Technische Referenz

Technische Referenz zum
Unified Intrusion Detection Management System
Prof. Dr. W. Kowalk
Department für Informatik
Universität Oldenburg

von

Bornhold, Johannes
Kamma, Martin
Kupperschmidt, Jochen
Meyer, Stefan
Müllner, Nils
Pölking, Dennis
Schulte, Kai

Betreuer:

Dipl.-Inform. Rüdiger Busch

Tag der Anmeldung: 1. Oktober 2004
Tag der Abgabe: 30. September 2005

Wir erklären hiermit, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Oldenburg, den 13. Oktober 2005

Inhaltsverzeichnis

1. Einleitung

Die Technische Referenz bildet eine detaillierte Beschreibung des unIDS Systems sowie gleichzeitig die Projektdokumentation der Projektgruppe „Intrusion Detection Management“.

2. Überblick über das unIDS System

2.1 Architektur

2.1.1 Vergleich zum alten unIDS

2.1.2 Änderungen an der Architektur

2.2 Kommunikationskonzept

Für die Kommunikation zwischen den einzelnen Komponenten des unIDS-Systems werden XML-Nachrichten verschickt. Dabei sind zwischen Agent, Proxy und Mittelschicht auf der einen Seite und Mittelschicht und grafischer Benutzerschnittstelle auf der anderen Seite leicht unterschiedliche Konzepte im Einsatz. Dies liegt darin begründet, dass sich auf Grund von möglichen Proxies auch mehrere Agenten hinter einer Verbindung befinden können. Dies ist auf der Seite der grafischen Benutzerschnittstelle nicht erforderlich, da hier an jeder Verbindung immer nur genau eine Benutzerschnittstelle beteiligt ist. Dieser Aufbau ist in Abbildung 2.1 schematisch dargstellt.

2.2.1 unIDS-Umschlag

Da bei der Kommunikation zwischen Agent, Proxy und Mittelschicht nicht von der Verbindung auf den Empfänger der Nachricht geschlossen werden kann, müssen die Nachrichten zusätzliche Informationen über den Absender und den Empfänger enthalten. Hierzu wurde das Konzept eines Umschlags, in den die Nachrichten verpackt werden, entwickelt. Aller Nachrichten zwischen Agent, Mittelschicht und Proxy müssen in einen solchen Umschlag verpackt werden.

Da die Nachrichten alle auf XML basieren konnte das Konzept eines Umschlags recht intuitiv umgesetzt werden. Hierzu ein Beispiel für eine `GreetingMessage`-Nachricht, wie sie bei jeder neuen Verbindung zwischen den Endpunkten ausgetauscht wird:

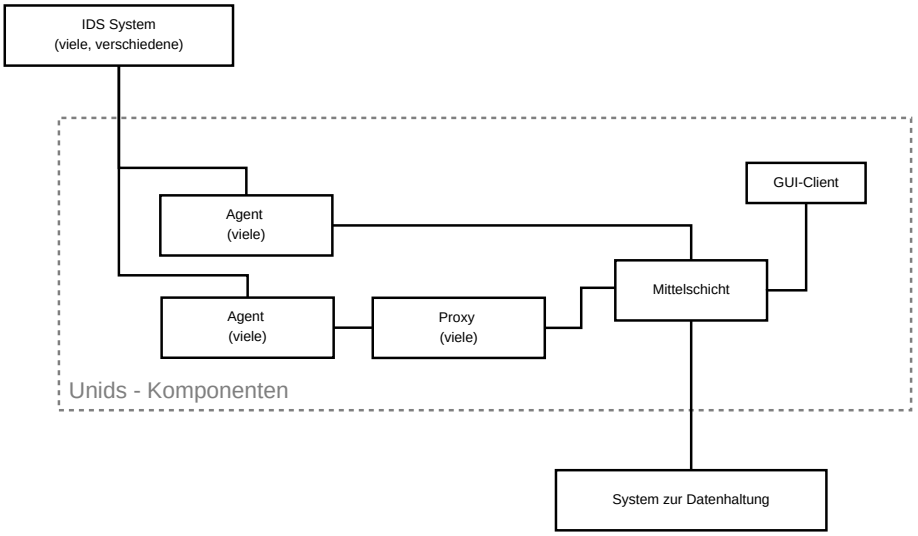

Abbildung 2.1: Aufbauschema des unIDS-Systems

```
<unids sender="ID" recipient="ID">
  <greeting/>
</unids>
```

Über die Attribute *sender* und *recipient* des Umschlags können alle beteiligten unIDS-Komponenten diese Nachricht richtig zuordnen und im richtigen Kontext weiterverarbeiten. Für den Absender und den Empfänger gelten die impliziten regeln, dass wenn kein Wert angegeben ist, implizit der direkte Endpunkt der Verbindung gemeint ist. Diese Eigenschaft wird dafür ausgenutzt, dass sich neue Komponenten, die noch keinen Identifikator erhalten haben trotzdem mit dem unIDS-System in Verbindung setzen können. Allerdings nur mit den direkt verbundenen Komponenten.

Als Gegenstück zur XML-Searialisierung des Umschlags existiert die Klasse `Message-Envelope` im Paket `unids.message.Message`. Über diese Klasse kann ein passender Umschlag erstellt werden, oder aber eine eingetroffene Nachricht kann in diese Implementierung verpackt und danach weiterverarbeitet werden.

2.2.2 Nachrichten zwischen Agent, Mittelschicht und Proxy

Auf der Ebene Agent, Mittelschicht und Proxy werden drei wesentliche Nachrichtentypen versandt: `StatusMessage`, `ConfigurationMessage` und `IdmefMessage`.

StatusMessage

Über die Status-Nachricht senden die verschiedenen Plugins des Agenten aktuelle Statusinformationen an die Mittelschicht. Diese Informationen werden in der Mittelschicht zwischengespeichert und an die grafischen Benutzerschnittstellen weitergeleitet und dort grafisch für die Nutzer des unIDS-Systems aufbereitet.

Statusnachrichten sind immer nach dem folgenden Schema aufgebaut:

```
<unids sender="ID" recipient="ID">
```

```
  <status plugin="NAME" device_id="ID">
    <dataset key="cpu" value="1">
      <data key="speed" value="3200" type="static"/>
      <data key="load" value="27" type="dynamic" max="100"/>
    </dataset>
    <dataset key="interface" value="eth0">
      <data key="mac" value="AA:BB:CC:DD:EE:FF"/>
      <data key="ipaddr" value="123.45.67.89"/>
    </dataset>
  </status>
</unids>
```

ConfigurationMessage

Die `ConfigurationMessage` wird genutzt, um Inforamtionen über die aktuelle Konfiguration von Agenten auszulesen, aber auch um diese Konfiguration zu ändern. Dazu sender der Agent bei jeder neuen Verbindung zum unIDS-System seine aktuelle Konfiguration und die Standardwerte für die einzelnen Parameter. Die Mittelschicht speichert den letzten Stand der Konfgiguration und Antwortet ggf. mit einer korrigierten Fassung, falls bestimmte Parameter abweichend von den Standardwerten gesetzt wurden. Die grafische Benutzerschnittstelle kann ebenfalls die Konfiguration in einer ganz leicht modifizierten Nachricht abrufen und Verändern. Die Mittelschicht sort dafür, dass die Änderungen der Gui so schnell wie möglich den entsprechenden Agenten erreichen.

Im folgenden ist ein Beispiel einer kurzen Konfigurations-Nachricht dargestellt:

```
<unids sender="ID" recipient="ID">
  <configuration>
    <plugins>
      <plugin name="Syslog" enabled="1"/>
      <plugin name="Snort" enabled="0"/>
      <plugin name="NetworkStat" enabled="1" update_interval="30"/>
    </plugins>
  </configuration>
</unids>
```

IdmefMessage

Idmef-Nachrichten sind in einem RFC definiert. Sie wurden in der unIDS-System neu eingeführt, damit IDS-Systeme die Idmef-Nachrichten ausgeben können möglichst einfach an das unIDS-System angebunden werden können. Auf ein Beispiel wurde an dieser Stelle verzichtet, da Idmef-Nachricht ausführlich im entsprechenden RFC dokumentiert sind.

2.2.3 Nachrichten zwischen Gui und Mittelschicht

Zwischen der grafischen Benutzerschnittstelle und der Mittelschicht werden wesentlich mehr verschiedene Nachrichtentypen und auch Anragetypen ausgetauscht, da

die Mittelschicht und die grafische Benutzerschnittstelle sehr eng zusammenarbeiten müssen. Zunächst wird der Anfragemechanismus vorgestellt, über den verschiedene Anfragen auf bestimmte Entitäten möglich sind. Danach werden die weiteren Nachrichten vorgestellt.

2.2.3.1 Anfragetypen

Aus Sicht der grafischen Benutzerschnittstelle können auf den verschiedenen Entitäten des Datenmodells eine Reihe von Anfragen ausgeführt werden. Daher wurde ähnlich zu dem Konzept des Umschlags in der Kommunikation zwischen Agent, Mittelschicht und Proxy ein Umschlag für den Anfragetyp entwickelt. Die folgenden Anfragetypen sind bisher vorgesehen:

- **Get** – Mit einer Get-Anfrage können Entitäten abgefragt werden. Eine solche Anfrage wird in der Regel mit der zur angefragten Entität passenden Nachricht beantwortet oder es wird eine Fehlernachricht geliefert, die Informationen darüber enthält, warum die Entität nicht angefordert werden konnte. Innerhalb der Get-Anfrage wird nur ein Rumpf der jeweiligen Entität angegeben, dies sieht für die Anfrage eines bestimmten Benutzers wie folgt aus:

  ```
  <get>
    <user id="user/5"/>
  </get>
  ```

  Die Mittelschicht antwortet hierauf mit einer Benutzer-Nachricht oder aber einem Fehler.

- **Add** – Die Add-Anfrage ist zu verwenden, wenn neue Entitäten angelegt werden sollen. Der Inhalt der Add-Anfrage ist in der Regel die zu der neu anzulegenden Entität passende Nachricht mit einem leeren Identifikator. Die Mittelschicht beantwortet solche Anfragen immer mit einer Statusnachricht, im Erfolgsfall vom Typ OK, im Fehlerfall ein möglichst passender Fehlerstatus. Für das Beispiel eines neuen Benutzers sähe dies folgendermaßen aus:

  ```
  <add>
    <user isadmin="false">
      [...]
    </user>
  </add>
  ```

  Die Punkte in den eckigen Klammern sind entsprechend durch weitere Angaben zu dem neu anzulegenden Benutzer zu ersetzen.

- **Update** – Beim Update handelt es sich fast um die gleiche Funktionalität, wie bei der Add-Anfrage. Der einzige wesentliche Unterschied ist, dass schon ein Identifikator für die zu aktualisierende Entität angegeben ist. Die Unterscheidung zwischen Add und Update wurde eingeführt, damit eine fehlerhafte Verwendung der Anfragetypen nicht unerkannt bleibt. Für das Benutzerbeispiel sieht die Anfrage folgendermaßen aus:

```
<update>
  <user id="user/2" isadmin="false">
     [...]
  </user>
</update>
```

- **Delete** – Mit der Delete-Anfrage können Entitäten aus dem Datenmodell gelöscht werden. Hier wird ähnlich wie bei der Get-Anfrage nur ein Rumpf der Entität angegeben, der über den Identifikator zugeordnet werden kann. Für das Benutzerbeipiel ergäbe sich die folgende Nachricht:

  ```
  <delete>
    <user id="user/5"/>
  </delete>
  ```

- **Register** und **Unregister** – Da das Datenmodell des unIDS-Systems recht komplex werden kann wird die grafische Benutzerschnittstelle nur über relevante Bereiche des Datenmodells auf dem Laufenden gehalten. Hierzu muss die grafische Benutzerschnittstelle sich für die relevanten Entitäten anmelden und bei erloschendem Interesse wieder abmelden. Zu diesem Zweck wurden die Anfragetypen Register und Unregister hinzugefügt. Für das Benutzerbeispiel ergäbe sich das Folgende:

  ```
  <register>
    <user id="user/5"/>
  </register>
  ```

  Falls die angegebenen Entitäten nicht existieren, wird von der Mittelschicht ein Fehlerstatus geliefert. Ansonsten wird die Nachricht mit einer einfachen OK-Nachricht beantwortet.

2.2.3.2 Datennachrichten

Für den Austausch von Inforamtionen über bestimmte Entitäten des unIDS-Systemzustands wurden eine Reihe von Datennachrichten definiert. Die wichtigsten werden nun vorgestellt.

Benutzernachrichten

Benutzerobjekte repräsentieren die Personen mit Zugriffsberechtigung auf das unIDS-System. Die Benutzernachrichten sehen folgendermaßen aus:

```
<user id="identifier" isadmin="bool">
  <name>name</name>
  <login>login</login>
  <description>beschreibung</description>
  <viewable-objects>
    <element id="identifier" type="type"/>
    <element id="identifier" type="type"/>
  </viewable-objects>
</user>
```

Im oberen Teil der Nachricht sind Informationen über den Benutzer enthalten, wie sein Name, sein Login oder eine allgemeine Beschreibung. Im Bereich `viewable-objects` ist eine Liste der für den Benutzer sichbaren Entitäten angegeben[1].

Element-Nachrichten

Die Struktur der Subnets und Devices und die Struktur des unIDS-Systems selbst werden aus einzelnen Element-Nachrichten aufgebaut, die für alle möglichen Entitäten innerhalb der Struktur stehen können. Einen einzelne ElementNachricht sieht dabei wie folgt aus:

```
<element type="device" id="device_2">
 <properties>
  <entry key="name"><![CDATA[Arbi]]></entry>
  <entry key="comment"><![CDATA[beschreibung hier]]></entry>
 </properties>
</element>
```

Struktur-Nachrichten

Aus den einzelnen Element-Nachrichten werden dann die verschiedenen im unIDS-System vorhandenen Strukturen zusammengesetzt:

```
<structure type="unids_system">
  <element type="middlelayer" id="middlelayer_0">
    <properties>
      <entry key="name"><![CDATA[Arbi]]></entry>
      <entry key="comment"><![CDATA[beschreibung hier]]></entry>
    </properties>
    <element type="agent" id="agent_1">
     <properties>
      <entry key="name"><![CDATA[Arbi]]></entry>
      <entry key="comment"><![CDATA[beschreibung hier]]></entry>
    </properties>
      <element type="device" id="device_12"/>
    </element>
    <element type="proxy" id="proxy_2">
      <properties>
        <entry key="name"><![CDATA[Arbi]]></entry>
        <entry key="comment"><![CDATA[beschreibung hier]]></entry>
      </properties>
    </element>
  </element>
</structure>
```

[1] Die Zugriffsberechtigungen von Benutzern konnten in der Version 1.0 des unIDS-Systems nicht vollständig fertiggestellt werden.

Funktionen aufrufen

Die Mittelschicht bietet der grafischen Benutzerschnittstelle eine Liste von Funktionen, die über die grafische Benutzerschnittstelle aufgerufen werden können. Hierzu sind die folgenden Nachrichten definiert:

```
<function name="FUNKTIONS_NAME"/>

<!-- Mit Parametern -->
<function name="FUNKTIONS_NAME">
  <parameter>
    ...
  </parameter>
</function>
```

Status-Nachrichten

Alle Anfragen der grafischen Benutzerschnittstelle werden wenn nicht eine spezielle Antwort gefordert ist mit einer Status-Nachricht beantwortet. Diese Nachrichten sehen folgendermaßen aus:

```
<reply>
  <status>STATUS_CODE</status>
  <description>DESCRIPTION</description>
  <!--+ Optional: Referenzen.
      | Insbesondere bei Add oder Delete Requests kann so aus
      | der Antwort alleine darauf geschlossen werden,
      | welche Entitäten verändert wurden und welche nicht.
      | Konfigurations-Parameter: reply.with.reference
      +-->
  <reference><element ... /></reference>
</reply>
```

Gui-Text

Die grafische Benutzerschnittstelle hat zusätzlich die Möglichkeit beliebige Zeichenketten unter von ihr verwalteten Schlüsseln abzulegen. Dies wird z.B. für das Merken der Fenstergröße genutzt:

```
  <gui-text id="">
    <text>...</text>
  </gui-texts>
```

2.2.4 IDMEF

Das Intrusion Detection Message Exchange Format (IDMEF) ist ein standartisiertes Data-Format das IDS-Systeme benutzen können um Nachrichten zu versenden. Der Standart wurde, beziehungsweise wird entwickelt um es zu ermöglichen das verschiedene Systemen, egal ob sie kommerziell oder Open-Source sind, ihre Daten auch untereinander austauschen zu können. Auch um des den Benutzern zu ermöglichen verschiedene Systeme miteinander einzusetzen. Für genauere Informationen siehe auch http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-14.txt.

2.2.5 IDXP

Quelle: http://www-rnks.informatik.tu-cottbus.de/de/materials/ws2002seRecent/krauz.pdf [2]
„Das Intrusion Detection Exchange Protocol (IDXP) ist ein verbindungsorientiertes Protokoll der Anwendungsschicht. Es wurde von der IDWG für den Austausch von Daten zwischen verschiedenen IDS ausgearbeitet. Dieser Datenaustausch findet hauptsächlich zwischen Analysierer und Manager des IDS statt. Primär realisiert es den Austausch von IDMEF-Nachrichten, kann aber auch einfachen Text oder binäre Daten übertragen. Das IDXP spezifiziert ein BEEP - Profil und hat dadurch die gleichen Eigenschaften und Vorteile, wie das BEEP-Framework generell. Authentifikation, Integrität, Zuverlässigkeit und andere wichtige Punkte des Protokoll-Designs müssen nicht neu implementiert werden, sondern können ohne großen Aufwand einfach, durch das Einbinden vorhandener BEEP-Profile (z.B. TLS oder SASL), übernommen werden. Das IDXP-Profil ist bei der IANA unter http://iana.org/beep/transient/idwg/idxp registriert. Es muß also nur in der Start-Nachricht, während des Kanalaufbaus zwischen dem Client und dem Server, vereinbart werden."

2.2.6 BEEP

Quelle: http://www-rnks.informatik.tu-cottbus.de/de/materials/ws2002seRecent/krauz.pdf [3]
„Das Blocks Extensible Exchange Protocol (BEEP) ist ein verbindungsorientiertes, asynchrones Protokoll-Framework der Anwendungsschicht. Es wurde entworfen, um den Aufbau von Netzwerk- Anwendungs-Protokollen zu vereinfachen und zu verbessern. BEEP liefert einen Mechanismus, der verschiedene Problemfelder des Protokoll-Designs, in eine Menge von wiederbenutzbaren Modulen integriert. Diese Module werden Profile (profiles) genannt. Immer wieder auftauchende Probleme des Protokoll-Designs sind z.B. Authentifikation, Verschlüsselung, Kompression oder die Fehlerbehandlung. Diese Elemente werden im BEEP in die wiederbenutzbaren Module gepackt und müssen dadurch nicht immer wieder neu implementiert werden. Das ist auch der entscheidende Vorteil gegenüber herkömmlichen Protokollen. Das Protokoll setzt auf ein vorhandenes Transportprotokoll auf, was im RFC 3081 (Mapping the BEEP Core onto TCP) am Beispiel TCP veranschaulicht wird. BEEP dient als Grundlage, des von der IDWG beschriebenen Intrusion Detection Exchange Protocols (IDXP), ist beschrieben im RFC 3080 und wurde im März 2001 von der BEEP Working Group vorgestellt."

2.2.7 Twisted

`Twisted` ist ein *event-driven networking framework* für Python, das im wesentlich bedeutet, dass twisted ereignisorientiert arbeitet. Für den Benutzer bedeutet dies, dass er so genannte *callbacks* schreibt, welche vom Framework aufgerufen werden. . Die verschiedenen Twisted-Projekte unterstützen im Moment TCP, UDP, SSL/TLS, multicast, Unix-Sockets sowie eine große Anzahl von Protokollen (HTTP, NNTP, IMAP, SSH, IRC, FTP, usw.).

Weitere Informationen zu Twisted sind auf der Originalwebseite: www.twistedmatrix.com, sowie bei wikipedia: http://en.wikipedia.org/wiki/Twisted zu finden.

[2] Stand: 13.10.2005
[3] Stand: 13.10.2005

3. GUI

Die graphische Benutzungsoberfläche ist die zentrale Verwaltungskomponente des Benutzers in unIDS. Sie ist dafür zuständig, dem Nutzer eine Möglichkeit zu bieten, Einfluss auf alle anderen Komponenten von unIDS zu nehmen, sowie dem Benutzer Informationen über die Arbeit aller Komponenten zu geben.
Im Folgenden wird zunächst darauf eingegangen, welche genauen Anforderungen an die GUI gestellt werden und welche Verbesserungen zur vorigen Gruppe zu verwirklichen sind. Nach der Betrachtung folgt ein Entwurfs und Implementierungs Abschnitt, in dem die Umsetzung der Anforderungen erörtert wird. Im letzten Abschnitt erfolgt eine abschließende Betrachtung des implementierten Ergebnisses. Dabei wird auf erreichte und nicht erreichte Ziele eingegangen.

3.1 Zielsetzung und Anforderung

In der graphischen Benutzungsoberfläche als zentrale Verwaltungs- und Informationsschnittstelle zum Benutzer müssen viele verschiedene Möglichkeiten existieren, um auf das Geschehen angeschlossener Komponenten Einfluss zu nehmen, sowie über ihren aktuellen Zustand informiert zu werden. In diesem Abschnitt wird zunächst auf eine kurze Bewertung des alten Benutzerinterfaces eingegangen, sowie auf Probleme hingewiesen, die bei der Verwendung und Weiterentwicklung entstehen.

3.1.1 Zielsetzung Verbesserung

Eine Zielsetzung war es, die Benutzerschnittstelle, die innerhalb der Projektgruppe 2003 entstanden ist, weiterzuentwickeln. Diese Benutzerschnittstelle zeichnete sich im großen und ganzen dadurch aus, dass sie sehr unabhängig von anderen Komponenten konzipiert worden war. Die graphische Benutzerschnittstelle hat dabei sehr viele Aufgaben wahrgenommen. Die Verteilung und das Erfüllen dieser Aufgaben soll im Rahmen des Projektes einer weitgehenden Überarbeitung unterzogen werden. Darüberhinaus wird es eine Vielzahl von Erweiterungen geben. Genauere Details zum Vorgängersystem sind in der Projektausarbeitung **??** sowie in der Seminarausarbeitung **??** zu finden. Weiterhin wird darauf hingewiesen, daß sich alle Angaben ausschließlich auf die uns vorliegenden Dokumente und Programmversionen beziehen, welche zum Zeitpunkt der Arbeit vorlagen und eventuell privat weitergeführte

Verbesserungen des Systems oder die zum Zweck der Präsentation angelegte Demonstrationsversion nicht mit einbeziehen.

3.1.2 Systemeinteilung

Die vorgänger Version der Benutzungsoberfläche wurde in zwei bereiche aufgeteilt. Der erste Bereich bestand aus einem, das die direkte Benutzerschnittstelle und Manipulationsoberfläche des Systems darstellte, sowie ein Proxy, der für die Kommunikation mit der Datenbank des Systems verantwortlich war.
Dieses sollte ausgebessert werden. Ein Applet hat den Vorteil, dass es bei jeder Benutzung erneut heruntergeladen werden muss und damit immer eine aktuelle Version beim Benutzer vorliegt. Andererseits ist ein Applet, wenn es in eine Internetseite eingebunden ist, sehr stark durch die Sicherheitseinstellungen beschränkt. Es dürfen beim Applet keine Verbindungen zu fremden Systemen aufgebaut werden. Weiterhin besteht ein eingeschränkter Zugriff auf Ressourcen des Client-Systems. Aus diesen Gründen ist in der neuen Version der Benutzungsoberfläche eine Javaapplication vorgesehen. Um die Vorteile des Applets weiterhin nutzen zu können und gleichzeitig die genannten Nachteile zu umgehen, haben wir uns für das Java Network Launch Protokoll(JNLP) entschieden. Dieses System schließt die verwenung von JAVA Applicationen mit ein. Näheres hierzu wird in **??** beschrieben.

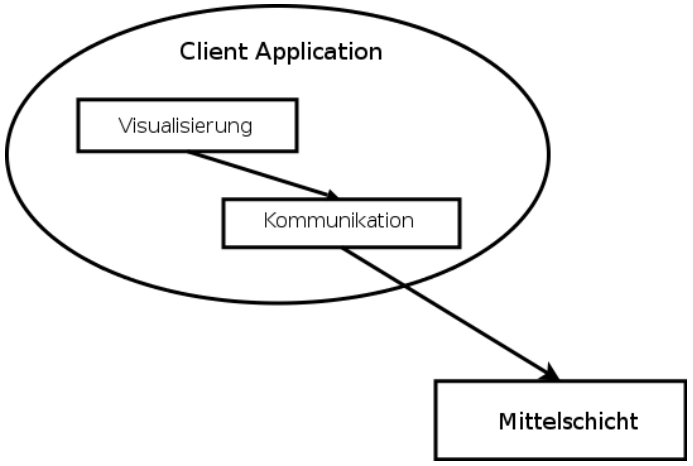

Abbildung 3.1: Aufbau unIDS-Architektur

3.1.3 Kommunikation

Die Kommunikation mit anderen Systemkomponenten wurde in der alten Benutzungsschnittstelle durch ein zweigeteiltes System realisiert. Bedingt durch die Verwendung eines Applets ist es nicht möglich, Verbindungen zu anderen Servern aufzunehmen als dem Server, von welchem das Applet geladen wurde. Dieses Design erfordert eine Weiterleitung aller Anfragen an die Datenbankschicht über eine Zweigstelle, da nicht gewährleistet wird, dass sich diese Systeme zwangsläufig auf dem selben Server befinden. Die Kommunikationsansicht wurde eigenständig als eine TCP-Socket-Verbindung vom Applet zum sogenannten GUI-Proxy implementiert. Die Verbindung vom GUI-Proxy zur Datenbankschicht erfolgte über eine normale JNDC mit Hilfe des MySQL Connectors/J.

Diese Kommunikation soll auch in der GUI durch eine Implementierung des BEEP-Frameworks ersetzt werden. Dieses erfordert einen sehr starken Eingriff in die Struktur der alten Benutzungsoberfläche, da die Verzahnung der Komponenten mit der Kommunikation seht stark ausgeprägt ist. Die Basis der neuen Kommunikation wird duch eine JAVA Implementierung des Beep - Frameworks ersetzt. Genauere Erklärungen des BEEP-Framework befindet sich in Kapitel **??**.

3.1.4 Datenhaltung

Die Datenhaltung des Benutzerschnittstelle wurde im alten System ausschließlich durch den GUI-Proxy realisiert. Er hatte die Aufgabe, bei einer Installation Benutzer und Gerätetabellen in der Datenbankschicht anzulegen. Dieser Proxy hatte die volle Verantwortung über die Benutzertabellen. Damit ist es für die Datenbank- und Analyseschicht nicht möglich gewesen, diese Daten in eine Auswertungen mit einzubeziehen. Ein weiterer Schwachpunkt war das Anfordern der verschiedensten Daten aus der Analyseschicht. Die angeforderten Daten wurden 1:1 in eine XML-Struktur umgewandelt und über den GUI-Proxy an die Benutzerschnittstelle weitergeleitet. Dadurch war eine Kapselung der Daten völlig ausgeschlossen.
Die Datenhaltung wird im Zuge der Überarbeitung eine grundlegende Änderung erfahren. Die gesamte Speicherung der Daten wird nur von der Datenhaltungs- und Analyseschicht(Mittelschicht) bestimmt. Weiterhin wird die Art der Speicherung direkt von der Nachrichtenerstellung getrennt. Näheres hierzu im Kapitel**??** und im Kapitel**??**.

3.1.5 XML Konzept

Das allgemeine XML-Konzept der Vorgängergruppe war die direkte Umwandlung einer MySQL-Struktur in eine XML-Repräsentation. Diese Struktur wurde allgemein über alle Ebenen in der Benutzungsoberfläche weitergereicht. Bevor es in die Darstellungsschichten gelangen konnte, wurde diese XML-Struktur in ein von der Vorgängergruppe implementieres Objekt umgewandelt. Diese Strukturelemente im Objekt waren folglich nur über direktem Wissen darüber, wie die Datenbanktabellen und deren einzelne Spalten benannt wurden, zugänglich. Daraus folgt, dass ein Java-Objekt, welches für die Visualisierung zuständig war, ein genaues Wissen über die genauen Datenbanktabellen haben musste, um eine Information exakt referenzieren zu können. Leider wurde in der Benutzungsoberfläche versäumt, diese Tabellen und Spaltennamen global abzulegen. Sie wurden stattdessen direkt mit Hilfe einer im Quellcode eingetragenen Zeichenkette ausgelesen. Damit war es unmöglich, Veränderungen an der Struktur vorzunehmen, da keine direkte Beziehung zwischen Objekten, die versuchten den gleichen Wert auszulesen, bestand. Die einzige Verbindung war die Gleichheit der Zeichenkette, die den Zugriff ermöglichte. Allerdings war diese Zeichenkette in den seltensten Fällen global eindeutig. In vielen Fällen wurde auch eine andere Technik benutzt. Hierzu wurden Informationen direkt im XML-Format in einer MYSQL-Datenbank gespeichert. Dabei handelte es sich um Zusammenstellungen von Netzwerkstrukturen, die mit Visualisierungselementen der graphischen Oberfläche angereichert waren. Die Zusammenstellung und Speicherung oblag einzig und allein der graphischen Benutzungsoberfläche. Dadurch ist es für die anderen Schichten nicht möglich gewesen, Kenntnis über das Ausmaß der eingetragenen Veränderungen zu gewinnen. Weiterhin gehen alle Vorteile, die durch Verwendung einer relationalen Datenbank entstehen, verloren. Es entstanden

somit kaum Tabellenverknüpfungen und Verbindungen innerhalb der Daten. Damit war es der Analyse- und Datenbankschicht nur bedingt möglich, eine Auswertung mit Hilfe der relationalen Datenbank und der Sprache SQL zu tätigen. Ein weiterer Nachteil dieser Methode ist der Zugriff von mehreren Benutzungsoberflächen. Da alle Daten immer ohne Rückfrage überschrieben wurden, ist eine Absicherung nicht möglich gewesen. Darüber hinaus entsteht bei dieser Methode unnötiger Datenverkehr. Bei jeder Änderung musste die gesamte Struktur samt redundantem unveränderten Inhalt gesendet werden. Dieses führt somit zu einem unnötigen Mehraufwand bei der Verarbeitung.
Dieser Aspekt musste in der Überarbeitung der GUI beachtet werden. Die vorgestellten weitreichenden Abhängigkeiten sollten durch eine Kapselung der einzelnen Schichten voneinander, sowie einer exakten Festlegung der einzelnen XML-Schemata realisiert werden. Weiterhin muss die Mittelschicht direkt über die Änderungen informiert werden, damit eine parallele Arbeit sichergestellt werden kann. Eine Reduzierung der redundanten Informationen bei einer Speicherung soll durch die ausschließliche Sendung von veränderten Daten realisiert werden. Ein ähnliches Schema gilt für die Abfrage von Daten. Hier soll durch eine explizite Anfrage der Aufruf spezifischer Daten möglich sein. Dieses gesamte System sollte möglichst einen logischen Aufbau haben, damit eine Erweiterung unter Verwendung bestehender Mechanismen ermöglicht wird.

3.1.6 Systeminformationsanzeige

Die Visualisierung der von den einzelnen Agenten gesammelten Informationen, wie CPU-Auslastung und Netzwerkverkehr, war der zentrale Gesichtspunkt des ursprünglichen unIDS -Systems. Hierfür wurde das Plugin-Konzept genutzt. Es ermöglichte, dynamisch das System um Netwerkansichten ohne Integrationsaufwand zu erweitern. Dabei wurden eine 2D-Raumplanansicht, eine 3D-Netzansicht, sowie die Nadelansicht als eine Form der Netzwerkansicht implementiert. Näheres hierzu siehe **??**.
Neben den Netzwerkansichten gibt es die Systeminformationsansicht im alten System. Hier muss ähnlich wie in der Netzwerkansicht von einer Plugin-Klasse abgeleitet werden. Diese Ansichten informieren den Benutzer über den Stand des aktuell gewählten Gerätes.
Diese Ansichten sollen ins neue System weitesgehend integriert werden. Dabei gilt es, besonders die 2-dimensionale Ansicht weiter auszubauen. Während sie vorher eine Art Netzwerkzusammenstellung mit Hintergrundbild war, soll sie nun zu einem aktiven Modelierungstool für Netzwerke ausgebaut werden. Hier sind Möglichkeiten vorgesehen, direkt durch Markieren von Subnetzen in dieses hineinzuzoomen sowie durch markieren von zwei Geräten eine virtuelle Verbindung zwischen den Geräten zu erstellen. Weiterhin sollen die ursprünglichen Modellierungsmöglichkeiten erhalten bleiben, sowie einige Eigenschaften aus dem Programm *Spectrum* des Herstellers *Aprisma* (www.aprisma.com) übernommen werden.

3.1.7 Installation

Im Ausgangsprodukt war eine vollständige Installation der Benutzerschnittstelle inklusive der restlichen Systemkomponenten nicht möglich. Um eine Installation erfolgreich durchzuführen, musste erst der Quellcode überarbeitet werden. Es wurden

unter anderem fehlerhafte Datenbanktabellen angelegt, sowie eine benutzerunfreundliche Installation mittels einer Konsole mitgeliefert.
Auch dieses sollte ausgebessert werden. Ziel ist es, den Benutzer durch die Installation zu führen und zu unterstützen. Daraus folgt, dass es eine geteilte Installation geben muss. Im ersten Teil werden sämtliche Komponenten der graphischen Benutzeroberfläche in einen sog. Web-Container kopiert. Auf diesem Web-Container kann nun der zweite Teil der Installation erfolgen. Die Daten, die im Web-Container vorliegen, müssen auf dem Client-System installiert und in die Systemumgebung integriert werden. Die vollständige Installation sollte dabei, möglichst ohne großen Aufwand zu bewerkstellien sein. Hierfür biete sich das JNLP-Web-Startsystem von Java an. Hier wird durch einfaches Auswählen und Starten der JNLP Datei eine Installationsroutine ausgelöst. Hierzu ist lediglich eine aktuelle Java-Version erforderlich. Dieses stellt keinen zusätzliches Aufwand dar, da zum Ausführen der Client-Software bereits eine Java-Version vorhanden sein muss. Genaueres hierzu im Kapitel Installation **??**.

3.1.8 Ergonomische Aspekte

Die alte GUI wies im Bereich der Benutzerführung große Mängel auf und sollte hierzu einer kompletten Überarbeitung unterzogen werden. Genaueres wird ausführlich in der Diplomantenarbeit **??** erläutert.

3.1.9 Pluginkonzept

In der vorigen Benutzungsoberfläche wurde ein Plugin-Konzept benutzt, um die GUI nachträglich zu erweitern. Dieses Plugin-Konzept hat die Möglichkeiten, zusätzliche Ansichten über das zu überwachende System bzw. Netzwerk zu integrieren.
Das vorhandenen Konzept sollte nicht nur verbessert werden, sondern auch um verschiedene Punkte erweitert werden. Für neue Implementierung ist ein automatisches Plugin-System geplant. Sobald ein Plugin dem System hinzugefügt wird, soll es auch ohne einen zusätzlichen Konfigurationsaufwand im System verfügbar sein. Weiterhin soll es z.B. die Möglichkeit geben, Plugins dynamisch in die GUI einzubinden.

3.1.10 Netzwerkaufbau

Das alte System beinhaltet einen sehr umständlichen Netzwerkaufbau, der nicht immmer intuitiv zu handhaben war. Beispielsweise musste ein Gerät als Wurzel deklariert werden. Hier wurde in einer hierarchischaufgebauten Baumstruktur eine Übersicht über das vorhandene Neztwerk dargestellt. Diese Einteilung wurde als nicht adäquat empfunden, da sie zum Beispiel bei einem Ring-Netzwerk Geräte mehrfach anzeigt.
Dieser Aufbau sollte durch eine Einteilung in Subnetze und Sub-Subnetze ersetzt werden. Ähnlich wie es im Programm *Spectrum* realisiert ist. Es sollte außerdem möglich sein, schnell ohne der Berücksichtigung von physikalischen Verbindungen eine Netzwerkeinteilung modellieren zu können.
Es sind zwei grundsätzlich verschiedene Netzwerkmodellierungen eingeplant. In der ersten Modedellierung gibt es die Möglichkeit, Subnetze und Geräte beliebig einzuteilen. Dieses wird anhand von Abbildung 3.2 dargestellt. Als Grundlage existiert das physikalische Netzwerk (Doppellinen) zwischen den einzelnen Geräten(PC 1-5). Darüber wird eine virtuelle Einteilung in Netze und Subnetze möglich sein (Subnetz

1 -3). Dabei können die Geräte beliebig verteilt sein. In einem Subnetz können weitere Subnetze sowie einzelne Geräte enthalten sein. Geräte wie in diesem Beispiel der „Drucker“, die noch von keinem Agenten überwacht werden können, werden als virtuelle Geräte in das Netzwerk eingefügt und können wie jedes andere Gerät in ein Subnetz eingeteilt werden. Das Gleiche soll für Verbindungen gelten. Sie sollen zwischen Geräten beliebig erstellt werden. Diese Modellierung soll Grundlage für eine Informationsansicht sein, in welcher man die Auslastungen des gesamten Netzwerkes beobachten kann. Eine zweite Ansicht ist die unIDS-Netzwerkansicht. Diese Ansicht wird in der Abbildung 3.3 illustriert. Diese Ansicht wird direkt aus der Mittelschicht von unIDS anhand der angemeldeten Agenten und Proxy-Systeme erstellt. Sobald ein Agent sich anmeldet, wird dieser automatisch mit samt der Geräte, die er überwacht, in die Netzwerkansicht eingetragen. Es ist geplant, dass die Mittelschicht beliebig viele Agenten und Proxys verwalten kann. Weiterhin muss jeder Proxy in der Lage sein mehrere Agenten zu überwachen. Darüber hinaus sollte jeder Agent die Fähigkeit haben mehrere Geräte zu überwachen.

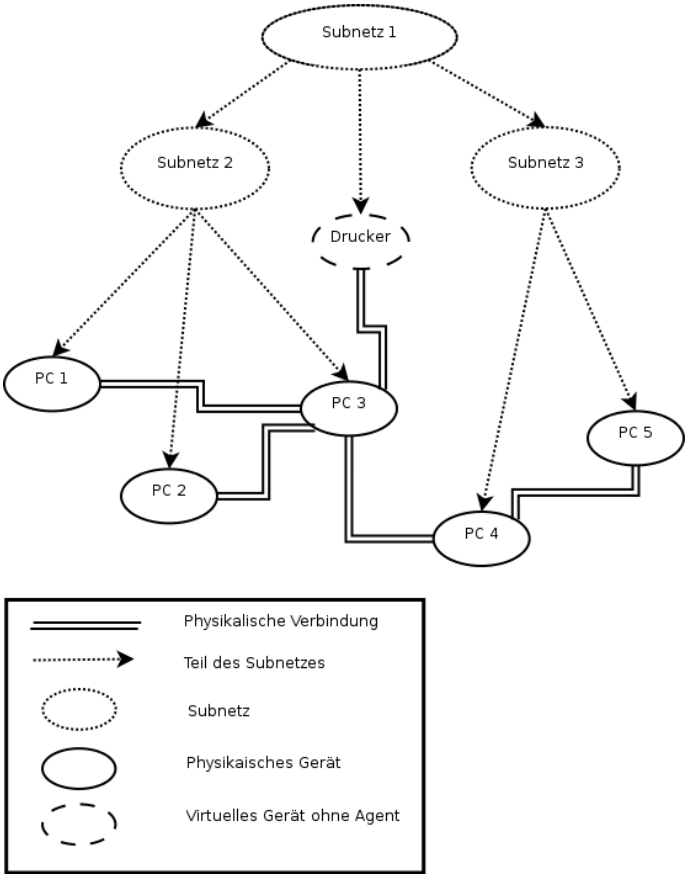

Abbildung 3.2: Aufbau Netzwerkstruktur

3.1.11 Internationalisierung

Im alten Programm wurde die Internationalisierung nur mäßig unterstützt. Sie war aus folgenden Gründen nicht nutzbar. Zum einen ist die Internationalisierung fest einprogrammiert worden. Im Installationsverzeichnis befindet sich eine Datei, die in Java geschrieben wurde und Sie beinhaltet fest einprogrammierte, durchnummerierte Zahlen, die einer Übersetzung zugeordnet wurden. Durch eine Verwendung von

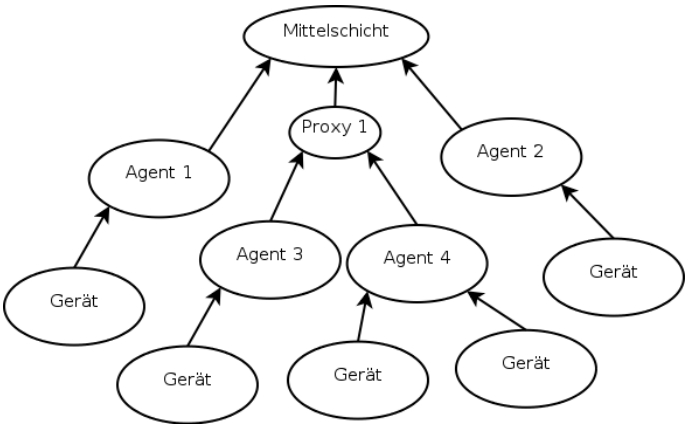

Abbildung 3.3: Aufbau unIDS Netzwerkstruktur

Zahlen ist es weiterhin nahezu unmöglich eine Assoziation zwischen dem Zahlenwert und der Übersetzung herzustellen. Dieses erschwert die Entwicklung und Überarbeitung des Systems stark, da alle Bezugspunkte durch eine Verwendung dieser Art verloren gehen.
Es soll möglich sein, verschiedene Sprachen auszuwählen und damit die Einstellungen an die eigenen Bedürfnisse anzupassen. Weiterhin muss das System in der Lage sein, eigenständig anhand der Systemumgebung eine Sprache aus den verfügbaren Sprachen eigenständig auszuwählen. Eine Erweiterung des Sprachumpfangs sollte weiterhin problemlos möglich sein, ohne eine Anpassung am Quellcode vornehmen zu müssen. Da es unmöglich ist, einer Zahl einen Übersetzungswert zuzuordnen, soll dieses System durch eine geordnetes System von statischen Variablen ersetzt werden, welche überall im Quellcode verfügbar sein sollen und eine aussagekräftige Bezeichnung erhalten. Näheres hierzu ist im Kapitel **??** zu finden.

3.1.12 Einflussnahme auf andere Schichten

Im alten System war es nur begrenzt bis gar nicht möglich, Einfluss auf die Funktionsweise anderer Systemkomponenten wie Mittelschicht und Agenten zu nehmen. Dieses sollte im Rahmen der Weiterentwicklung verbessert werden.
Es ist geplant, dass die Mittelschicht einen Satz von Befehlen und Einstellungsmöglichkeiten zur Verfügung stellt. Diese Befehle und Einstellungsmöglichkeiten sollen von der graphischen Benutzungsoberfläche aus bequem per Knopfdruck aktiviert werden können. Eine änliche vorgehensweise ist für den Agenten bereich geplant. Hier sollen alle Plugins separat einstellbar sein. Es ist geplant, dass jedes Plugin an und ausstellbar ist, sowie verschiedene parameter, die die Arbeitsweise des Plugins beeinflussen, über die Benutzungsoberfläche einstellbar sind.

3.1.13 2D Editor

Die 2D Raumansicht ist als eigentliches Herzstück der Benutzungsoberfläche geplant. Hier soll es möglich sein, die Ansicht auf ein Netzwerk zu modellieren. Diese Modellierungen sind als visuelle Grundlage zur Überwachung jedes einzelnen Netzbereiches gedacht. Jedem Gerät wird eine individuelle Ansicht des Inhaltes zu zugeortnet, so dass eine Navigieren durch die erstellte Struktur möglich ist. Eine zweite Aufgabe,

die der 2D Editor noch erfüllen sollte, ist das direkte Modellieren des Netzwerkes. Es sollte möglich sein, neue Geräte einzufügen. Diese Geräte können beliebeiger Art sein. Dabei kann es sich um virtuelle sowie nichtvirtuelle Geräte handeln. Um Geräte innerhalb des Netzwerkes zu verknüpfen soll es möglich sein Verbindungen, mit Hilfe des Editors zwischen Geräten anzulegen.

3.1.14 Oberflächenaufteilung

Die Oberfläche wird in sechs Bereiche eingeteilt. Dieses Bereiche haben jeweils verschiedene Aufgaben. Die Gesamtstruktur ist stark an bekannte Strukturen angelehnt, um einen möglichst hohen Wiedererkennungseffekt zu begünstigen. Im Folgenden werden die Aufgaben der sechs Bereiche kurz erläutert. Der schematische Aufbau der gesamten Oberfläche wird in Abbildung 3.4 gezeigt. Eine genaue Beschreibung der Funktionsweise und explizite Gründe werden in der Diplomarbeit **??** besprochen.

- *Menüzeile* Die Menüzeile beinhaltet Kurzeinträge für alle Unterbereiche der Benutzungsoberfläche. Von hier aus sollen alle Funktionen über eine alternative Ansteuerung verfügbar sein.

- *Buttonleiste* Die Buttonleiste beinhaltet Knöpfe mit Symbolen. Hier werden alle Funktionen deponiert, die für die alltägliche Arbeit mit dem System von besonderer Wichtigkeit sind.

- *Navigationsbaum* Im Navigationsbaum finden alle unIDS - Elemente Platz. Von hier kann bequem über einen strukturierten Baum auf die Elemente in ihrere jeweiligen Einordnung zugegriffen werden.

- *Pannelleiste* In der Panelleiste werden einzelne ausgewählte Inhalte angezeigt. Sie dient als Arbeits- und Anzeigefläche. Hier ist es möglich, über verschiedene Tabs auf bereits geöffnete inhalte zuzugreifen.

- *Historyanzeige* Die Historyanzeige dient als Dokumentaiontions- und Sammelstelle fuer eingehende Alarmnachrichten. Jede eingehende Nachricht erzeugt hier einen kurzen Eintrag. Von hier kann nun der Inhalt durch das Auswählen der Nachricht angesehen werden.

- *Statuspanel* Im Statuspanel werden Kurzinformationen über das letzte Ereigniss das eingetreten ist angezeigt.

3.1.15 Nichtfunktionale Anforderungen

- *Sicherheit* Das System soll ohne vorherige Authorisierung keinen Zugang gewähren. Weiterhin sollen die Manipulationsmöglichkeiten der administrativen Benutzergruppe von normalen Benutzern nicht verwendet werden können.

- *Zuverlässigkeit* Das System muss mit auftretenden Fehlern und Benutzerfehlern umgehen können, und außerdem muss ein problemloser Neustart bei schweren Fehlern möglich sein. Ebenso soll das System weiterhin einsatzbereit sein, wenn ein Teil ausfällt.

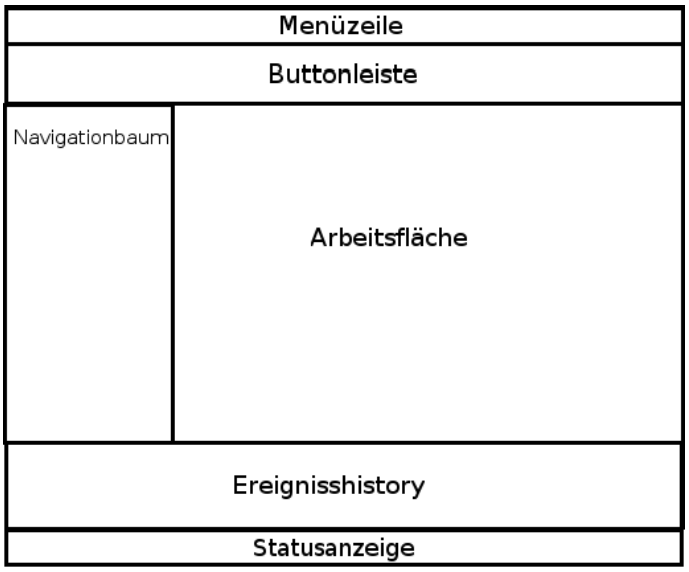

Abbildung 3.4: Schematische Darstellung der der Benutzungsoberfläche

- *Benutzbarkeit* Das System muss leicht zu bedienen und schnell zu erlernen sein. Neue Benutzer sollten sich schnell zurechtfinden können. Die Oberfläche sollte ergonomisch gestaltet werden und den Benutzer bei seiner Arbeit unterstützen.
- *Änderbarkeit* Die Komponenten des Systems sollen strukturiert sein, um eine Modifizierbarkeit und Analysierbarkeit zu ermöglichen. Außerdem soll es möglich sein, ohne Quellcodeänderungen Funktionen durch Plugins hinzuzufügen.
- *Übertragbarkeit* Das System soll möglichst in einer Plattformunabhängigen Sprache geschrieben werden. Weiterhin soll es möglichst mit der Standardkonfiguration der Systemumgebung lauffähig sein, um eine schnelle Installation zu ermöglichen und Umgebungsprobleme auszuschließen. Weiterhin müssen die einzelnen Komponenten unabhängig von anderen Teilen austauschbar sein.

3.2 Implementierung

In diesem Kapitel wird auf die Umsetzung der gestellten Anforderungen eingeangen. Dabei Zunächst wird beschreiben wie in der Entwicklung vorgegangen wurde, danach wird für jede einzelne Komponente beschrieben und auf einzelheiten der Implementierung eingegangen.

3.2.1 Entwicklungsprozess

Die genauen Anforderungen an das Gesamtsystem waren von Anfang an nicht eindeutig festgelegt. Für die Entwicklung der graphischen Benutzungsoberfläche galt es die bereits Vorhandene zu erweitern. Der genaue Umfang der geplanten Erweiterungen entwickelte sich erst im Verlaufe der Entwicklung. Im ersten Entwicklungsabschnitt wurde das alte System genau untersucht. Dieses war nötig, um ein möglichst großes Verständnis von den einzelnen Komponenten zu gewinnen. Darauf begann das Einarbeiten in die neu verwendeten Pakete und Technologien (BEEP,IDXP).Während der Entwicklungszeit musste beachtet werden, daß zu vielen Zeitpunkten eine begrenzt lauffähige Version des Programms vorliegen musste. Dieses gestaltete sich

sehr schwierig, da zu diesem Zeitpunkt ein Kompromiss zwischen den alten und neuen Konzepten aufrecht erhalten werden musste, damit die einzelnen Komponenten untereinander agieren konnten. In der späteren Entwicklungsphase, die nach der Umsetzung der Grundkonzepte begann, wurde eine hybrider Entwicklungsstiel zwischen dem Spiralmodell und dem evolutionären Modell betrieben. Dabei gab es einen Iterationszyklus von zwei Wochen. In jedem Zyklus wurde die Implementierung neuer beziehungsweise Verbesserung bestehender Konzepte unter Aufsicht einzelner Vertreter der jeweiligen Entwicklungsgruppen (GUI, Mittelschicht, Agentenschicht) besprochen. Die getätigten Absprachen galt es bis zum nächsten Treffen umzusetzen.

3.2.2 Konfigurations Datei

In diesem Abschnitt wird auf die Konfigurationsmöglichkeit der Oberfläche eingegangen. Dabei wird zunächst die verwendete Technologie erläutert. Darauf folgend wird erklärt, wie die Konfigurationsdatei aufgebaut ist, sowie auf die Bedeutung einzelner Parameter eingegangen.
Die Konfigurationsdatei dient dazu, externe Einstellungen global in der Benutzungsoberfläche verfügbar zu machen. Dabei wird die von JAVA mitgelieferte Property Klasse genutzt. Sie bietet eine Möglichkeit „name“, „Wert“Paare in XML sowie INI-Datei Format zu serialisieren. Weiterhin muss es möglich sein, die Konfiguration zu erweitern und global verfügbar zu machen, ohne Anpassungen am Quellcode vornehmen zu müssen (mit Ausnahme der Komponente, die den neuen Parameter nutzt). Die Konfigurationsdatei ist auf dem Webserver vorliegend und kann bei einer Verwendung der GUI aktuell ausgelesen werden. Näheres hierzu kann in **??** gefunden werden. Im folgenden wird eine Beispielkonfiguration demonstriert, die aus dem tatsächlichem Betrieb stammt.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>Gui Configuratation File</comment>
  <entry key="use_middlelayer">false</entry>
  <entry key="ip_middlelayer">134.106.22.151</entry>
  <entry key="port_middlelayer">10288</entry>
  <entry key="xslt_file">idmef-xsl/idmef.xsl</entry>
  <entry key="name">admin</entry>
  <entry key="password">XXX</entry>
</properties>
```

Die einzelnen Parameter werden wie folgt genutzt :

- *use_middlelayer :* Dieser Parameter legt fest, ob ein Versuch unternommen werden soll, sich zur Mittelschicht zu binden. Das heißt, ob ein lokaler Testlauf oder eine tatsächliche Benutzung stattfindet. Gültige Werte sind hier „true“(Nutzung der regulären Mittelschicht) und „false“autarker Testlauf.

- *ip_middlelayer :* hier wird die IP Adresse der Mittelschicht eingetragen. Sie wird für den Verbindungsaufbau benötigt.

- *port_middlelayer :* Hier kann eine Portadresse angegeben werden. Sie wird ebenfalls benötigt, um eine Verbindung zur Mittelschicht aufzubauen.
- *xsltfile :* Diese Einstellung ist der Pfad zu einer XSLT Datei. Diese Datei wird benötigt, um IDMEF die als XML vorliegen in HTML umzuwandeln. Diese Datei kann beliebig ausgetauscht werden.
- *name :* Hier kann zu Testzwecken ein Benutzername, der im Loginfeld erscheinen soll, eingetragen werden. Die Option wird eher zu Testzwecken verwendet, um ein schnelleres Einloggen zu ermöglichen.
- *password :* Ähnlich der des Parameters „name“kann hier analog ein zum Login gehöriges Passwort eingetragen werden. Aus sicherheitstechnischen Gründen wird davon abgeraten die Parameter „name“und „password“im regulären Betrieb zu nutzen.

3.2.3 Paketstruktur

In diesem Abschnitt wird genau auf die Einteilung der Programmabschnitte eingegangen. Wie in Abbildung 3.5 zu erkennen ist, wird der Aufbau der graphischen Benutzungsoberfläche in fünf verschiedene Bereiche eingeteilt. Dieses sind die Bereiche : XML-Verarbeitung, Kommunikation, Datenhaltung, Visualisierung und Hauptklasse. Im folgenden wird näher auf die einzelnen Bereiche und ihre enthaltenen Pakete eingegangen. Exakte Informationen zu jeder Klasse befinden sich **??** Anhang.

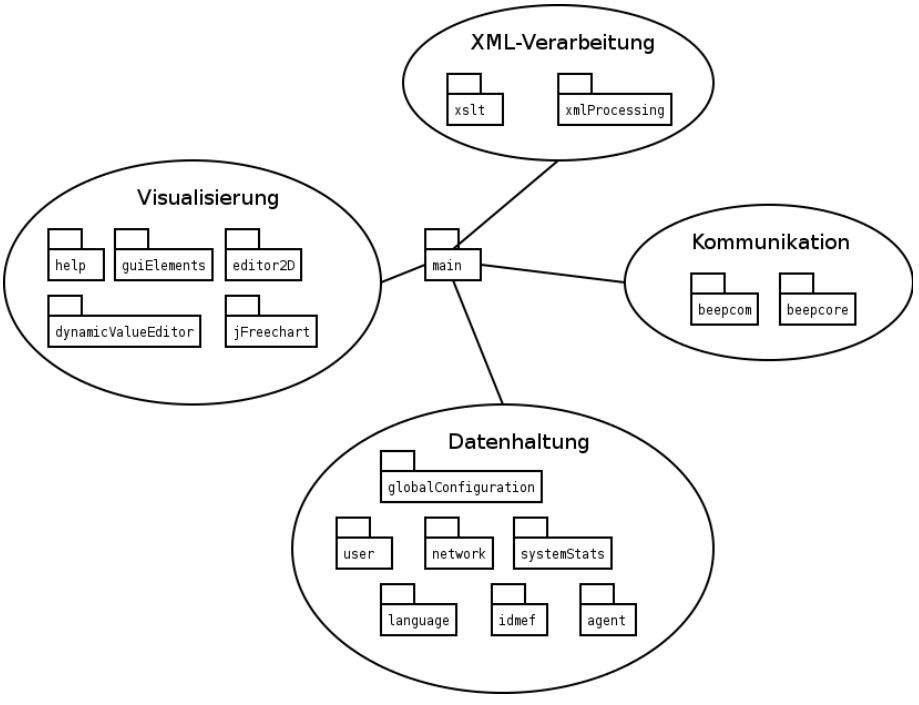

Abbildung 3.5: Schematische Darstellung der Architektur der GUI

3.2.3.1 Datenhaltung

In diesem Bereich befinden sich alle Pakete die Klassen beinhalten, welche eine interne Repräsentation eines Objektes, das im unIDS System Anwendung findet, bearbeitet. Jedes Paket modelliert einen bestimmten Weltausschnitt und beinhaltet

dazu passende Klassen. Diese Klassen bilden eine Ansammlung von Methoden für den Zugriff auf Daten des jeweiligen Bereiches. Dabei handelt es sich um Methoden zur Serialisierung beziehungsweise Deserialisierung von einzelnen XML Abschnitten, sowie Getter und Setter für den Zugriff auf einzelnen Attribute. Im Folgenden wird kurz auf jedes enthaltene Paket eingegangen und die genauen Aufgaben der enthaltenen Klassen erläutert.

- `globalConfiguration` : Das Paket globalConfiguration ist für die Userdatenverwaltung und Konfiguration der Oberfläche zuständig. In ihr befinden sich zwei Klassen. Die erste Klasse `ConfigItem` beinhaltet Methoden für die Datenhaltung, in Form von Benutzer- und Komponenteninformationen.Die zweite Klasse `GuiConfiguration` ist für das Auslesen und bereitstellen der Konfigurationsdatei notwendig. Genaues Konzept unter REFERNZ Konfiguration.

- `user` : In diesem Paket befindet sich lediglich das User Objekt. Es bietet Methoden zur Serialisierung und Deserialisierung von XML Elementen mit Benutzerinformationen. Dieses Sind unter anderem : Login, Name, Passwort, Beschreibung des Benutzer, sichtbare Geräte und Zugriffsrechte

- `network` : Das Paket `Network` beinhaltet Klassen die für den Aufbau von Netzwerkstrukturen verwendet werden können. Die Klasse `UnidsElementStrukture` dient als Container für Objekte aus der Klasse `UnidsNetObjekt`. Hier werden die Objekte in einer hirachischen Struktur abgebildet. Um direkte Verbindungen zwischen verschiedenen UnidsNetObjekt Objekten zu modellieren, wird die Klasse `Connection` verwendet. Sie beinhaltet alle Informationen zu einer einzelnen Netzwerkverbindung.

- `idmef` : Hierbei handelt es sich um ein Paket zur Serialisierung und Deserialisierung von IDMEF Nachrichten. Dabei handelt es sich um ein Sourceforge Fremdpaket. Es wurde entwickelt um den Inhalt von IDMEF Nachrichten in Form von einzelnen Objekten verfügbar zu machen. Dieses Paket wird genutzt, um eine vollständige Zugriffsmöglichkeit auf alle Aspekte einer IDMEF Nachricht zu erhalten. Dieses ermöglicht eine detaillierte Auswertung von in der IDMEF Nachricht enthaltenen Informationen durch künftige Erweiterungen des Systems durch Plugins.

- `languages` : Dieses ist das Sprachpaket. Es ist das Herzstück der Internationalisierung. Hier befinden sich Dateien, die alle Konstanten beinhalten, die benötigt werden um auf verschiedene Übersetzungen eines Schriftbereiches zuzugreifen. Weiterhin sind hier alle Sprachdateien enthalten, die jeweils eine Sprachübersetzung des Benutzerinterfaces beinhalten.

- `agent` : Das Paket Agent beinhaltet alle Klassen die einen unIDS-Agenten modellieren. Es sind wiederum Methoden zur Serialisierung und Deserialisierung von XML Elementen mit Informationen über Agenten vorhanden. Dieses sind Informationen wie Agenten ID und benutzte Plugins. Die zu einem Agenten zugeordneten Plugins werden in Instanzen der Klasse `AgentPlugin` gespeichert. Hier sind wiederum Methoden zur Transformation von und in XML vorhanden, sowie Speichermöglichkeiten für den Zustand eines Plugins und zugeordneter Plugin-Parameter. Näheres hierzu im Kapitel **??** sowie **??**.

- `systemStats` : Im Paket `systemStats` werden alle Statusinformationen gesammelt und verwaltet. Dafür steht eine Vielzahl von Klassen mit unterschiedlichen Teilaufgaben bereit. Die Klassen `DeviceStatus` ,`SystemData`, `SystemDataSet` werden benutzt um die Strukturierung der Systeminformationen nachzubilden und durch die Klasse `GlobalsSystemDataList` für Komponenten innerhalb GUI verfügbar zu machen. Damit eine Komponente eingehendeSysteminformationen nutzen kann muss das Interface *Statsrequester* implementiert werden.

3.2.3.2 Kommunikation

- `beepcore:` Beim Beepcore Paket handelt es sich um eine Referenzimplementierung der RFC3080 und RFC3081. Diese Implementierung ist in Java implementiert worden. Sie bietet Möglichkeiten Java-Applicationen unter Verwendung des BEEP Framework zu implementieren. Näheres hierzu kann im Kapitel **??** gefunden werden oder unter www.beepcore.org.

- `beepcom` : Dieses Paket beinhaltet alle Klassen, die benötigt werden um eine Kommunikationsverbindung unter Verwendung des BEEP Frameworks innerhalb des unIDS - Netzwerkes aufzubauen. Dabei übernimmt die Klasse `BeepCommunication` den eigentlichen Kommunikationsaufbau mit der Mittelschicht. Dabei werden zwei unterschiedliche Kanäle aufgebaut und zwei Profile für diese Kanäle implementiert. Beim ersten Kanal handelt es sich um einen Datenkanal der zur Mittelschicht aufgebaut wird. Er ist in der Klasse `GuiDataExchangeProfil` implementiert und wird ausschließlich für den Datenaustausch zwischen der GUI und der Mittelschicht verwendet. Der zweite Kanal dient ausschließlich zum Austausch von IDMEF Nachrichten zwischen den Agenten und den an der Mittelschicht angemeldeten Benutzerinterfaces. Dieses Profil ist in der Klasse `IDMEFProfil` implementiert.
  Weiterhin beinhaltet dieses Paket zwei Interfaces. Diese Interfaces (*IDMEFReceiver*, *MessageReceiver*) ermöglichen der implementierenden Klasse IDMEF Nachrichten, sowie Statusnachrichten aus den vorgestellten BEEP-Profilen zu empfangen. Die Klasse `ServerSystem` dient als Vermittlungsstelle zwischen der Visualisierungsschicht und der Kommunikationsschicht. Hier können Methoden angesprochen werden um direkt eine Zustandsänderung in der Mittelschicht auszulösen, wie zum Beispiel durch Änderung von Benutzerdaten.

3.2.3.3 XML-Verarbeitung

Im Bereich XML-Verarbeitung sind alle Pakete zusammengefasst, die sich um die Serialisierung und Deserialisierung von von XML Kommandos befassen, die an die Mittelschicht weitergeleitet werden. Weiterhin existiert ein Paket zur Umwandlung von XML durch XSLT.

- `XMLProcessing` : Dieses Paket beinhaltet die Klasse `XMLCreator`. Sie beinhaltet Methoden und Funktionen zur Zusammenstellung von Befehlen im XML Format. Die XML Dokumente können verwendet werden um bestimmte Aktionen in der Mittelschicht auszulösen. Beispiele hierfür sind das Einfügen, Aktualisieren, Abfragen und Löschen von bestimmten Benutzern, Geräten oder Agenten. Weiterhin werden hier Spezialanfragen wie zum Beispiel die Registrierung für Systeminformationen zusammengestellt.

- `XSL-t` : Hierbei handelt es sich um ein extern eingebundenes Paket. Es ist das XALAN Paket aus dem Apache XML Projekt. Es ist ein Paket zur Umwandlung von XML durch XSLT. Es wird innerhalb der GUI verwendet um IDMEF Nachrichten in HTML umzuwandeln, um eine benutzerfreundliche Visualisierung zu ermöglichen. Die Klasse `IDMEFSerializer` übernimmt hierbei die Zusammenstellung der XSLT Datei und leitet den eigentlichen Umwandlungsvorgang ein.

3.2.3.4 Visualisierung

In diesem Bereich befinden sich alle Pakete und Klassen, die benötigt werden um Informationen zu visualisieren. Es werden weiterhin Klassen bereitgestellt die Interaktionsflächen bieten, um Einfluss auf die Funktionsweise der Benutzungsoberfläche, der Mittelschicht, sowie Agentenschicht zu nehmen.

- `help` : Im Paket `help` sind alle Klassen zusammengefasst, die für die Visualisierung der Programmhilfe nötig sind. Hierbei handelt es sich grundsätzlich um die von JAVA SUN bereitgestellte JHELP. Näheres hierzu in Kapitel **??**.

- `dynamicValueEditor` : In diesem Paket sind alle Klassen zusammengefasst, die dafür zuständig sind alle Im unIDS System vorkommenden Objekte als Formular editierbar zu machen. Dabei übernimmt die Klasse `EditorCreator` die Umwandlung von Objekten wie Benutzer und Geräte in ein entsprechendes Formular. Die Klassen `EditorConfigItem` und `EditorConfig` beinhalten dabei eine Kopie der Objektinhalte. Um die Veränderungen wieder aus dem Formular zu gewinnen, gibt es zu jedem Objekt eine Klasse `SaveAction(Objekt)`

- `jFreechart` : Das Paket `jFreechart` biete viele Visualisirungsmöglichkeiten für statistische Daten. Es handelt sich hierbei um ein fremdeingebundenes Paket der Seite www.jfree.org. Ausführlich wird dieses in REFERNZ Diplomarbeit JENS erarbeitet.

- `guiElements` : Es ist das mit Abstand umfangreichste Paket im Visualisierungsbereich. Es beinhaltet alle strukturellen Elemente der Oberfläche. Es gibt unter anderem eine Klasse (`Browser`) um einen HTML Browser zu simulieren. Dieser Browser wird verwendet um IDMEF Nachrichten anzuzeigen. In den IDMEF Nachrichten befinden sich meist Hyperlinks zu Seiten, die den erkannten Angriff, welcher zum senden der IDMEF Nachricht geführt hat, beschreiben. Diese Seiten können leicht im gleichen Fenster aufgerufen werden, ohne das ein Ausführen eines externen Browsers notwendig wird.
  Zur Visualisierung von Systeminformationen werden die Klassen `Infoframe` und `Infopanel` verwendet. Sie bieten ein Grundgerüst zum Auflisten von System-Plugins. Alle eingehenden Systeme und Alarmnachrichten werden durch die Klasse `History` bearbeitet. Sie generiert aus den eingehenden Nachrichten eine Auflistung mit kurzer Zusammenfassung des Inhalts. Daneben sind noch viele andere Elemente der Oberfläche in Klassen strukturiert wie die Buttonleiste, Benutzer- und Oberflächeneinstellungen.

- `editor2D` : Hierbei handelt es sich um ein Paket mit einer Zusammenstellung von Klassen aus der Vorgängersoftware. Es wurde der 2D Raum-Editor mit

allen Abhängigkeiten zu alten Klassen beibehalten und versucht dieses Paket um aktive Netzwerkgestaltungsmöglichkeiten zu erweitern. Leider konnten hier nicht alle Ziele vollständig abgearbeitet werden. Näheres hierzu in Kapitel **??**

3.2.4 neues XML-Konzept

Zu diesem Zweck wurde eine eigene Syntax entwickelt. Sie besteht aus XML Befehlen und in XML serialisierten Objekten. Es gibt fünf verschiedene Anweisungstypen, die von der Benutzungsoberfläche unterstützt werden. Diese fünf Anweisungstypen bestehen jeweils aus einem XML Tag und umschließen dabei den eigentlichen Inhalt der Nachricht. Weiterhin gibt es vier verschiedene Inhaltstypen.Im folgenden werden alle Typen kurz erläutert:

- `<add>` Der Add Anweisungstyp gibt an, das alles innerhalb des Tags in den Datenbestand eingepflegt werden soll. Dabei kann es sich um einen oder mehrere Inhaltstypen handeln. Wenn dieser Anweisungstyp benutzt wird, ist es nicht erforderlich das die einzufügenden Objekte eine ID besitzen. Sie wird erst während des Einfügeprozesses in der Mittelschicht gebildet.

- `<update>` Der Anweisungstyp Update verhält sich ähnlich wie der Anweisungtyp Add. Alle Inhaltstypen die sich zwischen dem Update Tag befinden werden in der Mittelschicht aktualisiert. Dabei ist es zwingend erforderlich, das alle Elemente mit einer eindeutigen ID ausgestattet sind. Falls es sich bei den einzufügenden Inhalten um Inhalte handelt die Parameter aufweisen, werden auch diese im Sinne des Nachrichteninhaltes verändert.

- `<delete>` Durch das Delete Tag wird eine Löschung aller Elemente eingeleitet die in diesem Tag eingebettet sind. Dabei ist nur wichtig das diese Elemente mit einer eindeutigen ID ausgestattet sind. Eventuell vorhandene Attribute müssen nicht explizit mit gelöscht werden, sondern werden durch diesen Befehl von der Mittelschicht automatisch mit dem zu entfernenden Objekt entfernt.

- `<register>` Das Register Tag wird ausschließlich für Anfragen verwendet die sich auf Typen des Inhaltes „Element“ beziehen. Hier wird der Mittelschicht signalisiert, das die Oberfläche, welche diese Nachricht gesendet hat, Statusinformationen des Elementes anfordert.

- `<unregister>` Das analog zum Register Befehl kann mit dem unregister Befehl eine Statusanfrage auf ein bestimmtes Element aus dem unIDS-system aufgehoben werden.

Es gibt im unIDS-System genau fünf XML-Inhaltstypen die zwischen der Benutzungsoberfläche und der Mittelschciht verwendet werden. Im folgenen werden diese fünf Inhaltstypen vorgestellt und die jeweilige XML-Serialisierung gezeigt.

- `Benutzer` Der Inhaltstype „Benutzer“ beinhaltet alle zentrtalen Informationen für einen Benutzer. Die Serialaisierung sieht Beispielsweise folgendermassen aus:

```
<user id="user/8" isadmin="true">
   <name>Stefan Meyer</name>
   <login>Stefan</login>
   <description/>
   <viewable-objects/>
 </user>
```

Innerhalb von `<viewable-objects/>` können ein oder mehrere Elemente des Inhaltstypes `elmement` in Kurzschreibweise auftauchen.

Sobald das „Löschen“ Tag verwendet wird wird eine Kurzschreibweise verwendet. Hier reicht lediglich die Angabe der eindeutigen BenutzerID.

```
<user id="user/8">
```

- `element` Das der Elementinhaltstype ist der Zentrale Inhaltstype für alle unIDS Inhalte wie Agenten, Proxys, Geräte und Subnetzte. Eine direkte Beziehung welches Element teilmenge von einem Anderen Element ist, wird direkt über den hirarchischen XML- Aufbau ermöglicht. jedem Element kann eine beliebige Anzahl von Attributen angehoeren. Das Beispiel zeigt ein Gerät. Es hat die ID „device/124“ und verschiedene Attribute.

```
<element id="device/124" type="device">
      <properties>
          <entry key="comment"/>
          <entry key="status">1</entry>
          <entry key="name">Unnamed (device/124)</entry>
          <entry key="virtual">true</entry>
          <entry key="icon">0</entry>
        </properties>
      </element>
```

Auch hier gilt, bei der Benutzung und Anfrage auf Inhalt dieses Elementes wird die Kurzschreibweise verwendet.

```
<element id="device/124" type="device"/>
```

- `GuiDATA??` Der Inhaltstyp Guidaten wird genauer im Kapitel
- `connection` Der Connection Inhaltstyp kommt in vielen verschiedenen Nachrichten vor. Zum einen kann er Teil einer Elementnachticht sein. Er wird hier benutzt um verschiedene Elemente miteinander zu verbinden. Weiterhin kann er auch einzeln vorkommen. Dieses ist beim Erstellen, Löschen und Abfragen von connections der Fall.

```
<connection id="connection/1" source="device/1" target="device/3" virtual="
```

Im unIDSsytem gibt es eine Reihe von Speziealnachtichten, welche nicht in das vorgestellte Nachrichtenkonzept passen. Sie werden benutzt um spezielle Zustände oder Ereignisse in der Mittelschciht auszulösen oder dienen als Antwortnachrich auf das Ausführen eines bestimmten Ereignisses.

- *login* Die Loginnachticht ist eine dieser Spezialnachrichten. Sie muss gesendet werden damit sich ein bestimtmer Benutzer an der Mittelschicht anmelden kann. Im `user` Tag steht der Login des Benutzers und im Tag `passwort` das Password des Benutzers.

```
<login method="password">
  <user>Stefan</user>
  <password>test</password>
</login>
```

- *beepreply* Dieser Nachrichtentyp dient als Container für Informationen über die Verarbeitung der vorrangegangenen Nachricht. Sie ist eine Art Antwortnachticht. Wenn eine bestimmte Ressource angefordert wird und bei der Verarbeitung in der Mittelschicht ein Fehler autreten sollte, wird diese Nachricht mit einer genauen Beschreibung des Fehlers, statt dem Angeforterten Inhalt gesendet. Diese Nachricht wird auf jeden Fall gesendet, fals Aktion angefordert wurde. Das Tag `status` beinhaltet eine fest vereinbarte Konstante. Jeder Fehlertyp hat eine eigene Konstante. Im Tag `description` befindet sich eine nähere Beschreibung zum augetretenen Fehler. Diese Nachrichtentyp kann sowohl von der Mittelschicht als auch von der GUI genertiert werden.

```
<reply>
  <status>OK</status>
  <description>Login successful.</description>
</reply>
```

3.2.5 Statusinformationen

In der alten GUI wurde ein Polling von Systeminformationen realisiert. Die GUI fragt in regelmäßigen Abständen den Status der Datenbankschicht ab, um an aktuelle Systeminformationen zu gelangen.
Dieses System soll ersetzt werden durch ein Anmeldesystem. Einzelne Komponenten sollen sich innerhalb der GUI anmelden können für den Empfang von bestimmten Informationen eines Gerätes. Solange Anfragen für ein bestimmtes Gerät vorliegen, soll die Mittelschicht bei jeder Systeminformationsänderung ein Status zu den betreffenden Komponenten schicken. Damit wird gewährleistet, das Informationen sobald sie vorliegen die betreffenden Stellen erreichen.

In Abbildung 3.6 wird der Ablauf eines Systeminformationsempfangs auf der GUI-Seite anhand eines Sequenzdiagrammes illustriert. Im ersten Schritt(1.benutzt) öffnet der Benutzer innerhalb der Oberfläche ein Komponentennetz(Kom.1) das Systeminformationen eines bestimmten Gerätes anzeigen kann. Diese Komponente meldet sich im Schritt zwei bei der Klasse `Statinfo` an. Diese Klasse prüft nun ob bereits Systeminformationen für das Gerät aus älteren Abfragen vorliegen. Wenn dieses der

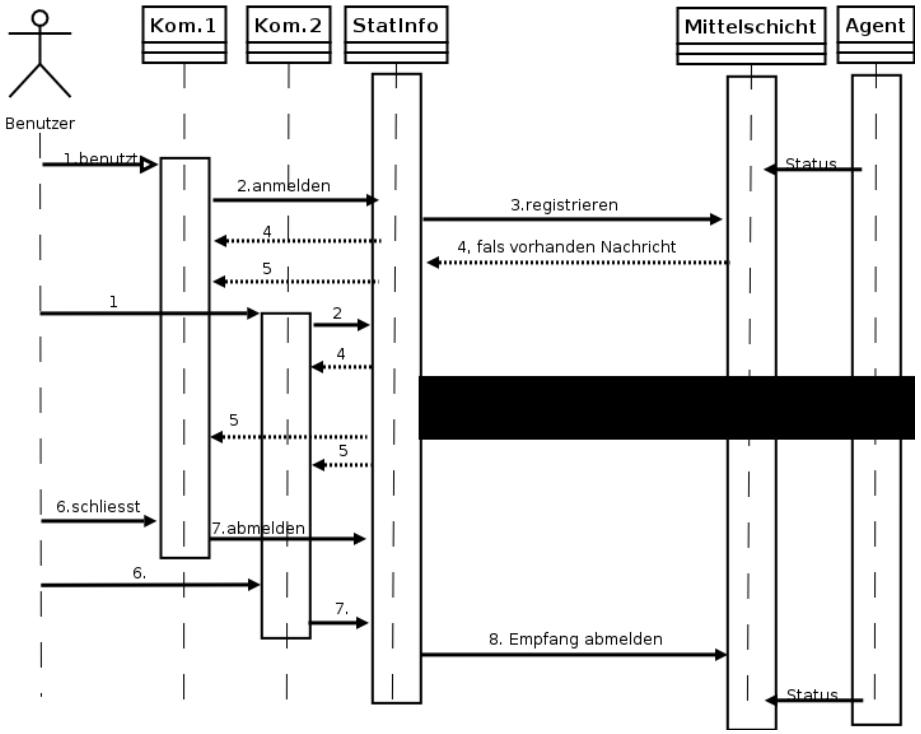

Abbildung 3.6: Sequenzdiagramm Systeminformationen

Fall ist, wird sofort ein alter Stand an die Komponente zurückgesendet(4.). Wenn für keine andere Systemkomponente eine Anfrage vorliegt, wird eine Registrierungsnachricht an die Mittelschicht für das zu betrachtende Element gesendet. Auch hier wirkt ein Zwischenspeicher der Mittelschicht. Wenn auch hier bereits Informationen zu diesem Gerät vorliegen, die im Allgemeinen aktueller sind als die Informationen die bei der Oberfläche eventuell vorliegen wird ebenfalls eine Aktualisierungsnachricht sofort an die GUI weitergeleitet. Mehr hierzu im Kapitel **??**. Innerhalb der Benutzungsoberfläche können mehrere Komponenten Informationen zum gleichen Gerät verlangen(Kom.2). Diese erneute Abfrage wird von der Mittelschicht gekapselt und führt zu einer Erweiterung der vorliegenden Anmeldungen innerhalb der Klasse `StatInfo`. Wenn nun ein Agent frische Informationen sammelt und zur Mittelschicht weiterleitet, wird automatisch eine Aktualisierungsnachricht von der Mittelschicht an die Benutzungsoberflächen geschickt. Hier findet nun eine Aktivierung aller angemeldeten Komponenten für das Gerät in der Aktualisierungsnachricht statt(5.).Sobald die Komponenten geschlossen werden, wird eine GUI- interne Abmeldung dieser Komponente in der Klasse `StatInfo`ausgelöst und die Komponente wird aus der Liste der wartenden Komponenten gestrichen. Sobald für ein Gerät keine Anforderung mehr vorliegt. Wird von der Klasse `StatInfo` eine Entregistrierung an die Mittelschicht geleitet.

3.2.6 Speicherung von Benutzereinstellungen

Die alte Benutzungsoberfläche hatte die Möglichkeit personalisierte Informationen zu speichern. Hierbei handelte es sich im wesentlichen um die Bildschirmaufteilung des Interfaces. Weiterhin konnte für jeden Benutzer festgelegt werden welches Gerät er einsehen bzw. nutzen kann. Hierbei war für jeden einzelnen Benutzer nur ein Eintrag für personalisierte Informationen möglich.

Im Rahmen der Weiterentwicklung sollen diese Möglichkeiten bereichert werden. Die Beschränkung auf einen Nutzer soll gelockert werden, sowie die Beschränkung der Speicherung von GUI internen Daten auf Benutzeransicht gelockert werden. Hierzu ist ein System geplant, das eine Speicherung von personalisierten Inhalten für Objekt / Komponentenpaare möglich ist. Das heißt das jede Komponente zu einem Gerät ihre eigenen Einstellungen speichern kann. Weiterhin muss eine Komponente die Möglichkeit haben beliebig viele Daten zu einer Objekt- / Komponentenrelation zu Speichern. Um eine möglichst hohe Lesbarkeit und Wartbarkeit der gespeicherten Informationen zu gewährleisten. Solle das System so genutzt werden, das es dem Entwickler und den Administratoren schnell möglich ist diese Daten zu verwalten beziehungsweise unter Ausnutzung dieses Konzeptes, die Software zu erweitern.
Durch die Kombination von Objekt und Komponente ist dieses gleichzeitig unter minimaler Kenntnis der Mittelschicht über den Inhalt der einzelnen Speicherung zu entscheiden welche Daten überflüssig sind. Beim Objekt handelt es sich um ein Unidsinternes Objekt was eindeutig an der ID zu erkennen ist. Wenn nun dieses Objekt (Gerät, Nutzer) gelöscht wurde, können nun auch automatisch alle Inhalte gelöscht werden, die mit dieser Objekt ID verbunden sind. Näheres hierzu im Kapitel REFERNZ MITTELSCHICHT GUIDATEN.

```
<update>
  <gui-text id="3;GuiElements.Preferences">
    <text><![CDATA[<?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
      <properties>
        <comment>ConfigItem</comment>
          <entry key="Preferences - new Window User">true</entry>
          <entry key="UnidsGuiApplication - Screen Height">768</entry>
          <entry key="UnidsGuiApplication Dialog - close">true</entry>
          <entry key="UnidsGuiApplication - Height">748</entry>
          <entry key="UnidsGuiApplication - Look and
          <entry key=" Preferences - new Window Subnet">true</entry>
          <entry key="UnidsGuiApplication - Width">1032</entry>
          <entry key="Preferences - new Window Device">true</entry>
          <entry key="Preferences - ToolTips">true</entry>
          <entry key="Preferences - Information">true</entry>
          <entry key="Preferences- new Window Help">true</entry>
          <entry key="UnidsGuiApplication - Language">0</entry>
          <entry key="UnidsGuiApplication - Screen Width">1024</entry>
      </properties>]]>
    </text>
  </gui-text>
</update>
```

3.2.7 Erstellte Plugins

In diesem Bereich wird auf erstellten Plugins eingegangen. Dabei wird jeweils die Funktionsweise erläutert und die Einbettung ins Gesamtsystem beschrieben.

3.2.7.1 Mittelschichtadministration

Neben den genannten indirekten Möglichkeiten wie Geräte-, Benutzer- und Agentenkonfiguration, gibt es eine Möglichkeit direkt auf den Zustand der Mittelschicht einzuwirken. Hierzu wurde ein Plugin verwirklicht, das beim initialen Start eine eine Anfrage an die Mittelschicht sendet. Bei dieser Anfrage handelt es sich um eine Abfrage der bereitgestellten externen Funktionen. Diese externen Funktionen werden an die Benutzungsoberfläche weitergeleitet und innerhalb des Plugins ausgewertet.Daraus ergibt sich eine Liste verfügbarer Kommandos. Diese Liste wird in Form von Buttons dem Benutzer präsentiert. Hier kann er nun durch Klicken auf den Button das Senden einer bestimmten Nachricht an die Mittelschicht auslösen. Alle möglichen Nachrichten, die gesendet werden können, werden innerhalb der innitialen Abfrage von der Mittelschicht geliefert. Für die fertige Version gab es bereits einige administrative Nachrichten Befehle sowie Nachrichten zum Einleiten von Testläufen. Unter anderem kann hier ein Befehl zum zurücksetzen der Mittelschicht in den Ausgangszustand abgesendet werden. Dabei wird alles gelöscht und durch ein kleines Dummy-Netzwerk sowie einer Ansammlung von Standardbenutzern ersetzt. Weitere Nachrichten bewirken das Senden von TEST IDMEF Nachrichten sowie das Generieren von Test Systeminformationen. Bei allen Funktionen, die für die Generierung von Testdaten verantwortlich sind, kann durch das erneute Senden dieser Nachricht die Arbeit eingestellt werden. Bei jedem Aufruf eines Nachricht bekommt der Benutzer ein Feedback über das ausgelöste Ergebnis in Form eines Informationsdialoges, wobei der Textinhalt von der Mittelschicht generiert wird. Dieser Gesamtaufbau hat den Vorteil, das eine Erweiterung des Administrationsumfangs der Mittelschicht automatisch unter der Benutzungsoberfläche verfügbar ist.

3.2.7.2 Alternativer Zugriff

Um dem Benutzer weite Möglichkeiten zu geben seine Arbeit auch mit Hilfe des Menüs zu erledigen und nicht nur unter alleiniger Ausnutzung des Strukturbaumes, wurden verschiedene Hilfs-Plugins entwickelt. Hier kann der Benutzer unter anderem die Useradministration über eine Benutzerliste durchführen, sowie eine Geräte und Agentenadministration über entsprechende Ausswahllisten.

3.2.7.3 Alternative Übersicht

Gelegentlich kann es sinnvoll erscheinen wenn Eigenschaften von mehreren Geräten in Form einer Übersichtsliste präsentiert werden. Zu diesem Zweck wurde das Plugin entwickelt. Es stellt alle Geräteinformationen zusammen und visualisiert sie, inklusive ihrer jeweiligen Attribute in Form einer Übersichtstabelle. Dieses Plugin wurde sowohl für Geräte also auch Subnetze verwirklicht.

3.2.7.4 Systeminformations Plugins

Bei den Systeminformations-Plugins handelt es sich um eine Ansammlung von verschiedenen Plugins zur Darstellung von Systeminformationen. Es sind die Plugins : `Debug_Plugin`,`CPU_Info_Plugin` `Device_Info_Plugin`,`Mount_Info_Plugin` und `Memory_Info_Plugin` Diese Plugins werden nun im folgenden Einzeln erläutert.

- `Debug_Info_Plugin` Die Hauptaufgabe dieses Plugins besteht darin, eingehende Informationen von Agenten auf dem Bildschirm auszugeben. Dabei wird die

Ankunft der letzten acht Nachrichten durch einen Zeitstempel dokumentiert und der aktuelle Inhalt der globalen Systeminformationsliste für das ausgewählt Gerät präsentiert.

- `CPU_Info_Plugin` : Beim CPU-Anzeige Plugin handelt es sich um ein Systeminfo-Plugin. Es dient in erster Linie dazu die CPU Auslastung in einem graphischen Verlauf zu demonstrieren. Abbildung 3.7 zeigt das aktive Plugin im InfoFrame. Hierbei ist zu erwähnen das dabei alle Möglichkeiten von JFreeChart freigeschaltet sind. Dieses ermöglicht ein problemloses Hereinzoomen und Speichern von Verlaufsbildern. Näheres hierzu ist in der Diplomarbeit **??** zu finden.

- `Memory_Info_Plugin` : Das `Memory_Info_Plugin` ist ebenfalls ein Systeminfo Plugin. Es visualisiert Informationen über die Speichernutzung auf dem Agentensystem. Hierzu wird ein Kuchendiagramm benutzt, das aus dem Repartoir von JFeeChart stammt.

- `Device_Info_Plugin`Um einem normalen Benutzer, der nicht über Administratorrechte verfügt, über die Eigenschaften eines Gerätes zu informieren, wird das `Device_Info_Plugin` benutzt. Es stellt im wesentlichen eine Kopie des Geräte Editors dar. Hierbei sind lediglich die Eingabefelder gesperrt um den Zugriff auf das Gerät einzuschränken.

- `Mount_Info_Plugin` Das `Mount_Info_Plugin` dient der Visualisierung von Festplatteninformationen auf dem ausgewählten Gerät. Es verschafft in Form eines Kuchendiagramms einen Überblick über die Anzahl der Partitionen sowie eine Übersicht über den verbrauchten Speicherplatz jeder einzelnen Partition.

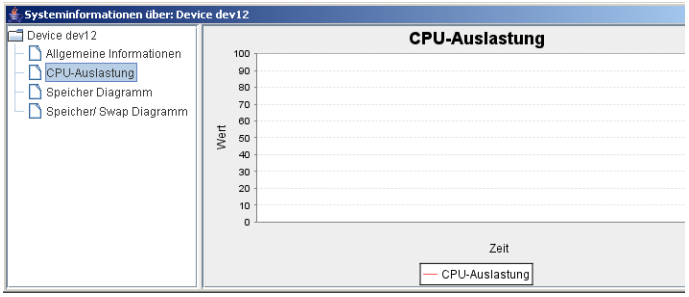

Abbildung 3.7: Implementiertes Prozessor Plugin

3.2.8 Einflussnahme auf andere Schichten

Um dieses Prinzip umzusetzen wird die Configurationsnachricht des Agenten verwendet. Sie kann vielseitig verwendet werden. Zum einen, kann über eine Configurationsnachricht der Status eines Agenten abgefragt werden. Zum anderen kann durch Veränderung der Parameter und Weiterleitung an den Agenten dieser Status verändert werden. Um eine Konfigurationsnachricht bearbeitbar zu machen, wird der in der Diplomarbeit **??** vorgestellte dynamische Editor verwendet. Abbildung 3.8 verdeutlicht den Vorgang einer Konfiguration des Agenten auf Seiten der GUI.

Zunächst ruft der Benutzer über das graphische Interface einen Agenten auf(1.). Diese Anfrage wird über eine Zwischenstelle in der GUI (2.) an die Mittelschicht (3.) weitergeleitet. Von der Mittelschicht wird eine Antwort gesendet die im Allgemeinen

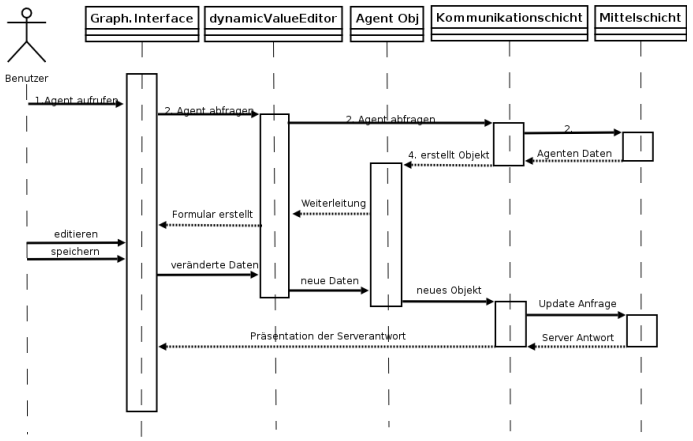

Abbildung 3.8: Sequenzdiagramm Agentenkonfiguration

die Agenteninformationen in einer XML Nachricht enthält(Agenten Daten). Diese Informationen werden durch Deserialisierungsmethoden innerhalb der GUI in ein Agenten-Objekt umgewandelt(4. Erstellt Objekt). Dieses Objekt wird im nächsten Schritt durch eine Hilfsklasse im dynamischen Editor in ein Formular umgewandelt, das dem Benutzer präsentiert wird. Hier kann er alle Einstellungen am Agenten vornehmen und per Knopfdruck eine Speicherung einleiten. Nach einleiten der Speicherung werden die im Formular enthaltenen Daten wiederum mit einer Hilfsklasse zurück in das Objekt gespeichert. Dieses neue Objekt wird nun über die Kommunikationsschicht an die Mittelschicht zurückgesendet. Dabei wird es wie im Kapitel **??** mit dem UPDATE TAG markiert. Je nachdem wie die Speicherung verläuft erhält der Benutzer eine graphische Rückmeldung von der Kommunikationsschicht.

3.2.9 Internationalisierung

All diese Möglichkeiten bietet ein bereits von JAVA mitgeliefertes Konzept zur Internationalisierung. Hier wird mit Hilfe von Property-Dateien, die nach einem bestimmten Schema aufgebaut und benannt sein müssen. Im Allgemeinen besteht der Dateiname aus einem suffix(in diesem Fall „unIDS-language“ einem Sprachprefix, bestehen aus Länder und Sprachkennung zum bespiel „_de_DE“ für Deutschland, deutsch) und der Dateiendung „.properties“. Im gleichen Schema können Dateien für jedes beliebige Land und Sprache erstellt werden.
Die Datei ist eine Property-Datei mit Schlüssel und dazugehörigem Wert. Hier können um eine Neue Sprache zu integrieren einfach die Werte der jeweiligen Schlüssel in die neue Sprache übersetzt werden. Um dieses zu verdeutlichen wird folgender Beispiel Ausschnitt aufgeführt.

Datei : unIDS-language_de_DE.properties

```
unIDS_Tree_New_User = Benutzer
insert = einf\u00FCgen
User_newPassword = neues Passwort
```

der äquivalente Ausschnitt der englischen (unIDS-language_en_US.properties) sieht folgendermaßen aus.

```
unIDS_Tree_New_User=User
insert=insert
User_newPassword=new Password
```

3.3 Zusammenfassung und Fazit

Im Bereich der Benutzungsoberfläche konnte viele Verbesserungen durchgeführt werden. Das gesamte System weist nun eine klarere Struktur auf. Einzelne Bereiche sind klar voneinander getrennt und weisen keine weitreichenden Verstrickungen mehr auf wie im Vorgabe Projekt. Diese Verstrickungen und klaren Abgrenzungen beziehen sich unter anderem auf die Gliederung der Oberfläche. Hier wurden viele Elemente aus einem ursprünglichen Zusammenwurf vieler Funktionen herauskristalisiert und gekapselt. Einzelne Elemente sind nun getrennt und in ein Paketsystem eingeordnet worden. Weiterhin wurde die XML Kommunikation von der Implementierung anderer Bereiche getrennt. Es wird in den meisten Fällen keine Überschreibung der Daten mehr vollzogen sondern eine direkte Auswertung getätigt. Dieses in Verbindung mit der entwickelten Scriptähnlichen XML Syntax, hat zu einer weitreichenden Verbesserung der möglichen Einsatzgebiete und Funktionen innerhalb des gesamten Systems geführt. Dieses ist unmittelbar mit der Neuen Struktur der Netzwerke verbunden. Hier sind Einteilungen in zusammengehörige Gruppen(Subnetze) nun möglich und damit konnte eine intuitive Darstellung des gesamten Netzwerkes realisiert werden. Ebenfalls wurde der Quellcode strukturierter, übersichtlicher und Objektorientierter gestaltet, so das für künftige Projekte eine bessere Chance entsteht das Produkt noch einmal aufzugreifen und Änderungen leichter zu bewältigen sind. Die Internationalisierung wurde eindeutig verbessert und kann sich nun mit der Implementierung in anderen Projekten die dieses als Entwicklungsziel hatten messen. Ein weiterer positiver Punkt ist die Verbesserung der Kommunikation auf Netzwerkebene. Hier wurde das selbst implementierte System durch einen Standard(BEEP) ersetzt, welches extra für Systeme dieser Art konzipiert wurde. Der Kommunikationsweg an sich führt nicht mehr über eine zwingend erforderliche Zweigstelle wie der GUI-Proxy, sondern ist in der Lage eine direkte Kommunikation mit der Mittelschicht aufzunehmen. Das Plugin-System gestaltet sich nun ergonomischer in zweierlei Hinsicht. Zum einen ist es nun einfacher neue Plugins zu entwickeln, da die Schnittstelle erheblich verkleinert wurde und nun allgemeiner ist. Es können nahezu alle Bereiche durch Plugins erweitert werden und nicht mehr nur Netzwerkansichten und Systeminformationsansichten. Weiterhin ist die Einbindung ergonomischer gestaltet worden. Plugins melden sich nun automatisch durch vorhanden sein im Installationsverzeichnis der Oberfläche an und müssen nun nicht mehr von Hand kompliziert eingebunden werden. Das Sammeln von Systeminformationen wird nun durch ein effektives Zusammenspiel von Oberfläche und Mittelschicht realisiert. Informationen kommen in Echtzeit beim Benutzer an und haben keine Latenzzeit mehr durch Polling verfahren. Die Anzeige der Informationen selbst wurde durch ein bewährtes Paket zur Visualisierung von statistischen Daten ersetzt. Damit ergeben sich eine Vielzahl von neuen Möglichkeiten der Darstellung.
Doch nicht alle Anforderungen konnten erwartungsgemäß durchgeführt werden. Es ist nicht gelungen einige besondere Darstellungsmöglichkeiten der vorigen Version im vollen Umfang auszubessern. Die 2D Ansicht sollte in einen 2D Editor umgewandelt werden. In dem es möglich ist interaktiv mit einem visuellen Feedback ein

Netzwerk zu modellieren. Dieses scheiterte zum einen an den Altlasten des benutzen Codes. Hier gestaltete sich eine Überführung von Objekten und Informationen zur Laufzeit schwierig. Andererseits mussten alle Veränderungen des Netzwerkes von der Mittelschicht bestätigt werden. Dieses erfordert eine Absicherung bei jedem Schritt. Weiterhin hatte dieses Konzept eine starke Abhängigkeit von der Komplementierung verschiedener anderer Konzepte innerhalb der Benutzungsoberfläche, so dass die Entwicklung an das Ende des Projektes verschoben wurde.
Insgesamt gesehen wurde das von der Gruppe gesteckte Ziel erreicht.

4. Mittelschicht

Die Mittelschicht ist die zentrale Komponente des unIDS-Systems. Sie ist für die Verwaltung aller anderen Komponenten, den Nachrichtenaustausch und die Nachrichtenanalyse verantwortlich. Im Folgenden wird zunächst ein Überblick über die Mittelschicht gegeben. Daran anschließend werden die Entwurfsentscheidungen sowie die Architektur der Mittelschicht erläutert.

Danach werden die einzelnen Komponenten der Mittelschicht geordnet nach Aufgabenbereichen beschrieben. Nach der Betrachtung der Komponenten folgt in Abschnitt 4.12 eine Darstellung der verschiedenen Erweiterungsmöglichkeiten der Mittelschicht. Abschließend folgt in Abschnitt 4.13 eine Beschreibung der nicht fertiggestellten Features, der Stellen, an denen Refactoring-Bedarf besteht, sowie eine Liste der bekannten Fehler mit einem daran anschließenden Fazit.

4.1 Aufgaben der Mittelschicht

In der Mittelschicht als zentrale Komponente fließen ganz unterschiedliche Aufgaben zusammen. Diese reichen von der Verwaltung der verschiedenen unIDS-Systemkomponenten über die Organisation des Systemzustandes bis hin zur Nachrichtenverarbeitung und -analyse.

In diesem Abschnitt wird zunächst ein Überblick über die verschiedenen grundsätzlichen Aufgaben der Mittelschicht gegeben. Detailliertere Beschreibungen sind in den daran anschließenden Abschnitten enthalten.

4.1.1 Verwaltung der unIDS-Systemkomponenten

Eine wesentliche Aufgabe der Mittelschicht als zentraler Komponente des unIDS-Systems ist die Verwaltung der anderen am System beteiligten Komponenten. Hierzu zählen auf der einen Seite die in Abschnitt 5 vorgestellten Agenten sowie auf der anderen Seite die in Abschnitt 3 beschriebene Benutzerschnittstelle.

Allen Komponenten innerhalb des unIDS-Systems wird dabei ein systemweit eindeutiger Identifikator zugeordnet. Über diesen Identifikator kann die Komponente dann innerhalb des unIDS-Systems angesprochen werden. Die Mittelschicht stellt dabei

sicher, dass jeder neu vergebene Identifikator systemweit eindeutig ist und dass an ihn adressierte Nachrichten über die Mittelschicht ihr Ziel erreichen können. Für das Kommunikationszept des unIDS-Sytem vergleiche auch Kapitel 2.2.

4.1.2 Nachrichtenaustausch

Neben der Verwaltung der das unIDS-System ausmachenden Komponenten ist die Mittelschicht dafür verantwortlich, dass zwischen den Komponenten Nachrichten ausgetauscht werden können. Hierbei sind generell zwei Nachrichtenklassen zu unterscheiden: Auf der einen Seite werden Nachrichten innerhalb eines speziellen Umschlags versandt, der den Empfänger und Absender in Form des zuvor beschriebenen Identifikators enthält. Die zwischen Agent beziehungsweise Proxy und Mittelschicht ausgetauschten Nachrichten gehören grundsätzlich dieser Klasse an[1] [2]. Auf der anderen Seite benötigen die Nachrichten zwischen der grafischen Benutzerschnittstelle und der Mittelschicht diesen Umschlag nicht, da sich der Empfänger implizit aus der Verbindung ergibt.

Die Mittelschicht muss neben dem reinen Austausch der Nachrichten auch in der Lage sein, diese zu verarbeiten und gegebenenfalls zugehörige Folgenachrichten oder Statusänderungen auszulösen. Dies bedeutet zum Beispiel, dass bei eingehenden IDMEF-Nachrichten diese dem richtigen Device-Eintrag zugeordnet werden und der Status für diesen Eintrag entsprechend verändert wird. Zusätzlich sind IDMEF-Nachrichten direkt an jede verbundene Benutzerschnittstelle sofort weiterzuleiten.

4.1.3 Verwaltung des Systemzustands

Das unIDS-System als soches besitzt einen systemweiten Zustand, der über die Mittelschicht verwaltet und auch aktualisiert wird. Andere Komponenten können den systemweiten Zustand durch von ihnen stammende Nachrichten beeinflussen oder Teile davon abfragen. Der Zustand teilt sich in einen persistenten Teil und einen Laufzeitteil, der bei jedem Neustart verloren geht.

Im Systemzustand sind die Informationen über alle bekannten Agenten, die Benutzerdaten sowie die letzten Statusinformationen zu den verschiedenen Devices enthalten. Im Allgemeinen bilden die Agenten in Bezug auf den Systemzustand die Informationslieferanten, damit sind sie in der Regel diejenigen, die den Systemzustand beeinflussen. Die grafische Benutzerschnittstelle auf der anderen Seite benötigt für die Visualisierung Informationen über den Systemzustand und bildet daher eine Senke.

4.1.4 Verarbeitung und Analyse

Die vor allem von den Agenten gelieferten Informationen müssen zunächst den richten Datensätzen innerhalb des Systemzustandes zugeordnet werden. Dazu muss die Mittelschicht die in den Nachrichten enthaltenen Informationen analysieren und in den Systemzustand einpflegen.

[1] Ausgenommen IDMEF-Nachrichten, da dies nicht mit den entsprechenden RFCs vereinbar wäre.

[2] FÜR IDMEF und IDXP siehe die Abschnitte 2.2.4 und 2.2.5

[2] vgl. dazu Abschnitt 2.2.4

Den Regelfall bilden dabei die Statusinformationen der verschiedenen Plugins der Agenten. Die in diesen Nachrichten enthaltenen Informationen müssen im Wesentlichen unverändert in den Systemzustand übernommen werden. Daneben werden durch die Agenten auch IDMEF-Nachrichten versandt, die als solche zwar direkt an alle angemeldeten Benutzerschnittstellen weitergeleitet werden, darüber hinaus aber auch zu Veränderungen im Systemzustand führen müssen. Auch hier muss zunächst eine korrekte Zuordnung zu dem entsprechenden Datensatz im Systemzustand hergestellt werden. Für diesen Datensatz wird dann der „Status-Parameter“ aktualisiert.

4.2 Anforderungen an die Mittelschicht

Im Folgenden werden die grundsätzlichen Anforderungen an die Mittelschicht in einer Aufzählung dargestellt.

- **Verbindung zu vielen Komponenten** – Sowohl zu den Agenten als auch zu den grafischen Benutzerschnittstellen müssen unter Umständen eine Vielzahl an Verbindungen unterhalten werden können.

- **Robustheit** – Da ein Ausfall der Mittelschicht das gesamte unIDS-System zum Stillstand bringen könnte, muss die Mittelschicht in besonderem Maße robust sein. Dies bedeutet, dass auftretende Fehler möglichst nicht zu einem kompletten Ausfall der Mittelschicht führen sollten.

- **Schnelligkeit** – Die Nachrichten sollten möglichst so schnell verarbeitet werden können, dass für den Benutzer an der grafischen Benutzerschnittstelle keine unangenehmen Verzögerungen auftreten. Dies heißt für die Mittelschicht, dass insbesondere bei Nachrichten, die direkt an die grafischen Benutzerschnittstellen weiterzuleiten sind, die Verzögerungszeiten möglichst gering sein sollten.

- **Persistenz** – Der unIDS-Systemzustand muss auch nach einem Neustart der Mittelschicht erhalten bleiben. Bestimmte Teile des Zustandes, die nur zur Laufzeit relevant sind dürfen dabei verloren gehen.

- **Portabilität** – Die Mittelschicht *sollte* auf möglichst vielen Betriebssystemen nutzbar sein. Sie *muss* mindestens auf Linux-kompatiblen Betriebssystemen nutzbar sein.

4.2.1 Anwendungsfälle

Um die Aufgaben der Mittelschicht besser abschätzen zu können, wurden verschiedene Anwendungsfälle durchgespielt.
Die hier vorgestellten Anwendungsfälle sind so sortiert das zuerst alles beschrieben wird was sich mit den Agenten beschäftigt. Danach folgen alle Anwendungsfälle die die GUI betreffen, und dann alle Fälle die in kein Schema passen.

4.2.1.1 Agent meldet sich an

Wenn ein neuer Agent sich erstmalig am unIDS-System anmeldet, muss er durch die Mittelschicht einen neuen systemweit eindeutigen Identifikator zugeordnet bekommen. Nach der erstmaligen Anmeldung muss der Agent bei jedem neuen Verbindungsaufbau seinen alten Identifikator übermitteln, damit die von ihm übertragenen Daten korrekt zugeordnet werden können. Dabei sind folgende Fälle zu unterscheiden:

- **Neuer Agent** – Es ist ein neuer Identifikator zuzuordnen. Für den Agenten wird ein neuer Device-Eintrag im persistenten unIDS-Systemzustand hinzugefügt.
- **Bekannter Identifikator** – Der Agent ist im System bereits registriert.
- **Unbekannter Identifikator** – Es ist ein Fehler aufgetreten. Der Agent gibt vor, bereits im System registriert zu sein, aber der entsprechende Datensatz ist nicht vorhanden.

Nach einer erfolgreichen Anmeldung ist der Agent im persistenten Teil des unIDS-Systemzustands eingetragen. Hat sich ein Agent erstmalig am unIDS-System registriert ist automatisch ein zugehöriger Device-Eintrag erstellt worden. Diesem Device werden alle foldenden Statusdaten durch die Mittelschicht automatisch zugeordnet.

4.2.1.2 Agent meldet sich ab

In Fällen, in denen das System, auf dem der Agent betrieben wird, neu gestartet werden muss, sollte sich der Agent nach Möglichkeit von unIDS-System abmelden. Bricht die Verbindung ohne Abmeldung ab, *sollte* der Agent automatisch abgemeldet werden. Dies macht das System robuster für Fälle, in denen ein System auf dem ein Agent betrieben wird unkontrolliert ausfällt[3] und danach wieder gestartet wird. Hiernach *muss* der Agent sich mit seinem alten Identifikator neu verbinden können.

Nach der Abmeldung darf der Agent nicht aus dem Systemzustand entfernt werden, da in der Regel von einer späteren neuen Verbindung auszugehen ist. Der Agent *soll* als nicht verbunden im persistenten Systemzustand markiert werden.

4.2.1.3 Agent wird aus dem System entfernt

Im Normalfall bleiben einmal registrierte Agenten auf unbestimmte Zeit im unIDS-Systemzustand registriert. Fallen Agenten weg und sollen sie aus dem unIDS-Systemzustand gelöscht werden, so erfordert dies einen manuellen Eingriff des Benutzers, da ein Agent nicht mitteilen kann, dass er gelöscht wurde. Hierzu *muss* eine entsprechende Anforderung durch die grafische Benutzerschnittstelle unterstützt und entsprechend verarbeitet werden.

[3] Dies kann z.B. bei einem Stromausfall oder Hardware-Fehler der Fall sein.

4.2.1.4 Benutzer meldet sich an

Ein Benutzer, der das unIDS-System nutzen will, meldet sich über die in Abschnitt 3 beschriebene grafische Benutzerschnittstelle am unIDS-System an. Die grafische Benutzerschnittstelle übermittelt daraufhin die Benutzerdaten „Benutzername“ und „Kennwort“ an die Mittelschicht, um Zugriff auf Informationen des unIDS-Systemzustands zu erhalten. Es sind folgende Szenarien möglich:

- **Korrekte Zugangsdaten** – Der Benutzerschnittstelle wird Zugriff auf Informationen über den unIDS-Systemzustand gewährt.
- **Falscher Benutzername** – Der Zugriff auf das unIDS System wird verweigert.
- **Falsches Kennwort** – Der Zugriff auf das unIDS-System wird verweigert.

Für den Fall, dass die Zugangsdaten korrekt sind, sendet die Mittelschicht an die Benutzerschnittstelle eine positive Quittung. In allen anderen Fällen *muss* die Mittelschicht eine negative Quittung an die Benutzerschnittstelle übermitteln. Dabei darf es nicht ersichtlich sein, ob der Benutzername unbekannt oder das eingegebene Kennwort nicht korrekt ist.

Nach einer erfolgreichen Anmeldung *soll* nur der Zugriff auf die Bereiche des unIDS-Systemzustands möglich sein, für die der Benutzer die entsprechenden Rechte hat. Die Sicht auf den Systemzustand *soll* entsprechend der Rolle beziehungsweise Rechte des Benutzers eingeschänkt sein. So lange der Benutzer angemeldet ist, *sollen* kritische Meldungen sofort an die entsprechende Benutzerschnittstelle weitergeleitet werden.

4.2.1.5 Benutzer meldet sich ab

Beim Absturz oder einer korrekten Terminierung der grafischen Benutzerschnittstelle ist der Systemzustand so weit zu aktualisieren, dass der entsprechende Benutzer nicht mehr als verbunden registriert ist.

Nach einer ordnungsgemäßen Abmeldung *muss* eine erneute Anmeldung des Benutzers möglich sein.

4.2.1.6 Konfiguration eines Agenten geändert

Über die Benutzerschnittstelle kann der Benutzer die Konfiguration von Agenten beeinflussen. Die Benutzerschnittstelle sendet hierzu eine entsprechende Anfrage an die Mittelschicht. Die Mittelschicht verarbeitet diese Anfrage und passt gegebenenfalls den unIDS-Systemzustand entsprechend an und informiert den betroffenen Agenten über die Änderung. Es können folgende Unterfälle auftreten:

- **Agent ist verbunden** – Der Agent wird sofort über die Änderungen informiert und der Systemzustand wird aktualisiert. Vergleiche dazu auch 5.2.4
- **Agent ist nicht verbunden** – Der Systemzustand wird akualisiert.

- **Agent unbekannt** – Es wird ein Fehler zurück gemeldet. Der Systemzustand bleibt unverändert.

In den Fällen, in denen ein Agent unter dem gegebenen Identifikator im unIDS-System bekannt ist, wird der Systemzustand in jedem Fall aktualisiert. Falls eine Verbindung zu dem entsprechenden Agenten besteht, wird er sofort über die Änderung der Konfiguration informiert, in allen anderen Fällen wird die Konfiguration bei der nächsten Verbindung des Agenten übermittelt.

Falls kein Agent mit dem gegebenen Identifikator im System bekannt ist, wird eine Fehlermeldung an die Benutzerschnittstelle zurückgesandt. Der Typ der Fehlermeldung ist „Objekt nicht gefunden."

4.2.1.7 Register-Anfrage für bestimmte Entitäten

Der Benutzer hat die Möglichkeit, über die grafische Benutzerschnittstelle nähere Informationen zu einzelnen Devices zu betrachten. Dazu registriert sich die Benutzerschnittstelle über eine „Register-Anfrage" bei der Mittelschicht. Die Mittelschicht sendet bei Statusänderungen der betroffenen Devices entsprechende Informationsnachrichten an die Benutzerschnittstelle, damit die Informationen in Echtzeit zur Verfügung stehen. Es sind folgende Unterfälle zu unterscheiden:

- **Register Anfrage** – Der Benutzer wird für die in der Anfrage enthaltenen Entitäten eingetragen.
- **Unregister Anfrage** – Der entsprechende Eintrag wird gelöscht.
- **Nicht vorhandene Entität** – Es wird mit der Fehlermeldung „Objekt nicht gefunden" geantwortet.

Nach einer Registrierung *müssen* alle Statusänderungen der betroffenen Entität umgehend an die entsprechende Benutzerschnittstelle weitergeleitet werden, damit die Informationen dem Benutzer in Echtzeit zur Verfügung stehen. Die Informationen über die Registrierungen werden im nicht-persistenten Bereich des Systemzustands gehalten. Nach einer „Unregister-Anfrage" werden Statusänderungen der betroffenen Entitäten nicht mehr an die Benutzerschnittstelle weitergeleitet.

Register-Anfragen, die sich auf nicht oder nicht mehr vorhandene Entitäten beziehen *sind* mit einer Fehlermeldung vom Typ „Objekt nicht gefunden" zu beantworten.

4.3 Entwurfsentscheidungen

Im Vorfeld des Entwurfs der Mittelschicht waren verschiedene Entwurfsentscheidungen zu treffen, die den gegebenen Anforderungen aber auch dem speziellen Umfeld des Entwurfs und der Implementierung gerecht werden sollten. Das Entwicklungsumfeld ist insbesondere speziell, da die Umgebung der Mittelschicht (beispielsweise Agenten, Proxies, grafische Benutzerschnittstelle, Datenhaltung) teilweise sehr undurchsichtig war. Erst im Projektverlauf konkretisiert sich dies, da die meisten der Komponenten, mit denen die Mittelschicht zusammenarbeitet, erst entwickelt werden mussten. Speziell für die Datenhaltung mittels einer Datenbank zeigten sich hier

wie in Abschnitt 4.8 dargestellt besondere Schwierigkeiten durch den Entwicklungsprozess für das gesamte unIDS-System.

In diesem Abschnitt werden zunächst die für die Mittelschicht getroffenen Entscheidungen zum Vorgehensmodell beschrieben. Daran anschließend werden die Entscheidungen zur verwendeten Programmiersprache sowie zu den genutzten Biliotheken und Frameworks dargestellt und begründet.

4.3.1 Vorgehensmodell

Da zu Projektbeginn nicht alle Anforderungen eindeutig definiert waren und die Auswirkungen verschiedener Entwurfsentscheidungen wie beispielsweise der Realisierung eines persistenten Systemzustands nur schwer abgeschätzt werden konnten, wurde die Entwicklung der Mittelschicht an ein evolutionäres Vorgehensmodell angelehnt[4].

So wurde zunächst eine sehr einfache Variante der Mittelschicht realisiert, die ihren Zustand komplett im Arbeitsspeicher halten konnte und keinerlei Persistenz zur Verfügung stellte und keine Benutzer und Zugriffsberechtigungen kannte.

Ausgehend von dieser einfachen Basis wurden neue Funktionalitäten spezifiziert, um die die Mittelschicht Schritt für Schritt erweitert werden sollte. Für die Verwaltung und Planung der neuen Funktionen wurde das Projektmanagement-Werkzeug `trac` verwendet, das eine Art Trouble-Ticket-System bietet. Einzelne Tickets lassen sich dabei Meilensteinen zuordnen, so dass immer ein Überblick über den gegenwärtigen Stand der Entwicklung möglich ist.

Die Entwicklung wurde dann etwa in Zwei-Wochen-Zyklen eingeteilt, an deren Anfang jeweils eine Liste mit geplanten neuen Funktionen stand. Zum Ende des Entwicklungszyklus sollte eine möglichst stabile Version stehen, anhand derer die zu Beginn jedes Zyklus getroffenen Entwurfsentscheidungen beurteilt und dann gegebenenfalls korrigiert werden konnten.

Zusätzlich änderten sich fortlaufend die Anforderungen durch die anderen Komponenten, die parallel entwickelt wurden, so dass auch hierdurch Modifikationen der Entscheidungen vorzunehmen waren. Speziell unter diesem Gesichtspunkt hat sich dieses Vorgehen bewährt.

4.3.2 Programmiersprache Python

Zur Auswahl für die Programmiersprache standen zum einen Java, wodurch sich Synergien zwischen der Entwicklung der Mittelschicht und der grafischen Benutzeroberfläche ergeben hätten und Python, einer interpretierten, objektorientierten Sprache, die ihre Stärken vor allem in der schnellen Entwicklung von Prototypen und einer umfangreichen Systembibliothek hat. Beide Sprachen sind ähnlich portabel.

Da für die Entwicklung zunächst kurzfristig ein lauffähiger Prototyp bestehen sollte, an dem weitere Entscheidungen getestet werden sollten, fiel die Wahl zunächst auf Python. Obgleich hierzu im Vergleich zu Java ein zunächst erhöhter Einarbeitungsaufwand nötig war, da bisher kaum Entwicklungserfahrungen in Python vorhanden waren.

[4] vgl. [Somm01] S. 58ff.

Die guten Erfahrungen bei der Erstellung des Prototypen und die umfangreichen zur Verfügung stehenden Bibliotheken führten im Laufe der Entwicklung zu der Entscheidung, auch die weitere Implementierung der Mittelschicht in Python zu realisieren. Speziell auch die Möglichkeit in Interaktion mit dem Interpreter neue Entwicklungen oder auch Bibliotheken in „Echtzeit“ zu testen hat sich als großer Vorteil im Vergleich zur Java-basierten Entwicklung gezeigt.

Im Vergleich zu Java stellen sich aber auch einige Nachteile insbesondere bei der Typsicherheit dar. Hier ist in hohem Maße ein sehr diszipliniertes Vorgehen der Entwickler gefordert, um übersichtlichen und verständlichen zu implementieren.

4.3.3 Verwendete Bibliotheken und Frameworks

Für die Entwicklung der unIDS-Mittelschicht wurden verschiedene Bibliotheken beziehungsweise Frameworks eingesetzt, die entweder bei der Entwicklung unterstützen sollen oder aber direkter Bestandteil der Funktionalität der Mittelschicht sind. Zunächst werden die Entscheidungen für Bibliotheken und Frameworks beschrieben, die einen direkten funktionalen Bestandteil der Mittelschicht ausmachen. Daran anschließend folgen die Bibliotheken und Frameworks, die eher zur Unterstützung der Entwicklung dienen[5].

4.3.3.1 Funktionale Bestandteile

Als Basis für die Entwicklung der Mittelschicht wurde das in Abschnitt 2.2.7 beschriebene Framework `Twisted` verwendet. Die Entscheidung erfolgte im Wesentlichen aus zwei Gründen:

- **Solides Serverframework** – `Twisted` bildet ein solides Serverframework. Es bringt eine Ereignissteuerung, die Verwaltung von Thread-Pools, robuste Netzwerkzugriffe sowie alle nötigen Funktionen, um schnell stabile Programme zu entwickeln, die als Serverprozess unter definierten Benutzerrechten laufen. Damit konnte die Entwicklung der Mittelschicht um wesentliche Arbeitsschritte erleichtert werden.
- **Komponentenkonzept** – `Twisted` bringt daneben ein umfangreiches Konzept zur Aufteilung der Anwendung in verschiedene loose gekoppelte Komponenten mit. Insbesondere in Bezug auf die unterschiedlichen Aufgaben der Mittelschicht sollte dies für eine übersichtlich gestaltete Architektur genutzt werden.

Die Grundlage für das Komponentenkonzept von `Twisted` bilden die im Rahmen des ZOPE-Projekts[6] entwickelten Python-Erweiterungen für Interfaces, so dass auch in Python das Konzept von Schnittstellen verwendet werden kann.

Als Basis für die Kommunikation mittels des BEEP-Frameworks wurde die Implementierung `BEEPy` verwendet. Die Bibliothek ist zum einen auf der Basis von Twisted

[5] Die Grenze ist an dieser Stelle nicht ganz klar zu ziehen und es existiert eine gewisse Grauzone, bei der beide Punkte zum Teil zutreffen. Die Einteilung wurde mit der Intention gewählt, eine bessere Orientierung zu ermöglichen.

[6] Siehe auch [Zopeb]

erstellt und wird zum anderen auch für die Implementierung des Agenten verwendet. Während der Entwicklung der Mittelschicht hat sich die relativ dünne Entwicklergemeinde um das `BEEPy`-Projekt als nachteilig erwiesen, da zum Teil erhebliche Fehler innerhalb der `BEEPy`-Implementierung mühsam selbst beseitigt werden mussten. [7].

Für den persistenten Systemzustand war zunächst eine Eigenimplementierung mit einem RDBMS im Hintergrund angedacht. Diese Entscheidung musste allerdings während der Entwicklung revidiert werden, da mit dieser Lösung nicht genug Flexibilität zur Reaktion auf geänderte Anforderungen durch die Umgebung der Mittelschicht geblieben war. Als wesentlich flexiblere Alternative wurde die Objektdatenbank `ZODB` des ZOPE-Projekts verwendet. Detailliertere Informationen hierzu finden sich im Abschnitt 4.9.

4.3.3.2 Nichtfunktionale Bestandteile

Im Zuge der Revidierung der Entwurfsentscheidungen zum persistenten Systemzustand und durch sich weiter ändernde Anforderungen an die Struktur des Systemzustands wurde auch die Struktur der Daten umgestellt. Zur Verwaltung der verschiedenen Strukturen über den vorhandenen Datensätzen wurden mehrere gerichtete Graphen eingesetzt, um auch bei weiteren Änderungen flexibel reagieren zu können. Als Basis für die Verwaltung der Datenstruktur wurde die Bibliothek `pygraphlib`[8] eingesetzt. Detailliertere Informationen folgen hierzu in Abschnitt 4.8

Da die Mittelschicht nicht über eine direkte Benutzerschnittstelle verfügt, wurde konsequent der Logging-Mechanismus von Python eingesetzt. Dieser stellt eine hierariche Logging-Umgebung zur Verfügung. Das Ziel dieser Entscheidung war die Visualisierung der Vorgänge innerhalb der Mittelschicht für die beteiligten Entwickler. Der Umfang der auszugebenden Meldungen sollte für die verschiedenen Bereiche der Mittelschicht einzeln konfigurierbar sein.

Zur Steigerung der Qualität bei der Entwicklung der Mittelschicht und zur Überprüfung der Spezifikation wurde vor allem bei der Entwicklung der Nachrichtenverarbeitung für die grafische Benutzerschnittstelle auf das in Python vorhandene Framework für Unit-Tests zurückgegriffen.

4.4 Architektur

In diesem Abschnitt wird die Architektur der Mittelschicht näher beleuchtet. Zunächst wird hierzu ein Überlick über das entwickelte Architekturmodell gegeben. Anschließend wird das für die Mittelschicht entwickelte Komponentenmodell beschrieben. Darauf folgend wird die Komponente `ServiceManager` beschrieben, die für die Verwaltung der Mittelschicht an sich und ihrer Bestandteile verantwortlich ist. Abschließend wird das Format der Konfigurationsdatei beschrieben, welches sich stark an dem Architekturmodell der Mittelschicht orientiert.

[7] Die während der Entwicklung der Mittelschicht beim BEEPy-Projekt eingestellten BUG-Reports sind im Wesentlichen zum Zeitpunkt der Erstellung dieser Dokumentation noch offen gewesen.

[8] Siehe auch [Albe]

4.4.1 Architekturmodell der Mittelschicht

Wie schon in Abschnitt 4.1 angedeutet, kommen in der Mittelschicht ganz unterschiedliche Aufgabenbereiche zusammen. Um die Mittelschicht möglichst übersichtlich zu halten und um die einzelnen Bestandteile der Mittelschicht voneinander weitestgehend unabhängig zu halten, wurde die Architektur sehr stark komponentenbasiert aufgebaut, wie in Abbildung 4.1 dargestellt.

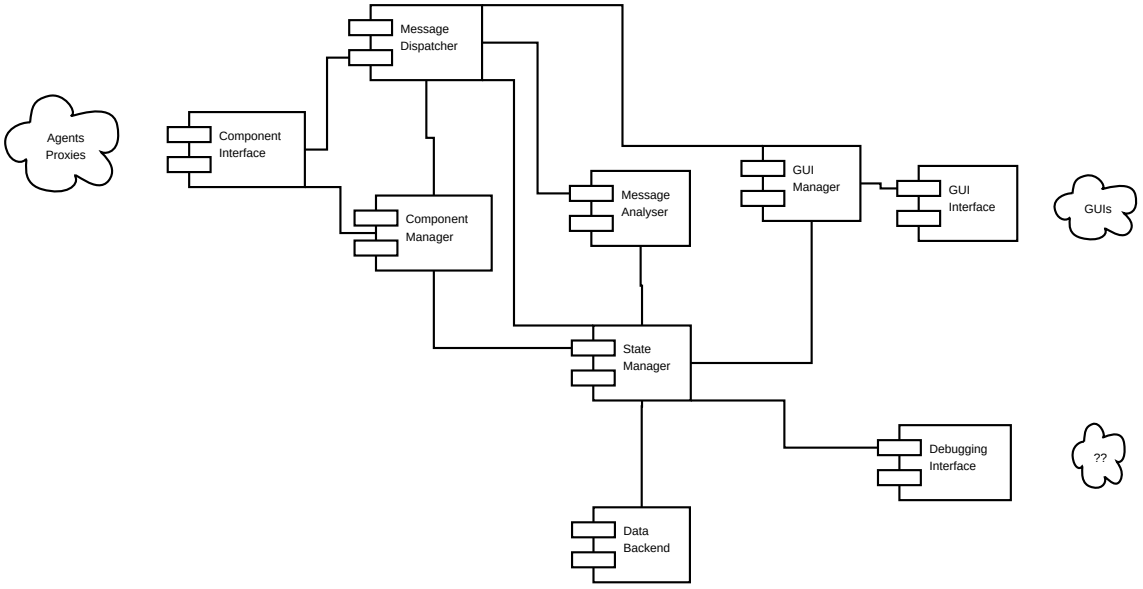

Abbildung 4.1: Schematische Darstellung der Architektur der Mittelschicht

Im Folgenden werden die in Abbildung 4.1 dargestellten Komponenten in einer Übersicht vorgestellt. Eine ausführlichere Beschreibung jeder einzelnen Komponente folgt in den späteren Abschnitten.

ComponentInterface – Das `ComponentInterface` ist für die Verbindung zu den einzelnen Komponenten (Agenten und Proxies) verantwortlich.

ComponentManager – Diese Komponente verwaltet die verschiedenen unIDS-Komponenten, die über das ganze Netzwerk verteilt sein können. Eine der wichtigsten Aufgaben ist die Sicherstellung der eindeutigen Identifizierbarkeit jeder unIDS-Komponente.

DataBackend – Das `DataBackend` ist für die Datenhaltung verantwortlich. Der vom `StateManager` festgehaltene Zustand muss passend auf eine Datenbank[9] abgebildet werden.

DebuggingInterface – Diese Komponente ist für Debuggingzwecke vorgesehen. Sie wird bei Bedarf implementiert. Ihr Vorhandensein darf das Verhalten des Unids-Systems genauso wenig ändern, wie ihr Fehlen[10].

[9] Hier ist nicht zwingend eine relationale Datenbank vorgesehen. Wie in Abschnitt 4.3.3 beschrieben wurde im Laufe der Entwicklung eine Objektdatenbank verwendet.

[10] Diese Komponente wurde tatsächlich nicht implementiert, da über verschiedene andere Möglichkeiten ausreichend Einblicke in die Abläufe der Mittelschicht gewonnen werden konnten.

GuiInterface – Das `GuiInterface` ist mit dem `ComponentInterface` vergleichbar. Es ist für die Verbindungen zu den aktiven Benutzerschnittstellen verantwortlich. Auf Grund der besonderen Eigenarten der Verbindung zur grafischen Benutzerschnittstelle wurde diese Funktion nicht mit im `ComponentInterface` implementiert.

GuiManager – Der `GuiManager` ist dem `ComponentManager` sehr ähnlich. Die Verwaltung der grafishen Benutzerschnittstellen unterscheidet sich jedoch erheblich von der Verwaltung der anderen Komponenten, so dass hierfür eine eigene Komponente geschaffen wurde.

MessageAnalyser – Der `MessageAnalyser` ist verantwortlich für die Analyse der eingehenden Nachrichten im Kontext des aktuellen Systemzustands[11].

MessageDispatcher – Der `MessageDispatcher` ist innerhalb der Mittelschicht für die richtige Zustellung der Nachrichten verantwortlich. Dabei können vom `ComponentInterface` kommende Nachrichten entweder direkt für die grafische Benutzerschnittstelle oder für die Mittelschicht bestimmt sein.

StateManager – Der `StateManager` verwaltet den Unids-Zustand. Wie schon in den Anforderungen in Abschnitt 4.2 beschrieben, besteht der Zustand aus einem persistenten und aus einem nicht persistenten Teil.

4.4.2 Komponentenmodell

Jede größere Funktionseinheit der Mittelschicht ist in eine oder mehrere Komponenten gekapselt. Für jede Komponente der Mittelschicht ist eine Schnittstelle über die Konfigurationsdatei definiert, die durch diese Komponente bereit gestellt wird. Die verschiedenen Komponenten der Mittelschicht kommunizieren ausschließlich über den `ServiceManager` und die der jeweiligen Komponente zugeordnete Schnittstelle[12] miteinander.

Alle Komponenten der Mittelschicht werden ausschließlich durch den `ServiceManager` verwaltet. Dazu müssen sie die in Abbildung 4.2 dargestellte Schnittstelle *unids.middlelayer.interfaces* implementieren[13].

Die Komponenten der Mittelschicht können dabei die in Abbildung 4.3 dargestellten Zustände einnehmen:

- **UNINITIALIZED** – Jede Komponente der Mittelschicht muss sich nach dem erstellen einer Instanz in diesem Zustand befinden.
- **INITIALIZED** – Nach dem Aufruf der Methode `init` befindet sich die Komponente in diesem Zustand oder im Zustand ERROR, falls ein Fehler auftrat, der nicht behoben werden konnte.

[11] Im Rahmen der Implementierung wurde die Funktionalität der Analyse in den Handlern zu den einzelnen Nachrichtenklassen untergebracht, so dass bisher keine eigene Komponente zur Analyse vorhanden ist.

[12] Die Schnittstelle muss dabei eine Ableitung der Schnittstelle *zope.interface.Interface* sein.

[13] Siehe dazu auch die Klasse `unids.middlelayer.interfaces.AbstractMiddlelayerComponent` im Quelltext der Mittelschicht.

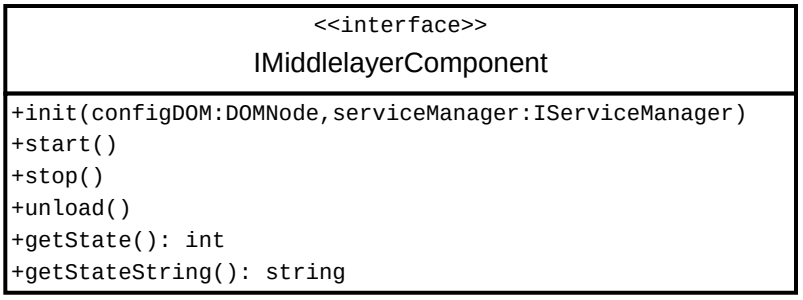

Abbildung 4.2: Schnittstelle IMiddlelayerComponent.

- **RUNNING** – Nach der Aktivierung durch die Methode `start` befindet sich die Komponente in diesem Zustand. So lange sie in diesem Zustand ist, kann sie durch die anderen Komponenten der Mittelschicht verwendet werden. Falls bei der Aktivierung ein nicht behebbarer Fehler auftritt, befindet sich die Komponente im Zustand ERROR.
- **ERROR** – Dieser Zustand wird eingenommen, wenn bei einem der Zustandswechsel ein nicht behebbarer Fehler aufgetreten ist. So lange die Komponente in diesem Zustand ist, kann sie nicht genutzt werden.
- **UNKNOWN** – Dieser Zustand ist die Vorbelegung für das Zustandsattribut. Falls dieser Wert erreicht wird, deutet dies auf einen Fehler in der Implementierung hin.

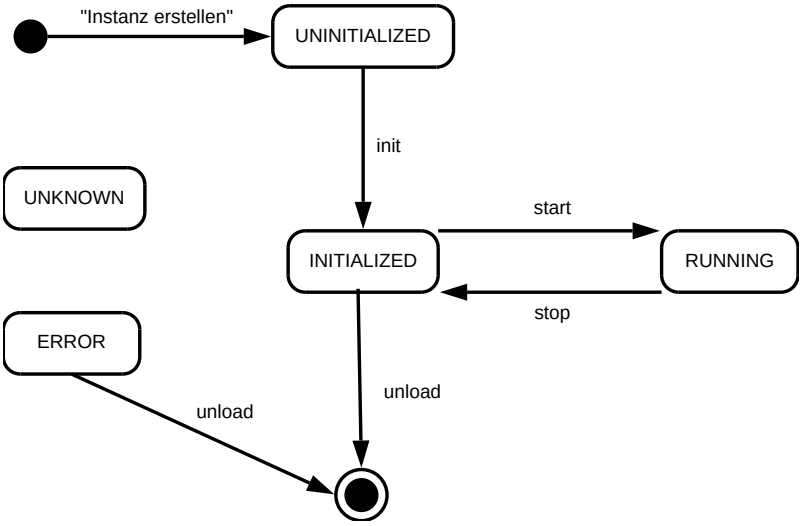

Abbildung 4.3: Mögliche Zustände einer Komponente der Mittelschicht.

Die Methoden in Abbildung 4.2 haben dabei folgende Bedeutung:

- **init** – Diese Methode wird direkt nach dem Instanziieren der Komponente aufgerufen. Der Komponente wird der für sie bestimmte Ausschnitt aus der Konfigurationsdatei der Mittelschicht im Parameter *configDOM* übergeben. Daneben erhält die Komponente über den Parameter *serviceManager* eine Referenz auf den sie kontrollierenden `ServiceManager`.

- **start** – Über diese parameterlose Methode wird der Komponente mitgeteilt, dass sie sich aktivieren soll. Nach dem Aufruf dieser Methode muss die Komponente vollständig nutzbar sein oder aber sich in einem Fehlerzustand befinden.
- **stop** – Diese Methode wird aufgerufen, um die Komponente zu deaktivieren. Die Komponente bleibt zunächst initialisiert, aber gegebenenfalls sind notwendige Resourcen freigegeben.
- **unload** – Beim Beenden der Mittelschicht wird der Komponente über diese Methode mitgeteilt, dass sie sich entgültig beenden muss.

4.4.3 ServiceManager

Der `ServiceManager` ist eine spezielle Komponente der Mittelschicht, die im in Abbildung 4.1 dargestellten Architekturmodell zunächst nicht enthalten ist. Der `ServiceManager` ist für die Verwaltung der Mittelschicht verantwortlich. Beim Start der Mittelschicht ist diese Komponente dafür verantwortlich, die Konfigurationsdatei einzulesen und die innerhalb der Konfigurationsdatei beschriebenen Komponenten zu erstellen und mit den konfigurierten Parametern zu starten. Nach dem Start des unIDS-Systems stellt der `ServiceManager` den Kontext für jede Komponente dar. Die die Mittelschicht ausmachenden Komponenten kommunizieren ausschließlich über den `ServiceManager` und definierte Schnittstellen miteinander. Das Format der Konfigurationsdatei und ihre Verarbeitung ist in Abschnitt 4.4.4 beschrieben. Zum Start der Mittelschicht arbeitet der `ServiceManager` die folgenden Schritte ab:

- **Laden der Konfigurationsdatei** – Zunächst wird die Konfigurationsdatei eingelesen. Diese Datei enthält weitere Informationen über die Zusammensetzung der Mittelschicht, insbesondere aus welchen Komponenten die Mittelschicht sich zusammensetzt.
- **Initialisieren der Logging-Umgebung** – Mit den Informationen aus der Konfigurationsdatei wird die Logging-Umgebung eingerichtet. Dies geschieht vor dem Erstellen der Komponenten, damit diese sich bereits an die voreingestellte Logging-Umgebung halten.
- **Erstellen der Komponenten** – Von jeder in der Konfiguration als *aktiv* konfigurierter Komponente wird eine Instanz erstellt.
- **Initialisierung der Komponenten** – Jede zuvor erstellte Komponente der Mittelschicht wird mit den innerhalb der Konfigurationsdatei angegebenen Parametern für diese Komponente initialisiert.
- **Start der Komponenten** – Nach der Initialisierung aller Komponenten, werden sie gestartet. Dabei kann sich eine Komponente darauf verlassen, dass alle anderen Komponenten bereits vollständig initialisiert sind.

Nach dem erfolgreichen Start der Mittelschicht wandelt sich der `ServiceManager` in eine eher passivere Komponente. Er ist dann mit einem Kontext vergleichbar. Die verschiedenen Komponten des unIDS-Systems können nun über den `ServiceManager` auf alle anderen Komponenten der Mittelschicht zugreifen. Für die Auswahl einer

Komponente dient eine Schnittstelle. Welche Implementierung der `ServiceManager` tatsächlich nutzt, ist innerhalb der Konfigurationsdatei eingestellt. Die einzige Information, die eine Komponente über die genutzten Kompontenten hat, sind die Schnittstellen, über die die Komponenten beim `ServiceManager` angefragt wurden.

Alle Komponenten der Mittelschicht erhalten eine Referenz auf den `ServiceManager`, der ihnen die in Abbildung 4.4 dargestellte Schnittstelle *IServiceManager* bereitstellt. Diese Schnittstelle spezifiziert zwei wesentliche Eigenschaften für den `ServiceManager`:

- Das „nur-lesen“ Attribut *context*, über das jede Komponente Zugriff auf den Kontext der Mittelschicht hat. Dieser Kontext besteht aus einer Zuordnung von Parameter-Namen zu Werten. Diese Parameter lassen sich über die Konfigurationsdatei frei konfigurieren[14].

- Die Methode `getRoleImplementation`, ist die Methode über die jede Komponente Zugriff auf andere Komponenten der Mittelschicht erhält. Als Parameter ist dieser Methode die Schnittstelle zu übergeben, die die gesuchte Komponente implementieren soll. Sofern in der Konfigurationsdatei für diese Schnittstelle ein entsprechender Eintrag vorhanden ist, liefert der `ServiceManager` eine Instanz, die diese Schnittstelle bereit stellt.

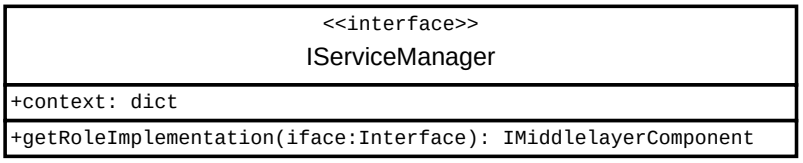

Abbildung 4.4: Die Schnittstelle IServiceManager.

Wird die Mittelschicht beendet, so ist der `ServiceManager` dafür verantwortlich, dass alle Komponenten der Mittelschicht sauber geschlossen werden, bevor die Mittelschicht an sich beendet wird. Dazu ruft der `ServiceManager` zunächst bei jeder Komponente der Mittelschicht die Methode `stop` auf, um die Komponenten zu deaktivieren. Danach wird für alle Komponenten die Methode `unload` aufgerufen. Erst wenn diese Methoden abgearbeitet wurden, beendet der `ServiceManager` die Mittelschicht.

4.4.4 Konfigurationsformat

Die unIDS-Mittelschicht wird über eine zentrale XML-Datei konfiguriert. In diesem Abschnitt wird zunächst der generelle Aufbau dieser Datei beschrieben und danach die möglichen Inhalte der verschiedenen Abschnitte.

Eine Beispielkonfiguration ist im Distributionspaket der Mittelschicht in der Datei `middlelayer/etc/ml_db.xconf` enthalten.

[14] Siehe hierzu auch Abschnitt 4.4.4

4.4.4.1 Überblick

Die Konfigurationsdatei der Mittelschicht ist zunächst innerhalb eines `config` Elements gekapselt. Innerhalb dieses Elements folgt zunächst ein Abschnitt `parameters`, in dem verschiedene Parameter für die Mittelschicht festgelegt werden können. Daran anschließend folgt ein Abschnitt `logging` über die das Logging-System sehr feingranular konfiguriert werden kann.

Danach folgt ein Block `instrumentation`, in dem alternative Implementierungen für bestimmte Schnittstellen festgelegt werden können. Den letzten Abschnitt bildet dann die `registry`, innerhalb derer die verschiedenen Schnittstellen und die möglichen Implementierungen spezifiziert werden.

4.4.4.2 Parameter

Innerhalb des Abschnitts `parameters` können verschiedene Parameter definiert werden, die Auswirkungen auf das Verhalten der Mittelschicht haben. Die hier festgelegten Parameter können später von den einzelnen Komponenten der Mittelschicht über das Attribute *context* des `ServiceManagers` abgefragt werden. Folgende Parameter werden bisher von der Mittelschicht unterstützt:

- `bugfix.slow_state_gui` – Ist dieser Parameter auf den Wert `true` gesetzt, werden Nachrichten an die grafische Benutzerschnittstelle mit einer zeitlichen Verzögerung gesendet.
- `debug.reply.exception` – Über diesen Parameter kann festgelegt werden, ob in Fehlermeldungen an die grafische Benutzerschnittstelle Hinweise auf die Fehlerursache untergebracht werden sollen. In der Regel wird ein Stack-Trace von der Exception ausgegeben, die den Fehler ausgelöst hat.
- `reply.with.reference` – Durch diesen Parameter kann festgelegt werden, ob normale Antworten im Allgemeinen eine zusätzliche Referenz auf die Entitäten oder Nachrichten enthalten sollen, die sie provoziert haben.

Der Parameter-Mechanismus wurde der Konfiguration der Mittelschicht mit der Intention hinzugefügt, flexibler auf Änderungen der Anforderungen reagieren und um nach Bedarf Debugging-Ausgaben aktivieren zu können. Zusätzlich sind temporäre Fehlerumgehungen in der Regel mit einem solchen Parameter versehen, damit sie ohne weiteres wieder deaktiviert werden können.

Die Namen der Parameter sind innerhalb der Mittelschicht als Konstanten definiert[15].

4.4.4.3 Logging

Durch den Abschnitt `logging` kann das Logging-System der Mittelschicht sehr feingranular konfiguriert werden. Zunächst kann innerhalb des Tags `logging` über das Attribut *level* ein Standardwert vorgegeben werden, der verwendet wird, falls nichts Weiteres festgelegt ist.

[15] Innerhalb des Moduls `unids.middlelayer.constants`.

Über einzelne `logger`-Einträge können dann genauere Einstellungen für einzelne Logger vorgenommen werden. Über das Attribut *name* wird der Name des zu konfigurierenden Loggers festgelegt und über das Attribut *level* kann dann ein entsprechender Logging-Level zugeordnet werden[16].

Innerhalb jedes Moduls der Mittelschicht sollen die Logger immer mit dem vollständigen Namen des Moduls initialisiert werden. Damit ist sichergestellt, dass sich das Logging-System auch hierarchisch konfigurieren lässt. Wird zum Beispiel der Logging-Level für einen Logger namens „unids.middlelayer“ gesetzt, so gilt dieser Wert auch für alle Module und Pakete unterhalb von „unids.middlelayer“, soweit für diese nicht ein anderer Logging-Level spezifiziert ist.

4.4.4.4 Konfigurationsmöglichkeiten

Über die Optionen -c und -l beim Erstellen der *unids.middlelayer.tap* bietet die Mittelschicht Konfigurationsmöglichkeiten, um diese bestmöglich an die eigenen Bedürfnisse anzupassen.
Der Schalter *-l* erlaubt es dabei, das Logging-Level der Mittelschicht zu setzen. Mögliche Werte sind hierbei:

- **CRITICAL** – Numerischer Wert: 50
- **ERROR** – Numerischer Wert: 40
- **WARNING** – Numerischer Wert: 30
- **INFO** – Numerischer Wert: 20
- **DEBUG** – Numerischer Wert: 10
- **NOTSET** – Numerischer Wert: 0

Ein Beispiel wäre:

```
mktap unids.middlelayer -l DEBUG -c ml_db.xconf
```

Je niedriger das Logging-Level eingestellt, desto mehr Rückmeldung gibt die Mittelschicht über ihre Aktivitäten. Dabei sollte bedacht werden, dass zuviele Ausgaben durch die Mittelschicht auch die Geschwindigkeit, sowie die Auslastung des Computers negativ beeinflussen können. Der Schalter *-l* ist optional.
Die wichtigere Konfigurationsmöglichlkeit stellt aber der Schalter *-c* da. Dahinter folgt normalerweise der Pfad zu der entsprechenden Konfigurationsdatei, also beispielsweise:

```
mktap unids.middlelayer -c /pfad/zu/der/konfigurationsdatei/ml_db.xconf
```

Die Datei `ml_db.xconf` wird in Abschnitt 4.4.4.5. Das Dateiformat wird im Abschnitt 4.4.4 erklärt.

[16] Die Informationen aus Abschnitt 4.4.4.4 gelten sinngemäß auch für die hier festlegbaren Logging-Level.

4.4.4.5 Konfigurationsdatei

In diesem Abschnitt sollen nur die wichtigsten Konfigurationsmöglichkeiten erklärt werden, um die Mittelschicht an die eigenen Bedürfnisse anpassen zu können. Deswegen wird nicht jeder einzelne Punkt erklärt. Für einen Überblick über das Konfigurationsformat der Datei sei auf Abschnitt 4.4.4 verwiesen.

- **logging**

```
<logging level="DEBUG">
  <logger name="beepy" level="INFO"/>
  <logger name="unids.beep.profiles.GuiProfile" level="DEBUG"/>
  <logger name="unids.middlelayer.stateimpl" level="DEBUG"/>
  <logger name="unids.middlelayer.statemgr" level="DEBUG"/>
  <logger name="unids.middlelayer.comp.webserver" level="DEBUG"/>
</logging>
```

  An dieser Stelle wird es dem Administrator ermöglicht, das logging für einzelne Komponenten genauer einzustellen. Das ist zum Beispiel nützlich, wenn genau bekannt ist welche Komponente Probleme bereitet, und es daher nur nötig ist, deren Ausgaben zu beobachten.
  Erlaubt sind die gleichen Werte wie im Abschnitt 4.4.4.4 schon erwähnt.

- **integrierter Webserver**

```
<component id="webserver">
  <class>
    <package>unids.middlelayer.comp.webserver</package>
    <name>Webserver</name>
  </class>
  <config>
    <bind iface="localhost" port="1079"/>
    <parameter name="guiroot" value="../../tmp/gui_www"/>
  </config>
</component>
```

  Die Mittelschicht bietet einen eigenen kleinen, integrierten Webserver an, der dazu benutzt werden kann, die `unIDS-GUI.jnlp` der GUI bereitzustellen.
  Über die Parameter hinter *iface* und *port* ist somit einstellbar, auf welchen Interface (beispielsweise: localhost, 127.0.0.1) und welchen Port der Webserver lauscht.
  Der Parameter *guiroot* beziehungsweise der Wert `value=` bestimmt, in welchen Verzeichnis der Webserver die `unIDS-GUI.jnlp` bedient.

- **statemgr**

```
<component id="statemgr">
 <class>
  <package>unids.middlelayer.statemgr</package>
  <name>StateManager</name>
```

```
 </class>
 <config>
  <!-- Use in memory storage -->
  <param name="db" value="file"/>
  <param name="db.file" value="dev_unids.middlelayer.fs"/>
 </config>
</component>
```

- **compiface**

```
<component id="compiface">
 <class>
  <package>unids.middlelayer.compiface</package>
  <name>ComponentInterface</name>
 </class>
 <config>
  <bind iface="127.0.0.1" port="2003"/>
 </config>
</component>
```

Der Abschnitt `compiface` beschreibt die Verbindung zu den Agenten, genauer gesagt auf welchen *iface* und *port* die Mittelschicht Verbindungen von den Agenten erwartet. Mit den Standarteinstellungen muss der Agent auf dem selben System laufen wie die Mittelschicht, da `127.0.0.1` bedeutet das nur lokal Verbindungen erlaubt werden.

- **guiiface**

```
<component id="guiiface">
 <class>
  <package>unids.middlelayer.guiiface</package>
  <name>GuiInterface</name>
 </class>
 <config>
  <bind iface="127.0.0.1" port="10287"/>
 </config>
</component>
```

Der Abschnitt `guiiface` beschreibt die Verbindung zu der GUI, genauer gesagt auf welchen *iface* und *port* die Mittelschicht den Verbindungsaufbau von der GUI erwartet. Mit den Standarteinstellungen muss auch die GUI auf dem selben System laufen wie die Mittelschicht, da `127.0.0.1` bedeutet das nur lokal Verbindungen erlaubt werden.

4.4.4.6 Alternative Implementierungen festlegen

Das Komponentenmodell der Mittelschicht ist so aufgebaut, dass über die Konfigurationsdatei festgelegt wird, welche konkrete Implementierung für eine bestimmte Schnittstelle verwendet werden soll. Innerhalb des Abschnitts `instrumentation`

können für einzelne Komponenten alternative Implementierungen festgelegt werden, die von der Standard-Implementierung abweichen.

Ursprünglicher Auslöser für diese Möglichkeit waren verschiedene Implementierungen für den Systemzustand, die an dieser Stelle austauschbar waren. Im gegenwärtigen Stand der Mittelschicht existiert für keine Schnittstelle mehr als eine Implementierung. Dieser Abschnitt bildet aber auch einen wichtigen Aspekt bei der Erweiterbarkeit der Mittelschicht. Über diesen Abschnitt können eigene Erweiterungen als „Ersatz“ für die Standard-Implementierung in die Mittelschicht eingebunden werden, wenn sie innerhalb der Registrierung dem System bekannt gemacht wurden.

4.4.4.7 Registrierung

Der Abschnitt `registry` bildet den wichtigsten Abschnitt der Mittelschicht. Zunächst werden im Unterabschnitt `roles` die dem System bekannten Rollen konfiguriert inklusive eines Verweises auf eine Standard-Implementierung. Eine Rolle in diesem Sinne ist dabei gleichbedeutend mit der Schnittstelle einer Komponente. Über diese Schnittstellen oder Rollen werden die Komponenten dann innerhalb der Mittelschicht angesprochen.

Die Definition einer Rolle enthält immer einen eindeutigen Identifikator für diese Rolle und die Angabe der Schnittstelle sowie einen Verweis auf eine Standardimplementierung.

Im zweiten Abschnitt der Registrierung werden die eigentlichen Komponenten definiert. Eine Komponente besteht dabei immer aus einem `class`-Tag, das auf die konkrete Implementierung verweist sowie aus einem Unterabschnitt `config`, über den spezielle Konfigurationseinstellungen für diese Komponente vorgenommen werden können.

4.5 Schnittstelle zu Agent und Proxy

Die Mittelschicht ist mit sämtlichen Agenten und Proxies direkt oder indirekt verbunden. In diesem Abschnitt wird beschrieben, wie innerhalb der Mittelschicht die Verbindung zu Agent und Proxies sowie die Agenten und Proxies selbst verwaltet werden. Dazu wird zunächst ein Überblick über die beteiligten Komponenten der Mittelschicht gegeben und danach detailliert auf jede einzelne Komponente eingegangen.

Die Schnittstelle zu Agent und Proxy wird innerhalb der Mittelschicht im Wesentlichen durch die Komponenten `ComponentInterface` und `ComponentManager` realisiert. Dabei ist das `ComponentInterface` für die konkreten Verbindungen zu den verschiedenen unIDS-Komponenten verantwortlich, während der `ComponentManager` für das Verwalten der unIDS-Komponenten an sich zuständig ist.

Informationen über die Kommunikation zwischen Agent / Proxy und Mittelschicht finden sich im unIDS-weiten Teil dieser Dokumentation im Abschnitt 2.2.

4.5.1 ComponentInterface

Das `ComponentInterface` ist für die folgenden Anwendungsfälle verantwortlich:

- **Agent / Proxy meldet sich an** – Der Agent / Proxy muss durch das `ComponentInterface` beim `ComponentManager` registriert werden.
- **Agent / Proxy senden Nachrichten** – Sofern der Absender der Nachricht dem System bekannt ist, wird die Nachricht weiter verarbeitet. Dies bedeutet in der Regel, dass sie an den `MessageDispatcher` weitergeleitet wird.
- **Die Verbindung bricht ab** – Sobald ein Zusammenbruch der Verbindung bemerkt wird, muss das `ComponentInterface` die unIDS-Komponente beim `ComponentManager` abmelden.
- **Nachrichten an Agent / Proxy senden** – Falls Nachrichten an einen Agenten oder Proxy gesandt werden sollen, ist das `ComponentInterface` für die Übertragung der Nachricht verantwortlich.

Bei vielen Aufgaben arbeitet das `ComponentInterface` eng mit dem `ComponentManager` zusammen[17].

4.5.2 ComponentManager

Der `ComponentManager` verwaltet die verschiedenen unIDS-Komponenten, die über das ganze Netzwerk verteilt sein können. Eine der wichtigsten Aufgaben ist die Sicherstellung der eindeutigen Identifizierbarkeit jeder unIDS-Komponente. Der `ComponentManager` stellt folgende Funktionen bereit:

- **Session-Identifikator erzeugen** – Jede pysikalische BEEP-Verbindung zur Mittelschicht ist eine Session. Der `ComponentManager` ordnet jeder Session einen eindeutigen Idenfikator zu, über den die Session erreichbar ist. Über diesen Identifikator wird der Session dann ein Kontextobjekt zugeordnet, welches eine Liste mit den über diese Session erreichbaren unIDS-Komponenten enthält.
- **Session-Identifikator löschen** – Wird eine Verbindung beendet, so muss auch der zugehörige Session-Identifikator gelöscht werden.
- **Anmeldung verarbeiten** – unIDS-Komponenten, die sich erstmalig oder aber mit einem schon vorhandenen Identifikator an der Mittelschicht anmelden, werden durch den `ComponentManager` als verbunden in den System-Zustand eingetragen. Analog hierzu muss das Abmelden verarbeitet werden.
- **Zugriff auf Kontexte** – Nach der Verarbeitung der An- und Abmeldevorgänge stellt der `ComponentManager` den anderen Komponenten der Mittelschicht eine Möglichkeit bereit, die Kontextobjekte der angemeldeten unIDS-Komponenten zuzugreifen. Über diese Kontextobjekte können andere Komponenten dann zum Beispiel Nachrichten an die unIDS-Komponenten senden.

Der `ComponentManager` ist also insbesondere für die Verwaltung des Teils des Zustands der Mittelschicht verantwortlich, der Informationen über die vorhandenen Proxies und Agenten[18] enthält.

[17] Siehe auch Abbildung 4.1 für einen Überblick über die Architektur der Mittelschicht.

[18] Proxies sind im Datenmodell zwar vorgesehen, wurden aber im Rahmen des Projektes nur als transparente Proxies realisiert, so dass sie bisher nicht im Systemzustand auftauchen können.

4.6 Schnittstelle zur grafischen Benutzeroberfläche

Die Schnittstelle zur grafischen Benutzerschnittstelle ist von den Komponenten her ähnlich aufgebaut, wie die Schnittstelle zu Agent und Proxy. Allerdings sind die Komponenten selbst anders realisiert, da zwischen der grafischen Benutzerschnittstelle und der Mittelschicht wesentlich mehr verschiedene Nachrichtentypen ausgetauscht werden als zwischen der Mittelschicht und dem Agent / Proxy.

Zunächst steht die Komponente `GuiInterface` für die Verwaltung der Verbindung zur grafischen Benutzerschnittstelle zur Verfügung. Diese nutzt den `GuiManager` zur Verwaltung der angemeldeten Benutzer und zur Regelung des Zugriffs.

Zunächst wird in diesem Abschnitt das Zusammenwirken der Komponenten `GuiInterface` und `GuiManager` beschrieben. Daran anschließend folgt eine detailliertere Beschreibung der inneren Struktur des `GuiInterface`.

4.6.1 Zusammenspiel von GuiInterface und GuiManager

Bei einer Anmeldung der grafischen Benutzerschnittstelle durch den Benutzer, wird vom `GuiInterface` über den `GuiManager` eine spezielle Sicht auf den Systemzustand der Mittelschicht angefordert. Diese Sicht ist schon personalisiert, das heißt, sie hat eine Referenz auf das ihr zugeordnete Benutzerobjekt. Sämtliche Zugriffe der grafischen Benutzerschnittstelle auf den Systemzustand der Mittelschicht erfolgen über diese personalisierte Sicht. An diesem Punkt würde auch die Zugriffssteuerung entsprechend der Rechte des aktuellen Benutzers aktiv werden[19].

Der `GuiManager` ist dabei ausschließlich für die Authorisierung des Benutzers verantwortlich. Hierzu stellt er die Eigenschaft *portal* bereit, über die ein Benutzer am System angemeldet werden kann. Das Ergebnis einer solchen Anmeldung ist eine Objekt, dass die Schnittstelle *IAvatar*[20] bereitstellt und darüber den Zugriff das in Abschnitt 4.8 beschriebene Datenmodell ermöglicht. Nach erfolgreicher Anmeldung eines Benutzers ist der `GuiManager` bis zum Ende der Sitzung des Benutzers nicht mehr aktiv.

Direkt nach der erfolgreichen Anmeldung übernimmt das `GuiInterface` alle weiteren Aufgaben, die sich im Wesentlichen auf die korrekte Verarbeitung der eingehenden Nachrichten beschränken. Die Nachrichtenverarbeitung ist schematisch in Abbildung 4.5 dargestellt. Dabei werden Nachrichten als XML-Zeichenketten von der grafischen Benutzerschnittstelle (Gui) über das Netzwerk an die Mittelschicht übertragen. Dort werden sie innerhalb der Komponente `GuiInterface` weiterverarbeitet. In der Abbildung ist die Anfrage aller Benutzer sinngemäß dargestellt. Nach der Umwandlung der Nachricht in ein passendes Nachrichten-Objekt wird die Anfrage über passende Zugriffe auf die Geschäftslogik[21] ausgeführt. Das Ergebnis wird dann wieder in eine XML-Zeichenkette verpackt und an die grafische Benutzerschnittstelle zurückgesandt.

[19] Diese Funktionalität konnte mit den vorhandenen Entwicklerkapazitäten leider nicht vollständig realisiert werden, siehe hierzu auch Abschnitt 4.13.1.

[20] Diese Schnittstelle ist im Paket `unids.middlelayer.statemodel` zu finden.

[21] In der Abbildung 4.5 steht *IAvatar* als Platzhalter für das gesamte Datenmodell. Auf die Darstellung der einzelnen Objekte wurde hier im Sinne der Übersichtlichkeit verzichtet.

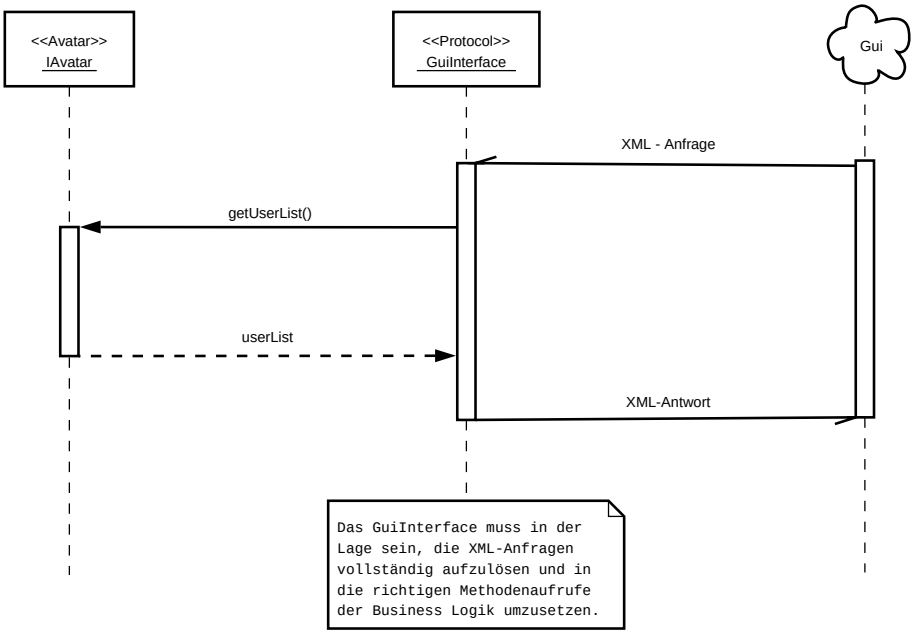

Abbildung 4.5: Überblick: Schema der Anfrageverarbeitung im GuiInterface

4.6.2 GuiInterface

Das `GuiInterface` wickelt die Kommunikation mit der grafischen Benutzerschnittstelle ab. Dabei gilt es die eintreffenden XML-Zeichenketten in die richtigen Nachrichten-Objekte zu verpacken und danach zu verarbeiten. In Abbildung 4.6 ist eine Übersicht über den Aufbau des `GuiInterface` dargestellt[22].

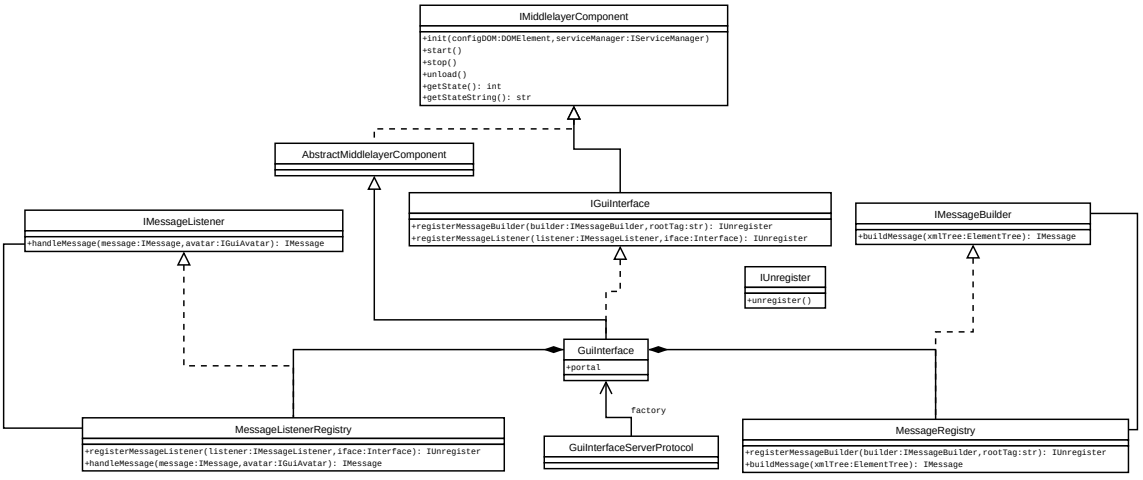

Abbildung 4.6: Innerer Aufbau der Komponente GuiInterface

Für die Verarbeitung der Nachrichten sind im `GuiInterface` zwei Bereiche für die Nachrichtenverarbeitung vorgesehen. Zunächst müssen die XML-Zeichenketten in die richtigen Nachrichten-Objekte umgewandelt werden. Hierz dient die `MessageRegistry`. Sie verwaltet eine Liste von Implementierungen der Schnittstelle *IMessageBuilder*, die in der Lage sind, bestimmte XML-Zeichenketten zu Verarbeiten und in die entsprechenden Nachrichten-Objekte umzuwandeln. Daneben ist die `Message-`

[22]Einige Methoden und Attribute sind der Übersichtlichkeit halber hier nicht dargestellt.

`ListenerRegistry` vorhanden, die eine Menge von Implementierungen der Schnittstelle *IMessageListener* verwaltet, die wiederum für bestimmte Nachrichtentypen registriert wurden und in der Lage sind genau diese Nachrichtentypen weiterzuverarbeiten.

Die Schnittstelle *IGuiInterface* beschreibt, wie die anderen Komponenten der Mittelschicht mit dem `GuiInterface` kommunizieren können. Die beiden Methoden dienen dazu, weitere Nachrichtentypen im `GuiInterface` zu registrieren[23].

4.7 Nachrichtenversand und -verarbeitung

Die Mittelschicht wird von den anderen unIDS-Komponenten über XML-Nachrichten angesteuert. Die Verarbeitung der Nachrichten auf der Agent / Proxy Seite erfolgt noch nicht nach dem gleichen Schema, wie auf der Seite der grafischen Benutzerschnittstelle[24].

Es ist grundsätzlich zwischen dem Nachrichtenversand und der Nachrichtenverarbeitung zu unterscheiden. Für den Nachrichtenversand ist die Komponente `MessageDispatcher` zuständig, bei der alle unIDS-Kmponenten, also Agent, Proxy und grafische Benutzerschnittstelle, registriert werden sobald sie eine Verbindung zu Mittelschicht aufbauen. Für den Nachrichtenversand stellt der `MessageDispatcher` eine Methode zur direkten Adressierung einer bestimmten unIDS-Komponente über ihren Identifikator und eine Methode zum Versand einer Nahricht an alle unIDS-Komponenten eines bestimmten Typs bereit.

Wie Eingangs beschrieben ist die Nachrichtenverarbeitung noch nicht vereinheitlicht. Auf der Seite der grafischen Benutzerschnittstelle ist die Verarbeitung der Nachrichten nach einem weiterentwickelten Konzept in der Komponente `GuiInterface` realisiert, wärend die gesamten Nachrichten von Agenten und Proxies noch direkt im `MessageDispatcher` verarbeitet werden[25].

Die Nachrichtenverarbeitung im `GuiInterface` ist in der Abbildung 4.7 in einer an ein Sequenzdiagramm angelehnten Darstellung verdeutlicht. Der erste Vorgang ist immer der Speziallfall „Benutzeranmeldung“, bei dem die aktuelle Sitzung initialisiert wird. Eine normale Nachricht von der grafischen Benutzerschnittstelle durchläuft dabei die folgenden Stationen:

- **MessageBuilderRegistry** – An dieser Stelle wird der von der grafischen Benutzerschnittstelle empfangene XML-Datenstrom in ein Nachrichtenobjekt umgewandelt. Dazu wurden in der `MessageBuilderRegistry` zuvor einige Instanzen registriert, die die Schnittstelle *IMessageBuilder* bereitstellen. Bei einer neu eingetroffenen Nachricht probiert die `MessageBuilderRegistry` nacheinander alle in Frage kommenden *IMessageBuilder* durch, bis einer in der Lage war die XML-Zeichenkette zu verarbeiten. Danach steht ein Nachrichten-Objekt zur Verfügung.

[23] Weitere Informationen hierzu finden sich im Abschnitt 4.12.3 über die Erweiterungsmöglichkeiten der Mittelschicht.

[24] Siehe hierzu Abschnitt 4.13.2 zu Ticket Nr. 224.

[25] Es war geplant, die Nachrichtenverarbeitung auch von den Agenten und Proxies entsprechend der Nachrichtenverarbeitung im `GuiInterface` zu überarbeiten. Dies konnte allerdings für die Version 1.0 des unIDS-Systems mit den vorhandenen Kapazitäten nicht mehr fertiggestellt werden.

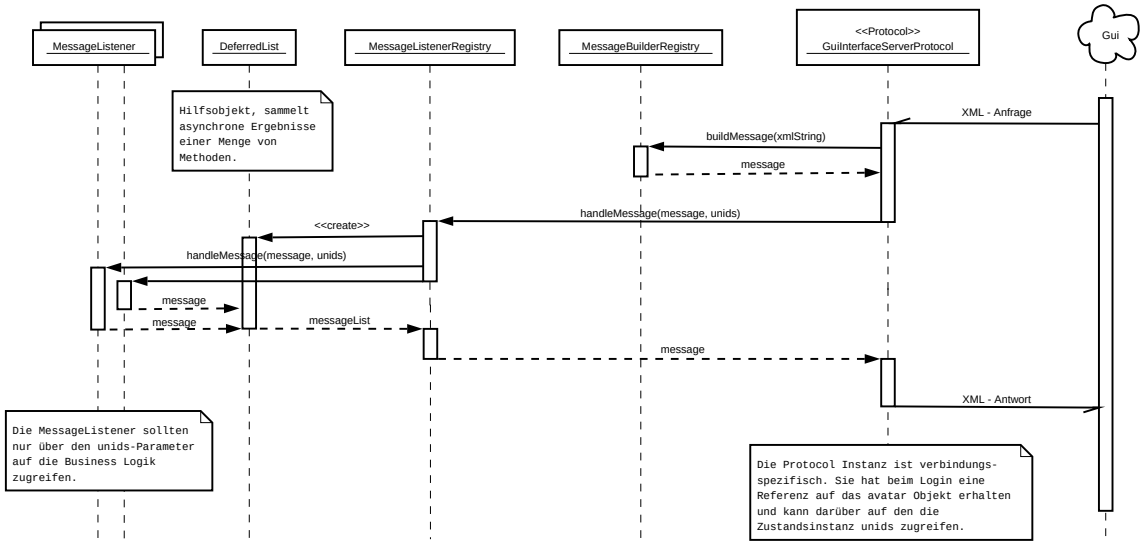

Abbildung 4.7: Anfrageverarbeitung im GuiInterface

- **MessageListenerRegistry** – Die zuvor erzeugte Nachricht wird dann in der `MessageListenerRegistry` an alle „interessierten“ *IMessageListener*-Implementierungen weitergeleitet. Hierzu sind bei der Initialisierung die *IMessageListener*-Implementierungen in der `MessageListenerRegistry` für einen oder auch mehrere Nachrichtentypen registriert worden.

- **IMessageListener** – Für jedes eintreffende Nachrichten-Objekt werden nun für alle von der Nachricht bereitgestellen Schnittstellen[26] die registrierten *IMessageListener* aktiviert. Sie erhalten hierzu die Nachricht sowie eine durch Transaktionen abgesicherte Sicht auf das Datenmodell. Nach der Verarbeitung steht für jeden *IMessageListener* eine Liste von Antworten zur Verfügung. Im Normalfall hat diese Liste die Länge 1 und die Ergebnis-Nachricht kann direkt an die grafische Benutzerschnittstelle zurückgeliefert werden. Da prinzipiell aber auch mehrere *IMessageListener* für einen Nachrichtentyp vorhanden sein können, werden falls mehrere Antworten auf eine Anfrage vorliegen, diese in einer Liste zusammengefasst und als Nachrichten-Liste an die grafische Benutzerschnittstelle zurückgeliefert.

4.8 Datenmodell

Für das Datenmodell wurde zunächst eine spezielle Implementierung in den Entitäten verwendet, um die unter den Entitäten vorhandenen Strukturen abzubilden. Dies führte innerhalb der Entwicklung und damit auch im Laufe der nötigen Anpassungen und Erweiterungen zu sehr komplexen Datentypen. In Abbildung 4.8 ist das Modell stark verkleinert dargestellt. Um das Datenmodell wieder flexibler zu gestalten, wurde es konzeptionell umgestellt, so dass die Datentypen jetzt sehr einfach gehalten sind und keine beziehungsweise nur noch einfache Informationen über ihre Strukturierung enthalten. Der weitestgehend hierarchische Aufbau des unIDS-Systems selbst, aber auch der Aufbau der Struktur des überwachten Netzwerkes wurden numehr über gerichtete Graphen realisiert, da dieses Modell auch für zukünftige Erweiterungen am flexibelsten erschien.

[26] Diese entsprechen den Nachrichtentypen.

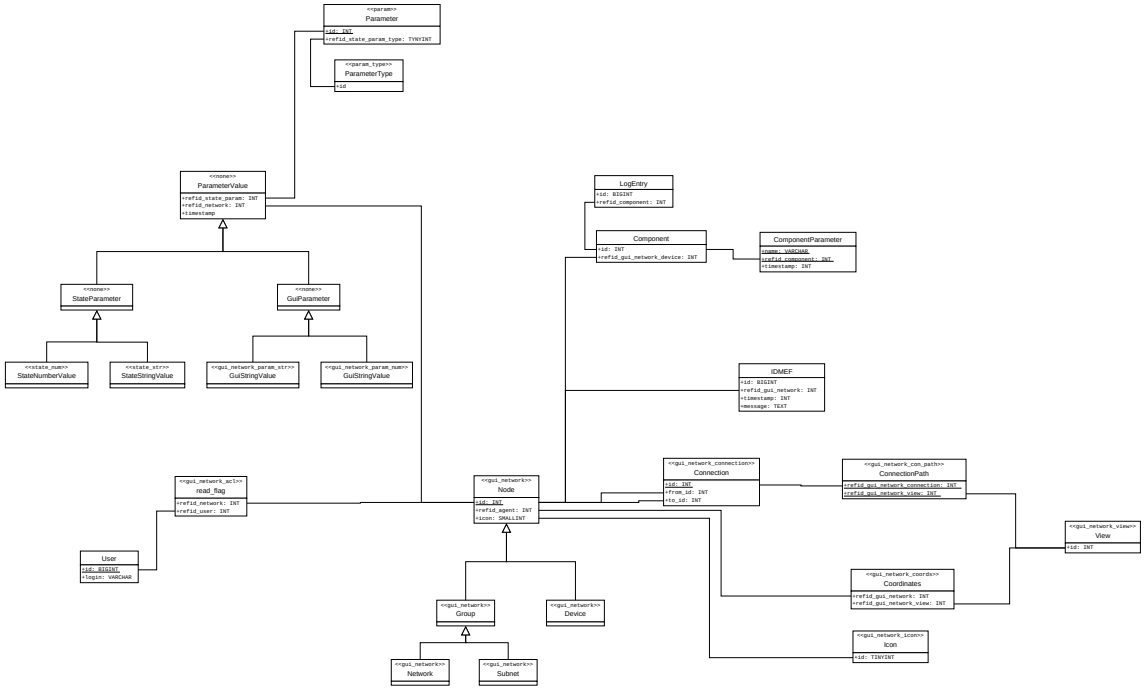

Abbildung 4.8: Datenmodell mit komplexen Abhängigkeiten zwischen den Entitäten.

Im Folgenden werden zunächst grundlegende Bestandteile des Datenmodells wie die Identifikatoren und die verwaltenden Teile des Datenmodells beschrieben. Im Anschluss daran werden die wesentlichen Entitäten vorgestellt. Die Schnittstellen aller Teile des Datenmodells sind im Paket `unids.middlelayer.statemodel` definiert. Ihre Implementierungen sind im Paket `unids.middlelayer.stateimpl` zusammengefasst. Innerhalb dieses Abschnitts liegt der Fokus auf den Schnittstellen.

4.8.1 Hierarchische Identifikatoren

Jede im unIDS-System vorhandene Entität erhält einen systemweit eindeutigen Identifikator. Sämtliche systemweit eindeutigen Identifikatoren werden durch die Mittelschicht vergeben. Konzeptionell sind die Identifikatoren dabei hierarchisch angelegt, da die einzelnen Entitäten wie später im Abschnitt 4.8.3 beschrieben durch verschiedene Kataloge verwaltet werden, erfolgt auch die Vergabe der Identifikatoren an verschiedenen Stellen. Die Identifikatoren selbst sind dabei hierarchisch aufgebaute Zeichenketten, die mit den Pfad eines Dateisystems verglichen werden können. Die später in Abschnitt 4.8.2 beschriebene Schnittstelle bildet dabei die Wurzel für den Zugriff auf das Datenmodell. Jeder Zugriff über einen Identifikator wird von dieser Stelle aus durch das Datenmodell so lange weitergeleitet, bis letztlich die gesuchte Entität zurückgeliefert wird oder aber bis feststeht, dass es zu dem Identifikator keine Entität im unIDS-Datenmodell gibt.

Die einzelnen hierarischen Stufen sind dabei „sprechend“ gewählt, so dass mitgeschnittene Auszüge der Kommunikation zwischen den unIDS-Systemkomponenten für die an der Enticklung beteiligten Personen leichter und damit schneller verständlich sind. Das folgende Beispiel zeigt, wie auf die Entität mit dem Identifikator `device/2` zugegriffen werden kann:

```
ident = "device/2"
```

```
# Abfrage, ob eine Entität zu dem Identifikator vorhanden ist
if ident in unids:
    # Zugriff auf die Entität
    device = unids[ident]
else:
    print "Device 'device/2' nicht vorhanden!"
```

Maßgebend für die Implementierungsentscheidungen an dieser Stelle waren, den Zugriff und die Abfrage des Datenmodells möglichst einfach und übersichtlich zu gestalten und dabei auf die in der Python-Welt gebräuchlichen Konzepte zurückzugreifen.

4.8.2 Basis IUnids

Die Basis für den persistenten Teil des Zustands bildet die Schnittstelle *IUnids*, die in Abbildung 4.9 dargestellt ist. Über die Komponente `StateManager` kann von anderen Komponenten aus auf den Systemzustand zugegriffen werden. Hierzu ist die Methode `getUnids` vorgesehen.

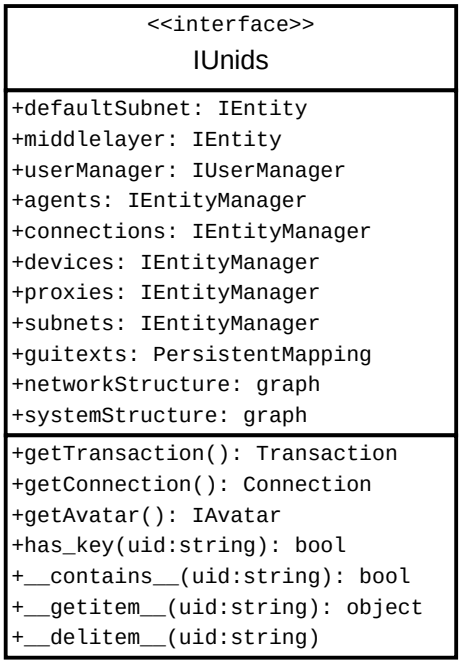

Abbildung 4.9: Die Schnittstelle IUnids.

Die Schnittstelle *IUnids* gliedert sich in drei wesentliche Bereiche.

4.8.2.1 Attribute

Zunächst schreibt die Schnittstelle *IUnids* einige Attribute vor, über die ein Zugriff auf weitere Entitäten des Systemzustands möglich ist. Auf Getter- und Setter-Methoden wurde an dieser Stelle bewusst verzichtet, da Python die Möglichkeit bietet solche Methoden auch nachträglich transparent beim Attributzugriff zwischenzuschalten.

Die ersten beiden Attribute *defaultSubnet* und *middlelayer* sind zwei ausgezeichnete Entitäten, die immer im Systemzustand vorhanden sind. Die Entität *middlelayer*

dient dabei als Referenz für die Wurzel des Zustands. Alle weiteren Entitäten sind innerhalb der verschiedenen Strukturen „unterhalb“ dieser Entität angeordnet. Daneben ist das Attribut *defaultSubnet* ein spezielles Subnetz , in den neue Entitäten standardmäßig angelegt werden oder aber auch verlorene[27] Entitäten auftauchen.

Das Attribut *userManager* enthält eine Referenz auf eine Implementierung der Schnittstelle *IUserManager*, die für die Verwaltung der im unIDS-System bekannten Benutzer verantwortlich ist. Siehe hierzu Abschnitt 4.8.4.

Die folgenden Attribute vom Typ *IEntityManager* sind entsprechend ihrer Bezeichnung spezielle Container für die Verwaltung bestimmter Entitäten. Die Schnittstelle *IEntityManager* ist in Abschnitt 4.8.3 beschrieben.

Die Eigenschaft *guitexts* dient einer speziellen Aufgabe. Die grafische Benutzerschnittstelle speichert verschiedene Statusinformationen wie zum Beispiel Fenstergrößen oder Ähnliches in Zeichenketten, die unter einem bestimmten Schlüssel im Attribut *guitexts* abgelegt werden. Der Datentyp `PersistentMapping` verhält sich dabei genauso wie der Typ `dict`, der bei Python standardmäßig vorhanden ist. Daneben stellt er bestimmte Eigenschaften bereit, die besser für die Verwendung mit der Objektdatenbank ZODB geeignet sind.

4.8.2.2 Strukturen

Die Attribute *networkStructure* und *systemStructure* bilden besondere Eigenschaften. Sie enthalten jeweils einen gerichteten Graph, der über die vorhandenen Devices und unIDS-Systemkomponenten eine Struktur legt. Über den Ereignismechanismus der später beschriebenen *IEntityManager*-Schnittstelle wird dafür gesorgt, dass diese Strukturen immer dem aktuellen Zustand des Datenmodells entsprechen.

Im Attribut *networkStructure* ist die Struktur der Subnetze und Devices untergebracht. Diese Struktur kann durch die grafische Benutzerschnittstelle durch entsprechende Nachrichten an die Mittelschicht modifiziert werden. Dadurch, dass im Hintergrund ein gerichteter Graph verwendet wird, lassen sich aus Sicht des Datenmodells auch sehr komplexe Strukturen definieren, die dann in der grafischen Benutzerschnittstelle dargestellt werden.

Im Attribut *systemStructure* ist die Struktur des unIDS-Systems enthalten. Über diese Struktur hat die Mittelschicht immer ein aktuelles Bild des gesamten unIDS-Systems. Hieraus lassen sich auch Informationen darüber gewinnen, über welche Wege bestimmte Komponenten des unIDS-Systems erreichbar sind. Bei der Version 1.0 des unIDS-Systems ist die Proxy als ein auch für das unIDS-System selbst transparenter Proxy realisiert und taucht daher in dieser Struktur nicht auf. Die Systemstruktur ist ebenfalls über die grafische Benutzerschnittstelle sichtbar, aber nur sehr eingeschränkt veränderbar, da sich viele Eigenschaften dieser Struktur daraus ergeben, wie die einzelnen unIDS-Komponenten letztlich mit der Mittelschicht verbunden sind.

4.8.2.3 Zugriff auf den Kontext

Der Systemzustand ist bei Verwendung aus Sicht der grafischen Benutzerschnittstelle immer in einem Kontext zu sehen. Über die Methode `getAvatar` kann auf diesen

[27] Als „verloren“ gelten Entitäten dann, wenn sie ausgehend von der Wurzel *middlelayer* über keinen Weg im Struktur-Graphen erreichbar sind.

personalisierten Kontext zugegriffen werden. Dies kann dann notwendig sein, wenn auf den aktuell angemeldeten Benutzer oder aber den Laufzeitzustand zugegriffen werden soll, da nicht immer eine Referenz auf die Komponente `ServiceManager` zur Verfügung steht[28].

4.8.3 IEntityManager

Die Schnittstelle *IEntityManager* beschreibt einen Katalog, mit dem Entitäten[29] verwaltet werden können. Zunächst werden die Anforderungen an diesen Katalog beschrieben. Daran anschließend folgt die Beschreibung der Schnittstelle sowie der Implementierung.

4.8.3.1 Anforderungen und Aufgaben

Dieser Katalog hat verschiedene Aufgaben. Zunächst ist er eine Art Factory für die durch ihn verwaltete Klasse von Entitäten. Mit seiner Hilfe erstellte Entitäten müssen durch diesen Katalog mit einem eindeutigen Identifikator versehen und an der Persistenzschicht angemeldet werden. Diese Vorgänge sollen aus Sicht des Benutzers dieser Schnittstelle möglichst transparent sein.

Neben dem Erstellen von Entitäten sind an den Katalog aber auch Anforderungen zur Verwaltung dieser Entitäten gestellt. Es muss zunächst also möglich sein, Entitäten auch wieder aus dem Katalog zu entfernen oder auf Entitäten über ihren Identifikator zuzugreifen. Da wie in der Einleitung zu Abschnitt 4.8 bereits angedeutet, die Beziehungen zwischen den Entitäen weitestgehend aus den Entitäten selbst ausgelagert wurden, um sie nicht zu komplex werden zu lassen, muss der Katalog auch in der Lage sein, bestimmte Ereignisse an registrierte Stellen weiterzuleiten.

4.8.3.2 Schnittstelle und Implementierung

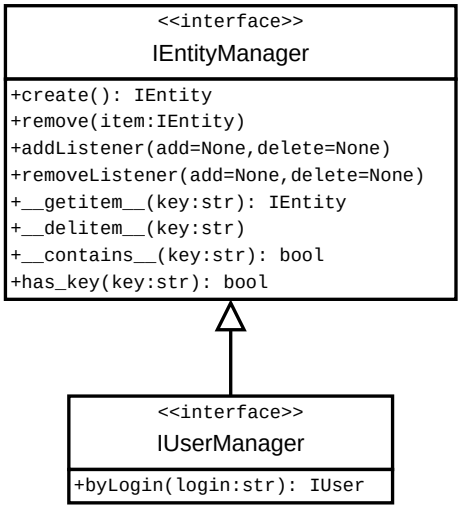

Abbildung 4.10: Schnittstellen IEntityManager und IUserManager.

[28] Dies ist insbesondere bei der Nachrichtenverarbeitung der Fall, da geplant war, die Zugriffe auf den Systemzustand vorher zu authorisieren. Dazu waren zunächst die Möglichkeiten des direkten Zugriffs auf den Systemzustand zu entfernen. Siehe hierzu auch in der Liste der Offenen Features auf Seite 76 Ticket Nr. 98.

[29] In diesem Zusammenhang ist eine Enität ein Objekt, dass die Schnittstelle *IEntity* bereitstellt.

Zur Erfüllung der Anforderungen wurde die Schnittstelle *IEntityManager* wie in Abbildung 4.10 entwickelt. Da der Katalog letztlich eine Menge von Entitäten über ihre Identifikatoren zuordnet stellt die Schnittstelle letztlich einen Ausschnitt der Schnittstelle des Python-Datentypen `dict` dar. Der Katalog wird bei seiner Intanzierung mit einer bestimmten Entitäts-Klasse initialisiert und kann danach auch nur zur Verwaltung von Entitäten dieser Klasse verwendet werden.

Neue Entitäten können dem Katalog ausschließlich über die Methode `create` hinzugefügt werden. Somit ist es insbesondere nicht möglich, an anderer Stelle erzeugte Instanzen der Entitäts-Klasse des Katalogs in diesen einzufügen. Dies liegt darin begründet, dass der Katalog für seine Klasse von Entitäten den Persistenz-Rahmen vorgibt und alleinig das Recht hat Identifikatoren zu vergeben, die innerhalb des gesamten unIDS-Systems eindeutig sind, also auch innerhalb der grafischen Benutzeroberfläche oder den Agenten und Proxies.

Um eine Entität wieder aus dem Katalog zu entfernen sind zwei leicht unterschiedliche Möglichkeiten vorgesehen. Zunächst lässt sich die Entität wie aus einer Liste mit der Methode `remove` entfernen. Hierzu muss die Entität selbst als Parameter übergeben werden. Die Alternative ist die Methode `__delitem__`, die aus der Schnittstelle für den Datentyp `dict` übernommen ist. Dieser Methode muss der passende Identifikator übergeben werden. Die leicht unterschiedliche Bedeutung ergibt sich aus dem zu übergebenden Parameter, einmal muss es die Entität selbst sein und an der anderen Stelle reicht der passende Identifikator aus. Das folgende Beispiel verdeutlich dies anhand eines Device-Eintrages mit dem Identifikator `device/8`.

```
ident = "8"

# A: __delitem__(ident)
del katalog[ident]

# B: remove
device = katalog.create()
katalog.remove(device)
```

An dieser Stelle wurde eine leichte Redundanz innerhalb der Schnittstelle in kauf genommen, da beide Methoden letztlich unterschiedliche Sichweisen auf den Katalog repräsentieren, die an verschiedenen Stellen innerhalb der Mittelschicht beider etwa gleichberechtigt vorkommen. Die Implementierung selbst stützt sich in beiden Fällen auf die selbe Methode um unterschiedliches Verhalten an dieser Stelle von vornherein auszuschließen.

Über die Methode `__getitem__` kann über den Identifikator auf die durch ihn bestimmte Entität analog zum Python-Datentyp `dict` zugegriffen werden. Ebenso wie beim Entfernen stellen die Methoden `__contains__` und `has_key` durch zwei verschiedene Ausdrücke die gleiche Funktionalität bereit, um den unterschiedlichen Sichtweisen auf den Katalog gerecht zu werden. Hier wurde allerdings von der Listen-Semantik insofern abgewichen, als dass bei beiden Methoden der Identifikator und nicht die Entität zu übergeben ist.

Über die Methoden `addListener` und `removeListener` bietet der Katalog die Möglichkeit, bestimmte Ergebnisse an andere Stellen weiterzuleiten. Dieser an das Observer-Pattern angelehnte Mechanismus wird innerhalb der Mittelschicht für die Verwaltung der Struktur über den Entitäten und die Einhaltung bestimmter Konsistenzregeln verwendet. Die in den Parametern zu übergebenden Objekte *add* und *delete* sind Funktionsobjekte die als einzigen Parameter die betroffene Entität übergeben bekommen.

4.8.4 IUserManager

Die Schnittstelle *IUserManager* ist wie in Abbildung 4.10 eine Spezialisierung der Schnittstelle *IEntityManager*, die speziell für die Verwaltung von Benutzern den Zugriff über den Benutzernamen ermöglicht. Eine Implementierung dieser Schnittstelle wird in er Mittelschicht für die Verwaltung der im unIDS-System bekannten Benutzer verwendet.

Benutzer spielen im unIDS-Datenmodell nur eine sehr untergeordnete Rolle. Sie sind nötig, um die Zugriffe über die grafische Benutzerschnittstelle einer Identität zuordnen zu können. Darüber hinaus sind sie darauf ausgelegt, als Basis für die Zugriffsberechtigungen und -beschränkungen innerhalb des unIDS-Systems zu dienen und Zugriffe durch die zugehörigen Rollen zu authorisieren. In der Version 1.0 des unIDS Systems ist das Benutzerkonzept allerdings nur in einer stark vereinfachten Version enthalten[30].

4.8.5 IEntity, die Basis für alle Entitäten

Alle Entitäten innerhalb des unIDS-Datenmodells müssen die Schnittstelle *IEntity* bereitstellen. Sie enthält dabei nur die zur Verwaltung von Entitäten notwendigen Attribute *id* und *uid*.

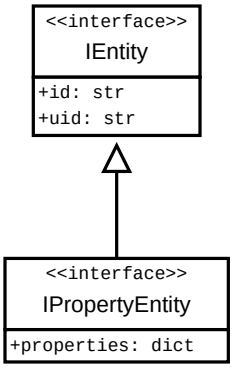

Abbildung 4.11: Schnittstellen IEntity und IPropertyEntity.

Dabei sind die Attribute *id* und *uid* meistens miteinander verbunden. Das Attribut *id* wird in der Regel durch den Katalog, der die Entität verwaltet, gesetzt und danach nicht mehr verändert. Der Wert von *id* ist muss dabei nicht geeignet sein, um die Entität systemweit identifizieren zu können. Diese Anforderung ist nur an den Wert des Attributes *uid* gestellt. Das Attribut *uid* ist nur lesbar und kann daher auch nur indirekt durch das Attribut *id* beeinflusst werden.

[30] Siehe hierzu auch das Ticket Nr. 98 auf Seite 76.

Die ebenfalls in Abbildung 4.11 dargestellte Schnittstelle *IPropertyEntity* erweitert *IEntity* um die Eigenschaft *properties*, in der verschiedene Eigenschaften der Entität abgelegt werden können. Dieses Attribut muss die Schnittstelle des Python-Datentyps `dict` bereitstellen und wird in der Regel für die Speicherung von Schlüssel-Wert-Paaren verwendet, wobei sowohl Schlüssel als auch der Wert Zeichenketten sein müssen. Die meisten der im Folenden vorgestellten Entitäten stellen die Schnittstelle *IPropertyEntity* bereit, da sie vom Datenmodell her eine unbestimmte Anzahl an Eigenschaften haben können. Diese Eigenschaft ist für die meisten Entitäten besonders wichtig, da erst mit dem Fortschreiten der Entwicklung des unIDS-Systems weitere Anforderungen an die unterschiedlichen Entitäten erkennbar wurden.

4.8.6 IComponent

Die Schnittstelle *IComponent* repräsentiert eine Komponente des unIDS-Systems im Datenmodell der Mittelschicht. Dabei wird eine Komponente als eine normale Entität mit Eigenschaften repräsentiert, die zusätzlich das Attribut *config* haben muss. Innerhalb dieses Attributs wird die Konfiguration für diese Komponente verwaltet. Immer wenn sich eine unIDS-Systemkomponente wie zum Beispiel ein Agent oder ein Proxy an der Mittelschicht anmelden, liefern sie ihre Standard-Konfiguration. Diese wird bei der ersten Anmeldung von der Mittelschicht ohne Änderungen übernommen. Bei späteren Anmeldungen wird nur noch geprüft, ob neue Parameter hinzugekommen sind. Auf der anderen Seite hat die grafische Benutzerschnittstelle Zugriff auf den Wert des Attributes *config* und kann Änderungen vornehmen. Bei Änderungen ist die Mittelschicht dafür Verantwortlich, die geängerten Teile der Konfiguration in den Wert des Attributes *config* zu übernehmen und falls die betroffene unIDS-Systemkomponente mit der Mittelschicht verbunden ist, diese über die Änderungen zu informieren.

4.8.7 IConnection

Für die Verwaltung von Verbindungen zwischen den einzelnen Devices im unIDS-Datenmodell ist die Schnittstelle *IConnection* vorgesehen. Wichtig im Vergleich zu den meisten anderen Entitäten ist, dass diese Schnittstelle direkt die Schnittstelle *IEntity* erweitert und somit keine allgemeinen Eigenschaften vorgesehen sind.

In Abbildung 4.12 ist die Schnittstelle *IConnection* grafisch dargestellt. Die Attribute *sourceUid* und *targetUid* geben die beiden Enden an, die über die Verbindung verbunden sind. Im Attribut *virtual* ist angegeben, ob es sich um eine virtuelle Verbindung handelt.

4.8.8 IUser

Die Schnittstelle *IUser* repräsentiert einen Benutzer im unIDS-Datenmodell. Diese Schnittstelle ist ebenfalls eine spezielle Entität, da sie nicht die Schnittstelle *IPropertyEntity* erweitert, sondern direkt von *IEntity* erbt. Die Attribute sind in Abbildung 4.13 dargestellt und werden im Folgenden erläutert.

In der Version 1.0 des unIDS-Systems ist bisher kein richtiges Rollenkonzept für die Benutzer vorgesehen. Bisher sind ausschließlich zwei Gruppen von Benutzern vorgesehen: Auf der einen Seite Administratoren, die uneingeschränkten Zugriff auf das System haben und auf der anderen Seite normale Benutzer, die die Struktur

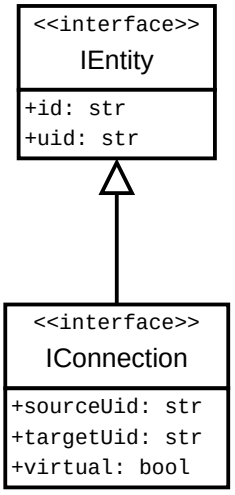

Abbildung 4.12: Schnittstelle IConnection

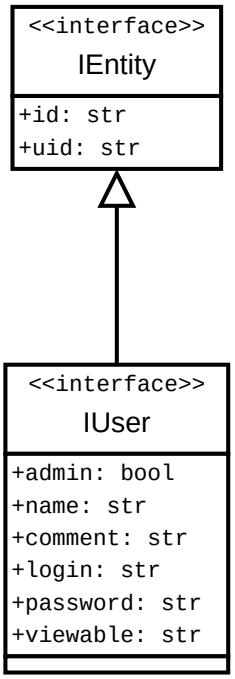

Abbildung 4.13: Schnittstelle IUser

des Datenmodells nicht bearbeiten können. Die Unterscheidung der beiden Gruppen findet anhand des Attributes *admin* statt.

Die Attribute *name*, *comment*, *login* und *password* speichern die entsprechenden Eigenschaften des Benutzers. Auch das Kennwort wird bisher im Klartext gespeichert[31].

Das Attribut *viewable* ist eine Zwischenlösung, die bisher komplett durch die grafische Benutzerschnittstelle verwaltet wird. In diesem Attribut wird bisher eine XML-Struktur abgelegt, die Informationen darüber enthält, welche Bereiche der Datenstruktur für den entsprechenden Benutzer sichtbar sein sollen. Vorgesehen war an dieser Stelle ein generelles Konzept, das die detaillierte Beschreibung von Zugriffsrechten auf alle Aspekte des Datenmodells ermöglicht. Da diese Eigenschaften aber nicht zur Kernfunktionalität des unIDS-Systems gehören, wurden sie mit eher niedriger Priorität verfolgt und konnten mit den vorhanden Resourcen für die Version 1.0 nicht umgesetzt werden.

4.9 Persistenz und Laufzeitzustand

Innerhalb dieses Abschnitts wird zunächst die Entwicklung der Entscheidung für die Verwendung der Objektdatenbank ZODB erläutert. Daran anschließend wird die für die Verwaltung des Systemzustands der Mittelschicht verantwortliche Komponente `StateManager` vorgestellt. Abschließend wird das entwickelte Datenmodell für die Mittelschicht erörtert.

4.9.1 Objektdatenbank ZODB

Die anfängliche Entwurfsentscheidung zur persistenten Datenhaltung bei der Verwendung eines RDBMS zu bleiben und damit den Ansatz aus der Vorgängerversion des unIDS-Systems zu übernehmen führte relativ schnell zu extremen Aufwänden. Zum einen bei der Entwicklung, durch sich permanent weiterentwickelnde Anforderungen und zum anderen bei den Datenbankzugriffen, denn immer mehr Strukturen zwischen den Entitäten mussten modelliert werden. Zunächst wurde versucht, mittels einer Formulierung des Datenmodells in einer XML-Datei und der Generierung eines passenden Schemas sowie der Datenklassen die Persistenz zu garantieren, allerdings nur mit erheblichem Aufwand und mäßigem Erfolg.

Um hier wieder Entwicklerressourcen freizusetzen und flexibler und vor allem kurzfristiger auf geänderte Anforderungen reagieren zu können, wurde die Entwurfsentscheidung zur Verwendung eines RDBMS schließlich zugunsten der Objektdatenbank ZODB revidiert. Innerhalb der Objektdatenbank ZODB können Python-Objekte direkt in einer sehr flexiblen Art und Weise gespeichert werden, die sich nur unwesentlich von der Ablage der entsprechenden Objekte im Arbeitsspeicher unterscheidet. Mit der Umstellung einhergehend wurde der Systemzustand der Mittelschicht in einen persistenten und einen nicht-persistenten Bereich geteilt.

[31] Hintergrund waren die Begrenzten Entwicklerkapazitäten und eine eher niedrige Priorität der Benutzerverwaltung an sich.

4.9.2 StateManager

Für die Verwaltung des Systemzustands der Mittelschicht ist die Komponente `StateManager` verantwortlich. Der `StateManager` ist die einzige Möglichkeit für andere Komponenten der Mittelschicht, auf den Systemzustand zuzugreifen. Der Systemzustand selbst ist dabei über eine Sammlung von Schnittstellen nutzbar, die in Abschnitt 4.8 beschrieben werden.

Der `StateManager` stellt den Zugriff auf den Systemzustand der Mittelschicht dabei in zwei Varianten zur Verfügung: Zunächst ist ein unauthorisierter Zugriff auf den Zustand möglich. Dies wird in der Regel nur von Komponenten der Mittelschicht verwendet, die nicht direkt über einen Benutzer bedient werden. In der anderen Variante stellt der `ServiceManager` dem `GuiInterface` einen personalisierten, authorisierten Zugriff für einen bestimmten Benutzeraccount zur Verfügung.

4.10 Nachrichtenanalyse

Vom Architekturmodell[32] der Mittelschicht her ist vorgesehen, dass eine oder auch mehrere eigenständige Komponenten für die Analyse der eintreffenden Nachrichten aber auch des gesamten unIDS-Systemzustands vorhanden sind. Im Rahmen der zur Verfügung stehenden Entwicklerkapazitäten konnte die Analyse nur sehr eingeschränkt realisiert werden. Zunächst werden die Nachrichten die von Agenten eintreffen je nach Nachrichtentyp den richtigen Devices zugeordnet und gegebenenfalls ohne Verzögerung an die grafische Benutzerschnittstelle weitergeleitet. Dies ist vor allem im Bereich der Statusinformationen der Fall[33].

Neben der Weiterleitung der Nachrichten, müssen diese zunächst den richtigen Device-Einträgen zugeordnet werden. Hierzu verwaltet die Mittelschicht eine strukturelle Informationssammlung darüber, welche Agenten zu welchen Devices gehören.

Für mittels IDMEF übertragene Nachrichten findet neben der Zuordnung und Weiterleitung zusätzlich eine Anpassung des Systemzustands statt. Im Datenmodell ist für Device-Eitnräge eine Status vorgesehen, der durch die grafische Benutzerschnittstelle für den Benutzer grafisch aufbereitet wird. Je nach Inhalt der Nachricht wird innerhalb der Mittelschicht für betroffene Device-Eitnräge der Status angepasst.

4.11 Weitere Komponenten der Mittelschicht

Weitere Komponenten der Mittelschicht sind der `Webserver` und der `FunctionPool`. Der `Webserver` sollte es ermögliche, die grafische Benutzerschnittstelle ohne weitere Arbeitsschritte direkt über die Mittelschicht abrufen zu können. Auf Grund der begrenzten Entwicklerkapazitäten ist in der Version 1.0 des unIDS-Systems eine rudimentärere Version des Webservers enthalten, die zwar voll funktionsfähig ist, aber noch nicht den „vollen Komfort“ bietet. Für die Bereitstellung der grafischen Benutzerschnittstelle über die Mittelschicht muss der Webserver zunächst in der Konfigurationsdatei der Mittelschich richtig konfiguriert werden[34]. Dann kann die grafische

[32] Siehe hierzu auch Abschnitt 4.4 und Abbildung 4.1.

[33] Die Weiterleitung dieser Nachrichten wurde so implementiert, dass nur die Statusinformationen weitergeleitet werden, die für den Benutzer gerade dargestellt werden.

[34] Siehe hierzu auch Abschnitt 4.4.4 über die Konfiguration der Mittelschicht.

Benutzerschnittstelle wie über jeden anderen Webserver direkt durch die Mittelschicht bereitgestellt werden. Hierdurch wird die Installation zusätzlicher Software und zusätzlicher Konfigurationsaufwand vermieden.

Die `FunctionPool` bietet eine einfache Möglichkeit, die Mittelschicht um kleinere Funktionen zu erweitern. Ein Beispiel, wie weitere Funktionen erstellt werden können ist in Abschnitt 4.12.2 beschrieben. Die im `FunctionPool` konfigurierten Funktionen können über die grafische Benutzerschnittstelle abgefragt und aufgerufen werden.

4.12 Erweiterbarkeit

Eine wesentliche Grundanforderung an das gesamte unIDS-System ist die Erweiterbarkeit. Für die Mittelschicht bedeutet dies, dass neue Funktionalität möglichst einfach hinzugefügt und vorhandene Funktionalität möglichst einfach angepasst werden können. Die folgenden Bereiche der Mittelschicht sind als möglichst leicht erweiterbar entworfen worden:

- **Komponenten** – Die Architektur der Mittelschicht ist stark komponentenbasiert. Weitere Komponenten sollen möglichst einfach und vor allem ohne Änderung der bereits bestehenden Komponenten in das System integriert werden können. Daneben soll es auch möglich sein, bereits vorhandene Komponenten durch alternative Implementierungen zu ersetzen.

- **FunktionPool** – Für kleinere Funktionseinheiten, die nicht zu komplex sind und für die eine eigene Komponente einen unverhältnismäßigen Aufwand darstellen würde, ist die Komponente `IFunctionPool` vorgesehen. Über sie lassen sich kleinere Funktionen als von der grafischen Benutzerschnittstelle aus ausführbare Einheiten realisieren.

- **Nachrichtenmodell und -verarbeitung** – Erweiterungen bringen es unter Umständen mit sich, dass weitere Nachrichtentypen zwischen der Mittelschicht und anderen unIDS-Systemkomponenten ausgetauscht werden müssen.

Innerhalb dieses Abschnittes werden die nötigen Schritte zur Erweiterung der Mittelschicht in diesen drei Bereichen vorgestellt und durch jeweils ein Beispiel aus der tatsächlichen Implementierung der Mittelschicht verdeutlicht.

4.12.1 Weitere Komponenten erstellen

Eine Komponente im Sinne der unIDS-Mittelschicht ist zunächst eine Klasse, die die Schnittstelle *IMiddlelayerComponent* bereit stellt[35]. Damit die Komponente dann auch in der Mittelschicht gefunden und automatisch geladen wird, muss sie noch innerhalb der Konfigurationsdatei beschrieben werden. Hierzu ist es zunächst erforderlich, die spezielle Implementierung der Komponente selbst dort einzutragen und danach eine passende Schnittstelle als Rolle einzutragen oder gegebenenfalls die für eine schon vorhandene Rolle eingestellte Implementierung durch eine andere zu ersetzen.

[35] Siehe hierzu Abbildung 4.2.

Somit ist für eine neue Komponente in der Regel auch eine entsprechende Schnittstelle zu entwerfen, die der Rolle der Komponente entspricht. Ein gutes Beispiel für eine Komponente bildet das Paket `unids.middelelayer.comp.webserver`. In diesem Paket ist die Implementierung des in die Mittelschicht integrierten Webservers enthalten. Hinzu kommen entsprechende Abschnitte in der Standard-Konfigurationsdatei.

Im Folgenden wird dies anhand der Komponente `Webserver` beispielhaft vorgestellt. Die Definition der Schnittstelle und auch die Implementierung des Webservers befinden sich im Paket `unids.middlelayer.comp.webserver` der Mittelschicht.

Rolle festlegen

Die Komponente `Webserver` soll die grafische Benutzerschnittstelle per HTTP bereitstellen, so dass man das unIDS-System auch ohne einen separaten Webserver nutzen kann. Für diese Aufgabe wird die Rolle *IWebserver* definiert. Innerhalb der Rolle sollte genau die Schnittstelle definiert sein, über die andere Komponenten der Mittelschicht auf diese Komponente zugreifen können. In diesem Fall stellt der Webserver keine Funktionen oder Attribute für andere Komponenten bereit und die Schnittstelle *IWebserver* wird ohne Erweiterungen von der Schnittstelle *IMiddlelayerComponent* abgeleitet[36].

Implementierung erstellen

Die tatsächliche Implementierung der Komponente muss immer auch die Schnittstelle ihrer Rolle umsetzen. In diesem Fall wurde für den Webserver die Klasse `Webserver` angelegt, die die Schnittstelle *IWebserver* implementiert. In der Regel werden die Komponenten der Mittelschicht von der Klasse `AbstractMiddlelayerComponent` abgeleitet. Bei Erweiterungen an den Methoden `init`, `start`, `stop` und `unload` ist es besonders wichtig, dass auch weiterhin die überschriebenen Methoden mit ausgeführt werden, da sie den internen Zustand der Komponente verwalten.

Konfiguration anpassen

Die neue Komponente und gegebenenfalls auch die neue Rolle müssen nun in der Konfigurationsdatei der Mittelschicht gemäß der Beschreibung in Abschnitt 4.4.4 eingetragen werden. Für den Webserver bedeutet dies, dass zunächst im Abschnitt `registry` innerhalb des Unterabschnitts `roles` die Rolle *IWebserver* bekannt gemacht werden muss. Hierzu erfolgt der folgende Eintrag:

```
<role id="webserverRole">
  <interface>
    <package>unids.middlelayer.comp.webserver</package>
    <name>IWebserver</name>
  </interface>
  <default-component ref="webserver"/>
</role>
```

[36]Siehe auch Abschnitt 4.4.2 für weitere Informationen über das Komponentenmodell der Mittelschicht.

Im zweiten Schritt muss im Unterabschnitt `components` die neue Implementierung der Rolle *IWebserver* hinzugefügt werden. Hierzu ist der folgende Eintrag geeignet:

```
<component id="webserver">
  <class>
    <package>unids.middlelayer.comp.webserver</package>
    <name>Webserver</name>
  </class>
  <config>
    <bind iface="localhost" port="1078"/>
    <parameter name="guiroot" value="../../tmp/gui_www"/>
  </config>
</component>
```

Damit ist nun festgelegt, dass für die Rolle *IWebserver* die Implementierung `Webserver` mit der innerhalb des `component`-Tags angegebenen Konfiguration verwendet werden soll. Beim nächsten Start der Mittelschicht wird somit automatisch eine instanz des Webservers erstellt und ihr wird der Inhalt des `config`-Tags als Konfiguration übergeben.

Weitere Hinweise zum Aufbau der Konfigurationsdatei finden sich in Abschnitt 4.4.4.

4.12.2 Neue Funktionen für den FunctionPool

Der `FunctionPool` selbst ist als eine eigene Komponente für die Mittelschicht realisiert. Er bietet die Möglichkeit, relativ einfache Funktionen über die grafische Benutzerschnittstelle bereitzustellen. Die Implementierung befindet sich im Paket `unids.middlelayer.functionpool`.

Um eine weitere Funktion hinzuzufügen ist diese zunächst zu erstellen. Danach kann über eine Erweiterung in der Konfigurationsdatei die neue Funktion in den `FunctionPool` eingetragen werden. Nach diesen Schritten ist die Funktion bereits über die grafische Benutzerschnittstelle anwählbar.

Im Folgenden werden die dafür nötigen Schritte anhand der Funktion `initState` beschrieben. Zweck dieser Funktion ist es für die Entwicklung des unIDS-Systems eine Möglichkeit zu schaffen, das Datenmodell der Mittelschicht wieer in einen definierten Zustand zu bringen. Die Funktion `initState` löst diese Anforderung dardurch, dass sie das Datenmodell komplett neu initialisiert und somit eine Art Reset durchführt.

Implementierung

Die Implementierung einer neuen Funktion für die Komponente `IFunctionPool` muss die Schnittstelle *IFunction* implementieren. Über diese Schnittstelle wird die Funktion später initialisiert und auch aufgerufen. Im Allgemeinen bietet es sich an, neue Funktionen von der Klasse `BaseFunction` abzuleiten. Die Implementierung der Function `initState` findet sich im Paket `unids.middlelayer.util.defaultstate`. Die Funktion selbst ist durch die Klasse `DefaultState` implementiert.

Der Rückgabewert jeder Funktion für den `FunctionPool` muss die Schnittstelle *IFunctionResult* bereitstellen, damit das Ergebnis von der Mittelschicht weiterverarbeitet werden kann und der aufrufenden Stelle korrekt zugestellt werden kann.

Konfiguration erweitern

Damit die Funktion auch tatsächlich innerhalb des unIDS-Systems zur Verfügung steht, muss sie zunächst in die Konfiguration der Komponente `FunctionPool` aufgenommen werden. Hierzu ist der Abschnitt `config` um folgende Zeilen zu erweitern:

```
<function name="initState"
          package="unids.middlelayer.util.initialstate"
          class="DefaultState"/>
```

Nach dem nächsten Neustart der Mittelschicht steht die neue Funktion dann zur Verfügung und kann über die grafische Benutzerschnittstelle aufgerufen werden.

4.12.3 Erweiterungen des Nachrichtenmodells

Der letzte Punkt, in dem spezielle Vorkehrungen zur leichten Erweiterbarkeit getroffen wurden ist der Bereich der Nachrichtenverarbeitung und des Nachrichtenmodells an sich[37]. Im folgenden wird die Erweiterung des Nachrichtenmodells anhand der Komponente `FunctionPool` vorgestellt, die auch schon im vorherigen Abschnitt angesprochen wurde.

Die Zielsetzung diese Komponente ist es, einen Pool für Funktionen bereitzustellen, der von der grafischen Benutzerschnittstelle aus aufgerufen werden kann. Für die Abfrage der vorhandenen Funktionen und für den Aufruf einer Funktion sind spezielle Nachrichten notwendig, die im bisherigen Nachrichtenmodell der Mittelschicht nicht enthalten waren. Hierzu werden die drei Nachrichten `FunctionMessage` (Aufruf von Functionen), `FunctionsMessage` (Abfrage einer Liste aller Funktionen) und `FunctionResultMessage` (Rückgabe / Ergebnis einer Funktion) definiert[38]. Im Folgenden werden die nötigen Schritte anhand der `FunctionMessage`-Nachricht beispielhaft vorgestellt.

Schnittstelle der Nachricht festlegen

Zunächst ist für jeden Nachrichtentyp eine Schnittstelle zu definieren, die festlegt, welche Attribute und Operationen Nachrichten diesen Typs bereitstellen. Die Schnittstelle erfüllt mehrere Zwecke: Zunächst stellt sie eine Art Vertrag dar, der festlegt, wie die Nachrichtenobjekte später zu benutzen sind. Zum anderen können sich ,wie später beschrieben, Komponenten zur Nachrichtenverarbeitung für bestimmte Nachrichtenschnittstellen registrieren. Sobald passende Nachrichten innerhalb der Mittelschicht zu verarbeiten sind, werden sie an alle für die von der Nachricht bereit gestellten Schnittstellen registrierten Stellen weitergeleitet. Die Schnittstelle für die `FunctionMessage` ist recht einfach gehalten, da für einen Funktionsaufruf nur der Name der Funktion und gegebenenfalls Parameter erforderlich sind. Daneben sind die für alle Nachrichten erforderlichen Eigenschaften der Schnittstelle *IMessage* bereitzustellen. Die Schnittstelle *IFunctionMessage* ist in Abbildung 4.14 grafisch dargestellt.

[37] In der Version 1.0 des unIDS-System ist dieser Aspekt bisher nur in der Kommunikationsschnittstelle zur grafischen Benutzerschnittstelle vollständig umgesetzt. Siehe hierzu auch den Abschnitt 4.13.1 über die offenen Features, die nicht implementiert werden konnten.

[38] Die beschriebenen Klassen finden sich im Paket `unids.middlelayer.functionpool`.

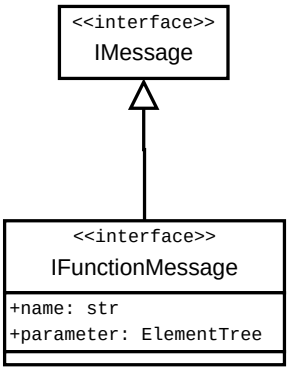

Abbildung 4.14: Schnittstelle IFunctionMessage

Implementierung der Nachrichtenklasse

Für jeden Nachrichtentyp muss mindestens eine Implementierung vorhanden sein. Somit muss für neue Nachrichtentypen immer auch eine Implementierung erstellt werden. Die Implementierung repräsentiert den Inhalt einer Nachricht und wird zum Einen Verwendet, um Nachrichten besser verarbeiten zu können und zum Anderen zur Erstellung der XML-Repräsentation einer Nachricht verwendet. Die Klasse `FunctionMessage` implementiert den gleichnamigen Nachrichtentyp. In Abbildung 4.15 ist die Vererbungshierarchie der verwendeten Klassen dargestellt.

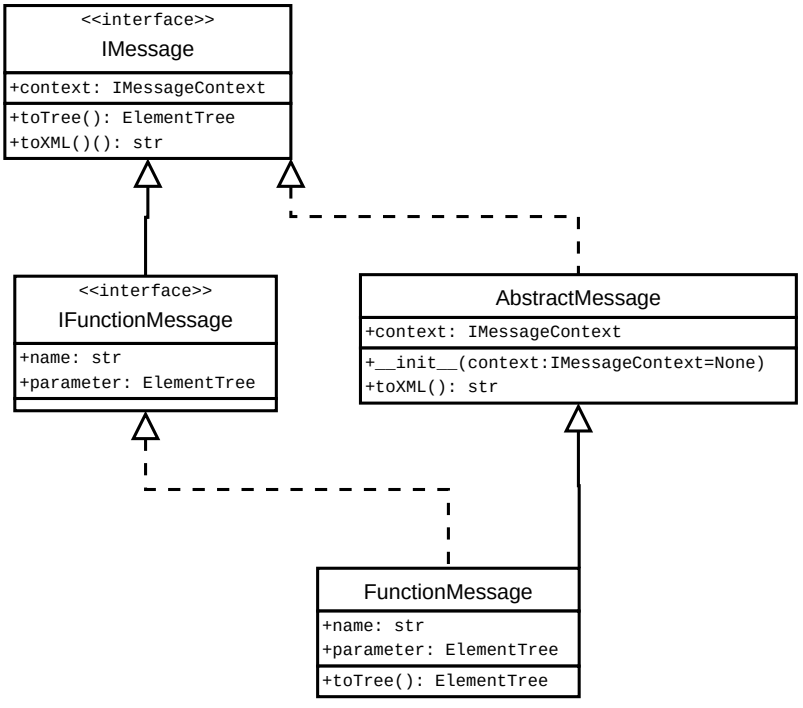

Abbildung 4.15: Klassenhierarchie zur Klasse FunctionMessage

Für die Erstellung neuer Nachrichtentypen bietet es sich an, diese von der Klasse `AbstractMessage` aus dem Paket `unids.middlelayer.guimsg.base` abzuleiten. Neben der Schnittstelle des Nachrichtentyps muss dann nur noch die Methode `toTree` implementiert werden. Die Methode `toXML` der Klasse `AbstractMessage` wandelt das Ergebnis von `toTree` dann nur noch in eine Zeichenkette um.

Factory erstellen

Damit eingehende Nachrichten im XML-Format korrekt in die passenden Nachrichtenformate zugeordnet werden können, muss für den neuen Nachrichtentyp noch eine passende Factory erstellt werden. Die Aufgabe dieser Factory ist die Analyse der eingehenden XML-Nachrichten und die Erstellung der richtigen Nachrichtenobjekte aus diesen XML-Nachrichten. Da für Funktionsaufrufe nur der Name und möglicherweise ein XML-Fragement als Parameter des Funktionsaufrufs verwendet werden, ist die Factory für den `FunctionPool` eher einfach gehalten und kann hier vollständig wiedergegeben werden:

```
class FunctionMessageBuilder(object):

    implements(interfaces.IMessageBuilder)

    def buildMessage(self, xmlTree):
        instance = FunctionMessage()
        instance.name = xmlTree.get(tag.NAME_ATTR)
        instance.parameters = xmlTree.find(tag.PARAMETER_TAG)
        return instance
```

Am Ende der Funktion `buildMessage` steht dann das fertige Nachrichtenobjekt und wird zurückgeliefert.

Listener für die Nachrichtenobjekte

Nach der Erstellung einer passenden Factory für die Nachrichtenobjekte fehlt noch eine passende Klasse für die Verarbeitung der Nachrichten. Hierzu ist eine Implementierung der Schnittstelle *IMessageListener* zu erstellen. Für die `FunctionMessage`-Nachrichten wurde die Klasse `FunctionMessageListener` erstellt. Sie ist in der Lage mit einem gegebenen *IFunctionMessage*-Objekt die durch es bezeichnete Funktion auszuführen. Dazu erhält der Nachrichten-Handler bei seiner Instanziierung eine Referenz auf die `FunctionPool`-Komponente der Mittelschicht.

Registrierung der Factory und des Listeners

Da Funktionen nur durch die grafische Benutzerschnittstelle aufgerufen werden können, muss die Factory sowie der Listener für den neuen Nachrichtentyp im System bekannt gemacht werden. Für die Entgegennahme der Nachrichten und ihrer Weiterverarbeitung ist die Komponente `GuiInterface`[39] verantwortlich. Daher müssen die Factory und der Listener bei dieser Komponente registriert werden, damit sie in der Lage ist, die neuen Nachrichtentypen zu verstehen. Die notwendigen Initialisierungsarbeiten finden in der Methode `init` der Komponente `FunctionPool` statt.

Zunächst ist eine Referenz auf die Komponente `GuiInterface` notwendig, damit bei ihr Factory und Listener registriert werden können. Dies wird innerhalb der `init`-Methode durch folgende Zeilen erreicht:

[39] Siehe auch Abschnitt 4.6.

```
guiiface = self._getRoleImplementation(
    interfaces.IGuiInterface)
```

Die von `AbstractMiddlelayerComponent` geerbte Methode `_getRoleImplementation` nutzt eine Referenz auf die Komponente `ServiceManager` um an die Implementierung der Rolle *IGuiInterface* zu gelangen[40]. Die Komponente `IGuiInterface` stellt zur Registrierung die Methoden `registerMessageBuilder` für die Factory und `registerMessageListener` für den Listener zur Verfügung. In den folgenden Zeilen der Methode `init` aus der Klasse `FunctionPool` werden dieser Methoden genutzt um die neuen Nachrichtentypen in die Mittelschicht zu integrieren:

```
guiiface.registerMessageBuilder(
    FunctionMessageBuilder(),
    tag.FUNCTION_TAG)
guiiface.registerMessageListener(
    FunctionMessageListener(self),
    IFunctionMessage)
```

Damit ist die Mittelschicht nun um neue Nachrichtentypen erweitert, die automatisch in die richtigen Nachrichten-Objekte umgewandelt und an die richtige Stelle zur Weiterverarbeitung geleitet werden. Die gesamte Implementierung der neuen Nachrichtentypen sowie der dazugehörigen Komponente `FunctionPool` findet sich im Paket `unids.middlelayer.functionpool` wieder.

4.13 Ausblick

Abschließend für den Bereich der Mittelschicht folgt ein Ausblick. Zunächst werden die geplanten aber nicht mehr realisierten Features der Mittelschicht beschrieben. Darauf folgt eine Aufzählung der Stellen an denen der Quelltext oder die Architektur der Mittelschicht überarbeitet werden sollte. Nach einem Überblick über den konzeptionellen Weiterentwicklungsbedarf in der Mittelschicht folg dann eine Liste der bekannten Fehler und zum Abschluss ein Fazit für den Bereich der Mittelschicht.

4.13.1 Offene Features

Eine liste aller offenen und wünschenswerter Features ist im `trac` zu finden. https://svn.wirebrain.de:8081/t/studies/unids2/report/3
Die hier zitierte Liste spiegelt dem Stand vom 03.10.2005 wider.

- **Ticket 126: Antworten der Gui verarbeiten** – Antworten der Gui werden von der Mittelschicht bisher nicht verarbeitet. Beispiel: Wenn eine IDMEF Nachricht zur Gui geschickt wird. Dies wird von der Mittelschicht in einer WARNING log-Meldung protokolliert.
  Die Antworten sollten in sinnvoller Weise verarbeitet werden. Vielleicht in einer Art Event-Log in der DB? Bisher fehlen hierfür die nötigen Konzepte.

[40] Weitere Informationen zum Zusammenhang von Komponenten und Rollen finden sich im Abschnitt 4.4.4 über die Konfiguration der Mittelschicht.

- **Ticket 25: Berechnung Maximum für Status-Werte** – Es ist festzulegen, wie das Maximum für die verschiedenen Status-Werte berechnet bzw. ermittelt werden soll. Bisher sind dazu keine Angaben gemacht. Danach ist diese Festlegung entsprechend umzusetzen.

- **Ticket 29: Schema Benutzerverwaltung** – - *keine beschriebung* -

- **Ticket 63: IDMEF - Speziellere Auswertungen** – IDMEF - Nachrichten könnten ggf. nach bekannten Angriffen ausgewertet werden und dafür entsprechende Statistiken erstellt werden. Damit könnte man für ein Gerät visualisieren, welche Angriffe wie häufig vorkommen. Oder auch in einer Matrix für eine Menge von Geräten. Zunächst ist hierzu konzeptionelle Vorarbeit zu leisten. Daraus sollten dann auch genauere Aufgaben resultieren, wie die Funktionalität umzusetzen ist.

- **Ticket 98: Zugriffsberechtigungen für alle Entitäten entwerfen** – In der späteren Version von unIDS muss für viele Entitäten ein Zugriffsschutz realisiert werden. Dazu möglichst ein generelles Konzept erstellen, das prinzipiell für fast alle Entitäten oder Funktionen genutzt werden kann.
  Schön wäre ein Rollenbasierte Lösung. Nach der Planung sind entsprechende Tickets f.d. Umsetzung zu erstellen und einzuplanen.

- **Ticket 120: Nachricht Netzwerkstruktur spezifisch f.d. aktuellen Benutzer** – In der Nachricht über die Netzwerkstruktur (Network, Subnet, Devices ...) werden bisher alle vorhandenen Objekte übermittelt, unabhängig davon, ob der aktuelle Benutzer diese sehen darf, oder nicht.
  Die Nachricht muss so zusammengestellt werden, dass nur noch die Objekte enthalten sind, die der aktuelle angemeldete Benutzer auch sehen darf.
  Weiter ist im Rahmen dieses Tickets festzulegen, wie mit Fällen umgegangen wird, wenn z.B. ein Subnetz nicht sichtbar ist, aber die darin enthaltenen Devices nicht gesperrt sind. Die Ergebnisse werden im Wiki dokumentiert und möglichst auch gleich in die technische Dokumentation übernommen.
  Nach Rücksprache mit der Gui wird die Nachricht für die Netzwerk-Struktur in diesem Rahmen auf das element - Konzept umgestellt. Siehe UnidsStruktur und NetworkStructure.

- **Ticket 123: Netzwerkinterfaces zu den Devices zuordnen** – Die Netzwerkstruktur wird um Schnittstellen (Netzwerk) erweitert. Diese sind in der Strukturnachricht unterzubringen und somit f.d. Gui verfügbar zu machen. Hierzu muss vermutlich auch der Agent weitere Informationen liefern, damit die Mittelschicht sagen kann, welche Netzwerkschnittstellen auf einem überwachten System zur Verfügung stehen. Dies ist mit der Agenten-Gruppe abzustimmen.
  Zunächst ist hierfür konzeptionelle Vorarbeit zu leisten, sowie die dafür notwendigen Schritte in Tickets zu gießen. Alle Vereinbarungen über Nachrichtenformate etc. sollen im Wiki dokumentiert werden.

- **Ticket 223: Beep Channel autostart** – Die Mittelschicht wird so erweitert, dass sie versucht, falls kein passender Kanal zur Verfügung steht, diesen selbst zu starten und dann erst der Nachrichtenversand scheitert.

4.13.2 Refactoring-Bedarf

Teilweise ist schon zu erkennen wo in der Mittelschicht noch Refactoring-Bedarf besteht. Wo dies erkannt wurde, sind daher auch schon Tickets im `trac` eingetragen, zurzeit allerdings noch unter dem Oberpunkt Offene Features. https://svn.wirebrain.de:8081/t/studies/unids2/report/3
Der zitierte Stand ist vom 03.10.2005.

- **Ticket 174: Refactoring: Filter für MessageListener hinzufügen** – Um genauer kontrollieren zu können, welche Nachricht wo behandelt werden soll, wird die MessageListenerRegistry um MessageFilter erweitert.
  Ein MessageFilter soll die Nachricht bekommen und entscheiden, ob sie durchgelassen wird (True) oder ob sie von diesem MessageListener nicht behandelt werden soll (False).

- **Ticket 180: Refactoring: ComponentContext refactoring** – Das interface des ComponentContext überarbeiten. Get- und Set-Methoden sollten in Attribute umgewandelt werden. Die Implementierung innerhalb des ComponentManagers ebenfalls anpassen und entsprechende TestCases erstellen, die zumindest das Interface validieren.

- **Ticket 182: Refactoring: MessageDispatcher wie eine MessageRegistry aufbauen** – Den MessageDispatcher wie eine MessageRegistry aufbauen.

- **Ticket 224: Refactoring: Nachrichtenverarbeitung angleichen** – In der Mittelschicht ist die Nachrichtenverarbeitung im ComponentInterface technisch überholt. Sie sollte auf die gleichen konzeptionellen Grundlagen gesetzt werden, wie die Nachrichtenverarbeitung im GuiInterface.

- **Ticket 226: Refactoring: Prototypischer Code im ComponentInterface** – Das folgende sollte vermutlich entfernbar sein:

```
# prototyp_start
      # TODO: Feature #
      if msg.payload.count('tracepath') > 0:
          rmsg = '<unids><set_id>1</set_id></unids>'
          log.debug('returning set_id message')
          # it's a trathpath, send set_id
          return rmsg
      # prototyp_stop
```

- **Ticket 227: Refactoring: Die Mittelschicht enthält an einigen Stellen noch legacy code** – Die Mittelschicht enthält in den Bereichen der Persistenz und des ComponentInterface aber teilweise auch im GuiInterface und anderen Bereichen noch einige Stellen mit Legacy-Code , der entfernt werden sollte.

4.13.3 Konzeptioneller Weiterentwicklungsbedarf

Konzeptionell sind vor allem zwei wesentliche Punkte noch weiterzuentwickeln. Dies ist zunächst die Einbindung von wirklichen Proxies in das unIDS-Modell. Hierzu

fehlen neben der dann geänderten Aufgabenverteilung zwischen Agent, Proxy und Mittelschicht auch die dazu nötigen Nachrichtenklassen. Strukturell ist das Datenmodell der Mittelschicht bereits so ausgelegt, dass auch Proxies mit untergrbracht werden könnten. Neben der Frage des Proxy-Konzepts steht der große Aufgabenbereich der Analyse in weiten Bereichen noch offen. Zunächst fehlen hierzu klare Zielsetzungen für das unIDS-Systems als solches und darauf aufbauend die erforderlichen Eigenschaften der Mittelschicht, damit eine zuverlässige Analyse des unIDS-Systemzustands möglich ist.

Neben diesen beiden großen Punkten bietet auch die Architektur vor allem im Bereich der Kommunikation zwischen Mittelschicht und Agent noch Entwicklungspotential hin zu einem ähnlichen Aufbau wie auf der Seite zur grafischen Benutzerschnittstelle.

4.13.4 Bekannte Fehler

Alle bekannten Fehler sind zusätzlich zu der hier folgenden Auflistung im trac zu finden. Die URL für das trac lautet: https://svn.wirebrain.de:8081/t/studies/unids2/report/3.
Die folgende zitierte Liste ist vom 03.10.2005.

- **Ticket 50: Abmelden der GUI** – Abmelden der Gui ist problematisch. Wenn die Verbindung abreißt, wird noch nicht richtig reagiert. Vermutlich muss man nur die entsprechende Exception richtig fangen und entsprechend die Session abschalten.

- **Ticket 225: Konfigurations Nachrichten werden von der Mittelschicht über den Datenkanal gesendet.** – Die Mittelschicht versendet Konfigurations Nachrichten über den Datenkanal, gemäß Spezifikation sollte dies über den Management - Kanal geschehen.

- **Ticket 231: Wenn die GUI sich abmeldet sollten alle Statusinformationsanforderungen entfernt werden** – - *keine Beschreibung* -

4.13.5 Verschiedenes

Hier werden verschiedene Sachen, Probleme und Ereignisse erläutert die für die Mittelschicht von Interesse sind.

4.13.5.1 BEEPy

`BEEPy` ist der Versuch das *Blocks Extensible Exchange Protocol* als Python-Bibliothek zu implentieren. Die Bibliothek hat derzeit ein Level erreicht das die Basis-Funktionalität bereitstellt, und somit geeignet ist um BEEP-Anwendungen zu implementieren.
Für BEEP siehe auch Abschnitt 2.2.6 BEEPy benötigt `twisted` um auf Socket-Ebene die Kommunikation über TCP bereitzustellen.

4.13.5.2 ZODB

`ZODB` ist nach eigener Aussage ein Persistenz-System für Python-Objekte [41]. Wobei ZODB auch als python-spezifische Objektorientierte-Datenbank verstanden werden kann, und dies auch von den Entwickler nicht verneint wird.
ZODB verwaltet dabei Objekte für den Programmierer, das heißt ZODB hält diese im Cache, schreibt sie auf die Platte oder löscht sie bei Bedarf ganz aus dem Cache. In der Mittelschicht benutzen wir ZODB für die Datenhaltung, siehe dazu auch Abschnitt4.9.1.

4.13.5.3 pygraphlib

`Pygraphlib` ist ein Python-Package zur Representation und Manipulation von (Netzwerk)-Graphen.
Die pygraphlib-Bibliothek enthält zum Beispiel Methoden um Graphen zu erstellen, zu traversieren oder um den kürzesten Weg auszurechen (shortest path alogrithm).

4.13.5.4 Persistenter Graph

In der Graphentheorie ist ein Graph eine Menge von Punkten (man nennt diese dann Knoten oder auch Ecken), die eventuell durch Linien (sog. Kanten bzw. Bögen) miteinander verbunden sind. Die Form der Punkte und Linien spielt in der Graphentheorie keine Rolle.
Persistenz bedeutet allgemein das etwas von dauerhafter Beschaffenheit ist, also nicht in unserem diesem Fall nicht nach beenden der Mittelschicht wieder gelöscht ist. Persistente Graphen werden in der Mittelschicht gebraucht um Netzwerkstrukturen zu speichen und auch zu manipulieren. Siehe dazu auch Abschnitt 4.13.5.3 oder Abschnitt 4.9.

4.14 Fazit

Das Aufgabenspektrum der Mittelschicht hat sich im laufe der Entwicklung des unIDS-Systems als wesentlich komplexer herausgestellt als zunächst bei der Planung des unIDS-Projekts angenommen. Erschwerend kam hinzu, dass von der alten unIDS-Version kaum Funktionalität übernommen werden konnte. Dies wurde insbesondere dadurch erschwert, dass die alte unIDS-Version nur sehr stark eingeschränkt nutzbar war. Die fatale Folge waren nur sehr knappe Entwicklerresourcen für die Mittelschicht, was dazu führte, dass nicht alle vorgesehenen Eigenschaften tatsächlich umgesetzt werden konnten, wie dies aus den vorangegangenen Abschnitten ersichtlich ist.

Speziell die zunächst unklaren Anforderungen an die Konkrete Datenhaltung und die Struktur der Daten an sich haben dazu geführt, dass einige umfangreichere Änderungen in der Architektur vorzunehmen waren, wie zum Beispiel der Wechsel vom relationalen Datenmodell hin zu einer Objektdatenbank, der einen erheblichen Teil der bisherigen Entwicklung überflüssig machte, dafür aber die Mittelschicht an sich wesentlich übersichtlicher und leicher wartbar gemacht hat.

[41] http://www.zope.org/Wikis/ZODB/FrontPage/guide/index.html, Stand: 04.10.2005

Positiv zu bewerten ist dass sich das Archtikturmodell, das letztendlich entwickelt wurde, bewährt hat. Es bietet komfortable Möglichkeiten um die Mittelschicht modular zu erweitern, wie dies bereits in Abschnitt 4.12 ausführlich beschrieben worden ist. Über den Rand der Mittelschicht hinaus hat sich zwischen den verschiedenen unIDS-Komponenten ein sehr flexibles und gut erweiterbares Nachrichtenmodell etabliert.

Insgesamt bildet die Mittelschicht eine robuste Basis, die im Vergleich zur alten unIDS-Version wesentlich stabiler, aufgeräumter und klarer ist, auch wenn einige der geplanten Eigenschaften mit den vorhandenen Mitteln nicht realisiert werden konnten. Somit wurde zwar nicht jedes einzelne kleine Ziel erreicht, aber dennoch die große Zielsetzung im Sinne einer Weiter- bzw. Neuentwicklung des unIDS-Systems auch im Bereich der Mittelschicht erfolgreich umgesetzt.

5. Agent

Ein Agent stellt im unIDS-System die unterste Schicht dar und ist ein sog. *Monitoring-Client*, das auf einem Host ausgeführt wird, um in erster Linie Informationen von IDS (*Intrusion Detection Systemen*) zu sammeln und diese weiter an die Mittelschicht zu leiten. Der Zugriff auf diese Informationen geschieht dabei über Plugins, die eine Schnittselle zwischen dem Agenten und den eingesetzten IDS bilden. Zusätzlich kann ein Agent über Plugins verfügen, die unterschiedliche Logdateien parsen oder sicherheitsrelevante Systeminformationen auslesen.

Bevor auf die Implementierung des Agenten eingegangen werden kann, werden im Abschnitt 5.1 die Zielsetzungen und Anforderungen an den zu entwerfenden Agenten formuliert. Im anschließenden Abschnitt 5.2 werden alle wesentlichen Anwendungsfälle definiert, die eine abgegrenzte Funktionalität des Agenten zusammenfassen. Nachdem nun die notwendigen Konzepte festgelegt wurden, widmet sich das Kapitel 5.3 zunächst der Realisierung des Agenten, gefolgt von den konkreten Implementierungshinweisen, welche die Entwicklung der einzelnen Plugins betreffen.

5.1 Zielsetzungen und Anforderungen

Aus Anwendersicht ist das gewünschte Ziel, dass der Agent auf einem Rechner ausgeführt wird, um auf diesem eine möglichst effiziente und konstruktive Überwachung von sicherheitsrelevanten Informationen durchzuführen. Hierzu verlässt sich der Agent u.a. auf die Vorgehensweise installierter IDS-Programme, die dem Agenten Informationen über stattfindende Angriffe bzw. Missbrauchsversuche liefern. Die gesammelten Information müssen dann an die Mittelschicht übergeben werden, bevor sie dem Anwender aufbereitet in der GUI zur Verfügung gestellt werden können. Als Schnittstelle zu den unterschiedlichen IDS ist der Einsatz von sog. Plugins unabdingbar. Die Plugins sind dafür zuständig, dass die Informationen eines IDS gesammelt oder Logdateien mit sicherheitsrelevanten Daten analysiert werden. Die Verwendung von Plugins gestattet es, dass der Agent nachträglich um unterschiedliche Funktionalitäten erweitert werden kann. Beispielsweise ist es durch dieses Konzept möglich, relativ problemlos ein Plugin für ein neues IDS zu entwickeln und in die vorhanden Software des Agenten einzubinden. Die unterschiedlichen Plugins stellen also

sehr flexible Detektionsfähigkeiten zur Verfügung, welche alternativ oder gleichzeitig genutzt werden können. Damit nun eine nachträgliche Integration von Plugins realisierbar ist, muss die Software des Agenten modular aufgebaut werden.

Ein vorrangiges Anliegen bei der Entwicklung des Agenten besteht also darin, ein sog. *Monitoring* von sicherheitsrelevanten Informationen zu realisieren und dabei ein anwendbares Rahmenwerk (*Framework*) für den Einsatz von Plugins bereitzustellen. Zur Erfüllung dieser Anforderungen muss der Agent in der Lage sein, folgende Aufgaben zu erfüllen:

- Auf- und Abbau einer Verbindung zur Mittelschicht
- Realisierung eines clientseitgen BEEP-Protokoll (Senden und Empfangen von BEEP-Nachrichten)
- Verarbeitung von XML-basierten BEEP-Nachrichten
- Aufrechterhaltung und Überwachung der Verbindung zur Mittelschicht
- Bereitstellung und Verwaltung von allokierten BEEP-Kanälen
- Laden von verfügbaren Plugins
- Aktivieren und Deaktivieren von einzelnen Plugins

5.2 Anwendungsfälle

Aus den oben beschriebenen Zielsetzungen lassen sich unterschiedliche Anwendungsfälle für den Agenten ableiten. In den folgenden Abschnitten sollen diese näher erläutert werden.

5.2.1 Anmelden in der Mittelschicht

Nachdem das *Management* Kanal zur Mittelschicht aufgebaut wurde, meldet der Agent seine Bereitschaft in der Mittelschicht an. Bei der Anmeldung sind zwei Fälle zu unterscheiden:

1. Verfügt der Agent über keine eindeutige ID, so liegt eine erstmalige Anmeldung vor, die mit einer ID-Anforderung an die Mittelschicht verbunden ist. Die Anmeldung erfolgt dabei mit der Aussendung einer sog. `GreetingMessage`, die das Attribut *request_id* beinhaltet. Die Mittelschicht quittiert diese Anforderung ebenfalls mit einer `GreetingMessage`, die neben dem Attribut *set_id* eine eindeutige ID für den Agenten enthält.
2. Verfügt der Agent bereits über eine eindeutige ID, die bei der Initialisierung des Agenten aus einer lokalen Datei eingelesen wird, so erfolgt die Anmeldung in der Mittelschicht ohne eine ID-Anforderung, d.h. die `GreetingMessage` wird ohne das Attribut *request_id* an die Mittelschicht gesendet. Die Mittelschicht quittiert diese Anmeldung ebenfalls mit einer `GreetingMessage`, die neben dem Attribut *set_id* die ID des Agenten enthält. Die ID dient dem Agenten als Absender-ID bei künftigen Nachrichtenübertragungen.

Nach der Anmeldung in der Mittelschicht initiiert der Agent den Aufbau von zusätzlichen BEEP-Kanälen für die Profile *Data* und *IDXP*, über welche letztendlich die gesammelten Information in Form von spezifizierten XML-Nachrichten an die Mittelschicht gesendet werden können.

5.2.2 Abmelden aus der Mittelschicht

Der Agent meldet sich aus der Mittelschicht ab, wenn das Programm des Agenten beendet wird. Hierzu sendet der Agent die XML-Nachricht `SignOffMessage`. Nach einer ordnungsgemäßen Abmeldung des Agenten aus der Mittelschicht ist eine erneute Anmeldung des Agenten in der Mittelschicht möglich.

5.2.3 Laden und Starten von einzelnen Plugins

Nach der erfolgreichen Anmeldung des Agenten an der Mittelschicht und dem Aufbau der drei logischen BEEP-Kanäle (*Managemant*, *Data* und *IDXP*) können im Agenten alle Plugins, die in einer Konfigurationsdatei aufgeführt sind, geladen werden. Die Konfigurationsdatei legt auch fest, welche von den verfügbaren Plugins unmittelbar nach dem Laden gestartet werden sollen.

5.2.4 Änderung der Konfiguration von einzelnen Plugins

Über die GUI hat der Anwender die Möglichkeit, die Konfiguration von einzelnen im Agenten verfügbaren Plugins zu beeinflussen. Die GUI sendet hierzu in einer XML-Nachricht eine entsprechende Konfigurationsänderung an die Mittelschicht. Die Mittelschicht verarbeitet diese Nachricht und leitet die Konfigurationsänderung in der XML-Nachricht `ConfigurationMessage` an den Agenten weiter. Der Agent vergleicht den Inhalt der empfangenen `ConfigurationMessage` mit der aktuellen Konfiguration und ändert gegebenenfalls die aktuelle Konfiguration entsprechend der neu angeforderten Konfiguration. Über eine `ConfigurationMessage` können beispielsweise einzelne Plugins ein- bzw. abgeschaltet oder bestimmte Plugin-Parameter nachträglich verändert werden.

5.2.5 Senden von gesammelten Informationen

Der wichtigste Anwendungsfall für das gesamte unIDS-System stellt die Aussendung von Informationen dar, welche die Plugins z.B. von einem IDS gesammelt oder aus einer Logdatei eingelesen haben. Ohne der Bereitstellung dieser Informationen hat der Agent gewissermaßen keine Relevanz für das gesamte unIDS-System. Um die Aussendung dieser Informationen zu ermöglichen, stellt die Software des Agenten ein Rahmenwerk zu Verfügung, das die erforderlichen BEEP-Kanäle (*Data* und *IDXP*) unterhält und eine Methode für das Senden von gesammelten Informationen implementiert. Sobald ein Plugin über entsprechende Informationen verfügt, sendet er diese in Form von XML-Nachrichten über den korrespondierenden BEEP-Kanal an die Mittelschicht.

5.3 Implementierung

Nachdem nun in den vorhergehenden Abschnitten die notwendigen Anforderungen und Anwendungsfälle vermittelt wurden, widmet sich dieses Kapitel der Implementierung des Agenten und der einzelnen Plugins. Im Abschnitt 5.3.1 werden zunächst

konkrete Aspekte der Implementierung des Agenten beschrieben, bevor im Abschnitt 5.3.2 die Realisierung von einzelnen Plugins erläutert wird. Des Weiteren wird auf einige Besonderheiten und Probleme eingegangen, die während dieser Phase aufgetreten sind.

5.3.1 Hauptkomponenten

Die Implementierung wurde in verschiedene Python-Quelltextmodule aufgeteilt. Jedes der Quelltextmodule enthält mehrere Klassen, die eng zusammenarbeiten und vom Sinn her zusammengehören. Durch die Aufteilung des Quelltextes auf mehrere Quelltextmodule ist der Quelltext besser organisiert und übersichtlicher. Neben den drei Modulen `Agent.py`, `Plugin.py` und `PluginManagement.py`, welche die Hauptkomponenten des Agenten bilden, ist jedes der implementierten Plugins in einem eigenständigen Modul gekapselt.

Die einzelnen Module sollen nun in den folgenden Abschnitten näher betrachtet werden. Zur Veranschaulichung werden dabei u.a. Klassendiagramme genutzt, die neben dem Aufbau von einzelnen Klassen (Klassenname, Attribute und Methoden) auch die Beziehungen zwischen den Klassen zeigen.

5.3.1.1 Agent.py

Das Modul `Agent.py` stellt die zentrale und damit wichtigste Komponente des Agenten dar. In `Agent.py` sind alle Klassen implementiert, die zur Ausführung des Agenten relevant sind.

Die gesamte Kommunikationsschnittstelle zur Mittelschicht stützt sich dabei auf die frei verfügbare Python-Bibliothek `BEEPy`, die ebenfalls bei der Entwicklung der Mittelschicht eingesetzt wurde. Folglich wurde auch im Agenten bei der Realisierung eines BEEP-Protokolls, der die Nachrichtenübertragung zwischen dem Agenten und der Mittelschicht regelt, auf die Klassen und Routinen von `BEEPy` zurückgegriffen. Die Verwendung von `BEEPy` erlaubt es, relativ einfach und schnell ein BEEP-Protokoll in der Programmiersprache Python umzusetzen.

Die gesamte Kommunikationsschnittstelle des Agenten ist in der Klasse `AgentProtocol` implementiert, welche von der `BEEPy`-Klasse BeepClientProtocol abgeleitet ist. In Ihr sind alle erforderlichen Attribute und Methoden gekapselt, die eine clientseitige auf BEEP-basierte Kommunikation zur Mittelschicht ermöglichen.

Bei einem Verbindungsaufbau mit der Mittelschicht erzeugt automatisch das `BEEPy`-Framework eine Instanz von der Klasse `AgentProtocol` und entfernt automatische diese, wenn die Verbindung abgebaut wurde. Nach einem erfolgreichen Verbindungsaufbau zur Mitteschicht und der damit verbundenen Instanzierung von `AgentProtocol`, wird vom `BEEPy`-Framework automatisch die Methode `greetingReceived()`, die in `AgentProtocol` überschrieben ist, aufgerufen. Innerhalb `greetingReceived()` erfolgt dann mittels der Methode `newChannel()` der Aufbau eines neuen logischen BEEP-Kanals, und zwar für den Kanaltyp *Management*, dessen Eigenschaften (*Profile*) in der Datei `ManagementProfile.py` definiert sind. Einen erfolgreichen Kanalaufbau signalisiert das `BEEPy`-Framework durch den Aufruf der Methode `channelStarted()`, die ebenfalls in `AgentProtocol` überschrieben ist. In `channelStarted()` wird dann die Anmeldephase initiiert, die im Abschnitt 5.2.1 beschrieben ist.

Für die Nachrichtenübertragung stehen dem Agenten die drei logischen Kanäle *Management*, *IDXP* und *Data* zur Verfügung, deren Eigenschaften in den *Profile*-Dateien `ManagementProfile.py`, `IdxpProfile.py`, und `DataProfile.py` spezifiziert sind. Die Behandlung von eingehenden Nahrichten aus der Mittelschicht geschieht ereignisgesteuert und asynchron. D.h. beim Eintreffen einer BEEP-Nachricht aus der Mittelschicht wird in `AgentProtocol` die Methode `processMSG()` bzw. `processRPY()` aufgerufen, um eine Verarbeitung der empfangenen BEEP-Nachricht, die vom Typ MSG bzw. RPY ist, einzuleiten. In den Methoden `processMSG()` bzw. `processRPY()` wird also die eigentliche Nachrichtenbehandlung durchgeführt.

Für das Senden von Nachrichten stellt die Klasse `AgentProtocol` die Methoden `sendManagementMsg()`, `sendDataMsg()` und `sendIdxpMsg()`zur Verfügung, deren Gebrauch abhängig vom korrespondierenden Kanal (*Management*, *Data* bzw. *IDXP*) ist. Bei IDMEF-Nachrichten beispielsweise, die über den logischen Kanal *IDXP* verschickt werden, wird demzufolge die Methode `sendIdxpMsg()` aufgerufen.

Das eigentliche Starten und Beenden des Agenten wird von der Klasse `AgentService`, die von der `Twisted`-Klasse `MultiService` abgeleitet ist, geregelt. In `AgentService` ist die Funktionalität eines sog. Services implementiert, welches das `Twisted`-Framework ausführt, wenn der Agent als eine `Twisted`-Applikation gestartet oder beendet wird. In der Methode `startService()` ist hierzu definiert, welche Operationen während eines Starts durchzuführen sind. Dazu zählen das Laden von Profile-Dateien, Einlesen von agentspezifischen Parametern aus einer Konfigurationsdatei sowie die Instanzierung und Aktivierung von `AgentProtocol`. In der Methode `loadConfiguration()` wird also aus einer in XML-Format aufgebauten Konfigurationsdatei (`agent.xconf`) die Konfiguration für den Agenten eingelesen. Neben den Parametern, wie Port und IP-Adresse für die Kommunikation mit der Mittelschicht, werden auch Informationen über verfügbare Plugins gesetzt. Eine Beispielkonfiguration ist im nachfolgenden Listing gegeben.

```
<configuration>
  <manager host="localhost" port="2004"/>
  <plugins>
    <plugin name="Syslog" enabled="1">
      <parameter key="input_facility" value="pipe"/>
      <parameter key="filename" value="/tmp/syslogpipe"/>
    </plugin>
    <plugin name="Snort" enabled="1">
      <parameter key="input_facility" value="tcp"/>
      <parameter key="host" value="localhost"/>
      <parameter key="port" value="50001"/>
    </plugin>
    <plugin name="SystemStat" enabled="1">
      <parameter key="update_interval" value="30" editable="1"/>
    </plugin>
    <plugin name="NetworkStat" enabled="1">
      <parameter key="update_interval" value="30" editable="1"/>
    </plugin>
  </plugins>
</configuration>
```

Nachdem der Agent konfiguriert wurde, wird mittels der `Twisted`-Methode `setServiceParent()` der Agent als eine `Twisted`-Applikation gestartet. Von nun an regelt das `BEEPy`-Framework in Zusammenarbeit mit `Twisted` die Ausführung von `AgentProtocol` und somit auch die Kommunikationsschnittstelle zur Mittelschicht.

Für ein ordnungsgemäßes Beenden des Agenten ist in `AgentService` die Methode `stopService()`, die vom `Twisted`-Framework aufgerufen wird, zuständig. `stopService()` terminiert zunächst die Ausführung von sämtlichen laufenden Plugins, bevor im zweiten Schritt mit der Aussendung einer `SignOffMessage` die Abmeldung aus der Mittelschicht eingeleiten wird (siehe Abschnitt 5.2.2). Im nächsten Schritt wird die aktuelle Konfiguration des Agenten in die Konfigurationsdatei `agent.xconf` geschrieben, falls diese von der Startkonfiguration abweicht (siehe auch Abschnitt 5.2.4). Als letztes wird `service.MultiService.stopService(self)` ausgeführt, um die `Twisted`-Applikation Agent zu beenden.

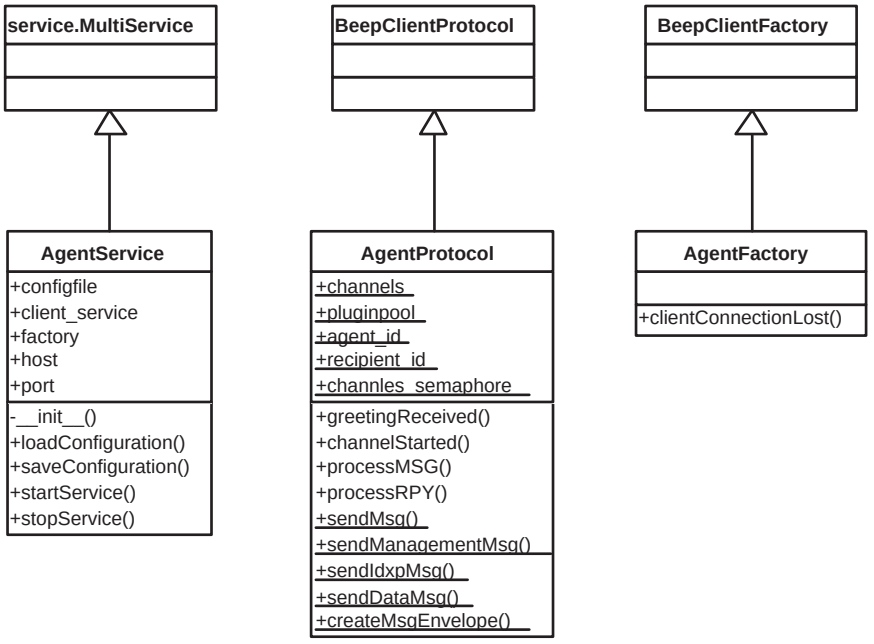

Abbildung 5.1: Klassendiagramm der Hauptkomponente Agent.py

5.3.1.2 Plugin.py

Das Modul `Plugin` enthält einerseits Elemente, um mehrere Agenten-Plugins parallel zu verwenden:

config – Dies ist ein Dictionary, das die Namen der in der Konfigurationsdatei angegebenen Plugins als Schlüssel und die zugehörigen `Configuration`-Objekte als Werte enthält.

safe_popen(), safe_pclose() – Diese beiden Funktionen sind Wrapper um die zum Öffnen und Schließen von Dateien verwendeten Methoden `os.popen()` und `<file>.close()`. Da `os.popen()` bei gleichzeitigem Zugriff aus mehreren Threads problemtaisch sein kann, werden hier Semaphoren verwendet, um den Zugriff zu regeln und Threadsicherheit herzustellen.

loadIdmefMessageId(), saveIdmefMessageId(), getIdmefMessageId() – Diese Funktionen dienen dem Zweck, durch Nutzung von Semaphoren threadsicher fortlaufende und zwischenspeicherbare IDs zu erzeugen, die zur eindeutigen Zuordnung in IDMEF-Nachrichten verwendet werden müssen.

Andererseits befindet sich dort, was zum Erstellen von Agenten-Plugins erforderlich ist:

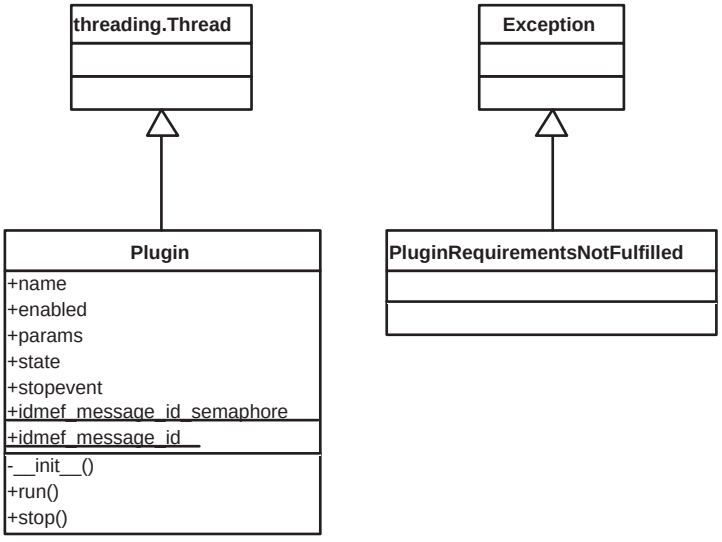

Abbildung 5.2: Klassendiagramm der Hauptkomponente Plugin.py

Plugin – Den Kern eines Agenten-Plugins bildet die Klasse `Plugin`. Von ihr muss ein Plugin abgeleitet sein. Die Klasse `Plugin` selbst ist von der Klasse `Thread` des in der Standardbibliothek von Python enthaltenen Moduls `threading` abgeleitet und bewirkt, dass ein Plugin selbst auch ein Thread ist.

PluginRequirementsNotFulfilled – Diese Exception kann beim Laden eines Plugin-Moduls geworfen werden. Sie signalisiert, dass das Plugin unter den Umständen der Umgebung, in der es geladen wurde, höchstwahrscheinlich nicht funktionieren wird. Die Ursache kann ein fehlendes Paket oder Modul sein, das z.B. zur Standardbibliothek gehört, aber nicht auf dem aktuellen Betriebssystem nutzbar ist, oder das zunächst von einem Drittanbieter beschafft und installiert werden muss. Möglich ist auch, dass die Plattform nicht die Voraussetzungen des Plugins erfüllt, etwa weil unter Windows keine UNIX-spezifischen Konfigurationsdateien vorhanden sind.

5.3.1.3 PluginManagement.py

Im Modul `PluginManagement` befinden sich Hilfsmittel für den Umgang mit mehreren Agenten-Plugins.

getAvailablePlugins() – Diese Methode liefert eine Liste der Namen aller Plugins, die in der Konfigurationsdatei eingetragen sind.

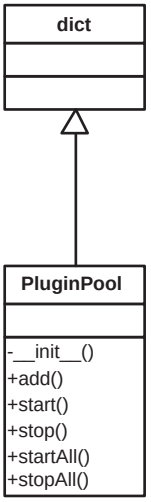

Abbildung 5.3: Klassendiagramm der Hauptkomponente PluginManagement.py

PluginPool – Der `PluginPool` ist eine Unterklasse eines Dictionary und enthält Instanzen der in der Konfigurationsdatei angegebenen Plugins. Die bereitgestellten Methoden erlauben das Importieren und Instanziiern von Plugins durch Angabe ihres Namens sowie das Starten und Stoppen einzelner oder aller im `PluginPool` eingetragener Plugins.

5.3.2 Plugins

5.3.2.1 Snort

Wie der Name vom Plugin Snort bereits vermuten lässt, ist Snort ein Plugin, das Informationen von dem IDS *Snort* sammelt. Das IDS *Snort* ist ein netzwerkbasiertes Einbruchserkennungsystem (NIDS), das den Datenverkehr auf einem bestimmten Netzsegment überwacht. Ein derartiges NIDS ist also in der Lage, Angriffe auf beliebige Hosts in dem zu überwachenden Netzwerk zu erkennen oder auch vom entsprechenden Host ausgehende Angriffe auf andere Netzwerke bzw. Hosts festzustellen. Die Daten werden dabei automatisch von einem Netzwerk-Sniffer aufgenommen und von einem Analysetool weiter verarbeitet, um sie auf eventuelle Angriffe zu analysieren. Das netzwerkbasierte IDS Snort arbeitet dabei mit einer Echtzeit-Missbrauchserkennung, d.h. seine Arbeit besteht im Wesentlichen darin, die gesammelten Daten auf bekannte Signaturen oder Angriffsmustern, die in einer Datenbank gespeichert sind, hin zu untersuchen. Die Erstellung solcher Signaturen kann entweder selbst vorgenommen werden oder es kann auf eine umfangreiche Basis verfügbarer Signaturen zurückgegriffen werden. Die neuesten Signaturen und Updates sind natürlich über die Webseite von *Snort* (www.snort.org) verfügbar.

Um auf die Informationen vom IDS *Snort* zu zugreifen, kommuniziert das Plugin Snort mit einem speziellen Output-Plugin von *Snort*, dem *IDMEF-Output-Plugin*. Das *IDMEF-Output-Plugin* wurde von *Silicon Defense* (www.silicondefense.com/idwg/snort-idmef/) geschrieben. Es gibt die Protokollmeldungen im IDMEF (*Intrusion Detection Message Exchange Format*) aus. Als Ausgabemöglichkeit steht hierzu entweder ein TCP-Socket oder eine Logdatei zur Verfügung. Bevor nun *IDMEF-Output-Plugin* genutzt werden kann, muss es aktiviert werden. Hierzu wird *output idmef* gefolgt von einer Argumentenliste in die Konfigurationsdatei `snort.conf`

hinzugefügt. Anschließend müssen die Regeln entsprechend angepasst werden. Regeln, die nun dieses *IDMEF-Output-Plugin* nutzen sollen, benötigen ein zusätzliches Attribut *idmef*. Nähere Informationen hierzu sind in der Dokumentation über das *IDMEF-Output-Plugin* zu finden.

Die Beispielkonfiguration

```
output idmef: $HOME_NET
              output=log
              dtd=idmef-message.dtd
              analyzer_id=Sensor1
              category=dns
              location=Oldenburg
              facility_default=file|/var/log/snort/idmef_alert
```

und die Regel

```
alert tcp any any -> any 80
(msg:"Directory Traversal"; flags: AP; content: "../";idmef: web;)
```

erzeugen eine folgende IDMEF-Meldung:

```
<IDMEF-Message version="0.1">
  <Alert alertid="329440" impact="unknown" version="1">
    <Time>
      <ntpstamp>0x3a2da04c.0x0</ntpstamp>
      <date>2002-05-11</date>
      <time>16:41:30</time>
    </Time>
    <Analyzer ident="Sensor1">
      <Node category="dns">
        <location>Odenburg</location>
      </Node>
    </Analyzer>
    <Classification origin="vendor-specific">
      <name>Directory Traversal</name>
    </Classification>
    <Source spoofed="unknown">
      <Node>
        <Address category="ipv4-addr">
          <address>3.3.3.3</address>
        </Address>
      </Node>
    </Source>
    <Target decoy="unknown">
      <Node category="dns">
        <location>Steinfurt</location>
```

```
        <Address category="ipv4-addr">
          <address>192.168.111.50</address>
        </Address>
      </Node>
      <Service ident="0">
        <dport>80</dport>
        <sport>1397</sport>
      </Service>
    </Target>
    <AdditionalData meaning="Packet Payload" type="string">
      GET ../../etc/passwd
    </AdditionalData>
  </Alert>
</IDMEF-Message>
```

Für den Zugriff auf die generierten IDMEF-Meldungen des *IDMEF-Output-Plugin* stehen dem Plugin Snort zwei Möglichkeiten zu Verfügung. Jede der beiden Möglichkeiten ist dabei in einer eigenen Methode implementiert:

1. `useInputFacilityTcp()` für den Zugriff über eine Socket-Verbindung: Das Plugin Snort baut zunächst eine clientseitige TCP-Socket-Verbindung zum *IDMEF-Output-Plugin* auf. Nach einem erfolgreichen Verbindungsaufbau wartet das Plugin auf eingehenden IDMEF-Meldungen, die über die aufgebaute TCP-Socket-Verbindung empfangen werden können. Die empfangenen IDMEF-Meldungen werden einer kleinen Modifikation unterzogen, bevor sie über den logischen BEEP-Kanal *IDXP* an die Mittelschicht gesendet werden. Die Modifikation der empfangenen IDMEF-Nachricht besteht im Wesentlichen darin, den Wert des Attributes *analyzerid* auf die ID des aktuellen Agenten zu setzen.

2. `useInputFacilityFile()` für den Zugriff über Logdatei: Das Plugin Snort liest in einem Polling-Verfahren IDMEF-Meldungen aus einer Logdatei, die als Ausgabemöglichkeit für das *IDMEF-Output-Plugin* spezifiziert wurde. In den ausgelesenen IDMEF-Meldungen wird ebenfalls der Wert des Attributes *analyzerid* auf die ID des aktuellen Agenten gesetzt, bevor sie über das BEEP-Kanal IDXP an die Mittelschicht übergeben werden.

Die Methode `useInputFacilityTcp()` bzw. `useInputFacilityFile()` führt ihre Operationen zum Auslesen von Informationen in einer Endlosschleife aus und repräsentiert sozusagen die Aktivität des Plugin-Thread. Die Endlosschleife wird verlassen, wenn entweder das Plugin-Snort deaktiviert wird oder ein Ausnahmefehler im Plugin Snort auftritt. Das Verlassen der Endlosschleife bewirkt, dass der Plugin-Thread terminiert wird. Damit jederzeit eine Thread-Terminierung im Falle der Socket-Verbindung möglich ist, befinden sich alle Funktionen der Socket-Schnittstelle im nicht-blockierenden Zustand.

In der Konfigurationsdatei `agent.xconf` kann eine der beiden obigen Möglichkeiten für das Plugin Snort definiert werden. Hier ein Ausschnitt aus `agent.xconf` mit einer Beispielkonfiguration für die 1. Möglichkeit:

```
...
<plugin name="Snort" enabled="1">
  <parameter key="input_facility" value="tcp"/>
  <parameter key="host" value="localhost"/>
  <parameter key="port" value="50001"/>
</plugin>
...
```

und einer Beispielkonfiguration für die 2. Möglichkeit:

```
...
<plugin name="Snort" enabled="1">
   <parameter key="input_facility" value="file"/>
   <parameter key="filename" value="/tmp/idmef_messages"/>
</plugin>
...
```

Für den Zugriff über die Socket-Verbindung war eine Anpassung in den C-Sourcen vom *IDMEF-Output-Plugin* notwendig. Die Anpassung verhindert, dass das IDS *Snort* terminiert, wenn *IDMEF-Output-Plugin* versucht eine IDMEF-Meldung über eine Socket-Schnittelle zu senden, die keine Verbindung zum angegeben TCP-Client (Plugin Snort) herstellen konnte. Die vorgenommene Modifikation betrifft die Funktionen `OpenFacilityTcp()` und `OutputMessageTcp()`, die sich in der Quelltextdatei `spo_idmef.c` befinden:

```
void OpenFacilityTcp(IDMEFOutputFacility* facility)
{
  struct hostent* he;
  int flags;

#ifdef IDMEF_DEBUG
  fprintf(stderr, "IDMEF::OpenFacilityTcp called\n");
#endif

  if (NULL == facility || NULL == facility->config)
  {
    FatalError("IDMEF: cannot open a NULL or unconfigured \
      facility");
  }

  if (tcp_conn != facility->type)
  {
    FatalError("IDMEF: OpenFacilityTcp was passed a \
      non-tcp facility.\n");
  }

  facility->state->tcp_state.sock_dest->sin_family = PF_INET;
  facility->state->tcp_state.sock_dest->sin_port =
```

```
  htons (facility->config->tcp_config.port_num);

  he = gethostbyname(facility->config->tcp_config.remote_host)
  if (he == NULL)
  {
    FatalError("Unknown host %s.\n",
               (facility->config->tcp_config.remote_host));
  }

  facility->state->tcp_state.sock_dest->sin_addr =
    *((struct in_addr *)he->h_addr);
  facility->state->tcp_state.tcp_socket =
    socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

  if (facility->state->tcp_state.tcp_socket == -1)
    FatalError("log_idmef: IDMEF socket open failure (%s:%i)\n",
               facility->config->tcp_config.remote_host,
               facility->config->tcp_config.port_num );

  // ########################################################
  // MK 01.06.05:
  // Modification for the TCP socket communication with the
  // Plugin Snort.

  if (connect(facility->state->tcp_state.tcp_socket,
              (struct sockaddr *)facility->state->tcp_state.sock_dest,
               sizeof(struct sockaddr)) == -1)
  {
    // Close the current TCP socket and try to reconnect
    // on next message.
    close(facility->state->tcp_state.tcp_socket);
    facility->state->tcp_state.tcp_socket = 0;
    printf("log_idmef: IDMEF socket connect failure (%s:%i):
            errno: %d, %s\n",
            facility->config->tcp_config.remote_host,
            facility->config->tcp_config.port_num,
            errno, strerror(errno));
  }

  // Set the socket to nonblocking for the socket connection.
  flags = fcntl(facility->state->tcp_state.tcp_socket, F_GETFL);
  fcntl(facility->state->tcp_state.tcp_socket,
        F_SETFL, flags|O_NONBLOCK);
  // ########################################################
}



void OutputMessageTcp(xmlDocPtr document,
```

```
                        IDMEFOutputFacility* facility)
{
  char* msg;
  int length;
  int rc;
  char dummy_buf[1024];

#ifdef IDMEF_DEBUG
  fprintf(stderr, "IDMEF::OutputMessageTcp called\n");
#endif

  if(NULL == facility ||
     (NULL == facility->state && NULL == facility->config))
  {
    FatalError("OutputMessageTcp: cannot output \
      messages on a NULL facility\n");
  }

  // ###########################################################
  // MK 01.06.05:
  // Modification for the TCP socket communication with
  // the Plugin Snort.

  if(facility->state->tcp_state.tcp_socket == 0)
  {
    // This is the first time this facilty has been used or
    // reconnection was initiated due to the closed socket.
    OpenFacilityTcp(facility);
  }

  // Detect closed socket and allow reconnection to the peer entity.
  if(facility->state->tcp_state.tcp_socket != 0)
  {
    // Check if the TCP socket is still open.
    rc = read(facility->state->tcp_state.tcp_socket,
              dummy_buf,
              sizeof(dummy_buf));
    if (rc > 0)
    {
      FatalError("OutputMessageTcp: unexpected message received \
        from the peer entity (%s:%i)\n",
      facility->config->tcp_config.remote_host,
      facility->config->tcp_config.port_num);
    }
    else if (rc == 0)
    {
      close(facility->state->tcp_state.tcp_socket);
      printf("Socket was closed by the peer entity (%s:%i)!\n",
      facility->config->tcp_config.remote_host,
```

```
      facility->config->tcp_config.port_num);

      // Allow to reconnect to the peer entity before sending
      // the next message.
      facility->state = NULL;
    }
    else /* rc < 0 */
    {
      switch(errno)
      {
        case EWOULDBLOCK:
          // Send the IDMEF message.
          msg = (char*)calloc(MAX_IDMEF_MESSAGE_LENGTH, sizeof(char))
          if (msg != NULL)
          {
            length = IdmefMessageToStr(document, &msg);
            write(facility->state->tcp_state.tcp_socket, msg, length);
            free (msg);
          }
          else
          {
            FatalError("OutputMessageTcp: calloc failed!\n");
          }
          break;
        default:
          FatalError("Socket Error: %d, %s\n", errno, strerror(errno));
          break;
      }
    }
  }
  // ########################################################
}
```

5.3.2.2 Syslog

Das Plugin Syslog ermöglicht einen Zugriff auf die Protokollmeldungen des Protokolldienstes *Syslog*. Aus den Protokollmeldungen generiert das Plugin entsprechende IDMEF-Meldungen, die dann an die Mittelschicht gesendet werden.

Der Protokolldienst *Syslog* ist unter UNIX/Linux ein sehr wichtiger Daemon für Diagnosezwecke. Dieser Dienst erhält seine Meldungen über einen Socket (`/dev/log`). Alle klassischen UNIX/Linux-Prozesse, die eine Protokollierung wünschen, senden ihre Meldungen an den *Syslog*-Daemon. Der *Syslog*-Daemon analysiert die empfangenen Meldungen und protokolliert sie in Abhängigkeit der Quelle (*facility*) und der Priorität (*priority*) an unterschiedlichen Orten. Die Konfiguration des Syslog-Daemon erfolgt in der Datei `/etc/syslog.conf`.

Auf die Protokollmeldungen von *Syslog*-Daemon greift das Plugin Syslog entweder über eine Logdatei oder eine sog. Named Pipe zu, in die der *Syslog*-Daemon seine Meldungen schreiben kann. Für jede der beiden Zugriffsmöglichkeiten gibt es im Plugin Syslog eine eigene Methode:

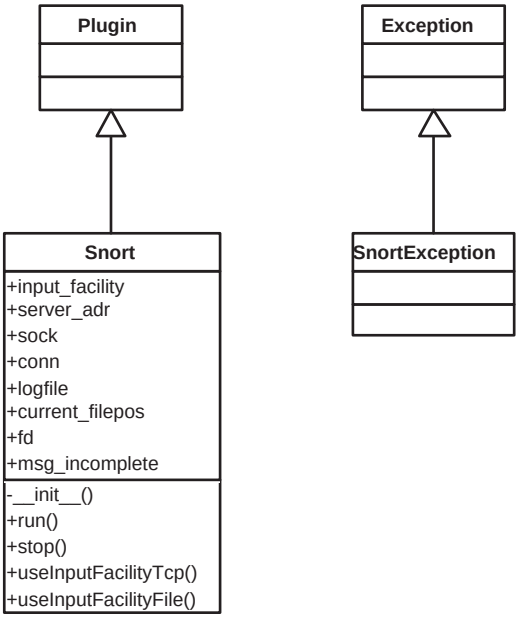

Abbildung 5.4: Klassendiagramm des Plugin Snort

1. `useInputFacilityPipe()` für den Zugriff über eine Named Pipe: Das Plugin Syslog öffnet zunächst eine Named Pipe, in die der *Syslog*-Daemon je nach Konfiguration unterschiedliche Protokollmeldungen schreiben kann. In einer Endlosschleife prüft das Plugin die Pipe auf eingegangene Meldungen. Aus den empfangenen Meldungen, die mit einem Zeitstempel, einem Rechnernamen sowie einem Prozessnamen versehen sind, generiert das Plugin entsprechende IDMEF-Nachrichten. Die IDMEF-Nachrichten sendet das Plugin über den logischen Kanal *IDXP* an die Mittelschicht.

2. `useInputFacilityFile()` für den Zugriff über Logdatei: Das Plugin Syslog liest in einem Polling-Verfahren Protokollmeldungen aus einer *Syslog*-Protokolldatei und erzeugt aus den eingelesenen *Syslog*-Meldungen entsprechende IDMEF-Nachrichten. Die IDMEF-Nachrichten werden dann vom Plugin über den *IDXP*-Kanal an die Mittelschicht gesendet.

Um an die Protokollmeldungen vom *Syslog*-Daemon zu gelangen, kann also das Plugin Syslog über die Konfigurationsdatei `agent.xconf` so konfiguriert werden, dass es entweder den Inhalt einer Logdatei oder einer Named Pipe ausliest. Gemäß den Einstellungen in `agent.xconf` muss die Konfigurationsdatei des *Syslog*-Daemon passende Einträge enthalten, d.h. der Name der Logdatei bzw. der Named Pipe muss in beiden Konfigurationsdateien derselbe sein.

5.3.2.3 SystemStat

Auf UNIX-Betriebssystemen sammelt das Plugin allgemeine Systeminformationen zur Prozessorlast, Belegung von Arbeits- und Auslagerungsspeicher, Uptime und ähnlichem. Dazu werden Dateien des virtuellen proc-Dateissytems ausgelesen. Dies sind:

- /proc/meminfo - Statistiken über die Speichernutzung

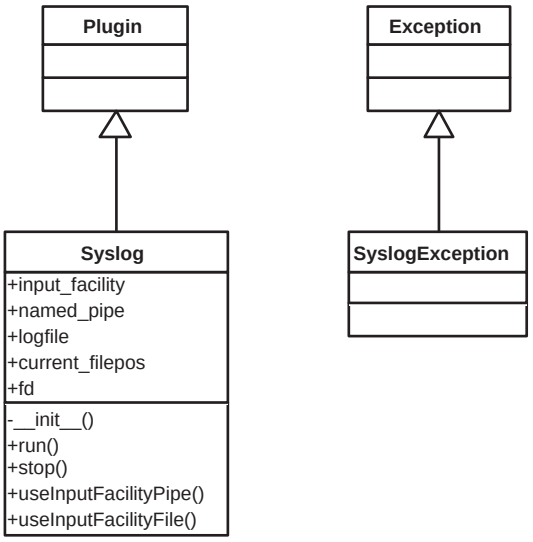

Abbildung 5.5: Klassendiagramm des Plugin Syslog

- /proc/loadavg - Systemauslastung
- /proc/stat - Diverse Systeminformationen
- /proc/uptime - Uptime und Idle-Uptime

Da diese Dateien die Werte im Klartext und durch Leerzeichen getrennt enthalten, lassen sie sich durch Pythons Qualitäten in der Stringverarbeitung sehr komfortabel verarbeiten.

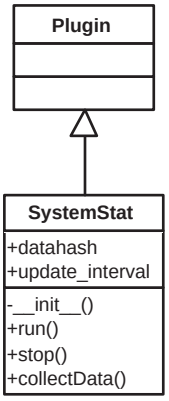

Abbildung 5.6: Klassendiagramm des Plugin SystemStat

5.3.2.4 NetworkStat

Ganz ähnlich dem Plugin SystemStat verfährt das Plugin um an Informationen über den Status der Netzwerklast zu gelangen. Dazu bedient es sich folgender Quellen:

- /proc/net/dev - Informationen über die Netzwerkgeräte
- /proc/net/snmp - Netzwerkmanagement-Informationen

- *netstat* - Über dieses Kommando lassen sich die aktuellen TCP- und UDP-Verbindungen ermittelt. Die Ausgabe wird ähnlich wie die proc-Dateien geparst.

Momentan wird jedoch nur aus der ersten Quelle gelesen. Die Funktionalität zum Erfassen von Daten aus den anderen beiden Quellen ist zwar implementiert, in der GUI aber nicht von Nutzen und wurde daher vorläufig aus Performancegründen deaktiviert.

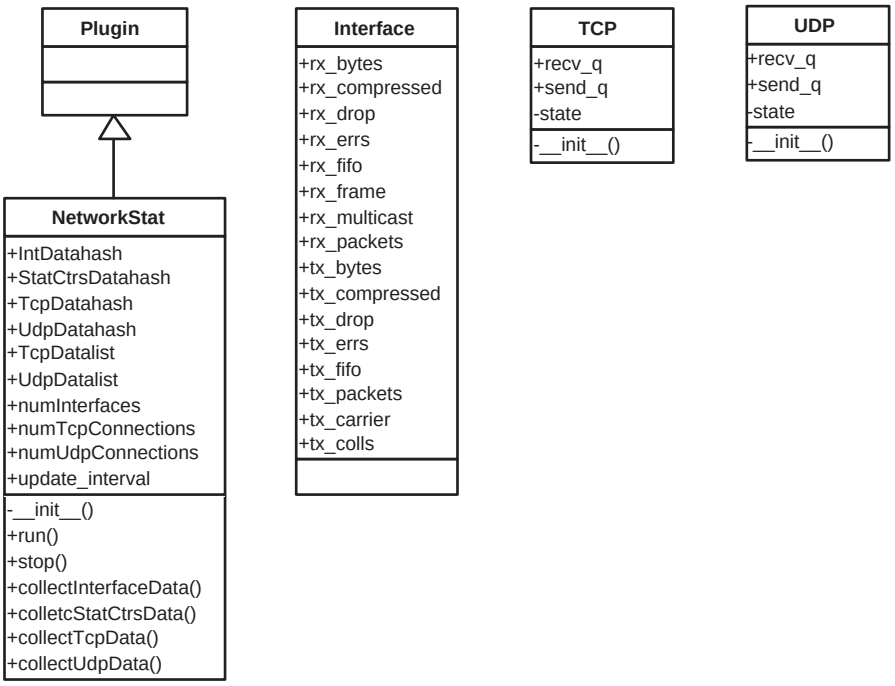

Abbildung 5.7: Klassendiagramm des Plugin NetworkStat

5.3.2.5 DiskUsage

Die Aufgabe des Plugins besteht darin, Auskunft über die in den Verzeichnisbaum eines Rechners eingehängten Laufwerke und deren Belegungsdichte zu geben.

Die Implementierung basiert entfernt auf der Software `pydf` (zu beziehen auf http://melkor.dnp.fmph.uniba.sk/~garabik/pydf/) von Radovan Garabik. Es handelt sich dabei um eine als Public Domain veröffentlichte Nachbildung des UNIX-Kommandos `du` („disk usage“) in Python. Daraus wurde der für dieses Plugin stark überarbeitete und ergänzte Teil als Grundlage verwendet, der sich um das Extrahieren von Daten über Mountpoints aus den UNIX-Konfigurationsdateien */etc/mtab*, */etc/mnttab* und */proc/mounts* kümmert. Sind diese Dateien nicht vorhanden, so wird versucht, die Ausgabe des Kommandos `mount` zu verarbeiten. Damit funktioniert dies auch unter Mac OS X, wo die genannten Konfigurationsdateien nicht existieren. Unter Windows-Betriebssystemen funktioniert das Auslesen der gewünschten Informationen über diesen Weg allerdings nicht, da es Mountpoints in dieser Form dort nicht gibt.

Die gesammelten Daten werden in einer Statusnachricht über den Datenkanal zur Mittelschicht gesendet. Zu jedem eingehängten Mountpoint wird ein `DataSet` erstellt. Als Unterelemente in Form von `DataElement` werden das Gerät (device), der

Ort im Verzeichnisbaum (mountpoint), der Typ des Dateisystems (type), der belegte (usedSpace) und der gesamte Speicherplatz (totalSpace) des Datenträgers eingefügt. Die letzten beiden Werte reichen aus, um daraus weitere Werte wie etwa den noch freien Speicherplatz oder die prozentuale Belegung zu errechnen, wenn dies für die Darstellung gewünscht ist.

Eine solche Nachricht kann beispielsweise so aussehen:

```
<status plugin="DiskUsage">
  <dataset key="mount" value="/dev/hda1">
    <data key="device" value="/dev/hda1"/>
    <data key="mountpoint" value="/"/>
    <data key="type" value="ext3"/>
    <data key="usedSpace" value="5767168"/>
    <data key="totalSpace" value="8388608"/>
  </dataset>
  <dataset key="mount" value="/dev/hda5">
    <data key="device" value="/dev/hda5"/>
    <data key="mountpoint" value="/home"/>
    <data key="type" value="reiserfs"/>
    <data key="usedSpace" value="24117248"/>
    <data key="totalSpace" value="52428800"/>
  </dataset>
</status>
```

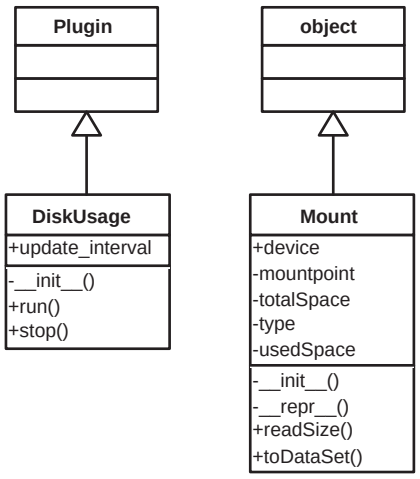

Abbildung 5.8: Klassendiagramm des Plugin DiskUsage

5.3.2.6 NetworkInterfaces

Das Plugin liefert Details über die Netzwerkschnittstellen eines Geräts. Es verwendet dazu die für viele Plattformen verfügbare Bibliothek `libdnet` (http://libdnet.sourceforge.net/) von Dug Song. Diese stellt neben anderem einen Iterator über die verfügbaren Netzwerkschnittstellen sowie deren Attribute zur Verfügung.

Als Statusnachricht über den Datenkanal wird pro Interface ein `DataSet` geschickt, dass je ein `DataElement` mit dem Namen, der IP-Adresse, der Subnetzmaske, der

MAC-Adresse, dem MTU(Maximum Transfer Unit)-Wert, einem Loopback- und einem Aktiviert-Flag dieses Interfaces, sofern diese Werte vorhanden sind, enthält.

Eine Beispielnachricht eines Rechners mit einem Loopback- und einem Ethernet-Interface sieht so aus:

```
<status plugin="NetworkInterfaces">
  <dataset key="interface" value="0">
    <data key="name" value="lo0"/>
    <data key="ip_address" value="127.0.0.1"/>
    <data key="netmask" value="8"/>
    <data key="mtu" value="1520"/>
    <data key="loopback" value="1"/>
    <data key="up" value="1"/>
  </dataset>
  <dataset key="interface" value="1">
    <data key="name" value="eth0"/>
    <data key="ip_address" value="192.168.0.1"/>
    <data key="netmask" value="24"/>
    <data key="mac_address" value="00:33:66:99:CC:FF"/>
    <data key="mtu" value="1500"/>
    <data key="loopback" value="0"/>
    <data key="up" value="1"/>
  </dataset>
</status>
```

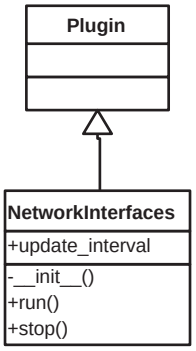

Abbildung 5.9: Klassendiagramm des Plugin NetworkInterfaces

5.4 Probleme

In diesem Abschnitt wird kurz auf die Probleme eingegangen, die während der Programmerstellung des Agenten aufgetreten sind bzw. die beim Testen des gesamten unIDS-Systems entdeckt werden konnten. Die aufgetretenen Probleme sind auf die benutzten Bibliotheken `BEEPy` und `Twisted` zurückzuführen. Sie treten im Speziellen bei der `BEEPy`-Methode `sendFrame()` auf, die indirekt vom Agenten zur Aussendung von BEEP-Nachrichten verwendet wird. Die `sendFrame()`-Methode funktioniert nur unzureichend und unzuverlässig, d.h. während einige BEEP-Nachrichten problemlos gesendet werden können, werden andere mit einer nicht unerheblichen Verzögerung

von mehreren Sekunden verschickt oder sogar verworfen. Offensichtlich werden einige Nachrichten vor der tatsächlichen Aussendung über einen BEEP-Kanal zunächst vom BEEPy-Framework zwischengespeichert. Des Weiteren werden die verzögerten Nachrichten auf einmal gesendet, so dass es in der Mittelschicht und in der GUI zu Folgefehlern in der Nachrichtenbehandlung führt. In den durchgeführten Tests konnte auch beobachtet werden, dass je länger die auszusendenden Nachrichten sind, desto geringer die Zuverlässigkeit ihrer Aussendung ist.

Um den Fehler zu beheben, wurde in dem BEEPy-Modul `tcp.py` die Methode `sendFrame()` etwas modifiziert (*Bugfix*):

```
def sendFrame(self, theframe):
  """
  sendFrame is used to push frames over the transport.

  With the addition of SEQ frames, this becomes a little
  more complex. We need to check that the amount of data
  we're about to send isn't larger than the allocated
  window size. If it is, then we fragment the data and
  only send bytes up to the window size. We then have to
  wait for a SEQ frame from the remote peer saying that
  it has room for more data before sending the rest.

  The sending of pending data happens asynchronously
  via the processQueuedData() method.
  """
  try:
    data = str(theframe)

    ##########################################################
    # MK 01.10.05: Bugfix for the unIDS-Project
    # Do not use the twsited method write() to send data.
    # Instead use the socket function send() to send the
    # date immediately.
    try:
      self.transport.socket.send(data)
    except:
      pass

    #result = self.transport.write(data)

    ##########################################################

  except Exception, e:
   log.debug('Exception sending frame: %s: %s' % (e.__class__, e))
   traceback.print_exc()
```

Wie aus dem obigen Listing zu entnehmen ist, wurde bei dem *Bugfix* der Aufruf der Twisted-Methode `write()`, die letztendlich die eigentliche problematische Methode

ist, durch die Socket-Funktion `send()` ersetzt. Im Gegensatz zu `write()` garantiert `send()` eine sofortige Aussendung von anstehenden Daten.

5.5 Fazit

Die in Abschnitt 5.1 gesetzten Ziele und Anforderungen konnten in der Software des Agenten erfolgreich umgesetzt werden. In den durchgeführten Tests haben der Agent und die jeweiligen Plugins bereits gezeigt, dass sie in der Lage sind, sowohl Informationen von dem IDS *Snort* zu sammeln als auch zahlreiche sicherheitsrelevante Informationen über den zu überwachenden Rechner selbst zu generieren. Nachdem das in Abschnitt 5.4 beschriebene Problem bzgl. der Aussendung von BEEP-Nachrichten gelöst werden konnte, läßt auch der Einsatz von `BEEPy` und `Twisted` die Schlussfolgerung zu, dass die Kommunikationsschnittstelle zur Mittelschicht sehr effektiv und zuverlässig funktioniert. Die BEEP-Nachrichten können somit problemlos zwischen der Mittelschicht und dem Agenten übertragen werden. Das Plugin-Konzept erwies ich auch während der Implementierungsphase als sehr vorteilhaft, weil es eine unabhängige Implementierung der einzelnen Plugins durch mehrere Entwickler ermöglichte und somit eine effektivere und schnellere Entwicklung zur Folge hatte.

6. Proxy

Ein Proxy repräsentiert im unIDS-System eine Vermittlungsstelle, die sich strukturell und logisch zwischen der Mittelschicht und den Agenten befindet. Die primäre Aufgabe eines Proxy besteht im Weiterleiten von Nachrichten. Für unIDS-System ist der Einsatz eines Proxy nicht zwingend erforderlich, d.h. der Anwender kann das unIDS-System auch ohne den Proxy benutzen, um ein Computernetzwerk auf Sicherheitsverletzungen hin zu überwachen.

6.1 Zielsetzungen und Anforderungen

Vor dem Entwurf sollen zunächst die Zielsetzungen und die verschiedenen Anforderungen an den Proxy gesammelt und konkretisiert werden.

Im Rahmen der Projektgruppe „Intrusion Detection Management“ ist die Funktionalität des Proxy nur auf das Weiterleiten von Nachrichten beschränkt. Aus Sicht der Mittelschicht verhält sich das Proxy-Programm wie ein Client, den Agenten gegenüber wie ein Server. Die Agenten kommunizieren so bewusst mit dem Proxy und bitten ihn, stellvertretend für sie die Kommunikation mit der Mittelschicht zu übernehmen. Auf diese Weise werden alle Nachrichten, die an die Mittelschicht gehen, automatisch über den Proxy geleitet. Von einem wirklich direkten Zugriff kann hier also nicht gesprochen werden, denn der Agent sieht nicht die Mittelschicht, sondern nur die Adresse des Proxy und kommuniziert so mit dem Proxy.

Der Proxy nimmt also stellvertretend für den Agenten die Verbindung zur Mittelschicht auf. Die Nachrichten an den Agenten schickt die Mittelschicht ebenfalls an den Proxy, welcher die empfangenen Pakete automatisch an den entsprechenden Agenten weiterreicht. Dabei ist der Proxy nur für die Agenten, nicht jedoch für die Mittelschicht sichtbar. So glaubt die Mittelschicht, dass sie direkt mit den Agenten kommuniziert. In der Mittelschicht muss also kein Proxy eingetragen werden, damit die Nachrichten an die entsprechenden Agenten geleitetet werden können.

Neben der grundlegenden Funktionalität bzgl. des Weiterleiten von Nachrichten soll die Implementierung des Proxy Raum für spätere Erweiterungen lassen. Eine Möglichkeit zur Weiterentwicklung des Proxy besteht darin, den Proxy so zu erweitern,

dass eine Analyse und Filterung von Nachrichten ebenfalls in dem Proxy erfolgen kann. Denkbar wäre auch eine Erweiterung des Proxy um Routing-Funktionen, die letzten Endes auch zur größeren Stabilität des Gesamtsystems beitragen würden. Einem Proxy könnten z.B. mehrere übergeordnete Proxies als Umleitung zugeordnet werden, so dass bei Nichterreichbarkeit eines Proxy eine alternative Route für die Nachrichten gefunden werden kann. Damit also diese Funktionalitäten nachträglich in die vorhandene Software eingebaut werden können, ist ein objektorientierter Aufbau des Proxy anzustreben.

6.2 Implementierung

Nachdem im vorhergehenden Kapitel die Zielsetzungen und Anforderungen an den Proxy beleuchtet wurden, befasst sich dieses Kapitel mit der konkreten Umsetzung des Proxy in Python-Code. Dabei werden einige ausgewählte Definitionen und Konzepte der Implementierung beschrieben, sowie einige Eigenheiten, die sich auf die Implementierung der Proxy beziehen, erläutert.

Die in der Programmiersprache Python umgesetzte Implementierung des Proxy befindet sich im Modul `Proxy.py`. Die Realisierung der Kommunikationsschnittstelle, die ein BEEP-Protokoll implementiert, erfolgt dabei mittels der frei verfügbaren Python-Bibliothek `BEEPy`. Wie bei der Entwicklung der Mitteschicht und des Agenten, erwies sich auch bei der Implementierung des Proxy die Bibliothek `BEEPy` als ein effektives und zugleich programmierfreundliches Framework zur Umsetzung des BEEP-Protokolls.

Um die Kompatibilität zu der Mittelschicht und dem Agenten zu gewährleisten, wird die auf `BEEPy` basierte Kommunikation auf die drei logischen Kanäle *Management*, *IDXP* und *Data* verteilt. Die Eigenschaften der drei Kommunikationskanäle werden dabei durch die bereits in der Mittelschicht und dem Agenten eingesetzten Profile (`ManagementProfile.py`, `DataProfile.py`, `IdxpProfile.py`) geregelt.

Gemäß der obigen Anforderungen soll der Proxy in der Lage sein, die Nachrichten zwischen der Mittelschicht und den beteiligten Agenten transparent zu übertragen. Der Proxy muss also, um seine Aufgaben erfüllen zu können, zwei BEEP-Schnittstellen zur Verfügung stellen:

1. Eine BEEP-Schnittstelle mit den drei logischen Kanälen *Management*, *Data*, und *IDXP* zu der Mittelschicht, die faktisch wie ein Client funktioniert und die Aufgabe hat, die Nachrichtenübertragung zwischen der Mittelschicht und dem Proxy zu realisieren.

2. Eine BEEP-Schnittstelle mit den drei logischen Kanälen *Management*, *Data* und *IDXP* zu jedem der beteiligten Agenten, die faktisch wie ein Server funktioniert und die Aufgabe hat, die Nachrichtenübertragung zwischen dem Proxy und den Agenten zu realisieren.

In der folgenden Abbildung ist die interne Nachrichtenbehandlung, die im Proxy realisiert ist, vereinfacht dargestellt.

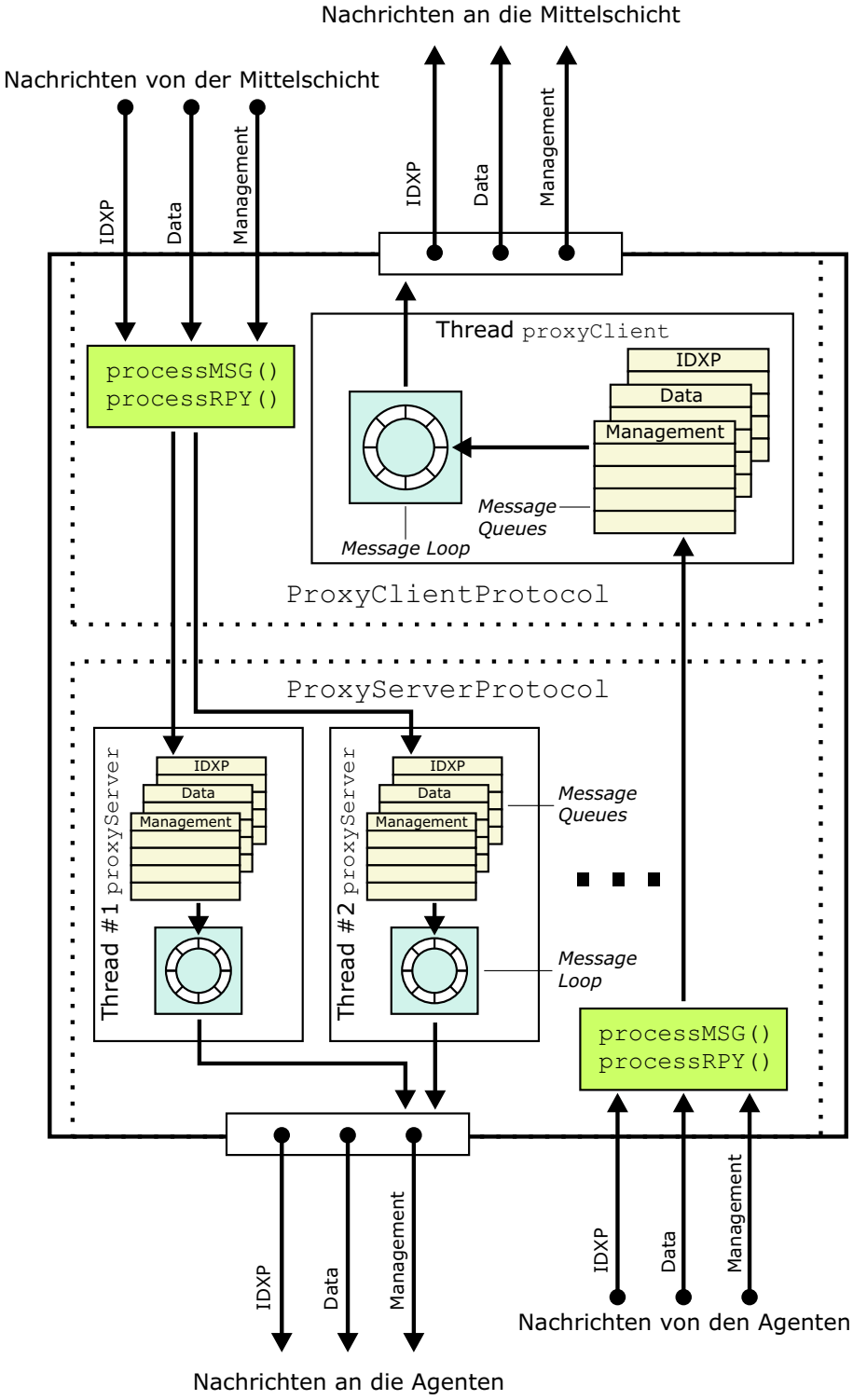

Abbildung 6.1: Vereinfachte Dartellung von der Nachrichtenbehandlung im Proxy.

6.2.1 Clientseitger Teil

Der clientseitige Teil des Proxy-Programms wird in der Klasse `ProxyClientProtocol`, welche von der `BEEPy`-Klasse `BeepClientProtocol` abgeleitet ist, implementiert. `ProxyClientProtocol` definiert sozusagen ein BEEP-Protokoll für die Nachrichtenübertragung zwischen dem Proxy und der Mittelschicht. Sobald eine Verbindung zur Mittelschicht aufgebaut wurde, erzeugt das `BEEPy`-Framework automatisch eine Instanz von der Klasse `ProxyClientProtocol`. Nachdem die Verbindung beendet wurde, wird diese Instanz des Protokolls auch wieder automatisch vom `BEEPy`-Framework entfernt. Die eigentliche Nachrichtenübertragung erfolgt bei einem derartigen ereignisgesteuerten BEEP-Protokoll asynchron, d.h. sobald eine Nachricht von der Mittelschicht beim Proxy eintrifft, wird in `ProxyClientProtocol` die Methode `processMSG()` bzw. `processRPY()` aufgerufen, um die Ankunft einer neuen BEEP-Nachricht vom Typ MSG bzw. RPY zu signalisieren. In `processMSG()` bzw. `processRPY()` erfolgt dann die eigentliche Nachrichtenbehandlung, die weiter unten in diesem Kapitel beschrieben ist.

Für den übrigen Teil des clientseitigen Protokolls, der für den Auf- und Abbau der Verbindung zur Mittelschicht zuständig ist, werden in der Klasse `ProxyClientProtocol` die Methoden `greetingReceived()`, `channelStarted()` und `connectionLost()`. überschrieben (siehe Dokumentation zu `BEEPy`). Die Methode `greetingReceived()` wird vom `BEEPy`-Framework automatisch aufgerufen, sobald eine Verbindung zur Mittelschicht hergestellt werden konnte. Innerhalb der Methode `greetingReceived()` erfolgt dann der Aufbau der drei logischen Kanäle zur Mittelschicht, und zwar für jeden Kanal durch den expliziten Aufruf der `BEEPy`-Methode `newChannel()`. Das `BEEPy`-Framework signalisiert jeden erfolgreichen Kanalaufbau mit einem automatischen Aufruf von `channelStarted()`. Erst nachdem alle drei Kanäle erfolgreich aufgebaut wurden, wird für den clientseitigen Teil des Proxy ein Thread-Objekt (*proxyClient*) erzeugt, der die Aufgabe hat, die von den Agenten eingegangenen Nachrichten an die Mittelschicht weiterzuleiten. Das Thread-Objekt *proxyClient* ist eine Instanz der Klasse `Peer`, die wiederum von der Python-Klasse `Thread` abgeleitet ist. In der Klasse `Peer` sind Attribute und Methoden für die Realisierung der Nachrichtenbehandlung gekapselt. Dabei gibt es für jeden der drei logischen Kanäle eine nach dem FIFO-Prinzip aufgebaute Nachrichtenwarteschlange (*Message Queue*), die in der Liste *tx_channelbuf* verwaltet wird. Die eigentliche Speicherung der Nachrichten für die Mittelschicht geschieht in den Methoden `processMSG()` und `processRPY()`, die in dem serverseitigen Proxy-Protokoll (`ProxyServerProtocol`) implementiert sind. In der Methode `run()`, die vom Thread-Objekt *proxyClient* ausgeführt wird, ist eine sog. Nachrichtenschleife (*Message Loop*) realisiert. Die Nachrichtenschleife ruft die bereits eingereihten Nachrichten aus den Nachrichtenwarteschlangen (*Message Queues*) ab und sendet diese über den entsprechenden logischen Kanal weiter an die Mittelschicht. Der Thread von *proxyClient* wird terminiert, wenn die Methode `connectionLost()` vom `BEEPy`-Framework aufgrund eines Verbindungsabbruches seitens der Mittelschicht aufgerufen oder die Proxy-Applikation beendet wurde.

6.2.2 Serverseitger Teil

Während in dem clientseitigen Teil des Proxy ein einziger Thread für die Nachritenübertragung zwischen der Mittelschicht und dem Proxy zuständig ist, sind in

dem serverseitigen Teil des Proxy mehrere Thread-Objekte der Klasse `Peer` für die Kommunikation mit den beteiligten Agenten notwendig. Für jede neue Verbindung zu einem Agenten wird also eine unabhängige Thread-Instanz erzeugt, deren dann die einzelnen an die Agenten gerichteten Nachrichten zugeteilt werden (*Thread Pool*). Die Verwendung einer solchen Multithreaded-Lösung hat gegenüber einer Mehrprozess-Lösung den Vorteil, dass die Thread-Objekte im Vergleich zu ausgewachsenen Prozessen wesentlich leichtgewichtigere Objekte sind und somit der Overhead zum Erzeugen und Terminieren von vielen Threads nicht so drastisch ausfällt wie bei Prozessen. Für die Verwaltung der einzelnen Thread-Objekte bzw. Instanzen gibt es zwei Datenstrukturen *proxyServer_anonymous* und *proxyServer*. *proxyServer_anonymous* ist eine Liste, in der die sog. „anonymen“ Thread-Objekte entsprechend dem FIFO-Prinzip verwaltet werden. Ein „anonymes“ Thread-Objekt hat keine eindeutige ID und wird einem Agenten zugeordnet, der gleichermaßen über keine eindeutige ID verfügt. Besitzt ein Agent dagegen bereits eine ID, kann diese aus dem XML-Umschlag (*Message Envelope*) der empfangenen Nachrichten ausgelesen und dem entsprechenden Thread-Objekt zugewiesen werden. Thread-Objekte mit einer zugewiesenen ID, anhand derer letzten Endes die Kommunikation zum entsprechenden Agenten referenziert wird, sind in dem Dictionary *proxyServer* gespeichert. Bekommt also ein Agent während der Anmeldephase eine eindeutige ID von der Mittelschicht zugewiesen, kann diese auch vom Client des Proxy ausgelesen und dazu benutzt werden, das bisher noch „anonyme“ Thread-Objekt aus der Liste *proxyServer_anonymous* in das Dictionary *proxyServer* zu verschieben. Erst nachdem nun ein Thread-Objekt über eine eindeutige ID verfügt, wird er mittels der Methode `start()` gestartet. Die Aktivität des Thread-Objektes beschränkt sich dabei auf die Ausführung seiner `run()`-Methode, in der eine Nachrichtenschleife (*Message Loop*) realisiert ist. Wie bei *proxyClient* wird in der Nachrichtenschleife zunächst geprüft, ob neue Nachrichten in *tx_channelbuf* eingereiht sind. Sobald eine Nachricht vorliegt, wird diese aus der entsprechenden *Message Queue* entfernt und über den korrespondierenden Kanal an den jeweiligen Agneten, dem das Thread-Objekt zugeordnet ist, gesendet. Für die Nachrichtenübertragung stehen jedem Thread-Objekt exklusiv die drei logischen Kanäle *Management*, *Data* und *IDXP* zur Verfügung.

Die Umsetzung des Protokolls für den serverseitigen Teil des Proxy ist in der Klasse `ProxyServerProtocol` implementiert und ähnelt der Implementierung von `ProxyClientProtocol`. Im Gegensatz zu `ProxyClientProtocol` haben in `ProxyServerProtocol` die Methoden `greetingReceived()` und `channelStarted()` keinen Einfluss auf den Auf- und Abbau einer Verbindung zum jeweiligen Agenten, da die Verbindung zum Proxy von den Agenten selbst auf- bzw. abgebaut wird. Lediglich die Methode `connectionLost()` dient dazu, das korrespondierende Thread-Objekt im Falle eines Verbindungsabbruchs seitens des Agenten zu beenden und aus dem Dictionary *proxyServer* zu entfernen. Für die Behandlung von Nachrichten gibt es im serverseitigen Teil des Proxy in der Klasse `ProxyServerProtocol` ebenfalls die beiden Methoden `processMSG()` und `processRPY()`. Wenn also eine MSG- bzw. RPY-Nachricht von einem der verbundenen Agenten empfangen wird, ruft das `BEEPy`-Framework die Methode `processMSG()` bzw. `processRPY()` auf, um die Nachrichtenbehandlung im Proxy einzuleiten. Die Nachrichtenbahandlung besteht dabei im Wesentlichen aus dem Speichern der Nachricht in der entsprechenden Nachrichtenwarteschlage (*tx_channelbuf)* des *proxyClient*. Da mehrere Thread-Objekte parallel in die Nachrichtenwarteschlage des *proxy-*

Client schreiben können, ist der Zugriff auf die Nachrichtenwarteschlange über ein Semaphore-Konzept synchronisiert.

6.2.3 Service-Klasse

Neben den Klassen `Peer`, `ProxyClientProtocol` und `ProxyServerProtocol` gibt es die Klasse `ProxyService`, die im Allgemeinen für das Starten und das Beenden der Proxy-Applikation zuständig ist. `ProxyService` ist von der `Twisted`-Klasse `MultiService` abgeleitet und definiert einen sog. Service, dessen Eigenschaften auf dem Service-Konzept von `Twisted` basieren. Beim Start des Proxy sorgt `ProxyService` in der Methode `startService()` dafür, dass zunächst die Profile für die drei logischen BEEP-Kanäle geladen, die Konfigurationsdatei eingelesen und die erforderlichen Startparameter gesetzt werden. Der Proxy benötigt für seine Ausführung folgende Parameter:

- Für den clientseitigen Teil des Proxy:
  Host: IP-Adresse oder Hostname des Rechners, auf dem die Mittelschicht installiert ist.
  Port: Portnummer, über welche die Mittelschicht mit dem Proxy/Agenten kommuniziert.

- Für den serverseitigen Teil des Proxy:
  Port: Portnummer, über welche der Proxy mit den verbundenen Agenten kommuniziert.

Die Parameter sind über das XML-Dokument `proxy.xconf` konfigurierbar und werden innerhalb der Methode `loadConfiguration()` eingelesen. Im nachfolgenden Listing ist eine Bespielkonfiguration für den Proxy zu sehen:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <proxyClient host="localhost" port="2005"/>
    <proxyServer port="2004"/>
</configuration>
```

Nachdem also die erforderlichen Parameter gesetzt wurden, wird mittels der `Twisted`-Klassen `TCPClient`, `TCPServer` und der `Twisted`-Methode `setServiceParent()` jeweils eine Instanz von `ProxyServerProtocol` und `ProxyClientProtocol` initialisiert und aktiviert. Das `BEEPy`-Framework steuert und verwaltet von nun an die Aktivitäten der beiden Proxy-Protokollimplementationen. Wird der Proxy beendet, so ist die `stopService()`-Methode der Klasse `ProxyService` dafür verantwortlich, dass alle laufenden Thread-Objekte terminiert werden, bevor durch den Aufruf von `service.MultiService.stopService(self)` der Proxy selbst beendet wird.

Das folgende Klassendiagramm gibt eine Übersicht über die in der Proxy-Software verwendete Klassenstruktur.

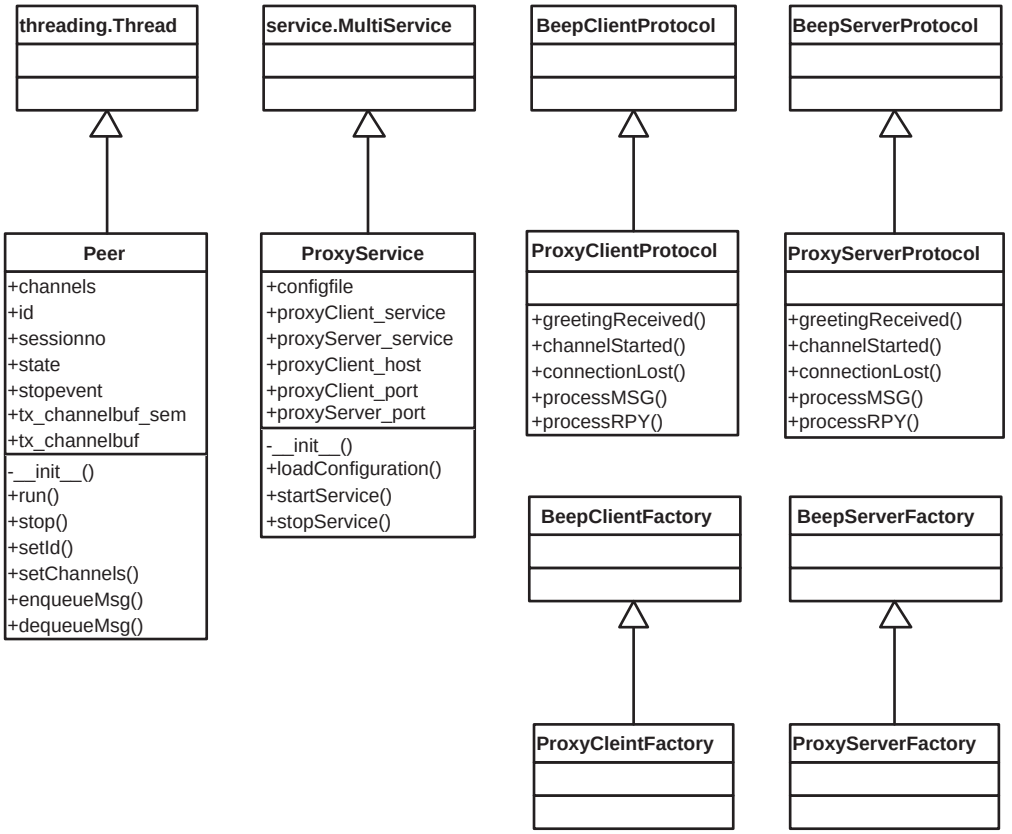

Abbildung 6.2: Klassendiagramm vom Proxy

6.3 Fazit

Die Implementierung des Proxy wurde erfolgreich abgeschlossen. In Tests hat sich die Software des Proxy bereits bewährt. Die Performance ist, soweit es feststellbar war, mehr als akzeptabel. Auch unter hoher Last erfolgt die Nachrichtenübertragung zwischen den im unIDS-System installierten Agenten und der Mittelschicht problemlos. Die hier entworfene Proxy-Architektur und der gewählte Ansatz zum transparenten Weiterleiten mittels des `BEEPy`-Framework haben sich damit als gut geeignet erwiesen, so dass der Proxy im unIDS-System eingesetzt werden kann. Durch den objektorientierten und übersichtlichen Aufbau des Proxy können zusätzliche Funktionalitäten, wie z.B. das Filtern von Nachrichten, problemlos nachträglich in die vorhandene Software eingebaut werden.

Abbildungsverzeichnis

Glossar

Anwendungsfälle – Anwendungsfälle beschreiben Abläufe die beim benutzen der Anwendung auftreten können.

BEEP – Blocks Extensible Exchange Protocol, verbindungsorientiertes, asynchrones Protokoll-Framework der Anwendungsschicht. Es wurde entworfen, um den Aufbau von Netzwerk-Anwendungs-Protokollen zu vereinfachen und zu verbessern. BEEP liefert einen Mechanismus, der verschiedene Problemfelder des Protokoll-Designs, in eine Menge von wiederbenutzbaren Modulen integriert.

Device – Bezeichnet im unIDS-System-Kontext ein Gerüst das vom System überwacht wird

Entitäten – Aus Wikipedia, der freien Enzyklopädie Siehe dazu: http://de.wikipedia.org/wiki/Hauptseite:
Als Entität werden in der Informatik unterscheidbare, in der realen Welt eindeutig identifizierbare einzelne Objekte bezeichnet, über die ein Unternehmen Daten speichert. Die Objekte können sowohl eine physische ('real', z.B. 'Konto Nr. 12345') oder eine konzeptionelle ('abstrakte', z.B. 'Abteilung RK12') Existenz haben.

Framework – Aus Wikipedia, der freien Enzyklopädie
Siehe dazu http://de.wikipedia.org/wiki/Hauptseite/Framework:
Framework ist ein Begriff aus der Softwaretechnik und wird insbesondere im Rahmen der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungs-Ansätzen verwendet.
Wörtlich übersetzt bedeutet Framework (Programm-)Gerüst, -Rahmen oder -Skelett. Darin drückt sich aus, dass ein Framework in der Regel, im Gegensatz zu einer reinen Klassenbibliothek, eine Anwendungsarchitektur vorgibt. Beispielsweise legt ein objektorientiertes Framework die Struktur wesentlicher Klassen und Objekte sowie ein Modell des Kontrollflusses in der Applikation fest. In diesem Sinn werden Frameworks im Wesentlichen mit dem Ziel einer Wiederverwendung architektonischer Muster entwickelt und genutzt.

IDMEF – Intrusion Detection Message Exchange Format, zur Vereinheitlichung der Kommunikation verschiedener IDS.

IDM – Intrusion Detection Management, Schnittstelle, die heterogene Quellen und Mechanismen (z.B. verschiedene IDS) nutzt und Usern über ein einheitliches Interface (GUI) das Sicherheitssystem eines Netzes vollständig verwalten lässt.

IDS – Intrusion Detection System, dies ist ein Programm, welches zur Erkennung von Angriffen auf Computersystemen dient. Ein Beispiel für ein IDS ist Snort.

IDXP – Intrusion Detection Exchange Protocol, verbindungsorientiertes Protokoll der Anwendungsschicht. Es wurde für den Austausch von Daten zwischen verschiedenen IDS ausgearbeitet. Primär realisiert es den Austausch von IDMEF-Nachrichten, kann aber auch einfachen Text oder binäre Daten übertragen.

Klassenbibliothek – Aus Wikipedia, der freien Enzyklopädie
Siehe dazu: http://de.wikipedia.org/wiki/Klassenbibliothek
Eine Klassenbibliothek ist eine spezielle Form einer Programmbibliothek, womit eine Sammlung selbständiger Programmkomponenten (in diesem Fall die Klassen) gemeint ist, die für die Wiederverwendung vorgesehen sind. Die Funktionalitäten der Klassenbibliothek sind in der Regel unabhängig vom Anwendungskontext.

Pygraphlib – Bibliothek für persistente Graphen, die in der Mittelschicht verwendet wird.

RDBMS – Die heute meistbenutzten Datenbankmanagementsysteme basieren auf dem relationalen Datenmodell, das im Jahre 1969 von E. F. Codd entwickelt wurde. Folglich wird diese Variante auch als Relationales Datenbankmanagementsystem (RDBMS, teilweise auch RDBVS) bezeichnet.

Refactoring – Refactoring beschreibt das nachträgliche bearbeiten des Codes, meist mit den zweck diesen zu verbessern

SNORT – Snort ist ein Network Intrusion Detection System, mit dem aktiv der Netzwerk-Verkehr auf einem BeOS-System überwacht wird, um mögliche Hacker-Angriffe frühzeitig zu erkennen. Snort analysiert in Echtzeit, wertet Protokolle aus, durchsucht Pakete nach bestimmten Inhalten, um Angriffsmuster, Portscans, CGI-Attacken und mehr zu erkennen.
Weitere Informationen unter http://www.snort.org.

Subnetz – Ein Subnetz ist sozusagen ein Teilnet von einem größeren Netzwerk

trac – Trac ist ein verbessertes Wiki und tracking-system für Softwareentwickler

Tripwire – Sicherheitswerkzeug, welches dem Benutzer kritische Veränderungen an Datenbeständen durch Vergleichen von log-basierten Protokollen aus Datenbanken mitteilt. Tripwire zählt ebenso wie Snort zu den IDS.

ZODB – Objektorientierte Datenbank die in der Mittelschicht benutzt wird. http://www.zope.org/Wikis/ZODB/FrontPage

Literatur

[Albe] Istvan Albert. *Pygraphlib - Python Graph Library.* http://pygraphlib.sourceforge.net/. Version 6.0.1.

[Beep] *BEEPy - A Python BEEP Library.* http://beepy.sourceforge.net/.

[Pyth05] Python Software Foundation, http://www.python.org/doc/2.3.5/. *Python 2.3.5 Documentation*, Februar 2005.

[Somm01] Ian Sommerville. *Software Engineering.* Pearson Studium. 6. Auflage, 2001.

[Twis] Twisted Matrix Laboratories, http://www.twistedmatrix.com. *Twisted.*

[vLFi01] Martin von Löwis und Nils Fischbeck. *Python 2, Einführung und Referenz der objektorientierten Skriptsprache.* Addison-Wesley. 2. Auflage, 2001.

[Zopea] Zope Corporation, http://www.zope.org/Documentation/Books/ZDG/current/. *Zope Developer's Guide.* 2.4 edition.

[Zopeb] Zope Corporation, http://www.zope.org. *Zope Homepage.*

Index