Inhaltsverzeichnis

Kapitel

1

Implementierung der Features

Das Ergebnis aus [c1]dem Entwurf des Maifestes wird eine erste Implementation eines Testszenarios sein. Dies ist ebenfalls Bestandteil dieser Arbeit und wird mittels einiger der in Kapitel **??** definierten Features umgesetzt. In dieser Arbeit werden nicht alle genanten Features umgesetzt, sondern lediglich ein Szenario im System, welches bereits softwareseitig umsetzbar und testbar ist. Die benötigten Klassen hierzu werde[c2]n weitestgehend neu implementiert, lediglich Schnittstellen sind bereits im System enthalten und werden als gegeben betrachtet.

[c1] ~~der~~

[c2] *Text added.*

Einen generellen Überblick aller [c3]relevanten Komponenten und deren Kommunikationsschnittstellen, sog. Interfaces, werden i[c4]m folgenden Komponentendiagramm kurz dargestellt. Dies soll die Verbindungen zwischen den Komponenten, sowie die Vernetzung innerhalb der Unterkomponenten vereinfacht darstellen. [c5]

[c3] ~~R~~

[c4] ~~n F~~

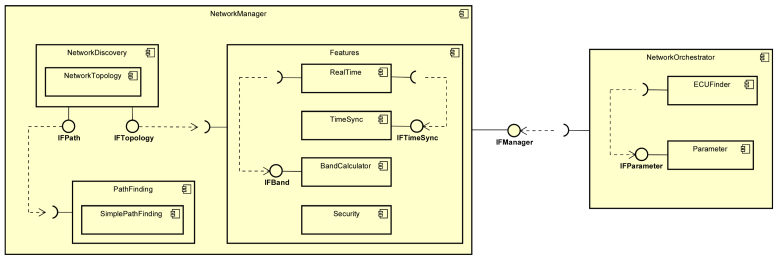

Abbildung 1: Komponentendiagramm Netzwerkmanager

[c5] Die Schrift in der Abbildung ist zu klein. Bitte die Abbildung vergrössern.

Da besonders im Bereich [c6]autonomes Fahren ein hoher Anspruch an die Ausfallsicherheit für eine Anwendungen im System gilt, werden im Nachgang noch mehrere Unit-Tests durch-

[c6] ~~A~~

geführt. Anhand dieser kann die Software auf verschiedene Eingabefälle geprüft werden. In [c7]den Unit-Test[c8]s werden isoliert[c9] Softwareeinheiten, wie einzelne Klassen[c10] mittels definierter Eingabe geprüft, um eine zuvor festgelegte Ausgabe zu erzeugen. Nur wenn dieses Ergebnis mit der gewünschten Ausgabe übereinstimmt, wird der Test als bestanden gewertet. Dies garantiert eine möglichst vielschichtige Abdeckung an Testfällen. Unit-Test[c11]s geben aber keine Auskunft über das Zusammenspiel von Klassen über Schnittstellen, dies muss im Nachgang noch manuell getestet werden.

[c7] *Text added.*
[c8] *Text added.*
[c9] ~~et~~
[c10] ~~,~~
[c11] *Text added.*

1.1 Testfall Ingress Filtering

Der gewählte Testfall bezieht sich auf das Ingress-Filtering eines Datenstroms, welche[c1]r bereits in Abschnitt **??** kurz angesprochen wurde. Dieser Filter wird empfängerseitig an den jeweiligen Ports, der sich in der Kommunikationsstrecke befindenden Geräten[c2] konfiguriert. Es handelt sich dabei um eine Filterung auf Layer[c3]-2[c4]-Eben[c5]e. Ingress-Filterring ist ähnlich aufgebaut wie eine Firewall. Im Initialzustand sind alle Kommunikationen verboten. Erst wenn Regeln definiert werden, die das [c6]Ein- oder [c7]Ausgehen von Nachrichten erlauben, kann eine Kommunikationsstrecke aufgebaut werden. So werden bei einer Ende-zu-Ende[c8]-Kommunikation lediglich die beiden Endpunkte zugelassen, die auch in den Regeln festgelegt sind. Hierbei werden die Partner anhand ihrer **MAC!** (**MAC!**)-Adressen in den Regeln definiert. So wird sichergestellt, dass nur die gewünschten Teilnehmer die Datenpakete empfangen und versenden können.

[c1] ~~s~~
[c2] ~~,~~
[c3] -
[c4] -
[c5] *Text added.*
[c6] ~~e~~
[c7] ~~a~~
[c8] -

Für die Regelung der Kommunikation haben sich hauptsächlich zwei Verfahren etabliert.

- *Application Whitelisting*
  Dieses Verfahren verbietet zuerst jegliche Kommunikation. Es müssen explizite Regeln für den Nachrichtenaustausch angelegt werden, um diese zu erlauben. In einem Netzwerk mit vielen Teilnehmern, bei denen nur wenige untereinander eine Verbindung aufbauen dürfen, ist dies

ein vielversprechender Ansatz. Es werden wenige Regel benötigt, da Verbote nicht definiert werden müssen.

- *Application Blacklisting*
  Beim *Blacklisting* ist grundsätzlich jegliche Verbindung zugelassen. Ein Verbot der Kommunikation zweier Partner kann nur durch eine Regel definiert werden. Dieser Ansatz[c1] ist nur dann sinnvoll, wenn es sehr viel Netzwerkteilnehmer gibt, bei denen die meisten untereinander [c2] kommunizieren dürfen. Es müssen somit nur wenige Regeln für Verbote definiert werden.

[c1] ~~macht nur Sinn~~

[c2] ~~K~~

Für dies[c3] e Arbeit wird [c4] der erste Ansatz verwendet, da es eine überschaubare Anzahl an Kommunikationspartnern gibt, die wiederum aus Sicherheitsgründen nicht alle untereinander kommunizieren dürfen. Ein weiter[c5] er Vorteil des Ansatzes ist[c6] es, dass durch das explizite Erlauben der Kommunikation das Fehlerrisiko einer Verfälschung der Daten durch andere Teilnehmer minimiert wird. Dies kann durch einen anderen Netzwerkteilnehmer erfolgen, der die Nachricht auch empfangen konnte und sie abgefälscht weiterleitet. Ein Verfälschen der Nachrichten kann sowohl mutwillig durch eine[c7] n manipulierten Netzwerkteilnehmer, als auch unwissentlich durch einen Defekt erfolgen. Beide Fälle müssen im System abgefangen werden, da durch sie ein Risiko entsteht.

[c3] *Text added.*

[c4] *Text added.*

[c5] *Text added.*

[c6] *Text added.*

[c7] *Text added.*

1.2 Konfiguration der Endgeräte

In der Umsetzung der Arbeit wird sich auf die Konfiguration der Switche in einer Kommunikationsstrecke beschränkt, weshalb in diesem Fall ein Endgerät [c8] mit einem Switch gleichzusetzen ist[c9]. Generell gibt es zwei Möglichkeiten der Konfiguration der Endgeräte. Die erste ist eine Erstellung einer Konfigurationsdatei im **XML!** (**XML!**)[c10] -Format, welche alle generierten Einstellungen für den jeweiligen Switch enthält. Durch [c11] das Aufspielen dieser Datei auf ein Endgerät, kann diese[c12] s neu konfiguriert werden. Die zweite Möglichkeit, auf die sich auch dies[c13] e Arbeit bezieht, ist das Nutzen eine im Projekt eigens für die Entwicklung geschriebene Library, welche alle Funktio-

[c8] *Text added.*

[c9] ~~mit einem Switch~~

[c10] -

[c11] ~~a~~

[c12] ~~r~~

[c13] *Text added.*

nen zur Konfiguration eines Endgerätes enthält. Diese wurde bereits im Vorfeld entwickelt und soll nun im Netzwerkmanager integriert und gesteuert werden. Hierzu wird von den in Abschnitt **??** definierten Feature[c14]s, eine Liste an Instruktionen erzeugt, welche am Ende einer Konfigurationsfindung von der Funktionalität der Library abgearbeitet werden soll.

[c14] ~~n~~

1.3 Umsetzung der Parametereinlesung

Wie in Kapitel **??** bereits erwähnt, bekommt der Netzwerkmanager seine Parameter mittels **JSON!** (**JSON!**)[c1]-File übermittelt. Dies[c2]e [c3]müssen im ersten Schritt ausgelesen[c4] und den entsprechenden Variablen im System zugeordnet werden, um im Nachgang damit zu arbeiten. Hierzu wurden die Klassen *json_translator* und *in_car_application* implementiert. Diese ermöglich[c5]en die Konvertierung der **JSON!**-Parameter in die benötigten Variablentypen, welche in der Klasse *in_car_application* definiert sind. Der *json_translator* ist auch in der Lage zu erkennen[c6], ob eine Anwendung mehrere Nachrichten versenden will und berechnet die Gesamtbandbreite der Applikation. Dies ist wichtig, damit der Netzwerkmanager später entscheiden kann, ob noch genügend Bandbreite für die Applikation vorhanden ist. In der Klasse *http_connector* ist die in Abschnitt **??** erwähne Technologie zum [c7]Hinzufügen einer neuen Applikation implementiert[c8]. [c9]Wird die Methode zum *adden* (dt. hinzufügen) einer neuen Anwendung zum System aufgerufen, starte[c10]t diese das Einlesen der Parameter über den *json_translator* und gibt als Rückgabewert ein Objekt vom Typ *InCarApplication* zurück. In dem Objekt sind alle im Applikationsmanifest definierten Parameter abgelegt und können im sog. Netzwerkstatus (Klasse *netwerk_flow*) jederzeit von ander[c11]en Klassen abgefragt werden. Für den weiteren Verlauf der Arbeit kann davon ausgegangen werden, dass eine Applikation vollständig in einem Objekt der Klasse *in_car_application* abgebildet ist.

[c1] -

[c2] *Text added.*

[c3] ~~muss~~

[c4] ~~werden,~~

[c5] ~~t~~

[c6] *Text added.*

[c7] ~~h~~

[c8] ~~;~~

[c9] ~~w~~

[c10] *Text added.*

[c11] *Text added.*

[c12]

[c12] Die Schrift in der Abbildung ist zu klein. Bitte die Abbildung vergrössern.

Der bereits erwähnte Netzwerkstatus spiegelt den aktuelle[c13]n Stand des Netzwerkes wieder. So sind dort beispielsweise Da-

[c13] *Text added.*

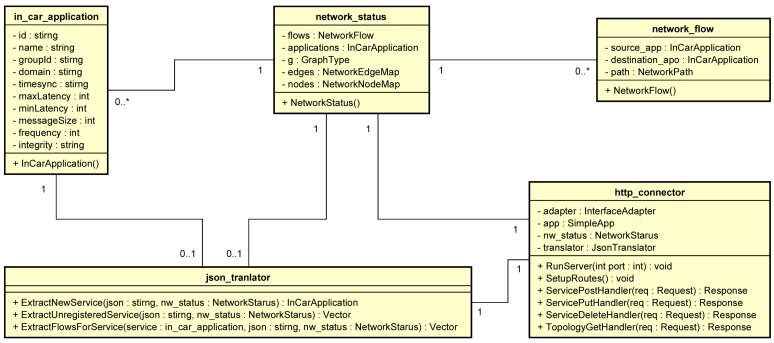

Abbildung 2: Klassendiagramm Parametereinlesung

tenflüsse, [c14]laufende Anwendungen und einige weitere Parameter enthalten. Der Status kann somit jederzeit von verschiedensten Funktionen abgefragt werden. Der Zusammenhang zwischen den verschieden Klassen[c15], die für die Parametereinlesung verantwortlich sind, ist im Klassendiagramm Abb. 2 dargestellt. Aufrufe von außen, wie beispielsweise durch den Orchestrator, erfolgen über den *http_connector*.

[c14] ~~L~~

[c15] *Text added.*

1.4 Umsetzung der Parameterauswertung mittels Feature

Nachdem die Parameter in den entsprechenden Variablen hinterlegt wurden, beginnt der Netzwerkmanager nacheinander die Features aus Abschnitt **??** abzuarbeiten. Diese liefern die gewünschten Konfigurationsbefehle für den Switch in Form einer Befehlsliste zurück. Im Testfall Ingress Filtering wird mit dem Feature zum Aufbau der Kommunikationsstrecke begonnen. Dazu werden die vom Orchestrator übergeben **ECU!** (**ECU!**)[c1]s als Start- und Endknoten verwendet.[c2] Die Kommunikationsstrecke[c3], über die [c4]die Daten der Applikation übermittelt werden, wird als Flow oder Datenflow bezeichnet. In einem Flow sind die sendende und [c5]die empfangende Applikation als Anfang[c6]s- und Endpunkt abgespeichert, anhand dieser kann später eine [c7]Zuordnung zwischen Applikation, Kommunikationspartnern und Datenflow erfolgen. Mithilfe der Klasse *simple_path_resolution* werden mögliche Pfade zwischen den Endpunkten gesucht. Ist ein geeigneter Pfad für einen Flow gefunden, werden beide in einer Map mit *Key-Value Pairs* abgelegt.

[c1] ,

[c2] Die Bedeutung des Satzes erschliesst sich mir nicht.

[c3] *Text added.*

[c4] *Text added.*

[c5] *Text added.*

[c6] *Text added.*

[c7] ~~zuordnen~~

So kann ein Pfad eindeutig einer Datenflow zugeordnet werden.

Ein Pfad kann aus mehren Edges bestehen, welche wiederum eine Teilstrecke ein[c1]es Pfades darstellt. Eine Edge entspricht einem Hop laut Netzwerkdefinition, d.[c2] h. dem Weg zwischen zwei Netzwerkgeräten. Edge[c3]s sind notwendig, da ein Pfad nur Anfang- und Endgerät kennt, jedoch nicht alle Teilnehmer[c4], die ein Datenpaket beim Versenden passieren muss. Eine Edge hat ebenfalls eine[c5]n Start- und [c6]einen Endk[c7]noten, diese werden im System als Nodes bezeichnet.

[c1] *Text added.*
[c2] -
[c3] *Text added.*
[c4] *Text added.*
[c5] *Text added.*
[c6] *Text added.*
[c7] ~~on~~

Nodes sind nichts anders als Geräte im Netzwerk wie beispielsweise ein Switch. [c8]Node[c9]s besitzen Attribute wie **MAC!**-Adresse, Ports und einem *Key-Value*[c10]*Pair*, welches eine Verbindung zwischen Ports und Node herstellt. Dieses Mapping ist notwendig um später, in der Verarbeitung der Features, alle dem Pfad zugehörigen Nodes den entsprechenden Ein- und Ausgangsport[c11]s zuordnen zu können, da ein Node in der Regel nicht nur einen, sondern mehrere Ports besitzt.

[c8] ~~In einem~~
[c9] *Text added.*
[c10] ~~-~~
[c11] *Text added.*

Um die Zusammenhänge der einzelnen Klassen zu verdeutlichen, wurde in Abb. 3 ein Klassenmodel erstellt. Diese enthält in den einzelnen Komponenten lediglich die für die Arbeit relevanten Variablen und Funktionen.

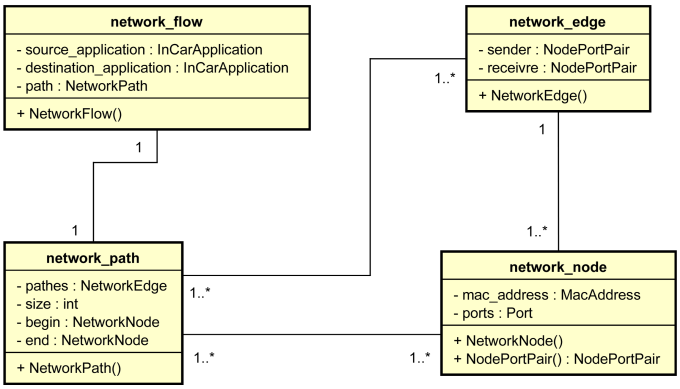

Abbildung 3: Klassendiagramm Netzwerkübersicht

Sobald die Pfadfindung abgeschlossen ist, werden aus der Klasse
feature_based_configuration_resolution heraus, nacheinander die einzelnen Features aus [c12]Abschnitt **??** aufgerufen. So wird das Feature Kommunikation, welches Ingress Filtering beinhal-

[c12] ~~Kapitel~~

tet und in der Klasse *feature_communication_ingress_filtering* implementiert ist, ebenfalls hier durchlaufen. Dieses sucht, mittels der übergeben[c13]den Applikation, dem Flow-Pfad[c14]- Mapping und dem aktuellen Netzwerkstatus de[c15]s [c16]dazu[c17] gehörigen Pfad[c18]für die Anwendung.[c19] Ist dieser gefunden, wird über alle Edges des Pfades iteriert und für alle Nodes auf [c20]der Empfängerseite die entsprechenden Portregelungen für das Ingress Filtering in der in Abschnitt 1.2 erwähnten Liste mit Instruktion abgelegt. Sobald alle Edges durchlaufen sind, gilt das Feature als abgearbeitet und es kann mit dem nächsten Feature begonnen werden. Sind alle abgearbeitet wird die Instruktion[c21]sliste an die Schnittstelle zur Konfiguration der Endgeräte[c22] übergeben. Dies konfiguriert mit den entsprechenden Funktionen, aus der erwähnten Library, das Endgerät. Im Fall Ingress Filtering werden die Portregelungen, wie Filtereigenschaften für Datenpakete, mit den entsprechenden Ein- und Ausgangsadressen der Netzwerkgeräte, übermittelt.

c13 *Text added.*
c14 -
c15 ~~n~~
c16 *Text added.*
c17 *Text added.*
c18 ~~es~~
c19 Was sucht hier was?
c20 *Text added.*
c21 ~~L~~
c22 ~~n~~

1.4.1 Testing der Implementation Ingress Filtering

Um sicher zu stellen[c1], das[c2]s das Konzept der Features[c3] zur automatischen Konfiguration[c4] richtig arbeitet, müssen alle Teilelemente der Implementation mittel Unittests überprüft werden. Hierzu wurde zuerst eine Testklasse zu[c5]r Parameter-Einlesung geschrieben, welche der Klasse *json_translator* ein Manifest im **JSON!**[c6]-Format übergibt und die zurückgegebenen Ist-Werte mit den zuvor definierten Soll-Werten vergleicht. Dies soll sicherstellen[c7], das[c8]s alle Parameter aus den Manifest in den richtigen Variablen des Netzwerkmanager[c9]s landen.

Die zweit[c10]e Testklasse ist speziell für den Testfall Ingress Filtering implementiert. Es besitzt ein[c11]e zuvor definierte Netzwerktopologie mit drei Netzwerkknoten. Anhand dieser wird der in Abschnitt 1.4 [c12]beschriebene Prozess durchlaufen. Nach der Abbildung der Topologie in der Software, mittels Flows, Pfaden und Edges, wird die automatisierte Portregelungs-Konfiguration in der Klasse *feature_communication_ingress_filtering* angestoßen. Nach dem vollständigen Durchlauf des Features[c13] wird

c1 *Text added.*
c2 *Text added.*
c3 ~~,~~
c4 ~~,~~
c5 *Text added.*
c6 -
c7 *Text added.*
c8 *Text added.*
c9 *Text added.*
c10 *Text added.*
c11 *Text added.*
c12 *Text added.*
c13 ~~,~~

die erstellte Instruktionsliste mit de[c14]m erwarteten Ergebnis verglichen.

Beide Test konnten positiv bewertet werden, was sicherstellt[c1], dass das Konzept der automatischen Netzwerkkonfiguration auf diese Weise umsetzbar ist. Ein nächster Schritt ist es die Konfiguration auf eine[c2]n realen Switch weiter zu leiten, um auch [c3]hardwareseitig noch einige Tests durchführen zu können. Dies ist jedoch nicht mehr Teil dieser Arbeit, wird aber bereits im Projekt A3F weiter verfolgt.

[c14] ~~r~~

[c1] ~~das,~~

[c2] *Text added.*

[c3] ~~H~~

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] *Bridges and Bridged Networks— Amendment 28: Per-Stream Filtering and Policing*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8064221. Abgerufen: 05.08.2018.

[2] *Dan Des Ruisseaux. Designing a Deterministic Ethernet Network (Whitepaper): Industrial Communication. Hrsg. von Schneider Electric Industries SAS*. http://www.schneider-electric.co.uk/documents/solutions/process-automation/open-connectivity/Designing%20a%20Deterministic%20Ethernet%20Network.PDF. Abgerufen: 23.05.2018.

[3] *Echtzeitanwendungen mit Automotive Ethernet*. https://www.elektroniknet.de/elektronik-automotive/bordnetz-vernetzung/echtzeitanwendungen-mit-automotive-ethernet-113002.html. Abgerufen: 20.06.2018.

[4] Eckert, Claudia: *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Walter de Gruyter GmbH & Co KG, Berlin, 9. Auflage, 2014, ISBN 978-348-685-916-4.

[5] *Heterogeneous Networks for Audio and Video: Using IEEE 802.1 Audio Video Bridging*. https://ieeexplore.ieee.org/document/6595589/. Abgerufen: 10.07.2018.

[6] *IEEE 802 Numbers*. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml. Abgerufen: 30.05.2018.

[7] *IEEE 802.1AS-Rev*.
https://1.ieee802.org/tsn/802-1as-rev/.
Abgerufen: 16.04.2018.

[8] *IEEE 802.1CB*.
https://1.ieee802.org/tsn/802-1cb/.
Abgerufen: 16.04.2018.

[9] *IEEE 802.1CM*.
https://1.ieee802.org/tsn/802-1cm/.
Abgerufen: 16.04.2018.

[10] *IEEE 802.1Qbu*. http:
//www.ieee802.org/1/pages/802.1bu.html.
Abgerufen: 16.04.2018.

[11] *IEEE 802.1Qbv*. http:
//www.ieee802.org/1/pages/802.1bv.html.
Abgerufen: 16.04.2018.

[12] *IEEE 802.1Qca*. http:
//www.ieee802.org/1/pages/802.1ca.html.
Abgerufen: 16.04.2018.

[13] *IEEE 802.1Qcc*.
https://1.ieee802.org/tsn/802-1qcc/.
Abgerufen: 16.04.2018.

[14] *IEEE 802.1Qci*.
https://1.ieee802.org/tsn/802-1qci/.
Abgerufen: 16.04.2018.

[15] *IEEE 802.1Qcr*.
https://1.ieee802.org/tsn/802-1qcr/.
Abgerufen: 16.04.2018.

[16] *IEEE 802.3*. http://www.ieee802.org/3/.
Abgerufen: 09.04.2018.

[17] *IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Security*.
http://www.ieee802.org/1/files/public/
docs2010/new-seaman-1AE-markup-for-

gcm-aes-256-0710-v2.pdf. Abgerufen: 30.05.2018.

[18] James F. Kurose, Keith W. Ross: *Computer networking: A top-down approach*. Pearson Education, Inc, New Jersey, 6. Auflage, 2013, ISBN 978-0-13-285620-1.

[19] Matheus Kirsten, Thomas Königseder: *Automotive Ethernet*. Cambridge University Press, Cambridge, United Kingdom, 2. Auflage, 2017, ISBN 978-110-718-322-3.

[20] Meyer, Gereon: *Advanced Microsystems for Automotive Applications 2012*. Springer-Verlag, Berlin Heidelberg, 6. Auflage, 2012, ISBN 978-3-642-29672-7.

[21] Meyer, Philipp: *Extending IEEE 802.1 AVB with time-triggered scheduling: A simulation study of the coexistence of synchronous and asynchronous traffic“*. IEEE, Boston, MA, USA, 1. Auflage, 2014, ISBN 978-1-4799-2687-9.

[22] Peter Mandl, Andreas Bakomenko, Johannes Weiß: *Grundkurs Datenkommunikation*. Vieweg+Teubner GWV Fachverlage GmbH, Wiesbaden, 1. Auflage, 2008, ISBN 978-3-8348-0517-1.

[23] R. Ramaswamy, Ning Weng, T. Wolf.: *Characterizing network processing delay*. IEEE Global Telecommunications Conference, 2004, GLOBECOM '04. IEEE, 1. Auflage, 2004, ISBN 0-7803-8794-5.

[24] *Road vehicles — Functional safety*. https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en. Abgerufen: 11.07.2018.

[25] Ross, Hans Leo: *Functional Safety for Road Vehicles*. Springer-Verlag, Berlin Heidelberg, 1. Auflage, 2016, ISBN 978-3-319-33360-1.

[26] Tanenbaum, Andrew S.: *Computer Network*. Pearson Education, Boston, 5. Auflage, 2011, ISBN 978-0-13-212695-3.