

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Distribuovaná simulace
podle standardu HLA pro použití
v simulační knihovně J-Sim

Abstract

Distributed simulation, in compliance with the HLA standard, for use with the J-Sim simulation library

The goal of this thesis is to design and implement distributed simulation, in compliance with the High Level Architecture (HLA), for use with the J-Sim simulation library.

This task consists of two parts. The first part concerns with the implementation of the HLA support for the J-Sim simulation library. The key design aspects were to retain the compatibility with non-distributed simulation and to make distributed simulation easy to use.

The second part deals with the implementation of the Runtime Infrastructure (RTI). There is an open source, freely redistributable, Java-based implementation of RTI interface, called Extensible Run-Time Infrastructure (XRTI).

Since J-Sim is a tool for discrete simulation, it needs to use some HLA time management functions. However, current version of XRTI does not support them. To solve this problem, these functions must be implemented into the XRTI.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 20. května 2005

Stanislav Kožina

Obsah

1	Úvod	1
2	Simulační knihovna J-Sim	2
2.1	Hlavní principy diskretních simulací	2
2.1.1	Základní pojmy	2
2.1.2	Průběh výpočtu diskretních simulací	3
2.1.3	Stavy procesů v simulaci	4
2.2	Struktura knihovny J-Sim	5
2.2.1	Třída JSimSimulation	5
2.2.2	Třída JSimProcess	5
2.2.3	Třída JSimHead	6
2.2.4	Třída JSimLink	7
2.2.5	Systém výjimek	8
2.2.6	Další funkce knihovny J-Sim	9
3	High Level Architecture	10
3.1	Vznik HLA	10
3.2	Základní pojmy	11
3.3	Pravidla HLA	11
3.3.1	Pravidla federace	11
3.3.2	Pravidla členů federace	12
3.4	Object Model Template	13
3.4.1	Interakce	13
3.4.2	Objekty	14
3.4.3	Komponenty OMT	14
3.5	Specifikace rozhraní	15
3.5.1	Runtime Infrastructure	16
3.5.2	Federation Management	16
3.5.3	Declaration Management	17
3.5.4	Object Management	18
3.5.5	Ownership Management	19
3.5.6	Time Management	19
3.5.7	Data Distribution Management	22

3.5.8	Podpůrné služby	23
4	Přehled implementací Runtime Infrastructure	24
4.1	pRTI	24
4.2	Extensible Run-Time Infrastructure	25
5	Simulační knihovna J-Sim s podporou HLA	26
5.1	Požadavky na řešení	26
5.2	Princip řešení	26
5.3	Komunikační protokol člena federace	28
5.3.1	Návrh FED souboru pro RTI	29
5.3.2	Návrh FDD souboru pro XRTI	31
5.4	Realizace řešení	32
5.4.1	Třída JSimHLASimulation	32
5.4.2	Třída JSimHLARemoteProcess	34
5.4.3	Třída JSimHLARemoteHead	34
5.4.4	Třídy JSimHLARequest, JSimHLARequestHeader	35
5.4.5	Třídy JSimHLAResponse, JSimHLAResponseHeader	37
5.4.6	Třída JSimHLAReceivedInteractionCallback	37
5.4.7	Třída JSimHLACallbackQueue	37
5.4.8	Třída JSimHLABarrier	38
5.4.9	Třída JSimHLAFederateAmbassador	38
5.4.10	Třída JSimHLAConnection	38
5.4.11	Metoda step() pro distribuovanou simulaci	45
5.4.12	Úpravy tříd JSimProcess a JSimHead	47
5.4.13	Úpravy třídy JSimLink	48
6	Úpravy software XRTI	49
6.1	Struktura XRTI	49
6.1.1	Třída XRTIAmbassador	49
6.1.2	Třída ExecutiveClientAmbassador	50
6.1.3	Třída FederationExecutionAmbassador	51
6.1.4	Třída XRTIExecutive	51
6.1.5	Třída ProxyAmbassador	51
6.2	Implementace času pro typ double	53
6.2.1	Třídy DoubleValuedLogicalTime, DoubleValuedLogicalTimeFactory	54
6.2.2	Třídy DoubleValuedLogicalTimeInterval, DoubleValuedLogicalTimeIntervalFactory	54
6.2.3	Úpravy třídy XRTIExecutive	54

6.3	Implementace služeb skupiny	
	time management	55
6.3.1	Služba enableTimeRegulation()	56
6.3.2	Služba enableTimeConstrained()	58
6.3.3	Služba nextMessageRequestAvailable()	58
6.3.4	Asynchronní doručování zpráv	61
6.4	Nové třídy pro time management	61
6.4.1	Třída HLAMessage	62
6.4.2	Třída HLAMessageSendInteraction	62
6.4.3	Třídy HLAMessageUpdateAttributeValues, HLAMessageDeleteObjectInstance	63
6.4.4	Třída TimeStampQueue	63
6.4.5	Třída ReceiveOrderQueue	63
6.5	Podpora časových razítek pro interakce	64
6.5.1	Úpravy struktury HLAbootstrapInteractionPayload	64
6.5.2	Úpravy třídy XRTIAmbassador	66
6.5.3	Úpravy třídy ExecutiveClientAmbassador	71
6.6	Další funkce XRTI	73
7	Závěr	74
	Zkratky	76
	Literatura	77
A	Uživatelská příručka k JSimHLA	79
A.1	Vytvoření distribuované simulace	79
A.1.1	Vytvoření objektu simulace	79
A.1.2	Lokální a vzdálené objekty simulace	80
A.1.3	Třída JSimHLARemoteProcess	80
A.1.4	Třída JSimHLARemoteHead	81
A.1.5	Zahájení a ukončení distribuované simulace	83
A.2	Spuštění distribuované simulace	83
A.3	Nastavení proměnné CLASSPATH	83
A.4	Překlad projektu	84
B	Uživatelská příručka k XRTI	85
B.1	Spuštění XRTIExecutive	85
B.2	Překlad projektu	86

Kapitola 1

Úvod

Cílem této práce je umožnit snadnou distribuovatelnost simulací podle standardu High Level Architecture (HLA), prováděných s využitím simulační knihovny J-Sim.

Simulační knihovna J-Sim je určena pro diskrétní procesově orientované simulace. Je napsána v jazyce Java. Popis knihovny J-Sim a hlavních principů diskrétních procesově orientovaných simulací lze nalézt ve druhé kapitole.

Standard High Level Architecture (HLA) byl vytvořen za účelem provádění paralelních a distribuovaných simulací. Standard HLA vznikl původně pro vojenské účely a jedná se tedy o poměrně silný a komplikovaný nástroj. Základní popis HLA lze nalézt ve třetí kapitole.

Čtvrtá kapitola přináší stručný přehled implementací Runtime Infrastructure (RTI). Tento software slouží jako server pro distribuovanou simulaci. V současné době existuje pouze několik implementací RTI. Většinou se jedná o komerční software, jehož pořízení není levná záležitost. Kromě toho existuje také volně šiřitelná implementace nazvaná Extensible Run-Time Infrastructure (XRTI), ke které jsou dostupné zdrojové kódy v jazyce Java. Tato implementace byla použita pro další práci.

Pátá kapitola se zabývá implementací podpory HLA pro knihovnu J-Sim. Hlavním požadavkem je vznik takového systému, který je maximálně kompatibilní s předchozími nedistribuovanými verzemi knihovny J-Sim. Dalším požadavkem je, aby použití distribuované simulace nebylo pro uživatele knihovny J-Sim příliš komplikované.

Vzhledem k tomu, že J-Sim je nástrojem pro diskrétní simulaci, potřebuje používat některé služby pro řízení simulačního času. Tyto služby HLA patří do skupiny Time Management. Současná verze XRTI tyto služby bohužel nepodporuje. Návrh řešení tohoto problému je podrobněji popsán v šesté kapitole.

Kapitola 2

Simulační knihovna J-Sim

Simulační knihovna **J-Sim** byla vyvinuta na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Další informace lze nalézt v [Kac01]. Aktuální verzi knihovny včetně zdrojových kódů lze stáhnout z [www stránek](http://www.j-sim.org) ([J-Sim]).

Knihovna je určena pro diskrétní procesově orientované simulace. Typickou oblastí použití J-Sim je ověření funkce distribuovaných, paralelních a FT (fault-tolerant) systémů a programů. Knihovna J-Sim je inspirována programovacím jazykem Simula (objektově orientovaný jazyk, využívaný pro simulační výpočty). Z pohledu uživatele nabízí alternativu ke knihovně C-Sim, napsané v jazyce C.

Knihovna J-Sim je kompletně napsána v jazyce **Java**¹, což přináší určité výhody: objektově orientovaný jazyk, snadná přenositelnost mezi různými platformami, využití vláken (*threads*) a synchronizačních metod `wait()` a `notify()`, které jsou součástí tohoto jazyka.

2.1 Hlavní principy diskrétních simulací

2.1.1 Základní pojmy

Před vysvětlením principů diskrétní simulace je vhodné nejprve uvést několik základních pojmů:

- **Model** – účelově konstruované napodobení reálného objektu. Při konstrukci modelu se obvykle využívá:
 - *zjednodušení* – v úvahu se berou pouze takové vlastnosti objektu, které jsou významné vzhledem k účelu vytvoření modelu.

¹<http://www.java.sun.com/>

- *zobecnění* – nemodeluje se jediný konkrétní reálný objekt, ale celá třída reálných objektů, které mají podobné vlastnosti.

Zjednodušení a zobecnění se obvykle označuje jako *abstrakce*.

- **Systém** – účelově definovaná množina prvků, mezi kterými existují určité vazby. Definování systému nad reálným objektem vyžaduje jednak dříve uvedené zjednodušení a zobecnění, jednak přesný popis vlastností prvků a vazeb mezi nimi.
- **Modelování** – proces konstrukce a využití modelu. Tato činnost vede k získání informací o reálném objektu prostřednictvím modelu systému.
- **Simulační model** – program vykonávaný v reálném čase na počítači s cílem napodobit chování reálného objektu. Simulační program lze vytvořit buď přepisem numerického matematického modelu do programovacího jazyka nebo přímým modelováním časových změn struktury a vlastností prvků systému v modelovém čase.
- **Simulace** – výzkumná metoda, jejíž podstata spočívá v nahrazení zkoumaného systému simulačním modelem. Se simulačním modelem se provádí experimenty, jejichž cílem je získání informací o původně zkoumaném systému.
- **Diskrétní simulace** – ke změnám v modelu dochází pouze v diskrétních časových bodech.

2.1.2 Průběh výpočtu diskrétních simulací

Pro řízení průběhu výpočtu diskrétních simulací se nejčastěji používají dvě základní metody:

- **Metoda interpretace událostí** – všechny události, které mohou nastat v budoucím vývoji modelu, jsou uloženy v seznamu, který se nazývá kalendář událostí. Tento seznam je vzestupně seřazen podle hodnoty modelového času vzniku události. Řídící algoritmus simulačního výpočtu postupně interpretuje jednotlivé události. Tato interpretace probíhá v diskrétním bodě (čas vzniku události) a realizuje se voláním podprogramu, který přísluší k danému typu události. Interpretační podprogram provede v modelu změny příslušné k události a následně je tato událost odstraněna z kalendáře. Pokud některá událost způsobí vznik další události v (budoucím) modelovém čase, bude provedeno naplánování nové události (událost bude přidána do kalendáře).

- **Metoda pseudoparalelních procesů** – je založena na objektové dekompozici řídicího algoritmu simulačního výpočtu. Jednotlivé části výpočtu jsou zapouzdřeny ve třídách objektů, které získávají charakter samostatných vlastních výpočetních procesů (mají vlastní „život“). Tyto výpočetní procesy jsou vykonávány *pseudoparalelně* – jejich vykonávání se střídá v modelovém čase (v každém časovém okamžiku reálného času je aktivní nejvýše jeden proces). Organizační datovou strukturou pro výpočet je opět kalendář. Události v kalendáři jsou uspořádány vzestupně podle hodnoty času aktivace události. Každá událost navíc obsahuje odkaz na program procesu. Řídící smyčka výpočtu postupně spouští procesy podle pořadí daného záznamy v kalendáři.

Knihovna J-Sim je založena na metodě pseudoparalelních procesů.

Diskrétní simulace se dělí na **kroky** (*steps*). Jeden krok se vykoná vždy v jednom bodě (nikoliv intervalu) simulačního času. Tento krok může být vykonáván v libovolně dlouhém intervalu reálného času. V jednom bodě simulačního času se může odehrát i více simulačních kroků.

Vykonávání simulačního kroku nemůže být přerušeno vnějším zásahem – to znamená, že běžící proces se po dokončení své části výpočtu sám vzdává řízení ve prospěch jiného procesu. Mezi dvěma po sobě následujícími kroky simulace může být libovolně velká pauza simulačního času – v reálném čase se tato pauza přeskočí.

2.1.3 Stavy procesů v simulaci

Každý proces v simulaci vytvořené s využitím J-Sim se v každém okamžiku simulace nachází v jednom z následujících stavů²:

- **Nový** – Proces se nachází v tomto stavu, pokud již byl vytvořen, ale dosud nebyl naplánován.
- **Pasivní** – Proces je *pasivní*, pokud již bylo zahájeno jeho vykonávání, ale nemá v kalendáři žádnou událost. Pokud nebude naplánován, již nepoběží.
- **Aktivní** – Proces je *aktivní*, pokud právě běží.
- **Plánovaný** – Proces právě neběží, ale má v kalendáři naplánovanou událost. Může tedy běžet v některém budoucím simulačním kroku.
- **Ukončený** – Proces doběhl na konec svého programu a jeho život skončil. Tento proces již nikdy nepoběží.

²Novější verze J-Sim (od verze 0.3.0) obsahují další stavy procesů. Uvedený seznam stavů je tedy třeba chápat jako zjednodušený s ohledem na vysvětlení metody pseudoparalelních procesů.

2.2 Struktura knihovny J-Sim

Knihovna J-Sim se skládá z několika tříd v jazyce Java. Všechny třídy patří do balíku (*package*) `cz.zcu.fav.kiv.jsim`³. Uvedený balík je třeba importovat v každém simulačním programu, který využívá služeb knihovny.

V následujícím textu bude uveden stručný popis nejdůležitějších tříd knihovny J-Sim.

2.2.1 Třída JSimSimulation

Třída `JSimSimulation` reprezentuje teoretický simulační model, který obsahuje další objekty: **procesy** (viz 2.2.2) a **fronty** (viz 2.2.3). Každý simulační objekt obsahuje jednu instanci třídy `JSimCalendar`, který obsahuje události pro všechny procesy, které se účastní simulace. Během jednoho simulačního kroku se z kalendáře vezme vždy jedna událost, která se po dokončení kroku z kalendáře odstraní.

Třída `JSimSimulation` nabízí metodu `step()` – jedno volání této metody provede vždy jeden simulační krok. Hlavní program simulace tedy obvykle obsahuje cyklus, který volá metodu `step()` buď do dosažení požadovaného počtu kroků simulace nebo požadované hranice simulačního času.

2.2.2 Třída JSimProcess

Třída `JSimProcess` je určena k reprezentaci procesu, tedy nějaké aktivity v simulačním modelu. Uživatel simulace by měl tuto třídu vždy zdědit a překrýt metodu `life()`, která obsahuje kód „života“ procesu. Metoda `life()` není uživatelem volána přímo, je zavolána automaticky jádrem knihovny J-Sim z metody `run()` vlákna procesu.

Kód procesu může být prakticky libovolný (i nekonečný, např. `while(true)`), proces se po provedení části svého výpočtu sám vzdává řízení. K tomu slouží metody `hold()` a `passivate()` (vysvětlení bude následovat) – volání jedné z těchto metod ukončuje aktuální simulační krok. Pokud program procesu dojde na konec metody `life()`, znamená to konec života procesu (současně i aktuálního simulačního kroku) a tento proces (tato instance) již v rámci simulace nepoběží.

V konstruktoru třídy odvozené od `JSimProcess` (konstruktor takové třídy musí existovat, protože `JSimSimulation` má pouze jeden konstruktor, který má dva parametry) je nutné nejprve zavolat konstruktor předka a předat mu jméno procesu a odkaz na platnou instanci třídy `JSimSimulation` (např. `super(name, simulation)`).

³Tento balík obsahuje další balíky, které knihovnu rozšiřují o možnost použití dodatečných funkcí.

Třída `JSimProcess` dědí od třídy `java.lang.Thread` a využívá podpory vláken jazyka Java. Metoda `run()` třídy `JSimProcess` nesmí být překryta (je označena `final`) ani být volána přímo – je volána automaticky z metody `start()` příslušného vlákna (to je dáno implementací vláken v Javě). Metoda `start()` je volána jádrem knihovny J-Sim při spouštění nového procesu. Metoda `run()` volá nejprve (privátní) metodu `getReady()` a následně metodu `life()`. Podrobnější popis je možné nalézt v dokumentaci k J-Sim.

Třída poskytuje tyto metody pro plánování procesů (jsou označeny jako `final`, takže je nelze překrýt):

- **`activate(when)`** – aktivace procesu. Používá se v případě, že některý proces potřebuje naplánovat jiný proces (tuto metodu nemůže proces použít sám na sobě). Plánovaný proces musí být ve stavu *pasivní*. Parametr **`when`** udává simulační čas, na který proces bude naplánován (nelze použít čas v minulosti).
- **`cancel()`** – zrušení naplánování. Používá se v případě, kdy některý proces chce zrušit naplánování jiného procesu. Metodu lze použít „z vnějšku“ na jiný proces, který je ve stavu *plánovaný* (nelze ji použít sám na sobě). Po volání této metody bude z kalendáře odstraněna příslušná událost.
- **`hold(deltaT)`** – pozastavení aktivity na dobu určitou. Volání této metody může proces použít pouze sám na sobě. Z toho vyplývá, že při volání musí být ve stavu *aktivní*. Do kalendáře bude zařazena nová událost a proces bude pozastaven. Proces bude „probuzen“ po **`deltaT`** jednotkách simulačního času.
- **`passivate()`** – pozastavení aktivity na dobu neurčitou. Volání této metody může proces použít pouze sám na sobě, musí tedy být ve stavu *aktivní*. Vykonávání procesu bude pozastaveno, do kalendáře se však nepřidává žádná nová událost. Pokud nebude volající proces znovu naplánován jiným procesem (příp. hlavním programem), již nikdy nepoběží.

2.2.3 Třída `JSimHead`

Třída `JSimHead` je ekvivalentem typu `HEAD` z jazyka Simula. Reprezentuje hlavu spojového seznamu, který může obsahovat prvky typu `JSimLink` (viz 2.2.4). V konstruktoru třídy je (kromě jména seznamu) třeba určit, do které simulace (instance třídy `JSimSimulation`) bude seznam patřit.

Metody třídy poskytují přístup k prvkům seznamu a základní statistické funkce:

- **`empty()`** – vrací `true`, pokud je seznam prázdný (jinak vrací `false`).

- `cardinal()` – vrací počet prvků seznamu.
- `first()` – vrací první prvek seznamu.
- `last()` – vrací poslední prvek seznamu.
- `clear()` – odstraní všechny prvky seznamu.
- `getLw()` – vrací hodnotu veličiny *střední délka fronty* L_w .
- `getTw()` – vrací hodnotu veličiny *střední doba čekání ve frontě* T_w .

Třída `JSimHead` neobsahuje žádné metody pro přidávání a odstraňování jednotlivých prvků. Vlastnost „být členem seznamu“ zde není chápána jako schopnost seznamu, ale jako schopnost prvku, který pro tyto akce poskytuje potřebné metody.

2.2.4 Třída `JSimLink`

Třída `JSimLink` představuje jednu položku spojového seznamu (viz 2.2.3). Ideově vychází z typu `LINK` jazyka `Simula`. Na rozdíl od `Simuly` ovšem nemá se seznamem společného předka `LINKAGE`, obě třídy jsou odvozeny přímo od `java.lang.Object`.

Nový prvek seznamu lze vytvořit některým ze dvou konstruktorů. V případě použití konstruktoru bez parametrů je ovšem nutné oddělené uložení případných uživatelských dat (pozdější přidání dat není možné). Výhodnější je tedy použít konstruktor s parametrem typu `Object`, kam si uživatel může při vytváření prvku vložit libovolná data (nejlépe v podobě instance uživatelem vytvořené třídy) a později k nim přistupovat s využitím metody `getData()`. Výhodou této implementace prvku je skutečnost, že lze využívat přímo instanci třídy `JSimLink` a není nutné tuto třídu dědit (uživatel naopak získává možnost, aby třídu pro uživatelská data mohl zdědit od libovolné jiné třídy).

Každá instance třídy `JSimLink` může být vložena nejvýše do jednoho seznamu (fronty) `JSimHead`. Vkládání do seznamu se provádí některou z následujících metod:

- `into()` – vloží prvek do určeného seznamu.
- `follow()` – vloží prvek *za* jiný určený prvek (tento již musí být členem nějakého seznamu).
- `precede()` – vloží prvek *před* jiný určený prvek (tento již musí být členem nějakého seznamu).

Odstranění prvku ze seznamu se provádí voláním `link.out()`, pro prvek to však neznamená konec jeho existence – může být opětovně vložen do (stejně či jiné) fronty nebo využit pro další zpracování. Pokud si uživatel simulace přeje prvek opravdu „zlikvidovat“, je třeba zrušit na něj všechny odkazy⁴, např. `link = null`;

2.2.5 Systém výjimek

Knihovna J-Sim disponuje vlastním systémem výjimek, používaných při výskytu chybových a/nebo neobvyklých situací. Následuje stručný přehled nejdůležitějších výjimek, podrobnější popis lze samozřejmě nalézt v dokumentaci nebo přímo ve zdrojových kódech.

Tyto výjimky jsou odvozeny od třídy `JSimException`⁵.

- `JSimInvalidParametersException` – tato výjimka je vyhozena metodou nebo konstruktorem při pokusu o použití neplatného argumentu (např. neplatný odkaz na objekt simulace předaný konstruktoru třídy `JSimProcess`).
- `JSimSimulationAlreadyTerminatedException` – je vyhozena při pokusu o přidání nového procesu do simulace, která je již ukončena. Simulace je ukončena tehdy, když už nejsou naplánovány žádné události nebo pokud neexistují žádné procesy.
- `JSimTooManyProcessesException` – tato výjimka je vyhozena, pokud již J-Sim nemůže do simulace přidat další nový proces.
- `JSimTooManyHeadsException` – je vyhozena, pokud již J-Sim nemůže do simulace přidat další frontu.
- `JSimInvalidProcessStateException` – tato výjimka je vyhozena metodou `setProcessState()` při pokusu o nepřípustnou změnu stavu procesu (viz 2.1.3).
- `JSimSecurityException` – je vyhozena, pokud se uživatel pokusí provést akci, jejíž provedení je v dané situaci zakázáno (např. pokus o zavolání metody `passivate()` nad procesem, který není ve stavu *aktivní*).

Následující výjimky jsou potomkem třídy `java.lang.RuntimeException`⁶.

- `JSimKernelPanicException` – vyskytuje se v případě vážné chyby jádra J-Sim. Neměla by být zachycena.

⁴Skutečnou fyzickou likvidaci provede až *garbage collector* JVM, ale to již uživatele nemusí zajímat.

⁵Tato třída je přímým potomkem třídy `java.lang.Exception`

⁶Překladač nenutí uživatele, aby na tyto výjimky nějak reagoval.

- `JSimProcessDeath` – interní výjimka J-Sim, používaná při *zabití* procesu. Je propagována z metod `hold()` nebo `passivate()` přes metodu `life()` až do `run()`, kde je zachycena. V žádném případě by neměla být zachycena uživatelem.

2.2.6 Další funkce knihovny J-Sim

Kromě výše vysvětlených funkcí poskytuje J-Sim některé další funkce. Jde například o generování náhodných čísel s požadovaným rozdělením pravděpodobnosti (exponenciální, rovnoměrné, gaussové).

Kapitola 3

High Level Architecture

3.1 Vznik HLA

High Level Architecture (HLA) je softwarová architektura pro vytváření distribuovaných počítačových simulací. Distribuovaná simulace je spolupráce několika simulací, které jsou spojeny do jednoho většího simulačního celku. Standard HLA vznikl původně pro vojenské účely a jedná se tedy o poměrně silný a komplikovaný nástroj.

HLA vznikla na půdě Úřadu pro modelování a simulace¹ Ministerstva obrany USA. HLA existuje ve dvou hlavních verzích. Starší verze, která je označována jako HLA 1.3, byla schválena jako standard Object Management Group (OMG) v roce 1998. Novější verze HLA 1516 byla přijata jako standard IEEE 1516 v roce 2000.

Standard HLA se skládá ze tří částí:

- **Pravidla HLA** (*HLA Rules*) – další informace budou uvedeny v odstavci 3.3
- **Šablona objektového modelu** (*Object Model Template, OMT*) – další informace budou uvedeny v odstavci 3.4
- **Specifikace rozhraní** (*Interface Specification*) – další informace budou uvedeny v odstavci 3.5

Vzhledem k tomu, že problematika HLA je značně rozsáhlá, v tomto textu budou vysvětleny pouze nejdůležitější principy HLA. Podrobnější informace lze nalézt v literatuře ([Kuh00], [CSU1], [CSU2]) a v příslušných částech standardu HLA.

¹Defense Modeling and Simulation Office, <https://www.dmsomil/public/>

3.2 Základní pojmy

HLA definuje tyto základní pojmy, které budou používány v dalším textu:

- **Federace** (*federation*) – simulační systém vytvořený z jednotlivých simulací
- **Člen federace** (*federate*) – každá simulace, která se podílí na tvorbě federace
- **Vykonávání federace** (*federation execution*) – proces společného vykonávání simulace

Každá federace obsahuje následující části:

- Podpůrná softwarová vrstva nazývaná **Runtime Infrastructure (RTI)**
- Objektový model pro výměnu dat mezi členy federace, nazývaný *Federation Object Model (FOM)*
- Několik členů federace. Člen federace je samostatnou jednotkou připojenou k RTI.

3.3 Pravidla HLA

Pravidla HLA jsou tvořena určitými principy a konvencemi, které musí být splněny, aby bylo zajištěno správné vykonávání federace. HLA stanovuje deset pravidel, která jsou rozdělena do dvou skupin po pěti pravidlech. První skupina stanovuje pravidla pro federaci, druhá skupina pro členy federace.

3.3.1 Pravidla federace

- **Pravidlo 1.** Každá federace musí mít FOM, vytvořený v souladu s OMT. FOM slouží jako „slovní zásoba“ pro celou federaci. Popisuje objekty a interakce, které člen federace poskytuje ostatním členům federace.
- **Pravidlo 2.** Všechny instance objektů se musí nacházet v členech federace, nikoliv v RTI.

Toto pravidlo vyjadřuje jednu z nejdůležitějších vlastností návrhu HLA: všechny služby RTI jsou velmi obecné a univerzální, aby mohly být používány pro mnoho různých aplikací. Současně toto pravidlo stanovuje důležité omezení: RTI neslouží jako databáze současných (nebo minulých) hodnot

atributů. Pravidlo je zjednodušeně vyjadřováno tvrzením, že „RTI nezachovává stav“. Implementace RTI sice uchovává řadu stavových informací spojených s jednotlivými službami, ale za hodnoty atributů a parametrů zodpovídají pouze členové federace.

- **Pravidlo 3.** Veškerá výměna dat (definovaných ve FOM) mezi členy federace musí během vykonávání federace probíhat pouze prostřednictvím RTI.

Toto pravidlo znamená, že členové federace spolu nikdy nekomunikují přímo. RTI zajišťuje veškerou výměnu dat mezi členy federace. Cílem tohoto pravidla je návrh univerzálních a znovupoužitelných simulačních komponent.

- **Pravidlo 4.** Členové federace mohou během vykonávání federace s RTI komunikovat pouze v souladu se specifikací rozhraní HLA.

Pravidlo potvrzuje význam specifikace rozhraní HLA. Pro splnění tohoto pravidla nestačí, že členové federace spolu komunikují pouze prostřednictvím RTI, ale navíc musí k této komunikaci používat pouze služby, které jsou uvedeny ve specifikaci rozhraní HLA. Pravidlo chrání členy federace před zvláštnostmi jednotlivých implementací RTI a umožňuje nahrazení RTI jinou implementací.

- **Pravidlo 5.** Každý atribut objektu může být během vykonávání federace v jednom okamžiku vlastněn nejvýše jedním členem federace.

Pravidlo řeší vztah vlastnictví atributu a zodpovědnosti za jeho aktualizaci. Specifikace rozhraní vyžaduje, aby aktualizaci atributu mohl provést pouze člen federace, který tento atribut vlastní. Pokud nebude tato podmínka splněna, RTI aktualizaci atributu zamítne.

3.3.2 Pravidla členů federace

- **Pravidlo 6.** Každý člen federace musí mít SOM, vytvořený v souladu s OMT.

Funkcionalita simulace člena federace, která poskytuje data jiným členům federace, musí být vyjádřena v termínech HLA. Toto vyjádření tvoří simulační objektový model (*Simulation Object Model, SOM*) člena federace.

- **Pravidlo 7.** Každý člen federace musí být schopen aktualizovat nebo reflektovat hodnotu atributu a posílat nebo přijímat interakce, v souladu se specifikací v SOM.

Člen federace musí při použití služeb RTI a při každé reakci na služby iniciované RTI nakládat s interakcemi i atributy v souladu se SOM.

- **Pravidlo 8.** Člen federace musí být schopen dynamicky převést nebo přijímat vlastnictví jednotlivých atributů, v souladu se specifikací v SOM.

Člen federace není pouhým producentem a konzumentem dat definovaných v SOM, ale je zodpovědný za převody vlastnictví dat v souladu s protokoly, které jsou definovány ve specifikaci rozhraní.

- **Pravidlo 9.** Člen federace musí být schopen měnit podmínky, podle kterých jsou zajišťovány aktualizace atributů, v souladu se specifikací v SOM.

SOM udává určité podmínky (uplynutí času, apod.), které způsobí, že člen federace musí provést aktualizaci určitého atributu.

- **Pravidlo 10.** Člen federace musí být schopen řídit simulační čas takovým způsobem, který umožní koordinaci výměny dat s ostatními členy federace.

Toto pravidlo od člena federace vyžaduje, aby používal určitou množinu (i prázdnou) funkcí skupiny *time management* pro řízení logického času a současně umožňoval řízení logického času ostatním členům federace. RTI podporuje několik schémat pro řízení simulačního času a jednotlivým členům federace umožňuje použití různých schémat.

3.4 Object Model Template

Každá federace je popsána objektovým modelem federace (*Federation Object Model, FOM*). Struktura tohoto modelu je předepsána šablonou objektového modelu (OMT). Člen federace musí FOM předat jako parametr pro RTI před zahájením vykonávání federace. Základními prvky OMT jsou **interakce** a **objekty**.

3.4.1 Interakce

Interakce (*interaction*) je kolekce dat poslaná členem federace ostatním členům federace. Interakce může reprezentovat událost v simulačním modelu, jejíž výskyt může zajímat ostatní členy federace. Interakce obsahují data, která se nazývají **parametry**. Interakce může být spojena s určitým bodem simulačního času. Interakce není perzistentní, existence interakce končí doručením příjemci interakce.

Interakce jsou uspořádány do hierarchie tříd s jednoduchou dědičností. Tato hierarchie má jediného společného předka, který se nazývá **InteractionRoot**. Každá interakční třída definuje vlastní parametry. Interakční třídy dědí všechny parametry tříd svých předků. Třída **InteractionRoot** nedefinuje žádné parametry. Postupný zápis jmen interakčních tříd, směrem od třídy **InteractionRoot** ke konkrétní interakční třídě, oddělený tečkami, se nazývá plně kvalifikované jméno (*fully qualified name*) interakční třídy.

3.4.2 Objekty

Objekt (*object*) je simulační entitou, která je předmětem zájmu více než jednoho člena federace. Objekt existuje v intervalu simulačního času, objekt je tedy perzistentní.

OMT definuje třídy objektů. Každá objektová třída má jméno. Objektová třída definuje množinu dat (může být prázdná), která se nazývají **atributy**. Členové federace vytváří instance objektů. Stav konkrétní instance objektu je dán hodnotami atributů objektu.

Objektové třídy tvoří hierarchii s jednoduchou dědičností. Společným předkem všech objektových tříd je třída **ObjectRoot**. Tato třída obsahuje jediný atribut **privilegeToDeleteObject**. Objektové třídy dědí atributy všech svých předků. Z toho vyplývá, že každá objektová třída má alespoň jeden atribut. Každá objektová třída má svoje plně kvalifikované jméno, které je definováno analogickým způsobem jako pro interakční třídy.

3.4.3 Komponenty OMT

Šablona objektového modelu vyžaduje, aby všechny komponenty OMT byly uspořádány v tabulkách. OMT se skládá z následujících tabulek²:

- Tabulka identifikace objektového modelu (*Object model identification table*)
- Tabulka struktury objektových tříd (*Object class structure table*) – obsahuje hierarchii objektových tříd federace.
- Tabulka struktury interakčních tříd (*Interaction class structure table*) – obsahuje hierarchii interakčních tříd federace.
- Tabulka atributů (*Attribute table*) – popisuje vlastnosti atributů objektů federace.
- Tabulka parametrů (*Parameter table*) – popisuje vlastnosti parametrů interakcí federace.
- Tabulka dimenzí (*Dimension table*) – souvisí se službami *Data Distribution Management*.
- Tabulka reprezentací času (*Time representation table*)
- Tabulka uživatelských dodatečných dat (*User-supplied tag table*)
- Tabulka synchronizací (*Synchronization table*)

²Uvedený seznam tabulek OMT odpovídá IEEE 1516.2.

- Tabulka typů přenosu (*Transportation type table*)
- Tabulka přepínačů (*Switches table*)
- Tabulka datových typů (*Datatype tables*)
- Tabulka poznámek (*Notes table*)
- FOM/SOM lexikon (*FOM/SOM lexicon*)

Každý objektový model musí obsahovat alespoň jednu objektovou nebo interakční třídu. Ostatní tabulky mohou být prázdné. Tabulky OMT se zapisují do souboru, jehož formát je daný používanou verzí HLA. Ve starší verzi HLA 1.3 se jedná o formát FED (*Federation Execution Data*). HLA 1516 nahrazuje FED soubory formátem FDD (*FOM Document Data*), který využívá standardního formátu XML. Příklad části FED souboru lze nalézt v odstavci 5.3.1, ukázkou FDD souboru v odstavci 5.3.2.

3.5 Specifikace rozhraní

Specifikace rozhraní obsahuje popis služeb, které jsou používány pro komunikaci mezi členem federace a implementací Runtime Infrastructure. Specifikace rozhraní definuje (ve svých přílohách) aplikační programové rozhraní (API) pro několik programovacích jazyků. Podporovanými jazyky jsou jazyky C++, Java a ADA.

RTI poskytuje členům federace rozhraní nazvané **RTIAmbassador**. Služby tohoto rozhraní jsou volány členem federace, proto se nazývají *federate-initiated services*. Na straně člena federace musí existovat rozhraní **FederateAmbassador**, které umožňuje provedení operací ze strany RTI – takové operace se nazývají *RTI-initiated services*³.

Forma uvedených rozhraní se pro jednotlivá API liší. Pro jazyk C++ je **RTIAmbassador** třídou s metodami, které odpovídají jednotlivým službám volaných členem federace. Rozhraní **FederateAmbassador** má podobu abstraktní třídy jazyka C++, kterou musí překrýt autor programu člena federace. V API pro jazyk Java mají rozhraní **RTIAmbassador** i **FederateAmbassador** podobu rozhraní tohoto jazyka. Za implementaci rozhraní **RTIAmbassador** je zodpovědný dodavatel software RTI, rozhraní **FederateAmbassador** musí implementovat autor programu člena federace.

³Služby iniciované RTI jsou velmi často označovány termínem *RTI callback* nebo pouze *callback*. Ve specifikaci rozhraní HLA bývá jejich název doplněn symbolem †.

3.5.1 Runtime Infrastructure

HLA je softwarová architektura, nikoliv implementace. Software, který implementuje rozhraní `RTIambassador`, se nazývá **Runtime Infrastructure** (RTI). Úkolem tohoto software je poskytnout členům federace služby, které jsou nezbytné pro řízení distribuované simulace.

Služby RTI jsou rozděleny do šesti skupin:

- **Federation Management** – správa federace
- **Declaration Management** – správa deklarací
- **Object Management** – správa objektů
- **Ownership Management** – správa vlastnictví
- **Time Management** – správa času
- **Data Distribution Management** – správa distribuce dat

Nejdůležitější služby budou stručně vysvětleny v následujících odstavcích, které se zabývají jednotlivými skupinami služeb. Názvy služeb budou uváděny podle API pro jazyk Java, parametry služeb budou vynechány v zájmu zachování přehlednosti zápisu. Popis činnosti služeb a jejich parametrů je pouze orientační, podrobné informace o jednotlivých službách lze nalézt přímo ve specifikaci rozhraní HLA 1.3 ([HLA98]) nebo HLA 1516 (standard IEEE 1516.1).

3.5.2 Federation Management

Správa federace definuje služby pro vytvoření federace, připojení k federaci, odhlášení z vykonávané federace a zrušení federace. Kromě těchto služeb do této skupiny patří ještě služby pro synchronizaci členů federace a služby pro uložení a obnovení federace.

Nejdůležitější služby správy federace jsou:

- `createFederationExecution()` – vytvoří federaci. Jako parametry pro tuto službu je třeba zadat jméno federace a odkaz na objektový model federace. Jméno vytvářené federace musí být jedinečné, jinak federace nebude vytvořena.
- `joinFederationExecution()` – připojení k federaci. Tuto službu musí použít každý člen federace, aby se mohl zúčastnit vykonávání federace.
- `resignFederationExecution()` – odhlášení člena federace z vykonávané federace.

- `destroyFederationExecution()` – zrušení federace. Federaci lze zrušit pouze tehdy, pokud již nejsou připojeni žádní členové federace.

Následující služby slouží pro bariérovou synchronizaci federace:

- `registerFederationSynchronizationPoint()` – zaregistrování synchronizačního bodu. Parametrem služby je jméno synchronizačního bodu, které musí být jedinečné v rámci federace. Úspěšná registrace synchronizačního bodu je potvrzena callbackem `synchronizationPointRegistrationSucceeded()`, v případě neúspěchu (např. pokud jméno synchronizačního bodu nebylo jedinečné) RTI pošle callback `synchronizationPointRegistrationFailed()`.
- `synchronizationPointAchieved()` – voláním této metody oznamuje člen federace, že provedl všechny operace nutné pro dosažení synchronizačního bodu. Federace je synchronizována tehdy, když synchronizačního bodu dosáhnou všichni členové federace (příp. množina členů federace uvedená při registraci synchronizačního bodu). Synchronizaci federace oznamuje RTI callbackem `federationSynchronized()`.

3.5.3 Declaration Management

Správa deklarací umožňuje specifikovat druhy dat, která budou předávána při komunikaci mezi členy federace. Člen federace nikdy neposílá data jmenovitě jinému členovi federace, ale federaci. RTI zajišťuje, aby data byla doručena těm členům federace, kteří o tato data mají zájem.

Služby správy deklarací umožňují členovi federace, aby projevil svůj zájem stát se producentem nebo konzumentem určitých dat. Pokud člen federace vytváří data určitého typu, musí tato data **publikovat** (*publish*). Člen federace, který si přeje zpracovávat data určitého typu, se musí **přihlásit k odběru** dat (*subscribe*).

Popis nejdůležitějších služeb správy deklarací je uveden v následujícím seznamu:

- `publishInteractionClass()` – publikuje interakční třídu. Parametrem služby je *handle* dané interakční třídy, který lze zjistit voláním služby `getInteractionClassHandle()` (viz 3.5.8). Pokud člen federace potřebuje posílat interakce určité třídy, musí nejprve oznámit RTI publikování této interakční třídy. V opačném případě RTI odeslání interakce zamítne. Zrušení publikování interakční třídy lze provést voláním `unpublishInteractionClass()`.

- `subscribeInteractionClass()` – přihlásí se k odběru interakční třídy. Parametrem je *handle* interakční třídy. Zrušení odběru interakcí této třídy lze provést voláním `unsubscribeInteractionClass()`.
- `publishObjectClassAttributes()` – publikuje atributy objektové třídy. Jako parametr je třeba předat *handle* objektové třídy a množinu publikovaných atributů. Publikování atributů objektové třídy lze zrušit službou `unpublishObjectClassAttributes()`, publikování celé třídy `unpublishObjectClass()`.
- `subscribeObjectClassAttributes()` – přihlásí se k odběru atributů objektové třídy. Zrušení odběru atributů se provádí voláním `unsubscribeObjectClassAttributes()` nebo pro celou třídu `unsubscribeObjectClass()`.

3.5.4 Object Management

Zatímco správa deklarací slouží pouze ke specifikaci druhů dat, která si budou členové federace předávat, služby správy objektů umožňují výměnu těchto dat. Člen federace využívá služby této skupiny vždy, když posílá nebo přijímá interakci. Do této skupiny patří také služby, které umožňují registraci nové instance objektové třídy, aktualizaci hodnot atributů objektu a odstranění instance objektové třídy.

Pro výměnu dat mezi členy federace slouží následující služby:

- `sendInteraction()` – odešle interakci publikované třídy. Při odesílání interakce je třeba uvést *handle* interakční třídy a množinu parametrů interakce. Členové federace, kteří se přihlásili k odběru této interakční třídy, obdrží RTI callback `receiveInteraction()`.
- `registerObjectInstance()` – zaregistruje instanci objektové třídy. Členové federace, kteří se přihlásili k odběru alespoň jednoho atributu této třídy, obdrží RTI callback `discoverObjectInstance()`.
- `updateAttributeValues()` – aktualizace atributů instance objektové třídy. Členové federace, kteří jsou přihlášení k odběru příslušných atributů, obdrží volání `reflectAttributeValues()`.
- `deleteObjectInstance()` – odstranění instance objektové třídy. Členové federace, kteří se přihlásili k odběru alespoň jednoho atributu této třídy, obdrží RTI callback `removeObjectInstance()`.
- `requestAttributeValueUpdate()` – umožňuje provedení aktualizace atributů „na vyžádání“. Člen federace, který je vlastníkem příslušného atributu,

bude vyzván RTI callbackem `provideAttributeValueUpdate()` k provedení aktualizace.

Při použití služeb `sendInteraction()`, `updateAttributeValues()` a `deleteObjectInstance()` může být uvedeno **časové razítko** (*time stamp*). Časová razítka mají význam v kombinaci s použitím služeb skupiny *time management* (viz 3.5.6).

3.5.5 Ownership Management

Služby správy vlastnictví řeší problém zodpovědnosti za aktualizaci atributů. Pravidla HLA číslo 5 a 8 (viz 3.3) stanovují, že člen federace, který potřebuje provést aktualizaci atributu, musí být vlastníkem tohoto atributu. V opačném případě RTI odmítne aktualizaci atributu provést.

Zodpovědnost za simulační entity může být realizována dvěma způsoby. Prvním způsobem je sdílené vlastnictví mezi jednotlivými členy federace. Pokud je simulační entita reprezentována instancí s více atributy, různí členové federace mohou vlastnit různé atributy. Každý člen federace je potom zodpovědný za aktualizace atributů, které má ve svém vlastnictví.

Druhou možností je postupná změna vlastníků atributů v průběhu vykonávání federace. Vlastnictví atributu totiž může být předáno jinému členovi federace. HLA poskytuje mechanismy pro převod vlastnictví, které mohou být iniciovány současným i budoucím vlastníkem atributu.

Kromě obou uvedených způsobů správy vlastnictví lze služby této skupiny ignorovat, pokud je federace nepotřebuje. Ostatní služby HLA jsou navrženy tak, aby se bez ohledu na správu vlastnictví chovaly způsobem, který od nich lze očekávat.

3.5.6 Time Management

Důležitým problémem, který musí být řešen v průběhu vykonávání federace, je zajištění takového pořadí událostí v „životech“ jednotlivých členů federace, aby byla zajištěna kauzalita celé distribuované simulace. HLA pro členy federace definuje **logický čas** (*logical time*). Každý člen federace má vlastní hodnotu logického času. Logický čas je abstraktní pojem a není spojen s žádnou konkrétní reprezentací nebo jednotkou času.

Služby řízení času mají dva hlavní úkoly:

- Umožňují členům federace posunovat jejich logický čas koordinovaně s ostatními členy federace.

- Řídí pořadí doručování zpráv (událostí) takovým způsobem, aby nemohla nastat situace, kdy člen federace dostane zprávu, která přijde v jeho „minulosti“ (tj. zpráva s časovým razítkem, jehož hodnota je menší než aktuální logický čas).

RTI umožňuje členům federace, aby si zvolili způsob, kterým se budou podílet na řízení simulačního času. Člen federace může být:

- **časově omezený** (*time constrained*) – to znamená, že nemůže posunout vlastní logický čas bez spolupráce s ostatními členy federace.
- **časově regulující** (*time regulating*) – to znamená, že reguluje posunování logického času jiných členů federace.

Kterýkoliv člen federace může být časově omezený, časově regulující, omezený i regulující současně nebo ani omezený ani regulovaný. Po připojení do federace není člen federace ani časově omezený ani časově regulující, o nastavení těchto příznaků musí členové federace požádat RTI.

Pro nastavení časového omezení a časové regulace slouží služby:

- **enableTimeRegulation()** – zapne časovou regulaci člena federace. Úspěšné zapnutí časové regulace člena federace musí být potvrzeno RTI callbackem **timeRegulationEnabled()**, jehož argumentem je nový logický čas tohoto člena federace. Tento čas je vždy větší nebo roven aktuálnímu času člena federace.

Při zapnutí časové regulace je nutné specifikovat hodnotu *lookahead*. Lookahead je (nezáporný) interval logického času, který stanovuje omezení na hodnoty časových razítek odesílaných zpráv. Pokud má člen federace přidělen logický čas t a hodnota jeho lookaheadu je l , znamená to, že nesmí vygenerovat zprávu s časovým razítkem $t_s < t + l$. Pokud se člen federace nachází ve stavu, kdy požádal RTI o zvýšení logického času (viz dále), nesmí vygenerovat zprávu s časovým razítkem, které je menší než součet *požadovaného* času a hodnoty jeho lookaheadu. Cílem uvedených omezení je zabránit doručování zpráv v minulosti jiných členů federace.

Vypnutí časové regulace lze provést voláním **disableTimeRegulation()**, které se nepotvrzuje callbackem RTI. Člen federace přestává být časově regulující ihned po zavolání této služby.

- **enableTimeConstrained()** – zapne časové omezení člena federace. Služba nemá žádné parametry. Zapnutí časového omezení musí být potvrzeno callbackem **timeConstrainedEnabled()**.

Vypnutí časového omezení se provádí voláním **disableTimeConstrained()**, které není ze strany RTI nijak potvrzováno. Člen federace přestává být časově omezený ihned po zavolání této služby.

Jak již bylo uvedeno v odstavci 3.5.4, služby `sendInteraction()`, `updateAttributeValues()` a `deleteObjectInstance()` umožňují uvedení časových razítek. Volání kterékoliv z těchto služeb se obecně označuje jako **zpráva**⁴. HLA rozlišuje dva druhy zpráv podle pořadí doručování:

- **Time Stamp Messages (TSO)** – RTI tyto zprávy uchovává v prioritní frontě, která je upořádána vzestupně podle hodnot časových razítek, uvedených odesílatelem zpráv. RTI zaručuje, že příjemce obdrží tyto zprávy v pořadí časových razítek.
- **Receive Order Messages (RO)** – RTI ukládá tyto zprávy do FIFO fronty podle pořadí jejich doručení od jednotlivých odesílatelů. Příjemce obdrží tyto zprávy v pořadí jejich zařazení do příslušné fronty zpráv (tedy bez ohledu na hodnotu časových razítek).

Aby mohl člen federace **odeslat** zprávu jako TSO, musí být současně splněny následující podmínky:

- Člen federace, který je odesílatelem zprávy, musí být časově regulující.
- Interakční třída nebo atribut objektové třídy odesílané zprávy musí mít v objektovém modelu federace deklarováno pořadí *time stamp order*.
- Při odesílání zprávy musí být uvedena hodnota časového razítka.

Ve všech ostatních případech bude odesílaná zpráva převedena na RO zprávu.

Aby mohl člen federace **přimout** zprávu jako TSO, musí být současně splněny následující podmínky:

- Zpráva byla odeslána jako TSO.
- Člen federace, který je příjemcem zprávy, musí být časově omezený.

V ostatních případech bude zpráva doručena jako RO zpráva.

Z důvodu zajištění správného pořadí doručení TSO zpráv, nemá člen federace dovoleno zvyšovat logický čas libovolným způsobem, ale o každé zvýšení času musí požádat RTI voláním některé z pěti služeb, které jsou k tomuto účelu určeny. Všechny tyto služby mají jediný argument, kterým je požadovaná hodnota logického času. Po odeslání žádosti o zvýšení času se člen federace dostává do stavu *time advancing*, ve kterém setrvává až do okamžiku přidělení nového času od RTI. Během čekání na udělení času nesmí člen federace znovu požádat RTI o čas.

⁴Termín zpráva (*message*) je používán v HLA 1516 a nahrazuje starší termín událost (*event*) z HLA 1.3.

Člen federace, který je časově omezený, dostává zprávy pouze ve stavu *time advancing*. V tomto stavu RTI doručí členovi federace všechny RO zprávy a TSO zprávy, jejichž časové razítko je menší nebo rovno hodnotě přiděleného času. Po doručení těchto zpráv přidělí RTI členovi federace novou hodnotu logického času. Protože člen federace, který není časově omezený, nemůže dostávat TSO zprávy, RTI takovému členu federace doručuje všechny zprávy ihned po jejich přijetí od odesílatele zprávy. Takové zprávy budou vždy doručeny jako RO, bez ohledu na způsob jejich odeslání.

RTI oznamuje členovi federace přidělení času (pro všechny služby, kterými lze požádat o zvýšení času) callbackem `timeAdvanceGrant()`. Argumentem tohoto callbacku je nový logický čas, jehož hodnota může být menší nebo rovna hodnotě požadované členem federace. Doručením callbacku `timeAdvanceGrant()` se člen federace dostává do stavu *time granted*, ve kterém zůstává až do okamžiku odeslání nové žádosti o zvýšení času. Časově omezený člen federace nedostává ve stavu *time granted* žádné zprávy.

V následujícím seznamu je uveden stručný popis služeb, jejichž voláním člen federace může požádat RTI o zvýšení logického času:

- `timeAdvanceRequest()` – při použití této služby bude hodnota přiděleného času rovna hodnotě požadovanému času. Podobným způsobem pracuje také služba `timeAdvanceRequestAvailable()`, která je využívána v případě, že člen federace používá nulový lookahead.
- `nextMessageRequest()` – při použití této služby závisí hodnota přiděleného času na obsahu fronty TSO zpráv a na hodnotě požadovaného času. Pokud TSO fronta není prázdná a hodnota časového razítka první zprávy v této frontě je menší než hodnota požadovaného času, bude přidělena tato hodnota. V opačném případě bude přidělena hodnota, kterou člen federace uvedl jako parametr požadované služby. Podobným způsobem pracuje také služba `nextMessageRequestAvailable()`, která je využívána v případě, že člen federace používá nulový lookahead.
- `flushQueueRequest()` – služba způsobí, že RTI provede okamžité vyprázdnění front všech RO i TSO zpráv (bez ohledu na hodnoty časových razítek). Služba je využívána při distribuované simulaci s optimistickou synchronizací.

3.5.7 Data Distribution Management

Správa distribucí dat (DDM) „vylepšuje“ řízení vztahů producent – konzument mezi členy federace. Zatímco služby správy deklarací umožňují definovat tyto vztahy pouze na úrovni interakčních a objektových tříd, DDM poskytuje možnost

definování těchto vztahů na úrovni jednotlivých instancí objektů a abstraktních směrovacích prostorů.

3.5.8 Podpůrné služby

Kromě výše uvedených šesti skupin služeb existují ještě podpůrné služby (*support services*), které umožňují provádění těchto akcí:

- Konverze jmen na celočíselné identifikátory (*handle*) a obráceně. Příkladem mohou být služby `getInteractionClassHandle()` a `getInteractionClassName()`.
- Nastavení „poradních“ přepínačů (*setting advisory switches*). Tyto přepínače umožňují členům federace, kteří publikují objektové či interakční třídy, zapnout či vypnout přijímání oznámení RTI o tom, že se jiný člen federace přihlásil k odběru těchto tříd. Tato oznámení slouží jako doporučení, aby člen federace zbytečně neprodukoval data, která nikdo nekonsumuje.

Kapitola 4

Přehled implementací Runtime Infrastructure

V současné době existuje pouze několik implementací software Runtime Infrastructure (RTI). Pokud některá implementace splňuje kompletní specifikaci rozhraní HLA, je možné získat certifikát DMSO, který tuto skutečnost potvrzuje. Počet dosud certifikovaných implementací je však velmi nízký, protože testy mají několik fází a testovací kritéria jsou velmi přísná. Mezi výrobce RTI patří společnosti: SAIC, Pitch AB, Mitsubishi Space Software, MAK Technologies a Virtual Technology Corporation.

4.1 pRTI

pRTI je produktem švédské firmy Pitch Technologies AB¹. Implementace pRTI existuje ve dvou produktových řadách. pRTI 1.3 používá starší standard HLA 1.3, pRTI 1516 implementuje specifikaci rozhraní HLA IEEE 1516. Obě řady produktů pRTI mají certifikáty DMSO, které potvrzují, že splňují příslušné specifikace rozhraní HLA. Produkt pRTI 1516 byl dokonce první implementací HLA IEEE 1516, která tento certifikát získala (a v době vzniku tohoto textu také jedinou).

Produkty pRTI jsou napsány v jazyce Java a pro členy HLA federace poskytují rozhraní v jazycích Java a C++. Vzhledem k získaným certifikacím je zřejmé, že se jedná o skutečně kvalitní implementace HLA. Z toho také vyplývá vysoká cena a licenční podmínky, které omezují nejen počet instalací podle počtu zakoupených licencí, ale také maximální počet členů federace.

¹<http://www.pitch.se/>

4.2 Extensible Run-Time Infrastructure

Extensible Run-Time Infrastructure (XRTI) je volně šiřitelná implementace specifikace rozhraní High Level Architecture IEEE 1516.1. Implementace XRTI vznikla na Naval Postgraduate School² (The MOVES Institute) v Kalifornii. Autorem projektu XRTI je Andrzej Kapolka, dalšímu vývoji se však již nevěnuje. Mezi výhody XRTI patří dostupnost zdrojových kódů v jazyce Java a licence, která umožňuje tyto zdrojové kódy dále modifikovat a šířit.

Nevýhodou XRTI je, že se stále nachází ve fázi prototypu, který implementuje pouze určitou podmnožinu rozhraní HLA. XRTI nepodporuje žádné služby skupin: Time Management, Ownership Management, Data Distribution Management. Ostatní skupiny služeb HLA neobsahují některé služby nebo jsou oproti specifikaci rozhraní HLA částečně zjednodušeny.

Přestože XRTI neimplementuje kompletní specifikaci rozhraní HLA, jedná se o velmi rozsáhlý projekt. XRTI se skládá z více než 60 tříd a rozhraní. Celková velikost XRTI přesahuje 27 tisíc řádek zdrojových kódů³.

Je třeba poznamenat, že možnosti výběru implementace RTI jsou velmi omezené. V úvahu připadá buď nutnost zakoupení licence na některou komerční implementaci nebo použití XRTI. Vzhledem k požadavkům v zadání této práce a předpokladu, že simulační knihovna J-Sim bude využívat pouze určitou podmnožinu služeb HLA, byla pro další práci použita implementace XRTI.

²<http://www.npsnet.org/>

³Uvedená čísla nezahrnují třídy a rozhraní, která jsou generována automaticky při překladu XRTI. Rovněž není započítán balík `hla.rti`, který je součástí specifikace rozhraní HLA pro jazyk Java.

Kapitola 5

Simulační knihovna J-Sim s podporou HLA

5.1 Požadavky na řešení

Hlavním požadavkem na implementaci podpory HLA pro knihovnu J-Sim je vznik takového systému, který je maximálně kompatibilní s předchozími nedistribuovanými verzemi knihovny J-Sim.

Dalším požadavkem je, aby uživatel knihovny J-Sim byl schopen navrhnout distribuovanou simulaci i v případě, že nemá hluboké znalosti problematiky HLA. Prakticky to znamená, že použití distribuované simulace nesmí být příliš komplikované. Tento požadavek by měl být splněn nejen pro návrh „nové“ distribuované simulace, ale i pro případnou změnu nedistribuované simulace na distribuovanou (a naopak).

Rovněž je třeba zmínit, že podpora HLA již byla implementována do simulační knihovny C-Sim ([Jir04]). Zřejmě by tedy bylo vhodné, aby řešení pro J-Sim nabízelo uživateli navenek stejné možnosti a vlastnosti jako dříve realizované řešení pro C-Sim, přestože je zřejmé, že v implementaci budou poměrně značné odlišnosti¹.

5.2 Princip řešení

Každá diskrétní simulace navržená s využitím J-Sim vyžaduje existenci simulačního objektu – instance třídy `JSimSimulation`. Všechny procesy (a podobně také

¹Tyto rozdíly jsou dané především odlišnostmi v implementaci knihoven C-Sim a J-Sim a jejich kořeny je třeba hledat v rozdílné „filosofii“ jazyků C a Java.

fronty), které se účastní této simulace, musí mít jako svého rodiče² uveden odkaz na tento simulační objekt. V jednom programu typicky existuje jeden simulační objekt, který „žije vlastním životem“ a nespolupracuje s objekty jiných simulací.

Distribuovaná simulace je technika spolupráce jednotlivých simulací, které jsou současně vykonávány na více procesorech nebo počítačích. Tyto počítače jsou za účelem spolupráce propojeny počítačovou sítí. Jednotlivé počítače, na kterých simulace běží, mohou být umístěny v geograficky různých lokalitách.

Distribuovaná simulace s využitím simulační knihovny J-Sim tedy znamená, že existuje více instancí (obecně umístěných na více počítačích) třídy **JSimSimulation**, které jsou schopny spolu určitým způsobem spolupracovat. Nutnou podmínkou zajištění spolupráce jednotlivých účastníků je schopnost vzájemné komunikace mezi jednotlivými simulacemi. Prakticky to znamená, že každá J-Sim simulace, která se účastní distribuované simulace, by měla být schopna pracovat nejen se svými vlastními procesy (tedy procesy, u kterých byla při jejich vytvoření uvedena tato simulace jako jejich rodič), ale také s procesy ostatních simulačních objektů v distribuované simulaci. Kromě procesů by měla být schopna podobně pracovat s frontami (příp. s dalšími objekty) ostatních účastníků distribuované simulace.

Z pohledu HLA lze na distribuovanou simulaci nahlížet jako na federaci. Členy této federace budou jednotlivé J-Sim simulace, které se budou distribuované simulace účastnit. Každému simulačnímu objektu tedy bude přiřazen právě jeden člen federace. Tento člen federace se před zahájením simulace připojí do federace, během simulace komunikuje s ostatními členy federace a po ukončení simulace opustí federaci. Standard HLA vyžaduje, aby členové federace spolu nekomunikovali přímo, ale pouze prostřednictvím služeb Runtime Infrastructure (RTI).

Každý simulační objekt je během simulace i nadále zodpovědný za svoje vlastní objekty (procesy, fronty). Tyto objekty budou v dalším textu označovány jako **lokální objekty**. Objekty, které se nachází v jiném členovi federace (a mají tedy jako svého rodiče uvedený jiný simulační objekt) budou nazývány jako **vzdálené objekty** (*remote objects*).

Pokud některý lokální objekt potřebuje provést nějakou akci nad jiným lokálním objektem (např. naplánování jiného procesu v téže simulaci), jejich spolupráce proběhne stejným způsobem jako v nedistribuované simulaci. Z pohledu jazyka Java se jedná o pouhé zavolání metody jiného objektu.

Jiná situace nastane v případě, že lokální objekt požaduje spolupráci s objektem, který je z pohledu prvního objektu vzdáleným objektem. Příkladem takové situace může být požadavek na naplánování vzdáleného procesu. Takový požadavek je třeba nejprve doručit členovi federace, ve kterém se tento objekt nachází jako lokální. Příjemce požadavku následně odešle původnímu objektu odpověď,

²Termín *rodič* je v tomto textu používán ve významu *parent simulation object*. Nejedná se o rodičovskou třídu z terminologie objektově orientovaného programování.

která obsahuje informaci o splnění jeho požadavku (příp. informaci o výskytu chyby). K odesílání i příjmu veškeré komunikace je třeba využít příslušných služeb RTI.

Je zřejmé, že ke spolupráci členů federace dochází v okamžiku, kdy některý člen federace požaduje provedení operace s objektem jiného člena federace. To znamená, že lze využít simulaci řízenou událostmi. Pro zajištění kauzality bude každá událost opatřena časovým razítkem. Jednotliví členové federace nemohou posunovat simulační čas libovolně, ale vždy pouze se souhlasem RTI. Podle HLA se jedná o konzervativně synchronizovanou federaci.

5.3 Komunikační protokol člena federace

Návrh komunikačního protokolu federace vychází z množiny služeb knihovny J-Sim, které lze volat nad vzdáleným objektem. Na každé volání služby nad vzdáleným objektem lze obecně nahlížet jako na **požadavek**. Podle druhu požadavku mohou být spolu s identifikací požadavku předány určité argumenty. Příkladem může být požadavek na naplánování vzdáleného procesu, kde argumentem požadavku je čas naplánování procesu.

Některé služby J-Sim vrací výsledek určitého datového typu, jiné služby žádná data nevrací (příslušná metoda má jako návratový typ uveden `void`). Vzhledem k tomu, že metoda služby bude volána v jiném členovi federace, volající člen federace potřebuje získat alespoň informaci, zda služba proběhla v pořádku nebo zda došlo k nějaké chybě (výjimce). To znamená, že při komunikaci mezi členy federace každá služba J-Sim musí vrátit nějaký výsledek. Tento výsledek lze považovat za **odpověď**, která je povinná. Navrhovaný protokol tedy bude využívat obvyklé schéma požadavek – odpověď.

Přehled všech služeb J-Sim (včetně dat požadavků a odpovědí), které využívají komunikační protokol člena federace, je uveden v následující tabulce:

Třída	Služba	Data požadavku	Data odpovědi
JSimProcess	activate	double	—
	cancel	—	—
	getState	—	int
	isIdle	—	boolean
JSimHead	empty	—	boolean
	cardinal	—	long
	first	—	JSimLink
	last	—	JSimLink
	getLw	—	double
	getTw	—	double
	getTwForAllLinks	—	double
JSimLink	into	JSimLink	—

V tabulce nejsou uvedena data, která jsou společná pro všechny požadavky (resp. odpovědi). Jedná se zejména o jméno objektu (např. jméno procesu), nad kterým bude požadovaná služba provedena a některé další informace potřebné pro identifikaci požadavku (odpovědi). Tato data budou uvedena v hlavičce požadavku (resp. odpovědi). Struktura hlavičky požadavku i odpovědi bude popsána v následujícím odstavci.

5.3.1 Návrh FED souboru pro RTI

Každá federace je popsána FED souborem, který je společný pro všechny členy federace. Implementace RTI potřebuje tento soubor pro vytvoření federace. FED soubor obsahuje definice objektových tříd a interakcí, které jsou používány jednotlivými členy federace při vzájemné komunikaci.

Členové federace J-Sim pro doručování požadavků a odpovědí využívají **třídy interakcí** (podobně jako u řešení pro C-Sim). Každá interakce může obsahovat **parametry**. Specifikace FED souboru neumožňuje uvedení datového typu parametrů, interpretace parametrů tedy závisí na programátorovi simulace. Jednotlivé třídy interakcí lze navrhnout různými způsoby. Jednou z možností je vytvořit pro každý druh požadavku (resp. odpovědi) samostatnou třídu interakcí s parametry, které odpovídají datům požadavku (resp. odpovědi). Takový FED soubor by obsahoval relativně velký počet tříd interakcí s různými parametry. Jinou možností je rozdělení požadavků (odpovědí) do několika kategorií.

Kritériem pro rozdělení požadavků do kategorií může být datový typ parametrů nebo druh služby, kterou požadavek provádí (tento návrh využívá řešení pro C-Sim). Pro J-Sim se také nabízí možnost rozdělení požadavků podle třídy, se kterou požadavek pracuje. To by znamenalo navrhnout jednu třídu interakcí pro požadavky nad procesy, další pro fronty a pro prvky front. Další třídy interakcí by vznikly pro odpovědi na požadavky.

Při návrhu řešení pro J-Sim byla nakonec zvolena jiná varianta, která umožnila použití menšího počtu interakčních tříd. Z toho vyplývá určité zjednodušení FED souboru a také menší počet tříd, které členové J-Sim federace musí objednat (*subscribe*) a publikovat (*publish*). FED soubor definuje pouze tři nové interakční třídy:

- **JSimRequest** – požadavek. Má dva parametry: *header* (hlavička požadavku) a *data* (data požadavku).
- **JSimResponse** – odpověď na požadavek. Podobně jako požadavek má dva parametry: *header* (hlavička odpovědi) a *data* (data odpovědi).
- **JSimEndOfSimulation** – zpráva o ukončení simulace.

Hlavička požadavku je povinná – to znamená, že při vytvoření požadavku musí být vyplněna. Hlavička obsahuje následující údaje: identifikace odesílatele (*handle* člena federace), jméno objektu (např. jméno procesu) a identifikaci služby (např. naplánování procesu). Typ dat požadavku závisí na tom, zda jsou s požadovanou službou spojeny nějaké argumenty a jakého jsou typu. Pokud s požadovanou službou nejsou spojeny žádné argumenty, data požadavku zůstanou nevyplněna.

Hlavička odpovědi je povinná a obsahuje identifikaci odesílatele požadavku, identifikaci odesílatele odpovědi a výsledek provedené služby. Typ dat odpovědi závisí na tom, jaká data vrací volaná služba. Například funkce pro zjištění délky fronty `cardinal()` vrací typ `long`. Pokud volaná služba nevrací žádnou hodnotu, zůstanou data odpovědi nevyplněna.

Následující výpis obsahuje část návrhu FED souboru pro J-Sim (výpis je zkrácen pouze na část, která definuje třídy interakcí pro J-Sim):

```
(class JSimRequest reliable timestamp
  (parameter header)
  (parameter data)
)
(class JSimResponse reliable timestamp
  (parameter header)
  (parameter data)
)
(class JSimEndOfSimulation reliable timestamp
  (parameter data)
)
```

Interakce pro J-Sim jsou definovány jako *reliable*, což znamená, že RTI garantuje doručení každé interakce a současně jako *timestamp*, což umožňuje doručení interakcí v pořadí časových razítek. Navržený FED soubor zůstává

stejný pro všechny distribuované J-Sim simulace. Formát FED souboru podporují všechny produkty RTI, které implementují specifikaci rozhraní HLA 1.3. Knihovna J-Sim však pracuje s implementací Extensible Runtime Infrastructure (XRTI), která místo FED souboru používá soubory FDD. Návrh FDD souboru bude popsán v následujícím odstavci.

5.3.2 Návrh FDD souboru pro XRTI

Implementace XRTI, která je používána s J-Sim, implementuje specifikaci rozhraní HLA 1516.1. Tato verze HLA nahrazuje dříve užívaný formát FED standardním formátem XML. Soubory s objektovými modely federace ve formátu XML jsou označovány jako FDD (*FOM Document Data*). FDD soubory se nachází v podadresáři **resources** distribuce XRTI.

XRTI umožňuje použití datových typů u atributů objektových tříd a parametrů interakcí v FDD souborech. Možnost interpretace parametrů podle uvážení programátora však samozřejmě zůstává zachována – u parametrů interakce stačí uvést `dataType="HLAopaqueData"`.

FDD soubor pro J-Sim lze vytvořit přepsáním³ FED souboru do formátu XML. Návrh FDD souboru pro J-Sim je uveden v následujícím výpisu:

```
<?xml version="1.0"?>
<objectModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="HLA.xsd"
  DTDversion="1516.2"
  name="JSim Object Model"
  type="FOM"
  version="1.0">

  <objects>
    <objectClass name="HLAobjectRoot"
      sharing="Neither">
      ...

    </objectClass>
  </objects>

  <interactions>
    <interactionClass name="HLAinteractionRoot"
      sharing="Neither"
```

³FED soubor pro J-Sim není příliš dlouhý a jeho přepsání do formátu XML lze bez problémů provést ručně. Navíc se jedná o jednorázovou záležitost, protože model federace zůstává stejný pro všechny J-Sim simulace. Pro větší FED soubory nebo pro častější konverzi by pochopitelně bylo vhodnější použít program, který konverzi provede.

```
        dimensions="NA"
        transportation="HLAreliable"
        order="Receive">

<!-- Interaction classes for J-Sim -->

<interactionClass name="JSimRequest"
    sharing="PublishSubscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="TimeStamp"
    semantics="JSim Request">
    <parameter name="header"
        dataType="HLAopaqueData"
        semantics=""/>
    <parameter name="data"
        dataType="HLAopaqueData"
        semantics=""/>
</interactionClass> <!-- JSimRequest -->

...

</interactionClass>
</interactions>
```

Výpis byl značně zkrácen. Kompletní dokument lze nalézt v souboru `JSimObjectModel.xml` v adresáři `resources/cz/zcu/fav/kiv/jsim` distribuce XRTI.

5.4 Realizace řešení

5.4.1 Třída `JSimHLASimulation`

Třída `JSimHLASimulation` patří do nově vytvořeného balíku `cz.zcu.fav.kiv.jsim.hla`. Je odvozena od třídy `JSimSimulation` a rozšiřuje schopnosti rodičovské třídy o možnost použití distribuované simulace.

Konstruktor této třídy vytvoří simulační objekt pojmenovaný podle hodnoty parametru `name` (stejně jako u třídy předka). Kromě parametru `name` má konstruktor ještě další dva parametry:

```
public JSimHLASimulation(String name,
                        Properties configurationRTI,
                        int numberOfFederates)
    throws JSimInvalidParametersException
```

Parametr `configurationRTI` umožňuje zadání konfiguračních parametrů pro implementaci RTI. Pro většinu simulací lze uvést pouze `new Properties()` bez dalšího nastavování, což znamená, že se použijí výchozí hodnoty. Celočíselný parametr `numberOfFederates` udává počet členů federace, kteří se budou podílet na distribuované simulaci. Hodnota parametru menší než dva nemá smysl a pokus o zadání takové hodnoty způsobí vyhození výjimky.

Při zadávání počtu členů federace je třeba si uvědomit, že nastavení této hodnoty může provést pouze člen federace, který federaci vytvořil. Tento člen federace bude následně čekat na připojení zadaného počtu členů federace. Případný pokus o zadání jiné hodnoty po vytvoření federace již nemá na vytvořenou federaci žádný vliv. Vzhledem k tomu, že kód jednotlivých členů federace může být spouštěn současně na více počítačích, není možné předem stanovit, který člen federace tuto hodnotu skutečně nastaví. Z tohoto důvodu je vhodné, aby u všech účastníků distribuované simulace byla zadána stejná hodnota.

Kromě nastavení jména simulace a počtu členů federace plní konstruktor ještě dva důležité úkoly. Prvním z nich je nastavení atributu `simulationHLA` na hodnotu `true`. Tento atribut ovlivňuje chování metody `step()` při použití distribuované simulace (viz 5.4.11). Druhým úkolem je vytvoření instance třídy `JSimHLAConnection`. Tato třída zajišťuje propojení mezi J-Sim a RTI. Podrobněji bude popsána v samostatné kapitole (5.4.10).

Třída `JSimHLASimulation` poskytuje uživateli dvě důležité metody:

- `beginFederationExecution()` – Zahajuje vykonávání federace pro J-Sim. Musí být zavolána *po* vytvoření všech lokálních i vzdálených objektů simulace a *před* prvním voláním metody `step()`. Metoda volá pouze metodu `beginFederationExecution()` instance třídy `JSimHLAConnection`.
- `endFederationExecution()` – Ukončuje vykonávání federace pro J-Sim. Musí být zavolána *po* posledním voláním metody `step()`. Metoda volá pouze metodu `endFederationExecution()` instance třídy `JSimHLAConnection`.

Třída `JSimHLASimulation` obsahuje ještě několik metod, které uživatel simulace nepotřebuje přímo volat. Význam takových metod bude vysvětlen u popisu tříd, které tyto metody volají. Pokud jsou tyto metody volány pouze z ostatních tříd stejného balíku, jsou označeny jako `protected`. Některé metody však musí být přístupné z jiného balíku (např. pro třídu `JSimSimulation`), a proto jsou označeny jako `public`, ačkoliv by je uživatel volat neměl. Uživatel je na tuto skutečnost upozorněn v dokumentačním komentáři metody.

5.4.2 Třída JSimHLARemoteProcess

Třída JSimHLARemoteProcess reprezentuje **vzdálený proces**. Tento proces existuje jako instance třídy JSimProcess (nebo instance potomka třídy JSimProcess) v simulačním programu jiného člena federace.

```
public JSimHLARemoteProcess(String name, JSimHLASimulation parent)
    throws JSimInvalidParametersException,
    JSimSimulationAlreadyTerminatedException
```

Konstruktor třídy JSimHLARemoteProcess vyžaduje zadání stejných parametrů jako konstruktor třídy JSimProcess. Rozdíl je v datovém typu parametru parent, kde je vyžadován odkaz na instanci třídy JSimHLASimulation místo třídy JSimSimulation.

Každý vzdálený proces se při vytvoření automaticky vloží do seznamu vzdálených procesů svého rodičovského simulačního objektu. K tomu slouží metoda addProcess() instance třídy JSimHLASimulation.

Instance třídy JSimHLARemoteProcess nabízí uživateli následující metody:

- activate() – naplánuje vzdálený proces na požadovaný čas.
- cancel() – zruší naplánování vzdáleného procesu.
- isIdle() – zjistí, zda vzdálený proces může být naplánován.
- getState() – zjistí stav vzdáleného procesu.

Metody vzdálených procesů vrací stejné výsledky jako metody lokálních procesů. V případě výskytu chyby také vyhazují stejné výjimky. Použití všech výše uvedených metod z pohledu uživatele je tedy naprosto stejné, jako použití stejně pojmenovaných metod třídy JSimProcess.

5.4.3 Třída JSimHLARemoteHead

Třída JSimHLARemoteHead reprezentuje **vzdálenou frontu**. Tato fronta existuje jako instance třídy JSimHead v simulačním programu jiného člena federace.

Konstruktor třídy JSimHLARemoteHead vyžaduje stejné parametry jako konstruktor třídy JSimHead. Jediný rozdíl je u parametru parent, který je typu JSimHLASimulation místo JSimSimulation.

Podobně jako vzdálený proces se vzdálená fronta při vytvoření automaticky vloží do seznamu vzdálených front svého rodičovského simulačního objektu. K tomu slouží metoda addQueue() instance třídy JSimHLASimulation.

Instance třídy JSimHLARemoteHead nabízí uživateli následující metody:

- `empty()` – test, zda je vzdálená fronta prázdná.
- `cardinal()` – vrací počet prvků ve vzdálené frontě.
- `first()` – vrací první prvek vzdálené fronty.
- `last()` – vrací poslední prvek vzdálené fronty.
- `getLw()` – vrací střední délku vzdálené fronty.
- `getTw()` – vrací střední dobu čekání ve vzdálené frontě pro všechny prvky, které byly odstraněny z fronty.
- `getTwForAllLinks()` – vrací střední dobu čekání ve vzdálené frontě pro všechny prvky, které byly vloženy do fronty.

Všechny metody vzdálené fronty vrací stejné výsledky jako metody běžné (lokální) fronty. Uživatel simulace tedy může používat uvedené metody stejným způsobem, jakým dosud používal stejně pojmenované metody třídy `JSimHead`.

5.4.4 Třídy `JSimHLARequest`, `JSimHLARequestHeader`

Třída `JSimHLARequest` reprezentuje požadavek, který třída `JSimHLAConnection` používá při komunikaci s RTI. Každý požadavek se skládá z povinné hlavičky typu `JSimHLARequestHeader` a dat požadavku typu `Object`. Atributy třídy `JSimHLARequestHeader` jsou úmyslně označeny jako `public`, protože třída je využívána pouze jako analogie typ záznam (nebo struktura, např. `struct` v C). Hlavička každého požadavku musí obsahovat následující údaje:

- `int requestSender` – celočíselný *handle* člena federace, který odeslal požadavek.
- `String objectName` – jméno J-Sim objektu. Služba J-Sim bude vykonána nad objektem s tímto jménem.
- `int serviceCode` – kód služby J-Sim (např. naplánování procesu). Tato služba bude vykonána nad výše uvedeným objektem.

Data poslaná s požadavkem (i s odpovědí) mohou být obecně různého typu. To znamená, že před odesláním požadavku je třeba data vhodným způsobem „zabalit“ a při příjmu požadavku je převést do původní podoby. K tomuto účelu lze použít mechanismus, nazývaný **serializace objektů**, který je v jazyce Java

přímo podporován⁴. Třída objektu, který bude serializován, musí implementovat rozhraní `java.io.Serializable`. Dále musí serializaci podporovat všechny nestatické objekty, na které třída odkazuje. Implementace rozhraní `Serializable` nepředstavuje složitý problém, protože rozhraní neobsahuje žádné metody. Jedná se o tzv. příznakové rozhraní (*tag interface*), které metodám provádějícím serializaci pouze oznamuje, aby serializaci nebránily. Případný pokus o serializaci objektu, který toto rozhraní nepodporuje, skončí vyhozením výjimky `NotSerializableException`.

Při serializaci objektu budou automaticky uloženy hodnoty všech atributů, které jsou deklarovány jako primitivní datové typy (např. `int`). Rozhraní `Serializable` je také implementováno třídami, které obalují primitivní datové typy (např. `Double`) a třídou `String`. Serializaci objektu lze provést voláním metody `writeObject()` třídy `ObjectOutputStream`. Metoda má jediný argument typu `Object`. Postup serializace ilustruje následující úsek kódu:

```
byte[] values = null;

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);

oos.writeObject(object);
oos.close();

values = baos.toByteArray();
```

Po provedení uvedeného kódu je obsah objektu uložen v poli typu `byte[]`. Získané pole lze použít jako parametr interakce, posílané metodou `sendInteraction()`. Při doručení interakce lze z pole opět získat původní objekt (po správném přetypování).

Pro odeslání požadavku by bylo výhodné, aby bylo možné serializovat hlavičku i případná data požadavku. Z výše uvedených typů atributů třídy `JSimHLARequestHeader` je zřejmé, že hlavičku lze serializovat (pochopitelně je nutné uvést, že třída implementuje rozhraní `Serializable`). Zbývá tedy vyřešit serializaci dat požadavků. Pohledem do tabulky komunikačního protokolu člena federace (viz 5.3) se lze přesvědčit, že data všech požadavků i odpovědí patří mezi datové typy, které lze serializovat. Jedinou výjimkou je třída `JSimLink` – tento problém bude vyřešen v kapitole 5.4.13.

⁴Problematika serializace objektů je poměrně rozsáhlá. V tomto textu jsou uvedeny pouze základní principy serializace, podrobnější informace lze nalézt v literatuře [Spe02].

5.4.5 Třídy `JSimHLAResponse`, `JSimHLAResponseHeader`

Třída `JSimHLAResponse` reprezentuje odpověď, používanou při komunikaci mezi třídou `JSimHLAConnection` a RTI. Podobně jako požadavek se odpověď skládá z povinné hlavičky typu `JSimHLAResponseHeader` a dat odpovědi typu `Object`. Hlavička odpovědi musí obsahovat následující údaje:

- `int requestSender` – celočíselný *handle* člena federace, který odeslal požadavek. Člen federace, který čeká na odpověď, musí tuto hodnotu porovnat s hodnotou, kterou uvedl ve svém požadavku. Pokud by tyto porovnání nebylo provedeno, mohla by jako výsledek operace být přijata odpověď, která byla určena pro jiného člena federace.
- `int responseSender` – celočíselný *handle* člena federace, který odeslal odpověď.
- `int errorCode` – výsledek provedené operace. Pokud byla služba provedena bez výskytu chyby, je vrácena hodnota `JSIM_RESULT_OK`, v opačném případě `JSIM_RESULT_EXCEPTION`.

Podle typů atributů třídy `JSimHLAResponseHeader` je zřejmé, že hlavičku lze serializovat (pochopitelně je nutné uvést, že třída implementuje rozhraní `Serializable`). Podobně jako data požadavků lze serializovat i data odpovědí.

5.4.6 Třída `JSimHLAReceivedInteractionCallback`

Třída `JSimHLAReceivedInteractionCallback` zapouzdřuje data o interakci doručené od RTI. Instance této třídy je vytvořena při každém volání metody `receiveInteraction()` instance třídy `JSimHLAConnection`. Následně je uložena do fronty typu `JSimHLACallbackQueue` (viz 5.4.7), kde čeká na další zpracování.

5.4.7 Třída `JSimHLACallbackQueue`

Tato třída slouží jako fronta pro doručené interakce. Jedná se o FIFO frontu, kterou lze snadno realizovat použitím třídy `java.util.LinkedList`. Metody pro manipulaci s prvky fronty jsou označeny jako `synchronized`, což zajišťuje bezpečný přístup k frontě v případě souběžného přístupu z více vláken. Prvky této fronty jsou instance třídy `JSimHLAReceivedInteractionCallback` (viz 5.4.6).

5.4.8 Třída JSimHLABarrier

Třída `JSimHLABarrier` slouží k vytvoření bariéry, kterou lze využít v okamžiku, kdy je další činnost programu podmíněna výskytem určité události, která může být doručena jiným vláknem programu. Typické použití bariéry nastává při volání služby HLA, jejíž provedení musí být potvrzeno doručením callbacku od RTI.

Pro použití bariéry jsou nejdůležitější metody `await()` a `lower()`. První z těchto metod zahajuje čekání na objektu bariéry. Druhá metoda provede dosažení („shození“) bariéry. K implementaci bariéry jsou využívány metody pro synchronizaci vláken, které jsou k dispozici pro každý objekt v jazyce Java (jsou deklarovány ve třídě `Object`). Metoda `await()` opakovaně volá metodu `wait()`, která způsobí čekání aktuálního vlákna až do jeho „probuzení“. K probuzení čekajícího vlákna se využívá metoda `notifyAll()`, která je volána z metody `lower()` při dosažení bariéry.

5.4.9 Třída JSimHLAFederateAmbassador

Tato třída implementuje rozhraní `hla.rti.FederateAmbassador`. Toto rozhraní je součástí standardu HLA a musí být implementováno každým členem federace, který bude využívat služby HLA. Třída implementující toto rozhraní slouží k příjmu callback zpráv od RTI. Příkladem callbacku může být metoda `receiveInteraction()`, která slouží pro příjem interakce poslané jiným členem federace.

Metody, které jsou potřebné pro komunikaci probíhající od RTI směrem k J-Sim, volají stejně pojmenovanou metodu třídy `JSimHLAConnection`. Ostatní metody jsou z pohledu J-Sim nevyužité a neobsahují tedy žádný kód, ale přesto musí být v kódu třídy uvedeny⁵.

5.4.10 Třída JSimHLAConnection

Třída `JSimHLAConnection` zajišťuje spojení mezi J-Sim a RTI. Při vytvoření HLA simulace se automaticky vytvoří instance třídy `JSimHLAConnection`. Uživatel simulace nepotřebuje vytvářet instance této třídy ani přímo volat její metody.

Většina metod třídy `JSimHLAConnection` je označena specifikátorem přístupu `protected`, protože jsou volány převážně ze tříd `JSimHLASimulation` a `JSimHLAFederateAmbassador`, které se nachází ve stejném balíku.

V kapitole 5.4.1 bylo uvedeno, že metody `beginFederationExecution()` a `endFederationExecution()` instance třídy `JSimHLASimulation` pouze volají

⁵Třída v jazyce Java, která implementuje rozhraní, musí vždy implementovat všechny metody tohoto rozhraní.

stejně pojmenované metody třídy `JSimHLAConnection`. Úkolem první z těchto metod je navázání spojení mezi J-Sim a RTI a příprava na vykonávání federace. Druhá metoda slouží ke korektnímu ukončení federace a následnému uzavření spojení mezi J-Sim a RTI. Činnost těchto metod bude nyní popsána podrobněji.

5.4.10.1 Zahájení vykonávání federace

Metoda `beginFederationExecution()` postupně volá několik metod, které jsou nutné pro navázání spojení mezi J-Sim a RTI a pro přípravu na vykonávání federace. Všechny tyto metody jsou `private`, protože jsou používány pouze v rámci své vlastní třídy.

Metoda `initRTIAmbassador()` vytváří instanci třídy `XRTIAmbassador`. Tato třída implementuje rozhraní `hla.rti.RTIAmbassador`, které je součástí specifikace rozhraní HLA. Instance této třídy, pojmenovaná `rtiAmbassador`, bude dále využívána k volání služeb RTI. Po úspěšném vytvoření objektu `rtiAmbassador` následuje vytvoření instance třídy `JSimHLAFederateAmbassador` (viz 5.4.9). Instance této třídy slouží pro příjem callbacků od RTI. Pokud se instanci třídy `rtiAmbassador` nepodaří vytvořit, znamená to, že došlo k chybě při inicializaci rozhraní RTI a provádění dalších akcí nemá smysl. Příčinou této chyby obvykle bývá skutečnost, že se nepodařilo navázat síťové připojení k aplikaci `XRTIExecutive`, která slouží jako centrální server pro distribuovanou simulaci (případně tato aplikace vůbec nebyla spuštěna).

Metoda `createFederationExecution()` se pokusí vytvořit federaci pro J-Sim. Zde je třeba zadat jméno federace a objektový model federace (viz také 5.3.2):

```
rtiAmbassador.createFederationExecution(  
    JSIM_FEDERATION_EXECUTION_NAME,  
    getClass().getResource(JSIM_OBJECT_MODEL)  
);
```

Neúspěch této metody nemusí nutně signalizovat chybu. Pokud je vyhozena (a následně zachycena) výjimka `FederationExecutionAlreadyExists`, znamená to, že federaci již vytvořil jiný člen federace a proces inicializace může pokračovat. V tomto okamžiku se také rozhoduje o správci federace. Pokud se federaci podaří vytvořit, příznak `isFederationManager` bude nastaven na `true`, jinak bude mít hodnotu `false`. Správce federace má oproti ostatním členům federace navíc určité povinnosti (a žádná práva). Zejména se jedná o odpovědnost za úvodní synchronizaci federace.

Protože člen J-Sim federace potřebuje přijímat a odesílat TSO zprávy, musí být současně **časově omezený** (*time constrained*) i **časově regulující** (*time regulating*). Člen federace není po přihlášení do federace ani regulující ani omezený.

To znamená, že je potřeba zavolat služby RTI, které provedou zapnutí těchto přepínačů. HLA sice nestanovuje závazné pořadí, ve kterém musí být tyto operace vykonány, přesto pro jejich provedení existuje určité doporučení ([Kuh00], str. 98). Toto doporučení říká, že zapnutí těchto přepínačů je vhodné provést před provedením publikací a přihlášení k odběru interakcí a atributů objektových tříd. Nastavení těchto přepínačů provádí *private* metody `enableTimeConstrained()` a `enableTimeRegulation()`, které volají stejně pojmenované metody nad objektem `rtiAmbassador`.

Volání služby `rtiAmbassador.enableTimeConstrained()` nevyžaduje žádné argumenty. Je však nutné počkat na doručení callbacku `timeConstrained-Enabled()`, které potvrzuje, že časové omezení člena federace bylo zapnuto. Při čekání na callback od RTI se využívá tzv. **bariéra** (podrobně viz 5.4.8).

Pro zapnutí přepínače časově regulující je třeba specifikovat *lookahead*. Jak již bylo dříve uvedeno (5.3), členové J-Sim federace spolu komunikují způsobem požadavek – odpověď. Člen federace po odeslání TSO zprávy s požadavkem vždy čeká na doručení TSO zprávy s odpovědí. Všichni členové J-Sim federace tedy používají nulovou hodnotu lookaheadu a žádají o přidělení času voláním služby `nextMessageRequestAvailable()`, aby byli schopni odeslat odpověď na požadavek se stejnou časovou značkou, jaká byla uvedena u přijatého požadavku. Zapnutí časové regulace musí být potvrzeno RTI callbackem `timeRegulation-Enabled()`.

```
lookahead = new DoubleValuedLogicalTimeInterval(0.0); // zero lookahead
rtiAmbassador.enableTimeRegulation(lookahead);
```

Metoda `publishAndSubscribe()` provede publikaci a přihlášení k odběru interakčních tříd, které byly definovány v objektovém modelu J-Sim federace. K tomu využívá služeb RTI `publishInteractionClass()` a `subscribeInteractionClass()`.

Člen federace, který úspěšně provedl všechny předchozí kroky, je připraven k simulaci. Uživateli simulace tedy zdánlivě nic nebrání ve volání metody `step()`. Distribuovaná simulace je však technikou *spolupráce* více simulací, proto může být zahájena teprve po dosažení okamžiku, ve kterém jsou k provedení simulace připraveni všichni členové federace. To znamená, že do federace musí být připojen zadaný počet členů federace a tito členové musí dát nějakým způsobem najevo, že jsou připraveni k zahájení simulace. K tomuto účelu slouží mechanismus úvodní synchronizace federace.

Člen federace, který se stal správcem federace, proto nejprve čeká na připojení požadovaného počtu členů federace. Ke zjištění aktuálního počtu členů sleduje aktualizace objektů třídy `HLAfederate`. Aktualizace těchto atributů provádí RTI voláním callbackové metody `reflectAttributeValues()`. Správce federace při každé aktualizaci objektu třídy `HLAfederate` vloží název instance do množiny (použití množiny zabrání duplicitnímu započítání stejného člena federace

při opakované aktualizaci, kterou nelze vyloučit). Při každé aktualizaci se kontroluje, zda počet prvků množiny dosáhl hodnoty `numberOfFederates`. Po splnění této podmínky se ve federaci nachází požadovaný počet členů federace, ale stále není možné předpokládat, že všichni členové federace již provedli všechny kroky potřebné k zahájení simulace⁶.

Správce federace následně požádá RTI o zaregistrování synchronizačního bodu se symbolickým označením (*synchronization point label*) začátku simulace. Je třeba poznamenat, že synchronizační bod je sice možné zaregistrovat před připojením požadovaného počtu členů federace, ale problém úvodní synchronizace tímto způsobem nelze vyřešit. V souladu s HLA totiž RTI oznamuje synchronizační body pouze členům federace, kteří se již připojili k federaci, což znamená, že později připojený člen by takové oznámení mohl zmeškat.

5.4.10.2 Ukončení vykonávání federace

Úkolem metody `endFederationExecution()` je korektní ukončení simulace, odhlášení člena federace a ukončení práce s RTI. Pokud aktuální simulace ještě nebyla ukončena, metoda `sendEndOfSimulation()` pošle všem členům federace zprávu o ukončení simulace.

Volání `rtiAmbassador.resignFederationExecution()` provede odhlášení z federace. Pokud má být federace zrušena (což je doporučeno), volá se také služba `destroyFederationExecution()`. Podle HLA však není možné zrušit federaci, ke které jsou ještě připojeni členové. Skutečnou likvidaci federace provede člen, který se z federace odhlásí jako poslední.

5.4.10.3 Odeslání požadavku

Pokud člen federace potřebuje provést akci nad vzdáleným objektem, musí tento požadavek doručit členovi federace, ve kterém se tento objekt nachází jako lokální. Způsob odeslání požadavku bude vysvětlen na příkladu, který provede naplánování vzdáleného procesu.

```
JSimHLARemoteProcess process3
    = new JSimHLARemoteProcess("Process 3", simulation);

...
process3.activate(2.5);
```

Metoda `activate()` vzdáleného procesu provede následující kód:

⁶Takový předpoklad by se vlastně opíral o předpoklad rychlosti paralelně spuštěných programů, což je nepřijatelné.

```

public void activate(double when)
    throws JSimSecurityException, JSimInvalidParametersException
{
    ...
    response = myParent.getJSimHLAConnection().
        sendRequestProcessActivate(myName, when);
    responseHeader = response.getHeader();
    ...
}

```

Metoda `sendRequestProcessActivate()` převezme požadavek a provede jeho „zabalení“ do instance třídy `JSimHLARequestHeader`. Požadavek naplánování procesu má jeden parametr typu `double`, který je třeba převést na instanci třídy `Double`, aby jej bylo možné předat konstruktoru třídy `JSimHLARequest` (viz 5.4.4), který očekává data typu `Object`. Vytvořený požadavek je předán metodě `sendRequest()`.

```

protected JSimHLAResponse sendRequestProcessActivate(
    String processName, double when)
{
    JSimHLARequestHeader header = new JSimHLARequestHeader(
        federateHandle, processName, JSIM_PROCESS_ACTIVATE);
    Double data = new Double(when);

    JSimHLAResponse response = sendRequest(
        new JSimHLARequest(header, data));
    return response;
}

```

Metoda `sendRequest()` provede serializaci parametrů požadavku a vloží parametry (hlavičku i data) do mapy typu `ParameterHandleValueMap`. Tato mapa je předána metodě `sendInteraction()`, která provede odeslání interakce:

```

rtiAmbassador.sendInteraction(theInteraction, suppliedParameters,
    new byte[0], logicalTime);

```

Po odeslání se volá metoda `waitForResponse()`, která čeká na příjem odpovědi (viz 5.4.10.6) k odeslanému požadavku.

5.4.10.4 Příjem požadavku

Všechny interakce (tedy požadavky i odpovědi) jsou doručovány ve formě callbacku `receiveInteraction()` instanci třídy `JSimHLAFederateAmbassador`. Data doručené interakce jsou vzápětí předána metodě `receiveInteraction()` instance třídy `JSimHLAConnection`, která je zařadí do fronty pro další zpracování. Fronta je typu `JSimHLAReceivedInteractionCallback` (viz 5.4.6).

Zpracování doručených interakcí provádí metoda `processReceivedInteractions()`. Aby zpracování interakcí proběhlo co nejdříve po jejich doručení (v opačném případě by některý člen federace musel dlouho čekat na odpověď), měla by uvedená metoda být volána v okamžiku, který následuje po možném doručení interakcí. Protože TSO zprávy jsou členovi federace doručovány pouze ve stavu *time advancing* (tedy před udělením času od RTI), nastává tento okamžik po každém přidělení času od RTI. Proto J-Sim volá metodu `processReceivedInteractions()` vždy po volání `nextMessageRequestAvailable()`.

Metoda `processReceivedInteractions()` postupně vybírá interakce z fronty a provádí jednoduchou klasifikaci. Pokud interakce obsahuje požadavek, je volána metoda `receiveRequest()`. Pokud se jedná o odpověď, je volána metoda `receiveResponse()` a pro zprávu o ukončení simulace metoda `receiveEndOfSimulation()`.

```
if (interactionClass.equals(ichJSimRequest))
{
    receiveRequest(theParameters);
    continue;
}
```

Metoda `receiveRequest()` zjistí, jaké třídy se uvedený požadavek týká a podle toho jej předá k dalšímu zpracování. Například pro naplňování procesu (viz příklad 5.4.10.3) bude volána metoda `receiveRequestProcessActivate()`. Hlavička každého požadavku obsahuje jméno objektu, kterého se daný požadavek týká. Pro vykonání požadavku je třeba získat z tohoto jména odkaz na lokální objekt, nad kterým bude služba provedena. K tomuto účelu poskytuje třída `JSimHLASimulation` tyto metody:

```
protected JSimProcess getLocalProcessByName(String name)
protected JSimHead getLocalHeadByName(String name)
```

První metoda prochází seznam lokálních procesů a v případě nalezení procesu s požadovaným jménem vrací odkaz na tento proces. Druhá metoda pracuje stejným způsobem nad seznamem lokálních front. Pokud objekt v daném členovi federace neexistuje, bude vrácena hodnota `null` (neznačená to chybu, protože objekt může existovat v jiném členovi federace). Pokud je vrácen platný odkaz na lokální objekt⁷, lze nad tímto objektem zavolat požadovanou službu J-Sim. Pro případ naplňování procesu se tedy vykoná:

```
process.activate(when);
```

Pro dokončení zpracování požadavku zbývá odeslání odpovědi. To bude vysvětleno v následujícím odstavci.

⁷Pro správnou funkci simulace by každý objekt měl existovat jako lokální právě v jednom členovi federace. Za splnění této podmínky je plně zodpovědný autor simulace.

5.4.10.5 Odeslání odpovědi

Pokud byla na základě požadavku jiného člena federace provedena služba J-Sim, která vrací nějaká data, musí být tato data vrácena jako součást odesílané odpovědi. Aby bylo tato data možné jednoduše přenést v parametrech interakce, použije se serializace objektů.

Pokud provedená služba J-Sim proběhne v pořádku, bude jako výsledek služby (atribut `errorCode` hlavičky odpovědi) vrácena hodnota `JSIM_RESULT_OK`. Téměř každá služba J-Sim však může skončit výjimkou. Protože impulsem k vyhození takové výjimky byl původní požadavek nad vzdáleným objektem, nemá smysl vyhození výjimky v program člena federace, který požadavek skutečně vykonal. Výjimku by tedy bylo vhodné nejprve zachytit a následně informaci o výskytu výjimky vhodným způsobem odeslat prvnímu členovi federace, který o chybě informuje uživatele simulace.

Jedním ze způsobů odeslání informací o vyhozené výjimce je zavedení kódů pro všechny výjimky, které mohou být jednotlivými službami J-Sim vyhozeny, nastavení výsledku odpovědi na kód vyhozené výjimky a připojení dodatečných informací o výjimce k datům odpovědi. Příjemce odpovědi by následně vyhodil takovou výjimku, jejíž kód by přečetl z hlavičky odpovědi. Nakonec bylo realizováno řešení, které využívá toho, že v jazyce Java je možné serializovat nejen instance „běžné“ třídy, ale také výjimky. Serializace výjimek nevyžaduje provedení žádných změn v kódu výjimek J-Sim, protože rozhraní `Serializable` je již implementováno třídou `java.lang.Throwable`, která je společným předkem všech výjimek jazyka Java. Výjimky J-Sim jsou tedy zachyceny příkazem `catch` a odeslány jako součást dat odpovědi. Vyhození výjimky provede člen federace, který službu nad vzdáleným objektem vyvolal.

5.4.10.6 Příjem odpovědi

Po odeslání požadavku se uvnitř metody `sendRequest` provede volání:

```
response = waitForResponse(request);
```

Úkolem metody `waitForResponse` je čekat tak dlouho, dokud není doručena odpověď na požadavek specifikovaný parametrem `request`. Pro doručení odpovědi je nutné opakovaně žádat RTI o udělení času s následným zpracováním doručených interakcí:

```
nextMessageRequestAvailable(logicalTime);  
processReceivedInteractions();
```

Uvedený úsek kódu je uzavřen v sekci `synchronized (requestLock)`. Tím je zaručeno, že člen federace nemůže poslat další požadavek bez doručení odpovědi na svůj předchozí požadavek. Do synchronizované sekce má přístup pouze

jediné vlákno⁸, které je ovšem „zaměstnáno“ čekáním na odpověď. Aby při volání metody `nextMessageRequestAvailable()` nedocházelo k nepřetržité komunikaci mezi členem federace a RTI, je v cyklu opakovaně prováděno volání metody `wait()` nad objektem `requestLock`. Pokud je během čekání vlákna doručena odpověď, bude čekající vlákno „probuzeno“ voláním `requestLock.notify()`.

5.4.10.7 Žádost o zvýšení času

Pokud člen J-Sim federace potřebuje požádat o přidělení času, volá metodu `nextMessageRequestAvailable()` instance třídy `JSimHLAConnection`. Tato metoda zavolá stejnojmennou metodu nad objektem `rtiAmbassador`. Tím se člen federace dostane do stavu *time advancing*, ve kterém jsou mu doručovány TSO a RO zprávy od RTI. V tomto stavu člen federace setrvává až do doručení callbacku `timeAdvanceGrant()`, který mu oznámí přidělení nového simulačního času. Při čekání na přidělení času je využívána instance třídy `JSimHLABarrier` (viz 5.4.8).

5.4.11 Metoda `step()` pro distribuovanou simulaci

Pro správnou funkci distribuované simulace je nutné částečně upravit chování metody `step()`, deklarované ve třídě `JSimSimulation`. Tyto úpravy musí být pochopitelně provedeny tak, aby byla zachována správná funkce nedistribuované simulace. Metodu `step()` nelze překrýt ve třídě potomka `JSimHLASimulation`, protože je deklarována jako `final`. Případné odstranění klíčového slova `final` by však problém nevyřešilo příliš elegantním způsobem, protože podstatná část (poměrně dlouhého) kódu metody `step()` by musela být ve třídě potomka duplikována.

Z toho důvodu byl do třídy `JSimSimulation` přidán atribut `simulationHLA` typu `boolean`, jehož hodnota ovlivňuje chování metody `step()`. Hodnota tohoto atributu při použití nedistribuované simulace je vždy `false`, nastavení na hodnotu `true` se provádí v konstruktoru třídy `JSimHLASimulation` (viz 5.4.1).

Pokud má atribut `simulationHLA` hodnotu `true`, znamená to pro metodu `step()` určitou změnu chování při spouštění procesů. Tato změna se projeví v okamžiku, kdy kalendář není prázdný, což při nedistribuované simulaci vede k výběru první události z kalendáře a následnému spuštění procesu, ke kterému patří tato událost. Přitom nastává posunutí simulačního času. Při distribuované diskrétní simulaci podle HLA však není možné posunout čas libovolným způsobem, ale je třeba o přidělení požadovaného času nejprve požádat RTI voláním služby

⁸V J-Sim může být aktivní nejvýše jeden proces (nebo hlavní vlákno), což znamená, že souběžný přístup z více vláken by teoreticky vůbec neměl nastat. Přesto není možné uživateli simulace zcela zabránit ve vytvoření konstrukce, která by byla schopna souběžně odeslat více požadavků, což by zřejmě vedlo ke zhroucení simulace.

`nextMessageRequestAvailable()`. Argumentem této služby bude čas prvního procesu v kalendáři (v následujícím kódu uložen v proměnné `firstProcessTime`). Aby bylo možné předat tento čas jako argument službě HLA, je třeba jej převést na instanci typu `DoubleValuedLogicalTime`⁹. Protože se vlastně jedná o lokální čas člena federace, je tato proměnná pojmenována `localTime`, zatímco čas přidělený od RTI bude uložen do proměnné `grantedTime`. Přitom je nutné si uvědomit dvě významné skutečnosti. Za prvé, pokud jiný člen federace vygeneruje událost s menším časovým razítkem, než je požadovaný čas, bude doručena tato událost a přidělený čas bude menší než požadovaný čas. To znamená, že člen federace v tomto okamžiku nesmí spustit první proces v kalendáři. Za druhé, doručenou událostí může být požadavek na naplánování procesu v čase, který je menší než čas dosud prvního procesu v kalendáři. V takovém případě se tedy změni čas první události v kalendáři a proces, jehož naplánování patří k této události (každý proces má nejvýše jednu událost v kalendáři). Tento proces může být následně spuštěn, stejně jako by byl naplánován běžným způsobem (lokálně z tohoto člena federace). Princip úprav metody `step()` ilustruje následující úsek kódu:

```
// HLA support code begin
if (simulationHLA)
{
    double firstProcessTime;

    // grantedTime is the time granted from RTI
    DoubleValuedLogicalTime grantedTime;

    // localTime is the time of the first event of the calendar
    DoubleValuedLogicalTime localTime;

    grantedTime = (DoubleValuedLogicalTime)
        jSimHLAConnection.getLogLogicalTime();

    firstProcessTime = calendar.getFirstProcessTime();
    localTime = new DoubleValuedLogicalTime(firstProcessTime);

    while (grantedTime.compareTo(localTime) < 0)
    {
        // We have to ask RTI to advance our logical time.
        jSimHLAConnection.nextMessageRequestAvailable(localTime);

        // We have to process all events received from other federate(s).
        jSimHLAConnection.processReceivedInteractions();

        grantedTime = (DoubleValuedLogicalTime)
```

⁹Tento typ implementuje rozhraní `hla.rti.LogicalTime`.

```

        jSimHLAConnection.getLogLogicalTime();

        firstProcessTime = calendar.getFirstProcessTime();
        localTime = new DoubleValuedLogicalTime(firstProcessTime);
    } // while

    // Time of the first process in calendar might have changed!!!
    //
} // if (simulationHLA)
// HLA support code end

time = calendar.getFirstProcessTime();
process = calendar.getFirstProcess();
...

```

5.4.12 Úpravy tříd JSimProcess a JSimHead

Aby bylo možné při návrhu distribuované simulace snadno změnit lokální objekty na vzdálené (a naopak), byla navržena rozhraní JSimHLAProcess a JSimHLAHead. Uživatel při návrhu simulace může potřebné procesy i fronty deklarovat jako proměnné typu rozhraní a následně je používat bez ohledu na to, zda se jedná o lokální či vzdálené objekty. Převedení lokálního objektu na vzdálený lze potom provést změnou kódu pouze v místě volání konstruktoru objektu. Volání konstruktoru třídy JSimProcess (nebo potomka této třídy) stačí nahradit voláním konstruktoru třídy JSimHLARemoteProcess (proces pochopitelně musí existovat ve vzdáleném členovi federace). Postup pro frontu je analogický.

Rozhraní JSimHLAProcess obsahuje metody, které lze stejným způsobem použít pro lokální i vzdálené procesy. Tyto metody byly uvedeny v seznamu metod třídy JSimHLARemoteProcess (viz 5.4.2). Třída JSimHLARemoteProcess již rozhraní JSimHLAProcess implementuje. Zbývá tedy zajistit, aby rozhraní JSimHLAProcess bylo implementováno ještě třídou JSimProcess. Analogický postup platí i pro třídu JSimHead a rozhraní JSimHLAHead. Úpravy kódu stávajících tříd J-Sim jsou tedy následující:

```

JSimProcess implements JSimDisplayable, Comparable, JSimHLAProcess
...

```

```

JSimHead implements JSimDisplayable, Comparable, JSimHLAHead
...

```

5.4.13 Úpravy třídy JSimLink

V kapitole 5.4.4 bylo uvedeno, že třída `JSimLink` nepodporuje serializaci, což znamená, že funkce J-Sim, které instanci této třídy používají jako argument (metoda `into()`) nebo jako návratovou hodnotu (metody `first()` a `last()`), by nebylo možné pro distribuovanou simulaci použít. Aby vůbec bylo možné instanci této třídy serializovat, musí být třída doplněna o následující kód:

```
JSimLink implements Serializable
```

Aby bylo možné prvek zařadit do vzdálené fronty, je nutné ještě přetížít metodu `into()`. To lze provést následujícím způsobem:

```
public final void into(JSimHLARemoteHead queue)
    throws JSimSecurityException
{
    if (myQueue != null)
        throw new JSimSecurityException(
            "JSimLink.into(): Already in a queue.");

    queue.linkIntoRemoteHead(this);
} // into
```

Je zřejmé, že přetížená metoda `into()` pouze volá metodu třídy `JSimHLARemoteHead` (viz 5.4.3), která se již postará o doručení objektu do fronty umístěné v jiném členovi federace.

Tím jsou provedeny potřebné úpravy třídy `JSimLink` a prvkům front již zdánlivě nic nebrání v použití pro distribuovanou simulaci. Problém však může nastat v případě, že uživatel simulace při vytvoření instance třídy `JSimLink` specifikuje tzv. uživatelská data. K tomuto účelu slouží přetížený konstruktér s parametrem typu `Object`. Pokud třída uživatelských dat (nebo alespoň jedna datová složka této třídy) neimplementuje rozhraní `Serializable`, bude za běhu distribuované simulace (při prvním pokusu o vložení prvku do vzdálené fronty) vyhozena výjimka, protože takový objekt se nepodaří serializovat. Prakticky to znamená, že uživatelská data, která budou používána se vzdálenými frontami, musí umožňovat serializaci.

Kapitola 6

Úpravy software XRTI

6.1 Struktura XRTI

Před popisem provedených úprav software XRTI budou uvedeny informace o nejdůležitějších třídách tohoto produktu. Tyto informace jsou nezbytné pro pochopení úprav, popsanych v následujícím textu. Všechna vysvětlení jsou poměrně stručná a v žádném případě nemohou sloužit jako náhrada dokumentace XRTI ([Kap03]). Dále je třeba poznamenat, že nezbytným předpokladem pro pochopení funkce XRTI je nejen prostudování dokumentace, ale zejména podrobné seznámení se zdrojovými kódy ([XRTI]).

6.1.1 Třída XRTIAmbassador

Třída `XRTIAmbassador` je nejrozsáhlejší¹ a nejdůležitější třídou XRTI. Třída implementuje rozhraní `hla.rti.RTIAmbassador`, které je součástí specifikace rozhraní HLA pro jazyk Java. Aby se člen federace mohl zúčastnit federace, musí vytvořit a inicializovat instanci této třídy. Následně je tato instance používána pro volání služeb RTI, tedy pro komunikaci směrem od člena federace k RTI.

Kromě vytvoření instance slouží třída `XRTIAmbassador` také jako rodičovská třída pro dvě důležité třídy: `ExecutiveClientAmbassador` (viz 6.1.2) a `FederationExecutionAmbassador` (viz 6.1.3).

Významnou vlastností třídy `XRTIAmbassador` je schopnost komunikace s aplikací `XRTIExecutive`. Komunikace probíhá oběma směry mezi instancí třídy `XRTIAmbassador` a instancí třídy `ExecutiveClientAmbassador`, která reprezentuje vzdálený protějšek každého člena federace. Klíčový význam pro tuto komunikaci mají **interakce**. Interakční třídy jsou v XRTI používány nejen pro posílání a příjem interakcí ve smyslu interakčních tříd HLA, ale také

¹Třída `XRTIAmbassador` má přibližně 9000 řádek zdrojového kódu.

pro veškeré služby HLA, jejichž vykonání požadují jednotliví členové federace, včetně všech callbacků, poslaných od RTI směrem k členovi federace.

Použití interakcí pro veškerou komunikaci má pochopitelně důsledky pro odesílatele i příjemce. Pro odesílatele to znamená, že každé volání služby HLA, přestože zdánlivě vůbec nemusí souviset s interakcemi, musí být nejprve převedeno na interakci určité třídy a následně odesláno voláním `sendInteraction()` jedním z komunikačních kanálů. Tyto kanály existují pro každý směr dva: jeden je určen pro spolehlivý přenos zpráv (*reliable*), druhý pro přenos zpráv s „nejlepším úsilím“ (*best effort*).

Na straně příjemce jsou z obou komunikačních kanálů postupně vybírány doručené zprávy. Čtení z kanálů provádí samostatná vlákna. Pokud je nějaká zpráva k dispozici, je předána metodě `interpretReceivedMessage()`. Všechny zprávy jsou zabaleny do „obálky“, která má pevně daný formát. Formát obálky je definován třídou `HLAbootstrapInteractionPayload` (další informace budou uvedeny v odstavci 6.5.1). Metoda `interpretReceivedMessage()` následně provede volání metody `receiveInteraction()` nad objektem `proxyAmbassador`. Tento objekt předá interakci k dalšímu zpracování tak, aby bylo zajištěno převedení interakce na původní zprávu.

Všechny zprávy jsou předávány k dalšímu zpracování prakticky okamžitě po jejich doručení. Ke zpracování zpráv navíc dochází v pořadí, v jakém byly doručeny. Takové pořadí může pochopitelně být odlišné od pořadí jejich odeslání (což není podstatné) a také od pořadí časových razítek (*time stamp*), pokud byla tato razítka uvedena (což je podstatné). Uvedené chování RTI je správné pouze v případě, že členové federace nejsou ani časově regulující (*time regulating*) ani časově omezení (*time constrained*) a všechny předávané zprávy jsou typu *receive order* (RO). Pro použití zpráv typu *time stamp order* (TSO) je nutné mechanismus doručování zpráv XRTI přepracovat. Tyto úpravy budou vysvětleny v kapitolách 6.3 a 6.4.

6.1.2 Třída `ExecutiveClientAmbassador`

Instance třídy `ExecutiveClientAmbassador` představuje vzdálený protějšek instance třídy `XRTIAmbassador`. Na každou instanci `XRTIAmbassador`, která již byla inicializována a navázala komunikaci s RTI, připadá právě jedna instance třídy `ExecutiveClientAmbassador`. Instance třídy `ExecutiveClientAmbassador` existuje vždy na straně serveru (tedy aplikace `XRTIExecutive`), nikoliv na straně klienta (člena federace). Instance třídy `ExecutiveClientAmbassador` vzniká ve třídě `XRTIExecutive` po navázání síťové komunikace člena federace s RTI.

6.1.3 Třída `FederationExecutionAmbassador`

Třída `FederationExecutionAmbassador` existuje (podobně jako třída `ExecutiveClientAmbassador`) vždy na straně serveru. Jedna instance této třídy odpovídá vždy jedné federaci. Instance třídy `FederationExecutionAmbassador` vzniká tehdy, když třída `ExecutiveClientAmbassador` přijme zprávu, které předcházelo volání metody `createFederationExecution()` některým z členů federace.

Třída `FederationExecutionAmbassador` udržuje seznam instancí třídy `ExecutiveClientAmbassador`, které odpovídají jednotlivým členům této federace. Položky do tohoto seznamu jsou přidávány metodou `registerExecutiveClientAmbassador()` po doručení zprávy, které předcházelo volání `joinFederationExecution()`. Po odhlášení člena federace voláním `resignFederationExecution()` (a doručení příslušné zprávy) je volána metoda `deregisterExecutiveClientAmbassador()`, která člena federace ze seznamu odstraní. Pokud některý z členů této federace vygeneruje zprávu pro ostatní členy federace (poslání interakce, aktualizace hodnoty atributů, vytvoření nebo odstranění instance objektové třídy), je tato zpráva doručena odpovídajícímu objektu `ExecutiveClientAmbassador`, který zprávu předá objektu `FederationExecutionAmbassador`. Tento objekt následně provede rozeslání zprávy všem členům federace (s výjimkou odesílatele).

6.1.4 Třída `XRTIExecutive`

Třída `XRTIExecutive` obsahuje metodu `main()`, což znamená, že ji lze spustit jako samostatnou aplikaci. Tato aplikace slouží jako centrální server pro HLA distribuovanou simulaci. Instance třídy `XRTIExecutive` po spuštění vytvoří vlákno `channelAcceptorThread`, které obsluhuje požadavky na připojení nových členů federace. Při každém připojení člena federace je vytvořena nová instance třídy `ExecutiveClientAmbassador`.

6.1.5 Třída `ProxyAmbassador`

Třída `ProxyAmbassador` implementuje rozhraní `hla.rti.FederateAmbassador`. Instance třídy `ProxyAmbassador`, která vznikne při inicializaci objektu třídy `XRTIAmbassador` slouží jako „kořenový“ příjemce všech callback zpráv od RTI. Tato instance udržuje vlastní seznam registrovaných instancí typu `FederateAmbassador`. Jednou z registrovaných instancí se stane také odkaz na instanci typu `FederateAmbassador`, který je předán při volání metody `joinFederationExecution()`. Všechny callback zprávy od RTI jsou následně rozesílány registrovaným instancím `FederateAmbassador`.

Mezi registrovanými instancemi typu `FederateAmbassador` je zaregistrována řada tříd², které jsou potomkem třídy `ProxyAmbassador`. Název těchto tříd končí slovy `ProxyAmbassador`, například `ManagementProxyAmbassador` nebo `MetaFederationProxyAmbassador`. Tyto třídy nejsou napsány přímo programátorem XRTI, ale jsou automaticky generovány nástrojem `ProxyCompiler` při překladu XRTI. Například třída `MetaFederationProxyAmbassador` je generována ze souboru `MetaFederationObjectModel.xml`³.

Cílem automaticky generovaných tříd je snaha o určité zjednodušení procesu odesílání a příjmu všech zpráv při komunikaci mezi členem federace a RTI. Standard HLA totiž obsahuje velké množství služeb s mnoha různými parametry, ale nedefinuje žádný protokol pro výměnu zpráv mezi členem federace a RTI. Jak již bylo uvedeno v odstavci 6.1.1, XRTI pro veškerou komunikaci používá interakce. Standard HLA je však příliš robustní a komplikovaný, což se projevuje mimo jiné tím, že odeslání každé interakce (což je poměrně elementární operace) vyžaduje provedení posloupnosti několika operací. Mezi tyto operace patří zejména nutnost získání všech potřebných celočíselných identifikátorů (*handle*) pro konkrétní interakční třídu a pro jednotlivé parametry interakce. Dále je nutné provést převedení všech parametrů požadované služby HLA na instanci třídy `ParameterHandleValueMap`. Automaticky vygenerované třídy, které jsou potomkem třídy `ProxyAmbassador`, jsou schopny provedení těchto operací úspěšně zajistit, což znamená, že se programátor o tyto „nízkoúrovňové“ záležitosti prakticky nemusí starat.

Příkladem takové komunikace může být situace, která nastane poté, co účastník simulace požaduje vytvoření nové federace a zavolá tedy metodu `createFederationExecution()` nad objektem třídy `XRTIAmbassador`. Odeslání požadavku k RTI zajistí následující kód:

```
public void createFederationExecution(String federationExecutionName,
                                     URL fdd)
    throws FederationExecutionAlreadyExists,
           CouldNotOpenFDD, ErrorReadingFDD,
           RTIInternalError
{
    ...
    metaFederationProxyAmbassador.sendHLACreateFederationExecution(
        federationExecutionName,
        encodedFDD, new byte[0]
    );
    ...
}
```

Na straně příjemce metoda `receiveInteraction()` porovná *handle* doru-

²Většinou se jedná o třídy z balíku `org.npsnet.xrti.proxies`.

³Tyto soubory lze nalézt v adresáři `resources/org/npsnet/xrti` distribuce XRTI.

čené interakce se všemi známými identifikátory a podle toho zavolá příslušnou metodu. Pro příklad s vytvořením federace je příjemcem takové události třída `ExecutiveClientAmbassador`, která implementuje automaticky generované rozhraní `MetaFederationInteractionListener` a musí implementovat metodu `receiveHLACreateFederationExecution()`.

6.2 Implementace času pro typ `double`

Standard HLA neurčuje konkrétní datový typ, který má být použit pro reprezentaci simulačního času. Specifikace rozhraní HLA pro jazyk Java obsahuje rozhraní `hla.rti.LogicalTime`. Toto rozhraní musí být implementováno datovým typem, který používají členové federace (a také RTI) pro konkrétní simulaci. Všechny metody rozhraní `RTIAmbassador` a `FederateAmbassador`, které pracují s logickým časem, mají ve svých deklaracích uveden právě typ `LogicalTime`.

Rozhraní `LogicalTime` obsahuje řadu metod, z nichž nejdůležitější jsou uvedeny v následujícím seznamu:

- `add()` – přičte zadaný interval k danému logickému času.
- `compareTo()` – porovná logický čas s jiným časem. Velmi důležité zejména pro všechny služby, které žádají o přidělení logického času.
- `equals()` – test na rovnost s jiným logickým časem.
- `encode()` – převede hodnotu logického času na pole typu `byte[]`. Toto pole lze použít při komunikaci mezi RTI a členem federace (např. jako parametr interakce).

Pro vytvoření instance logického času se obvykle nevyužívá konstruktor, ale některá z metod instance třídy, která implementuje rozhraní `LogicalTimeFactory`. Pro vytvoření výchozího logického času lze použít metodu `makeInitial()` volanou nad instancí třídy příslušné *factory*. Při implementaci nového typu logického času je tedy třeba implementovat také toto rozhraní.

S rozhraním `LogicalTime` také úzce souvisí rozhraní `LogicalTimeInterval`, které slouží zejména jako datový typ pro hodnotu *lookaheadu*. Z důležitých metod tohoto rozhraní je třeba zmínit alespoň metodu `isZero()`, která vrací `true`, pokud má daný interval logického času nulovou délku. Tato metoda má význam pro členy federace, kteří používají nulovou hodnotu *lookaheadu*⁴. Další metody mají podobný význam jako metody rozhraní `LogicalTime`. K rozhraní `LogicalTimeInterval` existuje také příslušné rozhraní `LogicalTimeIntervalFactory`.

⁴Nulový *lookahead* významně ovlivňuje chování některých služeb skupiny *time management*.

Obecně tedy lze říci, že při návrhu nového typu logického času je nutné vytvořit vždy čtyři nové třídy, které implementují příslušná rozhraní balíku `hla.rti`.

Simulační knihovna J-Sim používá pro simulační čas typ `double`. Implementace XRTI poskytuje programátorovi simulace typy `LongValuedLogicalTime` a `LongValuedLogicalTimeInterval`, které (jak název napovídá) pracují s typem `long`. Pro použití se simulační knihovnou J-Sim je tedy nutné napsat nové třídy, které budou používat typ `double`. Všechny třídy XRTI pro práci s časem jsou součástí balíku `org.npsnet.xrti.utilities`.

6.2.1 Třídy `DoubleValuedLogicalTime`, `DoubleValuedLogicalTimeFactory`

Třída `DoubleValuedLogicalTime` je velmi podobná třídě `LongValuedLogicalTime`. Pro naprogramování nové třídy zdánlivě stačí zkopírovat kód původní třídy a na potřebných místech nahradit typ `long` typem `double`. To je samozřejmě správná úvaha, ale ještě je třeba si uvědomit, že typ `double` reprezentuje desetinná čísla, která mohou být zdrojem problémů při vzájemném porovnávání čísel, jejichž hodnoty se od sebe liší pouze nepatrně. Prakticky to znamená, že metody `compareTo()` a `equals()` při testu na rovnost neporovnávají hodnoty typu `double`, ale hodnoty typu `long`, které lze získat voláním `doubleToLongBits()` (statická metoda třídy `Double`). Další informace k této problematice lze nalézt např. v [Her03].

6.2.2 Třídy `DoubleValuedLogicalTimeInterval`, `DoubleValuedLogicalTimeIntervalFactory`

Třída `DoubleValuedLogicalTimeInterval` reprezentuje logický interval pro *lookahead*. Pro implementaci této třídy včetně příslušné *factory* platí stejná pravidla jako pro třídu `DoubleValuedLogicalTime`.

6.2.3 Úpravy třídy `XRTIExecutive`

Při připojení člena federace do federace musí být metodě `joinFederationExecution()` kromě názvu požadované federace a odkazu na instanci typu `FederateAmbassador` předána instance třídy `hla.rti.MobileFederateServices`. Tato třída slouží k přepravě instancí typů `LogicalTimeFactory` a `LogicalTimeIntervalFactory`. Při připojení do federace musí být uvedena proto, aby bylo zřejmé, jakou reprezentaci logického času daný člen federace používá. S logickým časem však nepracuje pouze člen federace, ale také RTI. V případě

XRTI se jedná o aplikaci `XRTIExecutive`, což znamená, že je nutné provést určité úpravy stejnojmenné třídy.

Jak již bylo dříve uvedeno, při připojení nového člena federace vzniká instance třídy `ExecutiveClientAmbassador`. Konstruktor této třídy má dva parametry. Prvním je odkaz na instanci `XRTIExecutive`, ve druhém parametru typu `MessageChannel` se předává odkaz na komunikační kanál nově připojeného člena federace. Aby mohla instance třídy `ExecutiveClientAmbassador` správně interpretovat hodnoty logického času, které jí posílá příslušný člen federace, musí být rovněž informována o implementaci logického času, která se v rámci dané federace používá. Proto byl při zavedení podpory time managementu pro XRTI do konstruktoru třídy `ExecutiveClientAmbassador` doplněn třetí parametr typu `hla.rti.MobileFederateServices`, který slouží k předání této informace.

Aplikace `XRTIExecutive` by samozřejmě neměla být omezena pouze na použití s jediným typem logického času, který se používá právě pro knihovnu J-Sim. Konkrétní implementaci logického času proto lze určit jako konfigurační parametr pro `XRTIExecutive`. Jako výchozí hodnota je použita právě `DoubleValuedLogicalTimeFactory`:

```
Class logicalTimeFactoryClass = Class.forName(  
    configuration.getProperty(  
        "LogicalTimeFactory",  
        "org.npsnet.xrti.utilities.DoubleValuedLogicalTimeFactory"  
    )  
);  
logicalTimeFactory = (LogicalTimeFactory)  
    logicalTimeFactoryClass.newInstance();
```

6.3 Implementace služeb skupiny time management

Současná verze XRTI nepodporuje žádné služby skupiny *time management*. Třída `XRTIAmbassador` sice obsahuje všechny metody rozhraní `hla.rti.RTIAmbassador`, metody ze skupiny služeb *time management* však neprovádějí téměř žádný (v některých případech vůbec žádný) kód. Při zavedení těchto služeb bylo nutné upravit stávající třídy XRTI a také navrhnout a naprogramovat některé nové třídy (viz 6.4). Úpravy stávajících tříd se týkají zejména tříd `XRTIAmbassador`, `ExecutiveClientAmbassador` a `FederationExecutionAmbassador`. Vzhledem k tomu, že provedení těchto úprav není triviální, bude nyní uvedeno podrobnější vysvětlení.

6.3.1 Služba `enableTimeRegulation()`

Pokud člen federace zavolá metodu `enableTimeRegulation()`, bude v odpovídající instanci třídy `ExecutiveClientAmbassador`, zavolána metoda `receiveHLA-enableTimeRegulation()`. Před naprogramováním této metody je třeba vyřešit dva problémy. Prvním z nich je výpočet nového času pro člena federace. Tento čas musí být vypočítán tak, aby bylo zaručeno, že regulující člen federace nebude schopen poslat TSO zprávu s časovým razítkem, které by bylo menší než aktuální logický čas některého časově omezeného člena federace. Jinými slovy lze říci, že nesmí dojít k situaci, kdy některý časově omezený člen federace dostane zprávu ze své minulosti. Z toho vyplývá, že algoritmus výpočtu tohoto času musí brát ohled na časově omezené členy federace. Pokud se ve federaci žádný časově omezený člen federace nenachází, znamená to, že jako čas zapnutí časové regulace lze přidělit aktuální čas člena federace.

Z pohledu časově omezeného člena federace musí existovat hodnota logického času, pro kterou je zaručeno, že člen federace již nemůže obdržet žádnou zprávu s časovým razítkem, které je menší než tato hodnota. Tato hodnota se nazývá **Greatest Available Logical Time**⁵ (GALT). Hodnota GALT existuje pro všechny členy federace, význam však má pouze pro členy federace, kteří jsou časově omezeni. Tito členové federace nemají dovoleno posunout svůj logický čas za tuto hodnotu (odtud pochází termín *time constrained*).

Výpočet hodnoty GALT provádí metoda `computeGalt()`. Pokud žádný z členů federace není časově regulující, hodnota GALT není definována (ani pro ostatní členy federace). Pokud se ve federaci nachází časově regulující členové, lze hodnotu GALT vypočítat jako minimum z efektivního logického času těchto členů federace. Efektivní logický čas je definován jako hodnota minimálního časového razítka, které může časově regulující člen federace odeslat. Pokud je takový člen federace ve stavu *time granted*, jeho efektivní logický čas je roven součtu hodnoty jeho aktuálního času a hodnoty jeho lookaheadu. Ve stavu *time advancing* se efektivní logický čas vypočítá jako součet hodnoty požadovaného času a hodnoty jeho lookaheadu. Pokud člen federace požádal o zapnutí časové regulace, znamená to, že bude schopen posílat TSO zprávy, a proto je nutné zabránit mu v možném odeslání časového razítka, které by bylo menší než minimum z hodnot GALT ostatních členů federace. To znamená, že nový logický čas tohoto člena federace bude roven minimální hodnotě z GALT ostatních členů federace nebo jeho aktuálnímu logickému času, pokud je tento čas větší nebo pokud GALT není definován.

Druhým problémem je odeslání callbacku `timeRegulationEnabled()`.

⁵Verze HLA, která byla přijata jako standard IEEE 1516, přinesla některé změny v terminologii. Hodnota *Greatest Available Logical Time* (GALT) byla ve starší verzi HLA 1.3 nazývána *Lower Bound on Time Stamp* (LBTS). Termín *Least Incoming Time Stamp* (LITS) má stejný význam jako starší termín *Minimum Next Event Time* (MNET).

Automaticky generované proxy třídy totiž posílání ani příjem interakce tohoto typu neumožňují. Z toho vyplývá, že je nutné upravit některý soubor, ze kterého jsou proxy třídy generovány. V tomto případě byl rozšířen soubor `BootstrapObjectModel.xml`. Do tohoto souboru byly doplněny následující definice interakčních tříd:

```
<interactionClass name="HLAtimeRegulationEnabled"
    sharing="PublishSubscribe"
    transportation="HLAreliable"
    order="Receive"
    semantics="Notifies the federate that time regulation
        has been enabled.">
    <parameter name="time"
        dataType="HLAlogicalTime"
        semantics="the current logical time"/>
</interactionClass>

<interactionClass name="HLAtimeConstrainedEnabled"
    sharing="PublishSubscribe"
    transportation="HLAreliable"
    order="Receive"
    semantics="Notifies the federate that time-constrained
        mode has been enabled.">
    <parameter name="time"
        dataType="HLAlogicalTime"
        semantics="the current logical time"/>
</interactionClass>
```

Po provedení těchto úprav a překladu XRTI je možné v metodě `receiveHLA-enableTimeRegulation()` použít tento kód:

```
bootstrapProxyAmbassador.sendHLAtimeRegulationEnabled(
    encodedTime, new byte[0]);
```

Příjemcem této interakce je instance třídy `XRTIAmbassador`. Tato třída musí obsahovat metodu `receiveHLAtimeRegulationEnabled()`, která provede následující kód:

```
requestForTimeRegulationPending = false;
timeRegulationEnabled = true;
try
{
    LogicalTime decodedTime = decodeLogicalTime(time);
    proxyAmbassador.timeRegulationEnabled(decodedTime);
    logicalTime = decodedTime;
}
```

```
catch (RTIException rtie)
{
    throw new FederateInternalError(rtie.toString());
}
...
```

Teprve po zavolání metody `timeRegulationEnabled()` nad objektem `proxy-Ambassador` je časová regulace člena federace zapnuta a člen federace může posílat TSO zprávy.

Implementace služby `disableTimeRegulation()` je poměrně jednoduchá, protože vypnutí časové regulace nevyžaduje odeslání potvrzujícího callbacku. Člen federace přestává být časově regulující ihned po zavolání této služby.

6.3.2 Služba `enableTimeConstrained()`

Zapnutí časového omezení se provádí voláním služby `enableTimeConstrained()` a musí být potvrzeno callbackem `timeConstrainedEnabled()`. Kromě výpočtu nového času člena federace byla implementace této služby i příslušného callbacku realizována podobným způsobem, jaký byl popsán v předchozím odstavci.

Člen federace přestává být časově omezeným ihned po zavolání metody `disableTimeConstrained()`. Fronty TSO a RO zpráv na straně RTI však mohou obsahovat zprávy, které ještě nebyly členovi federace doručeny. Z tohoto důvodu musí instance třídy `ExecutiveClientAmbassador` po přijetí zprávy `receiveHLAdisableTimeConstrained()` příslušné fronty vyprázdnit. Protože člen federace již není časově omezený, budou mu všechny zprávy z obou front doručeny jako RO.

6.3.3 Služba `nextMessageRequestAvailable()`

Pro distribuovanou diskrétní simulaci je třeba naprogramovat některou službu pro přidělení simulačního času. Pro použití s knihovnou J-Sim se jedná o službu `nextMessageRequestAvailable()`. Volání této služby se na straně třídy `ExecutiveClientAmbassador` projeví voláním `receiveHLAnextMessageRequestAvailable()`.

Při implementaci této služby je třeba si uvědomit, že volání služby pro přidělení času obvykle není impulsem pro okamžité odeslání callbacku `timeAdvanceGrant()` ke členovi federace, který toto přidělení času požadoval⁶. RTI může časově omezenému členovi federace přidělit logický čas teprve v okamžiku, kdy

⁶Uvedené tvrzení platí pro všechny služby pro přidělení času s výjimkou služby `flushQueueRequest()`, která slouží k vyprázdnění front všech RO i TSO zpráv. Tato služba je využívána při distribuované simulaci s optimistickou synchronizací, která není obsahem této práce.

hodnota požadovaného času bude menší (případně menší nebo rovna) hodnotě GALT tohoto člena federace. Jak již bylo uvedeno v odstavci 6.3.1, hlavním faktorem pro výpočet GALT jsou hodnoty efektivního logického času regulujících členů federace. Pro konzervativně synchronizovanou federaci, kde jsou všichni členové federace současně časově regulující i omezení, z předchozí věty vyplývá, že žádný člen federace nemůže posunout simulační čas bez spolupráce s ostatními členy federace. Člen federace proto zůstává ve stavu *time advancing* až do okamžiku, kdy jeho hodnota GALT bude zvýšena tak, aby byla splněna výše uvedená podmínka. Hodnota GALT se zvyšuje v okamžiku, kdy o čas požádá regulující člen federace. Takový požadavek může způsobit zaslání callbacku `timeAdvanceGrant()` pro *jiného* člena (nebo členy) federace.

Protože všechny požadavky na přidělení času mají řadu společných vlastností, po doručení instanci třídy `ExecutiveClientAmbassador` jsou předávány metodě `handleTimeRequest()`. Parametry této metody jsou požadovaný čas a identifikace požadované služby. Tato metoda nastaví příznak `inTimeAdvancingState` na hodnotu `true`, dekóduje požadovaný logický čas a v případě použití nulového lookaheadu nastaví potřebné příznaky omezení pro nulový lookahead⁷. Metoda `handleTimeRequest()` následně předá požadavek stejnojmenné metodě instance třídy `FederationExecutionAmbassador`, která odpovídá aktuální federaci:

```
protected synchronized void handleTimeRequest(  
    ExecutiveClientAmbassador eca)  
    throws FederateInternalError  
{  
    computeAllGaltValues(eca);  
    sendAllPossibleTimeAdvanceGrants(eca);  
} // handleTimeRequest
```

Z uvedeného kódu je zřejmé, že po každém požadavku na posunutí času člena federace budou přepočítány hodnoty GALT všech členů federace. Pokud je některá hodnota (resp. hodnoty) GALT zvýšena a příslušný člen (resp. členové) se nachází ve stavu *time advancing*, může to pro takového člena federace znamenat přidělení logického času. Tento čas bude vždy menší nebo roven hodnotě požadovaného času, což závisí na hodnotě požadovaného času a druhu požadované služby.

Metoda `computeAllGaltValues()` prochází seznam aktuálních členů federace a nad každým členem federace (mimo odesílatele požadavku) zavolá metodu `computeGalt()`. Následně prochází stejný seznam metoda `sendAllPossibleTimeAdvanceGrants()` a pro každého člena federace volá metodu `sendTimeAdvanceGrantIfPossible()`. Činnost této metody si zaslouží podrobnější vysvětlení:

⁷Podrobnější informace o použití nulového lookaheadu lze nalézt v [Kuh00], kapitola Advanced Topics.

```
protected void sendTimeAdvanceGrantIfPossible()
    throws FederateInternalError
{
    if (!inTimeAdvancingState)
    {
        return;
    }

    if (requestedTimeService == SERVICE_FLUSH_QUEUE_REQUEST)
    {
        sendReceiveOrderMessages();
        sendTimeStampMessagesAll();
        sendTimeAdvanceGrant(requestedLogicalTime);

        return;
    }
    ...
}
```

Nejprve se ověřuje, zda se člen federace nachází ve stavu *time advancing*. Pokud tato podmínka není splněna, činnost metody okamžitě končí. Pokud je požadovanou službou `flushQueueRequest()`, budou členu federace okamžitě (tj. bez ohledu na hodnotu GALT) doručeny všechny RO a TSO zprávy, které se nachází v příslušných frontách.

```
...
if (requestedTimeService == SERVICE_NEXT_MESSAGE_REQUEST
    || requestedTimeService == SERVICE_NEXT_MESSAGE_REQUEST_AVAILABLE)
{
    LogicalTime headTimeStamp = timeStampQueue.getHeadTimeStamp();

    if (headTimeStamp != null
        && headTimeStamp.compareTo(requestedLogicalTime) <= 0)
    {
        requestedLogicalTime = headTimeStamp;
    }
}

if (canSendTimeAdvanceGrant())
{
    sendReceiveOrderMessages();
    sendTimeStampMessages(requestedLogicalTime);
    sendTimeAdvanceGrant(requestedLogicalTime);
}
} // sendTimeAdvanceGrantIfPossible
```

Při použití služby `nextMessageRequest()` nebo `nextMessageRequestAvailable()` závisí hodnota přiděleného času na obsahu fronty TSO zpráv a na hodnotě požadovaného času. Pokud TSO fronta není prázdná a hodnota časového razítka první zprávy v této frontě je menší než hodnota požadovaného času, bude přidělena tato hodnota. V opačném případě bude přidělena hodnota, kterou člen federace uvedl jako parametr požadované služby. Rozhodování, zda je možné přidělit simulační čas, provádí metoda `canSendTimeAdvanceGrant()`. Tato metoda může bezpečně vrátit hodnotu `true`, pokud je hodnota času požadovaného členem federace ostře menší než aktuální hodnota GALT. V případě, že hodnota požadovaného času je rovna hodnotě GALT, je rozhodování poněkud složitější. Výsledek závisí na službách požadovaných jednotlivými členy federace a používaném lookaheadu. Podrobnosti lze nalézt ve zdrojovém kódu metody `canSendTimeAdvanceGrant()`.

6.3.4 Asynchronní doručování zpráv

Časově omezený člen federace dostává TSO i RO zprávy pouze ve stavu *time advancing*. Pro RO zprávy však lze použít tzv. asynchronní doručování zpráv, což umožňuje jejich okamžité doručování. Pokud člen federace zapne asynchronní doručování, budou mu RO zprávy doručovány ve stavu *time advancing* i *time granted*. Na TSO zprávy se asynchronní doručování nevztahuje.

Zapnutí asynchronního doručování se provádí službou `enableAsynchronousDelivery()`, vypnutí službou `disableAsynchronousDelivery()`. Pro naprogramování uvedených služeb je třeba ve třídě `XRTIAmbassador` změnit specifikátor přístupu k atributu `asynchronousDeliveryEnabled` z `private` na `protected`. S hodnotou atributu totiž potřebuje pracovat třída potomka `ExecutiveClientAmbassador`, která rozhoduje o tom, zda RO zpráva bude zařazena do fronty nebo okamžitě odeslána ke členovi federace.

6.4 Nové třídy pro time management

V předchozím textu bylo vysvětleno, jak jsou naprogramovány služby pro zapnutí (příp. vypnutí) časové regulace i omezení členů federace. Naprogramování těchto služeb si vyžádalo řadu úprav stávajících tříd XRTI. Kromě těchto úprav bylo nutné také navrhnout a naprogramovat některé nové třídy. Jejich stručný popis bude uveden v následujících odstavcích.

6.4.1 Třída `HLAMessage`

Třída `HLAMessage` reprezentuje obecnou zprávu, kterou může obdržet časově omezený člen federace. **Zpráva** vzniká tehdy, když člen federace zavolá některou z následujících služeb:

- `sendInteraction()` – poslání interakce. Na straně příjemce této zprávy odpovídá zpráva `receiveInteraction()`.
- `updateAttributeValues()` – aktualizace hodnot atributů. Na straně příjemce této zprávy odpovídá zpráva `reflectAttributeValues()`.
- `deleteObjectInstance()` – odstranění instance objektu. Na straně příjemce této zprávy odpovídá zpráva `removeObjectInstance()`.

Třída `HLAMessage` je abstraktní a slouží pouze jako společný předek pro třídy, které reprezentují tři druhy výše uvedených zpráv. Mezi nejdůležitější atributy třídy `HLAMessage` patří:

- `sentOrdering` – tato informace udává, zda byla zpráva odeslána jako TSO nebo RO.
- `logicalTime` – časové razítko zprávy v případě, že zpráva byla odeslána jako TSO.
- `transportationType` – druh přenosu zprávy: *reliable* nebo *best effort*.

Třída `HLAMessage` implementuje rozhraní `Comparable`. Metoda `compareTo()` je naprogramována tak, aby porovnávala hodnoty atributů `logicalTime`, což umožňuje řazení zpráv v TSO frontě (viz 6.4.4) podle hodnot časových razítek.

6.4.2 Třída `HLAMessageSendInteraction`

Třída `HLAMessageSendInteraction` je potomkem třídy `HLAMessage`. Jak název napovídá, instance této třídy slouží k uložení zprávy, která vznikne tehdy, když člen federace pošle interakci. Kromě atributů z rodičovské třídy obsahuje atributy typu `InteractionClassHandle` pro uložení identifikátoru interakční třídy a `ParameterHandleValueMap` pro uložení parametrů interakce.

6.4.3 Třídy `HLAMessageUpdateAttributeValues`, `HLAMessageDeleteObjectInstance`

Tyto třídy slouží pro zprávy, které vzniknou v souvislosti se službami `updateAttributeValues()` a `deleteObjectInstance()`. Obě třídy jsou naprogramovány, přestože podpora časových razítek pro tyto služby není dokončena⁸.

6.4.4 Třída `TimeStampQueue`

Třída `TimeStampQueue` slouží jako fronta pro TSO zprávy. Jedná se o prioritní frontu, uspořádanou podle hodnoty časových razítek položek fronty. Položky fronty jsou typu `HLAMessage` (viz 6.4.1). Pro uložení položek se využívá třída `LinkedList`.

Třída poskytuje následující metody:

- `enqueue(HLAMessage message)` – zařazení prvku do fronty. Po vložení prvku bude fronta seřazena voláním statické metody `sort()` třídy `Collections`.
- `dequeue()` – pokud fronta není prázdná, odstraní první prvek fronty a vrátí jeho hodnotu. Pokud je fronta prázdná, vrátí hodnotu `null`.
- `getHeadTimeStamp()` – vrátí časové razítko první zprávy ve frontě. Neodstraňuje tuto zprávu z fronty. Metoda je využívána při použití služeb `nextMessageRequestAvailable()` (viz 6.3.3) a `nextMessageRequest()`. Pro prázdnou frontu je vrácena hodnota `null`.
- `isEmpty()` – vrátí `true`, pokud je fronta prázdná.

Všechny uvedené metody jsou označeny jako `synchronized`, což zajišťuje bezpečný přístup z více vláken.

6.4.5 Třída `ReceiveOrderQueue`

Třída `ReceiveOrderQueue` slouží jako fronta pro RO zprávy. Třída je naprogramována jako jednoduchá FIFO fronta, s využitím třídy `LinkedList`. Třída poskytuje stejné metody jako třída `TimeStampQueue` (viz 6.4.4). Rozdíl je v tom, že `ReceiveOrderQueue` přidává prvky vždy na konec fronty metodou `addLast()` a toto pořadí prvků zůstává zachováno.

⁸Pro použití se simulační knihovnou J-Sim nejsou tyto služby využívány. Třídy byly napsány proto, aby nebránily rozšíření XRTI o podporu dalších služeb HLA.

6.5 Podpora časových razítek pro interakce

Aby bylo možné posílat a přijímat interakce jako TSO zprávy, musí být odstraněny překážky, které brání v použití časových razítek. První z těchto překážek je struktura `HLAbootstrapInteractionPayload`, která je při komunikaci používána jako „obálka“ pro všechny interakce. Tato struktura totiž neumožňuje uložení časového razítka a informace o způsobu odeslání interakce (RO nebo TSO). Druhým problémem je implementace metody `sendInteraction()` s časovým razítkem, která interakci dokonce vůbec neposílá:

```
public MessageRetractionReturn sendInteraction(
    InteractionClassHandle theInteraction,
    ParameterHandleValueMap theParameters,
    byte[] userSuppliedTag,
    LogicalTime theTime)
    throws ...
{
    verifyFederateIsExecutionMember();
    verifyNoSaveInProgress();
    verifyNoRestoreInProgress();

    return null;
}
```

Posledním problémem, který musí být vyřešen, je zajištění správného přijetí interakce s časovým razítkem na straně člena federace, kterému byla interakce doručena.

6.5.1 Úpravy struktury `HLAbootstrapInteractionPayload`

Třída `HLAbootstrapInteractionPayload` je využívána jako obálka interakcí při komunikaci mezi členem federace a XRTI. Třída obsahuje atributy, které jsou společné pro všechny interakce:

- `federationExecutionHandle` – celočíselný identifikátor vykonávané federace.
- `userSuppliedTag` – pole typu `byte`. Může obsahovat libovolné hodnoty, které byly uvedeny při odeslání interakce.
- `interactionClassHandle` – celočíselný identifikátor interakční třídy.
- `parameterHandleValuePairList` – seznam parametrů interakce. Obsahuje dvojice *parameter handle* a hodnota parametru.

Aby bylo možné při posílání interakcí používat časová razítka, je třeba tuto třídu rozšířit o další dva atributy:

- **sentOrdering** – tato informace udává, zda byla interakce odeslána jako TSO nebo RO.
- **theTime** – časové razítko v případě, že interakce byla odeslána jako TSO. Při použití RO nemá význam.

Doplnění těchto atributů nelze provést úpravou zdrojového kódu třídy, protože třída `HLAbootstrapInteractionPayload` (podobně jako některé další třídy, viz 6.1.5) je generována automaticky během překladu XRTI. Potřebnou úpravu struktury `HLAbootstrapInteractionPayload` je tedy nutné provést přímo v souboru `BootstrapObjectModel.xml`.

Aby bylo možné doplnit parametry **sentOrdering** a **theTime**, musí být nejprve specifikovány datové typy obou parametrů:

```
<enumeratedDataTypes>
...
    <enumeratedData name="HLAorderType"
                    representation="HLAinteger32BE"
                    semantics="Order type to be used for sending
                              attributes or interactions.">

        <enumerator name="Receive"
                    values="0"/>

        <enumerator name="TimeStamp"
                    values="1"/>

    </enumeratedData>
...

<arrayDataTypes>
...
<arrayData name="HLAlogicalTime"
            dataType="HLAbyte"
            cardinality="Dynamic"
            encoding="HLAvariableArray"
            semantics="An encoded logical time. An empty array
                      shall indicate that the values is not defined."/>
```

Nyní již nic nebrání rozšíření struktury `HLAbootstrapInteractionPayload` o dva nové parametry:

```
<fixedRecordDataTypes>

    <fixedRecordData name="HLAbootstrapInteractionPayload"
                      encoding="HLAfixedRecord"
                      semantics="The payload of the bootstrap
                                interaction.">

        ...

        <field name="sentOrdering"
                dataType="HLAorderType"
                semantics=""/>

        <field name="theTime"
                dataType="HLAlogicalTime"
                semantics=""/>

    </fixedRecordData>
```

Po překladu⁹ XRTI se lze přesvědčit, že zdrojový kód třídy `HLAbootstrapInteractionPayload` již obsahuje všechny potřebné atributy.

6.5.2 Úpravy třídy `XRTIAmbassador`

Ke každé interakci, která má být odeslána voláním metody `sendInteraction()` (bez ohledu na použití časového razítka), se vztahuje jeden ze dvou způsobů přenosu: *reliable* a *best effort*. Způsob přenosu pro konkrétní interakční třídu je dán modelem federace. Protože metoda `sendInteraction()` instance třídy `XRTIAmbassador` musí způsob přenosu respektovat, volá vždy jednu z následujících metod:

- `sendInteractionReliable()` – odešle interakci s použitím spolehlivého způsobu přenosu.
- `sendInteractionBestEffort()` – odešle interakci s použitím přenosu *best effort*.

Obě metody jsou označeny jako `protected`, protože nejsou určeny pro volání uživatelem třídy `XRTIAmbassador` a současně musí být k dispozici ve třídě potomka `ExecutiveClientAmbassador`.

⁹Překlad XRTI se nepodaří dokončit, což však v této situaci není chyba. Důvodem je odkaz na konstruktor třídy `HLAbootstrapInteractionPayload` ve třídě `XRTIAmbassador`, u kterého po provedení úprav nesouhlasí očekávaný počet parametrů. Uvedená „chyba“ je samozřejmě opravena při následujících úpravách třídy `XRTIAmbassador`.

Je zřejmé, že uvedené metody musí být volány rovněž metodou `sendInteraction()`, která umožní použití časového razítka. Obě metody proto musí být rozšířeny o stejné parametry jako struktura `HLAbootstrapInteractionPayload`. Rozšíření metody `sendInteractionReliable()` je uvedeno v následující části kódu (metoda `sendInteractionBestEffort()` byla upravena analogickým způsobem):

```
protected void sendInteractionReliable(
    InteractionClassHandle theInteraction,
    ParameterHandleValueMap theParameters,
    byte[] userSuppliedTag,
    OrderType sentOrdering,
    LogicalTime theTime)
    throws ...
{
    ...

    if (sentOrdering.equals(OrderType.RECEIVE))
    {
        hlaSentOrdering = HLAorderType.Receive;
        encodedTime = new byte[0];
    }
    else // sentOrdering.equals(OrderType.TIMESTAMP)
    {
        hlaSentOrdering = HLAorderType.TimeStamp;
        encodedTime = new byte[ theTime.encodedLength() ];
        theTime.encode(encodedTime, 0);
    }

    HLAbootstrapInteractionPayload payload =
        new HLAbootstrapInteractionPayload(
            joinedFederationExecutionHandle,
            userSuppliedTag,
            ((XRTIInteractionClassHandle)theInteraction).getIdentifier(),
            parameterHandleValuePairs,
            hlaSentOrdering,
            encodedTime
        );

    try
    {
        HLAEncodingOutputStream hlaeos = new HLAEncodingOutputStream(
            channel.getOutputStream()
        );

        synchronized(channel)
```

```
        {
            hlaeos.writeHLAinteger32BE(MAGIC_NUMBER);
            hlaeos.writeHLAinteger32BE(bootstrapObjectModelVersion);

            payload.encode(hlaeos);

            hlaeos.flush();
        }
    }
    ...
}
```

Upravená metoda `sendInteraction()`, která nepoužívá časové razítko, dosadí za parametr `sentOrdering` hodnotu `OrderType.RECEIVE` a časové razítko nevyplní:

```
public void sendInteraction(InteractionClassHandle theInteraction,
                           ParameterHandleValueMap theParameters,
                           byte[] userSuppliedTag)
    throws ...
{
    ...
    if(icd.getTransportation().equals(TransportationType.HLA_RELIABLE))
    {
        sendInteractionReliable(
            theInteraction,
            theParameters,
            userSuppliedTag,
            OrderType.RECEIVE,
            null
        );
    }
    else // icd.getTransportation().equals(TransportationType.HLA_BEST_EFFORT)
    {
        sendInteractionBestEffort(
            theInteraction,
            theParameters,
            userSuppliedTag,
            OrderType.RECEIVE,
            null
        );
    }
    ...
}
```

Zbývá naprogramovat přetíženou metodu `sendInteraction()` s časovým razítkem. Před odesláním interakce se musí ověřit platnost časového razítka, což

provede metoda `verifyTimeStampIsValid()`, která při neplatném razítku vyhodí výjimku `InvalidLogicalTime`. Aby bylo možné interakci odeslat jako TSO, musí mít příslušná třída v modelu federace deklarováno pořadí `TimeStamp` a člen federace musí být časově regulující. Pokud nejsou tyto podmínky splněny, bude interakce převedena na RO a časové razítko bude ignorováno.

```
public MessageRetractionReturn sendInteraction(
    InteractionClassHandle theInteraction,
    ParameterHandleValueMap theParameters,
    byte[] userSuppliedTag,
    LogicalTime theTime)
    throws ...
{
    ...
    verifyTimeStampIsValid(theTime);

    OrderType sentOrdering;

    if (timeRegulationEnabled
        && icd.getOrder().equals(OrderType.TIMESTAMP))
        sentOrdering = OrderType.TIMESTAMP;
    else
        sentOrdering = OrderType.RECEIVE;

    try
    {
        if (icd.getTransportation().equals(TransportationType.HLA_RELIABLE))
        {
            sendInteractionReliable(theInteraction,
                                    theParameters, userSuppliedTag,
                                    sentOrdering, theTime
                                );
        }
        else
        {
            // icd.getTransportation().equals(TransportationType.HLA_BEST_EFFORT)
            {
                sendInteractionBestEffort(theInteraction,
                                           theParameters, userSuppliedTag,
                                           sentOrdering, theTime
                                       );
            }
        }
    }
}
```

Aby bylo možné zajistit správné přijetí interakce s časovým razítkem, je třeba částečně změnit chování metody `interpretReceivedMessage()` ve třídě `XRTIAmbassador`. Tato metoda je deklarována jako `private` a po do-

ručení interakce volá přímo metodu `receiveInteraction()` nad objektem `proxyAmbassador`. Uvedené chování metody beze změny dědí třída potomka `ExecutiveClientAmbassador`. Pro zavedení služeb *time management* je však nezbytné, aby se třídy `XRTIAmbassador` a `ExecutiveClientAmbassador` při přijetí interakce chovaly poměrně odlišným způsobem, což současné řešení neumožňuje. Z tohoto důvodu bylo „přímé“ volání metody `receiveInteraction()` nahrazeno voláním nově vytvořené metody¹⁰, která má následující hlavičku:

```
protected void proxyAmbassador_receiveInteraction(  
    InteractionClassHandle interactionClass,  
    ParameterHandleValueMap theParameters,  
    byte[] userSuppliedTag,  
    OrderType sentOrdering,  
    TransportationType theTransport,  
    LogicalTime theTime,  
    OrderType receivedOrdering)
```

Tuto metodu lze ve třídě potomka `ExecutiveClientAmbassador` překrýt. Přestože metoda `interpretReceivedMessage()` zůstává `private`, za běhu programu bude volat vždy „správnou“ metodu pro přijetí interakce.

Ve třídě `XRTIAmbassador` musí tato metoda být naprogramována způsobem, který zaručuje splnění následující podmínky: člen federace může obdržet TSO zprávu pouze tehdy, když zpráva byla odeslána jako TSO a současně je příjemce zprávy časově omezený. Při nesplnění uvedené podmínky bude zpráva (zde interakce) doručena jako RO zpráva. Řešení je uvedeno v následujícím úseku kódu:

```
if (sentOrdering.equals(OrderType.TIMESTAMP) && timeConstrainedEnabled)  
{  
    proxyAmbassador.receiveInteraction(interactionClass,  
        theParameters, userSuppliedTag,  
        sentOrdering, theTransport,  
        theTime,  
        OrderType.TIMESTAMP);  
}  
else  
{  
    proxyAmbassador.receiveInteraction(interactionClass,  
        theParameters, userSuppliedTag,  
        sentOrdering, theTransport);  
}
```

¹⁰V názvu metody je skutečně znak podtržení.

6.5.3 Úpravy třídy `ExecutiveClientAmbassador`

Každá interakce, kterou člen federace pošle prostřednictvím instance třídy `XRTIAmbassador`, bude po přenesení jedním z komunikačních kanálů doručena příslušné instanci třídy `ExecutiveClientAmbassador`. Tato instance je zodpovědná za doručení interakce všem členům federace, kteří se přihlásili k odběru dané interakční třídy. Důsledkem zavedení služeb *time management* je, že interakce určené pro časově omezeného člena federace již RTI nesmí doručovat okamžitě, ale musí provádět jejich zařazování do TSO nebo RO fronty zpráv. Kromě nastavení časového omezení jednotlivých členů federace musí RTI brát ohled také na případné asynchronní doručování zpráv (viz 6.3.4). Obecně platí, že různí členové federace mohou mít v jednom okamžiku tyto přepínače nastavené různými způsoby, což znamená, že může nastat situace, kdy stejná interakce bude některém členovi federace vložena do fronty, zatímco jinému členovi federace bude doručena okamžitě.

Po přijetí interakce metodou `proxyAmbassador_receiveInteraction()` se interakce dostane k instanci třídy `FederationExecutionAmbassador`, která reprezentuje aktuální federaci. Protože zástupce federace udržuje seznam aktuálních členů federace (viz 6.1.3), může být interakce předána ostatním členům federace s výjimkou odesílatele. Tím se interakce dostává ke všem registrovaným instancím třídy `ExecutiveClientAmbassador`. Předání interakce ostatním členům federace provádí metody `sendInteractionReliable()` a `sendInteractionBestEffort()`, které jsou k tomuto účelu ve třídě `ExecutiveClientAmbassador` překryty. Kód metody `sendInteractionReliable()` je uveden v následujícím výpisu:

```
protected void sendInteractionReliable(
    InteractionClassHandle theInteraction,
    ParameterHandleValueMap theParameters,
    byte[] userSuppliedTag,
    OrderType sentOrdering,
    LogicalTime theTime)
    throws ...
{
    if(subscribedInteractionClasses.contains(theInteraction))
    {
        boolean b1 = descriptorManager.isBootstrapHandle(theInteraction);
        boolean b2 = sentOrdering.equals(OrderType.RECEIVE)
            && asynchronousDeliveryEnabled;
        boolean asynchronous = b1 || b2;
```

Nejprve se kontroluje, zda je člen federace přihlášen k odběru interakční třídy. Pokud tomu tak není, činnost metody končí. Pokud je člen federace časově omezený a interakce nemá být doručena asynchronním způsobem (tedy okamžitě),

musí být vytvořena instance třídy `HLAMessageSendInteraction` (viz 6.4.2), která bude vložena do příslušné fronty podle hodnoty `sentOrdering`:

```
if (timeConstrainedEnabled && !asynchronous)
{
    // enqueue message into the queue
    HLAMessageSendInteraction message =
        new HLAMessageSendInteraction(
            theInteraction,
            theParameters, userSuppliedTag,
            sentOrdering,
            TransportationType.HLA_RELIABLE,
            theTime);

    if (sentOrdering.equals(OrderType.TIMESTAMP))
    {
        timeStampQueue.enqueue(message);
    }
    else // (sentOrdering.equals(OrderType.RECEIVE))
    {
        receiveOrderQueue.enqueue(message);
    }
}
```

V opačném případě bude interakce odeslána ihned¹¹ bez použití fronty. K odeslání interakce se volá metoda předka `sendInteractionReliable()`, která interakci zapíše do příslušného komunikačního kanálu.

```
else // (!timeConstrainedEnabled || asynchronous)
{
    // send message immediately without using a queue
    super.sendInteractionReliable(theInteraction,
                                  theParameters,
                                  userSuppliedTag,
                                  sentOrdering, theTime);
}
} // sendInteractionReliable
```

Kód metody `sendInteractionBestEffort()` je analogický.

¹¹Uvedenou (a žádnou další) činnost prováděla původní implementace metody.

6.6 Další funkce XRTI

Dosud popsané úpravy software XRTI se týkaly výhradně služeb skupiny *time management* a související problematiky časových razítek. Kromě těchto služeb byly do XRTI přidány ještě některé další funkce:

- **Služby pro synchronizaci federace** – tyto služby patří do skupiny *federation management* (viz 3.5.2). Původní verze XRTI tyto služby nepodporovala.
- **Grafické uživatelské rozhraní** aplikace *XRTIExecutive*. Protože původní verze XRTI běžela pouze v textovém režimu a všechny informace o činnosti aplikace byly vypisovány pouze na standardní (příp. standardní chybový) výstup, bylo navrženo jednoduché grafické uživatelské rozhraní (GUI). Hlavní okno aplikace zobrazuje seznam federací včetně aktuálně připojených členů federace, dále je možné zobrazit okno s informacemi o činnosti aplikace (*debug window*). Pro použití GUI je třeba při spuštění aplikace *XRTIExecutive* zadat na příkazovém řádku volbu `-gui` (jinak bude aplikace spuštěna v textovém režimu).

Kapitola 7

Závěr

Cílem této práce bylo umožnit snadnou distribuovatelnost simulací podle standardu High Level Architecture (HLA), prováděných s využitím simulační knihovny J-Sim. Teoretická část práce obsahuje stručný popis knihovny J-Sim a hlavních principů HLA.

Navrhované řešení je poměrně rozsáhlé a skládá se ze dvou hlavních částí. První část se zabývá implementací podpory HLA pro simulační knihovnu J-Sim. Za tímto účelem byla knihovna J-Sim rozšířena o nový balík `cz.zcu.fav.kiv.jsim.hla`, který obsahuje třídy potřebné pro vytvoření distribuované simulace a zajištění komunikace s implementací Runtime Infrastructure (RTI). Kromě naprogramování nových tříd bylo třeba také provést některé úpravy stávajících tříd knihovny J-Sim. Realizované řešení je plně funkční a rovněž splňuje požadavek na zachování zpětné kompatibility s nedistribuovanými simulacemi. Třídy potřebné pro distribuovanou simulaci byly navrženy takovým způsobem, aby uživatel knihovny J-Sim byl schopen navrhnout distribuovanou simulaci i v případě, že nemá hluboké znalosti problematiky HLA. Poměrně jednoduchým způsobem lze také provést změnu nedistribuované simulace na distribuovanou (a naopak).

Druhá část řešení se zabývá implementací Extensible Run-Time Infrastructure (XRTI), která slouží jako server pro distribuovanou simulaci podle specifikace HLA 1516. Tato implementace byla použita pro práci s J-Sim. Současná verze XRTI neobsahuje implementaci služeb skupiny Time Management, které jsou nezbytné pro použití se simulační knihovnou J-Sim, což si vyžádalo provedení určitých úprav XRTI. Přestože se předpokládalo, že půjde pouze o menší opravy a vylepšení, během realizace úprav se ukázalo, že splnění tohoto úkolu je mnohem obtížnější než rozšíření knihovny J-Sim. Do XRTI byly doplněny služby pro nastavení časové regulace i omezení členů federace, služby pro přidělení logického času, fronty pro TSO a RO zprávy a některé další funkce. Upravenou verzi XRTI je možné použít společně s knihovnou J-Sim. Přestože většina úprav byla provedena

právě s ohledem na použití s knihovnou J-Sim, nové služby jsou implementovány zcela obecně v souladu s HLA a realizace těchto úprav by neměla bránit případnému dalšímu vývoji XRTI.

Rozšíření knihovny J-Sim a úpravy XRTI se staly zdrojem mnoha zkušeností s programováním v jazyce Java a zejména s prací na rozsáhlejších projektech. Rovněž je třeba poukázat na velký význam dostupnosti zdrojových kódů podobných projektů. Bez jejich využití by tato práce vůbec nemohla vzniknout. Z tohoto důvodu budou zdrojové kódy¹ celé práce k dispozici ke stažení na Internetu. Autor práce věří, že se mohou stát užitečnými zdroji informací i základem pro další projekty související s problematikou distribuovaných simulací.

¹Součástí zdrojových kódů obou projektů jsou dokumentační komentáře (v angličtině), ze kterých lze vygenerovat příslušnou dokumentaci.

Zkratky

HLA	High Level Architecture
RTI	Runtime Infrastructure
XRTI	The Extensible Run-Time Infrastructure
OMT	Object Model Template
BOM	Bootstrap Object Model
MOM	Management Object Model
FOM	Federation Object Model
FDD	FOM Document Data
FED	Federation Execution Data
TSO	Time Stamp Order
RO	Receive Order
FIFO	First In First Out
GALT	Greatest Available Logical Time
DMSO	Defense Modeling and Simulation Office
IEEE	Institute of Electrical and Electronic Engineers

Literatura

- [Kac01] *Kačer, J.: J-Sim: A Java-based Tool for Discrete Simulation*
Plzeň, 2001
- [J-Sim] **J-Sim Home Page**
<http://www.j-sim.zcu.cz/>
- [Kuh00] *Kuhl, F. – Weatherly, R. – Dahman J.: Creating Computer Simulation Systems: An Introduction to the High Level Architecture*
Prentice Hall PTR, USA, 2000
- [Kap03] *Kapolka, A.: The Extensible Runtime-Infrastructure (XRTI): An experimental implementation of Proposed Improvements to the High Level Architecture*
Monterey, USA, 2003
- [XRTI] **The Extensible Runtime-Infrastructure (XRTI)**
<http://www.npsnet.org/~npsnet/xrti/>
- [CSU1] *McLeod Institute of Simulation Sciences: HLA Module 1 – Basic Concepts of the High Level Architecture (HLA)*
California State University, Chico, 2001
<http://www.ecst.csuchico.edu/~hla/courses.html>
- [CSU2] *McLeod Institute of Simulation Sciences: HLA Module 2 – Advanced topics*
California State University, Chico, 1999
<http://www.ecst.csuchico.edu/~hla/courses.html>
- [HLA98] **HLA Interface Specification, version 1.3**
U.S. Department of Defense (DMSO), 1998
- [Spe02] *Spell, B.: Java Programujeme profesionálně*
Computer Press, Praha, 2002
- [Her03] *Herout, P.: Java – Bohatství knihoven*
Nakladatelství KOPP, České Budějovice, 2003

LITERATURA

- [Rac02] *Racek, S.*: **Pravděpodobností modely počítačů**
Plzeň, 2002
- [Jir04] *Jirásek, M.*: **Implementace Runtime Infrastructure standardu HLA pro použití v simulační knihovně C-Sim**
Diplomová práce, Plzeň, 2004
- [Java] **Java 2 Platform, Standard Edition (J2SE)**
<http://java.sun.com/j2se/>

Příloha A

Uživatelská příručka k JSimHLA

Tato příručka poskytuje uživateli simulační knihovny J-Sim informace potřebné k vytvoření distribuované simulace s využitím této knihovny. Simulační knihovna J-Sim umožňuje použití distribuované simulace podle standardu High Level Architecture (HLA). Problematika HLA je poměrně komplikovaná, její znalost však pro návrh distribuované simulace s použitím simulační knihovny J-Sim není vyžadována. Tento text vysvětluje pouze rozdíly mezi použitím distribuované simulace a nedistribuovaných verzí knihovny J-Sim. Předpokládá se základní znalost použití knihovny J-Sim.

A.1 Vytvoření distribuované simulace

Při vytváření distribuované simulace podle HLA s použitím simulační knihovny J-Sim je třeba v každém simulačním programu importovat balík `cz.zcu.fav.kiv.jsim.hla`:

```
import cz.zcu.fav.kiv.jsim.*;
import cz.zcu.fav.kiv.jsim.hla.*;
```

A.1.1 Vytvoření objektu simulace

Vytvoření objektu simulace pro distribuovanou simulaci je stejné jako pro nedistribuovanou simulaci, s tím rozdílem, že místo třídy `JSimSimulation` se použije třída `JSimHLASimulation`. Tato třída je odvozena od třídy `JSimSimulation` a rozšiřuje schopnosti rodičovské třídy o možnost použití distribuované simulace.

Konstruktor této třídy vytvoří simulační objekt pojmenovaný podle hodnoty parametru `name` (stejně jako u třídy předka). Kromě parametru `name` má konstruktor ještě další dva parametry:

```
public JSimHLASimulation(String name,  
                          Properties configurationRTI,  
                          int numberOfFederates)  
    throws JSimInvalidParametersException
```

Parametr `configurationRTI` umožňuje zadání konfiguračních parametrů pro implementaci RTI. Pro většinu simulací lze uvést pouze `new Properties()` bez dalšího nastavování, což znamená, že se použijí výchozí hodnoty.

Celočíselný parametr `numberOfFederates` udává počet J-Sim simulací (v terminologii HLA počet členů federace), které se budou podílet na distribuované simulaci (v terminologii HLA se distribuovaná simulace nazývá federace). Hodnota parametru menší než dva nemá smysl a pokus o zadání takové hodnoty způsobí vyhození výjimky. Při zadávání počtu členů federace je třeba si uvědomit, že nastavení této hodnoty může provést pouze účastník simulace, který distribuovanou simulaci vytvořil. Tento účastník bude následně čekat na připojení zadaného počtu členů federace. Případný další pokus o zadání jiné hodnoty již nemá na vytvořenou federaci žádný vliv. Vzhledem k tomu, že kód jednotlivých účastníků simulace může být spouštěn současně na více počítačích, není možné předem stanovit, která simulace tuto hodnotu skutečně nastaví. Z tohoto důvodu je vhodné, aby u všech účastníků distribuované simulace byla zadána stejná hodnota.

Příklad vytvoření instance třídy `JSimHLASimulation`:

```
simulation = new JSimHLASimulation(  
    "JSim HLA Example Simulation", new Properties(), 2);
```

A.1.2 Lokální a vzdálené objekty simulace

Na distribuovanou diskrétní simulaci je možné (zjednodušeně) nahlížet jako na běžnou diskrétní simulaci s jedním významným rozdílem: všechny objekty v simulaci jsou buď **lokální** (*local*) nebo **vzdálené** (*remote*).

Simulační knihovna J-Sim s podporou HLA umožňuje použití těchto vzdálených objektů:

- `JSimHLARemoteProcess` – vzdálený proces
- `JSimHLARemoteHead` – vzdálená fronta

A.1.3 Třída `JSimHLARemoteProcess`

Třída `JSimHLARemoteProcess` reprezentuje **vzdálený proces**. Konstruktor této třídy vyžaduje zadání stejných parametrů jako konstruktor třídy `JSimProcess`.

Rozdíl je v datovém typu parametru `parent`, kde je vyžadován odkaz na instanci třídy `JSimHLASimulation` místo třídy `JSimSimulation`.

```
public JSimHLARemoteProcess(String name, JSimHLASimulation parent)
    throws JSimInvalidParametersException,
    JSimSimulationAlreadyTerminatedException
```

Aby bylo možné zajistit správnou funkci distribuované simulace, musí být splněna následující podmínka: **Každý vzdálený proces musí existovat jako lokální v simulačním programu právě jednoho účastníka distribuované simulace.** Prakticky to znamená, že proces se stejným jménem jako instance třídy `JSimHLARemoteProcess` musí vždy existovat jako instance třídy `JSimProcess` (nebo instance potomka třídy `JSimProcess`) v simulačním programu jiného účastníka simulace.

Vzdálený proces lze používat stejným způsobem jako lokální proces, který je instancí třídy `JSimProcess` (nebo potomka této třídy). Příklad použití vzdáleného procesu:

```
JSimHLARemoteProcess process3 =
    new JSimHLARemoteProcess("Process 3", simulation);

...
process3.activate(2.5);
```

Instance třídy `JSimHLARemoteProcess` nabízí uživateli následující metody:

- `activate()` – naplánuje vzdálený proces na požadovaný čas.
- `cancel()` – zruší naplánování vzdáleného procesu.
- `isIdle()` – zjistí, zda vzdálený proces může být naplánován.
- `getState()` – zjistí stav vzdáleného procesu.

Metody vzdálených procesů vrací stejné výsledky jako metody lokálních procesů. V případě výskytu chyby také vyhazují stejné výjimky. Použití všech výše uvedených metod je tedy naprosto stejné, jako použití stejně pojmenovaných metod třídy `JSimProcess`.

A.1.4 Třída `JSimHLARemoteHead`

Třída `JSimHLARemoteHead` reprezentuje **vzdálenou frontu**. Konstruktor této třídy vyžaduje stejné parametry jako konstruktor třídy `JSimHead`. Jediný rozdíl je u parametru `parent`, který je typu `JSimHLASimulation` místo `JSimSimulation`.

Každá vzdálená fronta musí existovat jako lokální v simulačním programu právě jednoho účastníka distribuované simulace.

Instance třídy `JSimHLARemoteHead` nabízí uživateli následující metody:

- `empty()` – test, zda je vzdálená fronta prázdná.
- `cardinal()` – vrací počet prvků ve vzdálené frontě.
- `first()` – vrací první prvek vzdálené fronty.
- `last()` – vrací poslední prvek vzdálené fronty.
- `getLw()` – vrací střední délku vzdálené fronty.
- `getTw()` – vrací střední dobu čekání ve vzdálené frontě pro všechny prvky, které byly odstraněny z fronty.
- `getTwForAllLinks()` – vrací střední dobu čekání ve vzdálené frontě pro všechny prvky, které byly vloženy do fronty.

Všechny metody vzdálené fronty vrací stejné výsledky jako metody běžné (lokální) fronty.

Příklad použití vzdálené fronty:

```
JSimHLARemoteHead queue5 =  
    new JSimHLARemoteHead("Queue 5", simulation);  
  
...  
JSimLink myLink = new JSimLink();  
queue5.into(myLink);
```

Instance třídy `JSimLink` lze používat stejným způsobem také pro vzdálené fronty. Jediný problém může nastat, pokud při vytvoření instance třídy `JSimLink` budou specifikována tzv. uživatelská data. K tomuto účelu slouží přetížený konstruktor s parametrem typu `Object`. Aby bylo možné přenést prvek fronty do vzdálené fronty (případně obráceně), musí být přenesena také tato data. K přenesení dat se používá mechanismus serializace objektů jazyka Java. Prakticky to znamená, že uživatelská data, která budou používána se vzdálenými frontami, musí implementovat rozhraní `java.io.Serializable`. Pokud toto rozhraní nebude implementováno, při prvním pokusu o vložení prvku do vzdálené fronty, bude vyhozena výjimka, protože objekt se nepodaří serializovat. Další informace lze nalézt v dokumentaci knihovny J-Sim s podporou HLA.

A.1.5 Zahájení a ukončení distribuované simulace

Vykonávání distribuované simulace se provádí opakovaným voláním metody `step()` podobně jako u nedistribuované simulace. Část kódu simulačního programu distribuované simulace je uvedena v následujícím výpisu:

```
// begin federation execution
simulation.beginFederationExecution();

// main simulation loop
System.out.println("Some simulation steps...");
while (simulation.step() == true)
{
    if (simulation.getCurrentTime() >= SIMULATION_FINAL_TIME)
        break;
}

// end federation execution
simulation.endFederationExecution();
```

Z uvedeného kódu je zřejmé, že distribuovaná simulace (ve srovnání s nedistribuovanou simulací) vyžaduje použití následujících dvou metod:

- `beginFederationExecution()` – Zahajuje vykonávání federace pro J-Sim. Musí být zavolána *po* vytvoření všech lokálních i vzdálených objektů simulace a *před* prvním voláním metody `step()`.
- `endFederationExecution()` – Ukončuje vykonávání federace pro J-Sim. Musí být zavolána *po* posledním voláním metody `step()`.

A.2 Spuštění distribuované simulace

Před spuštěním distribuované simulace je nutné spustit aplikaci **RTI Executive**. Aplikace RTI Executive je součástí použité implementace Runtime-Infrastructure. Pro knihovnu J-Sim se používá implementace **Extensible Runtime-Infrastructure** (XRTI). Další informace lze nalézt v uživatelské příručce k XRTI.

A.3 Nastavení proměnné CLASSPATH

Před použitím J-Sim je vhodné nastavit proměnnou prostředí `CLASSPATH` tak, aby obsahovala cestu k souboru `jsimhla.jar`, který obsahuje přeložené třídy knihovny J-Sim.

Současně je třeba nastavit i cestu k třídám použité implementace Runtime-Infrastructure. Pro implementaci Extensible Runtime-Infrastructure (XRTI) se jedná o soubor `xrti.jar`.

Příklad nastavení proměnné `CLASSPATH` pro MS Windows (cesty se oddělují středníkem):

```
set CLASSPATH=%CLASSPATH%;C:\somepath\jsimhla\archives\jsimhla.jar
set CLASSPATH=%CLASSPATH%;C:\somepath\xrti\archives\xrti.jar
```

Příklad nastavení `CLASSPATH` pro operační systémy typu UNIX (cesty se oddělují dvojtečkou):

```
set CLASSPATH=$CLASSPATH:/somepath/jsimhla/archives/jsimhla.jar
set CLASSPATH=$CLASSPATH:/somepath/xrti/archives/xrti.jar
export CLASSPATH
```

Pokud nebude proměnná prostředí `CLASSPATH` nastavena, bude nutné tyto cesty zadávat při překladu a spuštění každého simulačního programu.

A.4 Překlad projektu

Součástí knihovny J-Sim s podporou HLA je soubor `jsimhla.jar`, který obsahuje všechny třídy přeložené překladačem jazyka Java. Tento soubor se nachází v podadresáři `archives`. Vzhledem k tomu, že překladač jazyka Java překládá třídy do byte kódu, který je nezávislý na platformě, není třeba třídy znovu překládat. Pokud si však celý projekt chcete sami přeložit, doporučuje se za tímto účelem použít Apache Ant (<http://ant.apache.org/>).

Pro zadání všech uvedených příkazů se předpokládá, že aktuálním adresářem je adresář projektu (v tomto adresáři se nachází soubor `build.xml`).

Překlad celého projektu:

```
ant build
```

Překlad celého projektu včetně vygenerování API dokumentace:

```
ant release
```

Odstranění souborů, které vznikly při překladu (třídy, archivky, dokumentace):

```
ant clean
```

Další informace lze získat po zadání příkazu `ant info` nebo přímo v souboru `build.xml`.

Příloha B

Uživatelská příručka k XRTI

Extensible Run-Time Infrastructure (XRTI) je volně šiřitelná implementace specifikace rozhraní High Level Architecture IEEE 1516.1. Vzhledem k tomu, že standard HLA je velmi komplikovaný a rozsáhlý, XRTI neimplementuje kompletní specifikaci rozhraní, ale pouze určitou část.

K projektu XRTI jsou dostupné zdrojové kódy v jazyce Java. Při dodržení podmínek licence mohou být zdrojové kódy modifikovány a rozšiřovány (licence se nachází v souboru `linense.txt` v podadresáři `collateral`).

Tato příručka obsahuje pouze informace, které jsou nezbytně nutné pro použití XRTI se simulační knihovnou J-Sim. Podrobnější informace lze nalézt v dokumentaci projektu XRTI.

B.1 Spuštění XRTIExecutive

Aplikace **XRTIExecutive** musí být spuštěna před spuštěním simulačních programů jednotlivých členů federace. Před spuštěním XRTI je vhodné nastavit proměnnou prostředí `CLASSPATH` tak, aby obsahovala cestu k souboru `xrti.jar`, který obsahuje přeložené třídy XRTI. Pokud nebude proměnná prostředí `CLASSPATH` nastavena, bude nutné cestu zadat při každém spuštění XRTIExecutive.

XRTIExecutive se spouští příkazem:

```
java org.npsnet.xrti.XRTIExecutive
```

Alternativně je možné spuštění v grafickém uživatelském režimu:

```
java org.npsnet.xrti.XRTIExecutive -gui
```

B.2 Překlad projektu

Součástí projektu XRTI je soubor `xrti.jar`, který obsahuje všechny třídy přeložené překladačem jazyka Java. Tento soubor se nachází v podadresáři `archives`.

Třídy projektu není třeba znovu překládat. Pro případný překlad projektu se doporučuje použít Apache Ant (<http://ant.apache.org/>). Provedení překladu bez nástroje Apache Ant je teoreticky možné, ale prakticky poměrně obtížné.

Pro zadání všech uvedených příkazů se předpokládá, že aktuálním adresářem je adresář projektu (v tomto adresáři se nachází soubor `build.xml`).

Překlad celého projektu:

```
ant build
```

Překlad celého projektu včetně vygenerování API dokumentace:

```
ant release
```

Odstranění souborů, které vznikly při překladu (třídy, archivy, dokumentace):

```
ant clean
```

Další informace lze získat po zadání příkazu `ant info` nebo přímo v souboru `build.xml`.