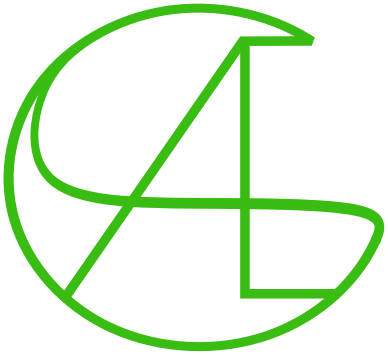
Technische Universität Ilmenau
Fakultät Informatik und Automatisierung
Fachgebiet Telematik/Rechnernetze

TECHNISCHE UNIVERSITÄT
ILMENAU

Drittes Review zum Thema

Abwehr von Denial-of-Service-Angriffen durch effiziente User-Space Paketverarbeitung

Codename: AEGIS



Autoren:

Fabienne Göpfert
Tim Häußler
Felix Hußlein
Robert Jeutter
Johannes Lang
Leon Leisten
Jakob Lerch
Tobias Scholz

Betreuer: Martin Backhaus

Entwurf 20. Juli 2021

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Denial-of-Service-Angriffe stellen eine ernstzunehmende und stetig wachsende Bedrohung dar. Im digitalen Zeitalter sind viele Systeme über das Internet oder private Netzwerke miteinander verbunden. Viele Unternehmen, Krankenhäuser und Behörden sind durch unzureichende Schutzmaßnahmen und große Wirkung zu beliebten Angriffszielen geworden [1]. Bei solchen Angriffe werden in der Regel finanzielle oder auch politische Gründe verfolgt, selten aber auch die bloße Störung oder Destruktion des Ziels.

Bei DoS[1]- und DDoS[2]-Attacken werden Server und Infrastrukturen mit einer Flut sinnloser Anfragen so stark überlastet, dass sie von ihrem normalen Betrieb abgebracht werden. Daraus kann resultieren, dass Nutzer die angebotenen Dienste des Betreibers nicht mehr erreichen und Daten bei dem Angriff verloren gehen können. Hierbei können schon schwache Rechner große Schaden bei deutlich leistungsfähigeren Empfängern auslösen. In Botnetzen können die Angriffe zusätzlich von mehreren Computern gleichzeitig koordiniert werden, aus verschiedensten Netzwerken stammen [2] und damit gleichzeitig die Angriffskraft verstärken und die Erkennung erschweren.

Das Ungleichgewicht zwischen der Einfachheit bei der Erzeugung von Angriffen gegenüber komplexer und ressourcenintensiver DoS-Abwehr verschärft das Problem zusätzlich. Obwohl gelegentlich Erfolge im Kampf gegen DoS-Angriffe erzielt werden (z.B. Stilllegung einiger großer „DoS-for-Hire" Webseiten), vergrößert sich das Datenvolumen der DoS-Angriffe stetig weiter. Allein zwischen 2014 und 2017 hat sich die Frequenz von DoS-Angriffen um den Faktor 2,5 vergrößert und das Angriffsvolumen verdoppelt sich fast jährlich [3]. Die Schäden werden weltweit zwischen 20.000 und 40.000 US-Dollar pro Stunde geschätzt [4].

Im Bereich kommerzieller DoS-Abwehr haben sich einige Ansätze hervorgetan (z.B. Project Shield [5], Cloudflare [6] oder AWS Shield [7]). Der Einsatz kommerzieller Lösungen birgt jedoch einige Probleme, etwa mitunter erhebliche Kosten oder das Problem des notwendigen Vertrauens, welches dem Betreiber einer DoS-Abwehr entgegengebracht werden muss. Folglich ist eine effiziente Abwehr von DoS-Angriffen mit eigenen Mitteln ein oft gewünschtes Ziel - insbesondere wenn sich dadurch mehrere Systeme zugleich schützen lassen.

[1] Denial of Service, dt.: Verweigerung des Dienstes, Nichtverfügbarkeit des Dienstes
[2] Distributed Denial of Service

Ziel dieses Softwareprojekts ist es, ein System zwischen der Internet-Anbindung und dem internem Netzwerk zu schaffen, das bei einer hohen Bandbreite und im Dauerbetrieb effektiv (D)DoS Angriffe abwehren kann, während Nutzer weiterhin ohne Einschränkungen auf ihre Dienste zugreifen können. Die entstehende Anwendung implementiert eine (D)DoS-Verkehrs-Inspektion und einen intelligenten Regelgenerator, wodurch interne Netzwerke vor externen Bedrohungen, die zu einer Überlastung des Systems führen würden, geschützt sind. Es enthält Algorithmen zur Verkehrsanalyse, die bösartigen Verkehr erkennen und ausfiltern können, ohne die Benutzererfahrung zu beeinträchtigen und ohne zu Ausfallzeiten zu führen.

1.2 Überblick

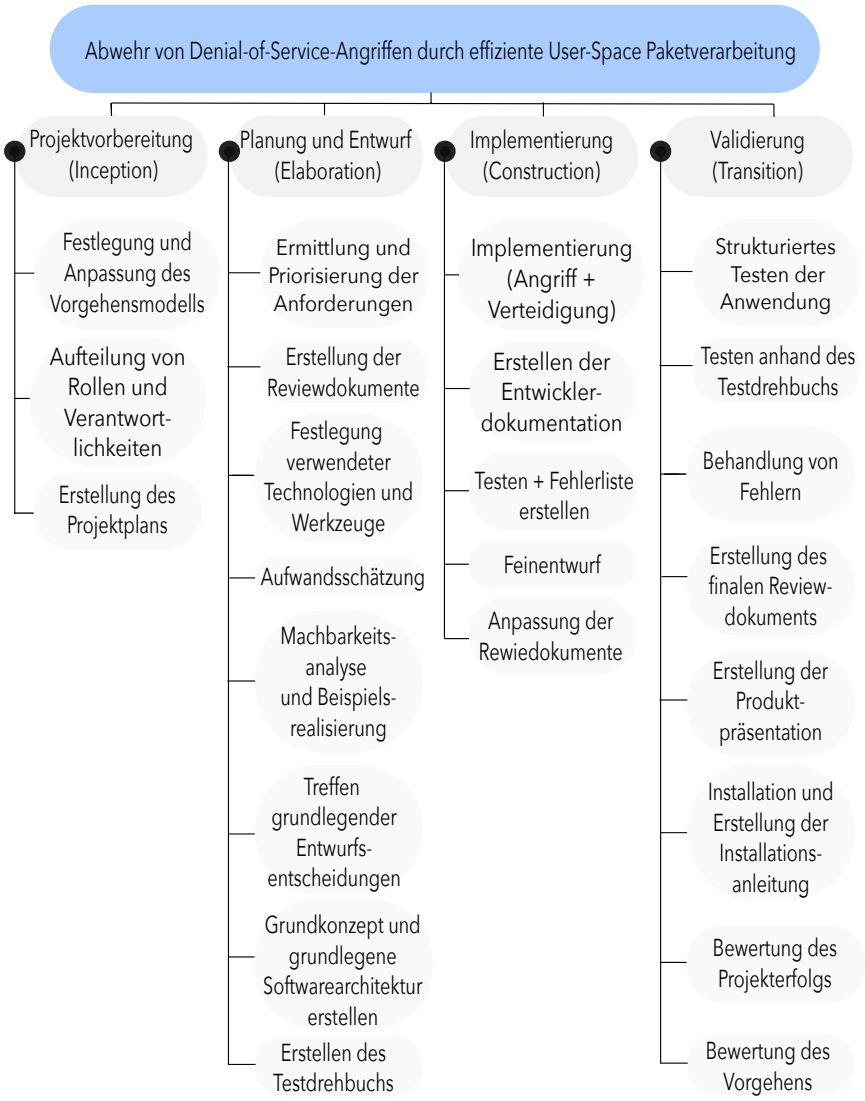


Abbildung 1.1: Projektstrukturplan

Das Softwareprojekt wurde vom zuständigen Fachgebiet in drei Teile aufgeteilt. Die Planungs- und Entwicklungsphase, die Implementierungsphase und die Validierungsphase dauern jeweils einen Monat und werden durch ein Review abgeschlossen. Zu diesen Reviews werden die Ergebnisse der vergangenen Phase vorgestellt und die erforderlichen Review-Dokumente abgegeben.

Zu Beginn des Projekts wurde sich auf den Unified Process als Vorgehensmodell geeinigt, damit sowohl ein gewisser Grad an Flexibilität als auch die Planbarkeit der Ergebnisse gewährleistet werden kann. Prinzipiell besteht dieses Vorgehensmodell aus vier Phasen, von denen die Konzeption und die Ausarbeitung beide in der Planungs- und Entwurfsphase lagen. Die Konstruktionsphase und die Inbetriebnahme decken sich zeitlich mit der Implementierungs- und der Validierungsphase.

Dieses dritte Review-Dokument bezieht sich auf die Validierungssphase. Das heißt, dass es auf den Ergebnissen der vorhergehenden Phase und dem zweiten Review-Dokument vom 23. Juni 2021 aufbaut.

Das erste Review-Dokument enthält die gängigen Inhalte eines Pflichtenhefts wie die funktionalen und nicht-funktionalen Anforderungen, eine Aufwands- und Risikoanalyse und Überlegungen zum Vorgehen und der internen Organisation. Außerdem umfasste es eine Entwurfsdokumentation für den Grobentwurf, die Anforderungsanalyse, ein Kapitel zu den Technologien und Entwicklungswerkzeugen, Ergebnisse zu den Machbarkeitsanalysen und Beispielrealisierungen und ein Testdrehbuch.

Im zweiten Review-Dokument wurden im Kapitel zum Grobentwurf zusätzlich zur erneuten Erläuterung der grundlegenden Architektur die für den Unified Process üblichen Überarbeitungen des Grobentwurfs dargestellt und begründet. Dabei wurde, genauso wie beim darauffolgenden Feinentwurf, Paket für Paket vorgegangen. Schließlich wurden in einem Bug-Review die offenen Anforderungen und Fehler beschrieben und die mittels des Tools Kimai erfassten Arbeitszeiten ausgewertet.

In diesem dritten Review-Dokument kommt nun ein ausführliches Kapitel zu sämtlichen Tests dazu. Außerdem behandelt es verschiedene Softwaremetriken und Statistiken, wie Konventionen und den Umfang der Software. Das Kapitel zur Auswertung der erfassten Arbeitszeiten enthält nun auch diese letzte Phase des Softwareprojekts. Am Ende des Dokuments kommt es zur umfangreichen Auswertung des Projekts.

Es bleibt also anzumerken, dass einige Teile dieses Dokuments dem ersten und zweiten Review-Dokument entnommen sind, weil dies vom Fachgebiet empfohlen wurde und dadurch die Veränderungen besonders gut dargestellt werden können.

Die Erstellung dieses Review-Dokuments stellt allerdings nur einen Teil der in dieser Phase erledigten Aufgaben dar. Hauptsächlich ging es um das Testen, aber auch um weitere Themen, was sich im Projektstrukturplan in Abb. 1.1 gut erkennen lässt.

Kapitel 2

Grobentwurf

Dieses Kapitel behandelt zunächst den Grobentwurf, wie er in der Planungs- und Entwurfsphase des Projekts erarbeitet wurde. Schließlich wird auf dessen Überarbeitung und dazugehörige Diagramme eingegangen.

2.1 Grundlegende Architektur

In folgendem Unterkapitel werden die grundlegenden Entscheidungen des Entwurfs erklärt und durch die Rahmenbedingungen begründet. Ein intuitiver Einstieg soll schrittweise an das System heranführen über Erklärung des Netzwerkaufbaus, dem grundlegenden Aufbau der Software, dem Kontrollfluss eines Pakets und verwendeter Verfahren.

2.1.1 Netzwerkaufbau

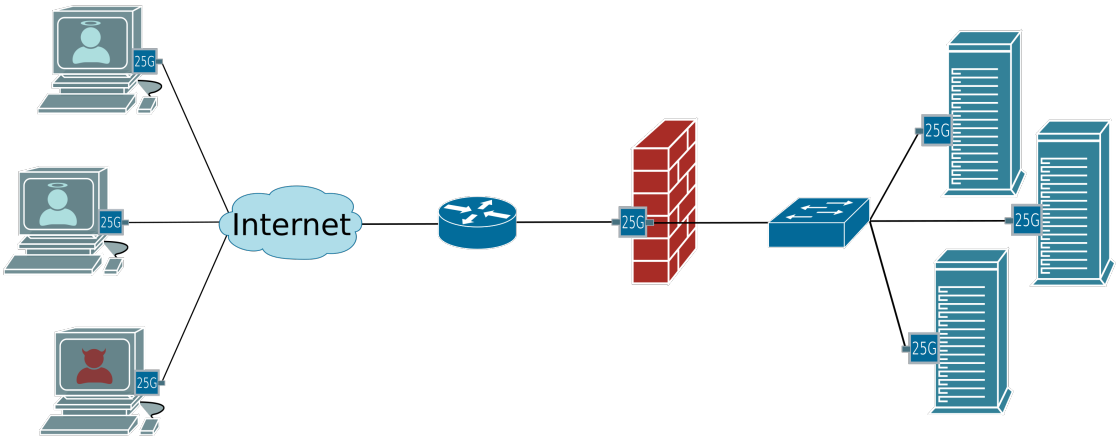


Abbildung 2.1: Realaufbau unter Verwendung eines Angreifers

Die Abbildung 2.1 zeigen den typischen, zu erwartenden Netzwerkaufbau, welcher in dieser Form im Internet und in der Produktivumgebung vorkommt. Das System untergliedert sich grob in

drei Teile. Links in der Abbildung ist jeweils das Internet zu erkennen. In diesem sind unterschiedliche Netzwerke mit jeweils verschiedenen Computern miteinander verbunden. Unter den vielen Computern im Internet, welche für Serversysteme teilweise harmlos sind, befinden sich allerdings auch einige Angreifer. Hier ist ganz klar eine Unterscheidung zwischen dem Angriff eines einzelnen Angreifers, oder einer Menge von einem Angreifer gekaperten und gesteuerten Computer, also eines Botnets, vorzunehmen.

Wird das Internet, hin zum zu schützenden Netzwerk, verlassen, so wird zuerst ein Router vorgefunden, welcher Aufgaben wie die Network Address Translation vornimmt. Hinter diesem Router befände sich im Produktiveinsatz nun das zu entwickelnde System. Router und zu entwickelndes System sind ebenfalls über eine Verbindung mit ausreichend, in diesem Fall 25Gbit/s, Bandbreite verbunden. Das System selbst agiert als Mittelsmann zwischen Router, also im Allgemeinen dem Internet, und dem internen Netz. Um mehrere Systeme gleichzeitig schützen zu können, aber dennoch die Kosten gering zu halten, ist dem zu entwickelnden System ein Switch nachgeschaltet, mit welchem wiederum alle Endsysteme verbunden sind.

Leider ist durch Begrenzungen im Budget, der Ausstattung der Universität sowie der Unmöglichkeit das Internet in seiner Gesamtheit nachzustellen ein exakter Nachbau des Systems für dieses Projekt nicht möglich. Deswegen musste ein alternativer Aufbau gefunden werden, der allerdings vergleichbare Charakteristika aufweisen muss.

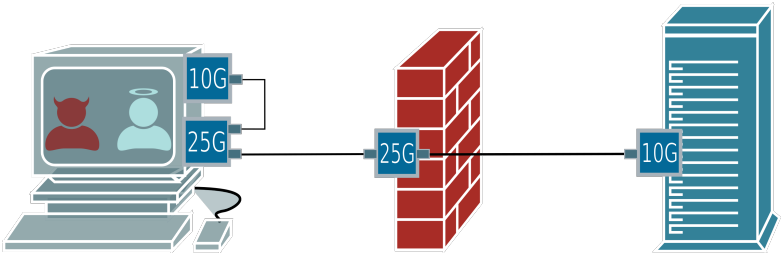


Abbildung 2.2: Versuchsaufbau

Der für das Projekt verwendete Versuchsaufbau untergliedert sich ebenfalls in drei Teile. Auch hier beginnt die Darstellung 2.2 ganz links mit dem System, welches Angreifer und legitimen Nutzer in sich vereint. Um die Funktionalität von Angreifer und Nutzer gleichzeitig bereitstellen zu können, setzt der Projektstab, in diesem Fall auf das Installieren zweier Netzwerkkarten, in einem Computer. Eine 10 Gbit/s Netzwerkkarte ist mit der Aufgabe betraut, legitimen Verkehr zu erzeugen. Da aufgrund der Hardwarerestriktionen keine direkte Verbindung zur Middlebox aufgebaut werden kann, wird der ausgehende Verkehr dieser Netzwerkkarte in einen Eingang einer zweiten, in demselben System verbauten Netzwerkkarte mit einer maximalen Datenrate von 25 Gbit/s eingeführt. Von dieser führt ein 25 Gbit/s Link direkt zur Middlebox. Intern wird nun im System, das sich in der Abbildung 2.2 auf der rechten Seite befindet, sowohl legitimer Verkehr erzeugt als auch Angriffsverkehr kreiert, wobei diese beiden Paketströme intern zusammengeführt werden, und über den einzigen Link an die Middlebox gemeinsam übertragen werden. Die Middlebox selbst ist nicht nur mit dem externen Netz verbunden, sondern hat über die selbe Netzwerkkarte auch noch eine Verbindung ins interne Netz. Das gesamte interne Netz wird im Versuchsaufbau durch einen einzelnen, mit nur 10 Gbit/s angebundenen Computer realisiert.

Die Entscheidung zur Realisierung in dieser Art fiel, da insbesondere der Fokus darauf liegen soll, ein System zu erschaffen, welches in der Lage ist, mit bis zu 25 Gbit/s an Angriffsverkehr und legitimen eingehenden Verkehr zurechtzukommen. Aus diesem Grund ist es ausreichend, eine

Verbindung zum internen Netz mit nur 10 Gbit/s aufzubauen, da dieses System bei erfolgreicher Abwehr und Abschwächung der Angriffe mit eben diesen maximalen 10 Gbit/s an legitimen Verkehr zurecht kommen muss. Ursächlich für die Verwendung der 10 Gbit/s Netzwerkkarte im externen Rechner, welcher hierüber den legitimen Verkehr bereitstellen soll, ist, dass der Fokus bei einem solchen Schutzmechanismus natürlich darauf beruht, die Datenrate des Angreifers zu maximieren, um das zu entwickelnde System in ausreichendem Maße belasten und somit Stresstests unterwerfen zu können.

2.1.2 Grundlegender Aufbau der Software

Das Grundprinzip der zu entwickelten Software soll sein, Pakete auf einem Port der Netzwerkkarte zu empfangen und diese zu einem anderen Port weiterzuleiten. Zwischen diesen beiden Schritten werden die Pakete untersucht, Daten aus diesen extrahiert und ausgewertet. Im weiteren Verlauf des Programms werden Pakete, welche einem Angriff zugeordnet werden, verworfen, und legitime Pakete zwischen dem internen und externen Netz ausgetauscht. Es bietet sich an, hier ein Pipelinemodell zu verwenden, wobei die einzelnen Softwarekomponenten in Pakete aufgeteilt werden. Im `ConfigurationManagement` werden die initialen Konfigurationen vorgenommen. Das `NicManagement` ist eine Abstraktion der Netzwerkkarte und sorgt für das Empfangen und Senden von Paketen. Die `PacketDissection` extrahiert Daten von eingehenden Paketen. Die `Inspection` analysiert diese Daten und bestimmt, welche Pakete verworfen werden sollen. Das `Treatment` behandelt die Pakete nach entsprechenden Protokollen. Um die Abarbeitung dieser Pipeline möglichst effizient zu gestalten, soll diese jeweils von mehreren Threads parallel und möglichst unabhängig voneinander durchschritten werden.

In den folgenden Sektionen wird auf den Kontrollfluss innerhalb des Programms, auf den Einsatz von parallelen Threads und auf die einzelnen Komponenten näher eingegangen.

2.1.2.1 Einsatz von parallelen Threads

Zunächst ist jedoch ein wichtiger Aspekt der Architektur hervorzuheben. Von der Mitigation-Box wird gefordert, eine hohe Paket- und Datenlast verarbeiten zu können. Das Hardwaresystem, auf welchem das zu entwickelnde Programm laufen wird, besitzt eine Multicore-CPU, d.h. das System ist in der Lage, Aufgaben aus unterschiedlichen Threads parallel zu bearbeiten. Dies hat das Potenzial, die Rechengeschwindigkeit zu vervielfachen und so die Bearbeitungszeit insgesamt zu verringern.

Dabei stellt sich die Frage, wozu die Threads im Programm genau zuständig sind. Es wäre zum Beispiel möglich, dass jeder Thread eine Aufgabe übernimmt, d.h. es gäbe einen Thread, der nur Daten analysiert oder aber einen Thread, der nur Paketinformationen extrahiert. Eine solche Aufteilung würde allerdings zu einem hohen Grad an Inter-Thread-Kommunikation führen. Diese ist nicht trivial und kann einen Großteil der verfügbaren Ressourcen benötigen, was den durch die Parallelisierung erzielten Gewinn wieder zunichte machen könnte. Um dieses Risiko zu vermeiden, soll stattdessen jeder Thread die gesamte Pipeline durchlaufen. So ist kaum Inter-Thread-Kommunikation notwendig. Außerdem ist es dann verhältnismäßig einfach, den Entwurf skalierbar zu gestalten: Wenn ein Prozessor mit größerer Anzahl an Kernen verwendet werden würde, könnten mehr Pakete parallel bearbeitet werden, ohne dass die Architektur geändert werden muss.

2.1.2.2 Kontrollfluss eines Paketes

In diesem Abschnitt soll veranschaulicht werden, wie genau die Behandlung eines Paketes vom `NicManagement` bis zum `Treatment` erfolgt. Dabei werden die Pakete selbst als Akteure angesehen und nicht deren Klassen. Hinweis: Ein Thread soll später mehrere Pakete auf einmal durch die Pipeline führen. In diesem Diagramm wird zur Übersichtlichkeit jedoch nur der Fluss eines Paketes gezeigt. Dieser lässt sich dann einfach auf eine größere Menge von Paketen anwenden. Ein Aktivitätsdiagramm ist unter Abbildung 2.3 am Ende der Sektion 2.1.2 zu finden.

2.1.2.3 Verwendung von Receive-Side-Scaling

Ein weiterer grundlegender Vorteil ergibt sich durch das von der Netzwerkkarte und von DPDK unterstützte Receive Side Scaling (RSS), siehe Abbildung 2.4: Ein auf einem Port eingehendes Paket wird einer von mehreren sogenannten RX-Queues zugeordnet. Eine RX-Queue gehört immer zu genau einem Netzwerkkartenport, ein Port kann mehrere RX-Queues besitzen. Kommen mehrere Pakete bei der Netzwerkkarte an, so ist die Zuordnung von Paketen eines Ports zu seinen RX-Queues gleich verteilt – alle RX-Queues sind gleich stark ausgelastet. Diese Zuordnung wird durch eine Hashfunktion umgesetzt, in die Quell- und Ziel-Port-Nummer und IP-Adresse einfließen. Das führt dazu, dass Pakete, die auf einem Port ankommen und einer bestimmten Verbindung zugehören, immer wieder zu der selben RX-Queue dieses Ports zugeordnet werden. Mit „Port" im Folgenden entweder der physische Steckplatz einer Netzwerkkarte gemeint oder jener Teil der Netzwerkadresse, die eine Zuordnung zu einem bestimmten Prozess bewirkt. Die Bedeutung erschließt sich aus dem Kontext.

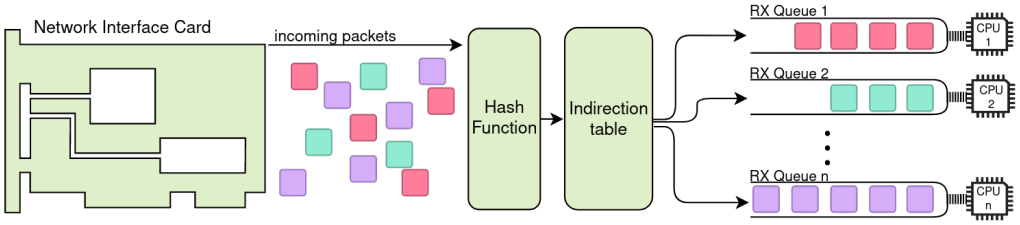


Abbildung 2.4: Beispielhafte Paketverarbeitung mit Receive Side Scaling

Ferner besteht die Möglichkeit, Symmetric RSS einzusetzen. Dieser Mechanismus sorgt dafür, dass die Pakete, die auf dem einen Port der Netzwerkkarte ankommen, nach genau der selben Zuordnung auf dessen RX-Queues aufgeteilt werden, wie die auf dem anderen Port ankommenden Pakete auf dessen RX-Queues. Dabei ist die Zuordnung auf dem einen Port „symmetrisch" zu der auf dem anderen Port. Das heißt, wenn bei Port 0 ein Paket mit `Src-IP: a, Dst-IP: b, Src-Port: x, Dst-Port: y` ankommt, wird es genauso dessen RX-Queues zugeteilt, wie ein Paket mit `Src-IP: b, Dst-IP: a, Src-Port: y, Dst-Port: x` auf RX-Queues von Port 1. So ergeben sich Paare von RX-Queues, die jeweils immer Pakete von den gleichen Verbindungen beinhalten. Angenommen, die RX-Queues sind mit natürlichen Zahlen benannt und RX-Queue 3 auf Port 0 und RX-Queue 5 auf Port 1 sind ein korrespondierendes RX-Queue-Paar. Wenn nun ein Paket P, zugehörig einer Verbindung V auf RX-Queue 3, Port 0 ankommt, dann weiß man, dass Pakete, die auf Port 1 ankommen und der Verbindung V angehören immer auf RX-Queue 5, Port 1 landen.

Neben RX-Queues existieren auch TX-Queues (Transmit-Queues), die ebenfalls zu einem bestimmten Port gehören. Darin befindliche Pakete werden von der Netzwerkkarte auf den entsprechenden Port geleitet und gesendet. Auf Basis dieses Mechanismus sollen die Threads wie folgt organisiert werden: Einem Thread gehört ein Paar von korrespondierenden RX-Queues (auf verschiedenen Ports) und daneben eine TX-Queue auf dem einen und eine TX-Queue auf dem anderen Port. Das bringt einige Vorteile mit sich: Es müssen zwei Arten von Informationen entlang der Pipeline gespeichert, verarbeitet und gelesen werden: Informationen zu einer Verbindung und Analyseinformationen/Statistiken. Daher ist kaum Inter-Thread-Kommunikation nötig, weil alle Informationen zu einer Verbindung in Datenstrukturen gespeichert werden können, auf die nur genau der bearbeitende Thread Zugriff haben muss. An dieser Stelle soll auch kurz auf eine Besonderheit von DPDK eingegangen werden: Im Linux-Kernel empfängt ein Programm Pakete durch Interrupt-Handling. Gegensätzlich dazu werden bei DPDK alle empfangenen Pakete, die sich derzeit in den RX-Queues der Netzwerkkarte befinden, auf einmal von der Anwendung geholt. In der zu entwickelnden Software geschieht dieses Paket-holen (engl. „Polling") durch den einzelnen Thread stets zu Beginn eines Pipeline-Durchlaufes.

Im Falle eines Angriffes ist die Seite des Angreifers (entsprechender Port z.B. „Port 0") viel stärker belastet, als die Seite des Servers (z.B. „Port 1"). Wegen der gleich verteilten Zuordnung des eingehenden Verkehrs auf die RX-Queues und weil ein Thread von RX-Queues von beiden Ports regelmäßig Pakete pollt, sind alle Threads gleichmäßig ausgelastet und können die Pakete bearbeiten. Ein günstiger Nebeneffekt bei DDoS-Angriffen ist, dass die Absenderadressen von Angriffspaketen oft sehr unterschiedlich sind. Das begünstigt die gleichmäßige Verteilung von Paketen auf RX-Queues, weil das Tupel aus besagten Adressen der Schlüssel der RSS-Hash-Funktion sind.

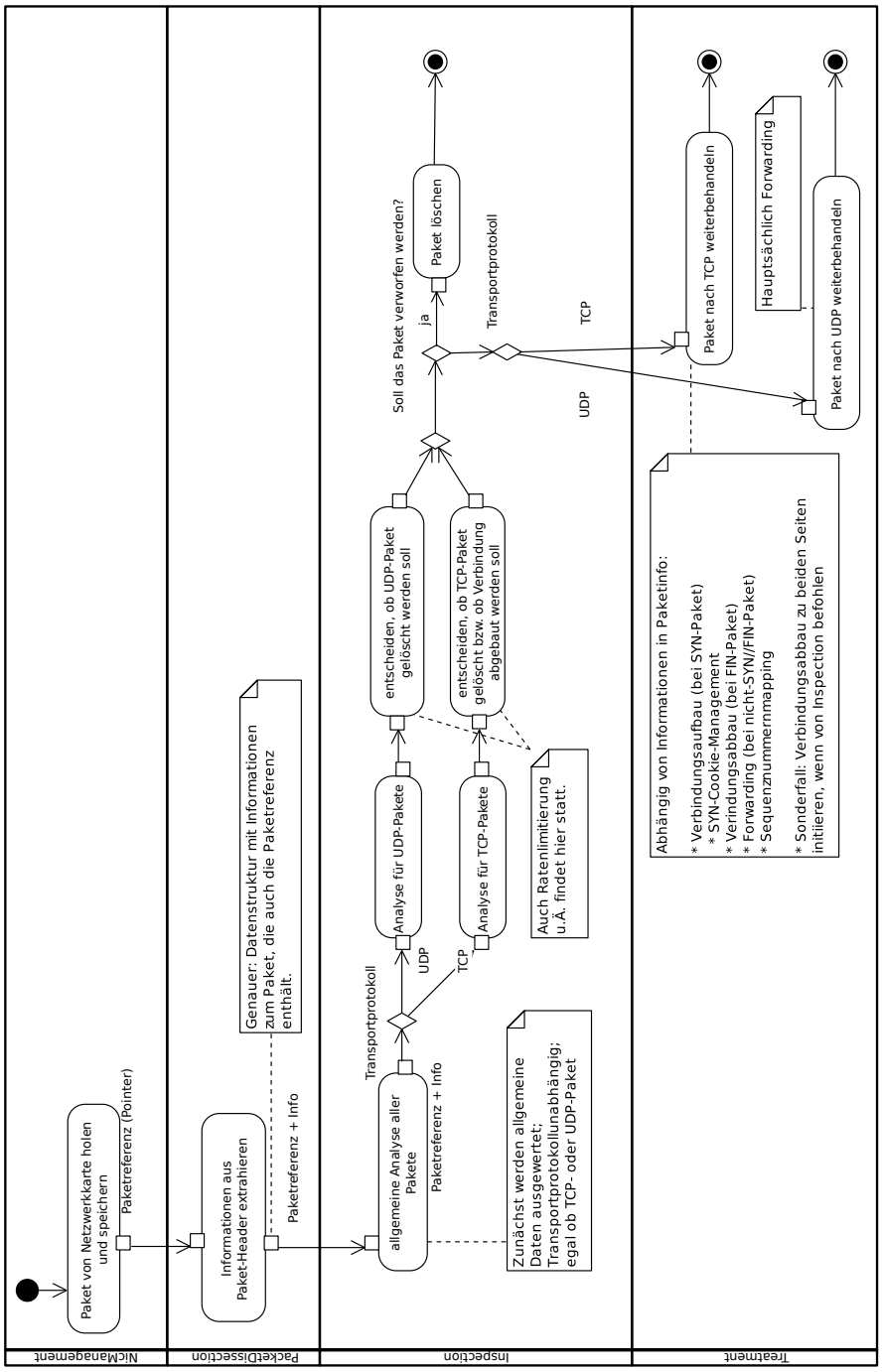


Abbildung 2.3: Schematische Darstellung des Kontrollflusses

2.2 Überarbeiteter Grobentwurf

Die in diesem Abschnitt erläuterten Änderungen wurden im Laufe der Implementierungsphase vorgenommen. Für das bei diesem Softwareprojekt genutzte Vorgehensmodell des Unified Process ist es typisch, dass sich auch während der Implementierung Änderungen am Entwurf ergeben. Für die Teammitglieder ist es besonders aufgrund der geringen Erfahrung bezüglich der Thematik des Projekts unerlässlich, wichtige Verbesserungen direkt vornehmen zu können.

2.2.1 Paketdiagramm

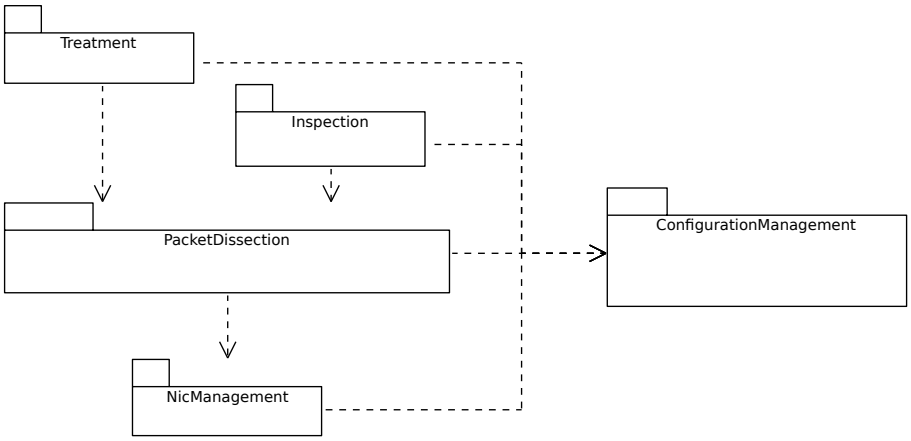


Abbildung 2.5: Paketdiagramm

Grundsätzlich ist es angedacht, wie im Paketdiagramm 2.5 ersichtlich, die zu entwickelnde Software in insgesamt 5 Teile zu untergliedern.

Das `NicManagement` wird eingesetzt, um die Kommunikation und Verwaltung der Netzwerkkarten und Ports zu ermöglichen, hier finden Operationen wie der Versand und Empfang von Paketen statt. Verwendet wird das `NicManagement` von der `PacketDissection`. Diese Komponente beinhaltet Klassen zur Paketrepräsentation für das `Treatment` und die `Inspection`. Sie liefert Operationen zum Löschen, Senden, Empfangen und Bearbeiten von Paketen. In der `PacketDissection` werden auch Informationen aus den einzelnen Headern eines Netzwerkpakets extrahiert.

Die extrahierten Informationen werden von der `Inspection` verwendet, um sowohl Angriffe erkennen zu können, als auch über den allgemeinen Zustand des Systems in Form von Statistiken Auskunft zu geben. Das `Treatment`, welches für die Abwehrwehrmaßnahmen der verschiedenen Angriffe zuständig ist, verwendet hierzu die von der `Inspection` bereitgestellten Ergebnisse und Informationen. Für das Versenden und Verwerfen von Pakten, sowie den Aufbau und das Terminieren von Verbindungen, verwendet das `Treatment` die `PacketDissection`, welche die Anweisung an das `NicManagement` weitergibt.

Sowohl `Treatment`, als auch `Inspection` und `PacketDissection` verwenden das `ConfigurationManagement`, welches Parameter für die Programmbestandteile in Form von Konfigurationsdateien vorhält. Das `ConfigurationManagement` bietet die Möglichkeit für den Nutzer, aktiv Einstellungen am System vorzunehmen.

2.2.2 NicManagement

Das `NicManagement` übernimmt, wie im letzten Review-Dokument erwähnt, das Senden, das Pollen und das Löschen von Paketen. Dieses Paket wurde eingeführt, um bestimmte Funktionen und Initialisierungsschritte vom DPDK zu kapseln. Dabei handelt es sich vor allem um folgende Operationen: `rte_eth_rx_burst()` und `rte_eth_tx_burst()` Es hat sich allerdings herausgestellt, dass die Operationen „Senden", „Empfangen" und „Löschen" in der Implementierung sehr wenig Aufwand bereiten. Das Zusammenbauen von Paketen wird von der Komponente `PacketDissection` übernommen. Der aufwändigere Teil ist die Initialisierung des DPDK, insbesondere die Ermöglichung von Multithreading und die Konfigurierung von symmetric Receive-Side-Scaling. Die dazu notwendigen Schritte werden jedoch von `Initializer` bzw in der main.cpp-Datei vor dem Starten der einzelnen Threads durchgeführt und sind nicht mehr Teil des `NicManagements`.

Aus diesem Grund und weil jeder nicht notwendige Funktionsaufruf Rechenzeit kostet, könnte das `NicManagement` aufgelöst und die bereitgestellten Funktionen an anderer Stelle implementiert werden. Die einzige Klasse, die das `NicManagement` zum jetzigen Zeitpunkt verwendet, ist die `PacketContainer`-Klasse in der Komponente `PacketDissection`. Es wäre möglich, den Inhalt der `NicManagement`-Aufgaben in diese Klasse zu verschieben.

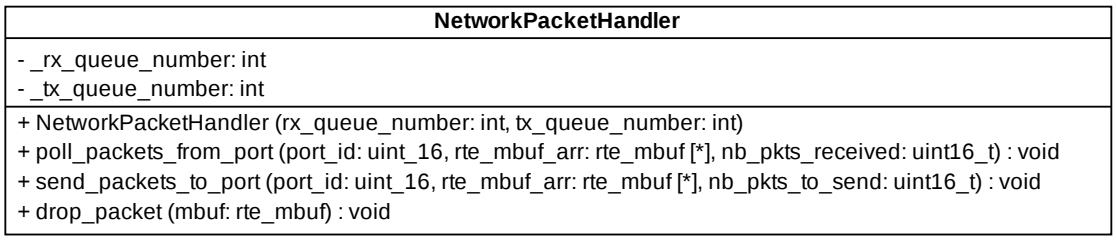


Abbildung 2.6: Klassendiagramm: `NetworkPacketHandler`

Um ein neues Objekt der Klasse `NetworkPacketHandler` zu erzeugen, muss der Konstruktor aufgerufen werden. Diesem müssen zwei Parameter übergeben werden (vgl. Abb. 2.6): Zum einen die ID der RX-Queues (`rx_queue_number`), zum anderen die der TX-Queues(`tx_queue_number`).

Für das Verwerfen von Paketen ist die Methode `drop_packet()` zuständig. Hierbei muss ein Pointer auf das Paket übergeben werden, das verworfen werden soll.

Die Methode `poll_packets_from_port()` holt Pakete von einem spezifischen Port. Dazu werden die ID des Ports, von denen die Pakete geholt werden sollen, ein Array, auf welches die Pointer der geholten mbufs geschrieben werden sollen und die Anzahl der Pakete, die in das Array geholt werden sollen, benötigt. Ähnliche Parameter werden der Methode `send_packets_to_port()` übergeben.

Folgende Änderungen ergaben sich während der Validierungsphase:

Es wurde festgestellt, dass, wie oben beschrieben, das Paket **NicManagement** nicht weiter benötigt wird. Es exisitiert zum jetzigen Zeitpunkt nicht mehr.

2.2.3 ConfigurationManagement

Das Paket `ConfigurationManagement` kümmert sich um die Initialisierung der Software. Des weiteren werden hier die ablaufenden Threads konfiguriert und verwaltet. Grundlegend ist das

Paket in drei Klassen eingeteilt: `Configurator`, `Initializer` und `Thread`.

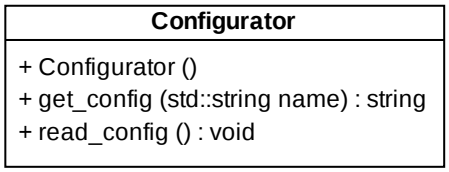


Abbildung 2.7: Klassendiagramm: `Configurator`

Die Klasse `Configurator` bietet eine Schnittstelle zu der Konfigurationsdatei, welche im Projekt liegt und die grundlegenden Einstellungen der Software enthält. An anderer Stelle kann dann über verschiedene Methoden auf Konfigurationsinformationen zugegriffen werden. Abbildung 2.7 zeigt das Klassendiagramm vom `Configurator`.

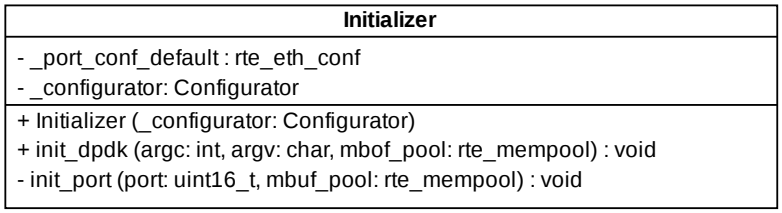


Abbildung 2.8: Klassendiagramm: `Initializer`

Die Klasse `Initializer` dient dazu die für die Bibliothek DPDK notwendigen Voraussetzungen zu schaffen. Das Klassendiagramm befindet sich in Abb. 2.8.

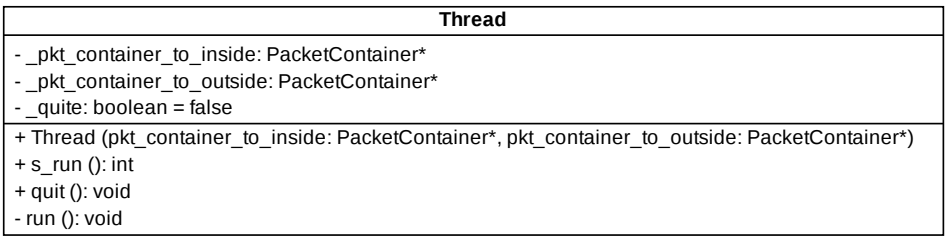


Abbildung 2.9: Klassendiagramm: `Thread`

Die Klasse `Thread` enthält den Ablaufplan für die Worker-Threads des Systems (vgl. Abb. 2.9).

2.2.4 PacketDissection

Der Zweck dieses Pakets ist, sämtliche Daten, die `Inspection` und `Treatment` für ihre Arbeit brauchen, aus den Paketen zu extrahieren.

Dafür war geplant, in dieser Komponente die Repräsentation eines Paketes - die Klasse `PacketInfo` - unterzubringen. Jedes Paket sollte einzeln repräsentiert durch die Pipeline des Programms gereicht werden. Es hat sich herausgestellt, dass dieses Vorgehen ineffizient ist. Näheres dazu ist im Feinentwurfskapitel beschrieben.

Aus diesem Grund wurde eine neue Klasse namens `PacketContainer` eingeführt. Diese dient als Repräsentation einer Folge von Paketen, die empfangen wurden. Enthalten sind sowohl die Pointer auf die tatsächlichen Pakete, als auch Metadaten in Form mehrerer Objekte der `PacketInfo`-Klasse. Im `PacketContainer` ist es möglich, Pakete zu entnehmen, hinzuzufügen und zu löschen. Weiterhin gibt es jeweils eine Methode zum Pollen neuer Pakete und zum Senden aller vorhandener Pakete.

Die `PaketInfo`-Klasse stellt immer noch alle relevanten Header-Informationen eines Paketes zur Verfügung. Allerdings werden Informationen nur noch auf Abruf extrahiert. Hierbei werden für die IP Versionen 4 und 6, sowie die Layer 4 Protokolle TCP, UDP und ICMP unterstützt. Darüber hinaus soll sie auch das verändern einzelner Informationen im Header ermöglichen.

Die letzte Klasse in der `PacketDissection` ist der namensgebende `HeaderExtractor`. Seine Aufgabe wandelte sich vom Extrahieren der Informationen zum Vorbereiten des Extrahierens auf Bedarf.

2.2.5 Inspection

Die zuvor globale Auswertung von Angriffen aller Threads durch eine einzige Instanz wurde ersetzt durch eine lokale threadeigene Auswertung. Berechnete Zahlen und Statistiken wie Paketrate und Angriffsrate werden per Interthreadkommunikation nur noch an eine globale Statistikinstanz gesendet. Dadurch können die Threads unabhängig voneinander agieren und reagieren, die Implementation der Methoden ist deutlich einfacher ausgefallen und die Interthreadkommunikation konnte auf ein Minimum begrenzt werden, was der Auswertungsgeschwindigkeit jedes Inspection-Threads zugute kommt und ein Bottleneck-Problem an der Inspection vorbeugt.

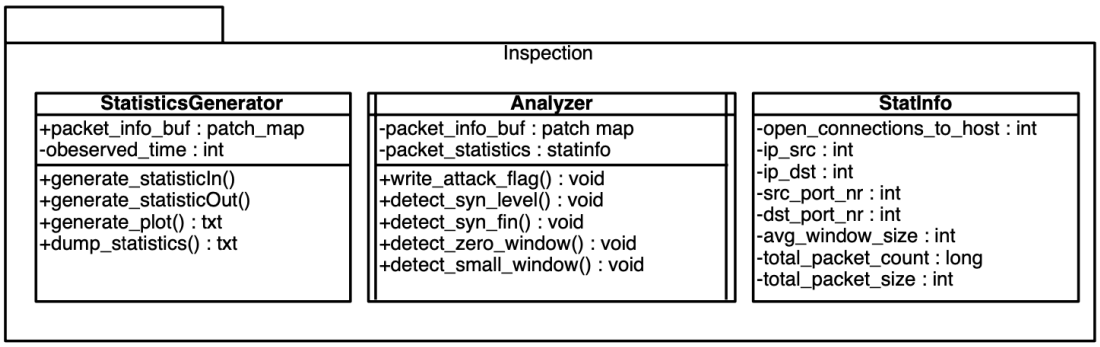


Abbildung 2.10: Altes Klassendiagramm der Inspection aus der Implementierungsphase

Durch den Einsatz von symetric Receive Side Scaling ist sowohl die Auslastung jeder Inspektion ausgeglichen und zusätzlich werden gleiche Paketströme (selbe Paketquelle und -Empfänger) durch denselben Thread verarbeitet. Dies erleichtert die Erkennung legitimer Pakete, da diese über eine eigene Patchmap für bestimmte Fälle von großteils illegitimen Verkehr unterscheidbar ist und die Variationen geringer sind.

Die Statistik wird statt durch eine eigene Klasse direkt in der `Inspection` erstellt und das Ergebnis an eine globale Statistik Instanz gesendet, um diese an den Nutzer auszugeben. Die `Inspection`-Klasse ist dadurch schlanker und folgt einem linearen Pipelinemodell für Paketverarbeitung.

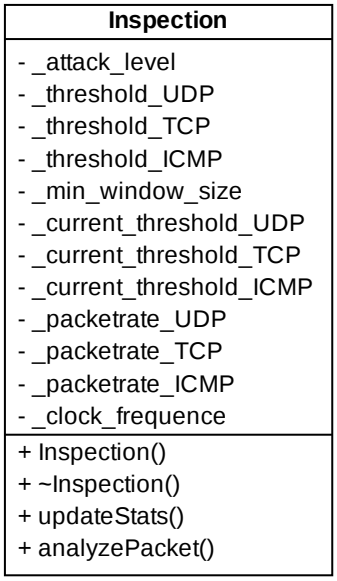


Abbildung 2.11: Aktuelles Klassendiagramm der `Inspection` aus der Planungs- und Entwurfsphase

In Abb. 2.10 und Abb. 2.11 lässt sich gut erkennen, wie sich die `Inspection` während der Überarbeitung verändert und vereinfacht hat.

2.2.6 Treatment

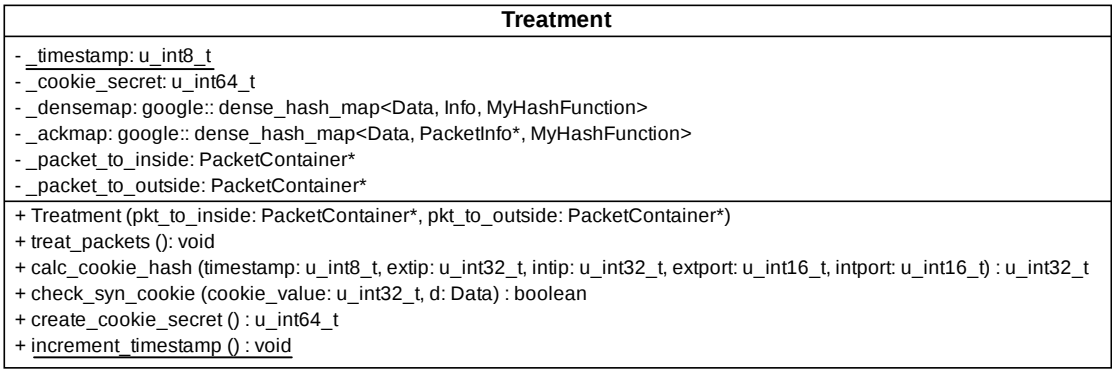


Abbildung 2.12: Klassendiagramm des Treatments aus der Implementierungsphase

Abbildung 2.12 zeigt das während der Implementierungsphase überarbeitete Klassendiagramm. Auf den ersten Blick unterscheidet sich dieses stark vom Grobentwurf des `Treatments` aus der Planungs- und Entwurfsphase (vgl. Abb. 2.13).

Das `Treatment` hat fortan die Aufgabe, die Implementierung von TCP-SYN-Cookies sowie die Realisierung eines TCP-Proxies zu übernehmen. Zur Realisierung des TCP-Proxies gehört insbesondere die Sequenznummernanpassung zwischen internen und externen Verbindungen.

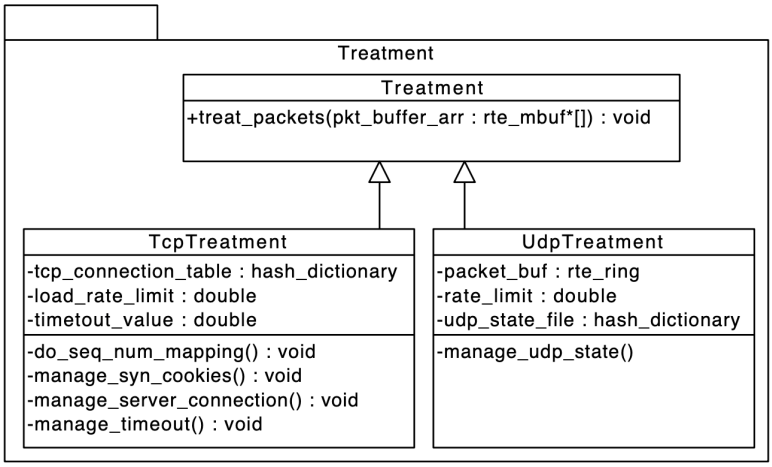


Abbildung 2.13: Altes Paket Treatments mit verschiedenen Klassen aus der Planungs- und Entwurfsphase

Es fällt auf, dass keine Vererbung mehr verwendet wird. Das heißt, dass nicht mehr zwischen `TcpTreatment` und `UdpTreatment` unterschieden wird. Der Grund hierfür ist die Auslagerung des UDP-Treatments in den `Inspection` (Paket `Inspection`). Es wird allerdings nicht nur das UDP-Treatment ausgelagert, sondern auch die Behandlung der SYN-FIN-Attacke sowie des TCP-Small- und Zero-Window-Angriffs. Dies ist darin begründet, dass bereits in der `Inspection` alle hierzu benötigten Informationen und Funktionalitäten bereitstehen. Dies führt letztlich dazu, dass Funktionsaufrufe oder function calls reduziert werden, welches es dem Programm ermöglicht, insgesamt eine bessere Performanz aufzuweisen. Durch den Wegfall der Klasse `UdpTreatment` entfällt die Notwendigkeit der Vererbung und die gesamte Implementierung des Treatments kann in einer einzigen Klasse erfolgen.

Das ursprüngliche Attribut `tcp_connection_table` wurde umbenannt in `_densemap`. Der Unterstrich vor dem Variablennamen zeigt, dass es sich um eine Member-Variable handelt. Durch die Umbenennung wird deutlich, dass es sich um eine Google-Densemap handelt und nicht um eine beliebige Map. Die Densemap untergliedert sich in drei Teile: `Data` ist der Key (vgl. Abb. 2.14), die Info ist die Nutzlast (vgl. Abb. 2.15) und der Hashwert aus `Data` ergibt die Position in der Map (vgl. Abb. 2.16). Hinzu kommt zusätzlich die `_ackmap`, bei der es sich ebenfalls um eine Densemap handelt. Die ACK-Map hat zur Aufgabe, diejenigen Pakete zwischenzuspeichern, welche im letzten ACK des externen Verbindungsaufbaus am System ankommen und nach erfolgreichem Verbindungsaufbau mit dem internen System an ebendieses weitergeleitet werden müssen. Der Wegfall von `load_rate_limit` und `timeout_value` ist ähnlich wie beim `UdpTreatment` durch die Auslagerung in der `Inspection` zu begründen. Die Variable `_timestamp`, die in der Implementierungsphase hinzugekommen ist, wird benötigt, um das Alter der ACKs, welche als Reaktion auf ein SYN-ACK erhalten werden, zu bestimmen. Das `_cookie_secret` wird im SYN-Cookie verwendet, um es einem potentiellen Angreifer schwieriger zu machen, eine illegitime Verbindung zum System aufzubauen, indem den Cookies ein weiterer schwieriger zu erratender Wert hinzugefügt wird. Bei den Variablen `_packet_to_inside` und `_packet_to_outside` handelt es sich um Pointer zu `PacketContainern`. Diese speichern die dem `Treatment` im Konstruktor übergebenen `PacketContainer`-Pointer für den weiteren internen Gebrauch.

Um ein Objekt der Klasse `Treatment` zu erzeugen, muss der Konstruktor aufgerufen werden

und die beiden Parameter `pkt_to_inside` und `pkt_to_outside` vom Typ `PacketContainer*` übergeben werden.

Die Sequenznummernzuordnung, die ursprünglich in der Methode `do_seq_num_mapping()` vorgenommen werden sollte, ist nun Teil der Methode `treat_packets()`, welche allumfassend für das gesamte Verbindungsmanagement des TCP-Verkehrs zuständig ist. Der Inhalt der Methode `manage_syn_cookies()` wurde mit der Überarbeitung auf verschiedene Methoden aufgeteilt: Der Hash-Wert des TCP-SYN-Cookies wird in der Methode `calc_cookie_hash()` berechnet. Das dazu benötigte Cookie-Secret ist der globale Wert `_cookie_secret`, der durch den Rückgabewert der Methode `create_cookie_secret()` initialisiert wird. Dieser Wert ändert sich während des Ablaufs des Programms nicht. `Check_syn_cookie()` vergleicht den Cookie eines ankommenden, zum Verbindungsaufbau gehörenden ACKs mit dem für diese Verbindung erwarteten Wert. Dazu wird der Methode unter anderem ein Pointer auf ein `Data`-Objekt übergeben. Der Aufbau der Klasse `Data` ist in Abb. 2.14 genauer dargestellt. Die Methode `manage_timeout()` wurde aus oben genannten Effizienzgründen und der Zugehörigkeit zur Behandlung der Sockstress-Attacken (TCP-Small- bzw. TCP-Zero-Window) ebenfalls in der `Inspection` verschoben. Die Methode `manage_server_connection()` wurde mit der Methode `treat_packets()` konsolidiert, um auch hier Funktionsaufrufe einzusparen.

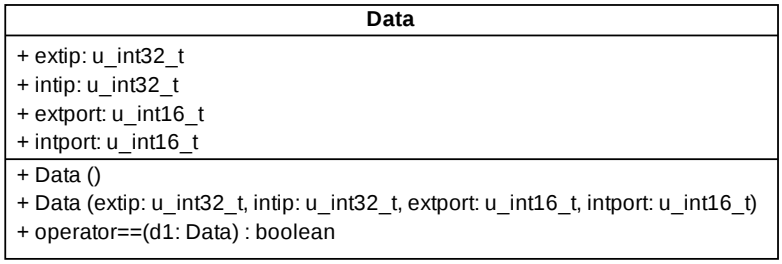


Abbildung 2.14: Klassendiagramm: `Data`

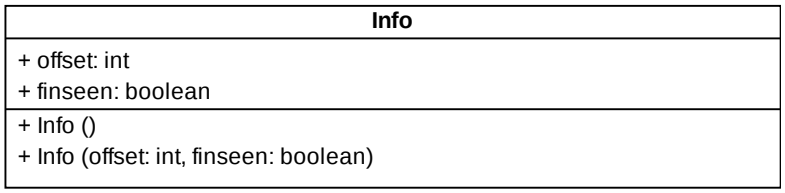


Abbildung 2.15: Klassendiagramm: `Info`

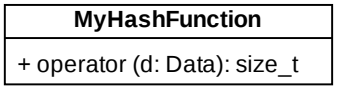


Abbildung 2.16: Klassendiagramm: `MyHashFunction`

Folgende Änderungen ergaben sich während der Validierungsphase:

Gemäß der Bennenungskonventionen wurde das Attribut `_timestamp` in `_s_timestamp` und die Methode `increment_timestamp()` in `s_increment_timestamp()` umbenannt.

Nach intensivem Testen der Realisierung mittels zwei Maps im Treatment stellte sich heraus, dass diese Realisierung aus performancetechnischer Sicht unvorteilhaft war. In Folge dessen wurden die Maps `_ackmap` und `_densemap` konsolidiert, sodass nunmehr insgesamt pro Verbindung ein Einfüge-, Such- und Löschvorgang gespart werden kann.

Die Methode `treat_packets()` wurde aufgeteilt: Die Funktion `treat_packets_to_inside()` bearbeitet diejenige Pakete, die als Ziel die internen Server haben. Die Funktion `treat_packets_to_outside()` bearbeitet diejenige Pakete, die als Ziel die externen Server haben.

Die Methode `create_cookie_secret()` wurde aufgrund der Kapselung in die Klasse `Rand` ausgelagert. Die dortige Methode `get_random_64bit_value()` ist statisch und verfügt über die gleiche Funktionalität.

Um die Kapselung der Klasse `Treatment` zu erhöhen, wurden außerdem alle Methoden bis auf `treat_packets_to_inside()`, `treat_packets_to_outside()` und `s_increment_timestamp()` auf `private` gesetzt. Diese drei sind weiterhin `public`, weil diese in der `main.cpp` und im `Thread` benötigt werden.

Die Klasse `info` wurde um vier Attribute ergänzt: `_finseen_to_inside`, `_finseen_to_outside`, `_ack_to_inside` und `_ack_to_outside`. Das Attribut `_finseen` wurde durch diese spezialisiertere Attribute nicht mehr benötigt. Die zusätzlichen Attribute werden benötigt, um den Verbindungsabbau vornehmen zu können. So wird nicht zu früh in den Zustand der Termination der Verbindung eingegangen.

Kapitel 3

Feinentwurf

In diesem Kapitel zum Feinentwurf wird der im vorherigen Kapitel beschriebene Grobentwurf verfeinert. Dabei wird für jedes Paket präzise erklärt, auf welche Art und Weise die entsprechende Komponente realisiert werden soll, sodass diese dann direkt implementiert werden kann. Zudem werden mithilfe von Aktivitätsdiagrammen die Kernfunktionen des jeweiligen Pakets übersichtlich und grafisch dargestellt.

3.1 ConfigurationManagement

Die folgenden Klassen sind im Paket `ConfigurationManagement` enthalten.

3.1.1 Configurator

Für die Software gibt es die Konfigurationsdatei `config.JSON`, welche innerhalb dieser Klasse eingelesen wird und diese Informationen global zur Verfügung stellt. Von dieser Klasse, dem `Configurator`, soll es im ganzen Programmablauf nur ein Objekt geben, damit keine Inkonsistenzen entstehen.

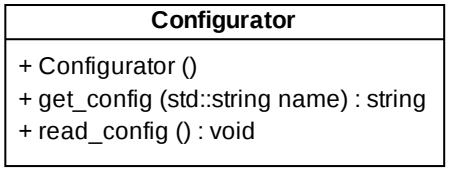


Abbildung 3.1: Klassendiagramm: `Configurator`

Aufgrund dieser Anforderungen kann ein spezielles Entwurfsmuster verwendet werden: Das Singleton. Das Singleton ist ein Erzeugungsmuster, welches automatisch dafür sorgt, dass nur eine Instanz des `Configurator` existieren kann, und es stellt ähnlich globalen Variablen Informationen global dar. Der Vorteil des Singleton besteht darin, dass das Singleton nur dann verwendet wird, wenn es wirklich benötigt wird. Die Klasse `Configurator` hat nur einen privaten Konstruktor, welcher in der zum Singleton gehörigen Methode der Instanziierung verwendet wird. In der ersten Verwendung des `Configurators` wird die Methode `read_config()` zur Einlesung

der Daten ausgeführt. Falls die Konfigurationsdatei nicht auffindbar ist, so wird eine Exception geworfen, da die Software ohne diese nicht ablaufen kann. Die ausgelesenen Daten werden dann in einem privaten JSON-Objekt hinterlegt. Hierzu wird `nlohmann::JSON` verwendet. Die Informationen der JSON-Datei werden über eine Schnittstelle `get_config(Datentyp)` anderen Klassen zur Verfügung gestellt, wobei es unterschiedliche Methoden je nach Datentyp gibt. Der explizite Aufruf des Auslesens erfolgt über die Methode `instance()`, mithilfe jener ein Zeiger auf das `Configurator`-Objekt zurückgegeben wird.

3.1.2 Initializer

Die Klasse `Initializer` dient dazu, dass grundlegende Initialisierungen für DPDK vorgenommen werden. Hierzu existiert die Methode `init_dpdk(int argc, char** argv)`.

3.1.3 Thread

Die Klasse `Thread` dient dazu, parallele Threads erzeugen zu können, welche dann die gesamte Paketbehandlung des Systems durchlaufen. Hierzu werden jedem Thread zwei `PacketContainer` übergeben. Ein `PacketContainer` dient zum Annehmen von Paketen, welche von außerhalb des Netzwerkes in das Netzwerk kommen. Der andere `PacketContainer` dient analog zum Annehmen von Paketen, welche von innerhalb des Netzwerkes nach außen sollen. Die `run()`-Methode der `Thread`-Klasse hat die Aufgabe, dass eine bestimmte Anzahl an Paketen geholt wird, und zwar mittels der Methode `poll_packets(int number)`. Dies gilt für beide `PacketContainer`. Nachdem die Pakete behandelt wurden, was innerhalb dieses Pollings vorgenommen wird, werden die restlichen Pakete mittels `send_packets()` in die jeweilige Richtung weitergeschickt.

3.2 PacketDissection

Die Aufgabe der `PacketDissection` ist es, Informationen über die zu untersuchenden Pakete bereitzustellen. Zudem wird die Kommunikation des `NicManagements` über die `PacketDissection` geleitet.

In Diagramm 3.2 wird das Polling von Paketen unter Benutzung des `PacketContainers` dargestellt. Der `PacketContainer` fungiert hierbei als zentrales Element zur Ablaufsteuerung.

3.2.1 PacketContainer

DPDK liefert beim Pollen von Paketen ein Array von Pointern auf sogenannte `mbuf`-Strukturen. Auch beim Senden muss dem Framework ein solches Array übergeben werden, denn die `mbuf`-Strukturen repräsentieren Pakete innerhalb von DPDK. Um nur die `PacketInfo`-Objekte durch das Programm zu reichen, wäre das Array von `mbuf`-Strukturen zu durchlaufen und die Pointer jeweils in die `PacketInfo`-Objekte zu schreiben. Ein `mbuf`(-Paket) gehört dabei genau einer `PacketInfo`. Wenn am Ende der Pipeline Pakete gesendet werden, müssten die Pointer der `mbuf`-Strukturen den `PacketInfo`-Objekten wieder entnommen und in ein Array geschrieben werden. Dies ist überflüssiger Aufwand, da es möglich ist, das empfangene `mbuf`-Array beizubehalten. Dies setzt der `PacketContainer` um.

Der `PacketContainer` ist keine aktive Klasse und wird aufgerufen, um spezielle, im Abb. 3.3 angegebene, Aufgaben umzusetzen. Eine dieser Aufgaben ist das Polling von Paketen, dessen Ablauf im Sequenzdiagramm 3.2 dargestellt wird. Eine weitere Aufgabe ist das Verwerfen von Paketen, welches durch `drop_packet(int index)` umgesetzt wird. Hierbei wird dem

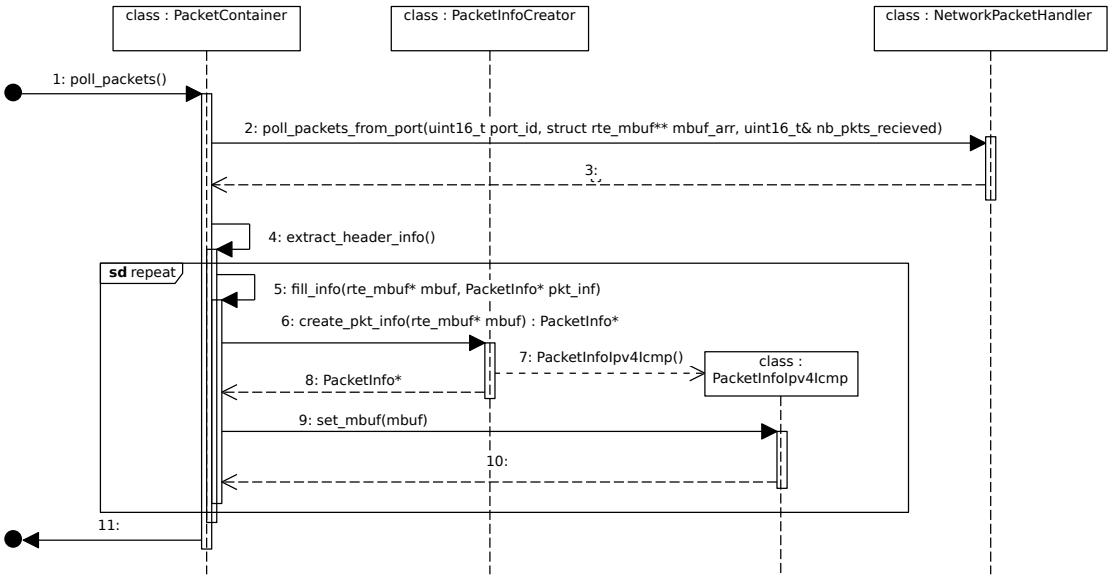


Abbildung 3.2: Sequenzdiagramm zum Polling von Paketen über den `PacketContainer`

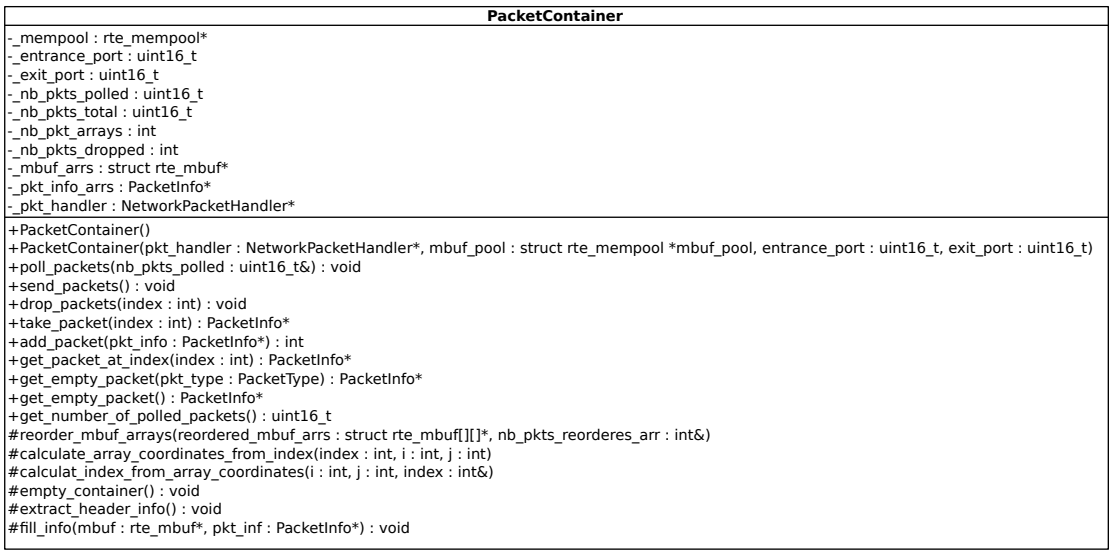


Abbildung 3.3: Klassendiagramm PacketContainer

`NetworkPacketHandler` mitgeteilt, welcher `mbuf` verworfen werden soll und die Referenzen im `PacketContainer` selbst gelöscht.

Es ist aber auch möglich, mittels der Methode `take_packet(int Index):PacketInfo*` Pakete aus dem `PacketContainer` zu entfernen, ohne sie zu löschen. Dafür werden nur die `PacketContainer`-internen Referenzen auf den `mbuf` und seine `PacketInfo` zu Nullreferenzen gesetzt und die `PacketInfo` zurückgegeben. Diese entnommenen Pakete können später wieder mit `add_packet(PacketInfo* pkt_info):int` eingefügt werden. Dafür wird dieses Paket hinter den bereits existenten Pakete im `mbuf`-Array gespeichert. Selbiges wird für die zugehörige `PacketInfo` gemacht. Zurückgegeben wird der Index, unter dem das neue Paket zukünftig erreichbar sein wird. Es können nicht nur zuvor entnommene Pakete einem `PacketContainer` hinzugefügt werden, sondern auch komplett neue. Dieses Erstellen von Paketen ist mit dem Befehl `get_empty_packet(PacketType pkt_type):PacketInfo*` möglich. Hierbei wird für einen neuen `mbuf` Speicher aus einem `mempool` alloziert und eine zugehörige `PacketInfo` vom gewünschten `PacketType` erstellt. Mithilfe dieser `PacketInfo`, kann der Paketkopf im Anschluss befüllt werden. Zuletzt müssen all diese Pakete auch wieder mit `send_packets()` versendet werden. Dafür wird das `mbuf`-Array, falls notwendig, umsortiert, da durch `drop_packet(int index)` Lücken entstehen können und DPDK nicht mit Nullreferenzen umgehen kann. Zuletzt wird das Array über den NetworkPacketHandler an DPDK zur Versendung übergeben.

Auch wenn bisher immer nur von je einem Array für `mbuf`s und `PacketInfos` gesprochen wurde, können es mehrere werden. Es gibt bei DPDK eine sogenannte BurstSize, welche angibt, wie viel Pakete maximal auf einmal entgegengenommen und wieder versendet werden können. Daran sind auch die Größen der Arrays angepasst. Da es aber durch Maßnahmen des `Treatments` und `Inspection` zur Erzeugung von neuen Paketen kommen kann, gibt es zusätzliche Arrays, falls die ersten bereits voll sind. Die Verwaltung dieser Arrays ist in allen Funktionen enthalten und hat nach außen keinen sichtbaren Effekt.

3.2.2 PacketInfo

Die genaue Umsetzung, sowie die daraus resultierende Befüllung hat sich im Laufe der Entwicklungsphase sehr stark verändert. Dies hatte vor allem Performance-Gründe. In der aktuellen Variante ist die `PacketInfo` selbst nur für die Verwaltung des `mbuf`s sowie das Speichern seines Layer 3 und 4-Protokolls verantwortlich.

Um ausschließlich notwendige Informationen zu speichern, wird diese `PacketInfo` in eine protokollspezifische Variante gecastet. Diese spezialisierten Varianten erben von der eigentlichen `PacketInfo` und erweitern sie um Getter- und Setterfunktionen für die relevanten Header-Informationen ihrer jeweiligen Protokolle.

Auch wenn in Diagramm 3.4 `PacketInfos` mit IPv6 aufgeführt werden, sind diese noch nicht funktionsfähig. Es wurde sich entsprechend der Anforderungen zuerst auf IPv4 konzentriert.

3.2.3 HeaderExtractor

Wie bereits erwähnt, wurde die Extraktion der Headerinformationen dezentralisiert und wird nur bei Abruf entsprechender Informationen durchgeführt. Dies führte zu einer Verringerung von Code für den `HeaderExtractor` im Laufe der Entwicklung, weshalb er in den `PacketContainer` integriert wurde. In obigen Sequenzdiagramm stellt er die Funktionen `extract_header_info()` und `fill_info(rte_mbuf* mbuf, PacketInfo* pkt_inf)`.

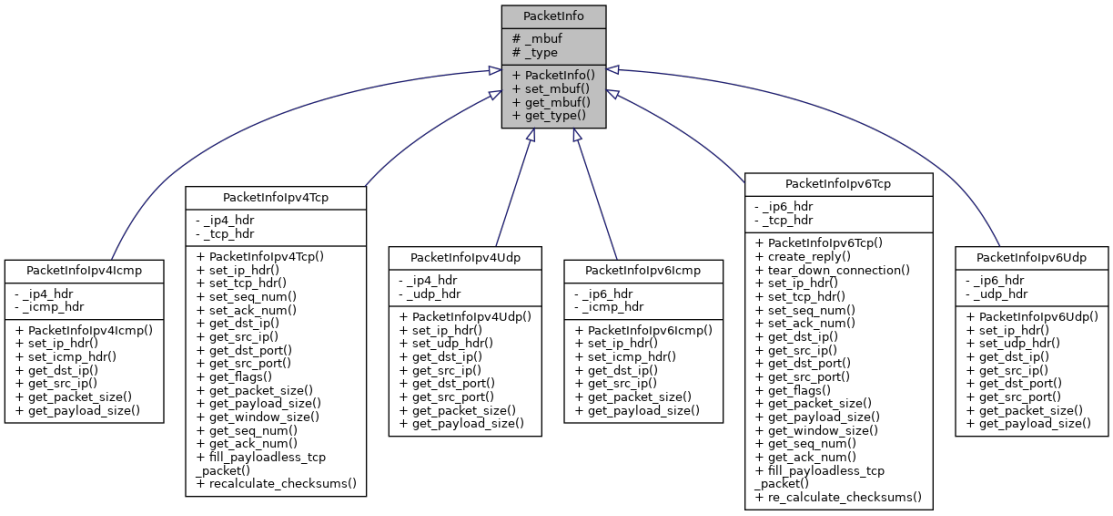


Abbildung 3.4: Klassendiagramm aller PacketInfo Varianten

Dabei wird in `extract_header_info()` über die einzelnen Elemente des `PacketContainers` iteriert und für jeden `mbuf` die Funktion `fill_info(rte_mbuf* mbuf, PacketInfo* pkt_inf)` aufgerufen, welche wiederum den `PacketInfoCreator` ausführt und den `mbuf` mit der zugehörigen `PacketInfo` verknüpft.

3.2.4 PacketInfoCreator

Diese Klasse ist ein Hilfsmittel, um Vererbungsketten zu vermeiden. Ihre Aufgabe ist es, die zum Paket passende `PacketInfo`-Version zu erzeugen. Dabei liest der `PacketInfoCreator` die Layer 3 und Layer 4-Protokoll-IDs aus, legt die entsprechenden `structs` auf den Speicher und speichert sie in der frisch erzeugten `PacketInfo`.

3.3 Inspection

Die `Inspection` ist für die Erkennung böswilliger IP Pakete zuständig und untersucht diese daher auf verdächtige Strukturen und Muster. Dazu wird auch eine eigene lokale Statistik erstellt, zur Auswertung genutzt und zur Informationsweitergabe mit einer globalen Statistik geteilt.

Der `Initializer` erstellt für jeden genutzten Thread eine eigene Inspektion, welche alle Pakete dieses Threads analysiert und DDoS-Attacken erkennt. Dazu wird der `Inspection` jeweils ein `PacketContainer` übergeben, der eine Menge von Paketen enthält, die über das `NicManagement` eingegangen sind.

Die Erkennung basiert auf einer Mustererkennung von zeitlich aufeinanderfolgenden Paketen nach einer Auftrennung in die Protokolle UDP, TCP und ICMP. UDP und ICMP Pakete werden ausschließlich mit einem vorher festgelegten Threshold geprüft, der sich an eine selbst berechnete Angriffsrate anpasst. TCP Pakete werden zusätzlich auf Zero und Small Window sowie auf SYN-FIN und SYN-FIN-ACK Muster überprüft.

Der Ablauf und Reihenfolge der Prüfungen der `Inspection` ist aus einer Versuchsdurchführung

mit einem Decision Tree für DDoS-Abwehr entstanden, um einen möglichst schnellen und effizienten Ablauf zu finden. Implementiert wurde eine statische, nicht veränderliche Pipeline, die nach den größten auszuschließenden Faktoren jedes Pakets vorgeht.

Der Ablauf kann grob in drei Filterstufen, auch Security Layers genannt, unterteilt werden:

1. RFC Compliance
2. Static Rules
3. Dynamic Rules

Ob ein Paket dem RFC Standard entspricht, wird bereits bei der `PacketInfo` klar. Die Klasse `Inspection` bietet die Möglichkeit, bestimmte Fehler zuzulassen oder Pakete mit bestimmten Fehlern zu blockieren und zu löschen.

Die zweite Stufe der Filter setzt sich aus fest definierten Angriffen und Angriffsmustern zusammen. So sind zum Beispiel bei SYN-FIN und SYN-FIN-ACK Angriffen immer die Flags SYN und FIN oder SYN, FIN und ACK gesetzt. So können diese sofort erkannt und das Paket verworfen werden. Weitere Angriffe, die in der statischen Abwehr erkannt werden, sind Zero- und Small-Window Angriffe.

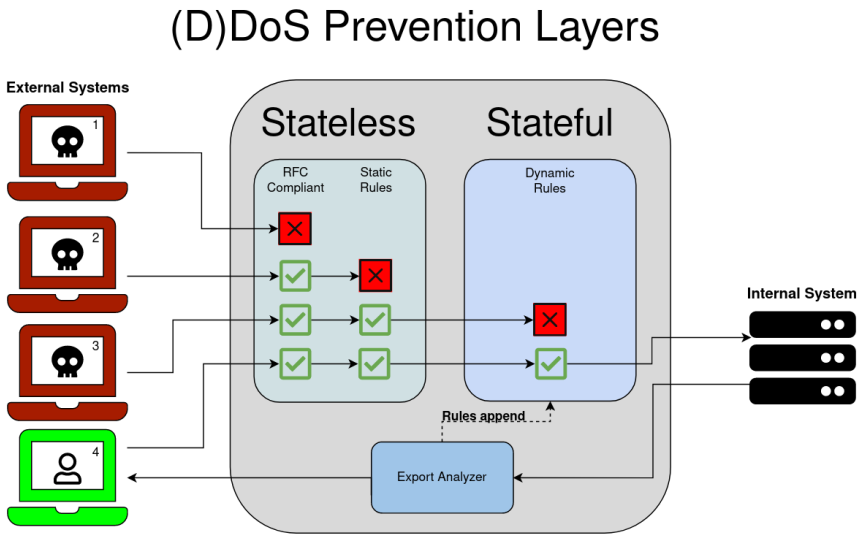


Abbildung 3.5: Stufen der Sicherheit

In der dynamischen Filterstufe werden die Filterregeln entsprechend dem aktuellen Netzwerkverkehr und vorher eingegangenen Paketen angepasst. So dient ein Limit der Paketrate (engl. Threshold) dazu, UDP und TCP Floods abzuwehren. Eigene Verbindungstabellen der ausgehenden Netzwerkverbindungen lassen jedoch legitime Pakete, die als Antwort auf eine bestehende Verbindung dienen, weiterhin zu, um den legitimen Netzwerkverkehr nicht einzuschränken.

Die Verknüpfung und Ablauf der Filterung wird in der Abbildung. 3.5 vereinfacht dargestellt.

Im Diagramm 3.5 muss Folgendes unterschieden werden: Die Computer 1 bis 3 sind Angreifer mit unterschiedlichen Angriffen, die ebenso in unterschiedlichen Filterstufen als Angriff erkannt werden und Computer 4 als Nutzer mit legitimen Anfragen an den Server, die den Filterregeln entsprechen. Ausgehender Verkehr aus dem internen System wird grundsätzlich vertraut und nicht zusätzlich gefiltert. Jedoch wird ausgehender Verkehr analysiert, um die dynamischen Regeln anzupassen.

Nach jedem Durchlauf eines `PacketContainer`s werden die lokalen und globalen Statistiken aktualisiert. Die Weitergabe der Informationen an die Statistik erfolgt über einen eigenen Interthread-Kommunikationskanal zum globalen Statistik-Thread. Die globale Statistik führt alle einzelnen Informationen zusammen und macht sie dem Nutzer in einfacher Weise abrufbar.

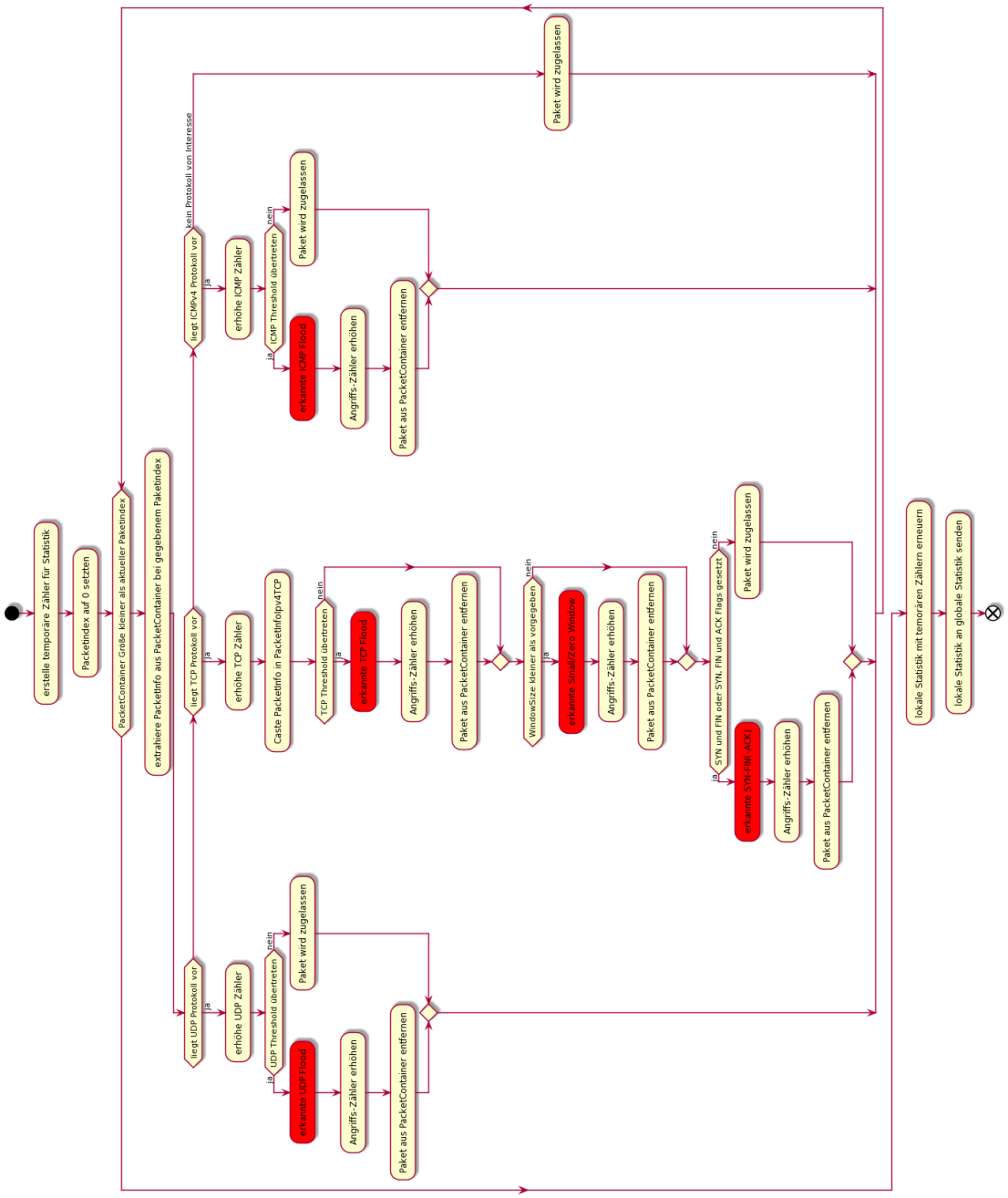


Abbildung 3.6: Aktivitätsdiagramm der Methode `analyzeContainer()` der Inspection

3.4 Treatment

Das `Treatment`, welches für die Behandlung der SYN-Flut zuständig ist, erhält vom `Thread` zwei Pointer auf `PacketContainer`. Für jede Senderichtung, sowohl von Intern nach Extern als auch umgekehrt, existiert einer dieser Container. Der Ablauf in der Behandlung von Paketen unterscheidet sich basierend auf deren Senderichtung. Jedes Paket wird im `Treatment` zwar einzeln, allerdings im Kontext der gesamten Verbindung betrachtet. Die Behandlung im `Treatment` beginnt mit dem Iterieren über die Einträge im jeweiligen `PacketContainer`. Hierbei wird zugleich geprüft, ob das gerade betrachtete Paket bereits gelöscht wurde, oder von einem Typ ist, welcher nicht im `Treatment` behandelt wird. Dies ist auch im globalen Ablauf der Funktion `treat_packets()` in Abbildung 3.7 sowie 3.8 zu erkennen. Sollte dies der Fall sein, wird ebendieser Eintrag übersprungen. Sollte es sich bei dem gerade betrachteten Paket beispielsweise um ein UDP-Paket handeln, so wird dieses im Treatment nicht weiter betrachtet, da dies bereits in der `Inspection` geschah.

Nach diesen ersten Tests findet jeweils eine Fallunterscheidung statt. Für Pakete, welche von extern nach intern geschickt werden sollen, gilt:

Falls es sich bei dem Paket um ein TCP-SYN-Paket handelt, so wird als Antwort hierauf ein SYN-ACK generiert, dessen Sequenznummer durch einen, vom Programm berechneten, SYN-Cookie ersetzt wird. Hierzu existiert die Methode `calc_cookie_hash()`, welche 24 der 32 Bit langen Sequenznummer generiert, welche später mit einem 8 Bit Zeitpunkt (engl. „Timestamp“) in der Methode `treat_packets()` aufgefüllt werden. Dieser SYN-Cookie enthält Informationen über die Verbindungsentitäten, sowie zur Verbesserung der Effektivität einen Zeitstempel und ein Secret. Dieser SYN-Cookie ermöglicht es, im Verlauf des Verbindungsaufbaus auf das Speichern von Informationen über die Verbindung zu verzichten. Somit wird die Angriffsfläche von SYN-Floods effektiv minimiert.

Sollte ein ACK als Reaktion auf dieses SYN-ACK erhalten werden, so ist durch die Funktion `check_syn_cookie()` zu überprüfen, ob der empfangene Cookie in Form der Sequenznummer plausibel ist (Siehe Abb. 3.9). Das bedeutet, dass der Zeitstempel maximal eine Zeiteinheit alt ist, welche in diesem Programm 64 Sekunden dauert, und der restliche Cookie mit dem erwarteten Cookie übereinstimmt. Der Cookie setzt sich insgesamt zusammen aus 8 Bit Timestamp, sowie 24 Bit Hashwert über externe und interne IP-Adresse, externe und interne Portnummer sowie dem Timestamp und dem Cookie_Secret. Des weiteren ist eine Verbindung mit dem internen Server, spezifiziert in der DestIP des ACK-Paketes, aufzubauen. Zudem muss die dem ACK hinzugefügten Payload gespeichert werden, auch dies geschieht in einer separaten Map, der ACKmap. Dieses ACK-Paket muss nach erfolgreichem Verbindungsaufbau mit dem internen Server an ebendiesen verschickt werden.

Wird ein SYN-ACK von extern empfangen, so ist dies ohne Veränderung an das interne Netz zuzustellen. Allerdings muss hier ein Eintrag in der Offsetmap erzeugt werden, wobei der Offset realisierungsbedingt null ist.

Werden Pakete ohne gesetzte Flags, beziehungsweise nur mit gesetztem ACK-Flag verschickt, so findet eine Sequenznummernzuordnung und eine Anpassung von Sequenznummern statt. Hierzu wird eine Densemap mit individueller Hashfunktion, in diesem Fall XXH3, verwendet. Bei der Densemap handelt es sich um eine besonders effiziente Hashmap, welche ein Einfügen, Suchen und Löschen in bis zu vier mal weniger Zeit als eine unordered_map ermöglicht. Die Auswahl der Hashfunktion XXH3 ist dadurch motiviert, dass sie extrem schnell ist und dennoch kaum Kollisionen erzeugt. Insbesondere werden durch sie bereits auf handelsüblichen Computersystemen Hashraten von bis zu 31.5 Gbit/s erzielt.

Der Ablauf bei Empfang eines solchen Paketes ist wie folgt: Bei eingehenden Paketen wird ein zuvor berechneter Offset, welcher in der Offsetmap für jede Verbindung gespeichert ist, von der ACK-Nummer subtrahiert.

Wird ein ACK empfangen, welches zu einer Verbindung gehört, in deren Info finseen auf true gesetzt ist, so muss die ACK-Nummer angepasst, das Paket an den internen Server geschickt und der Eintrag in der Densemap verworfen werden.

Falls ein Paket mit gesetztem RST-Flag von extern empfangen wird, wird der Eintrag in der Densemap gelöscht und das empfangene Paket an den internen Server weitergeleitet. Hierbei muss keine Anpassung der ACK-Nummer vorgenommen werden.

Sollte ein FIN empfangen werden, so muss im Info-Struct, welches Teil der Offsetmap ist, der Wert finseen auf true gesetzt werden. In diesem Fall ist das Paket nach Anpassung der ACK-Nummer weiterzuleiten.

Im zweiten Fall der übergeordneten Fallunterscheidung erhält das Programm anschließend den `PacketContainer` der Pakete, welche das Netz von intern nach extern verlassen wollen. Auch hier wird, bevor ein Paket der Behandlung unterzogen wird, geprüft, ob das Paket nicht bereits gelöscht wurde, oder es sich um ein Paket falschen Typs handelt.

Erhält das System ein SYN-Paket von einem internen Server, so wird dieses an das im Paket spezifizierte Ziel weitergeleitet. Eine Anpassung der Sequenznummer findet in diesem Fall nicht statt.

Erhält das System ein SYN-ACK aus dem internen Netz, so muss das System die Differenz aus der ACK-Nummer dieses Pakets, und der des in der ACKmap gespeicherten Paketes berechnen, und den Wert als Offset in der Offsetmap eintragen. Das von intern empfangene SYN-ACK Paket muss verworfen werden. Das zuvor in der ACKmap zwischengespeicherte Paket muss nun mit angepasster ACK-Nummer intACK = extACK-offset an den internen Host geschickt werden.

Wird ein Paket ohne gesetzte Flags oder mit gesetztem ACK-Flag von Intern nach Extern verschickt, so findet eine weitere Fallunterscheidung statt. Im Fall, dass finseen bereits auf true gesetzt ist, muss der Offset in der Offsetmap nachgeschlagen werden, der Eintrag daraufhin gelöscht werden und das empfangene Paket mit extSeq = intSeq + offset verschickt werden. Gesetzt den Fall, dass noch kein Eintrag in der Offsetmap existiert, muss ein neuer Eintrag in dieser erstellt werden. Der Offsetwert muss auf null, und finseen auf false gesetzt werden. Das empfangene Paket muss hiernach nach Intern weitergeleitet werden. Trifft keiner der beiden obigen Fälle ein, so muss der Offset in der Offsetmap nachgeschlagen werden und das empfangene Paket nach Intern weitergeschickt werden. Vor dem Versenden muss hierbei die Sequenznummer wie folgt angepasst werden: extSeq = intSeq + offset.

Sollte ein Paket mit gesetztem FIN-Flag erkannt werden, so ist diese Information in der Info an Stelle Key = Hash(extip, intip, extport, intport) mit dem Vermerk finseen = true, zu speichern. Das empfangene Paket ist durch das System nach extern mit extSeq = intSeq + offset weiterzuschicken.

Wird ein RST erhalten, so ist eine Anpassung der Sequenznummer vorzunehmen, das Paket entsprechend weiterzuleiten und der Eintrag in der Offsetmap an entsprechender Stelle zu entfernen.

Des weiteren könnte es unter Umständen erforderlich werden, die Einträge mit einem Timestamp zu versehen, welcher speichert, wann dieser Eintrag zuletzt verwendet wurde, sollte es zu Situationen kommen, in denen sowohl Sender als auch Empfänger die Verbindung nicht korrekt

terminieren können. Dies ist bisweilen nicht implementiert, die Idee wird allerdings basierend auf den Ausgängen der Tests auf dem Testbed weiter verfolgt oder verworfen.

Nachdem ein ACK als Reaktion auf ein SYN-ACK bei dem zu entwerfenden System angekommen ist, wird die Methode `check_typ_syn_cookie()` aufgerufen. Grundsätzlich wird hier überprüft, ob der Hash-Wert aus dem empfangenen Paket mit dem eigens berechneten Hash-Wert übereinstimmt. Falls dies nicht der Fall ist oder die Differenz der Zeitstempel zu groß ist, wird ein Paket mit gesetzten Reset-Flag (RST) an den Sender geschickt. Dieses Flag zeigt an, dass die Verbindung beendet werden soll. Andernfalls wird die Verbindung als legitim erkannt und das Paket in der ACKmap zwischengespeichert, bis die Verbindung mit dem internen System erfolgreich war.

Abbildung 3.10 zeigt die parameterlose Methode `create_cookie_secret()`. Zu Beginn werden drei 16 Bit lange Zufallszahlen generiert, wobei auf die Funktion `rand()` aus der C Standardbibliothek zugegriffen wird. Der erste mit `rand()` generierte Wert wird um 48 Bit nach links verschoben, der zweite um 32 Bit. Diese beiden Werte werden danach bitweise ODER miteinander verknüpft. Dieser verknüpfte Wert wird dann wiederum mit der dritten zufälligen 16 Bit Zahl bitweise ODER verknüpft. Das Ergebnis dieser Verknüpfung ist eine 64 Bit lange Zufallszahl, die von der Methode zurückgegeben wird.

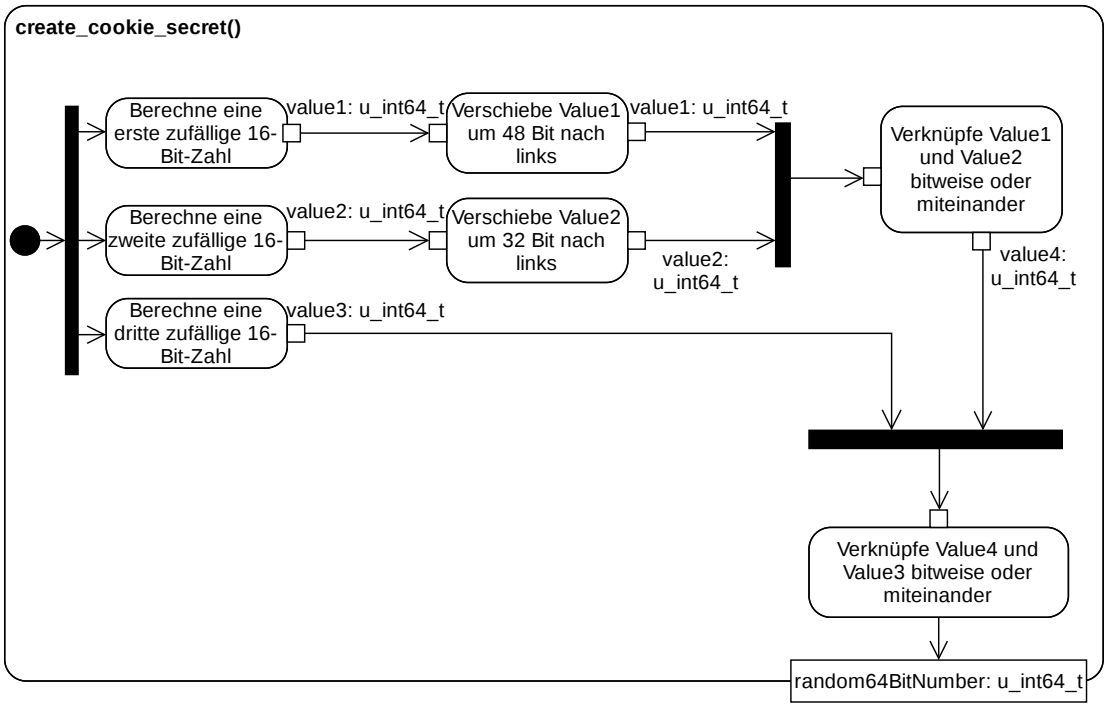


Abbildung 3.10: Aktivitätsdiagramm der Methode `create_cookie_secret()`

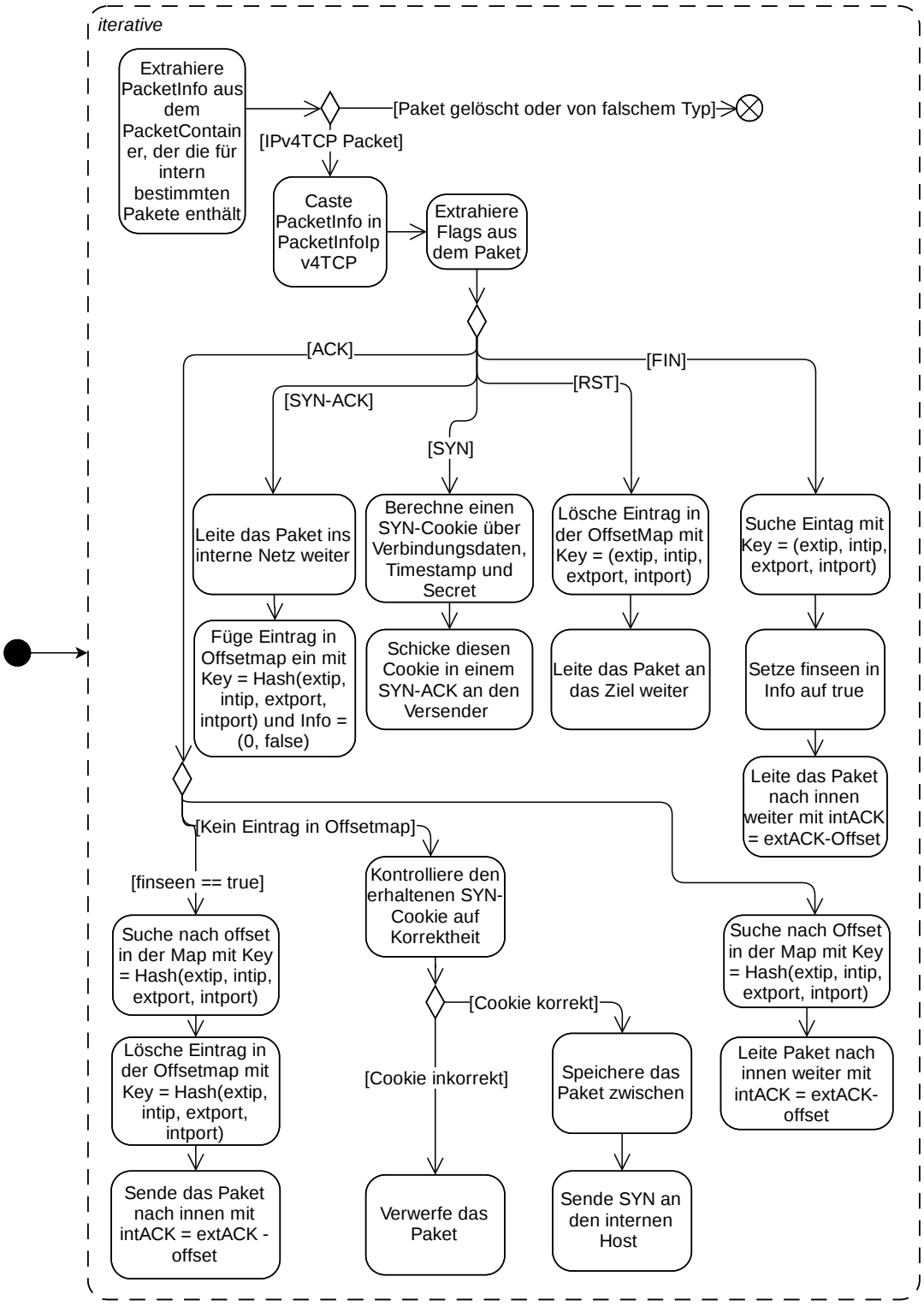


Abbildung 3.7: Aktivitätsdiagramm der Methode `treat_packets()`, Teil: Pakete nach Intern

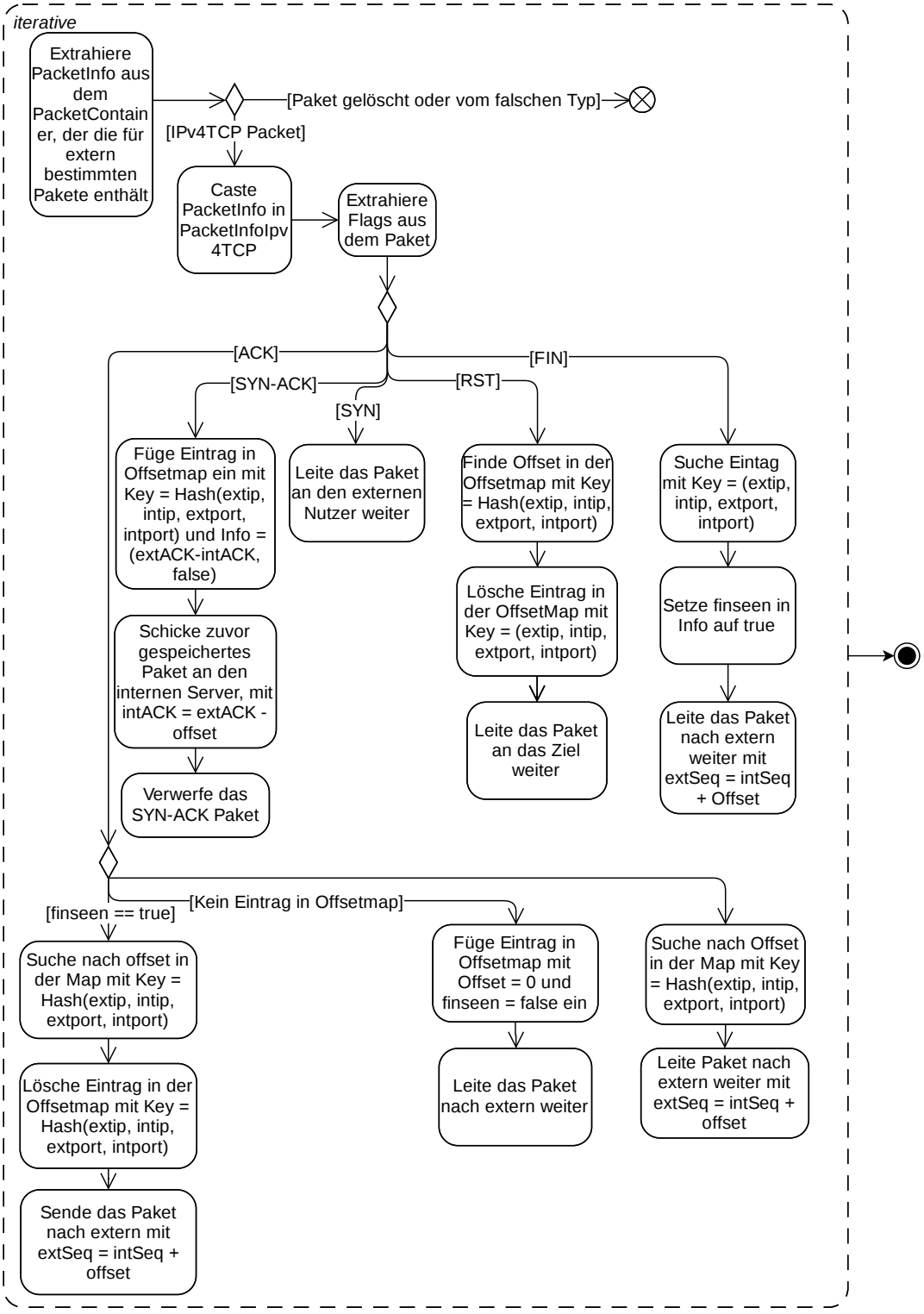


Abbildung 3.8: Aktivitätsdiagramm der Methode `treat_packets()`, Teil: Pakete nach Extern

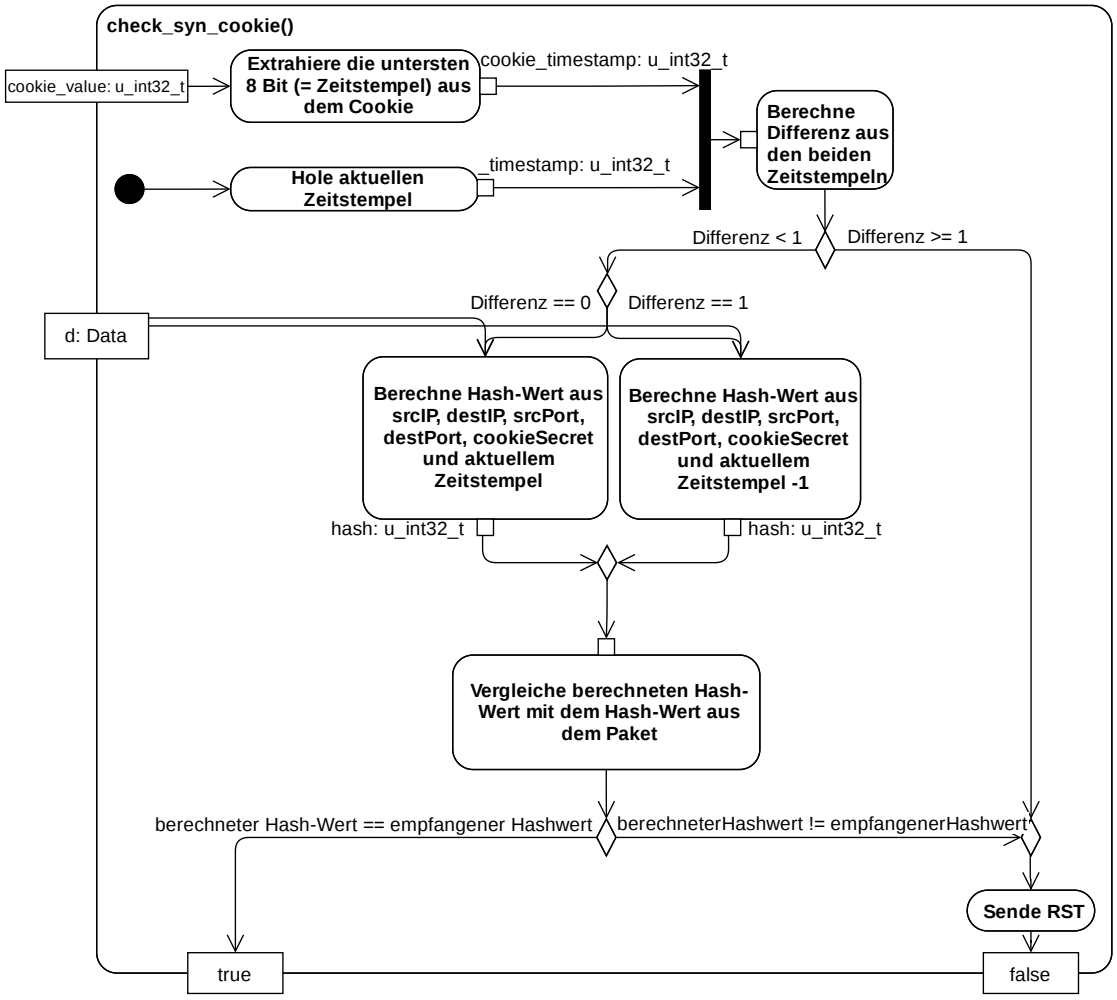


Abbildung 3.9: Aktivitätsdiagramm der Methode `check_syn_cookie()`

Kapitel 4

Testdokumentation

Im vorliegenden Softwareprojekt wurden verschiedene Arten von Tests durchgeführt. Zum einen wurden die einzelnen Komponenten mit Hilfe von Unit-Tests auf ihre korrekte Funktionalität geprüft, zum anderen wurde direkt auf dem Testbed getestet. Der Aufbau des Testbeds ist in Abbildung 2.2 (siehe Seite 9) dargestellt.

4.1 Unit-Tests

In sogenannten Unit-Tests, die auch als Modultest oder Komponententest bezeichnet werden, geht es um den Test einzelner Teile der Software. Unter Testen wird das Überprüfen, ob das Modell dem System entspricht, verstanden. Dies kann jedoch nur die Anwesenheit von Fehlern, nicht aber deren Abwesenheit nachweisen. Fast jede Komponente und die jeweils darin enthaltenen Klassen haben eine eigene Testdatei, in der die Unit-Tests ausgeführt werden konnten.

Da die Unit-Tests im vorliegenden Softwareprojekt bereits in der Implementierungsphase durchgeführt werden konnten, noch vor der eigentlichen Validierungsphase, war es möglich, Fehler bereits frühzeitig zu erkennen. Ein weiterer Vorteil des Unit-Testens besteht darin, dass beim Auftreten eines Fehlers dieser sehr genau eingegrenzt werden kann. Somit kann dieser Fehler schneller gefunden und dann auch behoben werden. Außerdem ermöglichen Unit-Tests eine parallele Bearbeitung, denn das Testbed existiert schließlich nur einmal.

4.1.1 Mocking: `libdpdk_dummy`

Mocking (dt.: Nachbildung oder Imitation) findet innerhalb der Unit-Tests Verwendung, um so isolierte Tests durchführen zu können. Da in diesem Projekt Tests bereits frühzeitig stattfinden sollten, sich DPDK jedoch nicht mit Unit Tests kompilieren ließ, wurde die Mocking-Bibliothek `libdpdk_dummy` erstellt. Diese kleine Bibliothek weist zwar eine geringere Funktionalität als das komplette DPDK-Framework auf, setzt aber genau das um, was bei den Unit-Tests gebraucht wird. So wurden genau die Header-Dateien nachgebildet, deren Funktionalitäten beim Testen benötigt wurden. Diese Nachbildung entstand durch Kopieren aus den „Original-DPDK-Headern“ und individuelle Anpassung an die Anforderungen des Projekts.

```
1 TEST_CASE("rte_mbuf", "[]"){
2     struct rte_mbuf* mbuf;
```

```
3     struct rte_mempool* mempool = nullptr;
4
5     mbuf = rte_pktmbuf_alloc(mempool);
6     CHECK(mbuf != nullptr);
7 }
```

Codeausschnitt 4.1: Unit-Test zu `lipdpdk_dummy`

Auch zu `lipdpdk_dummy` existiert ein Unit-Test, um zu überprüfen, ob sie so wie beabsichtigt arbeitet (vgl. Codeausschnitt 4.1). Hier werden zuerst Pointer auf einen `rte_mbuf` und einen `rte_mempool` angelegt. Danach wird überprüft, ob die Methode `rte_pktmbuf_alloc()` richtig arbeitet, indem gecheckt wird, ob nach der Allokation in `mbuf` kein Nullpointer liegt. Auf das triviale Löschen des `mbuf`s wird an dieser Stelle verzichtet, weil es bei diesem Test lediglich auf die grundlegende Funktionalität ankommt. Da das Löschen sehr einfach ist, ist das es in diesem Fall nicht unbedingt notwendig.

4.1.2 ConfigurationManagement

... toDo Leon

4.1.2.1 Beispiel: Einlesen einer JSON-Datei

```
1 TEST_CASE("Json Datei einlesen", "[]") {
2
3    REQUIRE_NOTHROW(Configurator::instance()->read_config(
4    "../test/ConfigurationManagement/config_test.json"));
5
6    REQUIRE(Configurator::instance()->get_config_as_bool("BOOLEAN") == true);
       REQUIRE(Configurator::instance()->get_config_as_unsigned_int(
7    "UNSIGNED_INT") == 42);
8    REQUIRE(Configurator::instance()->get_config_as_string("STRING") ==
9    "Hello World.");
10   REQUIRE(Configurator::instance()->get_config_as_float("FLOAT") == 1.337f);
11   REQUIRE(Configurator::instance()->get_config_as_double("DOUBLE") == -3.001);
12 }
```

Codeausschnitt 4.2: Unit-Test zum Einlesen einer JSON-Datei

4.1.2.2 Beispiel: Nicht exisitierende JSON-Datei

```
1 TEST_CASE("nicht existierende Json-Datei", "[]") {
2     REQUIRE_THROWS(Configurator::instance()->read_config("non-existent.json"));
3     REQUIRE_THROWS(Configurator::instance()->read_config("non-existent.json",
4     "typo.json"));
5 }
```

Codeausschnitt 4.3: Unit Test: Nicht existierende JSON-Datei

4.1.3 PacketDissection

Das Paket der `PacketDissection` wird in zwei verschiedenen Testdateien getestet: `PacketContainer_test.cpp` und `PacketInfo_test.cpp`.

Diese Aufteilung ermöglich einen gewissen Grad an Unabhängigkeit beim Testen. Außerdem findet man die Testfälle zur gewünschten Klasse auf diese Art wesentlich schneller.

4.1.3.1 Beispiele aus dem PacketContainer

```
1  TEST_CASE("PacketContainer", "[]") {
2      uint16_t inside_port = 0;
3      uint16_t outside_port = 1;
4      struct rte_mempool mbuf_pool_struct;
5      struct rte_mempool* mbuf_pool = &mbuf_pool_struct;
6      CHECK(mbuf_pool != nullptr);
7
8      NetworkPacketHandler pkt_handler(0,0);
9
10     PacketContainer pkt_container(pkt_handler, mbuf_pool, inside_port,
       outside_port);
11
12     ...
13
14 }
```

Codeausschnitt 4.4: Testfall PacketContainer

Innerhalb des Testfalls `PacketContainer` sind die folgend dargestellten Sektionen eingebettet (vgl. Codeausschnitt 4.4). Sektionen dienen im Allgemeinen zur weiteren Unterteilung bzw. Strukturierung der Testcases. Im Folgenden Abschnitt werden einige dieser Sektionen in `PacketContainer_test.cpp` näher erläutert.

Im obigen Testfall werden zuerst alle für die folgenden Unit-Tests benötigten Initialisierungen vorgenommen, wie zum Beispiel die des `rte_mempool` (siehe Z. 5) und des `NetworkPacketHandler` sowie des `PacketContainer` (siehe Z. 8 und 10).

```
1  SECTION("get_empty_packet", "[]") {
2
3      SECTION("default", "[]") {
4          CHECK(pkt_container->get_number_of_polled_packets() == 0);
5          CHECK(pkt_container->get_total_number_of_packets() == 0);
6
7          // by default it returns a TCP SYN packet
8          PacketInfo* pkt_info = pkt_container->get_empty_packet();
9          CHECK(pkt_info != nullptr);
10         CHECK(pkt_info->get_mbuf() != nullptr);
11         CHECK(pkt_info->get_type() == IPv4TCP);
12
13         CHECK(pkt_container->get_total_number_of_packets() >=
14         pkt_container->get_number_of_polled_packets());
15         CHECK(pkt_container->get_total_number_of_packets() == 1);
```

```
16          CHECK(pkt_container->get_number_of_polled_packets() == 0);
17      }
18
19      SECTION("IPv4TCP", "[]") {
20          CHECK(pkt_container->get_number_of_polled_packets() == 0);
21          CHECK(pkt_container->get_total_number_of_packets() == 0);
22
23          PacketInfo* pkt_info = pkt_container->get_empty_packet(IPv4TCP);
24          CHECK(pkt_info != nullptr);
25          CHECK(pkt_info->get_mbuf() != nullptr);
26          CHECK(pkt_info->get_type() == IPv4TCP);
27
28          CHECK(pkt_container->get_total_number_of_packets() >=pkt_container->
29          get_number_of_polled_packets());
30          CHECK(pkt_container->get_total_number_of_packets() == 1);
31          CHECK(pkt_container->get_number_of_polled_packets() == 0);
32      }
33  }
```

Codeausschnitt 4.5: Sektion `get_empty_packet` mit den zwei Untersektionen `default` und `IPv4TCP`

Im Codeausschnitt 4.5 zur Methode `get_empty_packet()` wird überprüft, ob die Getter-Methode zum Erhalten eines leeren Paketes wie gewünscht funktioniert. Zunächst wird dazu in Z. 4 f. sichergestellt, dass die bisherige Zahl der (gepollten) Pakete null ist. Anschließend wird für eine `PacketInfo`-Referenz die entsprechende Methode aufgerufen, wie in Z. 7 zu erkennen ist. In den anschließenden Code-Zeilen wird gecheckt, ob es keine Null-Pointer gibt und ob es sich um `IPv4TCP` handelt. Außerdem ist es wichtig, dass die Anzahl der Pakete insgesamt größer als die der abgefragten Pakete ist, genauer gesagt eins und null. Die entsprechenden Assertions sind in Z. 27 - 30 zu finden.

Die darauf folgende Section `IPv4TCP` unterscheidet sich nur insofern von der default-Variante, dass hier in Z. 22 beim Aufruf der `get_empty_packet()`-Methode zusätzlich `IPv4TCP` übergeben wird. Die Beschreibung der weiteren Bestandteile der Sektion findet man demzufolge im vorhergehenden Absatz.

Im Codeausschnitt 4.6 wird `get_empty_packet()` so oft aufgerufen, bis die `BURST_SIZE` erreicht ist. Direkt danach wird sichergestellt, dass die gesamte Anzahl an Paketen auch wirklich der `BURST_SIZE` entspricht.

```
1   SECTION("create more packets than burst size", "[]") {
2
3       SECTION("fill till BURST_SIZE", "[]") {
4           for (int i = 0; i < BURST_SIZE; ++i) {
5               PacketInfo* pkt_info = pkt_container->get_empty_packet();
6               CHECK(pkt_info != nullptr);
7           }
8
9           CHECK(pkt_container->get_total_number_of_packets() == BURST_SIZE);
10      }
11
```

```
12      SECTION("fill till BURST_SIZE + 1", "[]") {
13          for (int i = 0; i < BURST_SIZE + 1; ++i) {
14              PacketInfo* pkt_info = pkt_container->get_empty_packet();
15              CHECK(pkt_info != nullptr);
16          }
17
18          CHECK(pkt_container->get_total_number_of_packets() ==
19          BURST_SIZE + 1);
20      }
21
22      CHECK(pkt_container->get_total_number_of_packets() >=
23      pkt_container->get_number_of_polled_packets());
24  }
```

Codeausschnitt 4.6: Sektion „Create more packets than burst size" mit den zwei Untersektionen „fill till BURST_SIZE" und „fill till BURST_SIZE + 1"

In der anschließenden Sektion wird eine ähnliche Befüllung des `pkt_container` vorgenommen. Der Unterschied besteht darin, dass die Methode zum Erhalten der leeren Pakete einmal öfter aufgerufen wird.

Die Test-Section, die in Codeabschnitt 4.7 zu sehen ist, dient zum Test des Zugriffes auf einzelne Pakete an einem bestimmten Index. Dafür wurden zwei Unter-Sections geschrieben, eine für allgemeine Tests und eine, die sich auf Out-of-Bounds-Fehler bezieht.

In ersterer wird zunächst in Z. 5 ein leeres Paket hinzugefügt und wieder die Richtigkeit der Anzahl der Pakete getestet. Darauf hin kommt es für `pkt_info_1` zum Aufruf der `get_packet_at_index()`-Methode. Dabei wird der mit Hilfe von `get_total_number_of_packets()` ermittelte Index des zu Beginn erstellten leeren Pakets übergeben. Anschließend wird wieder einmal gecheckt, ob es sich auch wirklich um ein valide Paket handelt und die Anzahl aller Pakete und der abgerufenen Pakete korrekt ist. In Z. 18 f. wird zunächst getestet, ob es bei der bereits in Z. 9 aufgerufenen Methode auch wirklich zu keinen Fehlern kommt. Bei der Übergabe eines Indexes, unter dem kein Paket zu finden ist, muss es allerdings zum Wurf einer Exception kommen, was in Z. 20 kontrolliert wird.

```
1   SECTION("get_packet_at_index", "[]") {
2
3       SECTION("general", "[]") {
4           // add empty packet
5           PacketInfo* pkt_info_0 = pkt_container->get_empty_packet();
6           CHECK(pkt_container->get_number_of_polled_packets() == 0);
7           CHECK(pkt_container->get_total_number_of_packets() == 1);
8
9           PacketInfo* pkt_info_1 = pkt_container->get_packet_at_index(
10          pkt_container->get_total_number_of_packets() - 1);
11          CHECK(pkt_info_0 == pkt_info_1);
12          CHECK(pkt_info_1 != nullptr);
13          CHECK(pkt_info_1->get_mbuf() != nullptr);
14          CHECK(pkt_info_1->get_type() == IPv4TCP);
15
16          CHECK(pkt_container->get_total_number_of_packets() >=
```

```
17          pkt_container->get_number_of_polled_packets());
18          CHECK_NOTHROW(pkt_container->get_packet_at_index(
19          pkt_container->get_total_number_of_packets() - 1));
20          CHECK_THROWS(pkt_container->get_packet_at_index(
21          pkt_container->get_total_number_of_packets()));
22      }
23
24      SECTION("test out of bounds error", "[]") {
25          for (int i = 0; i < int(BURST_SIZE / 2); ++i) {
26              pkt_container->get_empty_packet();
27          }
28
29          CHECK(pkt_container->get_total_number_of_packets() ==
30          int(BURST_SIZE / 2));
31
32          for (int i = 0; i < int(BURST_SIZE / 2); ++i) {
33              CHECK_NOTHROW(pkt_container->get_packet_at_index(i));
34          }
35
36          for (int i = int(BURST_SIZE / 2); i < BURST_SIZE; ++i) {
37              CHECK_THROWS(pkt_container->get_packet_at_index(i));
38          }
39
40          CHECK_THROWS(pkt_container->get_packet_at_index(
41          pkt_container->get_total_number_of_packets()));
42          CHECK_NOTHROW(pkt_container->get_packet_at_index(
43          pkt_container->get_total_number_of_packets() - 1));
44      }
45  }
```

Codeausschnitt 4.7: Sektion „get_packet_at_index” mit den zwei Untersektionen „general” und „test out of bounds error”

In der zweiten Section des Codeabschnitts 4.7 wird wieder ähnlich wie oben vorgegangen. Zu Beginn wird für jedes `i` von 0 bis `BURST_SIZE / 2` ein leeres Paket abgerufen und geprüft, ob die Anzahl der Pakete stimmt. Somit darf es auch beim Aufruf von `get_packet_at_index` für die erste Hälfte des Intervalls von null bis `BURST_SIZE` zu keinem Fehler kommen, was in Z. 32 ff. getestet wird. In der darauf folgenden for-Schleife muss es hingegen zu einer Exception kommen. Die letzten beiden Checks in Z. 40 - 43 sind äquivalent zu denen in Z. 18 - 21.

```
1   SECTION("take_packet and add_packet", "[]") {
2
3       PacketInfo* pkt_info_0 = pkt_container->get_empty_packet();
4       CHECK(pkt_container->get_number_of_polled_packets() == 0);
5       CHECK(pkt_container->get_total_number_of_packets() == 1);
6
7       PacketInfo* pkt_info_1 = pkt_container->take_packet(0);
8       CHECK(pkt_info_0 == pkt_info_1);
9       CHECK(pkt_info_1 != nullptr);
10      CHECK(pkt_info_1->get_mbuf() != nullptr);
```

```
11      CHECK(pkt_container->get_packet_at_index(0) == nullptr);
12      CHECK(pkt_container->get_total_number_of_packets() == 1);
13      CHECK(pkt_container->get_total_number_of_packets() >=
14      pkt_container->get_number_of_polled_packets());
15
16      pkt_container->add_packet(pkt_info_1);
17      CHECK(pkt_container->get_total_number_of_packets() >=
18      pkt_container->get_number_of_polled_packets());
19      CHECK(pkt_container->get_number_of_polled_packets() == 0);
20      CHECK(pkt_container->get_total_number_of_packets() == 2);
21      CHECK(pkt_container->get_packet_at_index(1) == pkt_info_1);
22      CHECK(pkt_container->get_packet_at_index(0) == nullptr);
23
24  }
```

Codeausschnitt 4.8: Sektion „take_packet and add_packet"

In Codeabschnitt 4.8 wird getestet, ob das Herausnehmen und das Hinzufügen von Paketen funktioniert. Dazu wird zu Beginn wieder ein leeres Paket mittels `get_empty_packet()` erstellt. In den Zeilen 7 - 14 wird zunächst das leere Paket durch die Methode `take_paket()` dem `pkt_container` entnommen. Dieses wird damit auch aus dem Container entfernt. Danach kommt es zu verschiedenen Checks, wie zum Beispiel, ob das entnommene Paket das selbe ist, welches hinzugefügt wird. In Z. 16 ff. wird dann die Methode `add_packet()` getestet. Dafür wird `pkt_info_1` hinzugefügt und die Anzahl der Pakete überprüft. So muss die Anzahl aller Pakete nun zwei betragen. Die Methode `get_packet_at_index()` muss bei Übergabe des Wertes eins `pkt_info_1` zurückgeben, bei null muss `nullptr` zurückgegeben werden.

Die Methode `drop_packet()` wird durch die im Codeabschnitt 22 dargestellte Sektion getestet. Auch hierfür werden in den Zeilen 4 - 7 die gleichen Initialisierungen wie in der vorherigen Section vorgenommen. Nach dem einmaligen Aufruf der Methode `drop_packet()` werden die üblichen Checks durchgeführt. Besonders interessant ist Zeile 13, in der getestet wird, ob die Methode `get_packet_at_index()` für den Index 0 einen Null-Pointer zurückgibt. In Zeile 15 wird sicher gestellt, dass es beim erneuten Aufruf von `drop_packet()` nicht zu einer Exception kommt. Die letzten drei Zeilen können mit Z. 11 - 13 verglichen werden.

```
1  SECTION("drop_packet", "[]") {
2
3      SECTION("default") {
4          PacketInfo* pkt_info_0 = pkt_container->get_empty_packet();
5          CHECK(pkt_container->get_number_of_polled_packets() == 0);
6          CHECK(pkt_container->get_total_number_of_packets() == 1);
7
8          pkt_container->drop_packet(0);
9          CHECK(pkt_container->get_total_number_of_packets() >=
10         pkt_container->get_number_of_polled_packets());
11         CHECK(pkt_container->get_number_of_polled_packets() == 0);
12         CHECK(pkt_container->get_total_number_of_packets() == 1);
13         CHECK(pkt_container->get_packet_at_index(0) == nullptr);
14
15         CHECK_NOTHROW(pkt_container->drop_packet(0));
```

```
16          CHECK(pkt_container->get_number_of_polled_packets() == 0);
17          CHECK(pkt_container->get_total_number_of_packets() == 1);
18          CHECK(pkt_container->get_packet_at_index(0) == nullptr);
19      }
20
21  }
```

Codeausschnitt 4.9: Sektion „drop_packet"

Der letzte beispielhafte Codeabschnitt für den `PacketContainer` ist der für den Test von `poll_packets`. Auch hierfür wird zu Beginn überprüft, ob die Anzahl der Pakete null ist. In Zeile 6 wird der vorzeichenlose 16-bit-Ganzzahlwert `nb_pkts_polled` definiert. Zum Aufruf der Methode `poll_packets` kommt es in Z. 7. Anschließend wird erneut geprüft, ob die Anzahl der Pakete richtig ist. Die Zeilen 13 - 17 sind für den Test von `get_packet_at_index()` mit der Übergabe des Wertes `nb_pkts_polled - 1` wichtig. So darf es auch hier nicht zum Wurf einer Exception kommen und sowohl `pkt_info` als auch `pkt_info->get_mbuf()` dürfen kein Null-Pointer sein. Damit soll getestet werden, ob die Variablen, die von den entsprechenden Getter-Methoden zurückgegeben werden, richtig berechnet worden sind.

```
1  SECTION("poll_packets", "[]") {
2
3      CHECK(pkt_container->get_number_of_polled_packets() == 0);
4      CHECK(pkt_container->get_total_number_of_packets() == 0);
5
6      uint16_t nb_pkts_polled;
7      pkt_container->poll_packets(nb_pkts_polled);
8      CHECK(pkt_container->get_number_of_polled_packets() > 0);
9      CHECK(pkt_container->get_total_number_of_packets() ==
10     pkt_container->get_number_of_polled_packets());
11     CHECK(nb_pkts_polled == pkt_container->get_number_of_polled_packets());
12
13     CHECK_NOTHROW(pkt_container->get_packet_at_index(nb_pkts_polled - 1));
14     PacketInfo* pkt_info =
15     pkt_container->get_packet_at_index(nb_pkts_polled - 1);
16     CHECK(pkt_info != nullptr);
17     CHECK(pkt_info->get_mbuf() != nullptr);
18
19 }
```

Codeausschnitt 4.10: Sektion „poll_packets"

Auf die Sektion „poll_packets" würde noch eine weitere Sektion mit neun Untersektionen folgen, die das Senden von Paketen testet. Diese Sektion wird hier jedoch nicht näher beschrieben.

4.1.3.2 Beispiel aus der PacketInfo

... toDo für Tobias

4.1.4 Inspection

Die Unit-Tests der `Inspection` können in drei Teile gegliedert werden. Der erste Teil muss die korrekte Instanziierung überprüfen, der zweite Teil testet die korrekte Identifizierung der einzelnen Angriffe und der dritte und letzte Teil testet die korrekte Erstellung der Statistik und Auswertung der Paketdaten.

Die korrekte Instanziierung testet, ob aus der Konfiguration korrekte Werte geladen werden, die auch für das System verwendbar sind. Zu diesen Werten zählen beispielsweise nur positive Zahlen. Nicht nur aus den Konfigurationswerten werden während der Instanziierung die Startwerte für die `Inspection` gebildet. Die Werte der Konfiguration werden aus dem Dummy `Inspection_config.json` übergeben und in der Konfiguration eingelesen.

Im zweiten Teil wird jede Form des Angriffs einzeln auf korrekte Erkennung überprüft. Hierzu werden für SYN-Fin, SYN-FIN-ACK, Zero-Window und Small-Window Angriffe jeweils ein `PacketContainer` und in diesem `PacketContainer` wird ein Paket mit den jeweiligen Anforderungen (bestimmte Flags gesetzt oder kleine Window Größe) erstellt. Um eine falsch-richtige und richtig-falsche Erkennung ausschließen zu können, werden daneben auch `PacketContainer` mit korrekten Paketen ohne diese Anforderungen an die Erkennung erstellt und getestet. Nach jedem Durchlauf eines `PacketContainers` durch die `Inspection` muss dieser Container leer sein für richtige Angriffserkennung und nicht leer für eine richtige Nicht-Angriff-Erkennung.

```
1  SECTION("SYN-FIN Attack", "[]") {
2      PacketInfoIpv4Tcp* pkt_info = pkt_container->get_empty_packet(IPv4TCP);
3      // create packet with SYN-FIN Flag into packet container
4      // all values are obsolete except the flags
5      pkt_info->fill_payloadless_tcp_packet("00:00:00:00:00:00","
       00:00:00:00:00:00",0,0,0,0,0,0,0b00000011,0);
6      // packet container to inspection
7      testInspection.analyze_container(pkt_container);
8      // SYN-FIN Flag should be detected and packet removed
9      // Check if packetContainer is empty
10     CHECK(pkt_container->get_total_number_of_packets() == 0);
11 }
```

Codeausschnitt 4.11: Test von SYN-FIN Angriffen in `Inspection_test.cpp`

Für das korrekte Erkennen von UDP-/TCP-/ICMP-Flood Attacken werden jeweils mehrere `PacketContainer` erstellt, die mit mehr Paketen gefüllt werden als der Threshold in der Instanziierung vorgibt. Werden bei dem jeweils zweiten `PacketContainer` weniger Pakete weitergesendet, ist die Erkennung korrekt erfolgt. Um auch bei diesen Tests eine richtig-falsch und falsch-richtige Erkennung ausschließen zu können, werden daneben noch `PacketContainer` mit weniger Paketen, als der Threshold zulässt, erstellt und der `Inspection` übergeben. Diese `PacketContainer` müssen die Zahl ihrer Pakete beibehalten.

Die lokale Statistik kann einfach überprüft werden, indem der `update_stats()`-Methode feste Werte nach der Instanziierung übergeben werden und die berechneten Werte genau den erwarteten Werten entsprechen müssen.

4.1.5 Treatment

Die Lines of Code der Unit-Test-Datei der Klasse `Treatment` belaufen sich auf weit über 1000. Diese vergleichsweise hohe Menge ist unter anderem darauf zurückzuführen, dass in der Datei `Treatment_test.cpp` auch die verwendete Hash-Funktion (XXH3) und die verwendete Map (`dense_hash_map`) auf Tauglichkeit für die spätere Verwendung in der Klasse getestet wurde. Desweiteren wurde ein Benchmark erstellt und durchgeführt, welcher die Performance einer `std::unordered_map` mit der einer `dense_hash_map` vergleicht.

Durch die Einführung der Friend-Klasse `Treatment_friend` ist es möglich, auf die in der Klasse `Treatment` auf `private` gesetzten Attribute und Methoden mit Hilfe von Get-Methoden zuzugreifen. Der Wert des Attributs `_s_timestamp` kann außerdem durch eine Set-Methode neu zugewiesen werden.

Entgegen dem in der zweiten Phase vorgestellten Entwurf, ist aus Gründen der Kapselung die Methode `create_cookie_secret()` nicht mehr Teil der Klasse `Treatment`. An dessen Stelle tritt nun die Methode `get_random_64bit_value()` aus der Klasse `Rand`, welche die gleiche Funktionalität wie `create_cookie_secret()` aufweist. Um dennoch die bereits zuvor erstellten Unit-Tests ausführen zu können, welche die Methode `create_cookie_secret()` verwenden, gibt es in der Klasse `Treatment_friend` eine Methode mit dem ebendiesem Namen, welche allerdings die Methode `get_random_64bit_value()` der Klasse `Rand` aufruft.

`Treatment_friend` ist in der Datei `Treatment_test.cpp` definiert (vgl. Codeausschnitt 4.12).

```
1  class Treatment_friend{
2
3      Treatment* treatment = new Treatment();
4
5      public:
6
7          static void s_increment_timestamp(){
8            return Treatment::s_increment_timestamp();
9          }
10
11         void treat_packets_to_inside(){
12             treatment-> treat_packets_to_inside();
13         }
14
15         void treat_packets_to_outside(){
16             treatment-> treat_packets_to_outside();
17         }
18
19         u_int32_t calc_cookie_hash(u_int8_t _s_timestamp, u_int32_t _extip,
            u_int32_t _intip, u_int16_t _extport, u_int16_t _intport){
20             return treatment->calc_cookie_hash(_s_timestamp, _extip, _intip,
               _extport, _intport);
21         }
22
23         bool check_syn_cookie(u_int32_t cookie_value, const Data &d){
24             return treatment->check_syn_cookie(cookie_value, d);
25         }
26
```

```
27          u_int64_t create_cookie_secret(){
28              return Rand::get_random_64bit_value();
29          }
30
31          //Getter
32          u_int8_t get_s_timestamp(){
33              return treatment->_s_timestamp;
34          }
35
36          u_int64_t get_cookie_secret(){
37              return treatment->_cookie_secret;
38          }
39
40          PacketContainer* get_packet_to_inside(){
41              return treatment->_packet_to_inside;
42          }
43
44          PacketContainer* get_packet_to_outside(){
45              return treatment->_packet_to_outside;
46          }
47
48          google::dense_hash_map<Data, Info, MyHashFunction> get_densemap(){
49              return treatment->_densemap;
50          }
51
52          //Setter
53          void set_s_timestamp(u_int8_t value){
54              treatment->_s_timestamp = value;
55          }
56
57  };
```

Codeausschnitt 4.12: Friend-Klasse `Treatment_friend` in der Datei `Treatment_test.cpp`

Zuvor wurde in der Headerdatei der Klasse `Treatment` angegeben, dass sie mit `Treatment_friend` befreundet ist (vgl. Codeausschnitt 4.13).

```
1  ...
2  class Treatment{
3      ...
4      friend class Treatment_friend;
5  };
```

Codeausschnitt 4.13: Deklaration der friend_klasse `Treatment_friend` in der Datei `Treatment.h`

In den folgenden Sektionen werden beispielhaft einzelne, ausgewählte Testfälle erläutert:

4.1.5.1 Beispiel: `check_syn_cookie()`

```
bool Treatment::check_syn:cookie(u_int32_t cookie_value, const Data\& d){
    // Extract the first 8 bit of the cookie (= timestamp)
    u_int8_t cookie_timestamp = cookie_value \& 0x000000FF;

    u_int8_t diff = _s_timestamp - cookie_timestamp;

    if(diff<1){
        // Calculate hash
        u_int32_t hash;

        // Case: same time interval
        if(diff == 0){
            // calculate expected cookie-hash
            hash = calc_cookie_hash(_s_timestamp, d._extip, d._intip, d.extport
            , d._intport);
            hash = hash \& 0xFFFFFF00;
            // stuff cookie-hash with 8 bit _s_timestamp
            hash |= (u_int8_t) _s_timestamp;
        }
        if(diff == 1){
            // calculate expected cookie-hash
            hash = calc_cookie_hash((_s_timestamp-1), d._extip, d._intip, d.
    extport, d._intport);
            hash = hash \& 0xFFFFFF00;
            // stuff cookie-hash with 8 bit _s_timestamp
            hash |= (u_int8_t) (_s_timestamp-1);
        }
         // test whether the cookie is as expected; if so, return true
         if(hash == cookie_value){
            return true;
        }
    }

    //return false, so that treat_packets is able to continue
    return false;
 }
```

Codeausschnitt 4.14: Methode: `check_syn_cookie()` in `Treatment.cpp`

Die Methode `check_syn_cookie()` (vgl. Codeausschnitt 4.14) überprüft, ob der empfangene SYN-Cookie im richtigen Zeitintervall am System angekommen ist. Jenes ist zutreffend, sofern der Timestamp des empfangenen Cookie nicht mehr als eine Zeiteinheit von dem aktuellen Timestamp-Wert `_s_timestamp` abweicht. Falls dem so ist, wird überprüft, ob der empfangene Cookie dem erwarteten Cookie entspricht. Die Methode gibt `true` zurück, falls dies der Fall ist so ist. Sollte dem nicht so sein, so ist der Rückgabewert `false`.

```
TEST_CASE("check_syn_cookie", "[]"){
    ....
    SECTION("check_syn_cookie(): diff==1 with random numbers (without using the
```

```
    PacketDissection)", "[]"){
4       //Create a Treatment object
5       Treatment_friend treat;
6
7       //Generate a random 8-bit-number
8       u_int8_t ran_num = (u_int8_t) rand();
9
10      //increment _s_timestamp up to ran_num
11      for(int i=0; i<ran_num; i++){
12          treat.s_increment_timestamp();
13      }
14
15     CHECK(treat.timestamp==ran_num);
16
17      u_int32_t extip = rand();
18      u_int32_t intip = rand();
19      u_int16_t extport = rand();
20      u_int16_t intport = rand();
21
22      //Create cookie_value with timestamp ran_num - 1
23      u_int32_t cookie_value = treat.calc_cookie_hash((ran_num - 1), extip,
        intip, extport, intport);
24      cookie_value = cookie_value  & 0xFFFFFF00;
25      cookie_value |=(u_int8_t) (ran_num-1);
26
27     //Create a Data object
28      Data d;
29      d._extip = extip;
30      d._intip = intip;
31      d._extport = extport;
32      d._intport = intport;
33
34      CHECK(treat.check_syn_cookie(cookie_value, d));
35  }
36 ....
37 }
```

Codeausschnitt 4.15: Unit-Test für die Methode `check_cookie_secret()` in `Treatment_test.cpp`

Im Unit-Test in Codeausschnitt 4.15 wird untersucht, ob die Methode `check_syn_cookie()` als Rückgabewert `true` hat (siehe Zeile 34). Zudem wird in Zeile 15 überprüft, ob der Timestamp korrekt inkrementiert wurde.

Zuerst wird in Zeile 5 das Objekt `treat` der Klasse `Treatment_friend` erzeugt. In Zeile 8 wird anschließend eine 8-Bit lange Zufallszahl generiert, die in der Variable `ran_num` gespeichert wird. In einer for-Schleife wird dann der Timestamp des Objektes `treat` um exakt diesen zufälligen Wert erhöht. Auch die vier 32-Bit langen Variablen `extip`, `intip`, `extport` und `intport` bekommen eine zufällige Zahl zugewiesen. Mit Hilfe der Methode `calc_cookie_hash()` wird in Zeile 23 der Cookie-Wert erzeugt. Hier ist zu beachten, dass der erste Parameter dieser Methode

`ran_num-1` ist. Somit ist der Timestamp dieses Cookie-Wertes um genau 1 kleiner als derjenige, der in `treat` gespeichert ist. Das heißt, dass `diff` in Zeile 5 in dem Codeausschnitt 4.14 genau 1 ist. Somit sollte die Methode `true` zurückgeben.Zeile 24 sorgt dafür, dass der zuvor berechnete `cookie_value` auf die ersten 24 Bit gekürzt wird. An die Stelle der letzten acht Bit treten fortan der Wert des Timestamps `ran_num-1`. In den Zeilen 28 bis 32 wird ein Datenobjekt `d` im Stack angelegt und mit Werten gefüllt. Der zuvor generierte Cookie-Wert und das Datenobjekt werden anschließend in die zu überprüfende Methode `check_syn_cookie()` übergeben.

Das Verhalten der Methode `check_syn_cookie()` wird noch in acht weiteren Sektionen getestet. Hier werden unter anderem die Fälle durchlaufen, dass die Differenz des aktuellen Zeitstempels und des in `cookie_value` übergebenen Zeitstempels null, größer eins oder kleiner null ist. Ebenfalls wichtig sind hier die Grenztests, welche auch das Verhalten der Methode bei Überlaufen der `u_int8_t`-Werte testen. Außerdem gibt es einen Test, indem die IP-Adressen und Port-Nummern des Cookies und des Datenobjektes nicht übereinstimmen.

4.1.5.2 Beispiel: `s_increment_timestamp()`

```
1 void Treatment::s_increment_timestamp(){
2     // increment _s_timestamp by one
3     ++_s_timestamp;
4 }
```

Codeausschnitt 4.16: Methode: `s_increment_timestamp()`

Die Methode `s_increment_timestamp()`, die im Codeausschnitt 4.16 dargestellt ist, macht vergleichsweise wenig: Sie erhöht den Wert der Membervariable `_s_timestamp` um 1.

```
1  TEST_CASE("s_increment_timestamp()", "[]"){
2      ...
3      SECTION("Increment _s_timestamp up to 1000 (>255>size if u_int8_t)", "[]"){
4          Treatment_friend treat;
5
6          u_int8_t count = 0;
7
8          for(int i=0; i<1000; i++){
9              CHECK(treat._s_timestamp == count);
10             treat.s_increment_timestamp();
11             count++;
12         }
13     }
14     ...
15 }
```

Trotz des vergleichbar kleinen Funktionsumfangs muss getestet werden, wie sich die Variable `_s_timestamp` verhält, wenn sie öfter als 255 mal inkrementiert wurde. Denn der Datentyp der Membervariable `_s_timestamp` ist `u_int8_t` , welcher allerdings lediglich 8 Bit umfasst und somit einen Wertebereich von 0 bis 255 hat. Aus diesem Grund ist das erwartete Verhalten, dass es beim 256. Inkrementieren zum arithmetischen Überlauf kommt. Ab hier beginnt `_s_timestamp` wieder bei 0.

4.1.5.3 Beispiel: Benchmark

Um herauszufinden, welche Map sich am besten für das Softwareprojekt eignet, wurde ein Benchmark erstellt. Dieser vergleicht die Performance einer Unordered-Map mit der einer Dense-Map.

```
TEST_CASE("Benchmark", "[]"){
    typedef std::unordered_map<Data, Info, MyHashFunction> unordered;
    unordered unord;
    google::dense_hash_map<Data, Info, HashMyFunction> densemap;
    clock_t tu;
    clock_t tr;
    clock_t td;
    densemap.set_empty_key(Data(0,0,0,0));

    Info flix;
    flix._offset = 123;
    flix._finseen = 0;

    //-------------------------------------------------------

    //-------------------------------------------------------

    long runs = 15;
    clock_t uclock [runs] = {};
    clock_t dclock [runs] = {};
    long runner = 600000;
    Data arr [runner] = {};

    for(long r = 0; r < runs; ++r) {


        for(long i = 0; i < runner; ++i){
            arr[i]._extip = rand();
            arr[i]._intip = rand();
            arr[i]._extport = rand();
            arr[i]._intport = rand();
        }

        auto startu = std::chrono::high_resolution_clock::now();
        tu = clock();
        for(long i = 0; i < runner; ++i){
            unord.emplace(arr[i], flix);
            iu->first.extip << std::endl;
        }
        for(long i = 0; i < runner; ++i){
            unord.find(arr[i-1 % runner]);
            unord.find(arr[i]);
            unord.find(arr[i+1 % runner]);
            unord.find(arr[i+50 % runner]);
            iu->first.extip << std::endl;
```

```
46          }
47          tu = clock() - tu;
48          auto finishu = std::chrono::high_resolution_clock::now();
49
50
51          auto startd = std::chrono::high_resolution_clock::now();
52          td = clock() ;
53          for(long i = 0; i < runner; ++i) {
54              densemap.insert(std::pair<Data, Info>(arr[i], flix));
55          }
56          for(long i = 0; i < runner; ++i){
57              densemap.find(arr[i-1 % runner]);
58              densemap.find(arr[i]);
59              densemap.find(arr[i+1 % runner]);
60              densemap.find(arr[i+50 % runner]);
61              id->first.extip << std::endl;
62          }
63          td = clock() - td;
64          auto finishd = std::chrono::high_resolution_clock::now();
65
66          std::chrono::duration<double> elapsedu = finishu - startu;
67          std::chrono::duration<double> elapsedd = finishd - startd;
68          dclock[r] = td;
69          uclock[r] = tu;
70          BOOST_LOG_TRIVIAL(info) << "Elapsed time of unordered: " << elapsedu.
            count();
71          BOOST_LOG_TRIVIAL(info) << "Elapsed time of dense: " << elapsedd.
            count();
72
73      }
74      int sumd = 0;
75      int sumu = 0;
76      for (long x = 0; x < runs; ++x) {
77          sumd =  sumd + dclock[x];
78          sumu =  sumu + uclock[x];
79      }
80      BOOST_LOG_TRIVIAL(info) << "This is the average clock count of densemap of
        " << runs << " rounds, of each " << runner << " elements inserted, and " <<
         4*runner  << " elements searched : " << sumd/runs;
81      BOOST_LOG_TRIVIAL(info) << "This is the average clock count of
        unordered_map of " << runs << " rounds, of each " << runner << " elements
        inserted, and " << 4*runner  << " elements searched : " << sumu/runs;
82  }
```

Codeausschnitt 4.17: Benchmark zum Vergleich der Performance einer Unordered-Map und einer Dense-Map

Das Ergebnis des Benchmarks in Abb. 4.17 zeigt, dass die Densemap in jedem Durchlauf für die exakt selben Operationen durchschnittlich nur halb so viele CPU-Ticks braucht, wenn die

`std::unordered_map` als Vergleich verwendet wird. Eine Beispielhafte Ausgabe ist in Codeausschnitt 4.18 zu erkennen.

```
This is the average clock count of densemap of 10 rounds, of each 600000
 elements inserted, and 2400000 elements searched : 224847
This is the average clock count of unordered\_map of 10 rounds, of each
 600000 elements inserted, and 2400000 elements searched : 614058
```

Codeausschnitt 4.18: Beispielhaftes Ergebnis des Benchmarks zum Vergleich der Performance einer Unordered-Map und einer Dense-Map

Ebenfalls getestet wurde die Patchmap von 1ykos. Hierbei stellte sich allerdings heraus, dass die Performanz nicht den Erwartungen genügte.

4.1.5.4 Beispiel: Densemap

Um die Verhaltensweise der Densemap vor deren Nutzung im Code besser kennenzulernen, wurden mehrere Tests geschrieben, welche die Grundfunktionalitäten der Map wie zum Beispiel das Löschen oder Hinzufügen von Werten testen.

```
TEST_CASE("Map", "[]"){
    ...
    SECTION("Densemap: Erase one element whose key is known", "[]"){
        google::dense_hash_map<Data, Info, MyHashFunction> densemap;

        Data empty;
        empty._extip = 0;
        empty._intip = 0;
        empty._extport = 0;
        empty._intport = 0;
        densemap.set_empty_key(empty);

        Data deleted;
        deleted._extip = 0;
        deleted._intip = 0;
        deleted._extport = 1;
        deleted._intport = 1;
        densemap.set_deleted_key(deleted);

        Data d1;
        d1._extip = 12345;
        d1._intip = 12334;
        d1._extport = 123;
        d1._intport = 1234;

        Info i1;
        i1._offset = 3;
        i1._finseen = false;
        // calculates index over d1, store d1 as first an i1 as second
        densemap[d1] = i1;

```

```
32          Data d2;
33          d2._extip = 12345;
34          d2._intip = 12334;
35          d2._extport = 123;
36          d2._intport = 1234;
37
38          Info i2;
39          i2._offset = 3;
40          i2._finseen = false;
41          densemap[d2] = i2;
42
43          CHECK(densemap.size() == 2);
44          densemap.erase(d1);
45          CHECK(densemap.size() == 1);
46          densemap.erase(d2);
47          CHECK(densemap.size() == 0);
48      }
49  ...
50  }
```

Codeausschnitt 4.19: Unit-Tests zum Löschen von Elementen in der Densemap

Der obige Unit-Test (vgl. Abb. 4.19) verdeutlicht, dass direkt nach Anlegen der Densemap die Methode `set_empty_key()` aufgerufen werden muss. Ohne diese Methodenaufruf ist auch nicht der Aufruf weiterer `dense_hash_map`-Methoden möglich. Deshalb wird in den Zeilen 7 bis 12 ein solcher empty-Key angelegt und als Arugment der Methode `set_empty_key()` übergeben. Dieses Argument darf kein Schlüsselwert sein und wird niemals für legitime Einträge in der Map genutzt, der Wert Data(0,0,0,0) ist hierfür besonders gut geeignet, da legitime Verbindungen nie beide IPs auf den Wert null gesetzt haben.

Zudem wird zum Löschen von Einträgen in der Densemap mit der Methode `erase()` das Aufrufen der Methode `set_deleted_key()` benötigt. Dieser deleted-Key muss sich vom empty-Key unterscheiden, darf allerdings auch kein legitimer Schlüssel sein. Da in diesem Unit-Test das Löschen von Elementen getestet werden soll, wird ein Datenobjekt mit dem Namen `deleted` angelegt, befüllt und der Methode `set_deleted_key()` übergeben.

Danach wird die Map mit zwei weiteren Einträgen befüllt. Somit muss die Densemap nun die Größe von zwei haben. Nach dem Löschen von `d1` wird überprüft, ob die Größe der Densemap sich nun auf eins verringert hat. Nachdem der zweite Eintrag gelöscht wurde, wird in Zeile 47 nochmals auf das Übereinstimmen der Densemap-Größe mit dem Wert null getestet.

4.1.6 RandomNumberGenerator

In den folgenden Teilkapiteln geht es zunächst um den Zweck und die Funktionsweise des `RandomNumberGenerators`, worauf beispielhaft einige Unit-Tests vorgestellt werden.

4.1.6.1 Grundlegende Erläuterungen

Der `RandomNumberGenerator` (RNG) ist ein Pseudozufallszahlengenerator, der auf dem Xorshift-Algorithmus basiert. Dieser hat das Ziel, auf effiziente Weise möglichst zufällig verteilte Ganzzahlen zu generieren, welche vom Angreifer als Portnummern und IP-Adressen für von ihm

ausgehende Pakete verwendet werden können. Der entwickelte RNG enthält jeweils eine Methode zur Berechnung von 16-bit, 32-bit und 64-bit-Zahlen, weshalb die Typen `uint16_t`, `uin32_t` und `uint64_t` verwendet werden. Außerdem enthält er eine Methode, die einen pseudozufälligen `uint16_t`-Wert in einem bestimmten Intervall zurückgibt. Das ist nötig, weil wir die registrierten, aber nicht standardisierten Ports 1024 - 49151 verwenden.

Im Header `RandomNumberGenerator.h` werden mittels Member Initializer Lists die drei Seeds mit durch `rand()` generierten zufälligen Werten initialisiert. Wie sich im Code im Codeausschnitt **??** erkennen lässt, wird dieser Wert daraufhin in den Methoden durch Xor- und Shift-Operationen so verändert, dass die Ergebnisse pseudozufällig sind, also scheinbar zufällig, aber berechenbar. Für weiterführende Erläuterungen und Informationen empfiehlt sich eine Ausarbeitung von George Marsaglia [8].

4.1.6.2 Einfache Tests

Durch die im Codeausschnitt 4.20 dargestellten Tests soll geprüft werden, ob der Algorithmus bei gleichem Seed auch die gleiche Zahl generiert. Dies ist eine typische Eigenschaft von Pseudozufallszahlengeneratoren.

```
1  TEST_CASE("random_number_generator_basic", "[]") {
2      ...
3      SECTION("Check whether the same numbers are generated with the same seed "
4      "for 16 bit","[]") {
5          RandomNumberGenerator xor_shift_1;
6          RandomNumberGenerator xor_shift_2;
7          // set the seed to the same value in both RNGs
8          xor_shift_1._seed_x16 = 30000;
9          xor_shift_2._seed_x16 = 30000;
10         u_int16_t test_1_16_bit = xor_shift_1.gen_rdm_16_bit();
11         u_int16_t test_2_16_bit = xor_shift_2.gen_rdm_16_bit();
12         // check whether the results are the same too
13         CHECK(test_1_16_bit == test_2_16_bit);
14         std::cout << std::endl;
15     }
16
17     ...
18 }
```

Codeausschnitt 4.20: Test der Gleichheit der generierten Zahlen bei zwei RNGs mit gleichem Seed

Wie oben wurde dieser Test sowohl für die 16-bit-Methode als auch für die 32- und 64-bit-Methode geschrieben. Zunächst werden, wie in Zeile 5 f. zu sehen, zwei Objekte der Klasse `RandomNumberGenerator` erzeugt. Anschließend wird der mit `rand()` erzeugte Seed verändert und bei beiden RNGs auf den gleichen Wert gesetzt. In Zeile 10 und 11 wird dann für jedes der beiden RNG-Objekte die Methode zum Generieren einer 16-bit-Zahl aufgerufen. Nun kann in Z. 13 sichergestellt werden, dass die Zahlen `test_1_16_bit` und `test_2_16_bit` auch wirklich gleich sind.

4.1.6.3 Test der Verteilung der Zufallszahlen

Eine wichtige Eigenschaft des entwickelten `RandomNumberGenerator` muss eine gute Verteilung sein. Das heißt, dass z. B. nicht jede zehnte Zahl zehn mal so oft wie die theoretische Häufigkeit generiert werden darf und alle anderen Zahlen nie vom RNG ausgegeben werden. Dabei wird angenommen, dass die Wahrscheinlichkeit für eine Zahl, dass genau sie generiert wird, im Idealfall für alle Zahlen gleich hoch sein sollte. Schon vor dem Testen steht fest, dass der Algorithmus keine echt zufällige Zahlen generieren kann und dieses Ziel nicht erfüllt, allerdings kann das auch nie der Anspruch an einen Pseudozufallszahlengenerator sein. Um festzustellen, wie sehr die tatsächlichen Häufigkeiten von den theoretischen abweichen, eignet sich ein Chi-Quadrat-Test, welcher in dem Codeausschnitt 4.1 zu sehen ist.

```
1  TEST_CASE("RandomNumberGeneratorStatistics", "[]") {
2      SECTION("ChiSquare16", "[]") {
3          RandomNumberGenerator xor_shift;
4          // 65536 = 2 ^ 16 different numbers can be generated
5          int r = 65536 - 1;
6          // 1,000,000 numbers are generated
7          int n = 1000000;
8          u_int16_t t;
9          // this array counts how often each number from 0 to r is returned as a
10         // result
11         int f[r] = {};
12         for (int i = 0; i < r; i++) {
13             f[i] = 0;
14         }
15         for (int i = 1; i < n; i++) {
16             t = xor_shift.gen_rdm_16_bit();
17             f[t]++;
18         }
19         double chisquare = 0.0;
20         for (int i = 0; i < r; i++) {
21             // chi square is calculated
22             chisquare = chisquare + ((f[i] - n / r) * (f[i] - n / r) / (n / r))
       ;
23         }
24         std::cout << "chi square is: " << chisquare << std::endl;
25         double k = sqrt(chisquare / (n + chisquare));
26         std::cout << "k is: " << k << std::endl;
27         CHECK(k < 1.0);
28     }
29 }
```

Codeausschnitt 4.21: Chi-Quadrat-Test

Es ist zu erkennen, dass zunächst eine obere Intervallgrenze `r` festgelegt wird. Der RNG generiert demzufolge Zahlen von 0 bis `r` (Z. 5). Ein Array von 0 bis `r` wird in Z. 11 - 14 initialisiert und vollständig mit dem Wert 0 belegt. Schließlich werden in Z. 15 - 18 `n` Zahlen `t` generiert und durch die Inkrementierung der Werte im Array `f[]` die Häufigkeiten gezählt. Anschließend wird

der Wert `chisquare` nach der folgenden Formel berechnet:

$$\chi^2 = \sum \frac{(\text{beobachtete Häufigkeit} - \text{theoretische Häufigkeit})^2}{\text{theoretische Häufigkeit}} \tag{4.1}$$

Im geschriebenen Test kann in Z. 22 erkannt werden, dass die beobachtete Häufigkeit mit `f[i]` und die tatsächliche Häufigkeit mit `n / r` zu vergleichen ist.

Ein Problem des Chi-Quadrats ist allerdings die Abhängigkeit von n. Da sich bei Verdopplung von der Häufigkeiten auch das errechnete Ergebnis verdoppeln würde, ist dieser Wert allein nicht aussagekräftig. Aus diesem Grund wird in Z. 25 noch der Kontingenzkoeffizient `k` nach der Formel 4.2 berechnet.

$$K = \sqrt{\frac{\chi^2}{n + \chi^2}} \tag{4.2}$$

Das hierbei errechnete Ergebnis ist eine Zahl zwischen 0 und K_{max} mit $K_{max} \approx 1$, welche `n` berücksichtigt. Eine niedriger Kontingenzkoeffizient heißt, dass die generierten Zahlen gut verteilt sind und die tatsächlichen Werte nah an die theoretischen heranreichen. Ein höherer Kontingenzkoeffizient bedeutet, dass vermehrt Zahlen häufiger bzw. seltener als gewollt vorkommen.

Mit dem oben dargestellten Test hat sich ein `k` von ca. 0,003 ergeben, was sich als ein sehr gutes Ergebnis bezeichnen lässt. Wird allerdings die Methode `gen_rdm_16_bit_in_interval()` der im Codeausschnitt **??** dargestellten Datei aufgerufen und dabei die Werte 1024 und 49151 übergeben, so verschlechtert sich der Kontingenzkoeffizient auf einen mittelmäßig guten Wert von ca. 0,59.

Es lässt sich somit festhalten, dass die Verkleinerung des Intervalls der zurückgegebenen Zahl auf das von validen Portnummern eine Verschlechterung der Zufälligkeit des Algorithmus mit sich bringt, was allerdings kein Problem darstellt. Das ist eine logische Konsequenz aus der Tatsache, dass keine Zahlen außerhalb des Intervalls mehr zurückgegeben werden.

Die Methode `gen_rdm_32_bit()` und `gen_rdm_64_bit()` konnte wegen Fehlern aufgrund zu vieler zu großer Zahlen leider nicht ähnlich durchgeführt werden. Dennoch besteht Grund zu der Annahme, dass die berechneten integer-Werte ähnlich gut verteilt sind. Schließlich muss erneut darauf hingewiesen werden, dass bei beiden Methoden die Effizienz und nicht unbedingt die Qualität des Zufalls an erster Stelle steht.

4.1.6.4 Zeitlicher Vergleich mit rand()

Da der auf Xorshift basierende `RandomNumberGenerator` insbesondere aufgrund einer besseren Effizienz als die der Standardfunktion `rand()` implementiert wurde, ist ein Vergleich beider Zufallszahlengeneratoren von Interesse. Der für einen Vergleich benutze Test wird im Codeausschnitt 4.22 beispielhaft für die Methode `gen_rdm_32_bit()` gezeigt. Es wurden ebenfalls zwei äquivalente Sections für die andere Methode geschrieben. Dabei wurde stets darauf geachtet, dass die mit `rand()` generierten Zahlen das gleiche Intervall und die gleiche Größe wie beim RNG haben.

```
1 TEST_CASE("RandomNumberGeneratorTime", "[]") {
2     ...
3     SECTION("TestTime32", "[]") {
4         double time1 = 0.0, tstart;
5         tstart = clock();
```

```
6         RandomNumberGenerator xor_shift;
7         long n = 10000000;
8         uint32_t test_value;
9         for (long i = 0; i < n; i++) {
10            test_value = xor_shift.gen_rdm_32_bit();
11        }
12        time1 += clock() - tstart;
13        std::cout << "time needed to generate " << n
14        << " 32 bit numbers: " << time1 / CLOCKS_PER_SEC << " s"
15        << std::endl;
16        CHECK(time1 / CLOCKS_PER_SEC < 1.0);
17    }
18    SECTION("TestTime32Rand", "[]") {
19        double time1 = 0.0, tstart;
20        tstart = clock();
21        long n = 10000000;
22        uint32_t test_value;
23        for (long i = 0; i < n; i++) {
24            test_value = (uint16_t)rand();
25            test_value |= (uint16_t)rand() << 16;
26        }
27        time1 += clock() - tstart;
28        std::cout << "time needed to generate " << n
29        << " 32 bit numbers with rand() and shifting: "
30        << time1 / CLOCKS_PER_SEC << " s" << std::endl;
31        CHECK(time1 / CLOCKS_PER_SEC < 1.0);
32    }
33 }
```

Codeausschnitt 4.22: Test zum Vergleich der Zeiten vom RandomNumberGenerator mit rand()

Zum Erfassen der Zeiten wurde `time.h` inkludiert und in Z. 4 f. sowie 19 f. ein Timer initialisiert und gestartet. In diesem Fall werden 10 Mrd. integer-Werte generiert und kurzzeitig in einer Variable `test_value` gespeichert, was in den beiden for-Schleifen zu sehen ist. Es fällt auf, dass in Z. 25 eine Verschiebe-Operation verwendet wird, um sicherzustellen, dass es sich tatsächlich um 32 bit Zufall handelt. In den Zeilen 12 f. und 27 f. wird die Differenz zwischen Start- und Endzeitpunkt berechnet, welche darauf hin in Sekunden ausgegeben wird. Auch bei dem 64-bit-Vergleich wurde darauf geachtet, dass es sich bei der mit `rand()` generierten Zahl um echte 64 bit Zufall handelt. Das heißt, dass auch dafür mehrmals `rand()` aufgerufen werden musste.

Die Ergebnisse sind in der folgenden Tabelle dargestellt. Selbstverständlich unterscheiden sich die Zeiten bei jedem Ausführen des Tests, jedoch lediglich meist nur in hier nicht angegebenen Nachkommastellen.

Größe des generierten Wertes	**Xorshift-RNG**	**rand()**
16 bit	0,15 s	0,17 s
32 bit	0,15 s	0,33 s
64 bit	0,15 s	0,52 s

Wie zu erkennen ist, sind die Unterschiede zwischen dem selbst implementierten RNG und `rand()` minimal. Es bleibt anzumerken, dass der Unterschiede bei den anderen Größen auch nur deshalb größer wird, weil `rand()` dort mehrfach aufgerufen wird. Das ist allerdings auch nötig, weil sonst keine 32 bzw. 64 bit echten Zufall erhalten wird.

Es kann also festgehalten werden, dass sich Xorshift umso mehr lohnt, je größer die generierten Zahlen sein sollen. Auch George Marsaglia empfiehlt den Algorithmus in seinen Ausarbeitungen erst ab einer Größe von 32 bit [8].

4.1.7 Angreifer

Die vom `RandomNumberGenerator` generierten Werte können als IP-Adressen und als Portnummern vom Angreifer verwendet werden. Dieser `Attacker` wird zum Generieren der SYN-Flut benötigt. Somit ist er vor allem zum Testen dieser Attacke gedacht. Beim Angreifer kommt es nach allen notwendigen Initialisierungen zum Erstellen der benötigten `PacketContainer` für die Worker-Threads und zum Start sowie dem Ende der Attacke.

```
1  TEST_CASE("tsc timer", "[]"){
2      const uint64_t MAX_SECONDS = 30;
3      uint64_t cycles_old = 0;
4      uint64_t cycles = 0;
5      uint64_t hz = rte_get_tsc_hz();
6      uint64_t seconds = 0;
7      uint66_t delta_t = 0;
8
9      std::cout << "cycles : " << cycles << "\t"
10                   << "hz : " << hz << "\t"
11                   << "seconds : " << seconds << "\t" << std::endl;
12
13     while (seconds < MAX_SECONDS{
14     cycles_old = cycles;
15     cycles = rte_get_tsc_cycles();
16     hz = rte_get_tsc_hz();
17
18     delta_t =uint64_t(1/hz * (cycles - cycles_old));
19     seconds += delta_t;
20
21     std::cout << "cycles : " << cycles << "\t"
22                   << "hz : " << hz << "\t"
23                   << "seconds : " << seconds << "\t" << std::endl;
24     }
25 }
```

Codeausschnitt 4.23: Testen des Timers in `Attacker_test.cpp`

Im Testfall „tsc timer” werden die seit dem Teststart vergangenen Sekunden gezählt und ausgegeben (vgl. Codeausschnitt 4.23). Der Code zur Ausgabe befindet sich in den Zeilen 9 bis 11 und 21 bis 23. Nach 30 Sekunden endet der Test.

4.2 Testen anhand des Testdrehbuchs

4.3 Sonstige Tests am Testbed

toDo

Kapitel 5

Überprüfung der Anforderungen

In diesem Kapitel wird geklärt, in wie fern das entwickelte System den zu Beginn des Projekts aufgestellten funktionalen und nichtfunktionalen Anforderungen gerecht wird. Dafür wird zunächst kurz auf deren Priorisierung eingangen. Anschließend werden die Anforderungen aufgelistet und kurz erklärt, ob diese erfüllt worden oder nicht.

5.1 Priorisierung der Anforderungen

Um Anforderungen zu strukturieren und nach Wichtigkeit zu priorisieren, wird in der Regel ein System zur Klassifizierung der Eigenschaften verwendet. Hier wurde eine Priorisierung nach der **MuSCoW**-Methode vorgenommen:

Must: Diese Anforderungen sind unbedingt erforderlich und nicht verhandelbar. Sie sind erfolgskritisch für das Projekt.

Should: Diese Anforderungen sollten umgesetzt werden, wenn alle Must-Anforderungen trotzdem erfüllt werden können.

Could: Diese Anforderungen können umgesetzt werden, wenn die Must- und Should-Anforderungen nicht beeinträchtigt werden. Sie haben geringe Relevanz und sind eher ein „Nice to have“.

Won't: Diese Anforderungen werden im Projekt nicht explizit umgesetzt, werden aber eventuell für die Zukunft vorgemerkt.

5.2 Funktionale Anforderungen

Die funktionalen und die nichtfunktionalen werden in einzelnen Unterkapiteln getrennt behandelt. Für beide Arten wird zunächst die Tabelle aus dem Pflichtenheft erneut dargestellt. Nach der Auflistung wird eine Überprüfung zum einen anhand des Testdrehbuch und nach anderen Methoden vorgenommen.

5.2.1 Auflistung der funktionalen Anforderungen

Funktionale Anforderungen legen konkret fest, was das System können soll. Hier wird unter anderem beschrieben, welche Funktionen das System bieten soll. Die folgende Tabelle zeigt diese

funktionalen Anforderungen.

ID	Name	Beschreibung	MuSCoW
F01	Lokale Administration	Das System muss lokal per Command-Line-Interface administriert werden können.	Must
F02	Angriffsarten	Das System muss die Folgen der aufgelisteten (D)DoS-Angriffe abmildern können: • SYN-Flood • SYN-FIN Attack • SYN-FIN-ACK Attack • TCP-Small-Window Attack • TCP-Zero-Window Attack • UDP-Flood Dabei ist vorausgesetzt, dass das Ziel eines Angriffes eine einzelne Station in einem Netzwerk ist und kein Netzwerk von Stationen. Es sind also direkte Angriffe auf einzelne Server, Router, PC, etc. gemeint.	Must
F03	Keine zusätzliche Angriffsfläche	Besonders darf das System den unter „Angriffsarten" spezifizierten Angriffen keine zusätzliche Angriffsfläche bieten, d.h. es darf es auch nicht durch Kenntnis der Implementierungsdetails möglich sein, das System mit diesen Angriffen zu umgehen.	Must
F04	L3/ L4 Protokolle	Das System muss mit gängigen L3/ L4 Protokollen klarkommen.	Must
F05	Modi	Passend zum festgestellten Angriffsmuster muss das System eine passende Abwehrstrategie auswählen und ausführen.	Must
F06	Position	Das System soll zwischen dem Internet-Uplink und dem zu schützenden System oder einer Menge von Systemen platziert werden.	Must
F07	Weiterleiten von Paketen	Das System muss legitime Pakete vom externen Netz zum Zielsystem weiterleiten können.	Must
F08	Installation und Deinstallation	Das System muss durch Befehle in der Kommandozeile zu installieren und zu deinstallieren sein. Hilfsmittel hierzu sind: Installationsanleitung, Installationsskript, Meson und Ninja.	Must
F09	Mehrere Angriffe nacheinander und zeitgleich	Das System muss mehreren Angriffen nacheinander und zeitgleich standhalten, hierbei muss berücksichtigt werden, dass auch verschiedene Angriffsarten und Muster zur gleichen Zeit erkannt und abgewehrt werden müssen.	Must
F10	IPv4	Das System muss mit IPv4-Verkehr zurechtkommen.	Must

ID	Name	Beschreibung	MuSCoW
F11	Hardware	Das System soll nicht Geräte- bzw. Rechner-spezifisch sein.	Should
F12	Zugriff	Der Zugriff auf das lokale System soll per SSH oder Ähnlichem erfolgen, um eine Konfiguration ohne Monitor zu ermöglichen.	Should
F13	Betrieb	Das System soll auf Dauerbetrieb ohne Neustart ausgelegt sein.	Should
F14	Privacy	Das System soll keine Informationen aus der Nutzlast der ihm übergebenen Pakete lesen oder verändern.	Should
F15	Konfiguration	Der Administrator soll die Konfiguration mittels Konfigurationsdateien ändern können.	Can
F16	Abrufen der Statistik	Der Administrator soll Statistiken über das Verhalten des Systems abrufen können.	Can
F17	Starten und Stoppen des Systems	Der Administrator soll das System starten und stoppen können.	Can
F18	Informieren des Anwenders	Der Anwender soll über Angriffe informiert werden.	Can
F19	Administration über graphische Oberfläche	Das System soll über eine graphische Oberfläche administriert werden können.	Can
F20	IPv6	Das System soll mit IPv6-Verkehr zurechtkommen können	Can
F21	Weitere Angriffsarten	Das System schützt weder vor anderen außer den genannten DoS-Angriffen (siehe F02 „Angriffsarten“)-insbesondere nicht vor denjenigen, welche auf Anwendungsebene agieren-, noch vor anderen Arten von Cyber-Attacken, die nicht mit DoS in Verbindung stehen. So bleibt ein Intrusion Detection System weiterhin unerlässlich.	Won't
F22	Anzahl der zu schützenden Systeme	Das System wird nicht mehr als einen Server, Router, PC, etc. vor Angriffen schützen.	Won't
F23	Fehler des Benutzers	Das System soll nicht vor Fehlern geschützt sein, da es durch eine nutzungsberechtigte Person am System ausgeführt wird. So sollen beispielsweise Gefährdungen, welche aus fahrlässigem Umgang des Administrators mit sicherheitsrelevanten Softwareupdates resultieren, durch das zu entwickelnde System nicht abgewehrt werden.	Won't
F24	Softwareupdates	Das System soll keine Softwareupdates erhalten und soll nicht gewartet werden.	Won't

ID	Name	Beschreibung	MuSCoW
F25	Router-/Firewall-Ersatz	Das System soll nicht als Router oder als Firewall-Ersatz verwendet werden.	Won't
F26	Hardware-Ausfälle	Das System soll keine Hardwareausfälle (zum Beispiel auf den Links) beheben.	Won't
F27	Fehler in Fremdsoftware	Das System kann nicht den Schutz des Servers bei Fehlern in Fremdsoftware garantieren.	Won't

5.2.2 Überprüfung der funktionalen Anforderungen

Jede einzelne funktionale Anforderungen wird hier unter einer eigenen kleinen Überschrift überprüft. Die Reihenfolge ist dabei die gleiche wie in der oben stehenden Tabelle.

5.2.2.1 F01: Lokale Administration

Diese Muss-Anforderung wurde erfüllt. Ein Command-Line-Interace wurde mithilfe von Konventionen wie „Human first design" oder der Forderung, dass das Programm bei der Benutzung unterstützt. So soll es beispielsweise vorschlagen, was als Nächstes gemacht werden kann.

Mit Hilfe der Kommandos „help" oder „h" bekommt der Nutzer alle gültigen Eingaben angezeigt.

Die Eingabe von „exit"

5.2.2.2 F02: Angriffsarten

Das System kann alle geforderten (D)DoS Angriffe abmildern oder gänzlich verhindern. SYN-FIN, SYN-FIN-ACK sowie Zero Window und Small Window können vollständig erkannt und abgewehrt werden. Die UDP-/TCP- und ICMP-Flood können in ihrer Form abgemildert werden.

5.2.2.3 F03: Keine zusätzliche Angriffsfläche

Durch unvollständigkeit des Systems kann diese Anforderung nicht vollständig überprüft werden jedoch ist diese soweit das System implementiert wurde beständig.

5.2.2.4 F04: L3/L4 Protokolle

Sowohl Protokoll L3, zuständig für die Vermittlung von Daten über einzelne Verbindungsanschnitte und Netzwerkknoten und Adressierung der Kommunikationspartner, als auch Protokoll L4, das die Quell- und Zieladressen und weitere Informationen des Pakets enthält, werden vom System akzeptiert und verwendet.

5.2.2.5 F05: Modi

Das System erkennt und unterscheidet verschiedene Angriffsmethoden und errechnet selbst die passende Abwehrstrategie sowie eine Anpassung der Durchlassrate von Paketen. Die Abwehrstrategie wird von jedem Worker-Thread an seinen eigenen Verkehr angepasst.

5.2.2.6 F06: Position

In Kapitel 2 ist auf Seite 9 in Abbildung 2.2 der Versuchsaufbau zu sehen. Das System wurde im Labor im Zusebau der TU Ilmenau auch tatsächlich in dieser Reihenfolge aufgebaut. Damit ist diese Anforderung erfüllt.

5.2.2.7 F07: Weiterleiten von Paketen

Zur Überprüfung dieser Anforderung ist der erste Test des in der Planungs- und Entwurfsphase geschriebenen Testdrehbuchs gedacht. Für den Test der Paketweiterleitung werden zunächst Pakete mit DPDK von einem Port der Netzwerkkarte entgegengenommen und auf den andern Port weitergegeben. Danach wurde begonnen, einzelne Ping-Anfragen vom äußeren System über die Mitigation-Box zum Server laufen zu lassen. Im Anschluss wurde ein Lasttest durchgeführt. Diese Tests haben ergeben, dass...

5.2.2.8 F08: Installation und Deinstallation

Das System wird mit einer Installationsanleitung und Installationsskripten ausgeliefert. Das Installationsskript für abhängige und notwendige Systemeinstellungen und Programme installiert alle notwendigen zusätzlichen Programme und Bibiliotheken und nimmt alle notwendigen Systemeinstellungen vor, soweit möglich. Bei Fehlern wird dem Benutzer ein Hinweis zur Lösung des Problems angezeigt. Die Installation kann auch selbst mit der Installationsanleitung vorgenommen werden, die einzelnen Schritte sind in eigenen Unterkapiteln genauer erklärt. Für einen lokalen Bau der Software kann `Meson` und `Ninja` verwendet werden, deren Benutzung in der die Installationsanleitung fortführenden Seite `Usage` erklärt wird. Dort ist auch ein kurzer Einstieg in die Benutzung von `AEGIS` erklärt. Zur systemweiten Installation von `AEGIS` kann ebenfalls `Meson` genutzt werden, das durch ein Installationsskript erweitert wurde. Das gesamte Programm und zusätzlich installierten Programme können mit einem Deinstallationsskript wieder vom System gelöscht werden.

Diese Anforderung ist erfüllt.

5.2.2.9 F09: Mehrere Angriffe nacheinander und zeitgleich

Das System ist darauf spezifiziert, einzelne Angriffe und kombinierte Angriffe zu erkennen. Kombinierte Angriffe aus unterschiedlichen angriffsmethoden können erkannt und abgewehrt werden. Ein vollständiger Test konnte mit dem bisherigen Stand des Projektes noch nicht durchgeführt werden.

5.2.2.10 F10: IPv4

Das System kann IPv4 Verkehr vollständig verarbeiten, verwalten und stößt auf keine Probleme dabei. Diese Anforderung ist erfüllt.

5.2.2.11 F11: Hardware

Diese Should-Anforderung wurde mit dem entwickelten System nicht erfüllt. Die Software läuft im derzeitigen Zustand ausschließlich auf dem Testbed im Rechenlabor. Durch kleine Anpassung kann die Nutzung auf alternativer Hardware allerdings ermöglicht werden.

Eine zusätzliche Beschränkung der Hardware besteht in der Nutzung von DPDK und Kernel Bypässen die von der verbauten Netzwerkkarte der Hardware unterstützt werden muss.

5.2.2.12 F12: Zugriff

Es ist ein ssh-Zugriff auf das System möglich. Die Konfiguration ermöglicht die Authentifizierung nicht mit einem Passwort möglich ist. Mithilfe des Befehls `ssh-keygen` kann ein Schlüsselpaar generiert werden und ein Public-Key ist in einer Textdatei zu finden. Weiterhin ist es möglich, zu überprüfen, welche anderen Teammitglieder derzeit auf diese Art mit dem System verbunden sind.

5.2.2.13 F13: Betrieb

Zum Zeitpunkt des Projektendes konnte das System in kurzer Zeit betriebsfähig gehalten werden jedoch kein Dauerbetrieb mit Dauerbelastung ausreichend getestet werden. Neustarts zwischen starten und stoppen des Systems waren nicht notwendig. Die Anforderung wurde teilweise erfüllt.

5.2.2.14 F14: Privacy

Dem System ist es möglich auf Paketdaten zuzugreifen und zu lesen. Die Implementierung wurde jedoch genau darauf eingeschränkt, nur Teile der Paketdaten, die zur Erfüllung der Systemaufgaben notwendig sind, zu lesen und gegebenfalls zu verändern. Von dem System kann dadurch keine Nutzlast von Paketen gelesen oder verändert werden. Wird ein Paket als (D)DoS Attacke erkannt wird das Paket mitsamt seiner Nutzlast gelöscht. Ein Eingriff in Privatsphäre oder Analyse von Benutzerdaten außerhalb der systemrelevanten Spezifikationen erfolgt zu keiner Zeit.

Damit ist diese Anforderung erfüllt.

5.2.2.15 F15: Konfiguration

Der Administrator kann die Einstellungen von `AEGIS` durch zwei Konfigurationsdateien anpassen. Innerhalb der `meson_options.txt` Datei kann der Bau von Unit Tests und der Dokumentation ein- und ausgeschalten werden. In der Datei `config.json` können verschiedene Einstellungen wie die zu verwendenden Systemkerne und Durchlassraten eingestellt werden.

5.2.2.16 F16: Abrufen der Statistiken

Die globale Statistik mit Informationen über den Datenverkehr soll innerhalb des CLI abgerufen und angezeigt werden. Die Implementierung ist jedoch nicht bis dahin entwickelt. Die Anforderung ist nicht erfüllt.

5.2.2.17 F17: Starten und Stoppen des Systems

Das Programm `AEGIS` kann über ein Terminal vom Benutzer gestartet und gestoppt werden. Das CLI unterstützt den Nutzer dabei mit Hinweisen. Die Anforderung ist erfüllt.

5.2.2.18 F18: Informieren über graphische Oberfläche

Für die Präsentation ist eine graphische Oberfläche in Arbeit. Diese ist bisher aber nicht fertig und erfüllt die Anforderung noch nicht.

5.2.2.19 F19: Administration über grafische Oberfäche

Diese Can-Anforderung wurde aus Zeitgründen nicht umgesetzt. Die Administration erfolgt stattdessen über das in F01 beschriebene CLI.

5.2.2.20 F20: IPv6

Das System ist vorerst auf IPv4 ausgelegt und funktionsfähig aber schon darauf ausgerichtet auch IPv6 zu unterstützen. Die Anforderung ist nicht erfüllt kann aber später ohne große Abänderung am gegebenen System hinzugefügt werden.

5.2.2.21 F21: Weitere Angriffsarten

Das System schützt nur vor den in F02 angegebenen (D)DoS Angriffen und stellt keinen Schutz gegen weitere mögliche Angriffe. Die Verwendung weiterer Sicherheitsmechanismen ist weiterhin unerlässlich.

5.2.2.22 F22: Anzahl der zu schützende Systeme

Das System schützt nur ein einzelnes System. Mit moderaten Änderungen kann die Software aber auf anderen Servern etc. installiert werden und dadurch mehrere schützen. Aus hardwareseitigen Gründen wurde dies aber auch nicht getestet.

5.2.2.23 F23: Fehler des Benutzers

Auch nach der Entwicklung kann festgehalten werden, dass das System durch einen kompetenten Administrator installiert und genutzt werden muss. Fehler durch den Benutzer oder falsche Verwendung können vom System selbst nicht behoben werden.

5.2.2.24 F24: Softwareupdates

Das System, das während des Projekts entwickelt wurde, wird nach Abschluss der Lehrveranstaltung nicht direkt weiterentwickelt. Während der Erstellung dieses Dokuments wird allerdings davon ausgegangen, dass einige Studierende der Gruppe auch nach Abschluss der Lehrveranstaltung evtl. noch an dem System weiterarbeiten. Features, die möglicherweise dadurch noch hinzugefügt werden, können allerdings nicht als Softwareupdates angesehen werden.

5.2.2.25 F25: Router-/Firewall-Ersatz

Auch diese Won't-Anforderung wurde nicht erfüllt. Eine Firewall oder vergleichbare schützende Systeme bleiben nach wie vor unerlässlich.

5.2.2.26 F26: Hardware-Ausfälle

Aegis kann keine Hardwareausfälle beheben.

5.2.2.27 F27: Fehler in Fremdsoftware

Ebenso gibt keine Möglichkeit das System vor Fehlern einer anderen Software oder Softwareabhängigkeit zu schützen. Der Benutzer ist für die korrekte Installation und Konfiguration aller notwendigen Fremdsoftware und Abhängigkeiten verantwortlich. Die Installations- und Deinstallationsskripte, sowie Dokumentation bieten dem Benutzer Hilfe diese Fehler zu umgehen.

5.3 Nichtfunktionale Anforderungen

Auch hier wird genau wie bei den funktionalen Anforderungen vorgegangen.

5.3.1 Auflistung der nichtfunktionalen Anforderungen

Nichtfunktionale Anforderungen gehen über die funktionalen Anforderungen hinaus und beschreiben, wie gut das System eine Funktion erfüllt. Hier sind zum Beispiel Messgrößen enthalten, die das System einhalten soll. Im folgenden werden diese nichtfunktionalen Anforderungen beschrieben.

ID	Name	Beschreibung	MuSCoW
NF01	Betriebssystem	Die entwickelte Software muss auf einer Ubuntu 20.04 LTS Installation laufen. DPDK muss in Version 20.11.1 vorliegen und alle Abhängigkeiten erfüllt sein.	Must
NF02	Verfügbarkeit	Die Verfügbarkeit des Systems soll bei mindestens 98% liegen. Verfügbarkeit heißt hier, dass das System in der Lage ist, auf legitime Verbindungsanfragen innerhalb von 10 ms zu reagieren.	Must
NF03	Datenrate	Die anvisierte Datenrate, welche vom externen Netz durch das zu entwickelnde System fließt, muss bei mindestens 20 Gbit/s liegen.	Must
NF04	Paketrate	Die anvisierte Paketrate, welche vom zu entwickelnden System verarbeitet werden muss, muss bei mindestens 30 Mpps liegen.	Must
NF05	Transparenz	Der Anwender soll das Gefühl haben, dass die Middlebox nicht vorhanden ist.	Should
NF06	Abwehrrate SYN-Flood	Die für die Angriffe anvisierten Abwehrraten sind für die SYN-Flood, SYN-FIN und SYN-FIN-ACK jeweils 100%.	Should
NF07	False Positive	Der maximale Anteil an fälschlicherweise nicht herausgefiltertem und nicht verworfenem illegitimen Traffic, bezogen auf das Aufkommen an legitimem Traffic, soll 10% im Angriffsfall und 5% im Nicht-Angriffsfall nicht überschreiten.	Should
NF08	False Negative	Der maximale Anteil an fälschlicherweise nicht verworfenem bösartigem Traffic, bezogen auf das Gesamtaufkommen an bösartigem Traffic, soll 5% nicht überschreiten.	Should
NF09	Round Trip Time	Die Software soll die Round-Trip-Time eines Pakets um nicht mehr als 10 ms erhöhen.	Should

5.3.2 Überprüfung der nichtfunktionalen Anforderungen

Auch bei den nichtfunktionalen Anforderungen werden jeweils einzelne Unterkapitel genutzt.

5.3.2.1 NF01: Betriebssystem

Die genannte Software, also Ubuntu 20.04 LTS und DPDK 20.11.1, wurde von allen Teammitgliedern installiert. Schließlich wurde auch nur mit diesen Versionen des Betriebssystems bzw. Frameworks entwickelt und getestet. Es kann also dokumentiert werden, dass das System unter diesen Voraussetzungen wie bei den anderen Tests beschrieben funktioniert. Es kann keine Aussage darüber getroffen werden, inwiefern das System unter anderen Versionen funktioniert.

5.3.2.2 NF02: Verfügbarkeit

5.3.2.3 NF03: Datenrate

5.3.2.4 NF04: Paketrate

5.3.2.5 NF05: Transparenz

5.3.2.6 NF06: Abwehrrate SYN-Flood

5.3.2.7 NF07: False Positive

5.3.2.8 NF08: False Negative

5.3.2.9 NF09: Round Trip Time

Kapitel 6

Bug-Review

In diesem Bug-Review werden verschiedene Fehler gesammelt. Dabei wird auf die Datei, in der sie auftreten, auf eine Beschreibung und eine Kategorisierung eingegangen. Major Bugs sind versionsverhindernd, critical bugs können auch zur Arbeitsunfähigkeit anderer führen und minor bugs haben eine geringe Auswirkung und somit eine niedrige Priorität.

Datei	**Beschreibung**	**Kategorie**
PacketInfoIpv4Icmp	Wenn von einem Paket die Header extrahiert werden soll (fill_info), wird zuerst der mbuf in der `PacketInfo` verlinkt, dann IP version (IPv4) und Layer 4 Protokol (ICMP) ermittelt. Danach wird die `PaketInfo` in die entsprechende protokolspezifische `PacketInfo` gecastet. Auf dieser verwandelten `PacketInfo` wird set_ip_hdr ausgeführt und es kommt zum segmentation fault, der im Abbruch des Threads mündet.	critical bug
Initializer	Die maximale Anzahl an Threads ist 16. Das stellt kein Problem dar, weil nur 12 Threads benötigt werden. mlx5_pci: port 1 empty mbuf pool; mlx5_pci: port 1 Rx queue allocation failed: Cannot allocate memory. Dieser Fehler tritt beim Ausführen von rte_eth_dev_start(port) auf. Womöglich handelt es sich dabei um ein mempool problem.	minor bug

Schon während der Implementierungsphase wurden Bugs wenn möglich behoben. An den in dieser Tabelle genannten Problemen wird noch gearbeitet. Die Fehlerliste wird auch in der Validierungsphase laufend erweitert, sodass im Idealfall für das abschließende Review-Dokument eine vollständige Bug-Statistik erstellt werden kann.

toDo

Kapitel 7

Softwaremetriken und Statistiken

Der Zweck von Softwaremetriken besteht in der „Definition von Software-Kenngrößen und Entwicklungsprozessen" und der „Abschätzung des Prozess- und Kostenaufwands oder dem Qualitätsmanagement" [9].

Da das Reviewdokument noch vor Projektende fertiggestellt werden musste, sind die Daten, auf denen dieses Kapitel basiert, vom 10.07.2021.

7.1 Benennungs- und Programmierkonventionen

Während der Meetings wurde sich auf zahlreiche Konventionen geeinigt. Diese wurden dann wiederum sowohl in den einzelnen Meetingprotokollen als auch in einem eigenen Wikieintrag festgehalten.

Unter Programmierkonventionen werden „Vorgaben für Programmierer über die Gestaltung von Programmen" [10] verstanden. Sie zeigen auf, wie der Code sowohl formal als auch strukturell gestaltet sein soll. Diese Konventionen sollen zur Verbesserung der Softwarequalität führen, was sich unter anderem in der durch die Konventionen gesteigerten Verständlichkeit und Änderungsfreundlichkeit des Codes zeigt.

7.1.1 C/C++ und UML

Das System wird in der Programmiersprache C++ entwickelt. Um dieses System zu entwerfen, wurden verschiedene Modelle und Diagramme im Rahmen des Softwareprojektes erstellt (z.B. Klassendiagramme, Aktivitätsdiagramme, Sequenzdiagramme). Diese Diagramme wurden mithilfe der Unified Modeling Language (UML) entwickelt. Die UML gibt bereits einige Richtlinien vor, wie zum Beispiel die grafische Notation oder die Bezeichner für die bei einer Modellierung wichtiger Begriffe.

7.1.1.1 Namenskonventionen

Im Folgenden werden die Namenskonventionen im vorliegenden Softwareprojekt aufgezeigt und mit kurzen Beispielen unterlegt:

Klassen, **Enumerations** und **Pakete** werden entsprechend der UpperCamelCase-Schreibweise dargestellt. Allerdings muss darauf geachtet werden, dass z. B. bei Akronymen nicht mehrere Buchstaben hintereinander in Großbuchstaben geschrieben werden. Dementsprechend entspricht der Name `ConfigurationManagement` den Konventionen. Allerdings ist `TCPTreatment` syntaktisch nicht korrekt, da es den Vorgaben nicht entspricht, der Begriff `TcpTreatment` hingegen wäre syntaktisch korrekt.

Für **Methoden** werden ausschließlich Kleinbuchstaben verwendet. Sollte sich der Methodenname aus mehreren Worten zusammensetzen, kann dies über einen Unterstrich erfolgen. Beispiele für richtig benannte Methoden sind demzufolge `send_packets_to_port()` und `check_syn_cookie()`. Es bleibt anzumerken, dass statische Methoden durch ein `s_` vor dem eigentlichen Namen gekennzeichnet werden, wie bei `s_increment_timestamp()`.

Für **Variablen** gelten die gleichen Konventionen wie für Methoden, sodass `packet_inside` als Bespiel dienen kann. Zusätzlich soll ein Unterstrich vor dem Namen darauf hinweisen, dass es sich um eine Membervariable handelt, wie z. B. bei `_cookie_secret`. Statische Membervariablen beginnen somit mit `_s_`, wie bei `_s_timestamp`.

Auch bei **Objekten** gilt, dass nur Kleinbuchstaben verwendet werden sollen und mehrere Worte durch einem Unterstrich verbunden werden, wie in `nic_man`.

7.1.1.2 Formatierungs-Richtlinien

Die Formatierungsrichtlinien legen unter anderem fest, dass nur ASCII-Zeichen, also zum Beispiel keine Umlaute oder ß, verwendet werden dürfen. Die Einrückung im Code beträgt vier Leerzeichen. Zudem sollen „dauerhaft" geschweifte Klammern verwendet werden. Das heißt zum Beispiel, dass auch geschweifte Klammern einzeiliger if-Blöcken verwendet werden. Nach Methoden- und Klassennamen (oder Ähnlichem) stehen öffnende geschweifte Klammer. Hier soll kein Zeilenumbruch entstehen. Dies zeigt der Codeausschnitt 7.1 beispielhaft.

```
1  int i = rand();
2
3  // It should be like this:
4  if(i%2 == 0){
5      ...
6  }
7
8  //It should be not like this:
9  if(i%2 != 0)
10 {
11     ...
12 }
13
14 //And not like this:
15 if(i>100)
16     ...
```

Codeausschnitt 7.1: Formatierungsrichtlinie: Setzen von Klammern

7.1.1.3 Kommentare

Während in den Kommentaren das Festschreiben von ToDo's erlaubt ist, dürfen hier keine Fragen gestellt werden. In den Headerdateien soll die Doxygen-Syntax für Kommentare verwendet werden, um hiermit unter anderem die Entwicklerdokumentation zu generieren. Vor einem Block können mehrzeilige Kommentare verwendet werden. Derjenige Teil, der mit `*` beginnt, muss jeweils nochmals mit einem Leerzeichen eingerückt werden.

```
 /**
* Full description
*
* @brief short description
* @param msg message that is printed to the console
*/
void log(const string* msg){
    std::cout << msg << std::endl;
}
```

Codeausschnitt 7.2: Beispiel für einen mehrzeiligen Doxygen-Kommentar

Einzeilige Kommentare werden dadurch erzeugt, indem hinter einer Codezeile `///<` eingegeben wird. Diese Art von Kommentaren wird von Doxygen als Kurzbeschreibung verwendet.

```
string firstname;  //< first name of person
string lastname;   //< last name of person
```

Codeausschnitt 7.3: Beispiel für einen einzeiligen Doxygen-Kommentar

Es wurde sich auf die Verwendung folgender **Commands** geeinigt:

Bei `@brief` lässt sich sofort erkennen, dass es sich um eine **Kurzbeschreibung** handelt.

`@param` leitet hingegen die Beschreibung eines **Parameters**, der in eine Methode ein- oder von einer Methode ausgegeben wird, ein. `@param[in]` steht vor der Beschreibung eines **Eingabeparameters** und `@param[out]` vor einem **Ausgabeparameter**. Bei diesem handelt es sich um einen Parameter, der im C-Style einer Funktion mit Call-by-reference übergeben wird, damit er mit Werten gefüllt wird. Zur Beschreibung von Parametern, die sowohl ein- als auch ausgegeben werden, kann `@param[in, out]` verwendet werden.

Der Command `@return` hilft bei der Beschreibung eines **Rückgabewertes** und `@file` zur Erklärung des **Zwecks** einer Datei, also z. B. einer Klasse oder eines structs.

7.1.1.4 Source-Dateien

Pro Paket (vgl. Abb. 2.5) wird ein Ordner in `source/` angelegt, der den gleichen Namen wie das Paket erhält. Alle zu diesem Paket zugehörigen Klassen befinden sich dann wiederum in diesem Ordner. Pro cpp-Klasse soll es eine eigene .cpp-Datei geben. Die dazugehörige Header-Datei wird mit `#include <header-file>` inkludiert. Im ganzen Projekt gibt es eine `main.cpp`-Datei in `source/` mit der main-Routine.

7.1.1.5 Header-Dateien

Pro Source-Datei existiert eine Header-Datei. Jede dieser Header-Dateien wird mit `.hpp` benannt. Am Anfang der Datei wird `#pragma once` verwendet. Externe Dateien werden mit `#include`

`<file>` inkludiert.

7.1.2 Gitlab

Im Softwareprojekt wird die GitLab zur Versionsverwaltung genutzt. Diese Webanwendung basiert auf Git.

7.1.2.1 Git

Außer in Präsentationen und Review-Dokumenten werden in Git nur ASCII-Zeichen verwendet. Die **Sprache** ist auch hier grundsätzlich Englisch, wobei diese Festlegung auch bei den Präsentationen und Review-Dokumenten abweicht.

Die **Branchnamen** sind klein geschrieben. Die Worte in diesem werden mit Unterstrich („_”) verbunden. Dieselbe Regelungen gelten bei **Ordner- und Dateinamen**.

7.1.2.2 Issue

Grundsätzlich werden die Issues in Englisch benannt. Lediglich deutsche „Eigennamen” werden auf Deutsch geschrieben (Beispiel: Pflichtenheft). Die Issues werden klein und im Imperativ geschrieben. Leerzeichen sind erlaubt.

Wenn Issues nicht nur einer sondern mehreren Personen zugeordnet werden sollen, wird dem eigentlichen Issue-Namen mit „@” die Namen der zuständigen Teammitgliedern angehängt. Das heißt, der Issue-Name weist folgende Struktur auf: `<issue name> <space> @ <first name>`).

Mit `@all` ist das Issue für alle Teammitglieder für die Bearbeitung offen. Derjenige, der mit der Bearbeitung dieser Aufgabe beginnt, löscht „@all” trägt seinen eigenen Vornamen mit „@” in den Issue-Namen ein.

7.1.2.3 Label

In diesem Projekt werden die Labels grundsätzlich für zwei verschiedene Kategorien von Aufgaben verwendet verwendet: Zum einen für Status-Issues und zum anderen für Super-Issues (bzw. Tasks). Bei diesen Super-Issues handelt es sich um große Aufgaben, denen wiederum verschiedene Issues als Teilaufgaben zugeordnet werden.

Die Benennung sieht Folgendes vor:

Abbildung 7.1: Beispiel: Label für Status-Issus

Label für `Status-Issues` werden folgendermaßen benannt: `STATUS: <label name>`. Diese werden zur Darstellung auf dem Issue-Board verwendet. `STATUS: in process` (siehe Abb. 7.1) und `STATUS: to be tested or presented` sind Beispiele für diese Status-Issues.

Abbildung 7.2: Beispiel: Label für Super-Issus/ Tasks

Label für `Super-Issues bzw. Tasks` werden folgendermaßen benannt: `TASK <label name>`. Jedes Issue soll genau einem Ticket zugeordnet werden, um zu erkennen, welcher großen Aufgabe das Issue zugehört. Ein Beispiel für ein solches Label findet sich in Abbildung 7.2.

Abbildung 7.3: Beispiel: Label für außerordentliche Kategorien

Label, die keiner dieser beiden Kategorien zugeordnet werden könne, werden folgendermaßen benannt: `EXTRA: <label name>`. Als Beispiel hierfür kann das Label `EXTRA: open for all` genannt werden, welches es den Teammitgliedern verinfachen soll, noch nciht zugeteilte Aufgaben zu finden.

7.1.3 Latex

Zum Erstellen der Reviewdokumente und der Präsentationen wird Latex verwendet. Hierbei wurde sich darauf geeinigt, unter anderem um Merge-Konflikte zu verhindern, dass ein Tab vier Leerzeichen entspricht. Außerdem sollen neue Kapitel mit einem Kommentarblock eingeleitet werden, um sich besser in den Dokumenten zurechtzufinden.

7.2 Umfang der Software

Unter **Lines of Code** (Loc) wird die Anzahl der Zeilen inklusive Leerzeilen und Kommentare verstanden. Dagegen enthalten die **Source Lines of Code** (SLOC) nur Codezeilen ohne Leerzeilen und Kommentare. Die **Comment Lines of Code** (CLOC) geben die Anzahl der Kommentarzeilen an. Dabei werden in diesem Dokument in diese Zeilenanzahl keine Zeilen miteinbezogen, welche zugleich Code und Kommentaren beinhalten.

Die Lines of Codes von allen Header-Dateien betragen ungefähr 2000. Das davon über die Hälfte Kommentarzeilen sind, überrascht nicht. Denn die gesamte Codedokumentation bzw. Entwicklerdokumentation erfolgte in diesem Projekt mittels Doxygen, was die automatische Generierung von Dokumentation auf Basis spezieller Kommentare. Diese Dokumentation mittels Kommentarzeilen fand in den Header-Dateien statt.

Header-Datei	**LOC**	**CLOC**	**SLOC**
AttackThread.h	91	25	44
Cli.h	32	2	24
Configurator.h	114	58	45
DebugHelper.h	10	0	7
DefenseThread.h	36	17	13
Definitions.h	24	7	9
ForwardingThread.h	30	14	11
Initializer.h	52	27	18
Inspection.h	149	107	33
NetworkPacketHandler.h	62	44	13

Header-Datei	**LOC**	**CLOC**	**SLOC**
PacketContainer.h	320	235	55
PacketInfo.h	82	38	26
PacketInfoCreator.h	43	18	18
PacketInfoIpv4.h	79	39	25
PacketInfoIpv4Icmp.h	39	17	14
PacketInfoIpv4Tcp.h	156	99	35
PacketInfoIpv4Udp.h	56	27	18
PacketInfoIpv6Icmp.h	36	7	22
PacketInfoIpv6Tcp.h	71	21	41
PacketInfoIpv6Udp.h	39	7	26
PacketProtTcp.h	138	89	31
PacketProtUdp.h	31	16	11
rand.h	42	19	16
RandomNumberGenerator.h	56	33	16
StatisticsThread.h	43	0	34
Thread.h	20	6	10
Treatment.h	215	129	54
$\sum$	2066	1101	669

Von den insgesamt über 2500 LoC in den Source-Dateien sind ca. 13,25% Kommentarzeilen und ca. 67,41% Codezeilen ohne Kommentare und Leerzeilen.

Source-Datei	**LoC**	**CLOC**	**SLOC**
Attacker/main.cpp	103	10	67
AttackThread.cpp	206	13	150
Cli.cpp	102	18	57
Configurator.cpp	95	12	69
DebugHelper.cpp	64	9	43
DefenseThread.cpp	68	17	31
ForwardingThread.cpp	6	0	5
Initializer.cpp	249	29	181
Inspection.cpp	145	22	109
main.cpp	106	10	69
NetworkPacketHandler.cpp	41	1	32
PacketContainer.cpp	230	5	168
PacketInfo.cpp	73	18	38

Source-Datei	**LoC**	**CLOC**	**SLOC**
PacketInfoCreator.cpp	149	9	116
PacketInfoIpv4.cpp	57	0	44
PacketInfoIpv4Icmp.cpp	18	0	14
PacketInfoIpv4Tcp.cpp	110	3	77
PacketInfoIpv4Udp.cpp	30	0	23
PacketInfoIpv6Icmp.cpp	36	11	18
PacketInfoIpv6Tcp.cpp	60	11	34
PacketInfoIpv6Udp.cpp	50	11	30
PacketProtTcp.cpp	88	1	71
PacketProtUdp.cpp	10	0	7
StatisticsThread.cpp	55	10	34
Thread.cpp	5	0	3
Treatment.cpp	394	118	229
$\sum$	2550	338	1719

Die Test-Dateien beinhalten den Code der Unit-Tests. Über die Hälfte der LOC aller Test-Dateien kommen aus der Datei `Treatment_test.cpp`. In den Test-Dateien wurdemusste vergleichsweise wenig kommentiert werden, wenn die Namen der Testcases und der Sections aussagekräftig gewählt wurden und diese Aussagekraft durch die Strukturierung der Tests unterstützt wurde.

Test-Datei	**LoC**	**CLOC**	**SLOC**
Attacker_test.cpp	37	7	24
Configurator_test.cpp	52	0	43
Inspection_test.cpp	15	0	23
libdpdk_dummy_test.cpp	14	0	11
PacketContainer_test.cpp	299	4	237
PacketInfo_test.cpp	131	57	62
RandomNumberGenerator_test.cpp	456	106	315
Treatment_test.cpp	1170	95	846
$\sum$	2174	268	1561

Die lipdpdk-Header-Dateien wurden hier extra aufgeführt, da sie nicht komplett selbst geschrieben wurden, sondern vor allem durch Kopieren und Anpassung entstanden sind.

lipdpdk_dummy-Header-Datei	**LoC**	**CLOC**	**SLOC**
rte_branch_prediction.h	33	20	9
rte_byteorder.h	62	20	31
rte_common.h	14	0	12
rte_config.h	3	0	2
rte_cycles.h	14	1	10
rte_eal.h	1	0	1
rte_ethdev.h	44	20	20
rte_ether.h	23	0	18
rte_icmp.h	12	0	10
rte_ip.h	139	7	109
rte_lcore.h	3	0	2
rte_mbuf.h	136	4	104
rte_mbuf_core.h	8	0	5
rte_mempool.h	3	0	2
rte_tcp.h	16	0	14
rte_udp.h	10	0	8
$\sum$	527	72	357

Um das Testbed effizient nutzen zu können, wurden kurze Skripte zur Initialisierung der Hardware geschrieben. Insgesamt entstanden hierbei 26 LoC.

Skripte zur Intialisierung der Hardware	**LoC**	**CLOC**	**SLOC**
namespace_bob.sh	13	6	7
namespace_alice.sh	13	6	7
$\sum$	26	12	14

Insgesamt entstanden im vorliegenden Projekt ca. 7300 LOC, fast 1800 CLOC und ca. 4300 SLOC.

Datei-Art	**LoC**	**CLOC**	**SLOC**
Header-Dateien	2066	1101	669
Source-Dateien	2550	338	1719
Test- Dateien	2174	268	1561
lipdpdk_dummy-Header-Dateien	527	72	357
Skripte zur Initialisierung der Hardware	26	12	14

Datei-Art	LoC	CLOC	SLOC
$\sum$	7343	1791	4320

7.3 Repository-Analyse in Gitlab

Ein Teil der Funktionalität von Gitlab bilden die Repository-Analysen. Diese werden verwendet, um sich einen stark abstrahierten Überblick über das Git-Repository eines Projektes zu verschaffen. Die dort zur Verfügung gestellten Diagramme werden nach jedem Commit aktualisiert.

7.3.1 Verwendete Programmiersprachen

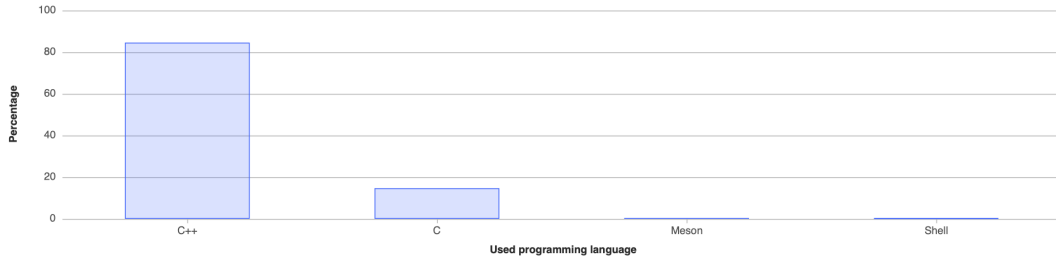


Abbildung 7.4: Verwendete Programmiersprachen in Prozent, gemessen an den Bytes of Code

Abbildung 7.4 zeigt, dass im Repository des vorliegenden Softwareprojektes vorwiegend mit 84,6% die Programmiersprache C++ verwendet wurde. Mit großen Abstand folgt C mit 14,74%. Der C-Code resultiert aus die Beispielprogramme von DPDK, die verwendet werden, um die Funktionalität von DPDK zu testen. Diese Beispielprogramme werden in dem Repoisitory, das später auf Github veröffentlicht wird, nicht mehr vorhanden sein. Meson (0,44%) und Shell (0,23%) stellen keinen wesentlichen Anteil dar.

7.3.2 Commit-Statistik

Insgesamt gab es vom 03.05.2021 bis zum 08.07.2021 im Master-Branch 1183 Commits. Merge-Commits sind hierbei ausgeschlossen. Aus diesen Zahlen lässt sich folgern, dass es durchschnittlich 17,7 Commits pro Tag gab. Abbildung 7.5 macht deutlich, das die Anzahl der Commits pro Tag des Monats stark schwankte. An manchen Tagen gab es weniger als 10 Commits (z.B. Tag 27), an anderen beinahe 80 (z.B. Tag 7, Tag 26). Abbildung 7.6 zeigt, dass es dienstags die meisten Commits pro Wochentag gab (215 Commits pro Tag). Die Wochentage Donnerstag (205 Commits pro Tag), Mittwoch (181 Commits pro Tag), Sonntag (170 Commits pro Tag), Montag (148 Commits pro Tag) und Freitag (139 Commits pro Tag) folgen. Mit 125 Commits pro Tag gab es die wenigsten Commits samstags. In Abbildung 7.7 sind die Commits pro Stunde des Tages dargestellt. Die Abbildung macht deutlich, dass kaum nachts gearbeitet wurde, insbesondere kaum zwischen 2 und 5 Uhr. Dagegen gab es vor allem am späten Vormittag bis späten Nachmittag viele Commits.

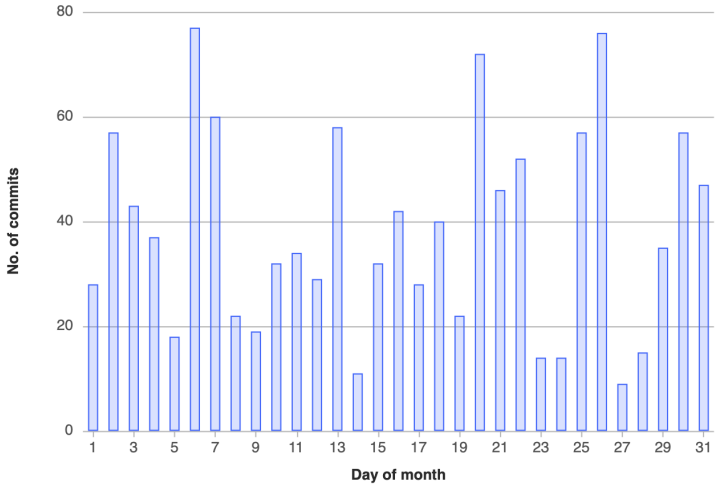


Abbildung 7.5: Commits pro Tag des Monats, Zeitraum: 03.05.2021 bis 08.07.2021

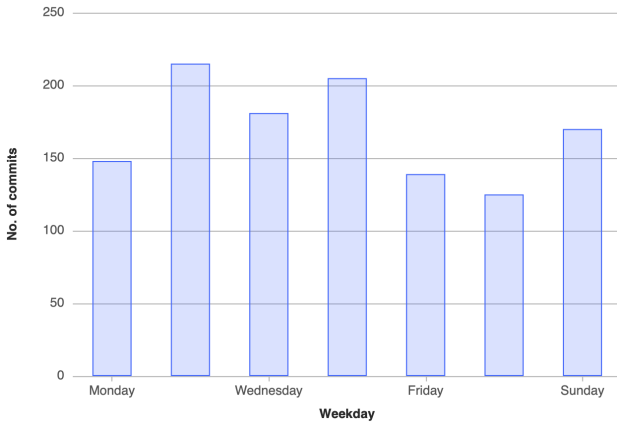


Abbildung 7.6: Commits pro Wochentag (UTC), Zeitraum: 03.05.2021 bis 08.07.2021

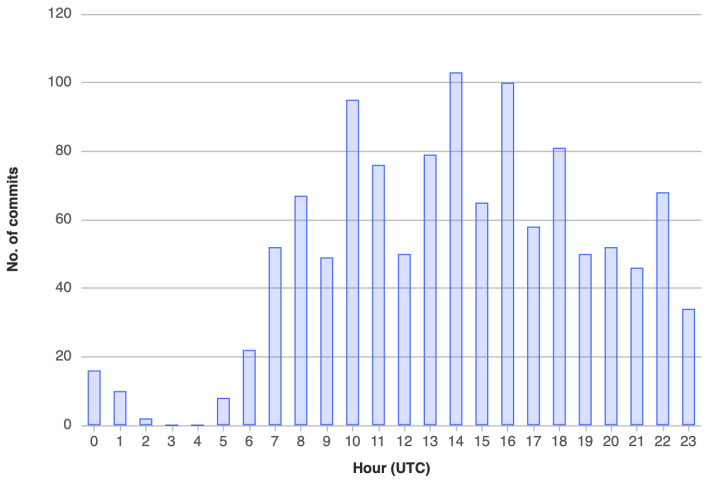


Abbildung 7.7: Commits pro Stunde des Tages (UTC), Zeitraum: 03.05.2021 bis 08.07.2021

Kapitel 8

Auswertung der erfassten Arbeitszeiten

Mithilfe des Zeiterfassungssystems Kimai können Arbeitszeiten der Teammitglieder erfasst werden. Dabei werden nicht nur der Beginn und das Ende der Bearbeitung einer Projektaufgabe von der Software aufgenommen, sondern es wird dieser auch eine passende Aktivitätskategorie und fakultativ ein kurzer Text hinzugefügt. Die verschiedenen Kategorien sind in diesem Projekt:

- Administration (z.B. Aktualisieren des Gantt-Diagramms)
- Entwurf (z.B. Erstellen des Klassendiagramms oder der Aktivitätsdiagramme)
- Dokumentation (z.B. Arbeiten am Review-Dokument)
- Implementierung (z.B. Programmieren einer konkreten Klasse)
- Installation (z.B. Herunterladen von DPDK)
- Meetings (z.B. Montagsmeeting mit dem Betreuer um 18:30 Uhr)
- Präsentationsvorbereitung (z.B. Erstellen der Präsentationsfolien)
- Recherche (z.B. Vergleichen verschiedener Maps durch Reports im Internet)
- Test (z.B. Ausführen von Unit Tests und anderen Tests)

Mithilfe dieses Tools kann somit jedes Teammitglied regelmäßig einen Überblick erhalten, was es wann wie lange für das Software-Projekt gemacht hat.
Am Ende jeder der drei Phasen (Planung- und Entwurf, Implementierung, Validierung) werden die vom Kimai erfassten Daten ausgewertet. Dies ermöglicht, Probleme zu identifizieren und so eventuell später aufkommende Komplikationen vorzubeugen. Falls sich somit in der Auswertung Probleme zeigen, werden Verbesserungsvorschläge und Gegenmaßnahmen entwickelt.
In diesem Kapitel werden die bisher erfassten Zeiten analysiert und in Diagrammen dargestellt. Diese Diagramme wurden automatisch aus den Daten einer Excel-Datei generiert. Diese Daten stimmen mit denen aus dem Kimai überein.

Den drei Phasen wurden folgende Kalenderwochen zugeordnet:

- Planungs- und Enturfsphase: KW 17 - KW 21
- Implementierungsphase: KW 22 - KW 24
- Validierungsphase: KW 25 - KW 28

Anmerkung: Die Kalenderwoche 29 wird nicht ausgewertet, da diese durch die Abgabe des Reviewdokumentes am Mittwoch nicht vollständig ausgewertet werden kann.

Die Phasen werden in den folgenden Kapiteln einzeln ausgewertet und deren Ergebnisse am Ende verglichen.

8.1 Planungs- und Entwurfsphase

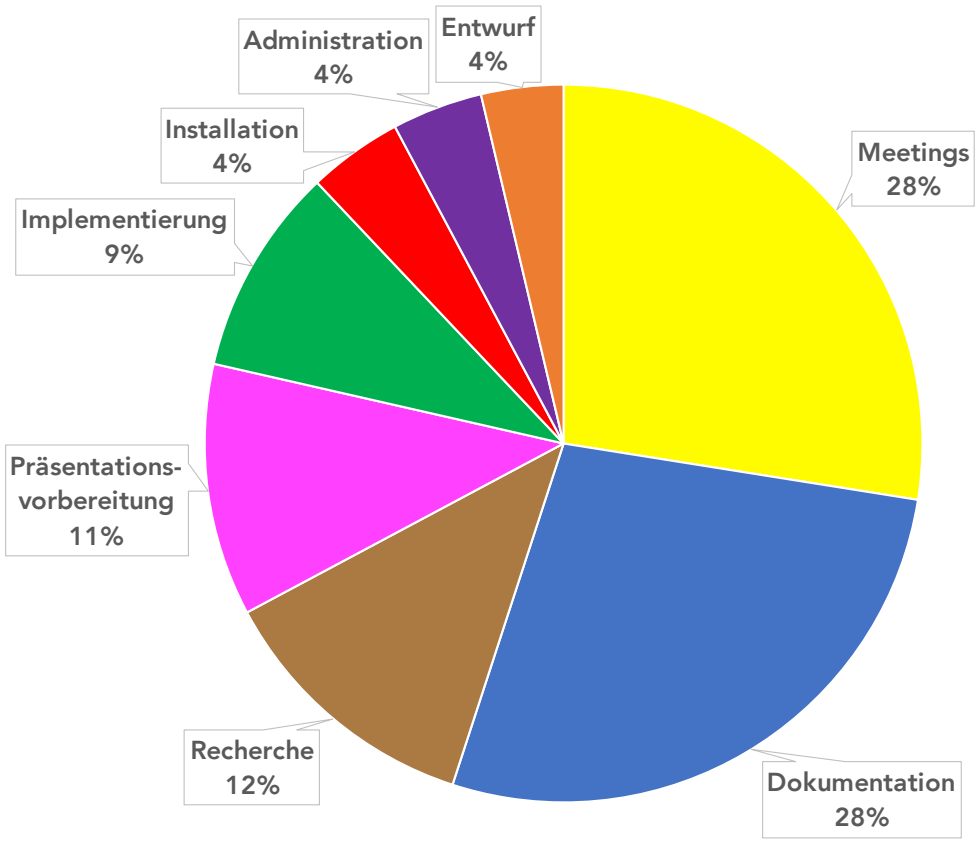


Abbildung 8.1: Planungs- und Entwurfsphase (KW 17-21): Anteile der Aufgabenkategorien an der insgesamt aufgebrachten Zeit

Das Diagramm 8.1 zeigt, dass die Zeit in der Planungs- und Entwurfsphase vorwiegend für **Meetings** aufgewendet wurde. Der Grund dafür könnten die häufigen langen Diskussionen in den Meetings mit allen Teammitgliedern sein. Zudem wurde oft die gemeinsame Arbeit an Aufgaben ebenfalls als Meeting gebucht, auch wenn eine andere Kategorie besser gepasst hätte. Beispielsweise wurde das Klassendiagramm in den wöchentlichen Meetings gemeinsam entworfen. Diese Entwurfsaktivität wurde hier aber unter Meetings und nicht unter Entwurf verbucht. Lösungsideen sind, Meetings themenspezifischer zu buchen und diese nach Möglichkeit in kleineren Gruppen abzuhalten, falls die Themen nicht alle Mitglieder betreffen. Jedoch ist zu beachten, dass hierbei die Konsistenz gewahrt wird. Das bedeutet, dass die „großen" Meetings mit dem gesamten Team weiterhin unter Meetings verbucht werden müssen. Dies wurde sowohl bei einem Meeting

als auch durch einen Eintrag im Wiki allen Teilnehmern des Projekts kommuniziert. Weiterhin könnten weniger bzw. kürzere Meetings gehalten werden und detaillierte Fragen direkt mit den zuständigen Personen geklärt werden. Dies dann entweder in einem eigenen Meeting oder über die Plattform Zulip.

Außerdem wurde mit 26% viel Zeit für die Kategorie **Dokumentation** in Anspruch genommen. Ein möglicher Grund dafür ist, dass auch die Arbeit am Entwurf (z.B. am Klassen- oder Paketdiagramm) und die dazugehörige Entwurfsdokumentation nur unter der Kategorie Dokumentation, nicht aber unter der Kategorie Entwurf gebucht wurde. Daraus ergibt sich, dass das Team in Zukunft detaillierter buchen soll und die Kategorien womöglich angepasst werden muss. Die für die **Recherche** aufgebrachte Zeit ist tendenziell verhältnismäßig und bedarf keiner Änderung.

Es fällt auf, dass für die **Vorbereitung** von Präsentationen mit 11% in dieser Projektphase vergleichsweise viel Zeit aufgewendet wurde. Dies resultiert daraus, dass jedes Teammitglied in der ersten Woche eine Präsentation zu verschiedenen Themen wie DoS-Angriffen oder DPDK gehalten hat und eine Vorlage für Präsentationen erarbeitet werden musste. Im weiteren Verlauf des Projektes sollte dieser Anteil sinken, da dann nur noch eine Präsentation pro Phase gehalten werden muss. Zudem müssen womöglich weniger Grafiken für das Review erstellt werden und das Präsentationsdesign kann wiederverwendet werden.

Für die **Installation** sind 6% der Zeit aufgebracht worden. Auch dieser Wert sollte im Verlauf sinken, weil ein Großteil der Installationen bereits erfolgt sind.

In der Planungs- und Entwurfsphase war der Zeitaufwand für die **Implementierung** sehr gering, was zu Beginn des Projektes allerdings kein Problem darstellt, da hier lediglich ein erster Prototyp mit stark reduziertem Funktionsumfang entwickelt wurde. Zudem kann es daraus resultieren, dass in dieser Phase mit der Implementierung lediglich zwei der acht Teammitglieder beauftragt waren. Es wird erwartet, dass mit Beginn der Implementierungsphase der für die Implementierung erfasste Aufwand sprunghaft ansteigt.

Der geringe Anteil des **Administrationsaufwandes** ist positiv zu bewerten, weil er auf eine effiziente Planung und Verwaltung hinweist.

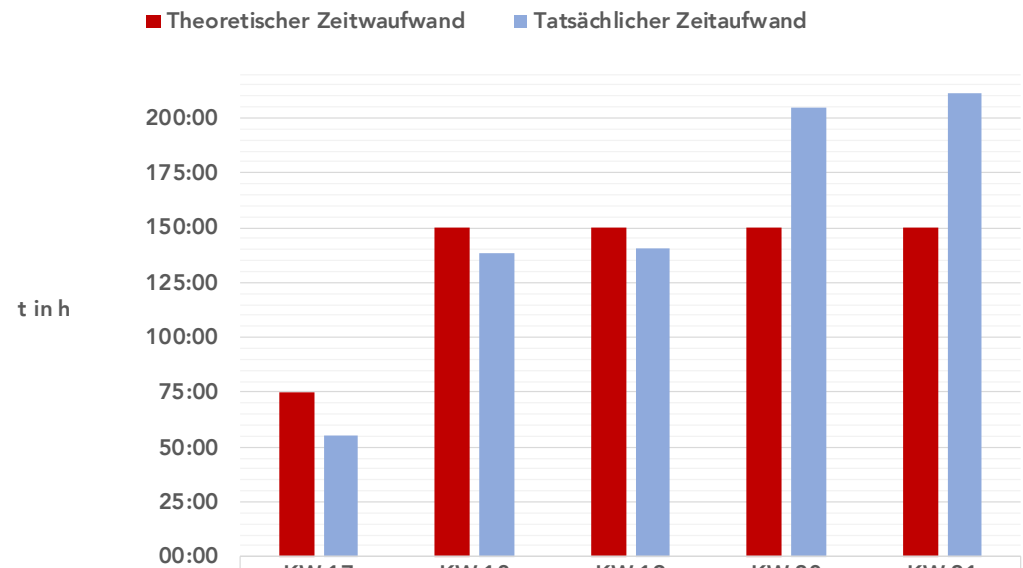
Die 3% der erfassten Zeit, die der **Entwurf** in Anspruch genommen hat, sind zu wenig. Das ist wahrscheinlich darauf zurückzuführen, dass andere Kategorien für Arbeiten am Entwurf beim Buchen verwendet wurden, beispielsweise Dokumentation oder Meetings (Erklärung siehe oben).

Im Balkendiagramm in 8.2 wird dem theoretischen Zeitaufwand pro Woche der tatsächliche Aufwand gegenübergestellt. Die hier angegebenen tatsächlichen Zeiten ergeben sich aus der Addition der wöchentlichen Zeiten aller Teammitglieder. Während Informatiker und Ingenieurinformatiker für das Softwareprojekt acht Leistungspunkte angerechnet bekommen, beträgt diese Punktzahl bei Wirtschaftsinformatiker lediglich sechs. Deshalb wird als Richtwert für die aufzubringende Zeit pro Woche zwischen den Studiengängen unterschieden: Bei Informatikern und Ingenieurinformatikern beträgt dieser Wert 20 Wochenstunden, bei Wirtschaftsinformatikern dagegen nur 15 Wochenstunden. Da das Team aus vier Informatikern, zwei Ingenieurinformatikern und zwei Wirtschaftsinformatikern besteht, beträgt der Soll-Wert pro Woche 150 Stunden.

$$2 \cdot 20h + 4 \cdot 20h + 2 \cdot 15h = 150h$$

In Woche 17 liegt Soll-Wert nicht bei 150 Stunden, sondern bei 75 Stunden, da das Projekt erst am Donnerstag begonnen hat und somit diese Woche kürzer als die anderen war.

In dieser Kalenderwoche liegt der Ist-Wert fast 20 Stunden unter diesen 75 Stunden, nämlich bei 55 Stunden und 9 Minuten. Dieses Defizit ist darin zu begründen, dass in dieser ersten Woche das



	KW 17	KW 18	KW 19	KW 20	KW 21
Theoretischer Zeitwaufwand	75:00	150:00	150:00	150:00	150:00
Tatsächlicher Zeitaufwand	55:09	138:42	140:21	204:32	211:27

Abbildung 8.2: Planungs- und Entwurfsphase (KW 17-21): Vergleich des theoretischen Aufwands mit dem tatsächlichen Aufwand

Zeiterfassungssystem den Teammitgliedern noch nicht zur Verfügung stand. Deshalb mussten die Zeiten in der kommenden Woche nachgetragen werden, was jedoch nicht vom gesamten Team gemacht wurde.

In der Kalenderwoche 18 und der Kalenderwoche 19 liegt der tatsächliche Wert nur noch knapp unter den angestrebten 150 Stunden.

In den beiden letzten Wochen der ersten Phase werden diese 150 Stunden sogar stark übertroffen. Somit lässt sich generell ein positiver Trend ablesen. Es wird jedoch erwartet, dass der in der kommenden Projektphase aufgebrachte Aufwand nicht weiter stark ansteigen wird, da es neben dem Softwareprojekt noch zahlreiche andere Aufgaben für das Studium zu erledigen gibt. Somit ist es für die meisten Studierenden kaum möglich, mehr als 20 bis 30 Stunden für dieses Projekt in Anspruch zu nehmen.

Den Abbildungen 8.3 und 8.4 ist zu entnehmen, dass teilweise große Unterschiede zwischen den Teammitgliedern in Bezug auf die wöchentlich aufgebrachten Stunden für das Softwareprojekt bestehen. [1] Aus dem Diagramm geht hervor, dass in Kalenderwoche 17 noch nicht jeder seine Arbeitszeiten in das Kimai eingetragen hat, da es zu diesem Zeitpunkt Ihnen noch nicht zur Verfügung stand und sie diese hätten nachtragen müssen. Somit liegt bei zwei Personen der Wert in KW17 bei 0. Es ist ein positiver Trend zu sehen, was auch in der Abbildung 8.2 deutlich wurde. Es ist hervorzuheben, dass einzelne Mitglieder weit über 20 Stunden pro Woche für das Softwareprojekt aufwenden. In der Kalenderwoche 20 liegt diese beispielsweise bei einem Teammitglied über 52 Stunden.

[1] Jede Linienfarbe in einem Projektmitglied zugeordnet. Auf die Beschriftung der einzelnen Linien wurde jedoch verzichtet, um die Anonymität der einzelnen Personen zu wahren.

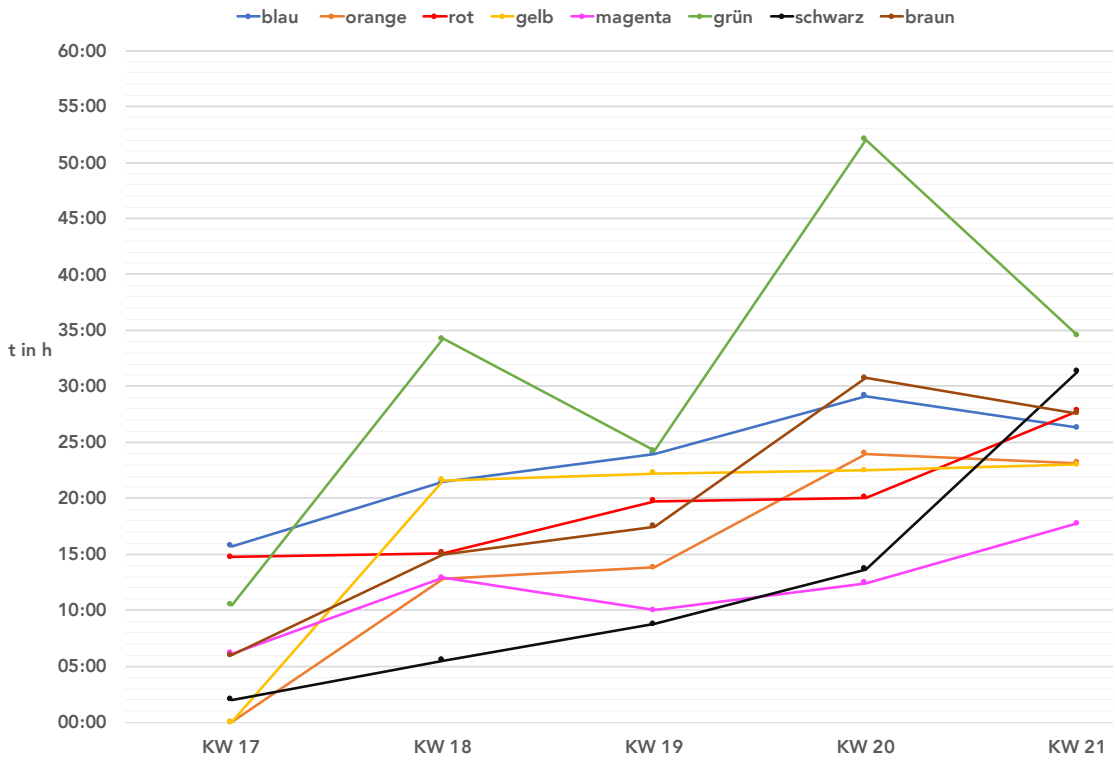


Abbildung 8.3: Planungs- und Entwurfsphase (KW 17-21): Zeitaufwand der einzelnen Teammitglieder

TEAMMITGLIED	KW 17	KW 18	KW 19	KW 20	KW 21	∑
Grün	10h 30min	34h 14min	24h 15min	52h 1min	34h 32min	155h 32min
Blau	15h 45min	21h 29min	24h	29h 8min	26h 19min	116h 41min
Gelb	0h	21h 38min	22h 15min	22h 30min	23h	89h 23min
Rot	14h 44min	15h 8min	19h 44min	20h 3min	27h 47min	97h 26min
Braun	6h	15h	17h 30min	30h 45min	27h 35min	96h 50min
Magenta	6h 10min	12h 53min	10h	12h 25min	17h 44min	59h 12min
Orange	0h	12h 49min	13h 50min	24h	23h 10min	71h 49min
Schwarz	2h	5h 31min	8h 47min	13h 40min	31h 20min	61h 18min
∑	55h 9min	138h 42min	140h 21min	204h 32min	211h 27min	750h 11min

Abbildung 8.4: Planungs- und Entwurfsphase (KW 17-21): Tabelle zum Zeitaufwand der einzelnen Teammitglieder

8.2 Implementierungsphase

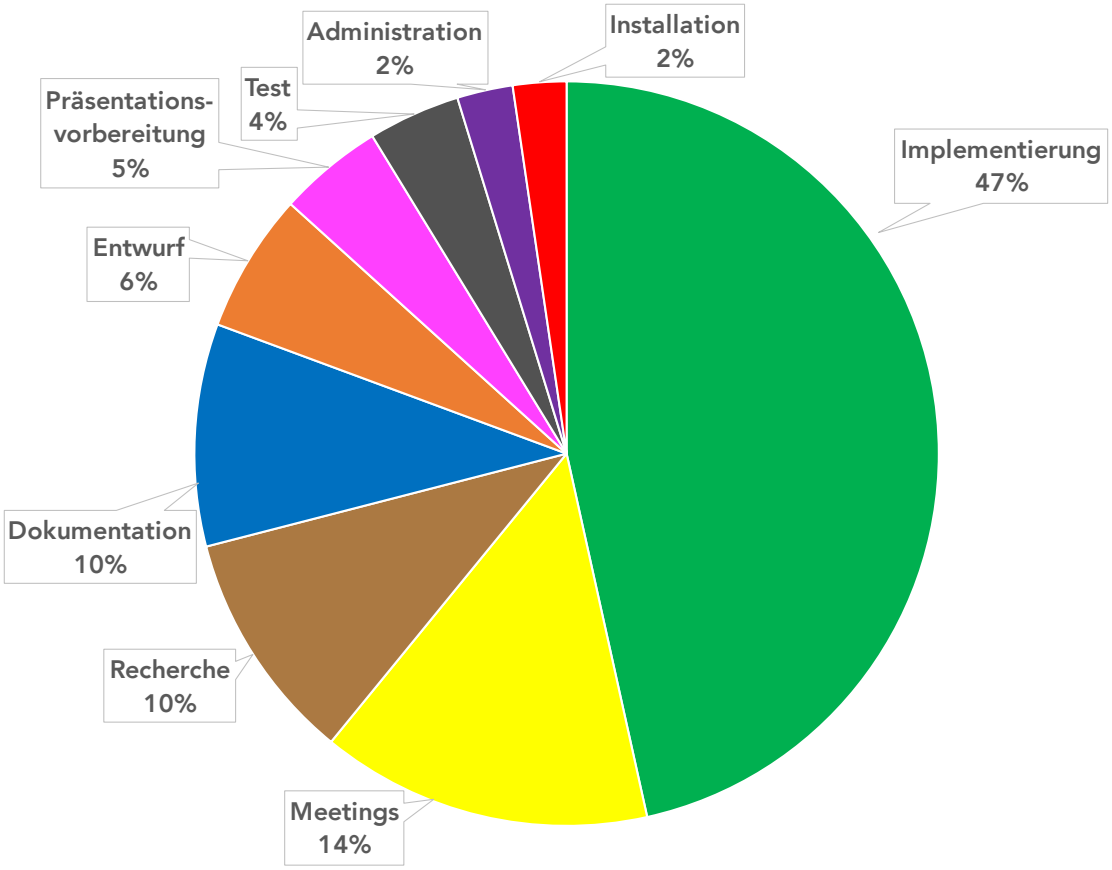


Abbildung 8.5: Implementierungsphase (KW 22-24): Anteile der Aufgabenkategorien an der insgesamt aufgebrachten Zeit

Wie in Abbildung 8.5 zu sehen ist, wurde fast die Hälfte der Zeit dieser Projektphase für die **Implementierung** verwendet. Dieser Anteil ist durchaus angemessen, weil in der Implementierungsphase der größte Teil des Systems programmiert werden soll.

Meetings stellen mit 14% die zweithäufigste Kategorie dar. Diese Größe bedarf nicht zwangsläufig einer Änderung, da die Meetings in dieser Phase lediglich wichtige Themen umfasst haben und so kurz wie möglich gehalten wurden. Außerdem bleibt zu erwähnen, dass in zwei Meetings ein Code-Review durchgeführt wurde, wodurch alle Teammitglieder voneinander lernen konnten und das System insgesamt sowie von anderen entwickelte Komponenten besser verstanden werden konnten.

Die Kategorien **Recherche** und **Dokumentation** nahmen beide ein Zehntel der insgesamt aufgebrachten Zeit in Anspruch und gehen somit mit einem angemessenen Anteil in das Projekt ein.

Die 6%, die dem **Entwurf** zugeordnet werden können, sind relativ wenig. Möglicherweise wurde für diese Kategorie aufgebrachte Zeit auch in andere Kategorien wie Dokumentation oder

Implementierung gebucht.

Positiv zu bewerten ist, dass die restlichen Kategorien, also **Präsentationsvorbereitung**, **Test**, **Administration** und **Installation**, höchstens 5% der Zeit in Anspruch genommen haben. Trotzdem ist es wichtig, dass die den Tests zugeordnete Zeit in der kommenden Validierungsphase sprunghaft ansteigt. In der Implementierungsphase hat es sich dabei vor allem um Unit-Tests gehandelt, die für die gesamte Validierung nicht ausreichen werden.

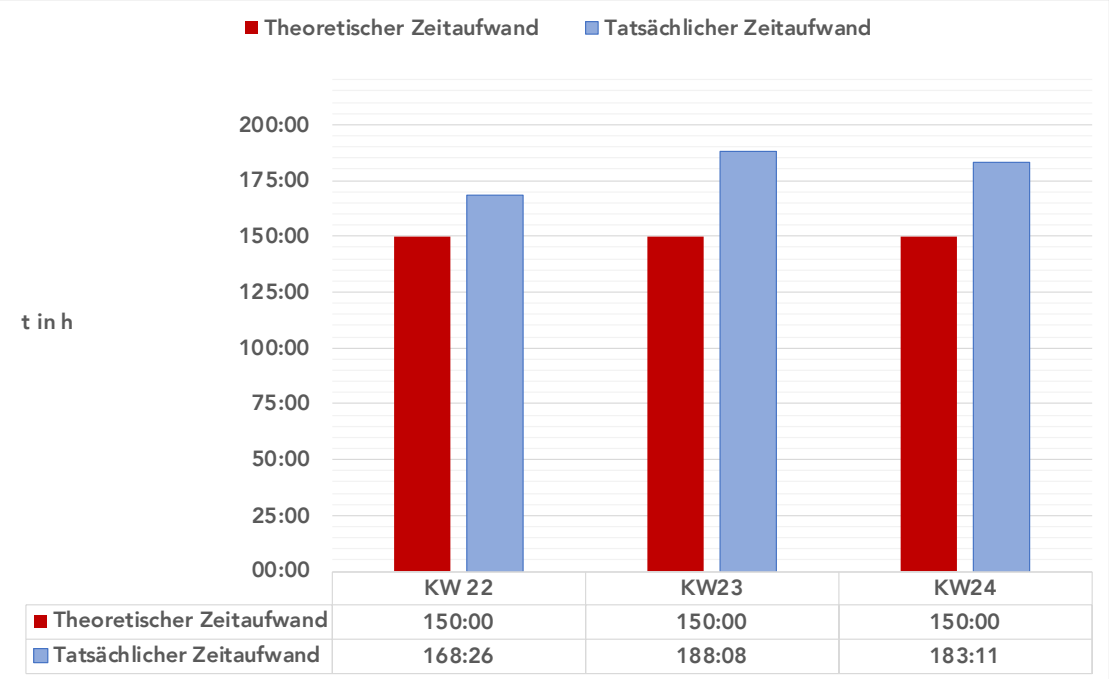


Abbildung 8.6: Implementierungsphase (KW 22-24): Vergleich des theoretischen Aufwands mit dem tatsächlichen Aufwand

Abbildung 8.6 macht deutlich, dass in jeder einzelnen Woche der Implementierungsphase (KW 22 - 24) der tatsächliche Zeitaufwand über dem theoretischen Zeitaufwand lag. Die vorgegebenen 150 Stunden Zeitaufwand pro Woche wurden in der Kalenderwoche 23 sogar um über 38 Stunden überschritten. Wenn man diese 38 Stunden durch die Anzahl der Teammitglieder teilt, die bei acht liegt, wird deutlich, dass in dieser Woche im Durchschnitt jedes Teammitglied 4 Stunden und 45 Minuten mehr Zeit für das Softwareprojekt aufgewendet hat als vorgegeben.

Aus Abbildung 8.7 lässt sich grundsätzlich kein starker Anstieg der Zeit, die jeder einzelne Beteiligte für das Softwareprojekt aufgewendet hat, erkennen. Bei sechs der acht Teammitglieder liegt ihr persönliches Maximum in diesem Zeitraum in KW 23. Hier liegt auch der Punkt, mit dem größten Wert in der Implementierungsphase. Der dazugehörige Wert beträgt 32 Stunden und 30 Minuten (vgl. Abb. 8.8). Bei einigen Teammitglieder kann erkannt werden, dass die empfohlene Zeit von 15 bzw. 20 Stunden in einzelnen Fällen unterschritten wird. Üblicherweise wird auf die gesamte Zeit bezogen der Soll-Wert allerdings erfüllt, was auch in Abbildung 8.6 erkannt werden kann.

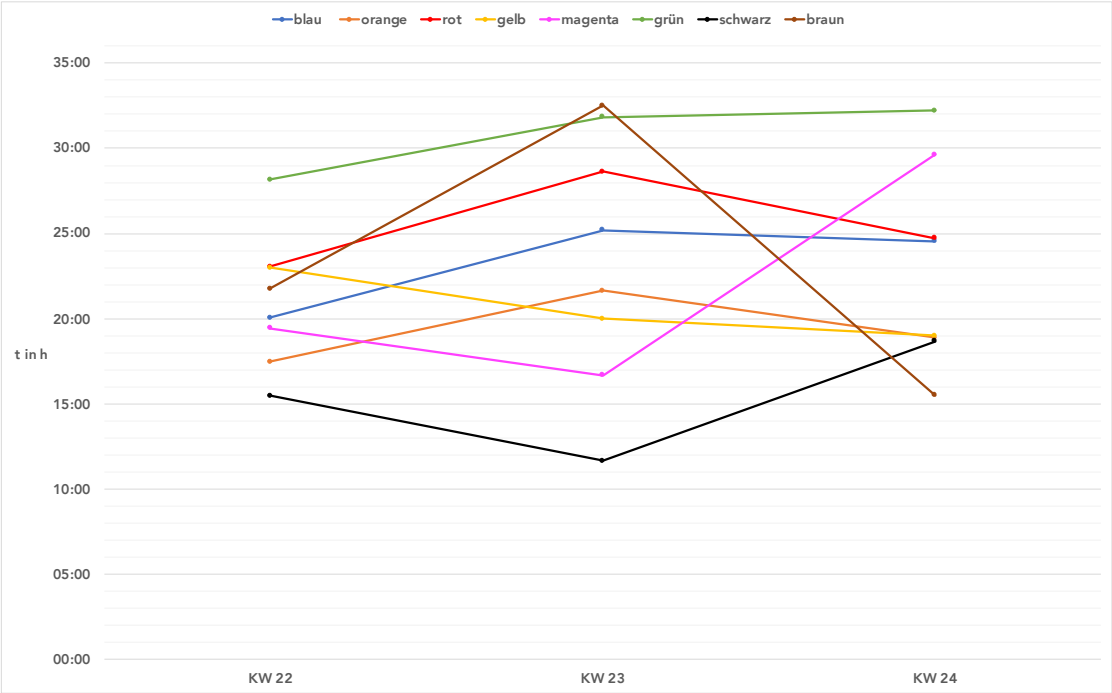


Abbildung 8.7: Implementierungsphase (KW 22-24): Zeitaufwand der einzelnen Teammitglieder

TEAMMITGLIED	KW 22	KW 23	KW 24	∑
Grün	28h 10min	31h 51min	32h 13min	92h 14min
Blau	20h 3min	25h 11min	24h 33min	69h 47min
Gelb	23h	20h	19h	62h
Rot	23h 4min	28h 38min	24h 44min	76h 26min
Braun	21h 45min	32h 30min	15h 30min	69h 45min
Magenta	19h 27min	16h 40min	29h 38min	65h 45min
Orange	17h 28min	21h 38min	18h 52min	57h 58min
Schwarz	15h 29min	11h 40min	18h 41min	45h 50min
∑	168h 26min	188h 8min	180h 41min	539h 45min

Abbildung 8.8: Implementierungsphase (KW 22-24): Tabelle mit den erfassten Zeiten

8.3 Validierungsphase

In der Validierungsphase, also der letzten Phase des Softwareprojekts, wurden wieder Zeiten in ganz anderen Kategorien gebucht im Vergleich zu den vorherigen Phasen. Die Verteilung kann man im Diagramm in Abbildung 4.x sehen.

Während für die **Implementierung** in der zweiten Phase des Projekts noch fast die Hälfte der Zeit aufgebracht wurde, hat sich dieser Wert in der Validierungsphase mit x% auf etwas mehr als ein Viertel reduziert. Das war auch zu erwarten, weil der größte Teil der eigentlichen Programmierung schon in diesem Zeitraum schon erledigt gewesen ist. Dennoch kann festgehalten werden, dass auch über die gesamte Phase noch implementiert werden musste.

Nicht immer klar von der Implementierung abzugrenzen ist die Kategorie **Test**. Diese gehen teilweise ineinander über, d. h. beide Tätigkeiten wurden des öfteren zusammen erledigt. Deshalb kann eine gewisse Ungenauigkeit der beiden Anteile nicht ausgeschlossen werden. Etwas weniger als ein Viertel der Gesamtzeit, genauer gesagt x%, wurde für das Testen gebucht. Vor Beginn der Phase hätte man vermuten können, dass dieser Wert höher ausfällt. Das Nichteintreten dieser Vermutung könnte aus der Überschneidung mit der Kategorie Implementierung und aus dem relativ hohen Zeitaufwand für andere Kategorien resultieren.

Während die **Meetings** in den vergangenen beiden Phasen an erster und zweiter Stelle standen, sind diese in der Validierungsphase erstmals an dritter Stelle zu finden, und zwar mit einem Anteil von x%. Beim Vergleich dieses Wertes mit der ersten Phase fällt auf, dass gegen Ende des Projekts nur noch ca. halb so viel Zeit mit Meetings verbracht wurde wie am Anfang, was positiv zu bewerten ist. Es mussten allerdings auch wesentlich weniger grundlegende Entscheidungen über das Projekt diskutiert und entschieden werden. Das am Montag Abend stattfindende Meeting dauerte fast immer am längsten, unter anderem Aufgrund, weil Martin Backhaus in dieser Zeit Feedback zum geschriebenen Code gab und wichtige Verbesserungen besprochen wurden. Die restlichen wöchentlichen Meetings vielen dann meist wesentlich kürzer aus und wurden meist für kurze Statusabfragen und für die Klärung individueller Fragen genutzt.

x% der Zeit wurde für die **Dokumentation** verwendet. Diese hat in der Validierungsphase etwas weniger Zeit in Anspruch genommen als vermutet, womöglich weil sich wenige Teammitglieder besonders viel mit der Dokumentation beschäftigt haben. Die Mehrheit des Teams hat sich gegen Ende des Projekts tendentiell eher für andere Aufgaben verantwortlich gefühlt, was allerdings kein Problem darstellt.

Die **Präsentationsvorbereitung** muss hier an nächster Stelle angemerkt werden, denn ihr ist ein Wert von x% zuzuordnen. Die Abschlusspräsentation findet in dieser Phase erstmals in Präsenz im Audimax der TU Ilmenau statt und bedarf daher einer anderen Vorbereitung. Die möglichst gut zu erkennende Darstellung der Funktionalität des Systems steht hierbei an erster Stelle. Für diese visuelle Darstellung wurde ein kurzes PyQT-Programm geschrieben.

Die **Administration** nahm x% der Zeit in Anspruch. Eine wichtige administrative Aufgaben dieser Phase war zum Beispiel das Erstellen und Auswerten der durchgeführten Umfrage, deren Ergebnisse im 9. Kapitel dieses Dokuments zu finden sind.

Die verbleibenden Kategorien **Recherche** (x%), **Installation** (x%) und **Entwurf** (x%) wurden in der letzten Projektphase kaum gebucht. Das lässt sich damit begründen, dass es sich bei diesen um typische Aufgaben für den Beginn des Projekts handelt.

Aus der Abbildung 4.x wird ersichtlich, dass das Ziel von 150 Wochenstunden in allen Wochen dieser Projektphase (KW 25-28) erreicht wurde. Es bleibt erneut anzumerken, dass auch während

der ersten drei Tage der KW 29 noch am Softwareprojekt gearbeitet wird. Sollte das Reviewdokument diese Zeiten auch noch enthalten, könnte es erst nach Ende des Projekts fertig gestellt werden. Es muss allerdings noch vor dem letzten Review abgegeben werden. Der erfasste Aufwand überstieg den theoretischen in allen Wochen sogar um über 30 Stunden. Die tatsächlich insgesamt aufgebrachten Zeit kann also durchaus als zufriedenstellend bezeichnet werden.

Die Unterschiede zwischen den einzelnen Teammitgliedern können aus Abbildung 4.x abgelesen werden. Die längste Wochenarbeitsdauer wurde mit 43:31 h in KW 25 erreicht. Ein Teammitglied konnte krankheitsbedingt in KW 26 nur weniger als 11 Stunden mit dem Softwareprojekt beschäftigt sein. Auch in dieser Phase kann wieder erkannt werden, dass der Plan-Wert in Einzelfällen unterschritten wurde. Diese Defizite wurden in den allermeisten Fällen dann allerdings in anderen Wochen aufgeholt.

8.4 Vergleich der Planungs- und Entwurfsphase mit der Implementierungsphase

Dieses Kapitel wurde gegen Ende der Implementierungsphase erstellt.

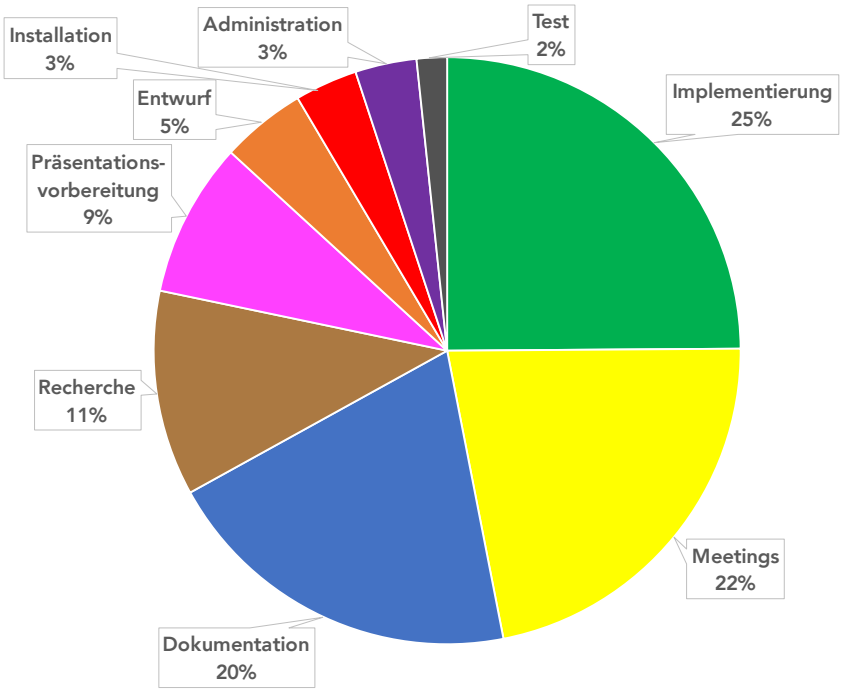


Abbildung 8.9: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Anteile der Aufgabenkategorien an der insgesamt aufgebrachten Zeit

In Abbildung 8.6 wird die Verteilung der Kategorien bezogen auf den gesamten Projektzeitraum, d. h. in diesem Fall die ersten beiden Phasen, dargestellt.

Bei einem Vergleich der Anteile der Kategorien an der insgesamt aufgebrachten Zeit zwischen Planungs- und Entwurfsphase und Implementierungsphase fällt als erstes auf, dass sich die für die **Implementierung** aufgebrachte Zeit vervielfacht hat. Dieser sprunghafte Anstieg wurde

auch prognostiziert, weil die Implementierung in der ersten Phase lediglich einen ersten Prototyp umfasste und der zentrale Bestandteil der jetzt abgeschlossenen Phase ist.

Weiterhin ist der Anteil der **Meetings** von 29 auf 14% gesunken. Da dieser Anteil im ersten Abschnitt des Softwareprojekts viel zu hoch war und schließlich auf einen adäquaten Wert gesenkt werden konnte, kann geschlussfolgert werden, dass die richtigen Maßnahmen getroffen wurden. So wurden Diskussionen, die nicht alle Teammitglieder betroffen haben, von den wöchentlichen Treffen in kleinere Gruppen verlagert und dann in der entsprechenden Kategorie themenbezogen gebucht.

Die **Dokumentation** hat während der Implementierungsphase ebenfalls weniger Zeit in Anspruch genommen als in der Planungs- und Entwurfsphase. Das umfassende Reviewdokument der ersten Phase, bestehend aus dem Pflichtenheft und einer Entwurfsdokumentation, stellte im ersten Monat des Projekts eine der Hauptaufgaben dar, während ihm in dieser Phase lediglich eine mittelhohe Priorität zugeordnet wurde.

Der **Rechercheaufwand** ist leicht gesunken, da sich zu Beginn des Projekts besonders viel Wissen angeeignet werden musste, welches im Laufe des Projekts dann zunehmen angewendet werden musste. Da fast alle Teammitglieder zu Beginn des Projekts recht unerfahren waren, scheinen die 11% der im gesamten Projektzeitraum aufgebrachten Zeit ein geeigneter Anteil zu sein.

Insgesamt wurden 9% der Zeit für die Vorbereitung von **Präsentationen** verwendet, was ebenfalls positiv zu bewerten ist. An dieser Stelle muss angemerkt werden, dass die Zeiterfassung der Implementierungsphase lediglich die Zeiten bis KW 24 umfasst, wodurch die letzten drei Tage vor der Präsentation nicht abgedeckt wurden. In dieser Zeit wird vermutlich noch eine nicht zu vernachlässigende Menge an Zeit für die Vorbereitung des zweiten Reviews investiert werden. Aus diesem Grund und weil jedes Teammitglied in der ersten Woche des Projekts eine Präsentation zu einem bestimmten Thema ausgearbeitet hat, ist der Wert von 11% auf 5% zurückgegangen.

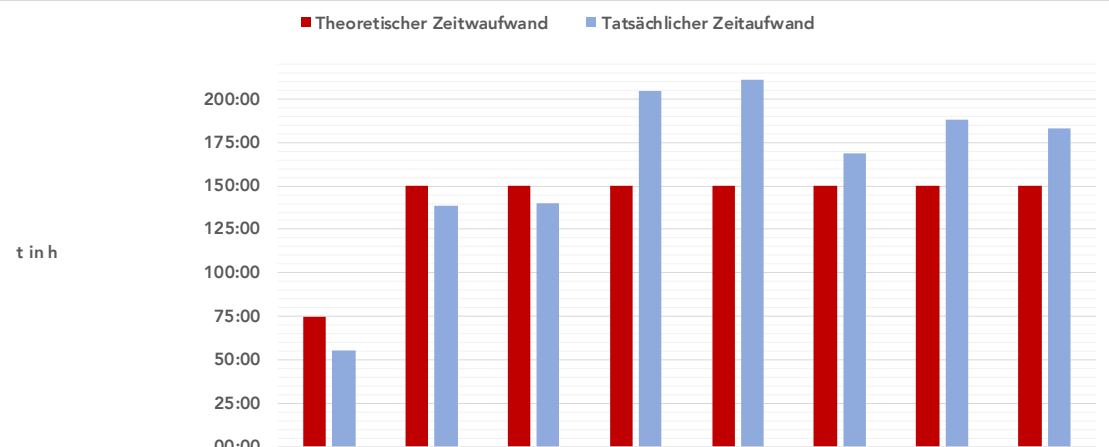
Der **Entwurf** umfasste insgesamt 5% der Zeit und wurde in beiden Phasen tendenziell zu wenig beziehungsweise in anderen, ähnlichen Kategorien erfasst.

Für die **Installation** musste sogar in der ersten Phase nur 5% der Zeit verwendet werden. In der Implementierungsphase sank dieser Anteil auf 2%. Da fast alle benötigten Komponenten bereits installiert sein sollten, lässt sich vermuten, dass der Anteil dieser Kategorie in der letzten Projektphase weiter gegen null geht.

Der **Administrationsaufwand** konnte schon in der ersten Phase des Projekts gering gehalten werden und ging auch in der Implementierungsphase in einem geeigneten Maß ein.

In der zweiten Phase hat sich der **Testaufwand** von 2 auf 4% verdoppelt. Das zeigt, dass die Projektverlauf zunehmende Bedeutung des Testens erkannt wurde. Wichtig ist allerdings, dass beim Testen in der nächsten Phase ein sprunghafter Anstieg ähnlich wie bei der Implementierung zwischen erster und zweiter Phase zu erkennen ist.

Folgendes lässt sich in Abbildung 8.10 erkennen: Während in den ersten drei Wochen des Softwareprojekts (KW 17-18) der tatsächliche Zeitaufwand noch leicht unter dem theoretischen Wert liegt, ändert sich dies in den darauffolgenden Wochen dahingehend, dass der tatsächliche Zeitaufwand den theoretischen (zum Teil stark) übersteigt. Die Feststellung, dass mehr Zeit aufgewendet wird bzw. werden muss als vorgegeben, stimmt mit der Aufwandsschätzung aus dem ersten Reviewdokument überein. Denn wenn die theoretischen Werte aller bisherigen Wochen (KW 17-24) aufaddiert werden, ergibt sich ein Wert von 1289 Stunden und 56 Minuten (vgl. Abb. 8.12).



	KW 17	KW 18	KW 19	KW 20	KW 21	KW 22	KW 23	KW 24
Theoretischer Zeitwaufwand	75:00	150:00	150:00	150:00	150:00	150:00	150:00	150:00
Tatsächlicher Zeitaufwand	55:09	138:42	140:21	204:32	211:27	168:26	188:08	183:11

Abbildung 8.10: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Vergleich des theoretischen Aufwands mit dem tatsächlichen Aufwand

Diesem stehen 1125 Stunden entgegen:

$$1 \cdot 75h + 7 \cdot 150h = 1125h$$

In der ersten Aufwandsschätzung schätzten wir den Aufwand des gesamten Projekts auf 2107 Stunden. Würden wir in den kommenden Wochen (KW 25-29) „lediglich" den theoretische berechnete Zeit brauchen, wären das zu den bisher benötigten 1289 Stunden und 56 Minuten noch 675 Stunden, also insgesamt 1964 Stunden und 56 Minuten. Dieser Wert kommt dem Schätzwert aus der Aufwandsschätzung sehr nahe.

$$4 \cdot 150h + 1 \cdot 75h = 675h$$

Die Abbildung 8.11 zeigt den Zeitaufwand der einzelnen Teammitglieder in den Kalenderwochen 17 bis 24. In den Kalenderwochen 17 bis 21 (Planungs- und Entwurfsphase) ist grundsätzlich ein Anstieg bei den meisten Teammitgliedern zu sehen. In den folgenden Wochen (Implementierungsphase) ist dieser nicht mehr so stark zu erkennen. Auch nimmt in den Kalenderwochen 22 bis 24 der Abstand zwischen den einzelnen Punkten in der Abb. 8.11 ab. Das heißt, dass der Zeitaufwand der einzelnen Teammitglieder sich nicht mehr so stark unterscheidet wie das zum Beispiel in KW 20 der Fall ist. So liegt etwa die Spannweite in KW 20 bei über 39 Stunden, während in KW 22 diese bei unter 13 Stunden liegt.

$$R_{KW20} = 52h\,1min - 12h\,25min = 39h\,36min$$

$$R_{KW22} = 28h\,10min - 15h\,19min = 12h\,51min$$

Der maximale Wert an Wochenstunden pro Person bleibt in KW 20 mit über 52 Stunden (vgl. Abb. 8.12).

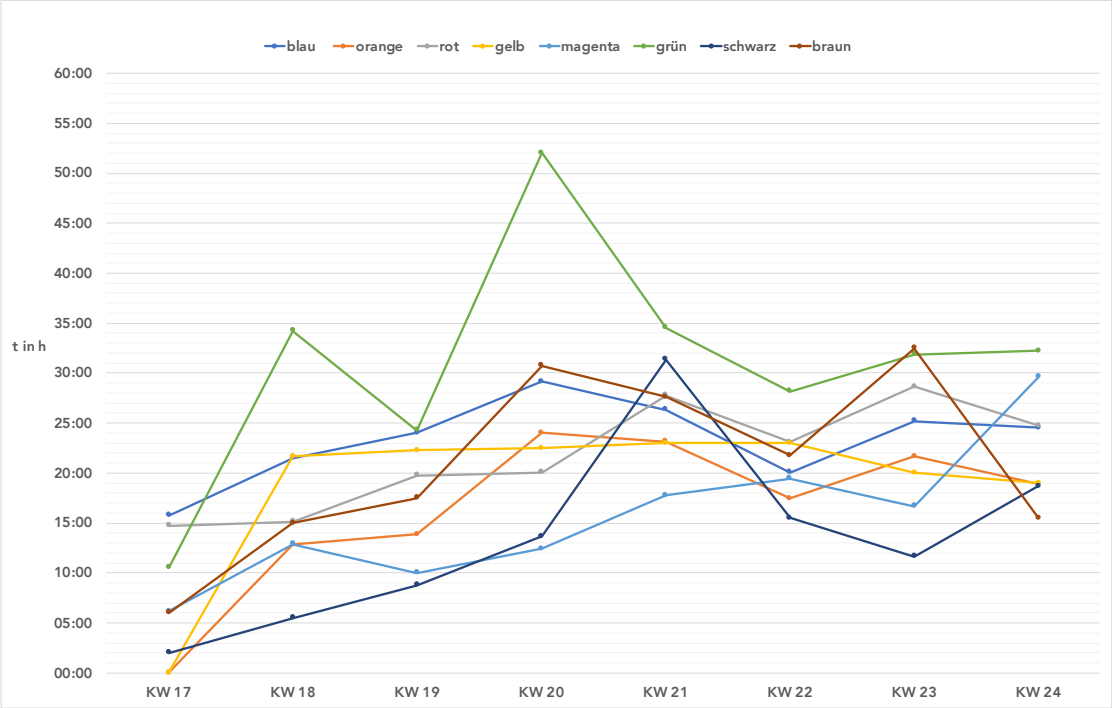


Abbildung 8.11: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Zeitaufwand der einzelnen Teammitglieder

TEAMMITGLIED	KW 17	KW 18	KW 19	KW 20	KW 21	KW 22	KW 23	KW 24	∑
Grün	10h 30min	34h 14min	24h 15min	52h 1min	34h 32min	28h 10min	31h 51min	32h 13min	247h 46min
Blau	15h 45min	21h 29min	24h	29h 8min	26h 19min	20h 3min	25h 11min	24h 33min	186h 28min
Gelb	0h	21h 38min	22h 15min	22h 30min	23h	23h	20h	19h	151h 23min
Rot	14h 44min	15h 8min	19h 44min	20h 3min	27h 47min	23h 4min	28h 38min	24h 44min	173h 52min
Braun	6h	15h	17h 30min	30h 45min	27h 35min	21h 45min	32h 30min	15h 30min	166h 35min
Magenta	6h 10min	12h 53min	10h	12h 25min	17h 44min	19h 27min	16h 40min	29h 38min	124h 57min
Orange	0h	12h 49min	13h 50min	24h	23h 10min	17h 28min	21h 38min	18h 52min	131h 47min
Schwarz	2h	5h 31min	8h 47min	13h 40min	31h 20min	15h 29min	11h 40min	18h 41min	107h 8min
∑	55h 9min	138h 42min	140h 21min	204h 32min	211h 27min	168h 26min	188h 8min	180h 41min	1289h 56min

Abbildung 8.12: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Tabelle mit den erfassten Zeiten

8.4.1 Vergleich aller drei Phasen

Das Kreisdiagramm in Abbildung 4.x findet man die Anteile aller Kategorien an der Gesamtzeit über den gesamten Zeitraum des Projekts hinweg, wieder mit Ausnahme der letzten drei Tage.

Im Softwareprojekt wurde am meisten an der **Implementierung** gearbeitet. Ein Viertel der insgesamt aufgewendeten Zeit (x%) wurde im Zeiterfassungssystem Kimai in dieser Kategorie gebucht. Während in der ersten Phase weniger als 10% der Zeit implementiert wurde, wurde wie vom Fachgebiet System- und Software-Engineering vorgesehen in der Implementierungsphase am meisten Zeit dafür verwendet. In der Validierungsphase sank dieser Wert wieder, allerdings nicht so sehr, wie ursprünglich erhofft. Aufgrund der Komplexität des Projekts stellt es allerdings kein Problem dar, dass die Implementierung nicht in der jeweiligen Phase abgeschlossen wurde. Mit diesem Wert können die Teammitglieder also in durchaus zufrieden sein.

Ziemlich genau ein Fünftel der gesamten Zeit wurde für die **Meetings** gebucht, x% um genau zu sein. Nach den anfänglichen Problemen konnte dieser Anteil in der Implementierungsphase gesenkt werden. Das Team hat es geschafft, diesen Trend auch in der letzten Phase des Projekts fortführen zu können. Entscheidend dafür ist vermutlich, dass die richtigen Maßnahmen getroffen und auch umgesetzt werden. Detailliertere Infos zu diesem Thema können auch dem vorherigen, gegen Ende der Implementierungsphase erstellten Kapitel entnommen werden.

Knapp danach kommt bei dieser absteigenden Sortierung die **Dokumentation** mit x%. Es kann noch hinzugefügt werden, dass immer gegen Ende einer Phase mehr Zeit in dieser Kategorie gebucht wurde als am Anfang, was wahrscheinlich aus der vermehrten Arbeit am Reviewdokument resultiert.

Die **Präsentationsvorbereitung** hat insgesamt x% der Zeit in Anspruch genommen. Wie bereits in den vorhergehenden Kapiteln beschrieben wurde, war der Anteil dieser Kategorie aufgrund der zum Thema hinführenden Kurzvorträge zu Beginn des Projekts am höchsten. Insgesamt kann hiermit festgehalten werden, dass auch die Anteile dieser Kategorie durchaus zufriedenstellend sind.

Auch der Anteil der **Recherche** ist mit x% vollkommen in Ordnung.

x% der insgesamt aufgebrachten Zeit wurde für das **Testen** genutzt. Vor allem gegen Ende des Projekts hätte womöglich noch etwas mehr getestet werden können. Es muss jedoch auch an dieser Stelle angemerkt werden, dass einige für das Testen aufgewendete Zeit in der Kategorie Implementierung gebucht worden ist.

Der geringe **Administrationsaufwand** von x% zeugt von einer hohen Effizienz bei administrativen Aufgaben über das gesamte Projekt hinweg.

Der ähnlich niedrige Wert für den **Entwurf** wurde in anderen Kapiteln schon ausführlich genug behandelt. Es ist definitiv angemessen, dass in der letzten Projektphase nicht mehr viel Zeit in dieser Kategorie gebucht wurde.

Wie gegen Ende der letzten Phase prognostiziert, ist der Wert für die **Installation** noch weiter gesunken. Dieser Trend ist durchaus positiv zu bewerten.

Zu den in den vorherigen Kapiteln zu findenden Überlegungen zum Vergleich des theoretischen Aufwands mit dem tatsächlichen lässt sich nicht mehr sehr viel hinzufügen. Nach ziemlich geringem tatsächlichen Aufwand zu Beginn überstieg das tatsächliche Pensum das mindeste in jeder Woche. Womöglich ist einigen Studierenden zu Beginn des Projekts nicht wirklich bewusst gewesen, wie viel Arbeit dieses wirklich erfordert. Das hat sich im Laufe des Projekts verbessert. Zu Beginn wurde allerdings auch noch nicht die gesamte aufgebrachte Zeit im Kimai-System gebucht.

In der Abbildung 4.x kann man die Verteilung der Gesamtzeit auf die einzelnen Teammitglieder erkennen. Das Maximum über das gesamte Projekt hinweg wurde nach wie vor in der Planungs-

und Entwurfsphase aufgestellt. Es lässt sich festhalten, dass sich das wöchentlichen Arbeitspensum mit Verlauf des Projekts etwas stabilisiert hat und gegen Ende weniger schwankt. Auch fällt auf, dass einzelne Teammitglieder dauerhaft mehr am Projekt gearbeitet haben als andere. Insgesamt kann man mit den aufgebrachten Zeiten der Studierenden sehr zufrieden sein. Weitere Ausarbeitungen zum Aufwand können im folgenden Kapitel gefunden werden.

8.5 Validierung der Aufwandsschätzung der Planungs- und Entwurfsphase

Im Abschnitt „Aufwandsschätzung nach dem COCOMO II aus der Planungs- und Entwurfsphase" wird zunächst das Aufwandsschätzverfahren COCOMO II kurz vorgestellt. Die dort erhaltenen Ergebnisse werden im Abschnitt „Vergleich der Ergebnisse aus der Aufwandsschätzung mit dem tatsächlichen Aufwand" mit dem tatsächlichen Aufwand verglichen. Dieser IST-Aufwand berechnet sich aus den von den Teammitgliedern in die Zeiterfassungs-Software Kimai eingetragenen Zeiten.

8.5.1 Aufwandsschätzung nach dem COCOMO II aus der Planungs- und Entwurfsphase

Bei COCOMO II (COnstrucitve COst MOdel), welches bereits 1981 durch den Softwareingenieur Barry W. Boehm entwickelt wurde, handelt sich es um ein algorithmisches Modell zur Aufwandsschätzung von Software. In diesem Modell werden zahlreiche Einflussfaktoren wie Quantität, Qualität oder Produktivität berücksichtigt. Zudem besteht COCOMO II aus drei Teilmodellen, welche sich unter anderem in den Skalenfaktoren oder den Modellkonstanten unterscheiden. Im Folgenden wird sich auf *„The Early Design Model"* (Die frühe Entwicklungsstufe) bezogen, da diese Stufe stark zum derzeitigen Projektstand passt. Auf dieser Stufe liegen schon sowohl die Anforderungen als auch ein erster Grobentwurf vor.

Das Modell baut im Wesentlichen auf folgender Formel auf:

$$PM = A \cdot \text{Größe}^{E} \cdot M \tag{8.1}$$

Für den Koeffizienten A wird der Erfahrungswert 2,5 angenommen. Die Größe wird in KLSLOC (kilo source lines of code) angegeben. Sie beträgt hier 3,6. Als Referenzprojekt dient POSEIDON [11]. In diesem Projekt wurde eine DDoS-Abwehr durch ungefähr 3600 C/C++-Codezeilen implementiert. Im Unterschied zu der hier zu entwickelnden Software wurde jedoch ein komplettes Endprodukt entwickelt und kein Prototyp, wie es in dem vorliegenden Projekt der Fall ist. Um den steigenden Aufwand bei wachsender Projektgröße zu berücksichtigen, wird der Exponent E, dessen Wert zwischen 1,01 und 1,26 liegt, verwendet. Durch die Bewertung unterschiedlicher Skalierungsfaktoren wird E berechnet.

Faktor	**Punkte**	**Bemerkungen**
Neuartigkeit	2	Bei einzelnen Teammitgliedern liegt noch geringe Erfahrung mit dieser Art von Projekten vor. Dies ist oftmals begründet durch das junge Alter. Jedoch gibt es online schon Lösungen zu ähnlichen Problemen vor, an denen sich orientierenwerden kann.

Faktor	**Punkte**	**Bemerkungen**
Entwicklungsflexibilität	2	Zwar liegen einige Vorgaben zum Ablauf des SW-Projektes vor (z.B. die Einteilung in drei Hauptphasen), jedoch erlaubt das Vorgehensmodell Unified Process noch eine gewisse Flexibilität im Entwicklungsprozess.
Architektur/ Risikoauflösung	3	Die Risiken wurden rechtzeitig identifiziert und Maßnahmen überlegt. Jedoch könnten aufgrund der geringen Erfahrung noch Risiken unentdeckt geblieben sein.
Teamzusammenhalt	1	Die Vertrautheit und Zusammenarbeit im Team ist optimal.
Ausgereiftheit des Prozesses	5	Der Prozess ist noch wenig ausgereift.

E lässt sich nun berechnen, indem auf den fixen Wert 1,01 ein Hundertstel von der Summe der Punkte addiert wird.

$$E = 1,01 + \frac{2+2+3+1+5}{100} = 1,01 + 0,13 = 1,14$$

M ergibt sich durch die Multiplikation folgender Projekt- und Prozessfaktoren:

- RCPX: Product Reliability and Complexity
- RUSE: Developed for Resuability
- PDIF: Platform Difficulty
- PERS: Personnel Capability
- PREX: Personnel Experience
- FCIL: Facilities
- SCED: Required Developement Schedule

Mithilfe der unteren Skala werden die einzelnen Projekt- und Prozessfaktoren bewertet. In der Farbe Rot sind diejenigen Faktoren gekennzeichnet, die auf das vorliegende Projekt am besten zutreffen.

	- - -	- -	-	~	+	++	+++
RCPX	0,49	0,60	0,83	1	1,33	1,91	2,72
RUSE			0,95	1	1,07	1,15	1,24
PDIF			0,87	1	1,29	1,81	2,61
PERS	2,12	1,62	1,26	1	0,83	0,63	0,50
PREX	1,59	1,33	1,22	1	0,87	0,74	0,63
FCIL	1,43	1,30	1,10	1	0,87	0,73	0,62
SCED		1,43	1,14	1	1	1	n/a

Nun lässt sich der Multiplikator M berechnen:

$$\begin{aligned} M &= PERS \cdot RCPX \cdot RUSE \cdot PDIF \cdot PREX \cdot FCIL \cdot SCED \\ &= 1,33 \cdot 0,95 \cdot 0,87 \cdot 1 \cdot 1,33 \cdot 1 \cdot 1 \approx 1,22 \end{aligned}$$

Jetzt sind alle Werte gegeben, um mithilfe der Formel 8.1 den Aufwand in Personenmonate zu schätzen.

$$PM = A \cdot \text{Größe}^{E} \cdot M = 2,5 \cdot 3,6^{1,14} \cdot 1,22 \approx 13,17$$

Nun kann der Aufwand in Personenmonat in Personenstunden umgerechnet werden die Folgende:

$$PS = 13,17 \cdot 160h = 2107h$$

Das Team besteht aus zwei Wirtschaftsinformatikern, die jede Woche 15 Stunden für das Softwareprojekt aufwenden sollen, sowie sechs Informatiker und Ingenieurinformatiker, deren Wochenstundenanzahl bei 20 Stunden liegt. Das Softwareprojekt soll innerhalb von 12 Wochen beendet werden.

Somit ist die zur Verfügung stehende Zeit:

$$(2 \cdot 15h + 6 \cdot 20h) \cdot 12 = 1800h$$

Interpretation der Ergebnisse:

Der Wert von 2107 Stunden liegt ca. 300 Stunden über dem angestrebten Wert von 1800 Stunden. Es kann unterschiedliche Gründe für diese Abweichung geben.

Ein Grund dafür könnte sein, dass im zu entwickelnden System kein schlüsselfertiges Produkt entwickeln wird, sondern vielmehr ein Prototyp. Somit könnte der Schätzwert von 3600 Codezeilen zu hoch gegriffen sein.

Durch die Komplexität des Projektes kann es zudem vorkommen, dass die 15 beziehungsweise 20 Wochenstunden nicht immer ausreichend sind und somit mehr Arbeitszeit aufgewendet werden muss. Zudem muss keine Zeit zur Berücksichtigung von Support vorgesehen werden, da es nach den 12 Wochen als abgeschlossen angesehen werden kann.

Bereits kleine Änderungen an den Projekt- und Prozessfaktoren haben große Auswirkungen. Falls bereits einer dieser oben aufgeführten Faktoren ungenau geschätzt wurde, kann das Ergebnis verfälscht sein. Abschließend kann noch angemerkt werden, dass es sich um eine Schätzung handelt. Schätzungen sind (fast) immer mit Ungenauigkeiten verbunden.

8.5.2 Vergleich der Ergebnisse aus der Aufwandsschätzung mit dem tatsächlichen Aufwand

Um die Ergebnisse der Aufwandsschätzung mit dem tatsächlichen (zeitlichen) Aufwand zu vergleichen, werden zunächst alle eingetragenen Werte aus der Zeiterfassungssoftware addiert. Da diese Auswertung bereits in KW 28 stattfand, stehen hier nur die genauen Werte von KW 17 bis KW 27 zur Verfügung. Der Zeitaufwand für die Wochen KW 28 und KW 29 wird aus den Vergangenheitswerten prognostiziert.

Kalenderwoche	**Zeitaufwand des Teams in [h:min]**
KW 17	55:09
KW 18	138:42
KW 19	140:21
KW 20	204:32
KW 21	211:27
KW 22	168:26
KW 23	188:08
KW 24	183:11
KW 25	188:23
KW 26	184:02
KW 27	213:22
$\sum$	1875:43

Um den Aufwand für KW 28 und KW 29 zu schätzen, wird der Mittelwert des Zeitaufwandes der Kalenderwochen 18 bis 27 ermittelt. KW 17 fließt nicht in die Rechnung mit einem. Der Grund liegt zum einen darin, dass in dieser Woche das Projekt erst am Donnerstag gestartet war und somit die Woche verkürzt war. Zum anderen stand in dieser Woche das Tool Kimai noch nicht zur Verfügung, wodurch die Zeiten von den Projektmitgliedern in der darauffolgenden Woche nachgetragen werden mussten. Das hatten jedoch nicht alle gemacht.

Die Variable x steht im Folgenden für die Kalenderwoche, die Variable y für die Zeit. Die Variable y wurde auf zwei Nachkommastellen genau berechnet.

x	**y**
17	55,15
18	138,70
19	140,35
20	204,53
21	211,45
22	168,43
23	188,13
24	183,18
25	188,38
26	184,03
27	213,22
$\sum$	1875,72
-17	55,15

x	y
$\sum$	1820,57

Der (gerundete) Mittelwert berechnet sich wie folgt:

$$\bar{y} = y_{28} = \frac{1}{n}\sum_{i=1}^{n} y_i = \frac{1}{10} \cdot (138,70 + 140,35 + 204,53 + 211,45 + 168,43 + 188,13 + 183,18 + 188,38 + 184,03 + 213,22) = \frac{1820,57}{10} \approx 182$$

Somit wird ein Wert von 182 h für die Kalenderwoche 28 prognostiziert.

$$y_{29} = \frac{3}{7} \cdot \bar{y} = \frac{3}{7} \cdot 182 = 78$$

Für KW 29 ergibt sich ein geschätzter Wert von 78h.

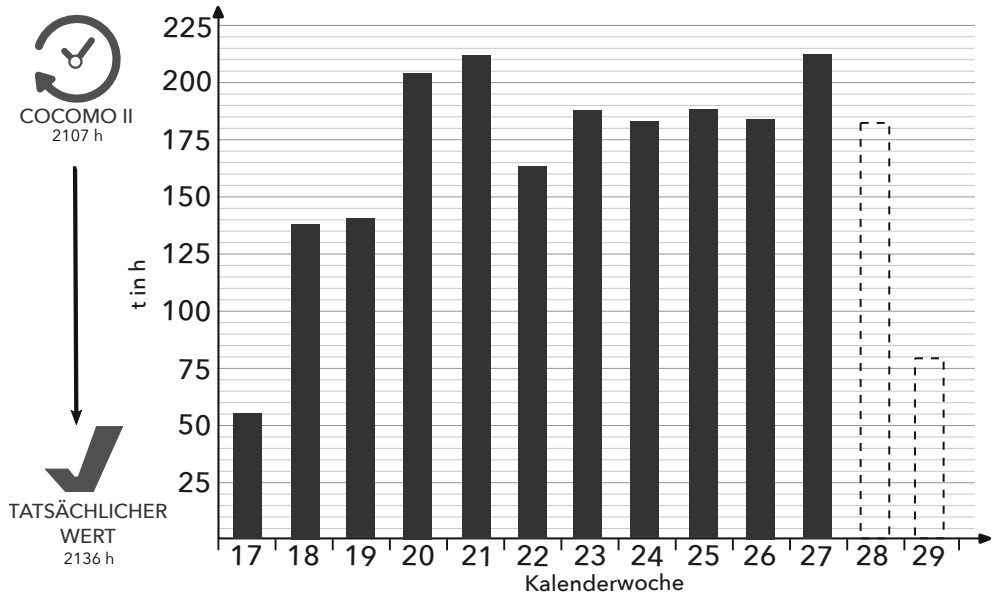


Abbildung 8.13: Tatsächlich aufgebrachte Zeit von KW 17 bis KW 27 und geschätzte Werte für KW 28 und KW 29

Diese beiden geschätzten Werte werden nun auf die 1875h 43min aufaddiert. Das Ergebnis ist 2135h 43 min. Die Werte aller Kalenderwochen sind in Abb. **??** visualisert.

Wenn ein Vergleich zwischen den 2135h 43 min und dem Ergebnis der Aufwandsschätzung aus COCOMO II (2107h) gezogen werden soll, zeigt sich, dass sich diese Ergebnisse nur minimal unterscheiden.

8.6 Vergleich: Vorgehensmodell

Gleich zu Beginn des Projekts wurde sich für den Unified Process als Vorgehensmodell entschieden.

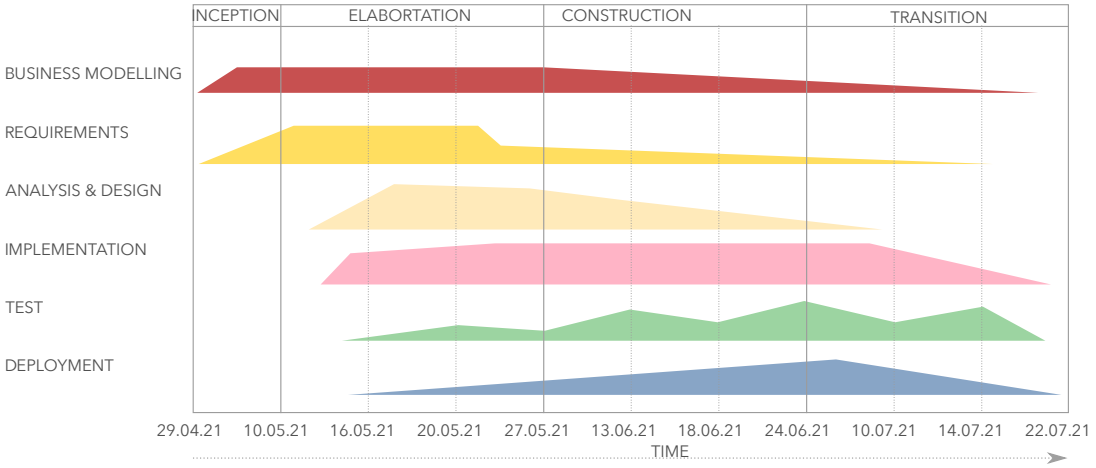


Abbildung 8.14: Idealisierte Abbildung zum Vorgehensmodell Unified Process aus der Planungs- und Entwurfsphase

In Abb. 8.14 sehen ist ein idealisierter Verlauf der Kernprozesse des Projekts abgebildet. In den vier Phasen des Projekts (Inception, Elaboration, Construction, Transition) laufen verschiedene dieser Kernprozesse parallel ab.

Die Kernprozesse können wie folgt ins Deutsche übersetzt werden: Business Modelling $\widehat{=}$ Geschäftsprozessmodellierung, Requirements $\widehat{=}$ Anforderungsanalyse, Analysis and Design $\widehat{=}$ Analyse und Entwurf, Implementation $\widehat{=}$ Implementierung, Developement $\widehat{=}$ Auslieferung.

Da im Projekt die Aktivitäten, die im Kimai verbucht wurden, nicht deckungsgleich mit den obigen Kernprozessen sind, muss eine Zuordnung vorgenommen werden:

Aktivitäten im Kimai	**Entsprechender Kernprozess**
Entwurf	Analysis & Design, Business Modelling
Implementierung	Implementation
Test	Test

Die anderen Aktivitäten im Kimai (z.B. Meeting, Präsentationsvorbereitung, Dokumentation) haben keine Entsprechung in den Kernprozessen.

Hier kommt noch was, Auswertung erst in KW29 mgl

Kapitel 9

Auswertung des Projekts

Während der letzten Phase des Softwareprojektes wurden zwei anonyme Umfragen mithilfe des Umfragetools LimeSurvey erstellt, durchgeführt und ausgewertet. Deren Ergebnisse werden in diesem Kapitel aufgezeigt und näher erläutert.

9.1 Kritische Bewertung des Projekterfolgs

Das folgende Teilkapitel bezieht sich auf die Bewertung des Projekterfolgs. Die weiteren Teilkapitel sind dem Vorgehen und den sonstigen Ergebnissen der Umfrage gewidmet.

9.1.1 Einhaltung der in der Planungs- und Entwurfsphase beschlossenen Werte

Zu Beginn des Projekts wurde sich auf acht (agile) Werte geeinigt (vgl. Abb 9.1), die unter anderem im Gitlab-Wiki und im ersten Reviewdokument festgehalten wurden. Diese Werte sollten die bestmögliche Zusammenarbeit im Team gewährleisten.

Im Folgenden werden die gemeinsam beschlossenen Werte kurz erklärt:

Jedes Teammitglied zeigt den anderen Studierenden **Respekt** und schätzt diese. Außerdem soll jedem Verständnis gegenüberbracht werden, zum Beispiel wenn eine Person Problem hat. Zudem soll auf Schwächere Rücksicht genommen werden.

Offenheit bedeutet hier, dass neue Informationen und Erfahrungen bewusst aufgenommen und nicht vorschnell als unwichtig bewertet werden. Jeder darf seine Meinung frei äußern, um Missverständnissen und Auseinandersetzungen vorzubeugen. Ebenso soll Transparenz herrschen und Problemen soll sofort auf den Grund gegangen werden.

Der Wert **Verpflichtung** kann so interpretiert werden, dass alle Teammitglieder sich der Projektaufgabe verbunden fühlen sollen.

Außerdem sind **Toleranz** und Vielfalt zentrale Werte. Personen mit anderen Sichtweisen werden nicht etwa als Konkurrenten gesehen, sondern vielmehr als Bereicherung für das Team.

Der **Fokus** liegt im Projekt vor allem auf den Aufgaben und den gemeinsamen Zielen. Verschwenden von Zeit und Kapazitäten sowie Ablenkungen sollen vermieden werden. Zudem hat

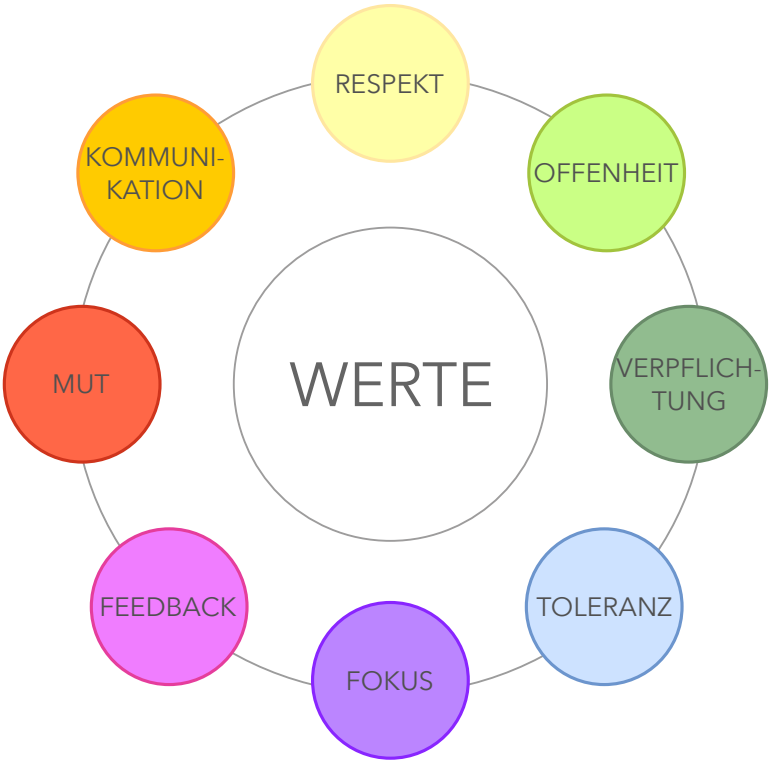


Abbildung 9.1: Die in der Planungs- und Entwurfsphase beschlossenen Werte

die Fertigstellung einer bereits begonnen Aufgabe Vorrang gegenüber dem Start einer neuen Aufgabe.

Indem in den Meetings regelmäßig der derzeitige Stand aufgezeigt wird, kann sich jedes Teammitglied individuelles **Feedback** von den anderen Beteiligten holen.

Zusätzlich soll jedes Teammitglied **Mut** aufweisen, indem es zum Beispiel neue Aufgaben übernimmt, auch wenn sie zuerst als schwer machbar und komplex erscheinen. Außerdem erfordert es Mut zu sagen, wenn Hilfe benötigt wird, oder auch mitzuteilen, wenn eine Aufgabe länger als zuvor erwartet dauert. Auch das Ausprobieren neuer Lösungswege fällt unter den Wert Mut.

Zudem ist die tägliche **Kommunikation** mit den Teammitgliedern für eine gute Zusammenarbeit erforderlich. So sollte jeder mehrmals täglich nach neuen Zulip-Nachrichten schauen und dort auf Fragen anderer antworten. Außerdem wurde gemeinsam beschlossen, dass bei Änderungen an Dokumenten andere benachrichtigt werden.

In der druchgeführten Umfrage sollte von den Teilnehmenden bewertet werden, wie stark die einzelnen Werte ihrem Ermessen nach während des Projekts beachtet wurden. Die Zahl 1 stand hierbei für „sehr wenig”, 2 für „wenig”, 3 für „teils-teils”, 4 für „stark” und die Zahl 5 für „sehr stark”.

Das Ergebnis der Umfrage zeigt, dass nach Meinung der Teammitglieder alle Werte stark bzw. sehr stark beachtet wurden. Die geringste Klassifizierung hat die **Verpflichtung** mit einem Wert von 3,857. Jedoch ist dieser Wert immer noch gut und sehr nahe an „stark”. Vor allem

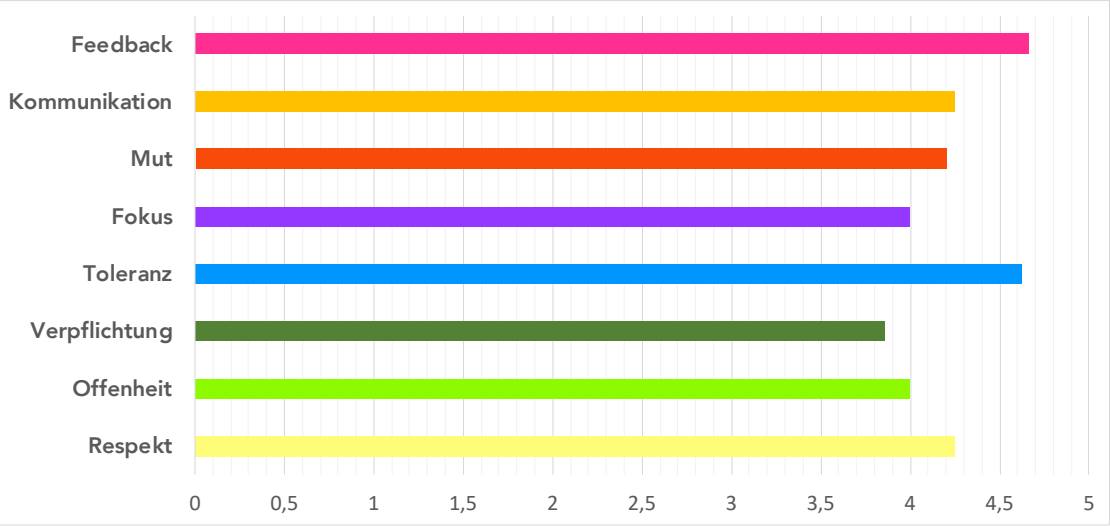


Abbildung 9.2: Klassifizierung der Einhaltung der Werte

der Wert **Toleranz** hat mit einer durchschnittlichen Klassifizierung von 4,625 einen sehr hohen Wert erhalten. Dies zeigt, dass die Teammitglieder ihre Kommilitonen in der Gruppe nicht als Konkurrenten sahen, sondern vielmehr als Bereicherung für das gesamte Team.

9.1.2 Zufriedenheit des Teams mit dem bisherigen Projekterfolg

Circa einanderhalb Wochen vor Projektabgabe wurden die Projektteilnehmer nach deren Zufriedenheit mit dem bisherigen Stand des Projekts gefragt. Nach den Ergebnissen dieser Umfrage äußerten vier Personen, zufrieden zu seien, zwei jedoch nur teilweise. Es sollte nun versucht werden, die Zufriedenheit des Teams in der restlichen Zeit deutlich zu erhöhen.

9.2 Kritische Bewertung des Vorgehens

Eine Aussage, die unter anderem eine der beiden Umfragen beinhaltete, lautete: „Die Wahl des Unified Process war die richtige.". Alle acht Teilnehmer dieser Umfrage stimmten dieser Aussage zu. Zudem gab eine Person an, dass das tatsächliche Vorgehen im Projekt voll und ganz dem des Unified Process entsprach. Die restlichen sieben Teilnehmer machten als Angabe, dass das tatsächliche Vorgehen stark dem des Unified Process entsprach.

Positiv ist außerdem zu bewerten, dass vier der acht Teammitglieder nach Auswertung der Umfrage daran interessiert sind, auch nach Ende der Lehrveranstaltung noch weiter am Projekt zu arbeiten.

In der Umfrage wurde außerdem gefragt, inwieweit der einzelne Teilnehmer mit seiner Rolle zufrieden sei. Diese Rollen wurden bereits im ersten Meeting des Projekts verteilt. Zwei Teilnehmer seien mit ihrer Rollenzuteilung sehr zufrieden, vier weitere zufrieden. Lediglich zwei gaben zur Angabe, dass sie „teils-teils" mit dieser Zuteilung zufrieden seien.

9.2.1 Probleme

In der Umfrage wurde auch nach Problemen gefragt, die nach Meinung des Teilnehmers bzw. der Teilnehmerin im Softwareprojekt auftraten. Die Teilnehmenden konnten zu den einzelnen aufgeführten Problemen einen Kommentar hinterlassen oder ein zusätzliches Problem, welches noch nicht aufgeführt wurde, hinzufügen und näher erläutern.

In der Umfrage aufgeführtes Problem	**Ja-Stimmen/Anzahl der Teilnehmenden**
fehlende Absprache innerhalb der Projektgruppe/ schlechte Kommunikation	4/8
Schwierigkeiten bei der Entscheidungsfindung innerhalb der Gruppe	2/8
fehlende Informationen	1/8
Zeitmangel	4/8
unklare Projektziele	0/8
schlechte Stimmung innerhalb des Teams	1/8
unterschätzte Komplexität	3/8
mangelnde Unterstützung seitens der Uni	2/8

Bei dieser Abstimmung wurde von einem Teilnehmer darauf hingewiesen, dass es zu Beginn bei der Absprache innerhalb der Projektgruppe und bei der Kommunikation Probleme gab. Jedoch sei dies jetzt besser geworden. Zudem wäre nach Angaben eines anderen Studierenden folgende Situation aufgetreten: „A wartet, dass B fertig wird, aber B braucht Anforderungen von A". Eine andere Person empfand vor allem zu Beginn bis zur frühen Mitte des Projekts die Situation als nicht ganz durchsichtig, vor allem im Bezug darauf, wie der Stand einzelner Personen war. Der Grund sei aber vor allem darin zu begründen, dass jeder viele verschiedene Aufgaben zu erledigen hätte und so nur das Wichtigste in den Meetings verkünden konnte. Ein weiterer möglicher Grund für mangelnde Kommunikation während der Anfangsphase des Projektes könnte darin begründet sein, dass sich die Teilnehmer untereinander noch kaum kannten, und ein normales Kennenlernen

aufgrund der Pandemiesituation nicht möglich war.

Zu Beginn beim Bau der Diagramme seien Schwierigkeiten bei der Entscheidungsfindung innerhalb der Gruppe aufgetreten. Außerdem seien teilweise unnötige Debatten geführt worden.

Als ein großes Problem wurde von den Teilnehmenden der Zeitmangel gesehen. Das Semester sei zu voll. Das Softwareprojekt sollte aber nach Meinung dieser einen Person nicht weniger Zeit in Anspruch nehmen, sondern vielmehr andere Kurse. Eine wiederum andere Person war der Auffassung, dass es zu viele Module in diesem Semester gäbe. Sie hätte zwar keine Wiederholungen offen und gäbe sein Bestes, trotzdem sei das Projekt sehr aufwendig. Auch wurde kritisiert, dass zum Zeitpunkt der Umfrage die zweite Phase längst beendet sei, jedoch das Produkt AEGIS immer noch nicht richtig funktioniere. Ein anderer Mitstreiter wiederum sah weniger das Problem im Zeitmangel, sondern eher in den vorgegeben 20h, die meist überschritten werden mussten.

Von einem wurde auch die unterschätzte Komplexität, vor allem in Hinsicht auf die Klasse `PacketInfo`, und die mangelnde Unterstützung seitens der Universität als Problem gesehen. Er fragte sich, ob die Teilnehmer wirklich bereit für ein solches Projekt seien oder ein großer Teil von denjenigen gemacht wurde, die schon vor der Uni über Wissen verfügten. Jemand anders würde gerne mehr bzw. andere Softwarelizenzen von der Universität zur Verfügung gestellt bekommen. Seiner Meinung nach sei zudem die Komplexität durch mangelndes Wissen über die Bedingungen, die Programmiersprache und die generellen Konstrukte erhöht worden.

Eine Person äußerte Probleme bezüglich fehlender Informationen und einer schlechte Stimmung innerhalb des Teams. So wäre es für sie interessant gewesen, über die genauen Umgebungsparameter informiert zu sein. Ihrer Meinung nach hätte es zwar am Anfang eine noch größere Belastung mit sich gebracht, jedoch sei es dann vielleicht planbarer gewesen. Auch spürte sie teilweise schlechte Laune in Meetings. Die Ursache sah sie im Stress. Hier soll angemerkt werden, dass lediglich ein Teilnehmender diese beiden Punkte als Problem sah.

Unklare Projektziele innerhalb des Teams wurde von keinem als Probleme identifiziert.

Auch wenn einige Probleme, die wahrscheinlich auf der Unerfahrenheit vieler Projektbeteiligten ruhten, während der Ausführung des Projektes auftraten, wurden diese im Laufe der Zeit reduziert beziehungsweise komplett gelöst. Dies soll hier positiv angemerkt werden und spricht für die Offenheit und Flexibilität des Teams.

9.2.2 Meetings

Es war den Umfrageteilnehmern möglich, eigene Anmerkungen zum Thema „Meetings" hinzuzufügen. So hielt ein Teilnehmer die internen Meetings am Anfang für etwas lang und unorganisiert, jedoch hätten sich diese Dinge im Laufe der Zeit stark verbessert. Er würde die Meetings mittlerweile als das unter diesen Umständen Bestmögliche bezeichnen. Für eine weitere Person waren ebenso die Meetings mit dem Betreuer Martin Backhaus sehr wichtig und hilfreich. Ihre Unberechenbarkeit in puncto Länge störte dieser aber. Denn ihrer Meinung nach wären die Montagsmeetings häufig sehr lange gewesen, während das Meeting am darauffolgenden Donnerstag meist nur eine halbe Stunde andauerte. Dies sei ungünstig, da am Donnerstag alle Teilnehmer drei Stunden Zeit haben.

Eine weitere Anmerkung bezog sich auf vielmehr auf die Frage, ob Online-Meetings Treffen in Präsenz völlig ersetzen könnten. Hier wurde angemerkt, dass durch Meetings in Präsenz Absprachen wahrscheinlich leichter zu treffen wären, sowie die Kommunikation einfacher gewesen wäre. Außerdem würden eventuell Fehler schneller aufgedeckt worden sein. Zudem würden die Online-Meetings das Bilden eines Teamgefühls erschweren. Generell wäre nach Auffassung dieses

Umfrageteilnehmenden das Testen am Testbed durch die direkte Anwesenheit bei Präsenzlehre um einiges einfacher gewesen.

Trotz allem wurde an den viermal wöchentlichen Meetings zahlreich teilgenommen. Von KW 17 bis KW 27 fanden insgesamt 36 Meetings statt. Von diesen 36 Meetings war der Betreuer an 20 anwesend. Die restlichen Treffen fanden gruppenintern statt.

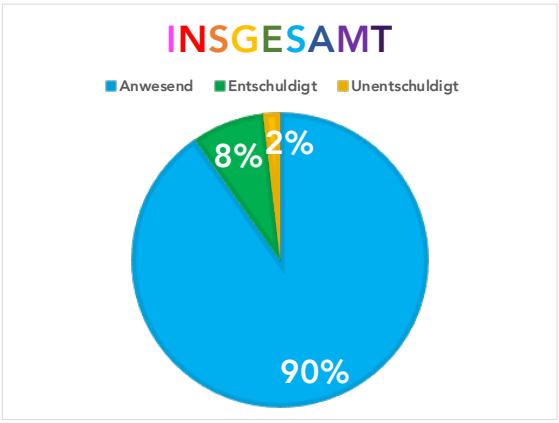


Abbildung 9.3: Durchschnittliche Anwesenheit

Insgesamt waren an den Meetings 90,24% der Teammitglieder anwesend (vgl. Abb 9.3). An 8,03% fehlt jemand entschuldigt und bei 1,73% unentschuldigt.

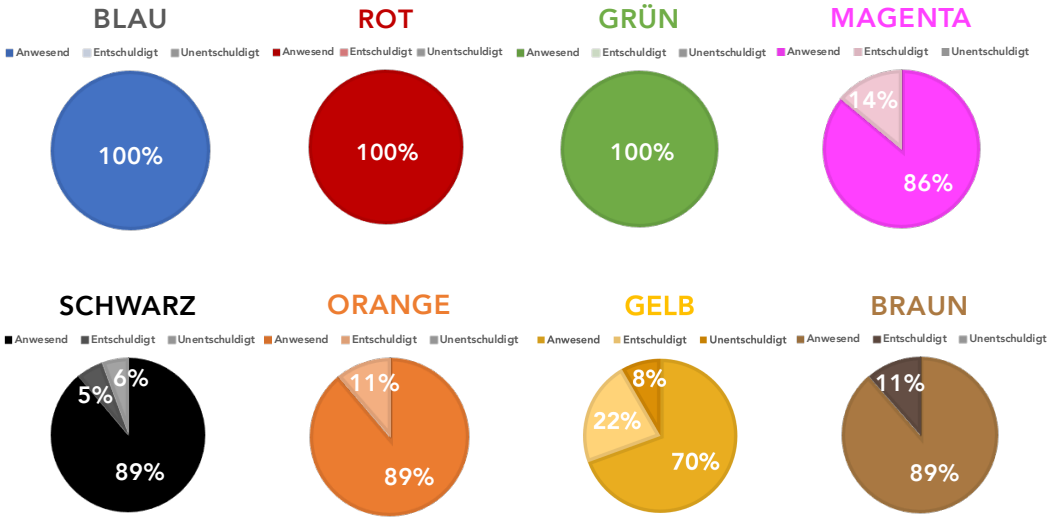


Abbildung 9.4: Vergleich der Anwesenheit zwischen den unterschiedlichen Projektteilnehmern

Drei der acht Projektteilnehmer waren an allen 36 Meetings anwesend. Weitere drei Personen

waren an 89% anwesend. Die Anwesenheitsquote eines weiteren Teammitglieds liegt mit 86% nur knapp darunter. Am seltesten war die „gelbe" Person anwesend. Von den Personen, die an manchen Terminen fehlten, hat sich der Großteil zuvor bei der Projektleiterin entschuldigt.

Anmerkung: Es wurden für die einzelnen Personen die gleichen Farben gewählt wie bei der Kimai-Auswertung in Kapitel 7.

9.2.3 Eintreten von Risiken

Folgende Risiken wurden in der Planungs- und Entwirfsphase identifiziert:

ID	Risiko	Eintrittswahrscheinlichkeit	Auswirkung	Maßnahmen
R01	Software wird unzureichend dokumentiert.	30%	Sicherheitslücken, falsch aufgefasste Anforderungen, Softwareanomalien, schwierige Wartung, Software kann nicht richtig getestet werden u. v. m.	Motivation der Teammitglieder dazu, dass diese gewissenhaft Dokumentation führen; Erklären der Wichtigkeit der Dokumentation; Einführen von Konventionen zur Dokumentation; Verwendung automatischer Dokumentationswerkzeuge
R02	Unzureichende Erfahrung des Projektteams bezüglich neuer Tools (z. B. DPDK, Ninja, Meson) und der Programmiersprache C++	80%	Zeitverzögerung (v. a. längere Dauer der Implementierung durch umfassende Einarbeitungsphase)	Gute Einarbeitung; Erstellen und Halten von Präsentationen zu schwierigen Themen; Gegenseitige Hilfe; Festlegen von Ansprechpartner für die einzelnen Themenbereiche
R03	Weniger Austausch und weniger effiziente Zusammenarbeit durch Online-Lehre	70%	Probleme werden später oder nicht sichtbar; schwieriger Überblick über den Arbeitsstand bei anderen Teammitgliedern; Statusmeldungen fehlen; Geringes Teamgefühl	regelmäßige Treffen ohne Zeitdruck; möglichst organisierte Kommunikation (z.B. über Zulip oder Webex)
R04	Hardware-Probleme (z. B. Ausfall oder andere Defekte)	20%	Zeitverzögerung, finanzielle Kosten	sorgfältiger Umgang mit der Hardware

ID	Risiko	Eintrittswahrscheinlichkeit	Auswirkung	Maßnahmen
R05	Testbed nicht optimal konfigurierbar (aufgrund der verringerten Geräteanzahl und beschränkter Optionen ist nicht jede vorteilhafte Konstellation möglich)	60%	erschwerte Implementierung, Zeitverzögerung, Nichterfüllen einzelner Anforderungen, finanzielle Kosten beim Kauf zusätzlicher Hardware	möglichst effiziente Nutzung der vorhandenen Hardware; Kauf zusätzlicher Hardware; Entwicklung eines Netzwerkplans

Abbildung 9.5 visualisiert die in der Planungs- und Entwurfsphase identifizierten Projektrisiken R01 bis R06, indem sie die Eintrittswahrscheinlichkeiten und die dazugehörigen Schadensausmaße ins Verhältnis setzt. Je weiter man sich im Diagramm nach rechts bewegt, desto höher ist die Eintrittswahrscheinlichkeit. Während diese Wahrscheinlichkeit ganz links bei 0% liegt, beträgt sie am rechten Rand 100%. Je weiter oben sich das Risiko im Diagramm befindet, desto größer ist das Schadensausmaß.

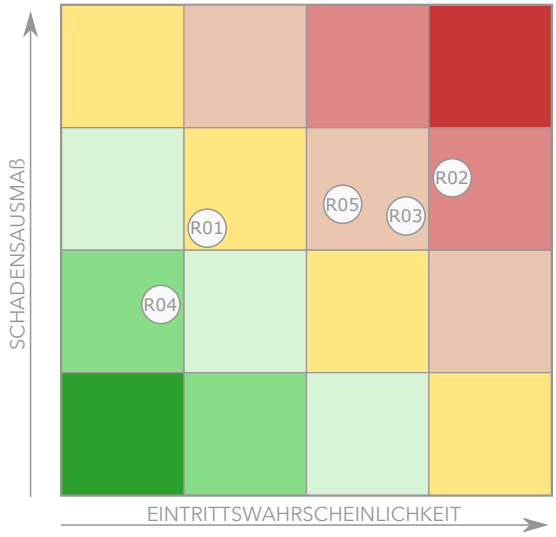


Abbildung 9.5: Risikomatrix aus der Planungs- und Entwurfsphase

In der zweiten Umfrage sollten die Projektteilnehmer bewerten, welche der oben genannten Risiken aufgetreten sind. Falls ihrer Meinung nach eines dieser Risiken aufgetreten ist, sollten sie kurz erläutern, inwiefern sich das Risiko während des Projektes gezeigt hat und wie stark es den Projekterfolg beeinflusst hat.

Risiko	Ja-Stimmen/Anzahl der Teilnehmenden
R01: Unzureichende Dokumentation	0/7
R02: Unzureichende Erfahrung des Projektteams	5/7
R03: Online-Lehre	2/7
R04: Hardware-Probleme	1/7
R05: Testbed nicht optimal konfigurierbar	1/7

Niemand der Umfrageteilnehmer war der Meinung, dass sich das Risiko R01 (Unzureichende Dokumentation) während des Projekts gezeigt hat.

Dagegen war über die Hälfte der Teilnehmer der Ansicht, dass dies bei R02 der Fall war. Vor allem die nötige Einarbeitung in DPDK und andere Systeme wie zum Beispiel Meson wurde als sehr arbeitsintensiv und zeitaufwendig empfunden. Auch musste die Projektteilnehmer sich anfangs in die Programmiersprache C++ gewöhnen.

Zwei der drei Umfrageteilnehmer sahen ein Problem in der Online-Lehre (R03). Dieses Problem wurde aber lediglich als gering eingestuft. Hierbei wurde der fehlende Kontakt außerhalb der „Arbeit" und der seltene physische Zugriff auf das Testbed geäußert.

Nur eine Person äußerte sich zum Thema Hardware-Probleme. Bei dieser sei gelegentlich beim Testen und Ausführen von AEGIS der eigene Computer ausgefallen, da dieser nicht so leistungsstark sei.

Auch dass das Testbed nicht optimal konfigurierbar sei, ist lediglich von einer Person als gering eingestuft worden. So wurden beispielsweise zur dessen Verwendung Namespaces benötigt worden.

Was sich zeigt, ist, dass Risiko R02, dass schon als Beginn als das mit der höchsten Eintrittswahrscheinlichkeit identifiziert worden war, nach Meinung der meisten Teilnehmenden eingetreten sei. Risiko R01 scheint gar nicht eingetreten zu sein. Die anderen drei Risiken sind durch ihre geringe Auswirkung so gut wie vernachlässigbar.

9.2.4 Planungs- und Entwurfsphase

Die Teilnehmenden wurden zur Zufriedenheit mit dem Ergebnis des ersten Reviewdokuments, der Präsentation und der Diskussion im Anschluss der Präsentation befragt. In Abb. 9.6 sind die Ergebnisse dargestellt. Das Reviewdokument wurde von fünf Umfrageteilnehmenden als sehr gut bezeichnet. Weiterhin wurde es von jeweils einer Person mit 2, 3 und 4 klassifiziert, was insgesamt zu einer durchschnittlichen Klassifizierung von 4,25 führt. Auch wenn eine Person mit der Präsentation zur Planungs- und Entwurfsphase vom 27.05.2021 sehr unzufrieden gewesen ist, wurde der Vortrag von allen anderen als gut bzw. sehr gut eingeschätzt und durchschnittlich mit 4,25 klassifiziert. Mit der an die Präsentation anschließenden Diskussion sind alle Befragten mindestens zufrieden gewesen, was bei einer Enthaltung zu einem Durchschnitt von ca. 4,7 führt.

Frage	durchschnittliche Klassifizierung durch die Umfrageteilnehmenden
Zeitliche und psychische Belastung	2,125

Frage	**durchschnittliche Klassifizierung durch die Umfrageteilnehmenden**
Verteilung der Arbeit	3,4

Zwar kann bei den oben beschriebenen Ergebnissen durchaus von einem anständigen Ergebnis ausgegangen werden, allerdings brachte die dazu benötigte Arbeit auch eine starke zeitliche und psychische Belastung mit sich. Auf die Frage, wie belastet sie gewesen seien, antworteten sieben der acht Umfrageteilnehmenden mit stark. Eine weitere Person war mittelstark belastet.

Einer anderen Frage zufolge ist die Arbeit nicht besonders gut und nicht besonders schlecht verteilt gewesen. Die Klassifizierung von 3,4 kann definitiv als verbesserungswürdig bezeichnet werden. Es fällt auf, dass die Antworten recht weit gestreut sind und sich die Zufriedenheit mit der Verteilung der Arbeit über die Planungs- und Entwurfsphase hinweg unter den Teammitgliedern ziemlich stark unterscheidet. Eine Person fand die Arbeit sogar sehr gut verteilt, eine andere fand sie schlecht verteilt und merkte an, dass sie hauptsächlich selbst für die schlecht verteilte Arbeitszeit verantwortlich sei. Die Belastung sei gerade gegen Ende der Phase kritisch, aber insbesondere aufgrund eines Feiertags gerade noch machbar gewesen. Gegen Ende der ersten Phase wurde besonders viel Zeit für den Entwurf in dessen Dokumentation im ersten Reviewdokument aufgewendet.

9.2.5 Implementierungsphase

Auch für die zweite Phase des Softwareprojekts, die Implementierungsphase, wurden die Teilnehmenden gefragt, wie zufrieden sie mit dem abgegebenen Reviewdokument und der Präsentation sowie der anschließenden Diskussion gewesen sind. Abbildung 9.7 zeigt die Ergebnisse.

Die Ergebnisse kommen bezüglich des Reviewdokuments und der Präsentation sehr nah an die der Planungs- und Entwurfsphase heran. Eine durchschnittliche Klassifizierung von 4,125 und 4 besagt, das die meisten Teammitglieder der Auffassung sind, dass das Reviewdokument den Ansprüchen genügen sollte. Bei der Präsentation zeigt sich mit einer durchschnittlichen Einordnung von 4,125 ein ähnliches Bild. Etwas anders sieht es hingegen bei der Diskussion, die nach dem Vortrag stattgefunden hat, aus. Eine Einordnung von 3,17 besagt, dass die meisten Befragten nicht wirklich zufrieden gewesen sind. Ein Grund dafür ist vermutlich die nicht besonders treffende Beantwortung von Fragen, zum Beispiel nach dem Stand des Testens und des Prototyps. Hier hätte noch genauer geantwortet und überzeugter von bereits erledigter Arbeit berichtet werden können.

Frage	**durchschnittliche Klassifizierung durch die Umfrageteilnehmenden**
Zeitliche und psychische Belastung	2,625
Verteilung der Arbeit	3,714

Wie man aus der oben stehenden Tabelle ablesen kann, wurde auf die Frage nach der zeitlichen und psychischen Belastung im Durchschnitt mit einem Wert von 2,625 geantwortet. Das heißt, dass sich dieser Wert im Vergleich zur Implementierungsphase leicht um 0,5 Punkte verbessert hat.

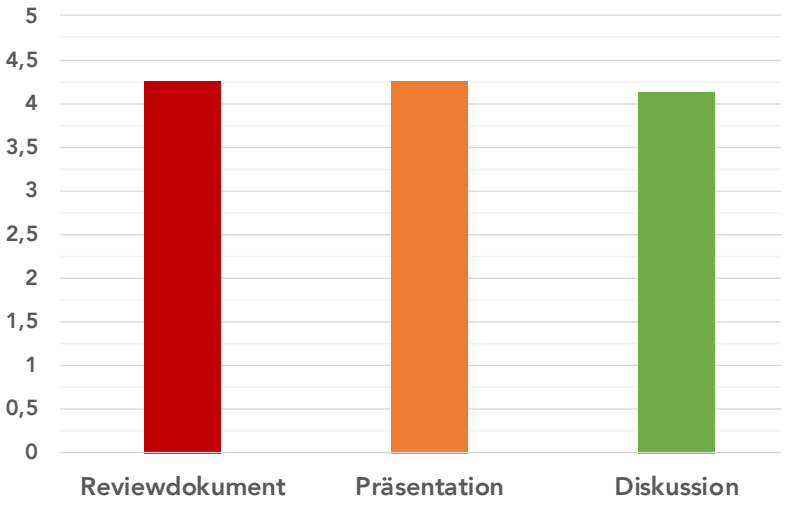


Abbildung 9.6: Planungs- und Entwurfsphase: Ergebnisse der Bewertung des Reviewdokuments, der Präsentation und der Diskussion im Anschluss

Ein ähnliches Bild zeichnet sich bei der Verteilung der Arbeit über die Projektphase hinweg ab. Mit 3,714 wurde im Durchschnitt abgestimmt, was eine leichte Verbesserung um ca. 0,3 bedeutet.

Die Verbesserungen resultieren bei einem Befragten zum Beispiel aus kürzer gehaltenen Meetings. Eine weitere Anmerkung besagt, dass die Motivation für das Projekt mit dem Verlauf stark schwankte.

9.2.6 Validierungsphase

Da zum Zeitpunkt der Umfrage und der Erstellung dieses Dokuments weder das Reviewdokument fertig gestellt worden war noch die Präsentation gehalten worden war, unterscheiden sich die Fragen bezüglich der Validierungsphase zu denen bezüglich der zwei vorherigen Phasen. Die Antworten können auch in diesem Teilkapitel der folgenden Tabelle entnommen werden.

Frage	**durchschnittliche Klassifizierung durch die Umfrageteilnehmenden**
Zufriedenheit mit dem bisherigen Ergebnis des Projekts	3,7
Zeitliche und psychische Belastung	2,3
Verteilung der Arbeit	4,25
Zufriedenheit mit dem bisherigen Stand des Reviewdokuments	5,0

Alle Antworten beziehen sich auf den Stand vom 12.07.2021.

Die Zufriedenheit mit dem bisherigen Ergebnis des Projekts kann als mittelhoch bis hoch bezeichnet werden, denn die durchschnittliche Klassifizierung beträgt 3,7. Ein möglicher Grund dafür ist zum einen die Tatsache, dass nach monatelanger harter Arbeit zum Zeitpunkt der Umfrage fast

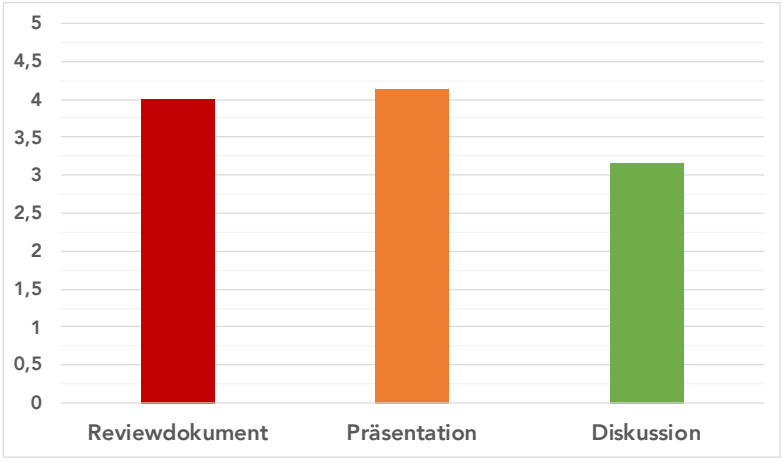


Abbildung 9.7: Implementierungsphase: Ergebnisse der Bewertung des Reviewdokuments, der Präsentation und der Diskussion im Anschluss

ein fertiger Prototyp steht. Auf der anderen Seite wären einige Umfrageteilnehmenden vermutlich zufriedener, wenn noch mehr Anforderungen, die über die Muss-Kriterien hinausgehen, erfüllt werden würden. Ca. eine Woche vor Ende des Softwareprojekts sieht es allerdings danach aus, dass viele spannende Features, die zu Beginn nicht mit Must priorisiert wurden, letzten Endes nicht ihren Weg in das System finden.

Wie man der Tabelle entnehmen kann, bringt die letzte Phase des Softwareprojekts noch einmal eine höhere Belastung für die Teammitglieder mit sich als die Implementierungsphase. Der Wert 2,3 bewegt sich zwischen dem der beiden anderen Phasen. Gerade zum Ende des Projekts sind viele Änderungen und Verbesserungen nötig, während gleichzeitig ein großer Zeitdruck besteht, wodurch für einige Studierende regelmäßig nach einem sowieso schon anstrengenden Tag noch eine Nachtschicht für die Arbeit am Projekt in Angriff nehmen müssen. Dieser und andere Faktoren führen schließlich zu einer fragwürdig starken Belastung über den gesamten Zeitraum des Projekts hinweg. Eine Person merkte an, dass vor allem die Suche nach diversen Fehlern sehr frustrierend sei.

Wichtig ist auch die Anmerkung, dass die Verteilung der Arbeit in dieser Phase so gut wie noch nie bewertet wurde. Der Wert von 4,25 entspricht einer Verbesserung von über einem halben Punkt im Vergleich zu Implementierungsphase. Das heißt, dass das Team im Laufe des Projekts dazugelernt hat und sich die Arbeit im Laufe des Projekts stetig besser verteilt hat. Eine bessere Verteilung ist sehr positiv zu bewerten, weil diese eine in den allermeisten Fällen eine geringere Belastung beziehungsweise eine höhere Produktivität mit sich bringt.

Zuletzt wurde nach der Zufriedenheit mit dem derzeitigen Stand dieses Dokuments gefragt. Zwar zeigt die Bewertung mit 5,0 zunächst eine sehr hohe Zufriedenheit, es muss dennoch angemerkt werden, dass lediglich zwei Teilnehmende bei dieser Frage abgestimmt haben, während es bei den anderen zu Enthaltungen kam. Womöglich haben sich einige Studierende aufgrund der intensiven Arbeit am System während der Validierungsphase noch nicht wirklich mit dem Reviewdokument beschäftigen können, weshalb sie keine unfundierte Antwort abgeben wollten.

9.3 Weitere Ergebnisse aus der Umfrage

In den Umfragen wurden auch Fragen gestellt, die sich nicht in die beiden Kapitel „Kritische Bewertung des Projekterfolgs" und „Kritische Bewertung des Vorgehens" einordnen lassen. Deshalb werden die dennoch relevanten Ergebnisse dieser Fragen hier näher beschrieben.

Unter anderem wurden die Umfrageteilnehmenden gefragt, ob sie das **vierte Semester als einen geeigneten Zeitpunkt für das Softwareprojekt** hielten. Dieser Aussage stimmten 5 Personen zu. Drei Weitere bejahten diese Aussage nicht. Dies liege unter anderem daran, dass sie neben dem Sofwareprojekt zu viele andere Module in diesem Semester hätten. So sei dieses Semester übermäßig voll. Sie klagten darüber, dass sie auf weit über 30 Leistungspunkte, bei einer Person sogar 37 Leistungspunkte, in diesem Semester kämen. Dies führe zu einer enormen Anstrengung und gehe auf die Substanz, vor allem da die 20 Wochenstunden für acht Leistungspunkte auch einen hohen Aufwand darstellten. Eine andere Person war der Meinung, dass auch im sechsten Semester die meisten Studenten nicht auf ein praxistaugliches Projekt vorbeitet seien. So sei es ihrer Meinung nach unerheblich, ob das Projekt im direkt ersten oder erst im letzten Semester planmäßig stattfinde.

Eine ähnliche Frage bezog sich darauf, ob die vom Studiengang abhängige **Anzahl an Leistungspunkten** (6 bzw. 8 LP) und der damit verbundene empfohlene **zeitliche Aufwand** (15 bzw. 20 Wochenstunden) als **angemessen** empfunden werde. Das Ergebnis von 2,875 (1: sehr unangemessen ... 5: sehr angemessen) kann so interpretiert werden, dass die Teilnehmenden die Anzahl an Leistungspunkte und den damit verbunden empfohlenen zeitlichen Aufwand als eher unangemessen empfanden. Es wurde dazu kommentiert, dass es um einiges mehr Zeit in Anspruch nehme, da man das Projekt nie als fertig betrachte und man sich somit praktisch dauerhaft damit beschäftigen müsse. Zudem werde erwartet, dass die Studierenden sehr viel Zeit für das Projekt aufwenden, und das parallel zu einem vollen Stundenplan. Jemand empfand den Aufwand für das Softwareprojekt als viel größer als bei einem regulären 6/8-LP Kurs. Auch eine andere Person hielt es nicht für gut, dass hier für eine Lehrveranstaltung fast so viel Zeit aufgewendet werde wie für alle anderen zusammen. Hier wurde angemerkt, dass durch die 20 Stunden für das Softwareprojekt lediglich 20 Stunden pro Woche für die anderen Fächer übrig blieben (Annahme: Ein durschnittlich intelligenter Student benötigt laut Prüfungsamtsmitarbeiterin IA 40 Stunden pro Woche, um den Verpflichtungen des Studiums nachzugehen). Da diese restlichen 20 Stunden nicht ausreichen würden, hätte dieser Studierende mit einem Workload von über 80h/Woche zu kämpfen.

Die Frage „**Wie fandest du die Unterstützung von den SSElern?**" wurde ebenfalls gestellt. Der Ergebnis-Wert liegt bei 2,5 (1: sehr schlecht ... 5: sehr gut). Da der Anfang zum Teil als Sprung in das kalte Wasser empfunden wurde, würde sich eine Person etwas mehr Anleitung bzw. Begleitung wünschen. Ein anderer Teilnehmer merkte an, dass er zwar nicht im direkten Kontakt mit den Mitarbeitern des Fachgebiets System- und Software-Engineering stand, aber er die Antworten auf Fragen als sehr schwammig und wenig hilfreich empfand. Verbesserungswünsche beziehen sich zudem auf die Übersichtlichkeit des Moodlekurses und der Kompromissbereitschaft bezogen auf die Wahl des Projektthemas mit externen Beratern. Positiv wurde hier aber angemerkt, dass zügig auf Fragen reagiert wurde.

Die **Unterstützung des Fachgebiets Telematik/Rechnernetze** wurde mit 4,25 bewertet (1: sehr schlecht ... 5: sehr gut). Dieser gute Wert ist vor allem dem Betreuer des Projektes hinzuzurechnen, was Kommentare wie „Martin ist der Beste!" zeigen. Die Kommunikation mit ihm wurde als sehr gut bewertet, er sei stets erreichbar gewesen und habe allen bei schwierigen Fragen unterstützt. Ebenso sei die Arbeit sehr gut organisiert gewesen. So wurden gleich zu

Beginn direkt Rollen vergeben und Vorträge sollten vorbereitet werden. Auch bei den Review-Dokumenten und den Vorträgen habe er konstruktiv kritisiert und es wurde noch einmal Druck gemacht, wenn etwas nicht so gut war. So habe dies nochmal das Beste aus dem Team herausgeholt. Zudem habe er auch bei schwierigen Fragem, insbesondere beim Programmieren, das Team stets unterstützt. Das sei vor allem deswegen gut, da die Teilnehmenden vorher keine ausführliche Programmiererfahrunung im Studium hätten. Auch eine andere Person meinte, dass ohne Martins Hilfe der Projektfortschritt viel geringer ausgefallen wäre. Was ebenfalls zu einem so positiven Ergebnis beitrug, war das Angebot des Hostes einer Website für 2 Jahre. Ein weiterer Kommentar sagte: „Martin nahm sich Zeit für meine Probleme und leistete Unterstützung bei fachlichen und allgemeinen Fragen".

Die Teilnehmer wurden auch gefragt, welche Dinge ihnen besonders **Spaß** gemacht haben. Zum einen war dies das Arbeiten im Projekt und als Team oder die gemeinsamen Meetings. Zum anderen wurden das Programmieren, die Freude, wenn etwas nach langem Testen funktioniert hat, und das Lernen von vielem Neuen, vor allem im Bezug auf C++, von vielen als Spaß empfunden. Einige fielen auch Gefallen an der Aufwandsschätzung, am Schreiben der Reviewdokumente und am Halten von Präsentationen.

Nach den Ergebnissen der Umfrage hat aber vor allem der hohe Zeitaufwand, und teilweise auch Zeitdruck, den Teilnehmern **keinen Spaß** gemacht. Auch die Suche von Fehlern, die Dokumentation, die Wartezeiten auf die Fertigstellung anderer Komponenten oder die Installation wurden hier geäußert. Während für manche die Implementierung Spaß machte, waren andere wiederum gegenteiliger Meinung.

Teilweise scheinen die Punkte, die Spaß bzw. nicht Spaß gemacht haben, konträr. So wurde beispielsweise von einigen das Programmieren als spaßig empfunden, von anderen wiederum nicht. Dies ist unter anderem auf die individuelle Wahrnehmung und Interessen der einzelnen Teammitglieder zurückzuführen.

Die Umfrage beinhaltete überdies eine Frage, bei welcher die drei wichtigsten Dinge aufgezählt werden sollten, welche die Person **während des Softwareprojektes gelernt** habe. Es wurde von beinahe allen Teilnehmern aufgeführt, dass sich deren Programmierkenntnisse verbessert haben, insbesondere im Bezug auf die Programmiersprache C++. Zusätzlich habe sich der Schreibstil bei einer Person verbessert. Auch der Umgang mit der Versionsverwaltungssoftware git, dem Betriebssystem Linux, das Lesen von Dokumentation und des Erlernen von Latex-Grundlagen wurden positiv bewertet. Das Erlangen von Projekterfahrung und praktischer Umgang mit Projektmanagement sehen einige als einen wichtigen Gegegenstand. So wurde auch praktisch erlernt, wie effizientes Arbeiten in der Gruppe gewährleistet werden könne. Das heißt zum Beispiel, wann es wichtig sei, ein bestimmtes Thema anzusprechen oder wann dieses lieber übersprungen werden sollte. So musste zudem der Prozess der Software-Entwicklung praktisch komplett durchlaufen werden und ein ganzes Projekt mit Planung und Entwurf umgesetzt werden. Innerhalb des Projekt habe sich auch gezeigt, dass sich Anforderungen schnell wechseln können. Es wichtig sei, von Anfang an feste Grenzen festzulegen. Auch die Relevanz der Kommunikation (u.a. mit Mitarbeitenden) sei durch das Projekt deutlich geworden. Zudem erlangte das Team Wissen über (D)Dos-Attacken.

Eine weitere Frage lautete: „**Inwiefern hat sich das Softwareprojekt von deinen Erwartungen unterschieden?**". Einen Teilnehmer empfand den Verlauf dieses Softwareprojekts als ziemlich gut, wenn er es mit den Erzählungen verglich, die er zuvor gehört habe. Bei diesen Erzählungen wurde über schlechte Kommunikation und Organisation geklagt. Jemand anderes war der Meinung, mit dem Betreuer sehr viel Glück gehabt zu haben. Denn ohne diesem hätte das

Team nicht so gut gewusst, welche Aufgaben es zu welchem Zeitpunkt und auf welche Art und Weise hätte bearbeiten sollen. Er erwartete weiterhin, dass am Projektbeginn mehr Anleitung bzw. Begleitung von den Mitarbeitern des Fachgebiets SSE eingeplant sei. Bei einer wiederum anderen Person unterschieden sich ihre Erwartungen überhaupt nicht von der Realität. Es sei genauso durcheinander, chaotisch, ungeplant und unfertig, wie sie es vermutet habe. Der Nächste wiederum empfand das Projekt als deutlich komplizierter als erwartet. Seiner Meinung nach sei außerdem die Rollenzuteilung teilweise irrelevant gewesen. Zudem habe er mehr Gruppenarbeit erwartet. Eine weitere Person hatte trotz des enormen Zeitaufwands viel Spaß. Von drei anderen wurde angemerkt, dass das Projekt deutlich mehr Aufwand und Zeit in Anspruch nehme als zunächst geschätzt. So sei aber auch das Thema mit etwas mehr Fallstricken und Schwierigkeiten besetzt als erwartet und das bei anderen Projekten der Fall sei.

Kapitel 10

Abkürzungsverzeichnis

DDoS Distributed Denial-of-Service

DoS Denial-of-Service

DRoS Distributed Reflected Denial-of-Service

Gbps Giga bit pro sekunde

KW Kalenderwoche

LoC Lines of Code

Mpps Million packets per second

UML Unified Modelling Language

Kapitel 11

Glossar

Middlebox auch Mitigation Box. Ein System zwischen zwei Systemen das in die Kommunikation der beiden Systeme eingreift.

NIC engl. Network Interface Card; dt. Netzwerkkarte zum Empfang und Senden von Paketen über ein kabelgebundenes oder kabelloses Netzwerk

Non-Worker-Thread Ein einzelner Thread neben den Worker-Threads der eine spezielle und einzigartige Aufgabe im System übernimmt.

Polling das Holen von Paketen von der Netzwerkkarte in den Kernel oder Userspace

RX/TX Queues Empfangs (engl. Receive) und Sende (engl. Transceive) Warteschlangen (engl. Queues) für Netzwerkkarten innerhalb eines Systems

Thread leichtgewichtiger Prozess, auch Ausführungsstrang für die Abarbeitung eines Programms

Worker-Thread Einer von mehreren parallelen Threads, die unabhängig voneinander die Pipeline bearbeiten.

Literaturverzeichnis

[1] infopoint security, "Cyber-angriffe auf deutsche krankenhäuser sind um 220 prozent gestiegen," 2021. Aufgerufen 23.05.2021.

[2] tecchannel, "Trend micro: Latente gefahr durch botnet sdbot." Website, 2009. Aufgerufen 23.05.2021.

[3] NEUSTAR, "2016 ddos report." Website. Aufgerufen 23.05.2021.

[4] datacenterknowledge.com, "Study: Number of costly dos-related data center outages rising," 2016. Aufgerufen 23.05.2021.

[5] ProjectShield, "Projectshield webseite." Website, 2021. Aufgerufen 23.05.2021.

[6] Cloudflare, "Cloudflare ddos protection." Website, 2021. Aufgerufen 23.05.2021.

[7] Amazon, "Aws shield." Website, 2021. Aufgerufen 23.05.2021.

[8] G. Marsaglia, "Xorshift rngs." Website, 2021. Aufgerufen 01.07.2021.

[9] S. Augsten, "Was sind softwaremetriken." Website, 2019. Aufgerufen 09.07.2021.

[10] R. Lackes, "Programmierkonventionen." Website, 2021. Aufgerufen 30.06.2021.

[11] M. Z. et al., "Mitigating volumetric ddos attacks with programmable switches." Website, 2000. Aufgerufen 11.05.2021.

Abbildungsverzeichnis

Verzeichnis der Codeausschnitte