

Studienjahresarbeit

# **Konzept eines Frameworks zur Auswertung von Traffic-Daten in paketvermittelten Netzwerken**

Markus Brückner

Matrikelnummer: XXXXX

Betreuer: Ralf Döring

1. August 2003

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Anforderungen und Ziele</b>	<b>3</b>
2.1	Voraussetzungen im FeM-Net . . . . .	3
2.2	Einordnung . . . . .	4
2.3	Ziele des Entwurfs . . . . .	4
<b>3</b>	<b>Realisierung</b>	<b>5</b>
3.1	Die grundlegende Idee . . . . .	5
3.1.1	Unterteilung des Systems . . . . .	5
<b>4</b>	<b>Implementierung</b>	<b>7</b>
4.1	Objektorientierter Ansatz . . . . .	7
4.2	Typsistem . . . . .	7
4.2.1	Basistypen . . . . .	7
4.2.2	Erweiterungen und Typsicherheit . . . . .	7
4.3	Einbindung mehrerer Datenquellen . . . . .	8
4.3.1	Gründe . . . . .	8
4.3.2	Ansätze . . . . .	8
4.3.3	Realisierung . . . . .	10
4.4	Ansätze für die Verwendung mehrerer Backends . . . . .	11
4.5	Wahl der Implementierungsumgebung . . . . .	11
4.5.1	Vorbetrachtungen . . . . .	11
4.5.2	Java . . . . .	11
4.5.3	C++ . . . . .	13
4.5.4	C# in Verbindung mit dem Microsoft .NET-Framework . . . . .	14
4.5.5	Laufzeiteffizienz der einzelnen Umgebungen . . . . .	15
4.5.6	Entscheidung für eine Entwicklungsumgebung . . . . .	16
4.6	Schnittstellenbeschreibung . . . . .	17
4.6.1	Die Basisklasse Module . . . . .	19
4.6.2	Das Modul InCodec . . . . .	21
4.6.3	Das Modul OutCodec . . . . .	22
4.6.4	Das Steuermodul . . . . .	23
4.6.5	Das Modul Backend . . . . .	24
4.6.6	Die Klasse TConfigurationTree . . . . .	25
4.6.7	Die Klasse TDataTree . . . . .	27

4.6.8	Die Klasse TNotifier . . . . .	27
4.6.9	Die Klasse TValue . . . . .	28
4.6.10	Der Aufzählungstyp EPacketError . . . . .	29
4.6.11	Der Aufzählungstyp EOutputError . . . . .	30
4.6.12	Der Aufzählungstyp ENotificationCode . . . . .	30
<b>5</b>	<b>Zusammenfassung und Ausblick</b>	<b>31</b>

## 1 Einleitung

Die Forschungsgemeinschaft elektronische Medien e.V. ([FeM]) ist ein studentischer Verein an der Technischen Universität Ilmenau, der ein derzeit etwa 1500 Nutzer umfassendes Campus-Netzwerk namens FeM-Net betreibt. Dieses Netzwerk ist über das Rechenzentrum der TU mit dem Deutschen Forschungsnetz ([DFN]) verbunden. Um eine bedarfsgerechte Bezahlung des DFN-Anschlusses zu ermöglichen, existieren verschiedene Traffic-Klassen, in die sich eine Forschungseinrichtung einordnen lassen muss. Diese umfassen je eine bestimmte Maximalmenge an Daten, die pro Monat aus dem DFN in das einrichtungseigene Netz übertragen werden darf.

Bedingt durch die Größe des von der FeM e.V. betriebenen Netzwerkes und der missbräuchlichen Nutzung einiger Mitglieder kam es im Juli 2001 zum Überschreiten dieser Maximalmenge an der TU Ilmenau. Daraufhin wurde nach Möglichkeiten gesucht, diesen Missbrauch zu verhindern. Da man das, was man verhindern möchte, erst erkenn- und messbar machen muss, entstand das Accounting-Projekt mit dem Ziel, die übertragene Datenmenge pro Nutzer aufgeschlüsselt nach verschiedenen Übertragungszielen zu messen und somit Reaktionen auf missbräuchliches Verhalten zu ermöglichen.

Diese Arbeit befasst sich mit einer Softwareumgebung, die eine umfassende Auswertung der beim Aufzeichnen von Datenübertragungen gewonnenen Daten ermöglicht. Sie entstand aus dem Wunsch, die für die FeM e.V. geschaffenen Speziallösungen durch ein einfach zu wartendes und allgemein verwendbares System abzulösen.

## 2 Anforderungen und Ziele

### 2.1 Voraussetzungen im FeM-Net

Das durch die FeM e.V. betriebene Campus-Netzwerk umfasst im Moment etwa 1700 Rechner, verteilt auf sechs Segmente. Diese laufen an einem Cisco Catalyst 6509 [CC6] zusammen, welcher diese sowohl untereinander als auch mit einem Gigabit-Ethernet-Link in das Universitätsrechenzentrum verbindet. Des Weiteren agiert der Cisco als Paketfilter, welcher die Kommunikation auf das TCP-Protokoll im Portbereich zwischen 0 und 1024 begrenzt. Um trotzdem Dienste wie FTP und HTTP vollständig nutzen zu können, sind einige Rechner von dieser Begrenzung ausgenommen und agieren als Application Level Gateways. Auf diesen Rechnern fallen daher ständig Logdaten an, welche in eine Traffic-Auswertung einbezogen werden müssen.

Aufgrund der Tarifstruktur des DFN besteht bei der Auswertung nur Interesse an den aus dem DFN in das Universitätsnetzwerk transferierten Daten. Datenströme in

das DFN sowie vom FeM-Net in das restliche Universitätsnetzwerk spielen eine eher untergeordnete Rolle. Daher ist es sinnvoll, die auszuwertenden Daten nach Quelle bzw. Ziel und Flussrichtung zu trennen und einzeln zu betrachten.

Letztlich war es wünschenswert die gewonnenen Daten statistischen Auswertungen zu unterziehen, um beispielsweise besonders häufig genutzte Dienste netzintern zu spiegeln und so die transferierte Datenmenge aus dem DFN zu minimieren.

### 2.2 Einordnung

Eine Umgebung zum Trafficmonitoring lässt sich in drei Teile einteilen (vgl. Abb. 1).

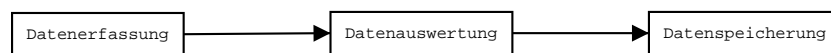


Abbildung 1: Einteilung einer Umgebung zum Trafficmonitoring

Für die *Datenerfassung* gibt es bereits ausgereifte und teilweise proprietäre Lösungen, wie beispielsweise Cisco NetFlow [CIN]. Zur *Datenspeicherung* existieren ebenfalls verschiedene Möglichkeiten, angefangen von einfachen Textdateien bis hin zu ausgereiften Datenbanksystemen, welche eine umfassende Weiterverarbeitung der gewonnenen Daten zulassen.

Diese Arbeit wird sich daher auf die Entwicklung eines Systems zur *Datenauswertung* konzentrieren. Die vorhandenen Lösungen zur Datenerfassung sind meist nicht an die speziellen Bedürfnisse einer spezifischen Auswertungs Umgebung anpassbar, so dass sich diese nach den gelieferten Daten richten muss. Auch können im Interesse einer besseren Skalierbarkeit des Gesamtsystems verschiedene Möglichkeiten zur Datenspeicherung in Betracht kommen, so dass auch hier die Auswertungs Umgebung flexibel sein muss.

### 2.3 Ziele des Entwurfs

Aus den bisher gewonnenen Erkenntnissen lassen sich folgende Entwurfsziele für ein Framework zur Implementierung eines solchen Systems ableiten:

**Erweiterbarkeit** Das Framework soll sich auf einfache Weise an neue Anforderungen bezüglich Auswertung und Eingabedaten anpassen können.

**Kapselung** Die einzelnen Teile des Frameworks sollen so weit voneinander gekapselt werden, dass ihr einfacher Austausch ohne weiteres möglich ist.

**Portierbarkeit** Das Framework soll unabhängig vom zugrundeliegenden System sein. Weder Betriebssystem noch Rechnerarchitektur sollen eine Rolle spielen.

**Einfachheit** Die Entwicklung neuer Systemteile soll nicht durch komplizierte APIs gebremst werden. Insbesondere ist auf eine klare Systemstruktur zu achten.

**Effizienz** Aufgrund der großen Datenmengen, die an schnellen Verbindungen anfallen können, ist es nötig, bei der Implementierung des Systems auf hohe Effizienz zu achten.

## 3 Realisierung

### 3.1 Die grundlegende Idee

Ausgehend von den oben genannten Zielen gilt es zunächst Gemeinsamkeiten zwischen den verschiedenen Arten von Logdaten zu finden.

Prinzipiell kann man alle Logdaten, die sich auf Verbindungen/Datentransfers beziehen, als Strom von Datenpaketen mit einer Quelle, einem Ziel, einem Zeitpunkt und einer definierten Größe auffassen. Dabei spielt es keine Rolle, ob Quelle und Ziel IP-Adressen, MAC-Adressen aus Ethernetpaketen, URLs oder sogar E-Mail- und Postadressen sind. Bei letzteren ergibt sich allerdings die Schwierigkeit, sie als Quelle und Ziel in Computernetzwerken zu verwenden.

Legt man dieses Modell von Logdaten zugrunde, so lässt sich daraus direkt das interne Modell eines Paketes ableiten:

```
Paket := Absender : Adresse;  
        Ziel      : Adresse;  
        Zeit      : Zeitpunkt;  
        Größe     : Integerwert;
```

Innerhalb des Frameworks werden nur Pakete dieses Aufbaus behandelt, unabhängig vom Format, in dem die Eingabedaten vorliegen.

#### 3.1.1 Unterteilung des Systems

Der Aufbau des Gesamtsystems auf einem einheitlichen Datenmodell lässt eine Trennung in einen *InCodec* zum Einlesen der Daten und ein *Backend* zum Ausführen der eigentlichen Auswertung zu. Um Coderedundanzen zu vermeiden und die Programmierung der Module weiter zu vereinfachen ist des Weiteren eine Auslagerung der Ausgabe der

gewonnenen Daten sowie der Initialisierung der Module wünschenswert. Das Framework bekommt damit die in [Abbildung 2](#) gezeigte Unterteilung in vier Teile.

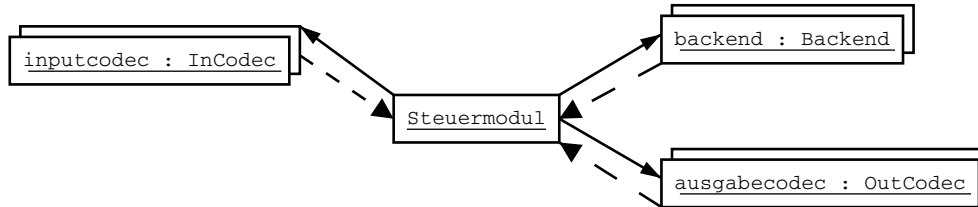


Abbildung 2: Unterteilung des Frameworks

Die in der Abbildung verlaufenden, durchgezogenen Pfeile stellen die Aufrufbeziehungen innerhalb des Frameworks dar. Eine Kommunikation der Systemteile ist nur über das Steuermodul vorgesehen. Des Weiteren erfolgt ein Aufruf einzelner Systemteile nur aus dem Steuermodul heraus. Aufgrund der in [4.3.3](#) genannten Gründe erfolgt die Verarbeitung im gesamten System synchron. Somit ist ein direkter Aufruf von Funktionen des Steuermoduls aus den anderen Systemteilen heraus nicht notwendig. Um trotzdem die Benachrichtigung des Systems aus eventuell in Komponenten laufenden Parallelprozessen zu ermöglichen, wurde die in [4.6.8](#) beschriebene Klasse `TNotifier` geschaffen. Deren Signale werden in [Abbildung 2](#) durch gestrichelte Pfeile symbolisiert.

Das `Steuermodul` übernimmt zentrale Aufgaben wie das Initialisieren der einzelnen Komponenten und die Kommunikation zwischen diesen. Der `OutCodec` dient zum Schreiben der ausgewerteten Daten in verschiedene Ziele (Datenbanken, Konsole, Dateien etc.). Der `InCodec` liest die Daten des Eingabestromes und wandelt sie in die interne Paketdarstellung um. Das `Backend` übernimmt die eigentliche Arbeit der Auswertung und stellt somit wohl den komplexesten Teil des Systems dar.

Wie aus [Abbildung 2](#) ersichtlich, können sowohl `InCodec`, `OutCodec` als auch das `Backend` mehrfach im System vorhanden sein. Aus Effizienzgründen ist es bei den anfallenden Datenmengen wünschenswert, mehrere Auswertungen gleichzeitig über die Eingabedaten laufen zu lassen, um diese nicht mehrfach einlesen zu müssen. Des Weiteren ist es zur einfachen Integration verschiedener Datenquellen ebenfalls von Vorteil, mehrere Datenquellen in einem Durchlauf einzulesen, um eine Neuinitialisierung des Backends zu vermeiden.

## 4 Implementierung

### 4.1 Objektorientierter Ansatz

Der Aufbau des Frameworks und die fünf eingangs genannten Ziele lassen eine objektorientierte Implementierung ideal erscheinen. Die *Erweiterbarkeit* des Frameworks wird durch die Vererbungsmechanismen der OO-Sprachen unterstützt. Ebenso ist eine *Kapselung* der einzelnen Module durch Klassen am einfachsten. Der *Portabilität* steht bei geeigneter Sprachwahl ebenfalls nichts im Wege, da viele objektorientierte Sprachen standardisiert und auf vielen Systemen verfügbar sind. Die *Einfachheit* der APIs wird zwar durch den objektorientierten Ansatz nicht besonders unterstützt, aber auch nicht behindert, so dass eine Entscheidung auch hier zugunsten einer objektorientierten Lösung ausfallen kann. Da für viele objektorientierte Sprachen auch hochoptimierende Übersetzer zur Verfügung stehen, steht dem Ziel der *Effizienz* nichts im Wege.

### 4.2 Typsystem

#### 4.2.1 Basistypen

**T**Packet**** Der Prototyp eines Paketes mit Quelle, Ziel, Zeitpunkt und Größe.

**T**Address**** Ein abstrakter Basistyp, dessen abgeleitete Klassen (bzw. deren Instanzen) Quelle und Ziel eines Paketes eindeutig kennzeichnen. Konkrete Adresstypen werden von diesem Basistyp abgeleitet. Ein Beispiel dafür wäre eine Klasse, welche den Endpunkt einer TCP-Verbindung kennzeichnet: eine Kombination aus IP-Adresse und Port.

**T**Timestamp**** Ein Wert, der den Zeitpunkt, zu dem ein Paket gesendet wurde, kennzeichnet.

**T**Size**** Ein Wert, der die Größe eines Paketes in Bytes kennzeichnet.

#### 4.2.2 Erweiterungen und Typsicherheit

Das gesamte Framework basiert auf der Idee, Trafficlogdaten als Strom von Paketen mit Quelle, Ziel, Zeit und Größe zu behandeln. Trotzdem kann es wünschenswert sein, Module zu entwickeln, die einerseits die Vorteile des Frameworks (beispielsweise unabhängig von dem Format der Eingabedaten und des Ausgabezieles zu sein) nutzen und andererseits eine bessere Anpassung an den Eingabestrom erreichen, um mehr Informationen gewinnen zu können. Als Paradebeispiel dafür sei eine Trafficstatistik getrennt

nach TCP/UDP-Ports angeführt. Normalerweise sollte es für Backends nicht nötig sein, genau zu wissen, welche Art von Adresse ein Paket trägt. In diesem Beispiel ist das jedoch nötig, da eine Portstatistik über Ethernet-Pakete beispielsweise aufgrund des dort fehlenden Konzeptes der Ports nicht sinnvoll machbar ist. Daher muss es für das Backend möglich sein, den genauen Typ eines Objektes festzustellen. In C++ würde sich zur Lösung dieses Problem beispielsweise der Mechanismus des *Runtime Type Information* (RTTI, vgl. [Str00] 15.4) anbieten. Je nach Wahl der Implementierungssprache sind eventuell andere Mechanismen nötig oder möglich.

### 4.3 Einbindung mehrerer Datenquellen

#### 4.3.1 Gründe

Aus 2.1 ergab sich beim Entwurf des Systems der Wunsch nach der Möglichkeit, mehrere Datenquellen parallel in das System einzuspeisen.

Es ist im FeM-Net wie in vielen anderen Netzwerken so, dass zum Zweck der Entlastung des Uplinks verschiedene Rechner als Proxy / Application Level Gateway fungieren. Bei einem Accounting auf IP-Paketbasis werden die Nutzdaten, welche über einen Proxy gezogen werden, als Traffic des Proxies erfasst und können daher nicht direkt nach Nutzern aufgeschlüsselt werden. Allerdings führen gängige Proxies, wie beispielsweise der Squid Web Proxy Cache ([Squ]) eigene Logfiles, in denen jede Verbindung vermerkt wird. Es ist also wünschenswert, diese Logfiles ebenfalls in den Auswertungsprozess mit einzubeziehen.

#### 4.3.2 Ansätze

Für dieses Einspeisen mehrerer Datenquellen gibt es verschiedene Ansätze:

1. Die Einspeisung erfolgt in verschiedenen Durchläufen. Das Steuermodul wird angewiesen, in einem weiteren Durchlauf einen passenden InCodec für die entsprechende Datenquelle zu laden und eine neue Auswertung anzustoßen.

*Vorteile:*

**Einfache Struktur des Steuermoduls** Es ist keine Verwaltung mehrerer Quellen nötig.

*Nachteile:*

**Steigende Komplexität des Backends** Dieses Verfahren entspricht prinzipiell dem mehrfachen Aufruf des Auswerteprogrammes in verschiedenen Konfigurationen mit verschiedenen InCodecs. Daher muss sich das Backend um die Persistenz seiner Daten kümmern.

**Keine Auswertung mehrerer dauerhafter Quellen möglich** Da ein Wechsel der Datenquelle erst nach dem Abarbeiten der vorhergehenden möglich ist, können im System keine Quellen Verwendung finden, welche kontinuierlich Daten liefern.

2. Die Einspeisung erfolgt in einem Durchlauf, getrennt nach Datenquellen. Dazu wird das Steuermodul angewiesen, nacheinander verschiedene InCodecs zu laden und deren Daten an das Backend weiterzureichen, ohne dieses in der Zwischenzeit neu zu initialisieren.

*Vorteile:*

**Einfache Struktur des Backends** Da alle Datenquellen in einem Durchlauf ausgewertet werden, ist es für das Backend unnötig, sich um die Persistenz seiner Daten zu kümmern.

*Nachteile:*

**Keine Auswertung mehrerer dauerhafter Datenquellen möglich** Auch hier erfolgt ein Wechsel zur nächsten Datenquelle erst nach dem Abarbeiten der aktuellen Daten. Damit können wieder keine Quellen zum Einsatz kommen, welche kontinuierlich Daten liefern.

3. Die Einspeisung erfolgt in einem Durchlauf nach einer Art *Round-Robin-Verfahren* über die Datenquellen. Dazu muss das Steuermodul alle nötigen Datenquellen gleichzeitig offen halten und jeweils ein Paket von einer Datenquelle entgegen nehmen um danach zur nächsten weiterzugehen. Kommt das Steuermodul bei der letzten Datenquelle an, wird die Auswertung wieder bei der ersten fortgesetzt.

*Vorteile:*

**Einfache Struktur des Backends** Wie bereits bei Variante 2 erfolgt die Auswertung der Daten in einem Durchlauf, so dass eine Herstellung von Persistenz der Daten des Backends unnötig ist.

**”Live-Auswertung” mehrerer Quellen möglich** Da nun pro Quelle immer nur ein Paket ausgelesen wird, ist auch bei kontinuierlich laufenden Datenquellen garantiert, dass mindestens einmal in  $n$  Schritten ( $n$  ist die Anzahl der Quellen) jede Datenquelle ausgelesen wird.

*Nachteile*

**Hohe Komplexität des Steuermoduls** Da das Steuermodul nun die Verwaltung mehrerer Datenquellen implementieren muss, steigt seine Komplexität.

Aufgrund der Tatsache, dass eine parallele Auswertung mehrerer kontinuierlicher Datenquellen in vielen Einsatzszenarien wünschenswert erscheint, ist Möglichkeit 3 die einzig sinnvolle. Der gestiegene Implementierungsaufwand für das Steuermodul ist vernachlässigbar, da dieses, im Gegensatz zu den anderen Systemteilen, nur einmal implementiert werden muss.

Im folgenden wird daher näher auf Details dieses Ansatzes eingegangen.

### 4.3.3 Realisierung

Eine wirkliche Parallelisierung verschiedener Datenquellen kann auf zwei Arten geschehen. Einerseits kann ein InCodec derart implementiert werden, dass er dem Steuermodul eine Unterbrechung oder das Ende des Eingabestromes über Rückgabewerte signalisiert (kooperatives Multitasking). Andererseits kann die Parallelität über die Implementierung von Threads im Steuermodul erreicht werden (preemptives Multitasking). Zwar vereinfacht das preemptive Multitasking die Implementierung des InCodecs, steigert aber andererseits die Komplexität des Steuermoduls erheblich. Des Weiteren ist die Unterbrechbarkeit der Eingabecodecs nur mit Unterstützung betriebssystemsspezifischer Hilfsmittel wie Threads oder Leichtgewichtsprozesse erreichbar. Diese wiederum erhöhen den Aufwand bei der Verarbeitung einzelner Datenpakete erheblich. Außerdem ist eine Unterbrechbarkeit der Eingabecodecs nicht immer sinnvoll. Je nach Datenquelle ist der Codec eventuell in der Lage bei jedem Aufruf Daten zu liefern, beispielsweise bei der Auswertung vorher gespeicherter Logdateien. Um eine effizientere Anpassung an die Charakteristika des Eingabedatenstromes zu erreichen, ist das mit weniger Overhead versehene kooperative Multitasking besser geeignet als das aufwändigere preemptive Multitasking. Außerdem verbessert der Verzicht auf Threading innerhalb des Frameworks dessen Portierbarkeit auf Plattformen, die Threads nur schlecht oder gar nicht unterstützen. Letzten Endes ist der zusätzliche Aufwand bei der Implementierung des InCodecs mit kooperativem Multitasking nur minimal. Eine Entscheidung kann daher bedenkenlos zugunsten dieser Variante fallen.

## 4.4 Ansätze für die Verwendung mehrerer Backends

Wie bereits angedeutet ist es aus Effizienzgründen wünschenswert, mehrere Backends und damit mehrere verschiedene Arten der Auswertung gleichzeitig in das System zu integrieren. Dies ist analog zu 4.3.2 realisierbar. Dabei gelten die dort genannten Vor- und Nachteile in leicht veränderter Form auch hier. Beide Ansätze, die jeweils alle Daten erst an ein Backend liefern und dann zum nächsten übergehen, machen eine "Live-Auswertung" unmöglich. Daher kann wiederum die Entscheidung über die Einbindung nur zugunsten eines Round-Robin-Verfahrens fallen. Um gegenseitige Beeinflussung der Backends auszuschließen, muss dabei darauf geachtet werden, dass diese die eingelieferten Daten nicht verändern. Dies kann durch geeignete Schnittstellendeklarationen der Module erreicht werden.

## 4.5 Wahl der Implementierungsumgebung

### 4.5.1 Vorbetrachtungen

Die Konzentration auf einen objektorientierten Ansatz verlangt die Unterstützung durch eine entsprechende Programmiersprache. Zwar ist es prinzipiell möglich in Sprachen wie C oder Pascal objektorientiert zu programmieren, jedoch lässt die Unterstützung des Ansatzes durch spezielle Sprachmittel stark zu wünschen übrig. Weitere Beschränkungen bei der Wahl der Implementierungsumgebung ergeben sich durch den Wunsch nach Portabilität und Effizienz. Auch das Design des Frameworks als offenes, erweiterbares System stellt besondere Anforderungen an die Umgebung hinsichtlich der Einbindung von Fremdmodulen.

Im Folgenden werden drei Umgebungen hinsichtlich ihrer Eignung zur Implementierung des Frameworks untersucht: C++, Java und C# in Verbindung mit dem Microsoft .NET-Framework. Sprachen wie Delphi, Visual Basic oder Smalltalk kommen durch ihre Beschränkung auf nur eine Plattform oder zu geringe Verbreitung nicht in Betracht.

### 4.5.2 Java

Die von der Firma Sun 1995 erstmals vorgestellte Sprache Java wurde von Anfang an als objektorientierte Sprache entworfen. Konzepte wie Polymorphie und Vererbung ziehen sich daher durch den gesamten Sprachentwurf.

Um konsistente und erweiterbare Schnittstellen im System zu schaffen, bietet sich das Konzept der Vererbung an. Da hierzu allerdings nur eine Festlegung der Schnittstelle getroffen werden soll und keine Funktionalität in den Basisklassen bereitzustellen ist, eignen sich abstrakte Basisklassen als Grundlage, von denen keine Objekte erzeugbar

sind. Dieses Konzept steht in Java in Form der sogenannten **interfaces** (Vgl. [JIF]) zur Verfügung. Mit Hilfe dieses Konstruktes ist es möglich, für die vier Systemteile Schnittstellen festzulegen, ohne bereits Funktionalität zu implementieren, welche eine weiterführende oder alternative Implementierung behindern könnte.

Die Austauschbarkeit der Module kann durch das Konzept der Polymorphie gewährleistet werden. Einer Implementierung eines Systemteils muss damit nur die Schnittstelle der Basisklasse der nötigen anderen Systemteile bekannt sein, welche unabhängig von einer konkreten Implementierung des anderen Teils ist. Ein Systemteil, welcher das ihm vorgegebene **interface** implementiert kann damit über eine Variable vom Typ des **interfaces** referenziert werden. Damit besteht eine Abhängigkeit nur noch im äußeren Verhalten der Systemteile, nicht mehr im konkreten Typ oder einer konkreten Implementierung.

Da die Sprache Java von Anfang an mit dem Ziel einer hohen Portierbarkeit geschaffen wurde, liegt ihr das Konzept einer virtuellen Maschine (sog. JVM, vgl. [JVM] 1.2.3) zugrunde. Damit ist ein Java-Programm theoretisch ohne Änderungen am Quell- oder Binärcode auf jeder Plattform lauffähig, auf welcher eine JVM in der entsprechenden Version zur Verfügung steht. In der Praxis gibt es dabei leider einige Schwierigkeiten. Aufgrund des Zwanges einer virtuellen Maschine ist Java-Programmen die direkte Interaktion mit dem Betriebssystem oder der Hardware nur über Umwege möglich. Diese Umwege (speziell in Form des sogenannten „Java Native Interface“ zum Aufruf fremder Binärmodule) schränken allerdings die Portierbarkeit eines Programmes wiederum auf die Portierbarkeit eventuell benötigter Binärmodule ein. Erschwerend kommen ständige Änderungen an der JVM oder den mitgelieferten Bibliotheken hinzu. Das sogenannte „Java Runtime Environment“, die Gesamtheit aus JVM und Laufzeitbibliotheken ist mittlerweile in Version 1.4.1 aktuell. Von Version 1.0 bis zu dieser aktuellen Version gab es derartige Veränderungen an den Laufzeitbibliotheken, dass kaum ein nichttriviales Programm auf allen Versionen lauffähig ist. Diese gravierenden Veränderungen sind auch in Zukunft zu befürchten, da die Vorgaben für die Sprache Java allein der Firma Sun und keinerlei internationalen Standards unterliegen. Ein weiteres Problem ist die Tatsache, dass Java-Programme aufgrund der Ausführung des Java-Bytecodes in einer virtuellen Maschine nie die Leistung von auf die Prozessorarchitektur angepasstem Binärcode erreichen. Zwar existieren verschiedene Ansätze (z.B. Just-In-Time-Compiler), welche die Ausführungsgeschwindigkeit des Bytecodes steigern; an Binärcode reichen diese Ansätze jedoch nicht heran. Für einige Plattformen existieren auch Übersetzer, welche in der Lage sind, aus Java-Programmen Binärcode zu erzeugen, dessen Ausführungsgeschwindigkeit anderen kompilierten Programmen in nichts nachsteht. Zwar beschränkt sich eine effiziente Nutzung einer Java-Implementierung damit auf Plattformen, für die ein entsprechender Übersetzer zur Verfügung steht, jedoch stellt dies bei entsprechender Ver-

breitung solcher Programme kein Hindernis dar. Hervorzuheben wäre in diesem Zusammenhang das Java-Frontend der GNU Compiler Collection [GCC], welches für alle verbreiteten Plattformen zur Verfügung steht.

Als speziell für die Implementierung des Frameworks wichtiger Vorteil sei die einfache Handhabung von dynamisch ladbaren Modulen in Java genannt. Diese vorkompilierten Module können aufgrund ihres standardisierten Binäraufbaus sehr einfach in ein bestehendes System eingebunden werden, unabhängig vom Compiler, mit dem sie ursprünglich übersetzt wurden. Somit ist es einfach möglich neue InputCodecs und Backends in das System zu integrieren, ohne dieses dabei komplett übersetzen zu müssen.

### 4.5.3 C++

Die 1998 in ISO/IEC 14882 standardisierte Sprache C++ wurde zu Beginn der 80er Jahre von Bjarne Stroustrup in den AT&T Bell Laboratories ursprünglich als Erweiterung zum damals gebräuchlichen C entworfen. Stroustrup bediente sich beim Entwurf der Sprache bei verschiedenen anderen Programmiersprachen (vgl. [Str00] 1.4). Daher entwickelte sich C++ zu einer Multiparadigmen-sprache, welche unter anderem auch objektorientierte Konzepte unterstützt.

Wie auch in Java existiert in C++ die Möglichkeit der abstrakten Schnittstellenbeschreibung, hier durch sogenannte abstrakte Basisklassen (vgl. [Str00] 12.3). Diese unvollständigen Klassen eignen sich zur Definition von Schnittstellen, welche dann über den Mechanismus der Ableitung implementiert werden, da sie zwar als Basisklasse dienen, jedoch nicht instantiiert werden können.

C++ unterstützt den Mechanismus der Polymorphie über virtuelle Funktionen. Damit ist es möglich, dass andere Systemteile nur den Typ der Basisklasse kennen müssen ohne über die Details der konkreten Implementierung informiert zu sein, was eine Erweiterung der bestehenden Module und Datentypen elegant ermöglicht.

Aufgrund der bereits erwähnten Standardisierung der Sprache und der im Standard aufgeführten umfangreichen Bibliotheken, welche für viele Anwendungsfälle geeignete Algorithmen und Datenstrukturen zur Verfügung stellen, existiert für nahezu jede Plattform eine C++-Implementierung. Daher ist prinzipiell eine weitreichende Portierbarkeit des Frameworks auf Quellcodeebene gegeben. Aufgrund der späten Standardisierung unterscheiden sich allerdings viele C++-Implementierungen im Detail. Diese Probleme schränken die Portierbarkeit des Frameworks wiederum ein.

Da sich C++ aus der Sprache C heraus entwickelt hat, welche bis heute die Sprache der Wahl für hardwarenahe und effiziente Programmierung ist, wurde von Anfang an Wert darauf gelegt, möglichst effizient mit vorhandenen Ressourcen umzugehen. Die Sprache bietet daher die gesamte Bandbreite von hardwarenaher bis zu umfangreicher,

generischer Programmierung mit Hilfe von Templates und Klassenhierarchien. Durch die Übersetzung in Maschinencode ist dabei auf jeder Plattform eine im Verhältnis zu Java hohe Ausführungsgeschwindigkeit bei geringem Speicherverbrauch gegeben.

Leider sieht der ISO-C++-Standard keine Einbindung von dynamisch ladbaren Modulen vor. Diese lässt sich immer nur mit Hilfsmitteln des zugrundeliegenden Betriebssystems erreichen. Eine Kapselung dieser Hilfsmittel wird durch verschiedene, frei verfügbare Bibliotheken erreicht. Allerdings ist für C++-Klassen, die als kompilierter Code vorliegen auch keinerlei Binärlayout standardisiert. Daher ist es nur selten möglich, bereits kompilierten Code in ein bestehendes Projekt einzubinden. Das würde erfordern, das alle Teile des Projektes mit dem selben Compiler übersetzt wurden. Zwar existieren Standardisierungsbemühungen, die das ABI<sup>1</sup> verschiedener Compiler kompatibel machen sollen, jedoch werden diese von vielen alten Compilern noch nicht unterstützt und sind daher zur Zeit noch von begrenztem praktischen Nutzen.

### 4.5.4 C# in Verbindung mit dem Microsoft .NET-Framework

Das von der Firma Microsoft [MS] 2002 freigegebene .NET-Framework [DNT] folgt im Ansatz der Idee von Java, eine Applikation einmal in eine binäre Repräsentation zu kompilieren und dann auf der Zielplattform eine Just-in-Time Kompilierung vorzunehmen. Dazu wurde die sogenannte MSIL (Microsoft Intermediate Language) entworfen und als Bestandteil der Common Language Infrastructure [CLI] beim ECMA [ECM] zur Standardisierung eingereicht. Daraus entstand der ECMA-Standard ECMA-335.

Im Zuge der Entwicklung des .NET-Frameworks wurde eine objektorientierte Sprache geschaffen, welche nach Marketingaussagen die Mächtigkeit von C++ und die Einfachheit von Visual Basic in sich vereinen sollte. Auch diese Sprache wurde beim ECMA zur Standardisierung eingereicht und unter der Bezeichnung ECMA-334 (vgl. [CS]) veröffentlicht. Nach kurzer Analyse der Sprache erweist sie sich als eine Mischung aus C++ und Java. Während einerseits Elemente von C++, wie z.B. Zeiger, benutzt werden können (wenn auch nur in sogenanntem „unsafe code“, d.h. Code, welcher nicht den Sicherheitsbeschränkungen der .NET-Laufzeitumgebung unterliegt), verfügt C# andererseits über das Konzept der „garbage collection“, welche den Programmierer davon entlastet, dynamisch angelegte Objekte zu überwachen und wieder freizugeben. Des Weiteren ist die größte Ähnlichkeit zwischen C# und Java ein einheitliches Objektsystem, welches alle Objekte von einer Basisklasse namens `Object` ableitet.

C# verfügt über das Konzept der virtuellen Funktionen, mit welchem Systemteile nur noch eine Schnittstelle implementieren müssen, ohne ihren genauen Typ anderen

---

<sup>1</sup>Application Binary Interface - der Aufbau der vom Compiler generierten Objektdateien

Systemteilen bekannt zu machen, um mit diesen zu arbeiten. Zusammen mit dem Konzept der abstrakten Funktionen lassen sich so elegant die gewünschten Schnittstellen entwerfen, um sie später in konkreten Implementierungen zu verwenden.

Ein interessanter Aspekt des .NET-Frameworks ist seine Sprachunabhängigkeit. Prinzipiell können Systemteile in jeder beliebigen Sprache, welche eine .NET-Anbindung besitzt, geschrieben werden und trotzdem letztendlich zusammenarbeiten. Dabei ist es beispielsweise möglich, ein Objekt in C++ zu implementieren und später in ein in C# geschriebenes System zu integrieren. Von Seiten des Systems macht die Wahl der Sprache keinen Unterschied mehr. Speziell für das Auswerten von Logfiles könnte das Vorteile bieten, da bereits eine Sprachbindung an Perl (vgl. [PL]) existiert. Diese Sprache wurde zum Auswerten von Texten entworfen, was die Arbeit mit im Textformat vorliegenden Logfiles vieler Applikationen sehr erleichtert.

Des Weiteren standardisiert .NET das dynamische Nachladen vorher unbekannter Systemteile, was die Integration neuer Funktionalität ohne Neukompilierung erheblich vereinfacht.

Nachteilig wirkt sich die Beschränkung des Frameworks auf die Windows-Plattform aus. Eine Referenzimplementierung von Microsoft namens *rotor*, welche für verschiedene Unix-Plattformen verfügbar ist, zeigte in ersten Test ein derart inakzeptables Laufzeitverhalten, dass man ohne Übertreibung von „unbrauchbar“ sprechen kann. Dieser Mangel wird durch die für mehrere Unix-Plattformen verfügbare, freie Implementierung *Mono* [MON] gemildert, welche mittlerweile einen Großteil des Frameworks implementiert. Anfängliche Befürchtungen bezüglich des Laufzeitverhaltens haben sich, wie in 4.5.5 ersichtlich, bei Mono nicht bestätigt.

### 4.5.5 Laufzeiteffizienz der einzelnen Umgebungen

Da rein von Sprachseite bei allen drei Sprachen hinreichende Unterstützung der angewandten Konzepte gegeben ist, muss eine Entscheidung aufgrund der Effizienz der verschiedenen Umgebungen hinsichtlich Einfachheit der Implementierung und Laufzeitgeschwindigkeit, sowie der Portabilität fallen.

Um verlässliche Aussagen über die Laufzeiteffizienz treffen zu können, wurde eine Testimplementierung in allen drei Sprachen vorgenommen und einem Test unter realen Bedingungen unterzogen. Als Testumgebung kam ein Intel Pentium III mit 800 MHz Taktfrequenz, 192 MB RAM und einer IBM Travelstar 20 GB Festplatte zum Einsatz. Als Betriebssystem fand ein Gentoo-Linux mit Kernel 2.4.19 und einem ReiserFS Dateisystem Verwendung. Diese Konfiguration erreichte beim Lesen aus dem Dateisystem eine Transferrate von etwa 9 MB/s, was sich nach ersten Messungen als mehr als ausreichend erweist, um den Einfluss der Festplatte auf die Messergebnisse vernachlässigbar klein zu

halten.

Die Aufgabenstellung war es, einen realen Testdatensatz (die aufgezeichneten Paketdaten eines Tages) einzulesen, Quelle, Ziel und Paketgröße zu analysieren, die Pakete verschiedenen Teilnehmern im FeM-Net zuzuordnen und die Summen der übertragenen Datenmenge auszugeben. Der Testdatensatz umfasste dabei 9202259 Pakete zu jeweils 29 Byte. Ausgegangen wurde bei der Analyse von 3062 möglichen Quellen im FeM-Net sowie 2 möglichen Zielnetzen (Netz der TU Ilmenau exklusive der 3062 Adressen sowie der Rest des IPv4-Adressraumes). Um Einflüsse durch Caching des Betriebssystems oder ähnliches zu eliminieren, wurde jede Implementierung einmal in einem Probedurchlauf gestartet, um dann zehn aufeinander folgende Testläufe zu absolvieren.

Es kamen folgende Entwicklungsumgebungen zum Einsatz:

Sprache	Entwicklungsumgebung	Version
Java	Blackdown-JDK	Blackdown-1.3.1-FCS, mixed mode
C++	GnuCC	2.95.3 20010315 (release)
C#	Mono ECMA CLI JIT Compiler	0.17

Folgende Laufzeiten wurden gemessen (Zeitangaben in Sekunden):

Durchlauf	Java	C++	C#
1	42,3	23,8	24,9
2	43,1	25,7	31,7
3	40,7	27,9	30,5
4	41,2	23,2	25,4
5	41,5	23,7	23,3
6	41,4	24,0	23,7
7	40,0	23,2	23,6
8	41,3	23,2	26,6
9	39,8	23,9	25,3
10	43,8	22,8	24,2
Mittelwert	41,5	24,1	25,9

#### 4.5.6 Entscheidung für eine Entwicklungsumgebung

Die Wahl der Testumgebung in 4.5.5 war an die aktuellen Gegebenheiten des bereits bestehenden Systems angelehnt. Die Ergebnisse der Tests sind daher mit Vorsicht zu bewerten. Einerseits kommen diese Ergebnisse einem realen Einsatz des Systems recht nahe, andererseits existieren andere Umgebungen, die möglicherweise komplett andere

Anforderungen stellen. So waren bei den Tests hauptsächlich Indexoperationen in Arrays zu erledigen, welche recht maschinennah erfolgen können. C# und Java haben hier aufgrund ihrer teilweise interpretierten Natur leichte Nachteile gegenüber C++. Diese könnten bei komplizierteren Operationen durchaus wettzumachen sein.

Die Laufzeiten der C++- und der C#-Implementierung liegen zwar fast gleichauf, andererseits ist durch die höhere Abhängigkeit der C#-Implementierung von ihrer Laufzeitumgebung eine wesentlich breitere Streuung der Laufzeiten beim Wechsel dieser zu erwarten. Um diese Vermutung zu untermauern wurde das Testprogramm mit *rotor*, der .NET Referenzimplementierung von Microsoft für unix-ähnliche Betriebssysteme, neu übersetzt und ebenfalls in den Test aufgenommen. Leider ergaben sich dabei derart inakzeptable Laufzeiten, dass selbst nach mehreren Stunden noch keine verwertbaren Ergebnisse vorhanden waren. Da diese Streuung der Laufzeiten eine sinnvolle Vorhersage der Effizienz nach der Portierung des Frameworks auf ein neues System nahezu unmöglich macht, fiel die Entscheidung letztlich zugunsten von C++. Diese Entscheidung wird auch durch den Wunsch nach maximaler Portabilität unterstützt. Bei sorgfältigem Entwurf des Frameworks ergeben sich nur relativ kleine Teile, welche systemabhängig gestaltet werden müssen. Der Aufwand diese zu portieren ist erheblich geringer, als die Portierung des kompletten .NET-Frameworks auf ein neues System, sollte dieses dort noch nicht vorhanden sein.

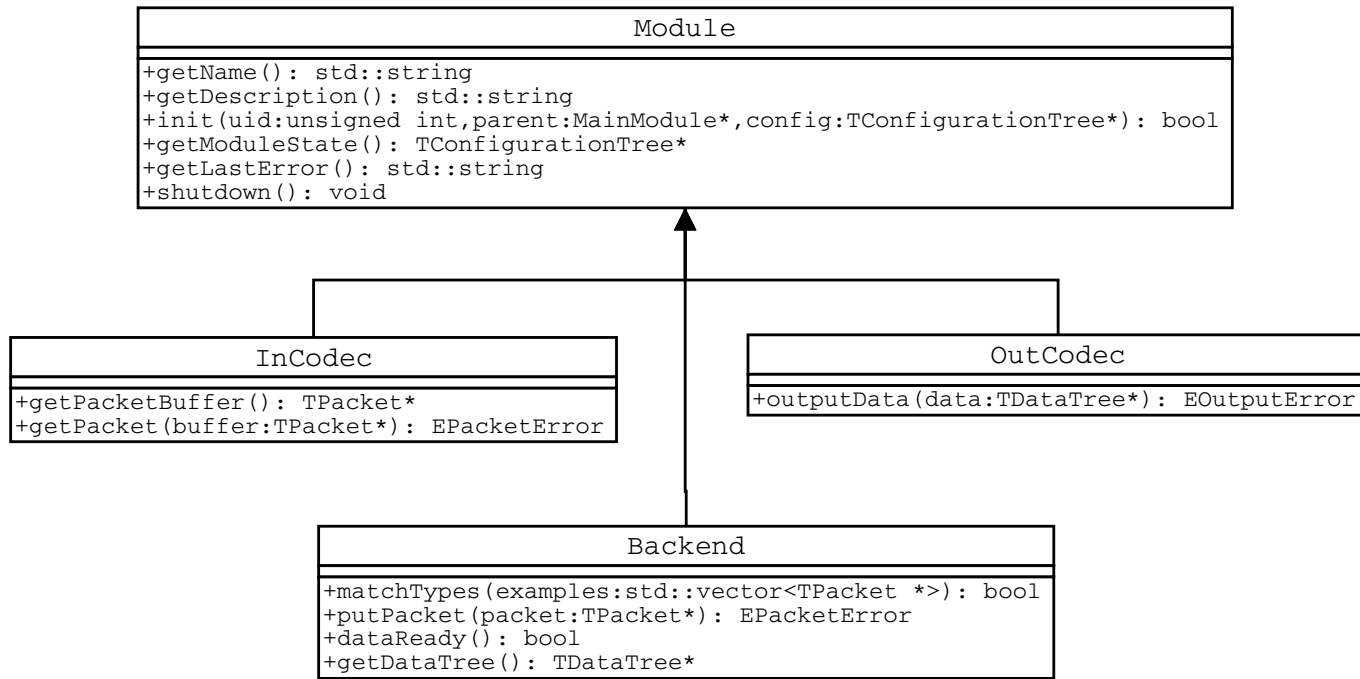
### 4.6 Schnittstellenbeschreibung

Die im Folgenden beschriebenen Schnittstellen werden sich aufgrund der in 4.5.6 getroffenen Wahl an die Datentypen der Programmiersprache C++ halten. Die Notation erfolgt in der Syntax der Unified Modelling Language (UML).

Für alle Schnittstellen werden abstrakte Basisklassen implementiert, von denen tatsächliche Implementierung ableiten müssen. Dadurch entsteht der in Abbildung 3 gezeigte Ableitungsbaum, welcher sowohl eine Erweiterung des Systems ohne Veränderung bestehender Systemteile, als auch eine wirksame Typprüfung zur Laufzeit mittels Runtime Type Information (vgl. [Str00] 15.4) ermöglicht.

Des Weiteren sind sämtliche Module auf kooperatives Multitasking ausgelegt. Daher sollten längere Berechnungen oder gar das Aufrufen blockierender externer Funktionen vermieden werden, da während dieser Zeit das gesamte System zum Erliegen kommt. Sollten derartige Funktionen notwendig sein, so sind diese innerhalb des Moduls geeignet zu behandeln, beispielsweise durch die Auslagerung in Threads. Wie in 4.3.3 bereits festgestellt wurde aus Effizienz- und Portabilitätsgründen eine generelle Auslagerung der Module in Threads verworfen. Damit bleibt dem Programmierer eines Moduls abhängig von den genauen Bedürfnissen desselben die Wahl der Implementierungsart.

Abbildung 3: Vererbungshierarchie der Systemklassen



### 4.6.1 Die Basisklasse Module

Module
<pre>+getName(): std::string +getDescription(): std::string +init(uid:unsigned int,parent:MainModule*,config:TConfigurationTree*): bool +getModuleState(): TConfigurationTree* +getLastError(): std::string +shutdown(): void</pre>

Abbildung 4: Die Basisklasse Module

Im Verlauf der Entwurfsphase stellte sich heraus, dass eine gewisse Grundfunktionalität von allen Modultypen geteilt wird. Die dafür notwendige Schnittstelle wird durch die in Abbildung 4 gezeigte Basisklasse `Module` realisiert. Von dieser Klasse ist jedes im System verwendete Modul abzuleiten.

**getName()** Diese Funktion wird vom Steuermodul aufgerufen um den Namen dieses Moduls zu erhalten. Da dieser Name vom Steuermodul verwendet wird, um die passenden, typspezifischen Konfigurationsdaten des Moduls zu identifizieren, sollte er eindeutig sein. Module, die verschiedene Funktionalitäten realisieren dürfen zwar den selben Namen zurückliefern, der Effekt wird aber in den meisten Fällen ein ungewünschtes Verhalten des Gesamtsystems sein.

**getDescription()** Die Beschreibung des Moduls, welche durch diese Funktion zurückgeliefert wird, dient allein der Information des Nutzers des Systems. Sie sollte einen kurzen Hinweis auf die genaue Funktion und den Autor des Moduls enthalten. Weiterführende Dokumentation, wie beispielsweise die Beschreibung verschiedener Konfigurationsmöglichkeiten, sollte hier nicht eingebunden werden.

**init(...)** Das Steuermodul ruft diese Funktion einmal nach der Konstruktion eines Objektes des Modultyps auf um ihm die Möglichkeit zur Initialisierung zu geben. Der Parameter `uid` ist eine vom Steuermodul vergebene, eindeutige Identifikationsnummer, mit welcher das Modul beispielsweise in der Klasse `TNotifier` referenziert werden kann. Mittels des Zeigers `parent` kann das Objekt das Steuermodul referenzieren, um sich zum Beispiel einen `TNotifier` erzeugen zu lassen. Dabei ist prinzipiell eine Erweiterung der Schnittstelle des Steuermoduls möglich, um weitere Funktionalität für die Objekte zur Verfügung zu stellen. Ein Modul sollte sich im Hinblick auf Portabilität allerdings nicht darauf verlassen, dass eine bestimmte

zusätzliche Funktionalität vorhanden ist. Der Parameter `config` enthält die spezifische Konfiguration des Moduls. Wie in 4.6.6 ausgeführt, existiert innerhalb des Konfigurationsbaumes einerseits ein typspezifischer Teil, der für alle Objekte dieses Typs gleich ist und andererseits ein objektspezifischer, welcher dazu geeignet ist, den vorher gespeicherten internen Zustand eines Objektes wiederherzustellen. Sowohl `config` als auch seine beiden Unterbäume können `NULL` sein. In diesem Fall hat das Steuermodul keinerlei geeignete Informationen vorgefunden.

Als Rückgabewert liefert `init(...)` `true`, wenn die Initialisierung erfolgreich war und `false` bei einem Fehlschlag. In diesem Fall muss eine menschenlesbare Fehlerbeschreibung zum Abruf durch `getLastError()` bereitgestellt werden.

**getModuleState()** Um eine Sicherung des aktuellen Zustandes des Objektes zu ermöglichen, wird diese Methode zur Verfügung gestellt. Das Modul liefert hier die zu sichernden Informationen in dem in 4.6.6 beschriebenen Format zurück. Auszufüllen ist dabei nur der objektspezifische Unterbaum. Informationen, die im typweiten Teilbaum eingetragen werden, ignoriert das Steuermodul. Sollen keinerlei Informationen gesichert werden, so ist `NULL` zurückzuliefern. Da diese Methode vom Steuermodul jederzeit gerufen werden kann (zum Beispiel als Reaktion auf ein Abbruchsignal des Betriebssystems), darf die Möglichkeit des Sicherns von Informationen nicht vom aktuellen Zustand des Objektes abhängig sein. Ein Fehlerfall an dieser Stelle zieht den Verlust des aktuellen Zustandes nach der Zerstörung des Moduls nach sich.

**getLastError()** Die umfassende Information des Nutzers ist im Fehlerfall meist die einzige Möglichkeit zur Behebung der Fehlerursache. Daher wird mittels dieser Methode eine nähere Beschreibung des aufgetretenen Fehlers gegeben. Die Beschreibung sollte alles Nötige zur Identifizierung der Fehlerbedingung enthalten, jedoch keine umfassenderen Lösungsvorschläge. Informationen wie bekannte Fehlergründe sind in externe Dokumentation auszulagern. Dies ist besonders wichtig, da die hier gelieferte Fehlerbeschreibung vom Steuermodul beispielsweise auch in das System-Log geschrieben werden könnte, wo seitenlange Fehlerbeschreibungen unnötig sind.

**shutdown()** Der Aufruf dieser Funktion ist der letzte in der Lebenszeit dieses Moduls. Nachdem `shutdown` aufgerufen wurde, wird das Steuermodul das betreffende Objekt zerstören. Eventuell angelegte temporäre Objekte müssen daher ebenfalls freigegeben werden, sowie geöffnete Verbindungen und Dateien geschlossen. Sollten noch weitere Daten zum Abruf durch `getPacket` vorhanden sein, so sind diese zu verwerfen.

## 4.6.2 Das Modul InCodec

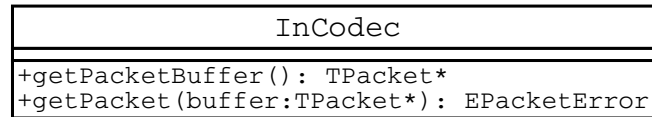


Abbildung 5: Schnittstelle einer InCodec-Klasse

Der `InCodec` (Abb. 5) ist für die Umwandlung eines spezifischen, von außen an das System gelieferten Datenstromes in das interne Datenformat zuständig. Da sich diese Arbeit aus Sicht des Steuermoduls auf ein einfaches Liefern von Paketen beschränkt, reicht eine recht schlanke Schnittstelle aus, die im folgenden näher beschrieben wird.

**getPacketBuffer()** Um ein unnötiges Kopieren der gelieferten Pakete im System, sowie das teure Allokieren und Freigeben von Speicher zu vermeiden, wird für jedes vom `InCodec` zurückgelieferte Paket ein die gesamte Lebensdauer des Steuermoduls existenter Puffer verwendet. Ein Problem ergibt sich dann durch die unterschiedliche mögliche Größe verschiedener Paketarten. Die korrespondierenden Klassen leiten zwar alle von `TPacket` ab, können aber trotzdem unterschiedliche Anzahlen von Feldern haben. Daher ist es wichtig beim Allokieren des nötigen Speichers die genaue Größe eines Objektes der Paketklasse zu kennen. Das wiederum ist aber nur im `InCodec` möglich. Um andererseits unterschiedliche, für einige Problemstellungen eventuell effizientere Pufferungsstrategien realisieren zu können, ist allerdings eine Verwaltung der Paketpuffer über das Steuermodul wünschenswert.

Diese Überlegungen führten zur Definition der Methode `getPacketBuffer()`. Sie liefert einen Zeiger auf ein vom entsprechenden `InCodec` erzeugtes Objekt zurück, dessen Typ von `TPacket` abgeleitet ist und so auch dessen Schnittstelle zur Verfügung stellt. Bei zwei Aufrufen dieser Methode müssen auch zwei verschiedene Objekte zurückgeliefert werden. Eine eventuelle Freigabe des dabei allokierten Speichers geschieht durch das Steuermodul. Daher ist eine zusätzliche Referenzierung der Paketpuffer innerhalb des `InCodec` weder nötig, noch erwünscht.

Im Falle eines Fehlers liefert `getPacketBuffer` einen `NULL`-Zeiger zurück und hinterlegt eine Fehlerbeschreibung zum Abruf durch `getLastError`.

**getPacket(...)** In dieser Funktion wird die eigentliche Konvertierungsarbeit vom Eingabestrom in das interne Paketformat vorgenommen. Jeder Aufruf liefert ein weiteres Paket aus dem Eingabedatenstrom zurück. Die entsprechenden Daten werden

dabei in die Felder des Parameters `buffer` geschrieben. Dieser Parameter zeigt auf ein zuvor durch `getPacketBuffer` im selben Modul allokiertes Objekt. Da das Objekt den genauen Typ von `buffer` kennt, ist ein Füllen zusätzlicher Felder möglich, welche nicht im Basistyp `TPacket` vorgesehen sind. Das Modul darf allerdings keine Annahme darüber machen, ob ein vorhandenes Backend von diesen Daten tatsächlich Gebrauch macht. Eine Ersetzung oder ein Ignorieren von Feldern des Basistyps darf daher nicht geschehen.

`getPacket` liefert einen Wert vom Typ `EPacketError` zurück (vgl. 4.6.10). Bei einem Rückgabewert ungleich `EPACKET_SUCCESS` wird eine Beschreibung des Fehlers zum Abruf durch `getLastError` hinterlegt.

### 4.6.3 Das Modul OutCodec

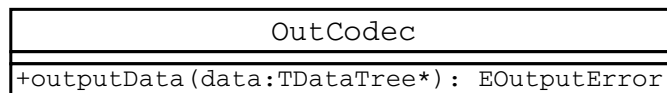


Abbildung 6: Schnittstelle einer OutCodec-Klasse

Zur einfacheren Integration des Systems in verschiedene Netzwerkmanagement- und Datenspeicherungsumgebungen sind die OutCodecs vorgesehen. In ihnen geschieht eine Anpassung der internen Datenstrukturen des Systems an das gewünschte Format externer Software. So ist beispielsweise als Datenspeicher für aggregierte Daten sowohl eine einfache Textdatei, wie auch eine hochentwickelte relationale Datenbank denkbar. Außerdem wären Anbindungen an Netzwerkmanagementsysteme auf SNMP-Basis oder eine Einbindung von Fremdgeräten in eine auf Cisco's Netflow [CIN] basierende Accountingumgebung möglich, um Beispiele für ein anderes Konzept als das eines externen Datenspeichers zu nennen. Diese Anbindung kann nur durch eine hohe Flexibilität des Systems geleistet werden, da nicht zu erwarten ist, dass eine bereits vorhandene Umgebung angepasst werden kann.

Da sich die Funktion des OutCodec auf eine einfache Ausgabe von Daten beschränkt, kann er mit einer sehr schlanken Schnittstelle ausgestattet werden.

**outputData** Wie in 4.6.7 ausgeführt, wurde für die Repräsentation der aggregierten Daten im System eine Baumstruktur gewählt. Ein Objekt der dazu entworfenen Klasse `TDataTree` wird zur Ausgabe der Daten an den OutCodec übergeben. Die

vorhandenen Daten dürfen dabei nicht verändert werden. Sollte das notwendig werden, so sind sie vom `OutCodec` zuerst zu kopieren.

`outputData` liefert einen Wert aus dem Bereich von `EOutputError` zurück. Im Fehlerfall wird dabei eine Beschreibung des Fehlers für `getLastError` hinterlegt.

#### 4.6.4 Das Steuermodul

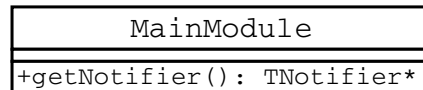


Abbildung 7: Schnittstelle des Steuermoduls

Aufgrund der Überlegungen hinsichtlich der Portabilität des Gesamtsystems wurden alle bekannten Abhängigkeiten von der Betriebssystemumgebung im Steuermodul zusammengezogen. Daher weist es in der Hinsicht wahrscheinlich die größte Komplexität auf. Da allerdings während der Laufzeit nahezu alle Aktionen vom Steuermodul ausgehen, ist seine Schnittstelle sehr schlank. Es ist für die anderen Module meist unnötig und unmöglich, mit dem Steuermodul zu interagieren.

**getNotifier()** Die einzige Funktionalität, die das Steuermodul für die restlichen Systemteile zur Verfügung stellen kann, ist die Bereitstellung einer asynchronen Benachrichtigungsmethode in Form eines `TNotifier`-Objektes. Mittels `getNotifier` kann ein Modul einen Zeiger auf ein solches Objekt erhalten. Sollte dieses Objekt nicht als Singleton implementiert sein (d.h. für jede Anfrage wird ein neuer `TNotifier` erzeugt), so übernimmt das Steuermodul die Verwaltung dieser Module selbst. Daher ist es bei seiner Zerstörung auch für die Freigabe dieser Objekte verantwortlich.

Im Fehlerfall liefert `getNotifier` `NULL` zurück. Eine nähere Beschreibung des Fehlers kann mittels `getLastError` abgerufen werden. Da die Implementierung eines asynchronen Benachrichtigungsmechanismus im Steuermodul nicht zwingend notwendig ist, kann auch das Fehlen einer solchen Implementierung als Fehler gemeldet werden. Module, welche zwingend auf diesen Mechanismus angewiesen sind, dürfen allerdings die Arbeit verweigern und einen Fehler bei der Initialisierung melden.

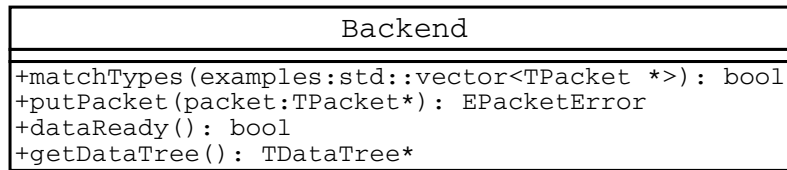


Abbildung 8: Schnittstelle einer Backend-Klasse

### 4.6.5 Das Modul Backend

Das **Backend** erledigt die eigentliche Arbeit des Systems. Sämtliche Möglichkeiten zur Auswertung der gelieferten Daten werden in Form von **Backend**-Klassen implementiert. Durch die standardisierte Schnittstelle zum Rest des Systems kann sich der Entwurf der Interna eines Backends auf dessen eigentliche Aufgabe konzentrieren.

**matchTypes(...)** Auch wenn ein Backend möglichst unabhängig von den zugrundeliegenden Daten arbeiten soll, ist es doch in den meisten Fällen unumgänglich zumindest einige Details des Datenstroms zu kennen. So ist es beispielsweise sinnlos, ein Backend zur Aufstellung einer Portstatistik zu entwerfen und dieses dann mit einem Ethernet-Datenstrom zu füttern, welcher die nötigen Daten nicht enthält. Um diese Fehler zu vermeiden bekommt jedes Backend die Möglichkeit, die Kompatibilität der Pakete sowohl untereinander, als auch zu seiner speziellen Aufgabe zu testen. Dies geschieht durch den Aufruf von **matchTypes**. Als Parameter erhält diese Funktion ein Array mit Beispielobjekten aller im System registrierten **InCodecs**. Diese Objekte sind nicht mit Daten gefüllt und dienen nur der Überprüfung der Typkompatibilität. Dazu können im **Backend** beispielsweise die RTTI-Mechanismen der Sprache C++ verwendet werden.

**matchTypes** liefert **true** zurück, wenn alle Beispielpakete unter den Gesichtspunkten dieses Backends kompatibel sind. Ist dies nicht der Fall, so sollten in der Fehlerbeschreibung für **getLastError** die Namen der inkompatiblen Pakete enthalten sein.

**putPacket(...)** Mittels dieser Methode wird der konvertierte Packetstrom vom Steuermodul eingeliefert. Dabei wird bei jedem Aufruf der Funktion genau ein Paket übergeben. Der Inhalt des Paketes darf vom Modul nicht verändert werden, da es möglicherweise noch weitere Backends geben kann, welche dieses Paket noch auswerten sollen. Sind zur Bearbeitung des Paketes längere Berechnungen nötig,

so sind diese in einen extra Thread oder Prozess auszulagern, um das laufende System nicht unnötig zu blockieren.

`putPacket` liefert einen Rückgabewert aus dem Bereich von `EPacketError` zurück. Im Fehlerfall ist dabei eine nähere Beschreibung des aufgetretenen Fehlers für `getLastError` zu hinterlegen.

**dataReady()** Wenn es im Steuermodul notwendig werden sollte, die bisher gesammelten Daten eines Backends auszulesen (beispielsweise auf Anweisung des Nutzers), so kann es mittels `dataReady` anfragen, ob sinnvolle Daten vorliegen. So kann es z.B. bei statistischen Daten nach nur wenigen vorhandenen Paketen nicht unbedingt sinnvoll sein, sich auf die Ergebnisse eines Backends zu verlassen. Um dieses anzudeuten liefert das Backend auf einen Aufruf dieser Methode `false` zurück. Zu beachten ist dabei, dass der Rückgabewert nur „beratende“ Funktion hat. Das Backend muss trotzdem beim Aufruf von `getDataTree` die vorhandenen Daten zurückliefern, auch wenn es bei `dataReady` `false` melden würde.

**getDataTree()** Die bisher gesammelten Daten sind beim Aufruf dieser Methode in Form eines `TDataTree` (vgl. 4.6.7) zurückzuliefern. Wenn eine Weiterverarbeitung der gewonnenen Daten durch das Steuermodul oder ein anderes System erfolgen soll, werden die Daten vom betreffenden Modul dupliziert, so dass aufwändige Kopieroperationen durch das Backend vermieden werden können.

Sollten keinerlei Daten vorliegen oder ein Fehler aufgetreten sein, so gib diese Funktion `NULL` zurück.

### 4.6.6 Die Klasse `TConfigurationTree`

Im laufenden Betrieb stellt sich das System für den Nutzer als eine Einheit dar. Die einzelnen Module sind dann nicht mehr als externe Bestandteile des Systems erkennbar und sollten daher auch der zentralen Konfiguration des Systems unterliegen. Des Weiteren bietet die Zentralisierung aller Konfigurationsinformationen den Vorteil, dass nicht jeder Systemteil eigene Routinen zum Lesen und Schreiben dieser Daten implementieren muss. Diese Überlegungen führten zum Entwurf der Klasse `TConfigurationTree`, welche eine Datenstruktur zur Speicherung aller Informationen bereitstellt.

Bereits in anderen Systemen, welche Konfigurationsdaten verarbeiten oder speichern müssen, hat sich eine Baumstruktur als Darstellungsform bewährt, so dass diese auch hier gewählt wird. Da bereits für den `TDataTree` in 4.6.7 eine Baumstruktur implementiert wird, kann diese mit einigen Modifikationen hier wieder Verwendung finden.

Der `TConfigurationTree` eines Typs gliedert sich in zwei Teile: einen typspezifischen, welcher Daten enthält, die für alle Objekte des Typs gelten und einen objektspezifischen, welcher genutzt werden kann, um ein spezielles Objekt des Typs in einen bestimmten Zustand zu versetzen. Innerhalb des Baumes finden nur Strings als Datentyp Verwendung. Die Umwandlung in den vom Objekt erwarteten Typ muss innerhalb des Objektes erfolgen, da sonst durch den `TConfigurationTree` eine aufwändige Speicherung der Typinformationen realisiert werden muss.

TConfigurationTree
+defaults: TDataTree
+settings: TDataTree
+getValues(id:std::string): std::vector<TValue>

Abbildung 9: Die Klasse `TConfigurationTree`

**defaults** Die typweiten Konfigurationsdaten des betreffenden Systemteils werden im Teilbaum `defaults` gespeichert. Dieser wiederum ist ein in 4.6.7 beschriebener `TDataTree`, so dass alle dort getroffenen Aussagen zum Zugriff auf Elemente des Baumes auch hier gelten.

**settings** Zur Konstruktion eines bestimmten Systemteils mit einem fest definierten Zustand können Informationen verwendet werden, welche im Unterbaum `settings` abgelegt werden. Beim Einlesen der Konfigurationsinformationen trifft das Steuermodul die entsprechenden Voraussetzungen, um für jede spezifische Konfiguration je ein Modul des Typs zu erzeugen und dieses dann mit den passenden Werten zu bestücken.

**getValues(...)** Zum schnellen Zugriff auf bestimmte Werte innerhalb des Konfigurationsbaumes dient die Funktion `getValues`. Sie bekommt eine Zeichenkette zur Identifikation der gewünschten Werte übergeben und liefert eine Liste aller passenden Werte zurück. Das Format der Zeichenkette ist dabei folgendermaßen: `(defaults|settings).unterbaum1.variable`. Mittels `defaults` und `settings` am Anfang wird zwischen den beiden Teilbäumen von `TConfigurationTree` unterschieden. `unterbaum1` ist der Name des Wurzelementes des zu erreichenden Unterbaumes.

*Beispiel:*

`getValues('defaults.listener.local_port')` liefert eine Liste aller definierten Werte von `local_port` im Unterbaum `listener` des typweiten Konfigurations-

baumes. Zu beachten ist dabei, dass nur genaue Treffer in der Liste enthalten sind. Eventuell vorhandene Unterbäume von `local_port` werden nicht berücksichtigt.

### 4.6.7 Die Klasse `TDataTree`

Aufgrund der Trennung von Backend und Ausgabe ist die Definition eines einheitlichen Formates zur Weitergabe der gesammelten Daten notwendig. Da von vornherein keine Informationen über die Struktur der zu behandelnden Daten vorliegen, ist für den Austausch ein Format zu wählen, welches möglichst einfach jede gewünschte Struktur nachbilden kann.

Diese Aufgabe übernimmt mit seiner Baumstruktur der Typ `TDataTree`. Sein Aufbau ist dabei von der Struktur eines LDAP-Verzeichnisses (vgl. z.B. [LDA]) inspiriert. Jeder Knoten im Baum kann einen Namen, einen Wert und beliebig viele Kindknoten haben. Dadurch ergibt sich der in Abbildung 10 beschriebene Aufbau eines Knotens. Der Baum in seiner Gesamtheit wird durch das Wurzelement referenziert. Ein Objekt des Knotentyps kann als Iterator zur Navigation innerhalb des Baumes dienen. Zur Traversierung der Unterbäume stehen die nötigen Funktionen von `std::multimap` zur Verfügung.

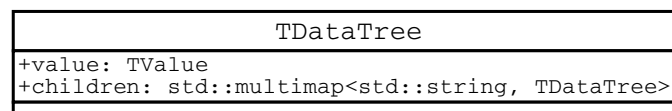


Abbildung 10: Prototyp eines Knotens

**value** Wie bereits erwähnt kann jeder Knoten einen Wert enthalten. Dieser Wert wiederum kann einen beliebigen Typ haben. Um das zu gewährleisten wird der in 4.6.9 beschriebene Basistyp `TValue` verwendet, von dem die realen Typen abgeleitet werden. Der `OutCodec` kann einfach die Serialisierung der Daten benutzen um sie ins Ausgabeformat umzuwandeln oder bei Bedarf mittels RTTI eine genauere Typbestimmung vornehmen, um eine passende Darstellung der Daten zu finden.

**children** Das assoziative Array `children` repräsentiert den Unterbaum, dessen Wurzel dieser Knoten ist. Als Schlüssel findet der Name des entsprechenden Unterknotens Verwendung.

### 4.6.8 Die Klasse `TNotifier`

Wie in 4.3.3 erwähnt ist eine Realisierung nebenläufiger Prozesse durch Threads in einzelnen Modulen durchaus vorgesehen. Aufgrund dieser möglichen Asynchronität von Ereignissen

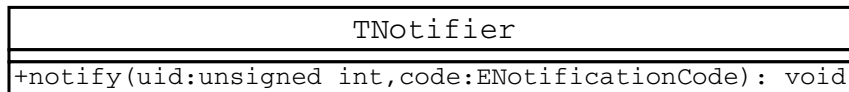


Abbildung 11: Die Klasse TNotifier

nissen ist es notwendig, das Steuermodul über deren Auftreten aus einem anderen Modul heraus zu informieren. Dazu gibt es prinzipiell zwei Verfahren: das ständige Abfragen des Modulstatus (Poll) und die asynchrone Einlieferung von Benachrichtigungen (Push). Aus Effizienzgründen wurde ein Poll-Verfahren in Laufe des Entwurfes verworfen. Leider sind Mechanismen zur asynchronen Einlieferung von Ereignissen meist vom zugrundeliegenden Betriebssystem abhängig, so dass unter der Wahl eines Push-Verfahrens fast zwangsläufig die Portabilität des Systems leidet. Um diese Einschränkung so minimal, wie möglich zu gestalten wurde die Klasse TNotifier geschaffen. In den von ihr abgeleiteten Klassen wird die gewählte Benachrichtigungsmethode des Steuermoduls gekapselt. Im Idealfall ist es also möglich, beispielsweise auf POSIX-kompatiblen Systemen die dort vorhandenen *signals* zu nutzen, während in anderen Umgebungen auf systemspezifische Dienste zurückgegriffen werden muss. Modifikationen wären dazu nur im Steuermodul bzw. in der Klasse TNotifier notwendig.

**notify(...)** Um das Steuermodul über das Auftreten eines asynchronen Ereignisses in Kenntnis zu setzen ruft das betreffende Modul die Methode `notify` des zuvor angeforderten TNotifier-Objektes auf. Übergeben wird dabei einerseits die bei der Initialisierung des Moduls vom Steuermodul übergebene eindeutige ID, sowie ein Wert aus dem Bereich von `ENotificationCode`, welcher die Art der Anforderung repräsentiert (vgl. 4.6.12). Der erhöhte Aufwand der Übergabe von `uid` erscheint gerechtfertigt, da er es erlaubt, TNotifier als zustandsloses Singleton zu implementieren und so für alle Module im System dasselbe Objekt zu verwenden. Das kann unter Umständen die Nutzung verschiedener, betriebssystemabhängiger Signalisierungsmechanismen vereinfachen.

Da TNotifier einer „fire and forget“-Philosophie folgt, d.h. es werden keinerlei Garantien hinsichtlich der Auslieferung von Benachrichtigungen gegeben, ist ein Rückgabewert für `notify` unnötig.

### 4.6.9 Die Klasse TValue

Beim Entwurf der Klasse TDataTree kam der Wunsch auf, verschiedene Datentypen innerhalb eines Baumes zu repräsentieren und dabei den verschiedenen Systemmodulen

die Möglichkeit zu geben, erweiterte Informationen über die betreffenden Datentypen zu erhalten. Zu diesem Zweck wurde die Klasse `TValue` geschaffen, welche als Basisklasse aller in `TDataTree` verwendeten Typen dient. Damit ist es innerhalb eines `OutCodec` beispielsweise mittels RTTI möglich, den genauen Typ eines Datenwertes festzustellen und so eventuell eine passende Kodierung dafür zu finden. Für die Behandlung unbekannter Datentypen stellt `TValue` eine Schnittstelle zur einfachen Umwandlung in eine Textdarstellung und die entsprechende Rücktransformation zur Verfügung, welche von jedem abgeleiteten Typ zu implementieren ist.

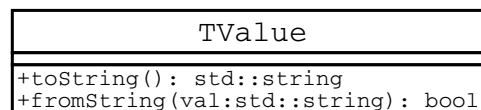


Abbildung 12: Die Klasse `TValue`

**toString()** Der Aufruf dieser Methode liefert eine Textdarstellung des betreffenden Wertes zurück. Diese sollte möglichst menschenlesbar sein und muss alle notwendigen Informationen enthalten, um von `fromString` wieder in denselben Wert umgewandelt werden zu können.

**fromString(...)** Diese Methode setzt den Wert dieser Variable auf den Wert, welcher sich aus der Umwandlung des Parameters `val` nach den Regeln des Typs ergibt. Im Erfolgsfall wird `true` zurückgeliefert, `false` im Fehlerfall.

### 4.6.10 Der Aufzählungstyp `EPacketError`

Alle Systemteile, die an der Verarbeitung von Paketdaten beteiligt sind, sollen eine einheitliche Möglichkeit haben, Fehlerbedingungen zu signalisieren. Zu diesem Zweck wurde der Aufzählungstyp `EPacketError` geschaffen. Seine verschiedenen Werte repräsentieren verschiedene Fehlerklassen, die beim Umgang mit Paketen auftreten können. Eine Variable des Typs kann einen der im folgenden spezifizierten Werte annehmen.

**EPACKET\_SUCCESS = 0** Dieser Wert repräsentiert nicht im eigentlichen Sinne eine Fehlerbedingung, sonder die erfolgreiche Ausführung der aufgerufenen Operation.

**EPACKET\_NOMOREDATA = 1** Mit der Rückgabe dieses Fehlerwertes signalisieren `InCodec` und `OutCodec`, dass sie nicht mehr in der Lage sind, Daten zu liefern oder entgegenzunehmen. Bei einem `OutCodec` können die gespeicherten Daten nach dieser Fehlermeldung noch ausgelesen werden.

**EPACKET\_WAITFORDATA = 2** Mit einem Fehler dieser Klasse signalisiert ein `InCodec`, dass im Moment keine Daten zum Auslesen vorhanden sind, aber keine Fehlerbedingung vorliegt, welche ein zukünftiges Auslesen verhindert. Vom Ansatz her ist dieser Fehlercode vergleichbar mit **EAGAIN** einiger POSIX-Funktionen. Es wird so verhindert, dass der `InCodec` blockieren muss, bis er Daten liefern kann. Dem Steuermodul wird signalisiert, dass im Moment nichts vorliegt. Somit wäre eine Implementierung eines Steuermoduls zum Beispiel in der Lage, seine Scheduling-Strategie so anzupassen, dass `InCodecs`, welche oft `EPACKET_WAITFORDATA` melden, seltener abgefragt werden.

**EPACKET\_UNKNOWN = 3** Fehler, welche nicht in eine der vorhergehenden Klassen fallen, werden durch den Fehlercode `EPACKET_UNKNOWN` repräsentiert. Aus Gründen der Robustheit sollte ein Steuermodul alle Fehlercodes größer als 2 als `EPACKET_UNKNOWN` betrachten.

### 4.6.11 Der Aufzählungstyp `EOutputError`

Zur Signalisierung von Fehlerbedingungen im Ausgabemodul wird der Typ `EOutputError` verwendet. Seine verschiedenen Werte werden im folgenden beschrieben.

**EOUTPUT\_SUCCESS = 0** Dieser Wertes steht für eine erfolgreiche Ausführung der Operation des Ausgabemoduls.

**EOUTPUT\_FORMATERROR = 1** Diese Fehlerklasse weist auf einen Fehler im übergebenen `TDataTree` hin. Ein Fehler dieser Art könnte zum Beispiel ein Bereichsfehler sein, bei dem ein Wert durch den `OutCodec` in eine Darstellung umgewandelt werden müsste, die nicht alle nötigen Informationen aufnehmen kann. Fehler dieser Art sind fast immer Implementierungsfehler in einem der beteiligten Systemteile.

**EOUTPUT\_UNKNOWN = 2** Dieser Wert repräsentiert alle Fehler, welche nicht in die Klasse `EOUTPUT_FORMATERROR` fallen. Ein Beispiel dafür wäre ein Abbrechen der Verbindung bei einem `OutCodec`, welcher seine Daten normalerweise über eine TCP-Verbindung an einen Client liefern würde. Aus Gründen der Robustheit sind auch hier alle Fehlerwerte größer als 1 als `EOUTPUT_UNKNOWN` zu betrachten.

### 4.6.12 Der Aufzählungstyp `ENotificationCode`

Zwar wird mit `TNotifier` eine Möglichkeit zum asynchronen Transport von Nachrichten bereitgestellt, eine Definition von verschiedenen Ereignisklassen allerdings fehlt bis-

her noch. Diesen Zweck erfüllt der Aufzählungstyp `ENotificationCode`. Da beim Entwurf bisher nur einmal der Bedarf für eine asynchrone Benachrichtigung auftrat, enthält `ENotificationCode` nur einen Wert.

**EN\_NEEDOUTPUTDATA = 0** Dieser Benachrichtigungscode wird von einem `OutCodec` versendet, wenn dieser eine Anforderung zur Ausgabe von Daten erhalten hat. Das Steuermodul muss daraufhin die notwendigen Daten von den Backends zusammenstellen und die Methode `outputData(...)` mit diesen Daten als Parameter aufrufen.

## 5 Zusammenfassung und Ausblick

Ziel der Studienarbeit war die Schaffung eines möglichst flexiblen Systems zur statistischen Auswertung von Datenströmen in paketorientierten Netzwerken. Ausgegangen wurde dabei von einem System, welches für die Forschungsgemeinschaft elektronische Medien e.V. in Ilmenau entwickelt wurde. Die Erfahrungen, welche bei der Entwicklung und dem Betrieb dieses System gewonnen wurden, konnten in diese Arbeit einfließen. Ausgehend von den Bedingungen, welche bei der Analyse von Trafficdaten vorliegen, wurde das System in vier verschiedene Teile aufgeteilt, welche eine umfassende Anpassung an verschiedenste Umgebungen ermöglichen. Zur Entscheidungsfindung hinsichtlich der Implementierungssprache des Systems wurde die bereits bestehende Umgebung in drei verschiedenen objektorientierten Sprachen neu implementiert um Einschätzungen hinsichtlich der Eignung dieser Umgebungen zur Implementierung des Systems treffen zu können. Die Wahl fiel letztlich aus Geschwindigkeitsgründen auf eine Implementierung in C++. Im Weiteren wurde der Entwurf der einzelnen Systemteile vorgestellt und deren Verbindungen beschrieben.

Eine Implementierung dieses Systems soll in Zukunft als Grundlage des Trafficaccountings der Forschungsgemeinschaft elektronische Medien e.V. dienen. Seit Beginn der Arbeit ergaben sich einige Änderungen an der vorhandenen Netzwerkstruktur, welche auch umfassende Änderungen an den technischen Grundlagen der Trafficauswertung nach sich ziehen. Daher zeigt die Flexibilität des Entwurfs bereits ihre Vorteile bei der Anpassung an die neuen Gegebenheiten.

## Literatur

- [CC6] *Catalyst 6000 Family - Multilayer Switches.*  
<http://www.cisco.com/univercd/cc/td/doc/pcat/ca6000.htm>.
- [CIN] *Cisco IOS NetFlow.*  
<http://www.cisco.com/warp/public/732/Tech/nmp/netflow/>.
- [CLI] *Common Language Infrastructure (CLI).*  
<ftp://ftp.ecma.ch/ecma-st/Ecma-335-part-i-iv.pdf>  
<ftp://ftp.ecma.ch/ecma-st/Ecma-335-part-v.pdf>.
- [CS] *C# Language Specification.*  
<ftp://ftp.ecma.ch/ecma-st/Ecma-334.pdf>.
- [DFN] *Deutsches Forschungsnetz e.V.*  
<http://www.dfn.de>.
- [DNT] *Introducing Microsoft .NET-connected Technologies.*  
<http://msdn.microsoft.com/netframework/productinfo/overview/default.asp>.
- [ECM] *ECMA - Standardizing Information and Communication Systems.*  
<http://www.ecma.ch>.
- [FeM] *Forschungsgemeinschaft elektronische Medien e.V.*  
<http://www.fem.tu-ilmenau.de>.
- [GCC] *GCC Home Page.*  
<http://gcc.gnu.org>.
- [JIF] *What Is an Interface?*  
<http://java.sun.com/docs/books/tutorial/java/concepts/interface.html>.
- [JVM] *The Java (TM) Programming Language Environment.*  
<http://java.sun.com/docs/white/langenv/Intro.doc2.html#379>.
- [LDA] *OpenLDAP.* <http://www.openldap.org>.
- [MON] *Home / Mono.*  
<http://www.go-mono.org>.
- [MS] *Microsoft Corporation.*  
<http://www.microsoft.com>.

- [PL] *Perl Mongers*.  
<http://www.perl.org>.
- [Squ] *Squid Web Proxy Cache*.  
<http://www.squid-cache.org>.
- [Str00] STROUSTRUP, BJARNE: *Die C++ Programmiersprache*. Addison-Wesley, Vierte Auflage, 2000.