.h"
#include "Enemy.h"

int main(int argc,char **argv)
{
    RNReplicaNet::ReplicaNet *myNetwork = new RNReplicaNet::ReplicaNet;

    if (!myNetwork)
    {
        printf("ReplicaNet failed to start\n");
        exit(-1);
    }

    printf("ReplicaNet started\nFinding a session...\n");

    // Start finding any sessions out there
    myNetwork->SessionFind();

    // Sleep for a second
    RNReplicaNet::CurrentThread::Sleep(1000);

    // Now enumerate through our list of found sessions
    std::string found;
    std::string firstFound;
    do
    {
        found = myNetwork->SessionEnumerateFound();
        // If a session is found then print the address
        if (found != "")
        {
            printf("Found '%s'\n",found.c_str());
        }
        // If we have not found any sessions yet and the session address
contains "Tutorial1" then store the address
        if (found.find(std::string("Tutorial1")) != std::string::npos &&
firstFound == "")
        {
            firstFound = found;
        }
    } while (found != "");

    // Did we find a suitable session?
    if (firstFound == "")
    {
        // If no session found then create a session
        printf("Failed to find a session, so starting a new session\n");
        myNetwork->SessionCreate("Tutorial1");
    }
    else
    {
        // Try to join the session
        printf("Joining '%s'\n",firstFound.c_str());
```

```
        myNetwork->SessionJoin(firstFound);
    }

    // We do not have to wait for the session to become stable, however in
a game with hundreds of
    // active objects it can take a second for the data to propagate to our
session.
    // We can wait for the session to become stable so we know when the
session is populated with
    // objects.
    while (!myNetwork->IsStable())
    {
        printf("Waiting for the session to become stable\n");
        RNReplicaNet::CurrentThread::Sleep(500);
    }
    printf("Session now stable\n");

    printf("Creating the local objects\n");
    // Allocate and publish our Player object
    RNReplicaNet::ReplicaObject *myPlayer = new Player;
    // Call the PostObjectCreate() function, since we allocated the object.
    // If ReplicaNet allocates an object then
ReplicaObject::PostObjectCreate() is called
    // after the object has been allocated and the initial DataBlock
variables have been initialised.
    myPlayer->PostObjectCreate();
    // Publish the myPlayer object so that it can appear on other machines
    myPlayer->Publish();

    // Allocate and publish our monster objects in a similar way to the
player
    RNReplicaNet::ReplicaObject *object;
    object = new Enemy;
    object->PostObjectCreate();
    object->Publish();
    object = new Enemy;
    object->PostObjectCreate();
    object->Publish();
    object = new Enemy;
    object->PostObjectCreate();
    object->Publish();
    object = new Enemy;
    object->PostObjectCreate();
    object->Publish();

    printf("Going to start the game poll loop\n");
    RNReplicaNet::CurrentThread::Sleep(5000);

    bool doGameLoop = true;
    while(doGameLoop)
    {
        RNReplicaNet::ReplicaObject *iterator;
        myNetwork->ObjectListBeginIterate();

        // Quickly scroll off the old data
        int i;
        for (i=0;i<80;i++)
        {
            printf("\n");
        }
```

```
        printf("New Frame\n");
        iterator = myNetwork->ObjectListIterate();
        while (iterator)
        {
            printf("Object session id %d and id %d with type
%d\n",iterator->GetSessionID(),iterator->GetUniqueID(),iterator-
>GetClassID());

            // Every object ctor uses SetOpaquePointer() to set the
BaseObject pointer
            BaseObject *baseObject = (BaseObject *)iterator-
>GetOpaquePointer();
            baseObject->Tick(myNetwork->GetLocalTime());


            iterator = myNetwork->ObjectListIterate();
        }
        myNetwork->ObjectListFinishIterate();

        RNReplicaNet::CurrentThread::Sleep(1000);
    }

    return 0;
}
```

Příloha C – *Zdrojový kód rozhraní rpl_service.h z knihoven RETIA, a.s.*

```
    .................
    .................

protected:
    // Požadavek replikační služby na zaslání dat - zaslat data musíme
vlastním spojením
    virtual void send(const uint8_t *data, uint32_t bytes, void *clientPtr
= NULL) = 0;

    // V replikační síti vznikl nový objekt
    virtual bool onAdd(Object */*object*/) { return false; }

    // Objekt se změnil
    virtual bool onChange(Object */*object*/, AttributeUInt /*attributes*/)
{ return false; }

    // Objekt byl smazán vlastníkem, nebo se vlastník odpojil
    virtual bool onRemove(Object */*object*/) { return false; }

    // Vlastník objektu byl změněn
    virtual bool onMigrate(Object */*object*/) { return false; }

    // Odpověď na keepalive zprávy
    virtual void onEchoReply(void */*clientPtr*/) { }

    // Volání které následuje po přijetí všech objektů po připojení do
replikační sítě
    virtual void onSync(void */*clientPtr*/) { }

    // Vrácení návratové hodnoty při vzdáleném volání
    virtual bool onReturn(Object */*object*/, MethodUInt /*method*/,
                          CallError /*error*/, InStream &/*retval*/) {
return false; }
    //Přijetí debugovací zprávy
    virtual void onDebug(Object */*object*/, const char */*prefix*/, const
char */*text*/) { }

    // přijetí vzdáleného volání
    virtual bool process(Object */*object*/,
                         AttributeUInt /*attribute*/, ActionUInt
/*action*/,
                         InStream &/*stream*/) { return false; }

    //Odmítnutí požadavku, který je právě obsluhován v metodě process
    void reject();

    // Obsluha volání
    virtual CallStatus process(Object */*object*/, MethodUInt /*method*/,
                               InStream &/*args*/, OutStream &/*retval*/) {
return CEPassToObject; }


    // Obsluha hromadného volání
    virtual bool process(Object */*object*/, MethodUInt /*method*/,
InStream &/*args*/)
    {
        return false;
    }
```

```
    // Přetížení při vytváření nového objektu
    virtual Object *create(const ClassInfo &/*info*/, InStream &/*object*/)
{ return NULL; }

    .................
    .................
```

Příloha D – *Zdrojový kód pluginu pro třídu rpl_testobject.h RETIA, a.s.*

```
class ObjectWidget : public DebObjectWidget, public rpl::TestObject
{
    Q_OBJECT
private:
    //Replicated object
    rpl::TestObject* pObject_;
public:
    //Create all GUI and methods calls
    ObjectWidget(void *object) : DebObjectWidget()
    {
        pObject_ = (rpl::TestObject *)object;
        className_="rpl::TestObject";
        createMethods();
        createMethodsListWidget();
    }

    //Connect buttons with method calls
    void createMethods(){
        DebParameterList *paramList01 = new DebParameterList();

        paramList01->addParameter(
        new DebParameter( "bool", "boolean",
        std::is_enum<bool>::value, std::is_integral<bool>::value,
        false,""));

        DebMethod *method01 = new DebMethod("void","setBoolean",
        paramList01, true);

        methodLayout_->addWidget(method01->getButton(),1,0);

        QObject::connect(method01->getButton(),SIGNAL(pressed()),this,
        SLOT(setBoolean01()));

        createWidgets(method01);
        methodList_->addMethod(method01);
        methodList_->hide();


    .................
    .................
    }
    //Method list widget to choose method
    void createMethodsListWidget()
    {
        QTreeWidgetItem * item1 = new QTreeWidgetItem();
        item1->setText(0,"0");
        item1->setText(1,"void");
        item1->setText(2,"setBoolean");
        item1->setText(3,"bool");
        treeWidgetMethodList_->addTopLevelItem(item1);
    .................
    .................
```

```
// Methods to call

public slots:
    .................
    .................

  void setInteger03()
    {
      DebParameterList *paramList =
      methodList_->at(2)->getParameterList();


        if(paramList && paramList->size()==1)
        {
            int parameter1 = (int)returnValue(
            paramList->at(0)->getValueWidget(),"int").toInt();

            pObject_->setInteger( parameter1);
            emit objectChanged();
        }
    }


    .................
    .................

//Interface to access plugin
class Plugin1 :  public DebPluginInterface,public rpl::TestObject
{


    .................
    .................

public:
    // Get list of parameters to construct this object
    DebMethodList* getConstructorParametersList()
    {
        DebMethodList * methodList = new DebMethodList();
        DebParameterList *paramList00 = new DebParameterList();

        paramList00->addParameter(
        new DebParameter("","",false,false,false));

        methodList->addMethod(new DebMethod("","TestObject",paramList00));
        return methodList;
    }

    // Get all methods GUI
    DebObjectWidget* getObjectWidget(void *object)
    {
        return new ObjectWidget(object);
    }

    .................
    .................
}
```