Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Simulationsgestützte Analyse von Time-Sensitive Networking in konvergenten Netzwerken

Michel Weitbrecht

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer/in:	M. Sc. Jonathan Falk
Beginn am:	5. Januar 2018
Beendet am:	2. August 2018

Kurzfassung

Time-Sensitive Networking (TSN) umfasst mehrere neue IEEE-Standards, die Ethernet-Netzwerke um Echtzeitfähigkeiten erweitern. Bei TSN werden zeitkritische Datenströme durch reservierte Zeitbereiche auf den Ethernet-Verbindungen vor Störungen durch andere Übertragungen geschützt. TSN ermöglicht damit die Koexistenz von herkömmlichem Datenverkehr und zeitkritschen Datenströmen in einem gemeinsamen, sog. *konvergenten* Netzwerk, ohne die Echtzeitgarantien des zeitkritschen Verkehrs zu verletzen. Diese Arbeit gibt zu Beginn einen Überblick über die TSN-Standards für Scheduling und Frame-Preemption sowie den Netzwerksimulator OMNeT++. Frame-Preemption ist eine Technologie um den Sendevorgang eines Frames abzubrechen und später fortzusetzen. Im darauffolgenden Kapitel wird die Implementierung von Frame-Preemption für ein existierendes Simulationsframework erläutert und eine Struktur zur Automatisierung von TSN-Simulationen vorgestellt. Schließlich werden automatisierte Simulationen verwendet, um Auswirkungen von TSN auf herkömmlichen Datenverkehr zu analysieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Auflistungen

1 Einleitung

Ethernet ist eine weit verbreitete Technologie für Netzwerkverbindungen und hat sich über Jahrzehnte als Standard-Lösung auf diesem Gebiet etabliert. Allerdings eignet sich Ethernet in seiner bisherigen Form nicht für die Übertragung von Echtzeitdaten. Das sind Datenübertragungen in einem Netzwerk, die Echtzeitanforderungen unterliegen, also in einer bestimmten Zeitspanne am Ziel angekommen sein müssen. Regelungssysteme, Fahrzeugnetzwerke und Industrieanlagen sind auf solche Echtzeitgarantien angewiesen, die Ethernet aber nicht bieten kann. Auf diesen Gebieten kommen meist proprietäre Lösungen zum Einsatz, die zwar auf Ethernet aufbauen, aber untereinander oder gar zu Ethernet selbst inkompatibel sind. Die IEEE, die Ethernet standardisiert, hat sich diesem Problem angenommen und erweitert Ethernet um Echtzeitfähigkeit. Die Erweiterungen für verschiedene Standards auf diesem Gebiet werden unter dem Begriff *Time-Sensitive Networking* (TSN) zusammengefasst.

Bei TSN werden Datenverkehre in Verkehrsklassen kategorisiert und können verschieden behandelt und priorisiert werden. Der wohl wichtigste Aspekt hierbei ist *Scheduling*, bei dem Zugriffe auf das Netzwerkmedium anhand eines *Schedules* (Zeitplan) kontrolliert werden. So kann zum Beispiel ein Zeitbereich festgelegt werden, in dem ausschließlich zeitkritische Daten übertragen werden dürfen. Mit dieser Reservierung der Verbindung können Verzögerungen und Störungen durch andere Daten verringert werden.

TSN standardisiert diese und weitere Funktionalitäten für zeitkritische Übertragungen und ermöglicht damit die Interoperabilität zwischen den Implementierungen verschiedener Hersteller. Gleichzeitig bleibt der Standard kompatibel zu Ethernet und kann damit existierende Technologien integrieren und Ressourcen wie Switche und Verbindungen teilen. Die geteilte Nutzung der Netzwerkverbindung, in der zeitkritischer und regulärer Datenverkehr koexistieren, wird als konvergentes Netzwerk bezeichnet. TSN beschränkt zwar die Ende-zu-Ende-Latenz für Echtzeitdaten, allerdings werden für reguläre Verkehre keine Aussagen zu der Service-Qualität getroffen.

Die Auswirkungen von TSN auf regulären, sog. *Best-Effort-Traffic* werden in dieser Arbeit anhand des Netzwerksimulators OMNeT++ untersucht. Netzwerksimulation stellt die Bedingungen von Netzwerkgeräten und Protokollen in einer Simulation nach und eignet sich, um Netzwerktechnologien zu untersuchen, für die keine Geräte zur Verfügung stehen. Aus einer früheren studentischen Arbeit existiert bereits ein TSN-Framework für OMNeT++, das im Rahmen dieser Arbeit um Frame-Preemption erweitert wird. Frame-Preemption ist eine Technologie um den Sendevorgang eines Frames abzubrechen und später fortzusetzen. Damit kann zeitkritischer Verkehr besser vor Verzögerungen beschützt und die verfügbare Bandbreite für regulären Verkehr erhöht werden.

Die Schedules für TSN werden vom Standard nicht vorgegeben, sondern sind Teil der Konfiguration, die ein Netzwerkadministrator vornimmt. In dieser Arbeit werden drei Verfahren für die Berechnung der TSN-Schedules betrachtet und die so erzeugten Schedules automatisiert in den TSN-Simulator integriert.

Kapitel 2 erläutert zu Beginn die Grundlagen von Netzwerksimulation und führt den verwendeten Netzwerksimulator sowie ein bekanntes Framework für Netzwerkprotokolle ein. Danach wird auf technische Details von TSN-Scheduling und Frame-Preemption, sowie das existierende TSN-Framework eingegangen und die Vorgehensweise der Scheduling-Verfahren beschrieben. In Kapitel 3 wird die Implementierung von Frame-Preemption anhand des Standards ausgeführt sowie die Architektur und einzelne Skripte zur Automatisierung und Analyse von TSN-Simulationen vorgestellt. In dem Kapitel wird außerdem darauf eingegangen, wie die TSN-Scheduler für das Projekt verwendet werden können. Im letzten Kapitel 4 wird schließlich das iterative Vorgehen für die Analyse der automatisierten Simulationen beschrieben und die Ergebnisse werden vorgestellt. Zuletzt wird die Arbeit in Kapitel 5 zusammengefasst.

2 Grundlagen

In diesem Kapitel wird zuerst das Konzept der ereignisorientierten Netzwerksimulation erläutert und ein Netzwerksimulator vorgestellt, der nach diesem Prinzip arbeitet. Im Anschluss wird Time-Sensitive Networking vorgestellt und zwei Konzepte daraus im Detail erläutert. Darauf folgt die Vorstellung dreier Forschungsarbeiten, die jeweils ein Scheduling-Verfahren für Time-Sensitive Networking entwickelt haben. Zuletzt wird ein Projekt vorgestellt, das die Simulation von Time-Sensitive Networking ermöglicht und im Rahmen dieser Arbeit weiterentwickelt wurde.

2.1 Ereignisorientierte Netzwerksimulation

2.1.1 Diskrete Ereignissimulation

Bei *diskreter Ereignissimulation* wird ein System simuliert, dessen aktueller Zustand sich mit jedem Ereignis ändert [Var18b, Kapitel 4.1.1]. Der neue Zustand hängt vom geschehenen Ereignis und dem vorherigen Zustand ab, weshalb die Simulation nur als eine Kette von zueinander abhängigen Ereignissen ausgeführt werden kann. Die Ereignisse geschehen zu einem bestimmten, *diskreten* Zeitpunkt innerhalb der Simulationszeit, jedoch ist diese unabhängig von der realen Zeit und wird in Sprüngen (mit jedem Ereignis) verändert.

2.1.2 Netzwerksimulation

Netzwerksimulationen werden verwendet, um die Interaktion zwischen Netzwerkkomponenten und -protokollen zu simulieren und zu testen. Durch die inkrementelle Veränderung der Simulationszeit kann reales Zeit-, Bandbreiten- und Performanzverhalten in Netzwerken nachgestellt werden. Netzwerksimulationen eignen sich insbesondere, um das Verhalten und die Eigenschaften neuer Netzwerkstandards zu erforschen, für die noch keine Hardwareimplementierungen existieren.

2.1.3 OMNeT++

Objective Modular Network Testbed in C++ (*OMNeT++*) ist eine Bibliothek für Netzwerksimulationen nach dem Prinzip der ereignisorientierten Simulation [Var17]. OMNeT++ umfasst neben der Simulationsbibliothek Kommandozeilenwerkzeugen zur Verwaltung und Unterstützung von Simulationen sowie einer Entwicklungsumgebung, die auf Eclipse basiert. Die in Abbildung 2.1 abgebildete Simulationsansicht der Entwicklungsumgebung ermöglicht die Betrachtung von Parametern, Modulen und Statistiken sowie eine graphische Darstellung von Objekten und Methodenaufrufen. Die Bibliothek stellt die Ausführungsumgebung für Simulationen, aber inhaltlich nur Grundbausteine wie Nachrichten, Kanäle, Komponenten zur Verfügung. Die Implementierung

von Netzwerkkomponenten kann aus weiteren Bibliotheken importiert oder in C++ oder anderen Programmiersprachen entwickelt werden. OMNeT++ ist für akademische Zwecke kostenfrei zu nutzen, kommerziell wird das Projekt unter dem Namen *OMNEST* angeboten. OMNeT++ nutzt „NED“ als domänenspezifische Sprache, in der Netzwerke, Module, Kanäle, Modul-Interfaces und Kanal-Interfaces verfasst werden.

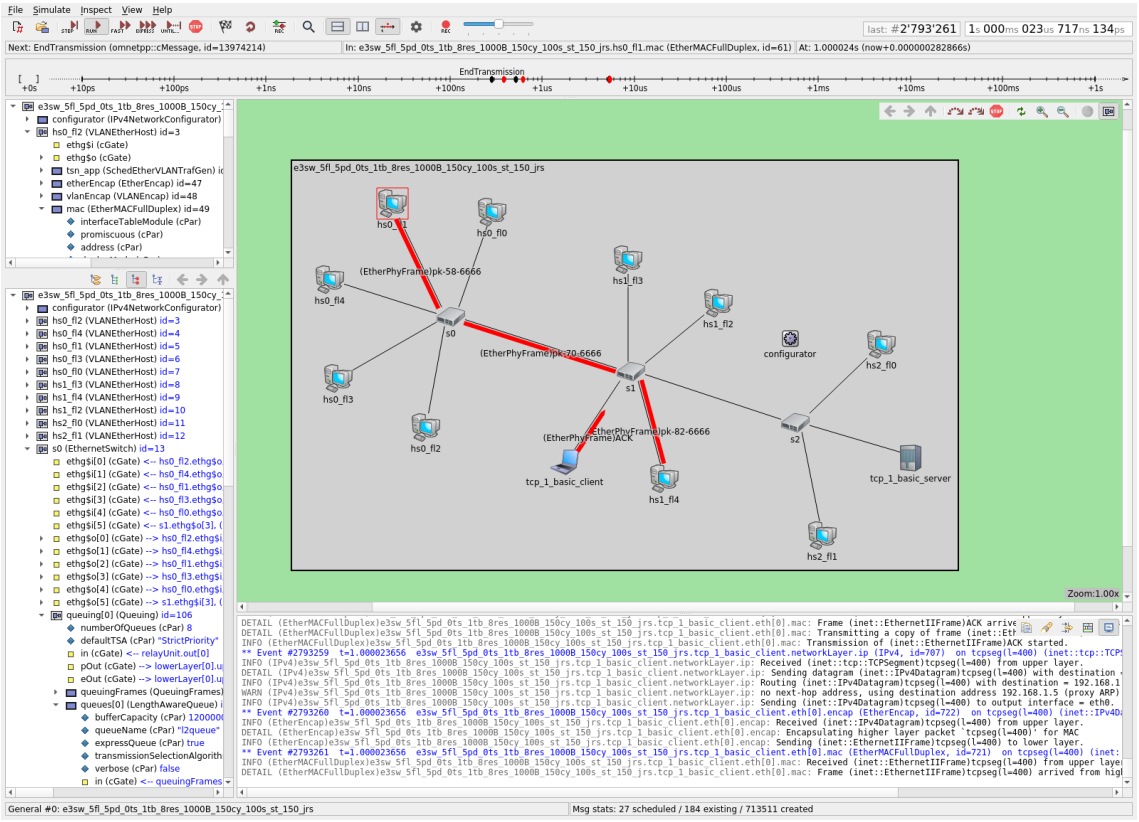


Abbildung 2.1: Eine laufende Netzwerksimulation im OMNeT++-Simulationsfenster

2.1.3.1 Module

Module sind handelnde Komponenten in OmNeT++, die als NED-Datei definiert sind und eine Funktion wie beispielsweise TCP-Kommunikation umsetzen. Sie können Funktionen von anderen Modulen erben und definieren Parameter und Gates (Schnittstellen) für Verbindungen zu anderen Modulen. Einfache („simple“) Module sind die aktiven Komponenten von Simulationen [Var18b, Kapitel 3.3], da sie durch die zugehörige C++-Klasse programmierbar sind. Zusammengesetzte („compound“) Module können hingegen nur existierende Module mit Kanälen zusammenfügen und Parameter der verwendeten Module setzen. Auflistung 2.1 zeigt die NED-Datei eines einfachen Moduls mit zwei Gates und sechs Parametern, teilweise mit vordefinierten Werten.

```
package nesting.ieee8021q.queue.gating;

simple TransmissionGate
{
    parameters:
        @display("i=block/source");
        @class(nesting::TransmissionGate);
        string gateControllerModule;
        string transmissionSelectionModule;
        string transmissionSelectionAlgorithmModule;
        string clockModule = default("^.^.clock");
        bool lengthAwareSchedulingEnabled = default(true);
        bool verbose = default(false);
    gates:
        input in;
        output out;
}
```

Auflistung 2.1: NED-Datei des einfachen Moduls `TransmissionGate.ned` aus dem NeSTiNg-Projekt [CDF+18]

2.1.3.2 Simulationen

Die Netzwerk-Topologie einer Simulation wird ebenfalls in einer NED Datei formuliert, allerdings wird die Konfiguration üblicherweise in einer INI Datei vorgenommen [Var18b, Kapitel 2.3.1]. Darin können allgemeine Einstellungen zur Simulation gesetzt werden, wie zum Beispiel die Laufzeit und das Verhalten im Fehlerfall. Weiterhin können die Parameter der verwendeten Module verändert werden, verschieden benannte Konfigurationen erstellt und mehrere Durchläufe definiert werden.

Simulationsergebnisse Ergebnisse von OMNeT++-Simulationen werden als Statistiken aufgezeichnet und können daraufhin maschinenlesbar analysiert werden [Var18b, Kapitel 12]. Jedes OMNeT++-Modul kann aus vorhandenen C++-Variablen Statistiken generieren, die durch OMNeT++ dann auf verschiedene Arten aufgezeichnet werden. Aus den Werten können Zeitreihen, Histogramme oder aggregierte Werte mit gängigen Statistik-Funktionen wie Summe, Durchschnitt, etc. aggregiert und die Ergebnisse aufgezeichnet werden [Var18b, Kapitel 4.15.2]. OMNeT++ enthält das *scavetool*, mit dem die Ergebnis-Dateien nach Typ, Statistik, Modul etc. gefiltert und anschließend exportiert werden können.

2.1.4 INET

INET ist ein Framework für OMNeT++ mit Modellen, Protokollen und Anwendungen für gängige Netzwerkszenarien [Var16]. Das Framework ist in C++ geschrieben und wird als freie Software [Var18a] entwickelt. INET implementiert OMNeT++-Module für gängige Protokolle der unteren vier Layern des ISO/OSI-Referenzmodells, sowie einige darauf aufbauende Anwendungen. Die

Kompatibilität, Flexibilität und Erweiterbarkeit für zukünftige Standards sind beim Design neuer Module wichtig [Var18c]. Im weiteren Verlauf werden werden OMNeT++-Module beschrieben, die auf INET-Modulen basieren.

2.2 Time-Sensitive Networking

Time-Sensitive Networking (*TSN*) bezeichnet eine Gruppe von IEEE Netzwerkstandards, die von der IEEE TSN Task Group erarbeitet werden [Ins18]. TSN verfolgt das Ziel, zeitkritischen Verkehr verlässlich und mit Garantien in Ethernet-Netzwerken zu übertragen. Größtenteils erweitern die TSN-Standards den VLAN-Bridging-Standard IEEE 802.1Q [Ins14] und befassen sich mit den drei nachfolgenden Themenbereichen:

Scheduling und Traffic Shaping bezeichnet die zeitliche Planung und Aufteilung von zeitkritischem Verkehr im gesamten Netzwerk

Pfadverwaltung und Fehlertoleranz (IEEE 802.1CB) umfasst redundante Pfade für zeitkritischen Verkehr, Ausfallsicherheit und die zentrale Verwaltung von TSN-Netzwerken

Zeitsynchronisation (IEEE 802.1AS) über das Netzwerk durch angepasste *Precision Time Protocol* (PTP) Profile

Die Übertragung zeitkritischen Datenverkehrs wird in dieser Arbeit als sog. *TSN-Flow*, ein zyklischer, gerichteter Strom von Daten, modelliert. Für diesen Datenverkehr gelten strenge Anforderungen bezüglich zeitkritischer Metriken, welche in TSN-Netzwerken im Gegensatz zu herkömmlichen Ethernet-Netzwerken garantiert werden können.

Folgende zeitliche Messgrößen werden für Flows durch TSN garantiert bzw. durch einen Worst-Case-Wert beschränkt:

Ende-zu-Ende Latenz die maximale Dauer der Übertragung vom Start- bis zum Zielhost

Jitter die Reihenfolge von Frames innerhalb eines Flows

Aktualität die pünktliche und regelmäßige Übertragung der Flows

Die Garantien gelten ausschließlich für die vor Laufzeit bekannten und modellierten Flows, den sog. *TSN-Traffic*.

2.2.1 Best-Effort-Traffic

Als *Best-Effort-Traffic* wird der nicht-zeitkritische Datenverkehr bezeichnet, der so schnell wie möglich, jedoch ohne Garantien mit variierenden Verzögerungen und ggf. mit Paketverlusten verschickt wird. Unter Best-Effort-Traffic versteht man den herkömmlichen, nicht zeitkritischen und nicht priorisierten Datenverkehr.

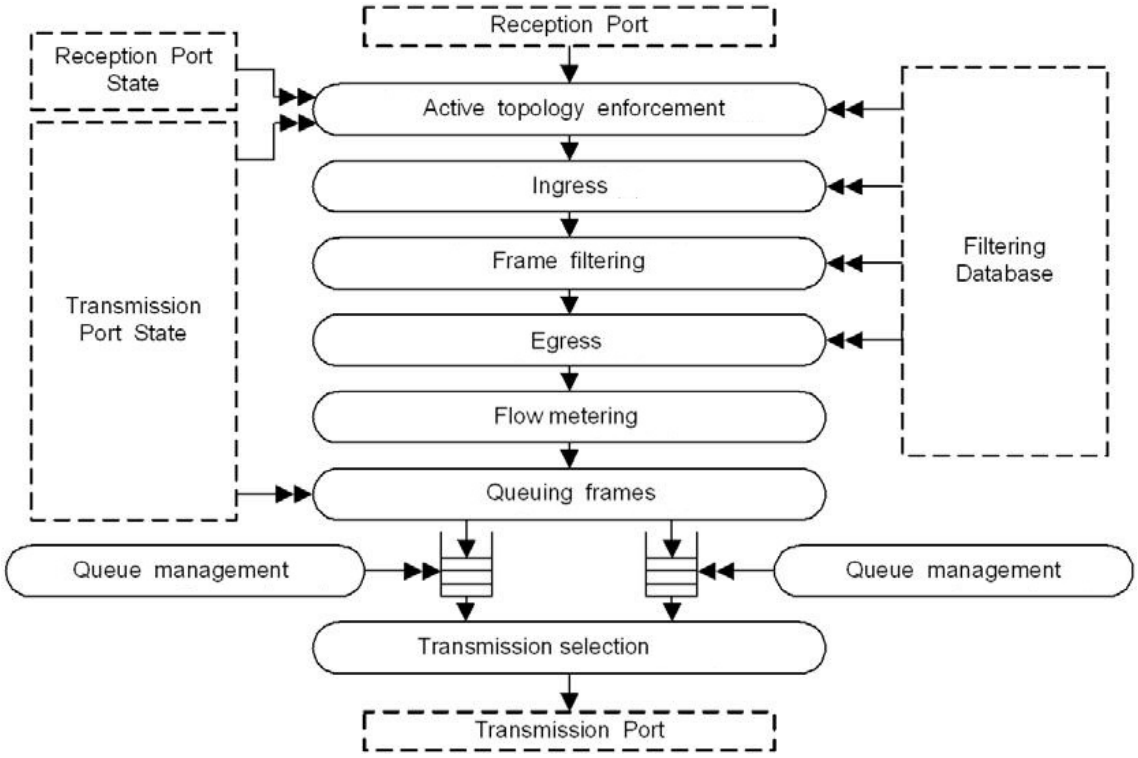


Abbildung 2.2: Architektur des Weiterleitungsprozesses in einem VLAN-fähigen Switch [Ins14, Abbildung 8-11]

2.2.2 Verkehrsklassen

TSN unterscheidet bei der Weiterleitung innerhalb eines Switches, dem *Bridging*, verschiedene Verkehrsklassen und erlaubt somit die Koexistenz von TSN-Traffic und Best-Effort-Traffic im selben Netzwerk. Jeder Ethernet Frame gehört einer Verkehrsklasse an und wird je nach Klasse verschieden behandelt und priorisiert. Abbildung 2.2 zeigt den Weiterleitungsprozess, den Frames von der Ankunft an einem Eingangsport oben, bis nach unten, zur Weiterleitung an einen Ausgangsport durchlaufen. Im *Queuing Frames* Abschnitt vor jedem Switch-Ausgangsport werden dabei ausgehende Frames anhand ihrer Verkehrsklasse in verschiedene Queues einsortiert. Für jede Verkehrsklasse gibt es eine Queue, je nach Implementierung sind zwischen einer und acht Verkehrsklassen vorhanden. In welcher Reihenfolge dann Frames zum Versand aus den Queues entnommen werden, hängt vom jeweiligen *Transmission-Selection Algorithmus* (siehe Unterabschnitt 2.2.4) und dem konfigurierten Schedule ab (siehe Unterabschnitt 2.2.3).

2.2.2.1 Priority Code Point

Die Einteilung in die Queues oder Zuordnung zu einer bestimmten Verkehrsklasse geschieht anhand des *Priority Code Point*-Werts (PCP). Dieser ist ein 3-Bit Datenfeld im VLAN-Tag des Frames in Abbildung 2.3, mit dem sich acht verschiedene Werte abbilden lassen. Da die Anzahl der verfügbaren Verkehrsklassen eines Switches variieren kann, gibt es im IEEE 802.1Q Standard [Ins14] eine Abbildungsmatrix, mittels der von einem PCP-Wert auf die korrekte Queue geschlossen

wird. Abhängig von den vorhandenen Verkehrsklassen kann die Priorität von Flows verschieden granular gewählt werden. Frames ohne VLAN-Tag werden in eine konfigurierbare Queue einsortiert, standardmäßig ist das die am niedrigsten priorisierte Queue [Ins14; Ins16a, Abschnitt 6.9.4].

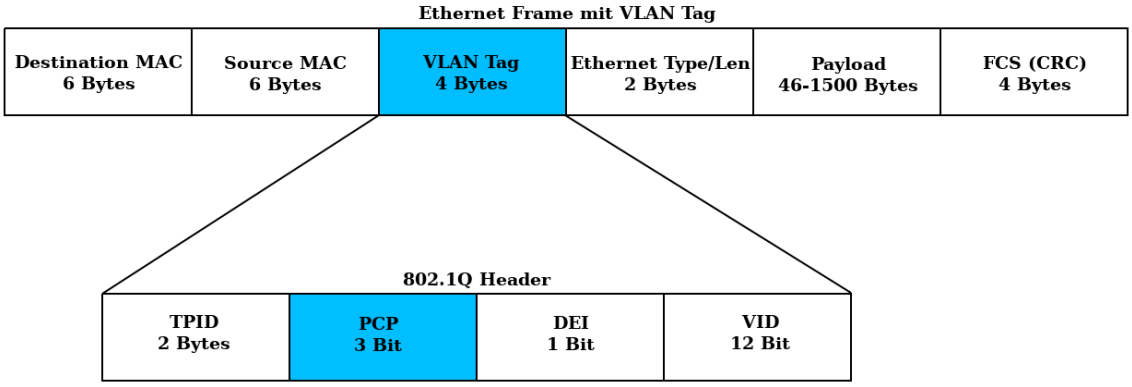


Abbildung 2.3: PCP-Wert im VLAN-Tag eines Ethernet Frames [Ins14; Ins16a]

2.2.3 Scheduling

Das Queuing-Netzwerk in Abbildung 2.4 umfasst eine Reihe von Komponenten, die vor den Ausgangsports eines TSN-Switches platziert sind und die Grundstruktur für Scheduling und Traffic Shaping bilden. Hinter jeder Queue befindet sich ein Gate (*Transmission Gate* in Abbildung 2.4), welches im Zustand „geöffnet“ oder „geschlossen“ sein kann. Es wird abhängig des Gatezustands beurteilt, ob Frames aus dieser Queue entnommen werden, und zur MAC Schnittstelle und damit auf das darunterliegende Medium übertragen werden können. Diese Gates werden zeitgesteuert geöffnet und geschlossen, mittels der sog. *Schedules*, siehe „Gate control list“ in Abbildung 2.4. Der *Schedule* eines Switches bzw. Ports ist ein Zeitplan, der periodisch abgearbeitet wird und die Zielzustände der Gates enthält. Über diesen Scheduling Mechanismus kann an TSN-Switchen kontrolliert werden, welche Verkehrsklasse zu welchem Zeitpunkt Frames senden darf. So kann beispielsweise ein Zeitbereich konfiguriert werden, in dem ausschließlich das Gate derjenigen Queue geöffnet ist, in der sich TSN-Traffic ansammelt. Pro Zeitpunkt oder Zeitspanne wird im Schedule definiert, welche Menge an Gates geöffnet ist, wodurch es möglich ist, dass mehreren Verkehrsklassen zur gleichen Zeit der Zugang zum Medium ermöglicht wird.

2.2.4 Traffic Shaping

Traffic Shaping wird die weitere Formung des Verkehrs durch bestimmte Algorithmen genannt. Diese sog. *Transmission-Selection Algorithmen* sind zwischen den Queues und den Gates in Abbildung 2.4 platziert und entscheiden ebenfalls ob Frames passieren dürfen. Es sind zwei Algorithmen vorgeschlagen [Ins14], der *Credit-Based Shaper* (Unterunterabschnitt 2.2.4.1) und *Enhanced Transmission-Selection*, bei der jeder Queue ein Anteil der Bandbreite zugewiesen wird. Weiterhin gibt es als transparentes Standardverfahren den sog. *Strict Priority Shaper*, welcher keine Kontrolle über den Datenverkehr ausübt und somit lediglich die Priorität der Gates zur Geltung kommen lässt. Der Standard hält die Konfiguration für weitere Implementierungen, auch durch Hersteller, offen.

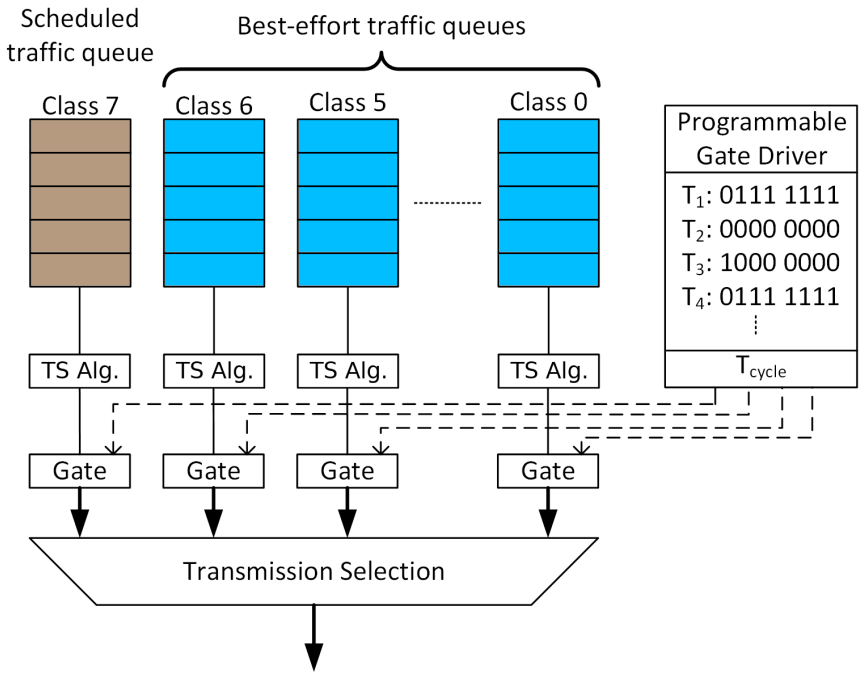


Abbildung 2.4: Das Queuing-Netzwerk am Ausgangsport eines TSN-Switches [DN16; TFB+13]

2.2.4.1 Credit-Based Shaper

Der *Credit-Based Shaper* verwaltet für die dahinterliegende Queue einen *Kreditwert*, anhand dessen Frames durchgelassen werden. Ein positiver Kredit oder ein Kredit gleich Null erlaubt, dass Frames weitergeleitet werden. Der Kreditwert einer Queue verringert sich, wenn ein Frame von ihr versandt wird und erhöht sich, sobald in der Queue Frames auf die Übertragung warten, da zeitgleich ein Frame aus einer anderen Queue verschickt wird. Hierbei kann über verschiedene Sammel- bzw. Verbrauchsraten des Kredits eine weitere Priorisierung erfolgen. Der Credit-Based Shaper ist somit ein Instrument zur Gewährleistung einer Bandbreitenverteilung zwischen Queues.

2.2.5 Transmission-Selection

Nach den Queues, Shapern und Gates entscheidet die *Transmission-Selection* (siehe Abbildung 2.2, Abbildung 2.4), ob und welcher Frame versendet wird. Dies geschieht in Reihe der Priorität und nur falls die darüberliegenden Komponenten dies – durch offene Gates und aktive Shaper – ermöglichen. d. h. Frames aus einer Queue werden genau dann übertragen, wenn

- zeitgleich kein anderer Frame übertragen wird
- der entsprechende Shaper es gestattet
- das entsprechende Gate geöffnet ist
- alle höher priorisierten Queues entweder keine Frames enthalten, ihr Gate geschlossen ist oder der entsprechende Shaper die Übertragung verweigert

2.2.6 Schutzband

Nachdem ein Frame das Queuing-Netzwerk verlässt, kann nicht sofort ein weiterer Frame in Richtung der MAC-Komponente weitergegeben werden, da die Übertragung des Frames eine bestimmte Zeitspanne in Anspruch nimmt. Diese Zeitspanne besteht zum einen aus dem Transmission Delay, das für die Modulierung des Frames auf das Medium benötigt wird (siehe Unterabschnitt 2.3.1), zum anderen können je nach Medium Pausen zwischen zwei Frames, wie das *Interframe Gap* [Ins16a], definiert sein. In dieser Zeit blockiert die MAC-Schnittstelle den Pfad zum Medium, indem sie keine neuen Frames mehr anfordert. Die Transmission-Selection prüft die notwendigen Bedingungen zur Weitergabe von Frames, jedoch nur zum Startzeitpunkt eines Frames und kann nicht berechnen, wie lange diese Bedingungen gelten werden. Dadurch kann der Fall auftreten, dass kurz vor Ende des

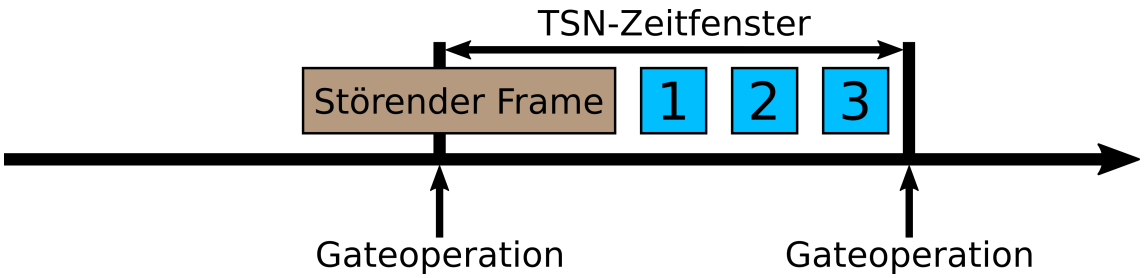


Abbildung 2.5: Spätes Eintreffen von Best-Effort-Traffic ohne Schutzband [Pit16]

zugeteilten Bereichs im Schedule ein Frame losgesendet wird, dessen Übertragung erst während des nächsten Zeitbereichs endet. Abbildung 2.5 zeigt diesen Fall, bei dem ein Best-Effort Frame am Ende seines Zeitbereichs verschickt wird und so in das für TSN-Traffic reservierte Zeitfenster hineinragt. Angesammelte oder ankommende TSN-Frames können nicht sofort versendet werden, sobald sich ihr Gate öffnet, sondern erst wenn der aktuell übertragene Frame der vorherig eingeteilten Verkehrsklasse vollständig versendet wurde. Im Worst-Case hat der hereinragende Frame die maximal mögliche Größe (*Maximum Transmission Unit, MTU*) und verzögert den eigentlich geplanten Datenverkehr um das entsprechend große Transmission Delay. Zusätzlich gibt es MAC-spezifische Verzögerungen wie den Interframe Gap, der die Pause, die zwischen der Modulierung zweier Frames eingehalten werden muss, definiert. Wenn die Zeitfenster für TSN-Verkehr knapp geplant sind, kann es sein, dass durch das „Hereinragen“ nicht alle im Zeitslot geplanten Frames übertragen werden können. Das Beispiel in Abbildung 2.5 zeigt diesen Fall, bei dem nur drei TSN-Frames übertragen werden können, obwohl ein Zeitbereich für die Übertragung von fünf Frames reserviert wurde. Die Verzögerung kann sich über den gesamten Pfad des Flows propagieren und verstärken und den Flow sogar um ganze Schedule-Zyklen verzögern, da die Frames am nächsten Hop so schon gar nicht erst pünktlich ankommen. Ebenfalls können Frames so aufgestaut werden, dass Queues über ihre maximale Größe befüllt werden und Frames weggeworfen werden müssen. Es ist als realistisch anzusehen, dass für TSN-Traffic relativ kurze Queues benutzt werden, wenn der Traffic darin ohne Verzögerung weitergeleitet wird.

Um diese Szenarien vorzubeugen und bestmögliche Zeitgarantien geben zu können, kommt ein implizites oder explizites Schutzband (engl. *Guard Band*) zum Einsatz. Das Schutzband wird vor einem Zeitfenster mit zeitkritischem Verkehr platziert, um dieses vor hereinragenden Frames zu schützen, wie in Abbildung 2.6 in blau dargestellt. Innerhalb des Schutzbands dürfen keine neuen Frames mehr übertragen werden, sondern nur noch begonnene Übertragungen zu Ende geführt werden.

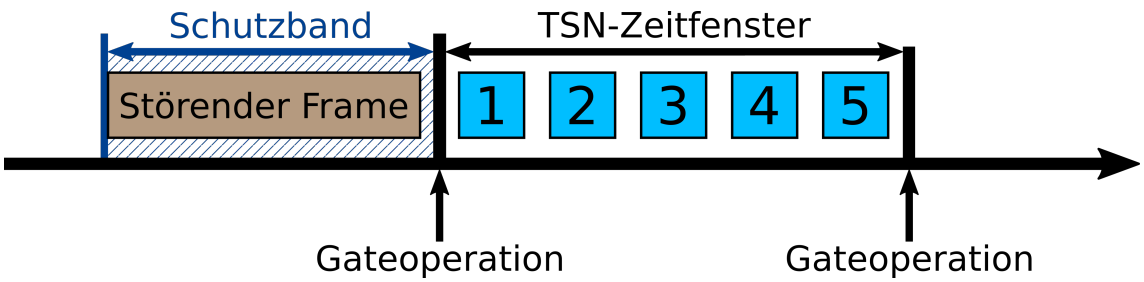


Abbildung 2.6: Spätes Eintreffen von Best-Effort-Traffic mit Schutzband [Pit16]

Explizit erfolgt das durch das Einfügen „leerer“ Einträge vor Zeitfenstern, die zeitkritischen Verkehr enthalten, d. h. Einträge, in denen alle Gates geschlossen sind. Bei der Auslegung der Schutzband-Größe wird mit der Worst-Case Annahme gerechnet, dass im letzten Moment vor der Gateoperation ein Best-Effort Frame maximaler Länge ankommt. Dieser muss im Schutzband vollständig – inklusive aller Pausen und Verzögerungen – übertragen werden können, damit im darauffolgenden Zeitfenster von Beginn an Frames aus der richtigen Verkehrsklasse gesendet werden können.

2.2.6.1 Length-aware Scheduling

Implizit kann das Schutzband mit dem *Length-aware Scheduling* Verfahren umgesetzt werden. Dabei werden nur solche Frames aus den Queues entnommen, deren Versand vollständig im aktuellen Zeitfenster möglich ist. Dafür müssen im Queuing-Netzwerk die Geschwindigkeiten der MAC-Schnittstelle bekannt sein, um die zur Übertragung benötigte Zeit zu berechnen. Des Weiteren muss der Switch für die Berechnung die Größe des Frames kennen, d. h. ihn vor Weitergabe vollständig erhalten und zwischengespeichert haben. Diese ist nicht bekannt, wenn der Switch nach dem *Cut-Through-Switching* Verfahren von [KK79] arbeitet, bei dem ein Frame – falls möglich – direkt weitergesendet wird, sobald die Zieladresse aus dem Ethernet Header gelesen wird.

Die Zeitspanne der Schutzbänder kann nicht für Datenverkehr eingeplant werden, wodurch verfügbare Bandbreite ungenutzt bleibt [LLPP16]. Daher sollten Schutzbänder selten eingesetzt und möglichst kurz gehalten werden, um möglichst wenig Bandbreite zu verschenken. Dieses Ziel kann durch darauf optimierte Scheduling-Verfahren wie in Abschnitt 2.3 oder durch *Frame-Preemption* wie in Unterabschnitt 2.2.7 verfolgt werden.

2.2.7 Frame-Preemption

Frame-Preemption ermöglicht die Unterbrechung (*Preemption*) und spätere Fortsetzung niederpriorisierter Frames, wodurch Schutzbänder verkürzt oder obsolet werden. Frame-Preemption ist eine Erweiterung der IEEE Standards, die sowohl im Bridging-Standard IEEE 802.1Q [Ins14], als auch im Ethernetstandard IEEE 802.3 [Ins16a] ansetzt [Ins16b] und sowohl in Zusammenarbeit mit Scheduling als auch unabhängig davon eingesetzt werden kann [Ins16d].

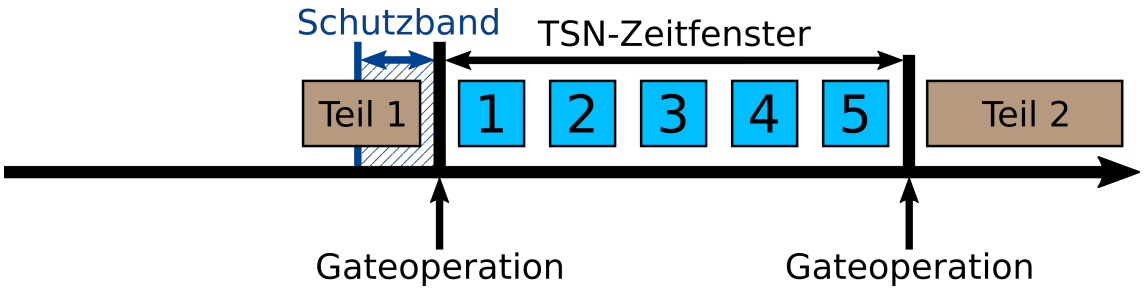


Abbildung 2.7: Spätes Eintreffen von Best-Effort-Traffic mit einem durch Frame-Preemption verkürztem Schutzband [Pit16]

Ein Best-Effort Frame, der, wie in Abbildung 2.5 abgebildet, in das wechselnde Zeitfenster des Schedules hereinragt, kann mit der Funktion unterbrochen werden. So kann der im nächsten Zeitfenster geschedulte Verkehr kann damit nahezu ohne Verzögerung passieren. Die Übertragung des unterbrochenen Frames wird im nächsten passenden Zeitfenster fortgesetzt, wie in Abbildung 2.7 schematisch dargestellt.

Bei Frame-Preemption wird zwischen *Preemptable-Frames* – unterbrechbaren Frames – und *Express-Frames* – bevorzugten Frames – unterschieden. Frames beider Kategorien sind inhaltlich normale Ethernet-Frames, werden aber abhängig ihrer Einteilung unterschiedlich behandelt. Preemptable-Frames können grundsätzlich durch Express-Frames unterbrochen werden, solange die konfigurierte Mindestlänge für Ethernet-Frames eingehalten wird. Express-Frames können nicht unterbrochen werden, sie haben die höchste Priorität.

Frame-Preemption kann auf Ethernet Verbindungen eingesetzt werden, wenn beide beteiligten MAC-Schnittstellen die Funktion unterstützen. Über das *Link Layer Discovery Protocol* (LLDP) wird kommuniziert und verifiziert, ob und mit welcher Konfiguration Frame-Preemption verwendet werden kann [Ins16b].

Abbildung 2.8 zeigt das Datenformat für sog. *MAC-Merge-Packets* (mPackets), das verwendet wird um Express- und Preemptable-Frames zu versenden. Die Übertragung von Frames beginnt wie im Ethernetstandard mit einer Präambel, um die Taktraten der beiden MAC-Schnittstellen zu synchronisieren. Die Präambel wird allerdings um ein Byte auf sechs Byte gekürzt, wenn die Fortsetzung eines unterbrochenen Preemptable-Frames übertragen wird, wie rechts in Abbildung 2.8 aufgelistet. In dem Fall wird ein Byte mit einem Fragmentzähler in der Struktur eingefügt, damit der Frame wieder in der richtigen Reihenfolge zusammengefügt werden kann. Frame-Preemption ersetzt das in Ethernet definierte 1-Byte *Start-Frame-Delimiter*-Datenfeld (SFD), um zusätzlich zum Beginn des Frames auch die Art des Frames zu kodieren: Express-Frame, Beginn oder Fortsetzung eines Preemptable-Frames. Startfragmente tragen dabei die gleiche Signatur wie der SFD für Ethernet-Frames, d. h. alle Express- und alle vollständigen Preemptable-Frames sind in der Übertragung äquivalent zu herkömmlichen Ethernet Frames. Nach der Unterbrechung eines Frames wird eine 4 Byte Prüfsumme über die versendeten Bytes berechnet und angehängt. Bei finalen Fragmenten wird diese allerdings nicht berechnet, sondern aus dem darunterliegenden, vollständigen Frame übernommen, da Ethernet-Frames an dieser Position bereits eine 4-Bytes Prüfsumme enthalten.

Preemptable-Frames können nicht an beliebiger Stelle abgebrochen werden, da für jedes versendete Fragment eine Mindestgröße gilt. In der Erweiterung IEEE 802.3br [Ins16b] ist eine minimale Framegröße von 64 Bytes definiert, jedoch kann die MAC-Schnittstelle des Empfängers eine

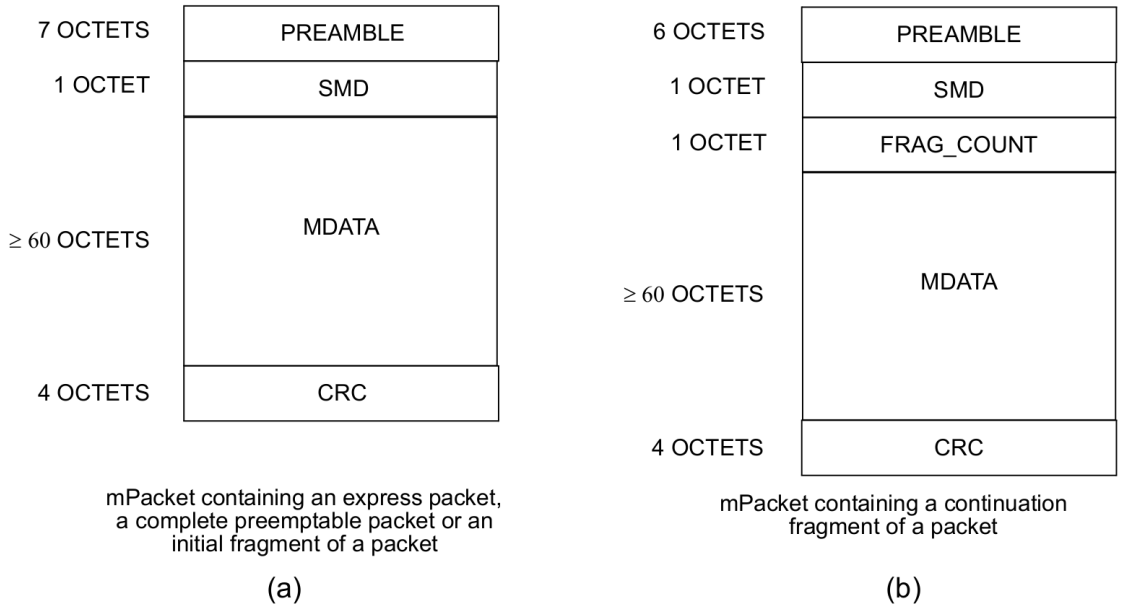


Abbildung 2.8: Übertragungsformat von Frames bei Frame-Preemption. Links: Express-Frame, Beginn eines Preemptable-Frames. Rechts: Fortsetzung eines Preemptable-Frames [Ins16b, Abb. 99-4]

Mindestgröße von bis zu 256 Bytes vorschreiben [Ins16d]. Das bedeutet, dass ein Preemptable-Frame nicht zu jedem Zeitpunkt der Übertragung unterbrechbar ist, z. B. wenn noch keine 64 Bytes übertragen wurden oder ein finales Fragment kleiner als 64 Bytes übrig bliebe. Bedingt durch diese Einschränkungen gibt es Frames, die generell nicht unterbrechbar sind, genau dann wenn diese kürzer als 124 Bytes sind [Ins16d]. Bezogen auf eine minimale Framegröße von 64 Bytes berechnet sich diese Limitierung aus dem kleinstmöglichen ersten Fragment von 60 Bytes + 4 Bytes Prüfsumme, sowie den übrigen 64 Bytes der Fortsetzung mit enthaltener Prüfsumme.

Mit Frame-Preemption kann das erforderliche Schutzband zwischen Best-Effort- und zeitkritischem Datenverkehr erheblich reduziert werden, wie im Vergleich zwischen Abbildung 2.7 und Abbildung 2.6 zu sehen ist. Die zu reservierende Zeit entspricht der Maximalgröße eines nicht-unterbrechbaren Frames, 123 Bytes. Im Vergleich dazu muss ohne Frame-Preemption ein Schutzband von 1522Bytes einberechnet werden, der Standard-MTU von Ethernet [Ins16a].

2.2.7.1 Hold und Release

Hold und *Release* sind Instruktionen an die Frame-Preemption Komponenten eines Switch-Ports, welche die Übertragung von Preemptable-Frames verbieten oder gestatten [Ins16d]. Ein *Hold-Request* sorgt dafür, dass ein potentiell in der Übertragung befindlicher Preemptable-Frame so bald wie möglich unterbrochen wird und keine weiteren Preemptable-Frames versendet werden, selbst wenn noch kein Express-Frame zur Übertragung bereitsteht. Ein *Release-Request* sorgt dafür, dass Frames wieder mit der üblichen Priorisierung übertragen werden, also erst jegliche Express-,

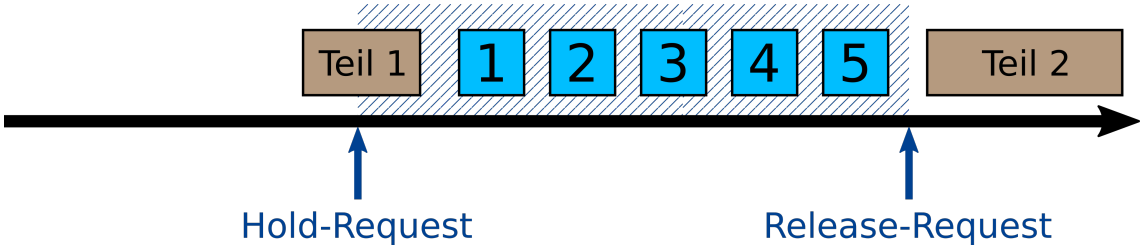


Abbildung 2.9: Frame-Preemption mit Hold und Release Mechanismus

dann ein begonnener Preemptable-Frame, dann neue Preemptable-Frames. Über den Hold und Release Mechanismus ist es möglich, ein explizites Schutzband um zeitkritischen Express Verkehr zu definieren, wie es in Abbildung 2.9 skizziert ist.

Ein Hold-Request muss koordiniert vor dem Zeitfenster für zeitkritischen Verkehr erfolgen, um diesen ausreichend zu schützen. Vor dem Wechsel des Zeitfensters muss ein ankommender, nicht unterbrechbarer Frame von 123 Bytes noch übertragen werden können und die nach Ethernetstandard [Ins16a] vorgeschriebene *Interframe Gap* vor Schalten der Gates verstreichen. Mit Hold and Release ist es ebenfalls möglich, die Gates für zeitkritischen und Best-Effort-Traffic gleichzeitig geöffnet zu lassen und lediglich den zeitkritischen Verkehr mit einem Hold-Request zu schützen. Dieses Szenario ist in Abbildung 2.9 veranschaulicht. Ohne den Hold und Release Mechanismus wäre es zwingend erforderlich, für zeitkritischen Verkehr isolierte Fenster zu haben, in denen nur jener Verkehr passieren kann. Alternativ könnten die Gates auch vollständig geöffnet bleiben, wenn pro Hop und Frame die Verzögerung durch die Übertragung eines nicht unterbrechbaren Frames in Kauf genommen werden kann.

2.2.8 Relevanz von TSN

TSN ermöglicht es, zeitkritische Anwendungen mit Ethernet zu realisieren und bringt damit eine standardisierte Technologie in viele Anwendungsbereiche, in denen bisher mehrere inkompatible Netzwerktechnologien parallel angewendet wurden. Die Standardisierung ermöglicht Kompatibilität zwischen Herstellern [Sch18] und erhöht die Entwicklungsbereitschaft bei vielen, vor allem kleinen, Herstellern [Avn]. TSN birgt als wesentlichen Vorteil, dass zeitkritischer Datenverkehr über das gleiche Netzwerk und simultan zu nicht zeitkritischem Verkehr gesendet werden kann [Jon18]. Das steht im Kontrast zum Status Quo, bei dem vollständig getrennte Bussysteme oder zu Ethernet inkompatible Erweiterungen verwendet werden müssen, um zeitliche Garantien für Datenverkehr zu geben. Vor allem in Industrieanlagen, Fahrzeugen, Flugzeugen und Versorgungssystemen gibt es über den zeitkritischen Verkehr hinaus niederpriore Kommunikation, die über getrennte Systeme realisiert werden muss. In diesen Domänen gibt es einerseits Steuergeräte, die klar zeitkritisch gesteuert oder sogar Regelkreisläufen unterliegen, aber auch Anwendungen, die keine Echtzeitanforderungen haben. Das umfasst andere Geräte im gleichen Umfeld wie Multimedia- und Komfortsysteme in Fortbewegungsmitteln, Überwachungskameras, Anwender-PCs oder Regelungen für unbeteiligte Systeme. Weiterhin können auch die zeitkritischen Steuergeräte Schnittstellen besitzen, denen die Servicequalität von Ethernet genügt, wie Wartungs-, Update-, Diagnose- und Statistikanwendungen. Durch die Kombination der beiden Netzwerke in ein *konvergentes Netzwerk* wird nicht nur der Overhead mehrerer Netzwerke gespart, sondern es kann auch auf bewährte Protokolle und Systeme auf Ethernetbasis zurückgegriffen werden. Mit Ethernet als etablierter Netzwerkbasis

wird Einheitlichkeit geschaffen, redundante und inkompatible Netzwerke werden vermieden, die Skalierbarkeit verbessert und die Anschaffungskosten werden gesenkt [Ada17]. Durch Gateways oder eine entsprechende Standardbehandlung des Verkehrs an TSN-Switchen können sogar Geräte verwendet werden, die TSN aufgrund von Alter oder vorgesehener Domäne nicht formal unterstützen. Steuerungssysteme auf Ethernetbasis zu realisieren, erleichtert zudem den Einstieg in und die Verwaltung solcher Netze, da keine weiteren, herstellergebundenen Kontrollsysteme notwendig sind und Ethernet selbst in Endanwendergeräten vorherrschend ist.

2.2.9 Verbreitung

Ein Teil der TSN-Standards ist bereits ratifiziert und publiziert, jedoch sind einzelne Standards seit 2017 noch in der Entwurfsphase der IEEE Spezifikation [Ins18]. Viele namhafte Hersteller haben die Entwicklung TSN-kompatibler Geräte angekündigt und haben sich in Konsortien wie der AVNU [Avn] oder dem Industrial Internet Consortium zusammengeschlossen. Im *Industrial Internet Consortium* arbeiten über 20 Hersteller zusammen, um kompatible Implementierungen zu realisieren [Ind], unter anderem mittels eines *Testbeds*, an dem Geräte direkt miteinander getestet werden können. Teilweise sind bereits integrierte Lösungen [NXP] oder einzelne standardkonforme Produkte [Hab17] [Ren] verfügbar. Weiterhin gibt es Hersteller, die existierende Produkte mit ähnlichem Zweck aktualisieren und um TSN-Funktionalitäten erweitern [Sie].

2.3 Scheduling-Verfahren

Der TSN-Standard IEEE 802.1Qbv [Ins16c] befasst sich nur mit der technischen Implementierung von Schedules, die inhaltliche Konfiguration ist dem Administrator überlassen und extern zu lösen. Um Schedules für realen Bedingungen zu erstellen, müssen gewisse Verzögerungen einbezogen werden, die in Ethernet-Netzwerken auftreten. Die Verzögerungen werden im Folgenden beschrieben. Anschließend wird auf drei Scheduling-Verfahren für TSN eingegangen, die in dieser Arbeit verwendet wurden.

2.3.1 Verzögerungen in Netzwerken

Bei der zeitlichen Planung und Analyse von Ethernet-Netzwerken sind verschiedene Verzögerungen (Delays) zu unterscheiden [DK+14]:

Processing Delay ist die benötigte Zeit im Weiterleitungsprozess eines Switches, um einen erhaltenen Frame an den richtigen Ausgangsport zu leiten. Auf State-of-the-Art Gigabit-Switchen beträgt das Processing-Delay zwischen 3μs und 5μs[DK+14].

Transmission Delay ist die Verzögerung beim Senden eines Frames, die durch die Modulierung auf das Medium benötigt wird. Das Transmission Delay ist abhängig der Übertragungsgeschwindigkeit, beispielsweise können 125Bytes bei Gigabit-Ethernit in 1μs übertragen werden.

Propagation Delay ist die Verzögerung eines Frames, die auftritt, während der Frame vom einen zum anderen Ende des Mediums geschickt wird. Sie hängt vom physischen Medium ab und bewegt sich bei Kupferkabeln in der Größenordnung Nanosekunden.

Queuing Delay ist die Verzögerung eines Frames, die durch Wartezeit in der Queue des Ausgangsports eines Switches auftritt.

2.3.2 Scheduling-Problem

Es ist nicht vorgesehen, dass Schedules zur Laufzeit berechnet werden, sie sind Konfigurationen, welche allenfalls über eine Administrationsschnittstelle getauscht werden können [Ins16c], ansonsten bleiben Schedules statisch. Als das *Scheduling-Problem* wird die Aufgabe bezeichnet, für eine gegebene Netzwerktopologie und eine Liste von Flows, einen Schedule zu berechnen. Bei der Berechnung können viele Einschränkungen eine Rolle spielen:

- Länge und Perioden der Flows
- Ende-zu-Ende Latenz von Flows
- vorberechnete Pfade (Routing)
- Bandbreitenauslastung einzelner Verbindungen
- Anzahl der Gate-Operationen
- Zeitbereiche, in denen kein TSN-Traffic gescheduled wird

Wie bei vielen Problemen der Informatik kann das Scheduling-Problem als Entscheidungsproblem oder Optimierungsproblem bezüglich mancher der genannten Eigenschaften angegangen werden. In [DN16] wird das Scheduling-Problem als *No-Wait Job-Shop Scheduling* Problem ausgedrückt und ist damit NP-schwer [SL86].

Für TSN-Scheduling wurden in existierenden Arbeiten verschiedene Schedulingverfahren entworfen, umgesetzt und analysiert. Die folgenden Schedulingverfahren sind Grundlage des Scheduling-Teils der in dieser Arbeit ausgeführten Netzwerksimulationen und standen als Python-Implementierung zur Verfügung.

2.3.3 Joint Routing und Scheduling

In [Jon18] werden zwei Möglichkeiten betrachtet, um zeitkritischen und Best-Effort-Traffic in einem konvergenten Netzwerk zu trennen und so Störungen zwischen den Verkehrsarten zu vermeiden: zeitliche (Scheduling) und räumliche Trennung (Routing). In dem Paper werden die beiden Verfahren kombiniert und damit das sog. *Joint Routing und Scheduling* Problem betrachtet. Es werden hierfür lineare Gleichungen für ein *Integer Linear Programming Framework* (ILP) formuliert, welches mögliche Lösungen finden soll. Für die Lösungsfindung muss für jeden Flow ein Pfad durch das gegebene Netzwerk gefunden werden, der unter Inbezugnahme der Delays entlang des Pfades die für den Flow verlangte Ende-zu-Ende Latenz einhält. Beim Scheduling ist das Problem genau auf TSN zugeschnitten, indem *First-In-First-Out Verbindungen* (FIFO) angenommen und *zero queuing* erfordert wird. Zero Queuing bedeutet, dass ein Flow in jedem Switch sofort weitergeleitet wird,

ohne zuerst in einer Queue warten zu müssen, abgesehen von technisch bedingten Delays. Die Python-Implementierung aus [Jon18] wurde in dieser Arbeit für die Generierung von Topologien, Flow-Anforderungen und die Lösung mittels des beschriebenen Verfahrens verwendet.

2.3.4 No-Wait Packet Scheduling

Bei *No-Wait Packet Scheduling* [DN16] wurde das Scheduling-Problem auf das NP-schwere [SL86] *No-Wait Job Shop Scheduling Problem* (*NW-JSP*) reduziert und anschließend ebenfalls mit einem ILP gelöst. Damit ist das Scheduling ebenfalls NP-schwer. Das No-Wait Job Shop Scheduling Problem ist eine Erweiterung des im Folgenden erklärten *Job Shop Scheduling Problems*.

2.3.4.1 Job-Shop Scheduling

Das *Job Shop Scheduling Problem* (NW-JSP) wird am Beispiel einer Werkstatt erklärt, in der Werkstücke an Maschinen bearbeitet werden. Dabei müssen die Bearbeitungsschritte eines Werkstücks in einer definierten Reihenfolge erledigt werden, an einer Maschine kann nur ein Werkstück gleichzeitig bearbeitet werden. Die *No-Wait* Verschärfung diktiert, dass sobald die Arbeit an einem Werkstück begonnen wurde, diese ohne Unterbrechung über alle Maschinen hinweg fortgesetzt werden muss [DN16]. Das Optimierungsziel von Job-Shop Scheduling ist eine niedrige *Makespan*, die Zeit, nach der alle Aufgaben erledigt wurden.

2.3.4.2 Anpassung an TSN-Scheduling

Für die Anwendung in TSN werden die Scheduling-Anforderungen und Komponenten auf das beschriebene NW-JSP abgebildet. Hierbei sind Switch-Ports als Maschinen und Flows als Werkstücke bzw. Aufgaben an den Maschinen vorzustellen. Durch die No-Wait Einschränkung genügen die Startzeiten der Flows als Lösungsformat. Es kommt eine Variation der Tabu-Suche zur Festlegung der Startreihenfolge zum Einsatz, bei der die Flows initial geordnet werden, die Makespan berechnet wird und pro Iteration der am spätesten platzierte Flow neu platziert wird.

Die Makespan-Optimierung führt beim TSN-Scheduling dazu, dass der TSN-Traffic in einem Block zu Beginn jedes Zyklus platziert wird. Das sorgt für eine geringe Anzahl an Gate-Operationen und damit für weniger durch Schutzbänder verschenkte Bandbreite.

2.3.5 SMT Scheduler

In [COCS16] kommt statt einem ILP ein *Satisfiability Modulo Theories Löser* (SMT) zur Erstellung von Schedules zum Einsatz. Bei SMT wird ein Gleichungssystem mit Prädikatenlogik formuliert, welches darauffolgend als Entscheidungsproblem gelöst wird. Die Forscher von TTTech orientieren sich in [COCS16] an Scheduling-Verfahren, die zuvor für das zu TSN-Scheduling ähnliche TTEthernet entwickelt wurden. Bei TTEthernet definiert der Schedule eines Ports, zu welcher Zeit ein Frame versendet wird. TSN-Schedules beschränken sich auf die Entnahme von Frames aus einer *Queue*, in welcher sich auch mehrere Flows verschiedener Länge sammeln können.

Die Verschachtelung mehrerer Flows an einem Switch-Port wurde daher wie im Folgenden beschrieben untersucht.

2.3.5.1 Egress Interleaving

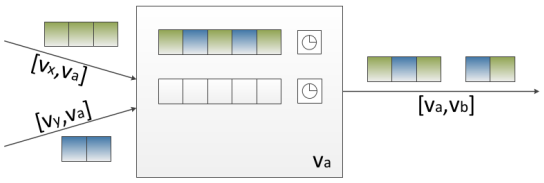


Abbildung 2.10: *Egress Interleaving* am Ausgangsport eines Switches [COCS16, Abbildung 2a]

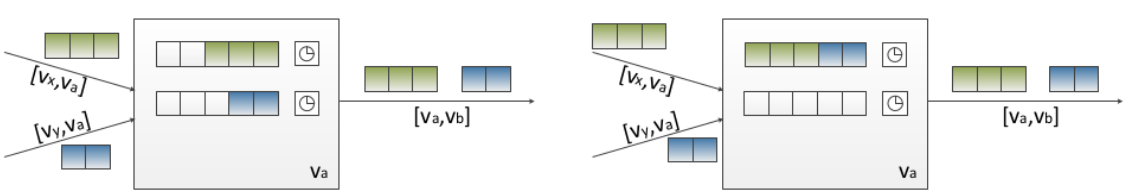


Abbildung 2.11: Lösungen zum Verhindern von Egress Interleaving [COCS16, Abbildung 2b,2c]

Egress Interleaving beschreibt die in Abbildung 2.10 abgebildete Situation, bei der Flows an einem Ausgangsport in der falschen Reihenfolge oder Aufteilung versendet werden. Eine solche Sortierung der Queues kann durch die parallele oder verschachtelte Ankunft zweier Flows an einem Switch oder verlorene Frames verursacht werden. Durch die falsche Aufteilung wie in Abbildung 2.10 können Verzögerungen und Jitter (siehe Abschnitt 2.2) auftreten, was sich über weitere Hops entlang dem Pfad des Flows propagieren kann. Initial wurde ein *Flow Isolation* Ansatz versucht, bei dem ein Flow nur in eine Queue eingeordnet wird, wenn der vorherige Flow vollständig dort übertragen wurde. Das kann durch die Aufteilung der Flows in verschiedene Queues geschehen, wie in Abbildung 2.11 links dargestellt. Ebenfalls ist eine zeitliche Trennung der Flows einer Queue möglich, wie in Abbildung 2.11 rechts veranschaulicht. Es lassen sich jedoch leicht Flow-Anforderungen definieren, mit denen bei diesen Einschränkungen mehr Queues benötigt werden als im System vorhanden sind. Darum wurde die ursprüngliche Anforderung gelockert, indem die Verschachtelung von Frames verschiedener Flows erlaubt wird, solange zu jedem Zeitpunkt nur Frames in einer Queue sind, die zum gleichen Flow gehören.

2.3.5.2 Scheduling und Optimierung

Ein SMT Löser liefert, sofern das Problem lösbar ist, eine mögliche Lösung als Modell zurück. In [COCS16] wird der Scheduler inkrementell implementiert, indem schrittweise Flows hinzugefügt werden und die vorhandene Lösung der vorherigen Runde als Einschränkung definiert wird. Existiert für ein Problem keine Lösung, wird *Backtracking* verwendet, indem der letzte und vorletzte hinzugefügte Flow entfernt und gemeinsam neu hinzugefügt werden. Die Forscher verwendeten

Z3 [Mic] als Löser, welcher neben SMT Lösungen lineare Optimierung anwendet. Ihre Implementierung sucht nach möglichen Schedules und minimiert die benötigten Queues bzw. gibt an, ab wie vielen Queues das gegebene Problem lösbar wird.

2.4 NeSTiNg-Projekt

NeSTiNg (*Network Simulator for Time Sensitive Networking*, [CDF+18]) ist ein OMNeT++-Projekt, das Simulationen von TSN ermöglicht. Im Rahmen eines Studienprojekts befassten sich sieben Studierende der Universität Stuttgart – inklusive dem Verfasser der Arbeit – für zwei Semester mit den TSN-Standards sowie dem Entwurf und der Implementierung des NeSTiNg-Projekts. Das Projekt wird durch Studierende und wissenschaftliche Mitarbeiter der Universität Stuttgart weiterhin aktiv entwickelt.

2.4.1 Funktionen

Das Resultat des Studienprojekts sind verschiedene OMNeT++-Module, mit denen unter anderem zeitgesteuerter Verkehr simuliert werden kann. Hosts können einen Schedule für zeitgesteuerten TSN-Traffic haben oder Best-Effort-Traffic senden, welcher an Switchen dann standardmäßig in eine bestimmte Queue eingeordnet wird. Switche unterstützen Gating und Scheduling, Length-aware Scheduling, den Credit-Based Shaper, MAC-basiertes Forwarding und Processing Delays.

2.4.2 Architektur

2.4.2.1 Switch

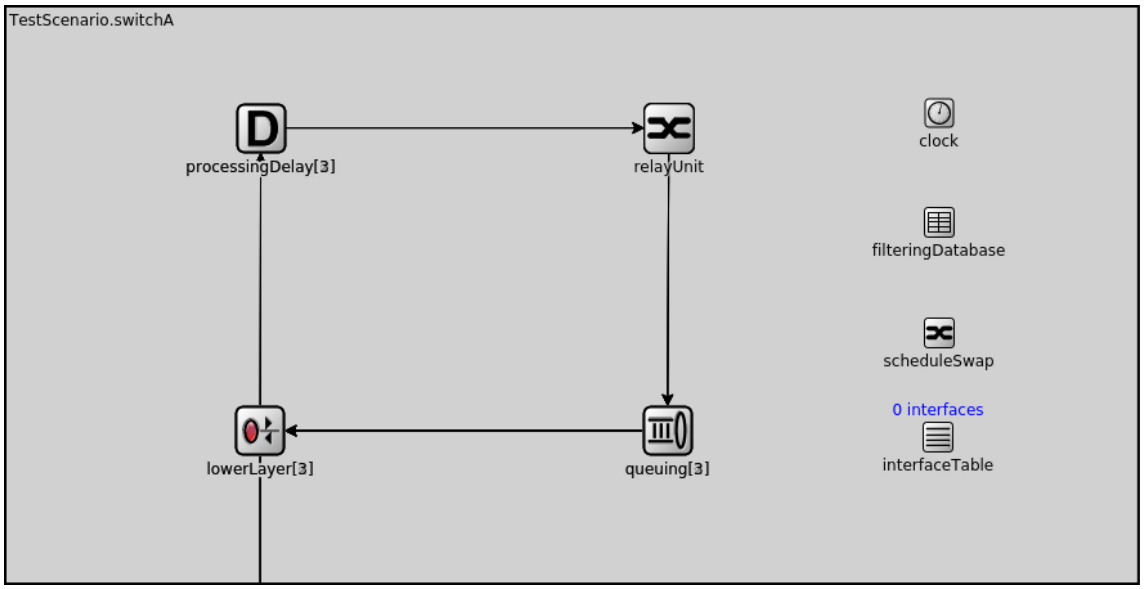


Abbildung 2.12: NeSTiNg TSN-Switch-Architektur in OMNeT++ [CDF+18]

In Abbildung 2.12 ist der strukturelle Aufbau des TSN-Switches im NeSTiNg-Projekt zu sehen. Komponenten mit eckigen Klammern stellen Vektoren von gleichartigen Modulen dar, wobei die Dimension hier der Anzahl der Switch-Ports entspricht. Pakete kommen links unten über die *niederen Layer* (MAC-Komponente und VLAN-Behandlung) an, durchlaufen den Switch im Uhrzeigersinn und verlassen ihn wieder durch die niederen Layer an einem entsprechenden Ausgangsport. Bis auf die *Relay Unit* sind alle Module im Pfad Vektormodule, d. h. die restlichen Module sind pro Port vorhanden und einem bestimmten Port zuzuordnen. In Richtung des durchlaufenden Frames ist alles „vor“ der Relay Unit einem Eingangsport zuzuordnen, alles danach einem Ausgangsport. In den niederen Layern befindet sich eine INET-MAC-Komponente und Module, die ankommende Frames von Ethernet- und VLAN-Tags befreien. Das Processing Delay verzögert die Weiterleitung um eine eingestellte Zeitspanne. Die Relay Unit leitet ankommende Frames anhand ihrer Ziel-MAC-Adresse an den entsprechenden Zielport weiter.

2.4.2.2 Queuing-Netzwerk

Im *Queuing-Netzwerk* finden die meisten Handlungen statt, die TSN in diesem Projekt betreffen.

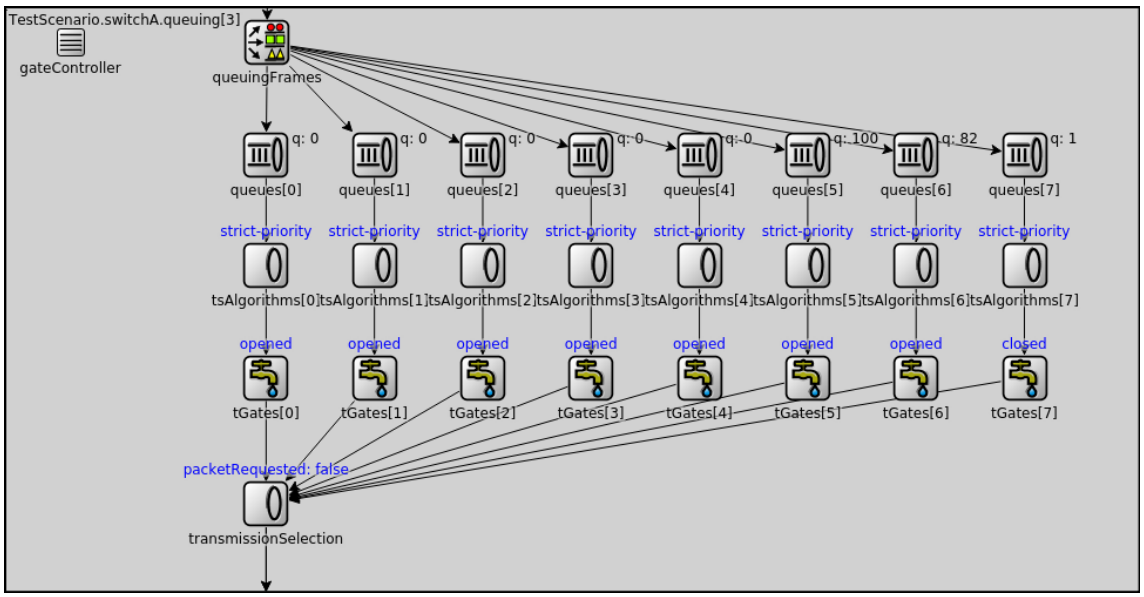


Abbildung 2.13: NeSTiNg TSN Queuing-Netzwerk in OMNeT++ [CDF+18]

Frames kommen im Queuing-Netzwerk oben an (Abbildung 2.13) und durchlaufen es durch die Queues nach unten, bis zur Transmission-Selection. Frames werden in den Queues (1. Ebene) angesammelt, danach ggf. von einem Transmission-Selection Algorithmus (2. Ebene) behandelt und durch ein Gate (3. Ebene) aufgehalten oder durchgelassen. Der Gate-Controller, links in Abbildung 2.13, steuert die Zustände der einzelnen Gates abhängig seines Schedules, der jeweils für eine Zeitspanne einen Bit-Vektor mit den jeweiligen Gate-Zuständen vorgibt. Die Abarbeitung des Schedules geschieht in Zusammenarbeit mit der Uhr des Switches, bei der sich Komponenten mit einer Zeitspanne registrieren können, nach deren Ablauf sie über die veränderte Zeitspanne informiert werden. Initial werden die Gates in den ersten Zustand des Schedules versetzt, danach werden die Zustände jeweils nach Ablauf der Zeit verändert. Die Transmission-Selection kommuniziert

über Methodenaufrufe mit der MAC-Komponente in den niederen Layern, welche mit einem *Packet-Request* neue Frames anfordert, sobald sie auf ihrem Medium wieder senden darf. Der Request wird an der Transmission-Selection gespeichert – so kann er später bedient werden, falls momentan keine Frames aus dem Netzwerk entnommen werden können. Wenn so eine Anfrage vorliegt, versucht die Transmission-Selection einen Frame aus den Queues zu entnehmen, sofern die Gates und Algorithmen das zulassen. Beim Einfügen von Frames werden ebenfalls in INET übliche Methoden angewendet, mit denen andere Komponenten über verfügbare Elemente in einer Queue informiert werden. So verhalten sich alle Elemente in den drei Ebenen wie eine INET-Queue, d. h. sie informieren die darunterliegende Ebene, sobald sie selbst bereit sind, einen Frame weiterzugeben. Dieses Informationsverhalten wird in der Implementierung von Frame-Preemption im folgenden Abschnitt verwendet und angepasst.

3 Design und Implementierung

In dieser Arbeit soll das Verhalten von TSN in Kombination mit Best-Effort-Traffic simuliert und die Ergebnisse untersucht werden. Die Analyse soll sich auf die Auswirkungen von TSN-Scheduling auf Best-Effort-Traffic konzentrieren, da dieser im Gegensatz zu TSN-Traffic nicht speziell auf Scheduling ausgelegt ist bzw. überhaupt Kenntnis über die Einschränkungen durch TSN hat.

Die Implementierung gliedert sich in zwei Bereiche, die Implementierungen im OMNeT++-Projekt *NeSTiNg*, um Frame-Preemption zu unterstützen, sowie die Automatisierung von OMNeT++-Simulationen inklusive der Schedule-Erzeugung und Auswertung.

3.1 Frame-Preemption

Das NeSTiNg-Projekt ist, wie in Abschnitt 2.4 beschrieben, eine TSN-Implementierung in OMNeT++ und wurde im Verlauf der Arbeit um Frame-Preemption-Funktionalitäten erweitert. Das NeSTiNg-Projekt enthielt schon zuvor eine Implementierung von Frame-Preemption, welche jedoch nicht vollständig und auf zu hohem Layer ansetzte. Die Stückelung fand schon im Queuing-Netzwerk statt, der IEEE 802.3br-Standard [Ins16b] platziert die Aufspaltung der Frames jedoch nahe der MAC-Schnittstelle, wo etwaige Pakete schon zu Frames verpackt sind und auch keine Unterscheidung von Verkehrsklassen mehr stattfindet.

Im Folgenden wird die durch den IEEE-Standard vorgeschlagene, und die für das NeSTiNg-Projekt gewählte Architektur für Frame-Preemption beschrieben.

3.1.1 Architektur im Standard

Der IEEE 802.3br Standard [Ins16b] realisiert Frame-Preemption mit zwei logischen MAC-Komponenten, eine für Express-Frames, die andere für Preemptable-Frames, wie in Abbildung 3.1 dargestellt. Zu sendende Ethernet-Frames werden entsprechend ihrer Klassifizierung an die Preemptable-MAC (*pMAC*) oder Express-MAC-Komponente (*eMAC*) weitergegeben. Der *MAC Merge Layer* kommuniziert vorerst über LLDP mit der MAC-Schnittstelle des nächsten Hops und aktiviert Frame-Preemption erst, wenn dieser es unterstützt. Dann werden ankommende Frames gemäß ihrer Klassifizierung an eine der beiden MAC-Komponenten weitergegeben. Insbesondere kümmert sich der MAC Merge Layer um die Priorisierung und Unterbrechung von Frames sowie den in Unterunterabschnitt 2.2.7.1 beschriebenen Hold und Release Mechanismus. Wird Frame-Preemption nicht unterstützt oder ist die Funktion nicht aktiviert, werden alle ankommenden Frames an die Express-MAC-Komponente weitergegeben und Frames anhand ihrer Klassifizierung nur priorisiert, jedoch nicht unterbrochen [Ins16b].

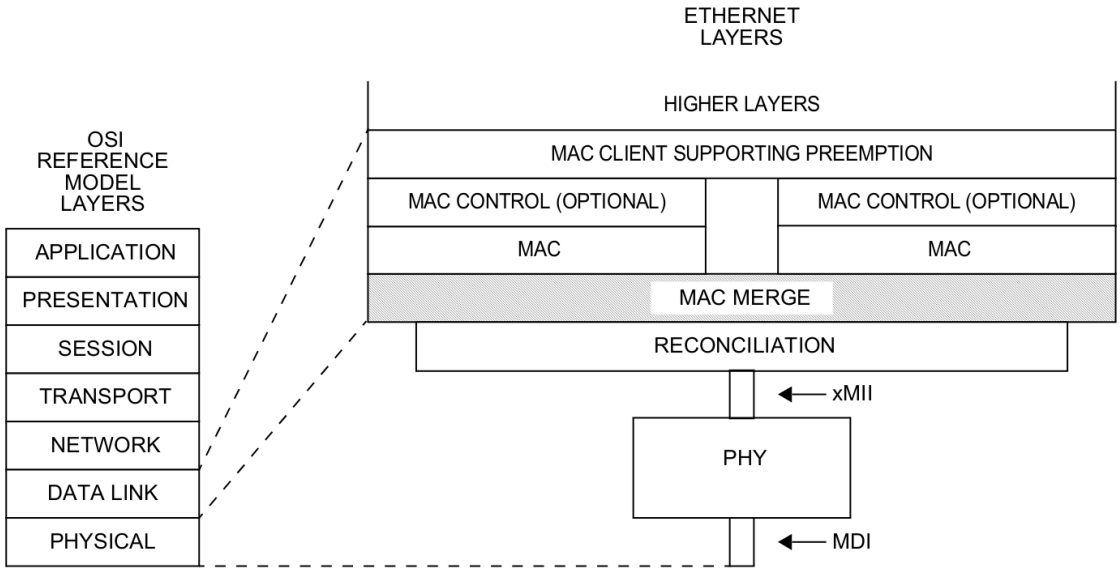


Abbildung 3.1: Architektur der MAC-Komponenten für Frame-Preemption in Relation zum ISO/OSI-Referenzmodell [Ins16b, Abbildung 99-1]

3.1.2 Umsetzung im NeSTiNg-Projekt

Die Frame-Preemption Architektur weicht in der NeSTiNg-Implementierung von der im Standard gewählten ab, da nicht alle Funktionen benötigt werden und die vorgeschlagene Struktur mit bestehenden INET-Komponenten schwierig umzusetzen ist. Auf die Konfiguration mittels LLDP wird hier verzichtet, da alle Switche zur Laufzeit bereits platziert sind und die Funktion in der INI-Datei einer Simulation zentral konfiguriert wird, wie in Auflistung 3.1 veranschaulicht. Weiterhin müssen Prüfsummen und Zählervariablen keine echten Werte enthalten, da von der Verwendung verlustfreier Verbindungen ausgegangen wird.

3.1.2.1 Queuing-Netzwek

```
[General]
**.switchA.lowerLayer[3].mac.enablePreemptingFrames = true
**.queues[0].expressQueue = false
**.queues[1].expressQueue = false
**.queues[2].expressQueue = false
**.queues[3].expressQueue = false
**.queues[4].expressQueue = false
**.queues[5].expressQueue = false
**.queues[6].expressQueue = false
**.queues[7].expressQueue = true
```

Auflistung 3.1: INI-Konfiguration für Frame-Preemption

Die Klassifizierung von Datenverkehr als Express- oder Preemptable-Frame erfolgt über die existierenden Verkehrsklassen bzw. Queues. Die Einteilung kann mit einem Parameter in der INI-Datei der Simulation wie in Auflistung 3.1 vorgenommen werden. Die mittleren drei Ebenen in Abbildung 2.13 implementieren ein neu eingeführtes C++-Interface `IPreemptableQueue`, über das die Transmission-Selection die gewählte Priorität abrufen kann. Die Transmission-Selection priorisiert damit zusätzlich zu den in Unterabschnitt 2.2.5 beschriebenen Bedingungen auch anhand der Einteilung als Express- bzw. Preemptable-Frame. Weiterhin erhält sie ein zweites ausgehendes NED-Gate und damit einen zweiten Pfad zu den niederen Layern, worüber die klassifizierten Frames getrennt übertragen werden.

In den niederen Layern sind ebenfalls getrennte Pfade notwendig, wie im folgenden Abschnitt erläutert wird.

3.1.2.2 Niedere Layer

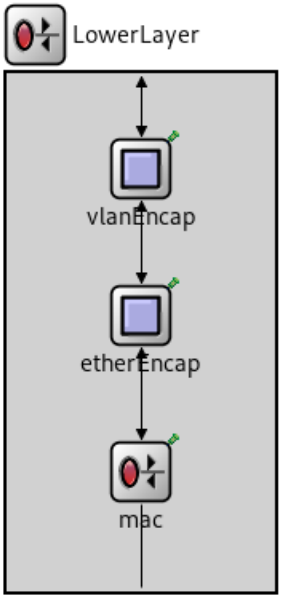
Abbildung 3.2 zeigt die Architektur des OMNeT++-Modules für die niederen Layer in NeSTiNg, vor und nach der Implementierung von Frame-Preemption.

Die Einteilung in Express- oder Preemptable-Frame ist nur in den Queues im darüber liegenden Layer bekannt und kann nicht aus übertragenen Paketen gelesen werden. Das *Lower Layer* Modul bietet folglich zwei Schnittstellen für eingehende Pakete höherer Layer, um die Einteilung über den gewählten Pfad zu erkennen. Die Komponenten für die Kapselung mit VLAN-Tags und Ethernet-Headern wurden in dem zusammengesetzten Modul in Abbildung 3.2b mehrfach instantiiert, jeweils für Preemptable- und Express-Frames. Die Kapselung muss mangels direkter Erkennung der Priorität eines Pakets getrennt für beide Pfade erfolgen.

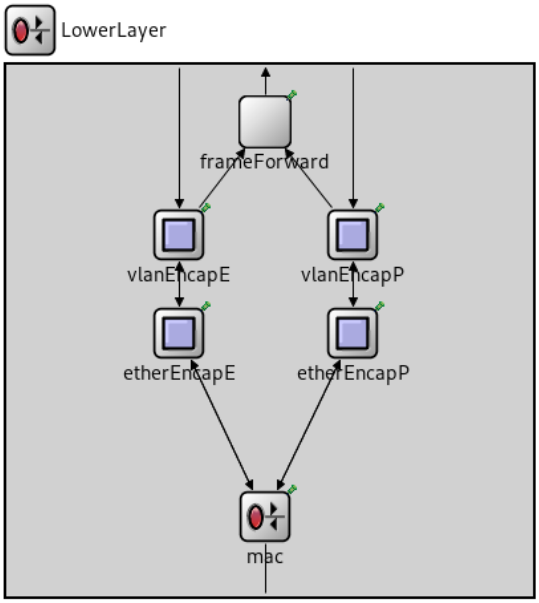
Für die Empfangsrichtung genügt das existierende NED-Gate, da vollständig empfangene Frames in den oberen Layern gleich behandelt werden. Die *Frame Forward* Komponente leitet empfangene Frames zu den höheren Layern weiter und ist nur dadurch notwendig, da zwei ausgehende NED-Gates in OMNeT++ nicht an ein eingehendes NED-Gate angeschlossen werden können.

3.1.2.3 Preemptable MAC-Komponente

Im NeSTiNg-Projekt wurde die Funktion der pMAC, eMAC und des MAC Merge Layers in einer MAC-Komponente zusammengefasst. Dies liegt daran, dass ein existierendes INET-Modul, `EtherMACFullDuplex`, für die Ethernet-Kommunikation verwendet wurde, um Kompatibilität zu erhalten und die erweiterte Komponente ggf. später zum INET-Framework beitragen zu können. Für eine Aufteilung der Funktionalitäten auf mehrere Komponenten hätte das Modul viele seiner internen Abläufe und Zustände nach außen hin kommunizieren müssen sowie komplexe Schnittstellen bereitstellen müssen. Die bisher verwendete INET-MAC-Implementierung `EtherMACFullDuplex` wurde in der NED-Definition und C++-Implementierung um Frame-Preemption Funktionalitäten zu dem Modul `EtherMACFullDuplexPreemptable` erweitert. In den folgenden Abschnitten ist mit der MAC-Komponente die neue, um Frame-Preemption erweiterte Version gemeint.



(a) Ursprüngliche Architektur der niederen Layer in NeSTiNg [CDF+18]



(b) An Frame-Preemption angepasste Architektur der niederen Layer in NeSTiNg [CDF+18]

Abbildung 3.2: Architektur der niederen Layer in NeSTiNg [CDF+18], mit und ohne der Frame-Preemption Implementierung

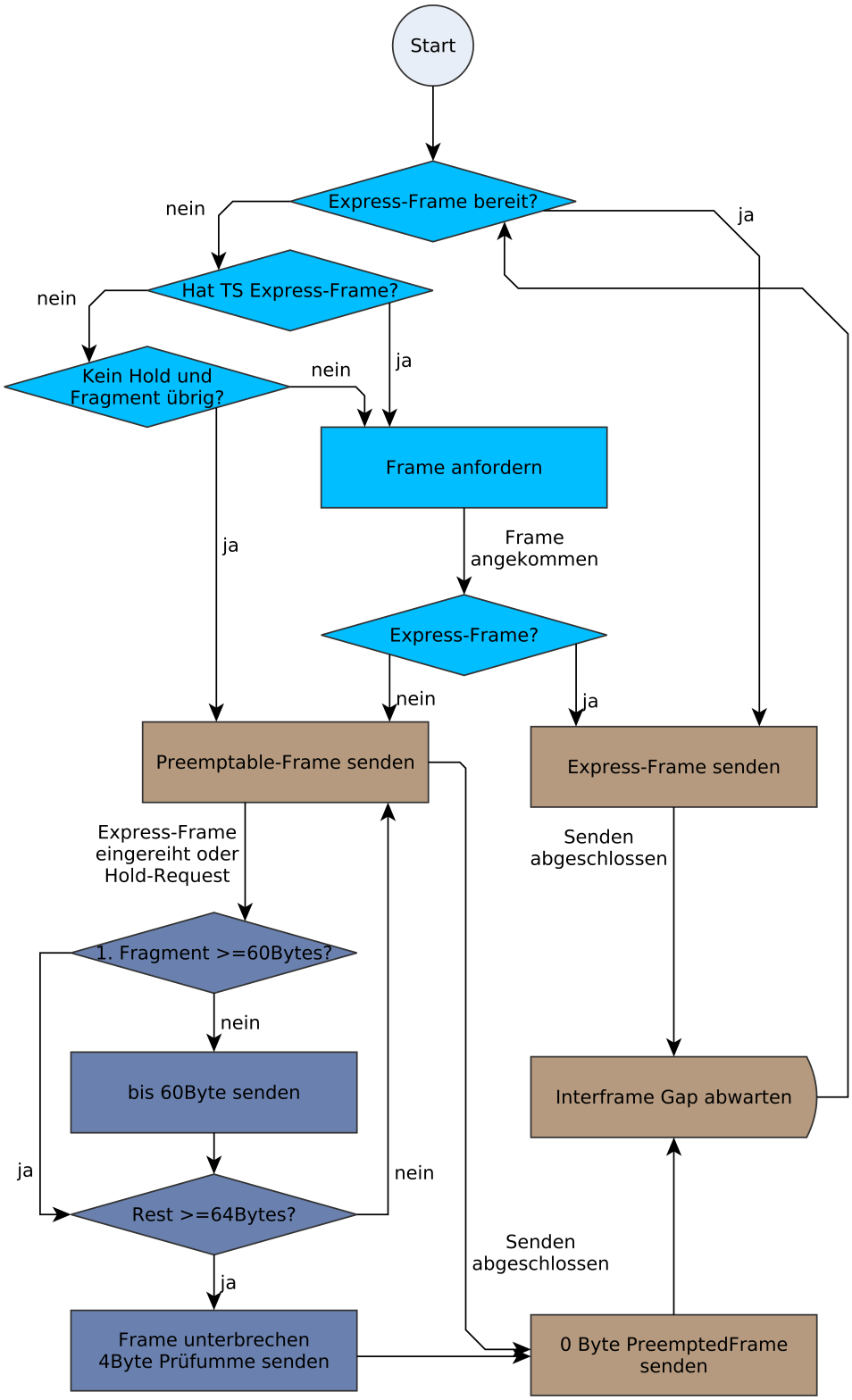


Abbildung 3.3: Ablauf der Frame-Auswahl, des Sendevorgangs und der Unterbrechung in `EtherMACFullDuplexPreemptable`. *TS* steht in diesem Kontext für Transmission-Selection.

Zustandsautomat Abbildung 3.3 zeigt den internen Ablauf des Sendevorgangs der MAC-Komponente, reduziert auf die für Frame-Preemption relevanten Bereiche. Hellblaue Komponenten beschreiben den Prozess, um im wartenden Zustand den nächsten zu sendenden Frame zu wählen, braune Komponenten den regulären Sendeprozess und dunkelblaue Komponenten den Fall der Unterbrechung. Bei der Auswahl des nächsten Frames gilt die folgende Priorität:

1. vorliegender Express-Frame
2. angeforderter Express-Frame
3. vorliegendes Preemptable-Frame Fragment – nur wenn kein Hold-Request vorliegt
4. angeforderter Preemptable-Frame – nur wenn kein Hold-Request vorliegt

Die für Unterbrechungen einzuhaltenden Beschränkungen sind hier am Beispiel einer minimalen Fragmentgröße von 64 Bytes formuliert, in der Implementierung lassen sich diese Größen durch Konstanten anpassen. Die erforderliche Mindestgröße für ein Fragment hängt wie in Unterabschnitt 2.2.7 erläutert davon ab, ob der übertragene Teil am Anfang, in der Mitte oder am Ende des Frames liegt. Fragmente am Anfang oder aus der Mitte des originalen Frames müssen nur 60 Bytes groß sein, da die Mindestgröße von 64 Bytes durch Anhang einer 4-Bytes-Prüfsumme erreicht wird. Dem finalen Fragment wird keine Prüfsumme angehängt, da die letzten vier Bytes bereits eine Prüfsumme des darunterliegenden, vollständigen Frames enthalten. Somit muss dieses Fragment 64 Bytes lang sein, um die Mindestgröße einzuhalten.

Im Weiteren wird die Implementierung der Unterbrechungsmechanismen sowie Hold und Release im Detail beschrieben.

Frame-Unterbrechung Der Versand von Frames über eine Verbindung geschieht in OMNeT++ als Objekt, das über einen sog. *Channel* (Kanal) transferiert wird. Das Objekt wird dabei vollständig versandt und nicht wie in echten Geräten über eine Zeitspanne bitweise moduliert. Verzögerungen wie das Transmission Delay werden durch Wartezeiten im Kanal realisiert, die von dem Bytegröße-Attribut des Objekts abhängen.

Das Abbrechen einer laufenden Übertragung gestaltet sich schwierig, da in den Kanälen für diesen Zweck nur die Methode `forceTransmissionFinishTime` [Var14a] existiert, welche den Kanal für eine weitere Übertragung freigibt, die gesendete Nachricht aber trotzdem nach der ursprünglich berechneten Zeit vollständig zustellt. Das Abbrechen des Nachrichten-Ereignisses durch die MAC-Komponente ist ebenfalls nicht möglich, da der Kanal das Objekt in der Zeitspanne der Übertragung besitzt, nur dieser könnte das Ereignis abbrechen. Auf die Erweiterung des Kanals um die Abbrechen-Funktionalität wurde verzichtet, um die Auswahl verschiedener Kanäle für Simulationen weiterhin offen zu halten.

Da Preemptable-Frames nicht gesendet und während der Übertragung abgebrochen werden können, wurde die Nachrichtenart `PreemptedFrame` erstellt, die eine Größe von null Bytes aufweist und einen Verweis auf das C++-Objekt des originalen Frames enthält. Soll an der MAC-Komponente ein Preemptable-Frame übertragen werden, verschickt diese vorerst nichts über den Kanal und versetzt sich in den „Preemptable-Frame Senden“-Zustand in Abbildung 3.3, in dem nichts weiteres übertragen wird. In dem Zustand bleibt sie so lange, wie die Übertragung des originalen Frames oder Fragmentes benötigen würde. Danach sendet sie einen `PreemptedFrame` mit der Referenz auf das Original-Objekt sowie der bis dahin insgesamt gesendeten Bytegröße des Fragments.

Soll ein Express-Frame den Preemptable-Frame unterbrechen, werden die geltenden Bedingungen für Unterbrechungen geprüft, wie in Abbildung 3.3 unten links dargestellt. Ist eine Unterbrechung möglich, wird die Wartezeit früher beendet und der `PreemptedFrame` mit einer Referenz auf den originalen Frame sowie der versendeten Anzahl an Bytes versendet. Die Unterbrechung geschieht nicht sofort, sondern erst nach dem Verstreichen der Zeit, die für die Übertragung der Prüfsumme benötigt worden wäre. Der Preemptable-Frame wird zusammen mit der bereits versandten Anzahl an Bytes gespeichert (ohne angehängte Prüfsumme) und fortgesetzt, wenn die Priorisierung oben links in Abbildung 3.3 so entscheidet.

Zusammenfügen von Segmenten Auf Empfängerseite wird der `PreemptedFrame` empfangen und die bis dahin versandte Bytegröße daraus gelesen. Entspricht die Größe der des originalen Frames, wird dieser an die höheren Layer weitergereicht, andernfalls wird der `PreemptedFrame` ignoriert. Die Nachrichten-ID des originalen Frames wird zwischengespeichert, um erkennen zu können, ob es sich bei einem empfangenen `PreemptedFrame` um eine Fortsetzung oder einen neuen Frame handelt.

Express-Frames Im existierenden Ablauf kennt die MAC-Komponente nur eine Quelle aus der ausgehende Frames bezogen werden und fordert erst nach erfolgter Übertragung des aktuellen Frames einen weiteren aus der Quelle an. Mit dieser Struktur lässt sich allenfalls eine Priorisierung von Express-Frames abbilden, welche die Transmission-Selection durch Zurückhalten von Preemptable-Frames vornimmt. Für die Unterbrechung eines Preemptable-Frames muss die MAC-Komponente während der laufenden Übertragung darüber in Kenntnis gesetzt werden, wenn ein Express-Frame im Queuing-Netzwerk eingereiht wurde. Hierfür registriert sich die MAC-Komponente bei der Transmission-Selection als Listener für das „packet enqueued“-Ereignis. Das Ereignis wird durch Einfügen eines Frames in eine vormals leere Queue verursacht und propagiert von dort nach unten bis zur Transmission-Selection in Unterunterabschnitt 2.4.2.2. In der `packetEnqueued`-Methode, die in der MAC-Komponente durch die Transmission-Selection aufgerufen wird, wird durch einen Methodenaufruf bei der Transmission-Selection geprüft, ob es sich bei dem eingefügten Frame um einen Express-Frame handelt. Der Informationsvorgang ist bewusst auf zwei Methoden aufgeteilt, da die zuerst aufgerufene `packetEnqueued`-Methode ursprünglich aus dem INET-Interface `IPassiveQueue` stammt und daher über jedes möglicherweise relevante Einfügen informiert.

Handelt es sich um einen Express-Frame, wird geprüft, ob momentan eine Unterbrechung stattfinden kann und diese gegebenenfalls, wie in Unterunterabschnitt 2.4.2.2 dunkelblau dargestellt, durchgeführt. Ist die Unterbrechung erst zu einem späteren Zeitpunkt möglich, wird dieser berechnet und die Unterbrechung zu diesem Zeitpunkt geplant. Das geschieht, indem die MAC-Komponente eine spezielle Nachricht „`preemptCurrentFrame`“ an sich selbst schickt, die um die fehlende Zeit bis zur nächstmöglichen Unterbrechung verzögert wird. Auf die Nachricht wird reagiert, indem die aktuelle Übertragung wie beschrieben unterbrochen wird.

Hold und Release Der Hold und Release Mechanismus kann, wie in Unterunterabschnitt 2.2.7.1 beschrieben, für einen Zeitraum ausschließlich die Übertragung von Express-Frames erlauben. Wird ein Hold-Request an die MAC-Komponente gestellt, soll sie fortan nur Express-Frames übertragen und einen momentan übertragenen Preemptable-Frame unterbrechen. Erst mit dem Erhalt eines

Release-Requests wird die Fortsetzung und Übertragung von Preemptable-Frames wieder möglich. Bei der Unterbrechung müssen die geltenden Bedingungen für die Größe der Fragmente ebenfalls eingehalten werden, wodurch es auch hier möglich ist, dass der Preemptable-Frame nicht sofort oder gar nicht unterbrochen werden kann. Die MAC-Komponente berechnet die Worst-Case-Verzögerung dieses Falles, welche von ihr als sog. *Hold-Advance* bereitgestellt wird. In der Implementierung kann dem Hold-Request eine Zeitspanne angefügt werden, um die die Ausführung des Requests verzögert wird.

Hold-Requests werden vom Gate-Controller aufgerufen, der wie in Unterunterabschnitt 2.4.2.2 den Schedule für die Schaltung der Gates abarbeitet. Mit dem Hold- und Release-Mechanismus soll Express-Verkehr isoliert und vor der Verzögerung eines hereinragenden Preemptable-Frames bewahrt werden. Der Hold-Request muss um die Zeitspanne des Hold-Advances vor der Gate-Zustandsänderung erfolgen, um Preemptable-Frames noch rechtzeitig vor dem Wechsel abbrechen zu können. Wird der Hold erst zum Zeitpunkt der Gate-Zustandsänderung ausgeführt, hat dieser bei isolierten Gate-Öffnungen für Express-Traffic keinen spürbaren Effekt, da in das Zeitfenster hereinragende Frames auch durch Express-Frames aus dem dann geöffneten Gate unterbrochen würden.

Hierfür wird die Routine des Gate-Controllers um einen Lookahead erweitert, der nach Schalten des aktuellen Gate-Zustands für den darauffolgenden Zustand im Schedule prüft, ob darin ein Gate für Express-Traffic Verkehrsklasse geöffnet sein wird. Ist das der Fall, wird die Zeit berechnet, nach der der Hold-Request erfolgen muss und dieser in der MAC-Komponente mit der entsprechenden Verzögerung veranlasst. Die Zeit berechnet sich aus der Dauer des aktuellen Gate-Zustandes unter Abzug des Hold-Advances. Der Release-Request erfolgt, sobald kein Gate mit Express-Traffic mehr geöffnet ist.

Zusätzlich zur Reaktion auf den Hold-Request in der MAC-Komponente blockieren die Gates für Preemptable-Traffic während des Hold-Zustandes die Weiterleitung von Frames ihrer Queue. Dies hat die Wirkung, dass zur Transmission-Selection ausschließlich Express-Frames weitergeleitet werden und die MAC-Komponente zu sendende Frames über die herkömmliche Quelle anfordern kann. Das ist oben links in Abbildung 3.3 zu sehen, wo im Hold-Zustand der allgemeine Aufruf zum Anfordern eines neuen Frames benutzt werden kann, da die Transmission-Selection ohnehin nur Express-Frames liefern wird.

3.2 Simulationsautomatisierung

Eine weitere große Aufgabe der Arbeit war es, eine Automatisierung für die Ausführung von NeSTiNg-Simulationen zu entwerfen, welche die vorgestellten Scheduling-Verfahren benutzt. Folgende Anforderungen liegen der Simulationsautomatisierung zugrunde:

1. Automatisches Erstellen von Simulationen aus den Ergebnissen der drei Scheduler
2. Kombination der erstellten Netzwerke mit Best-Effort-Traffic in Form von TCP-Kommunikation zwischen zwei Hosts
3. Vergleichsmöglichkeit der drei Scheduler unter sonst gleichen Bedingungen durch Scheduling der gleichen Flows
4. Reproduzierbare Umgebungen und Skalierbarkeit durch Nutzung von Docker-Containern

In den folgenden Kapiteln wird erläutert, wie diese Anforderungen umgesetzt wurden.

3.2.1 Ablauf

Dieser Abschnitt gibt einen groben Überblick über die Abläufe der Simulationsautomatisierung, welche darauf folgend genauer beschrieben wird.

Abhängig von Eingabeparametern wird eine zufällige Problemstellung in Form einer Netzwerktopologie und einer Liste von Flows erzeugt und mit den drei in Abschnitt 2.3 beschriebenen Schedulern gelöst. Die Netzwerktopologie wird durch zufällig platzierte sog. *Best-Effort-Hosts*, die miteinander kommunizieren, erweitert und anschließend mit den erzeugten Schedules und Best-Effort-Traffic in OMNeT++ simuliert. Der Best-Effort-Traffic wird abgebildet, indem jeweils zwei Hosts über TCP miteinander kommunizieren, wobei über die TCP-Verbindung Datenmengen in einem Verhältnis gesendet werden, das auch in echten Netzwerken anzutreffen ist

3.2.2 Erstellen der Schedules

Für alle in Abschnitt 2.3 beschriebenen Verfahren standen der Arbeit Implementierungen in Python zur Verfügung. Die verwendeten ILP- bzw. SMT-Löser sind entweder als freie Software verfügbar (Z3 [Mic, „It is licensed under the MIT license.“]) oder können kostenlos für akademische Zwecke benutzt werden (CPLEX [Pug16]).

Die drei Scheduler unterscheiden sich teilweise stark im Format und vorhandenen Parametern. Um mit den Schedulern das gleiche Problem zu lösen bzw. die Ergebnisse der Scheduler einheitlich zu verwenden sind daher einige Konvertierungen notwendig, die im folgenden Abschnitt beschrieben werden. Tabelle 3.1 gibt einen Überblick über die Fähigkeiten und Formate der drei behandelten TSN-Scheduler.

3.2.2.1 Erstellen der Problemstellung

Sowohl der JSSP (Unterabschnitt 2.3.4), als auch der SMT-Scheduler (Unterabschnitt 2.3.5) beinhalten Skripte, um zufällige Graphen nach dem Erdős–Rényi-Modell zu erstellen. Der JRS Scheduler aus Unterabschnitt 2.3.3 besitzt darüber hinaus Optionen, um vier weitere Algorithmen zur Graph-Erzeugung zu nutzen und mit einigen Parametern zu konfigurieren. Aufgrund der vielseitigen Auswahl an Graphalgorithmen wurde das JRS-Projekt als Basis für die Problemerzeugung gewählt.

3.2.2.2 Konvertierung des Problems

Da das Scheduling-Problem bei jedem Scheduler in einem anderen Format ausgedrückt wird, musste das Skript `jrs_export.py` erstellt werden, das aus einem „JRS-Scheduling-Problem“ für die beiden anderen Scheduler eine äquivalente Problemstellung formuliert. Dabei müssen die Dateiformate geändert werden, aber je nach Scheduler auch Informationen entfernt oder weitere ergänzt werden.

Scheduler	JRS	JSSP	SMT
Flow-Periode einstellbar	✓	(Faktor)	(Faktor)
Flow Größe einstellbar	×	✓	
mehrere Frames pro Flow	✓	×	
Ende-zu-Ende Latenz	✓	×	✓
Processing Delay einstellbar	(in Per-Hop-Delay)	✓	
Transmission Delay	✓		
Queue einstellbar	✓(in `jrs_import.py`)	×, 7	×, ab 0
Flow-Datei	Textdatei	DAT	
Topologie-Dateiformat	GraphML	DAT	
Schedule-Dateiformat	XML, GraphML	XML	
Knoten-Unterscheidung	(nur Switche)	Hosts beginnen mit “h,,	
Routing	Teil des Problems	• Shortest-Path • ECMP • Length-Aware • Load-Length-Aware	Shortest-Path

Tabelle 3.1: Format- und Parametervergleich der drei Scheduler

JRS Format Der JRS-Scheduler betrachtet keine Hosts, sondern nur Switche in seinem Graph, da Scheduling erst ab diesem Punkt relevant wird. Flows in der Problemstellung starten und enden also jeweils an einem Switch. In den anderen Schedulern wird hingegen angenommen, dass Flows jeweils an Hosts starten und enden, weswegen das erstellte Export-Skript für die Scheduler noch einen Start- und Zielhost pro Flow generiert. Diese Hosts werden in der Topologie am Start- und Ziel-Switch eines Flows hinzugefügt. Ursprünglich wurde pro Switch nur ein Host erstellt, hiermit konnte aber die Situation zweier Flows, die den Switch als ersten oder letzten Hop nutzen, nicht abgedeckt werden, da auf der Ethernet-Verbindung zu dem platzierten Host nur ein Frame gleichzeitig gesendet werden kann.

JRS verwendet für Operationen auf Netzwerktopologien intern *graph-tool*, eine Python-Bibliothek für Graphen [Jon18]. Graphdaten wie die erstellte Netzwerk-Topologie liegen darum im *Graph Markup Language*-Format (GraphML), einem XML-basierten Dateiformat [BELP13], vor. Das in der Arbeit entwickelte Export-Skript liest die Topologie mit einem herkömmlichen XML-Parser ein, da die durch graph-tool bereitgestellten Funktionen für den Export nicht benötigt werden und die Bibliothek auf manchen Betriebssystemen aufwändig zu installieren ist.

Die Flow-Eingabe für den JRS-Scheduler ist eine Textdatei wie in Auflistung 3.2, die tabellarisch die ID der Flows, Start- und Zielhost sowie Periode, Reservierung und maximale Ende-zu-Ende-Latenz auflistet. Die drei zuletzt genannten Parameter sind abstrakte Zeitintervalle ohne Einheit, um von tatsächlichen Netzwerkgrößen zu abstrahieren [Jon18, S. 3]. Interpretiert man die Zeitangabe als Mikrosekunden, stimmen sie mit denen bei Gigabit-Ethernet überein. Die Reservierung eines Flows gibt an, wie viele Zeiteinheiten für das Transmission Delay der Übertragung durch Gate-Öffnungen reserviert werden sollen. Bei bekannter Zeiteinheit und Verbindungsgeschwindigkeit kann die Reservierung auf eine Frame-Größe in Bytes abgebildet werden. Für Gigabit-Ethernet werden pro Sekunde eine Milliarde Bits übertragen, also 1000 pro Mikrosekunde. Da ein Byte aus acht Bits besteht, erhält man 125 Bytes als die pro Mikrosekunde übertragene Datenmenge. Die Verzö-

```
# origin destination period reservation delay
0 1 0 5000 1000 5000
1 1 0 5000 1000 5000
2 0 2 5000 1000 5000
3 0 2 5000 1000 5000
4 2 0 5000 1000 5000
5 2 0 5000 1000 5000
6 1 0 5000 1000 5000
7 2 1 5000 1000 5000
```

Auflistung 3.2: `flowparams.table`, eine Problemstellung mit acht Flows für den JRS-Scheduler.

gerungen durch Queuing, Propagation, Transmission und Processing Delay aus Unterabschnitt 2.3.1 werden in [Jon18] zu einem *Per-Hop Delay* zusammengefasst. Das Per-Hop Delay beschreibt folglich die Zeit zwischen der beginnenden Übertragungen desselben Flows an den Ausgangsports zweier aufeinanderfolgender Switche.

Bursts Der JRS-Scheduler sieht vor, dass ein Flow mehrere Frames hintereinander in einem *Burst* schicken kann, d. h. die Reservierung kann auch aufgeteilt auf mehrere aneinandergereihte Frames genutzt werden. Diese Funktion wird vom JSSP-Scheduler nicht unterstützt, weswegen solche Problemstellungen auf mehrere Flows mit je einem Frame abgebildet werden. Das SMT-Scheduling unterstützt ebenfalls mehrere Frames, allerdings ist diese Funktion in der vorliegenden Implementierung nicht enthalten. Für die äquivalente Aufteilung wird dem Export-Skript als Option mitgeteilt, wie viele Bytes pro Zeiteinheit verschickt werden sollen und wie viele der Zeiteinheiten dem Transmission Delay eines Frames entsprechen. Die Aufteilung auf mehrere Flows ist jedoch nicht komplett mit einem Burst gleichzusetzen, da Zeitangaben nicht Teil der Eingabe sind und die Flows somit getrennt voneinander gescheduled werden können. Ebenfalls macht diese Aufteilung die Problemstellung für die JSSP und SMT-Scheduler unverhältnismäßig schwieriger, da im Endeffekt mehr Flows unter der gleichen Begrenzung untergebracht werden müssen.

```
('hs1_fl0', 'hs0_fl0', 1, 1000)
('hs1_fl1', 'hs0_fl1', 1, 1000)
('hs0_fl2', 'hs2_fl2', 1, 1000)
('hs0_fl3', 'hs2_fl3', 1, 1000)
('hs2_fl4', 'hs0_fl4', 1, 1000)
('hs2_fl5', 'hs0_fl5', 1, 1000)
('hs1_fl6', 'hs0_fl6', 1, 1000)
('hs2_fl7', 'hs1_fl7', 1, 1000)
```

Auflistung 3.3: `flows.dat`-Eingabedatei für den JSSP-Scheduler

Konvertierung für den JSSP-Scheduler Der JSSP-Scheduler benötigt zwei Eingabedateien für Flows und die Topologie im DAT Format, einer Textdatei mit Leerzeichen-getrennten Werten. Die *flows.dat*-Datei in Auflistung 3.3 enthält eine Liste von Flow-Tupeln bestehend aus Start-Host,

Ziel-Host, Periode und Frame-Größe. Die Periode wird im Gegensatz zu JRS als Faktor einer Basis-Periode interpretiert, welche zur Aufteilung zum Zeitpunkt der Konvertierung bekannt sein muss. Die *links.dat*-Datei ist eine Liste von Tupeln, die je eine gerichtete Verbindung zwischen zwei Hosts bzw. Switches angeben.

```
('hs1_fl0', 'hs0_fl0', 1, 1000, 5000)
('hs1_fl1', 'hs0_fl1', 1, 1000, 5000)
('hs0_fl2', 'hs2_fl2', 1, 1000, 5000)
('hs0_fl3', 'hs2_fl3', 1, 1000, 5000)
('hs2_fl4', 'hs0_fl4', 1, 1000, 5000)
('hs2_fl5', 'hs0_fl5', 1, 1000, 5000)
('hs1_fl6', 'hs0_fl6', 1, 1000, 5000)
('hs2_fl7', 'hs1_fl7', 1, 1000, 5000)
```

Auflistung 3.4: `flows.dat`-Eingabedatei für den SMT-Scheduler

Konvertierung für den SMT-Scheduler Der SMT-Scheduler erwartet in der *flows.dat*-Datei die gleichen Tupel wie JSSP, zuzüglich einer maximalen Ende-zu-Ende Latenz wie in Auflistung 3.4 abgebildet. Die *links.dat*-Datei muss im Vergleich zu JSSP um die Anzahl von TSN-Queues pro Verbindung erweitert werden, da Scheduling über mehrere Queues unterstützt wird. Ebenfalls muss das Ende-zu-Ende Delay um das zweifache Per-Hop-Delay erhöht werden, um die längeren Pfade durch hinzugefügte Hosts zu berücksichtigen.

3.2.2.3 Import der Scheduler-Ergebnisse

Die Ergebnisse der Scheduler liegen ebenfalls in verschiedenen Formaten oder Datenstrukturen vor und müssen in das gleiche, durch NeSTiNg verwendete Format gebracht werden.

SMT-Scheduler Der SMT-Scheduler und das NeSTiNg-Projekt sind bezüglich des Schedule-Formats schon aufeinander abgestimmt, d. h. der SMT-Scheduler exportiert XML Dateien für den Schedule und die Topologie, welche vom NeSTiNg-Generierungsskript, beschrieben in Unterabschnitt 3.2.3, gelesen werden. Das liegt daran, dass die initiale Entwicklung von NeSTiNg zur gleichen Zeit wie die des SMT-Schedulers stattfand und das Format entsprechend aufeinander abgestimmt wurde. Auflistung 3.5 zeigt einen beispielhaften Schedule dieses Formates mit einem Flow, der über zwei Switche geroutet wird.

JSSP-Scheduler Zum Zeitpunkt der Arbeit bestand die Ausgabe von JSSP aus einer Log-Datei, die unter anderem die Startzeitpunkte sowie Transmission- und Processing Delays der Flows enthielt. Das Projekt wurde darum um einen XML-Export mit der gleichen Struktur wie beim SMT-Scheduler erweitert, sodass die Dateien direkt vom Generierungsskript in NeSTiNg (siehe Unterabschnitt 3.2.3) verwendet werden können. Die direkte Ausgabe bot sich an, da die Zeiten intern schon so vorbereitet waren, wie sie im XML-Format in Auflistung 3.5 aufgeführt werden.

```
<schedule>
        <flow>
                <flow_src>hs1_fl1</flow_src>
                <flow_dst>hs0_fl1</flow_dst>
                <flow_period>5000</flow_period>
                <flow_length>1000</flow_length>
                <link>
                        <link_src>hs1_fl1</link_src>
                        <link_dst>s1</link_dst>
                        <offset>52</offset>
                        <queue>0</queue>
                        <end_offset>60</end_offset>
                </link>
                <link>
                        <link_src>s1</link_src>
                        <link_dst>s0</link_dst>
                        <offset>65</offset>
                        <queue>0</queue>
                        <end_offset>73</end_offset>
                </link>
                <link>
                        <link_src>s0</link_src>
                        <link_dst>hs0_fl1</link_dst>
                        <offset>78</offset>
                        <queue>0</queue>
                        <end_offset>86</end_offset>
                </link>
        </flow>
</schedule>
```

Auflistung 3.5: Flow-basierter Schedule, Ausgabe des JSSP- und SMT-Schedulers sowie des JRS Importierungsskripts, gekürzt

JRS-Scheduler Die Ergebnisse des JRS-Schedulers bestehen abgesehen von der in Abschnitt 3.2.2.2 beschriebenen Topologie aus einer XML-Datei, die den Pfad jedes Flows zeitlich wiedergibt. Für jede durchlaufene Verbindung werden die ID der Graph-ML Kante sowie Start- und Endzeiten aufgezeichnet. Zur Konvertierung der JRS-Ergebnisse in das übliche genutzte XML-Format wurde das Python-Skript `jrs_import.py` geschrieben, welches ohne die Abhängigkeit von graph-tool auskommt.

Für die akkurate Abbildung wird, wie auch in Abschnitt 3.2.2.2, die Information benötigt, wie viele Reservierungseinheiten auf einen Frame abzubilden sind.

Die Dateien werden durch `jrs_import.py` eingelesen, um Start- und End-Hosts erweitert und in der beim SMT-Scheduler verwendeten XML-Struktur wie in Auflistung 3.5 gespeichert. Diese Datei wird, wie im Folgenden beschrieben, von dem NeSTiNg-Generierungsskript verwendet.

3.2.3 Erstellen einer OMNeT++-Simulation

```
<schedule>
        <cycle>5000</cycle>
        <host name="hs0_fl2">
                <entry>
                        <start>13</start>
                        <queue>7</queue>
                        <dest>00:00:00:00:00:0e</dest>
                        <size>970</size>
                </entry>
        </host>
        <switch name="s0">
                <port id="0">
                        <entry>
                                <length>78</length>
                                <bitvector>11111110</bitvector>
                        </entry>
                        <entry>
                                <length>8</length>
                                <bitvector>00000001</bitvector>
                        </entry>
                        <entry>
                                <length>4914</length>
                                <bitvector>11111110</bitvector>
                        </entry>
                </port>
                <port id="1">
                        <entry>
                                <length>26</length>
                                <bitvector>11111110</bitvector>
                        </entry>
                        <entry>
                                <length>8</length>
                                <bitvector>00000001</bitvector>
                        </entry>
                        <entry>
                                <length>4966</length>
                                <bitvector>11111110</bitvector>
                        </entry>
                </port>
        </switch>
</schedule>
```

Auflistung 3.6: Port-basierter Schedule, Ausgabe des NeSTiNg-Generierungsskripts, Eingabe für NeSTiNg-Simulationen

Als Ausgabe der Scheduler liegt eine Topologie mit Switchen, Hosts und Verbindungen, sowie ein Flow-basierter Schedule vor. In ihm ist für jeden Flow der genaue Pfad sowie die Startzeit an jedem Hop verzeichnet. Damit sind alle Informationen gegeben, die für die Konfiguration einer NeSTiNg-Simulation (siehe Abschnitt 2.4) vonnöten sind. Der Schedule muss mit der Topologie kombiniert und den einzelnen Switch-Ports zugeordnet werden, sodass ein Port-basierter Schedule entsteht, der vom Gate-Controller, wie in Unterunterabschnitt 2.4.2.2 beschrieben, abgearbeitet werden kann. Auflistung 3.6 zeigt einen Auszug dieses Port-basierten Schedules.

Hierfür kommt ebenfalls ein Python Skript, `generateSimulation.py`, zum Tragen, welches aus den Flow-orientierten XML-Schedules wie Auflistung 3.5, einer XML Topologie und optional einer XML-Beschreibung von TCP-Hosts (siehe Unterunterabschnitt 3.2.3.1), eine vollständige NeSTiNg-Simulation erstellt und konfiguriert. Es werden automatisch MAC-Adressen vergeben, die Schedules für Hosts und Switche in XML exportiert und die Konfiguration für das MAC-basierte Forwarding in den Switchen erstellt. Dieses Generierungsskript war bereits vor Beginn dieser Arbeit Teil des NeSTiNg-Projekts und wurde währenddessen überarbeitet und um die Platzierung von TCP-Hosts, ansehnlichere Graphen und weitere Einstellungsmöglichkeiten ergänzt. Abbildung 2.1 zeigt eine Simulation von TSN-Flows und einem Paar von TCP-Hosts, die mit dem Skript erstellt wurde.

3.2.3.1 Platzieren der TCP-Hosts

Die TCP-Hosts werden paarweise – Server und Client – und zufällig in einer bestehenden Netzwerk-Topologie für TSN eingefügt. Das `place_tcp.py`-Skript wählt aus der Topologie zufällige und paarweise verschiedene Switche und platziert die TCP-Hosts daran. Das erste Paar wird dabei immer an dem ersten und letzten Switch platziert, um bei einer Linientopologie einen möglichst langen und durch anderen Verkehr benutzten Pfad durch das Netzwerk zu verwenden. So ist die Wahrscheinlichkeit höher, dass Effekte von TSN auf den Best-Effort-Traffic sichtbar werden, da es häufiger vorkommt, dass TSN-Traffic und Best-Effort-Traffic sich Bandbreite teilen müssen. Die Ausgabe ist eine XML-Beschreibung der Hosts incl. verwendeter TCP-Anwendung, Konfigurationsparametern und dem jeweils verbundenen Switch. Das NeSTiNg-Generierungsskript liest diese Datei, erstellt einen INET-Standardhost, ein NED-Modul mit üblichen Applikationen eines PCs, mit den jeweiligen Einstellungen und konfiguriert Ethernet-Verbindungen sowie Forwarding entsprechend.

Die ca. 20 Konfigurationsoptionen werden nicht als Argumente, sondern als Konfigurationsdatei übergeben. Dies ermöglicht das Erstellen von benannten TCP-Profilen. Beschränkt auf einen Ordner mit Konfigurationsdateien kann solch ein TCP-Profil durch den Namen außerdem eindeutig identifiziert werden. Beim händischen Vergleich zweier Simulationen genügt es den TCP-Profilnamen zu vergleichen, um herauszufinden, ob die Simulationen dieselben TCP-Einstellungen verwenden.

INET enthält mehrere TCP-Applikationen, von denen zwei Kombinationen vielversprechend für eine realistische Analyse sind:

TCP Session-Anwendung Die `TCPSessionApp` [Var14e] öffnet eine TCP-Verbindung und verschickt nach einer Pause eine konfigurierte Anzahl von Bytes an einen anderen Host. Als passender Server kommt hierbei die `TCPEchoApp` [Var14c] zum Einsatz, welcher die empfangenen Daten entgegennimmt und dem Client zurückschickt. Beim Server ist ebenfalls eine Pause und ein Faktor konfigurierbar, mit dem die Datengröße vor dem Zurücksenden multipliziert wird.

Standard TCP-Anwendung Die `TCPBasicClientApp` [Var14b] stellt zyklisch mehrere Anfragen an einen Server und pausiert in einstellbaren Intervallen. Es lassen sich Sende- und Empfangsgrößen, die Anzahl von Anfragen pro Session sowie die Pausen zwischen Anfragen und Sessions einstellen. Ursprünglich war `TCPGenericSrvApp` [Var14d] als Server-Komponente angedacht – diese Anwendung liest aus einem Attribut der ankommenden Pakete die gewünschte Bytegröße der Antwort und sendet ein entsprechendes Paket zurück. Leider war es nicht möglich, die Anwendungen in dieser Kombination funktionstüchtig zu bekommen, weswegen als Server-Anwendung ebenfalls auf `TCPEchoApp` [Var14c] zurückgegriffen wird. Diese Anwendung kann für statische Konfigurationen die gleiche Funktionalität erfüllen, indem ein passender Faktor für die Antwortgröße gewählt wird.

3.2.3.2 Erstellen von Graphen

Die Ergebnisdateien von OMNeT++-Simulationen können schon bei der Simulation von einigen Sekunden mehrere Gigabytes groß werden. Um die Ergebnisse der Simulation mit Python analysieren zu können, werden diese mit dem in Abschnitt 2.1.3.2 beschriebenen *scavetool* gefiltert und als CSV-Datei exportiert. Das Python-Skript `tcp_analyze.py` liest die CSV-Datei ein und exportiert mit Matplotlib [Mat] Graphen für verschiedene Kombinationen von Modulen und OMNeT++-Statistiken. In einer ersten Entwicklungsstufe wurde die CSV-Datei mit den Ergebnissen komprimiert und die Vektordatei gelöscht. Mittlerweile ist das Vorgehen umgekehrt, da durch Speichern der Vektor-Datei zu einem späteren Zeitpunkt auch vormals nicht gefilterte OMNeT++-Statistiken analysiert werden können, ohne die Simulation erneut auszuführen. Das Python-Skript erstellt außerdem selbst eine CSV-Datei, die eine Übersicht der Simulation in Form von Eingabeparametern und Durchschnittswerten der OMNeT++-Statistiken enthält.

3.2.4 Automatisierung einer Probleminstanz

3.2.4.1 Ausführung in Docker

Um eine reproduzierbare Umgebung zu haben, Simulationen unabhängig voneinander auszuführen und mehrere Server zur Ausführung nutzen zu können wird die Simulationsumgebung mit Docker-Containern virtualisiert. Das erstellte Docker Image enthält alle Pakete und Abhängigkeiten die zum Lösen der Schedules, zur Ausführung der Simulationen und zur Analyse notwendig sind. Es wurde ein existierendes Image für verwandte Arbeiten angepasst und um CPLEX, INET und Python-Module erweitert. Über ein Volume werden die aktuellen Versionen der Skripte und Scheduler eingebunden, damit der Docker Container nicht bei jeder Erweiterung der Projekte neu erstellt werden muss. Das Volume wird pro Instanz in einen separaten Ordner eingehängt, um Simulationen mit den gleichen Skripten, Repository-Versionen und Parametern erneut durchführen zu können. Da die Ergebnis-Dateien durch den Linux-Account innerhalb des Docker-Containers erstellt werden, sollte für diesen die gleiche UID festgelegt sein. Andernfalls fehlen dem Linux-Account des Docker-Hosts nach der Ausführung die Berechtigungen, um auf die Ergebnisse zuzugreifen.

3.2.4.2 Ausführung der Scheduler

Die Generierung, Simulation und Analyse einer Problemstellung mit allen Schedulern wurde in dem Shell-Skript `scheduler_to_omnet.sh` so zusammengefasst, dass dieses nur noch mit 12 Parametern aufgerufen werden muss und danach alle Skripte mit passenden Parametern und Pfaden aufruft.

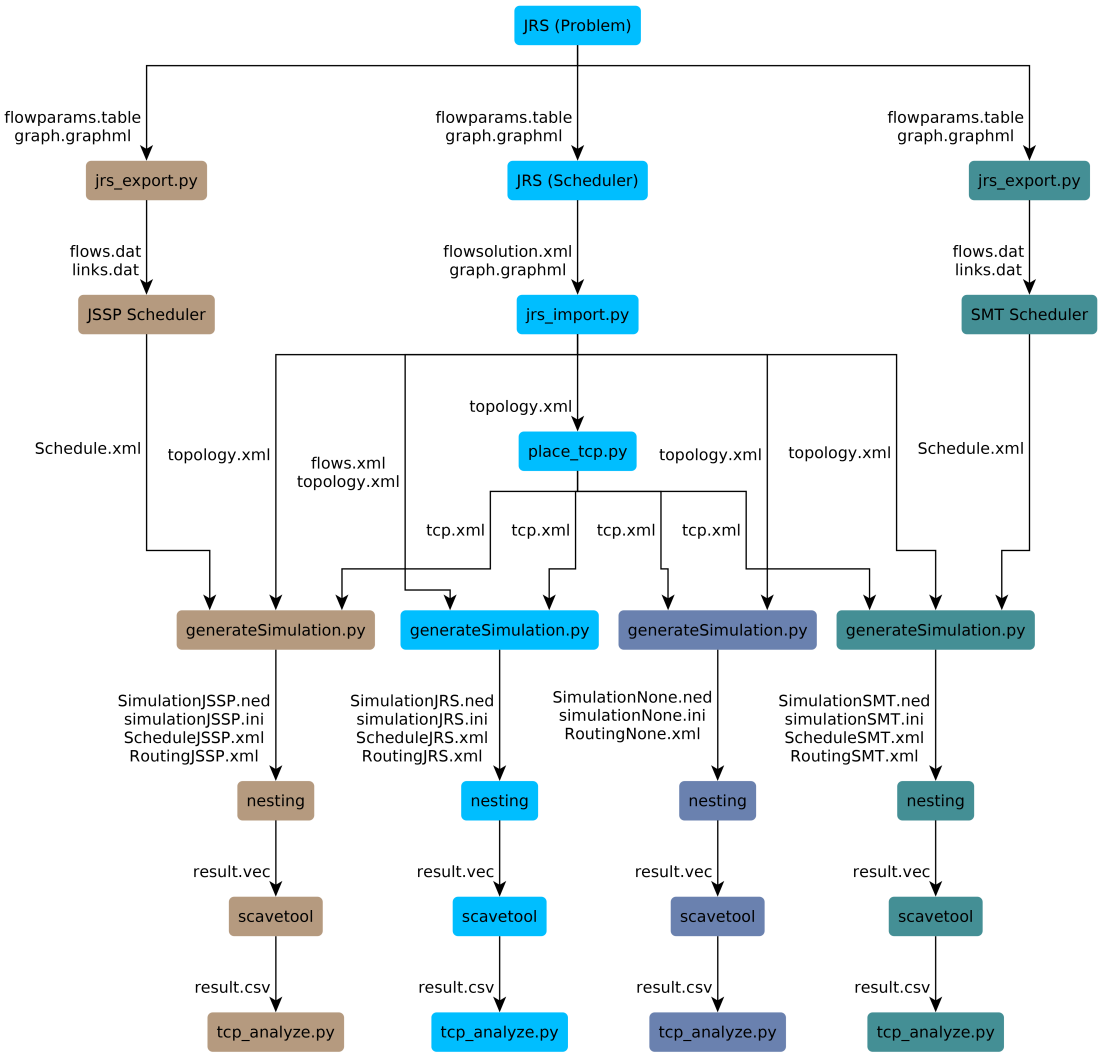


Abbildung 3.4: Struktur der automatisierten Simulation

Parameter Das Skript wird mit insgesamt 12 Parametern aufgerufen, welche teilweise noch weiteren Berechnungen unterliegen.

1. Anzahl der Switche
2. Anzahl der Flows
3. Processing Delay pro Switch [Mikrosekunden]

4. Anzahl von TCP-Session Paaren (siehe Abschnitt 3.2.3.1)
5. Anzahl von Standard TCP-Paaren (siehe Abschnitt 3.2.3.1)
6. Reservierung [Mikrosekunden] (siehe Unterunterabschnitt 3.2.2.2)
7. Schedule-Zykluszeit [Mikrosekunden]
8. Dauer des OMNeT++-Simulationslaufs als String mit Zeiteinheit (`s`, `ms`, `us`, etc.)
9. maximale Ende-zu-Ende Latenz [Mikrosekunden] (siehe Unterunterabschnitt 3.2.2.2)
10. zu sendende Bytes pro Reservierungseinheit (siehe Abschnitt 3.2.2.2)
11. Name der TCP-Konfigurationsdatei (siehe Unterunterabschnitt 3.2.3.1)
12. Queue-Größe der Switche [Anzahl von MTU-Frames]

Aus diesen Größen wird noch das Transmission Delay berechnet und der Processing Delay Wert für die TSN-Switche angepasst. Dieser wird um das Propagation Delay von 0,1 Mikrosekunden verkleinert, da diese Verzögerung in den Schedulern nicht berücksichtigt wird bzw. die Ergebnisse nur aus Ganzzahlen bestehen. Alle Projekte, Dateien und Ergebnisse werden in einem Ordner abgelegt, der nach den Parametern benannt ist, beispielsweise entsteht der Ordner `l3sw_5fl_5pd_0ts_1tb_20res_500B_150cy_50s_st_150_25B_a_10fr` beim Aufruf mit den Parametern `3 5 5 0 1 20 150 50s 150 25 a 10`.

Das NeSTiNg-Projekt, sowie die Scheduler werden jeweils in Git-Repositories verwaltet. Nach der Berechnung der Delays wird die aktuellste Version der Repositories geklont und NeSTiNg kompiliert.

Mit den gegebenen und berechneten Parametern wird dann durch den JRS-Scheduler eine Problemstellung erzeugt und wie in Unterunterabschnitt 3.2.2.2 für die beiden anderen Scheduler angepasst. Das Problem wird durch alle drei Scheduler gelöst und aus den Ergebnissen eine NeSTiNg-Simulation erzeugt. Für die Simulation wird die gleiche Topologie verwendet und durch das TCP-Profil und die Anzahl der TCP-Paare entsprechende TCP-Hosts platziert. Als Baseline wird noch eine weitere Simulation mit der Topologie und den TCP-Hosts, aber ohne aktiviertes Scheduling erzeugt. Die vier erstellten Simulation werden nacheinander ausgeführt und die Ergebnisse mit `tcp_analyze.py` verarbeitet. Im Anschluss werden nicht benötigte Zwischenergebnisse gelöscht, und die Simulationsumgebung sowie die Ergebnisse der Simulationen separat komprimiert.

Abbildung 3.4 zeigt, mit welchen Abhängigkeiten Skripte innerhalb von `scheduler_to_omnet.sh` aufgerufen werden. Knoten stellen Skripte oder aufgerufene Anwendungen wie die Scheduler dar, während Kanten aufzeigen, woher die Eingabedateien für weitere Aufrufe stammen. Manche Ergebnisse werden nur für einen Aufruf benötigt, während etwa die Topologie und darin platzierte TCP-Hosts mehrfach verwendet werden.

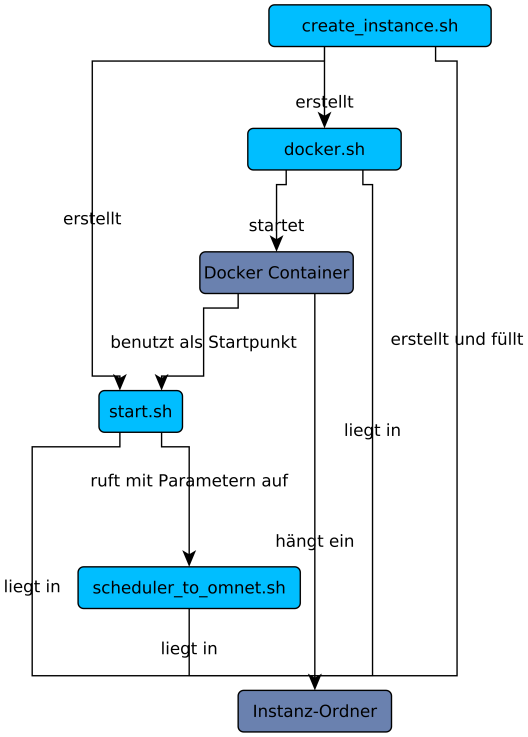


Abbildung 3.5: Struktur zum Starten der Skripte mit einem Docker Container

3.2.4.3 Automatisierung einer Instanz

Das `create_instance.sh`-Skript wurde erstellt, um eine Problem-Instanz unkompliziert in einem Docker Container auszuführen. Abbildung 3.5 zeigt Ressourcen, den strukturellen Ablauf und eine Einordnung des Skripts. Das `create_instance.sh`-Skript wird mit den gleichen Parametern wie `scheduler_to_omnet.sh` aufgerufen und erstellt nach dem gleichen Namensschema einen Ordner als Grundlage (unten in Abbildung 3.5). In diesen Ordner wird das Git-Repository mit `scheduler_to_omnet.sh` und weiteren Skripten geklont und zwei weitere Skripte erzeugt:

`start.sh` ist ein Skript, das `scheduler_to_omnet.sh` mit den gleichen Parametern aufruft und somit die Instanz ausführt.

`docker.sh` startet einen Docker-Container mit dem Instanz-Verzeichnis als Volume (eingehängter Mountpoint) und `start.sh` als Entrypoint (Startskript).

Um die Ausführung vieler Instanzen auf einmal zu vereinfachen, ruft `create_instance.sh` das `start.sh` auf, nachdem es erstellt worden ist. Auf diese Weise wurde die gesamte Ausführung einer Instanz mit mehreren zusammengehörenden Simulationen auf einen einzigen Befehl reduziert. Durch die Aufteilung auf einzelne Skripte ist aber weiterhin eine Granularität gegeben, beispielsweise um eine Simulation auch ohne Docker auszuführen oder die Analyse von Ergebnisdateien manuell zu wiederholen.

4 Durchführung und Analyse

Die beschriebene Architektur für die Simulationsautomatisierung wurde verwendet, um Effekte von Time-Sensitive Networking auf Best-Effort-Traffic zu untersuchen. Das folgende Kapitel beschreibt Rahmenbedingungen der Simulationen sowie gesetzte und gemessene Parameter für automatisierte Simulationen. Im Anschluss wird auf Ergebnisse der durchgeführten Simulationen eingegangen.

4.1 Rahmenbedingungen

In diesem Abschnitt wird die Ausführungsumgebung für die Simulationen und die Analyse beschrieben sowie erläutert, warum die Analyse inkrementell und parallel zur Entwicklung der Skripte stattfand.

4.1.1 Inkrementeller Entwicklungsprozess

Die Entwicklung der Simulationsskripte fand zeitlich und inhaltlich parallel zur Ausführung von Simulationen und der Analyse statt. Das liegt einerseits an der inkrementellen Integration der TSN-Scheduler und Simulationen, andererseits an der engen Verzahnung der Skripte mit dem Analyseprozess. Manche der Scheduler-Implementierungen wurden in dieser Arbeit erstmals für größere Simulationen genutzt und erforderten beispielsweise Anpassungen, um Netzwerk-Delays als Parameter setzen zu können. Die Integration der neu entwickelten Skripte zeigte ebenfalls Fehler auf, die erst beim Zusammenfügen der Skripte und Komponenten oder beim Aufruf mit bestimmten Parametern auftraten, beispielsweise dass die Scheduler verschiedene Auffassungen der Queue-Reihenfolge haben. Zusätzlich dazu stellten sich während der laufenden Ausführung von Simulationen und der Betrachtung der Ergebnisse weitere einzufügende Stellgrößen und relevante Messgrößen heraus, die integriert werden mussten. Tabelle 4.1 listet die Versionen der verwendeten Programme auf, die für die Ausführung und Analyse der Simulationen sowie für das Scheduling verwendet wurden.

Die Ausführungsdauer einer Instanz ist abhängig von der Netzwerk- und Problemgröße sowie der TCP-Konfiguration, da sich zum einen die reine Simulationsdauer vergrößert und zum anderen die TSN-Scheduler eine lange Zeit in Anspruch nehmen können, um eine Lösung zu finden. Welche Messgrößen für die Analyse gewählt wurden, trägt ebenfalls zur Dauer der Ausführung bei, da die Dauer zum Erstellen von Plots von der Anzahl und Komplexität beeinflusst wird. Mit Docker können mehrere Instanzen parallel ausgeführt werden, allerdings wird eine einzelne betrachtete Simulation hierdurch nicht beschleunigt. Dies hat zur Folge, dass Änderungen an den Skripten oder der Struktur der Simulationen erst nach der vollständigen Ausführung einer Instanz validiert werden können. Da die Ausführung größerer Instanzen mehr als 12 Stunden dauern kann, ist eine parallele Entwicklung und Analyse unabdingbar. Ebenfalls kann erst nach dieser Zeit begonnen

Programm	Version
OMNeT++	5.2.1
INET	3.6.3
Docker	18.05.0-ce (Server), 1.13.1 (Virtuelle Maschine)
CPLEX	12.8
Z3	4.6.1
Python	3.6.5
graph-tool	2.26-8.1
matplotlib	2.2.2

Tabelle 4.1: Für die Simulationen, Scheduler und Evaluation verwendete Programmversionen

werden, die Ergebnisse zu analysieren, um die Auswirkungen geänderter Parameter zu überprüfen. Besonders problematisch ist es, wenn ein gravierender Fehler erst nach mehreren Tagen entdeckt wird, da somit alle in dieser Zeit ausgeführten Simulationen unbrauchbar sind.

4.1.2 Ausführungsumgebung

Wie bereits in Unterunterabschnitt 3.2.4.1 erwähnt, ist mit Docker die Ausführung der Versuche auf mehreren Linux-Rechnern möglich. Dem Autor stand eine virtuelle Maschine mit 16 Kernen sowie ein Server mit 64 Kernen und fünf Terabyte Speicherplatz zur Verfügung, um die Simulationen durchzuführen. Die Nutzung mehrerer Server erhöht die Parallelisierung von Instanzen, allerdings geht mit der Verteilung ein erhöhter Aufwand einher, um Code und Parameter zu synchronisieren sowie einen Überblick über die ausgeführten Instanzen und Ergebnisse zu behalten.

Es kam ein separates Git-Repository zum Einsatz, um Anpassungen an den Skripten zu versionieren und den aktuellen Softwarestand leicht zwischen den beiden Rechnern synchronisieren zu können. In diesem sog. *TSN-Repository* wurden das Haupt-Skript `scheduler_to_omnet.sh` aus Unterunterabschnitt 3.2.4.2, das Analyse-Skript `tcp_analyze.py` aus Unterunterabschnitt 3.2.3.2, sowie die Skripte zur Docker-Automatisierung aus Unterunterabschnitt 3.2.4.3 entwickelt. Des Weiteren wurden dem TSN-Repository regelmäßig Skripte zum Ausführen mehrerer ähnlicher Instanzen hinzugefügt, ebenfalls zur Versionierung und Synchronisation.

Auflistung 4.1 zeigt einen Ausschnitt des Arbeitsverzeichnisses, das zur Strukturierung der Simulationen und Ergebnisse verwendet wird. Für jedes Set von Instanzen wird im Arbeitsverzeichnis ein Ordner mit dem Datum und der dazu relativen Nummer der Ausführung erstellt, aber nicht dem TSN-Repository hinzugefügt. Anschließend wird in diesen neuen Ordner ebenfalls das TSN-Repository geklont und darin das Ausführungsskript für das jeweilige Set aufgerufen. Dieses klont für jede Instanz das Repository und führt die Skripte, wie in Unterunterabschnitt 3.2.4.3 beschrieben, aus.

Vollständig ausgeführte Simulationen wurden als komplette Ordner auf den größeren Server verschoben, um dort einen Überblick über die Ausführungen zu haben und gesammelt darauf zugreifen zu können. Weiterhin war das Verschieben notwendig, um auf der virtuellen Maschine genug Speicherplatz für neue Simulationen zu haben, da manche größere Instanzen alleine über 30GiB

```
tsn_repository ..... (TSN-Repository zur Skript-Bearbeitung)
├── create_instance.sh
├── scheduler_to_omnet.sh
├── tcp_analyze.py
├── set_20180725_5.sh
└── 20180725_5 ..... (TSN-Repository für Instanzen-Set)
    ├── create_instance.sh
    ├── scheduler_to_omnet.sh
    ├── tcp_analyze.py
    ├── set_20180725_5.sh
    └── l3sw_20fl_5pd_0ts_1tb_1000res_1000B_10000cy_50s_st_10000_1B_a_30fr (TSN-Repository
        zur Ausführung)
        ├── create_instance.sh
        ├── scheduler_to_omnet.sh
        ├── tcp_analyze.py
        ├── set_20180725_5.sh
        ├── start.sh
        ├── docker.sh
        ├── log.log
        ├── nesting ..... (NeSTiNg-Repository)
        ├── jrs ..... (JRS-Repository)
        │   └── l3sw_20fl_5pd_0ts_1tb_1000res_1000B_10000cy_50s_st_10000_1B_a_30fr_jrs
        ├── jssp ..... (JSSP-Repository)
        │   └── l3sw_20fl_5pd_0ts_1tb_1000res_1000B_10000cy_50s_st_10000_1B_a_30fr_jssp
        ├── smt ..... (SMT-Repository)
        │   └── l3sw_20fl_5pd_0ts_1tb_1000res_1000B_10000cy_50s_st_10000_1B_a_30fr_smt
        ├── none
        │   └── l3sw_20fl_5pd_0ts_1tb_1000res_1000B_10000cy_50s_st_10000_1B_a_30fr_none
        └── plots
            ├── jrs_all.pdf
            ├── jrs_clients.pdf
            ├── jrs_switches.pdf
            ├── jrs_s0.pdf
            ├── jssp_all.pdf
            ├── smt_all.pdf
            ├── none_all.pdf
            ├── …
            ├── stats.csv
            └── raw.csv
```

Auflistung 4.1: Arbeitsverzeichnis der Simulationsumgebung

Statistik	Komponente	Typ
Anzahl aktiver TCP-Sessions	TCP-Anwendung	Vektor
Empfangene Paketgröße in Bytes	TCP-Anwendung	Vektor
Angebotenes TCP-Fenster	TCP	Vektor
Empfangene Acknowledgements	TCP	Vektor
Empfangene Sequenznummer	TCP	Vektor
Gesendete Acknowledgements	TCP	Vektor
Gesendete Sequenznummer	TCP	Vektor
Unbestätigte Bytes	TCP	Vektor
Round-Trip-Time	TCP	Vektor
Durchschnittliche Queue-Länge in Frames	Queue	Skalar
Maximale Queue-Länge in Frames	Queue	Skalar
Queueing-Zeit eines Frames	Queue	Vektor
Eingeordnete Paketgröße in Bytes	Queue	Vektor
Verworfene Pakete (Anzahl)	Queue	Skalar
Verworfene Pakete (Summe der Bytes)	Queue	Skalar
Gesendete Frames pro Sekunde	MAC	Skalar
Empfangene Frames pro Sekunde	MAC	Skalar
Empfangskanal-Nutzung (%)	MAC	Skalar

Tabelle 4.2: Betrachtete OMNeT++-Statistiken der MAC- Queue-,TCP-Komponente sowie der TCP-Anwendung

Speicherplatz belegen. Es kam das Tool `sshfs` [Nik] zum Einsatz, zum einen um Daten zwischen den beiden Servern zu verschieben, zum anderen um die Verzeichnisse aus der Ferne mit einem lokalen Linux-PC einzuhängen.

4.2 Analyse

Dieser Abschnitt beschäftigt sich mit der inhaltlichen Durchführung der Analyse und geht auf gesetzte Parameter und verfügbare Messgrößen ein.

4.2.1 OMNeT++-Statistiken

Wie bereits in Abschnitt 2.1.3.2 erläutert, wird in OMNeT++ zwischen Vektor-Statistiken und Skalar-Statistiken unterschieden. Vektor-Statistiken zeichnen Messwerte mit Zeitstempeln auf, damit können hinterher Verläufe beispielsweise als Plot dargestellt werden. Das kann allerdings zu einem hohen Speicherverbrauch führen, beispielsweise 30GiB für eine Simulation mit 30 Geräten bei 100 Sekunden Simulationszeit. Eine Skalar-Statistik besteht hingegen nur aus einem Wert, der über die gesamte Simulationszeit aggregiert wird, beispielsweise die Nutzungshäufigkeit eines Übertragungskanals. Tabelle 4.2 zeigt die Statistiken, die in den Simulationen aufgezeichnet und zur Analyse betrachtet wurden. Die Statistiken werden an mehreren Stellen durch die jeweilige

Komponente erfasst, beispielsweise die Queue-Länge durch die Queue-Komponente. Es wurden Statistiken entlang der Pfade von TSN-Traffic und Best-Effort-Traffic gewählt, insbesondere an den Stellen, die gemeinsam durchlaufen werden und so potentiell gegenseitige Auswirkungen aufzeigen können. Relevant sind hierbei Statistiken, die Geschwindigkeiten und Durchsatz messen, sowie auf die Anstauung von Frames hinweisen.

Die Statistiken der TCP-Anwendungen und TCP-Komponenten wurden in Gänze aufgezeichnet, da auch alle Verwendungen der Komponenten von Interesse sind. Es werden dagegen aber nur ausgewählte MAC- und Queue-Komponenten behalten und betrachtet, da die Komponenten auch an Stellen instantiiert sind, die für die Analyse nicht von Interesse sind. Aus der Netzwerktopologie wird importiert, welche MAC-Komponenten eines Switches mit anderen Switchen verbunden sind und nur diese sowie die MAC-Komponenten der TCP-Hosts erfasst. Dazu zählen z.B. Queues, die weder Best-Effort- noch TSN-Traffic enthalten und MAC-Komponenten, die höchstens von einer Verbindung durchlaufen werden. Somit wird ein Plot mit beispielsweise allen Queue-Längen eines Switches durch die nicht benötigten Werte nicht unübersichtlich.

Das Skript `tcp_analyze.py` aus Unterunterabschnitt 3.2.3.2 erstellt getrennt nach Schedulern und Komponenten einzelne Dateien mit Plots. Zusätzlich zu einer Übersicht mit allen Komponenten werden für TCP-Server, TCP-Client und Switch jeweils separate Dateien generiert. Die Menge der analysierten Statistiken und Arten der Verwertung wurde iterativ angepasst. In der ursprünglichen Implementierung von `tcp_analyze.py` wurden ausschließlich die Vektor-Statistiken der TCP-Module aufgezeichnet und damit zeitliche Verläufe geplottet. Das Skript wurde im Laufe der Analyse dahingehend erweitert, dass Minimum, Maximum und Durchschnitt der Vektoren berechnet und als CSV exportiert wurden. Damit lassen sich automatisiert Plots mit den Gesamtergebnissen mehrerer Simulationen erstellen und diese sich so vergleichen. Dies ermöglicht, zusätzlich zur Auswertung der Plots einer einzelnen Simulation, eine kombinierte Analyse, die Trends über verschiedene Simulationen hinweg aufzeigen kann.

Diese Zusammenfassung wird als CSV-Datei exportiert und ebenfalls mit Python und Matplotlib [Mat] verwertet. Die Entwicklung und Ausführung hierzu fand in einer interaktiven Python-Konsole „Jupyter Notebook“ [IPy] statt, in der Plots direkt dargestellt und modifiziert werden können. Die einzelnen CSV-Dateien werden mit Linux-Tools wie `tree` und `grep` rekursiv im Arbeitsverzeichnis gesucht und zur Analyse zusammengefügt.

4.2.2 Parameter und Größen

Wie in Abschnitt 3.2.4.2 erläutert, werden die Einstellungen und das Verhalten einer Simulationsinstanz letztendlich über zwölf Parameter getroffen. Aus den Parametern wird der Name der Simulationen abgeleitet, wodurch diese unterschieden werden können und von einem Namen direkt auf die verwendeten Parameter geschlossen werden kann. Tabelle 4.3 zeigt die erforderlichen Eingabegrößen sowie die simulierten Wertebereiche.

Das Processing Delay und die Simulationszeit wurden über fast alle Simulationen hinweg auf einem Wert belassen – 5μs Processing Delay bzw. 50s Simulationszeit. Der Wert von 5μs liegt etwa 0,5μs über der nach [DK+14] üblichen Verzögerung für Gigabit-Ethernet Switche. Diese Größenordnung ist für TSN-Switche nicht unwahrscheinlich, da Frames dort detaillierter behandelt werden müssen, beispielsweise um sie in die richtige Verkehrsklasse einzuordnen. Die Simulationszeit, also die

Dauer der simulierten Abläufe im Netzwerk, wurde auf 50 Sekunden festgelegt, da in der Zeitspanne mehrere Durchläufe der TCP-Kommunikationen durchgeführt werden können und sich die Durchführungszeit der Simulation in Grenzen hält.

Für die Abbildung von Best-Effort-Traffic wurden, wie in Unterunterabschnitt 3.2.3.1 erläutert, Paare von TCP-Hosts zufällig im Netzwerk platziert. Ein Paar von TCP-Hosts besteht aus einem Client und einem Server, die miteinander kommunizieren und außer dieser Kommunikation nichts versenden. Es wurden insgesamt fünf verschiedene TCP-Konfigurationen erstellt, die das Verhalten beider Paare definieren. Die Konfigurationsparameter sind in Tabelle 4.4 vollständig aufgelistet und unterscheiden sich hauptsächlich in der gesendeten Datengröße sowie der Häufigkeit der Verbindungen. Ursprünglich wurde die Anzahl der TCP-Paare beider Arten variiert und kombiniert, allerdings ließ sich so nicht komplett zuordnen, ob Verzögerungen etc. durch TSN-Traffic oder durch andere TCP-Kommunikationen verursacht werden. Seitdem wird nur jeweils ein Paar von TCP-Hosts platziert und das Verhältnis der Verkehrsarten im Netzwerk über die Menge der TSN-Flows reguliert.

Die Netzwerkgröße lässt sich einfach über die Anzahl der Switche regulieren, die in einer Linientopologie angeordnet sind und wurde meist auf drei bis fünf Switche festgesetzt und darüber hinaus nur testweise verändert. Vormals wurden auch vermaschte Netzwerktopologien verwendet, dabei kann es aber durch das nichtdeterministische Routing beider Verkehrsarten zu weniger gemeinsam durchlaufenen Pfaden kommen, die allerdings untersucht werden sollen. Durch die Erhöhung der Flows oder TCP-Paare kann dem entgegengewirkt werden, aber da nur begrenzt große Scheduling-Probleme gelöst werden können verschiebt sich damit der untersuchbare Bereich. Weiterhin kam es bei dem vorherigen Topologie-Modell häufig vor, dass nur der JRS-Scheduler das Problem lösen konnte, da dieser das Routing in den Scheduling-Prozess integriert, anstatt für jeden Flow eine feste Route zu definieren und die Flows nur zeitlich darin zu platzieren.

Neben der Netzwerkgröße fand die meiste Variation der Eingabeparameter bei den Einstellungen statt, die TSN-Flows betreffen. Das sind die Anzahl der Flows, die Zykluszeit, die Länge der Reservierung sowie die daraus gebildete Framegröße.

Die größte Variation der Parameter fand über die Flow-Anzahl sowie den Anteil der Reservierung für TSN-Traffic, über die Zykluszeit, die Reservierung und die Framegröße pro Reservierungseinheit statt. Es wurde meist eine Flow-Anzahl zwischen 10 und 20 gewählt, da selbst in kleinen Netzwerken erst in diesem Bereich Veränderungen auftreten und sich diese Flow-Anzahl noch in annehmbarer Zeit schedulen lässt. Jeder dieser Flows muss innerhalb des Zyklus seinen Pfad reserviert und garantiert durchlaufen können. Die Framegröße eines Flows berechnet sich als Multiplikation der Reservierung und der Framegröße pro Reservierung. Somit kann der Belegungsanteil eines Flows im Zyklus über die Zyklusgröße, Reservierung und Framegröße pro Reservierung reguliert werden. Weiter kann die Belegung mit der Anzahl der Flows skaliert werden. Zu beachten ist allerdings, dass eine erhöhte Flow-Anzahl einen Pfad nicht zwingend stärker belegt. Das liegt an der zufälligen Platzierung der Flows, die in der Linientopologie in zwei Richtungen zugewiesen werden können und auch nur einen Teil der Switche durchlaufen können.

Die Reservierungsgröße beschreibt nur bei Verwendung des JRS-Schedulers direkt den zu reservierenden Bereich. Bei den anderen beiden Schedulern wird nur der tatsächlich für das Versenden eines Frames benötigte Zeitraum reserviert.

Position	Parameter	Wertebereich
1	Anzahl Switche	2–20
2	Anzahl Flows	2–30
3	Processing Delay	1–50μs
4	Anzahl von TCP-Session-Paaren	0–20
5	Anzahl von TCP-Standard-Paaren	0–20
6	Reservierung	1–1500μs
7	Zykluszeit	40-100000μs
8	Simulationszeit	50–100s
9	Maximale Ende-zu-Ende Verzögerung	40–100000μs
10	Framegröße pro μs Reservierung	1–125B
11	TCP-Konfiguration	a–e (Tabelle 4.4)
12	Queue-Länge	1–100×1500B

Tabelle 4.3: Wertebereich der Eingabeparameter des Ausführungsskripts (siehe Abschnitt 3.2.4.2)

Möchte man zwei Simulationen, eine mit einem kurzen und eine mit einem langen Zyklus, ausführen und die Schedules zu gleichen Anteilen für TSN reservieren, stellt das ein Problem dar. Der gewünschte Reservierungsanteil könnte entweder durch längere Flows, oder durch mehr Flows erreicht werden. Wegen der Limitierung auf einen Frame pro Flow, die in Abschnitt 3.2.2.2 beschrieben ist und durch die MTU von Ethernet können Flows in diesem Umfeld nicht beliebig vergrößert werden. Die Flow-Anzahl kann durch die Komplexität des Scheduling-Problems ebenfalls nicht beliebig weit erhöht werden. Eine Möglichkeit, den gleichen Reservierungsanteil abzubilden, ist es, bei dem JRS-Scheduler mehr Zeiteinheiten zu reservieren als für die Übertragung des Frames benötigt werden. Der Gate-Controller setzt die im Schedule hinterlegte Reservierung um, unabhängig der tatsächlich verwendeten Zeit. Somit kann beispielsweise ein Schedule mit einer Zykluszeit von 10000μs und einem Flow mit 1000μs×1 Byte erstellt werden, der auf einer Verbindung das gleiche Belegungsverhältnis wie 10×125Bytes bei einer Zykluszeit von 100μs aufweist. Mit dieser Methode können zumindest für den JRS-Scheduler Schedules einer größeren Größenordnung simuliert werden.

Die genannten Parameter wurden einerseits in Kombination angepasst, um Einschränkungen der Service-Qualität von Best-Effort-Traffic verstärkt zu beschränken, als dies mit nur einzelnen Parametern notwendig ist. Damit lassen sich Effekte allerdings keiner bestimmten Parameteränderung zuordnen. Aus diesem Grund wurde danach die Strategie verfolgt, eine Grundbelegung für die Parameter zu wählen und davon mehrere Variationen abzuleiten, bei denen immer der selbe Parameter adaptiert wird. Mit dieser Methode lassen sich kombinierte Plots erzeugen, die zuordenbar eine Korrelation zwischen verändertem Parameter und veränderter Messgröße aufzeigen.

Es wurden 44 Sets von Instanzen ausgeführt, die nach diesem Schema erzeugt wurden. Hierbei waren in jedem Set zwischen 6 und 12 Parameteränderungen enthalten, damit die Sets auf beiden verfügbaren Linux-Servern ausgeführt werden konnten, ohne diese zu überlasten. Wenn mehr als 12 Änderungen eines Parameters von Interesse waren, wurden die Instanzen auf mehrere Sets aufgeteilt und die Ergebnisse im Anschluss kombiniert bewertet.

Im Folgenden werden die Ergebnisse dieser Simulationen beschrieben.

Parameter	TCP-Profil				
	a	**b**	**c**	**d**	**e**
TCP Session-Anwendung (Abschnitt 3.2.3.1)					
Verbindungsstart	1s				
Verbindungsende	200s				
Sendezeit	5s				
Sendegröße	9GiB	1GiB			
Multiplikator der Antwortgröße	2.0	10.0			
Antwort-Verzögerung	0				
TCP Standard-Anwendung (Abschnitt 3.2.3.1)					
Verbindungsstart	1s				
Verbindungsende	(nie)				
Anfragen pro Session	20		100	10	
Sendegröße	400B	50KiB	500B	10MiB	1MiB
Antwortgröße	500KiB	10MiB	1MiB	500MiB	50MiB
Denkzeit zwischen Anfragen	1s	0.3s			
Wartezeit zwischen Sessions	5s				
Timeout	10s				
Echo-Faktor der Antwort	1280	205	2098	50	
Antwort-Verzögerung	0.01s				

Tabelle 4.4: Konfiguration verschiedener TCP-Profile als Parameter für die INET Anwendungen.

4.3 Ergebnisse

4.3.1 Erwartungen

Vor der Ausführung wurde überlegt, welchen Einfluss bestimmte Parameter haben könnten. Erwartet wurde, dass der Best-Effort-Traffic durch den TSN-Traffic verzögert und beeinträchtigt wird. Hierbei dürfte es auf die Menge und Länge der reservierten Zeitbereiche im Schedule ankommen, da in diesen kein Best-Effort-Traffic versendet werden kann. Diese hängen wiederum von der gewählten Anzahl an Flows und der Reservierung sowie Länge eines Flows ab.

Die gewählte Zykluszeit kann ebenfalls von Bedeutung sein, da es zu Verzögerungen in einer anderen Größenordnung kommen kann. Vergleicht man zwei Schedules mit einer Zyklusdauer von 100μs und 1000μs, die jeweils in der ersten Hälfte für TSN-Traffic reserviert sind, dann ist die maximale Wartezeit für Best-Effort-Traffic beim längeren Schedule um Faktor 10 größer, obwohl die Bandbreite über lange Sicht im gleichen Verhältnis aufgeteilt ist. Das kann sich insbesondere dann auswirken, wenn die Queues mit einer geringen Puffergröße konfiguriert sind. Best-Effort-Frames, die sich über längere Zeit ansammeln können, würden die Queues an ihr Limit bringen, sodass diese enthaltene Frames verwerfen müssen. Da die Topologien und Hosts alle zufällig erzeugt werden, ist die Netzwerkgröße ebenfalls ein wichtiger Parameter. Je kleiner das Netzwerk ist, desto weniger verschiedene Pfade existieren und können für TSN- oder Best-Effort-Verkehr zufällig ausgewählt werden. Dadurch erhöht sich die Wahrscheinlichkeit, dass die Frames beide über die gleiche Verbindung geroutet werden bzw. die Bandbreite der Verbindung stärker genutzt wird.

4.3.2 Frame-Rate

Bei der Analyse wurde festgestellt, dass sich die Anzahl der Flows auf die Frame-Rate der MAC-Schnittstellen auswirkt.

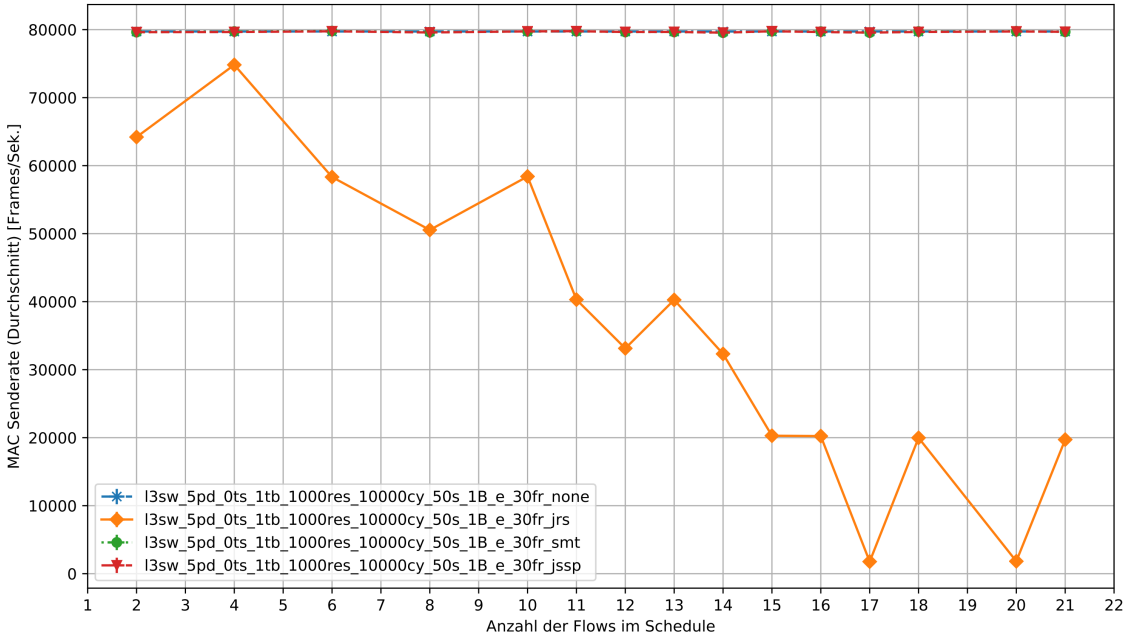


Abbildung 4.1: Frame-Rate ausgewählter MAC-Schnittstellen bei variierender Flow-Anzahl mit sonst identischen Parametern

Abbildung 4.1 zeigt den Einfluss einer variierenden Flow-Anzahl auf die Frame-Senderate ausgewählter MAC-Schnittstellen. In Unterabschnitt 4.2.1 wurde bereits erläutert, dass nur die Statistiken jener MAC-Schnittstellen aufgezeichnet wurde, durch die Best-Effort-Traffic verläuft. Der Plot zeigt die durchschnittliche Senderate der MAC-Schnittstellen für eine variierende Flow-Anzahl mit sonst gleichen Eingaben. Es kam eine Linientopologie mit drei Switchen zum Einsatz, sowie eine Zykluszeit von 10000μs, bei denen ein Flow immer 1000μs für seine Übertragung reserviert.

In orange ist die Senderate für Simulationen mit dem JRS-Scheduler zu sehen, die mit steigender Flow-Anzahl abfällt. Bei 17 bzw. 20 Flows kommt sogar jeglicher TCP-Traffic zum erliegen – die vorhandene Senderate ist an diesen Stelle ausschließlich auf TSN-Traffic an den gemeinsam genutzten Ports zurückzuführen. Dass es Instanzen mit noch höherer Flow-Anzahl gibt, für die wieder Best-Effort-Traffic gesendet werden kann, liegt an der zufälligen Platzierung der Flows. Es wurde bereits erläutert, dass Flows in verschiedenen Richtungen und über verschiedene Switche platziert sein können, wodurch eine steigende Anzahl von Flows keinen höheren Reservierungsanteil einer bestimmten Verbindung impliziert.

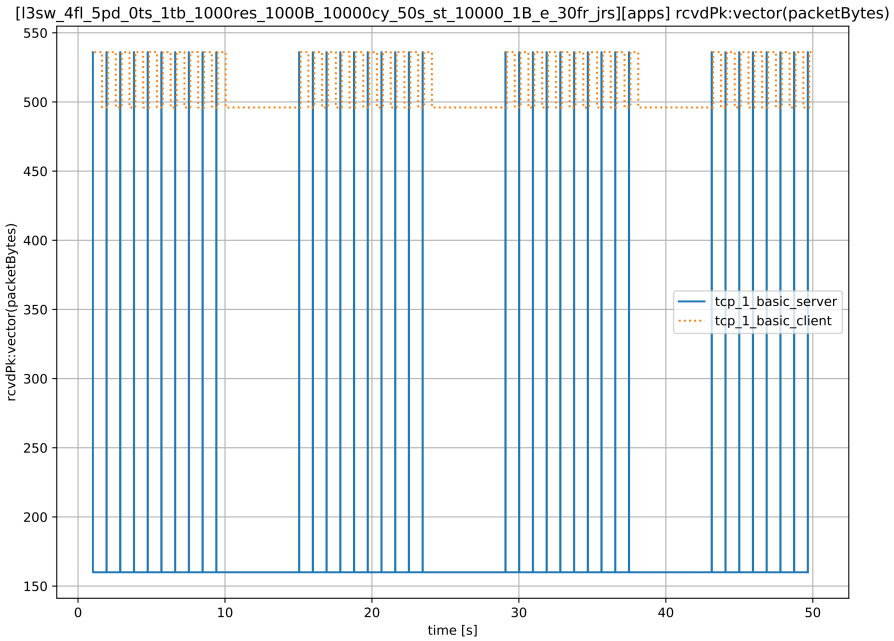
Die drei Linien oben im Graph stammen von den Ausführungen mit dem JSSP-Scheduler, mit dem SMT-Scheduler und komplett ohne aktiviertes Scheduling. In diesen drei Fällen sind die Linien deckungsgleich, da der Best-Effort-Traffic in der Senderate nicht behindert wird. Die beiden Scheduler beeinflussen den Verkehr nicht oder nur kaum, da sie wie in Unterabschnitt 4.2.2 erläutert

nur die tatsächlich benötigte Zeit für die Übertragung der TSN-Flows reservieren, in diesem Fall 8μs. Diese Dauer ist kurz im Vergleich zu den 1000μs, die der JRS-Scheduler pro Flow reserviert, und dadurch keine große Beeinflussung für Best-Effort-Traffic.

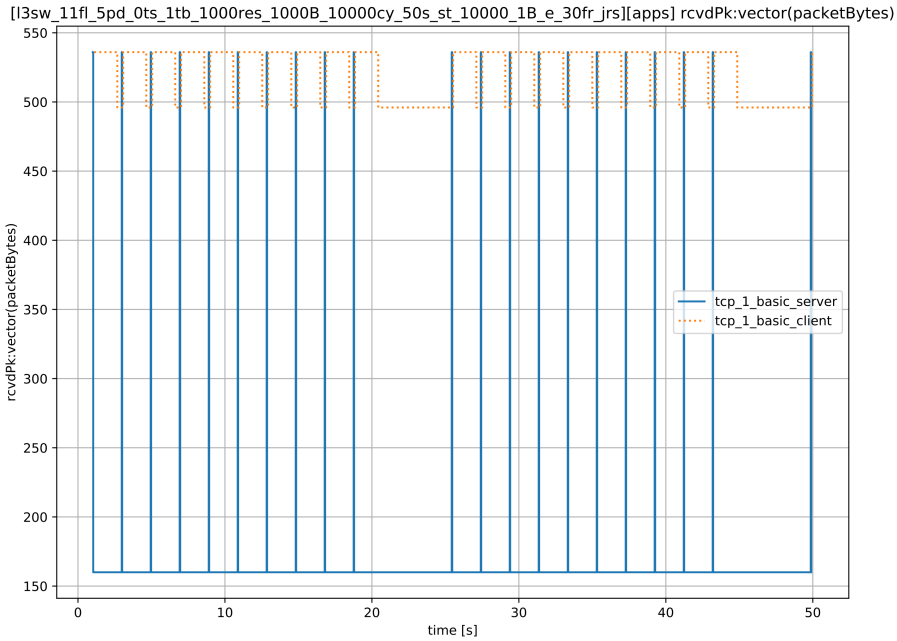
Als TCP-Profil wurde für die gezeigten Simulationen das Profil „e“ aus Tabelle 4.4 verwendet, bei dem der Client immer 10 Anfragen der Größe 1MiB stellt, die mit 50MiB beantwortet werden.

In Abbildung 4.2 sind die empfangenen Daten der TCP-Apps für vier und elf Flows veranschaulicht. Es sind jeweils Blöcke mit zehn Datenmengen zu sehen, die mit steigender Flow-Anzahl immer weiter auseinander gezogen werden. Das legt nahe, dass die Übertragung durch die Reservierungen für TSN-Traffic verzögert werden und die Übertragung einer festen Datenmenge somit mehr Zeit in Anspruch nimmt. Verglichen mit der Simulation mit vier Flows ist die erste TCP-Session in der Simulation mit elf Flows erst nach der doppelten Zeit abgeschlossen. Plots für weitere Simulationen setzen das Muster fort, nach dem die Übertragungen mit steigender Flow-Anzahl immer länger dauern.

Abbildung 4.3 zeigt für die gleichen beiden Simulationen die Queueing-Zeit der Queue 1, welche für Best-Effort-Traffic verwendet wird. Die Queueing-Zeit verfünffacht sich hierbei sogar auf 5 Millisekunden bei elf Flows. Diese Messgröße steigt ebenfalls an, wenn noch weitere Flows gescheduled werden.

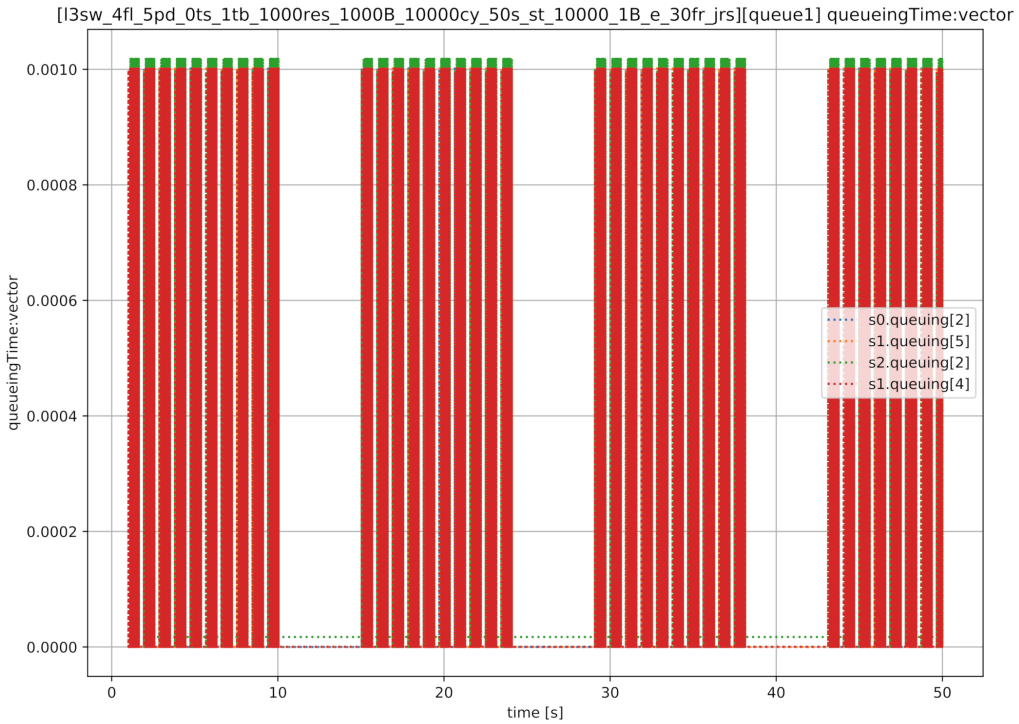


(a) Simulation mit vier Flows

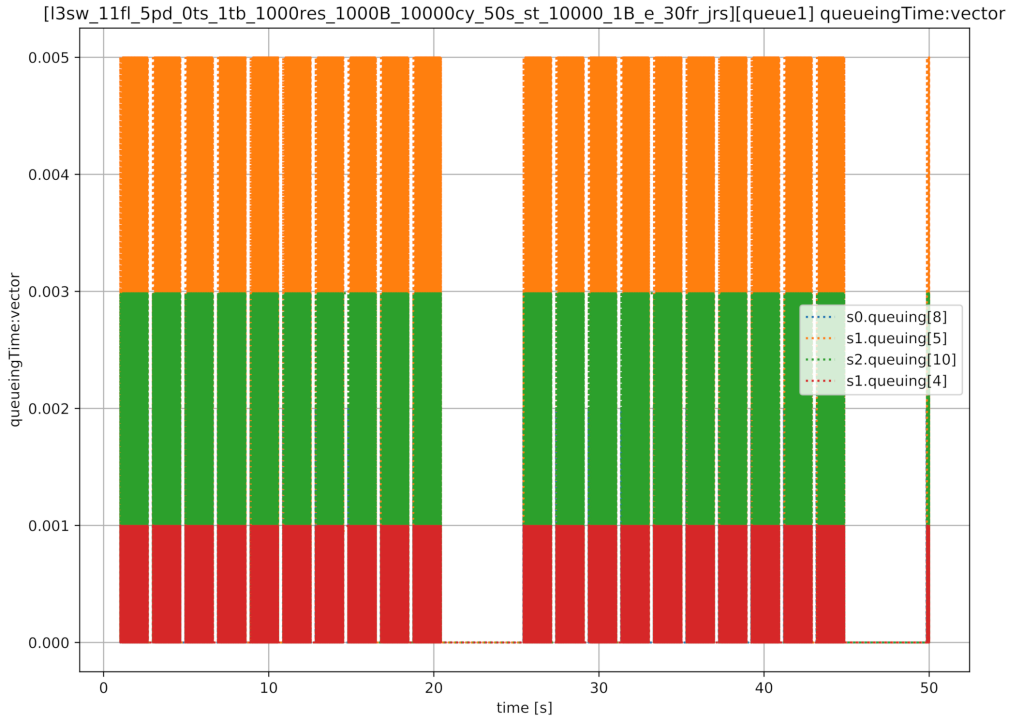


(b) Simulation mit elf Flows

Abbildung 4.2: Von TCP-Client und TCP-Server empfangene Frames in Zwei Simulationen mit einer Flow-Anzahl von vier bzw. elf. Die Konfiguration der restlichen Parameter bestand aus drei Switchen, dem TCP-Profil „e“, 1000μs Reservierung pro Flow und einem Zyklus von 10000μs.



(a) Simulation mit vier Flows



(b) Simulation mit elf Flows

Abbildung 4.3: Queueing-Zeit von Best-Effort-Frames in Queue 1, in zwei Simulationen mit einer Flow-Anzahl von vier bzw. elf. Die Konfiguration der restlichen Parameter bestand aus drei Switchen, dem TCP-Profil „e“, 1000μs Reservierung pro Flow und einem Zyklus von 10000μs.

4.3.3 Round-Trip-Time

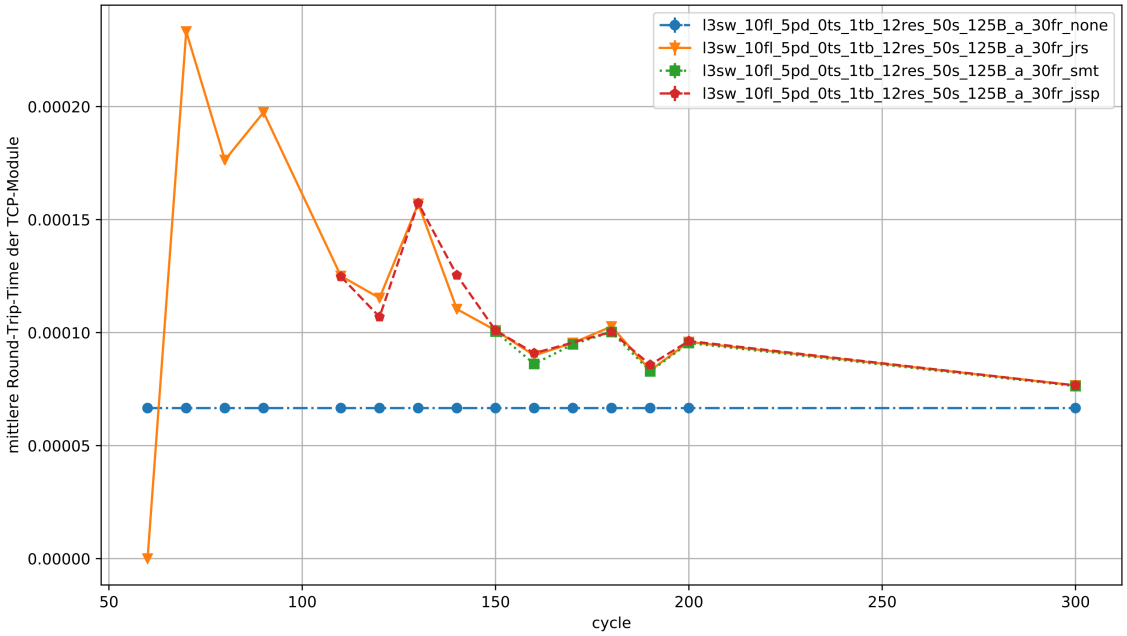


Abbildung 4.4: Die mittlere Round-Trip-Time für eine variierende Zykluszeit bei drei Switchen und 10 Flows mit je 1500 Bytes

Die *Round-Trip-Time* (RTT) ist eine interne Messgröße des INET-TCP-Moduls und beschreibt die Dauer, die TCP-Pakete vom Sender zum Empfänger und zurück benötigen. Die RTT ist eine Messgröße, die eine Aussage über die Latenz oder Aufstauung eines Pfades im Netzwerk treffen kann. Abbildung 4.4 zeigt einen Plot zur Übersicht, der die mittlere Round-Trip-Time für 10 Flows und verschieden große Zykluszeiten aufzeichnet. In dem Beispiel sind die Größen noch so gewählt, dass alle drei Scheduler das Problem gleichartig lösen können und alle reservierten Bereiche auch für die Übertragung verwendet werden. Die blaue Linie zeigt die RTT für die Baseline, eine Simulation der selben Topologie ohne Scheduling. Im Vergleich dazu ist die Round-Trip-Time in den gezeigten Versuchen verdoppelt bis verdreifacht und sinkt mit zunehmender Zykluslänge ab. Der erste Messpunkt mit einer RTT von 0 zeigt einen Fall, in dem der Zyklus so kurz ist, dass keine Bereiche mehr für Best-Effort-Traffic bleiben. Somit kommt nie eine TCP-Verbindung zustande und es kann folglich auch keine RTT gemessen werden. Das ist möglich, da Flows in beide Richtungen entlang des Pfades geschedulet sein können oder die Flows nicht alle Switche durchlaufen und es somit Ports gibt, die nicht durch jeden Flow durchlaufen werden.

5 Zusammenfassung

Diese Arbeit hat sich mit den technischen Aspekten von Time-Sensitive Networking beschäftigt, mit einem besonderen Fokus auf die Simulation der Technologie mit dem Netzwerksimulator OMNeT++. Es wurde das Simulationsframework NeSTiNg für die Simulation von TSN in OMNeT++ vorgestellt und um Frame-Preemption mit Hold und Release aus den TSN-Standards erweitert. Weiter wurden drei Scheduling-Verfahren betrachtet und eine vergleichende Nutzung der Scheduler durch die Anpassung ihrer Eingabe- und Ausgabe-Dateiformate ermöglicht. Durch Zuhilfenahme von Konvertierungsskripten können die erzeugten Schedules nun importiert und in einer NeSTiNg-Simulation verwendet werden. Mit weiteren Skripten wurde eine Automatisierung von TSN-Szenarien umgesetzt, die eine Problembeschreibung mit den Schedulern löst, das TSN-Netzwerk mit den Lösungen simuliert und die Ergebnisse anschließend graphisch darstellt. Mit diesem Verfahren wurden Netzwerke simuliert, bei denen TSN-Flows und Best-Effort-Traffic gemeinsam auftreten, mit dem Ziel, die Auswirkungen auf Best-Effort-Traffic analysieren. Die Ergebnisse zeigen, dass insbesondere die Zykluszeit und die Anzahl der Flows in einem Netzwerk die Service-Qualität von Best-Effort-Traffic mindern. Beide Werte beeinflussen den für TSN reservierten Anteil der Schedules und mindern somit die für Best-Effort-Traffic verfügbare Bandbreite.

Literaturverzeichnis

[Ada17] M. Z. Adam Taylor. „TSN: Converging Networks for a Better Industrial IoT. Success in the IIoT requires that information- and operational-technology networks work in tandem—time-sensitive networking can make it happen.“ In: *Electronic Design* (27. Dez. 2017). URL: http://www.electronicdesign.com/industrial-automation/tsn-converging-networks-better-industrial-iot (besucht am 26. 06. 2018) (zitiert auf S. 27).

[Avn] Avnu Alliance. *Our Members | Avnu Alliance*. URL: http://avnu.org/our-members/ (besucht am 26. 06. 2018) (zitiert auf S. 26, 27).

[BELP13] U. Brandes, M. Eiglsperger, J. Lerner, C. Pich. „Graph Markup Language (GraphML)“. In: *Handbook of graph drawing visualization*. Hrsg. von R. Tamassia. Discrete mathematics and its applications. Boca Raton [u.a.]: CRC Press, 2013, S. 517–541. ISBN: 978-1-58488-412-5 (zitiert auf S. 44).

[CDF+18] B. Carabelli, F. Dürr, J. Falk, R. Finkbeiner, D. Hellmanns, U. Limani, N. Nayak, A. Nieß, P. Schneefuss, N. Segedi, M. Weitbrecht. *NeSTiNg - Network Simulator for Time-sensitive Networking*. 6. Juli 2018. URL: https://gitlab.com/ipvs/nesting (zitiert auf S. 17, 31, 32, 38).

[COCS16] S. S. Craciunas, R. S. Oliver, M. Chmelík, W. Steiner. „Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks“. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Brest, France: ACM, 2016, S. 183–192. ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997470. URL: http://doi.acm.org/10.1145/2997465.2997470 (zitiert auf S. 29, 30).

[DK+14] F. Dürr, T. Kohler et al. „Comparing the forwarding latency of openflow hardware and software switches“. In: (Juli 2014) (zitiert auf S. 27, 59).

[DN16] F. Dürr, N. G. Nayak. „No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)“. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Brest, France: ACM, 2016, S. 203–212. ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997494. URL: http://doi.acm.org/10.1145/2997465.2997494 (zitiert auf S. 21, 28, 29).

[Hab17] A. Habekost. „Kontron stellt auf SPS IPC Drives Evaluations-Unit einer neuen TSN-Netzwerkkarte vor. Zeitsynchronisation mit garantierter Latenz und QoS im Netwerk - Support für Geräte in IIoT-Umgebungen – Private Labelling Version für OEMs geplant – Demo einer TSN-Umgebung am Messestand“. In: (28. Nov. 2017). URL: https://www.kontron.de/about-kontron/news-events/detail/171128_sps_tsn?query=PCIE-0200-TSN (besucht am 26. 06. 2018) (zitiert auf S. 27).

[Ind] Industrial Internet Consortium. *Time Sensitive Networking (TSN) Testbed*. URL: https://www.iiconsortium.org/time-sensitive-networks.htm (besucht am 26. 06. 2018) (zitiert auf S. 27).

[Ins14] Institute of Electrical and Electronics Engineers Inc. „IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks“. In: *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)* (Dez. 2014), S. 1–1832. DOI: 10.1109/IEEESTD.2014.6991462 (zitiert auf S. 18–20, 23).

[Ins16a] Institute of Electrical and Electronics Engineers Inc. „IEEE Standard for Ethernet“. In: *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)* (März 2016), S. 1–4017. DOI: 10.1109/IEEESTD.2016.7428776 (zitiert auf S. 20, 22, 23, 25, 26).

[Ins16b] Institute of Electrical and Electronics Engineers Inc. „IEEE Standard for Ethernet Amendment 5: Specification and Management Parameters for Interspersing Express Traffic“. In: *IEEE Std 802.3br-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE St802.3bw-2015, IEEE Std 802.3by-2016, IEEE Std 802.3bq-2016, and IEEE Std 802.3bp-2016)* (Okt. 2016), S. 1–58. DOI: 10.1109/IEEESTD.2016.7900321 (zitiert auf S. 23–25, 35, 36).

[Ins16c] Institute of Electrical and Electronics Engineers Inc. „IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic“. In: *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q— as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q—/Cor 1-2015)* (März 2016), S. 1–57. DOI: 10.1109/IEEESTD.2016.7572858 (zitiert auf S. 27, 28).

[Ins16d] Institute of Electrical and Electronics Engineers Inc. „IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks – Amendment 26: Frame Preemption“. In: *IEEE Std 802.1Qbu-2016 (Amendment to IEEE Std 802.1Q-2014)* (Aug. 2016), S. 1–52. DOI: 10.1109/IEEESTD.2016.7553415 (zitiert auf S. 23, 25).

[Ins18] Institute of Electrical and Electronics Engineers Inc. *Time-Sensitive Networking Task Group*. 25. Juni 2018. URL: http://www.ieee802.org/1/pages/tsn.html (zitiert auf S. 18, 27).

[IPy] IPython development team. *The Jupyter Notebook*. URL: https://ipython.org/notebook.html (zitiert auf S. 59).

[Jon18] K. R. Jonathan Falk Frank Dürr. „Exploring Practical Limitations of Joint Routing and Scheduling for TSN with ILP“. In: (2018) (zitiert auf S. 26, 28, 29, 44, 45).

[KK79] P. Kermani, L. Kleinrock. „Virtual cut-through: A new computer communication switching technique“. In: *Computer Networks (1976)* 3.4 (1979), S. 267–286. ISSN: 0376-5075. DOI: https://doi.org/10.1016/0376-5075(79)90032-1. URL: http://www.sciencedirect.com/science/article/pii/0376507579900321 (zitiert auf S. 23).

[LLPP16] H. Lee, J. Lee, C. Park, S. Park. *Time-aware preemption to enhance the performance of Audio/Video Bridging (AVB) in IEEE 802.1 TSN*. Okt. 2016 (zitiert auf S. 23).

[Mat] Matplotlib development team. *Matplotlib*. URL: https://matplotlib.org/ (zitiert auf S. 50, 59).

[Mic] Microsoft Research. *z3*. URL: https://github.com/Z3Prover/z3 (besucht am 02. 07. 2018) (zitiert auf S. 31, 43).

[Nik] Nikolaus Rath et. al. *sshfs - A network filesystem client to connect to SSH servers*. URL: https://github.com/libfuse/sshfs (besucht am 27. 07. 2018) (zitiert auf S. 58).

[NXP] NXP. *Time-Sensitive Networking Solution for Industrial IoT*. URL: https://www.nxp.com/support/developer-resources/nxp-designs/time-sensitive-networking-solution-for-industrial-iot:LS1021A-TSN-RD (zitiert auf S. 27).

[Pit16] G. Pitcher. „Deterministic data developments“. In: *New Electronics* (8. März 2016), S. 30. URL: http://www.newelectronics.co.uk/electronics-technology/time-sensitive-networking-is-set-to-meet-the-precise-communication-needs-of-the-industrial-iot-and-smart-cars/116514/ (besucht am 25. 06. 2018) (zitiert auf S. 22–24).

[Pug16] J. F. Puget. *CPLEX Is Free For Students (And Academics)*. 27. Sep. 2016. URL: https://www.ibm.com/developerworks/community/blogs/jfp/entry/CPLEX_Is_Free_For_Students?lang=en (besucht am 03. 07. 2018) (zitiert auf S. 43).

[Ren] Renesas Electronics Corporation. *Renesas Time-Sensitive Networking (TSN)*. URL: https://www.renesas.com/en-eu/solutions/factory/industrial-communications/tsn.html (zitiert auf S. 27).

[Sch18] U. Schulze. „Keine Zeit verschwenden. Echtzeitfähigkeit durch Time-Sensitive Networking (TSN)“. In: *iX* 1 (2018), S. 94–99. URL: https://www.heise.de/-3920173 (zitiert auf S. 26).

[Sie] Siemens AG. *Ganz klar: TSN – der Turbo für PROFINET und OPC UA*. URL: https://www.siemens.com/global/de/home/produkte/automatisierung/industrielle-kommunikation/industrial-ethernet/tsn.html (besucht am 26. 06. 2018) (zitiert auf S. 27).

[SL86] C. Sriskandarajah, P. Ladet. „Some no-wait shops scheduling problems: Complexity aspect“. In: *European Journal of Operational Research* 24 (3 1986), S. 424–438. ISSN: 0377-2217 (zitiert auf S. 28, 29).

[TFB+13] M. D. J. Teener, A. N. Fredette, C. Boiger, P. Klein, C. Gunther, D. Olsen, K. Stanton. „Heterogeneous Networks for Audio and Video: Using IEEE 802.1 Audio Video Bridging“. In: *Proceedings of the IEEE* 101 (2013), S. 2339–2354 (zitiert auf S. 21).

[Var14a] A. Varga. *cChannel Class Reference*. Hrsg. von O. Ltd. 2. Dez. 2014. URL: https://www.omnetpp.org/doc/omnetpp4/api/classcChannel.html#a707271d862ee64e99f4a691f5c44953d (zitiert auf S. 40).

[Var14b] A. Varga. *TcpBasicClientApp.ned*. Hrsg. von O. Ltd. 9. Dez. 2014. URL: https://github.com/inet-framework/inet/blob/v3.6.3/src/inet/applications/tcpapp/TCPBasicClientApp.ned (besucht am 03. 07. 2018) (zitiert auf S. 50).

[Var14c] A. Varga. *TCPEchoApp.ned*. Hrsg. von O. Ltd. 9. Dez. 2014. URL: https://github.com/inet-framework/inet/blob/v3.6.3/src/inet/applications/tcpapp/TCPEchoApp.ned (besucht am 03. 07. 2018) (zitiert auf S. 49, 50).

[Var14d] A. Varga. *TCPGenericSrvApp.ned*. Hrsg. von O. Ltd. 9. Dez. 2014. URL: https://github.com/inet-framework/inet/blob/v3.6.3/src/inet/applications/tcpapp/TCPGenericSrvApp.ned (besucht am 03. 07. 2018) (zitiert auf S. 50).

[Var14e] A. Varga. *TCPSessionApp.ned*. Hrsg. von O. Ltd. 9. Dez. 2014. URL: https://github.com/inet-framework/inet/blob/v3.6.3/src/inet/applications/tcpapp/TCPSessionApp.ned (besucht am 03. 07. 2018) (zitiert auf S. 49).

[Var16] A. Varga. *INET Framework for OMNeT++*. Hrsg. von O. Ltd. 22. Jan. 2016 (zitiert auf S. 17).

[Var17] A. Varga. *OMNeT++ User Guide*. Hrsg. von O. Ltd. 5.3. 2017. URL: https://www.omnetpp.org/doc/omnetpp/UserGuide.pdf (besucht am 25. 06. 2018) (zitiert auf S. 15).

[Var18a] A. Varga. *INET framework for the OMNeT++ discrete event simulator*. Hrsg. von O. Ltd. 26. Juni 2018. URL: https://github.com/inet-framework/inet (zitiert auf S. 17).

[Var18b] A. Varga. *OMNeT++ Simulation Manual*. Hrsg. von O. Ltd. 5.3. 2018. URL: https://www.omnetpp.org/doc/omnetpp/manual/ (besucht am 25. 06. 2018) (zitiert auf S. 15–17).

[Var18c] A. Varga. *Recent changes in the INET Framework*. Hrsg. von O. Ltd. Version INET-3.6.0 (June, 2017) - stable. 26. Juni 2018. URL: https://omnetpp.org/doc/inet/api-current/doxy/whatsnew.html (zitiert auf S. 18).

URLs wurden, sofern nicht anders angegeben, zuletzt am 28.07. 2018 geprüft.

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