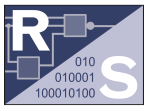
Implementierung eines multithreaded TCP/IP Stacks für einen auf AMIDAR basierten Java Prozessor

Bachelorarbeit
Robert Wiesner
31. Mai 2017

TECHNISCHE UNIVERSITÄT DARMSTADT



Erklärung gemäß § 22 Abs. 7 APB

Hiermit erkläre ich gemäß § 22 Abs. 7 der Allgemeinen Prüfungsbestimmungen (APB) der Technischen Universität Darmstadt in der Fassung der 4. Novelle vom 18. Juli 2012, dass ich die Arbeit selbstständig verfasst und alle genutzten Quellen angegeben habe und bestätige die Übereinstimmung von schriftlicher und elektronischer Fassung.

Darmstadt, den 31. Mai 2017	
Ort, Datum	Robert Wiesner

Fachbereich Elektro- und Informationstechnik
Institut für Datentechnik
Fachgebiet Rechnersysteme
Prüfer: Prof. Dr.-Ing. Christian Hochberger
Betreuer: Dipl.-Inform. Changgong Li

Inhaltsverzeichnis

1. Einleitung **4**

2. Grundlagen Netzwerk **5**
2.1. Netzwerk Schichtenmodell 5
2.2. Ethernet 5
2.2.1. Verfahren 6
2.2.2. Ethernet Frame 6
2.3. Internet Protocol Version 4 (IPv4) 6
2.3.1. Paket Aufbau 6
2.3.2. Adressierung 8
2.3.3. Fragmentierung 8
2.4. Transmission Control Protocol (TCP) 10
2.4.1. Paketaufbau 10
2.4.2. Zustände 12
2.4.3. Nutzeraufrufe 12
2.4.4. Sequenznummern 14
2.4.5. Verbindungsaufbau 14
2.4.6. Datenübertragung 15
2.4.7. Überlastkontrolle 15
2.4.8. Verbindungsabbau 16
2.5. Dynamic Host Control Protocol (DHCP) 17
2.5.1. Paket Format DHCP (Bootstrap) 17
2.5.2. Ablauf 19

3. AMIDAR Prozessor **21**

4. Implementierung des TCP/IP Stacks **22**
4.1. Überblick 22
4.2. IP Stack 23
4.2.1. IP Paket 23
4.2.2. Empfangen von IP Paketen 24
4.2.3. Senden von IP Paketen 25
4.2.4. Address Resolution Protokoll (ARP) 25
4.3. TCP 25
4.3.1. TCP Stack 25
4.3.2. TCP Verbindung 26
4.3.3. TCP-Paket 26
4.4. Zero Copy 30
4.5. DHCP 30
4.5.1. DHCP-Paket 30
4.5.2. DHCP-Controller 31

4.6. Packet Builder 31

5. Evaluation **32**
5.1. Testaufbau 32
5.1.1. Konfiguration des verwendeten AMIDAR-Systems 32
5.2. Grundlegende Funktionen 34
5.3. Leistung 34

6. Zusammenfassung und Ausblick **36**

A. Anhang **37**
A.1. Wireshark Screenshots 37

1 Einleitung

AMIDAR steht für Adaptive Microinstruction Driven Architecture. Dabei handelt es sich um ein Modell eines adaptiven Prozessors, das diesem erlaubt zur Laufzeit einer Anwendung auf deren spezifische Anforderungen zu reagieren. Dazu gehört das Anpassen von Busstrukturen und bestehenden Funktionseinheiten, sowie die Synthese von neuen Funktionseinheiten. Im Fachgebiet Rechnersysteme der TU-Darmstadt wird momentan ein Prototyp in Form eines Java-Prozessors implementiert.

Dieser wird zur Zeit erweitert, mit dem Ziel die Performance deutlich zu erhöhen. Um die Verbesserung der Laufzeit zu evaluieren fehlen Beispielanwendungen. Eine wichtige Anwendung im Umfeld von eingebetteten Systemen ist der TCP/IP Stack.

Zum Zeitpunkt der Arbeit verfügt AMIDAR mit der dazugehörigen API bereits über grundlegende Netzwerk-Funktionalitäten. Dazu gehören die Unterstützung für die Protokolle Ethernet, ARP, begrenzt IPv4 und UDP. Mit UDP kann keine verlustfreie Datenübertragung garantiert werden, welche für viele Netzwerkanwendungen vorausgesetzt wird.

Im Rahmen dieser Arbeit wurde ein TCP-Stack entwickelt, sowie der IP-Stack erweitert, um eine geordnete und verlustfreie Datenübertragung zu realisieren. Die Funktionalität des TCP/IP-Stacks wurde mit einem AMIDAR System getestet, welches auf einen FPGA synthetisiert wurde.

2 Grundlagen Netzwerk

In diesem Kapitel werden die Grundlagen von AMIDAR und der für diese Arbeit genutzten Netzwerkprotokolle erläutert.

2.1 Netzwerk Schichtenmodell

Die Netzwerk-Kommunikation zwischen Anwendungen wird üblicherweise als Schichtenmodell beschrieben. Die unterste Schicht stellt dabei das *Physical Layer* dar und beschreibt die physische Übertragung von Daten. Darüber sorgt die Sicherungsschicht für eine funktionierende Verbindung zwischen den Endgeräten und dem Übertragungsmedium. Auf dieser Schicht wird zum Beispiel das Ethernet Protokoll eingesetzt, das Netzwerkteilnehmer über die MAC-Adresse im lokalen Netz adressiert, die übertragenen Daten auf Fehler überprüft und im Zweifel verwirft. Darüber kommt die Vermittlungsschicht, in der die Endgeräte über mehrere Subnetze hinweg adressiert werden und Routing und Datenflusskontrolle gesteuert werden. Ein wichtiges Protokoll dieser Schicht ist das IP Protokoll.
Bei einer Datenübertragung von einer Anwendung zu einer auf einem anderen Endgerät laufenden, werden die Daten durch die Schichten nach unten gereicht, wobei in jeder Schicht ein neuer Header erzeugt wird, der für die jeweilige Schicht wichtige Informationen enthält. Zum Beispiel werden die Daten von der Anwendung mit betriebssystemabhängigen Systemaufrufen an den TCP-Stack übergeben. Dieser erzeugt ein TCP-Paket, das außer den Daten einen Header enthält, welcher Informationen bereitstellt, die sowohl für das richtige Zusammensetzen der einzelnen Datenpakete beim Empfänger, als auch für die Zuordnung der übertragenen Daten zu der jeweiligen Anwendung benötigt werden.

Bei der anschließenden Erzeugung des IP-Pakets bildet das TCP-Paket, bestehend aus Nutzdaten und TCP-Header, die zu übertragenden Daten. Der IP-Header enthält unter anderem die IP-Adressen des Ziel- und Quell-Geräts. Das IP-Paket bleibt im Normalfall unverändert, bis das Zielgerät erreicht ist. An der nächst unteren Ebene steht das Ethernet Datagramm. Es enthält neben dem IP-Paket die physischen Adressen des Quell-Endgeräts und der nächsten Zwischenstation auf dem Weg zum Ziel. Bei Zwischenstationen wird anhand der Informationen des IP-Headers der nächste Wegpunkt ermittelt und ein neues Datagramm erzeugt.
Wenn ein Datagramm das Ziel erreicht, wird das IP-Paket extrahiert und daraus das TCP-Paket. Anhand der Port Nummer kann das TCP-Paket der Anwendung zugeordnet werden.[Lay]

2.2 Ethernet

Das Ethernet nach der IEEE Norm 802.3 ist seit den 1990ern der am weitesten verbreitete Standard für lokale Netzwerke und beschreibt sowohl die Bitübertragungs- als auch die Sicherungsschicht.

2.2.1 Verfahren

Um zu ermöglichen, dass mehrere Endgeräte auf demselben physischen Medium kommunizieren können, wurde früher ein Zeitmultiplexverfahren eingesetzt, das durch den CSMA/CD Algorithmus gesteuert wurde. Wenn eine Stelle Daten zum Senden bereithielt, wartete diese bis das Medium ungenutzt war und fing dann an, die Daten zu übertragen. Wenn zwei Stellen gleichzeitig zu senden begannen, wechselten beide auf ein *Störung-erkannt* Signalmuster und beendeten die Übertragung. Nach einer zufällig langen Pause würde jeweils ein erneuter Übertragungsversuch gestartet.
Mittlerweile werden Kollisionen durch die Einführung von Switches verhindert. In diesen können Ethernet-Pakete zwischengespeichert werden, bis diese gesendet werden können. Dadurch wird eine Vollduplex-Übertragung zwischen Switches und anderen Endgeräten ermöglicht. Es kann jedoch vorkommen, dass Switches bei zu großen Datenaufkommen überlastet werden weswegen die *Ethernet-Flow-Control* Datenpakete verwerfen kann. Daher ist es wichtig, dass Protokolle auf den darüber liegenden Schichten verworfene Datenpakete erkennen und wiederholt senden können, um eine zuverlässige Datenübertragung zu gewährleisten. [Eth]

2.2.2 Ethernet Frame

Ein Ethernet Paket beginnt mit einer sieben Bit langen Präambel, die aus einer alternierenden Folge von Einsen und Nullen besteht. Diese wird für die Synchronisation der Verbindung benötigt und ermöglicht es, die folgenden Daten von Hintergrundrauschen zu unterscheiden. Unterbrochen wird die Präambel durch das auf eins gesetzte *Start of Frame* Bit was mit dem letzten Bit der Präambel zwei aufeinander folgende Einsen ergibt. Der eigentliche Ethernet Frame beginnt mit der aus sechs Byte bestehenden Ziel-MAC-Adresse gefolgt von der Quell-MAC-Adresse. Dazu kommen zwei Byte, die den Typ des darüber liegenden Protokolls angeben.0x0800 gibt beispielsweise den Typ IPv4 an. Dahinter kommen 46 - 1500 Bytes an Daten, gefolgt von 4 einer Frame Check Sequence, welche eine Größe von vier Byte besitzt. Diese besteht aus einer *CRC Checksum*. [Eth]

2.3 Internet Protocol Version 4 (IPv4)

Das IP Protokoll ist das für die Datenübertragung wichtigste Protokoll auf der Vermittlungsschicht. Es wurde entwickelt, um eine Paket vermittelte Kommunikation über mehrere Computernetzwerke hinweg zu ermöglichen. Quellen und Ziele der Übertragungen werden jeweils als Adressen mit fester 32 Bit-Länge angegeben. Es gibt keine Mechanismen für zuverlässige Übertragung, Flusskontrolle und Sequenzierung, weswegen das darüber liegende Protokoll dies sicher stellen muss. Es gibt jedoch Möglichkeiten zur Paketfragmentierung, falls Datenpakete die maximale Segment-Größe für Pakete der darunter liegenden Schicht überschreiten sollten.[IPr]

2.3.1 Paket Aufbau

Version: Gibt an, welche Version des IP Protokolls verwendet wird. (4Bit)

IHL: Steht für Internet Header Length und gibt an, wie viele 32-Bit-Wörter von dem IP-Header belegt werden.

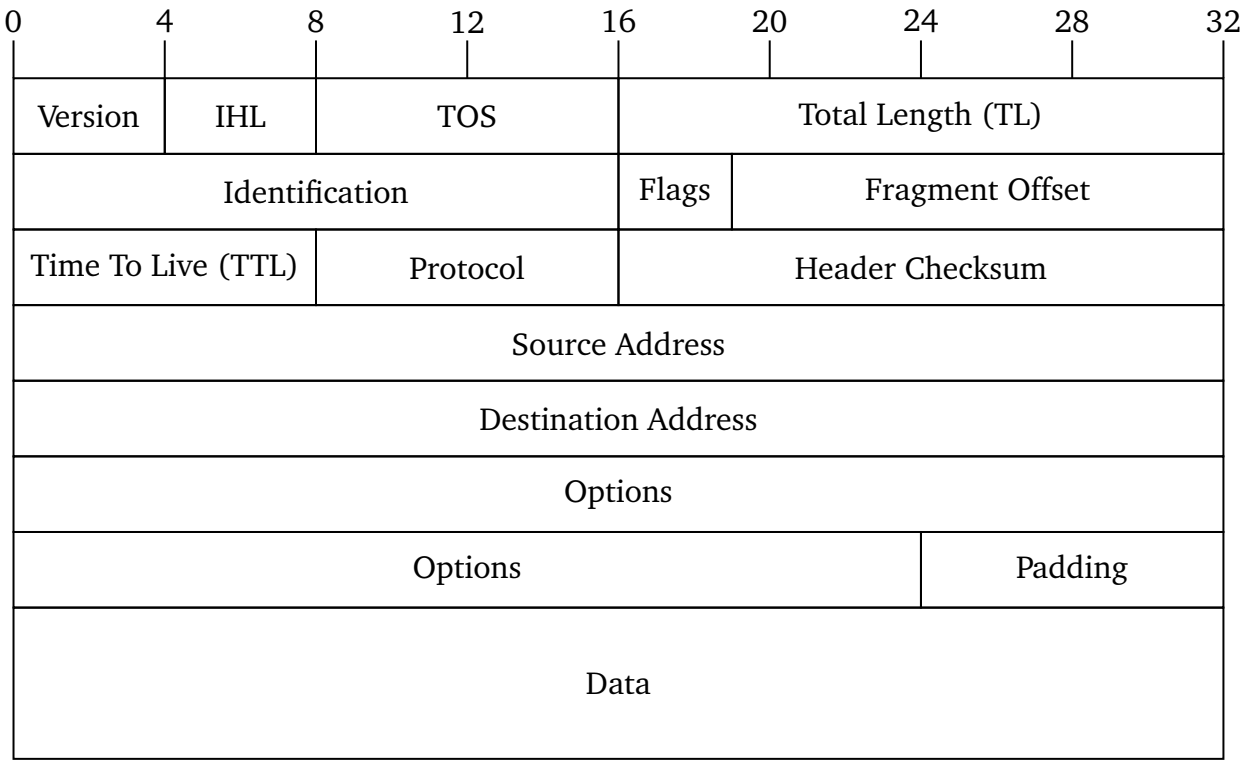


Abbildung 2.1.: IP Segment Format

ToS: Type of Service beinhaltet abstrakte Parameter zur Bestimmung der Qualität des gewünschten Services. Dabei geben die Bits Null bis Zwei die Priorität der Daten an. Die Bits Drei, Vier und Fünf stehen für niedrige Latenz, hohen Durchsatz und hohe Zuverlässigkeit. Die letztgenannten Parameter werden von Netzwerkgeräten unterschiedlich interpretiert, in den meisten Fällen ruft eine bessere Performance für einen der Parameter eine Verschlechterung bei einem anderen hervor.

Paketlänge: Gesamtlänge eines Pakets in Bytes einschließlich des Headers. Die 16Bit ergeben eine theoretische Gesamtlänge von 65.535 Bytes, was für viele Netzwerke jedoch nicht geeignet ist. Als Mindestgröße, die alle Hosts unterstützen müssen, wurden 576 Bytes festgelegt. Das ermöglicht es, 512 Byte Daten und 64 Byte Header in einem Paket zu übertragen. Da der IP Header selber nur 20 Byte benötigt, bleibt noch ein Puffer von 44 Byte für den Header des darüber liegenden Protokolls.

Kennung: Ein Identifikationswert, der benötigt wird, um fragmentierte Pakete zusammen zu setzen. (16Bit)

Flags: 3 Bits von denen das erste reserviert ist und dauerhaft auf Null gesetzt wird. Das zweite Bit gibt an, ob das Paket fragmentiert werden darf. Das letzte Bit wird gesetzt, wenn nach dem Paket noch weitere Fragmente folgen.

Fragment-Offset: Besteht aus 13 Bit und gibt an, an welche Stelle des Datagramms die Daten dieses Fragments gehören.

TTL (Time-to-live): Gibt die maximale Zeit in Sekunden an, die ein Paket im Netzwerk unterwegs sein darf. Sobald die TTL den Wert Null erreicht, muss das Paket gelöscht werden.

Da der Wert bei jeder Zwischenstation, unabhängig von der eigentlichen Verarbeitungszeit, ebenfalls um eins reduziert werden muss, ist die übliche Lebensdauer eines Pakets deutlich kürzer als in der TTL angegeben.

Protokoll: Gibt an, welches Protokoll in der darüber liegenden Ebene verwendet wird.

Header Checksumme: Checksumme nur über den Header. Da sich der Header auf dem Weg über Zwischenstationen verändern kann, zum Beispiel wegen der Time to Live , muss die Checksumme nach jeder Verarbeitung des Headers an den Zwischenstationen neu berechnet werden.

Quell-IP-Adresse: IP Adresse des Hosts, der das Paket ursprünglich versendet hat.(4 Byte)

Ziel-IP-Adresse: IP Adresse des Zielendgeräts. (4 Byte)

Optionen/Füllbits: Enthält IP-Optionen, die durch Füllbits auf 32-Bit-Wörter aufgerundet werden.

[IPr]

2.3.2 Adressierung

Um mehrere Netzwerke innerhalb eines Großen zu ermöglichen, werden IP Adressen interpretiert, indem eine bestimmte Menge der höherwertigen Bits das Netzwerk spezifiziert und die restlichen Bits den genauen Host des gewählten Netzwerks adressieren. Dabei soll Flexibilität bei der Unterteilung von kleineren Teilnetzwerken ermöglicht werden. Daher wurde das Adressfeld entsprechend in bestimmte Klassen unterteilt. Es wurden drei Klassen A,B,C angelegt. Die Klassen unterscheiden sich jeweils durch die Aufteilung des Adressbereichs zwischen Netzwerk-Adresse und Host-Adresse innerhalb des Netzwerks. Die ersten 1-3 Bits der Adresse geben jeweils Aufschluss darüber, zu welcher Netzwerkklasse die jeweilige IP-Adresse gehört. Um innerhalb dieser starren Netzwerke feiner aufgeteilte Subnetze zu erzeugen, kann die Subnetmask genutzt werden. Die Subnetmask ist ebenfalls eine vier Byte große Zahl, deren niederwertige Bits auf Null gesetzt werden und damit den variablen Teil innerhalb des Subnets angeben. Der Rest der Maske wird dabei auf Eins gesetzt.[IPr]

2.3.3 Fragmentierung

Paketfragmentierung wird nötig, wenn ein Paket aus einem Netzwerk kommt, das eine große maximale Segmentgröße erlaubt und durch eines geleitet wird, das nur eine kleinere Segmentgröße erlaubt. Dabei können die Pakete in eine theoretisch nahezu endlose Anzahl von kleinen Paketen zerlegt werden, wobei die Datenblöcke aller Fragmente abgesehen vom letzten ein Vielfaches von 64 Byte an Daten beinhalten müssen.

Damit fragmentierte Paket richtig zusammengesetzt werden können, haben alle Fragmente eines Pakets die gleiche Identifikationsnummer. Für das Zusammensetzen des Datenblock wird der Fragmentoffset benötigt, der angibt, an welcher Stelle des ursprünglichen Datenblocks die

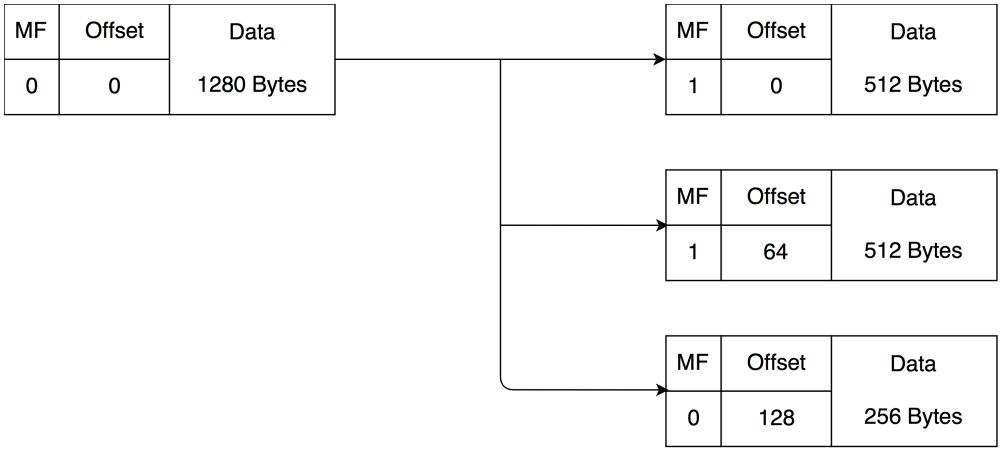


Abbildung 2.2.: IP-Paket Fragmentierung

Daten des Fragments stehen. Das fragmented-flag sagt dabei aus, ob noch weitere Fragmente folgen, oder ob dies das letzte Teil ist. Sobald alle Fragmente eines Pakets beim Ziel eingetroffen sind, kann das ursprüngliche Paket wiederhergestellt werden.[IPr]

2.4 Transmission Control Protocol (TCP)

Da die Protokolle der unteren Schichten, IP und Ethernet, keine fehlerfreie und verlustlose Übertragung der Daten garantieren können, wird ein Protokoll auf der Transportschicht benötigt, das eine zuverlässige Übertragung von Daten zwischen Anwendungen auf entfernten Hosts sicherstellen kann, auch wenn auf jedem Host eine große Anzahl an Anwendungen läuft, die TCP verwenden. Für diesen Zweck wurde das TCP-Protokoll erschaffen. Um dabei die Datenpakete jeweils der richtigen Anwendung zuordnen zu können werden Port-Nummern verwendet.[TCP]

Es handelt sich bei TCP um ein verbindungsorientiertes Protokoll. Das bedeutet, dass es zu Beginn einer Übertragung einen klar definierten Verbindungsaufbau gibt. Nachdem die Verbindung etabliert ist, können Daten vollduplex übertragen werden. Wobei durch Sequenz-Nummern und Acknowledge-Nummern die richtige Reihenfolge und Vollständigkeit der Datenpakete sichergestellt wird. Pakete, deren Erhalt nicht bestätigt wurde, werden automatisch neu übertragen. Des weiteren verfügt TCP über Mechanismen, um eine Überlast auf dem Übertragungsweg rechtzeitig zu erkennen und zu beheben.[TCP]

2.4.1 Paketaufbau

Quell Port (16Bit): Gibt die Portnummer der Anwendung auf Senderseite an.

Ziel Port (16Bit): Gibt die Portnummer der Anwendung auf Empfängerseite an.

Sequence Number (32Bit): Gibt die Sequenz-Nummer des Pakets an.

Acknowledge Number (32Bit): Gibt die Acknowledge Nummer des Pakets an.

Data Offset (4Bit): Anzahl von 32Bit Wörtern aus denen der Header besteht.

Reserved (6Bit): reserviert für zukünftige Nutzung

Steuerungsbits (6Bit): Sechs Flags die für die Ablauf-Steuerung der Übertragung genutzt werden:

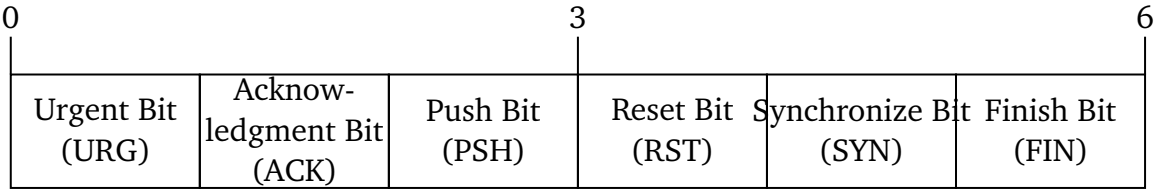


Abbildung 2.3.: TCP Control Bits

URG: Urgent Pointer, wird in moderner Software nicht mehr genutzt und wird von vielen Implementierungen ignoriert.

ACK: Acknowledgment gibt an, ob das Paket eine gültige Bestätigung für empfangene Pakete enthält.

PSH: Ist dieses Flag gesetzt, werden die empfangenen Daten sofort an die Hostsoftware weitergereicht.

RST: Wird im Fehlerfall gesendet und bricht die Verbindung ab.

FIN: Signalisiert das Ende der Verbindung, wenn es keine zu übertragenen Daten gibt.

Window (16Bits): Gibt die Größe des Empfangs-Windows des Absenders an. Gilt für den Empfänger als obere Grenze des Congestion-Windows.

Checksum (16Bits): Für die Berechnung der Checksumme über das TCP Paket wird vorher ein Pseudoheader bestehend aus der Quell- und Ziel IP-Adresse, des IP-Codes für das verwendete Protokoll und die Gesamtlänge des eigentlichen TCP-Pakets berechnet. Die Checksumme wird daraufhin über den Pseudoheader, den Header und die Payload berechnet. Dafür werden diese in 16 Bit große Blöcke aufgeteilt und im Einer-Komplement die Summe über diese gebildet.

UrgentPointer (16Bits): Gibt den Urgentpointer als Offset zu der Sequenznummer an. Wird nur interpretiert, wenn das URG Flag gesetzt ist.

Options: (Variabel): Optionale Informationen, die von TCP Implementierungen unterstützt werden können, um Sicherheit und Performance zu verbessern. Optionen bestehen entweder nur aus einem Optionsbyte oder haben zusätzlich ein weiteres Byte, das die Länge der jeweiligen Option angibt.

Padding: Falls der TCP Header mit den Optionen einen teilweise genutzten 32Bit Block hat, wird dieser mit Nullen aufgefüllt. [TCP]

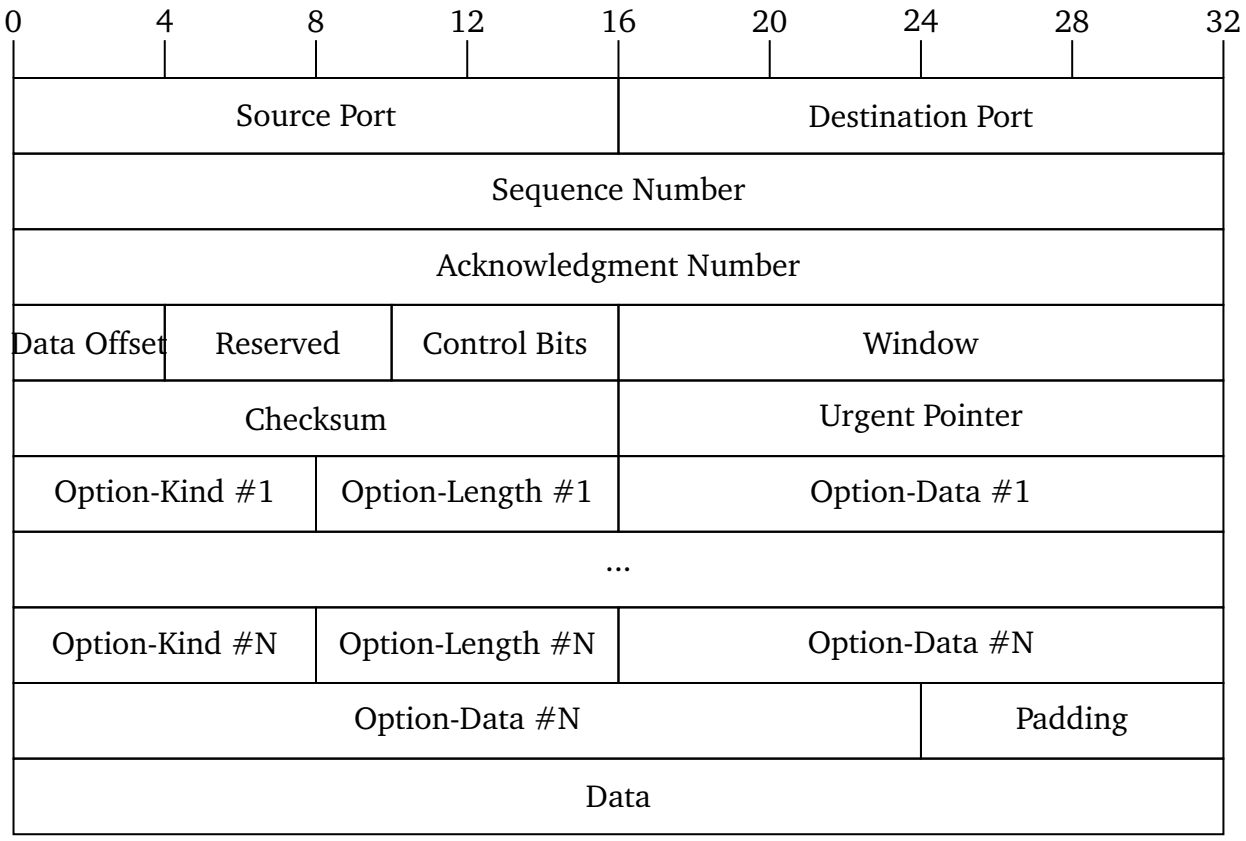


Abbildung 2.4.: TCP Segment Format

2.4.2 Zustände

TCP ist im Gegensatz zu den bisher genannt Protokollen zustandsorientiert, was bedeutet, dass es je nach Zustand anders auf Nutzeraktionen und ankommende Pakete reagiert. Zu den Zuständen gehören :[TCP, RYZ]

CLOSED: Das ist der Startzustand einer TCP Instanz. Die Verbindung ist geschlossen. Usercalls außer *Open* werden mit Fehlermeldungen quittiert und ankommende Pakete werden verworfen und mit einem Reset-Paket beantwortet.

SYN-SENT: Nachdem eine Verbindung initiiert wurde, wird auf eine Antwort des "remote Hosts" gewartet.

SYN-RECEIVED: Nachdem ein SYN Paket erhalten und ein SYN-ACK gesendet wurde, wird auf das ACK gewartet, um den 3-Wege-Handschlag abzuschließen.

ESTABLISHED: Nach dem der Verbindungsaufbau erfolgreich abgeschlossen wurde, befinden sich beide Hosts im ESTABLISHED Zustand, in dem eine Vollduplex Kommunikation möglich ist.

FIN-WAIT-1: Wenn ein Verbindungsabbau initiiert wurde, wird ein FIN Paket gesendet, in den Zustand FIN-WAIT-1 gewechselt und auf die Bestätigung des Erhalts gewartet.

FIN-WAIT-2: Falls die Gegenstelle noch Daten zu übertragen hat, bleibt die Verbindung einseitig offen, um die letzten Pakete zu empfangen.

CLOSING: Wartet auf das letzte ACK-Paket und wechselt, wenn dieses ankommt in den Zustand TIME-WAIT.

TIME-WAIT: Hält die Verbindung nach Beendigung einige Minuten offen, um auf verzögerte Pakete zu reagieren.

[TCP, RYZ]

2.4.3 Nutzeraufrufe

Zu den Zuständen kommen noch eine Reihe von Ereignissen, auf die, abhängig von den jeweiligen Zuständen, entsprechend reagiert werden muss. Dazu gehören neben eintreffenden Paketen und Timeouts die Usercalls.

Active OPEN: Öffnet einen Port und initiiert den Verbindungsaufbau zu einem Remote Host.

Passive OPEN: Öffnet einen Port ohne eine Verbindung zu initiieren und wartet auf einen Verbindungsaufbau,

SEND: Fügt dem Sendepuffer Daten hinzu und sendet gegebenenfalls ein Datenpaket.

RECEIVE: Überprüft, ob genug Daten vorhanden sind und gibt diese an die Anwendung zurück.

CLOSE: Initiiert den Abbau der Verbindung, wenn es keine Daten mehr zu übertragen gibt.

ABORT: Bricht die Verbindung ab. In den meisten Zuständen wird in dem Fall ein Reset Paket gesendet.

[TCP, RYZ]

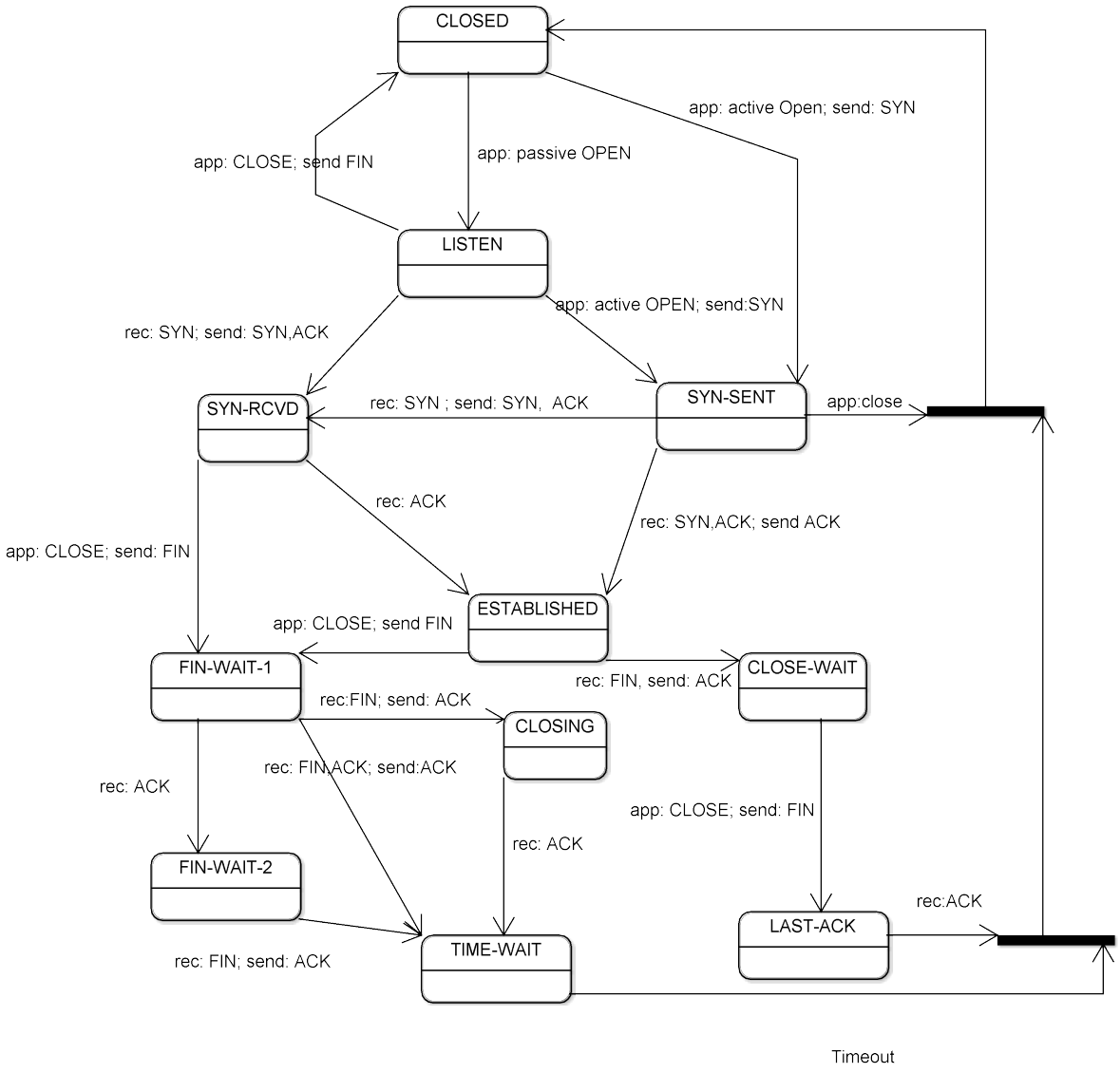


Abbildung 2.5.: vereinfachter Zustandsautomat TCP

2.4.4 Sequenznummern

Ein wichtiges Grundprinzip von TCP ist, dass jedem Byte an Daten eine eigene Sequenznummer zugewiesen werden kann. Dadurch ist es möglich, dass jedes Byte einzeln bestätigt werden kann. Der in TCP dazu verwendete Mechanismus arbeitet kumulativ. Das bedeutet, wenn eine Sequenznummer bestätigt wird, gelten alle Sequenznummern kleiner als die Bestätigungsnummer als angekommen. Dadurch wird es einfach, den Verlust einzelner Pakete zu erkennen und diese nochmal zu senden. Das erste Byte eines Datenpaketes entspricht dabei der Sequenznummer des Pakets. [TCP]

2.4.5 Verbindungsaufbau

Um zuverlässig einen sicheren Verbindungsaufbau zu gewährleisten, verwendet TCP einen Drei-Wege-Handshake. Soll eine neue Verbindung aufgebaut werden, wird die Initiale Sequenznummer zuverlässig generiert. Das SYN-Paket enthält nur den HEADER mit der um eins inkrementierten, initialen Sequenznummer und den SYN-Flag. Nach dem Sendevorgang wird in den Zustand SYN-SENT gewechselt. Wenn die Empfängerseite sich im Zustand LISTEN befindet, kann die Verbindungsanfrage bestätigt werden. Dafür wird ebenfalls eine initiale Sequenznummer generiert. Zur Antwort wird ein Paket erzeugt, das die neue initiale Sequenznummer verwendet und als Acknummer die um eins erhöhte Sequenznummer des SYN-Pakets verwendet. Die Flags für SYN und ACK werden gesetzt. Nach dem Senden wird in den Zustand SYN-RECEIVED gewechselt.
Nach dem Erhalt des SYN-ACK-Pakets sendet die initiierende Seite ein leeres ACK-Paket und wechselt in den ESTABLISHED Zustand. [TCP]

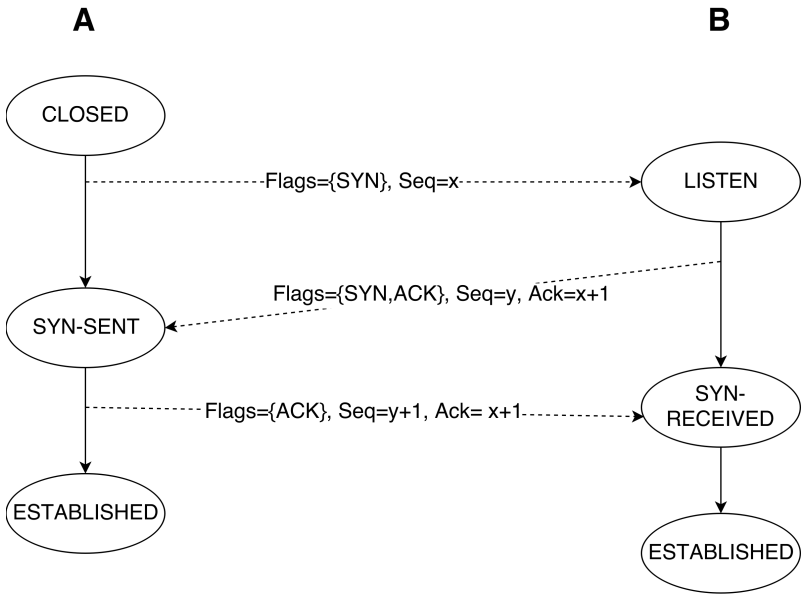


Abbildung 2.6.: TCP Handshake mit Zustandstransitionen

2.4.6 Datenübertragung

Wenn beide Seiten einer Verbindung den Established-Zustand erreicht haben, kann die eigentliche beidseitige Datenübertragung beginnen. Dabei wird die Übertragung der Daten als kontinuierlicher Datenstrom abstrahiert. Eine Anwendung schreibt dabei Bytes auf einen Datenpuffer im TCP-Stack. Die Daten werden nicht sofort übertragen, da sonst viele Pakete mit wenig Nutzdaten gesendet würden und die Verbindung schnell überlastet wäre, wobei ein großer Teil der übertragenen Daten für den durch TCP-, IP- und Ethernet-Header erzeugten Overhead genutzt werden müssten. Sobald der Schreibpuffer eine bestimmte Größe erreicht hat oder die Software einen PUSH signalisiert, wird ein TCP-Paket erzeugt, das die Daten aus dem Puffer überträgt. Jedes Datenpaket wird mit einem Zeitstempel in der Retransmitqueue zwischengespeichert, bis der Erhalt des Pakets bestätigt wurde. Nach dem Eintreffen eines Pakets, dessen ACK-Flag gesetzt wurde, gelten alle Pakete, deren Sequenznummer unter der Acknowledge-Nummer des angekommenen Paktes liegen, als bestätigt. Diese werden aus der Retransmitqueue entfernt. Wenn für ein Paket eine bestimmte Zeit lang keine Bestätigung eingetroffen ist, wird dieses nochmal gesendet.[TCP]

2.4.7 Überlastkontrolle

Ein weiterer Schwerpunkt bei TCP ist das Verhindern der Überlastung des Netzwerks, das zwischen Sender und Empfänger liegt, als auch die des Empfängers selber. Die beiden Mechanismen werden "Flow Control" und "Congestion Control" genannt. "Flow Control"verhindert dabei die Überlastung des Empfängers und "Congestion Control" die des Übertragungsweges.

Für das Senden von Daten wird für "Flow" und "Congestion Control" jeweils ein Fenster bestimmt. Das Fenster gibt an, wie viele Bytes gesendet werden können, ohne dass die vorherigen bestätigt wurden. Wenn die Differenz der Sequenznummern und der letzten erhaltenen Acknowledgenummer so groß ist wie das Fenster, werden keine neuen Datenpakete mehr versendet, bis weitere Pakete bestätigt wurden. Es wird jeweils das kleinere Fenster als Grenze genommen.

Die Größe des Flow "Control Windows" des Empfänger wird im TCP Header der von ihm versendeten Pakete angegeben. Das Problem bei der "Congestion Control" ist, dass die Kapazität der Verbindung nicht trivial zu bestimmen ist. Für eine Annäherung kommen im Normalfall vier Algorithmen zum Einsatz. Mit Hilfe der Variable *Slow Start Threshold* wird festgelegt, welcher der Algorithmen zu diesen Zeitpunkt zum Einsatz kommt. So lange die Größe des *Congestion Windows* unter der *Slow-Start-Threshold* liegt, wird der "Slow Start" Algorithmus verwendet. Eine Überlastung der Verbindung kann festgestellt werden, wenn mehrere Acknowledge Pakete für dieselbe Sequenznummer empfangen werden.

Zu Beginn einer Übertragung liegen noch keine Informationen über die mögliche Bandbreite des Netzwerks vor, weswegen die Bandbreite langsam sondiert werden muss. Dabei wird der "Slow Start"-Algorithmus genutzt, bis durch schnelles Ansteigen der Übertragungsrate der Grenzwert erreicht wird. Die "Slow Start"-Phase wird genutzt, bis die Größe des *Congestion Windows* die *Slow-Start-Threshold* übersteigt oder eine Überlastung der Verbindung festgestellt wird. Da die Grenze der Verbindungskapazität sondiert werden soll, wird die *Slow-Start-Threshold* zu

Beginn auf einen Wert gesetzt, der nicht erreicht werden kann. Wenn es zu einen Packet Timeout kommt, wird die *Slow Start Threshold* auf einen Wert, der einem Bruchteil des erreichten Fensters entspricht, gesetzt und der "Slow Start" Algorithmus mit einem größeren initialen Fenster weiter verwendet. Diesmal wird die "Slow Start Phase" durch das Überschreiten der *Slow-Start-Threshhold* beendet. Für den Rest der Übertragung wechseln sich die beiden Algorithmen ab, bis die Verbindung beendet ist.

Es gibt eine Reihe von Varianten für die Congestion Control Implementierung. Alle davon verwenden eine Variation der folgenden Algorithmen. Hier wird als Beispiel die Reno Variante genannt.[TCP, cc]

Slow Start: Vergrößert das *Congestion Window* mit jedem bestätigten Segment um die Größe des Segments. Effektiv eine Verdoppelung des Windows pro Roundtrip. N := Anzahl der im letzten Ack bestätigten Sequenznummern

Congestion Avoidance: Arbeitet nach dem Prinzip additiv erhöhen und multiplikativ erniedrigen.

Fast Retransmit: Findet statt, wenn mehrere ACKNOWLEDGE-Pakete für dasselbe Segment empfangen werden. Damit kann sicher davon ausgegangen werden, dass dieses Segment verloren gegangen ist. In diesem Fall wird nicht auf einen Timeout für dieses Paket gewartet, stattdessen wird dieses sofort erneut gesendet. In den meisten Algorithmen wird ein "Fast-Retransmit" nach einem dreimal wiederholten Acknowledge durchgeführt. Die *Slow-Start-Threshold* wird nach einem Fast Retransmit auf den Wert des halbierten Congestion Windows gesetzt und das *Congestion Window* auf die neue *Slow Start Threshold* gesetzt und um Drei erhöht. Damit wird die "Slow Start" Phase übersprungen. Durch die Ankunft der ACKNOWLEDGE-Pakete kann davon ausgegangen werden, dass es in dem Netzwerk nur einen temporären Engpass gab.

Fast Recovery: Der "Fast Recovery"-Algorithmus wird verwendet, wenn nach einem "Fast Retransmit" noch weitere duplizierte Acknowledge-Pakete ankommen. Dabei werden, wenn noch weitere Daten zum Senden bereitstehen, diese gesendet, da davon ausgegangen wird, dass nur das Paket, das im "Fast-Retransmit" gesendet wurde verloren gegangen ist und die Datenübertragung ohne große Geschwindigkeitseinbußen fortgesetzt werden kann.

Timeout: Wenn die Verbindung nicht mit den "Fast-Retransmit" und "Fast-Recovery" Algorithmen weitergeführt werden kann, oder erst gar keine Acknowledge-Pakete ankommen, muss von einer größeren Überlastung oder Änderungen auf dem Übertragungsweg ausgegangen werden. Nachdem ein Paket einen bestimmten Zeitraum lang nicht bestätigt wurde wird dieses nochmal gesendet. In dem Fall wird die *Slow Start Threshold* ebenfalls auf die Hälfte des *Congestion-Windows* gesetzt. Das *Congestion-Window* selber wird dabei jedoch auf Eins gesetzt und die Übertragung mit dem "Slow-StartÄlgorithmus fortgesetzt.[cc]

2.4.8 Verbindungsabbau

Bei dem Beenden einer Verbindung muss sicher gestellt werden, dass eine Verbindung abgebaut werden kann, ohne dass es zu einen Datenverlust kommt. Ein Host, der keine Daten mehr zum

Senden hat, kann die Close-Operation ausführen. Dabei sendet dieser ein Paket mit den FIN-Flag gesetzt, was der Gegenseite signalisiert, dass der Host die Verbindung beendet möchte. Er lässt die Verbindung aber weiterhin offen für eingehende Datenpakete, bis die Gegenstelle signalisiert, dass sie bereit ist, die Verbindung zu beenden. Die Gegenstelle reagiert auf das FIN-Paket entweder mit einem ACK-Paket, wenn sie noch Daten zu übertragen hat, oder mit einem FIN-Paket, wenn sie keine Daten mehr zu übertragen hat und die Verbindung beendet werden kann. [TCP]

2.5 Dynamic Host Control Protocol (DHCP)

Mit größeren lokalen Netzwerken mit wechselnden Teilnehmern kam der Bedarf nach einer zentralen Einrichtung, welche die Verteilung der IP-Adressen innerhalb eines Netzwerkes verwaltet. Dafür wurde aufbauend auf den älteren "Bootstrap-Protocol" DHCP entwickelt. DHCP ist daher weitgehend kompatibel mit Bootstrap, weswegen mit Bootstrap Clients und Servern zusammengearbeitet werden kann. Neben der IP-Adresse können auch andere Parameter abgefragt werden. Zum Beispiel Gateway, Netzmaske, Zeitserver und Nameserver. [DHC]

2.5.1 Paket Format DHCP (Bootstrap)

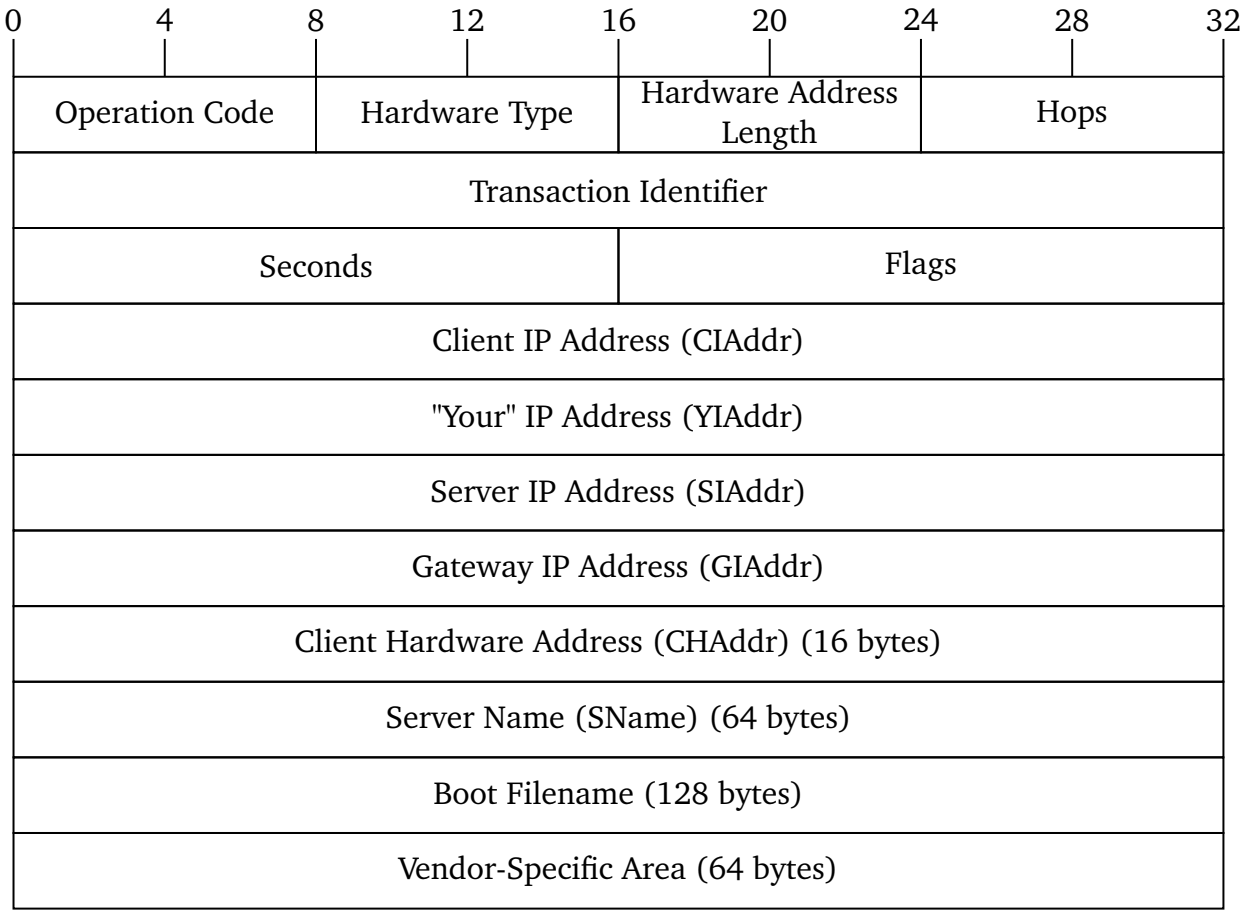


Abbildung 2.7.: BOOTP Message Format
[DHC]

–

Op: Gibt den generellen Typ der Nachricht an. Ein Wert von Eins signalisiert eine Anfrage, wobei eine Zwei eine Antwort signalisiert.

HType: Spezifiziert die Art der dem Netzwerk zugrunde liegende Hardware. Der Code 1 steht dabei für Ethernet.

HLen: Bezeichnet die Länge der physischen Adressen im Netzwerk. Im Falle von Ethernet Netzwerken sind dies sechs Byte.

Hops: Wird beim Absenden auf Null gesetzt und von jedem "Relay Agent" um Eins erhöht.

XID: Eine Identifikationsnummer, mit der die Antworten des Servers den Anfragen zugeordnet werden können.

Secs: Dieses Feld ist für das Bootstrap Protokoll ungenau definiert und wird oft nicht verwendet. Für DHCP wird es für die Zeit in Sekunden genutzt die vergangen ist, seit der Client damit begonnen hat nach einen neuen "Lease" zu fragen.

Flags: Acht Bits, die als Flags genutzt werden können. Das erste Bit wird gesetzt, falls die Nachricht als Broadcast gesendet wird, was dem DHCP Server signalisiert, dass dieser über keine gültige IP Adresse verfügt.

CIAddr: Steht für "Client-IP-Address" Der Client schreibt seine eigene IP-Adresse in dieses Feld. Im Falle von DHCP kann dieses nur in den Zuständen BOUND, RENEWING oder REBINDING genutzt werden. In allen anderen Fällen bleibt dieses Feld auf Null.

YIAddr: Abkürzung für "Your IP Address". Wird vom Server gesetzt um den Client eine IP-Adresse zuzuweisen.

SIAddr: Der Server gibt hier die Adresse des Servers an, dem der Client seine nächste Anfrage schicken soll.

GIAddr: Wird nur im "Bootstrap Protocol" für die Kommunikation zwischen Client und Server verwendet, wenn diese nicht im selben Netzwerk oder Subnet liegen. Wird im DHCP nicht dafür verwendet, das Standardgateway anzugeben.

CHAddr: Hardware Adresse des Clients. Bei Nutzung von Ethernet die Mac-Adresse.

SName: Der DHCP-Server kann hier optional seinen Namen angeben. Alternativ kann dieses Feld durch die "Option overload" Funktion auch für weitere Optionen genutzt werden.

File: Kann genutzt werden, um bei einem DHCPDISCOVER oder OFFER eine Bootdatei anzugeben.

Options: Für das Bootstrap-Protokoll gibt es eine große Menge an Optionen, einige davon werden exklusiv für DHCP genutzt.
Die Anzahl und Art der verwendeten Optionen ist variabel. Der Aufbau von diesen folgt jedoch einem definierten Format. Die ersten acht Bit eines Optionsblocks enthalten den Code der Option, weitere acht Bit geben die Länge des folgenden Datenfeldes an. Das Optionfeld beginnt im Falle von DHCP mit der "Magic Number" 99.130.83.99. Nur dann können die exklusiven DHCP-Optionen genutzt werden.[DHC] Dazu gehören:

Requested IP Address: Diese Option kann während eines DHCP-Discovers gesetzt werden, um die Verfügbarkeit einer bestimmten IP-Adresse anzufragen.
Code: 50; Länge: 4 Byte;

IPAddress Lease Time: Kann in einer DHCP-Request oder DHCP-Offer Nachricht gesetzt werden. Der Client kann dabei eine bestimmte Lease-Time in Sekunden angeben.
Code: 51; Länge: 4 Byte;

Option Overload: Diese Option signalisiert, dass Option Overload genutzt wird. Das bedeutet, dass weitere Optionen anstelle der Felder SNAME oder FILE geschrieben werden.
Code: 52; Länge: 1 Byte; Es gibt drei mögliche Werte, die geschrieben werden können:

- 1: Optionen stehen im File-Feld.
- 2: Optionen stehen im SNAME-Feld.
- 3: Optionen stehen in beiden Feldern.

DHCP Message Type Dieses Feld ist bei DHCP-Nachrichten immer vorhanden und gibt die Art der DHCP-Nachricht an.
Code: 53; Länge: 1 Byte;

- 1: DHCPDISCOVER
- 2: DHCPOFFER
- 3: DHCPREQUEST
- 4: DHCPDECLINE
- 5: DHCPACK
- 6. DHCPNAK
- 7: DHCPRELEASE
- 8: DHCPINFORM

Server Identifier: Die Option kann von einem Client bei einer DHCP-Request gesetzt werden, um bei Unicast-Nachrichten die Adresse von einem bestimmten Server anzugeben. Ein DHCP-Server kann dieses Feld setzten, damit ein Client mehrere DHCP-Offer unterscheiden kann.
Code: 54; Länge: 4 Byte;

Parameter Request List: Es kann eine Liste von Parametern übergeben werden um die Werte von diesen bei einem Server anzufragen. Dazu kann eine beliebig lange Folge aus DHCP/Bootstrap Optionscodes genutzt werden.
Code: 55; Länge: n Byte;

2.5.2 Ablauf

Zu Beginn eines DHCP-Vorgangs, wenn zum Beispiel ein Client hochgefahren wird, verfügt dieser weder über eine IP-Adresse, noch über andere Information die Netzwerktopologie betreffend. Als Erstes sendet der Client ein Discover-Paket. Dafür wird der Operationscode auf 1 gesetzt um ein Request-Paket zu signalisieren, die MAC-Adresse wird für die physische Client

Adresse verwendet, das Broadcast-Flag wird gesetzt und der DHCP-Message-Type wird auf Discover gesetzt. Aus diesem DHCP-Paket wird ein UDP-Paket erzeugt, mit dem Quellport 68 und dem Zielport 67. Die Nachricht wird als Broadcast gesendet.
Auf diese Anfrage können einer oder mehrere DHCP-Server antworten. Wenn kein bestimmter Server in der DISCOVER-Nachricht spezifiziert wurde, schicken alle Server in dem jeweiligen Netzwerk ein "OFFER". Diese Nachricht wird ebenfalls als Broadcast gesendet, da der Client noch keine eigene IP zugewiesen hat. Das OFFER enhält einen Vorschlag mit einer IP-Adresse in dem Feld "Your-IP-Address". Anhand der "Transaction ID" und dem Feld CHADDR kann der Client die Offer eindeutig seiner Anfrage zuordnen.
Der Client antwortet auf das "OFFER" mit einer "REQUEST"-Nachricht. Dafür wird die Option "Requested-IP-Address" gesetzt.
Auf das Request antwortet der Server wiederum mit einem ACK zu Bestätigung. Wenn der Client die Bestätigung erhält, prüft dieser, ob die IP-Adresse schon genutzt wird, in dem er eine ARP-Request für diese startet. Wird diese beantwortet, ist die Adresse schon in Benutzung. Ist dies nicht der Fall, übernimmt der Client die Adresse. Die ACK Nachricht enthält die Lease-Zeit, welche angibt, wie lange eine IP gültig ist. Nachdem die Hälfte der Lease-Zeit abgelaufen ist, sendet der Client eine weitere REQUEST-Nachricht. Da er zu diesem Zeitpunkt noch über eine gültige IP-Adresse verfügt, sendet er die Nachricht nicht als Broadcast, sondern als Unicast direkt an den DHCP-Server, von dem er die IP zugewiesen bekommen hat. In dem Fall, dass er auf den Request keine Antwort bekommen hat, startet er nach Ablauf der Lease Zeit mit einem DISCOVER.
Während dieses Vorgangs können neben der IP auch noch andere Informationen abgefragt werden. Wie beispielsweise die Adressen von DNS- und Zeitservern.[DHC]

3 AMIDAR Prozessor

Bei der Klasse der AMIDAR-Prozessoren handelt es sich um rekonfigurierbare Systeme, die zur Laufzeit an die Anwendung angepasst werden können. Sie bestehen jeweils aus mehreren "Function Units", die über ein Token- und ein Datennetzwerk verbunden sind. Ein Token stellt dabei eine Mikroinstruktion dar. Diese werden durch die Tokenmaschine aus den Prozessor-Instruktionen generiert. Dazu kommen FUs für Framestack, Heap, ALUs und Hardwarebeschleuniger. Bei den in Hardware implementierten Prototypen des Fachgebiets Rechnersysteme wird Java-Bytecode eingesetzt.

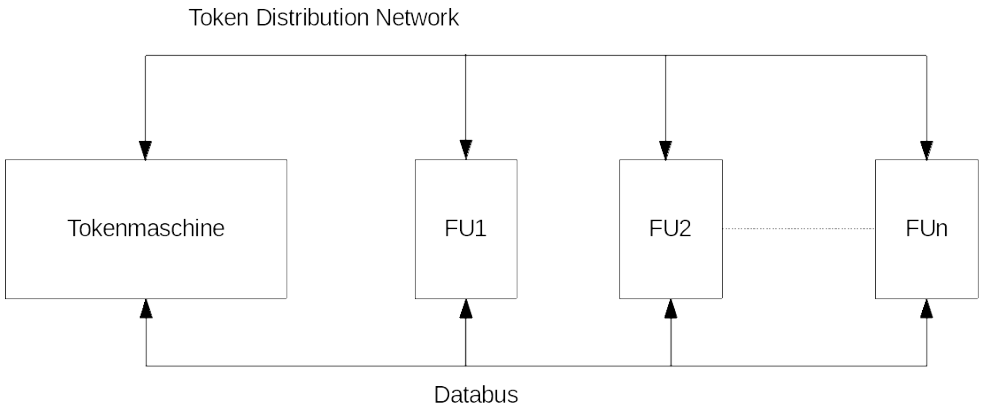


Abbildung 3.1.: AMIDAR Aufbau

4 Implementierung des TCP/IP Stacks

Im Rahmen dieser Arbeit wurde ein TCP Stack für die API des AMIDAR Microprozessor entwickelt. Darüber hinaus wurde der schon vorhandene IP-Stack angepasst um die gestellten Anforderungen erfüllen zu können. Zusätzlich wurde auch die Unterstützung für DHCP hinzugefügt.

4.1 Überblick

Vor Beginn dieses Projekts verfügte die AMIDAR-Java-API über grundlegende Netzwerk Funktionen. Dazu gehört der Netzwerktreiber, ein IP-Stack mit ARP-Funktionalität und ein UDP-Stack. Neu geschrieben wurde im Rahmen dieses Projekts der multithreading-fähige TCP-Stack. Dieser wurde in die vorhandene Software integriert. Des Weiteren wurde der IP-Stack erweitert und optimiert.

Sowohl beim Senden als auch beim Empfangen von Datenpaketen greifen die einzelnen Module ineinander über. Zum Empfangen von Daten prüft der Prozess des Netzwerktreibers ob neue Ethernet-Frames vorliegen. Wenn dies der Fall ist, wird eine Funktion im IP-Stack aufgerufen, die die Ethernet-Frames überprüft. Der IP-Stack unterscheidet die Pakete zwischen ARP und IP. ARP-Anfragen werden geprüft und gegebenenfalls beantwortet. Handelt es sich bei dem Datagramm um ein IP-Paket, wird ein entsprechendes Objekt erzeugt und nach weiterer Überprüfung entweder an den UDP-Stack oder an den TCP-Stack übergeben. Die Stacks für TCP und UDP beinhalten jeweils einen Table mit den vorhandenen Verbindungen, die durch Ziel- und Quell-Port identifiziert werden können. Ihnen können gegebenenfalls die erzeugten Pakete weitergegeben werden, wo sie zwischengespeichert werden. Im Falle von UDP wird die Payload ausgelesen, wenn die *receive*-Methode der UDP-Connection, von einem anderen Thread aufgerufen wird. Die TCP-Connections können jeweils in ihren eigenen Thread laufen, da eine zeitnahe Verarbeitung der angekommen Pakete nötig ist, um die Verbindung zu managen. In diesem Thread werden die angekommenen Pakete ausgewertet und die dazu entsprechenden Reaktionen berechnet und ausgeführt.
Wenn UDP Datenpakete versendet werden sollen, wird die *send*-Methode aufgerufen, die ein UDP-Paket erzeugt und dieses an den UDP-Stack weitergibt. Der wiederum erzeugt aus dem UDP-Paket ein IP-Paket, mit dem die *send* Methode des IP-Stacks aufgerufen wird. Die einen Ethernetframe erzeugt und den Sendevorgang im Netzwerktreiber startet.

Bei dem Senden von Daten über TCP wird von der Anwendung ebenfalls die *send* Methode der TCP-Connection aufgerufen. Die Daten werden dabei jedoch nicht sofort gesendet, sondern in einen Puffer zwischengespeichert, vorausgesetzt der aktuelle Status der Verbindung erlaubt das. Bei Ausführung des Threads der Verbindung werden gegebenenfalls die zu übertragenden TCP-Pakete in IP-Pakete umgewandelt und analog wie die UDP-Pakete versendet.

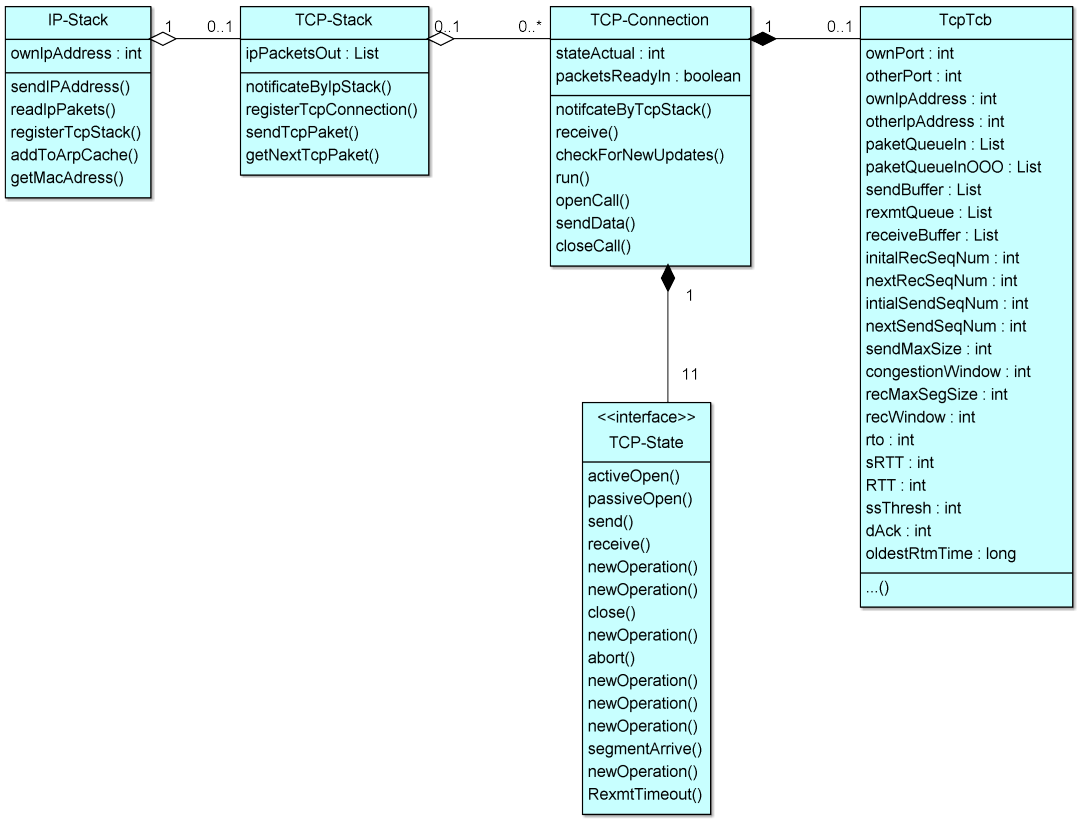


Abbildung 4.1.: Auszug Klassendiagramm TCP/IP

4.2 IP Stack

Der IP-Stack erfüllt mehrere Funktionen, die für eine zuverlässige Netzwerkkommunikation benötigt werden. Dazu gehört das Senden und Empfangen von IP-Pakten, ebenso wie die Unterstützung der ARP Funktionalitäten.

4.2.1 IP Paket

Für die Repräsentation von IP-Paketen wird die Klasse *IpPacket* genutzt. Diese enthält die Daten des Pakets in einem Array. Neben Methoden, welche die einzelnen Felder des Header zurückgeben, gibt es Konstruktoren, die als Parameter TCP-, UDP- oder Ethernet-Pakete entgegennehmen. Des weiteren gibt es Methoden zur Berechnung der Checksumme und der Umwandlung von String Repräsentationen von IP Adressen zu Integer-Werten.

4.2.2 Empfangen von IP Paketen

Die Methode *readIpPakets* des IP-Stacks wird vom Netzwerktreiber Thread aufgerufen. In dieser werden in einer Schleife die angekommenen Ethernet-Pakete eingelesen und die IP-Pakete dazu erzeugt. Dabei werden im Zweifelsfall Teile von fragmentierten Paketen zwischengespeichert, bis diese vollständig sind.

Bei den so erzeugten IP-Paketen wird das darüber liegende Protokoll ausgelesen.

Damit UDP- und TCP-Pakete zugestellt werden können, müssen die jeweiligen Stacks im IP-Stack registriert werden. Nach der Registrierung können angekommene Pakete dem jeweiligen Stack zugeordnet werden. Dafür implementieren beide Stacks die Funktion *notificateByIpStack* der eine Liste mit angekommenen UDP-Paketen übergeben wird.

Fragmentierung

IP-Pakete können während der Übertragung fragmentiert werden, um über Netzwerke mit einer zu niedrigen maximalen Segmentgröße übertragen zu werden. Diese werden vom Empfänger defragmentiert.

Fragmentierte Pakete können nach dem Erzeugen erkannt werden, indem das entsprechende Flag ausgelesen wird. Das Flag gibt nicht explizit an, dass dieses Paket ein Teil eines größeren fragmentierten Pakets ist, sondern dass sie ein Teil eines fragmentierten Pakets ist und weitere Paket Fragmente folgen. Das bedeutet, dass das letzte Fragment eines Pakets nicht das Flag gesetzt hat. Dieses kann man daran erkennen, dass der Fragment-Offset ungleich Null ist.
Der IP-Stack enthält eine Liste von fragmentierte Paketen, die Listen mit jeweils zusammengehörigen Fragmenten enthalten. Wann immer ein Paket ankommt, bei dem die "More Fragments" Flag gesetzt ist oder der Data-Offset ungleich Null ist, wird geprüft, ob dieses zu einem der fragmentierten Pakete gehört, und wird in diesem Fall in eine der Listen hinzugefügt. Die Zugehörigkeit wird dabei anhand der Identification des IP Pakets geprüft.
Falls ein Paket ohne Flag einem anderen fragmentierten Paket zugeordnet werden kann, ist dieses das letzte Fragment des ursprünglichen IP-Pakets, mit dem das kombinierte Paket erzeugt werden kann.

Für diesen Zweck verfügt die Klasse *IpPaket* über die statische Methode *fuseFragmentedIpPackets*, die eine Liste von zusammengehörigen IP-Paket-Fragmenten annimmt. Diese werden auf Kompatibilität geprüft, wobei auch die Größe der kombinierten Nutzlast berechnet wird. Die Fragmente müssen neben der identischen Identifikation über die selbe Quell- und Ziel-IP-Adresse verfügen. Des Weiteren müssen die Angaben des Frame-Offsets mit der Paketlänge plausibel sein.
Anschließend wird ein Array der entsprechenden Länge erzeugt und die Nutzlast der Pakete anhand des Frame-Offsets zusammengesetzt.
Danach kann der Konstruktor der IP-Paket-Klasse aufgerufen werden, wobei ihm die neu kombinierte Nutzlast übergeben wird. Das neu erzeugte Paket wird zurückgegeben und von dem IP-Stack weiterverarbeitet.

4.2.3 Senden von IP Paketen

Der IP-Stack verfügt über die Methode *sendIpPaket*, die im Normalfall von dem TCP- oder dem UDP-Stack aufgerufen wird. Diese nimmt ein übergebenes IP-Paket an und generiert daraus einen Ethernet-Frame. Daraufhin wird das Paket über das Ethernet versendet.

4.2.4 Address Resolution Protokoll (ARP)

Das ARP-Protokoll wird verwendet, um in einem lokalen Netzwerk die IP Adressen zu den physischen MAC-Adressen aufzulösen. Ein Host kann eine ARP-Request als Broadcast verschicken, um die MAC-Adresse zu erfahren, über die die Netzwerkressource mit der IP-Adresse zu erreichen ist.

Der IP-Stack verfügt über die Möglichkeit, ankommende ARP-Pakte zu verarbeiten. Für den Fall, dass es sich bei einem ankommenden Datenpaket anstatt eines IP-Pakets um ein ARP-Paket handelt, wird geprüft, ob es sich um eine "Request" oder eine "Response" handelt. Falls es sich um eine Request handelt, wird in einer entsprechenden Antwort die eigene physische MAC-Adresse verschickt.

Beim Versenden eines IP-Pakets muss ein Ethernet-Paket erzeugt werden. Dafür wird die MAC-Adresse unter der das Ziel oder die nächste Station auf dem Weg zu diesem erreichbar ist, benötigt. Für diesen Zweck verfügt der IP-Stack über eine Hash-Map, die als ARP-Cache dient. In dem Fall, dass für die IP kein Eintrag mit MAC-Adresse vorliegt, wird eine ARP-Request gesendet und auf deren Antwort gewartet. Nachdem die ARP-Response eingetroffen ist, wird die IP-Adresse mit der MAC-Adresse verknüpft und in dem ARP-Cache gespeichert.

4.3 TCP

Vor der Implementierung des TCP-Stacks wurde überprüft, ob es schon in Java implementierte TCP-Stacks gibt. Die einzige in Frage kommende Implementierung war der "ejlP"-TCP-Stack. Dieser verfügt jedoch über keinen Mechanismus für die erneute Übertragung verlorener Pakete, die von dem darüber liegenden Protokoll gewährleistet werden müsste. Aus diesem Grund wurde entschieden, eine eigene TCP-Variante zu schreiben, die keine umfassende Änderung des IP-Stacks und des Treibers erforderte.[Sch]

4.3.1 TCP Stack

Der TCP-Stack stellt den Datenaustausch zwischen den einzelnen TCP-Verbindungen und dem IP-Stack her und ist dabei in der Lage, mehrere Verbindungen parallel offen zu haben und diese einzeln zu verwalten. Er enthält eine Map mit allen angelegten TCP-Verbindungen. Es gibt eine Funktion, um neue TCP-Verbindungen zu registrieren. Eine der wichtigsten Methoden ist textitnotificateByIpStack, die vom IP-Stack aufgerufen wird und eine Liste von TCP-Paketen übergibt. Der TCP Stack untersucht diese und überprüft sie anhand der Checksumme auf Korrektheit. Fehlerhafte Pakete werden aussortiert. Fehlerfreie werden abhängig von der Ziel-Port-Nummer

an eine der TCP-Verbindungen übergeben oder verworfen, wenn diese keiner Verbindung zugeordnet werden können.

4.3.2 TCP Verbindung

Die Klasse *TcpConnection* verwaltet die Zustände und Datenströme einer Verbindung. Die Klasse stellt zum einen die Verbindung zu der Anwendung dar und zum anderen enthält sie den Zustandsautomaten, der die Abläufe bei dem Aufbauen und Aufrechterhalten der Verbindung managt. Des Weiteren befindet sich in dieser der Transmission-Control-Block (TCB), der nach dem Öffnen einer Verbindung die meisten für diese Verbindung spezifischen Variablen enthält.

4.3.3 TCP-Paket

TCP-Pakete werden von einer Klasse repräsentiert, welche die Daten des Paktes in einem Array speichert, auf dessen Datenfelder mit entsprechenden Methoden gezielt zugegriffen werden kann. Außerdem gibt es Methoden zur Berechnung der Prüfsumme.

Struktur Zustandsautomat

Eine TCP-Verbindung muss auf eine Reihe von Ereignissen abhängig von der aktuellen Situation reagieren, dabei bietet sich ein Zustandsautomat an, der jeden Zustand durch eine eigene Klasse abstrahiert. Da es bei der momentan von AMIDAR unterstützen JAVA-Version 1.4 noch keine Unterstützung für Enumerationen gibt, werden die einzelnen Zustände in einem Array von *TcpClass* gespeichert. *TcpClass* ist dabei das Interface, das alle Statusklassen implementiert. Der aktuelle Zustand wird jeweils von der Integer-Variable *actualState* angegeben, welche die Position im Zustandsarray angibt. Falls eine der zustandsabhängigen Methoden aufgerufen werden soll, wird diese auf dem Array aufgerufen, wobei *actualState* als Index genutzt wird. Das von den Zustandsklassen implementierte Interface enthält folgende Methoden:

activeOpen: Öffnet eine Verbindung und versucht mit dem übergebenen Port und der IP-Adresse den Drei-Wege-Handshake zu initiieren.

passiveOpen: Öffnet eine Verbindung, auf die von anderen Geräten ein Verbindungsaufbau initiiert werden kann.

send: Schreibt übergebene Daten in den Sende-Puffer des TCP-Stacks und erzeugt gegebenenfalls ein neues TCP-Paket, welches die übergebenen Daten verschickt.

receive: Überprüft, ob genug Daten vorhanden sind, um diese an die Anwendung zu übergeben.

close: Signalisiert, dass alle Daten übertragen wurden und die Verbindung abgebaut werden kann.

abort: Bricht die Verbindung ab.

segmentArrive: Untersucht und verarbeitet ein ankommendes TCP-Paket.

RexmtTimeout: Wird aufgerufen, nachdem ein Timeout für ein Paket in der *Retransmit Queue* aufgetreten ist.

Jede dieser Methoden gibt als Integer den nächsten Zustand zurück. *Active Open*, *Passive Open*, *send*, *receive*, *close* und *abort* werden durch Aufrufe der darüber liegenden Anwendung ausgelöst. *Segment Arrive* wird aufgerufen, nachdem vom TCP-Stack angekommene Pakete übergeben wurden und der Thread der Verbindung ausgeführt wird. *RexmtTimeout* wird aufgerufen, wenn der Retransmit-Timer für ein Paket ausläuft.

Neue Übertragung verlorener Pakete

Jedes gesendete Paket wird mit einem Zeitstempel versehen und der *Retransmit-Queue* hinzugefügt. Der Zeitstempel des ältesten Paket in der *Retransmit-Queue*, wird als Basis genommen, um zu bestimmen, ob ein Timeout vorliegt. Die vergangene Zeit, bis ein Retransmit ausgelöst wird, wird nicht willkürlich gewählt, da so bei Verbindungen mit hoher Latenz Pakete neu versendet würden, die nicht verloren gegangen sind. Diese Zeit, RTO genannt, wird abhängig von der Round-Trip-Time (RTT) berechnet. Die Round-Trip-Time gibt die Zeit an, die vergeht, nachdem ein Paket übergeben wurde, bis das Acknowledgement für dieses Paket eingetroffen ist. Um zu starke Schwankungen bei der RTO zu vermeiden, wird eine geglättete Round Trip Time verwendet.

$$SRTT = (\alpha * SRTT) + ((1 - \alpha) * RTT)$$
$$RTO = min[u, max[o, (\beta * SRTT)]]$$

Der Glättungsfaktor α wird zwischen 0,8 und 0,9 festgelegt. β ist ein Varianzfaktor und wird auf einen Wert zwischen 1,3 und 2,0 gesetzt.Die Variablen "u" und "o" stellen die obere und untere Grenze dar und werden beispielsweise auf eine Sekunde bzw. eine Minute gesetzt.

Transmission-Control-Block(TCB)

Der Transmission-Control-Block enthält alle für die Verbindung wichtigen Variablen, abgesehen von der Zustandsvariable. Dazu gehören der eigene Port und der Port des Zielrechners, die eigene IP-Adresse und die der Gegenstelle. Darüber hinaus werden die Sequenznummern sowohl zum Empfangen als auch zum Senden gespeichert. Des Weiteren finden sich dort alle Variablen, die zur "Congestion-Contol" benötigt werden.

Zustände im Detail

Sofern nicht anders beschrieben, löst ein Timeout eines Pakets eine neue Übertragung von diesem aus, wobei es am Ende der *Retransmission-Queue*, mit einem neuen Zeitstempel gespeichert wird. Wenn ein Paket nach dem 5. Timeout noch nicht angekommen sein sollte, wird davon ausgegangen, dass keine nutzbare Verbindung zum Host mehr existiert und die Verbindung wird abgebrochen. Bei allen weiteren in den jeweiligen Zuständen nicht näher beschriebenen Aktionen wird eine *IOException* geworfen.

CLOSED: In diesen Zustand wird bei einem *activOpen* ein Verbindungsaufbau gestartet. Dafür wird die initiale Sequenznummer zufällig generiert und ein SYN-Paket erzeugt. Anschließend wird die *SendUnacknowledged*-Nummer auf den Wert der initialen Sequenznummer

gesetzt und der Wert für die nächste Sendesequenznummer auf die um Eins erhöhte intiale Sequenznummer gesetzt. Des Weiteren wird der nächste Zustand auf SYN-SENT gesetzt.
Falls die Methode *passivOpen* aufgerufen wird, wird in den Zustand LISTEN gewechselt.
In beiden Open Methoden wird die Verbindung im TCP-Stack registriert und es wird ein Transmission-Control-Block angelegt.
Alle anderen Usercalls werden mit Fehlermeldungen beendet.

LISTEN: Im Zustand LISTEN wird auf einen Verbindungsaufbau von einem externen Host gewartet. Ein *activeOpen*-Usercall kann auch in diesem Zustand noch ausgeführt werden, mit demselben Ablauf wie im Zustand CLOSED. Bei einem *receive* Aufruf können noch keine Daten zurückgegeben werden, da keine aktive Verbindung aufgebaut ist. Der *Receive*-Usercall wird jedoch zwischengespeichert. Die Calls *close* und *abort* löschen den TCB und geben den Zustand CLOSED zurück. Bei eintreffenden Paketen wird geprüft, ob es sich um ein SYN-Paket handelt. In diesem Fall wird die Initiale Sequenznummer berechnet und ein SYN-ACK Paket in die Paket-out-Queue abgelegt. Daraufhin geht die Verbindung in den Zustand SYN-RECEIVED über. Alle anderen eintreffenden Pakete deuten auf eine Fehlfunktion der Verbindung der Gegenstelle hin und werden mit einen RST-Paket beantwortet.

SYN-RECEIVED: In diesem Zustand werden Aufrufe der *send*- und *receive*- calls zwischengespeichert, um in einem späteren Zustand ausgewertet zu werden. Bei Ausführung eines *close*-Aufrufs wird geprüft, ob der Sendepuffer leer ist. Wenn das nicht der Fall ist, verbleibt die Verbindung im aktuellen Zustand. Andernfalls wird ein FIN-Paket gesendet, um das Ende der Verbindung einzuleiten. Wird die "abort"-Funktion aufgerufen, wird ein RST-Paket gesendet, der TCB gelöscht und in den Zustand CLOSED zurück gegangen.

Ankommende Pakete werden nach Überprüfung der Gültigkeit anhand der Sequenznummer ausgewertet. Ist das angekommene Paket ein ACK-Paket, wird das bestätigte Paket aus der *Retransmit-Queue* entfernt und in den Established Zustand gewechselt.
Falls ein weiteres SYN- oder ein RST-Paket ankommt, wird die Verbindung beendet.

SYN-SEND: Die Aktionen für *send*, *receive*, *close* und *abort* werden genauso gehandhabt, wie im Zustand SYN-RECEIVED. Nach dem Eintreffen von gültigen SYN-ACK Paketen wird in den Zustand ESTABLISHED gewechselt. Wenn ein eintreffendes Paket nur ein SYN enthält wird der Zustand SYN-RECEIVED zurück gegeben und ein SYN-ACK-Paket gesendet. Bei einem RST-Paket wird die Verbindung abgebrochen. Alle anderen Pakete werden verworfen.

ESTABLISHED: In diesem Zustand beginnt die *send* Operation mit dem Speichern der übergebenen Daten in den Sendepuffer. Anschließend wird das maximale Limit an Daten bestimmt, die noch versendet werden dürfen. Dafür wird das Minimum von *Congestion Window*, und *Receiver-Window* genutzt. Anschließend wird ein Datenpaket erzeugt, wobei darauf geachtet wird, das sowohl dass Limit, als auch die maximale Segment-Größe nicht überschritten wird.

Ankommende Datenpakete werden einsortiert. Entspricht die Sequenznummer des Pakets der erwarteten Sequenznummer, können die Daten des Pakets in den Empfangspuffer geschrieben werden. Ist die Sequenznummer über der erwarteten Nummer, werden die

Pakete in den "Out-of-Order" Puffer zwischengespeichert. Diese werden, wenn das fehlende Paket ankommt, hinter dem neuen Paket in den Empfangspuffer einsortiert. Jedes ankommende Paket wird mit einem ACK-Paket quittiert. Dabei ist zu beachten, dass bei Paketen, die nicht in der erwarteten Reihenfolge ankommen, die Sequenznummer des erwarteten Pakets als Acknowledge-Nummer gesendet wird. Dadurch wird der Gegenseite signalisiert, dass dieses Paket möglicherweise verloren gegangen ist. Dadurch enstehen duplizierte ACKs, die für "Fast Retransmit" und " Fast Recovery" essentiell ist.

Des Weiteren wird das mit dem Paket mitgeschickte ACK überprüft. Entspricht die Acknowledge-Nummer der letzten nicht bestätigten gesendeten Sequenznummer, wird ein Zähler um eins hoch gesetzt um duplizierte ACKs zu erkennen. Bei einem mehr als dreimal duplizierten ACK wird ein "Fast Retransmit" durchgeführt. Dabei wird die *Slow-Start-Threshold* halbiert und die Größe des *Congestion-Windows* auf den Wert von diesem gesetzt. Zusätzlich wird das verlorene Paket aus der *Retransmit-Queue* herausgesucht und sofort neu übertragen. Wenn ein gültiges, nicht dupliziertes ACK eintrifft, werden alle bestätigten Segmente aus der *Retransmit-Queue* entfernt und das *Congestion-Window* entsprechend der "Slow Start" und "Congestion Avoidance" Algorithmen erhöht. Falls das FIN-Flag bei einem gültigen Paket gesetzt ist, wird der Zustand CLOSE WAIT zurück gegeben.

Falls es zu dem Timeout eines Pakets kommt, wird die *Slow Start Threshold* ebenfalls halbiert und das *Congestion-Window* auf den initialen Wert gesetzt.

FIN-WAIT-1: In diesem Zustand wird auf ankommende Pakete mit derselben Art reagiert wie im Zustand ESTABLISHED, mit dem Unterschied, dass nach einem ankommenden gültigen ACK-Paket in den Zustand FIN-WAIT-2 gewechselt wird. Ist zusätzlich auch das FIN-Flag gesetzt, wird in den Zustand TIME-WAIT übergegangen. Dazu kommt, dass bei duplizierten ACK-Paketen kein "Fast-Retransmit" mehr durchgeführt wird.

FIN-WAIT-2: Ankommende Pakete werden hier verarbeitet wie in der ESTABLISHED Phase mit dem Unterschied, dass bei einen FIN-Paket der Zustand TIME-WAIT zurückgegeben wird.

CLOSE-WAIT: In diesem Zustand funktioniert ein *send* Call genauso wie im Zustand ESTABLISHED. Bei ankommenden Segmenten werden nur noch ankommende ACK-Pakete verarbeitet, da die Gegenstelle keine Daten mehr senden wird. Nach der Ankunft von SYN- oder RST-Paketen wird die Verbindung beendet.
Nach einen Aufruf von *close* Call wechselt die Verbindung in den Zustand LAST-ACK und sendet ein FIN-Paket.

LAST-ACK: In diesem Zustand lösen alle Usercalls bis auf *Segment Arrive* eine *IOException* aus. Ankommende gültige Pakete werden daraufhin untersucht, ob diese das ACK zu dem gesendeten FIN-Paket sind. Ist dies der Fall, wird die Verbindung beendet und in den Zustand CLOSED übergegangen. Bei der Ankunft eines RST- oder SYN-Pakets wird die Verbindung abgebrochen und ebenfalls der Zustand CLOSED zurückgegeben.

CLOSING: Der Zustand CLOSING verhält sich weitgehend identisch zum Zustand LAST-ACK, mit dem Unterschied, dass in den Zustand TIME-WAIT übergegangen wird, wenn ein ankommendes Paket das ACK für das gesendete FIN-Paket enthält.

TIME-WAIT: In diesem Zustand ist der Verbindungsabbau abgeschlossen. Die Verbindung bleibt dennoch ein paar Minuten offen um stark verzögerte Pakete zu bestätigen. Das Eintreffen von RST- und SYN-Paketen beendet die Verbindung sofort.

Options

TCP-Optionen werden in einem optionalen Bereich mit variabler Länge des Headers abgelegt. Um diese zu repräsentieren, wurde eine Klasse erstellt, um den Optionsbereich des TCP-Headers abzubilden. Dafür wurde eine Methode geschrieben, um die Optioncodes in eine übersichtliche Datenstruktur zu parsen. Zusätzlich gibt es eine Methode, um Optionen in eine Datenstruktur zu überführen und daraus den Optionsblock im TCP Header zu generieren. Es wurde jedoch keine Unterstützung für konkrete Optionen implementiert.

4.4 Zero Copy

Die TCP- und IP-Pakete werden jeweils durch eine entsprechende Klasse repräsentiert. Die Objekte dieser Klasse enthalten jeweils ein Array, das Header und Nutzdaten des Pakets enthält. Beide Klassen verfügen über einen Konstruktor, der jeweils die andere der beiden Klassen als Parameter entgegennimmt. So kann beim Senden aus einem TCP-Paket ein IP-Paket erzeugt werden und beim Empfangen aus einen IP-Paket ein TCP-Paket. Dabei stellt das TCP-Paket die Nutzlast des IP-Pakets da. Um in diesem Fall lange Kopiervorgänge zu ersparen, wird es vermieden, ein neues Array anzulegen und die Daten hinein zu kopieren. Anstelle dessen wird in der TCP-Paket-Klasse im Array ein 20 Byte Puffer eingeplant, in den später der IP-Header geschrieben werden kann. Bei der Erzeugung eines IP-Pakets aus einen TCP-Paket wird das im TCP-Paket erzeugte Array weiterverwendet. Da die ersten 20 Byte leer sind, kann dort der IP Header geschrieben werden.

4.5 DHCP

Das Dynamic-Host-Control-Protocol verwendet UDP in der Transportschicht, um die Nachrichten zu senden. Bei der Implementierung muss eine Repräsentation der Pakete geschrieben werden um DHCP-Pakete verarbeiten zu können. Des Weiteren wird eine Klasse benötigt, die den Ablauf und die Kommunikation verarbeitet.

4.5.1 DHCP-Paket

Die Klasse *DHCPPacket* repräsentiet DHCP-Pakete. Diese enthält ein Integer-Array, welches die Paketdaten enthält. Es gibt Getter-Methoden, die mit Bitmaskierungen einzelne Datenfelder auslesen und deren Werte zurückgeben. Eine Schwierigkeit bei DHCP liegt darin begründet, dass die meisten wichtigen Informationen nicht in den festen Feldern, sondern in den variablen Optionsteilen abgelegt werden. Um diese auszulesen, musste ein Parser geschrieben werden, der den Optionsbereich einliest und nach dem übergebenen Optionscode sucht. Wenn dieser gefunden wird, werden anhand der folgenden Längenangaben die Optionsdaten ausgelesen. Beim Parsen muss beachtet werden, dass Optionen byteweise eingelesen werden müssen, obwohl diese als 32 Bit-Wörter vorliegen.

4.5.2 DHCP-Controller

Der DHCP-Controller enthält den Thread, welcher den DHCP-Client steuert. Für die Ablaufsteuerung wurde ebenfalls ein kleiner Zustandsautomat implementiert. Sobald der Thread gestartet wurde, wird eine DISCOVER-Nachricht als Broadcast gesendet und in den Zustand SELECTING gewechselt. In diesem Zustand wird auf eine OFFER-Nachricht des Servers gewartet. Die DHCP-Nachrichten werden identifiziert, indem der Optionscode 53 aus dem Optionsblock ausgelesen wird. Der Datenwert dieser Option gibt die Art des DHCP-Paketes an. Sobald ein DHCP-OFFER eintrifft, wird die vorgeschlagene IP mit einer REQUEST angefragt und in den Zustand REQUESTING gewechselt. Bei dieser Anfrage wird die IP-Adresse mit dem Optionscode 50 angefragt. Anschließend wird auf die ACK-Nachricht gewartet, in welcher die IP-Adresse bestätigt wird. Anschließend muss noch mit einer ARP-Request geprüft werden, ob diese Adresse schon von einem andern Host im Netzwerk verwendet wird.

4.6 Packet Builder

Ein Problem bei der Erzeugung der Paketobjekte war, dass jedes Mal eine Reihe von Parametern gesetzt werden muss, die bis auf wenige Fälle den selben Standardwert haben. Die müssen dennoch bei jeder Paket-Erzeugung gesetzt werden, was aufwändig und fehleranfällig ist. Um das zu vermeiden, können die Klassen für TCP- und DHCP-Pakete mit einen "Builder Pattern" erzeugt werden. Dabei wird eine spezielle Builder-Klasse genutzt, die als Attribute die Parameter der Paket-Klasse enthält. Diese werden vorinitialisiert. Über spezielle Setter-Methoden können die Werte gesetzt werden. Diese geben als Rückgabewert das Builderobjekt zurück. Der letzte Aufruf auf das Objekt ist jeweils die *build* Funktion, welche den Konstruktor der Paketklasse aufruft und dieses zurück gibt. Ein derartiger Aufruf sieht Beispielsweise wie folgt aus:

```
1 new TcpPaketBuilder(tcb).setSeqNum(NextSendSeqNum)
2 .setAckNum(NextRecseqNum).setAck().build()
```

Das erzeugen eines TCP-Pakets ohne die Verwendung des "Builder Patterns" sieht folgender maßen aus:

```
1 new TcpPaket(payLoad,length,tcpOptions,
2 connection, seqNum, ackNum, (short)winSize, flags ,urgP);
```

5 Evaluation

In diesem Abschnitt wird darauf eingegangen, unter welchem Szenario der TCP-IP Stack getestet wurde und welche Ergebnisse dabei heraus gekommen sind.

5.1 Testaufbau

Zur Evaluation wurde ein Nexys-Video-FPGA-Entwickler-Board verwendet. Auf diesem wurde Java-Prozessor der AMIDAR-Klasse synthetisiert. Als Netzwerkschnittstelle wurde eine Steckkarte mit zwei Ethernet RJ45 Schnittstellen verwendet. Diese wurde mit einem Patch-Kabel zu der Ethernet Schnittstelle eines Laptops verbunden. Der Laptop verfügt über eine Realtec Gigabit Ethernet Schnittstelle. Des Weiteren läuft auf dem Laptop Ubuntu 16.04 und ein "ISC-DHCPD" DHCP-Server. Für Testzwecke wurden außerdem diverse Java-Programme genutzt, welche die Java-Netzwerksockel nutzen.

5.1.1 Konfiguration des verwendeten AMIDAR-Systems

Für die Evaluation wurde ein System mit einer Taktrate von 100MHz verwendet. Dieses entspricht zum dem Zeitpunkt der Abgabe der neusten verfügbaren Konfiguration. Einschließlich des neuen Bus-Systems und Heaps, der über einen deutlich schnelleren Cache verfügt, als die ältere Variante.
Ein Nachteil dieser Konfiguration ist, dass ein Scheduler mit reduzierter Funktionalität verwendet wurde, da der alte Scheduler Fehler enthielt, mit denen ein Multithreading fähiger TCP-Stack nicht hätte umgesetzt werden können und der neue Hardware-Scheduler noch nicht einsatzbereit war. Aus diesen Grund könnten sich die hier gemessen Testergebnisse deutlich verbessern, sobald der neue Scheduler eingebaut ist.
Des Weiteren wurde zu diesem System über den AMIDAR-Systembuilder ein Ethernet-Controller hinzugefügt, mit dem die angeschlossene Netzwerk-Platine genutzt werden kann.

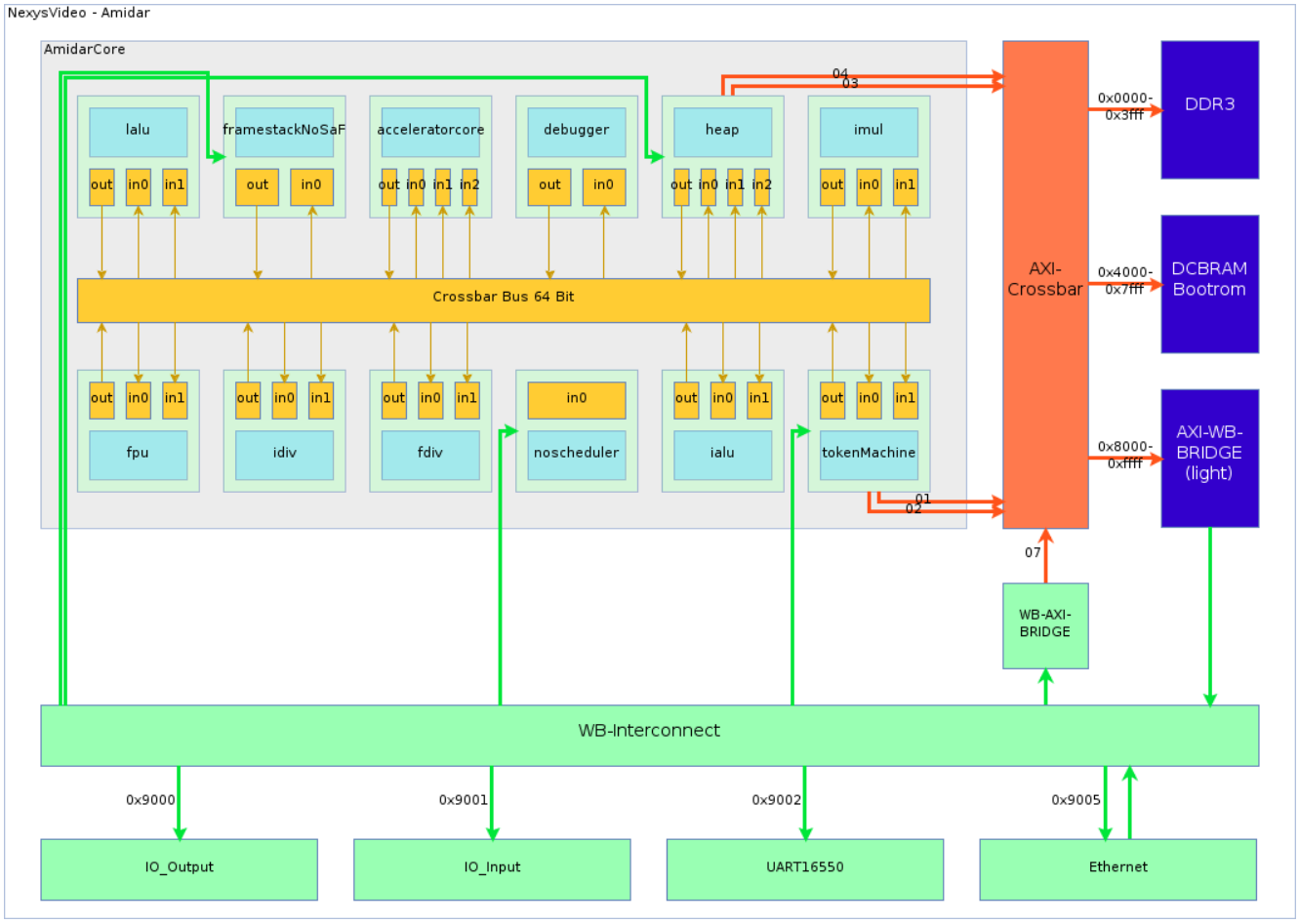


Abbildung 5.1.: Konfiguration des verwendeten AMIDAR-Systems, automatisch generiert durch den AMIDAR-Systembuilder

5.2 Grundlegende Funktionen

Für das Testen der Grundfunktionen wurden mehrere Testszenarien verwendet. Zum einen wurde ein einfacher Echo-Server auf AMIDAR laufen gelassen, der ankommende Wörter zurücksendet. Dieser wurde auf einem oder mehreren Threads aufgeführt. Eine auf dem Laptop laufende Java-Anwendung sendet dabei eingegebene Textnachrichten und wartet auf die Antwort des Servers, welche in der Konsole ausgegeben wird. Dabei wird sowohl der Verbindungsaufbau als auch das Empfangen und Senden von Nachrichten getestet. Dies funktioniert auch mit mehreren offenen Verbindungen parallel.

Eine Anfrage bei einem DHCP Server funktioniert ebenfalls problemlos, wie sich dem Wireshark Screenshot im Anhang entnehmen lässt. Abbildung A.1

5.3 Leistung

Für die Leistungsmessung wird eine Mischung von Daten verwendet, die entweder mit Wireshark ausgelesen wurden, in den Testanwendungen auf dem Laptop in eine .csv Datei geloggt, oder während dem Betrieb auf AMIDAR ermittelt und über UART ausgegeben wurden. Die Latenz der einzelnen Pakete wurde Wireshark entnommen. Da Wireshark auf dem Laptop lief, ist die Übertragungszeit des letzten Pakets, das vom Laptop in einem Handshake gesendet wurde nicht gemessen worden.
Beim drei Wege Handshake, der von dem Laptop initiiert wurde, betrug die RTT nach dem Senden des SYN-Paketes bis zum Eintreffen des SYN-ACK-Pakets 15-30 Millisekunden. Momentan gibt es eine recht große Schwankung der Reaktionszeit.
Falls die Übertragung von AMIDAR initiiert wird, benötigt der Laptop 900 Millisekunden, um auf die Anfrage zu reagieren. Nach weiteren 40 Millisekunden trifft das ACK-Paket von AMIDAR ein.

In einem weiteren Test wurden zehn Verbindungen zum selben Zeitpunkt geöffnet, über die jeweils in kurzen Abständen 500 Byte an Daten übertragen wurden. Diese Daten wurden von AMIDAR als Echo zurückgesendet. Dieses Szenario soll einen Server-Betrieb simulieren, bei dem Anfragen mehrerer Clients beantwortet werden sollen. Die Latenz dieser Übertragung wurde in einer Logdatei gespeichert und betrug zwischen 1,4 und 3,5 Sekunden. Dabei entfiel der größte Teil der Zeit auf das Auslesen der Daten aus den ankommenden Datenpaketen.Abbildung A.2

#	Start Time[ms]	End Time [ms]	Response Time[ms]
1	1495793145037	1495793148691	3654
2	1495793149191	1495793151370	2179
3	1495793151871	1495793153906	2035
4	1495793154406	1495793156688	2282
5	1495793157188	1495793158886	1698
6	1495793159387	1495793160988	1601
7	1495793161489	1495793163013	1524
8	1495793163513	1495793164989	1476

Auffällig dabei ist, dass die Reaktionszeiten der einzelnen Verbindung mit weiteren Iterationen besser werden.

Bei diesem Testszenario wurden mit dem Statistik-Modul der AMIDAR-IDE folgende Werte ermittelt:

jumpBytecodes	16842753
total	545259268
distributedToken	2004025412
axi.transaction	1
cache.hit	65793
cache.hitrate	0.996109
cache.miss	257
outputFifoEmptyAtferFirstTransaction	16780707

Ein Problem, das bei diesem Versuch sichtbar wurde, ist die schwankende Latenz. Die Dauer eines Timeouts wird bei TCP relativ zur RTT bestimmt. Nach dem schnellen SYN-ACK mit kleinen Paketen wird im PC ein entsprechend kurzes Zeitlimit für Timeouts gesetzt. Das hat zur Folge, dass es zu Neuübertragungen größerer Datenpakete kommen kann, wenn diese auf der Seite des AMIDAR-Prozessors verarbeitet werden müssen, obwohl sie nicht verloren gegangen sind. Abbildung A.3

Die Übertragungsrate beim Senden wurde ermittelt, in dem ein zufällig generiertes Datenpaket in einer Schleife gesendet wurde. Dabei wird eine Übertragungsrate von 1333 kBit/s erreicht. Die Größe der einzelnen Datenpakete hat dabei nur einen vernachlässigbaren Einfluss auf die Nettoübertragungsrate, was darauf schließen lässt, dass das Bottleneck bei dem Sendepuffer liegt.

Dabei wurden mit dem Statistik-Modul des Debuggers folgende Werte ermittelt:

jumpBytecodes	537919489
total	545259268
distributedToken	393412689
axi.transaction	1
cache.hit	65793
cache.hitrate	0.9960939
cache.miss	258
outputFifoEmptyAtferFirstTransaction	16781085

6 Zusammenfassung und Ausblick

Aufgabe dieser Bachelorarbeit war die Implementierung eines Multithreading-fähigen-TCP/IP-Stacks, der eine stabile Datenübertragung über mehrere parallele Verbindungen gewährleisten kann. Außerdem sollte die entstandene Lösung evaluiert werden.

Wie dieser Ausarbeitung zu entnehmen ist, wurde ein funktionierender TCP/IP-Stack implementiert und die vorhandenen Netzwerkfunktionen angepasst. Darüber hinaus wurde die Unterstützung für DHCP implementiert.

Wie in der Evaluation gezeigt wurde, funktioniert der TCP-Stack wie erwartet, wobei die Geschwindigkeit noch verbessert werden kann. Eine Anpassung der Datenpuffer für das Senden und Empfangen könnte die Performance deutlich verbessern. Darüber hinaus wird auch der Einbau des neuen Schedulers die Reaktionszeiten reduzieren.

Als zukünftige Erweiterung könnte die Unterstützung für TCP-Optionen implementiert werden, was die Stabilität und Geschwindigkeit in bestimmten Situationen weiter verbessern könnte.

Durch die Unterstützung von TCP können darauf aufbauende Protokolle wie z.B. HTTP, FTP oder SMTP implementiert werden.

A Anhang

A.1 Wireshark Screenshots

tcp||arp

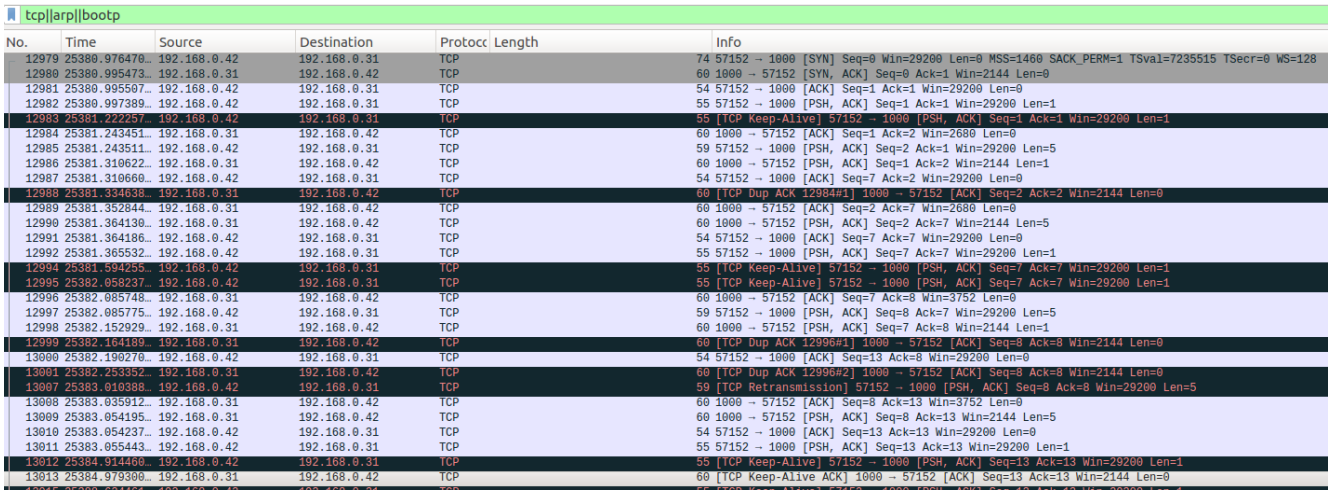
No.	Time	Source	Destination	Protocc	Length	Info
15948	28999.992732…	0.0.0.0	255.255.255.255	DHCP	310	DHCP Discover - Transaction ID 0x1bf33b60
15960	29000.993240…	192.168.0.42	255.255.255.255	DHCP	342	DHCP Offer - Transaction ID 0x1bf33b60
15961	29001.004009…	0.0.0.0	255.255.255.255	DHCP	318	DHCP Request - Transaction ID 0x1bf33b60
15962	29001.004302…	192.168.0.42	255.255.255.255	DHCP	342	DHCP ACK - Transaction ID 0x1bf33b60
16009	29008.267163…	192.168.0.42	192.168.0.31	TCP	74	54972 → 1000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=8146834 TSecr=0 WS=128
16010	29008.273513…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
16011	29008.273549…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=1 Ack=1 Win=29200 Len=0
16012	29008.275150…	192.168.0.42	192.168.0.31	TCP	55	54972 → 1000 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
16013	29008.299694…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=1 Ack=2 Win=2680 Len=0
16014	29008.299729…	192.168.0.42	192.168.0.31	TCP	64	54972 → 1000 [PSH, ACK] Seq=2 Ack=1 Win=29200 Len=10
16015	29008.317791…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [PSH, ACK] Seq=1 Ack=2 Win=2144 Len=1
16016	29008.317927…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=12 Ack=2 Win=29200 Len=0
16017	29008.336132…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=2 Ack=12 Win=2680 Len=0
16018	29008.338628…	192.168.0.31	192.168.0.42	TCP	64	1000 → 54972 [PSH, ACK] Seq=2 Ack=12 Win=2144 Len=10
16019	29008.338704…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=12 Ack=12 Win=29200 Len=0
16021	29008.839063…	192.168.0.42	192.168.0.31	TCP	55	54972 → 1000 [PSH, ACK] Seq=12 Ack=12 Win=29200 Len=1
16022	29008.850740…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=12 Ack=13 Win=3752 Len=0
16023	29008.850768…	192.168.0.42	192.168.0.31	TCP	64	54972 → 1000 [PSH, ACK] Seq=13 Ack=12 Win=29200 Len=10
16024	29008.868657…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [PSH, ACK] Seq=12 Ack=13 Win=2144 Len=1
16025	29008.868703…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=23 Ack=13 Win=29200 Len=0
16026	29008.887032…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=13 Ack=23 Win=3752 Len=0
16027	29008.889525…	192.168.0.31	192.168.0.42	TCP	64	1000 → 54972 [PSH, ACK] Seq=13 Ack=23 Win=2144 Len=10
16028	29008.889594…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=23 Ack=23 Win=29200 Len=0
16029	29009.389961…	192.168.0.42	192.168.0.31	TCP	55	54972 → 1000 [PSH, ACK] Seq=23 Ack=23 Win=29200 Len=1
16030	29009.401705…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=23 Ack=24 Win=4824 Len=0
16031	29009.401771…	192.168.0.42	192.168.0.31	TCP	64	54972 → 1000 [PSH, ACK] Seq=24 Ack=23 Win=29200 Len=10
16032	29009.419686…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [PSH, ACK] Seq=23 Ack=24 Win=2144 Len=1
16033	29009.419778…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=34 Ack=24 Win=29200 Len=0
16038	29009.438409…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=24 Ack=34 Win=4824 Len=0
16039	29009.440958…	192.168.0.31	192.168.0.42	TCP	64	1000 → 54972 [PSH, ACK] Seq=24 Ack=34 Win=2144 Len=10
16040	29009.441074…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=34 Ack=34 Win=29200 Len=0
16042	29009.941597…	192.168.0.42	192.168.0.31	TCP	55	54972 → 1000 [PSH, ACK] Seq=34 Ack=34 Win=29200 Len=1
16043	29009.953594…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=34 Ack=35 Win=5896 Len=0
16044	29009.953683…	192.168.0.42	192.168.0.31	TCP	64	54972 → 1000 [PSH, ACK] Seq=35 Ack=34 Win=29200 Len=10
16045	29009.971591…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [PSH, ACK] Seq=34 Ack=35 Win=2144 Len=1
16046	29009.971694…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=45 Ack=35 Win=29200 Len=0
16047	29009.990008…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=35 Ack=45 Win=5896 Len=0
16048	29009.992299…	192.168.0.31	192.168.0.42	TCP	64	1000 → 54972 [PSH, ACK] Seq=35 Ack=45 Win=2144 Len=10
16049	29009.992416…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=45 Ack=45 Win=29200 Len=0
16050	29010.492846…	192.168.0.42	192.168.0.31	TCP	55	54972 → 1000 [PSH, ACK] Seq=45 Ack=45 Win=29200 Len=1
16051	29010.504696…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=45 Ack=46 Win=6968 Len=0
16052	29010.504722…	192.168.0.42	192.168.0.31	TCP	64	54972 → 1000 [PSH, ACK] Seq=46 Ack=45 Win=29200 Len=10
16053	29010.522660…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [PSH, ACK] Seq=45 Ack=46 Win=2144 Len=1
16054	29010.522716…	192.168.0.42	192.168.0.31	TCP	54	54972 → 1000 [ACK] Seq=56 Ack=46 Win=29200 Len=0
16055	29010.540960…	192.168.0.31	192.168.0.42	TCP	60	1000 → 54972 [ACK] Seq=46 Ack=56 Win=6968 Len=0

Abbildung A.1.: DHCP, Drei-Wege-Handschlag, Datenübertragung

tcp||arp||bootp

No.	Time	Source	Destination	Protocc	Length	Info
13240	26029.830982…	192.168.0.42	192.168.0.31	TCP	74	57704 → 1009 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13241	26029.830998…	192.168.0.42	192.168.0.31	TCP	74	42470 → 1001 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13242	26029.832144…	192.168.0.42	192.168.0.31	TCP	74	39546 → 1002 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13243	26029.832149…	192.168.0.42	192.168.0.31	TCP	74	54720 → 1003 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13244	26029.832150…	192.168.0.42	192.168.0.31	TCP	74	35254 → 1004 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13245	26029.832198…	192.168.0.42	192.168.0.31	TCP	74	41230 → 1005 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13246	26029.832211…	192.168.0.42	192.168.0.31	TCP	74	57218 → 1000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13247	26029.832214…	192.168.0.42	192.168.0.31	TCP	74	58596 → 1007 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7397729 TSecr=0 WS=128
13248	26029.863224…	192.168.0.31	192.168.0.42	TCP	60	1000 → 57218 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
13249	26029.863260…	192.168.0.42	192.168.0.31	TCP	54	57218 → 1000 [ACK] Seq=1 Ack=1 Win=29200 Len=0
13250	26029.864903…	192.168.0.42	192.168.0.31	TCP	55	57218 → 1000 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
13251	26029.873061…	192.168.0.31	192.168.0.42	TCP	60	1001 → 42470 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
13252	26029.873096…	192.168.0.42	192.168.0.31	TCP	54	42470 → 1001 [ACK] Seq=1 Ack=1 Win=29200 Len=0
13253	26029.873249…	192.168.0.42	192.168.0.31	TCP	55	42470 → 1001 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
13254	26029.883392…	192.168.0.31	192.168.0.42	TCP	60	1002 → 39546 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
13255	26029.883428…	192.168.0.42	192.168.0.31	TCP	54	39546 → 1002 [ACK] Seq=1 Ack=1 Win=29200 Len=0
13256	26029.883568…	192.168.0.42	192.168.0.31	TCP	55	39546 → 1002 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
13257	26029.893404…	192.168.0.31	192.168.0.42	TCP	60	1003 → 54720 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
13258	26029.893442…	192.168.0.42	192.168.0.31	TCP	54	54720 → 1003 [ACK] Seq=1 Ack=1 Win=29200 Len=0
13259	26029.893617…	192.168.0.42	192.168.0.31	TCP	55	54720 → 1003 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
13260	26029.903626…	192.168.0.31	192.168.0.42	TCP	60	1004 → 35254 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
13261	26029.903661…	192.168.0.42	192.168.0.31	TCP	54	35254 → 1004 [ACK] Seq=1 Ack=1 Win=29200 Len=0
13262	26029.903801…	192.168.0.42	192.168.0.31	TCP	55	35254 → 1004 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
13263	26029.913832…	192.168.0.31	192.168.0.42	TCP	60	1005 → 41230 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
13264	26029.913863…	192.168.0.42	192.168.0.31	TCP	54	41230 → 1005 [ACK] Seq=1 Ack=1 Win=29200 Len=0
13265	26029.914003…	192.168.0.42	192.168.0.31	TCP	55	41230 → 1005 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
13266	26029.923940…	192.168.0.31	192.168.0.42	TCP	60	1006 → 34196 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
13267	26029.923988…	192.168.0.42	192.168.0.31	TCP	54	34196 → 1006 [ACK] Seq=1 Ack=1 Win=29200 Len=0
13268	26029.924242…	192.168.0.42	192.168.0.31	TCP	55	34196 → 1006 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1

Abbildung A.2.: Gleichzeitiger Verbindungsaufbau mehrerer Verbindungen



tcp||arp||bootp

No.	Time	Source	Destination	Protocc	Length	Info
12979	25380.976470…	192.168.0.42	192.168.0.31	TCP		74 57152 → 1000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7235515 TSecr=0 WS=128
12980	25380.995473…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0
12981	25380.995507…	192.168.0.42	192.168.0.31	TCP		54 57152 → 1000 [ACK] Seq=1 Ack=1 Win=29200 Len=0
12982	25380.997389…	192.168.0.42	192.168.0.31	TCP		55 57152 → 1000 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
12983	25381.222257…	192.168.0.42	192.168.0.31	TCP		55 [TCP Keep-Alive] 57152 → 1000 [PSH, ACK] Seq=1 Ack=1 Win=29200 Len=1
12984	25381.243451…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [ACK] Seq=1 Ack=2 Win=2680 Len=0
12985	25381.243511…	192.168.0.42	192.168.0.31	TCP		59 57152 → 1000 [PSH, ACK] Seq=2 Ack=1 Win=29200 Len=5
12986	25381.310622…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [PSH, ACK] Seq=1 Ack=2 Win=2144 Len=1
12987	25381.310660…	192.168.0.42	192.168.0.31	TCP		54 57152 → 1000 [ACK] Seq=7 Ack=2 Win=29200 Len=0
12988	25381.334638…	192.168.0.31	192.168.0.42	TCP		60 [TCP Dup ACK 12984#1] 1000 → 57152 [ACK] Seq=2 Ack=2 Win=2144 Len=0
12989	25381.352844…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [ACK] Seq=2 Ack=7 Win=2680 Len=0
12990	25381.364130…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [PSH, ACK] Seq=2 Ack=7 Win=2144 Len=5
12991	25381.364186…	192.168.0.42	192.168.0.31	TCP		54 57152 → 1000 [ACK] Seq=7 Ack=7 Win=29200 Len=0
12992	25381.365532…	192.168.0.42	192.168.0.31	TCP		55 57152 → 1000 [PSH, ACK] Seq=7 Ack=7 Win=29200 Len=1
12994	25381.594255…	192.168.0.42	192.168.0.31	TCP		55 [TCP Keep-Alive] 57152 → 1000 [PSH, ACK] Seq=7 Ack=7 Win=29200 Len=1
12995	25382.058237…	192.168.0.42	192.168.0.31	TCP		55 [TCP Keep-Alive] 57152 → 1000 [PSH, ACK] Seq=7 Ack=7 Win=29200 Len=1
12996	25382.085748…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [ACK] Seq=7 Ack=8 Win=3752 Len=0
12997	25382.085775…	192.168.0.42	192.168.0.31	TCP		59 57152 → 1000 [PSH, ACK] Seq=8 Ack=7 Win=29200 Len=5
12998	25382.152929…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [PSH, ACK] Seq=7 Ack=8 Win=2144 Len=1
12999	25382.164189…	192.168.0.31	192.168.0.42	TCP		60 [TCP Dup ACK 12996#1] 1000 → 57152 [ACK] Seq=8 Ack=8 Win=2144 Len=0
13000	25382.190270…	192.168.0.42	192.168.0.31	TCP		54 57152 → 1000 [ACK] Seq=13 Ack=8 Win=29200 Len=0
13001	25382.253352…	192.168.0.31	192.168.0.42	TCP		60 [TCP Dup ACK 12996#2] 1000 → 57152 [ACK] Seq=8 Ack=8 Win=2144 Len=0
13007	25383.010388…	192.168.0.42	192.168.0.31	TCP		59 [TCP Retransmission] 57152 → 1000 [PSH, ACK] Seq=8 Ack=8 Win=29200 Len=5
13008	25383.035912…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [ACK] Seq=8 Ack=13 Win=3752 Len=0
13009	25383.054195…	192.168.0.31	192.168.0.42	TCP		60 1000 → 57152 [PSH, ACK] Seq=8 Ack=13 Win=2144 Len=5
13010	25383.054237…	192.168.0.42	192.168.0.31	TCP		54 57152 → 1000 [ACK] Seq=13 Ack=13 Win=29200 Len=0
13011	25383.055443…	192.168.0.42	192.168.0.31	TCP		55 57152 → 1000 [PSH, ACK] Seq=13 Ack=13 Win=29200 Len=1
13012	25384.914460…	192.168.0.42	192.168.0.31	TCP		55 [TCP Keep-Alive] 57152 → 1000 [PSH, ACK] Seq=13 Ack=13 Win=29200 Len=1
13013	25384.979300…	192.168.0.31	192.168.0.42	TCP		60 [TCP Keep-Alive ACK] 1000 → 57152 [ACK] Seq=13 Ack=13 Win=2144 Len=0

Abbildung A.3.: keep-alive und Retransmissions, wegen schwankender Latenz, bei großen Datenpaketen

Abkürzungsverzeichnis

CPU central processing unit

FPGA Field Programmable Gate Array

RTT Round Trip Time

AMIDAR Adaptive Microinstruction Driven ARchitecture

IP Internet Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

API Application Programming Interface

MAC Media-Access-Control

IEEE Institute of Electrical and Electronics Engineers

TTL Time to Live

FU Function Unit

ALU arithmetic logic unit

Abbildungsverzeichnis

2.1. IP Segment Format 7
2.2. IP-Paket Fragmentierung 9
2.3. TCP Control Bits 10
2.4. TCP Segment Format 11
2.5. vereinfachter Zustandsautomat TCP 13
2.6. TCP Handshake mit Zustandstransitionen 14
2.7. BOOTP Message Format 18

3.1. AMIDAR Aufbau 21

4.1. Auszug Klassendiagramm TCP/IP 23

5.1. Konfiguration des verwendeten AMIDAR-Systems, automatisch generiert durch den AMIDAR-Systembuilder 33

A.1. DHCP, Drei-Wege-Handschlag, Datenübertragung 37
A.2. Gleichzeitiger Verbindungsaufbau mehrerer Verbindungen 37
A.3. keep-alive und Retransmissions, wegen schwankender Latenz, bei großen Datenpaketen 38

Literaturverzeichnis

[cc] *TCP Congestion Control*. https://tools.ietf.org/html/rfc5681. – stand 30.05.2017

[DHC] *DHCP Options and Bootp Vendor Extensions*. https://tools.ietf.org/html/rfc2132

[Eth] *IEEE 802.3 / Ethernet-Grundlagen*. https://www.elektronik-kompendium.de/sites/net/0603201.htm. – stand 30.05.2017

[IPr] *INTERNET PROTOCOL - PROTOCOL SPECIFICATION*. https://tools.ietf.org/html/rfc791. – stand 30.05.2017

[ISC] *ISC-DHCPD*. https://wiki.ubuntuusers.de/ISC-DHCPD/. – stand 30.05.2017

[Lay] *TCP/IP Five Layer Software Model Overview*. http://microchipdeveloper.com/tcpip:tcp-ip-five-layer-model. – stand 30.05.2017

[RYZ] Raid Y. Zaghal, Javed I. K.: *EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno*. http://www.medianet.kent.edu/techreports/TR2005-07-22-tcp-EFSM.pdf. – stand 30.05.2017

[Sch] Schoeberl, Martin: *ejlP: A TCP/IP Stack for Embedded Java*. http://orbit.dtu.dk/files/6633077/41129d01.pdf. – stand 30.05.2017

[TCP] *TRANSMISSION CONTROL PROTOCOL - PROTOCOL SPECIFICATION*. https://tools.ietf.org/html/rfc793. – stand 30.05.2017