Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Softwarový nástroj pro konfigurování distribuovaných simulačních modelů využívajících webovou simulaci

Bc. Štěpán Karták

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. Štěpán Karták**
Osobní číslo:	**I11387**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Softwarový nástroj pro konfigurování distribuovaných simulačních modelů využívajících webovou simulaci**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V úvodní části diplomové práce je nutné provést přehled problematiky webové simulace se zaměřením na problematiku síťové komunikace v prostředí programovacího jazyka Java a v technologii Java applet. Pozornost bude věnována také bezpečnostním opatřením webových prohlížečů.
Primárním cílem práce je návrh, implementace a ověření softwarového nástroje pro konfigurování distribuovaných simulačních modelů, a to včetně výpočetní i vizualizační komponenty, využívajících webovou simulaci. Základním požadavkem je možnost spustit veškeré logické procesy nezávisle v dnes běžně používaných webových prohlížečích (Google Chrome, Mozilla Firefox a Microsoft Internet Explorer).
Navržené řešení bude otestováno na modelu vybraného typu dopravního nebo obslužného systému. Např.: simulace provozu na dálničním úseku s možností zastávky vozidla u přilehlé čerpací stanice.
Pro potřeby konfigurování distribuovaných simulačních modelů bude vyvinuto jednoduché integrované vývojové prostředí.
Pro účely implementace se předpokládá použití technologií: Java, Java applet, PHP, HTML a CSS.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

[1] FUJIMOTO, Richard M. Parallel and distributed simulation systems. New York: J.Wiley, c2000, xvii, 300 s. Wiley series on parallel and distributed computing. ISBN 04-711-8383-0.
[2] BARBOSA, Valmir C. An Introduction to Distributed Algorithms. USA, 1996. ISBN 0-262-02142-8.
[3] KAVIČKA, A., KLIMA, V., ADAMKO, N. Agentovo orientovaná simulácia dopravných uzlov. Žilina: EDIS - vydavateľstvo ŽU, 2005. ISBN 80-8070-477-5

Vedoucí diplomové práce: **Ing. Jan Hřídel**
Katedra informačních 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

Prohlašuji:

Tuto práci jsem 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. 5. 2013 Bc. Štěpán Karták

Poděkování

Rád bych touto cestou poděkoval vedoucímu mé diplomové práce Ing. Janu Hřídelovi za rady, připomínky, čas a ochotu řešit vzniklé problémy. Dále bych chtěl poděkovat své rodině a svým blízkým, kteří mi po celou dobu studia byli oporou.

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

Práce se zabývá distribuovanou simulací s přihlédnutím na nezávislost a synchronizaci vzdálených logických procesů pomocí konzervativní synchronizační metody zasílání nulových zpráv na vyžádání. Součástí práce jsou testovací konfigurovatelné logické procesy a obecné administrační rozhraní pro tvorbu konfigurovatelného distribuovaného simulačního modelu z dodaných komponent představujících logické procesy.

Klíčová slova

simulace, distribuovaná simulace, synchronizace času, logický proces, síťová komunikace, Java, Java Applet

Title

The software tool for configuring distributed simulation model using a web simulation

Annotation

This work focuses on distributed simulation with consideration of independence and synchronization of remote logical processes using the conservative synchronization methods of sending on demand zero messages. The testing configurated logical processes and general administration interface for creating configurable distributed simulation model from supplied components representing the logical processes are also part of this work.

Keywords

simulation, distributed simulation, time synchronization, a logical process, network communication, Java, Java Applet

Obsah

SEZNAM ILUSTRACÍ

SEZNAM TABULEK

SEZNAM ZKRATEK A ZNAČEK

ADT	Abstraktní datový typ
AJAX	Asynchronous JavaScript and XML
CSS	Cascading Style Sheets
DOM	Document Object Model
IPv4	Internet Protocol version 4
JNLP	Java Network Launch Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LVT	Local virtual time, lokální virtuální čas
LCC	Local Causality Constraint, podmínka lokální kauzality
OOP	Object-oriented programming, objektově orientované programování
PHP	PHP: Hypertext Preprocessor, serverový programovací jazyk
TCP	Transmission Control Protocol
UDL	User Datagram Protocol
UML	Unified Modeling Language
(X)HTML	(Extensible) HyperText Markup Language
XML	Extensible Markup Language

ÚVOD

Diplomová práce je primárně zaměřena na vytvoření obecného konfiguračního nástroje pro distribuované simulace. Celá práce je koncipována tak, aby uživatel tohoto nástroje nepotřeboval jiný softwarový produkt, kromě dnes běžně dostupného internetového prohlížeče. S přihlédnutím k tomuto požadavku jsou využity nové prvky HTML5 pro samotný konfigurační nástroj a technologie z platformy Java – konkrétně Java Applety – pro realizaci samotných logických procesů distribuované simulace. Výhody i nevýhody tohoto řešení, stejně tak zdůvodnění použití této technologie, je podrobně vysvětleno.

Pro ověření řešení bude vytvořeno několik Java Appletů, na kterých budou demonstrovány možnosti samotného konfiguračního nástroje, ale také realizace samotného distribuovaného simulačního modelu. Tento model bude představovat prvek dálniční komunikace s obslužnými jednotkami představující službu občerstvení typu *drive-in*. Pro ověření správného fungování distribuované simulace bude tento případ realizován několika způsoby.

V práci se zaměříme také na použití technologií používaných pro realizaci internetových stránek a budou popsány základní techniky realizace distribuované simulace s přihlédnutím k vybranému řešení pro realizaci této práce.

Výsledkem práce, vyjma samotné aplikace, bude manuál pro uživatele a vývojáře dalších appletů, z důvodu budoucího využití aplikace pro realizaci libovolných distribuovaných simulačních modelů.

1 SIMULAČNÍ TECHNIKY A POJMY

Simulace zkoumají objekty reálného světa s cílem:

- ověřit již existující systém,
- nebo testovat plánovaný či jinak uvažovaný systém.

Následuje výčet a vysvětlení zásadních pojmů z oboru simulace s přihlédnutím k faktu, že tato práce se týká distribuovaných simulačních systémů. Kapitola čerpá ze sylabů přednášek prof. Kavičky [1] a publikace zabývající se distribuovanou a paralelní simulací [2].

1.1 Simulační model

Termínem **model** obecně rozumíme vztah mezi modelovaným (originálním) systémem a systémem, který tento modeluje, čímž je zpravidla myšlena abstrakce modelovaného systému na systém modelující, který zachycuje vlastnosti originálního modelu, které chceme sledovat, či které jsou pro modelovaný systém definující, a to v závislosti na konkrétní úloze, ke které chceme model použít.

Platí: $\forall P_x \approx O_x$, kde $x \in \{1, \dots, n\}$, n je počet prvků systému $S_{originál}$, P_x je modelovaný prvek systému $S_{originál}$ a O_x je modelující prvek systému $S_{modelovaný}$ odpovídající P_x.

Pod termínem **simulační model** se běžně rozumí dynamický systém, ve kterém existuje zobrazení τ, tedy $\tau = \forall P_x \approx O_x$ určené v časech t_0 až t_n, tedy $\tau(t_i) = \forall P_x(t_i) \approx O_x(t_i)$, kde $t_i \in \{t_0, \dots, t_n\}$. Zobrazení τ je neklesající, tedy $\tau(t_i) \leq \tau(t_{i+1})$, a z tohoto důvodu je nutné dodržovat kauzalitu času napříč jednotlivými částmi modelu, pokud model takové obsahuje (viz kapitoly 1.7 a 1.8).

1.2 Modelování a simulace

Modelování je technika, při které nahradíme simulovaný systém jeho obrazem – **modelem** – dostaneme **simulační model**. Při modelování uplatňujeme určitou míru abstrakce simulovaného modelu v závislosti na požadavcích činnosti, pro kterou modelování provádíme.

Simulace je výzkumná metoda, která uplatňuje modelování dynamického systému za účelem experimentování a získáním údajů o původním – modelovaném – simulovaném[1] systému.

1.3 Aktivity a procesy

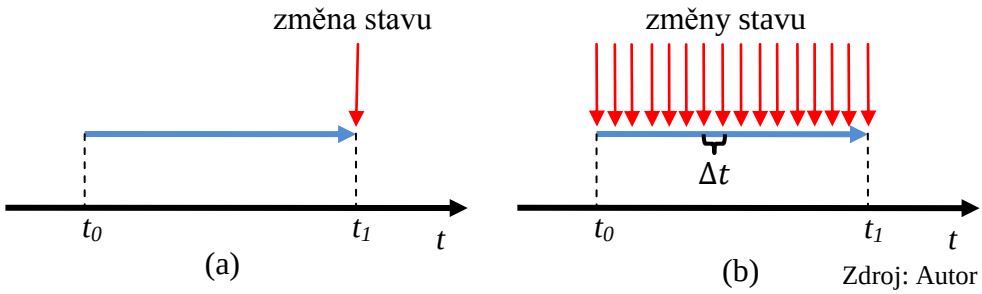
Základní hybnou silou v simulačním systému je **aktivita**. Každá aktivita je časově vymezená – můžeme určit čas zahájení aktivity a délku trvání aktivity (a tedy i čas jejího ukončení).

Rozlišujeme dva typy aktivit (viz obrázky 1a, 1b):

- **Diskrétní aktivita** mění stav systému pouze v čase ukončení svého trvání. Ukončení a změnu stavu systému nazýváme **událost**. Průběh takovéto aktivity nemusíme podrobně sledovat, zásadní pro nás je pouze okamžik události, tzn. že v simulačním modelu nemusíme sledovat aktivity, ale pouze výskyt událostí. Pokud simulační model obsahuje pouze diskrétní aktivity, pak takové simulaci říkáme **diskrétní simulace** (viz kapitola 1.4).
- **Spojitá aktivita** může měnit stav systému v průběhu času, po který je definována, a musíme tedy sledovat celé[2] časové kvantum od zahájení po ukončení aktivity. Simulační model, který obsahuje pouze spojité aktivity, nazýváme **spojitou simulací** (viz kapitola 1.5).

Proces je posloupnost aktivit, které dohromady tvoří celek. Proces je tedy vykonání činnosti s konkrétním cílem.

Aktivity a procesy představují dynamické chování sledovaného systému.



Zdroj: Autor

Obrázek 1 – (a) Diskrétní aktivita a (b) spojitá aktivita, Δt ... interval vzorkování

[1] Termíny originální, modelovaný, simulovaný systém či model se často v běžné praxi zaměňují a pro potřeby této práce mezi nimi nebude uvažován rozdíl, neboť tato práce si neklade za cíl detailně rozebírat teorii, názvosloví či matematický popis vědeckého oboru simulace a modelování.

[2] Sledovat celé časové kvantum je prakticky nemožné, vždy volíme určitou míru vzorkování intervalu času na několik časově často velmi si blízkých okamžiků, ve kterých vyhodnotíme (zpracujeme) stav, či činnost aktivity v tomto čase. Konkrétní techniky budou popsány později.

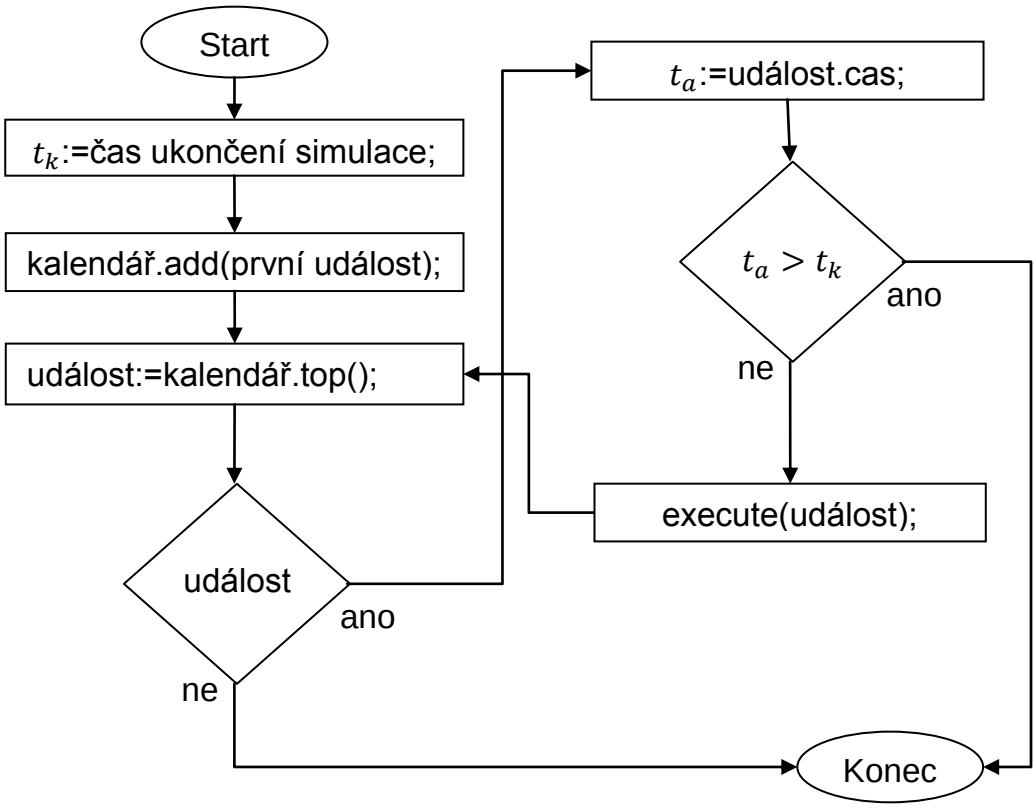
1.4 Diskrétní simulační model

Diskrétní simulační model obsluhuje pouze diskrétní aktivity. Zpracovává aktivity tak, že v okamžiku zahájení aktivity je naplánován i konec aktivity, kdy dojde k provedení všech stavových změn modelu provedených aktivitou.

Následuje přehled nejčastějších metod synchronizace a algoritmizace simulačního výpočtu. Podobně bude zmíněna metoda plánování událostí, neboť je využívána v praktické části práce.

Metoda plánování událostí

Nejrozšířenější a algoritmicky nejjednodušší způsob realizující běh simulace. Princip spočívá v plánování pouze událostí do kalendáře (nejčastěji realizováno pomocí ATD Prioritní fronta, kde za prioritu uvažujeme hodnotu časového razítka) dokud jsou zprávy v kalendáři nebo jsou v kalendáři pouze události s hodnotou časového razítka vyšší, než je časové kvantum vymezené pro simulaci. Algoritmus je zobrazen níže na obrázku 2.



Zdroj: Autor

Obrázek 2 – Algoritmus metody plánování událostí

Metoda interakce procesů

Tato metoda využívá principu metody plánování událostí, avšak aktivity sdružuje do procesů, přičemž proces nesmí být vykonán najednou. Je rozlišováno několik stavů (aktivní, ukončený, suspendovaný a pasivní) procesu. Simulační jádro pracuje a přepíná stavy procesů a podle jejich stavů s nimi pracuje a tím generuje postup aktivit v čase kupředu.

Metoda snímání aktivit

Touto metodou lze realizovat diskrétní i spojitou simulaci (případně jejich kombinaci), dále se stručně zaměříme na řešení diskrétní simulace.

Čas simulačního modelu se posunuje v každém kroku o zvolený interval a v každém kroku simulační jádro zjišťuje, zda nastaly podmínky k vykonání události v kalendáři.

Třífázová metoda

Metoda rozlišuje dva druhy aktivit:

- plánované – běžné aktivity s daným časem události,
- podmínkové – aktivity, které budou provedeny v okamžiku splnění (aktivitě) příslušných podmínek.

Po vykonání všech událostí plánovaných na aktuální čas, se zjišťuje, zda nenastaly podmínky pro některé podmínkové aktivity.

1.5 Spojitý simulační model

Tento typ simulačního modelu obsahuje pouze spojité aktivity a běžně se řeší metodou Snímání aktivit. Diskrétní část této metody je popsána v podkapitole 1.4.

Po zvolených intervalech je procházena časová osa, a pokud k příslušnému časovému okamžiku existuje spojitá aktivita (či aktivity), tak je provedena – tedy je vykonána aktivita v určitém okamžiku a jsou provedeny příslušné změny stavového prostoru simulačního modelu.

1.6 Kombinovaný simulační model

Simulační model obsahuje aktivity spojitého i diskrétního charakteru, nabízí se použití kombinace metody plánování událostí pro diskrétní aktivity a metody snímání aktivit pro spojité aktivity v intervalech mezi diskrétními událostmi, jejichž rozdíl časových razítek je

větší než nula – tedy je mezi nimi nějaký časový interval. Tento interval, pomocí metody snímání aktivit, se prochází po zvolených subintervalech (vzorkovací interval), a v každém časovém bodě se provedou příslušné aktivity a případně změna stavu modelu.

1.7 Distribuovaná simulace

Tento způsob simulace počítá s rozdělením modelu na několik výpočetních uzlů, které spolu komunikují – zasílají si **zprávy** – a dohromady simulují modelovaný systém.

Rozlišujeme dva druhy realizace:

- **paralelní simulaci** – výpočetní uzly jsou těsně spojené v rámci jednoho hardwaru,
- **distribuovanou simulaci** – výpočetní uzly jsou spojené počítačovou sítí, a právě tímto typem simulačního modelu se primárně zabývá tato práce.

Jednotlivé výpočetní uzly nazýváme logické procesy (více viz následující část 1.8).

Tento způsob simulace zavádíme z důvodu zvýšení výpočetního výkonu modelu jako celku, vzhledem k omezeným[3] možnostem samostatných výpočetních jednotek. Dalším důvodem pro zavedení distribuované simulace mohou být i finanční náklady na pořízení dostatečně výkonné hardwaru pro čistě diskrétní řešení simulace oproti soustavě nevýkonného, ale levného hardwaru. Musíme brát v úvahu, že ne každá úloha je vhodná pro distribuované řešení – viz kapitola 3.5 – testování modelové příkladu dálnice s občerstveními, kde je z tabulky 1 patrné, že distribuované řešení několikanásobně prodloužilo dobu výpočtu, neboť komunikační režie daleko přesahuje nároky na výpočetní výkon jednotlivých logických procesů.

1.8 Logický proces

Logickým procesem (dále označené jako LP) rozumíme jednu výpočetní jednotku distribuovaného simulačního modelu (kapitola 1.7), která je běžně realizována jako diskrétní, spojitý či kombinovaný simulační submodel (kapitola 1.6), který je informován o okolních logických procesech v rámci distribuovaného simulačního modelu, ke kterému přísluší. Pro uvažovaný LP_A se rozumí jako okolní logický proces – označme LP_B – ten, od kterého logický proces LP_A přijímá zprávy (LP_B zasílá zprávy LP_A). Informovanost o okolních

[3] Nejvýkonnější procesor na trhu v roce 2010 byl procesor IBM z196 taktovaný na 5,2 GHz, více viz *IBM Unveils World's Fastest Microprocessor*. [online]. [cit. 2013-04-21]. Dostupné z: http://www.foxnews.com/scitech/2010/09/01/ibm-unveils-worlds-fastest-microprocessor/

logických procesech je pro logický proces / diskrétní simulační model zásadní z důvodu zachování lokální kauzality (viz následující podkapitola 1.9).

Každý logický proces vystupuje jako zcela samostatná jednotka v rámci celé distribuované simulace.

Z pohledu distribuované simulace zavádíme pojem **lokální virtuální čas** (LVT). Každý logický proces pracuje s vlastním simulačním časem (LVT), který může, a často je, v celém simulačním modelu unikátní, a právě z tohoto důvodu zavádíme techniky synchronizace času logických procesů, neboť čas a sekvenční (podle času výskytu) zpracování událostí v rámci distribuovaného modelu je zásadní pro správný běh simulace.

1.9 Podmínka lokální kauzality

Podmínka lokální kauzality (LCC) zaručuje stav, kdy každý logický proces zpracovává zprávy v neklesajícím pořadí dle jejich časových razítek. Tato podmínka je zásadní pro správnou činnost simulačního modelu jako celku a její použití je vždy[4] vyžadováno vzhledem k charakteru času, jakožto veličiny neklesající a vždy plynoucí kupředu.

Existuje řada technik, jakým způsobem tento princip realizovat. Tyto techniky jsou popsány v kapitole 2.

[4] Lze samozřejmě realizovat simulační model nevyžadující podmínku lokální kauzality, avšak takovýto simulovaný systém nebude mít předobraz v reálném světě a lze o takové úloze prohlásit, že to není standardní a běžně užívaný typ vzhledem k časté vazbě simulovaného systému na vzor z reálného světa.

2 METODY SYNCHRONIZACE

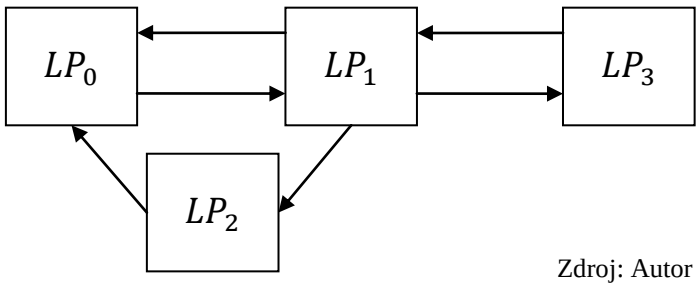
V této kapitole jsou zmíněny principy algoritmů používaných pro synchronizaci převážně logických procesů v rámci distribuovaných simulačních modelů. Jednotlivé metody jsou popsány na základě literatury [1, 2].

2.1 Konzervativní metody synchronizace

Konzervativní metody striktně dodržují podmínku lokální kauzality (viz kapitola 1.9). Obecně lze říci, že v simulovaném systému se nesmí objevit zpráva s časovým razítkem menším, než je nejmenší lokální simulační čas ze všech logických procesů.

Jedno z možných řešení je synchronizační technika zasílání nulových zpráv na vyžádání s uplatněním výhledu. Tento algoritmus je podrobně rozebrán v kapitole 3, neboť je použit v demonstračních výpočetních jednotkách modelového příkladu. Další možnosti jsou stručně popsány dále v této kapitole.

Na obrázku 3 je uveden iluzorní příklad. Logické procesy zasílají aktivity/**události** (tato práce používá výhradně diskrétními aktivity, a proto v dalším textu budeme hovořit přímo o událostech) ve směru šipek.



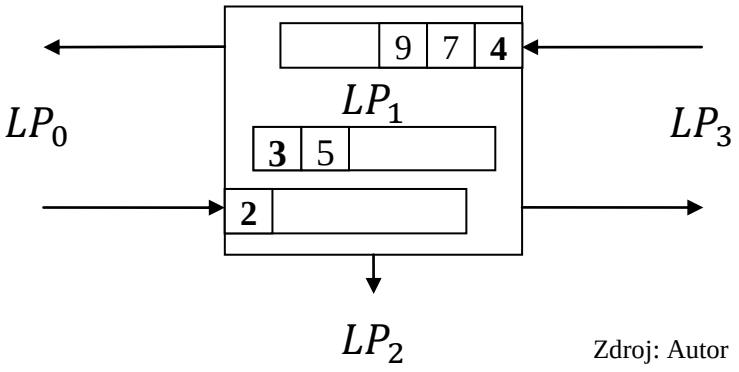
Zdroj: Autor

Obrázek 3 – Ilustrační příklad soustavy logických procesů

Podmínka lokální kauzality času LP je zajištěna, pokud do LP nevstupují aktivity s nižším časovým razítkem, než je LVT logického procesu. Z tohoto důvodu každý logický proces zajímá LVT pouze jeho nejbližších okolních logických procesů (to jsou ty logické procesy, se kterými může přímo komunikovat – dále označeno jako okolní LP), od kterých přijímá zprávy. V našem případě (obrázek 3) LP_3 zajímá pouze LVT LP_1,. LP_1 zajímá stav již více procesů LP_0 a LP_3. LVT samotných procesů přitom nemusíme nutně znát, stačí nám vědět nejnižší čas, ve kterém mohou být provedeny aktivity. Tento časový údaj lze vyčíst z aktivit příchozích logickému procesu, neboť pokud vezmeme v úvahu časovou kauzalitu provádění

aktivit, je jasné, že pro příklad do LP_3 od jeho okolního procesu LP_1, nemůže přijít aktivita s nižším časovým razítkem, než je nejnižší časové razítko aktivit zaslaných od LP_1. Pokud od každého okolního LP známe alespoň jeden časový údaj (aktivitu s časovým razítkem), můžeme podle toho určit, jaké zpracování aktivit LP bude bezpečné – známe vlastní LVT a nejnižší časový okamžik doručení zpráv od okolních LP.

Toto lze například zajistit udržováním speciálních front aktivit od okolních LP – ukázka viz obrázek 4.



Zdroj: Autor

Obrázek 4 – Ukázka LP s frontami pro každý okolní LP

Máme tedy vyřešen problém se zjištěním bezpečného stavu ve chvíli, kdy máme dostatek aktivit od okolních LP. Na obrázku 4 je uveden stav, kdy je bezpečné vykonat aktivity s časovými razítky 2 a 3 v tomto pořadí.

Musíme ještě vyřešit stav, kdy nemáme ve frontách okolních procesů žádné aktivity. Jednou z možností je zasílání nulových zpráv.

Zasílání nulových zpráv

Tato technika má mnoho variant. Nejjednodušší řešení, jak se vyhnout stavu, kdy okolní LP nemají informace o stavu LVT (existuje alespoň jedna prázdná fronta), je po každém provedení aktivity zaslat logickým procesům (kterým LP zasílá události) zprávu (nulová zpráva), která bude obsahovat pouze časovou informace, nikoliv nějakou událost. Tím je zaručeno, že procesy budou vždy mít ve frontě nějakou zprávu.

Pokud vznikne cyklus, ve kterém si budou LP zasílat LVT bez pokroku (každý bude čekat na nějakou konkrétní aktivitu) – nenulovou zprávu – algoritmus selže.

Z tohoto důvodu a z důvodu zvýšení paralelizace výpočtu zavádíme tzv. výhled.

Výhled logického procesu

Výhled LP, označme L (*lookahead*) je časový interval, po který neočekáváme žádnou novou aktivitu, která by mohla být zařazena do fronty.

Možné způsoby, jak určit výhled, jsou popsány v části 3.1.

Dolní hranici časových značek (LBTS) udává minimálně bezpečný čas, po který se v LP nevyskytne žádná nová aktivita, a tedy po tuto dobu můžeme bezpečně zpracovávat události, a právě to zvyšuje paralelismus simulace, neboť nemusíme čekat na překonání konkrétní hodnoty LVT.

Spočítáme takto:

$$LBTS = LVT + L, \text{ kde } L > 0$$

Nulové zprávy s výhledem

Místo zasílání LVT nulovými zprávami budeme zasílat hodnotu LBTS, tedy čas, pokud je jisté, že od okolního LP nepřijde žádná událost. Zároveň je zajištěn pokrok, neboť další naplánovatelná aktivita je v čase větším, než je LVT, a tedy postupujeme v čase (a zároveň simulačním výpočtu) kupředu.

Zatím zasíláme nulové zprávy v každém kroku, kdy byla vykonána určitá událost na LP všem okolním logickým procesům, kterým zasíláme události. Výsledkem je, že se v rámci distribuovaného modelu generuje značné množství zpráv, často zcela zbytečně. Například ve chvíli, kdy (dle obrázku 4) provedeme aktivity 2 a 3, tak vzniknou 4 nulové zprávy (2 aktivity × 2 LP). Pokud ale zároveň každá aktivita rozešle sama aktivity pro dotyčné procesy, pak jsme zaslali minimálně jednu sadu nulových zpráv zbytečně. Tento problém řeší algoritmus **zasílání nulových zpráv na vyžádání s uplatněním výhledu**, který je podrobně popsán v kapitole 3.

Dalším způsob, jak řešit synchronizaci konzervativní metodou, je použití bariér.

Bariérová synchronizace

Pro množinu LP nastavíme v budoucnosti časovou hranici bezpečných aktivit – bariéru. Logické procesy provedou všechny bezpečné události (tj. takové, které nemohou porušit

LCC, jejich činnost končí před bariérou) a zůstávají v blokovaném stavu až do doby, dokud tak neučiní všechny LP.

Ve chvíli, kdy všechny LP skončily činnost (nejsou k dispozici žádné události ke zpracování), tedy dosáhly bariery – jsou blokované a můžeme vytyčit další hranici – novou bariéru v budoucím čase a všechny LP spustit. Tento způsob se opakuje.

Tato koncepce je realizována celou řadou algoritmů, kterými se ale více nebudeme zabývat:

- **centralizovaná bariéra** – je zde centrální prvek, který řídí (a je informován o stavu dosažení bariéry všemi LP) a udržuje informace o bariéře,
- **stromová bariéra** – LP jsou uspořádány do stromu, LP na úrovni potomků, v tomto uspořádání, informují o dosažení bariéry své rodiče,
- a další.

2.2 Optimistické metody synchronizace

Narušení lokální kauzality dat je povoleno, ale je identifikováno, a ve chvíli kdy k tomuto stavu dojde, je nutné tento neplatný stav nějakým způsobem napravit. Jedním z možných algoritmů, z kategorie optimistických metod, je TimeWarp algoritmus.

TimeWarp algoritmus zpracovává aktivity do doby porušení lokální kauzality jako běžné algoritmy pro nedistribuované simulace, avšak ve chvíli, kdy je identifikován problém, je nutné se vrátit do stavu před příchodem zprávy s nižším časovým razítkem, než je aktuální lokální čas. Vrátit se (*rollback*) ke stavu v určitém čase, nepředstavuje větší[5] problém.

Případně se zavádí pojem **Kombinované metody synchronizace**, kde se používají předchozí uvedené metody konzervativní a optimistické najednou.

2.3 Porovnání obou přístupů

Při výběru nejvhodnějšího řešení je vždy brát v úvahu povahu simulačního modelu.

[5] Nejjednodušší řešení je ukládat celý stavový prostor logického procesu před provedením každé aktivity, což z podstaty problému není problém. Problémem tohoto řešení bude časem značná datová náročnost, avšak zde již záleží na konkrétní implementaci a potřebách modelu. Možné řešení je ukládat pouze změny, a z nich v okamžiku *rollbacku* sestavit stavový prostor logického procesu k určitému času.

Z uvedených principů můžeme odvodit, že:

- **Konzervativní metody**:
  - umožňují menší míru paralelismu (často čekají na synchronizaci),
  - mohou mít menší komunikační režii (značně závisí na konkrétním použitém řešení).

- **Optimistické metody** oproti tomu:
  - mají vyšší míru paralelismu, a lépe tak umožňují realizovat běh simulace v reálném časem,
  - ale musí řešit *rollback*, což může být náročný úkon vzhledem k možnému šíření chyb do dalších logických procesů.

3 SYNCHRONIZAČNÍ TECHNIKA ZASÍLÁNÍ NULOVÝCH ZPRÁV NA VYŽÁDÁNÍ S UPLATNĚNÍM VÝHLEDU

Tato technika patří do kategorie konzervativních metod (podkapitola 2.1), které vyžadují podmínku LCC. Tato kapitola čerpá především ze sylabů prof. Kavičky [1].

Každý logický proces může pracovat, pokud tím neohrozí podmínku LCC, a aby toto mohl LP zajistit, musí vědět, jaké jsou lokální virtuální časy (LVT) na LP, se kterými komunikuje – obecněji – od kterých mu může přijít požadavek na vykonání události.

3.1 Výhled

Výhled LP je nutné určit:

- před spuštěním distribuované simulace, pokud se interval LBTS nebude měnit,
- nebo v případě dynamického určování LBTS při běhu simulace podle požadavků simulovaného problému.

Samotnou hodnotu výhledu L určíme s ohledem na (příklady založeny na modelovém příkladu, kapitola 7):

- vlastnosti simulovaného systému (jsou známy určitá fakta o řešeném problému, které můžeme použít, např. minimální čas mezi příjezdy vozidel),
- nepřerušitelnost procesů či aktivit (aktivita průjezdu vozidla přes sledovaný úsek dálnice),
- jistá míra tolerance nepřesnosti LCC – následně s tímto stavem musíme při běhu simulace počítat,
- odhad či znalost trvání aktivit (známe minimální čas aktivity obsloužení vozidla u občerstvení).

3.2 Popis algoritmu

Algoritmus bude popsán z pohledu LP_0, tvořící distribuovanou simulace logických procesů LP_0 až LP_n. LP_x je množina logických procesů zasílajících události LP_0. Vycházíme z principů popsaných v podkapitole 2.1.

Inicializace

Před začátkem spuštění distribuované simulace musí LP_0 zjistit informace, či jinak být informován o svém okolí – primárně nás zajímají LP_x, které budou zasílat události LP_0.

Pro každý LP z LP_x si připravíme frontu (kalendář – obvykle implementován jako ADF Prioritní fronta) příchozích událostí. Ty fronty jsou nutné[6] pro zjištění informovanosti LP_0 o LVT okolních LP_x, neboť z pohledu zachování kauzality času můžeme prohlásit, že do LP_0 nepřijde od LP_1 ($LP_1 \in LP_x$) zpráva s menším časovým razítkem, než je nejnižší (v kalendáři na prvním místě) časové razítko.

Máme tedy jeden kalendář samotného LP_0, a pak další kalendáře pro každý LP_x.

Zjištění LBTS

Logický proces (LP_0) by měl znát dolní hranici časových značek, které se mohou vyskytnout v rámci jeho činnosti. LBTS musí být větší, než LVT LP_0. Tuto informaci LP nemusí zjišťovat před spuštěním simulace, stačí kdykoliv, kdy je LP_0 žádán o zaslání nulové zprávy. Pokud se výhled nebude měnit, pak je zbytečné tímto výpočtem zatěžovat LP_0 při každém dotazu.

Spuštění distribuované simulace

Ve chvíli, kdy všechny znají své okolí, může být simulace spuštěna.

Běh simulace

Před každým výběrem události s nejnižším časovým razítkem z kalendářů musí LP_0 zajistit, že tuto událost může provést bezpečně bez narušení LCC. Jinými slovy, můžeme provést pouze takovou událost, u níž jsme si jisti, že do LP_0 nepřijde žádná zpráva s nižším časovým razítkem, než má aktuální událost čekající na zpracování (ta s nejnižším časovým razítkem z kalendářů). A právě z toho důvodu jsme při inicializaci zavedli fronty / kalendáře událostí. Pokud máme v každém kalendáři (vyjma kalendáře vlastního LP_0) alespoň jednu událost, můžeme prohlásit, že žádná událost s nižším časovým razítkem, než je nejnižší časové razítko v kalendářích okolních logických procesů, již nepřijde (vzhledem ke kauzalitě času) a můžeme provést aktuální událost.

Pokud při procházení kalendářů zjistíme stav, že nemáme k dispozici žádnou událost (kalendář je prázdný), zašleme požadavek na příslušný LP_y, kde $LP_y \in LP_x$ s žádostí o zprávu o jeho čas LBTS (odtud pojem **nulová zpráva** – nebude nést žádnou informaci o události). Poté čekáme, dokud nebude ve všech frontách LP_x alespoň jedna zpráva (očekáváme

[6] Lze samozřejmě implementovat více způsoby, popisujeme princip.

odpovědi od okolních LP_x, kterým jsme poslali žádost o nulovou zprávu), a poté provedeme aktuální čekající událost.

Ukončení

Činnost LP_0 ukončíme podle potřeby simulace.

3.3 Slabina algoritmu

Algoritmus v základní podobě, tak jak je popsán v kapitole 2.1, má zásadní slabinu. Pokud nějakým způsobem centrálně nekorigujeme velikosti výhledu v rámci okolí logického procesu zasílajícího události, můžeme se dostat do stavu, kdy logický proces, který si vyžádal nulovou zprávu, takovou zprávu nepřijme, protože časové razítko bude nižší, než LVT logické procesu. Tím bude porušeno LCC a na výsledek takové simulace nebude mít význam.

Kdy se tak může stát

Obecně lze říci (zjištěno experimentálně na modelovém příkladu z kapitoly 7), že pokud spolu dva LP komunikují (zasílají si události navzájem), pak častěji tento problém vyvstane u LP s nižším výhledem. Závisí také na časech (posloupnosti) spuštění logický procesů. Architektura (vzájemné propojení logických procesů) celého distribuované řešení také hraje roli.

Řešení tohoto problému je popsáno v následující podkapitole 3.4.

3.4 Implementovaný algoritmus zasílání nulových zpráv a odpovídání

Ukázka průběhu činnosti dvou logických procesů LP_0 a LP_1. LP_0 přijímá události od LP_1. Algoritmus je publikován v článku v časopise Elektrorevue [3] – autoři Ing. Hřídel a Bc. Karták. Níže uvedený algoritmus je rozepsán podrobněji.

Logické procesy implementují možnost vykonat událost LBTS, tato událost nemá žádný význam pro samotnou simulaci, slouží pouze pro zjednodušení a sjednocení činností (události, nulové zprávy, případně cokoliv jiného) logického procesu.

Zavedeme pojem „s*ervice* události“, tyto události jsou vykonány okamžitě, nejsou zařazeny do kalendáře.

1. LP_0 nemá v kalendáři události od LP_1.

2. LP_0 odešle událost LBTS (typ *request*), zpráva označena jako *service*, odesílá i vlastní $LP_0.LVT$.
3. LP_0 čeká.
4. LP_1 přijme tuto událost LBTS.
5. LP_1 událost LBTS nezařadí (je *service*) do fronty, ale vykoná se okamžitě po skončení aktuální činnosti/události, kterou LP_1 vykonává.
6. LP_1 vykoná událost LBTS:
    a. Provede výpočet $LP_1.LVT + LP_1.L$ (známá hodnota pro LP_1) = $LP_1.LBTS$
    b. Pokud je $LP_0.LVT$ (odesláno v události) $> LP_1.LBTS$, pak by LP_0 nepřijal takovou událost / odpověď, tj. nulovou zprávu (došlo by k porušení LCC), potom:
        i. SC2.LBTS je zařazena do $LP_1.Queue$ (hlavní kalendář událostí) na čas $LP_0.LVT$, přičemž se odesilatele nepovažuje LP_0, ale samotný LP_1.
    c. Jinak je odeslána událost LBTS (typ *response*, s časem $LP_1.LBTS$) na LP_0 (již není označena jako *service* – chceme, aby se zařadila do fronty – právě na to čeká LP_0).
7. LP_1 pokračuje ve své činnosti.
8. LP_0 zařadí do fronty událost LBTS, kterou odeslal LP_1 (krok 6c).
9. LP_0 již zná dolní ohraničení času $LP_1.LBTS$ a může provést naplánované události před přijatou zprávou LBTS.

3.5 Porovnání techniky v různých prostředích a použití

V této části porovnáme na modelovém příkladu z kapitoly 7 (měříme pouze LP Dálnice a LP Občerstvení A, které samy o sobě tvoří dostačující distribuovaný simulační výpočet) čtyři případy řešení – nedistribuované řešení (případ A) a tři distribuované případy. Výsledky jsou k vidění v tabulce 1.

Ve všech případech sledujeme primárně vlákno, které vybírá z kalendáře události a dané události vykonává, pokud nečeká na synchronizaci. V rámci jedné realizace výpočtu LP zpravidla běží i další vlákna, ale ty nyní neuvažujeme, nicméně jejich režie samozřejmě také hraje roli na čase výpočtu.

Podmínky měření

LP Dálnice je uveden jako LP_0 a LP Občerstvení A jakoLP_1. Výhled LP_0 je 3000 ms, a výhled LP_1 je 5000 ms. Výhled je určen na základě nejkratší známé prodlevy mezi událostmi.

Testovací sestava viz příloha A – všechny případy jsou testovány při stejné zátěži 50% CPU.

Úroveň logování chyb 4, tzn. pouze chyby – žádný zápisy na disk a konzoly, ponechány pouze dodatečné výpisy na konzoly pro sledování výsledků měření, které nemají ekvivalentní výskyt v běžně logovaných datech.

Případ A: Jedno-vláknový přístup

Události obou logických procesů (Dálnice a Občerstvení) jsou zpracovávány jedním logickým procesem – nejedná se o distribuovanou simulaci.

Případ B: Každý LP odpovídá jednomu vláknu v rámci jednoho programu

Situace, kdy je distribuovaný výpočet realizován dvěma nezávislými vlákny, ale v rámci jednoho programu. Tato verze nepoužívá žádný centrální prvek – server, jako případ C – vlákna komunikují přes referenci jednoho na referenci druhého. Oproti případu A vidíme přibližně trojnásobné zpomalení, které je dáno především potřebou synchronizace vláken a zvýšeným počtem zpráv o události LBTS.

Případ C: Každý LP běží v samostatných aplikacích samostatně a komunikují přes síť v rámci domény localhost – lokálního počítače

Toto je standardní stav, kdy je každý logický proces prováděn samostatným programem. Síťová komunikace je realizována v rámci jednoho počítače přes adresu localhost (IPv4 127.0.0.1) – v tomto případě nesledujeme prodlevu samotné síťové komunikace mimo počítač. Sledujeme pouze zvýšenou režii předávání informací po síti – především čekání na spojení a případně odpovědi, které jsou realizovány přes serverový prvek, který se také podílí na zpomalení výpočtů.

Případ D: Každý LP běží v samostatných aplikacích na rozdílných počítačích přes síť

Logické procesy jsou prováděny na vzdálených počítačích v rámci lokální počítačové sítě s velmi nízkou latencí (ping < 0 ms). Server je umístěn na totožném počítači, kde běží LP_0. Toto je případ reálného použití distribuované simulace.

Shrnutí

Modelový případ, na kterém testujeme distribuovanou simulaci, a především synchronizační technika nulových zpráv na požádání, nejsou vhodné úlohy pro distribuované řešení. Konzervativní úloha sama o sobě výpočet pouze zpomaluje.

Tento stav je dán čistě charakterem úlohy, neboť se jedná o simulaci, kde jsou výpočty velice jednoduché a početně nenáročné – není důvod zavádět distribuovaný výpočet, ale také proto, že zpracovávané události (defakto předání entit – vozidel) jsou předávané z LP_0 (generátor entit) na LP_1 s relativně nízkou frekvencí, a LP_1 z toho důvodu velice často volá zprávy LBTS – požaduje synchronizaci času. Úplně stejně se chová (často vyžaduje synchronizaci času) LP_0, neboť zpracovávané entity (vozidla) jsou vraceny z LP_1 na LP_0 se stejnou nízkou frekvencí.

Tato úloha není vhodná pro řešení distribuovanou simulací.

Tabulka 1 – Porovnání užití synchronizace času

Parametr / Případy	**Případ A**	**Případ B**		**Případ C**		**Případ D**	
	$LP_0 + LP_1$	LP_0	LP_1	LP_0	LP_1	LP_0	LP_1
Počet vykonaných událostí	2424	3424	2520	3618	2440	4212	1987
Délka výpočtu	3 s	10 s		20 s		25 s	
Volání činnosti LBTS	-	452	968	566	1062	711	1557
Vykonání činnosti LBTS / Request	-	1112	462	1162	584	601	434
Vykonání činnosti LBTS / Response	-	452	961	554	1011	670	1474
Zpomalení oproti případu A	1 x	cca 3 x		cca 7 x		cca 8 x	

4 SOFTWAROVÝ NÁSTROJ PRO KONFIGURACI DISTRIBUOVANÉ SIMULACE

V dalším textu budeme tento nástroj označovat jednoduše **Administrace**. Ověření správnosti vytvořených sestav (konfigurací) distribuované systému je rozebráno v kapitole 7 na modelovém příkladu.

4.1 Požadavky

Základní prvky Administrace:

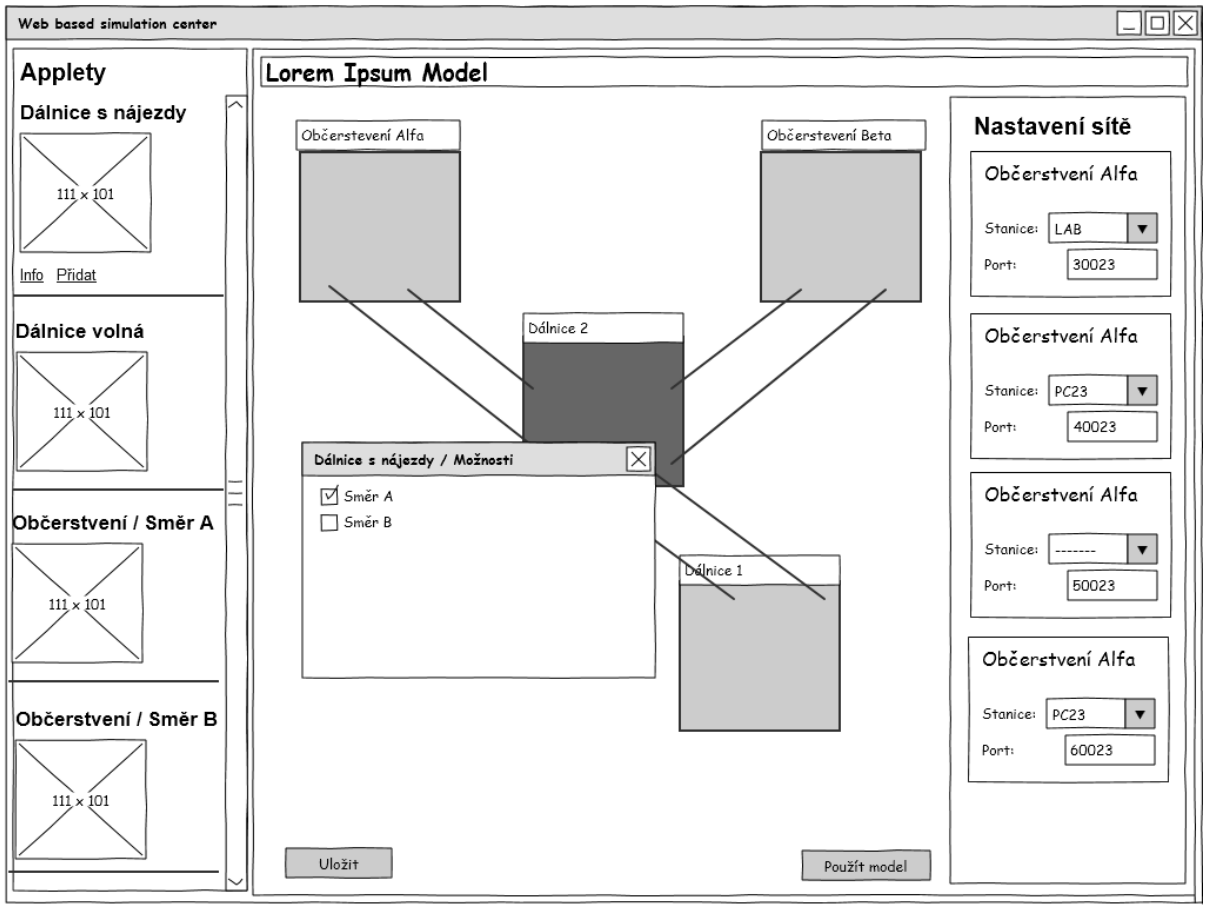
- **Simulace** – vizuální znázornění a editace distribuovaného modelu.
- **Sezení** simulace – konkrétní realizace běhu nakonfigurovaného simulačního modelu.
- **Applety** – výkonné jednotky simulace realizované pomocí technologie Java Applet (viz část 5.2.1).
- **Pracovní stanice** – jednoduché záznamy pro organizaci sezení simulace.
- Správa **serveru**.

Administrace umožní:

1. Konfigurovat distribuovaný simulační model jako soustavu Java Appletů libovolně (v rámci nakonfigurovaných omezení) v editoru ve webovém prohlížeči. Konkrétně:
    - validace distribuovaného modelu,
    - globální nastavení celého distribuovaného modelu,
    - nastavení jednotlivých logických procesů včetně nastavení síťových parametrů.
2. Spravovat (přidávat, měnit, mazat) logické procesy (Java Applety).
3. Definovat a spravovat pracovní stanice, které bude možné přiřadit k nakonfigurovaným appletům, pro lepší organizaci sezení.
4. Zorganizovat sezení nad konkrétním distribuovaným modelem.
5. Jednoduchá správa serveru – centrálního komunikačního směrovače.

4.2 Návrh

Návrhy[7] rozhraní Administrace jsou uvedeny v příloze D, zpravidla se jedná formulářové stránky dnes běžného typu. Specifickou stránkou bude samotný návrhář konfigurace simulací – dále jako Designer – viz obrázek 5.



Zdroj: Autor

Obrázek 5 – Návrh Designeru

Designer je rozdělen do třech hlavní částí:

- levý panel s dostupnými Applety,
- prostřední část se samotnou návrhářskou plochou – realizováno novým a moderním prvkem HTML5 <canvas>,
- pravý panel se síťovým nastavením vytvořených modelů v rámci simulace.

Pro zvýšení efektivity realizace sezení simulace bylo rozhodnuto vytvoření záznamů pracovních stanic s e-mailovou adresou, které bude možné přiřadit logickým procesům

[7] Návrhy – drátěné modely (wireframes) – jsou vytvořeny ve volně dostupném nástroj Pencil, licence GNU Public Licence 2, více informací viz *Pencil Project* [online]. 2012 [cit. 2013-05-02]. Dostupné z: http://pencil.evolus.vn/

a informace o sezení automaticky rozeslat e-mailem. Stejně tak budou jednotliví experimentátoři (pracovníci vázaní na pracovní stanice) informováni o aktuálních požadavcích na manipulaci s jim přiřazenými modely (logickými procesy).

Detailní informace o spouštění simulace a o práci a možnostech Administrace jsou k dispozici v dokumentaci v příloze Cdole – Manuál správce Administrace.

5 TECHNOLOGIE ZVOLENÉ PRO REALIZACI

Tato kapitola rozebírá zvolené technologie a důvody, proč byly vybrány pro realizaci. Konkrétní implementace a nečekané problémy zvolených řešení jsou popsány v kapitole 6.

5.1 Požadavky

Požadavky vycházejí z požadavků na výsledný softwarový nástroj (viz podkapitola 5.1). Pro realizaci administračního rozhraní bylo zvoleno čisté[8] HTML řešení využívající nejnovější prvky standardu HTML5[9], na straně serveru realizováno technologií PHP. Data na straně serveru jsou ukládány do databáze MySQL a do textových souborů XML.

Vzhledem ke značným omezením možností síťové komunikace v HTML (používá se JavaScript), bylo pro výpočetní jednotky distribuované simulace zvoleno řešení Java Appletů.

Jednotlivé technologie s popisem použití jsou v této kapitole podrobně rozepsány.

5.2 Technologie a programovací jazyk Java

Programovací jazyk Java je jedním ze souboru mnoha technologií označovaných jako platforma Java[10], realizovaných právě programovacím jazykem Java. Syntaxe jazyka vychází z jazyk C/C++, přičemž odstraňuje řadu pokročilých konstrukcí jazyků C a přidává vlastní řešení, pro ulehčení práce tvůrci – programátorovi.

Jazyk Java je jeden z nejrozšířenějších jazyků, převážně z důvodů:

- multiplatformnost,
- jednoduchý syntaxe (oproti C/C++[11]),
- open source licence.

[8] Čisté HTML(5) řešení by nebylo možné použít v komerčním nástroji, protože obecně se nelze spolehnout na fakt, že všichni uživatelé používají nejnovější verze internetových prohlížečů, protože některé prvky, které se v administraci používají, nejsou ve starších verzích prohlížečů dostupné.

[9] HTML5 je obecně používaný termín zahrnující nové prvky značkovacího jazyka HTML(5), jazyka pro popis zobrazení HTML spadajících do CSS3 a nových možností skriptovacího jazyka pro stranu klienta JavaScriptu.

[10] Zahrnuje technologie Java Standard Edition (Java SE) a další, včetně pro tuto práci zásadní, Java Applety (Java Applets).

[11] Java nepracuje s ukazateli – nahrazeny referencemi, automatická správa paměti, neexistence vícenásobné dědičnosti, jednodušší datový typy (např. neexistence *unsigned* typů čísel), apod.

Oproti těmto výhodám je výsledná Java aplikace znatelně pomalejší, než nativní kódy z důvodů:

- zavedení multiplatformnosti neumožňuje[12] kompilaci kódu do strojového kódu, používá se tzv. bytecode[13] (popsáno níže),
- nejprve načtení běhového prostředí – Java Virtual Machine (viz níž), které realizuje samotný běh programu – to zvyšuje režii běhu algoritmu a především zpomaluje start aplikace.

Java Byte Code

Jedná se o zdrojový kód (často používaný termín mezikód) pro Java Virtual Machine (bude zmíněno níž). Je to zparsovaný kód (v programovacím jazyce Java) do binárního formátu, který nelze vykonat / spustit, ale lze například dále upravovat. Kód programů napsaných v Javě je v této fázi multiplatformní a distribuovatelný uživateli pro další využití.

Soubory mají koncovku *.class*.

Java Virtual Machine (JVM)

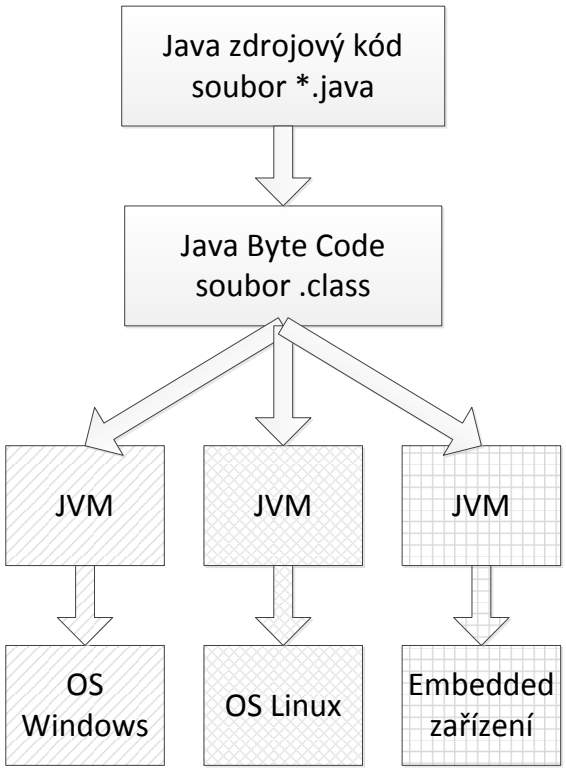
Běhové prostředí pro aplikace psané v Javě[14] se jmenuje Java Virtual Machine, jehož primárním úkolem je zprostředkovat virtuální stroj, překonávající rozdíly mezi odlišným hardwarem. JVM zkompiluje univerzální Java Byte Code do kódu (nemusí se nutně jednat o strojový kód) příslušného ke konkrétnímu hardwaru (tzn. just in time compilation) a tento kód JVM vykonává[15], přičemž dále poskytuje programu integrované datové struktury a funkce (Java API), čím je realizován multiplatformní přístup platformy Java.

JVM a Java API je distribuováno Java Runtime Environment (JRE).

[12] Způsoby, jak zkompilovat kód psaný v Javě, existují a mají použití u aplikací, které jsou zvlášť zaměřené na výkon. Kompilace Javy zachází nad rámec této práce a dále jí nebudeme uvažovat.
[13] Java používá konkrétní implementaci bytecodu Java Byte Code.
[14] JVM zpracuje jakýkoliv Java Byte Code, nemusí se nutně jedna o mezikód jazyka Java, avšak to je standardní použití.
[15] V současné době se používá složitější řešení většinou zahrnující analýzu bytecodu a následnou optimalizaci. Pro příklad uveďme technologii HotSpot compiler, která (krom jiného) analýzou bytecodu zjistí často používané části kódu a ty převede pro zrychlení běhu do strojového kódu počítače.

Evoluce programu psaného v Javě je zobrazena na obrázku 6. Na obrázku vidíme zdrojové soubory *.java* a *.class* společné pro rozdílené OS (či zařízení[16]), které jsou pro konkrétní OS realizovány pomocí pro zařízení příslušných JVM.

Speciální případ užití JVM je použití Java Appletu (viz následující část 5.2.1).



Zdroj: Autor

Obrázek 6 – Java Virtual Machine a Java program

JAR archiv

Soubor Java Archive (JAR) slouží především jako kontejner pro více souborů, tvořících celek – ve většině případů spustitelná aplikace, ale není to pravidlem. JAR soubor primárně obsahuje *.class* soubory bytecodu a soubor s manifestem *manifest.mf*, který slouží k dodatečnému nastavení obsahu, a pokud JAR archiv obsahuje spustitelnou aplikace, pak i k nastavení aplikace. Do JAR souboru lze uložit jakékoliv soubory, tedy i obrázky, dokumenty, pro použití v aplikaci, kterou, jak již bylo řečeno, JAR archiv může obsahovat.

JAR soubor je vhodný pro distribuci programů psaných v Javě, to znamená, i Java Appletů.

[16] JVM je nainstalován na miliardách zařízení, především embedded charakteru.

5.2.1 Java Applety

Java applet je program, běžící v JVM jako součást internetové stránky. JVM je spuštěn jako samostatný proces mimo samotný internetový prohlížeč.

Java Applety v době svého vzniku (rok 1995, ale tento stav trval víceméně do příchodu a rozšíření HTML5 – viz část 5.4.1) značně rozšiřovaly možnosti internetového prohlížeče, oproti HTML technologiím umožňovaly:

- realizovat libovolné kresby v prohlížeči (dnes obstarává tag <canvas>),
- vzdálený přístup k hostitelskému počítači appletu,
- jednoduchou (oproti JavaScriptu) tvorbu her a jiných interaktivních prvků.

Z důvodu výskytu appletů na internetu a jejich načtení bez zásahu[17] či vědomí uživatele, se dle omezení přístupu k hostitelskému počítači JVM, rozlišují dva typy appletů:

- sandbox,
- privileged.

Privileged applety

Tyto applety jsou podepsané certifikační autoritou – realizováno jako záznam(y) v souboru *manifest.mf* v JAR archivu. Takovýto applet nemá žádná omezení, lze používat všechny funkce a postupy právě tak, jako v případě desktopové aplikace.

Sandbox applety

Tyto applety nejsou podepsané certifikační autoritou a mají několik bezpečnostních omezení[18]:

- nelze používat funkce jazyka Java, které umožňují spouštět nativní kód či jinak spouštět programy (týká se i Java programů, či dynamického načítání kompilovaných *.class* souborů bytecodu) na hostitelském počítači,
- nelze přistupovat k *SecurityManager*[19],

[17] V určitých případech je uživatel upozorněn na požadavky Java Appletu a je vyžadováno explicitní potvrzení povolení dotyčných funkcí, toto bude zmíněno v dalším textu. Více informací viz ORACLE. *What Applets Can and Cannot Do*. [online]. [cit. 2013-04-25]. Dostupné z: http://docs.oracle.com/javase/tutorial/deployment/applet/security.html

[18] Jisté položky neplatí při spouštění místního souborového systému, další možností snížení bezpečnostních omezení je použití JNLP.

- nelze přistupovat k souborovému systému, tiskárnám a schránce[20] na hostitelském počítači,
- síťové spojení lze navázat pouze s počítačem (serverem) poskytujícím samotný Java Applet a i v tomto případě je uživatel před načtením appletu vyzván, zda chce applet připojený k síti načíst.

Soubor JNLP

Tento textový soubor lze použít při distribuci Java Appletu. Je to textový XML soubor, obsahující konfiguraci a informace o zdrojích Java Appletu – především příslušné JAR soubory, rozměry appletu, název.

Při použití souboru JNLP lze zmírnit bezpečnostní omezení sandbox appletů, pomocí JNLP API[21] lze přistupovat k souborovému systému, tiskárnám a schránce.

5.3 Technologie na straně serveru

Výběr programového řešení serveru byl podnícen zkušenostmi autora s programovacím jazykem PHP. Dále jsou použity technologie, které se ve spojení s PHP používají – primárně se jedná o databázi MySQL. I přesto byla provedena analýza 3 nejvýznamnějších[22] technologií používaných pro realizaci webových stránek (tabulka 2). Z tabulky je zřejmé, že technologie Java a ASP.NET jsou pro naše požadavky zbytečně složité, a proto bylo přistoupeno k realizaci Administrace technologií PHP.

[19] Implementuje možnost ověření bezpečnostních politik přístupu JVM k hostitelskému PC a podle toho řídit běh Java programu, více viz *Class SecurityManager*. [online]. [cit. 2013-04-25]. Dostupné z: http://docs.oracle.com/javase/6/docs/api/java/lang/SecurityManager.html
[20] Tento bod neplatí při použití JNLP souborů – bude zmíněno v dalším textu.
[21] Jedná se o třídy z balíčku *javax.jnlp*.
[22] Statistika viz *Usage of server-side programming languages for websites.* [online]. [cit. 2013-04-21]. Dostupné z: http://w3techs.com/technologies/overview/programming_language/all

Tabulka 2 – Porovnání technologií realizace administrace na straně serveru

	PHP	**ASP.NET**	**Java EE**
Znalost autora práce	**Ano**	Ne	Základní
Možnosti síťové komunikace – viz příloha Q	**Jednoduché**	Složité	Složité
Dostupnost technologie	**Free**	Placené	**Free**
Dostupné vývojové prostředí	**Ano**	**Ano**	**Ano**
Dokumentace	**Ano**	**Ano**	**Ano**
Vyžadován návrhový vzor pro realizaci	**Ne**	Ano	Ne/Ano
Debugovací nástroj	Ne[23]	**Ano**	Ano/S problémy
Přímá podpora DB	**Ano (**MySQL)	Ne/Přes drivery	Ne/Přes drivery

S programovacím jazykem PHP se běžně používá databáze MySQL. Tato databáze je velmi rozšířená, podporující relace a pro potřeby administrace (viz podkapitola 4.14.2) bude zcela dostačovat. Jiné alternativy, jako je řešení Oracle či MSSQL, nebyly uvažovány vůbec a to i z toho důvodu, že databáze MySQL je dostupná pod Freeware licencí, narozdíl od ostatních zmíněných, které jsou placené. Více o zvolené databázi a jejím použití v části 5.3.3.

5.3.1 PHP

Jedná se o nejrozšířenější[24] programovací jazyk pro generování webových stránek, avšak jazyk PHP umožňuje mnohem více[25].

PHP je sekvenční programovací typově volný[26] jazyk, který od 5. verze zavádí silnou podporu tříd a reaguje tak na aktuální trend OOP jazyků.

Administrace je naprogramována s využitím funkcí vyžadující minimálně PHP verzi 5.3.

Následuje popis zásadních výhod a nevýhod PHP z pohledu zkušeností autora práce.

Výhody

- Volně dostupný – licence[27] OpenSource,

[23] Ladící nástroje nejsou dostupné v běžně používané standardní bezplatné distribuci.
[24] K datu 18. 4. 2013 používá PHP 79% známých webových serverů, více viz poznámka pod čarou č. 22.
[25] Lze použít pro tvorbu desktopových aplikací, zastane činnost shellu, ... Více viz http://php.net/.
[26] Proměnné se neinicializují typově (vyjma instancí tříd), datový typ je přiřazen při prvním použité proměnné případně změněn (přetypován) podle případu užití (vstup funkce, atp.)

- multiplatformní – často provozován ve spojení s OS Linux,
- relativně jednoduchá syntaxe a použití,
- kompilace kódu probíhá vždy při požadavku na zobrazení/načtení stránky či souboru – jednoduché použití,
- značné množství integrovaných funkcí.

Nevýhody

- Nejednotné názvosloví funkcí a obecný chaos ve funkcionalitě rozdělené mezi funkce a třídy, případně statické třídy, daný historických vývojem,
- nejednoznačná[28] práce s UNICODE řetězci,
- žádné[29] ladící nástroje,
- kompilace kódu probíhá vždy při požadavku na zobrazení (načtení) stránky či souboru – značně vyšší režie oproti kompilovaným jazykům.

5.3.2 Způsob ukládání dat

Po administraci požadujeme ukládání katalogových údajů týkající se jednotlivých částí Administrace (viz podkapitola 4.1 s požadavky – první část) – vybrané pouze zásadní části:

- **Applet** – informace o konkrétním typu vzdáleného logického procesu včetně možností nastavení modelu vytvořeného dle appletu.
- **Pracovní stanice** – jednoduchá informace o pracovní stanici, ke které bude použitý applet náležet.
- **Simulace** – informace o konkrétním simulačním případě, zde musíme brát v úvahu:
    - topologii sestavení appletů a jejich konkrétní nastavení,
    - nastavení „globální“ pro všechny prvky distribuované simulace,
    - informace o sezení simulace.

Práce si klade za cíl vytvořit do co možná největší míry univerzální řešení schopné realizovat libovolný případ distribuované simulace. Z tohoto faktu vyvstává problém vytvoření dostatečně obecného řešení pomocí čistě databázového řešení, které se dnes používá primárně

[27] PHP je zaštítěno vlastní licencí (obsahuje licenci OpenSource), více viz: *PHP Licensing*. [online]. [cit. 2013-04-21]. Dostupné z: http://www.php.net/license/

[28] Nejjednodušší demonstrace tohoto problému je bezproblémové ukládání UTF-8 souborů, ale neschopnost přístupu k jednotlivým znakům pomocí operátoru hranatých závorek (odpovídá rozhraní ArrayAccess, i když to se v tomto případě nepoužívá).

[29] Ladící nástroje nejsou dostupné v běžně používané standardní bezplatné distribuci.

pro ukládání dat. Čistě databázové řešené je vždy[30] založeno na předem známé struktuře dat, se kterými bude databázový systém pracovat. Modely však mohou vyžadovat značně složité datové struktury pro specifikaci informací potřebných pro běh a z tohoto důvodu využijeme pro ukládání konfigurace textové XML soubory, které právě umožňují uchovávat data zcela[31] libovolně strukturovaná.

Také je nutné zajistit předávání uchovávaných dat appletům načítaných na straně klientů a počítat s problémy typu zabezpečení klientských počítačů pomocí např. firewallů, apod. Řešení tohoto problému, ale i podrobný popis použitých technologií následuje v části 5.3.3 – databáze MySQL a dále v části 5.3.4 – XML.

5.3.3 Databáze MySQL

Pro databázové řešení ukládání dat je použita databáze MySQL. Tato databáze byla zvolena bez většího průzkumu jiných možností neboť:

- jedná se o dostupné řešení s OpenSource licencí – bezplatné,
- značné rozšíření a z toho plynoucí možnosti podpory, ať již oficiální nebo ze strany komunity,
- znalost autora práce,
- jednoduché ovládání a správa,
- přímá podpora ze strany PHP,
- a v neposlední řadě i fakt, že databázi budeme používat pouze pro uchovávání základních katalogových údajů o stavu administrace bez potřeby vyšší[32] správy či optimalizace dotazů, jako nabízí např. řešení od společnosti Oracle.

Databázi budeme používat pouze na straně serveru, klienti se k administrační databázi nebudou připojovat, ale data budou získávat z XML souborů uložených na serveru. Tím nám odpadne problém zabezpečení přístupu k datům na straně serveru – museli bychom používat pokročilejší řízení uživatelských účtů v databázi (primárně restrikce použitelných dotazů na typ *SELECT*), případně přístup k datům řešit přes procedury apod. To vše by bylo

[30] Při použití principu tzn. 5. normální formy (NF) ukládání dat do DB je možné vytvořit téměř zcela obecný systém pro ukládání libovolně strukturovaných dat. 5. NF se v běžné praxi nepoužívá, neboť vyžaduje nepřiměřené (vzhledem k faktickému přínosu) nároky na návrh a realizaci a v neposlední řadě i zatížení a režii databázového systému, který bude takovouto strukturu obsluhovat.

[31] Existují drobná omezení jako je například výskyt jednoho kořenového elementu jako páteřního prvku uložených dat, případně možnost validace dat oproti tzn. schématům – toto bude rozebráno v příslušné kapitole 5.3.4 o XML).

[32] Databáze MySQL samozřejmě podporuje katalogy a optimalizaci dotazů, avšak oproti možnostem například komerčního řešení Oracle, které obsahuje kupříkladu materializované pohledy aj., se nedá příliš srovnávat.

samozřejmě možné, avšak by přineslo vyšší náklady na práci, případně by komplikovalo práci při budoucím rozšiřování administrace.

5.3.4 Jazyk XML a jeho možnosti

Primárním zdrojem informací o XML jsou internetové stránky konsorcia W3C [9, 10]. Značkovací jazyk XML je standard, který si klade za cíl uchovávat data:

- pro uživatele čitelné podobě,
- vhodně pro strojové zpracování s přihlédnutím k udržování vazeb a struktury uchovávaných dat,
- multiplatformě nezávisle,
- v libovolném kódování,
- snadné přenositelnosti (dnes hlavně skrze internet).

Výsledek je realizován jako běžné textové záznamy[33] (čitelné, přenositelné, multiplatformně nezávislé, strojově zpracovatelné), které na prvním řádku obsahují metainformaci o použitém kódování a dále obsahující již konkrétní strukturu dat. Jako nevýhoda je považováno relativně pomalé strojové zpracování, což vychází právě z předností formátu – jeho čitelná a textová podoba – data je třeba načíst jako textovou informaci (je nutné načíst celý[34] soubor) a posléze přetransformovat do podoby, které bude rozumět program.

Ukázka a složení XML dat

```
<?xml version="1.0" encoding="utf-8"?>
<simulation version="20130403">
      <name>Dálnice + Levé občerstvení</name>
      <saved>2013-04-20T15:38:55+02:00</saved>
      <start-date>2013-03-11T20:00:00+01:00</start-date>
      <random-seed>1366469363250</random-seed>
      <models>
            <model
                  unique="bb6b80191a42897cc33f2a8d8206ea69"
                  unique-designer="bb6b80191a42897cc33f2a8d8206ea69">
                  <id>8</id>
                  <workstation />
                  <xmlfile>dalnice.stred.xml</xmlfile>
                  <name>Dálnice středový prvek 3866</name>
                  <ports>
                        <server>8888</server>
                        <client>60038</client>
```

[33] Často běžné soubory, informace uložené v paměti počítače dnes i ukládané a zpracovávaní přímo databázemi – pro příklad Oracle, podrobnosti viz: ORACLE. *Introducing Oracle XML DB*. [online]. [cit. 2013-04-21]. Dostupné z: http://docs.oracle.com/cd/B14117_01/appdev.101/b10790/xdb01int.htm

[34] Textový soubor může být jakkoliv veliký, výjimkou nemusí být ani stovky MB, a tuto informaci je při zpracování samozřejmě nutné uchovávat v operační paměti, která je omezená.

```
                </ports>
                <settings>
                      <!-- ... další struktura -->
                </settings>
          </model>
          <model
                unique="08i23"
                unique-designer="08i23">
                <name>Model 08i23</name>
                <!-- ... další struktura -->
          </model>
    </models>
</simulation>
```

Na předchozí ukázce je vidět konkrétní struktura dat (zkrácená), kterou generuje Administrace – záznam konfigurace distribuované simulace. Složité – hierarchické – struktury dat je docíleno vnořováním elementů do sebe. V tabulce 3 je stručně popsán význam základní prvků XML kódu.

Tabulka 3 – Stručný přehled prvků XML dokumentu

Prvek dle ukázky XML	**Typ**	**Poznámka**
`<?xml version="1.0" ?>`	hlavička	Vyskytuje se pouze jednou
`<simulation>`	element	Tzn. root element, může být pouze jeden a zastřešuje ostatní prvky
`<models>`	element	Element, který má nějaký obsah
`<workstation />`	element	Zkrácený element, nemá žádný obsah (může obsahovat atributy – viz níž)
`version="{text}"`	atribut	Může obsahovat jakoukoliv jednoduchou textovou informaci
`<!-- {text} -->`	poznámka	Libovolný text, při zpracovávání se na takovou část nebere ohled

* První sloupeček *prvek* odpovídá ukázce XML kódu uvedeného výše na předchozí stránce

V dalším textu budou uvedeny zásadní a pro realizaci práce použité techniky práce s XML. Možností a standardů jak pracovat s XML je mnohem více – viz oficiální internetové stránky konsorcia W3C [9].

XML parsery

Ve všech dnes běžně dostupných programovacích jazycích jsou dostupné tzn. XML parsery. Parser slouží k přístupu, případně budování strukturované XML informace. Parsery vždy uchovávají celý „obraz“ – strukturu XML a s ní dále pracují. Vzhledem k faktu, že se jedná

o hierarchickou strukturu dat uspořádanou zcela libovolně[35], nelze říci, že práci s XML záznamem lze obecně optimalizovat.

Následuje jednoduchá ukázka práce v programovacím jazyce PHP s XML parserem DOM (jeden z v PHP dostupných parserů).

```
<?php
$Dom = new DOMDocument();     // Vytvoření základního objektu
$Dom->load("file.xml");       // Načtení souboru obsahující XML, obsah
                              // odpovídá ukázce XML na straně 40
$DomNodeList = $Dom->getElementsByTagName("simulation");
                              // Načten uzel <simulation>
$SimulationDomNode = $DomNodeList->item(0); // První <simulation> uzel
echo $SimulationDomNode->getElementsByTagName("name")->item(0)
                                                        ->textContent;
                              // Vypíše název simulace
?>
```

XPath

S XML parsery se často používá dotazovací jazyk XPath (XML Path Language). Tento jazyk značně rozšiřuje možnosti přístupu k hierarchické struktuře dat, kterou XML obsahuje, neboť umožňuje pokládat dotazy (zpravidla na XML parser) na výskyt informací v rámci hierarchie dat. XPath má velice volnou strukturu a lze použít mnoha způsoby od hledání konkrétního záznamu, po seznam uzlů XML odpovídající určitému předpisu. Lze vyhledávat i podmíněně a pracovat s vestavěnými i vlastními funkcemi. Opět lze říci, že takovou úlohu nelze obecně optimalizovat vzhledem k variabilitě samotné XML struktury, ale i k volnosti dotazů XPath.

Následuje jednoduchá ukázka práce v programovacím jazyce PHP s XPath (obsluhuje třída DOMXPath), pod XML parserem DOM.

```
<?php
// ... Navazujeme na předchozí příklad s parserem DOM
$DomXPath = new DOMXPath($Dom);
$DomNodeList
      = $DomXPath->query("//simulation/*/model[@unique='08i23']/name");
            // Dotazujeme s na:
            // 1. root element simulation
            // 2. jakýkoliv element
            // 3. element <model> s konkrétním atributem "unique"
            // 4. element <name>
echo $DomNodeList->item(0)->textContent;
            // Vypíše název konkrétního modelu
?>
```

[35] Logické uspořádání dat v XML záznamu nemusí být optimální pro datovou strukturu, se kterou pracuje parser.

Validace dat

Validace ověřuje shodu XML dokumentu se schématem. Používají se dva druhy dokumentů, oproti kterým se validace provádí:

- **XML DTD** – starší přístup, lze definovat (povinné) elementy, které by měl dokument obsahovat,
- **XML Schéma** – nový přístup, je použit pro validaci konfiguračních souborů v administraci, a proto je v další části kapitoly popsán podrobněji.

XML Schéma

XML Schéma je XML dokument, který popisuje správnou (povolenou) strukturu XML.

Je možné definovat:

- strukturu elementů, i tzv. *any* elementy – elementy libovolného typu či struktury,
- příslušnost atributů k elementům,
- datový typ elementu,
- vytvářet složité strukturované datové typy,
- vytvářet jednoduché datové typy specifikací restrikcí nad jednoduchými (vestavěnými) typy.

Parsery umožňují validovat dokument proti XML Schéma často s podrobným výpisem chyb. I přes „nevaliditu“ dokumentu, lze z takového souboru číst a pracovat s ním, což je výhoda dále rozšiřující možnosti použití XML, avšak zde již můžeme narazit na problémy s automatizovaným procházením souborů, neboť algoritmy zpravidla očekávají známou (často právě podle XML Schéma) strukturu, bez které nejsou schopné správně či bezpečně pracovat. Tento problém lze vyřešit definováním tzv. *any* elementů, neboli míst v XML dokumentu, která mohou obsahovat libovolná data (včetně dalšího XML strukturovaného kódu).

Ukázka XML Schéma:

```
<?xml version="1.0"?>
<xs:schema  version="1.0"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="model">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element name="name" type="xs:string" />
                        <xs:any
                              processContents="lax"
                              minOccurs="0"
                              maxOccurs="unbounded" />
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
</xs:schema>
```

Administrace používá element jednu pozici elementu *any* (definováno v příslušném XML Schéma), který umožňuje uložit jakoukoliv dodatečnou strukturu/y – tím je zaručena naprostá variabilita dat, pokud by se taková potřeba vyskytla (něco podobného v databázi realizovat jednoduše nelze).

5.4 Technologie na straně klienta

Strana klienta je realizována v internetovém prohlížeči jako běžné HTML stránky (generované na straně serveru pomocí PHP) s tím, že samotná výpočetní jednotka distribuované simulace probíhá v Java Appletu (popsáno ve části 5.2.1), který se načte v prohlížeči. Tato podkapitola čerpá z oficiálních specifikací HTML5 [11] a CSS(3) [14] od konsorcia W3C.

5.4.1 HTML5

Technologie HTML se používá pro zobrazování internetových stránek. Je to značkovací jazyk postavený nad standardem XML (viz část 5.3.4), kde se místo elementů zavádí pojem tag. Tagů je omezené množství a jsou mezi nimi vyžadovány určité vazby, případně povinnosti výskytu v určitých případech. Tagy mají také přednastavenou vizuální reprezentaci. Zobrazení tagů lze specifikovat pomocí jazyka CSS.

HTML5 je nejnovější verzí jazyka HTML, v současné době ho zcela nepodporuje žádný prohlížeč. K HTML5 se váže[9] nová verze CSS – verze 3.

HTML5 obsahuje novinky (výběr):

- <canvas> – možnost dynamické kreslící plochy přímo ve stránce (právě tato novinka je stěžejní pro tuto práci a je podrobně popsána níže v této kapitole).
- <embed> – vložení externího obsahu na stránku – teoreticky se nabízí použití při zobrazení Java Appletu (rozebráno v části 6.2.1).

Pro vizuální zobrazení internetové stránky jsou podstatnější novinky v CSS3, které jazyk CSS značně rozšiřují a nově umožňují následující (výběr):

- Ohraničení (*border*): použití obrázku, stín, zakulacené rohy, ...
- Barvy (*color*): průhlednost, přechody (viz
- obrázek 7), ...
- Animace a přechody (*transition*): umožňují animované změny CSS stylu, především mají význam při událostech – selektory (*:hover, :focus*, ...).
- Transformace (*transform*): umožňuje změny (často tvaru a pozice) elementů na stránce.

Tag <canvas>

Tento tag byl zaveden s novou verzí HTML5, je to způsob, jak přímo kreslit do plochy internetové stránky 2D objekty. Do této doby neexistoval způsob jak něco podobného realizovat. Ke kreslení je vyžadován JavaScript (viz

obrázek 7 s přechodem), který je koncepčně připraven i pro kreslení 3D objektů, nicméně žádný současný prohlížeč takovouto funkci neumožňuje.

Je možné kreslit:

- základní geometrická primitiva (obdélník, ovál, rovná čára, křivky – bezierovy kvadratické a kubické),
- průhlednost,
- přechody a vzory pro vyplňování objektů,
- transformace (rotace, měřítko (*scale*), posun),
- texty,

- ukládat a načítat stav elementu – kresbu je možné uložit, je možné ukládat stav do formátu obrázků (výchozí PNG, JPG, teoreticky jakýkoliv specifikovaný[36] MIMETYPE) – pro ukládání se používá formát base64.

Ukázka kreslení na <canvas>:

Zdroj: Autor

Obrázek 7 – Ukázka kreslení na <canvas>, přechod od černé po červenou s textem

Zdrojový kód pro obrázek 7:

```
<canvas id="c" width="500"
height="500">Nepodporujete
HTML5</canvas>
<script type="text/javascript">
var c=document.getElementById("c");
var ctx=c.getContext("2d");
var grd =ctx.createLinearGradient(
-50,100,450,0);
grd.addColorStop(0,"black");
grd.addColorStop(0.2,"purple");
grd.addColorStop(0.4,"blue");
grd.addColorStop(0.6,"green");
grd.addColorStop(0.8,"yellow");
grd.addColorStop(1,"red");
ctx.fillStyle=grd;
ctx.fillRect(0,00,500, 500);
ctx.font = "100px Arial";
ctx.textAlign = "center";
ctx.textBaseline = "middle";
ctx.strokeStyle = "#FFFFFF";
ctx.lineWidth = 5;
ctx.strokeText("<canvas>", 250,175);
ctx.strokeText(23, 250, 325);
</script>
```

5.4.2 JavaScript

Interpretovaný programovací jazyk JavaScript se zpravidla používá pro vytváření dynamických efektů a změn na internetové stránce.

JavaScript je dynamický objektový prototypový jazyk. Objekty s programové konstrukce jsou realizované až za běhu programu pomocí prototypování, pod čímž si můžeme představit možnost libovolného rozšiřování existujících objektů či prvků jazyka o množiny dalších atributů či funkcí. Toto je významná vlastnost, která umožňuje jednoduše vytvářet:

- složité struktury kódů (můžeme nazývat třídami[37]),
- libovolné rozšiřování existujících objektů[38] (ne jenom jich),

[36] Testované prohlížeče podporují formáty PNG, JPG, Chrome navíc WebPicture, Firefox BMP. Testováno na: *Testing canvas.toDataURL with different mime types.* [online]. [cit. 2013-04-21]. Dostupné z: http://kangax.github.io/jstests/toDataUrl_mime_type_test/

[37] JavaScript přímo nepoužívá prvky koncept OOP jako je třída, konstruktor, atp., ale je možné takové funkcionality docílit právě pomocí prototypování.

- asociativní pole[39],
- cokoliv, co dovolí tvůrci jeho fantazie, neboť JavaScript obsahuje minimum omezení a obsahuje přístup[40] téměř ke všemu.

Další významnou vlastností je přímý přístup a manipulace s XML strukturou zdrojového kódu (X)HTML internetové stránky (tzn. HTML DOM).

Byly zmíněny výhody JavaScriptu, nyní se zaměříme na jeho nevýhody.

Nevýhody a problémy

Nevýhody často plynou právě z prototypování, jakožto možnosti, která do programového kódu zavádí velikou svobodu, a kde se počítá s tím, že autoři JavaScriptových aplikací tímto principem vytvoří složitější konstrukce, dnes běžně dostupné[41] přímo v programovacích jazycích. Nevýhody:

- absence složitějších datových struktur, i pro jednoduché konstrukce se používá často řada frameworků, které řeší a zvyšují použitelnost jazyka.
- vysoká svoboda prototypování je snadnou cestou k chybám[42],
- dynamické určování datových typů spolu s prototypováním,
- nebezpečný přístup[40] k zásobníku funkcí,
- problémová optimalizaci[43] kódu,
- nestandardizovaná podpora (především k přístupu k HTML DOM struktuře) jazyka napříč internetovými prohlížeči[44].

---

[38] V tomto případě si pod pojmem objekt představme spíše prvek, či funkcionalitu samotného jazyka nebo uživatelského kódu.

[39] JavaScript nezná pojem asociativní pole. Samotná pole – Array, jenž jsou v JavaScriptu dostupná, jsou standardní pole s dynamických počtem prvků, indexovaných od nuly nahoru. Programový konstrukt odpovídající asociativnímu poli (podle C++ STL, či vyšších prog. jazyků) je vytvořen právě prototypováním – rozšiřování existujícího prvku (objekt, často pole - Array, ale i například čas – Date) o další atributy, jakoby prvky asociativního pole, a tím je docílen požadovaný efekt. O pole – datový typ Array – se ve skutečnosti vůbec nemusí jednat.

[40] Pomocí funkcí *caller()* a automaticky generovaný atribut *arguments*, při jejich správném použití je možné přistupovat k argumentům funkce, kterou byla aktuální funkce zavolána. Zásobník argumentů funkce obsahuje reference na hodnoty a z toho důvodu je možné měnit hodnoty funkcí, které byly v hierarchii volání aktuální funkce volány, a tedy měnit stavový prostor (atributy, funkce, ...) zcela cizích objektů.

[41] Pro příklad uveďme hashovací mapy, stromy, ale i samotné objekty z pohledu OOP.

[42] Chyby programátora samozřejmě není možné považovat přímo za chybu jazyka, avšak faktem je, že oproti silně typovým objektově orientovaným jazykům, je zde spousta prostoru pro zavedení chyby a jejího následného nečekaného chování.

[43] JavaScript obsahuje funkci *eval()*, která umožňuje vykonat kód uložený ve formě řetězce, vzhledem k dalším vlastnostem JavaScriptu (pro příklad uveďme možnost uložit funkci do proměnné a s ní dál pracovat) tato funkce přerušuje vazby na objekty, které používá optimalizátor, a výkonná optimalizace obdobná statickým programovacím jazykům není možná.

Ukázka prototypování

```
// Definice konstruktoru je definice funkce
function ProgramovaciJazyk(nazev) {
      this.nazev = nazev;
            // Objekt / třída / konstruktor se z funkce stane
            // použitím klíčové slova "this", kdy funkci přiřadíme
            // atribut, a rozšíříme jí tak nad běžné vlastnosti funkce
}

vypsat = ProgramovaciJazyk.prototype.vypsat = function () {
      alert("Programovací jazyk: " + this.nazev);
            // Pomocí klíčového slova "prototype" jsme rozšířily instanci
            // funkce "ProgramovaciJazyk" o funkci "vypsat", vznikl tak
            // programový konstrukt, který odpovídá OOP konceptu,
            // a funkci "ProgramovaciJazyk" s funkcí "vypsat"
            // můžeme považovat za třídu a metodu
};

vypsat.prototype.vypsatVice = function () { // Metodě přiřadíme funkci
      alert("...");
};

var programovaciJazyk = new ProgramovaciJazyk("JavaScript");
programovaciJazyk.vypsat(); // Vypíše "JavaScript"
var vypsat = new programovaciJazyk.vypsat();
      // Vytvoříme objekt z funkce, vypíše "undefined"
vypsat.vypsatVice(); // Voláme funkci funkce objektu "programovaciJazyk"
      // Vypíše "..."

var vypsat = new ProgramovaciJazyk.vypsat(); // Skočí chybou. Proč?
      // Předpis funkce "ProgramovaciJazyk" nemá žádnou
      // podsložku "vypsat", funkce "vypsat" přísluší pouze existující
      // instanci "programovaciJazyk"
```

Programovací jazyk JavaScript je v současné době jediný způsob, jak realizovat programy přímo v internetové stránce. S příchodem HTML5 se jeho užívání stává nutností, neboť některé prvky HTML5 není možné obsluhovat bez JavaScriptu, pro příklad uveďme tag <canvas> (více viz předchozí část 5.4.1 i s ukázkou kódu u obrázku 7), který umožňuje kreslit na (vyhrazenou) plochu internetové stránky, avšak pouze[45] JavaScriptem.

[44] Tento stav se týkal především prvních internetových prohlížečů na trhu.

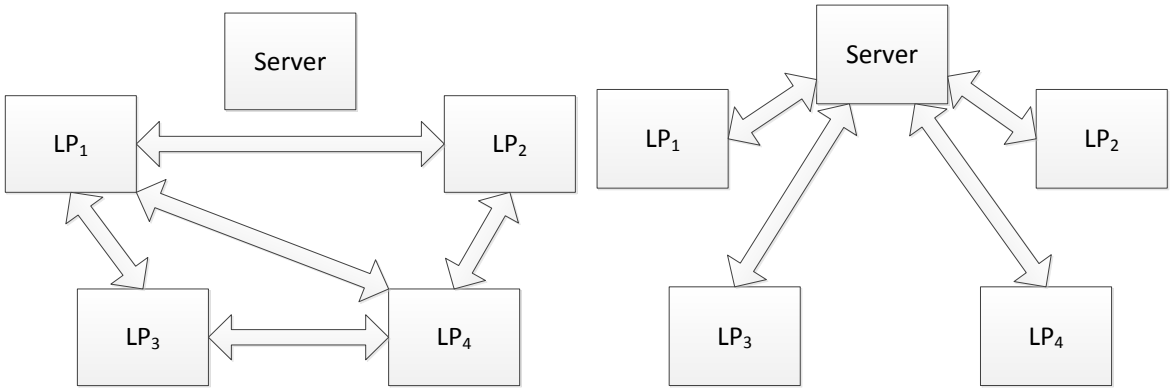
[45] Koncepčně připraveno na jakýkoliv obslužný kód, avšak JavaScript je v současnosti jediný programovací jazyk internetových stránek.

6 VYBRANÉ IMPLEMENTACE A ALGORITMY

V této části budou zmíněny vybrané implementace, případně problémy, které se v průběhu realizace objevily.

6.1 Síťová komunikace mezi distribuovanými logickými procesy

V části 5.2.1 je popsáno omezení síťové komunikace v Java Appletu, které je pro náš případ zcela zásadní. Optimální komunikace mezi logickými procesy vyžaduje peer-to-peer architekturu – viz obrázek 8a, jednotlivé logické procesy komunikují přímo, zcela bez účasti serveru[46]. Toto bohužel není možné realizovat v rámci Java Appletu.



(a) (b) Zdroj: Autor

Obrázek 8 – Schéma komunikace (a) peer-to-peer a (b) klient-server-klient

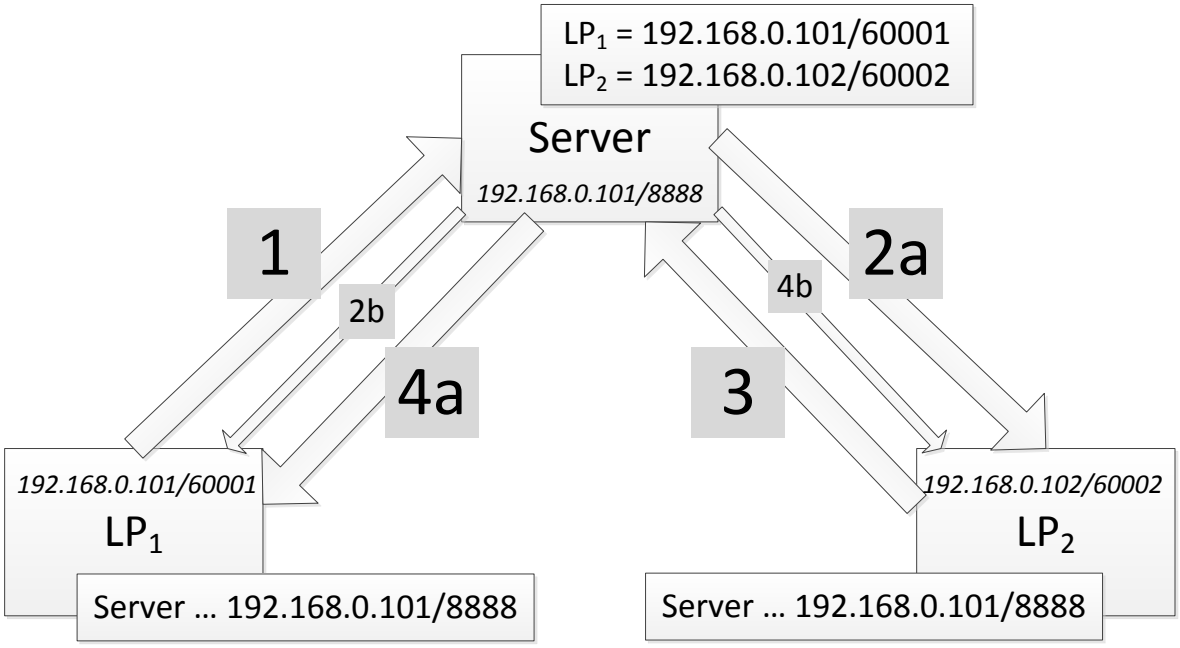
Realizováno bylo řešení klient-server-klient, kde za klienty považujeme logické procesy distribuované simulace. Komunikace mezi logickými procesy probíhá přes centrální prvek – server. Server přeposílá zprávy klientů. Tento způsob komunikace splní stejný účel jako peer-to-peer řešení, avšak se zvýšením režie komunikace. Zpomalení komunikace je způsobeno nutností odeslat zprávu dvakrát (klient-server a server-klient). Komunikační režie je navíc zvýšena obsluhou požadavku na serveru. Výsledná architektura spojení prvků distribuované simulace bude vypadat tak, jako je zobrazeno na obrázku 8b.

V dalším textu budou termíny logický proces a klient ekvivalentní, neboť logický proces je zpravidla realizován jako jeden z klientů centrálního serveru. Server jako takový nemá se samotnou distribuovanou simulací (myšleny výpočty nad modelovaným systémem) nic společného.

[46] Server je zpravidla používán pouze pro inicializaci komunikace, přiřazení adresa a vytvoření spojený mezi logickými procesy.

Server a architektura přeposílání zpráv klient-server-klient

Princip přeposílání zpráv se vysvětlíme na příkladu zobrazeném na obrázku 9.



Zdroj: Autor

Obrázek 9 – Znázornění síťové komunikace mezi dvěma klienty (logickými procesy simulace)

Server si udržuje tabulku známých LP, především jména[47], síťové adresy klienta a port, na kterém klient komunikuje.

Každý logický proces musí znát adresu serveru, přes který komunikuje. Pokud LP komunikuje s nějakým klientem, musí znát jeho jméno – v našem příkladu jednoduše „LP“ s dolním indexem.

Průběh zaslání zprávy z LP_1 na LP_2:

1. LP_1 zašle zprávu serveru, s informací, že příjemcem je LP_2 (obrázek 9 – bod 1).
2. Server zjistí z tabulky známých klientů síťové parametry LP_2 – síťovou adresu a port a přepošle (obrázek 9 – bod 2) z informací, od kterého logického procesu (LP_1) zpráva pochází.
3. Server zašle odpověď LP_1 o úspěchu přeposlání.
4. LP_1 čeká na odpověď, případně zpracování další události.

[47] Jméno není nepodstatné, slouží pouze pro lepší orientace. Pro konkrétní identifikaci je nutné znát síťovou adresu a port klienta.

5. LP_2 obdrží zprávu, zjistí, zda je zpráva „v pořádku“, a odešle odpověď (obrázek 9 – bod 3). Odpověď LP_2 je směrována na server, s tím, že příjemce odpovědi je LP_1
6. Server obdrží odpověď a přepošle ji příjemci odpovědi LP_1 (obrázek 9 – bod 4).[48]
7. Server pak zašle odpověď LP_2 o úspěchu přeposlání odpovědi.
8. LP_1 přijme odpověď a reaguje podle ní.

Zahájení síťové komunikace

Před zahájením samotné simulace je nutné správně nastavit spojení mezi jednotlivými klienty – v našem případě zaregistrovat klienty na serveru. Poté případně informovat jednotlivé logické procesy o jejich okolí – tedy o logických procesech, kterým budou zasílat zprávy o konkrétnímu logickému procesu. Následně je možné zahájit simulaci. Podrobné vysvětlení algoritmu:

Distribuovaná simulace bude probíhat podle obrázku 9. Předpokládáme, že jednotliví klienti jsou informováni o serveru (především umístění v síti), přes který komunikují.

Rozlišujeme tři hlavní typy zpráv:

- *hello* zpráva – informuje server o existenci logického procesu (server uloží především informace o síťovém umístění klienta),
- *register* zpráva – zpráva, kterou informuje logický proces své okolí (logické procesy, kterým bude posílat zprávy) o vlastní existenci, především z toho důvodu, aby s těmito zprávami (obecně logickými procesy) okolní logické procesy počítaly v rámci synchronizace svého LVT (viz kapitola 2 a podkapitola 1.8) – toto zprávu server přeposílá,
- *client* zpráva – zpráva samotnému vzdálenému klientovi (typicky zpráva předávající událost), kterou server přeposílá.

Samotný algoritmus (není podstatné, který logický proces – LP_1, LP_2 zahájí komunikaci v rámci jednotlivých částí A, B a C nehraje roli):

Část A – hello zprávy:

1. LP_1 zašle *hello* zprávu serveru.
2. Server uloží síťové informace od LP_1 – síťovou adresu a port, kam bude server přeposílat zprávy.

[48] Vzhledem ke snížení komunikační režie, jsou přeposílány původního odesílateli LP_1 pouze chybové zprávy.

3. LP_2 zašle *hello* zprávu serveru.
4. Server uloží síťové informace od LP_2 – síťovou adresu a port, kam bude server přeposílat zprávy.

Po zaslání *hello* zpráv všech logických procesů pokračujeme částí B:

Část B – *register* zprávy:

5. LP_1 zašle *register* zprávu, adresovanou LP_2, serveru.
6. Server přepošle zprávu na LP_2.
7. LP_2 bere tento fakt na vědomí – v případě použitého algoritmu z kapitoly 3 se vytvoří speciální fronta zpráv (kalendář) pro LP_1, mohli bychom zapsat jako $LP_2.Queue(LP_1)$.
8. LP_2 zašle *register* zprávu, adresovanou LP_1, serveru.
9. Server přepošle zprávu na LP_1.
10. LP_1 ekvivalentně s bodem 7 vytvoří $LP_1.Queue(LP_2)$.

Po zaslání *register* zpráv všech logických procesů pokračujeme částí C:

Část C – *client* zprávy:

11. LP_1 posílá zprávu LP_2 přes server.
    a. Server přepošle zprávu LP_2.
    b. LP_2 odešle na server odpověď LP_1.
    c. Server přepošle odpověď klientu LP_1.
    d. LP_1 reaguje podle odpovědi.
12. LP_1 může pokračovat v zasílání zpráv případně čeká na odpověď LP_2.
13. LP_2 zasílá zprávu LP_1 ekvivalentně s body 11 a 12.
14. Body 11 až 13 se opakují do ukončení (viz níže) jednoho z logických procesů.

Ukončení simulace

K ukončení simulace může dojít při:

- **Chyba** v logickém procesu:
    - chyba[49] vzniklá při samotné činnosti LP,
    - nebo při chybné odpovědi[49] zaslané jedním z okolních LP.

[49] Záleží na charakteru chyby.

- **Plánované** ukončí simulace:
    - vybraný[50] logický proces vyhodnotí svůj aktuální stav jako stav ukončující distribuovanou simulaci (jsou splněny ukončující podmínky) a ukončí simulaci.

Logický proces, který ukončuje svojí činnost (zpravidla i celou distribuovanou simulaci – tedy všechny logické procesy, kterých se simulace týká), rozešle zprávu každému logickému procesu, kterému zasílá události, o ukončení své činnosti. Každý logický proces, který takovouto zprávu přijme, je ukončen, a sám rozesílá zprávy okolním logickým procesům o ukončení své činnosti. Tímto způsobem se rozšíří informace o ukončení simulace všem logickým procesům.

Po ukončení logického procesu proces nepřijímá a nezpracovává žádný zprávy či události.

Při ukončení činnosti (ať již plánovaného či z důvodu výskytu chyby) logického procesu je zobrazena zpráva o stavu simulace (vlastně průběhu činnosti LP, neboť logický proces sám o sobě nezná kontext celé distribuované simulace) v tomto konkrétním logickém procesu.

Formát zpráv mezi klienty

Zprávy mají formát[51]:

```
sender=LP1&client.name=LP2&req=client/&udalost=prijezd&typ=vozidlo&cislo=23
&time.u=1365714014599&sender=LP1&time=1365714011599
```

Jednotlivé datové položky zpráva jsou tvořeny páry „klíč=hodnota“ oddělené znakem ampersand „&“. Časová razítka jsou v číselném formátu typu *long*[52] – podle epochy UNIXu v milisekundách v desítkové soustavě uložené ve formě řetězce.

Zpráva je rozdělena na dvě části znakem lomítka „/“, více viz rozbor výše uvedené ukázky:

- **serverová** část na levé straně obsahuje informace o odesílateli, cílovém klientu a typ požadavku (v našem případě *client* – zpráva bude přeposlána odesílateli),
- **klientská** část na pravé straně s informacemi o události (*udalost*), časovém razítku události (*time.u*), případně dalšími parametry (*typ, cislo*) týkajícími se události či odesílatele (*sender*)[53].

[50] Na ukončujících podmínkách se může účastnit libovolné množství logických procesů.
[51] Přesná specifikace formátu zpráv je popsána v příloze C – dokument Příručka vývojáře appletu.
[52] Datový typ používaný v programovacím jazyce Java. Obecně se jedná o 64bitové celé číslo.

6.2 Načtení Java Appletu a nalezené problémy

V této kapitole jsou rozebrány možnosti použití Java Appletu na internetu. Uvedené konstrukce jsou vytvořeny podle oficiální dokumentace společnosti Oracle [4, 5, 6] a otestovány ve všech aktuálních dostupných internetových prohlížečích – viz příloha A.

6.2.1 Vložení appletu na stránku

Vložit Java Applet do HTML stránky lze několika způsoby. Nejprve se zaměříme na starý (zastaralý – *deprecated*) přístup přes tag <applet>.

Tag <applet>

Tento tag je zastaralý ve specifikaci HTML 4.1 a ve specifikaci HTML5 není obsažen, ale ve všech testovaných prohlížečích funguje. Jednoduchá ukázka:

```
<applet codebase="SimulaceBase/build/classes/"
code="run/AppletDalnice.class" width="800" height="480">
        <param name="unique" value="bb6b80191a42897cc33f2a8d8206ea69" />
</applet>
```

Tag <applet> je párový a povinně obsahuje atribut *code* – specifikuje cestu k souboru spouštějícímu samotný program appletu. V ukázce dále vidíme atributy *width* a *height* určující rozměr. Tyto dva parametry je nutné použít pro správné zobrazení appletu. Dalším atributem je *codebase*, tento atribut specifikuje základní úroveň uložení zdrojových souborů (především samotný kód Java aplikace v *.class* souborech). Tag <applet> může obsahovat libovolné tagy <param>, kterými předává data, obsažená v atributu *value* pod názvem v atributu *name*, samotnému programu v appletu.

Zásadní na tomto zastaralém přístupu je, že bezchybně funguje napříč prohlížeči, na rozdíl od „moderního“ přístupu popsaného dále v textu.

Tagy <object> a <embed>

Podle specifikace W3C konsorcia by měly být objekty, mimo samotný HTML kód stránky, načteny pomocí tagu <object>. Tento přístup ale nefunguje ve všech majoritních prohlížečích a je nutné využít <embed>, který je nový v HTML5. Mezi použití tagy <object> a <embed> není prakticky žádný rozdíl. Zůstává tag <param>, stejně jako v případě použití tagu <applet>. Ukázka použití je zobrazena pod tabulkou 4.

[53] Položek je více, většinou se jedná o dodatečná data čistě informativního charakteru, jako například položka *time*, která obsahuje LVT odesílatele. Konkrétně položka *time* má význam čistě pro ladění programu.

V případě tagu <object> je nutné použít atribut *classid*, který specifikuje typ obsahu vkládaného tagem <object>.

Přehled použitelnosti jednotlivých tagů (i s ohledem na *classid*) je v tabulce 4.

Tabulka 4 – Přehled použitelnosti HTML tagů pro načtení Java Appletu

Tag	Opera	Mozilla Firefox	Google Chrome	MSIE
<applet>	Ano	Ano	Ano	Ano
<object>[54]	Ne	Ne	Ne	Ano[55]
<object>[56]	Ne	Ne	Ne	Ano
<embed>	Poznámka[57]	Poznámka[58]	Ano	Ano

* Java i se všemi potřebnými pluginy je na testovacím počítači nainstalována a funkční.

Se zachováním vyšší kompatibility řešení napříč prohlížeči je možné použít vnořené konstrukce (tagy <comment> a <noembed> jsou použity, pokud nelze zobrazit obsah příslušným způsobem):

```
<object classid="clsid:CAFEEFAC-0017-0000-0000-ABCDEFFEDCBA"
      width="800" height="480">
      <param name="code" value="run/AppletDalnice.class" />
      <param name="codebase" value="SimulaceBase/build/classes/" />
      <param name="unique" value=" bb6b80191a42897cc33f2a8d8206ea69" />
      <comment>
            <embed type="application/x-java-applet;jpi-version=1.7.0"
                  pluginspage="http://java.com/en/download/"
                  width="800" height="480"
                  code="run/AppletDalnice.class"
                  codebase="SimulaceBase/build/classes/">
                  <noembed>
                        No Java Support.
                  </noembed>
            </embed>
      </comment>
</object>
```

Na první pohled se nabízí použít tag <noembed> pro začlenění tagu <applet>, a tak docílit (s ohledem na fakta v tabulce 4) maximální kompatibility zobrazení appletu s cílem použít nejnovějšího přístupu HTML5. Takováto konstrukce ale není prakticky realizovatelná, neboť pokud prohlížeč nedokáže vykreslit obsah tagu <embed> (v terminologii prohlížečů

[54] Použití s *classid*="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" pro nejvyšší dostupnou verzi Javy.
[55] S problémy, lze se dostat do stavu, kdy Java hlásí že applet požaduje verzi 51.0 – taková neexistuje.
[56] Použití s *classid*="clsid:CAFEEFAC-0017-0000-0000-ABCDEFFEDCBA" pro verzi Javy 1.7.0.
[57] Zobrazí výzvu k instalaci Java pluginu, tato výzva vede na stránku tvůrce prohlížeče s obecnými informaci o použití a instalaci pluginů.
[58] Zobrazí výzvu a průvodce instalací pluginu Java. V dalším kroku oznámí informaci, že plugin nelze nalézt, a je nutné plugin instalovat ručně. Ruční instalace vede ke stažení Java JRE.

„chybějící zásuvný modul“), tak vizuálně <noembed> ignoruje a zobrazí vlastní grafiku, která uživatele informuje[59] o nutnosti instalace chybějícího pluginu.

Načtení skriptem Oracle

Společnost Oracle doporučuje použít vlastní řešení spočívající v užití funkce[60], která rozhodne o nejvhodnějším použití z možných tagů v závislosti na prohlížeči a dalších parametrech. Testováním bylo zjištěno, že tento způsob vždy generuje HTML kód s tagem <applet>.

Shrnutí

Tag <applet> je považován za zastaralý, a přesto je zmíněn jako jediný[61] na stránkách tutorialu týkajícího se používání Java Appletů přímo od společnosti Oracle. Z důvodu maximální kompatibility je pro realizaci použit tag <applet>.

6.2.2 Možnosti načtení zdrojového kódu pro applet

Jak je zmíněno v části 6.2.1, existuje více způsobů distribuce zdrojových kódů Java. Následuje rozbor možností distribuce přes *.class* soubory a JAR archivy.

Soubory Java Byte Code

Použití souborů bytecodu .class je jednoduché. Tagu <applet>[62] je nutné specifikovat atribut *codebase*, kterému je nutné předat adresu k adresáři obsahujícímu[63] zdrojové soubory *.class*. Dále je nutné uvést atribut *code*, který specifikuje *.class* soubor obsahující *main* metodu s třídou, která je potomkem *JApplet* či *Applet* – záleží na zvoleném vykreslovacím „podkladu“. U atributu *code* je vhodné poznamenat, že atribut musí obsahovat adresu relativní vzhledem ke *codebase*.

[59] Způsob a zobrazení informace se napříč prohlížeči liší.
[60] Příslušný kód JavaScriptu dostupný: *deployJava.js* [cit. 2013-04-29]. Dostupné z: http://www.java.com/js/deployJava.js
[61] Informace o tagech <embed> a <object> se podařilo dohled pouze na stránce věnované Java Appletu pro Javu ve verzi 1.5, bez uvedeného data vytvoření stránky – viz *Using applet, object and embed Tags*. [online]. [cit. 2013-04-26]. Dostupné z: http://docs.oracle.com/javase/1.5.0/docs/guide/plugin/developer_guide/using_tags.html
[62] Lze realizovat i pro jiná řešení, ale vzhledem k jejich špatné kompatibilitě se jimi nebudeme zabývat.
[63] Toto je nutné především z důvodu architektury programů v jazyce Java – systém balíčků.

Toto řešení je bezproblémově funkční. Jedinou nevýhodou je načítání značného množství[64] *.class* souborů ze serveru. Tento problém eliminuje použití JAR souborů.

JAR soubory

Tagu <applet>[62] je nutné specifikovat atribut *archive*, tímto atributem předáme adresu JAR archivu appletu. V atributu code uvedeme relativní adresu umístění *main* metody v JAR souboru stejně jako v případě atributu *code* při použití *.class* souborů.

Tento případ se bohužel nepodařilo realizovat žádným způsobem. Applet se vždy načetl, avšak následně došlo k zobrazení chybového hlášení (pokus o načtení modelu dálnice – viz kapitola 7 – modelový příklad):

```
java.lang.ClassFormatError: Incompatible magic value 1008813135 in class
file run/AppletDalnice
```

Byly zjištěny možnosti řešení:

- Chyba se týká Java Cache[65], lze řešit: „Ovládací panely (OS MS Windows) / Java Control Panel / General Tab / Temporary Internet Files / Settings / Delete Files...“ Uvedený postup nefunguje.
- Další možností je načtení chybové stránky místo obsahu appletu. Hodnota 1008813135 odpovídá 0x3C21444F, to může vyjadřovat v kódování Latin[66] text „<!DO“, kterým obvykle začíná kód internetové stránky. K tomuto by dojít nemělo, pokud k sestavení JAR souboru nedochází na žádném serveru (očekávaná je v tomto případě chyba *file not found* – soubor nebyl nalezen a ukončení skriptu generující JAR). Prověřením obsahu JAR souboru bylo zjištěno, že obsahuje správné *.class* soubory.

Bylo prověřeno více, zpravidla nepravděpodobných, možností a eventualit, ale žádný postup nevedl k nápravě. Z toho důvodu nebyla implementována do administrace podpora práce s applety v JAR archivech.

[64] Způsob organizace zdrojových souborů v jazyce Java zpravidla vede ke značnému množství souborů, protože se používá metoda zápisu jedné OOP třídy programu do jednoho souboru. Jedna aplikace běžně obsahuje stovky či tisíce takovýchto tříd.
[65] Více viz *Thread: Incompatible magic value 1008813135 in class file MyApplet*. [online]. [cit. 2013-04-26]. Dostupné z: https://forums.oracle.com/forums/thread.jspa?threadID=1291931
[66] Znaky: 0x3C = <, 0x21 = !, 0x44 = D, 0x4F = O

6.3 Použití síťových protokolů TCP a UDP

Distribuovaná simulace se zavádí, kromě jiného, i důvodu urychlení simulačního výpočtu. Pro síťovou komunikaci se běžně používají transportní protokoly TCP a UDP. Rozdíl mezi těmito protokoly je především – z pohledu práce, která se přímo síťovou komunikací nezabývá – v rychlosti přenášení zpráv (realizováno pomocí *datagramů*) a s ní související spolehlivostí doručení zprávy. Zdrojem informací jsou specifikace RFC [12, 13].

Zjednodušeně řečeno, UDP nabízí vyšší rychlost, za ztrátu spolehlivosti doručení. Naproti tomu TCP nabízí spolehlivé doručení za cenu vyšší režie a tedy pomalejší komunikace.

Pro běžnou komunikaci klienti-server-klient (viz podkapitoly 6.1 a 6.4) bylo užito UDP protokolu, jednak z důvodu zrychlení komunikace, ale také proto, že komunikujeme přes server, tzn. že musíme na každé předání zprávy mezi dvěma klienty / logickými procesy použít dvou „spojení“, tedy je vhodné využít dvakrát rychlejší protokol UDP, oproti dvakrát pomalejšímu TCP.

Je nutné uvést, že UDP zpráva, realizována v platformě Java, může mít maximální velikost 8 kB[67]. To nám bohatě stačí pro běžnou komunikaci mezi klienty, ale může to být limitující pro servisní informační výpisy pro Administraci. Z tohoto důvodu server komunikuje na obou protokolech UDP a TCP. UDP protokol zajišťuje komunikaci mezi klienty a „servisní“ zprávy a komunikaci obstarávání protokol TCP.

Jak již bylo zmíněno, použitím UDP nemáme zaručené bezpečné předávání zpráv, nicméně tuto funkcionalitu můžeme realizovat v samotné aplikaci, případně využít funkcionality platformy Java, která ji přímo implementuje – zde se především jedná o potvrzení spojení s „cílem“ zprávy, ve vymezeném časovém limitu – tento stav je pro nás dostačující.

6.4 Architektura serveru

Diagram tříd je znázorněn na obrázku 15 v příloze B.

Třída Server udržuje seznam klientů – třídy *ClientStation* v hashmapě (použita interní struktura Javy). V další hashmapě jsou umístěny třídy *ClientInfo*, které obsahují statické, aj.

[67] Důvod tohoto omezení se nepodařilo dohledat, neboť velikost by dle specifika RFC měla být 65 kB.

informace o komunikaci s dotyčným klientem. Pro obě hashmapy je použit textový klíč – název[68] klienta a můžeme říci, že existují v poměru 1:1.

V samostatném vlákně běží instance třídy *ServerThreadUdp*. Toto vlákno naslouchá na zadaném portu na protokolu UDP a přeposílá komunikaci klientů.

Další vlákno je *ServerThreadTcp*. Tato třída slouží pouze pro servisní zásahy do běhu serveru, případně pro zpětnou vazbu tvůrci sezení distribuované simulace o stavu serveru, a není kritická pro běh simulace.

Zdůvodnění „dvojité“ komunikace naleznete v podkapitole 6.3.

6.5 Architektura logického procesu

Diagram tříd je znázorněn na obrázku 16 v příloze B, tento diagram je velmi zjednodušen a obsahuje minimum nutné pro znázornění a pochopení základních vazeb a kompozice datových složek logického procesu.

Logický proces je realizován jako třída *SimulationCore,* odvozená od rozhraní *iSimulationCore* (potomek *iSimulationCoreBase*), tato třída řeší hlavní činnost logického procesu.

Realizujeme pouze diskrétní simulační aktivity – události (realizováno třídou *Udalost*).

Události lze vykonat nad třídami odvozenými od rozhraní *iCinnost*, tyto třídy – činnosti – realizují vykonání události a zároveň představují stavový prostor (přímo týkající se simulovaného problému).

Následuje podrobnější seznámení s vybranými zásadními třídami a postupy:

SimulationCore

Odvozeno od rozhraní *iSimulationCore*. Obsahuje instance základních tříd *Calendar*, *Environment*, *Modules*.

V metodě *runOneStep()* je realizován algoritmus výběru jedné událostí z kalendáře a jejího zpracování. V metodě je rozmístěno několik událostí (bodů vykonání programových kódů), pomocí kterých je realizována např. animace.

[68] Reálný stav je složitější, je použita složenina jména a portu, která zabezpečuje větší unikátnost v rámci simulace.

Calendar

Obsahuje frontu událostí (třída *Udalost*), realizováno jako ADT Prioritní fronta dle časového razítka události.

Všechny události, jak vykonatelné přímo logickým procesem (místní události), tak ty, které by měl vykonat vzdálený logický proces (vzdálené události), jsou předány této třídě a ta je zpracuje dle jejich typu. Místní události zařadí do kalendáře a vzdálené odešle příslušnému logickému procesu.

Environment

Třída je zodpovědná za poskytování informací o logických procesech, kterým posílá zprávy „lokální" logický proces, a o logických procesech, které zasílají zprávy „lokálnímu" LP.

EnvironmentRemoteSimulationCoreInfo

Uchovávají informace o všech okolních logických procesech (které zasílají, či kterým odesíláme zprávy). Význam má především pro sledování příchozích zpráv, je uchovávána informace o množství nezpracovaných událostí ve frontě, díky tomu je možné udržovat pouze jeden kalendář (třída *Calendar*) událostí pro všechny zasilatele zpráv a samotný logický proces.

Modules

Třída poskytuje napříč programem (všem částem, které mají k dispozici referenci na *SimulationCore*) dostupnost modulů – specifické třídy poskytující doplňující nekritickou funkcionalitu samotného simulačního jádra – logického procesu. Realizováno bylo několik modulů:

- **VisualLogger** slouží pro speciální logování činnosti do visuální komponenty appletu.
- **Informations** přímo poskytuje informace, které by bylo nutné jinak složitě skládat, pro příklad uveďme časový interval do další události v kalendáři.
- **TypeRegister**, typově bezpečný registr slouží k ukládání a předávání hodnot, např. pro rozlišení směru provozu na okéncích občerstvení (viz modelový příklad – kapitola 7), které jinak pro vykreslení používají shodné komponenty.

Udalost

Třída uchovávání informaci o výskytu události. Obsahuje časové razítko, atributy a název registrované činnosti logického procesu.

iCinnost

Rozhraní, které specifikuje činnost, bez ohledu na její umístění v rámci topologie distribuované simulace. Od toho rozhraní odvozujeme především dva druhy činností:

- **LocalCinnost**: Abstraktní předek všech činností, které jsou prováděny logickým procesem.
- **RemoteNetCinnost**: Obsahuje síťové informace a komunikační prostředky pro zaslání události, která je adresována jednomu z okolních logických procesů.

6.6 Realizace animace v appletu

V metodě *runOneStep()* – viz předchozí podkapitola 6.5 – je před odebráním události z kalendáře užita událost (*event*), která zjistí, zda je rozdíl LVT a časového razítka následující událost (třída *Udalost*) větší, než nula. Pokud je tento rozdíl roven nula, pak v daném čase nejsou odsimulovány všechny události.

Pokud je rozdíl LVT a časového razítka následující události větší, než nula, je k dispozici časový interval, ve kterém můžeme provést animaci. Tento časový interval předá simulační jádro animačnímu modulu, který se postará o animace scény pro tento interval.

Samotná animace je řešena obdobně jako metoda Snímání aktivit pro realizaci spojité simulace (viz podkapitola 1.5).

Animační modul obsahuje seznam (tento seznam je dynamický) animačních aktivit (dále aktivita), které dohromady vykreslují animační scénu. Aktivity se do tohoto seznamu mohou přidávat (plánovat) události.

Rozlišujeme dva druhy animačních aktivit:

- plánovaná aktivita obsahuje časový údaj o začátku a konci své existence v rámci animace a jsou vhodné pro zobrazení animační prvků, které reagují na události (*Udalost*),

- statická aktivita existuje po celou dobu trvání animace (v našem případě doba trvání animace odpovídá času, po který Java Applet spuštěný), tento typ aktivity má význam pro zobrazování stavových informací logického procesu, jako je například aktuální stav fronta, atp.

Každá aktivita obsahuje kreslící metodu *draw()*, které je předán aktuální čas animačního modulu (viz níže) a kreslící scéna animace. Aktivita při vykonání metody *draw()* realizuje vizualizaci informace, kterou má aktivita zobrazit pro daný čas, na scénu animace. Každá aktivita informuje animační modul o čase, po kterém nemá význam, a je nutné ji odebrat ze seznamu aktivit.

Animační modul si udržuje vlastní hodnotu časového razítka. Aktuální zobrazený stav animační scény odpovídá stavu animačních aktivit, právě v tomto časovém okamžiku.

Samotná animace je realizována jako kreslení scén pro konkrétní hodnotu časového razítka animačního modulu, s tím že toto časové razítko se po každém vykreslení scény zvyšuje o konstantní přírůstek. Tento stav se opakuje, dokud časové razítko „nepřeroste“ vyhrazený interval samotným jádrem simulace.

6.7 Použité frameworky

Jako základní použitý JavaScriptový framework je použit **jQuery** framework, který je jednoduchý a velice rozšířený, více viz tabulka 5 – možnosti hledány na internetu s přispěním příspěvku v blogu [8].

Tabulka 5 – Porovnání frameworků pro JavaScript

Požadavek	jQuery	Prototype	Ext JS	MooTools	Script.aculo.us
Kód	Jednoduchý	Složitý	Složitý	Jednoduchý[69]	Jednoduchý
Licence		MIT License	Komerční	MIT License	Free
Dokumentace	Podrobná	Podrobná	Podrobná	Podrobná	Podrobná
Podpora AJAXu	Jednoduchá	Složitější než jQuery, ale jednoduché	Složitý	Jednoduchá	Viz prototype
Stránka projektu[70]	jquery.com	prototypejs .org	sencha.com/ products/extjs	mootools.net	script.aculo.us
Poznámka			Zaměřeno na vizuální komponenty		Založeno frameworku Prototype

Pro kreslení na <canvas> bylo rozhodnuto o využití frameworku především z důvodu složitého přístupu pomocí čistého JavaScriptového kódu. Porovnání vybraných možností viz tabulka 6, požadavky na framework byly vybrány s ohledem na řešený problém a rozsah realizované kresby. Výběr možností proběhl na základě zdroje [7] a hledání na internetu. Vybrané řešení – **fabric** – bude popsáno podrobněji v části 6.7.4.

Tabulka 6 – Porovnání frameworků pro kreslení na <canvas>

Požadavek	oCanvas	Raphaël	Processing.js	Paper.js	Fabric
Podklad	canvas	Svg	Canvas	Canvas	canvas, svg
Kód	jednoduchý	Jednoduchý	Složitý	jednoduchý	jednoduchý
Licence	MIT License	MIT License	MIT License	MIT License	Free
Dokumentace	Podrobná	Dostatečná	Komentované ukázky	Podrobná	Dostatečná
Ukázky	Ano, skromné	Ano	Ano	Ano	Ano
Interaktivní prvky	Ne	Ne	Ne	Ne	Ano
Uložení stavu	Ne	Ne	Nezjištěno (pravděpodobně ne)	Ne	JSON, Image (base64)
Stránka projektu[71]	ocanvas.org	raphaeljs.com	processingjs.org	paperjs.org	fabricjs.com

[69] Do značné míry totožný s jQuery, který je ale mnohem starší.
[70] Domovské stránky [online]. [cit. 2013-04-20] Dostupné z: http:// a adresa viz tabulka
[71] Domovské stránky [online]. [cit. 2013-04-28] Dostupné z: http:// a adresa viz tabulka

6.7.1 jQuery

Tento JavaScriptový framework slouží především pro zjednodušení přístupu a procházení hierarchické DOM HTML struktury stránek. Framework překonává rozdíly mezi prohlížeči a dovoluje tak vývojáři se soustředit na vytváření algoritmů na místo řešení kompatibility vytvořeného kódu mezi řadou prohlížečů. Framework lze jednoduše rozšiřovat o vlastní funkce. Již v základním sestavení lze vytvářet animace HTML prvků (především je jedná o změny hodnot atributů CSS bez užití nových vlastností CSS3). Je zde jednoduše implementovaná AJAX komunikace.

Další významnou výhodou je rozsáhlá komunita vývojářů a značné množství dostupných pluginů – kódů rozšiřujících funkcionalitu samotného frameworku.

Domovská stránka projektu: http://jquery.com/.

6.7.2 DIBI

Licenčně[72] velice dostupná abstraktní programová vrstva mezi přístupem k databázi a PHP. Ve skutečnosti se nejedná o skutečný framework, spíše nástroj či sadu programových komponent. Řešení DIBI poskytuje výkonný a přitom jednoduchý přístup k pokládání dotazů na databázi. Za jednu z hlavních výhod lze považovat automatickou ochranu dotazů proti útoku SQL Injection. Právě z tohoto důvodu bylo zvoleno toto řešení, neboť dotazy a zpracování výstupů z databáze jsou zpravidla velice jednoduchá a většina funkcí DIBI se aktivně nepoužívá.

DIBI podporuje více druhů databází, teoreticky dovoluje psát SQL dotazy nezávisle na databázi.

Jednou ze zajímavých vlastností DIBI je možné dynamicky skládat SQL dotazy – tento postup je nazýván *DibiFluent,* samo-vysvětlující ukázka kódu:

```
$res = dibi::select('id')
            ->select('name')->as('nazev')
            ->from('products')
            ->innerJoin('orders')->using('(product_id)')
            ->where("id > %i", $id)
            ->orderBy('title')
                  ->execute();
```

Na ukázce je vidět princip skládání dotazů pomocí metod představujících jednotlivá klíčová slova jazyka SQL. Také vidíme typovou kontrolu (předpis *%i* vyžaduje na vstupu *id* hodnotu

[72] Licence New BSD nebo GNU GPL 2 nebo GNU GPL 3.

odpovídající celému číslu) vstupu do části *where* proměnou *id*, a právě toto je jeden z možných způsobů ochrany pro SQL Injection, který je v dynamicky typovém PHP zásadní. V proměnné *id* by mohl totiž mohlo být číslo, které očekáváme, nebo například řetězec, který již může být potenciálně nebezpečný.

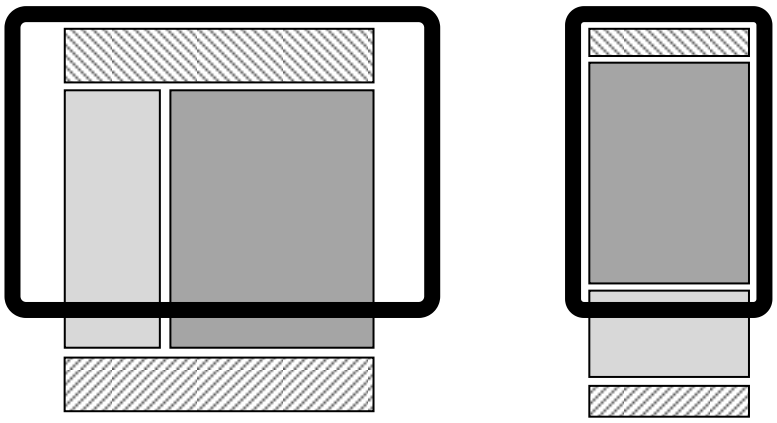
Domovská stránka projektu: http://dibiphp.com/.

6.7.3 Twitter Bootstrap

Twitter Bootstrap je volně dostupná[73] sada nástrojů pro vytváření vzhledu a vybraných dynamických prvků s přispěním JavaScriptu.

Responsivní design

Bootstrap od verze 2.0 podporuje responzivní design. Takovýto design zobrazuje stejný obsah vizuálně různě sestavený podle velikosti plochy zobrazovacího zařízení – typicky monitor, dnes často i displej mobilního telefonu či tabletu – viz obrázek 10. Takovouto funkcionalitu není problém vytvořit, například pomocí JavaScriptu, avšak přímá kontrola a sestavení kódu pomocí JavaScriptu se běžně nerealizuje, protože vlastnosti CSS umožňují určit vzhled jedné struktury HTML kódu podle rozlišení displeje bez zásahu do HTML struktury kódu.



(a) monitor (b) telefon Zdroj: Autor

Obrázek 10 – Ukázka responzivního designu na běžném designu stránky s levým menu a obsahem

Někdy rozlišujeme tři úrovně responsivního designu[74]:

- struktura sestavená z prvků s rozměry zadanými v procentech,
- dynamická velikost obrázků, příklad:

[73] Apache License v2.0, některé součásti distribuované pod jinými licencemi, většinou téměř neomezující užití.
[74] Platí pro použití internetových technologií, responzivní design lze realizovat v jakékoliv prostředí.

```
img { /* nepoužívat width a height atributy tagu img */
        max-width: 100%; height: auto;
}
```

- **Media Queries** – nová vlastnost CSS3 [14], která dovoluje „nastylovat“ design podle rozlišení displeje, ukázka:

```
@media (max-width: 1280px) and (min-width: 960px){
        body {
                background-color: gray;
        }
}
```

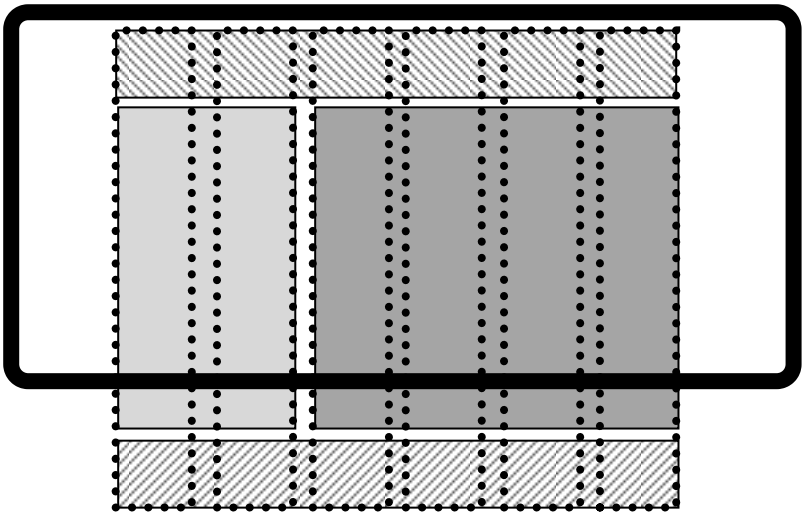
Bootstrap má tuto funkcionalitu přímo zabudovanou do základních layoutů a komponent, u který je tato „vlastnost“ očekávána, například přeskupení horizontální menu dle šířky displeje.

Grid framework

Způsob komponování internetových stránek. Myšlenka je založena na rozdělení plochy, ve které bude obsah do sloupců o stejné šíři. Výsledkem je přesné rozmístění prvků designu stránky a z toho důvodu vizuálně působivý výsledek. Používají se dva typy:

- **fluid grid system**: obsah přes celou plochu displeje,
- (static) **grid system**: pevná šířka obsahu bez ohledu na velikost displeje, ale často s ohledem na responzivní design.

Bootstrap používá 12 sloupců na plochu. Do této „mřížky“ se umisťují jednotlivé komponenty s rozměry odpovídajícími násobkům velikosti jednoho sloupce, ukázka viz obrázek 11.



Zdroj: Autor

Obrázek 11 – Šesti sloupcový grid layout

Základní prvky

Twitter Bootstrap obsahuje předpřipravené CSS předpisy pro celou řadu HTML prvků, které se běžně vyskytují:

- základní typografické prvky – nadpisy, velikosti a typ písmem pro různá použití, formát adresy, ...
- tabulky,
- formulářové prvky – tlačítka, vstupní prvky, ...
- ikonky,
- a jiné.

Vše je připraveno tak, aby vizuálně ladilo. Použití těchto prvků je možné i pro internetové stránky či aplikací s vlastním původní design, neboť na oficiálních stránkách frameworku lze nastavit spoustu parametrů dle vlastního uvážení. Pro administraci vytvořenou v rámci této práce je tento předpřipravený CSS vzhled velice vhodný a je jedním z důvodů, proč bylo přistoupeno k použití tohoto frameworku.

Komponenty a JavaScript

Framework sám používá jQuery, což je výhodné, neboť administrace bude obsahovat pouze jeden framework pro JavaScript.

Bootstrap obsahuje celou řadu složitých dynamických HTML struktur pro nejrůznější použití:

- horizontální menu,
- informační hlášky,
- složitá tlačítka,
- dropdown menu,
- a spoustu dalšího.

Tyto prvky lze často bez problémů kombinovat. Některé prvky je možné ovládat pomocí JavaScriptu, a tak docílit dynamického chování aplikace bez nutnosti vytvářet vlastní kód, případně původní možnosti dynamického chování rozšiřovat.

Domovská stránka projektu: http://twitter.github.io/bootstrap/.

6.7.4 Fabric

Zcela volně dostupný Fabric je framework pro kreslení na tag <canvas> (viz část 5.4.1), dále pouze canvas. Jeho výhodou je, že práce s canvasem, na rozdíl od samotné implementace prohlížeče, připomíná kreslení ve vyšším objektovém jazyce jako je C# nebo Java. Framework přímo umožňuje realizovat:

- interaktivně pracovat s prvky v canvasu (měnit velikost a pozici, otáčení),
- jednoduché vytváření grafických primitiv,
- vkládání obrázků,
- animace,
- odchyt událostí,
- podpůrné funkce pro výpočty (např. zjištění prvků na zadané souřadnici),
- možnost exportu stavu Fabriku – kresby,
- a další.

Výhodou také je, že je možné přistupovat ke kontextu (samotný obsah, na který se kreslí) canvasu i přímo pomocí JavaScriptu bez vědomí Fabricu. Tímto způsobem byly realizovány například kulaté rohy samostatných čar. Je nutné zmínit, že složitější zásahy zpravidla vedou k následným chybám samotného Fabricu, neboť se mění kontext canvasu bez vědomí frameworku.

Většina výše uvedených vlastností byla plánována k užití v designeru simulace. Při realizaci, ale bylo zjištěno, že některé tyto vlastnosti (např. změna pozice) nelze jednoduše použít v náročnější implementaci z důvodu zbytečně složitých konstrukcí či pomalého chodu takové řešení.

Posuny objektů, odchytávání událostí a dynamické kreslení čar (spojnic modelů) byly realizovány mimo hlavní možnosti Fabricu.

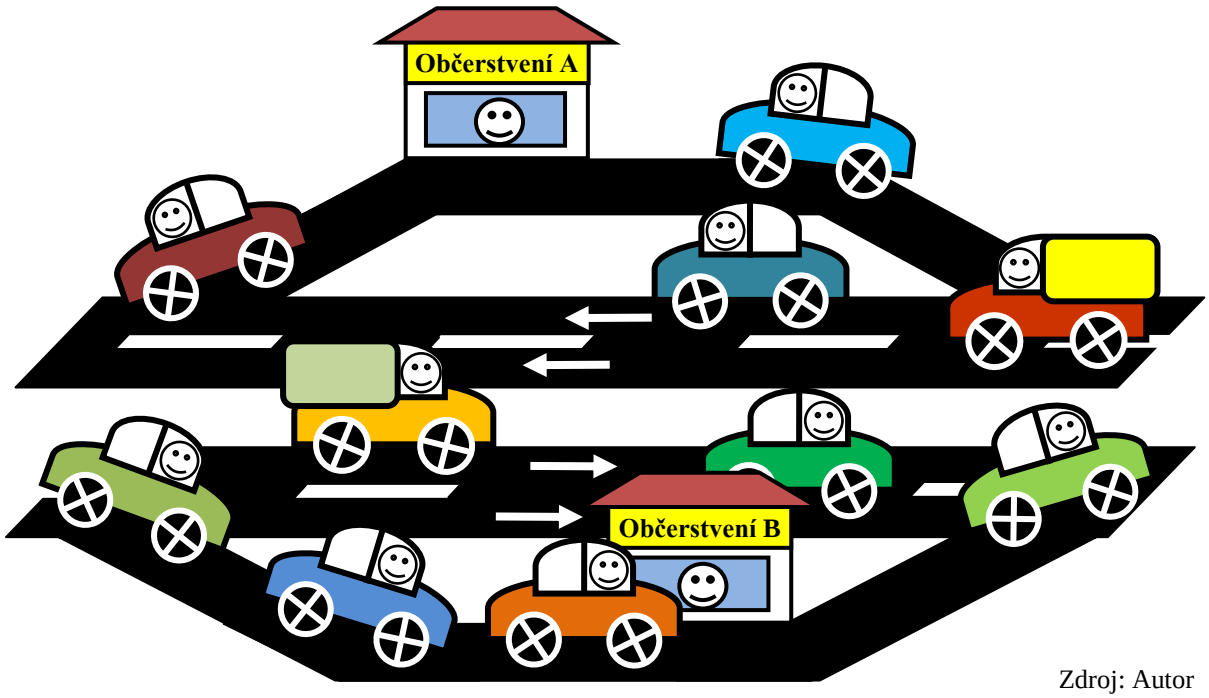
Domovská stránka projektu: http://fabricjs.com/.

7 MODELOVÝ PŘÍKLAD

Softwarový nástroj pro konfigurování distribuovaných simulačních (Administrace) je nástroj obecný, schopný vytvořit jakoukoliv konfiguraci v rámci možných komponent (konkrétně Java Appletů). Pro předvedení formální správnosti byl vybrán jeden konkrétní model. Pouze jedna ukázka je považována za dostačující vzhledem k tomu, že jednotlivé logické procesy (pracovní stanice) obsahují animační modul a chování nakonfigurovaného systému je možné ověřit[75] vizuálně.

7.1 Představení

Správnou činnost distribuovaného simulačního modelu ověříme na případu dálničních občerstvení (viz obrázek 12), kde dálnice (oba směry) jsou realizovány jedním logickým procesem (LP_D) a občerstvení pro každý směr A a B dvě dalšími (LP_A a LP_B).



Zdroj: Autor

Obrázek 12 – Ilustrační obrázek dálnice s občerstveními

[75] Samozřejmě ne vše lze vizuálně ověřit, ale testovací logické procesy jsou jednoduché a jednotlivé aktivity jsou ve většině případů podrobně animovány. Jednotlivé pracovní stanice neobsahují velké možnosti nastavení, a i ty lze pouhým pohledem verifikovat. Lze říci, že zasvěcený uživatel dokáže posoudit správnost činnosti a běhu simulace.

Shrnutí simulační činnosti (ve směru A):

Logický proces LP_D:

1. Vozidlo přijede na odbočku ve směru A.
2. S určitou pravděpodobností vozidlo odbočí na občerstvení:

    Obsluhuje logický proces LP_A:

    a. Vozidlo přijede.
    b. Vozidlo je zařazeno do fronty.
    c. Pokud občerstvení neobsluhuje žádné vozidlo, je zahájena obsluha prvního vozidla ve frontě, pokud takové existuje.
    d. Po dokončení obsluhy vozidla občerstvením, vozidlo odjíždí zpět na dálnici (LP_D):

        Obsluhuje logický proces LP_D

        i. Vozidlo přijede na nájezd z občerstvení.
        ii. Vozidlo dále pokračuje bodem LP_D č. 4.

3. Pokud vozidlo neodbočuje, pokračuje po dálnici a projíždí nájezdem na dálnici z občerstvení ve směru A.
4. Vozidlo z nájezdu odjíždí pryč ze sledované části dálnice ve směru A.

Ve směru B je průběh algoritmu totožný s tím, že LP_D generuje vozidla ve dvou proudech pro směr A a B. Tento „centrální“ LP_D nám zajistí potřebu synchronizovat čas u všech třech LP.

7.2 Validace a verifikace modelu

Tento konkrétní příklad byl realizován v softwarovém nástroji ARENA[76] (Rockwell Software) – viz obrázek 13. Na obrázku je realizace simulace pouze dvou logických procesů (jednoho směru dálnice), neboť jeden směr dálnice nemá vliv na druhý. Entity představují vozidla.

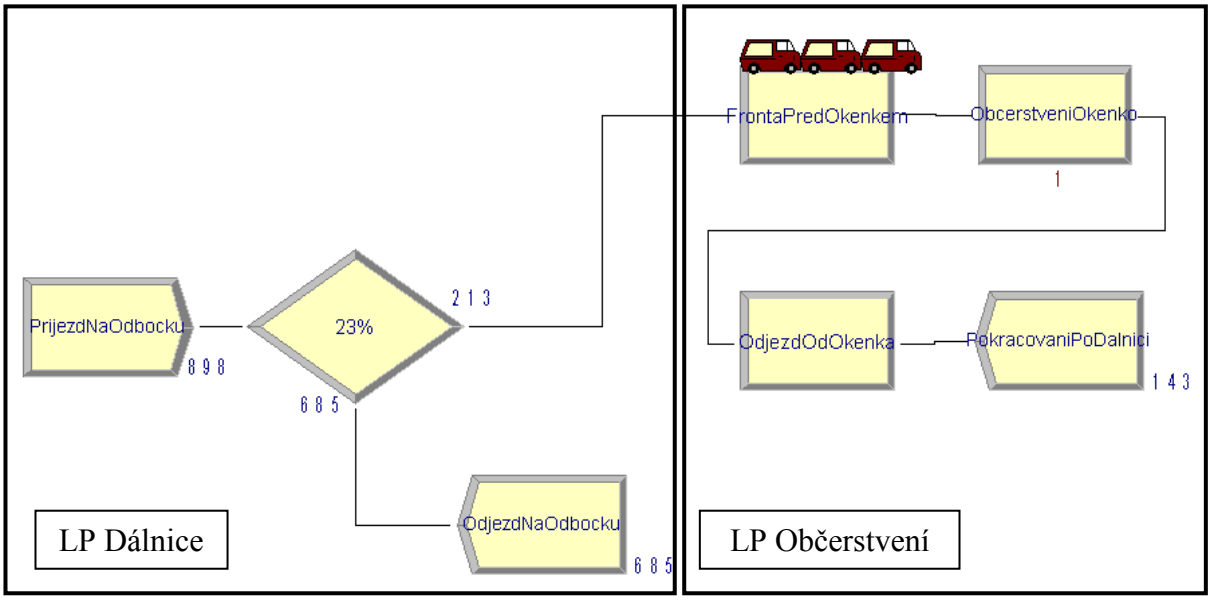
Generátory náhodných čísel, intervaly mezi entitami, atp. byly nastaveny stejně jako ve vlastní realizaci v programovacím jazyce Java (konkrétní hodnoty viz tabulka 7). Vstupy a výstupy byly v obou případech ekvivalentní, takže můžeme prohlásit, že model byl validován oproti komerčnímu a prověřenému řešení.

[76] Více informací na http://www.arenasimulation.com, pro testování použita verze licence Student.

Výstupy entit v ARENA i vlastním řešení odpovídají vstupním hodnotám, stejně tak jako jejich rozdělení v rámci logického větvení programu (simulační činnosti).

Tabulka 7 – Nastavení modelového příkladu

Logický proces	Položka	Hodnota
Dálnice	Pravděpodobnost odjezdu	23 %
Občerstvení	Interval časů obsloužení u okénka občerstvení	$\langle 20000; 30000 \rangle$ ms



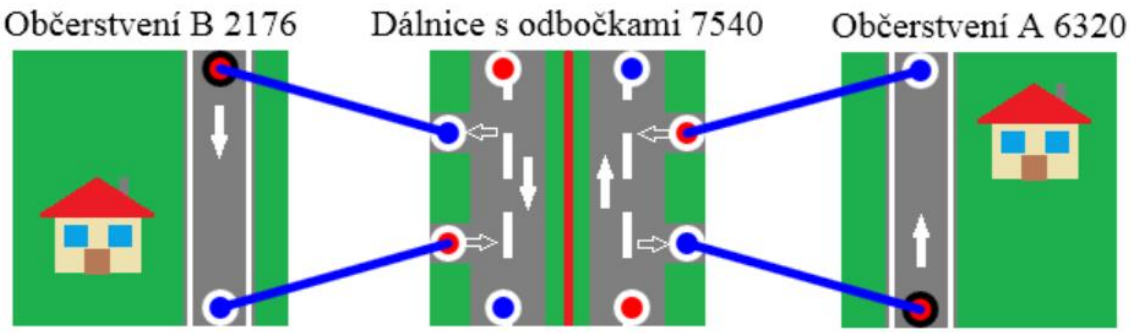
Zdroj: Autor

Obrázek 13 – Verifikační model v ARENA s rozdělením logických procesů

7.3 Realizace

Na obrázku 14 je vidět vizualizace konfigurace distribuovaného simulačního modelu pro modelový příklad. Výsledné Java Applety jsou zobrazeny v příloze F. Tyto tři logické procesy jsou ve skutečnosti realizovány jako dva logické procesy a to:

- logický proces Dálnice
- a logický proces Občerstvení, který se vykresluje dvěma způsoby – dle použití.



Zdroj: Autor

Obrázek 14 – Ukázka modelového případu v Designeru

Realizace samotného běhu je rozdělena do několika činností (které provádějí události, viz podkapitola 1.3), jejich schéma je zobrazeno v příloze G.

Tabulka 8 – Porovnání sledovaných hodnot modelového příkladu napříč různými implementacemi (průměry 10 replikací)

	Počet vstupních entit	**Počet* výstupních entit (dálnice/odbočka)**	**Průměrná délka čekání na obsluhu**	**Průměrná doba u okénka**
ARENA	901	685/210	533 s	25 s
Nedistribuovaná verze / Jedno vlákno	900	695/205	525 s	25 s
Distribuovaná řešení / Ve vláknech	900	694/207	514 s	25 s
Distribuovaná řešení / Vzdálené logické procesy, komunikace po síti	900	691/208	529 s	25 s

* Počet výstupních entit dálnice/odbočka by měl být stejný jako počet vstupních hodnot, drobné rozdíly jsou způsobeny vozidly, které v čase ukončení zůstaly „na cestě“

V podkapitole 3.5 je dokázáno, že tento příklad není vhodný pro distribuované zpracování z důvodu velmi časté nutnosti synchronizace logických procesů – a tím zpomalení výpočtu, nicméně právě z tohoto důvodu je to vhodná úloha pro testování synchronizačních technik.

8 ZÁVĚR

V rámci této práce byl vytvořen softwarový nástroj pro tvorbu libovolného distribuovaného simulačního modelu z množiny dostupných prvků, jejichž technické požadavky byly podrobně zdokumentovány, aby bylo možné softwarový nástroj doplňovat a zvyšovat do budoucna jeho hodnotu.

Při realizaci práce bylo zjištěno několik problémů a omezení zvolených technologií. Většinu problémů, případně nedostatků či technologických omezení – především týkajících se Java Appletů, se podařilo překonat či vytvořit alternativní řešení tak, aby byl zachován ráz a cíle práce mohly být bezpodmínečně splněny.

Pro demonstraci výsledků a možností softwarového produktu byly vytvořeny (a důkladně otestovány) Java Applety představující prvek dálniční sítě a na něho navazující jednotky občerstvení typu *drive-in* se zachováním maximální kompatibility pro budoucí rozšiřování základny appletů a teoretické tvorby libovolného distribuovaného modelu.

Cíle práce byly splněny v celém rozsahu, s tím, že práce i přesto nabízí prostor pro další rozšiřování jak konfiguračního nástroje, tak především samotných appletů, na kterých stojí výpočty a realizace distribuované simulace. Tato rozšiřitelnost je nejvyšší přidanou hodnotou práce, neboť byly vytvořeny a podrobně popsány obecné principy a postupy, bez problémů realizovatelné v jakémkoliv moderním programovacím jazyce, pro rozšiřování a realizaci simulací, bez ohledu na použitou technologii, programovací jazyk, prostředí internetového prohlížeče či desktopovou aplikaci, ale také další řešení synchronizačních technik a obecně principů realizace distribuovaného simulačního modelu. Tím je zaručeno budoucí využití aplikace a růst ekosystému appletů a jiných komponent – pro příklad serverových prvků. Je nutné kriticky uvést, že ne vše je implementováno – především z důvodu značného rozsahu práce – jako uživatelsky přístupné, avšak konfigurační nástroj, až na drobné výjimky, interně se vším výše popsaným pracuje a počítá, a jako takový je koncipován pro jednoduché doplnění chybějících uživateli dostupných funkcí, a je tak velice dobře připravena pro další rychlý, dynamický a hlavně efektivní rozvoj.

9 LITERATURA

[1] KAVIČKA, Antonín. *Pokročilé techniky modelování a simulace: Elektronické sylaby*. Pardubice: Univerzita Pardubice, 2011.

[2] FUJIMOTO, Richard M. GEORGIA INSTITUTE OF TECHNOLOGY. *Parallel and Distributed Simulation Systems*. Wiley Interscience, USA, 2000.

[3] HŘÍDEL, Jan a Štěpán KARTÁK. *Web jako platforma pro simulaci* [online]. Elektrorevue, 2012, roč. 2012, č. 69, s. 1-8 [cit. 2013-05-06]. ISSN 1213-1539. Dostupné z: http://www.elektrorevue.cz/cz/clanky/informacni-technologie/0/web-jako-platforma-pro-simulaci-1/

[4] ORACLE. The Java™ Tutorials. *Lesson: Java Applets* [online]. 2013 [cit. 2013-05-06]. Dostupné z: http://docs.oracle.com/javase/tutorial/deployment/applet/index.html

[5] ORACLE. The Java™ Tutorials. *Lesson: Packaging Programs in JAR Files* [online]. 2013 [cit. 2013-05-06]. Dostupné z: http://docs.oracle.com/javase/tutorial/deployment/jar/index.html

[6] ORACLE. The Java™ Tutorials. *Lesson: Deployment In-Depth* [online]. 2013 [cit. 2013-05-06]. Dostupné z: http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/index.html

[7] LEAFCUTTER. *Canvas Drawing Frameworks* [online]. [cit. 2013-04-28]. Dostupné z: http://leafcutter.com.au/labs/2013/03/canvas-drawing-frameworks/

[8] *Alternatives to JQuery* [online]. [cit. 2013-04-28]. Dostupné z: http://www.jscripters.com/popular-jquery-alternatives/

[9] W3C. *Extensible Markup Language (XML)* [online]. 2012 [cit. 2013-05-06]. Dostupné z: http://www.w3.org/XML/

[10] W3C. *XML Path Language (XPath): Version 1.0* [online]. 1999 [cit. 2013-05-06]. Dostupné z: http://www.w3.org/TR/xpath/

[11] W3C. *HTML 5.1 Nightly: A vocabulary and associated APIs for HTML and XHTML* [online]. 2013 [cit. 2013-05-06]. Dostupné z: http://www.w3.org/html/wg/drafts/html/master/single-page.html

[12] IETF. *RFC: 768: User Datagram Protocol* [online]. [cit. 2013-05-08]. Dostupné z: http://www.ietf.org/rfc/rfc768.txt

[13] IETF. *RFC: 793: Transmission Control Protocol* [online]. [cit. 2013-05-08]. Dostupné z: http://www.ietf.org/rfc/rfc793.txt

[14] *W3C Standards: CSS* [online]. W3C [cit. 2013-05-08]. Dostupné z: http://www.w3.org/TR/#tr_CSS

10 PŘÍLOHY

Příloha A: Testovací konfigurace

Windows 7 Ultimate SP1 64 bit; AMD Phenom^{tm} II X4 955 3,21 GHz; 12 GB RAM

Konkrétní prohlížeče, jsou pojmenovány dle jejich vlastního uvedení v titulku okna prohlížeče. Následují běhová prostředí PHP a Java.

Nejjednodušší způsob identifikace prohlížeče, pokud tuto informaci neposkytuje sám, je vytvořit jednoduchý skript, který tuto informaci zjistí:

```
<?php // Kód PHP
echo $_SERVER['HTTP_USER_AGENT'];
?>
```

Opera

Verze: 12.15; Sestavení: 1748; Win32

Identifikace: Opera/9.80 (Windows NT 6.1; WOW64) Presto/2.12.388 Version/12.15

Mozilla Firefox

Verze: 19.0.2

Identifikace: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:19.0) Gecko/20100101 Firefox/19.0

Google Chrome

Sestavení: 26.0.1410.64 (Oficiální sestavení 193017)

Identifikace: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.31 (KHTML, like Gecko) Chrome/26.0.1410.64 Safari/537.31

Windows Internet Explorer (MSIE)

Verze: 9.0.8112.16421

Identifikace: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)

PHP

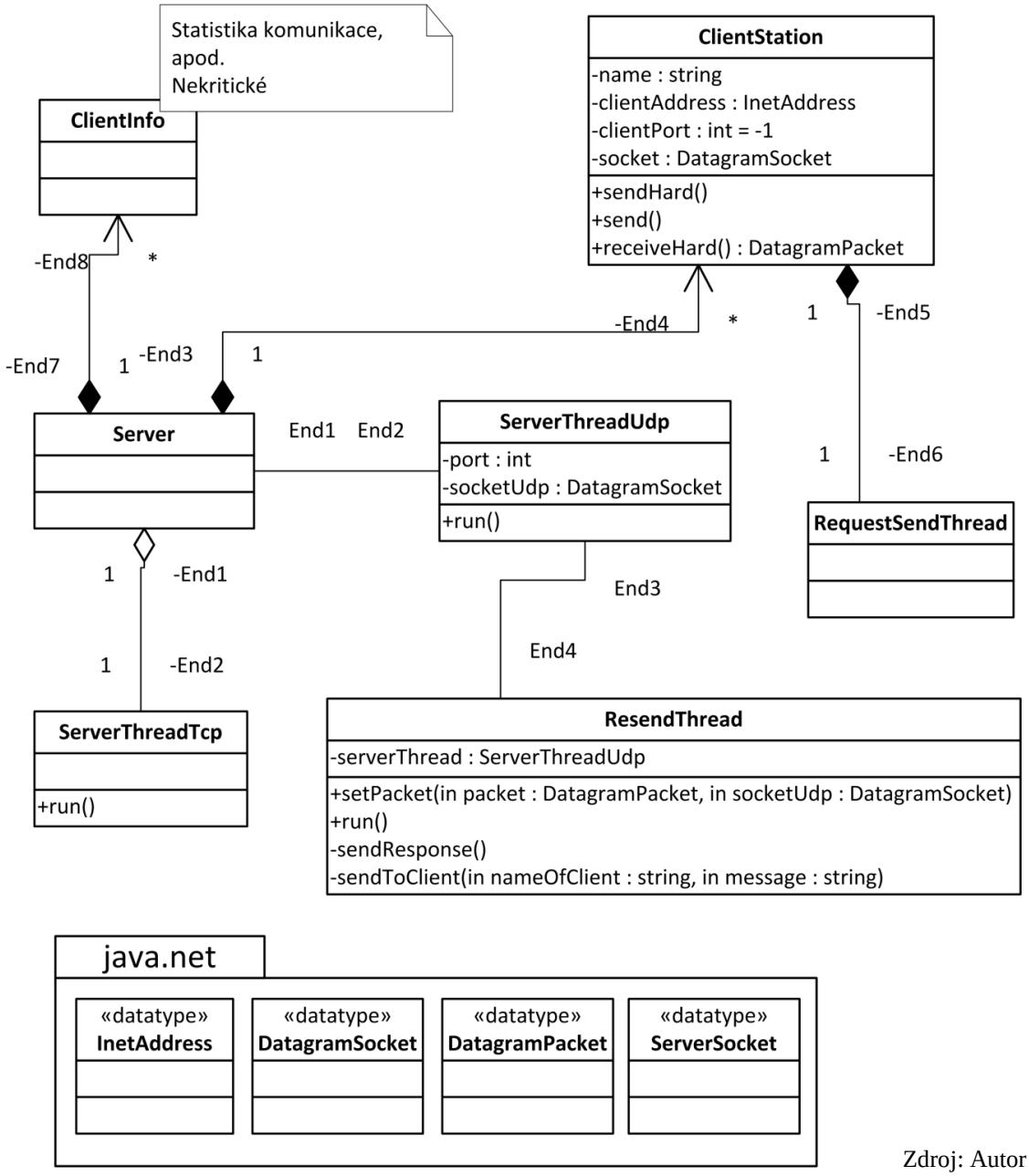
PHP Version 5.3.8

Java

Java version "1.7.0_02", Java(TM) SE Runtime Environment (build 1.7.0_02-b13)

Příloha B: UML diagramy

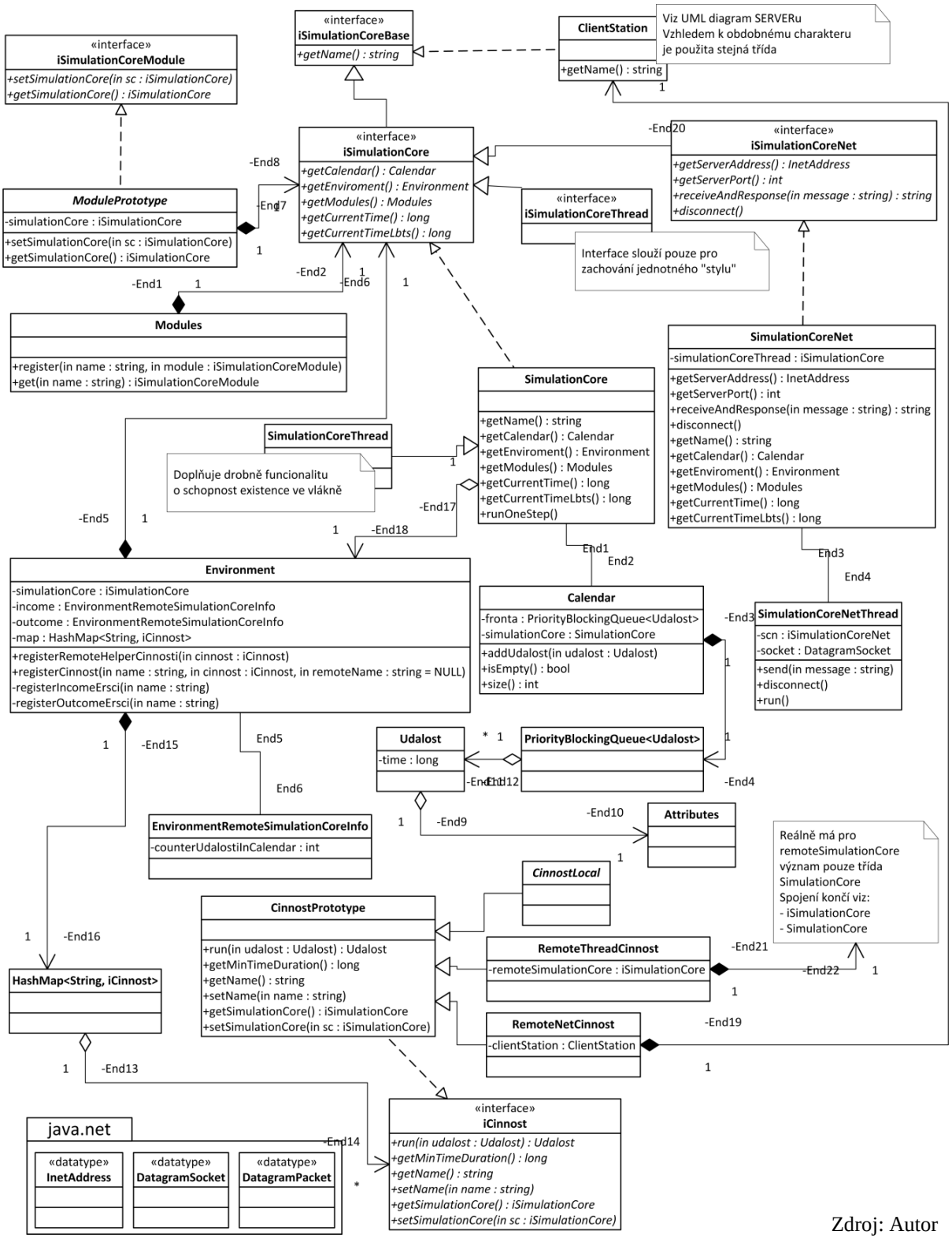
Zobrazeny pouze zásadní[77] třídy, metody a atributy. Kompletní řešení obsahuje přes 130 tříd a jiných prvků (enum, statické helper třídy, apod.). Diagramy jsou vytvořeny v nástroji Microsoft Visio, který je primárně určen pro modelování objektů realizovaných v technologiích[78] společnosti Microsoft, ale vzhledem k obecným principům standardu UML je použitelný i pro návrh programu realizovaného v jazyce Java.



Zdroj: Autor

Obrázek 15 – UML serveru

[77] Zpravidla nejsou uvedeny settery, gettery, iterátory, atributy udržující vazby na objekty, pokud je tento stav jasný z diagramu. Výjimečně jsou metody zjednodušeny.
[78] Například jazyk C#, Visual Basic, aj.



Zdroj: Autor

Obrázek 16 – UML logického procesu v rámci distribuované simulace

Příloha C: Obsah CD

Elektronická verze této práce

Umístění: //KartakS_SoftwarovyNastroj_JH_2013.pdf

Manuál správce Administrace

Tento dokument podrobně popisuje práci a možnosti konfigurace appletů a realizaci sezení Administrace.

Umístění: //KartakS_SoftwarovyNastroj_JH_2013_manual_spravce_administrace.pdf

Příručka vývojáře appletu

Detailní informace o konfiguraci, procesech a požadavcích na applet, použitelný s Administrací naleznete v dokumentu Příručka vývojáře appletu.

Umístění: //KartakS_SoftwarovyNastroj_JH_2013_prirucka_vyvojare_appletu.pdf

Aplikace Administrace

Zdrojové kódy se dále nekompilují, jsou přímo spouštěny serverem s PHP.

Umístění: //sources/www/

Java aplikace zahrnující server a applety

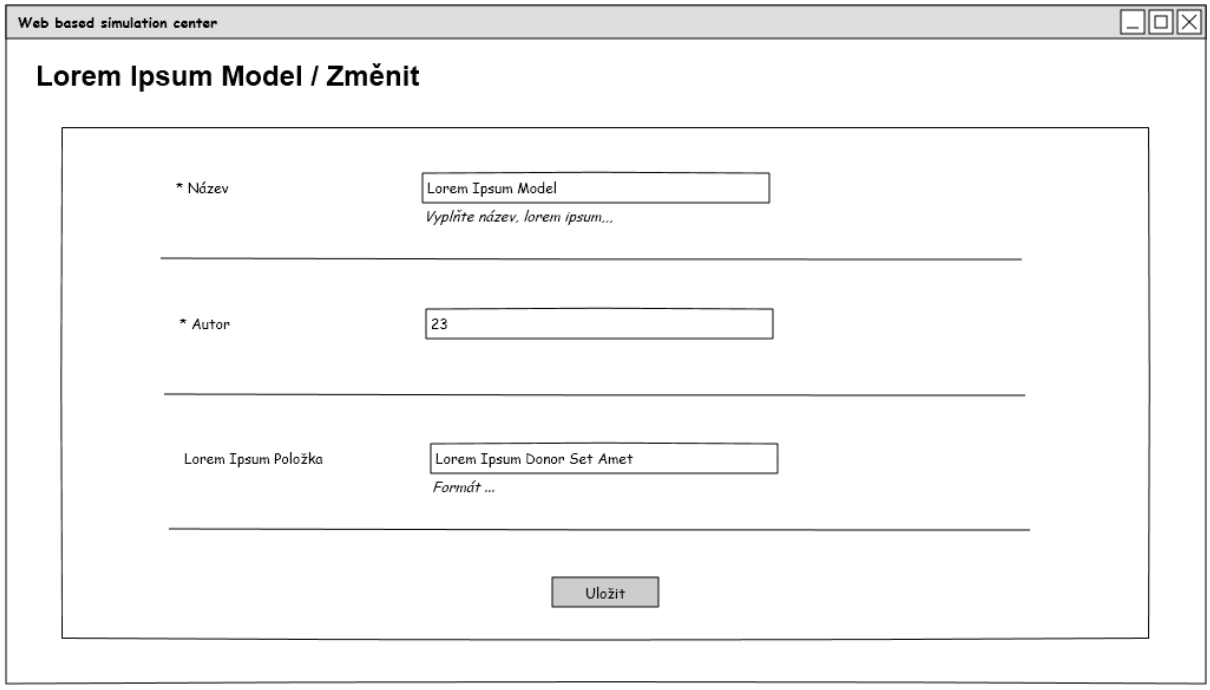
Umístění: //sources/SimulaceBase/

- **Spustitelný server:** //SimulaceBase/build/classes/, *main* třída: remote/Server.class
- **Java Applety:**
    - Dálnice: //SimulaceBase/build/classes/, *main* třída: run/AppletDalnice.class
    - Občerstvení A: //SimulaceBase/build/classes/,
      main třída: run/AppletObA.class
    - Občerstvení B: //SimulaceBase/build/classes/,
      main třída: run/AppletObB.class
- **Zdrojové kódy:** //SimulaceBase/src/

Další přílohy a materiály

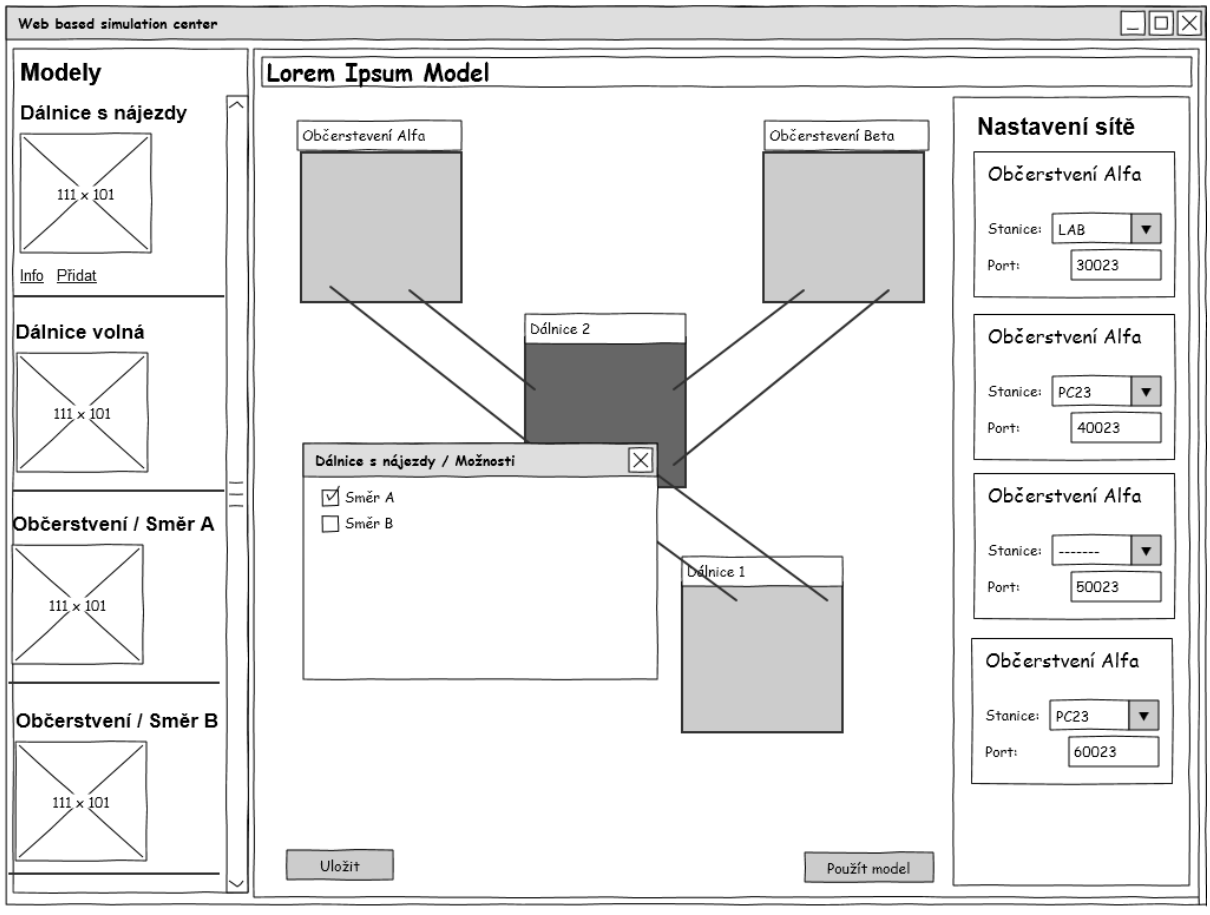
Umístění: //extra/

Příloha D: Návrh a koncepce Administrace



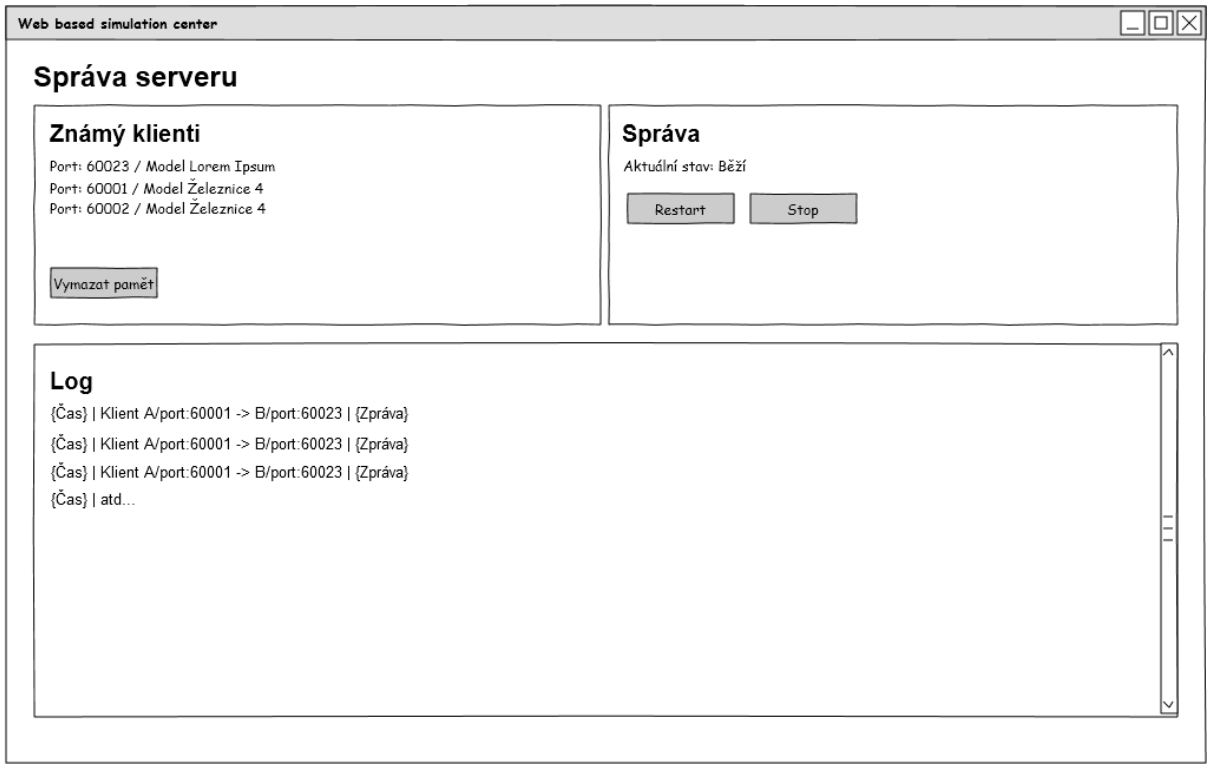
Zdroj: Autor

Obrázek 17 – Běžná formulářová stránka (návrh)



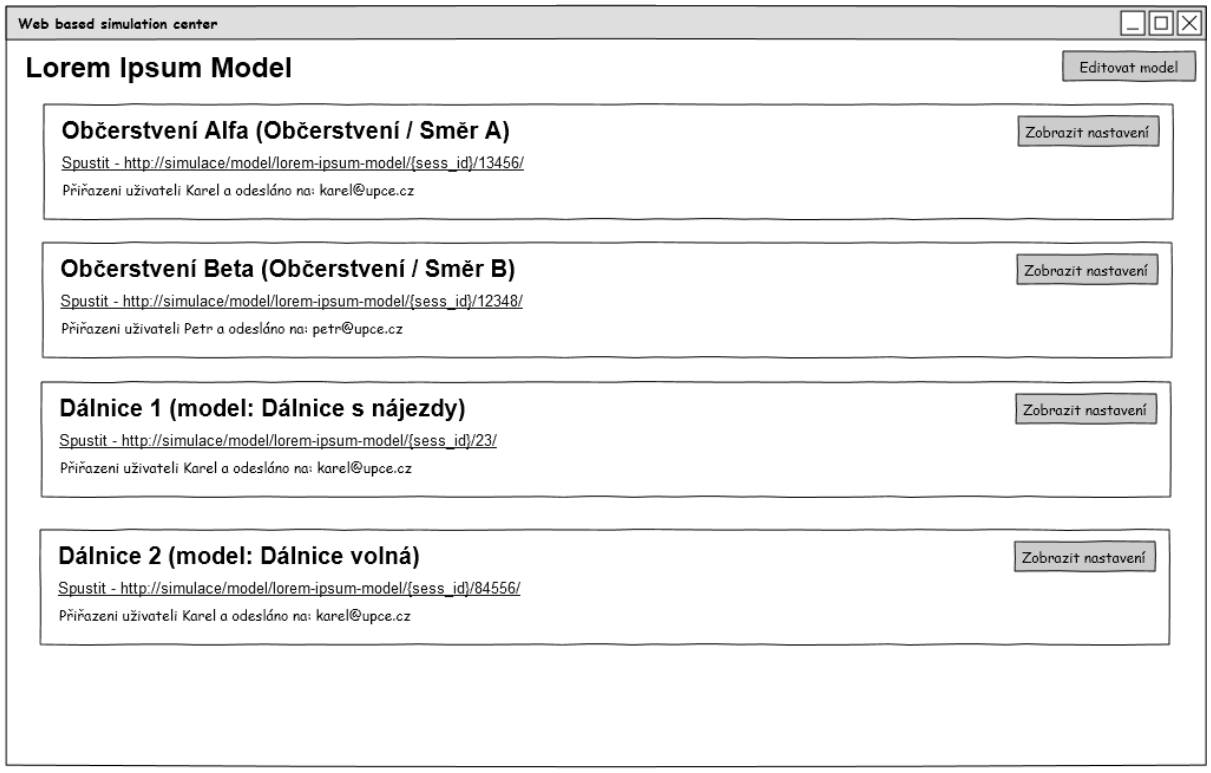
Zdroj: Autor

Obrázek 18 – Designer (návrh)



Zdroj: Autor

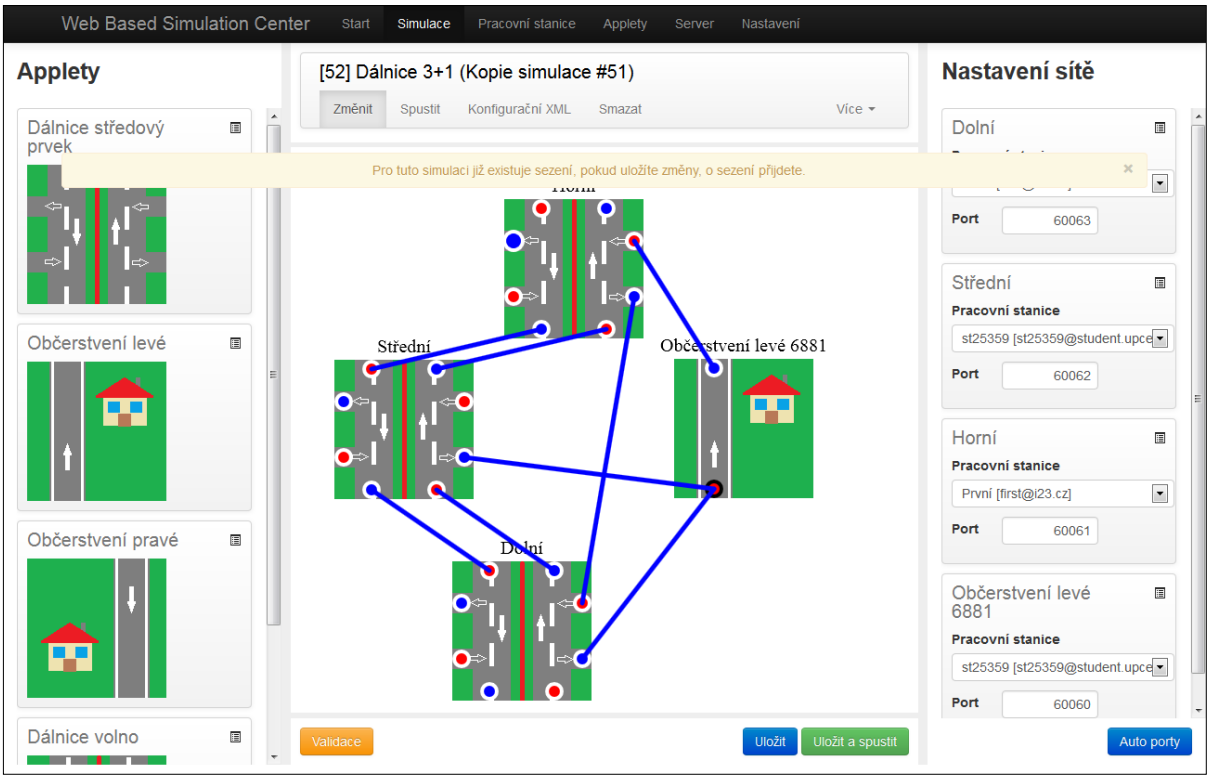
Obrázek 19 – Správa serveru (návrh)



Zdroj: Autor

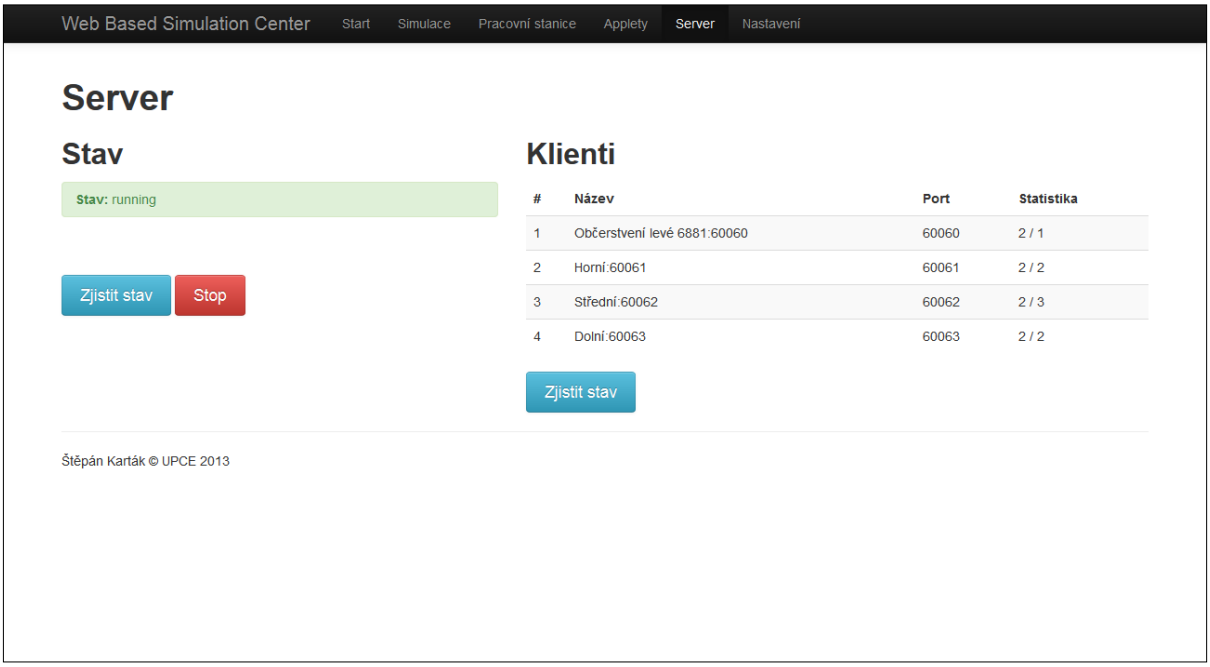
Obrázek 20 – Použití nakonfigurovaného simulačního modelu (návrh)

Příloha E: Ukázka výsledné aplikace



Zdroj: Autor

Obrázek 21 – Ukázka sestavení simulace – tzv. Designer



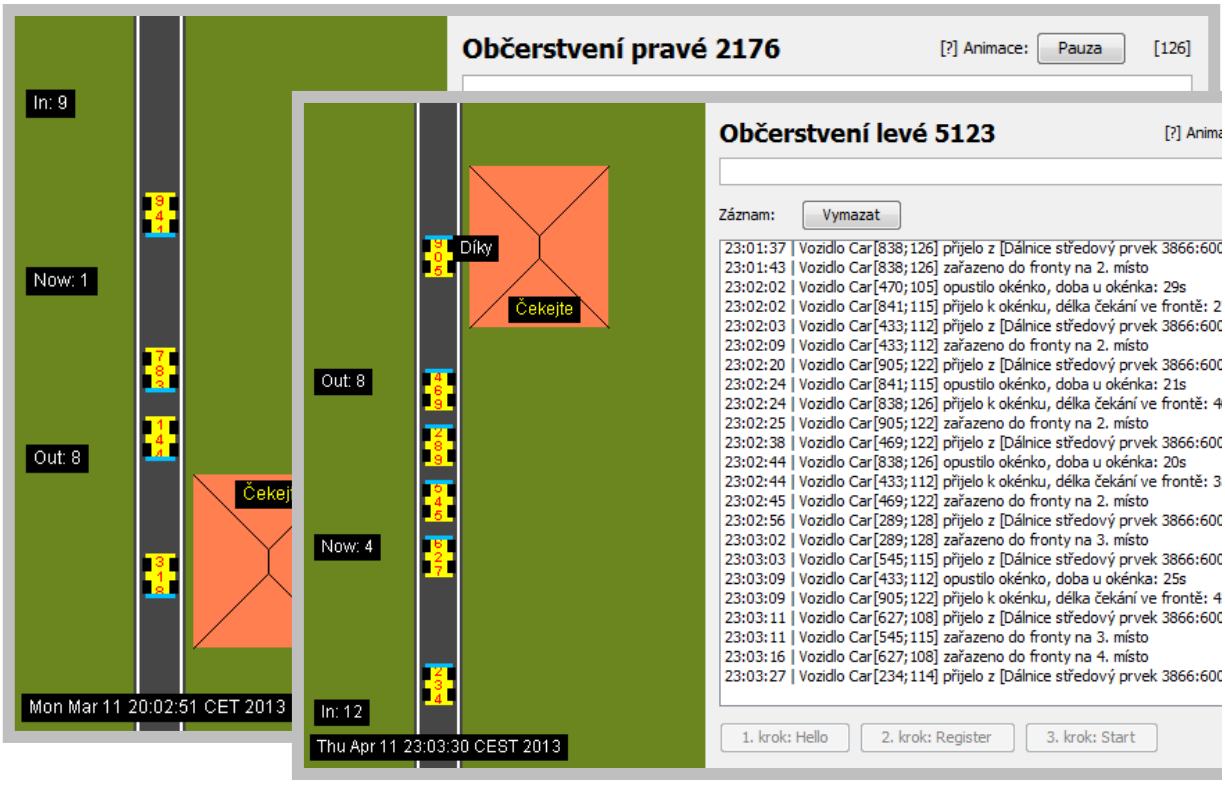
Zdroj: Autor

Obrázek 22 – Správa serveru

Příloha F: Java Applety modelového příkladu

Zdroj: Autor

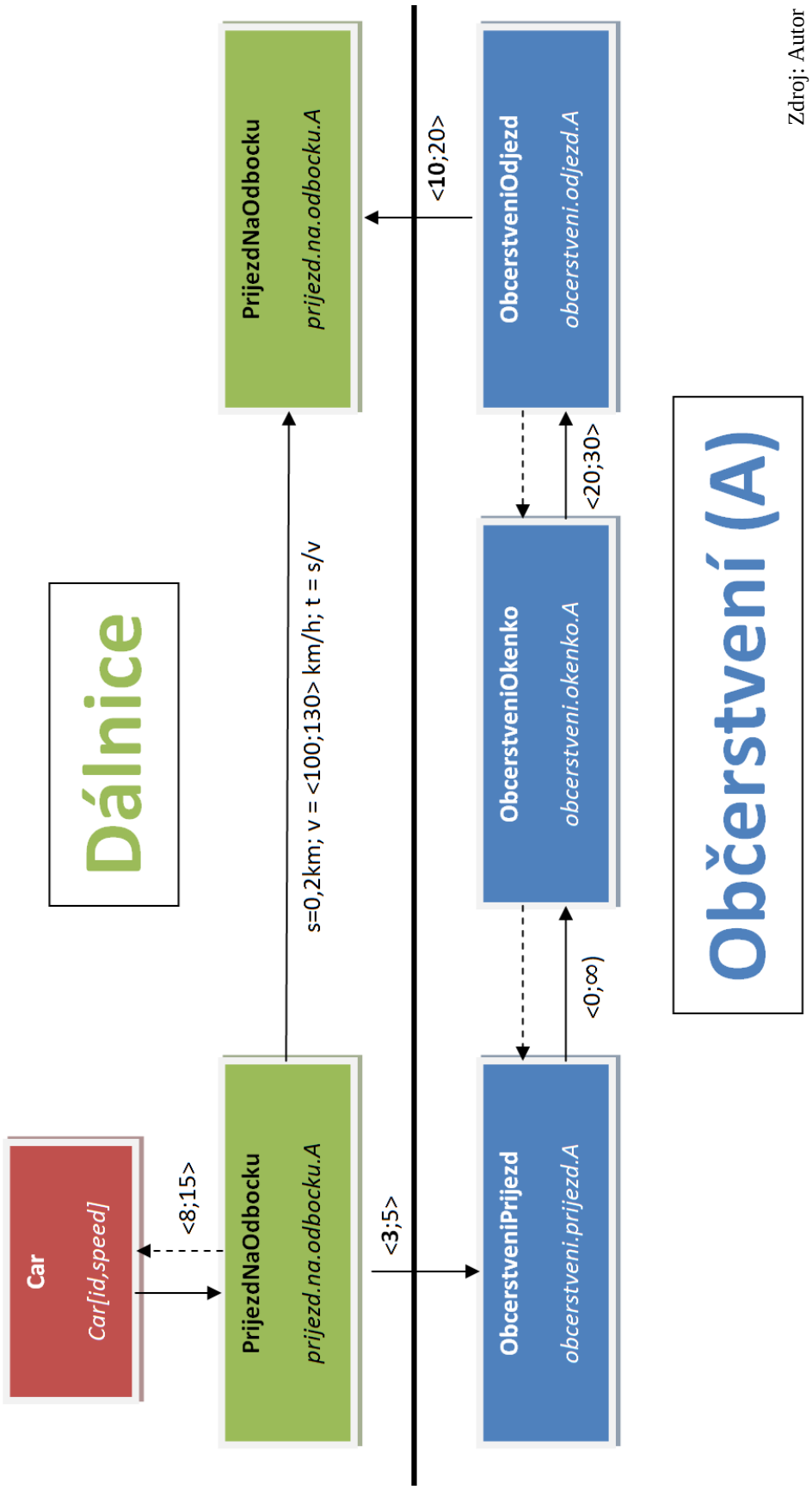
Obrázek 23 – Logický proces Dálnice (Java Applet)



Zdroj: Autor

Obrázek 24 – Logické procesy Občerstvení A a Občerstvení B (Java Applety)

Příloha G: Schéma činností modelového příkladu



Zdroj: Autor

Obrázek 25 – Schéma činností modelového příkladu Dálnice a Občerstvení *A*