Network Importer

Dokumentation zur Bachelorarbeit 05/06

Martial Seifriz
Aumatt 27
3175 Flamatt

24. September 2006

Bachelor of Science in Computer Science

Naturwissenschaftliche Fakultät der Universität Fribourg
Departement für Informatik
TNS Forschungsgruppe

Professor: Ulrich Ultes-Nitsche
Asisstenten: David Buchmann, Dominik Jungo

Abstract

Diese Arbeit dokumentiert die Entwicklung und Verwendung des Network Importers für das VeriNeC Projekt. Ziel dieses Importers ist es, mittels Packet-Sniffer, Traceroute und Host-Scanner ein existierendes Netzwerk abzubilden und möglichst viele Informationen darüber zu sammeln, zu verarbeiten und zu speichern.
VeriNeC soll es ermöglichen ein Netzwerk zuerst abstrakt zu konfigurieren und ausgiebig zu prüfen. Nach erfolgreichem Testen soll die erhaltene Konfiguration auf ein reales Netzwerk exportiert werden können. Damit ein existierendes Netzwerk in VeriNeC importiert werden kann, wurde bereits ein Packet-Sniffer implementiert [3], der einzelne Hosts in einem Netzwerk entdecken kann. Nachdem ein Host bekannt ist, können durch einen Port-Scanner die laufenden Dienste aufgelöst werden. Diese Arbeit dokumentiert nun, wie es möglich ist mittels Traceroute die Struktur eines Netzwerkes zu erhalten und abzubilden.
Im folgenden soll ein Überblick über die Entwicklung, Implementation und Verwendung eines in Java geschriebenen Traceroute Package gegeben werden. Zudem wird dessen Einbindung und Verwendung im VeriNeC Importer erklärt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

VeriNeC[1] ist ein Projekt der TNS[2] Research Group der Universität Fribourg[3]. VeriNeC steht für *Verified Network Configuration* und soll es ermöglichen ein Netzwerk zuerst abstrakt zu definieren und anschliessend ausgiebig auf seine Funktionstüchtigkeit und Sicherheit zu testen. Dafür werden die verschiedenen Komponenten und deren Dienste simuliert. Nach erfolgreicher Simulation soll die abstrakte Konfiguration für die verschiedenen, realen Komponenten übersetzt und auf diese exportiert werden können.

1.1 Beschreibung der Aufgabe

Um ein bestehendes Netzwerk zu verbessern, soll VeriNeC ein existierendes Netzwerk möglichst gut importieren können. Dafür wurde ein Importer Modul implementiert [3]. Dieses benutzt einen Packet-Sniffer um Netzwerkkomponenten (künftig auch *Nodes* genannt) zu entdecken, bildet diese in der VeriNeC XML Repräsentation ab und zeigt sie im GUI an. Um möglichst viele Informationen über die Netzwerkkomponenten zu erhalten werden diese anschliessend gescannt. Dies ermöglicht, nicht nur die Netzwerkkomponenten selbst zu entdecken, sondern auch die darauf laufenden Dienste.
Die Aufgabe der hier vorliegenden Arbeit war nun, das Importer Modul so zu erweitern, dass es möglich wird, die Verbindung zwischen einzelnen Nodes zu entdecken. Um dies zu erreichen sollte ein Traceroute-Modul implementiert werden, das die Informationen des Sniffers weiterverarbeitet und die Struktur des bekannten Netzwerkes zurück gibt. Die so erhaltene Netzwerkstruktur sollte anschliessend im existierenden GUI angezeigt werden.

1.2 Projektteam

Die Bachelorarbeit wurde von der Forschungsgruppe TNS der Universität Fribourg betreut:
Professor: Ulrich Ultes-Nitsche
Assistenten: David Buchmann, Dominik Jungo
Student: Martial Seifriz

1.3 Struktur der Arbeit

Das Kapitel 2 dreht sich um die Frage, wie einzelne Netzwerkkomponenten, wie z.B. Router, zwischen zwei Hosts gefunden werden können. Es wird erläutert, wie dies mit einem Traceroute erreicht werden kann, wie diese funktionieren und was für Möglichkeiten bestehen um ein Traceroute aus Java zu machen. Schlussendlich wird

[1] http://diuf.unifr.ch/tns/verinec
[2] http://diuf.unifr.ch/tns
[3] http://www.unifr.ch

die erarbeitete Implementation und deren Benutzung vorgestellt.
Im Kapitel 3 wird auf den ganzen Import-Prozess eines Netzwerkes eingegangen. Dazu wird dessen Funktionsweise und Design erklärt. Anschliessend wird die Implementation und einige wichtige Klassen erläutert.
Daraufhin wird in Kapitel 4 die Benutzung des Importers vorgestellt.
Abschliessend sind in Kapitel 5 die Grenzen und Erweiterungsmöglichkeiten des Importer-Moduls sowie eine persönliche Konklusion zu finden.
Der Anhang beinhaltet einige Worte zu Änderungen an bestehenden Klassen und die UML-Klassendiagramme des Traceroute- und Importer-Moduls. Zudem wird der Aufbau der beigelegten CD erklärt.

2 Entwicklung des Traceroute Moduls

Traceroute ist ein Analysewerkzeug, welches es ermöglicht die Netzwerkkomponenten, die ein Datenpaket zu einem Zielhost passiert, herauszufinden.

2.1 Der Traceroute Algorithmus

Der Algorithmus basiert auf IP (*Internet Protocol*) und ICMP (*Internet Control Message Protocol*). ICMP ist ein Steuerprotokoll und wird im RFC 792 [4] beschrieben. Es wird in einem IP-Paket gekapselt und definiert verschiedene Nachrichtentypen, die je nach Situation zum Einsatz kommen.
Im Traceroute Algorithmus werden drei dieser Nachrichtentypen verwendet. Die ersten zwei ICMP-Nachrichtentypen, die benutzt werden können, um festzustellen ob und wie schnell ein Host in einem Netzwerk erreichbar ist, sind die ICMP Echo-Request und ICMP Echo-Reply Nachrichten. Sobald ein Computer eine ICMP Echo-Request Nachricht erhält, muss er unverzüglich mit einer Echo-Reply Nachricht antworten. Die dritte ist vom Typ Time-Exceeded. Jedes IP-Paket besitzt, wie man in Abbildung 1 sehen kann, in seinem Header einen Zähler (Time-to-Live, abgekürzt TTL), der bei jedem Router um eins dekrementiert wird. Erreicht er null wird das Paket zerstört. Dies, damit ein Paket nicht unendlich lange in einem Netzwerk umherirren kann. Um den Sender des zerstörten Pakets zu informieren, dass seine Nachricht das Ziel nicht erreicht hat, wird ihm eine ICMP Time-Exceeded Nachricht zugestellt.
Der Traceroute Algorithmus versucht nun dem Zielhost eine ICMP Echo-Request Nachricht zu senden und von diesem eine Antwort zu erhalten. Damit jedoch alle Stationen, welche ein Datenpaket zu diesem Zielhost passiert, herausgefunden werden können, wird der TTL Wert im IP Header anfänglich bloss auf eins gesetzt. Dies führt dazu, dass die Nachricht bereits beim ersten Router weggeworfen und eine ICMP Time-Exceeded Nachricht zurückgeschickt wird. Jedoch kann nun aus dieser die IP-Adresse des ersten Routers gewonnen werden. Um nun den gesamten Weg zum Zielhost aufzulösen, wird das TTL kontinuierlich inkrementiert bis die Nachricht schliesslich beim Zielhost ankommt und dieser mit einer ICMP Echo-Request Nachricht antwortet. Auf diese Weise können alle IP-Adressen der Router bis zum Zielhost gesammelt werden.

2.2 Vorgehen

Java selbst bietet bloss die Möglichkeit, Datenströme über ein Netzwerk zu verschicken und zu empfangen. Dies ist sehr komfortabel zu benutzten und reicht für die meisten Anwendungen völlig aus. Möchte man jedoch einzelne Datenpakete erstellen, so ist dies in Java selbst nicht möglich. Folglich kann ein Traceroute nicht direkt implementiert werden. So wurde nach alternativen Möglichkeiten gesucht, wie man

[4] ftp://ftp.rfc-editor.org/in-notes/rfc792.txt

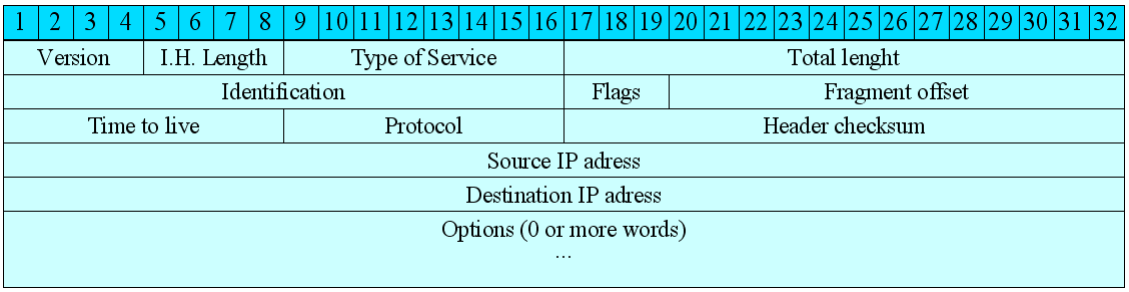

Abbildung 1: Der IP-Header

ein Traceroute aus Java machen könnte. Schlussendlich standen folgende Lösungen zur Auswahl:

- Eine eigene Implementation in C schreiben und mittels JNI [5] einbinden.
- Über die Kommandozeile entweder das Traceroute Programm des Betriebssystems oder ein sonst selbstständig laufendes Traceroute Programm aufrufen und danach dessen Output parsen.
- Eine Bibliothek für Java suchen, die RAW Socket Funktionalität zur Verfügung stellen kann und mit Hilfe dieser selbst ein Traceroute implementieren.

2.2.1 Evaluation der gefundenen Möglichkeiten

Implementation in C

Hätte man eine sehr saubere und einwandfrei funktionierende Implementation, welche man mittels JNI aufrufen könnte, so wäre dies sicher ein akzeptabler Weg. Doch eine saubere und fehlerfreie Version in C zu schreiben ist eher schwierig und die Wahrscheinlichkeit, dass man kleine Fehler mit einbaut ist relativ gross. Jedoch war die Lektüre verschiedener Tutorials über RAW Sockets in C sehr aufschlussreich und half, ein fundiertes Verständnis für das Thema zu entwickeln.

Kommandozeilenbefehl

Sowohl in Windows als auch in den meisten Linux Distributionen wird ein Programm mitgeliefert, mit welchem ein Traceroute gemacht werden kann. Dies sind *traceroute* (resp. *traceroute6*) für Linux und *tracert* für Windows. Von diesen darf angenommen werden, dass sie korrekt funktionieren und somit einen einfachen Weg darstellen. Das Problem hierbei ist, dass der Output nicht für eine Software, sondern für den Menschen gemacht wurde. Man kann nicht davon ausgehen, dass dieser bei allen Versionen des Betriebssystems genau gleich ist und daher ist das Parsen des Outputs eher heikel.

[5] JNI steht für Java Native Interface und ermöglicht es, native Methoden, die z.B. in C oder C++ implementiert sind, in Java zu verwenden.

Eine andere Möglichkeit besteht darin, ein nicht vom Betriebssystem-Entwickler implementiertes Traceroute Programm über die Konsole zu starten und wiederum den Output zu parsen. Hierbei könnte z.B. *tcptraceroute* [6] eingesetzt werden. Dies ist ein Traceroute Tool, das anstatt mit ICMP-Paketen mit TCP Verbindungen arbeitet. Dadurch kann mit verschiedenen Störfaktoren, wie Firewalls, welche teilweise ICMP Pakete blockieren, besser umgegangen werden.

Bibliothek für Java

Dies ist sicher die bessere Alternative als selbst eine C Implementation zu schreiben, da eine bereits existierende Bibliothek von vielen Benutzern getestet und immer wieder verbessert wurde. JPcap drängte sich als sehr gute Variante auf. Sie basiert auf Winpcap resp. auf libpcap für Linux.

2.2.2 Entscheidung

Schliesslich wurde beschlossen eine eigene Implementation mittels JPcap zu schreiben, jedoch wurde auch die Variante mit dem Kommandozeilenbefehl des Betriebssystems umgesetzt. Da der Parser des Kommandozeilen-Traceroutes auch für einige externe Programme funktioniert, wurde auch diese Variante implementiert. So bestehen nun also drei Möglichkeiten ein Traceroute zu machen.

2.3 Implementation

Als Ergebnis liegt nun ein Traceroute Package vor, welches erlaubt, die Stationen, welche ein Datenpaket zu seinem Ziel passiert, herauszufinden.

2.3.1 Verwendung des Moduls

Ein Benutzer kann ein Traceroute-Objekt erstellen, welches eine Methode anbietet, um ein Trace zu machen. Der Zielhost und, wenn erwünscht, auch die maximale Anzahl aufzulösender Nodes sowie das Timeout zwischen Senden und Empfangen einer Anfrage können dieser Methode als Parameter übergeben werden. Bei der Instanzierung des Traceroute-Objekts kann im Konstruktor der zu verwendende Algorithmus resp. die Möglichkeiten, ein Traceroute zu machen, spezifiziert werden. Zur Zeit sind es folgende:

- *TraceRoute.JPCAP*; die JPcap Implementation
- *TraceRoute.OS_SHELL_COMMAND*; der Kommandozeilenbefehl
- *TraceRoute.TCP_TRACEROUTE*; ein externes TCP Traceroute Programm

Wird kein Algorithmus ausgewählt, so wird als Standard der Kommandozeilenbefehl des Betriebssystems verwendet. Um die JPcap Implementation verwenden zu können, muss sowohl JPcap wie auch Winpcap resp. libpcap für Linux installiert sein

[6] http://michael.toren.net/code/tcptraceroute

und der Benutzer muss über Administratorrechte verfügen, denn die Benutzung von RAW Sockets verlangt dies. Auch für die TCP Traceroute wird jeweils Winpcap resp. libpcap benötigt. Alle benötigten Bibliotheken und Programme sind auf der beigelegten CD zu finden und werden im Anhang C ausführlich erläutert.
Ein Beispiel:

```
TraceRoute tr = new TraceRoute(TraceRoute.JPCAP);
Vector result = tr.traceroute("seifriz.ch");
```

Hier wird die JPcap Implementation verwendet und anschliessend ein Traceroute zu *seifriz.ch* gemacht. Die gefundenen Nodes werden in String Repräsentation im Vector *result* gespeichert.

2.3.2 JPcap Traceroute

Die Klasse *TRJPcap* implementiert den Traceroute Algorithmus, indem zum Senden und Empfangen von ICMP Nachrichten die Bibliothek JPcap verwendet wird. JPcap ist ein Java Wrencher für Winpcap resp. libpcap und ermöglicht, es RAW Socket Funktionalität in Java zu benutzen.
Es wurde der klassische Algorithmus mit ICMP Nachrichten und ansteigendem Time-to-live Wert im IP Header verwendet. Um mehrere synchrone Traceroutes zu ermöglichen, muss jeder Aufrufer der Traceroute Methode über eine eigene, einzigartige Identifikationsnummer verfügen. Diese wird jeweils zu Beginn über die statische Methode *requestID()* von *TRJPcap* geholt und am Ende mit *releaseID(id)* wieder freigegeben.

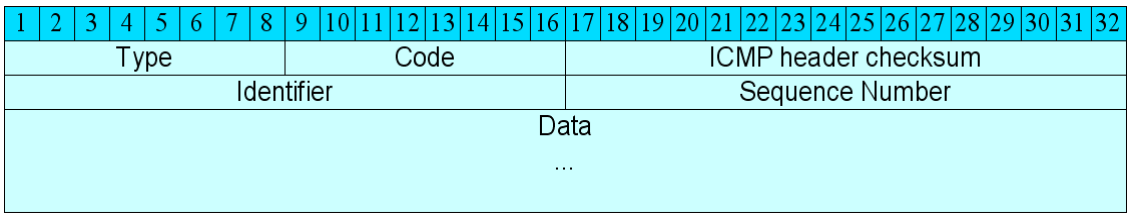

Abbildung 2: Header einer ICMP Echo-Request Nachricht

Diese Identifikationsnummer wird in den Header der ICMP Echo-Request Nachricht geschrieben. Dieser ist in Abblidung 2 zu sehen. Erhält man nun eine ICMP Time-Exeeded Nachricht, so kann die Nachricht einer bestimmten Identifikationsnummer zugeordnet werden, denn eine ICMP Time-Exeeded Nachricht enthält als Daten mindestens den IP-Header und die ersten 64 Bit des ursprünglichen Datenpakets. Dies wird in Abbildung 3 dargestellt. Da der ICMP Header einer Echo-Request Nachricht genau 64 Bit lang ist, wird dieser zurückgeschickt und die Identifikationsnummer kann daraus gewonnen werden. Die finale ICMP Echo-Reply Nachricht ist noch einfacher zuzuordnen, indem man die Sender IP-Adresse im Header des IP-Pakets ausliest und einer bestimmten Identifikationsnummer zuweist.

Da das Traceroute für den Importer von VeriNeC möglichst schnell ablaufen soll, weil der Weg zu vielen Hosts aufgelöst werden muss, wird nach eingetroffener ICMP Echo-Reply resp. Time-Exeeded Nachricht darauf verzichtet noch eine zweite oder gar dritte Anfrage für das selbe TTL zu machen. Anderen Implementationen, wie z.B. die des Betriebssystems, laufen etwas langsamer ab, da sie mehrere Pakete für das selbe TTL senden und auch den Mittelwert der Übertragungszeit messen. Tritt jedoch ein Timeout auf, dass heisst, die Zeitspanne zwischen dem Senden und Empfangen einer Nachricht ist grösser als ein bestimmter Grenzwert, so wird die Nachricht noch so oft wiedergesendet wie in *ATraceRoute.MAX_RETRIES* definiert wurde. Dies sind standardmässig drei Mal. Treten mehr Timeouts auf, als in diesem Wert spezifiziert, so wird angenommen, dass der Host aus irgendeinem Grund, z.B. wegen einer Firewall, nicht auflösbar ist resp. keine ICMP Nachricht zurückschickt. In diesem Fall wird dieser Host durch eine Standard IP-Adresse im Resultat repräsentiert. Diese ist in *ATraceRoute.UNREACHABLE_HOST_IP* definiert und sollte als Platzhalter angesehen werden. Können mehr Nodes, als in *ATraceRoute.MAX_UNREACHABLE_HOSTS* spezifiziert ist, nicht aufgelöst werden, so wird das Traceroute abgebrochen.
Am Ende erhält der Aufrufer einen Vektor, der die IPs der gefundenen Netzwerkkomponenten in String Repräsentation enthält, zurück.

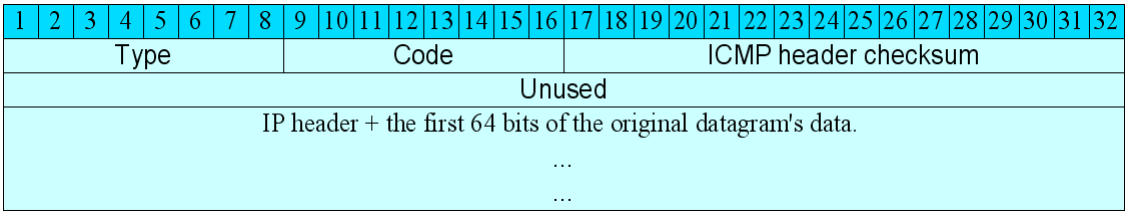

Abbildung 3: Header einer ICMP Time-Exeeded Nachricht

2.3.3 Kommandozeilenbefehl Traceroute

Die zweite Implementation eines Traceroutes war, den Kommandozeilenbefehl des Betriebssystems zu benutzen. Dies geschieht, indem Java über *Runtime.getRuntime().exec(command);* einen Befehl in der Command-Shell ausführen lässt. Java erhält den Output und kann ihn danach parsen. Dies geschieht in diesem Falle folgendermassen: Jeder Schritt des Resultates beginnt mit einer Anzahl Leerschläge und einer Zahl zwischen eins und 255, dies entspricht dem Time-to-live Wert im IP-Header. Darauf folgt, ob ein Host aufgelöst werden konnte oder nicht. Dies wird entweder durch ein Asterisk, wenn der Node nicht aufgelöst werden kann, oder durch eine Zahl, welche die benötigte Reaktionszeit in Millisekunden angibt, dargestellt. Danach wird entweder bloss die IP-Adresse oder zudem der Hostname des gefundenen Nodes ausgegeben. Abbildung 4 zeigt ein Traceroute nach *unifr.ch* in Windows mittels *tracert*. Mit dem Parameter *-d* wird ausgedrückt, dass die Hostnamen nicht aufgelöst werden sollen. Dies, damit die Ausführung etwas schneller abläuft. Der untenstehende Code

stellt eine Variante dar, wie man einen IP-Adressen Parser für den Kommandozeilen Output in Java implementieren kann.

```
...
//each line begins with withspace and a number 1, 2, 3 ...
Pattern lineStartPattern = Pattern.compile("^[\\s]*([0-9]{1,3})");
Matcher lineStartMatcher = lineStartPattern.matcher(output);
if(lineStartMatcher.find()){
    //an ip has the form xxx.xxx.xxx.xxx
    Pattern ipPattern = Pattern.compile("([0-9]{1,3}\\.)" +
            "([0-9]{1,3}\\.)" +
            "([0-9]{1,3}\\.)" +
            "([0-9]{1,3})" ); //long variante
    Matcher ipMatcher = ipPattern.matcher(output);
    if(ipMatcher.find()){
        ...
```

Kann jeder Node problemlos aufgelöst werden, so erhält der Aufrufer einen Vector, welcher die IP-Adressen der Netzwerkkomponenten in String Repräsentation enthält, zurück. Kann ein Host nicht aufgelöst werden, so wird der Standardeintrag aus *ATraceRoute.UNREACHABLE_HOST_IP* dem Resultat als Platzhalter hinzugefügt. Können mehr Nodes als in *ATraceRoute.MAX_UNREACHABLE_HOSTS* spezifiziert ist, nicht aufgelöst werden, so wird die Ausführung des Traceroutes abgebrochen und der Aufrufer erhält die bis zu diesem Zeitpunkt gefundenen Hosts zurück.

2.3.4 Externes Programm

Insofern externe, eigenständige Traceroute Programme ungefähr das selbe Output-Format einhalten, wie der Kommandozeilenbefehl Traceroute des Betriebssystems, so kann der gleiche Parser gebraucht werden. Dies ist für TraceTCP unter Windows und TCPTrancreoute für Linux der Fall. [7] Genau diese werden als dritte Variante eingesetzt. Mit *Runtime.getRuntime().exec(command);* wird ein externes Programm aufgerufen, der in 2.3.3 beschriebene Parser wird verwendet und der Benutzer erhält einen Vector der IP-Adressen zurück. Auch die Spezialfälle, wie unauflösbare Hosts, werden genau gleich behandelt wie im Kommandozeilenbefehl Traceroute.

2.3.5 Tests

TestTraceRoute aus dem Package *verinec.importer.analysis.traceroute* implementiert einige *JUnit* Tests. Für jede Traceroute Implementation wird der gleiche Test ausgeführt. Dies geschieht zuerst für eine einzelne Instanz und danach noch für mehrere

[7] Dies sind beides eigenständige Projekte und haben rein organisatorisch miteinander nichts gemein. TCPTraceroute ist unter http://michael.toren.net/code/tcptraceroute/ und TraceTCP unter http://tracetcp.sourceforge.net/ zu finden.

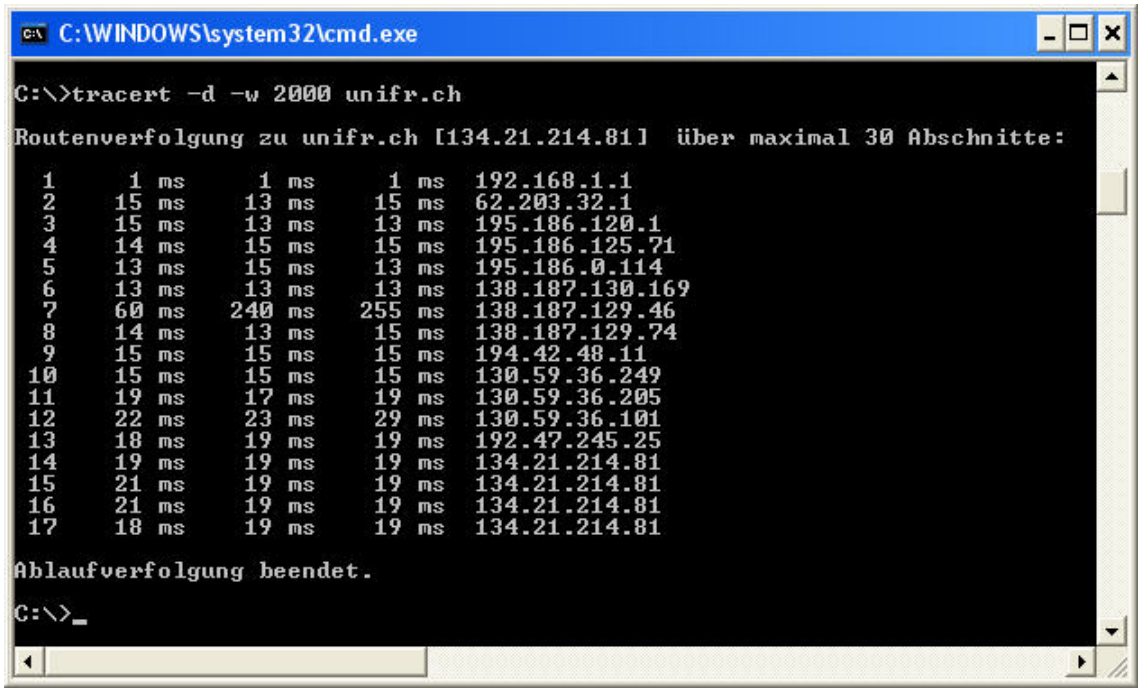

Abbildung 4: Traceroute Kommandozeilenbefehl nach *unifr.ch* in Windows

Threads, die alle gleichzeitig ein Traceroute machen. Dabei wird zuerst geprüft, was geschieht, wenn ein Zielhost gar nicht existiert. Daraufhin wird ein Traceroute nach *google.ch* gemacht, dessen Resultat einwandfrei sein sollte und eines nach *seifriz.ch*, bei welchem einige Hosts nicht auflösbar sind. Es ist sehr wichtig zu erwähnen, dass diese Tests jetzt funktionieren, jedoch in Zukunft fehlschlagen können. Denn die momentan erhaltenen Resultate können sich zu jedem Zeitpunkt ändern, sobald irgend ein gefundener Router oder Server modifiziert wird.
Zudem besteht auch die Möglichkeit, den Parser aus *ATRShellCommand* für die Kommandozeilen Traceroutes zu testen. Dafür wird ein in *testTRShellCommand.txt* gespeichertes Resultat eingelesen, dem Parser übergeben und schliesslich mit der Musterlösung verglichen.

3 Implementation des Network Importers

Der Importer Prozess läuft in vier Etappen ab. Der Sniffer versucht als erstes, einige Hosts im Netzwerk zu entdecken. Hat dieser seine Arbeit getan und ist es erwünscht, so wird mittels Traceroute versucht, die Verbindung zu diesen einzelnen Hosts herauszufinden. Darauf wird, wiederum nur, wenn es explizit gewünscht wird, mittels des Scanners versucht, die auf den Hosts laufenden Dienste herauszufinden. Wird jeweils ein Host vom Sniffer oder den Traceroutes gefunden, so versucht der Importer sofort, dessen Hostname aufzulösen. Sobald alle Aufgaben beendet sind, kehrt der Importer zum VeriNeC Studio zurück. Abbildung 5 bietet einen schematischen Überblick des Import Prozesses.

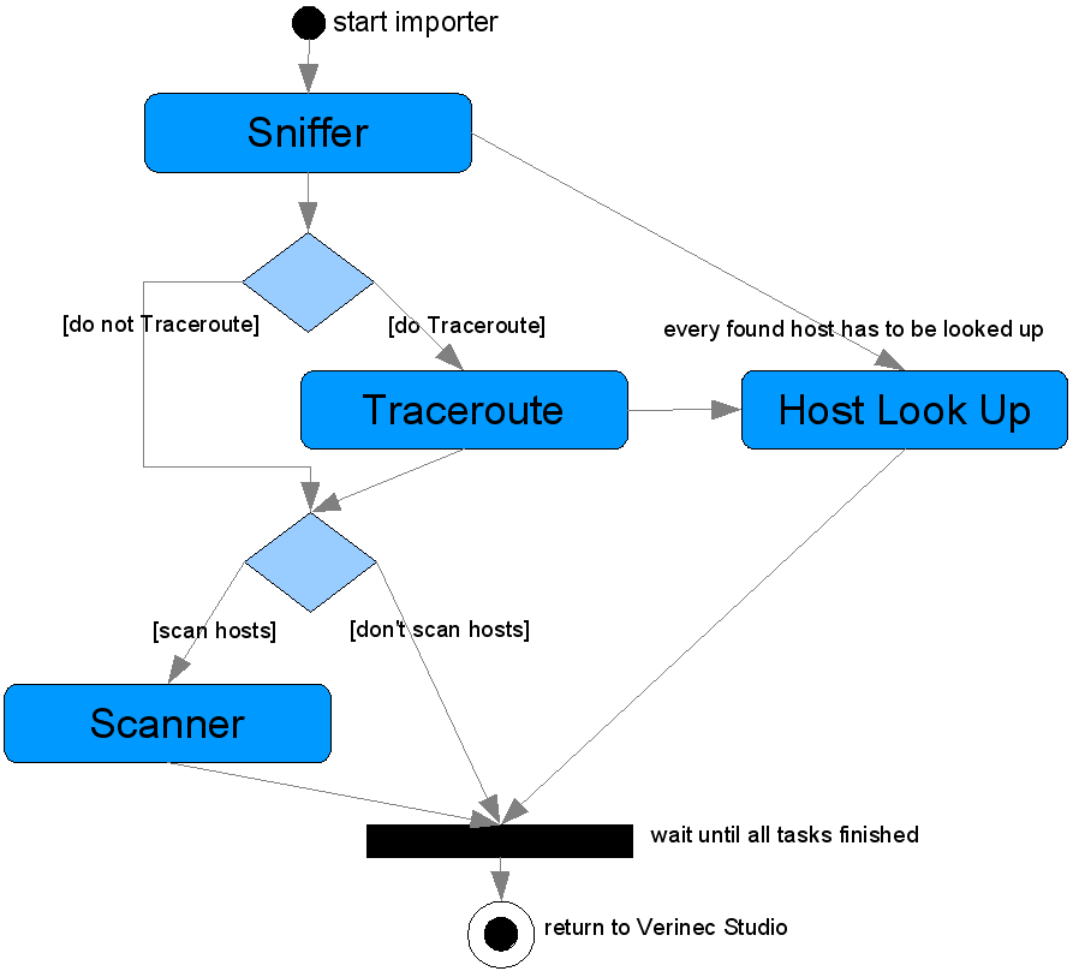

Abbildung 5: Überblick des Import Prozesses

3.1 Design

Die zentrale Instanz des Importers ist der *AnalysisThread*. Er verwaltet und kontrolliert die einzelnen Aufgaben des ganzen Import-Prozesses, unterhält die Datenstrukturen der Resultate und die aktuellen Zustände der einzelnen Importer-Aufgaben. Abbildung 6 zeigt das Design des Importer-Prozesses auf. Für jede der drei Aufgaben, Sniffer, Traceroute und Scanner, besitzt der *AnalysisThread* ein Start-Objekt. Mit diesem lassen sich die einzelnen Aufgaben starten und beenden. Erst diese Objekte erzeugen dann die Threads, deren Anzahl vom Benutzer spezifiziert wurde. Die einzelnen Threads arbeiten anschliessend ihre spezielle Aufgabe ab. Zum Speichern eines Resultates oder zum Aktualisieren des eigenen Zustandes ruft jeder Thread seinen Starter auf. Dieser wiederum leitet die Anfrage dem *AnalysisThread* weiter. Schliesslich werden dort die Resultate gesammelt und der Zustand jeder Aufgabe gespeichert, damit er dem Benutzer angezeigt werden kann. Sind alle Aufgaben beendet, werden die gefundenen Informationen in die VeriNeC XML Repräsentation überführt und die Objekte für das GUI des VeriNeC Studios erstellt.

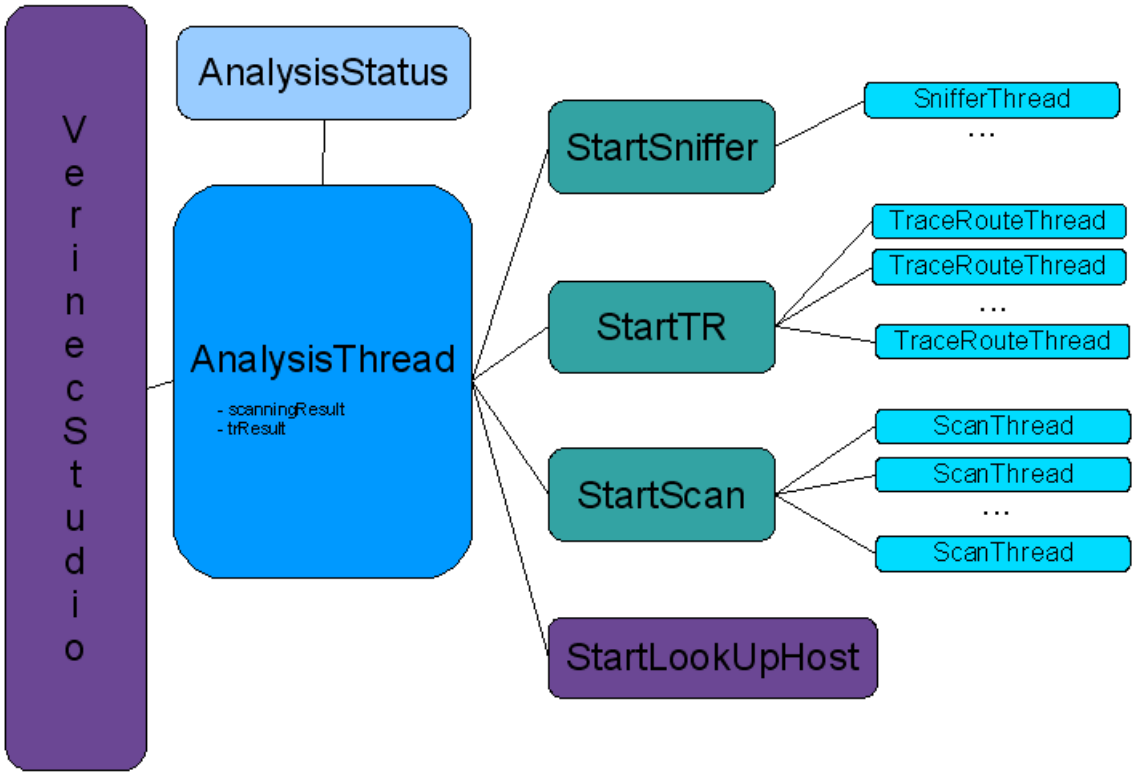

Abbildung 6: Design Importer - The big picture

3.2 Organisation des Source Code

Der zentrale Code rund um den *AnalysisThread* befindet sich im Package *verinec.importer.analysis*. In diesem Package befinden sich auch alle Interfaces und abstrakten Klassen der Aufgaben. Die Aufgaben selbst verfügen je über ein eigenes Package, d.h. der Sniffer ist in *verinec.importer.analysis.sniffer*, Traceroute in *verinec.importer.analysis.traceroute* und der Scanner in *verinec.importer.analysis.scan* untergebracht. Der Source Code befindet sich auf der beigelegten CD im Ordner *verinec/Importer/java/src/verinec/importer/analysis*. Im Anhang B zeigt Abbildung 12 ein UML-Diagramm mit den wichtigsten Klassen des Importers auf.

3.3 Die wichtigsten Klassen

StartSniffer

Die StartSniffer Klasse wurde von [3] übernommen, jedoch relativ stark abgeändert. Der *AnalysisThread* instanziert ein *StartSniffer-Objekt*, dieses erzeugt für jede Netzwerkkarte, deren Verkehr abgehört werden soll, einen *SnifferThread*, startet diese, kontrolliert sie und wartet auf deren Terminierung.

SnifferThread

Auch diese Klasse wurde zwar von [3] übernommen, doch sehr stark abgeändert. So wird nun eine neue Bibliothek zum Sniffen verwendet. [8] Einem *SnifferThread* wird eine Netzwerkkarte zugeteilt, auf der er den Netzwerkverkehr abhört. Kommt ein Paket an, so wird dieses analysiert und verarbeitet. Danach werden die gewonnenen Informationen, d.h. die IP-Adresse und die benutzten Ports, dem *StartSniffer* weitergegeben, um sie zu speichern.

StartTraceRoute

Der *AnalysisThread* hält ein Objekt dieser Klasse, um die Traceroutes zu den vom Sniffer gefundenen Hosts zu starten und zu kontrollieren. Ein *StartTraceRoute-Objekt* erzeugt so viele *TraceRouteThread* wie der Benutzer im Importer Dialog spezifiziert, startet diese und wartet auf deren Terminierung.

TraceRouteThread

Ein *TraceRouteThread* holt sich bei seinem Starter einen Host, der *getraced* werden soll. Danach führt er ein Traceroute zu diesem Host aus und gibt das Resultat seinem Starter zurück, dieser wiederum lässt das Resultat vom *AnalysisThread* speichern. Dieses Prozedere wiederholt sich so lange bis alle Hosts abgearbeitet wurden. Das

[8] Bizarrerweise existieren zwei grundverschiedene Java Anbindungen von libcap resp. Winpcap mit ein und demselben Namen! Die erste wurde von [3] verwendet. Sie ist auf `http://sourceforge.net/projects/jpcap` zu finden. Die zweite findet man auf `http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html` und wurde für das Traceroute Modul verwendet. Aus verschiedenen Gründen wurde dann entschieden den Sniffer an die neuere Variante anzupassen.

Traceroute Verfahren, das verwendet werden soll, kann vom Benutzer im Importer Dialog bestimmt werden und bleibt immer dasselbe.

StartScan

Auch von dieser Klasse hält der *AnalysisThread* ein Objekt. Es werden wiederum so viele *MHostScanThread* gestartet und kontrolliert, wie es ein Benutzer wünscht.

MHostScanThread

Damit möglichst viele Informationen über eine Netzwerkkomponente bekannt werden, kann ein Host gescannt werden. Dazu wird das Analysetool Nmap verwendet. Dieses schreibt die Analyse in eine XML Datei hinein, welche anschliessend geparst werden kann. Ein *MHostScanThread* [9] holt sich bei seinem Starter einen Host, welcher gescannt werden soll, und ruft via *Runtime.getRuntime().exec(command);* Nmap dafür auf, parst das Ergebnis, lässt es speichern und wiederholt diesen Vorgang, bis alle Hosts abgearbeitet wurden.

```
C:\>nmap.exe -O -sV -oX deee.xml 192.168.1.39

Starting Nmap 4.03 ( http://www.insecure.org/nmap ) at 2006-05-10 15:08
Interesting ports on 192.168.1.39:
(The 1662 ports scanned but not shown below are in state: closed)
PORT     STATE SERVICE        VERSION
21/tcp   open  ftp?
25/tcp   open  smtp           Mercury/32 smtpd (Mail server account Maiser)
80/tcp   open  http           Apache httpd 2.2.0 ((Win32) DAV/2 mod_ssl/2.2.0
105/tcp  open  ph-addressbook Mercury/32 PH addressbook server
106/tcp  open  pop3pw         Mercury/32 poppass service
110/tcp  open  pop3           Mercury/32 pop3d
135/tcp  open  msrpc          Microsoft Windows RPC
139/tcp  open  netbios-ssn
143/tcp  open  imap           Mercury/32 imapd 4.01b
443/tcp  open  ssl            OpenSSL
445/tcp  open  microsoft-ds   Microsoft Windows XP microsoft-ds
3306/tcp open  mysql          MySQL (unauthorized)
MAC Address: 00:0C:29:43:DB:56 (VMware)
Device type: general purpose
Running: Microsoft Windows 2003/.NET|NT/2K/XP
OS details: Microsoft Windows 2003 Server or XP SP2
Service Info: Host: localhost; OS: Windows

Nmap finished: 1 IP address (1 host up) scanned in 611.750 seconds
```

Abbildung 7: Ein Nmap Resultat

StartLookUpHost

Wird ein neuer Node entweder durch den Sniffer oder durch ein Traceroute entdeckt, so wird beim Speichern dessen Name aufgelöst. Das Verfahren dazu wurde von [3] übernommen, jedoch werden die Threads, welche die effektive Arbeit ausführen, neu von einem *StartLookUpHost-Objekt* koordiniert und kontrolliert. Zu erwähnen ist, dass für diese Implementation ThreadPools aus Java 1.5 verwendet werden.

[9] *M* steht für Multiple

AnalysisThread

Dies ist die zentrale Instanz des Import-Prozesses. Hier werden die einzelnen Aufgaben gestartet und Methoden und Datenstrukturen angeboten, um deren Resultate zu speichern. Die vom Sniffer gefundenen Nodes werden in einer HashTable abgespeichert. Wenn keine Traceroutes erwünscht werden, so testet *AnalysisThread* ob die Hosts bereits im GUI des Studios vorhanden sind, wenn nicht, wird die VeriNeC XML Repräsentation erstellt und die Hosts werden dem GUI hinzugefügt.
Soll hingegen die Struktur des Netzwerkes mit Hilfe der Traceroutes erstellt werden, so werden die Resultate der Traceroutes, zum schnelleren Auffinden, zwar auch in einer HashTable gespeichert, jedoch zudem untereinander verknüpft. Das heisst, dass man durch die Traceroutes mindestens eine Baumstruktur des Netzwerkes mit dem Host, auf dem VeriNeC läuft, als Wurzel erhält. Jeder weitere Node enthält mindestens einen Vorgänger und gegebenenfalls auch einige Kinder. Um eine einigermassen schöne Darstellung im GUI zu erhalten, wird genau dieser Tatbestand verwendet um die Hosts anzuordnen. Die Wurzel wird in der Mitte erstellt. Im Kreis um die Wurzel deren Kinder und auf den folgenden Kreisbahnen die Kindeskinder usw. Im letzten Schritt des *AnalysisThread* wird überprüft, ob im GUI bereits einige der gefundenen Nodes vorhanden sind. Falls dies der Fall ist, so werden diese mit den neu gefunden Informationen ergänzt resp. die neuen Informationen werden darüber gelegt. Wenn nicht, so werden alle Nodes neu ins GUI eingefügt und die VeriNeC XML Repräsentation erstellt. Im Falle eines kompletten Neuimportes ist das Resultat wesentlich ästhetischer.

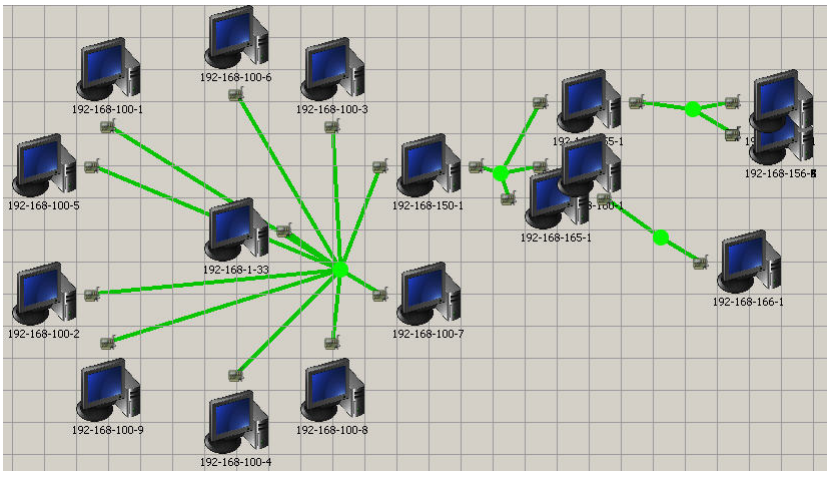

Abbildung 8: Importer Resultat eines Mini-Test-Netzwerkes

Last but not least: Auch der aktuelle Status jeder einzelnen Aufgabe wird im *AnalysisThread* gespeichert, ständig aktualisiert und dem Benutzer über einen Dialog sichtbar gemacht. Diese Informationen werden in der Inneren Klasse *AnalyserStatus* gespeichert und verwaltet.

3.4 Tests

Für den ganzen Import-Prozess an sich wurden keine *JUnit* Tests implementiert. Jedoch bietet die Klasse *AnalyserTestDataManager* die Möglichkeit, die Resultate des Sniffers und der Traceroutes zu speichern und bei Bedarf zu laden. Damit können diese beiden Prozesse simuliert und getestet werden. Um erhaltene Resultate des Sniffers und der Traceroutes zu speichern, muss die Variable *TEST_MODE_SAVE_RESULTS* des *AnalysisThread* auf *true* gesetzt werden. Die erhaltenen Informationen werden darauf in der Datei *importerTestData.xml* gespeichert. Möchte man nun gespeicherte Daten laden, so geschieht dies indem man die Variable *TEST_MODE_SNIFFING* für den Sniffer bzw. *TEST_MODE_TRACEROUTE* für die Traceroutes auf eine Anzahl zu ladeneder Resultate setzt. Sollen keine Daten geladen werden, so werden die Variablen auf *-1* gesetzt. Möchte man hingegen alle Resultate laden lassen, dann geschieht dies durch den Wert *0*.

4 Benutzerhandbuch

Wenn VeriNeC gestartet wird, so befindet man sich im *Configurator-Modus*. Um den Importer zu starten muss man zuerst in den *Importer-Modus* wechseln. Dies geschieht, indem in der Menuleiste das Item *Importer* angewählt wird. Danach muss im Menu *Importer* der Menueintrag *Start Network Analysis* angeklickt werden und der Importer Dialog öffnet sich. Sobald alle Aufgaben konfiguriert sind, kann die Netzwerkanalyse mit dem Button *Run* gestartet werden. Darauf wird ein kleiner Dialog mit dem aktuellen Status der Analyse angezeigt. Dieser bietet nicht nur eine Übersicht des Gesamtprozesses, sondern auch die Möglichkeit, die Analyse abzubrechen. Dies ist nicht zu empfehlen, da in diesem Falle bloss ein Teilresultat zurückgegeben werden kann.

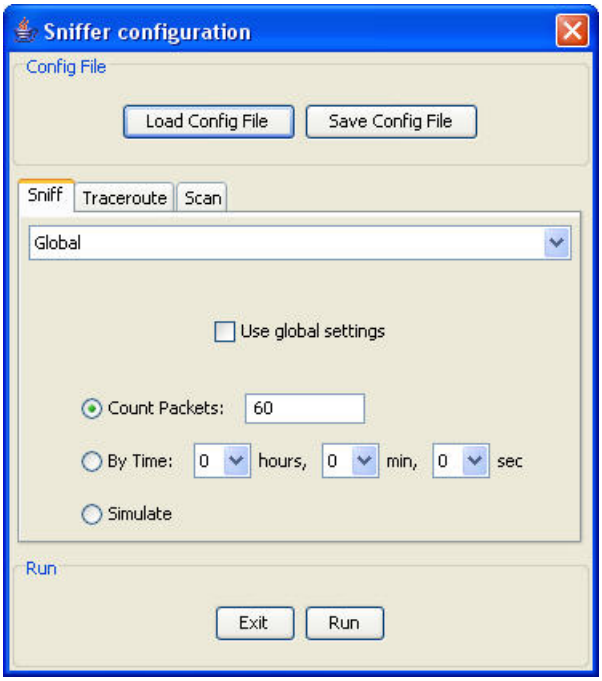

Abbildung 9: Der Importer Dialog mit dem Sniffing Panel

4.1 Sniffer

Für den Sniffer kann spezifiziert werden, auf welchen Netzwerkkarten er Pakete abfangen soll. Für jede ausgewählte Netzwerkkarte kann dann zusätzlich gewählt werden, ob eine bestimmte Anzahl Pakete gesnifft werden sollen oder ob der Sniffer einfach eine bestimmte Zeit laufen soll.

4.2 TraceRoute

Zuerst einmal kann gewählt werden, ob es erwünscht ist, dass Traceroute angewendet wird. Wenn man sich dafür entschliesst, so können diverse Einstellungen vorgenommen werden. Als erstes ist der Algorithmus resp. das Verfahren für die Traceroutes zu wählen. *JPCap Traceroute* ist sicher das schnellste Verfahren, hingegen liefert *TCP Traceroute* das beste Resultat, ist aber langsamer. Das *Shell Command Traceroute* sollte auf jedem Betriebssystem installiert sein, ist aber weder sehr schnell noch sehr gut.
Unabhängig von der Wahl des Algorithmus kann ausgewählt werden, über wie viele Hops ein Trace maximal laufen soll und wie lange nach einer Anfrage auf die Antwort eines Hosts gewartet werden soll, d.h. Timeout. Es kann auch angegeben werden, wie viele Traceroutes synchron ausgeführt werden. Zudem wird ein Filter angeboten, der es ermöglicht, dass nur Hosts eines gewissen Subnets getraced werden. Denn der Sniffer kann auch Informationen von irgend einem Host aus dem Internet liefern. Diese bei einer Analyse des lokalen Netzwerkes zu *tracen*, macht eher wenig Sinn.

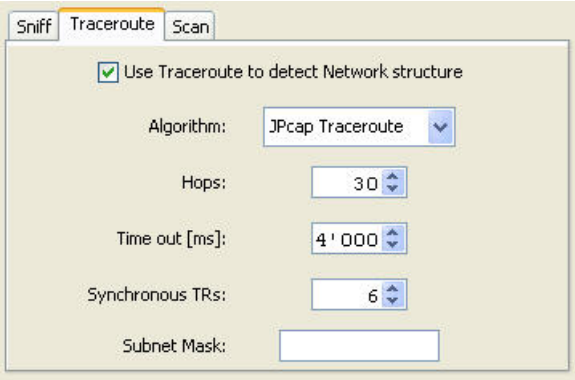

Abbildung 10: Das Traceroute Panel

4.3 Scanner

Sollen die gefundenen Hosts gescannt werden, so muss dies explizit gewählt werden. Zudem besteht wie bei den Traceroutes, die Möglichkeit, bloss Hosts eines Subnets zu scannen. Dies ist ausdrücklich empfohlen! Denn das Scannen eines unbekannten, fremden Hosts ist äusserst problematisch, denn es wird als unfreundlich Art empfunden und kann sogar rechtliche Folgen haben. Zudem ist der Vorgang sehr zeitintensiv. Auch wenn auf das Scannen verzichtet wird, können einzelne Hosts zu einem späteren Zeitpunkt im Studio GUI noch manuell gescannt werden.

5 Ausblick und Konklusion

5.1 Bekannte Probleme

TraceTCP für Windows funktioniert mit Wireless LAN auf meinem Laptop nicht. Das Programm kann die Netzwerkverbindung nicht nutzen. Da keine Tests auf anderen Geräten gemacht wurde, kann nicht ausgeschlossen werden, dass dieser Fehler nicht am Programm sondern an meinem Laptop liegt. Diese Fehlfunktion sollte nicht so tragisch sein, da einerseits Ausweichmöglichkeiten bestehen und zudem erwartet werden kann, dass dieser Fehler von Seiten der Entwickler von TraceTCP künftig korrigiert wird.
Andere schwerwiegende Probleme sind bis jetzt nicht bekannt. Das soll nicht heissen, dass es keine geben kann, sondern eher, dass sie noch nicht entdeckt wurden. Zudem konnten einige Funktionen nicht getestet werden, wie z.B. das synchrone Sniffen auf zwei Netzwerkkarten. Es konnte auch nie ein ganzes, grösseres Netzwerk ausgiebig, d.h. mit Sniffer, Traceroutes und Scanner, analysiert werden. Deshalb möchte ich mich von der Behauptung, es gäbe keine Fehler, distanzieren. Ich hoffe jedoch, dass auch tatsächlich keine grösseren und schlimmeren Probleme existieren.

5.2 Einschränkungen / Limiten

Mit dem momentan verwendeten Verfahren von Sniffen und anschliessenden Traceroutes kann bloss die Netzwerkstruktur aus der Sicht des Hosts, auf welchem VeriNeC ausgeführt wird, entdeckt werden. Dies entspricht meistens bloss einem Baum. Ein Netzwerk ist jedoch eher ein Graph mit Zyklen und kein Baum. Daher ist das aktuelle Resultat einer Analyse nicht 100% komplett. Zudem kann jeweils nur ein Netzwerkinterface eines Routers aufgelöst werden, nämlich jenes, mit dem eine Nachricht zum Sender, d.h. zum Host auf welchen VeriNeC läuft, beantwortet wird. Ein zweites kann momentan nicht identifiziert werden.

5.3 Verbesserungen und Erweiterungsmöglichkeiten

Um die Netzwerkstruktur besser abbilden zu können, könnte man versuchen, die Routingtabellen der Router zu erhalten, diese Informationen aufzuarbeiten und die Struktur damit zu ergänzen.
Eine weitere Möglichkeit wäre, eine Version des Importers, welche eigenständig läuft, zu implementieren, diese auf anderen Hosts im Netzwerk zu starten und dann z.B. mittels RMI diese anzusprechen, die Resultate zentral zu sammeln und zusammenzufügen. Diese Variante löst zwar das Problem nicht perfekt, denn ein solcher verteilter Importer würde wiederum nur die Netzwerkstruktur aus seiner Sicht liefern, doch wären nicht all zu viele Modifikationen am aktuellen Importer notwendig und es könnten wesentliche Teile übernommen werden.
Einen Punkt, den man unbedingt verbessern sollte, ist die Verwaltung des Netzwerkgraphen und dessen Abbildung im GUI. Zur Zeit wird die *HostCommRenderer*

Klasse, welche einen Host im Importer darstellt, so erweitert, dass man diese untereinander verbinden kann. Diese Variante ist zwar einfach aber nicht sehr ausgefeilt. Viel besser wäre, wenn ein bereits existierendes Framework, welches den Umgang mit Graphen ermöglicht, verwendet würde. Leider sprengte dies den Umfang dieser Arbeit.
Des weiteren könnte man einige Details im GUI verbessern. So wäre es toll, wenn beim Vergrössern resp. Verkleinern des aktuell sichtbaren Ausschnittes der Sichtbereich nicht verschoben würde. Zudem wäre es wünschenswert, wenn bei allen Zoomstufen gleich viele Hosts darstellbar sind. Zur Zeit sind bei einer Auflösung von 25% mehr Hosts darzustellen als bei 100% und beim Wechseln von einer kleineren auf eine grössere Stufe können einzelne Objekte verschoben werden.

5.4 Persönliche Konklusion

Die Bilanz fällt sicher sehr positiv aus. Die Arbeit bedingte es, viele unterschiedliche Technologien anzuwenden. So lernte ich sehr viel über IP und ICMP, sowie das Erstellen, Senden, Empfangen und Analysieren dieser Pakete. Daneben verwendete ich XML und Java Swing. Auch im Bereich Concurrent Programming lernte ich einiges dazu. Zudem ist das Software Engineering nicht zu vergessen. Auch hier konnte ich viel profitieren, musste aber zugleich einsehen, wie schwer es ist, vorhandene Strukturen zu verwenden und selbst Neues clever aufzubauen. Ich würde sagen, ich habe mir bei dieser Arbeit grosse Mühe gegeben und sehr viel Zeit investiert, aber auch Spass daran gehabt. Doch würde ich ein zweites Mal anders vorgehen und vielleicht ein etwas perfekteres Endresultat erhalten. So wollte ich immer gleich Resultate sehen. Es wäre intelligenter gewesen, nicht gleich auf das Endziel los zu rennen, sondern mehr Zeit in eine profunde Analyse und Planung zu investieren. Ich unterschätzte auch teilweise den Zeitaufwand enorm. Nachdem das Traceroute Package implementiert war, dachte ich, die Arbeit nun sehr bald abschliessen zu können. Jedoch irrte ich mich gewaltig, denn zu diesem Zeitpunkt begann der richtig schwierige und sehr zeitaufwändige Teil erst. Für die Integration in den bestehenden Importer brauchte ich sehr viel Zeit. Ich verlor immer wieder sehr viel Zeit um herauszufinden, wie dieses und jenes in VeriNeC funktioniert, wo, wie und wofür ich welche Methoden benutzen kann und schlussendlich wie diese funktionieren.
Jedoch denke ich, die Aufgabenstellung gut erfüllt und das VeriNeC Importer Modul erfolgreich erweitert zu haben. An dieser Stelle möchte ich David Buchmann und Dominik Jungo recht herzlich dafür danken, dass sie mir grossen Freiraum in der Implementation liessen und mich doch stets mit guten Ratschlägen unterstützt und sich für meine Probleme immer reichlich Zeit genommen haben.

Literatur

[1] Telecommunications, Networks & Security Research Group, University of Fribourg. http://diuf.unifr.ch/tns

[2] David Buchmann, Dominik Jungo *Verinec Translation Module (Verified Network Configuration - Working Paper)* http://diuf.unifr.ch/tns/projects/verinec/verinec_report.pdf

[3] Patrick Aebischer *Network Sniffer, Ein Modul für Verinec.* Bachelorarbeit 2004, TNS Research Group, Universität Fribourg.

[4] Andrew S. Tanenbaum. *Computernetzwerke.* Pearson Studium, 4. Auflage, 2003

[5] Christian Ullenboom *Java ist auch eine Insel.* Galileo Computing, 5. Auflage, 2006

[6] Elliote R. Harold, W. Scott Means *XML in a Nutshell.* O'Reilly, 3. Auflage, 2005

[7] David Flanagan *Java in a Nutshell.* O'Reilly, 4. Auflage, 2003

[8] RFC-Editor Webpage, http://www.rfc-editor.org/ (Letzter Besuch: 25.8.2006)

[9] Manuela Jürgens. *LATEX — eine Einführung und ein bisschen mehr.* Gesamthochschule in Hagen, Universitätsrechenzentrum Abt. Wissenschaftliche Anwendungen

A Modifikationen in bestehenden Klassen

Die meisten Klassen des bisher bestehenden Importers von [3] wurden zum Teil angepasst, um das Traceroute zu integrieren und die erhaltenen Resultate besser verwalten zu können. Auf alle Einzelheiten soll an dieser Stelle nicht eingegangen werden, da relativ viele Klassen betroffen sind und einige Änderungen bereits in 3.3 erwähnt wurden.
Zudem wurde die Klasse *NwBinding.java* aus dem Package *verinec.gui.core* leicht modifiziert. Die Methode *getWire()* wurde auf *public* gesetzt. Dies ist nötig, damit der *AnalysisThread* beim Erstellen der *PCNode-Objekte* für das Studio GUI allfällig vorhandene Nodes mit neu gefundenen verknüpfen kann, d.h. diese über eine Netzwerkverbindung verbinden.

B UML Diagramme

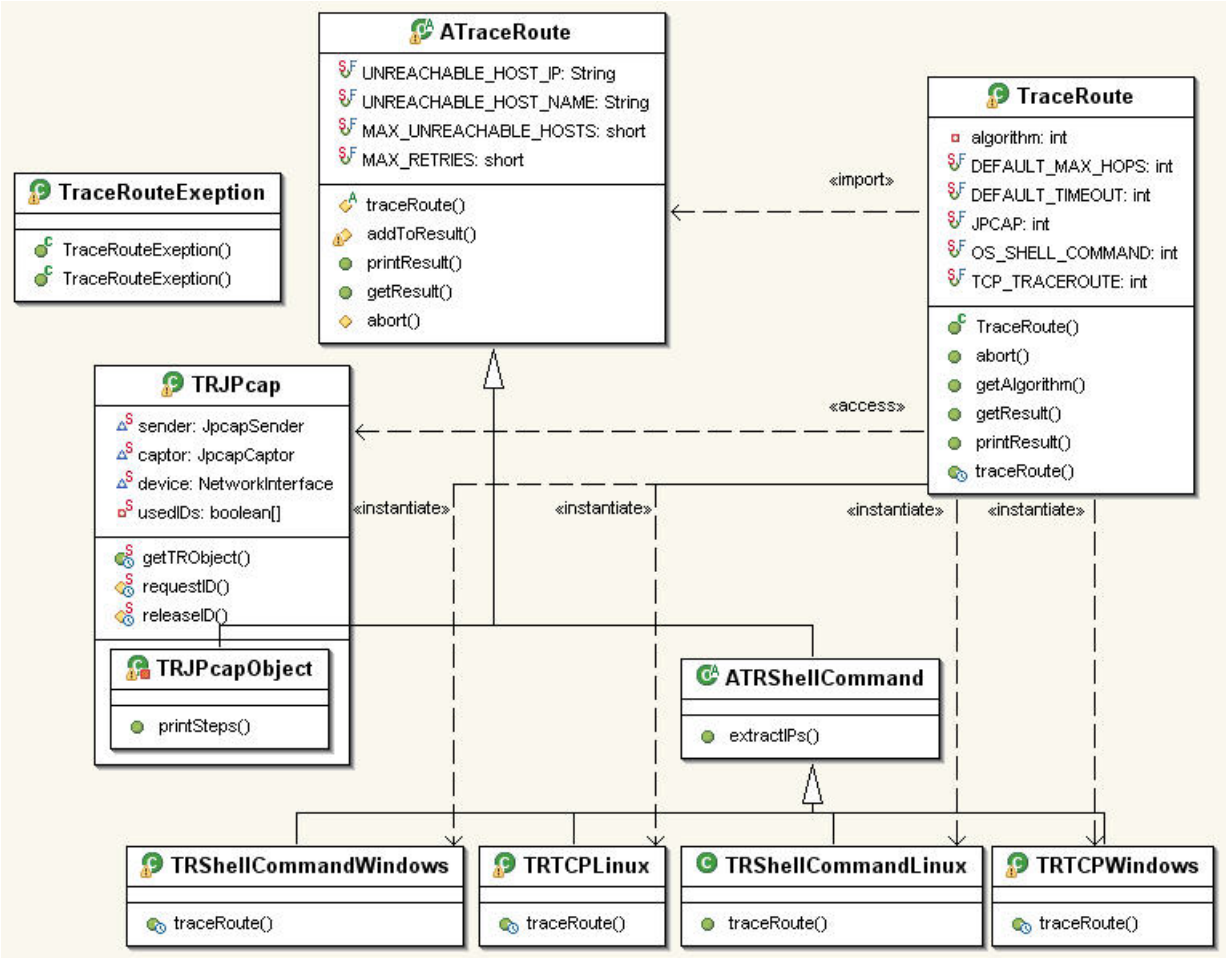

Abbildung 11: Das Traceroute Package

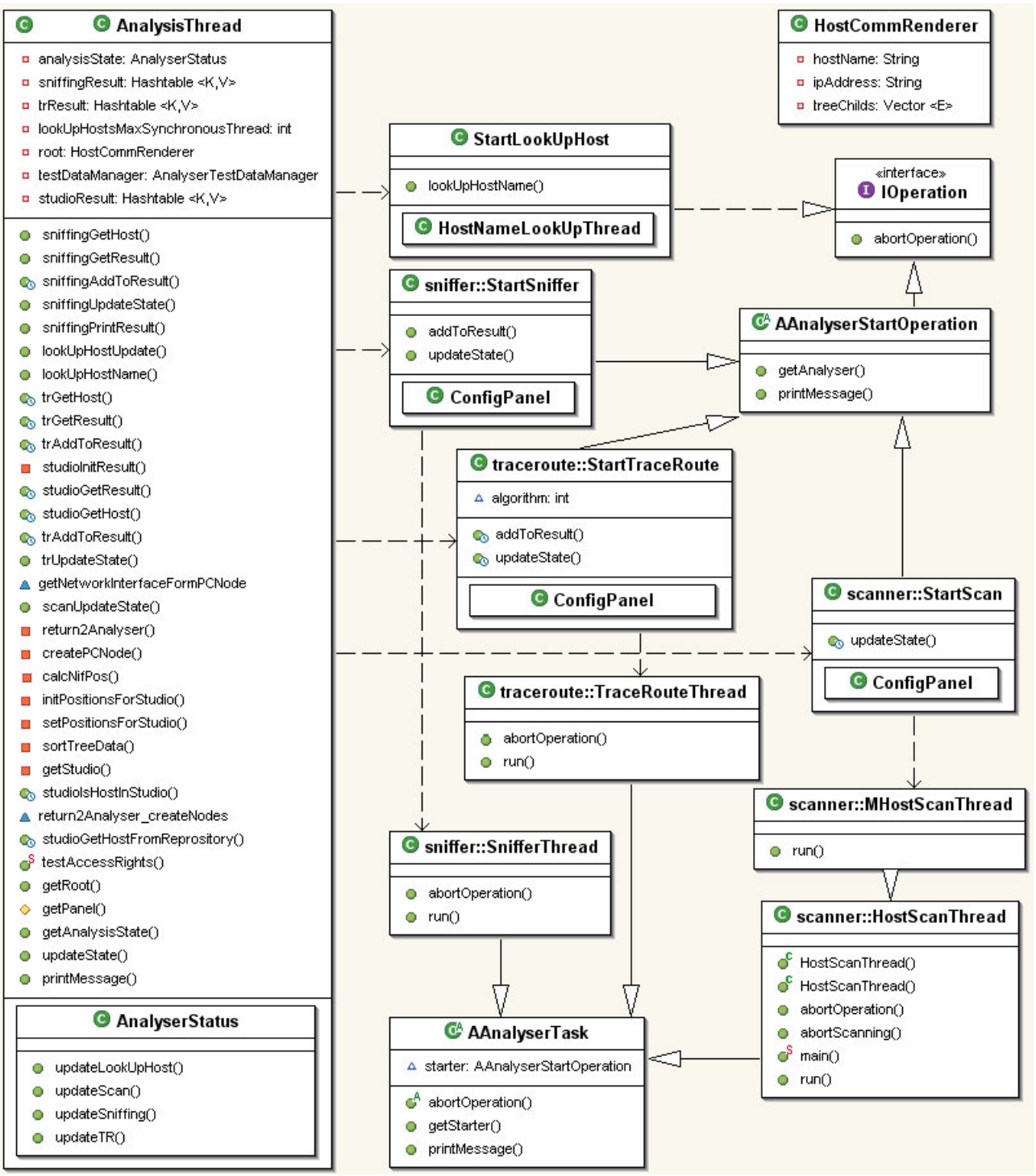

Abbildung 12: Das Importer Package

C Beigelegte CD

Die beigelegte CD ist wie folgt aufgebaut:

doc : Dokumentation zum Importer Modul. Diese beinhaltet die Java-Doc, also die Dokumentation der API des Importers. Des Weiteren ist auch dieses Arbeit als PDF-Datei, sowie der Latex Source Code zu finden.

verinec : Das gesamte VeriNeC Projekt. Es lässt sich sehr einfach als Java-Projekt in Eclipse importieren. Wird dies gemacht und soll es anschliessend ausgeführt werden, so müssen zuerst einige *built.xml* Ant-Dateien ausgeführt werden. Dies befinden sich in *./*, *Core/java*, *Adaption/java*, *Importer/java* und *Validation/java*. Danach kann die Klasse *VerinecStudio.java* aus dem Package *verinec.gui* als Java-Applikation ausgeführt werden.

bin : Dieser Ordner befindet sich in *verinec/Importer/java* und beinhaltet die Source-Codes und Binärdateien der benötigten Programme und Bibliotheken. Dies sind:

- `jpcap`: [10] Der Java Wrencher für Winpcap resp. libpcap. Für Windows ist die Installationsroutine *JpcapSetup_v5.1.exe* vorhanden. Soll JPcap unter Linux verwendet werden, so muss der C Source Code im Ordner *jpcap-0.5.1-source/src/c* neu kompiliert werden. Danach muss die erstellte Bibliothek und zudem die JAR-Datei *jpcap.jar* aus *jpcap-0.5.1-source/lib* für Java auffindbar sein.
- `libpcap`: [11] Bibliothek für das direkte Senden und Empfangen von Daten Paketen. Damit die Bibliothek installiert wird, muss in korrekter Reihenfolge *configure*, *make* und *install* *** TODO *** in *libpcap-0.8.1-source*, aufgerufen werden.
- `nmap-3.50-linux`: [12] Das Analyse Tool Nmap für Linux. Sollte die bereits vorhandene Binär-Datei nicht ausführbar sein, so müsste diese neu kompiliert werden, der Source Code und das Makefile befindet sich im Unterordner *nmap-4.11-source*.
- `nmap-4.03-win32`: [13] Das Analyse Tool Nmap für Windows. Der Source Code liegt zwar bei, sollte aber nicht benötigt werden.
- `tcptraceroute`: [14] Das TCP Traceroute Programm für Linux. Um dieses benutzen zu können, muss der Source Code, der sich in *tcptraceroute-1.4-source* befindet, kompiliert werden und die erhaltene Binärdatei in diesen Ordner kopiert werden. Damit die Kompilation erfolgreich sein kann muss zuerst die Bibliothek libpcap installiert sein.

[10] http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html
[11] http://www.tcpdump.org
[12] http://www.insecure.org/nmap
[13] http://www.insecure.org/nmap
[14] http://michael.toren.net/code/tcptraceroute/

- `tracetcp-0.99.4beta`: [15] Das TCP Traceroute Programm für Windows. Der Source Code liegt zur Ansicht im Ordner *tracetcp-0.99.4beta.source* bei, sollte aber sonst nicht benötigt werden.
- `winpcap`: [16] Dies ist das Pendent zu libpcap für Windows. Winpcap lässt sich sehr einfach mit der Installationsroutine *WinPcap_3_1.exe* installieren.

[15] http://tracetcp.sourceforge.net/
[16] http://www.winpcap.org/