

# Entwicklung eines Linux-Kernel-Moduls zum transparenten Tunneln von CAN Schnittstellen über IP-Netzwerke

An der Fakultät für Informatik und Mathematik der  
Ostbayerischen Technischen Hochschule Regensburg  
im Studiengang  
Technische Informatik

eingereichte

## Bachelorarbeit

zur Erlangung des akademischen Grades des  
Bachelor of Science (B.Sc.)

**Vorgelegt von:** Matthias Unterrainer  
**Matrikelnummer:** 3270790

**Erstgutachter:** Prof. Dr. rer. nat. Wolfgang Mauerer  
**Zweitgutachter:** Dr. Ing. Ralf Ramsauer  
**Externer-Betreuer:** Dr. Nils Weiß

**Abgabedatum:** 01.07.2024



## Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung und Evaluation eines Linux Kernel Moduls, welches das transparente Tunneln von CAN-Schnittstellen über IP-Netzwerke ermöglicht. Hintergrund der Arbeit ist, dass die Firma dissecto GmbH, bei der die Arbeit angefertigt wurde, eine eigene Hardware entwickelt hat, die eine Ethernet Schnittstelle für bis zu zwei physikalischen CAN-Bussen darstellt. Diese Hardware verwendet dabei ein dafür entwickeltes und auf UDP aufbauendes Protokoll, mit dem neben den CAN-Frames auch noch die präzisen Zeitstempel der einzelnen Frames übertragen werden können. Das Modul soll dabei das Gegenstück zu dem Gerät im Linux Kernel darstellen und eine Alternative zu bestehenden Userland Anwendungen wie beispielsweise *Cannelloni* oder *Socketcand* bieten. Die Kombination aus dem Modul und der Hardware soll es ermöglichen "remote" mit den angeschlossenen CAN-Bussen zu interagieren und präzise zeitliche Messungen an diesen durchzuführen, da durch das Mitübertragen der Zeitstempel Verzögerungen durch die Übertragung sich nicht auf die Messungen auswirken können. Das Modul behebt dabei einige Probleme mit den existierenden Lösungen und ermöglicht gleichzeitig eine effiziente und präzise Messung der Zeitstempel der CAN-Frames. Um zu zeigen, dass es sich bei dem Modul um eine leistungsfähige und zuverlässige Lösung handelt, wird es auf verschiedene Qualitätsparameter wie Durchsatzrate, Latenz und Stabilität getestet und anhand dieser auch mit den bereits bestehenden Lösungen verglichen.

# Inhaltsverzeichnis

<b>Zusammenfassung</b>	<b>I</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Zielsetzung und Abgrenzung der Arbeit . . . . .	2
1.3. Gliederung der Arbeit . . . . .	2
<b>2. Grundlagen</b>	<b>4</b>
2.1. CAN . . . . .	4
2.2. C2E-Protokoll . . . . .	6
2.3. Das C2E-Gerät . . . . .	8
<b>3. Implementierung</b>	<b>9</b>
3.1. Anforderungen . . . . .	9
3.2. Entwicklungsumgebung . . . . .	9
3.3. Aufbau . . . . .	10
3.4. Vorteile des Aufbaus gegenüber andere Lösungen . . . . .	11
<b>4. Leistungstests</b>	<b>12</b>
4.1. Qualitätsparameter . . . . .	12
4.2. Latenzzeit . . . . .	13
4.2.1. eBPF . . . . .	13
4.2.2. Aufbau . . . . .	13
4.2.3. Messungen . . . . .	15
4.3. Durchsatzrate . . . . .	19
4.3.1. Aufbau . . . . .	19
4.3.2. Messungen . . . . .	20
4.3.3. Paketverlust . . . . .	22
4.3.4. Fazit . . . . .	22
<b>5. Leistung des Moduls im Vergleich</b>	<b>23</b>
5.1. Cannelloni . . . . .	23
5.2. Socketcand . . . . .	23
5.3. Latenzzeit . . . . .	23
5.4. Durchsatzrate . . . . .	25
5.5. Paketverlust . . . . .	27

<b>6. Stabilitätstests</b>	<b>29</b>
6.1. Methodik . . . . .	29
6.2. Auswertung . . . . .	29
<b>7. Zusammenfassung und Ausblick</b>	<b>30</b>
<b>8. Anhang</b>	<b>31</b>
<b>I. Abbildungsverzeichnis</b>	<b>I</b>
<b>II. Tabellenverzeichnis</b>	<b>I</b>
<b>III. Abkürzungsverzeichnis</b>	<b>II</b>
<b>IV. Literatur</b>	<b>III</b>
<b>Erklärung zur Bachelorarbeit</b>	<b>V</b>

# 1. Einleitung

Die Einleitung soll dazu dienen, dem Leser einen umfassenden Überblick über die Aufgabenstellung sowie den strukturellen Aufbau der vorliegenden Arbeit zu vermitteln. Dazu wird in Kapitel 1.1 zunächst der thematische Kontext, in dem die Arbeit eingebettet ist, vorgestellt. Anschließend wird in Kapitel 1.2 die Problemstellung und Ziele der Arbeit erläutert, und zum Schluss wird in Kapitel 1.3 der Aufbau der Arbeit dargelegt.

## 1.1. Motivation

Der Controller Area Network (CAN)-Bus wurde ursprünglich im Jahr 1985 von BOSCH als ein Multi-Master, Message Broadcast System für Netzwerke im Automobilbereich entwickelt. Das Ziel dabei war es, die stetig wachsende Anzahl elektronischer Geräte in Fahrzeugen kostengünstig und effizient miteinander zu verbinden. CAN löste dabei den traditionellen Ansatz von Point-To-Point (PTP)-Verbindungen ab und wurde im Jahr 1993 ein internationaler Standard (ISO 11898).

Heute stellt CAN mit die am weitesten verbreitete Netzwerk-Technologie in der Automobil-Industrie dar und bietet mittlerweile auch eine Vielzahl von Higher-Level-Protokollen auf CAN an [7].

Die Firma dissecto GmbH, bei der die vorliegende Arbeit durchgeführt wurde, hat sich auf die Analyse und das Testen der Security von sowohl Hardware als auch Software für eingebettete Systeme spezialisiert, wobei der Fokus auf eingebettete Systeme im Automobilbereich liegt [1]. Da CAN aufgrund seiner weiten Verbreitung, als häufig die zentrale Kommunikationstechnologie in Automobilen, eine große Angriffsmöglichkeit darstellt, muss dieser daher auch bei der Sicherheitsevaluierung des Systems berücksichtigt werden.

Aus diesem Grund hat die Firma dissecto GmbH Hardware entwickelt, die in einen CAN-Bus eingehängt werden kann und dann über Ethernet eine Schnittstelle zu diesem darstellt. Dabei ermöglicht diese Schnittstelle, mittels Zeitstempel, das Durchführen von zeitlichen Messungen und Analysen des Netzwerkverkehrs am CAN-Bus, was dabei helfen soll mögliche Sicherheitslücken zu finden. Das Ziel dieser Arbeit ist es das Gegenstück zu dieser Hardware in der Form eines Linux Kernel Moduls zu entwickeln, um eine transparente Kommunikation mit dem CAN-Bus zu ermöglichen.

Die Kombination von Hardware und Modul ermöglicht, nur das CAN2ETH (C2E)-Gerät in den CAN-Bus einzuhängen und dann über das Linux Kernel Module (LKM) mit dem Bus zu interagieren. Dies hat gewisse Vorteile. So erlaubt es zum einen den Zugriff auf den Bus von deutlich mehr Endgeräten, da keine spezifische CAN Hardware notwendig ist, sondern, abhängig davon wie das ganze dann tatsächlich aufgebaut ist, nicht mal ein Ethernet Anschluss benötigt wird, sondern nur der Zugriff auf das Netzwerk, wenn das C2E-Gerät beispielsweise über das interne Netz erreichbar ist. Dadurch kann das C2E-Gerät auch dort physisch in den Bus eingebaut werden, wo es am besten passt, sodass die Länge des CAN-Busses und damit

verbunden auch dessen Übertragungsrate [9], wenn möglich, nicht beeinflusst wird. Wodurch auch die möglichen Ergebnisse nicht durch die Messung verfälscht werden. Stattdessen kann dann die Verbindung über Ethernet, bzw. beliebige IP-Netzwerke, entsprechend länger ausgelegt werden.

Da die Verbindung zwischen dem Rechner und dem Gerät nur über die entsprechenden IP-Adressen geregelt wird, kann des Weiteren bei diesem Aufbau auch schnell zwischen unterschiedlichen CAN-Bussen gewechselt werden, da jedes Mal nur die IP-Adressen angepasst werden müssen.

Darüber hinaus spielt beim Messen am Bus die durch die Übertragung eingebrachte Verzögerung keine so große Rolle, da die Zeitstempel sofort beim Auftauchen der CAN-Frames am Bus erstellt werden und beim Empfangen am Zielrechner lediglich übernommen werden. Beim Senden soll der Umstand, dass es sich um ein Kernel Modul handelt, dafür sorgen, dass die Verzögerung durch das Betriebssystem möglichst klein gehalten wird und hauptsächlich von der verwendeten Netzwerk-Technologie abhängt.

Zusammenfassend lässt sich also sagen, dass das LKM eine gute Möglichkeit bietet die Analyse von CAN-Netzwerken zu erleichtern.

## 1.2. Zielsetzung und Abgrenzung der Arbeit

Ziel dieser Arbeit ist es, ein verlässliches Gegenstück zum C2E-Gerät im Linux Kernel, in der Form eines ladbaren Moduls, zu implementieren. Dabei soll die Implementierung möglichst performant bleiben und die wichtigsten Features des C2E-Protokolls implementieren. Ein besonderer Fokus liegt dabei auf der richtigen Verarbeitung und Mitgabe der Zeitstempel an den Netzwerkstack, sodass die empfangenen CAN-Frames mit ihren korrespondierenden Zeitstempeln im Userland bereitgestellt werden, da dies mit das zentrale Element des C2E-Protokolls darstellt. Um die Leistungsfähigkeit des entwickelten LKMs zu zeigen, wird es anhand mehrerer Qualitätsparameter, wie Durchsatzrate und Latenz, getestet und mit bereits bestehenden Lösungen wie *Cannelloni* [18] oder *Socketcand* [19] verglichen. Des Weiteren wird auch noch die Stabilität des LKMs getestet, um zu zeigen, dass es sich dabei um eine zuverlässige Lösung handelt.

## 1.3. Gliederung der Arbeit

Die Arbeit gliedert sich in sieben Bestandteile, wobei der erste Teil diese Einleitung (Kapitel 1) darstellt. Hier wird die Arbeit kurz thematisch eingeordnet und die darin verfolgten Ziele definiert.

Im zweiten Abschnitt (Kapitel 2) werden erst mal die grundlegenden Begriffe und Konzepte, die in dieser Arbeit verwendet werden, näher erläutert.

Daran anschließend wird in Kapitel Kapitel 3 das Modul, das zur Lösung der Aufgabenstellung angefertigt wurde, vorgestellt und dessen Vorteile gegenüber den anderen Lösungen

erklärt.

In Kapitel Kapitel 4 werden die Qualitätsparameter anhand derer das LKM getestet wird, als auch das dabei verwendete Vorgehen, vorgestellt. Und anschließend ausgewertet und verglichen.

Gefolgt davon wird das Modul in Kapitel Kapitel 5 anhand er vorher definierten Qualitätsparameter mit den Protokollen *Cannelloni* und *Socketcand* verglichen.

Im Anschluss daran wird in Kapitel Kapitel 6 noch die Stabilität des LKM bewertet.

Und zum Abschluss werden in Kapitel Kapitel 7 die Ergebnisse der vorliegenden Arbeit nochmal zusammengefasst.



## 2. Grundlagen

Da sowohl CAN als auch C2E essenzielle Rollen in dieser Arbeit spielen, wird im folgenden Kapitel zunächst auf CAN, die verschiedenen Formen in denen es auftritt und dessen Aufbau eingegangen (Kapitel 2.1). Gefolgt davon wird der Aufbau des C2E-Protokolls vorgestellt (Kapitel 2.2) und abschließend wird noch kurz auf das C2E-Gerät eingegangen (Kapitel 2.3).

### 2.1. CAN

Wie bereits zu Beginn der Arbeit erwähnt wurde der CAN Bus ursprünglich von BOSCH als Multi-Master, Nachrichten-Broadcast System, mit einer maximalen Übertragungsrate von 1 Mbps, entwickelt. Dabei sendet CAN, anders als zum Beispiel USB oder Ethernet, nicht große Datenblöcke Punkt-zu-Punkt von Knoten A nach Knoten B unter der Aufsicht eines zentralen Busmasters. Im Gegenteil, in einem CAN Netzwerk werden viele kurze Nachrichten, wie Temperatur oder RPM, über das ganze Netzwerk gesendet. Dies gewährleistet die Konsistenz der Daten in jedem Gerät [10].

CAN definiert einen Event getrieben und damit zufälligen Zugriff auf den Bus. Das bedeutet, dass anders als bei zeitgesteuerten Systemen Nachrichten zu beliebigen Zeitpunkten von beliebigen Knoten im System gestartet werden können. Um bei diesen zufällig auftretenden Zugriffen, gleichzeitige Zugriffe auf den Bus zu handhaben, verwendet CAN nicht-destruktive, Bit-weise Arbitration mittels des CAN-Identifiers. Dabei bedeutet nicht-destruktiv, dass die Nachricht durch Arbitration nicht verändert wird, der Gewinner also einfach mit dem Senden fortfahren kann. Und Bit-weise, dass der Knoten, der als Letztes eine „0“ (dominant) als Identifier Bit sendet, während alle übrigen Knoten eine „1“ (rezessive) senden, die Kontrolle über den Bus behält. Damit stellt die „0“ als CAN-Identifier auch die höchste Priorität dar, die ein CAN-Frame haben kann [10].

CAN-Frames kommen in verschiedenen Ausprägungen vor. Zum einen kann ein Frame *Standard CAN* oder *Extended CAN* sein, zum anderen kann es CAN oder CAN-Flexible Datarate

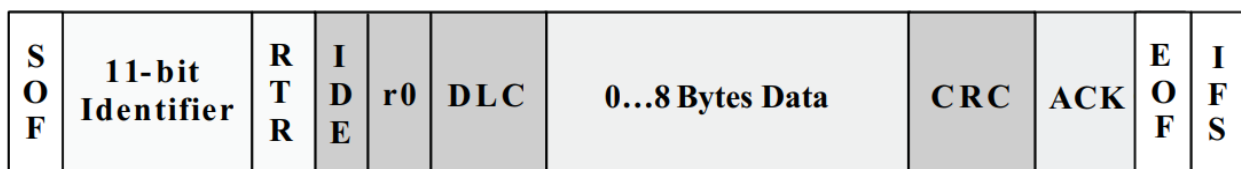


Abbildung 1: Aufbau eines Standard CAN-Frames [10]

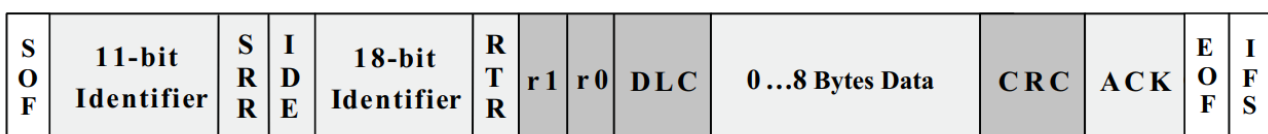


Abbildung 2: Aufbau eines Extended CAN-Frames [10]

(CAN-FD) sein.

**Standard CAN und Extended CAN** In Abbildung 1 ist der Aufbau eines Standard CAN Frames und in Abbildung 2 der eines Extended CAN-Frames zu sehen. In beiden Fällen wird der vollständige Frame so wie er am Bus übertragen wird angegeben. Die Darstellung von CAN-Frames im Speicher wird unter Linux von *SocketCAN*, eine CAN-Protokollimplementierung für Linux basierend auf der Berkeley Socket API [2], in Form der Strukturen `struct can_frame` bzw. `struct canfd_frame` implementiert und besteht nur aus den Informationen die nötig sind, um einen Frame über einen Bus zu senden oder empfangen [3]. Da es sich bei dem LKM nur um einen rein virtuellen CAN-Bus handelt verwendet es auch nur diese Definitionen der Frames. Demnach sind im C2E-Protokoll sowohl beim klassischen CAN als auch beim Extended CAN nur die Felder von Identifier bis Data enthalten, da nur diese Informationen beinhalten, welche auch außerhalb eines CAN-Busses von Bedeutung sind. Es umfasst als Erstes den CAN-Identifier, welcher gleichzeitig die Priorität des Frames beschreibt. Die Flags Remote Transmission Request (RTR) bzw. Substitute Remote Request (SRR) bei Extended CAN, Identifier Extension (IDE) und den Platzhalter für zukünftige Erweiterungen, r0 bzw. auch r1. Der Data Length Code (DLC) beschreibt die Anzahl der übertragenen Bytes in *Data* und *Data* beinhaltet die tatsächlich übertragenen Daten.

Im Gegensatz dazu beschreiben die Felder Start of Frame (SOF), Cyclic Redundancy Check (CRC), Acknowledgment (ACK), End of Frame (EOF) und Interframe Space (IFS) nur Eigenschaften, welche für die tatsächliche Übertragung des Frames auf einem CAN-Bus nötig sind. Wo beispielsweise das SOF Bit den Anfang einer Nachricht markiert um die einzelnen Knoten des Busses nach einer Idle-Phase wieder zu synchronisieren oder ACK dazu dient, dass alle Knoten die fehlerfreie Übertragung des Frames bestätigen. Die einzige Ausnahme liegt dabei bei CRC, welches die Prüfsumme darstellt und damit Fehler bei der Übertragung erkennen soll. Da C2E aber auf User Datagram Protocol (UDP) aufbaut, übernimmt das diese Rolle [10].

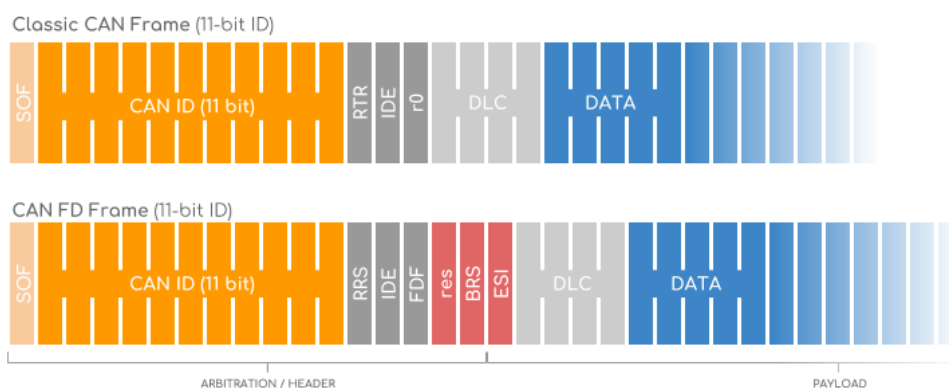


Abbildung 3: Ein Standard CAN-Frame im Vergleich zu einem CAN-FD-Frame mit 11 Bit Identifier. [11]

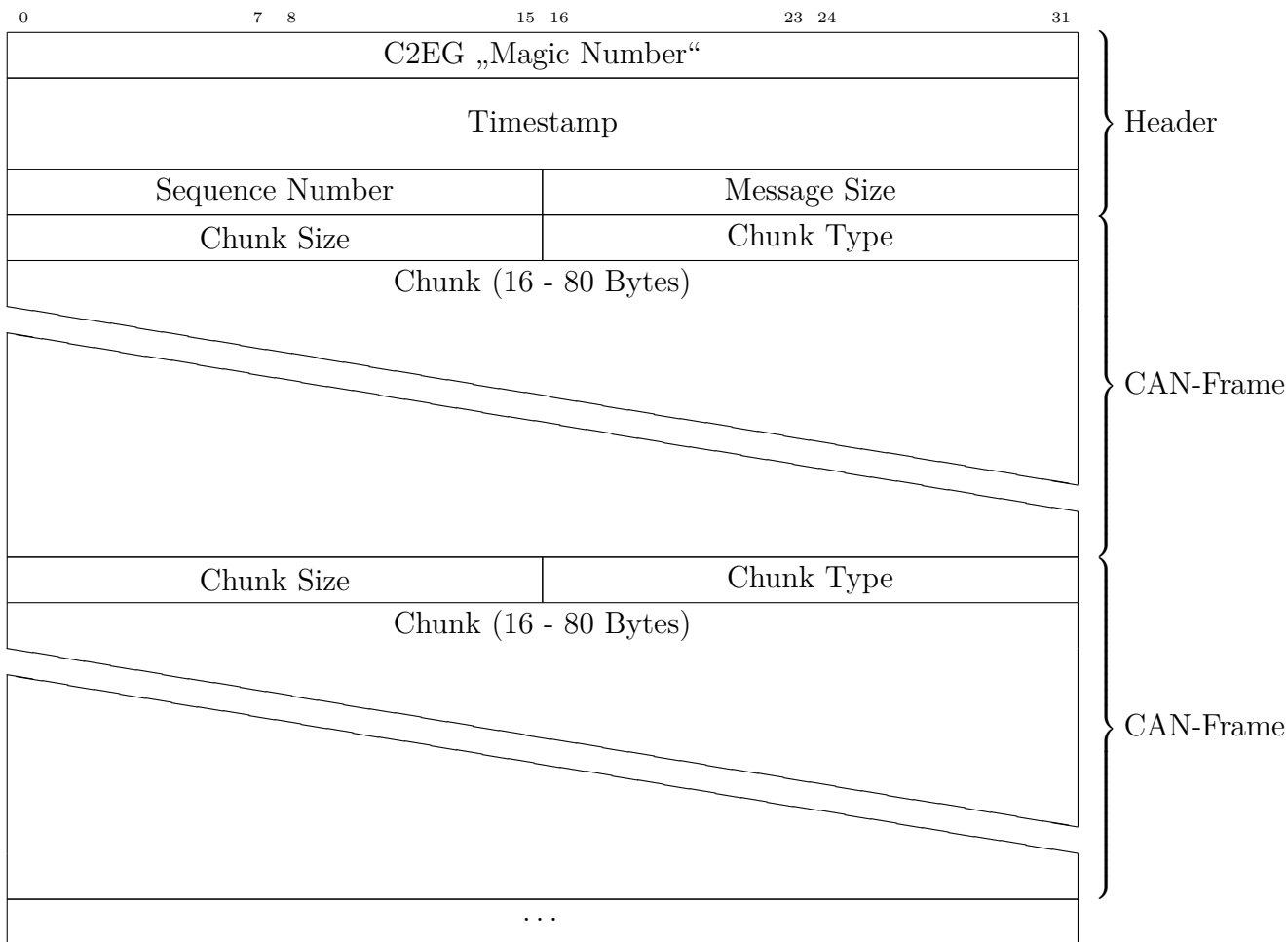


Abbildung 4: Schematischer Aufbau des C2E Headers gefolgt von mehreren „Chunks“, wie beispielsweise CAN-Frames

**CAN und CAN-FD** CAN-FD unterscheidet sich in zwei großen Punkten von CAN. Zum einen bietet es die Möglichkeit Payloads von bis zu 64 Byte in einem Frame zu verschicken, zum anderen ist die Übertragungsrate für die Nutzdaten nicht mehr durch die 1 Mbps von CAN beschränkt [11].

In Abbildung 3 ist beispielhaft der Anfang eines klassischen CAN-Frames und eines CAN-FD-Frames abgebildet. Wie zu sehen ist, ist der CAN-FD-Frame sehr ähnlich zum CAN-Frame aufgebaut mit nur ein paar zusätzlichen Flags die bei CAN-FD dazugekommen sind.

Ähnlich sieht das auch bei Extended CAN-Frames und CAN-FD-Frames mit einem 29 Bit Identifier aus. Für beide Fällen beinhaltet das C2E-Protokoll zusätzlich zu den CAN-Feldern, auch noch Kapazitäten für diese neuen CAN-FD spezifischen Felder.

## 2.2. C2E-Protokoll

Das C2E-Protokoll wurde von der Firma dissecto GmbH für die Verwendung in ihrem C2E-Gerät entwickelt und baut auf UDP auf. Es bietet die Möglichkeit CAN-Frames über IP zu verschicken.

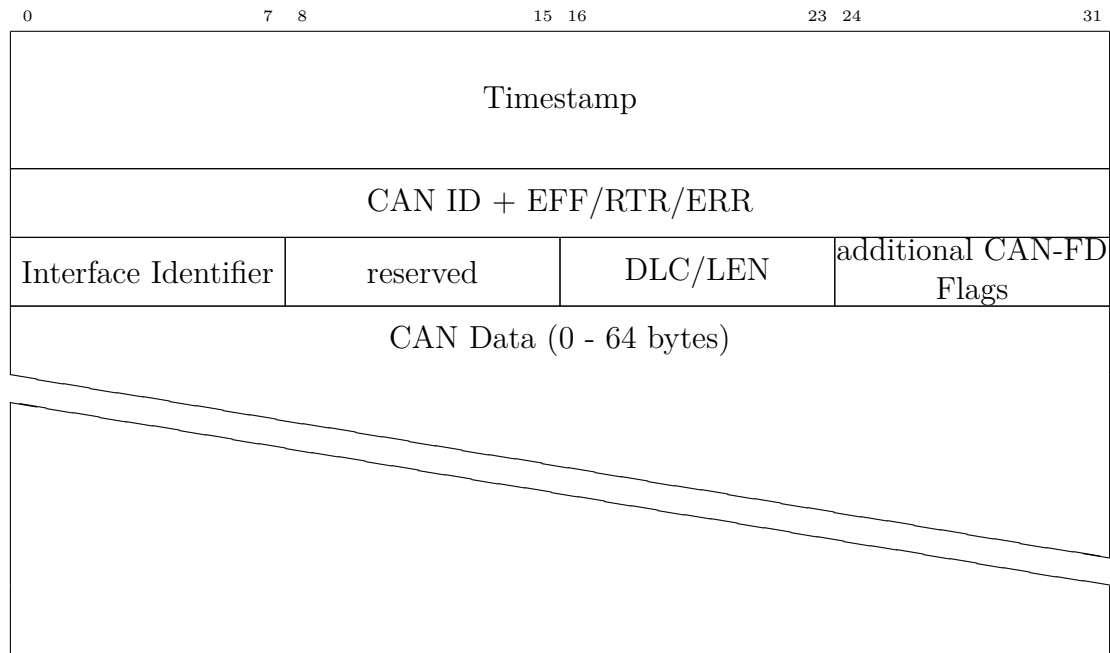


Abbildung 5: Aufbau eines *C2E-Chunk*s

Wie in Abbildung 4 zu sehen, besteht jedes C2E-Paket aus einem Header gefolgt von mehreren *Datachunks*. Der Header startet mit der *Magic Number* „C2EG“, gefolgt von dem Zeitstempel für das gesamte Paket. Abschließend folgen im Header die Sequenz Nummer und die Größe des gesamten Pakets. Die Sequenz Nummer ermöglicht es festzustellen, wenn Pakete verloren gehen, da UDP verbindungslos ist und damit keine verlässliche Kommunikation garantiert.

Der Rest der Nachricht besteht aus mehreren *Chunks*, die beispielsweise die einzelnen CAN-Frames mit allen dazugehörigen Informationen beinhalten, wobei ein *Chunk* immer nur einem CAN-Frame entspricht. Vorangehend zu jedem dieser *Chunks* ist dessen Größe und dessen Typ. Der Typ zeigt dabei an, worum es sich bei dem *Chunk* handelt, so sind nur *Chunks* vom Typen 0xC0FE CAN-Frames. *Chunks* vom Typen 0xFA11 sind *Error Chunks* und vom Typ 0x57A7 *Keep-Alive Chunks*, die in regelmäßigen Abständen gesendet werden und beispielsweise Information über die Gesamtzahl von weitergeleiteten Frames beinhalten. In Abbildung 5 ist der Aufbau eines CAN-Frame *Chunks* dargestellt. Der CAN-Frame wird dabei ähnlich zu der Definition eines CAN/CAN-FD Frames in *SocketCAN* aufgespalten. So wird beispielsweise die CAN-ID in einem 32 Bit Wert gespeichert um genug Platz für sowohl 11 Bit als auch 29 Bit Identifier, zusammen mit den drei zusätzlichen Bit EFF, RTR und ERR, zu haben.

Die restlichen beiden C2E spezifischen Felder sind zum einen der mit dem CAN-Frame assoziierte Zeitstempel und zum anderen der *Interface Identifier*, der die Kennung des Quell-Interfaces angibt, um eine entsprechende Zuordnung zum richtigen Ziel-Interface zu ermöglichen.

### 2.3. Das C2E-Gerät

Das C2E-Gerät wurde speziell als Ethernet-CAN Gateway von der Firma dissecto GmbH entwickelt. Es setzt dabei für die Übertragung über IP das dafür ebenfalls entwickelte C2E-Protokoll (siehe Kapitel 2.2) ein. Es besteht aus einem Mikrocontroller, der über eine Ethernetschnittstelle und zwei CAN-Interfaces verfügt. Es ist dabei nicht mit Linux realisiert worden, um die benötigte Hardware möglichst effizient zu halten, die Performance dieser aber maximal auszunutzen. Des Weiteren werden die einzelnen CAN-Frames nicht direkt von dem Gerät über Ethernet weggeschickt, sondern werden erst mal in einem Buffer gesammelt, der dann, sobald er voll ist oder die eingestellte Wartezeit verstrichen ist, gesammelt verschickt wird. Und die vom C2E-Gerät generierten Zeitstempel sind Monotonic Time, also die Zeit seit dem Start des Systems.

## 3. Implementierung

Im folgenden Kapitel werden zu Beginn die Anforderungen, die das Modul erfüllen muss, aufgezählt (Kapitel 3.1). Gefolgt davon wird die Entwicklungsumgebung, mit der gearbeitet wurde, vorgestellt (Kapitel 3.2) und es wird auf den generellen Aufbau des Linux Kernel Module (LKM) und einige der wichtigeren Designentscheidungen eingegangen (Kapitel 3.3). Zum Abschluss wird noch auf die Vorteile dieses Aufbaus gegenüber dem anderer Implementierungen eingegangen (Kapitel 3.4).

### 3.1. Anforderungen

Da das LKM funktional eine exakte Nachbildung des C2E-Gerätes sein soll, ergeben sich daraus auch die an das Modul gestellten Anforderungen. So soll das Modul folgende Funktionalitäten anbieten.

1. Es soll in der Lage sein mehrere voneinander unabhängige CAN-Interfaces zu unterstützen.
2. Es soll Frame Aggregation unterstützen, also schnell aufeinander folgende Frames zu einem größeren Paket zusammenfassen.
3. Es soll die CAN-Frames so an den Benutzer übergeben, dass sie die übermittelten Zeitstempel enthalten.

### 3.2. Entwicklungsumgebung

Die Entwicklung des Moduls fand in einer, mit QEMU erstellten, Virtual Machine (VM) statt. Eine virtuelle Umgebung wurde verwendet, da sie es ermöglicht das Modul leicht und schnell zu testen. Das Problem bei der Entwicklung eines Linux Kernel Moduls liegt darin, dass ein Fehler in dem Modul schnell zum Absturz oder Instabilität des Systems führen kann. Bei der Verwendung einer VM ist dies allerdings kein großes Problem, da diese schnell neu gestartet oder zwangsweise beendet werden kann. In der VM lief dabei *Arch Linux* mit dem standardmäßig konfigurierten „Linux“ Kernel der Distribution. Zu dem vollständigen Aufbau gehört auch noch das C2E-Gerät. Dieses hat zwei Seiten, die beide in die VM eingebunden werden sollen. Zum einen soll es von der VM aus über Ethernet erreichbar sein. Um dies zu bewerkstelligen, wird das Ethernet-Interface auf der Host-Maschine, über welches das Gerät mit dem Host verbunden ist, zu einer neuen Bridge hinzugefügt. Dieser neuen Bridge wird dann eine IP-Adresse zugewiesen und mittels *dnsmasq* wird auf der Bridge ein DHCP-Server gestartet. Dieser ist notwendig um dem C2E-Gerät eine IP-Adresse zuzuweisen. Die Bridge wird dann als zusätzliches Netzwerk-Interface der VM hinzugefügt, wodurch diese als neues Ethernet-Interface in der VM erscheint. Damit ist das C2E-Gerät sowohl vom Host, als auch von der VM aus erreichbar. Zum anderen sollen die beiden CAN-Interfaces des C2E-Geräts auch von der VM aus erreichbar sein. Dazu werden diese jeweils über einen physischen CAN-Bus mit einem USB2CAN-Wandler mit dem

Host verbunden. Da dieser Wandler ein USB-Gerät ist, kann diese leicht an die VM umgeleitet werden.

### 3.3. Aufbau

Die Lösung wurde als Loadable Linux Kernel Module umgesetzt. Module sind ein Feature von Linux, welches es ermöglicht, die vom Kernel angebotenen Funktionen zur Laufzeit zu erweitern oder zu verkleinern. Dabei gibt es eine Vielzahl von unterschiedlichen Arten von Modulen. Dazu gehören *Character Devices*, auf die wie auf einen Bytestream, z.B. eine Datei, zugegriffen werden kann. Weiterhin gibt es *Block Devices*, die zum Hosten von Dateisystemen verwendet werden, sowie, wie im vorliegenden Fall, *Network Interfaces*, die für die Netzwerktransaktionen des Systems zuständig sind. *Network Interfaces* können dabei sowohl Hardware steuern als auch, wie im Fall des Loopback Interfaces oder auch des im Rahmen dieser Arbeit erstellten Moduls, reine Softwaregeräte sein [8].

Das erstellte Modul besteht aus zwei Komponenten: einem Empfänger und einem Sender, die jeweils als eigenständiger Thread im Kernel Space laufen. Beide Komponenten teilen sich dabei einen UDP-Socket zur Kommunikation über Ethernet. Die Interaktion mit dem Userspace erfolgt über eine konfigurierbare Anzahl von CAN-Interfaces, die das Modul initialisiert und verwaltet. Obwohl bis zu 256 Interfaces vom C2E-Protokoll unterstützt werden (siehe Kapitel 2.2), werden standardmäßig nur zwei initialisiert, um die Zwei des C2E-Gerätes nachzubilden.

Der **Empfänger**-Thread wartet blockierend auf eingehende UDP-Paketen. Jeder CAN-Frame aus dem Paket wird dann an das entsprechende CAN-Interface weitergegeben. Die Frames werden dabei mit ihrem Zeitstempel aus dem C2E-Protokoll versehen, um eine genaue Zeitkorrelation zu ermöglichen. Die Abgabe läuft dabei über die von *SocketCAN* bereitgestellte Funktionalität `can_rx_offload`.

Der **Sender**-Thread umfasst einen von allen Interfaces geteilten Buffer, der in regelmäßigen Abständen bzw. wenn er voll ist geflusht, also in ein UDP Paket gepackt und versendet, wird. Der Thread wird dazu mittels `usleep_range()` für zwischen 0.9 ms und 1.1 ms schlafen gelegt, gefolgt von einer Überprüfung, ob der Buffer geflusht werden soll. Es ist hierbei ein Zeitfenster angegeben, da für Zeiten in dem Bereich von 10  $\mu$ s - 20 ms der Kernel dadurch das Aufwachen besser planen kann. [4]

Der Bufferinhalt wird dabei mit einem Header für das C2E-Protokoll versehen, in ein UDP-Paket verpackt und an alle konfigurierten Zieladressen gesendet. Dieser Prozess des Sammelns und gemeinsamen Versenden von Frames heißt Frame-Aggregation und ist für CAN-Frames sinnvoll, da wie in Kapitel 2.1 angesprochen, CAN mit vielen kleinen Nachrichten arbeitet.

CAN-Frames werden asynchron vom Userspace über die TX-Callback-Funktionalität `ndo_start_xmit()` der einzelnen CAN-Interfaces empfangen. Dabei wird der übergebene CAN-Frame, entsprechend der Definition des C2E-Protokolls, mit einem Zeitstempel und dessen

Größe versehen. Wobei der Zeitstempel *Monotonic Time* ist, also die Zeit sei Boot des Systems, um eine hohe Genauigkeit zu gewährleisten. Anschließend werden die Frames für die spätere Übertragung in den Buffer eingefügt.

Der Zugriff auf den Buffer wird dabei durch ein *Spinlock* koordiniert. *Spinlocks* können, anders als klassische Semaphoren, auch in Code, der nicht schlafen kann, verwendet werden. Ein *Spinlock* kann nur einen von zwei Werten haben, „locked“ oder „unlocked“. Code, der das Schloss annehmen möchte, testet diesen Wert. Wenn das Schloss verfügbar ist, wird es einfach auf „locked“ gesetzt und der Code kann in den kritischen Abschnitt eintreten. Sollte das Schloss nicht verfügbar sein, wird in einer Schleife der Status ständig überprüft bis das Schloss verfügbar wird. [8]

### 3.4. Vorteile des Aufbaus gegenüber andere Lösungen

Die Vorteile von diesem Aufbau sind, dass anders als bei *Cannelloni* oder *Socketcand*, die Anwendung nicht im Userspace, sondern im Kernelspace läuft. Durch die Implementierung im Kernel Space bietet das LKM einen direkteren und effizienteren Zugriff auf Systemressourcen im Vergleich zu Lösungen, die im User Space laufen. Dies führt zu einer höheren Performance, insbesondere in Bezug auf die Verarbeitungsgeschwindigkeit und die Reaktionszeit. Da Kernel-Space-Operationen näher an der Hardware ausgeführt werden, können Sie besser optimiert werden, um die Hardwareeffizienz zu maximieren. Dies ist besonders relevant bei Echtzeitanwendungen, wo jede Millisekunde zählt. Darüber hinaus kann die Kernel-Space-Implementierung die Systemstabilität und -sicherheit verbessern, da sie von den strengeren Kontrollmechanismen und dem privilegierten Zugang des Kernel Space profitiert.

Zudem gibt es, anders als bei *Socketcand* (siehe Kapitel 5.2), Frame-Aggregation für schnell aufeinander folgende Frames durch die Verwendung eines Sendebuffers. Dadurch können diese Frames zusammengefasst und gemeinsam übertragen werden, wodurch nicht für jeden CAN-Frame ein eigener Ethernet-Frame und UDP-Header erstellt und übertragen werden muss. Dies spart sowohl Bandbreite, als auch Ressourcen vom Host des LKM.



## 4. Leistungstests

Um feststellen zu können wie das LKM im Vergleich zu den anderen Lösungen abschneidet, wird es im folgenden Kapitel einigen Leistungstests unterzogen und deren Ergebnisse analysiert und mit anderen Protokollen verglichen. Dazu werden zuerst die Qualitätsparameter, nach denen das Modul getestet wird, definiert und ihre Relevanz begründet (siehe Kapitel 4.1). Anschließend folgen die einzelnen Messungen. Dabei wird immer erst das Vorgehen bei der Durchführung der Tests sowie die verwendete Testumgebung erklärt. Gefolgt davon werden die Testergebnisse vorgestellt und analysiert.

### 4.1. Qualitätsparameter

Das LKM wird anhand der folgenden Schlüsselparameter bewertet, die für die Beurteilung seiner Leistung und Effizienz entscheidend sind:

#### Latenzzeit

- **Definition:** Die Latenzzeit misst die Zeitspanne, die ein Paket benötigt, um das LKM von einem Endpunkt zum anderen zu durchqueren.
- **Relevanz:** Geringe Latenz ist kritisch für die nahtlose Interaktion mit Fahrzeugsystemen und das Einhalten von Echtzeit- und Timing-Anforderungen. Sie beeinflusst direkt die Reaktionsfähigkeit und Effizienz des Systems.
- **Messung:** Latenz wird in Millisekunden gemessen.

#### Durchsatzrate

- **Definition:** Die Durchsatzrate quantifiziert die Menge an CAN-Frames, die pro Zeiteinheit durch das LKM übertragen werden kann. Sie ist ein Maß für die Fähigkeit des Moduls, Daten effizient zu verarbeiten und weiterzuleiten.
- **Relevanz:** Ein hoher Datendurchsatz ist entscheidend für Anwendungen, die eine schnelle Datenübertragung erfordern, wie beispielsweise bei der Überwachung und Steuerung von Fahrzeugsystemen in Echtzeit.
- **Messung:** Die Durchsatzrate wird in Paketen pro Sekunde gemessen.

#### Paketverlust

- **Definition:** Der Paketverlust ist Teil der Durchsatzrate und gibt an, wie viele Pakete unter Last „verloren“ gehen.
- **Relevanz:** Je höher die Verlustrate, desto weniger Pakete, die empfangen werden sollen, kommen tatsächlich an. Damit stellt die Verlustrate eine wichtige Größe zur Beurteilung der Zuverlässigkeit dar.

- **Messung:** Die Paketverlustrate wird in Paketen pro Sekunde gemessen.

Diese Parameter bilden zusammen ein umfassendes Bild der Leistung und Effizienz des LKMs und sind entscheidend für seine Bewertung und weitere Optimierung.

Die im Anschluss folgenden Untersuchungen wurden nicht in der Entwicklungsumgebung (siehe Kapitel 3.2) durchgeführt, sondern ohne Virtualisierung direkt auf der Host-Maschine.

## 4.2. Latenzzeit

Bei der Betrachtung der Latenzzeit wurden zwei Größen betrachtet. Erstens die Zeit, die von einem CAN-Frame benötigt wird, um das Modul einmal zu durchlaufen, wenn dieses vom Userland her kommt. Also die Zeit zwischen dem Ankommen des CAN-Frames an einem der angebotenen CAN-Interfaces und dem Senden des dazugehörigen UDP Paketes. Und zweitens, die Zeit, die für den umgekehrten Prozess nötig ist, also von dem Ankommen eines C2E UDP Paketes bis zum Erscheinen des CAN-Frames am CAN-Interface.

### 4.2.1. eBPF

*eBPF* ist eine Technologie, die ihren Ursprung im Linux Kernel hat und mit der Code in einer speziellen Sandbox innerhalb des Linux-Kernels ausgeführt werden kann. Dies ermöglicht eine breite Palette von Anwendungen, darunter Netzwerküberwachung, Sicherheit, Leistungsoptimierung und vieles mehr. *eBPF*-Programme sind Event-getrieben, d.h. sie laufen ab, wenn der Kernel oder eine Anwendung bestimmte Hook Points in ihrer Ausführung erreichen. Einige solcher bereits Vor-definierter Hook Points sind z.B. system calls oder Funktionsein-/austritte. Wenn für bestimmte Anwendungsfälle kein Hook bereits existiert, dann können *Kernel probes* (*kprobe*) oder *User probes* (*uprobe*) verwendet werden, um *eBPF* Programme fast überall im Kernel oder User Anwendungen anzubringen [5]. *Kprobes* sind ein Mechanismus im Linux Kernel, welcher es ermöglicht dynamisch in Kernel-Routinen einzusteigen und Debugging- und Leistungsinformationen zu sammeln, ohne den Ablauf der Routine zu stören. Sie können dabei in zwei verschiedenen Varianten vor, den *kprobes* und den *kretprobes*, auch „return probes“ genannt. Der Unterschied zwischen den beiden ist, dass die mit einer *kprobe* verbundenen Handler-Routine aufgerufen wird, wenn die Anweisung, auf die die *kprobe* registriert ist, erreicht wird, und bei *kretprobes* die Handler-Routine erst aufgerufen wird, wenn eine bestimmte Funktion zurückkehrt. [13]

### 4.2.2. Aufbau

Für die Durchführung der Latenz-Messungen wurden zwei Ansätze parallel zueinander durchgeführt. Der erste Ansatz besteht aus einem C-Programm, in dem zwei Sockets geöffnet werden. Ein CAN und ein UDP Socket. Dabei wird über einen der beiden, abhängig von der zu messenden Richtung, ein entsprechender Frame gesendet und mithilfe von Monotonic Time die Zeit bis zum Empfangen der Reaktion am anderen Socket gemessen. Die Verwendung von Monotonic

Time stellt hierbei sicher, dass die Messungen präzise und konsistent sind, da bei Monotonic Time keine Diskontinuitäten auftreten. Des Weiteren garantiert Monotonic Time, dass bei wiederholten Aufrufen die zurückgelieferte Zeit nie zurückgeht, aller höchstens gleich bleibt [6]. Danach wird noch eine zufällige Zeit zwischen 0 ms und 100 ms gewartet, um möglichst zufällig verteilte auftretende Frames zu simulieren. Dabei wird der erste Zeitstempel vor dem Aufrufen der *Sende*-Funktion genommen und der zweite nach dem die *Empfangs*-Funktion zurückgekehrt ist.

Für den zweiten Ansatz wurde mittels *eBPF* bestimmte Funktionsaufrufe des LKM getrace und die dazwischen verstrichene Zeit gemessen, aus der sich die durch das LKM erzeugte Latenzzeit ergibt. Für die Richtung UDP nach CAN wurden als Tracepoints der Eintritt der Funktionen `kernel_recvmsg()`, welche im Modul verwendet wird, um UDP Pakete zu empfangen, und der Exit der Funktion `can_rx_offload_irq_finish()`, die den aus dem C2E-Protokoll wieder zusammengebauten CAN-Frame an den Network Stack abgibt, verwendet. Und für die Richtung CAN nach UDP wurde auf den Eintritt von `ctem_xmit`, die Funktion an die auf den CAN-Interfaces ankommende CAN-Frames übergeben werden, und den Exit von `kernel_sendmsg()`, das Äquivalent zu `kernel_recvmsg()`, mit dem die UDP-Frames verschickt werden, geachtet.

Um sicherzustellen, dass nur dann Zeitstempel genommen werden, wenn die Funktionen vom Modul aus aufgerufen werden, wird bei allen Funktionen außer `ctem_xmit`, welches einen eindeutigen Namen hat, nachdem der Zeitstempel genommen wurde noch überprüft, ob es sich um den richtigen Prozess handelt. Für die Funktionen `kernel_recvmsg()` und `can_rx_offload_irq_finish()` wird dies über die Prozess-ID festgestellt. Da hier einfach mittels der `ps` Utility [16] die Prozess-ID des Empfangsthreads (Kapitel 3.3) des Moduls herausgefunden werden kann, die dann mit der in der Handler-Funktion der *kprobe* ermittelten Prozess-ID verglichen wird. Für die Funktion `kernel_sendmsg()` musste dies allerdings etwas erweitert werden, da hier die Funktion entweder im Kontext des Sender-Threads (Kapitel 3.3) oder aber im Kontext der TX-Callback-Funktion (`ctem_xmit`), welche im Kontext des aufrufenden Prozesses läuft, ausgeführt wird. Deshalb wurde im Vorhinein die Prozess-ID von sowohl dem Sender-Thread als auch dem Test-Programm ermittelt und dann mit in dem Handler mit beiden verglichen. Der Grund warum `kernel_sendmsg()` in beiden Kontexten ausgeführt werden kann, liegt darin, dass beim Einfügen des mittels `ctem_xmit` erhaltenen CAN-Frames, gleichzeitig auch überprüft wird, ob es Zeit ist den Buffer zu verschicken. Sollte dies der Fall sein, wird er sofort, noch im Kontext von `ctem_xmit`, verschickt.

Es wurden diese beiden Verfahren angewendet, um ein möglichst vollständiges Bild über die Latenz des Moduls zu bieten. Die Messung mit *eBPF* misst genau die Latenz im Modul, da die Zeit zwischen den spezifischen Funktionsaufrufen im Modul gemessen wird. Dies bietet eine möglichst exakte Betrachtung der durch das Modul eingeführten Latenz. Die Messung mit dem C-Programm hingegen hat zwar deutlich mehr Overhead, da sowohl die Zeit im Network Stack,

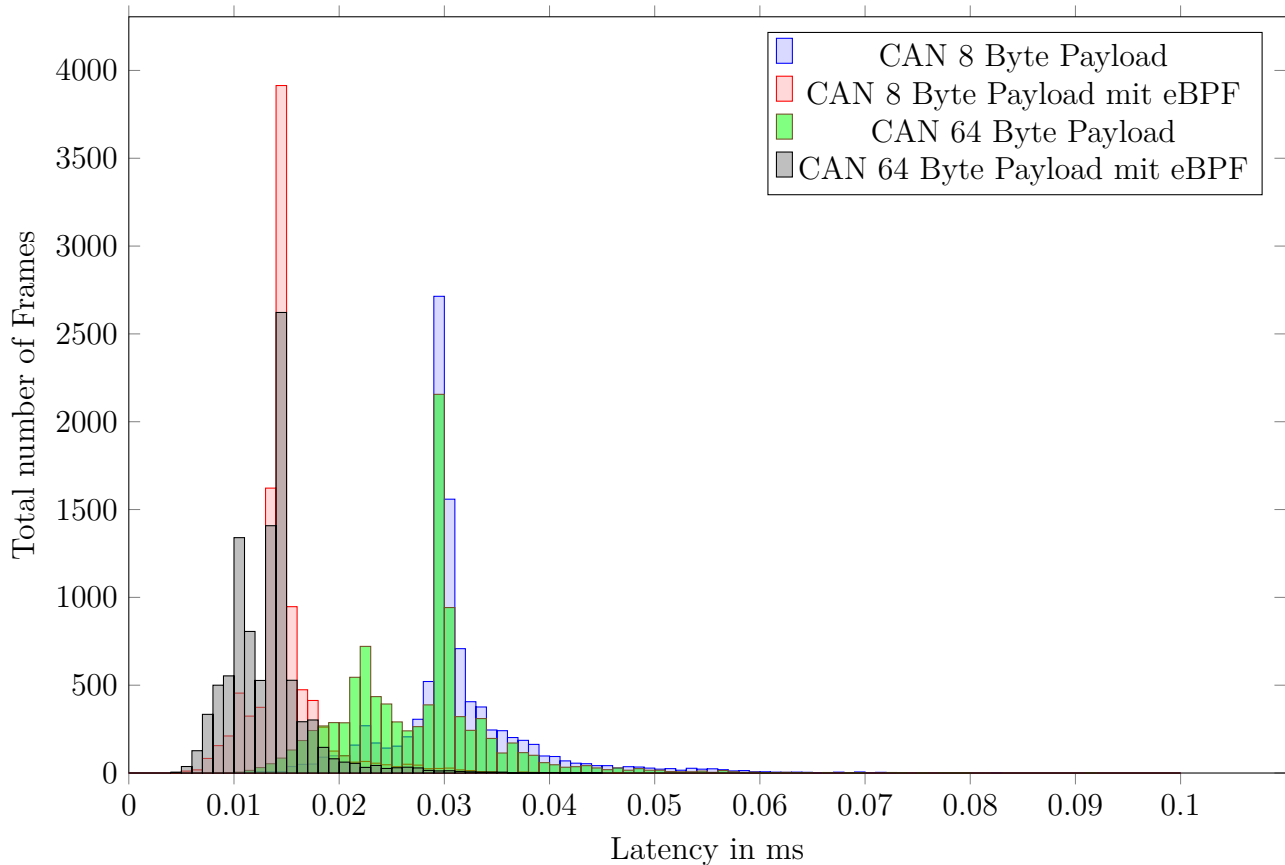


Abbildung 6: Verteilung der Latenzzeiten für CAN nach UDP ohne Timeout.

als auch die Socketoperationen an sich Teil der Messung sind und nicht nur die Zeit zwischen den entsprechenden Funktionsaufrufen im Modul gemessen wird. Allerdings stellt diese Messung einen deutlich realistischeren Fall dar, der in der tatsächlichen Anwendung von Relevanz ist, da, hier die Gesamtlatenz des Systems noch mit reinspielt. In beiden Fällen wird allerdings die Latenz des Netzwerkes nicht berücksichtigt.

Die Messungen wurden dann folgendermaßen durchgeführt. Das C-Programm, sendet 10.000-mal hintereinander, denselben Frame, UDP oder CAN, im Abstand von jeweils 100ms und misst, wie oben beschrieben, die Zeitabstände. Währenddessen wird gleichzeitig auch noch wie in Ansatz zwei beschrieben, mit eBPF die Zeit zwischen den Funktionsaufrufen gemessen.

### 4.2.3. Messungen

**CAN - UDP** Für die Messung von CAN nach UDP wurden zwei Fälle betrachtet. Der erste Fall, zu sehen in Abbildung 6, betrachtet die reine Latenz des Moduls ohne Wartezeit bei der Frame Aggregation. Das heißt alle ankommende CAN-Frames werden sofort versendet. In der Abbildung ist ein Histogramm zu sehen, welches die Häufigkeitsverteilung der verschiedenen aufgetretenen Latenzzeiten darstellt. Hierbei lässt sich ein deutlicher Unterschied zwischen den mit eBPF erfassten Messungen und den ohne erkennen, da die mit eBPF deutlich niedriger ausfallen. Beide häufen sich allerdings um ihre jeweiligen Mittelwerte. Dabei ist auch noch zu

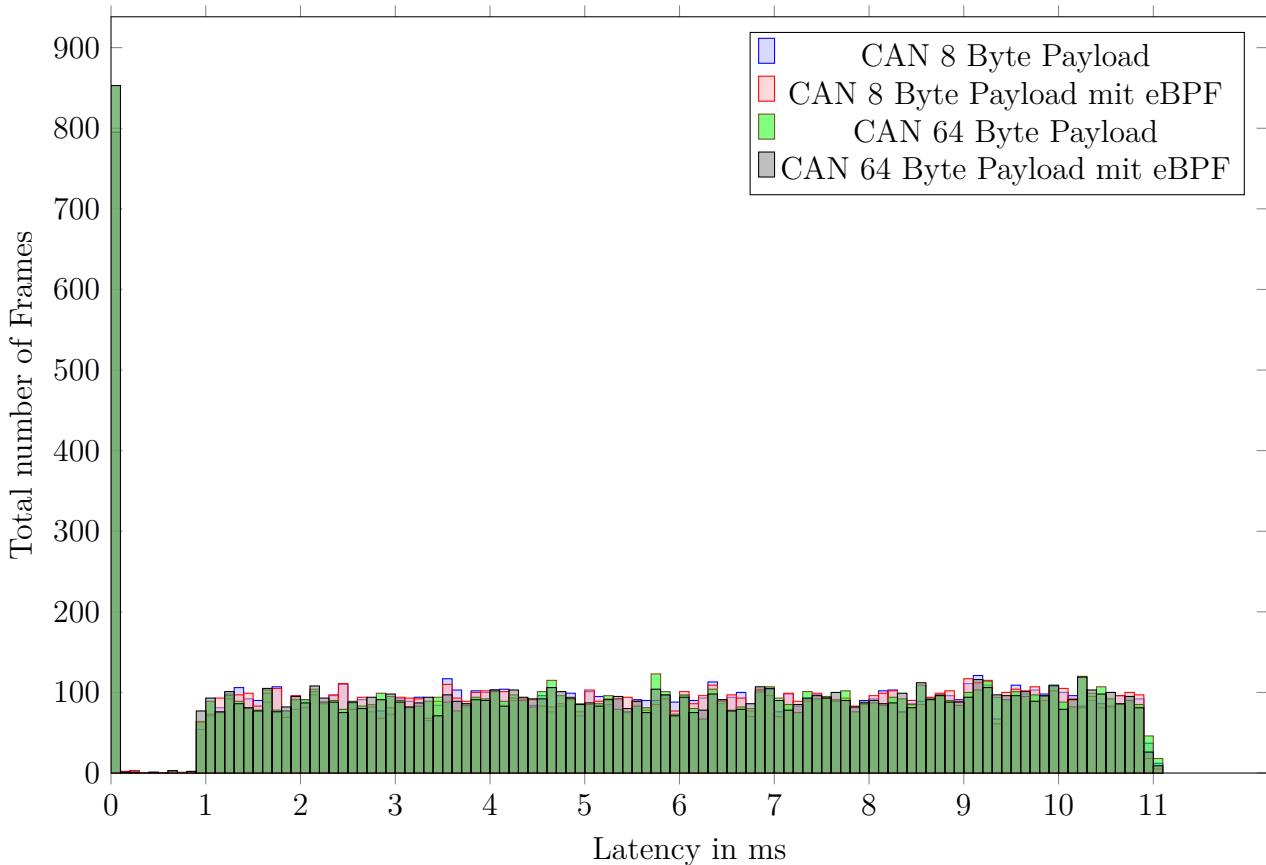


Abbildung 7: Verteilung der Latenzzeiten für CAN nach UDP mit einem 10ms Timeout.

beobachten, dass für 64 Byte große Frames die Latenzen tendenziell etwas niedriger ausfallen als für ihre 8 Byte Äquivalente. Das mittels eBPF niedrigere Latenzen aufgezeichnet worden ist zu erwarten, da eBPF innerhalb des Linux Kernels läuft und sofort beim Erreichen der entsprechenden Anweisungen in die Handler-Funktion springt (siehe Kapitel 4.2.1). Zudem wird bei der Messung über das C-Programm, eine zusätzliche Messlatenz von ca.  $0.065 \mu\text{s}$  eingeführt, da es im Durchschnitt ungefähr so lange dauert, um die beiden Zeitstempel zu nehmen. Die Häufungen um die Mittelwerte sprechen dafür, dass die Latenz relativ konstant ist.

Der zweite Fall, zu sehen in Abbildung 7, zeigt das Verhalten des Moduls bei einem eingestellten Timeout von 10 ms für die Frame Aggregation. Dieser Fall wurde zur Betrachtung ausgewählt, da dieser den Einstellungen des C2E-Gerätes entspricht. Die Abbildung zeigt hier, wie auch beim vorherigen Fall, die Häufigkeitsverteilung der Latenzzeiten. Hierbei ist zu beobachten, dass sich die Latenzen zwar über den gesamten Bereich von 0 ms bis zu 11.1 ms verteilen und dabei die Verteilung zwischen 1 ms und 11 ms relativ uniform ist. Auffällig ist allerdings eine Lücke zwischen 0.1 ms und 0.9 ms und ein großer Ausschlag zwischen 0 ms und 0.1 ms, welcher sich bei näherer Betrachtung, siehe Abbildung 8, sehr ähnlich zu der Verteilung im vorherigen Fall, siehe Abbildung 6, verhält. Diese Erscheinungen hängen damit zusammen, dass der Timeout für die Frame-Aggregation mittels eines Threads umgesetzt wurde, welcher

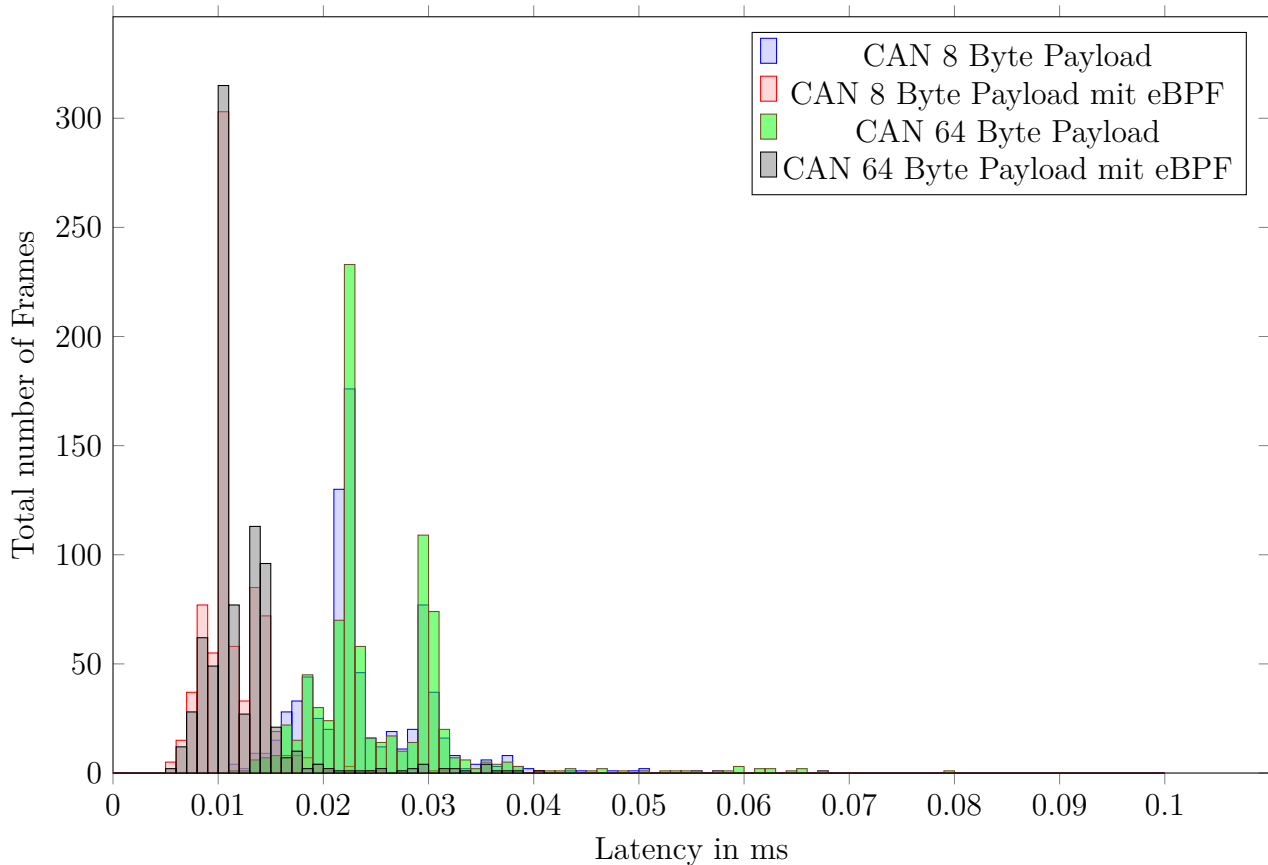


Abbildung 8: Verteilung der Latenzzeiten für CAN nach UDP mit einem 10ms Timeout in dem Bereich von 0 bis 0.1 ms.

überprüft ob der Timeout abgelaufen ist und dann für ca. 1 ms schlafen gelegt wird. Wobei hier, aufgrund des kleinen Zeitintervalls von nur 1 ms, eine Range von 0.9 ms bis 1.1 ms angegeben wurde (siehe Kapitel 3.3). Diese Range führt dazu, dass, abhängig vom Scheduling des Kerns, die einzelnen Millisekunden nicht genau eingehalten werden, wodurch es häufig dazu kommt, dass die eine Überprüfung des Timeouts zwischen 9 ms und 10 ms stattfindet, was dazu führt das erst bei der nächsten Überprüfung, die jetzt erst nach den 10 ms stattfindet, der Timeout als abgelaufen erkannt wird und das Paket gesendet wird. Dies erklärt sowohl Latenzen vom bis zu 11.1 ms, da bei Frames, die am Anfang des Intervalles auftreten, z.B. bei 0 ms, die vorletzte Überprüfung im Worst-Case erst bei 9.99.. ms stattfindet und dann erst wieder nach 1.1 ms, also bei 11.099.. ms. Als auch den Ausschlag zwischen 0 ms und 0.1 ms. Da für Frames, die in dem Bereich nach den 10 ms auftreten, wo der Timeout bereits abgelaufen ist, aber noch nicht als abgelaufen erkannt wird, kein Timeout existiert, da beim Einfügen eines Frames in den Buffer, der Timeout überprüft wird (siehe Kapitel 3.3). Um die Lücke zwischen 0.1 ms und 0.9 ms zu verstehen, wurde die durchschnittliche Zeit die durch `usleep_range` geschlafen wird ermittelt. Dazu wurde ein Zeitstempel jeweils vor und nach dem Aufruf der Funktion im Sender-Thread genommen, die Differenz in  $\mu\text{s}$  berechnet und das Ergebnisse dann mittels `pr_info` ausgegeben. Das Modul lief dann für ca. 60 s, wodurch ca. 6000 Messungen durchgeführt werden konnten. Das Ergebnis hierbei ist, dass im Durchschnitt mit `usleep_range` 1.09 ms geschlafen wird. Dies

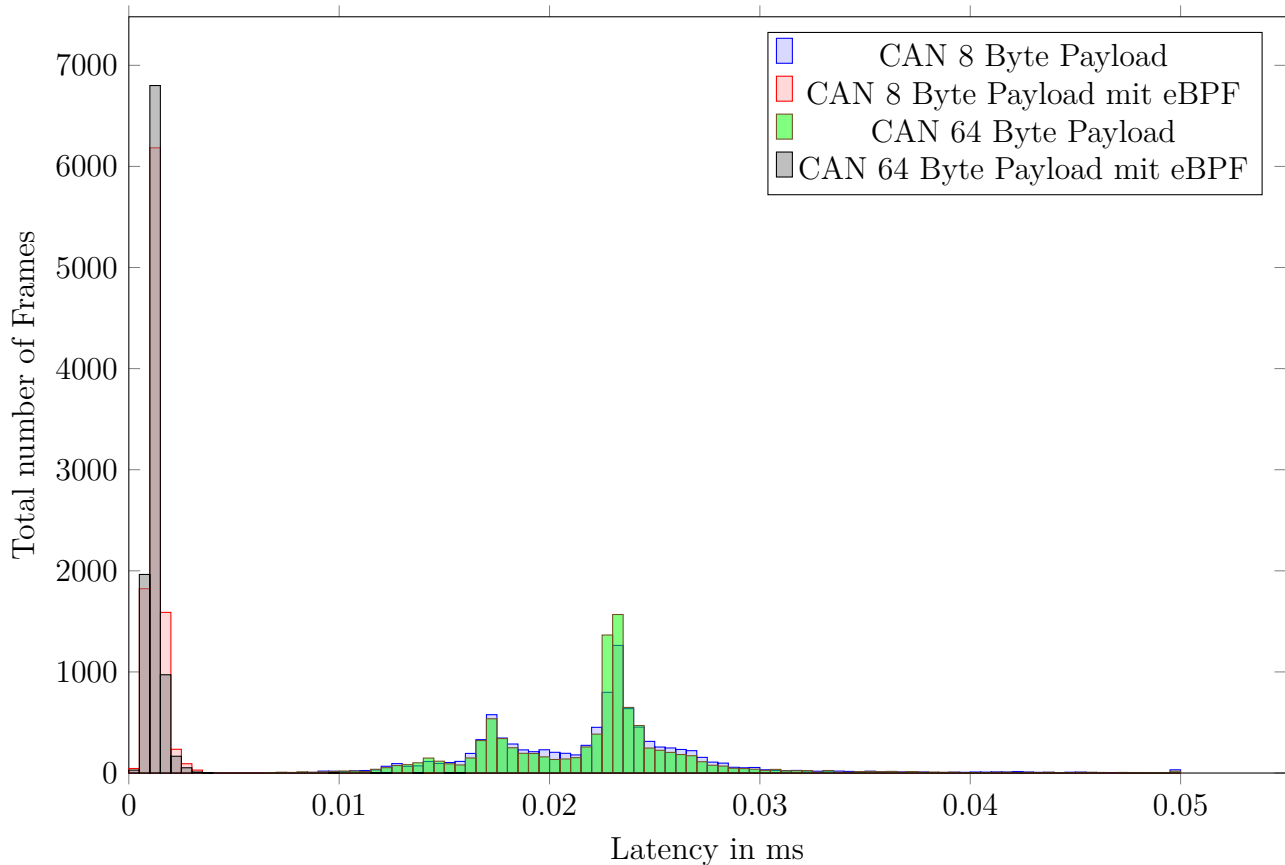


Abbildung 9: Verteilung der Latenzzeiten für UDP nach CAN.

führt dazu, dass meistens bei um die 9.8 ms eine Überprüfung stattfindet und das nächste mal dann erst bei etwa 10.9 ms. Dadurch ist es nahezu unmöglich Latenzen in dem Bereich von 0.1 ms bis 0.9 ms zu erreichen, da diese vor dem Ablauf der 10 ms auftreten müssen, um nicht ohne Timeout zu enden, aber dadurch, dass erst bei 10.9 ms die nächste Überprüfung stattfindet immer eine zusätzliche Latenz von mindestens 0.9 ms erhalten.

**UDP - CAN** Abbildung 9 zeigt die Ergebnisse für die Messung von UDP nach CAN. Dabei ist wieder, wie bei Abbildung 6, ein deutlicher Unterschied zwischen den Messungen die mit *eBPF* gemacht wurden und denen ohne zu erkennen. Der Grund dafür ist, wie in Kapitel 4.2.3 bereits beschrieben, dass *eBPF* im Kernel Space läuft und beim Erreichen der Anweisungen aufgerufen wird. Allerdings fällt hier der Unterschied zwischen den zwei Arten noch stärker aus, da vor allem die mit *eBPF* gemessenen Latenzen alle deutlich niedriger sind als noch in Abbildung 6. Dies ist auch bei den nicht *eBPF* Messungen zu beobachten, allerdings ändert sich hier wenig am unteren Ende der Latenzen, sondern die obere Grenze fällt niedriger aus. Dies führt auch dazu, dass die Latenzen generell weniger gestreut sind als bei CAN-UDP.

## 4.3. Durchsatzrate

### 4.3.1. Aufbau

Bei der Durchsatzrate wurde betrachtet, wie viele CAN-Frames das Modul pro Sekunde verschicken bzw. empfangen kann. Um die Durchsatzrate beim Versenden von CAN-Frames zu ermitteln, wurde entweder ein oder beide Interfaces mithilfe des Tools *CANGEN* "geflooded". *CANGEN* ist Teil von *CAN-utils*, einer Sammlung von Userspace Tools für das Linux CAN Subsystem, *SocketCAN*. [17]

Für die Messung wurden zum einen mehrere Instanzen von *CANGEN*, mit den Optionen *-g 0*, *-m*, *-i* und *-x* gestartet. Die Konfiguration mit *-g 0* bedeutet, dass keine Pause zwischen dem Senden einzelner CAN-Frames liegen soll, bzw. eine Pause von 0 ms. Die Option *-m* bedeutet, dass alle Arten von CAN-Frames zufällig erzeugt werden sollen. Die Option *-i* führt dazu, dass *CANGEN -ENOBUFFS*, *No buffer space available*, Rückgabewerte beim Schreiben auf das Interface ignoriert und *-x* deaktiviert lokales Loopback für generierte Frames. Durch diese Konfiguration sendet *CANGEN* so schnell wie möglich CAN-Frames zu dem Interface.

Bei dem LKM wurde als Zieladresse der Localhost, 127.0.0.1, mit einem freien UDP Port eingestellt und der Standard Timeout von 10 ms für die Frame Aggregation. Daneben wurde auch noch ein Socket auf dem entsprechenden Zielport geöffnet, um zusätzlichen Traffic durch *ICMP* Nachrichten zu verhindern, welcher zusätzliche Last auf dem System verursachen würde und damit die Messungen beeinflussen würde. *ICMP* Nachrichten werden verwendet um Netzwerkschichtinformationen zwischen Hosts und Routern auszutauschen. Unter anderem auch Fehlermeldungen, wenn beispielsweise kein Pfad zur Zieladresse gefunden werden kann [14]. Das Öffnen eines Sockets verhindert dies, da so die Zieladresse gefunden werden kann.

Der CAN-Traffic wird dann von dem Modul in UDP Paketen verschickt. Diese Pakete, sowohl der UDP Traffic, als auch der CAN Traffic pro Interface, wurden jeweils mittels *dumppcap* aufgezeichnet und im Anschluss an die Messung analysiert. *dumppcap* ist Teil von *Wireshark*, einem Netzwerk Paket Analyzer. Es erlaubt den Traffic auf einem Netzwerk aufzuzeichnen und den aufgezeichneten Traffic später zu analysieren [20]. Für die Analyse wurden die aufgezeichneten Pakete entsprechend gefiltert, sodass nur UDP Frames, die an den entsprechenden Port auf Localhost gesendet wurden, bei der Analyse betrachtet wurden. Dadurch ist es dann möglich die UDP Pakete leicht zu analysieren und die Zahl der CAN-Frames pro Paket und somit auch die aller Pakete, die in einer Sekunde kamen, zu erfassen. Da während der Messung eine große Menge an Paketen in sehr kurzer Zeit aufzuzeichnen sind, konnten diese nicht normal aufgezeichnet werden, da die Schreibzugriffe während dem Aufzeichnen zu lange dauerten. Deshalb musste zusätzlich noch mithilfe von *tmpfs* [21] ein virtuelles Dateisystem im RAM angelegt werden, um von den deutlich höheren Schreibgeschwindigkeiten von RAM zu profitieren, da diese ca. 2 - 3-mal schneller sind als die von der Harddisk. Damit konnten dann auch alle Pakete aufgezeichnet werden.

Für die Ermittlung der Durchsatzrate beim Empfangen von CAN-Frames in UDP Paketen, wurden mehrere Instanzen eines Pythonskripts gestartet, welches C2E-UDP Pakete mit zu-



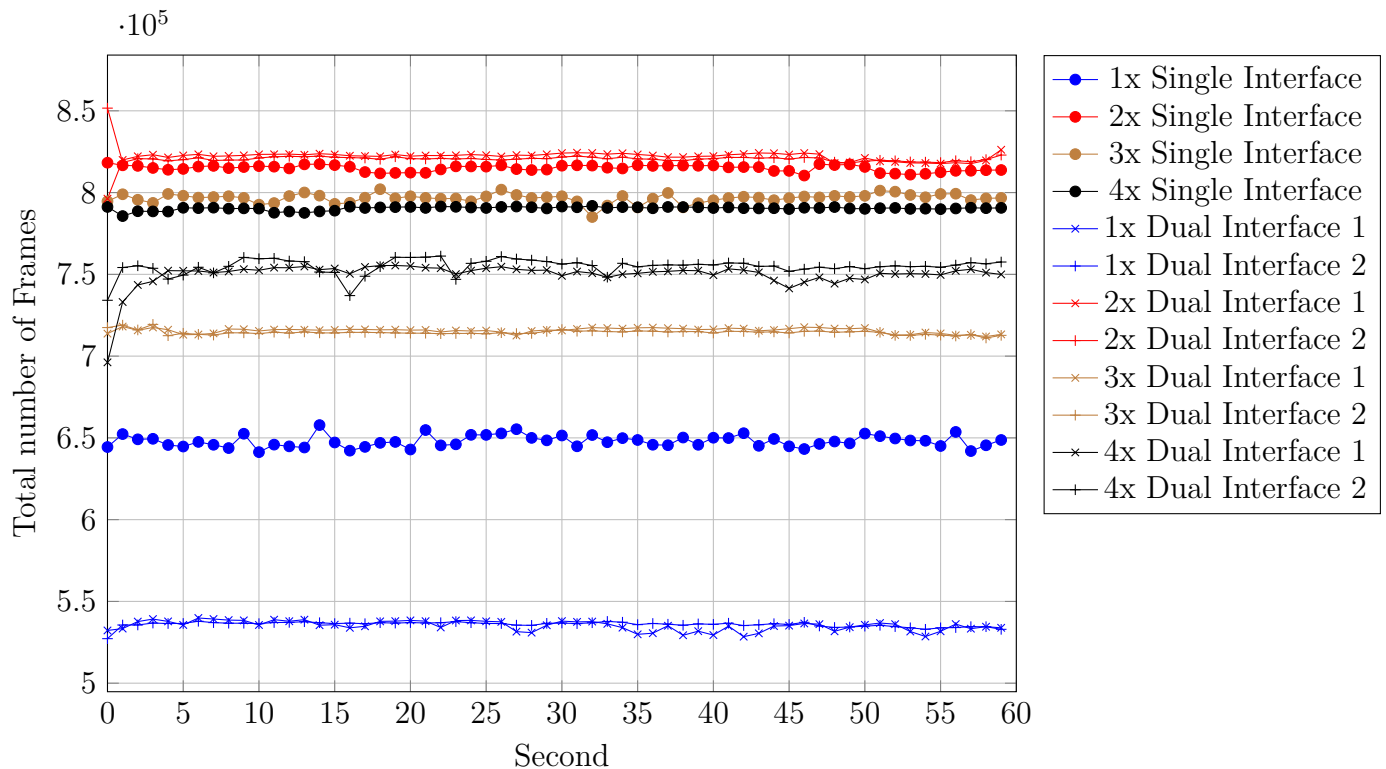


Abbildung 10: Durchsatzrate für CAN - UDP

fällig generierten CAN-Frames füllt und an das Modul schickt. Mithilfe von *dumppcap* wurden dann die CAN-Frames an den Interfaces aufgezeichnet und abgezählt, wie viele CAN-Frames pro Sekunde ankamen.

Bei beiden Verfahren wurde jeweils eine Minute lang Traffic erzeugt, aufgezeichnet und analysiert. Zudem wurde bei allen hier gezeigten Messungen auch gleichzeitig immer überprüft, ob die Anzahl der CAN-Frames in den UDP Paketen, mit der an den CAN-Interfaces übereinstimmt. Dies war so weit nicht anders gekennzeichnet bei allen diesen Messungen der Fall.

#### 4.3.2. Messungen

**CAN - UDP** In Abbildung 10 wird die, über die eine Minute gemessene, Durchsatzrate abgebildet. Dabei entspricht jeder Messpunkt, dem Durchsatz in der entsprechenden Sekunde. In der Abbildung sind dabei sowohl die Messungen für den Durchsatz eines einzelnen Interfaces und als auch die von zwei Interfaces, die gleichzeitig Frames empfangen. Es ist zu erkennen, dass die obere Grenze für den Durchsatz bei ca. 815000 Frames pro Sekunde liegt und mit mehr *CANGEN* Instanzen, der Durchsatz wieder abfällt. Des Weiteren ist zu sehen, dass die Leistung von zwei Interfaces zueinander nahezu identisch ist, aber der individuelle Durchsatz im dualen Betrieb generell unter dem des äquivalenten Single-Betriebes liegt. Zudem ist die Durchsatzrate die gesamte Minute über relative konstant und hat nur geringe Schwankungen. Der Grund für den Abfall, nach zwei *CANGEN* Instanzen, liegt daran, dass ab dem Punkt die

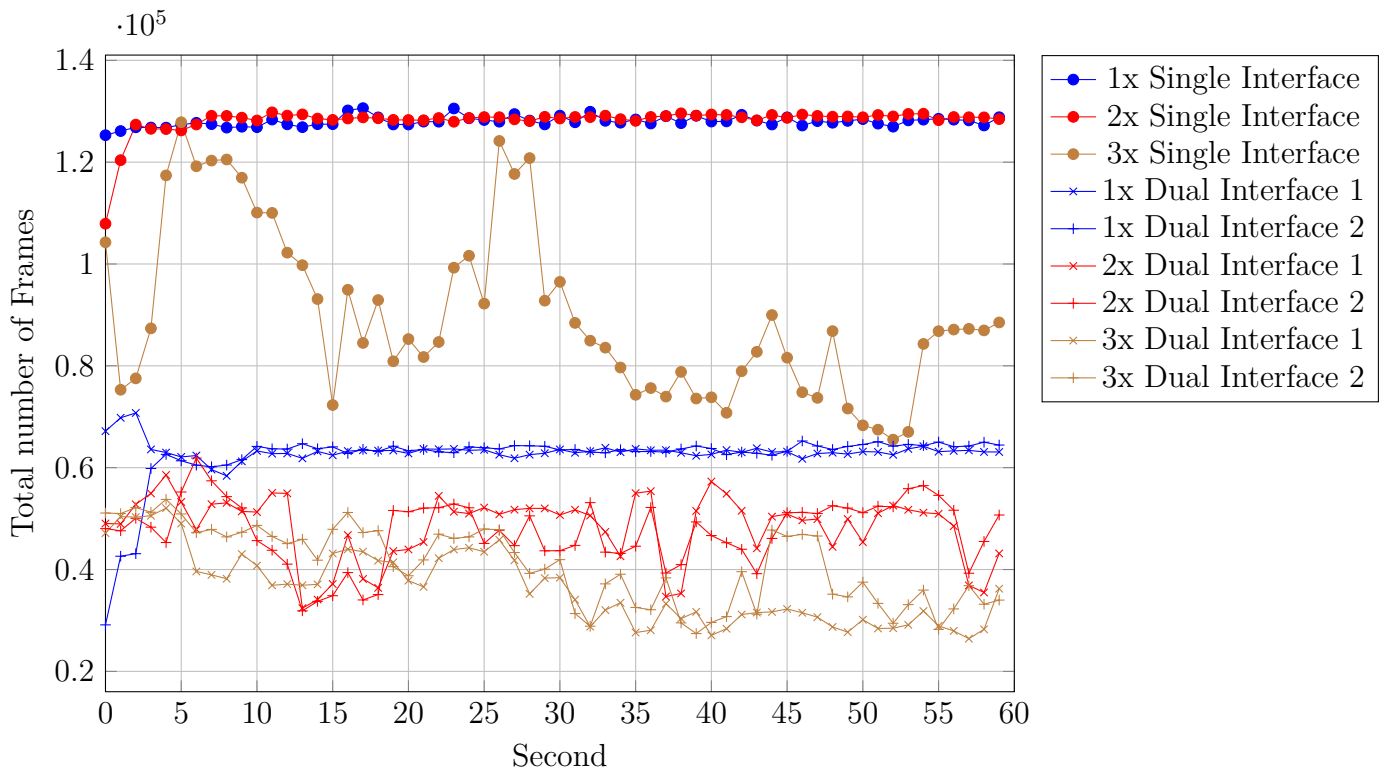


Abbildung 11: Durchsatzrate für UDP - CAN

Zahl der am CAN-Interface ankommenden Frames anfängt zu stagnieren, bzw. abzufallen. Das die einzelne Leistung im dualen Betrieb generell unter der im Single Betrieb liegt, ist darauf zurückzuführen, dass die beiden Interfaces sich einen Buffer teilen (siehe Kapitel 3.3) und daher der Zugriff auf den Buffer mit Locks synchronisiert wird. Dies führt dazu das die Schreiboperationen eines Interfaces regelmäßigen auf das Beenden der des anderen warten müssen, was zu einer allgemeinen Verzögerung beim Schreiben führt.

**UDP - CAN** Wie zuvor auch bildet Abbildung 11 die Durchsatzrate pro Sekunde ab. Dabei wird wieder zwischen der Durchsatzrate für ein einzelnes Interface im Single und dual Betrieb unterschieden. In der Abbildung ist zu sehen, dass für Interfaces im Single Betrieb das Limit bei ca. 130000 CAN-Frames pro Sekunde und im dualen Betrieb bei ziemlich genau der Hälfte, also ca. 65000 CAN-Frames pro Sekunde liegt. Des Weiteren ist zu erkennen, dass im Single Betrieb ab drei Threads die senden, die Durchsatzrate erheblich einbricht und unregelmäßige Ausschläge aufweist. Für Interfaces im dualen Betrieb tritt dies schon ab zwei Threads pro Interface auf. Das ein Interface im dualen Betrieb maximal die halbe Durchsatzrate hat, die es im Single Betrieb hätte, ist zu erwarten, da die Kommunikation über IP über von den Interfaces geteilten Ressourcen läuft (siehe Kapitel 3.3). Dadurch existiert für das Modul eine Gesamtdurchsatzrate, welche dadurch beschränkt ist, wie viele UDP Pakete das Modul insgesamt pro Sekunde abarbeiten kann. Da dies unabhängig von dem Interface ist, für welches das Frame ist, muss diese Gesamtdurchsatzrate von allen vom Modul bereitgestellten Interfaces geteilt werden.

### 4.3.3. Paketverlust

Da, wie bereits bei der Betrachtung der Durchsatzrate Kapitel 4.3 erwähnt, trotz Flooding keine Diskrepanz zwischen der Anzahl der CAN-Frames auf beiden Seiten festzustellen war, konnte kein Paketverlust von dem Modul festgestellt werden.

### 4.3.4. Fazit

Die Betrachtungen der Durchsatzrate haben ergeben, dass das Modul in der Lage ist über die CAN-Interfaces mehrere Hunderttausend Frames pro Sekunde und über UDP mehrere Zehntausend Frames pro Sekunde zu bearbeiten.

In einem realen Szenario können an einem CAN-Bus der mit 1 Mbps überträgt im Idealfall maximal nur ca. 21.276 Frames pro Sekunde übertragen werden. Da ein CAN-Frame am Bus mindestens 47 Bit lang ist und  $\frac{1.000.000\text{bps}}{47\text{bit}} \approx 21.276$ . Die 47 Bit ergeben sich daraus, dass ein kleinst möglicher Frame mit dem SOF Bit beginnt, gefolgt von den 11 Bit Identifier, dem RTR/SRR Bit, IDE Bit, r0 Bit, vier DLC Bits, keinen Data Bits, 16 CRC Bits, zwei ACK Bits, sieben EOF Bits und mindestens drei IFS Bits (siehe Kapitel 2.1). Zusammen macht das 47 Bit.

Wie in Kapitel 4.3.2 gezeigt stellt diese Anzahl an Frames kein Problem für das Modul dar, es sollte daher in der Lage sein, jeden physikalischen CAN-Bus zu handhaben. Vorallem da in einem echten Bus die Frames länger sind als in der oben betrachteten Rechnung, da in dieser keine Extended Frames oder überhaupt Payloads vorkamen.

## 5. Leistung des Moduls im Vergleich

Um zu etablieren, ob das Modul eine Verbesserung gegenüber bereits existierenden Lösungen bietet, wird es im folgenden mit den Protokollen *Cannelloni* und *Socketcand* anhand der in Kapitel 4.1 definierten Parametern verglichen.

### 5.1. Cannelloni

*Cannelloni* ist eine Userland Anwendung, die das Senden von CAN-Frames mittels UDP, TCP oder SCTP ermöglicht. Dabei unterstützt es Übertragungsraten von über 10 Mbit/s, CAN FD-Frames und Frame Aggregation in Ethernet Frames. Allerdings übermittelt *Cannelloni* nicht gesondert die Zeitstempel der einzelnen Frames [18]. *Cannelloni* erfüllt damit zwar von den in Kapitel 3.1 definierten Anforderungen Punkt 1, die Unterstützung mehrerer unabhängiger CAN Interfaces, auch wenn hier für jedes Interface Paar ein eigener Tunnel aufgemacht werden muss. Und es erfüllt auch Punkt 2, Frame Aggregation für schnell aufeinander folgende Frames, zumindest für UDP, welches aber auch die Netzwerk-Technologie ist, die von C2E verwendet wird. Jedoch erfüllt es nicht Punkt 3, das Übergeben von CAN-Frames mit ihren Zeitstempeln an den Benutzer, da bei *Cannelloni* Zeitstempel nicht Teil des Protokolls sind. Das ist auch der Grund, warum *Cannelloni* für den gesuchten Verwendungszweck nicht geeignet ist. Da *Cannelloni* keine Zeitstempel überträgt, ist ein Messen von Nachrichten am Bus von Endgeräten aus nicht möglich.

### 5.2. Socketcand

*Socketcand* ist ein daemon, welcher Zugriff auf CAN-Interfaces auf einer Maschine über Netzwerk Interfaces ermöglicht. Es wird dabei eine TCP/IP Verbindung zusammen mit einem eigenen ASCII-basierten Protokoll [15] verwendet [19].

Die Hauptprobleme mit *Socketcand* sind, dass zum einen keine Frame Aggregation für schnell aufeinanderfolgende CAN-Frames existiert. Und zum anderen ist ein ASCII-Protokoll für den Einsatz im Embedded Bereich unhandlich. Darüber hinaus unterstützt *Socketcand* keine CAN-FD-Frames.

### 5.3. Latenzzeit

Bei der Latenzzeit von *Cannelloni* und *Socketcand* wurden wieder, wie bei dem Modul (siehe Kapitel 4.2), die beiden Richtungen CAN-UDP und UDP-CAN getrennt voneinander betrachtet. Für die Messungen wurde ein virtuelles CAN Interface (*vcan*) angelegt. Dazu wurde *Cannelloni* mit dem *vcan* Interface als zu beobachtendes CAN Interface gestartet und, wie auch bei der Messung für das Modul (siehe Kapitel 4.2), ist *localhost* die Zieladresse für die UDP Pakete. Wenn UDP als Transportprotokoll verwendet wird, unterstützt *Cannelloni* auch Frame Aggregation. Allerdings lässt sich der dafür verwendete Timeout nur im Mikrosekundenbereich einstellen und nicht völlig ausstellen. Für die Messungen wurde daher sowohl bei *Cannelloni*,

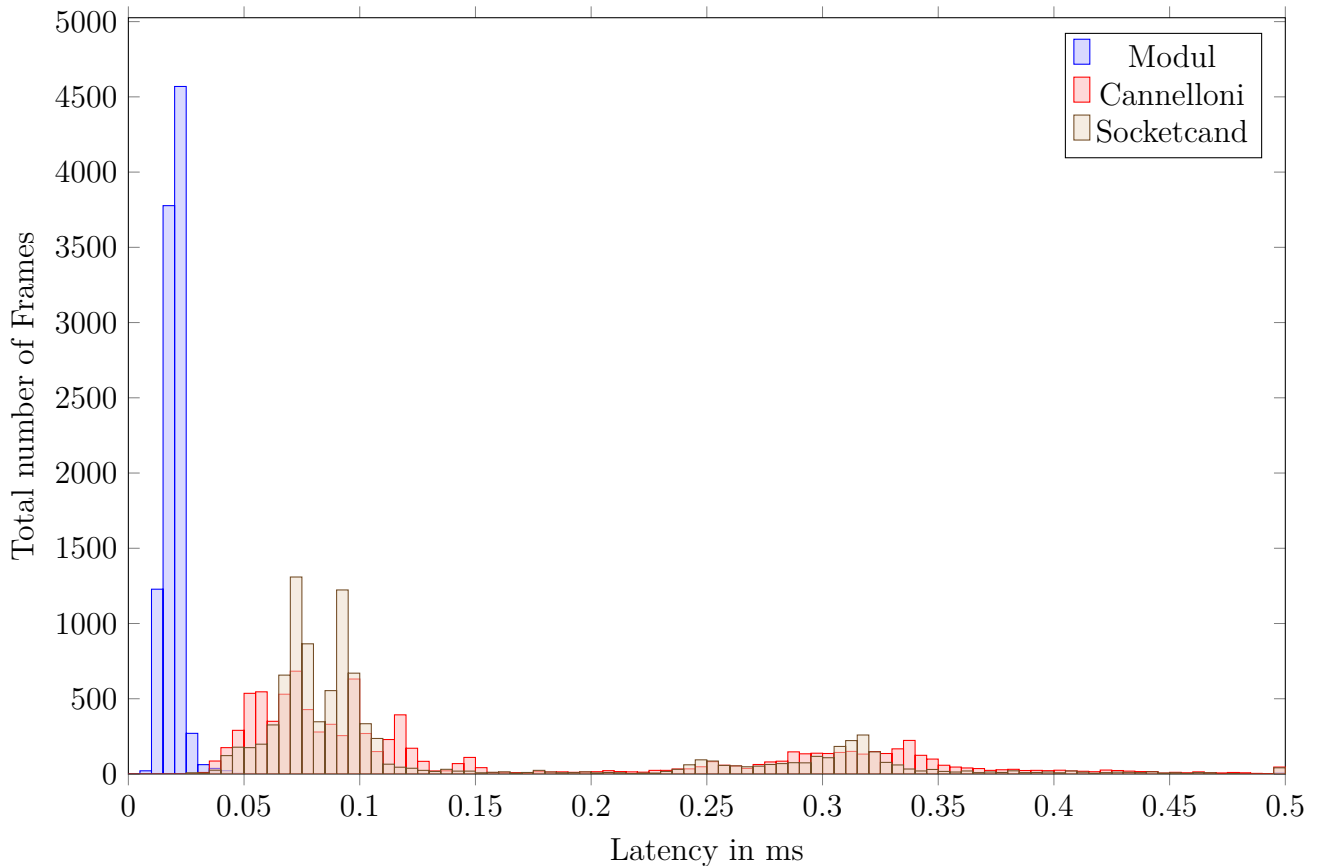


Abbildung 12: Latenz CAN-UDP: Die gemessenen Latenzen von *Cannelloni* und *Socketcand* gegen die des Moduls. Sowohl *Cannelloni* als auch das Modul haben hierbei einen Timeout von  $1\ \mu\text{s}$ , *Socketcand* hat keinen, da es keine Frame-Aggregation unterstützt.

als auch bei der Messung der Vergleichswerte des Moduls, der bei *Cannelloni* kleinst mögliche Timeout von  $1\ \mu\text{s}$  gewählt. Bei *Socketcand* wurde der *Socketcand*-Server auf dem vcan Interface gestartet, von dem Programm aus wurde ein TCP Socket geöffnet und die Verbindung zu dem Server hergestellt. Da *Socketcand* CAN-FD nicht unterstützt, wurden für bei der Messung nur klassische CAN-Frames mit einer 8 Byte großen Payload verwendet. Das weitere Vorgehen entspricht dem bereits in Kapitel 4.2.2 beschriebenen. Allerdings wurde für den Vergleich nur mittels des C-Programmes gemessen und nicht mit *eBPF*, da hier nur der Vergleich zwischen den beiden Protokollen relevant ist und durch die höhere Präzision von *eBPF* keine zusätzlichen Informationen gewonnen werden können.

Abbildung 12 und Abbildung 13 zeigen jeweils die Häufigkeitsverteilung der gemessenen Latenzen für das Modul, *Cannelloni* und *Socketcand* für die jeweiligen Richtungen. Dabei ist für beide Richtungen zu erkennen, dass das Modul generell eine niedrigere Latenz als *Cannelloni* oder *Socketcand* hat. Dies ist besonders deutlich für die Richtung CAN-UDP (Abbildung 12). Des Weiteren liegen bei der Richtung CAN-UDP die Latenzen für das Modul deutlich enger zusammen, ca. zwischen 0,005 ms und 0,045 ms, als bei *Cannelloni* oder *Socketcand*, bei denen

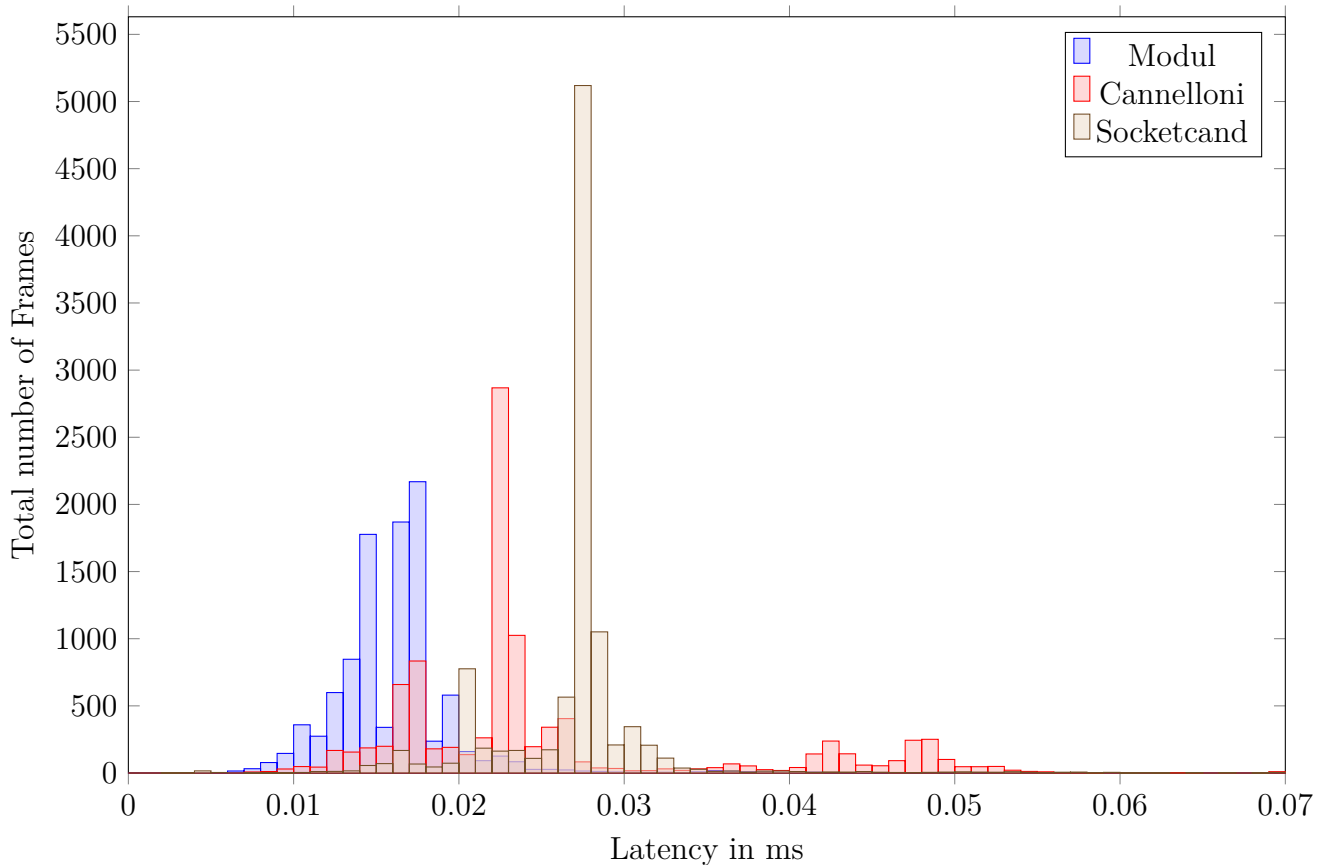


Abbildung 13: Latenz UDP-CAN: Die gemessenen Latenzen von *Cannelloni* und *Socketcand* gegen die des Moduls

sich die Latenzen von ca. 0.35 ms bis über 0.5 ms erstrecken.

Für die Richtung UDP-CAN fällt der Unterschied zwischen dem Modul und den beiden Protokollen deutlich geringer aus. Hier hat das Modul im Durchschnitt eine Latenz von ca. 0.016 ms, allerdings ist es damit nicht weit von *Cannelloni*, mit einer durchschnittlichen Latenz von ca. 0.026 ms, und *Socketcand*, ca. 0.027 ms, entfernt. Zudem verteilen sich hier die Latenzen beim Modul und *Socketcand* zwar über einen ähnlich großen Bereich, aber bei beiden liegen sie deutlich enger zusammen als bei *Cannelloni*.

## 5.4. Durchsatzrate

Die Durchsatzraten von *Cannelloni* und *Socketcand* wurden fast genauso wie die des Moduls (siehe Kapitel 4.3) ermittelt. Der einzige Unterschied liegt darin, dass die Option *-x* bei *CAN-GEN* weggelassen werden musste, da sonst keines der beiden Protokolle CAN-Frames registriert. Die Option *-x* deaktiviert den lokalen Loopback von generierten Events, weshalb durch das Weglassen dieser Option alle CAN-Frames „doppelt“ aufgezeichnet wurden. Einmal der eigentliche Frame und einmal der Loopback. Dies wurde bei den folgenden Messungen berücksichtigt und die Messung des Moduls wurde nochmal ohne die Option *-x* für die Richtung CAN-UDP durchgeführt. Für die Richtung UDP nach CAN wurden bei *Cannelloni*, genau wie

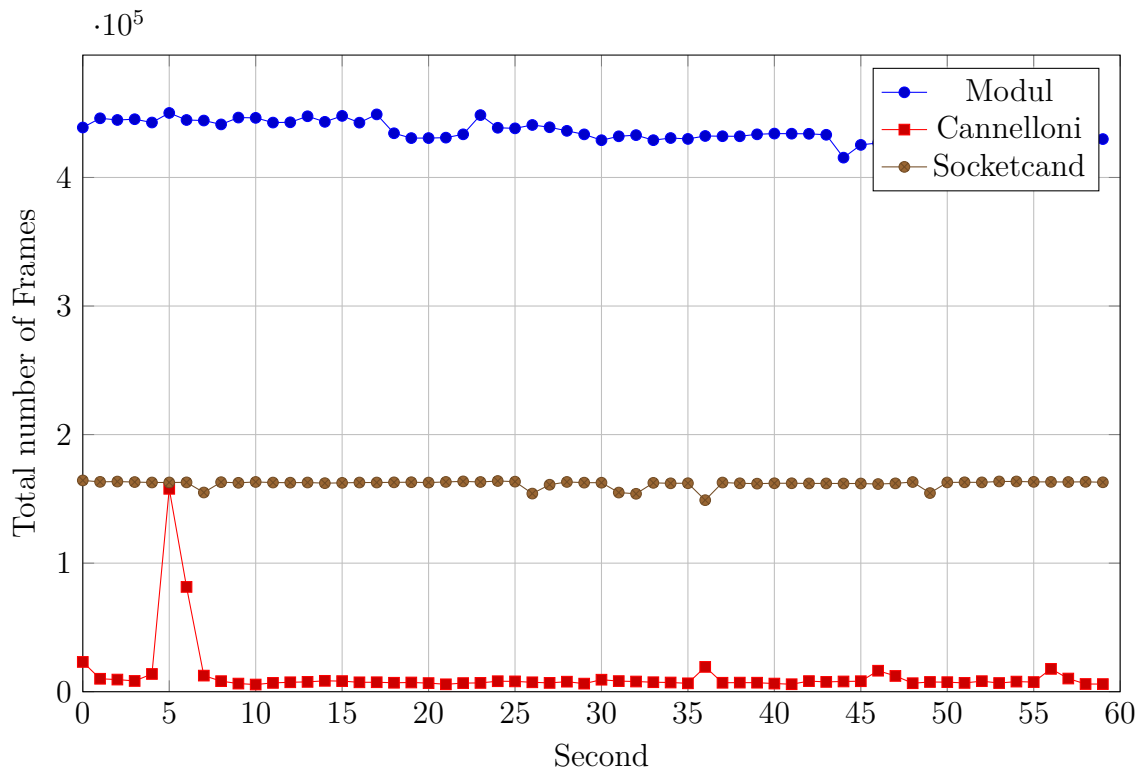


Abbildung 14: Durchsatzrate CAN-UDP: Der gemessene Durchsatz pro Sekunde von *Cannelloni* und *Socketcand* gegen den des Moduls

beim Modul, UDP Frames von maximal 1500 Byte verwendet. Bei *Socketcand* wurde entsprechend dem Protokoll [15] immer ein klassischer CAN-Frame pro TCP Paket gesendet.

In sowohl Abbildung 14 als auch Abbildung 15 ist zu sehen, dass die Durchsatzrate des Moduls deutlich über der von *Cannelloni* und *Socketcand* liegt. Besonders groß ist der Unterschied bei der Richtung CAN-UDP, wo *Cannelloni* im Durchschnitt gerade einmal einen Durchsatz von ca. 12.000 CAN-Frames hat, im Gegensatz zu dem Modul mit einem durchschnittlichen Durchsatz von ca. 430.000 CAN-Frames pro Sekunde. Damit erreicht *Cannelloni* nur ca. 2,8% des Durchsatzes des Moduls. Des Weiteren ist in Abbildung 14, dass mit dem „Echo“ der CAN-Frames auch die allgemeine Zahl der CAN-Frames an den CAN-Interfaces abnimmt, da auch die Durchsatzrate für das Modul unter dem liegen, was sie ohne das „Echo“ erreichen kann (vgl. Abbildung 10). Bei der Richtung UDP-CAN fällt der Unterschied deutlich geringer aus, hier hat das Modul im Durchschnitt eine Durchsatzrate von ca. 130.000 CAN-Frames pro Sekunde, wohingegen *Cannelloni* eine durchschnittliche Rate von ca. 110.000 CAN-Frames pro Sekunde hat. Der Unterschied der zwischen dem Modul und *Cannelloni* in Abbildung 15 auftritt ist nicht auf Paketverluste zurückzuführen, sondern vielmehr darauf, dass das Skript welches für das Senden von *Cannelloni* UDP-Paketen verwendet wurde etwas langsamer läuft als das, welches für das Modul verwendet wurde.

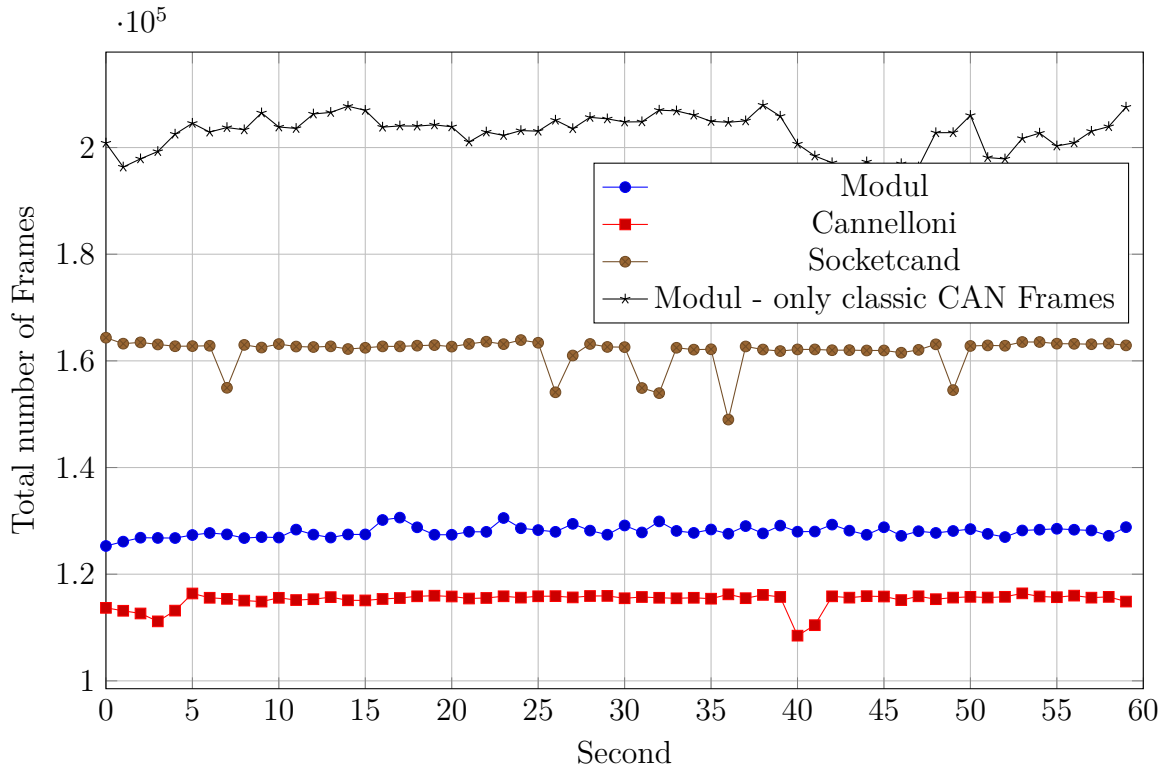


Abbildung 15: Durchsatzrate UDP-CAN: Der gemessene Durchsatz pro Sekunde von *Cannelloni* und *Socketcand* gegen die des Moduls

### 5.5. Paketverlust

Der Paketverlust wurde anhand der bei der Durchsatzmessung ermittelten Daten ausgewertet. Dabei wurde betrachtet, wie viele CAN-Frames, direkt über das Interface oder via UDP, an *Cannelloni* bzw. *Socketcand* gesendet und wie viele davon weitergeleitet wurden.

Abbildung 16 zeigt den Anteil der von *Cannelloni* weitergeleiteten Frames bei Empfangen über das CAN-Interface. Im Schnitt kamen pro Sekunde ca. 410.000 CAN-Frames an, von denen wurden durch *Cannelloni* allerdings nur etwa 12.000 verarbeitet. Also nur etwa 2,9%.

Abbildung 17 zeigt die Verluste bei *Socketcand*. Hier kamen auch wieder ca. 410.000 CAN-Frames pro Sekunde an, von denen wurden dann aber nur ca. 120.000 verarbeitet. Damit wurden von *Socketcand* nur ca. 29% der Frames weitergeleitet.

Beim Empfangen über UDP konnte bei keinem der beiden Protokollen Verluste festgestellt werden.



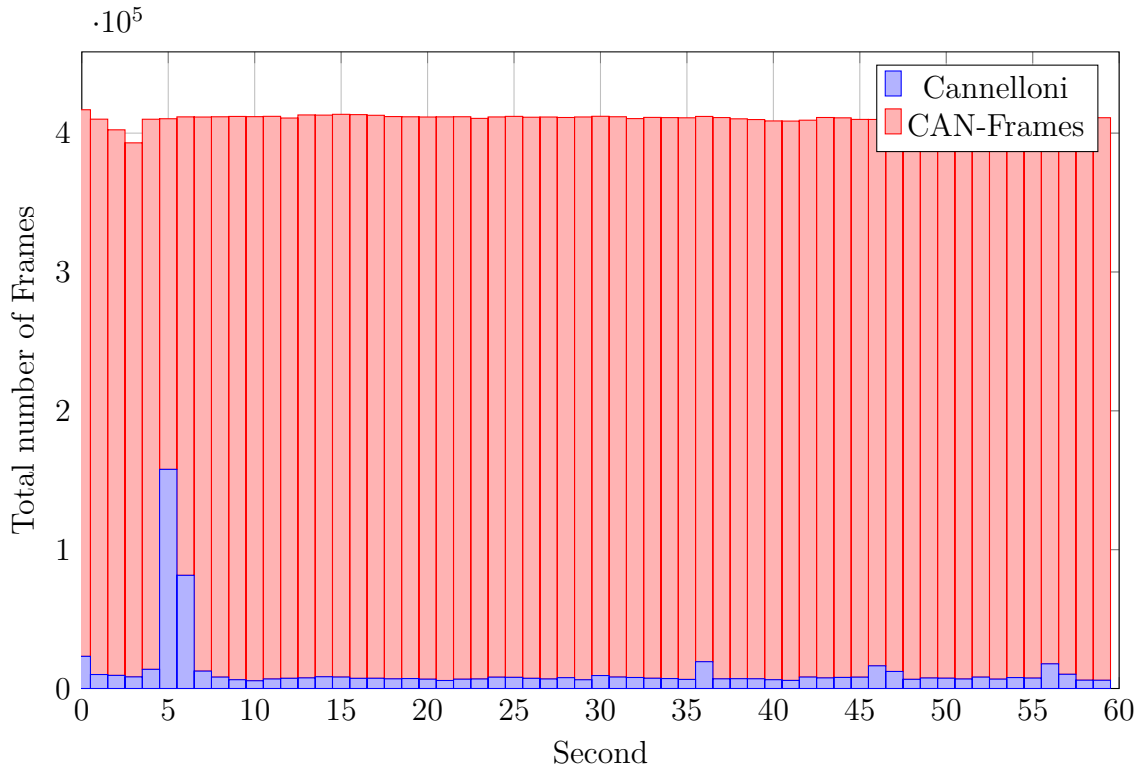


Abbildung 16: *Cannelloni* Paketverluste CAN-UDP: Anzahl der gesendeten CAN-Frames gegen die von *Cannelloni* weitergeleiteten.

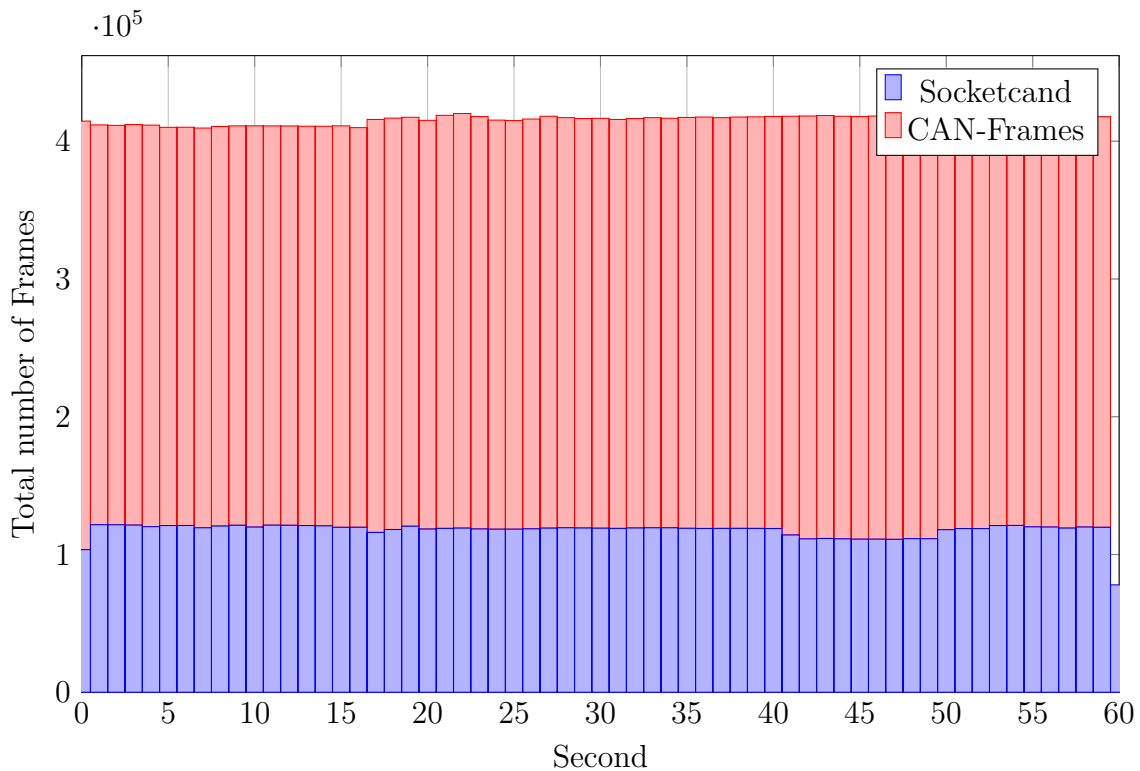


Abbildung 17: *Socketcand* Paketverluste CAN-UDP: Anzahl der gesendeten CAN-Frames gegen die von *Socketcand* weitergeleiteten.

## 6. Stabilitätstests

### 6.1. Methodik

Um die Stabilität des Moduls zu testen wurde das Modul über längere Zeit mithilfe von *scapy* "gefuzzed". *Fuzzing* ist eine Black Box Methode zum Testen von Software, bei der Bugs durch das automatisierte Injizieren von fehlerhaften bzw. teilweise fehlerhaften Daten gefunden werden. Der Vorteil von *fuzzing* ist, dass das Testdesign extrem simpel gehalten werden kann, ohne dabei viele Vorannahmen über das System zu treffen [12]. Für das *Fuzzing* des Moduls wurde hauptsächlich die UDP-CAN Richtung betrachtet, da das Empfangen der CAN-Frames über die Linux-API abläuft und *fuzzing* hier nur die Linux-API testen würde, nicht aber das Modul selbst.

Um das *Fuzzing* zu ermöglichen, wurden zuerst mittels *scapy* jeweils Definitionen für den C2E-Protokoll Header und den C2E-Chunk angelegt. Daraufhin wurden aus diesen Definitionen C2E-Pakete mit generierten Werten für die einzelnen Felder erzeugt. Um etwas gezielter zu testen, wurde bei bestimmten Feldern, vor allem Konstanten, die Wahrscheinlichkeit dafür, dass sie einen korrekten Wert enthalten erhöht. So wurde die Wahrscheinlichkeit das der Header die korrekte *Magic Number* hat auf 50 % und die Wahrscheinlichkeit das der Chunk Type ein CAN-Frame indiziert auf 25 % erhöht.

### 6.2. Auswertung

Nachdem das Modul für mehr als zwei Stunden lang "gefuzzed" wurde, wurde mittels *journalctl* der Systemlog seit dem letzten Boot auf Fehler in Bezug auf das Modul analysiert, um festzustellen, ob durch das Modul unerwartete Fehler oder Bugs aufgetreten sind.

Diesbezüglich konnte nichts gefunden werden.

## 7. Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es ein Linux Kernel Modul zu entwickeln, mit dem CAN-Schnittstellen transparent über IP-Netzwerke getunnelt werden können. Das Modul soll damit eine zuverlässige und effiziente Alternative zu *Cannelloni* und *Socketcand* darstellen. Insbesondere soll das Modul die Übertragung der Zeitstempel der einzelnen CAN-Frames ermöglichen, um dadurch präzise Messungen am CAN-Bus durchzuführen.

Dazu wurden zu Beginn der Arbeit erstmal die einzelnen Anforderungen, die das Modul erfüllen muss, herausgestellt (Kapitel 3.1), gefolgt von der Vorstellung der Entwicklungsumgebung (Kapitel 3.2) und den Details der Implementierung (Kapitel 3.3). Die zentrale Frage der Arbeit war, ob es sich bei dem Modul tatsächlich um eine „bessere“ Lösung handelt, oder ob ein einfaches Anpassen von beispielsweise *Cannelloni* genauso gut funktionieren würde. Um zu testen, ob es sich bei dem Modul um eine effizientere Lösung handelt, wurde es mit den beiden bereits existierenden Alternativen, *Cannelloni* und *Socketcand*, verglichen (Kapitel 5). Die Ergebnisse dieser Untersuchungen zeigen, dass das Modul in vielen Aspekten, besonders bezogen auf den Paketverlust an den CAN-Interfaces, eine deutlich effizientere Alternative bietet. Zudem zeigen die Stabilitätstests (Kapitel 6), zusammen mit den nicht existierenden Paketverlusten, weder an den CAN-Interfaces, noch über UDP, dass es sich durchaus bei dem Modul um eine zuverlässige Lösung handelt.

Eine zukünftige Entwicklung des Moduls könnte in einer besseren Integration des Moduls in der Linux Environment. So könnte die Konfiguration des Moduls in der Zukunft über *iproute2* stattfinden und das Hinzufügen bzw. Entfernen von mehreren Instanzen virtueller C2E-Geräten ähnlich zu anderen Netzwerk-Interfaces, wie z.B. *vcan*, erfolgen.

## 8. Anhang

Tabelle 1: Latenzzeiten CAN - UDP

Messung	0ms			10ms		
	min.	mean	max.	min.	mean	max.
CAN 8 Byte Payload	10.7 $\mu$ s	31.1 $\mu$ s	82.2 $\mu$ s	0.0112 ms	5.5 ms	11.1 ms
CAN-FD 64 Byte Payload	10.3 $\mu$ s	27.6 $\mu$ s	91.8 $\mu$ s	0.0112 ms	5.5 ms	11.4 ms
CAN 8 Byte Payload mit eBPF	5.0 $\mu$ s	14.8 $\mu$ s	46.5 $\mu$ s	0.0051 ms	5.5 ms	11.0 ms
CAN-FD 64 Byte Payload mit eBPF	4.7 $\mu$ s	13.2 $\mu$ s	44.2 $\mu$ s	0.0052 ms	5.5 ms	11.1 ms

Tabelle 2: Latenzzeiten UDP - CAN

Messung	min.	mean	max.
CAN 8 Byte Payload	7.8 $\mu$ s	22.2 $\mu$ s	123.4 $\mu$ s
CAN-FD 64 Byte Payload	6.9 $\mu$ s	22.0 $\mu$ s	90.9 $\mu$ s
CAN 8 Byte Payload mit eBPF	0.35 $\mu$ s	1.3 $\mu$ s	14.0 $\mu$ s
CAN-FD 64 Byte Payload mit eBPF	0.37 $\mu$ s	1.3 $\mu$ s	15.5 $\mu$ s

## I. Abbildungsverzeichnis

1.	Aufbau eines Standard CAN-Frames [10] . . . . .	4
2.	Aufbau eines Extended CAN-Frames [10] . . . . .	4
3.	Ein Standard CAN-Frame im Vergleich zu einem CAN-FD-Frame mit 11 Bit Identifier. [11] . . . . .	5
4.	Schematischer Aufbau des C2E Headers gefolgt von mehreren „Chunks“, wie beispielsweise CAN-Frames . . . . .	6
5.	Aufbau eines <i>C2E-Chunks</i> . . . . .	7
6.	Verteilung der Latenzzeiten für CAN nach UDP ohne Timeout. . . . .	15
7.	Verteilung der Latenzzeiten für CAN nach UDP mit einem 10ms Timeout. . . . .	16
8.	Verteilung der Latenzzeiten für CAN nach UDP mit einem 10ms Timeout in dem Bereich von 0 bis 0.1 ms. . . . .	17
9.	Verteilung der Latenzzeiten für UDP nach CAN. . . . .	18
10.	Durchsatzrate für CAN - UDP . . . . .	20
11.	Durchsatzrate für UDP - CAN . . . . .	21
12.	Latenz CAN-UDP: Die gemessenen Latenzen von <i>Cannelloni</i> und <i>Socketcand</i> gegen die des Moduls. . . . .	24
13.	Latenz UDP-CAN: Die gemessenen Latenzen von <i>Cannelloni</i> und <i>Socketcand</i> gegen die des Moduls . . . . .	25
14.	Durchsatzrate CAN-UDP: Der gemessene Durchsatz pro Sekunde von <i>Cannelloni</i> und <i>Socketcand</i> gegen den des Moduls . . . . .	26
15.	Durchsatzrate UDP-CAN: Der gemessene Durchsatz pro Sekunde von <i>Cannelloni</i> und <i>Socketcand</i> gegen die des Moduls . . . . .	27
16.	<i>Cannelloni</i> Paketverluste CAN-UDP: Anzahl der gesendeten CAN-Frames gegen die von <i>Cannelloni</i> weitergeleiteten. . . . .	28
17.	<i>Socketcand</i> Paketverluste CAN-UDP: Anzahl der gesendeten CAN-Frames gegen die von <i>Socketcand</i> weitergeleiteten. . . . .	28

## II. Tabellenverzeichnis

1.	Latenzzeiten CAN - UDP . . . . .	31
2.	Latenzzeiten UDP - CAN . . . . .	31

### III. Abkürzungsverzeichnis

**ACK** Acknowledgment. 5

**C2E** CAN2ETH. I, 1, 2, 4–10, 13, 14, 16, 19, 23, 29, 30

**CAN** Controller Area Network. I, 1, 2, 4–10, 12–15, 18–27, 29–31

**CAN-FD** CAN-Flexible Datarate. I, 4–7, 23, 24, 31

**CRC** Cyclic Redundancy Check. 5

**DLC** Data Length Code. 5

**EOF** End of Frame. 5

**IDE** Identifier Extension. 5

**IFS** Interframe Space. 5

**LKM** Linux Kernel Module. 1–3, 5, 9, 11–14, 19

**PTP** Point-To-Point. 1

**RTR** Remote Transmission Request. 5

**SOF** Start of Frame. 5

**SRR** Substitute Remote Request. 5

**UDP** User Datagram Protocol. 5

**VM** Virtual Machine. 9, 10

## IV. Literatur

- [1] URL: <https://dissec.to/> (besucht am 11.02.2024).
- [2] URL: <https://www.kernel.org/doc/html/next/networking/can.html> (besucht am 02.05.2024).
- [3] URL: <https://munich.dissec.to/kb/chapters/can/can.html#can-frame-in-memory> (besucht am 08.04.2024).
- [4] URL: <https://www.kernel.org/doc/Documentation/timers/timers-howto.txt> (besucht am 18.02.2024).
- [5] URL: <https://ebpf.io/what-is-ebpf/> (besucht am 17.02.2024).
- [6] URL: [https://man7.org/linux/man-pages/man3/clock\\_gettime.3.html](https://man7.org/linux/man-pages/man3/clock_gettime.3.html) (besucht am 17.02.2024).
- [7] *Controller Area Network (CAN) Overview*. Mai 2023. URL: <https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/controller-area-network--can--overview.html> (besucht am 06.02.2024).
- [8] Jonathan Corbet, Greg Kroah-Hartman und Alessandro Rubini. *Linux device drivers*. O'Reilly, 2005. URL: <https://lwn.net/Kernel/LDD3/> (besucht am 18.02.2024).
- [9] Steve Corrigan. *Controller Area Network Physical Layer Requirements*. Jan. 2008. URL: <https://www.ti.com/lit/an/s11a270/s11a270.pdf> (besucht am 12.02.2024).
- [10] Steve Corrigan. *Introduction to the Controller Area Network (CAN)*. Mai 2016. URL: <https://www.ti.com/lit/an/sloa101b/sloa101b.pdf> (besucht am 06.02.2024).
- [11] Martin Falch. *CAN FD Explained - A Simple Intro*. März 2022. URL: <https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro> (besucht am 06.02.2024).
- [12] *Fuzzing*. URL: <https://owasp.org/www-community/Fuzzing> (besucht am 04.04.2024).
- [13] Jim Kensiton, Prasanna S Panchamukhi und Masami Hiramatsu. *Kernel Probes (Kprobes)*. URL: <https://docs.kernel.org/trace/kprobes.html> (besucht am 23.03.2024).
- [14] James Kurose und Keith Ross. *Computernetzwerke - Der Top-Down-Ansatz*. Pearson, 2014. ISBN: 9783868942378.
- [15] *protocol*. URL: <https://github.com/dschanoeh/socketcand/blob/master/doc/protocol.md> (besucht am 02.01.2024).
- [16] *ps(1) - Linux manual page*. URL: <https://man7.org/linux/man-pages/man1/ps.1.html> (besucht am 24.03.2024).
- [17] *README*. URL: <https://github.com/linux-can/can-utils/blob/master/README.md> (besucht am 20.02.2024).
- [18] *readme*. URL: <https://github.com/mguentner/cannelloni#readme> (besucht am 26.12.2023).

- [19] *readme*. URL: <https://github.com/linux-can/socketcand/blob/master/README.md> (besucht am 02.01.2024).
- [20] Richard Sharpe, Ed Warnicke und Ulf Lamping. *Wireshark User's Guide*. Version 4.3.0. URL: [https://www.wireshark.org/docs/wsug\\_html/#ChIntroWhatIs](https://www.wireshark.org/docs/wsug_html/#ChIntroWhatIs) (besucht am 21.02.2024).
- [21] *Tmpfs*. URL: <https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html> (besucht am 06.04.2024).



# Erklärung zur Bachelorarbeit

1. Mir ist bekannt, dass dieses Exemplar der Abschlussarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Abschlussarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Regensburg, den 24. Juni 2024

*Unterrainer Matthias*

---

Matthias Unterrainer