Masterarbeit

Entwicklung einer Client-/Server-basierten Software für die Prüfung der Vertrauenswürdigkeit von Netzwerkkomponenten

von Martin Schmiedel

Sommersemester 2006

Fachhochschule Hannover
Fachbereich Informatik

Autor:
Martin Schmiedel
Freytagstr. 6
30169 Hannover

Fachhochschule Hannover
Fachbereich Informatik
Studiengang: Master of Science Angewandte Informatik

Erstprüfer:
Prof. Dr. Josef von Helden
Fachhochschule Hannover
Fachbereich Informatik
Ricklinger Stadtweg 120
30459 Hannover

Zweitprüfer:
Prof. Dr. Stefan Wohlfeil
Fachhochschule Hannover
Fachbereich Informatik
Ricklinger Stadtweg 120
30459 Hannover

Eigenständigkeitserklärung

Hiermit versichere ich, Martin Schmiedel, dass ich diese Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Hannover, 21. August 2006 Martin Schmiedel

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung und Ziel

Die Sicherheit von Endgeräten ist heute eine der wichtigsten Herausforderungen im Bereich der IT-Sicherheit. Wo Netze zunehmend öffentlich zugänglich und damit eine Vielzahl an Geräten Teil dieser Netze werden, können klassische Maßnahmen, wie z.B. zentrale Firewalls keine ausreichende Sicherheit gegen die heute vielfältigen Gefährdungen gewährleisten. Die Sicherheitsbemühungen müssen sich vielmehr an die Ränder des Netzes verlagern. Das bedeutet, dass die Integrität und damit die Vertrauenswürdigkeit der einzelnen Teilnehmer des Netzes überprüft werden muss.

Mehrere Architekturen behandeln dieses Thema, z.B. Ciscos Network Admission Control (NAC)[1], Microsofts Network Access Protection (NAP)[2] und Trusted Network Connect (TNC) von der Trusted Computing Group, welches ein sehr vielversprechender Ansatz ist.

Hauptziel dieses Masterprojekts ist daher die Implementierung eines TNC Clients (TNCC) und eines TNC Servers (TNCS) als Open Source Software, deren Aufgaben in den TNC-Spezifikationen detailliert definiert sind und welche auf Netzwerkkomponente (TNCC) und zentralem Server (TNCS) im Zusammenspiel für die Integritätsprüfung der Netzwerkkomponente zuständig sind.

Neben der reinen Umsetzung der TNC-Standards besteht die Herausforderung dieser Arbeit darin, Wege zu finden, TNC in bestehende Infrastrukturen (Switches und Betriebssysteme) und Standards zu integrieren und praktikable Lösungen für die Anbindung an die netzwerknahe Schicht mit 802.1x oder RADIUS zu finden. Wenn nötig, muss diese Schicht ebenfalls implementiert werden. Dabei soll eine leicht erweiterbare und portierbare Architektur entstehen. Zudem müssen Lücken in den Spezifikationen gefüllt werden und dabei die praktische Umsetzbarkeit der Spezifikationen validiert werden. Funktionalitätstests der Implementierung (im Zusammenspiel mit anderen TNC-Implementierungen) komplettieren die Arbeit.

[1] siehe: http://www.cisco.com/en/US/netsol/ns466/networking_solutions_package.html
[2] siehe: http://www.microsoft.com/technet/itsolutions/network/nap/default.mspx

1.2 Gliederung

Diese Arbeit gliedert sich in vier Teile: Im ersten Teil (Kapitel 2 bis 3) wird auf die für die Implementierung wichtigen Grundlagentechnologien, wie EAP, 802.1x und RADIUS eingegangen. Zudem wird die TNC-Architektur detailliert vorgestellt.

Im zweiten Teil (Kapitel 4 bis 7) wird gezeigt, wie die TNCC und TNCS Software entworfen wird, welche Problemstellungen dabei existieren und welche Lösungsalternativen für diese Aufgaben bestehen. Wie der Software-Entwurf mit den gewählten Lösungsansätzen konkret aussieht, wird detailliert beleuchtet.

Im dritten Teil der Arbeit (Kapitel 8 bis 11) steht die Implementierung im Fokus. Dabei wird exemplarisch gezeigt, wie der Entwurf konkret umgesetzt wird. Es werden die verwendeten und angepassten Entwicklungsumgebungen, sowie das Build-Verwahren und interessante Implementierungsdetails gezeigt. Außerdem wird die Tragfähigkeit der Implementierung in einer Testumgebung überprüft.

Zum Abschluss wird in Kapitel 12 eine Zusammenfassung und ein Ausblick gegeben.

1.3 Über die Masterarbeitsstelle

Diese Masterarbeit ist im Rahmen des fachhochschulinternen Projekts **TNC@FHH**[3] entstanden. Da die Fachhochschule Hannover während des Projekts den Status „Liasion Member“ in der TCG und der TNC Subgroup erhielt, wurde diese Arbeit von Mitgliedern der TCG unterstützt. So stellte z.B. Hewlett-Packard dem Projekt einen leistungsfähigen Switch zur Verfügung.

Im Rahmen dieses Projektes entstand eine zweite Arbeit von Daniel Wuttke [Wut06], dessen Aufgabenfeld auf der Integrity Measurement Ebene lag. So konnte in enger Kooperation mit ihm die Tragfähigkeit der hier vorgestellten Lösungen sichergestellt und überprüft werden.

1.4 Typographische Konventionen

In dieser Masterarbeit werden die folgenden Schreibweisen verwendet:

Serifenlos:	Bezeichnungen werden bei ihrer ersten Nennung bzw. wenn sie erklärt werden, einmal in serifenloser Schrift dargestellt.
`Typewriter`:	Komponenten-, Klassen- und Methodenbezeichnungen, sowie Dateinamen und -pfade werden in `Typewriter` dargestellt.
Kursiv:	Paket-Bezeichnungen (z.B. *EAPOL-Start* oder *Radius Access-Challenge*) werden *kursiv* gedruckt.
Fett:	Sonstige Hervorhebungen werden **fett** gedruckt.

[3] siehe: http://www.inform.fh-hannover.de/de/forschung/forschungsprojekte/tnc/index.html

2 Portbasierte Zugriffskontrolle mit 802.1x

Bevor im nächsten Kapitel auf die TNC-Architektur eingegangen wird, soll an dieser Stelle der Standard 802.1x vorgestellt werden, welcher die Authentisierung von Endgeräten im Netz ermöglicht. Die im Rahmen dieser Arbeit entstandene TNC-Implementierung nutzt 802.1x für die konkrete Kommunikation auf Netzwerkebene.

Da sich die Implementierung auf drahtgebundene Netze (Ethernet IEEE 802.3 [1]) beschränkt, bleiben im Folgenden dieser Arbeit drahtlose Netze (WLANs IEEE 802.11 [2]) außer Betracht, auch wenn 802.1x und TNC hierfür Lösungen vorsehen.

2.1 Idee von 802.1x

In herkömmlichen LANs können sich beliebige Endgeräte einkoppeln, sobald eine Netzwerkdose den Zugriff auf ein LAN bietet. Damit stehen die Netze offen für jegliche Geräte, welche sich Zugriff verschaffen können. Dieser Umstand birgt ein Sicherheitsrisiko für das gesamte Netz, weil so Geräte in das Netz gelangen können, welche nicht unter Kontrolle der Netzwerkadministratoren stehen und absichtlich (durch Angriffe) oder unbeabsichtigt (durch eigene Schwachstellen) das Netz gefährden.

Um nur authentisierten Geräten den Zugang zu einem Netz zu ermöglichen, wurde 802.1x entworfen. 802.1x ist ein IEEE-Standard [IEE04], der eine port-basierte Zugriffskontrolle in 802-LAN-Infrastrukturen ermöglicht.

Port-basiert bedeutet, dass die Zugriffskontrolle direkt dort ansetzt, wo ein Client auf das Netz zugreifen will, also z.B. am Switch-Port, in den das Gerät eingeklinkt wird. Der Client bekommt nicht automatisch Zugriff auf das Netz, sondern muss sich vorab authentisieren, bevor der Port freigeschaltet wird. An diesem Authentisierungsprozess sind drei Komponenten beteiligt (vgl. [IEE04, S. 12]):

- Der Client, welcher Zugriff auf das Netz erlangen möchte, wird von der 802.1x-Terminologie als Supplicant bezeichnet.
- Der Authenticator (z.B. ein Switch) blockiert den Zugriff des Supplicant bis zur erfolgreichen Authentisierung.

[1] Entwickelt von der IEEE-802.3 Working Group, siehe: http://grouper.ieee.org/groups/802/3/
[2] Entwickelt von der IEEE-802.11 Working Group, siehe: http://grouper.ieee.org/groups/802/11/

- Der Authentication Server führt die eigentliche Authentisierung des Supplicant durch und gibt dem Authenticator den Befehl zur Freischaltung des Ports bei erfolgreicher Authentisierung.

2.2 Ablauf der Authentisierung mit 802.1x

Die prinzipielle Interaktion zwischen den Komponenten ist in Abbildung 2.1 abgebildet.

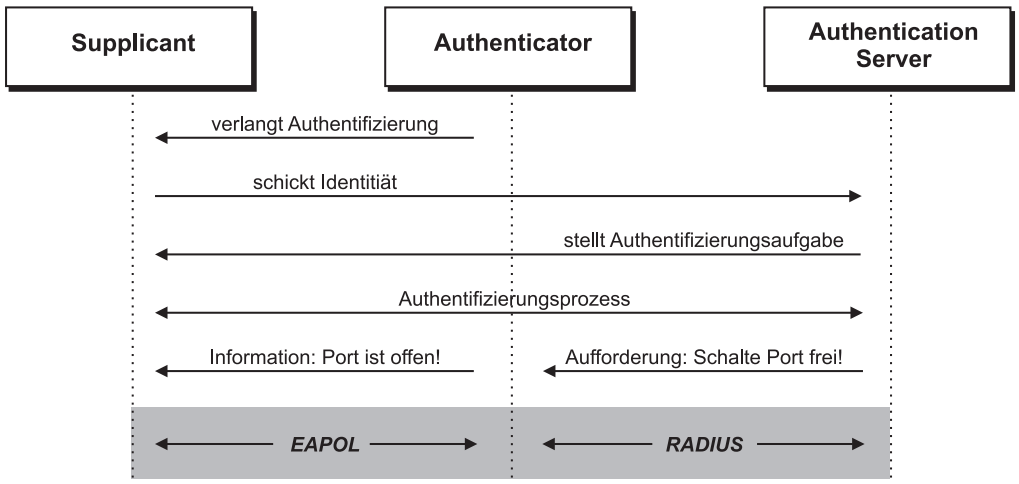


Abbildung 2.1: Prinzipieller Ablauf der Authentisierung eines Supplicants unter 802.1x

Der Authenticator blockt den Zugriffsversuch des Supplicant ab, wenn dieser auf das Netz zugreifen will und schickt ihm eine Authentisierungsaufforderung. Vorausgesetzt der Client unterstützt 802.1x, dann schickt dieser seine Identität an den Authenticator, mit der er sich authentisieren will. Der Authenticator leitet die Identität an den Authentication Server weiter und agiert jetzt nur noch als Durchgangsstation für die Nachrichten zwischen Supplicant und Authentication Server. Daraufhin stellt der Authentication Server dem Supplicant eine Authentisierungsaufgabe, die im folgenden Authentisierungsprozess (abhängig vom eingesetzten Authentisierungsverfahren) vom Supplicant zu lösen ist.

Wenn sich der Supplicant dem Authentication Server gegenüber erfolgreich authentisiert hat, gibt letzerer dem Authenticator den Befehl, den Port freizuschalten. Eine Informationsnachricht vom Authenticator informiert den Supplicant über den neuen Zustand. Der Supplicant erhält nun Zugriff auf das Netz.

Für die Kommunikation zwischen Supplicant und Authenticator wird in 802.1x EAPOL definiert (vgl. [IEE04, S. 23ff]), für die Kommunikation zwischen Authenticator und Authentication Server wird das RADIUS-Protokoll vorgeschlagen; dies ist

aber vom Standard nicht zwingend vorgeschrieben (vgl. [IEE04, S. 37]).[3] Der Switch muss die Nachrichten zwischen diesen beiden Formaten konvertieren. EAPOL und RADIUS werden im Folgenden näher vorgestellt.

2.3 EAPOL

EAPOL ist im 802.1x-Standard definiert. Es steht für EAP over LAN und erläutert, wie EAP im Ethernet-basierten LAN zur Authentisierung genutzt werden kann. Dabei ist beschrieben, wie die Authentisierung im LAN abläuft und wie die EAP-Pakete dafür innerhalb von Ethernetpaketen verpackt werden.

2.3.1 EAP als Basis

EAP steht für „Extensible Authentication Protocol“ und ist in einer Vielzahl von RFCs beschrieben. Das Basisdokument dazu in der aktuellsten Version ist RFC 3748[4]. Wie aus dem Namen bereits hervorgeht, beschreibt EAP ein Protokoll, das ermöglicht, mit unterschiedlichsten Verfahren Clients zu authentisieren. Dazu sind ein Nachrichtenformat und viele Spezialisierungen dieses Formats für die einzelnen Verfahren, sowie der Authentisierungsablauf definiert.

Abbildung 2.2 zeigt, wie anhand des sehr einfachen Authentisierungsverfahrens „MD5“[5] die Authentisierung mit EAP prinzipiell abläuft.

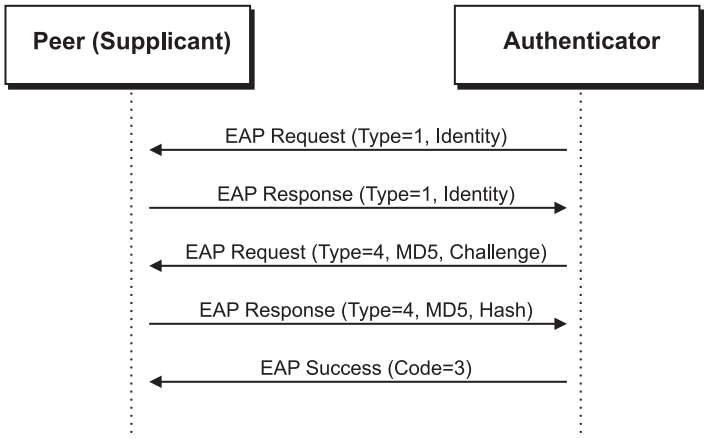


Abbildung 2.2: Ablauf einer Authentisierung in EAP am Beispiel von EAP-MD5 (vgl. [Aur05, Abb., S. 280])

[3] Alternativ ist auch ein Diameter-Server einsetzbar (siehe RFC 3588).
[4] http://www.ietf.org/rfc/rfc3748.txt
[5] Der Hash-Algorithmus MD5 wird z.B. in http://de.wikipedia.org/wiki/MD5 erklärt. Der MD5-Algorithmus innerhalb von EAP wird in RFC3748 erläutert.

Der Peer (entspricht dem Supplicant in 802.1x) möchte sich bei einem Authenticator authentisieren. Dazu schickt der Authenticator ein *EAP Request* an den Peer und fordert diesen auf, seine Identitität preiszugeben. Der Peer antwortet stets mit einer *EAP Response*; dieses Mal schickt er dabei seine Identität zurück, anhand derer er authentisiert werden will[6]. Beim MD5-Verfahren schickt der Authenticator nun eine Challenge (z.B. Zufallszahl) per *EAP Request* an den Peer, aus welcher der Peer per MD5 zusammen mit seinem Passwort und seiner Identität einen Hashwert erstellt und als *EAP Response* zurückschickt. Der Authenticator prüft diesen Hashwert und sendet daraufhin eine *EAP Success*-Nachricht (oder bei gescheiterter Authentisierung eine *EAP Failure*-Nachricht) an den Peer zurück.

2.3.2 Kapselung von EAP im LAN

Für den Einsatz des EAP-Protokolls im Ethernet-LAN definiert 802.1x einen Header, der zwischen den Ethernet-Header und das eigentliche EAP-Paket eingefügt wird. Ohne an dieser Stelle auf den Aufbau im Detail einzugehen, soll kurz auf zwei Packet-Typen eingegangen werden, welche über ein Flag in diesem Header definiert werden können und im Kontext dieser Arbeit von Bedeutung sind:

EAP-Packet Es wird ein EAP-Paket als Nutzlast übertragen.

EAPOL Start Der Supplicant kann dieses Paket senden, um den Authentisierungsprozess anzustoßen, wenn der Authenticator nicht automatisch damit startet. Es wird kein klassisches EAP-Paket übertragen.

2.4 Authentisierung und Autorisierung mit RADIUS

2.4.1 Idee von RADIUS

RADIUS steht für „Remote Authentication Dial In User Service“ und ist ein Standard, welcher aktuell im RFC 2865[7] definiert ist.[8]

Das RADIUS-Protokoll ermöglicht den Transport von Authentisierungs-, Authorisierungs- und Konfigurationsinformationen zwischen einem Network Access Server (NAS), welcher die an ihn gestellten Verbindungen authentisieren möchte, und einem Authentication Server (AS). Der NAS agiert als RADIUS-Client, der AS als RADIUS-Server.[9]

[6] Es gibt auch Verfahren, in denen diese Identität für den Authentisierungsprozess keine Rolle spielt. Trotzdem muss eine „Dummy“-Identität geschickt werden.
[7] http://www.ietf.org/rfc/rfc2865.txt
[8] Der erste RFC zu RADIUS (RFC 2058) stammt aus dem Jahr 1997.
[9] Eine detaillierte Einführung zu RADIUS findet sich in [Has02].

Darüber hinaus befassen sich noch eine Reihe weiterer RFCs mit Erweiterungen zu RADIUS, wie z.B. RFC 2869[10]. In diesem RFC wird u.a. beschrieben, wie EAP-Nachrichten in RADIUS gekapselt werden. So kann RADIUS dazu genutzt werden, auf unterschiedlichste Art die Authentisierung und Authorisierung durchzuführen. Sogenannte RADIUS-Attribute, in denen unterschiedlichste Informationen abgelegt werden können, ermöglichen diese flexible Erweiterung.

2.4.2 Kommunikation auf Basis von RADIUS

RADIUS definiert ein Paket-Format, mit dem Nachrichten auf Basis von UDP[11] zwischen Authentication Server und Network Access Server ausgetauscht werden. Abbildung 2.3 zeigt, wie die Kommunikation mit RADIUS abläuft und welche Pakettypen dabei ausgetauscht werden.

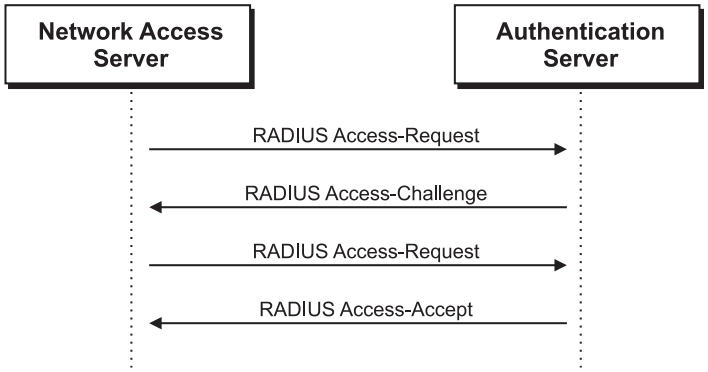


Abbildung 2.3: Ablauf der Authentisierung mit RADIUS

Wenn der NAS Anfragen von Clients erhält, agiert er selbst als RADIUS-Client und schickt einen *RADIUS Access-Request* an den AS in seiner Rolle als RADIUS-Server mit den Authentisierungsinformationen seines Clients in RADIUS-Attributen verpackt. Der AS wertet diese Anfrage aus und sendet ein *RADIUS Access-Challenge* zurück, in der weitere Authentisierungs- oder Autorisierungsinformationen erfragt werden. Nachdem die Antwort darauf vom NAS per RADIUS Access-Request wieder beim AS eingetroffen ist, entscheidet er darüber, ob der Client zugelassen wird und schickt dann eine *Access Accept*- oder *Access Reject*-Nachricht zurück, in denen Informationen darüber enthalten sind, wie sich der NAS gegenüber dem anfragenden Client verhalten soll. Der Authentisierungsprozess kann natürlich auch über mehrere Runden gehen.

[10] http://www.ietf.org/rfc/rfc2869.txt
[11] User Datagram Protocol; für eine Beschreibung von UDP siehe z.B. [Tan00, S. 576]

2.4.3 OpenSource-Implementierung: FreeRadius

Es existieren eine Vielzahl von RADIUS-Implementierungen. Zum Test der grundlegenden Funktionalitäten und zur späteren Erweiterung wird im Rahmen dieser Arbeit FreeRadius[12] eingesetzt. Weil es sich bei FreeRadius um eine OpenSource-Implementierung des RADIUS-Protokolls handelt und es sehr modular aufgebaut ist, eignet es sich besonders gut zur Erweiterung um TNC-Fähigkeiten. FreeRadius ist aus Modulen aufgebaut, die einzelne Authentisierungsverfahren realisieren und auch einzeln kompilierbar sind. Diese Module werden dynamisch ins FreeRadius-Framework geladen und können um weitere Module ergänzt werden (vgl. [Unk06]). FreeRadius hat in der eingesetzten Version 1.1.0 bereits einige Module implementiert, so auch für die Authentisierung mit EAP. Das EAP-Modul unterstützt dabei Authentisierungsmethoden wie MD5 (Message Digest Algorithm 5), TTLS[13] oder PEAP[14] und ist ebenfalls einfach erweiterbar (vgl. [Rag05]).

Fluss durch FreeRadius und EAP-Modul

Wenn ein *RADIUS Access-Request* FreeRadius erreicht, gelangt es zunächst ins generelle FreeRadius-Framework (siehe Abbildung 2.4). Anhand der mitgeschickten Identität und Einstellungen des Servers ermittelt der Server, welches das passende Modul ist und überträgt im Wesentlichen die RADIUS-Attribute des Requests an das entsprechende Modul. Wenn eine EAP-Nachricht im *Access-Request* übermittelt wurde, gelangt diese damit in das EAP-Modul, weil sie in den RADIUS-Attributen kodiert ist.

Das EAP-Modul extrahiert die *EAP Response* aus den RADIUS-Attributen und ermittelt anhand des EAP-Types das zugehörige „Teil-Modul“, welches für den entsprechenden Typen verantwortlich zeichnet. Eine Datenstruktur mit Namen `EAP_HANDLER` (mehr dazu in Kapitel 9.2.5) enthält die *EAP Response* und wird dem Typ-Modul zur Verfügung gestellt. Das Typ-Modul verarbeitet die *EAP Response*, generiert einen neuen *EAP Request* und steckt diesen in den `EAP_HANDLER`. Das EAP-Modul entnimmt aus dem `EAP_HANDLER` die *EAP Response*, fügt eine gültige EAP-ID ein und generiert die entsprechenden RADIUS-Attribute. Abschließend nimmt das FreeRadius-Framework diese RADIUS-Attribute entgegen und fügt sie in ein *RADIUS Access-Challenge*-Paket ein, das nun an den RADIUS-Client zurückgeschickt wird.

[12] http://www.freeradius.org/

[13] Tunneled Transport Layer Security: eine Einführung bietet http://www3.ietf.org/proceedings/02mar/slides/eap-1/

[14] Protected Extensible Authentication Protocol: eine Beschreibung von PEAP gibt [Aur05, S. 285f]

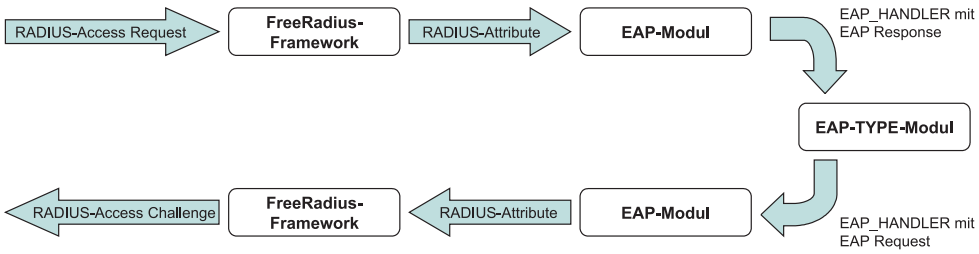


Abbildung 2.4: Datenfluss durch die Komponenten von FreeRadius bei einem eintreffenden RADIUS-Paket

2.5 Zusammenspiel von EAPOL und RADIUS in 802.1x

Nachdem EAPOL und RADIUS einzeln vorgestellt wurden, wird nun gezeigt, wie sie im Kontext von in 802.1x zusammen eingesetzt werden. In Abbildung 2.5 wird dieser Sachverhalt gezeigt. Dabei werden im Wesentlichen die Kommunikationspfeile aus Abbildung 2.1 (Kapitel 2.2) um die eingesetzten Technologien ergänzt.

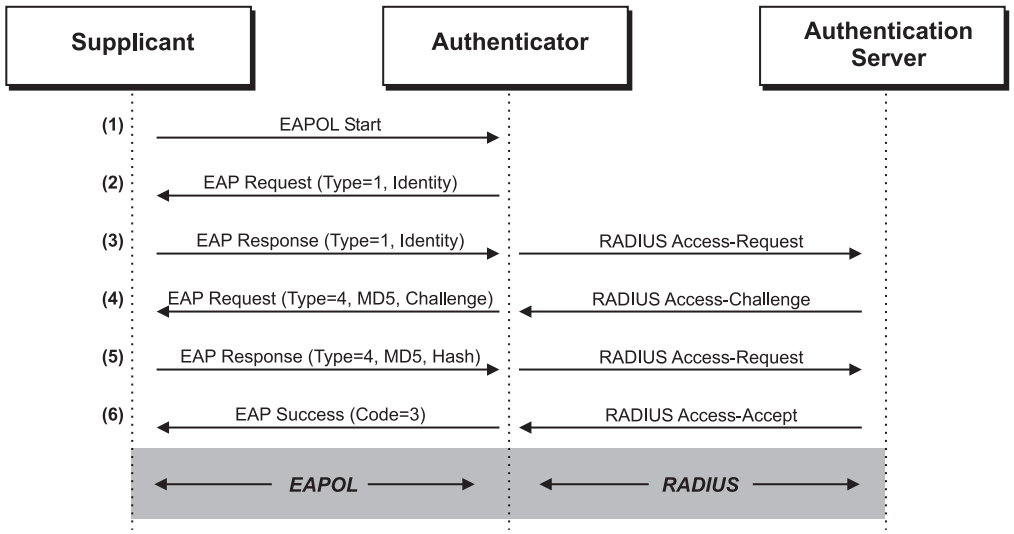


Abbildung 2.5: Zusammenspiel von EAPOL und RADIUS zur Authentisierung in 802.1x

1. Wenn sich der Supplicant physikalisch ins Netz einklinkt, schickt er ein *EAPOL Start* an den Authenticator, um ihn über seinen Verbindungswunsch zu be-

nachrichtigen. [15]

2. Der Authenticator bemerkt den neuen Client an einem seiner Ports und schickt eine EAPOL-Nachricht mit *EAP Request* an den Supplicant, das die Aufforderung an den Supplicant enthält, seine Identität mitzuteilen.

3. Der Supplicant antwortet dem Authenticator, indem er seine Identität als *EAP Response* übermittelt. Der Authenticator nimmt diese entgegen, packt das EAP-Paket aus dem EAPOL-Paket aus und schickt es als RADIUS-Attribut verpackt in einem *RADIUS Access-Request* an den im Switch eingetragenen RADIUS-Server (in seiner Rolle als AS).

4. Der RADIUS-Server erkennt anhand seiner Einstellungen, des verpackten EAP-Pakets und der übermittelten Identität, dass die Authentisierungsmethode „EAP“ benutzt werden soll (im Beispiel mit MD5-Challenge) und schickt die MD5-Challenge als *EAP Request* verpackt in einem *RADIUS Access-Challenge* an den Authenticator. Dieser packt das EAP-Paket aus dem empfangenen *RADIUS Access-Challenge* aus und überträgt den *EAP Request* als EAPOL-Paket an den Supplicant.

5. Der Supplicant generiert den MD5-Hashwert und schickt diesen als *EAP Response* in einem EAPOL-Paket zurück zum Authenticator, der wiederum daraus ein *RADIUS Access Request* macht und dieses an den AS weiterleitet.

6. Der AS überprüft den Hashwert und schickt, wenn die Authentisierung erfolgreich war, ein *RADIUS Access-Accept*, sonst ein *RADIUS Access Reject* an den Authenticator. Daraufhin weiß dieser, ob er den Port freischalten soll, und informiert den Supplicant per *EAP Success* bzw. *EAP Failure* über diese Entscheidung.

Damit diese 802.1x-Kommunikation reibungslos funktioniert, werden selbstverständlich einige Voraussetzungen an die Teilnehmer gestellt:

- Der Authenticator (im LAN üblicherweise der Switch) muss 802.1x-kompatibel sein, also vor allem die Fähigkeit besitzen, Ports zu blocken und freizugeben, sowie die Wandlung der Pakete von EAPOL in RADIUS (und umgekehrt) beherrschen.

- Der Client, der Zugang zum Netz erhalten möchte, muss 802.1x-kompatibel sein, d.h. als Supplicant interagieren können.

- Der Client und der RADIUS-Server müssen mindestens eine gemeinsame EAP-Authentisierungsmethode unterstützen. Der Authenticator ist davon ausgenommen, weil für ihn der Inhalt der EAP-Pakete von keiner Bedeutung ist.

[15] Dieser Schritt ist optional, weil viele Switches (in ihrer Rolle als Authenticator) automatisch mit Schritt 2 beginnen, wenn sie die neue physikalische Verbindung registrieren.

3 Integritätsprüfung mit TNC

Mit 802.1x kann die Authentität von Netzwerkteilnehmern sichergestellt werden. Dies ist aber aus Sicherheitsgesichtspunkten in vielen Fällen nicht ausreichend, da zwar damit geklärt ist, wer sich im Netz aufhält, aber nicht in welchem Zustand sich dessen Rechner (PC oder Laptop) befindet. Heutige Netze sind nach außen durch Firewalls und demilitarisierte Zonen immer besser gesichert, aber von innen stark verwundbar. Dies ist vor allem dann der Fall, wenn das Netz quasi öffentlich zugänglich ist, wie das Netz auf einem Hochschulcampus, und sich die teilnehmenden Endgeräte unter fremder Kontrolle (nämlich der des Besitzers) befinden. In Zeiten von Würmern, Viren und Trojanern können solche Rechner zu einem Sicherheitsrisiko im Netz werden, auch ohne den menschlichen Teilnehmern (kriminelle) Absicht zu unterstellen. Deshalb basieren viele heutige Aktivitäten auf dem Ansatz, die Integrität (also den Zustand und damit die Vertrauenswürdigkeit) des Endgeräts zu überprüfen, bevor es Zugang zu einem Netz erhält oder während es Mitglied des Netzes ist.

Ein vielversprechender Ansatz wird von der Trusted Computing Group[1] verfolgt, der deshalb im Folgenden vorgestellt wird und im Zentrum dieser Arbeit steht.

3.1 Über TCG und TNC

Die Trusted Computing Group (TCG) wurde als Organisation gebildet, um offene Standards im Bereich der Hardware-basierten sicheren IT und Sicherheitstechnologien im Allgemeinen zu definieren, zu entwickeln und zu veröffentlichen.[2] Die Bedeutung dieser Organisation macht die Mitgliederliste deutlich: Firmen wie Microsoft, AMD, Intel, Hewlett-Packard oder Sun beteiligen sich aktiv in der TCG.

In der TCG gibt es mehrere Untergruppen (sogenannte Subgroups), welche sich mit speziellen Themen beschäftigen. Die Trusted Network Connect Sub Group (TNC-SG) definiert eine offene Architektur mit einer Vielzahl und einer immer weiter wachsenden Zahl von Standards im Bereich der Endgeräte-Integrität. Netzwerkadministratoren können mit Hilfe von TNC sicherstellen, dass Endgeräte gewisse Integritätsbedingungen erfüllen, wenn sie sich in das Netz einklinken wollen oder auch während sie verbunden sind.

Die TNC-Standards sind offen definiert, so dass sie über Herstellergrenzen hinweg

[1] https://www.trustedcomputinggroup.org

[2] Informationen aus: https://www.trustedcomputinggroup.org/about/, siehe auch: [Tru04]

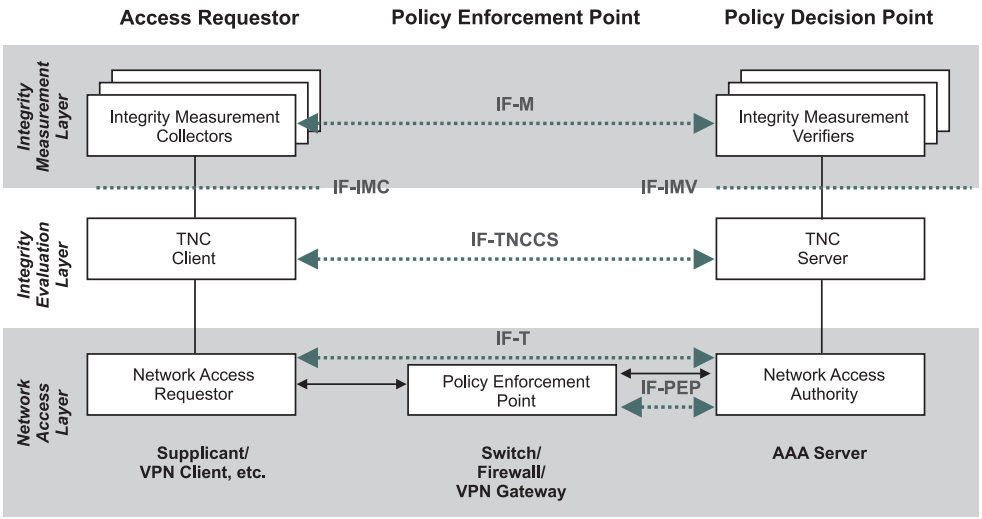


Abbildung 3.1: TNC-Architektur (vgl. [TCG05a, S. 13, Abb. 2])

unabhängig von bestimmten Endgeräten, Netzwerktechnologien und Richtlinien angewandt werden können. Dabei wird bewusst auf bestehende Standards, wie z.B. 802.1x aufgebaut und dadurch das Einsatzgebiet dieser Standards erweitert.

3.2 Architektur von TNC

Abbildung 3.1 zeigt die Architektur, wie sie TNC definiert. Sie besteht aus drei horizontal verlaufenden Schichten und einer vertikalen Aufteilung auf drei Entitäten: Access Requestor, Policy Enforcement Point und Policy Decision Point. Diese drei Entitäten bestehen aus Komponenten in den verschiedenen Schichten. Komponenten haben in der gleichen Schicht auf anderen Entitäten entsprechende Komponenten, mit denen sie über Schnittstellen kommunizieren. Außerdem definiert TNC teilweise Schnittstellen zwischen den Komponenten einer Entität.

Die einzelnen Schichten, Entitäten, Komponenten und Schnittstellen werden überblicksweise im Folgenden beschrieben. Für detaillierte Informationen wird auf die entsprechenden Spezifikationen verwiesen.

3.2.1 Entitäten von TNC

Folgende Entitäten sind in der TNC-Architektur definiert [TCG06a]:

Access Requestor (AR) Der Rechner, welcher Zugang zu einem Netz erhalten möchte, hat die Rolle des Access Requestors inne. Seine Aufgabe besteht darin, Informationen über die Integrität des Rechners zu liefern und diese an

den Policy Decision Point zu übertragen. Im Vergleich zu 802.1x entspricht die Entität des Access Requestors einem erweiterten Supplicant.

Policy Decision Point (PDP) Normalerweise bildet ein Server die Entität des Policy Decision Point. Seine Aufgabe besteht darin, die Integritätsinformationen der einzelnen AR entgegenzunehmen, weitere benötigte Informationen zu erfragen, um schließlich anhand einer Sicherheitsrichtlinie zu entscheiden, ob dem einzelnen AR Zutritt gewährt wird, der Zugriff verweigert oder ob ihm die Möglichkeit gegeben werden soll, seinen Integritätszustand zu verbessern. Diese Entscheidung übermittelt der Policy Decision Point dem Policy Enforcement Point. Im Vergleich zu 802.1x entspricht die Entität des Policy Decision Point einem erweiterten Authentication Server.

Policy Enforcement Point Im Ethernet bildet ein Switch die Entität des Policy Enforcement Point. Er hat wie der Authenticator in 802.1x die Aufgabe, dem AR so lange den Zutritt zu verwehren, bis seine Integrität überprüft und sichergestellt wurde. Er setzt damit die Entscheidung des PDP um. Erweitert zu 802.1x sieht die TNC-Architektur auch vor, einem AR eingeschränkten Zutritt zu gewähren. Der Policy Enforcement Point weist dann den AR einem geschützten Netz zu.

3.2.2 Phasen von TNC

In der TNC-Architektur sind drei Phasen definiert, die ein AR während der Integritätsprüfung durchlaufen kann:

Assessment In dieser Phase wird die Integrität des Endgeräts geprüft. Als Ergebnis dieser Phase wird der AR entweder vom PDP zugelassen, wenn er die Integritätsbedingungen erfüllt, oder er wird abgelehnt, weil die Integritätsbedingungen nicht erfüllt sind und auch nicht erreicht werden können oder isoliert, weil die Integrität zwar nicht ausreicht, aber nachgebessert werden kann. Diese Entscheidungen setzt der PEP um, indem er den Zugriff entsprechend gewährt oder verweigert.

Isolation Wenn der AR isoliert wird, landet er in einem abgeschotteten Netz mit Zugriff auf Updates, die den Integritätszustand verbessern.

Remediation Mit den zur Verfügung gestellten Updates kann der AR sich nun „sanieren“, um danach wieder in die Assessment-Phase zu wechseln.

3.2.3 Die Schichten von TNC

Im Folgenden werden die einzelnen Komponenten der Entitäten im Kontext ihrer Zugehörigkeit zu einer Schicht genauer beschrieben.

Integrity Measurement Layer

Die TNC-Architektur ist offen angelegt. Damit soll erreicht werden, dass eine Vielzahl von Produkten und Herstellern für die Integritätsbestimmung eines Rechners eingebunden werden können. Da eine einzelne Anwendung dieses nicht allein bewältigen kann, ist die TNC-Architektur so definiert, dass verschiedene sicherheitsrelevante Produkte in der obersten Schicht (Integrity Measurement Layer, kurz: IML) eingebunden werden.

Ein Hersteller oder Anbieter liefert zu seinem Produkt (z.B. Virenscanner) im einfachsten Fall zwei Komponenten[3]: Integrity Measurement Collector (IMC) und Integrity Measurement Verifier (IMV). Die IMC-Komponente befindet sich auf dem AR und sammelt dort Informationen über den Zustand ihres Produktes (z.B. Versionsstand, Aktualisierungsdatum) und übermittelt diese Informationen an die zugehörige IMV-Komponente. Diese wiederum ist auf dem PDP lokalisiert und kann von dort aus überprüfen, ob der Integritätsstand des Produkts ausreichend ist. Falls weitere Informationen benötigt werden, kann die IMV-Komponente diese von der IMC-Komponente anfordern. Wenn alle benötigten Informationen gesammelt sind, entscheidet die IMV-Komponente, ob aus ihrer Sicht dem AR der Zugang gewährt werden soll, und gibt diese Empfehlung an den TNC Server weiter. Die Kommunikation zwischen IMVs und IMCs wird in der proprietären Schnittstelle IF-M definiert (siehe Kapitel 3.3.6).

Integrity Evaluation Layer

Die Komponenten in der Schicht mit der Bezeichnung Integrity Evaluation Layer (IEL) sind dafür verantwortlich, die Komponenten aus der IML anzubinden und den Nachrichtenaustausch zwischen IMCs und IMVs zu ermöglichen. Die in der IML gemessene Integrität wird in dieser Schicht bewertet und die Konsequenzen daraus gezogen.

Der TNC Client ist eine Software-Komponente, welche auf dem AR läuft, dort die Integritätsinformationen von den IMCs sammelt und diese an den TNC Server übermittelt. Die Kommunikation zwischen IMCs und TNC Client wird in der Schnittstelle IF-IMC definiert (siehe Kapitel 3.3.1), die Kommunikation zwischen TNC Client und TNC Server in der Schnittstelle IF-TNCCS (siehe Kapitel 3.3.3).

Der TNC Server bildet das Gegenstück zum TNC Client auf Serverseite und kommuniziert dort mit den IMV-Komponenten über die Schnittstelle IF-IMV (siehe Kapitel 3.3.2). Er leitet die Nachrichten zwischen IMCs und IMVs auf Serverseite weiter und sammelt die Empfehlungen der IMVs ein. Die Aufgabe des TNCS besteht dann darin, diese Vorschläge anhand einer Richtlinie zusammenzuführen und zu einer abschließenden Entscheidung zu kommen. Diese Entscheidung wird an die

[3] Es sind auch Szenarien denkbar, in denen mehrere Komponenten für ein Produkt eingesetzt werden.

Network Access Authority in der untersten Schicht weitergegeben.

Network Access Layer

Die unterste Schicht heißt Network Access Layer (NAL). Die Komponenten dort sind für die physikalische Übermittlung der Nachrichten auf Netzwerkebene und für die physikalische Umsetzung der Richtlinien verantwortlich.

Die NAL kann z.B. mit Hilfe von 802.1x umgesetzt werden. In diesem Fall agiert die Network Access Requestor-Komponente als Supplicant innerhalb des AR. Sie ist dafür verantwortlich, die EAPOL-Verbindung zum PEP zu initiieren und die Nachrichten vom TNC Client entgegenzunehmen und als EAP-Pakete verpackt an die Network Access Authority (NAA) weiterzuleiten. Das „EAP-Binding“ ist in der Schnittstelle IF-T (siehe Kapitel 3.3.4) definiert.

Die NAA-Komponente agiert –wenn 802.1x eingesetzt– üblicherweise als RADIUS-Server und kommuniziert mit dem PEP über das RADIUS-Protokoll. Zusätzlich bindet die NAA den TNC Server an und leitet die Nachrichten zu ihm und von ihm weiter. Die Entscheidung, ob dem AR Zutritt gewährt wird, leitet sie an den PEP weiter. Dies ist in der Schnittstelle IF-PEP definiert (siehe Kapitel 3.3.5).

Der PEP (ein 802.1x-kompatibler Switch) setzt dann die Richtlinien auf physikalischer Ebene um.

3.3 Definierte Schnittstellen in TNC

Im folgenden wird ein kurzer Überblick über die definierten Schnittstellen in der TNC-Architektur gegeben, indem deren Ideen dargelegt werden.

3.3.1 IF-IMC: Kommunikation zwischen TNC Client und IMCs

Zusammen mit IF-IMV (siehe nächster Abschnitt) war IF-IMC eine der ersten Spezifikationen im Rahmen von TNC (Version 1.0: [TCG05b]). Mittlerweile existiert bereits Version 1.1 [TCG06b]. IF-IMC beschreibt die Kommunikationsmöglichkeiten zwischen IMC und TNC Client und erlaubt es dadurch dem TNC Client:

- IMCs zu finden,
- IMCs zu initialisieren,
- IMCs über Veränderungen des Verbindungszustandes (neue Verbindung, Handshake[4] zwischen IMVs und IMCs wird durchgeführt, Zutritt erlaubt/abgelehnt...) zu informieren,

[4] Als Handshake bezeichnet man den Austausch der Nachrichten zwischen IMCs und IMVs, bis die Integrität des AR festgestellt werden kann.

- Nachrichten von IMVs an IMCs zu übermitteln,
- IMCs explizit aufzufordern, Nachrichten zu übermitteln,
- IMCs nach erfolgreicher Arbeit oder im Fehlerfall zu beenden.

Die IMC-Komponente kann wiederum:

- dem TNC Client übermitteln, welche Nachrichten sie empfangen möchte,
- Nachrichten für IMVs an den TNC Client übermitteln,
- eine erneute Integritätsprüfung anfordern (wenn sich z.B. ihr Zustand geändert hat).

Darüber hinaus definiert IF-IMC sogenannte „Platform Bindings“ für C in Windows und Linux, so dass die Spezifikation auch praktisch eingesetzt werden kann.

3.3.2 IF-IMV: Kommunikation zwischen TNC Server und IMVs

IF-IMV wurde gleichzeitig mit IF-IMC spezifiziert (Version 1.0: [TCG05c]) und existiert ebenfalls seit Mai 2006 in der Version 1.1 ([TCG06c]). Diese beiden Spezifikationen ähneln sich sehr. IF-IMV spezifiziert aber die Kommunikationsmöglichkeiten zwischen TNC Server und IMVs. Es ermöglicht dem TNC Server:

- IMVs zu finden,
- IMVs zu laden,
- IMVs über Veränderungen des Verbindungszustandes zu informieren (z.B. wenn sich ein neuer TNC Client an den TNC Server gewandt hat, wenn der Handshake beginnt...)
- Nachrichten von IMCs an IMVs zu übermitteln,
- IMVs aufzufordern, eine Empfehlung abzugeben,
- IMVs aufzufordern, Nachrichten zu übermitteln,
- IMVs bei Beendigung des TNC Servers oder im Fehlerfall zu beenden.

Die IMV-Komponente kann wiederum mit Hilfe von IF-IMV:

- dem TNC Server übermitteln, welche Nachrichten sie empfangen möchte,
- Nachrichten für IMCs an den TNC Server übermitteln,

- eine erneute Durchführung des Handshake für einen TNC Client verlangen,
- eine Empfehlung darüber abgeben, wie mit dem AR aus ihrer Sicht zu verfahren ist (ablehnen, zulassen oder eingeschränkt zulassen).

Darüberhinaus definiert ebenfalls IF-IMV sogenannte „Plattform Bindings“ für C in Windows und Linux.

3.3.3 IF-TNCCS: Kommunikation zwischen TNC Client und TNC Server

Erst in der zweiten Spezifikationsphase der TNC-SG kam die Spezifikation IF-TNCCS dazu [TCG06f]. Sie beschreibt die Kommunikation zwischen TNC Client und TNC Server und ermöglicht somit die Interoperabilität zwischen TNC Clients und TNC Servern unterschiedlicher Hersteller.

IF-IMC und IF-IMV definieren zwar, wie die Nachrichten lokal auf Client und Server zu behandeln sind; in welchem Format diese über das Netz geschickt werden, wird darin aber nicht geklärt. Deshalb definiert IF-TNCCS im Wesentlichen ein Datenaustauschformat auf Basis von XML, welches in einer XMLSchema-Definition festgelegt wird.[5] Abbildung 3.2 zeigt den prinzipiellen Aufbau solch einer Nachricht.

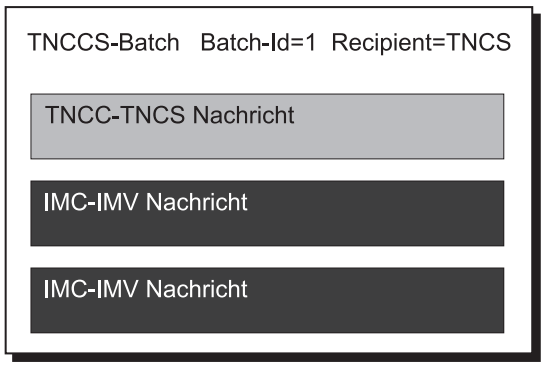


Abbildung 3.2: Aufbau einer TNCCS-Nachricht (nach [TCG06f, S. 19, Abb. 3])

Ein Datenpaket wird in IF-TNCCS als Batch bezeichnet und wird entweder zum TNCC oder zum TNCS geschickt (ensprechend des Recipient-Attributs). In solch einem Batch können zwei verschiedene Arten von Nachrichten vorkommen:

1. TNCC-TNCCS Nachrichten enthalten Informationen, welche zwischen TNC Client und TNC Server ausgetauscht werden, um z.B. Sitzungsinformationen

[5] Eine Einführung in XML gibt [ERH01].

zu übermitteln. Den Inhalt dieser Nachrichten bekommen weder IMCs noch IMVs zu sehen.

2. IMC-IMV Nachrichten werden zwischen IMCs und IMVs ausgetauscht und zwar über die IF-IMC bzw. IF-IMV Schnittstelle. Der Inhalt dieser Nachrichten ist für den TNC Client und TNC Server nicht von Interesse.

Abbildung 3.3 zeigt dieselbe Nachricht in ihrer XML-Form. Das Routing der Nachrichten von IMC zu IMV bzw. IMV zu IMC läuft über Nachrichtentypen ab, für die sich IMCs und IMVs registrieren können. Der Inhalt der IMC-IMV Nachrichten wird immer Base64-kodiert, die TNCC-TNCS Nachrichten dagegen können auch in XML vorliegen.

```
<?xml version="1.0"?>
<TNCCS-Batch BatchId="1" Recipient="TNCS"
    xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">

 <TNCC-TNCS-Message>
   <Type>0304BBDD</Type>
   <XML><MyTNCMessage id=99>Some content</MyTNCMessage></XML>
 </TNCC-TNCS-Message>

 <IMC-IMV-Message>
   <Type>AABB0001</Type>
   <Base64>342jkdslfjklsd34</Base64>
 </IMC-IMV-Message>

 <IMC-IMV-Message>
   <Type>AABB0002</Type>
   <Base64>123jkddf3slf5lfdd34</Base64>
 </IMC-IMV-Message>

</TNCCS-Batch>
```

Abbildung 3.3: Eine exemplarische TNCCS-Nachricht in XML (vgl. [TCG06f, S. 19, Abb. 3])

IF-TNCCS definiert eine besondere TNCC-TNCCS Nachricht, welche nach Abschluss des Handshake zwischen IMCs und IMVs vom TNC Server an den TNC Client geschickt wird, um ihn über die zusammenfassende Empfehlung des TNC Servers und die daraus resultierende Aktion des PEP zu unterrichten. In Abbildung 3.4 wird die Empfehlung „Access Requestor zulassen“ übermittelt.

3.3.4 IF-T: TNC über getunneltes EAP

Die bisher vorgestellten Schnittstellen befassen sich mit der Kommunikation auf der IE-Schicht und/oder IM-Schicht. Bisher außen vor geblieben ist aber, wie diese Nachrichten physikalisch über das Netz geschickt werden sollen. Prinzipiell lässt die TNC Architektur diese Frage offen, weil TNC in unterschiedlichsten Umgebungen eingesetzt werden kann. Einen Vorschlag für den Einsatz von TNC in einer 802.1x-basierten Umgebung macht die TCG aber in der Spezifikation IF-T [TCG06e], in

```
<?xml version="1.0"?>
<TNCCS-Batch BatchId="4" Recipient="TNCC"
   xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">

  <TNCC-TNCS-Message>
    <Type>00000001</Type>
    <XML>
      <TNCCS-Recommendation type="allow" />
    </XML>
  </TNCC-TNCS-Message>

</TNCCS-Batch>
```

Abbildung 3.4: Exemplarische TNCCS-Nachricht in XML mit enthaltender Empfehlung (vgl. [TCG06f, S. 20])

der beschrieben wird, wie TNC in getunnelten EAP-Methoden eingesetzt werden kann. Dazu definiert die TNC-SG einen neuen EAP-Typen, welcher wie jeder andere EAP-Typ im Kontext von 802.1x eingesetzt werden kann und innerhalb anderer sicherer EAP-Typen (wie EAP-TTLS) genutzt wird und damit „getunnelt“ wird.

Abbildung 3.5 zeigt die definierte EAP-TNC-Struktur, wie sie in ein Ethernet-EAPOL-Paket verpackt wird.[6]

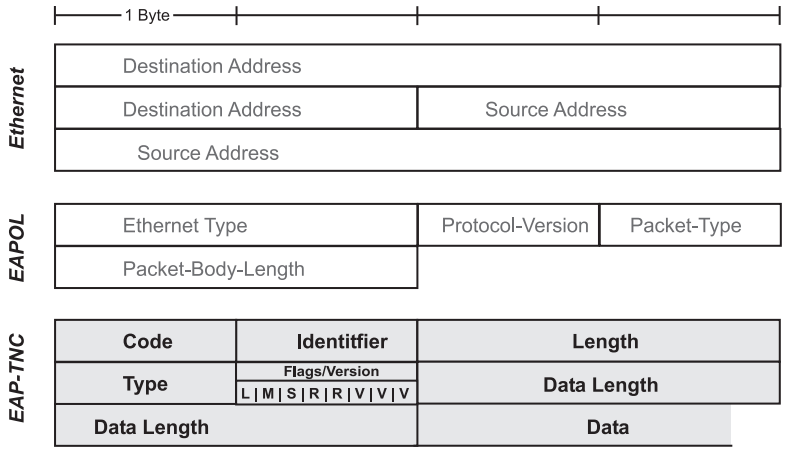


Abbildung 3.5: Aufbau des EAP-TNC-Pakets als Nutzlast eines EAPOL-Paketes in 802.1x

Ohne auf jedes einzelne Feld dieses Paket-Typen detailliert einzugehen, soll hier nur ein grober Überblick gegeben werden:

- Die EAP-TNC-Kommunikation wird mit einem *EAP-TNC-Start*-Paket gestartet. Dabei wird eine leere Nachricht übertragen, in der nur das Start-Flag (S) gesetzt ist.

[6] Dabei ist außen vorgelassen, dass IF-T verlangt, die EAP-TNC-Nachrichten durch einen weiteren EAP-Typen zu tunneln (wie EAP-TTLS), um die übertragenen Daten zu verschlüsseln.

- Die eigentliche Nutzlast (nämlich die TNCCS-Nachrichten) wird im Data-Feld übertragen.
- Da TNCCS-Nachrichten manchmal für umgebende Protokolle zu groß werden, sieht EAP-TNC die Fragmentierung (also die Aufteilung) der Daten vor. Über die Flags wird gesteuert, ob noch weitere Pakete folgen (moreFragments=1, (M)) und ob die Gesamtlänge der Daten übertragen wird (lengthIncluded=1, (L)).[7] Letzteres geschieht nur, wenn das erste Fragment einer Nachricht übertragen wird. Die Gesamtlänge der Daten befindet sich dann im Data Length-Feld.
- Fragmentierte Nachrichten werden vom Kommunikationspartner mit leeren Acknoledgement-Nachrichten quittiert, bis die vollständige TNCCS-Nachricht übertragen wurde.[8]

3.3.5 IF-PEP: TNC angebunden an RADIUS

Mit IF-T ist definiert, wie AR und PDP miteinander kommunizieren. Was bisher aber noch nicht geklärt wurde, ist, wie der PDP dem PEP (also normalerweise dem Switch) die durchzuführende Aktion zur Gewährung, Isolation oder Ablehnung des AR übermittelt. Dies spezifiziert IF-PEP [TCG06d] für das Einsatzumfeld, in dem sich PDP und PEP über das RADIUS-Protokoll unterhalten.

Im Wesentlichen definiert IF-PEP drei Wege, wie der PDP dem PEP über RADIUS die gewünschten Aktionen übermittelt (vgl. [TCG06d, S. 15f]):

Binäre Zugriffskontrolle Bei Einsatz der binären Zugriffskontrolle entfällt die Isolationsstufe. Über *RADIUS Access-Accept* oder *Access-Reject* wird dem PEP die Entscheidung des PDP übermittelt. Der PEP öffnet daraufhin den Port oder lässt ihn geschlossen.

VLAN-basierte Zugriffskontrolle Die VLAN-basierte Zugriffskontrolle ermöglicht eine fein-granulare Entscheidungsübermittlung. Mit Hilfe von RADIUS-Tunnel-Attributen (definiert in RFC2868) wird dem PEP übermittelt, in welches VLAN er den AR aufnehmen soll. Im einfachsten Fall existieren zwei VLANs: eins für das normale Netz und eins für das Isolationsnetz.

Filter-basierte Zugriffskontrolle Bei der Filter-basierten Methode[9] übermittelt der PDP dem PEP mit Hilfe des Filter-ID RADIUS-Attributs (definiert in RFC

[7] Die übrigen Flags im EAP-TNC Format bedeuten folgendes: R steht für reserviert und ist nicht belegt, V beinhaltet die aktuelle Version von EAP-TNC.

[8] IF-T spezifiziert nicht, was es unter einer leeren Acknoledgement-Nachricht versteht, eine Interpretation findet sich in Kapitel 7.3.2.

[9] Auf Cisco-Switches heißen diese Filter „Access Control Lists“ (ACL). Was ACLs sind, erklärt [Cis04, S. 28-1ff]

2865), welchen Filter er auf den Port des AR anwenden soll, um dessen Datenverkehr zuzulassen, zu beschränken oder zu verhindern.

Dabei ist die VLAN-basierte Methode die von IF-PEP favorisierte Methode; mindestens eine der drei Methoden muss die TNC-Architektur aber unterstützen, damit sie zu IF-PEP konform ist.

3.3.6 IF-M: Proprietäre Kommunikation zwischen IMCs und IMVs

Der Inhalt und spezielle Ablauf der Kommunikation zwischen IMCs und IMVs ist nicht von der TNC-SG definiert, da sie abhängig von den funktionalen Anforderungen der einzelnen Sicherheitsanwendung sind. Wie solche Kommunikation (IF-M genannt) aussehen kann, zeigt Daniel Wuttke in seiner Masterarbeit (siehe [Wut06]).

3.4 Ablauf der Integritätsprüfung

Nachdem alle Entitäten, deren Komponenten und die Schnittstellen im Einzelnen vorgestellt wurden, wird nun deren Zusammenspiel innerhalb der TNC-Architektur gezeigt. Die folgende Ablaufbeschreibung orientiert sich an den Nummern aus Abbildung 3.6.[10].

1. Bevor die Integritätsprüfung startet, lädt der TNC Client alle registrierten IMCs, so dass er diese ansprechen kann. Ebenfalls lädt der TNC Server alle verfügbaren IMVs.

2. Sobald der AR physikalisch in das Netz eingeklinkt wird, startet der NAR den Versuch, die Integritätsprüfung über eine EAPOL Start-Nachricht zu starten. Die Initialisierung der EAP-Verbindung findet nun statt.

3. Nachdem die EAP-Verbindung zwischen AR und PEP initialisiert wurde, leitet letzterer die Anfrage an die NAA weiter. Die folgende Kommunikation zwischen NAR und NAA findet über EAP-TNC (definiert in IF-T) statt.

4. Die NAA informiert den TNC Server über den neuen Verbindungswunsch.

5. Der TNC Server informiert die IMVs über die neue Verbindung, welche sie bedienen müssen. Gleichzeitig informiert der TNC Client die IMCs über die neue Verbindung, worauf diese dem TNC Client die ersten IMC-IMV-Nachrichten übermitteln.

[10] Die gesamte Beschreibung ist eine leicht abgewandelte Darstellung der Ablaufbeschreibung in [TCG05a, S. 17ff].

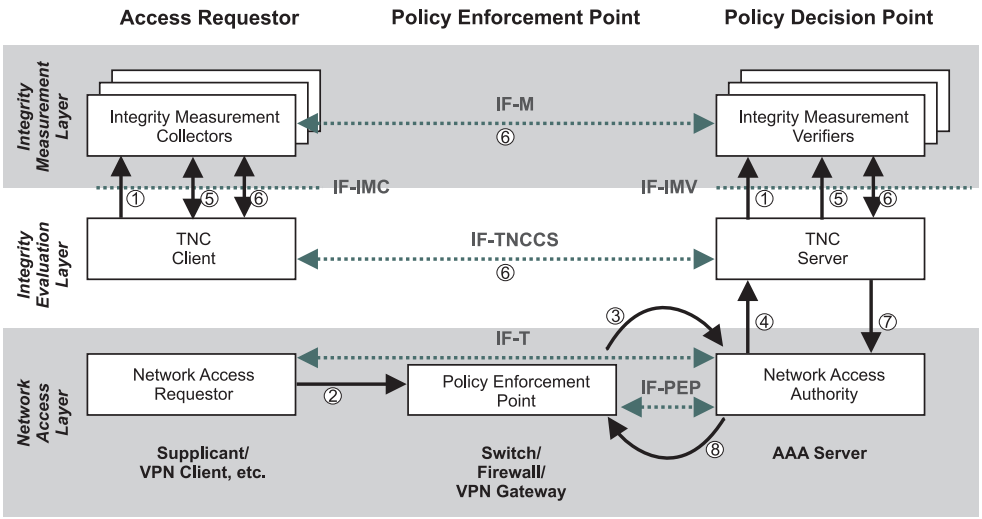


Abbildung 3.6: Ablauf der Integritätsprüfung in der TNC-Architektur (vgl. [TCG05a, S. 17, Abb. 3])

6. Über den physikalischen „Umweg“ NAR, PEP und NAA kommunizieren TNC Client und TNC Server nun über das IF-TNCCS-Protokoll miteinander und tauschen dabei die Nachrichten zwischen IMCs und IMVs aus. Wenn die IMVs genügend Informationen gesammelt haben (oder auf Nachfrage des TNC Servers), geben sie eine Empfehlung zu der Vertrauenswürdigkeit des AR ab.

7. Der TNC Server bildet sich anhand der übermittelten Empfehlungen und seiner Sicherheitsrichtlinie ein Gesamturteil und gibt dieses an die NAA weiter.

8. Über das in IF-PEP definierte Verfahren leitet die NAA diese Aufforderung an den PEP weiter, welcher die Verbindung zum AR entweder physikalisch sperrt, zulässt oder eingeschränkt zulässt.

4 Vorüberlegungen für Entwurf und Implementierung

Bevor auf den Entwurf und die Implementierung der Client- und Serversoftware eingegangen wird, sollen die Rahmenbedingungen und Anforderungen dargelegt werden, welche die Grundlage für Entwurf und Implementierung bilden.

4.1 Rahmenbedingungen

Da das Projekt „TNC@FHH“ keine kommerziellen Ziele verfolgt, sondern wissenschaftliche Erkenntnisse liefern soll, wird keine kommerzielle Software entwickelt. Die entwickelte Software soll stattdessen einer Open Source-Lizenz unterstellt werden. Daraus resultiert, dass innerhalb der Software nur auf existierende Komponenten (wie z.B. RADIUS-Server) zurückgegriffen wird, sofern diese innerhalb von Open Source-Lösungen genutzt werden dürfen. Ebenfalls soll die Entwicklung der Software möglichst mit frei zugänglichen Entwicklungsumgebungen möglich sein, damit die Weiterentwicklung leicht möglich ist.

Eine weitere Rahmenbedingung besteht darin, dass die TNC-Architektur auf Basis von 802.1x entwickelt werden soll und in Ethernet-basierten LANs Verwendung findet. Damit können sowohl andere Technologien innerhalb der NAL-Schicht wie VPN außer Betracht bleiben, als auch andere in 802.1x definierte Einsatzgebiete wie WLANs mit EAPOW vernachlässigt werden.

4.2 Policy Enforcement Point

Um die oben genannten Rahmenbedingungen zu erfüllen, wird ein Switch benötigt, welcher 802.1x unterstützt. Für dieses Projekt stehen ein Cisco 3550 und ein HP Procurve 5348xl (siehe Abbildung 4.1) zur Verfügung, welche beide diese Voraussetzung erfüllen.

Da der Procurve-Switch extra für dieses Projekt von HP zur Verfügung gestellt wurde und Cisco sich nicht innerhalb der TCG engagiert, wird die Testumgebung für dieses Projekt mit dem Procurve-Switch aufgebaut und die zu entwickelnde Software auf die Belange dieses Switches ausgerichtet. Trotzdem soll die entwickelte Software auch mit dem Cisco-Switch als Eigentum der FH lauffähig sein, was zum

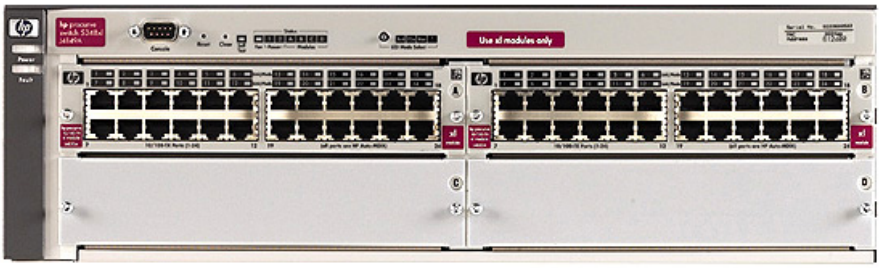


Abbildung 4.1: HP Procurve 5348xl (Quelle: http://www.hp.com)

Abschluss des Projekts sichergestellt werden muss.

Da beide Switches nur die Anbindung von RADIUS-Servern als Authentication Server unterstützen, erfolgt die Kommunikation in diesem Projekt zwischen Switch und PDP über RADIUS.

4.3 Nicht-funktionale Anforderungen

Zusammenfassend lassen sich folgende nicht-funktionale Anforderungen an die zu erstellende Software, bestehend aus Client und Server, definieren:

- Die entwickelte Software wird als Open Source zur Verfügung gestellt; genutzte Komponenten müssen dafür freigegeben sein.
- Die Software ist auf Client und Server in C oder C++ entwickelt, um die Platform Bindings aus IF-IMC und IF-IMV nutzen zu können.
- Auf Client-Seite läuft die Software unter Windows XP, da hier das größte Nutzer-Potenzial besteht und im zweiten Teilprojekt von Daniel Wuttke (siehe [Wut06]) IMCs für diese Umgebung erstellt werden.
- Auf Server-Seite läuft die Software unter Linux, weil die Server-Landschaft im Fachbereich Informatik Linux-basiert ist. Zusätzlich können dadurch die zwei unterschiedlichen von TNC in IF-IMC und IF-IMV definierten Bindings für Windows (im AR) und Linux (im PDP) getestet werden.
- Die erstellte Software kann möglichst einfach in das jeweils andere Betriebssystem portiert werden. Plattform-spezifische Bestandteile werden deshalb gekapselt und gekennzeichnet.
- Die Software läuft in der Testumgebung mit den oben genannten verfügbaren Switches.
- Die verfügbaren (und hier vorgestellten) TNC-Spezifikationen werden soweit möglich genau umgesetzt.

- Die Client-Komponente umfasst die TNC-Komponenten TNC Client und Network Access Requestor und bindet IMCs an.
- Die Server-Komponente umfasst die TNC-Komponenten TNC Server und Network Access Authority und bindet IMVs an.
- Es werden keine IMCs und IMVs entwickelt. Dies erfolgt im Projekt von Daniel Wuttke ([Wut06]). Allerdings können beliebige IMCs und IMVs gemäß IF-IMC bzw. IF-IMV angebunden werden. Abbildung 4.2 zeigt die in dieser Arbeit relevanten TNC-Komponenten und -Schnittstellen.
- Auf Netzwerkebene wird 802.1x eingesetzt. Kommuniziert wird gemäß IF-T.
- Die Software kann gut wiederverwendet und erweitert werden und wird dementsprechend strukturiert und dokumentiert.
- Mehrere anfragende Clients werden parallel geprüft.

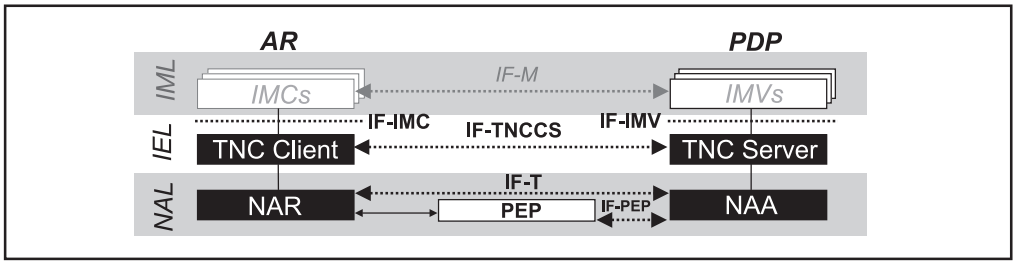


Abbildung 4.2: Die für diese Arbeit relevanten TNC-Komponenten und -Schnittstellen sind schwarz unterlegt und dick gedruckt (nicht grau und kursiv).

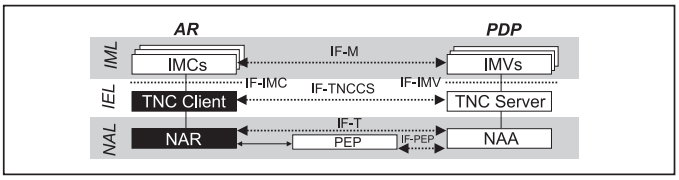
4.4 Funktionale Anforderungen

Die zu konzipierende Software muss sich folgenden funktionalen Anforderungen stellen:

- Die Integritätsprüfung für Windows-Clients wird ermöglicht.
- Die Integritätsprüfung liefert als Ergebnis je nach Integritätsstatus eine der drei Verhaltensempfehlungen: „Zutritt erlauben“, „Zutritt verwehren“ oder „Zutritt isoliert erlauben“.
- Die Verhaltensempfehlungen resultieren aus einer einfachen konfigurierbaren Sicherheitsrichtlinie.

- Der Switch setzt diese Verhaltensempfehlungen physikalisch auf dem entsprechenden Port um.
- Die Zuordnung des Ports je nach Verhaltensempfehlung in verschiedene VLANs wird ermöglicht (gemäß IF-PEP).
- Der Benutzer behält die Kontrolle über die Integritätsprüfung.
- Die automatische Sanierung des AR wird nicht unterstützt. Nach erfolgreicher manueller Sanierung im Isolierungs-VLAN durch den Benutzer wird die Integritätsprüfung neu gestartet.

5 Entwurf des Access Requestor unter Windows



In diesem Kapitel geht es darum, wie der Entwurf des AR unter den bereits vorgestellten Rahmenbedingungen und Anforderungen aussieht. Dazu wird in Kapitel 5.1 detailliert die Gesamtarchitektur des AR vorgestellt, bevor danach die einzelnen Schichten für sich betrachtet werden: Kapitel 5.2 hat die entscheidenen Konzepte innerhalb des NAR und Kapitel 5.3 innerhalb des TNC Client zum Inhalt. Abschließend wird in Kapitel 5.4 die Benutzerschnittstelle des AR vorgestellt.

5.1 Gesamtarchitektur

5.1.1 Architekturüberblick

Der AR ist als Gesamtapplikation konzipiert, welche vor der Integritätsprüfung manuell durch den Benutzer gestartet wird, und nicht ständig im Hintergrund läuft. Dadurch läuft die Integritätsprüfung für den Benutzer sichtbar und nachvollziehbar ab.

Die Gesamtarchitektur ist in Abbildung 5.1 dargestellt. Der AR besteht aus der Applikation `TNC-AR` und den IMC-Komponenten, die wie im Windows Platform Binding von IF-IMC festgelegt als DLLs angebunden werden.

`TNC-AR`, die im Rahmen dieser Arbeit entworfen und entwickelt wird, besteht aus zwei Komponenten, welche die gleichnamigen Komponenten aus der TNC-Architektur widerspiegeln (vgl. Abbildung 3.1 auf Seite 20). Diese Zwei-Teilung ermöglicht die Entkoppelung des TNC Clients von der darunter liegenden Netzwerk-Technologie (hier 802.1x) und die eventuelle spätere Portierung auf eine andere Technologie (z.B. VPN).

Ein detailliertes Klassendiagramm mit allen Klassen findet sich im Anhang in Kapitel A.2.

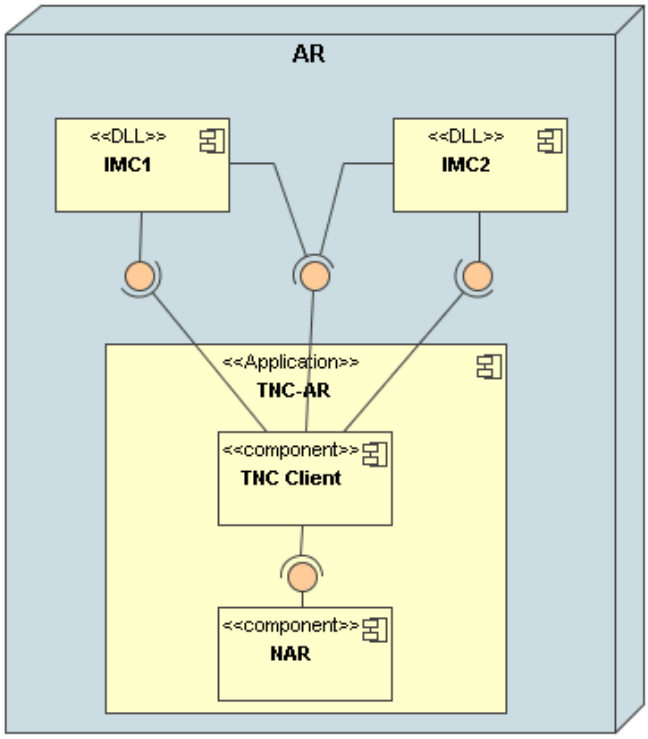


Abbildung 5.1: Gesamtarchitektur des AR als Deployment-Diagramm

5.1.2 Fassaden zwischen den Komponenten des AR

Die Komponenten des AR sind durch klar definierte und eng umgrenzte Fassaden voneinander getrennt, um sie möglichst lose zu koppeln (siehe Abbildung 5.2).

Zwischen IMCs und TNC Client werden die in IF-IMC definierten Schnittstellen genutzt, welche auf Seiten des TNC Client in zwei C-Dateien realisiert werden: `tnc_imc_fascade` beinhaltet den plattform-unabhängigen Teil der Schnittstelle, während `tnc_imc_winBinding` die Funktionen des Windows-Bindings bereitstellt.

Die Benutzeroberfläche `TNCC GUI`[1] nutzt `TnccFascade` als Schnittstelle, um Informationen zu erlangen (IMC-Namen, Netzwerkgeräte-Namen, Zustand der Prüfung) und Aktionen anzustoßen (Integritätstest starten, abbrechen).[2]

`TNC Client` selbst nutzt `NARFascade` als Schnittstelle zur darunterliegenden Komponente `NAR`.

[1] Weil die Benutzeroberfläche nicht zum fachlichen Teil des TNC Client gehört, ist sie auf Abbildung 5.2 außerhalb des TNC Clients dargestellt, gehört aber selbstverständlich dazu.
[2] Mehr zur Benutzerschnittstelle findet sich in Kapitel 5.4.

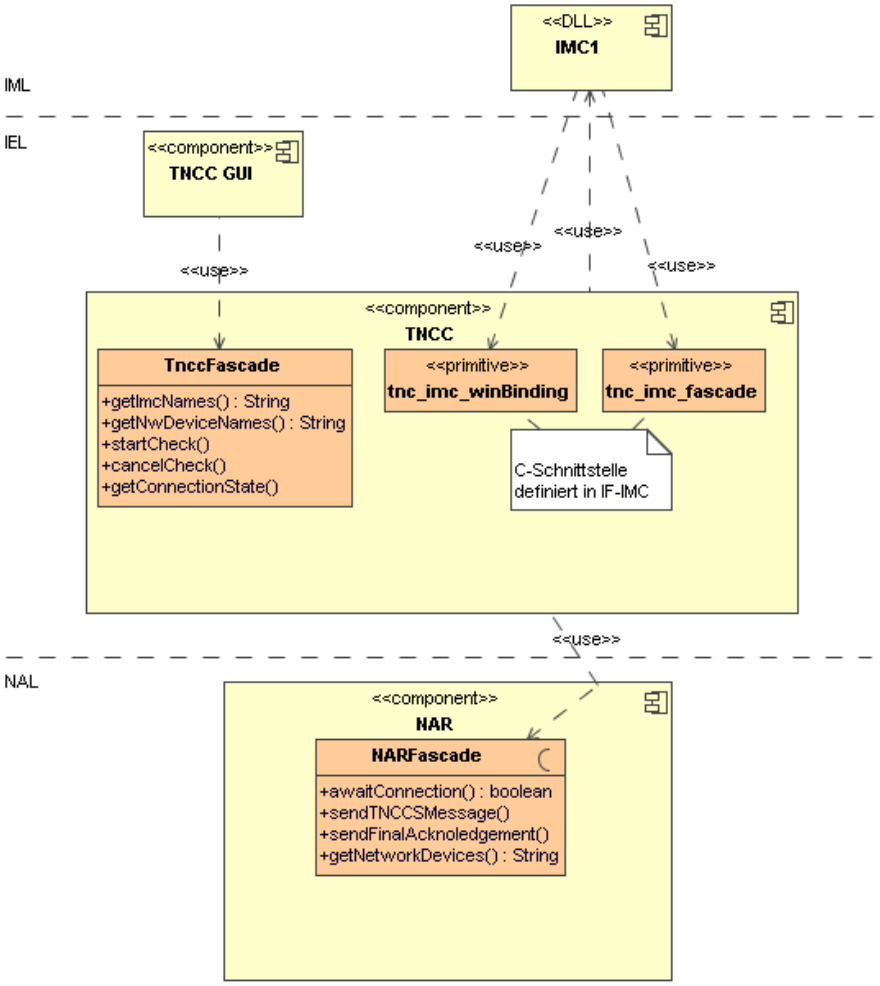
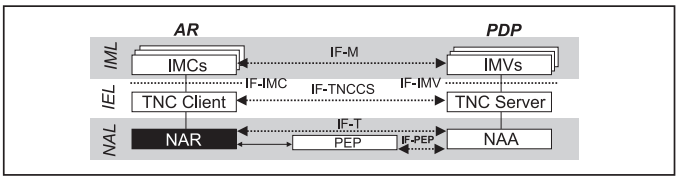


Abbildung 5.2: Fassaden zwischen den Komponenten des AR

5.2 Network Access Requestor



5.2.1 Problemstellung

In den nicht-funktionalen Anforderungen (siehe Kapitel 4.3) wurde festgelegt, dass die NA-Schicht auf 802.1x aufbaut. Somit muss im NAR folgendes realisiert sein:

- Der NAR agiert als 802.1x-Supplicant, das bedeutet:

  - Er baut EAPOL-Pakete und verschickt diese über das Netz.
  - Er empfängt EAPOL-Pakete aus dem Netz und analysiert diese.
  - Er realisiert den in 802.1x vorgesehenen Authentisierungsablauf.

- Der NAR implementiert EAP-TNC, das bedeutet:

  - Er bettet EAP-TNC-Nachrichten in die EAPOL-Pakete ein.
  - Er extrahiert EAP-TNC-Nachrichten aus empfangenen EAPOL-Paketen und interpretiert diese.
  - Er unterstützt die Fragmentierung von TNCCS-Nachrichten gemäß IF-T.

Nun stellt sich die Frage, inwiefern diese Funktionalität oder Teile davon aus anderen Lösungen wiederverwendet werden können, oder ob eine Neuimplementierung erforderlich ist.

5.2.2 Lösungsansätze

Es bieten sich einige Lösungsalternativen an, die auf ihre Tragfähigkeit untersucht werden müssen und in ihrer Auflistung von oben nach unten eine zunehmende Eigenimplementierung erfordern:

- Nutzung der 802.1x-Funktionalität von Xsupplicant
- Nutzung von Windows-Mitteln für einfache EAP-Nutzung
- Nutzung von Fremdlösungen für den Netzwerkzugriff
- Vollständige Eigenimplementierung

Nutzung der 802.1x-Funktionalität von Xsupplicant

Xsupplicant[3] ist ein OpenSource-Projekt, welches die Authentifizierung über 802.1x unter Linux ermöglicht. Dazu implementiert Xsupplicant einen einfachen 802.1x-Supplicant, der EAP-MD5, EAP-TLS sowie EAP-TTLS unterstützt.[4] Mittlerweile unterstützt Xsupplicant auch EAP-TNC (in einer proprietären Version zusammen mit Radiator[5]); diese Funktionalität konnte aber nicht mehr in die Analyse einfließen.

Die Nutzung von Teilen von Xsupplicant brächte einige Vorteile:

[3] http://open1x.sourceforge.net/
[4] In [Aur05, S. 305ff] findet man eine detaillierte Beschreibung zu Xsupplicant.
[5] Radiator ist ein Open Source RADIUS-Server: http://www.open.com.au/radiator/

- Die grundlegende 802.1x-Funktionalität inkl. Authentisierungsablauf ist implementiert und kann genutzt werden.
- Die Erzeugung von EAPOL-Paketen und deren Analyse kann genutzt werden.
- Xsupplicant ist vielfach eingesetzt und lässt dadurch Stabilität erwarten.

Die Nachteile überwiegen aber:

- EAP-TNC ist (in der untersuchten Version) nicht implementiert und muss neu hinzugefügt werden.
- Xsupplicant ist in C implementiert und damit nicht nach objekt-orientierten Maßstäben entwickelt.
- Die Einarbeitung in bestehende Software ist mit hohem Aufwand verbunden, so auch bei Xsupplicant.
- Xsupplicant ist für Linux entwickelt. Damit muss der Netzwerkzugriff neu für Windows entwickelt werden und größere Eingriffe in die bestehende Software sind nötig.

Da es nicht so scheint, als ob Xsupplicant in großen Teilen als „Blackbox“ eingesetzt werden kann, sondern fundamentale Anpassungen nötig sind, wird der Ansatz verworfen, Xsupplicant als Basis für den hier vorgestellten AR zu nutzen.

Nutzung von Windows-Mitteln für einfache EAP-Nutzung

Windows XP ermöglicht standardmäßig eine 802.1x-Authentifizierung, so dass ein naheliegender Ansatz darin besteht, zu versuchen, diese Authentifizierung zu nutzen und zu erweitern. Im MSDN[6] finden sich zum Thema „EAP“ zwei Einträge:

Der erste[7] beinhaltet eine EAP-API, mit der die EAP-Funktionalität von Windows XP angebunden und (theoretisch) erweitert werden kann. Dem Autor erschienen die Möglichkeiten sehr eingeschränkt und die gegebenen Informationen nicht ausreichend, um eine eigene EAP-DLL zu entwickeln. Daher wurde dieser Ansatz verworfen.

Eine zweite API „Extensible Authentication Protocol Host“[8] ist erst unter Windows Vista (dem Nachfolger von Windows XP) verfügbar und damit unter den gesetzten Anforderungen nicht nutzbar.

Damit ist der Ansatz, Windows-Mittel zu nutzen, ebenso nicht durchführbar.

[6] Microsoft Developer Network: http://www.msdn.microsoft.com
[7] http://msdn.microsoft.com/library/en-us/eap/eap/eap_start_page.asp
[8] http://msdn.microsoft.com/library/en-us/eaphost/eaphost/portal.asp

Nutzung von Fremdlösungen für den Netzwerkzugriff

Wenn keine fertigen 802.1x-Lösungen eingesetzt werden, wäre es doch zumindest von Vorteil, wenn der Zugriff auf das Netzwerk komfortabel möglich ist. Dazu wird eine API gesucht, mit der einfach Ethernet-Pakete über C bzw. C++ verschickt und empfangen werden können. Was im Umfeld von IP-Netzen in der Internet-Schicht über den Einsatz von Sockets eine Selbstverständlichkeit ist, existiert eine Schicht darunter, in der sogenannten „Host-an-Netz“-Schicht [Tan00, S. 52ff] oder Netzzugangsschicht[9] nicht. Die Aussage gilt zumindest für Ethernet unter Windows.

Vollständige Eigenimplementierung

Da alle bisher vorgestellten Ansätze nicht tragbar sind, wird eine vollständige Eigenimplementierung angestrebt, die folgende Bestandteile beinhaltet:

- Versand und Empfang von Ethernetpaketen unter Windows
- Aufbau und Analyse von EAPOL-Paketen
- Umsetzung des EAP-TNC-Protokolls mit eigenständiger Fragmentierung von TNCCS-Nachrichten.
- Implementierung des Ablaufs der 802.1x-Authentisierung als Basis für TNC

5.2.3 Netzzugriff über NDIS

Wie im vorangegangenen Abschnitt erläutert, wurden keine APIs gefunden, die den Zugriff auf Ethernet in der Netzwerk-Schicht aus Windows heraus zulassen. Daher muss dieser Zugriff sehr system-nah implementiert werden.

[mia03] zeigt für die system-nahe Implementierung eine Lösung auf: Die Idee besteht darin, einen sogenannten NDIS Protocol Driver zu erstellen. NDIS[10] ist ein innerhalb von Windows genutzter Standard zur Einbindung von Netzwerkkarten, der es u.a. erlaubt, mehrere Netzwerkkarten nebeneinander zu betreiben. Außerdem können damit mehrere Protokolle auf einer Netzwerkkarte genutzt werden, indem für jedes Protokoll ein NDIS Protocol Driver installiert wird. Im Microsoft Windows Driver Development Kit[11] ist unter den Beispielen solch ein Protokoll-Treiber (NDISProt) enthalten, welcher es ermöglicht, aus Programmen heraus Ethernet-Pakete zu versenden und zu empfangen. Dies funktioniert dann analog zum Dateizugriff.

[9] http://de.wikipedia.org/wiki/TCP/IP-Referenzmodell
[10] NDIS steht für Network Device Interface Specification, Quelle: http://de.wikipedia.org/wiki/NDIS
[11] http://www.microsoft.com/whdc/devtools/ddk/default.mspx

Welche Anpassungen an dem Treiber vorgenommen werden müssen, wie er installiert und genutzt wird, ist Inhalt von Kapitel 8.2. An dieser Stelle reicht die Erkenntnis, dass ein Weg gefunden wurde, Ethernet-Pakete zu empfangen und zu versenden und damit die letzte Hürde für die vollständige Eigenimplementierung genommen ist.

5.2.4 Architektur

Die statische Architektur des NAR zeigt Abbildung 5.3. Die Klassen des NAR haben

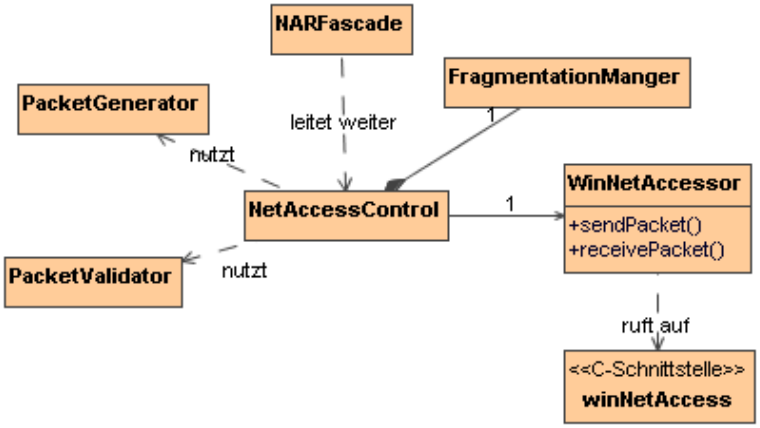


Abbildung 5.3: Statische Architektur des NAR als Klassendiagramm

eng umrissende Aufgaben:

NARFascade bildet wie in Kapitel 5.1.2 bereits beschrieben den Zugriffspunkt auf den NAR. Die Schnittstelle ist implementierungsunabhängig konzipiert, so dass für den TNC Client transparent bleibt, welche Netzwerk-Technologie dahinter im NAR genutzt wird.

NetAccessControl bildet die Ablaufsteuerung des NAR und steuert die übrigen Klassen und den korrekten 802.1x-Ablauf.

WinNetAccessor bietet Zugriff auf das Netz aus Windows heraus und ermöglicht den Versand und Empfang von Ethernet-Paketen mit Hilfe der C-Schnittstelle `winNetAccess`.[12]

PacketGenerator erstellt EAPOL-Pakete mit gekapselten EAP-TNC-Paketen (siehe Kapitel 5.2.5).

[12] `WinNetAccessor` implementiert die plattform-unabhängige Schnittstelle `NetAccessor`. Siehe dazu auch Kapitel 7.1.

PacketValidator analysiert empfangene Ethernet-Pakete auf ihre Inhalte (siehe Kapitel 5.2.5).

FragmentationManager kümmert sich um die Fragmentierung und Defragmentierung von Nachrichten (siehe Kapitel 7.3.2).

5.2.5 Implementierung von EAPOL und EAP-TNC

Die zwei Klassen `PacketGenerator` und `PacketValidator` (siehe Abbildung 5.4) sind für die Analyse und Erstellung der eingehenden und ausgehenden Ethernet-Pakete verantwortlich.

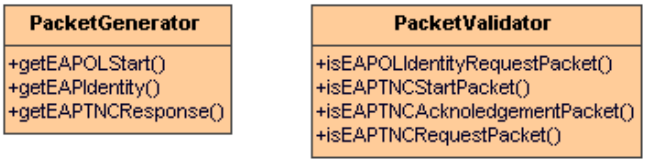


Abbildung 5.4: Klassen mit Schwerpunkt auf Paket-Generierung und -Validierung

`PacketGenerator` erstellt vollständige versendbare (binäre) Ethernet-Pakete und füllt sie mit den übergebenen Informationen. Drei Typen von Ethernet-Paketen kann der NAR verschicken, welche von den folgenden Methoden erzeugt werden:

getEAPOLStart liefert ein *EAPOL-Start*-Paket (vgl. Kapitel 2.3.2).

getEAPIdentity liefert ein *EAP Response*-Paket mit der übergebenen Identität (vgl. Kapitel 2.3.1).

getEAPTNCResponse liefert ein *EAP Response*-Paket vom Typ EAP-TNC. Durch die Übergabeparameter kann der Inhalt jedes einzelnen Feldes und Flags genau spezifiziert, sowie Daten angehängt werden.

`WinNetAccessor` liest beliebige Pakete vom LAN, so dass eine Instanz diejenigen Pakete identifizieren muss, welche für den Integritätstest von Bedeutung sind. Diese Aufgabe übernimmt `PacketValidator`. Diese Klasse erkennt nicht nur die gewünschten Pakete, sondern extrahiert die benötigten Felder aus den Byte-Blöcken des Pakets innerhalb der jeweiligen Methoden:

isEAPOLIdentityRequestPacket prüft, ob das übergebene Paket ein *EAP-Identity Request* ist.

isEAPTNCStartPacket prüft, ob das übergebene Paket ein *EAP-TNC-Start*-Paket (vgl. Kapitel 3.3.4) ist.

isEAPTNCAcknoledgementPacket prüft, ob das übergebene Paket ein *EAP-TNC-Acknoledgement*-Paket ist (vgl. Kapitel 7.3.3).

isEAPTNCRequestPacket prüft, ob das übergebene Paket ein *EAP-TNC-Request* ist, und gibt neben der enthaltenen TNCCS-Nachricht auch den Status der im Paket enthaltenen Flags zurück (vgl. Kapitel 3.3.4).

Mit Hilfe dieser Klassen kann `NetAccessControl` einfach Ethernet-Pakete erstellen und eingehende Ethernet-Pakete auf ihre Relevanz im aktuellen Kontext prüfen, indem jedes empfangene Paket gegen die entsprechende Methode von `PacketValidator` geprüft wird.

5.2.6 Ablauf beim Verbindungsaufbau

Grob kann man die Abläufe im NAR in zwei Phasen aufteilen: In der ersten Phase sorgt der NAR dafür, dass die Verbindung zum PDP hergestellt wird, so dass in der zweiten Phase die TNCCS-Nachrichten zwischen AR und PDP ausgetauscht werden können.

Der Ablauf der ersten Phase innerhalb des AR wird in Abbildung 5.5 dargestellt. Wichtig ist hier, dass für den Aufrufer `TncsFlowControl` aus dem TNC Client (1)[13] transparent bleibt, wie diese Verbindung hergestellt wird. Am Ende wird ihm signalisiert, ob die Verbindungsaufnahme erfolgreich war (25). Was dazwischen passiert, ist für ihn nicht von Interesse:

`NarFascade` leitet die Anfrage an `NetAccessControl` weiter (2), welche von nun an die Steuerung des Ablaufs übernimmt. Zunächst wird `WinNetAccessor` mitgeteilt, welches Netzwerk-Device genutzt wird[14] (3). Als erste Anfrage soll ein *EAPOL Start* über das Netz an den Switch gehen, welches `PacketGenerator` liefert (4+5). Den Versand des Pakets übernimmt `WinNetAccessor` (6-8), der aber zunächst die Netzwerkschnittstelle initialisiert (7).

Nun ist der Switch an der Reihe und muss ein *EAP Identity Request* schicken, auf das gewartet wird (9). Dafür horcht `WinNetAccessor` an der Netzwerkschnittstelle auf eintreffende Pakete und gibt diese zurück (10+11). Wenn eins davon das erwartete ist, erkennt `PacketValidator` dies (12+13).[15]

Der NAR ist nun an der Reihe, mit einem *EAP Identity Response*-Packet zu antworten. Dieses von `PacketGenerator` gelieferte Paket (12+13) verschickt `WinNetAccessor` (16-18).

[13] In dieser und den folgenden Beschreibungen von Sequenzdiagrammen entspricht die Nummer in Klammern als Orientierungshilfe den Nummern an den Nachrichten im Diagramm.

[14] Dies ist dann von Bedeutung, wenn mehrere Netzwerkkarten im Rechner installiert sind. Dann kann der Benutzer in der Benutzeroberfläche eine davon auswählen (siehe Kapitel 5.4).

[15] Im Sequenzdiagramm ist dieser Vorgang (und ähnliche folgende) so dargestellt, als ob das erste eintreffende Paket direkt das gesuchte ist. Dies ist hier und in folgenden Diagrammen aus Gründen der Übersichtlichkeit so dargestellt.

Abbildung 5.5: Ablauf innerhalb des NAR während des Verbindungsaufbaus

Nun wird im Hintergrund der Switch das erste Mal den RADIUS Server kontaktieren, welcher ein *EAP-TNC-Start* schicken wird. Deshalb wird nun auf dieses Paket gewartet (19), indem wieder auf ankommende Pakete gehorcht wird (20+21) und diese überprüft werden (22+23). Wenn das richtige Paket eingetroffen ist, ist die Initialisierungsphase abgeschlossen. Dies wird dem TNC Client übermittelt (24+25), der nun die erste TNCCS-Nachricht übermitteln kann.

5.2.7 Ablauf während des Handshake

In der zweiten Phase ist der NAR für den Versand und den Empfang von TNCCS-Nachrichten zuständig. Der Ablauf (siehe Abbildung 5.6) wird jedesmal vom TNC Client aufs Neue angestoßen, wenn eine neue TNCCS-Nachricht verschickt werden soll (1). Auch hier bleibt dem TNC Client verborgen, wie die TNCCS-Nachricht

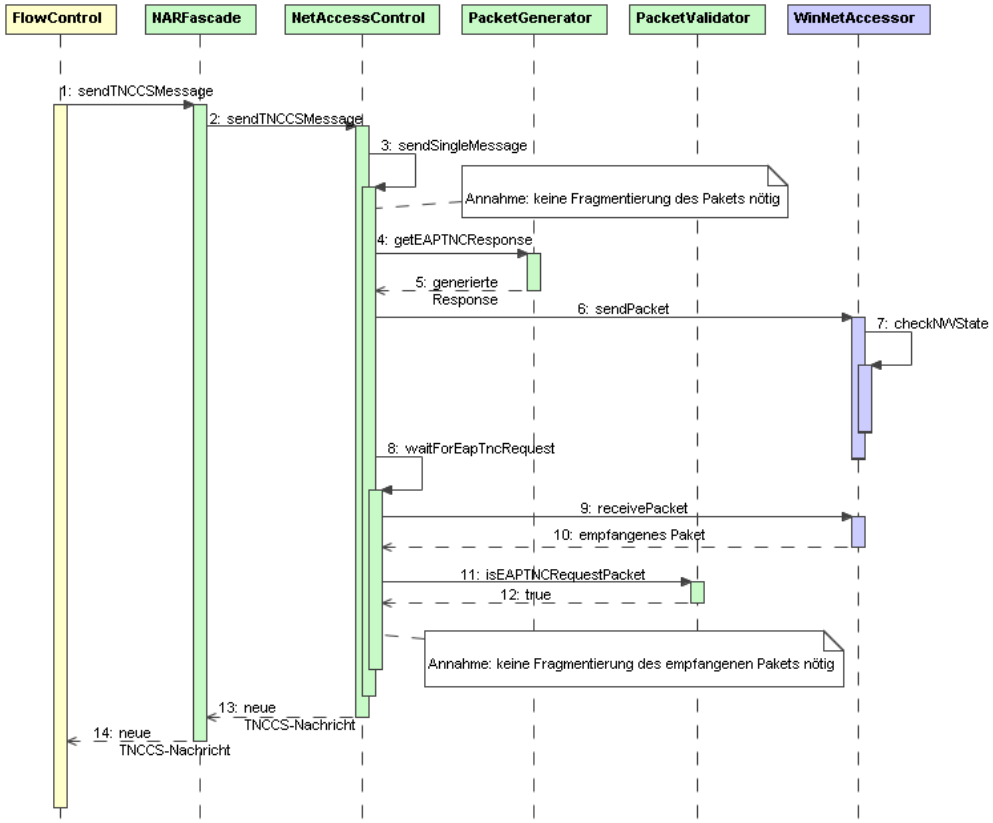


Abbildung 5.6: Ablauf im NAR beim Versand und Empfang von TNCCS-Nachrichten

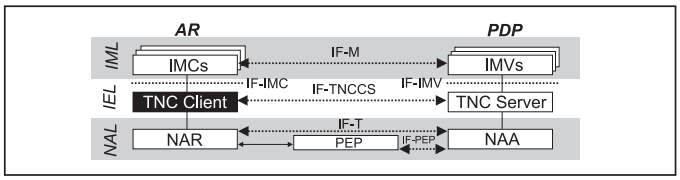
ihr Ziel findet und wie die Antwort darauf zustande kommt, die das Ergebnis dieses Aufrufs darstellt (13-14).

Für den dargestellten Ablauf wird vereinfacht angenommen, dass keine Fragmen-

tierung stattfindet[16], sondern Nachrichten einzeln als Ganzes verschickt werden können (3).

Der AR ist nun an der Reihe, eine *EAP Response* vom Typ EAP-TNC mit der enthalteten TNCCS-Nachricht zu verschicken. Dieses Paket liefert `PacketGenerator` (4+5); `WinNetAccessor` verschickt es (6+7). Als nächstes wird auf die Antwort des PDP gewartet (8). Die eingetroffenen Pakete (9+10) werden dahingehend überprüft, ob es sich bei einem von ihnen um den neuen *EAP Request* mit der TNCCS-Antwort handelt (11+12). Wenn dem so ist, wird direkt die TNCCS-Nachricht extrahiert und wieder an den TNC Client zurückgeliefert (13+14).

5.3 TNC Client



5.3.1 Architekturüberblick

Der TNC Client ist aus einer Reihe von Klassen aufgebaut, welche in Abbildung 5.7 zu sehen sind. Diese Klassen haben eng umgrenzte Aufgaben:

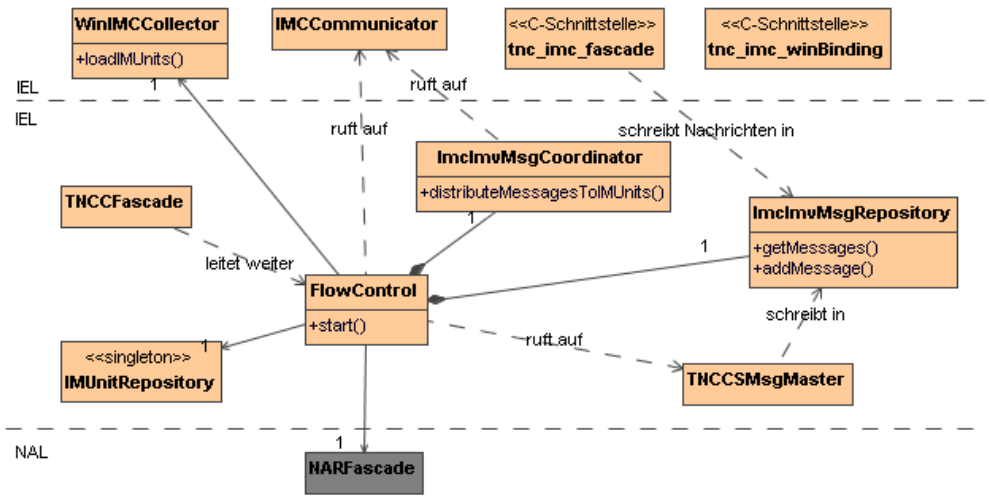


Abbildung 5.7: Klassen innerhalb des TNC Clients

FlowControl steuert den Ablauf innerhalb des TNC Client.

[16]Mit der Fragmentierung von Nachrichten befasst sich Kapitel 7.3.2 detailliert.

TNCCFascade bildet den Zugriffspunkt auf den TNC Client aus der Benutzeroberfläche (vgl. Kapitel 5.1.2).

TNCCSMsgMaster analysiert und generiert TNCCS-Nachrichten (für mehr Details siehe Kapitel 7.3.1).

ImcImvMsgCoordinator verteilt die IMC-IMV-Nachrichten an die passenden IMC-Komponenten.

IMCCommunicator kommuniziert direkt mit den IMCs über die IF-IMC-Schnittstelle.

WinIMCCollector lädt die IMCs gemäß dem Windows-Binding aus IF-IMC.

ImcImvMsgRepository speichert die Nachrichten, welche von IMCs an IMVs geschickt werden sollen und solche, die von den IMVs für IMCs eingetroffen sind.

Die Klassen rund um die Kommunikation mit den IMCs werden in Kapitel 7.2 gesondert vorgestellt, weil sie in großen Teilen genauso im TNC Server zum Einsatz kommen.

5.3.2 Abläufe innerhalb des TNC Client

Es soll an dieser Stelle auf den Ablauf innerhalb des TNC Client und die Interaktion der Klassen untereinander eingegangen werden. Dies ist im Sequenzdiagramm in Abbildung 5.8 dargestellt. Wenn der TNC Client gestartet wird (1), lädt `IMCCollector` zunächst alle installierten und registrierten IMCs (2). `IMCCommunicator` initialisiert die geladenen IMCs, indem er sie über die neue Verbindung informiert (3). Dann wird dem NAR der Auftrag gegeben, die Verbindung zum PDP herzustellen (4). Wenn dies geschehen ist, holt `IMCCommunicator` die ersten IMC-IMV-Nachrichten von den IMCs (5) und lässt `TNCCSMsgMaster` daraus eine TNCCS-Nachricht erstellen (6+7).

Der folgende Ablauf wird jetzt ständig wiederholt (8): Die erstellte TNCCS-Nachricht wird in den NAR zum Versand gegeben (9) und eine neue TNCCS-Nachricht erhalten (10). `TNCCSMsgMaster` liest diese neue Nachricht und extrahiet die IMC-IMV-Nachrichten (11). Wenn keine Empfehlung in der TNCCS-Nachricht enthalten ist (12), verteilt `ImcImvMsgCoordinator` diese Nachrichten an die passenden IMCs, die wiederum neue Nachrichten in den TNC Client geben (13). `TNCCSMsgMaster` erstellt aus diesen neuen Nachrichten wiederum eine TNCCS-Nachricht (14+15) und es beginnt wieder bei (9).

Wenn allerdings die TNCCS-Nachricht vom TNC Server eine Empfehlung enthält (16), informiert `IMCCommunicator` die IMCs über das Ergebnis (17) und das Ende des Handshakes (18). Dem NAR übergibt `FlowControl` den Auftrag, mit einer finalen Bestätigung den Erhalt dieser Empfehlung zu quittieren (19). Dann beendet

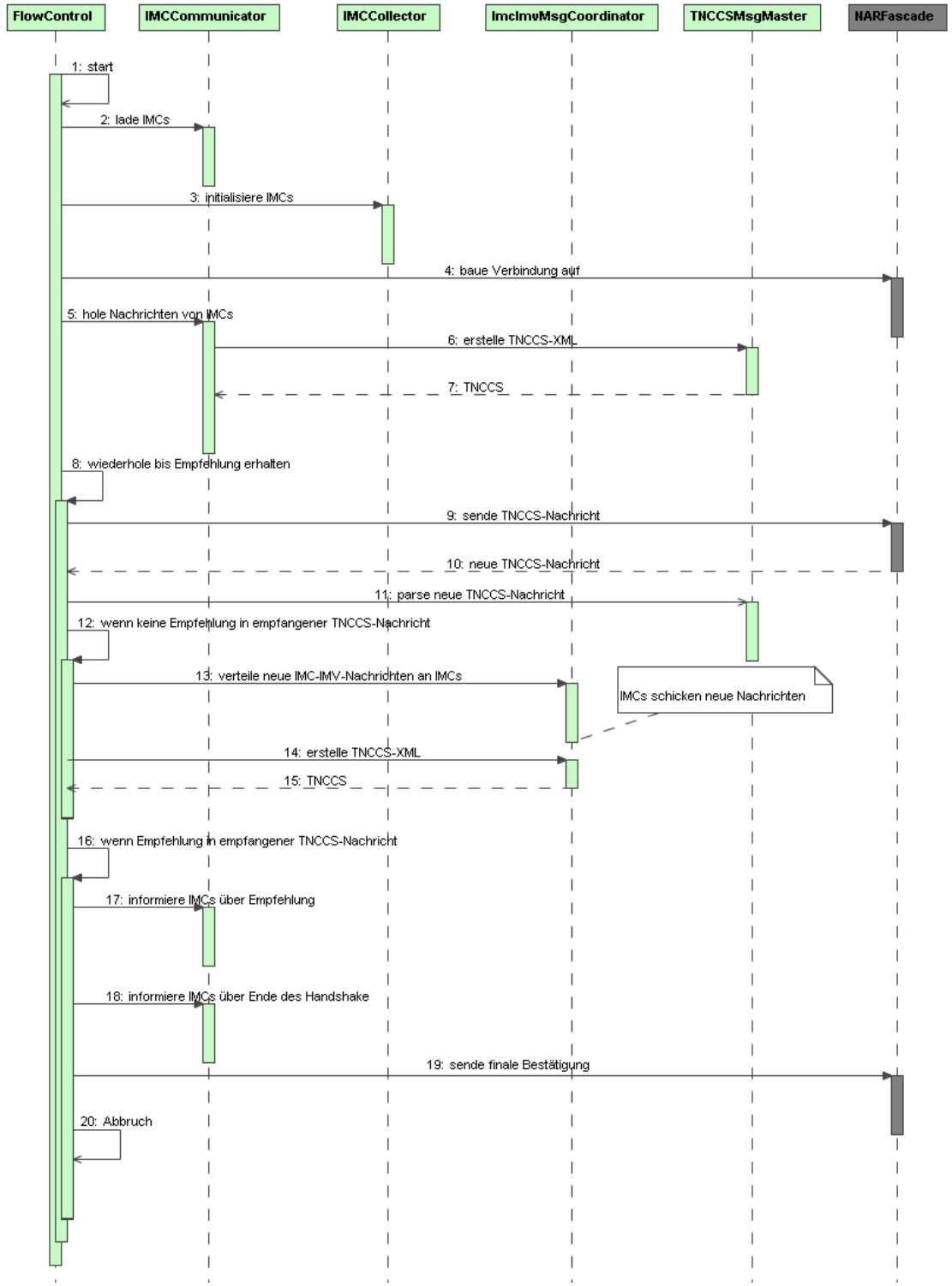


Abbildung 5.8: Überblick über die Abläufe im TNC Client

sich der TNC Client (20), der PDP wird nun die entsprechende Maßnahme (Zulassung, Isolation oder Ablehnung) initiieren. Detailliertere Sequenzdiagramme zu den Abläufen im TNC Client finden sich im Anhang in Kapitel A.1.

5.4 Benutzerschnittstelle

Benutzer werden dem TNC Client skeptisch gegenüber stehen, da die Gefahr besteht, dass dieses „Spionage-Tool“ persönliche Daten an „irgendwelche Server“ überträgt und damit ihre eigenen Sicherheitsbedürfnisse missachtet. Um die Benutzerakzeptanz zu erhöhen, ist es deshalb erforderlich, das Geschehen so transparent wie möglich zu gestalten. Dafür werden in TNC@FHH zwei Prinzipien umgesetzt:

- Die Kontrolle bleibt beim Benutzer.
- Der Benutzer erfährt genau, was passiert.

5.4.1 Grafische Benutzeroberfläche

Ein wichtiger Aspekt für den Benutzer ist, dass er die Kontrolle über das Geschehen behält. Dafür wird eine einfache Benutzeroberfläche entworfen, die als Prototyp in Abbildung 5.9 zu sehen ist. Weitere Screenshots finden sich im Anhang A.3. Mit dieser Oberfläche wird dem Benutzer folgendes ermöglicht:

- Der Benutzer kann auswählen, über welches Netzwerk-Interface die Integritätsprüfung abläuft. Dazu werden ihm die verfügbaren Netzwerk-Interfaces angezeigt.
- Der Benutzer kann sehen, welche IMCs installiert und für die Integritätsprüfung verfügbar sind und somit vertrauensunwürdige IMCs vom Integritätstest ausschließen.
- Detailliertes Logging (siehe Kapitel 5.4.2) kann aktiviert werden und der Pfad zu der Log-Datei spezifiziert werden. So bleiben die Informationen persistent erhalten.
- Über ein Log-Fenster wird der Benutzer jederzeit überblicksweise über den momentanen Stand des Integritätstests informiert.
- Der Benutzer kann jederzeit den Integritätstest abbrechen.

5.4.2 Logging des Integritätstests

Die Kontrolle über die Integritätsprüfung kann dem Benutzer nur eingeschränkt gegeben werden. Für einen reibungslosen Ablauf ist es nicht praktisch umsetzbar,

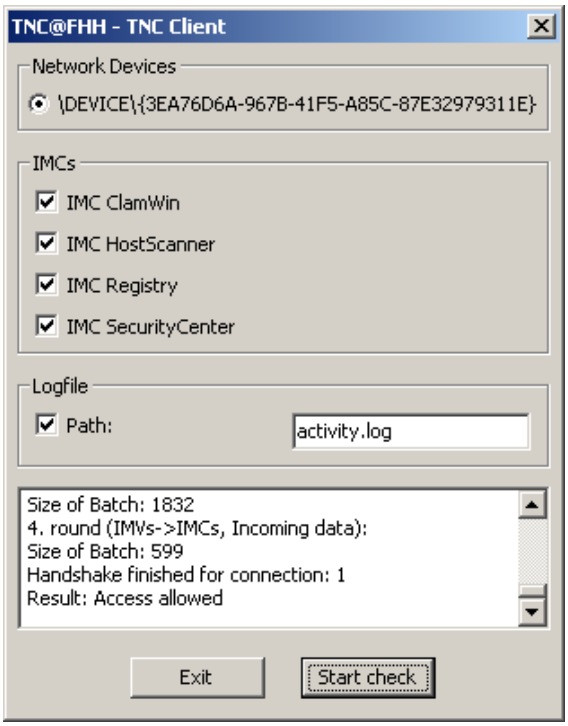


Abbildung 5.9: TNC Client-Oberfläche: Die Integritätsprüfung endet mit Zutrittserlaubnis.

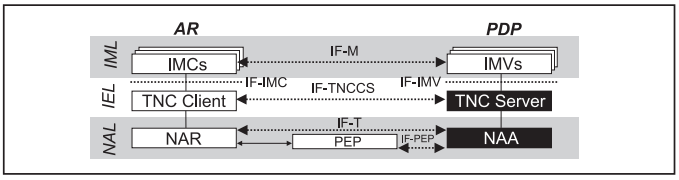
bei jeder zu übertragenen Nachricht zu fragen, ob diese übertragen werden darf. Daher ist es zumindest wichtig, dass der Benutzer umfassend informiert wird. Dies geschieht über eine Logging-Datei, in der der gesamte Integritäts-Handshake protokolliert wird und im speziellen,

- welche IMCs geladen werden,
- welche TNCCS-Nachrichten an den TNC Server geschickt werden,
- welche TNCCS-Nachrichten vom TNC Server eingegangen sind,
- welche Empfehlung vom TNC Server eingegangen ist.

Die IMC-IMV-Nachrichten werden innerhalb der TNCCS-Nachrichten base64-kodiert übertragen sind und damit für den Benutzer nicht einfach zu lesen. Zudem geht aus den TNCCS-Nachrichten auch nicht direkt hervor, an welche IMCs die IMC-IMV-Nachrichten gerichtet sind bzw. von welchem IMC sie stammen. Daher werden zusätzlich folgende Nachrichten im Klartext protokolliert:

- Nachrichten von IMCs mit Angabe ihrer Quelle
- Nachrichten zu IMCs mit Angabe ihres Ziels

6 Entwurf des Policy Decision Point unter Linux



In diesem Kapitel wird gezeigt, wie der Entwurf des Policy Decision Point unter den vorgestellten Rahmenbedingungen und Anforderungen aussieht. Dazu werden in Kapitel 6.1 zunächst Realisierungsmöglichkeiten für die Gesamtarchitektur bewertet sowie eine geeignete gewählt und diese vorgestellt. Danach werden die einzelnen Schichten des PDP einzeln betrachtet: Kapitel 6.2 hat die entscheidenden Konzepte der NAA zum Inhalt. Kapitel 6.3 zeigt die zentralen Konzepte des TNC Servers. Abschließend wird in Kapitel 6.4 darauf eingegangen, wie im PDP mit mehreren parallelen Anfragen umgegangen wird.

6.1 Gesamtarchitektur

Bevor Details des PDP konzipiert werden können, stellt sich zunächst die Frage, wie die generelle Gesamtarchitektur aussehen soll.

6.1.1 Problemstellung

Die alles bestimmende Anforderung an den PDP besteht darin, dass er in der NAL als RADIUS-Server agieren und die Verbindung zwischen dieser Funktion und dem TNC Server herstellen muss. Daraus ergeben sich die folgenden Anforderungen an den PDP:

1. Der PDP nimmt eingehende RADIUS-Anfragen an, versteht, verarbeitet und beantwortet sie.

2. Der PDP liest in RADIUS gekapselte EAP-Nachrichten aus, verarbeitet und erstellt sie.

3. Der PDP unterstützt bei der Verarbeitung der EAP-Nachrichten als speziellen EAP-Typen EAP-TNC.
4. Der PDP übermittelt TNCCS-Nachrichten vom NAA zum TNC Server und zurück.

Während sich die ersten drei Anforderungen auf die NAL beziehen, befasst sich die vierte Anforderung mit der in den TNC-Spezifikationen nicht behandelten Schnittstelle zwischen NAA und TNC Server.

6.1.2 Lösungsansätze

Für die Lösung dieser Problemstellung ergeben sich vier Ansätze für Architekturen, welche im Folgenden vorgestellt und mit ihren Vor- und Nachteilen bewertet werden. Abbildung 6.1 zeigt diese vier Architekturansätze als jeweilige Deployment-Diagramme.

Vollständige Eigenimplementierung

Beim ersten Ansatz (Architektur in Abbildung 6.1 oben links) wird eine neue Gesamtapplikation erstellt, in welcher NAA und TNC Server enthalten sind und die IMV-Komponenten als externe Shared Objects einbinden. Dies bedeutet, dass die gesamte RADIUS-Funktionalität von Grund auf neu entwickelt wird und die Schnittstelle zwischen NAA und TNC Server intern verläuft. Alle vier Anforderungen aus Kapitel 6.1.1 werden damit neu umgesetzt.

Die Vorteile dieses Ansatzes:

- Für die Entwicklung ist keine Einarbeitung in Fremdcode oder ein Fremd-Framework erforderlich, da die neue TNC-Funktionalität nirgendwo integriert werden muss.
- Die Entwicklung und das Build-Verfahren sind wie beim AR schlank möglich.
- Funktionalität kann, wo es sich anbietet (wie z.B. Teile der EAP-Kommunikation), problemlos vom AR übernommen werden, da beide in C++ entwickelte Applikationen darstellen.
- Die Schnittstelle zwischen NAA und TNC Server kann einfach über eine einfache Fassaden-Klasse realisiert werden.

Folgende Nachteile ergeben sich aber bei der vollständigen Eigenimplementierung:

- Stabile existierende Implementierungen wie z.B. die RADIUS-Funktionalität aus FreeRadius werden neu implementiert. Die Eigenimplementierung kann schwer die Reife der vielfach eingesetzten Applikationen erreichen.

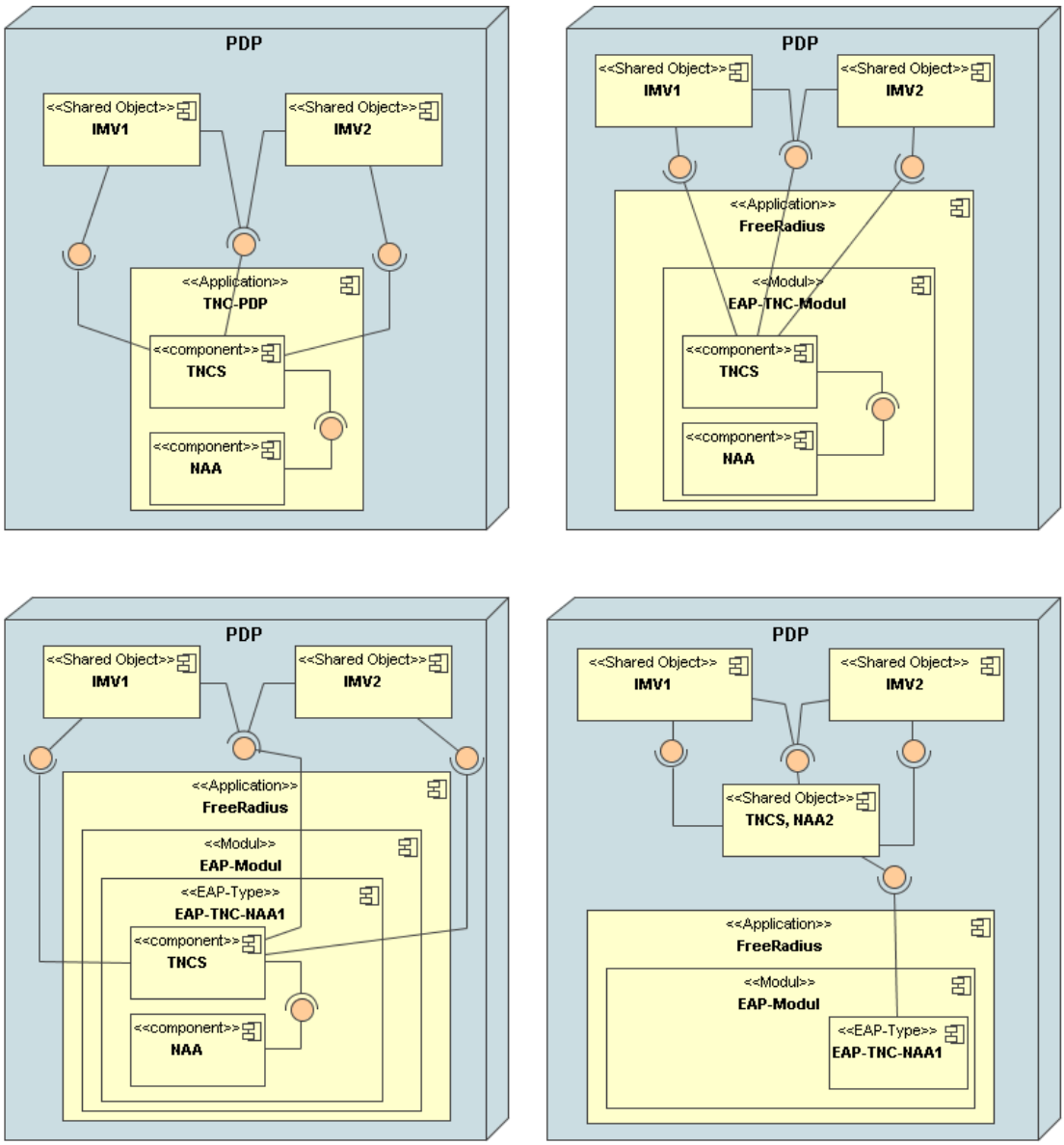


Abbildung 6.1: Realisierungsansätze für die Architektur des PDP

- Die Eigenimplementierung erfordert erheblichen Aufwand.

Der erhebliche Aufwand ist ein Hauptargument, diesen Architekturansatz zu verwerfen, zumal andere Ansätze viele Vorteile versprechen.

Erstellung eines neuen Moduls in FreeRadius

Der zweite Ansatz (Architektur in Abbildung 6.1 oben rechts) umfasst im Gegensatz zum ersten Ansatz keine vollständige Eigenimplementierung, sondern es wird die RADIUS-Funktionalität des FreeRadius-Servers genutzt, indem ein neues Modul dafür mit TNC-Funktionalität entwickelt wird, welches sowohl NAA als auch TNC Server umfasst. IMV-Komponenten werden wie im ersten Ansatz IF-IMV-konform als Shared Objects eingebunden. Die erste Anforderung aus Kapitel 6.1.1 wird damit von FreeRadius direkt erfüllt; die übrigen werden neu implementiert.

Folgende Vorteile zeigen sich bei diesem Ansatz:

- RADIUS-Funktionalität wird komplett von FreeRadius übernommen und muss nicht neu implementiert werden; damit ergibt sich „automatisch“ eine stabile RADIUS-Funktionalität mit vermindertem Entwicklungsaufwand.

- Die Schnittstelle zwischen NAA und TNC Server kann einfach intern realisiert werden.

- Die Erstellung von FreeRadius-Modulen ist einfach möglich und gut dokumentiert.

Dem gegenüber stehen die Nachteile:

- NAA und TNC Server müssen in C entwickelt werden, da in C++ entwickelte Module schwer bzw. unmöglich in FreeRadius integriert werden können. Damit ist eine Wiederverwendung von Code aus dem AR unmöglich und das Design der Applikation ohne Objekt-orientierte Konzepte erschwert.

- EAP-Funktionalität muss nach wie vor vom Entwickler implementiert werden, womit dort mit erheblichem Aufwand und Fehleranfälligkeit zu rechnen ist.

- Das Build-Verfahren des PDP muss in das von FreeRadius integriert werden, was die Entwicklung erschwert und schwerfälliger macht.

Die mangelnde Wiederverwendbarkeit von Komponenten zwischen AR und PDP und die prozedurale Entwicklung geben den Ausschlag, diesen Ansatz zu verwerfen.

Erweiterung des EAP-Moduls in FreeRadius

Der dritte Ansatz (Architektur in Abbildung 6.1 unten links) besteht darin, das in FreeRadius existierende EAP-Modul um einen neuen EAP-Typen, nämlich EAP-TNC, zu erweitern. Dieses „Typ-Modul“ umfasst dann wiederum NAA und TNC Server. Die Anforderungen 1 und 2 aus Kapitel 6.1.1 werden damit direkt von FreeRadius erfüllt. Bei der Anbindung von IMV-Komponenten ändert sich zu den bisherigen Architekturvorschlägen nichts.

Dieser Ansatz zeigt einige Vorteile:

- RADIUS-Funktionalität wird auch hier von FreeRadius übernommen und muss nicht neu implementiert werden, damit ergibt sich „automatisch“ eine stabile RADIUS-Funktionalität mit vermindertem Entwicklungsaufwand.
- Grundlegende EAP-Funktionalität wird ebenfalls von FreeRadius übernommen und muss nur um EAP-TNC-Spezifika erweitert werden.
- Die Schnittstelle zwischen NAA und TNC Server kann einfach intern realisiert werden.
- Die Erweiterung des EAP-Moduls ist relativ einfach möglich.

Die Nachteile sind:

- NAA und TNC Server müssen in C entwickelt werden, daraus folgt: keine Wiederverwendung von AR-Komponenten und keine Objekt-Orientierung.
- Das Build-Verfahren von FreeRadius muss genutzt werden.

Obwohl dieser Ansatz dem Entwickler bereits viel Arbeitserleichterung bringt, stört die Beschränkung auf C als Programmiersprache. Um dieses Manko zu beheben, ist eine kleine Änderung vorzunehmen, welche in den vierten Ansatz einfließt.

Erweiterung des EAP-Moduls in FreeRadius mit Anbindung eines externen TNC Servers

Der vierte Architekturvorschlag (Architektur in Abbildung 6.1 unten rechts) ist eine Erweiterung des gerade vorgestellten dritten Vorschlags: Anstatt den TNC Server (und große Teile der NAA) in das „Typ-Modul“ des EAP-Typs in FreeRadius zu integrieren, wird dieser extern entwickelt und als Shared Object angebunden. Diese Architektur bietet die gleichen Vorteile wie der vorherige Vorschlag, eliminiert aber einen entscheidenden Nachteil:

Der größte Teil des PDP kann nun in C++ entwickelt werden, sofern man den Teil innerhalb des „Typ-Moduls“ soweit wie möglich auf Netzwerk-nahe Aufgaben (EAP-TNC-Paket-Erstellung, -Auslesen, -Versand) eingrenzt und weitergehende Aufgaben (auch teilweise aus dem NAA-Bereich) ins Shared Object (in C++

entwickelt) verlagert. Damit wird die Wiederverwendung von Klassen zwischen AR und PDP ermöglicht.

Damit bleiben nur noch folgende Nachteile erhalten:

- Das Build-Verfahren von FreeRadius muss genutzt werden.
- Es existiert eine externe Schnittstelle innerhalb der NAA, die das Modul in der FreeRadius-Applikation mit dem Shared Object verbindet.

6.1.3 Gewählte Architektur

Da der vierte Architekturvorschlag die meisten Vorteile bietet und die entscheidenden Nachteile eliminiert, wird dieser für die Umsetzung gewählt. Diese gewählte Architektur ist noch detaillierter in Abbildung 6.2 dargestellt. In dieser Abbildung ist die Schnittstelle zwischen dem oberen Teil der NAA und TNC Server eingefügt, welche innerhalb des „Shared Object“ verläuft. Diese Architektur wird im Folgenden näher spezifiziert.

Ein detailliertes Klassendiagramm des PDP mit allen Klassen im Überblick findet sich im Anhang in Kapitel A.2.

6.1.4 Fassaden innerhalb des PDP

Eine der nicht-funktionalen Anforderungen an die entwickelte Software besteht darin, eine möglichst lose Kopplung zwischen den einzelnen Komponenten zu erreichen. Dafür sind auf dem PDP zwischen den einzelnen Schichten schlanke Fassaden konzipiert, welche den einzigen Zugriffspunkt aus anderen Schichten darstellen. Abbildung 6.3 zeigt diese Fassaden.

Weil der PDP zuerst in der NAL aktiv wird, wenn eine AR-Anfrage über RADIUS gestellt wird, läuft die Kommunikationsrichtung von unten nach oben durch die Schichten, so dass obere Schichten der jeweils darunter liegenden Schicht Fassaden zur Verfügung stellen. Die Komponente `NAA2` innerhalb der Komponente `TNCS, NAA2` bietet die Fassade `TNCSBind` an und ermöglicht es damit dem EAP-TNC-Modul aus FreeRadius heraus, TNCCS-Nachrichten oder TNCCS-Fragmente (zzgl. Verbindungsinformationen) in das in C++ entwickelte Shared Object über die Methode `exchangeTNCCSMessages()` zu übertragen und eine neue TNCCS-Nachricht oder TNCCS-Fragment als Antwort zu erhalten.[1]

Die Kommunikation zwischen den zwei Komponenten innerhalb `TNCS, NAA2` läuft über die Schnittstelle der Klasse `TNCSFlowControl`. Über diese Schnittstelle werden nur vollständige TNCCS-Nachrichten ausgetauscht.

[1] Alternativ zu der TNCCS-Nachricht kann auch ein Acknoledgement zurückgegeben werden, wenn die eingehende Nachricht fragmentiert ist, aber dazu später mehr.

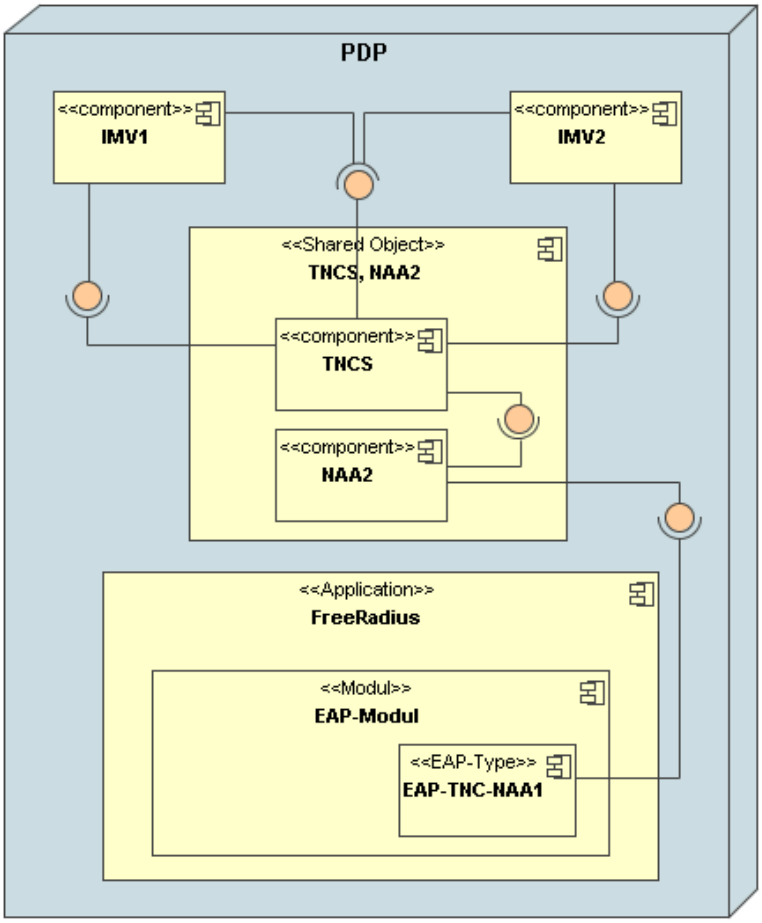


Abbildung 6.2: Realisierte Gesamtarchitektur des PDP

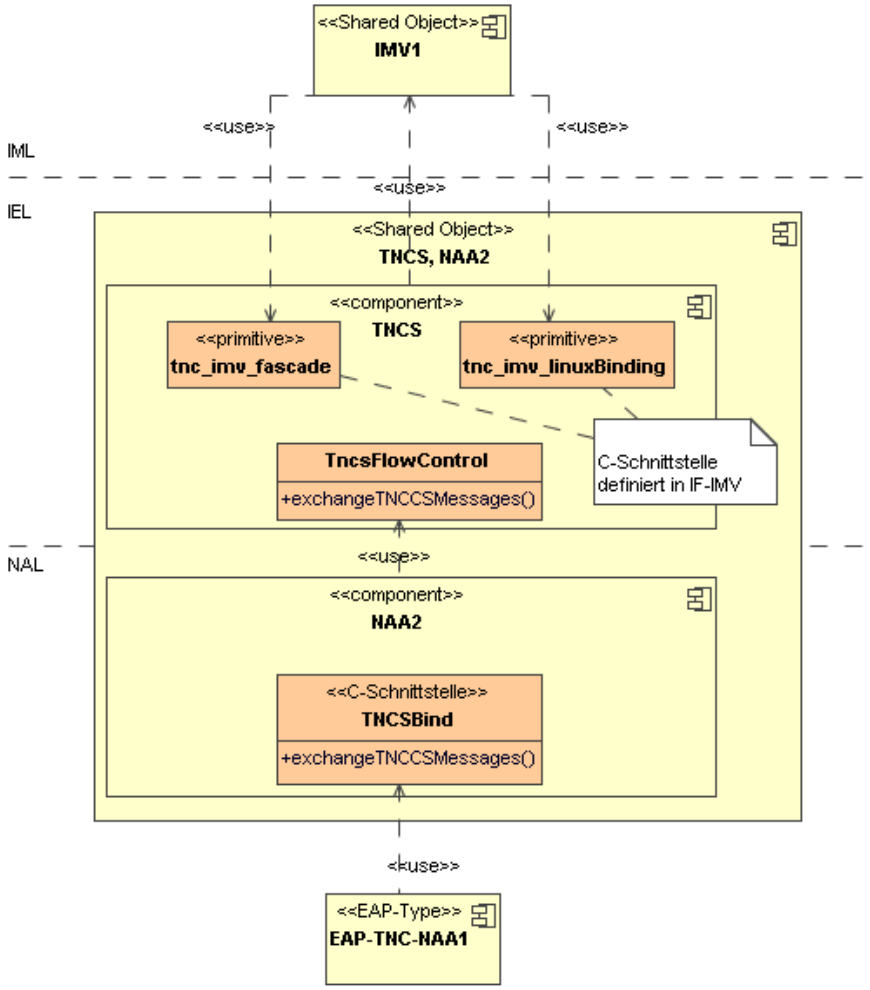
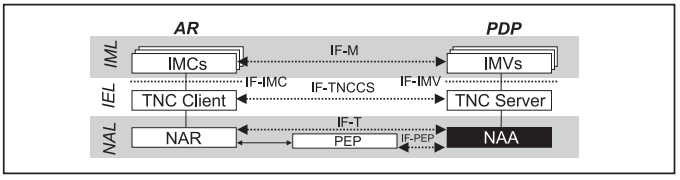


Abbildung 6.3: Fassaden zwischen den Komponenten innerhalb des PDP

Die Schnittstelle zwischen IMVs und TNC Server ist gemäß der IF-IMV-Spezifikation als C-Schnittstelle realisiert, wobei im TNC Server zwischen der generellen Schnittstelle und der Linux-spezifischen aus dem definierten Linux-Binding unterschieden wird. Für die IMVs bleibt diese Unterscheidung aber transparent.

6.2 Network Access Authority



6.2.1 Erweiterung des EAP-Moduls von FreeRadius

In der gewählten Architektur muss das EAP-Modul von FreeRadius um folgende Anforderungen erweitert werden:

- EAP-TNC-Nachrichten (gemäß IF-T) werden empfangen und interpretiert.
- Inhalt (TNCCS) und Meta-Informationen (Flags) werden an den TNC Server weitergeleitet und Antworten von ihm entgegengenommen.
- EAP-TNC-Nachrichten werden aus der Antwort des TNC Servers generiert und versendet.
- Gemäß der Empfehlung des TNC Servers werden (entsprechend IF-PEP) die RADIUS-Attribute zur Einordnung des AR ins passende VLAN gesetzt.

Um diese Anforderungen zu erfüllen, wird dem EAP-Modul ein neuer Typ mit Namen „EAP_TNC“ hinzugefügt. Da die Implementierung, wie bereits erläutert, im FreeRadius-Modul in C erfolgt, wird die Funktionalität innerhalb des Moduls bewusst auf die nötigsten Funktionen beschränkt. Weitergehende Aufgaben, wie z.B. die Fragmentierung der Nachrichten, wird ausgelagert. Ein Überblick über die Dateien und Methoden findet sich im Architekturüberblick in Kapitel 6.2.2, Implementierungsdetails und Konfigurationsmaßnahmen in FreeRadius später in Kapitel 9.1.3.

6.2.2 Architekturüberblick

In Abbildung 6.4 sind die wichtigsten Quelldateien und Klassen der NAA dargestellt. Unterhalb der gestrichelten Linie sind die Quelldateien des EAP-TNC-Moduls (aus FreeRadius in C) abgebildet, oberhalb die C++-Klassen aus dem Shared Object.

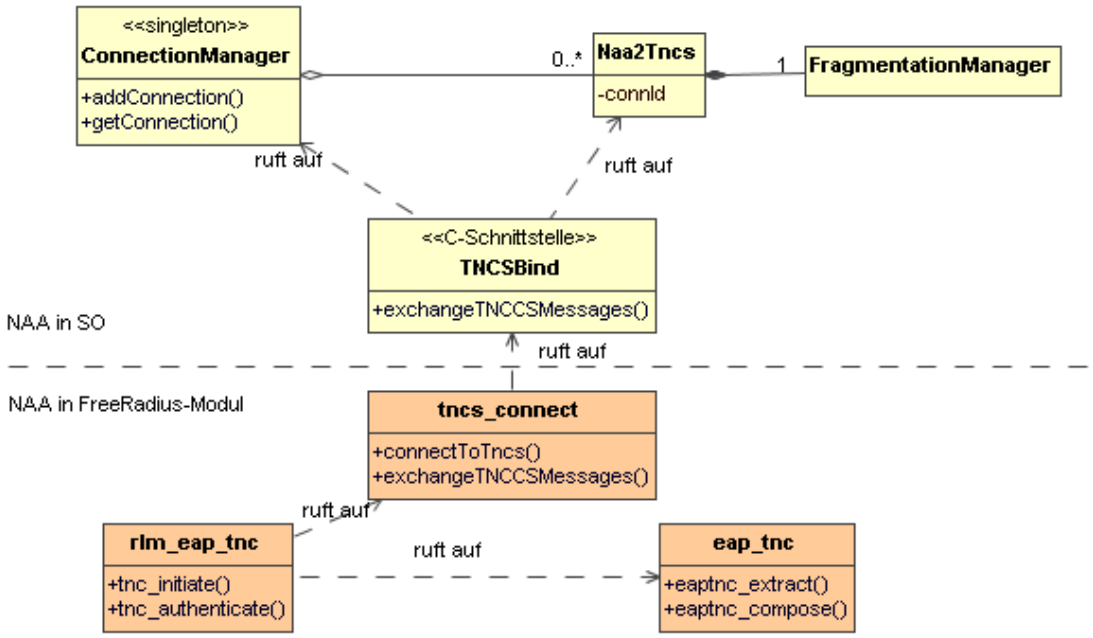


Abbildung 6.4: Statische Architektur der NAA-Komponente

Quelldateien aus FreeRadius

Die Quelldateien aus FreeRadius sind für folgende Aufgaben verantwortlich:

rlm_eap_tnc ist der externe Zugriffspunkt für das EAP-Modul. `tnc_initiate` wird beim jeweils ersten Aufruf eines Clients aufgerufen, `tnc_authenticate` bei jedem folgenden Aufruf. `tnc_authenticate` ist im ursprünglichen RADIUS-Sinne für die Authentisierung des Clients verantwortlich. Hier kümmert es sich um die Integritätsprüfung des AR und ruft die weiteren Methoden auf.

tnc_connect beinhaltet Methoden, welche den direkten Kontakt zum TNC Server (und dem weiteren NAA-Bereich) im Shared Object ermöglichen. `connectToTncs` stellt die initiale Verbindung her, indem es das SO lädt. `exchangeTNCCSMessages` ruft direkt `TNCSBind` auf.

eap_tnc beinhaltet Methoden, welche sich um die EAP-TNC-Spezifika kümmern. So extrahiert `eaptnc_extract` aus den binären (eingehenden) Daten des `EAP_HANDLER` das *EAP-TNC*-Paket und füllt die einzelnen Felder zur weiteren Verarbeitung in eine `Struct`. `eaptnc_compose` macht das Gegenteil und baut das binäre Paket „auf dem Rückweg“ wieder aus der Struct auf.

NAA-Klassen des Shared Object

Weitergehende Funktionalitäten sind in Klassen des Shared Objects implementiert:

TNCSBind stellt die C-Schnittstelle für das FreeRadius-Modul dar und nimmt TNCS-Anfragen von dort entgegen, leitet sie weiter und gibt die Antworten wieder zurück.

ConnectionManager gibt es als Objekt genau einmal und verwaltet alle Verbindungen, die gerade aktiv sind, indem er für jede Verbindung ein Objekt vom Typ `Naa2Tncs` hält. `ConnectionManager` lädt zudem die IMVs über den `LinuxIMVCollector` und informiert sie über neue oder beendete Verbindungen mit dem `IMVCommunicator`.

Naa2Tncs existiert für jede aktive Verbindung genau einmal und stellt die Verbindung zwischen Aufgaben, die NAA-nah sind, wie z.B. die Fragmentierung von TNCCS-Nachrichten, und dem reinen TNC Server (über `TncsFlowControl`) her. `Naa2Tncs` sorgt dafür, dass fragmentierte Nachrichten, die „von unten“ kommen, zusammengesetzt werden, bevor sie zur `TncsFlowControl` gelangen und fragmentiert zu große Nachrichten, welche aus der `TncsFlowControl` „nach unten“ gereicht werden.

FragmentationManager führt die von `Naa2Tncs` initiierte Fragmentierung und Defragmentierung aus. (Für Näheres zur Fragmentierung siehe Kapitel 7.3.2)

6.2.3 Ablauf in der NAA

Das Sequenzdiagramm in Abbildung 6.5 zeigt überblicksweise, wie die vorgestellten Funktionen und Klassen miteinander agieren:

Bei der ersten Anfrage eines bestimmten Clients ruft FreeRadius `tnc_initiate` auf (1). Wenn noch nicht geschehen lädt das EAP-TNC-Modul jetzt den Teil des PDP, der im Shared Object abgelegt ist, indem das Shared Object geladen wird (2). Wie später in Kapitel 9.2.5 beschrieben, wird eine neue ConnectionID (3) und gemäß IF-T ein binäres TNC Start-Paket ohne enthaltene TNCCS-Nachricht erzeugt (4), welches an FreeRadius zum Versand zurückgegeben wird (5).

Bei der nächsten Anfrage desselben AR wird von FreeRadius automatisch die Methode `tnc_authenticate` aufgerufen (6). Aus dem übermittelten Binär-Paket werden die Bestandteile des *EAP-TNC*-Pakets in eine `Struct` extrahiert (7). Die extrahierte TNCCS-Nachricht (wenn vorhanden) und weitere Meta-Informationen (Acknoledgement, Flags) gelangen zu `TNCSBind` (8), wo geprüft wird, ob bereits eine Verbindung für die eingehende Nachricht mit der ConnectionID besteht (9). Wenn nicht, wird eine neue Verbindung im `ConnectionManager` erzeugt und gespeichert (10), sowie (ob neu oder nicht) vom `ConnectionManager` geholt (11). Für die Verbindung gibt dieser das zugehörige `Naa2Tncs`-Objekt zurück (12), an das die Nachricht weitergeleitet wird (13).

Wenn die eingehende Nachricht fragmentiert oder ein Acknoledgement ist, regelt `Naa2Tncs` die Behandlung an dieser Stelle (14). Wenn die Nachricht vollständig (al-

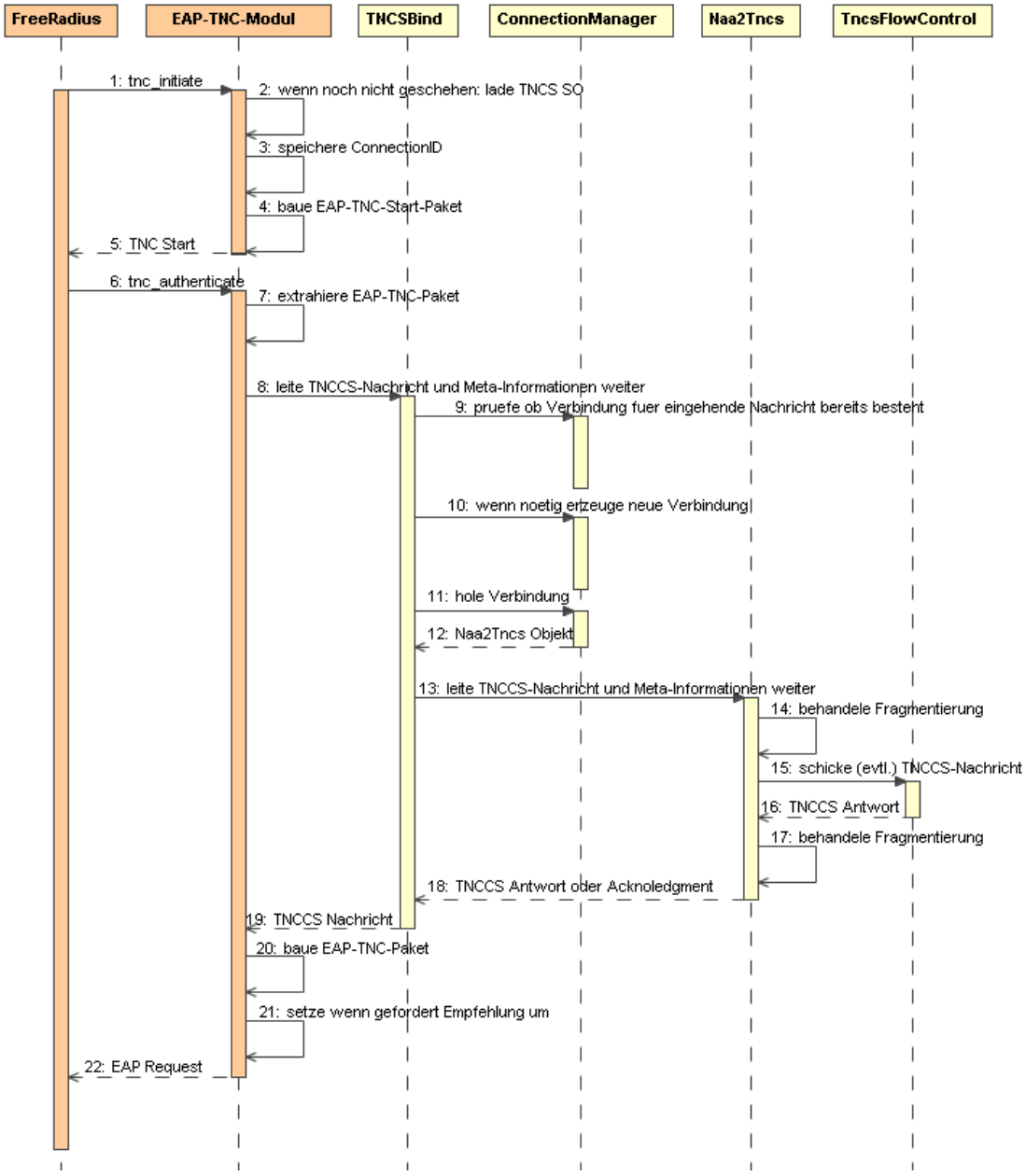
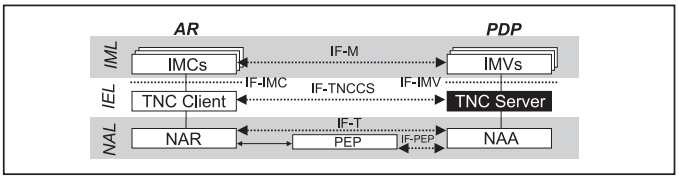


Abbildung 6.5: Abläufe in der NAA nach dem Empfang von RADIUS-Paketen

so nicht fragmentiert oder Fragmentierung beendet) ist, wird die TNCCS-Nachricht in den TNC Server übertragen (15). Der TNC Server schickt nach der Verarbeitung der eingehenden Nachricht eine vollständige TNCCS-Antwort zurück (16), welche fragmentiert wird, sollte sie zu groß sein (17). Daraufhin gibt `Naa2Tncs` eine TNCCS-Nachricht, ein TNCCS-Fragment oder ein Acknoledgment zurück an `TNCSBind` (18), welches ins EAP-TNC-Modul weitergeleitet wird (19).

Aus der empfangenen Nachricht wird nun ein neues binäres EAP-TNC-Paket gebaut (20). Wenn das Ende des Handshake vom TNC Server signalisiert wurde, werden zusätzlich die entsprechenden RADIUS-Attribute zur Durchsetzung der Empfehlung gesetzt (21). Abschließend wird das Paket an FreeRadius zur Aufteilung auf RADIUS-Attribute und zum Versand zurückgegeben (22).

6.3 TNC Server



6.3.1 Architekturüberblick

Die Architektur des TNC Servers ist als Klassendiagramm mit den wichtigsten Klassen in Abbildung 6.6 dargestellt.

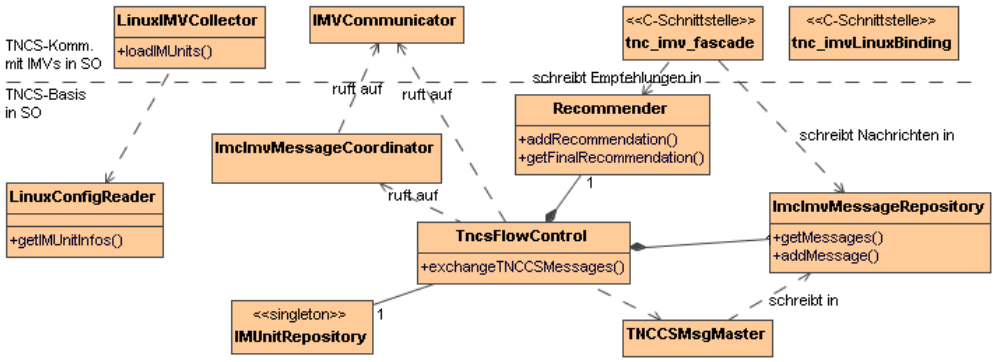


Abbildung 6.6: Statische Architektur des TNCS

Die Klassen sind vertikal angeordnet:

Die Klassen im Diagramm unter der gestrichelten Linie befassen sich mit dem korrekten Ablauf und Status innerhalb des TNC Server.

TncsFlowControl ist die zentrale Klasse, welche die TNCCS-Nachrichten von `Naa2Tncs` entgegennimmt und die weitere Verarbeitung anschiebt.

TNCCSMsgMaster liest die TNCCS-Nachricht aus und speichert die gefundenen IMC-IMV-Nachrichten in `ImcImvMessageRepository`. Zudem erzeugt `TNCCSMsgMaster` aus den vorliegenden Nachrichten (kommend von den IMVs) und wenn vorhanden aus der Verfahrensempfehlung die passende TNCCS-Nachricht als XML.

ImcImvMessageRepository speichert sowohl die IMC-IMV-Nachrichten von den lokalen IMV-Komponenten, als auch die IMC-IMV-Nachrichten von den entfernten IMC-Komponenten.

IMUnitRepository hält Informationen zu jeder registrierten IMV-Komponente darüber bereit, wie ihre Methoden angesprochen werden können (Zeiger auf die Funktionen).

LinuxConfigReader liest die Konfigurationsdatei `tnc_config` aus, in der wie in IF-IMV beschrieben, die Informationen über die verfügbaren IMV-Komponenten vorliegen.

Recommender speichert die von den IMV-Komponenten eingegangenen Empfehlungen über die Behandlung des AR und erzeugt nach einer Sicherheitsrichtlinie aus den vorliegenden Empfehlungen eine Gesamtempfehlung.

Die Klassen im Diagramm über der gestrichelten Linie stehen in direkter Kommunikation mit den IMV-Komponenten:

LinuxIMVCollector lädt die IMV-Komponenten beim Serverstart anhand der Informationen von `LinuxConfigReader`.

IMVCommunicator ruft die in IF-IMV spezifizierten Funktionen der IMV-Komponenten auf.

tnc_imv_fascade stellt zusammen mit `tnc_imv_linuxBinding` den IMV-Komponenten die in IF-IMV spezifizierten Funktionen zum Aufruf zur Verfügung.

6.3.2 Ablauf im TNC Server

Im folgenden soll ein kurzer Überblick über die Interaktion der einzelnen Klassen im TNC Server gegeben werden, indem der Ablauf gezeigt wird, der nahtlos an die dargestellten Abläufe in der NAA (Abbildung 6.5 auf Seite 62, Nummer 15) anschließt.

Abbildung 6.7 zeigt diese Abläufe im Sequenzdiagramm.

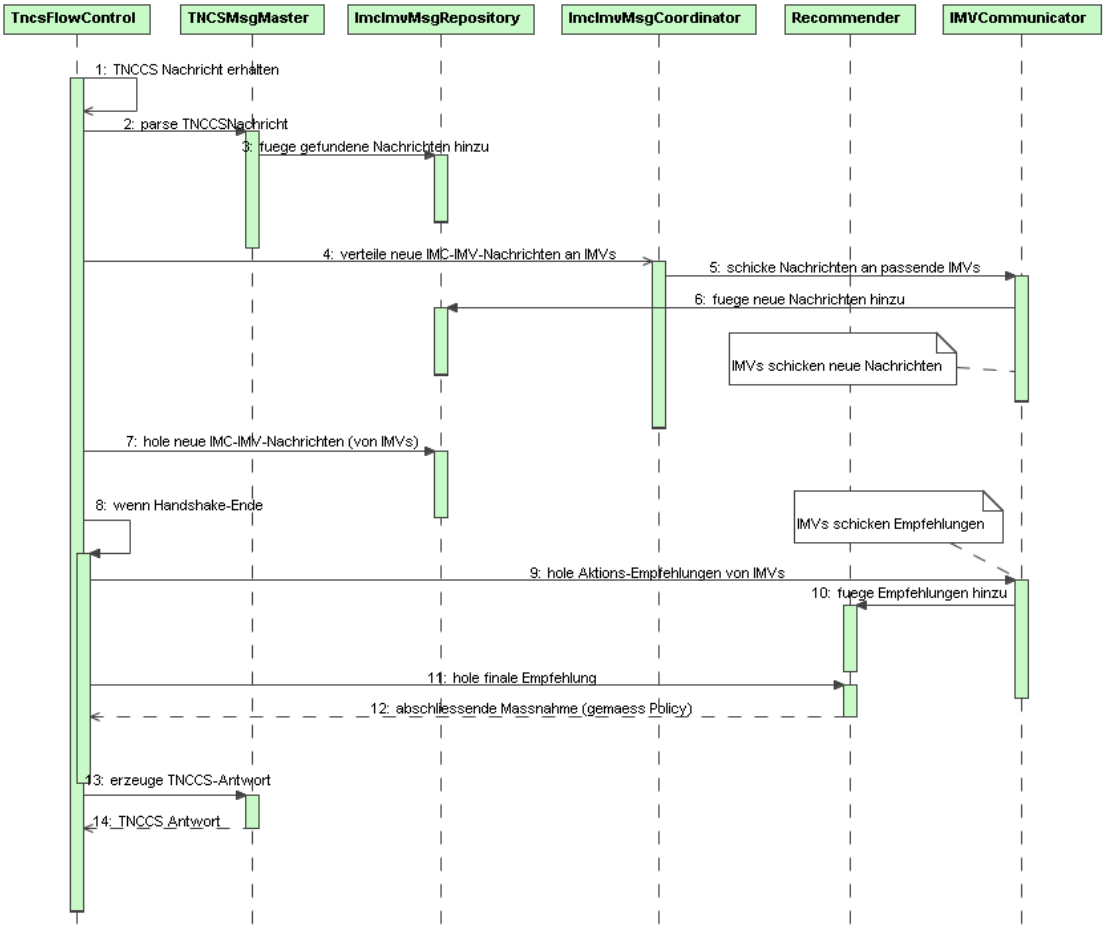


Abbildung 6.7: Ablauf im TNC Server zwischen Erhalt und Versand einer TNCCS-Nachricht

Aus der NAA hat `TncsFlowControl` eine vollständige TNCCS-Nachricht zur Verarbeitung erhalten (1). Diese Nachricht wird `TNCSMsgMaster` zum Auslesen übergeben (2). Die gefundenen und extrahierten IMC-IMV-Nachrichten fügt er in die `ImcImvMsgRepository` ein (3). `ImcImvMsgCoordinator` bekommt den Auftrag, diese Nachrichten an die passenden IMVs zu verteilen (4), woraufhin er die passenden IMVs identifiziert und diese mit den zugehörigen Nachrichten dem `IMVCommunicator` übermittelt (5). `IMVCommunicator` schickt diese Nachrichten dann an die IMVs, welche daraufhin neue Nachrichten übermitteln, welche wiederum in die `ImcImvMsgRepository` eingetragen werden (6). Nachdem dies abgeschlossen ist, holt `TncsFlowControl` diese neuen Nachrichten für die IMCs aus der `ImcImvMsgRepository` (7).

Falls das Handshake-Ende erreicht ist (8) (vgl. Kapitel 6.3.3) holt der `IMVCommunicator` die Aktions-Empfehlungen von den IMVs ein (9), welche dem `Recommender` übermittelt werden (10). Dieser gibt gemäß seiner Sicherheitsrichtlinie (vgl. Kapitel 6.3.3) eine Gesamtempfehlung ab (11), die die abschließende Maßnahme festlegt (12).

Entweder aus den neuen Nachrichten oder der Empfehlung des `Recommender` erzeugt der `TNCSMsgMaster` die TNCCS-Antwort (13) und gibt diese an `TncsFlowControl` zurück (14). Diese Nachricht entspricht derjenigen in Punkt 16 auf Abbildung 6.5, Seite 62.

6.3.3 Handshake-Ende und Policy-Management

Ende des Handshake

In den TNC-Spezifikationen ist nicht festgelegt, wie viele Batches zwischen TNC Client und TNC Server innerhalb eines Handshake ausgetauscht werden und wann IMVs ihre Empfehlungen abzugeben haben.

Deshalb wird für die hier vorgestellte Implementierung festgelegt und umgesetzt, dass der Handshake zu dem Zeitpunkt endet,

- wenn alle IMVs bereits eine Empfehlung abgegeben haben,
- wenn kein IMV mehr (trotz Aufforderung) neue IMC-IMV-Nachrichten in den TNC Server überträgt oder
- wenn eine bestimmte (einstellbare) Anzahl von Batches erreicht ist (vgl. Kapitel 9.3).

Durch diese Festlegungen wird erreicht, dass ein Handshake nicht unverhältnismäßig lange läuft und durch schlecht programmierte IMVs/IMCs in endlose Rekursionen geraten kann. Beides vermindert zusätzlich die Last auf dem PDP.

Policy-Management

Wenn der Handshake für eine Verbindung beendet ist, muss der TNC Server entscheiden, wie er mit dem dazugehörigen AR verfahren möchte. In IF-IMV wird spezifiziert, wie die einzelnen IMV-Komponenten ihre Empfehlungen an den TNC Server geben. Somit erhält der TNC Server von jeder IMV-Komponente eine der vier folgenden Empfehlungen:

- Zutritt erlauben
- Zutritt isoliert (zur Optimierung der Integrität) erlauben
- Zutritt verwehren
- keine Empfehlung möglich

Der letzte Fall kann z.B. dann eintreten, wenn die zugehörige IMC-Komponente auf dem AR nicht installiert oder aktiviert ist und deshalb die IMX-Komponente keine Integritätsinformationen liefert, mit denen sich die IMV-Komponente ein Urteil bilden kann. Optional kann die IMV-Komponente ihre Empfehlung durch Unterkategorien begründen und damit z.B. mitteilen, dass die Integritätsverletzung schwerwiegend oder weniger schwerwiegend ist.

Die Aufgabe des TNC Server besteht nun darin, diese Empfehlungen zu einem Gesamturteil über die Integrität des AR zusammenzufassen. Dies geschieht üblicherweise anhand einer Sicherheitsrichtlinie, die beliebig komplex sein kann. Die hier entworfene Sicherheitsrichtlinie basiert auf der Annahme, dass nicht bekannt ist, welche und wieviele IMVs im TNC Server geladen sind und bietet in diesem Rahmen schon relativ viele Möglichkeiten der Konfiguration.

Die Richtlinie bietet sechs verschiedene Sicherheitsstufen, welche darauf basieren, dass entweder

- das Votum mindestens einer IMV-Komponente ausschlaggebend ist,
- das Votum der Mehrheit der IMV-Komponenten ausschlaggebend ist,
- oder das Votum der IMV-Komponenten einstimmig erfolgen muss.

Diese drei Möglichkeiten werden in zwei verschiedene Richtungen betrachtet:

- Von der Sperrung über Isolation zur Freigabe der Verbindung (negativ) und
- von der Freigabe über die Isolation bis zur Sperrung der Verbindung (positiv).

Damit ergeben sich sechs Sicherheitsstufen, welche in einzelnen Ablaufdiagrammen dargestellt werden können.

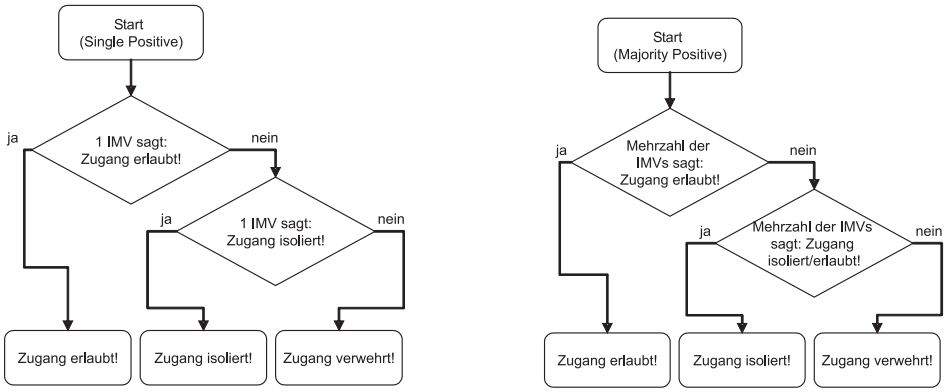


Abbildung 6.8: Die Richtlinien „Einfache Zustimmung“ und „Mehrheitszustimmung“ als Flussdiagramm

Abbildung 6.8 zeigt das Ablaufdiagramm für die Stufe Single-Positive (links im Bild), in der zunächst vom positiven Fall (Zugriff erlaubt) ausgegangen wird und das Votum einer IMV-Komponente entscheidend ist: Zunächst wird geprüft, ob eine IMV-Komponente die Empfehlung gegeben hat, den Zutritt zu erlauben. Wenn dies der Fall ist, wird der Zutritt gewährt; wenn nicht, wird bei einer einzigen Isolationsempfehlung der AR isoliert. Wenn auch keine Isolationsempfehlung gegeben wurde, wird der Zutritt gesperrt.

Analog dazu funktionieren die übrigen Stufen, welche in den Abbildungen 6.8 (rechts), 6.9 und 6.10 dargestellt sind.

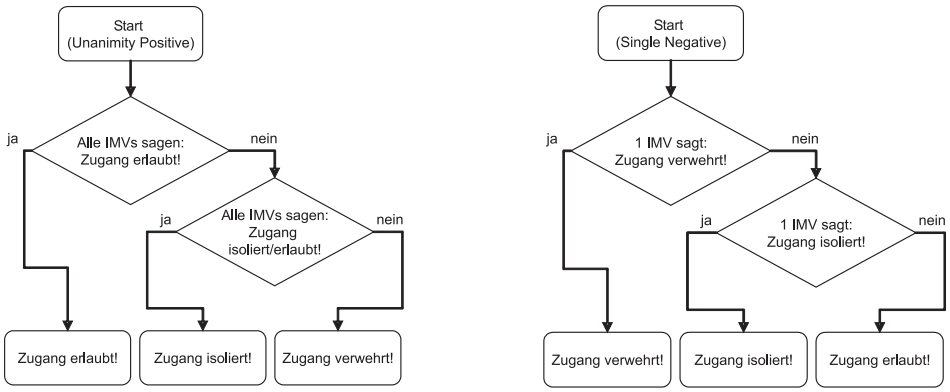


Abbildung 6.9: Die Richtlinien „Einstimmige Zustimmung“ und „Einfache Ablehnung“ als Flussdiagramm

Mehr lässt sich unter der Voraussetzung der unbekannten IMVs nicht umsetzen. Für eine detailliertere Sicherheitsrichtlinie müssen die einzelnen IMVs, ihr Sinn

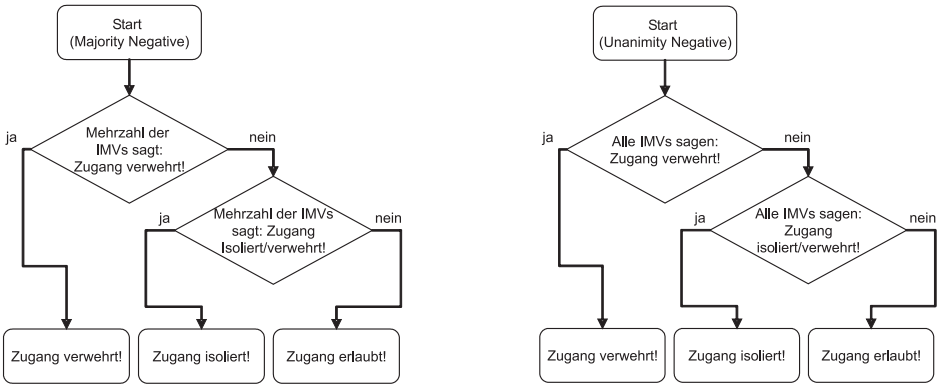


Abbildung 6.10: Die Richtlinien „Mehrheitsablehnung“ und „Einstimmige Ablehnung“ als Flussdiagramm

und Zweck, sowie ihre Relevanz für die Gesamtsicherheit betrachtet werden, was im Rahmen dieser Arbeit nicht möglich ist, aber im Fokus weiterer Untersuchungen stehen kann (siehe Ausblick in Kapitel 12.2.3).

6.4 Konzept für Unterstützung paralleler Anfragen

Im Wesen von allen Client- und Server-Architekturen liegt, dass sich mehrere Clients parallel an einen Server wenden können. Genauso können selbstverständlich auch mehrere AR parallel Anfragen an einen PDP stellen. Dieses muss im Design des PDP berücksichtigt werden, besonders weil C++ von selbst kein Multi-Threading kennt und die Basis-Bibliotheken nicht „Thread-safe“ konzipiert sind.

Deshalb werden im Design des PDP zwei Prinzipien umgesetzt:

1. Für jede Verbindung von einem AR werden, wenn möglich, eigene Objekte aller Klassen erstellt. Damit wird sichergestellt, dass keine parallelen Zugriffe auf ein Objekt erfolgen, was im schlimmsten Fall zu Inkonsistenzen und Speicherzugriffsfehlern führen kann.

2. Wo Objekte gemeinsam genutzt werden müssen, wird der Zugriff auf gefährdete Methoden solange für andere Aufrufer gesperrt, wie ein Aufrufer die Methode ausführt. Methoden sind gefährdet, wenn Schreibzugriffe auf die Attribute der Objekte (vor allem in Vektoren) durchgeführt werden. Lesende Zugriffe sind dagegen unproblematisch.

Um das erste Prinzip umzusetzen, wird für jede begonnene Verbindung ein Objektnetz aufgebaut, welches als grundlegende Instanz ein Objekt der Klasse `Naa2Tncs`

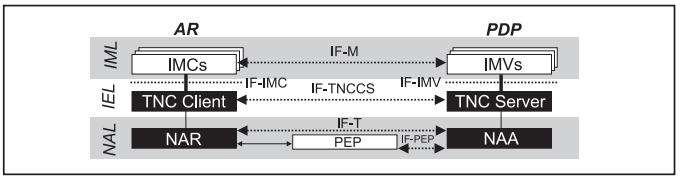
besitzt (vgl. Abbildung A.4 auf Seite A.4 im Anhang). Objekte der Klassen `TncsFlowControl`, `TNCCSMsgMaster`, `Recommender` und `ImcImvMessageRepository` existieren damit für jede einzelne Verbindung neu, wodurch der Zugriff auf diese Objekte nur für eine einzelne Verbindung erfolgt und keine parallelen Zugriffe möglich sind.

Problematisch sind solche Klassen, welche nur als einzelnes Objekt instanziiert werden, weil sie gemeinsame Informationen für alle Verbindungen beinhalten. Dies sind im Wesentlichen `ConnectionManager` und `IMUnitRepository`. `ConnectionManager` muss in einer Instanz existieren, weil es den zentralen Zugriffspunkt für Anfragen aus der NAA darstellt und erst mit seiner Information die ankommende Nachricht in das passende „Objektnetz“ für die entsprechende Verbindung geleitet wird.

`IMUnitRepository` beinhaltet die Informationen über die registrierten IMVs. Da die IMVs nur einmal bei Serverstart geladen werden (und nicht für jede Verbindung neu), liegen die Informationen zentral vor.

In beiden Fällen sind diejenigen Methoden von Bedeutung, in denen Objekte hinzugefügt und entfernt werden (einmal Verbindungsobjekte vom Typ `Naa2Tncs`, beim anderen Mal `IMUnit`-Objekte). Diese Methoden müssen dahingehend geschützt werden, dass sie nicht parallel aufgerufen werden können. Dies wird mit Hilfe von Semaphoren (siehe [Sun05]) realisiert.

7 Client/Server-Übergreifende Konzepte



Bisher wurden Konzepte vorgestellt, die für TNC Server und TNC Client jeweils spezifisch sind. Nun soll es um diejenigen Konzepte gehen, die sowohl für TNC Server als auch TNC Client gelten:

Die Entwicklung von Client und Server soll möglichst stark Plattformabhängigkeiten kapseln, um eine Portierung auf eine andere Plattform so einfach wie möglich zu machen. Dafür wird in der hier vorgestellten Implementierung das Factory-Pattern[1] eingesetzt, welches in Kapitel 7.1 vorgestellt wird.

Zwei große Konzepte unterscheiden sich auf TNC Client- und Server-Seite nicht bzw. nur unwesentlich in der Fachlichkeit und werden deshalb von Klassen realisiert, welche sowohl auf Client- als auch Serverseite genutzt werden[2]: Die Anbindung der IM-Schicht wird in Kapitel 7.2 und die Behandlung von TNCCS-Nachrichten mit ihrer Fragmentierung in Kapitel 7.3 vorgestellt.

7.1 Konzepte für Plattformunabhängigkeit

In der TNC-Architektur gibt es zwei Bereiche, die abhängig davon unterschiedlich entwickelt werden müssen, ob Windows oder Linux die Plattform bildet:

- Der Zugriff auf das Netzwerk funktioniert von Plattform zu Plattform unterschiedlich, da direkter Schreib- und Lesezugriff auf das Netz auf Ethernet-Ebene ein sehr systemnaher Vorgang ist (vgl. hierzu Kapitel 5.2.2).
- Die Anbindung der IMCs bzw. IMVs funktioniert plattform-abhängig, weil hier Bibliotheken geladen werden müssen. In der Windows-Welt werden DLLs geladen, wogegen die Linux-Welt die Shared Objects kennt.

[1] Eine Beschreibung des Factory-Pattern findet sich in [JD03, S. 165]
[2] Diese Klassen finden sich in beiden Implementierungen im Unterverzeichnis `shared`.

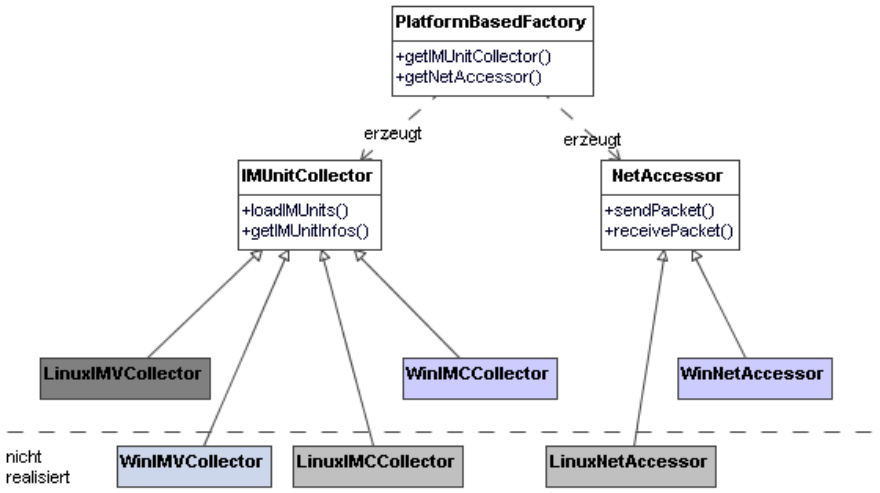


Abbildung 7.1: Konzept der plattformabhängigen Erzeugung über Fabrikmuster

Trotz dieser Unterschiede soll der Rest der Applikation möglichst nichts von den Unterschieden mitbekommen. Um dies zu erreichen existiert in der vorgestellten Implementierung die Klasse `PlattformBasedFactory`, deren Aufgabe darin besteht, Objekte für die gerade eingesetzte Plattform zurückzuliefern. Die Klassen dieser Objekte erweitern eine plattform-unabhängige Klasse, so dass für den Aufrufer transparent bleibt, welche Plattform im Hintergrund angesprochen wird. Abbildung 7.1 zeigt diese Klasse mit den von ihr erzeugbaren Objekten.

So liefert die `PlattformBasedFactory` für den Aufruf `getNetAccessor` ein Objekt vom Typ `WinNetAccessor` oder `LinuxNetAccessor`, je nachdem auf welcher Plattform die „Fabrik“ eingesetzt wird. Da aber sowohl `WinNetAccessor` als auch `LinuxNetAccessor` die Klasse `NetAccessor` erweitern[3], können die darin definierten Methoden unabhängig von der eigentlichen Implementierung genutzt werden. Da auf TNC Server FreeRadius die Anbindung des Netzwerks übernimmt, wird `PlattformBasedFactory` nur im TNC Client für die Erzeugung eines `NetAccessor` genutzt.

Genauso erfolgt die Anbindung der Schicht der IMCs und IMVs unabhängig davon, ob Windows oder Linux die Plattform bilden. Dort wird aber zusätzlich zwischen der Anbindung von IMVs und IMCs unterschieden, was im Detail in Kapitel 7.2.1 erläutert wird.

Mit Hilfe dieses Konzepts ist es beispielsweise leicht möglich, den TNC Client auf Linux zu portieren. Implementiert werden muss dafür nur ein `LinuxIMCCollector` und ein `LinuxNetAccessor`.

[3]Man kann `NetAccessor` auch als Schnittstelle definieren, nur gibt es keine Schnittstellen (wie z.B. in Java) in C++.

7.2 Anbindung der IM-Schicht

Die Anbindung der IM-Schicht funktioniert auf TNC Client und Server sehr ähnlich. Das liegt daran, dass sich die Spezifikationen IF-IMC und IF-IMV auch sehr ähneln[4].

Deshalb wird, so weit es geht, die Funktionalität zur Anbindung von IMCs und IMVs in gemeinsamen Klassen realisiert und sowohl im TNC Server als auch im TNC Client genutzt. Dafür wird eine neue Abstraktionsebene oberhalb von IMC und IMV eingeführt: `IMUnit`, also eine Einheit im Integrity Measurement Layer. `IMC` und `IMV` sind spezielle Erweiterungen von IMUnit, indem sie beispielsweise Zeiger auf unterschiedliche Funktionen beinhalten. Abbildung 7.2 zeigt alle Klassen, die sich mit der Anbindung der IML beschäftigen. Die weiß hinterlegten Klassen sind dabei plattform-unabhängig.

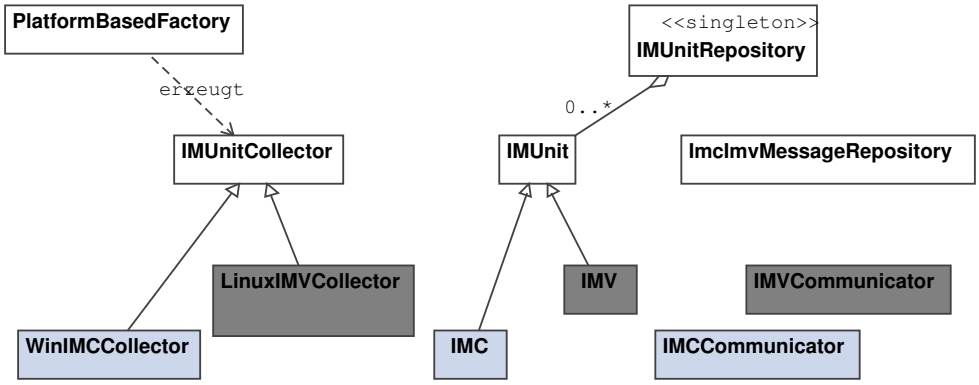


Abbildung 7.2: Klassen für die Anbindung der IML

`ImcImvMessageRepository` enthält IMC-IMV-Nachrichten; da sowieso Nachrichten in beide Richtungen (von IMC zu IMV und umgekehrt) unterstützt werden müssen, kann diese Klasse sowohl in TNC Server als auch TNC Client genutzt werden. Das Laden der IMCs und IMVs läuft in Grundzügen gleich, aber in Details doch unterschiedlich (zusätzlich noch abhängig von der Plattform) ab. Deshalb wird Grundfunktionalität in `IMUnitCollector` inklusive der Schnittstelle nach außen bereitgestellt und der eigentliche Ladevorgang in den abgeleiteten Klassen realisiert.

`IMVCommunicator` und `IMCCommunicator` dagegen beinhalten zu viele unterschiedliche Methoden[5], so dass sie unabhängig von einander die Kommunikation mit ihren IM-Einheiten regeln.

[4] Häufig besteht der Unterschied nur zwischen „V“ und „C“.

[5] Allein die Datentypen werden in IF-IMC und IF-IMV unterschiedlich benannt, auch wenn sie den gleichen Datentypen im Hintergrund darstellen, so dass sich sämtliche Methoden-Signaturen unterscheiden.

7.2.1 Ablauf bei der Kommunikation mit IM-Komponenten am Beispiel TNC Client

Abbildung 7.3 zeigt den Ablauf bei der Kommunikation mit den Komponenten der IM-Schicht am Beispiel des TNC Clients.

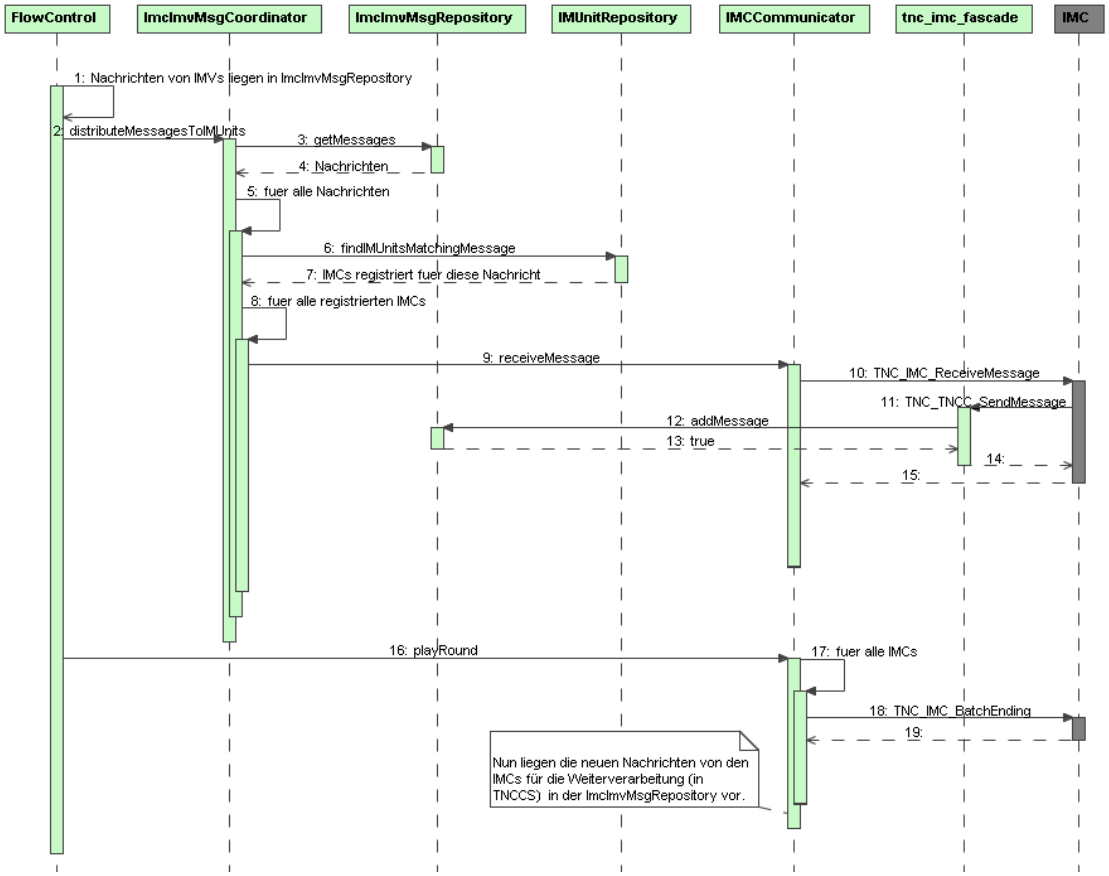


Abbildung 7.3: Kommunikationsablauf zwischen IMC-Komponenten und TNC Client

Die Ausgangssituation (1) besteht darin, dass die Nachrichten von den IMV-Komponenten bereits in der `ImcImvMsgRepository` liegen.[6] Dem `ImcImvMsgCoordinator` wird aufgetragen, die Nachrichten an die IMCs zu verteilen (2), woraufhin dieser die Nachrichten von `ImcImvMsgRepository` holt (3+4) und für jede dieser Nachrichten (5) aus der `IMUnitRepository` diejenigen IMCs ausliest, welche sich für den Nachrichtentyp der Nachricht registriert hat (6+7). Für alle diese IMCs (8) übermittelt `ImcImvMsgCoordinator` die Nachricht an `IMCCommunicator` zur Übertragung (9). Gemäß IF-IMC wird die Nachricht dann an den IMC übertragen (10) und der IMC überträgt die Antwort darauf (11), welche in `ImcImvMsgRepository`

[6] Wie die Nachrichten da hinein gelangen, wird im Kapitel 7.3.1 beschrieben.

gespeichert wird (12-15).

Abschließend beendet `IMCCommunicator` den Batch, indem er alle IMCs über das Batch-Ende informiert (16-19). Nun liegen die Nachrichten der IMCs zur Übertragung an den TNC Server abholbereit in `ImcImvMsgRepository`.

7.2.2 Unterschiede zwischen der Kommunikation mit IMCs und IMVs

Der in Abbildung 7.3 dargestellte Ablauf kann beinahe genauso im TNC Server ablaufen. Der einzige Unterschied besteht darin, dass nicht `IMCCommunicator`, sondern `IMVCommunicator` zum Einsatz kommt. Da der Ablauf aber für die übrigen Komponenten unverändert bleiben sollte, wird der entsprechende Communicator nicht direkt angesprochen (wie in Abbildung 7.3 vereinfacht dargestellt), sondern über den in Abbildung 7.4 gezeigten Mechanismus. `ImcImvMsgCoordinator` ruft dabei

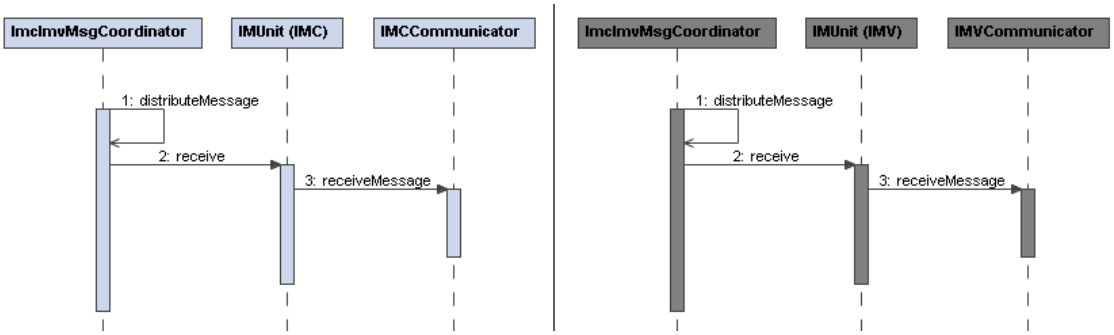


Abbildung 7.4: Gegenüberstellung: Kommunikation mit IMCs vs. Kommunikation mit IMVs

die `receive`-Methode von `IMUnit` auf, die aber erst in den Unterklassen `IMCCommunicator` und `IMVCommunicator` implementiert wird. Diese rufen dann „ihren“ Communicator auf, so dass jeweils automatisch der passende Communicator zum Einsatz kommt, ohne dass es `ImcImvMsgCoordinator` tangiert.

7.3 TNCCS-Kommunikation und Fragmentierung

IF-TNCCS definiert (wie in Kapitel 3.3.3 erläutert) ein Nachrichtenaustauschformat für die Kommunikation zwischen TNC Server und TNC Client auf Basis von XML. Sollten diese Nachrichten zu groß für die Übertragung über das Netz werden, müssen sie aufgeteilt (fragmentiert) werden. Den Aufbau und die Analyse der Nachrichten, sowie deren Fragmentierung übernehmen die Klassen aus Abbildung 7.5. Während die reine TNCCS-Kommunikation innerhalb der IEL geschieht, ist die Fragmentierung der Nachrichten Aufgabe der NAL.

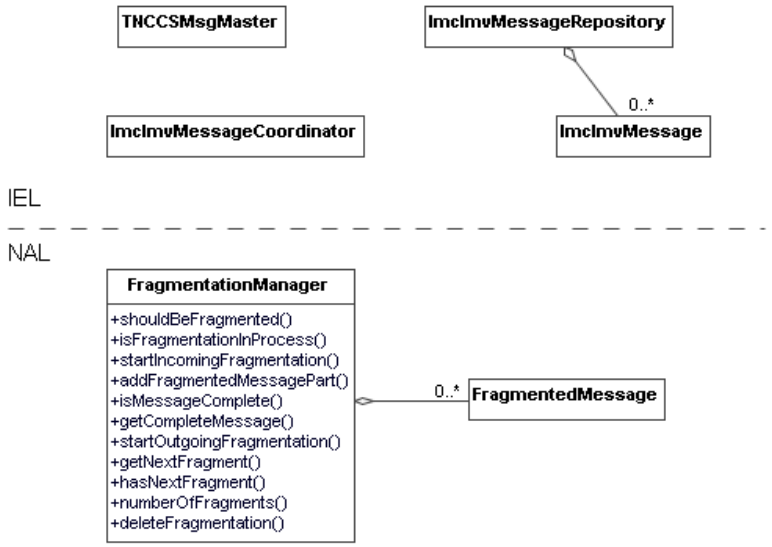


Abbildung 7.5: Klassen für die Implementierung von IF-TNCCS

7.3.1 TNCCS-Erstellung und -Analyse

`TNCCSMsgMaster` ist die zentrale Instanz in dem TNCCS-Kontext. Auf Basis von Xerces (siehe auch Kapitel 10.1) analysiert diese Klasse eingehende TNCCS-Nachrichten, extrahiert die darin enthaltenen IMC-IMV-Nachrichten und schreibt diese in `ImcImvMessageRepository`. Ebenso baut sie auch neue TNCCS-Nachrichten aus den IMC-IMV-Nachrichten zusammen, die in `ImcImvMessageRepository` gespeichert sind. Dieser Ablauf (am Beispiel TNC Client) ist in seiner zeitlichen Abfolge in Abbildung 7.6 dargestellt. Im TNC Server funktioniert dies analog.

7.3.2 Fragmentierung der Nachrichten

Die Fragmentierung der Nachrichten ist nötig, weil die Nachrichtenlänge in FreeRadius beschränkt ist.[7] Diese Aufgabe übernimmt `FragmentationManager`, der zwei Arten von fragmentierten Nachrichten unterscheiden muss:

1. Ausgehende Nachrichten, welche fragmentiert werden müssen. Diese Nachrichten kommen aus der IE-Schicht und liegen damit zu Beginn vollständig vor und müssen aufgeteilt werden. Teile von ihnen werden nach und nach über EAP-TNC verschickt, bis sie vollständig übertragen wurden.

2. Eingehende Nachrichten, welche fragmentiert werden müssen. Diese Nachrichten werden nach und nach aufgebaut. Die Fragmente kommen einzeln über

[7] FreeRadius schickt Pakete ab einer gewissen Länge nicht mehr ab. Tests ergaben als kritische Größe 4014 Bytes. Allerdings gibt es teilweise auch Probleme bei geringeren Größen, weshalb als Fragment-Größe 1275 Bytes gewählt wird.

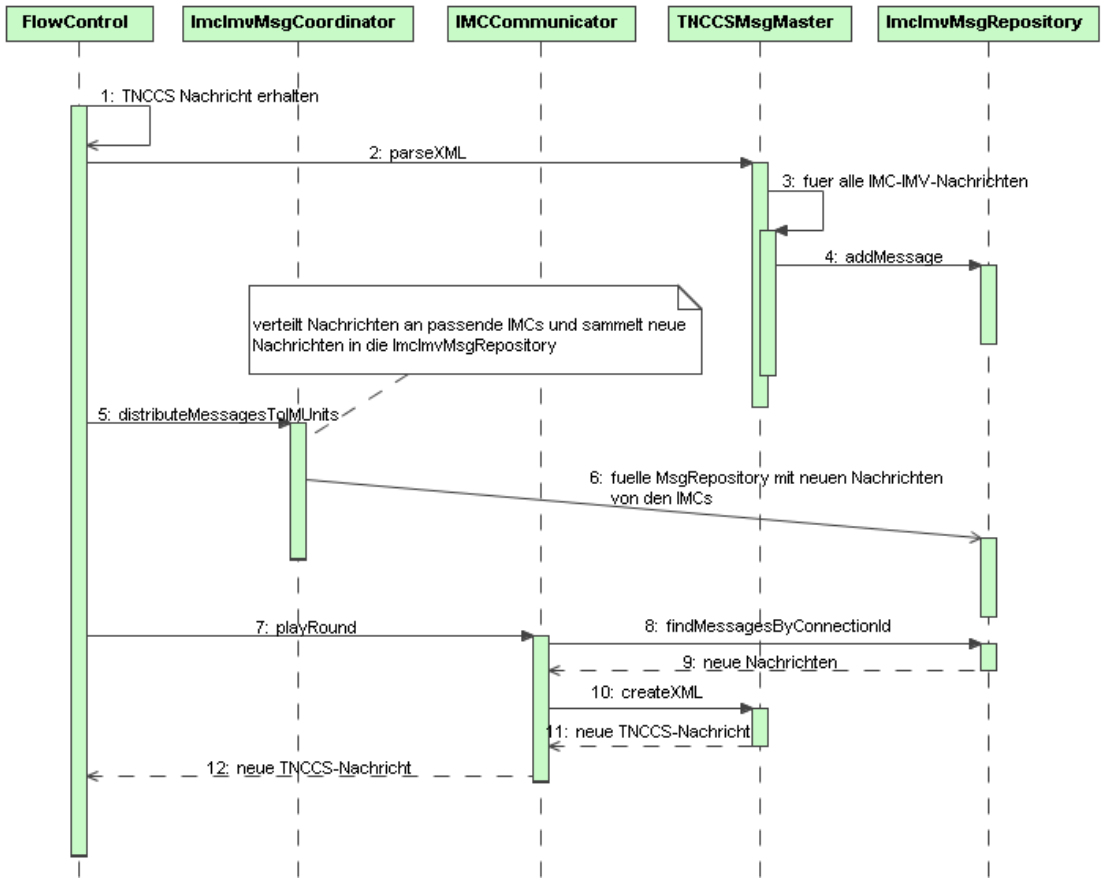


Abbildung 7.6: Ablauf mit Fokus auf die Analyse und Erstellung von TNCCS-Nachrichten

EAP-TNC im NAL an und werden, sobald sie vollständig vorliegen, in die IE-Schicht übertragen.

Um diese Nachrichten zu verarbeiten, bietet `FragmentationManager` eine umfangreiche Schnittstelle:

shouldBeFragmented gibt an, ob eine Nachricht so groß ist, dass sie fragmentiert werden muss.

isFragmentationInProcess gibt an, ob für eine bestimmte Verbindung und eine Richtung bereits ein Fragmentierungsprozess abläuft.

startIncomingFragmentation startet eine eingehende Fragmentierung für eine bestimmte Verbindung. Dabei wird die Gesamtgröße mit übergeben, damit `FragmentationManager` weiß, wann eine Nachricht vollständig ist.

addFragmentedMessagePart fügt zu einer eingehenden Fragmentierung einen eingegangenen Nachrichtenteil hinzu.

isMessageComplete zeigt an, ob alle fragmentierten Teile einer Gesamtnachricht bereits eingetroffen sind.

getCompleteMessage gibt die komplette wieder zusammengebaute eingegangene Nachricht zurück, sofern sie komplett vorliegt.

startOutgoingFragmentation startet eine ausgehende Fragmentierung. Dabei wird die Gesamtnachricht übergeben, die in Teile aufgeteilt werden soll.

getNextFragment gibt das nächste Fragment einer ausgehenden Gesamtnachricht zurück.

hasNextFragment zeigt an, ob noch ein Fragment für eine ausgehende Nachricht vorliegt, das noch nicht zurückgeliefert wurde.

numberOfFragments gibt die Zahl der Fragmente einer ausgehenden Nachricht an.

deleteFragmentation vernichtet eine fragmentierte Nachricht, wenn sie nicht mehr benötigt wird.

Der Fragmentierungs- und Defragmentierungsprozess verläuft sehr komplex, da eine Reihe von Faktoren zu berücksichtigen sind, wenn eine Nachricht über das Netz eintrifft:

- die Art der eingehenden Nachricht (vollständige TNCCS-Nachricht, TNCCS-Fragment oder Acknoledgement)
- die gesetzten Flags der eingehenden Nachricht (Start einer Fragmentierung, Teil einer Fragmentierung, Ende einer Fragmentierung oder gar keine Fragmentierung)
- die Einordnung der eingehenden Nachricht in den aktuellen Zustand einer Verbindung (Welche Nachricht wird erwartet?)

Daher wird der Ablauf nicht in einem Sequenzdiagramm (wie bisher üblich) dargestellt, sondern anhand eines Entscheidungsdiagramms losgekoppelt von der konkreten Implementierung. Das Diagramm in Abbildung 7.7 zeigt den Start des Prozesses, wenn eine EAP-TNC-Nachricht eintrifft sowie die Behandlung von eingehenden EAP-TNC-Acknoledgement-Nachrichten.

Abbildung 7.8 knüpft in (1) an das Diagramm aus 7.7 an und zeigt, wie eingehende TNCCS-Nachrichten bzw. TNCCS-Fragmente behandelt werden, bis zu dem Punkt, an dem eingehende TNCCS-Nachrichten an TNC Server bzw. TNC Client (lokal) übertragen werden (2). Wie die TNCCS-Antworten von TNC Server bzw. TNC Client behandelt und ggf. fragmentiert werden, zeigt das Diagramm in Abbildung 7.9.

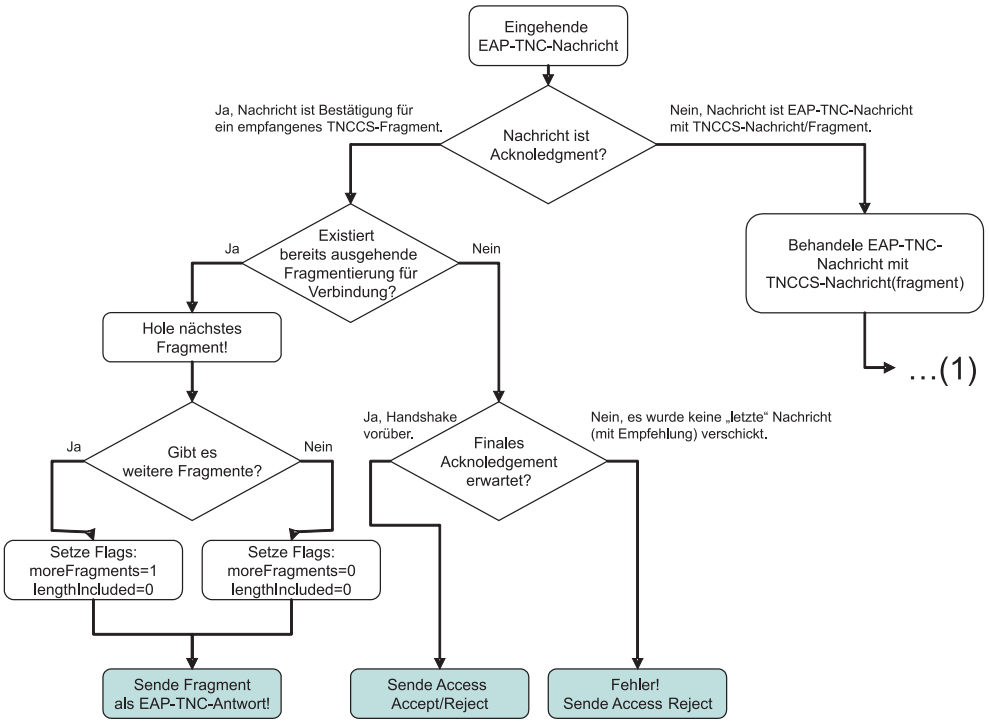


Abbildung 7.7: Start des Fragmentierungsprozesses mit der Behandlung von empfangenen Bestätigungen

Jeder Weg in diesen Diagrammen endet in einer abschließenden Aktion (grau hinterlegt), in der eine Nachricht festgelegt wird, welche über das Netz als Antwort auf die eingehende Nachricht (am Diagramm-Beginn) zurückgeschickt wird.

Die Fragmentierung läuft prinzipiell innerhalb von NAR und NAA gleich ab. Auf Seiten des AR wird sie von `NetAccessControl` gesteuert (vgl. Kapitel 5.2.7 und Abbildung 5.6 auf Seite 45). Innerhalb des PDP sorgt `Naa2Tncs` für die Steuerung des `FragmentationManager` (vgl. Kapitel 6.2.3 und Abbildung 6.5 auf Seite 62).

7.3.3 Acknoledgements

Bestätigungen auf Nachrichtenfragmente

IF-T besagt, dass auf fragmentierte Nachrichten (bei denen das moreFragments-Bit gesetzt ist) mit einem Acknoledgement-Paket zu antworten ist. IF-T spezifiziert dieses Acknoledgement-Paket als „EAP-TNC packet with no data" ([TCG06e, S.27]), ohne eine Aussage darüber zu machen, wie die Flags in diesem Paket gesetzt werden müssen. In der hier vorgestellten Implementierung wird diese Aussage so interpretiert, dass alle Flags auf „0" gesetzt werden, nur das Versions-Flag auf „1".

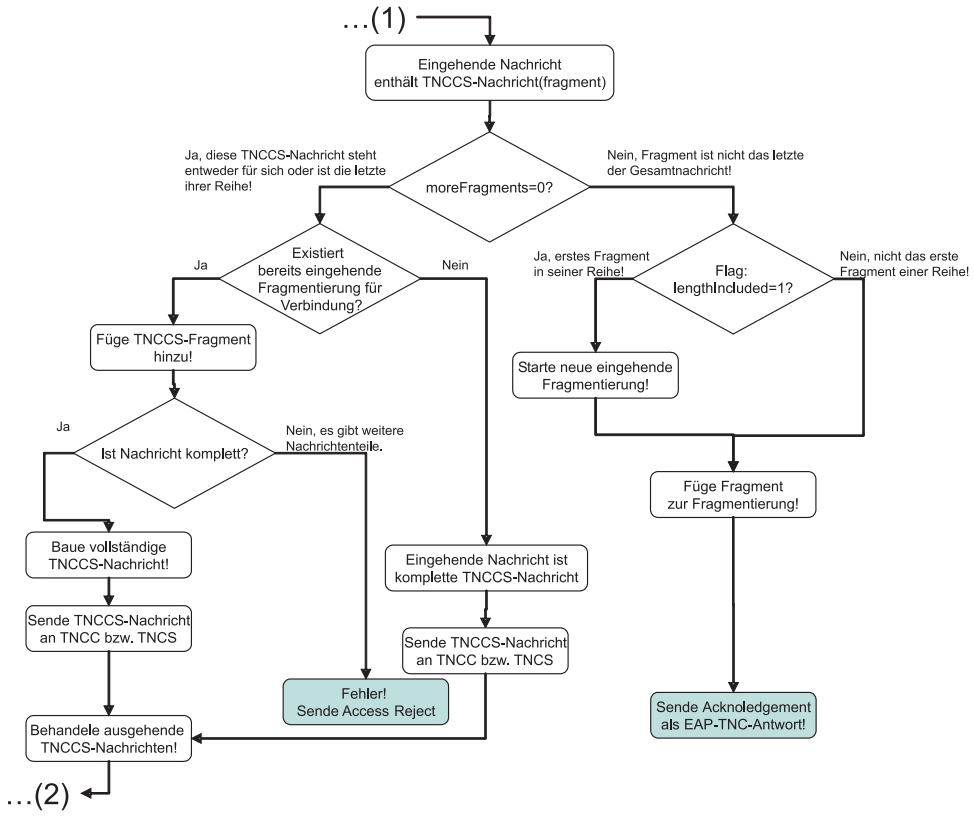


Abbildung 7.8: Behandlung von empfangenen TNCCS-Nachrichten

Bestätigung auf Handshake-Ende

In der TNC@FHH-Implementierung wird noch eine weitere Bestätigungsnachricht übertragen: Der AR schickt dem PDP diese finale Bestätigung, nachdem er die letzte TNCCS-Nachricht mit der enthaltenen Empfehlung erhalten hat. Dies hat folgende Bewandtnis:

Im TNCCS-Protokoll geht die letzte Nachricht vom TNC Server zum TNC Client und enthält die Empfehlung. In dem darunter liegenden EAP-TNC-Protokoll schickt der PDP immer den *EAP Request* an den AR, woraufhin der AR mit *EAP Response* antwortet. Demnach müsste der AR aber die letzte Nachricht verschicken. Diese Diskrepanz zwischen TNCCS und EAP-TNC wird durch die letzte (eigentlich unnütze) Bestätigung von AR zu PDP aufgehoben.

Jetzt könnte man einwenden, dass man doch einfach auf die letzte EAP Response des AR verzichten könnte. Leider ist dies aber nicht möglich, weil der PDP zwei Abschluss-Nachrichten schicken muss:

- Die erste abschließende Nachricht ist die TNCCS-Nachricht mit der Empfehlung an den TNC Client über EAP-TNC.

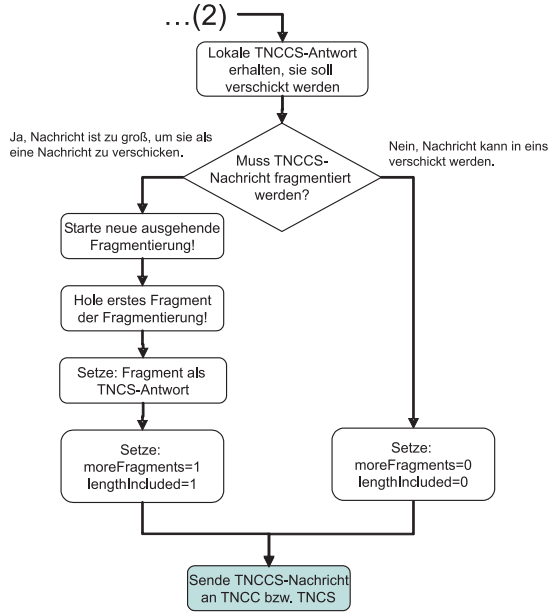


Abbildung 7.9: Behandlung von ausgehenden TNCCS-Nachrichten mit eventueller Fragmentierung

- Die zweite abschließende Nachricht ist ein *Radius Access-Accept* bzw. *Radius Access-Reject*, welches an den Switch gerichtet ist, damit dieser –wenn gefordert– den entsprechenden Port schließen kann und ein *EAP Success* bzw. *EAP Failure* an den AR weiterleiten kann.

Diese beiden Nachrichten können nicht zu einer zusammengefasst werden, weil EAP-RFC 3748 eindeutig besagt, dass *EAP Success* und *EAP Failure* keine zusätzlichen Daten enthalten dürfen (RFC 3748, S. 24)!

Die zweite Alternative wäre, dass der PDP zwei Nachrichten hintereinander wegschickt. Dies entspricht aber nicht dem RADIUS-Protokoll, da dort eindeutig festgelegt wird, dass *Radius Access-Accept* bzw. *Access-Reject* als Antwort auf *RADIUS Access-Request*-Nachrichten benutzt werden (RFC 2865, S. 17ff).

Nicht untersucht wurde, ob sich dieses Problem auflöst, wenn EAP-TNC wie in IF-T verlangt, in einem anderen EAP-Typen getunnelt wird.

7.4 Fazit zum Gesamtentwurf

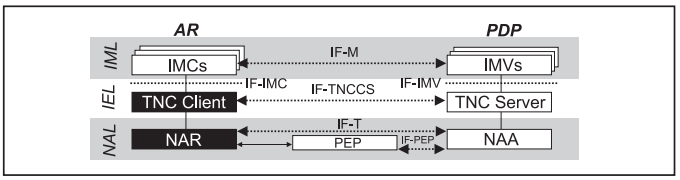
Der AR wurde als Gesamtapplikation entworfen, in der alle Fachlichkeiten, der Netzwerkzugriff inbegriffen, von Grund auf neu entwickelt werden. Die konzipierte

Benutzerschnittstelle soll für eine verbesserte Akzeptanz durch die Benutzer und für die Nachvollziebarkeit des Handshake sorgen.

Der PDP dagegen nutzt RADIUS- und EAP-Fachlichkeiten von FreeRADIUS, da entschieden wurde, das EAP-Modul von FreeRadius zu erweitern. Um die gemeinsame Nutzung von Klassen zwischen AR und PDP zu ermöglichen, sieht die Gesamtarchitektur aber vor, dass ein Großteil der PDP-Funktionalität aus dem EAP-Modul herausgezogen und extern angebunden wird. Für den TNC Server wurde ein Policy-Management und ein Konzept für die parallele Bearbeitung mehrerer ARs erarbeitet.

Die Kapselung der einzelnen Funktionalitäten in Klassen und die klare Trennung der Schichten in einer Entität, macht die Entwürfe leicht erweiter- und portierbar. Gemeinsame Fachlichkeiten wie die Anbindung der IM-Schicht und die TNCCS-Nachrichtenbehandlung mit Fragmentierung dieser Nachrichten werden Client/-Server-übergreifend entworfen, um Redundanzen zu vermeiden.

8 Implementierung des Access Requestor



In Kapitel 5 und Kapitel 7 wurden Konzepte für die Entwicklung des Acccess Requestor vorgestellt. Nun soll beschrieben werden, wie die Implementierung konkret umgesetzt wird. Dazu wird zunächst in Kapitel 8.1 gezeigt, in welcher Umgebung der AR entwickelt und wie er gebaut wird. Danach wird auf einzelne Aspekte der Implementierung eingegangen: Die Entwicklung des NDIS Protokoll Treibers ist Inhalt von Kapitel 8.2. Die Entwicklung der Benutzeroberfläche wird in Kapitel 8.3 vorgestellt.

8.1 Entwicklungsumgebung und Build-Verfahren

8.1.1 Entwicklungsumgebung

Für die Entwicklung des AR wird Eclipse 3.1.1 unter Windows XP eingesetzt[1]. Das Eclipse-Plugin CDT (C/C++ Development Tooling) SDK 3.0.2[2] ermöglicht die Entwicklung von C und C++-Applikationen aus Eclipse heraus.

8.1.2 Build des AR

Gebaut wird der AR als ausführbare Applikation (*.exe) für die Cygwin-Umgebung, die eine Linux-ähnliche Umgebung für Windows bildet.[3] Damit wird auch unter Windows ein emulierter gcc[4] zur Kompilierung genutzt.

[1] Eclipse ist eine freie Entwicklungsumgebung, die vor allem im Java-Umfeld stark eingesetzt wird: http://www.eclipse.org

[2] http://www.eclipse.org/cdt/, siehe auch: [IBM04]

[3] Cygwin emuliert über die DLL `cygwin1.dll` eine API für die wichtigsten Linux-Funktionen, so dass Linux unter Windows emuliert wird: http://www.cygwin.com/, siehe auch: [Red03]

[4] gcc ist der Standard-C/C++-Compiler unter Linux.

Der AR wird in Eclipse als „Managed Make C++ Project“ angelegt und kann dann mit den Mitteln des CDT aus Eclipse heraus vollständig gebaut werden.

Die Einstellungen, die dafür im Eclipse-Projekt zu tätigen sind, sind im Anhang in Kapitel B.1 aufgelistet. Besonders die Nutzung von wxWidgets für die Benutzeroberfläche (siehe auch Kapitel 8.3) erfordert eine Reihe von Anpassungen und Einstellungen.

8.2 Entwicklung des Netzwerktreibers NDISProt

8.2.1 Anpassung und Kompilierung von NDISProt

Für den Zugriff auf das Netzwerk wird der NDIS Protocol Driver „NDISProt“ benötigt (siehe Kapitel 5.2.3). Dieser ist im Windows Driver Development Kit eines der Beispiele in `samples`. Dort sind der Treiber und eine Applikation für den Test des Treibers enthalten. Beides muss vor der Nutzung kompiliert werden.

Anpassung von NDISProt

Für den Einsatz in der TNC-Architektur ist allerdings zunächst noch eine Anpassung nötig. Der Treiber kann standardmäßig direkt adressierte Pakete, Multicast- und Broadcast-Pakete empfangen. Da in 802.1x aber eine spezielle MAC-Adresse zur Adressierung genutzt wird, die aus einem reservierten Satz von MAC-Adressen stammt (siehe [IEE04, S. 31])[5], fällt sie in keinen dieser Bereiche.

In der Datei `ndisprot.h` wird konfiguriert, welche Paket-Typen empfangen werden sollen. Recherchen[6] und Tests ergaben, dass die in 802.1x genutzte MAC-Adresse in den Bereich „All Multicast“ fällt. Damit muss der Paket-Filter des NDISProt-Treibers um diesen Eintrag erweitert werden, damit der Empfang der EAPOL-Pakete funktioniert (siehe Listing 8.1).

```
#define NUIOO_PACKET_FILTER (NDIS_PACKET_TYPE_DIRECTED| \
                             NDIS_PACKET_TYPE_MULTICAST| \
                             NDIS_PACKET_TYPE_BROADCAST | \
                             NDIS_PACKET_TYPE_ALL_MULTICAST)
```

Listing 8.1: Anpassungen in ndisprot.h

Kompilierung von NDISProt

Über das im DDK mitgelieferte „Build Environment“ kann NDISProt nun, wie in der mitgelieferten Anleitung beschrieben, kompiliert werden (siehe Abbildung 8.1).

[5] Genutzt wird 01-80-C2-00-00-03, Ethereal identifiziert diese Adresse als „Spanning-tree-(for-bridges)_03“.

[6] Im Windows-DDK werden in `/inc/wnet/ntddndis.h` alle möglichen Paket-Filter-Bits definiert.

Danach liegt die für die Installation nötige Setup-Informationsdatei `ndisprot.inf`

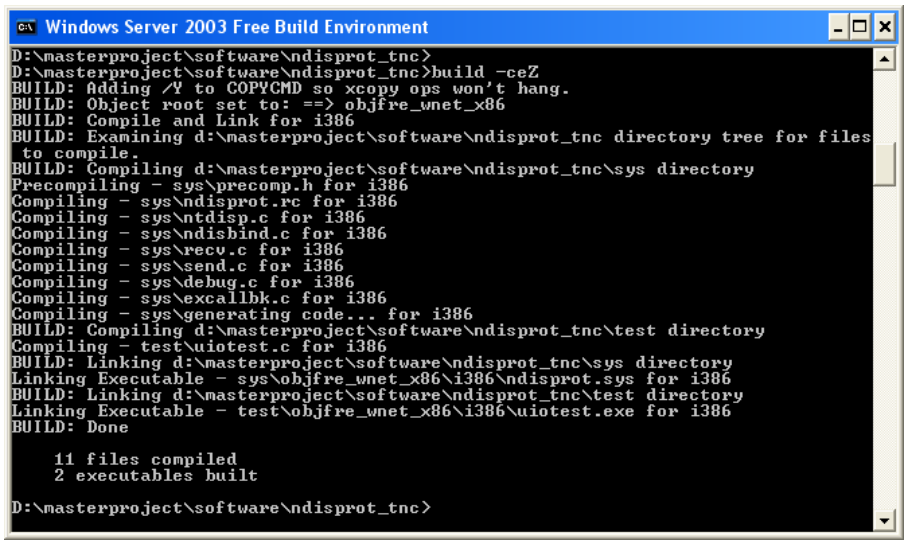


Abbildung 8.1: Build von NDISProt im Windows Server 2003 Free Build Environment

und der eigentliche Treiber `ndisprot.sys` vor.

8.2.2 Installation des NDISProt-Treibers

Der neu generierte Netzwerk-Treiber kann nun als Protokoll für die entsprechenden Netzwerkkarten installiert werden. Dies geschieht ausgehend vom Dialog „Eigenschaften der LAN-Verbindung“. Wie die Installation Schritt für Schritt abläuft, wird anhand Screenshots im Anhang in Kapitel B.3 detailliert erläutert.

Die Setup-Informationsdatei `ndisprot.inf` legt fest, dass der Treiber manuell gestartet wird. Dies geschieht über den Befehl:

```
net start ndisprot
```

Nun können programmativ Ethernetpakete per `ReadFile`-Methode empfangen und per `WriteFile`-Methode aus C/C++-Programmen verschickt werden.[7]

8.3 Entwicklung der Benutzeroberfläche

Die Benutzeroberfläche des TNC Clients (siehe Kapitel 5.4.1) wird mit wxWidgets entwickelt. wxWidgets [8] ist Open Source und ermöglicht es, Applikationen objektorientiert u.a. unter C++ für verschiedene Plattformen (z.B. Windows, Linux oder

[7] Eine Einführung in die Windows-Programmierung gibt: [Pet99]
[8] http://www.wxwidgets.org/

Mac) zu entwickeln. Die Applikation basiert aber trotzdem für alle Plattformen auf einer einzigen Code-Basis und entspricht dem jeweiligen „Look and Feel“ der Plattform. Diese Eigenschaften versprechen später eine leichte Portierung des TNC Clients nach Linux. Im Rahmen dieses Projekts wird wxWidgets in der Version 2.6.3 eingesetzt. Dabei wird das Augenmerk vor allem auf die grafischen Fähigkeiten von wxWidgets gelegt.

8.3.1 Einbindung von wxWidgets in die Entwicklung des TNCC

Vor der Nutzung muss wxWidgets installiert und gebaut werden. Dies wird im Anhang in Kapitel B.4 erläutert.

Dann kann wxWidgets zur Entwicklung der Oberfläche des TNC Clients genutzt werden. Welche Einstellungen dazu im Eclipse-Projekt zu tätigen sind, beschreibt [Don04]. Die in diesem Projekt benötigten Einstellungen finden sich in der Konfigurationsbeschreibung für das Eclipse-Projekt in Kapitel 8.1.2 und B.1 wieder.

8.3.2 Entwicklung der Oberfläche des TNCC

Die Oberfläche des TNC Client besteht aus einem einzigen Fenster, in dem ein Dialog dargestellt wird. Deshalb wird eine Klasse `TnccOptionsDialog` erzeugt, welche von `wxDialog` erbt. In diese Klasse werden die Darstellungs- und Steuerungselemente eingefügt, so dass die Oberfläche aussieht, wie in Abbildung 5.9 auf Seite 50 dargestellt. Die Ereignisbehandlung wird ebenfalls dort implementiert. Die Klasse `TnccApp` erbt von `wxApp` und bildet im wxWidgets-Framework mit seiner Methode `onInit` nun den Startpunkt der TNCC-Applikation. Dies wird über das Makro `IMPLEMENT_APP` festgelegt. Eine klassische `main`-Methode ist nicht mehr nötig. In Listing 8.2 ist dargestellt, wie die grobe Struktur der TNCC-Oberfläche mit wxWidgets definiert ist.

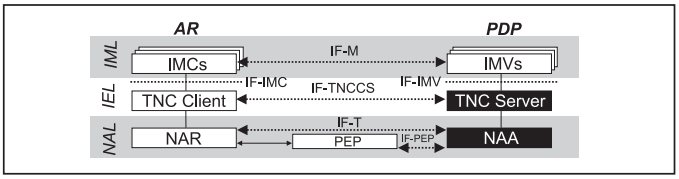
```
class TnccOptionsDialog: public wxDialog{
  ...
}

class TnccApp: public wxApp{
  virtual bool onInit();
  ...
}
IMPLEMENT_APP(TnccApp)
```

Listing 8.2: wxWidget-Anpassungen

Es soll hier nur die grundlegende Idee der GUI-Implementierung präsentiert werden. Für Details der GUI-Implementierung wird auf den Quellcode verwiesen. Die Entwicklung der Dialogapplikation orientiert sich an [Ghi99].

9 Implementierung des Policy Decision Point



In Kapitel 6 und 7 wurden die zentralen Konzepte des PDP aufgezeigt. Dieses Kapitel baut auf diesen Konzepten auf, indem dargelegt wird, wie sie konkret umgesetzt werden. Dafür wird zunächst in Kapitel 9.1 auf die eingesetzte Entwicklungsumgebung, sowie das Entwicklungs- und Build-Verfahren eingegangen, bevor in Kapitel 9.2 die Entwicklung des FreeRadius-Moduls im Mittelpunkt steht. Abschließend werden in Kapitel 9.3 die Konfigurationsmöglichkeiten des PDP beleuchtet.

9.1 Entwicklungsumgebung und Build-Verfahren

9.1.1 Entwicklungsumgebung

Für die Entwicklung des PDP wird als Umgebung Eclipse 3.1.0 unter Linux eingesetzt. Wie für die Entwicklung des AR unter Windows (vgl. Kapitel 8.1.1) wird auch hier das Eclipse-Plugin CDT SDK 2.1.1 benutzt. Das Plugin nutzt den unter Linux üblichen C/C++-Compiler gcc.

9.1.2 Entwicklungsunterstützung beim Bau des TNCS-SO

Die Entwicklung und der Bau des Shared Objects für den TNCS erfolgt mit der vollen Unterstützung des Eclipse-Plugins CDT. Der TNCS wird in Eclipse als „Managed Make C++ Project“ angelegt und kann dann mit den Mitteln des CDT aus Eclipse heraus vollständig gebaut werden. Die Einstellungen, die dafür im Eclipse-Projekt zu tätigen sind, sind im Anhang in Kapitel B.2 aufgelistet.

Damit steht am Ende des Erstellungsprozesses von TNCS ein Shared Object `libTNCS.so`, das im Gesamtbau des PDP genutzt wird.

9.1.3 Unterstützung bei der Entwicklung des EAP-TNC-Moduls

Da die gesamte FreeRadius-Applikation in einem komplexen Build-Verfahren gebaut wird, ist es nicht einfach möglich, dieses Verfahren innerhalb von Eclipse zu integrieren. Daher wird eine Kopie des gesamten Source-Verzeichnisses von FreeRadius in ein Eclipse-C-Projekt `EAP-TNC-Modul` integriert. So kann Eclipse als erweiterter Text-Editor genutzt werden, ohne die Build-Funktionen zu nutzen.

Das hat den Vorteil, dass der gesamte Code in einer Umgebung vorliegt und Suchfunktionen, Highlighting u.ä. von Eclipse genutzt werden können.

9.1.4 Bau des PDP

Wie bereits erläutert, kann das EAP-TNC-Modul nicht innerhalb von Eclipse gebaut werden. Daher wird der Gesamt-Build in mehreren Schritten durchgeführt:

1. Die TNCS-SO wird, wie im vorherigen Abschnitt erläutert, gebaut.
2. Die TNCS-SO wird aus dem TNCS-Eclipse-Projekt an ihre Zielposition kopiert.
3. Alle potenziell veränderten *.c- und *.h-Dateien (siehe folgendes Kapitel 9.2) werden aus dem `EAP-TNC-Modul`-Eclipse-Projekt in die ausgepackte FreeRadius-Distribution an die gleichen Stellen kopiert. Dies sind die Dateien aus den Verzeichnissen:
   - `modules/rlm_eap/types/rlm_eap_tnc`
   - `modules/rlm_eap/libeap`
4. das EAP-TNC-Modul, das EAP-Modul oder die gesamte FreeRadius-Distribution wird kompiliert. Dies kann danach variiert werden, wo Änderungen vorgenommen wurden.[1]
5. Die erzeugten Dateien werden an die entsprechenden Stellen kopiert.

Die Schritte 2 und 3 werden in einem Skript `copyAll` zusammengefasst, die Schritte 4 und 5 laufen nach dem FreeRadius-Build-Verfahren ab. Der Ablauf ist damit soweit automatisiert, dass eine Befehlszeile den gesamten Build durchführt (siehe Listing: 9.1). Diese Befehlszeile wird innerhalb des Verzeichnisses des zu bauenden FreeRadius-Teils (normal: `rlm_eap_tnc`) ausgeführt.

```
/home/tncuser/workspace/copyAll && make && make install
```

Listing 9.1: PEP-Eintrag in clients.conf

[1] Während der normalen Entwicklung reicht es aus, nur das EAP-TNC-Modul neu zu kompilieren.

9.2 Entwicklung des EAP-TNC-Moduls von FreeRadius

9.2.1 Konfiguration von FreeRadius

FreeRadius kann mit einer Ausnahme in seiner Standard-Konfiguration genutzt werden.

In `/usr/local/etc/raddb/clients.conf` muss der PEP als Client eingetragen werden. Im Rahmen der später vorgestellten Testumgebung (siehe Kapitel 11) sind die Einstellungen wie in Listing 9.2 dargestellt zu machen. Entscheidend sind aber nur die angegebene IP-Adresse des Switches und das Passwort, das genauso auch im Switch eingetragen wird.

```
client 192.168.1.1 {
  secret   = mypassword
  shortname = TNC-PEP
  nastype  = other
}
```

Listing 9.2: PEP-Eintrag in clients.conf

9.2.2 Erweiterung des EAP-Moduls

Änderungen in Konfigurationsdateien des EAP-Moduls

Folgende Änderungen an den Konfigurationsdateien des EAP-Moduls müssen vorgenommen werden (vgl. [Rag05]): In `/usr/local/etc/raddb/eap.conf` wird unter „Supported EAP-types“ ein neuer Eintrag für EAP-TNC angelegt und im Default-EAP-Typ auf diesen Eintrag verwiesen (siehe Listing 9.3). Da aus der ersten eintreffenden Nachricht (*EAP-Identity-Response*) nicht ersichtlich ist, um welchen EAP-Typen es sich handelt, muss dieser vorab definiert werden. In FreeRadius wird nur ein EAP-Typ zur gleichen Zeit unterstützt. Dieser wird in `default_eap_type` gespeichert.

```
eap{
  default_eap_type = tnc
  ...
  #Supported EAP-types
  tnc{
  }
  ...
}
```

Listing 9.3: Einstellungen in eap.conf

Änderungen am generellen Code des EAP-Moduls

Das Framework des EAP-Moduls muss die unterstützten EAP-Typen und die Abbildung von EAP-Type (Zahlenwert) auf das richtige Modul kennen. Dazu muss in `<FREERADIUS>/src/modules/rlm_eap/libeap/eapcommon.c` im Array `eap_types` an der 38. Stelle „tnc“ eingetragen werden, weil EAP-Type=38 TNC zugeordnet ist (siehe Listing 9.4).

```
static const char *eap_types[]={
...
"37",
"tnc",
"39"
};
```

Listing 9.4: EAP-TNC in eapcommon.c

Zudem muss eine ähnliche Ergänzing in `eap_types.h` vergenommen werden (siehe Listing 9.5).

```
#define PW_EAP_MAX_TYPES 39
#define PW_EAP_TNC 38
```

Listing 9.5: EAP-TNC in eap_types.h

Es wird hier die Zahl der EAP-Typen auf eine Zahl größer als 38 erhöht und der EAP-TNC-Type als 38 definiert.

Fazit

Mit diesen Anpassungen leitet das EAP-Modul von FreeRadius nun alle eingehenden EAP-Nachrichten in das EAP-Untermodul `EAP-TNC`, welches im nächsten Abschnitt erstellt wird.

9.2.3 Erzeugung des EAP-Teilmoduls EAP-TNC

Das neue EAP-TNC-Modul `rlm_eap_tnc` kann am einfachsten angelegt werden[2], indem ein bestehendes Modul, wie `rlm_eap_md5`, komplett kopiert wird und in allen Dateien „md5“ durch „tnc“ ersetzt wird.

Entscheidend dafür, dass das FreeRadius-Framework das Modul nutzen kann, ist nur, dass die Datei `rlm_eap_tnc.c` existiert und darin definiert wird, wie der Zugriff auf das Teilmodul erfolgt. Dies geschieht über die `struct rlm_eap_tnc`, in der die Funktionen für den initalen Aufruf eines NAR (`tnc_initiate`) und für jeden weiteren Aufruf während der Authentisierung (`tnc_authenticate`) definiert werden. Diese Funktionen müssen in dieser Datei implementiert werden (siehe Listing 9.6).

[2] Angelegt wird es in `<FREERADIUS>/src/modules/rlm_eap/types`.

```
static int tnc_initiate(void *type_data, EAP_HANDLER *handler){
...
}

static int tnc_authenticate(void *arg, EAP_HANDLER *handler){
  ...
}

EAP_TYPE rlm_eap_tnc= {
  "eap_tnc",
  NULL,            /* attach */
  tnc_initiate,    /* Start the initial request */
  NULL,            /* authorization */
  tnc_authenticate, /* authentication */
  NULL             /* attach */
}
```

Listing 9.6: rlm_eap_tnc.c

Diese zwei Methoden sind alles, was für ein funktionierendes EAP-Teilmodul gebraucht wird. Voraussetzung ist natürlich, dass die Methoden so implementiert sind, dass sie den `EAP_HANDLER` (vgl. auch Kapitel 9.2.5), der als Übergabeparameter in die Methoden gegeben wird, mit einem korrekten *EAP Request*-Paket (als Ergebnis dieser Methoden) füllen.

VLAN-Zuordnung

In FreeRadius kann die VLAN-Zuordnung benutzerorientiert vorkonfiguriert werden. Dies geschieht über die Benutzereinträge in der Konfigurationsdatei `/usr/local/etc/raddb/users` und sieht dann wie in Listing 9.7 abgebildet aus.

```
Lumpy Auth-Type := EAP, User-Password == "myPassword"
  Tunnel-Type = "VLAN",
  Tunnel-Medium-Type = "IEEE-802",
  Tunnel-Private-Group-ID = 96
```

Listing 9.7: Konfigurierte VLAN-Zuordnung in raddb/users

Der Switch-Port von Benutzer „Lumpy“ wird demnach nach dessen erfolgreicher Authentisierung in das VLAN mit der VLAN-ID 96 geschaltet.

Für die TNC-Belange ist diese Konfigurationsmöglichkeit aber nicht ausreichend, da Rechner abhängig vom Ausgang der Integritätsprüfung in unterschiedliche VLANs einsortiert werden. Deshalb wird völlig auf Benutzereinträge verzichtet[3] und stattdessen die VLAN-Zuordnung programmatisch gesetzt. Listing 9.8 zeigt, wie innerhalb von EAP-Modulen die benötigten RADIUS-Attribute (nach IF-PEP) erzeugt

[3] Eine Authentisierung der einzelnen Benutzer findet nicht statt.

und in den `EAP_HANDLER` gesetzt werden.[4]

```
VALUE_PAIR *tunnelType = pairmake("Tunnel-Type", "VLAN", T_OP_SET);
pairadd(&handler->request->reply->vps, tunnelType);

VALUE_PAIR *tunnelMedium = pairmake("Tunnel-Medium-Type", "IEEE-802", T_OP_SET);
pairadd(&handler->request->reply->vps, tunnelMedium);

char *vlanNumber="96";
VALUE_PAIR *vlanId = pairmake("Tunnel-Private-Group-ID", vlanNumber, T_OP_SET);
pairadd(&handler->request->reply->vps, vlanId);
```

Listing 9.8: Programmative VLAN-Zuordnung

9.2.4 Fazit

Mit den erläuterten Anpassungen in FreeRadius und dem EAP-Modul und dem neu erstellten EAP-Teilmodul kann nun die gesamte RADIUS-Funktionalität von FreeRadius und die EAP-Funktionalität des EAP-Moduls für die NAA im PDP genutzt werden. Diese Funktionalitäten werden um EAP-TNC und die Anbindung des Shared Object des `NAA-TNCS` (wie im Konzept-Teil beschrieben: vgl. Kapitel 6.2.2 und 6.2.3) erweitert.

9.2.5 Verbindungsmanagement

Innerhalb der TNC-Architektur werden auf dem TNC Server Anfragen unterschiedlicher AR durch eine sogenannte „ConnecionID“ unterschieden.[5] Über den gesamten Handshake mit einem AR hat die ConnectionID Bestand. Dies ist besonders wichtig in der Kommunikation mit den IMV-Komponenten, welche auf die IMC-IMV-Nachrichten von unterschiedlichen AR auch unterschiedlich reagieren müssen. Daraus ergibt sich die Anforderung, dass sie Anfragen unterschiedlicher AR differenzieren können.

Das Problem besteht nun darin, die einzeln eintreffenden EAP-Nachrichten der entsprechenden Verbindung mit einer ConnectionID zuzuordnen. Da die ConnectionID ein lokales Konstrukt ist, wird sie nicht übermittelt. Das EAP-Modul von FreeRadius hält für jede Client-Verbindung eine `Struct` vom Typ `EAP_HANDLER`, welche Client-spezifizische Informationen enthält. Die Zuordnung der eintreffenden Nachricht zu einem `EAP_HANDLER` und damit zu einer Verbindung erfolgt über die EAP-ID: Das EAP-Modul von FreeRadius sendet ein *EAP Request* mit einer eindeutigen EAP-ID an den AR, dessen Antwort dieselbe EAP-ID enthält. Damit kann

[4] Das Attribut „Tunnel-Medium“, das in IF-PEP [TCG06d, S. 16] genannt wird, funktioniert im Zusammenspiel mit dem Procurve-Switch nicht. Es muss „Tunnel-Medium-Type“ lauten und als Wert nicht „802“ beinhalten, sondern wie angegeben „IEEE-802“.

[5] Auf dem AR dient dagegen die ConnectionID zur Unterscheidung von Anfragen an verschiedene PDPs.

für jeden `EAP_HANDLER` die gerade gültige EAP-ID gespeichert werden und anhand dieser der richtige `EAP_HANDLER` für ein eintreffendes *EAP-Response* gefunden werden.

Kann man diesen Umstand für die Identifizierung der ConnectionID nutzen? Leider ist es innerhalb der Methoden `tnc_initiate` oder `tnc_authenticate` nicht möglich, herauszufinden, wie die folgende EAP-ID für den neuen *EAP Request* aussieht, weil diese erst nach Verlassen dieser Methoden vom EAP-Framework festgelegt wird. Man könnte spekulieren, dass sie um eins höher ist als die bisherige bekannte, aber dieser Ansatz ist gefährlich, da innerhalb des EAP-RFC (vgl. RFC 3748, S. 22) nicht festgelegt ist, dass die IDs von Anfrage zu Anfrage iteriert werden. Dies ist zwar üblich, aber es ist nicht klar, wie das EAP-Modul von FreeRadius mit der Vergabe der EAP-IDs verfährt, wenn mehrere parallele Anfragen unterschiedlicher Supplicants parallel eintreffen. Damit fallen die EAP-IDs zur Identifizierung der richtigen ConnectionID aus.

Im `EAP_HANDLER` findet sich kein Attribut, welches über die gesamte Verbindungsdauer konstant und eindeutig ist und damit eine Identifizierung zulassen würde. Aber das Attribut „opaque“ ist für EAP-typ-spezifische Daten vorgesehen und kann frei gefüllt werden. Deshalb wird in der vorgestellten NAA-Implementierung in „opaque“ die eindeutige ConnectionID (ein einfacher Zähler) innerhalb der Methode `tnc_initiate` eingetragen und innerhalb von `tnc_authenticate` ausgelesen, so dass alle Anfragen einer ConnectionID zugeordnet werden können.

9.3 Konfigurationsdatei tncs_fhh.conf

Einige Einstellungen des PDP müssen leicht veränderbar sein und dürfen nicht im Quellcode fest verdrahtet werden. In der Konfigurationsdatei `tncs_fhh.conf` werden diese Einstellungen abgelegt und bei Start des PDP ausgelesen. Die Datei hat den Aufbau einer normalen Properties-Datei[6]. Der Aufbau ist im Listing B.2 im Anhang abgebildet.

Die Konfigurationsdatei bietet folgende Einstellungsmöglichkeiten:

VLAN_ACCESS gibt die VLAN-ID des VLANs an, zu dem ein AR zugeordnet wird, der den Integritätstest bestanden hat.

VLAN_ISOLATE gibt die VLAN-ID des VLANs an, zu dem ein AR zugeordnet wird, der isolierten Zugang erhält.

BATCH_COUNT gibt die Zahl der Batches an, die ein TNCS maximal durchführt, bevor er den Handshake beendet (vgl. Kapitel 6.3.3).

[6]Eine Properties-Datei hat mehrere Einträge, die jeweils eine Zeile bilden und aus Name und Wert bestehen, die durch ein Gleichheitszeichen getrennt sind.

POLICY gibt die Nummer der aktiven Sicherheitsrichtlinie an. Die Richtlinien aus Kapitel 6.3.3 sind durchnummeriert.

TNCS_PATH gibt den Pfad zu dem TNCS-Shared Object an, das das EAP-TNC-Modul lädt.

Nachdem die nutzbaren IMVs bereits in `tnc_config` eingetragen werden (nach IF-IMV, Linux Binding), komplettieren die hier vorgestellten Optionen die Konfigurationsmöglichkeiten des im Rahmen dieses Projektes entwickelten PDP. Damit ist die Konfiguration des PDP, vor allem für Testzwecke, stark vereinfacht und ohne programmative Eingriffe möglich.

10 Hilfsklassen für die Implementierung

Für die Entwicklung von AR und PDP werden einige Hilfsklassen eingesetzt, die an dieser Stelle kurz vorgestellt werden.

10.1 Entwicklungen im Projekt TNC@FHH

Im Rahmen des Projekts TNC@FHH hat Daniel Wuttke eine Reihe von Hilfsklassen und -module entwickelt, die auch in diesem Projekt Verwendung finden. Sie werden als DLL `libTNCUtil.a` unter Windows und unter Linux als Shared Object `libTNCUtilLinux.a` eingebunden.

Da Daniel Wuttke diese Klassen oder Module in seiner Arbeit (siehe [Wut06]) näher vorstellt, wird an dieser Stelle nur ein kurzer Überblick gegeben.

TNCLog bietet einen Logging-Mechanismus, mit dem für jede Klasse ein eigener Log-Level festgelegt werden kann. `TNCLog` wird in nahezu allen Klassen im PDP und AR eingesetzt.

TNCIOHelper bietet eine API zum einfachen Schreiben in Dateien und Lesen aus Dateien. `TNCIOHelper` wird im AR von `UserLog` zum Schreiben der detaillierten Log-Datei genutzt. Auf Seiten des TNCS liest `LinuxConfigReader` die Datei `tnc_config` mit Hilfe des `TNCIOHelper` aus. `Recommender` nutzt TNCIOHelper, um die gültige Sicherheitslinie aus `tncs_fhh.config` zu extrahieren.

TNCXMLHelper bietet eine API für die einfache Behandlung von XML-Dokumenten. Dabei kapselt `TNCXMLHelper` Aufrufe an Xerces[1]. `TNCCSMsgMaster` nutzt `TNCXMLHelper`, um TNCCS-Nachrichten auszulesen und zu erzeugen.

[1] Xerces ist ein XML-Parser, der XML-Dateien erstellt und ausliest: http://xml.apache.org/xerces-c/

10.2 Base64-Behandlung von CyoTec

Innerhalb von IF-TNCCS ist definiert, dass IMC-IMV-Nachrichten Base64-kodiert zwischen TNCC und TNCS übertragen werden. Für die Wandlung von Klartext-Nachrichten in Base64-kodierte Nachrichten und umgekehrt, nutzt `TNCSMsgMaster` zwei Klassen von cyotec[2]:

CyoEncode wandelt Klartext-Nachrichten von IMCs/IMVs in Base64-kodierte Nachrichten für den Versand innerhalb von TNCCS um.

CyoDecode wandelt Base64-kodierte IMC-IMV-Nachrichten, entnommen aus einer TNCCS-Nachricht, in Klartext-Nachrichten um, bevor sie an IMCs/IMVs ausgeliefert werden.

[2] http://www.cyotec.com/

11 Testumgebung zur Überprüfung der Implementierung

Um die Funktionsfähigkeit der entwickelten Software zu überprüfen, wird eine Testumgebung eingerichtet. Die Struktur und Konfiguration dieser Umgebung wird in Kapitel 11.1 beschrieben. Daran anschließend wird gezeigt, wie die entwickelten Entitäten innerhalb dieser Testumgebung interagieren, so dass die Tragfähigkeit der Entwicklung gezeigt wird. Dies geschieht in Kapitel 11.2.

11.1 Struktur und Konfiguration

11.1.1 Überblick

Die Testumgebung besteht aus zwei handelsüblichen PCs und einem Switch vom Typ HP Procurve 5368xl. Diese drei Komponenten sind wie in Abbildung 11.1 dargestellt miteinander verbunden. Sie bilden ein abgeschlossenes Netz mit der

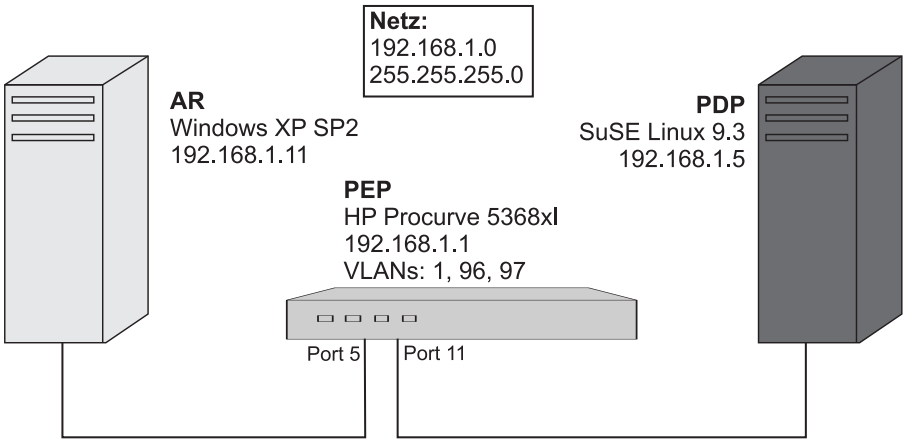


Abbildung 11.1: Struktur der Testumgebung des Projekts TNC@FHH

Netzwerkadresse 192.168.1.0. Ein Rechner, auf dem Windows XP SP2 installiert ist, bildet die Entität des AR. Der zweite Rechner agiert als PDP und hat SuSE Linux 9.3 als Betriebssystem installiert.

11.1.2 Konfiguration des Switch für TNC

Damit der Switch innerhalb dieser Testumgebung eingesetzt werden kann, müssen mehrere Dinge konfiguriert werden:

- Initiale Einstellungen des Switch
- Generelle 802.1x-Aktivierung und Einstellung des RADIUS-Servers
- 802.1x-Authentisierung für den Port A11, an dem der AR angeschlossen ist
- VLANs für das authentisierte Netz (VLAN-ID: 96) und das Isolationsnetz (VLAN-ID: 97)

Die folgenden Konfigurationsschritte gelten für den eingesetzten HP Procurve 5348xl (vgl. [HP 05]).

Inititale Konfiguration des Switch

Über ein serielles Kabel kann z.B. das HyperTerminal von Windows[1] mit dem Switch verbunden werden. Nach der Eingabe von `setup` kann man folgendes konfigurieren:

Manager-Password	tnclab
IP-Configuration	manual
IP-Address	192.168.1.1
Subnet-Mask	255.255.255.0

Generelle 802.1x-Aktivierung

Mit den folgenden Befehlen wird 802.1x auf dem Switch aktiviert:

`config`	wechselt in den Konfigurationsmodus
`aaa authentication port-access eap-radius`	definiert, dass für die Authentisierung RADIUS genutzt werden soll
`show auth`	zeigt die bisher gemachten Einstellungen (siehe Abbildung 11.2)
`radius host 192.168.1.5`	definiert die IP-Adresse des RADIUS Server (unser PDP)
`radius-server key mypassword`	definiert, dass „mypassword“ als Schlüssel beim Zugriff auf den RADIUS-Server genutzt wird
`aaa port-access authenticator active`	schaltet die 802.1x-Authentisierung aktiv

[1] Einstellungen: Bits: 115200, DataBits: 8, Parity: none, StopBit: 1

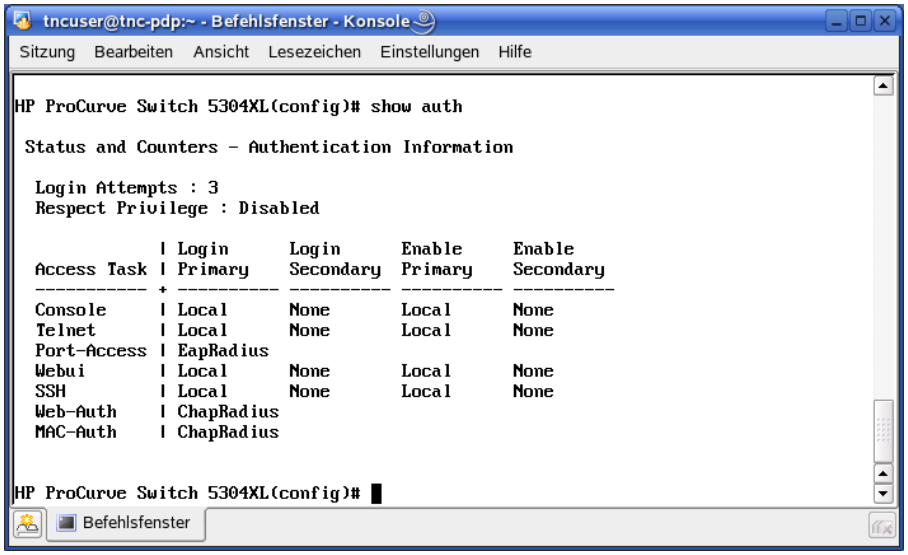


Abbildung 11.2: Die Authentisierungskonfiguration des Procurve-Switchs

802.1x-Authentisierung für einzelne Ports

`aaa port-access authenticator A11`	schaltet für Port A11 die 802.1x-Authentisierung an

Einrichtung der VLANs

VLANs lassen sich am einfachsten über die Webkonsole einrichten:

Der Link `http://192.168.1.1` (von einem ungeschützten Port aus) führt den Browser in ein übersichtliches Java-Applet.[2]

Über „Configuration“ und „VLAN Configuration“ können die bestehenden VLANs angesehen, sowie neue hinzugefügt werden.

Diese VLANs werden konfiguriert:

VLAN ID	VLAN Name	Erklärung
96	AuthZone	bildet das VLAN für die erfolgreich getesteten AR
97	IsoZone	bildet das VLAN für die in Isolation gesetzten AR

Abbildung 11.3 zeigt die fertige VLAN-Konfiguration, wie sie für die Testumgebung geeignet ist.

[2]Beim ersten Aufruf der Webkonsole sollte über das „First time installation“-Log Manager-Passwort, sowie Operatorname und -passwort gesetzt werden.

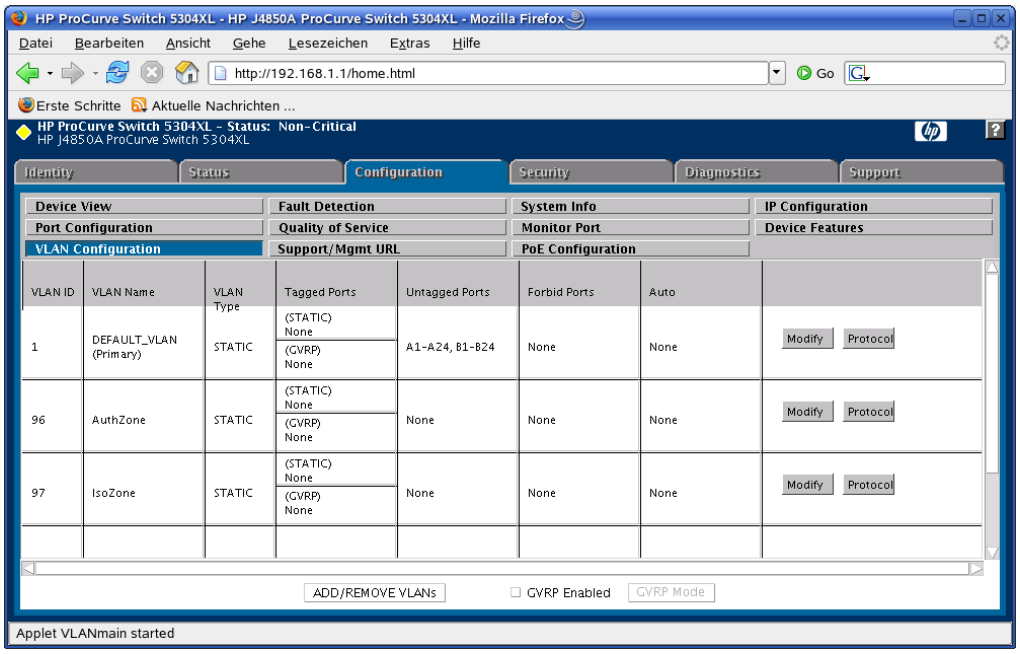


Abbildung 11.3: Die fertige VLAN-Konfiguration dargestellt in der Procurve-Webkonsole

Damit ist der Switch für seinen Einsatz in der Testumgebung fertig konfiguriert. Die Konfiguration eines Cisco-Switches für diesen Einsatz verläuft ähnlich und kann in [Cis04] nachgelesen werden.

11.1.3 Installation der Software

Installation auf AR

Auf Seiten des AR existiert ein Installationsprogramm, welches Daniel Wuttke im Rahmen seines Beitrags zu TNC@FHH entworfen hat (siehe [Wut06]). Dieses Programm erledigt folgendes automatisch:

- Der TNC Client und die verfügbaren IMC-DLLs werden auf den Rechner kopiert.
- Die IMCs werden in der Windows-Registrierung (gemäß Windows Binding aus IF-IMC) registriert.
- Die erforderliche cygwin-dll sowie alle anderen DLLs werden auf den Rechner übertragen.
- Der NDISProt-Treiber wird kopiert, aber nicht installiert.

Somit muss nur noch der NDISProt-Treiber von Hand installiert (siehe Kapitel 8.2.2) und gestartet werden. Daraufhin ist der TNC Client startklar. Dies funktioniert aber nur, wenn der aktuelle Benutzer über Administratorrechte verfügt, weil der direkte Zugriff auf das Netz nur Administratoren vorbehalten ist.

Installation auf PDP

Die für TNC@FHH-angepasste und bereits wie in Kapitel 9.2.1 konfigurierte FreeRadius-Distribution wird auf den Zielrechner kopiert. Zusätzlich werden die verfügbaren IMVs ins Dateisystem kopiert und die IMVs mit ihrem Installationspfad in `tnc_config` eingetragen. Im letzten Schritt werden `tncs_fhh.conf` (siehe Kapitel 9.3) und `tnc_log.properties` (siehe Kapitel 10.1) nach `/etc/tnc` kopiert.

Nun kann FreeRadius über das mitgelieferte Skript `startRadius` (mit Root-Rechten) gestartet werden.

11.2 Test des Zusammenspiels von AR, PEP und PDP

Für die Tests wurden die von Daniel Wuttke entworfenen IMCs und IMVs zur Integritätsprüfung eingesetzt. Dadurch konnte gezeigt werden, dass IF-IMC bzw. IF-IMV korrekt umgesetzt werden.

11.2.1 Beobachtungen während des Handshake

Über den Netzwerkverkehr mittels 802.1x und darin EAP-TNC wurde bisher in abstrakter Form viel gesprochen. Nun soll aber gezeigt werden, dass die Kommunikation wirklich so abläuft, wie beschrieben wurde. Dazu wird der Handshake zwischen AR und PDP durchgeführt und währenddessen der Netzwerkverkehr mit Hilfe von Ethereal[3] sowohl auf AR und PDP in der Testumgebung beobachtet. Da es zu weit führen würde, jede einzelne übertragene Nachricht zu zeigen, wird sich hier auf einige wenige exemplarische Nachrichten beschränkt:

RADIUS Access-Challenge mit EAP-TNC-Start

Abbildung 11.4 zeigt das erste vom EAP-TNC-Modul in FreeRadius verschickte Paket einer neuen Verbindung. Zu sehen ist hier, dass die EAP-Nachricht als Attribut in dem RADIUS-Paket eingebettet ist. Als EAP-Typ der EAP-Nachricht wird zwar „Tavakoli“ angegeben; dies liegt aber an der EAP-Typ-Nummer 38, welche wohl früher für „Tavakoli“ genutzt wurde, nun aber für EAP-TNC (laut IF-T)

[3] Netzwerkanalyse-Tool, http://www.ethereal.com

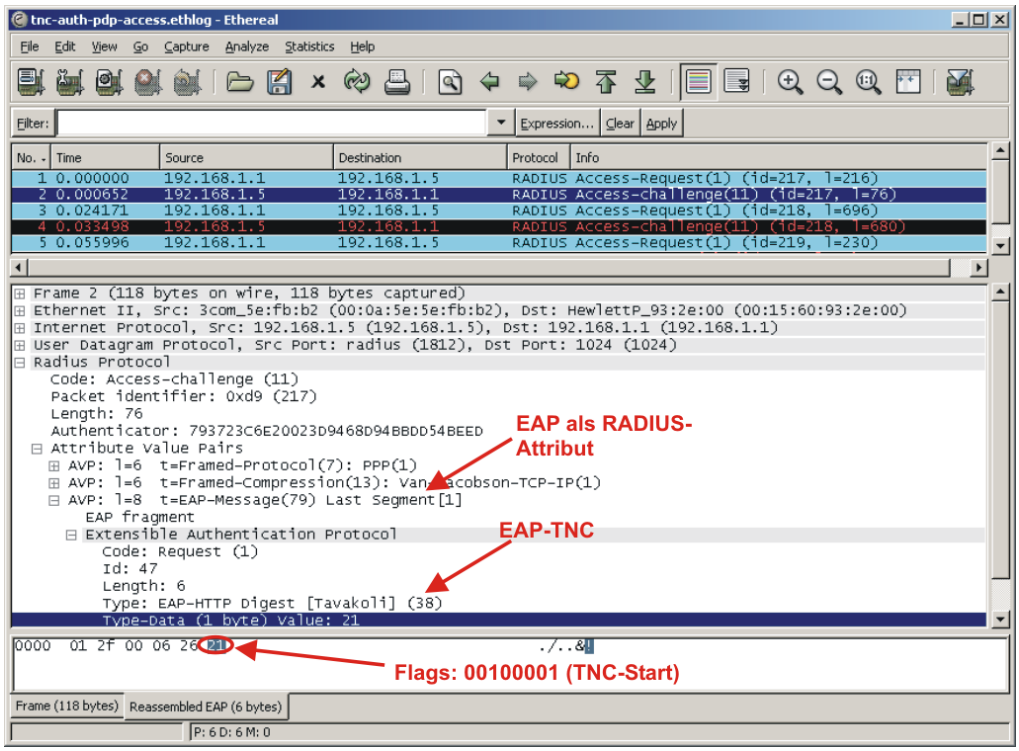


Abbildung 11.4: EAP-TNC-Start als RADIUS-Attribut in einer RADIUS Access-Challenge

genutzt werden soll. Ethereal kennt EAP-TNC nicht und kann deshalb die einzelnen Bytes nicht logisch auseinander nehmen, sondern zeigt nur die Byte-Folge unter „Type-Data“ an. Im *EAP-TNC-Start*-Paket enthält diese „Type-Data“ nur die Flags, so dass ersichtlich wird, dass außer dem Start-Flag nur das Versions-Tag (das letzte Tag) gesetzt ist.

EAPOL mit EAP-TNC-Request und erstem TNCCS-Fragment

Das zweite mitprotokollierte Paket (siehe Abbildung 11.5) ist ein *EAPOL*-Paket. Es enthält ein *EAP Request* vom Typ *EAP-TNC*. In diesem Paket wird das erste

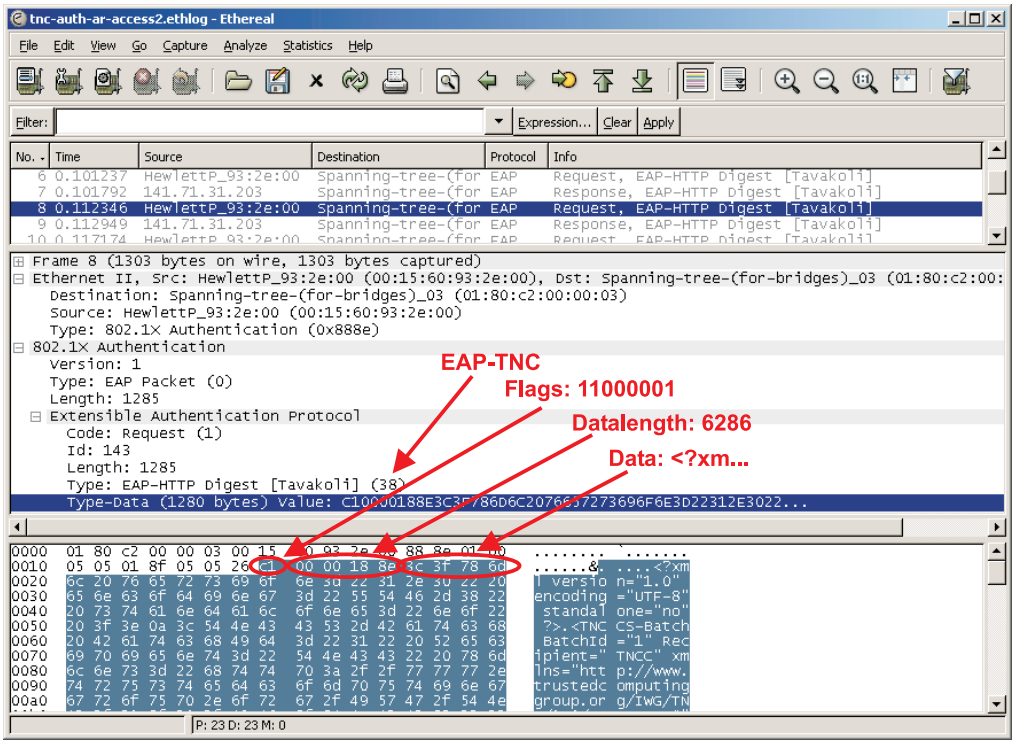


Abbildung 11.5: EAPOL-Paket mit EAP-TNC-Nachricht (inkl. Datalength-Feld) in Ethereal

Fragment einer TNCCS-Nachricht vom AR an den PDP übertragen. Im Gegensatz zum vorherigen Paket ist hier „Type-Data“ sehr viel länger und besteht aus den Flags (1 Byte), der Datenlänge der gesamten TNCCS-Nachricht (4 Bytes) und den eigentlichen Daten. In den Flags sind das erste Flag (lengthIncluded) und zweite Flag (moreFragments) gesetzt, die Version (letzte 3 Bytes) steht wieder auf 1. Dies entspricht den Anforderungen aus IF-T für das erste Fragment einer aufgeteilten TNCCS-Nachricht. Die Länge der Gesamtnachricht (hier: 6286) ist ebenfalls vorhanden.

EAPOL mit EAP-TNC-Response und TNC-Acknoledgement

Das dritte Paket (siehe Abbildung 11.6) wurde wieder auf dem AR mitprotokolliert und ist deshalb wieder ein *EAPOL*-Paket. Dieses Paket enthält eine *EAP-TNC-*

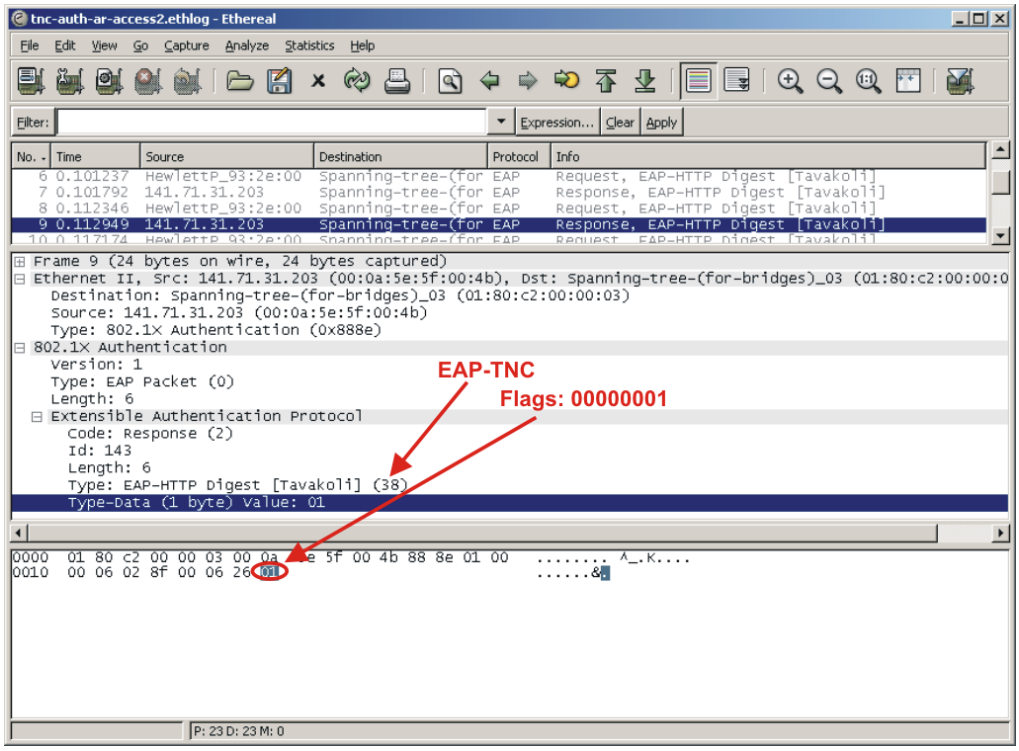


Abbildung 11.6: EAPOL-Paket mit TNC-Acknoledgement-Nachricht in Ethereal

Acknoledgement-Nachricht. Dies ist daran zu erkennen, dass keine Flags (abgesehen von der Versionsnummer) gesetzt sind und keine Daten angehängt sind.

EAPOL mit EAP-TNC-Request und weiterem TNCCS-Fragment

Das nächste dargestellte Paket (siehe Abbildung 11.7) wurde auch auf dem AR mitprotokolliert und enthält eine weiteres TNCCS-Fragment einer Gesamtnachricht. Dies ist daran zu erkennen, dass bei den Flags nur das moreFragments-Flag gesetzt ist, das Feld mit der Gesamtlänge nicht vorhanden ist (lengthIncluded-Flag=0) und Daten anhängen.

RADIUS Access-Accept und VLAN-Übermittlung

Das letzte dargestellte Paket (siehe Abbildung 11.8) enthält ein *RADIUS Access-Accept*. Dieses wird übermittelt, wenn der Integritätstest erfolgreich oder mit Isolation endet. Innerhalb des *RADIUS Access-Accept* wird ein *EAP Success* übertragen,

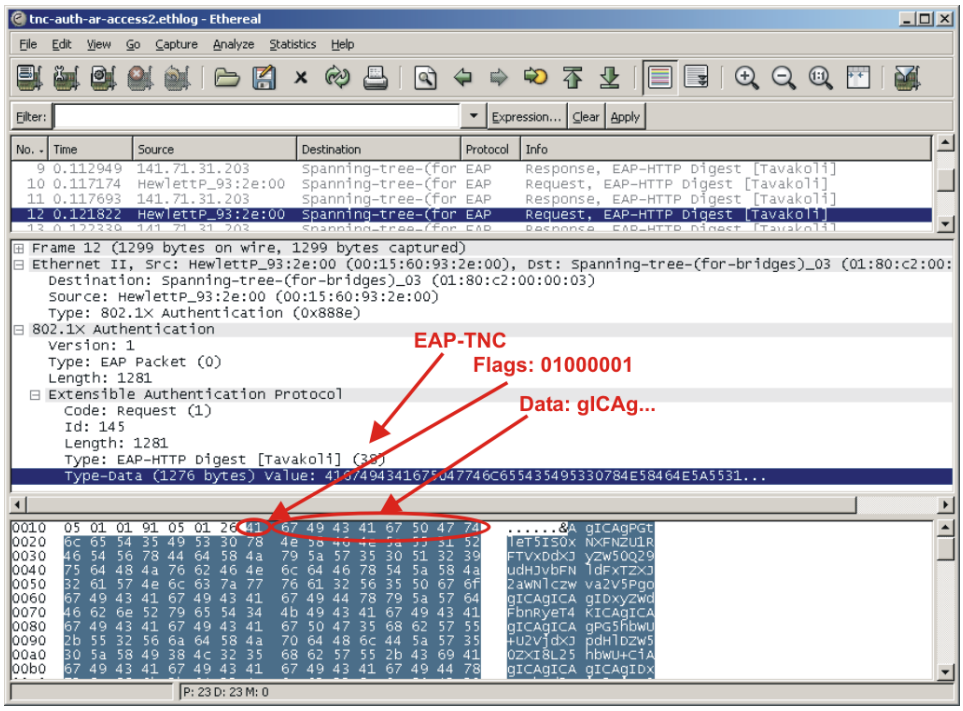


Abbildung 11.7: EAPOL-Paket mit EAP-TNC-Nachricht ohne Datalength-Feld

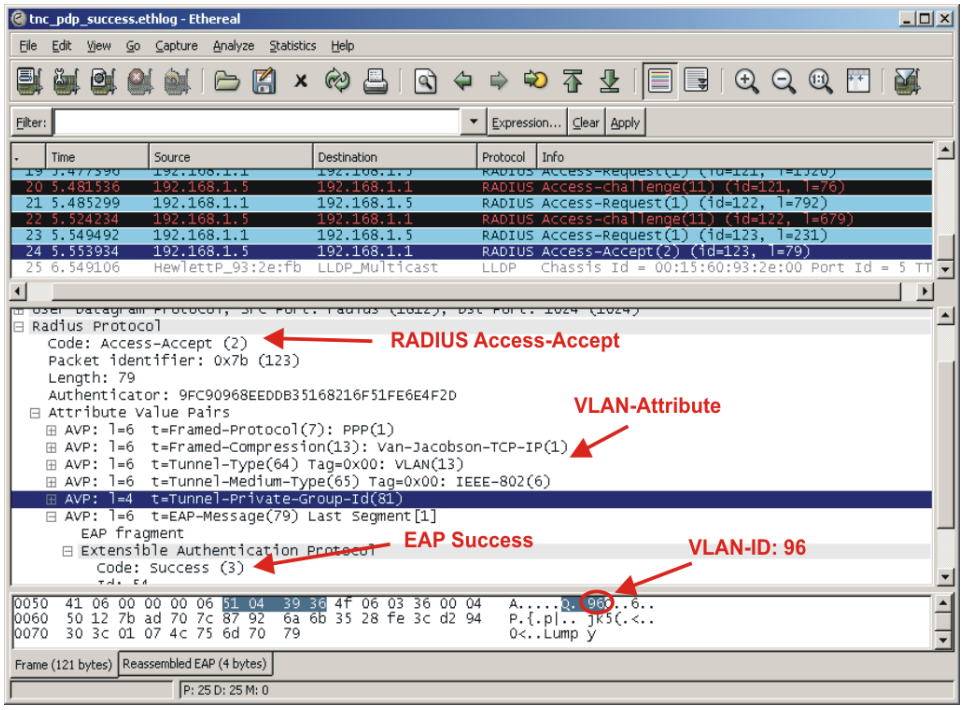


Abbildung 11.8: RADIUS Access-Accept-Paket mit EAP Success und VLAN-Zuordnung (VLAN-ID=96)

das der PEP dann später an den AR weiterleitet. In diesem Fall endete die Integritätsprüfung mit Erfolg. Das ist an der VLAN-Zuordnung zu erkennen, die als RADIUS-Attribut zum PEP übertragen wird und den AR in das „normale“ VLAN 96 steckt.

Die Auswahl dieser Pakete soll exemplarisch zeigen, dass die Kommunikation zwischen AR und PDP gemäß den Spezifikationen RADIUS, 802.1x, EAP-TNC und IF-PEP korrekt erfolgt.

11.2.2 Beobachtungen nach Abschluss des Handshake

Eine TNC-Architektur muss die Kommunikation zwischen PDP und AR korrekt umsetzen. Der Aufwand des Integritätstests ist aber umsonst, wenn die Entscheidung des PDP nach Abschluss des Handshake nicht vom PEP physikalisch umgesetzt wird. Deshalb soll jetzt gezeigt werden, dass in der hier vorgestellten Implementierung die Entscheidung des PDP in eine Aktion des PEP mündet.

Ergebnis einer erfolgreichen Integritätsüberprüfung

Wenn der Integritätstest ergibt, dass die Vertrauenswürdigkeit des AR gegeben ist, übermittelt der PDP *RADIUS Access-Accept* an den PEP und fordert ihn auf, den Port des AR ins VLAN für die sicheren Endgeräte aufzunehmen. In der Testumgebung muss also Port A11 zum VLAN 96 hinzugefügt werden. Das dies wirklich geschieht, zeigt Abbildung 11.9.

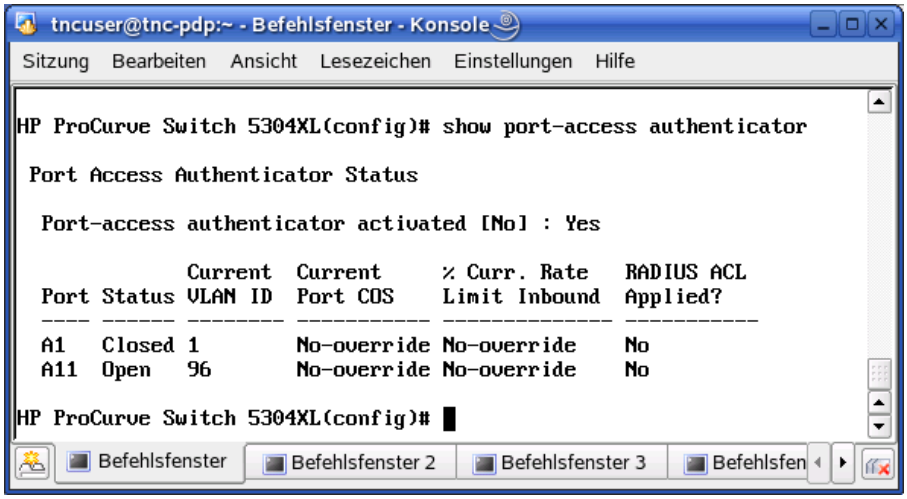


Abbildung 11.9: Nachdem die Integritätsprüfung eine Zutrittserlaubnis ergab, ist der Port A11 zum VLAN 96 zugeordnet.

Ergebnis einer Integritätsprüfung mit anschließender Isolierung

Wenn der Integritätstest ergibt, dass die Vertrauenswürdigkeit des AR zwar nicht gegeben ist, aber durch Nachbesserungen zu erreichen ist, dann übermittelt der PDP *RADIUS Acccess-Accept* an den PEP und fordert ihn auf, den Port des AR ins isolierte VLAN aufzunehmen. In der Testumgebung muss demnach Port A11 zum VLAN 97 zugeordnet werden, was wirklich geschieht. Dies zeigt Abbildung 11.10.

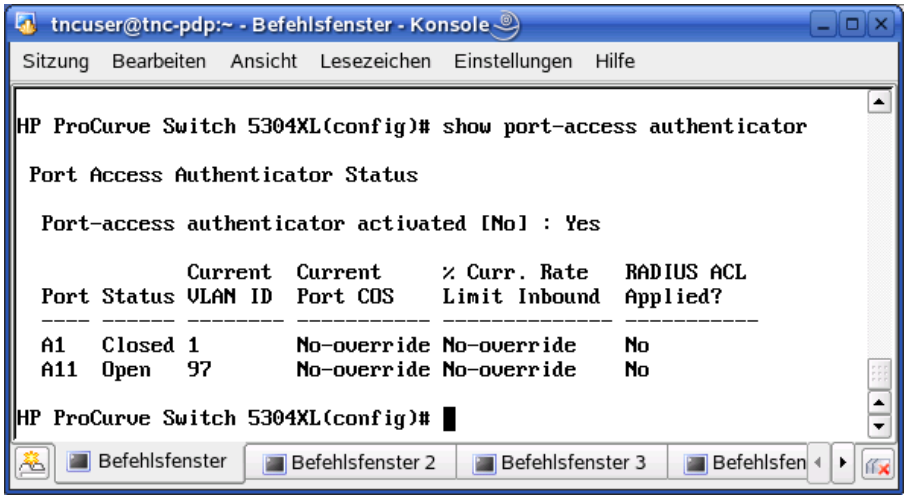


Abbildung 11.10: Nach dem Isolierungsbefehl ist der Port A11 dem VLAN 97 zugeordnet.

Ergebnis einer erfolglosen Integritätsprüfung

Wenn am Ende eines Integritätstests die Erkenntnis steht, dass die Vertrauenswürdigkeit eines AR nicht gegeben ist, so schickt der PDP *RADIUS Access-Reject* an den PEP. Daraus leitet der PEP direkt ab, dass er den Port des AR schließen soll. Für die Testumgebung bedeutet dies, dass Port A11 geschlossen wird. Dies setzt der PEP auch tatsächlich um (siehe Abbildung 11.11).

11.3 Fazit zur Implementierung

In den Kapiteln zur Implementierung wurde gezeigt, wie der Entwicklungsprozess von AR und PDP praktisch abläuft.

Im Rahmen der Implementierung mussten neben der Umsetzung der Konzepte eine Reihe von Detailproblemen gelöst werden: Auf Seiten des AR wurde ein Netzwerktreiber gebaut, der den Zugriff aus Windows heraus auf Ethernet ermöglicht und

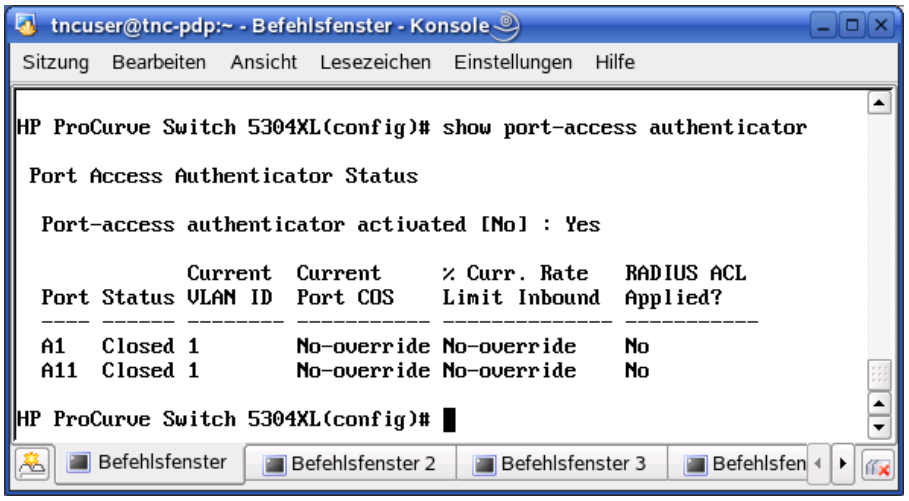


Abbildung 11.11: Weil der Integritätstest fehlschlug, ist der Port A11 geschlossen.

eine Benutzeroberfläche mit wxWidgets entwickelt. Für den PDP wurde das EAP-Modul von FreeRadius um EAP-TNC-Funktionalitäten erweitert. Zudem wurde ein Verbindungsmanagement hinzugefügt und die Konfigurierbarkeit des PDP über eine Konfigurationsdatei erhöht.

Beim Einsatz in der aufgebauten Testumgebung konnte gezeigt werden, dass die entwickelte Software gemäß der TNC-Spezifikationen funktioniert. Die Interoperabilität mit fremden IMCs und IMVs konnte ebenso gezeigt werden.

12 Zusammenfassung und Ausblick

12.1 Zusammenfassung

Im Rahmen dieser Masterarbeit wurde auf Grundlage der TNC-Spezifikationen eine Client- und eine Serverapplikation entwickelt, die im Zusammenspiel die Integrität einer Netzwerkkomponente überprüfen.

Neben der reinen Umsetzung der TNC-Spezifikationen bestand die Herausforderung der Arbeit darin, tragfähige und wiederverwendbare Architekturen zu schaffen, in denen eine Reihe von Teilproblemen zu lösen waren.

Als Rahmenbedingung war gegeben, dass die Integritätsprüfung im Ethernet-basierten LAN abläuft und auf 802.1x basiert. Zudem bestand die Anforderung, dass die Client-Applikation (Access Requestor) unter Windows und die Server-Applikation (Policy Decision Point) unter Linux läuft.

Der Entwurf von AR und PDP wurde aber besonders daraufhin ausgerichtet, dass die Komponenten einfach auf andere Plattformen portiert werden können. Dies erfolgte über eine klare Trennung von Verantwortlichkeiten in objekt-orientiert entworfenen Systemen und damit der Kapselung von plattform-spezifischem Code.

Für beide Komponenten musste zunächst evaluiert werden, inwiefern Teilfunktionalitäten in der NA-Schicht von anderen Programmen durch deren Anbindung übernommen werden können, oder ob die gesamte Funktionalität neu entwickelt werden muss.

Der **Access Requestor** wurde als Gesamtapplikation entworfen, in dem auch die NA-Schicht selbst entwickelt wurde, da keine existierenden Lösungen gefunden wurden, die leicht anzubinden waren. Für den Zugriff auf das Ethernet-Netzwerk aus Windows heraus wurde ein Netzwerktreiber gebaut. Die Architektur des AR ist durch eine klare Trennung der Schichten gekennzeichnet. Es existieren klare Zuständigkeiten der einzelnen Klassen im System, wie anhand statischer Diagramme und Ablaufbeschreibungen gezeigt wurde. Für eine bessere Benutzerakzeptanz wurde eine grafische Oberfläche konzipiert, die dem Benutzer die Kontrolle über die Integritätsprüfung gewährt und gleichzeitig die Abläufe transparent macht.

Für die Architektur des **Policy Decision Point** wurde ein anderer Ansatz gewählt: Der PDP ist zweigeteilt konzipiert. Ein Teil behandelt die Netzwerk-nahen Aufgaben und ist Bestandteil des EAP-Moduls von FreeRadius, während der größere Teil, der erweiterte Aufgaben der NA-Schicht und den TNCS enthält, als Shared Object vom EAP-Modul eingebunden wird. Dadurch kann die RADIUS-Funktionalität von

FreeRadius und die EAP-Funktionalität speziell vom EAP-Modul genutzt werden und der größte Teil des PDP objekt-orientiert entwickelt werden. Erweiterungen am EAP-Modul waren nötig, damit das Modul die Kommunikation basierend auf EAP-TNC unterstützt. Auch im PDP sind die Schichten klar getrennt und die Aufgaben fein-granular auf mehrere Klassen verteilt, wie gezeigt wurde. Im TNCS wurde ein einfaches Policy-Management entworfen, welches eine Reihe von unterschiedlichen Sicherheitskonfigurationen ermöglicht.

Da eine Reihe von Problemstellungen im AR und PDP ähnlich gelöst werden, wurden diese identifiziert und durch gemeinsame Klassen auf AR und PDP einheitlich umgesetzt. Dies sind die Anbindung der IM-Schicht, sowie die TNCCS-Kommunikation inklusive der Nachrichtenfragmentierung.

Anschließend wurde gezeigt, wie die Applikationen konkret implementiert wurden. Zum einen wurden die eingesetzten und angepassten **Entwicklungsumgebungen und Build-Verfahren** vorgestellt, zum anderen wurden **Implementierungsdetails** für Teilkomponenten der Systeme gezeigt. Dazu gehörte, wie der Netzwerktreiber für den AR entwickelt oder das EAP-Modul von FreeRadius erweitert wurde.

Zum Abschluss wurde das Zusammenspiel von entwickeltem AR und PDP in einer **Testumgebung** überprüft. Dabei wurde die Interoperabilität der Implementierungen mit den IMCs und IMVs aus Daniel Wuttkes Arbeit überprüft. Dabei zeigte sich zum einen, dass die TNC-Spezifikationen gut umsetzbar und weitgehend schlüssig sind, und zum anderen, dass die entwickelten Komponenten die an sie (von den TNC-Spezifikationen) gestellten Aufgaben erfüllen.

So kann abschließend festgestellt werden, dass die entwickelten Software-Komponenten die gestellten funktionalen und nicht-funktionalen Anforderungen sowie die TNC-Spezifikationen weitgehend (siehe Ausblick) korrekt umsetzen und dabei jeweils auf tragfähigen und einfach erweiterbaren Architekturen basieren.

12.2 Ausblick

Einige Aspekte sind während der Erstellung dieser Masterarbeit aufgefallen, die als Erweiterung dieser Arbeit und des Projekts „TNC@FHH“ in Zukunft behandelt werden könnten.

12.2.1 Einsatz von Trusted Plattform Modules

Der Integritätstest in der TNC-Architektur liefert nur ein vertrauenswürdiges Ergebnis, wenn sicher gestellt ist, dass die IMCs und der TNC Client selbst vertrauenswürdig sind. Dies ist nur dann der Fall, wenn sie selbst nicht verändert sind oder unterlaufen werden.

Dies sicherzustellen ist Ziel der TPM Work Group der TCG, welche das Trusted

Platform Module spezifiziert.[1] Dies ist ein Hardware-Modul auf dem Motherboard, welches Informationen zur Plattformintegrität, sowie kryptografische Schlüssel speichert und Kommandos auf geschützte Speicherbereiche ermöglicht. Dies führt dazu, dass kein falscher Zustand des Systems vorgetäuscht werden kann.

Die TNC-Architektur sieht vor, dass über das TPM IMCs und TNC Client abgesichert werden können (siehe [TCG06a, Kapitel 6, S. 26]). Eine Erweiterung dieser Arbeit wäre es, zu evaluieren, ob TPM verfügbar und einsetzbar ist, sowie die TNC Client-Implementierung dahingehend zu erweitern.

12.2.2 Verschlüsselung der EAP-TNC-Kommunikation

IF-T verlangt einen höheren Sicherheitsstandard (siehe [TCG06e, Kapitel 5.4.2, S. 20f]), als es in der hier vorgestellten Implementierung umgesetzt wurde. Es finden weder verschlüsselte Authentisierung zwischen NAA und NAR statt, noch wird die TNCCS-Kommunikation verschlüsselt.

Es könnte demnach diese Lücke in der TNC-Konformität geschlossen werden, indem NAR und das EAP-TNC-Modul um Authentisierung und Verschlüsselung (z.B. über EAP-TTLS) erweitert werden.

12.2.3 Policy-Management

Die im TNC Server verfügbaren Sicherheitsrichtlinien ermöglichen eingeschränkte Konfigurationsmöglichkeiten, weil sie die einzelnen IMV-Komponenten nicht individuell einbeziehen. So könnte z.B. das Votum der einzelnen IMVs unterschiedlich auf Grundlage ihrer Bedeutung für die Gesamtsicherheit gewichtet werden. Der TNC Server könnte um eine Policy-Komponente erweitert werden, welche auf einem komplexen Richtlinienmodell basiert, welches leicht konfigurierbar ist und die vorhandenen IMV-Komponenten individuell einbezieht.

[1] https://www.trustedcomputinggroup.org/groups/tpm/

Anhang A

Dokumente zum Entwurf

A.1 Sequenzdiagramme

Abbildung A.1: Gesamtablauf TNCCS und IMC-Kommunikation im TNC Client

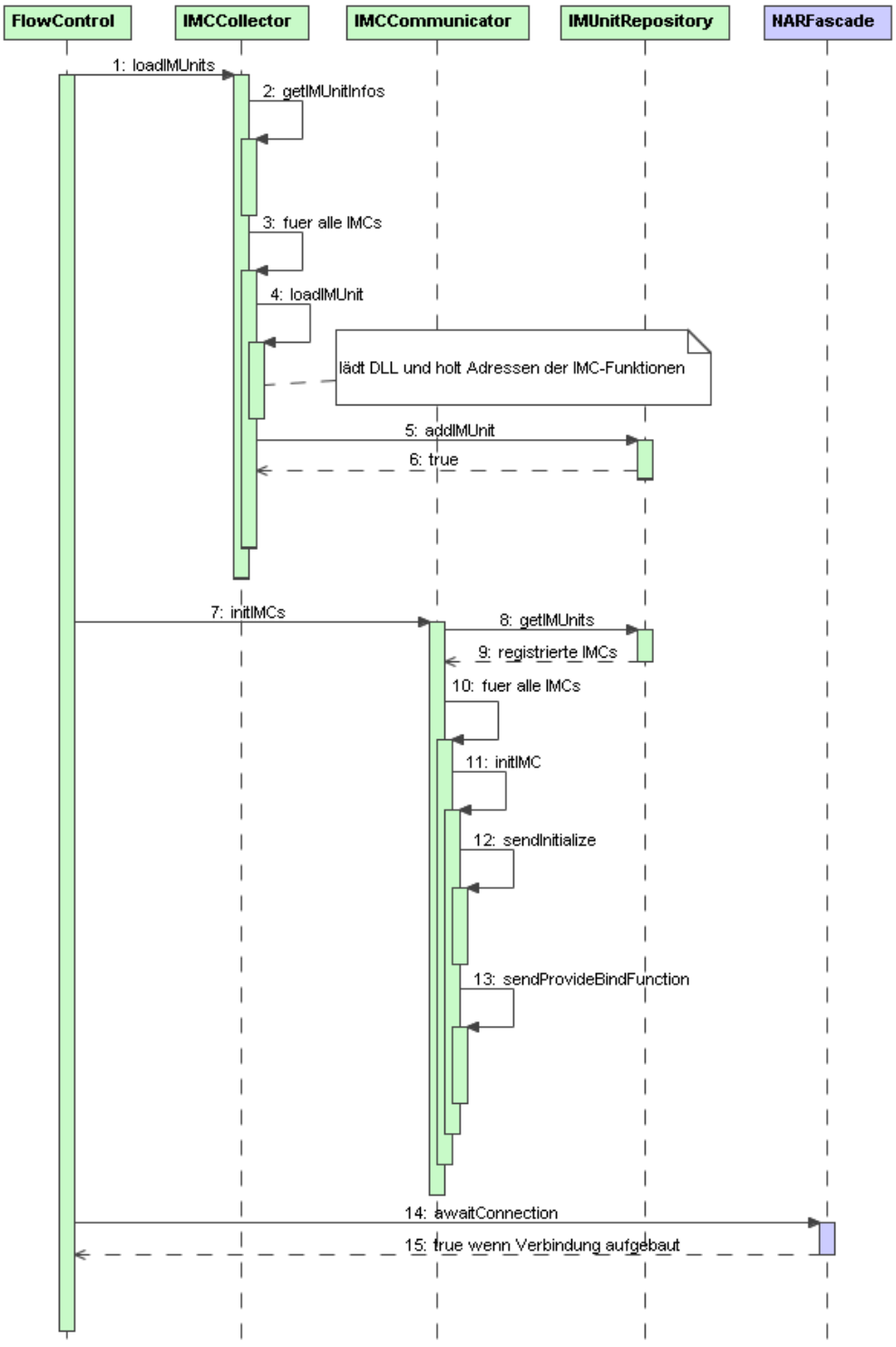


Abbildung A.2: Initialisierung des TNC Client

A.2 Klassendiagramme

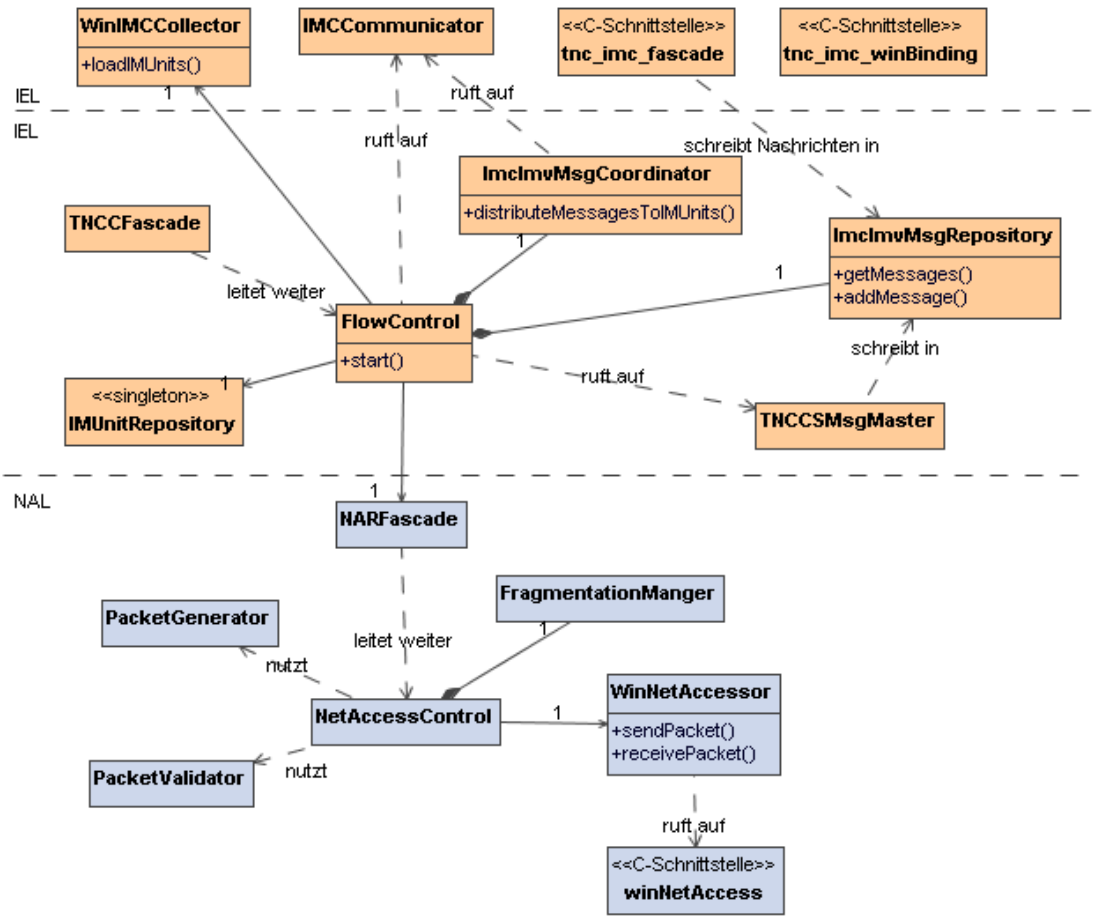


Abbildung A.3: Statische Gesamtarchitektur des AR; die einzelnen Komponenten sind durch waagerechte Linien getrennt

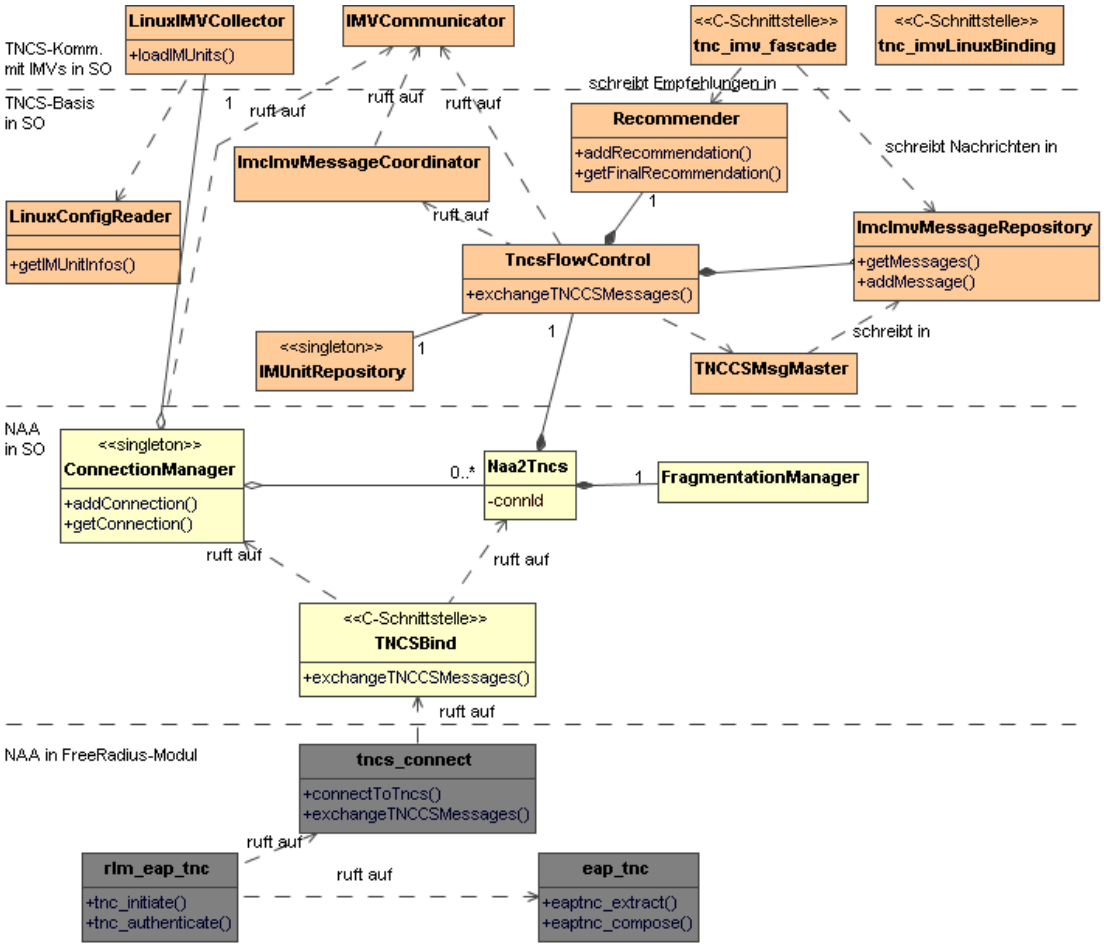


Abbildung A.4: Statische Gesamtarchitektur des PDP; die einzelnen Komponenten sind durch waagerechte Linien getrennt

A.3 Screenshots des TNC Client

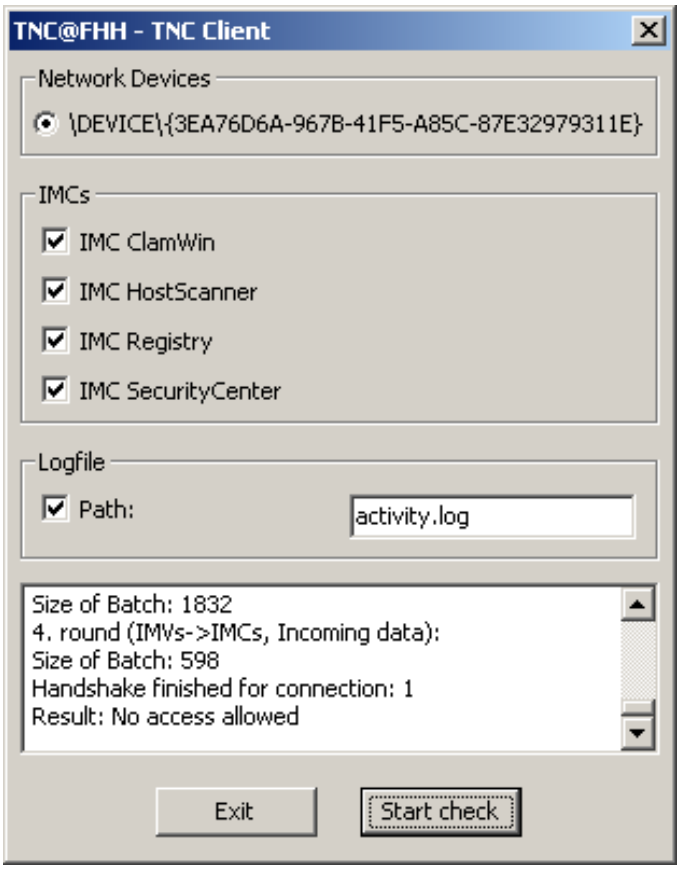


Abbildung A.5: TNC Client-Oberfläche: Die Integritätsprüfung endet mit Zutrittsverweigerung.

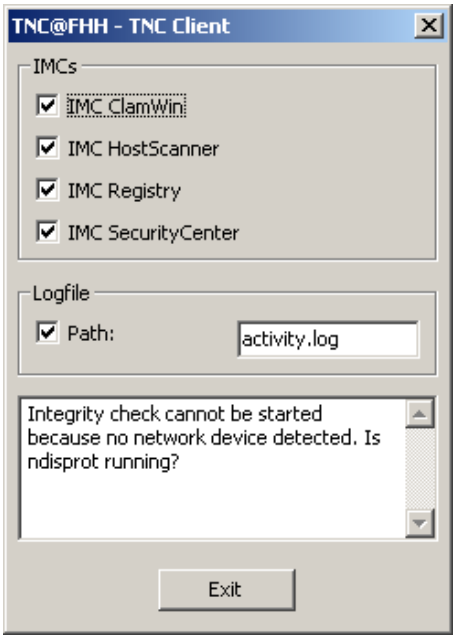


Abbildung A.6: TNC-Client-Oberfläche, wenn NDISProt nicht gestartet wurde. Die Integritätsprüfung kann nicht gestartet werden.

Anhang B

Entwicklung, Installation und Konfiguration

B.1 Entwicklungsumgebung unter Windows

Die folgenden Tabellen und Screenshots zeigen die in Eclipse nötigen Einstellungen, um den AR erfolgreich zu kompilieren und zu bauen. Die einzelnen Dialoge sind in Eclipse ausgehend von **Project/Properties/C C++ Build** zu erreichen.

GCC C++ Compiler/Preprocessor/Defined Symbols

Symbol	benötigt für
__GNUWIN32__	wxWidgets
STRICT	wxWidgets
__WXMSW__	wxWidgets
__WINDOWS__	wxWidgets
__WXDEBUG__	wxWidgets

Der Dialog zu den „Defined Symbols“ ist in Abbildung B.2 dargestellt.

GCC C++ Compiler/Directories/Include paths

Pfad	benötigt für
<WXW>/lib/mswd	wxWidgets
<WXW>/build-debug/lib/wx/-include/msw-ansi-debug-static-2.6	wxWidgets
<WXW>/include	wxWidgets
<WXW>/contrib/include	wxWidgets
<WXW>/src/regex	wxWidgets
<WXW>/src/png	wxWidgets
<WXW>/src/jpeg	wxWidgets
<WXW>/src/zlib	wxWidgets
<WXW>/src/tiff	wxWidgets

`<WXW>` steht für den Pfad zur wxWidgets-Installation. Der Dialog zu den „Include paths“ ist in Abbildung B.3 dargestellt.

GCC C++ Compiler/Miscellaneous/Other flags

Flags	benötigt für
-c fmessage-length=0 -fno-rtti -fexceptions	wxWidgets

Der Dialog zu den „Other flags“ ist in Abbildung B.4 dargestellt.

GCC C++ Linker/Libraries/Libraries

Bibliothek	benötigt für
TNCUtil	Nutzung von Util-Funktionen
xerces-c27	Xerces (XML-Parsing)
wx_mswd_core-2.6	wxWidgets
wx_based-2.6	wxWidgets
oleaut32	wxWidgets
ole32	wxWidgets
uuid	wxWidgets
comctl32	wxWidgets

GCC C++ Linker/Libraries/Library Search Path

Pfad	benötigt für
<Pfad zu libxerces-c27.dll.a>	xerces-c27
<Pfad zu libTNCUtil.a>	TNCUtil
<WXW>/build-debug/lib	wxWidgets
<WXW>/contrib/lib	wxWidgets

Der Dialog zu den eingebundenen Bibliotheken ist in Abbildung B.6 abgebildet.

GCC C++ Linker/Miscellaneous/Linker flags

Flag	benötigt für
-mwindows	wxWidgets

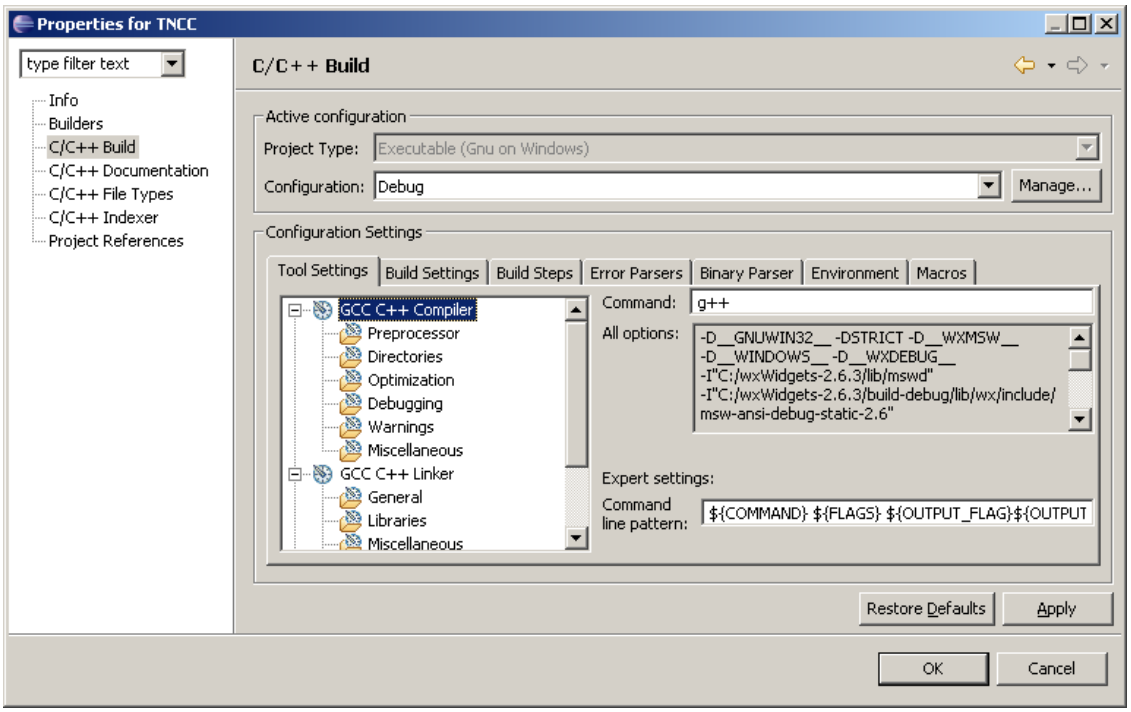


Abbildung B.1: Übersicht über die Compiler-Einstellungen

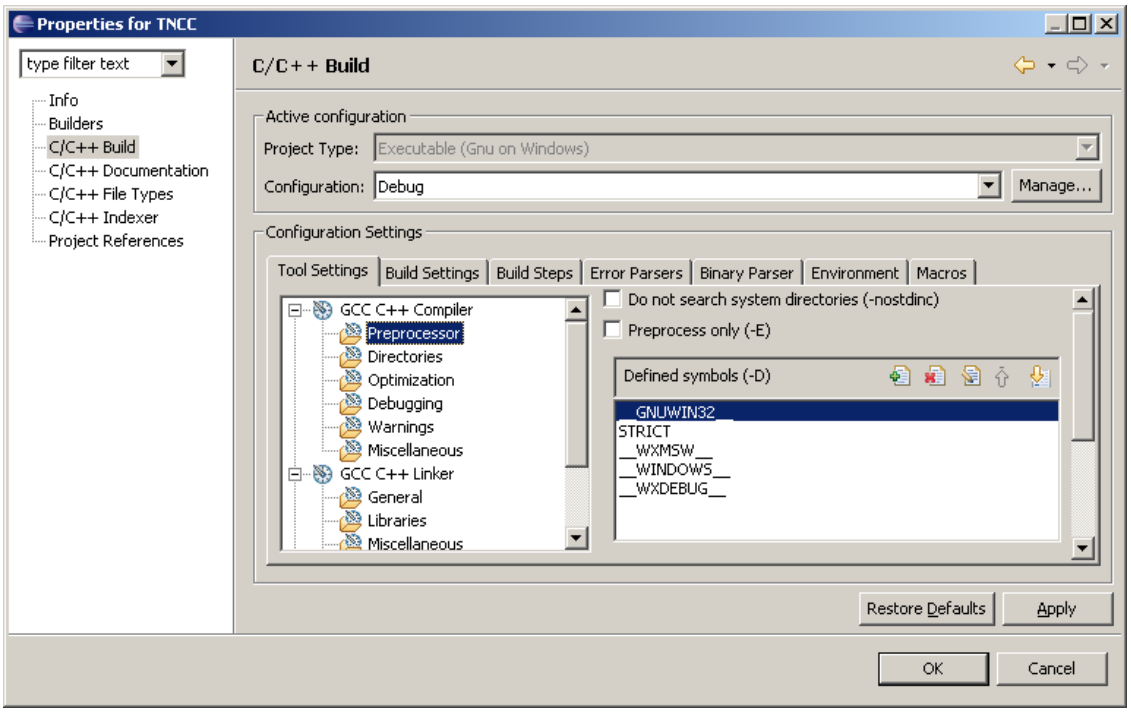


Abbildung B.2: Einstellungen für den Präprozessor

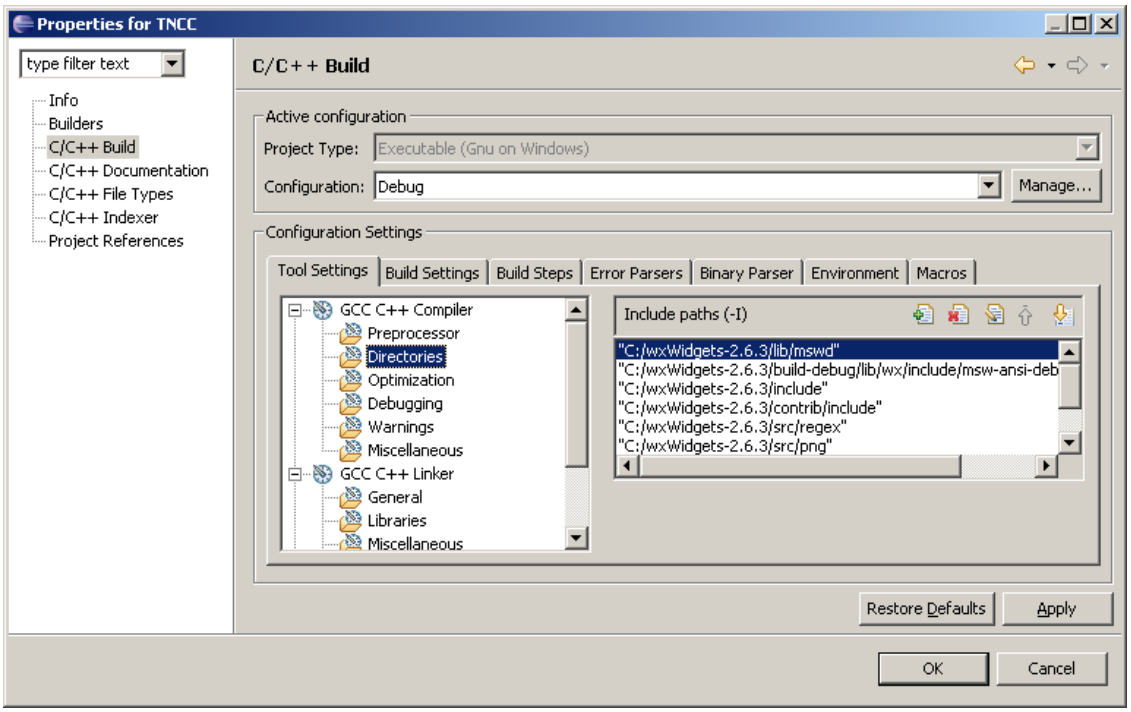


Abbildung B.3: Verzeichnisse, in denen der Compiler Quelldateien suchen soll

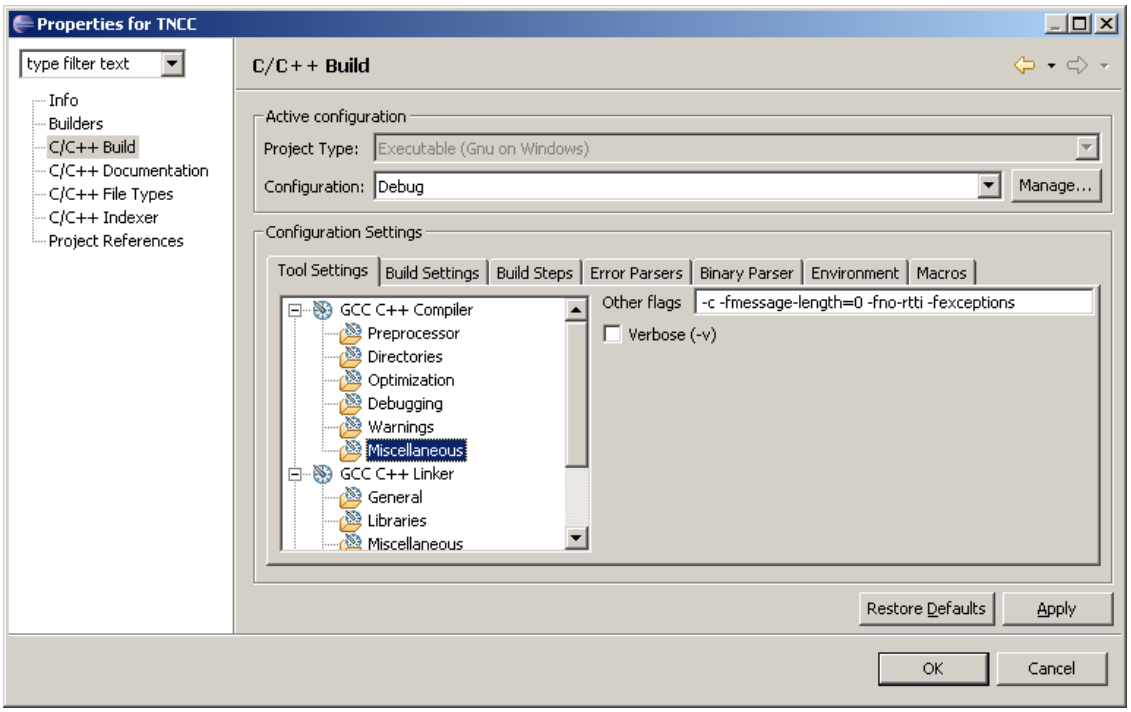


Abbildung B.4: Übrige Compiler-Einstellungen

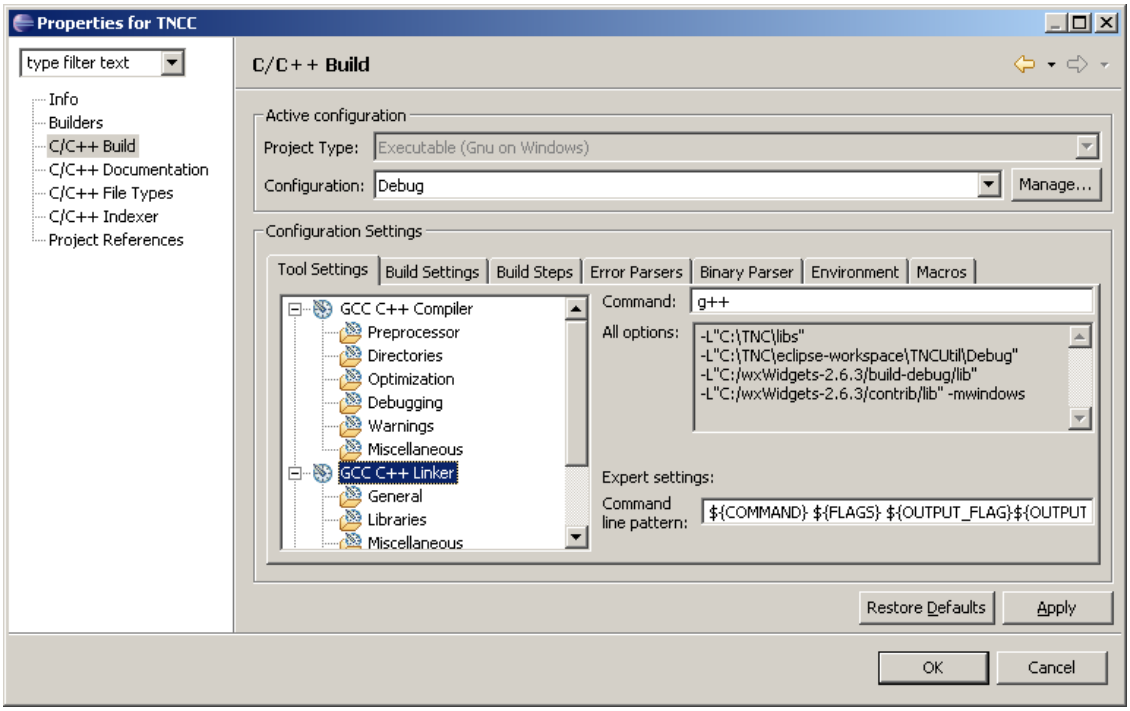


Abbildung B.5: Übersicht über die Linker-Einstellungen

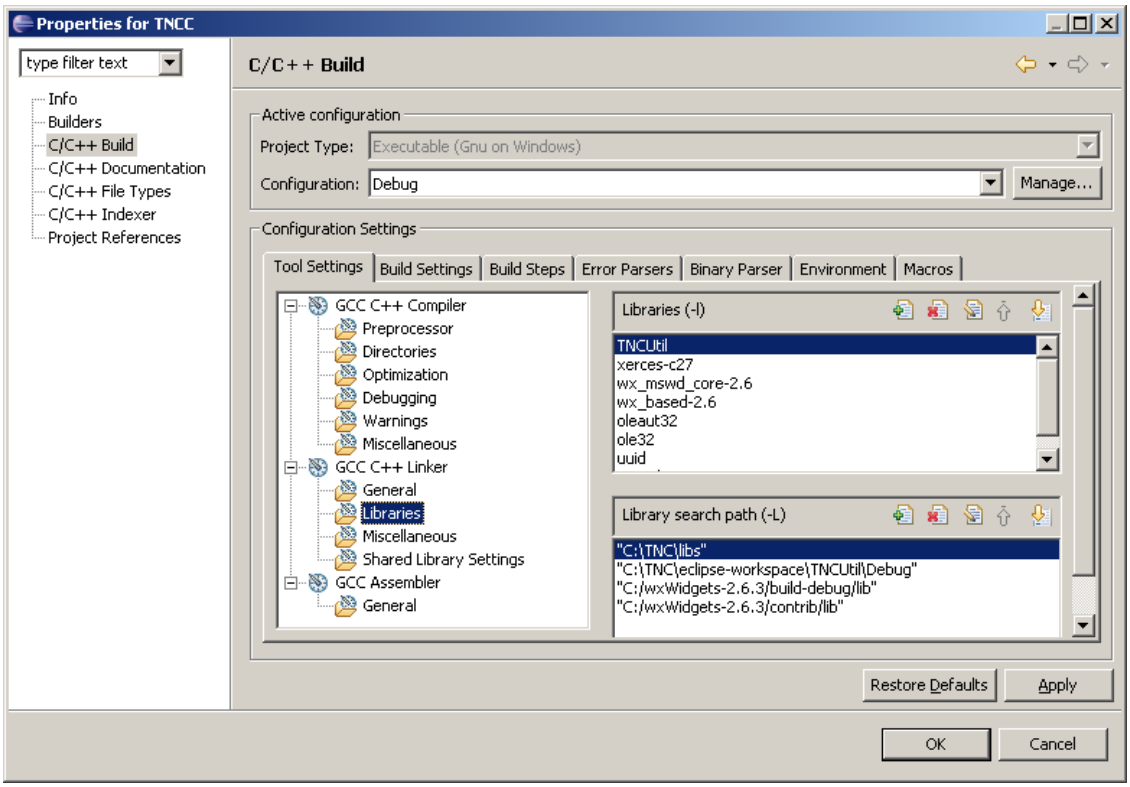


Abbildung B.6: Definition der Bibliotheken, die eingebunden werden sollen

B.2 Entwicklungsumgebung unter Linux

Die folgenden Tabellen und Screenshots zeigen die in Eclipse nötigen Einstellungen, um das Shared Object des TNCS erfolgreich zu kompilieren und zu bauen. Die einzelnen Dialoge sind in Eclipse ausgehend von **Project/Properties/C C++ Build** zu erreichen.

GCC C++ Linker/Libraries/Libraries

Bibliothek	benötigt für
dl	dynamisches Laden von SO-Dateien (IMV-SO-Dateien)
TNCUtilLinux	Nutzung von Util-Funktionen
xerces-c.so	Xerces (XML-Parsing)

GCC C++ Linker/Libraries/Library search path

Pfad	benötigt für
<Pfad zu libxerces-c.so.a>	xerces-c.so
<Pfad zu libTNCUtilLinux.so>	TNCUtilLinux

Der Dialog zu den Bibliothekseinstellungen ist in Abbildung B.8 dargestellt.

GCC C++ Linker/Shared Library Settings

Markierung gesetzt	benötigt damit
Shared	NAA-TNCS als Shared Object gebaut wird

Der Dialog zu den „Shared Library Settings“ ist in Abbildung B.9 dargestellt.

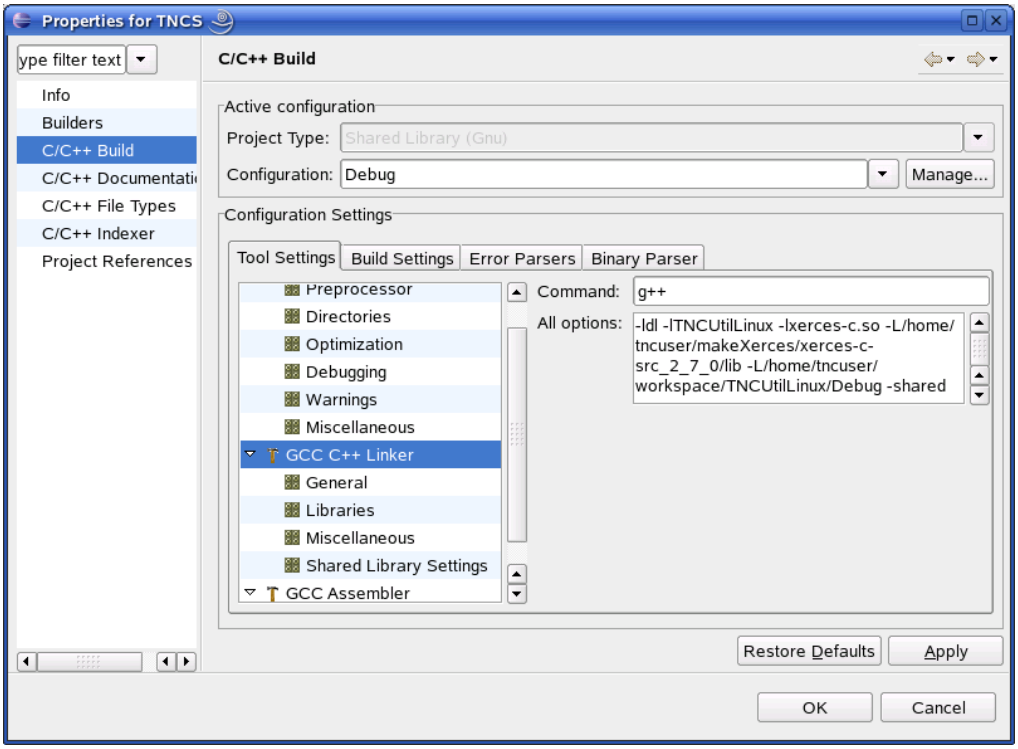


Abbildung B.7: Übersicht über die Linker-Einstellungen

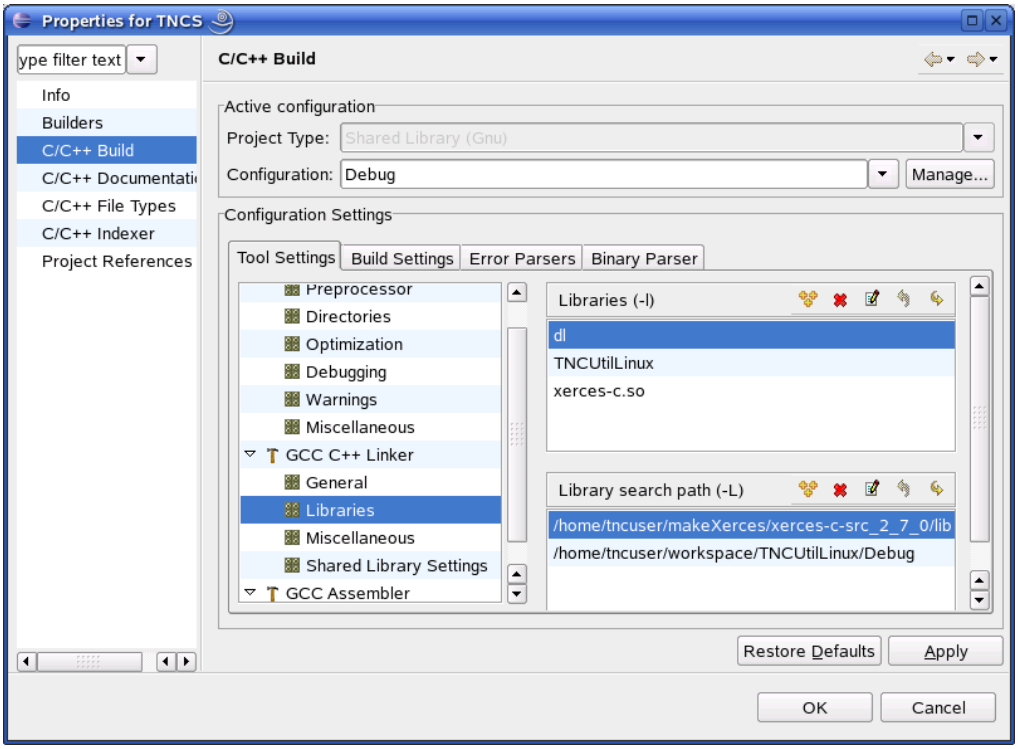


Abbildung B.8: Die Bibliothekseinstellungen des TNCS

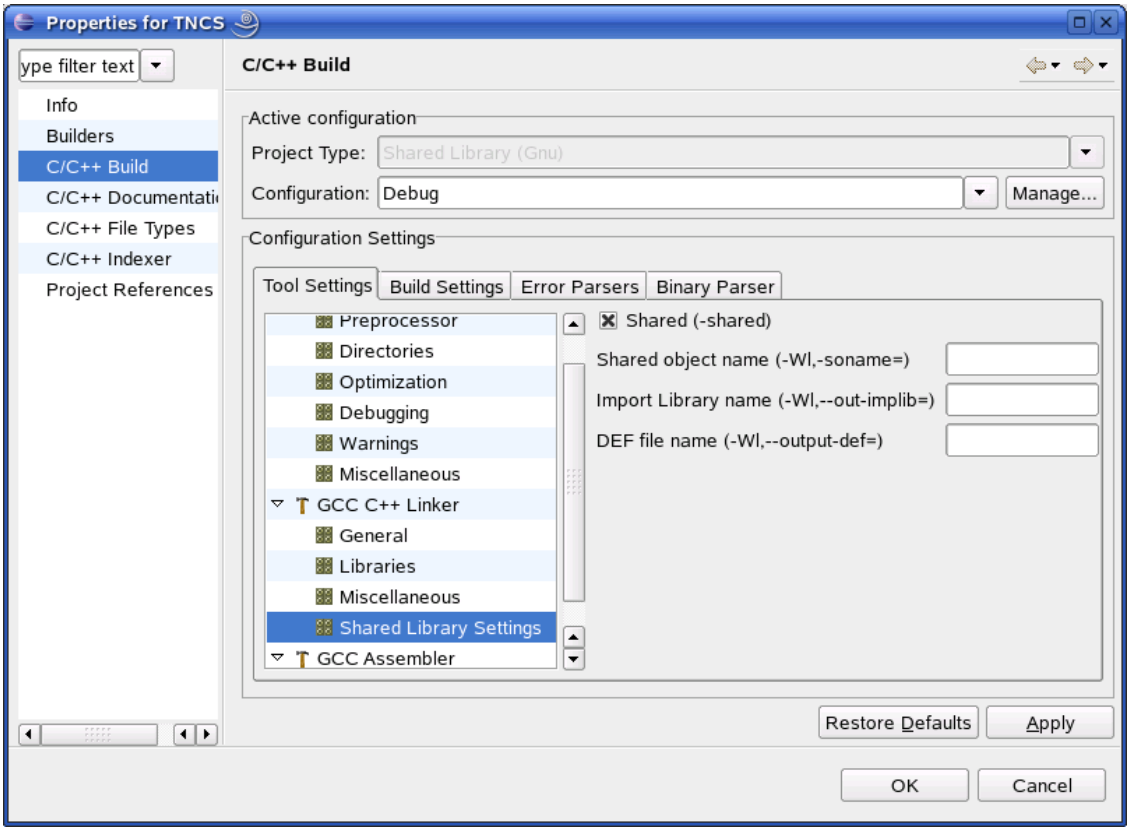


Abbildung B.9: TNCS wird als Shared Object erstellt.

B.3 Installation von NDISProt

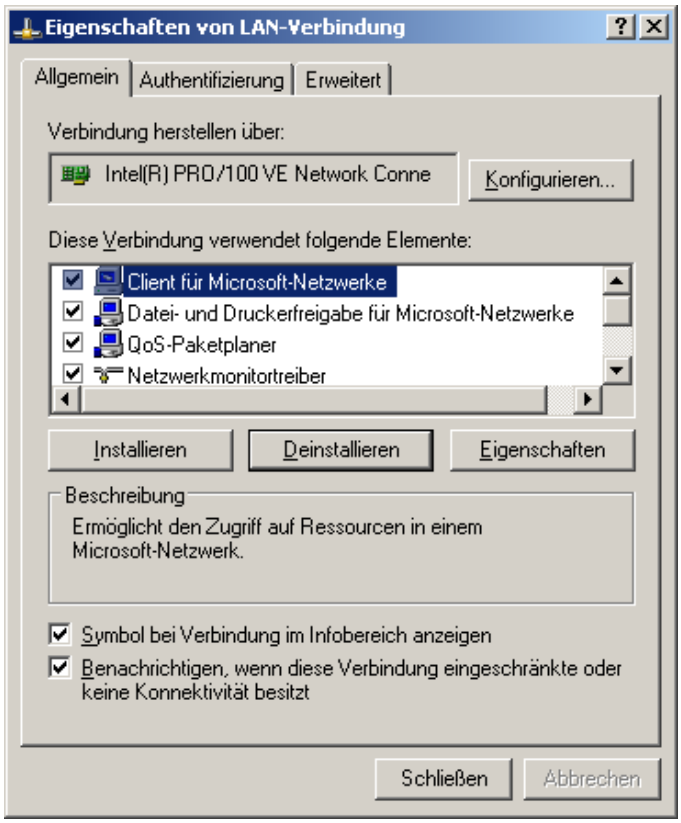


Abbildung B.10: Eigenschaften der LAN-Verbindung vor der Installation von NDISProt; Ein Klick auf „Installieren“ startet die Installation.

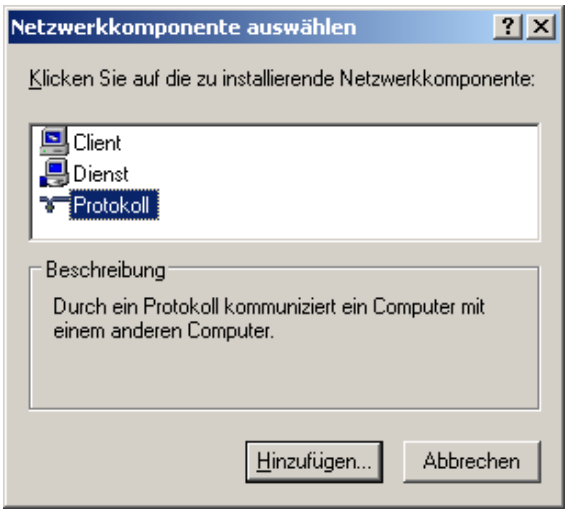


Abbildung B.11: Es wird nun ein „Protokoll“ hinzugefügt.

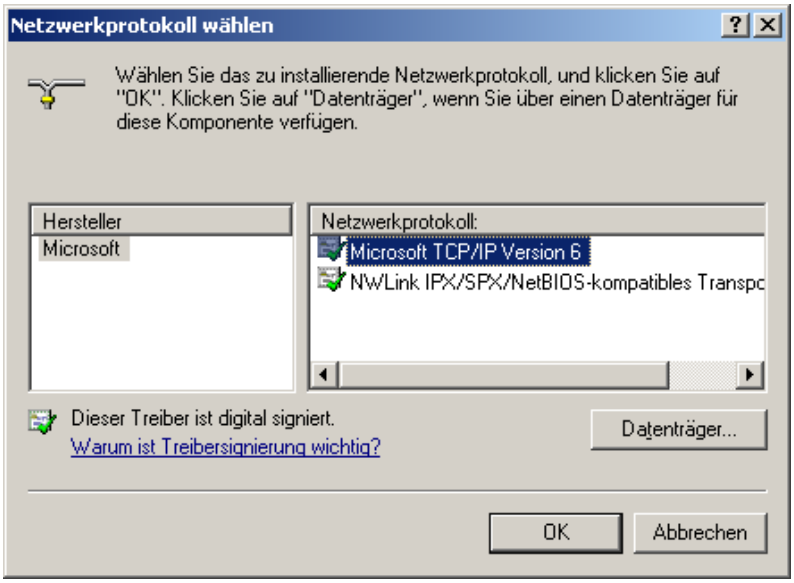


Abbildung B.12: Da NDISProt nicht angezeigt wird, wird über „Datenträger“ danach gesucht.

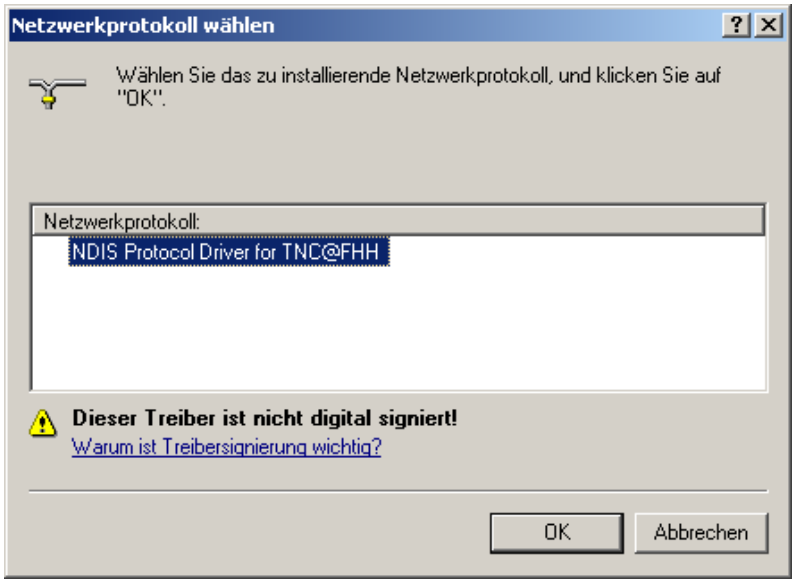


Abbildung B.13: Nachdem `ndisprot.inf` ausgewählt wurde, wird der Treiber gefunden und ausgewählt.

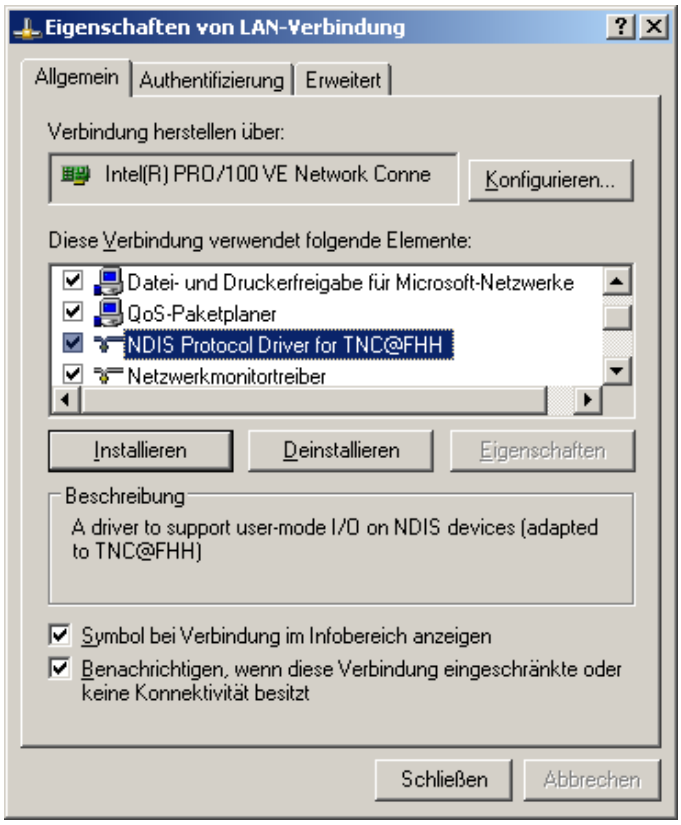


Abbildung B.14: Der Treiber ist installiert und wird in den Eigenschaften der LAN-Verbindung angezeigt.

B.4 Bau von wxWidgets

Bevor wxWidgets zur Entwicklung von Applikationen eingesetzt werden kann, muss es für die jeweilige Plattform gebaut werden. Im Fall des TNC Client ist dies die Cygwin-Umgebung unter Windows. Nach dem Durchlauf der Installationsroutine von wxWidgets liegt die Distribution in $WXWIN. Sie wird für Cygwin gebaut, indem die Schritte aus Listing B.1 in der „Cygwin Bash Shell“ ausgeführt werden.[1]

```
cd $WXWIN
mkdir build-debug
cd build-debug
../configure --with-msw --enable-debug --enable-debug_gdb --disable-shared
make
```

Listing B.1: Bau von wxWidgets für Cygwin

B.5 Konfigurationsdateien

```
#VLAN-Configuration
VLAN_ACCESS=96
VLAN_ISOLATE=97

#Batch-Configuration
BATCH_COUNT=5

#Policy-Configuration
POLICY=5

#possible values:
#1 (positive simple)
#2 (positive unanimity)
#3 (positive majority)
#4 (negative simple)
#5 (negative unanimity)
#6 (negative majority)

#TNCS-Connection (path max 200 tokens)
#if no valid path is given:
#default: /usr/local/lib/libTNCS.so
TNCS_PATH=/tmp/_master-temp/tncs/libTNCS.so
```

Listing B.2: tncs_fhh.conf

[1] Die Informationen stammen aus `INSTALL-MSW.txt`, die im Hauptverzeichnis der Distribution liegt.

Literaturverzeichnis

[Aur05] Aurand, Andreas: *LAN-Sicherheit - Schwachstellen, Angriffe und Schutzmechanismen in lokalen Netzwerken - am Beispiel von Cisco Catalyst Switches*. dpunkt.verlag, Heidelberg, 2005.

[Cis04] Cisco Systems, Inc.: *Catalyst 3550 Multilayer Switch Software Configuration Guide*, Mai 2004. Cisco IOS Release 12.1(20)EA2.

[Don04] Donaldson, Cameron: *C++ GUI Programming on Windows 2000/XP*, `http://howtos.mrcam.org/guides/development/cppguiprogramming.html` Dezember 2004. Stand: 08. August 2006.

[ERH01] Elliotte Rusty Harold, W. Scott Means: *XML In A Nutshell*. O'Reilly Verlag, Köln, 1. Auflage, 2001.

[Ghi99] Ghislanzoni, Marco: *Writing a dialog-based wxWidgets application*, `http://wxwidgets.org/docs/technote/dlgbased.htm` wxWidgets.org Tutorials, 1999. Stand: 15. August 2006.

[Has02] Hassel, Jonathan: *RADIUS*. O'Reilly, Köln, 2002.

[HP 05] HP Procurve Networking: *Access Security Guide*, Oktober 2005. ProCurve Switches E.10.02 (Series 5300xl), M.08.73 (Series 3400/6400cl).

[IBM04] IBM Corporation and others: *C/C++ Toolkit Development User Guide*, 2004.

[IEE04] IEEE Computer Society: *802.1X - Port-Based Network Access Control*, 13. Dezember 2004. IEEE Standard for Local and metropolitan area networks.

[JD03] Jürgen Dunkel, Andreas Holitschke: *Softwarearchitektur für die Praxis*. Springer, Berlin, 2003.

[mia03] miahrugger: *Raw Ethernet Packet Sending*, `http://www.codeproject.com/cs/internet/sendrawpacket.asp` Oktober 2003. Stand: 09. August 2006.

[Pet99] Petzold, Charles: *Windows-Programmierung - Das Entwicklerhandbuch zur WIN32-API*. Microsoft Press Deutschland, 1999.

[Rag05] Raghu: *Rlm eap Extensible Authentication Protocol*, `http://wiki.freeradius.org/index.php/Rlm_eap` Dezember 2005. FreeRadius Wiki Stand: 29. Mai 2006.

[Red03] Red Hat, Inc.: *Cygwin User's Guide*, 2003.

[Sun05] Sun Microsystems, Inc.: *Synchronization With Semaphores*, 2005. `http://docs.sun.com/app/docs/doc/816-5137/6mba5vpjj?a=view` Stand: 07. August 2006.

[Tan00] Tanenbaum, Andrew S.: *Computernetzwerke*. Pearson Studium, München, 3. revidierte Auflage, 2000.

[TCG05a] TCG Trusted Network Connect: *TNC Architecture for Interoperability*, 3. Mai 2005. Specification Version 1.0 Revision 4.

[TCG05b] TCG Trusted Network Connect: *TNC IF-IMC*, 3. Mai 2005. Specification Version 1.0 Revision 3.

[TCG05c] TCG Trusted Network Connect: *TNC IF-IMV*, 3. Mai 2005. Specification Version 1.0 Revision 3.

[TCG06a] TCG Trusted Network Connect: *TNC Architecture for Interoperability*, 1. Mai 2006. Specification Version 1.1 Revision 2.

[TCG06b] TCG Trusted Network Connect: *TNC IF-IMC*, 1. Mai 2006. Specification Version 1.1 Revision 5.

[TCG06c] TCG Trusted Network Connect: *TNC IF-IMV*, 1. Mai 2006. Specification Version 1.1 Revision 5.

[TCG06d] TCG Trusted Network Connect: *TNC IF-PEP: Protocol Bindings for RADIUS*, 1. Mai 2006. Specification Version 1.0 Revision 2.

[TCG06e] TCG Trusted Network Connect: *TNC IF-T: Protocol Bindings for Tunneled EAP Methods*, 1. Mai 2006. Specification Version 1.0 Revision 3.

[TCG06f] TCG Trusted Network Connect: *TNC IF-TNCCS*, 1. Mai 2006. Specification Version 1.0 Revision 2.

[Tru04] Trusted Computing Group: *TCG Specification Architecture Overview*, 28. April 2004. Specification Revision 1.2.

[Unk06] Unknown: *Modules*, `http://wiki.freeradius.org/index.php/Modules` 26. März 2006. FreeRadius Wiki Stand: 29. Mai 2006.

[Wut06] Wuttke, Daniel: *Erweiterung sicherheitsrelevanter Software für die automatische Integritätsprüfung von Endgeräten*, Masterarbeit, Fachhochschule Hannover 2006.

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

AAA	Authentication, Authorization, Accounting
API	Application Programming Interface
AR	Access Requestor
AS	Authentication Server
CDT	C/C++ Development Tooling
DDK	Driver Development Kit
DLL	Dynamic Link Library
EAP	Extensible Authentication Protocol
EAPOL	EAP over LAN
EAPOW	EAP over Wireless
GUI	Graphical User Interface
IEL	Integrity Evaluation Layer
IF	Interface
IMC	Integrity Measurement Collector
IML	Integrity Measurement Layer
IMV	Integrity Measurement Verifier
LAN	Local Area Network
MD5	Message Digest Algorithm 5
MSDN	Microsoft Developer Network
NAA	Network Access Authority
NAC	Network Admission Control
NAL	Network Access Layer
NAP	Network Access Protection
NAR	Network Access Requestor
NAS	Network Access Server
NDIS	Network Device Interface Specification
PDP	Policy Decision Point
PEAP	Protected Extensible Authentication Protocol
PEP	Policy Enforcement Point
RADIUS	Remote Authentication Dial In User Service
RFC	Request For Comment
SO	Shared Object
TCG	Trusted Computing Group
TLS	Transport Layer Security
TNC	Trusted Network Computing
TNC-SG	Trusted Network Computing Sub Group

TNCC ..... TNC Client
TNCS ..... TNC Server
TPM ..... Trusted Platform Module
TTLS ..... Tunneled Transport Layer Security
UDP ..... User Datagram Protocol
VLAN ..... Virtual Local Area Network
VPN ..... Virtual Private Network
WLAN ..... Wireless Local Area Network
XML ..... Extensible Markup Language