Technische Universität München

Lehrstuhl für technische Informatik - Rechnernetze
Prof. Dr. Hegering

Fortgeschrittenen-Praktikum

Realisierung eines Managementwerkzeugs in Java für das Management von IP-Netzen

Bernhard Kempter

10. November 1997

Inhaltsverzeichnis

1 Einführung

Durch die stetig wachsende Zahl von Computer-Netzwerken wird das Management solcher Netze immer schwieriger — aber auch wichtiger. Ein Problem in großen Netzen sind sogenannte "Eventstorms". Dabei handelt es sich um eine Flut von Fehlermeldungen, die beim Systemadministrator auflaufen, wenn z.B ein Router oder ein Webserver ausfällt. Es ist nun sehr schwierig, aus der Vielzahl von Meldungen die tatsächliche Fehlerursache herauszufinden. Ein **Eventkorrelator** soll bei dieser Aufgabe helfen, indem eine Vorverarbeitung der Fehlermeldungen stattfindet und nur noch die relevanten Meldungen beim Systemadministrator ankommen. Um dies zu bewerkstelligen, muß der Eventkorrelator die Topologie des Netzes kennen. Damit können dann Abhängigkeiten von Fehlermeldungen zueinander festgestellt werden.

1.1 Aufgabenstellung

In diesem Fortgeschrittenen-Praktikum (Fopra) wurde der Aufbau eines Abhängigkeitsgraphen implementiert. Dabei wird von einer gegebenen Management-Umgebung ausgegangen, für die dieser Graph erstellt wird. Die Abbildung 1 zeigt die Topologie eines solchen Netzes aus Sicht der IP-Ebene. Das Netz besteht aus Hosts mit mindestens je einem Interface sowie Routern. An diesem Abhängigkeitsgraphen kann man ablesen, über welche Komponenten ein IP-Paket, z.B. von Host A nach Host C gesendet wird (gepunktete Linie in der Abbildung). Daraus folgt, daß neben Komponenten des Netzes der Abhängigkeitsgraph auch Routing-Information enthalten muß.

1.2 FMA-Architektur

Das Fopra setzt auf den **flexiblen Management-Agenten (FMA)** auf. Dieser FMA wurde von Markus Wennrich und Alexander Hollerith im Rahmen ihrer Diplomarbeiten ([Wen97],[Hol97]) implementiert, welches im Wesentlichen eine Erweiterung des **Java Agent Template (JAT)** ([Fro97]) ist. Die FMA-Architektur eignet sich besonders für die gestellten Aufgabe, da man beliebig viele Agenten, die sich an einer beliebigen Stelle im zu managenden Netz befinden können, für die Problemlösung verwenden kann. Die wichtigsten Eigenschaften der FMA-Architektur werden nun vorgestellt.

Die Architektur ist verteilt, d.h. die FMAs befinden sich auf verschiedenen Hosts in einem Netz (Abbildung 2). Der FMA stützt sich auf die Java Virtual Machine (JVM) ab. Jeder FMA muß sich und seine Interpreter beim Starten bei einem ausgezeichneten Interpreter, dem Agent-Name-Server (ANS) anmelden. Der ANS ist also, wie der DNS im Internet, der Namensdienst, also

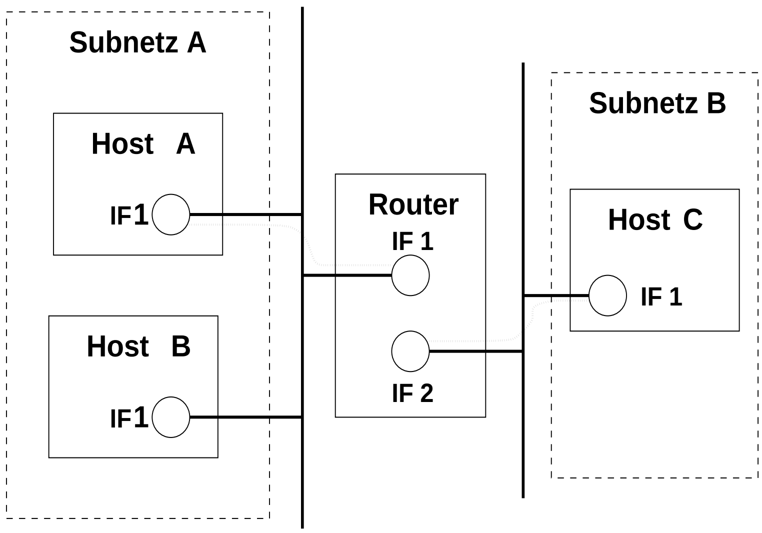

Abbildung 1: Paketverlauf über IP-Ebene

für die Adressauflösung zuständig. Beim Agent-Event-Server (AES) kann sich ein Interpreter anmelden, damit er benachrichtigt wird, wenn ein bestimmter Event auftritt. Jeder FMA gehört einer (FMA)-Domäne an. Pro Domäne muß mindestens ein ANS vorhanden sein. Jeder FMA kann beliebig viele Interpreter hochfahren und verwalten. Der Agent stellt die Grundfunktionalität wie Senden und Empfangen von Nachrichten zur Verfügung. Ein Interpreter ist für eine ganz bestimmte Managementaufgabe, z.B. für die Überwachung eines Druckers zuständig. Der Interpreter muß alle Nachrichten des Druckers verstehen und darauf reagieren können.

Die Kommunikation zwischen den Agenten wird über die **Knowledge Query and Manipulation Language (KQML)** abgewickelt. Diese Sprache wurde speziell für flexible Agenten entwickelt. Eine Nachricht besteht aus Schlüssel-Werte-Paaren und wird nur mit ASCII-Zeichen kodiert. Hier ein Beispiel einer KQML-Anfrage:

```
(ask-if  :sender    AIS
         :receiver  AHS
         :language  KQML
         :ontology  -
         :content   ( ask-resource  :type  address
                                    :name  AHS    ))
```

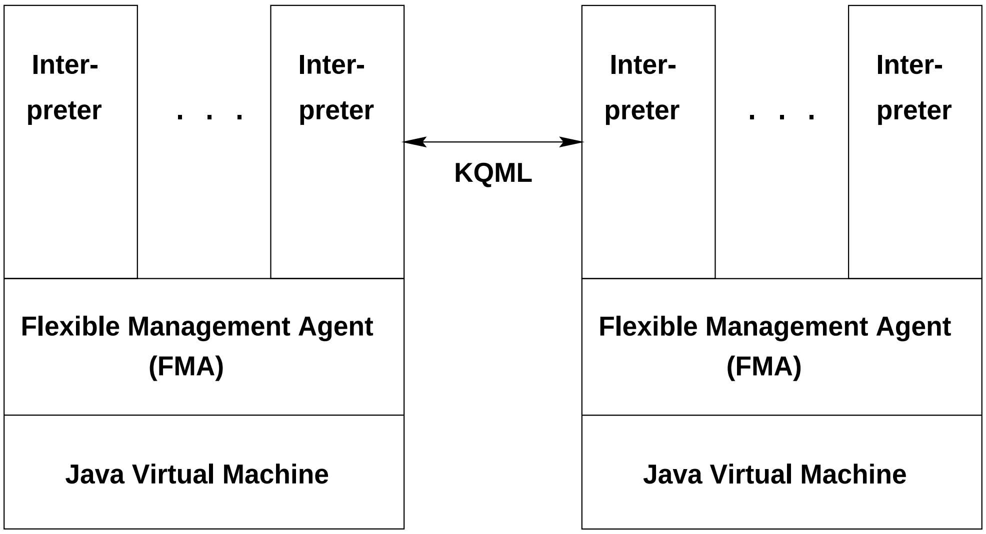

Abbildung 2: FMA-Architektur

1.3 Änderungen

Folgende Änderungen wurden an der Implementierung des FMA vorgenommen:

- Bei der Portierung von `java1.0` nach `java1.1` mußte das Event-Modell umgestellt werden.

  Betroffene Klassen:

  ```
  Agent/Agent.java
  Agent/AgentContext.java
  Agent/AgentFrame.java
  Agent/ComposeMessagePanel.java
  Agent/LoadMessagePanel.java
  Agent/LoadResourcePanel.java
  Agent/NetworkClassLoader.java
  Agent/SocketInterface.java
  Agent/SystemMessagesPanel.java
  Agent/ViewResourcePanel.java
  Interpreter_shared/AInterpreter.java
  Interpreter_shared/visualAlert.java
  ```

  Die Orginalzeilen bleiben auskommentiert im Quelltext stehen und sind durch den Zusatz "deprecated JDK1.1"gekennzeichnet.

2 Architektur

Die Voraussetzungen für die in diesem Kapitel gewählte Architektur sind:

- das zu managende Netz muß NIS verwenden und
- auf allen Hosts und Routern muß ein SNMP-Agent laufen, sowie
- auf den Hosts, auf den ein Agent laufen soll, die Java Virtual Machine (JVM) ausführbar sein.

In den weiteren Kapiteln werden die Voraussetzungen näher erläutert.

2.1 Design-Entscheidungen

Die Architektur der Implementierung richtet sich u.a. nach den benötigten Informationen bzw. den Informationsquellen. Die wichtigsten Informationen für den Abhängigkeitsgraphen sind:

- Hostnamen: dienen zur deren Identifizierung und werden aus dem Network Information Service (NIS) ausgelesen. Das NIS [?] ist eine zentrale Datenbank in einem Netz, indem Systeminformation wie z.B Hostnamen und Paßwörter gespeichert werden.
- IP-Adressen der Interfaces: dienen zur deren Identifizierung und werden aus der Management-Information-Base-II (mib-2) des Internet-Management [Ros94] ausgelesen.
- IP-Routing-Tabellen: dienen zur Entscheidung an welches Interface ein IP-Paket geschickt, und von welchem es abgesendet wird. Diese Tabellen werden ebenfalls aus der mib-2 ausgelesen.

Aus dieser Aufteilung folgt die Unterteilung in Module, die diese Informationen zur Verfügung stellen:

- **Host-Server** erzeugt eine Liste mit Hosts.
- **Interface-Server** erzeugt eine Liste der Interfaces (abh. von der Host-Liste).
- **IP-Routing-Server** erzeugt eine Liste der IP-Routing-Tabellen (abh. von der Interface-Liste).

Durch diese Unterteilung erhält man kleine, überschaubare Module.
Das KQML-Protokoll ist für die Übertragung von umfangreichen Objekt-Listen nicht besonders gut geeignet, denn eine Liste müßte zuerst in eine ASCII-Zeichenkette umgewandelt werden, übertragen und anschließend vom Empfänger wieder zu einer Liste (von Objekten) zusammensetzt werden. Deshalb werden die Listen mit Hilfe der **Remote Method Invocation (RMI)** ([RMI]) übertragen. Dieser RPC-Mechanismus ist ab `java1.1` vorhanden. Ein kleines Beispiel soll die Programmierung von RMI erklären.

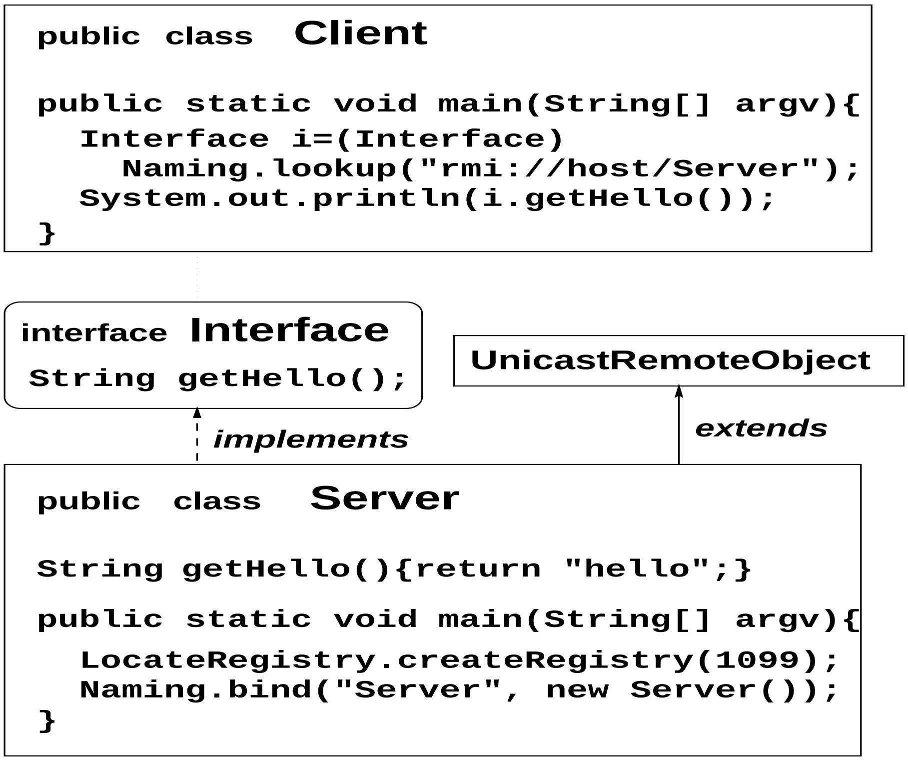

Abbildung 3: RMI - Beispiel

Wie in Abbildung 3 zu sehen, wird die Methode, die über RMI abgearbeitet werden soll, im `Interface` definiert. Diese Methode muß der `Server` implementieren. Die RMI-Registry muß gestartet werden. Eine Instanz des `Server`s und sein Name werden nun an die Registry gebunden. Der `Client` muß den Namen und den Host wissen, auf dem der Server läuft. Nun kann der `Client` über das `Interface` die Methode (`getHello()`) aufrufen, welche vom `Server` abgearbeitet wird.

Die Quelldateien werden mit dem javac-Kompiler übersetzt und anschließend der `Server` noch mit dem rmic-Kompiler, wobei die Klassen `Server_Stub` und `Server_Skel` generiert werden.

Da alle Interpreter von der Klasse `AInterpreter` und für RMI auch von der Klasse `UnicastRemoteObject` erben müßten, aber java nur Einfachvererbung bietet, ergibt sich eine weitere Unterteilung in Server und Interpreter. Der Interpreter bearbeitet die KQML-Anfragen und der (RMI-)Server generiert die jeweiligen Listen.

2.2 Modell

Aus dem letzten Abschnitt ergibt sich nun folgendes Modell:

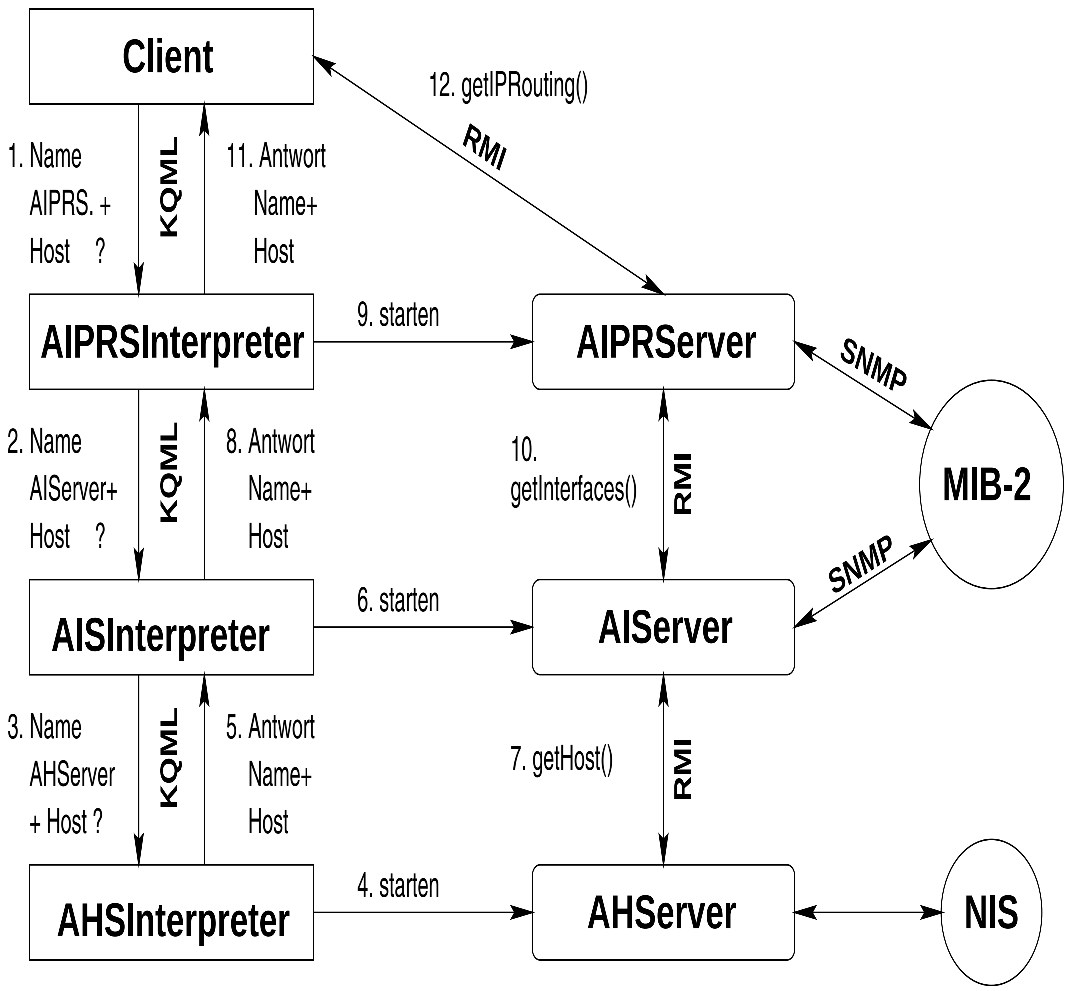

Abbildung 4: Interaktions-Modell

Ein Client, der (über RMI) eine Liste von einem der Server erhalten will, muß bei dem zuständigen Interpreter den Namen des Servers (über KQML) erfragen. Der Interpreter startet falls nötig den Server und gibt dem Client (über KQML) den Server und Host-Namen zurück. Nun kann der Client mit Hilfe der Server-Schnittstelle die Dienste des Servers in Anspruch nehmen. Die Funktionsweise der Interpreter und Server wird im nächsten Kapitel genauer erläutert. Hier nun die Liste der Abkürzungen:

- AHS: Agent-Host-Server
- AIS: Agent-Interface-Server
- AIPRS: Agent-IP-Routing-Server

3 Implementierung

Das Projekt wurde komplett mit JDK1.1.3 von Sun Microsystems auf dem SunOS 5.6-Betriebssystem implementiert.

Die in diesem Kapitel abgebildeten Klassenhierarchien sind wiefolgt zu verstehen:

- bei Schnittstellen sind die Ecken der Rechtecke abgerundet.
- Klassen haben "spitze"Ecken.
- die durchgezogenen Linien sind Vererbungslinien, die untere Klasse erbt von der oberen Klasse.
- ist der Klassenname *kursiv*, so ist diese Klasse abstrakt.
- gestrichelte Linien von einer Schnittstelle zu einer Klasse bedeutet, daß die Klasse die Schnittstelle implementiert.

3.1 Interpreter

Jeder Interpreter muß von der Klasse `AInterpreter` erben. Die Methoden `interpretMessage()`, `interpretLanguage()` und `run()` müssen überschrieben werden. Da alle Interpreter einen RMI-Server starten, wurde ein abstrakter `RMIInterpreter` implementiert, welcher die folgenden Aufgaben übernimmt:

- `createServerName()`: ein global eindeutiger Server-Name wird generieret. Es wird dazu der schon global eindeutige Name des Agenten (der den Interpreter startet) und die innerhalb des Agenten eindeutige `TaskID` verwendet.
- `createServer()`: SecurityManager und RMIRegistry (Port 1099) installieren, eine Instanz des Servers mit seinem Namen an die Registry binden.
- `interpretMessage()`: Weiterleitung der KQML-Nachricht, Aufruf von `interpretLanguage()` die die eigentliche Verarbeitung der Nachricht übernimmt.
- `run()`: Starten des Interperter als Thread.

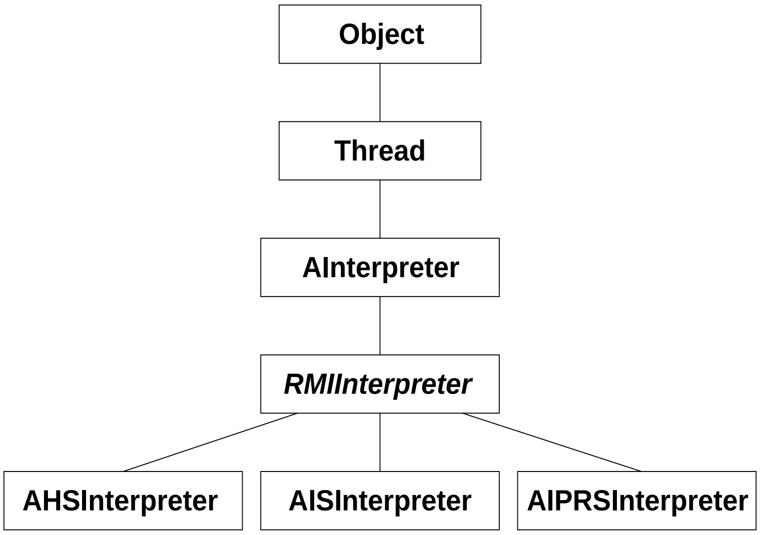

Abbildung 5: Klassenhierarchie der Interperter

Die Namensgebung der folgenden Klassen lehnt sich an den bereits implementierten Interpretern an. (Deshalb sind diese Abkürzungen gewählt worden). Die Klassen `AHSInterpreter` (Agent-Host-Server), `AISInterpreter` (Agent-Interface-Server) und `AIPRSInterpreter` (Agent-IP-Routing-Server) sind nun die konkreten Interpreter. Jeder Interpreter, der von `RMIInterpreter` abgeleitet ist, muß folgende Methoden überschreiben:

- `getServerObject()` instanziiert den dazugehörigen Server (der AHS-Interpreter instanziiert den AHServer usw.).
- `getNameForTask()` gibt den Namen des jeweiligen Interpreter an.

Die Kommunikation dieser Interpreter wird über die Klasse `RMIconnect` mit den Methoden `askRMI()` und `tellRMI()` durchgeführt. Dabei werden einfache `KQMLmessage`s generiert und vom FMA des Interpreters weiterverarbeitet.

3.2 Server

Alle Server-Klassen müssen vom `UnicastRemoteObject` erben, damit sie über RMI erreichbar sind. Genauso müssen sie eine Schnittstelle implementieren, dessen Methoden von einem Client aufgerufen werden können.

Die Klasse `AHServer` ermittelt alle Hosts des Netzes (Methode `getHosts()`). Die Hosts werden aus dem NIS mittels dem Konstrukt

```
Process process=
    (Process)runshell.exec(new String("/usr/bin/ypcat hosts"))
```

ausgelesen. Jeder Host kommt genau einmal in der `HostList` mit seinem Domain-Namen vor. Dabei wird der Domain-Name genommen, der mindestens einen Punkt enthält (z.B "hpheger8.nm.informatik.uni-muenchen.de"und nicht "hpheger8").

Die Klasse `AIServer` erweitert die `HostList` um die Interfaces zur `HostInterfaceList`. Dabei wird im Konstruktor `AIServer()` die Methode

```
AHServer.getHosts()
```

aufgerufen, um die `HostList` zu erhalten. Die Informationen für die Interfaces wird aus der Management- Information-Base (mib-2) ausgelesen. Dabei wird die Klasse `SNMPAccess` verwendet, die von Theodororos Kotselidis ([Kot97]) mit Abstützung auf das Advent Java SNMP Package Version 1.0.2-Paket ([Adv]) im Rahmen seines Fopras implementiert wurde. Die verwendeten Object-Ids sind:

- `interfaces.ifNumber` (Anzahl der Interfaces)
- `interfaces.ifTable.ifEntry.ifDescr` (Beschreibung)
- `ip.ipAddrTable.ipAddrEntry.ipAdEntAddr` (IP-Adresse)
- `ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex` (Zuordnung Interface - IP-Adresse)

Die Klasse `AIPRServer` erweitert die `HostInterfaceList` zur `HostIPRoutingList`. Im Konstruktor `AIPRServer` wird die Methode `AIServer.getInterfaces()` aufgerufen, um die `HostInterfaceList` zu erhalten.
Die benötigte Information für die IP-Routing-Tabelle wird wie vorher aus der mib-2 ausgelesen. Die verwendeten Object-Ids sind:

- `ip.ipRouteTable.ipRouteEntry.ipRouteNextHop` (Routing-Information)
- `ip.ipRouteTable.ipRouteEntry.ipRouteIfIndex` (Zuordnung Index - Interface)

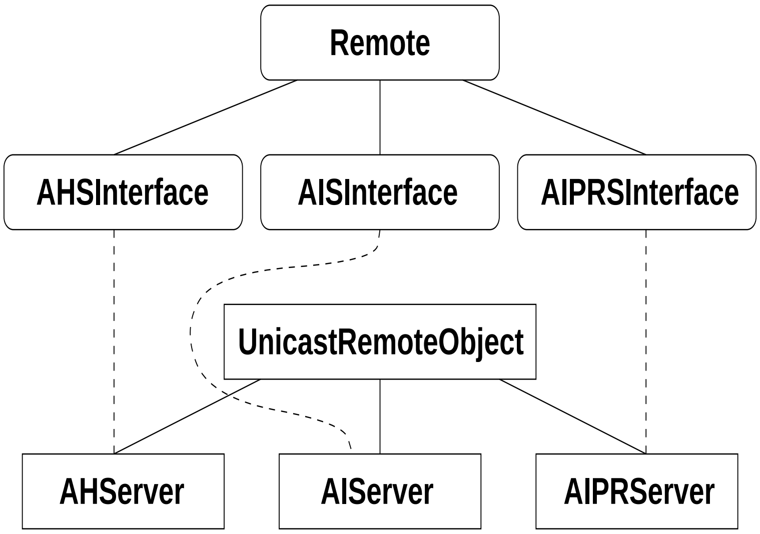

Abbildung 6: Klassenhierarchie der Server

3.3 Resultat-Listen

Die Resultat-Listen müssen so implementiert werden, daß sie jederzeit erweiterbar/reduzierbar sind. Die Klasse `java.util.Vector` besitzt genau diese Eigenschaft, sowie komfortable Such- und Einfügemethoden. Deshalb sind alle implementierten Listen von `java.util.Vector` abgeleitet. Zu jeder Listen-Klasse gehört eine Element-Klasse (z.B. HostList-HostElement). Jede nun beschriebene Klasse besitzt mehrere Konstruktoren und eine Methode zur String-Repräsentation der Klasse. Diese Methoden rufen jeweils die Methoden der übergeordneten Klasse auf.

Die Klasse `HostList` ist die Basis-Klasse für alle weiteren Listen. Die Elemente der `HostList` sind `HostElement`. Deshalb wurden vorallem Methoden hinzugefügt, die mit `HostElement` arbeiten. Ein `HostElement` besteht aus dem Domain-Name des Hosts. Schließlich bekommt man mit `getString()` eine String-Repräsentation der Klasse geliefert (der Standard-Name `toString()` konnte nicht verwendet werden, da dieser in `java.util.Vector` bereits als `final` deklariert ist).

Die Klasse `HostInterfaceList` erweitert die Klasse `HostList`. Der Konstruktor `HostInterfaceList(HostList hl)` wandelt die `HostList` in eine `HostInterfaceListe` um (und die `HostElement` in `HostInterfaceElement`). Ein `HostInterfaceElement` besteht aus der Anzahl der Interfaces und einer `InterfaceList`.

Eine `InterfaceList` hat als Einträge `InterfaceElement`, die wiederum

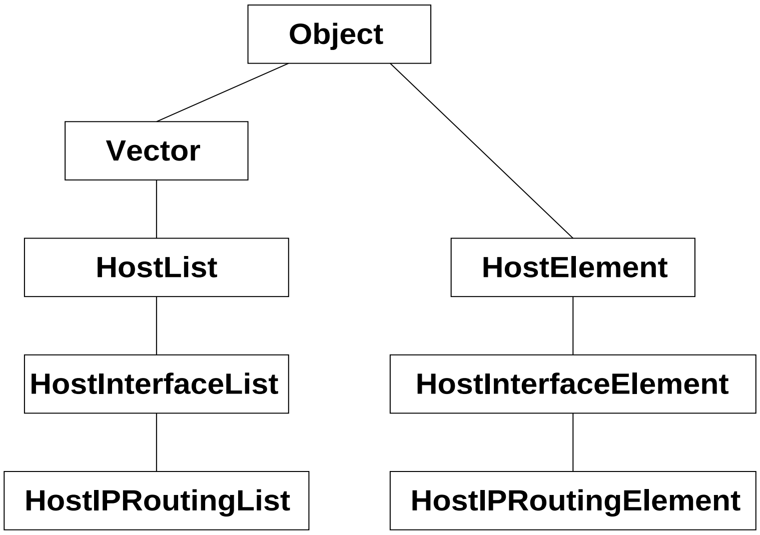

Abbildung 7: Klassenhierarchie der Resultat-Listen

aus der IP-Adresse und der Beschreibung des Interfaces besteht.

Die Klasse `HostIPRoutingList` fügt zur `HostInterfaceList` die IP-Routing-Tabelle (Klasse `IPRoutingList`) hinzu. Der Konstruktor

```
HostIPRoutingList(HostInterfaceList)
```

wandelt die `HostInterfaceList` in eine `HostIPRoutingList` um.

Eine `IPRoutingList` besteht aus `IPRoutingElementen`. Ein IP-Paket, das von einem Host gesendet werden soll, durchläuft solange der Reihe nach die `Pattern` einer jeden Zeile, bis ein `Pattern` paßt.

Am Beispiel der hpheger3.nm.informatik.uni-muenchen.de mit der IP-Adresse 129.187.214.23 (des Interfaces), wird das Pattern-Matching erklärt:

```
Pattern:129.187.214.23 NextHop:127.0.0.1       outon:127.0.0.1
Pattern:129.187.214.0  NextHop:129.187.214.23  outon:129.187.214.23
Pattern:127.0.0.1      NextHop:127.0.0.1       outon:127.0.0.1
Pattern:0.0.0.0        NextHop:129.187.214.254 outon:129.187.214.23
```

Soll nun z.B ein IP-Paket mit der Empfänger-Adresse 111.222.333.44 verschickt werden, so paßt erst das letzte `Pattern` und das Paket wird von der IP-Adresse 129.187.214.23 (`out on`) an die IP-Adresse 129.187.214.255 (`NextHop`) geschickt.

Hier noch ein Auszug aus einer `HostIPRoutingList`:

```
Host-Name: hpheger3.nm.informatik.uni-muenchen.de
Number of Interfaces: 4
1. InterfaceElement:
Description: ni0 Hewlett Packard Network Interface Pseudo Driver
IPAddress:   null
2. InterfaceElement:
Description: ni1 Hewlett Packard Network Interface Pseudo Driver
IPAddress:   null
3. InterfaceElement:
Description: lo0 Hewlett-Packard Software Loopback
IPAddress:   127.0.0.1
4. InterfaceElement:
Description: lan0 Hewlett-Packard LAN Interface Hw Rev 0
IPAddress:   129.187.214.23
IPRouting:
Pattern:129.187.214.23 NextHop:127.0.0.1       out on:127.0.0.1
Pattern:129.187.214.0  NextHop:129.187.214.23  out on:129.187.214.23
Pattern:127.0.0.1      NextHop:127.0.0.1       out on:127.0.0.1
Pattern:0.0.0.0        NextHop:129.187.214.254 out on:129.187.214.23
```

Die ersten beiden Interfaces sind Pseudo-Interfaces, und nicht physikalisch im Rechner vorhanden. Das dritte Interface ist das sogenannte Loopback-Device (erkennbar an der 127.xxx.yyy.zz IP-Adresse). Alle IP-Pakete, die an das Loopback-Device geschickt werden, gehen an den Host, von dem es geschickt wurde, zurück ("zurückschleifen").

3.4 Quellen

Alle implementierten Klassen ausgehend von `/proj/fagent/JAT/src`:

```
Interpreter_aiprs/AIPRSInterface.java
Interpreter_aiprs/AIPRSInterpreter.java
Interpreter_aiprs/AIPRServer.java
Interpreter_ais/AISInterface.java
Interpreter_ais/AISInterpreter.java
Interpreter_ais/AIServer.java
Interpreter_ahs/AHSInterface.java
Interpreter_ahs/AHSInterpreter.java
Interpreter_ahs/AHServer.java
Interpreter_shared/RMIconnect.java
Interpreter_shared/RMIInterpreter.java
```

```
Interpreter_snmp/SNMPVector.java
Resource/HostElement.java
Resource/HostIPRoutingElement.java
Resource/HostIPRoutingList.java
Resource/HostInterfaceElement.java
Resource/HostInterfaceList.java
Resource/HostList.java
Resource/IPRoutingElement.java
Resource/IPRoutingList.java
Resource/InterfaceElement.java
Resource/InterfaceList.java
```

4 Anwendung

Um die gesamte `HostIPRoutingList` zu bekommen, muß man folgende Schritte durchführen:

1. von einer von `AInterpreter` abgeleiteten Klasse `X` aus die Anfrage `RMIconnect.askRMI(this.agent,"MyName","AIPRS","AIPRS");` starten.

2. in der Methode `X.interpretLanguage()` den `:content` nach der performative `tell-resource` parsen.

3. den Host und den Server-Namen aus dem `:content` auslesen.

4. Die AIPRServer-Methode über die Schnittstelle aufrufen.

```
try {
  AIPRSInterface aiprsi = (AIPRSInterface)
    Naming.lookup("rmi://"+Host+"/"+Server-Name);
  HostIPRoutingList hostIPRList = aiprsi.getIPRouting();
  System.out.println(hostIPRList.getString());
}
catch (Exception e) { e.printStackTrace(); }
```

5. mit `make all` vom Verzeichnis `/proj/fagent/JAT` aus die Übersetzung starten. Übersetzt kann zur Zeit nur auf den sunhegering- Maschinen werden.

6. auf der sunhegering2 (ist im `init_file.ans` fest kodiert) mit `make ans` den ANS und AES starten (siehe 5.1). Falls dies nicht funktioniert, muß man zuerst im Makefile des Verzeichnisses `/proj/fagent/JAT` unter `ans:` nachschauen, welches Init-File abgearbeitet wird (sollte "init_file.ans"sein). Dann im Verzeichnis `/proj/fagent/JAT/Scripts` das entsprechende Init-File nach dem Host durchsuchen und `make ans` auf dem angegebenen Host ausführen.

7. auf beliebigen Hosts `make ahs`, `make ais` und `make aiprs` ausführen.

8. warten bis alle Agenten hochgefahren sind.

9. den eigenen Client starten.

Anzumerken ist, daß man natürlich auch Anfragen an die anderen beiden Interpreter richten kann.

5 Installation

Alle in diesem Abschnitt beschriebenen Verzeichnisse und Dateien sind ab dem Verzeichnis `/proj/fagent/JAT` zu verstehen.

5.1 Makefile

Damit das ganze Projekt kompiliert wird, muß man das "große" Makefile in `/proj/fagent/JAT` benutzt werden. Dieses Makefile lädt als erstes die Datei `make.def`, indem vorallem die Kompilerversion und der Klassenpfad gesetzt wird. Mit `make all` kann das gesamte Projekt übersetzt werden. Um die einzelnen FMAs zu starten, wird ebenfalls das Makefile benutzt. Dabei wird ein Init-Datei abgearbeitet, welche sich in dem Verzeichnis `Scripts` befinden. Beispiel eines Init-Dateis (`init_file.ans`):

```
(evaluate :sender init-file :receiver agent :language KQML
 :ontology agent :content (tell-arsurl :value
http://wwwmnmteam.informatik.uni-muenchen.de\
      /proj/fagent/JAT/classes/))

(evaluate :sender init-file :receiver agent :language KQML
:ontology agent  :content (tell-domain :value fopra.tb))

(evaluate :sender init-file :receiver agent :language KQML
:ontology agent :content (tell-resource :type address
:name ANS :value "129.187.214.2:5001;"))

(evaluate :sender init-file :receiver AES :language KQML
:ontology AES :content (init))
```

Die ersten drei `evaluate`-Anweisungen müssen in jeder Init-Datei vorhanden sein. In der ersten Anweisung wird Pfad für die zu ladenden Klassen angegeben. In der zweiten Anweisung wird ein Domänenname festgelegt. Und in der dritten Anweisung werden Host und Port des ANS festgelegt (muß in **allen** Init-Dateien identisch sein). Mit den restlichen Anweisung können nun beliebige Interpreter gestartet werden.

Die neueste Version der Online-Dokumentation erhält man mit `make doku`. Die Dokumentation wird unter dem Verzeichnis `/proj/fagent/htdocs` gespeichert.

In jedem einzelnen Quell-Verzeichnis befinden sich auch Makefiles, die nur das jeweilige Verzeichnis kompilieren. Neue Quelldateien werden von allen Makefiles automatisch übersetzt, das Makefile muß nicht verändert werden.

Literatur

[Adv] http://www.adventnet.com/snmpapi/index.html.

[Fla97] D. Flanagan. *Java in a Nutshell.* O'Reilly, second edition, 1997.

[Fro97] R. Frost. *The Java Agent Template v0.3.* http://cdr.standford.edu/ABE/JavaAgnet.html, 1997.

[Hol97] A. Hollerith. Entwurf einer Architektur für flexible Agenten auf der Basis des Konzepts von Management by Delegation. Master's thesis, Technische Universität München, February 1997.

[Wen97] M. Wennrich. Erstellung eines Kommunikationsmodells für kooperierende Agenten für das Management von verteilten Systemen. Master's thesis, Technische Universität München, February 1997.

[Kot97] T. Kotselidis. Erweiterung eines Java-Agenten um Managmentfunktionenen. Technical report, Technische Universität München, 1997.

[Mou97] M. Mountzia. *Flexible Agents in Integrated Network and Systems Management.* PhD thesis, Technische Universität München, 1997.

[RMI] http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/getstart.doc.html.

[Ros94] M. Rose. *The Simple Book.*