Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Simulation of Control Systems in IEEE 802.1Qbv Networks using Omnet++

Adriaan Nieß

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer/in:	Dipl.-Ing. Ben Carabelli
Beginn am:	28. August 2018
Beendet am:	14. März 2019

Kurzfassung

Cyber Physical Systems (CPSs) verbinden physikalische und digitale Komponenten in dem Sensoren und Aktoren mit Softwarekomponenten über ein Netzwerk rückgekoppelt werden. Zur Einhaltung der Stabilitätskriterien dieser Systeme ist in vielen Fällen die Zusicherung harter Echtzeitanforderungen erforderlich. Ein Simulationsmodell, dass Time-Sensitive Networking (TSN) mit Networked Control Systems (NCSs) verbindet wäre deswegen ein wertvolles Hilfsmittel zur Evaluation dieser Systeme. Mit dem Network Simulator for Time-Sensitive Networking (NeSTiNg) Framework, das am Institut für Parallele und Verteilte Systeme (IPVS) der Uni Stuttgart im Rahmen eins Studienprojekts entwickelt wurde, stehen die Komponenten zur Simulation TSN fähiger Switches bereits zur Verfügung. Deshalb wurden in dieser Arbeit die Host Module für Regler, Sensor bzw. Aktuator implementiert. Da hierfür viele Funktionen aus der linearen Algebra sowie Numerik benötigt werden, wurde die wissenschaftliche Programmiersprache Julia verwendet. Das eventbasierte Simulationsframework Omnet++ wurde zu diesem Zweck um eine Plug-in-Schnittstelle erweitert, mit der Simulationsmodule statt in C++ nun auch in Julia implementiert werden können. Die Julia Anbindung für Omnet++ wurde hierbei so modular aufgebaut, dass diese als eigenständige Bibliothek auch unabhängig von anderen Komponenten einsetzbar ist. Am Beispiel eines LQ-Reglers für ein inverses Pendel wurde eine Proof of Concept Simulation innerhalb eines IEEE 802.1Qbv Netzwerkes evaluiert und dabei die Einflussfaktoren verschiedener Netzwerkschedulingsverfahren beobachtet. Zuletzt wurde noch ein Performance Benchmark der Julia Erweiterung für Omnet++ durchgeführt und dabei der Performance Overhead gegenüber herkömmlichen in C++ implementierten Modulen gemessen. Mit der Julia Anbindung für Omnet++ und den entwickelten Host Modulen können somit auf einfache und intuitive Weise NCS Prototypen entwickelt und analysiert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Akronyme

AVB Audio Video Bridging. 20

CBS Credit Based Shaper. 20

CPS Cyber Physical System. 3

DCB Data Center Bridging. 21

DEI Drop Eligible Indicator. 22

ETS Enhanced Transmission Selection. 23

FFI Foreign Function Interface. 30

GC Garbage Collector. 37

HLA High Level Architecture. 25

IP Internet Protocol. 43

IPG Interpacket Gap. 48

IPVS Institut für Parallele und Verteilte Systeme. 3

JLSM Julia-Simple-Module. 7

MAC Medium Access Control. 22

MTU Maximum Transmission Unit. 48

NCS Networked Control System. 3

NED Network Description. 11

NeSTiNg Network Simulator for Time-Sensitive Networking. 3

PCP Priority Code Point. 22

PTP Precision Time Protocol. 20

QoS Quality of Service. 17

RNG Random Number Generator. 54

RTT Round Trip Time. 42

SDN Software-defined Networking. 17

TAS Time-aware Shaper. 21

TDMA Time Division Multiple Access. 21

TG Task Group. 20

TPID Tag Protocol Identifier. 22

TSN Time-Sensitive Networking. 3

UDP User Datagram Protocol. 34

VID VLAN-Identifier. 22

VLAN Virtual LAN. 22

1. Einleitung

Mit zunehmender Vernetzung und Digitalisierung gewinnen so genannte Cyber Physical Sytems in Bereichen wie Smart-Manufacturing oder Smart-Grid zunehmend an Bedeutung. Diese Systeme vereinen physische und rechnergestützte Komponenten, die mithilfe eines Netzwerkes gegenseitig rückgekoppelt werden. Von Sensoren erfasste Zustandsinformationen der physischen Komponenten werden hierbei an eine Software (Regler) gesendet welcher neue Steuerungsinformationen berechnet und zurück zu den physischen Komponenten (Aktoren) sendet, um so eine Zustandsänderung des Systems zu bewirken. Dieser Art von Regelkreisen müssen vom Netzwerk eine obere Schranke für Latenzzeiten garantiert werden, damit deren Stabilität gewährleistet werden kann. Da sich innerhalb von Cyber Physical Sytems meist mehrere NCS das Kommunikationsnetzwerk mit anderen Komponenten teilen, sind für die Einhaltung der Quality of Service (QoS) Priorisierungsmechanismen erforderlich. Andernfalls könnten Datenstaus bewirken, dass Steuerungspakete kritische Deadlines überschreiten, was sich negativ auf die Regelgüte der NCS auswirkt.

Ethernet ist eine weit verbreitete Kommunikationstechnologie, die sich großer Beliebtheit erfreut und als offener Standard im Projekt 802 der IEEE entwickelt wird. Die Standardisierung ermöglicht im Vergleich zu anderen proprietären Technologien eine große Interoperabilität zwischen Komponenten unterschiedlicher Hersteller und schafft so die Voraussetzungen für den großflächigen Einsatz in industriellen Anwendungen. Im Bereich Echtzeitkommunikation sind allerdings noch proprietäre Technologien wie Profinet oder EtherCAT vorherrschend. Erst durch relativ neue Erweiterungen des Ethernet-Bridging-Standards, angefangen mit IEEE 802.1Qbv - Enhancements for Scheduled Traffic, wurde der Einsatz in Bereichen mit harten Echtzeitanforderungen ermöglicht.

Einige Forschungsprojekte befassen sich deswegen heutzutage mit TSN und Software-defined Networking (SDN). Hierbei werden auch CPS bzw. NCS innerhalb konvergenter Netzwerke in Verbindung mit relativen neuen Entwicklungen im Ethernet Standard wie z.B. IEEE802.1Qbv untersucht. So wurde z.B. in [CBDR17] ein Verfahren zum Prioritätsscheduling in Verbindung mit mehreren CPS entwickelt. In [LCD+19] wurde ein auf Zeitscheiben basierendes Übertragungsmodell mit dem gewisse QoS Kriterien (in diesem Fall Stabilitätseigenschaften eines linearen Systems) sichergestellt werden können.

Mit dem Nesting Framework [FHC+19], dass im Rahmen eines Studienprojektes am IPVS entwickelt wurde, existiert bereits eine Implementierung von TSN fähigen Ethernet Switches für das eventbasierte Simulationsframework Omnet++. Viele Features des Ieee802.1Q Standards wie Prioritätsscheduling, time-aware Shaper oder Frame-Preemption werden bereits durch das NeSTiNg Framework unterstützt. Ein oder mehrere Host-Modelle für die Simulation der regelungstechnischen Komponenten wie Regler, Sensoren und Aktoren fehlen aber noch.

Mit einem Simulationsmodell, dass ein TSN fähiges Netzwerk mit verschiedenen NCS vereint, könnten verschiedene der am IPVS entwickelten Konzepte mit nur wenig Konfigurationsaufwand in verschiedenen Testszenarien evaluiert werden. Deswegen wurden im Rahmen dieser Arbeit Host Module für Omnet++ entwickelt, die in Kombination mit dem Nesting Framework eingesetzt werden können.

Da die Systemdynamik von Regelungssystemen mathematische Berechnung wie z.B. dass Lösen von Differentialgleichungen erfordert, ist der Einsatz einer für das wissenschaftliche Rechnen prädestinierten Sprache, die über entsprechende Bibliotheken im Bereich linearer Algebra verfügt sowie mathematische Operationen wie z.B. Matrixmultiplikation schon nativ unterstützt, besonders sinnvoll. Im Gegensatz zu C++ mit dem standardmäßig Komponenten für Omnet++ entwickelt werden, ist so nur wenig Boilerplate Code erforderlich was die Entwicklung von Prototypen in einem wissenschaftlichen Anwendungsbereich stark vereinfacht. Deswegen wurde ein Plugin-System für Omnet++ entwickelt auf dessen Basis Komponenten der Simulation, als vollwertige Alternative zu C++ mit Julia entwickelt werden können. Im Vergleich zu Sprachen wie Matlab oder R, welche ihre Ursprünge in den 1970ern Jahren haben [Mol04], ist Julia eine relativ junge Sprache, die am MIT entwickelt wurde und deren erste Release Version im August 2019 veröffentlicht wurde [BEKS14]. Hierbei kann sich Julia gegenüber diesen Sprachen neben einer moderneren Syntax u.a. durch eine wesentlich höhere Performance abheben. Die im Laufe dieser Arbeit entstandenen Julia Erweiterungen für Omnet++ sind als eigenständige Bibliothek mit dem Namen JLSM entwickelt worden und können komplett unabhängig von den anderen in dieser Arbeit entstandenen Artefakte für andere Projekte wiederverwendet werden.

Auf Basis der JLSM Bibliothek wurden anschließend Host-Module zur Simulation eines zeitdiskreten NCS entwickelt. Diese wurden anschließend dazu verwendet ein inverses Pendel als Proof of Concept Szenario in einem IEEE802.1Qbv Netzwerk zu simulieren. Anhand dieser Simulation wurde die Auswirkung des im IEEE802.1Qbv Standard spezifizierten Netzwerkschedulings in Verbindung mit Crosstraffic auf die Regelgüte des NCS evaluiert. Es konnte gezeigt werden, dass es mit dem im IEEE802.1Qbv Standard spezifizierten time-aware Shaper erwartungsgemäß zu keinen Überschreitungen von Deadlines kommt und dass Prioritätsscheduling alleine nicht ausreichend ist dies zu gewährleisten.

1.0.1. Gliederung

Die Arbeit ist wie folgt gegliedert:

Kapitel 2 - Grundlagen erläutert die für das Verständnis dieser Arbeit notwendige Inhalte.

Kapitel 3 - Verwandte Arbeiten beschreibt den aktuellen Stand der Forschung und gibt einen Überblick über ähnliche Werke

Kapitel 4 - Implementierung beschreibt das Vorgehen bei der Entwicklung der Softwarekomponenten.

Kapitel 5 - Evaluation befasst sich mit der Auswertung der im vorherigen Kapitel implementierten Komponenten.

Kapitel 6 - Zusammenfassung und Ausblick fasst die aus dieser Arbeit gewonnenen Erkenntnisse zusammen und gibt einen Überblick über mögliche zukünftige Arbeiten.

2. Grundlagen

2.1. Eventbasierte Simulation

Nach systemtheoretischer Sicht unterscheidet man zwischen kontinuierlichen und diskreten Systemen. Bei kontinuierlichen Systemen hängen die Zustandsvariablen kontinuierlich von der Zeit ab. Diskrete Systeme hingegen ändern ihren Zustand nur zu bestimmten, oft im vorraus bekannten Zeitpunkten. Neben synchronen Systemen, welche ihren Zustand in zyklischen Abständen ändern, sind eventbasierte Systeme Spezielfälle von diskreten Systemen. Je nach Systemmodell können eventbasierte Simulationen den benötigten Rechenaufwand erheblich reduzieren, da nur für die Zustandsänderungen von Teilmodellen eine Berechnung durchgeführt werden muss. Insbesondere für Warteschlangenprobleme und Rechnernetze ist diese Art der Simulation besonders gut geeignet.

Nach [Bas08] lässt sich ein eventbasiertes System formalisieren durch eine endliche Menge an Entitäten $V = \{v_1, \ldots, v_n\}$ die miteinander interagieren, sowie dem Systemzustand $z = (z_1, \ldots, z_n) \in Z_1 \times \ldots \times Z_n$ wobei jeder Entität v_i der Zustand z_i zugeordnet ist. Die Signale der Signalmenge $S = s_1, \ldots, s_m$ stehen für Aktivitäten durch welche Events ausgelöst werden. Auf diesen ist außerdem eine Ordnungsrelation definiert. Die Dauer einer Aktivität ist im allgemeinen vorher bekannt, wobei auch eine Erweiterung auf stochastische Prozesse grundsätzlich möglich ist. Die Menge der Events ist definiert als $E \subseteq \mathbb{N}_0 \times S$. Die Elemente der Eventmenge sind Tupel bestehend aus einem Signal s dem Zeitpunkt t zu dem s ausgelöst wird. Schließlich gibt es noch die Zustandsübergangsfunktion $F : E \times Z \to Z$ und und die Eventfunktion $N : E \times Z \to \mathcal{P}(E)$ die die nachfolgenden Events bestimmt. $Q_0 \subseteq E$ sind die initialen Events und $z_0 \in Z$ der Anfangszustand. Algorithmus 2.1 zeigt den algorithmischen Ablauf der Simulation. In der Praxis wird meist eine Prioritätswarteschlange verwendet um die Events zu sortieren.

Algorithmus 2.1 Eventbasierten Simulation

$t \leftarrow 0$
$Q \leftarrow Q_0$
$z \leftarrow z_0$
while $Q \neq \emptyset \wedge t \leq T_{end}$ **do**
    Bestimme $e \leftarrow (t_{min}, s_{min})$ so dass $\forall (e', s') \in Q : t' > t_{min} \vee (t' = t_{min} \wedge s' \geq s_{min})$
    $Q \leftarrow Q \setminus \{e\} \cup N(e, z)$
    $t \leftarrow t_{min}$
    Speichere Daten für Statistik
end while

2.1.1. Parallelisierung

Eine Parallelisierung eventbasierter Simulation ist nur schwer möglich. Falls sich nämlich zwei Events $e_1 = (s_1, t_1)$ und $e_2 = (s_2, t_2)$ mit $t_1 < t_2$ in der Event Queue befinden und parallel berechnet werden, könnte es vorkommen, dass von e_1 ein weiteres Ereignis $e_3 = (s_3, t_3)$ mit $t_3 < t_2$ erzeugt wird. Das das aber im Vorraus nicht bekannt, ist es ungewiss welche Events sich tatsächlich parallelisieren lassen.

Ein möglicher Ansatz ist die Simulationsdomäne anhand deren Entitäten auf verschiedene Prozesse mit jeweils eigener Event Queue zu verteilen und anhand von domänenspezifischem Wissen für jeden Prozess eine Zeitschranke zu definieren bis zu der dieser Events aus seiner Queue berechnen darf. Der Simulator Omnet++ kann diese Art der Parallelisierung nutzen insofern bestimmte Einschränkungen auf dem Simulationsmodell gelten [Var03].

Andere Ansätze aus der Molekulardynamik verwenden nur eine Eventqueue und nutzen komplexes Pipelining mit Transaktion in Kombination mit Rollbacks zur Parallelisierung [CAD09].

2.2. Time sensitive Networking

Die TSN TG ist eine Standardisierungsgruppe, die Erweiterungen des Ethernet Bridging Standards IEEE 802.1Q vorantreibt und sich hierbei speziell mit Priorisierunsmechanismen auseinandersetzt, um Datenübertagungen mit möglichst niedriger Latenz und Jitter zu ermöglichen. Ziel der TSN Task Group (TG) is es Ethernet echtzeitfähig zu machen, so dass dieses auch in industriellen Anwendungen wie Motion Control, Automobil oder Automatisierung eingesetzt werden kann. Ursprünglich ging die TSN TG aus der Audio Video Bridging (AVB) TG hervor welche 2005 ins Leben gerufen wurde. Deren ursprüngliches Ziel war es die Übertragungen von Audio und Video Streams zu optimieren. Nachdem die AVB Features jedoch auch zunehmend für andere Anwendungsfälle interessant wurden wie z.B. Automobil oder Industrie, änderte die AVB TG ihren Namen in TSN TG.

2.2.1. Standards

Der Ethernet Bridging Standard IEEE 802.1Q, kann als eine Zusammenfassung mehrerer Standards betrachtet werden. Diese Standards aus denen sich der Hauptstandard IEEE 802.1Q zusammensetzt werden i.d.R. durch ein Postfix gekennzeichnet. So ist z.B. der IEEE 802.1Qbv Standard *Enhancements for Scheduled Traffic*, ein Teil des IEEE 802.1Q Standard. In regelmäßigen Abständen werden neue Sub-Standards geschaffen oder bereits bestehende Sub-Standards erweitert und wieder mit dem Hauptstandard zusammengeführt. Ursprünglich wurde von der AVB TG u.a. die folgenden Standards zum Hauptstandard beigetragen:

- **IEEE 802.1AS - Timing and Synchronization**: Beschreibt die Nutzung von Precision Time Protocol (PTP) (IEEE 1588 Standard) zur Uhrzeitsynchronisation im Kontext von TSN. Im wesentlich stellt der Standard sicher, dass alle Switches kompatible PTP Einstellungen verwenden um deren Interoperabilität zu gewährleisten.

- **IEEE 802.1Qav - Forwarding and Queuing for Time-Sensitive Streams**: Spezifiziert einen Credit Based Shaper (CBS) Algorithmus zur Übertragung von Datenströmen mit Latenzen unter $250\mu s$ und Jitter.
- **IEEE 802.1Qat - Stream Reservation Protocol (SRP)**: Ermöglicht die Verwendung von Streams bzw. Flows auf Layer 2 sowie Bandbreitenreservierung und Zulassungskontrolle.

Schließlich kamen nach der Umbenennung in TSN TG u.a. noch die folgenden Standards hinzu:

- **IEEE 802.1AS-Rev - Timing and Synchronization for Time-Sensitive Applications**: Überarbeitet und ersetzt den bisherigen IEEE 802.1AS Standard. Hier wurde vor allem die Fehlertoleranz durch Erweiterung von PTP erhöht, so dass u.a. redundante Master Clocks bzw. mehrere Synchronisationsdomänen möglich sind.
- **IEEE 802.1Qbv - Enhancements for scheduled Traffic**: Definiert einen Time-aware Shaper (TAS), der auf dem Time Division Multiple Access (TDMA) Verfahren basierend zeitgesteuerte Datenströme erlaubt. Durch die Verwendung des TAS kann eine sehr hohe Determinismusstufe erreicht werden.
- **IEEE 802.1Qbu - Frame Preemption**: Ermöglicht das Aufsplitten und die teilweise Übertragung von Ethernet Frames für eine bessere Bandbreitennutzung in Kombination mit dem TAS aus IEEE 802.1Qbv.

2.2.2. Quality of Service

Ziel vieler im IEEE 802.1Q Standard beschriebenen Verfahren wie Traffic Shaper, Zeitsynchronisation, usw. existieren vorrangig dazu die QoS für Anwender von Ethernet Netzwerken zu optimieren. Je nach Anwendungsfall können die erforderlichen QoS Kriterien variieren. Innerhalb von Computernetzwerken richten sich diese jedoch meist nach den folgenden Einflussparametern:

- **Paketverlust**: Wahrscheinlichkeit für Paketverlust
- **Latenzzeit**: Ende-zu-Ende Verzögerung
- **Jitter**: Höhe der Abweichung vom Mittel der Latenzzeit
- **Bandbreite**: Pro Zeiteinheit übertragene Datenmenge

Während die TSN TG sich hauptsächlich mit Optimierung der QoS im Bezug auf Latenzzeiten und Jitter beschäftigt, werden die Features die sich auf Paketverlustrate und die zur Verfügung stehende Bandbreite beziehen von der Data Center Bridging (DCB) TG standardisiert werden.

2.2.3. Verzögerungen bei der Übertragung von Paketen

Die zeitkritischen QoS Komponenten sind sehr stark von den verschiedenen Verzögerungen die während der Datenübertragung anfallen abhängig. Insgesammt unterscheidet man zwischen den folgenden Delay Kategorien:

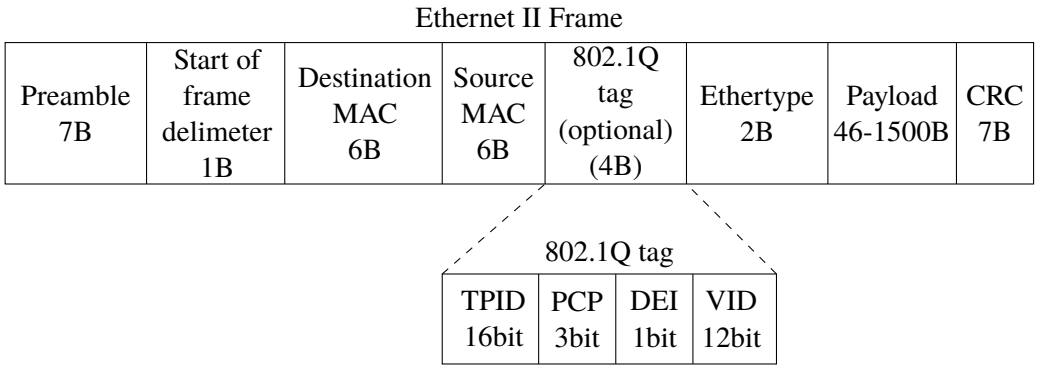


Abbildung 2.1.: Ethernet Frame Format mit IEEE 802.1Q Tag

- **Processing Delay**: Zeit die ein Host oder ein Switch benötigt um Informationen aus einem Paket zu verarbeiten. In einem Switch gehört hierzu u.a. die Zeit die benötigt wird zu entscheiden an welchen Port ein Paket weitergeleitet werden soll.
- **Transmission Delay**: Der Transmission Delay beschreibt die Zeitdauer die benötigt wird alle Bits eines Pakets zu übertragen. Der Transmission Delay ist abhängig von der Bandbreite des Übertragungskanals und der Größe eines Pakets.
- **Propagation Delay**: Zeit die ein Signal im physischen Medium Verzögert wird. Entspricht ca $2 \cdot 10^{-8}\mathrm{s\,m^{-1}}$ in einem Kupferkabel.
- **Queuing Delay**: Queuing Delay entspricht der Zeit, die ein Paket innerhalb einer oder mehreren Warteschlangen verbringt. Dieser ist abhängig von der Anzahl Pakete in den Queues sowie verwendeten Traffic Shapern.

2.2.4. Netzwerkscheduling im IEEE 802.1Q Standard

Im IEEE 802.1Q Standard werden Ethernet Pakete anhand eines Tagging Mechanismus mit einer Priorität ausgestattet. Nämlich den sogenannten Priority Code Point (PCP) Wert. Dieser entspricht einer 3bit Zahl und kann somit Werte von 0 bis 7 annehmen wobei 0 die niedrigste und 7 die höchste Priorität kodiert. Der grundsätzliche Aufbau und die Funktion des Tagging Mechanismus sowie des IEEE 802.1Q Tag wird in Abbildung 2.1 veranschaulicht. Grundsätzlich sind Tags optional. Ob ein Tag vorhanden ist oder nicht, ist daran zu erkennen, dass das *EtherType* bzw. das Tag Protocol Identifier (TPID) Feld des Frames auf den Wert 8100_{16} gesetzt wurde. Anhand des Kontexts kann ein Switch nun wissen, dass der eigentliche *EtherType* der das Layer 3 Protokoll identifiziert um einen Offset von 4B verschoben ist. Neben dem PCP Wert enthält der IEEE 802.1Q Tag noch weitere Felder wie den Drop Eligible Indicator (DEI) und den VLAN-Identifier (VID) Wert. Das DEI Feld gibt an ob ein Frame bei Datenstau verworfen werden soll und das VID Feld ordnet ein Frame einem Virtual LAN (VLAN) zu. Die Zuordnung von IEEE 802.1Q Tags kann von Endsystemen als auch von Switches erfolgen. Hier ist auch das Verschachteln von Tags möglich.

Nachdem ein Frame von einem Switch empfangen wurde, wird es anhand der Ziel Medium Access Control (MAC) Adresse intern an einen oder mehrere Queuing Netzwerke die jeweils einem Ausgangsport zugeordnet sind, weitergeleitet. Durch dieses Queuing Netzwerke werden schließlich die

für TSN erforderlichen Priorisierungsmechanismen umgesetzt. Abbildung 2.2 zeigt den beispielhaften Aufbau eines solchen Queuing Netzwerkes. Die genaue Konfiguration der Traffic Shaper und die Queue Anzahl kann je nach Modell des Ethernet Switches oder dessen Einstellungen variieren.

Wenn ein Frame einem Ausgangsport zugeordnet wird und in das Queuing Netzwerk einsortiert wird, erfolgt die Zuordnung zu einer Queue mithilfe einer Mapping Tabelle anhand der Priorität bzw. des PCP Wertes. Standardmäßig wird hierbei der PCP Wert direkt auf Queue ID abgebildet. Mit Ausnahme von Priorität 0 bzw. 1 welche auf Queue 1 bzw. 0 abgebildet wird.

Zu jeder Queue wird jeweils ein Transmission Selection Algorithmus bzw. ein Traffic Shaper zugeordnet. Standardmäßig wird hier der strict-priority Algorithmus verwendet, der im Endeffekt nur die zugehörige Queue als übertragungsbereit kennzeichnet falls diese nicht leer ist. Da der strict-priority Algorithmus das von einer Queue erwartete Verhalten weder einschränkt noch erweitert und sich somit nur neutral verhält, wurden die Queues 7, 1 und 0 welche diesen verwenden im Schaubild 2.2 nicht speziell gekennzeichnet. Neben dem strict-priority Algorithmus gibt es noch den CBS und den Enhanced Transmission Selection (ETS) Algorithmus. Der CBS funktioniert ähnlich wie der Leaky-Bucket Algorithmus. Mit ihm lässt sich eine maximale Senderate festlegen und der Algorithmus versucht gesendete Frames auf möglich regelmäßigen Sendezeitpunkte zu verteilen um den Jitter zu minimieren. Intern verwendet der Algorithmus Credits um den nächsten Sendezeitpunkt eines Frames zu bestimmen. Ist der Credit-Wert positiv kann der CBS Frames aus der zugehörigen Queue passieren lassen. Beim Senden eines Frames werden Credits verbraucht und bei passivem Warten angehäuft. Um eine Ansammlung von Credits zu vermeiden wie das beim Tocken Bucket Algorithmus der Fall sein kann, werden die Credits auf 0 zurückgesetzt falls keine übertragungsbereiten Frames vorhanden sind. Der sonst gebräuchliche Begriff Token wird hier deswegen nicht verwendet weil der Credit-Wert auch negativ werden kann. Zuletzt ist mit dem ETS Algorithmus Queue-übergreifendes Bandbreitenmanagement möglich. Hierzu werden mehrere Queues zu einer Gruppe zusammengefasst. Die insgesammt einer Gruppe zur Verfügung stehende Bandbreite wird dann je nach Konfiguration prozentual auf die einzelnen Queues verteilt. Implementiert wird ETS meist basierend auf einem Round Robin Verfahren.

Nach den Transmission Selection Algorithmen folgen die vom IEEE 802.1Qbv Standard spezifizierten Transmission Gates. Mit diesen ist es möglich einzelnen Queues die Übertragung von Frames zu erlauben oder diese gänzlich zu blockieren. Die Öffnungs und Schließungszeiten der Gates werden durch ein konfigurierbares Schedule festgelegt. Schedules bestehen aus einem Startzeitpunkt t_{start}, einer Zyklusintervalllänge $\Delta t_{\text{schedule}}$, sowie einer Liste die Tuples bestehend aus Bitvektoren und Zeitintervallen enthält. Wird ein Schedule aktiviert, so wird dieses zum nächsten möglichen Zeitpunkt $t_{\text{start}} + n \cdot \Delta t_{\text{schedule}}$ mit minimalen n gestartet. Auf diese Weise kann bei Verwendung mehrere Schedules die Synchronität der Schedules gewährleistet werden. Die Bitvektoren spezifizieren die Gate Zustände und die Zeitintervalle die Gültigkeitsdauer eines Bitvektors. Die Schedules werden nach Ablauf der Zykluszeit i.d.R. wiederholt. Sämtliche Zeitangaben bezogen auf die Zeitintervalle sind nach IEEE 802.1Q Standard in Nanosekunden definiert. Konkreten Switch Implementierungen steht es aber frei eine gröbere Granularität der Ticks zu wählen. Im Gegensatz zu ereignisbasiertem Scheduling bietet der IEEE 802.1Qbv Standard mit den Transmission Gates somit eine Möglichkeit für zeitgesteuerten Scheduling. Der Gate Mechanismus wird deswegen auch als time-aware Shaper bezeichnet (TAS). Damit das Zusammenspiel mehrere Schedules verteilt auf mehrere Switches verteilt aber funktionieren kann ist eine präzise Zeitsynchronisation mit PTP erforderlich.

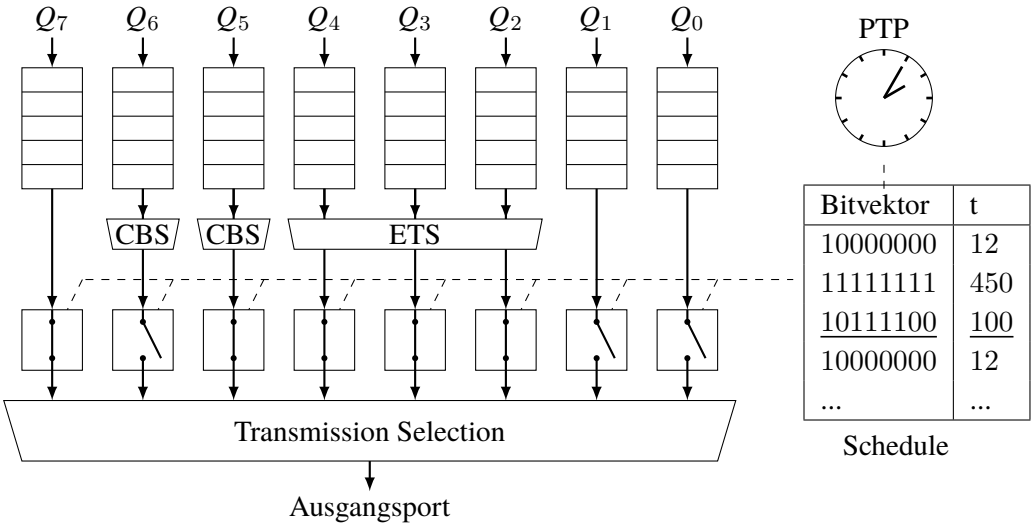


Abbildung 2.2.: Queuing Netzwerk nach IEEE 802.1Q Standard

Standard	**Transmission Selection Alg.**	**Traffic Klasse**	**Determinismus Grad**
IEEE 802.1Q	-	Klasse Best effort	Niedrig
	Strict Priority	Klasse Priorisiert	Mittel
IEEE 802.1Qav (AVB)	Credit Based Shaper (CBS)	Klasse B	Mittel
		Klasse A	Mittel
IEEE 802.1Qbv (TSN)	Time Aware Shaper (TAS)	Klasse CDT	Hoch

Tabelle 2.1.: TSN Spezifikationen der Datenströme [NL18]

Zuletzt werden die verschiedenen Datenflüsse wieder durch die Transmission Selection Komponente zusammengeführt. Diese wählt das als nächstes zu übertragende Frame aus der höchspriorisierten Queue aus und führt somit auf der letzten Hierarchiestufe des Queuing Netzwerks klassisches Prioritätsscheduling aus. Insgesammt wird also eine Queue zur Übertragung des nächsten Frames ausgewählt falls (1) die Queue nicht leer ist, (2) keine höher priorisierte Queue mit übertragungsbereitem Frame existiert, (3) der zur Queue gehörende Transmission Selection Algorithmus die Übertragung erlaubt und (4) das zugehörige Transmission Gate geöffnet ist.

Insgesammt kann man für TSN 5 Arten von Traffic Klassen unterscheiden wie Tabelle 2.1 zeigt. Der CBS wurde hierbei speziell für AVB Anwendungsfälle konzipiert. Es ist vorgesehen diesen an den Queues mit Priorität 5 und 6, also den Queues mit der zweit- bzw. dritthöchsten Priorisierungsstufe zu verwenden. Der zugehörige Traffic wird so kategorisiert in die Klasse A und B, wobei hier bei Verwendung von maximal 7 Hops 125µs bzw. 250µs Verzögerung für die Ende-zu-Ende Verzögerung garantiert werden können. Für zeitkritischen Steuerungstraffic müssen allerdings meist Latenzen von weniger als 100µs gewährleistet werden. Das kann nur durch die Verwendung des TAS gewährleistet werden [Tak11].

2.2.5. Guard Bands

Ohne die Verwendung von Frame Preemption oder length-aware Scheduling aus dem IEEE 802.1Qbu Standard wird die Übertragung eines Frames auch dann fortgesetzt wenn das zugehörige Transmisson Gate inmitten der Übertragung geschlossen wird. Bei der Erstellung von Schedules muss deswegen berücksichtigt werden, dass die Übertragung anderer Frames sich mit dem Öffnungsintervall einer geschützten Traffic Klasse überschneiden kann. Um negative Effekte zu vermeiden und Traffic Klassen durch TDMA komplett zu isolieren, wird der Einfluss von Cross Traffic dadurch vermieden, dass Gates schon um den Zeitraum, der zur Übertragung eines MTU großen Frames benötigt wird, früher geöffnet werden. Dieser einzuplanende Puffer wird auch als Guard Band bezeichnet. Das Guard Band kann somit die Interferenz mit anderen Traffic Klassen erfolgreich verhindern. Bei 100Mbit s^{-1} muss die für das Guard Band also $122\mu\text{s}$ eingeplant werden, bei 1Gbit s^{-1} $12.2\mu\text{s}$. Da die Konfiguration der Schedules durch das Guard Band verkompliziert wird, bietet der IEEE 802.1Qbv Standard mit length-aware Scheduling die Möglichkeit die Guard Bands implizit vom Switch managen zu lassen, so dass diese bei der Berechnung des Schedules nicht beachtet werden müssen. Ein wesentlicher Nachteil bei der Verwendung von Guard Bands, ist die Bandbreitenverschwendung. Um eine bessere Linkauslastung zu ermöglichen können bei Verwendung von Frame Preemption Ethernet Frames auch nur teilweise übertragen werden. Wird ein Gate während einer Übertragung geschlossen hat das dann eine Zerteilung des übertragenden Frames zur Folge. Auf diese Weise kann die Größe des Guard Bands auf die Zeit die zur Übertragung der kleinstmöglichen Framegröße benötigt wird, reduziert werden. Die Größe des Guard Bands kann also um einen Faktor von fast 24 reduziert werden.

2.3. Omnet++

Omnet++ ist ein eventbasiertes Simulationsframework das seit 1997 hauptsächlich zur Simulation und Analyse von Rechnernetzen und verteilten Systemen entwickelt wird [VH08]. Der Name Omnet++ steht hierbei für Objective Modular Network Testbed in C++. Entwickelt wird der Omnet++ Simulator von der Firma OpenSim Ltd. die diesen zur akademischen Nutzung frei zur Verfügung stellt. Es existiert noch eine komerzielle Version die unter dem Namen OMNEST vertrieben wird. Diese stellt u.a. im Gegensatz zu Omnet++ noch ein High Level Architecture (HLA) Interface zur Simulatorkopplung zur Verfügung. Neben umfassenden statistischen Analysewerkzeugen stellt Omnet++ zudem mehrere Simulationsumgebungen bereit. So können Simulationen sowohl grafisch visualisiert werden, also auch Recheneffizient mit einem Kommandozeilenwerkzeug ausgeführt werden. Module für den Omnet++ Simulator müssen entweder selbst in C++ programmiert werden oder werden durch Bibliotheken bereitgestellt. Simulation selbst werden in Omnet++ hierarchisch aufgebaut. Die wichtigsten Komponenten sind hierbei:

- **SimpleModules**: *SimpleModules* sind die wichtigsten atomaren Bausteine zum Aufbau größerer Simulationskomponenten. Ein *SimpleModule* besteht aus einer C++ Implementierung, also einem C++ Header sowie einer Source Datei. Zusätzlich gehört zu einem *SimpleModule* eine NED Datei. NED Dateien erlauben die Festlegung von Parameter die durch die Simulationskonfiguration festgelegt werden können. Z.B. die Senderate einer Anwendung. Es werden dort aber auch definiert welche Statistiken durch ein Modul erfasst werden, sowie die Ein- und Ausgänge über das ein Modul durch Message Passing mit anderen Omnet++ Komponenten

kommunizieren kann. NED Dateien werden hierbei in der eigenen Omnet++ spezifischen NED Sprache programmiert. Aus dem C++ Code kann schließlich mit der Omnet++ API auf Inhalte wie NED Parameter zugegriffen werden.

- **Channels**: Channels bzw. Kanäle werden dazu verwendet Module untereinander zu verbinden, so dass diese über Message Passing kommunizieren können. Damit Module mithilfe von Channels verbunden werden können, müssen diese über die entsprechenden Ein- und Ausgänge verfügen. Mithilfe von Channels ist es auch möglich Verzögerungen in der Kommunikation von Modulen zu modelieren.
- **CompoundModules**: Durch *CompoundModules* können aus anderen *CompoundModules* oder *SimpleModules* größere Module zusammengebaut werden. Dabei werden diese Submodules meist durch Channels miteinander verbunden. *CompoundModules* verfügen im Gegensatz zu *SimpleModules* über keine C++ Implementierung. Sie werden lediglich in einer NED Datei definiert. Genau wie *SimpleModules* können sie über Ein- und Ausgänge verfügen. Die zu Omnet++ gehörende Entwicklungsumgebung ermöglicht es auch *CompoundModules* rein grafisch zusammenzustellen.

Für die Entwicklung von eigenen *SimpleModules* muss die *cSimpleModule* Klasse aus Omnet++ spezialisiert werden. Dabei müssen hauptsächlich Eventhandler implementiert werden. Die wichtigsten sind:

- **numInitStages**: Die *numInitStages* Methode ist der allererste Eventhandler der aufgerufen wird. Dieser kann überschrieben werden um die Anzahl zu initialisierender Schichten als Rückgabewert festzulegen.
- **initialize**: Die *initialize* Methode wird vor dem Simulationsstart aufgerufen. Im Gegensatz zum Konstruktor kann hier auf die Simulationsparameter aus der zugehörigen NED Datei zugegriffen werden. Je nach Anzahl zu initialisierender Schichten wird dieser Eventhandler mehrmals aufgerufen. Das mehrstufige Initialisierungskonzept kann z.B. dazu genutzt werden Komponenten aus der Vermittlungsschicht erst nach den Komponenten der Sicherungsschicht zu initialisieren.
- **handleMessage**: Bei der *handleMessage* Methode handelt es sich vermutlich um den wichtigsten Eventhandler. Dieser wird u.a. jedes mal aufgerufen falls ein *SimpleModule* eine Nachricht über einen Kanal erhält.
- **finish**: Wird nach Beendigung der Simulation ausgeführt und meist dazu genutzt Statistiken zu erfassen.

2.3.1. INET

Da Omnet++ selbst überhaupt keine Modulimplementierungen zur Verfügung stellt, ist der Simulator alleine recht nutzlos. Deswegen wird Omnet++ meist immer in Kombination mit Bibliotheken bzw. Frameworks verwendet. Das vermutlich am häufigsten verwendete Framework ist INET. Der genaue Funktionsumfang von INET wird in [INE18] beschrieben. Im wesentlichen implementiert INET alle häufig verwendeten Kommunikationsprotokolle. Der Funktionsumfang von INET ist hierzu in folgende Kategorien unterteilt für die jeweils ein paar Beispiele angeführt sind:

- **Application Layer**: Protokolle wie HTTP, DHCP, BitTorrent, Video und Sprach- Übertragungen
- **Tranpsort Layer**: TCP, UDP
- **Network Layer**: IPv4, IPv6, ARP, ICMP
- **Routing**: Link-state Routing, BGP, RIP
- **Mobile Ad-hoc-Routing**: AODV, DYMO, GPSR, DSDV
- **MPLS**: MPLS, LDP, RSVP-TE
- **Wired**: PPP, Ethernet, STP, RSTP
- **Wireless**: 802.11, 802.11p, 802.1e, LTE
- **Mobility**: Satellite Mobility, Vehicular Mobility

2.3.2. Nesting

NeSTiNg ist eine weitere Bibliothek für das Omnet++ Simulationsframework, die im Rahmen eines Studienprojekts am IPVS der Universität Stuttgart entwickelt wurde. Aufbauend auf INET werden von NeSTiNg verschiedene TSN Standards wie z.B. IEEE 802.1Qbv und IEEE 802.1Qbu implementiert [FHC+19]. Das Projekt wird durch Arbeiten wie [Wei18] und durch studentische Mitarbeiter weiterhin aktiv entwickelt.

2.4. Networked Control Systems

Bei Networked Control Systems handelt es sich um eine Klasse von Regelungssystemen bei denen die Rückkopplung über ein Kommunikationsnetzwerk wie z.B. Ethernet geschlossen wird. Die Besonderheit dieser Systeme ist, dass Ein- und Ausgangswerte mithilfe von Netzwerkpaketen nur in zeitdiskreter Form übertragen werden können.

Im folgenden Kapitel wird zunächst die Funktionsweise eines LQ-Reglers beschrieben. Anschließend das kontinuierliche Systemmodell eines inversen Pendels. Zuletzt wird kurz erläutert wie ein kontinuierliches System diskretisiert werden kann, so dass dieses als NCS verwendet werden kann.

2.4.1. Linear-quadratischer Regler

Ein LQ Regler ist ein Zustandsregler für lineare dynamische Systeme. Solche Systeme lassen sich in folgender Form beschreiben:

$$\dot{x} = Ax + Bu \tag{2.1}$$

$$u = -Kx \tag{2.2}$$

Hierbei ist x der Zustandsvektor des Systems und u der Eingangsvektor. Die Vektoren x und u sind demnach zeitabhängig. Die Matrix A beschreibt den Einfluss des aktuellen Zustands und B den Einfluss der Eingänge auf das System. Man nennt A deswegen auch Systemmatrix und B Eingangsmatrix. Mithilfe der Rückführmatrix K lassen sich anhand des aktuellen Zustands x die Eingangssignale u ermitteln. Ein Regler wählt die Rückführmatrix K so, dass der Systemzustand gegen Null konvergiert. Um die optimale Rückführmatrix zu ermitteln, wird die die Kostenfunktion J definiert:

$$J = \int_0^\infty (x^T Q x + u^T R u) dt \tag{2.3}$$

Durch Minimierung von J werden die Matrizen Q und R bestimmt. Diese werden zur Gewichtung verschiedener Einflussfaktoren verwendet. Mit den Diagonalelementen der Q Matrix kann beschrieben werden, wie schnell einzelne Zustände gegen Null konvergieren sollen. Die anderen Matrixelemente von Q sind allerdings schwer zu interpretieren weswegen sie oft auf Null gesetzt werden. Es ist zu beachten, dass Q positiv definit sein muss, damit erste Summand größer Null ist um bei der Minimierung des Integrals keine großen Zustände zu belohnen. Mit der R Matrix werden analog zur Q Matrix die Einflüsse der Eingangssignale beschrieben. Die R Matrix muss hierbei positiv semidefinit sein, damit der zweite Summand größer oder gleich Null ist. Durch Einsetzen von 2.4 und 2.5 lässt sich J berechnen.

$$x = x(t) = x(0) \cdot e^{At} \tag{2.4}$$
$$u = u(t) = -Kx \tag{2.5}$$

Das Integral J ist somit abhängig von K und dort wo J minimal ist, ist K optimal. Berechnet man das Integral und bestimmt das Minimum, führt das schließlich zur algebraischen Riccati Gleichung 2.6. Ein ausführlicher Rechenweg ist in [PLB15] beschrieben.

$$PA + A^T P - PBR^{-1}B^T P + Q = 0 \tag{2.6}$$

Das resultierende Gleichungssystem 2.6 ist nicht linear, weswegen es potentiell mehrere Lösungen besitzen kann. Es kann jedoch gezeigt werden, dass nur eine Lösung P existiert [Rod94]. Leider ist die Gleichung nur numerisch lösbar. Deswegen wird diese meist mit Hilfsmitteln wie Matlab gelöst. Durch bestimmen von P kann man die Rückführmatrix K ermitteln:

$$K = R^{-1}B^T P \tag{2.7}$$

Ein LQR lässt sich also vollständig anhand des 4-Tuples (A, B, Q, R), also durch Systemmatrix, Eingangsmatrix, Systemgewichtungsmatrix und Eingangsgewichtungsmatrix definieren und relativ einfach mit einer wissenschaftlichen Programmiersprache automatisiert lösen.

Der Vorteil der LQR Methode liegt im wesentlichen darin, dass die Rückführmatrix K algorithmisch berechnet werden kann und gleichzeitig Systemeigenschaften relativ einfach und überschaubar durch die Q und R Matrizen konfiguriert werden können.

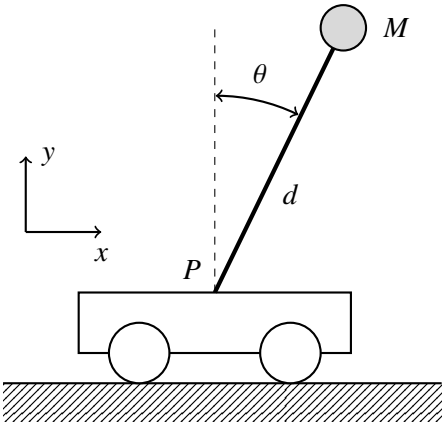


Abbildung 2.3.: Inverses Pendel

2.4.2. Inverses Pendel

Das inverse Pendel ist ein geläufiges Standardbeispiel der Regelungstechnik. Hierbei soll ein frei schwingendes Pendel mit dem Schwerpunkt überhalb der Drehachse balanciert werden. Es gibt verschiedene Varianten dieses Problem mit unterschiedlichen Freiheitsgraden. Am bekanntesten ist jedoch die Version bei der das Pendel auf einem Wagen montiert ist, der sich entlang einer festen Achse bewegen kann. In diesem Kapitel wird der physikalische Hintergrund des Pendels beschrieben und die für einen LQ-Regler erforderliche System- und Eingangsmatrix bestimmt.

In Abbildung 2.3 ist der konzeptuelle Aufbau des inversen Pendels zu sehen. Der Wagen bzw. Schlitten auf dem das Pendel am Aufhängungspunkt P montiert ist kann sich entlang der x-Achse frei bewegen. Am Ende des Pendels mit Länge d befindet sich eine punktförmige Masse M. Der Winkel θ gibt die Neigung des Pendels an. Somit lässt sich der Zustandsvektor des Pendels beschreiben als:

$$(x, \dot{x}, \theta, \dot{\theta})^T \tag{2.8}$$

Mithilfe des Lagrange Formalismus kann man die Gleichung für die Winkelbeschleunigung berechnen [Näg15] und erhält:

$$\ddot{\theta} = \underbrace{\frac{d}{d^2 + d^2}}_{=\mathrm{F}} (\ddot{x}\cos(\theta) + g \cdot \sin(\theta)) \tag{2.9}$$

Die Gleichung ist nicht linear und muss für die Verwendung durch einen LQR Regler erst linearisiert werden. Da ein Regler das Pendel in aufrechtem Zustand zu halten versucht, kann die Gleichung um $\theta \approx 0$ linearisiert werden:

$$\ddot{\theta} = F\ddot{x} + Fg\theta \tag{2.10}$$

Somit lässt sich das Zustandsraummodell aufstellen:

$$\begin{pmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & Fg & 0 \end{pmatrix} \begin{pmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ F \end{pmatrix} u \tag{2.11}$$

2.5. Julia

Julia ist eine high-level Programmiersprache für das wissenschaftliche Rechnen. Die Entwicklung der Sprache begann 2009 am MIT und 2012 wurde Julia als Open Source veröffentlicht. Mittlerweile haben über 500 Personen am Julia Projekt mitgewirkt und die Community hat über 1200 Pakete beigetragen [BEKS14]. 2019 wurden schließlich die Entwickler Jeff Bezanson, Steven L. Brunton, Jack Dongarra, Stefan Karpinski, and Viral B. Shah für ihre Arbeit am Julia Projekt mit dem J. H. Wilkinson Prize for Numerical Software ausgezeichnet [New19].

Julia zeichnet sich dadurch aus, dass von Anfang bei der Entwicklung der Fokus auf Performance gerichtete wurde. Zudem besitzt Julia ein dynamisches Typsystem, erlaubt aber die optionale Verwendung parametrisierter Type. Die Verwendung von Multiple Dispatch bzw. Multimethoden macht Julia zudem einfach in der Anwendung beim Lösen numerischer Probleme oder bei linearer Algebra. Anders als Matlab ist Julia außerdem Open Source und frei verfügbar unter der MIT Lizenz. Das motiviert natürlich dazu sich näher mit Julia zu beschäftigen.

2.5.1. Interoperabilität

Für viele Anwendungsfälle ist die Interoperabilität mit Bibliotheksfunktionen eine wichtige Grundlage. Julia stellt hierzu eine Foreign Function Interface (FFI) zum Aufruf von nativen C oder Fortran Methoden zur Verfügung. Jedoch ist die direkte Verwendung relativ mühsam, weswegen es verschiedene Julia-Pakete gibt die diesen Prozess vereinfachen und zum Teil automatisieren können.

Das Cxx.jl Paket ermöglicht den Aufruf von C++ Code aus Julia Anwendungen heraus. Auf diese Weise reicht es aus eine C++ Header Datei zu laden und optional können die C++ Aufrufe noch in native Julia Methoden geschachtelt werden. Somit kann Wrapper Code komplett in Julia geschrieben werden ohne, dass es nötig ist C++ oder C-Code zu schreiben [Fis18].

Einen anderen Ansatz wird von CxxWrap.jl verfolgt. Hier werden mithilfe einer C++ Komponente des CxxWrap.jl Pakets im C++ Codes Mappings definiert, die dann zur automatisierten Erzeugung von Julia Bindings verwendet werden. Innerhalb einer Julia Anwendung muss nur noch der Pfad zur Bibliothek definieren [Jan19].

3. Verwandte Arbeiten

3.1. Al-Hamuri: Co-Simulation von CPS mit ns-2 und Modelica

Ahmad T. Al-Hamuri beschreibt in seiner Arbeit *A comprehensive co-simulation platform for cyber-physical systems* [AlH12] ein Verfahren, dass von ihm genutzt wurde CPS in der Modellierungssprache Modelica zu entwickeln und diese Über ein Co-Simulationsinterface mit dem eventbasierten Netzwerksimulator ns-2 zu Koppeln. Auf diese Weise gelang es ihm verschiedene NCS zu simulieren. Anschließend beschreibt Al-Hamuri wie er das ursprünglich von ihm verwendete Verfahren dahingehend verbessert, so dass auch von Modelica ausgehend asynchron Nachrichten an den ns-2 Simulator gesendet werden können.

Das ursprüngliche Konzept von Al-Hamuri sieht vor, dass der ns-2 Simulator stets der Simulationszeit von Modelica voraus eilt. Nach jedem von ns-2 verarbeitetem Event wird die Simulation in Modelica weiter ausgeführt bis an die Stelle der aktuellen Simulationszeit von ns-2. Auf diese Weise kontrolliert der ns-2 Simulator das Fortschreiten der Modelica Simulation. Von der Modelica ausgehend ist eine asynchrone Kommunikation zum ns-2 Simulator nicht möglich, sondern es können nur jeweils an den Zeitpunkten Daten nach ns-2 kommuniziert werden an denen der ns-2 Simulator ein Event berechnet. Zur Interprozesskommunikation werden Named Pipes verwendet.

Diese Einschränkung wird durch Al-Hamuri behoben indem zunächst von beiden Simulatoren der Zeitpunkt des jeweils nächsten Events abgefragt wird. Schließlich wird das frühere der beiden Events ausgewählt. Ist das Event dem ns-2 Simulator zugeordnet, so wird die Berechnung der Modelica Simulation bis zu diesem Zeitpunkt fortgesetzt und dann die ns-2 Simulation. Zu den ns-2 Eventzeitpunkten lässt sich eine asynchrone von Modelica zu ns-2 hier relativ einfach realisieren. Das Vorgehen ist also jetzt dahingehend verändert worden, das nun die Modelica Simulation vor der ns-2 Simulation berechnet wird. Wird allerdings das nächste Event von der Modelica Simulation berechnet, so wird zunächst die Modelica bis zu dem Zeitpunkt des Events ausgeführt. Anschließend wird die ns-2 Simulation ausgeführt welche ein späteres Event berechnet. Durch einen Rollback wird die ns-2 Simulation nun auf den ursprünglichen Zeitpunkt vor dem Modelica-Event zurückgesetzt und fügt ein neues Event für den Zeitpunkt des Modelicas Events hinzu. Schließlich wird die ns-2 Simulation wieder ausgeführt deren nächstes Event nun zur gleichen Simulationszeit wie das der Modelica Simulation berechnet wird. Falls von der Modelica Simulation eine asynchrone Nachricht an die ns-2 Simulation gesendet wurde, kann diese nun entgegengenommen werden ohne temporale Invarianten zu verletzen.

Vorteilhaft bei diesem Verfahren ist das mit Modelica relativ intuitiv mithilfe grafischer Symbole komplexe dynamische Systeme modelliert werden können. Allerdings müssen für die technische Umsetzung jeweils alle Zustände (oder Zustandsänderungen) der ns-2 Simulation zwischengespeichert werden, so dass das Zurücksetzen des Zustands und eine Neuausführung der Simulation möglich ist. Das ganze ist demzufolge mit einem Performance Overhead verbunden. Leider existiert in [AlH12] hierzu kein Benchmark Szenario.

3.2. Yu Zhang: Co-Simulation von CPS mit Matlab und Qemu

Yu Zhang beschreibt in dem Paper *A Co-Simulation Interface for Cyber-Physical Systems* [ZDFH16] wie ein CPS durch Kopplung von Matlab und Quemu simuliert werden kann. In Matlab wird hierzu das Modell der Regelstrecke simuliert, während die Controller Hard- und Software in Qemu emuliert wird. Die Kopplung von Matlab und Quemu erfolgt über ein spezielles Matlab Interface und eine virtuelle Gerätedatei.

Mit dem hier vorgestellten Verfahren kann die originale Software des Reglers unter simulierten Bedingungen getestet werden bevor diese in der realen Welt eingesetzt wird. Der Fokus liegt hier aber eher im Bereich eingebetteter Systeme und nicht direkt auf der Simulation von NCS und den Einflussfaktoren durch das Übertragungsmedium bzw. Netzwerk.

3.3. Roth und Burns: HLA Schnittstelle für CPS Simulatoren

In [RB18b] wird das Design und die Anforderungen eines minimalen Interfaces beschrieben, dass benötigt wird Simulationsplattformen für CPS konform zum HLA Standard über eine Middlerware zu koppeln. Der HLA Standard wurde ursprünglich durch das US Verteidigungsministerium zur integrierten und verteilten Simulation entwickelt und wurde später als IEEE 1516 zum internationalen Standard.

Durch die Verwendung von HLA kann eine hohe Interoperabilität der verwendeten Modelle und Simulatoren erreicht werden, so dass diese möglichst einfach wiederverwendet werden können, allerdings ist der Aufwand für die Modell und Simulatorerstellung relativ hoch. Für schnelles Prototyping scheint HLA eher wenig geeigenet zu sein sondern eher für die Simulation sehr großer interdisziplinärer Projekte.

3.4. CoCPN - Co-Simulation von CPS mit Matlab und Omnet++

In [RJZH19] und [JRZ18] wird die Funktionsweise eines Simulationstools beschrieben, dass im Rahmen des CoCPN (Cooperative Cyber Physical Networking) Projekts am KIT entwickelt wurde. Hierbei werden CPS bzw. NCS durch Kombination von Omnet++ und Matlab simuliert. Während von Omnet++ die Netzwerkinfrastruktur simuliert wird, wird Matlab dazu verwendet die Feedback-Loops der dynamischen Systeme zu berechnen.

Realisiert ist die Kommunikation zwischen den Simulationswerkzeugen durch eine in Omnet++ implementierte CoCPN Komponente welche in periodischen Abständen Hooks in Matlab ausgelöst um die Feedback-Loops der NCS zu berechnen. Zwischen den diskreten Update Zeitpunkten werden Queues verwendet um die zwischen Matlab und Omnet++ kommunizierten Nachrichten zu puffern.

CoCPN ermöglicht letztendlich eine intuitive Reglerimplementierung von NCS in Matlab. Allerdings ist man innerhalb von Matlab an ein relativ restriktives Interface zur Kommunikation mit Omnet++ angewiesen. Die Erweiterbarkeit ist insgesamt sehr eingeschränkt.

4. Implementierung

Im Rahmen dieser Arbeit sollte ein Host-Model für Omnet++ entwickelt werden, dass zur Simulation von Networked Control Systems in Ethernet Netzwerken verwendet werden kann. Die Komponenten für das Netzwerk stehen bereits durch Verwendung des INET Frameworks und des Nesting Frameworks zur Verfügung. Diese enthalten eine große Zahl an Komponenten wie z.B. IEEE 802.1Qbv konforme Switches [FHC+19] oder Implementierungen von Netzwerstacks. Im Mittelpunkt der Implementierung stehen also die Host-Module, die die Bestandteilen eines Regelungssystems realisieren.

Da die Host-Module für Regler und Regelstrecke in Julia implementiert werden sollten, war es erforderlich eine Infrastruktur zu schaffen die es erlaubt in Julia geschriebenen Code im Omnet++ Framework zu integrieren. Dazu wurde die JLSM Bibliothek entwickelt, mithilfe derer Omnet++ Module als Alternative zu C++ auch in Julia implementiert werden können. Die Entwicklung der JLSM Bibliothek wird im ersten Teil des Implementierungskapitels beschrieben und anschließend in einem zweiten Teil der Aufbau der regelungstechnischen Komponenten auf Basis.

4.1. Julia Erweiterungen für Omnet++

Um Bestandteile eines NCS, wie z.B. Regler oder Regelstrecken in Julia implementieren zu können, ist ein Interface zu Omnet++ erforderlich, dass es ermöglicht Julia Funktionen im Eventhandling von Omnet++ zu integrieren. Gleichzeitig muss auch ein Datenfluss von Julia nach Omnet++ existieren, damit es Julia Funktionen nicht nur möglich ist passiv auf Events zu reagieren, sondern damit diese auch aktiv eine laufende Simulation beeinflussen können indem sie beispielsweise selbst Events erzeugen.

Bei der JLSM Bibliothek die das Interface zwischen Omnet++ und Julia bereitstellt, handelt es sich um eine besonders kritische Komponente, ohne die sich die Komponenten von NCS nicht implementieren ließen. Zusätzlich ist die Architektur des Interfaces ausschlaggebend für die Flexibilität mit welcher später weitere Komponenten in Julia entwickelt werden können. Deswegen wurde dieses mit großer Sorgfalt entworfen und nahm einen Großteil der Implementierungsarbeit in Anspruch. Auch in anderen Projekten wie [ZDFH16] wurde das Interface Design als aufwendigster Teil der Implementierung befunden.

Im Entwicklungsprozess wurden zunächst die Anforderungen an das Julia Interface analysiert. Anschließend wurden verschiedene Lösungsmöglichkeiten unterschiedlicher Teilaspekte betrachtet. Anhand der Abwägung von deren Vor- und Nachteilen wurde dann eine sinnvoll erscheinende Softwarearchitektur abgeleitet und implementiert.

4.1.1. Analyse der Anforderungen

Unter Berücksichtigung der Forschungsschwerpunkte am IPVS sowie im Gespräch mit wissenschaftlichen Mitarbeitern wurden eine Reihe von Anforderungen ermittelt. Diese wurden in muss- oder soll-Anforderungen unterteilt:

A1. Die Julia Anbindung *muss* über die nötigen Callbacks verfügen die die Implementierung eines NCS zur Simulation eines inversen Pendels ermöglichen. Dazu gehört im wesentlichen:

- Das Senden und Empfangen von Zustands- bzw. Steuerinformationen.
- Ein Update-Mechanismus, mit dem der Zustand des Systems zyklisch, in Intervallen variabler Länge, angepasst werden kann.
- Das Aufzeichnen statistischer Daten wie Regelgüte, Paketumlaufzeit oder Zustand der Regelstrecke.

A2. Das Interface *soll* möglichst flexibel sein um später auch die Implementierung anderer NCS zu ermöglichen. Es wäre wünschenswert, wenn zusätzlich zu zeitdiskreten linearen NCS wie einem inversen Pendel, mit wenig Aufwand z.B. auch eventbasierte NCS implementierbar wären.

A3. Es *soll* eine möglichst hohe Kompatibilität mit dem INET und dem Nesting Framework erzielt werden, um möglichst viele der von diesen Bibliotheken breitgestellten Komponenten und Features nutzen zu können. Dafür müssen die Julia Erweiterungen u.a. mit bestimmten Nachrichtentypen, die von diesen Bibliotheken definiert werden, kompatibel sind. Das ist deshalb so wichtig, weil das OSI-Referenzmodel und damit einhergehend Pakettypen für Ethernet, IP oder User Datagram Protocol (UDP) nicht im Omnet++ Framework direkt implementiert sind sondern in Bibliotheken wie INET.

A4. Neben INET und Nesting *soll* eine Integration von weiteren (noch unbekannten) Bibliotheken möglich sein, so dass deren Funktionsumfang von in Julia geschriebenen Modulen nutzbar ist.

A5. Die Anzahl der abhängiger Bibliotheken *soll* möglichst gering sein. Zum einen um bei sich ändernden Abhängigkeiten möglichst wenig Code neu schreiben zu müssen und um die Julia Erweiterungen so später einfacher in andere Projekte integrieren zu können.

A6. Die Julia Anbindung *soll* möglichst performant sein mit einem relativ geringen Overhead gegenüber C++.

Verteilte oder geschlossene Simulation

Beim Betrachten des aktuellen Standes der Technik, fällt auf, dass sich andere Projekte [**hla**; MK13; RB18a] zur Simulation von CPS vorwiegend, wenn nicht sogar ausschließlich im Bereich der Co-Simulation bzw. verteilten Simulation einordnen lassen. Auf Julia bezogen würde dies bedeuten, dass neben Omnet++ ein eigenständiger Julia Prozess gestartet werden müsste, der über eine eigene Simulationsschleife verfügt. Omnet++ und der Julia Prozess würden dann mittels Interprozesskommunikation miteinander interagieren. Bei [MK13] wird die Interprozesskommunikation mittels eines eigenen auf TCP basierenden Protokolls realisiert, bei [RB18a] durch Verwendung einer HLA konformen Middleware und [ZDFH16] nutzt eine virtuelle Gerätedatei. Als alternative zur

Co-Simulation wäre eine direkte Einbettung von Julia in das Omnet++ Framework als geschlossene Simulation denkbar. Julia stellt hierzu eine C-API zur Verfügung [Jul19] mit der sich das realisieren lassen würde.

Während der Vorteil von Co-Simulation darin liegt, dass sehr spezifische und/oder weit verbreitete Simulationswerkzeuge verwendet werden können, erlaubt geschlossene Simulation mit nur einem Simulationswerkzeug eine weitaus simplere Anwendung mit einfacherer Konfiguration und Installation. Da es sich bei Julia um eine relativ flexible Programmiersprache und nicht etwa um eine spezialisierte Simulationsumgebung handelt, würde dieser Vorteil der Co-Simulation hier nicht zum tragen kommen. Zusätzlich würde sich der Overhead durch die Interprozesskommunikation negativ auf die Performance auswirken, denn wie bereits im Grundlagen Kapitel erwähnt wurde, ist eine Parallelisierung von eventbasierter Simulation sehr schwierig. Co-Simulation in Verbindung mit Omnet++ könnte also auch zu keiner Steigerung der Parallelität führen. In [MK13], bei dem Omnet++ über Co-Simulation mit Matlab gekoppelt wurde, konnten so z.B. keine asynchrone Kommunikation verwendet werden. Stattdessen wurden die Zeiten beider Simulationsumgebungen über ein Request-Response-System synchronisiert.

Die direkte Einbettung von Julia in Omnet++ über ein Plugin-System schien deswegen die bessere Wahl zu sein. Dadurch ist im Vergleich zur herkömlichen Verwendung von Omnet++ kein komplexes Setup notwendig was die Adaption des Systems für Anwender enorm vereinfacht.

Spezifisches oder allgemeines Modul Interface

Die Funktionalität von Omnet++ kann durch die Spezialisierung von Simple-Modules erweitert werden. Um Julia Code in eine Simulation einbetten zu können, ist deswegen ein spezielles Simple-Module erforderlich, dass nicht direkt Anwendungslogik implementiert, sondern das Methodenaufrufe auf dem Modul über Callbacks an andere, in Julia implementierte Methoden, weiterleitet. Hierbei stellt sich die Frage welches Simple-Module man als Ausgangsbasis für das Interface verwendet. Man könnte ein allgemeines Simple-Module verwenden oder bereits spezialisierte Varianten wie z.B. eine UDP Anwendung aus dem INET Framework. Man könnte sogar ein odere mehrere eigens implementiertes Modul(e) speziell für Regelungsysteme verwenden.

Durch Verwendung spezifischerer Omnet++ Module als Grundlage für das Julia Interface, wäre dieses besser für einen bestimmten regelungstechnischen Anwendungsbereich zugeschnitten. Dadurch könnten NCS zunächst einfacher und schneller in Julia implementiert werden. Doch mit der höheren anwendungsbezogenheit kommt es auch gleichzeitig zu Einschränkungen in der Flexibilität. Wählt man z.B. eine UDP Anwendung als Grundlage, kann später das Transportprotokoll nicht mehr angepasst werden ohne Änderungen im C++ Code des Julia Interfaces vorzunehmen. Verwendet man aber andererseits ein allgemeines Simple-Module, so unterliegt man keinen großartigen Einschränkungen, ist aber unter Umständen dazu gezwungen mehr Code zu schreiben.

Letztendlich schien die Verwendung eines allgemeinen Simple-Modules als Grundlage für das Julia Interface am sinnvollsten zu sein. Der Entwicklungsaufwand für beide Varianten ist ähnlich hoch. Zudem können die Nachteile einer weniger anwendungsbezogenen Schnittstelle dadurch ausgeglichen werden, dass auch eine Spezialisierung der Module in Julia erfolgen kann. Neben NCS könnten also auch beliebige andere Systeme in Julia implementiert werden. Die Omnet++ Simple-Modules, die als Bestandteil der Julia Anbindung Methodenaufrufe an Julia Callbacks weiterleiten, nennen wir im weiteren Verlauf Julia-Simple-Modules.

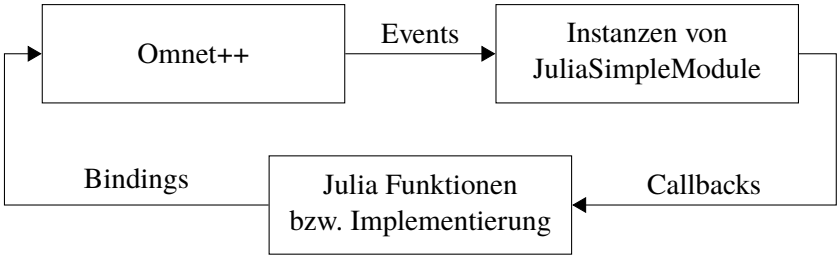


Abbildung 4.1.: JLSM Komponenten

Beteiligte Komponenten und deren Interaktion

Bei den Julia Erweiterungen für Omnet++ sind drei wichtige Komponenten beteiligt. Zum einen die aktuelle Omnet++ Simulation von der die Eventhandler der Julia-Simple-Modules aufgerufen werden. Zum anderen noch die Julia-Simple-Module-Instanzen welche die Aufrufe von Eventhandlern z.B. in Form von Callbacks mit Julia Funktionen verknüpfen. Zu guter letzt gibt es noch die Julia Funktionen die die Anwendungslogik der Julia-Simple-Modules implementieren. Die Funktionsaufrufe bzw. der Datenfluss von Simulation zu Julia-Simple-Modules und von Julia-Simple-Modules zu Julia Funktionen ist also offensichtlich. Allerdings müssen auc die Julia Funktion den Zustand der Simulation beeinflussen können, indem diese z.B. Events erzeugen. Andernfalls könnten die Julia Funktionen lediglich passiv Statistiken erfassen, wären aber nicht richtig in der Simulation integriert.

Für die Kommunikation von Julia Funktionen zu Omnet++ Simulation gibt es zwei Möglichkeiten. Die erste Möglichkeit wäre, dass die Julia Funktionen einen Wert an das entsprechende Julia-Simple-Module zurückgibt. Dieses könnte anhand des Rückgabewerts eine Aktion bestimmen und diese stellvertretend für die Julia Funktion ausführen. Die andere Möglichkeit wäre allgemeine Julia Bindings für Omnet++ bereitzustellen.

In Anbetracht dessen, dass im vorherigen Schritt schon die Entscheidung gefällt wurde das Interface möglichst allgemein zu halten, wäre es auch nicht sinnvoll nur relativ restriktive Interaktionsmöglichkeiten mit der aktuellen Omnet++ Simulation zu haben. Allgemeine Julia Bindings zu Omnet++ und eventuell noch anderen Bibliotheken wie INET oder Nesting steigern die Flexibilität enorm. Mit der Entscheidung Julia Bindings für Omnet++ zu verwenden, ergeben sich so die in Abbildung 4.1 dargestellen Komponenten und Interaktionen.

Julia Bindings

Die Bindings zu Omnet++ sind ein weiterer Punkt, der sich auf verschiedene Arten realisieren lässt. Auf niederster Abstraktionsebene könnte man C-Funktionen aus Julia heraus direkt aufrufen [Jul18]. Entweder direkt über den Namen einer exportierten Bibliotheksfunktion oder durch Verwendung eines Funktionszeigers. Neben diesen bereits nativ vorhandenen Julia Funktionen, gibt es auch Bibliotheken bzw. Julia Pakete mit denen sich Teile dieser Aufgabe vereinfachen lassen, nämlich Cxx.jl [Fis18] und CxxWrap.jl [Jan19].

Cxx.jl stellt ein FFI für C++ zur Verfügung, mit diesem lassen sich C++ Funktionen aus Julia heraus direkt aufrufen. Für eine optimale Integration können die C++ Aufrufe über in Julia geschriebene Wrapper-Funktionen gekapselt werden. Der für die Bindings relevante Code kann somit vollständig in Julia geschrieben werden. Einen ähnlichen Ansatz verfolgt CxxWrap.jl. Bei Verwendung dieser Bibliothek werden die Bindings jedoch nicht in Julia sondern im C++ Code definiert.

Letzten Endes wurde für die Implementierung CxxWrap.jl verwendet, obwohl Cxx.jl zunächst einfacher in der Handhabung zu sein schien. Das lag zum einen an Bedenken daran, dass das FFI von Cxx.jl relativ viele Bugs enthalten könnte, da es relativ schwierig ist alle nötigen Features einer so komplexen Sprache wie C++ korrekt abzudecken. So zeigte sich bei näherer Betrachtung, dass von Cxx.jl Features wie das Instanziieren von Templates im Moment nicht richtig unterstützt wird [Sch16]. Ein weiterer Punkt, der gegen Cxx.jl spricht, sind die Mängel bei der Unterstützung von Windows [Dan15]. Cxx.jl ist zudem momentan nicht auf Julia v1.0.0 und v1.1.0 lauffähig und scheint in der Ausführungsgeschwindigkeit und/oder Startup Zeit wesentlich langsamer zu sein. Bei einem Minimalbeispiel wurde für Cxx.jl eine durchschnittliche Ausführungszeit von 5.494s und bei CxxWrap.jl eine durchschnittliche Ausführungszeit von 1.089s gemessen. Allerdings musste hierbei Cxx.jl mit Julia v0.6.4 und CxxWrap.jl mit Julia v1.1.0 getestet werden aufgrund der verwendeten Paketversionen. Neben all den anderen Punkten war am Ende jedoch ausschlaggebend, dass CxxWrap.jl wesentlich einfacher für das Einbetten von Julia Code in C++ bzw. Omnet++ angepasst werden konnte. Denn ursprünglich wurde keine der beiden Bibliotheken für die Bereitstellung von Bindings von eingebetteten Julia Code zu einer einbettenden C++ Anwendung entworfen, sondern lediglich dafür in C++ geschriebene Bibliotheksfunktionen innerhalb von Julia Anwendungen nutzbar zu machen. Die Adressen der Funktionen können somit nicht einfach einer Symboltabelle entnommen werden, sondern müssen über einen Callback an das Julia Ökosystem kommuniziert werden. Während bei Cxx.jl die Adresse jeder abzubildenden C++ Funktion in Julia bereitgestellt werden müssten, reicht es bei CxxWrap.jl aus lediglich die Adressen von Funktionen, die als Einstiegspunkte zur Erzeugung der Bindings ganzer Module dienen, über einen Callback in Julia bereitzustellen. Da innerhalb von C++ wegen des statischen Typsystems keine Introspektion von Typen zur Laufzeit möglich ist, müssen die zu exportierenden Funktionen hart einkodiert werden. Das bedeutet, dass wenn z.B. 2 Julia Module a 100 C++ Funktionen verfügbar gemacht werden sollen, bei der Verwendung von Cxx.jl 200 Julia Callbacks mit fest kodierten Adressen als Parameter aufgerufen werden müssten. Zusätzlich müssten 200 individuelle Wrapper Funktionen in Julia definiert werden. Bei CxxWrap.jl hingegen müssten nur die Adressen von den 2 als Einstiegspunkte dienenden Funktionen für die Modulerzeugung an Julia weitergegeben werden. Selbstverständlich müssten auch noch die 200 bereitzustellenden Funktionen im C++ Code definiert werden. Anhand dieses Beispiels ist leicht zu erkennen, dass bei Verwendung Cxx.jl eine gewisse Redundanz bei der Definition der Bindings erforderlich wäre.

Speicherverwaltung in Julia und C++

Ein Problem, das man bei der Einbettung von Julia Code in ein C++ Programm überwinden muss liegt im Bereich der Speicherverwaltung. In C++ wird der Lebenszyklus langlebiger Objekte auf dem Heap oft manuell gesteuert. Ansonsten ist auch die Verwendung von Reference Counting mithilfe von Smart Pointern üblich. In Julia hingegen wird die Speicherverwaltung langlebige Objekte von einem Garbage Collector (GC) gehandhabt. Das kann zu Problemen im Zusammenspiel von Julia und C++ führen; Wenn keine entsprechenden Vorkehrungen getroffen werden, können hängende

Zeiger (engl. Dangling Pointer) entstehen, also Zeiger, die auf einen ungültigen Speicherbereich adressieren bzw. auf kein gültiges Objekt verweisen. Das kann beispielsweise der Fall sein, wenn aus der C++ Anwendung eine Variable aus dem eingebetteten Julia System adressiert wird. Da der Julia GC im allgemeinen nicht von der Existenz von den C++ Zeigern weiß welche Variablen aus dem Julia Ökosystem adressieren, kann er diese bei der Speicherverwaltung nicht berücksichtigen und somit ein Objekt löschen, dass zwar keine Julia Referenzen mehr besitzt jedoch noch C++ Referenzen. Die Auswirkungen dieser hängenden Zeiger ist im allgemeinen nicht-deterministisch und sind oftmals schwer zu debuggen. Während bei bestimmten Ausführungssequenzen überhaupt keine Fehler auftreten, kann es bei anderen Ausführungssequenzen zu Speicherkorruptionen oder Zugriffsverletzungen (Segmentation Faults) kommen.

Julia stellt deswegen schon von Haus aus die Möglichkeit zur Verfügung bestimmte Speicherbereiche von der Verwaltung durch den GC auszuschließen und zu schützen [Jul19]. Problematisch ist jedoch, dass diese von Julia bereitgestellten Features ausschließlich Stapel-basiert arbeiten. Das heißt am Anfang einer C++ Funktion werden die Adressen der zu schützenden Speicherbereiche auf den Stapel gelegt und gegen Ende der Funktion wieder entfernt um so dem GC die Kontrolle über das Speichermanagement wieder zurückzugeben. Exemplarisch wird das Funktionsprinzip in Anhang X näher erläutert. Für viele Anwendungsbereiche mag dieses Verfahren ausreichend sein, will man jedoch aus langlebigen C++ Objekten vom Julia Subsystem verwaltete Objekte referenzieren stößt dieses Verfahren an seine Grenzen. Mit dem Stapel-basierten Schutz von Variablen vor dem Julia GC lässt sich dieser Anwendungsfall nicht realisieren. Zur Lösung dieses Problems kann aber ein globaler Assoziativspeicher oder eine andere Datenstruktur in Julia verwendet werden. Durch das Speichern von Julia Objekten (bzw. Referenzen) in diesem Speicher wird so der Julia GC daran gehindert diese Objekte zu löschen. Dieses Verfahren wird auch bei der Implementierung der JLSM Bibliothek verwendet.

4.1.2. Architektur

Die Hauptfunktionalität der Julia Erweiterungen für Omnet++ bzw. der JLSM Bibliothek wird durch die in Abbildung 4.2 dargestellten Klassen implementiert, dabei wurden die wichtigsten Methoden und Attribute dieser Klassen eingezeichnet.

JuliaUtil: Die *JuliaUtil* Klasse ist dafür zuständig Julia Module zu generieren die als Bindings zu C++ Funktionen dienen, damit diese aus eingebettetem Julia Code verwendet werden können. Hierzu wird zuerst mit der *createModule* Methode ein neues leeres Modul erstellt. Dieses kann anschließend mit der *extendModule* Funktion um weitere Funktionen bzw. Bindings erweitert werden. Zusätzlich dienen die Funktionen *addModuleDependency* und *addExtensionDependency* dazu Abhängigkeiten zwischen Modulen und Modul-Erweiterungen festzulegen, damit diese zum entsprechenden Zeitpunkt korrekt erzeugt werden können. Die Klasse enthält außerdem die Funktionalität zum Laden des einzubettenden Julia Codes. Mit der *loadJuliaFile* Methode kann hierzu eine Datei mit Julia Code geladen werden. Alle diese Funktionen sind hierbei idempotent, d.h. dass einmaliges Aufrufen von beispielsweise der *loadJuliaFile* Methode den gleichen Effekt hat, wie das mehrmalige Aufrufen dieser Methode (mit gleichem Parameter). Da die *JuliaUtil*-Klasse für das Setup des Julia Subsystems zuständig ist, wurde diese mit dem Singleton-Pattern implementiert. Es gibt also nur eine global verfügbare Instanz dieser Klasse welche man mit der statischen *getInstance* Methode abrufen kann. Auch wenn die Verwendung des Singleton Patterns durchaus umstritten ist [Tan16] schien es hier angebracht zu sein. Es gibt nämlich nur maximal ein Julia Subsystem

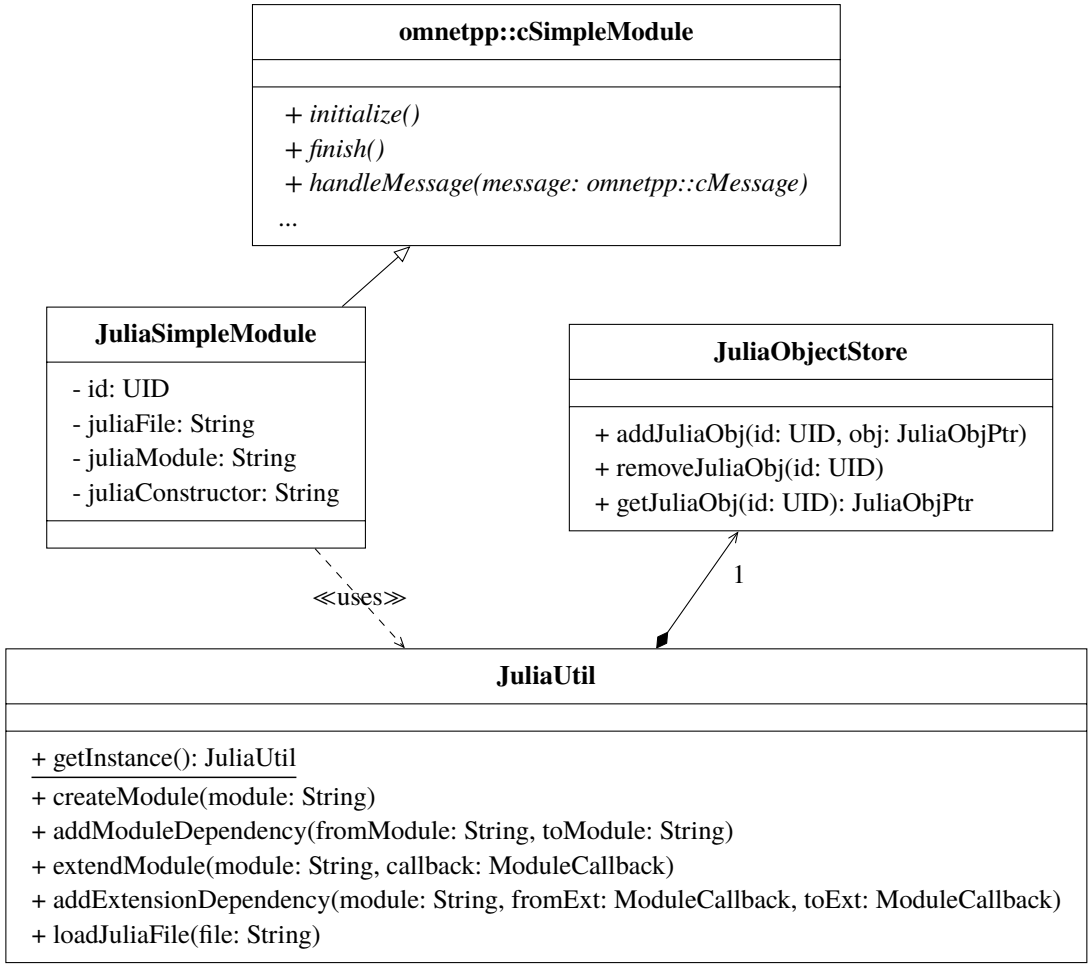


Abbildung 4.2.: JLSM Klassendiagramm

innerhalb einer Anwendung und zweitens gab es keine zuverlässige Möglichkeit einen Hook in der Initialisierungsroutinge von Omnet++ zu platzieren. Diese Randbedingungen machen es somit nicht vorhersehbar an welcher Stelle die JuliaUtil-Klasse zum ersten mal instanziiert wird.

JuliaObjectStore: Die *JuliaObjectStore* Klasse setzt das im vorherigen Kapitel vorgestellte Prinzip zum Schutz von Variablen aus dem Julia Subsystem vor dem Julia GC um. Letztendlich handelt es sich hierbei um einen Assoziativspeicher, der IDs auf Zeiger abbildet, welche Objekte aus dem Julia Subsystem referenzieren. Wird ein Zeiger auf ein Julia Objekt mit der *addJuliaObj* Methode dem Assoziativspeicher hinzugefügt, kann das referenzierte Objekt nicht mehr vom Julia GC gelöscht werden. Mit der *getJuliaObj* Methode kann die Referenz auf ein Objekt anhand dessen ID wieder abgefragt werden. Die *removeJuliaObj* dient dazu Referenzen wieder aus dem Assoziativspeicher zu löschen, sodass diese wieder vom Julia GC entfernt werden können falls diese über keine anderweitigen Referenz aus dem Julia Subsystem verfügen.

JuliaSimpleModule: Bei der *JuliaSimpleModule*-Klasse, handelt es sich um die einzigste Klasse die von Anwendern der JLSM Bibliothek direkt verwendet wird, denn es handelt sich hierbei um eine Unterklasse der *cSimpleModule* Klasse aus Omnet++. Instanzen von *JuliaSimpleModule* erben alle Methoden aus *cSimpleModule*. Anstatt deren Funktionalität aber selbst zu implementieren

werden die Aufrufe an öffentliche Funktionen an vorher definierte Julia Funktionen weitergeleitet. Hierzu besitzt die Klasse das *juliaFile* Attribut, dass den Pfad zur Julia-Quellcode-Datei enthält. Das *juliaModule* Attribut gibt an, welches Julia-Modul aus der Julia Datei verwendet werden soll. Auf diese Weise ist es möglich die Implementierung mehrerer *JuliaSimpleModule*-Instanzen innerhalb einer Datei zu definieren falls dies erwünscht ist. Der *juliaConstructor* spezifiziert eine Fabrikmethode aus der eben genannten Datei und dem Modul, dass dazu verwendet werden kann das *cSimpleModule* in Julia um weitere Attribute zu ergänzen sowie analog zu dem C++ Konstruktor verwendet werden kann.

Der Lebenszyklus einer Instanz des *JuliaSimpleModule* Typs ist im Sequenzdiagramm in Abbildung 4.3 veranschaulicht. Im allgemeinen werden die *JuliaSimpleModule* Objekte zu Beginn einer Simulation in der Initialisierungsphase instanziiert. Hierzu erzeugt das Simulationsobjekt vom Typ *omnetpp::cSimulation* ein neues *JuliaSimpleModule* Objekt und ruft dessen Konstruktor auf. Anschließend ruft das Simulationsobjekt die *numInitStages* Methode auf um die Anzahl der zu initialisierenden Protokollschichten zu bestimmen. Eigentlich ist die Anzahl der zu initialisierenden Schichten von der Julia Implementierung der *numInitStages* Methode abhängig. Allerdings ist der Zugriff auf die Parameters der zugehörigen NED Datei und damit die Bestimmung der Julia Quellcode Datei erst in der *initialize* Methode möglich. Somit ist es an dieser Stelle unglücklicherweiße noch nicht möglich zu wissen wie viele Schichten tatsächlich initialisiert werden müssen. Deswegen wird von der *numInitStage* Methode von *JuliaSimpleModule* einfach ein absurd hoher Wert k zurückgegeben von dem ausgegangen wird, dass keine Implementierung eines *JuliaSimpleModule* erfordert, dass mehr als k Schichten initialisiert werden müssen. Im Moment wurde für k der Wert 16 festgelegt, da im INET Framework die Anwendungsschicht den Wert 10 besitzt. Der Wert für k kann aber zu einem späteren Zeitpunkt immer noch relativ einfach erhöht werden falls dies erforderlich werden sollte. Nachdem das *cSimulation* Objekt weiß, dass es k zu initialisierende Schichten gibt, ruft dieses für jede Schicht i mit $i \in \{0, ..., k-1\}$ die *initialize* Methode auf. Das *JuliaSimpleModule* wertet nun beim ersten Aufruf der *initialize* Methode, also bei der Initialisierung von Schicht 0, zuerst die NED Parameter des *JuliaSimpleModule* aus und lädt die Julia Quellcode Datei und somit die Julia Implementierung des Moduls. Danach wird die in Julia definierte Konstruktorfunktion aufgerufen. Diese gibt einen beliebigen Julia Typ zurück der als Stellvertreter des *JuliaSimpleModule* Objekts innerhalb des Julia Subsystem dient. Meist ist dies ein *struct*-Typ der eine Referenz auf das JuliaSimpleModule enthält sowie weitere Attribute um die das *JuliaSimpleModule* im Julia Subsystem erweitert wurde. Dieser Julia Wert wird in Verbindung mit der ID des *JuliaSimpleModule* innerhalb des *JuliaObjectStore* gespeichert. Dadurch wird dieser Wert nicht einfach durch den Julia GC gelöscht. Als nächstes ruft das *JuliaSimpleModule* die *numInitStages* Methode der Julia Implementierung auf um die tatsächliche Anzahl n der zu initialisierenden Schichten zu bestimmen. Nachdem dieses Setup erfolgreich beendet ist, leitet das *JuliaSimpleModule* Objekt die *initialize* Aufrufe für jedes $i < n$ an die Julia Implementierung weiter. Aufrufe von *initialize* mit $n < i < k$ werden vom *JuliaSimpleModule* Objekt einfach ignoriert und nicht weitergeleitet. Auf diese Weise scheint die Initialisierung für die Julia Implementierung des *JuliaSimpleModule* exakt gleich abzulaufen wie dies bei der Implementierung einer Unterklasse von *cSimpleModule* der Fall ist.

Nachdem die Initialisierung aller *cSimpleModule* Instanzen und somit auch *JuliaSimpleModule* Instanzen durch das *cSimulation* Objekt abgeschlossen ist, folgt die Durchführung der eigentlichen Simulation. Hierbei entnimmt das *cSimulation* Objekt solange Events aus der Event-Warteschlange bis entweder keine Events mehr vorhanden sind oder der Zeitpunkt des nächstes Events die Gesamtlänge der Simulation überschreitet. Für diese Events die auf der zu betrachtenden *JuliaSimpleModule*

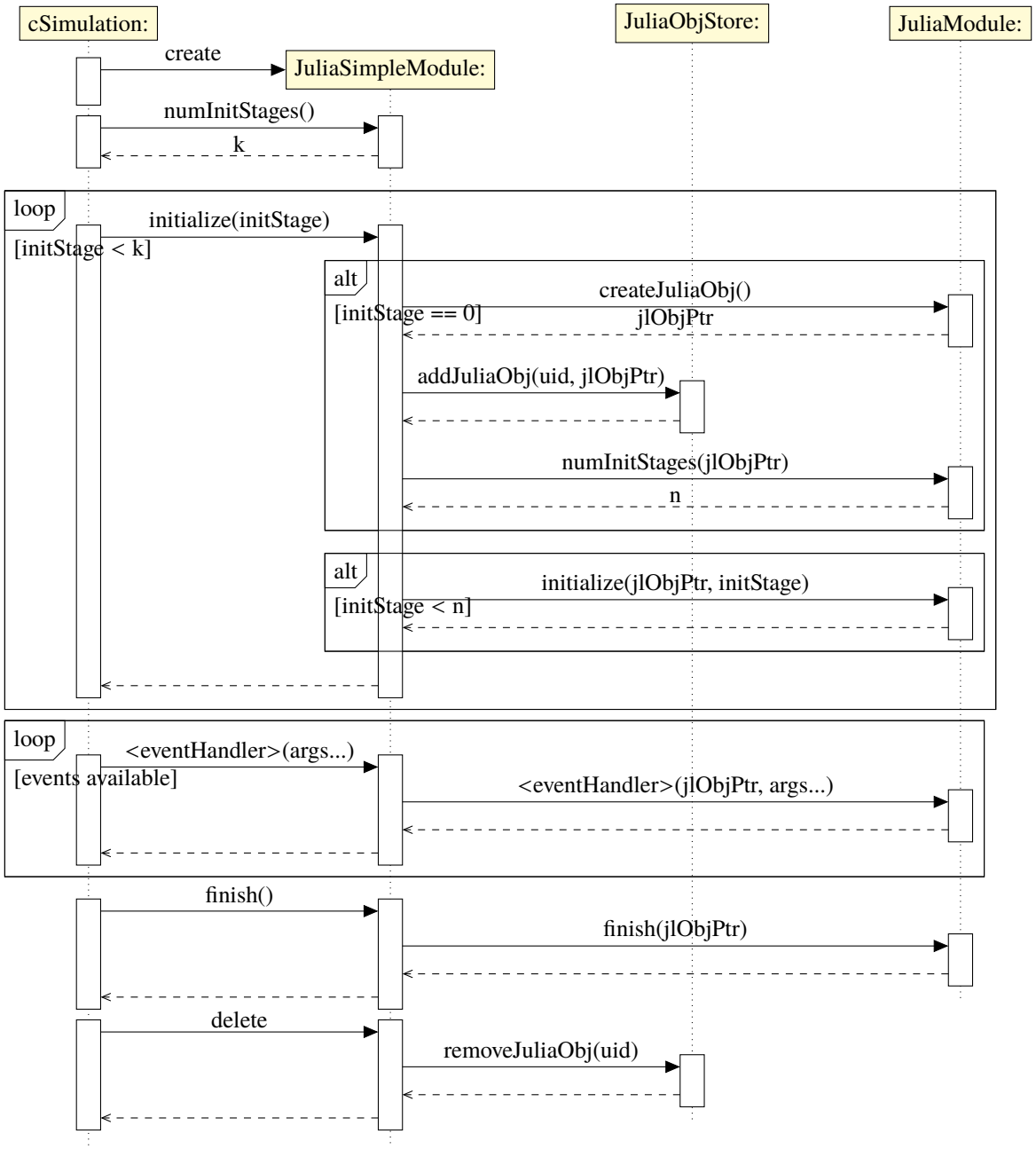


Abbildung 4.3.: JLSM Lebenszyklus von JuliaSimpleModule Instanzen

Instanz eintreten, wird der entsprechende Eventhandler aufgerufen. Meist handelt es sich hierbei um die *handleMessage* Methode die von *cSimpleModule* geerbt wird. Diese Eventhandler werden an eine gleichnamige Methode in der Julia Implementierung weitergeleitet falls diese existieren. Als Parameter bekommen diese Funktionen neben den Parametern des ursprünglich am *JuliaSimpleModule* ausgelösten Eventhandlers noch den zu dem *JuliaSimpleModule* anhand dessen ID zugeordneten Julia Wert. Dieser Schritt kann auch als eine Art idiomatischen Übersetzung von der in C++ gebräuchlichen Syntax in die von Julia verwendete Syntax betrachtet werden. Die entsprechenden Unterschiede wurden bereits in Kapitel 2.5 erläutert.

Wurde das Ende der Simulation erreicht, so wird noch der *finish* Eventhandler des *JuliaSimpleModule* Objekt ausgelöst und dieser leitet den Aufruf daraufhin weiter an die Julia Implementierung. Zuletzt löscht nun das *cSimulation* Objekt die *JuliaSimpleModule* Instanz wodurch dessen Destruktor aufgerufen wird. Im Destruktor wird nun das zugehörige Julia Objekt aus dem *JuliaObjectStore* entfernet, sodass dieses letztendlich auch vom Julia GC entfernt werden kann.

4.2. Implementierung von Host Modulen zur Simulation eines inversen Pendels

Zur Implementierung von Regler und Regelstrecke eines inversen Pendels wurden jeweils ein VLAN fähiger Ethernet Host aus dem Nesting Framework als Basis verwendet. Dieser wurde jeweils um zwei Delay Module, sowie einem *JuliaSimpleModule* erweitert. Sowohl Regler als auch Regelstrecke sind also konzeptuell gleich aufgebaut wie in Abbildung 4.4 zu sehen ist. Die Implementierung des als App dienenden *JuliaSimpleModule* erfolgt zum Großen Teil analog zu Kapitel 2.4. Zur Übertragung der Zustands- und Steuerinformationen des Pendels müssen lediglich Vektoren übertragen werden. Das hierzu notwendige Paketformat besteht deswegen aus einem 4B Zähler, sowie *n* 8B großen Fließkommazahlen mit doppelter Genauigkeit.

Theoretisch wäre es auch möglich die Pakete nicht beim Host, sondern über einen Trunk Port am Switch zu taggen. Allerdings kann das Tagging im Host flexibler verwendet werden. Beispielsweise wird in [LCD+19] ein Zeitschlitz basiertes Verfahren verwendet um die Stabilität mehrere NCS zu gewährleisten. Hierfür werden die gesendeten Pakete abhängig von deren Sendezeitpunkt als opportunistisch oder deterministisch gekennzeichnet und die Pakete müssen dementsprechend mit unterschiedlichen PCP Werten versehen werden.

Falls es zu Paketverlust oder der Überschreitung der maximalen Sampling Periode des Pendels kommt, können zwei verschiedene Policies für die Regelstrecke konfiguriert werden. Empfängt die Regelstrecke innerhalb einer Sampling Periode keine Steuerungsinformationen vom Regler, werden bei Verwendung der Hold Policy einfach die zuletzt empfangenen Eingangswerte werte zur Steuerung des Pendels verwendet. Bei der Zero Policy hingegen wird hier stattdessen der Nullvektor verwendet. Welches der beiden Verfahren Paketverluste besser kompoensieren kann ist vom Systemmodell abhängig.

Um das Pendel für die Simulation einfach konfigurierbar zu machen, wird das Systemmodell über NED Parameter festgelegt. Hierzu erben sowohl *ControllerApp* als auch *PlantApp* von *PendulumModel*, das wiederum von *JuliaSimpleModule* erbt. Zu sehen sind die NED Komponenten in Listing 4.1. In der Regelstrecke (Plant) wird außerdem noch der Initialzustand des Pendel festgelegt. Standardmäßig befindet sich dieses zu Beginn in einem neutralen Zustand, also $(x, \dot{x}, \theta, \dot{\theta}) = (0, 0, 0, 0)$.

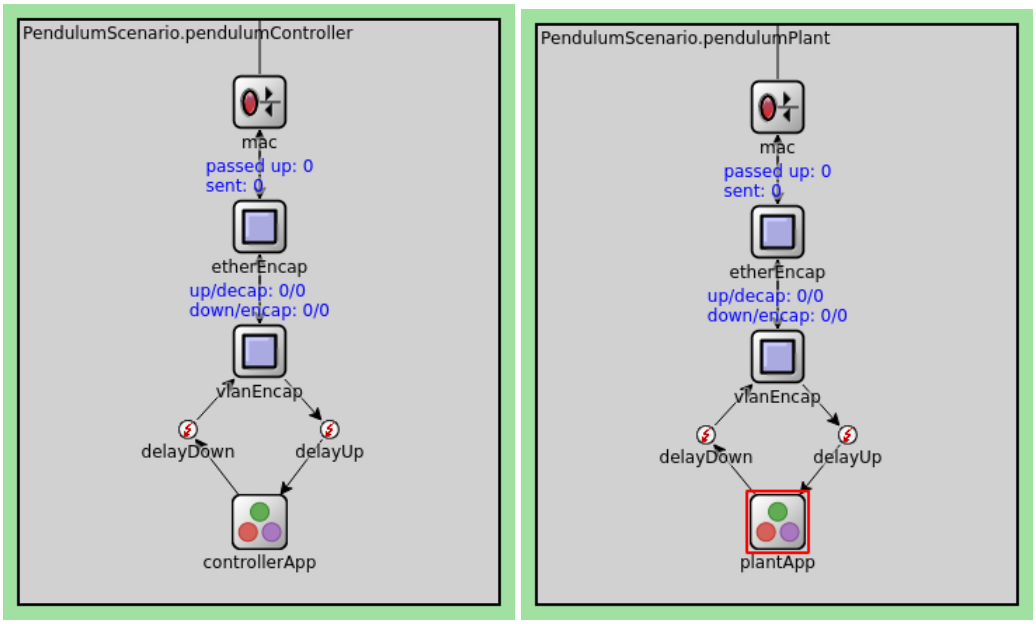


Abbildung 4.4.: Omnet++ Module des Reglers und der Regelstrecke

Zusätzlich können noch statistische Daten über Zustand, LQR Kosten, Round Trip Time (RTT) oder gesendete Zustandsinformationen und verlohrene bzw. zu spät eingetroffene Steuerungsinformation erfasst werden.

Im Moment werden die Ziel-Regler bzw. die Ziel-Regelstrecken anhand deren MAC Adressen festgelegt. Eine mögliche Verbesserung wäre die Implementierung von Regler und Regelstrecke als UDP Applikation. Dadurch könnten diese Anhand einer Internet Protocol (IP) Adresse identifiziert werden. Die Verwendung des UDP Protokolls wäre auch realitäts´näher als die momentane Verwendung von Ethernet Frames zum Transport der Daten. Aufgrund des beschränkten zeitlichen Rahmens wurde dies jedoch bis jetzt nicht umgesetzt.

Listing 4.1 NED Komponenten des inversen Pendels

```
simple InvertedPendulumModel extends JuliaSimpleModule {
  parameters:
    double samplingPeriod @unit(s);
    double mass @unit(kg);
    double distanceCenterOfMass @unit(m);
    double momentOfInertia @unit(kgm);
    string lqrStateCoefficients;
    double lqrInputCoefficient;
    string noiseCovarianceMatrix; }
simple PlantApp extends InvertedPendulumModel {
  parameters:
    @signal[pendulumAngle](type=double);
    @signal[pendulumAngularVelocity](type=double);
    @signal[pendulumPosition](type=double);
    @signal[pendulumVelocity](type=double);
    @signal[lqrCost](type=double);
    @signal[ctrlInputMissed](type=long);
    @signal[ctrlInputSend](type=long);
    @signal[roundTripTime](type=simtime_t);
    @statistic[pendulumPosition](record=vector,stats);
    @statistic[lqrCost](record=vector,stats);
    @statistic[ctrlInputMissCount](source=ctrlInputMissed; record=vector(sum),stats);
    @statistic[ctrlInputSendCount](source=ctrlInputSend; record=vector(sum),stats);
    @statistic[roundTripTime](record=vector,stats);
    juliaFile = "julia/PlantApp.jl";
    juliaModule = "PlantAppModule";
    juliaConstructor = "createPlantApp";
    // cart position; cart velocity; pendulum angle; pendulum angular velocity
    string initialState = default("0.0;0.0;0.0;0.0");
    string controllerMAC;
    int pcp = default(0); // PCP value of Ieee802.1Q tagged frames
    bool holdPolicyEnabled = default(true);
  gates:
    input in;
    output out; }
simple ControllerApp extends InvertedPendulumModel {
  parameters:
    @signal[estimatedPendulumAngle](type=double);
    @signal[estimatedPendulumAngularVelocity](type=double);
    @signal[estimatedPendulumPosition](type=double);
    @signal[estimatedPendulumVelocity](type=double);
    @statistic[estimatedPendulumPosition](record=vector,stats);
    juliaFile = "julia/ControllerApp.jl";
    juliaModule = "ControllerAppModule";
    juliaConstructor = "createControllerApp";
    string plantMAC;
    int pcp = default(0);
  gates:
    input in;
    output out; }
```

5. Evaluation

Dieses Kapitel beschäftigt sich mit der Auswertung der im Kapitel 4 beschriebenen Implementierung des Reglers bzw. Regelstrecke anhand eines einfachen Proof of Concept Szenarios. Hierfür wird ein NCS in einem IEEE 802.1Qbv fähigen Netzwerk simuliert und die Auswirkungen von Crosstraffic in Verbindung unterschiedlicher Netzwerkschedulingverfahren auf die Regelgüte des Systems evaluiert. Anschließend wird noch die Performance von in Julia implementierten Omnet++ SimpleModules mit klassischen in C++ implementierten Modulen untersucht um den Overhead der JLSM Bibliothek zu bestimmen.

5.1. Simulation eines Networked Control Systems

Zur Simulation eines NCS verwenden wir ein inverses Pendel, dessen theoretische Grundlagen bereits in Kapitel 2.4 erläutert wurden. Beim inversen Pendel handelt es sich um ein relativ weit verbreitetes Standardbeispiel eines Regelungssystems, dass oft als Demonstrator verwendet wird. Anhand verschiedener Simulationsdurchläufen mit unterschiedlicher Konfiguration sollen die Auswirkung der verwendeten Policy für die Reglerstrecke, der Größe des Datenstaus im Netzwerks bzw. Queuing Delays sowie Prioritäts- und time-aware Schduling aus dem IEEE 802.1Qbv Standard auf die Regelgüte ausgewertet werden.

5.1.1. Testsetup

Für das Testsetup wurde eine einfache sternenförmige Topologie gewählt, bei der alle Host-Systeme über einen zentralen Switch miteinander verbunden sind. Die Host-Systeme umfassen sowohl einen Regler (Controller), als auch eine Regelstrecke (Plant) sowie 4 weitere Host-Systeme die ausschließlich als Traffic-Quellen zur Erzeugung von Crosstraffic dienen. Veranschaulicht wird dies in Abbildung 5.1. Hier sind auch die Flows zwischen Endsystemen eingezeichnet. Die Regelstrecke sendet Zustandsinformationen zum Regler und dieser sendet daraufhin Steuerinformationen zurück zur Regelstrecken. Im Schaubild sind diese Flows durch den grünen Pfeil dargestellt. Zusätzlich senden die 4 Traffic-Quellen regelmäßig Datenpakete zur Regelstrecke. Die vier hierdurch entstehenden Flows von Traffic-Quellen zu Regelstrecke können den Link vom Switch zu der Regelstrecke überlasten und sind im Schaubild rot markiert. Anhand der Konfiguration der Simulation kann die Größe des Datenstaus auf dem Link variiert werden.

Die hierbei vom Simulationsmodell berücksichtigten Delays sind in Schaubild 5.2 zu sehen. Für diese können sowohl verschiedene Wahrscheinlichkeitsverteilungen als auch konstante Werte konfiguriert werden. Der Processing Delay der implementierten Host-Module für Regler und Regelstrecke kann in aufsteigender als auch in absteigender Richtung des Netzwerkstacks seperat modelliert

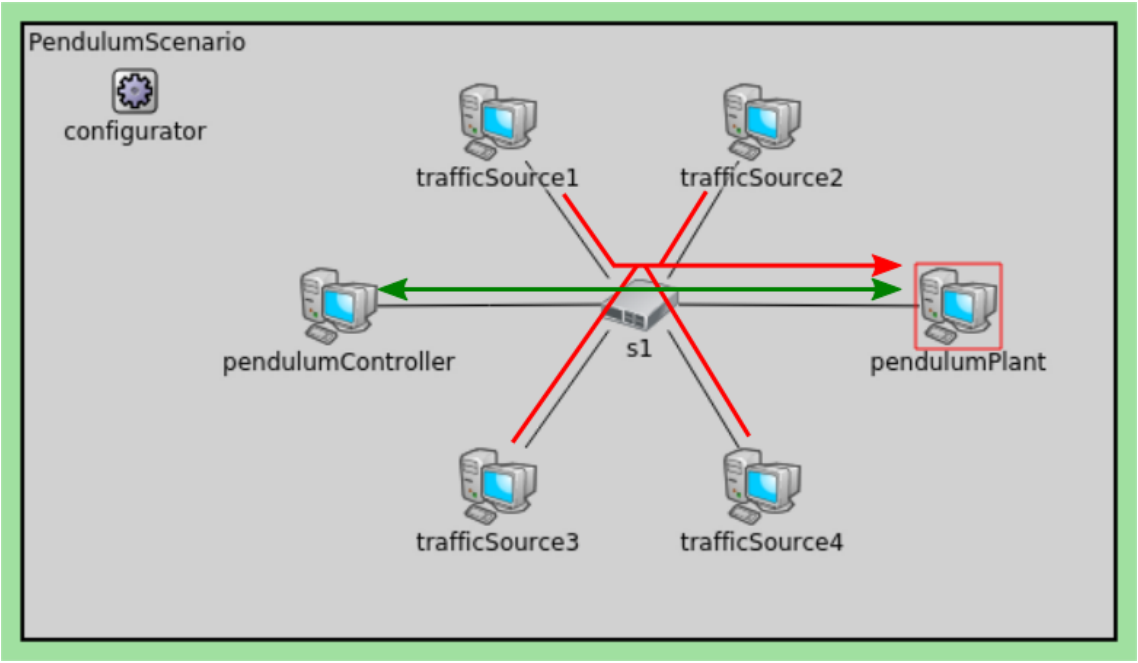


Abbildung 5.1.: NCS Evaluationsszenario mit Flows

werden. Der Switch der aus dem NeSTiNg Framework entstammt verfügt nur über eine Processing Delay Komponente. Im Switch findet die Auswertung des Processing Delay vollständig vor dem Einsortierens eines Ethernet Frames in das Queuing Netzwerk statt.

Für das Evaluationsszenario treffen wir ein paar Vereinfachungen bezüglich der Delays. Da Propagation Delay in etwa der Lichtgeschwindigkeit entspricht und somit nur ca. $2 \cdot 10^{-8}\text{s m}^{-1}$ innerhalb eines Kupferkabels benötigt, wird dieser für zukünftige Berechnungen vernachlässigt. Es gelte also:

$$\begin{aligned} \Delta t_{prop} &= \Delta t_{prop-link1} \\ &= \Delta t_{prop-link2} \\ &= 0 \end{aligned}$$

Zusätzlich fassen wir den Processing Delay der Hosts bzw. des Reglers und der Regelstrecke zusammen als $\Delta t_{proc-host}$. Der Processing Delay wird außerdem stets unabhängig von der Frame Größe modelliert und ist deswegen konstant. Die Processing Delays der Hosts in aufsteigender sowie absteigender Richtung des Stacks sind außerdem symmetrisch.

$$\begin{aligned} \frac{1}{2} \cdot \Delta t_{proc-host} &= \Delta t_{proc-h1-down} \\ &= \Delta t_{proc-h1-up} \\ &= \Delta t_{proc-h2-down} \\ &= \Delta t_{proc-h2-up} \end{aligned}$$

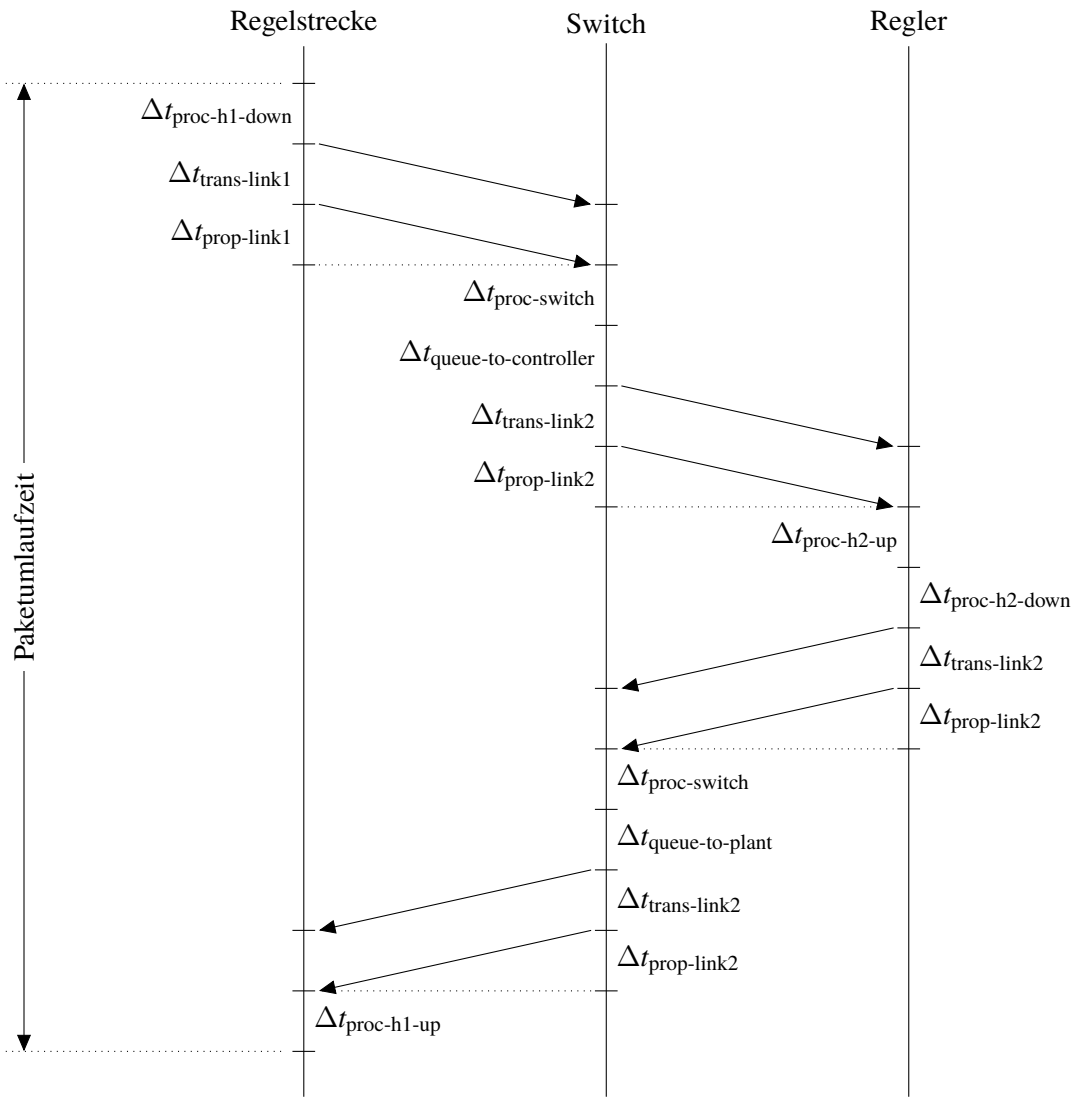


Abbildung 5.2.: Delays des Evaluationsszenarios

In [DK14] wurden für Hardware Switches experimentell ein mittlerer Wert von $3.787\mu s$ gemessen. Deswegen wurde der Wert für die Evaluation folgendermaßen festgelegt:

$$\Delta t_{proc-switch} = 4\mu s$$

In der Simulation werden auf allen Links Übertragungsgeschwindigkeiten von $100\text{Mbit}\,\text{s}^{-1}$ verwendet. Das ist nicht ganz konform mit dem IEEE 802.1Qbv Standard welcher die Verwendung des time-aware Shapers nur in Verbindung mit Übertragungsgeschwindigkeiten ab $1\text{Gbit}\,\text{s}^{-1}$ vorsieht. Allerdings müssen auf diese Weise in der Simulation viel weniger Events erzeugt werden und es wird weniger Rechenzeit und Speicher benötigt. Für den Transmission Delay gilt somit:

$$\begin{aligned}\Delta t_{trans}(x \text{ bit}) &= \Delta t_{trans-link1}(x \text{ bit}) \\ &= \Delta t_{trans-link2}(x \text{ bit}) \\ &= \frac{x \text{ bit}}{100\text{Mbit s}^{-1}}\text{s} = \frac{x}{100 \cdot 10^6}\text{s}\end{aligned}$$

Bei Ethernet Netzwerken beträgt zudem die Maximum Transmission Unit (MTU) 1.5kB und die kleinste Payload Größe eines Frames 46B. Für die Gesamtmenge der zu übertragenen Daten müssen aber noch der Header des Physical und des MAC bzw. Datalink Layers berücksichtigt werden. Das entspricht weiteren 26B. Zusätzlich folgt nach jedem Frame noch die Interpacket Gap (IPG), welche bei Fast Ethernet 12B entspricht. Insgesamt müssen also für die Übertragung eines Frames minimal 72 und maximal 1526 Byte-Zeiten eingeplant werden. Somit gilt:

$$\begin{aligned}min\ \Delta t_{trans}(x) &= \Delta t_{trans}(72\text{B}) = 5.76\mu\text{s} \\ max\ \Delta t_{trans}(x) &= \Delta t_{trans}(1526\text{B}) = 122.08\mu\text{s}\end{aligned}$$

Für das Systemmodel des NCS bzw. des inversen Pendels gelten die nachfolgenden Eigenschaften. Diese werden über NED Parameter der Simulation konfiguriert und sind somit auch sehr einfach für verschiedene Simulationsdurchläufe anzupassen.

$T_S = 10\text{ms}$	Sampling Rate des NCS
$(x_0, \dot{x}_0, \theta_0, \dot{\theta}_0) = (0, 0, 0, 0)$	Initialzustand des Pendels
$M = 1\text{kg}$	Schlittenmasse
$l = 0.5\text{m}$	Abstand des Massenschwerpunkts
$I = 0.25\text{kg}\cdot\text{m}^2$	Trägheitsmoment
$Q = (10, 1, 10, 1)^T$	LQR Zustandskoeffizienten
$R = 0.1$	LQR Eingangskoeffizient
$W = (10^{-6}, 10^{-6}, 10^{-3}, 10^{-2})^T$	Rauschkovarianzmatrix

Zur Übertragung der Zustandsinformationen der Regelstrecke sowie der Steuerungsinformationen des Reglers, wird das in Kapitel 4.2 vorgestellte Protokoll verwendet. Da für die Simulation nur Vektoren mit 1 bzw. 4 Fließkommazahlen mit doppelter Genauigkeit übertragen werden müssen und das Protokoll direkt auf Ethernet aufsetzt und somit keine großen Header Daten übertragen werden müssen, ist die Payload Größe der übertragenen Ethernet Frames maximal 40B. Insgesamt werden wegen der Mindestgröße des Payloads von 46B für die Übetragungen also stets 72 Byte Zeiten benötigt unter Berücksichtigung der Layer 1 Header und IPG berücksichtigt wird.

Um ein Systemmodel zu kreieren, bei dem sich die positiven Effekte von TSN hoffentlich besonders stark hervorheben, muss die RTT von Regelstrecke zu Regler besonders knapp bemessen sein. Hierzu wird der Processing Delay der Hosts so hoch wie möglich festgelegt. Es soll allerdings noch ein Puffer ε von 10µs berücksichtigt werden.

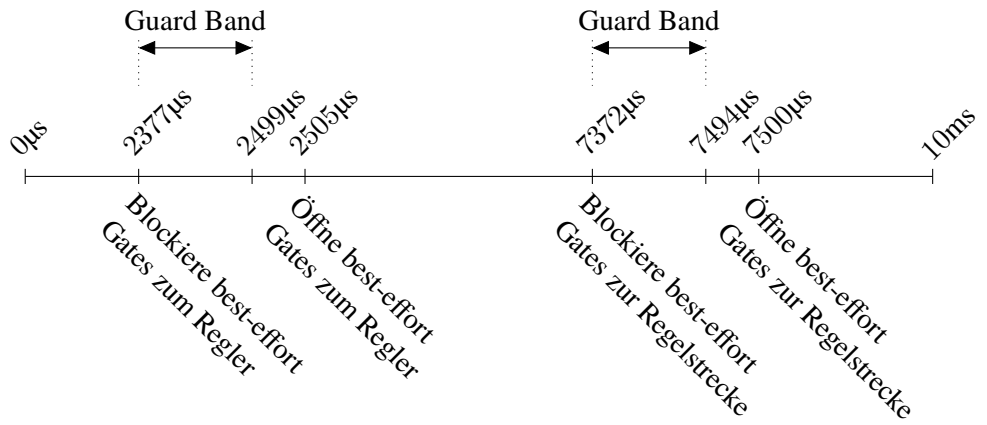


Abbildung 5.3.: Relevante Zeitpunkte für das TSN Schedule

$$
\begin{aligned}
RTT = 10ms - \varepsilon &= 2 \cdot \Delta t_{proc-host} + 2 \cdot \Delta t_{proc-switch} + 4 \cdot \Delta t_{trans}(72\text{B}) \\
\Delta t_{proc-host} &= 5ms - \frac{\varepsilon}{2} - \Delta t_{proc-switch} - 2 \cdot \Delta t_{trans}(72\text{B}) \\
&= 5ms - 5\text{µs} - 4\text{µs} - 2 \cdot 5.76\text{µs} \\
&= 4979.48\text{µs}
\end{aligned}
$$

In bestimmten Simulationskonfiguration wird der time-aware Shaper aus dem IEEE 802.1Qbv Standard verwendet. Hierzu müssen für die zwei Links zwischen Switch und Regelstrecke, sowie Switch und Regler jeweils ein Schedule erstellt werden. In Abbildung 5.3 sind die für das Schedule relevanten Zeitpunkte eingezeichnet. Diese lassen sich anhand der Delays in Abbildung 5.2 einfach berechnen. Zusätzlich zur Übertragungszeit der Daten muss noch ein Guard Band eingeplant werden wie in Kapitel 2.2 beschrieben.

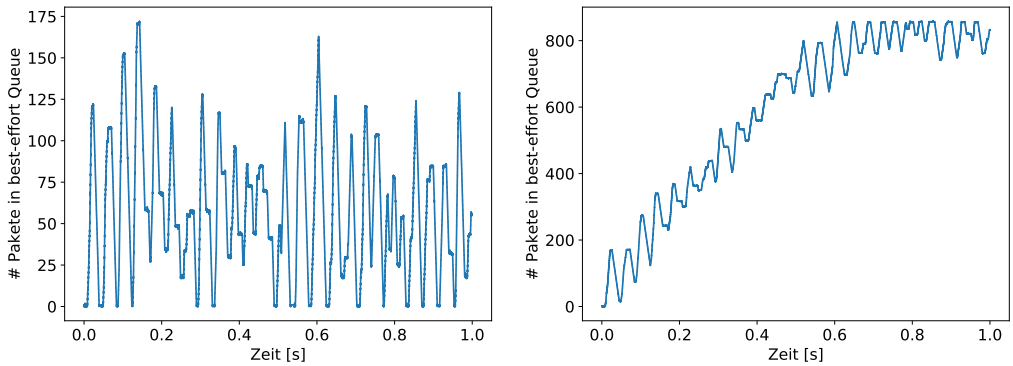
Die RTT zusammen mit der Zeit ε um die sich ein Datenpaket der Regelstrecke maximal verspäten darf entspricht der Zykluszeit des Schedules von 10ms. Nach 2377µs werden die Gates für den best-effort Traffic am Ausgangsport zum Regler für 128µs blockiert. Das gleiche Prinzip gilt für den Ausgangsport zur Regelstrecke. Hier wird nach 7372µs der best-effort Traffic blockiert. Auf diese Weise soll sichergestellt werden, dass die Kommunikation zwischen Regler und Regelstrecke niemals durch Crosstraffic beeinflusst wird.

Im Evaluationsszenario werden insgesamt 6 verschiedene Grundkonfigurationen getestet mit unterschiedlichen Priorisierungsmechanismen im Switch sowie unterschiedlicher Policy des Regelungssystems.

1. Hold Policy ohne Priorisierung von Paketen
2. Zero Policy ohne Priorisierung von Paketen
3. Hold Policy mit Prioritätsscheduling
4. Zero Policy mit Prioritätsscheduling
5. Hold Policy mit Prioritätsscheduling und TSN Schedule
6. Zero Policy mit Prioritätsscheduling und TSN Schedule

Congestion/Cross-Traffic	Keine	Wenig	Mittel	Hoch	Sehr hoch
Burst Größe	0	46	63	80	96
Mittlere Linkauslastung	0%	55.2%	75.6%	96%	115.2%

Tabelle 5.1.: Congestion Levels



(a) Hohes Congestion Level

(b) Sehr hohes Congestion Level

Abbildung 5.4.: Größe der best-effort Queue

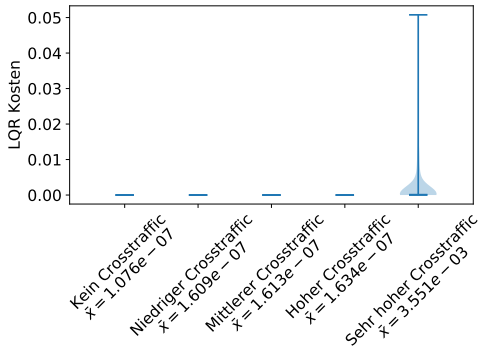
Die 6 genannten Grundkonfigurationen werden schließlich noch mit 5 verschiedene Congestion Levels untersucht. Je höher das Congestion Level, desto mehr Crosstraffic wird durch die *trafficSource* Hosts erzeugt. Die 4 Traffic Sources senden in normalverteilten Abständen mit $\mathcal{N}(40\text{ms}, 5\text{ms})$ Bursts bestehend aus 1.5kB großen Datenpaketen. Die Größe der Bursts in Abhängigkeit zum Congestion Level ist Tabelle 5.1 zu entnehmen.

Die Größe der Queues beträgt jeweils 10Mbit. Bei $10\text{Mbit}\,\text{s}^{-1}$ reicht das aus 10ms bzw. 833 Pakete mit MTU Größe zu puffern.

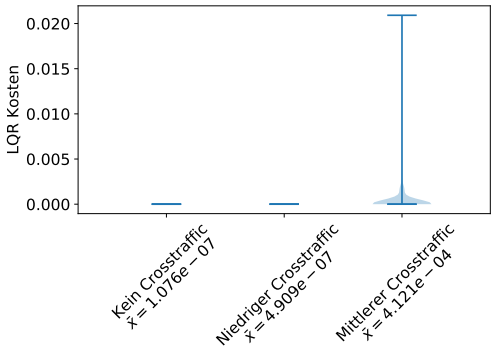
5.1.2. Ergebnisse der Simulationsdurchläufe

Für die Evaluation wurden somit insgesammt 30 verschiedene Testkonfigurationen ausgewertet. Bei hoher Congestion ist der Link vom Switch zur Regelstrecke zu 96% ausgelastet und bei sehr hoher Congestion zu 115.2%. Demzufolge ist zu erwarten, dass es bei sehr hoher Congestion zu einer Überlastung des Links kommt und Pakete verworfen werden müssen da, die Größe der best-effort Queue hier asymptotisch immer weiter ansteigt bis diese voll ist. Bei hoher Congestion sind lediglich temporäre Datenstaus zu erwarten, aber kein Paketverlust an der Queue durch Taildrop. Die Queue Größen der ersten Simulationssekunde sind in Schaubild 5.4a und 5.4b zu sehen.

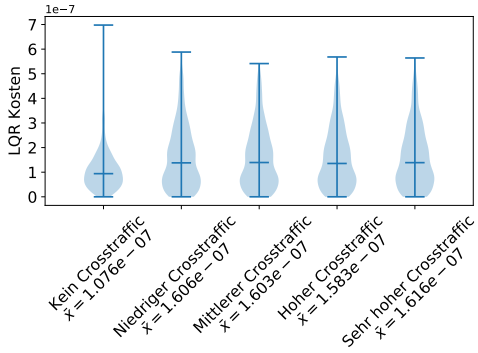
Im Hinblick auf die LQR Kosten kann aus Abbildung 5.5a und 5.5b in Verbindung mit 5.6a und 5.6b entnommen werden, dass ohne die Verwendung von mehreren Queues und Priorisierung der Pakete des Regelkreises, das System bei Überlastung des Links unabhängig von der Policy instabil wird. Zusätzlich kann man beobachten, dass bei Verwendung der Hold Policy das System innerhalb des Simulationszeitraums selbst bei 96% Linkauslastung stabil bleibt. Bei Verwendung der Zero Policy wird das System bereits bei einem mittleren Congestion Level instabil.



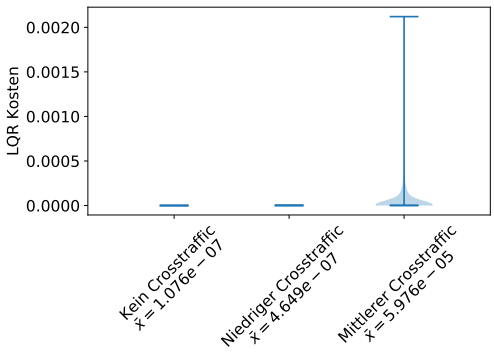
(a) Hold Policy ohne Prioritätsscheduling



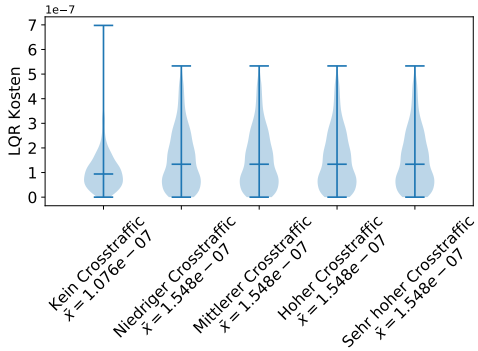
(b) Zero Policy ohne Prioritätsscheduling



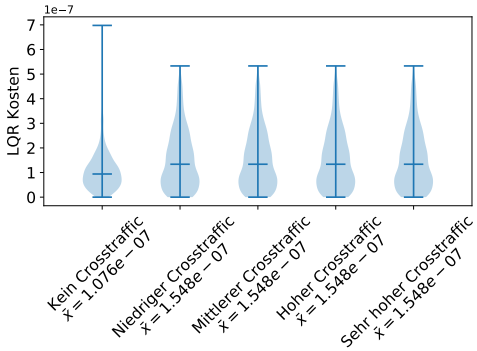
(c) Hold Policy mit Prioritätsscheduling



(d) Zero Policy mit Prioritätsscheduling

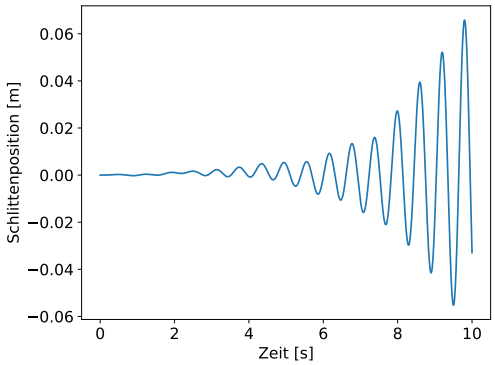


(e) Hold Policy mit Prioritätsscheduling und time-aware Shaper

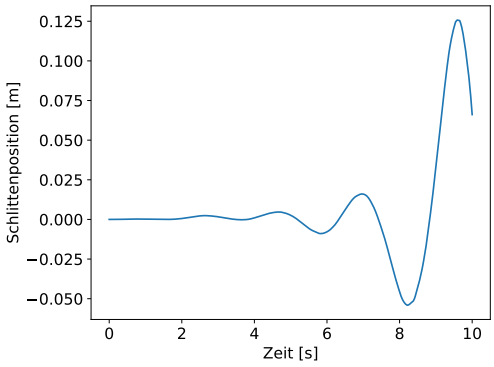


(f) Zero Policy mit Prioritätsscheduling und time-aware Shaper

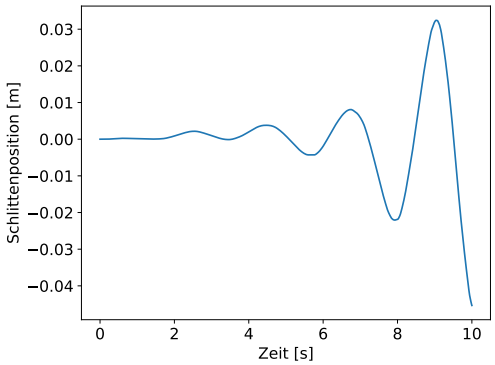
Abbildung 5.5.: LQR Kosten des inversen Pendels



(a) Hold Policy ohne Prioritätsscheduling bei hoher Congestion



(b) Zero Policy ohne Prioritätsscheduling bei mittlerer Congestion

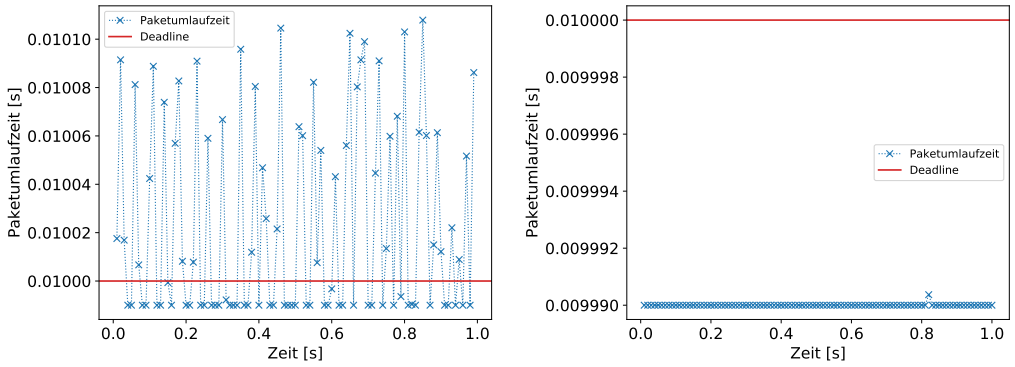


(c) Zero Policy mit Prioritätsscheduling bei mittlerer Congestion

Abbildung 5.6.: Schlittenposition des inversen Pendels

Bei Einsatz von Prioritätsscheduling mit unterschiedlich priorisierten Queues und einer erhöhten Priorisierung des Datenverkehrs des Regelungssystems ist eine Verbesserung der LQR Kosten zu beobachten. Aus Abbildung 5.5c kann man entnehmen, dass das Pendel bei Verwendung der Hold Policy innerhalb des Simulationszeitraums selbst bei sehr hohem Congestion Level stabil bleibt. Wird jedoch die Zero Policy verwendet, verbessern sich zwar die durchschnittlichen LQR Kosten, jedoch scheint das System dennoch aber mittlerem Congestion Level instabil zu werden wie anhand Abbildung 5.5d und 5.6c zu sehen ist.

Verwendet man den time-aware Shaper aus dem IEEE 802.1Qbv Standard in Kombination mit einem Schedule so ist die Stabilität des Regelungssystems auch bei sehr hoher Congestion gewährleistet wie in Abbildung 5.5e und 5.5f zu sehen ist. Der Unterschied in den hierbei zustande kommenden Latenzzeiten ist in Abbildung 5.7a und 5.7b zu erkennen. Während beim Einsatz eines Schedules jedes der gesendeten Steuerungspakete vor Ablauf der Deadline bei der Regelstrecke eintrifft, kommt es bei ausschließlicher Verwendung von Prioritätsscheduling ohne time-aware Shaper zu leichten Verspätungen bis zu ca 100µs.



(a) Prioritätsscheduling ohne time-aware Shaper bei wenig Congestion

(b) Prioritätsscheduling mit time-aware Shaper bei wenig Congestion

Abbildung 5.7.: Paketumlaufzeiten mit und ohne Verwendung des time-aware Shapers

5.1.3. Interpretation der Simulationsergebnisse

Wie zu erwarten ist an den Simulationsergebnisse zu erkennen, dass ohne Netzwerkscheduling bzw. Priorisierungsmechanismen die Stabilität des inversen Pendels ab einem bestimmten Congestion Level nicht mehr gewährleistet werden kann. Weiterhin zeigte sich, dass mit der Hold Policy bei Congestion niedrigere LQR Kosten erzielt werden und mit dieser selbst bei hoher Congestion ohne Netzwerk Scheduling das System innerhalb des Simulationszeitraum stabil bleibt. Es ist auch zu erkennen, dass Prioritätsscheduling ohne time-aware Shaping unter Umständen nicht dazu in der Lage ist die Stabilität bei Verwendung der Zero Policy zu gewährleisten.

Insgesamt zeigte sich, dass sich ein auf Julia basierende NCS erfolgreich mit der JLSM Bibliothek implementieren und in einem IEEE 802.1Qbv Netzwerk simulieren ließ. Die entwickelten Komponenten sollten somit auch in großflächigerer Simulation von CPS einsetzbar sein.

5.2. Performance Overhead der Julia Module

Der Performance Overhead der in Julia geschriebenen Simulationsmodule beeinflusst die Art der Anwendungsfälle für die die JLSM Bibliothek optimalerweiße verwendet wird und ob der hierbei entstehende Overhead eine Implementierung von NCS Komponenten mit Julia rechtfertigt. Anhand eines einfachen Testszenarios, dass sowohl in C++ als auch in Julia implementiert wird, wird der Umfang des Overheads evaluiert.

5.2.1. Testsetup

Als Testszenario zur Evaluation der Performance von Simulationsmodulen auf Grundlage von C++ und Julia verwenden wir eine Abwandlung des relativ bekannte TicToc Beispiel [Ope19]. Zwei *SimpleModules* sind hierbei über einen Channel miteinander verbunden und senden eine Nachricht

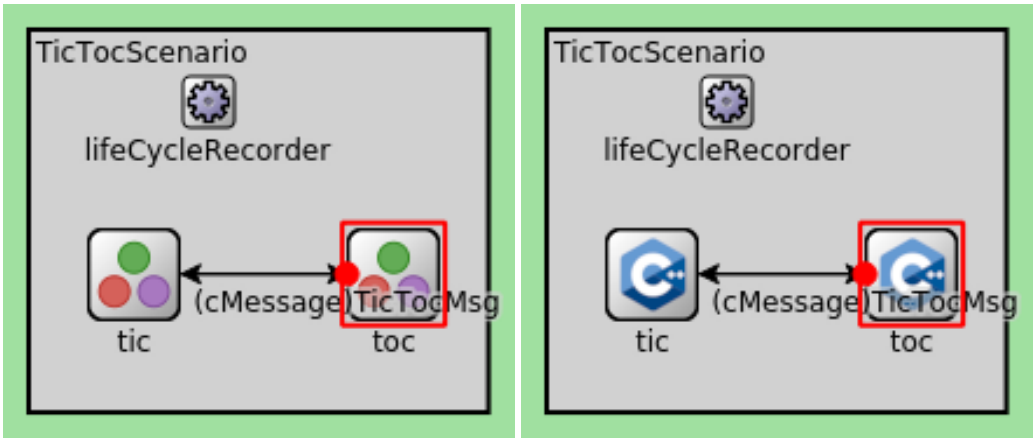


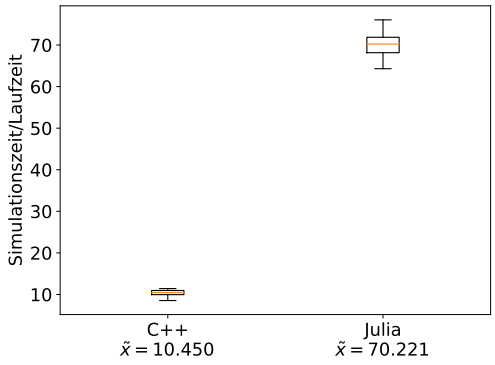
Abbildung 5.8.: Benchmark Simulation für den Vergleich von Julia und C++

hin- und her. Ein Bild des C++ und eines des Julia Szenarios sind in Abbildung 5.9 zu sehen. Es handelt sich beides mal um die selbe Simulation. In dem beide Module das gleiche NED Interface implementieren können dann jeweils C++ oder julia Implementierung eingesetzt werden.

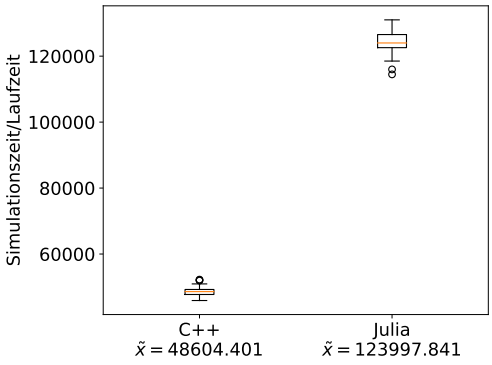
Im Testszenario ist außerdem die *lifecycleRecorder* Komponente zu sehen. Diese wird dazu verwendet die Simulationslaufzeit unabhängig von der Startup-Zeit der Simulation zu messen. Hierzu wird beim Aufruf des *initialize* Eventhandler und beim *finish* Eventhandler ein Timestamp der realen Zeit (nicht der Simulationszeit) gespeichert. Mit dem Programm *time* wird außerdem die Gesamtlaufzeit des Programms gemessen. Anhand der Differenz lässt sich somit auch die Startup-Zeit des Simulators bestimmen. Die Messung der Startup-Zeit sowie eine Messung der Ausführungszeit ohne Startup-Phase ist deswegen sinnvoll, weil das Julia Subsystem beim Start die Julia Skripte beim Laden präkompilieren muss und hierfür mehr Zeit benötigt als das bei Verwendung von reinen C++ Modulen der Fall wäre. Es ist deswegen sinnvoll die Startup Phase separat zu betrachten.

Es wird erwartet, dass das Dispatching der Eventhandler der *JuliaSimpleModule* Instanzen an die Julia Funktionen mit einem relativ großen Overhead behaftet ist. Deswegen wurden zwei Versuchsreihen durchgeführt. Bei der einen werden Eventhandler mit vernachlässigbarem CPU Overhead verwendet. D.h. wenn ein Modul eine Nachricht empfängt, sendet es diese sofort wieder zurück. In der anderen Versuchsreihe wird nach Empfang einer Nachricht eine CPU-lastige Berechnung durchgeführt bevor die Nachricht wieder zurückgesendete wird. In unserem Fall handelt es sich hierbei um die Faktorisierung einer normalverteilten Zufallszahl. Da sowohl das C++ Modul als auch das Julia Modul denselben Random Number Generator (RNG) mit gleichem Seed verwenden, werden beides mal exakt die selben Zahlen faktorisiert. Die Ergebnisse sind somit also durchaus vergleichbar.

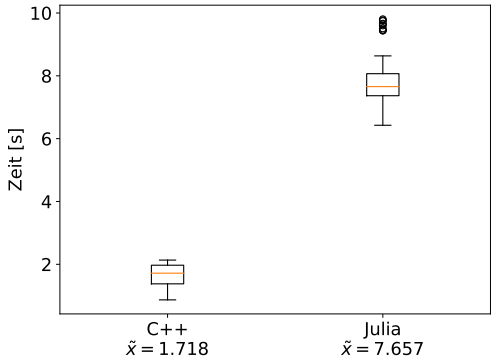
Getestet wurde das Szenario auf einer VM mit 24 Intel E56xx Cores mit jeweils 2.4GHz und insgesammt 4GB RAM. Insgesammt werden 100 Simulationsdurchläufe durchgeführt, wobei immer jeweils eine Prozessinstanz auf einem Core und maximal 10 Prozesse parallel ablaufen.



(a) Eventhandler mit niedriger CPU Last



(b) Eventhandler mit hoher CPU Last



(c) Startup-Zeit der Simulationsumgebung

Abbildung 5.9.: Benchmark von Julia und C++ Modulen

5.2.2. Performance Overhead

Nach Abbildung 5.9a liegt bei der Verwendung wenig CPU intensiver *handleMessage*-Eventhandlern der Median des Verhältnisses der realen Zeit zur Simulationszeit bei den C++ Modulen bei 10.450 und bei Julia Modulen bei 70.221. C++ Module sind hier also um einen Faktor von 6.719 schneller. Werden CPU intensivere Eventhandler verwendet liegt der Quotient nach Abbildung 5.9b bei C++ Modulen bei 48604.401 und bei Julia bei 123997.841. Die in C++ Module sind hier nur noch um einen Faktor von 2.551 schneller. Für die Startup-Zeit wurden Werte von 1.718s für C++ und 7.657s für Julia gemessen, was einem Faktor von 4.456 entspricht. Dies ist Abbildung 5.9c zu entnehmen.

5.2.3. Interpretation des Benchmark

Es war zu Erwarten, dass der Dispatching Aufwand an die Julia Funktionen relativ hoch ist, denn die *JuliaSimpleModule* Instanzen fragen für das Dispatching einen globalen Assoziativspeicher aus dem Julia Subsystem anhand dessen Namen ab und entnehmen aus diesem das zugehörige Julia Objekt. Das ist zwar im Normalfall in $O(1)$ möglich. Dennoch ist der Overhead relativ groß. Hier wäre eine Optimierung der Lookups sinnvoll. Trotzdem werden die Dispatching Kosten vermutlich immer relativ hoch bleiben.

Die um den Faktor 2.551 steigenden Rechenkosten bei CPU-lastigen Funktionen kommt vermutlich zu einem beachtlichen Teil durch den von Julia verwendeten GC zustande. Der Overhead scheint jedoch nicht so exorbitant zu sein, dass Julia zur Implementierung von Regelungssystemen nicht geeignet wäre. Insbesondere in Simulation bei denen der Großteil der Events an C++ Modulen anfällt und nur ein kleiner Teil an den Julia Modulen anfällt, kann der Performance Overhead vernachlässigbar sein. Dies war z.B. bei der Simulation des NCS bzw. inversen Pendels der Fall. Nur alle 5ms fällt ein Event am Regler oder der Regelstrecke an. Zwischenzeitlich fallen aber tausende Events an anderweitigen Modulen wie z.B. am Ethernet Switch an.

Der Startup-Overhead lässt sich bei der Verwendung von Julia Modulen nicht vermeiden. Allerdings kann dieser bei Ausführung von Simulationen über einen längeren Zeitraum vernachlässigbar sein. Weitere Performanceoptimierungen bei der Präkompilierung wären aber auch hier vorstellbar.

6. Zusammenfassung und Ausblick

In dieser Arbeit wurden die Grundsteine für die Simulation von Networked Control Systems mit Omnet++ und Julia gelegt. Es wurde eine Julia Erweiterung für Omnet++ entwickelt mit deren Hilfe Prototypen von NCS Anwendungen auf einfache Weise implementiert werden können. Weiterhin wurde anhand einer Proof of Concept Simulation eines inversen Pendels die Praxistauglichkeit der entwickelten Module bestätigt. Zu guter Letzt wurde noch anhand eines Performance Benchmark gezeigt, dass die in Julia implementierten Module im Moment nicht mit der Geschwindigkeit von C++ Modulen mithalten können.

Im Hinblick auf die JLSM Bibliothek ist geplant das Dispatching der Eventhandler zu optimieren, um eine bessere Performance im Vergleich zum Benchmark aus Kapitel 5.2 zu erreichen. Zusätzlich wäre es sinnvoll zu untersuchen ob sich die Performance weiter optimieren lässt durch verbesserte Unterstützung von Präkompilierung. Ein weiteres sinnvolles Feature wäre die automatische Codegenerierung um das Schreiben der Methodenstubs für die C++ Bindings zu automatisieren. Für die Zukunft ist geplant die Bibliothek aktiv weiterzuentwickeln und als Open Source Projekt zu veröffentlichen.

Das im Moment relativ niedrige Abstraktionsniveau der Implementierung von Regler und Regelstrecken könnte weiter angehoben werden, so dass neue Systemklassen noch einfacher und schneller implementiert werden können. Es wäre außerdem sinnvoll UDP als Kommunikationsprotokoll für Networked Control Systems zu verwenden anstatt den Payload direkt über Ethernet Frames zu versenden. Die Hosts könnten dann auch anhand einer IP Adresse identifiziert werden.

Literaturverzeichnis

[AlH12] A. Al-Hammouri. „A comprehensive co-simulation platform for cyber-physical systems“. In: *Computer Communications* 36 (Dez. 2012), S. 8–19. DOI: 10.1016/j.comcom.2012.01.003 (zitiert auf S. 31).

[Bas08] P. Bastian. *Grundlagen der Modellbildung und Simulation*. Juli 2008 (zitiert auf S. 19).

[BEKS14] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. „Julia: A Fresh Approach to Numerical Computing“. In: *CoRR* abs/1411.1607 (2014). arXiv: 1411.1607. URL: http://arxiv.org/abs/1411.1607 (zitiert auf S. 18, 30).

[CAD09] M. C. Herbordt, M. Ashfaquzzaman Khan, T. Dean. „Parallel Discrete Event Simulation of Molecular Dynamics Through Event-Based Decomposition“. In: Juli 2009, S. 129–136. DOI: 10.1109/ASAP.2009.39 (zitiert auf S. 20).

[CBDR17] B. Carabelli, R. Blind, F. Dürr, K. Rothermel. *State-dependent Priority Scheduling for Networked Control Systems*. Mai 2017. DOI: 10.23919/ACC.2017.7963084 (zitiert auf S. 17).

[Dan15] S. Danisch. *Cxx.jl Issue 62: Windows support*. Jan. 2015. URL: https://github.com/Keno/Cxx.jl/issues/62 (zitiert auf S. 37).

[DK14] F. Dürr, T. Kohler. *Comparing the Forwarding Latency of OpenFlow Hardware and Software Switches*. Stuttgart, Germany, Juli 2014 (zitiert auf S. 47).

[FHC+19] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, K. Rothermel. „NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++“. In: *Proceedings of the 2019 International Conference on Networked Systems (NetSys)* (März 2019) (zitiert auf S. 17, 27, 33).

[Fis18] K. Fischer. *Cxx.jl*. Nov. 2018. URL: https://github.com/Keno/Cxx.jl (zitiert auf S. 30, 36).

[INE18] INET Team, Hrsg. *INET Model Catalog*. INET. 2018. URL: https://inet.omnetpp.org/Protocols.html (zitiert auf S. 26).

[Jan19] B. Janssens. *CxxWrap.jl*. März 2019. URL: https://github.com/JuliaInterop/CxxWrap.jl (zitiert auf S. 30, 36).

[JRZ18] M. Jung, F. Rosenthal, M. Zitterbart. „Poster Abstract: CoCPN-Sim: An Integrated Simulation Environment for Cyber-Physical Systems“. Englisch. In: *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI), Orlando, FL, USA, 17-20 April 2018*. IEEE, Piscataway (NJ), 2018, S. 281–282. ISBN: 978-1-5386-6313-4. DOI: 10.1109/IoTDI.2018.00040 (zitiert auf S. 32).

[Jul18] Julialang, Hrsg. *Julia Manual: Calling C and Fortran Code*. Julialang. Nov. 2018. URL: https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/ (zitiert auf S. 36).

[Jul19] Julialang, Hrsg. *Julia Manual: Embedding Julia*. Julialang. Jan. 2019. URL: https://docs.julialang.org/en/v1/manual/embedding/index.html (zitiert auf S. 35, 38).

[LCD+19] S. Linsenmayer, B. W. Carabelli, F. Dürr, J. Falk, F. Allgöwer, K. Rothermel. *Integration of Communication Networks and Control Systems Using a Slotted Transmission Classification Model*. Englisch. Workshop-Beitrag. Las Vegas, NV, USA, Jan. 2019. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2019-01&engl=0 (zitiert auf S. 17, 42).

[MK13] J. Madsen, T. Kristensen. „Simulating Network Adapted Market Controller in a Smart Grid Scenario“. Master's Thesis. Aalborg University, Juni 2013 (zitiert auf S. 34, 35).

[Mol04] C. Moler. *The Origins of MATLAB*. 20104. URL: https://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html (zitiert auf S. 18).

[Näg15] D. Nägele. *A Demonstrator for Networked Control Systems*. Sep. 2015 (zitiert auf S. 29).

[New19] S. News. *January Prize Spotlight: Jeff Bezanson, Steven L. Brunton, Jack Dongarra, Stefan Karpinski, and Viral B. Shah*. Siam. Jan. 2019. URL: https://sinews.siam.org/Details-Page/january-prize-spotlight-jeff-bezanson-steven-l-brunton-jack-dongarra-stefan-karpinski-and-viral-b-shah (zitiert auf S. 30).

[NL18] S. Nsaibi, L. Leurs. „Abbildung von TDMA-basierten Industrial Ethernet Protokollen auf TSN am Beispiel von Sercos III“. In: *Kommunikation und Bildverarbeitung in der Automation*. Hrsg. von J. Jasperneite, V. Lohweg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, S. 122–135. ISBN: 978-3-662-55232-2 (zitiert auf S. 24).

[Ope19] OpenSim Ltd., Hrsg. *Learn OMNeT++ with TicToc*. OpenSim Ltd. 2019. URL: https://docs.omnetpp.org/tutorials/tictoc/ (zitiert auf S. 53, 63).

[PLB15] M. Papageorgiou, M. Leibold, M. Buss. *Optimierung - statische dynamische stochastische verfahren für die anwendung*. 2015. DOI: 10.1007/978-3-662-46936-1 (zitiert auf S. 28).

[RB18a] T. Roth, M. Burns. *A gateway to easily integrate simulation platforms for co-simulation of cyber-physical systems*. Apr. 2018. DOI: 10.1109/MSCPES.2018.8405394 (zitiert auf S. 34).

[RB18b] T. Roth, M. Burns. „A Gateway to Easily Integrate Simulation Platforms for Co-Simulation of Cyber-Physical Systems“. In: *Work-shop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)* (2018) (zitiert auf S. 32).

[RJZH19] F. Rosenthal, M. Jung, M. Zitterbart, U. Hanebeck. *CoCPN-Towards Flexible and Adaptive Cyber-Physical Systems Through Cooperation*. Jan. 2019. DOI: 10.1109/CCNC.2019.8651882 (zitiert auf S. 32).

[Rod94] C. Roduner. *Die Riccati-Gleichung*. Apr. 1994 (zitiert auf S. 28).

[Sch16] O. Schulz. *Cxx.jl Issue 305: Calling constructor of templated class via cxx-macro*. Nov. 2016. URL: https://github.com/Keno/Cxx.jl/issues/305 (zitiert auf S. 37).

[Tak11] J. Takeuchi. *Requirements for AutomotiveAVB System Profiles*. März 2011 (zitiert auf S. 24).

[Tan16] D. Tanzer. *6 Reasons Why You Should Avoid Singletons*. März 2016. URL: https://www.davidtanzer.net/david%27s%20blog/2016/03/14/6-reasons-why-you-should-avoid-singletons.html (zitiert auf S. 38).

[Var03] A. Varga. *Parallel Simulation Made Easy With OMNeT + +*. 2003 (zitiert auf S. 20).

[VH08] A. Varga, R. Hornig. *An overview of the OMNeT++ simulation environment*. Jan. 2008. DOI: 10.1145/1416222.1416290 (zitiert auf S. 25).

[Wei18] M. Weitbrecht. *Simulationsgestützte Analyse von Time-Sensitive Networking in konvergenten Netzwerken*. Aug. 2018 (zitiert auf S. 27).

[ZDFH16] Y. Zhang, Y. Dong, W. Feng, M. Huang. *A Co-Simulation Interface for Cyber-Physical Systems*. Aug. 2016. DOI: 10.1109/ICESS.2016.37 (zitiert auf S. 32–34).

Alle URLs wurden zuletzt am 17. 03. 2018 geprüft.

A. Omnet++ TicToc Beispiel mit Julia

Das TicToc Beispielszenario ist ein Minimalbeispiel für Omnet++. Vergleichbar mit einem klassischen ”Hello World”Programm, handelt es sich hierbei um das erste Beispiel innerhalb des offiziellen Omnet++ Tutorials [Ope19]. Die TicToc Simulation besteht aus zwei Knoten bzw. SimpleModules, die sich gegenseitig ein Paket zusenden. Anhand der Implementierung des für die Simulation benötigten SimpleModuls werden die Grundprinzipien des Eventhandlings in Omnet++ erläutert. Wir wollen dieses Beispiel hier analog zum offiziellen Omnet++ Tutorial zur Demonstration der JLSM Bibliothek nutzen. Hierfür werden die SimpleModules nicht mit C++ sondern mit Julia implementiert.

In Listing A.1 ist der Inhalt der NED Datei zu sehen in der die Topologie sowie der Aufbau des TicToc SimpleModules definiert sind. Der *sendMsgOnInit* Parameter des TicToc Moduls gibt an ob das Modul bei Initialisierung eine Nachricht senden soll. Weiterhin verfügt jedes TicToc Modul über ein bidirektionales Gate mit dem es mit anderen Modulen verbunden werden kann. Die drei anderen Parameter mit dem Präfix ”julia”werden benötigt um die entsprechende Julia Datei, sowie Julia Modul und Konstruktor Funktion zu laden. In der NED Datei ist außerdem spezifiziert, dass das TicToc Modul bei Empfang einer Nachricht ein Signal auslöst und das dieses *receiveMsg* Signal bei der Erfassung von Statistiken berücksichtigt wird. In diesem Fall wird einfach die Anzahl der empfangenen Nachrichten in Abhängigkeit der Zeit gezählt.

Im anderen Listing A.2 ist die Julia Implementierung zu sehen. Anfangs wird in der *initialize* Methode eine initiale Nachricht gesendet, falls der Wert des *sendMsgOnInit* Parameters wahr ist. In der *handleMessage* Funktion wird beim Empfangen einer Nachricht diese wieder auf dem gleichen Gate zurückgesendet. Hierbei wird die Nachricht mit einer zufälligen, normalverteilten Verzögerung gesendet. In der *initialize* sowie in der *finish* Methode wird noch das *receiveMsg* Signal ausgelöst mit dem Wert 0. Das dient dazu, dass die Statistik für die Anzahl empfangener Pakete einen Anfangs- und End-Wert besitzt damit ein sinnvolles Diagramm gezeichnet werden kann.

Listing A.1 NED Datei des TicToc Beispiels

```
simple TicToc extends JuliaSimpleModule
{
  parameters:
    @signal[receiveMsg](type=long);
    @statistic[receiveMsg](title="The number of received messages";record=vector(sum));
    bool sendMsgOnInit = default(false);
    juliaFile = "TicTocModule.jl";
    juliaModule = "TicToc";
    juliaConstructor = "createTicTocModule";
  gates:
    inout gate;
}

network TicTocScenario
{
  submodules:
    tic: TicToc {
      sendMsgOnInit = true;
    }
    toc: TicToc;
  connections:
    tic.gate <--> toc.gate;
}
```

Listing A.2 Julia Implementierung des TicToc SimpleModules

```
module TicToc
  using Main.Opp
  using Distributions

  mutable struct TicTocModule
    cSimpleModule
    receiveMsgSignal
  end

  function createTicTocModule(cSimpleModule)
    return TicTocModule(cSimpleModule, Opp.registerSignal("receiveMsg"))
  end

  function initialize(ticTocModule)
    sendMsgOnInitPar = Opp.par(ticTocModule.cSimpleModule, "sendMsgOnInit");
    sendMsgOnInit = Opp.boolValue(sendMsgOnInitPar);
    if sendMsgOnInit
      msg = Opp.newCMessage("TicTocMsg", 0)
      Opp.send(ticTocModule.cSimpleModule, msg, "gate\$o", -1)
    end
    Opp.emit(ticTocModule.cSimpleModule, ticTocModule.receiveMsgSignal, 0)
  end

  function handleMessage(ticTocModule, cMessage)
    Opp.emit(ticTocModule.cSimpleModule, ticTocModule.receiveMsgSignal, 1)
    delay = Opp.SimTime(rand(Normal(0.1, 0.02)))
    Opp.sendDelayed(ticTocModule.cSimpleModule, cMessage, delay, "gate\$o", -1)
  end

  function finish(ticTocModule)
    Opp.emit(ticTocModule.cSimpleModule, ticTocModule.receiveMsgSignal, 0)
  end
end
```

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift