UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Softwarový nástroj pro konfigurování distribuovaných simulačních modelů

Bc. Jiří Pénzeš

Diplomová práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jiří Pénzeš**
Osobní číslo:	**I11402**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Softwarový nástroj pro konfigurování distribuovaných simulačních modelů**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V úvodní části práce je nutné provést přehled problematiky distribuované simulace.

Primárním cílem diplomové práce je návrh, implementace a ověření softwarového nástroje pro konfigurování distribuovaných simulačních modelů. Navržené řešení bude otestováno na modelu vybraného typu dopravního nebo obslužného systému.

Pro potřeby konfigurování distribuovaných simulačních modelů bude vyvinuto jednoduché grafické prostředí umožňující definování konfigurace modelu s uplatněním deklarativního přístupu.

Pro účely implementace se předpokládá využívání jazyka Java.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. KAVIČKA, A., KLIMA, V., ADAMKO, N. Agentovo orientovaná simulácia dopravných uzlov. Žilina: EDIS - vydavateľstvo ŽU, 2005. ISBN 80-8070-477-5.
2. FUJIMOTO, R.M. Parallel and distributed simuilation systems, New York: John Wiley & Sons, 2000. ISBN 0-471-18383-0.
3. BANKS, J. Handbook of simulation. New York: John Wiley & Sons, 1998. ISBN 0-471-13403-1.

Vedoucí diplomové práce: **prof. Ing. Antonín Kavička, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2012**
Termín odevzdání diplomové práce: **17. května 2013**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

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

V Pardubicích dne 15. listopadu 2012

Prohlášení autora

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

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

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 17. 05. 2013 Bc. Jiří Pénzeš

Poděkování

Touto cestou bych rád poděkoval všem, kteří mě při tvorbě mé práce podporovali nebo mi jakkoliv pomohli. Zejména pak mému vedoucímu prof. Ing. Antonínovi Kavičkovi, Ph.D., který mi umožnil toto téma realizovat a při jeho realizaci mi dával cenné rady. Dále bych rád poděkoval mé rodině a přátelům za podporu během studia.

Tato diplomová práce vznikla v rámci řešení projektu „Podpora stáží a odborných aktivit při inovaci oblasti terciárního vzdělávání na DFJP a FEI Univerzity Pardubice, reg. č.: CZ.1.07/2.4.00/17.0107“ v týmu „TRANSIM simulace dopravních a obslužných systémů“.

Anotace

Diplomová práce se zabývá problematikou tvorby distribuované simulace. V teoretické části jsou představeny základní konzervativní metody synchronizace. Zejména algoritmus zasílání zpráv na vyžádání. Dále se práce zabývá implementačními prostředky a technikami, které se využívají při tvorbě distribuovaných aplikací. Se zaměřením na platformu Java a technologii Java RMI.

V další části se práce zabývá konkrétní implementací synchronizačního algoritmu a deklarativním zápisem distribuovaných simulačních modelů. Dále v praktické části je popsáno vývojové prostředí jDistsim IDE, které je hlavním výsledkem této práce. Aplikace slouží pro tvorbu distribuovaných simulačních modelů.

Klíčová slova

Distribuovaná simulace, konzervativní metody synchronizace, deklarativní zápis, distribuované aplikace, platforma Java, Java RMI, konkurenční programování, návrhové vzory, distribuované simulační modely

Title

Software application for configuration of distributed simulated models.

Annotation

My thesis describes possibilities of developing distributed simulation. In the theoretical part are presented the basic conservative synchronization methods. Mainly NullMessage algorithm is introduced. The thesis deals with the implementing options and techniques for distributed application development. With a focus on Java and Java RMI.

The second part deals with the specific implementation of the synchronized algorithm and declarative notation of distributed simulation models. In the practical part is described the development environment jDistsim IDE, which is the main result of this work. The application is for the development of distributed simulation models.

Keywords

Distributed simulation, conservative synchronization, declarative notation, distributed application, Java platform, Java RMI, concurrent programming, design patterns, distributed simulation models

Obsah

Seznam zkratek

API	Application Programming Interface (rozhraní pro programování aplikací)
DS	Distributed simulation (distribuovaná simulace)
CORBA	Common Object Request Broker Architecture
GUI	Graphical User Interface (grafické uživatelské rozhraní)
IDE	Integrated Development Environment (integrované vývojové prostředí)
JDK	Java development kit
JRMP	Java Remote Method Protocol (Java protokol vzdáleného volání metod)
JVM	Java Virtual Machine (Java virtuální stroj)
LBTS	Lower Bound Time Stamp (dolní hranice časového razítka)
LVT	Local Virtual Time (lokální virtuální čas)
LCC	Local Causality Constraint (podmínka lokální kauzality)
LP	Logical Process (logický proces)
MVC	Model View Controller (Model – Pohled – Řadič)
OOP	Object-oriented programming (objektově orientované programování)
RMI	Remote Method Invocation (vzdálené volání metod)
RMI-IIOP	Remote Method Invocation – Internet Inter-Orb Protocol
SDK	Software Development Kit
SSL	Secure Sockets Layer (vrstva bezpečných soketů)

Seznam obrázků

Část 1

Úvod

Není tomu tak dávno, co byl počítač omezen pouze na svůj výpočetní výkon. Dnes je tomu jinak. Máme k dispozici rozsáhlé počítačové sítě, které lze zužitkovat i pro sdílení výpočetního výkonu. Tento výkon můžeme mimo jiné využít i pro rozsáhlé simulační výpočty.

V současnosti tvoří modelování a simulace širokou oblast na poli informačních technologií. Tato oblast nám umožňuje zkoumat chování systémů reálného světa v čase, prostřednictvím jejich abstrakcí, aniž bychom museli tyto systémy fyzicky konstruovat. Významné uplatnění nalezneme v mnoha oblastech lidského působení – kupříkladu v armádě či při analýze a návrhu dopravních přepravních systémů.

Simulace je velmi často nákladná na výpočetní čas. Abychom výsledek získali v přijatelném konečném čase, musíme vynaložit značné prostředky pro získání vysokého výpočetního výkonu, kterého je zapotřebí. Řešením tohoto problému může být distribuovaná simulace.

Distribuovaná simulace spočívá v rozložení simulačního procesu na více oddělených individuálních strojů, které jsou vzájemně propojené v komunikační síti. Jednotlivé propojené stroje nazýváme výpočetními uzly či logickým procesy. Každý uzel se může soustředit na plnění odlišného úkolu. Soudobá neustále se zvyšující rychlost počítačové sítě nám dává možnost umístit výpočetní uzly do různých, zeměpisně odlišných, míst. Komunikace mezi uzly probíhá pomocí zasílání a výměny zpráv, například pomocí celosvětové sítě Internet. Počítačová síť Internet nám nabízí snadné propojení mnoha výpočetních uzlů z celého světa. Tato technika otevírá dveře k získání velkého výpočetního výkonu a zkrácení nezbytně nutné doby pro simulační výpočet. Klíčovým momentem při návrhu distribuovaného systému je správná dekompozice řešeného problému na více výpočetních uzlů.

Nejvýznamnější nevýhodou každého distribuovaného systému je jeho složitý návrh a implementace. Při implementaci distribuovaného systému se klade velký důraz zejména na správně navržený postup, který hlídá synchronizaci času mezi jednotlivými výpočetními uzly. Tato část je velmi důležitá. Narušení kauzality synchronizace času nesmí v simulačním procesu nikdy nastat.

Pro podporu distribuované simulace mnohdy narazíme i na problém samotné konfigurace simulačních modelů. Doposud neexistuje příliš mnoho nástrojů, pomocí kterých bychom mohli jednoduše popisovat distribuované simulační modely.

Distribuované simulace se začaly více používat až koncem devadesátých let. Jedná se tedy o velmi mladou technologii, která má stále velký potenciál v budoucnosti počítačových simulací a měla by se jí věnovat zvýšená pozornost.

1.1 Cíle diplomové práce

Práce je rozčleněna do dvou hlavních částí. V první části se budeme věnovat zejména teorii distribuované simulace, platformě Java a programovacím technikám. Cíle první části jsou následující:

1. Úvod do problematiky distribuované simulace.
2. Seznámení s konzervativními synchronizačními metodami.
3. Představení platformy Java a úvod do konkurenčního programování.
4. Hledání návrhových vzorů, které oceníme při tvorbě vývojového prostředí pro tvorbu distribuovaných simulačních modelů.

V druhé části práce se budeme věnovat praktické části. V praktické části se kladou zejména tři cíle:

1. Najít vhodný způsob pro konfigurování distribuovaných simulační modelů.
2. Implementovat simulační jádro s podporou distribuované simulace.
3. Vytvořit vývojové prostředí, které umožní deklarativní zápis simulační modelů.

Pro každý cíl je vyhrazena jedna kapitola a daná problematika je popsána i z implementačního hlediska. Důležitá je zejména poslední část, ve které je podrobně představeno vývojové prostředí pro konfiguraci distribuovaných simulační modelů, které je hlavním výsledkem této práce.

1.2 Struktura diplomové práce

Dokument je rozdělen do sedmi částí v následujícím pořadí:

1. Úvod

   První kapitola představuje krátké seznámení s tématem této práce, úvod do problematiky, cíle a celkovou strukturou dokumentu pro lepší orientaci.

2. Distribuovaná simulace

   Tato část je rozdělena do tří částí. Podíváme se na obecnou problematiku distribuované simulace a základní pojmy.

3. Konzervativní synchronizační algoritmy

   V této části se zaměříme na synchronizační algoritmy distribuovaného simulačního výpočtu a hlouběji si jeden z nich popíšeme.

4. Nástroje a techniky pro tvorbu distribuovaných aplikací

   Podíváme se na implementační prostředky, které oceníme při tvorbě distribuovaných aplikací. Popíšeme si vybrané návrhové vzory, programovací techniky, konkurenční programování a komunikační technologii Java RMI.

5. Konfigurace distribuovaných modelů

   Vstupujeme do praktické části. Představíme si metodiku deklarativní konfigurace distribuovaných simulační modelů. Blíže se podíváme i na její implementační stránku.

6. Jádro distribuované simulace

   V této části si popíšeme implementaci simulačního jádra s podporou distribuované simulace. Vysvětlíme si použitý algoritmus a také způsob použití jádra knihovny pro spuštění distribuované simulace a konfigurace modelu.

7. Distribuovaná simulace bez použití deklarativního přístupu

   První etapou vývoje bylo otestování myšlenky distribuovaného simulačního jádra bez deklarativního přístupu.

8. Vývojové prostředí jDistsim IDE

   V závěru praktické části bude představeno vývojové prostředí jDistsim, které slouží ke konfiguraci distribuovaných simulačních modelů. Je to hlavní výsledkem této práce. Mimo jiné se podíváme na architekturu aplikace, grafické rozhraní, ovládání a možnosti rozšiřitelnosti.

9. Závěr

   Celkové shrnutí a zhodnocení diplomové práce.

Část 2

Tato kapitola je rozdělena do tří částí. Nejprve se zaměříme na obecnou problematiku distribuovaných simulací. Seznámíme se se základními pojmy a přístupy, které se v distribuované simulaci používají. Vysvětlíme si rozdíl mezi paralelní a distribuovanou simulací. Zaměříme se i na výhody, nevýhody a praktické využití distribuované simulace. V druhé části se zaměříme na synchronizační algoritmy distribuovaného výpočtu. Hlouběji si jeden z nich popíšeme – konkrétně algoritmus, který pro synchronizaci využívá zasílání nulových zpráv. Ve třetí části této kapitoly jsou popsány nástroje a techniky platformy Java, které lze pro vývoj aplikace s podporou distribuované simulace využít.

2 Distribuovaná simulace

2.1 Úvod

Počítačovou simulaci si můžeme představit jako proces, který odráží chování nějakého skutečného reálného nebo imaginárního systému v čase. Se znalostí počítačových simulací můžeme analyzovat systémy, aniž bychom je museli konstruovat. Toho se využívá zejména v dopravních přepravních systémech, neboť reálná konstrukce takových systémů může být velmi nákladná na zdroje (zejména finanční), nebezpečná nebo dokonce neproveditelná. Simulace se také používá při předpovědí počasí, na ekonomické odhady nebo pro generování virtuálního prostředí, na základě kterého může fungovat například letecký trenažer nebo výcvikový simulátor. (Čerpáno z [1])

Oblast působení počítačové simulace je opravdu široká. Průběh simulace je ale často velmi nákladný na výpočetní čas. Abychom výsledek získali v přijatelném konečném čase, musíme vynaložit značné prostředky pro získání vysokého výpočetního výkonu, kterého je zapotřebí. Řešením tohoto problému může být paralelní a distribuovaná simulace, která nám umožňuje spouštět simulaci na více procesorech, které jsou vzájemně propojeny.

2.2 Paralelní a distribuovaná simulace

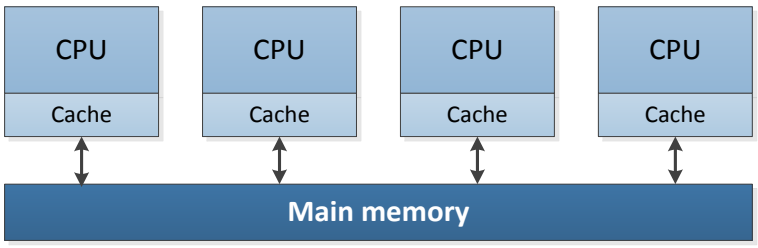
Rozdíl mezi paralelní a distribuovanou simulací spočívá v typu prostředí, ve kterém je simulace spuštěna. [1]

2.2.1 Paralelní simulace

Paralelní simulace běží na paralelních počítačích, které bývají vedle sebe. Nejčastěji se jedná o homogenní stroje. Komunikační linka mezi počítači bývá společná, a proto je komunikace velmi rychlá a bezpečná. Simulace nejčastěji probíhá v laboratorních podmínkách v rámci jednoho výpočetního centra (laboratoře). Rozlišujeme tři typy paralelních počítačů:

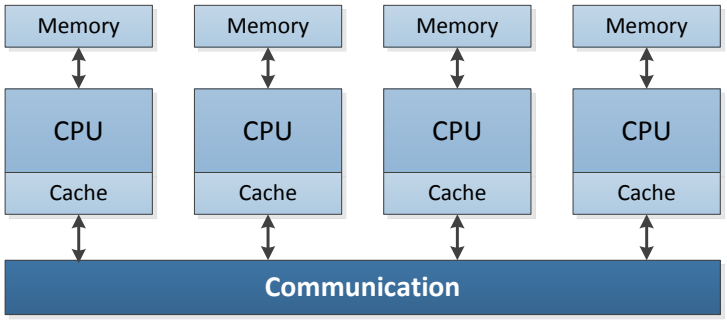
- paralelní počítače se sdílenou pamětí,
- paralelní počítače s distribuovanou pamětí,
- SIMD počítače (single instruction, multiple data set). [2]

Paralelní počítače se sdílenou pamětí jsou mezi sebou velmi úzce spjaty. Každý procesor má k paměti stejný přístup jako kterýkoliv jiný (Obrázek 1).



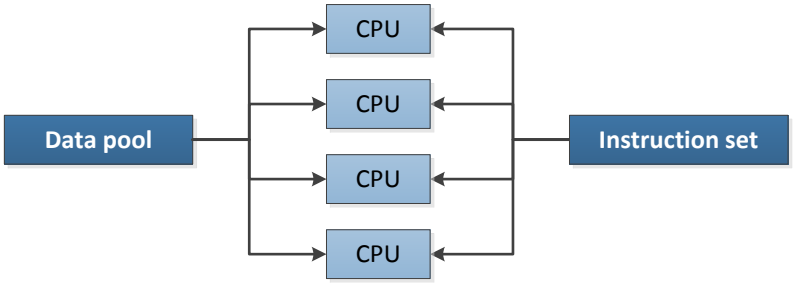
Obrázek 1 – Paralelní počítače se sdílenou pamětí
Zdroj: Vlastní zpracování

V případě paralelních počítačů s distribuovanou pamětí platí pravidlo, že každý procesor má k dispozici vlastní operační paměť, ke které nemají ostatní účastníci přístup, a propojení jednotlivých uzlů je řešeno pomocí speciální sběrnice, která je nejčastěji v topologii kruh, strom, pole nebo kostka (Obrázek 2). Procesory mezi sebou komunikují prostřednictvím zasíláním zpráv. (Zpracováno dle [1], [2])



Obrázek 2 – Paralelní počítače s distribuovanou pamětí
Zdroj: Parallel and Distributed Simulation Systems [1]

Třetím typem jsou paralelní počítače, které jsou architektury SIMD[1], dle Flynn [4] klasifikace (Obrázek 3). Tyto počítače jsou charakteristické tím, že mají jeden seznam instrukcí, který zpracovává několik proudů dat – tzv. vektorové počítače. Nejčastěji najdou uplatnění právě na poli vědeckých a technických výpočtů. (Upraveno podle [5])

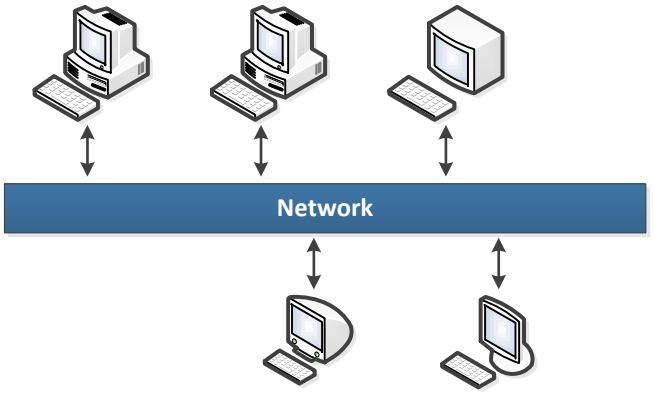


Obrázek 3 – SIMD paralelní počítače
Zdroj: Parallel and Distributed Simulation Systems [1]

[1] Single Instruction, Multiple Data

2.2.2 Distribuovaná simulace

Oproti paralelní simulaci nejsou v distribuované simulaci jednotlivé uzly tak úzce svázaný. Distribuované počítače mohou mít své působení daleko širší (třeba po celém světě). Propojeny jsou pomocí počítačové sítě (Obrázek 4). V dnešní době se nabízí zejména síť Internet, která má globální působení a přijatelnou rychlost. Na druhou stranu vzdálenost a komunikační režie, která se v distribuované simulaci vyskytuje, přináší daleko větší latenci. Toto zpoždění se může pohybovat v řádech stovek mikrosekund, pakliže jsou počítače v jedné budově. Má-li naše simulace geografické působení, kde mezi uzly může být vzdálenost až několik stovek kilometrů, pak latence v tomto případě může dosahovat až násobného nárůstu (řádově vteřiny). (Upraveno dle [1], [5])



Obrázek 4 – Schéma distribuované simulace
Zdroj: Parallel and Distributed Simulation Systems [1]

Prostředí, ve kterém distribuovaná simulace běží, bývá zpravidla heterogenní. Každý účastník distribuovaného systému má vlastní operační paměť, procesor a vstupně-výstupní zařízení. Také může být realizován na různé implementační platformě. Komunikaci sjednocuje komunikační protokol, kterým si účastníci systému vyměňují zprávy. Dostává se nám tak jedinečné možnosti jak spojovat simulační modely, které jsou implementovány na různých platformách. [1]

Síla distribuované simulace je také v její odolnosti. Pakliže vypadne ze sítě jeden výpočetní uzel, pak jeho roli mohou převzít ostatní uzly v síti. Totéž může platit i pro paralelní simulaci. Podstatný přínos je také v rapidním nárůstu potencionálního výkonu, který se nám v distribuované simulaci může dostat. Máme možnost řešit složité simulační modely v konečném, relativně krátkém, čase, oproti klasickému monolitickému přístupu.

Paralelní a distribuované simulace jsou si v mnoha aspektech velice podobné. Významný rozdíl je zejména v komunikaci a vzdálenosti mezi výpočetními uzly. Tento rozdíl je zásadní, neboť komunikační zpoždění způsobuje rozdílné problémy a vyžaduje odlišný přístup při řešení. V této práci se budeme dále věnovat pouze simulaci distribuované.

V případě distribuované simulace je zapotřebí řešit problémy, které v monolitickém přístupu nemohou nastat – řešení konfliktů a obecně synchronizace simulačního času

napříč všemi účastníky distribuovaného systému. Existují dvě obecné metodiky, které se používají při řešení distribuované simulace:

- optimistická metoda,
- konzervativní metoda.

2.3 Základní pojmy

Na úvod je dobré si objasnit některé základní pojmy, které se v distribuované simulaci a této práci vyskytují: (Zpracováno dle [1], [2], [3])

- **Model** – obraz napodobující odraz reálného zkoumaného objektu. Mezi napodobujícím a reálným objektem existuje relace.
- **Modelování** – proces nahrazování zkoumaného objektu s využitím znalosti modelu. Snažíme se získat co nejvíce informací o reálném objektu skrze model.
- **Simulace** – metodika, jak získat co nejvíce informací o zkoumaném systému. Výsledkem by mělo být nahrazení originálu jeho simulátorem, s kterým můžeme provádět experimenty.
- **Diskrétní simulace** – ke změnám v systému dochází pouze v diskrétních časových bodech (jen v okamžiku dokončení události).
- **Logický proces** – pod pojmem logický proces si můžeme představit jeden simulační model, který je součástí distribuovaného simulačního modelu. Logický proces disponuje vlastním simulačním jádrem. Označujeme jej zkratkou LP.
- **Distribuovaný simulační model** – model složený z konečné množiny logických procesů. Do distribuovaného simulačního modelu také patří komunikační kanály, prostřednictvím kterých LP mezi sebou komunikují.
- **Lokální virtuální čas** – každý logický proces má vlastní lokální čas, který označujeme jako lokální virtuální čas. Často se označuje zkratkou LVT[2].
- **Zpráva** – logické procesy mezi sebou komunikují pouze pomocí zasílání zpráv, které bývají zpravidla označeny časovým razítkem.
- **Lookahead** – bezpečný výhled.
- **Lokální kauzalita** – každý proces musí zpracovávat zprávy v neklesajícím pořadí jejich časových razítek.

[2] Local Virtual Time

2.4 Optimistická metoda

Jak již název metodiky napovídá, pracujeme zde s optimistickým výhledem. Předpokládejme systém, ve kterém mezi sebou komunikuje několik logických procesů. Tyto logické procesy si spolu vyměňují zprávy, které jsou označeny časovým razítkem. Zároveň má každý logický proces vlastní virtuální čas a neexistuje žádný globální pro celou distribuovanou simulaci. Každý logický proces pracuje optimisticky a neřeší žádnou synchronizaci. Z tohoto důvodu hrozí narušení lokální kauzality – může přijít zpráva s časovým razítkem nižším, než jaký byl čas poslední zpracované zprávy. [3]

V případě optimistické metody žádnou synchronizaci neřešíme do okamžiku, kdy dojde ke konfliktu narušení lokální kauzality. V okamžiku, kdy tento problém nastane, jej musíme identifikovat a napravit. Optimistická metoda musí obsahovat mechanismy, které se s tímto stavem dokáží správně poradit. (Upraveno oidle [3])

Častým řešením bývá technika, kdy se v případě problému provede tzv. rollback do bezpečného stavu. Tento proces může být časově velmi náročný, neboť musíme měnit veškerý stav systému, který jsme doposud měnili. Rollback se týká i dalších logických procesů, se kterými jsme byli v interakci od bezpečného stavu. Dalším mechanismem může být průběžné ukládání jednotlivých stavů celého systému. Toto řešení ale přináší značné paměťové nároky. (Zpracováno dle [3])

Existuje celá řada dalších algoritmů, které lze aplikovat v případě optimistických metod pro zaručení lokální kauzality. Optimistický přístup by se měl aplikovat vždy tam, kde mezi logickými procesy zřídka kdy dochází ke komunikaci. Opravné mechanismy, které nastávají v okamžiku přijmutí zatoulané zprávy, tuto metodu vždy velmi zpomalují.

2.5 Konzervativní synchronizační metoda

V případě konzervativní synchronizační metody je maximální snaha o identifikaci problému narušení kauzality a předcházet mu. Každou příchozí zprávu proto zpracováváme velmi opatrně. Musíme si být naprosto jistí, že zpracováním zprávy nemůže dojít ke konfliktu. Konflikt nastane v okamžiku, kdy byla zpracována zpráva s časovým razítkem nižším, než které měla poslední zpracovaná.

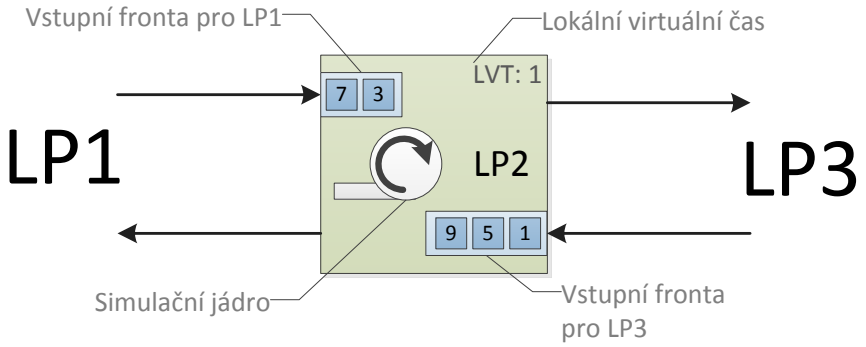
K předcházení konfliktů se využívají konzervativní synchronizační algoritmy, které striktně zaručují správnou posloupnost zpracování zpráv v neklesajícím pořadí časových razítek. [1]

Konzervativní přístup oceníme zejména v případech, kde mezi sebou logické procesy vyměňují velké množství zpráv. V této práci se budeme dále věnovat pouze této metodě a jejím synchronizačním algoritmům. [1]

3 Konzervativní synchronizační algoritmy

3.1 Konzervativní synchronizační metoda

Předpokládejme distribuovanou simulaci složenou z několika logických procesů. Každý logický proces pracuje zcela autonomně nad vlastním simulačním jádrem. Jádro je postavené nad konceptem metody plánování událostí z diskrétní simulace. To znamená, že obsahuje vlastní kalendář událostí (prioritní frontu), ve které jsou doposud nezpracované naplánované lokální události a vlastní lokální simulační čas (Obrázek 5). Nutno podotknout, že do těchto komponent nemohou ostatní účastníci nijak zasahovat. Dále logický proces eviduje speciální vstupní frontu pro každý sousední logický proces, se kterým je v přímém spojení. (Zpracováno dle [1], [4])



Obrázek 5 – Schéma logického procesu
Zdroj: Vlastní zpracování

Na obrázku výše je zachyceno schéma logického procesu LP2, který je v přímé interakci s LP1 a LP3. Všichni si mezi sebou vyměňují zprávy. Každá zpráva je opatřena časovým razítkem. Tyto zprávy se následně ukládají do front, kde jsou řazeny dle svých časových razítek. V těchto frontách čekají na zpracování. ([1], [2])

U zpráv, které nám přicházejí ze sousedních logických procesů, předpokládáme, že hodnota časových razítek bude v neklesajícím pořadí. To znamená, že poslední obdržená zpráva v daném spojení má nejnižší možné časové razítko ze všech přijatých zpráv, které mohou potencionálně dorazit v budoucnu. Logický proces pak může zaručit dodržení lokální časové kauzality, pakliže bude zpracovávat zprávy v neklesajícím pořadí dle časových razítek ze všech vstupních front a to pouze za předpokladu, že v každé vstupní frontě je alespoň jedna zpráva. Není-li v nějaké vstupní frontě žádná zpráva, pak logický proces musí svou činnost dočasně pozastavit a vyčkat na příchod zprávy. Toto tvrzení zachycuje následující obecný pseudokód algoritmu distribuované simulace:

```
while(Dokud není konec simulace) {
      while(Čekej, dokud není v každé vstupní frontě alespoň jedna zpráva);

      Zprava zpráva = připravZprávu();
      aktualizujLokálníČas(zprava);
      zpracuj(zpráva);
}
```

Metoda připravZprávu() najde zprávu s nejnižším časovým razítkem. Hledání probíhá ve všech vstupních frontách a také v lokálním kalendáři. Po nalezení zprávy se aktualizuje lokální simulační čas a zpráva se zpracuje. Po uběhnutí jedné iterace je nutné opět zkontrolovat, zda jsou všechny vstupní fronty neprázdné.

V jeden okamžik může mít každý logický proces, který je součástí systému, jiný lokální simulační čas – neexistuje žádný centralizovaný prvek, dle kterého by se prováděla synchronizace jednotlivých lokálních simulačních časů na stejnou hodnotu.

3.2 Problém synchronizace

Uvažme, že máme navrhnutý simulační model, který je složen z několika logických procesů. Spuštění takového modelu na sekvenčním počítači nám zaručí, že všechny zprávy budou zpracovávány v neklesajícím pořadí časových razítek, neboť výpočet pracuje sekvenčně. Jak tomu ale bude v případě, kdy se stane simulace distribuovanou?

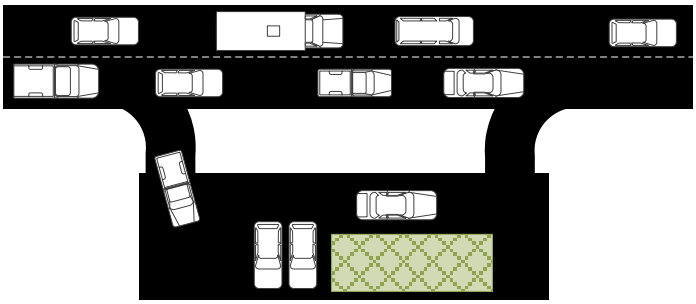
Vezměme stejný simulační model a spusťme ho v distribuovaném prostředí, kde každý logický proces může běžet na jiném počítači. Tyto počítače mohou mít odlišný výkon a mohou být od sebe různě vzdáleny. Bude výsledek distribuované simulace stejný, jako v případě sekvenčního spuštění?

Ano, bude. Potažmo musí být stejný. Pakliže by byl výsledek rozdílný, pak distribuovaná simulace nepracuje správně. O správnost se v distribuované simulaci stará vhodně zvolený synchronizační algoritmus. Je zapotřebí si uvědomit, že průběh zpracování zpráv může být odlišný, oproti sekvenčnímu přístupu. Celkový výsledek ale musí být vždy stejný. [1]

Dalším úkolem synchronizačního algoritmu je vyřešení stavu uváznutí, do kterého se distribuovaná simulace může dostat. Popřípadě detekce tohoto stavu.

3.3 Řešení stavu uváznutí pomocí nulových zpráv

Pojďme si přiblížit problém uváznutí simulace na reálném příkladu. Uvažme dopravní systém, který odráží reálnou dopravní situaci na dálnici a přilehlém rychlém občerstvení. Pro lepší interpretaci algoritmu si vymezíme model pouze na jeden silniční pruh. Modelovaný příklad ilustruje následující obrázek (Obrázek 6):



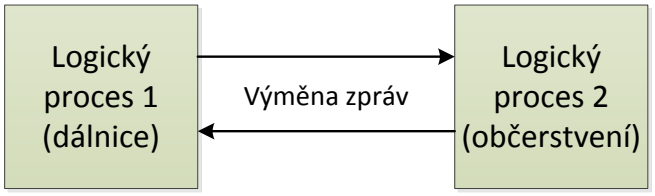
Obrázek 6 – Ilustrační obrázek dopravní situace
Zdroj: Vlastní zpracování

Předmětem zkoumání budou příjezdy, resp. odjezdy automobilů na této komunikaci a také nás zajímají automobily, které z dálnice odbočily do rychlého občerstvení. Každý automobil, který zamířil do rychlého občerstvení, se po dokončení obsluhy opět zařadí do provozu na dálnici a pokračuje dál v jízdě do svého odjezdu.

Ilustrační motivační obrázek (Obrázek 6) charakterizuje dopravní situaci, kterou se simulační model snaží zachytit. Z obrázku je zřejmé, že model lze rozdělit na dvě části:

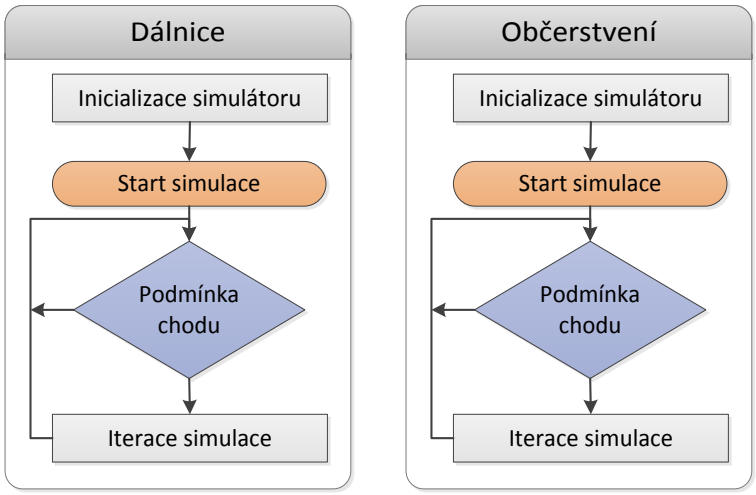
- dálnici,
- občerstvení.

Zmíněné dva stavební bloky modelu si lze představit jako samostatně fungující logické procesy – dálnici a občerstvení (Obrázek 7). Představa těchto dvou logických procesů nás přivádí k myšlence model škálovat a využít zde výhod distribuované simulace. Každý logický proces může být spuštěn na samostatném výpočetním stroji. Nemusí se vzájemně dělit o výpočetní výkon, ale každý bude mít svůj vlastní.



Obrázek 7 – Logické procesy
Zdroj: Vlastní zpracování

Při spuštění celého experimentu jsou pak tyto logické procesy vzájemně provázány a jsou mezi nimi vytvořeny vzájemné závislosti, které dohromady utváří uvažovaný problém.



Obrázek 8 – Životní cyklus algoritmu logických procesů
Zdroj: Vlastní zpracování

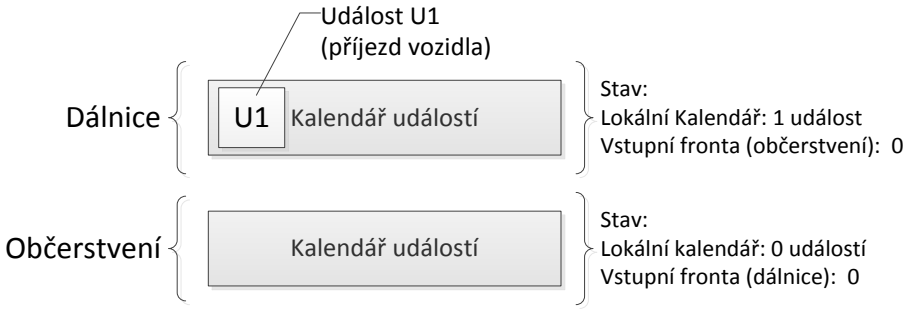
Životní cyklus simulace obou logických procesů je totožný (Obrázek 8). Proběhne inicializace simulátoru a následně samotný simulační algoritmus. Připomeňme si podmínku chodu simulace – v každé vstupní frontě musí být alespoň jedna zpráva a lokální

kalendář je neprázdný (kapitola 3.1). V navrhovaném modelu může nastat situace, která bude mít za následek stav uváznutí. Nejedná se o situaci nahodilou, nýbrž relativně častou. Přišli jste na ni?

Řekněme, že při inicializaci logického procesu dálnice je naplánovaná jedna událost (příjezd vozidla). Úkolem dálnice je přeposílat některá vozidla do rychlého občerstvení a očekávat příjezdy vozidel z rychlého občerstvení nazpět. Druhý logický proces, občerstvení, ve své inicializaci nic nepodnikne a pouze čeká na případné příjezdy vozidel z dálnice. Po zpracování zákazníka je automobil opět vyslán nazpět na dálnici. Oba procesy nyní spustíme.

Podíváme-li se na výše uvedený příklad pozorněji, pak zjistíme, že nastalo uváznutí a simulace se nikdy nedokončí v konečném čase, potažmo ani nezačne. Rozeberme si, co přesně provádějí jednotlivé procesy:

- dálnice (LP1):
    1. inicializace – naplánuje jednu událost (příjezd vozidla),
    2. čeká na platnost podmínky (neprázdný lokální kalendář a neprázdná vstupní fronta od LP2),
- občerstvení (LP2):
    1. inicializace – nic se neděje,
    2. čeká na platnost podmínky (neprázdný lokální kalendář a neprázdná vstupní fronta od LP1).



Obrázek 9 – Stav uváznutí simulace
Zdroj: Vlastní zpracování

Ani jeden proces nezaslal doposud žádnou zprávu a oba na sebe vzájemně stále čekají (Obrázek 9). Dálnice má připravené události ke zpracování, ale má prázdnou vstupní frontu. Musí se zablokovat a čekat, neboť existuje možnost, že by mohla přijít zpráva z rychlého občerstvení. Logický proces rychlého občerstvení nemá žádné lokální události ke zpracování a má prázdnou vstupní frontu, takže také čeká. Protože ani jeden logický proces nemůže bezpečně zpracovat žádnou událost, simulace je zablokována – i když se vyskytují dosud nezpracované události. Nastalo smrtelné objetí (stav uváznutí).

Uváznutí může být řešeno následujícím způsobem: nechť minimální doba pro příjezd vozidla do rychlého občerstvení je 7 časových jednotek. To znamená, že minimální hodnota časového razítka zprávy pro rychlé občerstvení je vždy 7 jednotek simulačního času. V tomto časovém intervalu je zaručeno, že dálnice může bezpečně simulovat, aniž by byla porušena lokální kauzalita. Bylo by tedy dobré, aby logický proces dálnice tuto informaci znal. Jak mu ji ale předat?

3.3.1 Zasílání nulových zpráv - algoritmus C/M/B

Je potřeba najít vhodný mechanismus, který bude schopný ostatní logické procesy informovat o nejnižších časových razítkách, které v budoucnu může odeslat. Pro tento účel lze použít tzv. nulové zprávy[3]. Tyto zprávy slouží výhradně pro synchronizační účely a nemají žádný jiný význam. Algoritmus se nazývá podle svých tvůrců Chandy / Misra / Bryant a někdy je zkráceně označován C/M/B. (Upraveno podle [1], [2])

Odeslaná nulová zpráva z LP1, která je označena časovým razítkem Tnull na LP2 lze také formulovat tak, že LP1 se zaručuje, že v budoucnu nezašle žádnou zprávu s hodnotou časového razítka menším než Tnull. (Upraveno podle [1], [2])

Nulové zprávy se odesílají při každé změně lokálního simulačního času. Nemůže tedy nikdy nastat situace, kdy je prázdná vstupní fronta. Předcházíme stavu uváznutí, který by jinak z důvodu vzniku cyklu prázdných front mohl nastat. Pseudokód simulačního algoritmu s použitím nulový zpráv vypadá následovně: [1], [5]

```
while(Dokud není konec simulace) {
      while(Čekej, dokud není v každé vstupní frontě alespoň jedna zpráva);

      Zpráva zpráva = připravZprávu();
      aktualizujLokálníČas(zpráva);
      zpracuj(zpráva);
      odešliVšemNulovéZprávy();
}
```

Po zpracování každé zprávy odešle všem logickým procesům, se kterým je daný logický proces ve spojení, nulovou zprávu. Ostatní mohou na základě této zprávy přehodnotit svůj čas a případně dál informovat své nejbližší sousedy. (Čerpáno z [1], [2])

3.3.2 Výhled

Jak správně určit hodnotu časového razítka nulových zpráv? Pro tento účel je zapotřebí získat hodnotu lookahead. Lookahead je hodnota bezpečného výhledu. Pakliže má logický proces lokální virtuální čas T a hodnota výhledu je rovna L, pak může naplánovat pouze takové události, které mají časové razítko vetší nebo rovno hodnotě T + L. Správné určení hodnoty výhledu je velmi důležité, neboť zvyšuje hodnotu paralelizace logických procesů.

Určit hodnotu výhledu lze několika způsoby. Lookahead lze získat ze znalosti simulačního modelu. Kupříkladu v případě dálnice a občerstvení známe dolní hranice intervalů mezi příjezdy automobilů a také známe nejkratší možnou dobu obsluhy v občerstvení. Tyto

[3] Z anglického „null messages“

fakta lze aplikovat pro výpočet hodnoty výhledu. Můžeme hodnotu výhledu také získat předpočítáním simulačních aktivit do budoucna. Do výhledu lze zapojit i toleranci dočasné nepřesnosti, která je spojena s doručováním zpráv. Máme možnost aplikovat i Epsilon výhled – minimální nenulová hodnota změny simulačního času. (Čerpáno z [1])

Správná hodnota výhledu je nesmírně důležitá. Je zapotřebí při jejím získávání být velmi obezřetný a opatrný. Bude-li hodnota výhledu velmi malá, pak je zřejmé, že se zvýší počet nulových zpráv, které si logické procesy zasílají. Může nastat i situace, kdy zpracujeme několik nulových zpráv, než se dostaneme ke standardní události. Může také nastat situace, kdy bude výhled nulový. V tomto případě můžeme opět dospět do stavu uváznutí. V okamžiku, kdyby vznikla nekonečná sekvence nulových zpráv s nulovým výhledem, algoritmus selže. Proto nesmí v distribuovaném simulačním modelu existovat cyklus logických procesů s nulovým výhledem. Hodnotu výhledu lze dynamicky měnit i v průběhu samotné simulace. (Zpracováno dle [1], [2])

3.3.2.1 Zasílání nulových zpráv na vyžádání

Můžeme zanést malou modifikaci, která optimalizuje množství zasílaných nulových zpráv. Budeme o nulové zprávy vždy žádat a to pouze tehdy, když detekujeme stav uváznutí. Principiálně to funguje tak, že LP v případě detekce prázdné vstupní fronty automaticky zašle požadavek, na daný logický proces, s žádostí o nulovou zprávu a čeká na odpověď. Tento algoritmus nazýváme „Zasílání nulových zpráv na vyžádání[4]“. [1]

Zavedením tohoto algoritmu znatelně klesne počet zasílaných nulových zpráv, které v systému proudí. Musíme si ale uvědomit, že pro získání jedné nulové zprávy je zapotřebí dvou zpráv – požadavek a odpověď. Názorná evoluce simulačního procesu, kde je použit právě tento algoritmus, je zachycena v příloze (Příloha A). (Čerpáno z [1], [2])

3.4 Detekce uváznutí a zotavení

Alternativou, pro algoritmus využívající nulové zprávy, může být metoda, kdy se budeme snažit uváznutí detekovat a následně napravit. Tato metoda vychází ze základního synchronizačního algoritmu, který navíc obohacuje o schopnost detekovat uváznutí. Lze říci, že algoritmus připouští vznik uváznutí – pokud ale uváznutí skutečně nastane, pak je neprodleně proveden proces zotavení. (Čerpáno z [1], [2])

Algoritmus zavádí řídící proces[5], aby byl schopný odhalit stav uváznutí. Tento proces musí být schopný komunikovat se všemi ostatními logickými procesy, neboť se snaží řídit jejich činnost. (Upraveno podle [1], [2])

Každý logický proces se může nacházet právě ve dvou stavech:

- aktivní,
- neaktivní.

[4] Demand driven null messages algorithm
[5] Controller process

Na základě těchto dvou stavů procesy rozlišujeme. Algoritmus z logických procesů vytváří jakýsi pomyslný strom. V tomto stromě se udržují všechny aktivní procesy. Pohledem na tento domnělý strom je i možné vidět, jak se šířil výpočet napříč celou distribuovanou simulací. (Čerpáno z [1], [2])

Každý logický proces musí dodržet následující pravidla:

- Pakliže obdrží zprávu takový proces, který se nachází ve stromě, pak musí tuto informaci neprodleně předat odesílateli – nedošlo k šíření výpočtu.
- Pakliže obdrží zprávu takový proces, který se nenachází ve stromě, pak se stane listem tohoto stromu. Není potřeba tuto informaci dále předávat. [2]

3.4.1 Algoritmus

Na počátku simulace považujeme celou simulaci za uváznutou. Dále následuje sled následujících kroků:

1. Řídící logický proces informuje ostatní logické procesy o bezpečných událostech. Informaci poskytuje formou zaslání zprávy.
2. Logické procesy začnou zpracovávat bezpečné zprávy. Pravděpodobně nastane situace, kdy tyto bezpečné události inklinují k zasílání dalších zpráv na další logické procesy, které je mohou začít zpracovávat. Tomuto jevu říkáme „šíření simulačního výpočtu.

   Šíření simulačního výpočtu představuje pomyslný strom, který byl zmíněn výše. Všechny aktivní procesy se nacházejí ve stromě. Pakliže jeden logický proces zasílá zprávu blokovanému procesu, pak se tento proces stane aktivním a také součástí stromu – jako potomek odesílatele zprávy.
3. Pakliže logický proces dospěl do fáze, kdy nelze zpracovat bezpečně žádné zprávy, pak se stává blokovaným.

   Nemá-li takový proces žádné potomky, pak je ze stromu vyřazen. O této změně je potřeba informovat nejbližší okolí – předka.
4. Nacházíme-li se ve fázi, kdy se řídící proces stal listem stromu, pak se simulace nachází ve stavu uváznutí. V takovém případě přecházíme na krok 1. (Zpracováno podle [1], [2])

V kroku jedna dochází v podstatě k zotavování simulačního výpočtu. Řídící proces musí vhodně určit zprávy, které lze považovat za bezpečné. Jak ale takové zprávy najít?

Řídící proces má přehled o všech ostatních účastnících. Tato skutečnost nám poskytuje nové možnosti. Jsme schopni zjistit, která aktuálně naplánovaná událost v celé distribuované simulaci má nejnižší hodnotu časového razítka. Událost s nejnižší hodnotou časového razítka v celé simulaci lze považovat za bezpečnou – jedná se o událost, která by

byla provedena jako další v sekvenční simulaci. Naleznutí této zprávy je snadné, neboť se simulace nachází ve stavu uváznutí a negenerují se žádné nové události. Řídící proces proto může bezpečně zasílat požadavky o zjištění události s nejnižším časovým razítkem svým sousedům. Ti pak tento požadavek propagují dál, dokud se neprojdou všechny logické procesy, které jsou do distribuované simulace zapojeny. (Čerpáno z [1], [2])

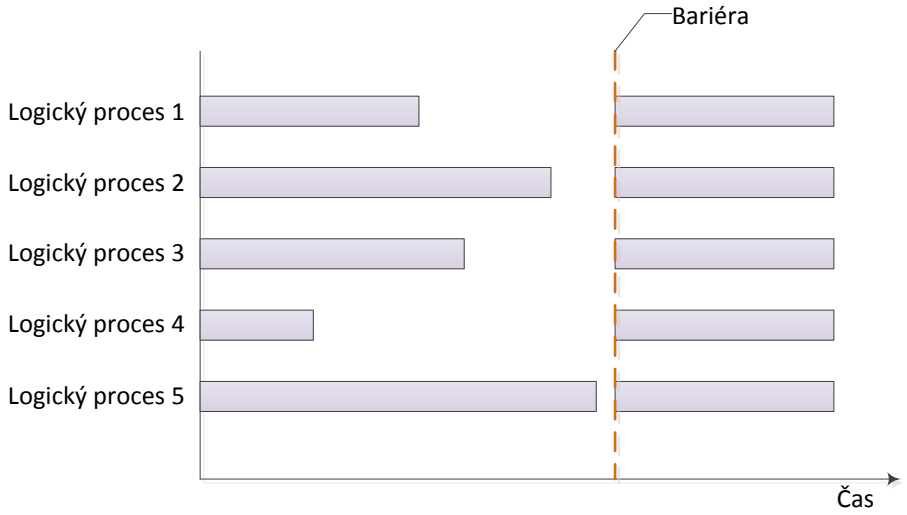
Pakliže bezpečné události nalezneme, pak můžeme prolomit stav uváznutí a pokračovat dále v simulačním výpočtu.

V praxi zjistíme, že se pravidelně střídají fáze simulačního výpočtu – uváznutí, zotavení, uváznutí a tak dále. Tyto cykly se neustále opakují do skončení simulačního výpočtu. I tento algoritmus lze rozšířit o zasílání hodnoty lookahead. Znalost výhledu se zavádí zejména z důvodu snazšího nalezení více bezpečných zpráv a to nám dopomůže ke zvýšení paralelismu. Tento algoritmus dovoluje hodnotu nulového výhledu, oproti mechanismu zasílání nulových zpráv. Je důležité si také uvědomit, že zda pracujeme i s časem nejbližší naplánované neprovedené události, můžeme se rovnou přesunout na tento čas, aniž bychom využívali hodnoty lookahead (nemusíme skákat po hodnotách výhledu). [1]

3.5 Bariérová synchronizace

Dalším typem synchronizačního algoritmu může být mechanismus využívající bariery. Můžeme se vydat cestou, kdy nebudeme čekat, až nastane čas uváznutí, ale definujeme si nějaký bod v čase, ve kterém celou synchronizaci pozastavíme. Jakmile všechny logické procesy dosáhnou tohoto bodu, pak simulaci opět odblokujeme a pokračujeme v simulačním výpočtu. Tento bod v čase se nazývá „bariéra“.

Pakliže logický proces dosáhne bodu bariéry, pak musí neprodleně pozastavit svou činnost a čekat do doby, dokud nedosáhnou tohoto bodu i ostatní. V okamžiku, kdy všichni dosáhnout tohoto bodu, můžeme pokračovat (Obrázek 10). (Zpracováno dle [1], [2])



Obrázek 10 – Bariéra
Zdroj: Vlastní zpracování

Jakmile dosáhnou všechny procesy bariéry, pak je nutné definovat novou bariéru. Musíme také správně určit, které události jsou bezpečné. Zpravidla je bezpečná zpráva ta, která má nejmenší hodnotu časového razítka. Samozřejmě i zde lze aplikovat mechanismus využívající výhled a určit bezpečné zprávy na základě výhledu. (Zpracováno dle [1], [2])

Různé algoritmy využívající bariéry, se odlišují zejména ve způsobu implementace bariéry. Mezi nejpoužívanější patří zejména:

- stromová bariéra,
- motýlková (butterfly) bariéra,
- centralizovaná bariéra. [1]

Z implementačního hlediska je nutné zabezpečit, aby v okamžiku, kdy dosáhne daný proces bariéry, skutečně logický proces nedělal žádnou činnost. Nevytvářel ani nezpracovával žádnou událost. Může být zrovna aktivován proces hledání bezpečných zpráv a ten musí vždy pracovat nad konzistentním (pozastaveným) simulačním výpočtem. Oproti předešlým algoritmům má bariérová synchronizace dopředu definovaný bod v čase, ve kterém se simulační výpočet nachází ve stavu, který známe.

3.6 Synchronní simulační protokol

Uvažme distribuovanou simulaci složenou z několika logických procesů. Každý logický proces musí definovat svou hodnotu výhledu (lookahead). U tohoto algoritmu potřebujeme mechanismus, pomocí kterého budeme schopni získat hodnotu Tmin, kterou vypočítáme ze znalosti hodnoty výhledu a času následující události u všech logických procesů. Pro n logických procesů (LPi) získáme hodnotu Tmin následovně. (Čerpáno z [1])

$$T_{min} = \min\{T_i + L_i\}\, pro\ i \in \{1, .., n\}$$

Kde Ti značí čas události s nejmenším časovým razítkem, a Li hodnotu výhledu pro daný LPi. Výsledná hodnota Tmin nám říká, že všechny události, které mají hodnotu svého časového razítka menší nebo rovno této hodnotě, jsou bezpečné. U tohoto algoritmu je tedy velmi důležité najít vhodnou cestu, jak získat toto globální minimum. Pseudokód simulačního algoritmu pak může vypadat následovně: (Upraveno podle [1], [2])

```
while(dokud není konec simulace) {
	Tmin = najdiMinimum(); // (Ti + Li) pro všechna LPi

	List<Zpráva> zprávy = najdiZprávy().where(zpráva => zpráva.Čas <= Tmin);
	zpracuj(zprávy);
	bariérováSynchronizace();
}
```

3.7 Shrnutí

Synchronizace simulačního času napříč všemi logickými procesy není jednoduchá. Je nutné myslet na spousty aspektů, které nám vstupují do cesty. Dle toho pak zvolit vhodný synchronizační algoritmus, který bude pro simulační model vhodný. Naštěstí nemusíme vymýšlet kolo, neboť již existuje mnoho algoritmů, které problém synchronizace simulačního času řeší za nás. Některé vybrané jsme si v této kapitole společně objasnili.

Je důležité si také uvědomit, že některé mechanismy mohou být náročné na implementaci. Komunikace mezi procesy může být velmi častá a proto i náročná. Musíme si také uvědomit, že doručení zprávy z jednoho procesu na druhý může nějaký ten čas trvat. Může se také stát, že zpráva nebude vůbec doručena z důvodu síťové chyby apod. I na tyto záležitosti je nutné v době implementace myslet a ujistit se, že vše bude bezpečné.

4 Nástroje a techniky pro tvorbu distribuovaných aplikací

V této části se podíváme na implementační nástroje a techniky, které jsou vhodné pro tvorbu distribuovaných aplikací. Podíváme se na možnosti, které nabízí platforma Java a zejména na komunikační technologii Java RMI. Dále se zaměříme na návrhové vzory a praktiky, které je vhodné při budování distribuované aplikace a vývojového prostředí pro tvorbu distribuovaných simulačních modelů využít.

4.1 Platforma Java

Soudobé vývojové platformy nám dávají prostor tvořit distribuované aplikace, aniž bychom se museli zabývat některými problémy a kritickými situacemi, které nastávají. Může to být synchronizace objektů, distribuce dat nebo problémy při konkurenčním programování. Java platforma za nás tyto problémy do jisté míry řeší pomocí speciálních programovacích technik, užití určitých klíčových slov, návrhových vzorů nebo speciálních balíčků ze standardního Java SDK. Tyto praktiky a nástroje přijdou velmi vhod pro tvorbu distribuovaných aplikací. (Čerpáno z [6])

Platforma Java je v mnoha aspektech stále velmi pokroková, a proto se budeme věnovat v této práci právě této platformě. Platforma Java je využita i při implementaci praktické části.

4.2 Vícevláknové programování v Javě

Jedním ze základních kamenů každé GUI aplikace je více vláknové programování. Žijeme v době souběžnosti a platforma Java má velmi dobře postaven model pro více vláknové programování. Podpora vláken je v Javě již od rané verze JDK 1.0. Klíčový zlom ale nastal až ve verzi 1.5, která přináší integraci balíčku `util.concurrent`, za kterým stál profesor Doug Lea. Balíček byl na svou dobu velmi pokrokový a mimo jiné balíček zavádí například synchronizované kolekce nebo několik primitiv pro vytváření souběžných aktivit. (Upraveno podle [7])

Standardní vlákna v Javě lze vytvářet dvěma způsoby. První z nich je vytvoření zcela nové třídy, která bude odvozená od bázové třídy `Thread`, kterou nalezneme v balíčku `java.lang`. Třída `Thread` má metodu `run()`, kterou můžeme překrýt a umístit do ní logiku našeho vlákna. Tato metoda je volána při spuštění vlákna. Životní cyklus vlákna končí v okamžiku skončení právě této metody nebo explicitního ukončení vlákna metodou `interrupted()`. Jakmile vytvoříme k takovéto třídě instanci, můžeme s ní pracovat jako s vláknem, například zavolat metodu `start()` a vlákno spustit. (Čerpáno z [8])

```
SimulatorRunner simulatorRunner = new SimulatorRunner(simulator);
simulatorRunner.start();
```

Ne vždy si ale můžeme dovolit dědičnost. Proto Java nabízí ještě druhý způsob, u kterého není potřeba žádné dědičnosti. Vlákna vytváříme skrze rozhraní `Runnable`. Jak to funguje? Rozhraní `Runnable` deklaruje metodu `run()`, kterou musíme implementovat. V těle této

metody bude obsažena logika našeho vlákna. Posledním krokem je třídu spustit. Pro spuštění se opět využívá třída `Thread`, které předáme instanci naší třídy skrze konstruktor.

```
Runnable simulatorRunner = new SimulatorRunner(simulator);
Thread thread = new Thread(simulatorRunner);
thread.start();
```

V ukázce je zachycena stejná situace jako v předešlém příkladu. Jen s tím rozdílem, že nyní třída `SimulatorRunner` nedědí ze třídy `Thread`, ale implementuje rozhraní `Runnable`.

Java nás při práci s vlákny v podstatě nijak nesvazuje. Na základě definování priority můžeme řídit přístup vláken k procesoru, dále máme možnost vlákna různé uspávat, notifikovat, apod. [8]

Celý runtime Javy je sám o sobě více vláknový. Vykreslování GUI aplikací či automatický úklid paměti v podobě *Garbage collectoru* jsou typickými vlákny, které soustavně běží v JVM. Použití vláken je v dnešní době nezbytné při tvorbě desktop aplikací. [7]

4.3 Synchronizace

Při práci s vlákny dřív nebo později narazíme na problém synchronizace. Souběžně prováděná vlákna zpravidla přistupují k nějakým sdíleným prostředkům, ke kterým je potřeba zaručit exklusivní přístup. Tento krok provádíme z důvodu, aby nám v aplikaci nevznikaly nekonzistence nebo nekonečná čekání. Nebezpečné části kódu, kde manipulujeme se sdíleným prostředkem, se nazývají kritické sekce. Pro řešení kritických sekcí má platforma Java připravené klíčové slovo `synchronized`. (Čerpáno z [10])

Můžeme synchronizovat celé metody nebo jen určité bloky kódů. Zabezpečení kódu pomocí synchronizačních bloků je v Javě velmi intuitivní a snadné.

```
public class SimulatorEnvironment {
    private double localTime;
    private final Object lock = new Object();

    // použití synchronizovaného bloku
    public void setLocalTime(double time) {
        synchronized (lock) {
            this.setTime(localTime);
        }
    }

    // použití synchronizované metody
    public synchronized void updateState() {
        // do something
        // ...
    }
}
```

Synchronizace v Javě využívá pro řešení kritických sekcí monitoru. Každému objektu v Javě je přidělen unikátní monitor. Vlákno získává monitor po vstupu do kritické sekce a vzdává se ho až po výstupu z kritické sekce. Označíme-li metodu klíčovým slovem

`synchronized`, pak máme zaručen exkluzivní přístup k této metodě a kód metody nebude nikdy vykonávat více jak jedno vlákno. Měli bychom se ale snažit tvořit metody co nejmenší, aby ostatní vlákna nebyla zbytečně brzděná. V případě synchronizovaných bloků je zapotřebí specifikovat objekt, který nám poskytne zámek (monitor) pro vyřešení přístupu ke kritické sekci. (Čerpáno z [7], [8], [10])

4.4 Verzování aplikace

Rozhodneme-li se vytvořit takovou aplikaci, která je pro nás v mnoha aspektech vstupem do slepých vod, pak je vhodné co nejdříve přejít na nějaký verzovací systém. V případě vytváření verzí se můžeme kdykoliv vrátit do bodu, kdy byla naše aplikace v pořádku. Verzování přináší nesčetný počet výhod. Máme k dispozici kompletní historii změn v našem softwaru a také to přináší možnost práce v týmu. Mezi nejznámější verzovací systémy patří CVS, Subversion, Git, Mercurial a další. Kde Git a Mercurial spadají do kategorie distribuovaných systémů správy verzí, pro které platí, že každý vývojář má u sebe kompletní přehled změn. Své změny každý vývojář zpravidla synchronizuje s nějakým hlavním repozitářem.

Pro vývoj praktické části jsem byl nucen taktéž použít vhodný verzovací systém. Často jsem se při vývoji vydal do slepé uličky a potřeboval jsem se mnohdy vrátit. Dalším aspektem výběru bylo také zálohování. Zvolil jsem verzovací systém Git, který je v současnosti velmi populární. Nad tímto systémem vzniklo například jádro systému Linuxu nebo velmi populární webový Framework Ruby on Rails. Kompletní repozitář praktické části je vystaven na serveru GitHub.com[6], kde je k náhledu i celková historie vývoje aplikace a lze nahlédnout i na zdrojové kódy (více v kapitole věnované praktické části). (Čerpáno z [9])

4.5 Návrhové vzory

Nedílnou součástí každé aplikace je beze sporu její návrh. Před implementací si musíme dobře promyslet, jak aplikaci budeme tvořit, abychom v případě nových požadavků a změn mohli rychle reagovat. Existuje nemálo návrhových vzorů, které můžeme použít a díky tomu nemusíme řešit problém, který před námi vyřešil již někdo jiný a správně. Některé vybrané vzory, které pro mne byly stěžejní, jsem do této kapitoly zanesl.

4.5.1 Observer

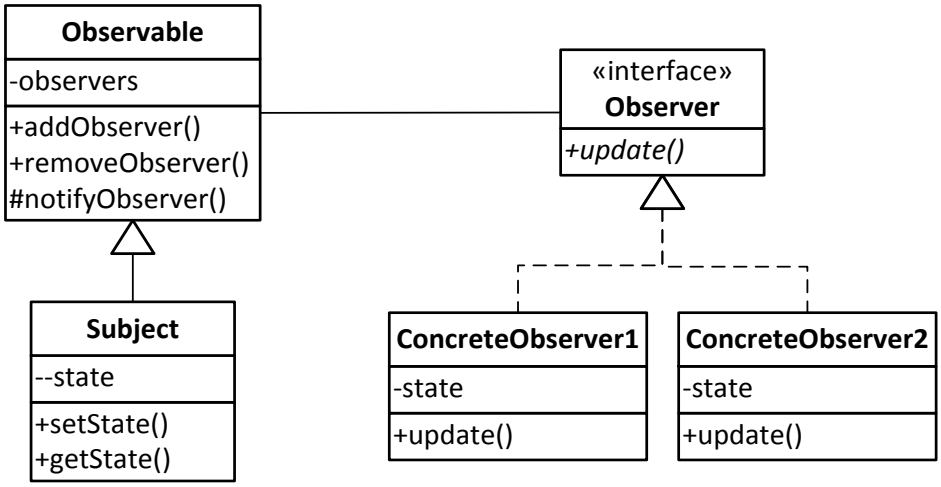
Mnohdy potřebujeme sledovat stav objektu a patřičně na jeho změny reagovat. Vzor `Observer` toto řeší. Definuje vztah mezi objekty tak, že když sledovaný objekt změní svůj stav – pak je tato informace bezprostředně oznámena všem, kteří tento objekt sledují. Zpravidla určíme jeden sledovaný objekt, který lze sledovat více objekty.

Grafické aplikace jsou velmi často vhodným adeptem na použití vzoru `Observer`. V GUI aplikacích se snažíme maximálně oddělit aplikační data od prezenční vrstvy – v okamžiku, kdy dojde ke změně zdrojových dat, se musí překreslit GUI, apod. Sledovaný objekt

[6] https://github.com/jirkapenzes/jDistsim

většinou ani neví, kým je pozorován. To je silná výhoda tohoto vzoru – nezatěžujeme totiž pozorovaný objekt touto logikou. (Zpracováno dle [11], [12])

Při implementaci rozlišujeme dva typy objektů – pozorovatel (`Observer`) a pozorovaný (`Observable`). Pozorovaný objekt při změně svého stavu vyvolává notifikaci, kterou ihned rozešle všem svým pozorovatelům. V rámci notifikace můžeme předat i informaci o dané změně, popř. odeslat referenci objektu, který notifikaci vyvolal. To se hodí zejména v okamžiku, kdy pozorovatel pozoruje více tříd současně a chce identifikovat, od koho změna přišla. Při implementaci tohoto vzoru vystačíme s jednou třídou a rozhraním – `Observable` a `Observer` (Obrázek 11). (Čerpáno z [11], [12])



Obrázek 11 – UML diagram návrhového vzoru Observer
Zdroj: Vlastní zpracování

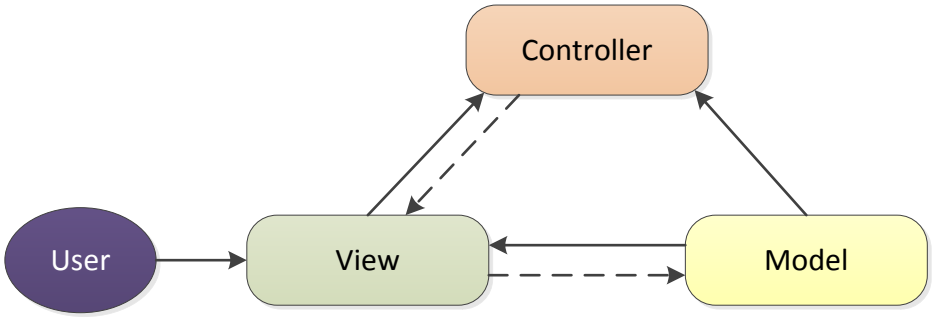
Klíčové jsou třídy `Observable` a `Observer`. `Observer` je pouhé rozhraní, které musí implementovat každý pozorovatel. V rozhraní je deklarována pouze jedna metoda, která je zavolána při vyvolání notifikace. Třída `Observable` je bázovou třídou, zpravidla abstraktní, pro objekt, který bude pozorován. V této třídě je udržován seznam všech pozorovatelů. Dále zavádí operace pro přidávání, resp. odebírání pozorovatele a operaci pro vyvolání notifikace. Tato operace je označená jako `protected` a voláme ji při změně stavu objektu. U všech pozorovatelů je pak zavolána metoda `update()`, která navíc předá odesílatele a případné argumenty (například název vlastnosti, která byla změněna).

```
public void setLocalTime(double time) {
	this.localTime = time;
	notifyObservers("localTime");
}

protected void notifyObservers(Object args) {
	for(Observer observer : observers) {
		observer.update(this, args);
	}
}
```

Při tvorbě větší aplikace tento vzor opravdu najde silné opodstatnění. Jsme schopni vytvářet závislosti mezi různorodými částmi aplikace, aniž bychom museli zasahovat do klientského kódu. Jednoduše jen řekneme, co chceme sledovat. Samozřejmě to sebou přináší i několik úskalí, které tento vzor má. Zásadním problémem může být nutnost dědit z bázové třídy u sledovaného objektu. Můžeme mít vlastní hierarchickou strukturu dědičnosti, do které je zapojen i sledovaný objekt. V tomto případě je na místě dát přednost kompozici před dědičností. Dalším problémem mohou být nekontrolované aktualizace. Tento stav nastává v okamžiku, kdy máme více sledovaných objektů a pozorovatelů, mezi kterými jsou závislosti. Jedna změna může vyvolat takovou notifikaci, která se začne kaskádovitě šířit dále. Tyto podvržené aktualizace se velmi špatně detekují a mohou vést až do stavu zacyklení. (Zpracováno dle [12], [13])

4.5.2 Architektura Model View Controller

Hlavní myšlenkou vzoru Model View Controller[7] je striktní oddělení aplikační logiky, datového modelu a uživatelského rozhraní do tří nezávislých komponent. Velmi populární je zejména na poli architektury webových aplikací, ale málokdo ví, že lze aplikovat i na klasické desktop aplikace. Existuje i mnoho Frameworků, které architekturu MVC používají – v případě platformy Java to je například velmi populární Framework Spring. MVC je velmi komplexní architektura a mohli bychom o ní napsat celou knihu – my si ale pouze představíme základní klíčové pilíře. (Čerpáno z [14])



Obrázek 12 – Základní schéma architektury MVC
Zdroj: Vlastní zpracování

Na obrázku výše (Obrázek 12) je zachycen koncept MVC architektury. Obsahuje tři stěžejní části: (Čerpáno z [15])

- **Model** (model) – reprezentace dat a business logiky aplikace (doménová vrstva),
- **View** (pohled) – převádí data z modelu do grafické podoby pro uživatele,
- **Controller** (řadič) – reaguje na všechny podněty a aktualizuje Model a View.

Princip životního cyklu MVC je následující. Na počátku je akce od uživatele (například stisk tlačítka), kterou převezme Controller. Controller se podívá na aktuální stav modelu a dle potřeby ho zaktualizuje. Controller vydá pokyn k aktualizaci pohledu. Komponenta

[7] MVC – Model View Controller

View se vykreslí dle aktuálního stavu modelu. Nyní opět čekáme na další akci od uživatele, který celý cyklus zopakuje. [14], [15]

Java v základu MVC nepodporuje a je proto nutné použít Framework třetí strany nebo si jednotlivé složky musíme navrhnout sami a dodržovat jejich odpovědnosti.

4.5.3 Inversion of control

Jedná se o obracené řízení toku programu oproti klasickému procedurálnímu programování. Co to znamená? V klasickém programování vytváříme nějakou třídu, která následně ke své činnosti využívá nějaké třídy, ty pak další a tak dále. Třídy jsou mezi sebou velmi těsně svázány a změna používané třídy za jinou vyžaduje velký zásah do kódu. Na druhou stranu obrácené řízení se snaží tuto vazbu uvolnit. [16]

Každá třída ví, co ke své činnosti přesně potřebuje (tzv. závislosti). Nesmíme ale dopustit, aby si naše třída vytvářela instance závislostí sama. Existuje několik způsobů, jak předávat hotové instance do třídy. Nejvíce používané jsou následující dva: [17]

- **Constructor Injection** – všechny závislé třídy jsou vkládány skrze kontruktor,
- **Setter Injection** – pro všechny závislosti musí být definovány patřičné settery.

Častěji se setkáváme s technikou Constructor Injection. Neznamená to ale, že by druhý přístup byl špatný – kupříkladu Framework Spring Setter Injection používá. Pojďme se podívat na praktické použití obráceného řízení s technikou Constructor Injection.

```
public class SimulatorRunner {
    private ISimulator simulator;

    public SimulatorRunner(ISimulator simulator) {
        this.simulator = simulator;
    }

    public void simulate(ISimulationModel model) {
        // prepare runner
        // ...
        simulator.simulate(model);
    }
}
```

V ukázce je třída `SimulatorRunner`, která řídí běh simulace. Ke své činnosti potřebuje pouze nějaký simulátor. Vyžádá si ho proto v konstruktoru. Povšimněte si, že jsme také docílili zvýšení abstrakce naší třídy. Nežádáme žádný konkrétní simulátor, nýbrž si pouze deklarujeme obecný simulátor s určitými operacemi (skrze rozhraní). Simulátor může být implementován mnoha způsoby a v případě jeho výměny nemusíme vůbec měnit kód třídy `SimulatorRunner`. [17]

Bez použití IoC by si třída `SimulatorRunner` vytvářela instanci simulátoru sama a pokud bychom chtěli simulátor vyměnit, pak bychom museli zasahovat do kódu třídy

`SimulatorRunner`. Dále také musí znát `SimulatorRunner` všechny závislosti, které jsou nezbytné pro sestavení simulátoru. A to není z hlediska architektury správný přístup.

Inversion of control přináší zejména zvýšení abstrakce a rapidní uvolnění vazeb mezi závislostmi našich tříd. Bez větších zásahů můžeme vyměňovat různé části aplikace, apod. Nezajímá nás, jakou konkrétní implementaci dostaneme, ale pouze požadujeme objekt s určitými operacemi. Často jsou tyto závislosti dokonce definovány úplně mimo aplikaci, například v XML souboru. V dalších případech existuje tzv. IoC kontejner, který strom závislosti sestavuje a stará se i o sestavování objektů. (Čerpáno z [17], [18])

4.5.4 Service Locator

Tento vzor oceníme zejména v situaci, kdy máme v aplikaci definované závislosti, které si nechceme předávat skrze konstruktor nebo setter. Ideou tohoto vzoru je, že máme nějaký globální objekt, který nám dokáže zprostředkovávat konkrétní služby, na základě znalosti rozhraní nebo nějakého parametru. To znamená, že Service Locator eviduje pro každý interface hotové implementace a zná všechny závislosti v celé aplikaci. [19]

```
ISecurityManager security = ServiceLocator.getService(ISecurityManager.class);
```

Na ukázce výše je zachycen typický příklad použití vzoru Service Locator. Potřebujeme získat nějakou low-level službu. Úkolem Service Locatoru je tuto službu poskytnout. Interně se Service Locator podívá do svého seznamu všech závislostí a vyhledá všechny služby, které splňují daný kontrakt. Ty posléze vrátí.

Používáním tohoto vzoru docílíme větší abstrakce a naše aplikace budou více rozšiřitelné. Své místo najde i při definování různých konfigurací (například globální správce grafiky).

4.5.5 Fluent interface

Fluent interface je návrhový vzor, který je určen pro vytváření hezkých objektově orientovaných API. Tvůrci tohoto návrhového vzoru jsou Eric Evans a Martin Fowler [13]. Síla tohoto vzoru zejména tkví v tom, že jeho používání vysoce zvyšuje čitelnost našeho kódu. Do češtiny je tento vzor často překládán jako *zřetězené volání*. (Čerpáno z [13])

Použití tohoto vzoru je velice jednoduché. Klíčem je, aby metody naší třídy vracely svůj kontext. To znamená, že metoda objektu, který je typu Fluent, provede svoji práci a následně vrátí objekt samotný. Proto můžeme volání postupně řetězit. V následující ukázce kódu jsou zachyceny dvě situace, kde v jedné vzor použit není a v druhé naopak je:

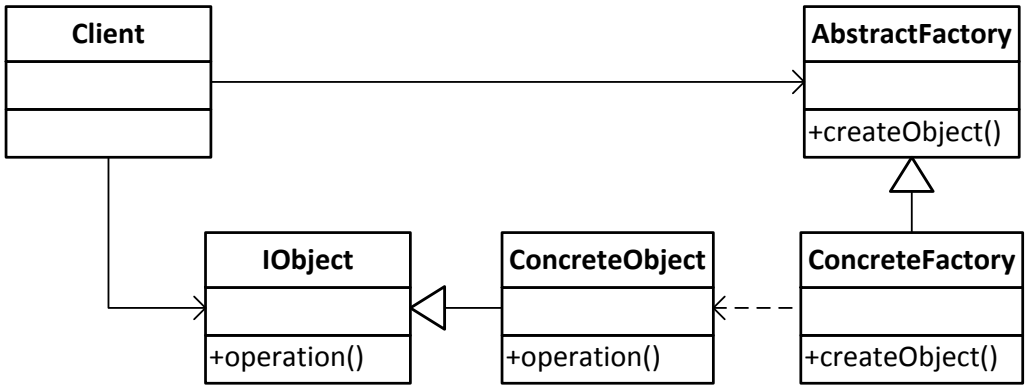
```
// nepoužíváme vzor Fluent interface
User user = new User();
user.setFirstName("Jirka");
user.setLastName("Pénzeš");
user.setAge(25);

// používáme vzor Fluent interface
User user = new User()
                .setFirstName("Jirka")
                .setLastName("Pénzeš")
                .setAge(25);
```

Ukázky se nijak funkčně nerozchází. Kód dělá totožnou činnost a ve stejném pořadí. Vzor Fluent pouze kód více zpřehledňuje a do jisté míry práci s objektem usnadňuje.

Obecně pro tento vzor platí, že by všechny metody měly vracet aktuální kontext (`this`) nebo nově vytvořenou kopii na základě aktuálního kontextu. Častěji se používá ta první varianta. Mnohdy ale chceme zřetězené volání explicitně ukončit, aby již dále nešlo pokračovat. Z tohoto důvodu bývá připravená i nějaká metoda, která má návratovou hodnotu typu `void` a slouží výhradně k terminaci zřetězeného volání. Své uplatnění vzor Fluid interface nachází zejména v různých konfiguracích nebo při sestavování složitých objektů (například databáze, dependency kontejnery, apod.). (Čerpáno z [13])

4.5.6 Vzor Factory

Návrhový vzor Factory má mnoho různých variací. My se budeme věnovat abstraktní továrně (abstract factory). Na úvod si ještě řekneme, k čemu se vzor abstraktní továrna používá. Cílem vzoru je maximální snaha odstínit klienta od vytváření instancí nových objektů. Definujeme si pouze obecné rozhraní pro vytváření objektů (továrnu). O samotné vytváření objektů se starají konkrétní implementace tohoto rozhraní (Obrázek 13). Zpravidla definujeme i obecné rozhraní nebo bázovou třídu pro objekt, který chceme vytvářet. (Čerpáno z [12], [13])



Obrázek 13 – UML diagram návrhového vzoru Abstract factory
Zdroj: Vlastní zpracování

Z diagramu je patrné, že klient pracuje s obecnou tovární třídou a s definicí objektu. Konkrétní detaily nezná a je od nich odstíněn. Z hlediska návrhu je správné přenechávat vytváření objektu na někom jiném. Využití nalezneme zejména v situacích, ve kterých objekty vytváříme často nebo jejich sestavení je náročné. Vytváření instancí objektů může vypadat například následovně: (Upraveno podle [12], [13])

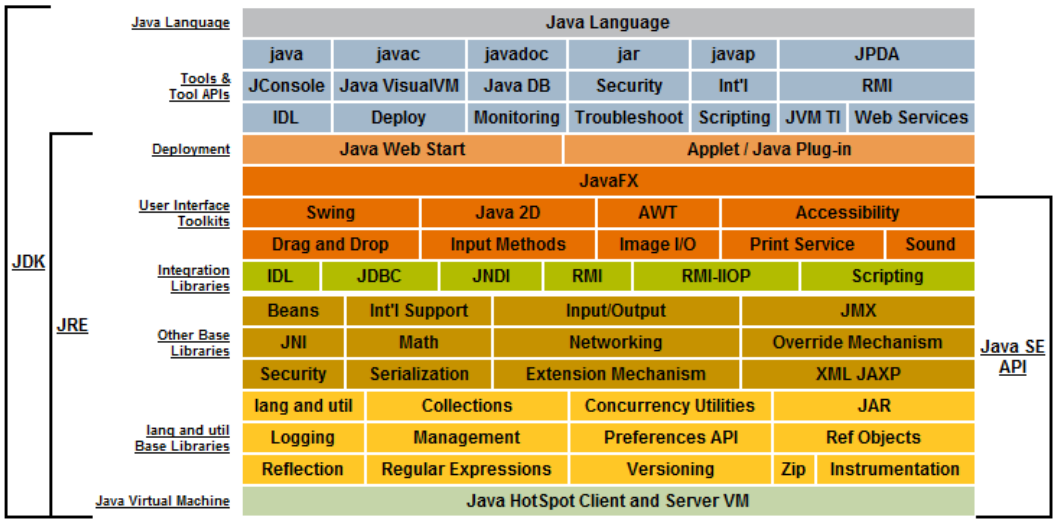
```
IObject object = factory.createObject();
```

Víme, jaký typ objektu dostaneme, ale už nevíme nic o jeho vytváření. Abstraktní továrna se může starat i o nastavení objektů, které vytváří.

4.6 Java Remote Method Invocation

Java RMI je speciální knihovna, která je integrovanou součástí platformy Java již od verze JDK[8] 1.1 [21] (Obrázek 14). Tato knihovna nám umožňuje volání a používání objektů, které mohou mít svou instanci na jiném JVM[9] (typicky v jiném adresním prostoru na jiném počítači), než ze kterého jsou volány. Mezi hlavní výhody Java RMI patří její používání. Se vzdálenými objekty pracujeme totožně, jako s těmi lokálními. Vývojář tedy nemusí vůbec tušit, že pracuje se vzdálenými objekty. Tento přístup je pro programátory velice přirozený a vývoj distribuovaných aplikací je díky tomu snazší. Běh klienta a serveru na různých JVM však není podmínkou.

První verze Java RMI, která byla uvedena v JDK 1.1, používala protokol Java Remote Method Protocol (JRMP). Tento protokol bylo možné používat pouze na platformě Java a nešlo to nijak obejít. Z toho vyplývá zásadní nevýhoda – klient a server musí být napsaný v Javě. S příchodem JDK 1.3 byla uvedena i nová verze knihovny RMI [21]. Tato verze přinesla jednu zásadní změnu – protokol JRMP byl zaměněn za protokol RMI-IIOP[10]. Tento protokol byl zvolen zejména z důvodu, aby bylo RMI kompatibilní s populárním standardem CORBA[11]. Standard CORBA je platformě nezávislým. Knihovna Java RMI se nachází v balíčku `java.rmi` (Obrázek 14).



Obrázek 14 – Java Standard Edition 1.7 API
Zdroj: http://docs.oracle.com/javase/7/docs/

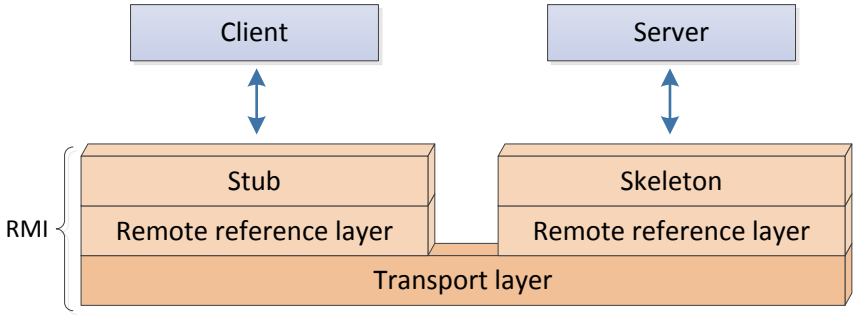
4.6.1 Architektura RMI

Architekturu RMI lze rozdělit do tří vrstev:

- stub a skeleton vrstva,
- vrstva, která se stará o řízení vzdálených referencí,
- transportní vrstva. [21]

[8] Java Development Kit
[9] Java Virtual Machine
[10] Remote Method Invocation – Internet Inter-Orb Protocol
[11] Common Object Request Broker Architecture

Vzájemná provázanost vrstev je znázorněna níže (Obrázek 15). Důležitou vlastností této architektury je skutečnost, že jednotlivé vrstvy lze rozšířit či zaměnit, aniž bychom ovlivnili ostatní části systému. Kupříkladu můžeme zaměnit transportní vrstvu, která implicitně používá protokol RMI/IIOP, za jiný protokol. (Čerpáno z [21], [22])



Obrázek 15 – RMI vrstvy
Zdroj: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Vrstva stub, respektive skeleton, zapouzdřuje speciální objekty Stub a Skeleton, prostřednictvím kterých komunikujeme (viz kapitola Objekty stub a skeleton). Další vrstvou je Remote reference. Tato vrstva je jádrem RMI mechanismu. Transformuje naše požadavky na zprávy, kterým komunikační protokol RMI rozumí. Dále dokáže navazovat spojení, ať už ze strany klienta nebo serveru (bindování a volání objektů v síti). Poslední, transportní vrstva, se již stará o samotnou komunikaci mezi různými JVM. Zajišťuje spojení mezi klientem i serverem. Vrstva je to velice odolná a poradí si s řadou síťových překážek. (Zpracováno dle [20], [23])

4.6.2 Princip

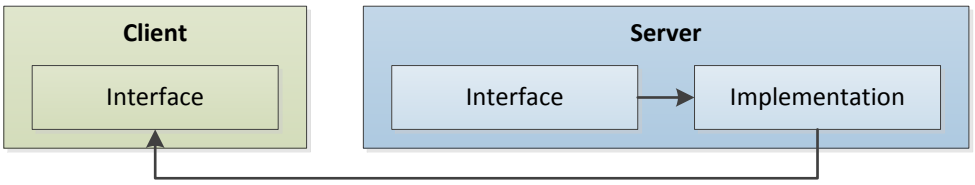
Aplikace, které využívají technologie RMI, jsou typicky postavené na principu architektury klient/server. Klient a server jsou reprezentovány dvěma samostatnými aplikacemi. Server poskytuje a definuje vzdálený objekt[12]. Tento objekt si můžeme také představit jako seznam poskytovaných služeb, které může klient následně využívat a volat. Při inicializaci spojení se využívá tzv. lazy loading - pakliže se nějaký klient přihlásí ke vzdálenému objektu, pak server tento objekt neposkytuje ihned, ale až v okamžiku, kdy jej klient skutečně požaduje – typicky volá nějakou metodu. (Čerpáno z [23], [25])

Jednotlivé role účastníků systému nejsou závazné. Server může být zároveň i klientem a komunikovat s objekty z jiného virtuálního stroje. Toto samozřejmě platí i opačně pro klienta, který může být zároveň serverem. Lze tak vytvořit rozsáhlou komunikační síť. Hlavním aspektem celého mechanismu je definice rozhraní vzdáleného objektu, které musí znát klient i server.

Chceme-li poskytnout vzdálený objekt, je nutné přesně definovat jeho rozhraní. Toto rozhraní musí obsahovat všechny metody, které chceme v rámci vzdáleného volání využívat. Jednotlivé metody mohou přejímat i libovolný počet vstupních parametrů

[12] Vzdálený objekt z anglického Remote Object

a mohou také definovat návratový typ. Lze říci, že RMI principiálně ke své funkčnosti využívá striktního oddělení chování třídy od její samotné implementace (Obrázek 16).

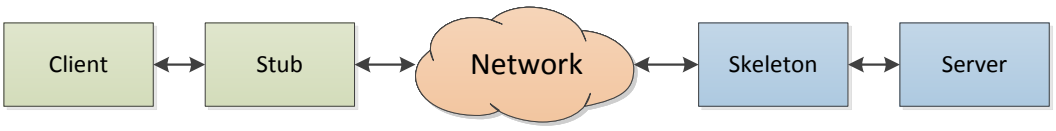


Obrázek 16 – Vztah mezi serverem a klientem
Zdroj: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Rozhraním je myšleno standardní Java rozhraní, které neobsahuje žádný spustitelný kód. Veškerý kód je obsažen až ve třídě, která toto rozhraní implementuje. [22]

Objekty stub a skeleton

Klíčovou roli v komunikaci hrají objekty stub a skeleton (Obrázek 17). Na tyto objekty nemáme vliv a obvykle bývají strojově generovány speciálním kompilátorem. Objekt skeleton si vytváří server, při registraci RMI objektu, a jedná se pouze o zástupný objekt pro skutečnou instanci. Role tohoto objektu spočívá v přejímání požadavků ze sítě, které transformuje a předává skutečné implementaci. Server může pod jedním rozhraním zaregistrovat i více objektů. Skeleton musí zajistit, aby klient dostal odpověď od té implementace, kterou požaduje. Pro tento účel má každý registrovaný objekt unikátní identifikátor. Každý vzdálený objekt je nutné zaregistrovat v tzv. RMI registru. [21]



Obrázek 17 – Jednoduché schéma průběhu RMI komunikace
Zdroj: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Protějšek ke skeletonu je stub, který běží lokálně u klienta. Klient nemá k dispozici skutečnou implementaci vzdáleného objektu – proto dostává pouze zástupný objekt stub. Objekt stub implementuje stejné rozhraní jako vzdálený objekt. Jestliže klient zavolá nějakou metodu vzdáleného objektu, pak přichází na scénu právě stub. Naváže spojení se vzdáleným objektem a vysílá požadavek, který následně dostává skeleton. Požadavkem je volání vzdálené metody. Předáváme-li metodě nějaké parametry, pak je potřeba každý parametr před odesláním serializovat (marshalling). Stub musí vždy počkat na dokončení metody, kterou volal. Po dokončení metody na serveru skeleton zašle odpověď. Odpovědí může být jakákoliv návratová hodnota volané metody. Veškeré data, která jsou v síti přenášena, musí být serializována. Je proto zapotřebí návratovou hodnotu před odesláním na serveru serializovat a po převzetí u klienta deserializovat (unmarshalling). Pokud volaná

metoda nemá žádný návratový typ, pak objekt stub dostává pouze informaci o dokončení metody na serveru.

Objekty stub a skeleleton je možné ručně generovat pomocí kompilátoru rmic. Od verze JDK 1.3 je tento proces plně zautomatizován a objekty jsou generovány automaticky pomocí reflexe[13]. Nemusíme se o nic starat. (Čerpáno z [21])

Interně je RMI postavené na návrhovém vzoru Proxy. Tento vzor je charakteristický tím, že nějaký objekt je reprezentován jiným objektem, který požadavky pouze přeposílá dál. V případě RMI je proxy objektem právě stub, který požadavky posílá skeletonovi.

Chceme-li vytvořit aplikaci, která bude využívat RMI, pak lze postupovat podle následujících pokynů:

1. Definujeme chování vzdáleného objektu (společné rozhraní).
2. Vytvoříme serverovou aplikaci.
    a. Implementujeme vzdálený objekt, dle definovaného společného rozhraní.
    b. Zaregistrujeme objekt do RMI registru.
3. Vytvoříme klientskou aplikaci.
    a. Prostřednictvím společného rozhraní lze volat vzdálené metody.
4. Generujeme objekty Stub a Skeleton (zautomatizováno).
5. Spustíme server a RMI registr.
6. Spustíme klientskou část a otestujeme funkčnost systému.

4.6.3 Implementace RMI v praxi

Prvním krokem je konstrukce společného rozhraní. Toto rozhraní musí být přímo nebo nepřímo odvozené od rozhraní `java.rmi.Remote`. Požadavkem RMI je, aby každá definovaná metoda obsahovala klauzuli `throw` pro výjimku `java.rmi.RemoteException`. Výjimka je vyžadována z důvodu možné chyby při samotné komunikaci mezi klientem a serverem. Při pokusu volat vzdálenou metodu může dojít k chybě. Kupříkladu nedostupnost serveru, výpadek připojení nebo neočekává výjimka v době zpracování vzdálené metody. Vstupními parametry metod mohou být primitivní, ale i referenční typy. Pokud chceme použít referenční typy, pak je nutné, aby implementovaly rozhraní `Serializable`. Stejná podmínka platí i pro návratovou hodnotu metody. Rozhraní musí obsahovat všechny služby, které chceme v rámci vzdáleného volání využívat – nic jiného nám nebude dovoleno. Rozhraní musí být deklarováno jako public. Nově vzniklé rozhraní může implementovat i jiná rozhraní, pro která ale platí stejná pravidla.

[13] Java Reflection API

Pro demonstraci RMI jsem zvolil jednoduchou aplikaci kalkulačku, která bude umět dvě matematické operace - sčítání a odčítání. Tyto operace budou implementovány na server, který je bude poskytovat jako služby skrze RMI. Klient je bude prostřednictvím vzdáleného volání metod využívat. Společné rozhraní kalkulačky:

```
public interface IRemoteCalculator extends java.rmi.Remote {

    public double add(double a, double b) throws java.rmi.RemoteException;

    public double subtract(double a, double b) throws java.rmi.RemoteException;
}
```

Vytvoření serverové části

Pokud máme definované rozhraní se všemi službami vzdáleného objektu, pak se můžeme pustit do implementace serverové aplikace. Server se skládá ze dvou částí, které je nutné implementovat. Nejprve musíme implementovat naší kalkulačku dle předepsaného rozhraní. Druhou částí je samotné spuštění serveru. Spuštění serveru provedeme standardně v metodě `main`. Úkolem serveru je vytvořit instanci vzdáleného objektu (kalkulačky) a tu následně zaregistrovat do RMI registru (rmiregistry). Tento registr je standardní součástí Javy a udržuje seznam všech registrovaných objektů. Každému objektu v registru náleží unikátní identifikátor. Tento identifikátor používá klient, když chce nějaký vzdálený objekt najít a získat. Identifikátor si volíme sami a nemusí se shodovat s názvem třídy, kterou registrujeme. To nám umožňuje pod jedno rozhraní zaregistrovat více instancí jednoho rozhraní. Klient pak jednotlivé instance rozlišuje na základně znalosti identifikátoru. Proto je nutné, aby byl identifikátor unikátní.

Samotná práce s registrem probíhá zejména prostřednictvím třídy `Naming`. Tato třída je `final` a nelze od ní vytvořit instance. Veškeré dostupné operace máme zpřístupněné skrze několik statických metod. Tyto metody slouží jak k registraci objektu, tak i deregistraci a získávání referencí vzdálených objektů. Seznam metod je následující:

- Naming.bind(String name, Remote obj) – registrace nového objektu,
- Naming.unbind(String name) – zrušení registrace,
- Naming.rebind(String name, Remote obj) – obnovení registrace,
- Naming.list(String name) – vrací seznam všech registrovaných objektů,
- Namig.lookup(String name) – získání vzdálené reference na objekt.

Při pokusu registrovat objekt, který je již registrován, skončí metoda bind výjimkou. Z tohoto důvodu je doporučováno využívat spíše metody rebind, která případně stávající referenci přepíše nebo vytvoří novou.

Vzdálený objekt musí navíc, mimo implementování našeho rozhraní, být ještě potomkem třídy UnicastRemoteObject. Jedna z výhod RMI je, že si vzdálený objekt pamatuje svůj

stav mezi voláním jednotlivých klientů. Pokud je tato výhoda pro nás spíše nevýhodou, pak je možné se tomu vyvarovat. Postačí, aby třída byla potomkem třídy `java.rmi.activation.Activable` namísto `UnicastRemoteObject`. Učiníme-li tak, pak bude při každém požadavku vytvořena nová instance (bude volán bezparametrický konstruktor).

```
import java.rmi.RemoteException;

public class RemoteCalculator extends UnicastRemoteObject
  implements IRemoteCalculator {

    public RemoteCalculator() throws RemoteException  {
        super();
    }

    @Override
    public double add(double a, double b) throws RemoteException {
        return a + b;
    }

    @Override
    public double subtract(double a, double b) throws RemoteException {
        return a - b;
    }
}
```

Povšimněte si, že třída obsahuje i konstruktor, který ovšem nezastřešuje žádnou funkcionalitu. Nutnost implementace konstruktoru je dalším požadavkem RMI z důvodu nutnosti deklarování výjimky `RemoteException` nad každou operací vzdáleného objektu, kterou je i konstruktor.

Dalším krokem serverové části je metoda `main`, ve které vytvoříme instanci kalkulačky, a následně ji zaregistrujeme.

```
public class Server {

    public static void main(String[] args) {
        try {
            IRemoteCalculator calculator = new RemoteCalculator();
            Naming.bind("calculator", calculator);

        } catch (AlreadyBoundException exception) {
        } catch (MalformedURLException exception) {
        } catch (RemoteException exception) {
        }
    }
}
```

Za zmínku stojí počet výjimek, které je nutné zachytit a případně na ně reagovat. Důvod je prostý. Knihovna RMI se snaží vývojáře relativně přesně informovat o chybě, která by mohla potencionálně vzniknout, abychom ji mohli patřičně odchytit. Zaregistrovaný objekt poběží v JVM tak dlouho, dokud nebude nikým zrušen. [21]

Vytvoření klienta

Nacházíme se ve stavu, kdy máme připravený vzdálený objekt na serveru. Zbývá implementace klienta. Úloha klienta je prostá – získat referenci na vzdálený objekt a využívat jeho služeb. Pro získání reference (stubu) je využita metoda `lookup`, která je obsažena ve třídě `Naming`. Metoda `lookup` má pouze jediný parametr a to název objektu, který chceme získat. Objekt je vyhledáván v RMI registru.

```
public class Client {

    public static void main(String[] args) {
        try {
            IRemoteCalculator calculator
                    = (IRemoteCalculator) Naming.lookup("calculator");
            double op1 = calculator.add(2, 4);
            double op2 = calculator.subtract(8, 4);
        } catch (Exception exception) {
        }
    }
}
```

Pro úsporu místa jsem všechny výjimky zachytil obecnou výjimkou `Exception`, od které jsou všechny RMI výjimky odvozeny.

4.6.4 Zavedení RMI přes síť

Po detailním zhlédnutí kódu výše jste si možná všimli, že jsme pracovali pouze na jednom virtuálním stroji. Naším cílem je ale vytvořit distribuovaný systém, který bude zpravidla postaven napříč dvěma JVM. Nikde jsme zatím nepracovali s IP adresou či portem, apod. Jak dostat naše objekty do sítě a jak je získávat?

Vytvoření registru

Prvním krokem, který nás čeká, je vytvoření RMI registru – ten bude naše objekty a jejich služby poskytovat dál klientům. Způsoby, jak vytvořit registr, jsou dva. První způsob je využití příkazové řádky, ve které spustíme soubor rmiregistry. Tento soubor je standardně součástí Java SDK[14]. Pohodlnější je ale způsob druhý – vytvořit registr programově. Výhodou je, že spouštění registru má pod kontrolou přímo naše aplikace a díky tomu máme možnost identifikovat problém s vytvářením registru či ovlivnit port, na kterém registr poběží. (Zpracováno dle [21], [22])

Pro práci s RMI registrem je připravena třída `LocateRegistry`, která je součástí balíčku `java.rmi.registry`. Tato třída má několik statických metod. Pro vytváření registru je podstatná metoda `createRegistry`. Parametrem této metody je port, na kterém bude registr dostupný. Implicitně se registr vytváří na portu 1099.

```
Registry registry = LocateRegistry.createRegistry(1099);
```

Metoda `createRegistry` vrací instanci třídy `Registry` – to je zmiňovaný registr. Nad touto instancí můžeme posléze registrovat vzdálené objekty.

[14] Software development kit

```
registry.rebind("calculator", calculator);
```

Všimněte si, že jsme pro registraci objektu nepoužili třídu `Naming`, kterou jsme používali v předchozích případech. Třída `Naming` si implicitně také vytváří registr, který běží defaultně na portu 1099. [21]

Získání vzdáleného objektu

Pokud se nám podařilo vytvořit RMI registr a úspěšně do něj zaregistrovat objekt, pak je před námi již jen jeden krok – získání objektu na straně klienta. Třída `Naming` pracovala nad lokálním strojem a očekávala, že registr poběží lokálně na portu 1099. Chceme-li pracovat v síti a získávat vzdálené objekty z jiného virtuálního stroje, pak musíme i u klienta použít třídu `LocateRegistry`. Naším úkolem je – dostat se ke vzdálenému registru a požádat ho o instanci (stub) ke vzdálenému objektu. Pro tento účel je připravena metoda `getRegistry`, která vyhledá registr v síti a ten se nám pokusí vrátit. Pro získání registru je zapotřebí znát jeho přesné umístění v síti.

```
// implicitní stav - localhost a port 1099
Registry registry = LocateRegistry.getRegistry();

// hledá registr lokálně na portu 9000
Registry registry = LocateRegistry.getRegistry(9000);

// hledá registr na adrese 192.168.1.10 a portu 1099
Registry registry = LocateRegistry.getRegistry("192.168.1.10");

// úplná identifikace
Registry registry = LocateRegistry.getRegistry("192.168.1.10", 9000);
```

Podaří-li se klientovi získat kýžený registr, pak posledním bodem je získání vzdáleného objektu. S tímto registrem pak můžeme pracovat identicky jako s třídou `Naming`. Pro získání vzdáleného objektu postačí využít metodu `lookup`.

```
IRemoteCalculator calculator;
calculator = (IRemoteCalculator) registry.lookup("calculator");
```

Dobrou vlastností RMI je, že vzdálený objekt přetypováváme na rozhraní, které očekáváme. Máme tak přístup pouze k metodám, které jsme definovali, a vše nám staticky hlídá kompilátor již na úrovni zápisu kódu.

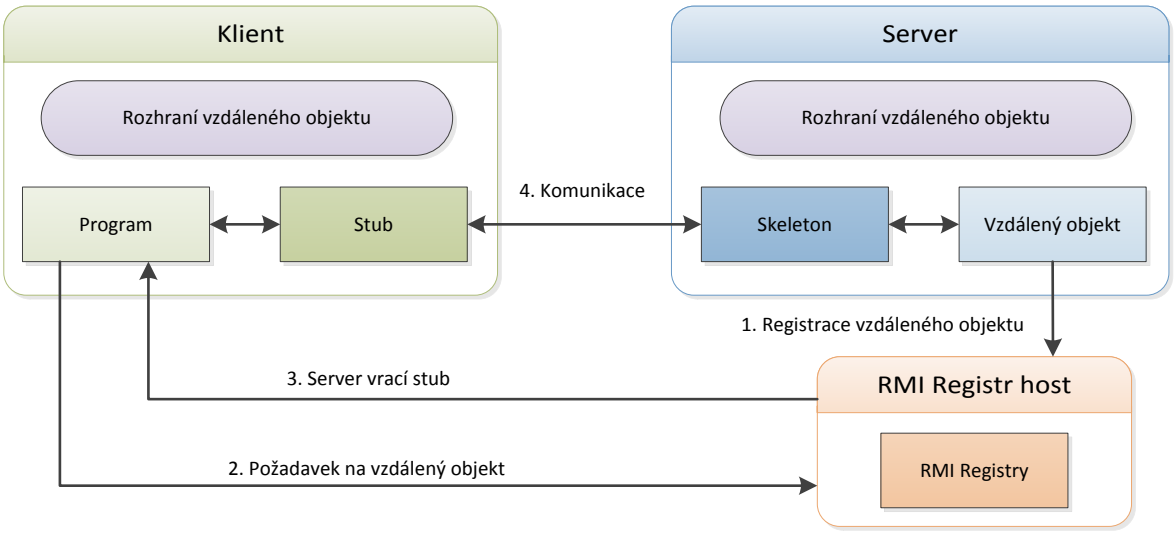
Shrnutí komunikace

Celkové schéma RMI komunikace lze shrnout ve čtyřech bodech:

1. zaregistrování vzdáleného objektu (vznikne skeleton),
2. hledání vzdáleného objektu,
3. vytvoření a předání stubu klientovi,
4. komunikace (vzdálené volání metod). [21]

Celý průběh RMI komunikace je znázorněn na obrázku níže (Obrázek 18). Jednotlivé kroky jsou ve schématu očíslovány. Všimněte si, že RMI registr je zobrazen mimo

serverovou aplikaci. Je tomu opravdu tak – RMI registr interně skutečně běží separátně vedle naší aplikaci.



Obrázek 18 – Schéma průběhu RMI komunikace
Zdroj: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

4.6.5 Bezpečnost

Vstupujeme-li na síť, pak je dobré zamyslet se i nad bezpečnostní politikou. Naše aplikace může dostávat data zvenčí a je proto potřeba řádně přehodnotit důvěru těchto dat, a obecně se zamyslet nad možnostmi, které klientovi v rámci naší služby poskytneme. Je zapotřebí zvolit bezpečnostní strategii, která vhodně omezí přístup k systémovým zdrojům (přístup k souborům, čtení a zápis na disk, komunikační kanály, …). I na bezpečnost knihovna RMI myslí – používají se zde speciální bezpečnostní manažeři (security manager). Tito manažeři řeší, co stuby mohou a nemohou provádět. Pro základní potřeby je přichystána třída `RMISecurityManager`. Nejsme ale nijak svázáni a nic nám nebrání k tomu, abychom si napsali vlastního bezpečnostního manažera. Do začátku a pro nejběžnější požadavky postačí ale ten základní, který umožňuje jen nejnutnější operace. (Čerpáno z [21], [23])

Třída `RMISecurityManager` se stará o spolehlivý chod skrze několik předdefinovaných metod. Tyto metody jsou nezbytné pro správné řízení bezpečnostních pravidel. Bezpečnostní strategie probíhá těsně před provedením potencionálně nebezpečné operace. Bezpečnostní manažer přehodnotí stav, a pakliže zjistí, že se jedná o nepovolenou operaci – vznikne výjimka `RMISecurityException`. Omezení, co stuby mohou provádět, jsou následující:

- accept – přijímat připojení,
- connect – vytvoření spojení mezi aplikacemi,
- listen – naslouchání na daných portech,
- resolve – převod jména na IP adresu.

Pro nás je podstatné, aby stuby mohly realizovat pouze operace, které jsou spojeny s předáváním parametrů metod a přečtením návratové hodnoty. Nic víc umět nepotřebují. Pro spojení mezi aplikacemi lze využít i protokol SSL[15], který chrání a šifruje proudící data. Bezpečnostního manažera je nutné zavést před zahájením RMI komunikace následovně:

```
System.setSecurityManager(new RMISecurityManager());
```

Nezavedeme-li žádného RMI bezpečnostního manažera, pak je použit implicitní, který Java standardně používá. Je jím instance třídy `java.lang.SecurityManager`, který poskytuje přístup pouze k lokáním souborům. (Zpracováno dle [21], [23])

4.6.6 Dynamické nahrávání tříd

Java RMI nám poskytuje relativně hodně svobody. Můžeme přenášet v podstatě jakákoliv data – jedinou podmínkou je, aby byla serializovatelná. Druhá strana si data převezme a provede deserializaci. Java je silně typovým jazykem, a proto musíme ve společném rozhraní všechny datové typy specifikovat. Po deserializaci se pokusí druhá strana objekt zavést. Tuto operaci může provést právě díky znalosti rozhraní, ze kterého vyčte definici třídy. Pro zavádění objektů ze sítě je navržen speciální zavaděč `RMIClassLoader`, který se na požádání pokusí třídu zavést, jestliže třída není zavedena lokálně. [23]

4.6.7 Garbage collector

Posíláme-li po síti různé objekty, pak je dobré se zamyslet i nad jejich životností. Java má velice dobře udělaný garbage collector, který se pečlivě stará o rušení, již nepotřebných, referencí. Klasický garbage collector nám v případě RMI ale nepostačí. Z tohoto důvodu se používá navíc tzv. dynamický garbage collector. Jeho funkce je stejná, jako toho lokáního – rušit již nepotřebné objekty (reference). Vzdálené objekty jsou automaticky rušeny v okamžiku, kdy na ně není odkazován žádný klient. Principiálně funguje na algoritmu počítání referencí. Životnost lze ovlivnit i explicitně pomocí proměnné TimeOut, která v případě uplynutí automaticky zruší objekt. Defaultně je nastavena na 10 minut. [21]

Distribuovaný garbage collector je silná přednost Javy, kterou většina konkurenčních technologií nedisponuje. Jedná se o velice efektivní nástroj, který značně šetří drahou operační paměť.

4.6.8 Shrnutí

Java RMI je velice dobrý nástroj, který maximálně zjednodušuje komunikaci po síti a umožňuje relativně snadno vytvářet distribuované aplikace. Nevýhodou RMI je, že se jedná pouze o jednostrannou komunikaci. To znamená, že jedna strana volá druhou, ale už tomu tak není obráceně. Druhá strana nemá žádné prostředky, jak druhou stranu kontaktovat. Nemůžeme tak klienta například upozornit na nová data, apod. Tuto nevýhodu lze ale obejít. Můžeme navázat nové spojení (opačný směr) nebo se opakovaně dotazovat serveru, zda nejsou nová data. Tyto techniky zavádějí další režii. Pro nás ale

[15] Secure Sockets Layer

takováto nevýhoda nemusí být nutně problémem a vystačíme si bohatě s jednostrannou komunikační linkou.

Další nevýhodou může být neustálá serializace a deserializace, která se provádí s každým vzdáleným voláním. Proces serializace, resp. deserializace může být u složitých objektů značně náročný na čas. Také fakt, že lze komunikovat pouze mezi aplikacemi, které jsou napsané v jazyce Java, může být nevýhodou RMI. Pakliže nás tato nevýhoda bude hodně omezovat, pak máme možnost zaměnit transportní vrstvu. Ovšem záměna transportní vrstvy je implementačně náročná.

Mezi zásadní výhody RMI patří bezesporu jednoduchost a přímočarost použití. Jako vývojáři vůbec nemusíme tušit, že pracujeme se vzdálenými objekty a komunikujeme po síti. Dále máme zaručenou zpětnou kompatibilitu se staršími běhovými prostředími. V neposlední řadě je i značnou výhodou bezpečnostní strategie, kterou RMI používá. Již v základu máme naservírované velice dobré řešení bezpečnosti, které můžeme v případě potřeby dále rozšiřovat a ovlivňovat.

Tato kapitola z velké části vychází z mé bakalářské práce, kde jsem se problematice Java RMI věnoval. [25]

Část 3

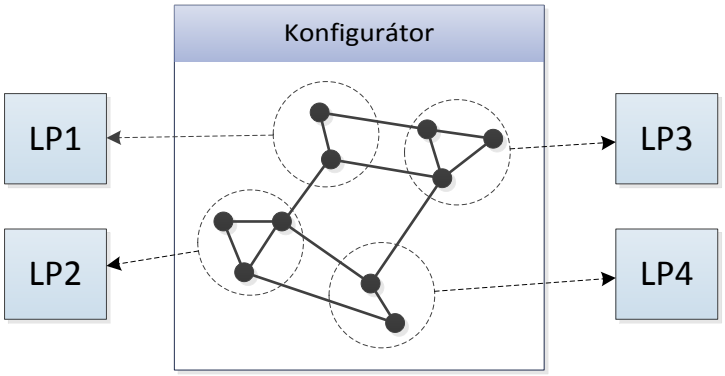
5 Konfigurace distribuovaných modelů

5.1 Centralizovaná a decentralizovaná konfigurace

Stěžejní náplní této práce je naleznutí vhodného mechanismu pro konfigurování distribuovaných simulačních modelů. Existuje několik cest, kterými se můžeme vydat. V podstatě máme dva hlavní způsoby, jakými můžeme distribuované modely konfigurovat:

- centralizovaná konfigurace,
- decentralizovaná konfigurace.

Centralizovanou konfigurací se rozumí, že celou distribuovanou simulaci navrhujeme na jednom počítači. Při spuštění simulačního výpočtu jsou poté alokovány jednotlivé výpočetní uzly v síti, na kterých simulace následně probíhá (Obrázek 19). Celý distribuovaný simulační model vidíme na jednom počítači a víme přesně, co na kterých výpočetních uzlech poběží a jaká je úloha logického procesu v rámci celé sítě.



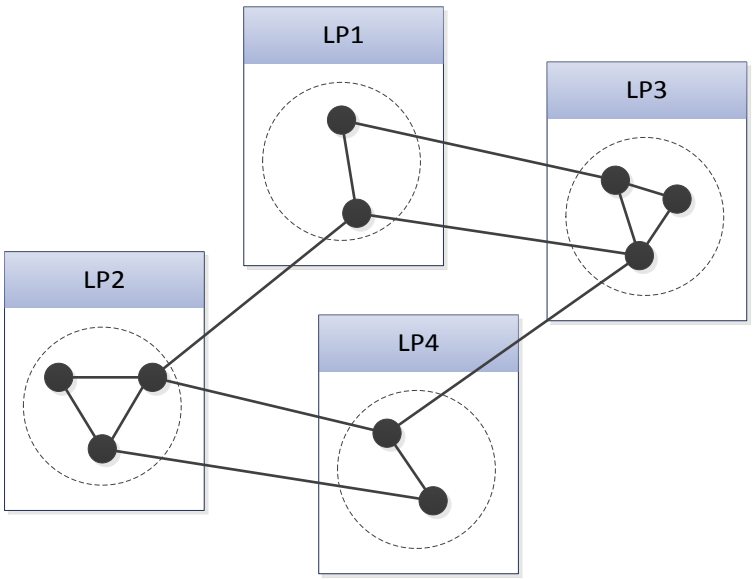
Obrázek 19 – Centralizovaný způsob konfigurace
Zdroj: Vlastní zpracování

Musíme určit jeden stroj, na kterém budeme model navrhovat a ze kterého budeme celou distribuovanou simulaci řídit (Obrázek 19). Musíme znát adresy všech výpočetních uzlů, na kterých chceme výpočet alokovat. Hlavní výhodou může být znalost celého distribuovaného simulačního modelu. Víme přesně, co který výpočetní uzel provádí, jak to provádí a kde to provádí.

Na druhou stranu decentralizovaný způsob konfigurace funguje obráceně. Není zde žádný centralizovaný prvek, na kterém bychom navrhovali celý distribuovaný simulační model.

Celý distribuovaný simulační model je logický rozčleněn na části (logické procesy). Tyto části jsou následně spouštěny na různých výpočetních uzlech – stejně tomu tak bylo i v případě centralizovaného konfigurování. Nyní ale navrhujeme distribuovaný model po těchto částech. Činnost každého logického procesu musíme navrhnout samostatně. Když

navrhujeme model logického procesu, tak víme, co od něj požadujeme. A také víme, jaké by mělo být jeho nejbližší okolí, se kterým chceme komunikovat. To znamená, že při návrhu jednoho logického procesu postačí znát pouze adresy logických procesů, které jsou v nejbližším okolí. Logický proces musí přesně definovat své vstupy a výstupy, prostřednictvím kterých lze navázat komunikaci.



Obrázek 20 – Decentralizovaný způsob konfigurace
Zdroj: Vlastní zpracování

Hlavní výhodu decentralizovaného přístupu vidím již v samotné konfiguraci. Každá část může být navrhována někým jiným a může běžet neustále. Jenom vyčkává na zprávy zvenčí. Máme velmi odstíněné jednotlivé části celého simulačního procesu – pokud by byl distribuovaný simulační model složen z mnoha logických procesů, pak postačí, aby každý logický proces znal pouze své nejbližší okolí. Nevýhodou tohoto přístupu je fakt, že musíme spouštět jednotlivé logické procesy samostatně a nemáme nad nimi žádnou globální moc.

Ve své práci jsem zvolil cestu decentralizovaného přístupu. Výsledkem práce je vývojové prostředí, ve kterém navrhujeme činnost jednoho logického procesu. Proto se budu dále věnovat pouze decentralizovanému přístupu.

5.2 Navrhování simulačních modelů

Jakým způsobem navrhovat simulační modely? Pomocí simulační knihovny a jazyka Java můžeme modely vytvářet celkem snadno. Všechny události si naprogramujeme na míru, nejsme takřka ničím limitování. Tuto hranici bych ale rád posunul. Snažil jsem se najít vhodný způsob, který by umožnil uživateli vytvářet distribuované simulační modely v grafickém prostředí deklarativním způsobem. Uživatel nemusí znát žádnou vývojovou platformu.

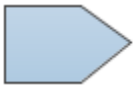
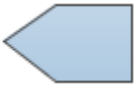
Simulační model v podstatě zachycuje životní cyklus entity. Například v simulačním modelu vlakového nádraží mohou být entitami vlaky, lidé, atd. Pomocí událostí řídíme chování a životnost těchto entit v našem systému. Tuto myšlenku jsem použil. Cílem návrhu simulačního modelu je zachycení kompletního životního cyklu entity. Od vzniku až po zánik.

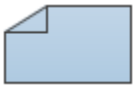
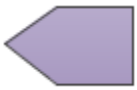
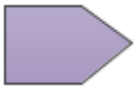
Každá operace, kterou s entitou můžeme provést, je popsána modulem. Moduly by měly striktně dodržovat princip soudružnosti a zastupovat vždy pouze jednu činnost. Tímto krokem docílíme vysokého stupně abstrakce. Uživatel prostřednictvím modulů deklarativním způsobem může popisovat celý simulační model.

5.3 Moduly

Simulační modely jsou popisovány prostřednictvím modulů. Modul si můžeme představit jako reprezentanta události, která se během simulace generuje. Entita putuje z modulu do modulu a na základě přechodů se generují události. Simulační model musí určovat vznik i zánik každé entity, se kterou se v modelu pracuje.

Při návrhu simulačního modelu jsme limitování pouze rozsahem palety modulů. Pro základní potřeby bylo implementováno osm modulů. V tabulce níže je charakterizována jejich funkce, vizuální reprezentace a kapacita vstupů a výstupů. Kapacita vstupů, resp. výstupu udává, kolik můžeme přivést na daný vstup, resp. výstup závislostí.

Modul		Vstupy	Výstupy
	Create Zařizuje vytváření entit. Máme možnost určit, v jakých intervalech se budou entity tvořit, kolik se jich celkově může vytvořit, jaké množství je vytvořeno po uplynutí intervalu, jak bude entita reprezentována a kdy bude poprvé entita vytvořena.	0	1
	Dispose Entita zaniká při vstupu do tohoto modulu. Modul zlikviduje život entity a zaznamená případné statistiky, které z atributů entity vyčte.	MAX	0
	Delay Tímto modulem definujeme časový blok, po který je entita pozdržena.	MAX	1

Modul		Vstupy	Výstupy
	Assign Ke každé entitě můžeme přiřazovat atributy. Tyto atributy lze využít pro přenášení informací mezi modely či pro finální statistiky. Modul Assign poskytuje možnost přiřazovat entitě atributy.	MAX	1
	Outputer Prostřednictvím tohoto modulu můžeme zapisovat vybraná data ze simulátoru a z entit do souboru.	MAX	1
	Condition Modul Condition poskytuje možnost do modelu zanést logiku. Na základě námi definované podmínky můžeme větvit a ovlivňovat život entity.	MAX	2
	Sender Odešle entitu na jiný simulační model v síti. Je zapotřebí definovat adresu vzdáleného stroje a klíč.	MAX	0
	Receiver Zařizuje příjem a vpuštění entity, která přijde zvenčí do našeho simulačního modulu. V nastavení modulu definujeme pouze klíč.	0	1

Zajímavé jsou zejména moduly Sender a Receiver. Prostřednictvím těchto modulů komunikujeme s ostatními simulačními modely v síti (prvek distribuovanosti). Výčet modulů nemusí být konečný. Knihovna je připravena pro případné rozšíření.

5.3.1 Spojování modulů

Moduly deklarují své vstupy a výstupy. Na tyto vstupy, resp. výstupy můžeme přivádět závislosti (moduly). Vytvářením závislostí mezi moduly, potažmo spojováním modulů, utváříme podobu našeho simulačního modulu (Obrázek 21).

Obrázek 21 – Spojování modulů
Zdroj: Vlastní zpracování

Na obrázku výše je zachycen simulační model, který je složen ze tří modulů. Je to úplný model. Povšimněte si, že modul Create nemá deklarovány žádné vstupní body a modul Dispose zase naopak žádné výstupní body. Stmelující je modul Delay, který dokáže utvářet závislosti jak na vstupu, tak i na výstupu.

5.3.2 Kořenové moduly

Speciálním typem modulů jsou tzv. kořenové moduly. Tyto moduly zpravidla určují počátek simulace. V naší paletce modulů je kořenovým modulem Create. Stojí na počátku života entity – vytvoří ji a vpustí do modelu. Do konfigurace modulu spadá i maximální počet generovaných entit. Simulace končí, pakliže dosáhneme maximálního počtu entit. Samozřejmě pouze v případě, kdy model není distribuovaný. Kořenové moduly mohou určovat i maximální počet entit, které vpustí do modelu.

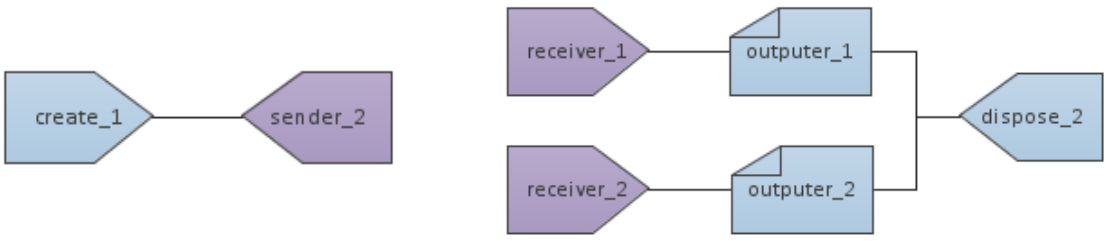
Kořenové moduly určují i počátek simulace. Při spuštění simulátoru jsou na základě těchto modulů generovány první události, které nastartují celý simulační výpočet.

5.3.3 Distribuované moduly

Dalším speciálním typem jsou distribuované moduly. Z naší paletky do této kategorie spadají moduly Sender a Receiver.

Prvním jmenovaným přeposíláme entitu na jiný simulační model v síti. Důležité je, abychom v nastavení modulu správně zadali adresu logického procesu, na který se má entita odeslat. Modul Receiver naopak slouží k přijmutí entity. Lze jej svým způsobem považovat i za kořenový modul, neboť zahajuje život entity v našem simulačním modelu – i když entita dorazila ze sítě. Entitu zpracujeme a se všemi atributy, které již má, vpustí entitu do modelu. Nijak entitu neovlivňuje.

Uvažme distribuovanou simulaci složenou ze dvou logických procesů (Obrázek 22).



Obrázek 22 – Distribuované moduly – LP1 a LP2
Zdroj: Vlastní zpracování

Logika obou logických procesů není nijak složitá. První pouze generuje entity, které následně přeposílá na druhý logický proces. Druhý logický proces entity přijímá, loguje do souboru a entity následně likviduje. Všimněte si, že LP2 má dva moduly Receiver. Který z nich zpracuje entitu, kterou odešle LP1?

Nesmí nastat, aby byla entita zpracována oběma moduly. Musíme jeden vybrat. Ale který? Nebudeme sázet na náhodu, a proto zavedeme klíče. V nastavení modulu Sender definujeme klíč, který bude entitě přiřazen. Moduly Receiver budou mít definován klíč v nastavení taktéž.

Rekapitulujme si to. Před odesláním entity z LP1 je entitě přiřazen klíč – například „car“. Entita dorazí do logického procesu 2. Ten se na základě klíče rozhodne, kterému modulu entitu přiřadí ke zpracování.

Je důležité podotknout, že všechny moduly Receiver musejí mít na úrovni jednoho simulačního modelu vždy unikátní klíče.

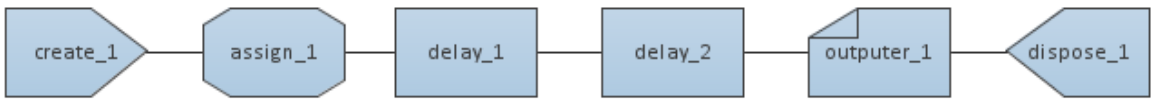
5.4 Navrhování simulačních modelů

5.4.1 Výrobní hala

Mějme výrobní halu, která vyrábí produkty. Problematiku výroby produktu si značně zjednodušíme – cílem je seznámení se způsobem navrhování simulačních modelů.

Hala funguje principiálně jednoduše. Připraví nehotový produkt a produktu přiřadí název. Dokončení produktu probíhá dále na dvou strojích, kde se zkompletuje. Zpracování stroji probíhá sekvenčně v přesném pořadí. Celou výrobu sleduje operátor. Konkrétně sleduje, jak dlouho trvala kompletace produktu. Předmětem zkoumání tohoto simulačního modelu bude právě sledování délky kompletace.

Naším cílem bude popsat uvažovanou výrobní halu deklarativním způsobem. Začněme nejprve modelem, který nebude distribuovaný (Obrázek 23).

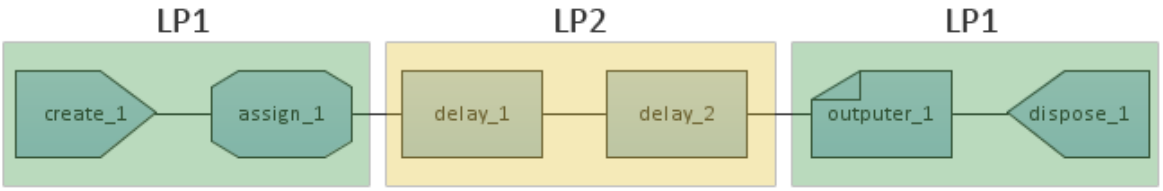


Obrázek 23 – Model výrobní haly
Zdroj: Vlastní zpracování

Simulační model počíná modulem create_1, který vytvoří entitu reprezentující produkt. Modul assign_1 přiřadí produktu jméno. Moduly delay_1 a delay_2 reprezentují zmiňované pracovní stroje, které produkt kompletují. Dále zapíšeme do souboru čas vstupu produktu do modelu a aktuální simulační čas, abychom věděli, jak dlouho byl produkt kompletován. Činnost modulu dispose_1 zařizuje opuštění produktu ze simulačního modulu.

Pokusíme se model výrobní haly rozložit na více logických procesů, aby simulace mohla být distribuovaná. Rozdělíme simulační model do dvou logických procesů (Obrázek 24).

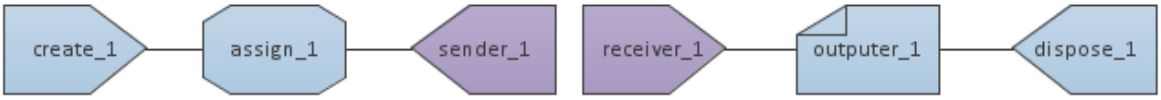
První logický proces bude zastupovat výrobní halu jako takovou. Na starost bude mít vytváření produktů, přiřazování jmen, sledování časů a opuštění produktu z výrobní haly. Činnost druhého logického procesu spočívá v kompletaci produktu. Po dokončení kompletace je produkt zaslán zpět do LP1, kde produkt ukončí svůj cyklus.



Obrázek 24 – Model výrobní haly – rozložení na více logických procesů
Zdroj: Vlastní zpracování

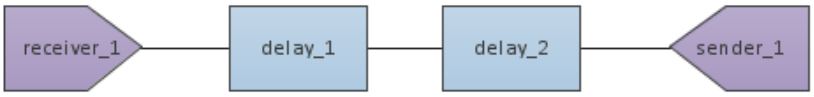
Rozložením docílíme vyrobení dvou simulačních modelů. Oba modely budou vycházet z nedistribuovaného návrhu (Obrázek 23). Obohatíme každý simulační model o distribuované moduly.

První logický proces vytvoří produkt (create_1), přiřadí produktu jméno (assign_1) a následně zašle produkt na logický proces 2 (Obrázek 25). Po návratu produktu z logického procesu 2 (receiver_1) zaznamenáme zkoumané statistiky do souboru (outputer_1) a ukončíme životní cyklus produktu v naší simulaci (dispose_1).



Obrázek 25 – Model výrobní haly – LP1
Zdroj: Vlastní zpracování

Při přijmutí produktu (receiver_1) druhým logickým procesem dojde ke kompletaci. Produkt je přesun ke strojům (delay_1 a delay_2). Po dokončení kompletace je přeposlán zpět na logický proces 1.



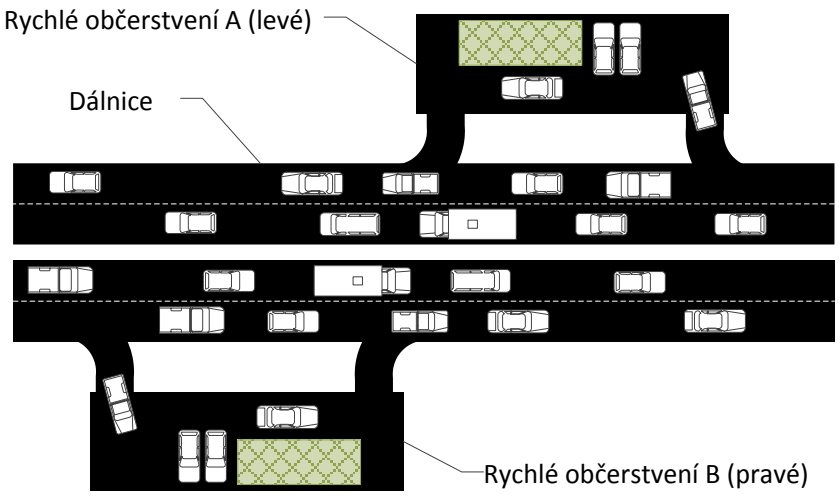
Obrázek 26 – Model výrobní haly – LP2
Zdroj: Vlastní zpracování

Oba logické procesy jsou v této fázi hotové a dalším krokem by již bylo pouze spuštění distribuované simulace.

5.4.2 Dálnice s přilehlými rychlými občerstveními

Uvažme dopravní systém, který odráží reálnou dopravní situaci na dálnici a přilehlém rychlém občerstvení. Předmětem zkoumání budou příjezdy, resp. odjezdy automobilů na této komunikaci a také nás zajímají automobily, které z dálnice odbočily do rychlého občerstvení. Každý automobil, který zamířil do rychlého občerstvení, se po dokončení

obsluhy opět zařadí do provozu na dálnici a pokračuje v jízdě do svého odjezdu. Modelovaný příklad ilustruje Obrázek 27.



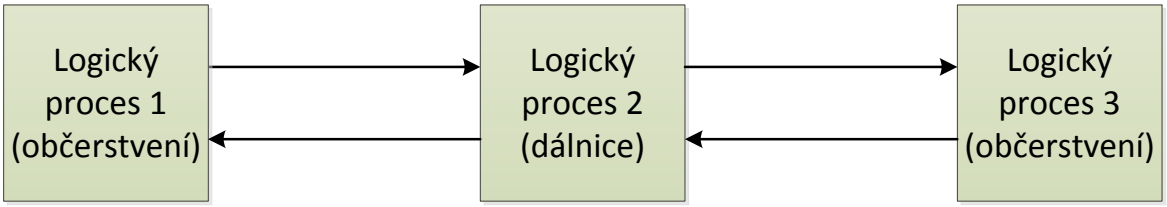
Obrázek 27 – Modelovaný příklad dopravního problému
Zdroj: Vlastní zpracování

Ilustrační motivační obrázek (Obrázek 27) charakterizuje dopravní situaci, kterou se simulační model snaží zachytit. Z obrázku je zřejmé, že model lze rozdělit na tři části:

- dálnice,
- rychlé občerstvení A (levé),
- rychlé občerstvení B (pravé).

Zmíněné stavební bloky našeho modelu si lze představit jako samostatně fungující celky, potažmo logické procesy.

Představa těchto tří logických celků (Obrázek 28) nás přivádí k myšlence model dále škálovat a využít zde výhod distribuované simulace. Každý proces poběží na samostatném stroji a bude mít svůj vlastní výpočetní výkon. Při spuštění celého experimentu jsou pak tyto logické procesy vzájemně provázány a jsou mezi nimi vytvořeny vzájemné závislosti, které dohromady utváří uvažovaný problém.



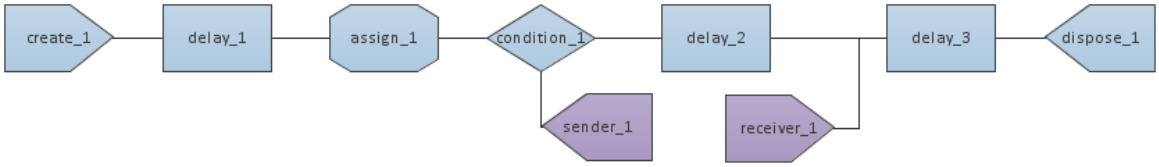
Obrázek 28 – Dopravní systém
Zdroj: Vlastní zpracování

Zkusme uvažovaný problém zapsat deklarativním formalismem, který jsme si popsali, a vytvořit simulační model všech tří logických procesů.

Logický proces – dálnice

Dálnice se skládá ze dvou směrů, kde každý směr má dva pruhy. V odbočovacím pruhu některá vozidla odbočí do rychlého občerstvení a v druhém pruhu vozidlo dálnici pouze projede. Logika je pro oba směry dálnice stejná.

Začněme postupně. Prvním krokem bude návrh odbočovacího pruhu, který bude směřovat do levého občerstvení. Uvažujme 30% pravděpodobnost, že vozidlo odbočí do rychlého občerstvení. Ostatní vozidla dálnicí pouze projedou (Obrázek 29).



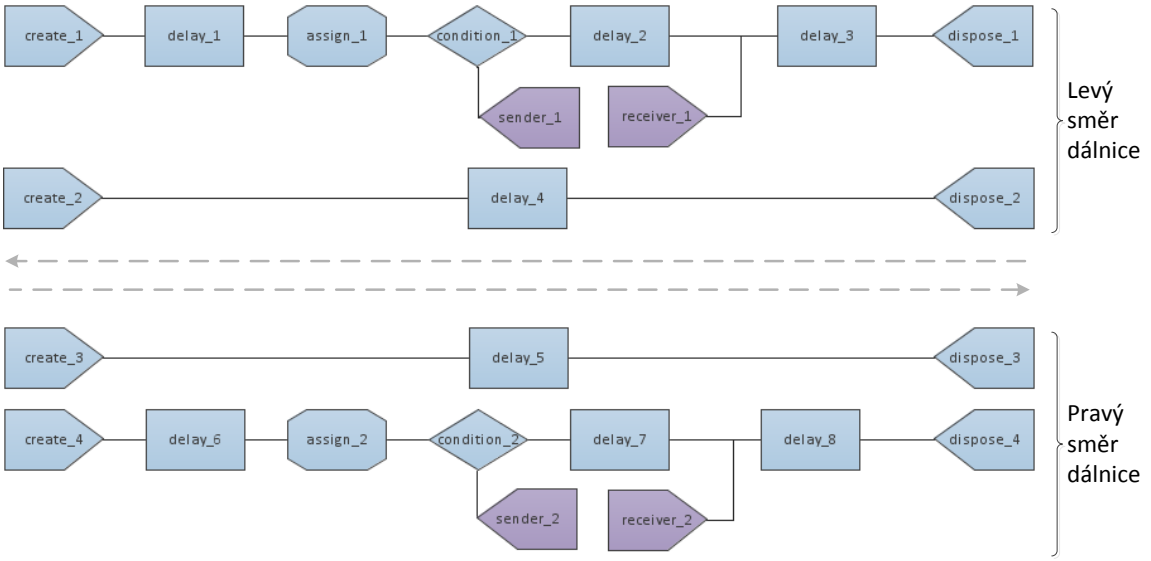
Obrázek 29 – Dálnice – odbočovací pruh
Zdroj: Vlastní zpracování

V prvé řadě musíme vygenerovat automobil (create_1). Automobil nějaký čas jede po dálnici (delay_1). Dojede k odbočovacímu pruhu. Modulem assign_1 přiřadíme vozidlu pravděpodobnost, která určuje, zda odbočí do rychlého občerstvení či dálnici pouze projede. Následuje rozhodovací blok (condition_1). Jestliže pravděpodobnost spadá do zmiňovaných 30%, pak je vozidlo přeposláno do logického procesu Rychlého občerstvení (levé) modulem sender_1. V opačném případě automobil pokračuje dále svou jízdu po dálnici (delay_2). O eventuální návrat z rychlého občerstvení se stará modul receiver_1. Po příjezdu vozidlo dojede zbytek dálnice (delay_3) a tím jeho život v našem simulačním modelu končí (dispose_1).

Druhý pruh má logiku o poznání jednoduší (Obrázek 30). Vygenerujeme vozidlo (create_2), které pouze nějaký čas pobyde na dálnici (delay_4) a dále jeho činnost končí (dispose_2).

Obrázek 30 – Dálnice – průjezdný pruh
Zdroj: Vlastní zpracování

Máme hotový jeden směr dálnice. Druhý směr uděláme identicky. Při konfiguraci jen nesmíme opomenout správně vyplnit klíče distribuovaných bloků. Receiver_1 očekává vozidla z levého občerstvení a modul Receiver_2 naopak vozidla z pravého občerstvení. Je zapotřebí zvolit u Receiver modulů odlišný klíč – například klíč „car:left“ a „car“right“. Při konfiguraci si taktéž musíme dát pozor na moduly typu Sender, aby odesílaly vozidla do kýžených logických procesů. Kompletní simulační model dálnice je modelován na následujícím obrázku (Obrázek 31).



Obrázek 31 – Dálnice – kompletní simulační model
Zdroj: Vlastní zpracování

Logický proces – Rychlé občerstvení A (levé)

O poznání méně komplikovaná, bude logika simulačního modelu rychlého občerstvení. Přijmeme vozidlo z dálnice (receiver_1), obsloužíme (delay_1) a následně pošleme zpět na dálnici (sender_1).

Obrázek 32 – Rychlé občerstvení – levé
Zdroj: Vlastní zpracování

Logický proces – Rychlé občerstvení B (pravé)

Logický proces rychlého občerstvení B na pravé straně dálnice má totožný simulační model, jako občerstvení A na levé straně. Při konfiguraci modelu nesmíme opomenout správné nastavení klíče v modulu Sender. Z důvodu, aby dálnice správně rozlišila, ze kterého občerstvení vozidlo přijelo, a přiřadila jej ke správnému modulu Receiver.

Obrázek 33 – Rychlé občerstvení – pravé
Zdroj: Vlastní zpracování

Nacházíme se ve fázi, kdy máme kompletně hotové simulační modely všech logických procesů. Modely společně tvoří celý distribuovaný simulační model. Podařilo se nám zachytit zamýšlenou podstatu logiky každého logického procesu. Dalším krokem by již bylo pouze spuštění celé distribuované simulace

Při návrhu pracujeme s vysokou mírou abstrakce. V modelovaném příkladu například neřešíme zařazování vozidel z rychlého občerstvení zpět na dálnici. Se současnou paletou nejsme schopni zařazování vozidel vyřešit. Možnou cestou by mohlo být vytvoření zcela nového modulu, který by byl interpretem nějakého jednoduchého jazyka, prostřednictvím kterého bychom zařazovací logiku vozidel mohli popsat a vyřešit.

5.5 Implementace modulu

Pojďme si moduly přiblížit více z pohledu implementace. Všechny podpůrné třídy pro tvorbu modulů se nalézají v balíčku `jDistsim.core.simulation.modules`.

Základem je abstraktní bázová třída Module. Od této bázové třídy musejí být odvozeny všechny moduly – ať už přímo nebo nepřímo. Třída `Module` vymezuje společné chování všech modulů. Operace, které konkretizují modul, jsou definovány jako abstraktní a je nutné je implementovat.

Při implementaci nového modulu začneme vytvořením nové třídy, která bude odvozena od bázové třídy `Module`. Kostra prázdné třídy nového modulu vypadá takto:

```
public class SampleModule extends Module<SampleSettings> {

    public SampleModule(SampleSettings settings, boolean defaultInitialize) {
        super(settings, defaultInitialize);
    }

    @Override
    protected void preInitialization() { }

    @Override
    protected void initializeDefaultValues() { }

    @Override
    protected void resetStates(ISimulator simulator) { }

    @Override
    protected void logic(ISimulator simulator, Entity entity) { }

    @Override
    protected void setProperty() { }
}
```

Moduly jsou v aplikaci vytvářeny dynamicky a většina abstraktních metod slouží pro ovlivnění životního cyklu modulu. Jednotlivé operace:

- `preInitialization` – metoda volána během konstrukce modulu.
- `initializeDefaultValues` – modul by měl definovat své implicitní nastavení.
- `resetStates` – vynulování statistik a všech vnitřních stavů.
- `logic` – konkrétní logika modulu. Co se bude dít s entitou během simulace.

- `setProperty` – obnovení stavových vlastností.

Povšimněte si, že třída je generická. Za generickým typem se skrývá nastavení třídy. Modul musí mít nějaké explicitní nastavení definováno, i když není potřeba cokoliv explicitně nastavovat. Součástí konfigurace modulu je totiž i globální nastavení, jako je název modulu, který musíme při vytváření vždy určit. Například v modulu Delay je v nastavení explicitně definován interval, po který Delay pozdržuje entitu. Pro nastavení modulu je připravená bázová třída `ModuleSettings`, od které jsou konkrétní deklarace konfigurace odvozeny.

5.5.1 Vizuální podoba modulu

Abstraktní třída Modulu popisuje pouze logiku samotného modulu. Nic víc. Pracujeme s deklarativním zápisem v grafickém rozhraní, a proto je nezbytné, abychom připravili pro modul i vizuální reprezentaci.

Render modulů je hodně důležitý, abychom dosáhli určité grafické jednotnosti mezi moduly. Vizuální rendering zajišťuje abstraktní bázová třída `ModuleView`. Z této třídy jsou pro nás důležité dvě metody, které jsme nuceni implementovat:

- `initializeConnectedPoints` – určíme pozicování vstupních – výstupních bodů.
- `getBounds` – v této metodě připravíme polygon, který určuje výsledný tvar modulu.

Vizuální podobu určujeme polygonem. Další metody třídy `ModuleView` lze dle potřeby překrýt. Kupříkladu vykreslení dodatečných grafických prvků docílíme překrytím metody `postDraw()`. Barevné schéma je řízeno aktuálním nastavením v globálním grafickém manažeru. Zaměnit ho můžeme překrytím metody defaultního grafického schématu. Jediné, na co nemůžeme přistoupit a změnit, je grafický kontext, který je před kreslením specificky konfigurován (přidán antialiasing, hlídání grafického schématu, …).

5.5.2 Třídy dle vzoru Factory

V úvodu jsem zmiňoval, že jsou moduly v aplikaci vytvářeny dynamicky. Pro tyto potřeby jsou zavedeny dvě tovární třídy (tovární třídy jsou popsány v kapitole 4.5.6). První z nich určuje, jak se mají moduly a jejich vizuální podoby vytvářet. Novou továrnu odvodíme od třídy `BaseModuleFactory`, která po nás požaduje implementovat metodu pro určení, jak vytvořit samotný modul (`Module`) a metodu pro vytvoření vizuální podoby (`ModuleView`). Druhá tovární třída slouží pro vytváření grafické podpory k modulům. Grafickou podporou jsou myšleny případné grafické dialogy, které v aplikaci můžeme využívat (př. dialog pro nastavení modulu).

5.5.3 Instalace modulu

Rozhodneme-li se přidat modul do mé knihovny, pak musíte implementovat všechny výše popsané části:

- `Module` – logika modulu (práce s entitou, deklarace vstupů/výstupů),

- `ModuleView` – definice vizuální podoby modulu,
- `ModuleSettings` – konfigurace parametrů modulu,
- `BaseModuleFactory` – popis, jak moduly vytvářet,
- `IModuleUIFactory` – grafická podpora (formuláře, …).

Zmíněné třídy jsou nezbytné minimum pro vytvoření funkčního modulu. Existují ještě další, pomocí kterých lze modul ještě více konkretizovat – příkladem může být třída, pomocí které popisujeme nápovědu k modulu (`ModuleDescription`).

Předpokládejme, že jsme zmíněné minimum naimplementovali. Co udělat dál? Modul musíme zapojit do hlavní knihovny, ve které jsou i ostatní moduly. Do této knihovny je třeba umístit modul včetně všech jeho částí pomocí speciálního kontejneru. Na následující ukázce je znázorněno nainstalování modulu Delay.

```
container.bind(Delay.class, new ModuleContainer())
        .toIndex(3)
        .toView(new DelayView())
        .toFactory(new DelayFactory())
        .toUIFactory(new DelayUIFactory())
        .withConfiguration(new ModuleConfiguration("delay",
               UIConfiguration.getInstance().getColorSchemeForBasicModule()))
        .build();
```

Globální kontejner, do kterého moduly registrujeme, se nachází ve třídě `ModuleLibrary` . Všimněte si použití vzoru Fluent Interface, který byl popsán v kapitole 4.5.5. Po dokončení konfigurace kontejneru voláme metodu `build()`, která má na starosti zavedení všech závislostí a modul nainstaluje do knihovny modulů.

5.6 Rozsah palety modulů

Na pestrosti palety modulů jsme závislí při tvorbě simulačních modelů. Moduly určují, jak moc detailně můžeme simulační modely modelovat. Koncepci modulů jsem pojal velmi otevřeně, aby bylo snadné nové moduly přidávat. Implementace by měla být velmi jednoduchá. V podstatě bychom se měli při implementaci zabývat skutečně jádrem problému modulu. Ostatní činnosti jsou již naimplementovány v bázových třídách a o správné nasazení se starají dependency kontejnery.

5.6.1 Podpůrné třídy

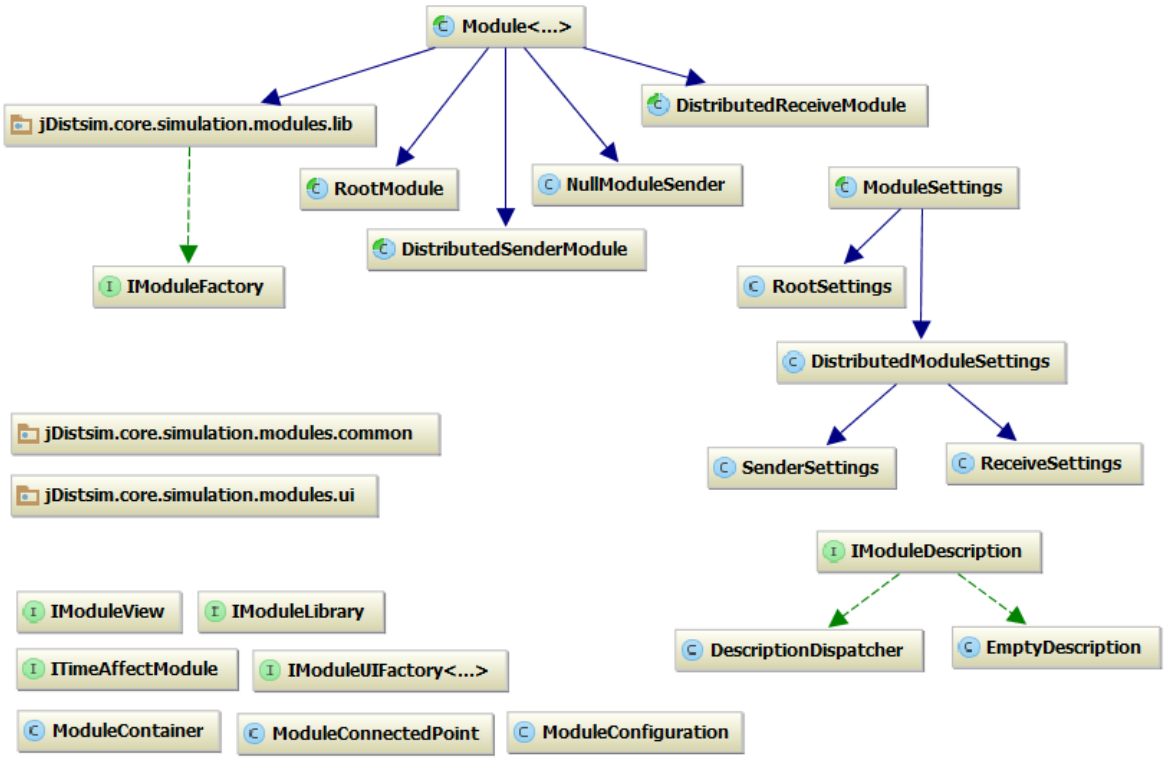
Všechny třídy, které souvisejí s implementací modulů, nalezneme v balíčku `jDistsim.core.simulation.modules`. Doposud jsme modul implementovali odvozením od bázové třídy `Module`. Tento postup lze uplatnit tehdy, když vytváříme klasický modul. Trošku jinak musíme postupovat při vytváření distribuovaných a kořenových modulů.

U kořenových modulů nové třídy odvozujeme od bázové třídy `RootModule`. V případě distribuovaných modulů záleží na tom, zda chceme vytvořit modul typu Sender nebo

Receiver. Dle zvoleného typu odvozujeme nové moduly od třídy `DistributedSenderModule`, resp. od `DistributedReceiveModule`. Pro všechny tři třídy jsou nepřímo odvozeny ze třídy `Module`.

Na obrázku níže (Obrázek 34) je zachycen diagram základních tříd balíčku `modules`. V diagramu je výčet všech klíčových tříd, které při implementaci můžeme použít. Také jsou ve schématu zachyceny tři balíčky:

- `modules.lib` – implementace základních modulů včetně všech částí,
- `modules.common` – pomocné třídy,
- `modules.ui` – obsahuje třídy pro grafickou podporu.



Obrázek 34 – Diagram tříd balíčku modules
Zdroj: Vlastní zpracování

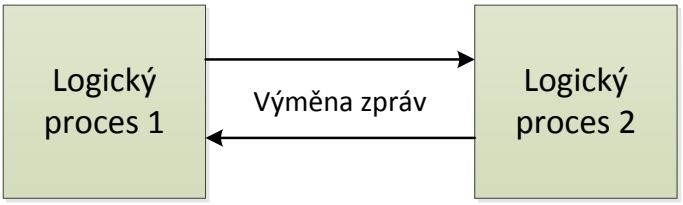
6 Jádro distribuované simulace

Jedním z cílů praktické části bylo implementovat simulační jádro s vhodným algoritmem pro podporu distribuované simulace decentralizovaného charakteru. V této kapitole se blíže podíváme na samotný algoritmus a také si představíme jednotlivé komponenty, ze kterých je jádro složeno. Knihovna je použita ve vývojovém prostředí, které je hlavním výsledkem této práce.

6.1 Algoritmus distribuované simulace

Algoritmus, který používám ve své práci, vychází z konzervativního synchronizačního algoritmu, který ke své synchronizaci využívá zasílání nulových zpráv na vyžádání (kapitola 3.3.2.1). Algoritmus byl lehce modifikován.

Pro lepší interpretaci mého algoritmu si vymezíme pouze dva logické procesy. Předpokládejme dva logické procesy (LP1 a LP2), které mezi sebou komunikují oběma směry. LP1 může přijímat zprávy od LP2 a zároveň LP1 může odesílat zprávy LP2. To samé platí i obráceně. Tyto logické procesy jsou na sobě maximálně závislé.



Obrázek 35 – Logické procesy
Zdroj: Vlastní zpracování

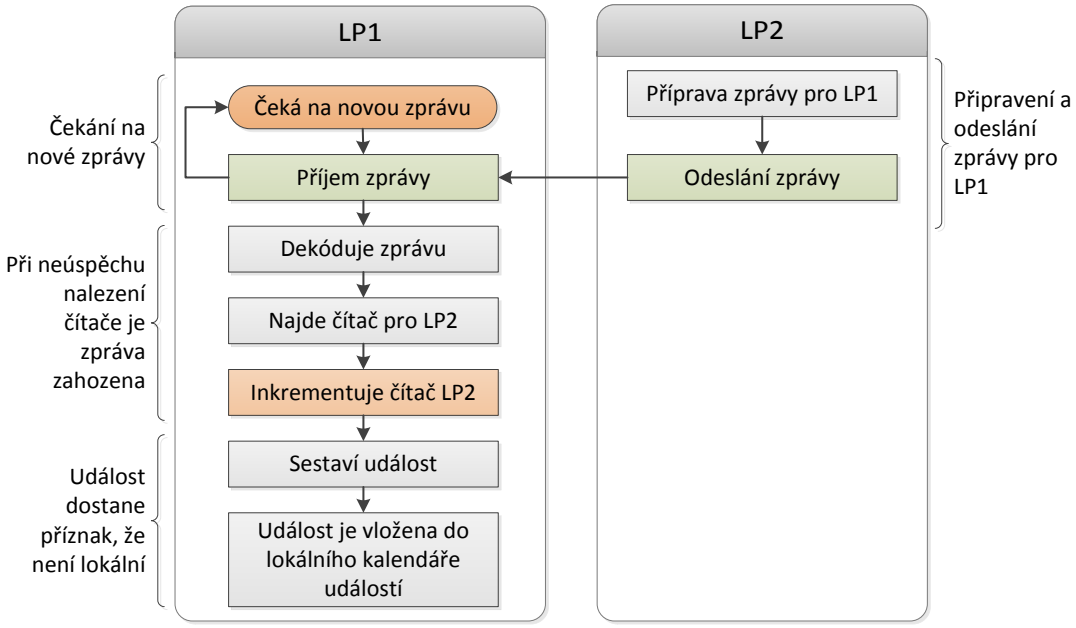
Připomeňme si, že každý logický proces disponuje vlastním simulačním jádrem a kalendářem událostí. K těmto komponentám nemá žádný jiný logický proces přístup. Každý logický proces navenek působí jako služba, skrze kterou s ním můžeme komunikovat. Nyní se ale zaměřme na samotné výpočetní jádro.

6.1.1 Nahrazení vstupních front čítači

Základní synchronizační metoda pracuje s frontami. Pro každý proces je vytvořená fronta a my kontrolujeme její prázdnost. V mém případě je fronta pouze pomyslná a je nahrazena speciálními čítači. Celá distribuovaná simulace je statická a my už při samotném spuštění víme o všech logických procesech, se kterými budeme potencionálně komunikovat. Na těchto procesech jsme závislí. Můžeme si proto dopředu vytvořit pro každý závislý logický proces čítač, namísto fronty.

Zavedení čítačů je aplikováno z důvodu optimalizace a vyhnutí se zbytečnému používání více datových struktur, než je nezbytně nutné. Čítače jsou řízeny samotným zpracováním událostí a při přijmutí nové zprávy zvenčí. Pakliže nám přijde zpráva zvenčí, pak se nejprve inkrementuje čítač (dle odesílatele zprávy), dále je zpráva dekódována a je z ní sestavena událost, která bude naplánována v lokálním kalendáři (Obrázek 36). Tato událost

je označena speciálním příznakem, aby bylo jasné, že se jedná o distribuovanou událost a při zpracování se s ní má zacházet trochu jinak než s lokální. K dekrementaci čítače dochází až při samotném zpracování události – pakliže je událost označena jako distribuovaná, pak je vyhledán patřičný čítač a jeho hodnota následně snížena o jedničku.



Obrázek 36 – Schéma průběhu zpracování zprávy
Zdroj: Vlastní zpracování

Ve schématu výše v podstatě není znázorněn průběh jedné simulační iterace. Doposud nevíme, jak funguje simulační jádro. Proto nevidíme okamžik, ve kterém nastává dekrementace čítače. Algoritmus samotné simulace běží paralelně vedle logiky. Aktuální hodnoty všech čítačů jsou klíčové právě pro chod simulačního jádra, konkrétně pro podmínku, která určuje, zda může, resp. nemůže nastat další iterace. Aby mohla simulace pokračovat, je nezbytné, aby byl neprázdný lokální kalendář událostí a hodnoty všech čítačů byly nenulové.

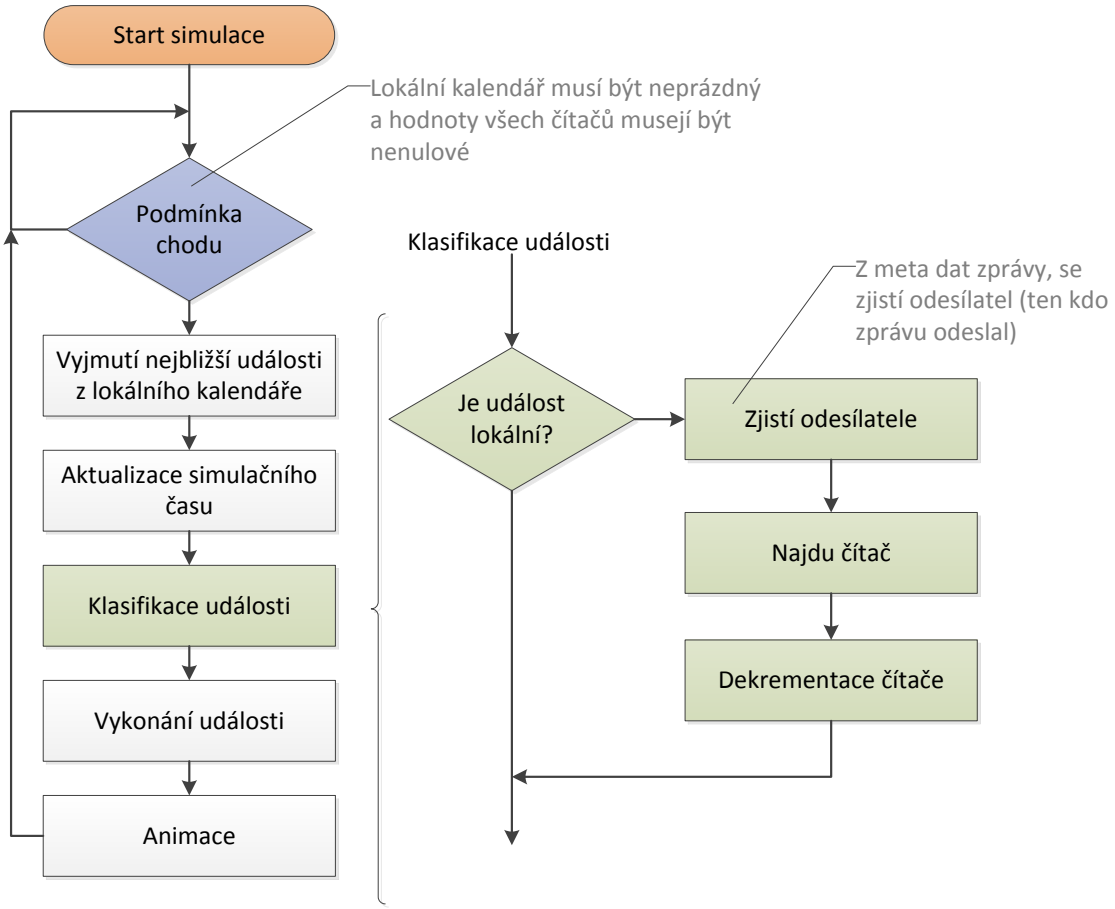
6.1.2 Klasifikace události

Pakliže tato podmínka není splněna – simulace nemůže dále pokračovat a musí čekat, dokud nebude podmínka pravdivá. Simulační jádro je postaveno na metodě diskrétní simulace. Vyjmeme z kalendáře událost s nemenším časovým razítkem a tu následně zpracováváme. Po vykonání události provedeme její klasifikaci. Simulační jádro rozlišuje dva typy událostí:

- lokální – událost vznikla lokálně.
- distribuovaná – událost vznikla na základně podnětu zvenčí.

Při klasifikaci události se právě zjišťuje, o jaký typ události se jedná. V případě, že se jedná o lokální událost, pak se při klasifikaci nic neděje. Zjistíme-li, že se jedná

o distribuovanou událost, pak je zapotřebí aktualizovat hodnoty čítače. Dle odesílatele (strůjce události) najdeme čítač a provedeme jeho dekrementaci.



Obrázek 37 – Průběh jedné iterace
Zdroj: Vlastní zpracování

6.1.3 Zaslání nulové zprávy

V úvodu jsem se zmínil, že koncepce algoritmu vychází z konzervativní synchronizační metody zasílání nulových zpráv na vyžádání. Zatím jsme se ale o žádných nulových zprávách nezmínili.

K odeslání požadavku o nulovou zprávu dochází v okamžiku, kdy je zjištěna nulová hodnota nějakého čítače. Další podmínkou je povolení zasílání nulových zpráv na daný logický proces. V rámci konfigurace algoritmu můžeme explicitně určovat, kterým logickým procesům budeme, resp. nebudeme zasílat nulové zprávy.

V mém případě dochází k odeslání požadavku po dokončení klasifikace. Zkontrolují se všechny hodnoty čítačů, a pokud je u některého naleznuta hodnota nula, pak podmínka chodu simulačního algoritmu je nastavena na false a je odeslán požadavek o zaslání nulové zprávy. Součástí požadavku musí být žadatelův lokální simulační čas. Musíme čekat.

Logický proces, který obdržel požadavek o zaslání nulové zprávy, se podívá na svůj aktuální stav a na základě něj se rozhodne, jak odpoví. V podstatě mohou nastat dvě

situace – buď logický proces odpoví ihned, nebo si naplánuje odpověď do budoucna (jako událost). Při rozhodování hraje roli několik činitelů – čas odesílatele zprávy, aktuální lokální čas, lookahead celého modelu, čas nejbližší události a aktuální kapacita kalendáře.

Je-li prázdný lokální kalendář, pak můžeme bezpečně vrátit odpověď s časem odesílatele zprávy, který navýšíme o hodnotu výhledu.

```
NullMessage.Time = Requester.LocalTime + Lookahead
```

Hodnota výhledu se počítá před zahájením simulace a je vyhodnocena ze všech možných událostí, které mohou v simulačním modelu nastat. Málo kdy se ale stane, že by byl neprázdný kalendář a v této situaci musíme být velmi opatrní, jak odpovíme.

V prvé řadě si spočítáme hodnotu dolní hranice bezpečných časových razítek (LBTS). Tato hodnota vychází ze součtu aktuálního lokálního času a výhledu.

```
LBTS = LocalTime + Lookahead
```

Dalším krokem je naleznutí nejbližší distribuované události. Konkrétně takové události, která komunikuje s jiným logickým procesem (typ Sender). V mém případě může nastat ve stejný simulační čas mnoho událostí – z podstaty zápisu distribuovaného modelu. Proto zjišťujeme, jestli už náhodou nemáme naplánovanou odpověď. Pakliže máme naplánovanou odpověď, tak prověříme časy.

```
ScheduleEvent immediateScheduleSender = findImmediateSender();
if (LBTS > immediateScheduleSender .Time) {
      LBTS = immediateScheduleSender.Time;
}
```

Může nastat situace, že hodnota LBTS bude menší, než čas odesílatele. Takovou hodnotu nemá smysl zasílat, a proto naplánujeme pouze událost do kalendáře. Událost bude naplánována na čas, který zaslal žadatel – v okamžiku zpracování takové události se odešle nulová zpráva.

```
if (Requester.LocalTime > LBTS) {
    plan(Requester.LocalTime, new NullModuleSender(Requester));
    return;
}
```

Pokud jsme se dostali až sem, pak zbývá poslední krok – odeslat nulovou zprávu zpět žadateli. Časové razítko zprávy bude hodnota LBTS.

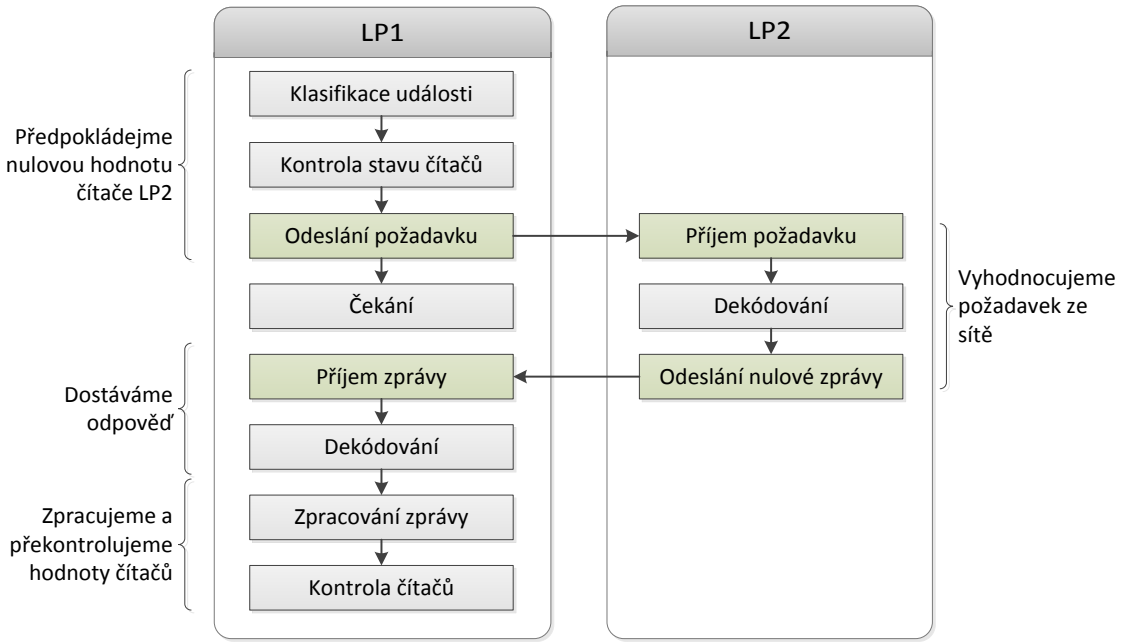
```
sendNullMessage(LBTS, this.ModelName);
```

Žadatel příjme zprávu a může případně dál pokračovat ve svém simulačním výpočtu. Kompletní pseudokód, který jsem nyní popisoval, je obsažen v příloze. Popisované ukázky jsou odstíněny od komunikační logiky po něco dalšího.

Poslední bezpečná zpráva také určuje hodnotu tzv. bezpečného času. Mnoho událostí pouze mění stav a samy o sobě simulační čas neposouvají dopředu. Pakliže je simulační

algoritmus pozastaven z důvodu nulové hodnoty některé z čítačů, pak jsou ještě simulovány všechny události, které mají hodnotu časového razítka rovno hodnotě bezpečného simulačního času.

Shrňme si v malém schématu (Obrázek 38), co se vlastně v kódu přesně děje. K inkrementaci čítačů dochází ve zpracování zprávy (viz Obrázek 37).



Obrázek 38 – Zasílání požadavku o nulovou zprávu
Zdroj: Vlastní zpracování

Přijímání zpráv ze sítě se provádí paralelně vedle chodu samotného simulačního jádra. Při přijmutí zprávy může být právě zpracovávána nějaká lokální událost apod. Je důležité si toto uvědomit a při vyhodnocování požadavku musíme být velmi obezřetní. Přistupujeme ke společným prostředkům (simulační čas, kalendář, …), a proto musíme citlivé části kódu zabezpečit synchronizací.

6.2 Komponenty simulačního jádra

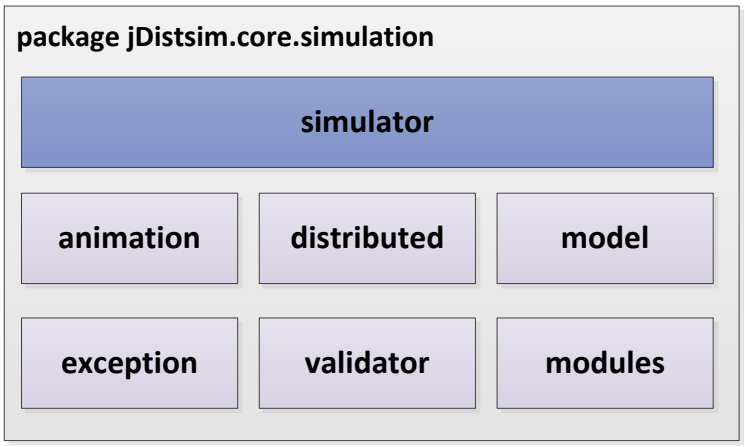
Doposud jsme si popisovali zejména logiku samotného simulačního algoritmu. Celé simulační jádro je ale daleko komplexnější a skládá se z několika komponent. Stěžejní komponenty jádra:

- **BaseSimulator** – hlavní motor celé simulace, zpravidla řídí ostatní komponenty.
- **DistributedSimulator** – obohacení komponenty BaseSimulator o část, které zastřešuje distribuovanou simulaci (komunikace, klasifikace, výpočet výhledu, …).
- **SimulatorEnvironment** – udržuje aktuální stavové prostředí simulace. Na základě této komponenty pak můžeme získávat statistiky a sledovat průběžné změny v simulačním procesu (například čas, počet entit, apod.).

- **SimulatorOutput** – simulátor si vede svůj vlastní log činnosti. Můžeme se k němu přihlásit a sledovat, co simulátor přesně provádí.
- **SimulatorModelValidator**– před samotným spuštěním simulačního algoritmu se provádí základní validace[16] modelu. Odhalí se chyby, které mohly vzniknout při návrhu modelu.
- **SimulatorAnimator** – jádro disponuje základním prostředkem pro vytvoření simulační podpory chodu simulačního algoritmu.
- **Communication** – komponenta starající se o komunikaci s ostatními logickými procesy v síti.
- **SimulatorRunner** – slouží ke spouštění simulace.

Těchto sedm komponent má na starost bezproblémový chod simulačního jádra a mimo jiné umožňují uživateli vše v reálném čase sledovat. Nutno dodat, že každá komponenta je navržena tak, aby byla konfigurovatelná a připravená pro případné další rozšíření.

Podíváme-li se na jádro ještě více z pohledu implementace – zjistíme, že struktura je relativně intuitivní a členěná do několika balíčků. Celé simulační jádro, které pohání mou aplikaci, se nachází v balíčku `jDistsim.core.simulation` a dále je členěno do sedmi hlavních balíčků a několika dalších zanořených balíčků:



Obrázek 39 – Kořenové uspořádání hlavních balíčku jádra
Zdroj: Vlastní zpracování

Stěžejním balíčkem je `jDistsim.core.simulation.simulator`, který obsahuje všechny hlavní komponenty a další podpůrné třídy, se kterými nejčastěji pracujeme. Celá paleta modulů, včetně jejich Factory tříd a grafiky nalezneme v balíčku `modules`. Aby bylo snadné identifikovat případný problém, který může nastat během simulace, jsem zavedl několik specifických výjimek. Tyto výjimky jsou obsaženy v balíčku `exception`.

[16] V této práci je význam validace používám ve smyslu formální kontroly simulačních modelů z hlediska návrhu. Nejedná se o klasickou validaci simulační modelů, která se používá v simulacích.

6.3 Spuštění a vytvoření simulačního modelu

Hlavní vlastností simulačního jádra je samozřejmě samotný proces distribuované simulace. Pokusím se nastínit cestu, jak postavit distribuovaný simulační model a následně spustit simulační proces. Nyní půjdeme cestou kódu – ukážeme si, jak se pracuje s knihovnou na úrovni v kódu. Navrhování a spouštění modelu pomocí vývojového prostředí si popíšeme v další kapitole.

V prvé řadě je nutné navrhnout model, který budeme modelovat. Jako ukázka postačí jednoduchý model, který nebude mít příliš složitou logiku (Obrázek 40).

Obrázek 40 – Jednoduchý distribuovaný simulační model
Zdroj: Vlastní zpracování

Logika distribuovaného modelu je prostá – vytvoříme entitu (modul create_1), kterou následně časově pozdržíme (delay_1). Po uplynutí tohoto času odešleme entitu na jiný distribuovaný model, který je někde v síti.

6.3.1 Vytvoření simulační modelu

Model, který vytvoříme v kódu, musí přesně odpovídat naší představě (Obrázek 40). Simulátor očekává, že dostane objekt typu `ISimulationModel`. Jedná se o rozhraní, které definuje, jak má vypadat simulační model. Nemusíme vytvářet vlastní implementaci. V balíčku `jDistsim.core.simulation.model`, je obsažena hotová implementace tohoto rozhraní, kterou můžeme použít. Od simulačního modelu toho moc potřeba není – v podstatě se jedná pouze o schránku, která zastřešuje všechny moduly. Třída `SimulationModel` disponujeme několika operacemi, které nám usnadní práci se simulačním modelem (například získání kořenových modulů, získání distribuovaných modulů, apod.).

Abychom mohli sestavit simulační model – musíme mít připravené moduly. Připravíme si proto patřičné moduly. Na základě naší představy o modelu (Obrázek 40) vytvoříme mezi moduly závislosti. Následně vložíme moduly do seznamu a vytvoříme simulační model. Každý simulační model musí mít nějaké jméno – proto připojujeme i název.

```
private static ISimulationModel buildModel() {
    Module create = new Create(new RootSettings("create_1"), true);
    Module delay = new Delay(new DelaySettings("delay_1"), true);
    Module sender = new Sender(new SenderSettings("sender_1"), true);

    create.addOutputDependency(delay);
    delay.addOutputDependency(sender);

    List<Module> modules = Arrays.asList(create, delay, sender);
    ISimulationModel model = new SimulationModel(modules, "model1");
    return model;
}
```

Z důvodu zjednodušení ukázky se zde nevěnuji konfiguraci samotných modulů. Je nutné si uvědomit, že moduly musí mít i konfiguraci. Například modul sender musí vědět, na jaký logický proces entitu zaslat. Všimněte si, že v konstruktoru modulu je nastavení očekáváno. Bez něj nelze modul sestavit. Pokud učiníte jako já, pak bude mít modul nastavené implicitní hodnoty, které jsou pro každé nastavení různé. Předpokládejme, že máme nastavené moduly správně.

6.3.2 Příprava simulátoru a spuštění distribuované simulace

Dalším krokem je připravit simulátor a spuštění simulace. Mimo simulačního modelu je zapotřebí ještě dalších objektů. Potřebujeme validátor modelu, abychom jej mohli verifikovat z hlediska správnosti návrhu. Validátor je opět specifikován pouze rozhraním. Jádro ale opět obsahuje již hotovou implementaci, kterou můžeme bez problému použít. Dále potřebujeme lokální síťovou konfiguraci. Konfigurace se skládá z názvu a portu, na kterém lokálně simulátor poběží. Pod touto konfigurací se mohou ostatní distribuované modely s tímto spojit.

Máme vše připravené a můžeme simulaci spustit. Vytvoříme instanci simulátoru. Dále si připravíme `SimulatorRunner`, který simulaci spouští separátně v samostatném vlákně.

```
public static void main(String[] args) {
    ISimulationModel model = buildModel();

    ISimulationModelValidator validator = new SimulationModelValidator();
    LocalNetworkSettings networkSettings = getLocalNetwork();

    ISimulator simulator = new DistributedSimulator(
                                        validator, networkSettings);

    SimulatorRunner runner = new SimulatorRunner(simulator, model);
    runner.start();
}
```

Metodou `start()` se spustí simulační výpočet. Přihlásíme-li se k výstupu jádra, pak můžeme sledovat chod celé simulace. Z pohledu implementace bylo při návrhu dbáno na objektově orientovaný přístup a pravidla čistého kódu. Proto by mělo být používání jednotlivých tříd velmi intuitivní a díky stylu programování proti rozhraní vzniká i případná možnost základní implementace vyměnit za jiné.

6.4 Počátek simulace

Co se vlastně stane v okamžiku, kdy spustíme simulátor? Před samotným spuštěním simulačního algoritmu musíme ještě provést několik operací:

1. **Příprava všech komponent jádra** – nastavení všech potřebných komponent (kalendář, animační podpora, sestavení simulační podmínky konce, …).
2. **Inicializace výstupu simulátoru** – příprava výstupních interpretů (render).

3. **Inicializace stavového prostředí simulátoru** – nastavení defaultních hodnot celého stavového prostředí jádra (například lokální čas).

4. **Nastavení simulačního modelu** – simulátoru je přiřazen konkrétní simulační model.

5. **Validace simulačního modelu** – verifikace simulačního modelu.

6. **Roztřídění modulů** – hledání sémantiky v simulačním modelu (kořenové moduly, distribuované moduly, …).

7. **Hledání síťových závislostí** – dle použitých distribuovaných modulů jsou získány síťové adresy závislých vzdálených logických procesů (simulátorů).

8. **Nastartování lokální komunikace** – zaregistrování lokálního simulátoru (logického procesu) pod definovaným názvem na určitý port.

9. **Připojení k vzdáleným simulátorům** – nastartování komunikace s nejbližšími logickými procesy.

10. **Autorizace u vzdálených simulátorů** – u každého vzdáleného simulátoru je zapotřebí se autorizovat.

11. **Čekání na stav připravenosti** – musíme čekat, až budou všechny logické procesy v síti připraveny k zahájení simulace.

12. **Inicializace modulů** – každý modul se musí připravit pro zahájení simulace (nastavení defaultních hodnot).

13. **Inicializace simulátoru** – naplánování prvních událostí (dle kořenových modulů).

14. **Spuštění simulace** - zahájení simulačního algoritmu.

Všechny operace, které jsou popsané výše, probíhají v uvedeném pořadí před zahájením simulace. Simulátor ukončí svou činnost vždy, když narazí na chybu v některé z operací – dál nepokračuje. Může se jednat například o zjištění neočekáváného stavu nebo špatný model, chybu v komunikaci a další. Operace 7, 8, 9, 10 a 11 se ignorují, pakliže se nejedná o distribuovaný simulační model.

Simulační algoritmus byl popsán v kapitole 3.3.2.1. V jedné simulační iteraci se navíc provádí dále aktualizace stavového prostředí, zaslání dat na výstup, nastavení aktuálního stavu entity, animační aktivita a počítání statistik.

6.5 Entita

Zápis simulačního modelu zachycuje životní cyklus entity (viz kapitola 5). Výše bylo zmíněno, že v jedné iteraci dochází k nastavení aktuálního stavu entity. Co to znamená?

Entita představuje nositele atributů, které můžeme definovat přímo v modelu. Některé atributy jsou ale definovány přímo jádrem.

Atributy, které jsou definované jádrem, nemůžeme ovlivňovat. Všem entitám je při vzniku přiřazen jejich tvůrce (modul), unikátní název a název ikony. Důležitý je zejména název entity. Ten musí být v rámci celé sítě logických procesů vždy unikátní. Nesmí se stát, aby se v síti střetly entity se stejným názvem. Jak správně zvolit název entity, aby byl vždy unikátní?

Entita žije se dvěma názvy, se kterými lze pracovat:

- úplný název,
- zkrácený název.

Simulátor pracuje s úplným názvem a uživatel pracuje se zkráceným názvem. Zkrácený název je zaveden pouze z důvodu příjemnější interpretace entity uživateli. Název entity se sestavuje na základě tohoto pravidla:

```
FullEntityName  = FullModelName.Creator.EntityIdentifier
ShortEntityName = EntityIdentifier;
FullModelName   = ModelName_Port
```

Příklad entity:

```
FullEntityName = highway_1099.create_1.car1
ShortEntityName = car1;
```

Při uběhnutí každé iterace je entitě zapsán do atributů i modul, který ji aktuálně zpracovával. Máme tak možnost sledovat kompletní historii entity. Může se stát, že entita není vytvořena lokálně, ale dostane se k nám z jiného logického procesu. V takovém případě je opět tato informace zapsána mezi atributy.

6.6 Vzdálený logický proces

Se vzdálenými logickými procesy provádíme interakce prostřednictvím modulů Sender a Receiver. Musíme ale vědět, s kým vlastně chceme komunikovat. Vzdálené moduly definujeme třídou `DistributedModelDefinition`. Tato třída slouží pouze pro definici informací o logickém procesu.

Všechny položky jsou povinné:

- ModelName – lokální název (tento název vidíme ve statistikách).
- RemoteModelName – název, pod kterým je model zaregistrován v síti.
- Address – síťová adresa, kde model běží.
- Port – port, na kterém je model zaregistrován.

- LookaheadRequest – povolení, zda můžeme zasílat požadavek o výhled.
- Receive – zda můžeme od logického procesu přijímat entity.

6.7 Verifikace modelu

Jedna z operací, která je prováděna před spuštěním simulace, je verifikace modelu. Dochází ke kontrole navrženého simulačního modelu. Víme, že model je složen pouze z modulů. Při verifikaci je každý modul náležitě zkontrolován. Z povahy věci musí každý modul sám o sobě vědět, kdy je pro simulaci validní a kdy není. Iterujeme všemi moduly a kontrolujeme jejich validitu.

Mnoho vlastností modulu lze kontrolovat abstraktně u všech totožně. Například unikátnost názvu a vstupně/výstupní zavilosti. Definované vstupy a výstupy musejí být vždy nastaveny dle specifikace v nastavení. Dále má každý modul vlastní validační mechanismus dle své funkce. Například u modulu Delay kontrolujeme správnost zadaného času, na druhou stranu u modulu Sender hlídá správnost nastavení vzdáleného modulu.

O verifikaci modelu se stará validátor. Při validaci se vždy zkontrolují všechny moduly. Při kontrole jsou veškeré chybné stavy a informace o nich akumulovány do speciálního seznamu. Validátor při ukončení do tohoto seznamu nahlédne a jeli neprázdný, pak simulátor ukončí svou činnost a všechny informace o chybách jsou předány uživateli.

6.8 Komunikace s logickým procesem, autorizace a řešení závislostí

Pojďme se blíže podívat na průběh komunikace mezi logickými procesy. Logické procesy mezi sebou komunikují prostřednictvím technologie Java RMI. Je tedy nezbytné, abychom definovali rozhraní a každý logický proces skrze něj vystupoval. Rozhraní vypadá následovně:

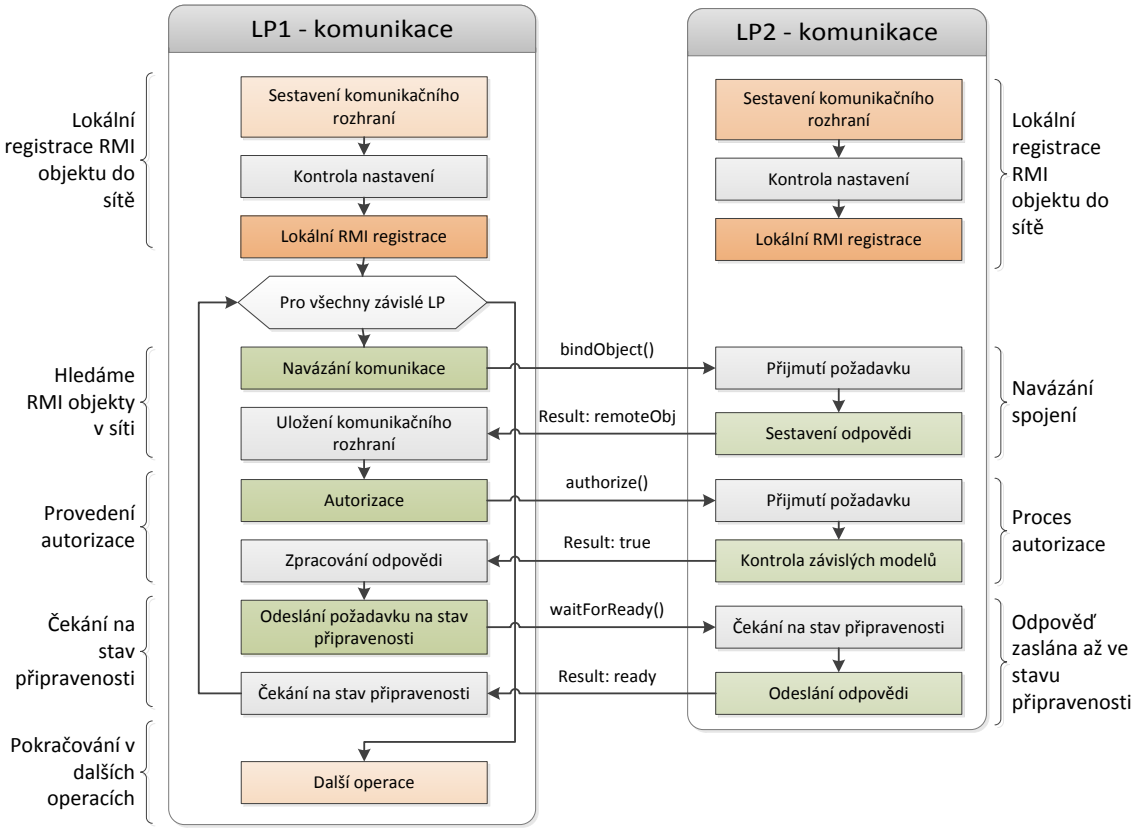
```
public interface IRemote extends Remote {
    // požadavek o autorizace
    public boolean authorize(String requesterModelName) throws RemoteException;
    // požadavek o získání výhledu (nulové zprávy)
    public void getLookahead(double requesterTime, String requester)
                                                    throws RemoteException;
    // požadavek na zpracování entity
    public void process(double time, Entity entity, String requester)
                                                    throws RemoteException;
    // zaslání nulové zprávy
    public void processNullModule(double time, String requester)
                                                    throws RemoteException;
    // LP neodpoví, dokud není připraven simulovat
    public void waitForReady() throws RemoteException;
}
```

Deklarujeme šest operací, o které můžeme žádat. V prvé řadě požadavek o autorizaci. Všechny logické procesy vědí, od koho chtějí dostávat zprávy (tyto informace se drží v tabulce). Požádá-li nás o autorizaci proces, který v této tabulce není – odpovíme neúspěchem a s takovým procesem nebudeme komunikovat (zakážeme mu používání

našich služeb). Metoda `getLookahed` je určena k vyslání požadavku o nulovou zprávu. Nulovou zprávu odesíláme metodou `processNullModule`. Odeslání entity na vzdálený logický proces provádíme operací `process`.

Rozhraní implementuje třída `RmiRemote`, se kterou pracujeme. O vystavení lokálního logického procesu na síť se stará třída `Communication`. K ostatním logickým procesům se připojujeme pomocí třídy `RmiBinder`. Tato třída vyhledává v síti požadovaný logický proces. Třída pracuje chytře a v případě neúspěchu se snaží o připojení opakovaně, dokud nedosáhne na TimeOut. Všechny třídy, které slouží výhradně ke komunikaci, jsou obsaženy v balíčku `jDistsim.core.simulation.distributed.communication`.

Pojďme si celý průběh komunikace, který se odehrává před samotným simulačním algoritmem, lépe přiblížit prostřednictvím diagramu (Obrázek 41):



Obrázek 41 – Průběh komunikace
Zdroj: Vlastní zpracování

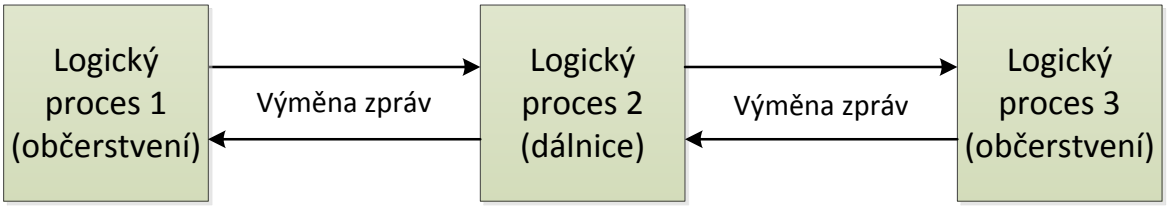
Z diagramu je patrné, že v prvé řadě je vždy nutné nejdříve provést lokální registraci. Nemůžeme komunikovat, pokud nemáme model registrován v lokálním RMI registru. Dále musíme navázat spojení se všemi logickými procesy, se kterými jsme přímo v kontaktu. Navážeme spojení a ihned se u vzdáleného logického procesu autorizujeme. Poté zasíláme požadavek, na který nám vzdálený logický proces odpovídá až v okamžiku, kdy je připraven.

V dalších operacích, které souvisejí se startem simulačního algoritmu, můžeme pokračovat až v okamžiku, kdy úspěšně navážeme komunikaci se všemi logickými procesy. Samozřejmě je nutné se zamyslet i nad situací, kdy něco neproběhne hladce – například nejsme autorizování. V takovém případě ukončíme spojení, se všemi logickými procesy, se kterými se nám podařilo již navázat spojení, a ukončíme činnost jádra.

6.8.1 Čekání na závislosti – stav připravenosti

Při popisu komunikace jsem mnohokrát zmiňoval stav připravenosti logického procesu. Stavu připravenosti dosáhne logický proces v okamžiku, kdy má připojené všechny závislosti.

Uvažme příklad tří logických procesů. Jako v předešlé kapitole, použijeme i zde příklad s dálnicí a dvěma rychlými občerstveními.



Obrázek 42 – Závislosti tří logických procesů
Zdroj: Vlastní zpracování

Máme tři logické procesy, které jsou mezi sebou v přímé interakci. Přímá závislost ale není mezi všema stejná:

- LP1 – závislé na LP2,
- LP2 – závislé na LP1 a LP3,
- LP3 – závislé na LP3.

Z výčtů závislostí si můžete povšimnout, že například LP1 vůbec nemusí vědět, že existuje LP3. Zajímá ho pouze LP2, který se nachází v jeho bezprostředním okolí.

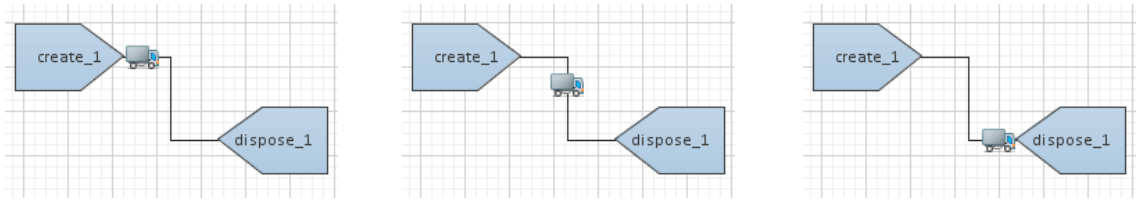
Jádro nemůže pokračovat, dokud nemá úspěšně navázanou komunikaci se všemi závislostmi. Do stavu připravenosti se dostáváme právě tehdy, když máme všechny závislosti připojené. Komunikace je jednosměrná. Pokud chceme komunikovat oběma směry, pak je zapotřebí navázat komunikaci z obou stran. LP1 naváže spojení s LP2 a zároveň LP2 naváže spojení s LP1.

Spustíme LP1. Ten se následně zaregistruje do sítě a začne vyhledávat své závislosti. LP1 se snaží najít LP2, který zatím do sítě připojen není. Spustíme tedy LP2. LP2 se zaregistruje a začne hledat také své závislosti. Mezi tím LP1 zjišťuje, že je LP2 online a snaží se s ním navázat komunikaci. LP1 se spojí s LP2 a posílá požadavek o stav připravenosti – LP2 ještě není připraven (nemá připojené LP3) a tak LP1 musí čekat. LP2 navazuje komunikace s LP1. LP2 také zasílá požadavek o zjištění stavu připravenosti LP1.

LP1 odpovídá, že je připraven – má všechny závislé logické procesy připojené. Spouštíme LP3. LP3 se zaregistruje do sítě a naváže komunikaci s LP2. LP2 mezi tím také navázal spojení s LP3. LP2 má připojené všechny závislosti. LP1 a LP3 se tuto informaci dozvídají. Všechny logické procesy jsou připraveny a simulace může započít svůj výpočet.

6.9 Animace

Jádro nabízí i podporu pro tvorbu animování simulačních aktivit. Animátor v mé práci je navržen odlišně, než tradiční animátor v diskrétní simulaci. Nepracuji s časovým kvantem a registrem simulačních aktivit. Animátor jDistsim je pouze jakousi pseudoanimací, která zachycuje přechody, mezi právě zpracovávanými moduly. Simulační aktivity neprobíhají souběžné, ale jedna podruhé sekvenčně. V jeden okamžik vidíme vždy pouze jednu simulační aktivitu (Obrázek 43). Během animace je činnost simulačního výpočtu potlačena.



Obrázek 43 – Průběh animace
Zdroj: Vlastní zpracování

V jádře se nachází pouze deklarace animátoru formou rozhraní. Rozhraní se nazývá `ISimulationAnimator` a nalezneme jej v balíčku `jDistsim.core.simulation.animation`. Implementace animátoru, se kterou pracuje vývojové prostředí, se nachází mimo hlavní jádro. Je to z důvodu zachování určitého stupně abstrakce jádra. Skutečný animátor již pracuje s konkrétním grafickým výstupem a sám si renderuje scénu podle potřeby.

6.10 Výstup z jádra

Jádro disponuje vlastním logovacím systémem. Tento systém byl zaveden z důvodu ladění a také možnosti uživateli nahlédnout do činnosti jádra. Je snaha maximálně uživatele informovat a aktuální činnosti. Sledování výstupu je flexibilní. V podstatě postačí implementovat rozhraní `SimulatorWriter` a přihlásit k výstupu prostřednictvím simulátoru:

```
simulator.getOutput().getSimulatorWriters().add(new SimulatorWriter() {
    @Override
    public void write(String text) {
        System.out.println(text);
    }
    @Override
    public void clear() {
        System.out.println("clear");
    }
});
```

Ukázka výše zachycuje rychlé přihlášení ke sledování výstupu, který se bude vypisovat na standardní Java výstup. Výstup je realizován pouze formou textových zpráv. Ukázka logu, který zachycuje chod simulačního výpočtu:

```
Local time: 2.0 Process module create_1
Local time: 2.0 Process module dispose_1 {entity: entity_13}
Local time: 3.0 Process module create_1
Local time: 3.0 Process module dispose_1 {entity: entity_14}
Local time: 4.0 Process module create_1
```

Rozsáhlejší ukázka komplexního výstupního logu z jádra je obsažena v příloze (Příloha C – výstupní log z jádra). Mimo simulačního výpočtu jsou v logu zachyceny i ostatní aktivity jádra – kupříkladu inicializace, příprava, chybové stavy nebo síťová komunikace.

7 Distribuovaná simulace bez použití deklarativního přístupu

První etapou během vývoje distribuovaného simulačního jádra bylo testování algoritmu zasílání nulových zpráv na vyžádání bez použití deklarativního přístupu. První implementace jádra byla odlišná než ta, kterou jsme si popisovali v předešlé kapitole. Nebyla tak komplexní a nepracovalo se ještě s žádnými moduly. Jádro pracovalo se standardními událostmi, které bylo nutné dopředu naprogramovat.

Jádro je v mnoha směrech podobné tomu, které jsem popisoval v předešlé kapitole. Vychází z algoritmu zasílání nulových zpráv na vyžádání a zanáší stejné prvky, které jsme si popisovali (čítače, klasifikace události, Java RMI apod.). Z hlediska implementace je ale odlišně postavené. Neskládá se z žádných komponent, a vytváření simulačního modelu provádíme odlišným způsobem.

Experiment s distribuovanou simulací se snažil realizovat příklad s dálnicí a přilehlými občerstveními, který jsme si představili v kapitole 5.4.2. Distribuovaná simulace se skládá ze tří logických procesů.

7.1 Události

V tomto experimentu se nepracovalo s žádnými modely a bylo zapotřebí všechny události, které se v modelu vyskytují, naprogramovat. Události se vytvářejí odvozením od bázové třídy `Event`. Na nás je implementovat metodu `execute`, ve které implementujeme kompletní logiku události. Parametrem metody je simulátor, který můžeme využívat pro plánování dalších událostí. Mimo jiné můžeme registrovat simulační aktivity. Události nevytváříme ručně, ale je zapotřebí o vytvoření požádat:

```
IEvent event = simulation.getConfiguration().getEventDescriptor()
                    .Resolve("IHighwayDepartureEvent", entity);
```

Simulátor musí mít všechny typy události před započetím simulačního výpočtu zaregistrované v deskriptoru. Totéž platí i pro případné simulační aktivity. Registraci událostí a animací provádíme přímo nad simulátorem:

```
simulator.getConfiguration().getEventDescriptor()
.Register("IHighwayArrivalEvent", HighwayArrivalEvent.class);

simulator.getConfiguration().getAnimationDescriptor()
.Register("HighwayArrivalAnimation", HighwayArrivalAnimation.class, canvas);
```

Při vytváření událostí se simulátor postará i o předání entity.

7.2 Práce se simulátorem a vytvoření simulačního modelu

Přiblížíme si práci se simulátorem a jak vytvořit simulační model. Pro tento simulátor není potřeba simulační model, simulátor s ním přímo nepracuje. Není to potřeba. Pouze se nadefinují první události, které budou mít za úkol nastartovat simulační výpočet. Samotné

události si poté plánují další události. Simulátor pouze ví, jaké události bude zpracovávat, ale už netuší, jak jsou události mezi sebou provázané.

Jak by vypadala implementace například logického procesu dálnice? Vytvoříme si například třídu `Highway`, kterou odvodíme od třídy `DistributedSimulation`. Naším úkolem je implementovat tří metody:

```
public class Highway extends DistributedSimulation implements IHighway {
    public Highway(String identifier, IDistributedSender sender) {
        super(identifier, sender);
    }

    @Override
    public void initialize() { }

    @Override
    public double getLookahead() { }

    @Override
    protected boolean endCondition() { }
}
```

V metodě `initialize` naplánujeme první události, které zahájí simulační výpočet (například příjezd vozidel). Dále musíme určit způsob, jak bude získávána hodnota výhledu. Do těla poslední metody napíšeme podmínku, na základě které se ukončí simulační výpočet.

Po dokončení implementace třídy `Highway` zbývá již poslední krok. Musíme provést ještě konfiguraci, která se skládá z několika kroků.

```
DistributedSimulation highway = new Highway("Highway", new RmiSender());
// registrace událostí
...
// registrace animačních aktivit
...

// binding dálnice do sítě
IDistributedModelContainer container = new RmiModelContainer(highway);
registry.rebind("Highway", container);

// registrace vzdálených objektů
highway.registerDistributedSimulationModel("FastFood-Right", ...);
highway.registerDistributedSimulationModel("FastFood-Left", ...);

// spuštění simulace
highway.start();
```

V případě implementace pravého a rychlého občerstvení postupujeme totožně.

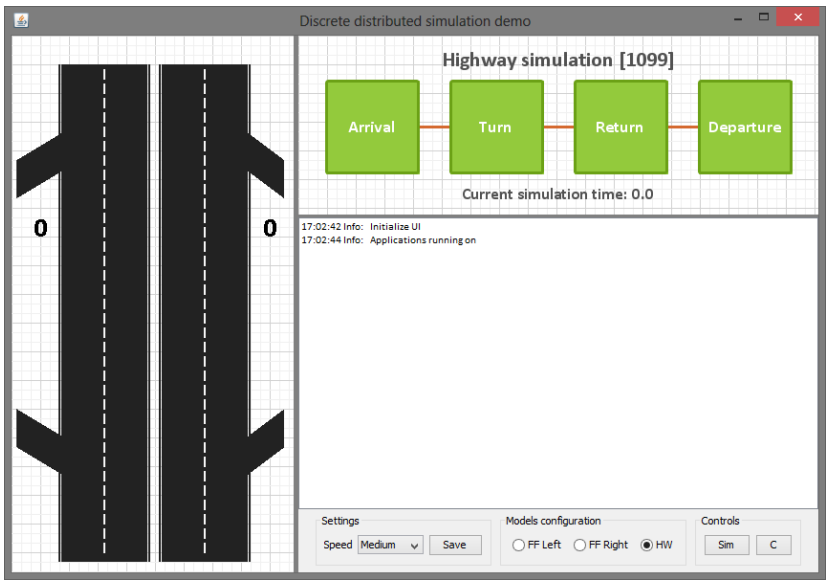
7.3 Animační jádro

Další výraznou změnou je přístup k animační podpoře. Popisované jádro v kapitole 6 pracuje pouze s přechody mezi moduly – ty jsou animovány. V případě tohoto experimentu

je snaha, aby animační jádro zobrazovalo průběh animace v reálném čase. Z toho důvodu registrujeme animační aktivity (během vykonávání události). Během simulace pak tyto aktivity dostávají časové kvantum, které určuje, jak dlouho budou aktuálně animovány.

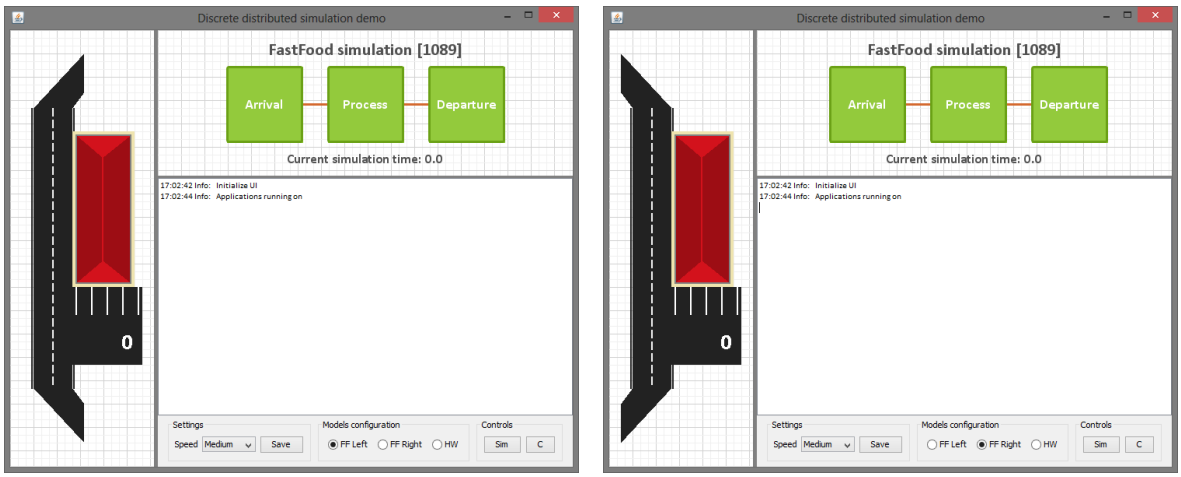
7.4 Grafická vizualizace

K simulačnímu jádru byla vytvořena i grafická nadstavba, aby byla názorně vizualizována distribuovaná simulace. Grafická nadstavba je standardní GUI aplikace, která obsahuje tři grafické scény: dálnici (Obrázek 44) a dvě přilehlá občerstvení (Obrázek 45).



Obrázek 44 – Scéna dálnice
Zdroj: Vlastní zpracování

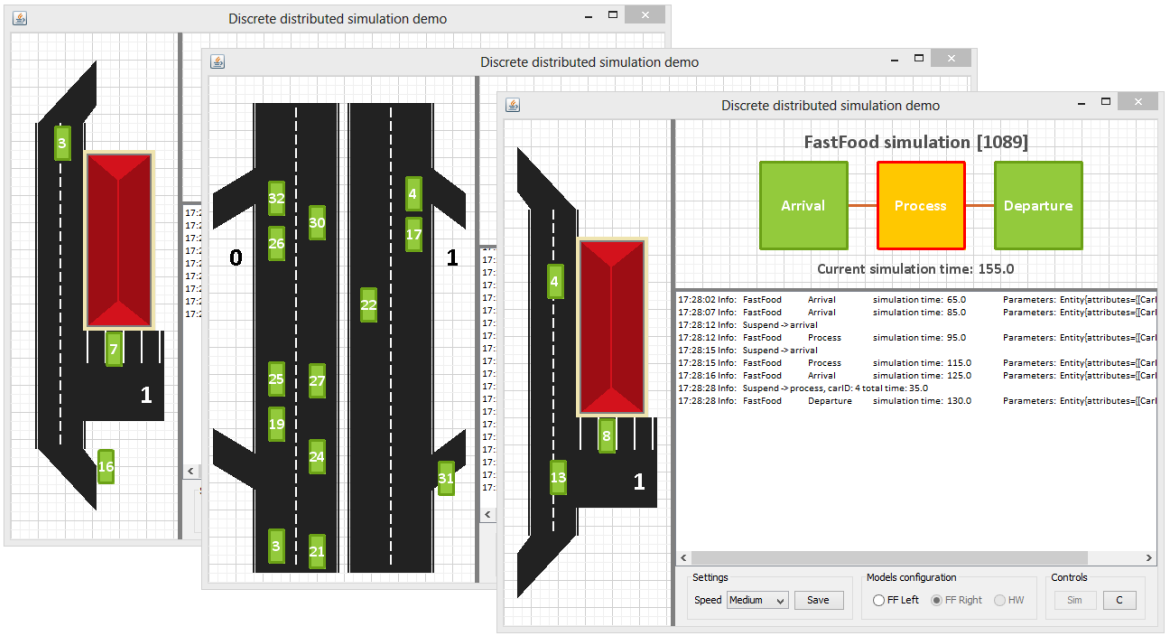
Vizuální scény přepínáme přímo v aplikaci. Dále aplikace nabízí možnost základní konfigurace síťového nastavení (adresa a port vzdálených logických procesů).



Obrázek 45 – Scéna rychlého občerstvení (lidské a pravé)
Zdroj: Vlastní zpracování

7.5 Spuštění distribuované simulace

Aplikace byla testována v distribuovaném prostředí (tři různé počítače propojené v síti). Na obrázku níže (Obrázek 46) je zachycen průběh distribuované simulace, který běží v lokálním prostředí – na jednom počítači. Pro účely experimentu to bylo dostačující.



Obrázek 46 – Distribuovaná simulace v praxi
Zdroj: Vlastní zpracování

Na obrázku je i patrný průběh animace. Po dálnici se pohybují vozidla a některá se přesouvají do vedlejších rychlých občerstvení.

7.6 Shrnutí experimentu

Experiment dopadl úspěšně. Měl jsem možnost otestovat svou myšlenku implementace jádra distribuované simulace. Pro tvorbu simulačního modelu nebyl kladen takový důraz na abstrakci. Události si implementujeme sami a máme možnost popsat i velmi složitou logiku. V případě deklarativního přístupu jsme odkázaní na paletu dostupných modulů. Pro potřeby plánovaného vývojového prostředí nebyla vzniklá knihovna v rámci tohoto experimentu dostačující. Je zapotřebí daleko větší abstrakce při budování simulačního modelu a přidání dalších funkcionalit. Po dokončení tohoto experimentu jsem jádro kompletně re-implementoval do podoby, kterou jsem popisoval v předešlé kapitole.

8 Vývojové prostředí jDistsim IDE

8.1 Úvod do aplikace

Při úvaze nad aplikací, která by umožňovala jednoduše vytvářet distribuované simulační modely, nebylo lehké něco vymyslet. Nakonec jsem došel k závěru, že nejlepší bude cesta té největší abstrakce – vytvořit vývojové prostředí. Nebude v kódu dopředu předpřipraven žádný konkrétní simulační model. Vše bude v uživatelově moci. Uživatel dostane možnost si nakonfigurovat takřka libovolný distribuovaný simulační model.

Vytvořit vývojové prostředí není lehký úkol. Při implementaci narazíme na spousty problémů, které se nám postaví do cesty. Například architektura, komunikace, volba abstrakce, uživatelské rozhraní a další. Jedná se o značně rozsáhlý typ aplikace, kde je zapotřebí mezi sebou provázat mnoho částí.

Název jDistsim IDE vychází ze slov Java DISTributed SIMulation Integrated Development Environment.

8.2 Požadavky

Na úvod se seznámíme s požadavky, které jsem si vymezil před implementací aplikace. Hlavními požadavky pro vývojové prostředí byly:

- jednoduché uživatelské rozhraní,
- deklarativní způsob zápisu modelu,
- spuštění simulačního modelu v rámci distribuované sítě simulačních modelů,
- verifikace modelů,
- rozšiřitelnost a stabilita aplikace,
- při budování modelu poskytnou uživateli nápovědu,
- zobrazení maxima informací o tvořeném simulační modelu,
- decentralizovaný způsob distribuované simulace,
- možnost uložení a načtení simulačního modelu ve formátu XML,
- použití platformy Java a pro komunikaci využít technologii Java RMI.

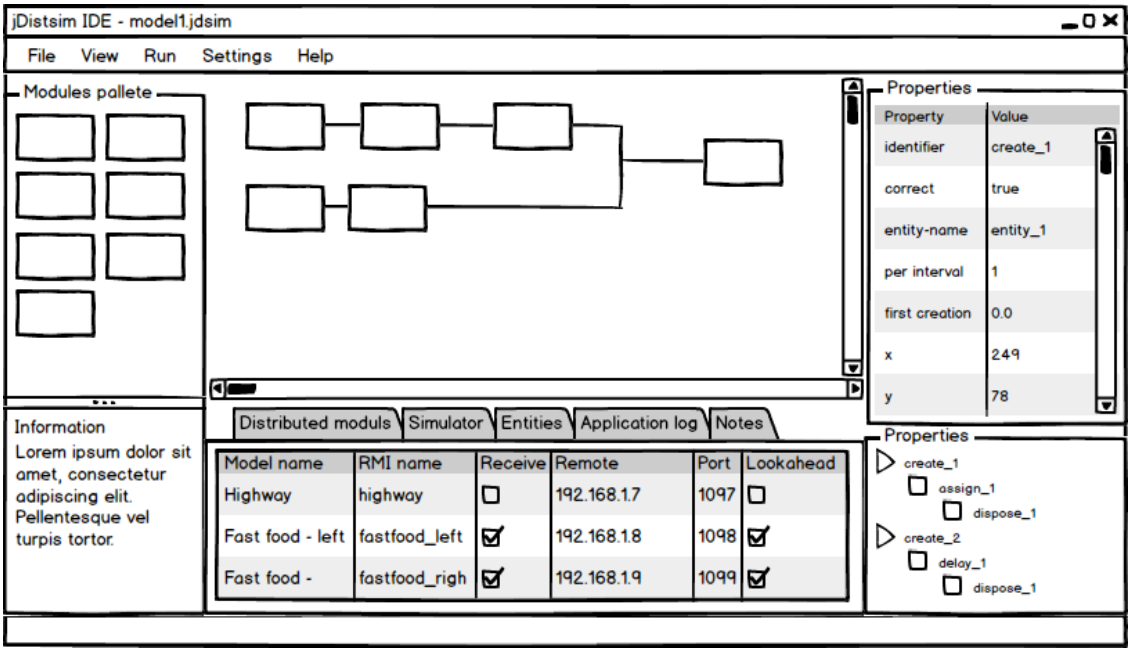
Těchto několik požadavků a spousty dalších menších, jsem si kladl za cíl při návrhu aplikace. Vývojové prostředí je v mnoha aspektech inspirováno aplikací Arena od firmy Rockwell Automation. Arena je určená zejména pro návrh simulačních modelů diskrétní simulace.

Platforma jDistsim je postavena na principu decentralizovaného řízení. Každá instance vývojového prostředí je svým způsobem server a zároveň může být pro ostatní i klientem. Uživatel si ve vývojovém prostředí navrhne kompletní simulační model. Není zde podmínka, že by model musel být nutně součástí distribuované simulace. Lze si vytvořit i takový model, který bude pouze lokální a nebude ke svému chodu potřebovat žádnou interakci s jinými modely. V aplikaci je ale umožněno do modelu přidávat i distribuované moduly (viz kapitola 5.3.3). Tyto moduly jsou schopné zajistit s ostatními modely v síti komunikaci. Při návrhu distribuovaného modelu uživatel pouze vytyčí modely, se kterými chce komunikovat. Sám je netvoří. Po návrhu simulačního modelu je možné spustit simulaci. V případě distribuovaného modelu se automaticky začne navazovat komunikace se všemi ostatními simulačními modely v síti. Simulační výpočet se spustí až tehdy, kdy je vše v naprostém pořádku.

8.3 Návrh vzhledu aplikace

Před zahájením implementace jsem experimentoval s pojetím grafického rozhraní. Když vyvíjíme nástroj, který slouží pro vývoj, tak na grafickém rozhraní hodně záleží. Uživatel by měl mít vše po ruce a neustálý přehled o svém simulačním modelu. Informace, které na první pohled nejsou vidět, by měly být stále dobře na dosah.

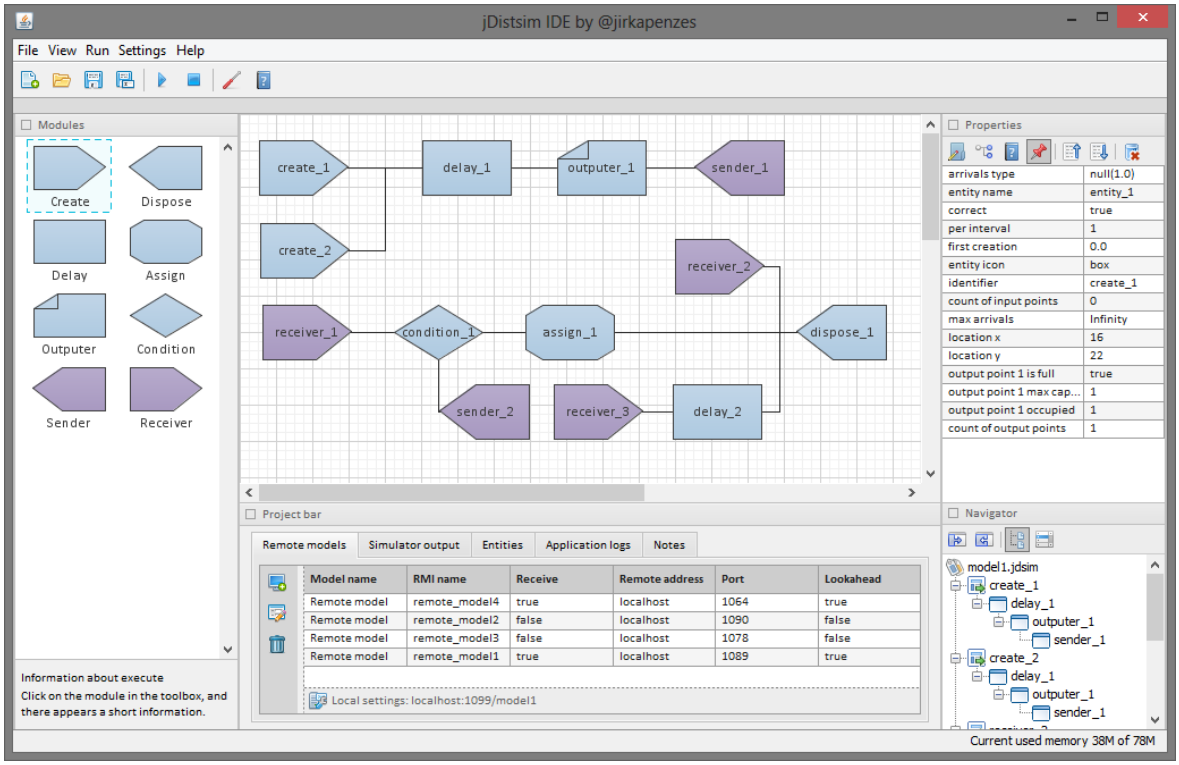
Proto jsem se vydal nejprve cestou WireFrame (Obrázek 47). WireFrame je oblíbený zejména při návrhu webových stránek. V poslední době ale zaznamenal velkou oblibu i při návrhu mobilních aplikací a desktop aplikací. Jde o přístup, kdy si vytvoříme tzv. drátěný model aplikace. V poslední době se často objevuje i termín *skica aplikace*. Definujeme pouze funkci a hrubý obsah. WireFrame by se mohl přirovnat k formě projektové dokumentace.



Obrázek 47 – WireFrame vývojového prostředí
Zdroj: Vlastní zpracování

WireFrame oceníme zejména v situaci, kdy si nevíme rady s uspořádáním ovládacích prvků. Návrh vývojového prostředí jDistsim IDE (Obrázek 47) relativně dobře znázorňuje uvažovanou představu aplikace. Dominantním bude zejména hlavní prostor uprostřed, kde budeme budovat simulační model. Na první pohled je patrné rozložení aplikace. V levé části nalezneme paletu modulů a pod paletou bude zobrazena malá nápověda pro právě vybraný modul. Uprostřed dominuje plátno, na kterém budeme kreslit simulační model, a ve spodní části se nachází informační panel, který je složen z několika záložek. V pravé části lze pohlédnout na vlastnosti modulu, který je aktuálně vybraný, a ve spodní části pod vlastnostmi se nachází navigátor.

V samotné implementaci vývojového prostředí jsem se snažil WireFrame návrhu držet. Výsledná realizace aplikace je vidět na obrázku (Obrázek 48).



Obrázek 48 – Výsledná aplikace
Zdroj: Vlastní zpracování

Aplikace je rozčleněna do šesti hlavních částí:

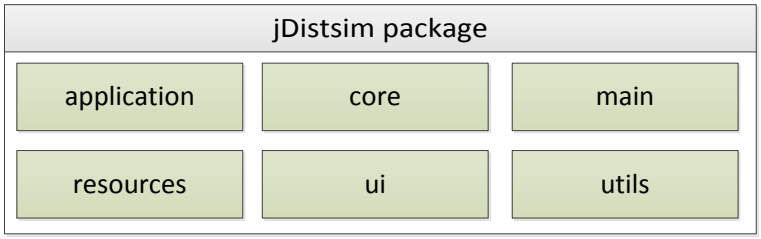
- Toolbox – rychlý přístup k ukládání, resp. načítání modelu a spouštění simulace.
- Modules – paleta modulů.
- Project bar – informace o budovaném modelu a logy (rozděleno do záložek).
- Properties – informace o nastavení aktuálně vybraného modulu.
- Navigator – rychlý přehled všech modulů v simulačním modelu.

- Model space – prostor, ve kterém navrhujeme simulační model.

8.4 Architektura

Architektura celé aplikace je částečné postavena na návrhovém vzoru MVC. Všechny výše popsané části jsou v mé aplikaci MVC složkami – každá má svůj vlastní Controller, Model a View. Pro tento účel jsem si navrhl vlastní malý MVC Framework, který při startu aplikace prováže Controllery, Modely a Pohledy, a sestaví hlavní Frame aplikace. Říkám tomu zavaděč a jeho činnost je postavena nad vzorem Service-locator, který mapuje instance všech složek k očekáváným rozhraním. Každá implementace MVC složky je skryta za rozhraním a je proto možné v tomto zavaděči případně zaměnit řídící a vizuální složky za jiné.

Struktura zdrojového kódu je rozčleněna do balíčků. Kořenovým balíčkem je `jdistsim`, který je dále členěný do šesti hlavních balíčků (Obrázek 49).

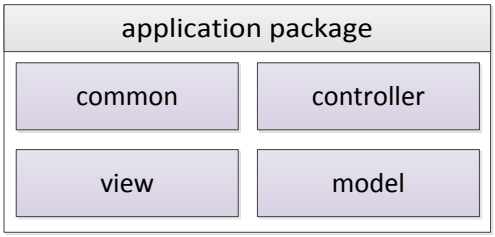


Obrázek 49 – Struktura hlavního balíčku jDistsim
Zdroj: Vlastní zpracování

Ve schématech balíčku jsou vyobrazeny pouze další balíčky, které jsou pro daný balíček důležité. Žádné třídy, rozhraní, apod.

8.4.1 Balíček application

Balíček `jdistim.application` obsahuje třídy, které řídí logiku a strukturu celé aplikace (Obrázek 50). Jsou zde zastoupeny všechny MVC složky a také akce, které můžeme v rámci modelu provádět. Akcí se rozumí například práce s moduly (pohyb, spojování, výběr apod.). V balíčku dále mimo jiné nalezneme i UI konfiguraci, registr modulů, hlavní frame nebo dialog builder.



Obrázek 50 – Struktura balíčku application
Zdroj: Vlastní zpracování

8.4.2 Balíček resources

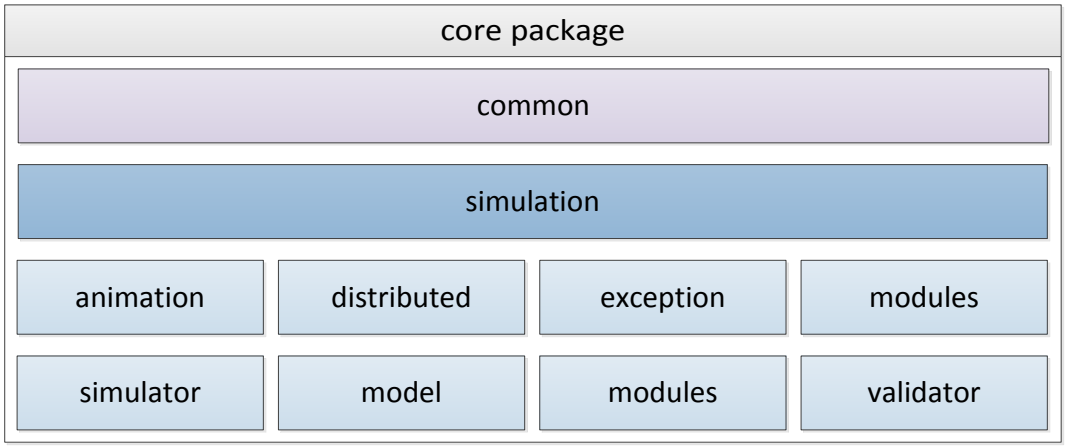
Balíček `resources` neobsahuje žádné třídy, resp. žádný kód. Jsou v něm umístěny veškeré podpůrné materiály, které se v aplikaci používají. Zejména ikony a obrázky.

Obrázek 51 – Struktura balíčku resources
Zdroj: Vlastní zpracování

8.4.3 Balíček core

Hlavní motor aplikace je skrytý v balíčku `jdistim.core` (Obrázek 52), který je dále členěn do dvou balíčků (vrstev) – `common` a `simulation`. Ve vrstvě `jdistim.common` nalezneme třídy zastupující logiku ukládání rozpracovaného modulu do souboru. Zajímavá je obzvláště vrstva `jdistim.simulation`. V tomto balíčku nalezneme vše, co se týká simulačního procesu. Nachází se zde simulační jádro, implementace základních modulů, výjimky, validátor, definice modelu a další. Nalezneme tu i komunikační vrstvu. Třídy pracující s RMI registrem, veřejné RMI rozhraní a binder třídy pro připojení ke vzdáleným logickým procesům.

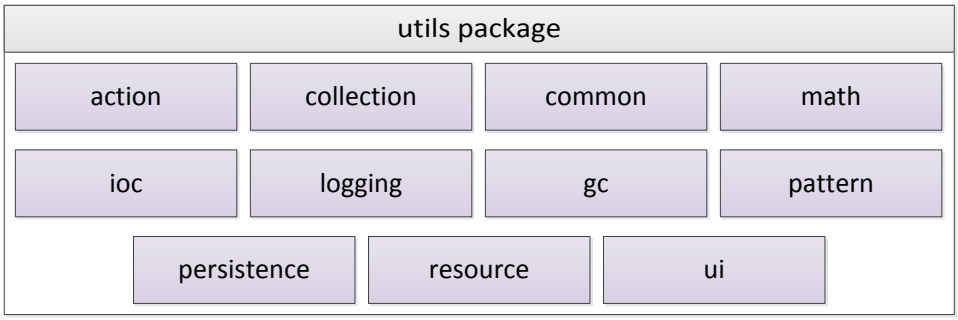
Část tohoto balíčku byla popsána v kapitole 6, ve které jsme se detailně zabývali jádrem distribuované simulace.



Obrázek 52 – Struktura hlavního balíčku core
Zdroj: Vlastní zpracování

8.4.4 Balíček utils

Jak již název lehce napovídá, v tomto balíčku nalezneme všechny pomocné nástroje, prostředky pro práci s vlákny, garbage collectorem nebo xml souborem. Jsou zde i implementace používaných návrhových vzorů (například zmiňovaný MVC Framework nebo IoC kontejner). Mimo dalšího i logovací knihovna, prostřednictvím které je celá aplikace logována.



Obrázek 53 – Struktura hlavního balíčku utils
Zdroj: Vlastní zpracování

V balíčku `jdistim.utils` narazíme i na grafickou podporu (ovládací prvky, posluchače, layouty, …) nebo na speciální kolekce – kupříkladu Observable list nebo Synchronizovanou (thread-safe) prioritní frontu.

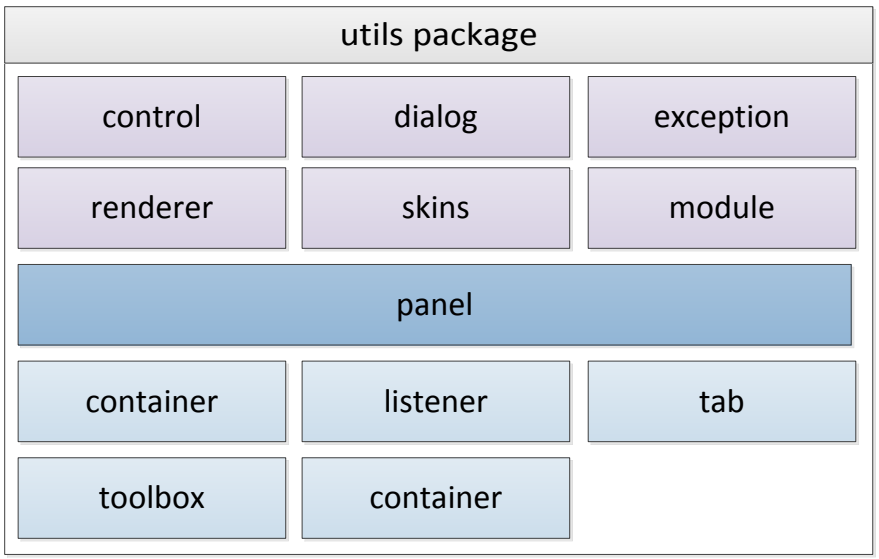
8.4.5 Balíček main

Balíček jdistim.`main` obsahuje pouze třídy, které se postarají o spuštění aplikace (zavolají příslušné objekty z aplikační vrstvy pro vykreslení GUI).

8.4.6 Balíček ui

Aby aplikace byla dobře použitelná a pro uživatele příjemná na použití, pak nesmíme opomenout práci s grafickým rozhraním. Při tvorbě vývojového prostředí hodně záleží na kvalitě grafické rozhraní – uživatelovi se musí s aplikací pracovat dobře a jednoduše.

Pokud nahlédnete do balíčku `jdistim.ui` více (Obrázek 54), pak zjistíte, že je velmi obsáhlý. Velká část grafického rozhraní jDistsim IDE je ručně vykreslována a jen v málo případech jsou použity hotové komponenty. V tomto balíčku jsou implementace všech panelů, ovládacích prvků, rendererů, grafiky modulů a další.



Obrázek 54 – Struktura hlavního balíčku ui
Zdroj: Vlastní zpracování

8.4.7 Událostně řízený model

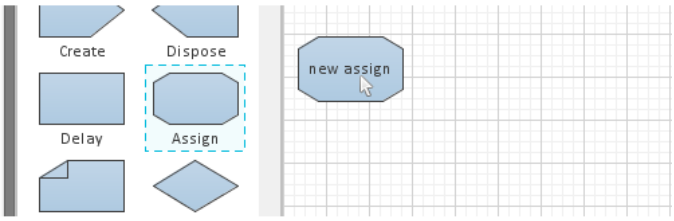
Mnoho částí aplikace mezi sebou potřebuje sdílet své stavy. Stavy se ale často mění a je nutné zabezpečit aktuálnost informací napříč celou aplikací. Mohli bychom jít cestou vytváření závislostí mezi třídami. Provázanost závislostí by se ale časem stala neúnosná, třídy by ztratily soudružnost a kód by přestával být znovupoužitelným, resp. lehce nahraditelným. Z tohoto důvodu jsem se vydal cestou notifikací, neboli událostně řízeného modelu. V případě nějaké změny je vyvolána notifikace, na kterou může kdokoliv zareagovat. Událost se šíří celou aplikací a zabezpečuje konzistenci. Třídy se mohou vzájemně sledovat. Je zde v hojné míře využit návrhový vzor Observer, který byl popsán v kapitole 4.5.1.

8.4.8 Kód, třídy, rozhraní a verzování

Během vývoje jsem se snažil dodržovat principy objektově orientovaného přístupu a zásady čistého kódu. Kód by měl být z velké části dobře čitelný a aplikace by měla jít relativně snadno rozšířit o další funkcionality nebo máme možnost některé části nahradit jinými. Funkci jednotlivých tříd v této práci popisovat nebudu. Zdrojové kódy aplikace obsahují více jak 300 tříd (případně rozhraní, apod.). Sám název třídy většinou vypovídá o její činnosti. Při vývoji byl použit verzovací systém Git a při nahlédnutí do commit zpráv lze také usoudit důvod vzniku třídy a případně i její činnost.

8.5 Ovládání aplikace – modelování

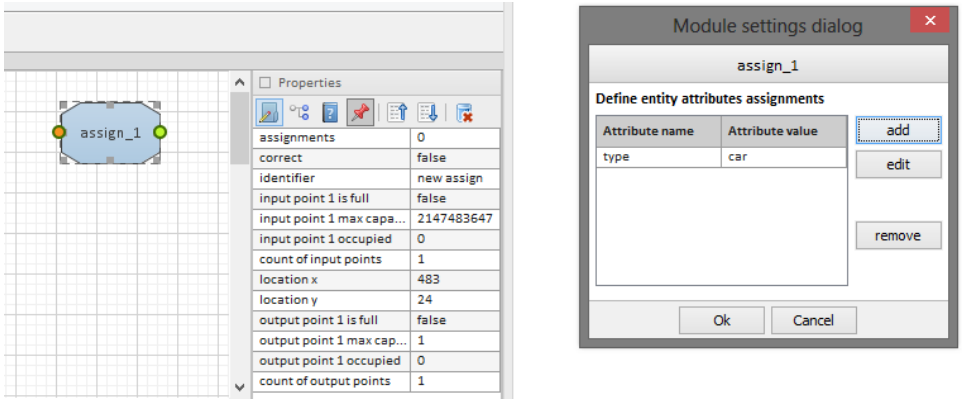
Simulační modely budujeme spojováním modulů. Vytváříme závislosti. Při tvorbě modelu lze použít pouze ty moduly, které jsou zobrazeny v paletě (panel Modules). Tvorba modelu je snadná. Vybereme si modul z palety a stiskem levého tlačítka myši jej přetáhneme (Obrázek 55) na kreslící plátno (způsobem drag-and-drop).



Obrázek 55 – Přetažení modulu na kreslící plátno
Zdroj: Vlastní zpracování

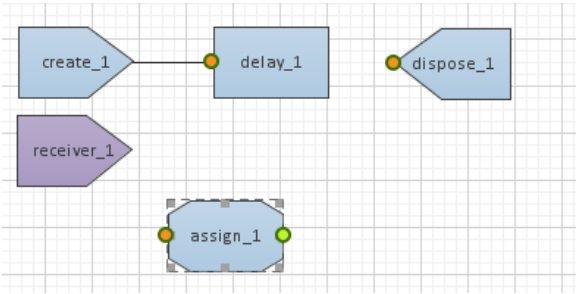
Po přetáhnutí modulu do kreslícího plátna je vytvořen dočasný modul, který nemá ještě žádné vlastnosti. Najetím myši zpět na paletu se zahodí.

Pracovat s modulem můžeme až po umístění do kreslícího plátna. Po dokončení přetažení se modulu přiřadí unikátní název a můžeme jej vybrat (kliknutím na modul). Po vybrání modulu se nám zobrazí v panelu vlastností (Properties) nastavení aktuálního modulu a po kliknutí na první ikonku v panelu se otevře dialogové okno s nastavením (Obrázek 56).



Obrázek 56 – Vlastnosti a editace modulu
Zdroj: Vlastní zpracování

Při vytváření modelu se snaží vývojové prostředí napovídat. Potažmo nepřipustí špatné vazby modulů. Je-li vybrán modul, pak se zvýrazní vstupní a výstupní body (Obrázek 57).



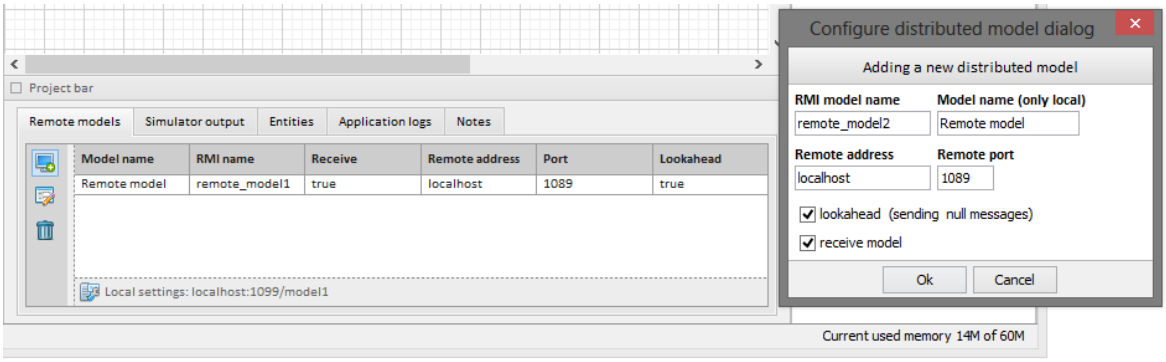
Obrázek 57 – Spojování modulů
Zdroj: Vlastní zpracování

Všimněte si, že se body zobrazily pouze u některých modulů. Vývojové prostředí již při návrhu hlídá správnost vazeb mezi modely. Zeleně je zvýrazněný bod, ze kterého můžeme vytvořit závislost (výstup). Oranžovou barvou jsou zvýrazněné ty body, se kterými může daný modul zhotovit závislost (vstupy). S jinými se nám to nepovede. Modul může vytvořit závislost i sám se sebou. Vazbu vytvoříme opět stylem drag-and-drop – tažením stisknutého tlačítka myši ze zeleného bodu do libovolného oranžového bodu.

8.5.1 Konfigurace distribuovaných modelů

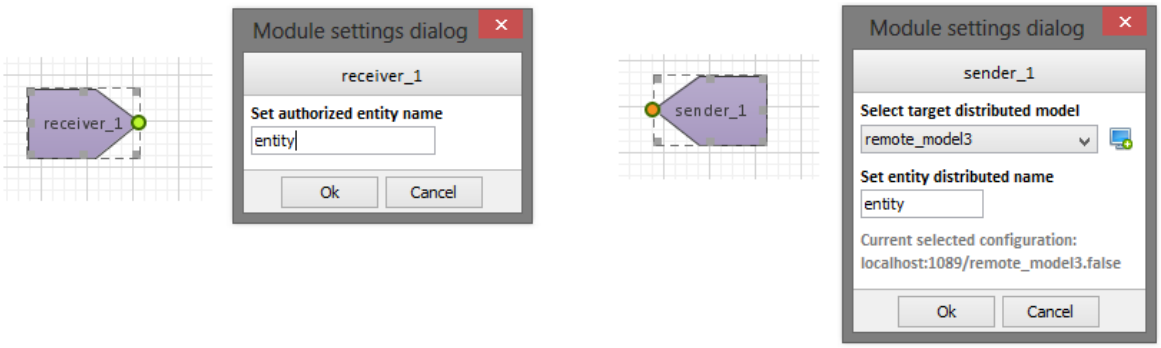
Ve spodní části uprostřed (pod kreslícím plátnem) se nachází informační panel, který obsahuje několik záložek. První záložkou je *Remote models* (Obrázek 58). Jedná se o tabulku, do které musíme přidat všechny simulační modely, se kterými chceme během simulace komunikovat. Jsou zde uvedeny jak modely, kterým chceme entity posílat, tak i modely, od kterých chceme entity přijímat.

Vzdálené distribuované modely musejí mít vždy unikátní název. Nelze přidat dva modely se stejným názvem. Aplikace si toto hlídá a nedovolí nám to.



Obrázek 58 – Konfigurování vzdálených modelů
Zdroj: Vlastní zpracování

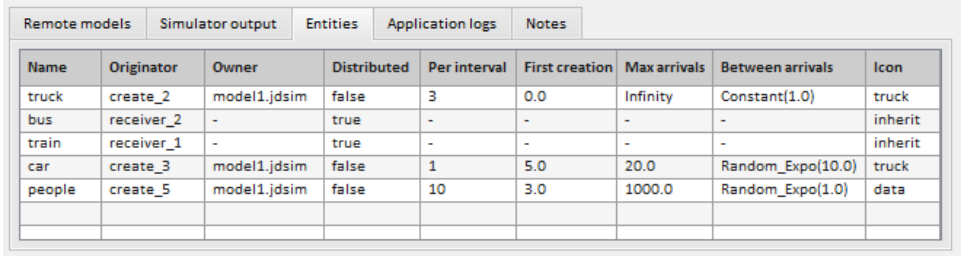
V pravé části se nacházejí tři ikony, které umožňují přidávání vzdálených modelů, editaci a případně jejich smazaní. Komunikaci s ostatními modely realizujeme prostřednictvím distribuovaných modulů (viz kapitola 5). Modulu Receive nastavujeme pouze klíč entity (Obrázek 59). Nevolíme žádný konkrétní vzdálený model. Příchozí entity jsou rozdělovány pouze na základě klíče. V případě modulu Sender je již zapotřebí vybrat model, na který se má entita zaslat a s jakým klíčem (Obrázek 59).



Obrázek 59 – Nastavení modulu Receiver a Sender
Zdroj: Vlastní zpracování

8.5.2 Entity v modelu

Entit se nám v modelu může vyskytovat mnoho. Můžeme si je vytvářet sami nebo nám můžou přicházet ze sítě od ostatních logických procesů. Přehled v entitách lze nalézt na třetí záložce v informačním panelu (Obrázek 60).

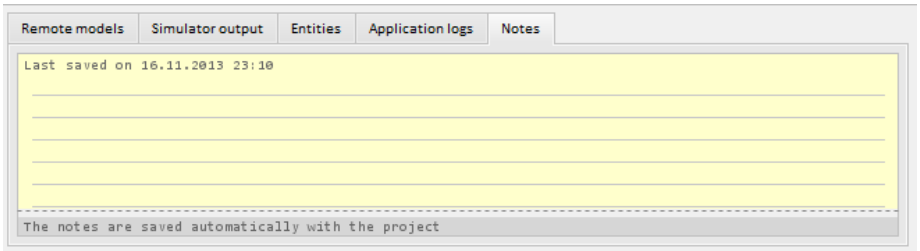


Obrázek 60 – Přehled entit v simulačním modelu
Zdroj: Vlastní zpracování

Na první pohled vidíme, které entity a kolik jich přichází zvenčí. Mimo jiné máme i přehled o lokálních entitách a jejich nastavených hodnotách.

8.5.3 Poznámky

Aplikace jDistsim IDE je určena pro tvorbu simulačních experimentů. Mnohdy je zapotřebí si během vývoje simulačních modelů zapisovat poznámky. Ve vývojovém prostředí je připraven i prostor pro zápis poznámek, aby uživatel nemusel sahat po další aplikaci (Obrázek 61).

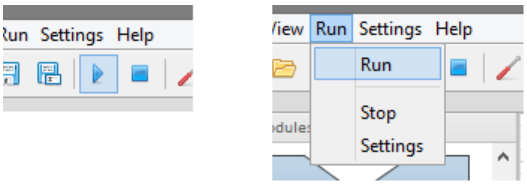


Obrázek 61 – Poznámky
Zdroj: Vlastní zpracování

Poznámky najdeme pod poslední záložkou v informačním panelu.

8.6 Zahájení simulace

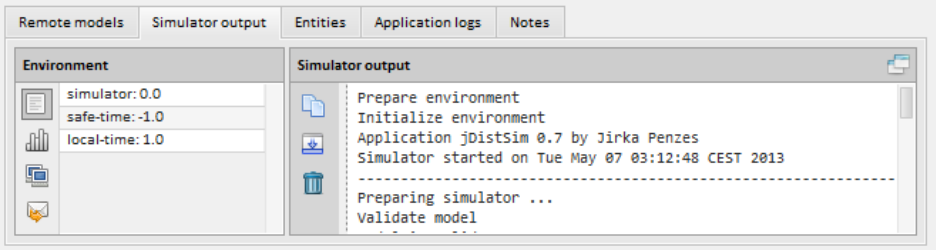
Pokud máme model hotový, pak můžeme spustit simulátor. Simulátor lze spustit kliknutím na ikonu Run v toolboxu nebo v menu Run/Run (Obrázek 62).



Obrázek 62 – Spuštění simulátoru
Zdroj: Vlastní zpracování

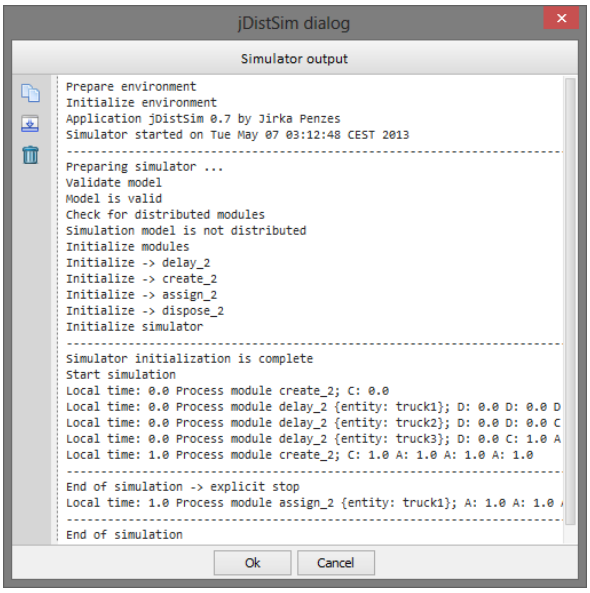
Pozastavení simulátoru lze provést obdobným způsobem. Kliknutím na ikonu Stop nebo v nabídce Run/Stop. Při pozastavení simulace dojde ke kompletnímu odrolování aktivit simulačního jádra. Tím je myšlena zejména komunikační vrstva – ukončení komunikace se vzdálenými logickými procesy a vynucené odstranění lokálního modelu z registru RMI. Ukončení komunikace má za následek zrušení simulačního výpočtu na všech logických procesech v celé síti. Nelze odpojit pouze jeden logický proces.

V okamžiku, kdy spustíme simulátor, se informační panel přepne na druhou záložku (Simulator output). Na této záložce máme možnost sledovat výstup ze simulátoru v reálném čase. Výstup ze simulátoru nám poskytuje informace o aktuálně prováděných operacích jádrem nebo o průběhu simulačního výpočtu. Dále na této záložce vidíme stavové hodnoty prostředí simulátoru (Obrázek 63).



Obrázek 63 – Výstup ze simulátoru 1
Zdroj: Vlastní zpracování

Nemusí být vždy pohodlné sledovat výstup v tomto malém textovém výstupu. Lze zobrazit výstup i jiným způsobem? Odpovědí je malá ikona v pravém horním rohu textového výstupu. Kliknutím na tuto ikonu se otevře nové okno, kde můžeme taktéž sledovat výstup ze simulátoru (Obrázek 64).

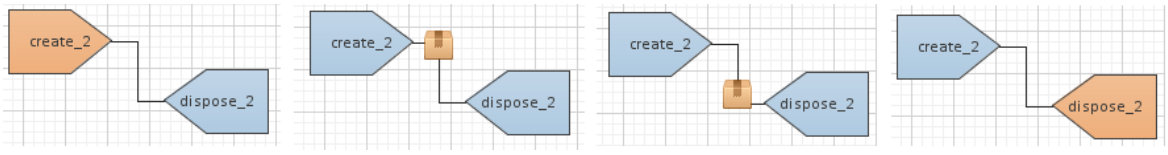


Obrázek 64 – Výstup ze simulátoru 2
Zdroj: Vlastní zpracování

Funkce simulátoru je podrobně popsána v kapitole 6.

8.6.1 Animace

Spuštěním simulačního výpočtu se zahájí i animace. Animují se stavové přechody entit mezi jednotlivými moduly. Po spojnicích se pohybují ikony a oranžovou barvou se zvýrazňují moduly – nejprve modul, ve kterém se entita aktuálně nacházela a následně modul, do kterého entita právě vstupuje (Obrázek 65).



Obrázek 65 – Animace
Zdroj: Vlastní zpracování

8.7 Aplikační log

Vedle logu simulátoru se v aplikaci vyskytuje i aplikační log. Aplikační log mimo simulátoru loguje i veškerou činnost ve vývojovém prostředí. Od samotného startu až po ukončení. Všechny akce jsou v tomto logu zaznamenány. Tento log je velmi detailní. I během simulačního výpočtu v něm můžeme nalézt více informací, než ve výstupu ze simulátoru – například lze sledovat detailně síťový provoz. Oproti výstupu pro nás nemusí být aplikační log tak přehledným, neboť je v něm logováno příliš mnoho informací. Je v něm zaznamenána třída a metoda, ze které byl log vyvolán, dále datum a čas, úroveň logu a logovací zpráva.

Aplikační log je zaveden zejména z důvodu laděni a rychlé identifikace vzniklé chyby během runtime běhu aplikace. Logovací knihovna je relativně komplexní a lze jí nalézt v balíčku `jDistsim.utils.logging`.

8.8 Ukládání a načítání souboru – formát jdsim

Rozpracované nebo hotové soubory lze ukládat do souboru. Modely se ukládají ve formátu XML s příponou jdsim. Společně se simulačním modelem je do souboru taktéž uložena i aktuální konfigurace vzdálených modelů, poznámky a mnoho dalších informací o aktuálním nastavení. Formát výsledného XML nazývám jdsim.

Při implementaci ukládání jsem se snažil být úsporný, co se velikosti výsledného XML souboru týče. Z toho důvodu je simulační model před uložením transformován na speciální objekt, který si poznamená několik informací o modelu. Na základě těchto informací dokáže vývojové prostředí posléze model zase postavit. Typicky se ukládají názvy tříd modulu a na základě Java Reflexe jsme schopni při sestavování utvářet instance modulů a vybudovat znovu simulační model. Objekt transformujeme do formátu XML a následně uložíme do souboru.

Ukázka formátu jdsim je obsažena v příloze (Příloha E – XML soubor).

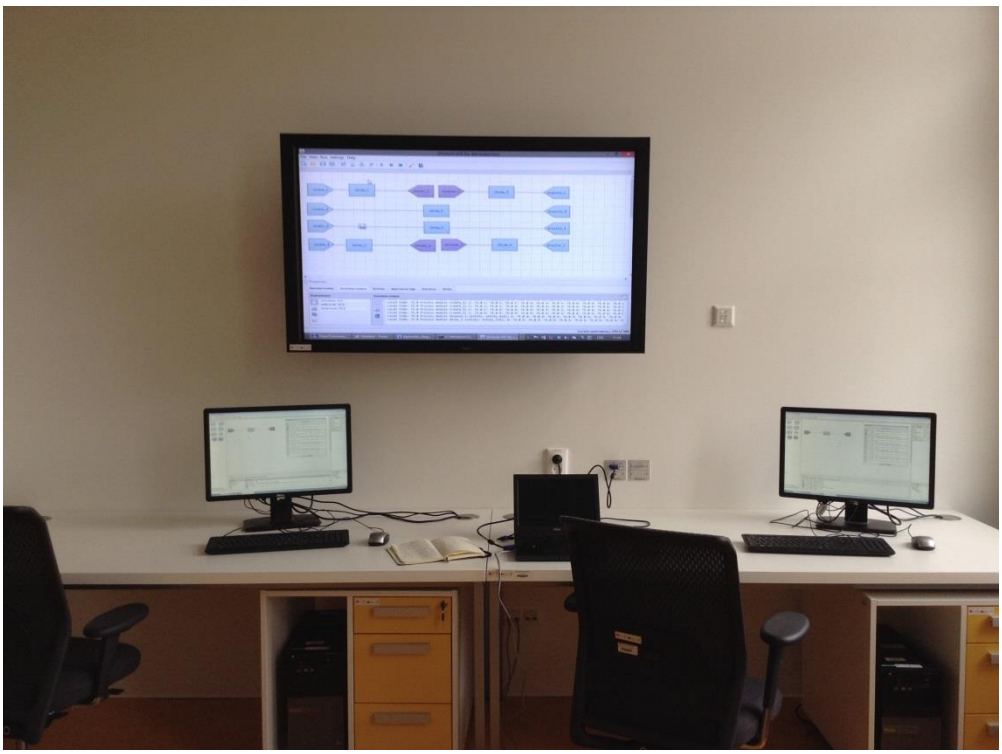
8.9 Problémy během vývoje

Během vývoje aplikace jDistsim IDE jsem narazil na celou řadu úskalí. Vývojové prostředí není malá aplikace a pracujeme s hodně vysokou mírou abstrakce. To přináší celou řadu problémů. Už před samotným vývojem musíme správně zvolit například architekturu. Dalším bodem je grafické rozhraní. Aplikace jDistsim IDE má většinu grafických komponent původních s vlastní renderovací logikou. Většina komponent je šita na míru přímo pro svou funkci. Vytváření vlastních komponent bylo sice náročné, ale přineslo to řadu výhod – ovládání je díky vzhledu velmi intuitivní. Během vývoje se objevilo mnoho problémů například se synchronizací a obecně během vláken, komunikací nebo XML serializací.

8.10 Nasazení

Po zkompilování bylo zapotřebí otestovat vývojové prostředí v reálném provozu. Testovacím příkladem byla dálnice s přilehlými občerstveními, se kterou jste se mohli seznámit v kapitole 5.4.2. Příklad se skládá ze tří logických procesů, kde předpokladem je, že každý logický proces poběží na samostatném počítači.

Test proběhl v laboratoři počítačové simulace na půdě Fakulty elektrotechniky a informatiky Univerzity Pardubice (Obrázek 66).



Obrázek 66 – Testování nasazení aplikace v reálném prostředí
Zdroj: Vlastní zpracování

Na obrázku výše jsou vidět tři počítače – dva klasické stolní a jeden notebook, který je připojen k externímu monitoru (uprostřed). Všechny počítače byly vedle sebe a bylo možné sledovat v podstatě okamžitý přechod entity z jednoho počítače na druhý během simulačního výpočtu.

8.11 Shrnutí

Vývojové prostředí splňuje všechny požadavky, které jsme si kladli v úvodu této kapitoly. Je zde velký potenciál pro další rozvoj celé aplikace nebo pouze simulační knihovny. Deklarativní přístup navrhování modelů je velmi intuitivní přístup a při bohatší paletě modelů nejsme takřka ničím limitováni a můžeme vytvářet jakékoliv simulační modely.

Aplikace jDistsim může sloužit i jako výukový prostředek pro názornou ukázku distribuované simulace v praxi.

Část 4

9 Závěr

V diplomové práci jsme se zabývali zejména problematikou distribuované simulace. Hledáme-li cestu, která přinese v simulaci velký výpočetní výkon, potom je distribuovaná simulace velmi vhodným kandidátem na nasazení. Musíme si ale uvědomit, že i samotný proces distribuovaného běhu přináší jisté požadavky na procesorový čas a komunikační režii. Pakliže máme spíše jednoduší simulační model, je dobré ještě zvážit, zda aplikace klasického sekvenčního přístupu není výhodnější cestou.

Nejvýznamnější nevýhodou každého distribuovaného systému je jeho složitý návrh a implementace. Při implementaci distribuované simulace se klade velký důraz zejména na správně navržený postup, který hlídá synchronizaci času mezi jednotlivými logickými procesy. Tato část je velmi důležitá. Narušení kauzality synchronizace času nesmí v simulačním procesu nikdy nastat. Jeden z mechanismů, jak se tomuto nezmaru vyvarovat, je kupříkladu zasílání tzv. nulových zpráv, které slouží výlučně k synchronizaci simulačního času. Při implementaci je také snaha maximálně detekovat stavy uváznutí, které mohou nastat, a předcházet jim.

Hlavním cílem práce bylo vytvořit vývojové prostředí, které bude sloužit k návrhu a spouštění distribuovaných simulačních modelů. Výsledkem je aplikace jDistsim IDE. Cesta vývoje nebyla snadná a obsahovala spousty problémů a slepých uliček. Prvním úkolem bylo najít způsob, jak distribuované simulační modely tvořit. Zde se osvědčil způsob zachycení životního cyklu entity pomocí speciálních modulů, které reprezentují logiku simulačního modelu. Komunikace mezi logickými procesy probíhá právě pomocí speciálních modulů. Dalším úkolem byla implementace knihovny, která bude podporovat distribuovanou simulaci s deklarativním způsobem tvorby simulačních modelů. Simulační jádro vychází z konzervativní synchronizační metody s mechanismem zasílání zpráv na vyžádání.

Etapovitý přístup řešení postupně vyústil ve zmiňovanou aplikaci jDistsim IDE, která byla naprogramována v platformě Java. Vzniklá aplikace umožňuje a pomáhá uživateli budovat deklarativním způsobem simulační modely v grafickém rozhraní. Vytvořené simulační modely je poté možné spustit i v distribuovaném simulačním prostředí. Vývoj nebyl snadný. Prostředí jDistsim IDE obsahuje tisíce a tisíce řádků kódu a je připraveno pro případné další rozšíření.

Cíle diplomové práce byly splněny. Práce mi byla ohromným přínosem a jsem rád, že se mi dostalo možnosti blíže poznat téma distribuované simulace a implementovat rozsáhlou aplikaci, ze které vzniklo pěkné vývojové prostředí jDistsim IDE.

Literatura

1. **Fujimoto, Richard M.** *Parallel and Distributed Simulation Systems.* New York : Wiley-Interscience, 2000. 978-0471183839.

2. **Kuneš, Antonín.** *Distribuovaná simulace s konzervativním synchronizačním algoritmem.* Plzeň, 2004. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

3. **Pastyřík, Jakub.** *Distribuovaná simulace s optimistickým synchronizačním algoritmem.* Plzeň, 2004. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

4. **Tropper, Carl.** *Parallel and Distributed Discrete Event Simulation.* Waltham : Nova Biomedical, 2002. 978-1590333778.

5. Kurz distribuované simulace. *BATCOS.* [Online] Západočeská univerzita v Plzni. [Citace: 20. Duben 2013.] http://athena.zcu.cz/kurzy/sspr/000/HTML/101/.

6. Java™ Platform, Standard Edition 7. *Java documentation.* [Online] Oracle, 2013. [Citace: 20. Duben 2013.] http://docs.oracle.com/javase/7/docs/api/.

7. **Jelínek, Lukáš.** Java (20) - vlákna. *Linuxsoft.* [Online] 2. Listopad 2005. [Citace: 20. Duben 2013.] http://www.linuxsoft.cz/article.php?id_article=1006. 1801-3805.

8. **Vogel, Lars.** Java concurrency (multi-threading). *Vogella.* [Online] 9. Duben 2013. [Citace: 20. Duben 2013.] http://www.vogella.com/articles/JavaConcurrency/article.html.

9. **Chacon, Scott.** *Pro Git.* : APress, 2009. 978-1430218333.

10. **Laird, Cameron.** Modern threading: A Java concurrency primer. *Java World.* [Online] Infoworld, Inc., 6. Červen 2012. [Citace: 20. Duben 2013.] http://www.javaworld.com/javaworld/jw-06-2012/120626-modern-threading.html.

11. Observer Pattern. *Object Oriented Design.* [Online] Open Source Matters, 2006. [Citace: 29. Duben 2013.] http://www.oodesign.com/observer-pattern.html.

12. **Bishopová, Judith.** *C#.NET - návrhové vzory.* Brno : Zoner press, 2010. 978-80-7413-076-2.

13. **Martin, Robert C.** *Čistý kód.* Praha : Computer press, 2009. 978-80-251-2285-3.

14. **Fowler, Martin.** GUI Architectures. [Online] 18. Červenec 2006. [Citace: 29. Duben 2013.] http://martinfowler.com/eaaDev/uiArchs.html.

15. **Sohi, Avtar Singh.** Understanding Basics of UI Design Pattern MVC, MVP and MVVM. *CodeProject.* [Online] 20. Červenec 2011. [Citace: 27. Duben 2013.] http://www.devbook.cz/mvc-architektura-navrhovy-vzor.

16. **Prasanna, Dhanji R.** *Dependency Injection.* New York : Manning Publications, 2009. 193398855X.

17. **Fowler, Martin.** http://www.martinfowler.com/articles/injection.html. [Online] 23. Leden 2004. [Citace: 25. Duben 2013.] http://www.martinfowler.com/articles/injection.html.

18. Inversion of Control, Dependency Injection. *Rising sun.* [Online] 15. Duben 2008. [Citace: 26. Duben 2013.] https://sqdw.signaly.cz/0804/inversion-of-control-dependency.

19. A Basic Introduction On Service Locator Pattern. *CodeProject.* [Online] 26. Duben 2007. [Citace: 29. Duben 2013.] http://www.codeproject.com/Articles/18464/A-Basic-Introduction-On-Service-Locator-Pattern.

20. **Grosso, William.** *Java RMI.* : O'Reilly Media, 2001. 978-1565924529.

21. **Harold, Elliotte Rusty.** *Java Network Programming.* : O'Reilly Media, 2004. 978-0596007218.

22. **Downing, Troy Bryan.** *Java RMI: Remote Method Invocation.* : John Wiley & Sons, 1998. 978-0764580437.

23. **Knoernschild, Kirk.** *Java Application Architecture: Modularity Patterns with Examples Using OSGi.* : Prentice Hall, 2012. 978-0321247131.

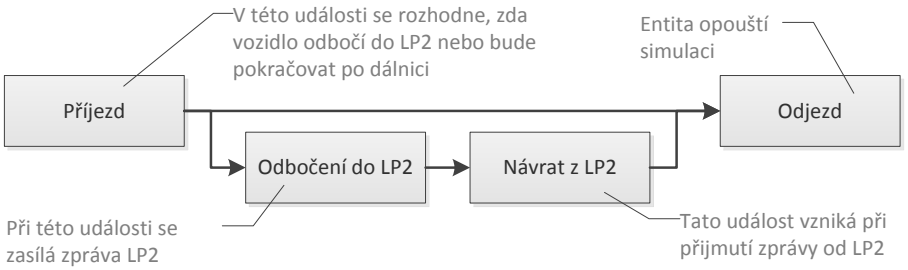
24. **Reilly, David.** Introduction to Java RMI. *Java Coffee Break.* [Online] 5. Červen 2006. [Citace: 28. Duben 2013.] http://www.javacoffeebreak.com/articles/javarmi/javarmi.html.

25. **Pénzeš, Jiří.** *Distribuované systémy platformy Java.* Pardubice : autor neznámý, 2011. Bakalářská práce. Univerzita Pardubice, Fakulta elektrotechniky a informatiky.

Přílohy

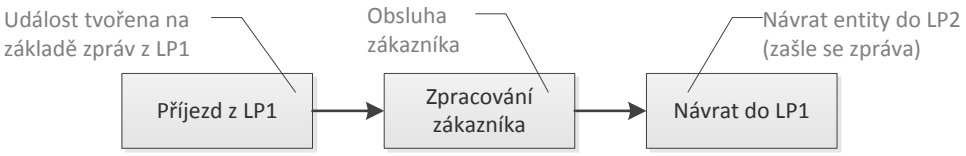
Příloha A – evoluce algoritmu nulových zpráv na vyžádání

Uvažujme problém – máme dálnici (LP1) a občerstvení (LP2). Stanovme, že LP1 může žádat o dobu výhledu proces LP2 a to pouze v případě detekce stavu uváznutí. Životní cyklus dálnice je zachycen na obrázku níže (Obrázek 67), kde je vyobrazen sled čtyř událostí, ze kterých se dálnice skládá.



Obrázek 67 – Logický proces dálnice
Zdroj: Vlastní zpracování

Proces rychlého občerstvení má podstatně jednodušší logiku a skládá se pouze ze tří událostí (Obrázek 68). Hlavní úloha spočívá v obsluze zákazníka a následném návratu vozidla zpět na dálnici.



Obrázek 68 – Logický proces občerstvení
Zdroj: Vlastní zpracování

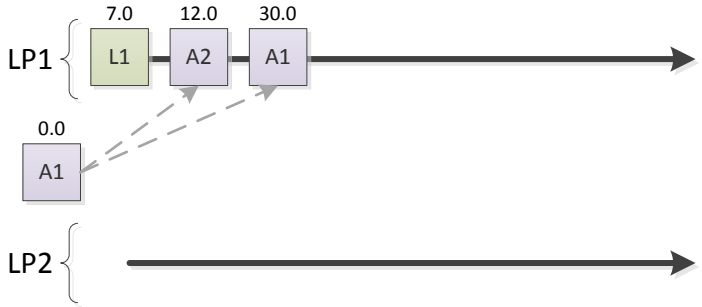
Nyní následuje ukázka evoluce simulačního procesu. Proces započne inicializací všech logických procesů. Předpokládejme, že každý proces běží na jiném počítači. Následuje sled jednotlivých kroků, které se odehrávají během simulace.

Dálnice (LP1) - inicializace			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
0.0	-	Příjezd A1	0

Rychlé občerstvení (LP2) - inicializace			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
0.0	-	-	0

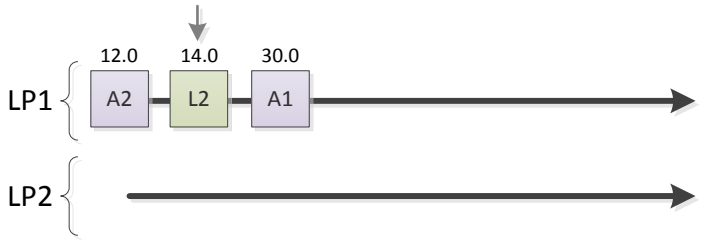
Při spuštění simulační jádro indikuje, že proces LP1 má prázdnou vstupní frontu LP2 a zároveň má nějaké události již připravené ke zpracování. V konfiguraci simulace jsme stanovili, že LP1 může žádat o dobu výhledu. Proto LP1 v tento okamžik zašle nulovou zprávu s žádostí o výhled LP2 a následně bude čekat na odpověď. LP2 odpovídá a zasílá událost nulovou zprávu L1. Můžeme pokračovat dál v simulaci, fronta je neprázdná.

Dálnice (LP1) - příjezd A1			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
0.0	příjezd A1	odjezd A1, příjezd A2	1

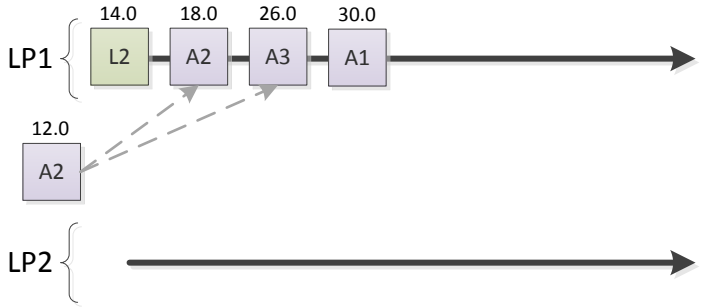


Dálnice (LP1) - synchronizační nulová zpráva L1			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
7.0	L1	-	0

Vstupní fronta LP2 je prázdná – je zapotřebí zaslat novou nulovou zprávu s žádostí o dobu výhledu.

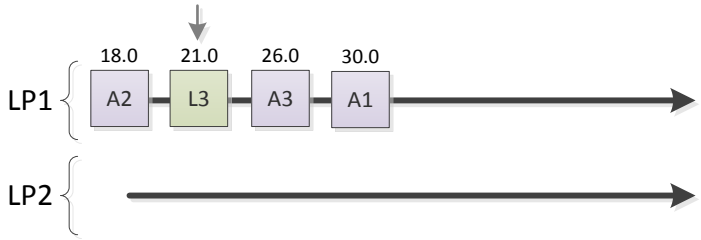


Dálnice (LP1) - příjezd A2			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
12.0	příjezd A2	odbočení A2, příjezd A3	1

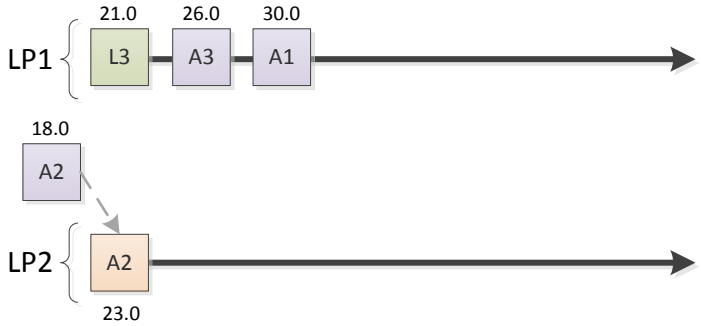


Dálnice (LP1) - synchronizační nulová zpráva L2			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
14.0	L2	-	0

Vstupní fronta je opět prázdná – musíme zaslat další žádost.



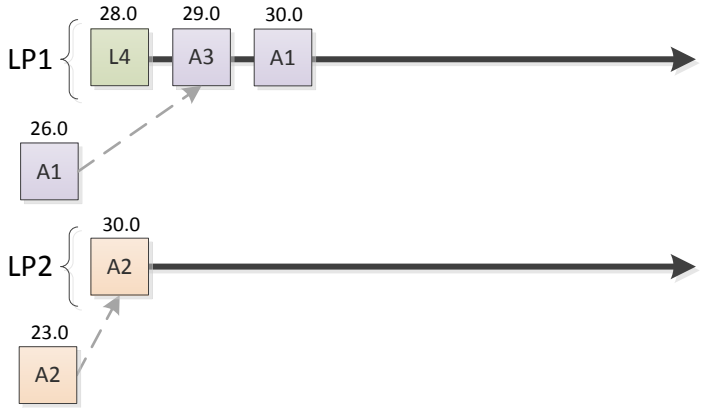
Dálnice (LP1) - odbočení A2			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
18.0	odbočení A2	přesun A2 do LP2	1



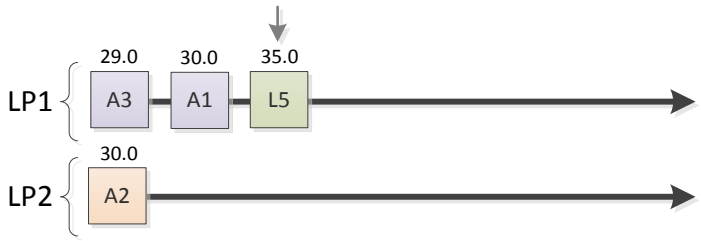
Dálnice (LP1) – příjezd A3			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
26.0	příjezd A3	odbočení A3	1

Rychlé občerstvení (LP2) - příjezd A2			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
23.0	příjezd A2	obsluha A2	0

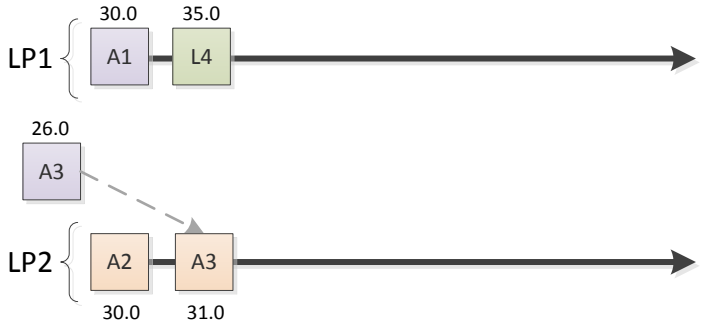
Nastala situace, ve které má LP2 má událost, ale hodnota vstupní fronty je prázdná. LP2 nemá povoleno zasílat nulové zprávy, a proto musí čekat, dokud nedostane další zprávu.



Dálnice (LP1) - synchronizační nulová zpráva L4			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
28.0	L4	-	0

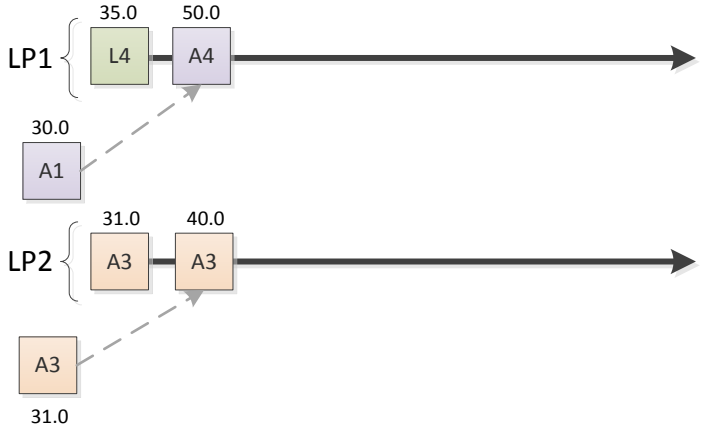


Dálnice (LP1) – odbočení A3			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
28.0	odbočení A3	přesun A3 do LP2	1



Dálnice (LP1) – odjezd A1			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
30.0	odjezd A1	příjezd A4	1

Rychlé občerstvení (LP2) - příjezd A2			
Simulační čas	Zpracovává se	Bude plánováno	Hodnota čítače LP2
30.0	obsluha A2	obsluha A2	0

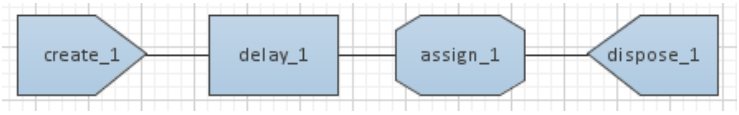


Příloha B – implementace reakce na nulovou zprávu

```
public void processNullMessageRequest(Requester requester) {
    synchronized (lock) {
      NullMessage nullMessage = NullMessage.create();
       try {
           if (calendar.isEmpty()) {
            nullMessage.setTime(requester.getLocalTime() + lookahead);
            sendNullMessage(nullMessage, this.modelName);
            return;
           }

           double lbts = getLocalTime() + lookahead;
           ScheduleEvent immediateScheduleSender = findImmediateSender();
           if (immediateScheduleSender != null) {
               if (lbts > immediateScheduleSender.getTime()) {
                   lbts = immediateScheduleSender.getTime();
               }
           }

           if (requester.getLocalTime() > lbts) {
               plan(requester.getLocalTime(),
                         new NullModuleSender(requester));
               return;
           }

           nullMessage.setTime(lbts);
           sendNullMessage(nullMessage, this.modelName);
       } catch (RemoteException exception) {
            super.throwCommunicationError();
       }
    }
}
```

Příloha C – výstupní log z jádra simulátoru

Níže zobrazený výstup ze simulačního jádra zachycuje počátek a průběh simulace tohoto simulačního modelu:




```
Prepare environment
Initialize environment
Application jDistSim 0.7 by Jirka Penzes
Simulator started on Tue May 07 03:45:16 CEST 2013
--------------------------------------------------------------------------------
Preparing simulator
Validate model
Model is valid
Check for distributed modules
Simulation model is not distributed
Initialize modules
Initialize -> delay_1
Initialize -> create_1
Initialize -> dispose_1
Initialize -> assign_1
Initialize simulator
--------------------------------------------------------------------------------
Simulator initialization is complete
Start simulation
Local time: 0.0 Process module create_1;
Local time: 0.0 Process module delay_1 {entity: entity_11};
Local time: 1.0 Process module create_1;
Local time: 1.0 Process module assign_1 {entity: entity_11};
Local time: 1.0 Process module delay_1 {entity: entity_12};
Local time: 1.0 Process module dispose_1 {entity: entity_11};
Local time: 2.0 Process module create_1;
Local time: 2.0 Process module assign_1 {entity: entity_12};
Local time: 2.0 Process module delay_1 {entity: entity_13};
Local time: 2.0 Process module dispose_1 {entity: entity_12};
Local time: 3.0 Process module create_1;
Local time: 3.0 Process module assign_1 {entity: entity_13};
--------------------------------------------------------------------------------
End of simulation -> explicit stop
--------------------------------------------------------------------------------
End of simulation
```

Příloha D – Aplikační log

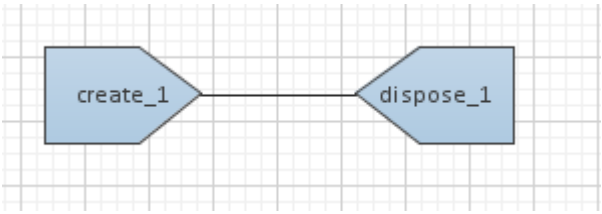
Ukázka aplikační logu – start aplikace jDistsim IDE.

```
16.04.2013: Info [jDistsim.main.MainApplication -> main()]: Default logger initialization
16.04.2013: Info [jDistsim.main.MainApplication -> main()]: Start jDistsim application
16.04.2013: Info [jDistsim.application.designer.DesignerGuiApplication -> Start()]: Create service locator
16.04.2013: Info [jDistsim.application.designer.DesignerGuiApplication -> Start()]: Prepare GUI jDistsim designer
16.04.2013: Info [jDistsim.application.designer.DesignerGuiApplication -> registerDependencies()]: Register
dependencies
16.04.2013: Info [jDistsim.application.designer.common.UIConfiguration -> <init>()]: Created designer ui
configuration
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module description not loaded ->
load dispatcher
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module descriptor is valid
16.04.2013: Info [jDistsim.core.simulation.modules.ModuleContainer -> build()]: Module configured
16.04.2013: Info [jDistsim.application.designer.MainFrame -> registerAllModels()]: Register all models: start
16.04.2013: Info [jDistsim.application.designer.MainFrame -> registerAllModels()]: Register all models: done
16.04.2013: Info [jDistsim.application.designer.MainFrame -> registerAllViews()]: Register all views: start
16.04.2013: Info [jDistsim.application.designer.MainFrame -> registerAllViews()]: Register all views: done
16.04.2013: Info [jDistsim.application.designer.MainFrame -> registerAllControllers()]: Register all controllers:
start
16.04.2013: Info [jDistsim.ui.panel.pInternalPanel -> <init>()]: Initialize internal panel -> Modules
16.04.2013: Info [jDistsim.ui.panel.atoolbox.ToolboxPanel -> <init>()]: Initialize event toolbox panel
16.04.2013: Info [jDistsim.ui.panel.nworkspace.ModelSpacePanel -> <init>()]: Initialize model space
16.04.2013: Info [jDistsim.ui.panel.dworkspace.ModelSpacePanel -> initialize()]: 0x0
16.04.2013: Info [jDistsim.ui.panel.aInternalPanel -> <init>()]: Initialize internal panel -> Properties
16.04.2013: Info [jDistsim.ui.panel.InternalPanel -> <init>()]: Initialize internal panel -> Navigator
16.04.2013: Info [jDistsim.ui.panel.workspace.InformationPanel -> <init>()]: Initialize information panel
16.04.2013: Info [jDistsim.application.designer.controller.tabLogic.EntitiesInfoTabLogic -> rebuildTableModel()]:
Rebuild entities info table
16.04.2013: Info [jDistsim.application.designer.controller.tabLogic.RemoteModelsTabLogic -> rebuildTableModel()]:
Rebuild remote models table
16.04.2013: Info [jDistsim.application.designer.MainFrame -> registerAllControllers()]: Register all controllers:
done
16.04.2013: Info [jDistsim.application.designer.MainFrame -> <init>()]: Initialize GUI designer MainFrame
16.04.2013: Info [jDistsim.application.designer.MainFrame -> layout()]: Component factory build main frame
16.04.2013: Info [jDistsim.ui.panel.ToolbarPanel -> <init>()]: Initialize toolbox panel
16.04.2013: Info [jDistsim.ui.panel.workspace.WorkSpacePanel -> <init>()]: Initialize workspace panel
```

Příloha E – XML soubor

```
<object type="jDistsim.core.common.SaveBox" id="0">
  <field name="containers">
    <object type="list" elementType="Object" length="2" id="1">
      <object type="jDistsim.core.common.SaveContainer" id="2">
        <field name="module" type="string" value="jDistsim.core.simulation.modules.lib.create.Create" />
        <field name="settings">
          <object type="jDistsim.core.simulation.modules.RootSettings" id="3">
            <field name="firsCreation" type="double" value="0.0" />
            <field name="baseEntityName" type="string" value="entity_1" />
            <field name="arrivalsType">
              <object type="jDistsim.core.simulation.modules.RootSettings$TimeBetweenArrivalsType" id="4" />
            </field>
            <field name="arrivalsTypeValue" type="double" value="1.0" />
            <field name="entityPerInterval" type="int" value="1" />
            <field name="maxArrivals" type="double" value="Infinity" />
            <field name="iconName" type="string" value="box" />
            <field name="identifier" type="string" value="create_1" />
            <field name="baseIdentifier" type="string" value="create" />
          </object>
        </field>
        <field name="location">
          <object type="java.awt.Point" id="5">
            <field name="x" type="int" value="20" />
            <field name="y" type="int" value="12" />
          </object>
        </field>
        <field name="in">
          <object type="list" elementType="Object" length="0" id="6" />
        </field>
        <field name="out">
          <object type="list" elementType="Object" length="1" id="7">
            <object type="jDistsim.core.common.SaveConnect" id="8">
              <field name="dependency" type="string" value="dispose_1" />
              <field name="sourcePointIndex" type="int" value="0" />
              <field name="targetPointIndex" type="int" value="0" />
            </object>
          </object>
        </field>
      </object>
      <object type="jDistsim.core.common.SaveContainer" id="9">
        <field name="module" type="string" value="jDistsim.core.simulation.modules.lib.dispose.Dispose" />
        <field name="settings">
          <object type="jDistsim.core.simulation.modules.lib.dispose.DisposeSettings" id="10">
            <field name="identifier" type="string" value="dispose_1" />
            <field name="baseIdentifier" type="string" value="dispose" />
          </object>
        </field>
        <field name="location">
          <object type="java.awt.Point" id="11">
            <field name="x" type="int" value="158" />
            <field name="y" type="int" value="12" />
          </object>
        </field>
        <field name="in">
          <object type="list" elementType="Object" length="1" id="12">
            <object type="jDistsim.core.common.SaveConnect" id="13">
              <field name="dependency" type="string" value="create_1" />
              <field name="sourcePointIndex" type="int" value="0" />
              <field name="targetPointIndex" type="int" value="0" />
            </object>
          </object>
        </field>
        <field name="out">
          <object idref="6" />
        </field>
      </object>
    </object>
  </field>
  <field name="remotes">
    <object idref="6" />
  </field>
  <field name="networkSettings">
    <object type="jDistsim.application.designer.common.LocalNetworkSettings" id="14">
      <field name="modelName" type="string" value="model1" />
      <field name="port" type="int" value="1099" />
      <field name="changed" type="boolean" value="false" />
    </object>
  </field>
</object>
```

Ukázkový XML soubor zachycuje uložený simulační model, který se skládá ze dvou modulů. Popisovaný simulační model:

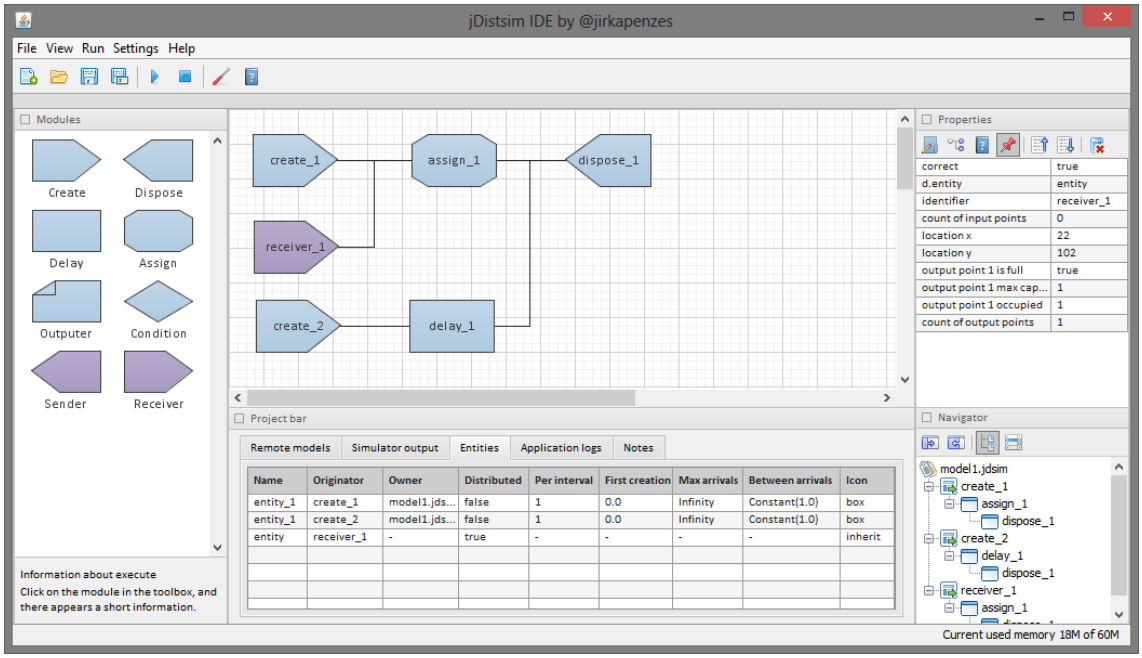


XML soubor jdsim obsahuje informace pro sestavení a vykreslení modulu. Dále jsou v souboru obsažené i informace o aktuálním nastavení aplikace.

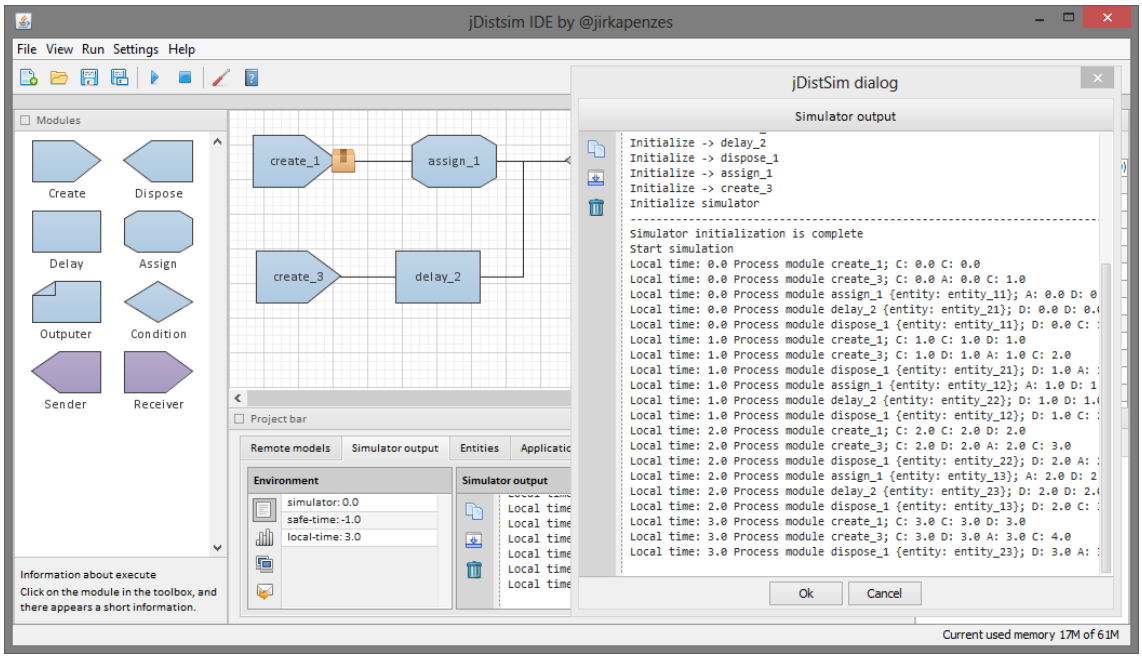
Příloha F – Přiložené CD

Na přiloženém CD se nachází diplomová práce v elektronické podobě a výsledná aplikace. Jsou přiloženy kompletní zdrojové kódy aplikace jDistsim IDE. Aplikace je obsažena i ve zkompilovaném (spustitelném) tvaru – formát jar.

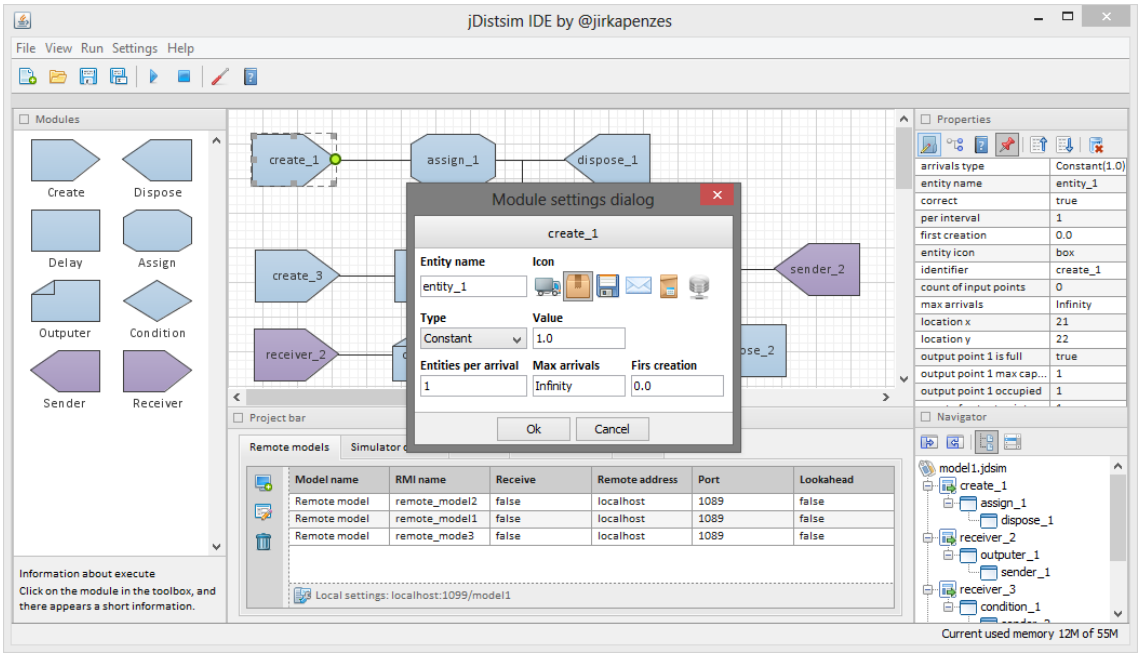
Příloha G – Screenshoty aplikace jDistsim IDE



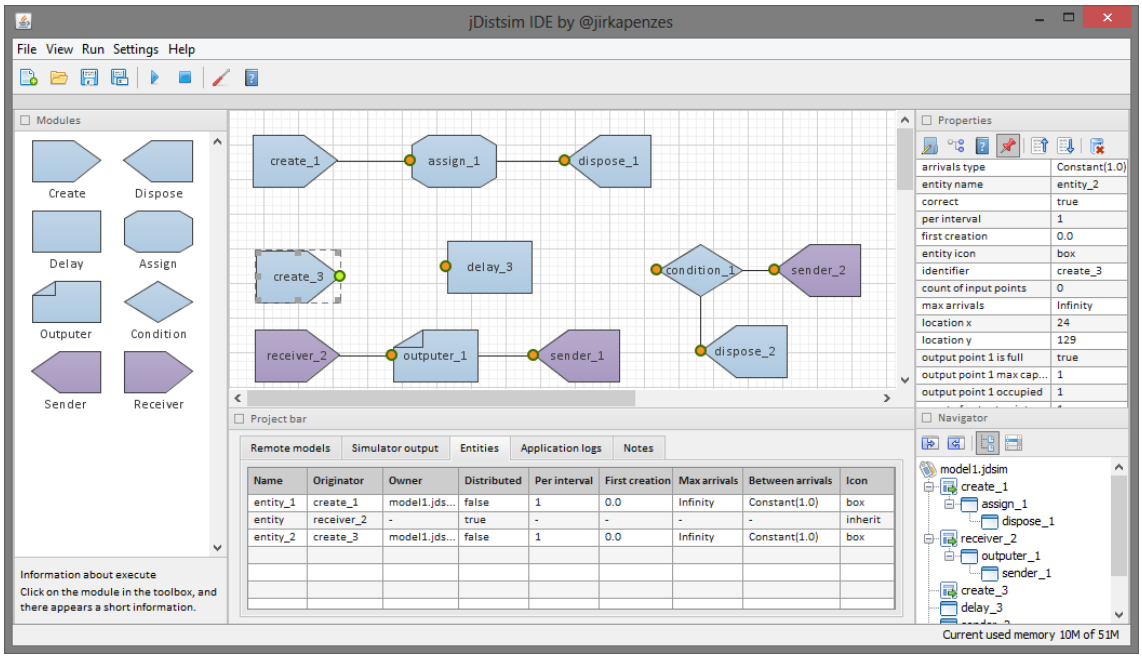
Obrázek 69 – Screenshot 1 – jDistsim IDE
Zdroj: Vlastní zpracování



Obrázek 70 – Screenshot 2 – jDistsim IDE
Zdroj: Vlastní zpracování



Obrázek 71 – Screenshot 3 – jDistsim IDE
Zdroj: Vlastní zpracování



Obrázek 72 – Screenshot 4 – jDistsim IDE
Zdroj: Vlastní zpracování