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

Diplomová práce

Rendez-vous v Javě

Plzeň, 2010 Miroslav Hendrych

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

V Plzni dne 19. května 2010 ……

Miroslav Hendrych

Abstract

A complex application usually utilizes a set of threads to perform its activities. These threads provide the application with the capability to execute several code fragments in parallel and independently on each other.

However, if two or more threads want to exchange some data, the developer has to deploy some mechanism that allows him to synchronize the execution of the threads to ensure the data integrity. In Ada programming language, the mechanism is named rendez-vous.

This thesis describes an implementation of similar functionality in the Java programming language and the consequent integration of this functionality into an environment of active networks. Namely, rendez-vous has been implemented into Smart Active Node project.

Obsah

1 Úvod ... 1
2 Seznámení s jazykem Ada ... 2
2.1 Ada a bezpečnost ... 2
2.2 Shrnutí programovacího jazyka Ada ... 5
2.3 Vlákna a různé programovací jazyky ... 5
2.4 Ada a vlákna ... 6
2.5 Různé varianty rendez-vous ... 9
2.6 Shrnutí rendez-vous v jazyce Adě ... 12
3 Rendez-vous v jazyce Java ... 12
3.1 Jak vytvořit volání předem neznámých metod ... 13
3.2 Jak předat metodě libovolný počet parametrů ... 15
3.3 Prostředky pro synchronizaci v Javě ... 16
3.4 Jak fungují vlákna ... 17
3.5 Synchronizované FIFO ... 19
3.6 Jak identifikovat jednotlivá entry ... 22
3.7 Selektivní akcepty ... 23
3.8 Podmíněné akcepty ... 23
3.9 Časově omezený akcept ... 25
3.10 Otestování výsledků ... 27
3.11 Shrnutí dosaženého výsledku ... 28
4 Projekt SAN ... 30
4.1 Aktivní sítě ... 30
4.2 Možnosti využití aktivních sítí ... 32
4.3 Bezpečnost aktivních sítí ... 33
4.4 Architektura projektu SAN ... 34
4.5 Vytváření aktivních programů v projektu SAN ... 36
4.6 Spouštění aplikací v projektu SAN ... 37
4.7 Interpretování aplikací ... 38
4.8 Interprety programů ... 39
4.9 Projekt SAN a interprety ... 39
4.10 Vlastní interpret ... 40
4.11 Princip činnosti interpretu ... 40
4.12 Problém spojení Soft a Hard referencí ... 41
5 Popis řešení rendez-vous v projektu SAN ... 43

5.1 Současný stav projektu SAN 43
5.2 Požadavky na rendez-vous v projektu SAN 43
5.3 Specifikace rendez-vous 43
5.4 Popis řešení 44
5.5 Vytvoření rendez-vous klienta 45
5.6 Vytvoření rendez-vous serveru 45
5.7 Ukázkové aktivní programy 46
5.8 Ukázkový program rendezvous 47
5.9 Ukázkový program rendezvous2 48
6 Problémy při implementaci synchronizace 50
6.1 Nepodporování polí interpretem 50
6.2 Podpora interpretu pro více tříd 51
6.3 Vyhledávání volaných metod 51
6.4 Převádění referencí u nativních funkcí 51
6.5 Problém převodů hard a soft referencí 52
6.6 Vyhledávání správné metody 54
7 Bezpečnostní problémy 55
8 Závěr 56
8.1 Vlastní přínos 56
Přehled pojmů a zkratek 57
Literatura k tématu 58
Příloha A – uživatelská dokumentace 59

1 Úvod

Při vytváření moderních aplikací často využíváme systému programových vláken, která svou činnost vykonávají paralelně. Činnost těchto vláken je občas potřeba synchronizovat a někdy potřebujeme, aby si vlákna navzájem vyměnila svá data. V programovacím jazyce Ada máme k dispozici synchronizační prostředek rendez-vous, který tuto funkci nabízí. V projektu Smart Active Node (SAN), který je vytvořen v programovacím jazyce Java, bychom chtěli podobnou funkci také využívat. Protože však Java přímo takovou funkcionalitu nemá, zabývá se tato práce jejím vytvořením a implementováním do projektu SAN.

Protože tato práce se zabývá vytvořením nového synchronizačního prostředku v prostředí Javy, bylo nejprve nutné tento prostředek vyvinout a vyzkoušet zvlášť jako samostatný program. Vývoj prostředku rendez-vous a jeho implementace do projektu SAN jsou dva zcela odlišné úkoly a této skutečnosti odpovídá i členění této práce.

V první části práce jsou popsána některá fakta o synchronizaci vláken, představen synchronizační prostředek rendez-vous jazyka Ada a vysvětlen způsob, jakým bylo postupováno při vývoji tohoto synchronizačního prostředku v prostředí Javy. V druhé části této práce je popsán projekt SAN a způsob, jakým byl vytvořený prostředek rendez-vous do tohoto projektu implementován.

2 Seznámení s jazykem Ada

Ada je programovací jazyk, který vznikl na přelomu 70. a 80. let pro potřeby amerického ministerstva obrany. Cílem bylo vyvinout univerzální programovací jazyk s dobrou čitelností rozsáhlých zdrojových kódů. Jazyk měl zvládnout i programy menšího rozsahu, ale primárně měl být používán u tzv. *mission-critical* projektů, např. leteckého software a chybám odolným systémům.

Vzniklý jazyk byl pojmenován po hraběnce Adě Lovelace, která žila v letech 1815-1852 a je považována za historicky první programátorku. Protože tento jazyk není příliš používaný a tato práce se zabývá jednou z jeho hlavních funkcionalit, dovolím si malé představení tohoto jazyka a jeho možností.

Vzhledem k zamýšlenému užívání jazyka byla při vývoji kladena pozornost na dobrou čitelnost rozsáhlejších programů. Jazyk Ada syntakticky připomíná programovací jazyk Pascal. Typický ukázkový program *Hello world* bychom v jazyce Ada zapsali takto:

```
with Ada.Text_IO;
procedure Hello is
begin
     Ada.Text_IO.Put_Line("Hello, world!");
end Hello;
```

2.1 Ada a bezpečnost

Při vývoji jazyka Ada byl jazyka kladen důraz na silnou typovost. Důsledné dodržování typů proměnných má odhalit a zabránit potenciálním chybám už při překladu zdrojového kódu. Jazyk C, jehož syntaxi využívá řada dnes běžně používaných jazyků, také hlídá typy proměnných, ale na rozdíl od Ady umožňuje rychlé přetypování. Rozdíl mezi oběma syntaxemi ukazují následující dva fragmenty zdrojových kódů.

Možnost přetypování v jazyce C:

```
typedef int JABLKA;
typedef int HRUSKY;
int main(int argc, char **argv) {
     JABLKA jablka = 10;
     HRUSKY hrusky = (HRUSKY) jablka;
}
```

V uvedeném fragmentu kódu v jazyce C jsme definovali dva typy proměnných: jablka a hrušky, a provedli jsme navzájem přiřazení. Syntakticky je to věc povolená a programátor očekává, že se konverze provede automaticky a správně.

Jazyk Ada takové přiřazení zakazuje. Je to z důvodů bezpečnosti - zaměnit jablka za hrušky je potenciální problém - a Ada toto chování neumožní již na úrovni překladače. Následující fragment kódu v jazyce Ada ukazuje syntakticky nesprávný program.

```
type Jablka is new Integer;
type Hrusky is new Integer;

declare
     pocet_Hrusek : Hrusky = 10;
     pocet_Jablek : Jablka;
begin
     pocet_jablek := pocet_hrusek;
end;
```

Vidíme zde opět přiřazení hrušky na jablka. Přestože se stejně jako v kódu jazyka C jedná o stejnou reprezentaci celočíselné proměnné a lze tedy předpokládat správnost při běhu takového programu, Ada takovýto program vůbec neumožní přeložit. Pro úplnost se sluší dodat, že Ada umožňuje toto omezení obejít tím, že si přiřazení sami nadefinujeme. Zde se však již pravděpodobně nebude jednat o programátorskou chybu.

Uvedený příklad demonstroval chybný program v jazyce Ada. Takové chování se může zdát na první pohled jako nevýhodné, ale v praxi může vést k odhalení potenciální chyby. Jako příklad bychom mohli uvést fyzikální vztah pro výpočet elektrického odporu. Ten je definován vztahem:

$$R\ [\Omega] = \frac{U\ [V]}{I\ [A]}.$$

V jazyce jakým je např. C bychom použili pro všechny tři veličiny některou z proměnných pro reálné číslo a vydělili hodnoty napětí a proudu. Ovšem nic nám nebrání, abychom vydělili hodnoty naopak. V takovém případě by výsledkem bylo opět reálné číslo, ale ne správná hodnota odporu. Ada nás od takové chyby ochrání právě zavedením nových typů proměnných. Zavedeme si všechny fyzikální jednotky, se kterými v programu chceme pracovat a nadefinujeme si vztahy mezi nimi. V našem případě by to byly veličiny elektrický odpor, napětí a proud. Definováním vztahů mezi typy proměnných pak předejdeme chybě v nějakých složitějších vzorcích. Následující fragment kódu tento program dokumentuje.

```
type Ampery is new Float;
type Volty is new Float;
type Ohmy is new Float;

declare
     proud : Ampery;
     napeti : Volty;
     odpor : Ohmy;
begin
     odpor := napeti / proud;
end;

function "/" (up : Volty, down : Ampery) return Ohmy is
begin
     return Ohmy(Float(up) / Float(down));
end;
```

Další ochranou, kterou nám Ada nabízí, jsou například intervaly. Pokud již dopředu známe rozsah hodnot, se kterými budeme v programu pracovat, můžeme si toto omezení v programu definovat. To demonstruje další fragment Ada kódu, ve kterém potřebujeme ukládat desetinou hodnotu úhlu v rozmezí 0-359°.

```
type Uhel is new Float range 0.0 .. 359.0;
declare
     uhel : Uhel := 361;  -- hodnota je mimo rozsah

begin

end;
```

V uvedeném zdrojovém kódu nám tuto chybu odhalí překladač, který neumožní překlad. Pokud by za běhu programu došlo k takovémuto přiřazení, například uživatelským vstupem, bude vygenerována výjimka. Ačkoliv by se zdálo, že toto omezení si může programátor ohlídat, anebo toto ignorovat, protože nám to v případě funkcí sinus a cosinus nevadí, v jiném případě už nám tato kontrola může ušetřit hodně času při nepříjemném ladění programu.

Příkladem problému z praxe, kde nám intervaly mohou pomoci vyvarovat se nechtěné chyby, může být čítač prvků v seznamu. Zde je naprosto jasné, že počet prvků v seznamu jistě nebude záporný. Pokud bychom se pokusili odebrat prvek z prázdného seznamu a snížit hodnotu deklarované proměnné v následujícím kódu, bude opět vygenerována výjimka. V jiných jazycích musíme provádět kontrolu sami a opomenutí může vést k chybám. V Adě nám stačí pro počet prvků použít definici:

```
type Pocet_Prvku is new Integer range 0 .. 2**31;
```

2.2 Shrnutí programovacího jazyka Ada

Všechny uvedené vlastnosti jazyka Ada z něj dělají velmi bezpečný a použitelný jazyk. Ada prošla vývojem jako každý programovací jazyk. Od verze Ada 95 již obsahuje i podporu OOP. Bohužel tato podpora je od jiných jazyků trošku odlišná.

Přesto se však Ada i přes zjevné výhody do většího zájmu programátorské veřejnosti nedostala. Odlišná podpora OOP může být jednou z mnoha příčin. Další příčinou může být větší náročnost na pochopení konstrukcí jazyka a jeho znalosti i pro triviálnější programy. Jedna z mnoha funkcionalit Ada však zajímavá je a tvoří hlavní část této práce. Touto funkcionalitou je synchronizace vláken typu rendez-vous.

2.3 Vlákna a různé programovací jazyky

S vlákny se setkáváme v moderních aplikacích poměrně často. Jestliže aplikace provádí nějaký déletrvající výpočet, pak po dobu trvání tohoto výpočtu se uživateli aplikace jeví jako zamrznutá. Proto se i v jednoduché aplikaci, která provádí déletrvající operace, setkáváme s vlákny. Budeme-li nazývat běžící program v rámci operačního systému jako proces, pak tento proces může mít více vláken, která svou činnost provádějí paralelně. Jednotlivá vlákna pracují v kontextu procesu, který je vytvořil, a bez něho nemohou existovat.

Ve vícevláknové aplikaci bývá typicky hlavní vlákno vyhrazené pro uživatelské interakce, další vlákna jsou spouštěna podle potřeby na provádění operací. K realizaci vláken je zapotřebí podpory operačního systému. Při vícevláknovém běhu programu často potřebujeme jednotlivá vlákna řídit a synchronizovat přístup ke společně sdíleným prostředkům. Podpora, řešení vláken a jejich synchronizace jsou v každém programovacím jazyku odlišné.

V jazyce C jsou vlákna nazývána *thread* a jsou řešena jako řada funkcí z knihovny *pthread*. Výkonný program vlákna je samostatná funkce. Pro synchronizaci vláken jsou k dispozici funkce ze stejné knihovny. Pro synchronizaci činnosti vláken máme k dispozici zámky, bariéry a jiné. Ne všechny funkce jsou však dostupné ve všech operačních systémech a vzniká tak problém s přenositelností zdrojových kódů.

V Javě je řešení vláken, která jsou také nazývána *thread*, velice odlišné. Pro vytvoření vlákna jsou v Javě 2 možnosti. První možností je vytvoření vlákna jako samostatné třídy, která rozšiřuje standardní vláknovou třídu *Thread*. Druhou možností je vytvoření třídy, která

implementuje rozhraní *Runnable* a po vytvoření instance této třídy ji předáme konstruktoru třídy *Thread*.

V obou případech nám vznikne instance třídy *Thread*. K řízení běhu vláken je nad tímto objektem definována řada metod. Pomocí nich můžeme řídit jak běh vlákna, tak i synchronizaci vláken. Pro synchronizaci přístupu máme k dispozici nad každým objektem monitor. Synchronizační metody vláknového objektu přímo komunikují s Java Virtual Machine a problém s přenositelností zdrojového kódu nám tedy nevzniká.

2.4 Ada a vlákna

Ada řeší vlákna trochu jinak a na vyšší úrovni abstrakce. Již samotné pojmenování pro vlákna je v tomto jazyce jiné a tím je *Task*. Vlákna, čili tasky, jsou v Adě speciální objekty se specifickou konstrukcí. Jednoduchý vláknový *Hello world* program v Adě by mohl vypadat takto.

```
with Ada.Text_IO;
procedure Hello_World is
     task type Hello_Task is
          -- specifikace rozhraní tasku, v tomto případě žádná
     end Hello_Task;
     task body Hello_Task is
     begin
          Ada.Text_IO.Put_Line("Hello world from task");
     end Hello_Task;

task1, task2 : Hello_Task;
begin
     Ada.Text_IO.Put_Line("Hello, world from main");
end Hello_World;
```

Takto zapsaný program po spuštění vytvoří 2 vlákna (tasky) a spustí je paralelně s primárním vláknem. Celkově po spuštění běží v procesu celkem 3 vlákna. Tento program nevykonává žádné operace nad sdílenými daty, proto žádná synchronizace není potřeba.

V paralelních programech se však často setkáváme se situací, že potřebujeme pracovat nad sdílenými daty. Dokud nad daty provádíme pouze čtení, pak vše funguje správně, jako v klasickém, jednovláknovém programu. V případě modifikace dat již musíme provádět synchronizaci. V jazyce C bychom zřejmě použili zámek, v Javě by to byl v případě jednoho objektu zřejmě zámek nebo použití *synchronized* metod. Čím více modifikací nad sdílenými

daty provádíme na více místech v programu, tím roste pravděpodobnost opomenutí a chybné funkčnosti programu. Proto je snaha přístup ke sdíleným datům ošetřovat, pokud možno, na jednom místě.

V Javě k tomu pomohou zmíněné *synchronized* metody, Ada má přímo prostředek pro synchronizaci nad sdílenými daty. Jsou to takzvané chráněné typy. Jejich použití spočívá v myšlence, že operaci čtení nad daty provádí funkce, které vracejí hodnotu. Modifikaci dat naopak provádějí procedury. Synchronizaci nám pak může zařídit už překladač. Použití dokumentuje následující ukázka, která ukazuje práci se sdíleným čítačem. Modifikaci provádí jen procedura *Inkrement*. Kdybychom nějaký příkaz modifikující hodnotu privátní proměnné zapsali do těla funkce, překladač by nám zahlásil chybu.

```
protected type Citac is
     procedure Inkrement;
     function Get_Hodnota return Integer;
private
     Hodnota : Integer := 0;
end Citac;
protected body Citac is
     procedure Inkrement is
     begin
          Hodnota := Hodnota +1;
     end;
     function Get_Hodnota return Integer is
     begin
          return Hodnota;
     end;
end Citac;
declare citac : Citac;
```

Fragment zdrojového kódu tasku, který provádí operace s čítačem by mohl vypadat asi takto:

```
if citac.Get_Hodnota < 100 then
     citac.Inkrementuj;
end if;
```

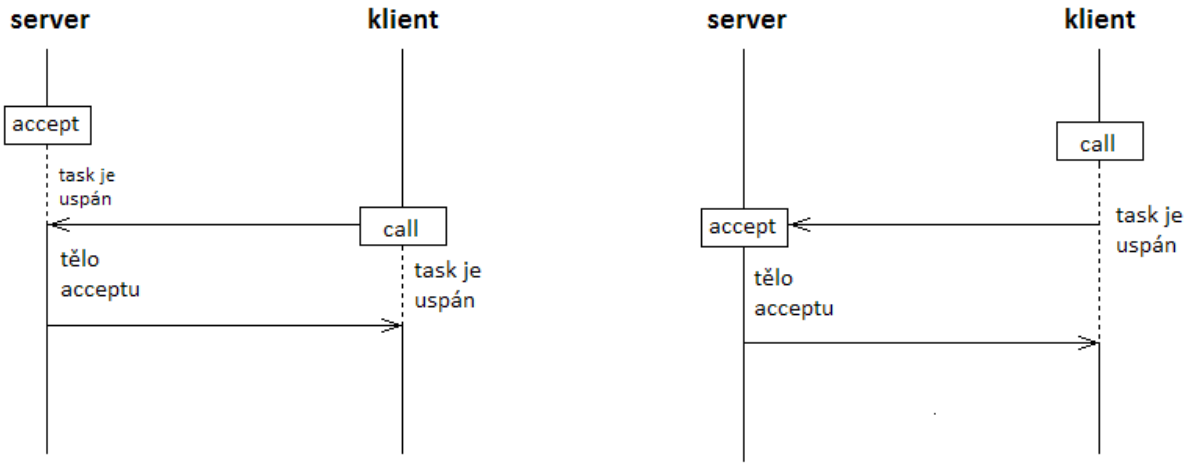
V uvedeném programu tasky pracují se sdílenými daty, avšak nekomunikují mezi sebou navzájem. Pro případ, že spolu jednotlivé tasky potřebují komunikovat, má Ada elegantní jazykovou konstrukci nazývanou rendez-vous. Termín pochází z francouzského *rendez-vous*, které bychom mohli přeložit jako setkání nebo rande, což je velmi výstižné.

Aby se v Adě mohly 2 tasky setkat a komunikovat, musí k tomu být definovaný bod setkání a oba tasky k tomu musí dát souhlas. Princip demonstruje následující ukázka. Program ve výpočetním vlákně vypočítá desetinásobek zadaného čísla.

```
task type Nasobek_Task is
     entry Vstup(X : integer);
     entry Vysledek(X : out Integer);
end Hello_Task;

task body Nasobek_Task is
     Value_Vstup, Value_Vysledek : integer;
begin
     accept Vstup(X : Integer) do
          Value_Vstup := X;
     end Vstup;

     Value_Vysledek := 10*Value_Vstup;

     accept Vystup(X : out Integer) do
          X := Value_Vysledek;
     end Vystup;

end Nasobek_Task;

nasobek : Nasobek_Task;
Vysledek : Integer;
begin
     nasobek.Vstup(10);
     nasobek.Vystup(Vysledek); -- Do výsledku se uloží 100
end;
```

V uvedeném programu jsou specifikována dvě rendez-vous místa, která jsou v Adě označovaná klíčovým slovem *entry*. Aby mohla proběhnout komunikace na určeném *entry,* musí obě strany toto *entry* volat. V těle obsluhujícího tasku je požadavek na komunikaci s druhým taskem přijímán klíčovým slovem *accept*. Ve druhém tasku je přímo volán (*call*) požadovaný entry na objektu tasku, se kterým chceme komunikovat. Volání probíhá pomocí obvyklé tečkové notace na objektu tasku.

Na časovém průběhu rendez-vous nezáleží, task, který první dorazí k určenému *rendez-vous entry* je uspán a čeká na druhý task. Pro jednoduchost budu task, který provádí accept na svém entry, nazývat serverem a task, který provádí call, budu nazývat klientem. Obě varianty časového průběhu rendez-vous zobrazuje obrázek 2.4.



Obr. 2.4 – varianty průběhu rendez-vous

V levé části obrázku je nejdříve proveden serverem accept a jeho task je uspán do doby, než druhý task zavolá dané entry. Poté je server probuzen, provede tělo entry a pokračuje ve své činnosti dál. Současně po dokončení těla entry je probuzen také klient, který také pokračuje dál.

V pravé části je situace opačná. Klient provede volání a je uspán. Jakmile server provede accept pokračuje ihned dál v těle entry a po dokončení těla entry je klient probuzen a oba pokračují ve svých činnostech dál.

2.5 Různé varianty rendez-vous

V předchozím příkladu výpočetní task přímo specifikuje pořadí, ve kterém má být provedeno volání druhým taskem. V prvním volání si server uloží do své globální proměnné zadanou vstupní hodnotu a rendez-vous tím končí. Po skončení provede užitečnou akci, v tomto případě vynásobení vstupní hodnoty. Při druhém volání do výstupní proměnné zapíše výsledek. Primární task mezi vstupem a výstupem může paralelně provést řadu jiných operací, než si vyžádá výsledek. Primární task však musí dodržet pořadí volání, nelze nejdříve zavolat *entry* pro výsledek.

Někdy však můžeme potřebovat, aby se serverový task sám rozhodl podle volání, které *entry* využije a Ada i k tomu má mechanismus. Ten dokumentuje následující program, který implementuje jednoduchý *buffer*.

```
task Buffer is
     entry Vlozit(X : Integer);
     entry Odebrat(X : out Integer);
```

```
end;
task body Buffer is
     maxSize : constant Integer := 100;
     buf : array(1..MaxSize) of Integer;
     bufStart : Integer := 1;
     bufEnd : Integer := 0;
begin
     loop
          select
               accept Vlozit(X : Integer) do
                    bufEnd := bufEnd mod maxSize + 1;
                    buf(bufEnd) := X;
               end;
               or accept Odebrat(X : out Integer) do
                    X := buf(bufEnd);
                    bufStart := bufStart mod maxSize + 1;
               end Odebrat;
          end select;
     end loop;
end Buffer;
```

Buffer v předchozím případě je implementován jako pole s přesně definovanou délkou. Druhý task, který by chtěl volat oba *entry* na vkládání a výběr prvku, může tyto *entry* volat v náhodném pořadí.

Uvedený program má však zásadní chybu, protože neošetřuje počet prvků v poli. Nejsou tak ošetřeny případy, když buffer přeteče, nebo naopak, pokud se nepokoušíme vybírat prvek z prázdného bufferu. Tato situace by se dala vyřešit přidáním proměnné pro počítadlo prvků s nadefinovaným intervalem. Pokud by nastal jeden z těchto případů, byla by vygenerována výjimka.

Jiné řešení spočívá v úpravě těla tasku tak, aby v případě pokusu o výběr z prádného bufferu byl vybírající task uspán až do doby, než nějaký jiný task provede vložení. Stejná situace by měla být i v případě naplnění bufferu. Proto by bylo vhodné doplnit accept o nějakou podmínku. Ada nám vytvoření takto podmíněného rendez-vous umožňuje. Navíc nám Ada ještě umožňuje čekání při acceptu omezit časově. Následující program modifikuje předchozí o ošetření situací při prázdném a plném bufferu. Navíc přidává časové omezení pro příchozí call a po stanovené době vyprázdní buffer. V programu je tato doba nastavena na 1 minutu.

```
task Buffer is
     entry Vlozit(X : Integer);
```

```
     entry Odebrat(X : out Integer);
end;
task body Buffer is
     maxSize : constant Integer := 100;
     maxTime : constant Integer := 60;
     buf : array(1..MaxSize) of Integer;
     bufStart : Integer := 1;
     bufEnd : Integer := 0;
     bufSize : Integer := 0;
begin
     loop
          select
               when bufSize<maxSize =>
                    accept Vlozit(X : Integer) do
                         bufEnd := bufEnd mod maxSize + 1;
                         buf(bufEnd) := X;
                         bufSize := bufSize + 1;
                    end Vlozit;
               or when bufSize>0 =>
                    accept Odebrat(X : out Integer) do
                    X := buf(bufEnd);
                    bufStart := bufStart mod maxSize + 1;
                    bufSize := bufSize - 1;
                    end Odebrat;
               or
                    delay(maxTime);
                    bufSize := 0;
                    bufStart := 1;
                    bufEnd := 0;
          end select;
     end loop;
end Buffer;
```

Na uvedených příkladech bylo demonstrováno použití synchronizačního prostředku rendez-vous při komunikaci 2 tasků v Adě. Rendez-vous nemusí být použito jen při komunikaci, stačí i pro jednoduchou synchronizaci tasků. Tělo acceptu může být prázdné a neprovádět žádnou užitečnou činnost.

Rendez-vous není jediný synchronizační prostředek, který nám jazyk Ada nabízí. Pozorný čtenář si jistě povšimnul, že v obou posledních příkladech bylo použito nekonečné smyčky. V primárním vlákně však není žádný příkaz, který činnost této smyčky ukončí. Činnost jiného tasku bychom mohli ukončit příkazem *terminate*, avšak není to nutné. Ada se nám automaticky postará o ukončení běžících tasků v okamžiku, kdy končí činnost tasku, který jej

vytvořil. To je rozdíl oproti Javě, kde bychom měli počkat na ukončení činnosti vláken metodou *join* nebo se jinak postarat o jejich ukončení. Ada tak eliminuje další potenciální chybu programátora.

2.6 Shrnutí rendez-vous v jazyce Adě

Na uvedených příkladech byly uvedeny možnosti synchronizačního prostředku rendez-vous. Jeho hlavní použití spočívá v předávání dat mezi dvěma tasky. Na rendez-vous se můžeme dívat jako na spuštění metody v kontextu jiného vlákna. Při použití si task akceptující rendez-vous může specifikovat pořadí pro jednotlivá volání, nebo příchozí volání akceptovat podle pořadí, ve kterém přichází. Task si také může specifikovat podmínky, za kterých accept provede a čekání na accept lze omezit také časově. To vše dohromady dělá z Adovského rendez-vous velmi silný nástroj, který v jiných jazycích schází. Otázkou této práce je, zda by se systém rendez-vous nechal vytvořit také v jiném programovacím jazyce, konkrétně v jazyce Java.

3 Rendez-vous v jazyce Java

Než se pustíme do vytváření nového prostředku, je důležité si nadefinovat, čeho se snažíme dosáhnout. Jedná se o vytvoření prostředku v Javě, který programátorovi umožní použít synchronizační prostředek ve více vláknech, který se co nejvíce podobá Adovskému rendez-vous a možnostem jeho užití, které byly zmíněny v předchozí kapitole.

Možnosti vytvoření tohoto prostředku jsou dvě. První možností by bylo navrhnout nějaký syntax pro rendez-vous a vytvořit nový překladač zdrojového kódu, který by převedl zdrojový kód do existujícího bytecodu. Výhodou tohoto řešení je, že bychom získali přesné možnosti, které máme při tvorbě rendez-vous v Adě. Nevýhodou tohoto řešení by byla nemožnost využít standardního překladače a tím i stávající vývojová prostředí, která nám při vývoji programů usnadňují práci.

Druhou možností je synchronizační prostředek vytvořit pouze za použití běžných prostředků v Javě. Pro jednotlivá *entry* jsou v Javě nejbližším ekvivalentem metody nějaké třídy. Cílem v tomto případě je vytvoření nějaké třídy, která nám bude poskytovat metody, které budou onu synchronizaci vytvářet a řídit ji přímo za běhu programu. Takový objekt musí umožňovat využití při několika základních podmínkách. První z nich je, že předem nevíme, kolik *entry* bude mít jaké vlákno definováno. Musí nám tedy umožnit si zaregistrovat předem neznámé

množství různě pojmenovaných metod. Každá taková metoda může mít předem neznámý počet libovolných parametrů a libovolný návratový typ. Dále by nám řídící objekt měl poskytnout metody pro accept volání na *entry*, které má vlákno zaregistrované a metodu pro volání *entry* druhým vláknem. Další metodou by mělo být vrácení seznamu zaregistrovaných *entry* pro konkrétní vlákno.

3.1 Jak vytvořit volání předem neznámých metod

K vytvoření volání pojmenovaných metod máme 2 možnosti řešení. Prvním řešením je využití mechanismu reflexe. Ta nám umožňuje zjistit za běhu programu jména metod a typy jejich parametrů a umožňuje nám dynamické volání. Pak by nám stačilo mít v řídícím objektu metody: zaregistruj a volej. První metodě bychom předali vláknový objekt a metoda by si z něj sama zjistila jména dostupných metod a parametrů. Možností by také bylo jména přímo specifikovat nebo naopak zakázat některá jména metod, na kterých nechceme rendez-vous realizovat. Při volání by pak stačilo zavolat metodu a v parametrech předat název metody a její parametry. Výhodou tohoto řešení je, že máme ve zdrojovém kódu programu vytvořené skutečné metody se skutečnými parametry a vláknový objekt tak bude klasická javovská třída. Nevýhodou je poměrně složité volání uvnitř řídící třídy a menší rychlost při volání metody.

Druhou variantou je použití systému tzv. *callback* metod. Callback metody jsou metody, které si nějaký objekt zaregistruje pro obsluhu určité události. V Javě takovou metodu můžeme vytvořit pomocí prostředků OOP, konkrétně pomocí rozhraní (*interface*). Potřebujeme vytvořit rozhraní s jednou metodou. Zaregistrování metody je pak realizováno uložením reference na instanci třídy, která implementuje dané rozhraní. Rendez-vous *entry* pak můžeme vytvořit jako několik vnitřních tříd implementujících dané rozhraní. Princip demonstruje ukázka programu.

Ukázka rozhraní pro callback metody:

```
public interface IEntry {
     public int entry(int param);
}

class ServeroveVlakno extends Thread {
     private int hodnota;
     private class VstupEntry implements IEntry {
          public int entry(int param) {
               hodnota = param;
               return 0;
          }
     }
     private class VystupEntry implements IEntry {
          public int entry(int param) {
               return hodnota;
          }
     }
     public ServeroveVlakno() {
          rendezvous.registerEntry(this, „vstup“, new
VstupEntry());
          rendezvous.registerEntry(this, „vystup“, new
VystupEntry());
     }
     public void Vstup(int hodnota) {
          rendezvous.call(this, „vstup“, hodnota);
     }
     public int Vystup() {
          return rendezvous.call(this, „vystup“, 0);
     }
}
```

Uvedený příklad ukazuje možnost využití callback metod, konkrétně 2 metod pro vstup a výstup. Klientské vlákno volá *public* metody Vstup a Vystup ze serverové třídy. Tyto metody provedou zavolání *entry* pomocí řídícího rendez-vous objektu a zaregistrovaných jmen *entry*. Řídící objekt má uložené reference na objekty s implementovaným rozhraním *IEntry*. Samotnou užitečnou činnost *entry* metod pak provádějí vnitřní třídy. Tyto vnitřní třídy mohou pracovat s vnitřními proměnnými vláknové třídy a řídící objekt nám má zařídit, že budou spuštěné v kontextu tohoto vlákna.

Výhodou tohoto řešení je využití standardních možností Javy a rychlejší spouštění metod. Nevýhodou je složitější tvorba vláknové třídy. Další nevýhodou je, že uvedené rozhraní

umožňuje pouze spuštění metody, která má vstup int a výstup int. V praxi bychom však potřebovali spouštět metody s jinými parametry, které dopředu neznáme. I přesto jsem se rozhodl využít tohoto způsobu, protože přináší větší možnosti ovlivnit, které metody skutečně mají sloužit jako rendez-vous entry.

3.2 Jak předat metodě libovolný počet parametrů

Ada obsahuje mnoho ochranných mechanismů, které ve snaze ochránit výsledný program před případnými programátorovými chybami, neumožňují užití řady prostředků, které jsou v Javě běžně dostupné. Programátor využívající nově vytvářený prostředek se však pro Javu z nějakého důvodu rozhodl. Ať už ho k výběru jazyka Java vedlo cokoliv, vzal na vědomí možnosti Javy a její případné výhody a nevýhody. Proto i při tvorbě nového prostředku lze plně vlastnosti Javy využívat. Jedním z takových základních prostředků je možnost volného přetypování.

V Javě můžeme využít té skutečnosti, že se jedná o jazyk založený na objektově orientovaném programování. K dispozici zde máme jen 9 základních datových typů. Jedná se o typy *char,byte, int, short, long, float, double, boolean* a *void*. Všechno ostatní jsou tzv. referenční proměnné, které odkazují přímo do paměti. To mohou být objekty nebo pole. Důležité je, že pro každý datový typ máme k dispozici obalovací třídu a tím můžeme každý datový typ vyjádřit jako objekt.

V Javě je každý objekt oděděn od třídy Object. Pokud tedy předáme cokoliv jako Object, můžeme zpětně tento Object přetypovat na původní objekt nebo na základní hodnotu. Jediné, co musíme znát, jsou původně předávané objekty. Tímto systémem můžeme předat do jakékoliv metody jakékoliv parametry, a to i když předem neznáme jejich počet a typ. Jakékoliv objekty můžeme přetypovat na pole Object a to předat jako Object. Jak předat několik hodnot do metody ukazuje následující fragment Java kódu, ve které předáváme do metody *entry* hodnoty z metody *main*.

```
static void entry(Object[] params) {
     short var_short = ((Short)params[0]).shortValue();
     int var_int = ((Integer)params[1]).intValue();
     char var_char = ((Character)params[2]).charValue();
     // zde bychom mohli provést nějaký výpis nebo další práci s parametry
}
```

```
public static void main(String[] args) {
     short var_short = 1;
     int var_int = 10;
     char var_char = 'x';
     entry(new Object[] {
                    new Short(var_short),
                    new Integer(var_int),
                    new Character(var_char)} );
}
```

V metodě *main* jsme použili obalovací třídy pro primitivní datové typy a jako pole objektů jsme předali do metody *entry*. Mírné zjednodušení bychom si mohli dovolit při volání metody a využít toho, že nám Java přetypování na obalovací třídy provede automaticky. Tímto principem jsme schopni předat jakékoliv parametry při volání vytvářeného rendez-vous. Správné programátorské řešení by bylo ještě otestování předaných hodnot, aby nedošlo k chybě při nějaké úpravě volající metody, ale na principu to nic nemění.

3.3 Prostředky pro synchronizaci v Javě

Jestliže dokážeme předat parametry předem neznámé metodě, zbývá nám ještě vyřešit samotný způsob synchronizace. Java nám nabízí několik prostředků pro tyto účely. Jak již bylo řečeno, každý objekt je v Javě oděděn od základní třídy Object z balíku java.lang. Základní třída Object má v sobě implementovaný synchronizační prostředek monitor a ten tak máme k dispozici na každém objektu. Pro ovládání synchronizace na monitoru jsou k dispozici 3 důležité metody: *wait, notify a notifyAll*.

Používání těchto metod je možné ve více vláknech a slouží k zajištění exkluzivního přístupu jednoho vlákna ke sdílenému objektu. Chce-li vlákno provádět modifikace, vstoupí do tzv. kritické sekce programu. Touto sekcí může být metoda, která je označená klíčovým slovem *synchronized* nebo blok v programu, který je tímto slovem označen a je určeno, na jakém objektu se chceme synchronizovat. Ve skutečnosti jsou oba způsoby pro vytvoření kritické sekce identické, *synchronized* metoda je převedena na *synchronized* blok a synchronizuje se na proměnné *this*.

Jakmile při běhu vlákna dojdeme k začátku kritické sekce, automaticky se zkontroluje, zda je monitor volný a jestliže ano, vlákno pokračuje bez přerušení dál a monitor se nastaví na obsazeno. Všechna další vlákna jsou po dosažení začátku kritické sekce na stejném objektu uspána, dokud první vlákno kritickou sekci neopustí.

Chování při synchronizaci ovlivňují již zmíněné 3 metody. Použijeme-li metodu *wait* uvnitř kritické sekce na synchronizovaném objektu, je vlákno, které metodu použilo, uspáno uvnitř této sekce a monitor je uvolněn pro jiná vlákna. Neurčíme-li maximální dobu uspání metodou *wait*, je vlákno uspáno až do doby, než jiné vlákno zavolá na témže objektu metodu *notify* nebo *notifyAll*. První z metod probudí jedno z případných uspaných vláken metodou *wait,* druhá probudí všechna takto uspaná vlákna. Probouzené vlákno by mělo být označeno jako běhu schopné (*runnable*), a jakmile je monitor uvolněn, mělo by ho dostat znovu k dispozici pro sebe a pokračovat v činnosti. Přesné chování však záleží na konkrétním JVM. Proto, pokud jsme vlákno uspali při vyhodnocení nějaké podmínky, měli bychom po probuzení zkontrolovat, zda podmínka pro uspání je stále ještě platná a případně vlákno znovu uspat. Z toho důvodu je vhodné při uspání používat zároveň i cyklus testující danou podmínku.

Můžeme použít tyto metody pro vytvoření rendez-vous? Kdybychom volající vlákno zastavili metodou *wait* a po dokončení acceptu serverovým vláknem ho vzbudili metodou *notify* ze serverového vlákna, pak bychom měli částečně realizovaný princip rendez-vous. Zdrojové kódy pro volání rendez-vous obou stran by mohly vypadat zhruba takto.

klientské vlákno:

```
entryCallObject.call(params);  // voláme definované entry
entryCallObject.wait(); // čekáme až server entry dokončí
```

serverové vlákno:

```
params = entryCallObject.accept(); // počkáme na příchozí call
// provedeme nějakou činnost acceptu
entryCallObject.notify();
```

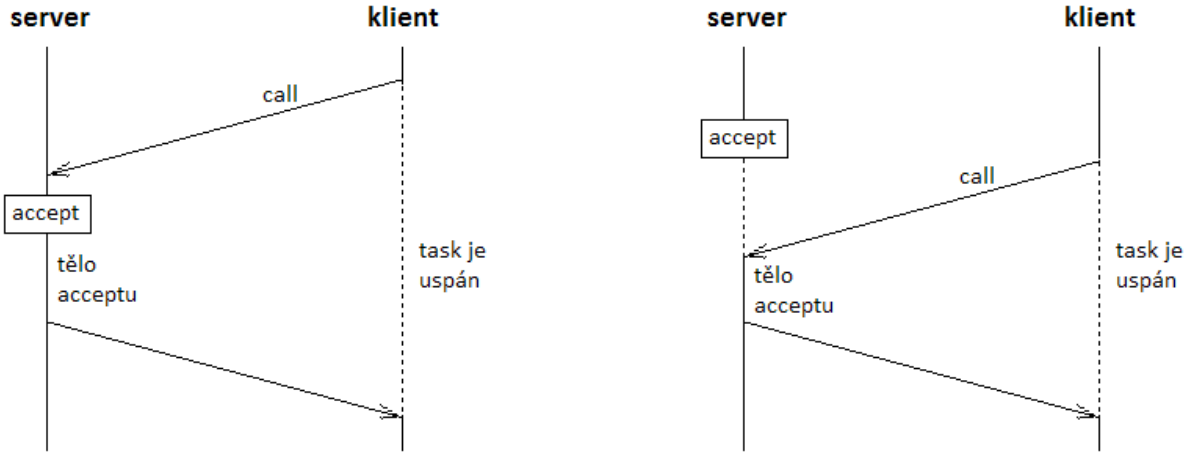
Bude tento princip fungovat? Na první pohled by se mohlo zdát, že ano. Klientské vlákno je po dobu acceptu uspáno a vzbuzeno pomocí *notify*. Aby bylo možno tuto otázku zodpovědět, je nutné se podívat trochu blíže na činnost vláken a jejich plánování na jednoprocesorovém stroji.

3.4 Jak fungují vlákna

Máme-li jednoprocesorový stroj, můžeme v jednu chvíli provádět pouze jedinou operaci. V jednu chvíli tedy může vykonávat činnost pouze jedno vlákno. Aby bylo možné na takovém stroji provádět z pohledu uživatele více operací, je nutné procesorový čas rozdělovat. K tomu nám slouží plánovač procesů, který je většinou vestavěn do operačního systému. Plánovač

procesů si udržuje frontu vláken, která jsou ve stavu, že mohou vykonávat činnost (*runnable*). Činnost plánovače spočívá v tom, že si procesorový čas rozdělí na více krátkých intervalů (epochy) a během jedné epochy přidělí procesor jednomu vláknu, které v danou chvíli běží. Po vypršení přidělené doby plánovač přeruší běh vlákna a procesor přidělí jinému naplánovanému vláknu. Jestliže je vlákno uspané (např. metodou *wait*), pak mu nejsou přidělovány plánované intervaly procesorového času. Při přepínání procesů se dbá pouze na to, aby nedošlo k přerušení během tzv. nedělitelných (atomických) instrukcí. Jinak může být běh vlákna přerušen kdykoliv.

Pokud se vrátíme k uvedenému způsobu synchronizace, pak si můžeme představit následující scénář. Klientské vlákno zavolalo *entry* a pak mu vypršel jeho strojový čas. Jeho činnost byla plánovačem přerušena ještě před voláním metody *wait*, která měla vlákno uspat. Poté plánovač přidělí procesor serverovému vláknu, které provede činnost acceptu a zavolá metodu *notify*. Tato metoda se pokusí probudit jedno z čekajících vláken, ale v danou chvíli žádné takové vlákno není. Po určité době je činnost serverového vlákna přerušena a k činnosti se postupně dostane opět klientské vlákno. To se ve svém programu dostane k zavolání metody *wait* a nastává tzv. deadlock. Vlákno čeká na *notify*, ale ten už přijít nemusí a vlákno tak zůstává trvale uspané. Uvedený způsob je tedy pro dané potřeby nepoužitelný. Je potřeba najít jiný způsob synchronizace obou vláken. Řešení by mohl zobrazovat upravený časový diagram rendez-vous, který je na obrázku 3.4.



Obr. 3.4. – diagram časového průběhu rendez-vous

Na obrázku vidíme schéma, které se nápadně podobá systému vzdáleného volání procedur. V Javě máme tento systém také, nalezneme ho zde jako RMI (*remote method invocation*). Dalo by se využít jako synchronizační prostředek? Vyloučené to není, ale vzhledem

k nutnosti komunikace přes externí program by byla režie takové komunikace příliš náročná a pro reálné použití nevhodná. Potřebovali bychom prostředek, který přímo umožní komunikaci mezi vlákny a data si zachová. Tedy takový, který nám uchová stav, že bylo signalizováno a v případě, že nebylo, pak vlákno uspí.

Takovým prostředkem by mohly být např. *Pipe*. To je vlastně proud dat (stream), do kterého jedno vlákno writerem zapisuje a druhé vlákno ze streamu čte. Protože čtení je blokující událost, mohli bychom tento systém využít. Pokud ve streamu data jsou k dispozici, pak voláním metody *read* jsou okamžitě přečtena a ze streamu odebrána. Jestliže data ve streamu nejsou, pak je vlákno uspáno až do doby, než jsou data druhým vláknem do streamu vložena. Toto chování můžeme použít pro rozlišení stavu, zda už signalizováno bylo nebo ne. Tento způsob má však nevýhodu, že musíme nejdřív obě strany *pipe* spojit, než se pokusíme komunikovat a obě strany mohou komunikovat právě s jedním protějškem. Při rendez-vous však může být serverové vlákno voláno z více vláken a s tím by mohly být problémy. Nešlo by tedy realizovat blokující čtení jiným prostředkem? Čtení bychom mohli nahradit blokujícím čtením z fronty. Jestliže vytvoříme synchronizované *FIFO* s blokujícím čtením, mohli bychom jím nahradit *Pipe*.

3.5 Synchronizované FIFO

Až doposud byla snaha vytvořit prostředek, který v jednom kroku zrealizuje požadovanou synchronizaci. Toto chování však požadujeme, aby fungovalo navenek vyvíjeného řídícího prostředku. Uvnitř bychom tento proces mohli rozdělit na několik částí. Podíváme-li se znovu na průběh serverového vlákna na obr. 3.4, pak vidíme, že při zavolání metody accept vlastně vybíráme jeden z příchozích call. Máme zde tedy blokující čtení fronty call, kterou bychom mohli uspořádat jako FIFO. Fronta FIFO je abstraktní datový typ, do kterého vkládáme prvky na konec seznamu a vybíráme je za začátku seznamu. Jako prostředek pro realizaci bychom mohli použít např. třídu LinkedList z balíku java.util.

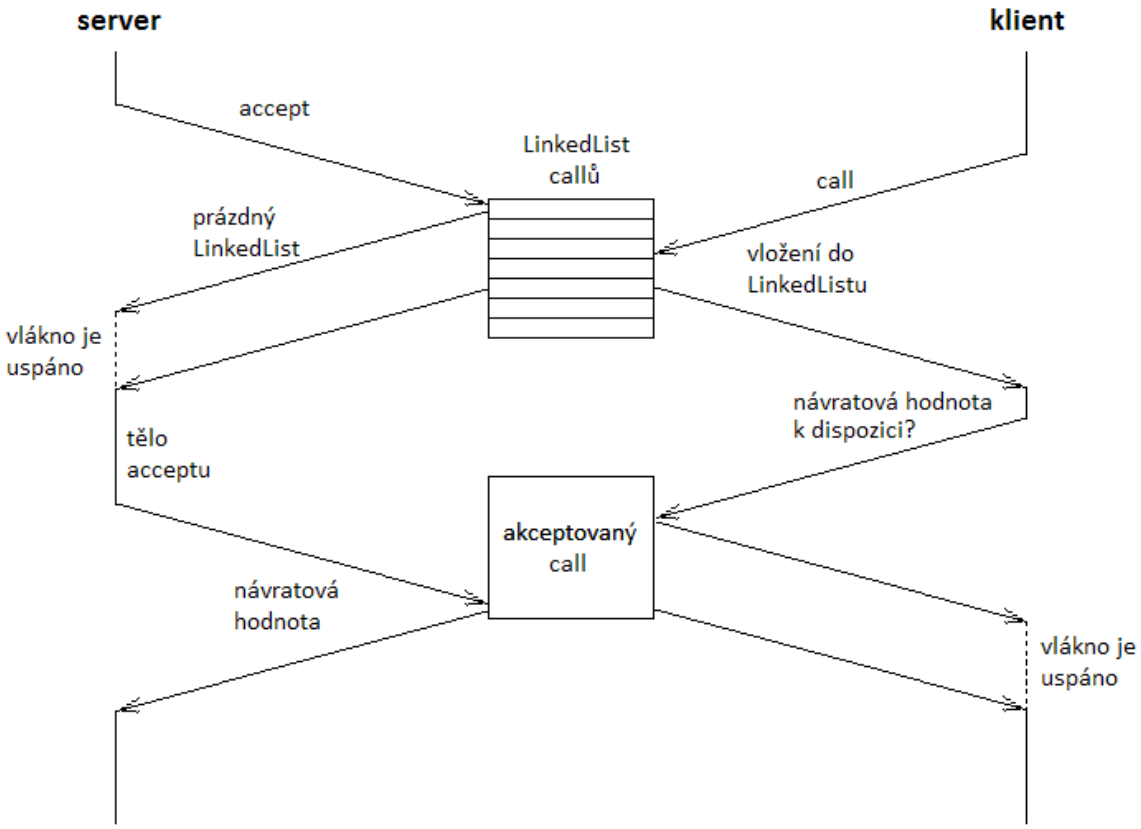
Třída LinkedList se oproti používanějšímu ArrayListu liší tím, že uchovává seznam prvků přesně v pořadí, v jakém do něj byly vkládány. Protože od Javy verze 1.5 máme navíc k dispozici typovost kolekcí, můžeme si vytvořit přímo seznam příchozích call se všemi parametry a referencemi, které si potřebujeme předat mezi vlákny. Postačí nám pouze, když si vytvoříme speciální třídu, která tyto hodnoty bude uchovávat. Její instance budu v dalším textu nazývat call objektem.

Velmi důležitou vlastností třídy LinkedList je, že přímo její činnost není synchronizovaná. Pokud s ní tedy pracujeme ve více vláknech, musíme si synchronizaci zajistit sami. Zde se nám velmi hodí, že LinkedList, stejně jako každý jiný objekt, má svůj vlastní monitor. Tím můžeme zajistit exkluzivitu přístupu při čtení i zápisu do seznamu. Při vstupu do kritické sekce musíme před samotným pokusem o čtení zkontrolovat, zda seznam není prázdný. Pokud se tak stane, pak v časovém průběhu rendez-vous proběhl accept před callem a serverové vlákno se musí uspat na objektu LinkedListu. V tomto případě nám přijde vhod zmíněná vlastnost metody *wait*, a sice automatické uvolnění monitoru pro další vlákna. Jakmile klientské vlákno provede vložení, může zavolat *notify* a máme jistotu, že dojde ke správnému vzbuzení serverového vlákna bez rizika deadlocku.

Nyní jsme ve stavu, že klientské vlákno provedlo zavolání call a stále je ve stavu runnable a zároveň jsme zajistili správné probuzení serverového vlákna k provedení těla acceptu. Po dobu provádění acceptu však potřebujeme mít klientské vlákno uspané a teprve po dokončení acceptu ho probudit. Situace je však odlišná od té předchozí. Zatímco serverové vlákno při acceptu čeká na příchozí call, které mohou přijít současně od více vláken, tak klientské vlákno čeká na odpověď pouze na svůj call. K synchronizaci tedy nelze použít sdílený seznam, ale je potřeba využít jiný objekt. Takovým objektem může být přímo call objekt, který jsme zapsali do LinkedListu callů a jehož referenci v tuto chvíli mají obě strany. Klasické *wait* a *notify* opět použít nemůžeme, protože by mohlo dojít k deadlocku, pokud by *notify* přišlo dřív, než *wait*.

Můžeme však na call objektu vytvořit synchronizované metody pro zápis a čtení návratové hodnoty acceptu. V těchto metodách, kromě uložení a čtení hodnot použijeme ještě podobného principu signalizace, jako v předešlé situaci. V předešlé situaci jsme vyhodnocovali, zda seznam je nebo není prázdný a získávali jsme hodnotu true / false. Nyní potřebujeme otestovat, zda již návratová data jsou k dispozici. Nestačí nám však pouze otestovat, zda data nejsou null, protože i null by mohla být návratová hodnota. Potřebujeme tedy ještě jednu proměnnou typu boolean, nejdříve nastavenou na false, kterou zápisová metoda nastaví na true. V metodě pro čtení návratového typu nejdříve otestujeme, zda už návratová data máme k dispozici a v případě, že nemáme, uspíme klientské vlákno. Serverové vlákno v zápisové metodě uloží data, nastaví příznak, že data jsou k dispozici a probudí klientské vlákno.

Celý princip synchronizace rendez-vous tedy spočívá v tom, že využíváme 2 různé monitory, které nám hlídají příznaky, které vždy může v exkluzivním přístupu měnit pouze jedna strana a pouze druhá strana číst. Tím se nám nemůže stát, že *notify* bude zavoláno dřív a dojde k deadlocku. Průběh vytvořeného rendez-vous ilustruje obrázek 3.5 a zápis synchronizace v Javě zobrazuje následující fragment zdrojového kódu pro obě vlákna.



Obr. 3.5 – princip rendez-vous pomocí synchronizované FIFO

```
LinkedList<Call> calls;

public void accept() {
     synchronized (calls) {
          while (calls.isEmpty())
               calls.wait();
          Call call = calls.pollFirst();  // objekt
                         identifikující volané entry
     }
     call.setResult(…);  // synchronizovaná metoda pro zápis
                         nějaké návratové hodnoty
}
```

```
public void call(String name, Object params) {
     Call call = new Call(name, params);
     synchronized (calls) {
          calls.add(call);
          calls.notify();
     }
     call.getResult();  // synchronizovaná metoda pro čtení
                         návratové hodnoty
}
```

3.6 Jak identifikovat jednotlivá entry

Rendez-vous *entry* můžeme v rámci programu považovat za určité metody vláknového objektu, a proto je nutné je jednoznačně identifikovat. Uvedený princip pro vytvoření rendez-vous a předchozí fragment zdrojového kódu nám umožňoval realizaci pro jedno serverové a více klientských vláken. V praxi se však v programu zřejmě bude vyskytovat více různých serverových vláken, která mohou mít stejně pojmenovaná entry a ta je nutno jednoznačně určit. Navíc je zřejmé, že musíme rozlišit příchozí volání, která jsme uložili v LinkedListu podle serverových vláken, kterým patří. Pro každé serverové vlákno musíme mít svou vlastní frontu a i tyto fronty musíme jednoznačně určit. Jestliže bychom mohli každé vlákno jednoznačně určit unikátním klíčem, pak můžeme využít možností typované třídy *HashMap.*

Vláknová třída je jako každá třída odvozená od základní třídy Object. Díky tomu máme k dispozici metodu *hashCode*. Tato metoda nám pro každý objekt vrací hodnotu int, která by měla být pro shodný objekt vždy stejná. Bohužel nám však nikdo nezaručí, že bude v celém programu unikátní. Příkladem by mohlo být chování třídy String, která pro stejný řetězec vrací stejnou *hash* hodnotu, ačkoliv se jedná o 2 různé objekty. Pokud metodu *hashCode* chceme využívat, je vhodné si její implementaci zajistit sami. To pro obecné použití není příliš vhodné. Od Javy 1.5 však pro objekty třídy Thread máme k dispozici metodu *getId,* která vrací hodnotu typu long a tato hodnota je v celém programu unikátní a lze ji tedy využít jako klíč. Jiným řešení by bylo vytvořit metodu, která umožní zaregistrovat serverové vlákno, a která nám pro každé takové vlákno vytvoří unikátní klíč.

Nyní už můžeme v celém programu jednoznačně identifikovat každé rendez-vous entry. V běžném programu určujeme metody pomocí tečkové notace, kde první část tvoří název referenční proměnné pro objekt a druhou část název metody. Stejný způsob adresování lze použít i pro *entry*, kdy první část bude určovat ID vlákna a druhá část bude název *entry*, který je pro vlákno unikátní.

Vnitřní systém pro uchování potřebných tříd se trochu změní. LinkedListy pro jednotlivá vlákna nyní můžeme uložit do *HashMapy*, kde klíčem bude ID vlákna. Stejný způsob použijeme i pro *HashSety*, ve kterých uložíme jednotlivá *entry* daného vlákna.

3.7 Selektivní akcepty

Současný systém umožňuje provést rendez-vous na všech *entry* najednou. To však v praxi zřejmě nebude příliš používané, častěji se setkáme se situací, že chceme akceptovat pouze jedno konkrétní *entry* nebo omezenou skupinu. Abychom mohli akceptovat pouze vybrané *entry*, je nutné změnit strategii výběru callu objektu z fronty. Klientské vlákno stále zapisuje call do fronty a vybraný call identifikuje názvem. Serverové vlákno však musí podle pořadí příchozích callů projít celou frontu až do doby, než najde příchozí call objekt, který chce akceptovat. Jestliže takový call objekt ve frontě nenajde, tak se serverové vlákno chová, jako kdyby žádný call neproběhl a je uspáno až do příchodu dalšího libovolného callu objektu.

Variant strategií pro prohledání fronty by však mohlo být více. Vhodné by mohlo být například rozdělení fronty na dvě, kdy ve druhé frontě by mohly být call objekty, které nebyly akceptovány, protože nepatří mezi skupinu *entry*, které akceptovat chceme. Tyto call objekty by ale po vykonání jiného akceptu mohly přijít na řadu a to opět v pořadí, v jakém k volání došlo. Výhodou tohoto řešení by bylo, že call objekt, který přišel naposled, by byl k dispozici ve frontě dřív než zamítnuté cally. Nevýhodou je přesouvání call objektů mezi frontami a po akceptu některého callu by se neakceptované call objekty přesouvaly zpět. Vzniká tak otázka, jestli takové přesouvání call objektů mezi dvěma frontami by bylo rychlejší, než jednoduchý průchod jedinou frontou. Zřejmě by záleželo na konkrétní situaci a procentuálnímu poměru akceptovaných callů. Pokud bychom většinu callů akceptovali okamžitě, pak by toto přesouvání příliš výhodné nebylo. Při mírné modifikaci můžeme využít velmi výhodné vlastnosti LinkedListu, který nám umožní vytvořit Iterator od námi zadané pozice v seznamu. Stačí si tedy zapamatovat, který call jsme prověřovali naposledy a od něj můžeme začít znovu procházet seznam. Výhodou je spojení obou postupů, při jediném akceptu prověřujeme každý příchozí call maximálně jednou a zároveň máme pouze jednu frontu.

3.8 Podmíněné akcepty

Jak již bylo zmíněno, Ada umožňuje provést akcept určitého callu pouze tehdy, pokud je splněná nějaká podmínka. Příkladem může být buffer a ohlídání, zda nevybíráme z prázdného nebo nevkládáme do plného. Vytvoření takovýchto akceptů v Javě můžeme docílit dvěma

strategiemi. Můžeme vyhodnocování podmínky spojit přímo s konkrétním *entry* anebo podmínku můžeme spojit s konkrétním akceptem callu. Obě strategie mají svá pro a proti a je nutné zvážit, která strategie je výhodnější pro žádané potřeby.

V Adě máme k dispozici možnost vytvořit podmíněný akcept tak, že podmínka je svázaná s konkrétním akceptem. Podmínka je zapsaná přímo v těle tasku, odkud je prováděn accept. Výhodou tohoto řešení je, že jestliže používáme jedno entry na více místech tasku s různými podmínkami, pak je specifikujeme přímo při konkrétním použití. V našem řešení se však akcept provádí voláním jediné metody, ve které předáváme seznam jmen entry, které chceme akceptovat. Při tomto volání metody accept bychom kromě tohoto seznamu museli ještě specifikovat definici podmínky pro každé přijímané entry. Řešením podmínky by mohl být podobný způsob, jaký využíváme pro vytvoření těla acceptu a to je systém callback. Tedy vytvoření rozhraní s jedinou metodou, která vrátí boolean hodnotu s vyhodnocením podmínky. Svázání konkrétního entry s podmínkou bychom mohli provést vytvořením typované HashMapy, kde klíčem by byl název entry a hodnotou by byla instance daného rozhraní. Akcept by pak byl prováděn podobně, jako ukazuje následující fragment kódu. Předpokládejme, že máme 3 *entry* nazvané entry1, entry2, entry3 a v určitém okamžiku chceme akceptovat za nějaké podmínky entry1 a vždy přijímat entry3.

```
rendezvous.registerEntry("entry1", Entry1());
rendezvous.registerEntry("entry2", Entry2());
rendezvous.registerEntry("entry3", Entry3());
     …
HashMap<String, IAcceptCondition> acceptedEntry =
     new HashMap<String, IAcceptCondition>();
acceptEntry.put(„entry1", new IAcceptCondition() {
     public boolean isAcceptable() { return podminka; }});
acceptedEntry.put(„entry3", new IAcceptCondition() {
     public boolean isAcceptable() { return true; } } );


rendezvous.accept(acceptedEntry);
```

Tento způsob nám dává možnost využít různé podmínky v různých částech vlákna. V jiné části bychom mohli hashmapu naplnit jinou konfigurací a použít stejné entry. Nevýhodou je, že bychom si museli hashmapu nějak předávat při použití ve více metodách či vláknech nebo si ji pokaždé vytvářet. Druhou možností je svázat podmínku přímo s konkrétním *entry*.

Vzhledem k tomu, že již tělo acceptu máme ve vlastní vnitřní třídě, na které máme implementovanou jedinou metodu *entry*, mohli bychom přidat ještě jednu metodu, která opět bude vracet hodnotu boolean podle vyhodnocení podmínky. V tomto případě by bylo lepší rozhraní IEntry upravit na třídu s abstraktní metodou *entry*. Metodu pro vyhodnocení metody bychom mohli implicitně implementovat, aby vracela true a v případě potřeby podmíněného acceptu, si tuto metodu překrýt svou vlastní. Metodu entry by programátor opět implementoval. Výhodou by bylo, že bychom podmínku měli pohromadě s tělem akceptu. Nevýhodou by bylo obtížnější využívání různých podmínek na více místech, ale i tento problém by se dal řešit přes nějaké další rozhraní jako v předchozím případě. To by mohlo být složitější, ale pokud bychom převážně používali jednu stejnou podmínku, pak by vzniklý zdrojový kód vlákna byl určitě jednodušší a čitelnější. Příkladem by mohl být již zmíněný buffer, kde se zřejmě podmínky akceptu nezmění, protože stále chceme hlídat stavy plného a prázdného bufferu. Toto otestování by bylo výhodné mít ve třídě s tělem akceptu. Navíc by se nám tento princip hodil při vytváření poslední funkcionality a tou je časově omezený pokus o akcept. Kompromisním řešením by mohlo být implementování obou řešení. Pak bychom však museli specifikovat vztah obou podmínek, jestli jsou ve vztahu logického součtu, součinu nebo jestli je např. podmínka v těle vlákna nadřazená.

3.9 Časově omezený akcept

V Adě máme možnost všechny zmíněné systémy použití ještě doplnit o časovou platnost. Pokud se nám v předem definované době nepodaří akceptovat žádný příchozí call, pak je pokus ukončen a je proveden speciální blok příkazů. I tuto funkcionalitu můžeme řešit dvěma různými způsoby. Na tento blok příkazů bychom se mohli dívat jako na speciální akcept. Tento akcept bychom museli speciálně zaregistrovat a implementovat pro něj opět dané rozhraní IEntry. Výhodou je podobný zápis těla tohoto speciálního případu akceptu (tedy vlastně neakceptu). Speciální registrace je nutná proto, že toto tělo nemá vlastní jméno a klientské vlákno ho nemůže volat. Registrace metody je v tomto případě trochu navíc a mohli bychom ji vynechat, protože se vlastně nikde klientským vláknům nezveřejňuje. Místo registrace bychom mohli vytvořenou instanci předat přímo metodě *accept* spolu s délkou doby, po kterou chceme čekat na akcept.

Druhou variantou by bylo speciální třídu pro tento případ nevytvářet a zapsat ho rovnou do těla vlákna. Pokud bychom metodu *accept* upravili, aby nám vracela boolean hodnotu, zda

skutečně došlo k akceptu, pak by nám stačilo tuto hodnotu otestovat a podle toho provést daný blok příkazů. Volání by v tomto případě bylo velice jednoduché.

```
if (!entryCall.accept(acceptedEntry, 20000)) {
     // blok příkazů, pro případ, že se žádný akcept nepovedl
}
```

Uvedený fragment kódu ukazuje velmi elegantní zápis časově omezeného akceptu, který je v tomto případě specifikován na 20 vteřin (vhodné by asi bylo dobu specifikovat v milisekundách). Protože je tento zápis velice jednoduchý, rozhodl jsem se ho v tomto případě upřednostnit. Pro časově neomezený akcept bychom mohli mít přetíženou metodu, kde nepředáme hodnotu intervalu a která nám vrátí void.

Jak bude vypadat implementace metody accept pro časové omezení? Lišit se od té předchozí příliš nebude. V Javě máme k dispozici metodu *wait* také s omezenou dobou platnosti v milisekundách. Na začátku metody accept nám postačí si pomocí voláním metody *System.currentTimeMillis()* zjistit aktuální čas v milisekundách a přičíst k němu požadovaný čas, který máme také v milisekundách. Tím zjistíme systémový čas, kdy končíme s čekáním na příchozí call. Poté celou již vytvořenou koncepci prohledávání fronty callů můžeme uzavřít do cyklu s podmínkou na konci. V této podmínce testujeme, zda systémový čas je menší než náš konečný čas. Poslední úpravou, kterou provedeme, je nahrazení nekonečného wait na objektu fronty za časově omezený *wait* a parametrem bude rozdíl doby aktuálního systémového času a času konce.

Probuzení vlákna z upraveného waitu může nastat ve dvou případech. Došlo ke konci čekání po stanovené době. V tom případě cyklus ukončíme a vrátíme na výstupu false, což znamená, že žádný akcept po stanovenou dobu neproběhl. Druhým případem probuzení vlákna je nový příchozí call. V tomto případě musíme projít frontu callů a pokusit se o nějaký akcept. Pokud se akcept povede, metodu ukončíme s návratovou hodnotou true. Pokud ne, provedeme další iteraci cyklu a znovu uspíme serverové vlákno. Celý systém acceptu ukazuje následující ukázka.

```
public boolean accept(HashMap<String, IAcceptCondition
acceptedEntry>, long timeOut) {
   long konec = System.currentTimeMillis() + timeOut;
   do {
     synchronized (calls) {
       if (!calls.isEmpty()) {
         for (Iterator<Call> it = calls.iterator(); it.hasNext; ) {
```

```
            Call call = it.next();
            if (acceptedEntry. containsKey(call.getName()) &&
                acceptedEntry.get(call.getName()).is Acceptable()) {
              // provedeme accept
              calls.remove(klic); // odstraníme call
              return true; // konec metody accept
            };
          }
        }
        list.wait(konec - System.currentTimeMillis() );
      }
    } while (konec > System.currentTimeMillis());
    return false;  // akceptování se v daném čase nezdařilo
}
```

3.10 Otestování výsledků

Na předchozích stránkách bylo zobrazeno, jakým způsobem jsem postupoval k vytvoření synchronizačního prostředku rendez-vous v běžném prostředí Javy. Vytvoření prostředku proběhlo v několika krocích, jejichž funkčnost bylo potřeba otestovat. Pro tyto účely vzniklo několik programů, které jsou dostupné na přiloženém CD. Všechny se spouštějí z běžného příkazového řádku a na příkazovém řádku také očekávají případné parametry, které jsou nutné pro běh programu. Pokud parametry nutné jsou, je uživateli vypsána stručná nápověda. Na CD jsou uložené ve tvaru projektu pro vývojové prostředí NetBeans verze 6.5.

První program má za úkol otestovat samotný princip synchronizace dvou vláken. Serverové vlákno nemá explicitně definovano žádné *entry*, program pouze testuje předání vstupních parametrů a převzetí výstupní hodnoty akceptu. Serverové vlákno v těle akceptu provádí pouze jednoduchou činnost, kdy na vstupu očekává objekt Integer a vrací také objekt Integer, jehož hodnota je zvýšena o hodnotu 1. Klientské vlákno provádí opakované volání a na konci kontroluje získanou a předpokládanou hodnotu. Mezi operace přijetí akceptu a otestování připravenosti výstupní hodnoty bylo zavedeno čekání o náhodné době, což mělo otestovat, jestli nedojde k deadlocku, pokud *notify* předběhne *wait*. Teprve, když byla ověřena funkčnost této konstrukce, bylo možné pokračovat v dalším vývoji.

Druhým krokem bylo vytvoření programu, který již obsahoval více serverových i klientských vláken a více definovaných entry. Tento program testuje, zda funguje správně proces front při více konkurenčních vláknech. Akcepty v něm nejsou podmíněné ani časově omezené a akceptují se všechna příchozí volání. Každé serverové vlákno má 3 entry: *plus, minus, stop*. Činnost jednotlivých akceptů odpovídá definovaným názvům: *plus* zvyšuje vstupní hodnotu,

minus ji snižuje. Entry *stop* ukončuje činnost serverového vlákna. Klientská vlákna prováděla předem definovaný počet volání plus a minus v náhodném pořadí na náhodném serverovém vlákně. Po dokončení činnosti vlákna se kontroluje získaná hodnota, zda souhlasí s předpokládanou hodnotou a výsledek testu je vypsán.

Třetí program již obsahuje kompletní systém rendez-vous, včetně podmínek a časového čekání. Serverové vlákno v tomto programu bylo pouze jedno a mělo definováno 3 entry: *plus, minus* a *hodnota*. Klientská vlákna zde jsou dvou druhů, jedno volá metodu *plus* a *hodnota*, druhá volají *minus* a *hodnota*. Obou vláken je stejný počet a pracují v náhodných intervalech. Vždy je nejdříve zavolána daná operace a poté entry *hodnota*, která vrací aktuální hodnotu. Akcept volání entry *plus* a *minus* je omezen tak, aby serverová hodnota byla stále v definovaném rozmezí, pokud by měla z tohoto rozmezí vybočit, není akcept proveden. Po zavolání jedné ze dvou operací je nutno nejdříve zavolat entry *hodnota* a teprve pak je možno pokračovat. Pokud není žádný akcept přijat po dobu 5 sekund, je činnost serverového vlákna ukončena. Tento program testuje, zda řídící rendez-vous objekt podporuje správně všechny funkcionality tak, jak je očekáváno.

3.11 Shrnutí dosaženého výsledku

V základním prostředí Javy byla vytvořena třída, jejíž instance umožňuje více vláknům provádět synchronizaci typu rendez-vous, které známe z programovacího jazyka Ada. Vytvořený objekt umožňuje serverovým vláknům zaregistrovat si neomezené množství různě pojmenovaných synchronizačních bodů – *entry*. Těmto vláknům pak umožňuje provádět podmíněné i nepodmíněné akcepty příchozích volání na definované skupině sebou registrovaných entry. Tento pokus navíc umožňuje omezit časově. Ostatním vláknům tento objekt poskytuje seznam definovaných entry pro libovolné vlákno, které je identifikováno svým jedinečným ID.

Tento prostředek vytváří rendez-vous za běhu programu, nezavádí žádný speciální jazykový konstrukt. Používání takového prostředku však může vést k programátorským chybám, které by bylo vhodné ošetřovat a programátora informovat pomocí systému výjimek. Příkladem takové chyby by mohl být pokus o akcept či pokus o volání neregistrovaných entry. V takovém případě by pravděpodobně došlo k uvíznutí některého vlákna. Další potenciální chybou by mohl být pokus o provedení volání nějakého vlastního *entry* serverovým vláknem. V tomto případě by zcela určitě došlo k deadlocku. Tyto chyby můžeme poměrně jednoduše detekovat ještě před provedením akce a vygenerovat nějakou vlastní výjimku.

Jiný typ chyby může nastat při samotném běhu programu a její detekce je problematická. Touto chybou je předčasné ukončení činnosti serverového vlákna. Jak již bylo zmíněno, každé serverové vlákno, které chce provádět rendez-vous, musí svá entry zaregistrovat, aby je bylo možné zjistit z klientských vláken a zavolat je. Na konci činnosti serverového vlákna je třeba provést odregistraci těchto entry, aby při případném volání byla vygenerována výjimka, ale nedošlo k uvíznutí klientského vlákna. Kdybychom měli k dispozici jazykový konstrukt, který by procházel překladačem, mohli bychom na chybu programátora upozornit, případně ji automaticky napravit. Ale i správně zapsaný program serverového vlákna, může skončit na některém výjimečném stavu a k odregistraci nemusí dojít. Takový stav je však problematické ošetřit, protože těžko můžeme ohlídat, zda serverové vlákno nějak nestandardně neukončilo svou činnost, pokud již klientské vlákno provedlo call. V tomto případě je totiž vlákno uspané a čeká se na provedení akceptu serverovým vláknem, který již nepřijde.

Problémem může být i efektivita tohoto programu. Zatímco přímo u jazykového prostředku může vhodně naprogramovaný překladač zajistit některé optimalizace zdrojového kódu, pak v tomto případě se spoléháme na obecný překladač a všechny případné kontroly provádíme až za běhu programu.

I přes tyto problémy máme fungující prostředek pro provedení požadovaného způsobu synchronizace v základní Javě a můžeme přistoupit k druhé části této práce a tou je úkol tento systém implementovat do projektu SAN. Nejdříve je ale důležité se s tímto projektem blíže seznámit a k tomu by měly sloužit následující kapitoly.

4 Projekt SAN

Projekt SAN neboli *Smart Active Node* je projekt, který je již několik let vyvíjen na Západočeské univerzitě v Plzni a slouží jako simulátor činnosti aktivních sítí. Projekt SAN vychází z projektu Grade32, který byl vytvářen v programovacím jazyce Delphi. Protože projekt Grade32 byl určen k jiným účelům, byl vytvořen nový projekt a pro tvorbu byl vybrán programovací jazyk Java.

4.1 Aktivní sítě

Současné počítačové sítě, které známe z běžného života, včetně jejich aktivních prvků bychom mohli označit jako pasivní sítě. Jejich činnost spočívá v pouhém přenášení dat mezi zdrojovým a jedním či více cílovými uzly. Aby mohlo dojít k přenosu dat, je potřeba data označit a přenést některé dodatečné informace, jako jsou zdrojové a cílové uzly, typ přenosového protokolu apod. Přenosový paket má tedy hlavičku a poté případně nějaká užitečná data. Hlavička musí mít přesný formát, aby mohla být pomocí aktivních prvků rozpoznána a paket zpracován. To bychom mohli označit jako hlavní nevýhodu současných počítačových sítí. Aktivní prvky mohou s paketem pracovat pouze tehdy, pokud jeho hlavičku rozpoznají a mají naprogramováno, jak s takovým paketem pracovat. Abychom mohli tedy nějakou obecnou službu v počítačové síti používat, musí tomu nejdříve předcházet nějaký proces standardizace, což obvykle trvá dlouhou dobu. Teprve pak je postupně nová služba podporována větším okruhem výrobců hardwaru a je zaváděna do běžného používání. Zavedení nového protokolu tak zabere dlouhou dobu, což můžeme v současné době například pozorovat na procesu zavádění IPv6 protokolu. Přesto, že jeho specifikace jsou známé již dlouhou dobu, stále je masivně používán protokol IPv4. Aby mohlo být používáno přímo IPv6, je nutno vyměnit veškeré síťové prvky, což je velmi nákladná záležitost. Stále se tedy používá protokol IPv4 a přes něj je pomocí systému tunelování možné provozovat IPv6. Kvůli těmto skutečnostem byl v agentuře DARPA *(Defense Advanced Research Projects Agency* – česky Agentura pro výzkum pokročilých projektů, která spadá pod americké ministerstvo obrany) vytvořen koncept aktivních sítí, který měl hlavní nedostatky pasivních sítí napravit.

Vzniklá koncepce měla tyto základní cíle:

- vytvoření architektury, která umožní flexibilně vyvíjet nové služby a nasazovat je do provozu
- povolení kontroly vytvořeným aplikacím nad aktivním prvkem
- vytváření zabezpečení prvků, které umožní vytvořit zabezpečenou síť bez centrálních zabezpečovacích prvků, jako jsou firewally

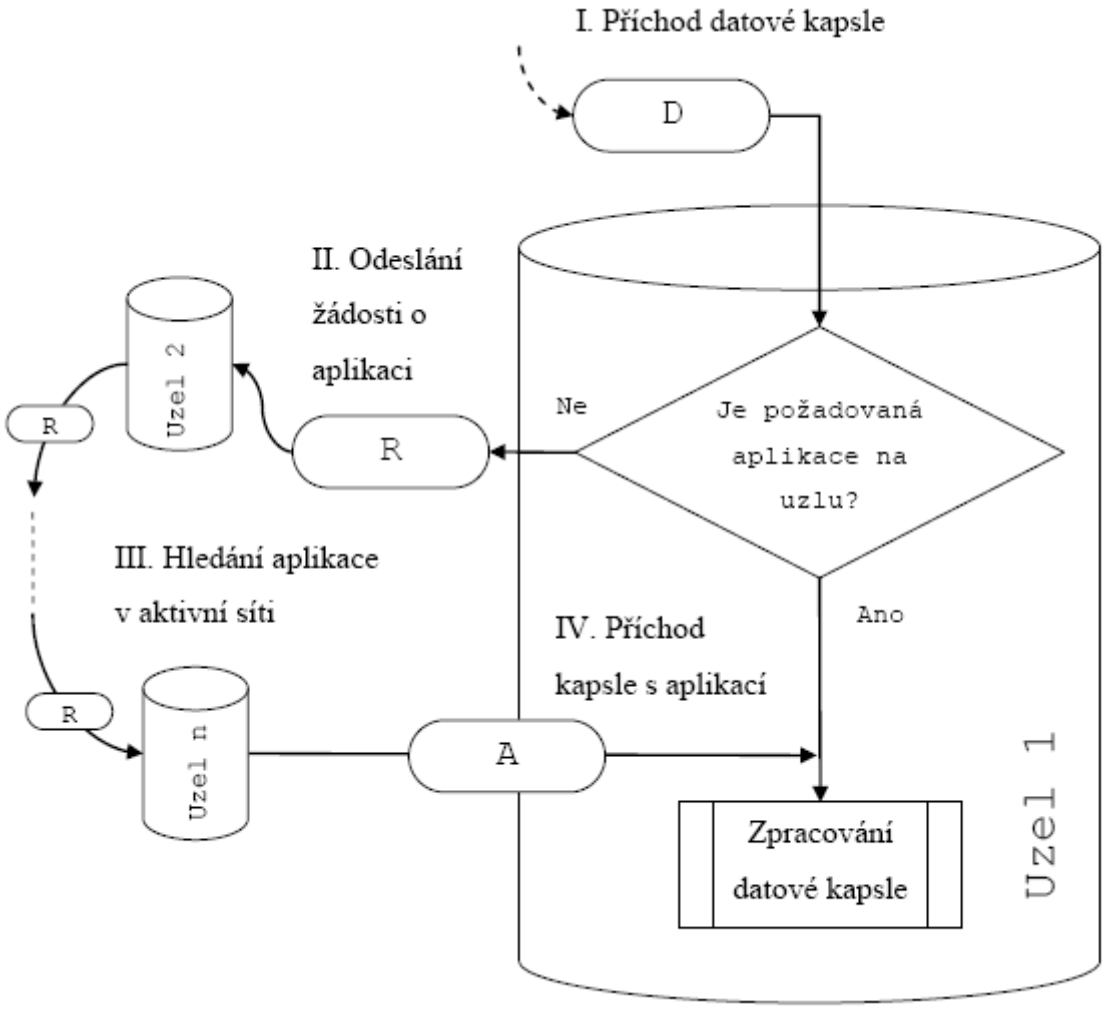
Podle těchto požadavků vznikl systém takzvaných kapsulí, které obsahují mimo jiné i identifikátor programu, který má být na směrovačích nad přenášenými daty spuštěn. Aby bylo možno program spustit, musí se na směrovač nejprve umístit, což lze provést pomocí 3 základních strategií:

- program na směrovač umístí administrátor. V tomto systému je uživateli pouze umožněn výběr aplikace a zadání hodnot jejích parametrů. Výhodou tohoto systému je, že jsou po síti přenášena jen užitečná data, přenos není nijak navýšen o zdrojový či binární kód aplikace. Další výhodou je otázka bezpečnosti, administrátor má nad sítí větší kontrolu. Nevýhodou je, že tento koncept příliš neodpovídá myšlence aktivních sítí, protože uživatel si aplikaci neprogramuje, pouze využívá funkcí, které mu připravil administrátor.

- program bude přenesen uživatelem před průchodem vlastních dat. V tomto systému si již uživatel vytváří potřebný program a před průchodem vlastních dat musí zajistit, že na všech potřebných směrovačích bude k dispozici vytvořená aplikace. To zvyšuje zatížení sítě a klade vyšší nároky na režii přenosu. Další nevýhodou je otázka bezpečnosti takových programů. Výhodou naopak je, že samotný přenos dat již nezvyšuje délku dat, protože aplikace již na směrovačích je k dispozici.

- program bude přenesen spolu s daty. V tomto systému je program přenášen společně s vlastními daty. To zvyšuje náročnost přenosu, protože přenášený program může být poměrně veliký. Systém je možné vylepšit tím, že data přeneseme pouze na počátku nebo jen pokud je to nutné. V kapsuli je opět umístěn identifikátor požadovaného programu, a jestliže na konkrétním uzlu v datovém úložišti žádaný program není, pak si směrovač může vyžádat daný program u zdrojového uzlu. Tento postup značně snižuje nároky na přenosovou kapacitu, protože program je přenášen pouze tehdy, pokud je to zapotřebí, což může být při prvním průchodu dat nebo pokud je program na směrovači vymazán.

Ať už máme systém doručování vytvořen jakoukoliv doručovací strategií, princip průchodu dat směrovačem je stejný. Při průchodu dat směrovačem je, podle identifikátoru uvedeném v kapsuli, rozpoznán program, který má být pro konkrétní data spuštěn. Po jeho provedení jsou data případně odeslána dál sítí tak, jak by to udělal běžný směrovač v pasivní síti. Systém hledání názorně dokumentuje obrázek 4.1, převzatý ze zdroje [d1]



Obr. 4.1 – vyhledání a spuštění aplikace na aktivním uzlu

4.2 Možnosti využití aktivních sítí

Koncepce aktivních sítí má mnoho potenciálních využití. Mezi hlavní bychom mohli zařadit již zmíněný rozvoj a využití experimentálních služeb a protokolů. Jiným využitím by mohla být problematika multicastového doručování, což bychom mohli aplikovat například při přenosu multimediálních obsahů (audio / video data) v reálném čase. Dalším využitím principu aktivních sítí by mohl být systém transparentní cache. To bychom mohli využít například při webovém provozu.

Dalším užitím by mohla být optimalizace zátěže v distribuovaném prostředí. V takových systémech máme pomocí počítačové sítě spojeno několik strojů dohromady, které provádějí nějaké složitější výpočty a tyto stroje spolu musí navzájem komunikovat. Přenos dat po síti je však vzhledem k výpočetním možnostem poměrně drahá záležitost a optimalizací může dojít k nárůstu výkonnosti celého systému. Pro simulaci a práci v tomto prostředí vznikl původní předchůdce projektu SAN, projekt Grade32.

Jiným praktickým využitím aktivních sítí by mohl být management sítě. Ke sledování sítě máme již dnes vytvořen protokol SNMP, jeho využití je však trošku těžkopádné a může být i poměrně náročné v provozu sítě. Systém aktivních sítí by mohl být mnohem efektivnější, mohl by lépe pracovat se síťovým prostředkem a lépe detekovat a reagovat na případné chyby v síťovém provozu. V neposlední řadě by aktivní sítě mohly pomáhat k bezpečnosti provozu v síti.

Velmi důležité by mohlo být i efektivnější řízení kvality přenášených služeb. Už dnes pociťujeme s nárůstem rychlosti připojení koncových uživatelů v internetu také velký nárůst přenosů multimediálního obsahu. Streamovaná videa, videokonference a podobné služby jsou dnes běžnou součástí internetu a je nutné věnovat velkou pozornost nastavení priorit služeb podle kvality (QoS).

Tento výčet použití aktivních sítí samozřejmě není konečný, příkladů možného použití by bylo mnohem více a není to předmětem této práce. Některé myšlenky aktivních sítí se dnes využívají, ale pouze na vyšších úrovních síťového přenosu. Koncept byl určen pro síťovou vrstvu v ISO/OSI modelu, ale k realizaci zatím příliš nedošlo. Celý návrh aktivních sítí byl poněkud odsunut na vedlejší kolej a v nejbližší době se nezdá, že by vývoj a případné nasazení tohoto konceptu pokračovalo. Tento fakt byl jeden z důvodů, proč vznikl projekt SAN a jeho cílem je simulace a prozkoumání možností, výhod a nevýhod systému aktivních sítí.

4.3 Bezpečnost aktivních sítí

Výhody systému aktivních sítí již byly zmíněny, ovšem s každou novou technologií a obzvláště s konfigurovatelnou a programovatelnou je nutné zvážit i možné nevýhody a problémy při použití, a to zejména v otázce bezpečnosti. Bezpečnost aktivních sítí bychom mohli rozdělit do několika druhů.

První obecnou problematikou je vlastní zabezpečení aktivního prvku. Zde je nutné vyřešit otázku přístupu k prvku. Má mít každý uživatel právo umístit na aktivní prvek svůj vlastní program? Má každý uživatel právo spouštět každý program?

Druhá oblast zabezpečení se týká přímo spouštěných programů. Běžné směrovače mají dnes vestavěnou podporu určitého množství protokolů, se kterými umějí pracovat. Tyto programy jsou pak většinou dobře odladěny a funkční. I tak však často musejí výrobci dodávat aktualizace firmwaru. Navíc i směrovač s dobře odladěným programem může selhat. K tomu stačí i např. výkyv napětí v elektrické síti, které způsobí selhání hardwaru.

K těmto možným chybám se nyní mohou přidat i chyby programátorů spouštěných aplikací. Zřejmě každý programátor se někdy setkal s nechtěnou chybou ve svém programu, která může způsobit jeho selhání nebo zacyklení. Programy ve směrovači jsou spouštěné při průchodu dat sítí a v dané chvíli mají k dispozici hardwarové prostředky směrovače. Těmito prostředky jsou především procesor a paměť, jejíž velikost je omezená. Neoptimální nebo chybný program by mohl způsobit havárii systému směrovače, popř. jeho zahlcení nebo zahlcení celé sítě. Chybný program také může poškodit přenášená data. Poslední hrozbou je, že by program mohl získat přístup i k jiným datům, která procházejí směrovačem. To by mohlo vést například k únikům přihlašovacích údajů přenášených ve standardních přenosových protokolech či k odchytávání šifrovacích klíčů apod.

Řešením některých zmíněných problémů by mohl být výběr vhodného programovacího jazyka a v tomto jazyce bychom mohli zakázat používání některých rizikových instrukcí, jako jsou skoky v programu z místa na místo, nekonečné cykl. Při vykonávání programu bychom mohli zakázat používání některých funkcí směrovače, apod. To ovšem znamená mít přímou kontrolu nad vykonáváním takového programu. Dalším úkolem v otázce bezpečnosti je zavedení autentifikace uživatelů a další.

4.4 Architektura projektu SAN

Simulátor aktivních sítí, projekt SAN, vychází ze zmíněných předpokladů a tomu odpovídá i architektura projektu. Pro vývoj projektu byl zvolen jazyk Java, jehož OOP architektura je pro vícevrstvou aplikaci ideální. Celý systém byl rozdělen na tři logické vrstvy.

První je síťová vrstva. Ta zajišťuje komunikaci mezi sousedními uzly pomocí standardního TCP/IP protokolu. Každý uzel má definován port, na kterém naslouchá pro příchozí spojení. Vyhledání ostatních uzlů probíhá pomocí *broadcastu.*

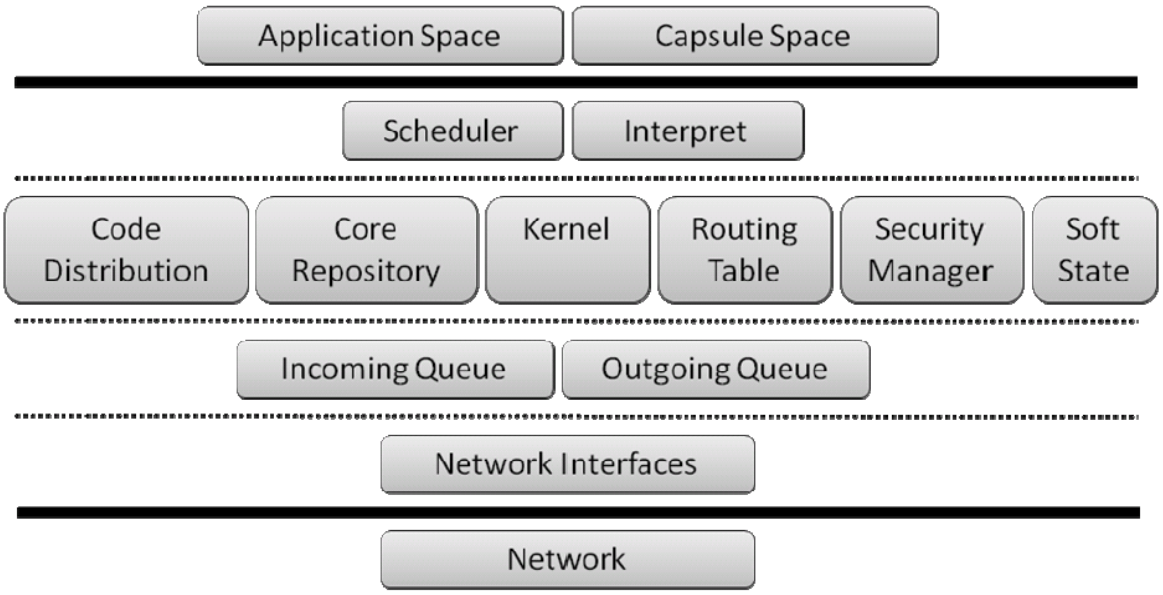
Druhou vrstvou je jádro, které zajišťuje několik funkcí. První hlavní funkcí je zajištění přenosu kapsulí mezi virtuálními uzly. K tomu využívá služeb síťové vrstvy. Každý virtuální uzel je adresován pomocí čísla sítě a čísla uzlu v síti. V konfiguračním souboru lze tyto parametry definovat a také stanovit virtuální cesty na ostatní uzly. Tím lze vytvořit virtuální síť, se kterou simulátor bude pracovat. Protože virtuálních sítí může být více, pak podle vytvořené topologie zajišťuje i routování přenosů kapsulí.

Druhou funkcí jádra je distribuce programových kódů a správa programového repositáře. Projekt SAN používá systém distribuce kódu zároveň s daty a to pouze, pokud je to nutné. Při přijetí dat je prohledán programový repositář a jádro se pokusí najít žádanou aplikaci. Jestliže aplikace v repositáři daného uzlu není, pak je tato aplikace vyžádána od uzlu, že byla data obdržena. Každá aplikace je identifikována názvem a hashovacím otiskem. Ten má zajistit například detekci, že požadovaná aplikace je jiné verze, než aktuálně uložená v repositáři aktivního uzlu.

Poslední funkcí jádra je spouštění vybrané aplikace či kapsulového programu. K tomu je využit interpret javovského bytecodu.

Třetí vrstvou v projektu SAN jsou již samotné interpretované programy. Spouštěné aplikace jsou uloženy v programovém repositáři daného uzlu. Aktivní programy v projektu SAN rozdělujeme na dva druhy. Aplikace, která je spouštěná pomocí příkazového řádku a kapsulový program, který je spouštěn při zpracování přenášených dat. Oba typy programů tvořící aplikaci mohou být vyvíjeny zvlášť a přeložené pomocí standardního javovského překladače. Aktivních programů může najednou běžet větší množství, o jejich plánování se stará plánovač jádra.

Architekturu projektu SAN přehledně dokumentuje obrázek 4.4, převzatý ze zdroje [d2]



Obr. 4.4 – architektura projektu SAN

Při vytváření synchronizačního prostředku rendez-vous se budeme pohybovat především ve druhé vrstvě s cílem vytvořit nový prostředek pro třetí vrstvu. Z toho důvodu je nezbytné si konkrétní úrovně popsat podrobněji.

4.5 Vytváření aktivních programů v projektu SAN

Jak již bylo zmíněno, v projektu SAN se ve třetí vrstvě setkáme se dvěma druhy programů. První druhem jsou aplikace spouštěné pomocí funkce jádra, tedy takové, které můžeme spustit z příkazové řádky konzole. A potom jsou zde kapsulové programy, spouštěné při průchodu dat uzlem. Aplikace i kapsulový program tvoří jeden aktivní program. Kapsulový program je v aplikaci nepovinný, jestliže aplikace neodesílá data (kapsule) sítí. Při tvorbě aplikace tak vytváříme dva samostatné programy, které jsou zabalené do jednoho souboru jar. Protože každý program je tvořen samostatnou třídou, vyskytuje se typicky v balíku více tříd. Aby bylo možno určit pro každý typ programu jeho konkrétní třídu se vstupní metodou, je nutné ještě definovat pomocný soubor Manifest.mf.

Aby oba dva typy programů mohly být dynamicky spouštěny pomocí druhé vrstvy systému a mohly pracovat s funkcemi, které jim tato vrstva poskytuje, je nutné, aby implementovaly určité rozhraní. Toto rozhraní definuje název vstupní metody a způsob předání parametrů. Oba programy se však liší svými možnostmi, a proto tato rozhraní jsou definována dvě.

Prvním typem programu je tzv. aplikace. Hlavní třída této aplikace implementuje rozhraní IApplication. Toto rozhraní definuje vstupní metodu *main* s polem *String*, které obsahuje parametry příkazové řádky. Druhým parametrem je rozhraní IApplicationAPI.

Toto rozhraní zpřístupňuje všechny funkce jádra, které aplikace pro svůj běh potřebuje. Příkladem metody z tohoto rozhraní je getStdOut, která nám vrací PrintWriter pro standardní výstup. V klasických java programech máme standardní výstup definován jako System.out, zde nám obdobu tohoto streamu zpřístupňuje daná metoda. Tento stream je pomocí jádra napojen na vstupně – výstupní zařízení, v případě projektu SAN je to konzole.

Třetím parametrem vstupní metody *main* je objekt *Guid*. Všechny spuštěné aplikace v systému jsou identifikovány jedinečným řetězcem *Guid*. Tento objekt byl v systému zaveden kvůli identifikaci aplikací při vlastním interpretování programu. Více o této problematice je v kapitole 4.12.

Druhým typem programu je kapsulový program pro zpracování průchozích dat. Hlavní třída tohoto programu implementuje rozhraní ICapsule, které opět zavádí vstupní metodu *main*. Tento typ programu je spouštěn při zpracování dat, a proto nemá žádné vstupní parametry příkazové řádky. Jediným parametrem metody *main* je rozhraní ICapsuleAPI, které zpřístupňuje vybrané funkce. Mezi těmito funkcemi jsou metody pro zpracování přenášené kapsule, jako jsou data v kapsuli, zdrojová a cílová adresa apod. Naopak tento program nemůže vypisovat žádná data na konzoli.

4.6 Spouštění aplikací v projektu SAN

Spouštění aplikací je záležitostí jádra, což je umožněno pomocí metody *runApplication* z rozhraní *IApplicationAPI*. Tato metoda spouští aplikaci z programového repositáře, která je definována jménem aplikace. Vytvoří pro ni nové *Guid* a spustí nový interpret a předá spouštěné aplikaci případné parametry.

Aplikace je možné spustit v konzoli z příkazové řádky. Zatímco konzole je vytvářená přímo jádrem, příkazová řádka je samostatná aplikace. Při startu projektu SAN je po zpracování konfiguračního souboru spuštěna aplikace *Init*. Ta pomocí příslušné metody vytváří konzoli a spouští aplikaci příkazového řádku – *Shell*. Tento program definuje základní příkazy, které v příkazovém řádku máme. Program *Shell* po zpracování příslušného příkazu pak spouští další programy z programového repositáře pomocí stejné metody, jako kterákoliv jiná aplikace.

4.7 Interpretování aplikací

V předcházející kapitole bylo popsáno, jakým způsobem se vytvářejí aktivní programy v projektu SAN. Než se pustím do popisu způsobu, jakým jsou samotné aplikace v projektu SAN prováděny, je nutné se podívat na fungování překladu a vykonávání standardních javovských programů. V této problematice jsou důležité dva termíny: kompilace a interpretace.

Při tvorbě jakéhokoliv programu používáme k zápisu příkazů programovací jazyk. Po vytvoření programu ve zdrojovém kódu je potřeba program přeložit do spustitelné formy pro počítač, kdy program bude zapsán jako sled strojových instrukcí. Tento proces se nazývá kompilace a potřebujeme k němu překladač (kompilátor). Výsledkem práce kompilátoru však nemusí být přímo spustitelný kód. Během procesu kompilace dochází většinou k několika dílčím operacím. Obecně bychom proces kompilace mohli rozdělit na několik kroků:

- lexikální analýza
- syntaktická analýza
- sémantická analýza
- optimalizace kódu
- generování cílového kódu

V tomto přehledu je mnoho kroků, které vedou k převodu zdrojového kódu do spustitelného programu. Ne všechny musí být prováděny, nebo naopak může být více kroků spojeno dohromady. Výsledkem procesu je převedení zdrojového kódu do spustitelného tvaru. U rozsáhlejších programů se občas setkáme s mezikrokem, kdy je program přeložen do tzv. objektového kódu. Výsledkem tohoto kroku je překlad programu do cílového strojového kódu, ale program je rozdělen na několik částí. V tomto mezikroku jsou části provázány různými návěštími umožňujícími pozdější spojení do spustitelného programu, tento proces (linkování) provádí program linker.

Po kompilaci a slinkování, které často bývá provedeno současně s překladem, máme spustitelný program jako sled strojových instrukcí. Tyto instrukce jsou různé pro každou platformu. V případě Javy je překlad realizován do tzv. bytecodu, což je obdoba strojového kódu, který je však platformově nezávislý. Aby mohl být spouštěn na konkrétní platformě, je potřeba mít k dispozici interpret bytecodu. Nejznámější takový program je Java Virtual Machine. Tento program vytváří virtuální stroj, který provádí překlad bytecodu pro skutečný operační systém. Tento proces se nazývá interpretace.

4.8 Interprety programů

Interpretem můžeme nazvat takový program, který provádí interpretaci námi vytvořeného programu. Podle způsobu jejich činnosti bychom je mohli rozdělit do tří skupin.

První typ interpretů provádí přímo interpretaci zdrojového kódu. Tento způsob provádění je výhodný zejména pro rychlý vývoj programů, protože odpadá někdy časově náročná kompilace, výsledky vidíme hned a můžeme je opravit. Další výhodou je přenositelnost programu, zdrojový kód přeneseme mnohem snadněji než přeložený program.

Druhým způsobem činnosti interpretu je překlad do optimálnějšího mezikódu, který je poté spuštěn. Mezikódem může být např. AST (*Abstract Syntax Tree*). Příkladem by mohl být např. jazyk Perl, Python apod.

Třetím způsobem je interpretování bytecodu. Jeho výhodou je, že program již byl zkompilován, je syntakticky správný a při vykonávání je jeho činnost tedy rychlejší. Typickým příkladem je právě Java a interpretem je Java Virtual Machine.

4.9 Projekt SAN a interprety

V projektu SAN dochází ke spouštění vlastních aplikací, a proto je potřeba mít také interpret. V kapitole 4.3 bylo zmíněno, že aktivní uzel by v rámci bezpečnosti měl mít plnou kontrolu nad interpretováním aplikace. Z tohoto důvodu nemůžeme použít běžné načtení balíku spouštěné aplikace pomocí standardního class loaderu. Aplikace by tak byla vykonávána přímo v JVM, ve kterém běží samotný program aktivního uzlu. Hlavní nevýhodu řešení demonstruje použití příkazu: *System.exit(0)* ve spouštěném programu. Tento příkaz by způsobil okamžité ukončení činnosti celého aktivního uzlu, což je nepřijatelné.

Hledalo se tedy řešení, jak provádět vlastní interpretaci kódu aplikací. Protože v začátcích vývoje projektu SAN nebyly přesně známy některé požadavky, byl na začátku nasazen interpret BeanShell. Program BeanShell je volně šiřitelný interpret javovského kódu, který byl vytvořen také v Javě. BeanShell provádí přímo interpretaci zdrojového kódu. Většinu funkcí tedy umožňuje interpretovat bez překladu do bytecodu. Jeho hlavní výhodou je, že ho lze spustit z javovské aplikace a protože pracuje ve stejném JVM jako spouštějící aplikace, umožňuje spouštěnému programu pracovat s objekty vytvořenými uvnitř této aplikace.

Použití BeanShellu mělo kromě výhod i některé nevýhody. V nasazené verzi BeanShellu nebyly například podporovány typované kolekce (Collection<Integer>), které již byly

zmíněny pro tvorbu rendez-vous. Další nevýhodou byla nemožnost mít více javovských tříd ve více souborech. Hlavní nevýhodou je ovšem fakt, který byl popsán jako výhoda, a to, že interpret běží ve stejném JVM, jako spuštěná aplikace. Při použití již zmíněného příkazu *Systém.exit*, opět dojde k okamžitému ukončení programu celého uzlu. Nemáme tak opět kontrolu nad vlastní interpretací. Tyto problémy a některé další vedly později k tomu, že byl vytvořen, rovněž v Javě, nový interpret, který kontrolu interpretace umožní. Současný stav projektu je tedy takový, že obsahuje dva nezávislé interprety. Aplikaci můžeme při vývoji přeložit dvěma různými způsoby a podle způsobu překladu je pak spuštěn příslušný interpret.

4.10 Vlastní interpret

Zatímco interpret BeanShell interpretuje přímo zdrojové kódy programu, byl nový interpret vytvořen tak, aby interpretoval zkompilovaný bytecode. Ačkoliv se původně uvažovalo o vytvoření i vlastního překladače, nakonec se tak nestalo, a tak interpret pracuje s bytecodem, který vytváří standardní javovský kompiler. Nově vytvořený interpret byl vytvořen až při používání BeanShellu, a proto kvůli kompatibilitě přebírá některé logiky a implementuje určitá rozhraní, aby se s oběma interprety nechalo společně pracovat. Hlavním cílem této práce je, aby synchronizace rendez-vous fungovala v tomto interpretu. Proto je zapotřebí podrobněji vysvětlit princip činnosti tohoto interpretu.

4.11 Princip činnosti interpretu

Tento interpret se od BeanShellu liší dvěma hlavními fakty. Provádí interpretaci přeloženého bytecodu a interpretace provádí ve svém paměťovém prostoru. Interpret si tedy sám řeší způsob, jakým bude vytvářet a uchovávat všechny proměnné, objekty, reference apod. Hlavní věcí, kterou musí interpret provádět, je načítání tříd. V interpretu se vyskytují dvě možnosti, jak třídu načíst: SoftClass a HardClass.

Třída HardClass se vytváří tehdy, pokud ji můžeme načíst v některém pracovním adresáři. Pokud takto dokážeme najít požadovanou třídu, pak máme její bytecode a ten můžeme sami interpretovat. V současné době je takto načítána převážná většina tříd, včetně standardních jako je java.lang.String. Třídy z balíčku aplikace by takto měly být načítány všechny.

Druhým případem je způsob SoftClass. Ten je využíván k některým třídám ze serveru SAN, nebo k těm, které se v definovaných cestách najít nepodařilo. Instance těchto tříd se poté zkoušejí vytvořit jako normální Java objekty. Práce s nimi potom probíhá přes reflexi

a vykonává ji standardní JVM. Původním záměrem bylo jako HardClass načítat pouze třídy obsažené v balíčku interpretované aplikace, všechny ostatní třídy se měly načítat jako SoftClass. Vedl k tomu předpoklad, že důležité je interpretovat pouze vytvořené programy, vše ostatní jsou vestavěné funkcionality Javy. Později došlo k úpravě a nyní jsou jako HardClass načítány i standardní javovské třídy.

Aby bylo možné s oběma typy načtených tříd pracovat, implementují společné rozhraní *Class* ze stejného balíku *classLoading*. Díky tomuto rozhraní můžeme načtené třídy také cachovat, což významně urychlí činnost interpretu. Pojmenování společného rozhraní *Class* však osobně považuji za poněkud nešťastné, protože při pozdějších nutných úpravách interpretu několikrát docházelo k záměně za stejně pojmenovaný objekt třídy *Class* z balíku *java.lang,* což velmi stěžovalo práci.

Jakmile máme třídu načtenou, můžeme s ní v interpretu pracovat. V programu obvykle práce s třídou začíná vytvořením její instance, což je provedeno klíčovým slovem *new*. Podle typu načtení je vytvořená instance interpretována jako třída *SoftReference* nebo *HardReference*. Zatímco třída *SoftReference* obsahuje pouze referenci na vytvořený objekt, který je uložen jako Object a získání proměnných či metod je vyřešeno pomocí reflexe, u *HardReference* si tyto operace zajišťuje interpret. Ve třídě *HardReference* tedy existuje bytebuffer a při manipulaci s proměnnými je nalezena příslušná pozice v bytebufferu a operace provedena.

4.12 Problém spojení Soft a Hard referencí

Při zápisu programu samozřejmě nerozlišujeme žádné třídy, jak budou prováděny v interpretu a proto se stává, že soft a hard reference spolu komunikují. Prvním takovým případem je již samotné odesílání a přijímání kapsulí. Jak již bylo zmíněno, při tvorbě aplikace je vstupním metodám předáno rozhraní, které umožňuje využívání některých funkcí aktivního uzlu. Tato API jsou v interpretu předávána jako soft reference. Jestliže aplikace chce přijímat příchozí data ze sítě, je nutné si zaregistrovat listener, který je zavolán při příchodu dat. Původní návrh bylo využití konstrukce *addOnSendDataListener(this)*. Parametr *this* v tomto případě je hard referencí a metoda *addOnSendDataListener* z předaného API je soft referencí. Při zavolání dochází k převodu na soft referenci, je vytvořena kopie objektu a obsluhu provádí JVM. Tím opět ztrácíme kontrolu nad interpretací kódu aplikace a to je nepřijatelné.

Z tohoto důvodu byl zaveden zmíněný identifikátor spuštěných aplikací *Guid*. Objekt *Guid* je tvořen jako pole bytů, s implementovanou metodou pro vzájemné porovnání. Pro snadnější

práci má i převod na textovou formu identifikátoru. Tento identifikátor přiděluje jádro při spouštění aplikace a je při spouštění předán také aplikaci. Při registraci listeneru je zaregistrován daný *Guid*. Převod na soft referenci v tomto případě nenastává, ale i kdyby nastal, nevadilo by to. Podle identifikátoru je v případě potřeby jen nalezena hard reference vstupní třídy aplikace. Jakmile jsou přijata data, je spuštěn nový interpret, předána mu hard reference podle daného *Guid* a spuštěná obslužná metoda.

5 Popis řešení rendez-vous v projektu SAN

5.1 Současný stav projektu SAN

Projekt SAN se stále vyvíjí a některé funkcionality a myšlenky v něm ještě nejsou doimplementované. Jednou z chybějících věcí je absence plánovače, se kterou se počítá. Současný stav v projektu je takový, že nově vytvořený a spuštěný interpret je nové vlákno v JVM. Jakmile je spuštěn, běží až do ukončení interpretovaného programu. Z tohoto důvodu můžeme některé věci ohledně synchronizace přenechat JVM, ale musíme je provádět mimo interpret.

Další důležitou skutečností je, že vytvořený interpret bytecodu nepodporuje vlákna. Chceme-li primární vlákno (jediné v programu) uspat, je nutno použít konstrukci *Thread.sleep* v podmíněné smyčce (aktivní čekání).

5.2 Požadavky na rendez-vous v projektu SAN

Zatímco v běžné Javě byla potřeba synchronizovat více vláken v jedné aplikaci, v projektu SAN je cílem vytvoření systému pro synchronizaci činnosti více jednovláknových programů. Projekt SAN rozlišuje dva typy programů: aplikace a kapsulové programy. Pro oba typy programů platí, že mohou být klientskou stranou rendez-vous. Naopak serverovou stranou může být pouze aplikace.

Jádro aktivního uzlu by mělo poskytnout metody pro provedení tohoto způsobu synchronizace, umožnit při něm výměnu dat apod. Servery je opět nutné definovat za běhu aplikace a vytvořit prostředky pro získání seznamu zaregistrovaných serverů a jejich *entry*. Hlavní podmínkou je fungování prostředku při použití vlastního interpretu.

5.3 Specifikace rendez-vous

Jako základ pro řešení byl do projektu SAN implementován již vytvořený způsob rendez-vous pro čistou Javu. Při vyzkoušení v *BeanShell* interpretu byl tento způsob funkční a nebylo potřeba jej měnit.

Důležité bylo určit, jakým způsobem budou jednotlivé rendez-vous servery a jejich entry identifikovány. Protože již v projektu SAN máme zavedený identifikátor *Guid*, bylo rozhodnuto, že rendez-vous servery jím budou také identifikovány. Jelikož ne každá aplikace však provádí rendez-vous, tak by server neměl být identifikován přímo *Guid* aplikací.

Aplikace, která se registruje jako rendez-vous server získá tedy pro tento server nové *Guid,* které s *Guid* aplikace nemá nic společného, mohlo by tedy být i stejné.

Dále je nutné určit, jakým způsobem identifikovat jednotlivá *entry*. Každý server může mít nekonečně mnoho různě pojmenovaných *entry,* v systému SAN však může být najednou více různých *entry* se stejným jménem. Ze zmíněných vlastností vyplývá, že každé *entry* můžeme jednoznačně určit jako *Guid* serveru a název *entry*. Protože popisování *entry* pomocí dvou objektů (*Guid* a String) by v některých případech mohlo být nešikovné, rozhodli jsme se ještě zavést jednoznačný identifikátor *entry* jako jediný objekt. Možností by bylo opět využít objekt *Guid*, ale protože bychom už v systému měli hodně objektů *Guid* s různými významy, rozhodli jsme pro identifikaci využít objekt Integer. Při registraci *entry* je tak přiděleno tomuto *entry* unikátní číslo v systému - ID. Každé *entry* tak můžeme adresovat dvěma způsoby: pomocí *Guid* serveru a názvu *entry*, nebo pomocí jeho ID. Oba tyto způsoby musejí být volně zaměnitelné, proto je potřeba vytvořit prostředky pro zjištění jejich seznamu a případně i převodu jednoho způsobu na druhý – tedy umožnit zjištění ID nebo dvojice Guid a název.

Posledním požadavkem byl způsob tvorby těla akceptu daného *entry*. Nakonec i z důvodu jednoduché tvorby serverové aplikace jsme se rozhodli, že rendez-vous server by měl implementovat nějaké rozhraní s metodou *entry*, které bude předán název akceptovaného *entry* a předávaná data.

5.4 Popis řešení

Nově vytvářený prostředek je tvořen několika třídami a rozhraními. Aby je bylo lehké v systému dohledat, jsou všechny umístěné do balíku san.rendezvous. Vytvořené řešení se příliš neliší od řešení v čisté Javě, avšak vzhledem k odlišným specifikacím k několika úpravám došlo. Tyto úpravy se týkaly především změn v identifikaci serverů a jejich *entry*. Servery jsou identifikovány jako Guid, proto bylo potřeba změnit definici klíčové hodnoty pro HashMapu, která ukládá jednotlivé servery. Stejně tak bylo potřeba vytvořit novou HashMapu pro uchování jednotlivých *entry* a jejich ID. Klíčová hodnota je ID entry a hodnotou je třída *EntrySet*, která uchovává pro každé entry trojici hodnot: ID, Guid serveru a název *entry*.

Pro možnost ovládání rendez-vous z aplikací i kapsulových programů bylo nutné vytvořit některé metody a ty zpřístupnit podle potřeby oběma typům. Zatímco metody pro klientskou stranu je nutné zpřístupnit oběma typům programů, serverové metody jen aplikacím. Jak již

bylo zmíněno, metody jádra jsou programům předávány při spuštění jako rozhraní IApplicationAPI a ICapsuleAPI. Tato dvě rozhraní jsou zcela nezávislá. Protože klientské rendez-vous metody jsou společné, bylo vytvořeno nové rozhraní ICommonAPI, které obě zmíněné rozhraní rozšiřují. Toto rozhraní definuje všechny potřebné metody pro klientskou stranu rendez-vous. Metody pro serverovou stranu jsou definovány v rozhraní IApplicationAPI.

5.5 Vytvoření rendez-vous klienta

Pro vytvoření rendez-vous klienta slouží několik metod z rozhraní ICommonAPI. Jsou zde dvě metody, které provádějí volání (call) a liší se způsobem adresování volaného *entry*. Volání je možné provádět buď pomocí ID *entry* nebo pomocí *Guid* serveru a názvu *entry*. Obě metody jako parametr přijímají typ Object. Nakonec jsem se rozhodl použít pouze proměnnou Object, ne pole Object. Důvodem je, že pokud bychom předávali pouze jednu proměnnou, je zbytečné vytvářet pole. Navíc pole objektů je také Object. Pro jednoduchou proměnnou Object také hovoří fakt, že pro programátora může být příjemnější si vytvořit novou třídu, která potřebné parametry obalí. Obě metody pro volání jako návratovou hodnotu vrací také typ Object.

Další metody v ICommonAPI slouží ke zjištění registrovaných *entry* na daném uzlu. Je zde metoda, která pro zadané jméno *entry* vrátí pole *Guid* serverů, které takto pojmenované *entry* mají registrované. Další metoda vrátí jména *entry*, které má zaregistrovaný server se zadaným *Guid*. Poslední tři metody slouží k převádění obou identifikačních způsobů na druhý. Jedna dvojice metod vrátí seznam jmen a seznam *Guid* pro zadané ID *entry* a pak je zde poslední metoda, která vrátí ID podle jména a *Guid* serveru. Pomocí těchto metod není obtížné vytvořit rendez-vous klienta. Stačí nám pouze zjistit identifikátor *entry* a provést volání.

5.6 Vytvoření rendez-vous serveru

Rendez-vous server představuje třída, která implementuje rozhraní *IRendezvousServer*. Toto rozhraní definuje metodu *entry*, která má parametr String s názvem akceptovaného *entry* a parametr typu Object, ve kterém je případný parametr volání. Metoda *entry* provádí tělo akceptu, její návratová hodnota je také typu Object. Tato hodnota je předána klientovi po skončení těla akceptu. Důležité je, že metoda *entry* vlastně tvoří tělo všech definovaných *entry*. Které *entry* je konkrétně obsluhováno, musí programátor rozlišit pomocí předaného názvu *entry*.

K vytvoření rendez-vous serveru slouží některé metody z rozhraní *IApplicationAPI*. Toto rozhraní definuje (mimo jiné) metodu *registerServer* pro zaregistrování serverového objektu. Tato metoda je přetížená a má dvě varianty. První varianta má pouze jediný parametr pro rozhraní *IRendezvousServer*. Tato metoda provede zaregistrování serveru a vrátí přidělený *Guid* pro server. V případě, že by došlo k nějaké chybě při registraci, je vrácena hodnota null.

Druhá varianta má navíc parametr *Guid*. Tím je umožněno si přímo definovat vlastní *Guid* pro daný server. Tato varianta vrací hodnotu boolean. V případě, že je požadované *Guid* pro server volné, je provedeno zaregistrování a vrácena hodnota true, v opačném případě hodnota false.

Po úspěšné registraci serveru je nutné zaregistrovat jednotlivá entry. K tomu slouží metoda *registerEntry*, která má parametry *Guid* zaregistrovaného serveru a String pro název daného *entry*. Metoda vrací typ Integer s ID zaregistrovaného *entry*. Identifikátory *entry* jsou v celém běhu aktivního uzlu jedinečné, tvoří postupnou číselnou řadu. Registrovaný název *entry* musí být jedinečný pro server s daným *Guid* a také *Guid* serveru již musí být zaregistrované. V případě nějaké chyby, například při nesplnění těchto podmínek, vrací metoda hodnotu null.

Jestliže máme nadefinované potřebné *entry*, můžeme provést akcept. K tomu slouží přetížená metoda *makeAccept*. První varianta této metody má pouze parametr *Guid* pro zaregistrovaný server, který akcept provádí. Tato varianta provádí akcept na všech *entry* registrovaných tímto serverem přesně v pořadí, v jakém bylo provedeno volání. Druhá varianta má navíc pole String, které umožňuje omezit akceptované *entry* na vybraný seznam. Časově omezené akcepty ani podmíněné akceptování implementováno prozatím nebylo.

Po skončení činnosti aplikace je nutné provést odregistrování serveru. K tomu slouží metoda *unregisterServer*, která má parametr *Guid* pro daný server. Tato metoda provede odregistrování všech serverem registrovaných *entry* a ukončí všechna případná volání klientských programů. Jako návratovou hodnotu akceptu v tomto případě vrátí hodnotu null. Důležité je, aby metoda byla skutečně zavolána pro každý zaregistrovaný server, v opačném případě by docházelo k uvíznutí programů volajících daná *entry*.

5.7 Ukázkové aktivní programy

Při tvorbě rendez-vous bylo potřeba toto řešení vyzkoušet, zda funguje správně. K tomuto účelu byly postupně vytvořeny dva testovací aktivní programy. Jejich jména jsou rendezvous

a rendezvous2. Spustit je můžeme v konzoli pomocí příkazu run s případnými parametry příkazové řádky oddělené mezerou.

Příkladem spuštění je příkaz: *run rendezvous 10 20*

5.8 Ukázkový program rendezvous

První program funguje pouze v rámci jednoho aktivního uzlu. Je to obdoba druhého testovacího programu popsaného v kapitole 3.9. Jeho činnost spočívá ve vytvoření zadaného počtu serverů a zadaného počtu klientů. Počet serverů i počet klientů je možné definovat jako parametr příkazové řádky. Prvním parametrem je počet serverů, druhým počet klientů. Při spuštění programu bez parametrů jsou použity výchozí hodnoty nastavené v programu, momentálně je to 5 serverů a 13 klientů.

Každý server má definované 3 *entry*: *plus*, *minus* a *stop*. Jako parametr je očekáván objekt Integer. Entry *plus* provede zvýšení zadané hodnoty o 1 a novou hodnotu vrátí zpět klientovi. Entry *minus* naopak provede snížení zadané hodnoty. Server provádí akcepty tak dlouho, dokud není akceptováno volání stop. Pak vypíše na standardní výstup počty jednotlivých přijatých volání *plus* a *minus* a ukončí svou činnost.

Klientská část této aplikace spočívá v tom, že je proveden přesně definovaný počet volání entry *plus* a *minus*. Nyní jsou počty nastaveny na provedení 100 volání *plus* a 50 *minus*. Klientské vlákno si na začátku zjistí *Guid* zaregistrovaných serverů pro obě *entry* a pak provádí zadaný počet volání v náhodném pořadí na náhodně vybraném serveru. Na konci provede výpis, zda získaná hodnota odpovídá předpokládané hodnotě – známe počet obou operací a při počáteční hodnotě 0 je předpokládaná hodnota rozdílem počtů obou volání. Poté klient svou činnost ukončí. Hlavní vlákno čeká na ukončení činnosti všech klientských vláken a poté provede volání stop na všech spuštěných serverových vláknech.

Protože SAN interpret nepodporuje činnost vláken, musela být vlákna nahrazena aplikacemi. Hlavní vlákno tvoří aplikace spuštěná z příkazové řádky. Ta spustí další instance sebe sama a podle parametrů příkazové řádky určí úkol spouštěné aplikace, zda bude provádět činnost serveru nebo činnost vlákna. První je nutné vytvořit serverové aplikace a teprve poté klientské. Poté hlavní aplikace čeká na dokončení činnosti vláknových aplikací. K tomuto účelu máme již vytvořenou metodu *waitForApplication*. Aplikaci, na kterou chceme čekat, určíme pomocí jejího *Guid*, které nám vrátí metoda jádra, která provádí její spuštění. Použitím této metody tak vlastně nahrazujeme vláknovou metodu *join*. Na stejném principu čekání na

spuštěnou aplikaci funguje také aplikace Shell. Po dokončení činnosti všech vláken pak zavolá *entry* stop a opět čeká na dokončení serverových aplikací. Poté končí svou činnost.

Protože klientské aplikaci si pomocí definovaných metod z ICommonAPI samy zjišťují seznam *Guid* vytvořených serverů, je nutné je spouštět až poté, co všechny serverové aplikace provedou zaregistrování svého serveru a všech 3 svých *entry*. K tomuto účelu si hlavní aplikace vytváří svůj vlastní server s *entry* pojmenovaným *readyForAccep*t. Při spouštění serverové aplikace jako jeden z parametrů předá *Guid* svého serveru. Po vytvoření všech serverových aplikací provede stejný počet akceptů svého entry. To volají serverové aplikace po dokončení svých registrací. Toto *entry* si nepředává žádná data – vstupní parametr i návratová hodnota jsou null – a také tělo akceptu neprovádí žádnou užitečnou činnost. Na tomto entry slouží tedy jen k vyčkání na to, až všechny vytvořené servery zaregistrují své vlastní entry. Toto dokumentuje další možnost využití rendez-vous, a tím je pouhá synchronizace dvou vláken.

Tato aplikace testuje, zda správně funguje konkurenční volání více vláken na více serverů při předem neznámém pořadí a cíli volání a zda nedochází k deadlocku žádného ze tří úkolů aplikace. Také testuje, zda správně funguje volání entry oběma způsoby adresace. Zatímco klientské aplikace volají entry *plus* a *minus* pomocí ID jednotlivých entry, serverové aplikace volají *entry* hlavní aplikace pomocí *Guid* a názvu *entry*.

5.9 Ukázkový program rendezvous2

Tento aktivní program provádí synchronizaci rendez-vous na více aktivních uzlech a její použití je trochu složitější. Aplikace se snaží pomocí kapsulového programu zjistit, zda na zadaném uzlu běží rendez-vous server a pokusí se provést rendez-vous na jeho entry a vyměnit si data, která poté kapsulí odesílá zpět. Základem této je aplikace ping.

Nejdříve je nutné na jednom z uzlů v síti spustit serverovou část rendez-vous. To je možné provést spuštěním aplikace s parametry

„-s <libovolný řetězec>“.

Příkladem příkazu je:

run rendezvous2 -s nejaky testovaci string

Po spuštění aplikace je vytvořen rendez-vous server s dopředu známým *Guid* a jedním *entry* pojmenovaným *QueryEntryCall*. V případě neúspěchu – například při obsazeném *Guid* – je vypsána chybová zpráva a aplikace skončí. V opačném případě aplikace čeká na příchozí volání. Tělo akceptu pouze vrací zadaný řetězec (tajenku) z příkazové řádky.

Po spuštění serveru je na jiném uzlu nutné spustit klientskou aplikaci a v parametru jí předat adresu uzlu, na kterém je spuštěn server. Protože základem je aplikace ping, bylo využito jejího způsobu zadání cílové adresy, proto parametry pro spuštění klientské aplikace jsou zapsány v tomto pořadí:

run rendezvous2 -c <číslo uzlu> <číslo sítě>

Klientská aplikace vytvoří datovou kapsuli, nastaví listener pro příchozí data a kapsuli odešle na zadanou adresu. Pak provádí čekání (aktivní) na příchozí kapsuli. Na zadaném uzlu je spuštěn kapsulový program, který se pokusí zavolat entry s dopředu definovaným *Guid* a názvem *entry*.

Jestliže na daném uzlu běží serverová aplikace, je pomocí provedeného rendez-vous předána zadaná tajenka kapsulovému programu. Poté je serverová část aplikace ukončena. Pokud na daném uzlu server s daným Guid neběží nebo nemá registrované entry daného jména, je při volání vrácena hodnota null. V obou případech jsou zjištěná data uložena do kapsule a odeslána zpět na uzel s klientskou aplikací. Zde jádro zavolá listener, který načte data z kapsule a buď vypíše chybové hlášení, nebo získanou tajenku. Po tomto výpise je ukončena také klientská aplikace.

Tato aplikace testuje, zda je možné provést synchronizaci rendez-vous také mezi aplikací a kapsulí. Také dokumentuje možnost zaregistrovat si rendez-vous server s dopředu daným identifikátorem *Guid*.

6 Problémy při implementaci synchronizace

Jak bylo již řečeno, při implementaci rendez-vous do projektu SAN byla nejdříve provedena implementace již vytvořeného řešení pro čistou Javu a ukázkové aplikace rendezvous. Toto řešení bylo nejprve vyzkoušeno pomocí interpretu BeanShell a bylo funkční. Protože BeanShell je v projektu SAN pouze z historických důvodů, byl hlavní požadavek, aby synchronizace fungovala také v interpretu SAN. Změna interpretu byla provedena jiným typem překladu, a protože BeanShell neumožňuje více tříd ve více souborech a testovací programy byly rozsáhlejší, bylo provedeno také rozdělení aktivních programů do více tříd. V tomto okamžiku se objevilo mnoho různých chyb, které bylo nutno odstranit.

6.1 Nepodporování polí interpretem

První chybou, která se objevila, byla nemožnost spouštění aplikací z aplikace. Ačkoliv program využíval přesnou kopii příkazu, jaký používá aplikace Shell, program nefungoval. Program končil s chybovou zprávou, že nebyla nalezena volaná metoda. Teprve po důkladném prozkoumání problematiky interpretu a volání soft metod jsem zjistil, že chybí podpora pro předání pole objektů. Tento typ parametru se vyskytuje právě u metody pro spuštění aplikace. Pole objektů je v tomto případě pole String, které předává parametry příkazové řádky. Tuto metodu v době vývoje využívaly pouze aplikace Init a Shell. Obě tyto aplikace však byly spouštěny interpretem BeanShell a tak problém nebyl odhalen.

Při vytváření podpory tohoto typu parametrů bylo nejprve nutné zjistit, jakým způsobem jsou metody hledány a volány. Každá metoda má kromě svého názvu také definovány své parametry a návratovou hodnotu, které jsou zapsány ve zkrácené formě. Základní datové typy jsou označené písmeny I - int, F - float, D - double, J - long, Z - boolean, B – byte ,C – char, S – short, V - void. Objekt je označen písmenem L, za kterým je uveden plný název objektu v lomítkové notaci a ukončen středníkem. Kterýkoliv parametr může mít před sebou znak „[“ jenž značí, že se jedná o pole tohoto typu. Metoda, kterou ve zdrojovém kódu zapíšeme jako:

long metoda(int a, String[] b, byte c);
bude zapsána takto: *metoda(I[Ljava/lang/String;B)J*.

Při volání soft metody parsujeme tento řetězec a podle parametrů vytváříme javovské objekty. V případě pole objektů můžeme využít konstrukce Class.forName() a v parametru uvést definici v tomto zkráceném tvaru, pro jeden parametr. Jedinou úpravou je převedení definice

názvu třídy na tečkovou notaci. Při zpracování návratového typu musíme pokračovat obráceně, zapsaný typ převést na některou třídu interpretu.

6.2 Podpora interpretu pro více tříd

Druhý větší problém, který se vyskytl, bylo používání více tříd uvnitř jednoho balíčku aplikace. Při zápisu zdrojového kódu máme aktivní program tvořen jako jeden balík (package) a tak se třídy navzájem vidí bez nutnosti importu. Protože jsou však tyto třídy uložené v nestandardně upraveném jar balíčku a uložené mimo classpath, tak je standardní class loader nenajde. Tato chyba se pak projevuje nejen při převodu z hard reference na soft referenci, ale i při samotném využití více tříd uvnitř programu. Interpret SAN tuto skutečnost nahlásí, že nenalezl příslušnou třídu v okamžiku jejího prvního použití (např. při vytváření její instance). Tento problém byl vyřešen implementací vlastního class loaderu, který jako první prohledává systémové balíky a balíky v classpath (nejčastěji načítáme standardní objekty jako String apod.) a poté se pokouší prohledat balíček spouštěné aplikace. Z důvodů optimalizace jsou tyto informace v interpretu cachovány v hashmapě.

6.3 Vyhledávání volaných metod

Třetí závažnou chybou, která se vyskytla, bylo nesprávné vyhledávání metod, které se měly volat uvnitř hard třídy. Při interpretování hard třídy je volaná metoda určena indexem. Tento index je nutné převést na definici metody a teprve poté je možné určit, zda budeme volat metodu jako soft či hard, zda metoda je abstraktní či statická apod. Jestliže je metoda definována uvnitř třídy, na jejíž instanci je volána, hledání probíhá bez problémů. Jakmile však třída implementuje rozhraní, je nutné prohledat i tato rozhraní. Problém nastal u zavedeného rozhraní ICommonAPI. Při volání metod pro klientskou stranu rendez-vous voláme metody z objektu, který implementuje buď IApplicationAPI či ICapsuleAPI. A v žádném z těchto rozhraní daná metoda nalezena není, protože obě rozhraní rozšiřují ICommonAPI. Řešením bylo zavedení rekurzivního prohledání také všech rodičů nalezených rozhraní.

6.4 Převádění referencí u nativních funkcí

Asi časově nejnáročnější byla oprava chyby, která se objevila při skutečném provádění rendez-vous v první ukázkové aplikaci. Klientská aplikace měla provádět volání náhodně vybraného *entry* na náhodně vybraném serveru. Pro výběr byl použit standardní generátor

náhodných čísel, třída java.util.Random. Tato třída má několik metod, které vracejí náhodnou hodnotu podle příslušné metody. Například opakované volání metody *nextInt()* vrací řadu náhodných čísel typu *int*. Při použití této třídy byla při každém spuštění programu vygenerována nová náhodná hodnota, ale při opakovaném volání byla vracena stále tatáž hodnota. K dohledání chyby bylo nutno nejprve detailně prozkoumat činnost tohoto generátoru náhodných čísel a poté detailně prozkoumat chování interpretu při tomto generování.

Počítač jako deterministický stroj sám o sobě vytvořit náhodnou hodnotu nedokáže a tak je tento proces simulován tzv. pseudonáhodnou hodnotou. Princip spočívá v tom, že vezmeme nějakou počáteční hodnotu (většinou vychází ze systémového času) a tuto hodnotu použijeme jako vstup do nějaké speciálně vytvořené funkce. Nově vzniklá hodnota je pak znovu použita atd. Z těchto hodnot pak třída Random vytváří požadované náhodné hodnoty. Z tohoto principu vyplývalo, že z nějakého důvodu je používána stále výchozí hodnota a není ukládána nově vypočítaná.

Při prohledávání třídy Random a postupném ladění její činnosti v SAN interpretu jsem se nakonec postupně dostal až k poměrně hluboko ukryté metodě, která při zavolání nativní javovské funkce měla provést aktualizaci předaného objektu. Pro vykonání nativní funkce je volán JVM, který volá nějakou konkrétní implementaci na dané platformě. Tato metoda byla skutečně volána a při běžném procházení programu vypadalo vše v pořádku. Po jejím otevření jsem však zjistil, že její tělo je prázdné, resp. obsahuje příkaz pro vygenerování výjimky, která měla zahlásit, že daná funkcionalita zatím není implementována. Zřejmě některý z předchůdců byl touto chybou obtěžován a tak tento příkaz zakomentoval. Program tedy bez chybového výpisu pokračoval dál a docházelo ke zmíněnému chování.

6.5 Problém převodů hard a soft referencí

Zásadní problém vznikl při vytváření rendez-vous synchronizace. Problém vznikl při předávání hard reference soft třídě. Tento problém byl již popsán v kapitole 4.12 u problematiky registrace listeneru pro příchozí data. Při registraci listeneru byl nejprve použit operátor *this*, později byl zaměněn za nově vytvořený *Guid*. Tento problém nastává také u rendez-vous serveru. Při registraci serveru je uložena reference instance třídy implementující rozhraní IRendezvousServer. Na objektu serverové třídy je při úspěšném akceptu volána metoda *entry*. Jakmile však předáváme referenci této hard třídy metodě z ApplicationAPI, je převedena pomocí reflexe převedena na klasický javovský objekt

a pracuje s ní JVM. Jakmile pak provedeme nějakou manipulaci s daty uvnitř této třídy, nebudou tyto změny dostupné v původní hard referenci uvnitř interpretu. V praxi se tato situace projevila tím, že při akceptu volání *stop*, byla změněna hodnota booleanové proměnné *running*, která je v serverové třídě, na hodnotu *false* a po provedení tohoto akceptu serverový program skončil. Tato změna vnitřní proměnné se však provedla pouze ve vytvořeném objektu, ale nedostala se do hard reference. Bylo tedy nutné použít podobný způsob, jaký používá jádro při volání listeneru příchozích dat.

V listeneru příchozích dat předáváme Guid aplikace, která chce naslouchat. V tomto případě se předpokládá, že listener bude implementován v hlavní třídě aplikace. V případě rendez-vous toto zaručit nelze, rozhraní může implementovat jakákoliv třída. Proto je nutné provést volání na jakémkoliv objektu. Abychom mohli toto volání zajistit, je potřeba identifikovat spuštěnou aplikaci a referenci objektu. Identifikaci reference však nemůžeme zjistit uvnitř programu, zde tato reference pro nás představuje operátor *this*, ani uvnitř synchronizačního prostředku, protože ten již pomocí soft class běží v JVM a tam je reference také existující objekt. Jediná úroveň, kde můžeme identifikátor reference zjistit je během převodu z hard reference na javovský objekt. Při převodu potřebujeme však volat metodu se stejnými parametry, která byla volána z hard class, a tato metoda očekává objekt s rozhraním IRendezvousServer. Proto byla vytvořena pomocná třída, která toto rozhraní implementuje, ale která má také potřebné metody a proměnné k uložení potřebných informací. Jakmile je zjištěna nutnost překladu objektu tohoto rozhraní, je objekt převeden na tuto třídu a původní objekt je do ní uložen. V případě, že potřebujeme s tímto objektem pouze pracovat v rámci JVM, bude metoda entry stále funkční, neboť volá původně převáděný objekt. Navíc však uchovává také *Guid* spuštěné aplikace a ID reference. Při volání metody *entry* při akceptu uvnitř řídícího rendez-vous objektu můžeme původní volání:

```
server.entry(name, params);
```

nahradit vytvořením nového interpretu, kterému je předán *Guid* aplikace, id reference a název metody *entry* s příslušnými parametry. Tím je provedeno *entry* na téže referenci, jako která byla původně zaregistrována a program funguje správně. Navíc konkrétní činnost synchronizace, která využívá prostředky vláken je prováděna přímo v JVM.

6.6 Vyhledávání správné metody

Jedna z méně závažných chyb se objevila při vytvoření ladícího výpisu na třídě *Guid*. Každá třída má k dispozici metodu *toString*, kterou si může v případě potřeby překrýt svou vlastní. Při interpretaci hard třídy a volání některé metody také z hard třídy je před spuštěním této metody připraven buffer, který má metoda k dispozici na vstupní a výstupní parametry a své vnitřní proměnné, kterých je předem známý počet. V původním programu se tato informace zjistila při vyhledání definice požadované metody. V případě metody *toString* se tak informace o počtu vnitřních proměnných zjistila u nejvyššího rodiče, což byl Object. Počet vnitřních proměnných metody tedy neodpovídal skutečně volané metodě, vytvářený buffer byl příliš malý a program končil na výjimce NullPointerException.

7 Bezpečnostní problémy

Vytvořený prostředek rendez-vous v projektu SAN má několik bezpečnostních rizik, které by bylo před ostrým nasazením vhodné ošetřit. Prvním rizikem je, že má server registrován vlastní *Guid*, kterým se při volání řídících metod identifikuje. Pokud nebudeme předávané *Guid* nijak kontrolovat, nic nebrání tomu, aby jakýkoliv program odchytával volání určená cizímu serveru. Stačí zjistit *Guid* některého registrovaného serveru a s tímto *Guid* zavolat metodu *makeAccept* a je provedené volání odchyceno. Možností by bylo důsledně kontrolovat také *Guid* aplikace a tak toto chování neumožnit.

Dalším problémem je nekontrolované ukončení činnosti serveru. Jestliže server z nějakého důvodu ukončí předčasně činnost nebo záměrně neprovede odregistrování, pak může docházet k uvíznutí klientských programů. Jediné použitelné řešení by bylo po ukončení činnosti aplikace zkontrolovat na systémové úrovni aktivního uzlu a případně provést odregistraci serverů, které zůstaly zaregistrované.

Dalším krokem k posílení bezpečnosti je doimplementovat systém výjimek, které zvýší bezpečnost a použitelnost prostředku. Například by bylo vhodné ošetřit situaci volání neregistrovaného entry, nebo oznámení, že volání nebylo akceptováno, protože server končí. V takovém případě je nyní vrácena pouze hodnota null, která ovšem může být normálním výstupem, což programátor nemůže nijak ošetřit.

8 Závěr

Tato práce si kladla za cíl vytvořit prostředek pro synchronizaci vláken typu rendez-vous, který je znám z Ady, tento prostředek implementovat v prostředí Javy a poté v projektu SAN. Ačkoliv je projekt SAN naprogramován v Javě, jeho vlastní interpret neumožňuje pouhé integrování do projektu, ale musely být provedeny potřebné úpravy. Některé úpravy byly provedeny na základě požadavků vedoucího projektu, pana ing. Koutného. Všechny požadované úkoly se podařilo splnit v čistém prostředí Javy, kde vznikl efektivní prostředek pro daný typ synchronizace, který podporuje většinu funkcionalit podle předlohy z jazyka Ada. V projektu SAN k některým implementacím nedošlo, ale je možné je později doplnit.

8.1 Vlastní přínos

Vytvořený prostředek nemusí sloužit pouze k výměně dat či prosté synchronizaci vláken či aplikací, ale mohl by být jeho princip použit také při vylepšení stávajícího projektu SAN. Současný projekt nemá podporu vláken uvnitř jedné aplikace a například pro čekání na příchozí kapsule lze použít pouze aktivní čekání, což je velmi neefektivní. Stačilo by ovšem čekání na příchozí kapsuli implementovat jako čekání na klientské volání a samotnou událost příchodu implementovat jako toto volání, ihned máme realizované uspání vlákna aplikace. Podobných užití bychom mohli najít více.

Při ostrém nasazení by bylo nutné ještě vyřešit některé bezpečností problémy, implementovat ošetřování některých situací pomocí systému výjimek a rozšířit možnosti vytvořeného prostředku v projektu SAN o další funkcionality. I přes některé nedostatky má však projekt SAN velmi silný nástroj pro různá budoucí využití.

Přehled pojmů a zkratek

OOP	Objektově Orientované Programování
SAN	projekt Smart Active Node
JVM	Java Virtual Machine
FIFO	Abstraktní datový typ fronta, se kterou pracujeme ve stylu First In, First Out

Entry	Synchronizační bod v programu při rendez-vous
Call	Provedení volání rendez-vous z klientského vlákna
Accept	Pokus o akceptování příchozího volání (call) serverovým vláknem

Literatura k tématu

Diplomové práce

[d1] : ŠTĚPÁNEK Petr: Code Distribution in Active Networks, ZČU KIV 2009

[d2] : REJDA, Michal: Smart Active Node, ZČU KIV 2008

[d3] : SYROVÁTKA, Jakub: Code Interpreter for Smart Active Node, ZČU KIV 2009

Elektronické zdroje

[1] : Duarte, Joao Carlos Mendes Nunes a Racek, Stanislav. Using a simple rendezvous mechanism in Java. In Proceedings of 6th International carpathian control conference. Miskolc : University of Miskolc, 2005, s. 173-178.].
Dostupné na WWW :
<http://www.kiv.zcu.cz/research/groups/dss/download/presentation-2005-03-21.ppt>

[2] : MIRANDA, Javier. *The Ada Rendezvous* [online]. Canary Islands : 2002. Version 1.0 [cit. 17.3.2010]. Dostupné na WWW :
<http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/node21.htm>

[3] : PAŠKA, Marek. Copaté programování – Root.cz. Dostupné na WWW :
<http://www.root.cz/clanky/copate-programovani/>

[4] : PAŠKA, Marek. Bezpečné programování ala Ada – Root.cz. Dostupné na WWW :
< http://www.root.cz/clanky/bezpecne-programovani-ala-ada/>

[5] : KOUTNÝ, Tomáš. Přednášky PPR. Kapitoly 4, 5. Dostupné na WWW :
<http://www.kiv.zcu.cz/~txkoutny/download/ppr/prednasky/>

[6] : Ada Programming / Tasking
<http://en.wikibooks.org/wiki/Ada_Programming/Tasking>

[7] : Ada programming language
<http://en.wikipedia.org/wiki/Ada_Programming_language>

Knižní tituly

[8] : AUSNIT-HOOD, Christine ... [et al.] (ed.). *Ada 95 quality and style : guidelines for professional programmers*. Berlin : Springer, 1997. 14, 292 s. ISBN 3-540-63823-7.

[9] : HEROUT, Pavel. *Java a XML*. 1. vyd. České Budějovice : Kopp, 2007. 313 s. ISBN 978-80-7232-307-4.

[10] : BURNS, Alan a WELLINGS, Andrew. *Concurrent and real-time programming in Ada 2005*. Cambridge : Cambridge University Press, 2007. ISBN 9780521866972. 14, 461 s.

[11] : NAIDITCH, David. *Rendezvous with Ada : A programmer's introduction*. New York : Wiley, 1989. 16, 477 s.

Příloha A – uživatelská dokumentace

V této dokumentaci je popsána aplikace aktivního programu rendezvous. Je na ní vysvětlena tvorba serverového vlákna.

```
/* hlavní metoda serverového vlákna */
public void run(IApplicationAPI applicationAPI, PrintWriter out, Guid
guid_main) {
    /* nejdříve je nutné si svou třídu zaregistrovat jako server
       tím dostaneme Guid přidělené tomuto serveru
     */
    Guid guid = applicationAPI.registerServer(this);

    /* nyní si zaregistrujeme jednotlivé entry s přiděleným Guid */
    applicationAPI.registerEntryCall(guid, "plus");
    applicationAPI.registerEntryCall(guid, "minus");
    applicationAPI.registerEntryCall(guid, "stop");

    try {
        // provedeme rendez-vous jako klient, voláme entry hlavního vlákna
        // dáváme tak zprávu hlavnímu vláknu, že server je připraven
        applicationAPI.callEntryCallByName(guid_main,
                              RendezVous.ENTRY_CALL, null);
    } catch (InterruptedException ex) { return; }

    // nyní provádíme cyklus až do doby, než bude akceptováno volání stop
    while (running) {
        try {
            // akceptujeme jakákoliv volání na všech entry pod naším Guid
            applicationAPI.makeAccept(guid);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
    // na konci činnosti je důležité odregistrovat náš server a naše entry
    applicationAPI.unregisterServer(guid);
}

// tato metoda je volána a provádí činnost těla akceptu pro všechna entry
public Object entry(String name, Object param) {
    /* parametry od klienta dostáváme jako objekt, přetypování si musíme
zajistit sami */
    Integer number = (Integer)param;
    /* postupně prozkoumáme, na kterém entry bylo akceptováno volání */
    if (name.equals("plus")) {
        number = number.intValue()+1;
    }
    if (name.equals("minus")) {
        number = number.intValue()-1;
    }
    if (name.equals("stop")) {
        running = false;
        return null;
    }

    /* návratová hodnota, která bude předána klientskému vláknu */
    return number;
}
```

Nyní následuje popis činnosti klientského vlákna ve stejné aplikaci. Termín vlákno je v tomto případě trochu nepřesný, vlákno v projektu SAN je samostatně spuštěná aplikace. Klientské vlákno (aplikace) provádí pouze volání jednoho ze dvou entry, na některém ze zjištěných serverů podle jména entry. Ačkoliv bychom mohli využít stejného systému volání, je z testovacích i názorných důvodů využito volání pomocí ID entry.

```
/* hlavní metoda klientského vlákna */
public void run(IApplicationAPI applicationAPI) {
    /* zjistíme si seznam ID entry, která jsou zaregistrovaná pod jménem*/
    Integer[] plusyArr = getCallsID(applicationAPI, "plus");
    Integer[] minusyArr = getCallsID(applicationAPI, "minus");
    int cnt=plus+minus;

    /* provádíme určený počet kroků */
    for (int i=0; i<cnt; i++) {
        /* náhodně vybereme, které entry budeme volat */
        boolean spustPlus;
        do {
            spustPlus = random.nextBoolean();
        } while (spustPlus && plus==0 || !spustPlus && minus==0);
        if (spustPlus) plus--; else minus--;

        try {
            /* provedeme volání na náhodně vybraném serveru */
            Object obj = applicationAPI.callEntryCallByID(
                   getRandomID(spustPlus ? plusyArr : minusyArr),
                   new Integer(val));
            val = (Integer)obj; // uložíme návratovou hodnotu
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

/* metoda vrátí seznam ID registrovaných entry, podle jména entry */
private Integer[] getCallsID(IApplicationAPI applicationAPI, String
entryName) {
    /* nejprve zjistíme Guid serverů, které mají daná entry registrované */
    Guid[] guids = applicationAPI.getServerGuidsByName(entryName);
    Integer[] pole = new Integer[guids.length];
    /* provedeme převod z Guid a jména na ID entry */
    for (int i=0; i<guids.length; i++)
        pole[i] = applicationAPI.getEntryCallIdByGuidName(
                        guids[i], entryName);
    return pole;
}

/* metoda náhodně vybere jedno entry ze seznamu */
private Integer getRandomID(Integer[] pole) {
    int p = random.nextInt(pole.length);
    return pole[p];
}
```