Prohlašuji, že jsem disertační práci vypracoval samostatně a pouze s využitím citovaných pramenů.

V Plzni dne 29. října 2001 Ing. Martin Šimek

Anotace

Cílem disertační práce je návrh objektově orientovaného systému pro podporu mobility agentů v prostředí Internetu. Navržený systém mobilních agentů se skládá z několika vrstev, které jsou navzájem striktně odděleny, což dovoluje přesně definovat úrovně bezpečnostní politiky. Jednotlivé komponenty systému pak řeší identifikaci a adresování objektů, přístup ke službám na uzlech, autentifikaci a autorizaci objektů a komunikaci mezi objekty. Hlavním bodem práce je návrh systému, jehož programová realizace v programovacím jazyce Java byla ověřena na typických aplikacích mobilních agentů. Z výsledků vyplývá, že navržený systém má poněkud větší vlastní režii než odpovídající systémy, což souvisí s navrženou strukturou, kdy se při migraci vytváří nové objekty nezbytné pro běh systému.

Annotation

Main purpose of the thesis is design of object-oriented mobile agent system in the Internet environment. The designed system is composed of a few strictly separated layers. This structure allows define level of secure policy accurately. Components of the system perform object identification and addressing, access to services in the network nodes, object authentification and authorization and communication among objects. A proxy-based access control mechanism is used for secure access to services. The Java language with standard extension packages is used for implementation of proposed system. Finally, the implementation is verified by typical applications of mobile agents. Results are compared with existing Java-based mobile agent system-Aglets. Designed system has higher overhead than comparable system because it has more complex structure and it is necessary to create new objects during the migration.

Annotation

Die vorgelegte Dissertation beschäftigt sich mit Entwurf und Entwicklung des objekt-orientiertes Unterstützungssoftwaresystem mit dem Ziel, die Mobilität von Softwareagenten in der Internet–Umgebung zu unterstützen. Das entwickelte System von mobilen Agenten besteht aus mehreren Applikationsschichten, die untereinander scharf getrennt sind und diese Trennung eine eindeutige Bestimmung von Sicherheitsebenen des entwickelten Systems ermöglicht. Einzelne Systemkomponenten gewährleisten die Identifizierung und Adressierung von Objekten, Zugriff zu den von Netzknoten angebotenen Systemdiensten, Authentifizierung und Authorisierung von Objekten und zuverlässige Kommunikation zwischen den Objekten. Entwurf und Implementation des oben beschriebenen Unterstützungssystems kann als Schwerpunkt der vorgelegten Dissertation erkannt werden. Die Implementation wurde in der Programmiersprache Java realisiert und ihre Funktionsmerkmale wurden an der typischen Aufgabe der Anwendung von mobilen Agenten demonstriert. Aus den Ergebnissen von ausgeführten Experimenten folgt es, daß das entwickelte System weisst ein bißchen höhere Werwaltungskosten als die vergleichbaren existierenden Systeme auf; diese Erhöhung hängt mit der entwickelten Systemstruktur zusammen, da bei der Migration die neuen, für den Systemlauf notwendigen Objekte erzeugt werden sollen.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Kapitola 1

Úvod

V této kapitole budou vysvětleny základní termíny používané v této práci jako např. pojem *agent* a *mobilní agent*. Agenti budou rozděleni do tříd a budou popsány výhody jejich použití. V závěru kapitoly budou uvedeny příklady aplikací, které mohou těžit z výhod mobilních agentů.

Zájem o síťové aplikace a jejich programování v poslední době vzrůstá spolu s exponenciálním nárůstem uživatelů Internetu. Internet se osvědčil jako platforma pro šíření informací, elektronický obchod a zábavu. Růst popularity Javy jako programovacího jazyka, který podporuje přenos kódu po síti, je další hnací silou tohoto vývoje. Použití internetových technologií v privátních sítích (např. vytváření *intranetů* nebo *extranetů*) přináší vyšší požadavky na síťové aplikace. Jako odezva na tento trend se neustále vyvíjí nové techniky, jazyky a přístupy usnadňující jejich tvorbu. Asi nejslibnější mezi novými formami je použití *mobilních agentů*. V této kapitole budou popsány základní termíny týkající se agentů a bude naznačen jejich historický vývoj. Při popisu bude brán zřetel na jejich využití v některých typech aplikací, ale také nevýhody jejich použití v otevřených sítích jako např. Internetu.

1.1 Použitá terminologie

V oblasti mobilních agentů (a agentů vůbec) není zcela ustálena terminologie. V této práci jsou používány pojmy:

agent: Pojmem agent je většinou označen funkční program (tj. v operační paměti). Někdy je však použit místo plného pojmu *stav agenta*.

systém (*framework*): Místo plného pojmu *systém mobilních agentů* je používána tato zkrácenou verze. Vědomě se tím sice riskuje, že může dojít k záměně za operační systém počítače, avšak v tomto dokumentu se pojem operační systém vyskytuje jen zřídka a na těchto místech je použito jeho plné označení.

síť (*network*): Pojmem síť je vždy míněna počítačová síť, ve které je systém provozován. Předpokládá se síť typu Internet (protokoly rodiny TCP/IP).

uzel (*node*), počítač (*computer*): Oba dva tyto pojmy označují totéž: počítač, zapojený do počítačové sítě.

hostitel (*host*): Pojmem hostitel je míněno prostředí (součást systému) na určitém uzlu, ve kterém agenti fungují. Může označovat uzel, stroj, nebo jenom jeho část. Ve většině případů se pojmem hostitel míní souhrn uzlu a stroje.

stroj (*engine*): Pojmem stroj je označen jeden celý program, který vykonává funkci hostitele. Význam má především v případě jazyka Java, kdy označuje také JVM (*Java Virtual Machine*), ve které program běží.

programový kód (*code*): Spustitelný kód programu, konkrétně agenta.

stav (*state*): Pokud je tento termín použit bez přívlastku, míní se tím stav výpočtu (*execution state*).

uživatel (*user*): Osoba (člověk) využívající služeb systému (jakýmkoliv způsobem).

vlastník (*owner*): Osoba (člověk), která agenta vlastní. Agent svého vlastníka zastupuje a prokazuje (autorizuje) se jeho identitou.

klient (*client*): Program (v některých případech i počítač, na kterém běží), který agenta vytváří, komunikuje s ním a případně ho opět přijímá. Klient je typicky obsluhován (nebo alespoň provozován) vlastníkem agenta.

tvůrce (*creator*): Entita, která agenta vytvořila. Většinou se jedná o klienta, nicméně to nemusí být pravidlem.

migrace (*migration*): Proces, kdy je agent přenesen z jednoho hostitele na jiného.

1.2 Co je to agent?

Existují nejméně dva důvody, proč je těžké přesně definovat tento pojem. První je ten, že každý, kdo využívá agenty, definuje tento termín jiným způsobem, neboť využívá jejich různých vlastností. Druhým důvodem je používání termínu **agent** jako zastřešujícího pojmu pro různé druhy výzkumu. Každý vývojový tým pak definuje tento pojem jiným způsobem a tím jen zvětšuje zmatek. V současné době existuje mnoho synonym pro původní termín **agent**: *knowbot (knowledge-based robot)*, *softbot (software robot)*, *taskbot (task-based robot)*, *userbot*, *robot*, *personal agent*, *autonomous agent* atd. Existuje několik důvodů pro jejich existenci.

Agenti totiž mohou pracovat v různém prostředí, simulovat chování reálného světa, instalovat software na vyžádání atd. Proto se většinou před termín *agent* vkládá ještě označení oblasti jeho použití. Tímto způsobem pak vznikly termíny *search agent*, *report agent*, *navigation agent*, *search agent*, *help agent* a mnoho dalších [Kin1995].

Je zřejmé, že pojem **agent** není možné definovat jednoduchým způsobem. Dále bude ukázáno, jak se dá použití tohoto termínu poměrně dobře vymezit a tím zároveň vytvořit různé skupiny agentů.

1.3 Vlastnosti softwarových agentů

Samostatnost (*Autonomy*): Agent pracuje bez přispění lidí nebo jiných subjektů a kontroluje své akce a vnitřní stav. Autonomní agent je systém uvnitř prostředí, které vnímá a reaguje na jeho změny na základě svého vlastního programu [Fra1996]. Příkladem autonomních agentů jsou viry a červi (*worms*).

Komunikativnost (*Communication*): Jednou ze základních vlastností agentů je schopnost komunikace s ostatními agenty nebo s uživateli (viz kapitola 1.4.2). Pro komunikaci mezi agenty se používají:

- Tabule (*Blackboard*): Agenti zapisují a čtou zprávy ze sdíleného vyhrazeného místa.
- KQML (*Knowledge Query and Manipulation Language*): Je jazyk a protokol pro popis výměny informací a znalostí mezi agenty používající **soupisy akcí** (*performatives*).
- KIF (*Knowledge Interface Format*)
- COOL: Strukturovaná konverzace mezi agenty založená na KQML. Používá se pro koordinaci agentů [Bar1996].

Spolupráce (*Collaboration/Cooperation*): Agenti jsou schopni spolupráce, jestliže dokáží pracovat společně s ostatními. Agent musí být schopen komunikace s ostatními, samostatně uvažovat a koordinovat své akce podle ostatních. Spolupracující agenti se dají použít, pokud daná úloha vyžaduje více samostatných systémů.

Samostatné uvažování (*Deliberation*): Agenti odvozují své reakce od vnitřního symbolického modelu výpočtu [Woo1995].

Mobilita (*Mobility*): Tato vlastnost popisuje schopnost agenta migrovat mezi systémy. Existuje několik různých druhů mobility:

- Mobilita umožňující migraci agenta mezi stejnými systémy.
- Mobilita umožňující migraci agenta mezi různými systémy.
- Mobilita umožňující agentovi přerušit jeho práci na jednom systému, přesunout jej na jiný systém a pokračovat.
- Mobilita umožňující agentovi přesunovat sám sebe, místo aby *byl přesunut* systémem.
- Mobilita, která je vytvoří duplikát agenta na jiném systému (*cloning*).
- Agent přenáší své znalosti na jiný systém.

Obecně je mobilita kombinací těchto kategorií.

Učení se (*Learning*): Existují dvě definice schopnosti učit se:

- Agent je schopen učit se, jestliže dokáže získávat znalosti (data).
- Agent je schopen učit se, jestliže je schopen tyto znalosti využívat k přizpůsobování svého chování.

Navzdory tomu, že schopnost učit se je důležitou součástí inteligence, mnoho agentů tuto vlastnost nemá. Většina agentů vlastní pouze pevně daná pravidla, podle kterých se řídí.

Vlastní aktivita (*Pro-activeness*): Tato vlastnost označuje schopnost agenta vytvářet změny prostředí místo prosté reakce na změny okolí. Většinou se tato vlastnost zaměňuje se schopností vlastního uvažování.

Reaktivita (*Reactivity*): Agent vnímá své okolí a včas reaguje na jeho změny.

Bezpečnost (*Security*): Ochrana agenta před nebezpečnými elementy.

Plánování (*Planning*): Tato vlastnost zohledňuje schopnost samostatně uvažovat a vytvářet vlastní aktivity podle svých znalostí.

Delegace (*Delegation*): Agent může požádat někoho jiného, aby vykonal část jeho úkolu. Tato vlastnost je velmi důležitá pro úlohy zaměřené na vyrovnávání zátěže.

1.4 Typy agentů

Prvním hlediskem, podle kterého se dají agenti dělit, je jejich vnitřní sémantická stavba. Agenti se pak dělí na:

- **uvažující** (*deliberative*) – agenti odvozují své reakce od vnitřního symbolického modelu výpočtu,

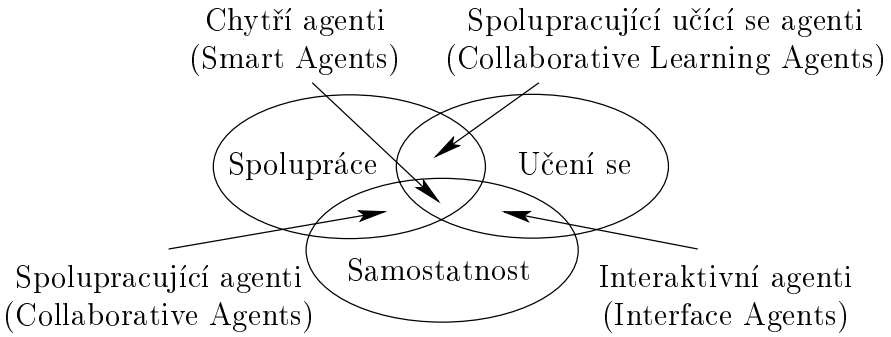
- **reaktivní** (*reactive*) – agenti nemají žádný vnitřní model výpočtu a pouze reagují na požadavky od systému nebo na jeho změny [Bro1986, Agr1987]. Přestože by se mohlo zdát, že reaktivní agent nemůže mít vlastní inteligenci, v [Bro1991] je dokázáno, že inteligentního chování agentů se dá dosáhnout i bez explicitního symbolického zápisu jeho vnitřní struktury.

Dalším hlediskem, podle kterého se agenti dají rozdělit, je jejich *mobilita*, tj. schopnost migrovat po síti. Agenti se pak dělí na **stacionární** (*stationary agents*) a **mobilní** (*mobile agents*).

Agenti mohou být také děleni podle charakteristických vlastností. Existují tři nejdůležitější vlastnosti popisující jejich chování: **samostatnost** (*autonomy*), **schopnost učit se** (*learning*) a **schopnost spolupracovat** (*cooperation*).

Samostatnost agentů zobrazuje jejich schopnost pracovat nezávisle (bez lidského zásahu). Klíčovým prvkem samostatnosti je **vlastní aktivita** (*pro-activeness*) tj. schopnost vytvářet vlastní iniciativu místo prostých reakcí na změny prostředí [Woo1995]. Schopnost spolupráce je důležitou vlastností agentů, neboť odráží jejich vlohy při dosahování společného cíle. Přitom musí být agent schopen dodržovat určité **sociální chování** (*social ability*), aby byl schopen komunikace (s ostatními agenty nebo i s lidmi). V neposlední řadě musí mít agent schopnost učit se tj. reagovat na změny prostředí a přizpůsobovat jim svoje chování.

Použitím těchto základních charakteristik agentů lze rozdělit agenty do několika skupin: **spolupracující** (*collaborative agents*), **spolupracující učící se** (*collaborative learning agents*), **interaktivní** (*interface agents*) a **chytré** (*smart agents*) viz obr. 1.1.



Obrázek 1.1: Typy agentů

Toto rozdělení není konečné, neboť ne každý agent se dá do těchto skupin zařadit. Pokud se však použijí i další vlastnosti agentů, vznikne již poměrně detailní rozdělení agentů do skupin např. *stacionární uvažující spolupracující agent* atd.

1.4.1 Spolupracující agenti (*Collaborative agents*)

Jak je vidět na obrázku 1.1 spolupracující agenti jsou autonomní a dokáží spolupracovat (s ostatními agenty). Mohou se učit, ale na tuto vlastnost není kladen důraz. V případě koordinace více spolupracujících agentů je potřeba, aby dokázali upravovat své akce podle ostatních.

Základními charakteristikami těchto agentů je tedy samostatnost, sociální chování a vlastní aktivita. Většinou se jedná o stacionární agenty [Huh1994]. Využívají se při řešení problémů, které jsou příliš rozsáhlé (z důvodu rozsáhlosti dat nebo nedostatečného centrálního systému), při spojení několika systémů do jednoho (např. expertní systém a systém rozhodování) nebo pro řešení problémů, které jsou přirozeně distribuované (např. letecký provoz, zdravotní péče).

1.4.2 Interaktivní agenti (*Interface agents*)

Tito agenti jsou samostatní a učící se. Jedná se o systémy, které nahrazují asistenty spolupracující nikoli s ostatními agenty, ale s uživatelem. Komunikace s uživatelem nevyžaduje žádný speciální jazyk jako komunikace s jinými agenty.

Většinou se jedná o agenty, kteří pomáhají uživateli s jeho prací. Učení probíhá také od uživatele sledováním jeho akcí a dotazováním se. Typickými představiteli těchto agentů jsou asistenti při psaní programů [Mae1994].

1.4.3 Mobilní agenti (*Mobile agents*)

Jedná se o agenty schopné migrovat po síti, využívat zdroje na vzdálených uzlech, shromažďovat informace a po návratu tyto informace předávat uživateli. Zadaným úkolem může být např. rezervace letenek nebo správa telekomunikační sítě. Výhody používání mobilních agentů jsou naznačeny v kapitole 1.8.

1.4.4 Informační agenti (*Information agents*)

Jedná se o agenty schopné spravovat, manipulovat a porovnávat informace z mnoha distribuovaných zdrojů. Typicky se jedná o programy schopné filtrovat informace tak, aby odpovídaly požadavkům uživatele [Dav1995].

1.4.5 Reaktivní agenti (*Reactive agents*)

Tento typ agentů představuje speciální kategorii agentů bez vnitřního symbolického modelu výpočtu. Místo toho pouze reagují na vnější stimuly. První zmínka o reaktivních agentech je v [Bro1986, Agr1987].

Velkou výhodou těchto agentů je jejich jednoduchost. Proto se používají k takovým úkolům, kde je tato vlastnost využívána: pro kritické aplikace (ošetření chyb) nebo pro přirozenou dekompozici úlohy, kde je kladen důraz na rychlost odezvy (v automobilovém průmyslu pro řízení výroby i samotných automobilů).

1.4.6 Hybridní agenti (*Hybrid agents*)

Tento typ agentů je kombinací některých předchozích typů, neboť každý má nějaké výhody a nedostatky. Pokud například vytvoříme agenta částečně spolupracujícího a reaktivního, agent bude robustní s rychlou odezvou (reaktivní), ale budeme mít možnost ušetřit výpadky za provozu (spolupracující). Obecně se dá říci, že každý agent je hybridní.

1.5 Klasifikace softwarových agentů

Softwarový agent může být klasifikován podle [Gil1995] v rámci prostoru definovaného třemi dimenzemi: *inteligencí*, *činností* a *mobilitou* (viz obr. 1.2).

Inteligence: První rozměr má kořeny ve výzkumných programech umělé inteligence, kde byl termín agent používán pro „softwarového robota“. Cílem bylo vytvoření takového agenta, který bude používat prostředky umělé inteligence pro vyřešení daného problému. Jeho inteligence je klasifikována podle jeho schopnosti logicky myslet, plánovat a učit se. Detailnější rozdělení a klasifikace inteligentních agentů je v [Sho1994].

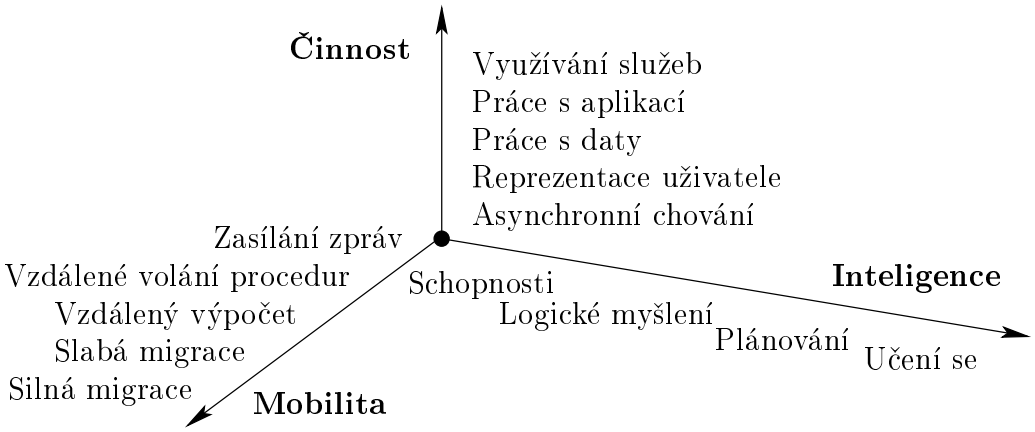
Činnost: Druhý rozměr v sobě zahrnuje stupeň samostatnosti agenta při využívání zdrojů a interakci s ostatními entitami systému. Agenti mohou pracovat s daty, využívat aplikace a služby nebo jiné agenty. Schopnosti agentů vytvářet nějakou činnost jsou detailněji popsány v [Cha1994].

Mobilita: Poslední dimenze prostoru softwarových agentů se objevila teprve nedávno a byla podnícena rychlým růstem síťového výpočetního prostředí. Cílem bylo vytvoření infrastruktury pro vzdálené vykonávání činnosti a mobilitu dat i výpočtu. Agenti se podle této vlastnosti dále dělí na stacionární, intranet agenty, internet agenty a síťové agenty.

1.6 Softwarový agent

Během let se objevilo mnoho různých definic termínu **softwarový agent**. Objevily se také snahy o sjednocení těchto definic do jedné *všezahrnující* nebo alespoň o vymezení rozsahu tohoto pojmu [Fra1996]. V obecném smyslu se slovem **agent** označuje někdo (něco), co aktivně jedná za účelem dosažení určitého cíle, zvláště pokud zastupuje někoho jiného. To je také důvod, proč je zavedena definice agenta z pohledu uživatele.

Agent z hlediska uživatele
Agent je program, který pomáhá uživatelům provádět jejich činnost. Uživatelé pověřují agenty svojí prací.



Obrázek 1.2: Prostor vlastností softwarových agentů

Tato definovaný pojem je však příliš vágní a implikuje otázku: čím se vlastně agent liší od obyčejného programu? Proto existuje celá řada definic softwarových agentů, které více či méně omezují rozsah tohoto pojmu. Většina z nich zdůrazňuje některý rys, především pak inteligenci a autonomní chování (oblast softwarových agentů patří do oboru umělé inteligence) nebo mobilitu (mobilní agenti). Nejobecnější definice popisuje softwarového agenta pomocí jeho charakteristik:

Agent z hlediska systému

Agent je softwarový objekt, který:

- *je umístěn uvnitř výpočetního prostředí,*
- *má tyto vlastnosti:*
  - *samostatnost: agent pracuje bez přispění lidí nebo jiných subjektů a kontroluje svoje akce a vnitřní stav,*
  - *sociální chování: agent spolupracuje s jinými agenty a komunikuje s nimi*
  - *reaktivitu: agent vnímá své okolí a včas reaguje na jeho změny,*
  - *vlastní aktivitu: agent nejenom reaguje na změny prostředí, ale je také schopen vytvářet vlastní iniciativu,*
- *navíc může mít některé z následujících vlastností:*
  - *komunikativnost: je schopen komunikace s ostatními agenty,*
  - *mobilitu: může se pohybovat z jednoho uzlu na druhý,*
  - *schopnost učit se: přizpůsobuje se podle svých předchozích zkušeností.*

Různé systémy [Ibm1998, Gen1997] např. implementují *samostatnost* tak, že vytvoří pro každého agenta samostatné vlákno. Pokud ale bude systém obsahovat

velké množství agentů, pravděpodobně se zahltí. Proto bývá *samostatnost* agenta chápána spíše jako *pocit* uživatele při návrhu takového systému.

Z hlediska distribuovaných systémů nepřináší softwarový agent nic nového. Agenti mohou spolupracovat prostřednictvím počítačové sítě, agent může být realizován distribuovaně (více částí na různých uzlech v síti). Skutečně novou koncepcí jsou až **mobilní agenti**.

1.7 Mobilní agent

Mobilita je jednou z vlastností, kterou nemusí všichni agenti obsahovat. Agent může být stále na jednom uzlu a pouze komunikovat se svým okolím. Potom se jedná o **stacionárního agenta** (*stationary agent*).

Stacionární agent

Stacionární agent provádí výpočet pouze na uzlu, na kterém vznikl. Jestliže chce komunikovat s agenty na jiném uzlu, nemá jinou možnost, než používat komunikačních mechanismů systému.

Naproti tomu *mobilní* agent není nijak svázán s uzlem, na kterém začal svůj výpočet. Mobilní agent má absolutní svobodu migrace mezi uzly systému. Migrace agenta je proces, který přenese kód agenta spolu s jeho stavem na nový uzel, kde je jeho výpočet obnoven. Kódem agenta se rozumí popis jeho výpočtu (v programovacím jazyce). Stav agenta je hodnota všech proměnných agenta v době jeho přerušení.

Mobilní agent

Mobilní agent není svázán s uzlem, na kterém začal svůj výpočet. Má jedinečnou schopnost migrovat mezi uzly systému. Tato schopnost dovoluje agentovi migrovat jako objekt na jiný uzel, který obsahuje agenty, s nimiž chce agent komunikovat.

1.7.1 Složení agenta

Hostitel musí znát a umět zpracovat vnitřní složení agenta. Bez této znalosti není možná migrace, protože agent se musí přesunout na jiný uzel (do jiného hostitele), aniž by došlo k jakékoliv jeho změně. Program agenta se skládá ze dvou hlavních složek:

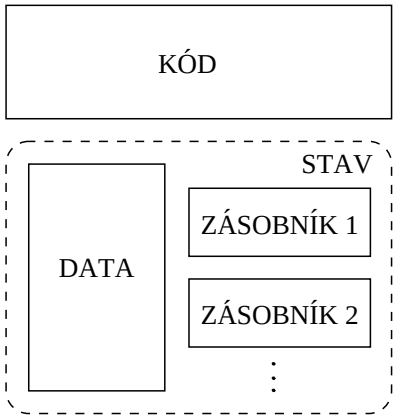
Programový kód (*Code*): Souhrn všech příkazů a instrukcí. Je společný pro více instancí agenta a v čase se nemění. Výjimkou je tzv. samomodifikující se kód, ovšem tato programátorská praktika je všeobecně považována za velmi špatnou. Proto ji systémy mobilních agentů většinou nedovolují použít (samozřejmě to se netýká jazyků, kde lze jen velmi těžko odlišit kód od dat, jako je např. *LISP*).

Forma kódu závisí na použitém programovacím jazyce. Většinou to jsou definice tříd (a jejich metod) nebo kód použitých procedur.

Stav výpočtu (*Execution State*): Momentální stav agenta se neustále mění a je rozdílný pro různé agenty se stejným kódem. Právě přenos a uchování stavu je základní částí celého systému, protože migrace je především přenos stavu. Výpočetní stav samotný se skládá ze dvou částí:

- **Stav dat (*Data State*):** Všechna data, která má agent v paměti. Teoreticky by tato data mohla být uložena i jiným způsobem (na disku, v databázi,...), ale prakticky se této možnosti nevyužívá a agent má všechna potřebná data v operační paměti.
- **Stav vláken (*Thread State*):** Stav vláken určuje, kde se právě nachází běh programu. Reprezentován je zásobníkem (někdy se používá jen termín zásobník – *stack*), čítačem instrukcí a případně dalšími informacemi (záleží na procesoru, interpretru apod.).

  Pokud se agent skládá z více vláken, pak stav zahrnuje všechny zásobníky těchto vláken. Je zřejmé, že bez těchto informací nelze po migraci obnovit běh agenta v původní podobě.



Obrázek 1.3: Typické složení programu (agenta)

1.7.2 Historický vývoj

Zasílání zpráv (*Message Passing*): Komunikace mezi uzly distribuovaného systému může probíhat několika způsoby. Metoda **zasílání zpráv** (*Message Passing – MP*) je prvním z nich. Uzly spolu komunikují pomocí zpráv, které jsou explicitně odesílány a přijímány. Tento koncept byl navržen pro asynchronní i synchronní zasílání zpráv. Asynchronní zprávy jsou velmi

flexibilní a jejich podpora je zabudována v mnoha systémech. Na druhou stranu vývoj a ladění systémů založených na metodě zasílání zpráv je velmi obtížné.

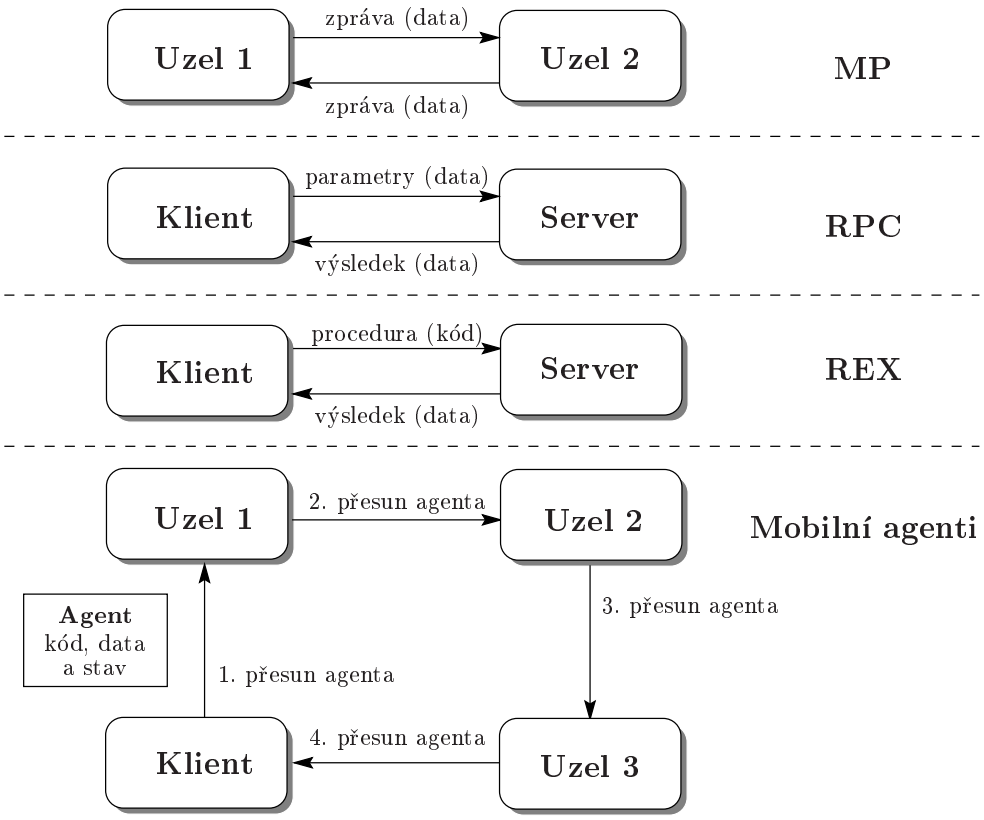
Vzdálené volání procedur (*Remote Procedure Call*): Na vyšší úrovni abstrakce komunikace je postavena koncepce **vzdáleného volání procedur** (*Remote Procedure Call – RPC*) [Tay1990]. Komunikující procesy vyvolávají vzdálené procedury místo aby explicitně posílaly a přijímaly zprávy. RPC je založeno na modelu klient-server: klient posílá dotazy na server, který vykonává dotazované procedury a nazpátek vrací výsledky. Většina implementací RPC podporuje jak asynchronní tak synchronní volání procedur. Většina složitých komunikačních mechanismů je skrytá v tzv. **spojkách** (*stubs*), které jsou automaticky generovány ze specifikace rozhraní. Tyto spojky umožňují definovat metody zakódování dat pro přenos v heterogenním prostředí, zvolit formát předávaných dat, atd.

Vzdálený výpočet (*Remote Execution*): RPC samozřejmě funguje správně pouze v případě, že volaná procedura je přítomna na vzdáleném uzlu. Tento požadavek značně omezuje použití principů RPC v rozlehlých distribuovaných systémech. V mnoha případech je potřeba *odeslat* proceduru na vzdálený uzel a spustit ji tam (např. když klient potřebuje zpracovat rozsáhlá data na vzdáleném uzlu a výsledek odeslat zpět). Tento model komunikace se nazývá **vzdálené vyhodnocení** (*Remote Evaluation*) a je popsán v [Sta1990]. Jedná se pouze o rozšíření koncepce RPC. V tomto případě je kromě parametrů volané procedury přenášen i její programový kód. Tato koncepce může být zobecněna v tom smyslu, že kód může být přenášen nejenom ve směru od klienta na server, ale i opačně. Například *Java aplety* a prvky *ActiveX* jsou technologie podporující přenos ve směru od serveru, kdežto *Java servlety* dovolují přenos z klienta na server. Všechny komunikační modely podporující přenos v jednom nebo v druhém směru spadají do pojmu **vzdálený výpočet** (*Remote EXecution – REX*). Potom je koncepce vzdáleného vyhodnocení pouze speciálním případem vzdáleného výpočtu.

Mobilní agenti (*Mobile Agents*): V systémech jako *R2D2* [Vit1981] a *Chorus* [Ban1986] byl zaveden pojem **aktivních zpráv** (*active message*), které mohou putovat mezi uzly sítě a obsahují programový kód, který je pak na uzlech prováděn. Obecnějším pojmem jsou **mobilní objekty** (*mobile object*), které zapouzdřují data spolu s množinou operací (metodami) nad těmito daty. Mobilní objekty pak mohou být přenášeny mezi uzly počítačové sítě. *Emerald* [Jul1988] je příkladem systému, který poskytuje objektům možnost migrace, ale je omezen na homogenní počítačové sítě.

Mobilní agenti se vyvinuly právě z těchto předchůdců. Obrázek 1.4 ukazuje rozdíl mezi metodami RPC a REX. V RPC jsou data posílána mezi klientem

a serverem oběma směry. Při použití metody REX se posílá ve směru od klienta programový kód a data se pouze vrací. Naproti tomu **mobilní agent** je program (zahrnující *programový kód*, *data* a *stav* – více viz kapitola 1.7.1), který může migrovat mezi uzly sítě, odesílat informace klientovi nebo se vrátit zpět na klienta. Agent je tedy více autonomní než prosté volání procedur.



Obrázek 1.4: Vývoj distribuovaných aplikací

Telescript [Whi1995], vyvinutý firmou General Magic, byl prvním systémem určeným primárně pro programování mobilních agentů. Třebaže byl komerčně neúspěšný a v dnešní době již není k dispozici, drží si historické prvenství mezi jazyky s podporou mobilních agentů. Za ním následovaly systémy *Tacoma* [Joh1995] a *Agent Tcl* [Gra1996], ve kterých bylo tělo agenta zapisováno pomocí skriptu. Vývoj programovacího jazyka Java a programovacího prostředí s podporou mobilního kódu [Gos1996, Lin1996] vedl ke zrychlení výzkumu v této oblasti. *Aglets* [Kar1997], *Voyager* [Obj1997] a *Concordia* [Mit1997] jsou příkladem systémů mobilních agentů založených na jazyce Java.

1.7.3 Typy mobility agentů

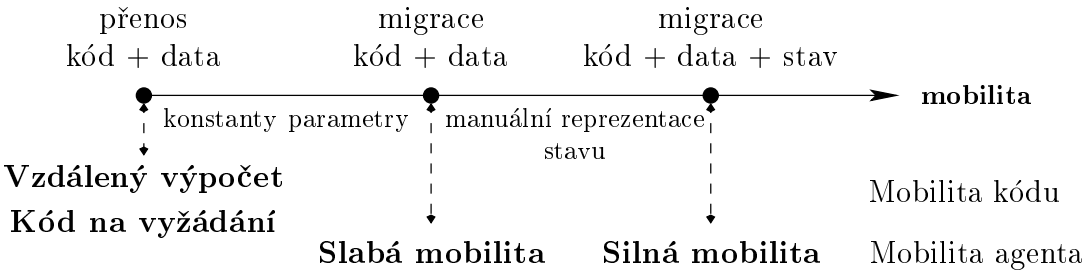
V případě použití koncepce **vzdáleného výpočtu** je kód přenesen na jiný uzel ještě před svým spuštěním. Tam je prováděn až do svého ukončení, tj. kód je přenesen pouze jednou. Přenos zahrnuje programový kód a případné parametry výpočtu. Jakmile je agent spuštěn, může použít stejný mechanismus pro spuštění jiných agentů. Velmi podobný mechanismus přizpůsobený modelu klient-server je **vzdálené vyhodnocení**. V tomto případě je agent opět přenesen před svým spuštěním a jeho výpočet je dokončen na jednom uzlu. Výsledky výpočtu se však odesílají zpátky na původní uzel (klient). Tato metoda muže být použita rekurzivně pro vytvoření stromové struktury výpočtu.

V předchozím případě byl cílový uzel vybrán entitou (uzel nebo agent), která také sama přenos provedla. Pokud se použije opačný přístup, tj. přenos bude iniciován cílovou entitou, nazývá se tato koncepce **kód na vyžádání** (*code on demand*). Toto komunikační schéma se používá v modelu klient-server, kdy jsou programové kódy umístěny na serveru a mohou být staženy na klienta na vyžádání.

Oba dva předchozí přístupy podporují spíše „mobilitu kódu“ než „mobilitu agentů“, neboť kód je přenesen jen jednou ještě před svým spuštěním. Další dvě metody migrace podporují nejen přenos agenta, ale i jeho stavu (viz kapitola 2.1). **Stav agenta** (*agent state*) se skládá ze dvou částí: **stavu dat** (*data state*) a **stavu výpočtu** (*execution state*). Zatímco první z nich obsahuje stav globálních proměnných agenta, druhý obsahuje stav lokálních proměnných a parametrů včetně **stavu vlákna** (*thread state*). Potom lze rozlišit dva typy mobility: **slabou mobilitu** (*weak mobility*) a **silnou mobilitu** (*strong mobilitu*).

Silná mobilita (*Strong Mobility*): V tomto případě je přenesen celý **stav agenta**, tj. stav dat i stav výpočtu, spolu s programovým kódem. Jakmile je agent přijat, je jeho stav obnoven. Z hlediska programátora je tento typ migrace velmi zajímavý, neboť o zachycení stavu, přenos i obnovení výpočtu se postará systém zcela transparentně. Na druhou stranu v heterogenním prostředí je potřeba zajistit správnou syntaxi přenosu. Navíc systém musí podporovat zachycení a obnovení stavu agenta, což podporují pouze některé jazyky např. *Tycoon*. Protože stav agenta může být velký (třeba v případě vícevláknových agentů), může být silná mobilita časově velmi náročná.

Slabá mobilita (*Weak Mobility*): Problémy zmíněné v předchozím odstavci vedly k definování pojmu slabá mobilita, kdy je přenášen pouze **stav dat**. Velikost tohoto stavu může být omezena například tak, že programátor může definovat, které proměnné se budou přenášet a které ne. Následkem toho je také programátor zodpovědný za reprezentaci stavu pro přenos a musí definovat metody, které se vyvolají po migraci na novém uzlu. Tato koncepce tedy podstatně snižuje množství přenášených dat, ale více zatěžuje programátory a výsledný kód agentů je složitější.



Obrázek 1.5: Stupně mobility

Na první pohled vypadá silná mobilita podobně jako koncepce **migrace procesů** jak je popsána v [Smi1988], nicméně existuje zde zásadní rozdíl. Při migrace procesů rozhoduje o tom, který proces bude kdy odeslán samotný systém, tj. z hlediska aplikace je migrace zcela transparentní. Naproti tomu migrace agenta je zcela v režii agenta. Agent sám rozhodne kam bude kdy přenesen. Následkem toho musí být v programovacím jazyce použitém pro zápis kódu agenta speciální příkaz, kterým se migrace zahájí.

1.8 Výhody použití mobilních agentů

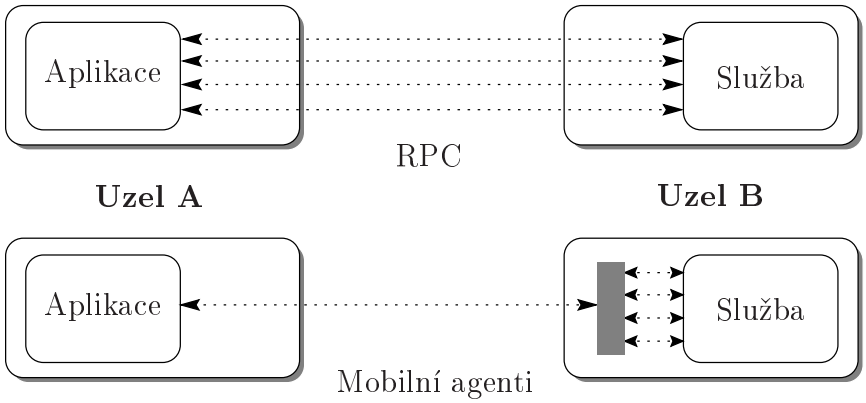
V [Har1995] jsou srovnáváni mobilní agenti s RPC a systémem posílání zpráv. Přestože jsou výhody mobilních agentů zřejmé, je potřeba zmínit několik důvodů, proč je využívat.

- **Snižují množství přenášených dat:** Distribuované systémy se obvykle spoléhají na nějakou množinu komunikačních protokolů sdružující operace vhodné pro danou úlohu. Ty mají vlastní režii a výsledkem je větší zátěž počítačové sítě. Mobilní agent ale obsahuje jak kód tak potřebná data. Jakmile je odeslán na cílový uzel, veškerá komunikace probíhá lokálně, jak je vidět na obrázku 1.6.

  Použití mobilních agentů samo o sobě nesníží množství přenášených dat. Agenti

  1. redukují počet interakcí mezi entitami na různých uzlech,
  2. snižují počet přenášených dat mezi uzly.

  Závěr je jednoduchý: přesouvat raději programový kód k datům než data na uzel s kódem. Samozřejmě, že použití agentů je výhodné pouze v tom případě, že zvýší výkonnost systému. Výkonnostní modely systémů mobilních agentů jsou v [Chi1997, Str1997]. Množství přenášených dat může být sníženo již při použití REX, ale mobilní agenti poskytují větší prostor pro optimalizaci.



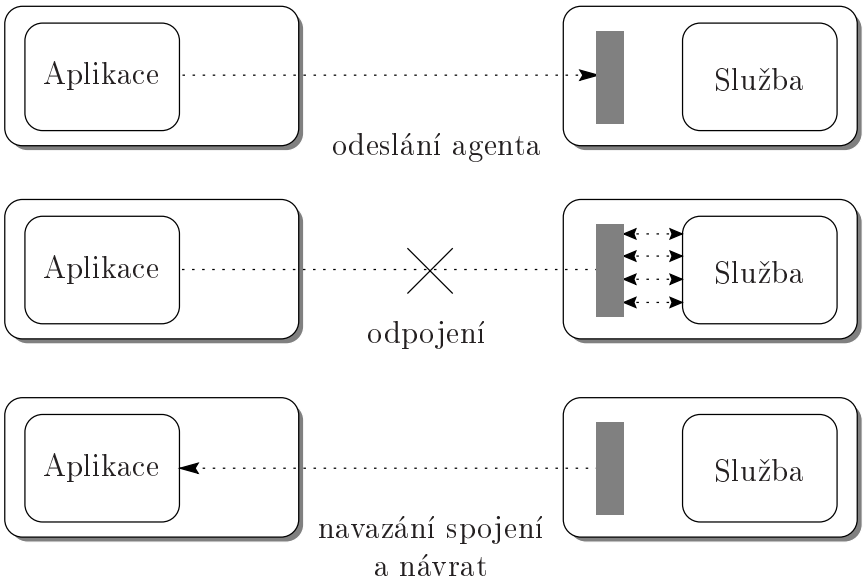
Obrázek 1.6: Mobilní agenti a snižování zátěže počítačové sítě

- **Obchází zpoždění počítačové sítě:** Systémy reálného času potřebují krátkou dobu odezvy. Ovládání takových systémů přes počítačovou síť způsobuje prodlevy v jejich řízení. Pro časově kritické systémy není takové chování sítě akceptovatelné. Mobilní agenti jsou možným řešením, protože mohou být odesláni centrálním prvkem a zasahovat přímo v místě události.

- **Jsou vykonáváni asynchronně a autonomně:** Asynchronní komunikační mechanismy umožňují asynchronní zpracování požadavků. Protože požadavky vznikají asynchronně, musí systém zajistit jejich neustálou obsluhu. Nepřetržité spojení mezi oběma komunikujícími stranami může být velmi náročné (někdy i nemožné). Použitím mobilních agentů lze odeslat část aplikace přímo na server. Jakmile je přenos ukončen, je může být uzel odpojen od sítě (viz obrázek 1.7). Později se aplikace dotáže na výsledky zpracování. Je důležité, aby systém garantoval doručení agenta bez ohledu na komunikační chyby. Bohužel v současné době ani jeden systém mobilních agentů nepodporuje tento typ odolnosti proti poruchám.

- **Zapouzdřují komunikační protokoly:** Komunikace v distribuovaných systémech vyžaduje, aby každý uzel měl implementován množinu komunikačních protokolů potřebnou pro odesílání dat resp. příjem dat a jejich interpretaci. Protože se protokoly neustále vyvíjí, je nepohodlné ne-li přímo nemožné implementovat všechny nové možnosti. Následkem toho si s sebou protokoly nesou „dědictví“ ve formě kompatibility s předchozími verzemi. Naproti tomu mobilní agenti komunikují s okolím pomocí tzv. kanálů založených na vlastních protokolech.

- **Dynamicky se přizpůsobují:** Mobilní agenti mají schopnost reagovat na změny výpočetního prostředí. Více agentů se může samostatně pohybovat po síti a tak přizpůsobovat svoji konfiguraci konkrétní úloze.



Obrázek 1.7: Autonomní vykonávání mobilních agentů

- **Jsou přirozeně heterogenní:** Uzly počítačové sítě jsou většinou heterogenní, ať už z hlediska hardware nebo software. Protože mobilní agenti jsou závislí pouze na vlastním výpočetním prostředí (nejsou závislí ani na použitém hardwaru ani na komunikační infrastruktuře), poskytují optimální prostředí pro integraci různých systémů.
- **Jsou robustní:** Protože agenti dokáží reagovat na neočekávané události, dají se použít k návrhu robustních systémů, které jsou odolné proti poruchám. V případě poruchy jednoho uzlu systému, jsou všichni agenti o této události informováni a je jim poskytnut čas pro odeslání na jiný uzel systému.

1.9 Typické aplikace mobilních agentů

Prozatím nebyla nalezena aplikace, která by jednoznačně neumožňovala použití mobilních agentů. Aplikace využívající mobilních agentů jsou zmíněny v [Har1995, Lan1998]. Nejzajímavější jsou:

- **Monitorování:** Agenti mohou posílat monitoru určité informace nebo čekat na specifickou událost. Jakmile na stane změna, agenti na ní reagují podle svého algoritmu, tj. odešlou zprávu nebo nakoupí akcie na burze. Jiným příkladem může být monitorování a management počítačové sítě [Bau1997a]. Hlavní výhodou mobilních agentů je v tomto případě flexibilita a asynchronní a autonomní činnost.

- **Paralelní výpočty:** Paralelní procesy mohou být pomocí mobilních agentů distribuovány velmi snadno. Stejně tak snadno mohou reagovat na změny prostředí (např. zatížení uzlů). Použitím takového systému se výpočet rozdělí na menší části, které lze samostatně počítat. Ty jsou pak distribuovány automaticky a využijí lépe celou výpočetní kapacitu sítě uzlů.

- **Distribuované vyhledávání informací:** Místo aby se velké objemy dat přenášely počítačovou sítí na klienta a na něm se teprve zpracovávaly, agenti mohou být odesláni na uzel s informacemi a tam je lokálně zpracovávat. Mobilní agenti pak mohou obsahovat specifické vyhledávací algoritmy, které dovolují vytvářet sémantické závislosti mezi nalezenými daty. Pokud je hledaná informace rozprostřena na více uzlech, je použití mobilních agentů velmi užitečné [The1999].

- **Automatická instalace software:** Mobilní agenti mohou být využiti k automatické instalaci software a k jeho aktualizaci. Předtím, než je softwarový balík přenesen na cílový uzel, agent může zjistit informaci o prostředí např. verze jiných balíků. Uživateli je nabídnuta možnost voleb instalace a agent potom hlídá případné další aktualizace.

- **Elektronický obchod:** Elektronický obchod je rozhodně jednou z nejatraktivnějších oblastí aplikací mobilních agentů. Obchodní transakce vyžadují nepřetržitý přístup na zdroje umístěné na vzdálených uzlech a okamžitou reakci na jejich změnu, což není možné při použití pomalých počítačových sítí. agenty je *TabiCan* [Ibm1997].

- **Šíření informací:** Mobilní agenti mohou rozšiřovat informace (např. novinky) mezi zákazníky. Mohou se také starat o přístupová práva, např. zabezpečují, aby zákazníci mohli číst články pouze pokud již zaplatili. Jedním z příkladů takového systému je *MEDIA* [Kon1996].

- **Systémy řízení práce:** V řídících aplikacích dochází k neustálým výměnám informací mezi spolupracujícími subjekty. Mobilní agenti zde mohou využívat svou mobilitu, vlastnost snižující zatížení počítačové sítě. Datový tok je pak nezávislý na různých aplikacích.

1.10 Shrnutí

V této kapitole byl definován agent jako softwarový objekt, který je situován uvnitř výpočetního prostředí a má určité vlastnosti (viz kapitola 1.3). Mobilní agent je agent, jehož život není svázán s uzlem, na kterém svůj výpočet začal. Tato vlastnost (mobilita) umožňuje mobilnímu agentovi přesunout se na ten uzel sítě, na němž je entita, s níž chtěl komunikovat.

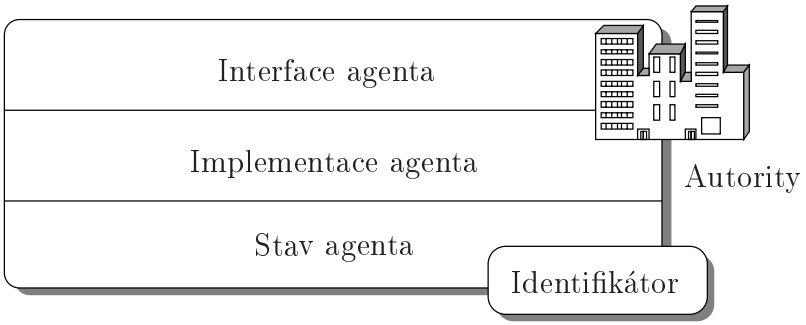
Kapitola 2

Principy systémů mobilních agentů

V této kapitole budou vysvětleny základní pojmy týkající se systémů mobilních agentů. Dále bude popsáno chování agentů v průběhu jejich životního cyklu a programová primitiva pro operace nad agentem. Nejdůležitější vlastnosti mobilních agentů, *mobilitě*, bude věnována samostatná podkapitola. Dále budou vysvětlena témata související se systémem – přístup ke službám, identifikace a autentifikace součástí systému, bezpečnost a komunikace.

2.1 Agent

Mobilní agent je entita, která má 5 atributů: *identifikátor, stav, implementaci, rozhraní* a *autority* (viz obrázek 2.1). Když se agent pohybuje po síti, všechny tyto vlastnosti si bere s sebou.



Obrázek 2.1: Atributy agenta

Identifikátor agenta (*Identifier*): Každý agent má identifikátor, který je jednoznačný, jedinečný a nezměnitelný během celého jeho života. Identifikátor nemusí být součástí agenta, ale mohou jej poskytovat například **adresářové služby** (*directory services*). Protože je jednoznačný a nezměnitelný, může se používat k jednoznačné identifikaci agenta.

Stav agenta (*State*): Pokud agent migruje, přenáší si s sebou i svůj stav. Stav je důležitý pro to, aby mohl být spuštěn na cílovém uzlu. Stav agenta je tedy **otisk** (*snapshot*) jeho výpočtu. Mnoho programovacích jazyků dělí stav agenta na **stav výpočtu** (*execution state*), což je stav programu (ukazatel na prováděnou instrukci a zásobník), a **stav dat** (*data state*), což je hodnota proměnných objektu.

Systémy agentů vždy nepotřebují zachytit a odeslat stav výpočtu, protože tento stav se dá zrekonstruovat ze stavu objektu. Jak bude uvedeno později, tento postup se využívá u agentů napsaných v jazyce Java, který nedovoluje přistupovat ke stavu výpočtu. Java totiž nepodporuje přístup k zásobníku programu, protože jeho reprezentace je na různých platformách různá.

Implementace agenta (*Implementation*): Stejně jako každý program i agent potřebuje programový kód, který má být spuštěn. Když agent migruje, je buď přenášen celý programový kód, nebo se předpokládá, že je už kód na cílovém uzlu přítomen.

Kód agenta musí být spustitelný na cílovém stroji, ale jeho běh musí být také bezpečný. Skripty a interpretované jazyky, které jsou většinou nezávislé na počítačové platformě, poskytují pro běh agenta bezpečné prostředí, které nedovolí přistupovat k nepovoleným zdrojům na uzlu.

Rozhraní agenta (*Interface*): Agent poskytuje **rozhraní** (*interface*) umožňující ostatním agentům a systému s ním komunikovat. Rozhraním může být jakákoli množina metod, které agent poskytne okolí a které dokáže okolí využívat – příkladem může být *Knowledge Query and Manipulation Language* (KQML).

KQML je jazyk popisující komunikaci mezi agenty. Poskytuje různé typy zpráv označované jako **soupisy akcí** (*performatives*), které přesně vyjadřují, co má agent provádět za operace. Jejich formát vyjadřuje elementární komunikační mechanismy při komunikaci mezi agenty.

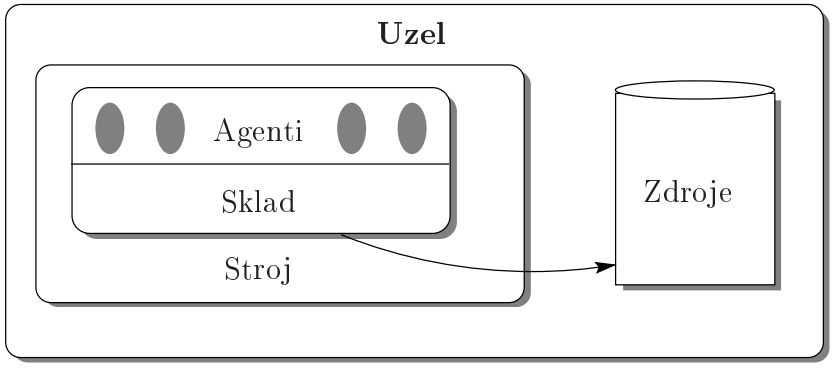
Autority (*Principals*): Autorita je entita, jejíž identita může být ověřena jakýmkoli systémem, který k ní chce přistupovat. Autoritou se může stát jednotlivec, organizace nebo sdružení. Její identita se skládá ze jména a dalších atributů. Pro agenta jsou potřeba nejméně dvě autority:

- **Výrobce** (*Manufacturer*) je autor, tedy ten, kdo napsal kód agenta.

- **Vlastník** (*Owner*) je autorita, která má legální a morální odpovědnost za chování agenta.

2.2 Sklad

V předchozí části byly popsány základní atributy agenta a nyní zbývá popsat prostředí, ve kterém agent pracuje. Když je agent spuštěn, obvykle migruje mezi různými uzly systému. Obecně lze popsat **sklad** (*repository*) jako prostředí umožňující práci agentům (viz obrázek 2.2). Představuje tedy vstupní bod pro agenta, který přicestoval a chce být spuštěn. Sklad poskytuje jednotnou množinu služeb, kterých agent může využívat. V neposlední řadě se také sklad dá označit za „operační systém“ pro agenty. Sklad má 4 atributy: *stroj, zdroje, poloha* a *autority*.



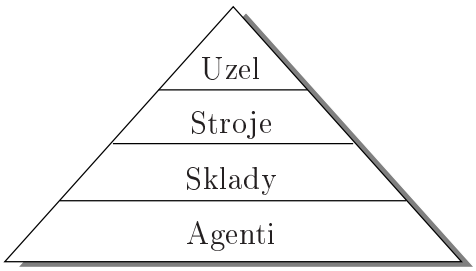
Obrázek 2.2: Sklad agentů a stroj

Stroj (*Engine*): Sklad sám nemůže agenta spustit. Agent je spuštěn ve stroji, který slouží jako virtuální stroj pro sklad a agenty. Pro systémy napsané v jazyce Java je jádro identické s tzv. *Java virtual machine* a operačním systémem.

V hierarchické struktuře na obrázku 2.3 je vidět, že na každém uzlu může být spuštěno několik strojů, každý stroj může mít několik skladů a každý sklad obsahuje několik agentů. Skutečnost, že každý stroj obsahuje více než jeden sklad vyžaduje, aby měl sklad jedinečné jméno ve stroji.

Zdroje (*Resources*): Stroj a sklad poskytují kontrolovaný a bezpečný přístup k lokálním zdrojům a službám jako jsou síť, databáze, procesor a paměť, disk a ostatní hardware i software.

Poloha (*Location*): Poloha je velmi důležitým pojmem. Bývá definována jako kombinace jména skladu, ve kterém agent běží a adresy stroje, ve kterém je



Obrázek 2.3: Hierarchická struktura systému

sklad spuštěn. Poloha je typicky popsána IP adresou uzlu a portem stroje se jménem skladu jako parametrem.

Autority (*Principals*): Stejně jako agent i sklad má dvě autority:

- **Výrobce** (*Manufacturer*) je autor nebo poskytovatel implementace skladu.
- **Vlastník** (*Owner*) je autorita odpovědná za operace ve skladu.

2.3 Životní cyklus agenta

Během svého života se agent nachází v několika stavech, které jsou společné pro většinu mobilních agentů. Tyto stavy jsou výsledkem operací, prováděných hostitelem. Souhrn všech operací, stavů a jejich vzájemných vazeb se nazývá **životní cyklus** (*lifecycle*). Lze ho definovat i u stacionárního agenta, ale zásadní význam má až u agentů mobilních.

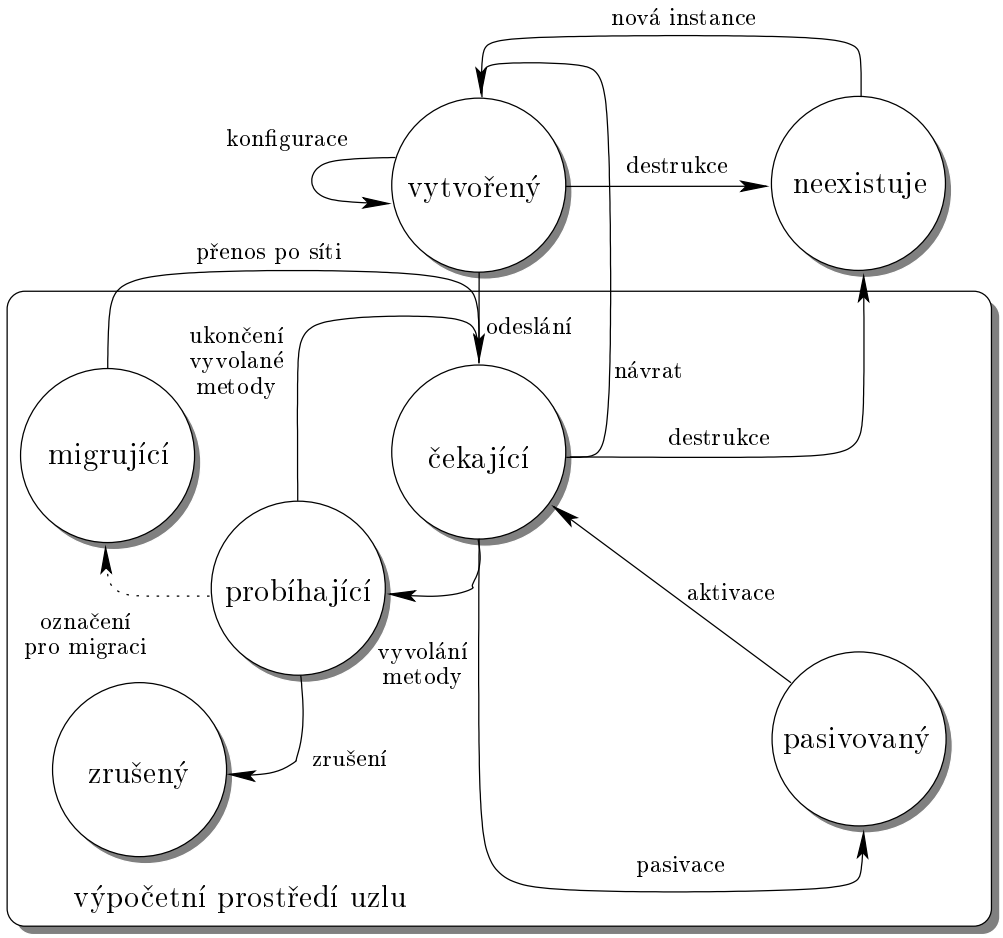
Životní cyklus agenta je jednou ze základních charakteristik systému mobilních agentů. Každý definuje životní cyklus jinak, ale všechny dohromady sdílejí několik základních principů. Část životního cyklu je dokonce společná pro stacionární a mobilní agenty.

Graf znázorňující životní cyklus mobilního agenta je na obrázku 2.4. Jednotlivé operace a stavy, kterých se jimi dosáhne, jsou popsány dále. Je myslím nutné zdůraznit, že vesměs neexistuje jednotná terminologie. Jediné široce používané pojmy jsou **migrace** (*migration*) a případně ještě **pasivace** (*passivation*).

2.3.1 Stavy

připravený, vytvořený (*prepared*): Agent je vytvořen, ale ještě nefunguje (není prováděn) a není ani ve správném prostředí (na hostiteli).

aktivní (*active*): V tomto stavu je agent aktivní, tj. vykonává programový kód. V některých systémech se tento stav dělí na:



Obrázek 2.4: Životní cyklus agenta

čekající (*waiting*): Agent je na hostiteli, s přidělenými zdroji, v konzistentním stavu. Není však momentálně prováděn (nemá žádné aktivní vlákno). Tento stav může nastat buď dobrovolně (agent čeká na nějakou událost) nebo z rozhodnutí hostitele.

běžící, žijící (*executing*, *alive*): Agent provádí nějakou aktivní činnost.

pasivovaný (*passivated*): Agent je persistentně uložen na trvalém médiu (disku, v databázi, ...). K tomuto uložení může dojít v případě dlouhodobé nečinnosti agenta (šetří se tak zdroje) nebo pokud hostitel končí činnost (po opětovném spuštění bude agent obnoven).

zrušený (*aborted*): Do tohoto stavu se agent dostane, pokud provede nepovolenou operaci, dosáhne nekonzistentního stavu nebo v něm dojde k jiné chybě. Agent je vlastně ukončen, ale není zrušen a klient (popř. jiná speciální entita) si může jeho stav vyžádat a provést analýzu.

navrácený (*retrieved*, *retracted*): Agent byl navrácen zpět klientovi, který z něj může získat všechna nasbíraná data a výsledky činnosti.

migrující (*migrating*): Momentálně probíhá migrace, tj. přesun z jednoho uzlu (hostitele) na jiný.

2.3.2 Vytvoření a vyslání agenta

Tyto dvě operace jsou sice v grafu zakresleny zvlášť, ale ne vždy se rozlišují. Mnohé systémy je považují za jednu operaci.

- **vytvoření** – operace **vytvoření** (*creation*) reprezentuje vznik samotného agenta. Typicky se vytvoří instance tříd, alokuje se paměť, inicializují se data, atd. Operace může také zahrnovat počáteční komunikaci s uživatelem, kdy jsou zadána vstupní data (úkoly). Výsledkem je již existující agent, který však ještě „nefunguje“.

  Operace je společná pro stacionární i mobilní agenty a provádí ji vždy klient (pokud v systému existuje jako samostatná entita) nebo hostitel (pokud samostatný klient neexistuje a je součástí hostitele).

- **vyslání (spuštění)** – mobilní agent může fungovat pouze v hostiteli a je tedy nutné ho na něj přesunout. Protože se tím vlastně agent oživí a vyšle plnit své úkoly, nazývá se tato operace **vyslání** (*dispatch*).

  Tato operace je velice podobná migraci a postup je z větší části shodný. Rozdíly jsou zejména:

  – při vyslání není nutné rušit dosavadního agenta a jeho přidělené zdroje (tento rozdíl je zcela zřejmý),
  – migrace je vyvolána většinou z iniciativy agenta, zatímco vyslání je vyvoláno klientem,
  – při vypuštění ještě nemusí být agent kompletní. Mnoho systémů například přiděluje agentovi jméno nebo adresu až při vypuštění na straně hostitele (zajistí se tak plná kontrola nad jejich jednoznačností).

  Po dokončení spuštění je agent v aktivním stavu (*Active*) a může pracovat.

2.3.3 Ukončení

Agent je ukončen buď na vlastní přání (práce byla dokončena a jeho další existence není nutná) nebo na požadavek klienta. Běh agenta se zastaví, odeberou (zruší) se všechny přidělené prostředky a zruší se všechny odkazy na něj (např. ve jmenných službách apod.). Po provedení této operace již agent neexistuje.

2.3.4 Navrácení

Navrácení nebo též **vyžádání** (*retract*) je opakem **vyslání**. Klient si od hostitele (nemusí to být tentýž jako v případě vyslání) vyžádá, aby mu poslal agenta zpět. Hostitel agenta ukončí a jeho stav pošle klientovi. Klient tento stav může analyzovat a extrahovat všechna data a výsledky, které agent za dobu své činnosti nasbíral.

Ne všechny systémy tuto operaci podporují. Alternativní metodou je navázání komunikace s agentem, přijmutí všech výsledků a následné ukončení agenta.

2.3.5 Zrušení

Po dobu běhu agenta se může stát, že dojde k fatální chybě, kterou agent neošetří. Může jít o chybu v programu (*bug*) nebo o úmyslný pokus o napadení hostitele. Hostitel v takovém případě rozhodne o tom, že agent již nemůže pokračovat v činnosti a zruší ho (*abortion*). Na rozdíl od normálního ukončení se zachová stav agenta, zakóduje se a pošle klientovi, vlastníkovi agenta nebo jiné entitě (záleží na systému). Poslání nemusí být provedeno aktivně, může si stav uchovat a počkat, až si ho někdo (něco) vyzvedne.

Takto uchovaný stav agenta (společně s informací od hostitele, proč byl zrušen) lze analyzovat a odhalit tak chyby v programu. Navíc lze extrahovat již získaná data, jejichž ztráta by mohla být nepříjemná.

2.3.6 Pasivace a aktivace

V některých situacích se hostitel může rozhodnout, že je možné agenta dočasně „uspat“ (*suspend*) a odložit do perzistentního (trvalého) úložiště (např. disk nebo databáze). Mezi typické situace patří ukončení činnosti hostitele (stroje), kdy je nutné, aby po opětovném spuštění byli agenti obnoveni. Jinou častou situací je případ, kdy hostitel potřebuje uvolnit některé prostředky.

Tento proces se nazývá **pasivace** (*passivation*) a agent v perzistentním úložišti (*persistent storage*) se nazývá **pasivovaný** (*passivated*). Opačný proces, tj. obnovení agenta z úložiště se nazývá **aktivace** (*activation*).

Jakým způsobem se pasivace provede a v jakém formátu je agent uložen závisí zcela na systému (konkrétně hostiteli). Většinou jsou operace pasivace a aktivace podobné procesu migrace, kdy se migruje na stejného hostitele.

2.3.7 Migrace

Operace **migrace** je nejdůležitější částí systému mobilních agentů, protože zajišťuje právě onu mobilitu (společně s operací **vyslání**, která však u některých systémů chybí). Během migrace se agent přesune z jednoho hostitele (uzlu) na jiný.

Ačkoliv migrace je jen jedna operace, která by měla být atomická, v praxi trvá nezanedbatelně dlouhou dobu. Během ní agent nemůže měnit svůj stav a nesmí tedy nic dělat. Je také nutné zabránit jakýmkoliv jiným operacím. Proto je do životního cyklu agenta přidán speciální stav **migrující** (*migrating*), ve kterém se agent nachází během migrace. Z tohoto stavu vede zpět do aktivního stavu nepojmenovaný přechod, který reprezentuje dokončení migrace.

2.4 Migrace agenta

Základní vlastností mobilního agenta je schopnost migrace, tedy přesunout se na jiný uzel sítě. Migraci si však agent nezajišťuje sám (to by bylo z různých důvodů velice nepraktické), ale zajistí mu ji aktuální hostitel.

Z pohledu agenta pak migrace vypadá přibližně takto:

1. Vlastní činnost.
2. Rozhodnutí o migraci.
3. Provedení migrace.
4. Pokračování v činnosti, tentokrát však na jiném uzlu.

Samotná migrace by pro agenta měla být zcela transparentní. Podle způsobu provedení se může jednat o migraci slabou nebo silnou (viz kapitola 1.7.3). Z konkrétních způsobů silnou migraci provádí jen **plná migrace** a **přenos kompletního stavu**. Slabá migrace se používá častěji, protože ji lze jednodušeji realizovat.

2.4.1 Plná migrace

Celý agent je přenesen, aniž by se změnilo jeho prostředí. Přenos se týká nejen kódu a kompletního stavu (dat a zásobníku), ale i služby hostitele a operačního systému, jako jsou například otevřené soubory. Systém musí definovat primitivum (například ve formě funkce) pro migraci, které provede migraci a agent pak pokračuje v činnosti (viz výpis 2.1). Přenos je tak naprosto transparentní.

```
doSomething();
migrate(somewhere);
doSomethingOnNewHost();
```

Výpis 2.1: Princip plné migrace

Tato metoda je sice ideální, bohužel je však velmi obtížně realizovatelná. Musí ji totiž zajišťovat přímo operační systém. V obecném operačním systému je prakticky nerealizovatelná.

2.4.2 Přenos kompletního stavu

Situace se výrazně zjednoduší, pokud požadujeme jen přenos samotného agenta a není nutné zachovávat jeho prostředí. Dostačující je přenést kód a kompletní stav agenta. Vyvolání je zcela shodné jako u plné migrace.

Tato metoda vyžaduje, aby byl hostitel schopen přečíst celý zásobník a poslat jej cílovému hostiteli, který zásobník opět vytvoří a po obnovení zbytku stavu spustí běh v tom bodě, kde byl přerušen.

Hostitel však zpravidla tuto možnost nemá. Výjimkou jsou systémy, v nichž je agent prováděn speciálním interpretrem, který má vše potřebné pro tuto operaci zabudováno (např. *Agent TCL*, *Telescript*, *ARA*). Při použití jiného jazyka (například Javy) je nutné použít speciální preprocesor přetvářející kód tak, že se ve skutečnosti použije některá z dalších metod [Hoh1998]. Tento postup sice zjednoduší práci programátora agenta, ale otázkou zůstává, zda se to vyplatí.

Navíc tento způsob migrace je problematický u agentů, jejichž činnost probíhá ve více vláknech. Ve chvíli, kdy jedno z nich provede migraci, nemusí být ostatní připraveni. Najednou bez zjevné příčiny se změní prostředí. Bylo by tedy nutné migraci nějakým způsobem synchronizovat. Většina systémů používajících tuto metodu nemá problém, protože v nich použití vícevláknového agenta není možné.

2.4.3 Přenos neaktivního agenta

Když se agent rozhodne migrovat, oznámí to svému hostiteli. Migrace se však nespustí okamžitě. Agent musí ukončit veškerou svoji činnost (tak, aby se jeho data zachovala v konzistentním stavu). Teprve když jsou všechna vlákna agenta ukončena, provede hostitel migraci. Tímto postupem se zcela eliminuje potřeba

```
public void running() {
    // nějaká činnost...
    doSomething();
    // zde došlo k rozhodnutí o migraci
    migrateTo(newHost);
    // nyní je agent označen pro migraci, ale stále ještě zůstává na místě
    doSomeFinishing();
    return;
    // Teprve po ukončení dojde k samotné migraci
}

public void continueOnNewHost() {
    // V pořádku, agent je již na novém hostiteli
}

public void migrationFailed(MigrationFailureEvent evt) {
    // Migrace se nepodařila
}
```

Výpis 2.2: Princip migrace neaktivního agenta

přenášet stavy (zásobníky) vláken a stačí přenést jen kód a data. Na novém hostiteli se pak předem definovaným způsobem (typicky vyvolání funkce nebo metody) nový spustí agent.

Teprve po úspěšném dokončení přenosu se starý agent zruší. Pokud dojde k chybě, opět se původní agent spustí a předá se mu zpráva, že migrace neproběhla.

Zdánlivě zde vzniká časový interval, kdy agent existuje ve dvou exemplářích na dvou různých místech. Ve skutečnosti není stará kopie aktivní a nemůže tedy provádět žádnou činnost (je ve stavu *migrating* viz obrázek 2.4). Nevýhodou této metody zůstává vyšší složitost kódu agenta.

2.4.4 Vytvoření kopie

Často se používá varianta předchozí metody, která se z pohledu agenta více blíží metodě přenosu kompletního stavu. Rozdíl je v tom, že při zavolání funkce (metody) migrate se migrace provede *okamžitě*. Na novém hostiteli se tak vytvoří kopie, která se spustí stejně jako při metodě přenosu neaktivního agenta.

Po provedení přenosu dojde (na původním místě) k návratu ze stavu *migrate*. Agent by se měl okamžitě ukončit, neboť podle veškeré logiky již zde není! Než se tak stane, existuje ve dvou exemplářích na dvou místech.

Tato metoda je poměrně často používaná (používají ji např. systémy *Aglets* a *Voyager*), přičemž je ponecháno na dobré vůli programátora, aby agent existoval dvojitě jen co nejkratší okamžik.

Navíc, stejně jako u metody přenosu plného stavu je problém s přenosem agentů běžících v několika vláknech současně. Programátor musí zajistit synchronizaci tak, aby v okamžiku migrace byl agent v konzistentním stavu.

```
public void running() {
    // nějaká činnost...
    doSomething();
    // zde došlo k rozhodnutí o migraci
    try {
        prepareForMigration();
        migrateTo(newHost);
        // Migrace se podařila, nová kopie běží
        return; // Zde se musím ukončit!
    } catch (MigrationException ex) {
      // Migrace se nepodařila
    }
    // ...
}

public void continueOnNewHost() {
    // V pořádku, agent je již na novém hostiteli
}
```

Výpis 2.3: Princip migrace vytvořením kopie

2.4.5 Reprezentace stavu pro přenos

Aby mohl být agent přenesen pomocí síťových služeb systému, musí být celý zakódován do vhodné formy. Tato zakódovaná forma musí obsahovat všechny potřebné informace, aby se z nich dal na cílovém hostiteli agent opětovně vytvořit a spustit.

Kromě jazyků, ve kterých to nemá smysl, se kóduje a přenáší odděleně programový kód a stav agenta. U obou je nejenom jiná forma dat, ale také jiná sémantika přenosu. Proto je přenosu programového kódu věnována celá další podkapitola.

Stav agenta je jedinečný a nedělitelný (všechny informace v něm obsažené jsou nutné pro jeho rekonstrukci). Proto je jeho přenos prováděn najednou, jako jediný blok dat, který je přenesen mezi hostiteli. Výsledkem zakódování stavu pro přenos je právě vytvoření takového bloku dat (typicky realizovaného jako pole bytů).

Při kódování informace je nutné brát v úvahu, že hostitelé mohou být realizováni na různých platformách (samozřejmě jen pokud to systém umožňuje). Především je nutné brát v úvahu různé pořadí bytů ve slově (*byte ordering*: *big endian* nebo *little endian*).

Kódování stavu může provádět:

- **agent** – při migraci hostitel zavolá funkci (metodu), která zakóduje celý stav. Tato metoda je velmi nepraktická a prakticky se nepoužívá,
- **hostitel** – zakódování stavu pro přenos provede hostitel (přičemž může dát agentovi šanci si některé části zakódovat vlastním způsobem, jako je tomu v Javě). Způsob provedení je zcela na něm.

Java

Ačkoliv způsob kódování stavu je interní záležitostí systému, je vhodné použít standardní metodu (pokud je k dispozici). V jazyce Java je k dispozici tzv. *serializace*. Umožňuje standardním způsobem uložit objekty (instance tříd) a potom je opět obnovit. Formát je přesně definován, ale třída si může nadefinováním speciální metody implementovat svůj vlastní formát uložení.

Nedostatkem standardní serializace je velká citlivost na změnu kódu (definice tříd). Pokud se změní některá třída, její serializovaný obsah nemusí být dekódovatelný (obnovitelný). Proto je při použití této technologie nutné zajistit, aby se kód agenta nemohl během jeho života změnit.

Alternativou je použít jinou formu perzistentního ukládání objektů, například do (v současné době velmi populárního) formátu *XML*. Bohužel však žádná taková technologie není standardizována. V současné době probíhá proces standardizace ukládání komponent *JavaBeans* do *XML*, ale ještě není dokončena[1]

[1] Z tohoto důvodu zde nelze uvést odkaz, protože neexistuje pevné umístění informací na stránkách firmy Sun.

a navíc by přinesla výrazné komplikace pro autora (agent by se musel skládat výhradně z *beanů*).

2.4.6 Přenos programového kódu agenta

V některých jazycích lze obtížně rozlišit kód od dat (např. *LISP*). Při jejich použití tedy není nutné (a ani žádoucí) speciálně přenášet kód, protože se přenese jako součást stavu.

Ve většině jazyků je však kód zcela oddělený a nezávislý na stavu agenta (tj. nemění se jeho činností[2]). Tento kód je navíc společný pro mnoho agentů stejné aplikace, proto ho není nutné přenášet vždy.

Sémantika přenosu kódu agenta

Součástí kódu nutného k běhu agenta jsou i různé knihovny. Některé z nich se dají považovat za zcela standardní a jsou povinně k dispozici na hostiteli. U ostatních je však nutné zajistit, aby v případě potřeby k dispozici byly. Jednoduchý a elegantní (a také jediný prakticky používaný) způsob, jak toho dosáhnout, je přidat tyto knihovny ke kódu agenta. V praxi se používají tři přístupy k přenosu kódu, které jsou závislé na jeho umístění:

Kompletní přenos: Při migraci se všechen kód jednoduše přenese (společně se stavem) viz obrázek 2.5a). Tento přístup je sice nejjednodušší, ale velmi neefektivní (přenášejí se i knihovny, které již na hostiteli jsou) a prakticky se používá jen zřídka.

Jedinou výraznou výhodou je nezávislost na funkci sítě – jakmile proběhne přenos, kód bude fungovat i kdyby přestala fungovat počítačová síť.

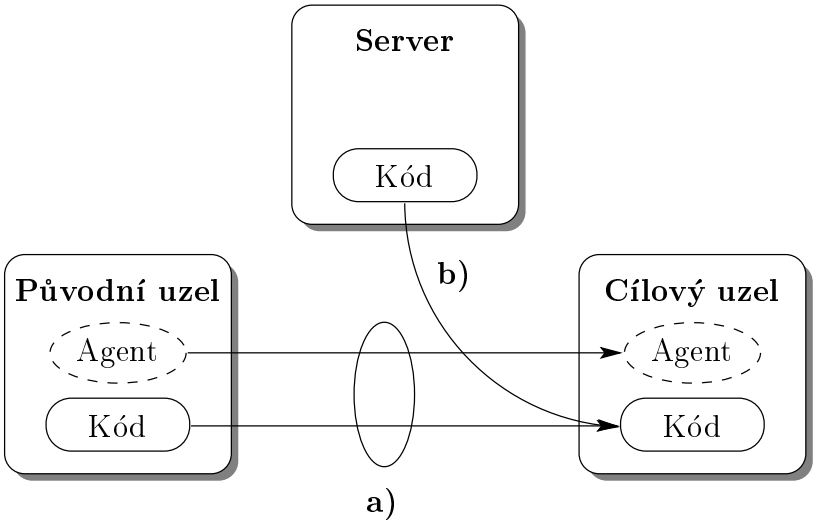
Přenos kódu na vyžádání: Při migraci samotné se nepřenáší žádný kód. Cílový hostitel si sám zjistí, kterou část kódu (typicky knihovny, případně jednotlivé třídy) potřebuje a vyžádá si ji.

Tento přístup je výrazně efektivnější než kompletní přenos, ale vyžaduje udržování spojení mezi oběma hostiteli. Navíc nastávají problémy, pokud agent pokračuje v migraci na další hostitele: požadavky o kód se mohou řetězit a celá situace se stává velmi citlivou na případné výpadky sítě.

Přenos kódu odkazem: Určitou variantou předchozího přístupu je poslání pouhého odkazu (adresy, například ve formě *URL*) na místo, kde je kód umístěn (*code repository*, *codebase server*) viz obrázek 2.5b). Způsob naložení s touto informací je pak zcela ponechán na cílovém hostiteli. Může si buď vyžádat veškerý kód najednou, nebo pouze ty části, které potřebuje.

[2] I kdyby to jazyk dovoloval, modifikace vlastního kódu je velmi špatná programovací technika.

Pokud agent pokračuje v migraci na další hostitele, odkaz na umístění kódu může zůstávat stále stejný.



Obrázek 2.5: Přenos kódu agenta

Reprezentace pro přenos

Každý jazyk definuje formát (nativní kód, bytecode, zdrojový text), ve kterém je kód uložen v souboru na disku. Tento formát je shodný pro všechny platformy na kterých může být provozován, takže pro přenos po síti je zcela postačující přenést tyto soubory.

2.4.7 Realizace přenosu

Samotný přenos po síti může být realizován mnoha způsoby, nejčastější jsou:

- **vlastní protokol** – systém si definuje vlastní protokol, který je navržen speciálně pro tyto účely. Realizace je však složitější a mohou nastat problémy např. při použití za firewallem.
- **využití standardních protokolů** – typickým příkladem je *FTP* nebo *HTTP*. Výhodou je, že tyto protokoly jsou standardní a rozšířené.
- **RPC nebo RMI** – vzdálené volání procedur (metod) sice není příliš vhodné pro přenos většího množství dat, ale provádí téměř kompletní zakrytí síťové vrstvy.

  Tyto protokoly lze navíc použít i pro komunikaci, takže není nutné používat několik specializovaných pro různé účely.

2.5 Zdroje a služby

Aby mohl agent na hostiteli provádět nějakou činnost, potřebuje k ní mít přiděleny různé zdroje. Má sice vždy přidělenou operační paměť a procesor (ve formě vlákna či procesu), ale to by postačovalo jen k provádění samostatných výpočtů bez jakékoli vazby na okolí. Takový mobilní agent postrádá jakýkoliv smysl.

Pro smysluplnou funkci agentů jim musí hostitel poskytovat různé zdroje. Pojmem **zdroj** (*resource*) se označuje:

- hardware počítače, na kterém hostitel běží, včetně periferií,
- přístup k operačnímu systému,
- přístup k datům v různých formách (např. k souborům na lokálním disku),
- komunikace s jinými programy (např. databázové servery).

Mezi typické příklady patří soubory na lokálním disku, databáze nebo různé periferie. Právě přístup k lokálním zdrojům je ve většině případů smyslem existence mobilního agenta.

V mnoha systémech není přístup ke zdrojům vyřešen a je plně v režii programátora. Agent tak může využívat jen standardních možností programovacího jazyka (např. přístup k souborům na disku) nebo různých knihoven (např. *JDBC* pro přístup k databázi v Javě). Nevýhody tohoto „řešení“ jsou:

- Hostitel má jen omezené možnosti, jak kontrolovat přístup ke zdrojům, u některých dokonce tuto možnost vůbec nemá. Vzniká tím prostor pro případný útok.
- Neexistuje standardní metoda, kterou by mohl agent zjistit, jaké zdroje jsou k dispozici.
- Pro používání většiny zdrojů je nutné použít speciální knihovnu. Agent musí předpokládat, že je knihovna k dispozici na hostiteli (pak musí být ošetřen případ, že není), nebo mít vlastní verzi dané knihovny (čímž se riskuje možná nekompatibilita, nemluvě o zbytečném plýtvání místem).
- Každý zdroj má naprosto jiný způsob, jakým se s ním pracuje a jak se k němu přistupuje. Pokud jich agent používá více, kód se komplikuje.

Pro sjednocení přístupu ke zdrojům se zavádí tzv. **služby** (*services*). Ty slouží jako prostředník mezi agentem a zdroji a agent by neměl ke zdrojům přistupovat jinak než prostřednictvím služeb.

2.5.1 Realizace služeb

Služba je entita, se kterou lze přesně definovaným způsobem komunikovat. Každá služba poskytovaná hostitelem je určitého typu (třídy). Typ služby určuje, jak se chová a k jakému typu zdroje zajišťuje přístup. Všechny operace prováděné nad službou jsou tímto typem přesně definovány. Každá konkrétní služba (instance) musí tyto operace umožňovat

Výše zmíněné charakteristiky odpovídají charakteristikám třídy (nebo *interface*) v objektově orientovaných jazycích. Je tedy logické, že služby se implementují jako objekty. Typu služby pak odpovídá třída nebo *interface* (v jazycích, které podporují tuto koncepci, jako je např. Java).

V neobjektovém prostředí bude realizována jinak, ale se stejnou sémantikou. Koncepce služeb je však typická pro systémy implementované v objektových jazycích a proto je v dalším textu předpokládán právě tento případ.

Od jednoho typu může existovat více instancí, které zajišťují přístup k různým zdrojům stejného typu (pokud to má smysl, např. více databází). V dalším textu je pojmem služba míněna instance služby.

Každá služba poskytovaná jednomu agentovi je jednoznačně identifikována (jakým způsobem závisí na implementaci systému).

2.5.2 Poskytování služeb

Agent nemůže přistupovat přímo ke zdrojům, musí vždy použít patřičnou službu. Po svém spuštění na hostiteli nemá k dispozici žádné služby. Hostitel mu však nabízí několik operací pro přidělování služby:

Zjištění množiny nabízených služeb (*Available Services*): Hostitel na požádání sdělí, jaké služby nabízí. Nabídka je ve formě množiny (pole) identifikátorů služeb. U každé služby musí být jasné, jakého je typu (jednoduchý a elegantní způsob je ten, že typ je součástí identifikace).

Pro zjednodušení práce může hostitel nabízet operaci zjištění, zda je jedna konkrétní služba k dispozici.

Přidělení služby (*Service Acquisition*): Agent si zažádá o přidělení některé z nabízených služeb. Hostitel mu ji přidělí (může však také přidělení odmítnout). Agent tak získá ke službě (reprezentované objektem) přístup a může ji libovolně využívat.

Uvolnění služby (*Service Release*): Pokud agent již službu nepotřebuje, měl by ji uvolnit a od toho okamžiku ji již nesmí používat. Hostitel tak může uvolnit veškeré zdroje, které služba využívala .

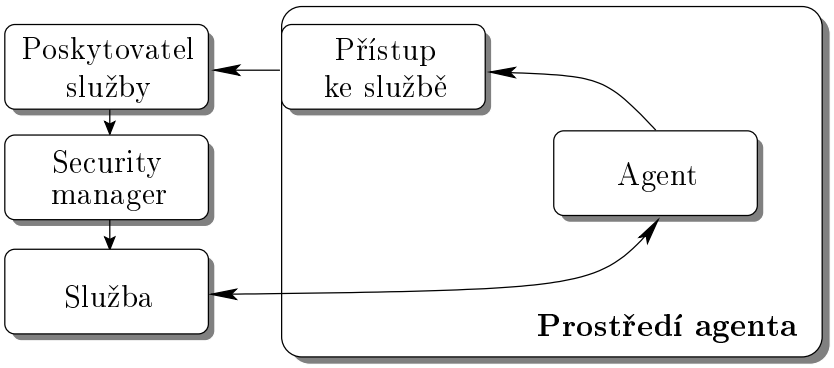
Postup agenta při používání služby je pak následující:

1. Agent dotazem na hostitele zjistí, zda je služba k dispozici.
2. Nechá si službu přidělit.
3. S přidělenou službou provádí cokoliv potřebuje a co mu tato služba dovolí.
4. Službu opět uvolní.

2.5.3 Řízení přístupu

Významným důvodem pro implementaci služeb je zajištění kontroly nad přístupem ke zdrojům. Ta je velice důležitá pro omezení rizika napadení agentem viz 2.8.2. Řízení a kontrola přístupu probíhá hned na dvou úrovních:

- Hostitel má kontrolu nad tím, zda určitou službu přidělí. Může tak tímto způsobem znemožnit agentovi úplně přístup ke zdroji.
- Služba sama provádí jemnější omezení přístupu. Při každé operaci zkontroluje, zda je povolená a pokud není, odmítne ji provést. Navíc může vzít bezpečnostní aspekty v úvahu a změnit parametry operace nebo výsledek (například může skrýt existenci určitých souborů a podobně).



Obrázek 2.6: Řízení přístupu ke službě

Funkce řízení přístupu je znázorněna na obrázku 2.6. Důležitými entitami na tomto obrázku jsou dvě speciální součásti hostitele:

Poskytovatel služby: Provádí nabídku služeb a zpracovává požadavky na přidělení.

Security manager: Provádí všechny bezpečnostní kontroly a činí rozhodnutí na nich založená. Obsahuje celou bezpečnostní politiku hostitele.

Agent je uzavřen ve svém prostředí a komunikovat může jen s *poskytovatelem služeb*. Jeho požádá o přidělení služby. Poskytovatel požadavek zkonzultuje se *security managerem* a zjistí tak, zda má službu přidělit nebo nikoliv (čárkovaná šipka). Pokud ano, předá agentovi referenci na tuto službu.

Agent pak službu využívá prováděním operací (voláním metod). Při každé takové operaci si služba u *security manageru* ověří (čárkovaná šipka), zda je možné tuto operaci provést. Pokud ne, vrátí agentovi chybu.

Zmíněná rozhodnutí o tom, jaký má agent přístup ke službě, se nazývají **autorizace** (*authorization*). Ta bývá založena na identitách, kterými se agent prokazuje, především pak na identitě *vlastníka*.

2.5.4 Změna nabízených služeb

Množina nabízených služeb nemusí být v čase konstantní. Dochází k jejich přidávání a odebírání.

O těchto změnách musí být agent informován. Mohl by samozřejmě průběžně dotazy zjišťovat, jaké služby jsou momentálně k dispozici a tyto údaje porovnávat (tzv. *polling*), ale tato činnost je značně neefektivní a komplikovaná. Jednodušší a efektivnější je použít asynchronní události (*events*).

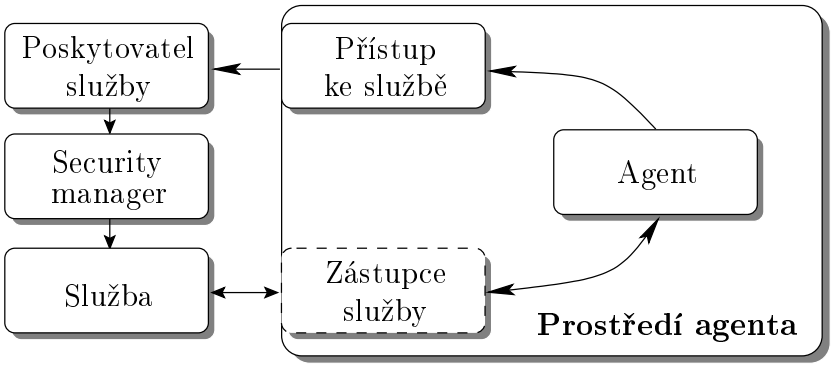
Agent hostiteli oznámí, že chce být upozorněn, když je dostupná nová služba (upozornění může být omezeno na konkrétní typ nebo dokonce instanci). Potom může pokračovat v jiné činnosti, ke které tuto službu nepotřebuje, nebo může nedělat nic a čekat (pokud ji potřebuje opravdu nutně). Jakmile je tato služba k dispozici, hostitel pošle agentovi zprávu. Ten ji zpracuje, ověří že je to přesně ta, kterou chce, a nechá si službu přidělit.

Stejná metoda funguje i v opačném případě, tj. když služba přestane být nabízena. V takovém případě je však situace komplikovanější, protože agent takovou službu *musí* uvolnit. Je tedy agentovou povinností, aby si nechal posílat zprávy o rušených službách a reagoval na ně tím, že tyto služby uvolní.

Problémy nastávají v okamžiku, kdy agent svoji povinnost nedodrží a neuvolní službu, pokud již není déle k dispozici. U některých služeb je taková situace kritická, protože je nutné uvolnit alokované zdroje. Uvolnění takových služeb je tedy nutné zajistit explicitně pro případ zákeřných (*malicious*) nebo chybných agentů.

Vynucené uvolnění služeb lze zajistit použitím **zástupců** (*proxy*). Zástupce zastupuje jiný objekt (v tomto případě instanci přidělené služby), implementuje stejný *interface* (stejnou třídu) a všechna volání přesměruje na objekt, který zastupuje. Z vnějšího pohledu tak není možné rozeznat, že se nejedná o původní objekt.

Při každém přidělení služby hostitel vytvoří pro tuto službu zástupce a předá jej agentovi. Ten pak může používat zástupce jako pravou službu, jak je vidět na obrázku 2.7. Jakmile dojde ke zrušení služby (již není dále nabízena), hostitel nastaví zástupce do stavu, kdy nepředává další volání a ani neudržuje referenci



Obrázek 2.7: Přístup ke službě prostřednictvím zástupce

na původní objekt. Této operaci se říká **rozpojení** (*disconnect*). Od toho okamžiku nelze zástupce používat (každý pokus skončí výjimkou) a hostitel tak může považovat službu za uvolněnou.

2.6 Identifikace a ověření identity

Jmenné služby jsou jednou z nejdůležitějších částí systémů mobilních agentů. Bez nich by bylo velice obtížné sledovat agenty při jejich migraci. Nejjednodušší metodou je použití *itineráře*. Ten obsahuje odkazy na všechny uzly, které agent během svého života navštíví. Jestliže chce vlastník komunikovat s agentem, postupně se dotáže všech uzlů na jeho přítomnost. Tento přístup má ale svá omezení, protože agenti nemohou migrovat zcela volně. Vždy musí být v kontaktu se svým vlastníkem.

Aby mohli agenti zcela volně migrovat, je potřeba použít *jmenných služeb* systému. Jmenné služby jsou pak zodpovědné za neustálé sledování migrace všech agentů v systému. Agent oznámí své nové umístění pokaždé, když se přesune na nový uzel.

Jinou metodou je **zřetězení uzlů** (*chaining*). Její princip spočívá v tom, že každý uzel poskytuje služby pro **předávání dotazů** (*forwarding service*). Buď jsou všechna volání agenta přesměrována na jeho nové umístění nebo je vlastníkovi vráceno nové umístění agenta a ten pak komunikuje přímo s ním. Tato metoda se může použít, pokud nastává velká prodleva mezi migrací agenta a oznámením jeho nového umístění v systému, nebo pokud se očekává malá interakce agenta se svým vlastníkem. Nevýhodou tohoto řešení je delší prodleva při první komunikaci s agentem. Vlastník by se mohl domnívat, že došlo k chybě v komunikaci, a přitom je jeho dotaz přenášen přes mnoho uzlů vzniklého řetězu.

2.6.1 Jména

Jméno je údaj, který označuje právě jednu entitu. Dvě různé entity mají zcela jistě různá jména. Tato vlastnost se nazývá jedinečnost (*uniqueness*). Jedinečnost musí být zajištěna v celém prostoru, kde se toto jméno používá. Opačným směrem však podobná závislost nemusí platit, jsou možné různé poměry:

1:1 Entita má přiřazeno právě jedno jediné jméno, dvě různá jména označují různé entity. Tento typ se označuje jako **jednoznačné jméno** nebo též **identifikátor** (*identifier*, *ID*).

1:N Entita může mít více jmen a tedy různá jména mohou označovat stejnou entitu. Všechna tato jména jsou rovnoprávná. V tomto případě je problém posoudit, zda dvě jména označují stejnou entitu.

Aby se tento problém vyřešil, jednomu ze jmen se přiřadí výsadní postavení. Tato forma se nazývá **kanonická** nebo **standardizovaná** (*canonical form*) a ostatní jména se nazývají **aliasy**. Alias lze použít kdekoliv, ale musí existovat způsob, kterým se převede do kanonické formy. Tu pak lze použít k porovnání jmen, zda označují stejnou entitu.

Příkladem takových jmen jsou DNS jména nebo URL adresy.

Důležitou vlastností jména je také **perzistence**. Jméno musí stále označovat tutéž entitu po celou dobu její existence.

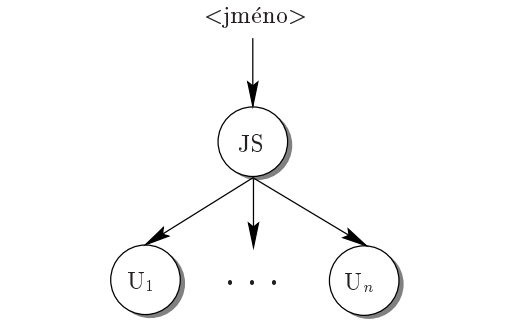
Na jména se často kladou i další nároky. Ne zcela zanedbatelný je požadavek, aby jména (alespoň některá) byla čitelná pro člověka (*human readable form*). Takové jméno musí být implementováno jako text (znakový řetězec).

Typy jmen

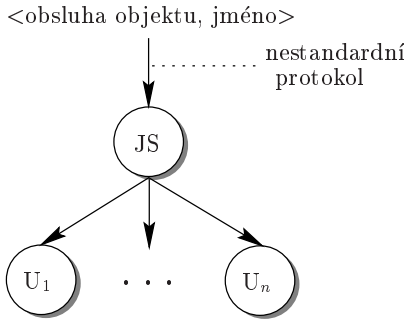
Jednoduché jméno (globální jméno) – standardní systém
Nejběžnější konvencí jmen je pravděpodobně použití jednoho objektu Reference (dlouhé jméno), který obsahuje pouze jednu položku RefAddr (krátké jméno) odkazující na jmenný server. Tento případ se označuje jako *globální jméno*[3]. Jmenný server je tedy jediným uzlem, který zná konkrétní umístění agenta.

Více jmen (záložní jména)
Nejčastějším případem jmen v systémech odolných proti poruše je více krátkých jmen v dlouhém jméně, která odkazují na různé jmenné servery. Jestliže primární server vypadne, je vybrán další záložní. Jména mohou také odkazovat na stejný jmenný server, ale pomocí různých protokolů.

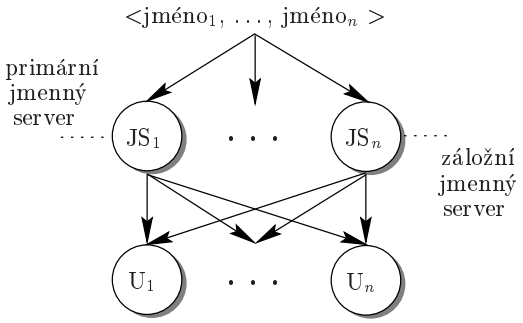
[3]Představa lokálního jména je velmi podobná globálnímu. Rozdíl je pouze v tom, že klient a jmenný server běží na stejném uzlu pro rezoluci lokálních jmen



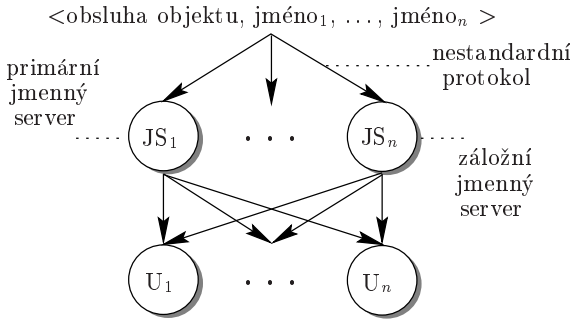
jednoduché jméno - standardní systém (globální jméno)



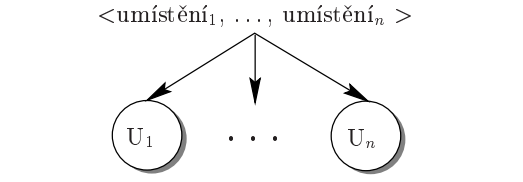
jednoduché jméno - nestandardní systém



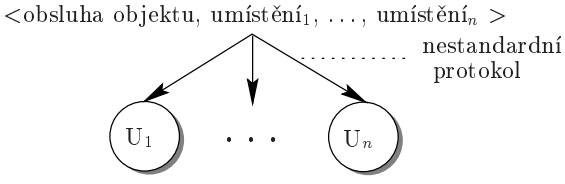
více jmen - záložní jména



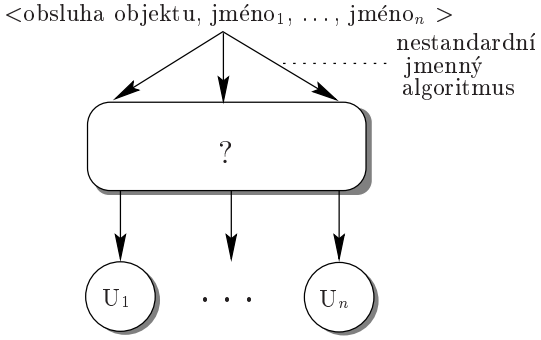
více jmen - nestandardní systém



pojmenování pomocí umístění (itinerář)



pojmenování pomocí umístění nestandardní systém



uživatelem definovaná jména

Obrázek 2.8: Typy jmen

Pojmenování pomocí umístění (itinerář)
Pro některé typy aplikací není jméno nezávislé na umístění nutností. Jestliže je uzel, na který agent migruje, znám předem, lze jako krátké jméno agenta použít jména všech uzlů. Uzly jsou postupně dotazovány, dokud není agent nalezen. Tento případ pojmenování je často označován jako *itinerář*.

Pojmenování pomocí umístění (itinerář) – nestandardní systém
V případě itineráře se může použít nestandardních protokolů. Dlouhé jméno pak navíc obsahuje jméno obsluhy objektů ObjectFactory a její umístění. V případě více uzlů s různými přístupovými protokoly jméno obsahuje více obsluh objektů.

Jednoduché jméno (globální jméno) – nestandardní systém
Přístup pomocí jednoho jména nezávislého na umístění může být rozšířen o použití nestandardních jmenných protokolů. Dlouhé jméno pak obsahuje jméno obsluhy objektů ObjectFactory i s jejím umístěním. Obsluha objektů je použita při komunikaci se jmenným serverem pro získání současného umístění agenta.

Více jmen – nestandardní systém
V systémech s více jmennými servery může být také použito nestandardních protokolů. Stejně jako v případě nestandardního jednoduchého jména, je třeba přidat jméno obsluhy objektů pro obsluhu protokolů. Každé jméno odkazuje na jiný jmenný server nebo specifikuje jiný protokol.

Uživatelem definovaná jména
Ve speciálních případech může být použit speciální algoritmus pro vyhledávání jmen, který je rozdílný od všech předešlých. V tomto případě pak obsluha objektů nezajišťuje pouze obsluhu protokolů, ale také uživatelem definované konvence jmen.

2.6.2 Identifikace částí systému

Systém musí zajistit identifikaci všech jeho částí:

Uzel (počítač): Uzel se označuje svou síťovou adresou a síťovým jménem. Jedinečnost IP adresy a DNS jména musí být zajištěna správcem sítě. V případě IP adresy je navíc většinou zajištěna jednoznačnost (nemusí to však platit vždy).

Hostitel: Pojmenování hostitele je komplikovanější a výrazně se u různých systémů liší. Hostitel zahrnuje uzel a program, který na tomto uzlu běží a zajišťuje funkci hostitele. Tento program nazývám pojmem **stroj** (*engine*).

Jméno hostitele pak může být určeno různými kombinacemi:

- Jméno uzlu je přímo jméno hostitele. Tento přístup je nejjednodušší (mimo jiné protože jedinečnost je již zaručena) a asi nejčastější. Omezením je, že na jednom uzlu může být současně provozován pouze jeden jediný stroj. Většina systémů však s tímto omezením počítá a rovnou definuje uzel=stroj.
- Jméno hostitele je určeno pouze jménem stroje a nemá tak žádnou souvislost s uzlem. Výhodou je, že umístění stroje je zcela libovolné. Jedinečnost jména však musí zajistit administrátor systému, navíc pro uživatele mohou tato jména být matoucí (uživatel očekává, že agent migruje na určitý uzel, a tato informace mu ve jméně hostitele chybí).
- Kombinací jména uzlu a jména stroje vznikne jméno hostitele. Výhody jsou:
  - Na uzlu může být provozováno více strojů (tedy více hostitelů). Tato možnost je však pro praktické použití zbytečná, smysl má spíše pro testovací a demonstrační účely.
  - Pro zajištění jedinečnosti postačuje, aby bylo jméno stroje jedinečné v rámci daného uzlu. Oproti předchozímu typu jména se tak zjednoduší administrace.
  - Jméno hostitele obsahuje jméno uzlu. Zjednoduší se tak orientace uživatele.

Součásti hostitele: Hostitel často mívá interní strukturu, kdy je rozdělen do několika částí (nazývají se různě). Tyto části většinou mívají jméno čitelné pro člověka (řetězec) a jedinečnost stačí zajistit v rámci jednoho hostitele.

Služby: Protože jsou služby lokální pro každého hostitele, stačí zajistit jedinečnost pojmenování pouze v rámci tohoto hostitele. Tento požadavek se ještě sníží, pokud jsou služby nabízeny zvlášť pro každého agenta. Potom stačí zajistit jedinečné jméno jen v množině služeb nabízených jednomu agentovi.

Agent: Identifikace agenta je komplikovanější než u ostatních částí systému. Proto je tomuto problému věnována samostatná část.

Uživatel: Uživatelské jméno závisí na formě ověřování identity uživatele. Teoreticky by bylo možné použít „reálné“ jméno (jméno, příjmení a další jednoznačně identifikující informace). V praxi se však takový údaj velmi obtížně zpracovává a ještě obtížněji ověřuje.

Některá používaná jména jsou:

- *Jednoduché uživatelské jméno* (jedno slovo bez mezer a speciálních znaků s omezenou délkou). Takové jméno se používá v operačních systémech. Pokud má být zajištěna jedinečnost v rámci celého systému,

musí být uživatelské jméno doplněno údajem o tom, odkud toto jméno pochází (jméno počítače, adresa jmenné služby a podobně).

- *Jméno uživatele v globální jmenné službě DAP* (např. *LDAP*). Takové jméno se používá v certifikátech podle standardu $X.509$, přičemž jeho jedinečnost je zaručena certifikační autoritou, která certifikát podepsala.

Klient: Pokud je klient samostatná entita, musí být nějakým způsobem identifikovatelná a musí tedy mít jméno. Může se jednat například o kombinaci jména uzlu, na kterém běží se jménem uživatele. Jinou variantou je, že se prokazuje certifikátem a tudíž používá jméno majitele tohoto certifikátu.

2.6.3 Identifikace agenta

Jméno agenta musí být jednoznačné po celé síti (na všech uzlech, hostitelích), na které systém funguje. V tomto ohledu je situace obdobná jako u hostitele.

Oproti ostatním částem systému je rozdíl především v tom, že agenti jsou vytvářeni dynamicky. Jméno je agentovi přiřazeno až v okamžiku vytvoření (nebo vyslání). Navíc je nutné, aby po ukončení agenta nebylo systémem jeho jméno opět přiděleno. Aplikace komunikující se starým agentem by navázaly spojení s novým (z jejich hlediska by se jednalo o chybu).

Jméno může přidělovat:

- **klient** – jméno je agentovi přiděleno ihned v okamžiku jeho vytvoření. Nevýhoda tohoto postupu je ta, že se klient musí postarat o to, aby toto jméno bylo jedinečné. Tím se klient výrazně komplikuje. Navíc, praktické výhody oproti přidělování hostitelem nejsou žádné.

- **hostitel** – při přijmutí vyslaného agent hostitel tomuto agentovi přidělí jméno, které sdělí klientovi. Ten se tak nemusí o jedinečnost tohoto jména starat (zaručuje ji hostitel).

  Nevýhodou je, že agent nemá žádnou kontrolu nad jménem a nemůže tedy do jména dát nějakou charakteristiku tohoto agenta (např. jméno aplikace, funkci, jméno uživatele...). Výsledné jméno pak pro uživatele má minimální hodnotu (samozřejmě, pokud jméno vůbec není v čitelném formátu, tato nevýhoda odpadá). Velice jednoduché řešení je však takové, že klient takovouto informaci pošle hostiteli, který ji pak může přidat do jména.

Dalším problémem při generování jmen je zajištění jejich jedinečnosti v rámci celého systému. Existují tři přístupy přidělování jmen:

- **jmenné služby** – pokud je v systému použit nějaký systém jmenných služeb, lze jich využít. Algoritmus pak vypadá takto:

1. Hostitel vytvoří jméno.
2. Ve jmenné službě ověří, zda již existuje nebo existoval.
3. Pokud ne, toto jméno přidělí agentovi a zaregistruje ho.
4. Jméno upraví (např. přidáním čísla) a pokračuje bodem 2.

Problémy nastávají v okamžiku, kdy se o přidělení stejného jména pokouší více hostitelů. Přístup ke jmenné službě je nutné zajistit tak, aby zpracování bodů 2 a 3 proběhlo jako transakce. Tím se celý postup výrazně komplikuje.

Mezi další nevýhody patří především nutnost evidence jmen ukončených agentů (jména nejsou znovupoužitelná) a citlivost na výpadek sítě.

- **centralizované přidělování** – podobně jako v předchozím případě, existuje centrální jmenná služba evidující všechna přidělená jména agentů. Tato jmenná služba však přímo sama přiděluje jména a zaručuje tak jejich jedinečnost.

  Centralizovaná služba má tu základní nevýhodu, že její výpadek způsobí ochrnutí celého systému.

- **složené jméno** – nejvhodnější metoda je plně distribuované přidělování, kdy není nutné nijak komunikovat. S výhodou se dá využít toho, že jméno hostitele je jedinečné. Ke každému agentovi stačí pak vygenerovat identifikátor v rámci hostitele. Jejich složením pak vznikne globálně jedinečné jméno agenta.

  Typickým příkladem generovaného identifikátoru je pořadové číslo.

  Výsledné jméno však neobsahuje žádnou informaci o agentovi. Pro uživatele je pak kombinace jména hostitele s nějakým „nesmyslným“ údajem zcela nečitelná. Z tohoto důvodu je vhodné jméno ještě doplnit o další část, kterou dodá klient.

2.6.4 Autentifikace

Jako všechny distribuované aplikace v potencionálně nebezpečném prostředí i systém mobilních agentů může být napaden. Bezpečnosti je věnována samostatná část textu, zde je pouze jeden aspekt související s identifikací: ověření identifikace neboli **autentifikace** (*authentification*).

Ověřování se týká všech součástí systému. V některých případech ho není nutné provádět (např. hostitel může přijímat agenty vyslané z jakéhokoliv klienta), v jiných případech je tato otázka naopak klíčová (např. při ověřování identity majitele agenta). Bez ověření identity nelze entitě důvěřovat, protože se může jednat o maskovaného útočníka.

Všechny metody autentifikace vyžadují nějaký druh informace dokazující, že entita je opravdu to, za co se vydává. Tyto důkazy jsou především:

- Informace, jejichž pravost a souvislost s entitou je nějakým způsobem (hardware, operační systém, síťové prostředí) zaručena. Jedná se například o síťové adresy, informace poskytnuté operačním systémem, externí autentifikace (čipové karty) a podobně. Tento způsob autentifikace přesouvá odpovědnost na někoho (něco) jiného.

- Nějaký druh tajné informace[4], kterou nemá k dispozici nikdo jiný. Tuto informaci pak při ověřování buď přímo poskytne, nebo z ní nějakým algoritmem vygeneruje důkaz.

  Tajné informace jsou především:

  - **hesla** – tento způsob autentifikace je nejčastější a většinou se používá jako primární — teprve po ověření kombinace jméno/heslo získá uživatel přístup ke svým kryptografickým klíčům a certifikátům.

    Zabezpečení autentifikace pomocí hesel musí být zaručeno na úrovni operačního systému a síťového přenosu (bezpečný kanál). Odposlouchávání a zneužívání hesel patří k nejčastějším druhům bezpečnostních útoků.

  - **šifrovací klíč** – Ověření identity proběhne tak, že se identifikace (jméno, případně další informace) zakryptují. Protože bez znalosti šifry nelze vytvořit správnou zašifrovanou formu, prokáže tím vlastnictví šifry a tedy i správnou identitu.

    Takto jednoduchý postup je však snadno napadnutelný (především odposloucháváním a opakováním) a tedy prakticky nepoužitelný. Používané algoritmy jsou výrazně komplikovanější a většinou se využívá asymetrických šifer.

  - **digitální podpis** – Jedná se vlastně o speciální využití asymetrických šifer, kdy jako tajná informace slouží privátní (soukromý) klíč (*private key*), zatímco jméno entity, veřejné informace o ní a veřejný klíč (*public key*) jsou k dispozici veřejně ve formě tzv. **certifikátu** (*certificate*). Tento certifikát je podepsán certifikační autoritou. Lze tak ověřit, zda je certifikát skutečně pravý.

    Samotné prokazování identity potom probíhá tím způsobem, že se přesně definovaným algoritmem vytvoří tzv. **podpis** (**signatura**, *signature*). Podpis lze ověřit pomocí certifikátu a bez znalosti privátního klíče ho nelze falšovat.

    Tato metoda je velice rozšířena a standardizována, takže její použití není komplikované a je asi nejvýhodnější. Bohužel, nelze ji použít jako primární (uživatel si nemůže pamatovat svůj privátní klíč).

[4] Může se jednat i o metainformaci (například algoritmus), ale takové systémy se prakticky příliš nepoužívají.

Samotné ověřování se pak může provádět:

- **centralizovaně** – ověření provede centrální autentifikační server. Ten, kdo ověření požadoval, tak nemusí mít žádné speciální informace kromě způsobu, jak komunikovat s tímto serverem.

  Mezi nevýhody patří především implementační složitost a citlivost na napadení nebo vyřazení serveru. Výhodou je snadná správa. Tento způsob se používá tam, kde již nějaký ověřovací systém (např. Kerberos) existuje.

- **decentralizovaně** – ověřování provádí přímo ten, kdo ho vyžaduje. Téměř výhradně se používá autentifikace pomocí šifer, většinou pak digitálním podpisem.

Ze všeho zde uvedeného plyne, že jednoznačně nejvýhodnější metodou ověřování je použití asymetrických šifer. Tato metoda je obtížně napadnutelná, distribuovaná, standardizovaná a velmi rozšířená.

2.6.5 Ověřování identity součástí systému

Pro každou součást systému jsou kladeny jiné nároky na autentifikaci.

Uzel: Ověření pravosti jména nebo adresy uzlu (počítače) sice je možné, ale ve většině případů se neprovádí. Zbytek systému používá takové bezpečnostní mechanismy, které na pravosti jména uzlu nezávisí. Případný útočník nic nezíská vydáváním se za jiný uzel, maximálně se může tímto způsobem maskovat.

Vnitřní součásti hostitele, služby: Hostitel má plnou kontrolu nad tím, co se v něm děje. Pravost identity jeho vnitřních součástí je tak zaručena na úrovni algoritmů a jakékoliv narušení je možné jen důsledkem programové chyby.

Klient: Ověřování identity klienta není vždy nutné, ve většině případů klient svou identitu prokazovat nemusí (prokazuje se identitou svého uživatele, který má roli vlastníka agenta). Případný ověřený klient pak může být brán jako důvěryhodnější.

V některých systémech si agent uchovává jméno (identitu) klienta společně se jménem vlastníka a prokazuje se obojím. Klient tak vlastně reprezentuje aplikaci, zatímco vlastník reprezentuje uživatele této aplikace. Hostitel tak má jemnější možnost rozhodovat, protože například i důvěryhodný uživatel může používat neznámou aplikaci (a naopak, některé důvěryhodné a zabezpečené aplikace může používat libovolný uživatel).

V takovém případě platí prakticky totéž co pro identitu vlastníka.

Agent: Protože agent může existovat jen uvnitř hostitele, je zcela na hostiteli jak zajistí, aby jeho jméno nemohlo být změněno (jak vnějším zásahem, tak z vlastní vůle). Řešení je velice jednoduché: jméno agenta (a ostatní identifikační údaje, jako je jméno vlastníka) je uchováváno odděleně od agenta. Při migraci pak musí důvěřovat hostiteli, od kterého agenta přijal (to však musí v každém případě), viz dále.

Protože agent je entita dynamická, většina bezpečnostních rozhodnutí je prováděna nikoliv na základě identity agenta samotného, ale na základě informací o jeho vlastníkovi a klientovi, kde byl vytvořen. V úvahu je také nutné brát informaci, z jakého hostitele agent přicestoval.

Hostitel: Hostitel svoji totožnost prokazuje především ve dvou případech:

- **vyslání agenta** – klient může (ovšem nemusí) požadovat, aby hostitel prokázal svoji totožnost. Hostitel napadený (nebo přímo vytvořený) útočníkem může z agenta získat jakoukoliv informaci v něm obsaženou. Tyto informace (především identitu vlastníka) lze různě zneužít.
- **migrace** – při předávání migrujícího agenta si musí hostitelé navzájem důvěřovat.
  - Hostitel, který agenta přijímá, musí důvěřovat odesílajícímu hostiteli, že mu předává všechna data ve správné formě a že s nimi nijak nemanipuloval. Pokud zná útočník vnitřní implementaci hostitele, může spolu s vhodnou formou agenta provést zdařilý útok a získat neoprávněný přístup k jiným agentům nebo k interním částem hostitele.
  - Odesílající hostitel musí důvěřovat příjemci, že nebude takovou manipulaci provádět. Navíc předáním agenta nespolehlivému hostiteli může dojít nejenom k úniku dat, ale i ke ztracení agenta.

Vlastník: Identita vlastníka je velice významná informace ze dvou důvodů:

- Na jejím základě jsou agentovi poskytována práva a služby. Zfalšováním identity vlastníka by tak agent mohl získat práva, která mu nepřísluší.
- Vlastník má možnost manipulovat s agentem za jeho života, ukončovat ho a vyžádat si jeho návrat. Pokud by se podařilo útočníkovi získat identitu vlastníka, mohl by agenta ukončit nebo od něj získat diskrétní informace.

Primární ověřování většinou probíhá pomocí hesla na úrovní operačního systému nebo v samotném programu klienta (za proces ověřování se dá považovat import privátního a veřejného klíče).

Klient pak do nově vytvářeného agenta vloží informaci o vlastníkovi. Agent si ji pak během svého života nese s sebou, přičemž jednotliví hostitelé si ji ověřují. Nemusí si ji ověřit všichni – nutné je to jen u prvního (který agenta přijímá od klienta). Pokud hostitel přijme migrujícího agenta od jiného hostitele, kterému zcela důvěřuje, může ověření identity vlastníka agenta vynechat.

Protože všechny informace obsažené v agentovi jsou zcela viditelné pro každého hostitele a dají se odposlechnout při nezabezpečeném transportu, je nevhodné, aby s sebou nosil nějakou tajnou informaci použitelnou k ověření (např. heslo). A už vůbec není přípustné, aby měl agent k dispozici tajný klíč (jiný než vlastní). Proto se k zajištění ověřitelnosti používá digitální podpis: Agent (resp. jeho část obsahující důležité údaje) je vlastníkem podepsán a obsahuje jeho certifikát (veřejný klíč). Hostitel (nebo kdokoliv jiný, komu jsou tato data poskytnuta) pak ověřením certifikátu (ten je podepsán certifikační autoritou) zkontroluje, zda je identita správná (platná) a kontrolou podepsaných dat pak ověří, že s agentem nebylo manipulováno.

2.7 Komunikace

Důležitou vlastností agenta je komunikativnost. Významnou roli hraje nejen u agentů mobilních, ale i u stacionárních. Komunikace mezi softwarovými agenty obecně je oblast intenzivního výzkumu v oblasti umělé inteligence.

V případě mobilních agentů se výrazně komplikuje realizace. Běžné metody síťové komunikace totiž nepočítají s tím, že jedna strana (či dokonce obě) mění svoje umístění.

Ostatní části systému spolu samozřejmě také komunikují, ale protože zůstávají stacionární, jedná se o zcela běžnou komunikaci v distribuovaném systému.

2.7.1 Formy komunikace

Komunikace může probíhat mnoha formami. Ty se mohou rozlišovat podle několika charakteristik; nejdůležitější jsou:

Synchronizace: *Synchronní* komunikace probíhá tak, že vysílající strana pošle zprávu a čeká, až ji přijímající strana zpracuje. Ta většinou vrátí nějakou odpověď. Vysílající strana čeká na dokončení i v případě, kdy žádnou odpověď neočekává.

Při *asynchronní* komunikaci odesílající strana prostě pošle zprávu a pokračuje v běhu. Není tak zdržována druhou stranou a zpožděním při komunikaci. Odpověď musí být poslána stejným mechanismem. Nevýhodou je poměrně komplikovaný mechanismus zjišťování, zda je zpráva odpověď a pokud ano, na *co* je to odpověď.

Umístění: Komunikace může probíhat **lokálně** nebo **vzdáleně** (*remote*). Lokální komunikace je rychlejší a jednodušší, ale vyžaduje, aby obě strany byly na stejném uzlu.

Adresace příjemce: Podle toho, komu je zpráva určena, se rozlišuje několik variant.

- Příjemce je pevně a jednoznačně určen. Tento způsob je nejčastější a nazývá se *unicast*.
- Zpráva je určena celé skupině příjemců. Jedná se tedy o **skupinovou adresaci** (*multicast*, *broadcast*).
- Příjemce nemusí být vůbec specifikován. Zpráva se jen uloží na pevně určené místo (schránka) a označí se její obsah. Příjemce zjistí obsah schránky a pokud mu obsah zprávy vyhovuje, vybere si ji.

Identifikace zdroje: Zaslaná zpráva může být anonymní (nelze určit odesílatele) nebo jednoznačně identifikovatelná (odesílatel je uveden).

2.7.2 Prakticky používané formy komunikace

Prakticky se používá jen několik forem komunikace, z nichž každá má svoje výhody a nevýhody.

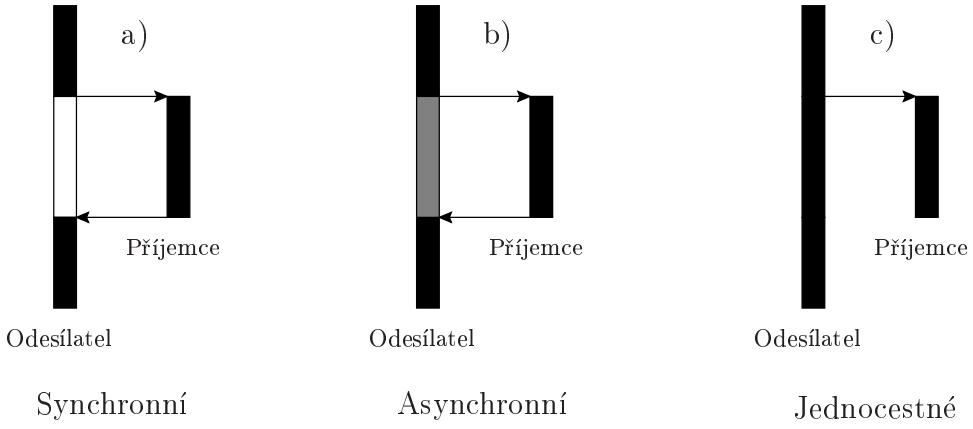
Zasílání zpráv (*Message Passing*): Zprávou je v tomto případě speciální blok dat specifikovaný uživatelem. Jeho vnitřní struktura nebývá určena a je zcela na obou komunikujících stranách, jak jsou data zakódována. Zpráva však mívá hlavičku, která obsahuje: identifikace odesílatele, adresa příjemce a typ zprávy.

Komunikace mezi agenty může probíhat podle následujících čtyř schémat:

Synchronní zprávy: Toto je nejčastější model posílání zpráv. Agent odešle zprávu jinému agentovi. Odesílatel je blokován, dokud neobdrží odpověď nebo potvrzení od příjemce (viz obrázek 2.9a).

Asynchronní zprávy: Zpráva je odeslána jedním agentem ostatním. Odesílatel pokračuje ve svém běhu, tj. není blokován komunikací. Místo toho vyhradí místo pro odpověď, které pak může testovat na její přítomnost. Toho lze s výhodou využít, pokud komunikuje více agentů s jedním uzlem (viz obrázek 2.9b).

Jednocestné zprávy: Tento typ zpráv je také asynchronní (viz obrázek 2.9c). Agent odešle zprávu jinému agentovi. Odesílatel ale není blokován dokud neobdrží odpověď, dokonce ji ani neočekává. Toto komunikační schéma se použije v systémech spojených nespolehlivou počítačovou sítí.



Obrázek 2.9: Komunikační schémata

Skupinové zprávy: Agent najednou odešle zprávy více než jednomu příjemci. Lze je tedy s výhodou použít v systémech mobilních agentů.

Obecně neexistuje žádný standard pro komunikaci posíláním zpráv, ale pro některé případy lze využít standardních protokolů. Velice často se například u dlouho pracujících agentů používá elektronická pošta pro sdělení důležitých událostí a výsledků.

Vzdálené volání podprogramů (*Remote Procedure Call*): Vzdálené volání podprogramu (nebo metody) umožňuje zcela odstínit komunikační záležitosti, protože není nutné provádět speciální akce. Pouze se zavolá procedura (metoda). Ta pak volání přesměruje na jiný uzel (v případě mobilních agentů na jiného hostitele), kde je vyvolána odpovídající metoda se stejnými argumenty. Po dokončení se návratová hodnota vrátí zpět a volání se ukončí. Iniciátor komunikace tak vůbec nemusí poznat, že komunikuje vzdáleně.

RPC probíhá vždy synchronně. Před použitím musí být navázáno spojení a toto spojení musí být také udrženo po celou dobu, kdy se komunikace používá.

Kvůli jednoduchosti svého používání se princip *RPC* používá velmi často. Zvláště to platí v případě objektového prostředí, kde se mechanismem *ORB* (*Object Request Broking*) reprezentují celé objekty umístěné na vzdáleném počítači. V takovém případě se samozřejmě nejedná o *RPC*, ale o *RMI* (*Remote Method Invocation*).

Tabule (*Blackboard*): Tato forma komunikace byla navržena speciálně pro softwarové agenty. Probíhá lokálně, asynchronně, anonymně, skupinově a je omezena pouze na komunikaci mezi agenty.

Základem je schránka na zprávy, tzv. **tabule** (*blackboard*). Kterýkoliv agent na ní může zapsat zprávu, aniž by znal příjemce. Jiný agent si tuto zprávu přečte a pokud dojde k závěru že on je tím správným adresátem, zpracuje ji. Agenti tak nevědí, s kým komunikují, ale tabule jim umožňuje sdílet všechny potřebné informace. Hlavní význam má tabule v případě spolupracujících agentů, kteří se s její pomocí domlouvají ve skupině.

U mobilních agentů se použití tabule rozšiřuje tím, že v ní může agent nechat zprávu pro jiného agenta, který je momentálně na jiném uzlu. Až tento přijímající agent přicestuje na tohoto hostitele, z tabule si přečte tuto zprávu.

Služby: Speciální způsob komunikace je možný v systémech, kde hostitel nabízí agentům různé **služby** (viz kapitola 2.5). Může totiž umožnit, aby agent sám nějaké služby nabízel ostatním agentům. Ti pak k těmto službám přistupují standardním způsobem a mnohdy ani nemusí vědět, že ve skutečnosti komunikují s jiným agentem.

Tento způsob je samozřejmě možný pouze lokálně. Pokud agent odmigruje na jiný uzel, komunikace nemůže pokračovat.

2.7.3 Navázání komunikace s agentem

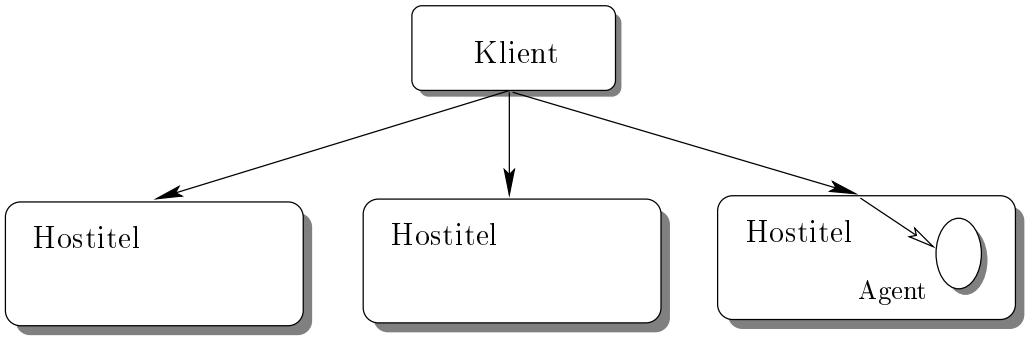
Nejjednodušší situace nastane v případě, kdy spolu dva agenti mohou komunikovat jen na stejném hostiteli. Celou komunikaci tak zajistí hostitel. Lokální komunikace by měla být upřednostňována (je jednou z významných výhod použití mobilních agentů), ale vzdálená komunikace s agentem je také velice důležitá. Klient by měl mít možnost komunikovat s klientem (zjistit současný stav apod.) a agenti by měli mít možnost spolupracovat i vzdáleně (minimálně by měli být schopni „domluvit si schůzku“ na hostiteli, kde si vymění informace).

Pokud klient nebo agent[5] chce komunikovat s jiným agentem, situace je poměrně komplikovaná. Systém musí zajistit, že toto spojení bude navázáno, ať se cílový agent nachází kdekoliv. V dalším textu bude popsáno spojení klient-agent, pro komunikaci agent-agent je vše zcela shodné.

Prohledání všech hostitelů (*Search*): Klient musí znát všechny hostitele, na kterých se může agent nacházet. Každého z nich se dotáže, zda je na něm agent momentálně přítomen (obrázek 2.10). Jeden z těchto hostitelů odpoví kladně a zajistí navázání spojení.

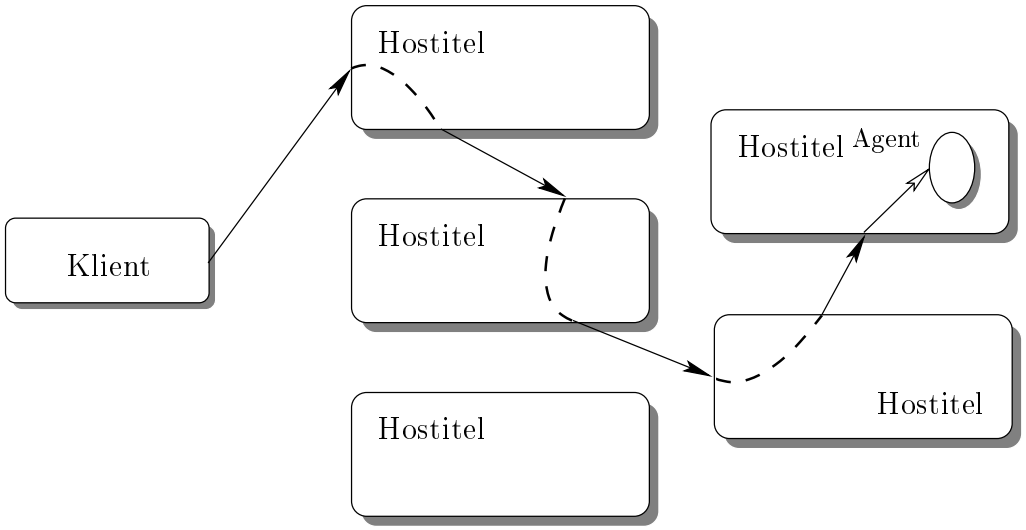
Předávání spojení (*Forwarding*): Pokaždé, když agent odmigruje z hostitele pryč, hostitel si zaznamená jeho nové umístění . Pokud přijde požadavek na spojení s tímto agentem, naváže spojení s tímto hostitelem. Vzniká tak celý

[5] Agentovi komunikaci zajišťuje většinou hostitel, protože je zcela zbytečné aby ji každý agent implementoval sám.



Obrázek 2.10: Nalezení agenta prohledáváním

řetěz přes všechny hostitele, kudy agent prošel (viz obrázek 2.11). Proces předávání komunikace na dalšího hostitele se nazývá *forwarding*.



Obrázek 2.11: Navázání spojení předáváním komunikace

Oproti předchozímu způsobu, je zde výrazně jednodušší implementace klienta. Nepotřebuje znát všechny hostitele a prohledávat je, stačí mu znát jednoho jediného, na kterém se agent zaručeně nacházel (např. hostitel, na který byl agent vyslán). Ten pak výše uvedeným mechanismem zajistí, že se naváže spojení s agentem na jeho aktuální pozici.

Nad těmito výhodami však výrazně převažují nevýhody:

- Každý hostitel si musí zapamatovat každého agenta, který přes něj kdy prošel.
- Komunikace zbytečně zatěžuje hostitele i síť.
- Výpadek jednoho hostitele znamená, že spojení s agentem již nelze navázat.
- Pokud je vytvořený řetěz příliš dlouhý, zpoždění přenosu (součet všech zpoždění mezi hostiteli a uvnitř nich) může být natolik dlouhé, že ho klient může považovat za selhání sítě.

V praxi se tato metoda dá použít jen pro jednoduché systémy o malém počtu hostitelů na lokální síti. Ani tam však není příliš vhodná.

Sledování stop (*Tracking*): Většina nevýhod obou předchozích způsobů se odstraní jejich kombinací. Hostitel si opět uchovává informaci o tom, kam agent odmigroval. Na dotaz pak klientovi tuto adresu dalšího hostitele sdělí. Klient tak postupně sleduje, kudy agent procházel, až najde jeho aktuálního hostitele.

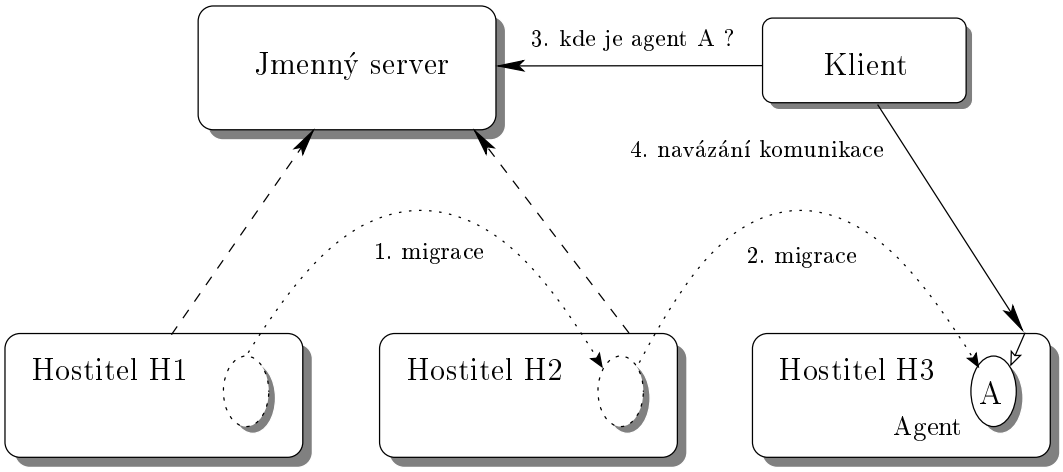
Agent nemusí prohledávat všechny hostitele a ani je nemusí vůbec znát. Na druhou stranu, hostitel si stále musí uchovávat informace o všech agentech a pokud je ztratí, mohou být tito agenti nedostupní.

Využití jmenných služeb: Součástí systému je jmenný server (může jich samozřejmě být i více). Ten uchovává informace o aktuální poloze agenta. Klient si pak dotazem na jmenný server tuto polohu zjistí a naváže spojení přímo s agentem (viz obrázek 2.12). Samotná komunikace je tak velice jednoduchá.

Jmenný server však musí mít neustále aktuální informace o tom, kde se agent právě nachází. Při každé migraci se na něm musí záznam o patřičném agentovi změnit tak, aby obsahoval novou pozici. Systém proto musí zajistit, že při každé migraci bude jmennému serveru zaslána zpráva o tom, který agent kam migruje. Zprávu může posílat kterýkoli ze zúčastněných hostitelů nebo agent sám. Důležité je, aby migrace i změna údajů na tomto serveru proběhly jako jedna jediná transakce. Musí totiž být zaručena atomicita, konzistence a trvalost těchto změn (izolace není nutná, protože změna se týká jen jednoho agenta a nemůže jich probíhat více najednou).

Nevýhodou jmenného serveru je, že celý systém je závislý na jeho funkci. Pokud dojde k výpadku (nebo napadení), nebude komunikace s agenty možná. V praxi se to řeší tím, že jmenných serverů je více.

Důležitým aspektem je také ochrana jmenného serveru před falešnými informacemi. Útočník totiž může jmennému serveru poslat falešnou zprávu o migraci a agent se tak stane nedostupným. Řešením je, že každá taková



Obrázek 2.12: Navázání spojení pomocí jmenných služeb

zpráva bude vyžadovat autentifikaci jejího odesílatele. V úvahu pak budou brány jen zprávy od důvěryhodných hostitelů nebo agentů.

Příklad použití je znázorněn na obrázku 2.12, kde události probíhají takto:

1. Agent `A` migruje z hostitele `H1` na hostitele `H2` (tečkovaná čára). Jmenný server obdrží od `H1` o migraci oznámení (čárkovaná čára) a zaznamená si, že agent `A` se nachází na hostiteli `H2`.
2. Agent `A` migruje z hostitele `H2` na hostitele `H3`. Situace je stejná jako v předchozím případě. Po provedení se agent nachází na hostiteli `H3` a jmenný server má odpovídající záznam.
3. Klient chce navázat spojení s agentem `A` a zeptá se tedy jmenného serveru na momentální pozici. Ten odpoví, že agent `A` se právě nachází na hostiteli `H3`.
4. Klient kontaktuje hostitele `H3` a ten mu zprostředkuje komunikaci s agentem.

Ve většině systémů se používají jmenné servery. V jednodušších systémech není problém navázání komunikace explicitně řešen a musí se použít prohledávání všech hostitelů.

2.7.4 Udržení komunikace

Poté, co bylo spojení navázáno, klient (agent) může s agentem libovolně komunikovat. To však platí jen do doby, než agent opět odmigruje někam jinam a spojení se přeruší. Pokud má komunikace pokračovat, spojení je nutné udržet nebo alespoň obnovit.

Automaticky je problém řešen, pokud je komunikace předávána přes hostitele (obrázek 2.11). Při migraci jednoduše odesílající hostitel naváže spojení s přijímajícím a prodlouží tak celý řetěz.

Při použití jiných metod se spojení rozpojí a je nutné jej znovu obnovit. K tomu je však nutné znát novou polohu agenta. Jednoduché je to v případě, když agent (nebo některý hostitel) pošle klientovi zprávu o migraci obsahující jméno a adresu cílového hostitele. S touto informací je pak navázání spojení jednoduchou záležitostí.

V nejhorším případě klient nedostane žádnou zprávu, pouze najednou zjistí, že v komunikaci došlo k chybě (druhá strana již neexistuje). V takovém případě musí znovu provést celý proces navázání komunikace (pokud se nezdaří, nebyl výpadek spojení způsoben migrací, ale selháním agenta nebo hostitele). Tento postup je sice komplikovaný, ale má také tu výhodu, že se spojení obnoví i po dočasném výpadku sítě.

Protože mechanismus komunikace, navázání a obnovení spojení je stejný pro různé části systému, je výhodné ho realizovat jako knihovnu. Na této úrovni lze dokonce zajistit, že spojení bude obnoveno automaticky, aniž by o tom využívající program (klient, agent, hostitel) věděl.

V objektovém prostředí je možné toto zajistit pomocí **zástupce** (*proxy*). **Zástupce** je objekt, který imituje chování jiné třídy a zastupuje tak jiný objekt, v tomto případě vzdáleného agenta. Komunikace se zástupcem se automaticky převádí na komunikaci s agentem a mechanismus je zcela transparentní.

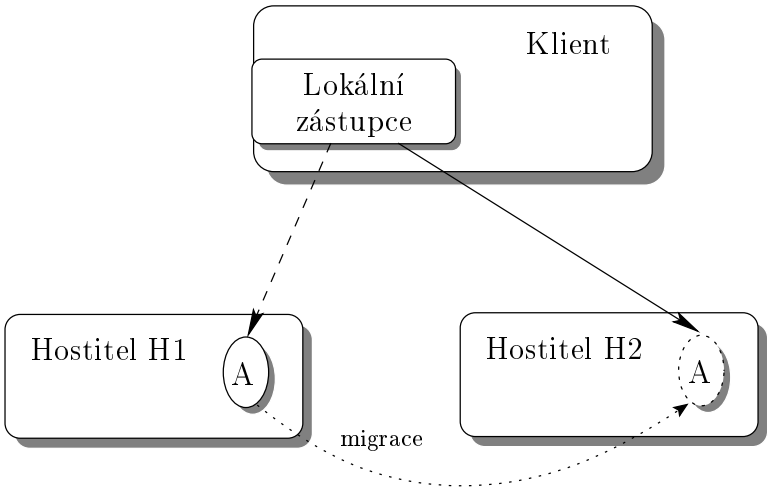
Podle umístění zástupce a způsobu ošetření migrace lze rozlišit dva druhy (jejich analýzu a zhodnocení je možné nalézt v [Deu1999]):

Lokální zástupce (*Local Proxy*): Zástupce je umístěn na klientovi. V případě migrace dojde k výpadku, který zástupce detekuje a znova naváže spojení. Spojení se také naváže po momentálním výpadku sítě. Zástupce tak zajišťuje nejen ošetření migrace, ale i zvětší spolehlivost a zcela oddělí mechanismus komunikace od klienta.

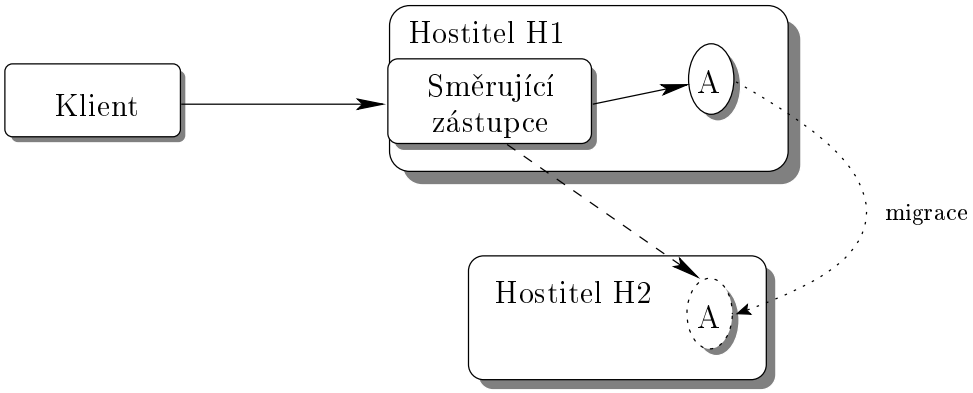
Směrující zástupce (*Forwarding Proxy*): Zástupce je umístěn na hostiteli a klient místo s agentem komunikuje s ním. V případě migrace agenta zástupce zajistí, že veškerá komunikace bude předávána na nového hostitele. Výsledek je obdobný jako při navazování spojení předáváním, ale řetězec hostitelů se vytváří jen přes hostitele, přes které agent prošel během komunikace.

Takto umístěný zástupce zcela zabrání výpadku komunikace. Migrace je tak zcela transparentní, ale klient si musí sám zajišťovat síťovou komunikaci. Výhodné je tedy použít oba dva druhy zástupců najednou, čímž se spojí všechny výhody.

Princip funkce lokálního i směrujícího zástupce je znázorněn na obrázcích 2.13 a 2.14. Tečkovanou čarou je znázorněna migrace, čárkovaně pak stav po migraci.



Obrázek 2.13: Použití lokálního zástupce pro udržení komunikace



Obrázek 2.14: Použití směrujícího zástupce pro udržení komunikace

2.7.5 Syntaxe přenášených zpráv

V předchozím textu bylo uvedeno, jakým způsobem lze s agentem vyměňovat zprávy a obecně data. Tyto mechanismy jsou ale jen *prostředkem*, kterým je komunikace uskutečňována. Neméně důležité je také to, jakým způsobem jsou zakódovány informace a jak se s těmito informacemi pracuje.

Na tuto problematiku existují dva pohledy. Liší se komplexností i nabízeným řešením.

Standardizace datového formátu

Z pohledu distribuovaných systémů a softwarového inženýrství je nutné řešit především tyto problémy:

- Data musí být jednoduše přenositelná po síti i v heterogenním prostředí.
- Datový formát by měl být přesně definován a standardizován. Výhodou je, pokud je definice umístěna mimo vlastní programový kód aplikace. Mohou jej tak sdílet různé aplikace.

Výsledkem těchto požadavků je **metaformát**, tedy způsob jak popsat formát a pak jej realizovat. Prakticky to znamená, že je definován jazyk, popisující strukturu a reprezentaci dat. Podle tohoto jazyka lze kdykoliv a kdekoliv data zakódovat a pak zase dekódovat, přičemž vstupní ani výstupní forma nemusí být shodná (obsah samozřejmě musí zůstat nezměněn).

Příklady takových jazyků jsou *ASN.1* (*Abstract Syntax Notation*), *XDR* (používán v *SUN RPC*) nebo *IDL* (*Interface Definition Language*). Tyto jazyky jsou často využívány standardními prostředky pro komunikaci (*SNMP*, *SUN RPC* a *CORBA*) a používají se automaticky.

V současné době se velmi používá formát *XML* [Bra98] a do budoucna bude pravděpodobně ještě rozšířenější. Je navržen jako jednoduchý, člověkem čitelný (textový) formát s pevně definovanou strukturou, určený právě ke standardizované výměně informací a odstranění velkého množství nekompatibilních formátů. *XML* dokument může obsahovat libovolná data a jeho struktura je popsána v tzv. *DTD* (*Document Type Definition*).

Jazyky pro komunikaci mezi agenty

Z hlediska umělé inteligence (do níž softwaroví agenti patří) je důležité definovat, jak si agenti předávají informace, dotazy a odpovědi. Je nutné předpokládat, že data nebudou mít vždy stejný formát, protože se mohou předávat různé informace. Agenti musí za své činnosti rozhodnou, jak informaci zakódovat a jak ji rozkódovat (toto rozhodnutí normálně činí programátor). Proto byly vyvinuty jazyky pro komunikaci mezi agenty, které definují nejenom formát zpráv, ale také jak se s těmito zprávami pracuje.

Tyto jazyky vycházejí z principu používaných v umělé inteligenci, především pak z predikátové logiky a deklarativních jazyků. Agenti si mezi sebou sdělují fakta ve formě predikátů, vyjadřujících vlastnosti jich samých nebo nějakých entit. Při tom je zřejmě nutné zajistit, aby všechny pojmy (entity, vlastnosti, výroky) měly ve všech aplikacích stejný význam. Tyto významy jsou shrnuty v tzv. **ontologiích** (*ontology*). Komunikující agenti tak musí ontologie sdílet, jinak není možné, aby se domluvili.

Dominantní postavení mezi samotnými komunikačními jazyky má *KQML* (*Knowledge Query and Manipulation Language*). Používá syntaxi podobnou LISPu: vše je uzavřeno v závorkách, první element za závorkou určuje význam zprávy (tzv. *performative*), například dotaz, odpověď, doporučení, omluvu a podobné (několik jich je standardizováno, nicméně lze definovat libovolné vlastní). Příklad jednoduché komunikace popsané v jazyce *KQML* je na výpisu 2.4. Agent

se jménem `joe` se ptá (*performative* `ask-one`) agenta jménem `book-market` na cenu knihy. Samotný dotaz je specifikován pomocí standardu *KIF* jako predikát, určující cenu knihy `12-3456-78` jako proměnnou (označeno otazníkem). Agent `book-market` tento dotaz zpracuje, doplní predikát a vrátí ho ve druhé zprávě (*performative* `tell`). Bližší informace o jazycích pro komunikaci mezi agenty lze nalézt například v [Lab1998] nebo v [Lab1999].

```
(ask-one
  :language KIF
  :content (Price 12-3456-78 ?x)
  :sender joe
  :receiver book-market
  :reply-with joe-1234)

(tell :sender book-market
  :language KIF
  :content (Price 12-3456-78 19.95)
  :receiver joe
  :in-reply-to joe-1234)
```

Výpis 2.4: Ukázka komunikace v KQML

2.8 Bezpečnost

Princip mobilního kódu a speciálně mobilních agentů vytváří mnoho bezpečnostních problémů. Ty je nutné řešit, zvláště pak v otevřeném a nezabezpečeném prostředí.

Oproti klasickým distribuovaným systémům je nutné řešit výskyt **zákeřných** (*malicious*) agentů, tedy agentů, kteří se snaží úmyslně provést některý z bezpečnostních útoků (neoprávněný přístup k informacím).

Nejdůležitější bezpečnostní problémy jsou následující:

- Hostitel je vystaven nebezpečí útoku od zákeřných agentů. Ty se mohou pokusit ho napadnout a získat tak přístup k citlivým informacím v něm obsaženým.
- Zákeřný agent se může pokusit získat citlivé informace z jiného agenta.
- Agent může zneužíváním jemu přidělených prostředků provést tzv. *denial of service* útok. Cílem je dosáhnout stavu, kdy hostitel není schopen zajistit provoz ostatních agentů a v nejhorším případě dokonce ani svůj vlastní.
- V síti se může vyskytnout i zákeřný hostitel (vytvořený útočníkem). Takový hostitel pak může nejenom získávat informace z agentů, ale může dokonce i s agentem manipulovat. Zvláště nebezpečné je, když z normálního důvěryhodného agenta vytvoří agenta zákeřného.

- Přenos agenta po síti by měl být zabezpečen. Při napadení přenosu může dojít nejen k úniku informací, ale při podvržení dat může dojít také k modifikaci agenta (stejně jako v předchozím bodě).
- Komunikace by měla být zabezpečena. Tento aspekt je zcela shodný jako v klasických distribuovaných systémech.

Systém mobilních agentů by měl všechny tyto případy nějakým způsobem ošetřit.

2.8.1 Bezpečný přenos a komunikace

Přenos agenta a komunikace jsou implementovány stejným způsobem, tj. jako přenos dat po síti. Ten je však velice snadno napadnutelný.

Obrana proti těmto útokům je stejná jako u normálních distribuovaných systémů, ale ne všechny metody jsou vždy přímo použitelné. Rozdíl spočívá v tom, že agent s sebou nemůže nosit žádné tajemství jako jsou šifrovací (privátní) klíče. Navíc většina provozu probíhá bez účasti uživatele, takže nejsou použitelné ani metody vyžadující jeho účast.

Pasivní útok

Útočník se do komunikace nijak nevměšuje, jen ji sleduje. Nejjednodušší a nejčastější (a také nejnebezpečnější) pasivní útok je **odposlouchávání** (*eavesdropping*), kdy útočník získá přímo přenášená data. Složitější metody zahrnují **analýzu provozu** (*traffic analysis*).

Pasivní útoky se dají velmi obtížně detekovat, ale lze jim snadno zabránit použitím kryptografických metod (šifrováním komunikace).

Pro šifrování komunikace lze využít jak symetrické tak asymetrické šifry. Kvůli své výpočetní náročnosti se však asymetrické šifry příliš nepoužívají a jejich použití je vyhrazeno spíše na autentifikaci. V této oblasti navíc existuje několik standardů.

Jak již bylo řečeno, agent nemůže mít žádné tajné klíče (aniž by byla zpochybněna jejich spolehlivost). Proto mu zabezpečení komunikace musí zajistit hostitel vytvořením bezpečného kanálu.

Standard *SSL* (*Secure Socket Layer*) umožňuje vytvořit bezpečný kryptovaný kanál, se kterým pak lze pracovat stejným způsobem jako s normálním TCP spojením. Prakticky každý protokol využívající TCP tak lze provozovat i pomocí SSL včetně *RMI* nebo *CORBA*. SSL navíc umožňuje jednostrannou nebo oboustrannou autentifikaci s použitím *X.509* certifikátů.

CORBA obsahuje standardní způsob řešení bezpečné komunikace: *CORBA Security Service*. Umožňuje nejen šifrovanou komunikaci a autentifikaci pro přenos, ale i předávání autentifikace při vnořeném volání (volaná metoda vyvolá metodu z jiného vzdáleného objektu). Identitu volajícího lze používat i na aplikační úrovni.

Aktivní útok

Aktivní útok zahrnuje jakékoliv zasahování do komunikace ze strany útočníka. Nejjednodušší je prostá změna přenášených dat, kdy se tato data poruší a přijatá data jsou tak neplatná nebo obsahují zavádějící informace.

Častější, komplikovanější a mnohem nebezpečnější je útok, kdy se útočník vydává za někoho jiného. Pokud se mu to podaří, získá přímý přístup k informacím a může sám poskytovat pečlivě vybraná data (např. zákeřného agenta).

Na rozdíl od pasivních, aktivní útoky lze poměrně snadno odhalit (použitím šifrování, zabezpečení dat a autentifikace). Zabránit jim je však velice obtížné.

Základem obrany proti aktivním útoků je **autentifikace**, tedy ověření identity druhé strany komunikace. Pokud selže, pravděpodobně se jedná o útočníka a komunikaci je nutné okamžitě přerušit.

O autentifikaci je podrobněji psáno v kapitole 2.6.4.

2.8.2 Ochrana hostitele a zdrojů

Agent je prováděn na hostiteli a ten mu poskytuje přístup ke zdrojům. Zákeřný agent toho však může zneužít a pokusit se tyto zdroje napadnout. Agent se tak vlastně chová jako virus.

Obrana hostitele spočívá v omezení přístupu agenta ke zdrojům. Na úrovni provádění programu je nutné zamezit přímému přístupu k hardware (to zajišťuje operační systém) a k operačnímu systému (to již musí zajistit hostitel sám). Některé jazyky, jako například Java, mají již implementovány patřičné prostředky.

Agent je tedy zcela odkázán na hostitele, k jakým zdrojům mu zajistí přístup. To je však nedostačující, protože není žádoucí, aby měl agent *plný* přístup ke zdrojům. Je nutné jej nějakým způsobem omezit.

Pro tyto účely se používá koncepce služeb, jak byla popsána v kapitole 2.5. Když agent požádá o službu, hostitel rozhodne zda ji přidělí nebo ne. Pokud by její přidělení bylo příliš nebezpečné, požadavek odmítne. Na této úrovni je tedy možné znemožnit přístup k celé službě. Jemnější kontrolu přístupu pak provádí samotná služba.

2.8.3 Ochrana agenta

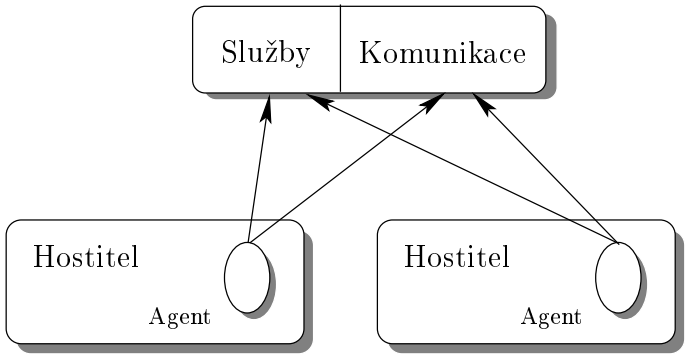
Během agentova života je nutné zajistit, aby nedošlo k úniku informací v něm obsažených ani k jejich změně. Ochranu si však agent nemůže zajistit sám, je zcela odkázán na hostitele.

Ochrana před jinými agenty

Není možné předpokládat, že by agent byl na hostiteli sám. Pravděpodobně jich na hostiteli bude hned několik a zákeřný agent se může pokusit napadnout ostatní

a získat tak data v nich obsažená. Hostitel samotný napaden nebude a ani nemusí útok zaregistrovat. Detekce takového napadení je obecně velmi obtížná a nevyplatí se, protože prevence bývá daleko účinnější.

Jak již bylo uvedeno v předchozí podkapitole, agent má velice omezené své možnosti kvůli zamezení napadení hostitele. Toto omezení agenta je nutné ještě rozšířit o oddělení jednotlivých agentů od sebe, jak je znázorněno na obrázku 2.15.



Obrázek 2.15: Oddělení agentů

Každý agent je obklopen *bariérou*, která jej odděluje jak od hostitele tak od ostatních agentů. Má pouze povolen přístup k nabízeným službám (a prostřednictvím nich ke zdrojům) a má zajištěnu komunikaci[6] (s ostatními agenty, ale nejen s nimi). V každém případě agenti k sobě nemohou přistupovat přímo.

Napadení jiného agenta je možné provést několika způsoby:

Získání soukromých dat: V objektových jazycích lze tomuto útoku velice efektivně zabránit pomocí **zapouzdření** (jedné ze základních vlastností OOP). Nicméně i v takových jazycích často existuje možnost, jak zapouzdření obejít. Tyto metody sice vyžadují znalosti vnitřní reprezentace dat, nicméně ta bývá standardizována. Data tak lze získat například analýzou reprezentace C++ objektu v paměti nebo serializovaného Java objektu.

Přepsání dat: Situace je velice obdobná předchozímu případu, ale provedení je mnohem obtížnější a vyžaduje přístup k „neobjektové“ nižší úrovni, jako je tomu například v C++. Tomuto útoku tak lze většinou zcela zabránit (např. v Javě není možný vůbec).

Předstírání hostitele: Pokud by se nějakému zákeřnému agentovi podařilo vydávat se za hostitele, mohl by tím nejenom z agenta získat mnoho informací, ale mohl by mu také nějaké falešné informace podstrčit. V extrémním případě by pak mohl předstírat komunikaci s vlastníkem.

[6]Schéma se ještě zjednoduší, pokud je přístup ke komunikaci zajištěn pomocí služeb.

Takovýto druh útoku není možný v případě procedurálních jazyků, kde je komunikace s hostitelem prováděna pomocí procedur a vazba je tam na úrovni programového kódu. Totéž platí, pokud je komunikace s hostitelem zajištěna přímo interpretrem.

Napadnutelný je tak pouze hostitel, který je implementován objektově a jeho proces probíhá ve stejném prostoru (stejném interpretru) jako v něm obsažení agenti. To je typický případ systému v jazyce Java. Agent při spuštění na hostiteli obdrží referenci na objekt (případně jich obdrží více), který pro něj reprezentuje hostitele. Voláním metod tohoto objektu pak komunikuje s hostitelem. Pokud by se jinému agentovi podařilo předat mu tímto způsobem referenci na jeho vlastní objekt (samozřejmě kompatibilní, tedy implementující stejný interface), napadený agent by si myslel, že komunikuje s hostitelem.

Zavádění nějakých metod autentifikace zde nemá velký význam, pouze by se tím kód výrazně zkomplikoval. Pro zabránění tomuto útoku se používají metody prevence, kde musí hostitel a chráněný agent spolupracovat. Princip spočívá v tom, že žádný jiný agent nesmí získat referenci na objekty, které by toto napadení umožnily (tj. objekt, kterému hostitel předává zmíněnou referenci na sebe). Hostitel ji nikdy neposkytne – není k tomu žádné opodstatnění. Jediným místem úniku tak zůstává přímá komunikace mezi agenty na úrovni volání metod. Hostitel takovou komunikaci nemusí umožnit a vše nechá probíhat přes RMI.

Jinou metodou je používání **zástupce**, který komunikaci zprostředkuje. Funkce je naprosto stejná jako v případě služeb (viz obrázek 2.7) a také účel je shodný. Agent pro své okolí zpřístupní pouze jeden určitý interface. Skutečný objekt, který tento interface implementuje, udržuje reference na jiné objekty agenta a jejich analýzou lze získat referenci na objekt umožňující napadení.

Proxy však tento postup znemožňuje. Lze vyvolávat její metody a vyvolávají se tak metody skutečného objektu, ale množina těchto metod je omezena na patřičný interface.

Změna programového kódu: Hostitel by měl zajistit, že programový kód nelze změnit za žádných okolností (často je toto omezení již provedeno na úrovni interepreteru nebo operačního systému). Agenti tím nebudou fakticky nijak omezeni, protože změna programového kódu je velmi špatná a nezdůvodnitelná programovací technika.

Existuje však daleko lepší způsob napadení, který je možný v případě že hostitel optimalizuje využití paměti a umožní více agentům sdílet stejný kód. Napadení je vedeno proti agentům se známým programovým kódem, ke kterému útočník vytvoří upravenou verzi. Tato upravená verze se chová

jako původní, ale přidává akce navíc jako například, že posílá citlivá data a podobně. Změna se může omezit jen na několik tříd.

Útočník vytvoří agenta A s tímto napadeným kódem a pošle ho hostiteli. Důležité je, aby na hostitele dorazil dříve, než jakýkoli agent se správným kódem (například ihned po startu hostitele). Tento agent normálně běží a předstírá užitečnou činnost. Později na hostitele dorazí agent B, který používá původní programový kód. Hostitel si však myslí, že tento kód již má k dispozici, neboť ho používá agent A. Spustí tedy B s tímto kódem aniž by tušil, že došlo k napadení. Také B vůbec nemůže zjistit, že byl napaden.

Hostitel si samozřejmě může ověřovat pravost kódu (digitální podpisy, například podepsaný *jar* soubor v Javě). Ověří si tak jen pravost celých balíků (knihoven), zatímco napadení může probíhat na úrovni tříd (balíky kódu se mohou lišit). Navíc problém se sdílením kódu může nastat i neúmyslně, kdy různí agenti používají různé verze knihoven. Dokonce může nastat případ, kdy více programátorů nezávisle na sobě vytvoří třídu se stejným jménem. Ze všech těchto důvodů vyplývá, že při sdílení programového kódu je nutné zachovat maximální opatrnost a sdílet jen tu část kódu, která je prokazatelně shodná.

Jakým způsobem je toto oddělení zajištěno závisí zcela na implementaci a programovacím jazyce. Na nejnižší úrovni může být využito oddělení výpočetních procesů, které zajišťuje operační systém. Na vyšší úrovni pak lze zajistit, aby jednotliví agenti neměli společný jmenný prostor a měli vždy svůj vlastní kód.

Tento problém (a ještě mnohé další) by řešil programovací jazyk umožňující skutečně účinně omezovat vyvolávání metod. Klasické objektové jazyky totiž řídí přístup jen podle toho, *co* je volání a nikoliv podle toho, *kdo* toto volání provádí.

Ochrana před hostitelem

Ochrana před hostitelem je velice obtížně realizovatelná, protože hostitel musí mít vždy plný přístup k agentovi. Jediný způsob, jak napadení zabránit je tedy vůbec nemigrovat na hostitele, kterým nelze důvěřovat. Zčásti to může zajistit i hostitel, když odmítne předat agenta hostiteli, kterého nezná nebo kterému se nedá důvěřovat. Rozhodnutí o důvěryhodnosti musí učinit správce systému.

I když manipulaci s daty v agentovi nelze zabránit, při použití speciálních metod ji lze detekovat. Tyto metody využívají asymetrických šifer.

Data pouze pro čtení (*Read-only Data*): Agent s sebou nosí mnoho dat, která jsou inicializována při jeho vytvoření a za běhu se nemění. Jedná se především o informace o něm (vlastníkovi, klientovi), zadání úkolu a podobně. Tato data je možné ochránit tak, že jakákoli manipulace s nimi bude odhalena.

Kontejner pro přidávání (*Append-only Container*): Agent často na své cestě nasbírá data, která pak uchovává po celý zbytek své existence (typicky se jedná o získané výsledky). Pro tyto účely může použít kontejner, do kterého lze jen přidávat bloky dat. Kromě tohoto vložení nelze provést žádnou operaci, která by obsah kontejneru měnila. Pokud některý hostitel změní obsah, tato změna bude detekována.

Realizace je opět založena na asymetrických šifrovacích metodách a je podrobně popsána v [Kar1998c].

2.9 Shrnutí

V této kapitole byly popsány hlavní zákonitosti, jimiž je potřeba se řídit při vývoji systémů mobilních agentů. Z pohledu systému se jedná hlavně o podporu mobility agentů, identifikaci součástí systému a jejich autentifikaci. Z pohledu agenta je třeba definovat jeho programový model, programovací jazyk a samozřejmě základní programová primitiva.

Kapitola 3

Existující systémy

Několik akademických a komerčních skupin v současné době zkoumá a vytváří systémy mobilních agentů. Nyní bude prezentována jejich podmnožina, uvedená podle přibližného chronologického pořadí, jak je prezentováno v [Kar1998a].

3.1 Výběr systému

Agenti mohou být během svého životního cyklu spuštěni na mnoha systémech. Samozřejmě nelze předpokládat, že všechny systémy budou mít stejnou architekturu nebo budou používat stejný operační systém. Z tohoto důvodu musí být agenti naprogramováni v jazyce, který je nezávislý na prostředí a je široce používán. Mnoho používaných aplikací vyžaduje internetový přístup ke zdrojům. Uživatel chce agenta odeslat např. z laptopu nezávisle na jeho fyzickém umístění. Tito agenti využívají zdroje umístěné kdekoli na síti. Z tohoto důvodu by měla být infrastruktura rozšiřitelná (a použitelná) na rozlehlé sítě. Přenositelnost a rozšiřitelnost jsou tedy hlavními požadavky při tvorbě systému mobilních agentů. Pro ilustraci použitých přístupů při jejich tvorbě budou použity příklady existujících mobilních systémů. V tabulce 3.1 je seznam těch nejznámějších spolu s URL.

3.1.1 Mobilita agentů

Hlavním znakem mobilních agentů je jejich schopnost samostatně migrovat z uzlu na uzel. Proto je také podpora mobility agentů základním požadavkem na infrastrukturu systému. Pokud se chce agent přesunout, musí se zjistit stav jeho výpočtu, systém jej musí zapouzdřit a odeslat na cílový uzel. Na cílovém uzlu se musí stav agenta obnovit, agent musí být spuštěn a tak dokončen celý proces migrace.

Systém	URL
Agent Tcl	http://agent.cs.dartmouth.edu/general/agenttcl.html
Aglets	http://www.trl.ibm.co.jp/aglets
Ajanta	http://www.cs.umn.edu/Ajanta
Concordia	http://www.concordiaagents.com/
D' Agents	http://agent.cs.dartmouth.edu
Grasshopper	http://www.grasshopper.de
JAE	http://www-i4.informatik.rwth-aachen.de/jae
JAS	http://www.java-agent.org
Knowbot	http://www.cnri.reston.va.us/home/koe
Messengers	http://www.ics.uci.edu/~bic/messengers
Mole	http://mole.informatik.uni-stuttgart.de
Tacoma	http://www.cs.uit.no/forskning/DOS/Tacoma
Voyager	http://www.objectspace.com/products/voyager

Tabulka 3.1: Některé systémy mobilních agentů

Stav agenta obsahuje všechna potřebná data, programový kód a stav výpočtu. Jestliže může být stav výpočtu odeslán spolu s agentem, cílový uzel aktivuje výpočet přesně v místě, kde došlo k jeho přerušení. Tato vlastnost může být využívána např. při transparentním **vyrovnávání zátěže** (*load balancing*), kdy dochází k vyrovnávání zatížení uzlů v systému, nebo také ve *fault-tolerant* programech, kde v případě poruchy dochází ke spuštění úlohy od kontrolního bodu. V případě zpracovávání dat může také dojít k přerušení agenta na vyšší úrovni (na úrovni jazykových instrukcí). Agent tak může kontrolovat svou migraci na cílový uzel. Většina současných systémů využívá pro běh agenta **virtuální stroje** (*virtual machine*) a programovací jazyky, které obvykle nepodporují přerušení agenta na úrovni vlákna.

Dalším problémem při implementaci mobility agentů může být serializace kódu agenta. Jednou možností je přenos celého agenta (kódu i stavu) najednou. Některé systémy nepodporují přenos kódu, ale vyžadují, aby kód agenta byl „předinstalován“ na cílovém uzlu. Z hlediska bezpečnosti má tento mechanismus tu výhodu, že nemůže být spuštěn neznámý kód. Naopak omezením je nízká flexibilita takového systému. Druhou možností je použití specializovaného uzlu, který poskytuje kódy agentů na vyžádání. Jakmile se během provádění agenta zjistí, že některá část jeho kódu chybí, je tento uzel zkontaktován a kód je přenesen.

3.1.2 Rozpoznání jména

Různé entity v systému jako agenti, sklady, zdroje atd. potřebují mít přiřazen jednoznačný identifikátor. Agent musí mít jednoznačné jméno, pomocí kterého

jej vlastník ovládá a komunikuje s ním. Sklad také potřebuje mít jednoznačné jméno, aby agent mohl přesně specifikovat cíl své migrace. Několik entit může sdílet jmenný prostor – např. agent a jmenný server. Tato vlastnost dovoluje jednoduše komunikovat v rámci jednoho skladu.

Systém musí poskytovat mechanismus pro nalezení současného umístění všech entit. Tento proces se nazývá **rozpoznání jména** (*name resolution*). Jméno entity může být závislé na jejím umístění, což dovoluje jednoduchou implementaci mechanismu rozpoznávání. Systémy jako *Agent Tcl* [Gra1996], *Aglets* [Ibm1998] a *Tacoma* [Joh1995] používají jména založená na jméně uzlu a čísle portu. Pro rozpoznání jména používají systém DNS [Moc1987]. Se změnou umístění agenta se změní i jeho jméno. Z hlediska vyhledávání agenta je tento postup poněkud nešikovný. Proto je žádoucí, aby bylo možné vytvářet jména entit, která jsou nezávislá na jejich umístění. Toho lze docílit dvěma způsoby:

1. Poskytovat lokální **zástupce** (*proxy*) pro vzdálené entity. Systém při každé změně umístění entity obnoví informace v lokálních zástupcích, což zajistí transparentnost jmen na aplikační úrovni. Např. *Voyager* [Obj1997] zajišťuje tímto způsobem pojmenování agentů, ale uzly jsou pojmenovávány pomocí DNS.

2. Zavést globální pojmenování entit nezávislé na jejich umístění v systému. Systém musí poskytovat jmenné služby, které budou mapovat symbolická jména entit na jejich fyzické umístění (např. *Ajanta* [Kar1998a] používá jednotné pojmenování všech entit v systému).

3.1.3 Bezpečnost

Mobilní agenti, kteří mohou migrovat počítačovou sítí, způsobují bezpečnostní problémy. V uzavřených sítích patřících jedné organizaci je možno důvěřovat okolí. Uživatelé pak budou ochotni akceptovat libovolné mobilní agenty, aby na jejich počítačích zpracovávaly svá data. Ale v otevřených sítích jako např. Internet, je takřka jisté, že agent a systém budou spadat do různých administrativních domén. Potom samozřejmě mají daleko menší vzájemnou důvěru. Může vzniknout několik typů bezpečnostních problémů:

- Server je vystaven nebezpečí průniku „zlomyslných“ agentů, kteří mohou prozradit citlivé informace.

- Citlivé informace obsažené v agentech mohou být ohroženy např. jejich napadením na nezabezpečené síti nebo se agent může dostat na nebezpečný server.

- Kód agenta, jeho data nebo výsledky výpočtu mohou být zneužity nebezpečným serverem.

- Agent může způsobit zahlcení serveru a tím odmítání dalších agentů, kteří by na něm chtěli běžet.

Systém mobilních agentů musí poskytovat několik bezpečnostních mechanismů pro detekci a odstranění těchto napadení. Ty zahrnují *přístupový mechanismus* (pro ochranu soukromého kódu a dat), *ověřovací mechanismus* (ověření identit komunikujících stran) a *autorizační mechanismus* (poskytuje agentům kontrolovaný přístup ke zdrojům na uzlu).

3.2 Výběr jazyka

Programovací jazyky podporující mobilní agenty budou zkoumány ze dvou hledisek: modelu programování a jazykových primitiv poskytovaných jazykem.

3.2.1 Programovací jazyky

Protože se agenti mohou pohybovat v heterogenním prostředí, přenositelnost jejich kódu je hlavním požadavkem. Z toho důvodu je většina systémů mobilních agentů založena na interpretovaných jazycích [Tho1997], které poskytují přenositelné virtuální stroje. Jiným důležitým kritériem je bezpečnost. Jazyky podporující typovou kontrolu, zapouzdření a výhradní přístup do paměti jsou vhodné pro implementaci bezpečného prostředí.

Několik systémů používá skriptovací jazyky jako *Tcl*, *Python* nebo *Perl* pro implementaci agentů. Jedná se o jazyky snadno zapamatovatelné a lehce použitelné, které umožňují rychlý návrh malých agentů. Mají vyzrálé prostředí umožňující rychlý přístup ke zdrojům na uzlu. Na druhou stranu se skripty vyznačují nesnadnou modularizací kódu, nedovolí programový kód zapouzdřit a jsou také méně výkonné. Je také velice obtížné vytvořit větší programový balík. Jiné systémy proto používají objektové jazyky jako *Java*, *Telescript* nebo *Obliq* [Tho1997]. Agenti jsou pak definováni jako objekty, které zapouzdřují jak stav tak programový kód a systém podporuje jejich migraci. Tyto systémy těží z výhod objektově orientovaného návrhu agentů.

Systémy mobilních agentů se také významně liší v použitém programovacím modelu. V některých případech je agent pouze skript často s žádnou nebo velmi malou kontrolou toku. Jindy je programovací jazyk rozšířen o vlastnosti objektově orientovaných jazyků (např. *Python*). V některých systémech se aplikace vytvoří jako množina distribuovaných interagujících objektů, z nichž každý má vlastní vlákno a tak může samostatně migrovat po síti. Jiné používají tzv. *call-back* programovací model, kdy systém signalizuje agentům různé události během jejich života. Agent je pak entita obsahující množinu metod pro obsluhu událostí.

Procedurální jazyky: Procedurální programovací styl je obecně nejpoužívanější a je považován za základní. Nabízí vše, co umožňuje operační sys-

tém a systémové knihovny. Pro implementaci mobilních agentů není příliš vhodný zejména z těchto důvodů:

- Většina procedurálních jazyků používá jeden jediný jmenný prostor. Kvůli zabránění kolizím tak musí být každý agent spuštěn jako zcela samostatný proces. To samozřejmě velice komplikuje celou strukturu systému a jeho kontrolu nad agenty.
- Program se může skládat z modulů pouze v omezené míře, většinou nemá žádnou strukturu. S takovým programem se velice obtížně manipuluje.
- Data nejsou nijak uspořádána, navenek jsou reprezentována jen jako datový prostor programu.

Funkcionální jazyky: Funkcionální jazyky jsou zřídka používané a omezují se spíše na speciální úlohy. Mnohé z nich jsou vhodné pro implementaci samostatného chování agenta, ale zajištění mobility a interakce s prostředím je velmi problematická.

Jediným funkcionálním jazykem, který se k implementaci mobilních agentů používá, je *LISP*. Ten sice patří mezi funkcionální jazyky, umožňuje však použití procedurálního programovacího stylu. Kombinuje tak výhody obou. Navíc je již tradičně používán v oblasti umělé inteligence pro implementaci inteligentních agentů [Sho1994].

Protože *LISP* je interpretovaný jazyk a navíc jen minimálně rozlišuje data a kód, nepředstavuje implementace podpory migrace žádný problém.

LISP má však také několik nevýhod:

- Chybí podpora pro paralelní provádění (podpora vláken). Tento problém se řeší kooperativním multitaskingem[1], který má ale zjevné nevýhody a je velice nebezpečný. V současné době již moderní implementace např. *CMUCL* [Cmu2001] vlákna podporují, nicméně jedná se stále o novinku.
- Neexistuje žádný způsob, jak omezit provádění programu a zaručit bezpečnost dat. Každý program (a tedy i agent) může dělat cokoli, a tedy i škodit nebo sledovat cizí data. I kdyby byli všichni agenti důvěryhodní, stále může být velká škoda napáchána programátorskou chybou.

  Tento problém je velmi závažný a je (mimo jiné) překážkou pro širší praktickou využitelnost systémů na bázi jazyka *LISP*.

[1] Existují varianty LISPu, speciálně určené pro provádění paralelních výpočtů na multiprocesorových strojích. Jedná se však o koncepci **SPMD** (*Single Program Multiple Data*), která je z hlediska mobilních agentů zcela nepoužitelná.

- *LISP* je v současné době spíše okrajová záležitost, mimo oblast umělé inteligence se prakticky nepoužívá.

Objektově orientované jazyky: Objektový programovací styl (v dnešní době obecně preferovaný) je pro tvorbu mobilních agentů nejvhodnější a také nejpoužívanější. Zvláště to platí o programovacím jazyce *Java*. Výhody objektově orientovaného přístupu jsou následující:

- Program má přehlednou modulární strukturu. Tato výhoda je v případě mobilních agentů obzvláště důležitá, protože na rozdíl od ostatních stylů je přesně definováno z čeho se agent skládá a lze s ním tedy výrazně jednodušeji manipulovat.
- Všechna data jsou uspořádána v objektech, takže jejich extrakce a přenos je jednodušší. Navíc některé jazyky (např. *Java*) používají standardní postup kódování dat.
- Koncepce zapouzdření umožňuje efektivní kontrolu nad operacemi prováděnými s daty. Lze tak zajistit ochranu a skrytí dat.
- Znovupoužitelnost, polymorfismus a dědičnost jsou vlastnosti, které obecně usnadňují tvorbu jakýchkoliv programů.

3.2.2 Provádění programu

Nativní kód: Nativní kód má výhodu ve vysoké efektivitě (rychlost, paměťové nároky), ale pro účely implementace mobilních agentů je nepoužitelný. Je totiž vždy pevně svázán jen s jedinou počítačovou architekturou a jedním konkrétním operačním systémem (případně s jejich omezenou množinou).

V případě, že je zaručeno homogenní výpočetní prostředí, je možné agenty v nativním kódu použít. Zůstává však ještě další nevýhoda – řízení běhu agenta lze provádět pouze na úrovni operačního systému. Systém mobilních agentů by tak musel velmi komplikovaně spolupracovat s operačním systémem, případně by musel být přímo jeho součástí.

Interpretovaný kód a bytecode: Mnohem výhodnější než nativní kód jsou interpretované jazyky. Umožňují větší kontrolu nad během agenta a poskytovanými službami (záleží samozřejmě na jazyku) a především heterogenní prostřední není problémem. Speciální jazyky mohou dokonce obsahovat podporu pro mobilní agenty.

Nevýhodou interpretovaného kódu je jeho malá efektivita, protože analýza kódu je poměrně náročná. Proto moderní interpretované jazyky používají tzv. částečný překlad. Při spuštění programu se provede jeho analýza a částečný překlad do **pseudoinstrukcí** (např. *bytecode*). Tyto pseudoinstrukce se pak provádějí mnohem rychleji, než kdyby se interpretoval přímo text.

Některé jazyky (resp. jejich implementace) dokonce umožňují překlad (alespoň částečný) do nativního kódu, čímž se dosáhne výrazně vyšší rychlosti běhu. Protože tento překlad probíhá až v okamžiku spuštění, nazývají se takové kompilátory **JIT** (*Just In Time*).

3.2.3 Programovací primitiva

V této kapitole budou popsána programovací primitiva potřebná pro implementaci aplikací využívajících agentů. Programovací primitiva se dají rozdělit na:

- Základní správa agentů: vytvoření, odeslání a migrace agenta.
- Komunikace mezi agenty a jejich synchronizace.
- Monitorování a kontrola agentů: zjištění stavu, obnovení a ukončení agenta.
- Odolnost proti poruchám: kontrolní body, obsluha výjimek, sledování agenta.
- Ochrana agentů a jejich bezpečnost: kódování, zabezpečení dat, ...

Vytvoření a odeslání agenta: Tato programovací primitiva dovolují programátorovi vytvářet agenty. Vytvoření agenta spočívá v jeho odeslání do systému. V objektově orientovaných systémech je agent vytvořen vytvořením instance třídy, která je implementací agenta. Systém dohlíží na to, aby byl předkládaný kód v pořádku a nemohl obcházet bezpečnostní politiku systému. Současně je agentovi přiřazen jednoznačný identifikátor.

Nově vytvořený agent je pouze pasivní část kódu, protože mu není přiřazeno vlákno. Aby se stal aktivním, musí být odeslán na uzel. Ten ověří agenta na základě jeho identifikace a přiřadí mu práva pro přístup ke zdrojům. Pak je mu přiřazeno vlákno.

Druhou variantou vytvoření agenta je vytvoření jeho kopie. Ta potom může běžet paralelně s původním agentem. Systém *Aglets* podporuje tzv. **klonování** (*cloning*). Jinou metodou je **rozvětvení** (*forking*) např. v systému *Agent Tcl*, kdy je nově vytvořený agent v relaci se svým rodičem. To dovoluje programátorovi vytvářet agenty, kteří dědí vlastnosti, privilegia atp. od svých rodičů.

Migrace agenta: Během provádění kódu agenta může dojít k situaci, že agent se chce přesunout na jiný uzel. Aby toto systém dovoloval, musí obsahovat potřebná programová primitiva. Uzel musí pozastavit běh agenta, zachytit jeho stav a přenést na cílový uzel. Na cílovém uzlu se musí agent přijmout, obnovit jeho stav a spustit se. Agent specifikuje buď *absolutní* odkaz na cíl (tj. agentem je specifikováno celé jméno uzlu) nebo *relativní* odkaz (agentem specifikované jméno musí být doplněno na celé). Většina systémů podporuje

absolutní odkazy. Systémy *Telescript*, *Tacoma* a *Ajanta* [Tri1998] podporují relativní odkazy. V tabulce 3.2 jsou shrnuta programová primitiva podporující přenos agentů poskytovaná vybranými systémy.

Systém	**Jmenné služby**	**Migrace agenta**
Telescript	Závislé na umístění (používá DNS)	Absolutní (go) i relativní (meet)
Tacoma	Závislé na umístění (používá DNS)	Jeden příkaz (meet) pro relativní i absolutní
Agent Tcl	Závislé na umístění (používá DNS) nebo symbolické jméno	Jenom absolutní (agent_jump), agent_fork odešle kopii agenta
Aglets	URL založené na DNS (závislé na umístění)	Jenom absolutní (dispatch)
Voyager	Globální identifikátor nezávislý na umístění	Jeden příkaz (moveTo) pro relativní i absolutní
Concordia	Závislé na umístění (používá DNS)	Jenom absolutní, závislost na obsahu *itineráře*
Ajanta	Globální identifikátor	Jeden příkaz (go) pro relativní i absolutní

Tabulka 3.2: Podpora mobility agentů v různých systémech

Komunikace mezi agenty a jejich synchronizace: Protože více agentů může být prováděno paralelně, musí se synchronizovat s ostatními. Komunikace mezi nimi může probíhat pomocí různých mechanismů. Jedním z přístupů je metoda **posílání zpráv** (*message passing*), kdy agent posílá asynchronní zprávy jednotlivě (v datagramech) nebo se navazuje spojení a zprávy jsou posílány v **proudech** (*stream*). Systém *Aglets* poskytuje pouze primitiva pro nespojované služby, kdežto *Agent Tcl* dovoluje používat oba typy posílání zpráv.

Volání metod (*method invocation*) je další metodou komunikace. Pokud jsou dva agenti umístěni na stejném uzlu, mohou si zjistit identifikátor opačné strany a komunikovat spolu pomocí volání metod. Například *Ajanta* a *Telescript* poskytují primitiva pro získání identifikátorů agentů, kteří jsou umístěni na stejném uzlu. Pokud jsou agenti na jiném uzlu, jedná se o *vzdálené volání metod*. Tento typ komunikace podporuje *Voyager*.

Skupinová komunikace se používá pro zaslání zprávy skupině agentů. Ostatní programovací primitiva pro koordinaci agentů jako např. bariéry mohou být implementovány pomocí tohoto typu komunikace. *Concordia* [Mit1997]

podporuje skupinovou komunikaci, ale typem zprávy může být pouze událost. *Voyager* používá hierarchický model zasílání zpráv, aby snížil zatížení sítě. Většina ostatních systémů nepodporuje sdružování agentů do skupin a tedy ani skupinovou komunikaci.

Komunikaci mezi agenty lze také uskutečňovat pomocí sdílených dat. Například v systému *Ajanta* mohou agenti přistupovat do sdílených objektů a tak mezi sebou komunikovat. To dovoluje agentům komunikovat asynchronně, i když nejsou přítomni na stejném uzlu. Podobně *Tacoma* poskytuje na každém uzlu sdílenou schránku nazývanou **skříň** (*cabinet*), do které může každý agent odložit sdílená data. Systém *Concordia* používá sdílené objekty pro implementaci synchronizačních primitiv.

Poslední možností komunikace agentů je metoda **zasílání událostí** (*event signaling*). Události jsou obvykle implementovány jako asynchronní zprávy. Agent může požádat systém o seznam událostí, které nastaly a z nich si vybrat tu, kterou potřebuje. Tento model chování se označuje jako **oznam-odeber** (*publish-subscribe*). Druhou možností je zasílání událostí skupině agentů. Systémy *Concordia* a *Voyager* podporují právě tento typ komunikace.

Monitorování a kontrola agentů: Systémy mobilních agentů potřebují monitorovat chování agentů na všech uzlech. Jestliže se v průběhu jejich výpočtu vyskytne chyba, systém musí mít možnost agenta ukončit. Proces ukončení agenta zahrnuje jeho vyhledání v systému a zaslání zprávy s požadavkem na jeho ukončení na vyhledaný uzel. Systém *Ajanta* poskytuje primitiva pro tento účel.

Vlastník může po agentovi požadovat, aby se vrátil na původní uzel pomocí funkce **návratu** (*recall*). K tomuto účelu může vlastník používat mechanismus zasílání zpráv nebo vyvolání výjimky. Systémy *Aglets* a *Ajanta* poskytují podobnou operaci *retract*, systém *Ajanta* navíc ještě poskytuje samostatnou operaci *recall*, která nepřeruší běh agenta, ale jenom jej požádá, aby se vrátil. Agent tak může sám přerušit svůj běh až to uzná za vhodné.

Schopnost vzdáleného ukončování a návratu se také dotýká bezpečnosti systému – jenom vlastník agenta jej může ukončit. Do těchto programovacích primitiv musí být vnořen ověřovací mechanismus tzn. systém musí mít jistotu, že objekt, který provádí tyto operace s agentem, je opravdu jeho vlastníkem. Jenom systém Ajanta poskytuje tento ověřovací mechanismus.

Aby vlastník zjistil, jestli má být agent požádán o návrat, musí průběžně zjišťovat stav agenta. Tyto dotazy mohou být zodpovězeny službou, která sleduje informace o stavu agentů na konkrétním uzlu. V tabulce 3.3 jsou shrnuta komunikační a kontrolní primitiva některých systémů.

Systém	Komunikace	Události a monitorování	Kontrola agentů
Telescript	Lokální volání metod	Události jsou podporovány na úrovni jazyka	Není podporována
Tacoma	Lokální výměna dat přes *briefcase* pomocí meet	Není podporováno	Není podporována
Agent Tcl	Posílání zpráv pomocí agent_send a agent_receive, spojované služby agent_meet a agent_accept	Události jsou stejné jako zprávy	Není podporována
Aglets	Odeslání/příjem Message objektů, synchronní, asynchronní i jednocestné zprávy	Není podporováno	Návrat agenta pomocí retract
Voyager	Podpora RMI, CORBA, DCOM, synchronní, asynchronní, jednocestná a skupinová komunikace	JavaBeans model událostí	Není podporována
Concordia	Lokální volání metod, integrace s CORBA, skupinová komunikace použitím konstruktoru AgentGroup	Oznam-odeber a skupinové události	Není podporována
Ajanta	Lokální volání metod přes proxy, RMI přes proxy	Stav agenta podporován serverem	Požadavek na návrat agenta recall, okamžitý návrat retract, kontrola přístupu

Tabulka 3.3: Komunikační a kontrolní primitiva

Systém	Bezpečná komunikace	Ochrana zdrojů na uzlech	Ochrana agentů
Telescript	Přenos agenta ověřován RSA a kódován RC4	Události jsou podporovány na úrovni jazyka	Není podporována
Tacoma	Není podporována	Není podporována	Není podporována
Agent Tcl	Používá PGP pro ověřování i kódování	Safe Tcl jako bezpečné prostředí, ověřování vlastníka není podporováno	Není podporována
Aglets	Není podporována	Statická přístupová práva, dvě bezpečnostní kategorie – *trusted* a *untrusted*	Není podporována
Voyager	Není podporována	Programátor musí rozšířit *SecurityManager*, dvě bezpečnostní kategorie – *native* a *foreign*	Není podporována
Concordia	Přenos agenta ověřován i kódován SSL	*SecurityManager* řídí přístup pomocí statických přístupových práv založených na identitě agenta	Agent je chráněný pomocí řízeného přístupu ke zdrojům
Ajanta	Přenos agenta ověřován a kódován DSA a ElGamal protokoly	Řízený přístup ke zdrojům, ověřování založeno na identitě vlastníka	Detekce chybových stavů agenta

Tabulka 3.4: Ochrana dat a bezpečnost agentů

Odolnost proti poruchám: Kontrolní bod představuje stav agenta, který může být uložen v permanentní paměti. Jestliže agent (nebo uzel) zhavaruje, vlastník spustí proces obnovy. Vybere se poslední kontrolní bod agenta a uzlu se pošle požadavek na restart agenta. Uzel může navíc udržovat informaci o přesunech agenta a vlastník tak zjistí, na kterém uzlu došlo k chybě. Systémy jako *Tacoma*, *Voyager* a *Concordia* podporují kontrolní body.

Jestliže agent obdrží výjimku, kterou neumí zpracovat, uzel mu ji pomůže zpracovat. Např. může poslat vlastníkovi zprávu a ten agenta ukončí nebo může agenta odeslat nazpátek vlastníkovi. To dovolí prozkoumat vlastníkovi stav agenta lokálně a případně jej restartovat. Tento postup podporuje systém *Ajanta*.

Bezpečnost agentů: Protože se agent může pohybovat nezabezpečenou sítí, systém musí poskytovat alespoň základní bezpečnostní programovací primitiva pro ochranu přenášených dat. Mezi ně patří kódování a dekódování dat a podepisování zpráv. Jestliže je k dispozici kryptování s použitím veřejného klíče, musí systém poskytovat primitiva pro bezpečnou výměnu klíčů. Identita agenta se pak prokazuje právě pomocí těchto prostředků. Ačkoli žádný z popisovaných systémů nedisponuje podobnými prostředky, některé z nich obsahují základy pro jejich implementaci. Souhrn vlastností jednotlivých systémů z hlediska bezpečnosti je v tabulce 3.4.

3.3 Příklady systémů mobilních agentů

3.3.1 Telescript

Telescript [Whi1995], vytvořený firmou General Magic, obsahuje objektově orientovaný jazyk se silnou typovou kontrolou. Servery (označované jako *places*) nabízí služby tak, že vytvoří stacionární agenty, kteří interagují s ostatními agenty. Agenti používají příkaz `go` pro absolutní migraci na uzel, jehož umístění je specifikováno službami DNS. Systém pak okamžitě přeruší běh agenta. Relativní migrace je také možná pomocí příkazu `meet`. Agenti na stejném uzlu spolu komunikují pomocí volání metod. Události jsou také podporovány.

Telescript podporuje bezpečnost agentů [Tar1996] prostřednictvím řízeného přístupu. Každý agent má jednu autoritu, která při každém pokusu agenta o přístup k prostředkům na uzlu, zkontroluje, zda k tomu má oprávnění. Ke každému prostředku je přidělena *kvóta*. Jakmile agent tuto kvótu překročí, je ukončen a vlastníkovi je zaslána výjimka.

Telescript nebyl komerčně úspěšný, protože vyžadoval, aby se programátor naučil nový jazyk. Firma General Magic nyní odložila další vývoj tohoto systému a pracuje na systému *Odyssey* [Gen1997] založeném na programovacím jazyce

Java. Tento systém používá stejnou kostru jako Telescript. Má stejný nedostatek jako ostatní systémy založené na jazyce Java – neumožňuje zachytit stav výpočtu na úrovni vlákna.

3.3.2 Tacoma

Tacoma [Joh1995] je společným projektem univerzity v Tromsø a Cornellovy univerzity. Tělo agenta je napsáno v jazyce Tcl, i když je možné využívat i jiné skriptovací jazyky. Stav agentů je uložen ve **složkách** (*folders*), které jsou soustředěny v **aktovkách** (*briefcases*). Agent je vytvořen, jakmile je vložen do speciální složky označené CODE. Jméno uzlu, kam se má agent přesunout, je uložené ve složce HOST. Absolutní migrace na cílový uzel se provede pomocí příkazu meet. Obsah —*aktovky* tj. složky CODE, HOST a další definované v těle agenta, je odeslán na cílový uzel. Systém nepřeruší běh agenta na úrovni vlákna, ale místo toho příkaz ag_tcl spustí agenta od začátku.

Agenti mohou také využívat příkaz meet pro komunikaci s okolím a pro výměnu aktovek mezi uzly. Je podporována synchronní i asynchronní komunikace. Alternativní metodou komunikace jsou tzv. **skříně** (*cabinets*), což jsou statické objekty, ve kterých mohou agenti odkládat a vybírat sdílená data. Nejsou implementovány žádné bezpečnostní mechanismy. Pro odolnost proti poruchám jsou v systému Tacoma implementovány kontrolní body a speciální agenti nazývaní **zadní hlídky** (*rear-guard*), kteří sledují agenty při jejich migraci [Joh1996].

3.3.3 Agent Tcl

Systém Agent Tcl [Gra1996, Kot1997] dovoluje Tcl skriptům migrovat mezi uzly. Modifikovaný interpret jazyka Tcl je používán pro spouštění těchto skriptů a zároveň umožňuje přerušit agenta na úrovni vlákna. Když dojde k migraci agenta, je tento přenesen včetně dat a stavu výpočtu. Migrace agenta je absolutní, místo určení je závislé na pojmenování uzlu. Je také možno agenta naklonovat a odeslat na vybraný uzel. Agenti mají identifikátor závislý na umístění (pro pojmenovávání se používá systém DNS), takže se při každé migraci musí změnit. Komunikace mezi agenty může být prováděna pomocí posílání zpráv nebo lze vytvářet proudy dat. Události jsou také podporovány, ale jsou identické se zprávami.

Agent Tcl používá bezpečné výpočetní prostředí *Safe Tcl* [Lev1995] umožňující bezpečný přístup k prostředkům na uzlu. Safe Tcl nedovolí agentům provádět nebezpečné operace na uzlu. Využívá k tomu přístupová práva – každému novému agentovi je přiřazeno stejné bezpečnostní pravidlo. Agent Tcl může využívat externích prostředků (*PGP* [Zim1995]) pro ověřování i kódování přenášených dat.

3.3.4 Aglets

Aglets [Ibm1998] je systém založený na jazyce Java. Agenti (v systému nazývaní *aglety*) jsou přenášeny mezi uzly (nazývanými *aglet context*). Hlavním rysem systému je tzv. *call-back* programovací model. Systém vyvolává metody agenta, jakmile dojde ke změně jeho stavu. Například pokud agent dorazí na nový uzel, automaticky se vyvolá jeho metoda onArrival. Programátor implementuje agenta tak, že oddědí metody z hlavní třídy Aglet a přepíše je svým vlastním kódem.

Migrace agentů je možná pouze na absolutní místo specifikované ve formátu URL. Mobilita agentů je podporována serializačními možnostmi jazyka Java. Zachycení stavu výpočtu na úrovni vlákna není podporováno. Jakmile je agent přenesen na cílový uzel, je vyvolána metoda run. Programátor tedy musí tuto metodu přepsat pro případnou další migraci. Posílání zpráv je jediná možnost komunikace mezi agenty – aglety nemohou vyvolávat metody jiných agletů. Zprávy mohou být synchronní, asynchronní i jednosměrné. Systém poskytuje příkaz retract, který způsobí návrat agenta na původní uzel. Nicméně nepodporuje přístupová práva k tomuto příkazu, takže každý uživatel může odeslat i cizí aglety! Systém Aglets podporuje bezpečnost pouze omezeně. Kompletní přehled bezpečnostní politiky agletů je v [Kar1997], ale dodnes není implementován.

3.3.5 Voyager

Voyager [Obj1997] je systém agentů založený na jazyce Java a vyvinutý firmou ObjectSpace. Zcela novým přínosem tohoto systému je utilita vcc, jejímž parametrem může být libovolná třída a výstupem je její vzdáleně přístupný ekvivalent označovaný **virtuální třída** (*virtual class*). Instanci virtuální třídy je pak možno vytvořit na vzdáleném uzlu a k metodám se přistupuje přes **virtuální odkazy** (*virtual references*). Tímto způsobem se docílí přístupu k třídě, který není závislý na jejím umístění. Tento mechanismus je použit pro implementaci agentů.

Agentovi je přiřazen globálně jednoznačný identifikátor a případně symbolické jméno. Jmenné služby vyhledají agenta buď podle identifikátoru nebo podle symbolického jména. Virtuální třídy poskytují metodu moveTo, která dovolí agentovi migraci na vybraný uzel. Cíl může být specifikován buď pomocí služeb DNS nebo pomocí virtuálního odkazu na objekt, s nímž chce být agent na stejném uzlu. Stav výpočtu není zachycen na úrovni vlákna, ale metoda moveTo obsahuje speciální metodu, která se vyvolá, jakmile je migrace agenta ukončena. Objekt forwarder zůstává na původním uzlu a zajišťuje kontakty s agentem na novém místě.

Komunikace mezi agenty je možná voláním metod pomocí virtuálních odkazů. Skupinová komunikace je možná, protože agenti mohou být sdružováni do hierarchických skupin. Jednoduché kontrolní body jsou také implementovány.

3.3.6 Concordia

Systém Concordia [Mit1997] vyvinutý firmou Mitsubishi Electric podporuje mobilní agenty napsané v jazyce Java. Stejně jako většina systémů využívá Concordia serializační možnosti jazyka Java pro podporu mobility agentů a také neumožňuje zachytit stav výpočtu na úrovni vlákna. Každý agent je svázán s objektem Itinerary, který obsahuje uzly systému (reprezentované jmény DNS) pro migraci agenta a také metody, které se mají na jednotlivých uzlech provádět.

Concordia má rozsáhlou podporu komunikace mezi agenty. Podporuje jak asynchronní zasílání událostí tak metody pro skupinovou komunikaci. Migrace agentů je bezpečná, je zaručeno jejich doručení. Jsou také implementovány kontrolní body. Uzly mohou omezit přístup ke svým prostředkům pomocí statických přístupových práv. Každý agent má vlastníka a při migraci se přenáší jeho zakódované heslo spolu s agentem. Každý uzel má přístup ke globálnímu souboru hesel, kde si může heslo vlastníka ověřit.

3.3.7 Ajanta

Ajanta je systém založený na jazyce Java vytvořený na univerzitě v Minnesotě. Pro podporu mobility využívá standardní serializace jazyka Java. Kód agenta je přenášen ze specializovaného serveru na vyžádání. Přenos kódu agenta i jeho stavu při migraci je zabezpečen použitím protokolu s veřejným klíčem. Absolutní i relativní migrace agenta je podporována jednotně a jmenné služby jsou využívány pro vyhledání entit s globálně jednoznačným jménem (*Uniform Resource Location – URN*).

Agent je spuštěn v izolované chráněné doméně, aby se zabránilo ovlivňování okolními agenty. Pro zdroje jsou vytvořeny proxy, které se dynamicky přizpůsobují každému agentovi. Stejný mechanismus je použit při bezpečné komunikaci mezi agenty pomocí vyvolávání metod. Komunikace po síti je také možná pomocí vyvolání vzdálených metod. Pomocí autorizačních metod můžeme agenta donutit k návratu (metoda recall) nebo jej ukončit. Ajanta také řeší problém ochrany agentů před nebezpečnými servery. Systém poskytuje bezpečný mechanismus pro detekci práv vlastníků agenta. Agenti také udržují část svého stavu pouze u sebe a mohou v pravidelných intervalech informovat specializovaný uzel.

3.3.8 Obecná specifikace – OMG MASIF

Konsorcium OMG (*Object Management Group*) se zabývá především systémy a prostředími pro distribuované výpočty v objektovém prostředí. Proto je jen logické, že se také (i když zatím bohužel jen okrajově) zabývají problematikou mobilních agentů. Výsledkem je specifikace *MASIF* (*Mobile Agent System Interoperability Facility*). Tato specifikace se však zabývá jen spoluprací hostitelů a konkrétně řeší:

- Správu a ovládání agentů (*Agent Management*), jejich vytváření, ukončování, zastavování (pasivace) a opětovné spouštění (aktivace).
- Přenos agentů mezi hostiteli, potencionálně i mezi hostiteli různých typů.
- Jména agentů a hostitelů.
- Obecně komunikaci mezi hostiteli.

Veškerá řešení využívají jiné doporučení a standardy OMG, především pak *CORBA* a její rozšíření. To s sebou nese řadu problémů při implementaci. Typickým příkladem je bezpečnost, která je zcela ponechána na použití *CORBA Security Service* — toto rozšíření však nepatří mezi rozšířené a často implementované (nachází se jen v několika málo komerčních implementacích).

3.4 Shrnutí

Existuje celá řada různých systémů mobilních agentů, ale většina z nich implementuje pouze některé vlastnosti mobilních agentů. Pouze systém *Ajanta* je navržen tak, aby obsahoval všechny vymoženosti mobilních agentů. Ten je ale komerční a navíc nevyužívá posledních známých standardů (je založen na technologii Java 1 a tudíž plně nevyužívá možností balíků *JAAS* a *JNDI*). Všechny systémy také nejsou schopné vzájemné spolupráce. Jedním z pokusů o standardizaci rozhraní pro mobilní agenty je *OMG MASIF*, který je postaven nad standardem *CORBA*.

Kapitola 4

Cíle práce

Tato práce byla motivována použitím mobilních agentů pro sběr dat ze sítě. Primárním cílem práce bylo vytvoření systému, který bude schopen shromažďovat a třídit data. Původním záměrem byl sběr dat v prostředí Jednotného identifikačního systému (JIS) na ZČU. Na základě zhodnocení současného stavu řešené problematiky byly formulovány požadavky na systém.

4.1 Popis cílů

Bezpečné výpočetní prostředí: Uzel systému i agent potřebují bezpečné výpočetní prostředí. Z hlediska agenta je potřeba zajistit jeho integritu během jeho migrace mezi uzly. Stejně tak musí systém ověřovat příchozí požadavky pomocí metod ověřování identity.

Pro nalezení strojů a agentů jsou použity jmenné servery. Pro přístup ke zdrojům (a nejen k nim) jsou používány služby. Pro přístup ke službám i pro komunikaci je použit mechanismus zástupců.

Agent je silně oddělen od stroje. Standardně má k dispozici jen mechanismus přidělování služeb, veškerá další komunikace s hostitelem probíhá pomocí služeb.

Spolupráce agentů: Agenti během svého života potřebují komunikovat se svým okolím ať už se jedná o agenty nebo jiné entity v systému. Pro podporu spolupráce je potřeba zajistit vhodné pojmenování a vyhledávání entit v systému (*jmenné služby*) a vytvořit komunikační infrastrukturu.

Vzdálená komunikace probíhá pomocí RMI. Komunikace agentů na jednom stroji je zajištěna lokálně, tj. bez použití RMI. Pro agenta jsou použitá metoda a protokol zcela transparentní.

Podpora migrace agentů: Migrace bude založena na metodě přenosu neaktivního agenta. Agent se sám označí, že chce migrovat a po přechodu do neaktivního stavu je migrace provedena.

Podpora multithreadingu: Agent může být prováděn ve více vláknech.

Využívání dostupných standardů: Pro dosažení přenositelnosti agentů, bezproblémové spolupráce mezi uzly systému a možnosti širokého výběru implementačních možností musí být systém založený na dostupném standardu.

Snadná konfigurace: Je stále více obtížné spravovat existující softwarové vybavení a přizpůsobovat ho potřebám konkrétní aplikace. V případě systémů mobilních agentů se tento problém týká nejen agentů, ale také systému samotného.

Rozšiřitelnost: Jedním z nejdůležitějších požadavků na systém agentů je jeho rozšiřitelnost. Návrh systému musí počítat s nároky uživatelů na jeho budoucí rozšiřování.

V neposlední řadě je potřeba umožnit odložení nevyužitých agentů z paměti a jejich opětnou aktivaci po příchodu požadavku na ně. Tím se odstraní blokace prostředků na uzlu.

Jednoduché použití: Uživatelé nebudou využívat rozsáhlé rozhraní systému (API) v případě, že budou chtít vytvořit agenty. Navíc nesmí být omezováni jednoduchým a přímočarým modelem programování, protože pak by nemohli využívat všech možností daného jazyka.

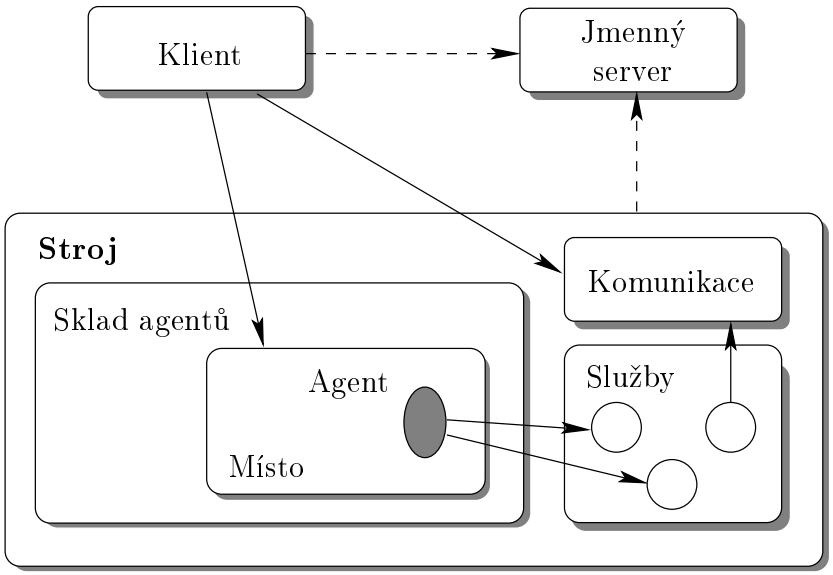
Kapitola 5

Struktura navrženého systému

SMAS (*Simple Mobile Agent System*) je systém mobilních agentů (*mobile agent framework*) v jazyce Java. Klade si za cíl jednoduchost a co nejširší použití standardních prostředků, které tento jazyk nabízí. Jedná se vícevrstvý systém, což umožňuje jasně definovat bezpečnostní politiku.

5.1 Složení systému

Základní přehled částí systému je na obrázku 5.1. Jednotlivé detaily budou popsány v dalších kapitolách.



Obrázek 5.1: Přehled částí systému

Systém obsahuje tři druhy součástí, které jsou typicky (není to zcela nutné) realizovány oddělenými programy a mohou být umístěny na různých uzlech. Jsou to:

- **Klient** – je program, který vytvoří agenta a vyšle jej. Během činnosti agenta s ním může komunikovat a po ukončení se agent opět navrátí na klienta.

  Klient nemusí být nutně v Javě, je možné ho napsat v libovolném jazyce. Musí však zvládat komunikační protokol pro komunikaci se strojem (*Java RMI* nebo *CORBA*) a se jmenným serverem. Taková implementace není programátorsky jednoduchá, ale může mít nižší hardwarové (a softwarové) nároky a proto by mohla mít své opodstatnění například u přenosných zařízení. Záleží na konkrétní implementaci, jestli to dovolí.

- **Jmenný server** – udržuje databázi strojů a agentů, u každého z nich pak momentální lokaci ve formě *JNDI URL*.

- **Stroj** – základní částí systému je **stroj** (*engine*), který realizuje funkci hostitele. Stroj realizuje služby pro agenty a také komunikační subsystém, ke kterému mají agenti přístup (prostřednictvím služby).

  Stroj obsahuje jeden nebo více **skladů** (*agent repository*). V těchto skladech se nacházejí **místa** (*place*), v každém místu jeden agent (nebo vyjádřeno obráceně: každý agent má své místo).

  Místo zprostředkovává styk mezi strojem a agentem. Směrem k agentovi reprezentuje stroj (umožňuje přidělování služeb), směrem „ven“ ke stroji reprezentuje agenta (uchovává informace o něm a umožňuje komunikaci).

  Jednotlivé sklady se liší tím, jakým způsobem (podle jaké politiky) přidělují práva agentům.

5.2 Identifikace a adresování

V SMASu jsou výrazně rozlišena jména a adresy. Obojí má jiný účel a jiné vlastnosti (liší se především druhem vazby).

5.2.1 Jména

Jméno se používá pro jednoznačnou identifikaci a jeho vazba tedy musí být 1 : 1 (viz kapitola 2.6.1).

Všechna jména používaná v SMASu mají pro člověka srozumitelnou formu, tedy řetězec (String). Jedinou výjimkou jsou služby, používající speciální mechanismus identifikace viz kapitola 5.5.1.

Jména stroje a agenta jsou strukturovaná a jejich řetězcová forma je sekundární (neměla by se používat pro jiné účely než pro zobrazení uživateli). Primární

formou je pak objekt reprezentující strukturované jméno podle standardu *JNDI*: javax.naming.Name (resp. implementující třída).

Definovaná jména jsou:

- **Uzel** – používá se jméno stroje, tedy typicky primární DNS jméno. Pokud není k dispozici, použije se IP adresa. Jménem je vždy řetězec.

- **Stroj** – jméno stroje je třídy Name a musí obsahovat jméno uzlu, na kterém běží. Pokud je jediný, může obsahovat *pouze* jméno uzlu. Pokud je strojů na uzlu více, musí jméno obsahovat ještě další komponenty, které jej od ostatních odliší.

  Vnitřní struktura (počet a pořadí komponent jména) není definována. Se jménem se vždy pracuje v jeho řetězcové formě a předpokládá se, že má pouze jednu komponentu.

- **Agent** – jméno agenta musí být reprezentováno třídou ComponentName a má tři složky (důvody viz složené jméno, kapitola 2.6.3). První složka je jméno stroje, na kterého byl agent vyslán (tento stroj jméno přiděluje). Druhá složka je řetězec dodaný klientem, který by měl označovat aplikaci nebo druh agenta (obsah tohoto řetězce záleží výhradně na klientovi). Třetí složkou je řetězec, který je pro každého agenta vygenerován a v rámci daného stroje je jedinečný. Tato poslední složka je typicky pořadové číslo agenta.

  Třída CompoundName přesně definuje, jak vypadá řetězcová forma a umožňuje převod jak do ní tak z ní. Jméno agenta tak může vypadat přibližně takto:

  jumbo.kiv.zcu.cz/hledani/123456

5.2.2 Adresy

Adresa umožňuje nalézt entitu a navázat s ní komunikaci. Vazba je většinou $1 : N$, protože entitu lze adresovat několika způsoby (několika protokoly, různé formulace apod.).

Jednoduchá adresa je URL, tedy řetězec. Entita však často má těchto adres víc a sdružují se do tzv. **kompletní adresy**. Pojmem adresa v dalších kapitolách je většinou míněna právě kompletní adresa.

Kompletní adresa je reprezentována třídou javax.naming.Reference, která obsahuje několik JNDI adres. Ty jsou ve většině případů právě URL, ale implementace musí povolovat i použití jiných druhů adres (pokud je podporuje některý *JNDI provider*). Implementace však může takovéto adresy prostě ignorovat, není povinna je nijak zpracovat.

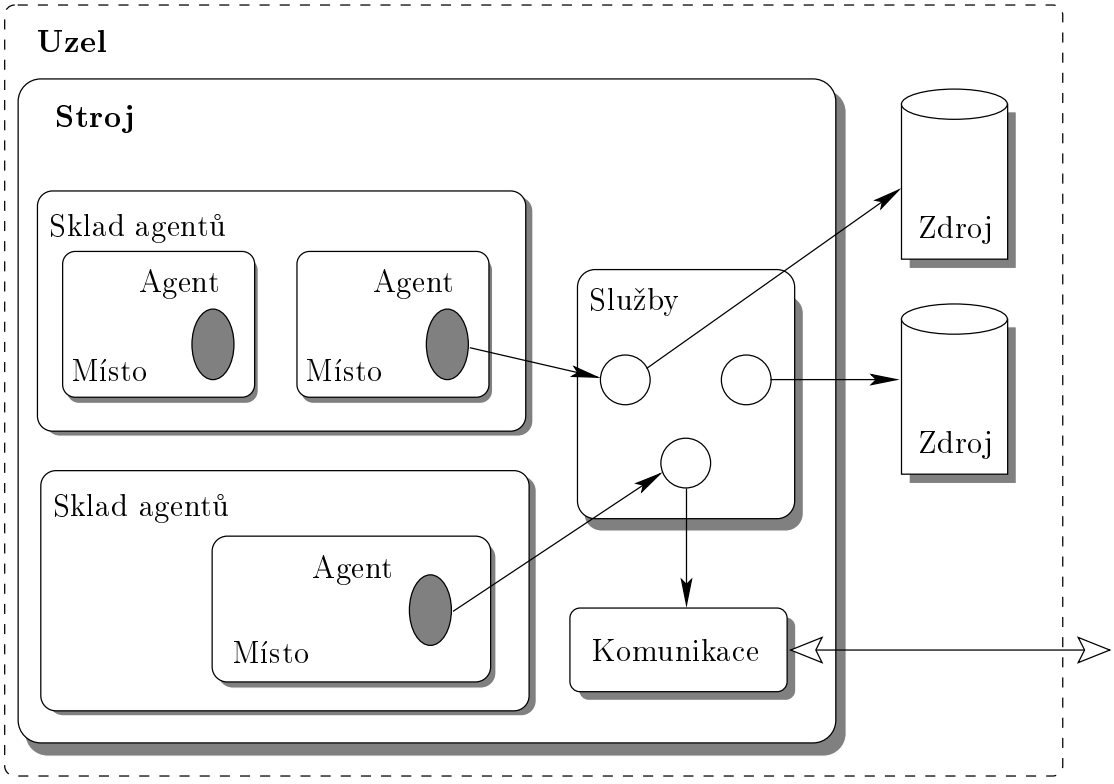
Převod jména na adresu je prováděn pomocí jmenného serveru. Pro opačný převod je nutné navázat s entitou komunikaci a ta sdělí své jméno. Tímto způsobem se zároveň ověří, zda adresy označují stejnou entitu.

Podrobnější informace o adresách jsou v kapitole 5.7.3.

5.3 Hostitel

Pojem **hostitel** (*host*) je velice široký a zahrnuje v sobě nejenom program realizující funkce potřebné pro funkci mobilních agentů, ale i uzel, na kterém běží. Na úrovni designu takto vágní pojem není možné použít a většinou se používá pojem **stroj**.

Stroj je *JVM* spuštěná na uzlu a v ní běžící program. Hrubá struktura je znázorněna na obrázku 5.2.



Obrázek 5.2: Struktura stroje

5.3.1 Sklad

Sklad (*agent repository*) je kontejner, do něhož se vkládají agenti (ne přímo, ale se svým místem – viz další podkapitola). Stroj může obsahovat jeden nebo více skladů.

Každý sklad má svoje jméno. Toto jméno je řetězec (String) a jediné nároky na jeho obsah jsou takové, aby jej mohl uživatel snadno používat. Jméno nesmí být prázdné a mělo by obsahovat jen tisknutelné znaky (tedy ne mezery).

Jeden sklad na hostiteli má výsadní postavení: je **implicitní** (*default*), tj. všichni příchozí agenti se implicitně umísťují do něj. Prázdné jméno (null nebo prázdný řetězec) vždy označuje právě tento sklad (samozřejmě sklad má i normální jméno). Pokud chce klient vyslat agenta (nebo agent při migraci) do jiného skladu, musí explicitně specifikovat jméno tohoto skladu. Jinak bude vložen právě do implicitního skladu.

Sklad zajišťuje vkládání a rušení agentů, vytváření a rušení míst, migraci, pasivaci a aktivaci agentů. Udržuje také množinu nabízených služeb.

Jednotlivé sklady se liší množinou nabízených služeb a bezpečnostní politikou. Ačkoliv většina služeb je globální pro celý stroj (to neplatí pouze pro individuální služby, viz další podkapitola), ne všechny jsou nabízeny všem agentům. Správce stroje při konfiguraci rozhodne, které služby budou v tomto skladu nabízeny a které ne. Každý sklad má také zcela vlastní metodiku (konfiguraci), podle které agentům přiděluje práva a omezení.

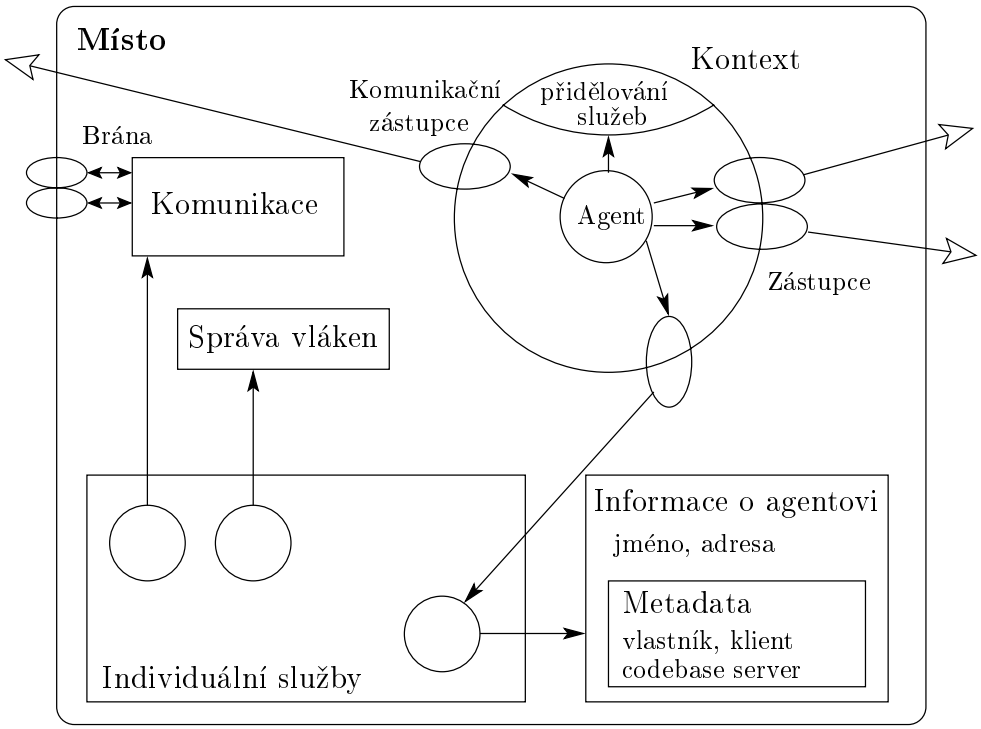
Sklad také může mít požadavky na agenty – přijetí agentů jen od určitých vlastníků či klientů, jen s kódem podepsaným určitou autoritou a podobně. Pokud agent těmto požadavkům nevyhovuje, může sklad odmítnout jeho přijetí. Toto však neplatí pro implicitní sklad, který si nesmí klást žádné speciální požadavky na agenta (základní ověřování důvěryhodnosti agenta je prováděno již v komunikačním subsystému při přijímání).

Smyslem výše uvedené architektury je umožnit oddělení různých aplikací. Implicitní sklad slouží pro všechny agenty bez rozdílu účelu. Tento sklad nabízí jen omezenou množinu služeb a základní bezpečnostní pravidla. Pro speciální aplikace existují další sklady, z nichž každý nabízí rozšířenější množinu služeb. Tyto sklady však mohou vyžadovat od agentů speciální autorizaci.

5.3.2 Místo

Jak bylo ukázáno v kapitole 2.8.2 a 2.8.3, agenty je nutné oddělit od stroje i od ostatních agentů. Zároveň je také nutné každému agentovi zajistit přidělování služeb a další prostředky nutné pro jeho funkci. Toto vše zajišťuje **místo** (*place*). Struktura místa je znázorněna na obrázku 5.3.

Agent sám je uzavřen v tzv. **kontextu**. Tento kontext je to jediné, s čím může agent komunikovat přímo (tj. voláním metod). Veškerý ostatní styk s okolím je prováděn pomocí zástupců. Kontext sám pak umožňuje pouze přístup ke službám.



Obrázek 5.3: Struktura místa

Bez použití služeb nemůže agent dělat nic (dokonce ani nezná vlastní jméno). Ke každé přiřazené instanci služby se vytvoří nový zástupce.

Kromě přístupu k externím službám (nabízených skladem, na tomto obrázku nejsou) nabízí místo i speciální *individuální služby*. Tyto služby slouží jen pro zprostředkování komunikace s různými subsystémy místa. Jsou vytvořeny místem při jeho vzniku a při zániku místa jsou zničeny. K individuální službě místa může přistupovat pouze a jen agent umístěný v tomto místě; dochází tak automaticky k autentifikaci a autorizaci. Podrobněji viz kapitola 5.5.3.

Informace o agentovi jsou uloženy odděleně a agent sám k nim má přístup pouze pomocí speciální služby. Tyto informace obsahují jednak **metadata** vytvořená klientem (viz kapitola 5.4.3), dále pak jméno a adresu agenta a aktuální informace (jméno skladu, kde se právě nachází). Tímto oddělením agenta od vlastních informací je dosaženo toho, že plnou zodpovědnost za zabezpečení informací přebírá stroj. Pro autentifikaci a autorizaci se vždy používají tyto údaje a tudíž ani agent sám nemá možnost si je měnit a měnit tak svoji identitu. Omezí se tak možnosti zákeřných agentů, aniž by se omezila normální funkce (tyto informace musí být *vždy* zaručeně dostupné a nic nebrání agentovi, aby je pro své interní účely měl kopie přímo u sebe).

Místo zajišťuje běh agenta, tedy přidělování vláken. O přidělení vlákna si musí agent požádat prostřednictvím speciální služby. Přidělen mu může být jen

omezený počet vláken, jak je tento počet vysoký rozhoduje sklad. Agent musí být také kdykoli připraven ukončit všechna svoje vlákna.

Důležitou funkcí agenta je zajištění komunikace s agentem (komunikaci agenta s okolím zajišťuje přímo stroj). Místo totiž navenek agenta reprezentuje. Při navázání spojení s agentem se ve skutečnosti naváže spojení s jeho místem. Pro tento účel místo udržuje jednu nebo více **bran** (*gate*), které slouží právě k navázání spojení viz kapitola 5.7.2. Agent sám může zaregistrovat svůj vlastní komunikační interface.

5.4 Agent

Při tvorbě agenta je nutné vzít v úvahu především dvě okolnosti:

- Agent bude prováděn ve svém *kontextu* a musí tedy být schopen s ním správně komunikovat (viz kapitola 5.3.2). Pro tento účel je použit standardní balík *Java beans*, zvláště pak koncepce *kontextu* (do kterého se *bean* vkládá). Celý agent se navenek jeví jako jeden *bean*.
- Při různých příležitostech je nutné zjistit kompletní stav agenta. K tomu slouží **serializace**. Agent musí umožnit svoji serializaci. Podrobněji viz kapitola 5.4.2.

Vnitřní struktura agenta není nijak specifikována, záleží jen na rozhodnutí programátora. Obecně platí, že se skládá z libovolného množství objektů (v krajním případě jen jednoho). Je sice vhodné agenta také složit z *beanů*, ale není to nijak nutné.

Agent se samozřejmě během svého života mění a jeho struktura nemusí být stále stejná. Kromě stavu `Aktivní` však vždy musí splňovat všechny výše uvedené požadavky.

5.4.1 Hlavní objekt

Pro své okolí je agent reprezentován jediným objektem, nazývaným **hlavní objekt** (*main object*). Ten musí implementovat interface smas.agent.MainAgentObject, který právě označuje hlavní objekt agenta. Nedeklaruje žádné metody ani atributy a jediný interface, od kterého dědí, je Serializable. Ten označuje, že agent může být serializován.

Pouhá implementace MainAgentObject však ani zdaleka nepostačuje. Hlavní objekt agenta se musí také správně chovat jako *Java bean*, který lze vložit do *bean kontextu* [Mat1999]. Prakticky to znamená, že musí implementovat jeden (právě jeden!) z interface uvedených ve výpisu 5.1 (tento výpis není kompletní!).

Obě možnosti jsou povoleny a nelze implementovat oba interface najednou. Liší se tím, kdo danou funkčnost implementuje – zda přímo tento objekt, nebo nějaký jiný objekt.

```
public interface BeanContextProxy {
    BeanContextChild getBeanContextProxy();
}

public interface BeanContextChild {
    void setBeanContext(BeanContext bc) throws PropertyVetoException;
    BeanContext getBeanContext();

    void addPropertyChangeListener(String name, PropertyChangeListener pcl);
    void removePropertyChangeListener(String name, PropertyChangeListener pcl);
    void addVetoableChangeListener(String name, VetoableChangeListener vcl);
    void removeVetoableChangeListener(String name, VetoableChangeListener vcl);
}
```

Výpis 5.1: Třídy BeanContextChild a BeanContextProxy

Použitím BeanContextProxy lze veškerou činnost delegovat na jiný objekt. Tento objekt musí být navrácen jako výsledek metody getBeanContextProxy. Použití tohoto způsobu je vhodné, pokud je hlavní objekt potomkem některé třídy a plná implementace BeanContextChild by byla příliš obtížná nebo nežádoucí. V ostatních případech je lepší implementovat přímo BeanContextChild.

V interface BeanContextChild jsou důležité první dvě metody. setBeanContext má jako parametr kontext, do kterého je agent vkládán. Hodnota null znamená, že je vyjímán. Ačkoli podle definice interface má objekt právo nastavovanou hodnotu odmítnout vygenerováním výjimky, v tomto konkrétním případě agent hodnotu odmítnout *nesmí*. Pokud metoda vygeneruje výjimku, agent se považuje za defektní a bude zrušen.

Metoda getBeanContextProxy umožňuje zjistit v jakém kontextu se agent nachází. Po kontextu pak lze žádat přidělení služeb (viz kapitola 5.5). Je to vlastně jediná přímá reference na objekt ve stroji, kterou má agent k dispozici.

5.4.2 Serializace

Strukturu agenta lze znázornit orientovaným grafem, kde uzly jsou objekty a hrany jsou reference (kromě atributů označených jako transient). Při serializaci některého objektu se uloží nejen tento objekt, ale také všechny objekty, které jsou z něj dosažitelné[1]. Podrobnější informace o serializaci lze nalézt v [Lin1996] a [Sun2001c].

Při vytváření zakódovaného stavu agenta se serializuje jeho hlavní objekt. Vzhledem k výše uvedenému je zřejmé, že *každý* potřebný objekt agenta musí být dosažitelný z hlavního objektu. Pokud by některý objekt takto dosažitelný nebyl, nebude uložen! Naopak, pokud je nežádoucí některý objekt uchovávat (typicky

[1]Samozřejmě kromě **serializace** je možno použít i **externalizaci** viz dokumentace k java.io.Externalizable [Gos1996, Sun2001a].

objekty, které nejsou serializovatelné), musí být atributy obsahující referenci na něj označeny jako transient.

Agent by měl být připraven na to, že může být serializován prakticky kdykoli mimo dobu aktivního života (když má k dispozici nějaká vlákna).

Pokud serializace selže (typicky při pokusu o serializaci objektu, který serializovat nejde), pokládá se to za chybu agenta (je v nekonzistentním stavu). Serializace by pro agenta měla být zcela transparentní. Pomocí deklarace některých metod však agent může zjistit, že je serializován. Na tento fakt by neměl nijak reagovat (pouze může provést speciální nestandardní kódování).

5.4.3 Části uchované odděleně

Informace o agentovi: O agentovi je nutné znát a uchovávat některé standardní údaje, které ho charakterizují (jako je například jméno nebo identita vlastníka). Tyto údaje samozřejmě mohou být interní součástí agenta, který je pak na požádání sděluje. Tento přístup je však nejenom nepohodlný (z hlediska stroje), ale také málo bezpečný – umožňuje agentovi měnit svoji identitu a své vlastnosti. To je však zcela nepřípustné a ani by tuto možnost nešlo využít žádným korektním způsobem (tj. lze ji využít jen pro pokus o napadení).

Z těchto důvodů jsou ve SMASu tyto informace uchovávány odděleně (viz kapitola 5.3.2). Navíc se dělí podle způsobu jejich vzniku na dva druhy:

- **Metadata** – vytváří klient a obsahují základní údaje o agentovi: vlastníka, tvůrce (klienta) a umístění kódu. Všechny tyto údaje jsou pak klientem podepsány a pomocí tohoto podpisu pak může být ověřena jejich pravost (toto ověření je však bezpečné jen při známé identitě klienta, protože metadata lze podvrhnout jako celek).
- **Jméno a adresa** – tyto údaje jsou vygenerovány strojem při vyslání agenta a nemohou být tedy součástí metadata. Podrobnosti o jméně agenta a adresách jsou v kapitole 2.7.

Tyto údaje jsou teoreticky součástí kompletního agenta, protože bez nich nemůže existovat. Fakticky se však o jejich uchování stará výhradně stroj a jsou přenášeny odděleně (ale vždy) od samotného agenta[2]. Z těchto důvodů nejsou v rámci terminologie SMASu zahrnuty do pojmu **agent**.

Metadata jsou kompletně uložena ve třídě AgentMetaData. Nejedná se o interface, protože objekt musí být plně přenositelný a navíc nejsou přípustné žádné odchylky v chování. Důležité metody (bez implementací) jsou ve výpisu 5.2. Všechna data se nastavují konstruktorem a poté je již měnit nelze, jen číst.

[2]Při vyslání se samozřejmě posílají jen metadata a samotný agent.

Metadata mají zásadní význam pro autentifikaci a autorizaci a proto jsou podrobněji popsána v kapitole 2.6.4.

```
package smas.host;

public class AgentMetaData implements Serializable {
    public AgentMetaData(Subject owner, Subject creator,
                         Reference codeBase, Properties cfg);
    public Subject getOwner();
    public Subject getCreator();
    public Reference getCodeBase();
    public String getProperty(String name);
}
```

Výpis 5.2: Třída AgentMetaData

Programový kód: Téměř veškerou režii spojenou s přenosem programového kódu zajišťuje přímo Java. Kód může být uložen ve formě jednotlivých class souborů nebo ve formě jar archívu a adresuje se pomocí *URL* některým ze standardních protokolů (*HTTP, FTP*).

Informace o veškerém programovém kódu je v metadatech (výpis 5.2) a nazývá se *codebase*. Reprezentována je objektem typu javax.naming.Reference, nejedná se však o adresu. Tento objekt je používán jako pouhé pole *URL*.

Každý agent má svůj vlastní *classloader* (blíže k tomuto pojmu viz [Gos1996, Lin1996, Sun2001a]). V Javě to znamená, že má také vlastní instanci tohoto kódu. Nemůže tak dojít ke konfliktu verzí ani k úmyslnému napadení pomocí upraveného kódu.

5.4.4 Aktivní činnost

Zcela výjimečné postavení mezi prostředky, které jsou agentovi poskytovány, mají **vlákna** (*threads*). Veškerá aktivní činnost agenta by měla být prováděna v přidělených vláknech. Při jiném vyvolání (např. při komunikaci, při zpětném volání z některé služby) někdo jiný čeká na návrat. Navíc, v přidělených vláknech má agent všechna dostupná práva (viz kapitola 5.6.3).

Pro účely funkce agentů je standardní systém používající třídu Runnable zcela nedostačující. Chybí mu totiž tyto požadované vlastnosti:

- *Možnost bezpečného zastavení:* Existuje sice možnost vlákno ukončit metodou Thread.stop, ale toto násilné řešení by ponechalo agenta v nekonzistentním stavu (nicméně stále zůstává poslední variantou pro případ, že by agent odmítal spolupracovat). Z těchto důvodů je metoda stop v nových verzích *JDK* označena jako **zastaralá** (*deprecated*) a neměla by se používat.

- Počet vláken používaných agentem v jednom okamžiku musí být omezen.

- *Možnost vygenerování výjimky:* Jednotlivá vlákna se tak nemusí starat o ošetření všech výjimek.

Navíc je vhodné, aby jedna instance (lépe řečeno její metoda run) nebyla prováděna více vlákny najednou.

Pro provádění vláken v agentovi slouží interface třídy smas.agent.RunnableAgent (viz výpis 5.3). Instance třídy implementující tento interface obsahuje kód prováděný *jedním* vláknem. V rámci agenta tak reprezentuje samotné vlákno.

- **metoda run** – Obsahuje samotný prováděný kód. Odpovídá stejnojmenné funkci v interface Runnable, ale umožňuje vygenerovat libovolnou výjimku.

  Pokud se aplikace chová korektně, měla by při vyvolání této metody zjistit, zda již není prováděna v jiném vlákně. Pokud je tomu tak, měla by vygenerovat výjimku AlreadyRunningException.

- **metoda stop** – Vyvolání této metody (z jiného vlákna než je prováděné) by mělo způsobit korektní ukončení běhu. Implementace pravděpodobně použije nějakou formu příznaku.

- **metoda isRunning** – Vrátí logickou hodnotu, zda je kód prováděn.

- **metoda isStopped** – Vrátí logickou hodnotu, zda bylo vlákno zastaveno pomocí metody stop. Vrácená hodnota true ještě neznamená, že vlákno skončilo. Naopak, vrácená hodnota false neznamená, že vlákno běží (mohlo také skončit bez vnějšího zásahu).

```
package smas.agent; public interface RunnableAgent {
    void run() throws Exception;
    void stop();
    boolean isRunning();
    boolean isStopped();
}
```

Výpis 5.3: Interface třídy smas.agent.RunnableAgent

Ve většině případů není nutné implementovat RunnableAgent, protože stačí vytvořit potomka třídy AbstractActive a přetížit metodu readRun.

5.5 Služby

5.5.1 Přidělování služeb

Pro přidělování služeb se využívá koncepce *bean kontextů*. Jak již bylo uvedeno v kapitole 5.4.1, agent (resp. jeho hlavní objekt) musí implementovat chování BeanContextChild (respektive BeanContextProxy). Kontext, do kterého je agent vkládán, musí implementovat smas.host.AgentContext, který rozšiřuje BeanContextServices. Tento kontext pak umožňuje registrovat a přidělovat služby.

```
package java.beans.beancontext;

public interface BeanContextServices extends BeanContext {

    Object getService(BeanContextChild child, Object requestor, Class serviceClass,
                      Object serviceSelector, BeanContextServiceRevokedListener bcsrl)
        throws TooManyListenersException;
    void releaseService(BeanContextChild child, Object requestor, Object service);

    boolean hasService(Class serviceClass);
    Iterator getCurrentServiceClasses();
    Iterator getCurrentServiceSelectors(Class serviceClass);

    void addBeanContextServicesListener(BeanContextServicesListener bcsl);
    void removeBeanContextServicesListener(BeanContextServicesListener bcsl);

    boolean addService(Class serviceClass, BeanContextServiceProvider serviceProvider);
    void revokeService(Class serviceClass, BeanContextServiceProvider serviceProvider,
                       boolean revokeCurrentServicesNow);
}
```

Výpis 5.4: Interface třídy BeanContextServices

Každá služba je identifikována dvěma parametry: třídou (*class*) a *selectorem*. Třída určuje typ služby, konkrétně pak přímo jaký interface tato služba implementuje. *Selector* pak umožňuje výběr z více instancí. U mnohých služeb existuje jen jedna instance a *selector* nemá smysl – v takovém případě bývá null.

Ačkoli *beancontext API* umožňuje, aby byla použita jakákoli třída, v SMASu lze použít jen interface (tj. nikoliv třídy). Toto omezení je nutné především proto, aby bylo možné dynamicky vytvářet zástupce.

Přidělení služby provádí metoda getService. Jako parametry se kromě identifikace žádané služby předají autentifikační parametry. Konkrétně se jedná o referenci na objekt vložený v tomto kontextu (v případě agenta je to většinou jeho hlavní objekt) a *requestor*, tedy konkrétní objekt, který službu vyžaduje (služba ho může použít pro autorizaci, prakticky však není nijak použitelný). Posledním parametrem je tzv. *listener*, tedy objekt, kterému bude zaslána zpráva o zrušení služby. Jediné povinné parametry, které nesmí být null, jsou třída služby a objekt vložený v kontextu.

Návratovou hodnotou metody getService je instance služby (respektive zástupce, viz další podkapitola). Tento objekt implementuje interface předaný jako parametr určující třídu služby (serviceClass). Pokud nebyla služba nalezena, návratová hodnota je null.

Uvolnění služby se provádí metodou releaseService. Parametry jsou obdobné jako při přidělování, service je zástupce vrácený metodou getService. Zda je nutné službu uvolnit nebo ne se liší podle jednotlivých služeb – u některých to je nutné (například přístup do databáze, kdy je nutné uvolnit spojení), u jiných je to zcela

zbytečné a při uvolňování se žádná akce neprovede (to je případ individuálních služeb, které pouze zpřístupňují interní objekty a informace).

Informaci o nabízených službách lze získat pomocí metod getService, getCurrentServiceClasses a getCurrentServiceSelectors. Pro služby, které není nutné uvolňovat, je však vhodnější použít *listenery* (viz níže).

Příklad použití služby i s přidělením a uvolněním je vidět na výpisu 5.5.

```
AgentContext ctx = (AgentContext)mainObj.getBeanContext();
AgentInfoService srv;

try {
    srv = ctx.getService(mainObj, this, AgentInfoService.class, null, this);
    if (srv!=null) {
        // OK, služba přidělena, použiji ji
        System.out.println("My name is "+srv.getName());

        // AgentInfoService se sice uvolňovat nemusí, ale pro pořádek
        ctx.releaseService(mainObj, null, srv);
    } else {
        // služba není k dispozici
        // v případě AgentInfoService to však není možné, jedná se o povinnou službu
    }
} catch (TooManyListenersException ex) {
    // nemůže nastat, protože poslední parametr getService() byl null
}
```

Výpis 5.5: Příklad použití služby

5.5.2 Změna nabízených služeb

Množina nabízených služeb se během agentova života mění, a to nejen v případech, kdy se služba skutečně stává nedostupnou a opět dostupnou. Nejčastější příčinou této změny je případ, kdy je agent ve stroji spouštěn (vyslání, migrace, aktivace) nebo ukončován (migrace, pasivace, ukončení).

V okamžiku vložení do kontextu (setBeanContext, viz kapitola 5.4.1) nejsou k dispozici žádné služby. Ty jsou až poté postupně přidávány (viz kapitola 5.8.1). Vyjmutí pak probíhá opačně – služby jsou nejprve všechny zrušeny a teprve potom je agent vyjmut.

O těchto změnách může být agent informován. O zrušení služby dokonce v praxi být informován musí, protože všechny přidělené služby tohoto typu je *nutné* uvolnit (pokud k tomu nedojde dobrovolně, uvolní se pomocí zástupců).

Oznámení probíhá prostřednictvím tzv. *listenerů*. *Listener* je objekt, který implementuje patřičný interface a zaregistruje se u zdroje události. Když je událost vygenerována, zavolá se jeho patřičná metoda.

V tomto konkrétním případě se *listenery* přidávají a ubírají pomocí metod

addBeanContextServicesListener a removeBeanContextServicesListener

Tuto registraci je vhodné provést v okamžiku, kdy je agent vložen do kontextu. Bude pak upozorněn na všechny služby, které budou k dispozici. O služby, které bude potřebovat často a není nutné je ihned uvolňovat (tedy typicky individuální služby), je tak možno požádat ihned jakmile jsou k dispozici. A naopak, jakmile je služba zrušena, agent ji musí uvolnit.

K tomuto účelu slouží pomocná třída smas.agent.ServiceHandler. Instance této třídy se zaregistruje jako *listener* a potom automaticky žádá o přidělení nebo službu uvolňuje. Navíc ošetřuje i případ, kdy byla služba násilně uvolněna. Pokusí se o znovupřidělení služby a pokud i to selže, vygeneruje výjimku. Tato třída výrazně zjednodušuje práci s mnoha službami.

Oddělení agenta od služeb

Jak již bylo uvedeno v kapitole 2.5.4, služby by měly být od agenta odděleny. Toho se zajistí pomocí **zástupců** (*proxy*), přičemž ve SMASu se využívá standardního mechanismu z *JDK* (tento mechanismus je obdobný jako při komunikaci).

Při žádosti o přidělení služby (metoda getService) agent službu přidělí, ale nevrátí přímo objekt, který ji reprezentuje. Místo toho vytvoří zástupce, který na ni obsahuje referenci. Tento zástupce implementuje pouze interface, kterým agent žádal o službu (tj. typ služby). Přestože služba (objekt, který ji reprezentuje) má jistě i další metody mimo tento interface, ty nejsou dostupné. Objekt je tak oddělen od služby a nemůže získat neoprávněný přístup.

„Zákeřný“ agent se může pokusit získat interní informace tím, že by objekt serializoval a poslal někam, kde ho lze deserializovat (nebo přímo analyzoval serializovaný tvar). Tomu je zabráněno tím, že odkaz na samotnou službu je v zástupci označen jako transient a při serializaci se tedy neukládá.

Samotný zástupce serializovatelný je a agent si jej může s sebou nést i na jiný stroj. Vzhledem k výše uvedenému tím ale nic nezíská, protože bude zcela nefunkční.

Další významnou vlastností zástupce je, že může být **odpojen** (*disconnect*). Touto operací se zruší odkaz na službu (nastaví se na null) a zástupce je od toho okamžiku nefunkční. Použití je významné především při uvolnění služby, kdy se tato operace provede a zabrání se tak používání již uvolněného zástupce (k tomu může dojít jen omylem nebo se špatnými úmysly).

Implementace zástupce

V *JDK* verze 1.3 byl zaveden obecný mechanismus vytváření zástupců. Tradiční způsob totiž vyžadoval, aby byl zástupce napsán jako normální třída (ale všechny jeho metody pouze předaly volání dál). Takovýto přístup je však velice nepraktický a omezující.

Pro dynamické vytváření zástupce slouží třída

```
java.lang.reflect.Proxy
```

Tato třída nemá veřejný konstruktor a nelze tedy vytvořit její instanci. Instance této třídy by však stejně neměla smysl. Důležitou vlastností je, že lze *dynamicky* (za běhu) vytvářet potomky této třídy pomocí statické metody getProxyClass. Parametrem je (kromě použitého *classloaderu*) pole interface, které má tato nově deklarovaná třída implementovat.

Nově vygenerovaná třída má jediný konstruktor, který jako parametr bere *invocation handler*. Všechny ostatní metody pak jsou obslouženy speciálním způsobem: zavolá se metoda *invocation handleru* invoke a jako parametr se jí předá volaná metoda a všechny její parametry. *Invocation handler* pak tuto metodu vyvolá na objektu, který zástupce reprezentuje.

5.5.3 Individuální služby

Jak již bylo uvedeno při popisování místa v kapitole 5.3.2, místo vytváří speciální tzv. *individuální služby*, které jsou poskytnuty jen agentovi v tomto místě uloženém. Individuální služby vesměs nahrazují přímou komunikaci s místem nebo strojem.

Tyto služby se při veškeré své činnosti automaticky autorizují, aniž by se tím musel agent sám zabývat. Agent sám se autorizuje jen při přidělování služby (tím, že vůbec zná referenci na její kontext).

Jinou možností, jak by se dala výše uvedená vlastnost zajistit, by bylo poskytovat každému agentovi vlastní instanci služby (tak tomu ve skutečnosti také je). Tyto instance by se rozlišovaly pomocí *selectoru* a agent by se autorizoval *requestorem*.

Použití individuálních služeb je však jednodušší jak z hlediska agenta (nemusí používat žádný *selector* ani *requestor*), tak i z hlediska implementace stroje. Všechny informace patřící k jednomu agentovi jsou centrálně shromážděny v místě, místo aby byly roztroušeny po jednotlivých službách.

Všechny individuální služby jsou rozšířením interface IndividualService. Tento interface však slouží pouze jako označení. Standardní individuální služby, které musí místo poskytovat jsou:

AgentInfoService: Poskytuje aktuální informace o agentovi: jméno, adresu, jméno stroje, adresu stroje, jméno skladu a zda je agent v aktivním stavu.

AgentMetaDataService: Zpřístupňuje agentovi jeho metadata.

CommService: Komunikační služba umožňuje jednak navázat spojení s jakýmkoli strojem nebo agentem (pokud je známa adresa) a jednak umožňuje vystavit **komunikační rozhraní** (*communication interface*). Podrobněji je komunikace popsána ve vlastní kapitole 2.7.

OperationsService: Služba umožňuje agentovi operace se sebou samým, především pak migraci (metoda migrate) a ukončení (metoda terminate).

ProxyService: Pomocí této služby může agent vytvářet zástupce pro jeho objekty, které například předává jinému agentovi.

ThreadAssignmentService: Služba pro přidělování vláken. Pokud agent potřebuje nové vlákno, požádá o jeho vytvoření pomocí této služby.

5.5.4 Vytváření a implementace služeb

Z hlediska agenta sice služby vytváří, přiděluje a uvolňuje *kontext*, ten však ve skutečnosti funguje jen jako prostředník. Vytváření služeb, samotné přidělení a uvolnění provádí tzv. *service provider*. Pro každý typ služby (serviceClass) je zaregistrován právě jeden *service provider*, který tyto služby zajišťuje.

Service provider musí implementovat interface uvedený na výpisu 5.6.

```
package java.beans.beancontext;

public interface BeanContextServiceProvider {
    Object getService(BeanContextServices bcs, Object requestor, Class serviceClass,
                      Object serviceSelector);
    public void releaseService(BeanContextServices bcs, Object requestor,
                               Object service);
    Iterator getCurrentServiceSelectors(BeanContextServices bcs, Class serviceClass);
}
```

Výpis 5.6: Interface třídy BeanContextServiceProvider

Provider se v kontextu zaregistruje pomocí metody addService, viz výpis 5.4 strana 92. Od té chvíle je v kontextu služba (služby) tohoto typu dostupná a všem *listenerům* se o tom pošle zpráva.

Pokud agent zažádá o službu (metoda getService), kontext najde odpovídajícího *providera* a zavolá jeho metodu getService. Význam parametrů je stejný jako u stejnojmenné metody volané agentem, návratovou hodnotou je přidělená služba. Zástupce je vytvořen až samotným kontextem. Uvolnění probíhá naprosto stejným způsobem, tj. zavolá se releaseService *providera*.

Třetí metoda, kterou musí *provider* implementovat, vrací seznam všech použitelných *selektorů*.

Pokud již služby tohoto typu nemají být k dispozici, *provider* se odregistruje z kontextu pomocí metody revokeService (viz výpis 5.4 na straně 92). Kontext ihned všem *listenerům* oznámí, že služba již není dostupná.

5.6 Autentifikace a autorizace

Veškerou autentifikaci a autorizaci lze rozdělit na dvě části:

1. autorizace agenta – zabezpečení stroje před zákeřnými agenty,
2. zabezpečení komunikace proti aktivnímu napadení.

5.6.1 Identita vlastníka a tvůrce

Pro autorizaci akcí agenta (především jeho přístupu ke službám) se jen výjimečně použije identita jeho samého (jediný použitelný případ je ten, kdy jsou mu práva omezena pro „nevhodné chování“). Téměř výhradně se používají identity **vlastníka** (*owner*) a **tvůrce** (*creator*, většinou klient) uložené v *metadatech*.

Pro uložení této identity se používá třída z balíku *JAAS* (*Java Authentification and Authorization Service*):

javax.security.auth.Subject

Použití je však mírně odlišné od standardního. Instance této třídy by měla být vytvořena jako výsledek autentifikace, zde tomu tak být nemůže (stroj ji již přijme vytvořenou).

Třída Subject obsahuje dva typy údajů: *principals* (obsahující jména) a *credentials* (překlad neexistuje, možná „průkazy“).

Principals: *Principal* je objekt, který obsahuje jméno. Protože entita může vystupovat v mnoha rolích, mívá několik jmen v různých jmenných prostorech. *JAAS* formu a význam těchto jmen nijak nedefinuje.

Stroj však potřebuje znát význam těchto jmen, aby podle nich mohl agenta autorizovat. S tím souvisí i další, velice závažný problém: metadata jsou přenášena na různé stroje a tyto stroje musí mít k dispozici programový kód *všech* tříd v metadatech obsažených. Pokud by tedy byl použit *principal*[3], jehož třída není stroji k dispozici, stroj nemůže tyto metadata správně vytvořit a tedy nelze vůbec agenta přijmout. Teoreticky by šlo tyto třídy získat ze stejného místa jako třídy agenta, tato možnost je však nepřípustná kvůli zajištění bezpečnosti.

Z těchto důvodů je předem definováno několik typů, které musí stroj standardně rozeznávat:

- smas.auth.SimpleNamePrincipal – obsahuje prosté jméno jako řetězec. Význam tohoto jména není specifikován, ani není nijak definováno, jakým způsobem by se mělo ověřit. Prakticky zde může být jakákoli hodnota a tudíž na jeho základě by se neměla provádět žádná rozhodnutí. Může však být vhodné pro zobrazení uživateli.
- smas.auth.LoginNamePrincipal – reprezentuje login name. Toto jméno je však platné jen ve velmi omezeném jmenném prostoru a také jsou potíže s jeho ověřením. K tomuto účelu je kromě jména uložen i řetězec určující, kdo ověřování provádí.

  Ve standardní implementaci je toto jméno ignorováno, protože není znám způsob jak ho ověřit. Jiná implementace však může využívat nějakou centrální službu pro ověřování (například *Kerberos*).

[3] Totéž platí i o *credentials*.

- smas.auth.RealNamePrincipal – obsahuje reálné jméno uživatele (jméno, příjmení). Není znám žádný standardní způsob jak ho ověřit, takže z bezpečnostního hlediska je použitelné jen ve speciálních implementacích.
- smas.auth.X500Principal – reprezentuje jednoznačné X.500 jméno. Toto jméno se používá v X.509 certifikátech. Toto jméno lze jako jediné opravdu ověřit pomocí signatur a certifikátů (viz dále).

  Problém s touto třídou je ten, že interně nerozlišuje strukturu jména a porovnání provádí jako prosté porovnání řetězců (X.500 jméno však umožňuje několik ekvivalentní zápisů, lišících se například nevýznamnými mezerami na některých místech).

Credentials: Pojmem *credential* je míněna informace o entitě, kterou se tato entita může prokazovat a používat ji k autorizaci. *Credential* může být jakýkoli objekt, ale platí zde výše uvedené omezení, že kód použité třídy musí být k dispozici na jakémkoli stroji.

SMAS definuje jediný používaný typ *credentials*, který však má zásadní význam. Jsou to **certifikáty** potomci třídy

java.security.cert.Certificate

Jediný implementovaný a podporovaný druh certifikátů je X.509. Tyto certifikáty obsahují identitu svého vlastníka a jsou podepsané certifikační autoritou, takže lze ověřit jejich pravost. Protože jsou však zcela veřejné, je nutné ověřit, zda skutečně entita má oprávnění tyto certifikáty používat.

5.6.2 Zabezpečení a přenos metadat

Protože metadata obsahují důležité informace, je nutné se ujistit o jejich pravosti, aby případný útočník nemohl vysílat zákeřné agenty s cizí identitou. Bohužel, toto zabezpečení je velice citlivé na zabezpečení komunikace a při volnějších autorizačních pravidlech (např. přijímání agentů i od neznámých klientů a vlastníků) ho nelze zcela zajistit (podrobněji viz dále).

Metadata nejsou přenášena nikdy přímo, ale ve speciální formě. Ta se skládá z bloku dat (pole byte[]) obsahujících zakódovaná metadata a z pole signatur (podpisů) tohoto bloku dat. Pro zakódování slouží serializace, stroj si vždy po přijetí deserializací vytvoří pravý objekt AgentMetaData.

Výše zmíněný blok dat je podepsán libovolným množstvím signatur, u každé z nich je uveden algoritmus a certifikát. Stroj pak ověří ty, které ověřit umí (zná daný algoritmus) a ověří i patřičné certifikáty.

Důležité podpisy jsou především podpisy ověřitelné některým certifikátem patřícím vlastníkovi nebo tvůrci. Protože nikdo jiný než majitel certifikátu nemůže

vyrobit podpis (samozřejmě pokud někdo nezískal jeho privátní klíč), je tím ověřeno, že se tvůrce těmito certifikáty skutečně může prokazovat (ať už patří přímo jemu nebo uživateli). Certifikáty, k nimž nepatří žádná signatura, by měly být ignorovány (pokud neexistuje jiný způsob, jak ověřit jejich příslušnost k entitě, tedy jiný druh autentifikace).

Bohužel, celý tento způsob zabezpečení vůbec nebrání případu, kdy se zfalšují celá metadata. Pokud by totiž útočník získal kompletní metadata jiného agenta, může je použít pro svého vlastního agenta a nelze to nijak zjistit. Takovýto útok je však velmi komplikován tím, že tento „podvržený“ agent by musel mít stejný kód jako agent původní (umístění kódu je také podepsáno).

5.6.3 Autorizace agenta

Jazyk Java verze 2 obsahuje sám o sobě poměrně propracovanou bezpečnostní architekturu založenou na **právech** (*permissions*). Ta je však navržena pouze pro případ mobilního kódu (tzv. appletů) a proto autorizace probíhá jen na úrovni kódu. Doplňkový balík *JAAS* (*Java Authentification and Authorization Service*) sice tuto architekturu rozšiřuje o autorizaci na základě identity, ovšem jen na úrovni vláken.

Pro autorizaci přístupu ke službám je celá bezpečnostní architektura Javy obtížně použitelná a proto `SMAS` zavádí tzv. **omezení** (*constraints*). Obě tyto architektury se navzájem doplňují.

Práva: Architektura práv v Java 2 platformě je poměrně komplexní [Sun2001a, Sun2001e]. Programovému kódu agenta je normálně přiřazeno jen minimum práv, která potřebuje (například potřebuje některá práva na přístup k síti, aby mohl používat komunikaci). Tyto práva jsou v konfiguraci přiřazena úplně všem, tj. nejsou omezena na umístění ani na určitý podpis. Administrátor může rozhodnout o tom, že některý programový kód je zcela bezpečný a přiřadit mu více práv.

Bohužel, samotnému agentovi nelze přidělit speciální práva mimo jeho vlastní vlákna. Proto platí, že pokud používá „vypůjčené“ vlákno (je volána jeho metoda, tedy především při volání *listenerů* a při komunikaci), má pouze základní práva. Veškerou činnost vyžadující více práv musí provádět v jemu přiřazených vláknech.

Při vytvoření vlákna pro agenta se samotný běh (metoda run viz kapitola 5.4.4) provede s identitou vlastníka pomocí funkce Subject.doAs [Sun2001e]. Běh pak probíhá s právy, které jsou mechanismem *JAAS* přiřazeny na základě této identity. Konfigurace přidělování práv na základě identity je standardně obsažena v konfiguračním souboru, ale *JAAS* umožňuje použít vlastní způsob. Konfiguraci tak lze mít například v databázi a průběžně ji měnit. Bohužel, tato konfigurace musí být globální a nelze tedy mít různou politiku přidělováni práv pro různé sklady.

Omezení: Použití mechanismu práv pro řízení přidělování zdrojů a služeb by bylo velice obtížné a nepružné. Proto je využíván jiný mechanismus, zcela obecný.

Každému agentovi jsou přidělena tzv. **omezení** (*constraints*). Všechny údaje jsou identifikovány řetězcem (jméno omezení) a hodnota je také řetězec. Jak je vidět na výpisu 5.7 (tento výpis je pouze informativní), metodě je předáno jméno žádaného omezení a objekt obsahující informace o agentovi (tento objekt je interní záležitostí implementace, obsahuje mimo jiné metadata i jméno agenta). Vrácená hodnota pak určuje toto omezení.

```
package cz.zcu.smas.host.auth;
public interface Constraints {
    String getConstraintValue(String param, AgentInfo agent);
}
```

Výpis 5.7: Reprezentace omezení agenta

Omezení přiděluje sklad při vkládání agenta. Mohou být statická (implementace pak využívá standardní třídy Properties) nebo i dynamická (o konkrétní hodnotě se rozhodne až při volání metody getConstraintValue).

Jediné konkrétní omezení je předem pevně definováno

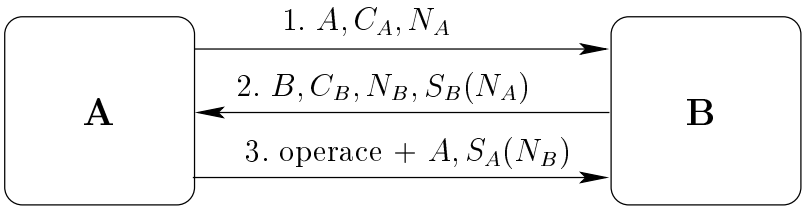
maxThreadCount
: Obsahuje maximální počet vláken, která mohou být agentovi přidělena. Minimální hodnota je vždy 1 (agent má nárok alespoň na jedno vlákno).

5.6.4 Autentifikace při komunikaci

Při komunikaci je nutné, aby si strany navzájem prokázaly svou identitu (autentifikace). Teprve podle prokázané identity pak mohou rozhodnou, zda se druhé straně dá důvěřovat (autorizace).

Pro autentifikaci se používá obdobný algoritmus jako v [Kar1998c] (optimalizovaná verze) s automatickou výměnou veřejných klíčů (certifikátů). Jedná se o algoritmus **výzva-odpověď** (*challenge-response*) pro authetifikaci při vzdáleném volání metody využívající výměnu a podepisování náhodně generovaných čísel.

Algoritmus probíhá ve třech fázích, v prvních dvou se provede základní autentifikace a ve třetí se pak již volají operace, u kterých je nutné identitu prokázat. Postup je znázorněn na obrázku 5.4. Jedna strana je aktivní a navazuje komunikaci (**A**), druhá je pasivní (**B**), komunikace je bezestavová.



Obrázek 5.4: Autentifikační protokol

Základní algoritmus

1. **A** pošle **výzvu** (*challenge*) obsahující jeho jméno (A), certifikát (C_A) a náhodně vygenerované číslo (tzv. *nonce*) N_A. Tato čísla si uchová spolu s informací, komu jej poslal.

2. **B** odpoví **odpovědí** (*reply*, *response*), která obsahuje signaturu (podpis) obdrženého čísla $S_B(N_A)$. Tento podpis zaručuje, že **B** je skutečně **B** a nikdo se za něj nevydává. Odpověď je platná pouze jednou, nelze tedy zaútočit opakováním zprávy.

   Odpověď zároveň slouží jako *výzva* pro **A**. Po přijetí této odpovědi již obě strany mají dostatek informací o té druhé a mohou se přímo provádět autentifikované operace.

3. Když chce **A** provést některou operaci, musí si vyrobit **lístek** (*ticket*). Tento *ticket* obsahuje dostatečné informace, aby se jím prokázala totožnost **A**: jméno (A) a signaturu čísla obdrženého od **B** ($S_A(N_B)$). **B** ví, jaké číslo to je a zná certifikát **A**, takže tento podpis jednoduše ověří.

   Důležité je, že *ticket* je platný pouze jednou. Jakmile je přijat, ověřující strana přestane považovat N_B za platné. Útok pomocí odposlechnutí *ticketu* tedy nemůže uspět.

   Zabezpečení je vhodné ještě doplnit tím, že číslo přestane být platné i při přijetí nesprávného *ticketu* a také tím, že na použití *ticketu* je vyhrazena určitá doba.

Ověření certifikátu

Po přijetí certifikátu je velice důležité, aby se tento certifikát ověřil:

- Pokud certifikát není znám, je nutné ověřit, zda je pravý. Pro tento účel je podepsán certifikační autoritou a při znalosti veřejného klíče je možné podpis ověřit. Stačí tedy jen mít k dispozici certifikáty těchto autorit a není nutné distribuovat veřejné klíče.

V některých případech je možné, že certifikační autorita má sama certifikát podepsaný jinou, nadřízenou autoritou. Vzniká tak celý řetězec, který končí u **kořenové** (*root*) certifikační autority. Protože ověřující strana může znát jen tuto, posílá se s certifikátem i celý jeho řetězec podepisujících certifikátů.

Certifikát může mít omezenou dobu platnosti a také může nastat situace, kdy je explicitně prohlášen za neplatný (například když dojde k úniku privátního klíče).

Postup, jak ověřovat certifikáty, je u každého typu definován a standardizován. Pro X.509 certifikáty je postup ověřování standardizován v [Hou1999].

- I když je certifikát platný, ještě je nutné ověřit zda entita udává správně své jméno. Musí se zabránit napadení, kdy by útočník **C** poslal **B** výzvu pod jménem A, ale se svým vlastním certifikátem C_C. **B** tento certifikát ověří a uzná za platný, *musí* však poznat, že ve skutečnosti patří **C**, který se ho pokoušel oklamat.

  X.509 certifikáty (pravděpodobně i jiné druhy, tento je však jediný v současné době používaný) obsahuje jméno svého majitele. Základní řešení výše uvedeného problému je takové, že jméno uváděné při autentifikaci je shodné se jménem uvedeným v certifikátu (v tom případě neexistují pochyby). Toto řešení je velice vhodné pokud se klient autentifikuje jako vlastník (při manipulaci s agentem).

  Takové řešení je však nepraktické, pokud spolu komunikují dvě součásti systému, které mají jednoznačné jméno, typicky stroje. Problém je však ten, že toto jméno má zcela jinou formu, než jaká je vyžadována jako jméno používané v certifikátu – v případě X.509 certifikátu je to X.500 jméno. Pokud tedy stroj použije své pravé jméno, musí druhá strana vědět, pod jakým jménem mu byl vydán certifikát. Nejjednodušší realizace může používat například konfigurační soubor nebo databázi. Poměrně efektivní řešení lze použít, pokud je administrátor systému zároveň certifikační autoritou: X.500 jméno může jako jeden element obsahovat `SMAS` jméno svého vlastníka.

Optimalizace

Protokol ve své základní podobě má jednu velkou nevýhodu: výměna autentifikačních informací je velmi časově náročná jednak kvůli přenosu dat po síti a jednak kvůli častým vytvářením signatur. Použití asymetrických šifer je velice pomalé.

Řešení je takové, že po jedné autentifikaci (výzva-odpověď) lze vyrobit několik *ticketů*. Je však nutné zachovat pravidlo, že každé číslo se použije jen jednou. Zcela postačující však je, když se budou čísla postupně zvyšovat o 1.

Ve výzvě a odpovědi se kromě vygenerovaného čísla N_A, N_B pošle také rozsah použitelných čísel R_A, R_B. Pro vytváření *ticketů* se pak používají postupně čísla

$$N_B, N_B + 1, \ldots \quad N_B + R_B - 1$$

Zůstává tak platit pravidlo, že každý *ticket* je jen na jedno použití. I když útočník při odposlouchávání může předvídat, jaké bude další číslo, stále nemůže vyrobit signaturu. Jednou ze základních podmínek pro digitální podpis je, že i při znalosti vztahu mezi podepisovanými daty (nebo dokonce při znalosti těchto dat) nelze ze signatur zjistit privátní klíč.

Implementace

Implementace je navržena tak, že nevyžaduje vůbec jazyk Java a lze ji použít i například při komunikaci v prostředí *CORBA*. Zprávy jsou sice třídy, ty však obsahují pouze jednoduchá data (řetězce, čísla, pole bytů). Jediné implementované metody slouží pro přístup k uloženým datům. Třídy ve výpisu 5.8 tak vlastně odpovídají pouhým strukturám v jazyce C.

Certifikát se přenáší v zakódované formě jako pole bytů. Pro certifikáty typu X.509 (které jsou jediné podporované) je tato forma definována v [Hou1999].

Používaná čísla jsou 64-bitová se znaménkem. Při generování je nutné vzít v úvahu, že při zvyšování čísla o 1 může dojít k přetečení.

Vzhledem k různým formám reprezentace čísla je nutné při podpisu používat speciální formát. Ten je definován jako výpis tohoto čísla v desítkové soustavě a ve znakové sadě *ASCII*. V Javě se tato reprezentace vytvoří takto:

```
byte [] signForm = Long.toString(n).getBytes("US-ASCII");
```

Společně s certifikátem se přenáší i jméno jeho typu a řetěz certifikačních autorit. Ten však může být prázdný.

5.7 Komunikace

Komunikační systém je oddělenou programovou jednotkou, protože může být součástí jak stoje, tak klienta (případně ještě i dalších programů). Komunikace je používána nejen ke komunikaci se samotnými agenty, ale především pro jejich přenos.

5.7.1 Forma a protokoly

Pro veškerou síťovou komunikaci mezi prvky systému je používáno **vzdálené volání metod** (*Remote Method Invocation – RMI*). Jedná se o synchronní komunikaci mezi dvěma stranami, přičemž identita volajícího je většinou neznámá. Jiné formy komunikace (např. asynchronní) lze však pomocí vzdáleného volání implementovat.

```
package smas.auth;

public class CertificateChain implements Serializable {
    private String type;
    private byte[] certificate ;
    private byte [] [] signingCertificates ;
}

public class AuthentificationChallenge implements Serializable {
    private CertificateChain cert ;
    private String name;
    private long nonce;
    private long nonceRange;
}

public class AuthentificationReply extends AuthentificationChallenge {
    private long replyN;
    private String replyNSignatureAlgorithm;
    private byte[] replyNSignature;
}

public class Ticket implements Serializable {
    private String signatureAlgorithm ;
    private byte[] nonceSignature;
    private String name;
}
```

Výpis 5.8: Zprávy zasílané při autentifikaci

Největší výhody této formy komunikace je její standardizace (protokoly jsou široce používány) a především jednoduchost použití. Samotný přenos přes síť je do velké míry transparentní a chová se jako lokální vyvolání metody. V komunikačním subsystému SMASu je tato transparentnost ještě zesílena do té míry, že za určitých okolností může komunikace dokonce probíhat lokálně (viz kapitola 5.7.5).

Důležitým požadavkem zohledněným při návrhu komunikace je možnost použití různých komunikačních protokolů. Jazyk Java obsahuje komunikační rozhraní *Java RMI* (nebo jen *RMI*), které je velice efektivní a jednoduché na použití. Je však svázáno pouze s tímto jediným programovacím jazykem a z tohoto důvodu se častěji používá systém *CORBA* (jeho použití je však výrazně komplikovanější).

Definování dalších komunikačních protokolů je dobrovolné, jediný povinný protokol je právě *RMI*. Přidat komunikační subsystém například pro systém *CORBA* by neměl být závažný problém. Výjimku ze všeho výše uvedeného tvoří jmenné servery, se kterými se komunikuje zcela odlišným způsobem. Více bude uvedeno ve zvláštní podkapitole.

```
package smas.comm;

public interface Host {
    Reference getAddress() throws Exception;
    Name getName() throws Exception;

    String [] getRepositories () throws Exception;
    Reference [] getAllAgents () throws Exception;
    Name[] getAllAgentNames() throws Exception;
    Reference getAgentAddress(Name agentName) throws Exception;

    AuthentificationReply authentificate ( AuthentificationChallenge chlg)
        throws AuthentificationFailedException , Exception;

    Reference dispatchAgent(String repository , String clientNamePart,
        byte [] agentState, AgentMetaDataTransport metaData, Ticket tck)
        throws InvalidTransferRequestException ,
            FailedInsertingAgentException , Exception;
    void migrateAgent(String repository , Name name, Reference address,
        byte [] agentState, AgentMetaDataTransport metaData, Ticket tck)
        throws InvalidTransferRequestException ,
            FailedInsertingAgentException , Exception;
    void terminateAgent(Name agent, Ticket tck) throws Exception;
}

public interface Agent {
    int STATE_LIVE = 1;
    int STATE_MIGRATING = 2;
    int STATE_PASSIVATED = 3;
    int STATE_DEAD = 4;
    int STATE_ABORTED = 5;

    boolean isCommunicationLocal() throws Exception;

    Reference getAddress() throws Exception;
    Name getName() throws Exception;
    Reference getCurrentEngineAddr() throws Exception;
    Name getCurrentEngineName() throws Exception;
    String getCurrentRepository () throws Exception;

    int getCurrentState () throws Exception;

    Object getInterface ( Class interfaceClass , String selector ) throws Exception,
        UnsupportedRemoteProtocolException, TargetNotReadyException;
}
```

Výpis 5.9: Komunikační brány stroje a agenta

5.7.2 Brány a lokace

Pro adresování uzlů a navazování komunikace je vhodné použít *JNDI* (obecný systém pro práci s adresářovými službami (viz [Sun2001f]). Výsledný mechanismus je jednoduchý a snadno rozšiřitelný standardními prostředky.

Jak již bylo řečeno, pro komunikaci se používá vzdálené volání metod. Tato metoda pracuje na úrovni objektů a požadovanou komunikační entitou je agent nebo stroj (popř. i klient, ale komunikace s klientem není nijak definována). Tato entita musí být reprezentována vzdáleně přístupným objektem, který se nazývá **brána** (*gate*).

Bran může být několik, ale jejich chování musí být zcela shodné. Liší se pouze komunikačním protokolem (pokud je jediným podporovaným protokolem *RMI*, existuje jen jedna brána). Brány pro stroj a agenta jsou na výpisu 5.9 (interface brány stroje se nazývá Host).

Každá brána je adresována pomocí *JNDI URL*. *JNDI* musí použitý protokol podporovat (standardně podporuje jak *RMI*, tak systém *CORBA*). K těmto *URL* se *JNDI* chová tak, jako by na nich byly uloženy přímo tyto vzdálené objekty (brány). Pro navázání komunikace tak stačí pouze provést:

```
Object remoteGate = new InitialContext().lookup(url);
```

Jednotlivé *URL* se uloží do objektu typu javax.naming.Reference (ten sice může obsahovat i jiné druhy adres než URL, ve SMASu však nejsou využívány). Tyto objekty mají v *JNDI* speciální význam (ten je využit pro ukládání záznamů do databáze jmenných serverů). Zpracování probíhá tak, že se postupně (v pořadí, v jakém byla vkládána) zkouší všechna v něm uložená *URL* a pokud se u některého z nich podaří vyvolat metodu lookup, získaná hodnota se vrátí.

Tento objekt je pojmenován **lokace** (*location*), aby nedošlo k záměně za pojem **adresa**. Lokace samotná je zcela dostačující k navázání komunikace se statickým programem, například strojem. Komunikace je automaticky navázána s první nalezenou branou, jejíž protokol podporují obě strany.

Pro adresování mobilních agentů však *lokace* použít nelze, protože *URL* v ní obsažené se vždy vztahují na jeden konkrétní uzel (dokonce na jeden konkrétní stroj). Pro jejich nalezení je nutné použít jmenné servery.

5.7.3 Jmenné servery

Jako jmenný server lze použít jakoukoli adresářovou službu (*directory service*), která je podporována *JNDI* a do které lze ukládat reference. Ve standardní distribuci *JNDI* je jedinou takovou službou *LDAP*, která je také použita. Veškerá práce se jmenným serverem však probíhá prostřednictvím *JNDI*, takže není problém implementovat vlastní *JNDI provider* [Sun2001f] a jako samotný jmenný server použít cokoli jiného (relační databázi, vlastní program, atd.).

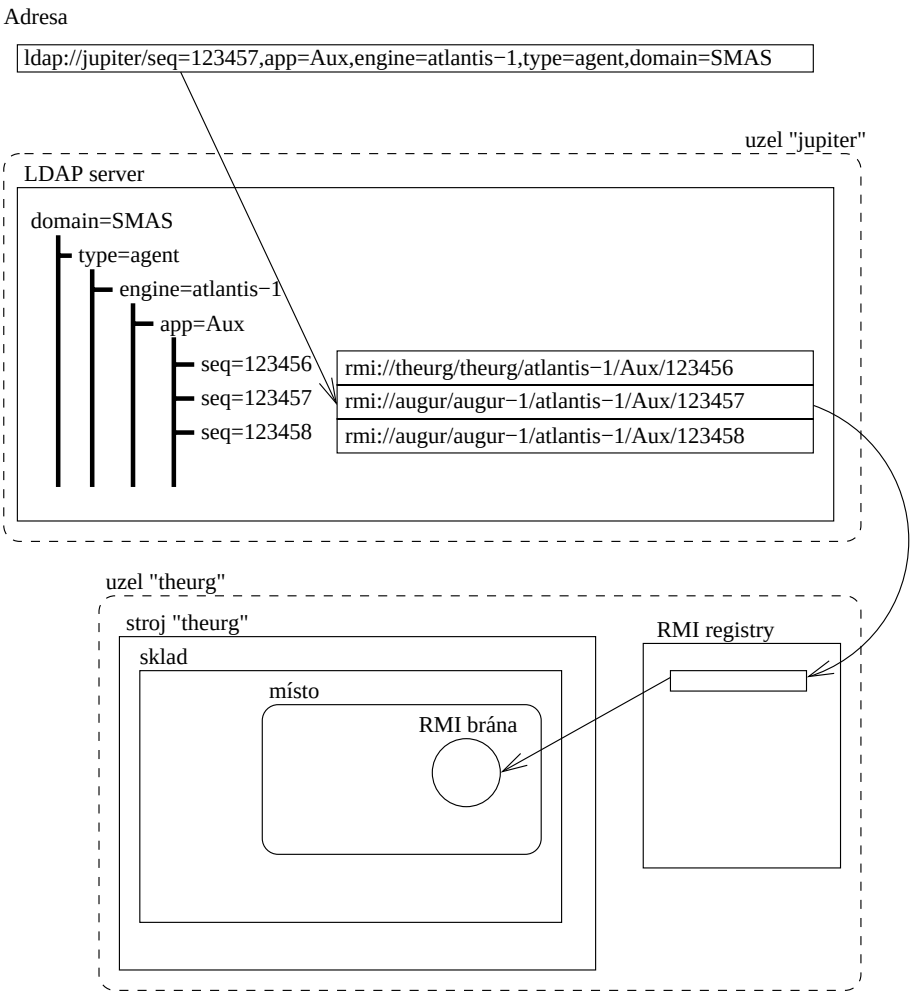
K přístupu do jmenného serveru opět slouží *JNDI URL* obsahující jméno

protokolu, adresu (DNS jméno) uzlu a plnou cestu k patřičnému záznamu (LDAP kompletní jméno). Celou komunikaci zajišťuje *JNDI*.

Na tomto místě je pak uložena *lokace*, tedy instance třídy Reference. *JNDI* však s tímto objektem pracuje speciálním způsobem: slouží totiž k tomu, aby se do adresáře uložil *místo* pravého objektu. Pokud se provede metoda lookup, nevrátí se přímo *lokace*, ale provede se její zpracování a vrátí se výsledek tohoto zpracování, tj. již navázané spojení.

Při použití jednoho jmenného serveru by byla komunikace příliš závislá na jeho stabilitě, proto je vhodné použít jmenných serverů více. Všechna *URL* na tyto servery se pak složí do jednoho objektu třídy Reference. Zpracování probíhá opět standardním způsobem.

Tento objekt se nazývá **kompletní adresa** nebo jen **adresa**. Její složky (*URL*) jsou pak **jednoduché adresy**.



Obrázek 5.5: Nalezení agenta pomocí jmenného serveru

Obrázek 5.5 ukazuje postup při navázání spojení s agentem pomocí *RMI*. Případ je poněkud zjednodušen tím, že je použit pouze jeden jmenný server a jedna brána. Adresa se tak skládá jen z jednoho *URL*. V *LDAP* je na této adrese uložena *lokace*, která se také skládá z jednoho *URL*. Toto URL ukazuje do *RMI registry* (jmenná služba *RMI*). *RMI registry* pak už přímo slouží pro navázání spojení.

Vytváření adres a manipulace se záznamy

Adresy se vytváří při vzniku dané entity, tedy při startu stroje a vyslání agenta. Jednotlivé jednoduché adresy mohou mít jakoukoli podobu závislou na druhu jmenného serveru, ale tato adresa musí být jedinečná.

Jedna možnost jak jedinečnost zajistit je nechat si tyto adresy generovat náhodně nebo sekvenčně a jedinečnost si nechat zajistit jmenným serverem. Mnohem praktičtější je však vytvářet adresu přesně definovaným postupem ze jména entity. S výhodou lze využít toho, že jmenný server je adresářová služba a že jméno stroje i agenta je reprezentováno *JNDI* jménem. `SMAS` definuje standardní rozhraní, pomocí kterého lze obecně zajistit použití jakéhokoli jmenného serveru.

Syntaxi a sémantiku používanou jmenným serverem reprezentuje interface NamingSemantics (viz výpis 5.10). Pomocí něj se jméno agenta a jméno stroje překonvertuje do jména (hierarchického) použitelného v dané službě. Tato přeložená jména se používají společně, takže by měla obsahovat informaci o typu entity. Pro LDAP tento překlad vypadá takto:

- **stroj** – stroj má jednoduché jméno, u kterého se nerozlišuje vnitřní struktura. Toto jméno se do *LDAP* přeloží velice jednoduše:

  `engine=`<*jméno*>`,type=engine`

- **agent** – u agenta je situace komplikovanější, protože má hierarchickou vnitřní strukturu (viz kapitola 5.2.1). Tuto strukturu je vhodné ponechat, protože se pak zjednoduší případná orientace ve větším množství jmen na jmenném serveru.

  `seq=`<*sekvenční číslo*>`,app=`<*aplikace*>`,engine=`<*stroj přidělující jméno*>`,type=agent`

```
package smas.naming;

public interface NamingSemantic {
    Name translateAgentName(Name agentName) throws NamingException;
    Name translateEngineName(Name engineName) throws NamingException;
}
```

Výpis 5.10: Interface třídy smas.naming.NamingSemantic

Tato forma překladu jmen pro adresu je použita na obrázku 5.5. Přeložené jméno lze pak použít v adresáři (*JNDI* používá termín *context*, který by zde však mohl způsobit zmatení pojmů), který je na jmenném serveru pro tento účel vyhrazen. Tento adresář se nazývá **kořenový** (*root*) a všechna přeložená jména jsou relativní k němu. Složení kořenového adresáře (identifikovaného plným *URL*) a přeloženého jména entity je zajištěno standardními prostředky *JNDI*.

Manipulace se záznamy ve jmenném serveru je triviální: stačí jen použít metody bind, rebind a unbind pro umístění, změnu a odstranění záznamu. Jako objekt se použije přímo **lokace**. Vše ostatní již zajistí *JNDI*.

5.7.4 Zástupci

Pro udržení komunikace při migraci agenta je používán mechanismus **lokálních zástupců** (*local proxy*) viz kapitola 2.7.4. Používá se při veškeré komunikaci, nejen při komunikaci s agentem (zástupce umožňuje ošetřit jakoukoli dočasnou nedostupnost druhé strany, nejenom migraci agenta).

Komunikační zástupci jsou implementováni pomocí standardního mechanismu *JDK*, který používá třídu java.lang.reflect.Proxy. Pomocí této třídy lze dynamicky (za běhu) vytvářet zástupce pro jakýkoli interface (i jejich kombinaci). Bližší informace lze nalézt v [Sun2001a].

Komunikační zástupce může být ve dvou stavech: **připojený** (*connected*) a **rozpojený** (*disconnected*). Pokud je *připojený*, obsahuje přímo referenci na cílový objekt. V tomto stavu je zcela transparentní a je možné normálně komunikovat. Pokud je *rozpojený*, není komunikace navázána.

Dokud je zástupce *připojený* a nenastane žádná chyba, chová se zcela transparentně – všechna volání prostě předá cílovému objektu. Pokud je při tomto volání vygenerována výjimka, otestuje se zda tato výjimka indikuje chybu komunikace (například jestli se jedná o potomka třídy RemoteException). Pokud ano, je zřejmé, že komunikace selhala a vzdálený objekt je nedostupný. Pokusí se tedy spojení obnovit.

Pro účely obnovení spojení si zástupce uchovává plnou adresu. Při pokusu o obnovení spojení pak zavolá metodu resolve (viz výpis 5.11). Pokud je tato metoda úspěšně dokončena, spojení se pokládá za obnovené a normálně se opět provede volání. Pokud se obnovení spojení nepodaří (nebo další volání opět selže), zástupce se dostane do stavu *rozpojený* a vygeneruje se výjimka

TargetDisconnectedException

Pokud je zástupce *rozpojený* a provede se vyvolání některé metody, pokusí se navázat spojení výše uvedeným postupem. Pokud se to podaří, přepne se do stavu *spojený* a volání proběhne. Pokud se navázání spojení nezdaří, vygeneruje se výše uvedená výjimka.

```
package cz.zcu.smas.naming;

public class CommResolver implements NamingResolver {
    public Object resolve(Reference addr, Class desiredClass) throws NamingException;
    public Agent  resolveAgent(Reference addr) throws NamingException;
    public Host   resolveHost(Reference addr) throws NamingException;
    public Object getRemoteProxy(Reference addr, Class desiredClass,
                        boolean connectionRequired, ClassLoader ldr) throws Exception;
}
```

Výpis 5.11: Třída CommResolver

Tímto způsobem je zajištěno udržení spojení při migraci agenta. Jakmile agent „odmigruje“ na jiný stroj, další volání selže (brána již neexistuje). Zástupce se však pokusí obnovit spojení, a protože na jmenném serveru je již uložena nová *lokace* agenta, naváže se spojení s novou branou. Volání metody se potom úspěšně provede na této bráně.

Vytvoření nového zástupce se provede vyvoláním metody getRemoteProxy třídy CommResolver (viz výpis 5.11). Parametr desiredClass určuje typ (interface) cílového objektu (brány). Zástupce může být vytvořen *rozpojený* nebo *připojený*, o tom rozhoduje parametr connectionRequired.

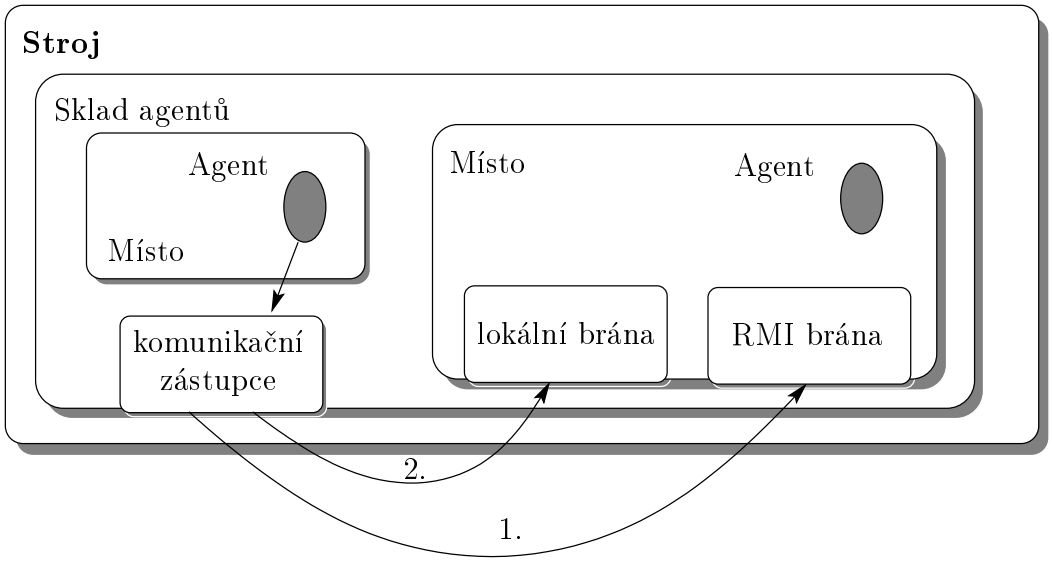
5.7.5 Lokální komunikace

Při komunikaci dvou agentů může nastat případ, kdy jsou oba na jednom stroji. Agenti by dokonce měli takovéto uspořádání preferovat, protože intenzivní komunikací mezi stroji se ztrácí jedna ze základních výhod mobilních agentů. Intenzivní komunikace mezi agenty by tedy měla probíhat lokálně v rámci jednoho stroje.

Pro lokální komunikaci mezi agenty je samozřejmě možné použít *RMI* nebo obdobné protokoly, je to však zcela zbytečné. Oba agenti jsou umístěni v jedné *JVM* a mohou spolu komunikovat přímo pomocí „normálního“ volání metod. Navíc má tento typ komunikace tu výhodu, že ji agent nemusí žádným speciálním způsobem ošetřovat (viz další podkapitola o komunikaci s agentem).

SMAS takovou komunikaci podporuje rozšířením standardního postupu při navazování spojení. Nejprve se spojení naváže normálním postupem, jak již bylo popsáno. Pokud se jedná o agenta, před návratem se zjistí jméno stroje, na kterém se právě nachází. Pokud se jedná o jiný stroj, žádná speciální akce se neprovede.

Pokud je však nalezený agent na stejném stroji jako agent, který komunikaci inicioval, zjistí se jeho jméno a nalezne se prohledáním všech *skladů*. Jako výsledek navázání spojení se vrátí **lokální brána** (*local gate*) – tuto bránu pak používá komunikační zástupce. Při volání metod se pak pouze volají metody této lokální brány a komunikace probíhá čistě na úrovni *virtual machine*. Postup navazování lokální komunikace je znázorněn na obrázku 5.6. Nejprve se naváže spojení přes *RMI* (nebo jiný obdobný protokol) a teprve po zjištění, že agent se nachází na stejném stroji, se naváže spojení přímé.



Obrázek 5.6: Lokální komunikace

5.7.6 Komunikace s agentem

Komunikační brány agenta vytváří a obsluhuje nikoli agent, ale jeho místo. Jak je patrné z výpisu 5.9, tato brána umožňuje zjistit jen různé informace o agentovi (jméno, adresu, umístění, stav).

Aby bylo možné komunikovat přímo s agentem, musí si tento agent nechat zpřístupnit svůj vlastní objekt, se kterým půjde komunikovat. Tento proces se označuje termínem **zveřejnit** (*publish*). Zveřejněný objekt je nazýván *interface* (dochází sice ke zmatení pojmů, bohužel jsem nenašel vhodnější termín) nebo *published interface*.

Metoda getInterface (viz výpis 5.9) vrátí zveřejněný *interface*, komunikace s ním probíhá stejným způsobem a stejným protokolem jako s branou, pomocí které byl získán. Pokud *interface* nepodporuje používaný protokol, vygeneruje se výjimka

UnsupportedRemoteProtocolException

Bohužel to znamená, že tento *interface* musí být přístupný pomocí tohoto protokolu. V případě *RMI* tedy musí implementovat třídu java.rmi.Remote a mít vygenerovaný *stub* (viz dokumentace [Sun2001a]). V případě jiných protokolů je situace obdobná. Je zřejmé, že se ztrácí nezávislost komunikace na použitém protokolu, navíc realizace se výrazně komplikuje.

Jiná je však situace v případě lokální komunikace. Protože se místo protokolů vzdáleného volání metody používá „normální“ (lokální) volání metody, je možné přistupovat ke každému *interface*, který agent poskytuje a není nutné splnit jakékoli speciální podmínky.

Doporučenou strategií je, aby agent většinu své komunikace prováděl lokálně. Pokud chce komunikovat s jiným agentem, může prostě odmigrovat na stroj, kde se tento agent nachází.

Identifikace a poskytování *interface*

Zveřejňování *interface* má mnoho společných rysů s nabízením služeb (kapitola 5.5). Především je pak použit téměř shodný systém identifikace (pojmenování).

Každý zveřejněný *interface* je identifikován dvěma údaji: **třída** (*class*), kterou implementuje a **selektor** (*selector)*. Stejně jako u služeb platí, že třída musí být ve skutečnosti interface a poskytovaný objekt ho musí implementovat (v opačném případě by nešlo vytvářet zástupce). *Selektor* pak slouží pro rozlišení mezi různými poskytovanými objekty stejné třídy. V tomto případě však musí být *selektor* vždy řetězec (String). Tento řetězec může být prázdný (hodnota null se musí interpretovat jako shodná s prázdným řetězcem – ne všechny protokoly musí umět pracovat s hodnotami null).

```
package smas.agent;

public interface Publisher {
    Object    getInterface ( Class  interfaceClass , String  selector );
    String []  getCurrentSelectors ( Class  interfaceClass );
}
```

Výpis 5.12: Interface třídy smas.agent.Publisher

Pokud chce agent určitý objekt zveřejnit, musí vytvořit tzv. *publisher* implementací třídy java.lang.Publisher (výpis 5.12). *Publisher* pak stačí zaregistrovat metodou publishInterface ve službě CommService. Každá poskytovaná třída může mít jen jednoho *publishera*. Když je na bráně vyvolána metoda getInterface, najde se patřičný *publisher* a na něm se zavolá metoda getInterface. Vrácený objekt je pak zpracován podle použitého protokolu, například v případě *RMI* je *exportován*. Pokud s tímto protokolem nemůže být použit, vygeneruje se výjimka.

V případě lokální komunikace se pro objekt vytvoří zástupce.

Problémy při použití *interface*

Při lokální komunikaci je sice samotný *interface* chráněn pomocí zástupce, pro jeho návratové hodnoty to však neplatí. Agent by to měl vzít v úvahu a vracet jen nově vytvořené objekty bez vazeb na jeho vnitřní části. Ve většině případů je vhodné, aby si nechal pro objekt vytvořit zástupce (k tomu slouží standardní služba ProxyService).

Dalším problém je použití nestandardních tříd. Je zcela zřejmé, že problémy nastanou pokud jeden agent druhému předá objekt, jehož třídu nemá druhý agent k dispozici. Druhý agent nemůže vyvolat metody tohoto objektu (kromě metod, které oddědědil od třídy Object).

Méně zřejmé je, že stejný problém nastane i v případě, když spolu dva stejní agenti (míněno agenti se stejným programovým kódem, tedy se shodnými třídami) komunikují lokálně a předají si navzájem instance tříd z jejich vlastního kódu. Tyto třídy jsou sice shodné, pocházejí se stejného místa, jsou však nalezeny jinými *classloadery*. Takové třídy Java považuje za odlišné. Pokusy o volání metody vygenerují výjimku.

V případě samotných *interface* tento problém řeší používaný lokální zástupce. V případě objektů předaných jako parametry (nebo vrácených jako návratová hodnota) si ošetření tohoto problému musí zajistit sám agent. Bohužel asi jedinou metodou je volání metod pomocí *Java reflection* [Sun2001b]. Tuto metodu právě používá zástupce pro lokální komunikaci.

5.7.7 Zabezpečení komunikace

Jakým způsobem je zabezpečena autentifikace je popsáno v samostatné kapitole 2.6.4. Zabezpečení komunikace proti odposlechu a změně přenášených dat není ve `SMAS`u nijak explicitně řešeno. Toto zabezpečení se realizuje pomocí šifrování komunikace, kterou lze zajistit na několika úrovních.

Veškerá komunikace probíhá pomocí protokolů pro vzdálené volání metod. Na této úrovni (aplikační) je téměř nemožné bezpečný šifrovaný přenos zajistit. Výraznou překážkou je především bezestavový charakter komunikace.

Pro používané protokoly však vesměs existují jejich zabezpečené verze. Často se používá jejich modifikace pracující nad protokolem *SSL* [Die1999], jako je *RMI over SSL* nebo *IIOP over SSL* (*CORBA*). V případě systému *CORBA* je to však spíše náhradní řešení, protože existuje tzv. *CORBA Security Service*, která bezpečný přenos (ale i například autentifikaci) definuje (není však povinná a není ani příliš rozšířená). Šifrování přenosu lze zajistit i na nižší úrovni, například pomocí bezpečných kanálů (*ssh*) nebo *VPN*.

Ze všech těchto důvodů plyne, že řešení této otázky v návrhu systému by bylo poměrně zbytečné omezení. Stávající návrh umožňuje dokonce použít různé metody, protože zabezpečené verze protokolů se mohou (často musí, neboť nejsou zcela nekompatibilní) považovat za různé protokoly. Stroj tak například může nabízet dvě brány: *RMI over SSL* a *RMI* (v tomto pořadí). Pokud klient implementuje bezpečnou verzi, použije ji, pokud ne, použije normální *RMI*.

5.8 Manipulace s agentem

V této kapitole jsou popsány všechny přechody mezi stavy agenta (životní cyklus viz obrázek 2.4 na straně 23) a tedy i všechny operace s ním. Jedinou nepopsanou operací zůstává vytvoření, které je zcela závislé na klientovi. Přechod z a do stavu *Aktivní* není uveden v samostatné podkapitole, ale pro větší názornost je popsán společně s podrobným popisem vkládání agenta.

5.8.1 Vložení a vyjmutí agenta

Vložení agenta do místa a jeho vyjmutí nejsou nikdy prováděny samostatně, ale jako součást většiny operací. Postup je však (až na některé detaily) shodný. Navíc agent musí při vkládání a odebírání správně reagovat.

Vložení agenta

Zjednodušený diagram průběhu vkládání je na obrázku 5.7. Vkládání je rozděleno do dvou fází:

1. Sklad vytvoří pro agenta místo. Místo samotné pak ve svém konstruktoru vytvoří všechny lokální služby (fáze **1.1**). V diagramu je pro jednoduchost znázorněna pouze služba pro přidělování vláken (ThreadAssignmentService). Ta, podobně jako mnohé další individuální služby, má pouze jednu instanci a tato instance je zároveň i *provider* (viz kapitola 5.5.4).

2. Sklad zavolá metodu insertAgent, která provede samotné vložení:

   (a) Vytvoří nový kontext.

   (b) Do tohoto kontextu vloží (add) hlavní objekt agenta. Kontext zavolá na tomto objekt setBeanContext (**2.2.1**).

   V místě okamžiku agent ví, že je vkládán do kontextu (a tedy do místa). Tento okamžik je však *jediný zaručený* případ, kdy je agent zavolán jinak, než prostřednictvím *listeneru* nebo komunikačního interface. Proto by si měl zaregistrovat alespoň jeden objekt jako *listener* **2.2.1.1**).

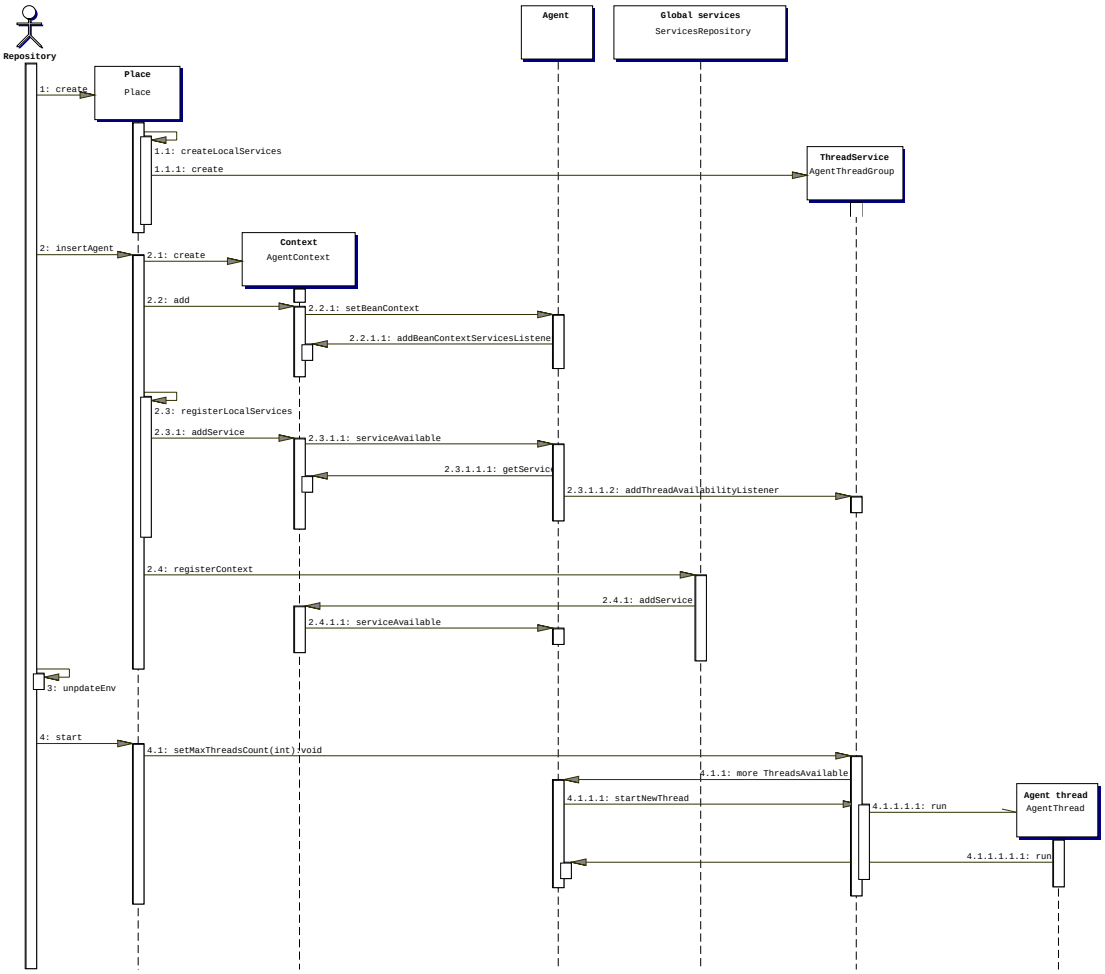
   Žádnou další akci by však agent neměl provádět, nicméně bez služeb ani nic provádět nemůže.

   (c) Všechny individuální (lokální) služby jsou zaregistrovány v kontextu. Pořadí není definováno a agent by na to měl být připraven.

   O každé zaregistrované službě je agent (resp. jeho příslušný *listener* informován (**2.3.1.1**) a může si o ni hned zažádat (**2.3.1.1.1**).

   (d) Nakonec jsou zaregistrovány i všechny globální služby, uložené v globálním skladu služeb (jakým způsobem jsou služby uloženy je implementačně závislé, takovýto způsob používám já ve své implementaci). Jinak platí totéž co pro individuální služby.

Po dokončení výše uvedeného postupu je agent vložen do kontextu, ale to ještě neznamená, že je plně funkční. Ještě je nutné provést další akce, jako je například přiřazení práv a omezení, inicializace komunikačních bran a podobě. Tyto akce se však liší podle operace a proto jsou zde znázorněny jen schematicky fází **3**. Teprve po jejím provedení je agent zcela funkční a ve stavu *Čekající*.



Obrázek 5.7: Vložení agenta

Přechod do a ze stavu *Aktivní*

Do stavu *Aktivní* lze přejít pouze ze stavu *Čekající*. Na obrázku 5.7 je tento přechod proveden ve fázi **4**. Jedinou akcí, kterou místo provede je, že zjistí kolik vláken má mít agent k dispozici (tato hodnota je v jeho **omezeních**) a tuto hodnotu nastaví ve službě pro přidělování vláken (**4.1**). Pokud si agent v této službě zaregistroval některý objekt jako *listener*, je mu poslána zpráva o zvýšení počtu možných vláken (tento počet je ve stavu *Čekající* vždy 0). Agent by měl zareagovat tak, že si okamžitě zažádá o vlákna, která ke své činnosti nutně potřebuje (**4.1.1.1**). Tyto vlákna mu jsou pak přidělena a agent již může provádět libovolnou vlastní činnost.

Zastavení agenta a přechod do stavu *Čekající* je na obrázku 5.8, fáze **1**. Nejprve se nastaví maximální počet vláken na 0, aby agentovi nemohlo být již žádné další přiděleno (**1.1**). Potom se zavolá metoda stopAllRunnable (**1.2**), která

všechny přidělené thready ukončí pomocí metody stop, jak je popsáno v kapitole 5.4.4. Agent je povinen všechna vlákna co nejdříve ukončit (tak, aby zůstal v konzistentním stavu). Pokud tuto povinnost nedodrží do určitého maximálního času, budou všechna vlákna ukončena „násilně“ – takové ukončení však většinou znamená, že data zůstanou v nekonzistentním stavu.

Vyjmutí agenta

Vyjmutí agenta probíhá opačně než jeho vložení jak je znázorněno na obrázku 5.8 fází **2**.

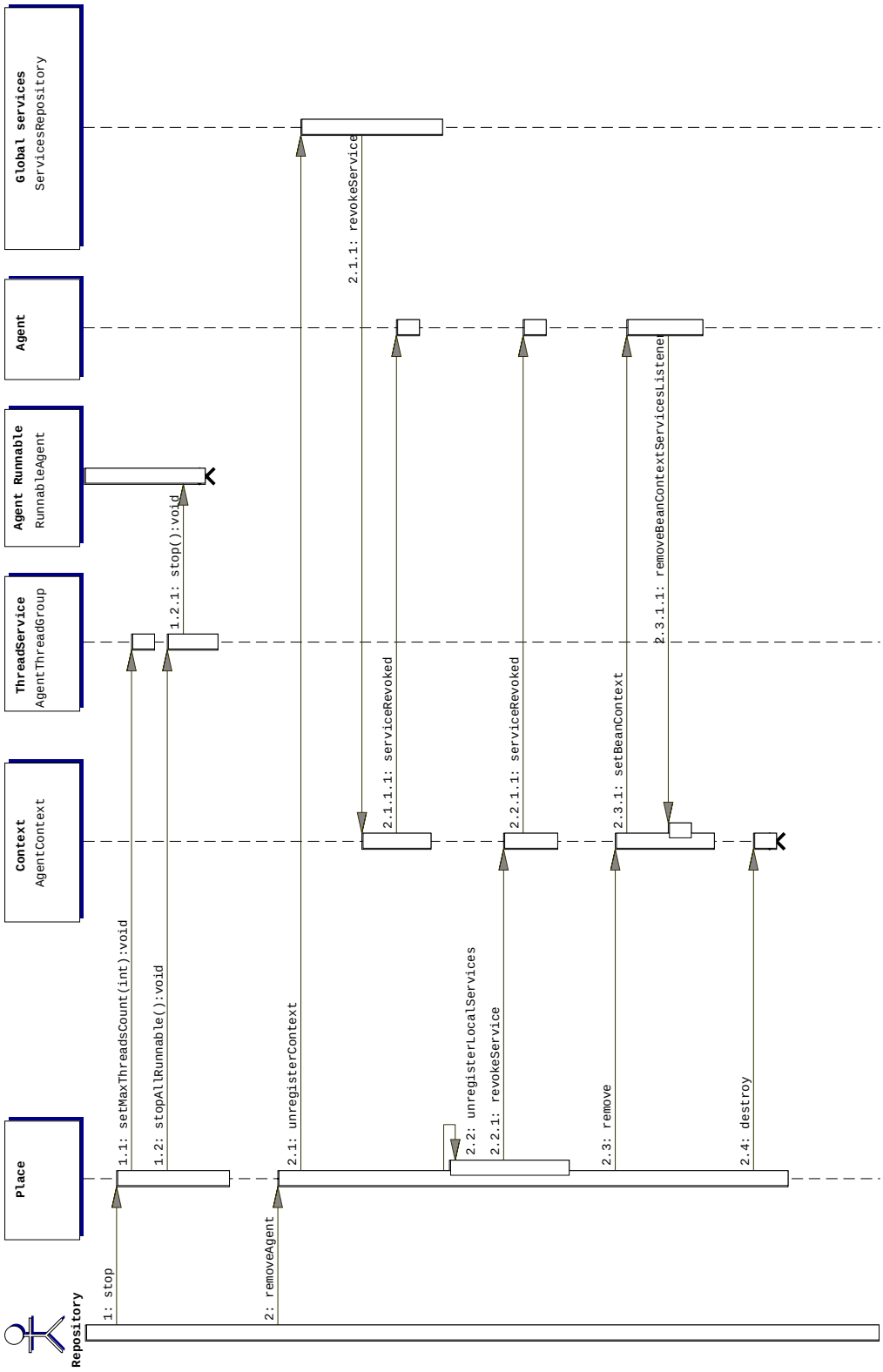
- Nejprve se odregistrují všechny globální služby v globálním skladu služeb (**2.1**). Z hlediska agenta jsou tyto služby rušeny, což mu je oznámeno (**2.1.1.1**, pro každou službu zvlášť) voláním zaregistrovaného *listeneru*. Pokud má agent přiřazenu nějakou službu, je povinen mít zaregistrovaný *listener* a na tuto zprávu zareagovat uvolněním služby.
- Jako další krok se uvolní individuální služby (**2.2**). Platí totéž co v předchozím případě.
- Po uvolnění veškerých služeb se agent odstraní z kontextu (**2.3**). Agent by měl odregistrovat všechny zaregistrované *listenery*. Od tohoto okamžiku již nemá žádnou vazbu na stroj.
- Nakonec se zruší kontext, tj. odstraní se všechny odkazy na něj a skutečně zrušen bude až *garbage collectorem* (**2.4**).

5.8.2 Vyslání

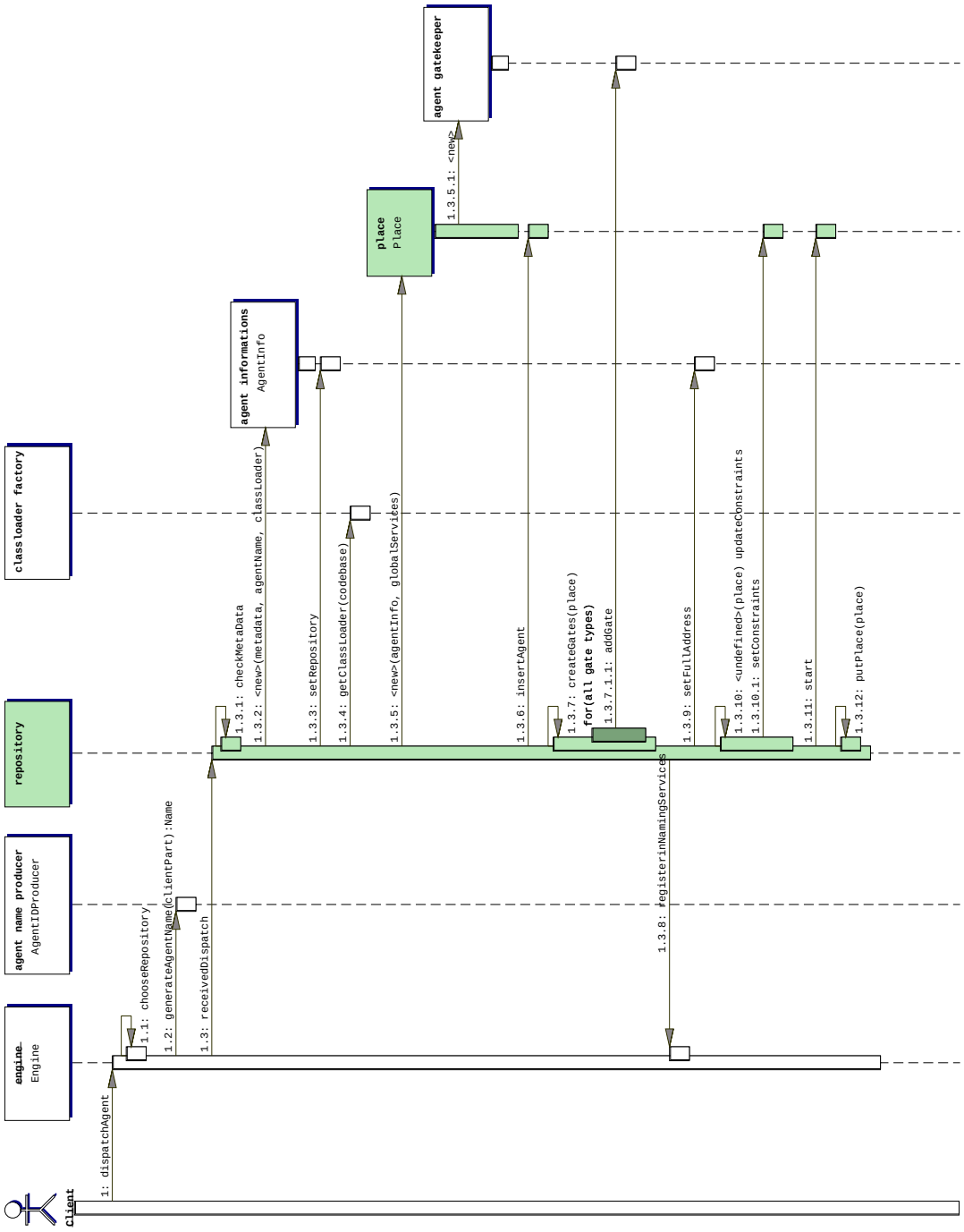
Průběh vyslání je znázorněn na obrázku 5.9. Klient musí navázat spojení se strojem, na který chce agenta vyslat, a zavolá metodu dispatchAgent (viz výpis 5.9 na straně 105).

Po ověření autorizace a ověření výběru skladu (zda tento sklad existuje) se vygeneruje jméno nově vytvořeného agenta (**1.2**). Přijatý zakódovaný stav (serializovaný agent) a metadata se předají patřičnému skladu. Ten teprve provede ověření metadat (**1.3.1**) a rozhodne, zda agenta neodmítne. Pokud ne, provede samotné vložení a oživení agenta:

- Všechna dostupná data o agentovi shromáždí (**1.3.2** a **1.3.3**), vytvoří místo a vloží do něj agenta (postup byl již popsán).
- Vytvoří všechny brány sloužící pro komunikaci s agentem. Z těchto bran (kromě lokální samozřejmě) se vytvoří **lokace** a ta se uloží do jmenných serverů (**1.3.8**). Protože je agent vytvářen, je vytvořena nová kompletní adresa, která se agentovi přiřadí (**1.3.9**) a zůstane jeho adresou až do ukončení.



Obrázek 5.8: Vyjmutí agenta



Obrázek 5.9: Vyslání agenta

- Sklad rozhodne o **omezeních** (*constraints*) agenta, vytvoří jejich reprezentaci a předá ji místu (**1.3.10**).
- Agent je spuštěn.
- Až nakonec, proběhne-li vše v pořádku, si sklad zaregistruje nově vytvořené místo. Pokud před tímto okamžikem cokoli selže, vše se vrátí do původního stavu (agent není vytvořen).

Při úspěšném provedení je klientovi vrácena kompletní adresa agenta.

5.8.3 Ukončení

Ukončení může provést buď vlastník (respektive klient, který se může prokázat identitou vlastníka) nebo klient sám. Postup je na znázorněn na obrázku 5.10:

- Agent je nejprve zastaven (převeden dos stavu *Čekající*). Pro případ, že by dobrovolně neukončil svá vlákna (ať již úmyslně nebo vinou chyby v programu), je naplánováno jejich násilné ukončení. To se provede po uběhnutí určitého časového intervalu. Tento interval by měl být dostatečný na to, aby se agent mohl sám korektně ukončit.
- Sklad odstraní místo ze svého seznamu míst. Od tohoto okamžiku již agent ve stroji formálně neexistuje.
- Agent je vyjmut z místa a místo je zrušeno.
- Nakonec jsou zrušeny všechny záznamy o agentovi ve jmenných serverech.

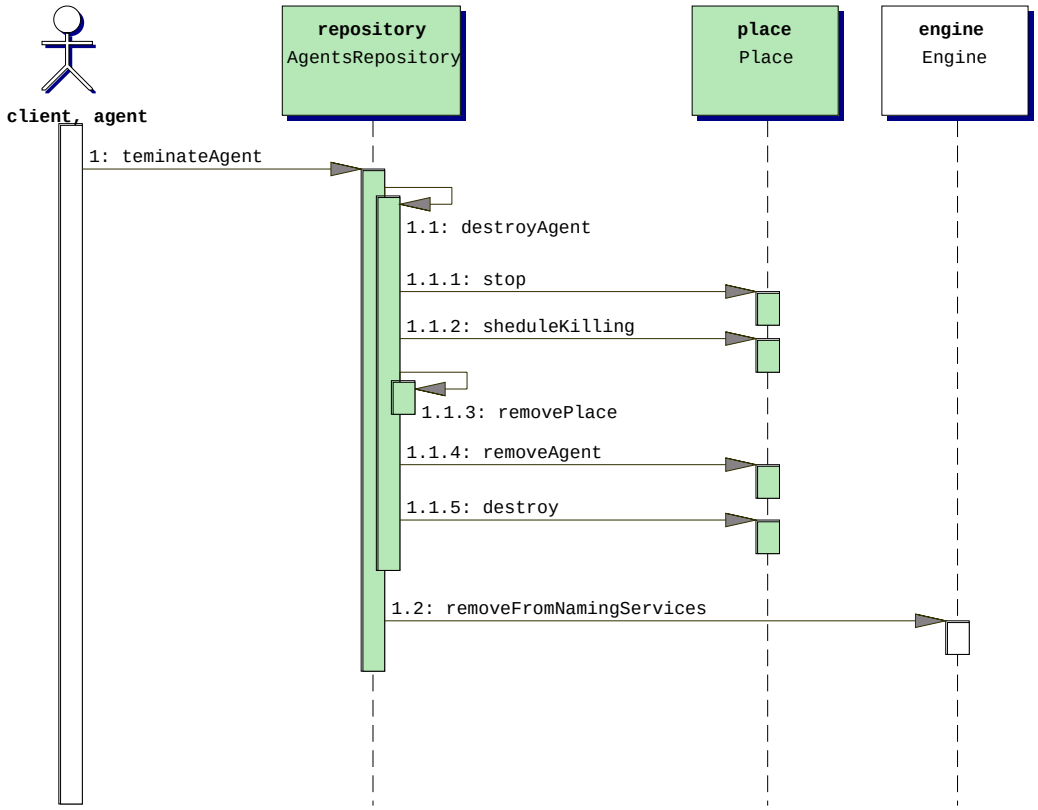
5.8.4 Migrace

Migrace je nejkomplikovanější operací s agentem vůbec. Skládá se ze dvou částí, každá z nich probíhá na jiném stroji.

Odeslání: První část migrace, odehrávající se na původním hostiteli, je znázorněna na obrázku 5.11. Migrace je vždy iniciována agentem, který zavolá metodu migrate služby OperationsService, která ji spustí (**1.1**). Důležité je, že samotný proces migrace probíhá ve vlastním vlákně. Volající vlákno tak může být korektně ukončeno.

1. Nejprve se otestuje, zda cílový stroj a sklad nejsou shodné se strojem a skladem, kde se agent již nachází. V takovém případě není nutné provádět žádné akce.

   Odlišná je situace, pokud chce agent migrovat na stejný stroj, ale do jiného skladu. Migrace pak musí normálně proběhnout (i když stroj samozřejmě může použít zjednodušený postup).



Obrázek 5.10: Ukončení agenta

2. Agent je zastaven *synchronně*, tj. počká se až agent ukončí všechna svá vlákna.
3. Je vytvořen zakódovaný stav pro migraci. Tento zakódovaný stav je prostě serializovaný agent (**1.1.3.2** a **1.1.3.3**).

   Serializaci však není vhodné provést bezprostředně, protože agent má přidělené služby a musel by si sám ošetřit jejich ztrátu. Korektnější tedy je, aby před serializací byly agentovi odebrány veškeré služby. Dojde tak k částečnému vyjmutí z kontextu.

   Teoreticky by tyto služby nebylo nutné agentovi vůbec vracet. Migrace však může kdykoli selhat a v takovém případě je nutné agentovi opět všechny služby vrátit. Stroj však nemusí agenta po celou dobu udržovat v takovémto mezistavu (částečně vložený, částečně vyjmutý) a může mu služby vrátit ihned po serializaci.
4. Zavoláním metody migrate cílového stroje je proveden přenos agenta.
5. Pokud migrace proběhla v pořádku a agent je již funkční na novém stroji, jeho stará verze se zruší. Postup je stejný jako při ukončování, pouze se neprovede odstranění záznamů ze jmenných serverů.

Přijmutí: Přijmutí migrujícího agenta je velice podobné postupu při vyslání. Rozdíly jsou následující:

- Jméno není generováno, přenáší se s agentem.
- Totéž platí pro adresu, která zůstává nezměněna. Místo vytvoření záznamů se tak ve jmenných serverech pouze umístí nová lokace.

5.8.5 Návrat

Návrat (*retract*) agenta je poměrně jednoduchý a má shodné části s migrací, takže při jeho popisu je možné se odkazovat na obr. migrace Klient, který návrat vyžaduje, se musí prokázat identitou vlastníka agenta.

Postup je následující:

1. Pokud si agent zaregistroval nějaké *listenery* pro přijímání zpráv o operacích (implementující interface OperationsListener), je jim všem oznámeno, že byl vyžádán návrat agenta (zavolá se metoda retracting).

   Tato metoda vrací hodnotu typu boolean, která znamená, zda byl požadavek akceptován. Pokud kterýkoli *listener* vrátí false, požadavek je tím zamítnut a vrátí se okamžitě prázdná hodnota.

   Klient si v požadavku určuje, zda návrat vyžaduje v každém případě (v tom případě se případné zamítnutí ignoruje) parametrem force.

2. Agent je zastaven (pokud je ve stavu **Aktivní**).

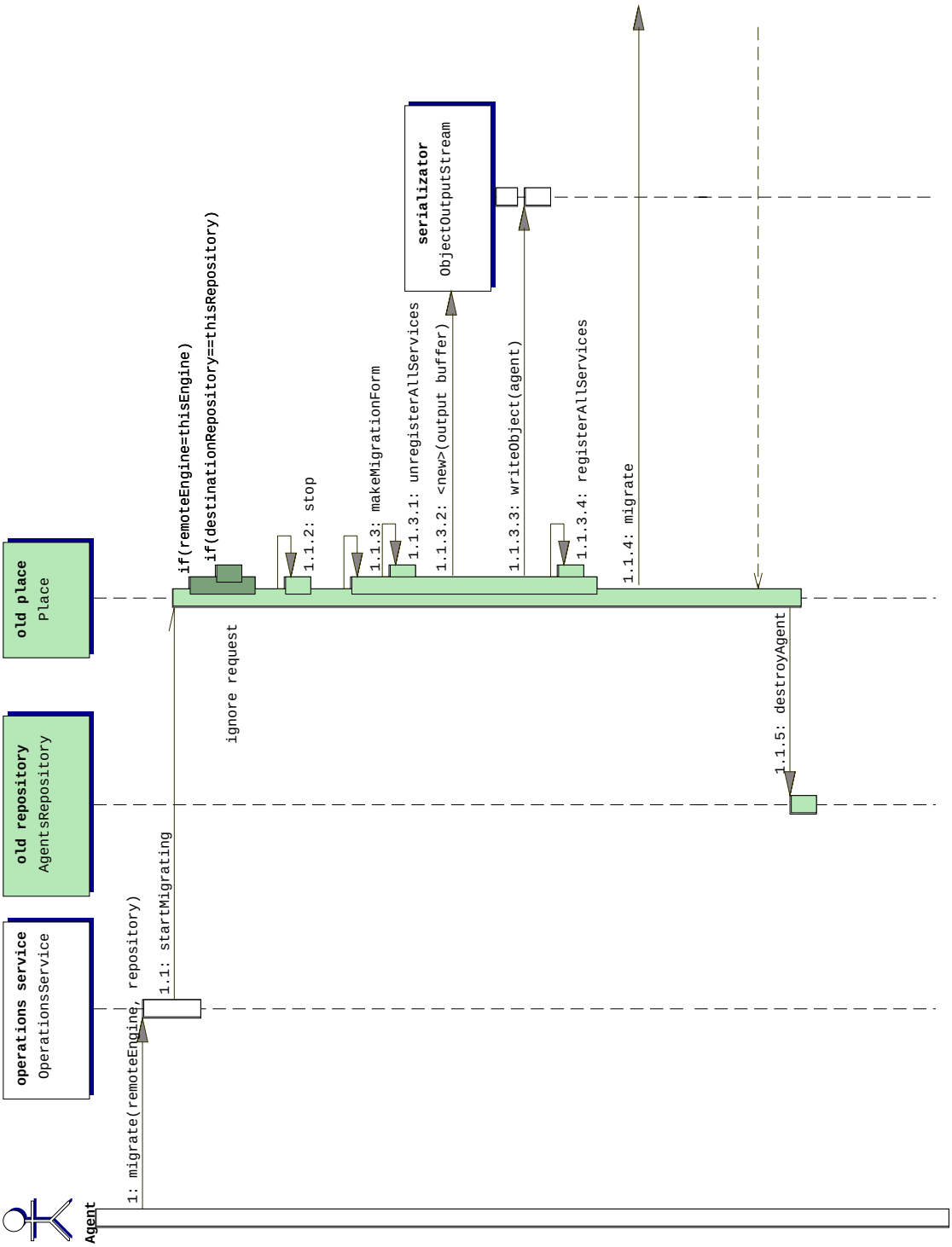
3. Je vytvořen serializovaný stav agent stejně jako při migraci viz obr. migrace fáze 1.1.3. Tato forma je pak vrácena klientovi.

4. Agent je standardním způsobem ukončen viz kapitola 5.8.3.

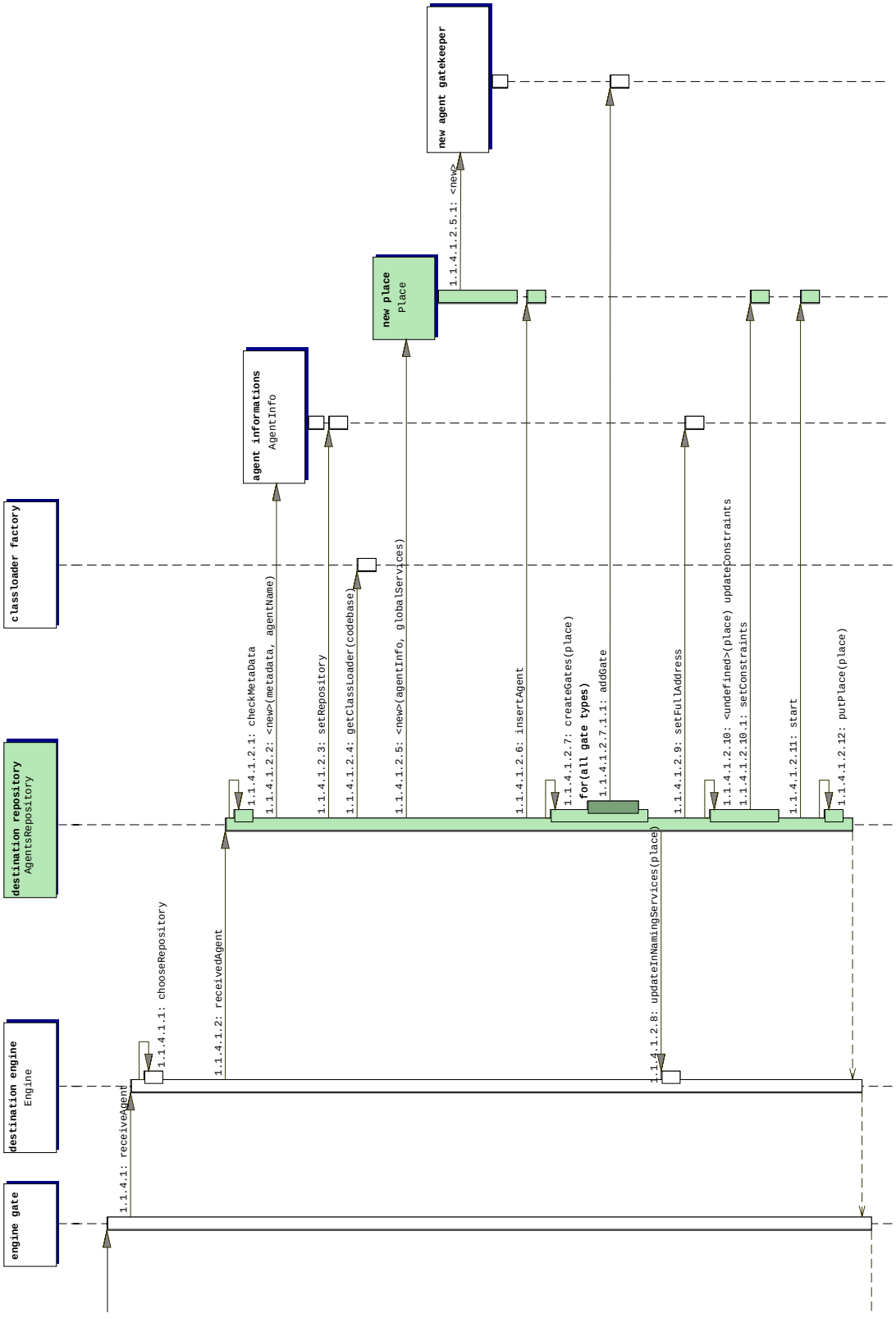
Vrácenou hodnotou je serializovaný stav agenta nebo null, pokud se agent odmítl vrátit.

5.9 Shrnutí

V této kapitole byl prezentován návrh systému mobilních agentů SMAS. Byla popsána jeho struktura i chování. Při návrhu byl kladen hlavní důraz na bezpečnost, což se odrazilo na jeho složitosti. Návrh byl realizován a v průběhu implementace pozměňován tak, jak se objevovali buď nedostatky v návrhu nebo omezení použitého jazyka Java.



Obrázek 5.11: Migrace, 1.část (odeslání)



Obrázek 5.12: Migrace, 2. část (přijmutí)

Kapitola 6

Ověření systému

Pro otestování navrženého systému bylo vytvořeno několik aplikací. V této kapitole jsou prezentovány výsledky experimentů.

6.1 Model interakce mezi entitami systému

Existuje mnoho možností, jak spolu mohou komunikovat entity (objekty, agenti) distribuovaného systému [Bau1997b]. Jako typický zástupce komunikačního modelu klient-server byl zvolen koncept RPC. Tento typ komunikace je v distribuovaných systémech široce používán. Volaná procedura je provedena na vzdáleném uzlu (serveru). Musí tam být tedy z klienta přenesena a výsledky se opět musí vrátit zpět [Tay1990]. Migrace agenta je naproti tomu proces, při němž je výpočet navázán na cílovém uzlu [Tar1996]. Tento proces zahrnuje přenos kódu, stavu výpočtu a stavu dat agenta na cílový uzel. V systémech mobilních agentů je tento proces zpravidla iniciován samotným agentem a nikoli systémem.

6.1.1 Jednoduchá interakce

Nyní bude zkoumána jednoduchá interakce v modelu klient-server probíhající za následujících podmínek:

- Cílový uzel je předem znám (nemusí se vyhledávat).
- Množství přenášených dat pro dotaz/odpověď je předem známo a je neměnné.
- Průměrné hodnoty zpoždění a průchodnosti sítě jsou známy.
- Čas potřebný pro převod dat do/z formátu vhodného pro přenos sítí roste lineárně s velikostí dat.
- Na všech uzlech je úloha prováděna stejně rychle.

Interakce pomocí RPC

V případě interakce mezi agenty může být RPC využíváno pro volání procedur (metod) poskytovaných jiným agentem. Komunikace pomocí RPC zahrnuje spojení se serverem, serializaci, přenos a deserializaci požadavku (*request*), následné provedení požadavku a serializaci, přenos a deserializaci odpovědi (*reply*). Pokud se použijí zmíněné omezující podmínky, tak může být vynechán čas potřebný pro navázání spojení se serverem, protože tento je předem znám. Také může být zanedbán čas provádění požadavku na serveru, neboť nemá vliv na interakci. Doba serializace závisí na velikosti požadavku. Výkonnostní model je tedy závislý pouze na komunikační části RPC.

Zatížení sítě (*network load*) B_{RPC} (v bytech) pro jedno volání procedury z uzlu L_1 na uzel L_2 se skládá z velikosti požadavku B_{req} a velikosti odpovědi B_{rep}

$$B_{RPC}(L_1, L_2, B_{req}, B_{rep}) = \begin{cases} 0 & proL_1 = L_2 \\ B_{req} + B_{rep} & jinak \end{cases}$$

Doba výpočtu (*execution time*) T_{RPC} pro jedno volání procedury z uzlu L_1 na uzel L_2 se skládá z doby serializace a deserializace požadavku a odpovědi (proměnná μ) a doby přenosu dat po síti s průchodností $\tau(L_1, L_2)$ a zpožděním $\delta(L_1, L_2)$

$$T_{RPC}(L_1, L_2, B_{req}, B_{rep}) = 2\delta(L_1, L_2) + \left(\frac{1}{\tau(L_1, L_2)} + 2\mu\right) \cdot B_{RPC}(L_1, L_2, B_{req}, B_{rep})$$

Interakce migrací agenta

Nyní bude interakce probíhat pomocí migrace agenta mezi dvěma uzly sítě. Proces migrace zahrnuje serializaci, přenos a deserializaci programového kódu a stavu výpočtu agenta. Za pomoci stejných omezujících podmínek může být doba výpočtu zanedbána. Doba serializace roste lineárně spolu s velikostí dat a zachyceného stavu výpočtu, zatímco kód agenta je již připraven v serializované podobě a je přenesen pouze na vyžádání (tj. v případě, že není na cílovém uzlu ještě k dispozici). Agent se skládá z kódu B_{code}, dat B_{data} a stavu výpočtu B_{state} a je popsán trojicí $B_A = (B_{code}, B_{data}, B_{state})$. Velikost požadavku na proceduru B_{req} není obsažena v datech B_{data}. Velikost odpovědi od vyvolané procedury B_{rep} je redukována na uzlu L_2 na hodnotu $(1-\sigma)B_{rep}$, kde σ je v intervalu $< 0; 1 >$ a označuje selectivitu, tedy schopnost agenta redukovat velikost přenášených dat.

Zatížení sítě při migraci agenta A z uzlu L_1 na uzel L_2

$$B_{Mig}(L_1, L_2, B_A) = \begin{cases} 0 & proL_1 = L_2 \\ P(B_{cr} + B_{code}) + B_{data} + B_{state} & jinak \end{cases}$$

kde P označuje pravděpodobnost, že kód agenta ještě není k dispozici na uzlu L_2 a B_{cr} je velikost požadavku na přenos jeho kódu z uzlu L_2 na L_1. Jestliže navíc

agent pošle odpověď zpátky na uzel L_1, zatížení sítě bude

$$B_{MR}(L_1, L_2, B_A, \sigma, B_{rep}) = B_{Mig}(L_1, L_2, B_A) + \begin{cases} 0 & proL_1 = L_2 \\ (1-\sigma)B_{rep} & jinak \end{cases}$$

kde B_{rep} je velikost odpovědi a σ označuje selektivitu agenta.

Odpovídající **doba výpočtu** obsahuje zpoždění sítě δ, průchodnost sítě τ a dobu (de)serializace μ pro jeden přenos agenta z uzlu L_1 na uzel L_2

$$T_{Mig}(L_1, L_2, B_A) = (1+2P)\cdot\delta(L_1, L_2) + \frac{B_{Mig}(L_1, L_2, B_A)}{\tau(L_1, L_2)} + \begin{cases} 0 & proL_1 = L_2 \\ 2\mu(B_{data} + B_{state}) & jinak \end{cases}$$

při přenosu odpovědi zpátky na uzel L_1 pak

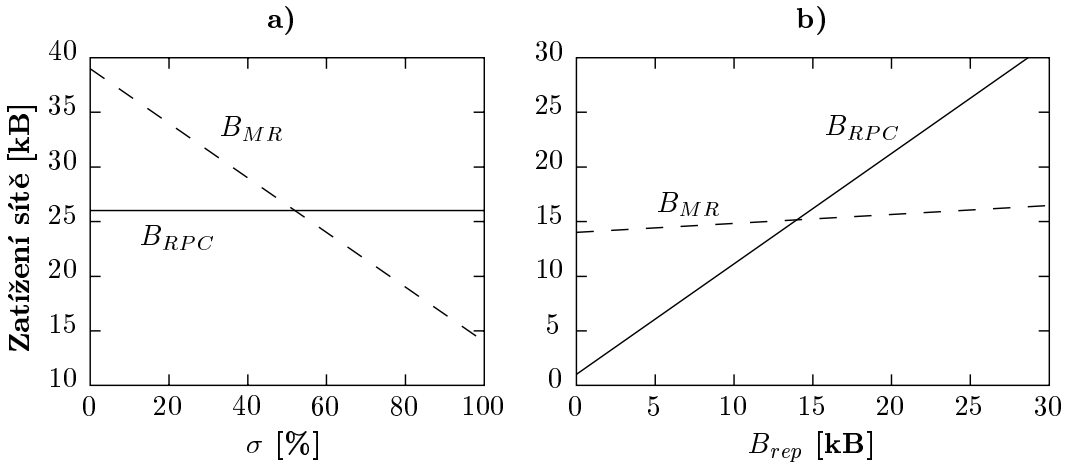
$$T_{MR}(L_1, L_2, B_A, \sigma, B_{rep}) = T_{Mig}(L_1, L_2, B_A) + \delta(L_1, L_2) + \begin{cases} 0 & proL_1 = L_2 \\ \left(\frac{1}{\tau(L_1,L_2)} + 2\mu\right)(1-\sigma)B_{rep} & jinak \end{cases}$$

Vyhodnocení jednoduché interakce

Pro srovnání předchozích modelů jednoduché interakce pomocí RPC a jednoduché interakce migrací agenta bude uvažován následující scénář: kód agenta $B_{code} = 39kB$, data $B_{data} = 5kB$ a stav výpočtu $B_{state} = 5kB$. S pravděpodobností $P = 10\%$ není programový kód agenta přítomen na cílovém uzlu. V tom případě je iniciován proces přijetí kódu a požadavek má velikost $B_{cr} = 1kB$. Požadavek interakce $B_{req} = 1kB$. Na obrázku 6.1a) je vidět srovnání zatížení sítě při využití RPC (B_{RPC}) a migrace agenta (B_{MR}) pro pevnou velikost odpovědi $B_{rep} = 25kB$ v závislosti na selektivitě agenta.

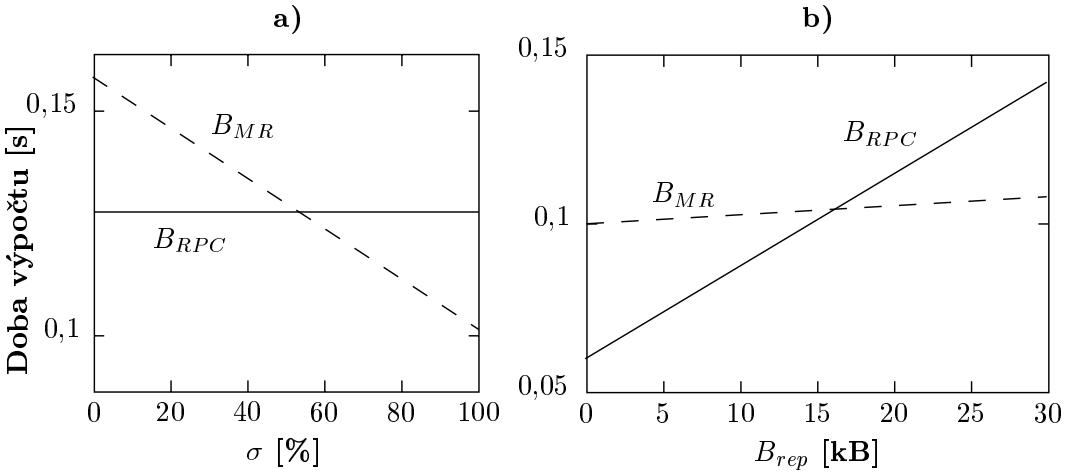
Na obrázku 6.1b) je vidět srovnání zatížení sítě pro interakci pomocí RPC (B_{RPC}) a migrací agenta (B_{MR}) pro pevně danou selektivitu $\sigma = 0,9$ v závislosti na velikosti odpovědi B_{rep} v rozmezí 0 až $30kB$.

Tyto grafy ukazují, že použití migrace agenta je výhodné (sníží se zatížení sítě) jenom v případě, že velikost uvažované odpovědi je velká nebo když je velká selektivita agenta.



Obrázek 6.1: Grafy zatížení sítě

Další grafy 6.2a) a 6.2b) zobrazují odpovídající doby výpočtu pro danou síť charakterizovanou zpožděním $\delta = 30ms$, průchodností $\tau = 400kB/s$. Doba (de)serializace je $\mu = 0s/kB$.



Obrázek 6.2: Grafy doby výpočtu

Zatímco grafy jsou takřka shodné, hlavní rozdíl je v kritickém době (kde se oba grafy protnou). V případě grafu 6.1a) zobrazujícího zatížení sítě je to v bodě $\sigma = 52\%$ a v případě grafu 6.2a) zobrazujícího dobu výpočtu je to v bodě $\sigma = 62\%$.

6.1.2 Sled interakcí

Nyní bude zkoumán sled interakcí s různými uzly. Definujme následující omezující podmínky:

- Sled interakcí s uzly stejně jako velikost odpovědi, velikost dotazu, selektivita a počet interakcí s daným uzlem je předem znám.
- Zpoždění a průchodnost sítě pro každou interakci jsou známy.

Nechť $S = (I_1, \ldots I_n)$ je sled interakcí. i-tá interakce je popsána

$$I_i = \{R_i, m_i, B_{req}, B_{rep}, \sigma_i\}$$

kde R_i je vzdálený uzel pro komunikaci. Každá komunikace se skládá z m_i (vzdálených nebo lokálních) volání procedury s velikostí dotazu B_{req}, velikostí odpovědi B_{rep} a selektivitou σ_i. Velikost agenta po interakci i pro $i = 0, \ldots, n$ je

$$B_{A_i} = (B_{code_i}, B_{data_i}, B_{state_i})$$

kde B_{code_i} a B_{state_i} zůstávají neměnné, zatímco

$$B_{data_i} = B_{data_{i-1}} + m_i(1 - \sigma_i)B_{rep_i}$$

Proces mobility agenta je popsán cílovým vektorem $D = (D_0, \ldots, D_n)$. Při i-tou interakci je agent přenesen na uzel D_i. Migrace agenta mezi interakcí $i-1$ a i se tedy provede pouze v případě, že $D_i \neq D_{i-1}$.

Zatížení sítě a doba výpočtu pro smíšenou interakci pomocí RPC a pomocí migrace agenta jsou následující:

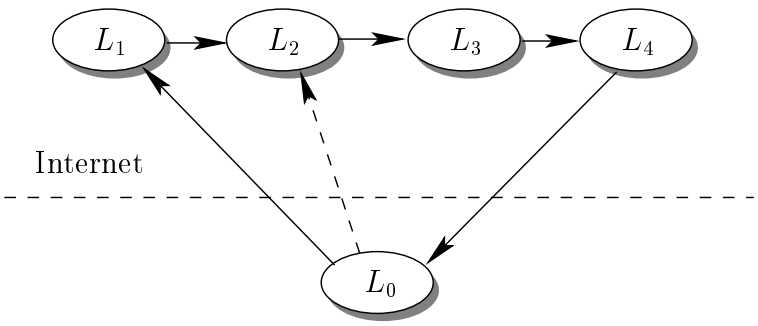
$$B_{Seq}(S, D, B_{A_0}) = \sum_{i=1}^{n} \Big(B_{Mig}(D_{i-1}, D_i, B_{A_{i-1}}) + m_i \cdot B_{RPC}(D_i, R_i, B_{req_i}, B_{rep_i}) \Big)$$

$$T_{Seq}(S, D, B_{A_0}) = \sum_{i=1}^{n} \Big(T_{Mig}(D_{i-1}, D_i, B_{A_{i-1}}) + m_i \cdot T_{RPC}(D_i, R_i, B_{req_i}, B_{rep_i}) \Big)$$

Vyhodnocení sledu interakcí – typický příklad 1

Pro vyhodnocení sledu interakcí v tomto výkonnostním modelu bude uvažována typická aplikace mobilních agentů. Uzel L_0 je spojen se sítí Internet linkou s malou průchodností. Charakteristika této „nehomogenní“ sítě je následující:

- Interakce mezi uzly x a y uvnitř sítě Internet má zpoždění $\delta(x, y) = 10ms$ a průchodnost $\tau(x, y) = 400kB/s$.
- Interakce mezi uzlem L_0 a Internetem má zpoždění $\delta(x, y) = 120ms$ a průchodnost $\tau(x, y) = 50kB/s$.



Obrázek 6.3: Typická aplikace mobilních agentů

Agent začíná svoji migraci na uzlu L_0 a ten postupně interaguje s uzly $L_1, \ldots, L_4$ než se agent vrátí na uzel L_0.

Komunikace s uzly L_2 a L_4 je častější než s jinými uzly.

- Je uvažován agent s těmito charakteristikami: $B_{code} = 10kB$, $B_{data} = 5kB$ a $B_{state} = 5kB$.
- Kód agenta není přítomen na cílovém uzlu ($P = 100\%$).
- Velikost data agenta se nezvětšuje ($\sigma = 1$).

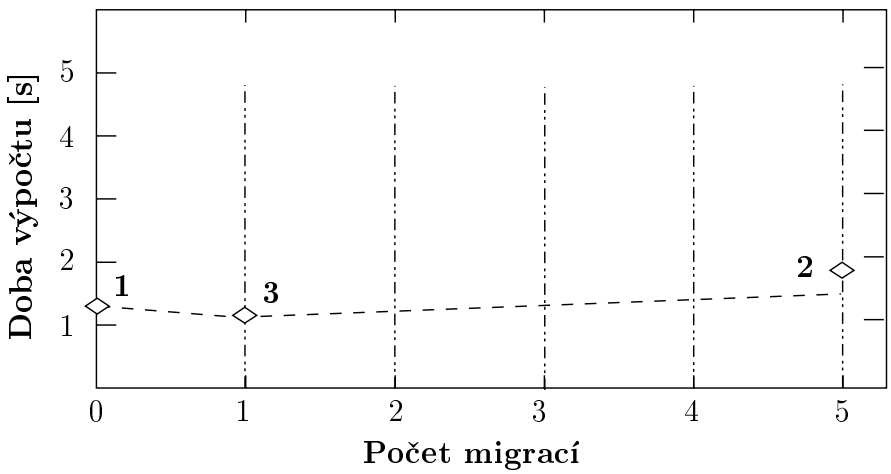
Proces migrace agenta (sled interakcí S_1) je vidět v tabulce 6.1.

i	R_i	m_i	B_{req_i}	B_{rep_i}	σ_i
1	L_1	1	50	2000	1
2	L_2	1	500	4000	1
3	L_3	1	50	2000	1
4	L_4	1	500	4000	1
5	L_0	1	500	10	1

Tabulka 6.1: Sled interakcí pro příklad 1

Na grafu 6.4 je vidět doba výpočtu pro všechny uzly z cílového vektoru D v závislosti na počtu migrací.

Tři doby výpočtu jsou v grafu označeny a odpovídající hodnoty cílového vektoru D, doby výpočtu T_{Seq} a zatížení sítě B_{Seq} jsou v tabulce 6.2. Výsledek číslo 1 ukazuje dobu výpočtu v případě, že agenti používají pouze RPC a vůbec nemigrují. Výsledek číslo 2 naproti tomu zobrazuje případ, kdy agenti vždy migrují na



Obrázek 6.4: Graf doby výpočtu pro typický příklad 1

cílový uzel pro každou z pěti interakcí. Doba výpočtu je v tomto případě mnohem větší než při použití RPC komunikace.

Výsledek č.	Cílový vektor D	T_{Seq}	B_{Seq}
1	$L_0, L_0, L_0, L_0, L_0, L_0$	1,2220	13100
2	$L_0, L_1, L_2, L_3, L_4, L_0$	1,8075	105000
3	$L_0, L_2, L_2, L_2, L_2, L_2$	1,1117	30110

Tabulka 6.2: Výkonnostní hodnoty pro příklad číslo 1

Je tedy jasné, že interakce pomocí migrace agenta není v případě pěti interakcí nejrychlejším řešením. Protože uzel L_0 je připojen pomalou linkou, je lepší použít alespoň jednu migraci do sítě Internet (uzly $L_1 \ldots L_4$) a výsledky získat pomocí RPC. Nejkratší doby výpočtu se dosáhne v případě číslo 3, kdy agent migruje pouze jednou. Z cílového vektoru je jasné, že agent migruje pouze na uzel L_2 před interakcí I_1 (ne na uzel L_1 s nímž chce interagovat, ale na první uzel s větším počtem interakcí), jak je zobrazeno na obrázku 6.3 čárkovanou čarou, a zůstane zde do konce sledu interakcí. Zatížení sítě je opět větší než v případě číslo 1, ale inteligentní použití jedné migrace snížilo dobu výpočtu oproti případu číslo 1 (interakce pouze pomocí RPC).

Vyhodnocení sledu interakcí – typický příklad 2

Ve druhé typické aplikaci bude uvažován stejný agent (B_A, P a σ), ale poněkud se změní sled interakcí podle tabulky 6.3. Hlavně se změnily interakce I_2 a I_4,

když místo jedné interakce s velkým objemem dat dotazu/odpovědi se použije 10 interakcí s menším objemem dat, přičemž celkové množství dat se nezměnilo.

i	R_i	m_i	B_{req_i}	B_{rep_i}	σ_i
1	L_1	1	50	2000	1
2	L_2	10	50	400	1
3	L_3	1	50	2000	1
4	L_4	10	50	400	1
5	L_0	1	500	10	1

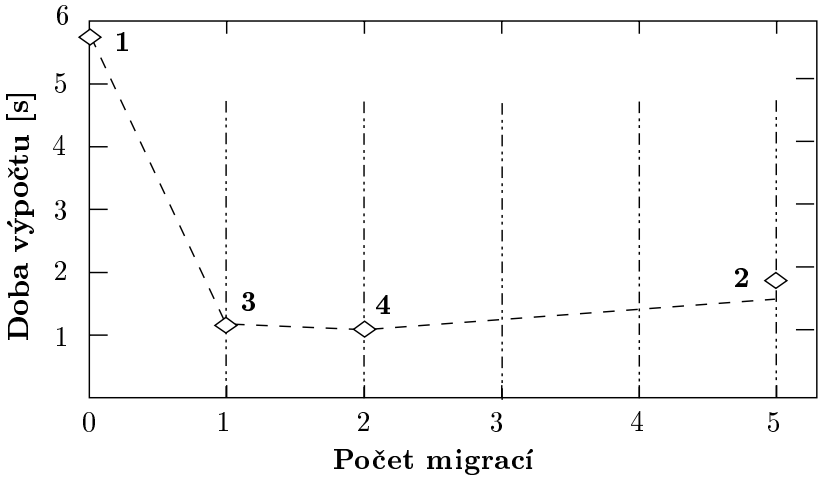
Tabulka 6.3: Sled interakcí pro příklad 2

Na grafu 6.5 je vidět doba výpočtu pro všechny uzly z cílového vektoru D v závislosti na počtu migrací. Nejkratší doby výpočtu (výsledek číslo 4) se dosáhne pro dvojnásobnou migraci agenta. Tabulka 6.4 obsahuje odpovídající hodnoty cílových vektorů, doby výpočtu a zatížení sítě.

Výsledek č.	Cílový vektor D	T_{Seq}	B_{Seq}
1	$L_0, L_0, L_0, L_0, L_0, L_0$	5,5420	13100
2	$L_0, L_1, L_2, L_3, L_4, L_0$	1,8075	105000
3	$L_0, L_2, L_2, L_2, L_2, L_2$	1,2917	30110
4	$L_0, L_2, L_2, L_2, L_4, L_4$	1,1629	46610

Tabulka 6.4: Výkonnostní hodnoty pro příklad číslo 2

Výsledek číslo 1 ukazuje hodnoty pro interakci pomocí RPC. Přestože objem dat zůstal nezměněn, doba výpočtu se značně prodloužila z důvodu většího počtu komunikací (každá komunikace pomocí RPC prodlužuje dobu výpočtu o dvě zpoždění sítě). Výsledek číslo 2 zobrazuje použití migrace agenta vždy na odpovídající uzel pro interakci. Výsledky se nijak neliší od příkladu předchozího, protože všechny komunikace jsou prováděny lokálně na uzlech. Je tedy vidět, že použití migrace v tomto případě je lepší než čisté RPC. Doba výpočtu ve výsledku číslo 3 již není optimální pro tento případ, neboť se objem dat pro komunikaci s uzlem L_4 je větší než migrace agenta na tento uzel před interakcí I_4. Lepší je tedy výsledek číslo 4.



Obrázek 6.5: Graf doby výpočtu pro typický příklad 2

6.2 Experimentální ověření

Pro ověření výkonnostních modelů byl použit navržený systém mobilních agentů `SMAS`. Uzel L_0 byl spojen se sítí pomalejší linkou a uzly $L_1, \ldots L_4$ byly umístěny přímo v univerzitní síti. Tato organizace uzlů poskytuje pomalé spojení uzlu L_0 se zbývajícími uzly, které již spolu komunikují rychle po síti s vyšší průchodností. Hodnoty zpoždění $\delta(x, y)$, průchodnosti $\tau(x, y)$ a doba serializace/deserializace μ byly změřeny pomocí systému mobilních agentů.

i	R_i	m_i	B_{req_i}	B_{rep_i}	σ_i
1	L_1	1	700	3000	1
2	L_2	1	3500	15000	1
3	L_3	1	700	3000	1
4	L_4	1	3500	15000	1
5	L_0	1	3000	700	1

Tabulka 6.5: Sled interakcí pro experimentální příklad 3

V tabulce 6.5 je sled interakcí, kdy komunikace s uzly L_2 a L_4 je větší než v případě uzlů L_1 a L_3. Tato tabulka odpovídá přibližně teoretickému případu 1 na straně 129. V tabulce 6.6 je jedna interakce s velkým objemem dat s uzly L_2 a L_4 nahrazena pěti menšími tak, aby se celkové množství přenášených dat nezměnilo. Tento příklad odpovídá teoretickému příkladu 2 na straně 131.

i	R_i	m_i	B_{req_i}	B_{rep_i}	σ_i
1	L_1	1	700	3000	1
2	L_2	5	700	3000	1
3	L_3	1	700	3000	1
4	L_4	5	700	3000	1
5	L_0	1	3000	700	1

Tabulka 6.6: Sled interakcí pro experimentální příklad 4

Aby mohlo být měření provedeno, byl vytvořen jeden statický agent na uzlu L_0, který vytvořil ostatní mobilní agenty, kteří komunikovali s uzly $L_1, \ldots, L_4$ podle obrázku 6.3. Statický agent měřil čas inicializace mobilních agentů. Jejich změřené charakteristiky jsou: $B_{code} = 10kB$, $B_{data} = 32kB$ a $B_{state} = 2kB$. Kód agenta nebyl nikdy odeslán (byl již na uzlech přítomen, $P = 0\%$) a velikost dat se nezměnila ($\sigma = 1$).

Mobilní agenti se chovali podle jedné z následujících tří strategií:

- **Pouze RPC** – agenti zůstávali na uzlu L_0 a používali RPC pro interakci.
- **Vždy migrace** – vždy se přemístili na další uzel as pak použili lokální volání procedury.
- **Optimalizace** – používali vztah pro výpočet T_{Seq} (na straně 129), aby rozhodli kdy a na jaký uzel mají migrovat.

6.2.1 Výsledky experimentů

Doba výpočtu pro sled interakcí S_3 je v tabulce 6.7. Pro každou strategii byl opakován pokus 50-krát. Podobně jako v příkladě 1 je strategie „pouze RPC“ rychlejší než „vždy migrace“ a „optimalizace“ nabízí pouze malé zlepšení. Při použití strategie „optimalizace“ mobilní agenti migrovali právě jednou ve 49 z 50 měření. 47-krát migrovali na uzel L_2 a 2-krát na uzel L_4, což způsobilo neustále se měnící zatížení sítě, které každý agent měřil.

Doba výpočtu pro sled interakcí S_4 a zprůměrovaná pro 50 měření je v tabulce 6.8. Jak bylo očekáváno, strategie „vždy migrace“ byla lepší než „pouze RPC“ a strategie „optimalizace“ přinesla další zlepšení. Při použití strategie „optimalizace“ mobilní agenti migrovali právě jednou ve 30 případech (11-krát na uzel L_2, 6-krát na uzel L_3 a 13-krát na uzel L_4) a ve 20 případech mobilní agenti migrovali právě 2-krát (14-krát na uzly L_2 a L_4 a 6-krát na uzly L_3 a L_4). Neustále se měnící poměry v síti opět způsobily tyto odchylky.

Strategie	Doba výpočtu [ms]	Průměrná odchylka měření [ms]
pouze RPC	7501	748
vždy migrace	9793	1140
optimalizace	7462	1341

Tabulka 6.7: Výsledky měření pro příklad číslo 3

Strategie	Doba výpočtu [ms]	Průměrná odchylka měření [ms]
pouze RPC	19127	1516
vždy migrace	11394	1414
optimalizace	10953	1341

Tabulka 6.8: Výsledky měření pro příklad číslo 4

6.3 Srovnání s jinými systémy

Pro podporu mobility využívá standardní serializace jazyka Java. Kód agenta je přenášen ze specializovaného serveru (*codebase server*) na vyžádání. Přenos kódu agenta i jeho stavu při migraci je zabezpečen použitím protokolu s veřejným klíčem.

Pojmenování částí systému je založené na DNS, přičemž je striktně odděleno používání *jmen* (jednoznačná identifikace, forma čitelná pro člověka) a *adres* (vstupní bod pro komunikaci). Migrace agentů je možná pouze na absolutní místo specifikované ve formátu URL. Mobilita agentů je podporována serializačními možnostmi jazyka Java. Zachycení stavu výpočtu na úrovni vlákna není podporováno. Agent je přenášen metodou *neaktivní kopie*.

Agent je vkládán do *skladu* spolu se svým *místem*. Pro zdroje jsou vytvořeny *zástupci*, kteří se dynamicky přizpůsobují každému agentovi. Stejný mechanismus je použit při bezpečné komunikaci mezi agenty pomocí volání vzdálených metod, pak se jedná o *komunikační zástupce*. Komunikace po síti je také možná pomocí volání vzdálených metod.

V systému je také vyřešen problém ochrany agentů před nebezpečnými servery. Systém poskytuje bezpečný mechanismus pro detekci práv vlastníků a tvůrců agenta, který je založen na *principals* a *credentials*. Množina služeb poskytovaná agentů skladem agentů (*globální služby*) nebo místem (*individuální služby*) je omezena *právy* (statické přidělování) a *omezeními* (dynamické přidělování).

6.3.1 Srovnání na aplikační úrovni

Pro otestování výkonnosti systému SMAS a jeho srovnání s jinými systémy byla vytvořena typická aplikace mobilních agentů. Při jejím vytváření se vycházelo ze zamýšleného použití tohoto systému, tedy sběr informací z uzlů propojených sítí Internet, přičemž klient je spojen se sítí pomalou linkou. Tato aplikace klade nároky hlavně na bezpečnost systému (data jsou přenášena sítí Internet) a také na použitou metodu komunikace (vlastní režie systému).

Pro porovnání výsledků byl vybrán systém *Aglets* [Ibm1998], který je volně šířen a navíc široce používán. To dovoluje srovnat výsledky obou systémů a rozhodnout, který je výkonnější. Graf migrace je shodný s obrázkem 6.3. Klient na uzlu L_0 vytvoří agenta resp. aglet a odešle jej na uzel L_1 atd. Na každém uzlu se agent resp. aglet pokouší najít v databázi výskyt řetězce. Byly měřeny následující veličiny:

- doba výpočtu na každém uzlu (od spuštění agenta do jeho ukončení) – tím se zjistí výpočetní výkonnost systému a režie při přidělování služby,
- celková doba výpočtu (doba od vyslání agenta do jeho navrácení) – pro zjištění režie systému při migraci,
- zatížení sítě bylo z technických důvodů (přepínaná síť) možno měřit pouze na uzlu L_0.

Měření bylo provedeno 50-krát, kód agenta byl vždy na uzlech přítomen.

	SMAS	Aglets
Celková doba výpočtu [ms]	32183	20695
Průměrná doba výpočtu na 1 uzlu [ms]	4212	3021
Zatížení sítě [B]	50456	30147

Tabulka 6.9: Výsledky měření

Jak je vidět v tabulce 6.9, systém *Aglets* je ve všech ohledech rychlejší. Je však třeba si uvědomit, že není koncipován jako bezpečný. Systém SMAS naproti tomu musí ověřovat certifikáty agenta při každé migraci i při přístupu ke službě systému. Navíc se na každém uzlu musí vytvořit nové objekty (místo, lokální a komunikační zástupce, brány, metadata atd.), které jsou nezbytné pro běh systému.

6.4 Shrnutí

V této kapitole byly shrnuty výsledky experimentů s mobilními agenty. Byl představen výkonnostní model mobilních agentů, ve kterém mohou agenti komunikovat voláním vzdálených metod nebo se přesunout k partnerovi a komunikovat lokálně. Následně byl model rozšířen na sled interakcí. Z výsledků vyplývá, že použití kombinace obou přístupů je lepší než použití čistě jedné nebo druhé metody. Teoretické výsledky byly ověřeny v systému `SMAS`. V druhé části byl porovnán navržený systém se systémem existujícím – Aglets. Poněkud vyšší režie systému souvisí s navrženou strukturou, kdy se vytváří nové objekty nezbytné pro běh systému.

Kapitola 7

Závěr

Použití mobilních agentů pro vývoj a vytváření distribuovaných aplikací přináší výhody, které z nich dělají atraktivní alternativu k tradičnímu pojetí komunikace v modelu klient-server. Jedná se zejména o větší autonomii při jejich vykonávání, snížení množství předávaných dat, dynamická aktualizace služeb, atd. Mnoho nových aplikací, které vznikají spolu s dynamickým vývojem Internetu – jako jsou elektronický obchod, online výuka, získávání informací ze sítě nebo dolování dat (*data mining*) – jsou vhodné pro použití mobilních agentů již ze své podstaty. Stejně tak se dají použít i jako alternativní řešení při vývoji distribuovaných vědeckých výpočtů, kdy se využijí pro distribuci zatížení na jednotlivé uzly sítě. Přesto se zatím mobilní agenti v praxi příliš nepoužívají, neboť nemají dostatečné zabezpečení a také nejsou navzájem kompatibilní. Zavedení mobilních agentů do praxe není možné bez kontroly integrity jejich kódu a dat, spolehlivé autentifikace agentů, ochrany zdrojů na uzlech atd. I když bylo vytvořeno mnoho experimentálních (i komerčních) systémů mobilních agentů, žádný z nich nezohlednil bezpečnost jako hlavní část návrhu. Tato práce se proto zabývala návrhem systému mobilních agentů skládajícího se z několika vrstev, což dovoluje přesně definovat a oddělit jednotlivé úrovně bezpečnostní politiky.

7.1 Vlastní přínos práce

Byla navržena a implementována infrastruktura systému mobilních agentů, tak jak byla popsána v předešlých kapitolách. Systém podporuje všechny operace s agenty, jejich migraci a poskytuje přístup ke službám na uzlech.

- *Vícevrstvý návrh:* Při návrhu struktury se vycházelo hlavně z bezpečnostních požadavků. Agent je tak naprosto oddělen od okolí svým *místem*. Veškerý styk s okolím je prováděn pomocí *zástupců*. Kromě přístupu k externím službám nabízí místo i speciální *individuální služby*. Tyto služby slouží jen pro zprostředkování komunikace s různými subsystémy místa. Jsou vytvořeny místem při jeho vzniku a při zániku místa jsou zničeny. K individuální

službě místa může přistupovat pouze a jen agent umístěný v tomto místě; dochází tak automaticky k autentifikaci a autorizaci.

Nadřazená vrstva (*sklad*) zajišťuje vkládání a rušení agentů, vytváření a rušení míst, migraci, pasivaci a aktivaci agentů. Udržuje také množinu nabízených služeb. Jednotlivé sklady se liší množinou nabízených služeb a bezpečnostní politikou.

- *Přenos kódu na vyžádání:* Informace o veškerém programovém kódu agenta je uchována odděleně v tzv. *metadatech* a nazývá se *codebase*. Každý agent má svůj vlastní *classloader* což znamená, že má také vlastní instanci tohoto kódu. Nemůže tak dojít ke konfliktu verzí ani k úmyslnému napadení pomocí upraveného kódu.

- *Vlastník a tvůrce:* Pro autorizaci akcí agenta (především jeho přístupu ke službám) se jen výjimečně použije identita jeho samého. Téměř výhradně se používají identity *vlastníka* (*owner*) a *tvůrce* (*creator*) uložené v *metadatech*. Obě identity obsahují objekty *principals* a *credentials*. Principals obsahuje jméno entity a slouží k ověření identity vlastníka nebo tvůrce. Credentials se používá k autentifikaci pomocí certifikátů.

- *Práva a omezení:* Jazyk Java verze 2 obsahuje sám o sobě poměrně propracovanou bezpečnostní architekturu založenou na *právech* (*permissions*). Ta je však navržena pouze pro případ mobilního kódu (tzv. appletů) a proto autorizace probíhá jen na úrovni kódu. Doplňkový balík *JAAS* sice tuto architekturu rozšiřuje o autorizaci na základě identity, ovšem jen na úrovni vláken.

  Pro autorizaci přístupu ke službám je celá bezpečnostní architektura Javy obtížně použitelná a proto byly zavedeny tzv. *omezení* (*constraints*). Omezení jsou vytvářena během života agenta *skladem*. Obě tyto architektury se navzájem doplňují.

- *Zástupce a komunikační zástupce:* Oddělení agenta od služeb je zajištěno pomocí *zástupců* (*proxy*), přičemž byl použit standardní mechanismus *JDK*.

  Další významnou vlastností zástupce je, že může být *odpojen* (*disconnect*). Touto operací se zruší odkaz na službu a zástupce je od toho okamžiku nefunkční. Použití je významné především při uvolnění služby, kdy se tato operace provede a zabrání se tak používání již uvolněného zástupce.

  Pro udržení komunikace při migraci agenta je používán mechanismus *lokálních zástupců* (*local proxy*). Používá se však při veškeré komunikaci v systému, nejen při komunikaci s agentem. Komunikační zástupce může být ve dvou stavech: *připojený* (*connected*) a *rozpojený* (*disconnected*). Pokud je *připojený*, obsahuje přímo referenci na cílový objekt. V tomto stavu je zcela

transparentní a je možné normálně komunikovat. Pokud je *rozpojený*, není komunikace navázána.

- *Komunikace:* Pro komunikaci se používá vzdálené volání metod *RMI*. Tato metoda pracuje na úrovni objektů, zatímco požadovanou komunikační entitou je agent nebo stroj. Tato entita musí být reprezentována vzdáleně přístupným objektem, který se nazývá *brána* (*gate*).

  Bran může být několik, ale jejich chování musí být zcela shodné. Liší se pouze komunikačním protokolem (pokud je jediným podporovaným protokolem *RMI*, existuje jen jedna brána).

- *Autentifikace při komunikaci:* Při komunikaci je nutné, aby si strany navzájem prokázaly svou identitu (autentifikace). Teprve podle prokázané identity pak mohou rozhodnou, zda se druhé straně dá důvěřovat (autorizace).

  Pro autentifikaci se používá algoritmus s automatickou výměnou veřejných klíčů (certifikátů). Jedná se o algoritmus *výzva-odpověď* (*challenge-response*) pro authetifikaci při vzdáleném volání metody, využívající výměnu a podepisování náhodně generovaných čísel.

S využitím prostředků jazyka Java a standardních rozšiřujících balíků byl navržen a vytvořen robustní a bezpečný systém splňující cíle práce. Jeho funkčnost byla otestována na několika typických aplikacích mobilních agentů.

7.2 Možná rozšíření návrhu

- Z bezpečnostních důvodů je agent při požadavku návratu (*retract*) vrácen pouze v serializované formě tj. bez metadat, jména a adresy. Tyto informace totiž nejsou chráněny žádnou autentifikací a tudíž by je mohl získat kdokoli bez autorizace. Pro některé aplikace např. vyrovnávání zátěže by však bylo lepší, aby se do serializované podoby přidaly i informace o agentovi.

- V návrhu není řešena podpora pro vzdálené monitorování činnosti agentů. Hostitel nemůže vzdáleně získávat informace o stavu agenta, pokud je agent sám neposkytuje. Při návrhu nebyla tato podpora uvažována, neboť by tím vzniklo slabé místo k proniknutí do systému – v tomto případě ze strany nebezpečného hostitele.

- Z hlediska perzistentního uchování agentů by bylo vhodné rozšířit návrh o možnost ukládání objektů do vhodné databáze. Tím by se usnadnila správa celého systému.

- S předchozím bodem úzce souvisí správa verzí agentů a obecně objektů v systému. Tím by vznikla možnost existence několika verzí téhož kódu, což by jistě významnou měrou usnadnilo správu systému.

7.3 Shrnutí

Tato disertační práce poskytuje přehled typů softwarových agentů. Agenti jsou rozděleni na základě svých vlastností, přičemž důraz je kladen na mobilní agenty. Jedná se o moderní trend realizace distribuovaných aplikací, který je prozatím nejobecnějším modelem komunikace mezi uzly distribuovaného systému. Pro využití mobilních agentů je třeba zajistit podporu ze strany výpočetního prostředí. Systém mobilních agentů musí mít pevně definovanou strukturu, aby mohl poskytovat služby pro přístup ke svým prostředkům. Při návrhu systému je však třeba vyřešit i otázku bezpečného přístupu k prostředkům. Řešením těchto zdánlivě protichůdných požadavků je systém skládající se z několika vrstev, které jsou navzájem striktně oddělené. Jednotlivé komponenty systému řeší identifikaci a adresování objektů, přístup ke službám na uzlech, autentifikaci a autorizaci objektů a komunikaci mezi objekty.

Hlavním cílem práce byl návrh systému pro podporu mobility agentů v prostředí Internetu. Z toho vyplývá, že bezpečnost systému se v průběhu návrhu stala hlavním požadavkem. Ostatní vlastnosti se systému se jí musely přizpůsobit tj. všechny akce v systému jsou autorizovány. Hlavním bodem práce byl tedy návrh takového systému, jehož programová realizace byla ověřena na typické aplikaci mobilních agentů. V závěru práce je realizace systému otestována a výsledky srovnány se systémem RPC a systémem Aglets.

Literatura

[Agr1987] P.E. Agre, D. Chapman: *Pengi: An Implementation of a Theory of Activity.*
Proceedings of the 6th National Conference on Artificial Intelligence, San Mateo, 1987

[Ban1986] J.S. Banino: *Parallelism and Fault Tolerance in Chorus.*
Journal of Systems and Software, 1986

[Bar1996] M. Barbuceanu, M.S. Fox: *The design of a coordination language for multi-agent system.*
Inteligent Agent III. Agent Theories, Architectures, and Languages, 1996

[Bau1997a] J. Baumann, N. Radouniklis: *Agent groups for mobile agent systems.*
Distributed Applications and Interoperable Systems,
Chapman & Hall, 1997

[Bau1997b] J. Baumann, N. Radouniklis, K. Rothermel: *Communication Concepts of Mobile Agent System.*
Proceedings of the 1st International Workshop, Mobile Agent 97, Springer, 1997

[Bra98] T. Bray, J. Paoli, C. M. Sperberg: *Extensible Markup Language.*
W3C, February 1998
`http://www.w3.org/TR/1998/REC-xml19980210`

[Bro1986] R.A. Brooks: *A Robust Layered Control System for a Mobile Robot.*
IEEE Journal of Robotics and Automation 2(1), 1986

[Bro1991] R.A. Brooks: *Inteligence without Representation.*
Artificial Inteligence 47, 1991

[Cab1998] L. Cable: *Extensible Runtime Containment and Services Protocol for JavaBeans.*
Sun Microsystems, Inc., 1998

[Cmu2001] CMU: *A CMU Common Lisp Documentation Collection.*
`http://www.cons.org/cmucl`, 2001

[Dav1995] N.J. Davies, R. Weeks: *Jasper: Communicating Information Agents.*
Proceedings of the 4th International Conference on the World Wide Web, 1995

[Die1999] T. Dierks, C. Allen: *The TLS Protocol Version 1.0.*
RFC 2246, January 1999

[Deu1999] D. Deugo, M. Weiss: *A Case for Mobile Agent Patterns.*
Mobile Agents in the Context of Competition and Cooperation MAC^3, 1999

[Fra1996] S. Franklin, A. Graesser: *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.*
Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, 1996

[Gil1995] D. Gilbert, M. Aparicio: *IBM Intelligent Agent Strategy.*
IBM Corporation, 1995

[Gen1997] General Magic, Inc.: *Odyssey web page.*
General Magic, 1997
`http://www.genmagic.com/technology/odyssey.html`

[Gos1996] J. Gosling, B. Joy, G. Steele: *The Java Language Specification.*
Addison-Wesley, August 1996

[Gra1996] R.S. Gray: *Agent Tcl: A flexible and secure mobile-agent system.*
Proceedings of the Fourth Annual Tcl-Tk Workshop TCL'96,
July 1996

[Har1995] C.G. Harrison, D.M. Chess, A. Kershenbaum: *Mobile Agents: Are they a good idea?.*
Technical Report, IBM Research Division, T.J. Watson Research Center, March 1995

[Hoh1998] M. Hohlfield, B. Yee: *How to migrate agents.*
Technical Report CS98-588, Computer Science and Engineering Department, University of California, June 1998

[Hou1999] R. Housley, W. Ford, W. Polk, D. Solo: *Internet X.509 Public Key Infrastructure Certificate and CRL Profile.*
RFC 2459, January 1999

[Huh1994] M.N. Huhns, M.P. Singh: *Distributed Artificial Intelligence for Information Systems.*
CKBS-94 Tutorial, June 1994

[Cha1994] B. Chaib-Draa: *Distributed Artificial Inteligence: An Overview.*
Encyclopedia of Computer Science and Technology, Vol 29, 1994

[Chi1997] T.H. Chia, S. Kannapan: *Strategically Mobile Agents.*
Proceedings of the First International Workshop on Mobile Agents MA' 97, Springer, 1997

[Ibm1997] IBM Tokyo Research Laboratory: *Aglets-based e-marketplace: Concept, architecture and applications.*
IBM Tokyo Research Laboratory, Research Report RT-0253, 1997

[Ibm1998] IBM, Inc.: *IBM Aglets Documentation Web Page.*
`http://aglets.trl.ibm.co.jp/documentation.html`, 1998

[Joh1995] D. Johansen, R. van Renesse, F.B. Schneider: *An Introduction on TACOMA Distributed System.*
Technical Report 95-23, Department of Computer Science, University of Tromsø, June 1995

[Joh1996] D. Johansen, R. van Renesse, F.B. Schneider: *Operating System Support for Mobile Agents.*
Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems HotOS-V, May 1996

[Jul1988] E. Jul, H. Levy, N. Hutchinson, A. Black: *Fine-Grained Mobility in the Emerald System.*
ACM Transactions on Computer Systems 6(1), February 1988

[Kar1997] G. Karjoth, D. Lange, M. Oshima: *A Security Model for Aglets.*
IEEE Internet Computing, July-August 1997

[Kar1998a] N.M. Karnik, A.R. Tripathi: *Design Issues in Mobile Agent Programming Systems.*
IEEE Concurrency 6(6), July-September 1998

[Kar1998b] N.M. Karnik, A.R. Tripathi: *Agent Server Architecture for the Ajanta Mobile-Agent System.*
Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA' 98, 1998

[Kar1998c] N.M. Karnik: *Security in Mobile Agent Systems.*
Department of Computer Science, University of Minnesota, 1998

[Kin1995] J.A. King: *Intelligent Agents: BringingGood Things to Life.*
AI Expert, February 1995

[Kon1996] D. Konstantas, J.H. Morin, J. Vitek: *MEDIA: a platform for the commercialization of electronic documents.*
Object Applications, University of Geneva, 1996

[Kot1997] D. Kotz, R. Gray, S. Nog, D. Rus: *Agent Tcl: Targeting the Needs of Mobile Computers.*
IEEE Internet Computing, July-August 1997

[Lab1998] Y. Labrou, T. Finin, Y. Peng: *Mobile agents can benefit from standards efforts on inter-agent communication.*
IEEE Communications, Vol. 36, Nr. 7, 1998

[Lab1999] Y. Labrou, T. Finin, Y. Peng: *The current landscape of Agent Communication Languages.*
IEEE Intelligent Systems, Vol. 14, No. 2, March-April 1999

[Lan1998] D. Lange, M. Oshima: *Seven Good Reason for Mobile Agent.*
Communication of ACM 42, March 1998

[Lan1999a] D. Lange, M. Oshima: *Programming and Deploying Java Mobile Agents with Aglets.*
Addison-Wesley, 1999

[Lan1999b] D. Lange: *Java Mobile Agents.*
JavaOne '99 BOF, 1999

[Lev1995] J.Y. Levy, J.F. Ousterhout: *A Safe Tcl Toolkit for Electronic Meeting Places.*
Proceedings of the First USENIX Workshop on Electronic Commerce, July 1995

[Lin1996] T. Lindholm, F. Yellin: *The Java Virtual Machine Specification.*
Addison-Wesley, 1996

[Mae1994] P. Maes: *Agents that Reduce Work and Information Overload.*
Communications of ACM 37, 1994

[Mat1999] V. Matena, M. Hapner: *Enterprise JavaBeans Specification Version 1.1 Final.*
Sun Microsystems, Inc., 1999

[Mil1999] D. Milojičić, F. Douglis, R. Wheeler: *Mobility: Processes, Computers, and Agents.*
Addison-Wesley, February 1999

[Mit1997] Mitsubishi Electric: *Concordia: A Infrastructure for Collaborating Mobile Agents*.
Proceedings of the 1st International Workshop of Mobile Agents MA' 97, April 1997

[Moc1987] P. Mockapetris: *Domain Names – Concepts and Facilities*.
RFC 1034, November 1987

[Obj1997] ObjectSpace, Inc.: *ObjectSpace Voyager Core Package Technical Overview*.
ObjectSpace, Inc., 1997

[Sho1994] Y. Shoam, B. Thomas: *Agent-Oriented Programming*.
Encyclopedia of Computer Science and Technology, Vol 29, 1994

[Smi1988] J.M. Smith: *A Survey of Process Migration Mechanisms*.
ACM Operating System Review 22(3), 1988

[Sta1990] J.W. Stamos, D.K. Gifford: *Remote Evaluation*.
ACM Transactions on Programming Languages and Systems 12(4), October 1990

[Str1997] M. Straßer, M. Schwehm: *A Performance Model for Mobile Agent Systems*.
International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA' 97, 1997

[Sun2001a] Sun Microsystems, Inc.: *Java 2 SDK Documentation*.
Sun Microsystems, Inc., 2001

[Sun2001b] Sun Microsystems, Inc.: *Java 2 Platform API*.
Sun Microsystems, Inc., 2001

[Sun2001c] Sun Microsystems, Inc.: *Object Serialization*.
Sun Microsystems, Inc., 2001

[Sun2001d] Sun Microsystems, Inc.: *JavaBeans*.
Sun Microsystems, Inc., 2001

[Sun2001e] Sun Microsystems, Inc.: *Java Authentication and Authorization Service*.
Sun Microsystems, Inc., 2001

[Sun2001f] Sun Microsystems, Inc.: *Java Naming and Directory Interface*.
Sun Microsystems, Inc., 2001

[Tar1996] J. Tardo, L. Valente: *Mobile Agent Security and Telescript.*
Proceedings of COMPCON Spring ‘96, 1996

[Tay1990] B.H. Tay, A.L. Ananda: *A Survey of Remote Procedure Calls.*
Operating Systems Review 24(3), July 1990

[The1999] W. Theilmann, K. Rothermel: *Maintaining specialized search engines through mobile filter agents.*
Proceedings 3rd Int. Workshop on Cooperative Information Agents CIA‘99, July 1999

[Tho1997] T. Thorn: *Programming Languages for Mobile Code.*
ACM Computing Surveys 29(3), September 1997

[Tri1998] A.R. Tripathi, N.M. Karnik, M.K. Vora, T. Ahmed: *Ajanta – A System for Mobile Agent Programming.*
Technical Report TR98-016, Department of Computer Science, University of Minnesota, April 1998

[Vit1981] J. Vittal: *Active Message Processing: Messages as Messengers.*
Computer Message System, 1981

[Wah1997] M. Wahl, T. Howes, S. Kille: *Lightweight Directory Access Protocol.*
RFC 2251, December 1997

[Whi1995] J.E. White: *Mobile Agents.*
Technical Report, General Magic, Inc., October 1995

[Woo1995] M. Wooldridge, N.R. Jennings: *Agent Theories, Architectures, and Languages: a Survey.*
Intelligent Agents, 1995

[Zim1995] P.R. Zimmermann: *The Official PGP User‘s Guide.*
The MIT Press, 1995

Publikace autora

[1] J. Ledvina, M. Otta, M. Šimek, T.Q. Trung, V. Vavřička: *Systém pro autorizaci přístupu.*
Sborník celostátní konference technických univerzit a průmyslu TRANSFER 98, 1998

[2] M. Šimek: *Plánování procesů v distribuovaném výpočetním prostředí.*
Proceedings of XXIst International Colloquium ASIS 1999 Advanced Simulation of Systems, ISBN 80-85988-41-0, TU Ostrava 1999, str. 279–254

[3] P. Brada, P. Lederbuch, J. Ledvina, M. Otta, L. Petrlík, V. Skala, M. Šimek: *Počítačová grafika a vizualizace dat v paralelním a distribuovaném prostředí.* Výzkumná zpráva projektu VZ 97 155, Plzeň, 1999

[4] M. Otta, M. Šimek: *Debugging Distributed Computations.*
Proceedings of XXIInd International Colloquium ASIS 2000 Advanced Simulation of Systems, ISBN 80-85988-51-8, TU Ostrava, 2000, str. 213–218

[5] M. Otta, M. Šimek: *SMAS - A Simple Mobile Agent System.*
Proceedings of XXIIIrd International Colloquium ASIS 2001 Advanced Simulation of Systems, ISBN 80-85988-61-5, TU Ostrava, 2001, str. 245–250

[6] M. Šimek: *Mobile Agent System Facilities.*
Proceedings of International Conference on Software, Telecommunications and Computer Networks SoftCOM 2001, ISBN 953-6114-44-5, FEST Split, 2001, str. 183–190