NENA – Ein Rahmenwerk für maßgeschneiderte Kommunikationsnetze

zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Dipl.-Inform. Denis Martin

aus Schlema

Tag der mündlichen Prüfung: 17. April 2014

Erste Gutachterin: Prof. Dr. Martina Zitterbart
Karlsruher Institut für Technologie (KIT)

Zweiter Gutachter: Prof. Dr.-Ing. Phuoc Tran-Gia
Universität Würzburg

Für meine Familie

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Telematik des Karlsruher Instituts für Technologie (KIT). Ohne die Unterstützung meiner Kollegen, meiner Freunde und meiner Familie wäre sie in dieser Form nicht möglich gewesen.

An erster Stelle gilt mein Dank Frau Prof. Dr. Martina Zitterbart, die mir durch die Anstellung an ihrem Institut nicht nur die Promotion, sondern auch die Mitarbeit an interessanten Forschungsprojekten ermöglichte. Während meiner Zeit am Institut begleitete Sie mich dabei mit hilfreichen Diskussionen und Ratschlägen. Ebenso bedanken möchte ich mich bei Herrn Prof. Dr.-Ing. Phuoc Tran-Gia, der sich trotz zahlreicher Verpflichtungen in Forschung und Lehre sofort bereit erklärte, das Korreferat für meine Promotion zu übernehmen. An dieser Stelle möchte ich mich auch für seine wiederkehrende Fachtagung *EuroView* bedanken, die mir und anderen Wissenschaftlern einen regen Austausch ermöglichte.

Bei allen Kolleginnen und Kollegen des Instituts für Telematik möchte ich mich herzlich für die sehr angenehme, gemeinsame Zeit bedanken. Besonders die kritischen Fragen und Diskussionen bei zahlreichen Gelegenheiten, aber auch insbesondere innerhalb der *Service Composition AG*, waren sehr hilfreich. Am Institut haben sich einige Freundschaften entwickelt, die ich nicht mehr missen möchte.

Bedanken möchte ich mich zudem bei den Studierenden, die durch ihre Abschlussarbeiten, Praktika und HiWi-Tätigkeiten die Entwicklung von NENA mit vorangetrieben haben. Ebenso gilt mein Dank Partnern aus dem EU FP7 Projekt 4WARD und dem BMBF Projekt G-Lab für die gute Zusammenarbeit und für die zahlreichen, wichtigen Diskussionen.

Nicht zuletzt möchte ich besonders meiner Familie und meinen Freunden danken: meinen Eltern, weil sie mir u. a. durch eine für sie sehr wichtige Entscheidung vieles erleichtert haben; meinen Geschwistern und meinen Freunden für die Ablenkung und Motivation außerhalb des Instituts. Insbesondere gilt mein Dank einem guten Freund, der mich durch Schule und Studium begleitet hat, inzwischen aber nicht mehr unter uns ist.

Denis Martin, Karlsruhe im Juni 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Das Internet hat in den letzten Jahren einen immer wichtigeren Stellenwert in der Gesellschaft errungen. Durch einfache und intuitive Bedienbarkeit und Omnipräsenz auch unterwegs ist es heute fester Bestandteil im Alltag. Private Termine mit Freunden werden über Facebook verabredet, die neusten Erlebnisse über Twitter verbreitet und Wissenslücken bei Stammtischdiskussionen über Wikipedia geschlossen. Aktuelle Endgeräte wie Smartphones, Tablet PCs und Media Centers für den Fernseher zu Hause vereinfachen den Zugriff und den Konsum der über das Internet angebotenen Inhalte enorm. Durch den global erreichbaren Konsumentenkreis ergeben sich für Firmen attraktive Möglichkeiten, selbst Nischenangebote gewinnbringend über das Internet abzusetzen. Neben klassischen Online-Versanddiensten gibt es aber auch immer mehr Dienste, die ausschließlich über das Internet erbracht werden. Dazu gehören z. B. Internet-Telefonie, Internet-TV und Radio, Online-Spiele und Online-Video-Verleihdienste. Während die Nutzung einiger dieser Dienste bereits früh möglich war, hat deren Verbreitung erst zugenommen, als die Bedienbarkeit eine für den Massenmarkt taugliche Reife erlangt hatte.

Aber nicht nur der Zugriff auf diese Dienstangebote wird einfacher: Mit dem Web-Browser steht eine mächtige, geräteunabhängige Plattform für die Entwicklung unterschiedlichster Web-Anwendungen zur Verfügung. Zudem erlauben einfache Programmierschnittstellen in den Entwicklungsumgebungen der mobilen Endgeräte den Zugriff auf komplexe Funktionen wie die Standortbestimmung über Mobilfunkmasten und GPS. Auf Server-Seite ermöglichen sogenannte „Cloud“-Dienste, die mit der Nutzerzahl skalieren,

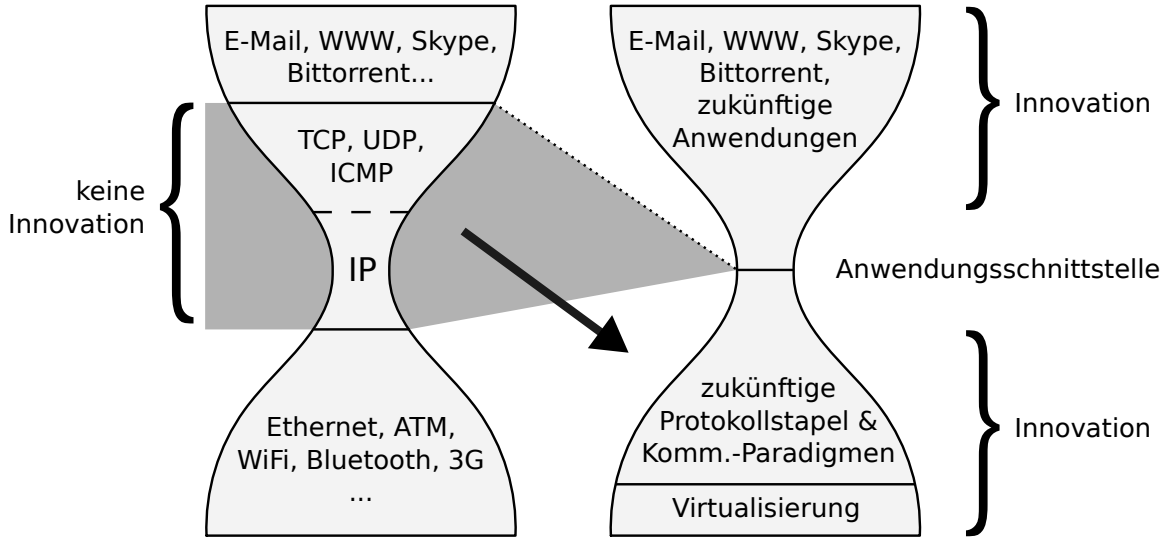


Abbildung 1.1 Sanduhrenmodelle der Innovationsmöglichkeiten heute (links) und durch eine geeignete Entkopplung von Anwendungen und Protokollen (rechts)

kleinen Unternehmen neue, innovative Dienste ohne teure Infrastruktur-Anschaffungen ins Netz zu stellen. Auch die Übertragungstechnologien entwickeln sich stets weiter. Höhere Bandbreiten und geringere Latenzen nicht nur im Festnetz sondern gerade auch in der drahtlosen Übertragung haben letztendlich dazu beigetragen, dass Dienste zu jeder Zeit und jeden Orts genutzt werden können.

Obwohl sich Anwendungen und Übertragungstechnologien stetig weiterentwickeln, haben sich die Kernprotokolle für Vermittlung und Transport im Internet in den letzten Jahrzehnten kaum verändert (Abbildung 1.1 links). Dies als Konstante war zwar zum einen mit für den Erfolg des Internet maßgeblich, stellt aber zum anderen immer mehr Hürden für seine weitere Entwicklung auf, da die Leistungsfähigkeit dieser Protokolle in heutigen Umgebungen immer mehr an ihre Grenzen stößt. Diese Protokolle zu ändern ist jedoch ein langwieriger und aufwändiger Prozess, was sich bei der Einführung von IPv6 gezeigt hat: Obwohl die Notwendigkeit zur Einführung des Internet Protokolls der Version 6 (IPv6) u. a. aufgrund der Adressknappheit gegeben war, zögerten viele Dienstanbieter mit dessen Nutzung, da sie technische Probleme befürchteten. Viele Anwendungen und Dienste mussten und müssen schließlich aufgrund dieser Änderung angepasst werden, was zusätzliche Fehlerquellen mit sich bringt und damit die Dienstverfügbarkeit potentiell einschränkt.

Weitere Änderungen zeichnen sich jedoch bereits ab: Große Dienstanbieter im Internet wie Google haben erkannt, dass Änderungen an den Internet-Protokollen vorteilhaft sind, um die Ressourcennutzung zu optimieren und

die Antwortzeiten zu verkürzen (SPDY [BePe12], TCP-FastOpen [CCRJ11], QUIC [Rosk13]). Aber auch hier müssen Anwendungen und Dienste angepasst werden, um von diesen Änderungen profitieren zu können. Neben diesen „kleinen“ Änderungen existieren zudem viele sog. „Clean-Slate“-Ansätze, die einen kompletten Neuentwurf dieser Kernprotokolle anstreben und teilweise komplett neue Kommunikationsparadigmen vorschlagen. Diese versprechen eine bessere Ressourcennutzung oder Performance für bestimmte Anwendungsfälle. Beispiele hierfür sind die inhaltsbasierte (*content-centric*) und die dienstbasierte (*service-centric*) Kommunikation. Neue Ansätze für spezielle Problemstellungen existieren also, haben aber mangels Unterstützung im Netz und Verbreitung bei den Anwendungsentwicklern ein Akzeptanzproblem.

Um langfristig flexibler bei der Einführung neuer Protokolle zu sein, müssen neue Abstraktionen gewählt werden. Durch geeignete Schnittstellen können die Abhängigkeiten zwischen Anwendungen und Protokollen und zwischen Protokollen und Hardware-Infrastruktur reduziert werden (Abbildung 1.1 rechts). Dadurch ergibt sich eine deutliche Steigerung der Innovationsfähigkeit in den einzelnen Bereichen.

1.1 Annahmen und Zielsetzung

Da sich viele neue Ansätze nur auf spezielle Problemstellungen beschränken, in diesen Fällen aber eine effizientere Ressourcennutzung versprechen, liegt es nahe, dass es keine „One-Size-Fits-All“-Lösung gibt, die in allen Fällen gleich gut funktioniert. Da diese Ansätze allerdings oft inkompatibel zueinander sind, können sie nicht immer effizient im selben Netz betrieben werden. Mit Netzwerkvirtualisierung und Overlay-Netzen existieren Ansätze und Technologien, die es erlauben verschiedene Netze parallel auf einer gemeinsamen Infrastruktur zu betreiben. Overlay-Netze sind dabei bereits heute im Einsatz. Außerdem sind unter dem Stichwort Software-Defined Networking (SDN) aktuell Bestrebungen im Gange, die es Dienstanbietern erlauben, das Netzwerkverhalten in ihren eigenen Netze dem eigenen Bedarf anzupassen und auch so die Ressourcennutzung zu optimieren.

Die vorliegende Arbeit geht daher von einem Szenario aus, in dem Netze von Anwendungs-Dienstanbietern direkt bis zum Endbenutzer bereitgestellt werden (Abbildung 1.2). Durch Virtualisierungs- und Overlay-Techniken können innerhalb dieser Netze angepasste Protokolle verwendet werden, die vom jeweiligen Dienstanbieter ausgewählt wurden oder eigens für diese Netze entworfen worden sind. Allerdings sind drei weitere Voraussetzungen wesentlich für dieses Szenario, welche zugleich die Zielsetzungen dieser Arbeit beschreiben:

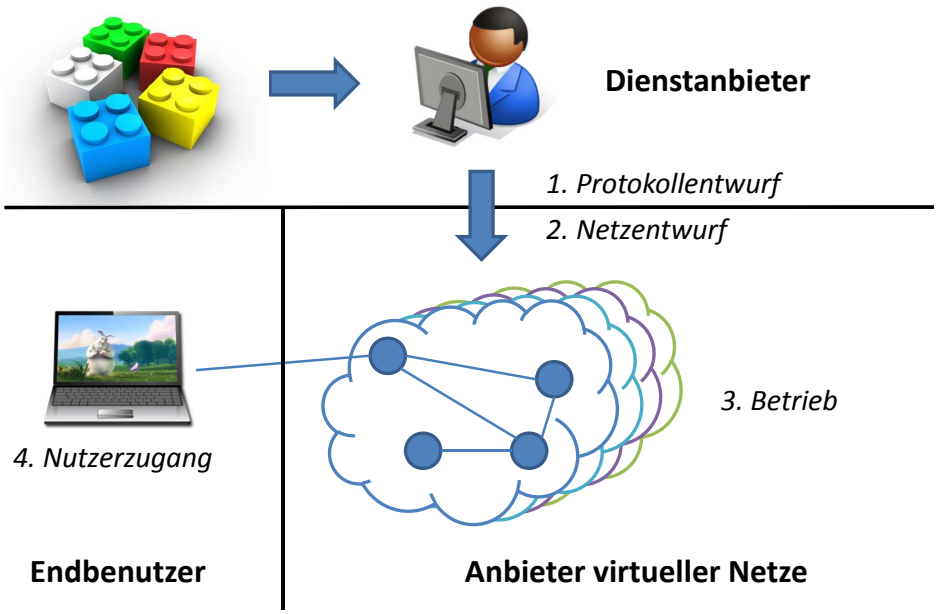


Abbildung 1.2 Dienstanbieterspezifische Netze: Der Dienstanbieter entwirft die für ihn passenden Protokolle auf Basis von Protokollkomponenten (1) sowie das dazugehörige Netz (2). Dieses betreibt er als virtuelles oder Overlay-Netz (3). Endbenutzer verbinden sich direkt mit dem Dienstanbieternetz (4).

1. Unterstützung für den Entwurf dienstanbieterspezifischer Protokolle

   Da dieser Ansatz die dienstanbieterspezifische Entwicklung von Kommunikationsprotokollen in den Mittelpunkt stellt, ist es wichtig die dazugehörige Entwicklung dieser Protokolle zu vereinfachen. Die Ziele dieser Vereinfachung sind dabei eine verkürzte Entwicklungszeit und eine Reduzierung von Fehlerquellen.

2. Entkopplung von Anwendungen und Kommunikationsprotokollen

   Wesentlich am Erfolg der Internet-Protokollfamilie ist die Bereitstellung von stabilen Abstraktionen sowohl nach oben, zu den Anwendungen, als auch nach unten, zur Übertragungstechnologie. Gerade die Schnittstelle zu den Anwendungen setzt allerdings Netzwerkwissen beim Anwendungsentwickler voraus, da er neben der Namensauflösung und Adressauswahl auch die Protokollauswahl vornehmen muss. Dies verhindert zum einen, dass sich ohne eine Anpassung der Anwendung die Adressierungsart ändern kann, und zum anderen, dass alternative Protokolle verwendet werden können, die vom Anwendungsentwickler nicht vorgesehen waren.

3. Nebenläufiger Betrieb mehrerer Protokollstapel für verschiedene Netze

   Mit der Entkopplung von Anwendungen und Protokollen ist es möglich, verschiedene Kommunikationsprotokolle transparent zur Anwen-

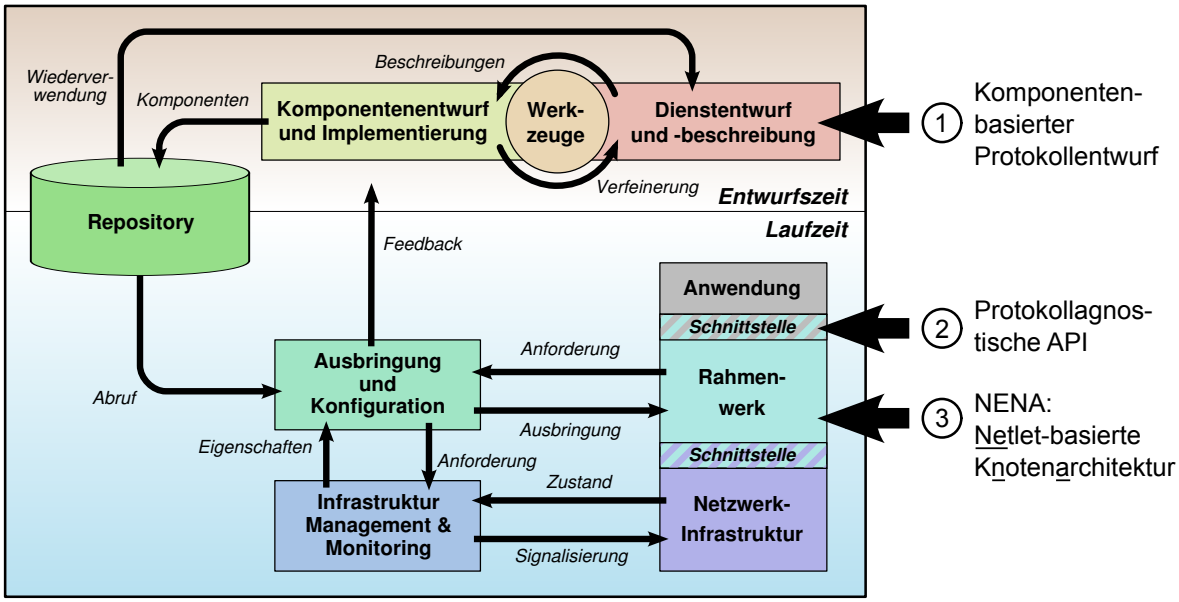


Abbildung 1.3 Lebenszyklus von Protokollkomponenten [VMRB+09]: Entwurfs- und Laufzeit mit den drei Beiträgen dieser Arbeit (1) komponentenbasierter Protokollentwurf, (2) protokollagnostische Anwendungsschnittstelle und (3) NENA

dung zu verwenden. So können bspw. auch neue Protokolle oder Protokollmodifikationen verwendet werden, die zum Entwicklungszeitpunkt der Anwendung noch nicht bekannt waren. Diese können dann auf den speziellen Anwendungsfall zugeschnitten sein und auf die Bedürfnisse des Dienstanbieters optimiert sein. Dies bedeutet allerdings, dass eine Vielzahl von Protokollstapeln und Netzen gleichzeitig im Einsatz sein können, der Zugriff durch den Benutzer bzw. durch die Anwendungen jedoch einfach gehalten werden muss.

1.2 Beiträge und Gliederung

Basis dieser Arbeit ist der in Abb. 1.3 dargestellte Lebenszyklus für dienstanbieterspezifische Protokolle [VMRB+09]. Ein Lebenszyklus (Life-Cycle) beinhaltet dabei die Zeit zwischen Konzeption einer Komponente und deren Außerdienststellen [IEE90]. Der hier vorgestellte Lebenszyklus unterteilt sich in Entwurfszeit und Laufzeit, wobei ein sogenanntes Repository als Schnittstelle dazwischen dient. Im Repository werden entworfene Protokolle und Protokollkomponenten abgelegt und von dort wieder abgerufen – sei es zum Entwurf neuer Protokolle oder für den Betrieb zur Laufzeit.

Der in der Abbildung oben dargestellte Entwicklungsprozess zur Entwurfszeit besteht aus den beiden Hauptaktivitäten *Dienstentwurf* und *Komponentenentwurf*. Der Dienstentwurf beschreibt dabei die Zielsetzungen des Dienst-

anbieters sowie sonstige Anforderungen, beispielsweise bzgl. der Integration in bestehende Umgebungen. Der Komponentenentwurf hingegen beschreibt den Entwurf einzelner Protokolle oder Protokollkomponenten, die aus dem Dienstentwurf abgeleitet werden. Werkzeuge können hierbei zur Unterstützung der Entwurfsaktivitäten beitragen.

Zur Laufzeit bietet Netzwerkvirtualisierung [ChBo10] Möglichkeiten zur Isolation der Netze voneinander. Außerdem wird so die Hardware-Infrastruktur von den Netzwerkarchitekturen der Dienstanbieter entkoppelt, was eine unabhängige Evolution beider ermöglicht. Neben Überwachung und Steuerung der Hardware-Infrastruktur müssen vom Infrastrukturbetreiber auch Schnittstellen zur Konfiguration der dienstanbieterspezifischen Netze bereitgestellt werden. Diese Schnittstellen erlauben dem Dienstanbieter, der hier als Betreiber seines eigenen Netzes agiert, seine virtuellen Netzwerkknoten zu konfigurieren und den Zugang von Endbenutzern zu seinem Netz zu kontrollieren.

Während durch Virtualisierung angepasste Protokolle im jeweiligen Dienstanbieternetz verwendet werden können, sind auf den Endsystemen der Benutzer zusätzliche Anpassungen erforderlich. Transport- und Netzwerkprotokolle sind heute als Teil des Betriebssystems realisiert. Damit Dienstanbieter ihre Protokolle bis zum Endbenutzer hin anpassen und Anwendungen die angepassten Protokolle nutzen können, muss die Integration der Protokolle auf den Endsystemen flexibilisiert werden. Mit Hilfe einer geeigneten Anwendungsschnittstelle und eines Rahmenwerks, das den Bezug und den Betrieb dieser angepassten Protokolle ermöglicht, kann dies erreicht werden.

Für diesen Lebenszyklus werden in der vorliegenden Arbeit drei wesentliche Beiträge geleistet:

1. Komponentenbasierter Protokollentwurf

   Für viele Kommunikationsaufgaben existieren bereits Lösungen in Form von Protokollen oder einzelnen Protokollmechanismen. Durch eine geeignete Kapselung kann der Dienstanbieter aus einem Baukastensystem die für ihn passenden Protokollkomponenten heraussuchen und kombinieren. Ein strukturierter Ansatz auf Basis sog. Protokollschablonen und die Eingliederung in den Entwicklungsprozess zur Entwurfszeit werden in Kapitel 4 vorgestellt.

2. Protokollagnostische Anwendungsschnittstelle

   Eine protokollagnostische Anwendungsschnittstelle erlaubt die Entkopplung von Anwendungen und Kommunikationsprotokollen, indem die Anwendung lediglich ihren Kommunikationswunsch abstrakt formuliert und die passenden Protokolle basierend auf den übergebenen

Anforderungen ausgewählt werden. Dadurch ergibt sich eine Verschiebung der Kommunikationsabstraktionen für die Anwendung, wie sie im Sanduhrenmodell in Abbildung 1.1 dargestellt ist. Um zukünftige Änderungen an dieser Schnittstelle gering zu halten, dabei aber auch flexibel bei der Einführung neuer Protokolle zu bleiben, wird der Anwendungsentwickler vollständig von der Adressauflösung, Adressauswahl und Protokollauswahl entlastet. Eine Anwendungsschnittstelle, die die notwendigen Abstraktionen dazu liefert, wird in Kapitel 5 vorgestellt.

3. NENA – Netlet-basierte Knotenarchitektur

   Ein Rahmenwerk für angepasste Kommunikationsprotokolle wird mit der Netlet-basierten Knotenarchitektur NENA (Netlet-based Node Architecture) in Kapitel 6 vorgestellt. Diese erlaubt das dynamische Laden und den nebenläufigen Betrieb von Protokollen, gekapselt als sogenannte *Netlets*. Außerdem übernimmt sie die notwendige Protokollauswahl basierend auf den Anwendungsanforderungen wie sie von der API übergeben werden.

Bevor diese Beiträge beschrieben werden, geht Kapitel 2 auf die notwendigen Grundlagen ein. In Kapitel 3 werden relevante Arbeiten untersucht und beschrieben, die evolutionäre oder auch revolutionäre Vorschläge für ein zukünftiges Internet anbieten. In Kapitel 7 werden die in dieser Arbeit vorgeschlagen Konzepte systematisch mit Hilfe von Fallstudien untersucht. Dabei werden möglichst unterschiedliche Netzwerkprotokolle und neue Paradigmen betrachtet. Kapitel 8 fasst die wesentlichen Ergebnisse zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

2. Grundlagen

In diesem Kapitel werden die für diese Arbeit notwendigen Grundlagen der heutigen Internet-Protokollarchitektur beschrieben und anschließend diskutiert, welche Probleme damit heute bestehen. Außerdem werden Begriffe definiert, die im weiteren Verlauf der Arbeit verwendet werden (siehe auch Glossar im Anhang).

Beschrieben wird zunächst das Dienstemodell heutiger Computer-Netzwerke am Beispiel des Internet. In Anlehnung an [KuRo13] wird dazu ein sog. Top-Down-Ansatz verwendet, d. h., die Beschreibung beginnt zunächst aus Sicht der Anwendung, um dann durch die einzelnen Schichten des Internet-Modells hinabzusteigen und dort auf einzelne Protokolle näher einzugehen. Hierbei liegt der Fokus insbesondere auf dem Transportprotokoll TCP, da dessen Funktionalität in Abschnitt 4.3.2 auf Basis von Protokollkomponenten nachgebildet wird. Am Ende dieses Kapitels folgt zudem eine Beschreibung der groben Architektur des heutigen Internet. Im nächsten Kapitel werden dazu alternative Ansätze diskutiert.

2.1 Kommunikationsdienste

Eine verteilte Anwendung besteht aus mehreren Prozessen, welche auf unterschiedlichen Endsystemen ausgeführt werden und untereinander kommunizieren. Die Prozesse der verteilten Anwendung tauschen dabei Informationen über ein Netzwerk aus. Sie nutzen dazu einen *Kommunikationsdienst*, der über *Kommunikationsendpunkte* von einem *Kommunikationssystem* bereitgestellt wird. Ein Kommunikationsendpunkt kann dabei von der Anwendung über eine *Anwendungsschnittstelle* erstellt werden. Dies ist in Abbildung 2.1

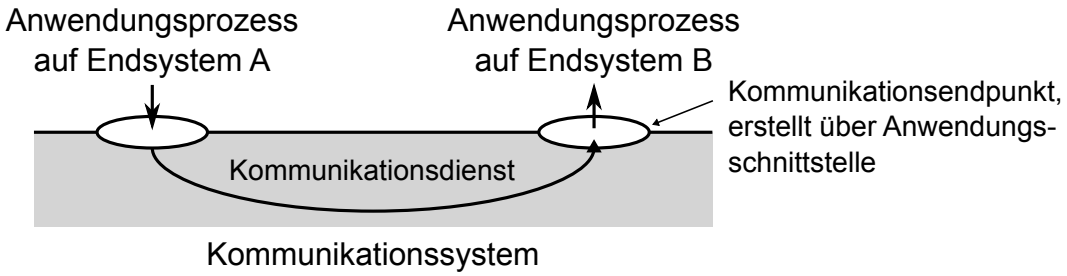


Abbildung 2.1 Nutzung eines Kommunikationsdienstes über eine Anwendungsschnittstelle durch eine verteilte Anwendung

skizziert. Im Rahmen dieser Arbeit werden die Prozesse der verteilten Anwendung allgemein als Anwendungen bezeichnet, weil die Prozesse sich in vielen Fällen je nach Rolle deutlich unterscheiden: Bei einer Client/Server-Architektur bspw. werden oft Client- und Server-Anwendungen unabhängig voneinander entwickelt. Sie nutzen dabei jedoch ein gemeinsames *Anwendungsprotokoll*, auf welches sich die Entwickler der Anwendungen im Vorfeld geeinigt haben.

Im Kommunikationssystem wird der Kommunikationsdienst über mehrere Protokolle und Netzwerkkomponenten (Geräte) hinweg realisiert. Er besteht aus mehreren *Schichten*, wobei jede Schicht einen eigenen Dienst bereitstellt (Abbildung 2.2). Jede Schicht enthält auf jedem Gerät dazu eine *Protokollinstanz*, die mit anderen Protokollinstanzen der selben Schicht auf anderen Geräten kommunizieren kann. Die Kommunikation zwischen Protokollinstanzen der selben Schicht wird als *horizontale* Kommunikation bezeichnet, die Kommunikation zwischen den Schichten als *vertikale* Kommunikation. Die vertikale Kommunikation erfolgt dabei über *Dienstzugangspunkte* (Service Access Points, SAPs) zwischen den Schichten, und jede Schicht erfüllt einen Dienst, der durch die darüberliegende Schicht in Anspruch genommen wird. Da die Protokollinstanzen auf einem Gerät durch die Schichten übereinander „gestapelt“ sind, wird die Gesamtheit der Protokollinstanzen auf einem Gerät als *Protokollstapel* bezeichnet.

Zwischen den Schichten werden Informationen wie folgt ausgetauscht: Eine zu versendende Dateneinheit wird als Dienstdateneinheit (Service Data Unit, SDU) an die darunterliegende Schicht über den SAP übergeben (vertikale Kommunikation). Diese Schicht behandelt die SDU als Protokolldateneinheit (Protocol Data Unit, PDU) und fügt Protokollkontrollinformationen (Protocol Control Information, PCI) an diese PDU an. Die PCI enthalten dabei Kontrollinformationen für die Protokollinstanz der selben Schicht auf einem anderen Gerät und dienen damit der horizontalen Kommunikation. Sie werden meist

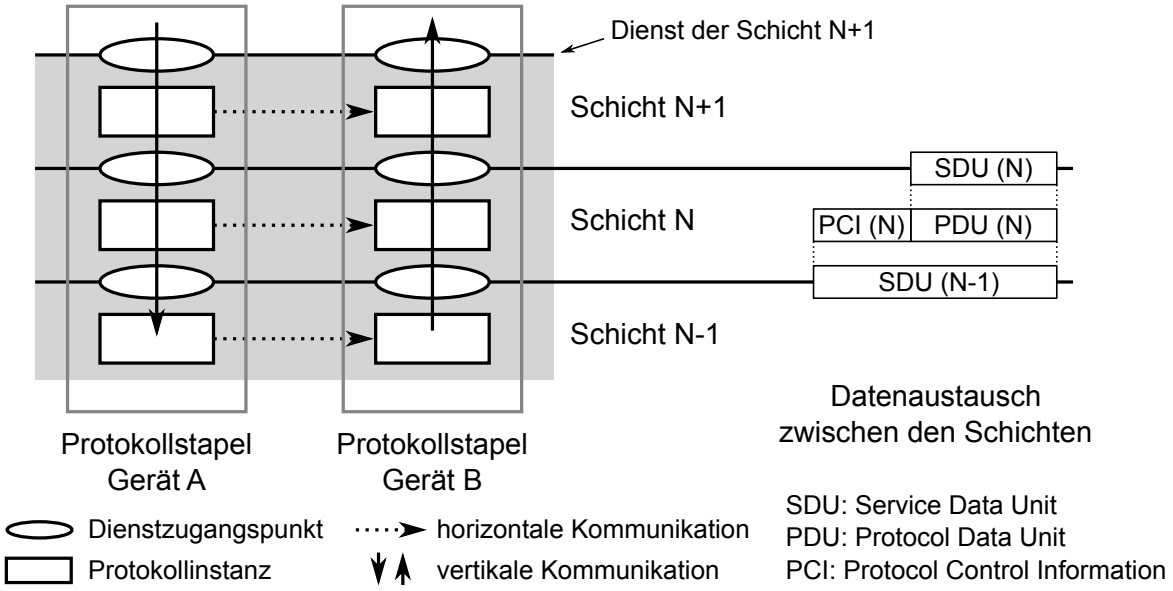


Abbildung 2.2 Schichten und ihre Dienstzugangspunkte

als Paketkopf vor die PDU gestellt. PCI und PDU bilden dann die neue SDU für die darunterliegende Schicht.

In jeder Schicht können unterschiedliche *Kommunikationsparadigmen* verwendet werden. Ein Kommunikationsparadigma beschreibt grundlegende Elemente einer Kommunikation, d. h. die Subjekte, Objekte und Prädikate. Bei einer klassischen Ende-zu-Ende-Kommunikation zwischen Anwendungen sind die grundlegenden Elemente die Kommunikationsendpunkte der beteiligten Anwendungen, die von diesen geöffnet und adressiert werden. Darüber werden Daten als Strom oder Datagramme gesendet oder empfangen. Auf Vermittlungsschicht sind diese Elemente die Schnittstellen der Geräte, die adressiert werden und über die Datagramme weitergeleitet werden.

Die Aufteilung des Kommunikationsdienstes auf verschiedene Schichten bietet eine Möglichkeit zur Strukturierung und führt zu Modularität. Eine solche Aufteilung der Schichten für verschiedenen Aufgaben ist in Abbildung 2.3 mit dem Internet-Referenzmodell dargestellt. Hier sind zwei Endsysteme und ein Router als Zwischensystem abgebildet, welche die an der Kommunikation beteiligten Geräte des Kommunikationssystems darstellen. Die Aufgaben der Schichten sind:

- Anwendungsschicht

  In der Anwendungsschicht befinden sich die tatsächlichen Anwendungen und ihre Anwendungsprotokolle. Im ebenfalls in der Literatur verbreiteten ISO/OSI-Referenzmodell werden hier zusätzlich eine Darstellungs- und eine Sitzungsschicht unterhalb der Anwendungsschicht eingeführt. Das Internet-Modell vereint die Aufgaben dieser Schichten je-

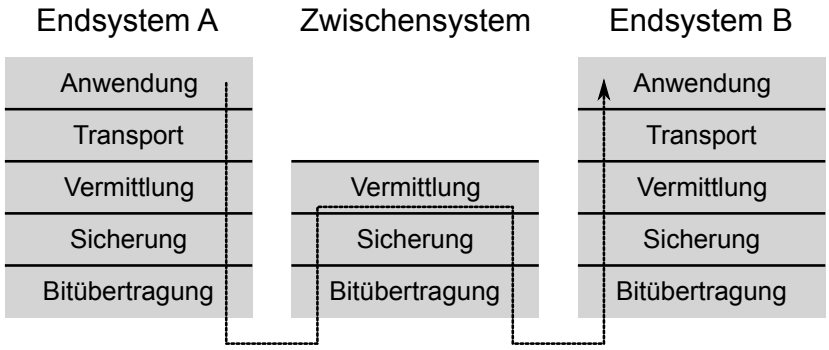


Abbildung 2.3 Schichtenprinzip und der tatsächliche Verlauf der Nachricht durch die Schichten

doch in der Anwendungsschicht, was bei den meisten heutigen Anwendung auch der Realität entspricht.

- Transportschicht

  Die Protokolle der Transportschicht sind für die Ende-zu-Ende-Übertragung der Anwendungsnachrichten zwischen den Endsystemen über das Netzwerk zuständig. Die Art des Transportdienstes wird dabei vom verwendeten Protokoll festgelegt. Dieses wird wiederum von der Anwendung ausgewählt.

- Vermittlungsschicht

  Die Vermittlungsschicht ist für die Übertragung der Nachrichten über mehrere Übertragungsabschnitte und Systeme hinweg entlang eines Pfades zuständig. Neben der Festlegung der notwendigen Adressierung der Systeme gehören zu den Aufgaben dieser Schicht auch die Weiterleitung der Pakete.

- Sicherungsschicht

  In der Sicherungsschicht befinden sich Protokolle, die für die Übertragung der Nachrichten bzw. Dateneinheiten als Pakete über einen Übertragungsabschnitt zwischen zwei Geräten verantwortlich sind. Beispielsweise wird hier auch der Zugriff auf das Medium gesteuert.

- Bitübertragungsschicht

  Während die Sicherungsschicht für die Übertragung kompletter Pakete verantwortlich ist, realisiert diese Schicht die Übertragung einzelner Bits über das jeweilige Medium.

Durch diese Strukturierung soll es ermöglicht werden, die Protokolle einzelner Schichten zu modifizieren, ohne dass andere Schichten dazu angepasst

werden müssen. In der Realität ist dies aber nicht immer gegeben. Zudem bringt diese Vorgabe fester Schichten einige weitere Probleme mit sich, die in Abschnitt 2.4 beschrieben werden.

2.2 Protokolle im Internet

In diesem Abschnitt wird eine Auswahl heutiger Protokolle im Internet in den oberen drei Schichten vorgestellt: Anwendungsprotokolle, Transportprotokolle und Vermittlungsprotokolle. Die hier als Beispiele verwendeten Protokolle werden in späteren Kapiteln dieser Arbeit wichtig.

2.2.1 Anwendungsprotokolle

Damit eine Anwendung auf einem Endsystem mit einer Anwendung auf einem anderen Endsystem kommunizieren kann, muss die entfernte Anwendung adressierbar sein. Unabhängig vom verwendeten Anwendungsprotokoll ist dabei in heutigen TCP/IP-Netzen die Angabe der Ziel-IP-Adresse, des Transportprotokolls und der Portnummer notwendig. Anwendungsprotokolle können dabei Vorgaben treffen, die die Verwendung vereinfachen:

- Mit dem Domain-Name-System (DNS) existiert eine anwendungsprotokollübergreifende Möglichkeit, global eindeutige Namen auf IP-Adressen abzubilden.
- Anwendungsprotokolle können vorgeben, mit welchen Transportprotokollen sie zu verwenden sind. Zusätzlich können sie Portnummern registrieren, die ein Abbildung eines *Dienstnamens* auf Vorgabewerte ermöglicht [CETW+11].

Mit Hilfe des letzten Punktes kann so bspw. der Dienstname „http“ auf das Protokoll TCP und die Portnummer 80 statisch abgebildet werden. Da eine statische Abbildung nicht immer sinnvoll ist, können über sogenannte DNS-Service-Records [GuVE00] diese Informationen auch im DNS hinterlegt werden und so dynamisch bei Gebrauch durch die Anwendung abgerufen werden.

Über ein Anwendungsprotokoll wird mit einem *Anwendungsdienst* kommuniziert. Der Anwendungsdienst kann beispielsweise die Bereitstellung von Inhalten als Datei- oder Web-Server sein. Er definiert dabei mögliche Primitiven, die durch das Anwendungsprotokoll über das Netzwerk übertragen werden müssen. Neben Übertragung der Primitiven muss das Anwendungsprotokoll auch sicherstellen, dass die Repräsentation der Daten beim Gegenüber verständlich ist (z. B. Zeichenkodierung oder Dateiformat).

Ein solches Anwendungsprotokoll ist HTTP [FGMF+99]. Der Anwendungsdienst, der von einem Client angesprochen wird, ist dabei ein Dienst, der

Web-Inhalte bereitstellt. Das durch HTTP bereitgestellte Kommunikationsparadigma kann daher als inhaltsbasiert bezeichnet werden:

- Das Subjekt der Kommunikation ist eine Ressource, die über eine URL[1] (Uniform Resource Locator) identifiziert wird.
- Die Objekte der Kommunikation sind die Daten, also z. B. das HTML-Dokument.
- Die Prädikate werden durch die HTTP-Primitiven formuliert, z. B. GET, PUT, POST und DELETE.

Da die Anwendung das Anwendungsprotokoll implementiert, muss sie allerdings dieses inhaltsbasierte Paradigma auf das von der Transportschicht bereitgestellte Ende-zu-Ende-Paradigma abbilden. Dies geschieht dadurch, dass der Server-Name, der Teil der URL ist, über DNS auf eine IP-Adresse abgebildet wird. Anschließend wird zu dieser IP-Adresse eine TCP-Verbindung an die Portnummer 80 (oder einer anderen, als Teil der URL formulierten Portnummer) hergestellt. Der verbleibende Teil der URL enthält dann nur noch den lokalen Pfad des Inhalts auf dem Server.

2.2.2 Transportprotokolle

Zur Übertragung der Anwendungsdaten – oder vielmehr Anwendungsprotokolldaten – wählt die Anwendung ein Transportprotokoll aus, welches einen für die Anwendung (bzw. für das Anwendungsprotokoll) geeigneten Dienst bereitstellt. Die zwei prominentesten Vertreter sind TCP und UDP, welche jeweils folgende Dienste bereitstellen:

- TCP stellt einen verbindungsorientierten, zuverlässigen Dienst bereit. Vor dem Austausch der Anwendungsdaten wird also zunächst eine Verbindung zwischen den Teilnehmern hergestellt, die sicherstellt, dass beide zum Austausch der Anwendungsdaten bereit sind. Die Anwendungsdaten werden dann zuverlässig zugestellt, d. h. der Sender kann davon ausgehen, dass die Anwendungsdaten korrekt, in der gleichen Reihenfolge, ohne Verluste und ohne Duplikate vom Empfänger ausgelesen werden können.
- UDP bietet dagegen einen einfachen Dienst, der lediglich sicherstellt, dass empfangene Anwendungsdaten korrekt sind. Hier kann es also vorkommen, dass der Empfänger gar nicht erreichbar ist, dass Daten verloren gehen, dupliziert werden oder nicht in der selben Reihenfolge beim Empfänger eintreffen.

[1] Da die femininen Formen von URL und URI im allgemeinen Sprachgebrauch häufig vertreten sind, werden diese für die Abkürzung verwendet, auch wenn der *Locator* bzw. *Identifier* die maskulinen Formen nahelegen.

2.2.2.1 Mechanismen

Um die zuvor erwähnten Transportdienste den Anwendungen anbieten zu können, sind verschiedene Aufgaben von den Transportprotokollen zu erfüllen. Während UDP lediglich die Prüfung der Korrektheit der empfangenen Anwendungsdaten und das Multiplexing und Demultiplexing verschiedener Anwendungen ermöglicht, muss TCP zur Bereitstellung eines zuverlässigen Dienstes noch folgendes leisten:

- Paketverlust-, Duplikat- und Phantomdateneinheiterkennung[1]
- Sicherstellung der Paketreihenfolge
- Übertragungswiederholung
- Flusskontrolle
- Staukontrolle

Um diese Aufgaben zu erfüllen, werden verschiedene Mechanismen eingesetzt, die im Folgenden erläutert werden. UDP nutzt davon lediglich Portnummern und die Prüfsumme. TCP setzt alle hier erwähnten Mechanismen ein.

Portnummern

Für das Multiplexing und Demultiplexing verschiedener Kommunikationsbeziehungen zwischen Anwendungen setzen UDP und TCP Portnummern ein. Dabei erhält jeder Kommunikationsendpunkt eine Portnummer und ist zusammen mit der IP-Adresse und dem Protokoll eindeutig identifizierbar. Eine Kommunikationsbeziehung lässt sich so eindeutig durch das 5-Tupel

(Quelladresse, Quellport, Protokoll, Zieladresse, Zielport)

identifizieren.

Prüfsumme

Zur Überprüfung der Korrektheit der empfangenen Daten setzen UDP und TCP jeweils Prüfsummen ein, mit denen Bitfehler erkannt werden können.

Sequenznummern

Einzelne Nachrichten bzw. Pakete müssen zur Paketverlust- und Duplikaterkennung identifizierbar sein. TCP nutzt dabei einen fortlaufenden Zähler, der sich mit jedem versendeten Byte erhöht, als Sequenznummer. Diese wird auch für verschiedene weitere Mechanismen genutzt.

[1] Als Phantomdateneinheit wird ein Paket bezeichnet, dass zwar an den aktuellen Kommunikationsendpunkt adressiert ist, aber nicht zur laufenden Verbindung gehört (und bspw. von einer alten Verbindung stammt).

Bestätigungen (Quittungen)

Bestätigungen (acknowledgments, ACKs) werden von der Empfängerinstanz des Transportprotokolls zurück an den Sender versendet, um den erfolgreichen Empfang eines Paketes zu quittieren.

Sendewiederholung

Bleibt der erwartete Empfang einer Bestätigung für eine gewisse Zeit aus, werden die unbestätigten Nachrichten erneut übertragen. Dieser Fall wird über einen Zeitgeber realisiert, dessen Dauer auf Schätzungen der aktuellen Paketumlaufzeit (Round-Trip Time, RTT) basiert. Neben der zeitgesteuerten Sendewiederholung kann ein notwendiges erneutes Versenden auch durch den Empfang duplizierter ACKs erkannt werden (vgl. folgenden Abschnitt).

Flusskontrolle

Die empfangende Anwendung muss vor einer Überlastung durch den Sender geschützt werden. Dazu wird ein Empfangsfenster bestimmt, dessen Größe dem aktuell freien Empfangspuffer entspricht. Der Empfangspuffer ist zwischen Anwendung und Transportprotokoll platziert und wird von der empfangenden Anwendung ausgelesen. Ist dieser Puffer voll, können keine weitere Daten mehr empfangen werden und das Empfangsfenster wird auf 0 gesetzt. Die aktuelle Größe des Empfangsfenster wird über die Bestätigungen dem Sender mitgeteilt und dort auch als Flussfenster bezeichnet.

Staukontrolle

Während die Flusskontrolle die empfangende Anwendung vor Überlastung schützt, reagiert die Staukontrolle auf Überlastungen im Netz. Wird eine solche erkannt, wird das Staukontrollfenster reduziert und bei nachlassender Last wieder erhöht. Hierzu existieren verschiedene Algorithmen für TCP, von denen die Reno-Variante im folgenden Abschnitt näher beschrieben wird.

Sendekredit

Durch das Flussfenster und das Staukontrollfenster wird die Anzahl der sich in der Übertragung befindlichen Pakete (Flight Size) beschränkt. Der Sendekredit (auch Sendefenster genannt) wird dabei durch das Minimum von Staukontrollfenster und Flussfenster bestimmt. Um eine Deadlock-Situation bei einem Flussfenster von 0 zu vermeiden (d. h. der Empfangspuffer ist temporär vollständig gefüllt), darf der Sender jedoch immer mindestens ein Byte versenden. Dadurch muss der Empfänger eine Bestätigung mit der aktuellen Größe seines Empfangsfenster versenden.

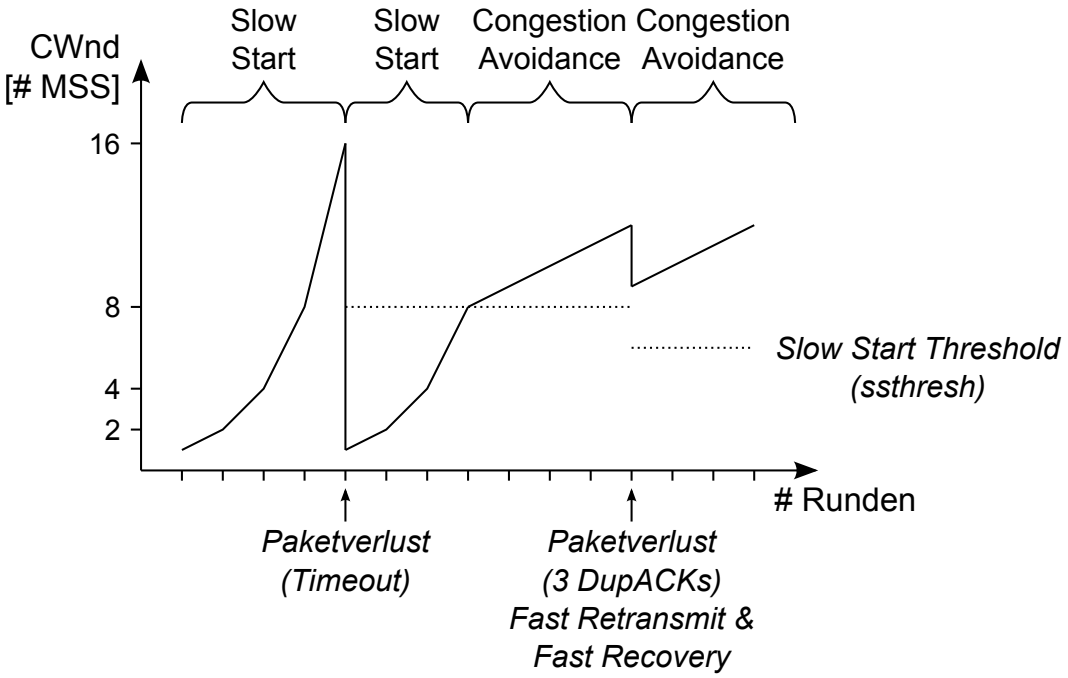


Abbildung 2.4 Beispiel zum Verhalten des Staukontrollfensters CWnd (Congestion Window) in Anzahl MSS (Maximum Segment Size) bei TCP-Reno

2.2.2.2 Staukontrolle bei TCP

Die TCP-Staukontrolle in der TCP-Reno-Variante [AlPB09] besteht aus drei Phasen, die beispielhaft am Verlauf des Staukontrollfensters in Abbildung 2.4 illustriert sind. Die Phasenübergänge werden dabei durch verschiedene Ereignisse ausgelöst, bei denen die Erkennung von Paketverlusten eine große Rolle spielt. Paketverluste werden dabei auf zwei Arten erkannt: (1) durch Ablauf des Zeitgebers zur Sendewiederholung (d. h., erwartete Quittungen bleiben aus) oder (2) durch den Empfang drei duplizierter ACKs (DupACKs). Wird ein Paketverlust auf Basis drei duplizierter ACKs erkannt, wird dies als Zeichen gewertet, dass zwar ein Paketverlust eintrat, Folgepakete jedoch ihr Ziel erreicht haben. Da der Empfänger bei jedem empfangenen Paket ein ACK versendet, dieses aber nur Pakete bis zum letzten in Reihenfolge empfangenen Paket quittiert, bedeutet dies, dass die ACKs die gleiche Sequenznummern enthalten und damit Duplikate darstellen. Sendewiederholungen, die aufgrund duplizierter ACKs durchgeführt werden, werden als Fast Retransmit bezeichnet, da hier nicht auf das Auslaufen des Sendewiederholungszeitgebers gewartet wird. In den einzelnen Phasen verhält sich der Staukontrollalgorithmus wie folgt:

- Slow Start

  Die Slow Start Phase dient als „Herantasten“ an die verfügbare Bandbreite. Das Staukontrollfenster (Congestion Window, CWnd) wird da-

bei initial auf eine MSS (Maximum Segment Size) gesetzt. Die MSS beschreibt die maximale Größe einer TCP-Dateneinheit, die zwischen Sender und Empfänger versendet werden kann. Für jede Bestätigung einer MSS wird CWnd um eine MSS erhöht, was ein exponentielles Wachstum von CWnd zur Folge hat: Wird in der ersten Runde 1 MSS gesendet, wird CWnd bei einer Bestätigung der Dateneinheit um eine MSS erhöht und es können 2 MSS in der nächsten Runde gesendet werden. Werden beide Dateneinheiten wieder bestätigt, wird das CWnd auf 4 MSS erhöht und es können 4 Dateneinheiten versendet werden. Dies wird solange fortgeführt bis entweder ein Paketverlust erkannt wird oder der Schwellenwert ssthresh (Slow Start Threshold) erreicht wird. Am Anfang einer Verbindung wird dieser auf ∞ gesetzt, was zur Folge hat, dass auf jeden Fall ein Paketverlust eintreten wird. Der Wert ssthresh wird bei Paketverlust auf CWnd/2 gesetzt.

Wird der Paketverlust aufgrund des Ablaufens des Zeitgebers zur Sendewiederholung erkannt (tritt also ein Timeout ein), wird CWnd auf eine MSS reduziert und die Slow Start Phase beginnt von neuem. Wird der Paketverlust aufgrund drei duplizierter ACKs erkannt, geht der Algorithmus in die Fast Recovery Phase über. Wird hingegen der Schwellenwert ssthresh erreicht, geht er in die Congestion Avoidance Phase über.

- Congestion Avoidance

  Die Congestion Avoidance Phase wird betreten, wenn CWnd den Wert ssthresh erreicht oder wenn die Fast Recovery Phase erfolgreich verlassen wird. Hier wird CWnd nicht mehr exponentiell erhöht, sondern nur noch linear. Dies wird solange fortgeführt, bis ein Paketverlust erkannt wird, in welchem Fall ssthresh auf CWnd/2 gesetzt wird. Abhängig davon, ob der Paketverlust aufgrund eines Timeouts oder aufgrund drei duplizierter ACKs erkannt wurde, wird in die Slow Start Phase oder in die Fast Recovery Phase gewechselt.

- Fast Recovery

  Die Fast Recovery Phase wird betreten, wenn drei duplizierte ACKs empfangen werden, woraufhin ein Fast Retransmit durchgeführt wird. CWnd wird hier auf ssthresh + 3 MSS (für die drei duplizierten ACKs) gesetzt und bei jedem weiteren duplizierten ACK ebenfalls um eine MSS erhöht. Die Fast Recovery Phase wird verlassen, wenn alle ausstehenden Pakete bestätigt wurden ($\Rightarrow$ Congestion Avoidance) oder ein Timeout eintritt ($\Rightarrow$ Slow Start).

Zu der hier beschriebenen Reno-Variante existiert eine Erweiterung namens NewReno [HFGN12], die eine modifizierte Fast Recovery Phase vorschlägt.

Die Idee dahinter ist, bei sog. partiellen Quittungen – also wenn nach einem Fast Retransmit lediglich ein Teil der ausstehenden Pakete quittiert wurde – den Zeitgeber zur Sendewiederholung zurückzusetzen, um eine frühzeitige Sendewiederholung zu vermeiden. Dies ist dann sinnvoll, wenn in einem Sendefenster mehr als ein Paket verloren gegangen ist.

2.2.3 Vermittlungsprotokolle

IPv4 und dessen Nachfolger IPv6 sind Protokolle der Vermittlungsschicht im Internet. Sie haben u. a. folgende Aufgaben:

- Adressierung von Rechnern bzw. dessen Netzwerkschnittstellen

  Hierzu legt IP ein Adressformat und dessen Interpretation fest. Quell- und Zieladresse werden dabei im IP-Paketkopf vermerkt.

- Multiplexing der darüberliegenden Protokolle

  Ein Protokoll, das über IP genutzt wird, erhält eine Nummer, die in einem Feld des IP-Paketkopfes vermerkt wird. Auf dem Endsystem des Empfängers wird diese Nummer genutzt, um die Dateneinheit an die richtige Protokollinstanz „nach oben“ weiterzureichen.

- Weiterleitung von Paketen über Zwischensysteme

  Auf Basis der Zieladresse und der in jeder IP-Protokollinstanz vorhandenen Weiterleitungstabelle (Forwarding Table) wird ein IP-Paket über eine entsprechende Ausgangsschnittstelle an das nächste System weitergeleitet. Entspricht die Zieladresse einer Schnittstellenadresse des eigenen Systems, wird sie an das darüberliegende Protokoll weitergereicht.

Während die Wegewahl (Routing) ebenfalls in der Vermittlungsschicht angesiedelt ist, werden entsprechende Wegewahlprotokolle und -algorithmen nicht von IP festgelegt. Hier können in verschiedenen Netzen auch unterschiedliche Wegewahlprotokolle verwendet werden.

2.3 Aufbau des Internet

Das Internet besteht aus mehreren Netzen, die miteinander verbunden sind, und ist hierarchisch aufgebaut. Diese Hierarchie ist jedoch nicht streng, sodass auch Verbindungen zwischen Netzen über Hierarchie-Stufen hinweg möglich sind. Abbildung 2.5 veranschaulicht die Hierarchien [KuRo13]: Auf der Zugangsebene befinden sich Internetdienstanbieter (Internet Service Providers, ISPs), die Endbenutzern einen Anschluss an das Internet ermöglichen. Neben traditionellen Telekommunikationsunternehmen können dies

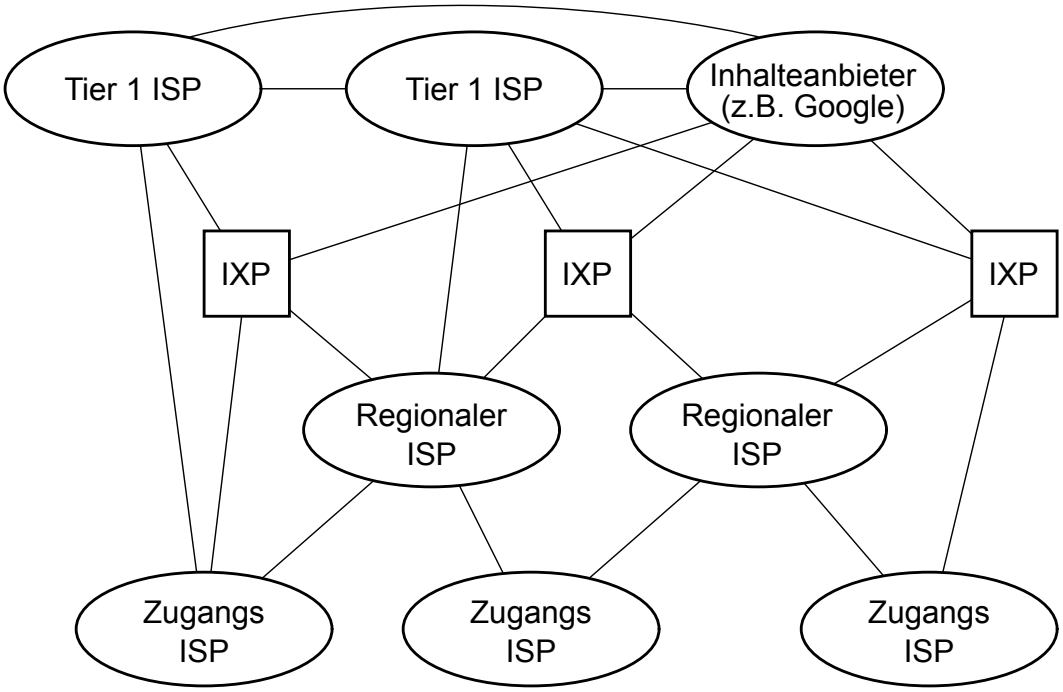


Abbildung 2.5 Schematischer Überblick über die Interkonnektivität von ISPs (nach [KuRo13]).

auch Firmen- und Universitätsnetze sein, die ihren Mitarbeitern einen entsprechenden Anschluss zur Verfügung stellen. Auf der nächsten Ebene sind regionale ISPs angesiedelt, die Städte, ganze Regionen oder ganze Länder abdecken. Diese wiederum sind mit sog. Tier 1 ISPs verbunden, die globale Ausdehnungen besitzen. Weder die regionalen noch die globalen ISPs verfügen über eine vollständige Abdeckung ihrer Region oder des Globus, sodass mehrere auf den gleichen Ebenen existieren und auch untereinander verbunden sind. Diese Konnektivität auf gleicher Ebene wird dabei als Peering bezeichnet. Inhalteanbieter können dabei ebenfalls mit einem eigenen Netz auftreten, welches mit mehreren anderen ISP-Netzen verbunden ist. Solche mehrfachen Verbindungen werden auch als Multihoming bezeichnet. Sogenannte Internet Exchange Points (IXPs) von Drittanbietern können weiterhin dazu dienen, Verbindungen zwischen den einzelnen ISPs herzustellen. Dies sind meist große Rechenzentren an einem Ort, in denen mehrere Leitungen verschiedener ISPs zusammenlaufen.

Die Konnektivität und wie am Ende die Daten tatsächlich ausgetauscht werden hängen häufig von ökonomischen und politischen Entscheidungen ab. Zur Interoperabilität wird hier das IP-Protokoll mit global eindeutigen Adressen genutzt. Diese Adressen werden in Blöcken von der IANA an die regionalen Internet Registries (RIRs) vergeben, die diese wiederum an lokale Internet Registries (LIRs) weitergeben. Letztere sind zumeist ISPs.

2.4 Probleme

Aus dem Aufbau der Internet-Protokollarchitektur beginnend mit der Anwendungsschicht über die Transportschicht bis hin zur Vermittlungsschicht ergeben sich einige Probleme, die die Einführung neuer Protokolle oder von Erweiterungen erschweren:

- Vermittlungsschicht
  - Adressen sichtbar bis zur Anwendung

    IP-Adressen werden auf der Vermittlungsschicht durch das Vermittlungsprotokoll definiert, sind aber bis zur Anwendung hin sichtbar. Ändert sich die IP-Adresse oder gar deren Format wie es beim Übergang von IPv4 auf IPv6 der Fall ist, wirkt sich dies bis zur Anwendung hin aus, die entsprechend angepasst werden muss. Obwohl die Anwendung in der Regel keinen weiteren Nutzen aus der IP-Adresse zieht, muss sie trotzdem damit umgehen können und solche Adressformatänderungen berücksichtigen.
  - IP zur Interoperabilität zwischen ISPs

    Ein weiteres Problem ergibt sich aus der Nutzung von IP zur Interoperabilität zwischen verschiedenen ISP-Netzen. Änderungen an IP müssen hier von allen ISPs mitgetragen werden. Zusätzlich sind Übergangslösungen notwendig, da ein kontrollierter „Neustart“ des Internets aufgrund von Protokolländerungen wegen dessen Größe nicht mehr durchführbar ist.

- Transportschicht
  - Auswahl des Transportprotokolls durch die Anwendung

    Um mit einer anderen Anwendung zu kommunizieren, muss die Anwendung ein Transportprotokoll auswählen. Durch diese explizite Angabe des Transportprotokolls legt sie sich jedoch auf dieses Protokoll fest. Existieren andere Protokolle, die einen ähnlichen oder gar besseren Kommunikationsdienst bereitstellen, können diese nicht genutzt werden, wenn sie nicht explizit durch die Anwendung unterstützt werden. Gerade Anwendungsentwickler, die kein ausgeprägtes Netzwerkwissen besitzen, ziehen etablierte Protokolle vor, und neue, alternative Protokolle fristen ein Nischendasein. Ein Beispiel hierfür sind die Protokolle SCTP [Stew07] und DCCP [KoHF06], die zwar schon lange existieren, jedoch kaum Verwendung finden.
  - Annahmen im Netz

Obwohl Transportprotokolle nach dem Ende-zu-Ende-Prinzip arbeiten, treffen einige Komponenten im Netz Annahmen über verwendete Transportprotokolle. Ein Beispiel hierfür sind Middleboxes wie NATs und Firewalls, die unbekannte Protokolle behindern können. Dies kann ebenfalls dazu führen, dass Anwendungsentwickler diese Protokolle meiden.

- Anwendungsschicht

  – Enge Verknüpfung des Anwendungsprotokolls mit dem Transportprotokoll

    Inzwischen setzen immer mehr Anwendungen auf ein einziges Anwendungsprotokoll auf: HTTP (vgl. Abschnitt 5.1). Dies sind dabei nicht nur Anwendungen, die im Web-Browser dargestellt werden, sondern viele, die einen Web-Dienst im Hintergrund verwenden. Dazu zählen insbesondere auch viele Smart Devices (Smart Phones, Smart TVs etc.). Dies ist zwar zum einen ein Zeichen dafür, dass HTTP für viele Anwendungen geeignete Abstraktionen gewählt hat, führt aber auch zum anderen dazu, dass das darunterliegende Protokoll mit TCP de facto festgelegt ist.

  – Zusätzliche Protokolle und Protokollerweiterung in Anwendungen

    Einige Protokolle oder Protokollerweiterungen werden dem Anwendungsentwickler „aufgebürdet“, insbesondere wenn es um sicherheitsrelevante Funktionen geht (z. B. TLS [DiRe08]). Diese werden zwar häufig auch als Anwendungsbibliotheken angeboten, die dem Anwendungsentwickler eine entsprechende Schnittstelle bereitstellen. Allerdings muss der Anwendungsentwickler auch hier entscheiden, welche er verwenden will. Sollen mehrere unterstützt werden, muss der Anwendungsentwickler dies bei der Implementierung berücksichtigen.

- Austausch von Informationen zwischen den Schichten

  Durch die strenge Trennung der Schichten ist es schwierig, Informationen über verschiedene Schichten hinweg auszutauschen. Bspw. geht das Transportprotokoll TCP bei einem Paketverlust automatisch von einer Überlastsituation im Netz aus, was zu entsprechenden Maßnahmen wie die Reduktion des Staukontrollfensters führt. Der Sicherungsschicht liegen jedoch ggfs. Informationen vor, dass ein Paket aufgrund eines Übertragungsfehlers verworfen wurde und nicht aufgrund einer Stausituation. Mit solchen Informationen könnte das Transportprotokoll anders reagieren, was dessen Durchsatz verbessern würde.

Durch die in der Einleitung beschriebene Entkopplung von Anwendungen und Protokollen können diese Probleme reduziert werden. Zusammen mit dem Szenario mehrerer nebenläufiger Netze für verschiedene Dienstanbieter wird so die Flexibilität bei der Einführung neuer Protokolle und Ansätze erhöht. Im nächsten Kapitel wird gezeigt, dass es solche neuen Vorschläge gibt, die sich allerdings jeweils nur mit Teilproblemen befassen.

3. Das Internet der Zukunft: Existierende Ansätze und Visionen

Unter dem Stichwort „Future Internet“ starteten insbesondere um die Jahre 2008 bis 2012 verschiedenen Initiativen, die sich mit Problemen der heutigen Internet-Architektur oder mit Teilen davon beschäftigten. Einen guten Überblick über diese Initiativen bieten Paul et al. in [PaPJ11]. Bestrebungen dieser Art fanden allerdings nicht nur in den letzten Jahren statt. Insbesondere Anfang der 1990er Jahre wurden architekturelle Ansätze bereits in Frage gestellt und neu formuliert. Bspw. diskutierten Clark et al. damals [ClTe90] wie heute [Clar09] grundlegende Design Entscheidungen.

In diesem Kapitel werden einige Ansätze näher betrachtet, die für die vorliegende Arbeit relevant sind. Dabei wird in Abschnitt 3.1 zunächst auf die Schnittstelle zwischen Anwendung und Kommunikationssystem eingegangen. Vorschläge zu alternativen Protokollen und Architekturen werden in Abschnitt 3.2 beschrieben. Am Ende beider Abschnitte wird darauf eingegangen, wo die Anknüpfungspunkte, Unterschiede und Verbesserungen der vorliegenden Arbeit im Vergleich zu bisherigen Ansätzen liegen. Mit Abschnitt 3.3 wird schließlich noch auf die grundsätzliche Funktionsweise von Netzwerkvirtualisierung eingegangen, welche eine wichtige Voraussetzung für das in der Einleitung motivierte Szenario darstellt.

3.1 Anwendungsschnittstellen

In Abschnitt 2.4 wurde herausgestellt, dass die bisherige Schnittstelle für Kommunikationsdienste relativ viele protokollspezifische Angaben von Seiten der Anwendung benötigt, wie bspw. die IP-Adresse und das zu verwendende Protokoll. In diesem Abschnitt werden existierende Ansätze vorgestellt, die sich zumindest teilweise diesem Problem widmen.

Eine in der IETF gestartete, aber inzwischen eingestellte Bestrebung sah vor, die bisherige Anwendungsschnittstelle – die Socket-Schnittstelle – zu erweitern, sodass statt IP-Adressen Namen verwendet werden können [UXMV10]. Dies hat den Vorteil, dass Anwendungen direkt die existierenden Rechnernamen aus dem DNS zum Verbindungsaufbau verwenden können und keine Adressauflösung mehr durchführen müssen. Unterschiede zwischen IPv4 und IPv6 oder generell die Auswahl der IP-Adresse im Fall von Multihoming bleiben somit vor der Anwendung verborgen, was eine Vereinfachung für den Anwendungsentwickler darstellt. Neben der Namensauflösung wurden in Diskussionen dazu auch die Auswahl des Transportprotokolls und der Portnummer auf Basis des Dienstnamens (z. B. „http“) erwähnt. Dennoch ist dieser Ansatz nach wie vor auf die von der IETF standardisierten Protokolle UDP, TCP, DCCP und SCTP beschränkt.

Ein davon unabhängiger, aber ergänzender Ansatz wird in [WeJG11] vorgeschlagen. Hier wurden die Protokolle UDP, UDP-Lite, TCP, DCCP und SCTP daraufhin untersucht, welche Transportdienste sie jeweils bereitstellen. Dazu wurde die Liste mit Transportdiensten auf die folgende reduziert:

- *Flow Characteristics*. Hiermit wird die Art der Staukontrolle festgelegt. Unterschieden wird zwischen einer Staukontrolle, die TCP-ähnlich ist (TCP-Like, wobei hier nicht zwischen den TCP-Varianten unterschieden wird), einer Staukontrolle die geeignet für Media-Streaming ist (Smooth, TCP-freundlich) und einer Staukontrolle die für kleine, kontinuierlich gesendete Pakete geeignet ist (Smooth-SP, Small Packets). Während die Intention der Unterscheidung nachvollziehbar ist, sind die Optionen unglücklich benannt, da hier eine Protokolleigenschaft zur Anwendung hin exponiert wird, die für sie nicht weiter von Interesse ist: die Art der Staukontrolle.

- *Application PDU Bundling*. Hiermit wird festgelegt, ob das Protokoll die Rekonstruktion von Anwendungsdateneinheiten (Application Protocol Data Unit) unterstützt. So können Anwendungen beliebig große Nachrichten versenden, die in der gleichen Form (mit markiertem Anfang und Ende) beim Empfänger ankommen.

- *Error Detection*. Mit der Fehlererkennung wird festgelegt, ob Bit-Fehler erkannt werden sollen und die Nachricht bei erkannten Fehlern entsprechend verworfen werden soll.
- *Delivery Order*. Hiermit wird angegeben, ob die Reihenfolge der Pakete eingehalten werden muss.
- *Reliability*. Hiermit wird der Umfang der Zuverlässigkeit in der Zustellung festgelegt: zuverlässig (vgl. Abschnitt 2.2.2.1), unzuverlässig oder teilweise zuverlässig. Mit der teilweisen Zuverlässigkeit ist hier nur eine zeitlich begrenzte Zuverlässigkeit gemeint, bei der Zustellversuche nur für eine benutzerdefinierte Zeit wiederholt werden. Bei dem Begriff „Reliability“ ist hier zwar die Fehlererkennung, aber nicht die Einhaltung der Reihenfolge inbegriffen.
- *Multi Homing*. Auf Server-Seite muss zusätzlich angegeben werden, ob Multi-Homing unterstützt werden soll, wenn der Server sich nur auf eine Untermenge der vorhandenen Netzwerkschnittstellen binden möchte.

Diese Bemühung zur Definition sinnvoller Diensteigenschaften zur Auswahl von Transportprotokollen wird aktuell im Rahmen einer IETF-Initiative fortgesetzt [MCWR+13]. Die direkte Adressierung mit IP-Adressen bleibt aber bei diesem Ansatz weiterhin bestehen. Trotzdem können diese Definitionen auch von anderen Schnittstellen verwendet werden, die ebenfalls eine Dienst- bzw. Anforderungsbeschreibung erwarten (z. B. [Lier+11, Reut10]).

Auf Anwendungsschicht existiert eine Vielzahl von Bibliotheken, die je nach Einsatzzweck sehr spezielle Kommunikationsabstraktionen bereitstellen. Ein programmiersprachenübergreifendes Beispiel ist dabei ZeroMQ [Hint13], welches verschiedene Paradigmen wie Publish/Subscribe oder Request/Response unterstützt. Hier, aber auch bei vielen anderen muss explizit ein Transportprotokoll von der Anwendung vorgegeben werden oder es ergibt sich implizit aufgrund festgelegter Konfigurationen. Ziel der vorliegenden Arbeit ist es jedoch, eine möglichst einfache Schnittstelle bereitzustellen, die nur die notwendigsten Maßnahmen trifft, um eine geeignete Entkopplung von Anwendungen und Protokollen zu erreichen. Eine solche Schnittstelle wird in Kapitel 5 vorgestellt.

3.2 Protokolle und Architekturen

In den nächsten beiden Abschnitten werden Änderungen und alternative Vorschläge zu heutigen Protokollen und die Gründe dafür näher betrachtet. Anschließend werden diese in Abschnitt 3.2.3 zusammengefasst und daraus die Motivation für den in dieser Arbeit vorgestellten komponentenbasierten

Ansatz abgeleitet. In Abschnitt 3.2.4 werden schließlich Ansätze vorgestellt, die eine Änderung der gesamten Internet-Architektur nach sich ziehen. Ein Ziel des Rahmenwerks, welches in Kapitel 6 vorgestellt wird, ist es auch solche Ansätze zu unterstützen.

3.2.1 Änderungen heutiger Protokolle und Alternativen

Für die heute eingesetzten Protokolle werden fortlaufend Vorschläge zu Änderungen gemacht, um aufgetretenen Problemen entgegenzuwirken. Diese Probleme hängen oft mit dem Wachstum des Internets, neuen Anwendungen und dem gesteigerten Leistungsbedarf zusammen. Neben Änderungen, die eine neue Protokollversion mit sich bringen (vgl. Änderungen zwischen IPv4 und IPv6) werden auch inkrementelle Änderungen an existierenden Protokollen vorgeschlagen. Als wichtigstes Transportprotokoll im Internet werden in diesem Abschnitt einige aktuelle Änderungsvorschläge und Alternativen für TCP vorgestellt.

In [Math12] werden architekturelle Änderungen an TCP vorgeschlagen, die es erleichtern sollen, neue Algorithmen zur Staukontrolle und zur Übertragungssteuerung (Transmission Scheduling) unabhängig voneinander zu realisieren. Laut den Autoren ist das Hauptproblem in dessen aktueller Spezifikation [AlPB09] die gemeinsame Nutzung von Zustandsvariablen wie ssthresh und CWnd. Durch die Einführung alternativer Variablen lässt sich eine bessere Trennung der Algorithmen erreichen. Der Ansatz zeigt, dass eine Modularisierung auch innerhalb eines Protokolls einer Schicht notwendig ist. So sind einfachere Weiterentwicklungen oder Anpassungen des Protokolls möglich.

Mit UDT (UDP-based Data Transfer) [GuGr07] wurde ein Transportprotokoll vorgestellt, welches – bezogen auf die Staukontrolle – ebenfalls einen modularen Ansatz nutzt. Die Autoren gehen dabei sogar soweit, dass sie der Anwendung ermöglichen, eigene Algorithmen zur Staukontrolle zu realisieren. Dazu werden relevante Ereignisse als Callback-Funktionen angeboten:

- onACK – wird bei Eintreffen einer Bestätigung aufgerufen. Hierüber können auch Paketverluste aufgrund duplizierter Bestätigungen festgestellt werden.
- onLoss – wird bei Eintreffen einer negativen Bestätigung (NACK) aufgerufen. Diese Funktion bleibt für TCP-Varianten der Staukontrolle ungenutzt.
- onTimeout – wird bei Eintreten eines Sendewiederholungszeitgebers ausgelöst. Der Wert kann entweder von der Anwendung gesetzt werden oder es wird das Verfahren aus [PaAl00] genutzt.

- onPktSent / onPktReceived – wird beim Versand bzw. beim Empfang von Datenpaketen aufgerufen.

- processCustomMsg – wird beim Empfang eines benutzerdefinierten Paketes aufgerufen.

Neben verschiedenen existierenden Staukontrollverfahren haben die Autoren auch ein eigenes Verfahren realisiert, welches sich besonders für Netze mit einem hohen Bandbreitenverzögerungsprodukt[1] eignet. Dieses bringt im Vergleich zu den TCP-Varianten einige Änderungen mit sich:

- Bestätigungen (ACKs) werden nicht mehr bei eintreffenden Datenpaketen versendet, sondern periodisch. Damit ist das ACK-Aufkommen unabhängig von der tatsächlichen Datenrate. Die Periode wird dabei auf Basis der aktuell geschätzten Paketumlaufzeit bestimmt.

- Es werden zusätzlich negative Bestätigungen (NACKs) verwendet, um den Verlust von Datenpaketen explizit beim Sender anzeigen zu können. Diese werden ebenfalls periodisch gesendet.

- Treffen beim Sender ACKs ein, werden diese mit ACK2-Nachrichten bestätigt. Dies dient dem Empfänger dazu, die aktuelle Paketumlaufzeit zu schätzen.

- Neben der kreditbasierten Übertragungssteuerung verwendet UDT zudem eine ratenbasierte Übertragungssteuerung. Hierbei wird vom Empfänger die aktuelle Verbindungsbandbreite auf Basis von Packet Pairing abgeschätzt: Bei Packet Pairing werden vom Sender (mindestens) zwei aufeinander folgende Pakete Richtung Empfänger gesendet. Auf Basis des im Vergleich zum Versand veränderten zeitlichen Abstands zwischen den beiden Paketen beim Empfänger kann so die limitierende Bandbreite auf dem Pfad geschätzt und eine Senderate bestimmt werden.

UDT ist nicht wie TCP oder UDP im Betriebssystemkern realisiert, sondern setzt auf UDP auf. Einen ähnlichen Ansatz wählt QUIC [Rosk13], wobei hier das Ziel ist, initiale Verbindungsaufbauzeiten zu verkürzen und weitere Optimierungen für Web-Datenverkehr zu treffen. QUIC ist speziell für diesen Anwendungsfall von Mitarbeitern des Dienstanbieters Google entworfen worden, dessen Interesse es ist, seine eigenen Dienste responsiver zu gestalten und somit die Erfahrung für den Benutzer zu verbessern. Aus dem gleichen Hause stammt SPDY [BePe12] welches Optimierungen für HTTP bietet.

[1] Produkt aus Bandbreite und Verzögerung einer Transportverbindung

Diese Beispiele zeigen, dass es für Dienstanbieter sinnvoll sein kann, eigens optimierte Protokolle zu verwenden, die auf ihre Dienste zugeschnitten sind. Nicht jeder Dienstanbieter bringt jedoch die Infrastruktur mit, die notwendig ist, um spezialisierte Protokolle effektiv einsetzen zu können. Mit dem in der Einleitung beschriebenen Szenario dienstanbieterspezifischer Netze und den in dieser Arbeit entwickelten Ansätzen ist es jedoch auch für kleine Dienstanbieter möglich, Optimierungen auf Netzebene zu treffen und damit mit großen Anbietern zu konkurrieren.

3.2.2 Komponentenbasierte Protokolle und Ansätze

Während die Schichtenarchitektur bereits zu einer Modularisierung des Kommunikationsdienstes führt, können die Schichten selbst immer noch sehr komplexe Aufgaben realisieren. Zur Modularisierung und Komposition einzelner Protokolle gibt es verschiedene Ansätze [HeSK10], wobei zwei Hauptzielsetzungen zu identifizieren sind:

- Bessere Parallelisierbarkeit

  Dies ist insbesondere in der Hochleistungskommunikation wichtig und zielt darauf, die Verarbeitung von Kontrollnachrichten und Daten zu parallelisieren. Beispiele hierzu sind die Protokolle Patroclos [BrSc94] und RBA [BrFH03].

- Höhere Flexibilität

  Hierunter zählt die Konfigurationen eines Kommunikationsdienstes, sodass er für die Anforderungen der Anwendung oder für die Eigenschaften des Netzes optimiert ist. Zusätzlich soll auch vermieden werden, dass sich einzelne Protokollmechanismen auf verschiedenen Schichten wiederholen. Beispiele hierzu sind F-CSS [ZiST93], SILO [DRBB+07], RNA [ToPi08], ANA [KHMB+08], Click [KMCJ+00] und SONATE bzw. SOCS [MüRe09, Reut10].

Patroclos

Patroclos [BrSc94] zielt u. a. auf die Trennung von Kontroll- und Datenverarbeitung ab. Insbesondere Protokollmechanismen zur Steuerung der Übertragung werden hier voneinander getrennt (Abbildung 3.1) und als Protokollfunktionen bezeichnet. Protokollfunktionen werden als endliche Automaten (Finite State Machines, FSMs) modelliert, die jeweils aus einer Sender- und einer Empfängerinstanz bestehen. Sender- und Empfängerinstanzen kommunizieren dabei direkt über individuelle Kontrollpakete miteinander, sodass die Abhängigkeiten zwischen den Protokollfunktionen reduziert werden. Informationen, die zwischen verschiedenen Protokollfunktionen auf dem selben Knoten ausgetauscht werden müssen, werden als lokale Ereignisse peri-

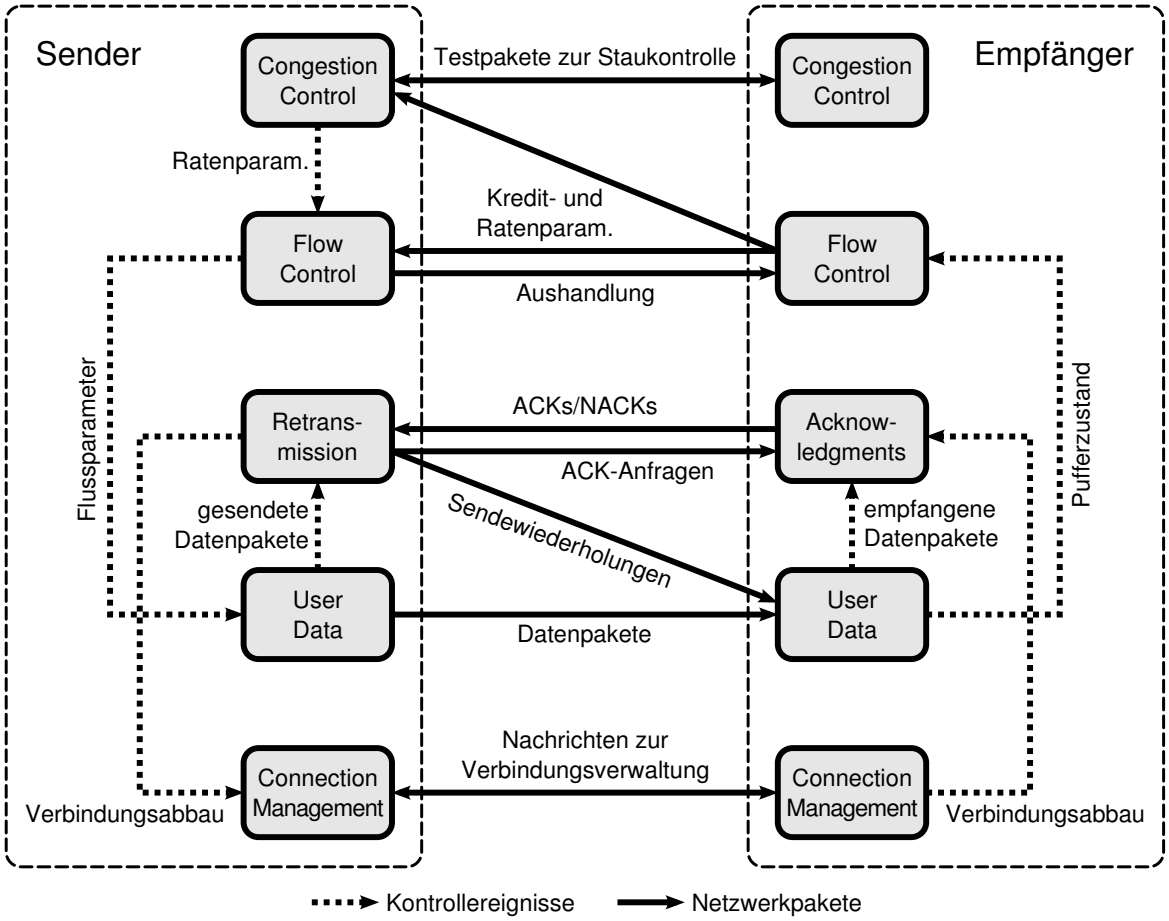


Abbildung 3.1 Patroclos: Protokollfunktionen und ihre Interaktion (nach [Brau93])

odisch versendet. Die Modularisierung bietet außerdem den Vorteil, dass Anwendungen auswählen können, welche Protokollmechanismen sie verwenden wollen [Brau95].

Das Problem bei Patroclos ist die Vielzahl an verschiedenen Kontrollpaketen, die über das Netzwerk versendet werden. Hier ist es auch nicht vorgesehen, dass Kontrollinformationen „Huckepack“ (*piggy-back*) mit den Daten übertragen werden.

Role-based Architecture (RBA)

Mit der Role-based Architecture (RBA) [BrFH03] wird ein Ansatz vorgeschlagen, bei dem unterschiedlichen Paketkopfeldern Rollen zugewiesen werden. Protokollmechanismen können dabei eine oder mehrere Rollen annehmen und sich so auf die entsprechenden Felder im Paketkopf registrieren. Rollen kommunizieren untereinander – sowohl über das Netzwerk als auch lokal – ausschließlich über Felder im Paketkopf. Dies bietet den Vorteil, dass die Rollen voneinander entkoppelt sind und beliebige neue Rollen mit rollenspe-

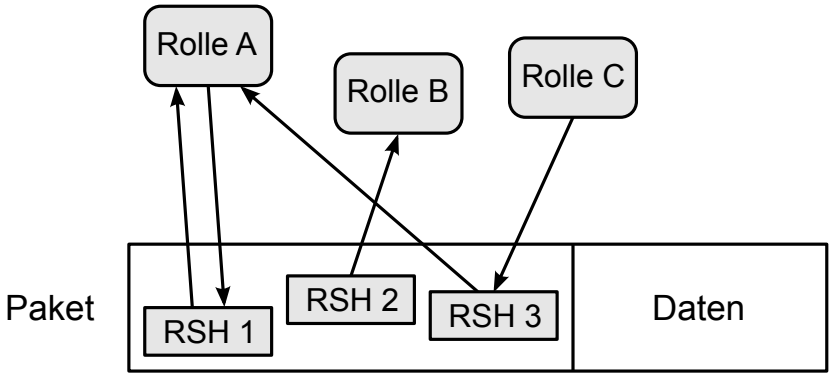


Abbildung 3.2 RBA: Protokollmechanismen als Rollen und die Nutzung rollenspezifischer Paketkopffelder (Role-Specific Header, RSH)

zifischen Paketkopfeldern (Role-Specific Header, RSH) hinzugefügt werden können (Abbildung 3.2). Eine in manchen Fällen notwendige Synchronisation zwischen den Rollen wird durch sog. Sequenzregeln erreicht, in denen eine Reihenfolge bei der Verarbeitung eines Paketes durch die Rollen festgelegt werden kann.

Function-Based Communication Subsystem (F-CSS)

Bei F-CSS [ZiST93] steht zunächst die Modularität zur dynamischen Konfiguration von Kommunikationsdiensten zur Laufzeit im Vordergrund. Protokollmechanismen können dabei als sog. Protocol Functions (PFs) abhängig von den Wünschen der Anwendung innerhalb einer sog. Protocol Machine (PM) instantiiert werden. Die Komposition von PFs übernimmt ein sog. PM Management Agent zur Laufzeit. Die so konstruierten PMs werden einer Session zugeordnet, die den Kommunikationsendpunkt für ein oder mehrere Anwendungsdatenströme darstellt. Zur Auswahl der gewünschten PFs bietet F-CSS eine Anwendungsschnittstelle, mit der feingranular der gewünschte Kommunikationsdienst spezifiziert werden kann. Zur Vereinfachung für Anwendung, die eine solche detaillierte Spezifikation nicht benötigen, werden Dienstklassen definiert, die eine Vorauswahl an PFs beinhalten.

Services Integration, controL, and Optimization (SILO)

SILO [DRBB$^+$07] schlägt die explizite Trennung von abstrakten *Diensten* und konkreten Implementierungen vor. Letztere werden dabei als *Methoden* bezeichnet, die konkrete Protokollmechanismen realisieren. Für einen Dienst können mehrere Methoden existieren, die den Dienst erfüllen. Dienste besitzen dienstspezifische Schnittstellen (sog. *Tuning Knobs*) zur Parametrisierung, die von der entsprechenden Methode implementiert werden müssen. Dienste, die von einer Anwendung benötigt werden, werden vertikal in einem sog. *Silo* angeordnet. Über eine Ontologie werden Abhängigkeiten zwischen den

Diensten modelliert, die bei der Komposition berücksichtigt werden müssen. Fokus von SILO liegt hier auf der dienstübergreifenden Optimierung eines konstruierten Silos mit Hilfe der Schnittstellen, die durch die Dienste bereitgestellt werden.

Recursive Network Architecture (RNA)

Mit RNA [ToPi08] wird ein sog. Meta-Protokoll eingeführt, welches für alle Schichten des Kommunikationssystems instantiiert werden kann. Das Meta-Protokoll unterstützt verschiedene Protokollmechanismen, von denen für eine konkrete Instanz eines Protokolls die gewünschten Mechanismen selektiert werden. Der Vorteil dadurch ist, dass Mechanismen einmalig für das Meta-Protokoll realisiert werden müssen und für jede daraus abgeleitete Instanz wiederverwendet werden können. Zugleich wird vermieden, dass sich Mechanismen auf unterschiedlichen Schichten wiederholen, da das Meta-Protokoll in jeder Schicht auf die Bedürfnisse der darüberliegenden und der darunterliegenden Schicht optimiert werden kann.

Die Reihenfolge der Abarbeitung einzelner Mechanismen in einer Protokollinstanz wird dabei durch das Meta-Protokoll definiert. Die Mechanismen sind dazu in einem gerichteten, azyklischen Graphen angeordnet. Nicht ausgewählte Mechanismen werden in einer konkreten Instanz einfach übersprungen.

Autonomic Network Architecture (ANA)

ANA [KHMB+08] führt zwischen Protokollkomponenten, die als Functional Blocks (FB) bezeichnet werden, sog. Information Dispatch Points (IDP) ein. Diese bilden eine Art Fassade für den eigentlichen FB und erlauben das einfache Austauschen eines FBs durch einen anderen FB. Ein FB kann auch mehrere IDPs bedienen. Die Weiterleitung von Nachrichten zwischen FBs geschieht dabei ausschließlich anhand der IDPs. Durch diese Indirektion über IDPs bleiben dem Entwickler der FBs viele Freiheiten was die Granularität der FBs angeht. Hinzu kommt, dass durch die IDPs auch die Netzwerkschnittstellen gekapselt werden und so keine Unterscheidung zwischen dem Nachrichtenaustausch mit einem anderen FB und dem Netzwerk getroffen werden muss.

Da mit ANA eine Vielzahl unterschiedlicher Protokollstapel möglich ist, wird das Konzept des *Compartments* eingeführt. Ein Compartment beinhalten dabei alle Protokollinstanzen und Geräte, die zu einer gemeinsamen Architektur gehören. So ist bspw. die TCP/IP-Protokollfamilie in einem Compartment angesiedelt.

Click Modular Router

Ein generisches Rahmenwerk zur Implementierungen von modularen Protokollen ist der Click Modular Router [KMCJ+00] (im Folgenden Click). Wie

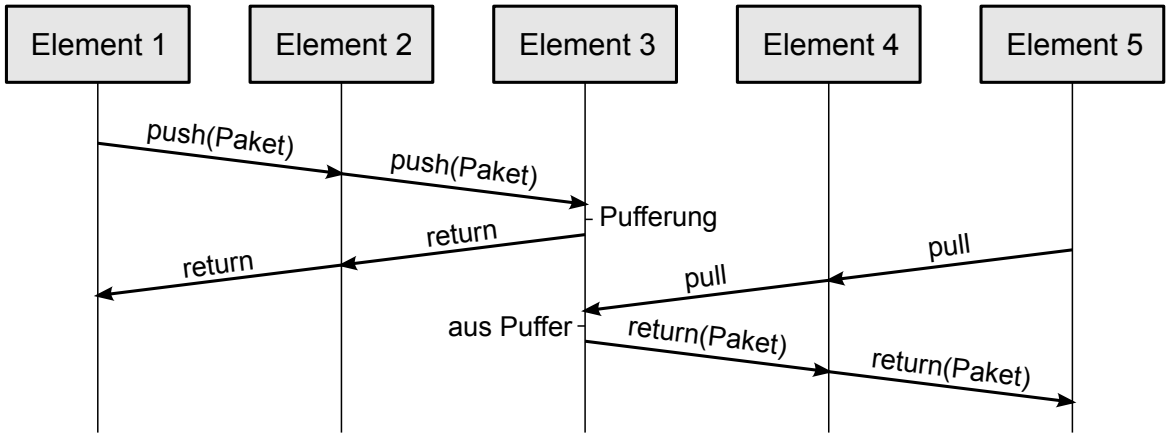


Abbildung 3.3 Click: Weiterreichen der Pakete zwischen den Elementen durch Methodenaufrufe

der Name andeutet, liegt der Hauptfokus auf Router-Funktionalität: Einzelne paketverarbeitende Module (*Elemente* genannt) werden dazu entlang eines gerichteten Graphen zusammengesetzt und in der so definierten Reihenfolge ausgeführt. Pakete werden zwischen den Elementen durch Methodenaufrufe weitergereicht (vgl. Abb. 3.3), entweder aktiv (push) oder reaktiv (pull). Dies bedeutet, dass bei einer Verarbeitung eines Pakets durch mehrere Elemente zunächst durch alle Elemente rekursiv abgestiegen wird, bevor das ursprüngliche Element weiter ausgeführt wird. Dies verhindert die Parallelisierbarkeit, wie sie bspw. von Patroclos angestrebt wird. Ereignisbasierte Architekturen lassen sich damit ebenfalls nur bedingt umsetzen.

Service-Oriented Architecture

Mit SONATE (Service-Oriented Network Architecture) [MüRe09] und SOCS (Service-Oriented Communication System) [ReHe08, Reut10] werden Ansätze verfolgt, in dem die Komposition von Protokollen auf Grundlage dienstorientierter Architekturen (Service-Oriented Architecture, SOA) erfolgt. SOA ist ein Ansatz, bei dem abstrakte Dienste mit Dienstschnittstellen definiert werden. Konkrete Mechanismen, die diese Dienste realisieren, bleiben nach außen transparent.

Die Zusammensetzung der verschiedenen Dienste geschieht hier auf Basis kompatibler Dienstschnittstellen und wird als sog. Workflow beschrieben. Auf Basis dieses Workflows können dann die einzelnen Dienste zur Laufzeit instantiiert und miteinander verknüpft werden. Schwerpunkt dieser Arbeiten liegt auf der Beschreibung und Aushandlung der verwendeten Dienste zwischen verschiedenen Kommunikationspartnern zur Laufzeit. Dazu wird ein generisches Protokoll verwendet, über das die Dateneinheiten der Dienste gemultiplext werden. Zur Aushandlung der Dienste zwischen zwei Kommunikationspartnern ist ebenfalls ein entsprechendes Protokoll notwendig.

	UDT[1]	QUIC[1]	Patroclos	RBA	F-CSS	SILO	RNA	ANA	Click	SOA	Ziel dieser Arbeit
Anpassung an spez. Aufgaben		✓	✓		✓	✓	✓	✓	✓	✓	✓
Lockerung der Schichtenarchitektur		✓		✓	✓	✓	✓	✓		✓	✓
Abstrakte Dienste & Austauschbarkeit	✓	✓		✓		✓				✓	✓
Flexible Kopplung der Dienste				✓	✓			✓	✓	✓	✓
Kontrollinform. über Ereignisse	✓		✓					✓		✓	✓
Keine Invarianten im Netz					✓	✓		✓	✓		✓

[1] UDT und QUIC bieten hier lediglich Anpassungen für jeweils einen speziellen Kommunikationsdienst. Die anderen Ansätze sind allgemeiner gehalten und erlauben Anpassungen für verschiedene Kommunikationsdienste.

Tabelle 3.1 Ziele und Konzepte komponentenbasierter Ansätze und die Ziele des komponentenbasierten Ansatzes der vorliegenden Arbeit

3.2.3 Konzepte komponentenbasierter Ansätze

Die in den letzten beiden Abschnitten beschriebenen Ansätze für aktuelle Protokolle und für komponentenbasierte Protokolle weisen teilweise gemeinsame Ziele und Konzepte auf (Tabelle 3.1):

- **Anpassung** der Protokolle **an spezielle Aufgaben**

  Hier liegt der Fokus auf der Auswahl von Protokollmechanismen, die entweder auf die Anwendung oder auf das Netzwerk zugeschnitten sind.

- **Lockerung der Schichtenarchitektur**

  Einige Ansätze heben die strikte Trennung des Internet-Schichtenmodells auf. Sie verzichten entweder komplett auf Schichten (z. B. RBA) oder nutzen sie weiterhin zur Strukturierung, allerdings ohne Vorgabe fester Schichten.

- **Abstrakte Dienste und Austauschbarkeit** konkreter Funktionalität

  Einige Ansätze trennen konkrete Mechanismen und Dienste, die durch sie bereitgestellt werden, um hier mehr Flexibilität bei der Einführung neuer Mechanismen für dieselben Dienste zu erreichen.

- **Flexible Kopplung der Dienste**

  Teilweise wird eine Trennung zwischen den Diensten angestrebt, die eine beliebige Anordnung und Kommunikation (= Kopplung) der Dienste untereinander ermöglicht. Andere Ansätze machen jedoch strenge Vorgaben, in welcher Reihenfolge Dienste ausgeführt werden sollen.

- Austausch von **Kontrollinformationen über Ereignisse**

  Kontrollinformationen werden teilweise nicht mehr nur mit den Dateneinheiten ausgetauscht, sondern können auch unabhängig davon durch eigene Ereignisse an andere Dienste weitergereicht werden. Der Vorteil liegt hier in der besseren Entkopplung der Dienste voneinander und in der möglichen Parallelisierbarkeit.

- **Keine Invarianten im Netz**

  Während einige Ansätze keine festen Vorgaben zu Paketformaten oder Protokollen machen, sind sie bei anderen notwendig, um bspw. die verwendeten Protokollmechanismen auszuhandeln. Solche Vorgaben werden als Invarianten bezeichnet.

Ziel des komponentenbasierten Ansatzes, der als Teil dieser Arbeit in Kapitel 4 vorgestellt wird, ist es, diese Konzepte zu vereinen ohne eine strenge Sequenz von Diensten vorzugeben. Die dazu benötigte Flexibilität wird erreicht, indem die Komposition der Protokolle nicht mehr zur Laufzeit, sondern zur Entwurfszeit durchgeführt wird. Zur Laufzeit wird Flexibilität erhöht, indem mehrere, sehr unterschiedliche Protokolle (auch von verschiedenen Netzwerkarchitekturen) nebenläufig betrieben werden (Kapitel 6) und Anwendungen über eine protokollagnostische Schnittstelle darauf zugreifen (Kapitel 5).

3.2.4 Neue Netzwerkarchitekturen

Die bisher in diesem Abschnitt vorgestellten Ansätze beschäftigen sich entweder mit Alternativen zu konkreten Protokollen (UDT, QUIC, Patroclos) oder generell mit der Konstruktion von Protokollen. Vorschläge, die ganzheitliche Alternativen zur heutigen Internet-Architektur und zur TCP/IP-Protokollfamilie darstellen, gibt es jedoch ebenfalls. Bspw. wird mit Content-Centric Networking (CCN) [JSTP+09] ein Ansatz für eine gänzlich inhaltsbasierte Netzwerkarchitektur vorgestellt: Selbst die Vermittlungsschicht adressiert Inhalte an Stelle von Knoten bzw. Netzwerkschnittstellen. Ein anderer Ansatz ist XIA [HADL+12], welcher es erlaubt, Protokolle und Netze mit verschiedenen Kommunikationsparadigmen nebenläufig zu betreiben. XIA führt dazu sog. *Principals* ein, wobei jeder Principal ein eigenes Kommunikationsparadigma repräsentiert. Mit einem Satz von grundsätzlichen Protokollen

zur Vermittlung und zum Transport, die auf Basis dieser Prinicpals arbeiten und die die Interoperabilität zwischen verschiedenen Netzbetreibern herstellen, wird es Anwendungen ermöglicht, neue Paradigmen mit dann bereits existierenden Protokollen zu nutzen. CCN und XIA werden im Rahmen dieser Arbeit in Kapitel 7 näher betrachtet.

Im Gegensatz zu XIA, dessen Protokolle hauptsächlich auf dem Datenpfad angesiedelt sind, konzentriert sich das Framework for Internet Innovation (FII) [KSBF+11] auf den Kontrollpfad, um die Interoperabilität zwischen verschiedenen Netzbetreibern (Domänen) zu ermöglichen. Dazu definiert FII drei wesentliche Schnittstellen:

- Interdomain-Schnittstelle zwischen Netzbetreibern

  Hier wird das Konzept eines *Pathlets* definiert. Ein Pathlet beschreibt einen (virtuellen) Übertragungsabschnitt innerhalb einer Domäne, welcher nach außen an andere Domänen bekanntgegeben wird. Der Ende-zu-Ende-Pfad besteht dann aus mehreren Pathlets, die zueinander kompatibel sind. Die Kompatibilität wird auf Basis der tatsächlichen verwendeten Protokolle innerhalb dieser Pathlets bestimmt.

- Schnittstelle zwischen Anwendungen und FII

  Die Anwendungsschnittstelle von FII kann verschiedene Ausprägungen besitzen, die als (engl.) *Schema* bezeichnet werden. Eine Ausprägung kann bspw. eine Publish/Subscribe-Schnittstelle oder eine RPC[1]-Schnittstelle sein. Die unterstützten Schemas können von der Anwendung vor Benutzung abgefragt werden. So können später auch neue Schemas hinzugefügt werden.

- Schnittstelle zum Schutz vor DDoS-Angriffen

  Diese Schnittstelle ermöglicht es einem Empfänger (oder Zwischenknoten), auf Wunsch die Weiterleitung von Nachrichten an sich selbst mit einer sog. *Shut-up Message* (SUM) zu unterbinden. Gerade bei verschiedenen Architekturen, die zusammenspielen sollen, sahen die Autoren aufgrund ihrer Erfahrung es als notwendig an, eine solche Funktionalität als zentrale Anforderung zu stellen.

Neben diesen drei hauptsächlichen Schnittstellen werden weitere Schnittstellen definiert, u. a. zum Bootstrapping und zur Protokollaushandlung. Während der Vorteil einer solchen Architektur ist, dass jeder Netzbetreiber eigene Protokolle und Netzwerkarchitekturen entwerfen kann, muss doch jeder auch die entsprechenden Schnittstellen realisieren und auf die eigene Netzwerkarchitektur abbilden. Eine Möglichkeit, die eine stärkere Entkopplung

[1] Remote Procedure Call

von der Infrastruktur des Netzbetreibers und den tatsächlich verwendeten Protokollen ermöglicht, ist Netzwerkvirtualisierung. Diese wird im folgenden Abschnitt näher beschrieben.

3.3 Netzwerkvirtualisierung

Die in der Einleitung beschriebene Entkopplung von Anwendungen und Protokollen erlaubt es, Anwendungen so flexibel zu halten, dass zu deren Entwicklung noch keine konkrete Auswahl an Kommunikationsprotokollen getroffen werden muss. Dies ermöglicht die Bereitstellung neuer Protokolle, von denen existierende Anwendungen profitieren können. Da Protokolle allerdings oft an eine bestimmte Netzwerkarchitektur angepasst sind und diese gewisse Invarianten vorgibt, ist die Flexibilität zum Entwurf neuer Protokolle zum Teil stark eingeschränkt – insbesondere dann, wenn eine Unterstützung auf Zwischenknoten im Netz für diese neuen Protokolle notwendig wird. Eine mögliche Lösung diesbzgl. stellt der parallele Betrieb mehrerer Netzwerkarchitekturen gleichzeitig dar. Hierbei werden Netze soweit voneinander isoliert, dass innerhalb dieser Netze eine beliebige Netzwerkarchitektur betrieben werden kann, ohne dass eine Kompatibilität mit anderen Netzen berücksichtigt werden muss. Ein wichtiges Werkzeug zum parallelen Betrieb unterschiedlicher Netzwerkarchitekturen ist Netzwerkvirtualisierung. Sie stellt dem Dienstanbieter neben einer freien Wahl der Protokolle auch eine gewisse Elastizität seines Netzes zur Verfügung und erlaubt es ihm, Link- und Knoteneigenschaften zu definieren, zu erweitern und zu reduzieren. Ein Überblick über aktuelle Techniken und über Vorschläge zu neuen Virtualisierungsarchitekturen gibt [ChBo10]. Die grundlegenden Abstraktionen sind jedoch vielen gemein und werden in diesem Abschnitt beschrieben (vgl. Abbildung 3.4).

Infrastrukturbetreiber sind im Besitz physischer Netze und können vom Umfang her mit den ISPs aus Abschnitt 2.3 verglichen werden. Sie können dabei mit anderen Infrastrukturbetreibern verbunden sein und ähnlich wie die Zugangs-ISPs Endbenutzern einen physischen Anschluss an ihr Netz bereitstellen. Die Inhalte, die der Endbenutzer anfordert, werden jedoch nicht direkt vom Infrastrukturbetreiber bereitgestellt, sondern von einem *Dienstanbieter*.

Der Dienstanbieter fordert von einem oder mehreren Infrastrukturbetreibern Ressourcen für ein *virtuelles Netz* an und pachtet diese. Diese Ressourcen sind virtuelle Knoten und virtuelle Links. Der Infrastrukturbetreiber teilt dazu die Ressourcen seiner physischen Komponenten auf, um die virtuellen Komponenten bereitzustellen. Die Anforderung von Ressourcen von verschiedenen Infrastrukturbetreibern kann dabei auch über einen *Broker* erfolgen, der zwischen dem Dienstanbieter und mehreren Infrastrukturbetreibern vermittelt.

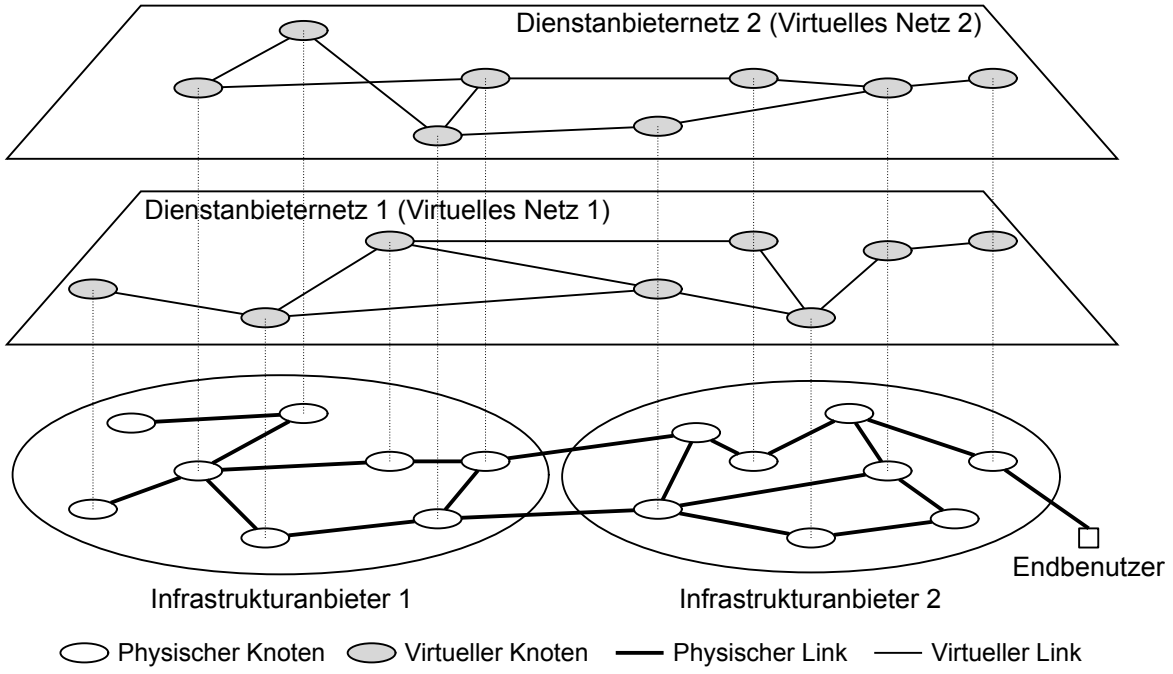


Abbildung 3.4 Netzwerkvirtualisierung (nach [ChBo10])

Der Broker wird in einigen Ansätzen auch als Anbieter virtueller Netze bezeichnet (*virtual network provider*).

Die Daten, die innerhalb des virtuellen Netzes ausgetauscht werden, sind für den Infrastrukturbetreiber transparent. Das bedeutet, dass der Dienstanbieter frei wählen kann, welche Protokolle und welche Dienste er innerhalb seines virtuellen Netzes verwendet. So kann er Protokolle auswählen, die auf seine Bedürfnisse zugeschnitten sind. Aufgrund einer möglichen Zusicherung von virtuellen Ressourcen wie Bandbreiten und Rechenzeit durch den Infrastrukturbetreiber muss der Dienstanbieter auch keine Wechselwirkungen mit anderen virtuellen Netzen befürchten. So kann er auf entsprechende Protokollmechanismen verzichten, wenn er sie nicht für konkurrierende Verbindungen in seinem eignen Netz benötigt.

Netzwerkvirtualisierung ist damit die ideale Voraussetzung für das Szenario dienstanbieterspezifischer Netze, wie es in der Einleitung beschrieben wurde. Ergänzend dazu bietet die vorliegende Arbeit weitere Abstraktionen, die den Dienstanbieter bei der Auswahl und beim Entwurf der Protokolle für sein virtuelles Netz unterstützen und die Endsysteme der Benutzer für diese neuen Protokolle vorbereiten.

4. Komponentenbasierter Protokollentwurf

In diesem Kapitel wird der in dieser Arbeit entwickelte komponentenbasierte Protokollentwurf mittels Protokollschablonen beschrieben. Abschnitt 4.1 führt dazu zunächst eine Einordnung in den in der Einleitung eingeführten Lebenszyklus vor und detailliert die Prozesse zur Entwurfszeit [VMRB+09]. Hier wird ebenfalls beschrieben, auf welcher Abstraktionsebene der komponentenbasierte Protokollentwurf angesiedelt ist. Auf die Protokollkomponenten bzw. Protokollbausteine geht Abschnitt 4.2 detailliert ein, während der Protokollschablonenansatz und ein ausführliches Beispiel für eine Transportprotokollschablone in Abschnitt 4.3 beschrieben werden. Zu dieser Transportprotokollschablone werden zudem Variationen diskutiert, und es wird gezeigt, dass Änderungen aufgrund der gegebenen Schablone mit moderatem Aufwand durchführbar sind. Abschnitt 4.4 skizziert abschließend ein Beispiel, wie Modellierungswerkzeuge eingesetzt werden können, um Teile der Protokollbeschreibung automatisiert in Programmcode umsetzen zu können. Abschnitt 4.5 fasst die wesentlichen Beiträge und Ergebnisse des Kapitels noch einmal zusammen.

4.1 Entwurf

Dieser Abschnitt geht zunächst auf die verschiedenen Abstraktionsebenen ein, mit denen eine Netzwerkarchitektur und deren Protokollkomponenten mit verschiedener Granularität beschrieben werden. Anschließend wird ein Entwicklungsprozess [VMRB+09] vorgestellt, der die wichtigsten Entwurfs-

phasen beschreibt. Im letzten Unterabschnitt werden die Grundideen des komponentenbasierten Entwurfs beschrieben.

Am Lebenszyklus einer Netzwerkarchitektur sind verschiedene Akteure beteiligt, deren Aufgabengebiete voneinander getrennt werden können. Je nach Expertise können die einzelnen Rollen aber auch auf weniger Akteure vereint werden. Die Rollen sind:

- Der *Dienstanbieter*, der Inhalte oder Anwendungsdienste bereitstellen möchte, auf die seine Benutzer zugreifen. Er ist der Auftraggeber für ein neues, dienstanbieterspezifisches Netz mit angepasster Kommunikationssoftware.
- Der *Architekt*, der basierend auf den Anforderungen des Dienstanbieters passende Kommunikationsdienste beschreibt und ggfs. aus existierender Kommunikationssoftware passende Komponenten auswählt, die wiederverwendet werden können.
- Der *Entwickler*, der die wiederverwendbaren Komponenten existierender Kommunikationssoftware anpasst und fehlende Komponenten basierend auf den Beschreibungen des Architekten implementiert.
- Der *Netzbetreiber*, der für den ordnungsgemäßen Betrieb des Dienstanbieternetzes zuständig ist.
- Der *Infrastrukturbetreiber*, der die physische Infrastruktur für den Betrieb des Dienstanbieternetzes bereitstellt.
- Der *Endbenutzer*, der Inhalte oder Dienste nutzt, die über das neue Dienstanbieternetz angeboten werden.

Der Dienstanbieter, der Infrastrukturbetreiber und der Endbenutzer sind dabei direkt auf die Rollen abbildbar, die im Kontext der Netzwerkvirtualisierung verwendet werden (vgl. Abschnitt 3.3). Für den Netzbetreiber existiert bei einigen Virtualisierungsansätzen ebenfalls eine entsprechende Rolle (*virtual network operator*). Ein Broker (auch *virtual network provider*), der zwischen Netzbetreiber und Infrastrukturbetreiber vermittelt ist hier ebenfalls möglich, aber für die vorliegende Arbeit nicht weiter relevant.

4.1.1 Abstraktionsebenen

Die zur Beschreibung einer Netzwerkarchitektur notwendigen Informationen unterscheiden sich je nach Einsatzzweck stark in ihrem Umfang und lassen sich durch die jeweiligen Abstraktionensebenen, wie sie in Abbildung 4.1 dargestellt sind, differenzieren:

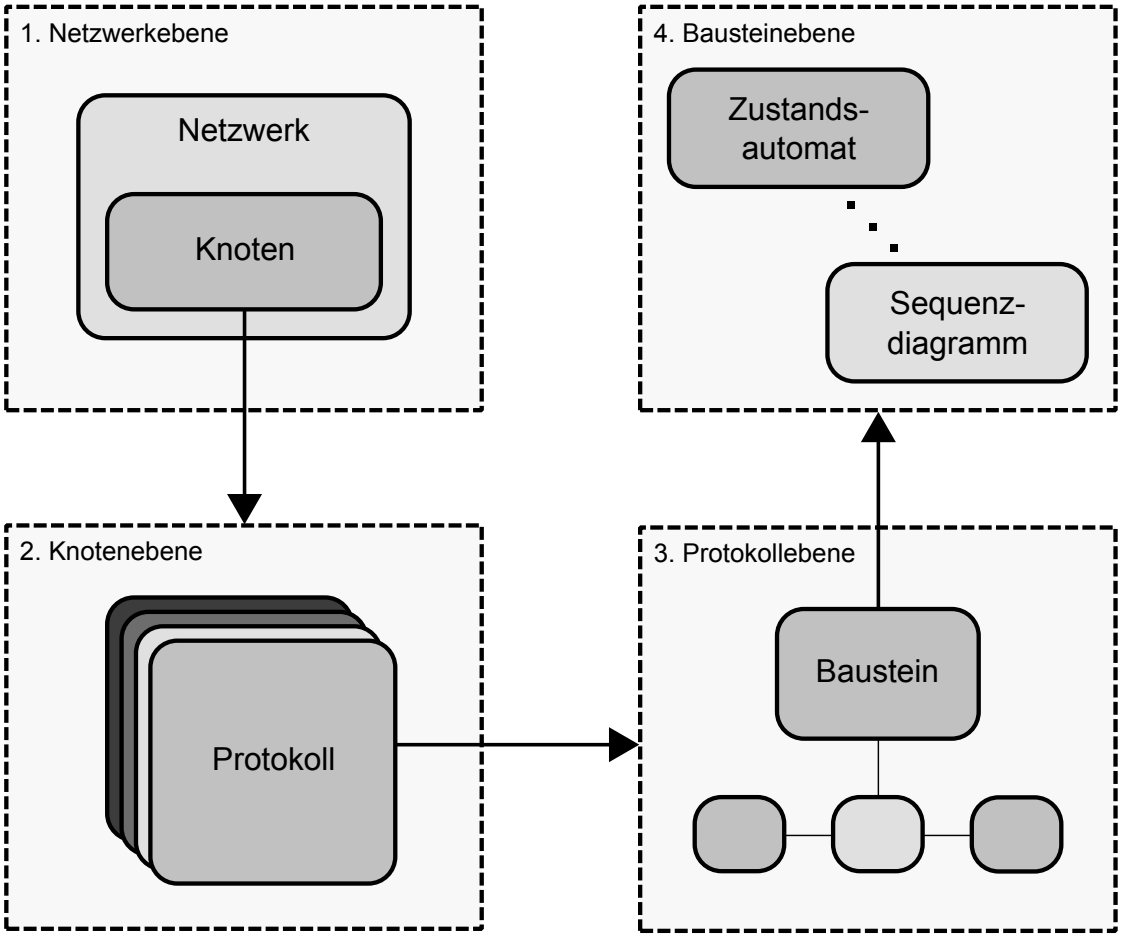


Abbildung 4.1 Abstraktionsebenen für den Entwurf von Netzwerkarchitekturen. Mit dem Entwurf der Netzwerkarchitektur kann dann eine konkrete Netzinstanz entworfen werden, wie es in Abb. 1.2 angedeutet wird.

1. Netzwerkebene

   Auf dieser Ebene werden die verschiedenen funktionellen Einheiten im Netzwerk mit ihren Aufgaben und Schnittstellen durch den Architekten beschrieben und auf entsprechende Knotentypen der Netzwerkarchitektur abgebildet. Zu diesen Knotentypen zählen z. B. Endsysteme, Router, Gateways und Middleboxes. Zusammen mit zugehörigen Leistungsmerkmalen werden diese Informationen vom Netzbetreiber bei der Netzplanung genutzt. Er fordert dazu die notwendigen Ressourcen beim Infrastrukturbetreiber an.

2. Knotenebene

   Auf dieser Ebene beschreibt der Architekt, welche Protokolle auf den jeweiligen Knotentypen zum Einsatz kommen. Eine Liste mit eindeutigen Protokollbezeichnungen ist dabei ausreichend.

3. Protokollebene

   Auf Protokollebene wird der Kommunikationsdienst, der vom jeweiligen Protokoll erbracht wird, vom Architekten beschrieben. Zusätzlich legt der Entwickler den Aufbau des Protokolls bestehend aus *Protokollbausteinen* fest bzw. modifiziert existierende Protokolle. Protokollbausteine können dabei einzelne Protokollmechanismen oder vollständige Protokolle realisieren, die ihrerseits wieder aus Protokollbausteinen zusammengesetzt sind.

4. Bausteinebene

   Die Bausteinebene beschreibt die Realisierung eines konkreten Protokollbausteins durch den Entwickler. Neben herkömmlichen Beschreibungen als RFC, können hier auch formale Methoden wie SDL verwendet werden [KuGW06].

Die vollständige Beschreibung einer Netzwerkarchitektur beinhaltet Beschreibungen auf allen Ebenen. Für die Protokollausbringung zur Laufzeit ist die Netzwerk- und die Knotenebene relevant. Die Kommunikationssoftware für die Netzwerkarchitektur wird dagegen nur durch die Protokoll- und Bausteinebene beschrieben. Der in dieser Arbeit beschriebene Entwurf von Protokollen durch Komposition beschäftigt sich dabei hauptsächlich mit der Protokollebene.

4.1.2 Entwicklungsprozess

Software-Entwicklungsprozesse haben eine lange Historie und reichen über sehr detailliert spezifizierte Prozesse wie dem V-Modell [LuLi07] bis hin zu agilen Methoden wie Scrum [PiRo11], die nicht mehr als Rahmenwerke bzw. Richtlinien zum Entwicklungsprozess und Projekt-Management definieren. Diese Entwicklung basiert teilweise auf der Erkenntnis, dass Entwicklungsprozesse sehr individuell sind: Sie müssen auf die Unternehmensstruktur, aber auch auf die Entwickler und Kunden angepasst werden, sodass selbst detaillierte Prozesse aus dem Lehrbuch nicht ohne Anpassung in Software-Entwicklungshäusern eingesetzt werden. Daher werden hier lediglich Prozessrahmen für die Protokollentwicklung vorgestellt, ausgehend von denen die beteiligten Akteure ihre eigenen Prozesse entwickeln oder die sie in ihre existierenden Prozesse integrieren können.

Der in Abb. 1.3 auf Seite 5 oben dargestellte Entwicklungsprozess zur Entwurfszeit besteht aus den beiden Hauptaktivitäten *Dienstentwurf* und *Komponentenentwurf*. Der Dienstentwurf beschreibt dabei zum einen die Zielsetzungen des Dienstanbieters sowie Anforderungen bzgl. der Integration in bestehende Umgebungen. Zum anderen werden hier vom Architekten die

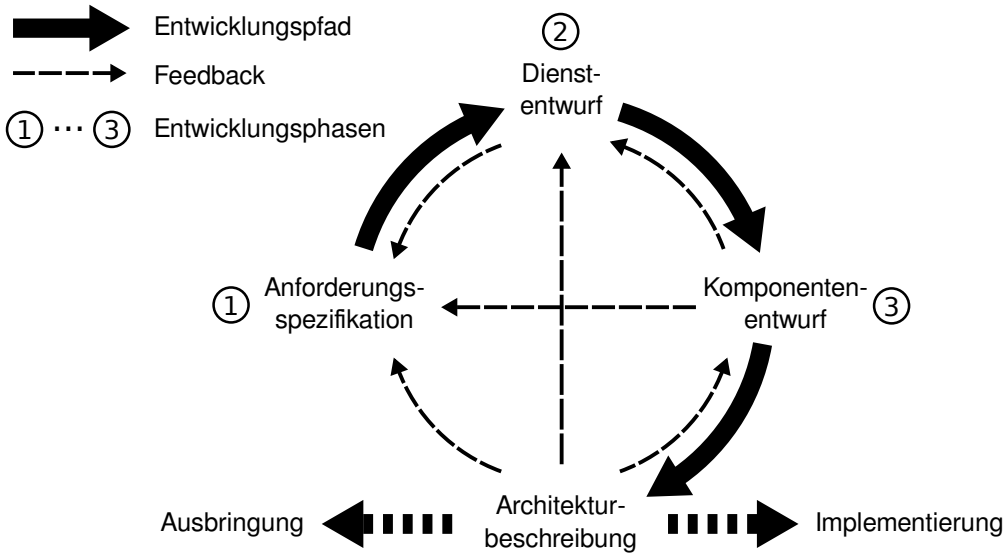


Abbildung 4.2 Iterativer Entwicklungsprozess

notwendigen Kommunikationsdienste beschrieben und/oder unter existierenden ausgewählt. Der Komponentenentwurf hingegen beschreibt den Entwurf einzelner Protokolle oder Protokollkomponenten, die aus dem Dienstentwurf abgeleitet werden. Werkzeugunterstützung (bspw. [Rohr09, Baue13]) spielt hierbei eine wichtige Rolle, zum einen zur Identifikation der notwendigen und eventuell bereits existierenden Protokollkomponenten, zum anderen zur Modellierung der Protokolleigenschaften der fertigen Komposition. Zusammen mit dem Repository vereinfacht dies die Wiederverwendung existierender Lösung und beschleunigt die Entwicklung.

Abbildung 4.2 detailliert den Entwicklungsprozess in einzelne Phasen, die ähnlich zu den Phasen typischer Software-Entwicklungsprozesse sind: Die Anforderungsspezifikation (1) hilft dem Architekten, die Anforderungen zusammen mit dem Dienstanbieter zu formulieren und ggfs. iterativ zu verfeinern. Basierend auf der Anforderungsspezifikation wählt der Architekt aus existierenden Protokollen und Architekturen geeignete Komponenten aus und/oder stellt Anforderungen für neue, zu entwerfende Komponenten (2). Während des Komponentenentwurfs (3) werden die ausgewählten Komponenten durch den Entwickler konfiguriert, oder es wird die Spezifikation neuer Komponenten durchgeführt. Das Ergebnis dieser drei Phasen ist dann eine Beschreibung der neuen bzw. angepassten Netzwerkarchitektur, die auf die Anforderungen des Dienstanbieters zugeschnitten ist. Die Beschreibung erstreckt sich dabei über alle Abstraktionsebenen hinweg. Mit ihr kann die Implementierung durch den Entwickler bzw. die Ausbringung durch den Netzbetreiber beginnen. Zusätzlich können die neuentworfenen Komponenten zukünftigen Netzwerkarchitekturen zur Verfügung gestellt und somit wie-

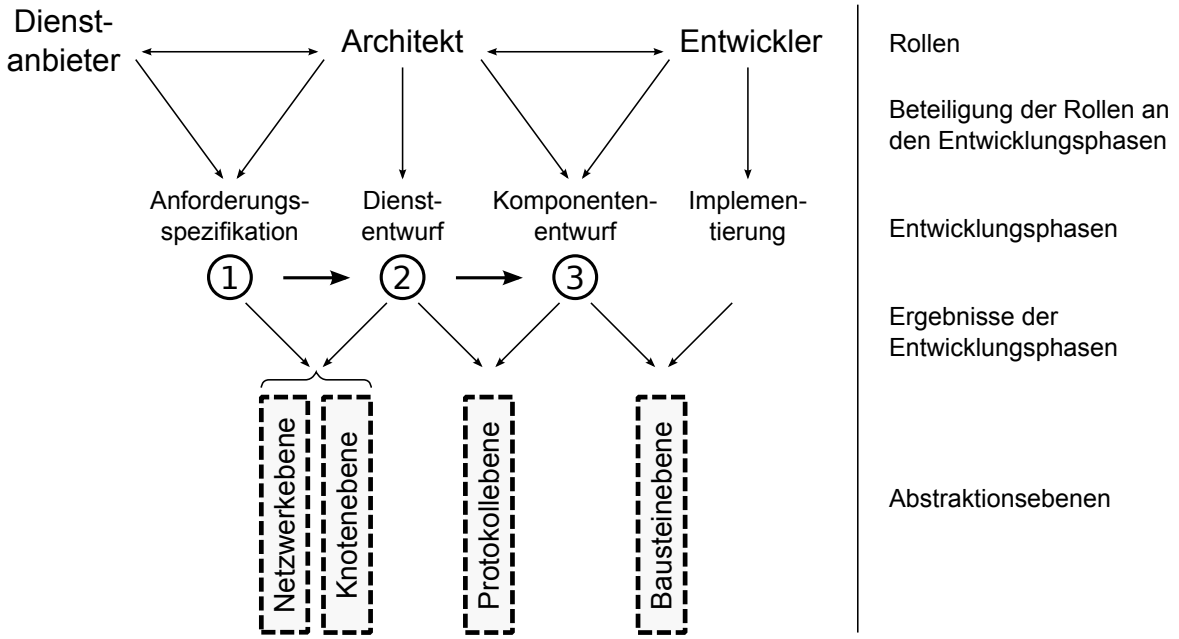


Abbildung 4.3 Zusammenhänge zwischen den beteiligten Rollen, den Entwurfsphasen und den Abstraktionsebenen

derverwendet werden. Dieser iterative Prozess erlaubt dabei Rückmeldungen der einzelnen Phasen in eine beliebige vorherigen Phase, um frühzeitig und agil auf Probleme reagieren zu können.

Die Interaktion und die Zusammenhänge zwischen den beteiligten Rollen des Entwurfs, den Entwurfsphasen und den Abstraktionsebenen sind in Abbildung 4.3 skizziert.

4.1.3 Komponentenbasierter Entwurf

Der Entwurf von neuen Protokollen erfolgt in dieser Arbeit durch die Komposition von existierenden und evtl. neu zu entwickelnden Protokollkomponenten. Ansätze zur Protokollkomposition basierend auf einzelnen Protokollmechanismen gab es in den letzten Jahrzehnten verschiedene (vgl. hierzu Abschnitt 3.2.2). Die Vorteile liegen auf der Hand: Gewünschte Protokollmechanismen können basierend auf den gegebenen Anforderungen und den aktuellen Netzeigenschaften selektiert werden, und existierende Lösungen für sich wiederholende Problemstellungen können wiederverwendet werden.

Aus gleicher Motivation verfolgt auch diese Arbeit einen Protokollkompositionsansatz. Entgegen vieler existierender Vorschläge soll die Auswahl der Protokollkomponenten allerdings zur Entwurfszeit erfolgen und nicht erst zur Laufzeit, wenn ein Kommunikationsdienst durch eine Anwendung angefordert wird. Dies hat zwar den Nachteil, dass nicht auf neue Anwendungsanforderungen oder auf Änderungen von Netzeigenschaften so dynamisch

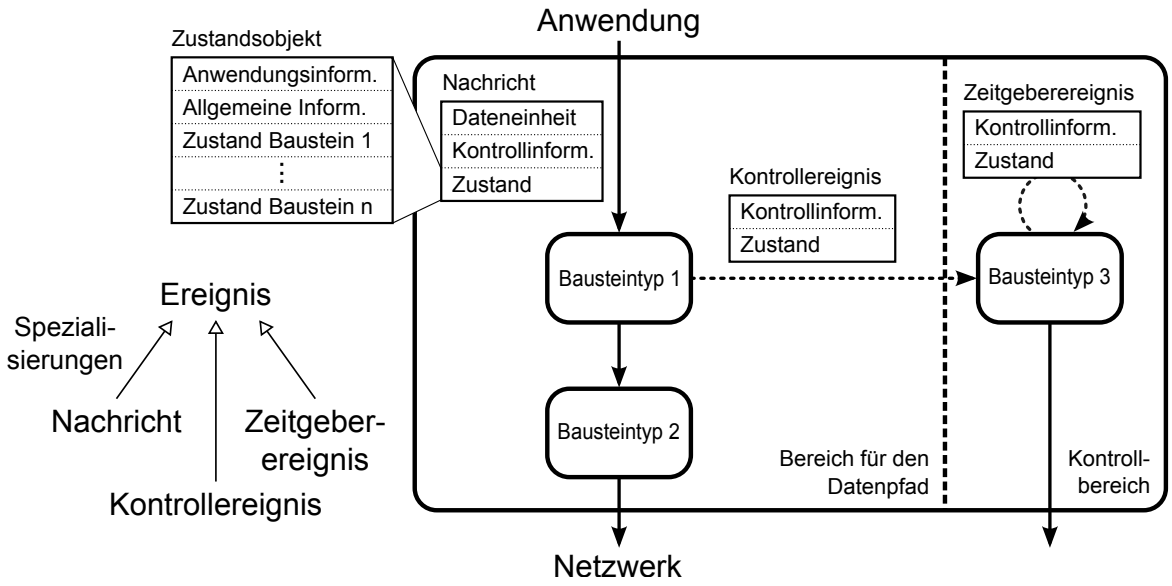


Abbildung 4.4 Skizze einer Protokollschablone

reagiert werden kann wie bei einigen Laufzeitkompositionsansätzen. Jedoch wird dadurch der Aufwand zur Laufzeit drastisch reduziert, da keine dynamische Komposition pro Kommunikationswunsch stattfindet. So ist außerdem besser vorhersagbar, wie sich ein zusammengesetztes Protokoll im Produktiveinsatz verhält, da es im Vorfeld getestet werden kann.

Die Protokollkomposition zur Entwurfszeit bietet aber dennoch eine deutlich gesteigerte Flexibilität als der traditionelle Protokollentwurf durch kürzere Entwicklungszyklen: Sind Änderungen an Mechanismen bisher verwendeter Protokolle notwendig oder vorteilhaft, können die Komponenten (auch *Bausteine* genannt), die diese Mechanismen realisieren, ausgetauscht werden und das aktualisierte Protokoll anschließend wieder ausgebracht werden. Der in der vorliegenden Arbeit vorgestellte Kompositionsansatz zur Entwurfszeit kann wie folgt skizziert werden (vgl. Abbildung 4.4):

- Protokolle und deren Mechanismen sind prinzipiell ereignisgesteuert. Sie werden nur ausgeführt, wenn Nachrichten gesendet oder empfangen werden, wenn Kontrollereignisse eintreffen oder wenn abgelaufene Zeitgeber eintreten. Alle Ereignisse enthalten grundsätzlich Kontrollinformationen und einen Verweis auf ein Zustandsobjekt, in dem auch bausteinspezifische Zustandsinformationen abgelegt werden. Nachrichten enthalten zudem die zu versendende oder die empfangene Dateneinheit.

- Protokollmechanismen sind in Bausteinen realisiert. Eine Familie von Mechanismen (bspw. zur Staukontrolle) wird durch einen sog. Baustein-

typ repräsentiert. Konkrete Realisierungen verfeinern einen Bausteintyp zu einem Baustein.

- Ein Protokoll wird mit der Auswahl und Komposition von Bausteintypen auf Basis des Dienstes, den das Protokoll erbringen soll, entworfen. Dies ergibt eine sog. *Protokollschablone*. Konkrete, in Bausteinen realisierte Mechanismen werden in einem zweiten Schritt ausgewählt und in die Schablone eingesetzt. Ob hierbei eine Trennung zwischen Kontrollbereich und Datenpfad erfolgt, hängt von der jeweiligen Protokollschablone ab (dies bleibt also dem Entwickler überlassen).

Protokollschablonen bieten hier einen Mittelweg zwischen der heute verbreiteten, informellen Spezifikation von Protokollen (bspw. im Rahmen von RFCs) und der nur in wenigen Bereichen verwendeten formalen Spezifikation (bspw. über SDL). Während der Entwurf von Schablonen für eine Klasse von Protokollen weiterhin eine komplexe Aufgabe ist und von Experten durchgeführt wird, sind spätere Anpassungen für einen Dienstanbieter durch einen Architekten einfacher und damit schneller durchführbar, da die grundlegende Funktionsweise bereits durch die Schablone definiert ist. Diese Anpassungen können erfolgen, indem passende Mechanismen für die in der Schablone vorgegebenen Bausteintypen selektiert werden und/ oder einzelne Bausteine komplett weggelassen werden.

Bevor auf die Protokollschablonen in Abschnitt 4.3 näher eingegangen wird, wird im folgenden Abschnitt der Aufbau von Protokollbausteinen erläutert.

4.2 Protokollbausteine

Protokollbausteine realisieren einzelne Protokollmechanismen bis hin zu kompletten Protokollen. Bausteine, die zu versendende Nachrichten weiterverarbeiten, sind in dieser Hinsicht – also auf dem Datenpfad – nach dem Schichtenprinzip angeordnet – ein Konzept, welches sich in der Vergangenheit als erfolgreich herausgestellt hat. Die Kombinationsmöglichkeiten von Bausteinen bzgl. des Datenpfads sind in Abbildung 4.5 zusammengefasst. Neben Verzweigungen im Rahmen eines gerichteten Graphen sind auch Bausteine möglich, die andere Bausteine enthalten (*Vorkomposition*). Durch die Kombination eines Bausteins zur *Verteilung* und eines Bausteins zum *Multiplexing* können auch verschiedene Aufgaben parallel ausgeführt werden.

Da der Austausch von Kontrollinformationen oft schichtenübergreifend notwendig ist, sollen diese nicht ausschließlich über ein- und ausgehende Nachrichten ausgetauscht werden, sondern auch über Kontrollereignisse und über Zustandsinformationen. In den folgenden Unterabschnitten wird daher auf alle Schnittstellen und die Zustandshaltung eines Protokollbausteins eingegangen.

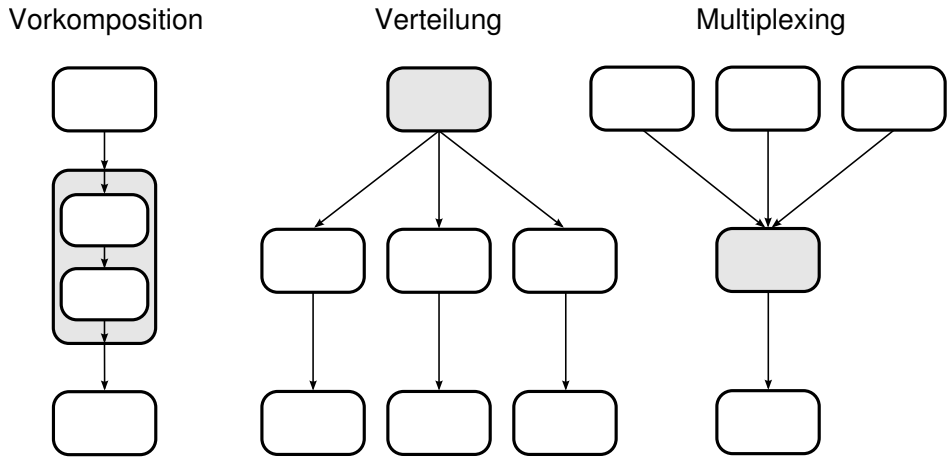


Abbildung 4.5 Protokollkompositionen auf dem Datenpfad

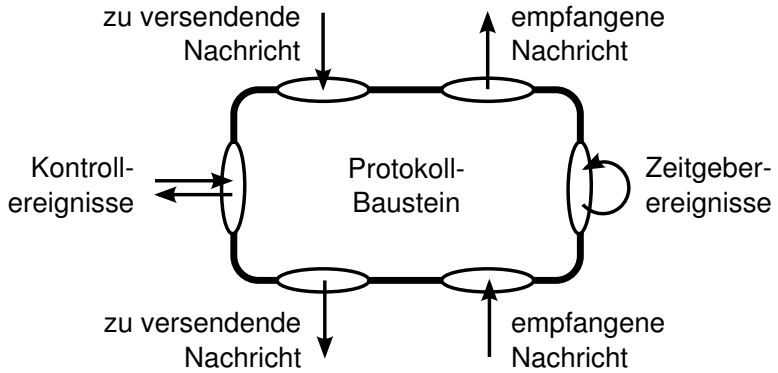


Abbildung 4.6 Ereignisschnittstellen eines Protokollbausteins

4.2.1 Schnittstellen

Die verschiedenen Schnittstellen eines Protokollbausteins sind in Abbildung 4.6 dargestellt. Für zu versendende Nachrichten gibt es eine „obere“ Schnittstelle und eine „untere“, die beide im wesentlichen mit dem Dienstzugangspunkt (Service Access Point, SAP) aus dem OSI-Modell vergleichbar sind. Über die „untere“ Schnittstelle werden die zu versendende Dateneinheiten zusammen mit Kontrollinformationen als Nachricht an den nächsten Baustein (die nächste Schicht) weitergereicht. Für eingehende Nachrichten existieren entsprechende Schnittstellen.

Zusätzlich können Kontrollinformationen auch als Kontrollereignisse durch einen Baustein versandt werden. Ebenso kann der Baustein Kontrollereignisse empfangen, die von anderen Bausteinen ausgelöst werden. Dies bedeutet insbesondere, dass Bausteine im Kontrollbereich (also die nicht direkt die zu versendenden oder die empfangenen Nachrichten verarbeiten) beliebig angeordnet sein können. Schließlich kann ein Baustein Zeitgeber setzen, die bei Auslösung ein Zeitgeberereignis erzeugen.

4.2.2 Zustandshaltung

Einige Protokollmechanismen müssen Informationen bzgl. ihrer Parametrisierung und ihres Zustands halten. Diese Informationen können sein:

- Flow-spezifisch: Informationen, die mit einer aktuellen Kommunikationsbeziehung (Flow) assoziiert sind. Ein Beispiel hierfür ist die Größe des aktuellen Sendefensters eines entsprechenden Protokolls.
- Systemspezifisch: Informationen, die über alle Kommunikationsbeziehung hinweg relevant sind. Ein Beispiel hierfür sind die belegten Port-Nummern von TCP, die für neue Flows bekannt sein müssen, damit eine freie Portnummer ausgewählt werden kann.

Da in dieser Arbeit eine Trennung zwischen Bausteintyp und einer konkreten Realisierung vorgenommen wird, ist es wichtig, dass allgemeine Zustandsvariablen und Parameter des Bausteintyps vollständig dokumentiert werden. Diese Zustandsvariablen und Parameter werden dann von der konkreten Bausteinrealisierung verwendet.

Die Zustände aller Bausteine, die bei der Bearbeitung eines Anwendungs-Flow involviert sind, werden in einem sog. Zustandsobjekt zusammengefasst. Dieses wird mit jedem Ereignis dem entsprechenden Baustein mit übergeben (vgl. Abb. 4.4). So kann der Baustein auch auf die Zustände der anderen Bausteine zugreifen, sofern er den jeweiligen Bausteintyp kennt. Neben dem Versenden von Kontrollereignissen ist dies eine weitere Möglichkeit Informationen bausteinübergeifend auszutauschen.

4.2.3 Beschreibung

Die Beschreibung von Protokollbaustein*typen* kann in drei Bereiche unterteilt werden, die in Abbildung 4.7 links zusammengefasst sind. Zur Formulierung dieser Bereiche eignen sich maschinenlesbare Formate wie XML oder gar eigens definierte domänenspezifische Sprachen (Domain Specific Languages, DSLs). Im Folgenden wird bei der Erläuterung der Bereiche jeweils beispielhaft ein einfaches Pseudo-Format verwendet, um die jeweils enthaltenen Informationen zu verdeutlichen:

- Meta-Beschreibung

  Die Meta-Beschreibung enthält eine eindeutige ID des Bausteintyps als URI, dessen Namen, eine informelle, allgemeine Beschreibung des Dienstes und eine Dokumentation. Diese Informationen werden vom Architekten benötigt, um existierende Bausteine zu finden. Für den Fall, dass der Architekt einen neuen Baustein entwirft, muss der Architekt diese Informationen für den Entwickler erstellen. Die Meta-Beschreibung kann bspw. so formuliert werden:

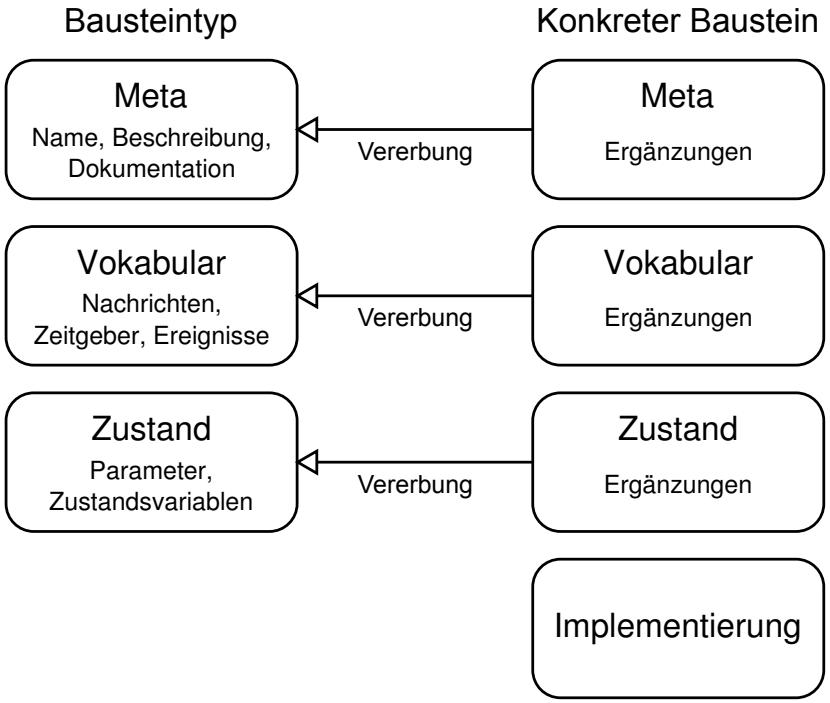


Abbildung 4.7 Überblick über die Beschreibung von Protokollbausteinen

```
Meta: {
  ID: <URI>,
  Name: <Freitext>,
  Beschreibung: <Freitext>,
  Dokumentation: <Freitext>
}
```

- Vokabular

  Mit dem Vokabular werden alle Nachrichten, Kontrollereignisse und Zeitgeber beschrieben. Diese Beschreibung wird ebenfalls vom Architekten für den Entwickler erstellt und sieht im Pseudo-Format so aus:

  ```
  Vokabular: {
    Nachrichten: {
      Nachricht1: { ... }, Nachricht2: { ... }, ...
    },
    Kontrollereignisse: { ... },
    Zeitgeber: { ... },
  }
  ```

  Die jeweiligen Nachrichten, Kontrollereignisse und Zeitgeber sind dabei weiter zu detaillieren, z. B.:

  ```
  Nachricht1: {
    Name: <Freitext>,
  ```

```
  Beschreibung: <Freitext>,
  Dokumentation: <Freitext>,
  Paketformat: {
    Version: 4-Bit vorzeichenloser Zahlwert,
    Verkehrsklasse: 8-Bit vorzeichenloser Zahlwert,
    ...
  }
}
```

Für die Angaben zum Paketformat können etablierte Beschreibungssprachen genutzt werden, z. B. ABNF [CrOv08] oder Google Protocol Buffers [Goog14].

- Zustandsvariablen und Parameter

  Auch Zustandsvariablen und Parameter, die bereits durch den Bausteintyp vorgegeben werden, müssen beschrieben werden, damit diese von konkreten Bausteinen genutzt werden können und andere Bausteine darauf zugreifen können. Diese Beschreibung wird vom Architekten initial erstellt und vom Entwickler ergänzt. Im Pseudo-Format sieht diese beispielhaft so aus:

```
Zustand: {
  Parameter: {
    InitialeFenstergroesse: {
      Name: <Freitext>,
      Beschreibung: <Freitext>,
      Dokumentation: <Freitext>,
      Typ: 16-Bit vorzeichenloser Zahlwert
  },
  Variable: { ... },
}
```

Konkrete Protokollbausteine erben dabei die Beschreibungen der Protokollbausteintypen und ergänzen bzw. konkretisieren diese (Abbildung 4.7 rechts). Zusätzlich wird für Protokollbausteine auch deren Implementierung und Verhalten beschrieben. Vorgaben dazu werden hier jedoch keine gemacht.

Die Definition der Beschreibungen in einem maschinenlesbaren Format, welches an das Pseudo-Format angelehnt ist, bringt folgende Vorteile:

- Die Suche und Verarbeitung durch Werkzeuge während der Entwurfszeit wird vereinfacht. Bausteine können mit der Beschreibung innerhalb von Entwurfswerkzeugen wie z. B. [Rohr09, Baue13] nachgeladen und genutzt werden.

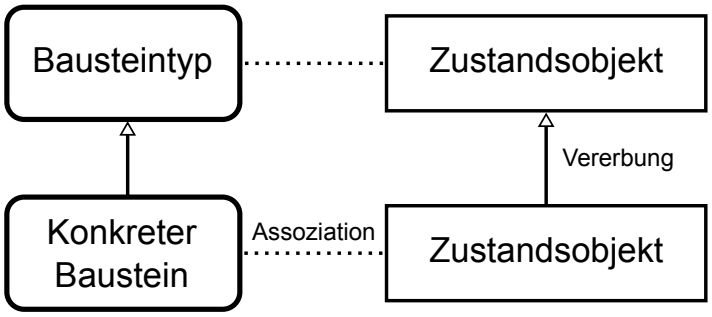


Abbildung 4.8 Bausteintypen mit ihren Zustandsobjekten und deren Vererbung für konkrete Bausteine.

- Implementierungsspezifische Datenstrukturen können aus der Definition des Vokabulars, der Zustandsvariablen und der Parameter automatisiert abgeleitet werden. Für ein Laufzeitrahmenwerk wie z. B. NENA (Kapitel 6) können die so erzeugten Datenstrukturen von der Implementierung verwendet werden. Fehler z. B. in der Dimensionierung der Datenstrukturen werden so reduziert, da die Dimensionierung Teil der Beschreibungen ist.

Gerade für den letzten Punkt, der die einfache Generierung von Code betrifft, existieren Modellierungswerkzeuge, die das Erstellung entsprechender Generatoren vereinfachen. Hierfür wird in Abschnitt 4.4 ein kurzes Beispiel gegeben.

4.3 Protokollschablonen

In diesem Abschnitt werden die Entwurfsprinzipien für Protokollschablonen zusammengefasst und es wird eine konkrete Schablone für Transportprotokolle vorgestellt. Für diese werden anschließend Anpassungen für alternative Transportprotokolle diskutiert [Fran12, Mart13].

4.3.1 Entwurfsprinzipien

Die Entwurfsprinzipien für die hier vorgeschlagenen Protokollschablonen lassen sich wie folgt zusammenfassen (vgl. Abschnitt 4.1.3):

- Protokolle und deren Mechanismen sind ausschließlich ereignisgesteuert und als Bausteine realisiert.
- Mechanismen bzw. Bausteine können zu einem Bausteintyp generalisiert werden, der vom konkreten Algorithmus und von der konkreten Implementierung abstrahiert. Die genutzten Zustandsinformationen können dabei ebenfalls generalisiert werden (vgl. Abbildung 4.8). Dies

ist auch durch die Vererbbarkeit der Beschreibung widergespiegelt (vgl. Abschnitt 4.2.3).

- Bausteine auf dem Datenpfad sind in einem gerichteten Graph angeordnet (vgl. Abbildung 4.5). Konkrete Schichten werden nicht festgelegt.

Kontroll- und Zustandsinformationen können zwischen Bausteinen auf folgende Arten ausgetauscht werden (vgl. Abbildung 4.4):

- Bausteine auf dem Datenpfad können Kontrollinformationen als Eigenschaften an die zu sendende bzw. an die empfangene Nachricht anhängen.
- Bausteine können beliebige, eigens definierte Kontrollereignisse auslösen, auf die sich andere Bausteine registrieren können.
- Bausteine können Kontrollinformationen in einem Zustandsobjekt ablegen, welches jedem Baustein bei jedem Kontrollereignis, Zeitgeberereignis oder bei jeder Nachricht mit übergeben wird.

4.3.2 Transportprotokollschablone

Als beispielhafte Realisierung einer Protokollschablone wird in diesem Abschnitt eine vollständige Protokollschablone für ein zuverlässiges Transportprotokoll [Fran12] vorgestellt, welches das gleiche Verhalten wie TCP-NewReno (vgl. Abschnitt 2.2.2.2) aufweist. In Abschnitt 4.3.3 werden anschließend Variationen diskutiert, mit denen gezeigt wird, dass mit moderatem Aufwand auch alternative Transportprotokolle realisiert werden können. Im Vergleich zu einem kompletten Neuentwurf zeigt sich dadurch der Mehrwert von Protokollschablonen.

4.3.2.1 Überblick

Ein zuverlässiges Transportprotokoll muss für folgende Aufgaben Mechanismen realisieren (vgl. Abschnitt 2.2.2):

- Fehlererkennung
- Paketverlust-, Duplikat- und Phantomdateneinheiterkennung
- Sicherstellung der Paketreihenfolge
- Übertragungswiederholung
- Flusskontrolle
- Staukontrolle

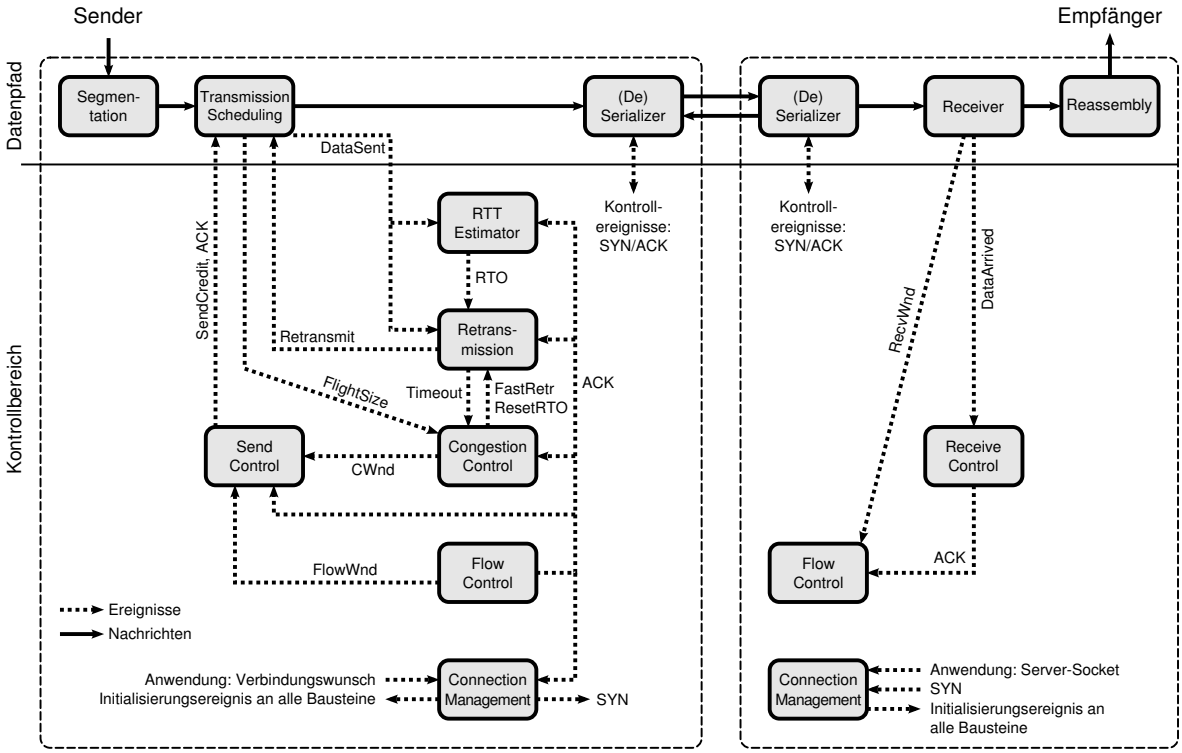


Abbildung 4.9 Kopplung der Bausteintypen über Ereignisse in der hier vorgestellten Transportprotokollschablone. Über diese Schablone lassen sich TCP-ähnliche, zuverlässige Transportprotokolle realisieren. Die eingezeichneten Ereignisse und Bausteintypen werden in dieser Form für ein Protokoll genutzt, dass das gleiche Verhalten wie TCP aufweist.

Das hier vorgestellte Transportprotokoll basiert dabei auf einer Aufteilung dieser Aufgaben auf Mechanismen wie sie in Patroclos ([BrSc94], Abschnitt 3.2.2) verwendet wurden und verfeinert und flexibilisiert diese. Ähnliche Aufteilungen sind auch in der Literatur (z. B. [Day08]) zu finden. Während also die Mechanismen für ein zuverlässiges Transportprotokoll und deren grundsätzliches Zusammenspiel auf existierenden Arbeiten basiert, ist die damit erstellte Schablone und deren Variationen zur Erstellung weiterer Transportprotokolle eine wichtige Neuerung.

In Abbildung 4.9 ist ein Überblick über die Kopplung der Bausteine einer Transportprotokollschablone gegeben, mit der zuverlässige Transportprotokolle realisiert werden können, die ähnliche Mechanismen wie TCP aufweisen. Eingezeichnet sind dabei die Bausteintypen zusammen mit Ereignissen, die eine Kopplung zu anderen Bausteintypen herstellen. Für die vollständige Dokumentation der Schablone sind neben dieser Kopplung der Bausteintypen folgende Informationen über die Bausteintypen selbst nötig (vgl. Abschnitt 4.2.3):

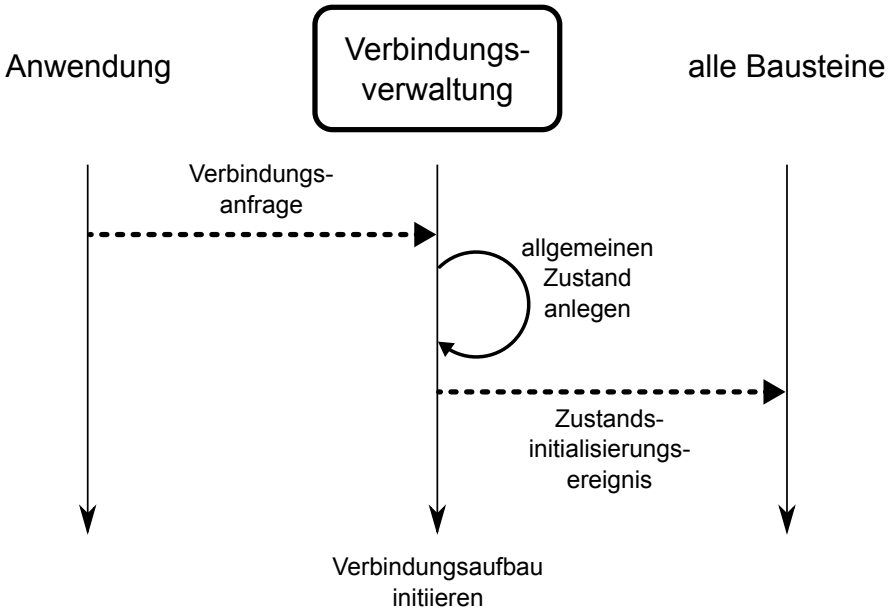


Abbildung 4.10 Zustandsinitialisierung

- die Meta-Beschreibung (allgemeine Dokumentation)
- die Vokabularbeschreibung (Nachrichten, Zeitgeber, Kontrollereignisse)
- die Zustandsbeschreibung (Parameter und Zustandsvariablen)

Die Bausteintypen stellen Platzhalter für Bausteine dar, welche die in den Abschnitten 4.3.2.3 (Datenpfad) und 4.3.2.4 (Kontrollbereich) beschriebenen Aufgaben übernehmen. Dort wird auch die Kopplung mit anderen Bausteinen näher beschrieben. Konkrete Bausteine müssen zudem die Beschreibung des Bausteintyps, den sie spezialisieren, verfeinern und insbesondere ihr Verhalten dokumentieren. Zusätzlich müssen sie Angaben über das Paketformat machen, welches sie verwenden.

4.3.2.2 Zustandshaltung und Initialisierung

Ausgehende Verbindungsanfragen von Anwendungen werden vom Connection Management (Verbindungsverwaltungs-) Baustein verarbeitet, der Parameter der Anwendung in einem gemeinsamen Zustandsobjekt hinterlegt und gemeinsame Zustandsvariablen initialisiert. Anschließend erhalten alle Bausteine in der Schablone ein Initialisierungsereignis[1], sodass diese weitere Variablen initialisieren können. Neben Variablen können hier auch notwendige Puffer angelegt werden. Dieser Ablauf ist in Abbildung 4.10 veranschaulicht.

[1] Die Initialisierungsereignisse wurden in Abbildung 4.9 aus Gründen der Übersichtlichkeit weggelassen.

Empfangenen Nachrichten wird dabei durch den (De-)Serialisiererbaustein der passende Zustand zugeordnet[1], sodass allen Ereignissen und Nachrichten der jeweils richtige Zustand anhängt. Eine konkrete Umsetzung dieser Art der Zustandshaltung wird in Abschnitt 6.3 beschrieben.

Signalisiert die Anwendung die Bereitschaft Verbindungsanfragen entgegenzunehmen (bspw. durch die Erstellung eines Server-Sockets), ist der Ablauf zur Initialisierung ähnlich. Bei ankommenden Verbindungsanfragen über das Netzwerk ist jedoch der Verbindungsverwaltungsbaustein zunächst dafür verantwortlich, diesen Umstand der Anwendung zu signalisieren, bevor die Daten weiterverarbeitet werden.

4.3.2.3 Bausteine auf dem Datenpfad

In diesem Abschnitt werden die Bausteine auf dem Datenpfad beschrieben und Variationen diskutiert, die durch einen einfachen Austausch des Bausteins erzielt werden können.

Segmentierung / Reassemblierung

Durch den Segmentierungsbaustein wird festgelegt, ob und wie die Anwendungsdaten für die Übertragung in Dateneinheiten unterteilt werden. Bei einem Bytestrom-orientierten Transport wird in der Regel versucht, soviele Nutzdaten wie möglich in einer Dateneinheit zu übertragen, deren Größe auf eine netzabhängige, maximale Übertragungsgröße beschränkt ist (Maximum Segment Size, MSS). Dies kann eine kurzzeitige Pufferung der Anwendungsdaten erfordern, um die Dateneinheit zu füllen. In einigen Fällen können andere Strategien jedoch sinnvoller sein: Bei interaktiven Terminalsitzungen bspw. sollte jeder Tastendruck direkt gesendet werden, damit das Resultat innerhalb von einer Paketumlaufzeit im Terminal angezeigt werden kann. Andere Anpassungen sind nötig, wenn das Protokoll die Übertragung von Anwendungsdateneinheiten (Application Data Units, ADU) unterstützen soll. In diesem Fall muss der Segmentierer entsprechende Markierungen in der Nachricht setzen, die vom Serialisierer in das Paket übertragen werden. Analog zum Segmentierer muss das Gegenstück, die Reassemblierung, die Daten entsprechend wieder zusammensetzen.

Transmission Scheduling (Übertragungssteuerung)

Der Transmission Scheduling Baustein hat zwei wesentliche Aufgaben: Zum einen puffert er ausgehende Nachrichten im Sendewiederholungspuffer bis sie von der Gegenstelle quittiert werden. Zum anderen implementiert er die Strategie zur Übertragung der Nachrichten: z. B. Burst-artig oder gleichmäßig verteilt im Sendefenster (*pacing*).

[1] In der in Abschnitt 7.5 untersuchten Implementierung wird diese Zuordnung bereits durch ein anderes Protokoll vorgenommen.

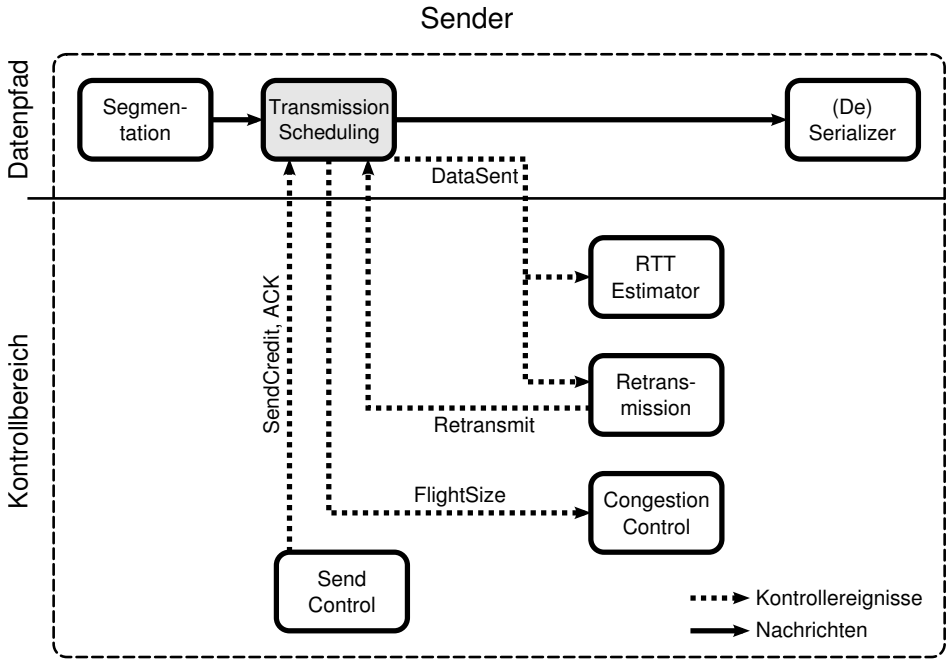


Abbildung 4.11 Kopplung des Transmission Scheduling Bausteins mit anderen Bausteinen über Kontrollereignisse (ohne Initialisierungsereignis).

Als Eingabe nutzt der Baustein folgende Kontrollereignisse (vgl. Abbildung 4.11):

- SendCredit – der durch die Sendesteuerung berechnete Sendekredit (Sendefenster), welcher in der Regel auf dem Minimum des Flussfensters (= Empfangsfenster der Gegenstelle) und des Staukontrollfensters basiert.

- ACK – Eintreffen einer Bestätigung. Empfangene Bestätigungen werden dazu genutzt, um bisher unbestätigte Nachrichten aus dem Sendewiederholungspuffer zu entfernen.

- Retransmit – das erneute Versenden von Nachrichten, wie es vom Sendewiederholungsbaustein entschieden wurde. In diesem Fall werden die zur Wiederholung vorgesehenen Pakete erneut versendet.

Hierbei ist von konkreten Transmission Scheduling Bausteinen zu beachten, dass beim Empfang von ACK und Retransmit-Ereignissen zunächst auf eine Aktualisierung des Sendekredits gewartet werden muss, bevor neue Nachrichten versendet werden dürfen. Dies wird in Abschnitt 4.3.2.5 näher beschrieben.

Von diesem Baustein können folgende Kontrollereignisse erzeugt werden:

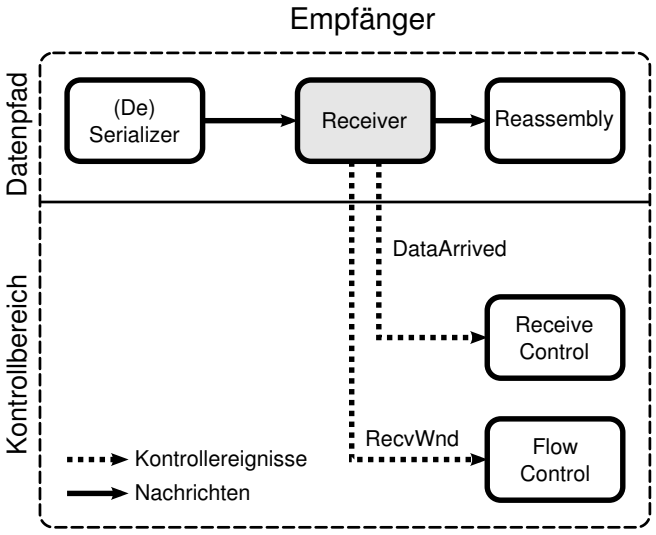


Abbildung 4.12 Kopplung des Receiver Bausteins mit anderen Bausteinen über Kontrollereignisse (ohne Initialisierungsereignis).

- DataSent – wird für jedes versandte Paket erzeugt und enthält dessen Sequenznummer und den Sendezeitpunkt. Dieses Ereignis wird vom RTT Estimation Baustein genutzt, um die aktuelle Umlaufzeit zu schätzen, wenn das dazugehörige ACK-Ereignis eintrifft, und vom Retransmission Baustein, um den Sendewiederholungszeitgeber zu starten.
- FlightSize – enthält die Anzahl der sich in der Übertragung befindlichen, unbestätigten Pakete und wird immer bei einer Änderung dieser Anzahl ausgelöst. Dieser Wert kann von Staukontrollverfahren genutzt werden, die in einem Congestion Control Baustein realisiert sind (z. B. NewReno [HFGN12]).

Receiver (Übertragungsempfänger)

Das Gegenstück zum Transmission Scheduling Baustein ist der Receiver Baustein. Seine Aufgaben sind allerdings unabhängig von der gewählten Transmission Scheduling Strategie, weswegen dieser in einem eigenen Bausteintyp resultiert. Seine Aufgaben sind:

- Erkennung von Duplikaten und Phantomdateneinheiten
- Erkennung von Paketverlusten
- Wiederherstellung der Nachrichtenreihenfolge

In der einfachsten Realisierung wird lediglich die nächste erwartete Nachricht an den nächsten Baustein weitergeleitet, alle anderen werden verworfen.

Alternativ können korrekt, aber nicht in der richtigen Reihenfolge empfangene Nachrichten solange im Empfangspuffer zwischengespeichert werden, bis die fehlenden Nachrichten eintreffen (was bspw. durch eine Wiederholungsanforderung geschieht).

Der Baustein versendet an die Bausteine im Kontrollbereich folgende Ereignisse (vgl. Abbildung 4.12):

- DataArrived – für jede empfangene Nachricht wird dieses Ereignis zusammen mit der entsprechenden Sequenznummer versendet.
- RecvWnd – die aktuelle Größe des Empfangsfenster.

Das Empfangsfenster stellt dabei die aktuelle Größe des freien Empfangspuffers zur Anwendung hin dar.

(De-)Serialisierer

Der Serialisierer erzeugt aus der zu versendenden Nachricht und den bis dahin zur Verfügung stehenden Informationen Pakete, die über das Netzwerk versendet werden können. Das bedeutet insbesondere, dass er das Paketformat, welches durch die Vokabularbeschreibung der Nachrichten festgelegt wird, umsetzt (vgl. Abschnitt 4.2.3). Der Deserialisierer bildet das entsprechende Gegenstück und trennt die Kontrolldaten aus dem Paketkopf von der Nachricht, die über weitere Bausteine Richtung Anwendung weitergeleitet wird. Wird durch konkrete Bausteine das Paketformat erweitert oder verändert, muss der (De-)Serialisierer angepasst werden.

Kontrollereignisse, die von diesem Baustein versendet werden und in Abbildung 4.9 aufgrund der Übersichtlichkeit nur durch allgemeine Pfeile dargestellt sind, werden für eintreffende Kontrollpakete erzeugt:

- SYN-Pakete werden als Kontrollereignisse an den Connection Management Baustein weitergereicht.
- ACK-Pakete werden als Kontrollereignisse an den Flow Control Baustein weitergereicht, der sie daraufhin an weitere Bausteine verteilt.
- SYNACK-Pakete werden als Kontrollereignis zunächst an den Connection Management Baustein weitergereicht, der die Initialisierung finalisiert und daraufhin ein ACK-Kontrollereignis an den Flow Control Baustein versendet. Diese Indirektion über den Connection Manager ist notwendig, um die Einhaltung der Reihenfolge sicherzustellen.

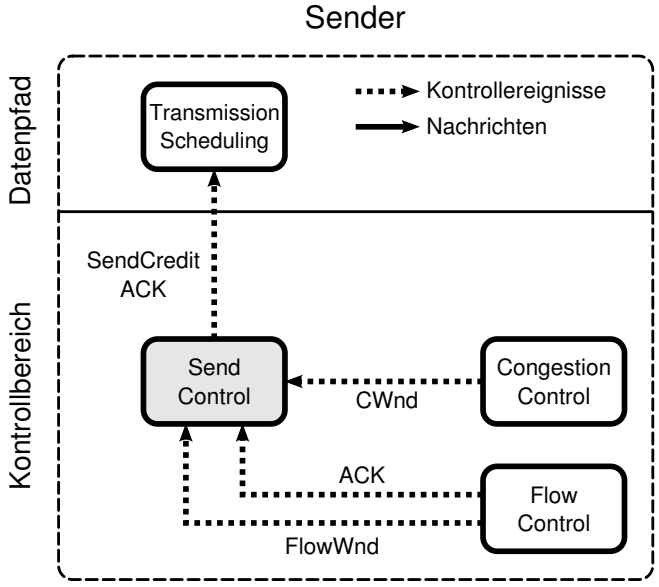


Abbildung 4.13 Kopplung des Send Control Bausteins mit anderen Bausteinen über Kontrollereignisse (ohne Initialisierungsereignis).

4.3.2.4 Bausteine im Kontrollbereich

Die Bausteine auf dem Kontrollpfad reagieren ausschließlich auf Kontroll- und Zeitgeberereignisse und versenden Kontrollereignisse an andere Bausteine. Sie verarbeiten Anwendungsnachrichten nicht direkt sondern nehmen lediglich Einfluss auf die entsprechenden Bausteine auf dem Datenpfad.

Send Control (Sendesteuerung)

Die Sendesteuerung ermittelt die aktuelle Größe des Sendekredits (SendCredit), welches in der Regel aus dem Minimum von Staukontrollfenster (CWnd) und Flussfenster (FlowWnd) gebildet wird, und leitet diesen an den Transmission Scheduling Baustein weiter (vgl. Abbildung 4.13). Der Sendekredit muss dabei minimal 1 sein, um Deadlock-Situationen aufgrund einer Flussfenstergröße von 0 zu vermeiden.[1] Informationen zu bestätigten Nachrichten (ACK) müssen ebenfalls an den Transmission Scheduling Baustein weitergereicht werden, damit dieser die zur Wiederholung gepufferten Nachrichten verwerfen kann.

Bevor der Sendekredit und die Bestätigung jedoch an den Transmission Scheduling Baustein weitergereicht werden können, sind gewisse Voraussetzung nötig, die in Abschnitt 4.3.2.5 näher beschrieben werden.

[1] Bei TCP (vgl. Abschnitt 2.2.2.1) ist ein Sendekredit von 0 möglich, allerdings darf der Sender auch in diesem Fall mindestens ein Byte versenden. Bei der Umsetzung hier wird diese Situation über den Sendekredit abgebildet.

Receive Control (Empfangssteuerung)

Die Empfangssteuerung (Receive Control) realisiert das Bestätigungsverhalten. Mit alternativen Realisierungen können durch einen einfachen Austausch des Bausteins folgende Verhalten realisiert werden:

- sofortige, kumulierte Bestätigung (Acknowledgment, ACK) der letzten, in der richtigen Reihenfolge empfangenen Nachricht
- sofortige Bestätigung jeder empfangenen, aber noch nicht bestätigten Nachricht (Selective Acknowledgment, SACK)
- negative Bestätigungen (NACKs) können hier ebenfalls realisiert werden; allerdings ist dann auch eine weitere Anpassung des Senders erforderlich (vgl. Abschnitt 4.3.3.3)

Informationen zu allen empfangenen Nachrichten (auch den Duplikaten) erhält der Baustein vom Receiver Baustein. Bei Bedarf erzeugt der Receive Control Baustein daraufhin ein Kontrollereignis für eine *ausgehende* Bestätigung, welches zunächst an den Flow Control Baustein gesendet wird, da dieser noch Informationen zum aktuellen Empfangsfenster hinzufügen muss.

Flow Control (Flusskontrolle)

Der Flusskontrollbaustein realisiert das Verfahren zur Flusskontrolle und ist dabei in einen Empfänger- und in einen Senderteil unterteilt. Im Empfänger wird das aktuelle Empfangsfenster (RecvWnd) mit einer zu versendenden ACK-Nachricht gemeinsam verschickt. Auf Seite des Senders der Daten wird die Information über das Empfangsfenster aus der ACK-Nachricht extrahiert und als Flussfenster (FlowWnd) an die Sendekontrolle weitergereicht. Zusätzlich wird die ACK-Nachricht als Ereignis an andere Bausteine im Kontrollbereich gesandt.

RTT Estimation (Schätzer für Umlaufzeit)

Dieser Baustein hat die Aufgabe, die aktuelle Paketumlaufzeit zu schätzen. Zur Schätzung sind Messungen notwendig, die bspw. aus den Zeitstempeln versandter Nachrichten und dem Zeitpunkt des Eintreffens der dazugehörigen Bestätigungen berechnet werden können (wozu die beiden Kontrollereignisse DataSent und ACK verwendet werden, vgl. Abbildung 4.14). Alternativ sind auch Messungen durch dedizierte Kontrollnachrichten möglich. Zusätzlich wird hier auch der Zeitgeber für notwendige Übertragungswiederholungen bestimmt (Retransmission Timeout, RTO) und dessen Wert an den Retransmission Baustein versendet. Die Art und Weise der Messung und Glättung der Umlaufzeit sowie die Bestimmung des initialen und aktuellen RTO kann durch einen Austausch dieses Bausteins angepasst werden.

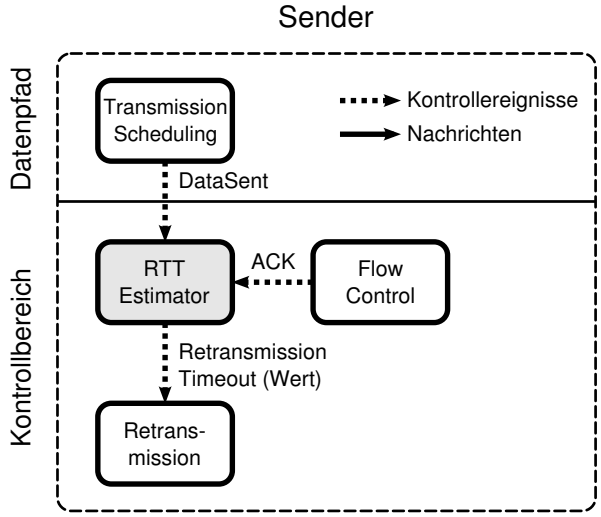


Abbildung 4.14 Kopplung des Bausteins zur Schätzung der Paketumlaufzeit mit anderen Bausteinen über Kontrollereignisse (ohne Initialisierungsereignis).

Congestion Control (Staukontrolle)

Mit einem Staukontrollbaustein wird das konkrete Staukontrollverfahren realisiert, welches die aktuelle Größe des Staukontrollfensters (CWnd) berechnen muss. Zusätzlich gehören Entscheidungen zum Fast Retransmit und zum Zurücksetzen des Retransmission Timeouts für Fast Recovery dazu. Die empfangenen und versendeten Kontrollereignisse sind (Abbildung 4.15):

- ACK – Empfangene Bestätigungsnachrichten werden zum Self-Clocking verwendet, d. h. das Staukontrollverfahren wird immer bei empfangenen Bestätigungen aufgerufen und das Staukontrollfenster wird angepasst.

- FlightSize – Dieser Wert kann für einige Verfahren in die jeweiligen Berechnungsvorschriften mit einfließen [AlPB09] und bezeichnet die Anzahl gesendeter, bisher nicht bestätigter Pakete. Ein entsprechendes Ereignis wird vom Transmission Scheduling Baustein erzeugt.

- Timeout – Der Zeitgeber zur Übertragungswiederholung wird vom Retransmission Baustein gestartet und das entsprechende Zeitgeberereignis trifft auch bei ihm wieder ein. Da bei einem solchen Timeout allerdings auch Anpassungen am Staukontrollfenster nötig sein können (z. B. für Slow Start), muss der Congestion Control Baustein über den eingetretenen Timeout informiert werden.

- CWnd – Die aktuelle Größe des Staukontrollfensters muss von einem Staukontrollbaustein bestimmt und an den Send Control Baustein ver-

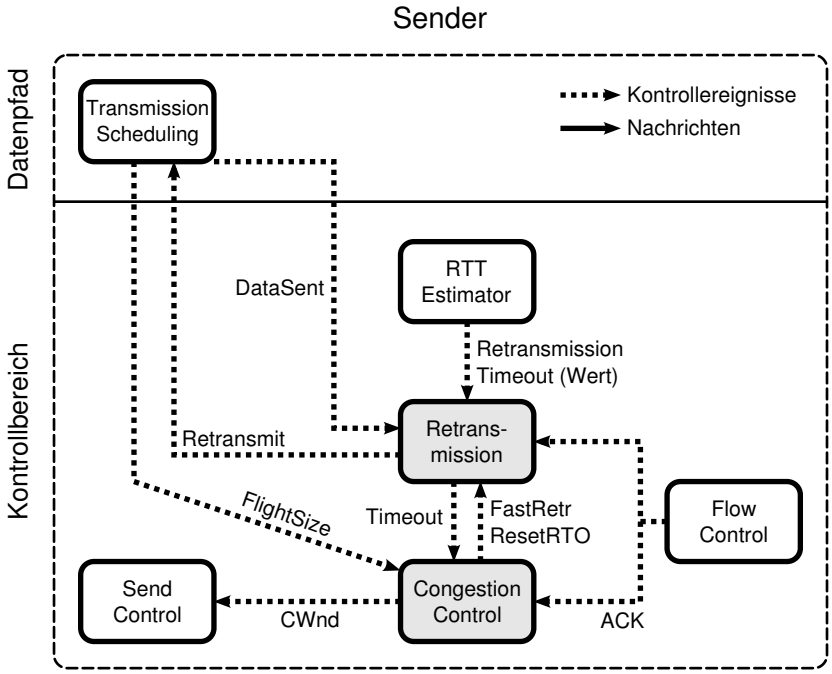


Abbildung 4.15 Kopplungen des Congestion Control und des Retransmission Bausteins mit anderen Bausteinen über Kontrollereignisse (ohne Initialisierungsereignis).

sandt werden, der daraufhin die aktuelle Größe des Sendekredits bestimmen kann.

- FastRetransmit – Als Reaktion auf mehrfache, doppelte Bestätigungen für das selbe Paket kann ein Fast Retransmit angestoßen werden. Um doppelte Sendewiederholungen zu vermeiden (bspw. weil der Zeitgeber zur Sendwiederholung nahezu zeitgleich mit einem FastRetransmit ausgelöst wird), muss das FastRetransmit Ereignis über den Sendewiederholungsbaustein geleitet werden. Da bei ihm beide Ereignisse eintreffen, dient er an dieser Stelle zur Synchronisation.

- ResetRTO – Während des Fast Recovery kann es notwendig werden, den laufende Sendewiederholungszeitgeber neuzustartet [HFGN12]. In diesem Fall kann der Staukontrollbaustein ein entsprechendes Kontrollereignis an den Sendewiederholungsbaustein senden.

Die Kopplung des Staukontrollbausteins über diese Kontrollereignisse mit anderen Bausteinen erlauben Staukontrollverfahren wie sie für TCP vorgeschlagen wurden. Für alternative Verfahren sind Änderungen an der Schablone notwendig, was in Abschnitt 4.3.3.3 beschrieben wird.

Retransmission (Sendewiederholung)

Dieser Baustein veranlasst Sendewiederholungen, indem das Kontrollereignis Retransmit an den Transmisson Scheduling Baustein gesandt wird. Folgende Kontrollereignisse dienen dabei als Eingabe (vgl. Abbildung 4.15):

- DataSent – Versand eines (bisher unbestätigten) Paketes. Diese Informationen dienen dem Retransmission Baustein dazu, einen Zeitgeber zu starten. Wenn dieser abgelaufen ist, ohne dass Bestätigungen der versendeten Pakete eingetroffen sind, wird eine Sendewiederholung veranlasst.
- ACK – Bestätigung zum erfolgreichen Empfang eines oder mehrerer Pakete. Der entsprechende Sendewiederholungszeitgeber wird zurückgesetzt.
- RTO – Wert des Sendewiederholungszeitgebers basierend auf den Schätzungen des RTT Estimation Bausteins.
- FastRetransmit – explizite Veranlassung einer Sendewiederholung durch den Staukontrollbaustein.
- ResetRTO – Veranlassung des Neustarts eines laufenden Sendewiederholungszeitgebers durch den Staukontrollbaustein.

Der Retransmission Baustein führt eine Liste mit Informationen über unbestätigte Pakete, die er mit dem Ereignis DataSent vom Transmission Scheduling Baustein mitgeteilt bekommt. Für diese Pakete startet er einen Zeitgeber (Retransmission Timeout), nach dem eine Sendewiederholung stattfinden soll. Sollte bis zum Ablauf des Retransmission Timeout keine Bestätigung für die Pakete eingegangen sein, wird die erneute Übertragung durch den Transmission Scheduling Baustein mit einem Retransmit-Ereignis angefordert. Zusätzlich wird der Congestion Control Baustein über das Ereignis „Timeout“ informiert, damit dieser eventuell notwendige Anpassungen am Staukontrollfenster vornehmen kann. Wie beim Empfang einer Bestätigung ist auch im Fall einer Übertragungswiederholung auf die richtige Reihenfolge der internen Ereignisse zu achten. Hierauf geht der folgende Abschnitt näher ein.

4.3.2.5 Synchronisation von Ereignissen

Die Verarbeitung von Bestätigungen (ACKs) und von Sendewiederholungsereignissen erfordert, dass einige Informationen, die von unterschiedlichen Bausteinen bereitgestellt werden, berechnet sind bevor der Transmission Scheduling Baustein weitere Nachrichten versenden kann. Da die Ereignisverarbeitung in der Regel asynchron durch die einzelnen Bausteine geschieht, muss eine notwendige Synchronisation zur Einhaltung der Reihenfolge der

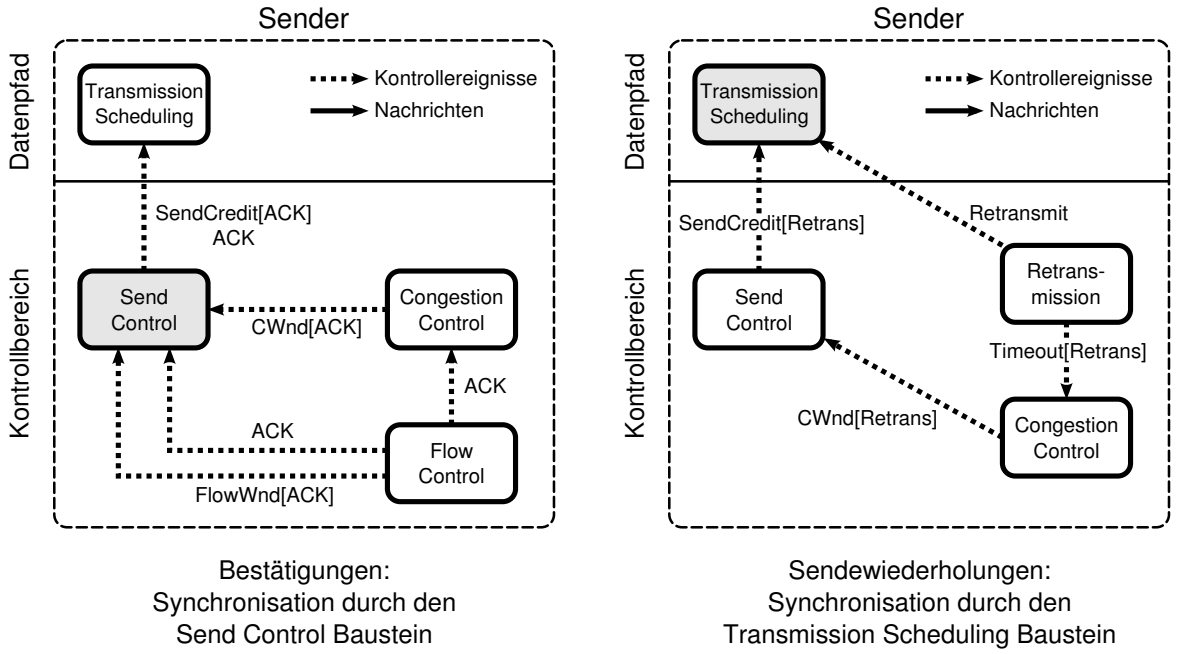


Abbildung 4.16 Synchronisation bei der Verarbeitung von Ereignissen. Links: Bei eintreffenden Bestätigungen wartet der Send Control Baustein auf die Werte für das Staukontrollfenster und das Flussfenster. Rechts: Bei anstehenden Übertragungswiederholungen wartet der Transmission Scheduling Baustein auf den aktualisierten Sendekredit.

Berechnungen durch die Bausteine realisiert werden. In der hier vorgestellten Schablone geschieht dies wie folgt (Abbildung 4.16):

- Im Falle von empfangenen Bestätigungen muss der Send Control Baustein zunächst darauf warten, dass sowohl das Staukontrollfenster als auch das Flussfenster aktualisiert werden, bevor der Sendekredit bestimmt werden kann und der Transmission Scheduling Baustein darüber informiert wird. Dazu wird an diesen Ereignissen ein Verweis an das dazugehörige ACK-Ereignis angehängt, über das der Send Control Baustein die Zuordnung (Synchronisation) der Ereignisse vornimmt. Erst wenn alle drei Ereignisse (ACK, FlowWnd[ACK] und CWnd[ACK]) beim Send Control Baustein eingetroffen sind, wird die Information an den Transmission Scheduling Baustein weitergereicht.

- Im Falle von Übertragungswiederholungen aufgrund eines Timeouts erfolgt die Synchronisation der Ereignisse im Transmission Scheduling Baustein. Auch hier wird ein Vermerk zur Sendwiederholung an die entsprechenden Ereignisse (CWnd[Retrans], SendCredit[Retrans]) angehängt.

4.3.3 Variationen der Schablone

Wie in den beiden Abschnitten 4.3.2.3 und 4.3.2.4 diskutiert, sind Anpassungen des Transportprotokolls bereits durch den Austausch von Bausteinen möglich, die jeweils vom gleichen Typ abgeleitet werden können. Für Anpassungen, die eine Änderung der Kopplung der Bausteine untereinander nach sich ziehen – bspw. weil neue Bausteintypen gefordert sind oder neue Ereignisse notwendig sind – muss die Schablone selbst verändert werden. Dazu werden in diesem Abschnitt die folgenden drei Beispiele beschrieben:

- ECN – explizite Benachrichtigungen über Stausituationen im Netz
- DCCP – keine Flusskontrolle und keine Übertragungswiederholung
- UDT-Staukontrolle – empfängerseitige Staukontrolle

Während ECN und DCCP nur kleine Anpassungen erfordern, sind die Anpassungen für die UDT-Staukontrolle am umfangreichsten. Trotzdem wird auch hier gezeigt, dass die Anpassungen durch die Protokollschablone strukturiert durchgeführt werden können und so ohne viel Aufwand eine weitere Schablone aus der existierenden ableitbar ist.

4.3.3.1 Explicit Congestion Notification (ECN)

Mittels ECN [RaFB01] können Zwischensysteme im Netz wie Router bei drohenden Stausituationen Pakete mit einem Congestion-Experienced-Flag (CE) markieren. Der Empfänger der Pakete sendet diese Information an den Sender mit den Bestätigungsnachrichten zurück (ECE – ECN Echo). Der Sender kann daraufhin sein Staukontrollfenster reduzieren. Eine Problematik bei der TCP/IP-Architektur ist dabei, dass die notwendigen Informationen jeweils aus unterschiedlichen Schichten stammen: Router setzen dazu das CE-Flag im IP-Kopf, welches der Empfänger auswerten muss und im TCP-Kopf als ECE zurück an den Sender senden muss. Der Sender signalisiert dem Empfänger daraufhin im TCP-Kopf, dass aufgrund des ECE-Flags das Staukontrollfenster reduziert wurde (CWR – Congestion Window Reduced). Die Herausforderungen von ECN sind hier:

- Der Austausch von Informationen ist schichtenübergreifend notwendig, da die Empfängerinstanz des Transportprotokolls auf das CE-Flag des Vermittlungsprotokoll reagieren muss.
- Es müssen zusätzliche Informationen zwischen Sender- und Empfängerinstanz des Transportprotokolls ausgetauscht werden.

Für den ersten Punkt sind entsprechende Kontrollinformationen zu definieren, die von einem Protokoll unterhalb der Transportprotokollschablone gesetzt und an empfangene Nachrichten angehängt werden. Im Gegensatz zu

den bisher diskutierten Beschreibungen werden diese Definitionen auch außerhalb der Schablone genutzt und dienen dem *schablonenübergreifenden* Informationsaustausch. Alternativ kann in solchen Fällen auch ein Ereignis definiert werden, das von dem Protokoll unterhalb der Transportprotokollschablone definiert wird und auf das sich die Transportprotokollschablone registriert.

Für den zweiten Punkt sind Anpassungen am Paketformat des Protokolls notwendig. Insgesamt gestalten sich die Änderungen innerhalb der Transportprotokollschablone wie folgt:

- Auf Empfängerseite müssen das DataArrived- und das ACK-Ereignis um ein entsprechendes CE-Flag erweitert werden. Der Receiver-Baustein fügt dieses CE-Flag dem DataArrived-Ereignis hinzu, wenn die empfangene Nachricht eine entsprechende Kontrollinformation der darunterliegenden Protokolle aufweist. Der Receive Control Baustein überträgt diese Information in das ACK-Ereignis.
- Der (De-)Serialisierer muss angepasst werden, um das ECE- und das CWR-Flag im Paketkopf des Transportprotokolls kodieren zu können. Auf Senderseite muss er das ACK-Ereignis um das ECE-Flag ergänzen, wenn das Flag vom Empfänger gesetzt wurde.
- Auf Senderseite muss der Congestion Control Baustein verändert werden, damit er auf Basis des ECE-Flag das Staukontrollfenster entsprechend anpasst. Diese Anpassung muss als CWR-Flag über die erweiterten Ereignisse CWnd und SendCredit an den Transmission Scheduling Baustein weitergereicht werden, der beim nächsten zu versendenden Paket den Serialisierer instruiert, das CWR-Flag mitzusenden.

Zusammenfassend muss also lediglich eine schablonenübergreifende Kontrollinformation, die an empfangene Pakete angehängt wird, neu definiert werden. Weitere Änderungen sind durch Erweiterungen der existierenden Ereignisse gegeben, sowie durch angepasste Bausteine, die mit diesen Informationen umgehen können.

4.3.3.2 DCCP-ähnliches Protokoll

Das Datagram Congestion Control Protokoll (DCCP) [KoHF06] realisiert ein unzuverlässiges Transportprotokoll mit Staukontrolle. Dabei können verschiedene Staukontrollverfahren integriert werden, die sich bspw. in der Schätzung der Paketumlaufzeiten oder im Bestätigungsverhalten unterscheiden. Die Herausforderungen, die zu einer Änderung der Schablone führen, sind:

- Zuvor als Bausteintypen definierte Mechanismen entfallen, sodass der Kontrollfluss der Ereignisse angepasst werden muss.

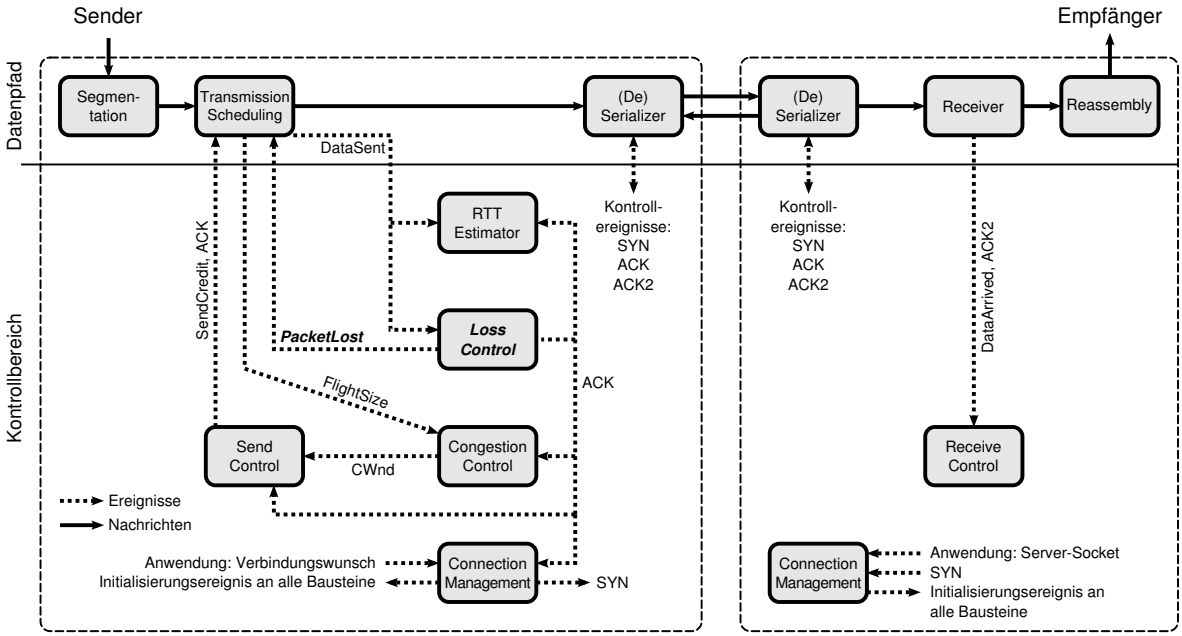


Abbildung 4.17 Protokollschablone nach Anpassung für DCCP.

- Es wird eine zusätzliche Kontrollnachricht (ACK2, s. u.) eingeführt.

Die notwendigen Änderungen an der Schablone für DCCP sind in Abbildung 4.17 skizziert:

- Der Baustein zur Flusskontrolle entfällt.
- Der Retransmission Baustein wird durch einen Loss Control Baustein (Verlustkontrolle) ersetzt.
- An Stelle eines Ereignisses zur Auslösung einer Übertragungswiederholung wird lediglich ein festgestellter Paketverlust an den Transmission Scheduling Baustein gesendet (d. h. das Retransmit-Ereignis wird durch ein PacketLost-Ereignis ersetzt). Zusammen mit dem ACK-Ereignis kann der Transmission Scheduling Baustein so die Anzahl gesendeter, unbestätigter Datenpakete kontinuierlich anpassen und ggfs. weitere Dateneinheiten senden, solange es das Staukontrollfenster erlaubt.
- Da DCCP eine Option besitzt, mit der Bestätigungsnachrichten zuverlässig zugestellt werden können, wird eine ACK2-Nachricht hinzugefügt, deren Versand vom Send Control Baustein beim Empfang einer ACK-Nachricht ausgelöst wird.

Zudem sind alternative konkrete Bausteine für den Transmission Scheduling Baustein und für den Receiver Baustein nötig, da beide keine Pufferung der zu sendenden bzw. der empfangenen Nachrichten vornehmen müssen. Die

Bausteine zum Connection Management und zur (De-)Serialisierung müssen ebenfalls durch andere konkrete Bausteine ausgetauscht werden, die auf DCCP zugeschnitten sind. Die Bausteine zur Segmentierung und Reassemblierung sind optional.

Obwohl daraus eine neue Protokollschablone für unzuverlässige Transportprotokolle mit Staukontrolle resultiert, sind einige der Ereignisse und Bausteintypen wiederverwendbar. Im Vergleich zu DCCP, das für unterschiedliche Staukontrollverfahren unterschiedlich konfiguriert werden muss, können mit der Protokollschablone zur Entwurfszeit bereits die passenden Bausteine ausgewählt werden, sodass in der Implementierung keine Unterscheidungen und keine Aushandlungen mehr notwendig sind.

4.3.3.3 UDT-Staukontrolle

Die Besonderheit von UDTs Staukontrolle ist, dass sie empfängerseitig arbeitet (vgl. Abschnitt 3.2.1). Dazu wird vom Empfänger (anstatt vom Sender) die aktuelle Paketumlaufzeit geschätzt, welche wiederum genutzt wird, um das Intervall periodischer ACK- und NACK-Nachrichten zu bestimmen. Mit diesen Nachrichten werden dem Sender der aktuelle Sendekredit, die gewünschte Senderate sowie die aktuell geschätzte Paketumlaufzeit mitgeteilt. Die zu sendenden Bestätigungsnachrichten sind somit nicht länger von der Datenrate abhängig, sondern nur noch von der Verzögerung zwischen den beiden Kommunikationsteilnehmern. Die geschätzte Paketumlaufzeit wird dabei vom Sender benötigt, um einen Retransmission Timeout zu bestimmen. Zur empfängerseitigen Schätzung der Paketumlaufzeiten führt UDT eine ACK2-Nachricht ein, die vom Sender an den Empfänger sofort beim Empfang einer ACK-Nachricht versandt wird.

Die Herausforderung, durch die eine Änderung der Protokollschablone notwendig wird, ist die gänzlich andere Funktionsweise der Staukontrolle. Die notwendigen Änderungen an der Protokollschablone sind in Abbildung 4.18 dargestellt. Neben den neuen Funktionalitäten auf Empfängerseite muss auch die Senderseite modifziert werden. Dies geschieht durch den Austausch einiger Bausteine, um die neuen Ereignisse und das geänderte Verhalten zu realisieren. Insbesondere bedeutet dies:

- Der RTT Estimation Baustein zur Bestimmung der Paketumlaufzeit auf Senderseite wertet nun lediglich die vom Empfänger gesendete Paketumlaufzeit aus und bestimmt damit den Retransmission Timeout (RTO).

- Der Send Control und der Transmission Scheduling Baustein müssen zusätzlich die vom Empfänger ermittelte Senderate auswerten und beim Senden berücksichtigen.

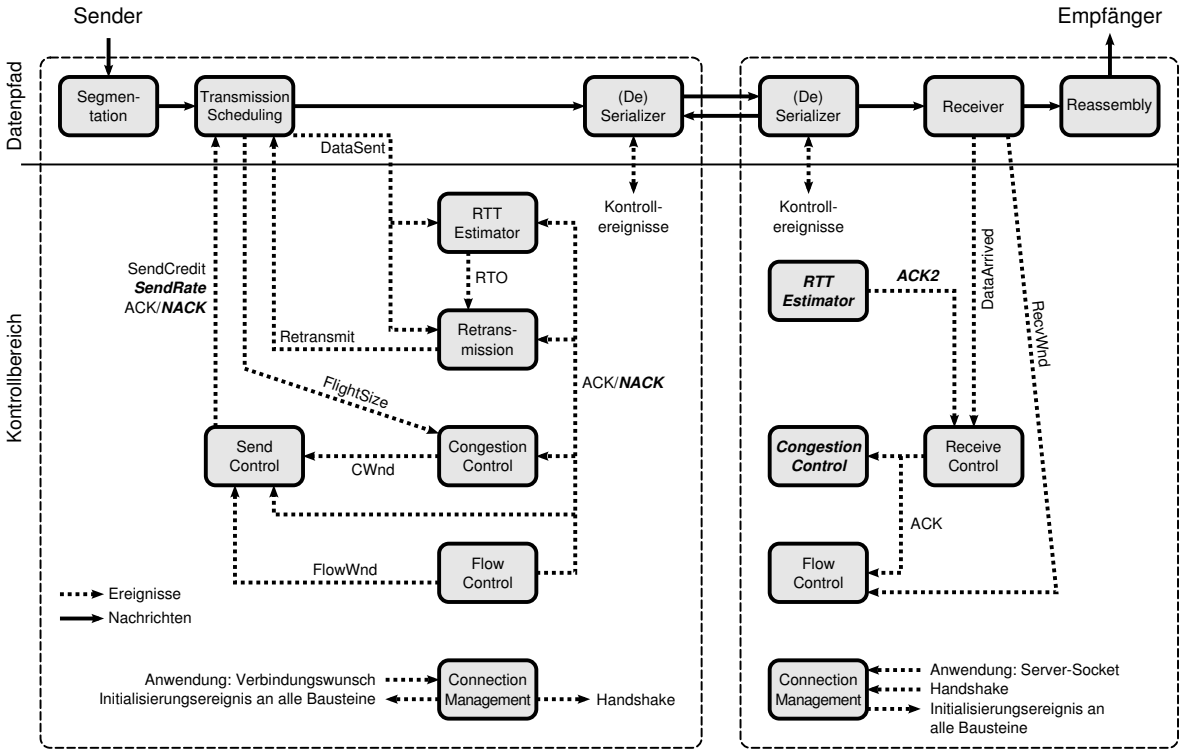


Abbildung 4.18 Protokollschablone nach Anpassung für UDT.

Die Bausteine zur Flusskontrolle und zur Sendewiederholung müssen nur geringfügig angepasst werden, um die neuen Ereignisse zu unterstützen.

Die Staukontrolle auf Empfängerseite nutzt zur Bestimmung der Senderate eine Schätzung der aktuellen Empfangsrate und der aktuell verfügbaren Bandbreite auf Basis von Packet Pairing (also auf Basis des zeitlichen Abstands zwischen zwei empfangenen Paketen, vgl. Abschnitt 3.2.1).

Um die Staukontrolle von UDT zu realisieren, muss die ursprüngliche Schablone deutlich angepasst werden. Dennoch ist die Erstellung dieser Schablone inkrementell auf Basis der existierenden möglich, und viele Bausteintypen können unverändert wiederverwendet werden. Einige Änderungen, wie beispielsweise die Berücksichtigung einer Senderate, können sogar wieder zurück in die ursprüngliche Schablone übernommen werden, da es sich dabei um Ereignisse handelt, die bei Bedarf weggelassen werden können.

Denkbar ist hier auch die Rückführung weiterer Änderungen in die ursprüngliche Schablone (z. B. des ACK2-Ereignisses), um diese umfassender zu gestalten, sodass das gewünschte Verhalten durch eine einfache Konfiguration der Schablone erreicht werden kann. Dies hat allerdings den Nachteil, dass die Schablone an Komplexität gewinnt und weitere Anpassungen schwieriger werden. Aus diesem Grund wird hier vorgeschlagen, unterschiedliche Schablonen beizubehalten, wobei einzelne Bausteintypen jedoch gemeinsam in beiden Schablonen verwendet werden können.

4.3.4 Bewertung

In diesem Abschnitt wurde gezeigt, dass sorgfältig entworfene Protokollschablonen bereits Anpassungen durch einen einfachen Austausch von Bausteinen erlauben, wobei der jeweilige Bausteintyp und die verwendeten Schablonen-Ereignisse unverändert bleiben. Die Schablone erfordert dabei eine ausführliche Dokumentation nicht nur der Aufgabe eines jeden Bausteintyps, sondern auch der jeweils verwendeten Zustandsvariablen und Ereignisse, die bei Bedarf durch Spezialisierungen erweiterbar sind.

Weiterreichende Anpassungen können dabei eine Modifikation der Schablone mit sich bringen, wie an den Beispielen von DCCP und UDT gezeigt wurde. Trotzdem kann dabei jeweils ein Großteil der existierenden Bausteintypen und Ereignisse wiederverwendet werden, oder sie sind als Ausgangspunkt für Anpassungen nutzbar. Diese Wiederverwendbarkeit ist ein wichtiger Bestandteil des Lebenszyklus, der in Abbildung 1.3 auf Seite 5 dargestellt ist.

Protokollschablonen mit deren Bausteintypen, Kontrollereignissen und Zustandsinformationen erlauben eine gezielte Anpassung der betreffenden Teile des Protokolls. Somit werden Komplikationen durch Erweiterungen existierender Protokolle, wie sie bspw. in [Math12] beschrieben werden, verringert (vgl. Abschnitt 3.2.1).

Eine Implementierung der Transportprotokollschablone wurde für das in Kapitel 6 beschriebene Rahmenwerk durchgeführt [Fran12]. Das Laufzeitverhalten dieser Implementierung wird in Abschnitt 7.5 untersucht, sowohl in einem Simulator als auch auf einem realen System.

4.4 Verwendung von Modellierungswerkzeugen

Die in Abschnitt 4.2.3 eingeführte Beschreibung von Protokollkomponenten kann bereits genutzt werden, um die Implementierung der Komponenten zu vereinfachen. So können bspw. Code-Teile für die Datenstrukturen der Nachrichten und der Zustandsvariablen, sowie Stub-Code für die Ereignisverarbeitung generiert werden. Hierzu eignen sich Modellierungswerkzeuge, die das Erstellung entsprechender Generatoren deutlich vereinfachen. Mit dem Eclipse Modelling Framework (EMF [SBPM09]) und Erweiterungen wie *xtext* und *xtend* ist es bspw. möglich, dass zunächst ein internes, EMF-spezifisches Modell aus der maschinenlesbaren Beschreibung der Datenstrukturen des Bausteins erzeugt wird, über welches dann ein Generator eine andere Repräsentation erzeugen kann. Eine solche Repräsentation ist dann bspw. Programm-Code für die Zielplattform. Der Ablauf mit dieser Werkzeugsammlung ist in Abbildung 4.19 skizziert: Zunächst ist es notwen-

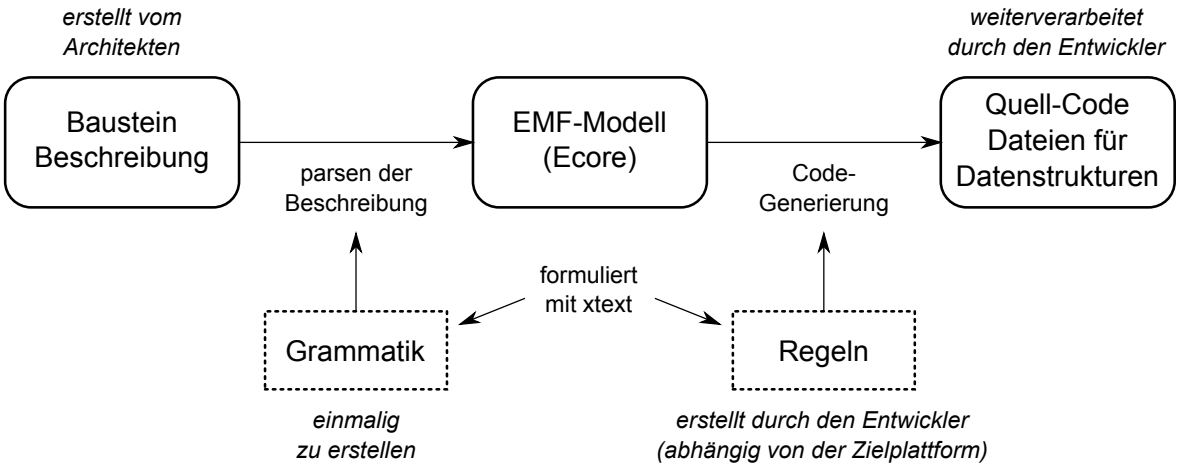


Abbildung 4.19 Ablauf zur Generierung von Datenstrukturen und Code-Fragmenten.

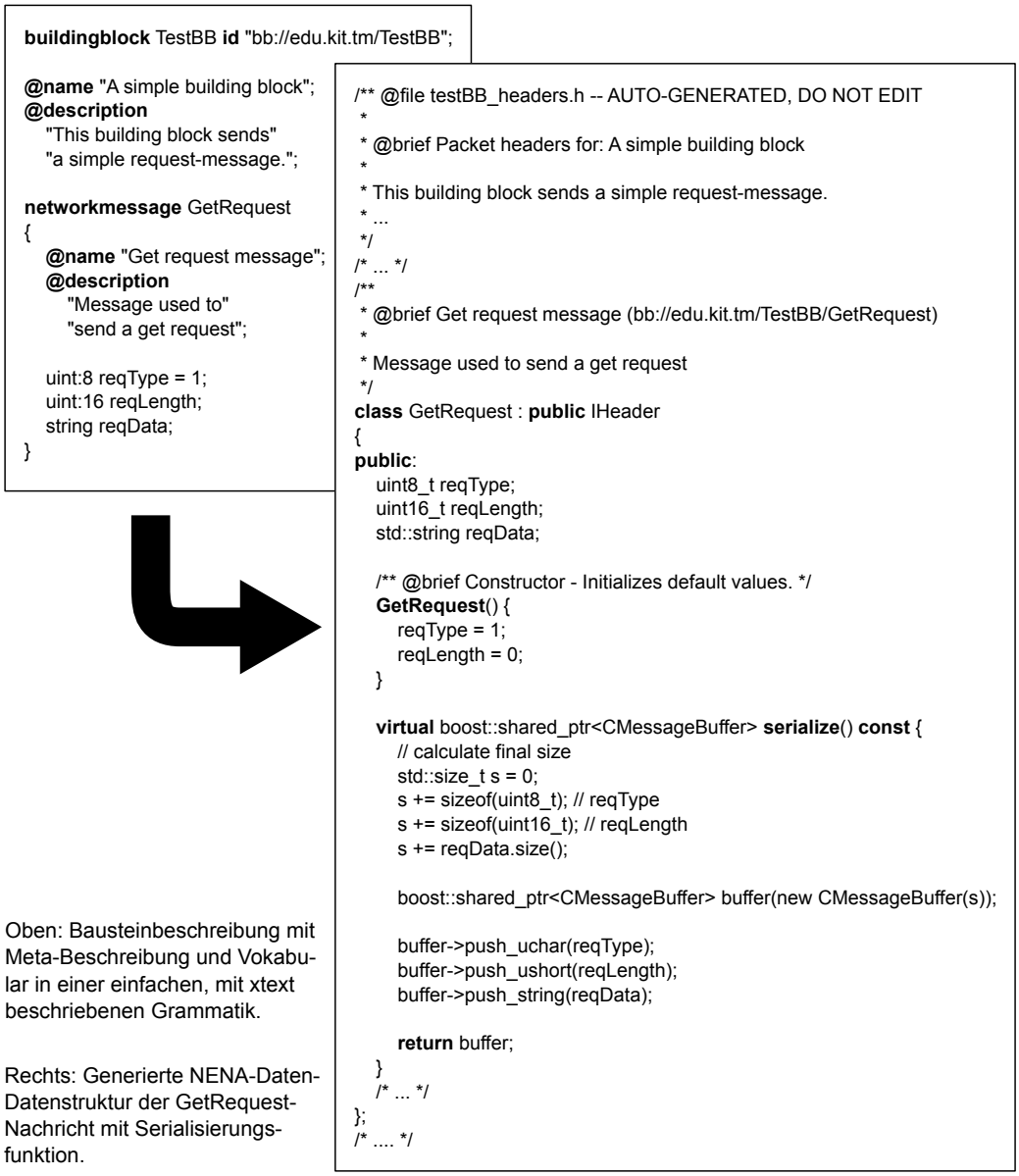
dig, dass eine Grammatik formuliert wird, mit der die maschinenlesbare Beschreibung in ein EMF-Modell umgewandelt wird. Diese Grammatik muss nur einmal für die maschinenlesbare Beschreibung erstellt werden.[1] Unterschiedliche Zielplattformen erfordern ggfs. unterschiedlichen Programm-Code, weswegen ein jeweils passender Generator für jede Zielplattform notwendig ist. Ein einfaches Beispiel einer so generierten Datenstruktur wird in Abbildung 4.20 gezeigt.

Das Verhalten des Protokollmechanismus muss jedoch nach wie vor vom Entwickler implementiert werden. Dieser richtet sich dabei nach der Dokumentation, die vom Architekten gegeben ist, und beinhaltet Reaktionen auf eingehende Nachrichten, Kontrollereignisse und Zeitgeberereignisse. Da diese Ereignisse bereits Teil der Vokabular-Beschreibung sind, können bereits Code-Fragmente generiert werden (sogenannter Stub-Code), sodass der Entwickler lediglich sog. Handler-Routinen implementieren muss.

4.5 Zusammenfassung

In diesem Kapitel wurde der Entwurf von komponentenbasierten Protokollen mittels Protokollschablonen vorgestellt. Dabei wurde zunächst auf die allgemeine Beschreibung des Entwicklungsprozesses zusammen mit dessen beteiligten Akteuren und Abstraktionsebenen eingegangen. Anschließend wurde die notwendige Beschreibung von Protokollbausteinen detailliert, mit der auch die automatische Generierung von Code-Teilen – insbesondere betreffend den Datenstrukturen – möglich ist. Das Prinzip der Protokollschablone

[1] Mit einem domänenspezifischen Ansatz wird in der Regel die domänenspezifische Sprache bereits über eine Grammatik oder ein Modell formuliert. Diese Grammatik oder das Modell kann hier wiederverwendet werden.



```
buildingblock TestBB id "bb://edu.kit.tm/TestBB";

@name "A simple building block";
@description
   "This building block sends"
   "a simple request-message.";

networkmessage GetRequest
{
   @name "Get request message";
   @description
      "Message used to"
      "send a get request";

   uint:8 reqType = 1;
   uint:16 reqLength;
   string reqData;
}
```

```
/** @file testBB_headers.h -- AUTO-GENERATED, DO NOT EDIT
 *
 * @brief Packet headers for: A simple building block
 *
 * This building block sends a simple request-message.
 * ...
 */
/* ... */
/**
 * @brief Get request message (bb://edu.kit.tm/TestBB/GetRequest)
 *
 * Message used to send a get request
 */
class GetRequest : public IHeader
{
public:
   uint8_t reqType;
   uint16_t reqLength;
   std::string reqData;

   /** @brief Constructor - Initializes default values. */
   GetRequest() {
      reqType = 1;
      reqLength = 0;
   }

   virtual boost::shared_ptr<CMessageBuffer> serialize() const {
      // calculate final size
      std::size_t s = 0;
      s += sizeof(uint8_t); // reqType
      s += sizeof(uint16_t); // reqLength
      s += reqData.size();

      boost::shared_ptr<CMessageBuffer> buffer(new CMessageBuffer(s));

      buffer->push_uchar(reqType);
      buffer->push_ushort(reqLength);
      buffer->push_string(reqData);

      return buffer;
   }
   /* ... */
};
/* .... */
```

Oben: Bausteinbeschreibung mit Meta-Beschreibung und Vokabular in einer einfachen, mit xtext beschriebenen Grammatik.

Rechts: Generierte NENA-Daten-Datenstruktur der GetRequest-Nachricht mit Serialisierungsfunktion.

Abbildung 4.20 Beispiel zur Generierung von Datenstrukturen mit xtext.

wurde dann am ausführlichen Beispiel einer Protokollschablone für zuverlässige Transportprotokolle erläutert. Anhand kleiner Anpassungen, die lediglich den Austausch einzelner Bausteine vom gleichen Bausteintyp notwendigen machen, konnte bereits eine Vereinfachung für Anpassungen gezeigt werden. Komplexere Änderungen wurden durch Modifikationen der Protokollschablone erreicht, die den Wiederverwendungswert existierender Schablonen und Bausteine zeigen.

5. Anwendungsschnittstelle

Eine wesentliche Voraussetzung für die Entkopplung von Anwendungen und Kommunikationsprotokollen ist eine geeignete Abstraktion von Kommunikationsdiensten zur Nutzung durch die Anwendung bzw. durch den Anwendungsprogrammierer. Im folgenden Abschnitt wird zunächst auf Basis existierender Studien untersucht, welche Arten von Anwendungsverkehr im heutigen Internet am verbreitetsten sind. Anschließend werden die Entwurfsziele und Anforderungen an die in dieser Arbeit entworfene Anschwendungsschnittstelle begründet, und schließlich deren Entwurf beschrieben. Neben Beispielen zur Verwendung für verschiedene Kommunikationsparadigmen wird auch die Implementierung der Schnittstelle als Bibliothek beschrieben.

5.1 Anwendungen heute

Untersuchungen [MFPA09, LIJMO+10] zeigen, dass Anwendungen, die über HTTP kommunizieren, ca. 50-60 % des Datenverkehrs ausmachen, der in Zugangsnetzen durch Endbenutzer anfällt. Neben Web-Seiten enthält dies auch Video- und Audio-Streaming-Daten (ca. 25-40 % der HTTP-Daten) und Datei-Downloads (ca. 30 % der HTTP-Daten). Aktuelle Untersuchungen [Cisc13, Sand13] bestätigen diese Zahlen und zeigen zudem, das Online-Video und Online-Gaming immer populärer werden, während der Anteil an Peer-to-Peer-Filesharing-Verkehr nur noch langsam zunimmt. Obwohl regionale Unterschiede vorhanden sind (Online-Video-Dienste werden z. B. in Nordamerika stärker genutzt als in Europa) sind die Trends weltweit ähnlich. Einige Anwendungen nutzen eigene Anwendungsprotokolle, die bspw. zusätzli-

che Funktionen zur Aushandlung von Inhaltsformaten anbieten (z. B. RTSP/SDP [ScRL98]) oder gar eigene, virtuelle Topologien als Overlay aufbauen (P2P-Anwendungen). Die eingesetzten Transportprotokolle beschränken sich dabei meist auf TCP oder UDP.

In [PoGS10] argumentieren die Autoren, dass HTTP inzwischen der De-Facto-Standard für neue Dienste im Internet sei, und tatsächlich wurden und werden darüber viele neue Dienste angeboten. Die Konzepte von HTTP wurden mit dem Representational State Transfer (REST) generalisiert, was seitdem die Grundlage für die Realisierung skalierbarer Web-Anwendungen bildet [FiTa00]. Mit REST wird gefordert, dass einzelne Anfragen der Clients immer alle notwendigen Informationen enthalten, sodass der Server keinen Zustand halten muss. REST kennt dabei Ressourcen (das Ziel der Anfrage), deren Lokation (als *Uniform Resource Locator, URL*), deren Repräsentation (z. B. HTML Dokument) und deren Metadaten (z. B. das letzte Änderungsdatum). Zusätzlich wird über sog. Verben angegeben, was mit der Ressource geschehen soll, also z. B. ob sie angefordert wird (GET) oder ob etwas hinzugefügt werden soll (POST). Außerhalb von HTTP wurde das Konzept der URL – welche zunächst hauptsächlich die Lokation einer Ressource beschreibt – mit der URI (engl. *Uniform Resource Identifier*) [Bern$^+$05] verallgemeinert. URIs werden dabei heute genutzt, um Ressourcen global eindeutig zu identifizieren und bestimmten Anwendungen zuzuordnen.

Im Gegensatz zu dieser naheliegenden Abstraktion von Ressourcen geht die heutige Anwendungsschnittstelle für Kommunikationsdienste – die Berkeley- bzw. BSD-Socket-Schnittstelle – von einer geringeren Abstraktionsebene aus und überlässt alles weitere den Anwendungen. Die Socket-Schnittstelle stellt als Kommunikationsabstraktion einen sog. Endpunkt bereit, auf den die Anwendung lesend und schreibend zugreifen kann, um Daten mit einem anderen Endpunkt auszutauschen. Die Anwendung muss zur Erstellung des eigenen Endpunkts folgende Parameter angeben:

- domain – die Netzwerkarchitektur bzw. Protokollfamilie.

  Darüber wird ausgewählt, über welche Netzwerkarchitektur kommuniziert werden soll, und es wird zugleich das Adressformat festgelegt. Neben IPv4 und IPv6 können bspw. auch X.25 und IPX ausgewählt werden.

- type – der Typ des Endpunkts.

  Hierüber wird festgelegt, wie die Daten ausgetauscht werden sollen, bspw. als zuverlässiger Byte-Strom oder als unzuverlässige Datagramme.

- protocol – das Protokoll.

Mit diesem Parameter legt die Anwendung das zu verwendende Transportprotokoll fest. Hiermit kann sie bspw. TCP, SCTP oder UDP auswählen.

Nach dem Erstellen des Endpunkts unterscheidet sich der weitere Ablauf abhängig vom angegebenen Typ: Von einem Byte-Strom wird angenommen, dass dieser über ein verbindungsorientiertes Protokoll ausgetauscht wird. Daher ist zunächst ein connect-Aufruf notwendig, bei dem die Anwendung die Zieladresse angeben muss. Bei einem Datagramm-Endpunkt kann jedes Datagramm an eine andere Zieladresse versandt werden. Hier ist bei jedem sendto-Aufruf die Angabe der jeweiligen Zieladresse notwendig. Das Format und die Bedeutung der Zieladresse unterscheiden sich dabei abhängig von der Netzwerkarchitektur.

Hieraus ergeben sich drei wesentliche Probleme: Der Anwendungsprogrammierer muss wissen, über welche Netzwerkarchitekturen seine Anwendung kommunizieren können soll, welches Protokoll sie dazu nutzen soll und wie das netzwerkarchitekturspezifische Adressformat aussieht. Sollen verschiedene Netzwerkarchitekturen und Protokolle unterstützt werden, muss der Anwendungsprogrammierer eine entsprechende Unterstützung für alle implementieren und bei gleichzeitig vorhandenen Alternativen selbst die Auswahl vornehmen.

Für den Anwendungsprogrammierer naheliegender ist jedoch die direkte Angabe der Ressource, die er in seiner Anwendung verwenden möchte, zusammen mit den gewünschten Eigenschaften des Kommunikationsdienstes. Während HTTP bzgl. der Ressourcenabstraktion bereits den richtigen Weg geht, ist es bei der Wahl und Austauschbarkeit der darunterliegenden Transportprotokolle stark eingeschränkt. Zudem erfordern Erweiterungen wie TLS und SPDY zusätzliche Unterstützung in der Anwendung, was die Komplexität für den Anwendungsprogrammierer deutlich steigert. Tabelle 5.1 fasst die wesentlichen Unterschiede zwischen den HTTP bzw. REST Abstraktionen und Berkeley-Sockets zusammen.

5.2 Anforderungen

Die drei wesentlichen Anforderungen an die im nächsten Abschnitt vorgestellte Anwendungsschnittstelle sind

- die ausschließliche Verwendung von URIs zur Identifikation von Ressourcen,
- die protokollunabhängige Beschreibung des gewünschten Kommunikationsdienstes mittels Anwendungsanforderungen und

	HTTP/REST	Berkeley-Sockets
Gegenstand der Kommunikation	Ressource	Endpunkt
Identifikation	URI	Adresse (z. B. IP-Adresse und Port)
Aktionen	Verben (GET, POST, ...)	—
Meta-Daten	Inhaltsbeschreibungen in Anfrage und Antwort	Adress- und Portinformationen der Gegenstelle
Transport	TCP	abhängig vom Typ des Endpunkts wählbar

Tabelle 5.1 Gegenüberstellung der Abstraktionen von HTTP/REST und Berkeley-Sockets.

- die Bereitstellung von grundlegenden Primitiven zur Beschreibung der gewünschten Aktionen auf einer Ressource.

Im letzten Abschnitt wurde gezeigt, dass URIs (mit URLs als eine Untermenge davon) eine verbreitete und geeignete Ressourcenabstraktion darstellen. Daher sollen diese auch weiterhin zur Angabe des Namens der Ressource in dieser Arbeit verwendet werden, um auf protokollspezifische Adressen verzichten zu können.

Um auch auf die konkrete Protokollauswahl durch die Anwendung verzichten zu können, muss der von der Anwendung angeforderte Kommunikationsdienst allgemeiner beschrieben werden. Dazu eignen sich protokollunabhängige Dienstbeschreibungen wie „zuverlässiger Transport“, was alle Eigenschaften eines zuverlässigen Transportdienstes zusammenfasst. Die einzelnen Eigenschaften können dabei auch feingranularer festgelegt werden, sodass bspw. nur die „Einhaltung der Reihenfolge“ gefordert werden kann, nicht aber die vollständige Auslieferung der Daten. Da diese Dienstbeschreibungen die Anforderungen der Anwendung an den Kommunikationsdienst darstellen, werden sie als *Anwendungsanforderungen* bezeichnet.

Wie im letzten Abschnitt gezeigt, nutzt ein Großteil der Anwendungen HTTP, um Anfragen an entfernte Ressourcen zu stellen. Andere Anwendungen nutzen ebenfalls Protokolle, die heute als Anwendungsprotokolle bezeichnet werden, um Anfragen und Metadaten zum Ziel zu übertragen. Um auch hier den Anwendungsprogrammierer von der Implementierung dieser Anfrageprotokolle zu entlasten und die Einführung neuer Anfrageprotokolle zu erleichtern, wird gefordert, dass diese durch eine geeignete Schnittstelle abstrahiert werden können.

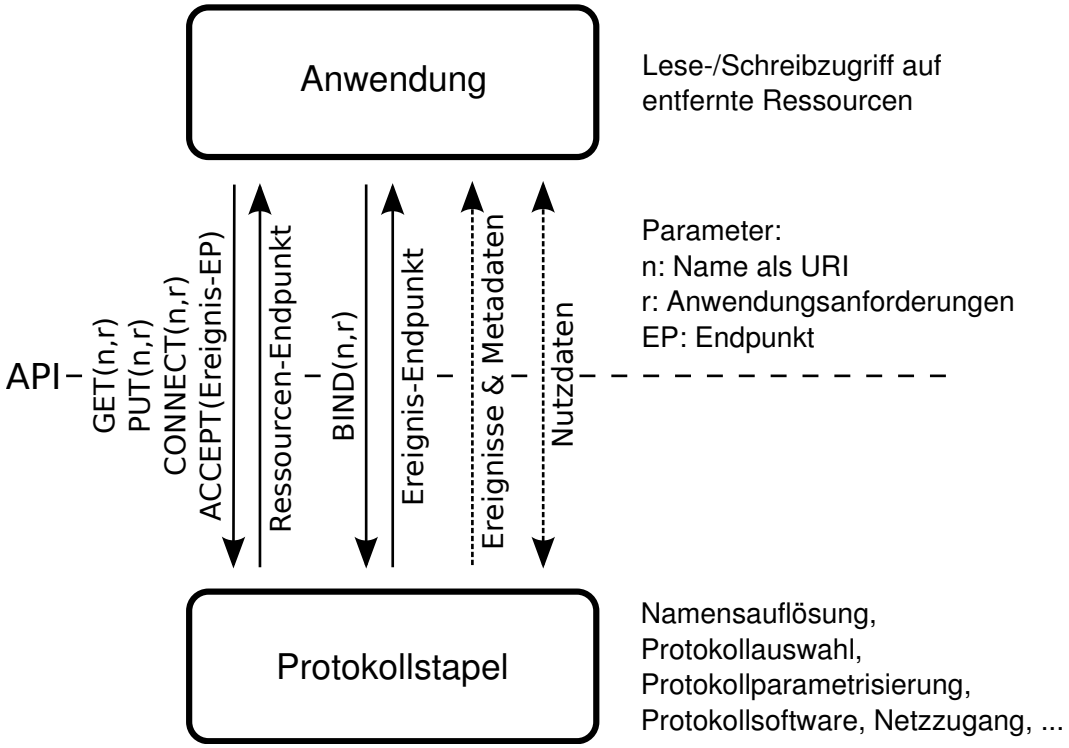


Abbildung 5.1 Überblick über die protokollagnostische Anwendungsschnittstelle.

5.3 Entwurf

Basierend auf den Beobachtungen in Abschnitt 5.1 und den daraus abgeleiteten Anforderungen an die Anwendungsschnittstelle in Abschnitt 5.2 wurde im Rahmen dieser Arbeit eine *protokollagnostische* Anwendungsschnittstelle [MaWB11] entworfen, bei der Anwendungen ihren Kommunikationswunsch ohne Angabe eines konkreten Protokolls äußern können. Neben URIs zur Identifikation des Ziels der Kommunikation (Abschnitt 5.3.2) werden mehrere Primitiven eingeführt (Abschnitt 5.3.3), die die Intention der Kommunikation angeben, also bspw. ob Daten angefordert werden sollen, versendet werden sollen oder ob eine interaktive Kommunikation stattfinden soll. Ereignisse und Metadaten zur Kommunikationsbeziehung (Abschnitt 5.3.4) werden dabei ebenso berücksichtigt wie Anwendungsanforderungen, die der Anwendung dazu dienen, den gewünschten Kommunikationsdienst protokollunabhängig zu beschreiben. Diese Abstraktionen ermöglichen zudem erweiterte Einsatzszenarien, die in Abschnitt 5.3.5 vorgestellt werden.

5.3.1 Übersicht

Die wesentlichen Abstraktionen, die von der Anwendungsschnittstelle verwendet werden, sind in Abbildung 5.1 dargestellt und bestehen aus:

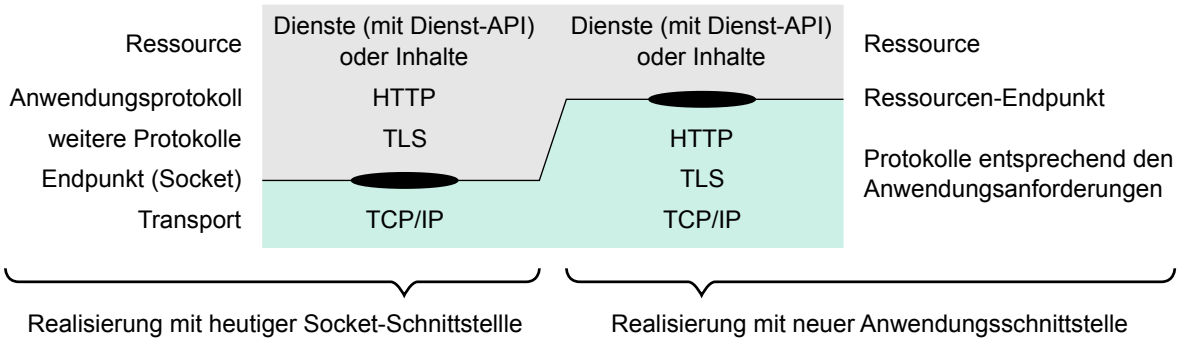


Abbildung 5.2 Kommunikationsendpunkt der Socket-Schnittstelle (links) und ein Ressourcen-Endpunkt der neuen Anwendungsschnittstelle (rechts).

- Primitiven (GET, PUT, CONNECT, ACCEPT, BIND)
- Namen (URIs) und Anwendungsanforderungen als Parameter
- Ressourcen-Endpunkten und Ereignis-Endpunkten
- Ereignissen, Metadaten und Nutzdaten

Mit Hilfe der Primitiven GET, PUT, CONNECT und ACCEPT, die von der Anwendung aus aufgerufen werden, werden Endpunkte erzeugt. Im Unterschied zur heutigen Socket-Schnittstelle handelt es sich hierbei jedoch nicht um Kommunikationsendpunkte im traditionellen Sinne, sondern um *Ressourcen*-Endpunkte, da hierüber die Ressource direkt gelesen oder beschrieben werden kann. Ressourcen umfassen dabei Inhalte, Dienste oder gar Geräte (vgl. Abschnitt 5.3.5). Handelt es sich bei der Ressource also bspw. um eine Datei, muss die Anwendung kein weiteres Anwendungsprotokoll realisieren, um darauf zuzugreifen. Handelt es sich bei der Ressource allerdings um einen Dienst, muss die Anwendung nach wie vor die dienstspezifische Schnittstelle (Dienst-API) implementieren, wie es bei heutigen Web-Diensten üblich ist. Abbildung 5.2 veranschaulicht diese Verschiebung der Endpunkt-Abstraktion mit beispielhaften Protokollen. Dabei wird ersichtlich, dass heutige Anwendungsprotokolle nicht mehr durch die Anwendung realisiert werden müssen.

Neben den Primitiven zur Erstellung eines Ressourcen-Endpunktes existiert die Primitive BIND zur Erstellung eines *Ereignis*-Endpunktes (vgl. Abb. 5.1). Dieser weist Ähnlichkeiten zu einem Server-Endpunkt der Socket-API bei Verwendung verbindungsorientierter Transportdienste auf. Im Gegensatz zu einem Ressourcen-Endpunkt werden darüber lediglich Ereignisse wie ankommende Ressourcen-Anfragen der Anwendung mitgeteilt und keine Nutzdaten ausgetauscht. Über Ressourcen-Endpunkte können neben Nutzdaten auch Ereignisse und Metadaten ausgetauscht werden.

Durch die Ressourcen-Sicht müssen einige kommunikationsspezifischen Aufgaben nicht mehr von der Anwendung gelöst werden, sondern können vom Protokollstapel „unterhalb“ der API übernommen werden. Dazu zählen bspw. die Namensauflösung und die konkrete Protokollauswahl. Der Anwendungsprogrammierer muss sich so nicht mehr auf bestimmte Protokolle festlegen und kann die Kommunikationsaufgaben seiner Anwendung protokollagnostisch implementieren.

5.3.2 Namen und Anwendungsanforderungen

Zur Identifikation des Kommunikationsziels bzw. der Ressource werden URIs als Parameter den Primitiven übergeben. Die URI ist eine beliebig lange Zeichenkette mit der minimalen Form

```
namensschema://name
```

Das Namensschema gibt dabei die Semantik und die Form des Namens vor. Der Name kann abhängig von der Semantik Inhaltsobjekte, Dienste oder andere Ressourcen beschreiben. Ein allgemeines, typisches Beispiel für einen Namen in Form einer URI ist

```
file:///home/alice/cv.pdf
```

welche eine lokale Datei bezeichnet. Durch das Namensschema `file` wird dabei festgelegt, dass es sich beim nachfolgenden Namen um einen Dateinamen inklusive des vollständigen Pfades handelt. Weitere Beispiele einer URI sind

```
mailto:bob@example.com
tel:+49-030-1234567
```

welche eine E-Mail-Adresse [DuMZ10] bzw. eine Telefonnummer [Schu04] identifizieren (abhängig vom Namensschema sind die beiden Schrägstriche also nicht immer vorhanden). Weiterhin kann mit

```
http://example.com/time?timezone=+0200
```

ein Dienst angesprochen werden, der abhängig von gegebenen URI-Parametern entsprechende Ergebnisse zurückliefert (im verwendeten Beispiel die aktuelle Uhrzeit angepasst auf die als Parameter übergebene Zeitzone). Zum Austausch der Daten wird das Anwendungsprotokoll HTTP genutzt. Diese Beispiele zeigen, dass die Namensschemata nicht nur das Format, sondern auch die Semantik der Ressource festlegen. Zusätzlich wird in einigen Fällen von der netzspezifischen Adressierung abstrahiert: Ob z. B.

`example.com` über IPv4 oder IPv6 verwendet werden soll, wird nicht festgelegt. Ähnliches gilt für die Netzwahl bei Verwendung der Telefonnummer: Diese kann über einen Festnetzanschluss oder auch über eine Internetbasierte Voice-Anwendungen gewählt werden.

In einigen Fällen kann durch die Angabe der angeforderten Ressource als URI bereits aufgrund des Namensschema abgeleitet werden, welche Eigenschaften der Kommunikationsdienst haben muss, um die Ressource an die anfordernde Anwendung zu übertragen. Wird bspw. eine Datei von einem Dateiserver angefordert, muss der Inhalt in der Regel vollständig und in der richtigen Reihenfolge beim Empfänger ankommen. Dies ermöglicht es dem Protokollstapel passende Übertragungsprotokolle auszuwählen, ohne dass eine zusätzliche Angabe von Anwendungsanforderungen notwendig ist.

In anderen Fällen hingegen kann die Anwendung – abhängig vom Typ der übertragenen Daten – bspw. mit unvollständigen Daten umgehen, hat aber gewisse Anforderungen an die maximale Latenz. Bei einer interaktiven Sprachverbindungen können z. B. einige Audio-Samples verloren gehen, aber es wird eine Schranke für die maximale Verzögerung benötigt, um den Dienst sinnvoll nutzen zu können. Daher erlaubt es die Anwendungsschnittstelle, dass die Anwendung bei der Erstellung eines Endpunktes neben dem Namen der Ressource auch eine Menge von Anwendungsanforderungen übergeben kann. Dies geschieht durch einen entsprechenden Parameter der jeweils verwendeten Primitive.

Als weitere Quelle von Anforderungen an den Kommunikationsdienst kann der Benutzer oder der Systemadministrator identifiziert werden: Durch sie können systemweite Richtlinien festgelegt werden, welche z. B. die Sicherheitsanforderungen betreffen.

Neben den Anforderungen bzgl. der Eigenschaften des Kommunikationsdienstes können auch Anforderungen bzgl. des Inhaltes gestellt werden. Z. B. ist es möglich – ähnlich zu HTTP – eine Liste mit unterstützten Inhaltsformaten mit der Kommunikationsanfrage mitzuschicken. Die Server-Anwendung wertet diese dann über die Metadaten aus (vgl. Abschnitt 5.3.4).

Zusammenfassend bilden sich also die Anforderungen an den Kommunikationsdienst und den Inhalt einer Ressourcen-Anfrage über ein Kommunikationssystem aus folgenden Quellen:

- Aus dem Namensschema der angeforderten Ressource. Diese werden als *Standard-Anforderungen* bezeichnet.

- Aus den *Anwendungsanforderungen* der beiden Kommunikationspartner. Diese enthalten sowohl Anforderung bzgl. des Inhalts als auch bzgl. des Kommunikationsdienstes.

- Aus den Richtlinien, die durch den Benutzer oder Administrator vorgegeben werden.

Wie diese Eigenschaften von Kommunikationsdiensten aus Sicht der Anwendung beschrieben werden können, wird in verschiedenen Arbeiten diskutiert [Back10, Reut10, Völk12, WeJG11, MCWR+13] (vergleiche auch Abschnitt 3.1). In dieser Arbeit werden daran angelehnt lediglich einfache Beispielanforderungen verwendet, um die Eigenschaften des gewünschten Kommunikationsdienstes und des angeforderten Inhalts zu spezifizieren. Diese werden als Menge aus Schlüssel-Wert-Paaren in folgender Form dargestellt:

$$\{A_1 : W_1,\ A_2 : W_2,\ \ldots,\ A_n : W_n\}$$

wobei A_i die Anwendungsanforderung bezeichnet und W_i den dazugehörigen Wert ($i \in \{1, 2, \ldots, n\}$). Mit der Menge folgender Anwendungsanforderungen

$$\{\text{reliable} : 1,\ \text{content-type} : \text{“image/jpeg”}\}$$

wird bspw. ein zuverlässiger Transport angefordert und als Inhaltstyp ein JPEG-Bild erwartet.

5.3.3 Primitiven

Die Nutzung der Anwendungsschnittstelle unterscheidet sich je nach dem ob auf Ressourcen direkt zugegriffen wird (d. h., ein Ressourcen-Endpunkt erstellt wird) oder ob lediglich Ereignisse zu einer Ressource empfangen werden sollen. Diese Unterscheidung lässt sich am Beispiel des Client/Server-Paradigmas wie folgt erläutern: Clients greifen direkt auf eine Ressource zu und fordern diese an, modifizieren sie oder interagieren mit ihr. Server hingegen bedienen die von Clients ankommenden Ressourcen-Anfragen. Sie erstellen dazu zunächst einen Ereignis-Endpunkt, über den dann für jede Client-Anfrage ein Ressourcen-Endpunkt auf Server-Seite erstellt wird. Dieser Ablauf folgt dem gleichen Prinzip wie ein klassischer verbindungsorientierter Socket weshalb die notwendigen Primitiven die gleichen Bezeichnungen verwenden wie die Socket-Schnittstelle (CONNECT, BIND, ACCEPT).

Wie in Abschnitt 5.4.1 anhand weiterer Paradigmen gezeigt wird, muss es sich dabei nicht immer um Clients und Server handeln. Zur vereinfachten Erläuterung wird als Beispiel in den folgenden beiden Abschnitten jedoch das Client/Server-Paradigma herangezogen, bevor auf alternative Paradigmen eingegangen wird.

5.3.3.1 Ressourcen-Endpunkt

Für den Nutzer einer Ressource (z. B. eine Client-Anwendung) werden drei Primitiven bereitgestellt, die jeweils einen Ressourcen-Endpunkt erzeugen: CONNECT, GET und PUT. Während CONNECT am ehesten dem heutigen

socket()-Aufruf mit anschließendem connect()-Aufruf entspricht, werden GET und PUT als grundlegende Primitiven ergänzt, um hier eine explizite Unterstützung für die vielen Anwendungen zu bieten, die primär an Inhalte interessiert sind und bspw. heute auf HTTP aufsetzen. Dies ist notwendig, um eine effektive Entkopplung dieser Anwendungen von HTTP und den darunterliegenden Transportprotokollen zu erreichen und um damit schließlich alternative Protokolle zu ermöglichen. Der Aufruf dieser Primitiven gestaltet sich wie folgt:

- Ressourcen-Endpunkt = CONNECT(Ziel-URI, Menge an Anforderungen)
- Ressourcen-Endpunkt = GET(Ziel-URI, Menge an Anforderungen)
- Ressourcen-Endpunkt = PUT(Ziel-URI, Menge an Anforderungen)

Wie bei anderen Ein-/Ausgabe Funktionen eines Betriebssystems können anschließend auf diesen Ressourcen-Endpunkten Lese- und Schreiboperationen durchgeführt werden, und der Ressourcen-Endpunkt kann am Ende der Kommunikation mit einer entsprechenden close-Funktion wieder geschlossen werden. Der konkrete Ablauf einer GET-Anfrage sieht dabei wie folgt aus:

```
ressourcenEndpunkt = GET("app://ressourcenname", {});
daten = read(ressourcenEndpunkt);
close(ressourcenEndpunkt);
```

In diesem Fall wurden keine weiteren Anwendungsanforderungen an den Kommunikationsdienst gestellt, weshalb der entsprechende Parameter eine leere Menge darstellt ({}). Dies bedeutet, dass die Standard-Anforderungen genutzt werden, die dem Namensschema zugeordnet sind (vgl. Abschnitt 5.2).

Während es sich bei den Ressourcen, die mittels GET und PUT angefordert werden, um Inhalte handelt, über die lediglich der Typ und das Format bekannt sein muss, kann mit CONNECT ein Anwendungsdienst angesprochen werden. Die ausgetauschten Informationen müssen in diesem Fall der jeweiligen Dienst-API entsprechen.

5.3.3.2 Ereignis-Endpunkt

Der Anbieter einer Ressource (z. B. eine Server-Anwendung) bindet sich auf eine URI, unter der er Inhalte oder Dienste für andere Teilnehmer im Netz bereitstellt. Dies geschieht durch die Primitive BIND. Mit der Primitive ACCEPT werden daraufhin eingehende Anfragen akzeptiert. Der grundlegende Ablauf lehnt sich dabei an die heutige Socket-Schnittstelle an, ohne jedoch konkrete Protokolle und protokollspezifische Adressen verwenden zu müssen. Der Aufruf der Primitiven geschieht wie folgt:

- Ereignis-Endpunkt = BIND(URI, Menge an Anforderungen)
- Ressourcen-Endpunkt = ACCEPT(Ereignis-Endpunkt)

Der hier erzeugte Ereignis-Endpunkt dient dabei dem Austausch von Ereignissen wie dem Eingang einer Anfrage. Durch eine entsprechende wait-Funktion kann die Anwendung auf diese eingehenden Anfragen warten. Der Ablauf auf Anbieterseite lässt sich im Pseudocode wie folgt zusammenfassen:

```
ereignisEndpunkt = BIND("app://ressourcenname", {});
wait(ereignisEndpunkt);
ressourcenEndpunkt = ACCEPT(ereignisEndpunkt);
write(ressourcenEndpunkt, daten);
close(ressourcenEndpunkt);
close(ereignisEndpunkt);
```

Eine Besonderheit des Ereignis-Endpunktes ist es, dass sich Anwendungen nicht nur auf eindeutige Ressourcen-Namen binden können, sondern auch auf sog. *Basis*-URIs. Eine Basis-URI ist dabei ein gemeinsamer Präfix mehrerer Ressourcen-Namen. Anfragen, die an eine Ressource mit diesem Präfix gerichtet sind, werden dabei ebenfalls als Ereignis an den Ereignis-Endpunkt weitergereicht. Bindet sich die Anwendung also bspw. auf

```
app://dateiserver/dateien/
```

erhält sie auch Anfragen, die an

```
app://dateiserver/dateien/cv.pdf
```

gerichtet sind. Somit ist es auf einfache Art und Weise möglich einen Datei- oder Web-Server zu realisieren, ohne dass sich die Server-Anwendung auf bestimmte Anwendungsprotokolle wie HTTP oder FTP festlegen muss. Diese protokollunabhängige Namensbindung bietet zwei wesentliche Vorteile:

- Die Server-Anwendung legt sich nicht auf ein bestimmtes Protokoll fest. Das für die Kommunikation verwendete Protokoll kann also je nach Anfrage variieren.
- Die Entscheidung über die Art der Zustellung (z. B. zuverlässig oder unzuverlässig) kann den Clients überlassen werden. Diese können wiederum basierend auf dem jeweiligen Verwendungszweck individuell entscheiden, welche Anforderungen sie an den Kommunikationsdienst stellen.

Diese Vorteile für Anwendungen bzw. für Anwendungsprogrammierer führen allerdings auch zu einem Mehraufwand für die darunterliegenden Protokollstapel. Ein Rahmenwerk für Protokollstapel, welches hier Unterstützung bietet, wird mit dem NENA-Rahmenwerk in Kapitel 6 vorgestellt.

5.3.4 Ereignisse und Metadaten

Über Ereignis-Endpunkte können Anwendungen Ereignisse wie ankommende Anfragen zu einer Ressource erhalten. Über Ressourcen-Endpunkte kann eine Anwendung ebenfalls Ereignisse empfangen, bspw. bzgl. Abbrüchen und Fehlern während der Kommunikationsbeziehung. Zusätzlich können Metadaten zu einer Ressource abgerufen werden, die Art und Formatierung der Daten oder auch Regeln zum Zwischenspeichern der Inhalte (Caching) beschreiben. Während die anfragende Anwendung diese Metadaten als Anwendungsanforderungen beim Aufruf der Primitive übergibt, werden von der gegenüberliegenden Anwendungen die Metadaten über eine weitere Funktion abgerufen:

- Metadaten = GET_METADATA(Ressourcen-Endpunkt)

Ebenfalls zu den Metadaten gehören die verwendete Primitive und die vollständige URI. Dies ist insbesondere deswegen wichtig, weil durch ein BIND zunächst nicht eingeschränkt wird, mit welchen Primitiven eine anfragende Anwendung auf die Ressource zugreifen kann, der Server aber abhängig von der Primitive andere Aktionen ausführt. Der vollständige Name der Ressource muss insbesondere bei der Bindung auf eine Basis-URI für die bereitstellende Anwendung abrufbar sein (vgl. Abschnitt 5.3.3.2). Da die anfragende Anwendung über die Anfrage auch weitere Anforderungen an den Inhalt der Ressource stellen kann, muss die bereitstellende Anwendung diese ebenfalls abrufen können. Liegt eine Ressource bspw. in verschiedenen Formaten vor, kann so die anfragende Anwendung die von ihr unterstützten Formate bei der Anfrage mit übermitteln.

5.3.5 Erweiterte Einsatzszenarien

Da URIs nicht nur verteilte Ressourcen bezeichnen, die über ein Kommunikationsnetz zu erreichen sind, sondern auch lokale Ressourcen, kann die bisher vorgestellte Anwendungsschnittstelle mit weiteren Primitiven ergänzt werden, die Anwendungen noch mehr Vereinfachungen bieten: Statt die empfangenen Daten der Ressource direkt entgegenzunehmen, kann die Anwendung ein lokales Ziel angeben, an das die angeforderte Ressource „gesendet“ werden soll. Am besten lässt sich dies am Beispiel eines Datei-Downloads verdeutlichen: Durch

```
GET(„web://example.com/program.pdf“,
                „file:///home/alice/Downloads/program.pdf“, {})
```

wird der Protokollstapel instruiert, die von der Ressource empfangenen Daten direkt in den lokalen Ordner „Downloads“ zu schreiben. Der Download wird also an den Protokollstapel *delegiert*.

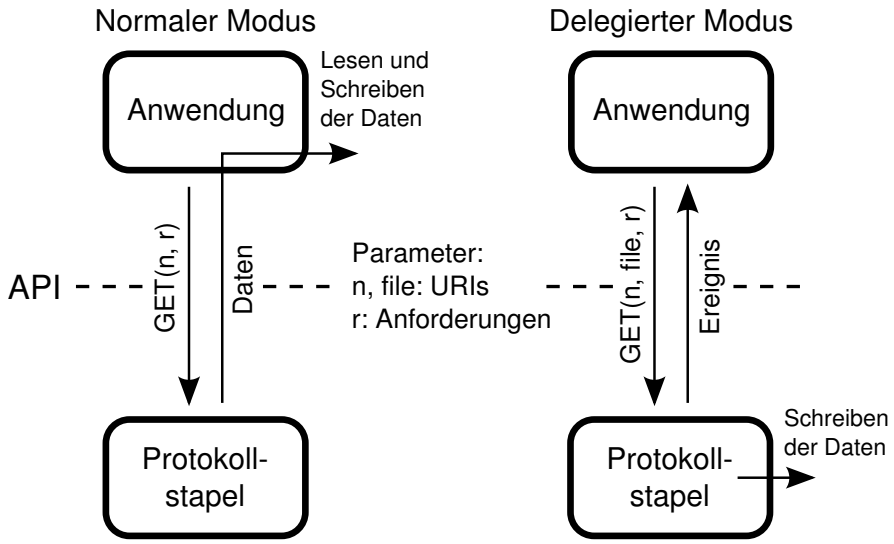


Abbildung 5.3 Normaler Modus und delegierter Modus am Beispiel eines Datei-Downloads.

Im Vergleich zum normalen Modus (Abb. 5.3 links) ist die Anwendung im delegierten Modus (Abb. 5.3 rechts) nicht weiter involviert und muss die Daten nicht selbst abspeichern. Über die in Abschnitt 5.3.4 erwähnten Ereignisse kann die Anwendung jedoch über den Fortschritt und über die Beendigung der delegierten Übertragung informiert werden. Ähnliche Vorteile bieten sich beim Hochladen der Daten mittels PUT. Die delegierenden Primitiven sehen wie folgt aus:

- Ereignis-Endpunkt = GET(entfernte URI, lokale URI, Anforderungen)
- Ereignis-Endpunkt = PUT(lokale URI, entfernte URI, Anforderungen)
- Ereignis-Endpunkt = CONNECT(Quell-URI, Ziel-URI, Anforderungen)

CONNECT ist in diesem Fall für direktes Streaming geeignet: Werden Daten über ein an den Rechner angeschlossenes Gerät erzeugt (z. B. durch ein Tonaufnahmegerät) und über einen mit einer URI adressierbaren, lokalen Dienst bereitgestellt (z. B. über einen Audio-Dienst), kann der dort erzeugte Datenstrom vom Betriebssystem direkt an den Protokollstapel weitergereicht und versandt werden. Analog dazu kann ein empfangener Audio-Strom direkt an den Audio-Dienst des Betriebssystems weitergegeben werden. Der Audio-Dienst muss in beiden Fällen die angeforderten Formate unterstützen und selbst für Pufferung und (De-)Kodierung sorgen. Gibt es von Seiten des Betriebssystems weitere lokale Dienste die mittels einer URI angesprochen werden können, können auch weitere Aufgaben von der Anwendung an das Betriebssystem delegiert werden: Bspw. ist so das direkte Abspielen eines Vi-

deos auf einem Video-Canvas[1] denkbar, ohne dass die Anwendung selbst die Dekodierung und die graphische Darstellung der Video-Daten übernehmen muss.

Beschränkt man sich dabei nicht nur auf lokale Ressourcen, sind weitere vielseitige Einsatzszenarien denkbar. So kann eine Anwendung z. B. einen Video-Strom aus dem Internet direkt an einen Fernseher in der Nähe delegieren. Proprietäre Lösungen verschiedener Hersteller (z. B. Sony und Samsung Smart-TVs in Verbindung mit den jeweiligen Mobilgeräten) zeigen, dass ein solches Szenario für den praktischen Einsatz sinnvoll ist, auch wenn hier meist das Tablet oder das Smartphone als Zwischengeräte dienen (welche bei der delegierten Übertragung entlastet werden können). Durch die Integration dieser Funktionalität in diese Schnittstelle kann dieses Konzept verallgemeinert und optimiert werden.

5.4 Verwendung

Dieser Abschnitt beschreibt die Verwendung der Anwendungsschnittstelle für verschiedene Einsatzszenarien. Dazu werden zunächst verschiedene Kommunikationsparadigmen diskutiert und anschließend ein Beispiel aus einem konkreten Anwendungsszenario beschrieben.

5.4.1 Kommunikationsparadigmen

Neben der klassischen Ende-zu-Ende Kommunikation unterstützt die protokollagnostische Anwendungsschnittstelle auch weitere Kommunikationsparadigmen, deren Nutzung hier vorgestellt wird. Eine Bewertung dieser Nutzung wird in Kapitel 7 durchgeführt.

5.4.1.1 Ende-zu-Ende Kommunikation

Für eine traditionelle Ende-zu-Ende-Kommunikation über TCP, wie man sie heute über die Socket-Schnittstelle nutzen würde, sieht die Verwendung der in dieser Arbeit entwickelten Anwendungsschnittstelle wie folgt aus. Die Server-Anwendung öffnet zunächst einen Ereignis-Endpunkt:

Ereignis-Endpunkt = BIND(„tcp://192.168.0.1:4200“, {})

Durch das Namensschema wird angegeben, dass TCP als darunterliegendes Transportprotokoll verwendet werden soll. Der Name kann dann direkt auf die notwendigen Parameter zur Erstellung eines TCP-Sockets abgebildet werden, und es sind keine weiteren Anwendungsanforderungen mehr notwendig. Client-seitig wird ein CONNECT-Aufruf auf dieselbe URI durchgeführt:

[1] Mit einem Video-Canvas ist hier ein Zeichenbereich in der graphischen Benutzeroberfläche (GUI) der aufrufenden Anwendung gemeint.

Ressourcen-Endpunkt = CONNECT(„tcp://192.168.0.1:4200“, {})

Ersetzt man in diesem Beispiel die IP-Adresse durch einen Host-Namen, kann der Protokollstapel die notwendige Namensauflösung durchführen und ggfs. selbst entscheiden, ob die Kommunikation über IPv4 oder IPv6 erfolgen soll.

Mit dieser Art der Nutzung wird eine direkte Kompatibilität zu heutigen Netzen hergestellt, und der Anwendungsentwickler wird durch die Verwendung der Anwendungsschnittstelle bereits entlastet. Zudem müssen nicht beide an der Kommunikation teilnehmenden Anwendungen gleichzeitig auf die neue Anwendungsschnittstelle umgestellt werden, da auf der Gegenseite noch die traditionelle Socket-Schnittstelle verwendet werden kann. Dies ermöglicht die schrittweise Einführung der neuen Anwendungsschnittstelle.

Zur besseren Entkopplung von Anwendungen und Protokollen ist jedoch die Verwendung eines anwendungsspezifischen Namensschemas oder eines generischen Namensschemas für Anwendungsdienste eleganter:

Ereignis-Endpunkt = BIND(„app://RechnerA/ChatAnwendung“, {})

An dieser Stelle wird noch kein Transportprotokoll festgelegt. Auch die Anforderungen an den Kommunikationsdienst sind zunächst noch offen, was bedeutet, dass die Anwendung auch Anfragen mit unzuverlässig zugestellten Nachrichten zulässt. Nicht möglich sind allerdings Anfragen, die weitere Informationen von der anbietenden Anwendung benötigen, wie bspw. Zertifikate zur Authentifizierung.

Eine Client-Anwendung auf Rechner B, die sich mit dem beispielhaften Chat-Anwendungsdienst verbinden möchte, öffnet mit folgendem Befehl einen entsprechenden Ressourcen-Endpunkt:

Ressourcen-Endpunkt = CONNECT(„app://RechnerA/ChatAnwendung“,
{reliable: 1})

Hier legt der Client außerdem fest, dass die Verbindung zuverlässig erfolgen soll (reliable: 1). Über den erstellten Ressourcen-Endpunkt müssen nun Funktionen der Schnittstelle des Chat-Anwendungsdienstes (Dienst-API) aufgerufen werden. Diese Dienst-API legt das Format und die Typen der ausgetauschten Nachrichten für beide Kommunikationspartner fest.

Durch die Verwendung von PUT und GET und durch geeignete URIs kann diese Dienst-API auf ein Minimum beschränkt werden. Mit folgendem Aufruf kann bspw. eine zu sendende Chat-Nachricht direkt an den durch die URI adressierten Empfänger gesendet werden:

Ressourcen-Endpunkt = PUT(„app://RechnerA/ChatAnwendung/Alice“,
{reliable: 1, encoding: utf8})

Die Server-Anwendung erkennt durch die erweiterte URI, an welchen Benutzer die Nachricht adressiert ist und kann anhand der Metadaten erkennen, mit welcher Zeichenkodierung die Nachricht zu interpretieren ist. Auf ähnliche Weise kann auch der Abruf der Chat-Nachrichten erfolgen:

Ressourcen-Endpunkt =
GET(„app://RechnerA/ChatAnwendung/Bob/NeuesteNachricht“,
{reliable: 1, session-key: <Schlüssel>})

Über die Metadaten kann die Anwendung zusätzliche Informationen über die Nachricht abrufen, z. B. den Zeitstempel der Erstellung und den Absender der Nachricht. Sicherheitsrelevante Funktionen bspw. zur Authentifizierung der jeweiligen Benutzer müssen separat aufgerufen werden. Dabei entstehende Sitzungsschlüssel (*session keys*) können ebenfalls als Metadaten mit übertragen werden. Eine solche Authentifizierung und Erzeugung eines Sitzungsschlüssels kann dabei wie folgt aussehen:

Ressourcen-Endpunkt =
GET(„app://RechnerA/ChatAnwendung/Bob/Auth“,
{reliable: 1, passphrase: <Passwort>})

Anschließend kann der Sitzungsschlüssel aus dem Ressourcen-Endpunkt ausgelesen und bei späteren Anfragen wieder verwendet werden. Zu beachten ist hier, dass das darunterliegende System entsprechende Zustände halten muss, um dies zu ermöglichen. Eine Verschlüsselung der zu übertragenden Nachrichten kann ebenfalls von der Anwendung angefordert werden, welche dann die Daten nach wie vor im Klartext an die Anwendungsschnittstelle übergibt. Die Ver- und Entschlüsselung wird dann vom darunterliegenden System ausgeführt.

Während die Variante mit CONNECT der traditionellen Ende-zu-Ende-Verbindung zwischen Anwendungen entspricht, ist die Lösung mit GET und PUT bereits eine inhaltsbasierte Abstraktion, auf welche im nächsten Abschnitt näher eingegangen wird.

5.4.1.2 Inhaltsbasierte Kommunikation

Bei der inhaltsbasierten Kommunikation (Content-Centric Networking, CCN) wird nicht der Ort adressiert, an dem die Daten liegen, sondern die Daten selbst werden adressiert. Während diese Sichtweise auf Anwendungsseite bereits heute durch URIs verbreitet ist, bietet sie auch Vorteile für die Datenverteilung im Netz, worauf in Abschnitt 7.3.4 weiter eingegangen wird. Der Vorteil einer inhaltsbasierten Kommunikationsabstraktion für den Anwendungsprogrammierer ist, dass er sich ausschließlich auf die eigentlichen

Inhalte konzentrieren kann, ohne sich mit Kommunikationsdetails auseinandersetzen zu müssen (bspw. zur Namensauflösung).

Mit der inhaltsbasierten Kommunikation gibt es keine Server-Seite aus Anwendungssicht mehr. Daten werden hier vom Inhaltsanbieter unter einer URI veröffentlicht (PUT) und von der anfragenden Anwendungen über diese URI wieder abgerufen (GET). Die anfragende Anwendung verhält sich also beim Abruf der Daten mittels GET genauso wie bei der traditionellen Ende-zu-Ende-Kommunikation wie sie im vorangegangenen Abschnitt beschrieben wurde. Der Inhaltsanbieter hingegen veröffentlicht seine Inhalte einmalig woraufhin sie in Zwischenspeichern im Kommunikationssystem abgelegt werden. Anfragen werden daraufhin vom Kommunikationssystem beantwortet, ohne dass die bereitstellende Anwendung weiter involviert ist. Dadurch können die Daten vom Kommunikationssystem je nach Nutzung dynamisch dupliziert und näher bei den Benutzern zwischengespeichert werden. Während sogenannte Content Distribution Networks (CDN) ähnliche Zielsetzungen aufweisen, ist CCN ein verallgemeinerter Ansatz, der für alle Benutzer gleichermaßen funktioniert.

Die Primitiven CONNECT und BIND finden bei der inhaltsbasierten Kommunikation zunächst keine Verwendung.

5.4.1.3 Publish / Subscribe

Für inhaltsbasierte Kommunikation, aber auch für nachrichtenbasierte Kommunikation wird häufig das Publish/Subscribe-Paradigma genutzt: Anwendungen, die neue Informationen veröffentlichen möchten, rufen hierzu ein *publish* mit einem Namen als Parameter auf, unter dem die Informationen veröffentlicht werden sollen. Anwendungen, die Informationen empfangen wollen, die unter einem bestimmten Namen veröffentlicht werden, rufen hingegen *subscribe* mit diesem Namen als Parameter auf. Dieses *subscribe* ist zugleich ein Abonnement für alle zukünftigen Inhalte, die unter diesem Namen veröffentlicht werden. Aus diesem Grund wird der Name oft auch als *Thema* (engl. *topic*) bezeichnet.

Die Veröffentlichung kann wie im vorherigen Abschnitt mittels PUT realisiert werden. Die Funktionsweise von *subscribe* geht allerdings über ein reines GET hinaus: Während mit GET existierende Inhalte einmalig abgerufen werden, handelt es sich bei *subscribe* um eine Abonnement des Themas. Daher ist in diesem Fall die Nutzung von BIND vorzuziehen:

Ereignis-Endpunkt = BIND(„topic://NachrichtenAnbieter/Ticker“, {})

Stehen neue Veröffentlichungen zu einem Thema an, wird dies über den durch den BIND-Aufruf erzeugten Endpunkt an die Anwendung signalisiert, welche wieder ein ACCEPT aufrufen muss, bevor die neue Veröffentlichung empfangen werden kann.

5.4.2 Dienstankündigung

Das Binden eines Anwendungsdienstes an eine URI ist möglich, ohne dass direkt ein bestimmtes Protokoll festgelegt werden muss (vgl. Abschnitt 5.3.3.2). Einige Protokolle benötigen jedoch lokalen Zustand, damit ankommende Anfragen beantwortet werden können. Für TCP und UDP bedeutet dies bspw., dass für beide Protokolle lokale Ports geöffnet werden müssen. Während für einige Anwendungsdienste Portnummern registriert sind [CETW+11], sind spätestens für Anwendungen, die nicht direkt für TCP oder UDP konzipiert sind, keine festen Portnummern vorhanden. Hier ist ein Namensdienst notwendig, um den als URI übergebenen Dienstnamen im Netz bekanntzugeben.

Für IP-basierte Netze existieren dazu bereits Vorschläge, wie DNS für die Identifikation von Diensten verwendet werden kann: Mittels DNS Service Records [GuVE00] können hier Adressen und Protokolle hinterlegt werden, über die ein Dienst erreichbar ist. Für lokale Netze kann dazu Bonjour bzw. Multicast-DNS eingesetzt werden [ChKr13]. Diese Lösungen beziehen sich dabei lediglich auf TCP/IP-Netze. Für andere Netzwerkarchitekturen sind entsprechende Lösungen notwendig. Ihnen allen gemein ist jedoch, dass sie transparent zu den Anwendungen realisiert werden können, was notwendig für die Entkopplung von Anwendungen und Protokollen ist.

5.4.3 Beispiel-Szenario: Ein Video-On-Demand-Dienst

Als Machbarkeitsnachweis für die hier vorgestellte Anwendungsschnittstelle wurde diese implementiert (siehe Abschnitt 5.5) und im Rahmen eines Experiments mit mehreren Anwendungen und Netzen genutzt [BaMW12]. Das im Experiment verwendete Szenario beschreibt einen Online-Video-Dienst, dessen Daten über drei verschiedene Netze bereitgestellt werden (Abbildung 5.4):

- ein Netz zur Präsentation des Angebots als Web-Seite (nenaweb://),
- ein Netz zum Download der Video-Dateien (nenacdn://) und
- ein Netz zum Streaming der Video-Daten (nenavideo://).

Diese Netze sind in diesem Szenario auf den jeweiligen Anwendungsfall hin optimiert. Das Experiment verwendet dabei zur Kommunikation das in Kapitel 6 vorgestellte NENA-Rahmenwerk mit vereinfachten Protokollen zum Transport. Während die Web-Seite des Angebots auf einem Server liegt, werden die Video-Daten von einem anderen Server bereitgestellt. Je nach Anwendungsfall – Download oder Streaming – werden die Daten jedoch über unterschiedliche Netze und Protokolle transportiert: Im Streaming-Fall

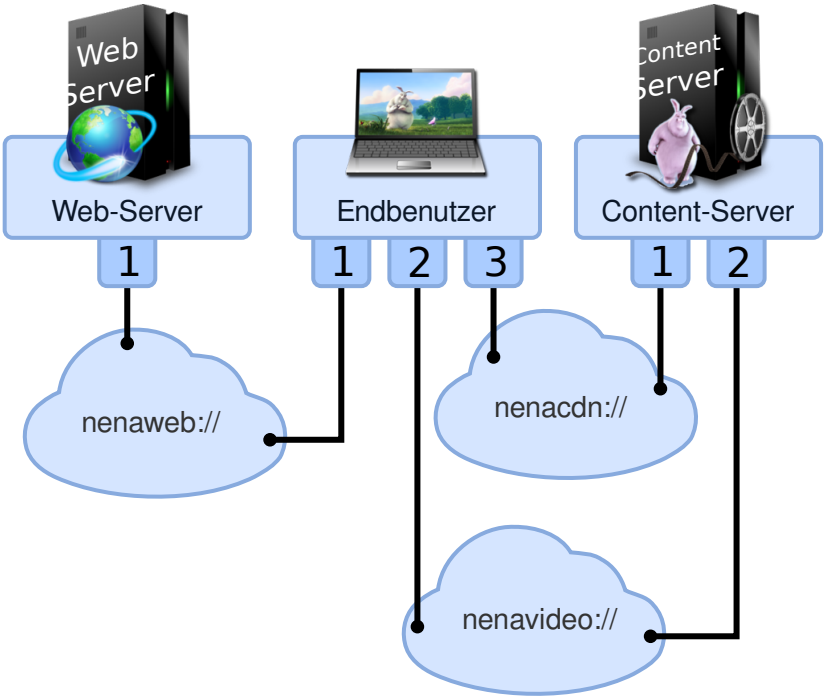


Abbildung 5.4 Beispiel-Szenario: ein Online-Video-Angebot mit unterschiedlichen Netzen [BaMW12]

wird neben einem Ressourcen-Endpunkt für die Video-Daten ein zusätzlicher Ressourcen-Endpunkt zu einem Anwendungsdienst auf dem Streaming-Server geöffnet, der die Steuerung des Videos erlaubt (pausieren, vor- und zurückspulen). Die Protokolle oder Adressen, die in den jeweiligen Netzen verwendet werden, spielen für die Anwendungen keine Rolle. Insbesondere sind zur Darstellung der Web-Seite und zum Herunterladen der Videos keine Anwendungsprotokolle notwendig, da hier Ressourcen-Endpunkte verwendet werden, die direkt den Inhalt der jeweiligen Datei bereitstellen. Lediglich zur Steuerung des Videos musste ein entsprechendes Protokoll in der Anwendung realisiert werden.

Abbildung 5.5 zeigt die Web-Seite des Video-Dienstes, die über eine URI mit dem Namensschema nenaweb:// abgerufen wurde. Die hier verwendete Anwendung – der Web-Browser – hat hierfür die in diesem Kapitel vorgestellte Anwendungsschnittstelle verwendet. Mit der URI beginnend mit nenacdn:// (rechts unten) wird vom Web-Browser der Download der Video-Datei über die Anwendungsschnittstelle für das spätere Abspielen gestartet. Über das Feld links unten kann das Video abgespielt werden. Dazu wird eine externe Anwendung gestartet, die ebenfalls die in diesem Kapitel vorgestellte Anwendungsschnittstelle verwendet. Wie oben beschrieben, nutzt diese dazu zwei Verbindungen, eine für die Video-Daten, eine weitere zur Steuerung des Videos. Tabelle 5.2 gibt einen Überblick über die jeweils verwendeten Primi-

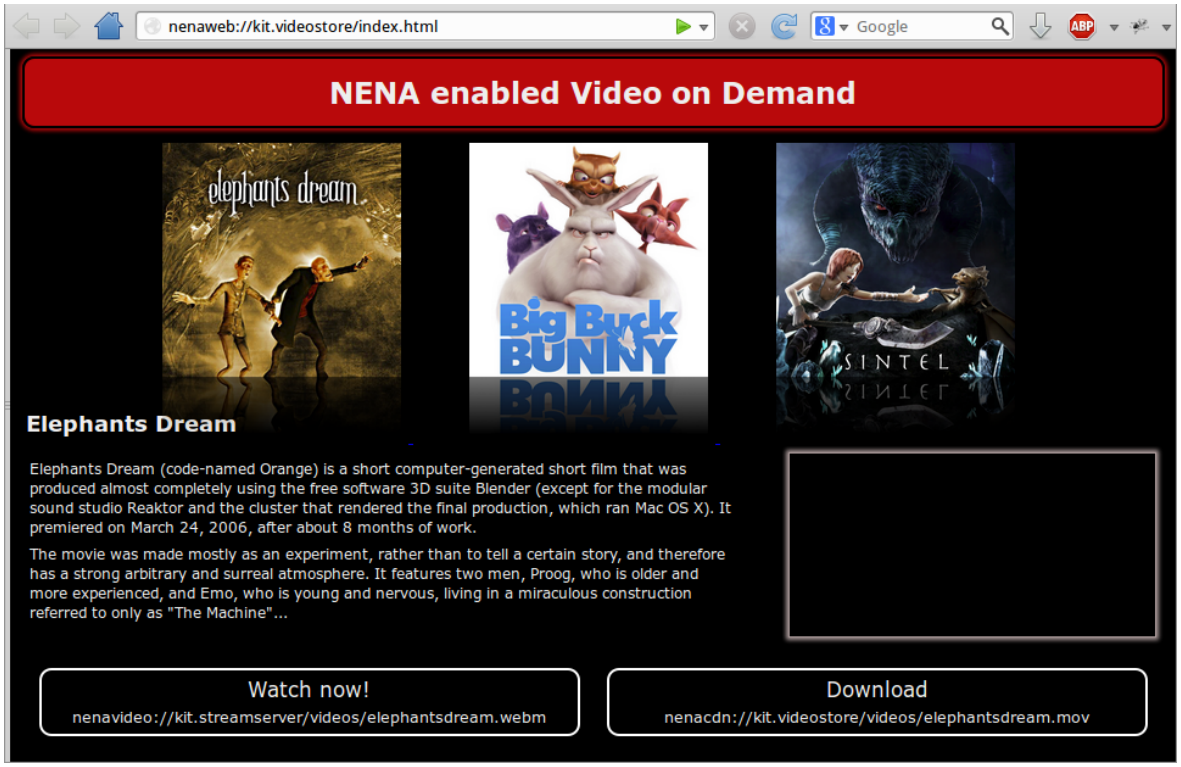


Abbildung 5.5 Beispiel-Szenario: die Benutzerschnittstelle (Web-Seite) des Online-Video-Angebots. Quelle & Copyright der Bilder und Videos: Blender Foundation | www.blender.org

tiven. Für die Steuerung der Video-Streaming-Anwendung über den zweiten Ressourcen-Endpunkt ist in der Praxis noch die Übergabe eines Sitzungsschlüssels notwendig, der über die Metadaten des ersten Ressourcen-Endpunktes abgefragt werden kann.

Damit konnte gezeigt werden, dass Anwendungen verschiedene Protokolle und Netze nutzen können, ohne genauere Kenntnis von ihnen zu besitzen. Die Verwendung eines existierenden Web-Browsers zeigt zudem, dass sich dieses Konzept nahtlos in heutige Umgebungen integrieren lässt.

5.5 Implementierung

Die im Rahmen dieser Arbeit entworfene protokollagnostische Anwendungsschnittstelle wurde als C++-Bibliothek implementiert. Diese stellt zudem eine C-Schnittstelle bereit, welche es auf einfache Art ermöglicht Adapter für verschiedensten Programmiersprachen zu erstellen. Ein solcher Adapter wurde für die Skriptsprache Python realisiert. Abbildung 5.6 skizziert das Zusammenspiel der Bibliothek mit Anwendungen und dem im nächsten Kapitel beschriebenen NENA-Rahmenwerk.

Anwendung	Verwendete Primitive und URI
Web-Server	BIND(„nenaweb://kit.videostore/“, {})
Content-Server Download	BIND(„nenacdn://kit.videostore/videos/“, {})
Content-Server Streaming	BIND(„nenavideo://kit.videostore/videos/“, {}) BIND(„nenavideo://kit.videostore/videoctrl“, {})
Endbenutzer Web-Seite	GET(„nenaweb://kit.videostore/index.html“, {reliable : 1})
Endbenutzer Download	GET(„nenacdn://kit.videostore/videos/ed.mov“, {reliable : 1})
Endbenutzer Streaming	CONNECT(„nenavideo://kit.videostore/videos/ed.webm“, {}) CONNECT(„nenavideo://kit.videostore/videoctrl“, {})

Tabelle 5.2 Verwendete API-Primitiven im Beispiel-Szenario

Kernstück der API-Implementierung bildet die *tmnet*-Bibliothek. Diese stellt alle notwendigen Primitiven und Konstanten bereit. Durch ein Plugin-System können verschiedene Implementierung dieser Primitiven realisiert werden. Die Implementierung für NENA greift dabei auf die *nenai*-Klassen zurück, die die API-Aufrufe über Interprozesskommunikation an NENA weiterreichen.

C/C++-Anwendungen können die *tmnet*-Bibliothek direkt einbinden und verwenden. Für Python-Anwendungen stellt der Python-Adapter die C-API über eine Python-Klasse bereit. Basierend auf der Python-Klasse wurde ein weiterer Adapter implementiert, der einen lokalen HTTP-Dienst bereitstellt. HTTP-Anfragen werden hierbei direkt in entsprechende API-Aufrufe umgesetzt und die Ergebnisse dieser Aufrufe wiederum über HTTP bereitgestellt. Somit ist es auf einfache Art und Weise möglich, HTTP-Anwendungen über die neue Anwendungsschnittstelle kommunizieren zu lassen. Für den Web-Browser Firefox wurde dazu eine Erweiterung geschrieben, die Zugriffe auf bestimmte URI-Namensschemata auf diesen HTTP-Dienst umleitet. Dies wurde bspw. für das Experiment genutzt, das in Abschnitt 5.4.3 beschrieben wurde.

Die Anwendungsanforderungen werden als Objekte in der JavaScript Object Notation (JSON) [Croc06] durch die Anwendung als Zeichenkette formuliert. Ein einfaches Beispiel für ein Anwendungsanforderungsobjekt ist

```
{
  "content-type": "image/jpeg",
  "reliable": 1
}.
```

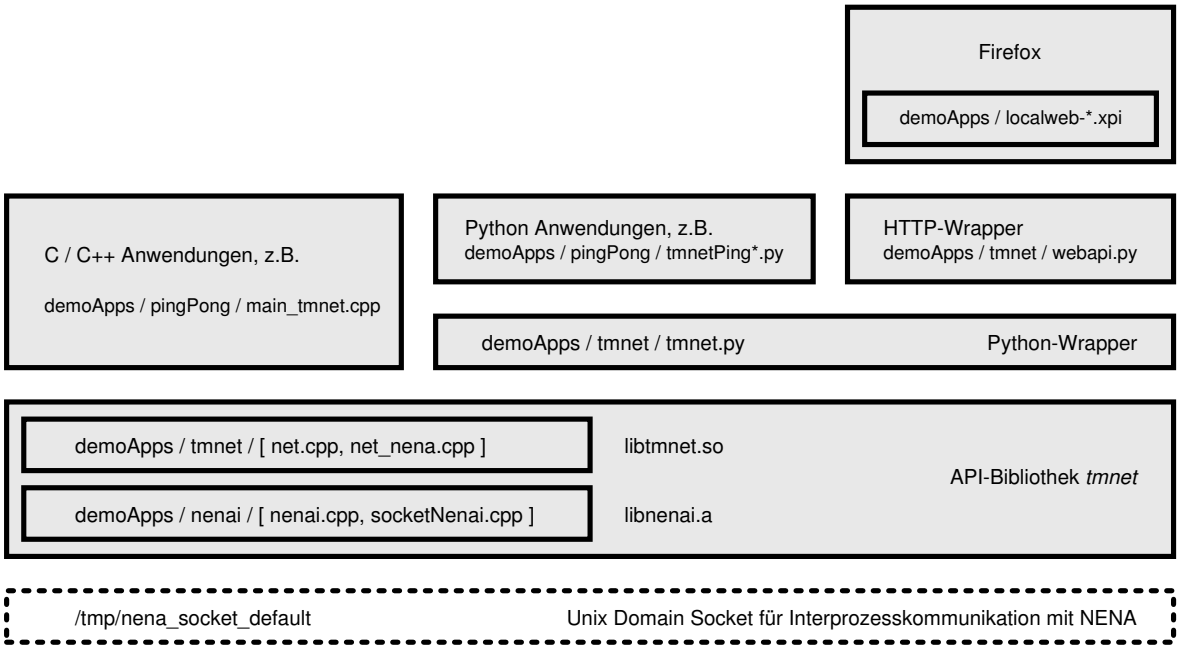


Abbildung 5.6 Überblick über die Implementierung der Anwendungsschnittstelle

Obwohl diese Art der Darstellung Mehraufwand durch das Parsen bedeutet, ist sie erweiterbar und portabel und bei Anwendungsentwicklern zur Zeit verbreitet. Darüber hinaus existieren effiziente Implementierungen von JSON-Parsern für viele Programmiersprachen. Hinzu kommt, dass dieser Mehraufwand lediglich bei neuen Kommunikationsanfragen anfällt, nicht aber während der Kommunikation selbst.

Die Implementierung der Anwendungsschnittstelle steht als Open-Source-Software als Teil des NENA-Quellcodes öffentlich zur Verfügung.[1]

[1] http://nena.intend-net.org/

6. NENA: Ein Rahmenwerk für Kommunikationssoftware

Durch die Anwendungsschnittstelle, die im letzten Kapitel vorgestellt wurde, müssen kommunikationsspezifische Aufgaben wie z. B. Namensauflösung und Protokollauswahl nicht mehr durch die Anwendung erfüllt werden. Mit Hilfe der durch die Schnittstelle zur Verfügung stehenden Informationen bzgl. eines Kommunikationswunsches übernimmt nun das Kommunikationssystem diese Aufgaben. Mit dem Szenario mehrerer, nebenläufig verwendeter Protokollstapel für unterschiedliche Netze kommen weitere Aufgaben hinzu, die das (Nach-)Laden und den Betrieb dieser Protokollstapel ermöglichen. Die kommunikationsspezifischen Aufgaben eines Gerätes des Kommunikationssystems werden in dieser Arbeit in einem *Kommunikationsrahmenwerk* zusammengefasst.

In diesem Kapitel wird die Netlet-basierte Knotenarchitektur NENA (Netlet-based Node Architecture [MaVZ11]) vorgestellt. NENA ist ein Kommunikationsrahmenwerk, welches notwendige Voraussetzungen auf End- und Zwischensystemen für maßgeschneiderte Kommunikationsnetze liefert. Die Anforderungen, die sich an NENA stellen, werden zunächst in Abschnitt 6.1 beschrieben. Abschnitt 6.2 geht auf das Rahmenwerk und dessen Architekturkonzepte im Detail ein. Die Abschnitte 6.3 und 6.4 beschreiben wichtige Mechanismen, die in NENA eingesetzt werden: die kontextspezifische Zustandshaltung und die Nachrichten- und Ereignisverarbeitung. Als Beispiel zur Verwendung der Konzepte und Mechanismen wird in Abschnitt 6.5 die einfach gehaltene *Simple Architecture* (SimpA) vorgestellt. Aspekte zur Reali-

sierung und Implementierung des Rahmenwerks beschreiben schließlich die Abschnitte 6.6 und 6.7.

6.1 Anforderungen

Die Anforderungen an NENA aus Sicht der Anwendungen ergeben sich aus der im letzten Kapitel vorgestellten Anwendungsschnittstelle. Über sie erhält NENA folgende Informationen bzgl. eines Kommunikationswunsches einer Anwendung:

- den angeforderten **Ressourcen-Name** als URI
- die verwendete **Primitive**
- die gewünschten **Anwendungsanforderungen**

Auf Basis dieser Informationen muss NENA einen passenden Protokollstapel auswählen und den erstellten Ressourcen- oder Ereignis-Endpunkt diesem Protokollstapel zuordnen. Bei der Verwendung von BIND – also dem Binden des Endpunktes auf einen Ressourcen-Namen, über den die Anwendung Ereignisse empfangen möchte – soll es zudem möglich sein, dass Ereignisse über verschiedene Protokollstapel und Netze empfangen werden können. Die Anforderungen aus Sicht der Anwendung bzw. Anwendungsschnittstelle lassen sich wie folgt zusammenfassen:

- **Anforderungsbasierte Netz- und Protokollauswahl.** Das Rahmenwerk muss basierend auf den von der Anwendung übergebenen Informationen ein geeignetes Protokoll auswählen. Die benannte Ressource kann dabei prinzipiell über mehrere Netze erreichbar sein, wobei in jedem Netz unterschiedliche Protokolle zur Verfügung stehen können.
- **Protokollunabhängige Namensbindung.** Eine Anwendung, die Dienste oder Inhalte anbietet, muss sich zunächst protokollunabhängig an einen Namen binden können. Die im Rahmenwerk vorhandenen Protokollstapel müssen selbstständig für eine geeignete Dienst- oder Inhaltsankündigung sorgen, die abhängig von der Netzwerkarchitektur unterschiedlich aussehen kann.
- **Protokollunabhängiger Kommunikationskontext.** Informationen wie Ziel der Kommunikation und Anforderungen an die anstehenden Kommunikation stehen bereits durch den API-Aufruf zur Verfügung und müssen zwischengespeichert werden. Netzwerkereignisse wie eingehende Verbindungsanfragen oder Verbindungsabbrüche, die für die Anwendung relevant sind, müssen in einer generischen Form an die Anwendung weitergegeben und von ihr verstanden werden, ohne dass diese spezielle Protokolleigenschaften kennen muss.

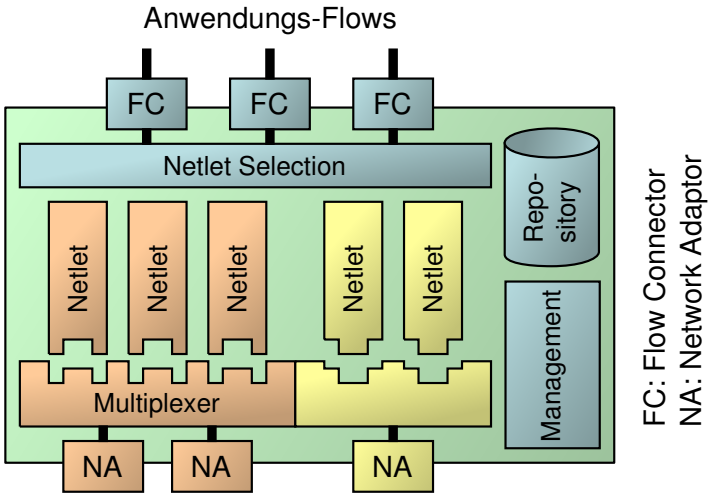


Abbildung 6.1 Die Netlet-basierte Knotenarchitektur NENA

Aus Sicht der Protokollstapel muss das Rahmenwerk einen nebenläufigen, unabhängigen Betrieb der Kommunikationssoftware dieser Protokollstapel gewährleisten. Dies bedeutet, dass Protokollstapel kein Wissen über andere vorhandene Protokollstapel haben müssen und dass keine Einschränkungen zur Realisierung ihrer Protokolle entstehen. Daraus leiten sich folgende Anforderungen an das Rahmenwerk ab:

- **Architekturagnostische Rahmenwerksfunktionen.** Das Rahmenwerk darf keine Annahmen über die Netzwerkarchitektur treffen. Adressierung, Protokolle und Paketformate dürfen beispielsweise nicht durch das Rahmenwerk beeinflusst werden.
- **Architekturspezifische Netzschnittstellen.** Das Rahmenwerk muss eine beliebige Anzahl an Netzschnittstellen unterstützen, die Netzzugänge über verschiedene Kommunikationshardware zu den Dienstanbieternetzen bereitstellen. Die Netzschnittstellen müssen dabei eindeutig einer bestimmten Kommunikationssoftware zuordenbar sein, und die übertragenen Informationen dürfen nicht vom Rahmenwerk interpretiert oder verändert werden.

6.2 Aufbau des Rahmenwerks

Der Aufbau des NENA-Rahmenwerks ist in Abbildung 6.1 skizziert. Das Rahmenwerk in dieser Form befindet sich auf den Systemen der Endbenutzer, und Anwendungen auf diesen Systemen wie z. B. Web-Browser sprechen darüber entfernte Ressourcen an.

Das Rahmenwerk verbindet sich mit verschiedenen Dienstanbieternetzen über sog. *Network Adaptors* (NA). Network Adaptors werden dabei über Me-

chanismen des Betriebssystems bereitgestellt und repräsentieren entweder einen physischen oder einen virtuellen Adapter, über den Nachrichten- oder Byte-Ströme in das Dienstanbieternetz gesendet und von diesem empfangen werden.

Die Network Adaptors werden dem *Multiplexer* zugeordnet, der zur Kommunikationssoftware der Netzwerkarchitektur gehört, die in dem Netz betrieben wird. Der Multiplexer kapselt die *Basisprotokollschicht* der jeweiligen Netzwerkarchitektur. Im traditionellen Internet-Modell ist dies die Netzwerk- bzw. Vermittlungsschicht. Individuelle Protokolle, die auf der Basisprotokollschicht der jeweiligen Netzwerkarchitektur aufsetzen, werden als *Netlets* gekapselt. Multiplexer und Netlets stellen *Module* dar, die im Betrieb hinzugefügt oder entfernt werden können. Die Kommunikationssoftware einer Netzwerkarchitektur besteht dabei aus einem Multiplexer und einem oder mehreren Netlets.

Wenn sich eine Anwendung mit dem Rahmenwerk verbindet, wird für jeden Endpunkt zunächst ein *Flow Connector* (FC) erzeugt, der Informationen zu dem von der Anwendung angeforderten Kommunikationsdienst zwischenspeichert. Dazu gehören der Name des Kommunikationsziels und die Anforderungen an den Kommunikationsdienst (bspw. bzgl. Zuverlässigkeit). Außerdem wird über den Flow Connector die Interprozesskommunikation mit der Anwendung gekapselt.

Die *Netlet Selection* Komponente bildet das Bindeglied zwischen den Flow Connectors und den Netlets: Abhängig von den Parametern, die die Anwendung beim Aufruf über die Anwendungsschnittstelle angibt, wird eine entsprechende Netlet-Auswahl getroffen. Außerdem wird pro Flow Connector ein sogenanntes *Flow State* Objekt erzeugt, an welches Flow-spezifische Zustandsinformationen angehängt werden können.

Einstiegspunkt für Verwaltungsaufgaben ist die *Management*-Komponente. Da für einige Verwaltungsaufgaben jedoch Wissen über die jeweilige Netzwerkarchitektur notwendig ist, können Verwaltungsanfragen an die Module der jeweiligen Kommunikationssoftware weitergereicht werden.

Die *Repository* Komponente ist für das Laden der Module zuständig und kapselt den Zugriff sowie das Beschaffen. Zu den Modulen gehören neben Netlets und Multiplexern auch sog. *Servlets*. Servlets (nicht abgebildet, siehe Abschnitt 6.2.1.3) können Netzdienste kapseln, die keiner bestimmten Netzwerkarchitektur zugeordnet werden müssen bzw. die architekturübergreifende Dienste anbieten.

Nach diesem Überblick über das Rahmenwerk, werden in den folgenden Abschnitten nun die NENA-Komponenten näher beschrieben.

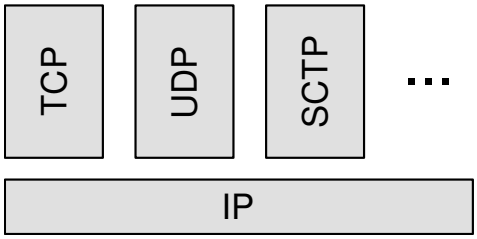


Abbildung 6.2 Vereinfachte Darstellung der Internet-Protokollarchitektur

6.2.1 Module

Um ein Laden der Kommunikationssoftware zur Laufzeit zu ermöglichen, muss diese in für die Distribution geeignete Module „verpackt“ sein. Eine Kommunikationssoftware für eine Netzwerkarchitektur kann aus verschiedenen Protokollen bestehen, wobei es in der Regel ein Basisprotokoll (gekapselt in einem Multiplexer) gibt, über die alle höherschichtigen Protokolle (gekapselt in Netlets) kommunizieren. Am besten lässt sich dies an der heutigen Internet-Protokollarchitektur veranschaulichen (Abb. 6.2): Das Basisprotokoll bildet das Internet-Protokoll IP. Darauf aufbauend bieten Transportprotokolle wie TCP weitere Dienste an. Die Basisprotokollschicht dient dabei als Multiplexer für die darüberliegenden Protokolle und legt im konkreten Fall von IP u. a. auch das Kommunikationsparadigma der hostbasierten (genauer: schnittstellenbasierten) Kommunikation fest und übernimmt weitere Aufgaben (z. B. zur Vermittlung).

Neben Multiplexer und Netlets dienen Servlet-Module dazu, Kommunikationsdienste zu kapseln, die über verschiedene Netzwerkarchitekturen hinweg angeboten werden können. In den folgenden Unterabschnitten werden diese drei Modul-Konzepte detailliert. Alle Module werden dabei eindeutig über URIs der Form

`multiplexer://`<Name>
`netlet://`<Name>
`servlet://`<Name>

identifiziert. Dies ist u. a. für das Laden der Module aber auch für die lokale Kommunikation untereinander notwendig.

6.2.1.1 Basisprotokollschicht: Multiplexer

Die Aufgaben des Multiplexers – also des Moduls für die Basisprotokollschicht einer Netzwerkarchitektur – gehen oft über ein reines Multiplexen von Protokollen hinaus. Pro Netzwerkarchitektur gibt es genau einen Multiplexer. Aufgrund dieser Beziehung, kann ein Multiplexer genutzt werden, um allgemeine, vom Transport unabhängige Aufgaben zu übernehmen.

Empfehlungen, welche Aufgaben im Multiplexer realisiert werden sollten, sind:

- **Namensauflösung.** Die Namensauflösung sollte so spät wie möglich erfolgen, damit Abhängigkeiten zur Adressierungsart in darüberliegenden Protokollen gering gehalten werden.
- **Adressierung und Vermittlung.** Die Entscheidung, über welchen Network Adaptor eine Nachricht versendet werden soll, hängt von der Adresse ab und sollte im Multiplexer erfolgen.
- **Basispaketkopf.** Jede versendete und empfangene Nachricht muss für die Netzwerkarchitektur interpretierbar sein. Ein minimaler Paketkopf, der die netzwerkarchitekturspezifischen Adressen und Informationen zum (De-)Multiplexen der Nachrichten enthält, sollte im Multiplexer hinzugefügt bzw. von ihm ausgewertet werden.

Vorgaben, welche dieser Aufgaben tatsächlich im Multiplexer zu realisieren sind, werden von NENA jedoch keine gemacht und sind für das Rahmenwerk selbst nicht von Belang.

6.2.1.2 Protokolle: Netlets

Protokolle, die auf die Basisprotokollschicht einer Netzwerkarchitektur aufsetzen (z. B. Transportprotokolle), werden in Netlets gekapselt. Ein Netlet ist dabei immer einem bestimmten Multiplexer zugeordnet, wobei es mehrere Netlets geben kann, die dem selben Multiplexer zugeordnet sind.

Ein Netlet muss dabei nicht nur genau ein Protokoll enthalten, sondern kann auch aus mehreren bestehen. Beispielsweise kann das Anwendungsprotokoll HTTP und das Transportprotokoll TCP in einem Netlet zusammengefasst werden. Insbesondere können Netlets aus einzelnen Bausteinen bestehen, die Protokollmechanismen umsetzen. Diese wiederum können beliebig miteinander über Ereignisse oder Zustandsobjekte interagieren, was den in Kapitel 4 vorgestellten Ansatz der Protokollschablonen ermöglicht.

Für den weiteren Verlauf in diesem Kapitel kann ein Netlet allerdings als ein Modul betrachtet werden, das beliebig komplexe Kommunikationsprotokolle enthalten kann. Unterschieden wird hier lediglich zwischen Übertragungs-Netlets, die Anwendungsanfragen und -daten übertragen können, und Kontroll-Netlets. Letztere übernehmen Kontroll- und Management-Aufgaben wie Routing und bedienen keine Anwendungsanfragen.

6.2.1.3 Netzdienste: Servlets

Innerhalb von Multiplexern und Netlets können netzwerkarchitekturspezifische Protokolle realisiert werden. Allerdings sind auch Kommunikations-

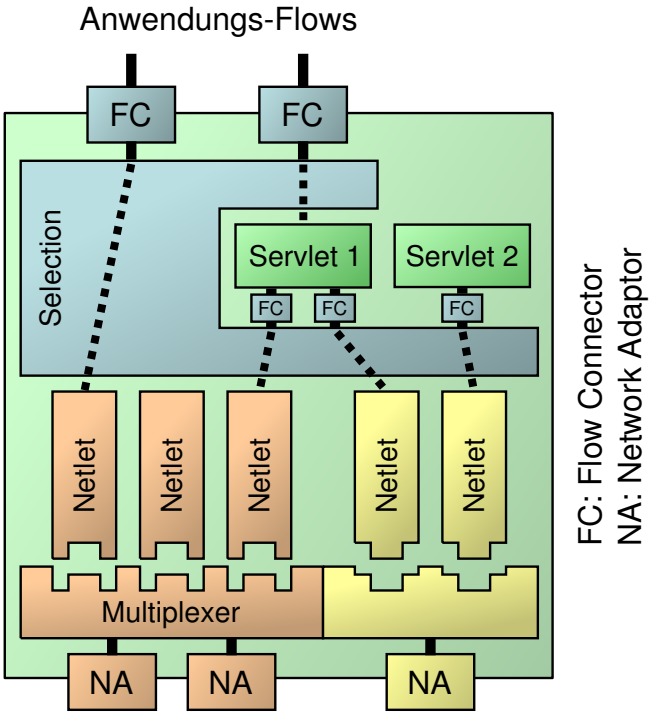


Abbildung 6.3 Integration von Servlets in NENA zur Realisierung von Overlay- und Middleware-Funktionalitäten.

dienste denkbar, die über verschiedene Netze hinweg realisiert werden können. Heute sind diese oft als Overlay innerhalb einer Middleware realisiert, die sich zwischen Anwendung und konkreten Netzwerkarchitekturen wiederfindet.

Mit dem Konzept der Servlets (Abbildung 6.3) ist es möglich, solche Dienste innerhalb von NENA zu realisieren, ohne dass eine zusätzliche Middleware mit individueller Anwendungsschnittstelle notwendig wird. Ähnlich wie Anwendungen benötigen Servlets Netlets zur Kommunikation. Hierfür wird analog zu Anwendungs-Flows ein Flow Connector für das Servlet erzeugt, wenn dieses einen Kommunikationswunsch hat. Mit mehreren Flow Connectors ist es dem Servlet auch möglich, gleichzeitig über mehrere Netzwerkarchitekturen zu kommunizieren und somit architekturübergreifende Dienste anzubieten (vgl. Servlet 1 in Abb. 6.3). Wenn Servlets ihre Dienste auch lokalen Anwendungen zur Verfügung stellen, können sie bei der Netlet-Auswahl mit berücksichtigt werden und so Middleware-Funktionalität den Anwendungen bereitstellen. Andere Servlets können dabei auch Dienste anbieten, die nicht von lokalen Anwendungen direkt genutzt werden, wie z. B. Server für Namensdienste (Servlet 2). Diese Konzepte werden im Rahmen einer Realisierung eines Delay-Tolerant Networking Dienstes in Kapitel 7 näher betrachtet.

6.2.2 Netzwerkarchitekturen

Die Kommunikationssoftware einer Netzwerkarchitektur, die auf ein bestimmtes Dienstanbieternetz zugeschnitten ist, wird in NENA durch eine Menge von Modulen repräsentiert. Zu diesen Modulen gehören pro Netzwerkarchitektur genau ein Multiplexer und eine beliebige Anzahl von Netlets. Zusätzlich müssen durch diese Module eine Menge von Schnittstellen bereitgestellt werden, die die Integration der Kommunikationssoftware in das Rahmenwerk erlauben und den koexistenten Betrieb mehrerer Netzwerkarchitekturen ermöglichen. Diese Schnittstellen sind:

- eine Schnittstelle zur Kandidatenauswahl für einen anstehenden Kommunikationswunsch einer Anwendung,
- eine Schnittstelle zur Registrierung von Anwendungsdiensten (d. h. Dienste, die durch Anwendungen auf dem lokalen System für andere, entfernte Anwendungen bereitgestellt werden) und
- eine Management-Schnittstelle zur netzwerkarchitekturspezifischen Ressourcen-Verwaltung.

Vorgaben, durch welche Module diese Schnittstellen zu implementiert sind, werden keine gemacht. Komponenten der Netzwerkarchitektur müssen sich an den entsprechenden Stellen im NENA-Rahmenwerk registrieren. Diese Flexibilität bei der Realisierung erlaubt es dem Netzwerkarchitekten, besser auf die Bedürfnisse der einzelnen Netzwerkarchitekturen einzugehen.

6.2.2.1 Kandidatenauswahl: Request Mapper

Der Prozess zur Netlet-Selektion wird in Abschnitt 6.2.3 beschrieben. Von jeder im System vorhanden Netzwerkarchitektur wird dazu jedoch eine Liste mit in Frage kommenden Kandidaten angefordert. Damit Netlets einer Netzwerkarchitektur bei der Selektion mit berücksichtigt werden, muss die Netzwerkarchitektur ein Modul bei der Netlet Selection Komponente als *Request Mapper* registrieren. Die Netlet Selection fordert dann bei einer anstehenden Selektion eine Liste von Kandidaten an, die zu den angegebenen Parametern passen. Diese Parameter beinhalten den Namen des Kommunikationsziels, die Methode der Kommunikation, und die Kommunikationsanforderungen (vgl. hierzu Kapitel 5). Der Request Mapper hat dann die Aufgabe, die angegebenen Parameter auf eine Liste von Netlets abzubilden, die als Kandidaten für die angeforderte Kommunikation in Frage kommen.

6.2.2.2 Registrierung von Anwendungsdiensten

Ähnlich zum Request Mapper, muss sich auch ein Modul der Netzwerkarchitektur bei der Netlet Selection Komponente registrieren, das über neue

Anwendungsdienste informiert werden möchte. Eine Anwendung registriert sich am Rahmenwerk mit einem öffentlichen Namen, unter dem sie für andere, entfernte Anwendungen erreichbar sein möchte, mittels BIND. Wenn dies geschieht, wird ein Ereignis von der Netlet Selection Komponente ausgelöst, das an alle dafür registrierten Module gesendet wird. Die jeweilige Architektur hat dann die Aufgabe, die notwendigen Reservierungen und/oder Ankündigungen im Netz vorzunehmen. Dies könnte z. B. ähnlich zu TCP und UDP rein lokal geschehen, indem Ports geöffnet werden. Allerdings sind auch komplexere Aktionen möglich, die den Dienst im Netz verbreiten, z. B. durch Anlage von Einträgen in verteilten Hash-Tabellen oder durch Dienstankündigung mit existierenden Namensdiensten (z.B. Multicast-DNS [ChKr13], vgl. Abschnitt 5.4.2).

6.2.2.3 Management-Schnittstelle

Management-Anfragen an das NENA-Rahmenwerk benötigen Unterstützung in den jeweiligen Netzwerkarchitekturen für deren Realisierung. Insbesondere, wenn Ressourcen wie Speicher oder Bandbreite zwischen mehreren Architekturen aufgeteilt werden müssen, ist eine Koordination zwischen den Netzwerkarchitekturen notwendig. In [WGFZ11] wird dazu eine hierarchische Management-Struktur vorgeschlagen, dessen Wurzelknoten in der zentralen Management-Komponente des NENA-Rahmenwerks liegt. Von dort ausgehend gibt es eine Management-Komponente pro Netzwerkarchitektur, die Management-Aufgaben von der zentralen Komponente entgegen nimmt und Überwachungsdaten über die Ressourcen-Nutzung an die zentrale Komponente zurückliefert. Wie auch bei den anderen Schnittstellen wird nicht vorgegeben, durch welches Modul diese Schnittstelle zu realisieren ist.

6.2.3 Netlet-Auswahl

Die Entkopplung von Anwendungen und Netzwerkprotokollen erfordert eine geeignete Wahl der Netlets basierend auf den Informationen, die von der Anwendung für eine Kommunikationsanfrage übergeben werden. Die grundsätzliche Feststellung, ob ein Netlet für eine Anfrage geeignet ist, erfolgt durch die Kommunikationssoftware, zu der das Netlet gehört (vgl. Abschnitt 6.2.2.1). Danach muss die Netlet Selection aus allen Kandidaten ein Netlet auswählen.

Als erster Schritt im Auswahlprozess wird eine Filterung aller vorhandenen Netlets durchgeführt. Dies geschieht im Zuge der Abfrage der Request Mapper der einzelnen, in NENA vorhandenen Netzwerkarchitekturen. Dabei wird bereits geprüft, ob die Ressource potentiell über eine Netzwerkarchitektur erreichbar ist: Durch die Verwendungen von URI-Namensräumen können so bspw. einfache, lokale Entscheidungen getroffen werden. Außerdem können Anforderungen z. B. bzgl. Zuverlässigkeit bereits dazu führen, dass bestimmte Netlets ausgeschlossen werden können.

Mit der Kandidatenliste ist es dann Aufgabe der Netlet Selection, eine Rangfolge der Kandidaten festzulegen, aus der dann der Kandidat ausgewählt wird, der den Anwendungsanforderungen am besten entspricht. Die Bestimmung dieser Rangfolge ist jedoch nicht trivial und hängt u. a. von der Gewichtung verschiedener Anforderungen ab, die zudem nicht immer vergleichbar sind. Verfahren aus entscheidungstheoretischen Ansätzen können hier helfen, normalisierte Nutzwerte für einzelne Eigenschaften zu bestimmen, und diese Nutzwerte dann miteinander in Relation zu setzen, um somit einen Gesamtnutzwert für einen Kandidaten zu bestimmen [VMWZ+09, Völk12]. Mit entscheidungstheoretischen Ansätzen können auch große Mengen an Kandidaten und Eigenschaften im Auswahlprozess berücksichtigt werden, und so eine möglichst passende Alternative ausgewählt werden.

In vielen Fällen wird aber bereits durch den Namen oder durch das Namensschema der angeforderten Ressource ein bestimmtes Dienstanbieternetz festgelegt, wenn die Ressource nur durch einen Anbieter bereitgestellt wird. Innerhalb dieses Netzes wird man nur noch eine kleine Auswahl an Protokollen vorfinden, die parallel betrieben werden. Durch Vorfilterung der Alternativen anhand von notwendigen qualitativen Kriterien kann diese Anzahl weiter reduziert werden, wodurch nur noch wenige Alternativen übrig sind, die sich anhand von quantitativen Eigenschaften unterscheiden. Um die quantitativen Eigenschaften jedoch genau bestimmen zu können, muss meist der Netzwerkpfad mit berücksichtigt werden. Nicht jede Netzwerkarchitektur stellt hier jedoch kontinuierliche Messung an, sodass solche Eigenschaften nicht immer vorhanden sind.

6.2.4 Repository und Management

Das in Abschnitt 1.2 eingeführte logisch zentrale Repository wird in NENA durch die Repository-Komponente repräsentiert. Anfragen zur Beschaffung und Instantiierung neuer Module werden an diese Komponente gerichtet. Liegen die notwendigen Daten bereits auf dem lokalen System vor, werden die Module geöffnet und geladen. Nach der Initialisierung stehen sie dann für passende Kommunikationswünsche bereit. Wenn es sich jedoch um Module handelt, die bisher nicht lokal vorhanden sind, ist die Repository-Komponente für das Auffinden und Nachladen zuständig. Dabei ist sie auch für die Authentizitäts- und Integritätsprüfung verantwortlich [HaWi11], sowie für das Einspielen von Aktualisierungen der bereits vorhandenen Module.

Die Management-Komponente ist der Einstiegspunkt für alle Verwaltungsaufgaben. Neben Überwachungsaufgaben und Ressourcen-Verwaltung der verschiedenen Netzwerkarchitekturen (vgl. Abschnitt 6.2.2.3), ist diese Komponente auch dafür zuständig Kommunikationsanfragen weiterzuverarbeiten, die durch keine bisher vorhandene Netzwerkarchitektur erfüllt werden können. Neben der Beauftragung der Repository-Komponente die fehlenden

Module nachzuladen, ist ggfs. auch die Anbindung an ein neues Dienstanbieternetz notwendig [WiHa12].

6.3 Zustandsverwaltung: Flow States

In NENA wird auf die strikte Trennung von Code und Daten in den Modulen gesetzt, wobei den Modulen die Daten als Kontextinformationen bei jedem Ereignis übergeben werden. Dies ermöglicht die parallele Ausführung des selben Moduls für unterschiedliche Anwendungs-Flows, ohne dass das Modul selbst auf Synchronisationsmechnismen zurückgreifen muss.

Ein Großteil der Zustandsinformationen eines Protokolls ist dabei Flow-spezifisch. Der Kontext eines Flows wird durch einen Flow Connector erstellt und durch sog. Flow State Objekte (FSO) gekapselt [MaWi13a]. Protokolle, die diesen Flow bedienen, können eigene Zustandsinformationen als Zustandsobjekte (State Objects, SOs) anhängen und durch einen eindeutigen Schlüssel referenzieren. FSOs können dabei beliebige und beliebig viele SOs verwalten.

Das FSO kennt die drei Zustände *valid* (gültig), *ended* (beendet) und *stale* (ungültig). Am Anfang der Lebenszeit ist das FSO im Zustand *valid*. Es wechselt in den Zustand *ended*, wenn entweder die lokale Anwendung die Verbindung zu NENA schließt oder wenn ein Protokollbaustein diesen Zustand setzt. Dies geschieht bspw. bei verbindungsorientierten Protokollen, die Informationen darüber haben, wenn der entfernte Kommunikationspartner die Kommunikation beendet. Analog dazu wird in den Zustand *stale* gewechselt, wenn entweder die lokale Anwendung die Verbindung zu NENA abbricht (bspw. durch den Absturz der Anwendung) oder wenn ein Protokollbaustein feststellt, dass der entfernte Kommunikationspartner nicht (mehr) erreichbar ist. Das FSO wird nach dem Wechsel in die beiden Endzustände *ended* und *stale* nicht sofort gelöscht, sondern wird noch für eine gewisse Dauer vorgehalten. Diese Dauer ist abhängig von den Anforderungen der dazugehörigen Netzwerkarchitektur und kann bspw. dazu genutzt werden um veraltete Nachrichten zu filtern, die nach dem Ende der Kommunikation noch empfangen werden.

Neben den Zustandsinformationen der einzelnen involvierten Protokolle und Protokollmechanismen kann das FSO auch weitere Kontextinformationen wie Ressourcenverbrauch und Flow-Statistiken verwalten. Außerdem spielt das FSO ein wichtige Rolle zur internen Flusskontrolle: Da die Kommunikation zwischen den Komponenten nachrichtenbasiert ist, kann es zu wachsenden Puffern kommen, wenn eine Nachricht in einer Komponente eine längere Verweildauer aufweist als in einer anderen. Dies kann insbesondere passieren, wenn die Nachricht über ein Netzwerk geschickt wird und ein Transportprotokoll mit Sendewiederholungsmechanismus (Automatic Repeat Request, ARQ) die zu sendenden Nachrichten zwischenpuffert. Mittels des FSO kann

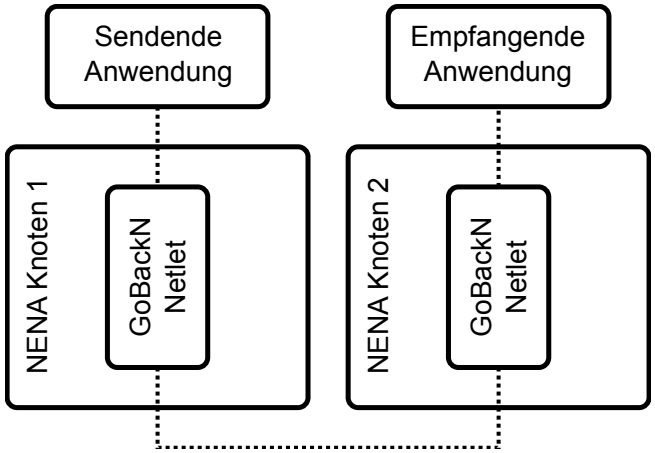


Abbildung 6.4 Versuchsaufbau zur Veranschaulichung der Floating Messages.

nun die Flusskontrolle des Transportprotokolls Einfluss auf das von NENA verwendete nachrichtenbasierte System nehmen, indem es die Anzahl der sich im sendenden System befindlichen Nachrichten limitieren kann. Wird dieses Limit erreicht, darf der Flow Connector keine weiteren Nachrichten von der Anwendung entgegennehmen. Bei einer blockierenden Netzwerk-API bedeutet das, dass die Anwendung blockiert werden muss.

Die Nachrichten, die im sendenden System verarbeitet oder gepuffert werden, werden hier als *Floating Messages* bezeichnet. Nachrichten, die bereits gesendet aber noch nicht bestätigt wurden, werden dagegen als *Flying Messages* bezeichnet. Bei einem traditionellen ARQ-Mechanismus befindet sich in der Regel eine Kopie jeder Nachricht, die unterwegs ist aber noch nicht bestätigt wurde, im Sendewiederholungspuffer. Die Größe des Puffers entspricht der aktuellen Größe des Sendefensters. Das Limit der Floating Messages kann somit auf die aktuelle Größe des Sendefensters gesetzt werden: Ist der Sendewiederholungspuffer voll, werden keine weiteren Daten von der Anwendung entgegengenommen.

Das so gesetzte Limit muss allerdings als „weiches“ Maximum (Soft-Maximum) gewertet werden, da es kurzzeitig überschritten werden kann. Dies ist z. B. dann der Fall, wenn Kopien der Nachrichten durch Protokollmechanismen angelegt werden. Dieses Verhalten wird mit dem einfachen Szenario aus Abbildung 6.4 unter Verwendung eines Go-Back-N ARQ-Mechanismus veranschaulicht: Eine Anwendung auf Knoten 1 sendet dabei Daten über einen blockierenden Endpunkt an eine Anwendung auf Knoten 2. Beide kommunizieren dabei über ein Netlet, das einen Go-Back-N ARQ-Baustein enthält. Dieser Baustein hat ein statisches Sendefenster der Größe 128 Nachrichten und sendet Bestätigungen (Acknowledgements) verzögert

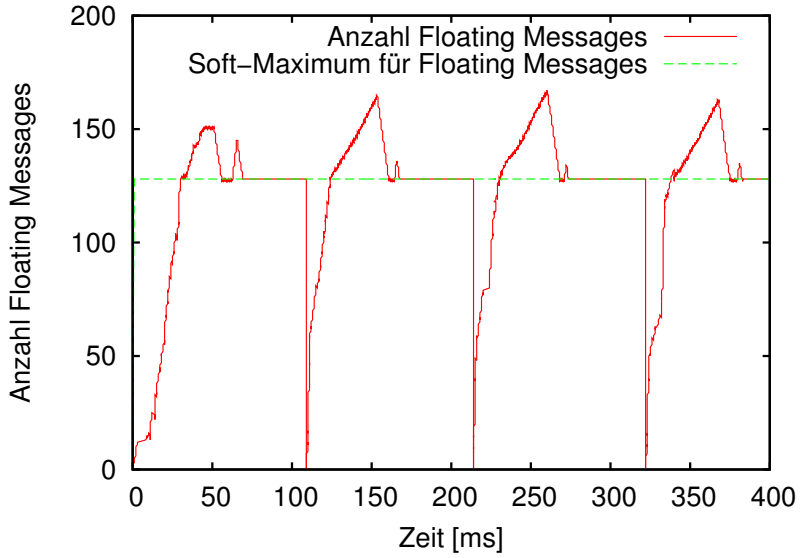


Abbildung 6.5 Anzahl der Floating Messages über die Zeit bei einem Go-Back-N ARQ (N = 128 Nachrichten, verzögerte Bestätigungen nach 100 ms).

und kumuliert nach 100 ms. Das Soft-Maximum der Floating Messages wird dabei auf die Größe des Sendefensters, also auf 128 Nachrichten gesetzt.

Abbildung 6.5 zeigt den Verlauf der Anzahl der im sendenden NENA-Knoten befindlichen Floating Messages für diesen Anwendungs-Flow mit eingetragenem Soft-Maximum. Dieses Soft-Maximum wird dabei immer dann überschritten, wenn zu sendende Nachrichten aus dem Sendewiederholungspuffer dupliziert werden. Das Soft-Maximum kann dabei um bis zu 100 % überschritten werden, was beispielsweise dann passiert, wenn eine Retransmission angestoßen wird und der gesamte Sendewiederholungspuffer erneut übertragen wird. In diesem Fall werden Kopien der Nachrichten (bzw. der Referenzen) angelegt, was die Anzahl der Floating Messages solange kurz erhöht, bis die Kopien das Rahmenwerk verlassen haben.

Sobald die Anzahl der Floating Messages im sendenden System das Soft-Maximum unterschreitet (was insbesondere nach dem Eintreffen der kumulierten Bestätigung geschieht, da dann die Kopien der bestätigten Nachrichten freigegeben werden), wird ein Flow State Ereignis ausgelöst. Dieses Ereignis erlaubt es beispielsweise dem Flow Connector, weitere Daten von der Anwendung entgegen zu nehmen.

6.4 Nachrichten- und Ereignisverarbeitung

Um eine effiziente Entwicklung von Netzwerkprotokollen zu ermöglichen, sollte sich der Entwickler der Protokollbausteine hauptsächlich auf die

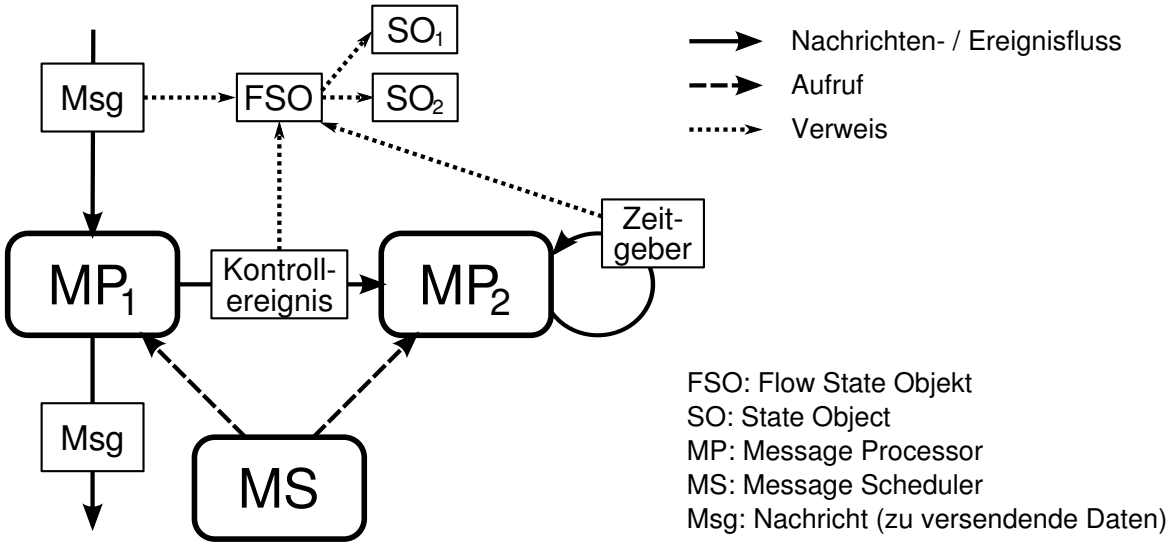


Abbildung 6.6 Ereignisbasierte Basiskomponenten in NENA und ihre Zusammenhänge.

Netzwerkkommunikation konzentrieren. Von Optimierungsaufgaben zur effizienteren Ausführung der Protokollsoftware muss der Entwickler dabei möglichst entlastet werden. Eine heute wichtige Eigenschaft von Rechner-Hardware ist allerdings, dass diese mehrere Ausführungseinheiten besitzt. Damit Software einen Nutzen davon tragen kann, muss diese so geschrieben sein, dass sie parallelisiert ausführbar ist.

Protokollsoftware hat die Eigenschaft, dass sie ausschließlich ereignisgesteuert ist: Sie wird nur dann ausgeführt, wenn Anwendungsdaten zu versenden sind, wenn Daten empfangen werden, wenn Ereignisse durch andere Protokollmechanismen ausgelöst werden oder wenn ein Zeitgeber abgelaufen ist. Somit muss der Entwickler lediglich die Funktionen implementieren, die ausgeführt werden, wenn die jeweiligen Ereignisse eintreten (sog. Handler-Funktionen). Aus diesem Grund wurde für die Realisierung und für die Interaktion von (Protokoll-)Komponenten in NENA ein ereignisbasiertes System gewählt, was in diesem Abschnitt erläutert wird. Wie in Abschnitt 4.1.3 beschrieben, sind Nachrichten, Kontrollereignisse und Zeitgeber jeweils Spezialisierungen des allgemeinen Begriffs „Ereignis“.

Ereignisverarbeitende Komponenten in NENA – z. B. Netlets und Multiplexer, aber auch einzelne Protokollbausteine – werden als *Message Processors* (MPs) bezeichnet. Diese können Nachrichten und andere Ereignisse entgegennehmen, verarbeiten, weiterversenden oder neue erzeugen. Ein *Message Scheduler* (MS) verwaltet eine Menge von Message Processors (MPs) und bedient diese, wenn Ereignisse, die an diese MPs gerichtet sind, auf deren Abarbeitung warten. Dies geschieht durch den Aufruf der entsprechenden

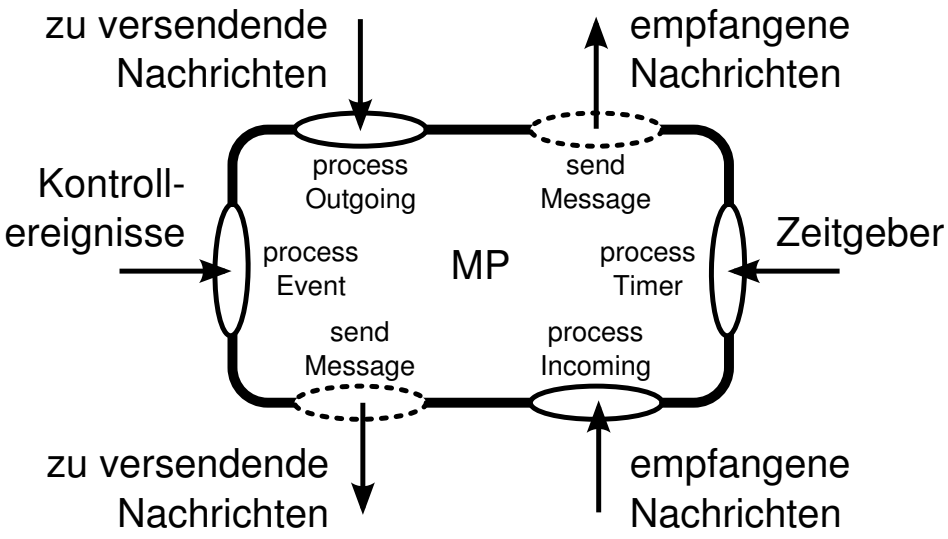


Abbildung 6.7 Basisklasse für ereignisverarbeitende Komponenten in NENA: Message Processor (MP).

Handler-Funktion des MPs mit dem wartenden Ereignis als Parameter. Das Zusammenspiel zwischen Nachrichten und anderen Ereignissen, Flow State Objekten (FSOs) und MPs ist in Abbildung 6.6 dargestellt. Nachrichten, Kontrollereignisse und Zeitgeber verweisen dabei immer auf ein FSO, das den Kommunikationskontext repräsentiert. MP-spezifische Informationen werden an dieses FSO als State Objects (SOs) angehängt und sind damit mit jeder Nachricht oder jedem Ereignis verfügbar (vgl. Abschnitt 6.3).

6.4.1 Message Processors

NENA stellt eine ereignisverarbeitende Basisklasse für Protokollkomponenten bereit: den Message Processor (MP). Dieser unterscheidet zwischen Nachrichten, die von der Anwendung bzw. vom Netzwerk kommen, und Ereignissen wie Zeitgeber oder individuellen Kontrollereignissen anderer MPs (Abbildung 6.7).

Der Versand von Ereignissen geschieht durch einen Methodenaufruf (*sendMessage*), dem als Parameter lediglich das Ereignis selbst übergeben wird. Das Ereignis verfügt dabei über verschiedene Attribute:

- Sender- und Empfänger-MP des Ereignisses;
- eine Liste mit Eigenschaften, die als (Schlüssel, Wert)-Tupel abgelegt werden;
- ein Flow State Objekt, welches diesem Ereignis zugeordnet ist;
- weitere Daten.

Das Flow State Objekt erlaubt es dem MP dabei, Informationen über den Kommunikationskontext zu erhalten und seine eigenen Zustandsinformationen ebenfalls an dieses Objekt anzuhängen. „Weitere Daten“ sind hierbei abhängig vom Ereignistyp. Eine Nachricht, die von der Anwendung bzw. vom Netzwerk kommt, beinhaltet grundsätzlich einen Datenpuffer, der die zu sendenden oder die empfangenen Daten enthält. Dieser Puffer kann auch aus mehreren Teilpuffern bestehen, was das Hinzufügen von Paketköpfen ermöglicht, ohne, dass die restlichen Daten kopiert werden müssen.

Kontrollereignisse und Zeitgeber können MP-spezifische Informationen enthalten, deren Format individuell vom MP festgelegt werden kann. MPs können eigene Ereignisse definieren, die sie auslösen. Andere MPs können sich auf diese Ereignisse nach dem Beobachter-Entwurfsmuster registrieren: Wird das Ereignis ausgelöst, wird dieses Ereignis an alle registrierten Beobachter versandt.

6.4.2 Message Scheduler

Ein Message Scheduler (MS) steuert die Ausführung einer Menge von Message Processors (MPs). Er verwaltet die Warteschlangen dieser MPs, ruft die Handler-Funktionen der MPs auf, wenn Ereignisse anstehen, und nimmt neue zu versendende Ereignisse entgegen, um sie in die Warteschlange des Ziel-MPs einzureihen. Die Abarbeitung dieser Ereignisse geschieht nach dem Round-Robin-Verfahren, wobei auch andere Verfahren realisiert werden können.

Ein MS ist dabei genau einem Betriebssystem-Thread zugeordnet. Der Ereignisaustausch zwischen MPs, die vom gleichen MS verwaltet werden, kann daher ohne aufwändige Thread-Synchronisation stattfinden. Um von mehreren Ausführungseinheiten des System-Prozessors zu profitieren, können mehrere Threads erzeugt werden, wobei jedem Thread ein eigener Message Scheduler zugeordnet wird. Der Austausch von Ereignissen zwischen MPs, die von unterschiedlichen MSs verwaltet werden, muss dann synchronisiert werden. Diese Synchronisation kann aber vom MS übernommen werden, sodass der Entwickler der Protokollkomponente hier nichts weiter beachten muss. Durch geschickte Zuordnung von MPs zu MSs kann so der Aufwand zur Synchronisation minimiert werden, während die Ausnutzung mehrerer Ausführungseinheiten maximiert werden kann. Automatisierte Verteilungsalgorithmen, die das Kommunikationsverhalten der MPs untereinander einbeziehen, sind hier denkbar, wurden im Rahmen dieser Arbeit aber nicht weiter untersucht.

6.5 Beispiel-Netzwerkarchitektur für NENA

Als Beispiel einer Netzwerkarchitektur in NENA wird in diesem Abschnitt die *Simple Architecture* (SimpA) beschrieben. Während sie vom Aufbau her

an die IP-Architektur angelehnt ist, wurde sie absichtlich einfach gehalten, um eine mögliche Aufteilung der verschiedenen Dienste und Mechanismen in NENA zu veranschaulichen.

Zur Kommunikation über ein reales Netzwerk werden die Netzwerkpakete, die von der SimpA versandt werden, über UDP getunnelt. Ein Network Adaptor (NA) der SimpA wird daher als UDP-Endpunkt realisiert. Die Adresse des UDP-Endpunkts nach dem Schema <IP-Adresse>:<UDP-Port> wird dabei als Schnittstellenadresse verwendet. Die SimpA differenziert jedoch nicht zwischen IP-Adresse und Portnummer, sondern nutzt das Tupel insgesamt als eine vollständige Adresse. Knoten, die Teil eines Netzes mit der SimpA sind, besitzen einen Knotennamen der Form `node://`<Knotenname>. Dieser Knotenname wird durch einen Namensauflösungsdienst auf eine oder mehrere Schnittstellenadressen abgebildet. Die Abbildung ist dabei statisch vorkonfiguriert.

Da die Nachbarschaftsfindung – also die Entdeckung der Knoten, die sich in der gleichen Broadcast-Domäne befinden – in der SimpA automatisch geschehen soll, wird für jeden Network Adaptor eine Broadcast-Adresse konfiguriert. Während in Broadcast-Netzen reale IP-Broadcast-Adressen verwendet werden können, kann es in einigen Netzen passieren, dass Nachrichten an die Broadcast-Adresse nicht zugestellt werden können, bspw. aufgrund von Filterregeln. Für diesen Fall kann eine Liste von Unicast-Adressen angegeben werden, an die die zu versendende Broadcast-Nachricht verschickt werden soll. Der Network Adaptor übernimmt transparent die Duplizierung der zu versendenden Broadcast-Nachricht. Durch die Konfiguration der Broadcast-Adresse mit Unicast-Adressen kann auch eine bestimmte Topologie für Testaufbauten erzwungen werden.

Abbildung 6.8 gibt einen Überblick über die Module der SimpA und deren Funktionen, auf die die nachfolgenden Abschnitte weiter eingehen.

6.5.1 Multiplexer

Der Multiplexer übernimmt für die SimpA folgende Funktionen:

- Auflösung von Anwendungsdienst auf Knotenname
- Auflösung von Knotenname auf Schnittstellenadresse
- Weiterleitung an den nächsten Knoten
- Realisierung des Request Mappers

Darüberliegende Protokolle, die in Netlets gekapselt sind, verwenden immer Namen statt Adressen. Die SimpA realisiert damit ein sog. Late-Binding, also

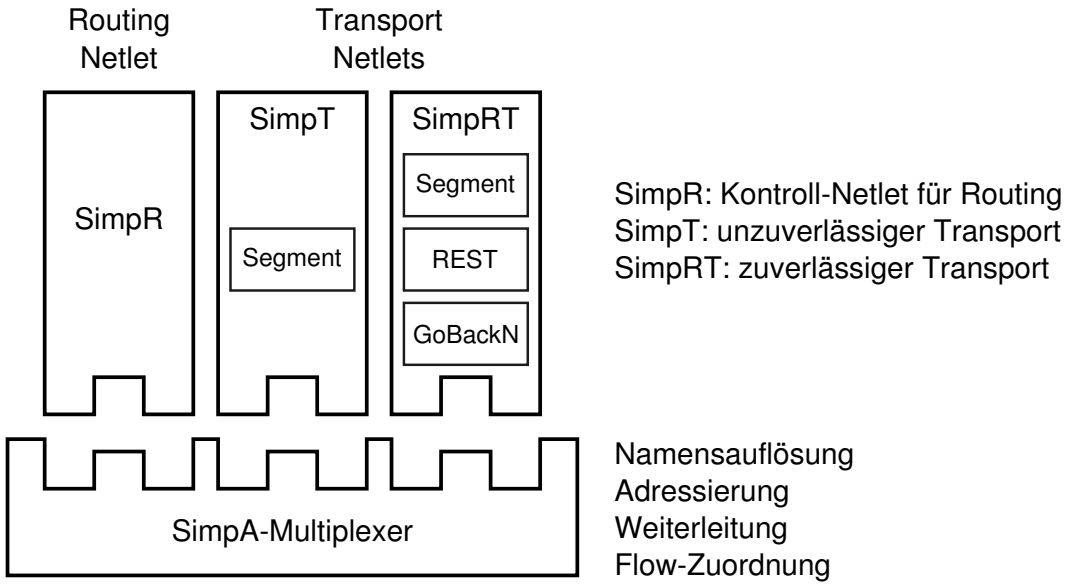


Abbildung 6.8 Die Netlets und der Multiplexer der Simple Architecture (SimpA)

eine späte Bindung von Namen an Adressen. Eine Ausnahme bildet hier das Routing-Protokoll, welches in einem späteren Abschnitt erläutert wird.

Anwendungen fordern entfernte Anwendungsdienste über NENA mit einer URI der Form

`app://`<Knotenname>`/`<Dienstname>`/`<Dienstparameter>

an. Durch eine kanonische Abbildung bestimmt der Multiplexer daraus die vollständige URI des Zielknotens, auf dem der Anwendungsdienst zu finden ist:

`node://`<Knotenname>

Die Auflösung von Knotenname nach Schnittstellenadresse ist ebenfalls im Multiplexer enthalten und wird durch den *Name-to-Address-Mapper* realisiert. Die Auflösung geschieht hier jedoch statisch nach einer vorkonfigurierten Liste.

Anhand der Schnittstellenadresse des Ziels wird nun in der *Forwarding Information Base* (FIB) der Network Adaptor und der nächste Knoten zum Ziel bestimmt und anschließend die Nachricht über den entsprechenden Network Adaptor versandt. Die FIB wird dabei von einem Routing-Protokoll befüllt (siehe Abschnitt 6.5.2.1). Ein Zwischenknoten der SimpA, der das Weiterleiten von Paketen übernimmt, ist über mindestens zwei Network Adaptors (also über zwei UDP-Endpunkte) angeschlossen. Wenn dieser Zwischenknoten ein Paket empfängt, welches nicht an ihn selbst gerichtet ist, verfährt er genauso wie beim Versand der eigenen Pakete: Anhand der Zieladresse wird

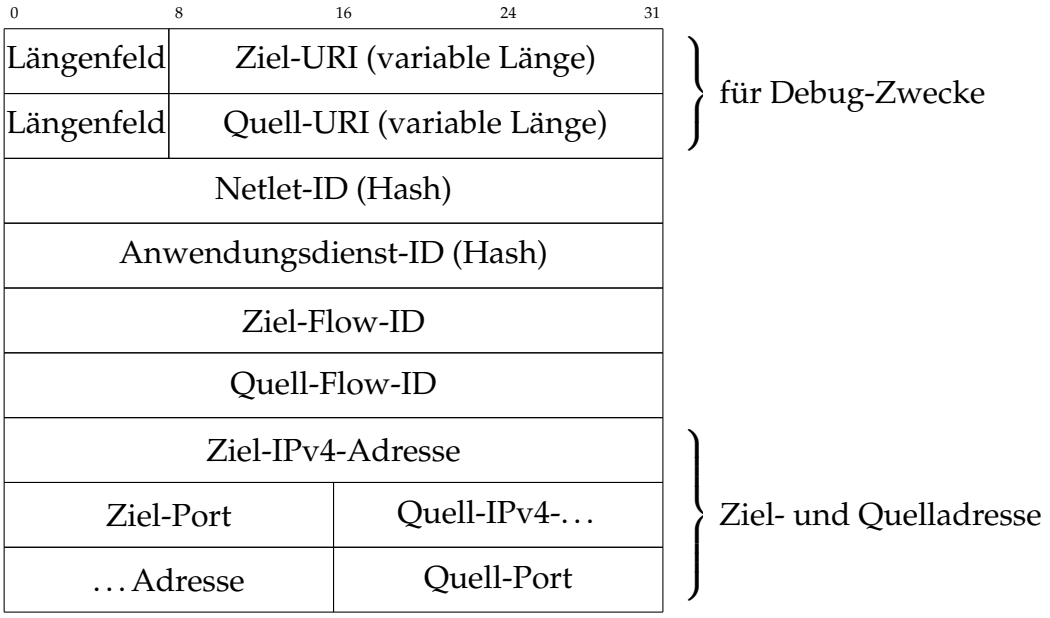


Abbildung 6.9 Der durch den Multiplexer hinzugefügte Paketkopf der SimpA.

die Schnittstelle und der nächste Knoten zum Ziel bestimmt und die Nachricht daraufhin weitergeleitet.

Zusätzlich zur Weiterleitung ein- und ausgehender Pakete erfüllt der Multiplexer also noch weitere Aufgaben durch folgende Teilkomponenten:

- der Name-to-Address Mapper bildet Knotennamen auf Schnittstellenadressen ab und
- die Forwarding Information Base dient zur Bestimmung des Ausgangs-Network-Adaptors und des nächsten Knotens.

Der Name-to-Address Mapper dient dabei gleichzeitig als Request Mapper (vgl. Abschnitt 6.2.2.1). Er leitet die Anfrage der Netlet Selection an die Factory-Komponenten (siehe Abschnitt 6.7.1) aller SimpA-Netlets weiter, die wiederrum individuell prüfen, ob das jeweilige Netlet den Kommunikationswunsch erfüllen kann. Die Factory-Komponente prüft dabei, ob der angeforderte Name der oben beschriebenen Form entspricht und ob die zusätzlichen Kommunikationsanforderungen den Eigenschaften des Netlets entsprechen. Ist dies der Fall, fügt der Name-to-Address-Mapper das Netlet in die Ergebnisliste für die Netlet Selection ein.

Zur Weiterleitung, zum Multiplexen und zum Auffinden der lokalen Anwendungsdienste (die über BIND einen Ereignis-Endpunkt erstellt haben) werden verschiedene Informationen dem Paketkopf hinzugefügt (Abbildung 6.9). Dazu zählen:

- Quell- und Ziel-Knoten-URI mit Längenfeld (nur für Debug-Zwecke)
- Netlet-ID (32 Bit ELF[1] Hash über die Netlet-URI)
- Anwendungsdienst-ID (32 Bit ELF Hash über die Dienst-URI)
- Quell- und Ziel-Flow-ID
- Quell- und Zieladresse (je 48 Bit bestehend aus IPv4-Adresse und Port)

Quell- und Zieladresse bezeichnen dabei die jeweiligen Schnittstellenadressen. Die Netlet-ID wird durch eine 32-Bit-Hashfunktion, die auf die URI des Netlets (vgl. Abschnitt 6.2.1) angewendet wird, erzeugt. Die Anwendungsdienst-ID wird auf gleiche Weise erzeugt, wobei hier lediglich die Basis des Anwendungsdienstnamens genutzt und der Dienstparameterteil ignoriert wird:

`app://`<Knotenname>`/`<Dienstname>

Die Quell-Flow-ID ist für den Empfänger wichtig, damit er Antwortpakete an die richtige Flow-ID des Absenders adressieren kann. In dieser Hinsicht erfüllt sie den gleichen Zweck wie bspw. die Portnummer des Absenders bei UDP. In ähnlicher Weise erlaubt die Ziel-Flow-ID dem Empfänger die Zuordnung der Nachrichten zu einem seiner Flows. Ist die Ziel-Flow-ID unbekannt (bspw. zu Beginn der Kommunikation) wird diese auf 0 gesetzt, und die Zuordnung zum Anwendungsdienst muss über die Anwendungsdienst-ID geschehen (siehe Abschnitt 6.5.3).

6.5.2 Netlets

Für die SimpA wurden mehrere Netlets entwickelt, die typische Aufgaben innerhalb einer Netzwerkarchitektur realisieren. Als ein Beispiel für ein Kontroll-Netlet dient ein Routing-Protokoll, welches für die SimpA implementiert wurde. Für den Datentransport werden zwei Netlets vorgestellt, die unterschiedliche Dienste anbieten, aber gemeinsame Protokollbausteine nutzen: je ein Netlet für zuverlässigen und unzuverlässigen Transport.

6.5.2.1 Routing

Die Aufgabe eines Routing-Netlets für die SimpA ist die Erkennung von benachbarten Knoten und die Bereitstellung von Informationen, die dazu dienen weiter entfernte Knoten über Zwischenknoten zu erreichen. Diese Informationen müssen in der Forwarding Information Base des Multiplexers hinterlegt werden, damit eine Weiterleitung von Netzwerkpaketen möglich ist.

[1] Executable and Linking Format, ein Binärformat für ausführbare Programme (bspw. unter Linux)

Ein einfaches Routing-Netlet wurde mit dem *Simple Routing* (SimpR) Netlet realisiert. Es ist ein Distanzvektorprotokoll: Benachbarten Knoten werden die bisher bekannten Ziele und deren Entfernung in Anzahl der Knoten (Hops) mitgeteilt. Empfangen Knoten solche Nachrichten, aktualisieren sie ihre Forwarding Information Base und fügen bisher unbekannte Knoten hinzu oder aktualisieren Einträge zu Knoten, wenn sich deren Entfernung im Vergleich zur bisherigen Entfernung über den sendenden Nachbarn verkürzt. Mit periodischen Aktualisierungen werden nach und nach Erreichbarkeitsinformationen im gesamten Netz verteilt. Das SimpR-Netlet erkennt allerdings keine Knotenausfälle und ist daher für dynamische Netze eher ungeeignet.

6.5.2.2 Transport und Inhalte

Für den Datentransport stehen in der SimpA die beiden Transport-Netlets *Simple Transport* (SimpT) Netlet und *Simple Reliable Transport* (SimpRT) Netlet zur Verfügung. Beide nutzen dabei Bausteine zur Realisierung der verschiedenen Transportdienste.

Das SimpT-Netlet stellt einen unzuverlässigen Transportdienst bereit. Da die SimpA einen solchen Dienst bereits aufgrund der Tatsache, dass sie auf UDP aufsetzt, anbietet, wird lediglich ein Baustein zur Segmentierung von Anwendungsdaten benötigt. Dieser sorgt dafür, dass die Nachrichten, die von der Anwendung Richtung Netzwerk geschickt werden, eine konfigurierte Maximalgröße nicht überschreiten. Diese Maximalgröße ist so eingestellt, dass keine IP-Fragmentierung im darunterliegenden Netz notwendig wird.

Das SimpRT-Netlet besteht dagegen aus drei Bausteinen: aus dem Segmentierbaustein, der auch bereits im SimpT-Netlet verwendet wird; aus einem Baustein, der REST-Befehle signalisiert; und aus einem Baustein, der ein Go-Back-N ARQ Verfahren realisiert.

Die Aufgabe des Bausteins zur Signalisierung von REST-Befehlen besteht darin, die Informationen, die von der Anwendung durch die Anwendungsschnittstelle übergeben werden, zum Ziel zu signalisieren (siehe Kapitel 5). Zu diesen Informationen gehören unter anderem die Primitive (PUT, GET oder CONNECT) und die vollständige URI des Ziels, die neben dem Knotennamen und Dienstnamen auch Dienstparameter enthalten kann.

Der Go-Back-N-Baustein realisiert das gleichnamige Verfahren zur automatischen Sendewiederholung. Der Sendewiederholungspuffer wird dabei auf einen festen Wert vorkonfiguriert. Zudem wird das verzögerte Senden von akkumulierten Quittungen unterstützt, deren Verzögerung ebenfalls auf einen festen Wert vorkonfiguriert ist. Durch diese beiden Parameter, sowie die maximale Größe einer Dateneinheit, die durch den Segmentierbaustein forciert wird, kann so der maximal mögliche Durchsatz festgelegt werden.

6.5.3 Flow-State-Nutzung

In diesem Abschnitt wird die Verwendung der Flow-State-Objekte (FSOs) am Beispiel der SimpA erläutert. Dabei wird zunächst auf den Zeitpunkt der Erstellung dieser Objekte eingegangen, sowie auf deren Lebensdauer. Anschließend wird am Beispiel des Go-Back-N-Bausteins gezeigt, wie einzelne Bausteine ihre Zustandsinformationen mit Hilfe des FSOs verwalten.

6.5.3.1 Erstellung der Flow-State-Objekte

Während die Erzeugung von FSOs mit der Erstellung des Flow Connectors für Ressourcen- und Ereignis-Endpunkte durch NENA geschieht, muss die Zuordnung und die Entscheidung zur Erzeugung eines neuen FSOs für hereinkommende Anfragen durch die Netzwerkarchitektur getroffen werden. Dies ist damit begründet, dass NENA keinerlei Wissen über den Inhalt der Daten hat, die über die verschiedenen Network Adaptors empfangen werden. Die Auswertung der Daten wird erst vom Multiplexer der jeweiligen Netzwerkarchitektur vorgenommen.

In der SimpA extrahiert der Multiplexer dazu die notwendigen Daten aus dem Paketkopf (vgl. Abschnitt 6.5.1). Ist die Ziel-Flow-ID angegeben, wird das passende FSO herausgesucht und als Attribut an die Nachricht angehängt. Anschließend wird die Nachricht zur weiteren Verarbeitung an das entsprechende Netlet weitergereicht.

Bei unbekannter Ziel-Flow-ID wird zunächst die Anwendungsdienst-ID genutzt, um das zugehörige FSO des Ereignis-Endpunkts der Anwendung zu bestimmen. Dieses FSO stellt die Beziehung zur Anwendung über den Flow Connector her, über den die Anwendung die Bereitschaft signalisiert hereinkommende Anfragen zu akzeptieren. Um mehrere, parallele Anfragen von verschiedenen Quellknoten mit eigenem Kommunikationskontext zu ermöglichen, muss für jede hereinkommende Anfrage ein separates FSO erzeugt werden. Aus diesem Grund wird ein Kind-FSO zum FSO der Anwendungsverbindung erzeugt. Das so erzeugte Kind-FSO wird dem Eltern-FSO und der aktuellen Nachricht als Attribut angehängt. Die Nachricht wird anschließend zur weiteren Verarbeitung an das entsprechende Netlet weitergereicht. Sobald sie im Netlet Selector angekommen ist, wird die Anwendung über den Ereignis-Endpunkt über die neue Anfrage informiert. Die Anwendung muss daraufhin einen Ressourcen-Endpunkt mit ACCEPT erstellen, dem dann das Kind-FSO zugeordnet wird. Um den weiteren Nachrichtenaustausch zu vereinfachen, werden anschließend die entsprechenden Flow-IDs in den Netzwerkpaketen mitgeschickt.

6.5.3.2 Zustandsobjekt für Go-Back-N

Nachrichten von Anwendungen oder vom Netzwerk enthalten aufgrund der im vorherigen Abschnitt beschriebenen Mechanismen grundsätzlich einen

Zustandsvariable	Beschreibung
localSeqNo	nächste zu vergebende Sequenznummer
localConnState	unidirektionaler Verbindungszustand
sendBuffer	Sendepuffer
retrBuffer	Sendewiederholungspuffer
retrTimer	Sendewiederholungszeitgeber
remoteSeqNo	nächste erwartete Sequenznummer
remoteConnState	unidirektionaler Verbindungszustand
ackTimer	Quittungszeitgeber

Tabelle 6.1 Zustandsobjekt (SO) des Go-Back-N-Bausteins

Verweis auf ein FSO, welches den Verbindungskontext eindeutig identifiziert. Protokollkomponenten können an dieses FSO eigene Zustandsobjekte (State Objects, SOs) anhängen. Die Identifikation des SO geschieht dabei über ein eindeutiges Schlüsselwort, welches im Allgemeinen die URI der Protokollkomponente ergänzt um eine Instanz-ID ist.

Während zustandslose Mechanismen ein solches SO nicht benötigen, ist dies für zustandsbehaftete Mechanismen essentiell. Der einzige Transportbaustein der SimpA, der eigene verbindungsspezifische Zustände hält, ist der Go-Back-N-Baustein. Die für ihn notwendigen Zustandsvariablen sind in Tabelle 6.1 aufgeführt. Die Sequenznummer des Go-Back-N-Bausteins beschreibt Paketsequenznummern (localSeqNo, remoteSeqNo), deren initialer Wert am Anfang einer Verbindung gewürfelt wird. Durch eine Synchronisationsnachricht (syn – synchronize) während des verwendeten 2-Wege-Handshakes wird dem entfernten Kommunikationspartner der initiale Wert mitgeteilt (vgl. Abb. 6.10). Dieser antwortet mit einer Quitting (ack – acknowledgment), die die Sequenznummer des nächsten zu erwartenden Pakets enthält. Die Verbindung wird dabei zunächst lediglich in eine Richtung initialisiert. Der Verbindungsaufbau in die Gegenrichtung erfolgt nur dann, wenn auch Daten zurückgesendet werden. Dazu wird dann analog ein 2-Wege-Handshake durchgeführt. Für eine erfolgreiche Kommunikation über den Go-Back-N-Baustein ist es notwendig, dass die Verbindung wieder ordnungsgemäß durch eine Finalisierungsnachricht (fin – finalize) abgebaut wird. Werden unerwartete Nachrichtentypen empfangen, wird eine Nachricht zum Zurücksetzen der Verbindung (rst – reset) versendet. Die Verbindung befindet sich dann in einem Fehlerzustand und das FSO wird auf den Zustand *stale* gesetzt. Insgesamt kennt jede Halbverbindung des Go-Back-N-Bausteins folgende Zustände (localConnState, remoteConnState):

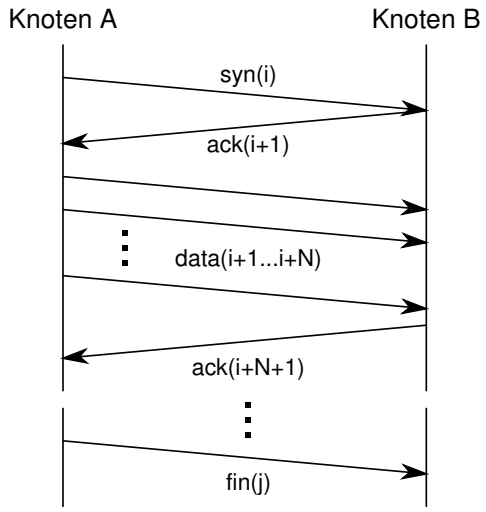


Abbildung 6.10 Verbindungsaufbau, Kommunikation und Verbindungsabbau über den Go-Back-N-Baustein

- *none* – Verbindung ist nicht initialisiert,
- *syn* – Verbindung ist in der Synchronisationsphase,
- *rdy* – Verbindung ist bereit und kann genutzt werden,
- *fin* – Verbindung wurde ordnungsgemäß beendet,
- *error* – Verbindung befindet sich in einem Fehlerzustand (bspw. nach wiederholt erfolglosen Zustellversuchen oder nach einem Verbindungsabbruch durch den entfernten Kommunikationspartner mittels einer rst-Nachricht).

Neben diesen Zustandsvariablen wird auch der Sendewiederholungspuffer (retrBuffer) an das FSO angehängt. In diesem befinden sich Nachrichten, die bereits versendet, aber vom entfernten Kommunikationspartner noch nicht bestätigt wurden. Zusätzlich ist noch ein Sendepuffer (sendBuffer) notwendig, in dem Anwendungsnachrichten während der Synchronisationsphase zwischengespeichert werden. Referenzen auf den Sendewiederholungszeitgeber (retrTimer) und dem Quittungszeitgeber (ackTimer) werden ebenfalls im SO hinterlegt, um diese bei Bedarf abbrechen oder neustarten zu können.

Der vollständige verbindungsspezifische Zustand des Go-Back-N-Bausteins wird also im FSO verwaltet. Mit dem Ende der Lebenszeit des FSO werden auch alle angehängten SOs freigegeben.

6.6 Realisierungsalternativen

Traditionell sind die Protokolle der Schicht 4 und abwärts im Kernel des Betriebssystems (Transport- und Vermittlungsprotokolle) oder in Hardware rea-

lisiert (Sicherungsschicht; teilweise auch Prüfsummenberechnung für Transportprotokolle). Die Gründe dafür liegen in der schnelleren und effizienteren Bearbeitung von Netzwerkpaketen und dem damit möglichen Gewinn im Durchsatz. Gerade bei leistungsschwachen Rechnersystemen sind solche Optimierungen essentiell. Jedoch zeigen Implementierungen, dass mit heutigen Systemen auch hohe Leistungen im User Space des Betriebssystems möglich sind: Gerade UDT (Abschnitt 3.2.1) konnte dies mit wiederholten Erstplatzierungen in Hochleistungs-Benchmarks für Transportprotokolle zeigen (z. B. [Szal+08]). Zudem verfolgen Mikro-Kernel-Architekturen generell den Ansatz, möglichst viel Funktionalität im User Space des Rechners zu realisieren, also gerade auch Kommunikationsprotokolle.

Während das NENA-Rahmenwerk, wie im nächsten Abschnitt gezeigt wird, ausschließlich im User Space ähnlich einer Middleware-Software realisiert ist, schränkt das eine betriebssystemnahe oder gar Hardware-gestützte Umsetzung nicht ein, da die Verwendung durch Anwendungen über eine entsprechende API-Bibliothek gekapselt ist. Damit sind verschiedene Realisierungen des Rahmenwerks denkbar:

1. Traditionell im Kernel

   Das Rahmenwerk wird mit all seinen Komponenten im Betriebssystemkernel realisiert und die API-Aufrufe werden in entsprechende Kernel-Aufrufe umgesetzt. Viele Kernel sind heute schon modular aufgebaut, sodass Netlets und Multiplexer auch zur Laufzeit als Kernel-Module (Linux) oder Treiber-Software (Windows) nachgeladen werden können. Die möglichen Nachteile sind dabei Systeminstabilitäten, die durch Fremdsoftware im Kernel hervorgerufen werden können (wie es bei heutigen Gerätetreibern auch schon der Fall sein kann).

2. als Hintergrundanwendung (Dämon) bzw. Middleware-Software

   Wird das Rahmenwerk als Prozess im User Space realisiert, können viele Funktionen des Betriebssystems zur Prozess-Isolation verwendet werden. Das Nachladen von Modulen ist als Bibliothek möglich, und die Module können in eigenen Kind-Prozessen ausgeführt werden, was deren Isolation untereinander und vom Rest des Betriebsystems ermöglicht. Nachteilig kann sich hier die Performance auswirken, da bspw. auch bei reiner Paketweiterleitung mehrere, teure Übergänge zwischen Kernel und User Space notwendig sind.

3. Hybride Umsetzung

   Teile des Rahmenwerks können sich je nach Performance-Anforderungen entweder im Kernel oder im User Space des Betriebssystems wiederfinden. Zudem können bei einer evolutionären Einführung der Konzepte dieser Arbeit viele Funktionen im User Space realisiert werden,

während die traditionellen Protokollimplementierungen des Betriebsystems weiterhin genutzt werden.

Orthogonal zu diesen drei Möglichkeiten ist nach wie vor die Verwendung von Hardware-Beschleunigern in allen Fällen möglich: Einzelne Bausteine des Protokolls können bei vorhandenen Beschleunigern ausgetauscht werden. Diese kapseln dann den Aufruf der Beschleuniger-Hardware.

Da die Implementierung des NENA-Rahmenwerks als Machbarkeitsnachweis dienen soll und daher prototypischen Charakter hat, wurde der Einfachheit halber die zweite Option ohne Hardware-Unterstützung gewählt. Eine ganze oder teilweise Umsetzung im Betriebsystemkernel wird jedoch nicht ausgeschlossen.

6.7 Prototyp

In diesem Abschnitt wird ein Überblick über die prototypische Implementierung des NENA-Rahmenwerks als Hintergrundanwendung (Dämon) im User Space gegeben. Als Orientierung wird dazu Abbildung 6.11 verwendet, die neben der Anordnung der Komponenten auch Verweise auf die Verzeichnisstruktur des in C++ geschriebenen Quellcodes enthält (vgl. dazu auch die Implementierungsübersicht der Anwendungsschnittstelle in Abbildung 5.6 auf Seite 96). Dieser Prototyp bildet die Basis für weitere Implementierungen, die im Rahmen dieser Arbeit durchgeführt wurden und ist als Open-Source-Software öffentlich verfügbar[1].

Um Abhängigkeiten gering zu halten, wurden hauptsächlich Funktionen aus der *Standard Template Library* (STL) von C++ genutzt, ergänzt um Bibliotheken aus dem *Boost* Projekt. Die Boost-Bibliotheken dienen als Vorschläge für zukünftige C++-Erweiterungen und werden teilweise in neue Standards mit aufgenommen. Obwohl in dieser Arbeit ausschließlich Linux verwendet wurde, ist es dadurch mit geringem Aufwand möglich, den Prototyp auch auf andere Betriebssystemplattformen zu portieren.

Der NENA-Prototyp stellt anderen Anwendungen seine Dienste über Interprozesskommunikation bereit. In den folgenden Unterabschnitten werden die Implementierungen der verschiedenen Aspekte logisch gruppiert skizziert.

6.7.1 Message Processors und Module

Alle in Abbildung 6.11 innerhalb von NENA dargestellten Komponenten implementieren das `IMessageProcessor`-Interface und können somit Nachrichten, Kontrollereignisse und Zeitgeber empfangen und versenden (vgl.

[1] http://nena.intend-net.org/

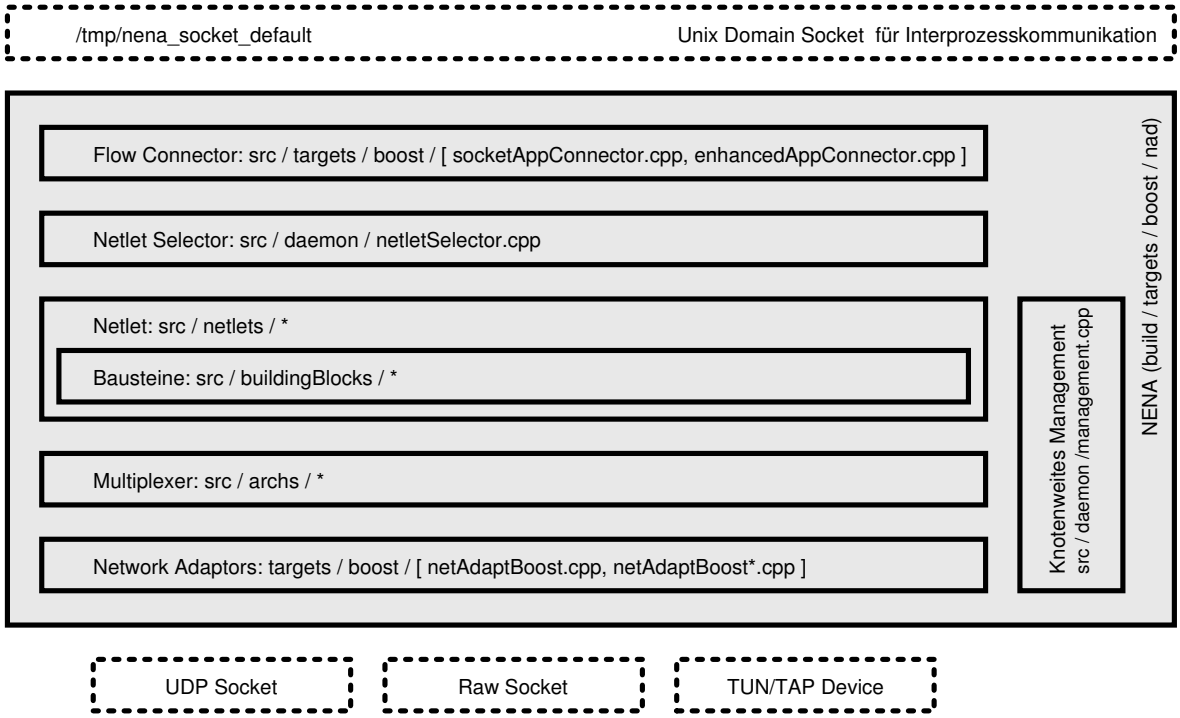


Abbildung 6.11 Überblick über die Struktur des NENA-Prototyps.

hierzu Abb. 6.7 auf Seite 111). Mit Ausnahme von Multiplexer und Netlets sind alle Komponenten Teil des NENA-Dämons, der als eigenständige Hintergrundanwendung ausgeführt wird.

Multiplexer und Netlets werden als externe Bibliotheken für NENA kompiliert und können zur Laufzeit des Dämons durch die Repository-Klasse nachgeladen werden. Beim Öffnen solcher Bibliotheken wird automatisch eine statische Klasse erzeugt, die sich als sogenannte Factory beim NENA-Dämon registriert. Eine Factory ist ein Software-Entwurfsmuster, bei dem die Instantiierung von Objekten nicht direkt durch den Aufrufer geschieht, sondern durch eine Factory-Methode gekapselt ist. Dadurch können beliebige Sub-Typen der Klassen erstellt werden, ohne dass der Aufrufende den genauen Typ kennen muss.

Bei der SimpA wird hier eine weitere Indirektion vorgenommen: Statt der Factory für den eigentlichen Multiplexer bzw. für das eigentliche Netlet, wird eine Factory-Klasse für die Factory-Klasse statisch instantiiert. Dies erlaubt die Instantiierung verschiedener SimpAs, die sich lediglich in ihrer eindeutigen Bezeichnung unterscheiden. Diese Funktion wurde realisiert, um zu Testzwecken beliebig viele Netzwerkarchitekturen für NENA instantiieren zu können.

Multiplexer und Netlets implementieren die entsprechenden Basisklassen `IMultiplexer` bzw. `INetlet`. Beide erben dabei das `IMessageProcessor`-Interface. Zur Realisierung von Protokollkompositionen können inner-

halb von Netlets und Multiplexern Klassen verwendet werden, die von `IBuildingBlock` erben. `IBuildingBlock` ist dabei ebenfalls eine Spezialisierung des `IMessageProcessor`-Interface.

6.7.2 Network Adaptors

Als Network Adaptor (NA) existieren drei Alternativen: ein UDP-Socket, ein Raw-Socket und ein Schicht-2-Tunnel-Gerät (TUN/TAP-Gerät). Diese unterscheiden sich wie folgt:

- UDP-Socket

  Die netzwerkarchitekturspezifischen Daten werden über UDP zwischen zwei NENA-Knoten getunnelt. Dies ist die am einfachsten zu realisierende Variante, da hierzu keine speziellen Berechtigungen vom Betriebssystem benötigt werden. Außerdem ist es sehr einfach möglich mehrere virtuelle NAs zu erstellen, die jeweils mit einem anderen UDP-Endpunkt terminiert sind.

- Raw-Socket

  Mit einem Raw-Socket wird die Schicht-4-Implementierung des Betriebssystems umgangen, und der NA bekommt direkten Zugriff auf die IP-Pakete einer vorgegebenen Netzwerkschnittstelle (z. B. eth0). Da dadurch alle IP-Pakete, die über die Netzwerkschnittstelle empfangen werden, an den NA weitergereicht werden, sind spezielle Berechtigungen, die dem Dämon erteilt werden müssen, Voraussetzung. Mehrere virtuelle NAs benötigen zudem eine virtuelle Netzwerkschnittstelle, die vom Betriebssystem erstellt werden muss.

- TUN/TAP-Geräte

  Mit TUN/TAP-Geräten wird eine virtuelle Netzwerkschnittstelle auf dem Betriebssystem erstellt, über die der NA Schicht-2-Pakete empfangen und versenden kann. Der dazugehörige Rahmen des MAC-Protokolls kann dabei vollständig gelesen werden, muss aber auch beim Versenden zur Verfügung gestellt werden. Durch eine korrekte Vergabe von Zugriffsberechtigungen auf das TUN/TAP-Gerät benötigt der NA keine speziellen Berechtigungen zur Laufzeit.

Von diesen drei Technologien sind TUN/TAP-Geräte die flexibelsten und werden von vielen Netzwerkvirtualisierungs- und VPN-Technologien genutzt. Am einfachsten und portabelsten sind allerdings NAs, die über UDP-Sockets realisiert sind, da diese auch über existierende Netze hinweg meist problemlos funktionieren.

6.7.3 Interprozesskommunikation

Zur Interprozesskommunikation mit Anwendungen wird ein Unix-Domain-Socket verwendet. Dieser wird über einen Dateinamen adressiert und stellt eine verbindungsorientierte Kommunikation bereit: NENA öffnet dabei einen Server-Socket, mit dem sich andere Anwendungen verbinden können. Jede neue Verbindung muss von NENA akzeptiert werden, was zur Erstellung eines neuen Flow Connectors führt. Die bereitgestellte Kommunikation ist Byte-Strom-orientiert, sodass ein minimalistisches Protokoll zur Formatierung des Byte-Stroms notwendig ist. Dieses Protokoll nutzt dazu eine Typ-Länge-Wert-Kodierung (Type-Length-Value, TLV) mit verschiedenen Nachrichten-Typen, mit denen strukturiert die notwendigen Informationen über den Kommunikationswunsch von der Anwendung zu NENA übergeben werden. Zu diesen Informationen gehören die gewünschte Primitive, die URI des Ziels und die dazugehörigen Anwendungsanforderungen. Zur Vereinfachung für die Anwendung wird dieses Protokoll durch eine API-Bibliothek implementiert (vgl. Abschnitt 5.5).

6.7.4 Kapselung des Basissystems

Die Beschreibungen der Network Adaptors und der Interprozesskommunikation in den letzten beiden Unterabschnitten beziehen sich auf den Betrieb des NENA-Rahmenwerks auf Linux-Systemen. Betriebssystemspezifische Teile sind dabei jedoch so gekapselt, dass sie einfach ausgetauscht werden können. Damit ist es auch möglich, das NENA-Rahmenwerk innerhalb des diskreten Ereignissimulators OMNeT++ [Varg01] zu instantiieren. Der C++-Kern des NENA-Dämons sowie alle Netlets und Multiplexer können dabei unverändert genutzt werden. Durch die Bereitstellung eines anderen Flow Connectors, der an Stelle der Interprozesskommunikation eine simulierte Anwendung realisiert, können beliebige Traffic-Generatoren verwendet werden. Ein angepasster Network Adaptor kapselt zudem die Netzwerkkommunikation und versendet NENA-Nachrichten als OMNeT-Nachrichten an andere simulierte NENA-Instanzen. Jede NENA-Instanz ist dabei als `cSimpleModule` in OMNeT realisiert. Optional unterstützt wird dabei auch das INET-Framework, sodass ein IP-Netz als darunterliegenden Infrastruktur genutzt werden kann.

Die Möglichkeit, die implementierten Protokolle sowohl auf realen Systemen als auch im Simulator ausführen zu können, ist eine wichtige Eigenschaft für den in Kapitel 4 beschriebenen Entwurfsprozess. So können neue Protokolle zunächst in simulierten Umgebungen getestet und evaluiert werden, bevor dieselbe Implementierung für Produktivsysteme eingesetzt wird. Im Simulator getestet wurden so das in Abschnitt 4.3.2 beschriebene Transportprotokoll (vgl. Abschnitt 7.5.1) sowie einige der in Abschnitt 7.3 beschriebenen Fallstudien.

7. Evaluation

Die Konzepte und Schnittstellen, die in dieser Arbeit entwickelt wurden, erleichtern langfristig die Einführung und nebenläufige Verwendung neuer dienstanbieterspezifischer Netze und Protokolle und verlangen dabei kein protokoll- oder netzwerkspezifisches Wissen mehr vom Anwendungsentwickler, dessen Anwendungen über diese Netze kommunizieren sollen. Die dazu beigetragene Entkopplung von Anwendungen und Protokollen wird in diesem Kapitel anhand von Fallstudien untersucht [MaWi13b, MaWi13c]. Nach Beschreibung der verwendeten Methodik und der Kriterien in Abschnitt 7.1, wird zunächst die Nutzung der neuen Anwendungsschnittstelle durch heute verbreitete Anwendungen untersucht (Abschnitt 7.2). Anschließend wird in Abschnitt 7.3 die Integration verschiedener Netzwerkarchitekturen in das NENA-Rahmenwerk und die Abbildung der Anwendungsschnittstelle auf die jeweilige Protokollfunktionalität beschrieben und bewertet. In der abschließenden Gesamtbewertung der Konzepte in Abschnitt 7.4 wird gezeigt, dass für die untersuchten Fallstudien das NENA-Rahmenwerk eine geeignete Umgebung bietet, um verschiedene Protokolle und Architekturen so zu betreiben, dass Anwendungen mit minimalem Wissen darauf zugreifen können. Eine Untersuchung des Laufzeitverhaltens wird schließlich in Abschnitt 7.5 anhand der in NENA implementierten Transportprotokollschablone aus Abschnitt 4.3.2 durchgeführt, um daraus Optimierungsmöglichkeiten abzuleiten. Hierauf folgen generelle Anmerkungen zur Skalierbarkeit der vorgestellten Konzepte.

7.1 Methodik und Kriterien

Für die Evaluation der Anwendungsschnittstelle und des NENA-Rahmenwerks wurden verschiedene Fallstudien durchgeführt und anhand der in diesem Abschnitt vorgestellten Kriterien untersucht. Die Anwendungsschnittstelle wird dabei zunächst nur aus Anwendungssicht betrachtet, unabhängig von darunterliegenden Protokollen. Anschließend wird aus Protokollsicht untersucht, wie Protokolle Anfragen über die Anwendungsschnittstelle umsetzen. Die Kriterien für beiden Sichtweisen sind in Abschnitt 7.1.1 beschrieben. Neben der Umsetzung der Anwendungsschnittstelle durch verschiedene Protokolle wird auch die Integration dieser Protokolle in das NENA-Rahmenwerk bewertet. Die dazu aufgestellten Kriterien werden in Abschnitt 7.1.2 beschrieben.

7.1.1 Anwendungsschnittstelle

Die Anwendungsschnittstelle, wie sie in Kapitel 5 beschrieben wurde, ist der für die Anwendung relevante Beitrag zur Entkopplung von Anwendungen und Protokollen. Sie fördert zum einen, dass viele Anwendungen kein eigenes Anwendungsprotokoll mehr implementieren müssen, zum anderen, dass Eigenschaften der darunterliegenden Netzwerkarchitektur wie z. B. die Adressierung transparent gehalten werden. Zur Evaluation, ob eine solche Entkopplung aus Anwendungssicht erfolgreich ist, soll untersucht werden, wie verschiedene Anwendungen diese Schnittstelle nutzen und welche Annahmen sie treffen bzw. formulieren müssen. Außerdem wird damit festgestellt, ob die untersuchten Anwendungen zusätzliche Informationen vom darunterliegenden Protokoll benötigen. Aus der Nutzung, den Annahmen, die die Anwendungen treffen, und aus den evtl. zusätzlich benötigten Protokollinformationen ergeben sich die Bewertungskriterien:

- E_{A1} – Art und Weise der Nutzung
- E_{A2} – Annahmen über Eigenschaften des Kommunikationsdienstes
- E_{A3} – benötigte Zusatzinformationen über das Protokoll

Aus Protokollsicht ist die Anwendungsschnittstelle aus dem Grund relevant, da darüber die notwendigen Informationen von der Anwendung bezogen werden, um entscheiden zu können, mit welchen Protokollen und Parametern die Kommunikation durchgeführt werden soll. Wie die Schnittstelle zur Nutzung der jeweiligen Protokolle aufgerufen werden muss, welche Annahmen das Protokoll über die Anwendungen trifft und ob die erhaltenen Informationen ausreichend sind, beschreiben die Bewertungskriterien:

- E_{P1} – Art und Weise der Nutzung

- E_{P2} – Annahmen über Anwendung
- E_{P3} – benötigte Zusatzinformationen über die Anwendung

7.1.2 NENA-Rahmenwerk

Das NENA-Rahmenwerk stellt eine *Architektur* zur Ausführung von Protokollsoftware dar. Um Architekturen zu bewerten, wird in [ABEH+04] die Analyse von Invarianten der Architektur vorgeschlagen. Invarianten bezeichnen die *Stützpunkte* einer Architektur und sind unveränderlich solange die Architektur selbst nicht verändert wird. Sie geben damit den Rahmen vor, an dem sich Realisierungen innerhalb der Architektur orientieren müssen. Unterschieden wird hierbei zwischen *expliziten* Invarianten – also Invarianten, die beim Entwurf der Architektur vorgesehen waren – und *impliziten* Invarianten. Letztere wurden beim Entwurf der Architektur nicht direkt vorgesehen, haben sich aber aus anderen Vorgaben ergeben. Implizite Invarianten stellen insofern eine Gefahr für den Erfolg der Architektur dar, als dass sie die Realisierungen innerhalb der Architektur unvorhergesehen einschränken können.

Für NENA bedeutet dies, dass sich durch die Architektur des Rahmenwerks ggfs. implizite Invarianten ergeben haben, die zukünftige Protokollsoftware oder gar komplette Netzwerkarchitekturen einschränken. Zur Bewertung der Integration der Protokollsoftware in NENA werden daher folgende Kriterien festgelegt:

- I_1 – Vollständigkeit (*Was fehlt?*)

  Dieses Kriterium beschreibt, ob die Konzepte, die NENA bietet, ausreichend sind, um die jeweilige Netzwerkarchitektur zu betreiben.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Dieses Kriterium gibt an, ob alle Konzepte nutzbringend sind (d. h., dass keine Konzepte überflüssig sind).

- I_3 – Limitationen (*Was stört?*)

  Limitation bzw. Einschränkungen bei der Umsetzung von Protokollsoftware für NENA geben Hinweise auf ungünstig gewählte Konzepte oder nicht vorgesehen Einschränkungen (implizite Invarianten) durch NENA.

Ob das NENA-Rahmenwerk für alle zukünftigen Netzwerkarchitekturen geeignet ist, kann durch diese Kriterien nicht vorhergesagt werden. Durch die Realisierung der Protokollsoftware für möglichst unterschiedliche Netzwerkarchitekturen in NENA, die zum Teil unterschiedliche Kommunikationsparadigmen verwenden, kann jedoch ein großer Bereich aktueller und aufkommender Ansätze abgedeckt werden.

7.2 Anwendungen

In diesem Abschnitt wird die Nutzung der Anwendungsschnittstelle (API) durch verschiedene Anwendungen mit Hilfe der in Abschnitt 7.1.1 eingeführten Kriterien beschrieben und beurteilt. Die Anwendungen und deren Besonderheiten sind im Überblick:

- *Web-Browser.* Dies ist eine der meistgenutzten Anwendungen heute. Durch die Integration verschiedener Medien und durch stetige Erweiterungen ist sie heute bereits auch eine Plattform, auf der verschiedene Internet-basierte Anwendungen aufsetzen. Damit wird der Web-Browser auch in Zukunft eine wichtige Rolle spielen.

- *Chat bzw. Instant Messaging.* Der Austausch von Kurznachrichten ist spätestens seit Aufkommen der SMS ein beliebtes Kommunikationsmittel, welches durch moderne Alternativen wie WhatsApp weiter genutzt wird. Variationen wie Twitter bieten hierbei auch Alternativen, die die sozialen Aspekte der Kommunikation hervorheben. Hier ist bspw. die Anzahl der Teilnehmer an einer Kommunikation variabel bzw. den Teilnehmern im Vorfeld nicht unbedingt bekannt.

- *Video-Streaming* bietet aufgrund von Kodierungsverfahren, die auf die Übertragung in unterschiedlichen Netzen angepasst sind, eine Anwendung, die von einem engen Zusammenspiel mit den Kommunikationsprotokollen profitiert. Da eine Anwendungsschnittstelle durch ungünstige Abstraktionen dieses Zusammenspiel erschweren kann, werden hierzu verschiedene Video-Streaming-Alternativen untersucht.

- *Datei-Server.* Als Beispiel einer Anwendung, die einen Dienst im Netz bereitstellt, wird ein Datei-Server verwendet. Hier wird untersucht, ob die Server-Anwendung alle notwendigen Informationen von ihren Clients erhalten kann, ohne selbst ein eigenes Anwendungsprotokoll realisieren zu müssen.

- *Gruppenkommunikationsanwendungen* stellen eine Besonderheit insofern dar, als dass Verwaltungsaufgaben von der Anwendung gesteuert werden können. Dazu gehören die Gruppenerstellung, der Gruppenbeitritt und der Gruppenaustritt.

7.2.1 Web-Browser

Der Web-Browser ist eine weit verbreitete Anwendung zur Darstellung von Multimedia-Inhalten, die über HTTP angefordert und übertragen werden. Durch das Experiment eines Online-Video-Dienstes, das in Abschnitt 5.4.3 beschrieben wurde, konnte gezeigt werden, dass einfache HTTP-Anfragen

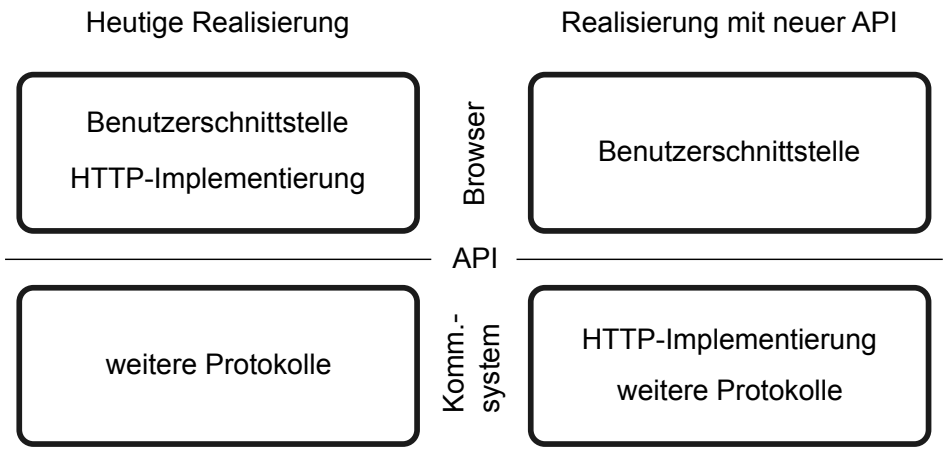


Abbildung 7.1 Schematische Darstellung der Realisierung eines Web-Browsers heute (links) und mit neuer API (rechts).

auf Anfragen an die Anwendungsschnittstelle (im Folgenden API-Anfragen) abgebildet werden können. Die Realisierung eines Adapters zwischen HTTP-Anfragen und API-Anfragen (siehe Abschnitt 5.5) zeigt die generelle Umsetzbarkeit.

Nutzung der Anwendungsschnittstelle

Mit der neuen Anwendungsschnittstelle muss das Anwendungsprotokoll HTTP nicht länger von der Anwendung implementiert werden (Abbildung 7.1). Das bedeutet allerdings auch, dass das verwendete Namensschema in der URI angepasst werden muss: Während mit http:// ein Ressourcen-Endpunkt angefordert wird, über den der Anwendungsdienst HTTP angesprochen wird, sollen nun die Inhalte (HTML-Dokumente, Bilder etc.) direkt mit der GET-Primitive der API angefordert werden. Aus diesem Grund wird ein neues Namensschema (nenaweb://) gewählt. Durch das Namensschema werden außerdem die Anforderungen festgelegt: Die Daten sollen in der Form empfangen werden, wie sie ursprünglich vorliegen. Außerdem müssen die empfangenen Ressourcen mit Informationen annotiert werden, um z. B. den Typ, die Zeichenkodierung oder den Zeitstempel des Inhalts zu beschreiben. Diese Informationen können der Anwendung über die Metadaten bereitgestellt werden.

Besonderheiten

HTTP-POST und HTTP-PUT werden auf die PUT-Primitive abgebildet. HTTP-POST beschreibt dabei das Hinzufügen oder Senden einer Information an eine Ressource, während HTTP-PUT die Ressource selbst modifiziert (also z. B. überschreibt oder neu anlegt). Diese Unterscheidung ergibt sich in der Praxis oft implizit auf Basis der angegebenen Ressource: Handelt es

sich bei der Ressource um ein Skript, werden die gesendeten Informationen dem Skript übergeben. Handelt es sich bei der URI um einen Namen für eine statische Ressource (z. B. eine Datei), wird diese geschrieben oder überschrieben. Bietet die Ressource jedoch beide Primitiven an, muss die Unterscheidung durch die Anwendungsanforderungen abgebildet werden. Weitere HTTP-Methoden müssen ebenfalls über die Anwendungsanforderungen umgesetzt werden: HTTP-HEAD und HTTP-OPTIONS können bspw. über die GET-Primitive und einer Anforderung abgebildet werden, die die Art der angeforderten Informationen näher beschreibt (in diesem Fall ist die Art „Meta-Information“ oder „Optionen“).

Bewertung

Zusammenfassend und bezogen auf die Kriterien ergibt sich für eine Web-Browser-Anwendung:

- E_{A1} – Nutzung der API

  Die Abbildung der HTTP-Methoden auf die API-Primitiven zur Formulierung der Anfrage ist auf einfache Art und Weise möglich. In seltenen Fällen ist jedoch die Angabe von Anwendungsanforderungen zur genaueren Beschreibung der gewünschten Informationen notwendig. Meta-Informationen über die empfangenen Dateien können mit der API über die Metadaten des Ressourcen-Endpunkts abgerufen werden.

- E_{A2} – Annahmen über den Kommunikationsdienst

  Es wird ein zuverlässiger, Byte-stromorientierter Transport benötigt, der auch Informationen zur Anfrage und Meta-Informationen über den Inhalt übermitteln kann. Der Erfolg oder Misserfolg der Anfrage (Reaktionszeit) wird innerhalb weniger hundert Millisekunden erwartet. Diese Annahmen können als Anforderung dem entsprechenden Namensschema zugeordnet werden, sodass von der Anwendung keine weiteren Angaben bzgl. des Kommunikationsdienstes notwendig sind.

- E_{A3} – zusätzliche Informationen

  Über das verwendete Kommunikationsprotokoll werden keine weiteren Informationen benötigt.

7.2.2 Chat / Instant Messaging

Ein Beispiel einer interaktiven Anwendung mit moderaten zeitlichen Anforderungen und einem geringen Datenvolumen ist eine einfache Chat-Anwendung. Neben Instant-Messaging, bei dem die 1:1-Kommunikationsbeziehung zwischen den Teilnehmern im Vordergrund steht, sind auch Gruppen-Chats verbreitet. Auf diese wird im Kontext der Gruppenkommunikation in Abschnitt 7.2.5 weiter eingegangen.

Nutzung der Anwendungsschnittstelle

Eine mögliche Realisierung einer einfachen Chat-Anwendung mit der in Kapitel 5 vorgestellten Anwendungsschnittstelle wird in Abschnitt 5.4.1.1 beschrieben. Dabei kann entweder mit einem Anwendungsprotokoll gearbeitet werden (unter Verwendung von CONNECT) oder inhaltsorientiert (mit GET und PUT). Alternativ kann auch eine Kombination von BIND und PUT verwendet werden, um ein Publish/Subscribe-Paradigma zu realisieren (vgl. Abschnitt 5.4.1.3). Die verschiedenen Zugriffsarten sind auch in Kombination einsetzbar: So kann eine Anwendung vereinfacht mit GET und PUT arbeiten, eine andere mit BIND und PUT. Während diese Nutzung den Austausch von Chat-Nachrichten beschreibt, können Präsenzinformationen (z. B. Freundeslisten und Online-Status) über spezielle URIs abgerufen bzw. geändert werden (vgl. auch XMPP [Sain08]).

Besonderheiten

Für den Anwendungsdienst (die Server-Anwendung) bedeuten die verschiedenen Nutzungsmöglichkeiten, dass eine Unterstützung aller Primitiven implementiert werden muss. Je nach verwendeter Netzwerkarchitektur kann eine solche Server-Anwendung allerdings auch entfallen: Bei inhaltsbasierter Kommunikation können die Nachrichten direkt unter einer entsprechenden URI „im Netz“ abgelegt und von dort wieder abgerufen werden (siehe Abschnitte 7.3.4 und 7.3.7).

Bewertung

Zusammenfassend und bezogen auf die Kriterien ergibt sich für eine Chat-Anwendung:

- E_{A1} – Nutzung der API

  Hier sind verschiedene Zugriffsarten sinnvoll, die auch in Kombination eingesetzt werden können. Die zusätzlich notwendigen Meta-Informationen über die empfangenen Daten können auch hier über die Metadaten des Ressourcen-Endpunkts abgerufen werden.

- E_{A2} – Annahmen über den Kommunikationsdienst

  Es wird ein nachrichtenorientierter Transport benötigt, der auch Meta-Informationen über den Inhalt übermitteln kann, die dann über den Ressourcen-Endpunkt abrufbar sind. Die gewünschten Eigenschaften des Kommunikationsdienstes müssen von der Anwendung angegeben werden. Ob Erfolg oder Misserfolg zurückgemeldet werden sollen, oder in welchem zeitlichen Rahmen die Zustellung erfolgen soll, hängt ebenfalls von den durch die Anwendung angegebenen Anforderungen ab.

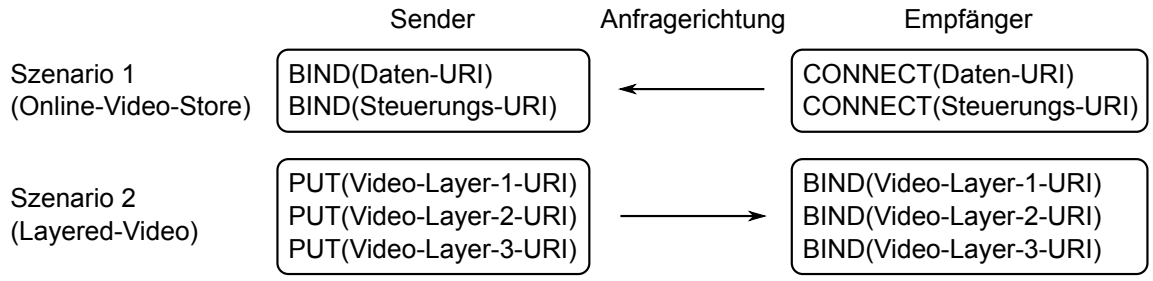


Abbildung 7.2 Unterschied der ersten beiden Video-Übertragungsszenarien.

- E_{A3} – zusätzliche Informationen

  Über das verwendete Kommunikationsprotokoll werden keine weiteren Informationen benötigt.

7.2.3 Video-Streaming

Für Video-Streaming wurden drei unterschiedliche Szenarien untersucht: Das erste Szenario repräsentiert den Online-Video-Dienst aus Abschnitt 5.4.3 und stellt Steuermöglichkeiten für den Benutzer bereit (Abb. 7.2 oben). Das zweite Szenario ermöglicht die Übertragung eines Video-Stroms in unterschiedlichen Qualitätsstufen (Abb. 7.2 unten). Das dritte Szenario zeigt, dass das verwendete Transportprotokoll ohne Änderung an der Anwendung ausgetauscht werden kann (nicht abgebildet).

Nutzung der Anwendungsschnittstelle

Der Empfang der Video-Daten im ersten Szenario geschieht über einen CONNECT-Aufruf mit einer URI, die direkt auf die Video-Daten verweist. Zusätzlich wird mit einem weiteren CONNECT ein Steuerungskanal geöffnet, der die Steuerung der Video-Daten (Pause, Vor- und Zurückspulen, Stop) erlaubt. Dieser Steuerungskanal wird durch eine zusätzliche URI identifiziert. Für die Video-Daten und für die Kontrolldaten wird jeweils ein zuverlässiger Kommunikationsdienst angefordert.

Die Video-Streaming-Anwendung aus dem zweiten Szenario ist ursprünglich im Rahmen von [Zinn12] entstanden und wurde von der Universität Würzburg auf die in Kapitel 5 vorgestellte Anwendungsschnittstelle angepasst. Diese Anwendung überträgt dabei die einzelnen Schichten eines Layered Video Codec (SVC – Scalable Video Codec) über separate API-Aufrufe, wobei jede Schicht des Video-Codec über eine erweiterte URI angesprochen wird. Zu jeder Schicht des Video werden unterschiedliche Anforderungen angegeben: Während die Basisschicht des Videos zuverlässig zugestellt werden soll, ist für die Zusatzschichten ein unzuverlässiger Kommunikationsdienst ausreichend. Der Initiator der Übertragung ist dabei zugleich die Video-Quelle,

weshalb dieser zur Initiierung der Übertragung die PUT-Primitive verwendet. Die Video-Senke signalisiert ihre Bereitschaft mittels eines BIND-Aufrufs.

Im dritten Szenario wurde gezeigt [MBVW+09], dass bei unveränderter Anwendung der verwendete Kommunikationsdienst ausgetauscht werden kann. So kann die Übertragungsqualität des Videos durch einen neuen, auf die Netzeigenschaften angepassten Kommunikationsdienst gesteigert werden, ohne dass die Anwendung modifiziert werden muss.

Besonderheiten

In den ersten beiden Szenarien nutzt die Anwendung jeweils zwei oder mehr Kommunikationsbeziehungen mit unterschiedlichen URIs für unterschiedliche Zwecke und Anwendungsanforderungen. Die notwendige Synchronisation der Daten insbesondere bei der Verwendung des Layered-Video-Codec muss dabei in der Anwendung erfolgen.

Bewertung

Zusammenfassend und bezogen auf die Kriterien ergibt sich für Video-Streaming-Anwendungen:

- E_{A1} – Nutzung der API

  Auch hier sind verschiedene Zugriffsarten abhängig von den Intentionen der Anwendungen sinnvoll. Mehrere API-Aufrufe für dasselbe Video – sei es zur Steuerung oder für unterschiedliche Schichten eines Videos – sind dabei kombinierbar. Zusammen mit der Angabe von unterschiedlichen Anwendungsanforderungen an den Kommunikationsdienst sind hier komplexe Beziehungen zwischen Video-(Teil)-Daten und den jeweils verwendeten Kommunikationsprotokollen formulierbar.

- E_{A2} – Annahmen über den Kommunikationsdienst

  Die gewünschten Eigenschaften des Kommunikationsdienstes müssen von der Anwendung angegeben werden. Live-Video-Streaming-Anwendungen gehen dabei in der Regel von Verzögerungen im Sub-Sekundenbereich aus, wobei für aufgezeichnete Videos Verzögerungen von mehreren Sekunden akzeptabel sind. Die Übergabe des erwarteten, durchschnittlichen Bandbreitenbedarfs kann dabei den darunterliegenden Protokollen helfen, die Übertragung zu optimieren.

- E_{A3} – zusätzliche Informationen

  Über das verwendete Kommunikationsprotokoll werden keine weiteren Informationen benötigt. Informationen über den Netzzustand (z. B.

Paketverluste, Laufzeitschwankungen) können der Anwendung jedoch helfen, schnellere Anpassungen durchzuführen (bspw. Erhöhung des Abspielpuffers oder Abschalten einer Video-Schicht).

7.2.4 Datei-Server

Ein Datei-Server bildet das Gegenstück zum im Abschnitt 7.2.1 bewerteten Web-Browser. Er muss in der Lage sein, Anfragen für verschiedene Ressourcen bedienen zu können, ohne sich auf jede einzeln binden zu müssen.

Nutzung der Anwendungsschnittstelle

Eine Datei-Server-Anwendung lässt sich wie in Abschnitt 5.3.3.2 beschrieben durch die Angabe einer Basis-URI realisieren, auf die sich die Server-Anwendung mittels BIND bindet. Im Rahmen des in Abschnitt 5.4.3 vorgestellten Szenarios eines Online-Video-Angebots wurde eine Python-basierte Anwendung genutzt, um die Inhalte der Web-Seite bereitzustellen. Eine weitere Instanz dieser Anwendung wurde zur Bereitstellung der zum Download angebotenen Video-Daten genutzt.

Besonderheiten

Zu den Metadaten, die über die API bei einer ankommenden Anfrage auf Server-Seite abgerufen werden können, gehört u. a. die angeforderte URI und die Primitive, die die Client-Anwendung verwendet hat. Basierend auf diesen Informationen kann die Server-Anwendung die passende Datei identifizieren und senden. Ein zusätzliches Anwendungsprotokoll ist nicht notwendig.

Bewertung

Bezogen auf die Kriterien ergibt sich für Anwendungen, die Inhalte über eine hierarchische Verzeichnisstruktur anbieten:

- E_{A1} – Nutzung der API

  Durch einen einzelnen BIND-Aufruf können beliebige weitere Ressourcen angeboten werden. Die genaue Ressourcenbezeichnung und die Primitive, die zur Anfrage genutzt wurde, können dabei über die Metadaten in Erfahrung gebracht werden.

- E_{A2} – Annahmen über den Kommunikationsdienst

  Die anbietende Anwendung muss nicht die Art des Kommuniktationsdienstes auswählen, sondern kann dies den anfragenden Anwendungen überlassen.

- E_{A3} – zusätzliche Informationen

  Über das verwendete Kommunikationsprotokoll werden keine weiteren Informationen benötigt.

7.2.5 Gruppenkommunikationsanwendungen

Anwendungen, die Gruppenkommunikationsdienste nutzen, können ebenfalls mit der hier vorgestellten Anwendungsschnittstelle realisiert werden. Hierzu muss die Anwendung zunächst einer Gruppe beitreten, über die sie dann Daten empfangen und versenden kann.

Nutzung der Anwendungsschnittstelle

Da das Publish/Subscribe-Paradigma (vgl. Abschnitt 5.4.1.3) auch für den Einsatz von Gruppenkommunikation geeignet ist, ist die Nutzung der API für Gruppenkommunikationsdienste ähnlich: Mit BIND tritt die Anwendung der Gruppe, die durch die URI identifiziert wird, bei oder erstellt diese. Für jeden Sender, der Daten an die Gruppe sendet, wird ein Ereignis über den Ereignis-Endpunkt signalisiert. Wird dieses mittels ACCEPT angenommen, können die Daten über den dadurch erstellten Ressourcen-Endpunkt empfangen werden. Mit PUT kann die Anwendung selbst Informationen an die Gruppe senden.

Besonderheiten

BIND und ACCEPT werden hier nicht zur Bereitstellung von Ressourcen genutzt, wie das bei klassischen Client/Server-Anwendungen der Fall ist, sondern um Daten von anderen Teilnehmern zu empfangen.

Bewertung

Bezogen auf die Kriterien ergibt sich für Anwendungen, die Inhalte mit einer Gruppe austauschen möchten:

- E_{A1} – Nutzung der API

  Genutzt werden BIND/ACCEPT zum Empfang der Daten und PUT zum Versand der Daten. Im Gegensatz zu dem in Abschnitt 5.4.1.3 beschrieben Publish/Subscribe-Paradigma bleibt GET ungenutzt, da in einer Gruppe ältere Nachrichten in der Regel nicht zwischengespeichert werden.

- E_{A2} – Annahmen über den Kommunikationsdienst

  Die Angabe der Art der Zustellung und der zeitliche Rahmen bleibt dem Sender bzw. dem Ersteller der Gruppe überlassen.

- E_{A3} – zusätzliche Informationen

  Über das verwendete Kommunikationsprotokoll werden keine weiteren Informationen benötigt.

Anwendung	E_{A1} Nutzung der API	E_{A2} Annahmen	E_{A3} zus. Inform.
Web-Browser	GET, PUT und Anforderungen	über Namensschema zuordenbar	keine
Chat	alle Primitiven	von Anw. formuliert	keine
Video	alle Primitiven	von Anw. formuliert	keine
Datei-Server	BIND	keine	keine
Gruppenanw.	BIND und PUT	von Gruppenersteller oder Sender formuliert	keine

Tabelle 7.1 Anwendungen und ihre Nutzung der Anwendungsschnittstelle

7.2.6 Zusammenfassung

In Tabelle 7.1 wird die Nutzung der Anwendungsschnittstelle durch die in den letzten Abschnitten beschriebenen Anwendungen zusammengefasst. Durch die Integration der GET/PUT-Semantiken, der URI-basierten Namensbindung und der Unterstützung von Metadaten kann in vielen Fällen auf ein entsprechendes Anwendungsprotokoll wie z. B. HTTP oder FTP verzichtet werden. Je nach Anwendung werden nicht immer alle Primitiven genutzt. Über die Gesamtheit der untersuchten Anwendungen haben sich jedoch alle als notwendig herausgestellt, wenn auf Anwendungsprotokolle zur Übermittlung der Anfrage und der Metadaten verzichtet werden soll.

Zur Realisierung eines Anwendungsdienstes müssen Anwendungen allerdings nach wie vor eine dienstspezifische Schnittstelle (Dienst-API) implementieren. Neben der Definition der zu verwendenden URIs für verschiedene Funktionen (z. B. für den Austausch von Präsenzinformationen in Chat-Anwendungen) müssen auch mögliche Metadaten und erlaubte Primitiven spezifiziert werden.

In vielen Fällen können die notwendigen Anforderungen an den Kommunikationsdienst bereits durch das verwendete URI-Namensschema abgeleitet werden. Da diese Namensschemata bereits heute durch die IANA verwaltet werden, können die entsprechenden Spezifikationsdokumente zur Definition der minimalen, voreingestellten und insgesamt möglichen Anforderungen genutzt werden.

Insgesamt lassen sich die Vor- und Nachteile durch die Verwendung der in Kapitel 5 beschriebenen Anwendungsschnittstelle wie folgt zusammenfassen:

- ✚ Verzicht auf Anwendungsprotokolle ist möglich
- ✚ leichter Umstieg möglich, da URIs heute schon in Verwendung sind
- ✚ Anforderungen sind zum Teil bereits dem Namensschema der URI zuordenbar
- ➖ bei einigen Funktionalitäten müssen Anforderung explizit formuliert werden

7.3 Netzwerkarchitekturen

In diesem Abschnitt werden die Realisierungen verschiedener Protokolle und Netzwerkarchitekturen für NENA beschrieben und anhand der Kriterien aus Abschnitt 7.1 bewertet. Es wird gezeigt, dass sich alle Protokolle und Netzwerkarchitekturen mit den Konzepten, die in dieser Arbeit entwickelt wurden, umsetzen lassen. Obwohl so nicht ausgeschlossen werden kann, dass für zukünftige Architekturen und Protokolle Anpassungen notwendig sind, wird durch die Vielfalt der hier gezeigten Beispiele mit jeweils unterschiedlichen Zielsetzungen ein breites Spektrum an Architekturen abgedeckt. Die durchgeführten Fallstudien und deren Besonderheiten sind:

- *Heutige Protokolle.* Wichtig ist zunächst die Kompatibilität mit heute eingesetzten Protokollen, die sich nahtlos in die Konzepte dieser Arbeit integrieren lassen. Dazu zählen traditionelle Ende-zu-Ende-Transportprotokolle und Anwendungsprotokolle, die durch die Verwendung der Anwendungsschnittstelle nicht mehr auf Anwendungsschicht realisiert werden müssen. Auf diese Protokolle wird in Abschnitt 7.3.1 eingegangen.

- *Video-Übertragung.* Ein heute weit verbreiteter Anwendungsfall ist die Übertragung von Video-Daten, sei es für Konferenzschaltungen oder für Streaming. In Abschnitt 7.2.3 wurden zwei Alternativen (Online-Video-Store und Layered-Video) betrachtet, die sich hauptsächlich aus Anwendungssicht unterscheiden. In Abschnitt 7.3.2 wird ein Protokoll näher betrachtet und bewertet, welches Video-Kodierungsfunktionalität enthält.

- *Schwarm-Download: BitTorrent.* Zur effizienteren Nutzung verfügbarer Bandbreiten und zur Lastverteilung hat sich BitTorrent etabliert. BitTorrent ist ein Schwarmprotokoll, das Teilnehmer mit gleichen Interessen an bestimmte Daten in einem Schwarm zusammenfasst. Die Daten werden dann blockweise unter den Teilnehmern ausgetauscht. Dies bedeutet allerdings, dass die Daten nicht mehr sequentiell bei einem Teilnehmer ankommen, wie es bei einem traditionellem Download der Fall ist. Eine

an BitTorrent angelehnte Realisierung wird in Abschnitt 7.3.3 beschrieben und bewertet.

- *Content-Centric Networking: CCNx.* Ein gänzlich neues Kommunikationsparadigma wird mit CCN, der inhaltsbasierten Kommunikation, angestrebt. Dieses Paradigma verwendet eine Abstraktion, die für viele Anwendungen beim Endbenutzer heute am verbreitesten ist: der Inhalt wird direkt adressiert, unabhängig von dessen Lokation. Die Integration von CCN in die in dieser Arbeit entwickelten Konzepte wird in Abschnitt 7.3.4 vorgestellt und bewertet.

- *eXtensible Internet Architecture (XIA).* Als Vorschlag für einen erweiterbaren Ersatz für die heutige TCP/IP-Architektur wurde XIA vorgeschlagen. Während die Grundarchitektur ähnlich zu TCP/IP ist, unterstützt XIA jedoch verschiedene Adressierungsparadigmen. Die Integration und die Bewertung von XIA für NENA wird in Abschnitt 7.3.5 beschrieben.

- *Delay-Tolerant Networking (DTN).* DTN ist aus zwei Gründen interessant: Zum einen handelt es sich um ein Store-And-Forward-Prinzip, welches mit Konnektivitätsunterbrechungen im Minuten, Stunden oder gar Tagebereich umgehen kann. Zum anderen erlaubt das als Referenz verwendete Protokoll die Kommunikation über verschiedene Netzwerkarchitekturen hinweg. Die Integration und die Bewertung der DTN Realisierung für NENA werden in Abschnitt 7.3.6 beschrieben.

- *Nachrichten-Broker: Publish/Subscribe.* Ein Dienst, der ebenfalls innerhalb von NENA realisiert wurde, ist ein einfacher Nachrichten-Broker mit seinen dazugehörigen Protokollen. Dieser wird in Abschnitt 7.3.7 beschrieben und bewertet.

Die jeweiligen Realisierungen sind in Abbildung 7.3 zusammengefasst. Durch die Abbildung wird angedeutet, welche Funktionalitäten bzw. Protokolle innerhalb von Netlets, Multiplexern bzw. Servlets realisiert wurden. Neben dieser Aufteilung auf die NENA-Komponenten ist auch die Umsetzung anderer NENA-spezifischer Aufgaben für die Bewertung relevant. Dazu gehören:

- die Kandidatenabfrage über den Request Mapper (Abschnitt 6.2.2.1)
- die Registrierung von Anwendungsdiensten (Abschnitt 6.2.2.2)
- die Zuordnung der Flow-State-Objekte für ankommende Nachrichten (Abschnitt 6.3)

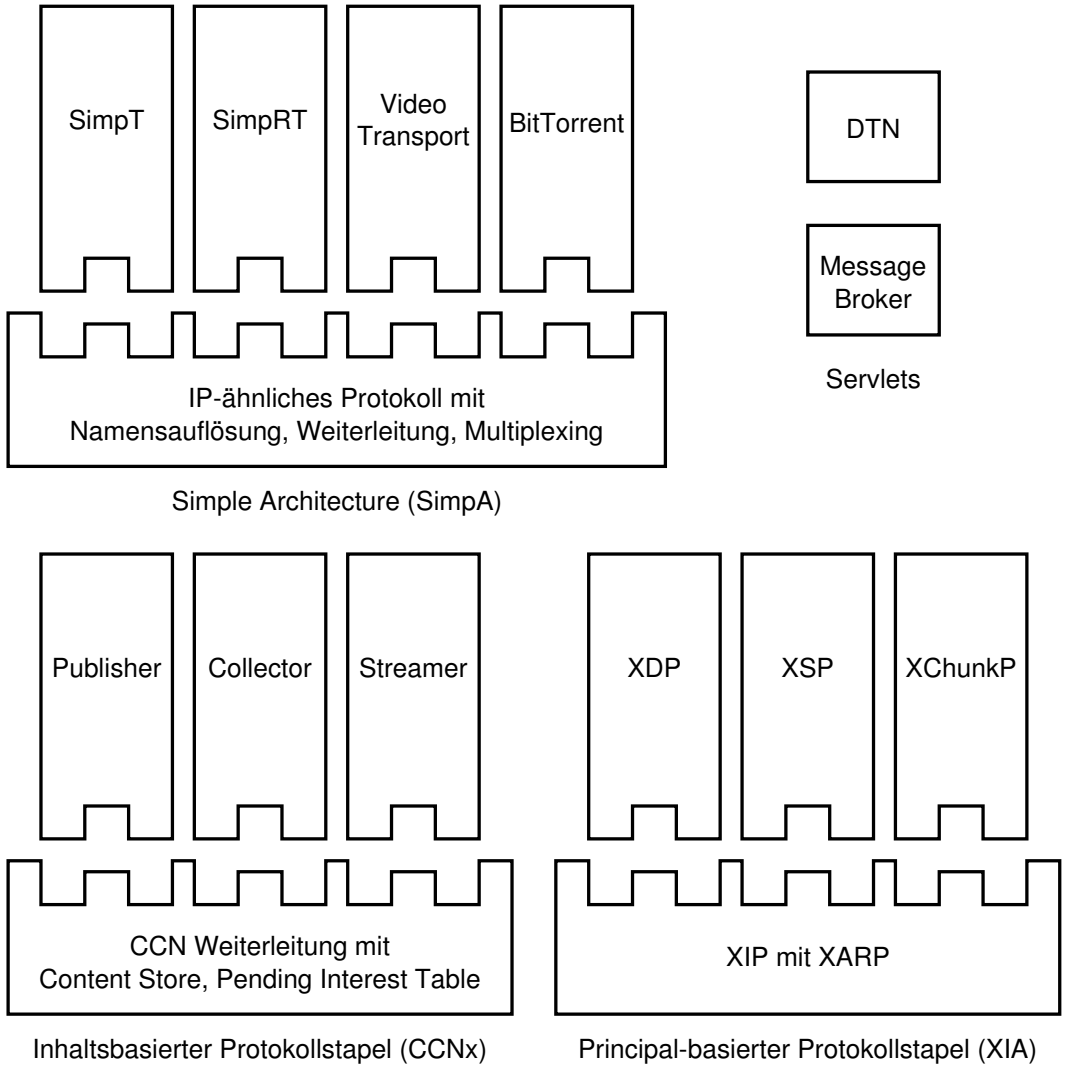


Abbildung 7.3 Überblick über die in NENA realisierten Protokolle und Netzwerkarchitekturen.

7.3.1 Heutige Protokolle

Zunächst wird die Umsetzung heutiger Kommunikationsparadigmen untersucht. Dazu zählt die Ende-zu-Ende-Kommunikation zwischen Anwendungen und die REST-basierte, an HTTP angelehnte Kommunikation.

7.3.1.1 Aufbau

Zur Bewertung wird die in Abschnitt 6.5 im Detail vorgestellte SimpA verwendet. Sie ist an die heutige TCP/IP-Architektur angelehnt: Eine Netzwerkschnittstelle, die eine Konnektivität zu einem Netz herstellt, erhält eine Schnittstellenadresse. Rechner (Hosts) haben Namen, die auf mehrere Schnittstellenadressen aufgelöst werden können. Quelladressen, Zieladressen und Protokoll-IDs werden in einem Basispaketkopf mit jedem Netzwerkpaket mitgeschickt (Schicht 3). Das Protokoll, welches diesen Basispaket-

kopf hinzufügt, übernimmt zudem die Weiterleitung von Netzwerkpaketen über die vorhandenen Schnittstellen. Verschiedene Transportprotokolle bieten unterschiedliche Transportdienste (Schicht 4). Darauf aufbauend erlaubt ein Protokoll die Übermittlung von Anfragen, die die Primitive (GET, PUT, CONNECT), die angeforderte URI und die Anwendungsanforderungen überträgt.

Umsetzung für NENA

Im Multiplexer befindet sich ein Schicht-3-Protokoll, welches neben dem Multiplexen der darüberliegenden Transportprotokolle auch die Weiterleitungsfunktionalität übernimmt (Abschnitt 6.5.1). Im Gegensatz zu IP wird an dieser Stelle auch die Namensauflösung durchgeführt. Außerdem übernimmt der Multiplexer das Multiplexen der Anwendungsströme und die Zuordnung der Zustandsobjekte zu den empfangenen Paketen. Zur Registrierung von Anwendungsdiensten werden keine speziellen Maßnahmen getroffen: Bei ankommenden Nachrichten, die einen unbekannten Anwendungsdienst adressieren, wird die von NENA bereitgestellte Liste mit registrierten Anwendungen durchsucht. Existiert ein passender Eintrag, wird ein Kind-Flow-State-Objekt (FSO) erstellt und der Nachricht zugeordnet. Anschließend wird die Nachricht an das passende Netlet weitergereicht.

Der Request Mapper zur Kandidatenauswahl ist als Teil des Multiplexers realisiert, da bei diesem alle Netlets der SimpA registriert sind. Eine neue Anwendungsanfrage wird an jedes registrierte Netlet weitergereicht, welches wiederum die Entscheidung trifft, ob es eine gegebene Anfrage grundsätzlich bedienen kann. Dazu gehört die Überprüfung der von der Anwendung verwendeten Primitive, der URI und der übergebenen Anforderungen. Welches Netlet am Ende ausgewählt wird, bleibt der Netlet-Selection-Komponente überlassen.

Ein Routing-Protokoll (SimpR), welches die Weiterleitungstabelle im Multiplexer verwaltet (Abschnitt 6.5.2.1), und zwei Transportprotokolle (SimpT und SimpRT, Abschnitt 6.5.2.2) sind als Netlets realisiert. Während das Routing-Protokoll seinen Dienst im Hintergrund zeitgesteuert übernimmt, bearbeiten die Transportprotokolle die Anfragen der Anwendungen.

Wie am einfachen Beispiel der SimpA-Transport-Netlets und am ausführlichen Beispiel der Protokollschablone für Transportprotokolle (Abschnitt 4.3) gezeigt wurde, können einzelne Protokollteile oder Mechanismen als Bausteine in verschiedenen Protokollen wiederverwendet werden. Alle Bausteine nutzen dabei Zustandsobjekte, die sie an die Flow-State-Objekte anhängen. Somit haben die Bausteine immer die relevanten Zustandsinformationen ohne selbst die Zuordnung zu den verschiedenen Flows vornehmen zu müssen.

Durch einen zusätzlichen REST-Baustein können durch das SimpRT-Netlet Anwendungsanfragen mit allen Primitiven bedient werden: Dieser Baustein

sendet die gegebene Primitive, die URI und die Anwendungsanforderungen an die entfernte Anwendung.

7.3.1.2 Bewertung

Im Vergleich zur TCP/IP-Architektur weißt die SimpA einige Unterschiede auf:

- Die Auflösung von Namen auf Schnittstellenadressen findet in der Basisprotokollschicht statt. Bei TCP/IP wird die DNS-Namensauflösung in der Anwendung angestoßen.

- Die Zuordnung des Anwendungs-Flows und des Anwendungsdienstes für ankommende Nachrichten findet ebenfalls in der Basisprotokollschicht statt. Bei TCP/IP findet die Port-Zuordnung im Transportprotokoll statt.

- Anfrageprotokolle sind im Netlet umgesetzt. Traditionell werden diese allerdings auf der Anwendungsschicht realisiert.

Diese Design-Entscheidungen in der SimpA sind jedoch Erweiterungen, die erst mit der neuen Anwendungsschnittstelle und mit dem Rahmenwerk möglich geworden sind. Eine direkte Umsetzung von TCP/IP in NENA ist ebenfalls möglich, mit entsprechendem Mehraufwand für den Anwendungsprogrammierer.

Nutzung der Anwendungsschnittstelle

Aus Sicht der SimpA-Protokolle ergibt sich als Bewertung für die Anwendungsschnittstelle:

- E_{P1} – Art und Weise der Nutzung

  Die Anwendungsschnittstelle wird in vollem Umfang unterstützt.

- E_{P2} – Annahmen über Anwendung

  Es wird angenommen, dass Anwendungen mit anderen Anwendungsdiensten direkt (Ende-zu-Ende) kommunizieren.

- E_{P3} – benötigte Zusatzinformationen über die Anwendung

  Die Anwendung muss selbst entscheiden, welche Art des Transports (zuverlässig oder unzuverlässig) sie wünscht und dies explizit angeben.

Integration in NENA

Die Integration der SimpA in NENA führte zu einigen Design-Unterschieden im Vergleich zur traditionellen TCP/IP-Architektur. Diese sind aber ausschließlich auf Vereinfachungen zurückzuführen, die das Rahmenwerk und die Anwendungsschnittstelle bieten.

- I_1 – Vollständigkeit (*Was fehlt?*)

  Bei der Umsetzen für NENA wurden keine Konzepte oder Abstraktionen vermisst.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Es mussten keine unnötigen Schnittstellen und Konzepte von NENA bei der Umsetzung berücksichtigt werden.

- I_3 – Limitationen (*Was stört?*)

  Bei der Umsetzung waren keine Konzepte von NENA störend oder verursachten unverhältnismäßigen Mehraufwand.

7.3.2 Video-Übertragung

Wie in Abschnitt 7.2.3 beschrieben ist, wurden drei unterschiedliche Video-Streaming-Alternativen untersucht. Während sich die ersten zwei Alternativen (Online-Video-Store und Layered-Video) hauptsächlich aus Sicht der Anwendung unterscheiden und die genutzten Transportprotokolle dieselben sind, wird bei der dritten Alternative ein spezielles Video-Transport-Netlet [MBVW+09] genutzt, welches in diesem Abschnitt beschrieben wird.

7.3.2.1 Aufbau

Die Hauptaufgabe des Video-Transport-Netlets ist die Kodierung von Video-Daten für eine geeignete Übertragung über das Netzwerk (und umgekehrt die Dekodierung der empfangenen Daten). Das Video-Transport-Netlet besteht aus den vier Bausteinen, die in Abbildung 7.4 dargestellt sind:

1. *Converter.* Dieser Baustein formt die Video-Daten, wie sie von der Anwendung bereitgestellt werden, in 8×8 Blöcke zur weiteren Verarbeitung um. Diese Funktion in einen Baustein auszulagern hat den Vorteil, dass andere Video-Formate unterstützt werden können, indem dieser Baustein ausgetauscht wird.

2. *IDCT.* Dieser Baustein führt für ausgehende Video-Daten eine diskrete Kosinustransformation und für eingehende Video-Daten eine inverse diskrete Kosinustransformation zur weiteren Verarbeitung durch. Diese

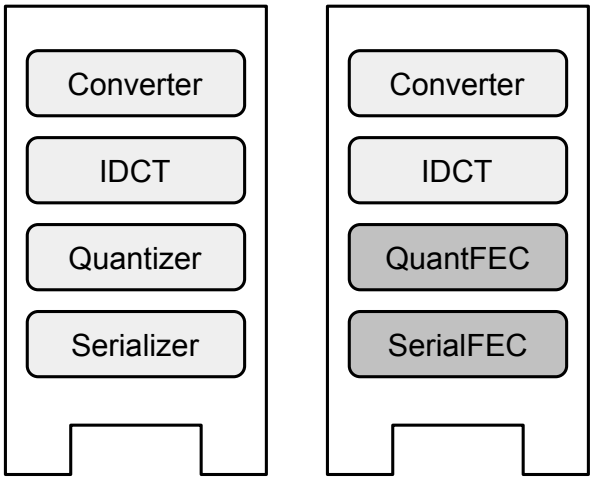


Abbildung 7.4 Video-Transport-Netlet mit angepasstem Encoder und Decoder.

Funktion in einen Baustein auszulagern bietet den Vorteil, dass andere, auf die Hardware optimiert Transformationsalgorithmen verwendet werden können.

3. *Quantizer.* Dieser Baustein quantifiziert die Matrix-Koeffizienten, die durch den IDCT-Baustein gegeben sind. Durch diesen Baustein wird die stärke der verlustbehafteten Kompression reguliert.

4. *Serializer.* Dieser Baustein serialisiert die Bildblöcke zur Übertragung über das Netzwerk.

Für den Quantizer und für den Serializer existieren alternative Bausteine, die Informationen für die Vorwärtsfehlerkorrektur (Forward Error Correction, FEC) hinzufügen.

Umsetzung für NENA

Das Video-Transport-Netlet erwartet eine URI mit Namensraum `video://`. Über diesen Namensraum ist das Format, in dem Anwendung und Netlet die Video-Daten austauschen, festgelegt. Die Transporteigenschaften werden ebenfalls darüber bestimmt, sodass die Anwendung keine weiteren Angaben zu Anwendungsanforderungen machen muss. Die Entscheidung, ob das Netlet mit oder ohne FEC verwendet wird, wird in diesem Szenario durch die Netzeigenschaften – in diesem Fall durch die Paketverlustrate – bestimmt. Als Basisprotokollschicht nutzt das Video-Transport-Netlet die SimpA.

7.3.2.2 Bewertung

Die beiden Varianten des Video-Transport-Netlets – mit und ohne FEC – sind ein Beispiel dafür, dass Protokolle auch ohne Wissen und Änderung der Anwendung ausgetauscht werden können. Außerdem sind die Zuordnung der

Transporteigenschaften zum Namensraum der URI und die Entkopplung des Codecs zur Übertragung hier eine sinnvolle Möglichkeit, den Anwendungsprogrammierer von Übertragungsdetails zu entlasten.

Nutzung der Anwendungsschnittstelle

Aus Sicht der Video-Transport-Netlets ergibt sich für die Anwendungsschnittstelle:

- E_{P1} – Art und Weise der Nutzung

  Es werden nicht alle Primitiven unterstützt, da kein Anfrageprotokoll realisiert wurde. Das Netlet meldet sich als Kandidat nur, wenn der passende Namensraum in der URI angefordert wird.

- E_{P2} – Annahmen über Anwendung

  Die Video-Daten müssen von der Anwendung in einem vorgegebenen Format an das Netlet weitergegeben werden. Dieses Format wird mit dem Namensraum festgelegt. Weiterhin wird angenommen, dass die Anwendung mit unvollständigen Video-Bildern umgehen kann (was wiederum mit dem Namensraum festgelegt wird).

- E_{P3} – benötigte Zusatzinformationen über die Anwendung

  Es werden keine Zusatzinformationen von der Anwendung erwartet.

Integration in NENA

Bei der Integration der Video-Transport-Netlets für die SimpA in NENA traten keine Schwierigkeiten auf. Die Bausteine mit FEC-Funktionalität erben dabei vom gleichen Bausteintyp wie die Bausteine ohne FEC-Funktionalität.

- I_1 – Vollständigkeit (*Was fehlt?*)

  Bei der Umsetzen für NENA wurden keine Konzepte oder Abstraktionen vermisst.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Es mussten keine unnötigen Schnittstellen und Konzepte von NENA bei der Umsetzung berücksichtigt werden.

- I_3 – Limitationen (*Was stört?*)

  Bei der Umsetzung waren keine Konzepte von NENA störend oder verursachten unverhältnismäßigen Mehraufwand.

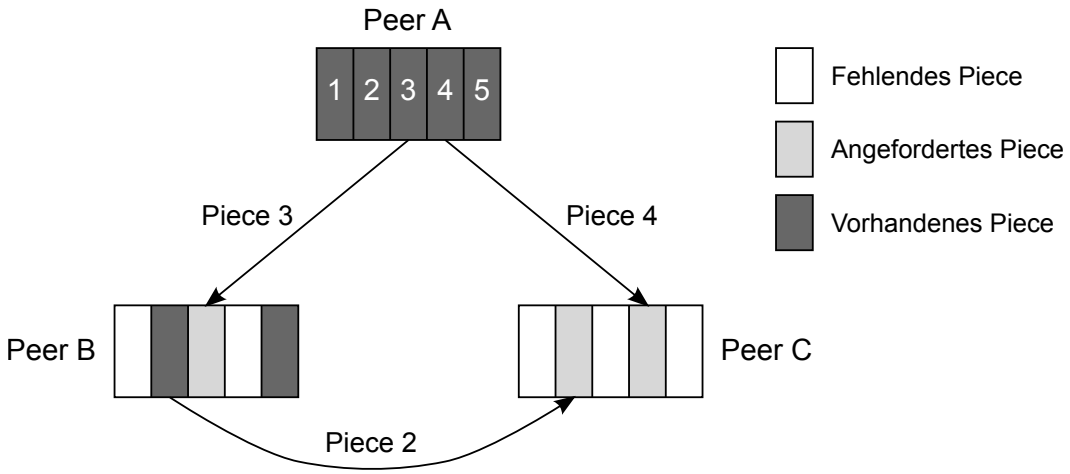


Abbildung 7.5 Datenaustausch in einem Schwarm über BitTorrent.

7.3.3 Schwarm-Download: BitTorrent

BitTorrent [Cohe03] ist ein Peer-to-Peer-(P2P)-Protokoll, welches Teile einer größeren Datei gleichzeitig von verschiedenen anderen P2P-Teilnehmern herunterläd. Die zwei wesentlichen Besonderheiten an BitTorrent sind zum einen der nicht-sequentielle Empfang der Daten, zum anderen die benötigten Zusatzinformationen, die in sog. *Torrents* abgelegt sind.

7.3.3.1 Aufbau

BitTorrent unterteilt Dateien in sog. *Pieces*, die einzeln von verschiedenen Teilnehmern (*Peers*) angefordert werden können (Abbildung 7.5). Die Menge an Teilnehmern, die dabei die gleiche Datei besitzen oder an ihr interessiert sind, wird als *Schwarm* bezeichnet. Innerhalb des Schwarms werden die Pieces nach bestimmten Fairness-Richtlinien ausgetauscht. So wird gefördert, dass ein Teilnehmer die Daten nicht nur herunterläd, sondern gleichzeitig auch wieder bereitstellt. Auch wenn der Teilnehmer selbst noch nicht alle Pieces der Datei besitzt, kann er bereits als Anbieter der Pieces auftreten, die er bereits heruntergeladen hat.

Während der Datenaustausch dezentral realisiert ist, setzt die ursprüngliche BitTorrent-Variante einen Server als sog. *Tracker* voraus, der eine Liste aller Teilnehmer an einem bestimmten Schwarm bereitstellt und aktuell hält. Durch die Verwendung verteilter Hash-Tabellen (DHTs) sind mittlererweile allerdings auch trackerlose Varianten möglich. Die Meta-Informationen zu einer Datei (oder einer Sammlung von Dateien) werden in einem sogenannten Torrent beschrieben. Zu diesen Meta-Informationen gehören Hash-Werte und Piece-Größen, die die Identifikation und das Zusammenfügen der Datei erlauben.

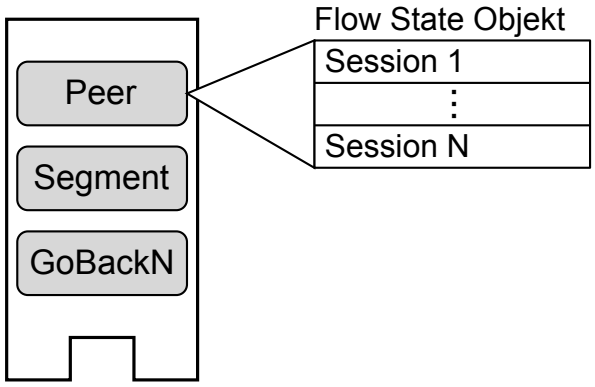


Abbildung 7.6 Die von BitTorrent genutzten Bausteine und die Nutzung des Flow State Objekts.

Umsetzung für NENA

Eine Anwendung, die Dateien herunterladen möchte, soll dies protokollunabhängig tun können. Aus diesem Grund wurde die Verwendung von Torrents aus der Anwendung selbst verbannt, und ein Baustein des Netlets bildet die gegebene URI auf den passenden Torrent ab. Die Anwendungsschnittstelle wird wie folgt verwendet: Mit GET wird eine Datei angefordert, woraufhin die URI auf den entsprechenden Torrent aufgelöst wird. Mit BIND wird eine neue Datei bereitgestellt, was das Generieren eines Torrents zur Folge hat. PUT und CONNECT werden nicht unterstützt.

Die Bausteine, mit denen BitTorrent in NENA realisiert worden ist [Volk13], sind in Abbildung 7.6 skizziert. Der Peer-Baustein übernimmt die Verarbeitung von Verbindungsanfragen von lokalen Anwendungen oder entfernten Peers. Dazu löst er auch URIs auf die entsprechenden Torrents auf. Für jede Anfrage wird ein *Session*-Objekt erzeugt, welches für die Verwaltung des Downloads und des Verbindungs-Managements verantwortlich ist. Dieses Session-Objekt wird im Flow-State-Objekt hinterlegt. Da die vom Peer-Baustein zu versendenden Nachrichten die maximal zulässige Größe überschreiten können, ist ein Segmentier-Baustein notwendig. Für die zuverlässige Übertragung wurde ein existierender Baustein genutzt, der ein GoBackN-ARQ-Verfahren realisiert.

7.3.3.2 Bewertung

Der Empfang großer Dateien mittels GET ist aufgrund des nicht-sequentiellen Downloads insofern eine Herausforderung, als das im schlechtesten Fall die gesamten Daten durch NENA gepuffert werden müssen, bevor sie an die Anwendung ausgeliefert werden können. Dies ist dann der Fall, wenn das erste Piece der Datei als letztes empfangen wird. Bei der Verwendung delegierter Übertragungen (vgl. Abschnitt 5.3.5) besteht dieses Problem nicht, weswegen diese hier vorgezogen werden sollten.

Nutzung der Anwendungsschnittstelle

Aus Sicht des BitTorrent-Protokolls ergibt sich für die Kriterien zur Nutzung der Anwendungsschnittstelle:

- E_{P1} – Art und Weise der Nutzung

  Als vorteilhaft ergibt sich hier die Verwendung der delegierten Übertragung. Zur Nutzung von BitTorrent über die Anwendungsschnittstelle musste eine Abbildung von URIs auf ein Torrent durch einen zusätzlichen Baustein im Netlet realisiert werden.

- E_{P2} – Annahmen über Anwendung

  Bei der Anforderung der Daten über eine (nicht-delegierte) GET-Anfrage wird davon ausgegangen, dass die Anwendung nötigenfalls auch damit umgehen kann, wenn zunächst für eine längere Zeit keine Daten über die Anwendungsschnittstelle bereitgestellt werden.

- E_{P3} – benötigte Zusatzinformationen über die Anwendung

  Es werden keine Zusatzinformationen von der Anwendung benötigt. Durch die automatische Erstellung von Torrents bleiben der Anwendung jedoch zunächst Optionen wie die Größe der Pieces bei der Bereitstellung neuer Dateien vorenthalten. Durch spezielle Anwendungsanforderungen können aber auch diese Optionen konfiguriert werden.

Integration in NENA

Für die Integration von BitTorrent als Netlet in NENA ergibt sich:

- I_1 – Vollständigkeit (*Was fehlt?*)

  Bei der Umsetzen für NENA wurden keine Konzepte oder Abstraktionen vermisst.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Es mussten keine unnötigen Schnittstellen und Konzepte von NENA bei der Umsetzung berücksichtigt werden.

- I_3 – Limitationen (*Was stört?*)

  Der sequentielle Datenaustausch zwischen Anwendung und Netlet ist für BitTorrent ungeeignet. Mit der delegierten Übertragung steht hier jedoch eine sinnvolle Alternative zur Verfügung.

7.3.4 Content-Centric Networking: CCNx

Content-Centric Networking (CCN) [JSTP+09] ist ein Kommunikationsparadigma, bei dem der Inhalt im Vordergrund steht. Während der Abruf der Inhalte aus heutiger Client-Sicht für die Anwendung keine große Änderung bedeutet, ist die Bereitsstellung der Inhalte anders als heute.

7.3.4.1 Aufbau

Mit CCN wird der angeforderte Inhalt direkt mit seinem Namen adressiert, welcher unabhängig von der Lokation des Inhalts ist. Der Inhalt kann zudem auf Netzwerkknoten wie Router zwischengespeichert werden, sodass weitere Anfragen an denselben Inhalt direkt von diesem Zwischenspeicher bedient werden können. Inhalte, die häufig nachgefragt werden, können so im Durchschnitt schneller ausgeliefert werden, als wenn sie bei jeder Anfrage direkt vom Server bereit gestellt werden müssen. Zudem wird dadurch der Netzwerkverkehr auf Kosten von (günstigem) Speicherplatz reduziert.

Eine Realisierung des CCN-Ansatz ist CCNx[1]. Hier wird ein neuer Schnittstellentyp eingeführt, der *Face* heißt. Dieser Begriff leitet sich von *Interface* (Schnittstelle) ab und soll im wesentlichen widerspiegeln, dass für das Weiterleitungsverhalten von CCNx die Art der Schnittstelle – also ob es sich um eine Schnittstelle in Richtung Netzwerk oder in Richtung Anwendung handelt – keine Rolle spielt. Anfragen auf Inhalte werden mit *Interests* formuliert, die als Nachrichten versandt werden. Ein Knoten, der ein Interest über ein Face empfängt, fügt dieses Interest seiner *Pending Interest Table* (PIT) hinzu, mit dem Vermerk über welches Face das Interest hereinkam. Sofern das Interest noch nicht in seiner PIT enthalten war, leitet er es zusätzlich über die Faces in Richtung des Ziels weiter. Diese Faces werden aus einer *Forwarding Information Base* (FIB) bestimmt. Wird ein Datenpaket passend zu einem Interest aus der PIT empfangen, wird dieses über die Faces weitergeleitet, über die Interests zu diesem Datenpaket empfangen wurden. Zusätzlich wird das Datenpaket im sog. *Content Store* für zukünftige Anfragen abgelegt, sofern dieser genügend Speicherplatz zur Verfügung hat.

Umsetzung für NENA

Die Umsetzung von CCNx für NENA [Funk12] ist in Abbildung 7.7 skizziert. Die CCN-Weiterleitungsfunktionalität, die PIT, die FIB und der Content Store sind im Multiplexer realisiert. Die Message Passing Schnittstellen zu Netlets und zu den Network Adaptors werden als Faces verwendet. Die Aufgaben zur Bearbeitung von Anwendungsanfragen werden in verschiedene Netlets gekapselt.

[1] http://www.ccnx.org

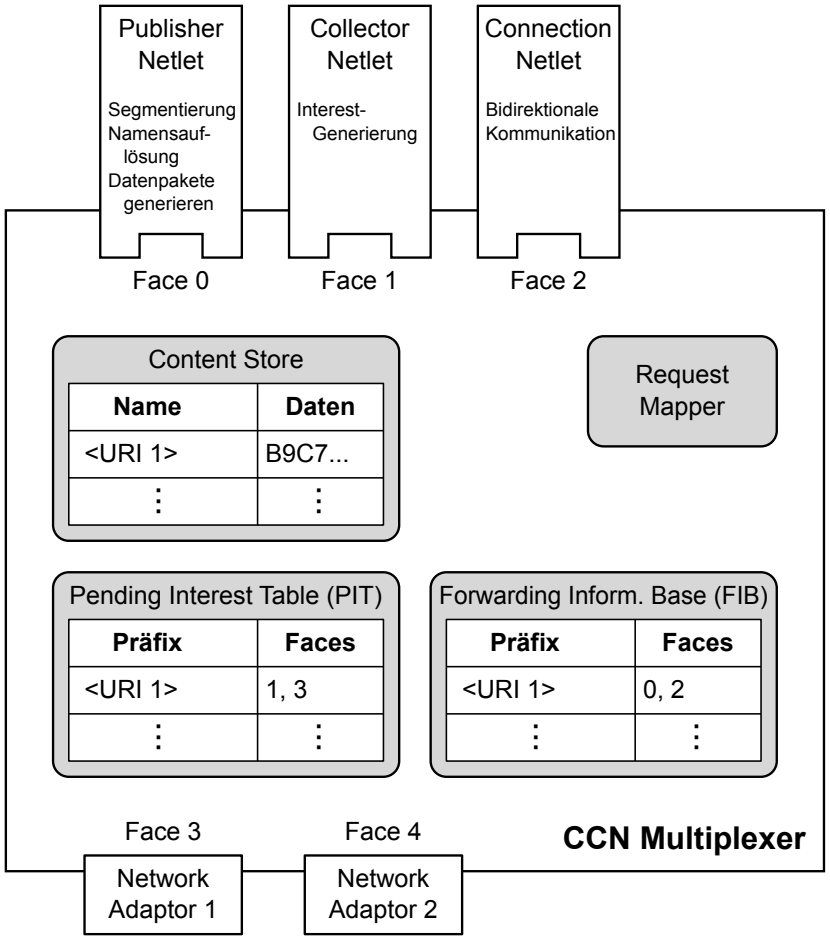


Abbildung 7.7 Umsetzung von CCNx in NENA.

Für GET-Anfragen ist das Collector Netlet zuständig, welches passende Interests zur von der Anwendung übergebenen URI generiert. Interests fordern dabei immer nur Teile einer Datei an, sodass eine Anwendungsanfrage in mehrere Interests resultieren kann. Das Collector Netlet setzt die empfangenen Datenpakete dann wieder zusammen und liefert sie an die Anwendung aus.

Das Publisher-Netlet übernimmt die Bereitstellung von Inhalten. Im Gegensatz zu klassischer Ende-zu-Ende-Kommunikation muss die Anwendung hier nicht warten, bis Anfragen anderer Anwendungen empfangen werden: Die Anwendung übergibt die zu veröffentlichen Inhalte komplett an den Protokollstapel, der diese zunächst im lokalen Content Store ablegt. Anschließende Anfragen werden daraufhin direkt vom Content Store bedient. Aufgrund der Tatsache, dass die Inhalte direkt an den Protokollstapel übergeben werden, wird diese Funktionalität über die PUT-Primitive den Anwendungen bereitgestellt.

Für interaktive Ende-zu-Ende-Verbindungen zwischen Anwendungen, bspw. für Sprachkommunikation, schlägt [JSBP+09] einen Ansatz vor, der kleins-

te Einheiten (in diesem Fall Audio-Samples) in einzelne Datenpakete unterteilt, die kontinuierlich mit entsprechenden Interests angefordert werden. Diese Funktionalität wurde im Rahmen des CCN Connection Netlet umgesetzt, welches wiederum mit den Primitiven CONNECT und BIND angesprochen wird.

Zur Bezeichnung der Inhalte verwendet CCNx URIs. Diese sind zusammengesetzt aus einem *Application Supplied Name* und CCNx-spezifischen Teilen, die u. a. die Version und die Blocknummer enthalten. Bei der Umsetzung für NENA übergeben Anwendungen den Application Supplied Name direkt über die Anwendungsschnittstelle, während die CCNx-spezifischen Teile automatisch durch die Netlets ergänzt werden.

Der für NENA zu implementierende Request Mapper ist wie bei der SimpA im Multiplexer realisiert. Neue Anwendungsanfragen werden auch hier wieder an die Netlets weitergereicht, die individuell entscheiden, ob sie eine Anfrage bedienen können.

7.3.4.2 Bewertung

Durch die Bereitstellung der Primitiven GET und PUT über die Anwendungsschnittstelle lässt sich der CCN-Ansatz problemlos von Anwendungen nutzen, die primär an Inhalte interessiert sind. Für die interaktive Kommunikation wird zwar auch ein Netlet bereitgestellt. Durch die Generierung von vielen Interests ist diese Art der Kommunikation über CCN allerdings ungünstig und sollte vermieden werden. Durch NENA kann dieser Nachteil des CCN-Ansatzes allerdings durch andere Protokollstapel kompensiert werden, die für die interaktive Kommunikation besser geeignet sind.

Nutzung der Anwendungsschnittstelle

Die CCN-Umsetzung in NENA profitiert vor allem von den inhaltsorientierten Primitiven GET und PUT.

- E_{P1} – Art und Weise der Nutzung

  Die CCN-Umsetzung in NENA lässt sich über alle Primitiven von der Anwendung aus nutzen, wobei GET und PUT für die inhaltsbasierte Kommunikation verwendet werden, CONNECT und BIND für interaktive Kommunikation. CONNECT und BIND werden dabei ausschließlich für die interaktive Kommunikation genutzt, sodass es nicht vorgesehen ist, mit BIND Inhalts-Ressourcen anzubieten (wie bspw. bei einem Datei-Server).

- E_{P2} – Annahmen über Anwendung

  Von der CCN-Umsetzung werden keine Annahmen bzgl. der Anwendung getroffen.

- E_{P3} – benötigte Zusatzinformationen über die Anwendung

  Wenn die Anwendung die Daten nur für eine gewisse Zeit zur Verfügung stellen möchte, ist ggfs. die Angabe eines Ablaufdatums notwendig. Dies kann über die Anwendungsanforderungen übergeben werden.

Integration in NENA

Bei der Integration in NENA traten keine Schwierigkeiten auf. Der Multiplexer enthält einen Großteil der CCN-Funktionalität, während die Netlets im Vergleich wenig Funktionalität beinhalten.

- I_1 – Vollständigkeit (*Was fehlt?*)

  Bei der Umsetzen für NENA wurden keine Konzepte oder Abstraktionen vermisst.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Es mussten keine unnötigen Schnittstellen und Konzepte von NENA bei der Umsetzung berücksichtigt werden. Das Flow State Objekt wird bei CCN zwar nur zwischen Flow Connector und Netlets genutzt, bietet dort aber die Möglichkeit Flow-spezifische Informationen einfach zu hinterlegen. Das Flow-Konzept ist im Multiplexer selbst nicht mehr notwendig.

- I_3 – Limitationen (*Was stört?*)

  Bei der Umsetzung waren keine Konzepte von NENA störend oder verursachten unverhältnismäßigen Mehraufwand.

7.3.5 eXtensible Internet Architecture (XIA)

Die eXtensible Internet Architecture XIA [HADL+12] definiert eine neue Internet-Architektur. Sie ist zwar grundsätzlich an die heutige Architektur angelehnt, jedoch erlaubt sie unterschiedliche Paradigmen in Subnetzen und beschreibt Regeln zur Interoperabilität.

7.3.5.1 Aufbau

XIA weist viele Ähnlichkeiten zur heutigen Internet-Architektur auf. So existieren Protokolle, die die Dienste von ARP, UDP, TCP und DHCP anbieten. Auch die Schnittstelle ist ähnlich zur heutige Socket-Schnittstelle, mit dem Unterschied, dass statt einer Adresse ein gerichteter, azyklischer Graph (*directed acyclic graph*, DAG) angegeben werden muss. Dieser gibt u. a. die zu benutzenden *Principals* an. Über Principals erlaubt XIA die Erweiterung der Architektur durch verschiedene Paradigmen. Als Principals sind bisher definiert:

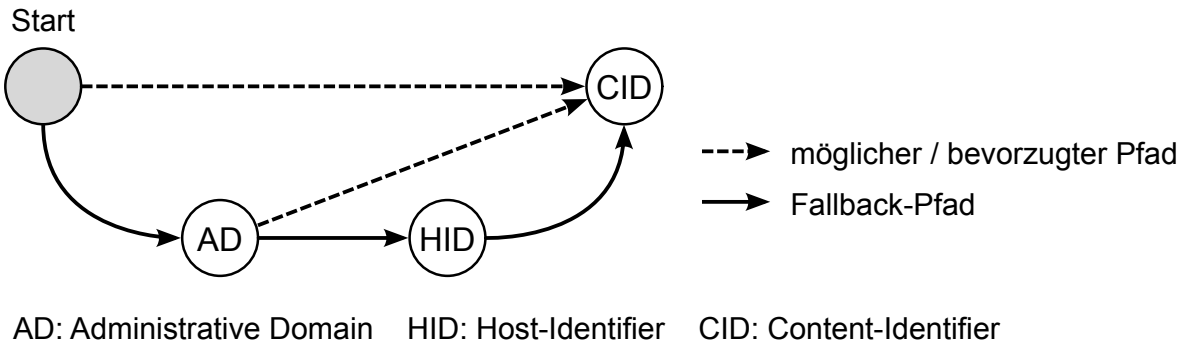


Abbildung 7.8 Ein DAG nach XIA mit verschiedenen Principals.

- Administrative Domain (AD) für Verwaltungsaufgaben
- Host Identifier (HID) zur Adressierung von einzelnen Systemen (Hosts)
- Service Identifier (SID) zur Adressierung von Diensten
- Content Identifier (CID) zur Adressierung von Inhalten (ähnlich zu CCN)

Ein DAG kann dabei mehrere Principals enthalten, die sog. Fallback-Alternativen angeben, falls über einen gegebenen Principal am aktuellen Knoten keine Weiterleitung erfolgen kann. Dies ist in Abbildung 7.8 veranschaulicht: Es wird vom sendenden Knoten zunächst versucht eine Nachricht über den CID an ihr Ziel zuzustellen. Ist dies nicht möglich, wird auf die AD als Fallback zurückgegriffen. Von dort wird erneut versucht, über den CID die Nachricht zuzustellen. Gelingt auch dies nicht, kann wieder auf einen anderen Principal, in diesem Fall dem HID, zurückgegriffen werden.

Umsetzung für NENA

XIA lässt sich für NENA auf ähnliche Weise umsetzen [Schl13] wie heutige Protokolle (Abb. 7.9): XIP dient als Basisprotokoll und ist im Multiplexer realisiert. Dort befindet sich auch die Forwarding Information Base auf Basis von DAGs. Die Auswahl, über welchen Principal eine Nachricht weitergeleitet wird, erfolgt ebenfalls hier. Wie auch bei der SimpA wird die Auflösung von URIs nach Adressen (hier: DAGs) ebenfalls im Multiplexer durchgeführt.

Die im XIA-Prototyp bisher existierenden Transportprotokolle sind als Netlets realisiert. Dazu gehören ein unzuverlässiger Transport (XDP), ein zuverlässiger Transport (XSP) und ein inhaltsbasierter Transport (XChunkP). CONNECT-Anfragen können dabei von XDP und XSP bedient werden, GET- und PUT-Anfragen werden ausschließlich von XChunkP bedient. Bindet sich eine Anwendung auf eine URI, um einen Dienst anzubieten, wird dieser Dienst als SID Principal bei einem Namensdienst im XIA-Netz registriert. Der Multiplexer ergänzt dabei den DAG um die AD und dem HID.

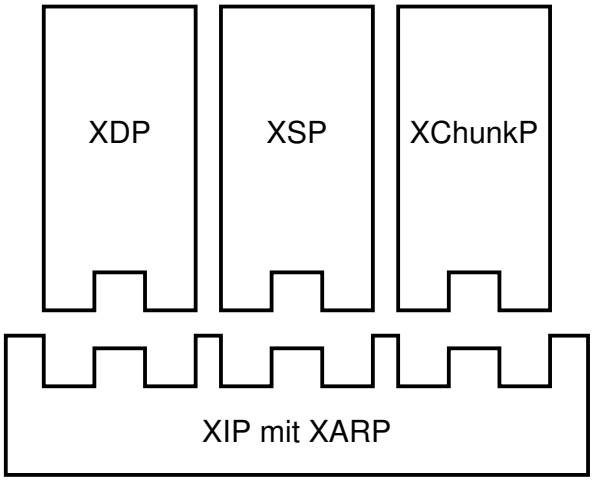


Abbildung 7.9 XIA-Protokolle aufgeteilt auf Multiplexer und Netlets.

7.3.5.2 Bewertung

Ein Vorteil bei der Integration von XIA in NENA ist, dass die Namensauflösung von URI nach DAG nicht mehr durch die Anwendung erfolgen muss, wie es mit der ursprünglich vorgeschlagenen Xsocket-Schnittstelle für XIA vorgeschlagen wurde. Durch die Flexibilität von XIA kann eine Vielzahl von Netzwerkparadigmen durch das Hinzufügen von Principals erreicht werden.

Nutzung der Anwendungsschnittstelle

Aus Sicht von XIA ergibt sich für die Kriterien zur Nutzung der Anwendungsschnittstelle:

- E_{P1} – Art und Weise der Nutzung

  Es werden alle Primitiven unterstützt. Durch die Angabe von Anwendungsanforderungen kann zudem die Art des Transportdienstes gewählt werden. Durch die zusätzliche Abbildung von URIs auf DAGs muss der Anwendungsentwickler keine DAGs mehr angeben wie das bei der XIA-Socket-Schnittstelle der Fall ist.

- E_{P2} – Annahmen über Anwendung

  Es werden keine weiteren Annahmen über Anwendungen getroffen.

- E_{P3} – benötigte Zusatzinformationen über die Anwendung

  Es werden keine XIA-spezifischen Zusatzinformationen über die Anwendung benötigt.

Integration in NENA

Die Integration von XIA in NENA weist viele Parallelen zur Integration der SimpA auf.

- I_1 – Vollständigkeit (*Was fehlt?*)

  Bei der Umsetzen für NENA wurden keine Konzepte oder Abstraktionen vermisst. Auch hier konnten die Flow State Objekte zur einfacheren Zuordnung von Anwendungs-Flows und zur Zustandsverwaltung der einzelnen Protokolle genutzt werden.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Es mussten keine unnötigen Schnittstellen und Konzepte von NENA bei der Umsetzung berücksichtigt werden.

- I_3 – Limitationen (*Was stört?*)

  Bei der Umsetzung waren keine Konzepte von NENA störend oder verursachten unverhältnismäßigen Mehraufwand.

7.3.6 Delay-Tolerant Networking (DTN)

Delay-Tolerant Networking (DTN) beschreibt die Kommunikation zwischen Endsystemen über ein Netzwerk mit hohen Verzögerungen oder gar unterbrochener Konnektivität.

7.3.6.1 Aufbau

In DTNs wird häufig das Store-And-Forward-Prinzip angewandt: Eine Nachricht (auch als *Bundle* bezeichnet) kann dabei auf Zwischensystemen zwischengespeichert werden bis es zum nächsten System weitergeleitet werden kann. Ein DTN-Protokoll ist das Bundle Protocol [ScBu07], welches für die Realisierung von DTN in NENA als Referenz verwendet wurde [Lien13].

Das Bundle Protocol ist ein Protokoll auf Anwendungsschicht, welches zur Adressierung des Empfängers URIs verwendet. Es wurde unabhängig von der darunterliegenden Netzwerkarchitektur entworfen, sodass auch andere Protokollstapel als TCP/IP verwendet werden können. Die namensbasierte Adressierung und die architekturübergreifende Funktionalität waren der Grund für die Auswahl dieses Protokolls.

Umsetzung für NENA

Netlets sind in NENA immer einem bestimmten Multiplexer und damit einer bestimmten Netzwerkarchitektur zugeordnet. Aus diesem Grund sind sie für einen DTN-Dienst ungeeignet, wenn dieser über verschiedene Netzwerkarchitekturen hinweg angeboten werden soll. Stattdessen werden für die Realisierung von DTN *Servlets* genutzt (vgl. Abschnitt 6.2.1.3). Neben dem DTN-Servlet, welches das BP realisiert, wurde ein weiteres Servlet entworfen, um die Erreichbarkeit von anderen Systemen zu überwachen: das Discovery-

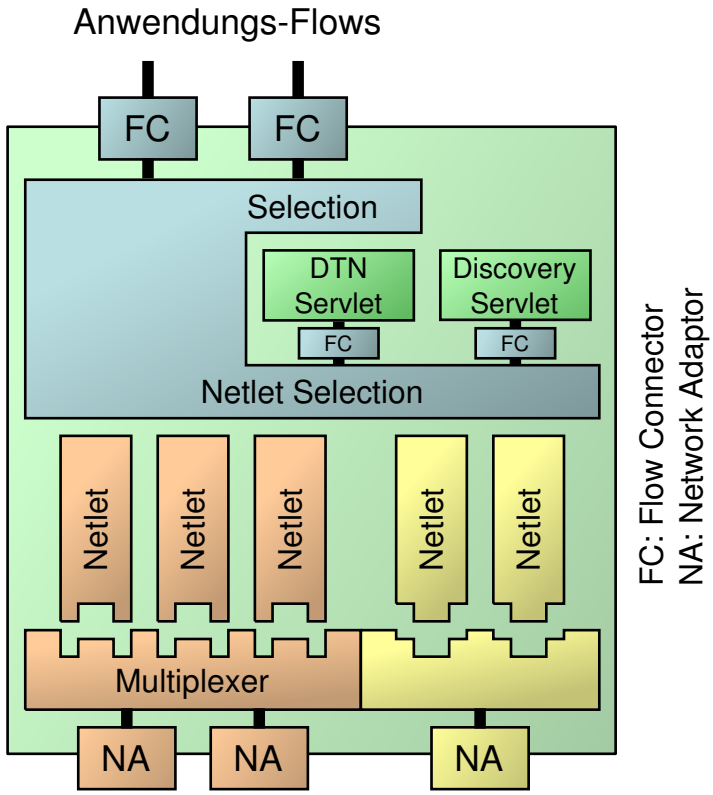


Abbildung 7.10 NENA mit DTN-Servlet und Discovery-Servlet

Servlet. Während das DTN-Servlet seine Dienste Anwendungen zur Verfügung stellt, realisiert das Discovery-Servlet ausschließlich einen Netzdienst, der unabhängig von Anwendungen agiert (siehe Abbildung 7.10). Wird eine Konnektivität zu einem neuen Netz hergestellt, sendet das Discovery-Servlet eine Ankündigung (HELLO) per Broadcast. Andere Systeme mit einem Discovery-Servlet, die diesen Broadcast empfangen, antworten daraufhin mit einer STILL-ALIVE-Nachricht. Systeme, die das Netzwerk verlassen, kündigen dies mit einer GOOD-BYE-Nachricht an. Zum Versenden der Nachrichten nutzt das Discovery-Servlet URIs der Form:

```
service://<Name des Systems>/discovery
```

Analog dazu bindet sich das Servlet auf eine entsprechende URI, um Nachrichten empfangen zu können.

Das DTN-Servlet ist ein weiterer Netzdienst, der sich ebenfalls auf eine URI bindet:

```
service://<Name des Systems>/dtn
```

Zusätzlich registriert sich das Servlet als Request Mapper bei der Netlet Selection Komponente, um bei der Kandidatenwahl für Anwendungsanfragen

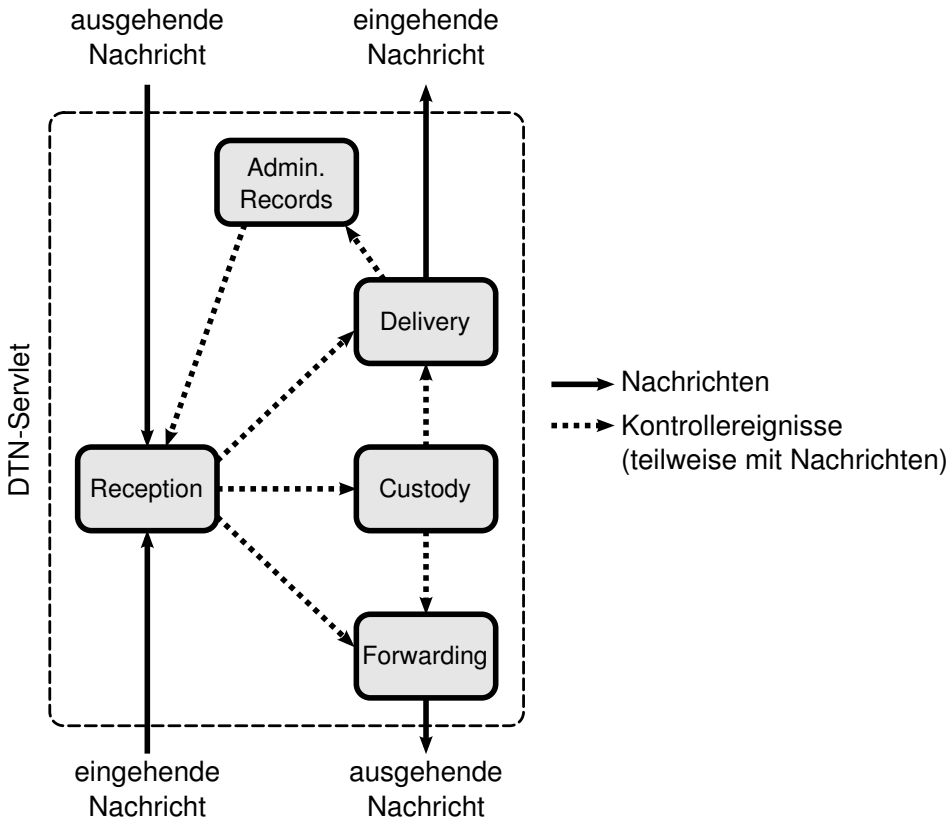


Abbildung 7.11 Aufbau des DTN-Servlet.

berücksichtigt zu werden. Das DTN-Servlet selbst ist aus verschiedenen Bausteinen aufgebaut, die in Abbildung 7.11 skizziert sind. Die Bausteine sind:

- *Reception.* Dieser Baustein übernimmt das Zusammenstellen des zu verschickenden Bundles für ausgehende Nachrichten. Eingehende Nachrichten leitet er je nach Nachrichtentyp an andere Bausteine weiter.

- *Delivery.* Eingehende Nutzdaten, die an eine lokale URI adressiert sind, werden vom Reception-Baustein an den Delivery-Baustein weitergegeben. Dieser leitet die Nachricht an die Anwendung weiter, die sich auf die URI gebunden hat. Ist aktuell keine Anwendung auf dieser URI gebunden, wird die Nachricht für eine spätere Auslieferung zwischengespeichert.

- *Custody.* Der Custody-Baustein ist für die persistente Speicherung von Bundles (z. B. auf der Festplatte) zuständig. Dies geschieht dann, wenn für dieses Bundle ein sog. Custody-Transfer angefordert wurde, der die Aufbewahrung des Bundles fordert, bis es entweder weiter Richtung Ziel übertragen werden kann oder bis die Lebenszeit des Bundles abgelaufen ist.

- *Forwarding.* Der Forwarding-Baustein ist für die Weiterleitung eines Bundles in Richtung des Ziels zuständig. Dieser muss über das Discovery-Servlet die Erreichbarkeit des nächsten Systems prüfen.

- *Administrative Records.* Dieser Baustein ist für das Senden von Kontrollnachrichten zuständig. Dazu gehören sog. *Status Reports* (für Statistiken) und *Custody Signals* (zur Signalisierung der Übernahme des Bundles bei Custody-Transfer).

Die Bausteine kommunizieren dabei ausschließlich über Ereignisse und direktem Nachrichtenaustausch. Durch die Trennung der Funktionalität können einige Strategien bspw. zur Zwischenspeicherung ausgetauscht werden.

7.3.6.2 Bewertung

Die Integration von DTN als Servlet und die Nutzung über die Anwendungsschnittstelle gestaltet sich ohne Probleme. Der Dienst ist jedoch nicht transparent zur Anwendung.

Nutzung der Anwendungsschnittstelle

- E_{P1} – Art und Weise der Nutzung

  Anwendungen, die Daten über den DTN-Dienst empfangen möchten, binden sich auf eine URI mittels BIND. Sendende Anwendung nutzen die PUT-Primitive. Andere Primitiven werden nicht unterstützt.

- E_{P2} – Annahmen über Anwendung

  Bei DTN wird angenommen, dass Anwendungen ihre zu versendenden Daten „gebündelt“ zur Verfügung stellen, da diese u. U. längere Zeit unterwegs sind. Damit diese beim Empfänger nutzbar sind, sollten die Bündel möglichst einzeln verwertbar sein.

  Weiterhin ist das DTN-Konzept grundlegend verschieden zu dem was heutige Anwendung erwarten, wenn sie einen Kommunikationsdienst anfordern. Bei DTN müssen sie mit langen Verzögerungen in der Übertragung umgehen können, und das DTN-Protokoll darf nicht ausgewählt werden, wenn dies nicht der Fall ist. Um dies sicherzustellen, muss die Anwendung explizit durch eine Anwendungsanforderung mitteilen, dass lange Verzögerungen in der Zustellung erlaubt sind.

- E_{P3} – benötigte Zusatzinformationen über die Anwendung

  Wie bei CCN ist es sinnvoll, wenn Anwendungen die Lebenszeit der Daten über die Anwendungsanforderungen angeben. Zudem erlaubt DTN die Zustellung von Übertragungsbestätigungen auch dann, wenn sich die Anwendung zwischen Versand der Nachricht und dem Empfang der

Bestätigung vom NENA-Rahmenwerk getrennt hat (bspw. wegen eines Neustarts des sendenden Systems). Um diese Zuordnung zwischen Bestätigung und neuverbundener Anwendung herstellen zu können, sind weitere Informationen von der Anwendung notwendig.

Integration in NENA

Mit den Servlets existiert das passende Konzept zur Umsetzung der beiden Dienste für DTN und Discovery.

- I_1 – Vollständigkeit (*Was fehlt?*)

  Bei der Umsetzen für NENA wurden keine Konzepte oder Abstraktionen vermisst.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Es mussten keine unnötigen Schnittstellen und Konzepte von NENA bei der Umsetzung berücksichtigt werden. Die Servlets wurden hier gleich für zwei netzwerkarchitekturübergreifende Dienste genutzt. Eine Realisierung ohne Servlets hätte die mehrfache Instantiierung eines DTN-Netlets in jeder Architektur erfordert, wobei sich alle DTN-Instanzen synchronisieren müssten.

- I_3 – Limitationen (*Was stört?*)

  Bei der Umsetzung waren keine Konzepte von NENA störend oder verursachten unverhältnismäßigen Mehraufwand.

7.3.7 Nachrichten-Broker: Publish/Subscribe

Message Queue Telemetry Transport (MQTT) [IBEu10] ist ein Protokoll, welches ein Publish/Subscribe-Paradigma mit einem Nachrichten-Broker als zentralem Element umsetzt. Ein Nachrichten-Broker registriert dazu alle Abonnenten (*Subscriber*) zu einem Thema (*topic*). Sobald eine Anwendung (der *Publisher*) eine Nachricht zu einem Thema veröffentlicht, werden alle Abonnenten informiert.

7.3.7.1 Aufbau

MQTT wurde als einfaches Protokoll für Maschine-zu-Maschine-Kommunikation entworfen und besteht aus verschiedenen Nachrichtentypen, die je nach angeforderten Dienst zur Nachrichtenzustellung verwendet werden. Unterstützt werden dabei die Dienstklassen:

- At most once (höchstens einmal): Die Nachricht wird höchstens einmalig zugestellt, deren Zustellung aber nicht garantiert. Dies ist die einfachste Form, die lediglich eine sog. Publish-Nachricht erfordert.

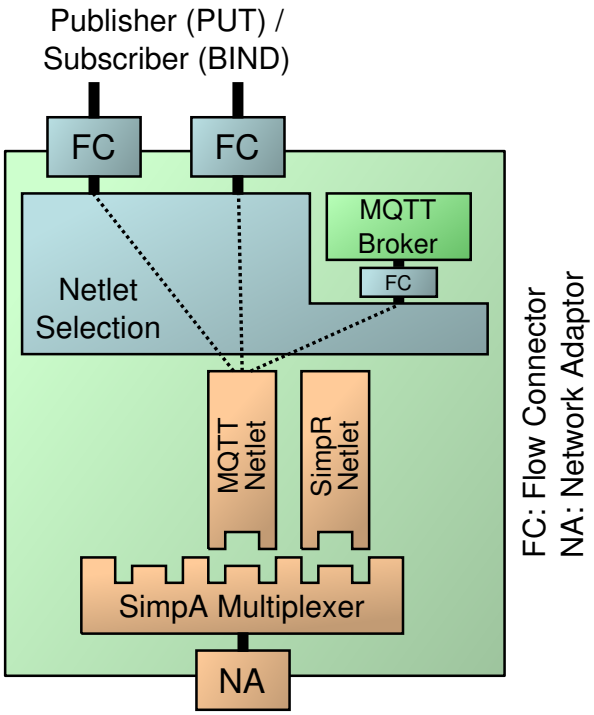


Abbildung 7.12 Realisierung des MQTT Protokolls und des Nachrichten-Brokers.

- At least once (mindestens einmal): Die Zustellung der Nachricht wird an den Publisher bestätigt, nachdem sie an alle Subscriber zugestellt wurde. Trifft keine Bestätigung nach einer gewissen Zeit ein, wird ein erneuter Zustellversuch unternommen. Bei Verlust der Bestätigung kann es so vorkommen, dass die Nachricht mehr als einmal veröffentlicht wird.
- Exactly once (genau einmal): Doppelte Nachrichten werden verhindert, indem zunächst mit einem 3-Wege-Handshake eine Verbindung zum Broker hergestellt wird und nach erfolgreicher Veröffentlichung der Nachricht an die Subscriber wieder abgebaut wird.

Der Broker verwaltet eine Liste mit existierenden Themen (topics). Pro Thema können sich beliebig viele Subscriber beim Broker registrieren. Zu veröffentlichende Nachrichten, die von Publishern an den Broker gesendet werden, werden daraufhin an die registrierten Subscriber verteilt.

Umsetzung für NENA

Die Realisierung des MQTT Protokolls und des Brokers ist in Abbildung 7.12 skizziert. Das Protokoll selbst wurde als Baustein in einem Netlet realisiert [Prat13]. Dieser übernimmt die Generierung der Kontrollnachrichten. Zusätzlich wurde der Nachrichten-Broker innerhalb von NENA als Dienst in einem Servlet realisiert. Der Broker nutzt dazu ein BIND auf eine Basis-URI:

```
broker://<Broker-Name>/topics/
```

Der Publisher nutzt die PUT-Primitive mit der URI des Themas zur Veröffentlichung einer neuer Nachricht:

```
topic://<Broker-Name>/topics/temperature
```

Der Subscriber bindet sich auf diese URI, um Nachrichten zum Thema zu empfangen. Die `topic://`-URI wird dabei vom MQTT-Protokollbaustein auf eine `broker://`-URI umgesetzt, und die Publish- bzw. Subscribe-Nachricht wird zum entsprechenden Broker gesendet. Zusätzlich kann die GET-Primitive genutzt werden, um die letzte Nachricht, die für ein Thema veröffentlicht wurde, anzufordern.

7.3.7.2 Bewertung

Wie das Discovery-Servlet des DTN-Dienstes (Abschnitt 7.3.6) kann der Broker auch hier als eigenständiges Servlet implementiert werden, sodass für den Broker keine eigene Anwendung erforderlich ist. Der Broker kann so auch Anfragen über verschiedene Architekturen und Protokolle verarbeiten, da er hier nicht an das Nachrichtenformat von MQTT gebunden ist.

Nutzung der Anwendungsschnittstelle

Durch MQTT wird ein Publish/Subscribe-Paradigma realisiert, welches aus Anwendungssicht bereits in Abschnitt 5.4.1.3 beschrieben wurde.

- E_{P1} – Art und Weise der Nutzung

  Die Pub/Sub-Primitive Publish wird über die PUT-API-Primitive realisiert, Subscribe über BIND. Zusätzlich kann ohne Abonnement mit einem GET die zuletzt veröffentlichte Nachricht zu einem Thema abgerufen werden.

  Der Nachrichten-Broker als Servlet nutzt die Primitiven auf andere Art und Weise: Über BIND kündigt er seine Dienste als Broker an. Empfange Anfragen werden je nach verwendeter Primitive des Anfragenden als Veröffentlichung, Abonnement oder einmaliges Abrufen der letzten Nachricht zu einem Thema interpretiert.

- E_{P2} – Annahmen über Anwendung

  Es wird angenommen, dass Anwendung, die entsprechende URIs verwenden, das Pub/Sub-Paradigma nutzen wollen, da sie mit einem BIND auf eine `topic://`-URI bspw. keine eigenen Dienste über MQTT anbieten können sondern nur veröffentlichte Nachrichten empfangen können. Nutzen Anwendung BIND allerdings im Zusammenhang mit einer `broker://`-URI, können sie auch selbst einen Broker-Dienst im Netz anbieten.

- E_{P3} – benötigte Zusatzinformationen über die Anwendung

  Die gewünschte Dienstklasse bei der Nachrichtenzustellung muss von der Anwendung als Anforderung mit angegeben werden.

Integration in NENA

Wie bei DTN bietet das Servlet-Konzept auch hier Vorteile, in diesem Fall zur Realisierung eines Nachrichten-Broker.

- I_1 – Vollständigkeit (*Was fehlt?*)

  Bei der Umsetzen für NENA wurden keine Konzepte oder Abstraktionen vermisst.

- I_2 – Notwendigkeit (*Was wird nicht benötigt?*)

  Es mussten keine unnötigen Schnittstellen und Konzepte von NENA bei der Umsetzung berücksichtigt werden.

- I_3 – Limitationen (*Was stört?*)

  Bei der Umsetzung waren keine Konzepte von NENA störend oder verursachten unverhältnismäßigen Mehraufwand.

7.4 Gesamtbewertung

In diesem Abschnitt werden die wesentlichen Erkenntnisse durch die Fallstudien, die in den vorherigen Abschnitten beschrieben wurden, zusammengefasst und abschließend beurteilt. Dabei wird zunächst auf die Umsetzung der Anwendungsschnittstelle eingegangen und anschließend auf die Integration der Protokolle und Architekturen in NENA als Module.

7.4.1 Anwendungsschnittstelle

Tabelle 7.2 fasst die Umsetzung der Anwendungsschnittstelle durch die untersuchten Protokolle und Architekturen zusammen. Die Primitiven der Anwendungsschnittstelle sind dabei von mehreren Protokollen jeweils bedienbar. Zu bemerken ist hierbei, dass die Primitive BIND je nach Anwendung und Protokoll unterschiedliche Bedeutungen hat:

- Im klassischen Fall der Bereitstellung eines Anwendungsdienstes wird durch den BIND-Aufruf die Bereitschaft signalisiert, Anfragen entfernter Anwendungen entgegenzunehmen und zu verarbeiten. Aus Protokollsicht ist dabei lediglich die Signalisierung der Anfrage über den Ereignis-Endpunkt mit entsprechender Bereitstellung der Metadaten erforderlich. Die gebundene Anwendung tritt also als Erbringer des Anwendungsdienstes auf. Ein Beispiel hierfür ist der Datei-Server (Abschnitt 7.2.4).

	E_{P1} Nutzung der API	E_{P2} Annahmen	E_{P3} zus. Inform.
Ende-zu-Ende	alle Primitiven	Ende-zu-Ende Kommunikation	Art des Transportdienstes
Video	CONNECT, BIND	Datenformat bekannt	keine
BitTorrent	GET, BIND	Anw. erwartet keinen kontinuierlichen Strom	keine
CCN	GET, PUT, (CONNECT, BIND)	keine	Lebenszeit der Daten
XIA	alle Primitiven	keine	Art des Transportdienstes
DTN	PUT, BIND	gebündelte Informationen, Umgang mit hohen Latenzen	Lebenszeit; Verzögerungstoleranz
MQTT	PUT, BIND, (GET)	Anw. erwartet einen Pub/Sub-Dienst	Dienstklasse

Tabelle 7.2 Protokolle bzw. Netzwerkarchitekturen und ihre Umsetzung der Anwendungsschnittstelle

- Im Falle von Pub/Sub wird BIND hingegen von Konsumenten eines Anwendungsdienstes genutzt: Sie signalisieren damit ihr Interesse an bestimmten Informationen. Beispiele hierfür sind die Chat-Anwendung (Abschnitt 7.2.2) und der Nachrichten-Broker (Abschnitt 7.3.7).

Dies zeigt, dass die Interpretation der tatsächlichen Bedeutung der Primitiven vom jeweiligen Anwendungsfall abhängt. Dies ist aber unproblematisch, da sich die beteiligten Anwendungen mit dem Namensraum der verwendeten URI bereits auf die Semantik des Kommunikationsparadigmas einigen. Somit sind auch Subjekte, Objekte und Prädikate der Kommunikation festgelegt (vgl. Chat-Anwendung in den Abschnitten 5.4.1.1 und 7.2.2).

Im Fall des inhaltsbasierten Kommunikationsparadigmas ist hier zu beobachten, dass eine solche Einigung auf die Semantik auch asymmetrisch sein kann (Abbildung 7.13): Während die Anwendung des Endbenutzers in diesem Beispiel ausschließlich eine inhaltsbasierte Sichtweise aufgrund der verwendeten Primitiven nutzt, kann der Inhalt sowohl über eine dienstbasierte Server-Anwendung (Abbildung oben) als auch über eine ebenfalls inhaltsbasierte Anwendung (Abbildung unten) bereitgestellt worden sein.

Einige Protokollfunktionalitäten wie bspw. zusätzliche Primitiven können durch die vorhandenen Primitiven nicht direkt abgebildet werden und müssen durch entsprechende Zusatzinformationen in den Anwendungsanforde-

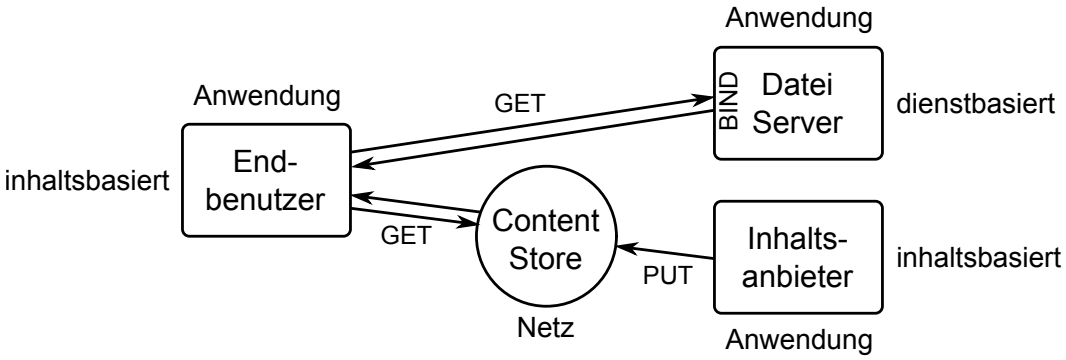


Abbildung 7.13 Symmetrische und asymmetrische Verwendung von Kommunikationsparadigmen.

rungen kodiert werden. Um bspw. lediglich Metadaten zu einer Ressource zu empfangen ohne die Ressource selbst zu übertragen kann dies durch die Angabe des Anforderungstupels `metadataonly: 1` der Protokollsoftware mitgeteilt werden. Für Primitiven, die sich auf diese Weise nicht intuitiv formulieren lassen, wird eine zusätzliche Abstraktionsebene über der hier beschriebenen Anwendungsschnittstelle empfohlen: Ähnlich zu den API-Schemata wie sie in FII [KSBF+11] vorgeschlagen werden erlaubt dies die Benutzung angepasster Anwendungsschnittstellen (vgl. Abschnitt 3.2.4).

Die wichtigste Beobachtung bei der Umsetzung der Anwendungsschnittstelle durch die Protokolle betrifft jedoch die Transparenz der verwendeten Protokolle zur Anwendung: Auch wenn bspw. die Anwendung basierend auf den unterstützten Primitiven ein inhaltsbasiertes Kommunikationsparadigma über DTN nutzen kann, muss die Anwendung Wissen darüber besitzen, dass der Transfer der Daten sehr hohe Latenzen aufweisen kann. Heutige Anwendung gehen bei der Nutzung der Socket-Schnittstelle von der impliziten Annahme aus, dass der Kommunikationspartner direkt erreichbar ist. Dies ist mit der hier vorgestellten Anwendungsschnittstelle nicht mehr zwangsläufig gegeben. Ähnliche Probleme treten beispielsweise bei BitTorrent auf: Hier kann es vorkommen, dass die Anwendung anfänglich keine Daten über die Anwendungsschnittstelle empfängt. Neben dem Kommunikationsparadigma, also den Subjekten, Objekten und Prädikaten, müssen zusätzliche Eigenschaften zum Kommunikationsverhalten mit angegeben werden, welche die Anwendung bei der Auswahl der unterstützten URI-Namensräume mit berücksichtigen muss.

Zusammenfassend lässt sich für die Anwendungsschnittstelle feststellen:

- Anwendungen müssen sich auf auf die unterstützten Kommunikationsparadigmen einigen. Diese müssen jedoch nicht zwangsweise symmetrisch sein.

Anwendung	inhaltsbasiert	dienstbasiert	Pub/Sub
Web-Browser	×		
Chat / IM	×	×	×
Video	×	×	
Datei-Server		×	
Gruppenkomm.	×		×
Arch./Protokoll			
Ende-zu-Ende	×	×	
BitTorrent	$\times^1$		
CCN	×		
XIA	×	×	×
DTN	$\times^2$		×
MQTT	×		×

$\times^1$, $\times^2$ – inhaltsbasiert mit zusätzlichen Eigenschaften

Tabelle 7.3 Überblick über die Kommunikationsparadigmen und die damit verwendeten Anwendungen und Architekturen bzw. Protokolle.

- In einigen Fällen kann es sinnvoll sein, eine zusätzliche Abstraktionsebene zwischen der hier vorgestellten Anwendungsschnittstelle und den Anwendungen einzuführen, um spezielle Primitiven zu unterstützen.
- Mit dem Namensraum der URI müssen neben dem Kommunikationsparadigma auch grundlegende Eigenschaften des Kommunikationsverhaltens beschrieben werden.

Während dies zusätzliche Spezifikationen bzgl. der URI-Namensschemata mit sich bringt, wird sich die Anzahl der grundlegend verschiedenen Namensschemata in Grenzen halten: Die hier vorgestellten Architekturen und Anwendungen decken mit der inhaltsbasierten, dienstbasierten und Pub/Sub-basierten Kommunikation wichtige Kommunikationsparadigmen ab. Tabelle 7.3 gibt einen Überblick über die Paradigmen und die damit verwendeten Anwendungen und Architekturen, die im Rahmen dieser Arbeit untersucht wurden. Bei der inhaltsbasierten Verwendung von BitTorrent und DTN sind dabei zusätzliche Eigenschaften wichtig für die Anwendung.

7.4.2 NENA-Rahmenwerk

Die Ergebnisse zur Bewertung der Integration der untersuchten Architekturen und Protokolle in NENA sind in Tabelle 7.4 basierend auf den festgelegten Kriterien zusammengefasst. Generell ist zu sehen, dass einige kommunikati-

	l_1 Vollständigkeit	l_2 Notwendigkeit	l_3 Limitationen	Besonderheiten
Ende-zu-Ende	✓	✓	keine	Namensauflösung im Multiplexer
Video	✓	✓	keine	Codec im Netlet realisiert
BitTorrent	✓	✓	sequentielle API	delegierte Übertragung vorteilhaft
CCN	✓	✓	keine	Multiplexer umfangreich
XIA	✓	✓	keine	keine
DTN	✓	✓	Anwendungskontext nach Neustart	mit Servlets realisiert
MQTT	✓	✓	keine	Broker als Servlet realisiert

Tabelle 7.4 Netzwerkarchitekturen bzw. Protokolle und ihre Umsetzung der Anwendungsschnittstelle.

onsspezifische Aufgaben nicht mehr durch die Anwendung erfolgen müssen, sondern in der Kommunikationssoftware realisiert werden können:

- Namensauflösung

  Die Namensauflösung kann relativ spät stattfinden, sodass sie auch erst im Multiplexer durchgeführt werden kann (*late binding*). Dies hat den klaren Vorteil, dass alle darüberliegenden Protokolle und Anwendungen unabhängig von konkreten Adressen realisiert werden können und die Adressauswahl nach netzspezifischen Kriterien erfolgen kann.

- Multimedia-Codecs, die für den Transport optimiert sind

  Mit Kenntnis über den Inhalt der zu übertragenden Informationen können Optimierungen zum Transport unabhängig von der Anwendung durchgeführt werden. Diese Optimierungen können dabei auf das entsprechende Kommunikationsnetz und die aktuelle Netzsituation angepasst sein. Bspw. kann so die Kodierung von Multimedia-Inhalten verändert werden, um eine höhere Kompression zu erhalten oder um besser mit Verlusten umgehen zu können.

- Peer-to-Peer-Protokolle (BitTorrent)

  Komplexe Anwendungsprotokolle können ebenfalls als Netlet realisiert werden, was auch hier Optimierungen in den jeweiligen Netzen ermöglicht. Gerade für Peer-to-Peer-Datenverkehr können so Optimierungen der Netzbetreiber verwendet werden, die zu einer effizienterer Ressourcen-Nutzung führen.

- Zwischenspeichern von Inhalten (CCN, DTN)

  Die Zwischenspeicherung von Inhalten kann in bestimmten Fällen nutzbringend sein, wird aber bisher nur auf Anwendungsschicht realisiert. Längeres Zwischenspeichern als grundlegende Netzfunktionalität bringt in manchen Fällen jedoch weitere Vorteile wie eine höhere Zustellrate (DTN) oder eine schnellere Bedienung von Anfragen (CCN). Hier wurde gezeigt, dass NENA auch für solche Ansätze geeignet ist.

- Netzwerkarchitekturübergreifende Dienste (DTN, Nachrichten-Broker)

  Einige Netzdienste sind heute als Anwendung realisiert. Mit DTN und einem Nachrichten-Broker wurde gezeigt, dass solche Dienste jedoch auch in NENA als Servlet realisiert werden können. Da diese Dienste auch ereignisgesteuert sind, hat sich NENA als geeignete Umgebung dafür erwiesen. Andere Netzdienste, wie bspw. DNS, eignen sich daher ebenfalls zur Realisierung als Servlet.

Während es eines der Hauptziele der NENA-Konzepte war, möglichst viel Netzfunktionalität innerhalb der NENA-Module zu realisieren, wurde angenommen, dass ein Großteil dieser Aufgaben in Netlets und Servlets realisiert werden würde, und der Multiplexer lediglich eine „dünne“ Basisschicht darstellt. Bei der Realisierung von CCN ist allerdings zu beobachten, dass ein Großteil der CCN-Funktionalität tatsächlich im Multiplexer realisiert wurde. Dies lässt sich mit der Architektur von CCN selbst begründen: Hier stehen nicht die Anwendungs-Flows oder Transportverbindungen im Vordergrund, sondern die Inhalte – unabhängig von den Anfragen der Anwendung. Die Funktionalität, die im Multiplexer realisiert ist, behandelt lokale Anwendungsanfragen und Anfragen anderer Knoten gleich. Spezifische Funktionalität für eine lokale Anwendungsanfrage besteht dabei lediglich aus dem Generieren von Interest-Nachrichten, was innerhalb von Netlets umgesetzt wurde. Trotzdem hat sich dies nicht als Nachteil herausgestellt und es konnte gezeigt werden, dass die NENA-Konzepte ausreichend flexibel sind. Generell kann also der Umfang der einzelnen Module (Netlets, Servlets, Multiplexer) so gewählt werden, wie es für die jeweilige Architektur oder das jeweilige Protokolle am geeignetsten ist.

Bzgl. der Integrationskriterien lässt sich feststellen, dass bei der Integration der jeweiligen Architekturen und Protokolle keine Konzepte im NENA-

Rahmenwerk vermisst wurden (I_1) oder sich als unnötig herausgestellt haben (I_2). Jedoch konnten zwei Einschränkungen (I_3) beobachtet werden, die sich allerdings durch die Anwendungsschnittstelle begründen:

- Sequentielle API

  Der direkte Datenaustausch zwischen Anwendung und Rahmenwerk ist nur sequentiell möglich. Dadurch können Nachteile bei der Verwendung von blockorientierten Übertragungen wie bei BitTorrent entstehen, da zum einen das Rahmenwerk die nicht in Reihenfolge empfangenen Datenblöcke zwischenspeichern muss, zum anderen die Anwendung erst sehr spät die ersten Daten vom Ressourcen-Endpunkt lesen kann. Die delegierte Datenübertragung, wie sie in Abschnitt 5.3.5 beschrieben wird, bietet hier jedoch eine elegante Lösung an.

- Anwendungskontext nach Neustart

  DTN bietet Anwendungen die Möglichkeit, über die Zustellung eines Bundels auch nach einem Neustart der sendenden Anwendung oder des Systems informiert zu werden. Der notwendige Kontext kann in diesem Fall nicht über das Flow State Objekt hergestellt werden, da dieses nur für eine laufende Interaktion zwischen Anwendung und Rahmenwerk erstellt wird. In diesem Fall müssen bspw. Sitzungsschlüssel vereinbart werden (vgl. Abschnitt 5.4.1.1).

Aufgrund der Verschiedenheit der durchgeführten Fallstudien, die jeweils durch konkrete Implementierungen überprüft wurden, kann gefolgert werden, dass NENA geeignete Konzepte zur flexiblen Realisierung von Protokollen und Netzwerkarchitekturen bereitstellt. Gepaart mit der Möglichkeit, weitere Funktionalitäten bei Bedarf nachzuladen, bietet das NENA-Rahmenwerk mit dessen Anwendungsschnittstelle eine mächtige Umgebung für heutige und viele zukünftige Protokolle und Netzwerkarchitekturen. Anwendungen können dadurch deutlich einfacher gehalten werden und profitieren von neuen Protokollen, die von Netzwerkexperten entworfen und implementiert wurden.

7.4.3 Zusammenfassung der Invarianten

Trotz des Ziels der NENA-Konzepte, möglichst wenige Vorgaben zu machen, sind einige explizite und implizite Invarianten zu berücksichtigen, wenn eine neue Architektur oder ein Protokoll für NENA realisiert werden soll. Diese stellen jedoch keine Einschränkungen dar, sondern bieten den Rahmen zur Verwendung neuer Architekturen und Protokolle durch existierende Anwendungen. Diese Invarianten werden im Folgenden noch einmal abschließenden zusammengefasst.

Explizite Invarianten

URIs und deren Namensräume. Wenn eine Anwendung einen Kommunikationswunsch äußert, hat sie gewisse Erwartungen an die Ressource, die sie anfordert. Die Anwendung muss wissen, ob es sich bei der Ressource um ein Inhaltsobjekt oder einen Dienst handelt. Zudem muss sie wissen, in welchem Format der Inhalt vorliegt, oder wie der Dienst anzusprechen ist. Aus diesem Grund spielen die URI-Namensräume eine essentielle Rolle für eine namensbasierte API. Daher sollten diese Namensräume standardisiert werden, damit so die Struktur und die Semantik der Namen definiert werden.

API-Primitiven. Die hier verwendeten API-Primitiven GET, PUT, CONNECT und BIND decken einen wichtigen Teil der Kommunikationsparadigmen ab, und es lassen sich inhaltsbasierte, dienstbasierte und Pub/Sub-basierte Paradigmen verwenden. In einigen Fällen können zusätzliche Primitiven jedoch sinnvoll sein, um höherwertige Abstraktionen zu erzielen und so die Benutzung durch den Anwendungsprogrammierer weiter zu vereinfachen. Diese können jedoch überhalb der hier beschriebenen API realisiert werden (vgl. Abschnitt 7.4.1).

Anwendungsanforderungen. Die Anwendungsanforderungen sind ein wichtiger Bestandteil der API, da hierüber die jeweils gewünschte Aktion detaillierter beschrieben werden kann. So kann bspw. angegeben werden, ob nur Meta-Informationen zu einem Namen angefordert werden sollen. Anforderungen müssen jedoch ebenfalls zusammen mit den URI-Namensräumen standardisiert werden. Dies hat den Vorteil, dass mit jedem URI-Namensraum auch eine neue Menge an Anwendungsanforderungen festgelegt werden kann, wodurch die API auch in Zukunft erweiterbar bleibt.

NENA-spezifische Schnittstellen. Eine Architektur oder ein Protokoll muss NENA-spezifische Schnittstellen zur Kandidatenabfrage (Request Mapper) und zur Registrierung von Anwendungsdiensten implementieren. Zusätzlich müssen sie empfangenen Paketen die von NENA zentral verwalteten Zustandsobjekte zuordnen.

Module: Netlets, Multiplexer und Servlets. Protokolle oder ganze Architekturen müssen ihre Funktionalitäten auf diese drei Module aufteilen. Diese Aufteilung hat sich zur besseren Strukturierung bewährt: Der Multiplexer als Basisprotokollschicht übernimmt generelle Aufgaben der jeweiligen Architektur und ist zwingend erforderlich. Netlets bedienen Anwendungsanfragen oder enthalten Kontrollprotokolle für die Architektur (wie z. B. Routing-Protokolle). Servlets dienen der Realisierung von Netzdiensten oder Diensten für Anwendungen, die netzwerkarchitekturübergreifend angeboten werden sollen. Als Netzdienst ist hier bspw. DNS denkbar. Beispiele für netzwerkarchitekturübergreifende Dienste für Anwendungen sind DTN (Abschnitt 7.3.6) und der Nachrichten-Broker (Abschnitt 7.3.7).

Implizite Invarianten

Die hier beschriebenen impliziten Invarianten haben sich durch die durchgeführten Fallstudien ergeben und besitzen praktische Relevanz für die Umsetzung in NENA. Sie ergänzen die expliziten Invarianten, führen aber zu keinen zusätzlichen Einschränkungen.

Flusskontrolle. Flusskontrolle ist in der Regel Teil des jeweils verwendeten Transportprotokolls, sofern dieses einen entsprechenden Dienst bietet. Der Mechanismus, mit dem die Rate gesteuert wird, mit der die Anwendung ihre Daten an das Rahmenwerk weitergibt, ist allerdings Teil des Rahmenwerks. Es muss also eine Schnittstelle geben, mit der das Transportprotokoll dem Rahmenwerk signalisieren kann, dass keine neuen Daten mehr von der Anwendung entgegengenommen werden sollen (vgl. Abschnitt 6.3).

Flows. Zur Identifizierung verschiedener Anwendungs-Flows wird jedem Zustandsobjekt von NENA eine interne ID zugeordnet, über die das Zustandsobjekt wieder referenziert werden kann (bspw. um das Objekt empfangenen Nachrichten zuordnen zu können). Diese ID hat nur lokale Bedeutung und die jeweilige Netzwerkarchitektur muss eigene Mechanismen verwenden, um eine Zuordnung zwischen Paketen und Zustandsobjekt zu ermöglichen. Sie sollte diese ID also insbesondere nicht im Paketkopf ihrer eigenen Pakete verwenden, um zu vermeiden, dass NENA-spezifische Elemente in die Netzwerkarchitektur einfließen.

7.5 Laufzeitverhalten

Während in den letzten Abschnitten die architekturellen Aspekte von NENA und der Anwendungsschnittstelle mittels Fallstudien näher betrachtet wurden, wird in diesem Abschnitt das Laufzeitverhalten des komponentenbasierten Ansatzes aus Abschnitt 4.3 in NENA untersucht. Dazu wird in Abschnitt 7.5.1 zunächst die entworfene Transportprotokollschablone mit der TCP-Implementierung eines Simulators verglichen, um zu zeigen, dass ein identisches Verhalten reproduzierbar ist. Anschließend werden in Abschnitt 7.5.2 die Kosten der Komposition im NENA-Prototyp auf einem realen System ermittelt.

7.5.1 Vergleich mit TCP in OMNeT++

Die Transportprotokollschablone aus Abschnitt 4.3.2 in der NewReno-Variante wurde für NENA implementiert und mit der TCP-NewReno-Implementierung des INET-Frameworks für OMNeT++ [Varg01] verglichen [Fran12]. OMNeT++ ist ein diskreter Ereignissimulator, für den NENA wie in Abschnitt 6.7.4 beschrieben angepasst wurde, sodass NENA-Instanzen als simulierte Netzwerkknoten verwendet werden können. OMNeT++ wurde dabei

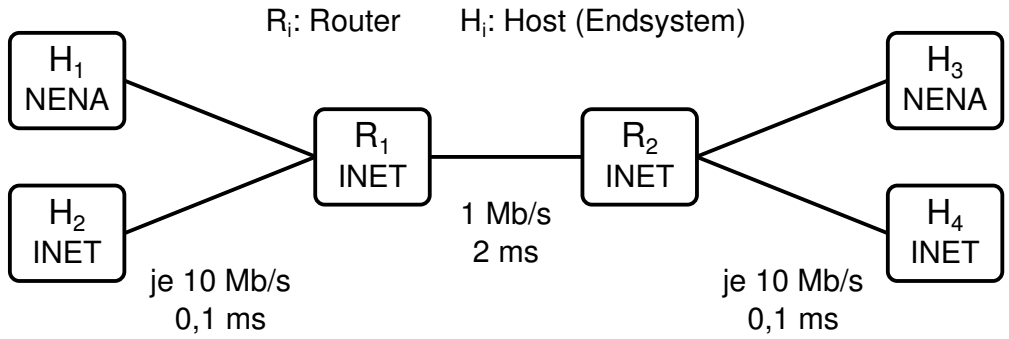
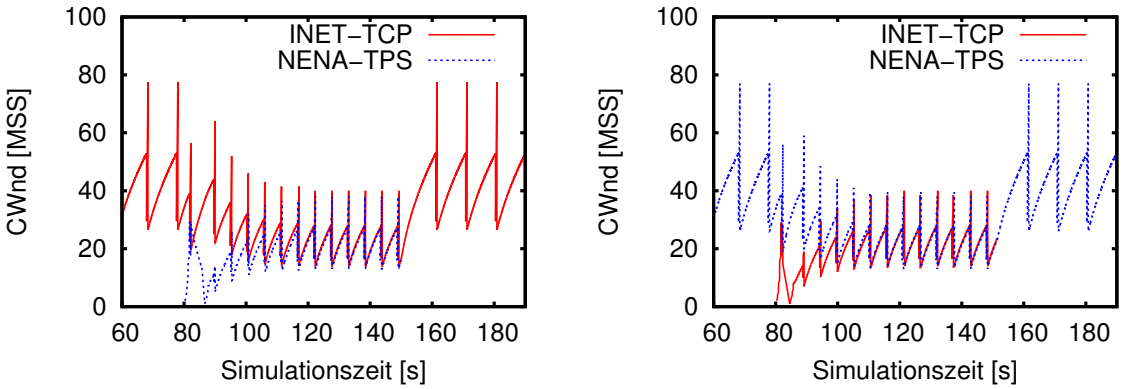


Abbildung 7.14 Topologie für den Vergleich der Transportprotokollschablone mit der TCP-Implementierung in INET für OMNeT++.

in der Version 4.2.2, das INET-Framework in der Version INET-20111118 verwendet.

Abbildung 7.14 zeigt die verwendete Topologie: Die Endsysteme H_1 und H_3 bestehen aus NENA-Instanzen, die je eine Instanz der Transportprotokollschablone enthalten und miteinander kommunizieren. Die Transprotokollschablone ist dabei innerhalb eines Netlets für die SimpA (s. Abschnitt 6.5) realisiert, was bedeutet, dass auch der SimpA-Multiplexer verwendet wird. Die Endsysteme H_2 und H_4 bestehen entsprechend aus INET-TCP-Instanzen. Als Netzwerkprotokoll verwenden alle Systeme INET-IPv4, wobei die NENA-Instanzen dies lediglich zum Tunneln der Nachrichten benutzen (vgl. UDP Network Adaptors in Abschnitt 6.7.2). Die Verbindung zwischen den beiden Routern R_1 und R_2 dient hier als Engpass, um den beide Transportverbindungen konkurrieren: Während die Verbindung zwischen den beiden Routern eine Bandbreite von 1 Mb/s und eine Verzögerung von 2 ms aufweist, sind die Endsysteme mit jeweils 10 Mb/s und einer Verzögerung von 0,1 ms an die jeweiligen Router angebunden. Die Größe der Warteschlangen in den Routern beträgt jeweils 50 Pakete. Ist die Warteschlange voll, werden Folgepakete verworfen (*drop tail*). Dies stellt die einzige Quelle von Paketverlusten in diesem simulierten Netzwerk dar.

Es werden zwei Szenarien getestet: Zunächst muss die Transportprotokollschablone (im Folgenden kurz TPS) mit einer bestehenden INET-TCP-Verbindung konkurrieren. Anschließend muss die INET-TCP-Verbindung mit einer bestehenden TPS-Verbindung konkurrieren. Dazu fangen die nachträglich startenden Verbindungen jeweils bei 80 s Simulationszeit an zu senden und hören bei 150 s Simulationszeit wieder auf. In allen Fällen sendet die Anwendung mit der maximal möglichen Datenrate, die über die Transportverbindung erreichbar ist. Zum Vergleich wird hier die Größe des aktuellen Staukontrollfensters CWnd, gemessen in Anzahl der MSS (Maximum Segment Size, vgl. Abschnitt 2.2.2.2), herangezogen, dessen Verlauf für beide Szenarien



(a) Bestehende INET-TCP Verbindung mit kurzzeitiger NENA-TPS Verbindung.

(b) Bestehende NENA-TPS Verbindung mit kurzzeitiger INET-TCP Verbindung.

Abbildung 7.15 Verlauf der Staukontrollfenster (CWnd) der Transportprotokollschablone (NENA-TPS) in H_1 und von INET-TCP in H_2 über die Simulationszeit.

in Abbildung 7.15 dargestellt ist. Die sendenden Knoten sind in beiden Fällen H_1 und H_2.

Für die laufenden Verbindungen sind jeweils die Congestion Avoidance Phase und die Fast Recovery Phase zu erkennen. Die Congestion Avoidance Phase führt zu einer langsamen Erhöhung des CWnd bis es zu einem Paketverlust kommt, der über drei dupliziert ACKs erkannt wird. In der Fast Recovery Phase wird daraufhin

$$\mathsf{CWnd} = \mathsf{CWnd}_{\text{alt}}/2 + 3\ \text{MSS}$$

gesetzt und anschließend für jedes weitere ACK um 1 MSS erhöht. Da hier jeweils nur ein Paket der gesamten ausstehenden Pakete verloren gegangen ist, führt dies dazu, dass für alle ausstehenden Pakete (abzüglich des verlorenen Paketes) das CWnd um 1 MSS erhöht wird und am Ende der Fast Recovery Phase auf

$$\mathsf{CWnd}_{\text{alt}}/2 + (\mathsf{CWnd}_{\text{alt}} - 1\ \text{MSS})$$

steht. In Abbildung 7.15 ist dies durch den starken Anstieg des CWnd nach einem Paketverlust zu erkennen. Wird die Fast Recovery Phase wieder verlassen und in die Congestion Avoidance Phase übergegangen, wird das CWnd wieder auf ssthresh ($= \mathsf{CWnd}_{\text{alt}}/2$) gesetzt. Für die kurzzeitigen Verbindungen sind jeweils die Slow Start Phase mit anschließender Congestion Avoidance und Fast Recovery Phase zu beobachten. Nach ca. 20 s gleichen sich die Staukontrollfenster der beiden Verbindungen an, was auf eine faire Aufteilung der Engpassbandbreite schließen lässt.

Insgesamt ist zu erkennen, dass der Verlauf der Staukontrollfenster in beiden Fällen nahezu identisch ist, unabhängig davon ob die Transportprotokollschablone oder INET-TCP um eine bestehende Verbindung konkurrieren muss. Dies zeigt, dass die Funktionsweise von TCP-NewReno mit einer ereignisorientieren Implementierung realisierbar ist und die Protokollschablone prinzipiell für ein zuverlässiges Transportprotokoll geeignet ist.

7.5.2 Kosten der Ereignisverarbeitung

Während im letzten Abschnitt gezeigt wurde, dass der Schablonenansatz aus Abschnitt 4.3 und insbesondere die dort entworfene Transportprotokollschablone dazu geeignet sind, um die Funktionsweise von TCP in NENA zu implementieren, wird in diesem Abschnitt untersucht, welche Kosten für die Ereignisverarbeitung im NENA-Prototyp mit der Transportprotokollschablone entstehen. Da bei der Implementierung der Schablone und des NENA-Prototyps die Machbarkeit im Vordergrund stand und nicht die absolute Leistungsfähigkeit, wird ein relativer Vergleich der Komponenten untereinander vorgenommen. Hierbei sollen neben den Bausteinen der Transportprotokollschablone auch andere NENA-Komponenten mit berücksichtigt werden. Insbesondere der Aufwand im Message Scheduler soll hier als Kriterium dienen, da dieser keine direkte Funktionalität zur Netzwerkkommunikation bereitstellt und als reiner Mehraufwand zur Realisierung des ereignisorientierten Systems in NENA (vgl. Abschnitt 6.4) betrachtet werden kann.

Als Ausgangspunkt dient dieselbe Implementierung der Transportprotokollschablone in der NewReno-Variante, die im letzten Abschnitt bei der Untersuchung in OMNeT++ verwendet wurde. Diese ist innerhalb eines Netlets (Template Netlet) gekapselt und bietet ihren Transportdienst als Teil der SimpA über den SimpA Multiplexer an. Für die folgenden Messungen wurde dazu der in Abbildung 7.16 dargestellte Aufbau verwendet: Auf demselben Rechner wurden zwei NENA-Instanzen gestartet, die über lokale UDP-Sockets (also über localhost) miteinander kommunizieren können. Geladen wurden die abgebildeten Module: der SimpA Multiplexer, das SimpA Routing Netlet und das Netlet mit der Transportprotokollschablone (Template Netlet).

Als Testanwendung wurde eine C++-Anwendung geschrieben, die über die tmnet-API-Bibliothek (Abschnitt 5.5) auf eine NENA-Instanz zugreifen kann und somit eine Implementierung der in Kapitel 5 beschriebenen Anwendungsschnittstelle verwendet. Diese Anwendung wird einmal als Server über NENA-Instanz 2 (im Folgenden Empfänger-NENA-Instanz) und einmal als Client über NENA-Instanz 1 (im Folgenden Sender-NENA-Instanz) gestartet. Als Client sendet die Anwendung 512 MiB als einen kontinuierlichen Datenstrom mit maximaler Datenrate, limitiert durch die Geschwindigkeit, mit

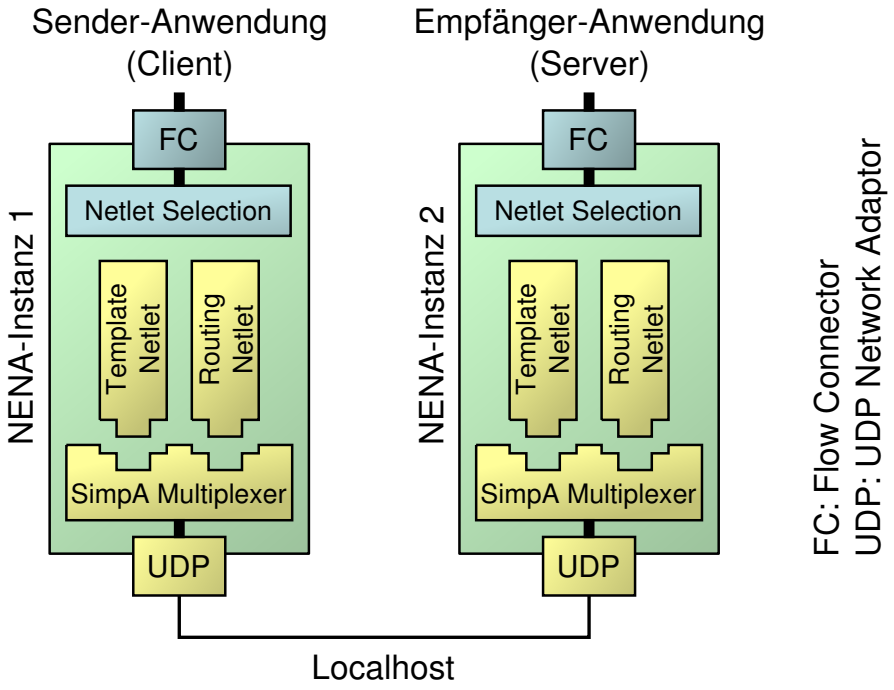


Abbildung 7.16 Aufbau des Testszenarios auf einem Rechner zur Messung der Ausführungszeiten der einzelnen Komponenten und Module in NENA.

der die NENA-Instanz die Daten entgegennimmt. Dies resultierte in 500.000 Datenpakete mit je 1024 Byte an Anwendungsdaten.

Als Test-Umgebung wurde ein Rechner mit folgender Spezifikation genutzt:

- Prozessor: Intel® Core™ 2 Duo CPU (P8700) mit 2,53 GHz
- Arbeitsspeicher: 8 GB
- Betriebssystem: Xubuntu 12.04 LTS (Precise Pangolin) mit Linux Kernel 3.2.0

Gemessen wurde die reine Ausführungszeit der involvierten Message Processors (MPs) in NENA (also aller Komponenten, Module und Bausteine). Die Ausführungszeit eines MP beinhaltet dabei alle Aktionen, die der MP zur Verarbeitung eines Ereignisses durchführt, inkl. Ein-/Ausgabe-Aktionen. Eine Ausnahme bildet hier der UDP Network Adaptor, bei dem der Empfang der Daten asynchron verläuft und somit nicht Bestandteil der Messungen ist. Der Versand erfolgt jedoch synchron und ist damit Teil der Messungen.

Messfunktionen wurden ausschließlich im Message Scheduler verwendet. Für jedes Ereignisse ruft der Message Scheduler den jeweils zuständigen MP auf, der das Ereignis verarbeitet und ggfs. neue Ereignisse erzeugt, für

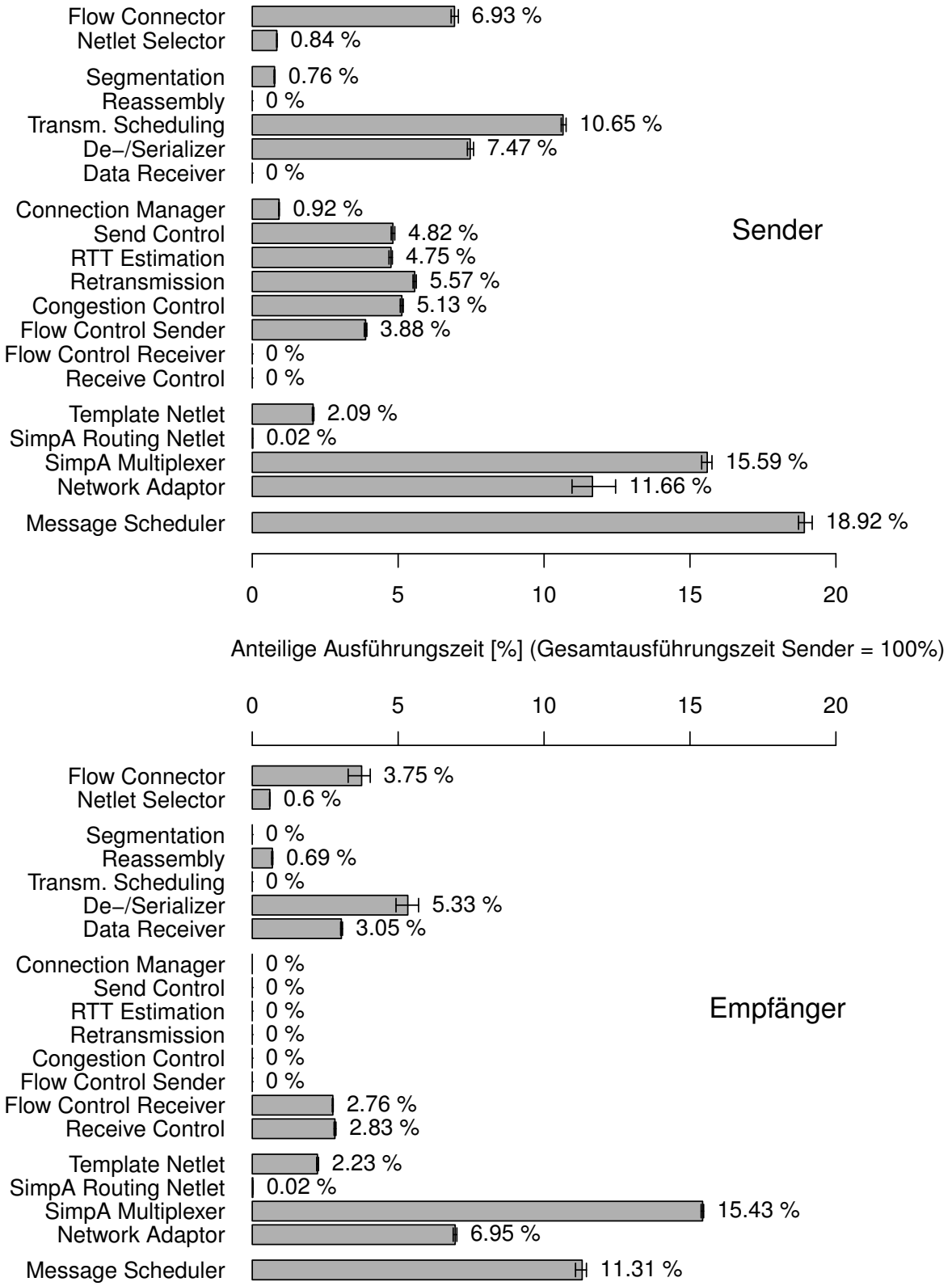


Abbildung 7.17 Anteilige Ausführungszeiten der einzelnen Message Processors in NENA, unterteilt in Sender (oben) und Empfänger (unten). Als Transportprotokoll wurde die NewReno-Variante der Transportprotokollschablone über die SimpA verwendet.

die in der nächsten Iteration der Message Scheduler wieder den zuständigen MP aufruft. Zur Messung wurde ein hochauflösender Zeitnehmer der boost::chrono-Bibliothek (chrono::high_resolution_clock) verwendet, der mittlererweile auch Teil des aktuellen C++11 Standards ist. Auf dem Testsystem hatte dieser eine Auflösung[1] von 1 ns. Betrachtet wurden dazu alle beteiligten MPs, die in Abbildung 7.17 dargestellt sind. Für den Message Scheduler wurde die Ausführungszeit ebenfalls bestimmt. Hier wurde die Zeit gemessen, in der der Message Scheduler gerade keinen MP bedient und nicht gerade schläft, weil keine Nachrichten anstehen. Basis der in der Abbildung dargestellten Messungen sind 10 Testläufe, deren Messergebnisse gemittelt wurden (arithmetisches Mittel) und von denen die minimalen und maximalen Werte als Fehlerbalken eingezeichnet sind. Aufgrund der geringen Abweichungen wurde auf weitere Testläufe verzichtet. Durch die hohe Anzahl an verarbeiteten Ereignissen pro Testlauf lagen zudem ausreichend Messwerte vor. Die Gesamtanzahl der Ereignisse pro Testlauf betrug durchschnittlich:

- 16.500.890 Ereignisse in der Sender-NENA-Instanz ($\pm$ 22 Ereignisse)
- 9.000.887 Ereignisse in der Empfänger-NENA-Instanz ($\pm$ 21 Ereignisse)

Die Ereignisse enthalten dabei alle Nachrichten, Kontrollereignisse und Zeitgeberereignisse für die einzelnen MPs.

In Abbildung 7.17 sind die anteiligen Ausführungszeiten der einzelnen MPs als Balkendiagramme dargestellt, unterteilt in Sender- und Empfänger-NENA-Instanz. Als Basis der Anteile wurde für Sender- und Empfänger-Instanz die Gesamtausführungszeit des Senders herangezogen, um eine direkte Vergleichbarkeit zwischen Sender- und Empfänger-MPs zu erreichen. Die Summe der Ausführungszeiten auf der Sender-NENA-Instanz betrug durchschnittlich ca. 179,6 s (= 100 % Gesamtausführungszeit) und in der Empfänger-NENA-Instanz ca. 98,7 s (= 54,97 % der Ausführungszeit der Sender-NENA-Instanz). Die NENA-Instanzen wurden dazu kurz vorher gestartet und nach der Übertragung sofort beendet. Die nicht von der Übertragung verursachten Ereignisse werden hauptsächlich durch das Routing Netlet verursacht und können vernachlässigt werden (vgl. anteilige Ausführungszeit des Routing Netlets).

Der Aufwand des Message Schedulers ist am unteren Ende der Balkendiagramme dargestellt. Beim Sender verursacht der Message Scheduler knapp ein fünftel der gesamten Ausführungszeit und damit den größten anteiligen Aufwand. Beim Empfänger ist der anteilige Aufwand mit ca. 11 % etwas niedriger.

[1] Die Genauigkeit unterliegt dabei Betriebssystem-üblichen Schwankungen und liegt im Mikrosekundenbereich. Für die durchgeführten Messungen ist dies jedoch ausreichend.

Die einzelnen Bausteine des Template Netlets auf Senderseite verursachen in der Summe ca. 44 % der Gesamtausführungszeit, wobei ca. 19 % der Gesamtausführungszeit auf den Datenpfad und ca. 25 % auf den Kontrollbereich entfallen. Auf Empfängerseite sieht das Verhältnis etwas anders aus, da hier ca. 9 % auf den Datenpfad und nur knapp 6 % auf den Kontrollbereich entfallen. In der Summe beansprucht das Template Netlet auf Empfängerseite also knapp 15 % der Gesamtausführungszeit des Senders. Diese Verhältnisse, sowohl zwischen Sender und Empfänger als auch zwischen Datenpfad und Kontrollbereich lassen sich sehr gut an der Struktur der Transportprotokollschablone nachvollziehen. Der Kontrollbereich auf Senderseite hat hier bspw. die größte Komplexität.

Die Ausführungszeiten der Interprozesskommunikation mit der Anwendung (enthalten im Flow Connector) und die Nutzung des Protokollstapels des Betriebssystems, von dem ein UDP-Socket als Network Adaptor genutzt wurde, betragen insgesamt knapp 19 % auf Senderseite und knapp 11 % auf Empfängerseite. Hier ist jedoch zu berücksichtigen, dass der Empfang der UDP-Pakete asynchron erfolgt und nicht Teil der Messungen ist (s. o.). Sonstige Verwaltungsaufgaben, die in NENA bei der Kommunikation entstehen, sind im Netlet Selector zu finden, der hier nach dem Verbindungsaufbau die Nachrichten lediglich zwischen Flow Connector und Netlet vermittelt (jeweils weniger als 1 % auf Sender- und Empfängerseite). Die Ausführungszeit des Template Netlet (ohne dessen Bausteine) kann ebenfalls als Verwaltungsaufwand aufgefasst werden, da dieses die Nachrichten zwischen Multiplexer bzw. Netlet Selector und dem jeweiligen ersten Baustein auf dem Datenpfad vermittelt (jeweils ca. 2 % auf Sender- und Empfängerseite).

Einen großen Anteil an der Gesamtausführungszeit nimmt der Multiplexer mit jeweils ca. 15,5 % auf Sender- und Empfängerseite ein. Dies ist zum einen damit zu begründen, dass er Informationen als Paketkopf jeder ausgehenden Nachricht hinzufügt und von jeder eingehenden Nachricht wieder entfernt. Zum anderen aber auch damit, dass er für die Namensauflösung für ausgehende Nachrichten verantwortlich ist und für eingehende Nachrichten das Flow State Objekt und das zuständige Netlet bestimmt. Der Aufwand ist bei Sender und Empfänger in diesem Fall nahezu identisch. Dies ist im wesentlichen der Implementierung des Multiplexers geschuldet, der hier keine weiteren Optimierungen vornimmt und alle zu versendenden Nachrichten gleich behandelt (also insbesondere auch für ausgehende Bestätigungsnachrichten im Empfänger eine Namensauflösung durchführt). Eine wichtige Optimierung wäre bspw. die Ablage der Adressinformationen beider Kommunikati-

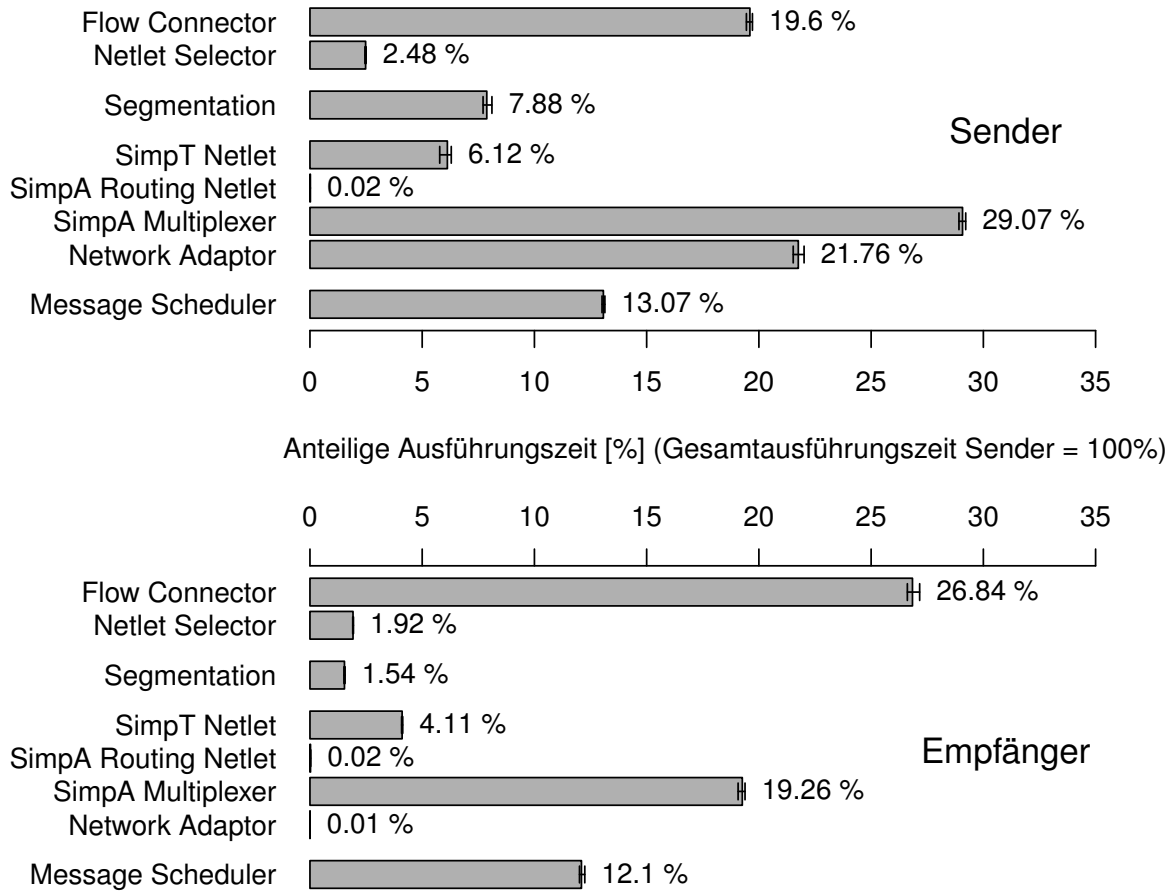


Abbildung 7.18 Anteilige Ausführungszeiten der einzelnen Message Processors in NENA, unterteilt in Sender (oben) und Empfänger (unten). Als Transportprotokoll wurde das SimpT Netlet verwendet.

onspartner im Zustandsobjekt, sodass die Namensauflösung[1] nicht bei jedem zu versendenden Paket erfolgen muss.

Abbildung 7.18 zeigt eine Vergleichsuntersuchung unter Verwendung des SimpT-Netlets, welches einen unzuverlässigen Transportdienst bereitstellt und lediglich einen Segmentierungsbaustein enthält (s. Abschnitt 6.5.2.2). Die Messungen fanden dabei unter den gleichen Bedingungen statt wie beim zuvor untersuchten Template Netlet. Die durchschnittliche Gesamtausführungszeit des Senders lag bei ca. 55,7 s (= 100 % der Gesamtausführungszeit) und beim Empfänger bei ca. 36,6 s (= 65,7 % der Ausführungszeit des Senders). Die durchschnittliche Gesamtanzahl der Ereignisse betrug

- 3.500.210 auf Senderseite ($\pm$ 16 Ereignisse)

[1] Die Namensauflösung in der SimpA ist ein Look-up in einer statischen Hash-Map, bei dem die Knotennamen als Schlüssel verwendet werden. Der aufwändige Teil hierbei ist die Bestimmung eines Hashwertes für die Zeichenkette des Knotennamens.

- 3.000.209 auf Empfängerseite ($\pm$ 14 Ereignisse)

Der Unterschied in Höhe der Anzahl der versendeten Pakete (500.000) lässt sich damit begründen, dass der Empfang durch den UDP Network Adaptor nicht mit gemessen wurde. Dies spiegelt sich auch in seiner geringen Ausführungszeit wider, die lediglich durch den Versand von Routing-Nachrichten verursacht wird. Am Unterschied der Aufwände der Multiplexer zwischen Sender und Empfänger ist zu erkennen, dass die Namensauflösung, die hier nur auf Senderseite durchgeführt werden muss, einen nicht unerheblichen Teil der Ausführungszeit ausmacht: Der Empfangsaufwand beträgt ca. nur zwei drittel des Sendeaufwands.

Während mit dem SimpT Netlet damit eine Datenrate von ca. 8 MiB/s erreicht wurde, waren es beim Template Netlet lediglich ca. 1,7 MiB/s (jeweils mit Messfunktionen im Message Scheduler). Dies liegt zwar zum einen an der deutlich höheren Ereignisanzahl, die im Template Netlet verarbeitet werden muss, zum anderen aber auch an den fehlenden Optimierungen, sowohl in den Bausteinen des Template Netlets als auch in den restlichen Komponenten wie bspw. dem Multiplexer. Letzterer verursacht durch die mindestens doppelte Anzahl an Nachrichten durch die Bestätigungen auch einen höheren Aufwand beim Template Netlet.

Mit dieser Untersuchung lassen sich drei wesentliche Schlüsse ziehen:

1. Für ein ereignisorientiertes System, wie es in NENA verwendet wird, ist es bei feingranularer Unterteilung der Bausteine wichtig, dass der Ereignisaustausch effizient realisiert wird. Dass solche Systeme effizient realisierbar sind, zeigen Ansätze zu Betriebssystem-Mikro-Kerneln, die auf eine schnelle und effiziente Kommunikation der einzelnen Systemkomponenten untereinander angewiesen sind. Zudem sollte das ereignisorientierte System Gebrauch von mehreren Prozessorkernen machen, sodass Ereignisse parallel abgearbeitet werden können. Das ereignisorientierte System in NENA sieht dies zwar prinzipiell vor, es wurde aber im Rahmen dieser Arbeit nicht näher untersucht.

2. Der hier verwendete SimpA Multiplexer verursacht ebenfalls einen verhältnismäßig hohen Aufwand im Vergleich zu den restlichen Komponenten und Bausteinen, da er mit der Namensauflösung für ausgehende Nachrichten und mit der Zuordnung des Flow State Objektes für eingehende Nachrichten wichtige Aufgaben übernimmt. Auch hier sind weitere Implementierungsoptimierungen notwendig, die im Rahmen dieser Arbeit jedoch nicht durchgeführt wurden.

3. Das in NENA realisierte ereignisorientierte System, welches als zentrales Element den Message Scheduler verwendet, eignet sich hervorragend zur Analyse eines komponentenbasierten Protokolls (*profiling*).

Damit ist eine zielgerichteten Optimierung einzelner Bausteine, die sich als Engpass mit einer vergleichsweise hohen Ausführungszeit herausstellen, möglich. Für die dazu notwendigen Messungen muss lediglich der Message Scheduler ähnlich wie bei der Untersuchung hier angepasst werden.

7.5.3 Anmerkungen zur Skalierbarkeit

Der in dieser Arbeit beschriebene Ansatz sieht vor, dass verschiedene Dienstanbieternetze mit jeweils eigenen Protokollen nebenläufig betrieben werden. Netzwerkvirtualisierung (vgl. Abschnitt 3.3) erlaubt es dabei, die Datenströme unterschiedlicher virtueller Netze über dieselbe physische Infrastruktur zu leiten. Bandbreitenzusicherungen durch die Infrastrukturbetreiber können dabei teilweise im Vorfeld vereinbart werden. Werden keine ausgehandelt, können die Daten der virtuellen Netze weiterhin mit statistischem Zeit-Multiplexing übertragen werden, was eine ähnlich effiziente Nutzung der Bandbreite wie heute bedeutet.

Für Zwischensysteme im Netz (bspw. Middleboxes) sehen einige Netzwerkvirtualisierungsansätze vor, Techniken der Betriebssystemvirtualisierung zu verwenden. Das bedeutet jedoch, dass für jede Instanz eines virtuellen Zwischensystems große Teile eines Betriebssystems repliziert werden. Dies hat entsprechend mehr Speicher- und Rechenaufwand zur Folge. NENA bietet in solchen Fällen durch die Module die Möglichkeit, individuelle Funktionalität pro Netz bereitzustellen, ohne ein gesamtes Betriebssystem zu replizieren. Insbesondere mit Netlets und Servlets stehen mächtige Abstraktionen für maßgeschneiderte Netzdienste bereit. Auf eine entsprechende Isolation der Module muss jedoch bei der Implementierung des NENA-Rahmenwerks geachtet werden. Bei der Implementierungen der Module hingegen müssen keine speziellen Maßnahmen zur Isolation getroffen werden, was deren Entwicklung deutlich vereinfacht. Das NENA-Rahmenwerk bietet hier also eine bessere Skalierbarkeit für virtuelle Knoten und für die Entwicklung individueller Dienste auf diesen Knoten. Für wie viele Dienstanbieter eine NENA-Instanz dabei Dienste ausführt hängt im wesentlichen von der Komplexität der Dienste ab und muss vom Infrastrukturbetreiber bei der Zuteilung der Ressourcen bewertet werden. Hier kann der Infrastrukturbetreiber jedoch auf zusätzliche Hardware ausweichen und weitere NENA-Instanzen verwenden.

8. Zusammenfassung und Ausblick

Mit neuen Übertragungstechnologien und innovativen Anwendungen finden täglich viele Neuerungen Einzug in den Kommunikationsalltag. Vorschläge für das Vermittlungsprotokoll des Internet oder für dessen Transportprotokolle werden dagegen nur langsam oder gar nicht eingesetzt, obwohl neben den Dienstanbietern auch die Benutzer von optimierten Protokollen profitieren würden. Nicht alle Vorschläge sind dabei für alle Anwendungsdienste gleichermaßen sinnvoll, weswegen es Ziel dieser Arbeit war, ein Szenario zu ermöglichen, in dem Dienstanbieter ihre eigenen Netze mit maßgeschneiderten Protokollen verwenden können. Netze für verschiedene Dienste oder Dienstanbieter werden dabei nebenläufig betrieben. Eine Grundvoraussetzung dazu bietet Netzwerkvirtualisierung. Jedoch stellen sich in diesem Szenario weitere Herausforderungen, für die in der vorliegenden Arbeit folgende Beiträge geleistet wurden:

- Unterstützung des Entwurfs mit komponentenbasierten Protokollen

  Mit einem einfachen Entwurfsprozess für die komponentenbasierte Protokollentwicklung, wofür sog. Protokollschablonen verwendet werden, kann ein Dienstanbieter mit der Hilfe eines Experten neue, angepasste Protokolle entwerfen. Der Experte wählt dazu auf Basis der Anforderungen des Dienstanbieters existierende Protokollkomponenten aus, führt ggfs. Anpassungen durch und entwirft bei Bedarf neue Komponenten.

- Entkopplung von Anwendungen und Protokollen durch eine protokollagnostische Anwendungsschnittstelle

Damit Anwendungen flexibler bei der Verwendung von Kommunikationsprotokollen werden, wurde eine protokollagnostische Anwendungsschnittstelle entworfen, mit der keine protokollspezifischen Angaben mehr von der Anwendung verlangt werden. Die Anwendung gibt lediglich Namen in Form von URIs und Anforderungen an den gewünschten Kommunikationsdienst an, welcher potentiell von verschiedenen Protokollen erfüllt werden kann.

- Nebenläufiger Betrieb unterschiedlicher Protokollstapel mit dem NENA-Rahmenwerk

  Protokolle verschiedener Dienstanbieter müssen geeignet gekapselt sein und nebenläufig auf einem System ausführbar sein. Ein Rahmenwerk, welches dies leistet und zusätzlich die protokollagnostische Anwendungsschnittstelle implementiert sowie die Realisierung komponentenbasierter Protokolle erleichtert, wurde in dieser Arbeit mit der Netlet-basierten Knotenarchitektur (NENA) vorgestellt.

In den folgenden beiden Abschnitten werden die Ergebnisse der Arbeit zusammengefasst und es wird ein Ausblick auf Anknüpfungspunkte für weiterführende Arbeiten gegeben.

8.1 Ergebnisse der Arbeit

Mit verschiedenen Fallstudien und konkret implementierten Anwendungen, Protokollen und ganzen Netzwerkarchitekturen konnte gezeigt werden, dass die in dieser Arbeit entwickelten Konzepte vielerlei Vorteile bringen und wichtige Beiträge zum nebenläufigen Betrieb dienstanbieterspezifischer Netze darstellen:

- Vereinfachungen für Anwendungen

  Mit der protokollagnostischen Anwendungsschnittstelle muss sich der Anwendungsentwickler nicht mehr mit protokollspezifischen Angaben (z. B. zur Adressierung) auseinandersetzen. Durch die gewählten Abstraktionen (Namen, Anforderungen und Ressourcen-Endpunkte) und die zusätzlichen inhaltsbasierten Primitiven (GET, PUT) können einige Anwendungen sogar auf die Realisierung eines Anwendungsprotokolls wie HTTP oder FTP verzichten. Solche Anwendungsprotokolle können somit unterhalb dieser Anwendungsschnittstelle realisiert werden, womit deren Entwicklung ebenfalls von den Anwendungen entkoppelt ist und auch hier Alternativen für existierende Anwendungen verwendet werden können. Zum Nachweis und als Vorschlag für die Nutzung

der Anwendungsschnittstelle wurden verschiedene Anwendungen näher betrachtet: ein Web-Browser, eine Chat-Anwendung, eine Video-Anwendung, ein Datei-Server und eine Gruppenkommunikationsanwendung.

Zudem wurde mit der delegierten Übertragung gezeigt, das weitere Vereinfachungen sinnvoll sind: Hier gibt die Anwendung nur noch Quelle und Ziel der Übertragung an (wobei hier jeweils auch lokale Ressourcen zugelassen sind) und wird nach dem Anstoß der Kommunikation lediglich über den Fortschritt informiert. Dies ermöglicht bspw. die direkte Speicherung eines Datei-Downloads auf der lokalen Festplatte, ohne dass die Anwendung selbst die Daten in Empfang nehmen und auf die Festplatte schreiben muss.

- Flexible Protokollauswahl

  Da die Anwendung keine Protokolle mehr direkt auswählt, sondern nur noch den Kommunikationsdienst mit Anwendungsanforderungen beschreibt, können alternative Protokolle eingesetzt werden, die den gleichen angeforderten Kommunikationsdienst erfüllen. Hierdurch können auch für existierende Anwendungen neue Protokolle eingesetzt werden, die auf die Bedürfnisse des jeweiligen Dienstanbieters zugeschnitten sind.

- Vereinfachungen für den komponentenbasierten Protokollentwurf

  Anhand des Entwurfs und der Implementierung eines detaillierten komponentenbasierten Transportprotokolls wurde gezeigt, dass der vorgestellte Protokollschablonenansatz geeignet für die Realisierung und die anschließende Modifikation von Protokollen ist. Wichtige Eigenschaften der Schablone sind dabei der ereignisorientierte Ansatz mit dem Austausch von Kontrollinformationen über sog. Zustandsobjekte. Während einige Modifikationen durch den einfachen Austausch von Bausteinen erreicht werden konnten (z. B. alternative Staukontrollalgorithmen), waren für andere geringe Änderungen an der Schablone notwendig (z. B. um neue Ereignisse oder Bausteine zu integrieren). In allen Fällen konnten jedoch große Teile des existierenden Entwurfs wiederverwendet werden.

  Durch die Laufzeituntersuchungen wurde schließlich gezeigt, dass es mit der Transportprotokollschablone möglich ist, das Verhalten des verbreiteten TCP-Protokolls in der NewReno-Variante umzusetzen. Zudem wurde festgestellt, dass eine effiziente Implementierung des ereignisorientieren Systems zur Umsetzung wichtig ist, aber auch, dass der komponentenbasierte Ansatz eine detaillierte Laufzeitanalyse der einzelnen Bausteine ermöglicht.

- Nebenläufiger Betrieb verschiedener Netzwerkarchitekturen

  Mit NENA wurden Konzepte erarbeitet, die die nebenläufige Verwendung verschiedener Protokollstapel und gänzlich unterschiedlicher Architekturen ermöglichen. An verschiedenen Protokollen und ganzen Netzwerkarchitekturen wie CCN wurde untersucht, wie gut NENA für die unterschiedlichen Anforderungen der Protokolle und Architekturen geeignet ist. Dazu wurde zunächst eine Umsetzung der Protokolle und Architekturen für das NENA-Rahmenwerk konzipiert und schließlich für dessen Prototyp implementiert. Hier konnte gezeigt werden, dass eine geeignete Umsetzung in NENA immer möglich war. Durch die Konzepte konnten sogar Protokolle und Dienste, die heute auf Anwendungsschicht realisiert sind, innerhalb von Netlets und Servlets umgesetzt werden. Die Aufgaben des Multiplexers variieren dabei in ihrer Komplexität abhängig von der jeweiligen Netzwerkarchitektur. Wegen der zentralen Rolle des Multiplexers in jeder Architektur (ihn passieren alle ausgehende und ankommende Nachrichten), ist auf dessen effiziente Implementierung zu achten, wie im Rahmen der Laufzeituntersuchungen beobachtet wurde.

Aus den so gesammelten Erfahrungen kann schließlich gefolgert werden, dass das Ziel eines generischen Rahmenwerks für maßgeschneiderte Kommunikationsprotokolle mit einer geeigneten Schnittstelle für Anwendungen und Unterstützung für komponentenbasierte Protokolle erreicht wurde. Zusammen mit Netzwerkvirtualisierung sind damit wichtige technische Voraussetzungen für das Szenario dienstanbieterspezifischer Netze gegeben.

Das NENA-Rahmenwerk und dessen Prototyp-Implementierung stellen wichtige Teilergebnisse des EU FP7 Projektes 4ward[1] und des BMBF Projektes G-Lab[2] dar. Der Prototyp wurde zudem auf Fachtagungen vorgeführt [MBVW+09, BaMW12] und als Open Source Software veröffentlicht[3].

8.2 Ausblick und weiterführende Arbeiten

Die Ergebnisse dieser Arbeit zeigen, dass geeignete Abstraktionen für eine Entkopplung von Anwendungen und Protokollen möglich sind und dass der nebenläufige Betrieb dienstanbieterspezifischer Netze technisch realisierbar ist. Jedoch sind weitere Arbeiten im Sinne dieses Szenarios möglich und notwendig:

- Anwendungsanforderungen, die den gewünschten Kommunikationsdienst beschreiben, müssen standardisiert werden. Mit [MCWR+13]

[1] http://www.4ward-project.eu/
[2] http://www.german-lab.de/
[3] http://nena.intend-net.org/

sind dazu bereits Bestrebungen im Rahmen der IETF im Gange, die auf eine Festlegung von Eigenschaften für Transportdienste zielen. Aber auch dies kann nur als Anfang gewertet werden, da die Konzepte der vorliegenden Arbeit mehr als nur Transportdienste ermöglichen: Mit DTN, BitTorrent, CCN und einem Nachrichten-Broker wurden bereits Protokolle und Architekturen betrachtet, die deutlich mehr Eigenschaften besitzen, als die heute verbreiteten Transportprotokolle.

- Ein wichtiger Bestandteil des beschriebenen Szenarios ist die Ausbringung von dienstanbieterspezifischen Protokollen und Architekturen auf die Geräte innerhalb eines virtuellen Netzes und auf die Endsysteme der Benutzer. Gerade bei den Endsystemen der Benutzer muss hier sichergestellt werden, dass die nachgeladenen Module – also Multiplexer, Netlets und Servlets – keine Schadsoftware enthalten. Ansätze dazu zeigen, dass entsprechende Infrastrukturen denkbar sind [WiHa12]. Weitere praktische Untersuchungen sind hierzu jedoch notwendig.

- Im Zusammenhang mit Software-Defined Networking (SDN) [Open12] eröffnen sich vielfältige Möglichkeiten zur Nutzung von effizienter Standard-Hardware für dienstanbieterspezifischer Netze: SDN setzt auf die Trennung von Daten und Kontrollfunktionalität, um eine effiziente Weiterleitung von Paketen auf Zwischensystemen zu erreichen. Gleichzeitig bietet es aber auch eine zentrale Möglichkeit, den Netzwerkverkehr individuell nach den Bedürfnissen des Betreibers anzupassen. Dies wird erreicht, indem sog. Controller-Anwendungen, die in Software geschrieben sind, zentral Regeln vorgeben können, die auf den einzelnen SDN-fähigen Zwischensystemen im Netz installiert werden. Diese Regeln werden dann durch spezielle Hardware auf die zu verarbeitenden Pakete angewandt. Somit muss nicht aus Kosten- und Leistungsgründen auf Hardware, die nur etablierte Protokolle verarbeitet, zurückgegriffen werden.

Aus nicht-technischer Sicht ergeben sich jedoch ebenfalls einige Fragestellungen. Zentral ist hier die Einführung eines kompletten Ökosystems für das Szenario dienstanbieterspezifischer Netze. Dazu müssen Geschäftsmodelle für die verschiedenen beteiligten Rollen – also z. B. für den Dienstanbieter, den Architekten und den Infrastrukturbetreiber – erstellt werden. Obwohl dieses Szenario neue Geschäftsfelder ermöglicht, müssen diese aus ökonomischer Sicht zunächst weiter untersucht werden.

Literaturverzeichnis

[ABEH+04] B. Ahlgren, M. Brunner, L. Eggert, R. Hancock und S. Schmid. Invariants: A New Design Methodology For Network Architectures. In *Proc. of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA'04)*, Portland, OR, USA, 2004. S. 65–70.

[AlPB09] M. Allman, V. Paxson und E. Blanton. TCP Congestion Control. RFC 5681 (Standards Track), September 2009.

[Back10] H. Backhaus. Towards a Property and Requirement-based Application Interface for Future Networks. In *10th Würzburg Workshop on IP: Joint ITG, ITC, and Euro-NF Workshop "Visions of Future Generation Networks" (EuroView2010)*, Würzburg, Germany, August 2010.

[BaMW12] H. Backhaus, D. Martin und H. Wippel. A Network Pluralist's Approach to Online Video Stores. In *12th Würzburg Workshop on IP: ITG Workshop "Visions of Future Generation Networks"(EuroView2012)*, Würzburg, Germany, Juli 2012.

[Baue13] R. Bauer. Effektaggregation bei komponentenbasierten Netzwerkprotokollen. Studienarbeit, Institut für Telematik, KIT, Mai 2013.

[BePe12] M. Belshe und R. Peon. SPDY Protocol. Internet Draft (draft-mbelshe-httpbis-spdy-00), August 2012.

[Bern+05] Berners-Lee und andere. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standards Track), Januar 2005.

[Brau93] T. Braun. Performance of a Parallel Transport Subsystem Implementation. In *2nd IEEE Workshop on the Architecture and Im-*

plementation of High-Performance Communication Subsystems, Williamsburg, Virginia, USA, September 1993.

[Brau95] T. Braun. PATROCLOS: A Flexible and High-Performance Transport Subsystem. In *Proc. of the 4th International IFIP Workshop on Protocols for High Speed Networks*, Band 8, London, UK, 1995. S. 205–223.

[BrFH03] R. Braden, T. Faber und M. Handley. From Protocol Stack to Protocol Heap: Role-Based Architectures. *SIGCOMM Comput. Commun. Rev.* 33(1), Januar 2003, S. 17–22.

[BrSc94] T. Braun und C. Schmidt. Parallel transport subsystem implementation for high-performance communication. *Concurrency: Practice and Experience* 6(4), Juni 1994, S. 375–391.

[CCRJ11] Y. Cheng, J. Chu, S. Radhakrishnan und A. Jain. TCP Fast Open. Internet Draft (draft-cheng-tcpm-fastopen-02), Dezember 2011.

[CETW+11] M. Cotton, L. Eggert, J. Touch, M. Westerlund und S. Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335 (Best Current Practice), August 2011.

[ChBo10] N. M. K. Chowdhury und R. Boutaba. A survey of network virtualization. *Computer Networks* 54(5), 2010, S. 862–876.

[ChKr13] S. Cheshire und M. Krochmal. Multicast DNS. RFC 6762 (Standards Track), Februar 2013.

[Cisc13] Cisco Systems Inc. Cisco Visual Networking Index (VNI): Forecast and Methodology 2012–2017. White Paper, 2013.

[Clar09] D. D. Clark. Toward the design of a Future Internet. Whitepaper, Oktober 2009. Version 7.

[ClTe90] D. D. Clark und D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *SIGCOMM Comput. Commun. Rev.* 20(4), 1990, S. 200–208.

[Cohe03] B. Cohen. Incentives Build Robustness in BitTorrent. In *Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, Juni 2003.

[Croc06] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), Juli 2006.

[CrOv08] D. Crocker und P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 5234 (Standards Track), Januar 2008.

[Day08] J. Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall. 2008.

[DiRe08] T. Dierks und E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Standards Track), August 2008.

[DRBB+07] R. Dutta, G. N. Rouskas, I. Baldine, A. Bragg und D. Stevenson. The SILO Architecture for Services Integration, controL, and Optimization for the Future Internet. In *Proc. of the IEEE International Conference on Communications (ICC 2007)*, Glasgow, Scotland, UK, 2007. S. 1899–1904.

[DuMZ10] M. Duerst, L. Masinter und J. Zawinski. The 'mailto' URI Scheme. RFC 6068 (Standards Track), Oktober 2010.

[FGMF+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Juni 1999. Updated by RFC 2817.

[FiTa00] R. Fielding und R. Taylor. Principled Design of the Modern Web Architecture. In *Proceedings of the 2000 International Conference on Software Engineering*, Limerick, Ireland, 2000. S. 407–416.

[Fran12] E. Frank. Reengineering the Internet: Protokolldekomposition und -neuentwurf mit wiederverwendbaren Bausteinen. Master-Arbeit, Insitut für Telematik, KIT, Juni 2012.

[Funk12] I. Funke. Realisierung des Content-Centric Networking-Ansatzes in der Netlet-based Node Architecture. Bachelor-Arbeit, Institut für Telematik, KIT, November 2012.

[Goog14] Google Inc. Google Protocol Buffers. Online: http://code.google.com/p/protobuf/, zuletzt abgerufen im Januar, 2014.

[GuGr07] Y. Gu und R. L. Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks* 51(7), Mai 2007, S. 1777–1799.

[GuVE00] A. Gulbrandsen, P. Vixie und L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782 (Standards Track), Februar 2000.

[HADL+12] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan und P. Steenkiste. XIA: Efficient Support for Evolvable Internetworking. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, USA, April 2012.

[HaWi11] O. Hanka und H. Wippel. Secure Deployment of Application-Tailored Protocols in Future Networks. In *Proc. of the International Conference on the Network of the Future (NoF 2011)*, Paris, France, November 2011. IFIP/IEEE.

[HeSK10] C. Henke, A. Siddiqui und R. Khondoker. Network functional composition: State of the art. In *Australasian Telecommunication Networks and Applications Conference (ATNAC 2010)*, Auckland, Neuseeland, Oktober 2010.

[HFGN12] T. Henderson, S. Floyd, A. Gurtov und Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC6582 (Standards Track), April 2012.

[Hint13] P. Hintjens. *ZeroMQ: [messaging for many applications]*. O'Reilly Media. 2013.

[IBEu10] IBM und Eurotech. MQTT V3.1 Protocol Specification, August 2010.

[IEE90] IEEE Standard Glossary of Software Engineering Terminology (IEEE Std. 610.12-1990), Dec 1990.

[JSBP+09] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton und R. L. Braynard. VoCCN: Voice over Content-Centric Networks. In *Proc. of the 2009 Workshop on Re-architecting the Internet (ReArch'09)*, Rome, Italy, Dezember 2009.

[JSTP+09] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs und R. L. Braynard. Networking Named Content. In *Proc. of the 5th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2009)*, Rome, Italy, Dezember 2009.

[KHMB+08] A. Keller, T. Hossmann, M. May, G. Bouabene, C. Jelger und C. Tschudin. A System Architecture for Evolving Protocol Stacks. In *Proc. of 17th International Conference on Computer Communications and Networks (ICCCN 2008)*, St. Thomas, Virgin Islands, USA, August 2008.

[KMCJ+00] E. Kohler, R. Morris, B. Chen, J. Jannotti und M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18(3), 2000, S. 263–297.

[KoHF06] E. Kohler, M. Handley und S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), März 2006.

[KSBF+11] T. Koponen, S. Shenker, H. Balakrishnan, N. Feamster, I. Ganichev, A. Ghodsi, P. B. Godfrey, N. McKeown, G. Parulkar, B. Raghavan, J. Rexford, S. Arianfar und D. Kuptsov. Architecting for innovation. *SIGCOMM Comput. Commun. Rev.* 41(3), Juli 2011, S. 24–36.

[KuGW06] T. Kuhn, R. Gotzhein und C. Webel. Model-Driven Development with SDL - Process, Tools, and Experiences. *Model Driven Engineering Languages and Systems* Band 4199/2006, 2006, S. 83–97.

[KuRo13] J. F. Kurose und K. W. Ross. *Computer Networking - A Top-Down Approach*. Pearson. ISBN 978-0-273-76896-8, 6. Auflage, 2013.

[Lien13] C. Lienhard. Integration von Delay-Tolerant Networking in das NENA Framework. Bachelor-Arbeit, Institut für Telematik, KIT, April 2013.

[Lier+11] F. Liers und andere. GAPI: A G-Lab Application-to-Network Interface. In *Proceedings of the 11th Würzburg Workshop on IP: Joint ITG and Euro-NF Workshop "Visions of Future Generation Networks"(EuroView2011)*, Würzburg, Germany, August 2011.

[LIJMO+10] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide und F. Jahanian. Internet inter-domain traffic. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, New Delhi, India, August 2010. ACM, S. 75–86.

[LuLi07] J. Ludewig und H. Lichter. *Software Engineering*. dpunkt Verlag. 2007.

[Mart13] D. Martin. Protocol Composition Revisited: A Template-based Transport Architecture. In *Proc. of the 2nd International Conference on Future Generation Communication Technologies (FGCT 2013)*, London, UK, Dezember 2013.

[Math12] M. Mathis. Laminar TCP and the case for refactoring TCP congestion control. Internet Draft (draft-mathis-tcpm-tcp-laminar-01), Juli 2012.

[MaVZ11] D. Martin, L. Völker und M. Zitterbart. A Flexible Framework for Future Internet Design, Assessment, and Operation. *Computer Networks* 55(4), März 2011, S. 910–918.

[MaWB11] D. Martin, H. Wippel und H. Backhaus. A Future-Proof Application-to-Network Interface. In *Proc. of the International Conference on the Network of the Future (NoF 2011)*, Paris, France, November 2011.

[MaWi13a] D. Martin und H. Wippel. API Usage and Message Passing in NENA. Technischer Bericht TM-2013-1, ISSN 1613-849X, http://doc.tm.kit.edu/tr/, Institute of Telematics, KIT, Januar 2013.

[MaWi13b] D. Martin und H. Wippel. Evaluating a Framework for Different Networking Paradigms. In *Proc. of the 38th IEEE Conference on Local Computer Networks (LCN 2013)*, Sydney, Australia, Oktober 2013. S. 527–530.

[MaWi13c] D. Martin und H. Wippel. Evaluating a Framework for Different Networking Paradigms. Technischer Bericht TM-2013-2, ISSN 1613-849X, http://doc.tm.kit.edu/tr/, Institute of Telematics, KIT, Juli 2013.

[MBVW+09] D. Martin, H. Backhaus, L. Völker, H. Wippel, P. Baumung, B. Behringer und M. Zitterbart. Designing and Running Concurrent Future Networks. In *34th IEEE Conference on Local Computer Networks (LCN 2009)*, Zurich, Switzerland, Oktober 2009. Poster+Demo.

[MCWR+13] T. Moncaster, J. Crowcroft, M. Welzl, D. Ros und M. Tuexen. Problem Statement: Why the IETF Needs Defined Transport Services. Internet Draft (draft-moncaster-tsvwg-transport-services-01), Dezember 2013.

[MFPA09] G. Maier, A. Feldmann, V. Paxson und M. Allman. On dominant characteristics of residential broadband internet traffic. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, New York, NY, USA, 2009. ACM, S. 90–102.

[MüRe09] P. Müller und B. Reuther. Future Internet Architecture – A Service Oriented Approach. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* 50(6), September 2009, S. 383–389.

[Open12] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. Whitepaper, April 2012.

[PaAl00] V. Paxson und M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Standards Track), November 2000.

[PaPJ11] S. Paul, J. Pan und R. Jain. Architectures for the future networks and the next generation Internet: A survey. *Computer Communications* 34(1), Januar 2011, S. 2 – 42.

[PiRo11] R. Pichler und S. Roock. *Agile Entwicklungspraktiken mit Scrum*. dpunkt Verlag. 2011.

[PoGS10] L. Popa, A. Ghodsi und I. Stoica. HTTP as the Narrow Waist of the Future Internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets 2010)*, Monterey, CA, USA, 2010.

[Prat13] P. A. Prats. Implementation of MQTT protocol inside NENA. Bachelor-Arbeit, Institut für Telematik, KIT, Februar 2013.

[RaFB01] K. Ramakrishnan, S. Floyd und D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001.

[ReHe08] B. Reuther und D. Henrici. A model for service-oriented communication systems. *Journal of Systems Architecture* 54(6), 2008, S. 594–606.

[Reut10] B. Reuther. *Ein serviceorientierter Ansatz zur Abstraktion von Kommunikationsprotokollen im Internet*. Dissertation, Technischen Universität Kaiserslautern, Oktober 2010.

[Rohr09] T. Rohrberg. Werkzeug zur eigenschaftsorientierten Protokollkomposition. Studienarbeit, Institut für Telematik, Universität Karlsruhe (TH), Juni 2009.

[Rosk13] J. Roskind. QUIC: Design Document and Specification Rational. Whitepaper, Juni 2013.

[Sain08] P. Saint-Andre. Internationalized Resource Identifiers (IRIs) and Uniform Resource Identifiers (URIs) for the Extensible Messaging and Presence Protocol (XMPP). RFC 5122 (Standards Track), Februar 2008.

[Sand13] Sandvine Inc. Global Internet Phenomena Report 1H 2013. White Paper, Juli 2013.

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro und E. Merks. *EMF - Eclipse modeling framework*. Addison-Wesley, Boston, Mass., USA. 2. Auflage, 2009.

[ScBu07] K. Scott und S. Burleigh. Bundle Protocol Specification. RFC 5050 (Experimental), November 2007.

[Schl13] R. Schlenker. Entwurf und Implementierung eines XIA-Endsystems in NENA. Bachelor-Arbeit, Institut für Telematik, KIT, April 2013.

[Schu04] H. Schulzrinne. The tel URI for Telephone Numbers. RFC 3966 (Standards Track), Dezember 2004.

[ScRL98] H. Schulzrinne, A. Rao und R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.

[Stew07] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Standards Track), September 2007.

[Szal+08] A. Szalay und andere. GrayWulf: Scalable Clustered Architecture for Data Intensive Computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing 2008)*, Austin, Texas, USA, November 2008.

[ToPi08] J. Touch und V. Pingali. The RNA Metaprotocol. In *Proc. of 17th International Conference on Computer Communications and Networks (ICCCN 2008)*, St. Thomas, Virgin Islands, USA, August 2008.

[UXMV10] J. Ubillos, M. Xu, Z. Ming und C. Vogt. Name-Based Sockets Architecture. Internet Draft (draft-ubillos-name-based-sockets-03), März 2010.

[Varg01] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM 2001)*, 2001.

[VMRB+09] L. Völker, D. Martin, T. Rohrberg, H. Backhaus, P. Baumung, H. Wippel und M. Zitterbart. Design Process and Development Tools for Concurrent Future Networks. In *3rd GI/ITG KuVS Workshop on The Future Internet*, Munich, Germany, Mai 2009. GI/ITG Kommunikation und Verteilte Systeme.

[VMWZ+09] L. Völker, D. Martin, C. Werle, M. Zitterbart und I. El Khayat. Selecting Concurrent Network Architectures at Runtime. In

Proceedings of the IEEE International Conference on Communications (ICC 2009), Dresden, Germany, Juni 2009. IEEE Communication Society.

[Völk12] L. Völker. *Automatisierte Wahl von Kommunikationsprotokollen für das heutige und zukünftige Internet*. Dissertation, Institut für Telematik, KIT, KIT Scientific Publishing, Karlsruhe, 2012.

[Volk13] D. Volk. Entwurf und Implementierung von P2P-Bausteinen für NENA. Bachelor-Arbeit, Insitut für Telematik, KIT, April 2013.

[WeJG11] M. Welzl, S. Jörer und S. Gjessing. Towards a Protocol-Independent Internet Transport API. In *Proc. of the 4th International Workshop on the Network of the Future (FutureNet IV)*, Kyoto, Japan, Juni 2011.

[WGFZ11] H. Wippel, T. Gamer, C. Faller und M. Zitterbart. Hierarchical Node Management in the Future Internet. In *Proc. of 4th International Workshop on the Network of the Future (FutureNet IV)*, Kyoto, Japan, Juni 2011.

[WiHa12] H. Wippel und O. Hanka. End User Node Access to Application-Tailored Future Networks. In *Proc. of the 21st International Conference on Computer Communication Networks (ICCCN 2012)*, Munich, Germany, August 2012.

[Zinn12] T. Zinner. *Performance Modeling of QoE-Aware Multipath Video Transmission in the Future Internet*. Dissertation, Julius-Maximilians-Universität Würzburg, 2012.

[ZiST93] M. Zitterbart, B. Stiller und A. Tantawy. A model for flexible high-performance communication subsystems. *IEEE Journal on Selected Areas in Communications* 11(4), May 1993, S. 507–518.

Glossar

Anwendungsdienst Ein Anwendungsdienst ist ein Dienst, der durch eine Anwendung für andere Anwendungen zur Verfügung gestellt wird.

Dienstanbieter Ein Dienstanbieter (meist eine Organisation oder Firma) stellt einen Anwendungsdienst bereit (z. B. Video-Streaming). Dies ist zu unterscheiden von einem Internetdienstanbieter (Internet Service Provider, ISP), der einen Anschluss an das Internet bereitstellt.

Dienstanbieternetz Ein Dienstanbieternetz wird von einem Dienstanbieter (oder einem Dritten, der im Auftrag des Dienstanbieters agiert) entworfen und verwaltet. In diesem Netz befinden sich in der Regel Server und/oder andere Netzkomponenten, die zur Dienstbereitstellung notwendig sind.

Kommunikationsparadigma Ein Kommunikationsparadigma beschreibt grundlegende Elemente einer Kommunikation, d. h. die Subjekte, Objekte und Prädikate. Im Internet-Schichtenmodell werden in jeder Schicht unterschiedliche Kommunikationsparadigmen verwendet. Bei einer klassischen Ende-zu-Ende-Kommunikation zwischen Anwendungen sind die grundlegende Elemente Client und Server, die jeweils Endpunkte öffnen und adressieren. Darüber werden Daten als Strom oder Datagramme gesendet oder empfangen. Auf Vermittlungsschicht sind diese Elemente Knoten, die adressiert werden, und über die Datagramme weitergeleitet werden. Ein neueres Kommunikationsparadigma ist bspw. die inhaltsbasierte Kommunikation, bei der stattdessen Datenblöcke direkt adressiert und angefordert werden.

Kommunikationssoftware Als Kommunikationssoftware wird die Implementierung aller Protokolle und Funktionen auf einem Gerät bezeichnet, die der Kommunikation über ein oder mehrere Netzwerke dienen.

Protokollteile, die dabei auf speziellen Hardware-Komponenten ausgelagert sind, zählen ebenfalls dazu.

Kommunikationssystem Ein Kommunikationssystem enthält alle Funktionen und Mechanismen, die zur Übertragung von Informationen zwischen zwei Anwendungen auf verschiedenen Geräten notwendig sind. Es enthält dabei neben den Protokollen auch die notwendige Hardware.

Nachricht Eine Nachricht enthält neben der Dateneinheit, die an ein anderes System gesendet werden soll bzw. von diesem empfangen wurde, auch Kontroll- und Zustandsinformationen. Dies ist zu unterscheiden von einem Paket, das die Dateneinheit und notwendige Kontrollinformationen als eine Folge von Bytes kodiert.

Netzwerkarchitektur Eine Netzwerkarchitektur beschreibt die Konstruktion, die Struktur und den Aufbau von Netzwerken, die dieser Netzwerkarchitektur entsprechen. Dazu zählen die zu verwendenden Protokolle, Netzkomponenten (z. B. Router, Switches) und Kommunikationsparadigmen.

Paket Ein Paket repräsentiert eine Nachricht (oder Teile davon) und notwendige Kontrollinformationen als eine Folge von Bytes, die über ein Netzwerk gesendet werden oder die über das Netzwerk empfangen wurden.

Protokollfamilie Eine Protokollfamilie besteht aus mehreren Protokollen, die auf unterschiedlichen Schichten angesiedelt sein können und zu einer Netzwerkarchitektur gehören. Die Kommunikationsparadigmen der Protokolle in einer Schicht können dabei auf die Paradigmen der darunterliegenden Schicht abgebildet werden. Ein prominentes Beispiel ist die TCP/IP-Protokollfamilie, zu der auch Anwendungsprotokolle wie HTTP und DNS zählen.

Protokollsoftware Als Protokollsoftware wird die Implementierung eines Protokollstapels auf einem Gerät bezeichnet. Protokollteile, die dabei auf speziellen Hardware-Komponenten ausgelagert sind, zählen ebenfalls dazu.

Protokollstapel Ein Protokollstapel ist eine Schichtung von konkreten Protokollinstanzen auf einem Gerät, z. B. HTTP über TCP über IPv4 über Ethernet.