1 Úvod

Jini je prostředek pro budování tzv. Jini Networks. Jini síť je tvořena množinou distribuovaných objektů, které si vzájemně poskytují své služby. Nezáleží na tom, zda je distribuovaným objektem poskytujícím službu, pevný disk, tiskárna, časová služba či jiné softwarové nebo hardwarové zařízení. Primárním úkolem Jini je zabezpečit komunikaci mezi klientem a službami a vést registraci těchto služeb bez ohledu na typ, implementaci či hardwarovou platformu.

Cílem této práce je prostudovat možnosti Jini technologie, komunikace uvnitř Jini síťe a s využitím těchto znalostí navrhnout množinu Jini služeb, které budou poskytovat vybrané informace o monitorované síti získané pomocí RMI ze SNMP monitorovací stanice. Těchto informací bude k prezentaci využívat Jini klientská aplikace. Výsledný systém by měl ověřit získané teoretické poznatky a prověřit jednoduchost použití Jini technologie pro distribuované programování.

RMI rozhraní, z něhož budou služby získávat objekty pro komunikaci se SNMP agenty běžícími na konkrétních strojích, je předmětem jiné bakalářské práce. Tyto dvě práce jsou svázány pomocí navrženého rozhraní, jenž reprezentuje třída *NetworkMonitor* .

2 Teoretická část

Tato kapitola bude věnována seznámení se s Jini technologií. V úvodu si řekneme, co je to Jini a v jakých projektech již bylo s úspěchem využito. Dále se tato kapitola bude zabývat architekturou Jini systému, komunikací uvnitř systému a dalšími vlastnostmi, které jsou nezbytné pro tvorbu Jini aplikací.

2.1 Úvod do Jini technologie

Jini je název pro distribuovaný počítačový systém, který nabízí kvalitní mechanismus plynulého přidávání, odebírání a vyhledávání služeb připojených k počítačové síti (Network plug and play). Zařízení nebo software připojený k síti je schopen snadno ohlásit svoji přítomnost. Klient pak může službu vyhledat a zavolat jí, aby vykonala nějaký úkol. Službou v tomto systému může být cokoliv, co vykonává nějakou užitečnou činnost, kterou může využít některý z klientů - pevný disk poskytující možnost uložit data, tiskárna nebo stroj s připojením na internet.

2.1.1 Jini specifikace a její implementace

Jini je specifikace množiny middleware komponent. Jednak obsahuje API, takže programátor může psát služby využívající těchto middleware komponent. Za druhé obsahuje implementaci těchto komponent ve formě Java balíčků. Použitím těchto balíčků u klientské aplikace nebo služby se můžou použít Jini middleware protokoly k přihlášení do Jini systému. Jako bonus jsou v Jini Starter Kit dodávány i zdrojové kódy těchto balíčků a implementace některých standardních služeb (Transaction manager, Lookup Discovery Service a další).

Jini specifikace byla vydána společností Sun Microsystems. Ta posléze vydala i první implementaci postavenou výhradně na programovacím jazyku JAVA a systému volání vzdálených metod RMI. V současné době je k dispozici implementace Jini systému verze 2.1 (http://java.sun.com/developer/products/jini/).

2.1.2 Motivace

Jini technologie bylo s úspěchem využito v mnoha projektech . Seznam několika takových projektů lze nalézt například na webových stánkách společnosti Sun Microsystems
http://www.sun.com/software/jini/news/success.xml

Informant

Pro potřeby Florida Department of Law Enforcement (FDLE) vyvinula společnost Templar systém nazvaný Informant. Tento systém řešil obrovské potíže s prohledáváním šesti databází zločinců. Každá z těchto databází byla postavena na jiné databázové platformě, používala jiných protokolů, dotazovacích jazyků, měla různé struktury tabulek, apod. Problémem bylo, že každá z těchto databází (DNA záznamy, sexuální delikventi, obchodníci s drogami, …) musela být prohledávána zvlášť. Ve většině případů musely být prohledány všechny, protože například člen gangu mohl mít dříve problémy s drogami. Prohledání všech šesti databází trvalo 18 minut. Při vynásobení počtem podezřelých toto číslo mnohdy narůstalo do neúnosných rozměrů, navíc analytici museli podstoupit zvláštní kurz a obdržet oprávnění pro každou databázi. Důležitým požadavkem na tento systém byla i jeho funkčnost při výpadku nějakého prvku.

Řešením se ukázal být Jini systém, který umožňuje přizpůsobování se prostředí. Dynamické vytváření sítě založené na

Jini technologii poskytuje zásadní výhodu, a to přizpůsobení se neustále se měnícímu OSI (Office of State Intelligence) operačnímu prostředí, které zahrnuje databáze na různých místech, jež se připojují a odpojují v různém čase, z různých plánovaných i neplánovaných důvodů. Zdroje dat jsou reprezentovány jako Jini služby, které zapouzdřují detaily závislé na jednotlivých zdrojích a poskytují jednotné rozhraní systému Informant.

Uvádí se, že tento systém zefektivnil vyhledávání natolik, že se prohledávání databází zkrátilo z původních 18 minut na 10 vteřin, což ušetřilo 2,4 milionu dolarů ve formě lidské práce ročně, které mohly být přesměrovány z hledání záznamů na jiné užitečné věci.

2.2 Architektura Jini systému

V následující části si popíšeme základní komponenty Jini systému, pravidla pro registraci služeb a proces komunikace mezi klientskou aplikací a službou.

2.2.1 Komponenty Jini systému

Service - služba je entita, která může být použita osobou, programem nebo jinou službou k výpočtu, uložení dat, komunikaci nebo k libovolnému dalšímu účelu. Klient některé služby může zároveň poskytovat službu jinému klientu

Lookup service - základním stavebním kamenem každého Jini systému je Lookup Service (k označení této entity se běžně používá zkratka LUS, která bude dále používána i v tomto textu). U této služby se všechny další služby registrují a všichni klienti při vyhledávání služby kontaktují LUS, která zprostředkuje komunikaci mezi klientem a službou

Java Remote Metod Invocation - mechanismus volání vzdálených metod. Tento mechanismus neumožňuje pouhý přenos dat od objektu k objektu ale přímo přenos celých objektů sítí

2.2.2 Komunikace v Jini systému

Aby mohla klientská aplikace využívat některé ze služeb, musí být daná služba zaregistrována u LUS. Stejně tak klient se ptá u LUS, zda-li je požadovaná služba zaregistrována. Z předchozích řádek je zřejmé, že prvním krokem každé entity je vyhledání LUS. K tomu účelu lze použít buď *unicast discovery* procesu nebo *broadcast discovery* procesu. Po nalezení LUS se služba zaregistruje pomocí procesu *join*. Klient použije procesu *lookup* k vyhledání vhodné služby. V následující části se budeme zabývat těmito procesy.

Unicast Discovery proces

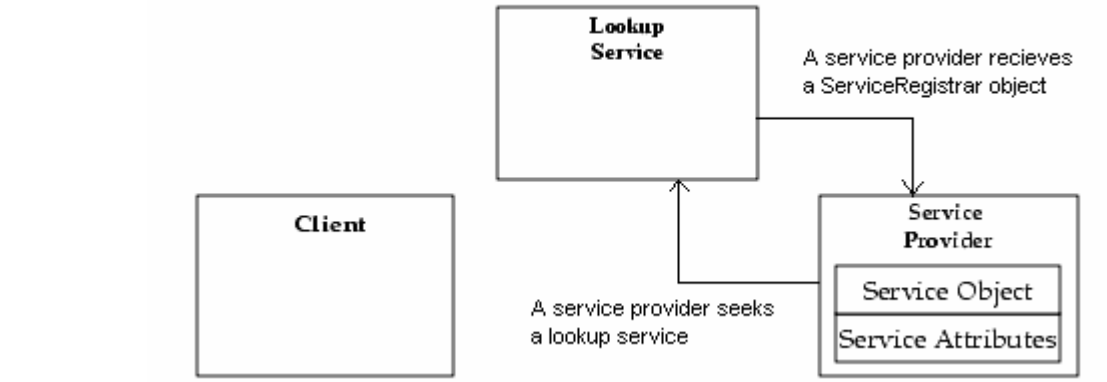
LookupLocator - tento proces nalezení LUS je možné využít, pokud známe její přesné umístění. Pro tento účel se používá třída *LookupLocator*.

```
package net.jini.core.discovery;

public class  LookupLocator {
    LookupLocator(java.lang.String url) throws
                  java.net.MalformedURLException;
    LookupLocator(java.lang.String host,int port);
}
```

Tato třída má dva konstruktory (viz. Zdrojový kód 2.1). V prvním případě musí být URL ve tvaru `jini://host/` nebo `jini://host:port/`, pokud není zadán žádný port, je defaultně použit port 4160. *Host* musí být správné DNS jméno nebo IP adresa.

Po vytvoření objektu typu *LookupLocator* zavoláme na tomto objektu metodu *getRegistrar()*, která navrátí objekt typu *ServiceRegistrar*, pomocí tohoto objektu, pak dále komunikujeme s LUS.

Proces objevení LUS je graficky znázorněn na obrázku 2.1



Broadcast Discovery proces

Tento proces se používá, pokud neznáme přesné umístění LUS. UDP podporuje multicast mechanismus, který používá současná implementace Jini. Jelikož je multicast příliš náročný na administraci a většina routerů blokuje multicast pakety, používá se broadcast discovery pouze v lokálních počítačových sítích.

Groups - některé Lookup Service mohou být dostupné pro všechny, na druhou stranu některé mohou být dostupné pouze pro někoho. Například tiskárna formátu A1 může být ve firmě dostupná pouze pro skupinu designerů. Proto se tato služba registruje u LUS, které bylo nastaveno, že je pouze pro danou skupinu. Tato vlastnost se dá nastavit při startu LUS jako pole typu *String*

LookupDiscovery - třída *LookupDiscovery* se používá k Broadcast discovery procesu a má dva konstruktory, které vidíme ve zdrojovém kódu 2.2 .

```
package net.jini.discovery

public class  LookupDiscovery {
     LookupDiscovery(java.lang.String[] groups)
     LookupDiscovery(java.lang.String[] groups, Configuration
                      conf)
}
```

- Pokud je pole *groups* rovno *LookupDisicovery.ALL_GROUPS* nebo *null* budou nalezeny všechny LUS. Tato varianta se používá nejčastěji.
- Pokud je pole *groups* rovno *LookupDiscovery.NO_GROUPS*, nebo je prázdné, bude objekt vytvořen, ale později bude třeba volat metodu *setGroups()*, abychom mohli vyhledat LUS.
- Pokud je pole *groups* neprázdné, budou vyhledány pouze ty LUS, které jsou určeny pro danou skupinu

DiscoveryListener - po odeslání multicast paketu aplikace očekává, že ze sítě přijde odpověď. Nikdy není předem známo, kolik těchto odpovědí přijde a kdy. Chceme-li tedy, aby naše služba byla upozorněna na odhalení nové Lookup service, je nutné, aby aplikace

registrovala listener s *LookupDiscovery* objektem (viz. zdrojový kód 2.4). Toho dosáhneme implementací rozhraní *DiscoveryListener* (viz. zdrojový kód 2.3) a následnou registrací s tímto listenerem.

```
package net.jini.discovery;

public abstract interface DiscoveryListener {
    public void discovered(DiscoveryEvent e);
    public void discarded(DiscoveryEvent e);
}
```

```
LookupDiscovery discover = null;
try {
    discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
} catch (Exception e) {
    System.err.println(e.toString());
    System.exit(1);
}

discover.addDiscoveryListener(this);
```

DiscoveryEvent - parametr metody *discovered()* z rozhraní *DiscoveryListener* je objekt *DiscoveryEvent*. Tento objekt má jedinou metodu *getRegistrars()*, která vrací pole objektů typu *ServiceRegistrar*, jenž mají stejnou funkci jako u unicast discovery procesu.

Proces join

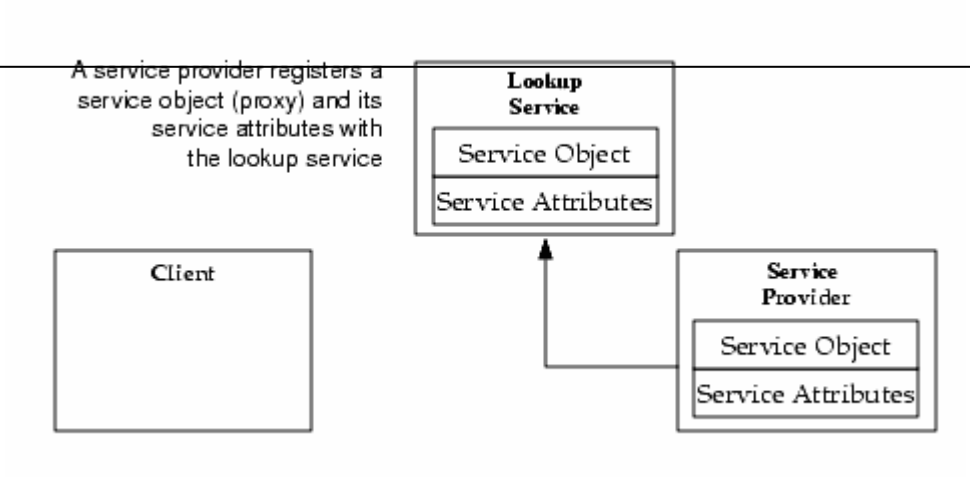
ServiceRegistrar - v okamžiku, kdy entita úspěšně dokončí discovery proces, obdrží od LUS objekt typu *ServiceRegistrar*. Tento objekt umožňuje komunikaci s LUS jak službě, tak klientovi. Služba se pomocí tohoto objektu může zaregistrovat voláním metody *register()* (viz Zdrojový kód 2.5), tento proces se nazývá *join*.

```
package net.jini.core.lookup;

public Class ServiceRegistrar {
    public ServiceRegistration register(ServiceItem item,
    long leaseDuration) throws java.rmi.RemoteException;
}
```

ServiceItem – metodě *register()* je jako parametr předáván objekt typu *ServiceItem* (viz Zdrojový kód 2.6). Jedná se o balík objektů, pomocí kterých je poskytovaná služba dostatečně specifikovaná. Jako jeden z parametrů konstruktoru *ServiceItem* je množina atributů *Entry*, která popisuje službu a bude jí věnována samostatná kapitola. *ServiceID* je při prvním přihlášení *null*, lookup service posléze přiřadí službě unikátní číslo, které by mělo být použito například při opětovné registraci. Objekt je před zasláním k LUS serializován, proto nemohou být použity objekty, které neimplementují rozhraní *Serializable*.

Metoda *register()* pošle kopii objektu *ServiceItem* LUS, kde je tento objekt uložen. Jakmile je toto hotovo, je služba úspěšně zaregistrována a je k dispozici klientům (viz Obrázek 2.2).

```
package net.jini.core.lookup;

public Class ServiceItem {

    public ServiceItem(ServiceID serviceID, Object service,
                       Entry[] attrSets);
}
```



Proces lookup

Stejně jako služba obdrží i klient po vyhledání Lookup service objekt typu *ServiceRegistrar*. Klient ovšem k vyhledání požadované služby zavolá na tomto objektu Metodu *lookup(),* jejíž signatura je vidět ve zdrojovém kódu 2.7 .

```
public Class ServiceRegistrar {
    public java.lang.Object lookup(ServiceTemplate tmpl)
                            throws java.rmi.RemoteException;
    public ServiceMatches lookup(ServiceTemplate tmpl,
                                 int maxMatches)
                            throws java.rmi.RemoteException;
}
```

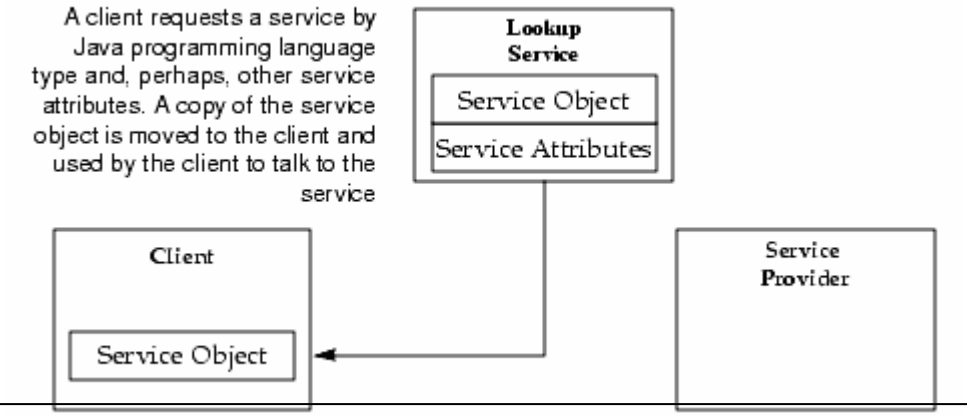
ServiceTemplate - jako parametr je metodě *lookup()* předáván objekt typu *ServiceTemplate*. Tento objekt slouží jako kritérium pro vyhledání služby. Ve svém konstruktoru (viz. Zdrojový kód 2.8) obsahuje atribut *serviceID*, který ovšem často není klientu znám, takže jeho hodnota bývá *null*. Dalším atributem je objekt typu *Class*. Služba implementuje rozhraní, které je dobře známé jak jí, tak klientovi (tzv. well-known interface) a je podle něho služba

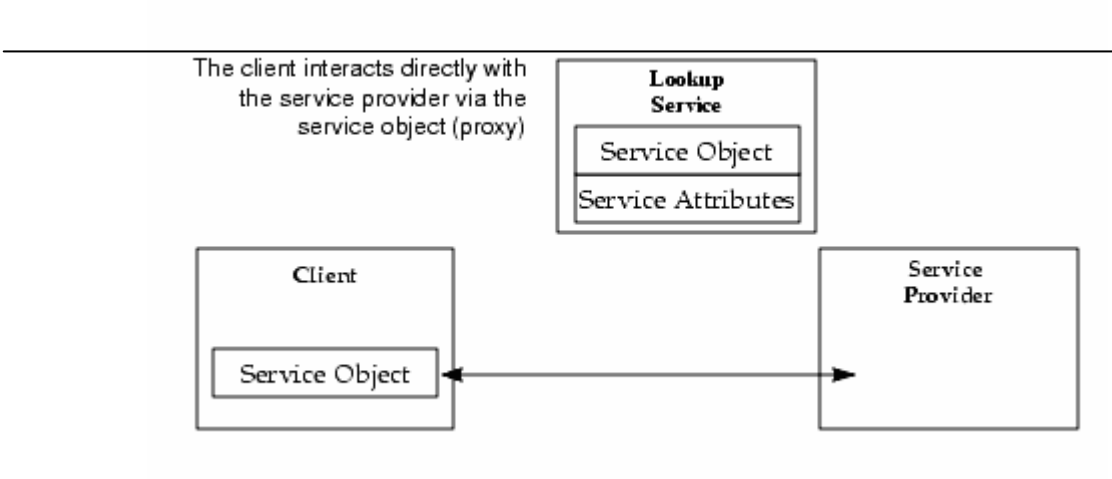
vyhledávána nejčastěji. Atribut *Entry* obsahuje, jak už bylo zmíněno, informace, které službu popisují a bude diskutován později.

```
package net.jini.core.lookup;

public Class ServiceTemplate {

    ServiceTemplate(ServiceID serviceID,
                    java.lang.Class[] serviceTypes,
                    Entry[] attrSetTemplates);
}
```

ServiceMatches - metoda *lookup()* je přetížena. Kromě typu služby, kterou chceme vyhledat, můžeme také specifikovat, kolik nalezených odpovídajících služeb má být maximálně nalezeno. Pokud je k dispozici například 10 tiskáren, můžeme specifikovat, že chceme najít maximálně 4.

Vyhledání příslušné služby a navázání komunikace mezi službou a klientem je graficky znázorněno na obrázcích 2.3 a 2.4 .





2.3 Entry

Vyhledává-li klient službu, je mu známo rozhraní, které služba implementuje a podle toho ji vyhledává. Služba ovšem může poskytovat také sadu atributů *Entry*, které jí dodatečně specifikují.

Existují-li například dvě služby pro převod měny, kterými jsou *EuroConverterService* a *DolarConverterService* a které implementují rozhraní *Converter*, mohu službu převádějící na Eura vyhledat následujícímy dvěma způsoby:

- Klient požádá o instanci třídy EuroConverterService
- Klient požádá o instanci třídy Converter, která umí převádět eura

Abychom mohli vyhledávat způsobem, který je uveden jako druhý, zavedeme třídu Currency, která určuje, jakou měnu je služba schopna převést. Tato třída implementuje rozhraní Entry, jak ukazuje zdrojový kód 2.9 . Službu pak specifikujeme pomocí pole atributů, které předáme jako atribut konstruktoru ServiceItem.

```
public class Currency implements Entry {
    public String type;     // Currency type

    public Currency(String type) {
       this.type = type;
    }
}
```

2.3.1 Omezení

Stejně jako ServiceItem tak i Entry objekt je při registraci serializován. Při porovnání, zda-li požadovaná služba odpovídá vyhledané službě, jsou porovnány Entry objekty v jejich serializované podobě. Entry nemůže být tedy primitivních typů, například *int* nebo *char*. Jestliže chceme použít některý z primitivních typů, musíme je „zabalit" do třídy, v našem případě *Integer* nebo *Character*. Atributy entries musí být *public*, *non-static* a *non-final*.

2.3.2 Podtřídy třídy Entry

Implementace od společnosti Sun Microsystems obsahuje v balíku *jini-ext.jar* užitečné podtřídy třídy *Entry*.

- *Address* - adresa fyzické komponenty nebo služby.
- *Comment* - komentář k službě.
- *Location* - adresa fyzické komponenty nebo služby. Rozdíl od Address je v tom, že Location se používá pro lokalizaci lokální organizační jednotky (jméno budovy, číslo patra, číslo kanceláře)
- *Name* - Jméno služby. Služba může mít více jmen.
- *ServiceInfo* - základní informace o službě. Zahrnuje jméno výrobce, produktu a prodejce.
- *ServiceType* - Typ služby popsaný "lidským" způsobem.

- *Status* - rodičovská třída pro jiné třídy určující stav služby.

2.4 Leasing

Leasing je systém, jak udržovat přehled o tom, které služby jsou stále aktivní, které se odpojily nebo u kterých nastala libovolná chyba a přestaly fungovat. Při volání metody *register()* na objektu *ServiceRegistrar* předáváme jako parametr dobu, po kterou chceme, aby byla služba dostupná. To je možné učinit několika způsoby:

- *Lease.ANY* - správce služby nechává vyhledávací službu rozhodnout, na jak dlouho ji předá pronájem.
- *Lease.FOREVER* - požadavek, aby pronájem nikdy nevypršel.
- Číslo ve tvaru *long*, které udává čas v milisekundách

2.4.1 Lease

Poskytovatelem pronájmu je Lookup service, která rozhoduje, na jak dlouho pronájem přidělí. Je tedy třeba si po úspěšné registraci na ní vyžádat objekt typu *Lease* (viz Zdrojový kód 2.10), který poskytuje metody ke zjištění doby vypršení leasingu.

```
package net.jini.core;

public interface Lease {
     void cancel() throws UnknownLeaseException,
                          java.rmi.RemoteException;
     long getExpiration();
     void renew(long duration) throws LeaseDeniedException,
                                      UnknownLeaseException,
                                      java.rmi.RemoteException;
     ...
}
```

Zavoláním metody *cancel()* na objektu *Lease* může služba zrušit svůj leasing. Objekt *Lease* komunikuje s lease managementem u LUS kde leasing zruší.

Vyprší-li pronájem, Lookup service to nijak neoznámí, je tedy na správci služby, aby se staral o jeho obnovování pomocí metody *renew()*, kterou poskytuje objekt *Lease*

2.4.2 LeaseRenewalManager

Jini poskytuje třídu *LeaseRenewalManager* (viz Zdrojový kód 2.11), která se stará o správu leasingu voláním metody *renew()*.

```
package net.jini.lease;

public Class LeaseRenewalManager {

    public LeaseRenewalManager();
    public LeaseRenewalManager(Lease lease, long expiration,
                               LeaseListener listener);
    public void renewFor(Lease lease, long duration,
                         LeaseListener listener);
    public void renewUntil(Lease lease, long expiration,
                           LeaseListener listener);
    ...
}
```

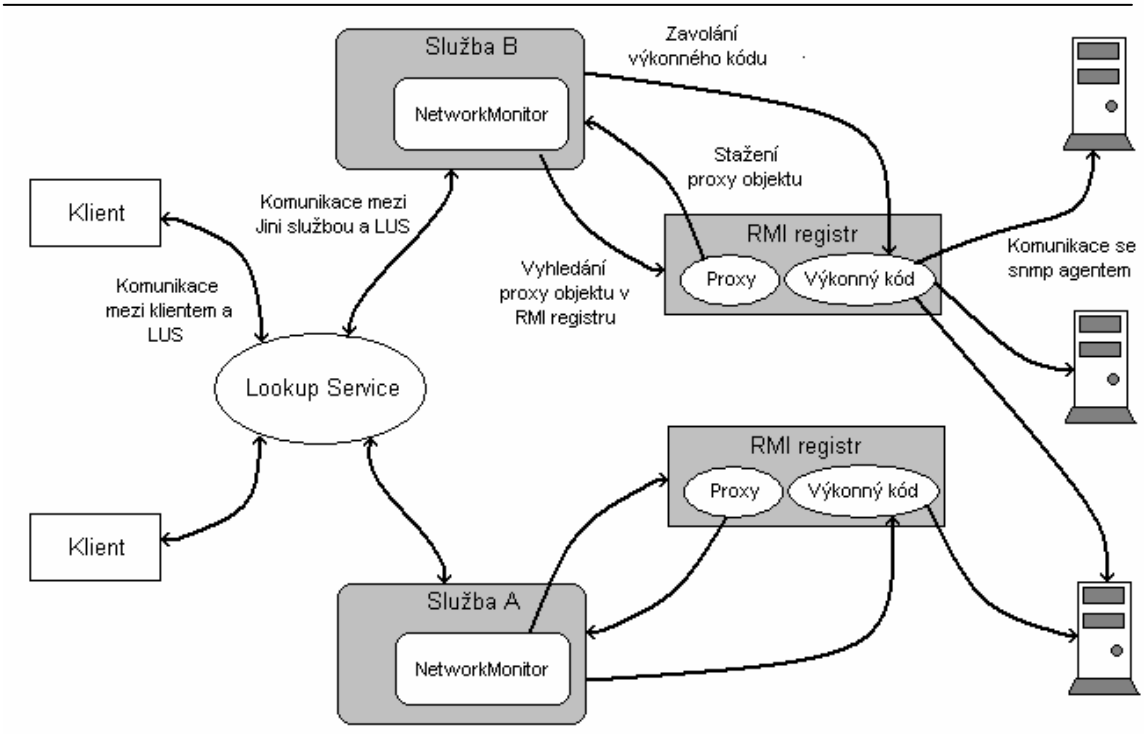
Prvním parametrem konstruktoru této třídy je pronájem, který chceme manažovat. Druhý parametr je požadovaná doba pronájmu klientem. Nejedná se tedy o dobu, která je garantována vyhledávací službou v pronájmu (*lease.getExpiration()*), ale o dobu, na kterou bychom rádi pronájem získali. Třetím parametrem je posluchač, který obdrží oznámení, když dojde k jakékoliv výjimce při pokusu o obnovení pronájmu.

3 Realizační část

Tato kapitola se bude zabývat implementací monitorovacího systému. Konkrétně Jini služeb pro monitorování sítě a klientské aplikace, která tyto služby využívá. Na úplném začátku probereme také návrh celého systému včetně částí, které nejsou řešeny v rámci této práce, ale jsou jeho nezbytnou součástí.

3.1 Návrh monitorovacího systému

Celý monitorovací systém se skládá z několika nezbytných součástí (viz obrázek 3.1), bez kterých by se komunikace mezi klientem a koncovou monitorovanou stanicí neobešla. V této části si všechny součásti stručně popíšeme, v dalších částech pak rozebereme ty, které jsou předmětem této práce podrobněji.



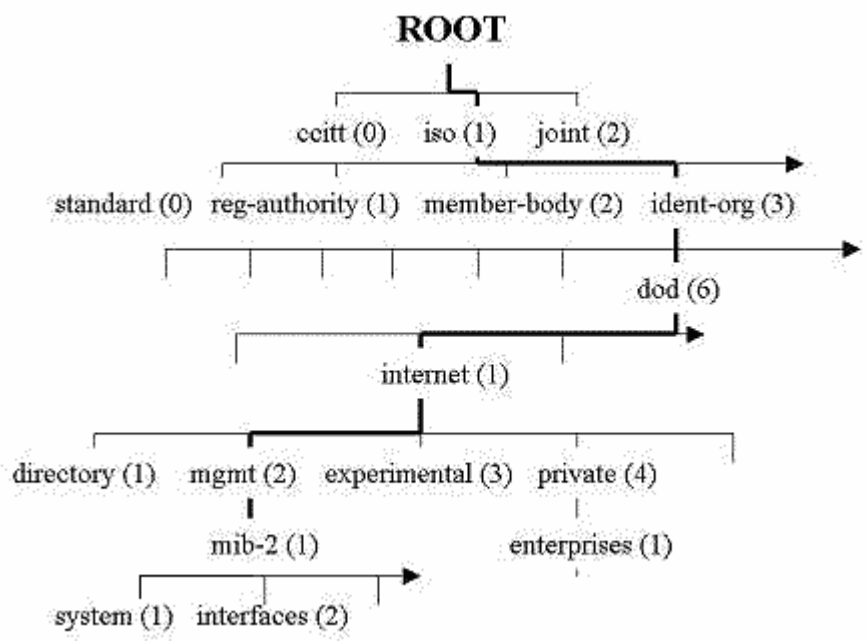
SNMP protokol – K samotnému monitorování jednotlivých stanic systém používá protokol SNMP (Simple Network Management Protocol). Tento protokol je založen na modelu klient/server, kdy klient komunikuje se serverem v podobě Snmp agenta na monitorovaném stroji. Informace, poskytované agentem, jsou uspořádány v databázi MIB (Management Information Base). Každý agent, vztaženo na různé druhy zařízení, má rozdílnou strukturu MIB. Tato struktura je definována jako strom s určitou hierarchií. Tento strom má hierarchii, jakou můžeme vidět například na obrázku 3.2 . Každou položku, která je v MIB obsažena, lze jednoznačně popsat pomocí odkazů na jednotlivých úrovních. Tento popis se nazývá "object identfier", neboli identifikátor objektu a v literatuře je označován zkratkou OID. Například klientská aplikace, která byla vytvořena v rámci této práce, umožňuje monitorovat vybrané informace z větve 1.2.3.6.1.2.1., jak je vyznačeno na obrázku 3.2 .

Java RMI – Komunikace mezi SNMP monitorovací stanicí a Jini službami, které poskytují informace klientské aplikaci, se provádí pomocí Java RMI. Objekty, které poskytuje monitorovací stanice k získání informací z databáze MIB, jsou vloženy do RMI registrů, kde jsou k dispozici Jini službám, které si je mohou stáhnout a používat jejich metody k získání potřebných informací o monitorované stanici.

Rozhraní – Pro získání objektů z RMI registrů používají Jini služby třídu *NetworkMonitor*. Tato třída je jakýmsi rozhraním mezi touto bakalářskou prací a prací na téma *RMI rozhraní pro monitorování počítačové sítě pomocí protokolu SNMP*. Funkcí této třídy se budeme zabývat v oddílu 3.2 .

Jini služby - Součástí systému je množina služeb založených na Jini technologii. Tyto služby se po spuštění přihlásí do Jini systému a jsou k dispozici klientské aplikaci, která je může využít k monitorování sítě.

Klient – Aplikace s grafickým uživatelským rozhraním, která umožňuje monitorovat vybrané základní informace o síti (např. vyhledání strojů na určitém segmentu sítě, popis systému, …). I klientská aplikace je navržena jako Jini entita, která se po spuštění připojí do Jini sítě a umožní uživateli monitorovat síť díky komunikaci s dostupnými službami.



3.2 Třída NetworkMonitor

Pro komunikaci mezi Jini službami a RMI registry bylo třeba vytvořit kvalitní rozhraní, které umožní komunikaci mezi těmito dvěma částmi.

Hlavní požadavky byly, aby se rozhraní nemuselo jakkoliv měnit, naprogramuje-li se nová služba, která bude chtít volat nově vytvořenou akci nahranou do RMI registru. Dalším požadavkem bylo, aby se na straně Jini služby nemuselo nijak pracovat s RMI. Tedy aby si programátor vytvořil objekt tak, jak je zvyklý z programování běžných java aplikací a nemusel se zajímat o to, jak funguje java RMI a co tato technologie přináší za problémy. Řešením se ukázala třída *NetworkMonitor,* jejíž využití si popíšeme v této části. Detailní implementací se zde zabývat nebudeme, neboť ta je předmětem bakalářské práce *RMI rozhraní pro monitorování počítačové sítě pomocí protokolu SNMP*.

Pro získání reference na konkrétní objekty uložené v RMI registru stačí Jini službě zavolat statickou metodu *getMonitorAction()* třídy *NetworkMonitor,* jíž předá jako parametr název příslušné akce, kterou v registru vyhledává (viz. Zdrojový kód 3.1).

```
public HashMap<String,String> getServerList(String startIp,
                                            String endIp){
     HashMap<String,String> serverList = null;
     SnmpLookupAction snmpLookup = (SnmpLookupAction)
     NetworkMonitor.getMonitorService("SnmpLookupAction");

     try{
          serverList = snmpLookup.getSnmpServersList(startIp,
                                                     endIp);
     }
     catch(RemoteException e){
          System.out.println("Chyba ve službě NetworkScan: \n" +
                             e.getMessage());
     }
     return serverList;
}
```

Aby třída *NetwokMonitor* věděla, ve kterém registru má příslušnou akci hledat, je v adresáři *conf* umístěn soubor *config.xml* (viz. Zdrojový kód 3.2), ve kterém se tyto informace nastavují.

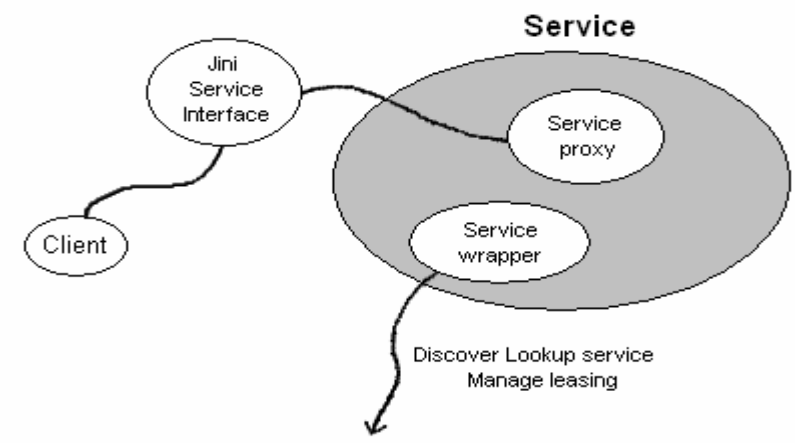
Opět zde nebudeme popisovat, jak třída získává informace z tohoto souboru, protože to není předmětem této práce. Pouze si popíšeme, k čemu slouží který element.

- Element *actionsLoader* obsahuje název třídy, která je použita pro vytvoření objektu Actions loaderu a uživatel ho v praxi bude měnit, pouze pokud vytvoří svůj vlastní loader.
- Dále jsou zde elementy *action*. První má parametr *name* nastaven na *default*. Tento element určuje defaultní adresu, kde se bude hledat umístěni RMI registru, pokud není definován žádný element *action* se jménem požadované akce. V opačném případě se bude RMI registr hledat v takto definovaném umístění.

```
<networkMonitor>
   <client>
      <actionsLoader name="RMIActionsLoader"/>
         <actions>
            <action name="default" host="localhost"/>
            <action name="SnmpLookupAction" host="localhost"/>
         </actions>
   </client>
</networkMonitor>
```

3.3 Obecná struktura Jini aplikace

Dříve, než se pustíme do popisu naprogramované aplikace, rozebereme, jak vypadá základní struktura Jini aplikací, kterou vidíme zobrazenou na obrázku 3.3 .



Client – Klientská aplikace se z hlediska Jini architektury zpravidla skládá pouze z jednoho objektu, kterým je posluchač. Ten slouží k zachycení oznámení od nově spuštěných Lookup service.

Rozhraní služby – Tato část by měla být známa v době implementace klienta. Bývá hlavní položkou vyhledávacího dotazu, kterým klient specifikuje požadovanou službu, která tato rozhraní implementuje.

Hlavní třída služby – Service wrapper je ta část služby, která vytváří objekt, jenž poskytuje vlastní funkce služby. Tato třída řídí zapojení Jini služby do Jini síťe, udržuje pronájem registrace a stará se o zveřejnění proxy objektu u lookup service.

Service proxy - Je uveřejněn pro stažení případným zájemcům o službu. Na klientské straně funguje jako ovladač pro komunikaci se vzdálenou službou. Proxy objekt implementuje metody uvedené v Jini service interface.

3.4 Množina monitorovacích služeb

Tato část práce popisuje množinu Jini služeb, které byly navrženy pro potřeby monitorování síťe v rámci této práce. Dále se seznámíme s implementací těchto služeb a na příkladech si ukážeme jejich zapojení do Jini síťe. Tyto služby se nacházejí v balíku *cz.hruby.services.** v archivu jini-netmon.jar .

3.4.1 Popis služeb

NetworkScan – Tato služba poskytuje základní funkce pro monitorování sítě. Díky metodě *getServerList(String startIp, String endIp)* umožňuje prohledat segment sítě. Tato metoda má návratovou hodnotu *HashMap<String, String>*, kde klíč je IP adresa stroje a hodnotou je název stroje. Dále služba poskytuje metodu *getSingleServer(String ip, int port),* pomocí které lze nalézt stroj s konkrétní ip adresou na konkretním portu, na kterém běží snmp agent. Tato služba má návratovou hodnotu *String*, která obsahuje jméno stroje. IP adresa je již známa, neboť je předávána jako parametr metodě.

CpuLoad – Tato služba monitoruje výkon procesoru konkrétního stroje, na který se může klient dotázat zavoláním metody *getCpuLoad(String ip, int port)* . Návratová hodnota obsahuje průměrný výkon procesoru za poslední minutu.

SystemDescription – Tato služba umožňuje získat následující popisující informace o monitorovaném stroji:

- Metoda *getSystemDescription(String ip, int port)*
  poskytuje základní údaje o systému, jako je informace o

operační systému, popis architektury a další, které jsou obsaženy v návratové hodnotě *String*.

- Metoda *getMemSize(String ip, int port)* navrací velikost operační paměti.
- Metoda *getInterfacesSpeed(String ip, int port)* navrací hodnotu typu *HashMap<String, Integer >*, kde klíčem je název síťového rozhraní a hodnotou je rychlost tohoto rozhraní.

SystemUpTime – Tato služba umožňuje monitorovat dobu běhu stroje voláním metody *getSystemTimeUp(String ip, int port)*.

3.4.2 Struktura služby

Struktura všech výše popsaných služeb je stejná. Proto mechanismy, které budu prezentovat na službě *NetworkScan,* platí beze zbytku i pro všechny ostatní poskytované služby. Části kódu, které zde budou uvedeny, jsou stejně jako všechny další v této práci zkráceny tak, aby ukazovaly pouze relevantní části vzhledem k probírané problematice.

Implementace služby - Každá služba se skládá ze dvou tříd. Jednou z nich je implementační třída. Struktura této třídy je vcelku jednoduchá. Jediným požadavkem na tuto třídu je implementace dvou rozhraní. Za prvé musí třída implementovat rozhraní *java.io.Serializable*, neboť instance této třídy je obalovací třídou nahrána k Lookup service a jak víme z teoretické části, takovýto objekt je před zasláním serializován

Za druhé musí třída implementovat tzv. well-known interface (v případě služby *NetworkScan* viz. zdrojový kód 3.3). To je rozhraní,

které je známo klientovi. Toto je nutné, pro snadné vyhledání služby. Tato rozhraní nalezneme v knihovně *service-interfaces.jar*

```
package cz.hruby.services.interfaces;

import java.util.HashMap;

public interface NetworkScanInterface {

    public HashMap<String,String> getServerList(String startIp
                                                ,String endIp);
    public String getSingleServer(String ip, int port);
}
```

Obalovací třída – Jak je popsáno v odstavci týkajícím se obecné architektury Jini aplikace, obalovací třída se stará o zapojení služby do Jini síťe a další náležitosti, které je potřeba splnit, aby byla služba plnohodnotnou Jini entitou.

V našem případě služba implementuje dvě rozhraní. Prvním je rozhraní *DiscoveryListener* . Služba po spuštění rozešle multicast paket a čeká, jestli se ozve některá Lookup service. Odezvy zachytává právě *DiscoveryListener* a v implementované metodě *discovered()* zaregistruje službu u Lookup service, která o sobě dala vědět. Proces registrace služby ukazuje zdrojový kód 3.4 .

Druhým rozhraním, které obalovací třída implementuje, je rozhraní *LeaseListener*. Jak vidíme ve zdrojovém kódu 3.4, po zaregistrování služby u Lookup service je pověřen objekt typu *LeaseRenewalManager* spravováním pronájmu služby zavoláním metody *renewUntil()*. Jako jeden z parametrů je této metodě předáván objekt typu *LeaseListener*. Toto rozhraní obsahuje metodu *notify()*, která je zavolána při jakémkoliv problému během pokusu o obnovení pronájmu.

```
    public void discovered(DiscoveryEvent evt){

        // získání objektů ServiceRegistrars pro komunikaci s LUS
        ServiceRegistrar[] registrars = evt.getRegistrars();

        for (int i = 0; i < registrars.length; i++) {
            ServiceRegistrar registrar = registrars[i];

            Entry[] entries = new Entry[] {new
                              Name("NetworkScan")};
            // vytvoření objektu ktery se bude registrovat u LUS
            // impl – implementace služby,
            //serviceID – identifikátor
            ServiceItem item = new ServiceItem(serviceID, impl,
                                               entries);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " +
                                   e.toString());
                continue;
            }
        System.out.println("Service registered with id " +
                           reg.getServiceID());

        // nastavení objektu typu LeaseManager pro správu leasingu
        leaseManager.renewUntil(reg.getLease(), Lease.FOREVER,
                                this);
        }
    }
```

Identifikátor – Proběhne-li úspěšně registrace u Lookup service, je služba zaregistrována pod unikátním 128 bitovým identifikátorem. Tento identifikátor přidělí službě Lookup service v okamžiku registrace, nebo se může služba pokusit registrovat pod konkretním *ServiceID*. Je dobrým zvykem, aby si služba po prvním přihlášení, kdy je jí identifikátor přidělen, tento identifikátor uložila a použila ho například při dalším přihlášení, neboť podle něho může být vyhledávána. Například služba *NetworkScan* si ukládá svůj identifikátor do souboru *NetworkScan.id* .

3.5 Klientská aplikace

Poslední částí celého systému je klientská aplikace. Tato aplikace umožňuje uživateli zobrazovat údaje o síti. Klient je rozdělen do dvou modulů. V hlavním okně aplikace může uživatel zobrazit stroje, které jsou zapojeny v síti. Dále je možno spustit ke každému zobrazenému stroji monitorovací okno, ve kterém lze zobrazit další údaje týkající se konkrétního stroje.

Client - Hlavní třídou klientské aplikace je třída *cz.hruby.client.Client.class* . Tato třída má dvojí funkčnost. Kromě vytvoření a zobrazení hlavního okna se stará o zapojení klientské aplikace do Jini síťe.

Stejně jako každá ze služeb i tato třída implementuje rozhraní *DiscoveryListener,* aby mohla zachytávat odpovědi od Lookup service, které zareagují na rozeslaný multicast paket. Aplikace zároveň umožňuje připojení k Lookup service pomocí *unicast discovery procesu*. Realizace nalezení Lookup service pomocí unicastu je vidět ze zdrojového kódu 3.5 .

```
LookupLocator locator = null;

String s = // ziskani adresy z InputDialog
try{
    locator = new LookupLocator("jini://"+s);
}catch(java.net.MalformedURLException e) {
    // vypis chyboveho hlaseni o spatnem formatu adresy
}
registrars = new ServiceRegistrar[1];

// získání objektu pro komunikaci s LUS
try{
    registrars[0] = locator.getRegistrar();
}catch (java.io.IOException e) {
    registrars = null;
    // vypis chyboveho hlaseni
} catch (java.lang.ClassNotFoundException e) {
    // vypis chyboveho hlaseni
    System.exit(1);
}
```

Třída *Client* zároveň poskytuje dvě základní metody:

- **scan()** – Metoda pro prohledání zadaného segmentu sítě. Tato metoda vyhledá u Lookup service službu (viz. zdrojový kód 3.6), která se stará o prohledání sítě a pomocí metody *getServerList()* objektu, který služba nahrála k Lookup service, nalezne všechny SNMP agenty běžící na strojích daného segmentu sítě. Názvy strojů a jejich IP adresu zobrazí v hlavním okně aplikace.
- **add()** – Metoda pro přidání jednoho stroje. Tato metoda vyhledá u Lookup service stejný objekt jako metoda *scan(),* ale zavolá na něm metodu *getSingleServer()*. Tato metoda vrátí *null,* pokud stroj se zadanou adresou neexistuje nebo na něm neběží SNMP agent. V opačném případě vrátí metoda jméno stroje.

```
public void scan(){
     snmpServers = new HashMap<String, String>();
     // získání třídy kterou budeme vyhledávat u LUS
     Class [] lookupClasses = new Class[]
                              {NetworkScanInterface.class};
     // další atribut pro vyhledání služby je jméno služby
     Entry[] lookupEntries = new Entry[]
                              {new Name("NetworkScan")};

     NetworkScanInterface networkScan = null;

     // vytvoření vzoru pro vyhledání služby
     ServiceTemplate lookupTemplate = new ServiceTemplate(null,
                                      lookupClasses,
                                      lookupEntries);
     ServiceRegistrar registrar = registrars[0];
     try {
          // samotné vyhledání služby
          networkScan = (NetworkScanInterface)
                        registrar.lookup(lookupTemplate);
     } catch(java.rmi.RemoteException e){
          e.printStackTrace();
          continue;
     }
     // zde bychom měli získat IP adresy z GUI

     // zavolání metody na objektu služby který jsme získali
     snmpServers = networkScan.getServerList(startIP, endIP);
}
```

MonitoringModul – Druhou zásadní třídou klientské aplikace je třída *MonitoringModul*. Tato třída vytváří grafické uživatelské rozhraní pro monitorování konkrétního stroje a poskytuje následující metody k uskutečňení tohoto monitorování:

- **showCpuLoad()** – Tato metoda vyhledá službu pro monitorování vytížení procesoru a vytvoří instanci třídy *Timer*. Tento se následně stará o volání metody *getCpuLoad()* na objektu získaném od Lookup service. Toto volání probíhá každou minutu, neboť minuta je interval, po kterém se mění údaje o průměrném vytížení procesoru v MIB tabulce SNMP agenta. Vytvoření timeru je v tomto případě důležité, neboť časovač běží ve

vlastním vlákně a je tedy možné monitorovat více strojů najednou.

- **showDesc()** – Metoda *showDesc()* má za úkol vyhledat službu, která poskytuje popisující informace o monitorovaném stroji a tyto informace zobrazit.
- **showSystemUpTime()** – Tato metoda vyhledá příslušnou službu a zobrazí na panelu čas, po který již běží monitorovaný stroj.

Zobrazení dat – Klientská aplikace musí data získaná od Lookup service zobrazovat. V našem případě třídy, které se starají o zobrazení dat implementují jednoduchá rozhraní, která obsahují vždy jednu metodu (např. třída *SystemDescriptionPanel* implementuje metodu *writeDesription()* z rozhraní *SystemDescriptionViewer*). Implementace této metody má za úkol reprezentovat získaná data u klientské aplikace. Jestli bude aplikace údaje zobrazovat graficky nebo na příkazové řádce je pouze na programátorovi.

4 Hodnocení

K otestování implementovaného systému byl použit běžný domácí počítač, na kterém běžela lookup service a Jini služby. RMI registry byly spuštěny na počítači zapojeném v domácí síti kolegy, který implementoval druhou polovinu systému, taktéž na běžném domácím počítači. Systém spolehlivě monitoroval všechny implementované funkce, které konkrétní zařízení podporovala.

Tuto práci můžeme hodnotit ze dvou podhledů. Jako systém sloužící k monitorování sítě a jako prostředek pro otestování funkčnosti a flexibility Jini technologie.

Z pohledu monitorování sítě musíme říci, že systém má svoje výhody i svoje nevýhody. Výhodou je jistě možnost lehce doprogramovat další služby, pokud jich není pro potřeby správce sítě dostatek. Z toho důvodu aplikace obsahuje pouze několik služeb, které demonstrují funkci celého systému. Nevýhodu, kterou spatřujeme, je množství prvků, přes které je vedena komunikace od klientské aplikace k monitorovanému počítači. Klient musí vyhledat službu u lookup service, následně na získaném objektu volat metodu, která zjistí ve kterém RMI registru se nachází objekt, který poskytuje požadovanou akci. Tento proxy objekt z registru stáhnout a pomocí něho komunikovat s výkonným kódem, který je v registru umístěn. Tento kód teprve komunikuje přímo se strojem, na němž běží SNMP agent. Tento systém přináší na jednu stranu pozitiva, neboť ten, kdo programuje například Jini službu, nemusí vědět nic o ostatních částech. Pouze si zavolá statickou metodu *getMonitorService()* třídy *NetworkMonitor*, která vrátí příslušný objekt, přes který služba dále komunikuje, aniž by se programátor musel starat o to, jak komunikace probíhá. Na druhou stranu, čím více prvků, tím více komunikace mezi každými dvěma sousedními prvky, která

samozřejmě zpomaluje tok dat od koncového prvku, kterým je SNMP agent, ke klientské aplikaci. Z testování aplikace, ale vyplynulo, že tato komunikace není natolik zdlouhavá aby nějak výrazně ovlivnila její chod.

Z hlediska otestování funkčnosti, užitečnosti a obtížnosti využití Jini technologie posloužila tato práce dostatečně. Implementovat základní vlastnosti Jini entit jako je zapojení do Jini síťe, jak multicast discovery procesem, tak unicast discovery procesem, registrace služeb u lookup service, vyhledání služeb klientem a obsluha leasingu, je poměrně jednoduché. V tomto projektu služby vystupují pouze jako poskytovatel objektu, přes který klient získává informace o monitorované síti. Nicméně tato technologie má podle mého mínění mnohem větší potenciál, než jaký byl využit v tomto projektu. Zejména ve spojení s rozličným hardwarovým vybavením by se Jini dalo využít k zajímavým projektům, kde by se více využila hlavní přednost, kterou je adaptibilita na změny v síti. Například představa domácí Jini sítě, kde si domácí spotřebiče oznamují navzájem různé informace (na televizi se objeví oznámení o nově příchozím e-mailu, nebo o tom, že jídlo je v troubě již požadovanou dobu), je velice lákavá. Taková síť by neměla na uživatele žádné nároky, kromě úvodního nastavení, neboť všechna zařízení by se ohlásila navzájem po spuštění a pak už by si pouze poskytovaly příslušné služby.

5 Závěr

Tato práce měla za cíl zjistit, jaké možnosti poskytuje Jini technologie a tyto poznatky demonstrovat vytvořením monitorovacího systému. V první části rozebírám po teoretické stránce jak fungují všechny základní důležité procesy v Jini systému. Ve druhé části ukazuji na implementaci služeb pro monitorování počítačové sítě jak všechny procesy popsané v první části použít v praxi. Realizační část obsahuje několik užitečných příkladů zdrojového kódu, které by, v pozměněné formě, neměly chybět v žádné správně napsané Jini aplikaci.

Kapitola 4 obsahuje zhodnocení práce. V této části shrnuji klady i zápory implementovaného systému. Najdete zde i odstavec, který se zabývá užitečností Jini technologie a můj názor na to, jaký potenciál tato technologie skrývá. Práce splňuje zadání ve všech bodech.