Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

InES Institute of Embedded Systems

Projektarbeit Informatik

PA14 wlan 1

Performance-Evaluation Ethernet für Echtzeit-Datenerfassung

Autoren	Mauro Guadagnini (guadamau@students.zhaw.ch) Prosper Leibundgut (leibupro@students.zhaw.ch)
Hauptbetreuung	Hans Weibel (wlan@zhaw.ch)

Datum	19.12.2014

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

.....

Unterschriften:

.....

.....

.....

Zusammenfassung

Für Echtzeit-Systeme, die aus mehreren Komponenten bestehen, welche miteinander über ein Netzwerk verbunden sind, haben Faktoren wie eine ausfallsichere Datenübertragung und garantierte Datenübertragungszeiten eine signifikante Bedeutung. Ein Ansatz, um diesen Kriterien gerecht zu werden, bietet das High Availability Seamless Redundancy (nachfolgend auch bezeichnet als HSR) Protokoll. Redundante Datenübertragung und die Struktur der Verbindungen umgesetzt als Ring-Topologie zählen zu den Haupt-Charakteristiken des HSR-Protokolls. Im Zentrum dieser Arbeit liegt die Modellierung verschiedener Szenarien, basierend auf einem HSR-Ring-Netzwerk mit verschiedenen Instanzen von HSR-Endgeräten. Zu diesem Zweck wird ein virtueller HSR-Knoten programmiert, der einige Regelwerke implementiert. Diese Regelwerke beziehen sich auf die Behandlung von Ethernet Frames und den Datenfluss in einem HSR-Ring. Für das Erreichen garantierter Höchstübertragungszeiten werden die Ethernet Frames mit einer Priorität versehen, was die Unterbrechung von niedrig priorisierten Ethernet Frames ermöglicht. Hierzu wird der IEEE Draft P802.3br/D1.0 [3] als Grundlage verwendet, um die Verhaltensweise von Interspersing Express Traffic (IET) in Form von MAC Merge Frames (mFrames) zeitlich zu simulieren. Als Simulationsumgebung wird OMNeT++ eingesetzt, eine diskrete Ereignissimulationsumgebung für Kommunikationsnetzwerke. Die Ergebnisse der vorliegenden Arbeit zeigen auf, wie sich die Verzögerungszeiten von Ethernet Frames unter Berücksichtigung verschiedener Regelwerke und mannigfaltigem Datenaufkommen im HSR-Ring verhalten. Ferner werden Aspekte beleuchtet, wie sich der Datenrückstau bezüglich verschiedener Datenflussregeln in einzelnen HSR-Knoten verhält und wie viele Ethernet Frames jeweils während eines Simulationslaufs unterbrochen werden.

Abstract

In real-time systems consisting of several components, which are connected by a network, aspects like failsafe data transmission and guaranteed data transmission times are of significant importance. An approach to fulfil these requirements is provided by High Availability Seamless Redundancy (hereafter also referred to as HSR) protocol. Redundant data transmission and the structure of the compounds implemented as a ring topology rank as the main characteristics of the HSR protocol. The main focus of this project consists of the modelling of various scenarios based on a virtual HSR ring network composed of distinct autonomous instances of HSR end devices. For this purpose a virtual HSR node which implements certain rule sets is programmed. These rule sets apply to the treatment of Ethernet frames and the dataflow in a HSR ring. In order to achieve guaranteed maximum transmission times, Ethernet frames are prioritized, which allows the preemption of low prioritized Ethernet frames. For this purpose, the IEEE Draft P802.3br / D1.0 [3] is used as a basis to simulate the timed behaviour of Interspersing Express Traffic (IET) in the form of MAC Merge Frames (mFrames). The simulation environment used is OMNeT++, a discrete event simulation environment for communication networks. The results of this project work reveal the impacts on behaviour of delay times of Ethernet frames in regard to different rule sets and varied data traffic in a HSR ring. Moreover, other factors are also discussed; among these are data backlog in relation to various dataflow rules of particular HSR nodes and how many Ethernet frames are preempted during one simulation run.

Vorwort

Ab dem 4. Semester des Informatik-Studiums werden erstmals Module unterrichtet, die spezifisch für die jeweilig gewählte Fachrichtung der Studierenden ausgelegt sind. Wir haben zum einen die Fachrichtung Service Engineering (Mauro Guadagnini) und zum anderen Embedded Systems (Prosper Leibundgut) selektiert. In der Fachrichtung Service Engineering geht es primär darum, die Fachkenntnisse über das Errichten und Unterhalten umfangreicher Services und Netzwerk-Infrastrukturen weiter zu vertiefen. Die Fachrichtung Embedded Systems befasst sich hauptsächlich damit, Firmware für eingebettete Systeme zu entwickeln, die ohne oder nur mit minimalistischen Betriebssystemen betrieben werden. Solche Geräte haben oft die Anforderung zu erfüllen, Datenverarbeitungen in Echtzeit auszuführen. Nicht selten müssen diese Geräte auch über eine Netzwerkschnittstelle verfügen, um Daten an entfernte Netzwerkteilnehmer zu übermitteln, beziehungsweise diesen einen Zugriff auf besagte Geräte zu ermöglichen.

Unser gemeinsames Interesse liegt im Bereich von netzwerkbasierten Anwendungen und performanter Datenübertragung. Bei der Wahl der ausgeschriebenen Projektarbeiten fiel uns die Arbeit «Performance-Evaluation Ethernet für Echtzeit-Datenerfassung» auf. Die Entscheidung, uns für diese Arbeit einzuschreiben, war nach kurzer Zeit getroffen. Wir zeigen beide grosses Interesse an gehaltvollen Technologien und so sahen wir es als zusätzliche Gelegenheit, ein tieferes Verständnis für Netzwerksimulationen, das HSR-Protokoll und Interspersing Express Traffic zu gewinnen, was uns im Verlaufe der Umsetzung dieser Arbeit auch gelungen ist.

Unser herzlicher Dank gilt Herrn Prof. Hans Weibel für die geduldige und fachkundige Arbeitsbetreuung in Form von Impulsen und Diskussionen, die zu geeigneten Lösungsansätzen verhalfen.

Inhaltsverzeichnis

Teil I.

Einführung und Grundlagen

1. Einleitung

1.1. Ausgangslage

1.1.1. Stand der Technik

Das Unterbrechen der Übertragung von Frames durch Express-Frames (IET) sowie das dadurch zu verwendende mFrame-Format sind noch nicht in Hardware implementiert. Es ist ein Entwurf in Entwicklung [5], der voraussichtlich Ende 2015 zum Standard werden soll und seit Mai 2014 keine weiteren Features mehr erhält, technische Änderungen jedoch noch bis März 2015 eingeführt werden können [6].

Aufgrund dieser Tatsache findet man im Internet kaum etwas zu IET sowie dem mFrame-Format, ausser einigen Unterlagen vom IEEE-Verband, der den Standard entwickelt.

Bezüglich HSR (High Availability Seamless Redundancy) gibt es einiges mehr zu finden. Das HSR-Protokoll ist seit Februar 2010 unter dem Titel «IEC 62439-3 Cl. 5» ein Standard und wurde schon bei einigen Firmen implementiert [8]. Da IET noch nicht in Hardware implementiert wurde, existieren auch keine Berichte zur Implementation von IET in einem HSR-Netzwerk.

1.1.2. Bestehende Arbeiten

Die bestehende Vertiefungsarbeit [1] behandelt das Simulieren von Frames in einem HSR-Netzwerk, jedoch ohne IET-Implementation, die dazugehörige Frame-Priorisierung und dem mFrame-Format. Aufgrund des derzeitigen Standes der Technik wurden keine Arbeiten gefunden, die sich mit unserer Thematik (HSR-Netzwerk mit IET-Implementation) beschäftigen.

1.2. Zielsetzung / Aufgabenstellung / Anforderungen

Durch das Institute of Embedded Systems der ZHAW wurde den Autoren am 24. September 2014 eine Aufgabenstellung [13] (siehe Kapitel 11 auf Seite 125) zugestellt, welche die nachfolgenden Hauptanforderungen umfasst:

1.2.1. Modell für HSR-Knoten erweitern

Das betrachtete Netzwerk ist ein HSR-Ring. Die bestehende Simulationsumgebung [1] soll so erweitert bzw. angepasst werden, dass folgende Funktionen/Mechanismen simuliert werden können **[13]**:

- Der Knoten soll zwei Prioritäten unterstützen, d.h. zwei Warteschlangen pro Interface bewirtschaften.
- Der Knoten soll Interspersing Express Traffic (IET) unterstützen, d.h. Express-Frames können die aktuell ablaufende Übertragung eines Frames unterbrechen.
- Der in den Ring einfliessende Traffic kann limitiert werden.
- Die Vortrittsregeln bezüglich der im Ring zirkulierenden Frames und den Frames, die in den Ring einfliessen, können variiert werden (z.B. «zirkulierende Frames haben immer Vortritt» oder «minimaler Zufluss wird garantiert»).
- Der Knoten implementiert ein Zeitschlitzverfahren, welches dem zeitkritischen Traffic und dem Bulk Traffic je eine Phase zuordnet.

1.2.2. Lastmodell beschreiben und implementieren

Das durch die Anwendung generierte Verkehrsaufkommen ist zu studieren und zu beschreiben. Lastgeneratoren sollen implementiert werden, die das Verkehrsaufkommen für die Simulation generieren durch die Überlagerung von Strömen mit folgender Charakteristik [13]:

- Lastgenerator mit konstanter Framerate.
- Lastgenerator mit zufälliger zeitlicher Verteilung der Frames.
- Lastgenerator, der spontane Einzelmeldungen erzeugt.

1.2.3. Simulationen durchführen und Resultate interpretieren

Das Zeitverhalten der verschiedenen Weiterleitungsvarianten soll durch entsprechende Simulationsläufe ermittelt werden. Die Resultate sind zu vergleichen und zu interpretieren [13].

1.2.4. Erwartetes Resultat

Das Resultat der Arbeit soll verschiedene Verhaltensweisen von Frames in einem HSR-Ring mit IET-Implementation bei unterschiedlichen Bedingungen aufzeigen. Durch eine Interpretation der Verhaltensweisen ist dann die bestmögliche Konfiguration des HSR-Rings zu ermitteln, mit welcher zeitkritische Frames am schnellsten übermittelt werden.

1.2.5. Vorausgesetztes Wissen

In den theoretischen Grundlagen (siehe Kapitel 2 auf der nächsten Seite) werden unter anderem OMNeT++, das HSR-Protokoll und der Aufbau eines HSR-Netzwerks inklusive dessen Gerätetypen behandelt.

Zum Verständnis dieser Projektarbeit ist ein Vorwissen über die allgemeine Netzwerkkommunikation nötig. Dieses Vorwissen umfasst folgende Bereiche:

- Allgemeine Netzwerk- und Hardware-Begriffe wie z.B. MAC-Adresse, Ethernet-Port oder Ethernet-Frame.
- Funktionsweise eines Netzwerks inklusive der Übertragung eines Ethernet-Frames und dem Aufbau dessen Headers.

2. Theoretische Grundlagen

2.1. OMNeT++

OMNeT++ ist ein C++-Framework, welches es ermöglicht, Netzwerke und all deren Komponenten mit einem sehr hohen Detaillierungsgrad zu modellieren und den Netzwerk-Datenverkehr zu simulieren. Die Simulationen können grafisch dargestellt werden. Zur Auswertung der Simulationen steht eine grosse Auswahl an verschiedenen Diagrammtypen zur Verfügung. In dieser Projektarbeit wird die Version 4.5 verwendet.

Die Abhandlung der Simulationen in OMNeT++ erfolgt sequentiell, trotzdem lassen sich gleichzeitige Vorkommnisse simulieren, da rein die Simulationszeit in Betracht gezogen wird und somit in der Simulation zum Beispiel zwei Frames zum Zeitpunkt t=2.0s versendet werden können (im Hintergrund werden diese immer noch nacheinander abgehandelt). Somit hat diese Eigenschaft keinen Einfluss auf die Resultate der Simulation. Der Nachteil ist, dass man keine Schleifen in die Simulation implementieren kann (die Simulation hängt sich auf, wenn sie sich in einer While-Schleife befindet). Stattdessen muss man mit der Simulationszeit und Events arbeiten, was aber nach kurzer Zeit kein grosses Hindernis mehr ist. Es besteht die Möglichkeit, die Simulation in mehreren Threads zum Laufen zu bringen, jedoch erfordert dies einen sehr hohen Aufwand und bringt eine hohe Komplexität mit sich, weshalb nur dazu geraten wird, wenn es absolut unabdingbar ist [12].

2.2. High Availability Seamless Redundancy (HSR)

HSR ist ein Ethernet-Redundanz-Protokoll, das im Fehlerfall keine Ausfallzeit hat und es erlaubt, Geräte zusammen zu schliessen, um ein kosteneffektives Netzwerk betreiben zu können. Es ermöglicht komplexe Topologien wie Ringe und Ringe von Ringen und ist einfach in Hardware zu implementieren. Besonders für Anwendungen, bei denen Unterbrüche nicht tolerierbar sind, ist HSR ein attraktives Protokoll [8].

Die Bedingung für ein Gerät in einem HSR-Netzwerk ist, dass es mindestens zwei Ethernet Ports haben muss. Ein nicht HSR-fähiges Gerät wird mittels einer Redundancy Box (RedBox) angeschlossen, welche die HSR-Funktionen stellvertretend erbringt. Wenn ein Frame versendet werden muss, wird eine Kopie davon erstellt und auf den Ring gleichzeitig in beide Richtungen übertragen. In einem fehlerfreien Netzwerk kommen somit immer zwei oder mehr Frames beim Empfänger an. Es ist die Aufgabe des Empfängers, Duplikate nicht an höhere Schichten weiter zu reichen. Unicast-Frames und dessen Duplikate werden vom Empfänger vom Ring entfernt. Multicast- und Broadcast-Frames werden vom Sender vom Ring entfernt. Der Sinn davon ist, dass bei einem Ausfall eines Gerätes mindestens ein Frame trotzdem an seinen Empfänger

gelangt. Der Frameverlust wird demnach bei einem Fehlerfall verhindert [7]. Höhere Schichten bekommen von dem Ganzen nichts mit [8].

Die Duplikat-Erkennung findet anhand der Quell-MAC-Adresse und der Sequenznummer eines Frames statt. Jedes Gerät führt eine Liste mit den Sequenznummern für jede MAC-Adresse und kann somit einfach bereits erhaltene Frames feststellen. Diese Einträge werden solange behalten wie sich das Frame im HSR-Netzwerk befindet.

2.2.1. Gerätetypen

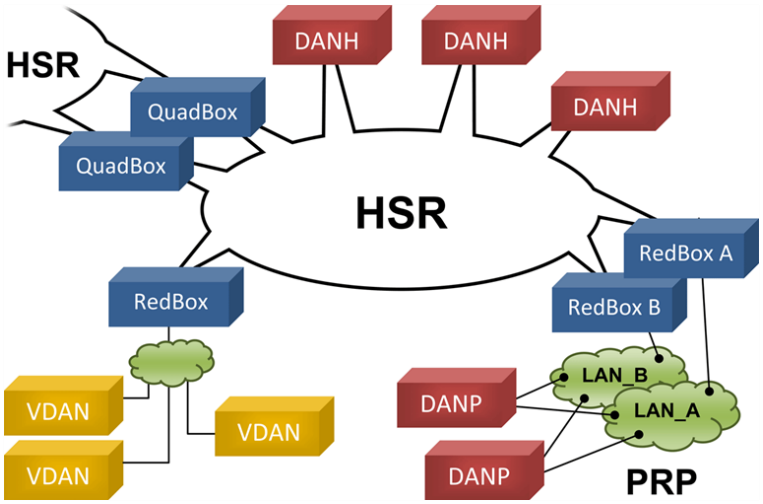


Abbildung 2.1.: HSR-Ring mit allen möglichen Gerätetypen[7]

Name	Beschreibung
DANH	**Double Attached Node implementing HSR**: Direkt in den Ring eingefügter Knoten mit zwei Ethernet Ports. Die beiden Ports teilen sich dabei die MAC- und IP-Adresse [9].
RedBox	**Redundancy Box**: Dient dazu, nicht HSR-fähige Geräte an einem Netzwerk anzuschliessen.
QuadBox	Werden verwendet, um HSR-Ringe miteinander zu koppeln. Eine Quadbox hat 4 Ethernet Ports. Um zwei HSR-Ringe miteinander zu koppeln benötigt es zwei Quadboxen.
VDAN	**Virtual Doubly Attached Node**: (Nicht HSR-fähige) Knoten, die mittels RedBox am HSR-Netzwerk angeschlossen sind.
DANP	**Double Attached Node implementing PRP**: Endknoten in einem PRP-Netzwerk, welcher über eine RedBox an einem HSR-Netzwerk angeschlossen ist. Ein DANP muss für die Kommunikation mit einem DANH das HSR-Protokoll nicht kennen.

Tabelle 2.1.: Gerätetypen in einem HSR-Netzwerk [7]

2.3. Interspersing Express Traffic (IET)

Interspersing Express Traffic ist ein Entwurf des Standardisierungsgremiums IEEE unter der Bezeichnung «IEEE 802.3br», welcher voraussichtlich Ende 2015 zum Standard werden soll [6]. Die Recherche für den Entwurf wurde von einer Gruppe beim IEEE erarbeitet, die unter dem Namen «IEEE 802.3 Distinguished Minimum Latency Traffic in a Converged Traffic Environment (DMLT) Study Group» daran gearbeitet hat. Für das Erarbeiten des Entwurfs ging die Gruppe dann zur «IEEE P802.3br Interspersing Express Traffic Task Force» über [2].

Ziel ist es auf OSI Layer 2 «Data Link» den Standard IEEE 802.3 «Ethernet» zu erweitern, um IET zu ermöglichen. Mittels IET wird es möglich sein, dass Geräte, die IET unterstützen, zwischen sogenannten normalen Frames und Express-Frames unterscheiden, den Sendevorgang der normalen Frames unterbrechen und somit Express-Frames (auch IET Frames genannt) schneller senden können [4].

Der Grund dafür ist, dass neue Märkte wie zum Beispiel die industrielle Automatisierung und Transport (Flugzeuge, Züge und grosse Fahrzeuge) Ethernet adaptiert haben und die Nachfrage nach einer kleinen Latenz aufgrund von ihrer Hochverfügbarkeit steigt [4]. Dringende Meldungen können dann dem üblichen Traffic vorgezogen und schneller erkannt werden.

Für den Standard ist unter anderem vorgesehen, dass das Ethernet-Frame-Format beibehalten wird, keine Änderungen auf OSI Layer 1 «Physical» gemacht werden und die Express-Frames von Geräten, die nicht IET-fähig sind, verworfen werden [5]. Die normalen Frames werden, wenn sie durch Express-Frames unterbrochen werden, aufgeteilt in mFrames («MAC Merge Frame», siehe Kapitel 2.4 auf Seite 18), von denen man auf OSI Layer 1 nichts mitbekommt [5]. So wird das bereits Gesendete nicht verworfen, sondern wird mit dem später ankommenden Rest wieder zusammengesetzt. Express-Frames werden nicht fragmentiert.

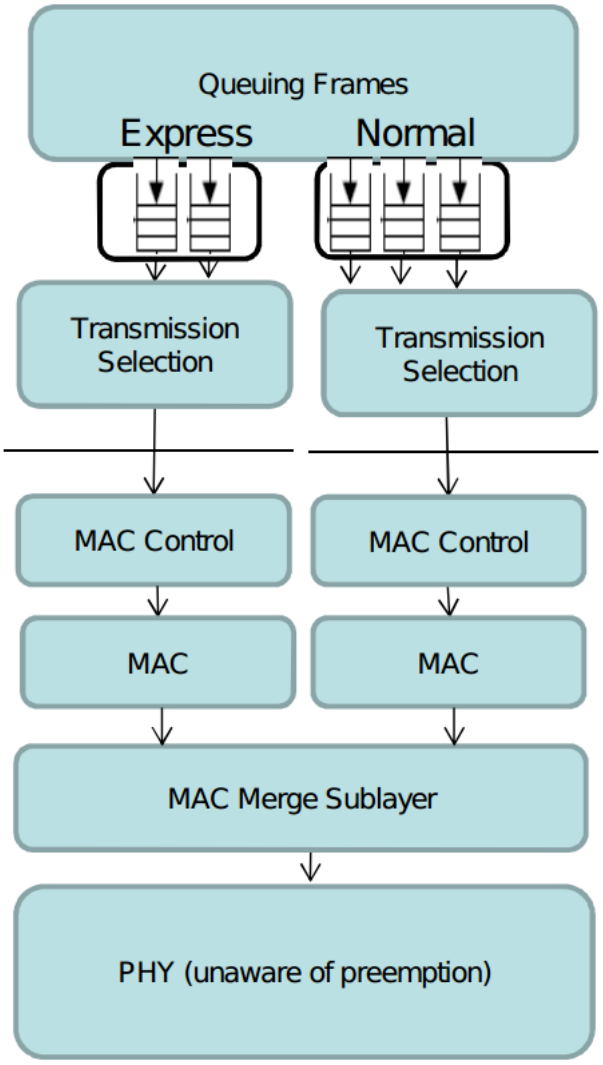


Abbildung 2.2.: MAC Merge Layer [5]

2.4. mFrame

Ein mFrame (steht für «MAC Merge Frame») ist eine Einheit, welche ganze Frames und Fragmente von unterbrechbaren Frames beinhalten kann und wie ein normales Frame auf dem OSI Layer 1 aussieht [5]. Die Struktur eines mFrames ist dem eines Ethernet Frames sehr ähnlich.

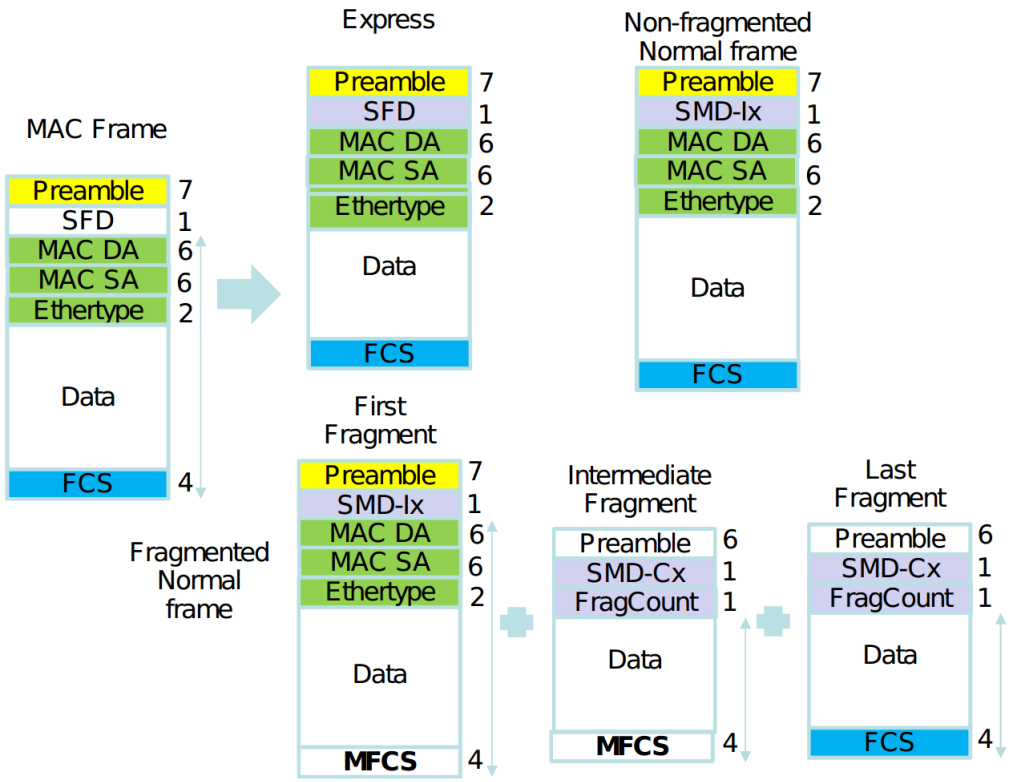


Abbildung 2.3.: mFrame Format [5]

Das SMD-Feld nach der Präambel (anstelle des SFD (Start Frame Delimiter) Feldes) wird in einem mFrame verwendet, um die Fragmente zu kennzeichnen. So hat das erste Fragment einen SMD-Ix-Wert, um zu signalisieren, dass es sich um das erste Fragment eines Frames handelt. Die darauf folgenden Fragmente haben einen SMD-Cx-Wert und ein FragCount-Feld. Im SMD-Ix- und SMD-Cx-Feld wird jeweils eine Frame-Nummer vergeben, wobei im FragCount-Feld die Nummer des Fragments steht. Dabei wechselt jedes dieser Felder jeweils zwischen 4 verschiedenen Werten (siehe Tabelle 2.2 auf der nächsten Seite). Für jedes neue Fragment wird das FragCount-Feld mittels Modulo-4-Zähler hochgezählt [3]. Gibt es dann ein Fragment mit dem FragCount Wert 3, so hat das nächste Frame wieder den FragCount-Wert 0. Mittels FragCount-Feld wird davor geschützt, dass falsche Frames zusammengesetzt werden, wenn bis zu 3 Fragmente verloren gegangen sind. Die FragCount-Werte haben jeweils eine Hamming-Distanz von 4 zueinander, um den Wert bei Bitfehlern in der Übertragung trotzdem erkennen zu können [5].

mFrame Typ	Frame #	SMD
SFD (express)	NA	0xD5
SMD-Ix	0	0xE6
	1	0x4C
	2	0x7F
	3	0xB3
SMD-Cx	0	0x61
	1	0x52
	2	0x9E
	3	0xAD

FragCount	Frag
0	0xE6
1	0x4C
2	0x7F
3	0xB3

Tabelle 2.2.: SMD und FragCount Codierungen [5]

Die Grösse des Data-Felds umfasst vom ersten Oktett nach dem SFD/SMD-Feld bis und mit dem letzten Oktett vor dem CRC und hat mindestens eine Grösse von 60 Bytes [3]. Da im ersten Fragment im Data-Bereich mehr als nur Nutzdaten sind, beträgt dort der effektiv kleinste Datenbereich (abzüglich MAC-Empfänger-, MAC-Sender-Adresse und EtherType-Feld, also 14 Bytes) 46 Bytes. Hat das ursprüngliche Frame zudem einen VLAN-Tag, können die effektiven Daten mindestens 42 Bytes gross sein.

Eine Fragmentierung findet jedoch nur statt wenn mindestens 64 Data-Bytes versendet und mindestens 64 Data-Bytes noch ausstehen. Dabei muss das bereits Versendete ein Vielfaches von 8 sein [3].

MFCS ist die Blockprüfzeichenfolge (oder Frame Check Sequence (FCS)) eines nicht-finalen Fragments, dessen Wert derselbe wie einer FCS ist, wobei die ersten 2 der 4 Bytes invertiert sind (XOR FFFF0000). Das letzte Fragment hat dann wieder eine FCS anstelle einer MFCS, um zu signalisieren, dass es sich um das letzte Fragment dieses Frames handelt [5].

2.5. Beispiel mit IET und mFrame

Folgendes Szenario ist zu betrachten: Es wird gerade ein normales Frame mit der Priorität High oder Low gesendet. Während diesem Sendevorgang trifft ein Express-Frame ein. Der Sendevorgang des normalen Frames muss nun unterbrochen werden, indem das normale Frame fragmentiert wird. Die Sequenz der zu sendenden Frames würde folgendermassen aussehen (mFrame Format siehe Abbildung 2.3 auf Seite 18):

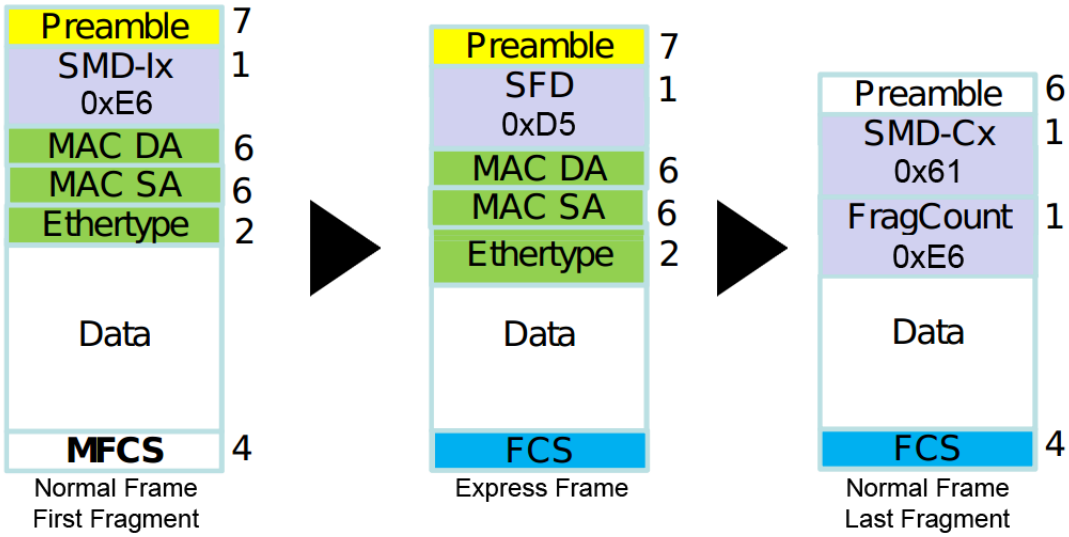


Abbildung 2.4.: Beispiel eines Sendevorgangs, bei dem ein Express Frame ein normales Frame in zwei Fragmente aufteilt

Sobald das letzte Fragment eintrifft, wurde das Frame komplett übertragen und dessen Frame-Nummer (siehe Tabelle 2.2 auf der vorherigen Seite) wird wieder für ein neues Frame verfügbar.

Teil II.

Engineering

3. Vorgehen / Methoden

3.1. Aufbau der Simulation

In OMNeT++ wird ein HSR-Ring-Netzwerk aufgebaut, in welchem der Traffic simuliert wird. Mittels Konfigurationsdateien kann man die zu verwendenden Prioritäten (siehe Kapitel 3.1.1) und Mechanismen (siehe Kapitel 3.1.2) definieren, die in den weiteren Kapiteln genauer behandelt werden.

3.1.1. Prioritäten der Ethernet-Frames

Für die Frames sind 3 verschiedene Prioritäten vorgesehen, nämlich sogenannte Express-, High- und Low-Prioritäten. Dabei handelt es sich um das in Kapitel 2.3 auf Seite 16 beschriebene Express-Frame, wobei die High- und Low-Frames als normales Frame gelten und sich lediglich bei der Priorisierung unterscheiden.

In dieser Arbeit werden wir diese Frames wie folgt auf technischer Ebene unterscheiden:

- Ein Express-Frame wird in der Simulation im VLAN-Tag mit der Priorität 2 versehen. Die hier erwähnte Kennzeichnung ist eine der Autoren ausgedachten Lösungen. Im Standard wird anhand des SFD-/SMD-Feldes zwischen normalen Frames und Express-Frames unterschieden.

- Bei einem High-Frame handelt es sich um ein normales Ethernet-Frame, das über ein VLAN-Tag mit der Priorität 1 gekennzeichnet ist. Da das VLAN-Priority-Feld 3 Bits gross ist wären 8 verschiedene Prioritäten möglich, jedoch wird in dieser Arbeit keine Prioritätsnummer, die höher als 2 ist, verwendet.

- Als ein Low-Frame gelten alle Frames, welche einen VLAN-Tag mit der Priorität 0 oder keine besondere Kennzeichnung haben.

3.1.2. Mechanismen zur Trafficregelung

In diesem Kapitel werden Mechanismen erläutert, die den Traffic regeln und in der Simulation angewendet werden können. Jeder Node (DANH oder Redbox, siehe Kapitel 2.2.1 auf Seite 15) im Netzwerk verfügt über einen internen Switch, der entscheidet, wohin die Frames gesendet werden sollen. In der Simulation verfügt ein DANH über drei Ports, zwei für die Frames vom und auf den Ring (Ethernet-Ports) und einen für die Frames von und zur CPU.

Für jeden möglichen Ausgang ist ein Scheduler zuständig (d.h. bei einem DANH-Gerät hat es drei Scheduler), in welchem die Mechanismen zur Trafficregelung implementiert sind und die Frames priorisiert abgehandelt werden. Der Scheduler ist innerhalb des Switches implementiert und erhält vom Switch Zugang zu dessen Ausgängen, die er benötigt und die Frames, die er für seinen Ausgang zu koordinieren hat. Es besteht dann die Möglichkeit, für jeden Knoten zwischen den Mechanismen auszuwählen oder diese sogar zu kombinieren.

3.1.2.1. First Come First Serve

Die Frames werden, unabhängig von ihrer Herkunft (von Aussen oder vom Ring), nur nach ihrer Priorität behandelt. Das erste Frame, das von der jeweiligen Priorität eintrifft, wird dem anderen der selben Priorität vorgezogen.

3.1.2.2. Zuflusslimitierung

Jedes Gerät generiert intern eine Anzahl an Tokens bis zu einem bestimmten Maximum. Möchte das Gerät ein neues, intern generiertes Low-Frame versenden, kann es das Frame erst versenden wenn genügend Tokens vorhanden sind. Dabei verbraucht ein Low-Frame mit n Bytes genau n Tokens. Wenn zum Beispiel ein Low-Frame mit 800 Bytes versendet werden muss und nur 600 Tokens vorhanden sind, muss das Gerät mit dem Versand warten, bis genügend Tokens vorhanden sind.

Es werden lediglich Low-Frames limitiert, damit nichts vom hoch priorisierten Traffic limitiert wird. So kann der Zufluss von neuem «unwichtigem» Traffic limitiert werden, womit die Frames mit der Priorität High und Express weniger von Frames mit der Priorität Low gestört werden.

3.1.2.3. Vortrittsregeln bezüglich Frames im und zum Ring

Damit die Frames innerhalb des Rings (HSR-Netzwerks) schneller zum Ziel kommen, kann diesen Frames der Vortritt gegenüber Frames von Aussen gewährt werden. Dies ist bei gleich priorisierten Frames der Fall. Es kann hier jedoch die Möglichkeit geben, dass so nie Frames von Aussen versendet werden.

Je nach auftretenden Prioritäten kann es auch sein, dass von Aussen ein Express-Frame und vom Ring ein High-Frame kommt. Dann hat das Express-Frame trotz der Herkunft von Aussen aufgrund dessen Priorität Vortritt.

Die andere Möglichkeit ist, den Frames, die von Aussen auf den Ring sollen, den Vortritt zu gewähren. So kann ein minimaler Zufluss auf den Ring garantiert werden. Hier kann es den umgekehrten Fall wie wenn die Frames vom Ring Vortritt haben geben: Wenn ständig neue Frames von Aussen kommen, werden die Frames vom Ring nie versendet.

3.1.2.4. Reissverschluss

Ein Derivat des vorherigen Mechanismus ist, den Vortritt zwischen Frames vom Ring und von Aussen zu alternieren. Die Frames würden dann wie folgt priorisiert werden (Angenommen es kommen alle 3 Prioritäten (Express, High und Low) vor):

Liste 1

1. Express-Frame vom Ring
2. Express-Frame von Aussen
3. High-Frame vom Ring
4. High-Frame von Aussen
5. Low-Frame vom Ring
6. Low-Frame von Aussen

Liste 2

1. Express-Frame von Aussen
2. Express-Frame vom Ring
3. High-Frame von Aussen
4. High-Frame vom Ring
5. Low-Frame von Aussen
6. Low-Frame vom Ring

So würde die Priorisierung zwischen der aus Liste 1 und der aus Liste 2 nach jedem Frame-Versand abwechseln.

3.1.2.5. Zeitschlitzverfahren

Im Zeitschlitzverfahren werden Zeitschlitze definiert, in denen man High- und Low-Frames senden kann. Ein Express-Frame kann immer versendet werden.

Jeder Zeitschlitz hat zwei gleich grosse Phasen, davon eine sogenannte Grün-Phase, in welcher High-Frames gesendet werden können. Die Rot-Phase wird für den restlichen Sendevorgang oder für Low-Frames verwendet. Sollten keine High-Frames da sein, so können auch Low-Frames gesendet werden. Dies wiederholt sich in jedem Zeitschlitz. Mit diesem Mechanismus wird verhindert, dass Low-Frames nicht gesendet werden, wenn ständig High-Frames zum Senden bereitstehen.

Die Grösse einer Phase kann in Bytes selber gesetzt werden und jede zweite Phase ist die vorhin erwähnte Grün-Phase. Anhand der gesetzten Grösse in Bytes und der vorhandenen Übertragungsrate wird eine Zeitdauer berechnet, die dann die effektive Phasengrösse als Zeitfenster beträgt.

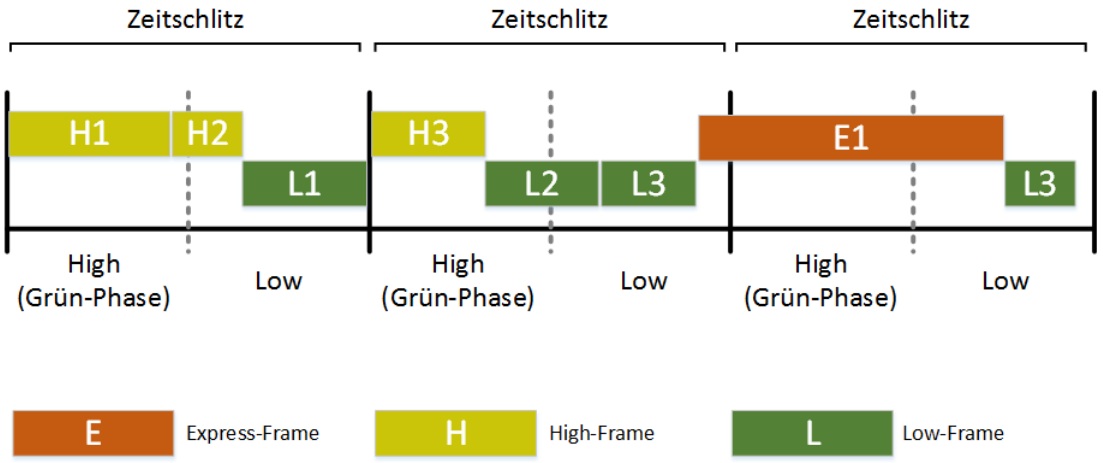


Abbildung 3.1.: Beispiel von Frame-Abfolge mit Zeitschlitzverfahren

3.1.3. Definition der Knoten

Jede Komponente eines Knotens (Netzwerkinterface, Switch, etc.) ist eine eigene C++-Klasse, welche die jeweiligen Funktionen enthält. Ein Knoten wird dann aus diesen Komponenten zusammengesetzt, indem eine sogenannte NED-Datei (NED steht für Network Description) erstellt wird, in welcher die Komponenten initialisiert und verbunden werden [11]. Der Knoten selbst existiert nicht als C++-Klasse, sondern lediglich als NED-Datei.

Der Aufbau eines DANH-Knoten in OMNeT++ sieht wie folgt aus:

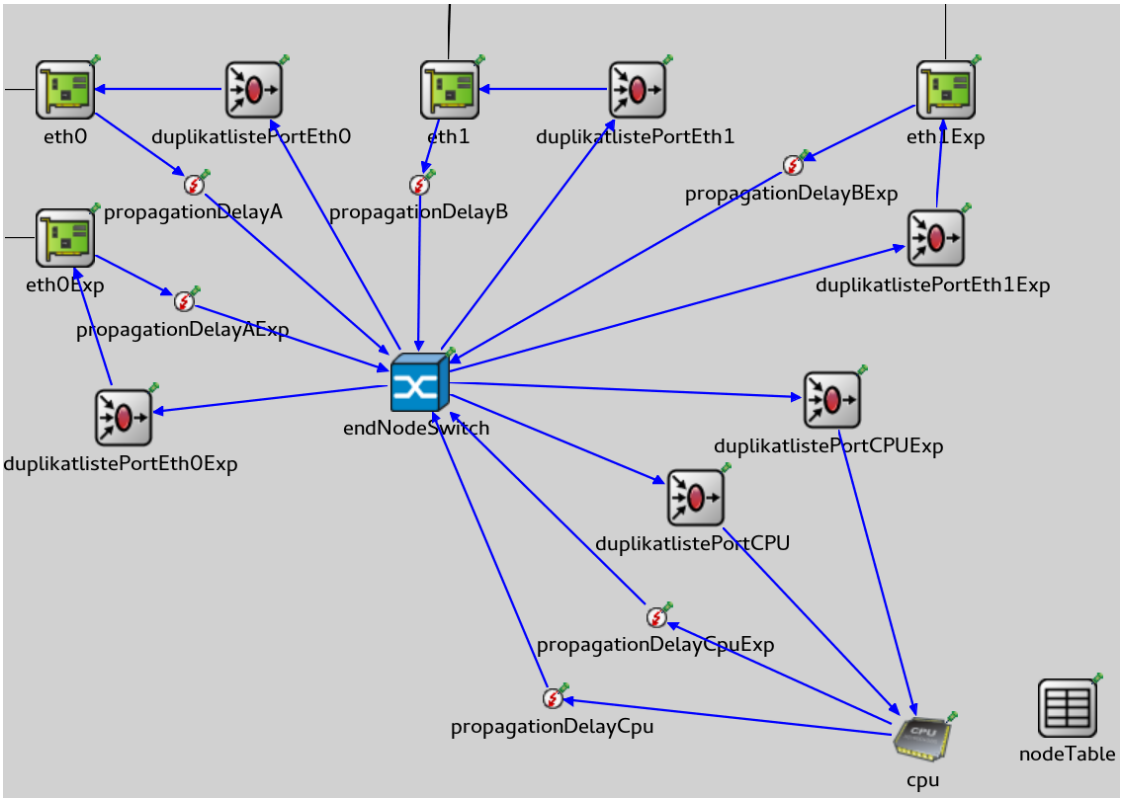


Abbildung 3.2.: DANH-Knoten in OMNeT++

Es gibt in einem DANH-Gerät 3 Ports und für jeden Port gibt es zwei Netzwerkinterfaces, einen für normale und einen für Express-Frames. Der genaue Grund, warum für jeden Port zwei Netzwerkinterfaces existieren ist im Kapitel 3.1.5.2 auf Seite 30 aufgeführt. Zwei Ports (im Bild «eth0» und «eth1») sind für den Anschluss an den Ring. Der dritte Port ist mit der CPU verbunden, die für die Generierung der neuen Frames verantwortlich ist.

Zwischen jedem Port und Komponente befindet sich eine Duplikatliste, durch die jedes Frame hindurch läuft. Die Duplikatliste überprüft anhand der Sequenznummer und der Quell-MAC-Adresse, ob dasselbe Frame schon einmal die Duplikatliste passiert hat. Wenn dasselbe Frame bereits einmal von der Duplikatliste erkannt wurde wird es gelöscht. Dank dieser Duplikatliste wird kein Frame mehrmals über den selben Port versendet.

Alle Frames, die beim DANH-Knoten ankommen oder von dessen CPU generiert werden, gelangen zum internen Switch. Dieser analysiert das Frame und weist es dem entsprechenden

Scheduler zu, der die Frames je nach Priorität und aktiven Mechanismen in die entsprechende Warteschlange stellt. Dem Scheduler ist immer ein Port (also zwei Netzwerkinterfaces) zugeteilt, was heisst, dass ein Switch über 3 Scheduler verfügt. Der Scheduler ist zudem für das Senden der Frames zuständig. Er entscheidet wann über welches Netzwerkinterface welches Frame versendet wird.

Des Weiteren verursacht ein Knoten eine Verzögerung von $6\mu s$ pro Frame (Vorgabe von unserem Betreuer, siehe Kapitel 12.2.11 auf Seite 133). Dies ist die Zeit, die ein Knoten benötigt, um beim Cut-Through-Switching genug vom Frame eingelesen zu haben, damit er Entscheiden kann, was er mit dem Frame machen soll.

3.1.4. Netzwerkaufbau

So wie ein Knoten als NED-Datei seinen Aufbau mit den Komponenten beschreibt, wird ein Netzwerk auch als NED-Datei definiert, in welchem die Knoten initialisiert und verbunden werden. Verbunden werden die Knoten über eine 20m lange 100Mbps-Verbindung, welche in der Simulation durch eine Verzögerung von 100ns dargestellt wird. Das folgende Bild zeigt ein Beispielaufbau mit 10 DANH-Knoten, die als Merging und Protection Units definiert wurden (mehr dazu siehe Kapitel 3.3.1 auf Seite 49):

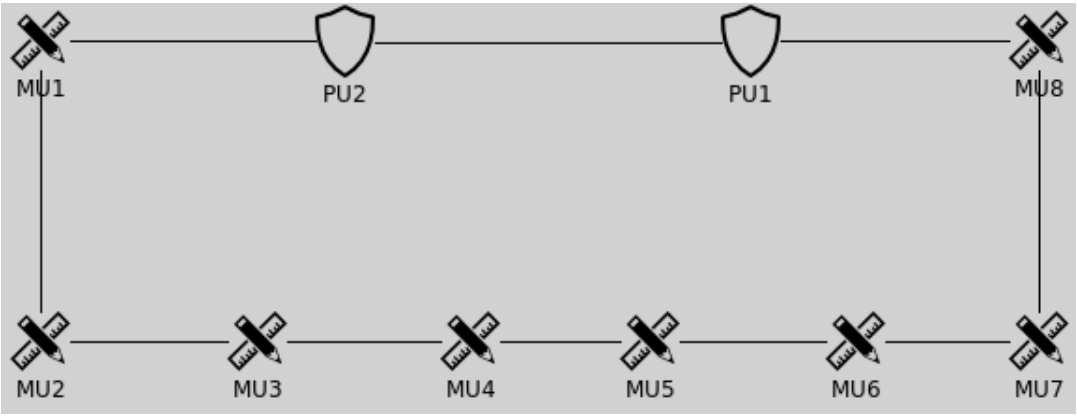


Abbildung 3.3.: Beispielaufbau eines Netzwerks in OMNeT++

Das Verhalten in diesem Beispiel ist bei jedem Knoten das Selbe, da es sich lediglich um DANH-Knoten handelt. Was variiert, sind die MAC-Adressen (in der NED-Datei des Netzwerks definiert) und die Anzahl und Häufigkeit der Frames, die generiert werden. Für die Generierung von Frames wird eine XML-Datei definiert, bei der für jeden Knoten bestimmt wird, wie viele Frames dieser Knoten an welchen Host sendet. Das Generieren von Multi- und Broadcast-Frames ist neben Unicast-Frames auch möglich.

Während der Simulation wird die Komponente, die gerade eine Aktion durchführt, rot umrandet. Zudem verändert sich die Farbe der Verbindungen und der Netzwerkinterfaces innerhalb der Knoten («eth...», siehe Abbildung 3.2 auf Seite 26). Die Farben haben folgende Bedeutungen:

Gelb	Am übertragen
Rot	Kollision erkannt (wird in den Simulationen dieser Arbeit nicht vorkommen, da Vollduplex-Verbindungen verwendet werden)

Tabelle 3.1.: Bedeutung der Farben bei den Verbindungen und Netzwerkinterfaces in der Simulation [10]

Es folgt ein Beispiel einer XML-Konfiguration für den Lastgenerator eines bestimmten Knoten:

```
<!--
  ########################################################################
   Tag-description (see current configurations for syntax)
  ########################################################################
      source: MAC-Address of source {
          lastmuster {
              startzeit:       Time when the first frame will be generated
              stopzeit:        Generate frame until this time is reached
              interval:        Generate a frame every x seconds
              epsilon:         Generate a random value between 0 and epsilon
                               that will be added to the value of the
                               previous tag every time a new frame
                               will be generated
              destination:     Set destination of frame
                               (can be Uni-, Multi- or Broadcast)
                               The broadcast-address is FF-FF-FF-FF-FF-FF
              paketgroesse:    Set datasize of frame in Bytes
                               (Size + 24 = Total Framesize)
                               Example: the total framesize is
                               160 Bytes when 136 is set
              priority:        Set priority of frame
                               ("EXPRESS", "HIGH", "LOW")
          }
      }
-->

  <!-- # Express-Frames with unicast destination # -->

  <source>00-15-12-14-88-01
      <lastmuster>
          <startzeit>0.001</startzeit>
          <stopzeit>0.0011</stopzeit>
          <interval>0.00025</interval>
          <epsilon>0</epsilon>
          <destination>00-15-12-14-88-02</destination>
          <paketgroesse>136</paketgroesse>
          <priority>EXPRESS</priority>
      </lastmuster>
  </source>

  <!-- # # # -->
```

Listing 3.1: Konfiguration des Lastgenerators eines Knotens

Für einen Knoten können mehrere solcher Konfigurationen gleichzeitig gesetzt werden, wenn ein Knoten z.B. Frames der Priorität High und Low versenden soll. Des Weiteren können auch Knoten im Netz ohne einen Eintrag im XML existieren. Diese Knoten generieren dann zwar selber keine Frames, leiten jedoch Frames normal weiter wie jeder andere Knoten auch.

Sind keine Frames mehr im Umlauf und werden keine mehr generiert (wenn die Simulationszeit über der Zeit im «Stopzeit»-Tag ist), dann wird die Simulation automatisch beendet. Während der Simulation wurde einiges aufgezeichnet, was nachher eingesehen und als Grafik dargestellt werden kann (Siehe Kapitel 4 auf Seite 54).

3.1.5. Abhandlung der Frames innerhalb eines Gerätes

Sobald ein Gerät ein Frame erhält, wird es in dessen Switch analysiert und die Herkunft und das Ziel ermittelt, um entscheiden zu können, an welchen Ausgang das Frame weitergeleitet werden muss. Je nach Ausgang wird das Frame dann an den jeweiligen Scheduler weitergeleitet. Muss ein Frame an alle Ethernet-Ports gesendet werden (wenn z.B. ein Frame von der CPU des Gerätes generiert wird), dann wird dies vom Switch dupliziert und an die betroffenen Scheduler gesendet. Der Scheduler ermittelt dann die Priorität (und eventuell die Herkunft) des Frames, um diese dann in die entsprechende Queue einzuordnen. Die Herkunft ist nur je nach Mechanismus notwendig, wenn z.B. Frames, die vom Ring kommen, priorisiert werden (siehe Kapitel 3.1.2.3 auf Seite 23).

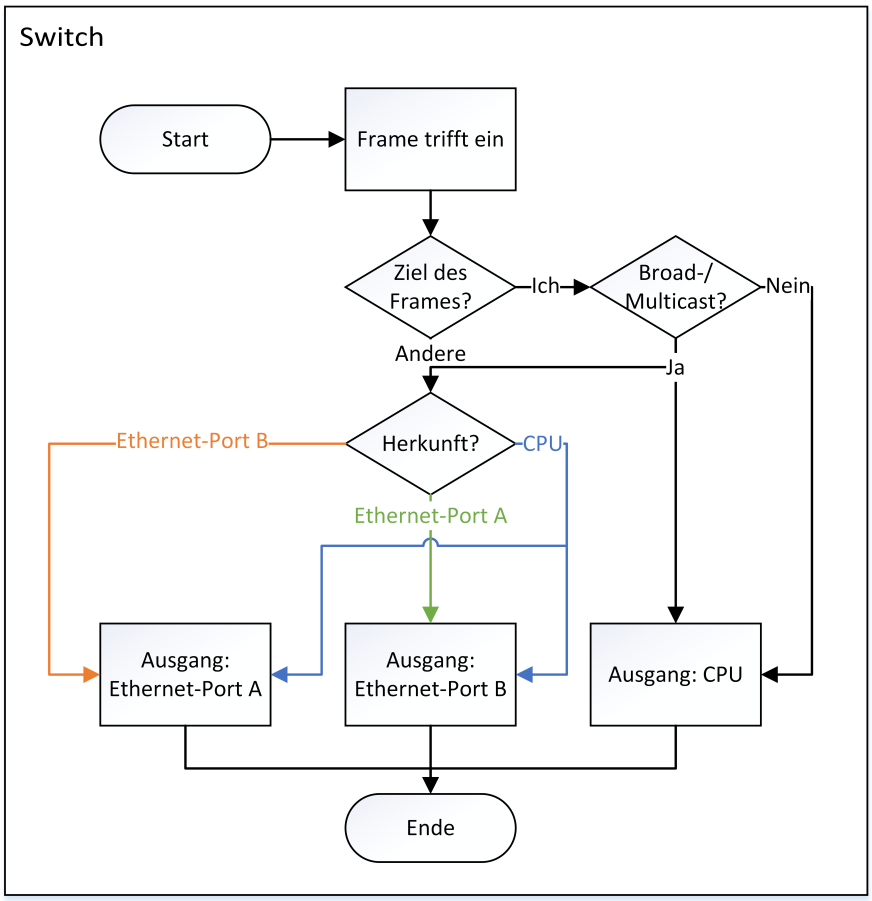


Abbildung 3.4.: Schedulerzuordnung eines Frames in einem Switch

Der Scheduler arbeitet nach jedem neuen Frame seine Queues ab. Dabei überprüft er, ob auf seinem Ausgang gerade etwas gesendet wird und sendet bei einem freien Ausgang das nächste Frame, das je nach Priorität und Mechanismus an der Reihe ist. Wird gerade etwas auf dem Ausgang übertragen, versucht es der Scheduler erst nochmals sobald die Übertragung zu Ende ist. Die Ausnahme bildet das Express-Frame, das den Sendevorgang normaler Frames unterbrechen kann.

3.1.5.1. Express-Frame

Bei einem Express-Frame muss der Scheduler sofort handeln. Anders als bei einem normalen Frame, muss das Express-Frame auch bei einem besetzten Ausgang versendet werden, was impliziert, dass das Frame, das gerade übertragen wird, wenn möglich fragmentiert werden muss.

Wichtig ist, dass kein Fragment unter der kleinstmöglichen Ethernet-Frame-Grösse von 64 Bytes liegt. Aus diesem Grund kann es sein, dass Frames manchmal nicht fragmentiert werden können und der Scheduler warten muss, bis dieses versendet wurde.

Kann das Frame jedoch fragmentiert werden, so wird das derzeitige Fragment zu Ende gesendet und anschliessend das Express-Frame versendet. Danach werden die restlichen Fragmente versendet.

3.1.5.2. Fragmentierung und Zeitberechnung

In der Simulation findet keine richtige Fragmentierung statt, jedoch wird diese mittels Zeitberechnung durchgeführt. Das heisst, dass ein Express-Frame bei einem belegten Ausgang gesendet wird, wenn das Fragment inklusive IFG versendet worden wäre. Die Ankunftszeit des «fragmentierten» Frame wird dann um die Dauer des Express-Frames inkl. dessen IFG verlängert.

Da keine wirkliche Fragmentierung stattfindet, ist der Ausgang, bei dem das zu unterbrechende Frame gesendet wird, immer noch belegt. Aus diesem Grund gibt es für jeden Ausgang einen weiteren Ausgang ausschliesslich für Express-Frames.

Somit sind die Ankunfts- und Sendezeiten der Frames dieselben, wie wenn eine tatsächliche Fragmentierung stattgefunden hätte.

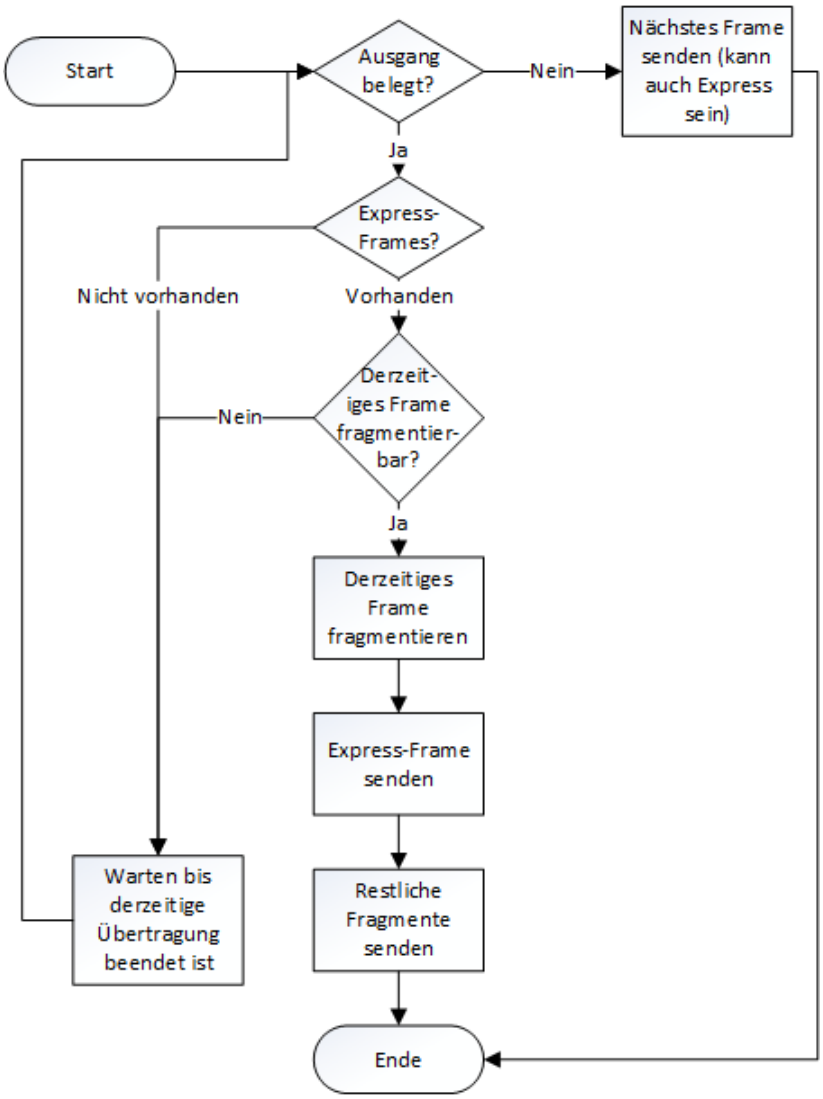


Abbildung 3.5.: Abhandlung von Frames inklusive Express-Frames

Der Datenbereich eines mFrames umfasst vom ersten Oktett nach dem SFD/SMD-Feld bis und mit dem letzten Oktett vor dem CRC und hat mindestens eine Grösse von 60 Bytes. Dabei muss dessen Grösse durch 8 (Alignment der Fragmentierung) teilbar sein (siehe Kapitel 2.4 auf Seite 18).

Natürlich kann bei einem Frame, das gerade übertragen wird nur noch der Teil fragmentiert werden, der aussteht.

Demnach wird bei der Fragmentierung wie folgt vorgegangen:

Variablen:

$$simtime = Derzeitige\,Simulationszeit\,(Jetztiger\,Zeitpunkt)\,[s]$$
$$sendtime = Zeit,\,an\,der\,die\,\ddot{U}bertragung\,der\,Präambel\,begonnen\,hat\,[s]$$
$$F = Grösse\,des\,zu\,fragmentierenden\,Frames\,ohne\,Präambel\,und\,SMD\,[Byte]$$
$$V = Bereits\,versendete\,Bytes\,inkl.\,Präambel\,und\,SMD\,[Byte]$$
$$O = Noch\,offen\,stehender\,Teil\,des\,zu\,fragmentierenden\,mFrames\,[Byte]$$
$$D = Grösse\,des\,Datenbereichs\,eines\,mFrames,\,wenn\,man\,jetzt\,fragmentieren\,würde\,[Byte]$$
$$t = Zeitpunkt,\,zu\,dem\,das\,Express\,Frame\,versendet\,wird\,[s]$$

Konstanten:

$$datarate = \ddot{U}bermittlungsrate\,[Byte/s]$$
$$P = Präambel\,+\,Start\,of\,Frame\,bzw.\,mFrame\,Delimiter = 8\,Bytes$$
$$C = FCS\,bzw.\,MFCS = 4\,Bytes$$
$$I = Interframe\,Gap\,(IFG) = 12\,Bytes$$
$$A = Alignment\,der\,Fragmentierung = 8\,Bytes$$

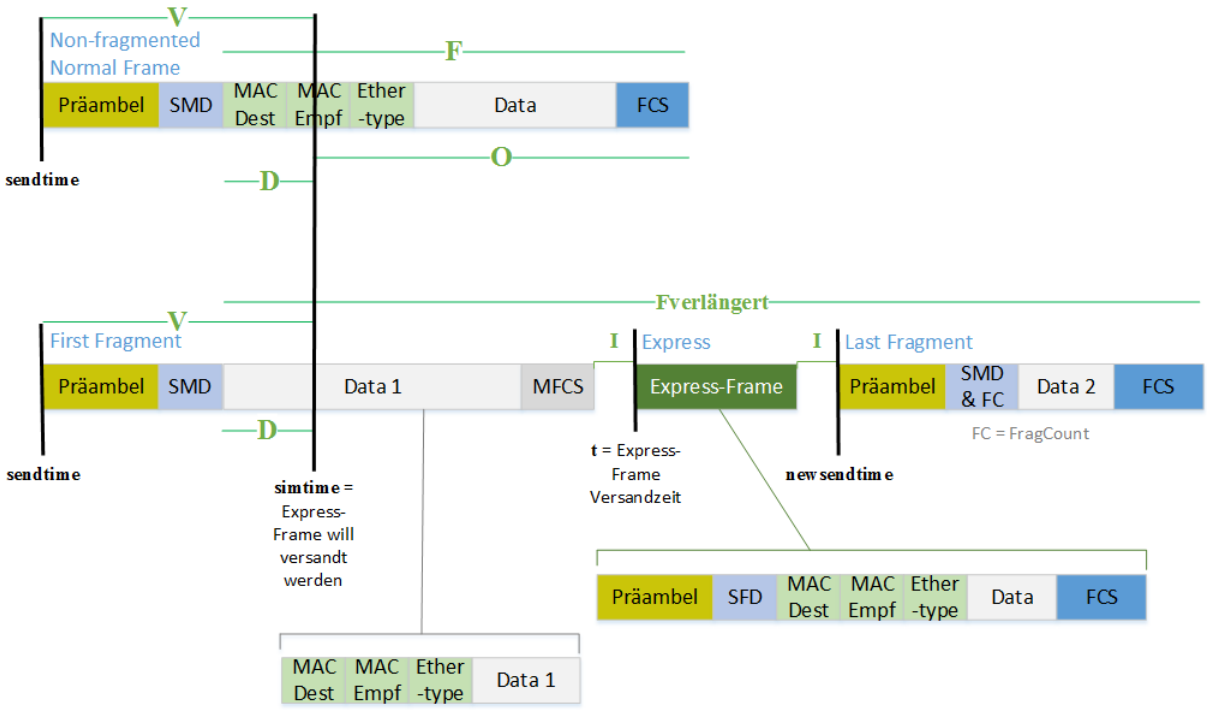


Abbildung 3.6.: Berechnung der Express-Frame-Versandzeit

Mehr Informationen zu den Feldern dieser Abbildung sind in Kapitel 2.4 auf Seite 18 und Abbildung 2.3 auf Seite 18 zu finden.

1. Express-Frame muss versendet werden.

2. Hole Grösse des gesamten Frames «F», das gerade gesendet wird.
   Beträgt «F» weniger als 128 Bytes (64 Bytes Data für das erste Fragment + 60 Bytes Data für das zweite Fragment + «C» des gesamten Frames zusammen), kann das Frame nicht fragmentiert und der weitere Vorgang abgebrochen werden. Das heisst, dass man mit dem Senden des Express-Frames warten muss, bis das nicht fragmentierbare Frame und dessen IFG fertig versendet wurden.

3. Berechne «V» und «D»:

$$V = (simtime - sendtime) * datarate$$

$$D = V - P - C - I = V - 24$$

4. Berechne «O» und rechne die «C», die pro neuem Fragment dazukommt, hinzu. Des Weiteren muss «P» aus «V» nicht mit abgezogen werden, da die Präambel und SFD/SMD nicht zur Framegrösse gehören:

$$O = F - V + C + P = F - V + 12$$

5. Wenn «V» grösser gleich 72 Bytes (Präambel + SMD + Data + (M)FCS) und IFG (von der Simulationsumgebung vorgegeben) zusammen beträgt, «D» durch 8 geteilt 0 ergibt und «O» grösser als 64 Bytes ist (Data-Felder je mindestens 60 Bytes + 4 Bytes (M)FCS), kann das Express-Frame jetzt gesendet werden. Die Schritte 6 und 7 müssen nicht mehr erledigt werden, da das Express-Frame versendet wurde und der Vorgang somit erfolgreich war. In diesem Fall beträgt «t»:

$$t = simtime$$

   Da es sich hier um eine Zeitberechnung haltet, wird das normale Frame nicht wirklich fragmentiert, sondern verlängert. Die Ankunftszeit des verlängerten Frames ist die gleiche wie wenn alle Fragmente des Frames ankommen würden.

   Das «fragmentierte» Frame wird wie folgt verlängert:
   Grösse normales Frame «F»
   + MFCS des ersten Fragments («C»)
   + IFG des ersten Fragments («I»)
   + Präambel und SFD des Express-Frames («P»)
   + Grösse Express-Frame
   + IFG des Express-Frames («I»)
   + Präambel und SMD des nächsten Fragments («P»)
   = Totale Grösse des verlängerten Frames «Fverlängert» (siehe vorherige Abbildung)

$$Fverlängert = F + C + I + P + (Grösse\,Express\,Frame) + I + P$$

$$= F + C + 2 * I + 2 * P + (Grösse\,Express\,Frame) = F + 44 + (Grösse\,Express\,Frame)$$

Die «sendtime» des fragmentierten Frames wird nun auf die Zeit gesetzt, an der das Express-Frame + IFG fertig versendet wurde («newsendtime», siehe Abbildung 3.6 auf Seite 32). So wird sichergestellt, dass wenn das Frame erneut fragmentiert werden muss, es ab Beginn des nächsten Fragments betrachtet wird. So können keine zu kleinen Fragmente entstehen.

$$newsendtime = Zeitpunkt,\, zu\, dem\, Express\, Frame + I\, fertig\, versandt\, wurden$$

$$sendtime = newsendtime$$

Somit kann dann bei der erneuten Fragmentierung das nächste Fragment so betrachtet werden, als wäre es das erste Fragment des Frames (die Framegrösse muss jedoch dementsprechend angepasst werden, nur noch das Fragment mit dem «Data 2» Feld in Abbildung 3.6 auf Seite 32 betrachten).

6. Wenn «V» nicht grösser gleich 72 Bytes und IFG zusammen beträgt oder «D» bei der Division durch 8 nicht 0 ergibt, «O» aber grösser gleich 64 Bytes ist, dann muss abgewartet werden, bis folgender Zeitpunkt «t» erreicht ist:

$$t = \begin{cases} (V \geqq 72 + I)\ \&\ (D\%8 = 0) & simtime \\ (V < 72 + I)\ \&\ (D\%8 = 0) & simtime + \frac{72+I-V}{datarate} \\ (V \geqq 72 + I)\ \&\ (D\%8\,! = 0) & simtime + \frac{8-(D\,\%\,8)}{datarate} \\ (V < 72 + I)\ \&\ (D\%8\,! = 0) & simtime + \frac{72+I-V+(8-(D\,\%\,8))}{datarate} \end{cases}$$

Diese Formel berechnet den nächsten Zeitpunkt, ab dem eine Fragmentierung möglich wäre. Ist «V» schon über oder gleich der Minimalgrösse (72 Bytes) + IFG, muss lediglich abgewartet werden bis «D» durch 8 dividiert 0 ergibt. Ist «V» kleiner als die Minimalgrösse + IFG, muss abgewartet werden, bis diese erfüllt ist und die selben Bedingungen erfüllt sind wie wenn «V» die Minimalgrösse + IFG schon erreicht hat.

7. Ist der Zeitpunkt erreicht, muss die Überprüfung wieder von Anfang an gemacht werden.

3.1.6. Generierung von Traffic (Lastprofile)

In jedem Knoten ist ein Generator implementiert, welcher mittels Konfigurationsparameter Frames in verschiedenen Folgemustern generieren kann. In diesem Kapitel werden verschiedene Muster von Traffic (Lastprofile) aufgezeigt, die in der Simulation generiert werden können.

Die Lastprofile teilen dabei folgende Eigenschaften bezüglich der Prioritäten der Frames:

- Express-Frames sind klein und kommen selten vor. Sie sollen Alarme in einem System darstellen, welche schnellstmöglich übermittelt werden müssen.
- Frames der Priorität «High» stellen die zu übertragenden Messwerte mit bestimmter Grösse dar.
- Die Frames mit der Priorität «Low» bilden den Background-Traffic, der oft vorkommt und wie der normale Verkehr verschieden gross sein kann. Dies kann unter anderem Down- und Upload von Dateien, Monitoring, etc. sein.

Bezüglich der Auftretenswahrscheinlichkeit der Frames lässt sich demnach folgendes sagen: Je höher die Priorität, desto seltener kommt das Frame vor.

Ein Beispiel eines Lastprofils ist in Kapitel 3.3.1 auf Seite 49 aufgeführt.

3.2. Überprüfung der Implementation

Damit man sich sicher sein kann, dass die Simulation wie geplant verläuft, wird die Implementation vor der Simulation der Szenarien überprüft. Diese Überprüfungen können in der mit dieser Arbeit mitgelieferten Umgebung auch selber durchgeführt werden. Beim Starten der Simulationsumgebung wird jeweils nach einer Konfiguration gefragt. Dieser Konfigurationsname wird beim entsprechenden Testfall in der Zeile «Konfig.» aufgelistet.

3.2.1. Aufbau der Testumgebung

Für die Tests wird in der Simulation ein HSR-Ring mit 3 DANH-Geräten («Node1», «Node2» und «Node3») implementiert.

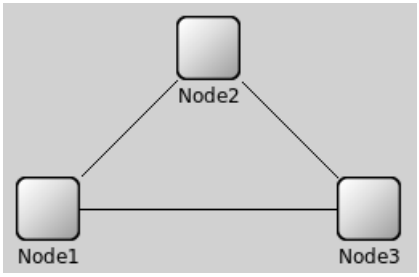


Abbildung 3.7.: HSR-Ring, in dem die Verhaltensüberprüfung statt findet

3.2.2. Verhaltensüberprüfung

Für den Nachweis der Tests wird der Nachrichtenverlauf der jeweiligen Simulation verwendet oder eine Grafik, die den Queueverlauf (wann sind wie viele Frames mit welcher Eigenschaft bei einem Ausgang vorhanden) aufweist, angezeigt. Wie diese Grafiken zu lesen sind wird in Kapitel 4 auf Seite 54 erklärt. Im «Name»-Feld des Frames im Nachrichtenverlauf steht der Name des Knotens, der das Frame generiert hat, die Priorität und die Sequenznummer des Frames. Am Anfang des «Name»-Feldes steht zudem, ob es sich um ein Unicast-, Multicast- oder Broadcast-Frame handelt.

Wenn «Node1» am Anfang der Simulation 2 Multicast-Frames hintereinander mit der Priorität High generiert, würden dessen Namen wie folgt lauten:

- «Multi From: Node1 (High) Seq#0»
- «Multi From: Node1 (High) Seq#1»

Der Namen des Frames bleibt immer gleich, auch wenn z.B. «Node2» dieses weiterleitet. Die Zeiten im Nachrichtenverlauf zeigen die Zeitpunkte, zu dem die Frames den Knoten verlassen haben (Generierungszeitpunkt + Interner Delay + Verzögerung durch andere Frames) auf.

Des Weiteren ist bei der Zeit folgendes zu beachten: Der Zeitpunkt, zu dem ein Frame generiert wird ist nicht der Sendezeitpunkt, da jeder Knoten intern einen Delay von $6\mu s$ hat (siehe Kapitel 3.1.3 auf Seite 26). Wenn ein Frame eines Knotens zum Zeitpunkt t1=0.001 generiert wird, wird es zum Zeitpunkt t2=0.001006 versendet, wenn gerade nichts anderes versendet wird.

Sofern nicht anders erwähnt, ist der verwendete Mechanismus «First Come First Serve».

3.2.2.1. Frames richtig weiterleiten und empfangen

Konfig.	Verh_ueberpruef_01_Weiterleiten_und_Empf
Soll	Ein Frame wird von einem Knoten, der nicht der einzige Empfänger des Frames ist, an den nächsten Knoten weitergeleitet. Das Frame wird bei einem Unicast-Frame vom Empfänger und bei einem Multi-/Broadcast-Frame vom Sender vom Netz entfernt.
Ist	Der sich im Knoten befindende Switch analysiert die Quell- und Ziel-Adresse und den Herkunftsport des Frames und leitet es an den entsprechenden Port (links, rechts auf den Ring oder zur CPU) weiter. Frames, welche vom eigenen Switch stammen und nicht von der CPU her kommen werden entfernt, um zirkulierende Frames (können Uni-, Multi- oder Broadcast sein) zu verhindern. Multi- und Broadcast-Frames, die nicht vom Knoten selbst stammen, werden an alle Ports ausser dem Herkunftsport weitergeleitet.
Nachweis	Folgende Frames werden mit der selben Priorität High generiert: • 1x Unicast-Frame von «Node1» zu «Node2» (Generiert an t=0.001) • 1x Multicast-Frame von «Node1» (Generiert an t=0.002) • 1x Broadcast-Frame von «Node1» (Generiert an t=0.003) Die Merkmale der Frames sind bis auf die oben erwähnten Eigenschaften gleich (Herkunft, Grösse, Priorität). Der verwendete Mechanismus ist «First Come First Serve».

Event#	Time	Src/Dest	Name
#9	0.001006	Node1 --> Node2	Uni From: Node1 (HIGH) Seq#0
#10	0.001006	Node1 --> Node3	Uni From: Node1 (HIGH) Seq#0
#32	0.00102554	Node3 --> Node2	Uni From: Node1 (HIGH) Seq#0
#52	0.002006	Node1 --> Node2	Multi From: Node1 (HIGH) Seq#1
#53	0.002006	Node1 --> Node3	Multi From: Node1 (HIGH) Seq#1
#78	0.00202554	Node2 --> Node3	Multi From: Node1 (HIGH) Seq#1
#80	0.00202554	Node3 --> Node2	Multi From: Node1 (HIGH) Seq#1
#106	0.00204508	Node3 --> Node1	Multi From: Node1 (HIGH) Seq#1
#107	0.00204508	Node2 --> Node1	Multi From: Node1 (HIGH) Seq#1
#138	0.003006	Node1 --> Node2	Broad From: Node1 (HIGH) Seq#2
#139	0.003006	Node1 --> Node3	Broad From: Node1 (HIGH) Seq#2
#164	0.00302554	Node2 --> Node3	Broad From: Node1 (HIGH) Seq#2
#166	0.00302554	Node3 --> Node2	Broad From: Node1 (HIGH) Seq#2
#192	0.00304508	Node3 --> Node1	Broad From: Node1 (HIGH) Seq#2
#193	0.00304508	Node2 --> Node1	Broad From: Node1 (HIGH) Seq#2

Wie man der Grafik entnehmen kann, werden Unicast-Frames beim Empfänger und Multi-/Broadcast-Frames beim Sender gelöscht. Des Weiteren werden die Frames korrekt weitergeleitet.

Tabelle 3.2.: Verhaltensüberprüfung: Frame-Ablauf

3.2.2.2. Beachten der Priorisierung

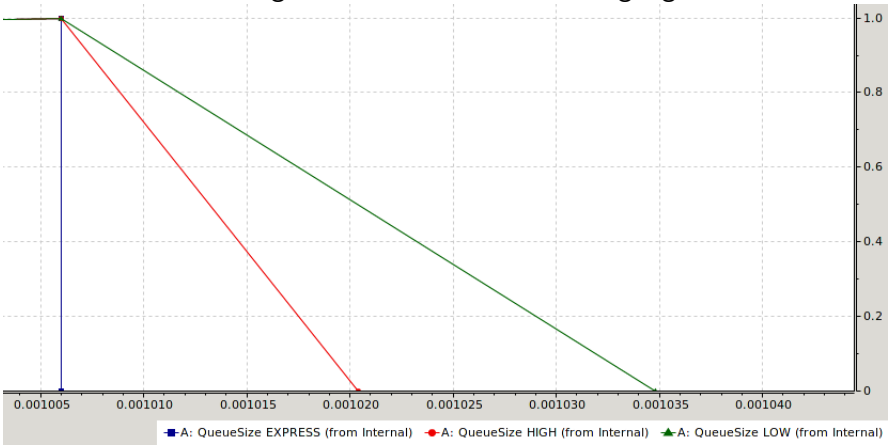
Konfig.	Verh_ueberpruef_02_Priorisierung
Soll	Wenn drei Frames mit unterschiedlichen Prioritäten ankommen, soll das mit der höheren Priorität vor dem anderen verarbeitet werden.
Ist	Der sich im Knoten befindende Switch hat für jeden seiner Ports einen Scheduler. Der Switch fügt ein ankommendes Frame beim jeweiligen Scheduler in die jeweilige Queue ein. Der Scheduler beginnt darauf mit dem Abarbeiten der Queues und startet mit der Queue der höchsten Priorität. Wenn mehrere Frames mit unterschiedlichen Prioritäten zur selben Zeit ankommen, wird garantiert das Frame mit der höheren Priorität zuerst versendet.
Nachweis	Folgende Frames werden zur exakt selben Zeit generiert (t=0.001):

- 1x Unicast-Frame von «Node1» zu «Node2» mit Priorität Express
- 1x Unicast-Frame von «Node1» zu «Node2» mit Priorität High
- 1x Unicast-Frame von «Node1» zu «Node2» mit Priorität Low

Die Merkmale der Frames sind bis auf die oben erwähnten Eigenschaften gleich.

Event#	Time	Src/Dest	Name
#21	0.001006	Node1 --> Node2	Uni From: Node1 (EXPRESS) Seq#0
#22	0.001006	Node1 --> Node3	Uni From: Node1 (EXPRESS) Seq#0
#37	0.001020399999	Node1 --> Node2	Uni From: Node1 (HIGH) Seq#1
#38	0.001020399999	Node1 --> Node3	Uni From: Node1 (HIGH) Seq#1
#48	0.00102554	Node3 --> Node2	Uni From: Node1 (EXPRESS) Seq#0
#63	0.001034799998	Node1 --> Node2	Uni From: Node1 (LOW) Seq#2
#64	0.001034799998	Node1 --> Node3	Uni From: Node1 (LOW) Seq#2
#80	0.001039939999	Node3 --> Node2	Uni From: Node1 (HIGH) Seq#1
#112	0.001054339998	Node3 --> Node2	Uni From: Node1 (LOW) Seq#2

Die nächste Grafik zeigt den Ablauf bei einem Ausgang von «Node1»:



Hier ist ersichtlich, dass alle 3 Frames zur gleichen Zeit eingereiht wurden, jedoch die Zeitpunkte, an denen die Frames die Queues verlassen haben, unterschiedlich sind. Des Weiteren kann man feststellen, dass die Frames ihrer Priorität entsprechend behandelt werden.

Tabelle 3.3.: Verhaltensüberprüfung: Beachten der Priorisierung

3.2.2.3. Express-Frames und Fragmentierung

Konfig.	Siehe jeweiliges Szenario in den nächsten 3 Tabellen
Soll	Soll ein Express-Frame versendet werden, so wird dies sofort versandt, wenn derzeit keine anderen Frames auf demselben Ausgang versendet werden. Wird jedoch bereits ein High- oder Low-Frame darauf gesendet, so wird das Express-Frame zu dem Zeitpunkt versandt, an dem ein Fragment des anderen Frames (inklusive IFG) fertig versendet worden wäre. Kann das normale Frame jedoch nicht fragmentiert werden, kann das Express-Frame erst nach dem normalen Frame versendet werden. Die Berechnung zur Fragmentierung und dem Zeitpunkt des Express-Frame-Versandes ist in Kapitel 3.1.5.2 auf Seite 30 aufgeführt.
Ist	Die Frames der Priorität High und Low werden, wenn möglich, fragmentiert und das Express-Frame wird zum korrekten Zeitpunkt versendet.
Nachweis	3 Szenarios werden für den Nachweis erstellt: • Szenario 01 (Fragmentierbares Frame) • Szenario 02 (Nicht fragmentierbares Frame) • Szenario 03 (Ein langes Frame wird mehrmals fragmentiert) Wenn ein Express-Frame unmittelbar nach einem normalen, fragmentierbaren Frame generiert wird (d.h. das normale Frame wird bereits versendet, jedoch wurde davon noch kein Byte versendet), müsste das Express-Frame laut Berechnung aus Kapitel 3.1.5.2 auf Seite 30 zu folgendem Zeitpunkt «tsendexp» versendet werden: $$tsend = Zeit,\, an\, dem\, das\, normale\, Frame\, versendet\, wurde\, [s]$$ $$tsendexp = Zeit,\, an\, dem\, das\, ExpressFrame\, versendet\, wird\, [s]$$ $$datarate = 100\, Mbit/s$$ $$t = \frac{72 + I}{datarate} = \frac{84}{(100000000/8)} = 0,00000672s$$ $$tsendexp = tsend + t$$ Für die Bedeutung der Variablen siehe vorhin erwähnte Berechnung. Da in OMNeT++ für die Zeitberechnung Double-Werte verwendet werden, kann eine gewisse Präzision verloren gehen, weshalb sich die effektiven Zeiten zur hier berechneten Zeit marginal unterscheiden können [11].

Tabelle 3.4.: Verhaltensüberprüfung: Express-Frames und Fragmentierung (Allgemein)

Konfig.	Verh_ueberpruef_03_Exp_und_Frag_01_Frag
Szenario	01: Fragmentierbares Frame
Nachweis	Folgende Frames werden generiert: • 1x Unicast-Frame von «Node1» zu «Node2» mit 128 Bytes und Priorität High (Generiert an t=0.001) • 1x Unicast-Frame von «Node1» zu «Node2» mit 100 Bytes und Priorität Express (Generiert an t=0.001001)

Event#	Time	Src/Dest	Name
#11	0.001006	Node1 --> Node2	Uni From: Node1 (HIGH) Seq#0
#12	0.001006	Node1 --> Node3	Uni From: Node1 (HIGH) Seq#0
#19	0.001012759999	Node1 --> Node2	Uni From: Node1 (EXPRESS) Seq#1
#20	0.001012759999	Node1 --> Node3	Uni From: Node1 (EXPRESS) Seq#1
#54	0.00102298	Node3 --> Node2	Uni From: Node1 (HIGH) Seq#0
#64	0.001029739999	Node3 --> Node2	Uni From: Node1 (EXPRESS) Seq#1

Wie man sehen kann beträgt «tsend» 0.001006s und «tsendexp» 0.001012759999s, was einer Differenz «t» von 0.00000676s entspricht. Dadurch, dass bei der Berechnung in OMNeT++ eine gewisse Präzision verloren geht, kann man sagen, dass «t» dem vorhin berechneten Wert (0.00000672s) entspricht und die Fragmentierung per Zeitberechnung somit bewiesen wurde.

Tabelle 3.5.: Verhaltensüberprüfung: Express-Frames und Fragmentierung (Szenario 01)

Konfig.	Verh_ueberpruef_03_Exp_und_Frag_02_NoFrag
Szenario	02: Nicht fragmentierbares Frame
Nachweis	Folgende Frames werden generiert:

- 1x Unicast-Frame von «Node1» zu «Node2» mit 127 Bytes und Priorität High (Generiert an t=0.001)
- 1x Unicast-Frame von «Node1» zu «Node2» mit 100 Bytes und Priorität Express(Generiert an t=0.001001)

Wenn das normale Frame nicht fragmentiert werden kann, müsste «t» bei einem normalen Frame mit 127 Bytes wie folgt berechnet werden:

$$t = \frac{127 + I}{datarate} = \frac{139}{(100000000/8)} = 0.00001112s$$

$$tsendexp = tsend + t$$

Event#	Time	Src/Dest	Name
#11	0.001006	Node1 --> Node2	Uni From: Node1 (HIGH) Seq#0
#12	0.001006	Node1 --> Node3	Uni From: Node1 (HIGH) Seq#0
#31	0.001017759999	Node1 --> Node2	Uni From: Node1 (EXPRESS) Seq#1
#32	0.001017759999	Node1 --> Node3	Uni From: Node1 (EXPRESS) Seq#1
#42	0.0010229	Node3 --> Node2	Uni From: Node1 (HIGH) Seq#0
#70	0.001034659999	Node3 --> Node2	Uni From: Node1 (EXPRESS) Seq#1

Wie man sehen kann, beträgt «tsend» 0.001006s und «tsendexp» 0.001017759999s, was eine Differenz «t» von 0.00001176s beträgt. Dadurch, dass bei der Berechnung in OMNeT++ eine gewisse Präzision verloren geht, kann man sagen, dass «t» dem oben berechneten Wert (0.00001112s) entspricht und das Szenario somit bewiesen wurde.

Tabelle 3.6.: Verhaltensüberprüfung: Express-Frames und Fragmentierung (Szenario 02)

Konfig.	Verh_ueberpruef_03_Exp_und_Frag_03_Long_Mult
Szenario	03: Ein langes Frame wird mehrmals fragmentiert
Nachweis	Folgende Frames werden generiert:

- 1x Unicast-Frame von «Node1» zu «Node2» mit 1500 Bytes und Priorität High (Generiert an t=0.001)
- 2x Unicast-Frame von «Node1» zu «Node2» mit 100 Bytes und Priorität Express(Generiert an t1=0.001001 und t2=0.001002)

Das zweite Express-Frame wird unmittelbar nach dem ersten Express-Frame generiert, weshalb das zweite Express-Frame direkt nach dem ersten Express-Frame versendet wird. Die beiden Express-Frames müssten zu folgenden Zeiten «tsendexp1» und «tsendexp2» versendet werden:

$$t = \frac{72 + I}{datarate} = \frac{84}{(100000000/8)} = 0,00000672s$$

$$texp = \frac{100 + I}{datarate} = \frac{112}{(100000000/8)} = 0,00000896s$$

$$tsendexp1 = tsend + t$$

$$tsendexp2 = tsend + t + texp = tsend + 0,00001568s$$

Event#	Time	Src/Dest	Name
#14	0.001006	Node1 --> Node2	Uni From: Node1 (HIGH) Seq#0
#15	0.001006	Node1 --> Node3	Uni From: Node1 (HIGH) Seq#0
#26	0.001012759999	Node1 --> Node2	Uni From: Node1 (EXPRESS) Seq#1
#27	0.001012759999	Node1 --> Node3	Uni From: Node1 (EXPRESS) Seq#1
#42	0.001022359998	Node1 --> Node2	Uni From: Node1 (EXPRESS) Seq#2
#43	0.001022359998	Node1 --> Node3	Uni From: Node1 (EXPRESS) Seq#2
#53	0.001027499999	Node3 --> Node2	Uni From: Node1 (EXPRESS) Seq#1
#81	0.001037099998	Node3 --> Node2	Uni From: Node1 (EXPRESS) Seq#2
#117	0.00113274	Node3 --> Node2	Uni From: Node1 (HIGH) Seq#0

Wie man sehen kann, beträgt «tsend» 0.001006s und «tsendexp1» 0.001012759999s, was einer Differenz «t» von 0.00000676s entspricht. «tsendexp2» beträgt 0.001022359998s, was wenn man «t» und «texp» abzieht 0.00100668s (ungefähr «tsend») beträgt.
Dadurch, dass bei der Berechnung in OMNeT++ eine gewisse Präzision verloren geht, kann man sagen, dass die simulierten Ergebnisse den berechneten Werten entsprechen und das Szenario somit bewiesen wurde.

Tabelle 3.7.: Verhaltensüberprüfung: Express-Frames und Fragmentierung (Szenario 03)

3.2.2.4. Zuflusslimitierung

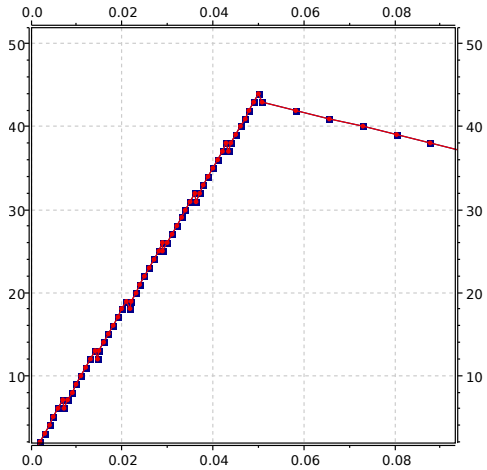
Konfig.	Verh_ueberpruef_04_Zuflusslimitierung
Soll	Der Zufluss von neuen Frames mit der Priorität Low soll limitiert werden können. Als Limit kann eine Anzahl an Bytes, die maximal pro Sekunde von Low-Frames verbraucht werden können, festgelegt werden. Es werden lediglich die Low-Frames limitiert, die intern generiert werden.
Ist	Sofern eine Zuflusslimitierung auf x Byte/s gesetzt wurde, kann ein Knoten nur so viele Low-Frames pro Sekunde generieren und versenden wie Tokens generiert werden können.
Nachweis	Damit in den Simulationsergebnissen zwischen den Frames vom Ring und den intern generierten unterschieden werden kann, wird folgender Mechanismus verwendet: Die intern Generierten Frames haben gegenüber den Frames vom Ring Vortritt (siehe Kapitel 3.1.2.3 auf Seite 23). Es wird eine Zuflusslimitierung von 100Kbit/s (12500 Byte/s) gesetzt. Dies bedeutet, dass intern maximal 12500 Tokens generiert werden können. Folgende Frames mit der Priorität Low werden generiert: • 50x Unicast-Frame von «Node1» zu «Node2» mit 1000 Bytes (Generiert ab t=0.001 mit einem Intervall von 0.001 bis zu tstop=0.051) Es folgt ein Graph, der die Anzahl vorhandener, intern generierter Low-Frames an einem Ausgang zu jedem Zeitpunkt darstellt:  Zu Beginn müssen die Tokens generiert werden, deshalb waren dann zu wenige Tokens verfügbar und die Frames wurden nicht direkt nach ihrer Generierung versendet. Die Frames stauten sich somit bis zum Zeitpunkt «tstop» an, nach dem keine neuen Frames mehr generiert wurden. Nach «tstop» wurden die Frames gleich schnell wie vorhin abgearbeitet.

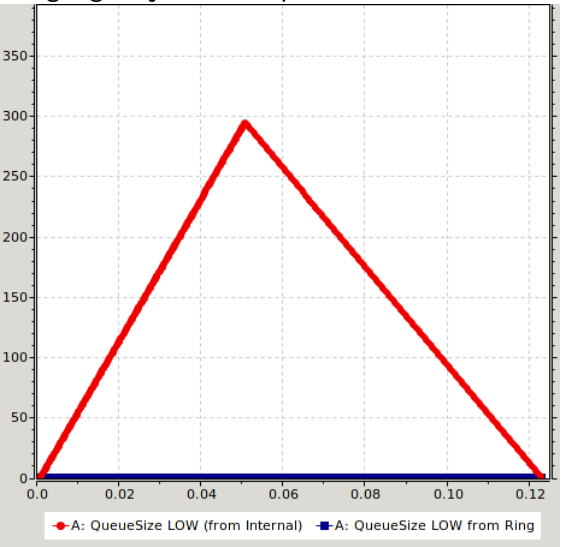
Tabelle 3.8.: Verhaltensüberprüfung: Zuflusslimitierung

3.2.2.5. Vortrittsregeln im und zum Ring

Konfig.	Siehe jeweiliges Szenario in den nächsten zwei Tabellen
Soll	Wenn zwei Frames derselben Priorität beim Knoten ankommen (ein Frame vom Ring und ein Frame intern generiert), gewährt der Knoten je nach Mechanismus dem entsprechenden Frame gegenüber dem anderen den Vortritt.
Ist	Wenn ständig Frames vom Ring her kommen und ständig welche mit der selben Priorität intern generiert werden, werden je nach Mechanismus bestimmte Frames nie weitergeleitet solange neue von der anderen Sorte kommen. Haben die Frames vom Ring den Vortritt werden die intern generierten Frames kaum versendet. Haben die Frames von Aussen den Vortritt werden die Frames vom Ring kaum weitergeleitet.
Nachweis	Zwei Szenarios werden für den Nachweis erstellt: • Szenario 01 (Frames vom Ring haben Vortritt) • Szenario 02 (Frames von Aussen intern generiert) haben Vortritt) Bei beiden Szenarios werden folgende Frames mit der Priorität Low ab t=0.001 mit einem Intervall von 0.0001 bis zu tstop=0.051 generiert: • 500x Multicast-Frame von «Node1» mit 1000 Bytes • 500x Multicast-Frame von «Node2» mit 1000 Bytes • 500x Multicast-Frame von «Node3» mit 1000 Bytes Bei jedem der drei Knoten in der Testumgebung ist der gleiche Ablauf vorzufinden, da alle drei das selbe Lastprofil zur Framegenerierung haben.

Tabelle 3.9.: Verhaltensüberprüfung: Vortrittsregeln im und zum Ring (Allgemein)

Konfig.	Verh_ueberpruef_05_Vortritt_01_Ring
Szenario	01: Frames vom Ring haben Vortritt
Nachweis	Es folgt ein Graph, der die Anzahl Low-Frames von Aussen und vom Ring an einem Ausgang zu jedem Zeitpunkt darstellt:



Wie man sehen kann, stauen sich bis zum Zeitpunkt «tstop» (0.051s) die intern generierten Frames (rot) an. Es werden zwar intern generierte Frames versendet (ansonsten hätte kein Knoten etwas zum weiterleiten), jedoch sehr wenige. Nach diesem Zeitpunkt sind die Knoten zwar noch andere Frames vom Ring am weiterleiten, jedoch generiert kein Knoten mehr neue Frames, weshalb mehr Platz entsteht, um mehr intern generierte Frames versenden zu können.

Tabelle 3.10.: Verhaltensüberprüfung: Vortrittsregeln im und zum Ring (Szenario 01)

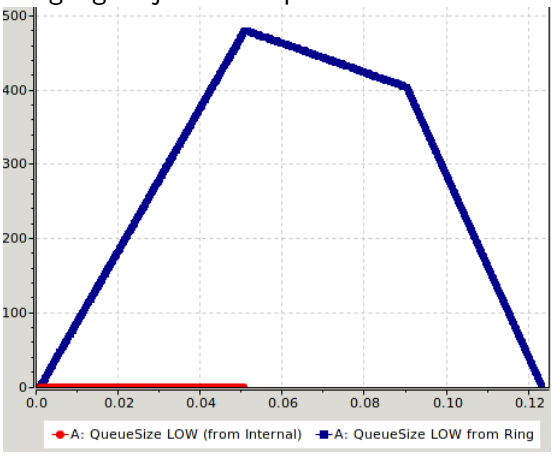
Konfig.	Verh_ueberpruef_05_Vortritt_02_Intern
Szenario	02: Frames von Aussen (intern generiert) haben Vortritt
Nachweis	Es folgt ein Graph, der die Anzahl Low-Frames von Aussen und vom Ring an einem Ausgang zu jedem Zeitpunkt darstellt:  Hier sieht man das Gegenstück zum vorherigen Szenario. Die intern generierten Frames (rot) können alle sofort versendet werden (weshalb es nach «tstop» keine mehr gibt) und die weiterzuleitenden Frames (blau) stauen sich an. Nach «tstop» werden die angestauten Frames weitergeleitet und ab ca. 0.09s hat sich das Netzwerk ein wenig «erholt» und mehr Platz geschaffen, weshalb die restlichen Frames schneller weitergeleitet werden können.

Tabelle 3.11.: Verhaltensüberprüfung: Vortrittsregeln im und zum Ring (Szenario 02)

3.2.2.6. Reissverschluss

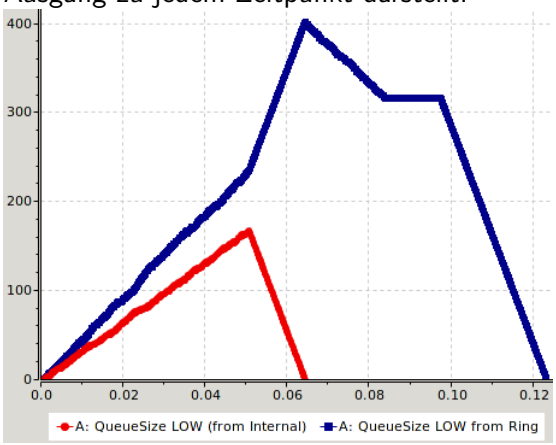
Konfig.	Verh_ueberpruef_06_Reissverschluss
Soll	Abwechslungsweise soll ein Knoten mal dem gleich priorisierten Frame vom Ring und mal von Aussen den Vortritt gewähren (siehe Kapitel 3.1.2.4 auf Seite 24).
Ist	Es wird in einem Knoten, wenn gerade einem Frame vom Ring Vortritt gewährt wurde, einem Frame von Aussen den Vortritt gewährt und umgekehrt. Anders als beim vorherigen Test (siehe Tabelle 3.9 auf Seite 44) stauen sich keine Frames an, weil ständig Frames vom anderen Typ gesendet werden. Ein Anstauen ist dennoch möglich, wenn zu viele Frames versendet werden müssen, jedoch würde es sich dann bei beiden Typen anstauen.
Nachweis	Es werden folgende Frames mit der Priorität Low ab t=0.001 mit einem Intervall von 0.0001 bis zu tstop=0.051 generiert (selbes Lastprofil wie beim vorherigen Test, siehe Tabelle 3.9 auf Seite 44): • 500x Multicast-Frame von «Node1» mit 1000 Bytes • 500x Multicast-Frame von «Node2» mit 1000 Bytes • 500x Multicast-Frame von «Node3» mit 1000 Bytes Es folgt ein Graph, der die Anzahl Low-Frames von Aussen und vom Ring an einem Ausgang zu jedem Zeitpunkt darstellt:  Man kann anhand der Grafik feststellen, dass die beiden Typen einander beeinflussen. Es ist ersichtlich, dass nach «tstop» die Anzahl interner Frames (rot) schneller abgebaut werden und somit mehr Frames vom Ring (blau) entstehen. Nachdem alle internen Frames versendet wurden (bei ca. 0.065s), können alle restlichen Frames vom Ring abgearbeitet werden. Bei jedem der drei Knoten in der Testumgebung ist der gleiche Ablauf vorzufinden, da alle drei das selbe Lastprofil zur Framegenerierung haben.

Tabelle 3.12.: Verhaltensüberprüfung: Reissverschluss

3.2.2.7. Zeitschlitzverfahren

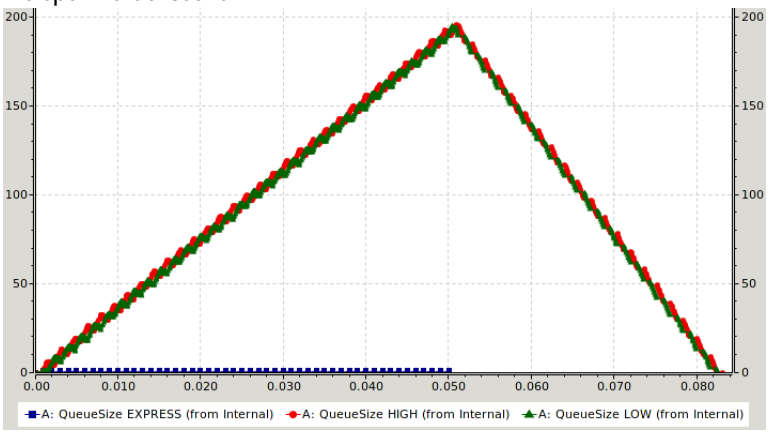
Konfig.	Verh_ueberpruef_07_Zeitschlitz
Soll	Es sollten mit dem Zeitschlitzverfahren (siehe Kapitel 3.1.2.5 auf Seite 25) trotz einer ständigen Generierung von High-Frames auch Frames der Priorität Low versendet werden, da die High-Frames nur innerhalb der definierten Grün-Phase versendet werden können.
Ist	Obwohl ständig High-Frames generiert werden, findet auch der Versand von Low-Frames statt. Gibt es keine High-Frames zum Versenden, so können Low-Frames auch in der Grün-Phase versendet werden. Express-Frames können immer versendet werden.
Nachweis	Es werden folgende Frames generiert: • 50x Unicast-Frame von «Node1» zu «Node2» mit 100 Bytes und Priorität Express (Generiert ab t=0.001 mit einem Interval von 0.001 bis zu tstop=0.051) • 500x Unicast-Frame von «Node1» zu «Node2» mit 1000 Bytes und Priorität High (Generiert ab t=0.001, Interval von 0.0001 bis zu tstop=0.051) • 500x Unicast-Frame von «Node1» zu «Node2» mit 1000 Bytes und Priorität Low (Generiert ab t=0.001, Interval von 0.0001 bis zu tstop=0.051) Die Phasengrösse wird auf 10000 Bytes gesetzt, was bei einer gegebenen Übertragungsrate von 100Mbit/s 0.0008s beträgt. Das heisst, das alle 0.0016s für 0.0008s lang High-Frames versendet werden können. Es folgt ein Graph, der die Anzahl aller Frames an einem Ausgang zu jedem Zeitpunkt darstellt:  Wie man feststellen kann, werden High- (rot) und Low-Frames (grün) gleichmässig abgebaut, obwohl von beiden Prioritäten ständig neue Frames generiert werden. Die Express-Frames können immer versendet werden, haben aber in diesem Fall keinen erwähnenswerten Einfluss auf die Low- und High-Frames.

Tabelle 3.13.: Verhaltensüberprüfung: Zeitschlitz

3.3. Simulation

In diesem Kapitel werden die Szenarien und Simulationen definiert, die in dieser Arbeit simuliert werden. Die Resultate der jeweiligen Simulation sind im Kapitel 4 auf Seite 54 aufgeführt. Diese Simulationen können in der mit dieser Arbeit mitgelieferten Umgebung auch selber durchgeführt werden. Wie diese Szenarien selber simuliert werden können, wird in der Anleitung in Kapitel 13 auf Seite 135 beschrieben. Beim Starten der Simulationsumgebung wird jeweils nach einer Konfiguration gefragt. Dieser Konfigurationsname wird bei der entsprechenden Simulation aufgelistet.

3.3.1. Szenario 1: Substation Automation

Der Aufbau dieses Szenarios ist ein HSR-Ring mit DANH-Knoten. Die Aufgabe im Anwendungsfall Substation Automation ist den Schutz von Schaltern und Leitungen sicher zu stellen.

Im HSR-Ring befinden sich 14 sogenannte Merging Units (MU), welche die Messwerte mit Sensoren erfassen und diese mit Zeitstempel und sonstigen Steuerinformationen in ein Frame verpacken. Ein solches Frame hat eine Gesamtgrösse von 160 Bytes (inklusive Header) und die Priorität High. Eine MU verschickt ca. 4000 mal pro Sekunde ein solches Frame via Multicast (Publisher / Subscriber Modell). Ziel dieser Frames sind zwei Protection Units (PU) und eventuell andere MUs. Protection Units treffen anhand der erhaltenen Messwerte Entscheidungen. MUs und PUs sind DANH-Knoten.

Neben den erwähnten Frames gibt es spontane Einzelmeldungen (Express-Frames, auch Multicast), welche selten und zufällig vorkommen. Die Grösse dieser Express-Frames beträgt total 100 Bytes.

Als Background-Traffic wird von 2 Knoten TCP-ähnlicher Traffic generiert, bei dem ein Knoten (Knoten A) alle 200 Sekunden Frames mit einer Gesamtgrösse von 1500 Bytes an den anderen Knoten (Knoten B) sendet (Unicast). Knoten B sendet zur selben Zeit auch alle 200 Sekunden Frames à 64 Bytes an Knoten A.

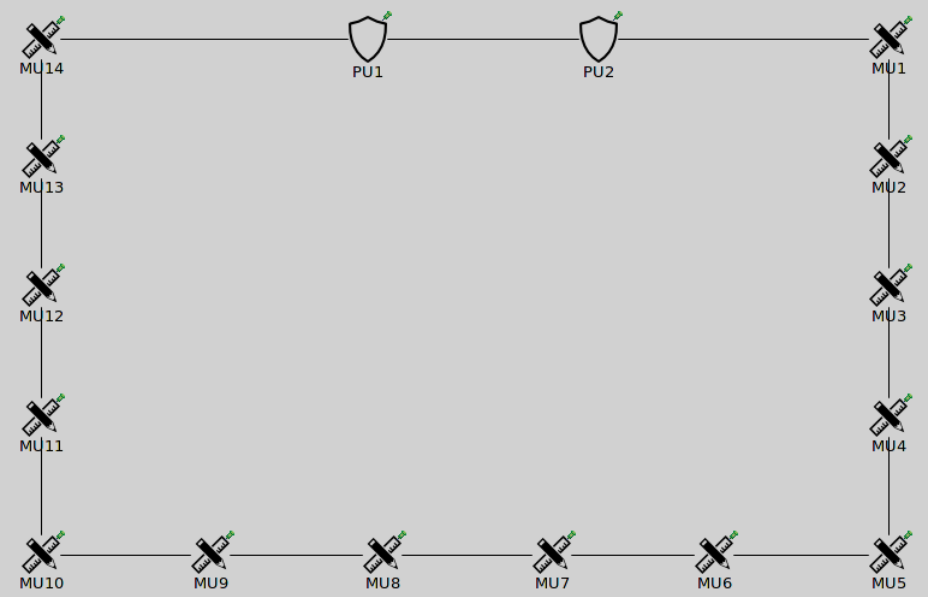


Abbildung 3.8.: Netzwerkaufbau Szenario 1: Substation Automation

Konkret wird für dieses Szenario folgendes Lastprofil angewendet:

Priorität	Quelle	Ziel	Intervall (Alle x Sekunden)	Grösse in Bytes	Zweck
Express	MU1	Multicast	0.01s + ϵ_1	100	Alarm
	MU8				
High	MU1	Multicast	0.00025s + ϵ_2	160	Messwerte
	MU2				
	MU3				
	MU4				
	MU5				
	MU6				
	MU7				
	MU8				
	MU9				
	MU10				
	MU11				
	MU12				
	MU13				
	MU14				
Low	MU5	PU1	0.005s	1500	TCP-ähnlicher Traffic
	PU1	MU5		64	

Tabelle 3.14.: Lastprofil Szenario 1: Substation Automation

- ϵ_1 = Zufallswert zwischen (0s und 0.01s)
- ϵ_2= Zufallswert zwischen (0s und 0.0000125s)

Diese Frames werden ab dem Zeitpunkt t=0.001s bis zu tstop=0.56s generiert.

Sofern es nicht anders erwähnt wird, befinden sich in den Simulationen 14 MUs und 2 PUs, die in einem Ring mit einer 100Mbps-Verbindung verbunden sind. Die Prioritäten der Frames werden in jeder Simulation berücksichtigt.

3.3.1.1. Simulation 1.1: First Come First Serve

Konfigurationsname: Scen1_Simulation_1_1_FCFS

Szenario 1 wird ohne spezielle Mechanismen in den Knoten simuliert. Es spielt in dieser Variante keine Rolle, ob ein Frame von Aussen oder vom Ring kommt.

3.3.1.2. Simulation 1.2: Vortritt für Frames vom Ring

Konfigurationsname: Scen1_Simulation_1_2_Vortritt_Ring

Jeder Knoten im Ring gewährt den Frames, die vom Ring kommen, den Vortritt gegenüber den Frames, die von Aussen kommen. Die Frames, die von Aussen kommen, sind dabei die, die vom Knoten selbst generiert werden.

3.3.1.3. Simulation 1.3: Vortritt für Frames von Aussen

Konfigurationsname: Scen1_Simulation_1_3_Vortritt_Internal

Jeder Knoten im Ring gewährt den Frames, die von Aussen kommen, den Vortritt gegenüber den Frames, die vom Ring kommen. Die Frames, die von Aussen kommen, sind dabei die, die vom Knoten selbst generiert werden. Wenn ein Knoten ständig Frames generiert, kann es sein, dass die Frames vom Ring bei diesem Knoten nie zum Zuge kommen und somit nie weitergeleitet werden.

3.3.1.4. Simulation 1.4: Vortritt für Frames von Aussen mit Zuflusslimitierung

Konfigurationsname: Scen1_Simulation_1_4_Vortritt_Int_Zuflusslim

Die Simulation 1.3 wird mit einer Zuflusslimitierung erweitert. Mit Hilfe dieser Limitierung kann ausgeschlossen werden, dass Frames vom Ring nie weitergeleitet werden, wenn ein Knoten ständig Frames generiert.

3.3.1.5. Simulation 1.5: Reissverschluss

Konfigurationsname: Scen1_Simulation_1_5_Reissverschluss

Es wird in jedem Knoten abwechslungsweise der Vortritt für Frames vom Ring und von Aussen gewährt. Wurde zum Beispiel ein Frame vom Ring versendet, wird als nächstes ein Frame von Aussen versendet, sofern eines gesendet werden muss.

3.3.1.6. Simulation 1.6: Zeitschlitzverfahren

Konfigurationsname: Scen1_Simulation_1_6_Zeitschlitz

In jedem Knoten wird das Zeitschlitzverfahren (siehe Kapitel 3.1.2.5 auf Seite 25) angewendet.

3.3.1.7. Simulation 1.7: Maximale Auslastung

Konfigurationsname: Scen1_Simulation_1_7_Max_Auslastung

Die Knoten verwenden den «First Come First Serve»-Mechanismus.

Anstelle von 14 werden 17 MUs eingesetzt. Zudem werden in dieser Simulation lediglich Messwerte und Express-Frames versendet, der TCP-ähnliche Traffic wird hier nicht simuliert. Laut unserem Betreuer (siehe Besprechung in Kapitel 12.2.10 auf Seite 132) sollte dann neben dem Traffic nichts mehr sonst Platz auf dem Ethernet haben.

Das Lastprofil für diese Simulation sieht wie folgt aus:

Priorität	Quelle	Ziel	Intervall (Alle x Sekunden)	Grösse in Bytes	Zweck
Express	MU1	Multicast	0.01s + ϵ_1	100	Alarm
	MU8				
High	MU1	Multicast	0.00025s + ϵ_2	160	Messwerte
	MU2				
	MU3				
	MU4				
	MU5				
	MU6				
	MU7				
	MU8				
	MU9				
	MU10				
	MU11				
	MU12				
	MU13				
	MU14				
	MU15				
	MU16				
	MU17				

Tabelle 3.15.: Lastprofil Simulation 1.7: Maximale Auslastung

- ϵ_1 = Zufallswert zwischen (0s und 0.01s)
- ϵ_2= Zufallswert zwischen (0s und 0.0000125s)

Diese Frames werden ab dem Zeitpunkt t=0.001s bis zu tstop=0.56s generiert.

3.3.1.8. Simulation 1.8: Maximale Auslastung mit TCP-ähnlichem Verkehr

Konfigurationsname: Scen1_Simulation_1_8_Max_Auslastung_TCP_similar

Es wird dieselbe Simulation wie Simulation 1.7 «Maximale Auslastung» (siehe Kapitel 3.3.1.7 auf der vorherigen Seite) ausgeführt, jedoch wird zusätzlich TCP-ähnlicher Traffic generiert.

Das Lastprofil aus Simulation 1.7 «Maximale Auslastung» wird um folgende Frames erweitert:

Priorität	Quelle	Ziel	Intervall (Alle x Sekunden)	Grösse in Bytes	Zweck
Low	MU5	PU1	0.001s	1500	TCP-ähnlicher Traffic
	PU1	MU5		64	

Tabelle 3.16.: Erweiterung des Lastprofils aus Simulation 1.7: Maximale Auslastung

4. Resultate und Interpretation

Die Resultate der Simulationen aus Kapitel 3.3 auf Seite 49 werden in diesem Kapitel aufgeführt und interpretiert. Folgende Eigenschaften werden während der Simulation aufgezeichnet, um sie anschliessend analysieren zu können:

- **Übertragungsdauer eines Frames von der Erstellung bis zur Löschung**
  Wird ein Frame erstellt, wird es in die jeweilige Queue beim Knoten eingereiht. Das heisst, dass die Zeit zwischen der Erstellung und dem ersten Versand auch zur Übertragungsdauer gehört. Ein Unicast-Frame wird gelöscht, wenn es bei seinem Empfänger oder wieder beim Sender ankommt. Multi-/Broadcast-Frames werden gelöscht, sobald sie wieder beim Sender ankommen.

  Die Resultate dieser Eigenschaft werden in einem Histogramm gespeichert.

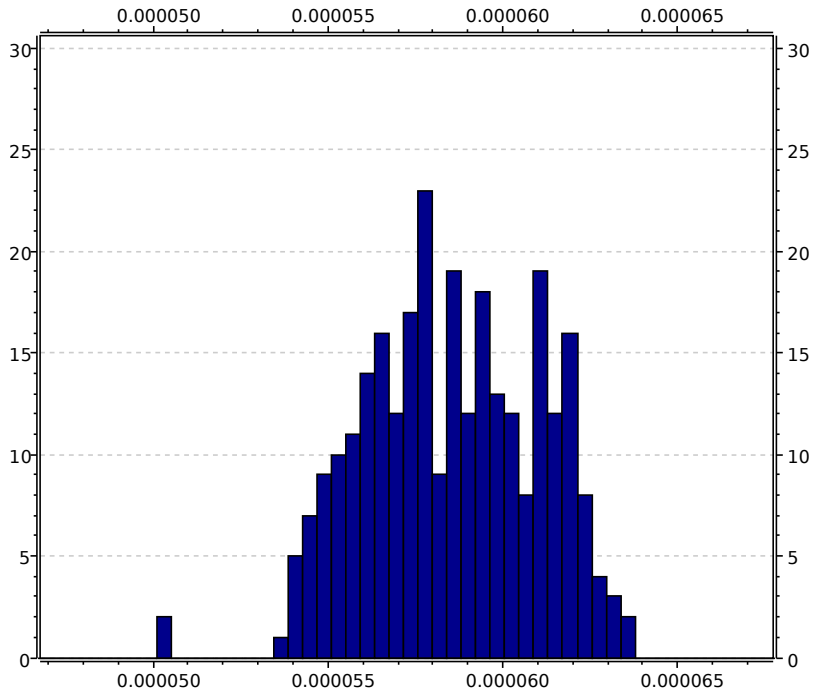


Abbildung 4.1.: Beispiel eines Histogramms für Frames einer bestimmten Priorität

Bei diesem Histogramm ist ersichtlich, wie viele Frames (Y-Achse) welche Übertragungsdauer in Sekunden (X-Achse) haben. Dieser Grafik kann man entnehmen, dass kein Frame äusserst lange verzögert wurde und alle Frames eine ähnliche Übertragungsdauer hatten.

- **Grösse der Queues bei den Schedulern**
  Sobald ein Frame bei einer Queue eingereiht oder von einer Queue entnommen wird, wird die Grösse der Queue jedes mal aufgezeichnet. So lässt sich feststellen, zu exakt welchem Zeitpunkt ein Frame in die Queue kommt und wann ein Frame die Queue verlässt.

  Dabei besitzt jeder DANH-Knoten 3 Ausgänge (2 Verbindungen zum Ring, 1 Verbindung zur CPU) und für jeden Ausgang einen Scheduler. Ein Scheduler verwendet 6 Queues für das Verarbeiten der Frames. Beim «First Come First Serve»-Mechanismus werden lediglich 3 Queues verwendet, da dort nicht unterschieden wird, ob das Frame vom Ring oder von der CPU kommt.

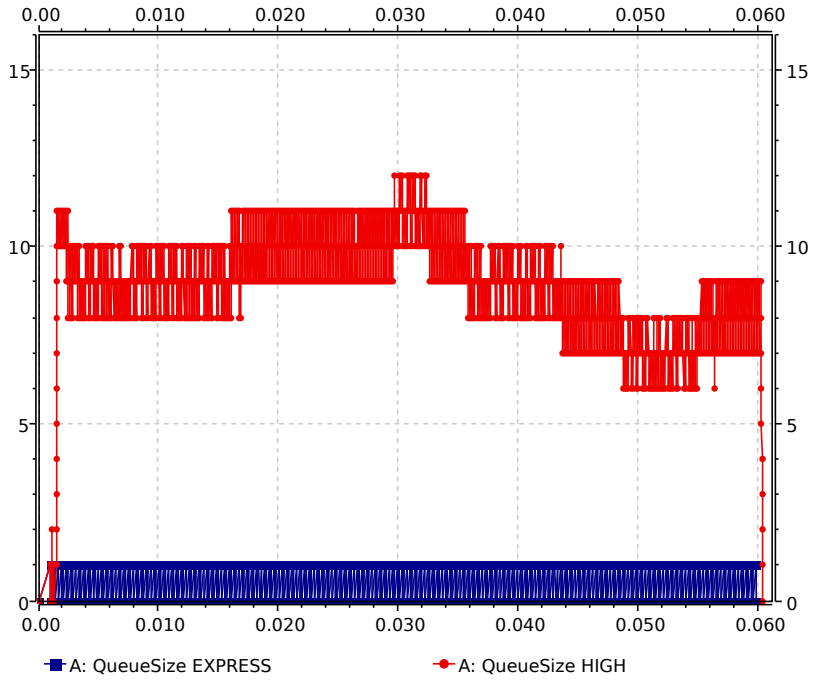


Abbildung 4.2.: Beispiel eines Queueverlaufs für Frames von 2 Queues

In dieser Grafik sieht man zu welchem Zeitpunkt in Sekunden (X-Achse) wie viele Frames (Y-Achse) in welcher Queue eingereiht sind. So lässt sich auch feststellen, wann ein Frame zur Queue hinzugefügt oder entfernt wurde. In diesem Beispiel ist ersichtlich, dass ständig Express-Frames versendet werden, während sich High-Frames anstauen und nicht immer umgehend versendet werden können.

Des Weiteren kann es vorkommen, dass sich Frames in einer Queue nicht anstauen und dann ein ähnlicher Graph wie folgender dabei herauskommt:

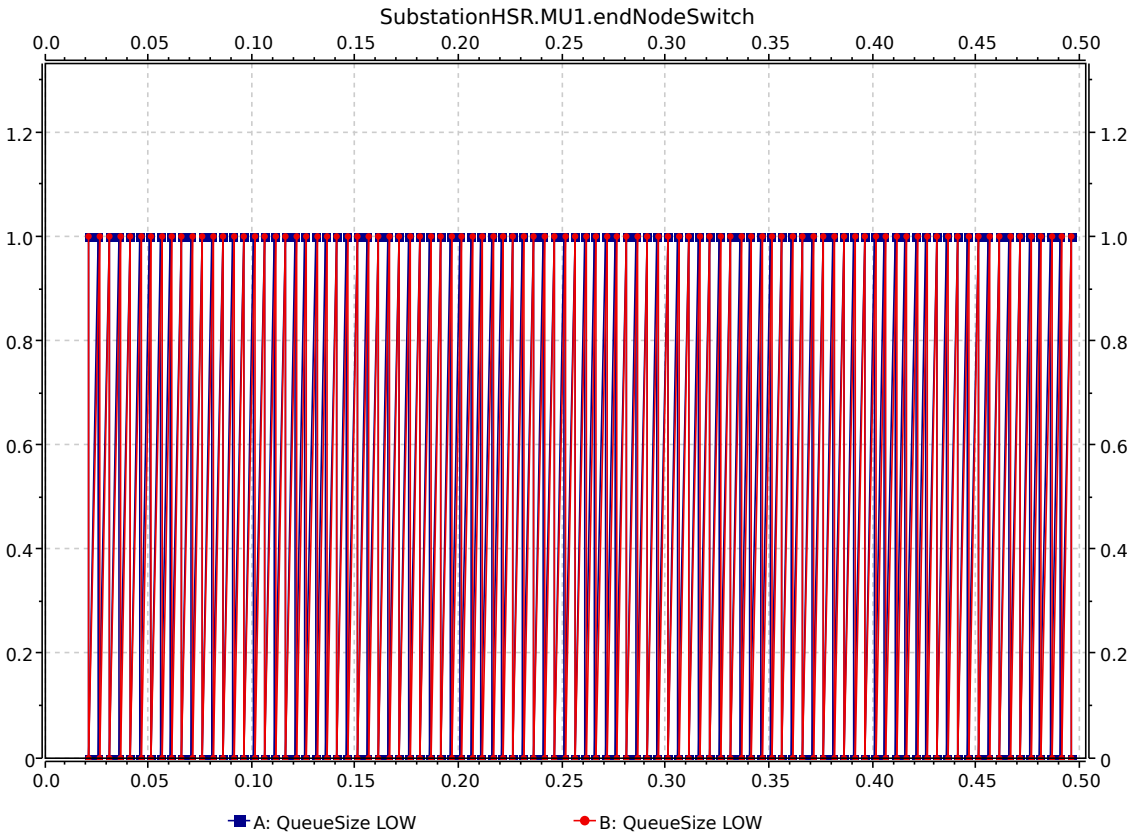


Abbildung 4.3.: Beispiel einer Queue, bei der sich die Frames nicht anstauen

Falls ein solches Verhalten in diesem Kapitel vorkommt, wird aus Platzgründen ein Vermerk anstelle einer Grafik aufzufinden sein.

- **Anzahl Unterbrechungen durch Express-Frames**
  Wird die Übertragung eines normalen Frames durch ein Express-Frame «unterbrochen», wird dies vermerkt. So kann man sehen, zu welchem Zeitpunkt wie viele Unterbrechungen stattgefunden haben. Dies ist bei der jeweiligen Simulation im Graphen im entsprechenden Unterkapitel mit dem Titel «Anzahl Unterbrechungen (Preemptions)» einsehbar.

Bei diesen Grafiken kann man, wie es in den vorherigen Grafiken gezeigt wurde, auch die Eigenschaften mehrerer Prioritäten / Queues kombiniert ausgeben. Dies macht jedoch nicht immer Sinn, da die Skalierung der Werte sehr unterschiedlich ausfallen kann.

Die Werte werden lediglich innerhalb des sogenannten Steady-States aufgezeichnet, da die Werte beim Start und Stop der Simulation nicht von Interesse sind. Somit wird nur das Verhalten innerhalb des stetigen Zustands analysiert und der Aufbau und Abschluss ausser Acht gelassen.

In diesem Kapitel werden nicht die Resultate aller Knoten im Netzwerk angegeben und ausgewertet, da viele Knoten dasselbe Verhalten haben. Anhand der Dateien, die sich auf dem USB-Stick befinden, der dieser Arbeit beiliegt, hat man die Möglichkeit, die entsprechende Simulation nochmals selbst durchzuführen und alle Resultate zu betrachten.

4.1. Szenario 1: Substation Automation

Für dieses Szenario werden die Werte ab dem Zeitpunkt t=0.02s bis zu tstop=0.5 aufgezeichnet. Dieser Bereich beinhaltet für dieses Szenario garantiert nur das Verhalten innerhalb des Steady-States.

Über die Grösse der Queues mit Express-Frames lässt sich betreffend jeder Simulation in diesem Szenario folgendes sagen: In keinem Knoten haben sich die Express-Frames bei einem Ausgang angestaut. Jedes generierte Express-Frame konnte schnellstmöglich versandt werden. Wie sich die Queues mit anderen Frames verhalten, wenn sie sich anstauen, wird in den einzelnen Simulationen dieses Szenarios beschrieben.

Je nach Mechanismus verhalten sich die Queues andersartig, während die Charakteristiken der Queues mit Express-Frames immer ähnlich verlaufen. Lediglich die Reihenfolge der Express-Frames untereinander und der exakte Zeitpunkt der «Fragmentierung» normaler Frames durch Express-Frames kann von Simulation zu Simulation variieren, jedoch ist dieser Unterschied für die Simulationen dieses Szenarios untereinander betrachtet gering und vernachlässigbar. Der Einfluss der Express-Frames auf die anderen Frames ist in der jeweiligen Simulation in den entsprechenden Queues ersichtlich.

Als Beispiel für den Einfluss von Express-Frames auf Andere ist die Abbildung 4.2 auf Seite 55 zu betrachten. Dort ist klar ersichtlich, dass sich die Frames in der High-Queue anstauen, während fortlaufend Express-Frames versendet werden.

Es wurde von jedem Typ ein einzelnes Frame via Multicast durch das Netzwerk gesendet, um die kleinstmögliche Zeit für den kompletten Durchlauf eines Frames bis zum erneuten Erreichen des Senders herauszufinden:

Priorität	Grösse in Bytes	Dauer $[\mu s]$
Express	100	$241.84\mu s$
High	160	$318.64\mu s$
Low	1500	$2033.84\mu s$
Low	64	$195.76\mu s$
Low	1500 & 64 im Durchschnitt	$1114.80\mu s$

Tabelle 4.1.: Übertragungszeiten einzelner Multicast-Frames in Szenario 1

4.1.1. Simulation 1.1: First Come First Serve

4.1.1.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$272.92\mu s$	$368.83\mu s$	$445.25\mu s$

Tabelle 4.2.: Szenario 1 / Simulation 1.1 / Datenangabe

Express-Frames

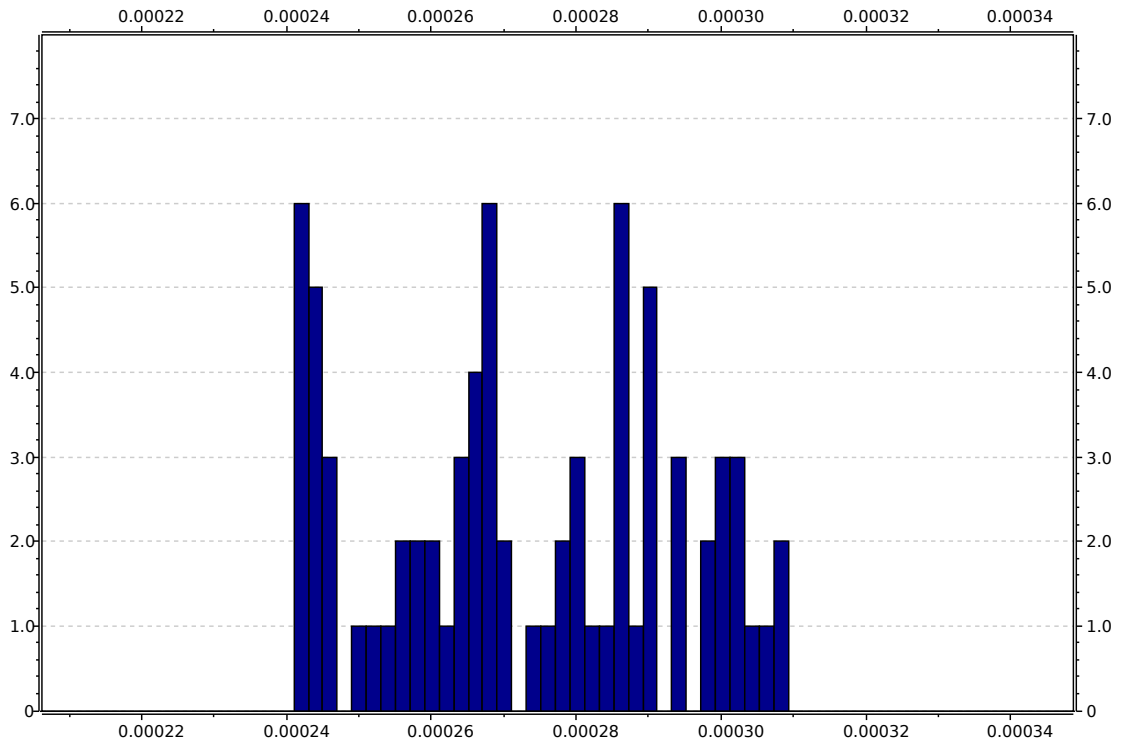


Abbildung 4.4.: Szenario 1 / Simulation 1.1 / Übertragungszeiten von Express-Frames

Die Übertragungszeiten der Express-Frames liegen nahe beieinander und sind alle im selben Bereich verteilt. Wenn alle Express-Frames sofort nach ihrer Erzeugung hätten versendet werden können, wäre nur ein Balken im Histogramm ersichtlich. Hier hingegen kann man feststellen, dass mit dem Versand zeitweise abgewartet werden musste, bis ein Fragment eines normalen Frames erstellt werden konnte oder Platz frei war, um das Express-Frame zu senden. Dank der Frame-Preemption musste kein Express-Frame zu lange warten, bis es versendet werden konnte.

Frames mit der Priorität High

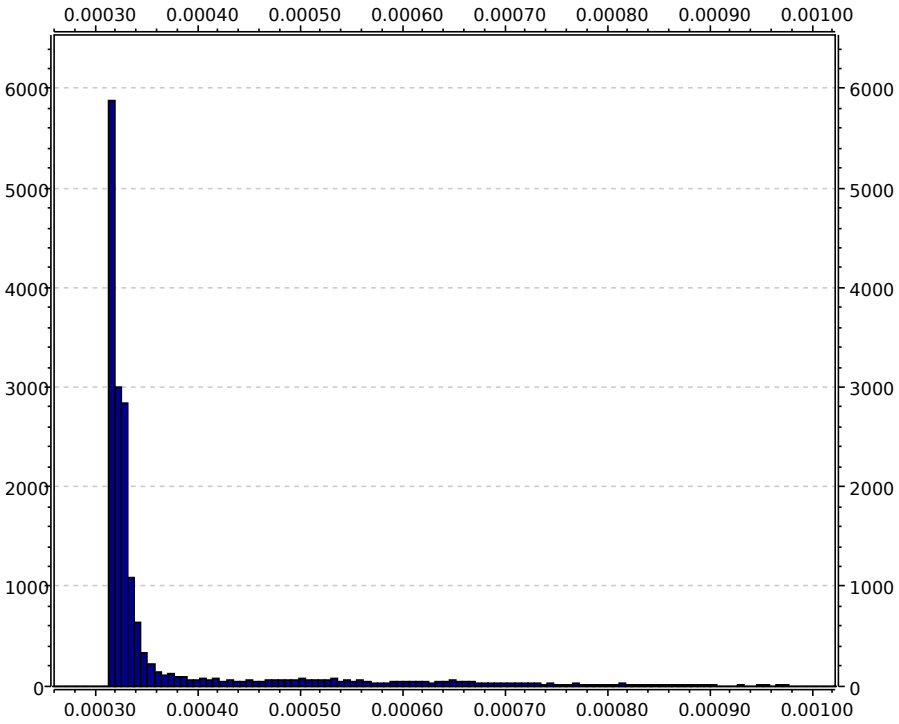


Abbildung 4.5.: Szenario 1 / Simulation 1.1 / Übertragungszeiten von High-Frames

Man kann gut erkennen, dass der grösste Teil aller High-Frames schnellstmöglich übertragen werden konnte. Jedoch existieren vereinzelte Frames, die aufgrund Preemption oder Frames, die gerade versendet werden, eine markante Verzögerung aufweisen.

Frames mit der Priorität Low

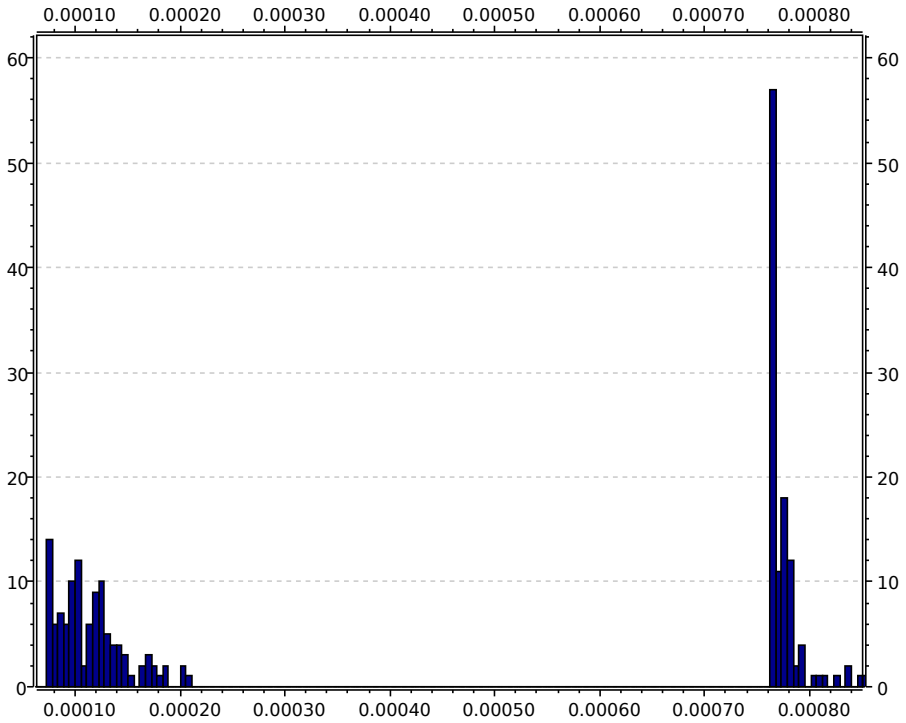


Abbildung 4.6.: Szenario 1 / Simulation 1.1 / Übertragungszeiten von Low-Frames

In dieser Grafik kann man klar feststellen, welches die Low-Frames à 64 Bytes und welche die mit 1500 Bytes sind. Dies resultiert daraus, dass hier zwischen zwei Knoten TCP-ähnlicher Unicast-Traffic simuliert wird. Kein Frame hat eine auffallend längere Übertragungsdauer, jedoch sieht man an der Zeitachse, dass die durchschnittliche Übertragungszeit deutlich länger ist, als diejenige der höher priorisierten Frames.

4.1.1.2. Queues

Frames mit der Priorität High

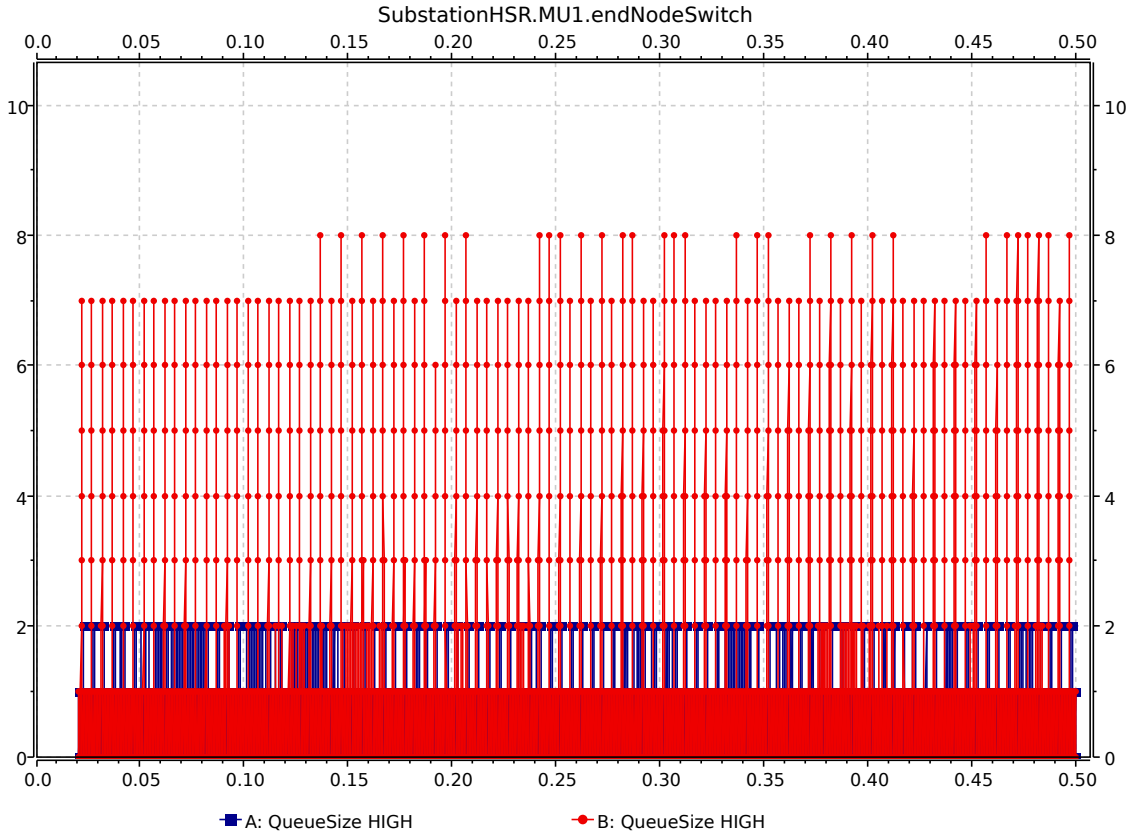


Abbildung 4.7.: Szenario 1 / Simulation 1.1 / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen

Die High-Frames stauen sich zwar zeitweise an, jedoch kann dieser Stau immer wieder abgebaut werden, bevor sich weitere Frames aufstauen. Maximal werden beim Ausgang A 2 Frames und beim Ausgang B 8 Frames auf einmal in einer Queue gespeichert. Diese Divergenz lässt sich dadurch erklären, dass «MU2» beim Ausgang A weniger oft High-Frames versendet, da diese selbst mit der Generierung von Frames beschäftigt ist. An Ausgang B hingegen ist «PU2» angeschlossen, welche die Frames einfach nur weiterleiten muss und selbst keine generiert.

Frames mit der Priorität Low

Aufgrund der (verglichen mit den Frames anderer Prioritäten) niedriger Erzeugungsrate stauten sich bei den Frames mit der Priorität Low keine bei einem Ausgang eines Knotens an.

4.1.1.3. Anzahl Unterbrechungen (Preemptions)

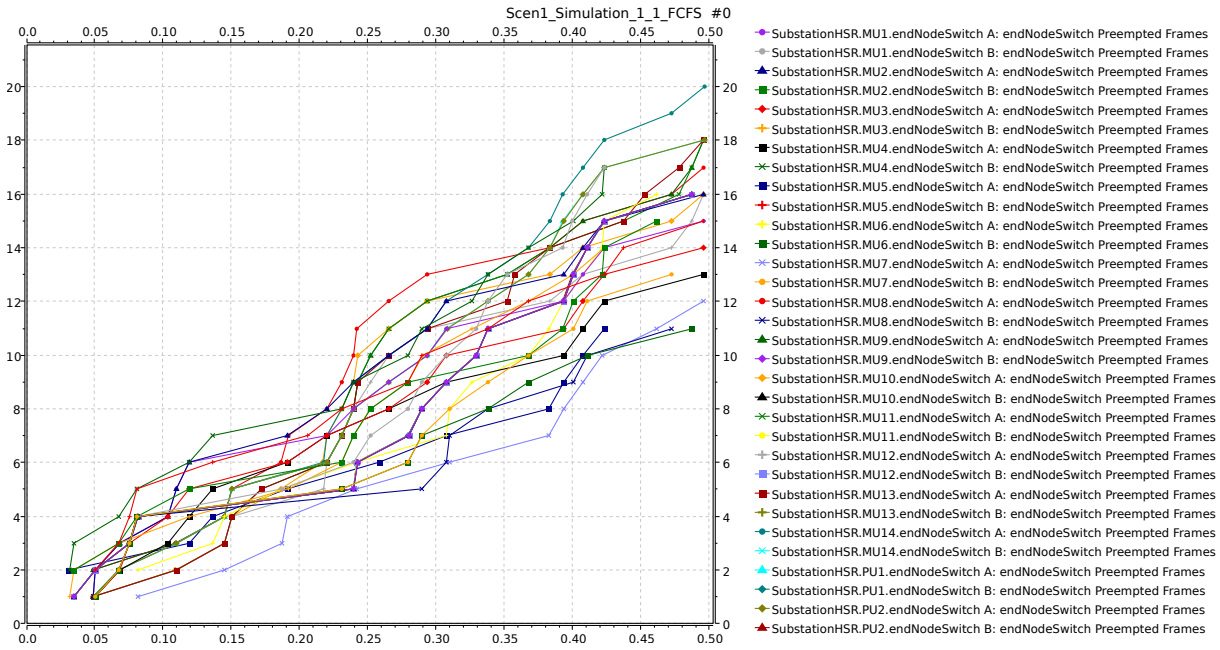


Abbildung 4.8.: Szenario 1 / Simulation 1.1: Anzahl Unterbrechungen durch Express-Frames

4.1.2. Simulation 1.2: Vortritt für Frames vom Ring

4.1.2.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$274.13\mu s$	$369.06\mu s$	$446.28\mu s$

Tabelle 4.3.: Szenario 1 / Simulation 1.2 / Datenangabe

Express-Frames

Die Übertragungszeiten der Express-Frames in dieser Simulation sind dieselben wie in Abbildung 4.4 auf Seite 58. Die Erläuterung dazu ist unter besagter Abbildung zu finden.

Frames mit der Priorität High

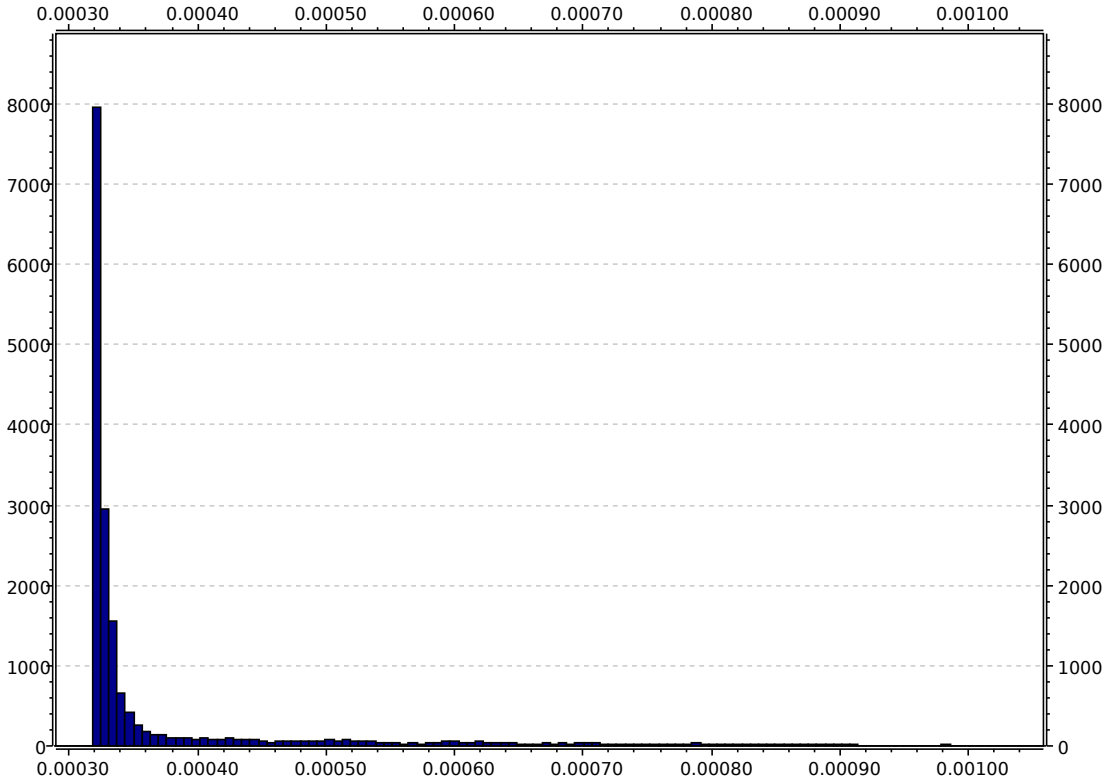


Abbildung 4.9.: Szenario 1 / Simulation 1.2 / Übertragungszeiten von High-Frames

Man kann gut erkennen, dass der grösste Teil aller High-Frames schnellstmöglich übertragen werden konnte. Jedoch existieren vereinzelte Frames, die aufgrund Preemption oder Frames, die gerade versendet werden, eine markante Verzögerung aufweisen.

Frames mit der Priorität Low

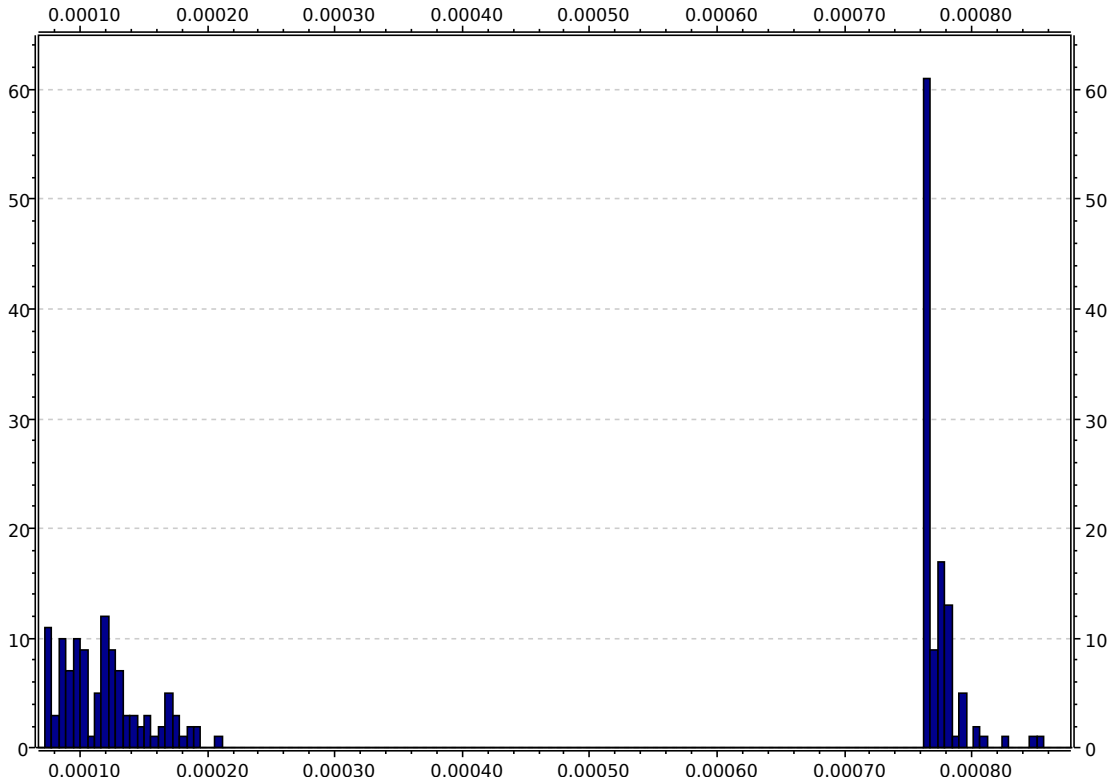


Abbildung 4.10.: Szenario 1 / Simulation 1.2 / Übertragungszeiten von Low-Frames

In dieser Grafik kann man klar feststellen, welches die Low-Frames à 64 Bytes und welche die mit 1500 Bytes sind. Dies resultiert daraus, dass hier zwischen zwei Knoten TCP-ähnlicher Unicast-Traffic simuliert wird. Kein Frame hat eine auffallend längere Übertragungsdauer, jedoch sieht man an der Zeitachse, dass die durchschnittliche Übertragungszeit deutlich länger ist, als diejenige der höher priorisierten Frames.

4.1.2.2. Queues

Frames mit der Priorität High

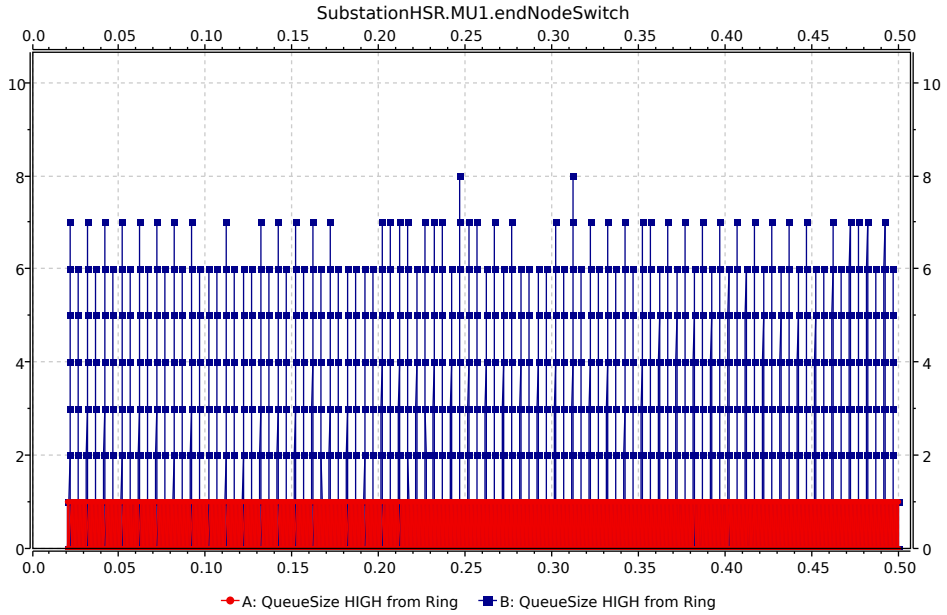


Abbildung 4.11.: Szenario 1 / Simulation 1.2 / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen mit Frames, die vom Ring kommen

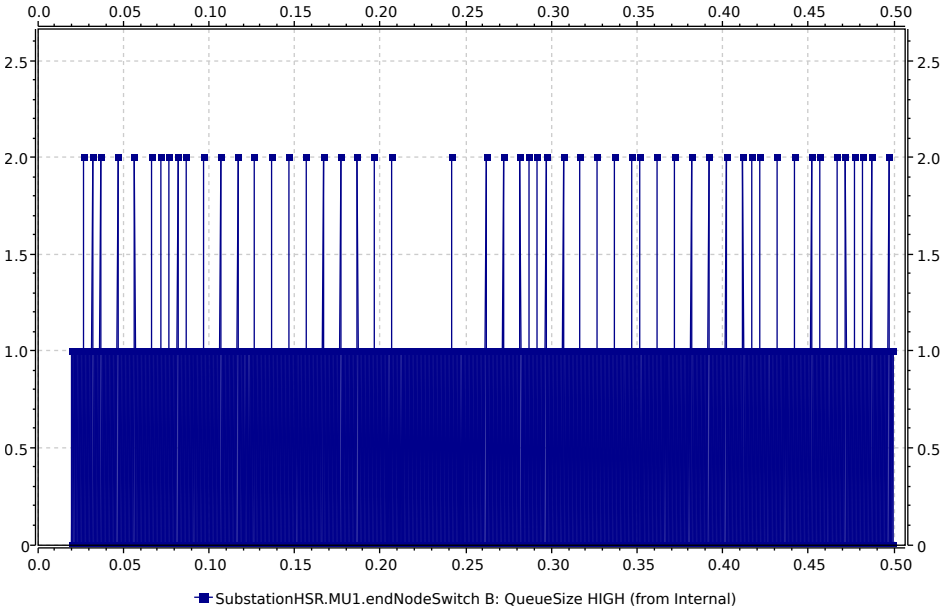


Abbildung 4.12.: Szenario 1 / Simulation 1.2 / Knoten «MU1»: Grösse der High-Queues bei Ausgang B mit Frames, die intern generiert werden (Frame A hat das selbe Verhalten)

Gesamthaft ist das Verhalten der High-Queues dasselbe wie bei der vorherigen Simulation, mit dem Unterschied, dass hier die Queues nach ihrer Herkunft aufgeteilt sind. Da von den anderen Knoten viel mehr Traffic eintrifft, als intern (von «MU1») generiert wird, staut es sich bei der Ring-Queue mehr an, obwohl diesen Frames der Vortritt gewährt ist. So lässt sich jedoch der Stau wieder aufheben und es gibt genügend Platz, um zwischendurch intern generierte Frames versenden zu können.

Frames mit der Priorität Low

Aufgrund der (verglichen mit den Frames anderer Prioritäten) niedriger Erzeugungsrate stauten sich bei den Frames mit der Priorität Low keine bei einem Ausgang eines Knotens an.

4.1.2.3. Anzahl Unterbrechungen (Preemptions)

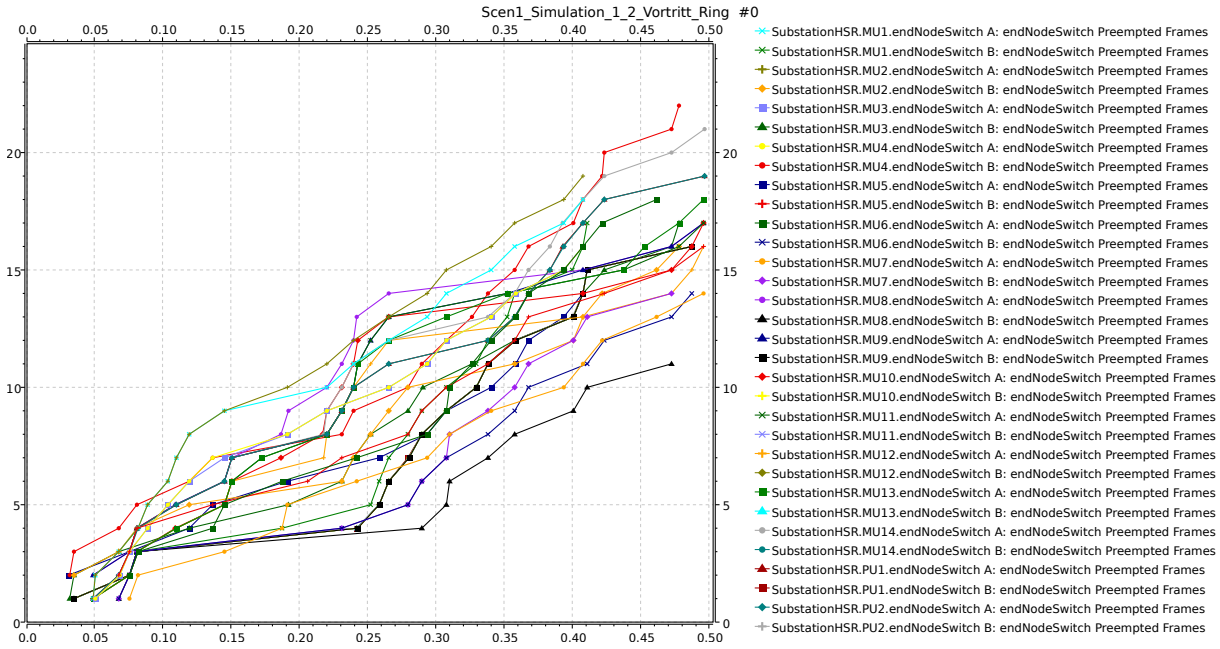


Abbildung 4.13.: Szenario 1 / Simulation 1.2: Anzahl Unterbrechungen durch Express-Frames

4.1.3. Simulation 1.3: Vortritt für Frames von Aussen

4.1.3.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$273.02\mu s$	$368.48\mu s$	$445.25\mu s$

Tabelle 4.4.: Szenario 1 / Simulation 1.3 / Datenangabe

Express-Frames

Die Übertragungszeiten der Express-Frames in dieser Simulation sind dieselben wie in Abbildung 4.4 auf Seite 58. Die Erläuterung dazu ist unter besagter Abbildung zu finden.

Frames mit der Priorität High

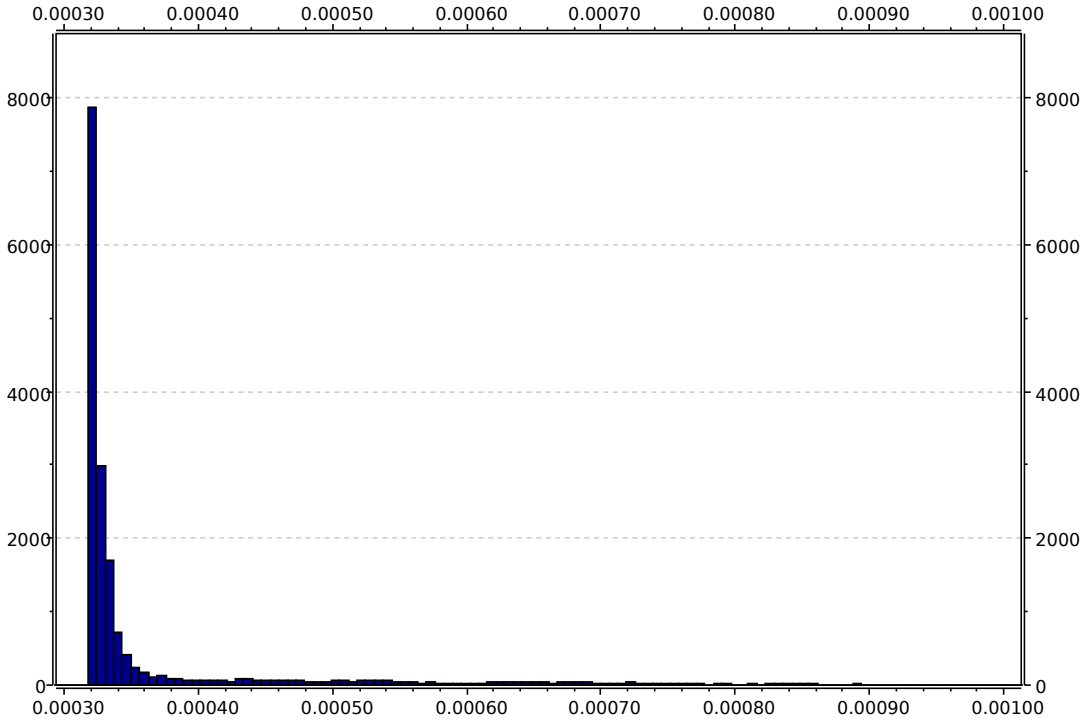


Abbildung 4.14.: Szenario 1 / Simulation 1.3 / Übertragungszeiten von High-Frames

Man kann gut erkennen, dass der grösste Teil aller High-Frames schnellstmöglich übertragen werden konnte. Jedoch existieren vereinzelte Frames, die aufgrund Preemption oder Frames, die gerade versendet werden, eine markante Verzögerung aufweisen.

Frames mit der Priorität Low

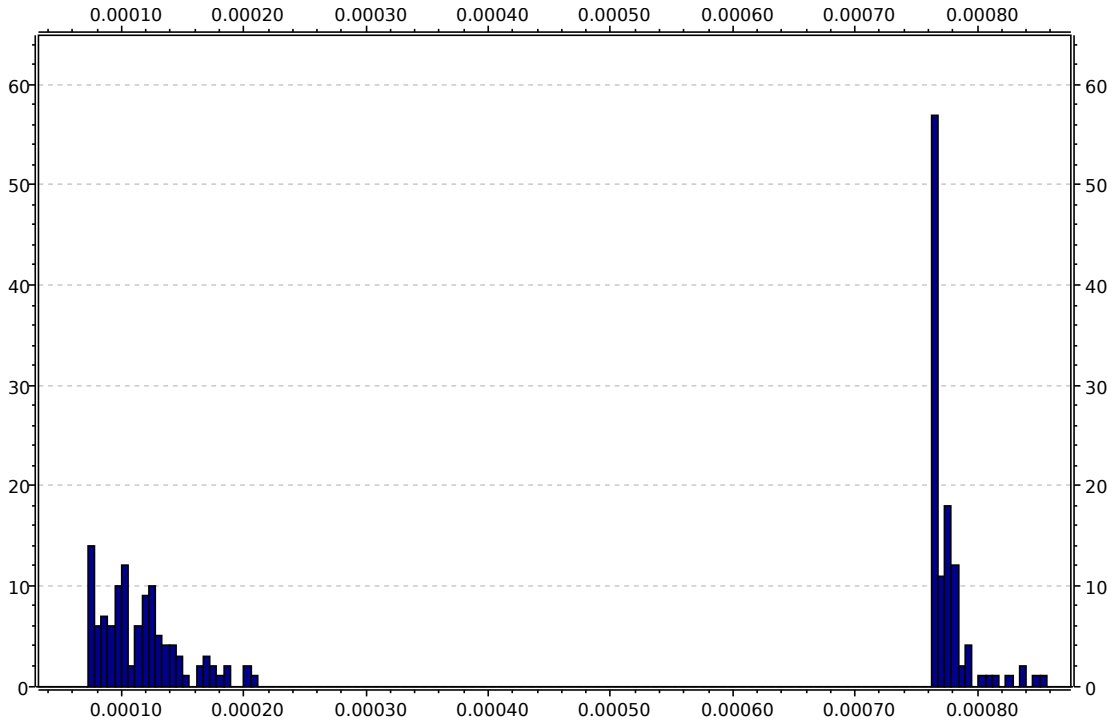


Abbildung 4.15.: Szenario 1 / Simulation 1.3 / Übertragungszeiten von Low-Frames

In dieser Grafik kann man klar feststellen, welches die Low-Frames à 64 Bytes und welche die mit 1500 Bytes sind. Dies resultiert daraus, dass hier zwischen zwei Knoten TCP-ähnlicher Unicast-Traffic simuliert wird. Kein Frame hat eine auffallend längere Übertragungsdauer, jedoch sieht man an der Zeitachse, dass die durchschnittliche Übertragungszeit deutlich länger ist, als diejenige der höher priorisierten Frames.

4.1.3.2. Queues

Frames mit der Priorität High

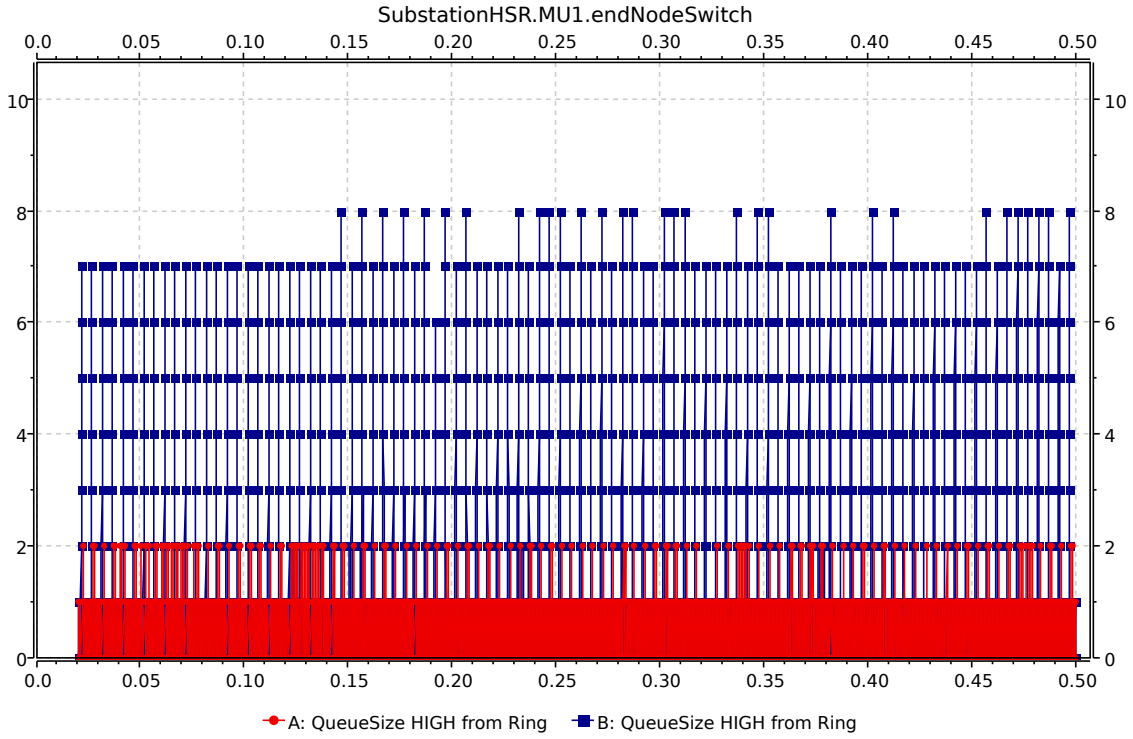


Abbildung 4.16.: Szenario 1 / Simulation 1.3 / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen mit Frames, die vom Ring kommen

Bei dieser Simulation werden intern generierte Frames den Frames, die vom Ring her kommen, vorgezogen. Aus diesem Grund können die intern generierten Frames sofort versendet werden ohne einen Rückstau entstehen zu lassen. Bei den Frames gibt es, wie man sehen kann, einen Rückstau, der jedoch wieder abgearbeitet werden kann.

Frames mit der Priorität Low

Aufgrund der (verglichen mit den Frames anderer Prioritäten) niedriger Erzeugungsrate stauten sich bei den Frames mit der Priorität Low keine bei einem Ausgang eines Knotens an.

4.1.3.3. Anzahl Unterbrechungen (Preemptions)

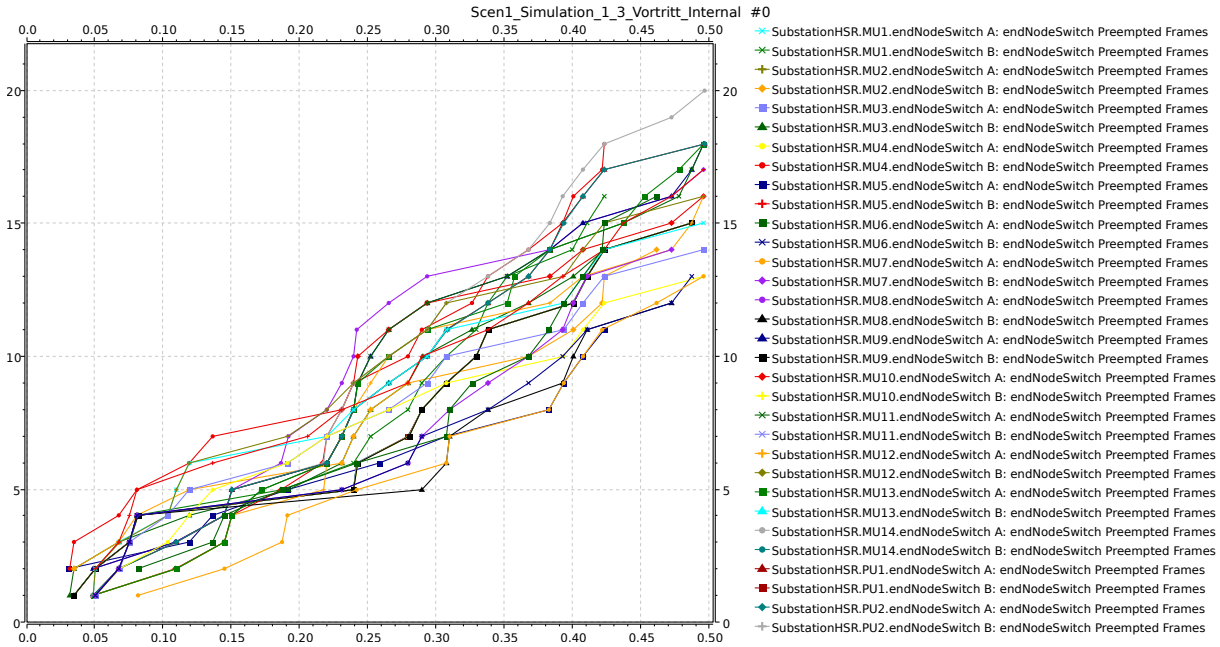


Abbildung 4.17.: Szenario 1 / Simulation 1.3: Anzahl Unterbrechungen durch Express-Frames

4.1.4. Simulation 1.4: Vortritt für Frames von Aussen mit Zuflusslimitierung

Die Zuflusslimitierung wurde auf ein Maximum von 88 Kbit/s gesetzt. Im weiteren Verlauf wird das Ergebnis mit Simulationen, die eine Zuflusslimitierung von 76 und 100 Kbit/s gesetzt haben, verglichen.

4.1.4.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$269.93\mu s$	$347.18\mu s$	$12417.16\mu s$

Tabelle 4.5.: Szenario 1 / Simulation 1.4 mit 88 Kbit/s Limitierung / Datenangabe

Express-Frames

Die Übertragungszeiten der Express-Frames in dieser Simulation sind dieselben wie in Abbildung 4.4 auf Seite 58. Die Erläuterung dazu ist unter besagter Abbildung zu finden.

Frames mit der Priorität High

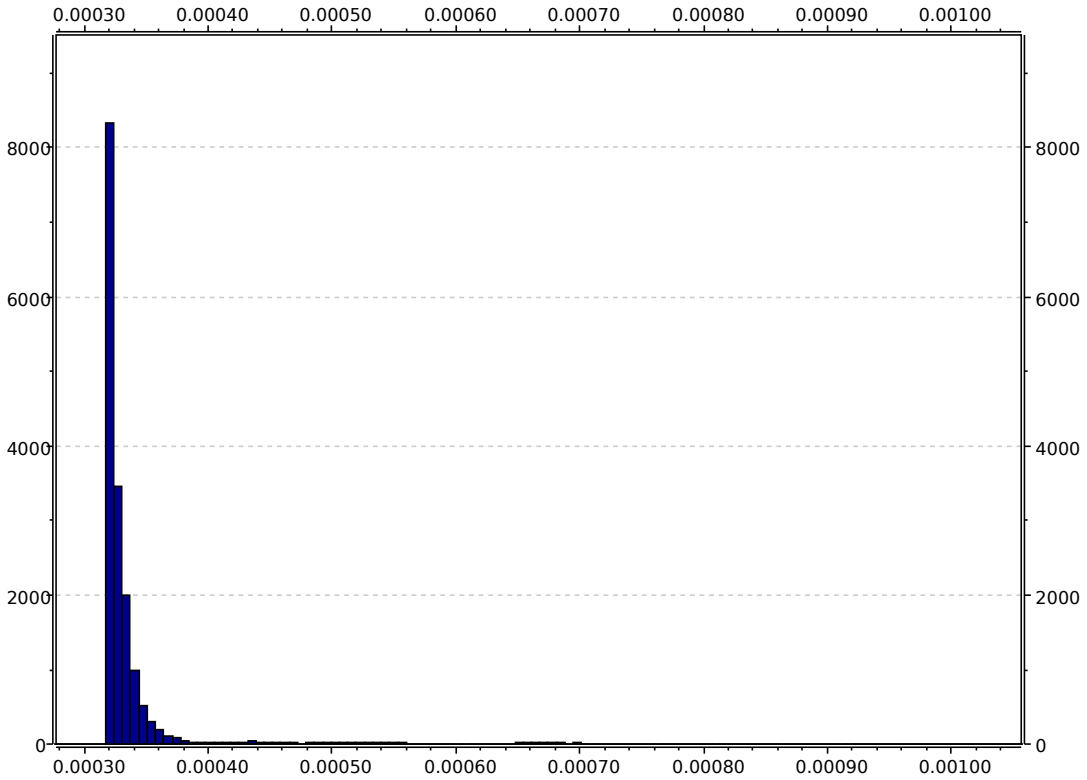


Abbildung 4.18.: Szenario 1 / Simulation 1.4 mit 88 Kbit/s Limitierung / Übertragungszeiten von High-Frames

Man kann gut erkennen, dass der grösste Teil aller High-Frames schnellstmöglich übertragen werden konnte. Jedoch existieren vereinzelte Frames, die aufgrund Preemption oder Frames, die gerade versendet werden, eine markante Verzögerung aufweisen.

Frames mit der Priorität Low

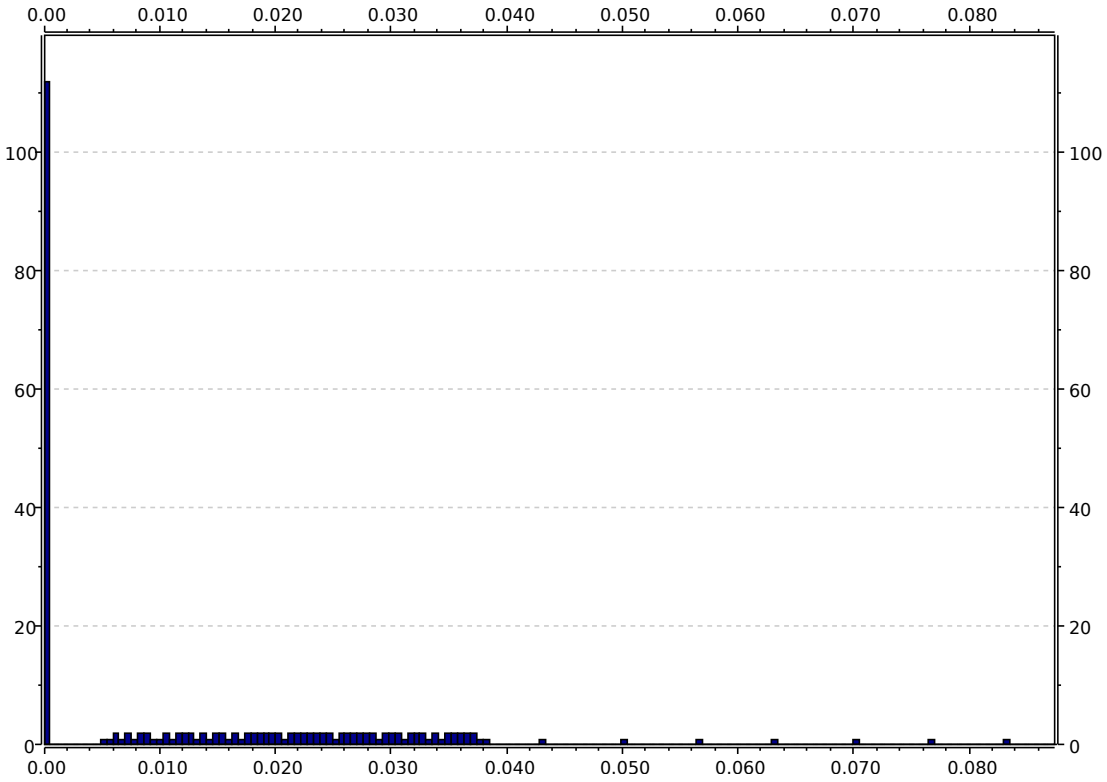


Abbildung 4.19.: Szenario 1 / Simulation 1.4 mit 88 Kbit/s Limitierung / Übertragungszeiten von Low-Frames

Da die Zuflusslimitierung sich auf Low-Frames beschränkt, ist hier ersichtlich, dass fast die Hälfte der niedrig priorisierten Frames um einige Zeit verzögert werden. Werden die Übertragungszeiten mit einem Histogramm aus einer der vorherigen Simulationen (siehe Abbildung 4.15 auf Seite 68) verglichen, sieht man, dass hier die maximale Übertragungszeit ca. $82000\mu s$ beträgt wobei die im referenzierten Histogramm maximal $860\mu s$ aufweist. Dies ist dadurch zu erklären, dass durch die Zuflusslimitierung Low-Frames zurückgehalten werden.

4.1.4.2. Queues

Frames mit der Priorität High

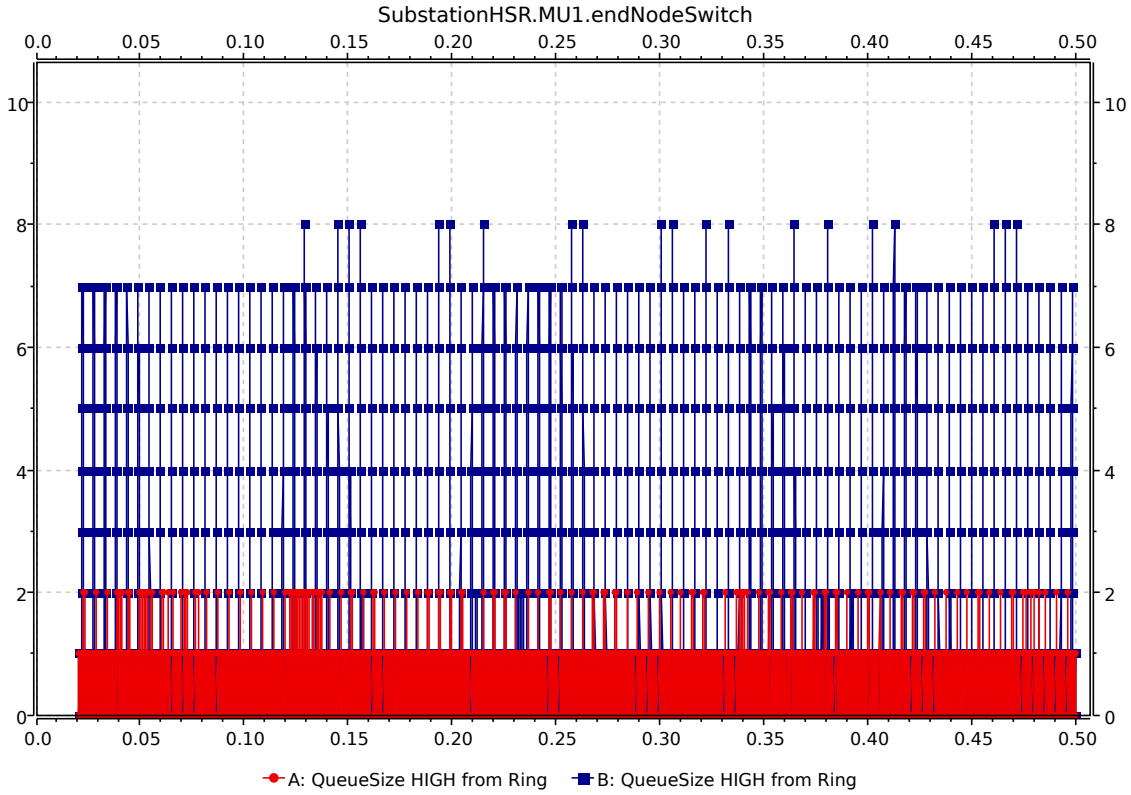


Abbildung 4.20.: Szenario 1 / Simulation 1.4 mit 88 Kbit/s Limitierung / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen mit Frames, die vom Ring kommen

Das Verhalten ist grundsätzlich dasselbe wie bei der Abbildung 4.16 auf Seite 69, jedoch ist erkennbar, dass der Rückstau hier marginal kleiner ausfällt als bei der verwiesenen Abbildung, da weniger Low-Frames versendet werden und es somit mehr Platz für High-Frames hat.

Frames mit der Priorität Low

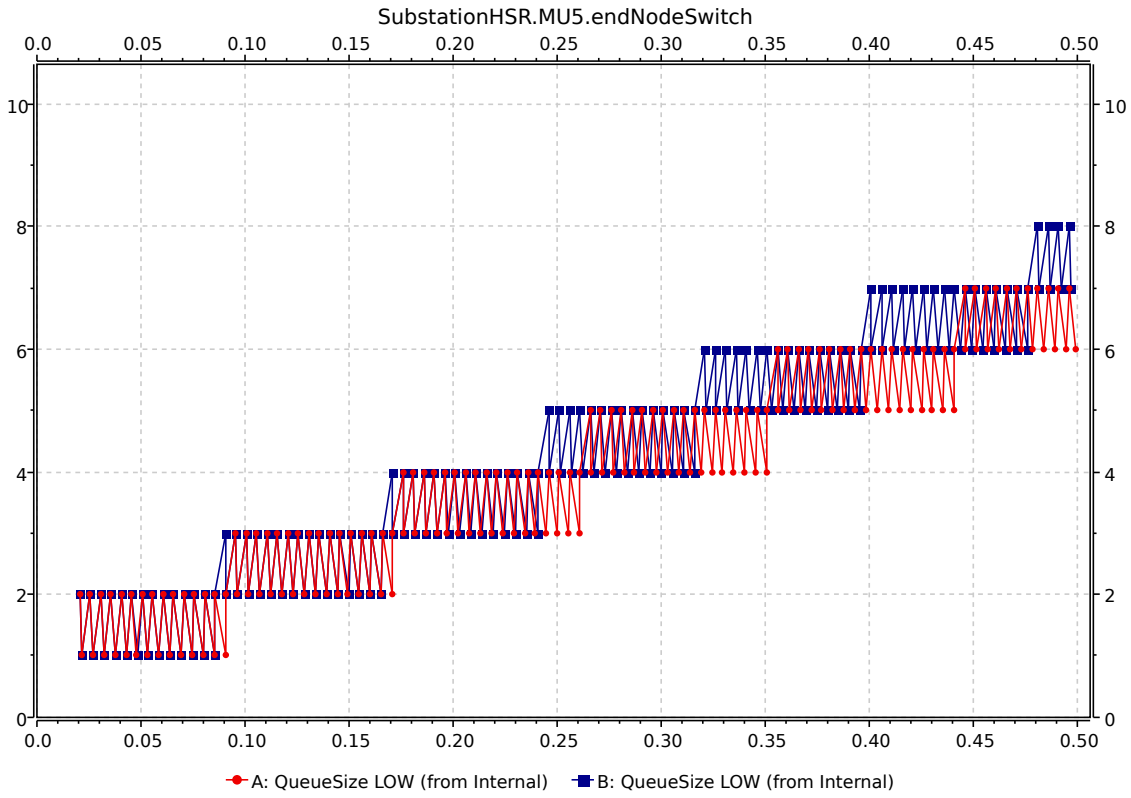


Abbildung 4.21.: Szenario 1 / Simulation 1.4 mit 88 Kbit/s Limitierung / Knoten «MU5»: Grösse der Low-Queues bei beiden Ausgängen mit Frames, die intern generiert werden

Diese Abbildung zeigt markant auf, wie die Zuflusslimitierung fungiert. Solange Low-Frames generiert werden, staut sich die Queue an und kann je nach konfiguriertem Zuflusslimit mit entsprechender Geschwindigkeit abgearbeitet werden. Wird ein kleines Limit eingestellt, staut sich die Queue schneller an als mit einem grösseren Wert. Dies kann in den Kapiteln 4.1.4.4 auf Seite 77 und 4.1.4.5 auf Seite 79 betrachtet werden.

4.1.4.3. Anzahl Unterbrechungen (Preemptions)

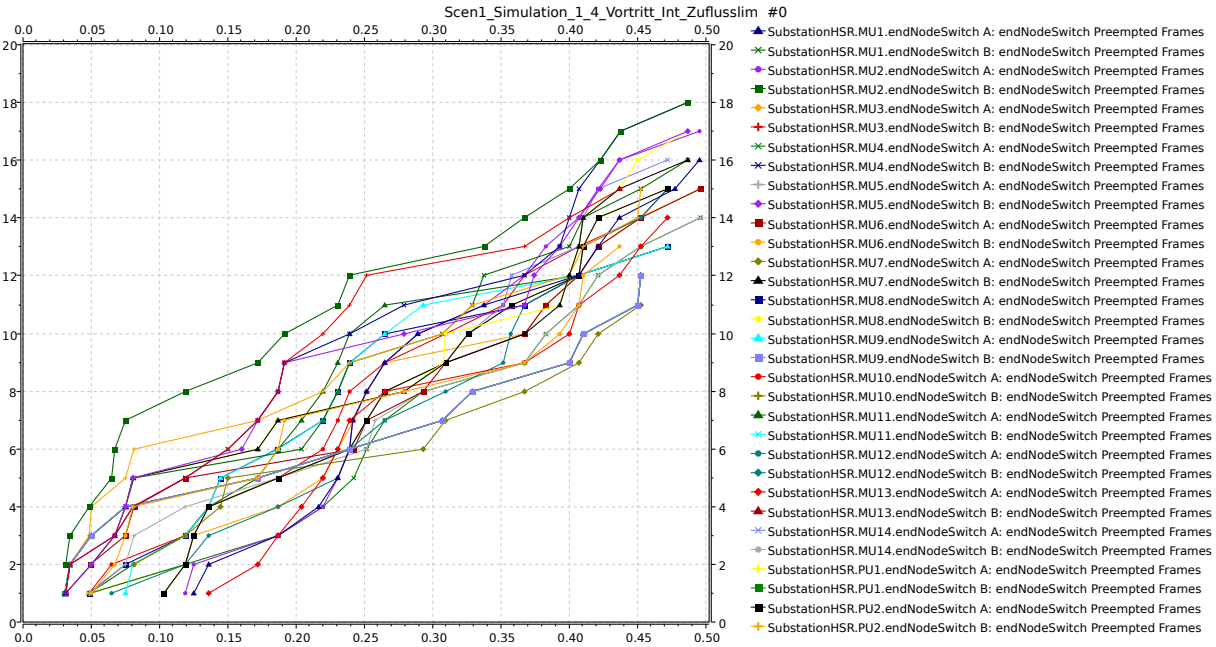


Abbildung 4.22.: Szenario 1 / Simulation 1.4 mit 88 Kbit/s Limitierung: Anzahl Unterbrechungen durch Express-Frames

4.1.4.4. Vergleich mit einer kleineren Zuflusslimitierung von 76 Kbit/s

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$269.84\mu s$	$345.44\mu s$	$25962.19\mu s$

Tabelle 4.6.: Szenario 1 / Simulation 1.4 mit 76 Kbit/s Limitierung / Datenangabe

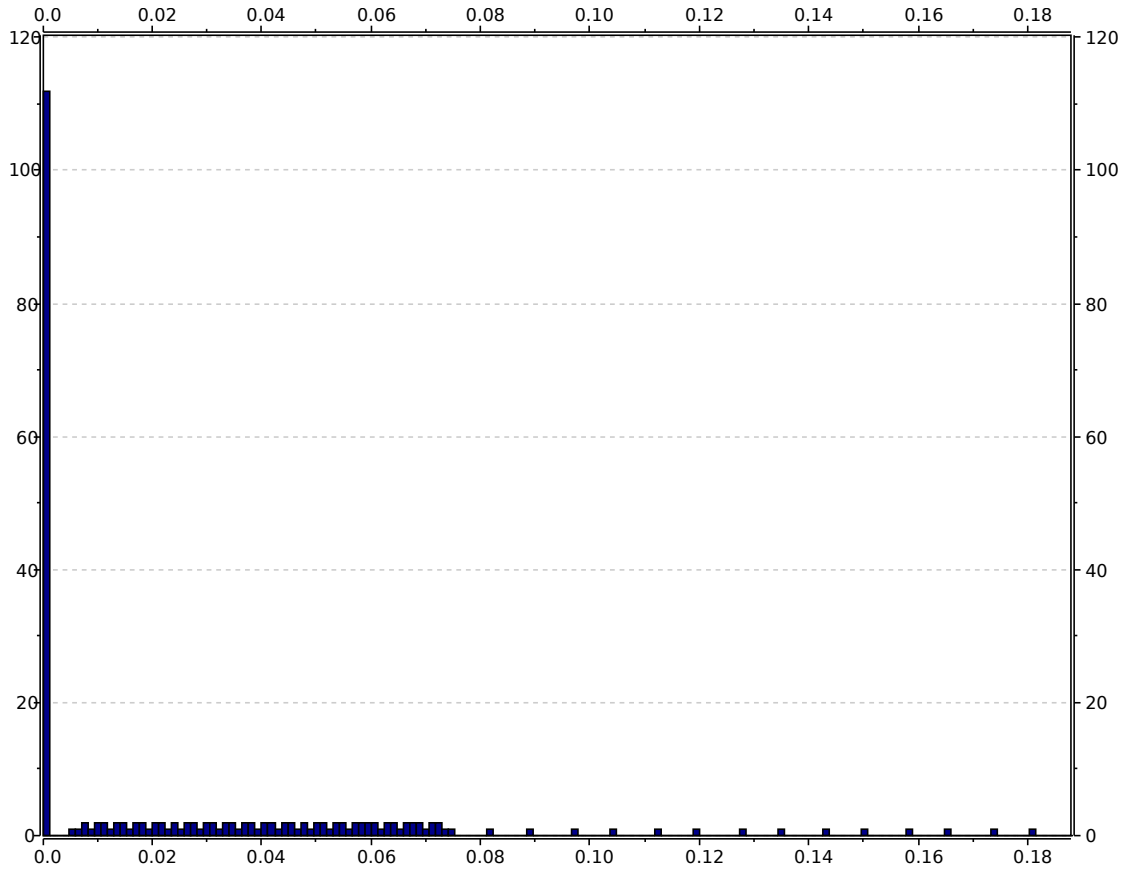


Abbildung 4.23.: Szenario 1 / Simulation 1.4 mit 76 Kbit/s Limitierung / Übertragungszeiten von Low-Frames

Anhand der Zeitachsen dieser und Abbildung 4.19 auf Seite 73 kann man sehen, dass mit einer Limitierung von 76 Kbit/s die Frames weitaus längere Übertragungszeiten haben. Das Maximum liegt hier bei ca. $180000\mu s$ und bei der referenzierten Grafik bei ca. $82000\mu s$. Dafür haben hier die höher priorisierten Frames im Durchschnitt eine kürzere Übertragungsdauer.

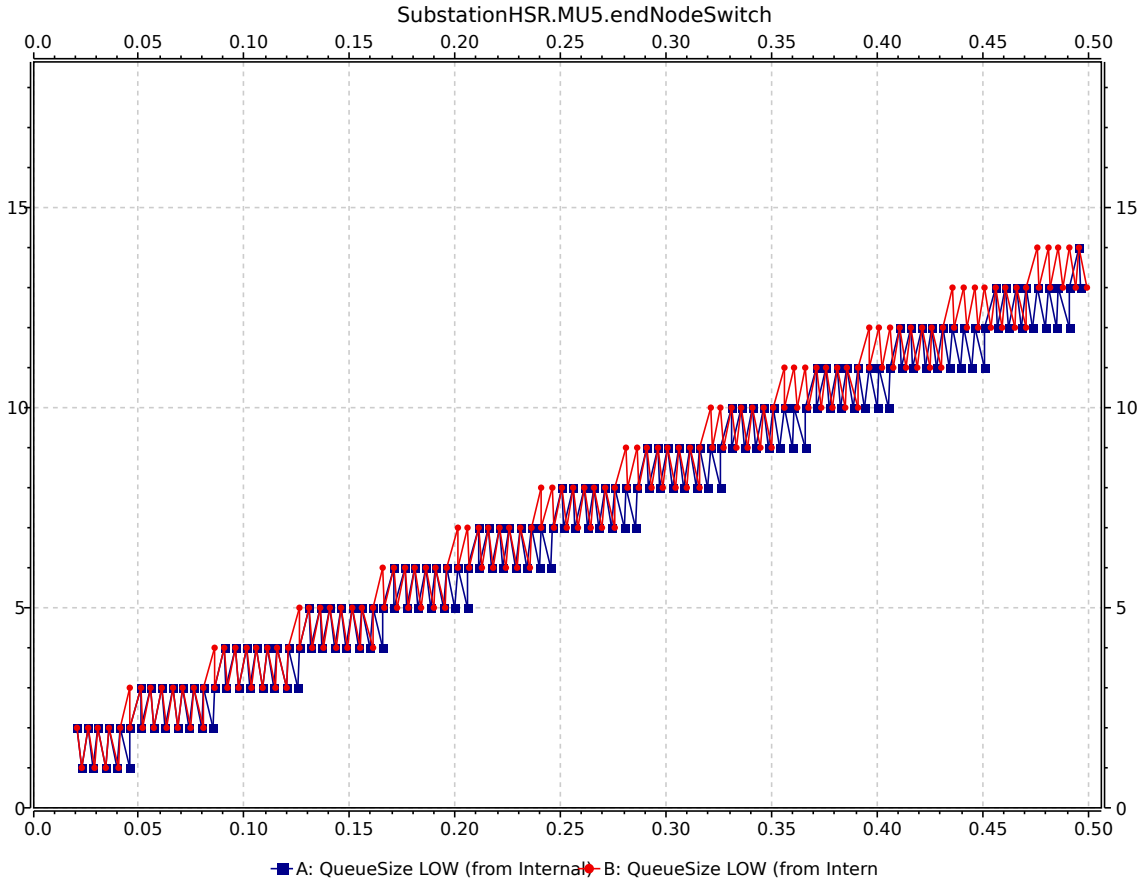


Abbildung 4.24.: Szenario 1 / Simulation 1.4 mit 76 Kbit/s Limitierung / Knoten «MU5»: Grösse der Low-Queues bei beiden Ausgängen mit Frames, die intern generiert werden

Hier werden die Frames im Vergleich zur Abbildung 4.21 auf Seite 75 schneller angestaut (grössere Steigung).

4.1.4.5. Vergleich mit einer grösseren Zuflusslimitierung von 100 Kbit/s

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$270.14\mu s$	$380.69\mu s$	$2572.05\mu s$

Tabelle 4.7.: Szenario 1 / Simulation 1.4 mit 100 Kbit/s Limitierung / Datenangabe

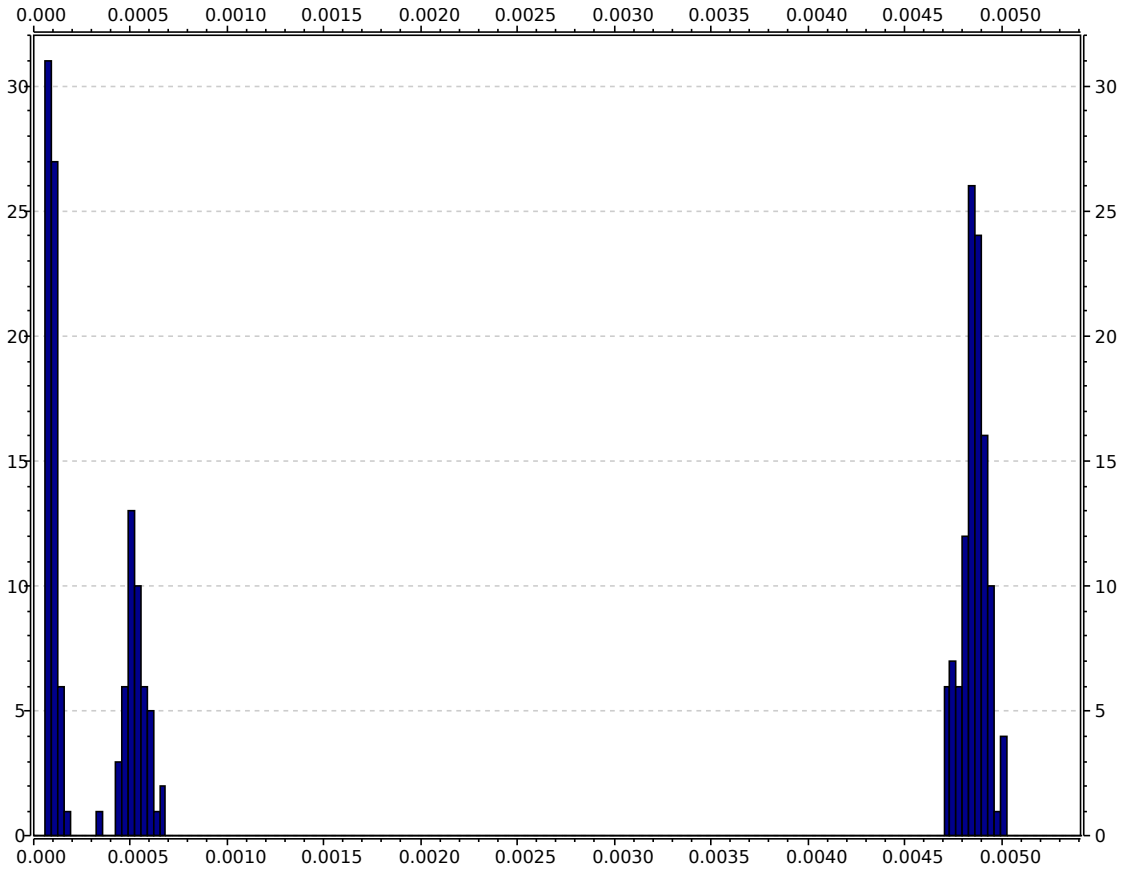


Abbildung 4.25.: Szenario 1 / Simulation 1.4 mit 100 Kbit/s Limitierung / Übertragungszeiten von Low-Frames

Beim Vergleich dieser und der Abbildung 4.15 auf Seite 68 kann man sehen, dass mit einer Limitierung von 100 Kbit/s die Frames zurückgehalten werden. Im Vergleich zu den Limitierungen mit 76 und 88 Kbit/s werden hier die Übertragungszeiten weniger markant verlängert.

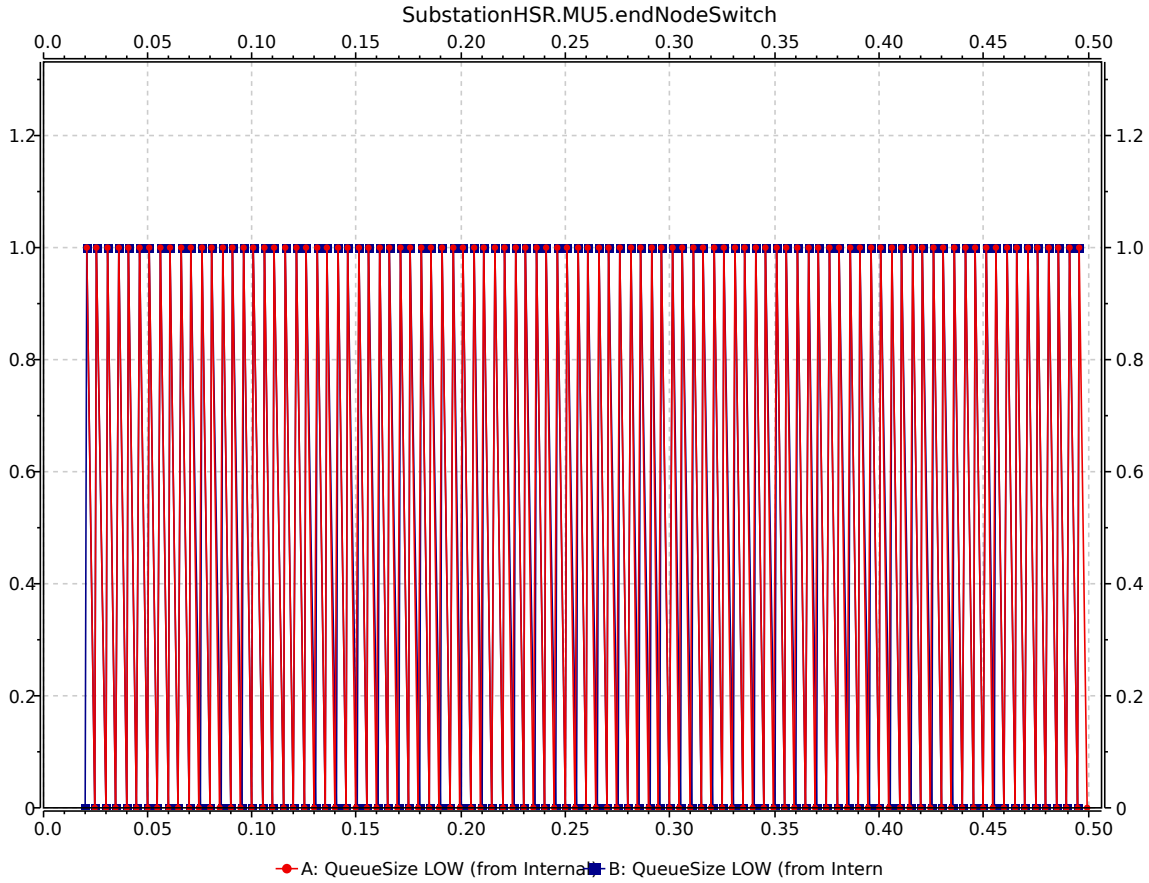


Abbildung 4.26.: Szenario 1 / Simulation 1.4 mit 100 Kbit/s Limitierung / Knoten «MU5»: Grösse der Low-Queues bei beiden Ausgängen mit Frames, die intern generiert werden

Betrachtet man lediglich diese Grafik, kann man die Zuflusslimitierung nicht feststellen. In Kombination mit dem vorher aufgeführten Histogramm, lässt sich sagen, dass die Frames zurückgehalten werden, jedoch verschickt werden, bevor ein Neues erzeugt wird.

4.1.5. Simulation 1.5: Reissverschluss

4.1.5.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$273.02\mu s$	$368.59\mu s$	$445.25\mu s$

Tabelle 4.8.: Szenario 1 / Simulation 1.5 / Datenangabe

Express-Frames

Die Übertragungszeiten der Express-Frames in dieser Simulation sind dieselben wie in Abbildung 4.4 auf Seite 58. Die Erläuterung dazu ist unter besagter Abbildung zu finden.

Frames mit der Priorität High

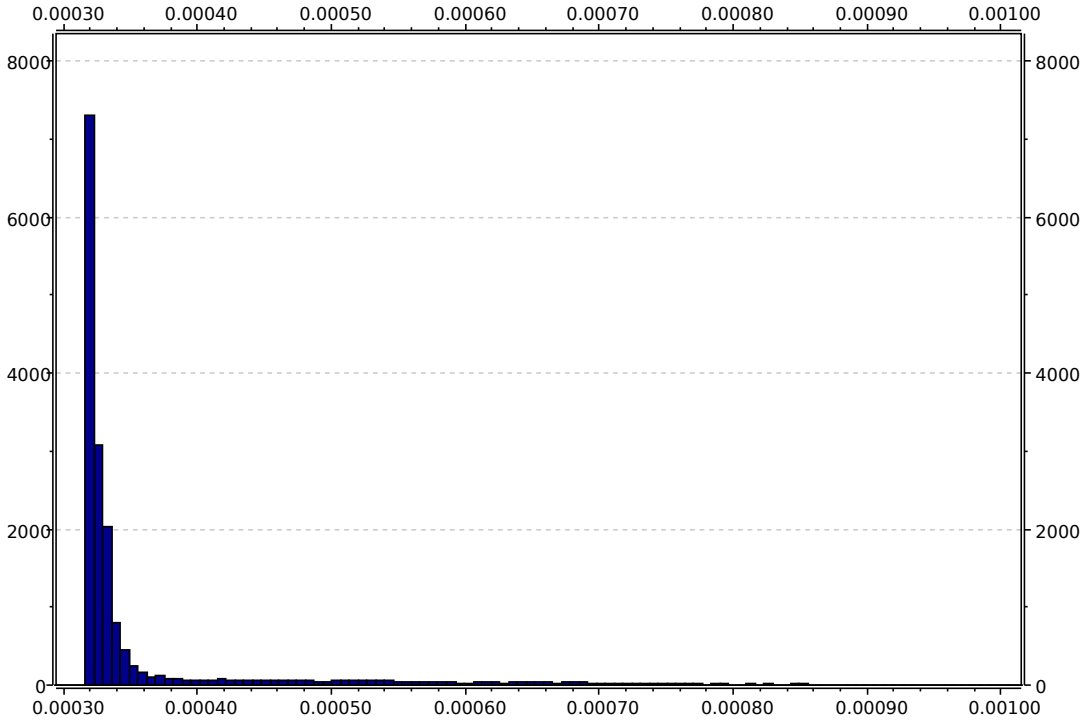


Abbildung 4.27.: Szenario 1 / Simulation 1.5 / Übertragungszeiten von High-Frames

Man kann gut erkennen, dass der grösste Teil aller High-Frames schnellstmöglich übertragen werden konnte. Jedoch existieren vereinzelte Frames, die aufgrund Preemption oder Frames, die gerade versendet werden, eine markante Verzögerung aufweisen.

Frames mit der Priorität Low

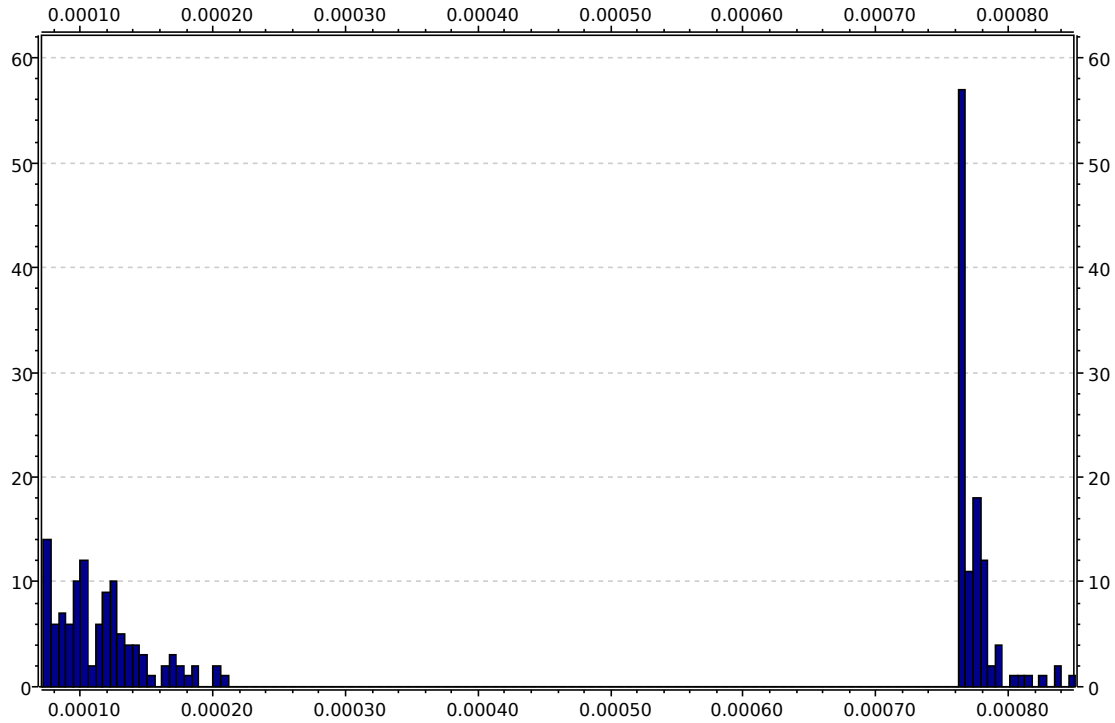


Abbildung 4.28.: Szenario 1 / Simulation 1.5 / Übertragungszeiten von Low-Frames

In dieser Grafik kann man klar feststellen, welches die Low-Frames à 64 Bytes und welche die mit 1500 Bytes sind. Dies resultiert daraus, dass hier zwischen zwei Knoten TCP-ähnlicher Unicast-Traffic simuliert wird. Kein Frame hat eine auffallend längere Übertragungsdauer, jedoch sieht man an der Zeitachse, dass die durchschnittliche Übertragungszeit deutlich länger ist, als diejenige der höher priorisierten Frames.

4.1.5.2. Queues

Frames mit der Priorität High

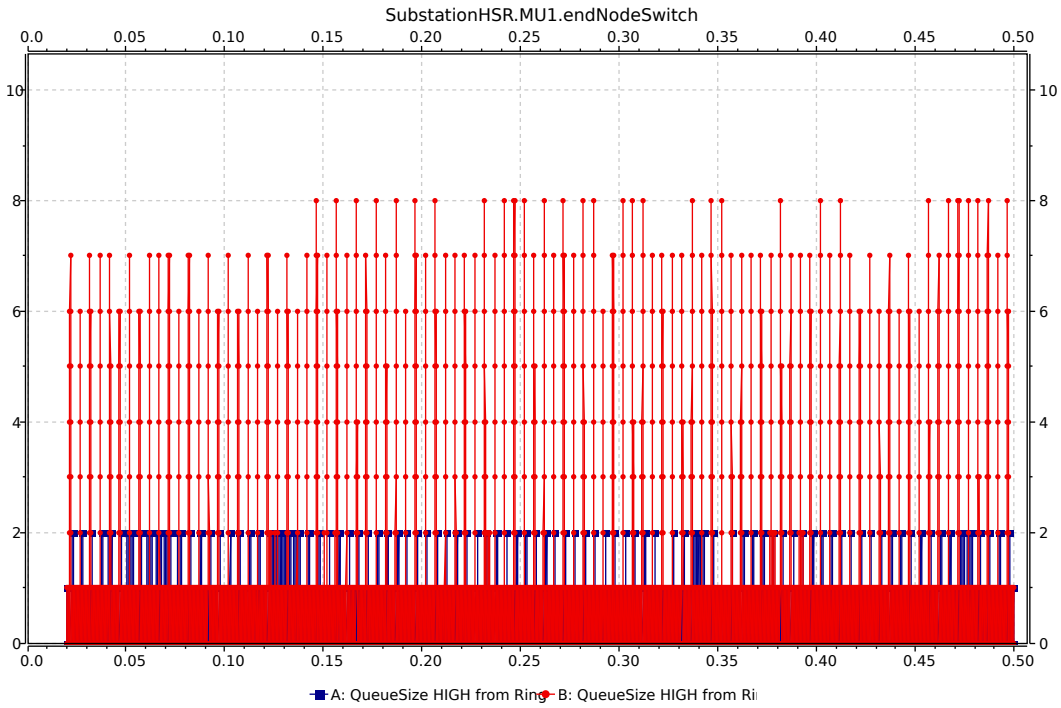


Abbildung 4.29.: Szenario 1 / Simulation 1.5 / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen mit Frames, die vom Ring kommen

Da von den anderen Knoten viel mehr Traffic eintrifft als intern (von «MU1») generiert wird, staut es sich bei der Ring-Queue mehr an, obwohl abwechslungsweise den Frames vom Ring und den intern generierten Frames der Vortritt gewährt wird. Wie auch bei vorherigen Simulationen kann der Stau wieder gelöst werden.

Frames mit der Priorität Low

Aufgrund der (verglichen mit den Frames anderer Prioritäten) niedriger Erzeugungsrate stauten sich bei den Frames mit der Priorität Low keine bei einem Ausgang eines Knotens an.

4.1.5.3. Anzahl Unterbrechungen (Preemptions)

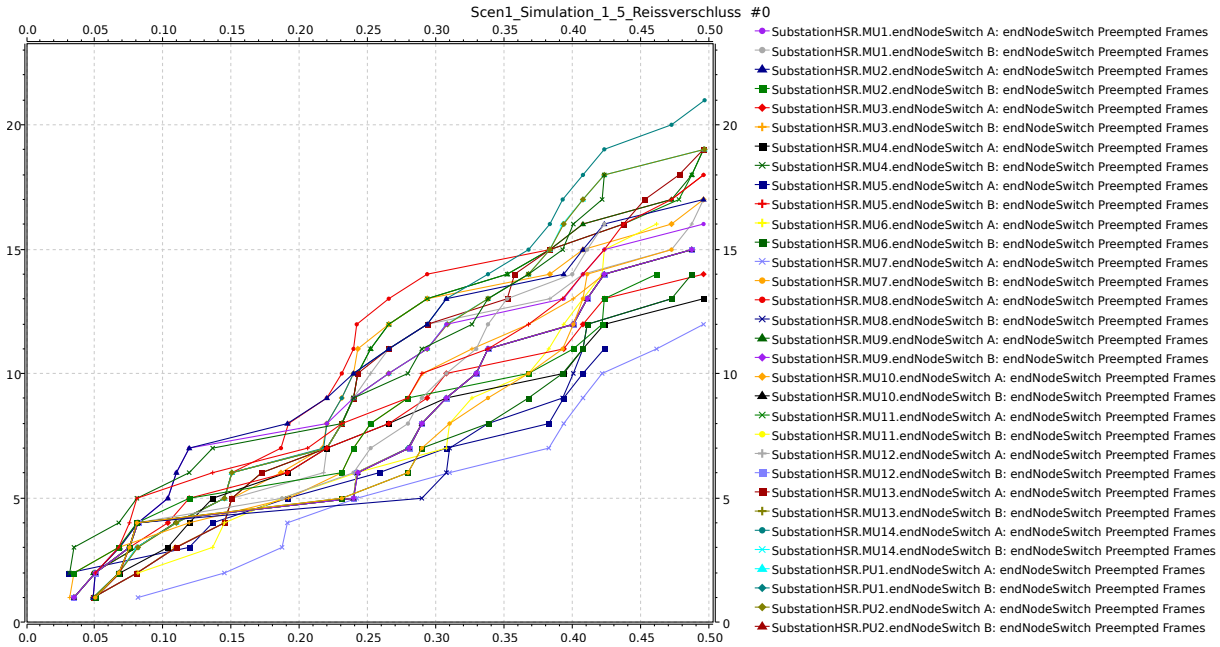


Abbildung 4.30.: Szenario 1 / Simulation 1.5: Anzahl Unterbrechungen durch Express-Frames

4.1.6. Simulation 1.6: Zeitschlitzverfahren

Es wurde ein Zeitschlitz mit einer Grün- und Rot-Phase von je 10000 Bytes definiert, wobei lediglich in der Grün-Phase High-Frames versendet werden können. Dies bedeutet, dass bei einer Übertragungsrate von 100 MBit/s eine Phase $800\mu s$ dauert.

4.1.6.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	75	17454	224
Mittelwert:	$263.68\mu s$	$1131.17\mu s$	$613.94\mu s$

Tabelle 4.9.: Szenario 1 / Simulation 1.6 / Datenangabe

Express-Frames

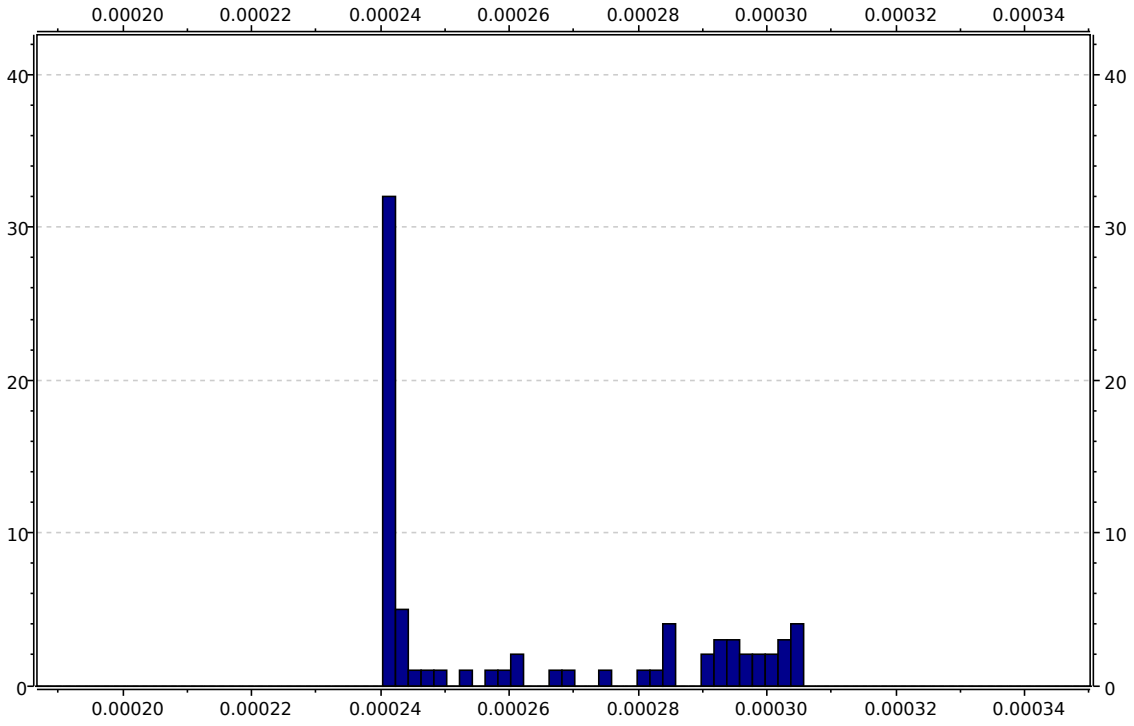


Abbildung 4.31.: Szenario 1 / Simulation 1.6 / Übertragungszeiten von Express-Frames

Die Express-Frames haben hier im Vergleich zu den anderen Simulationen im Durchschnitt die geringste Übertragungszeit, da hier am wenigsten High-Frames in einer bestimmten Zeit versendet werden und somit mehr Platz für Express-Frames freigeben.

Frames mit der Priorität High

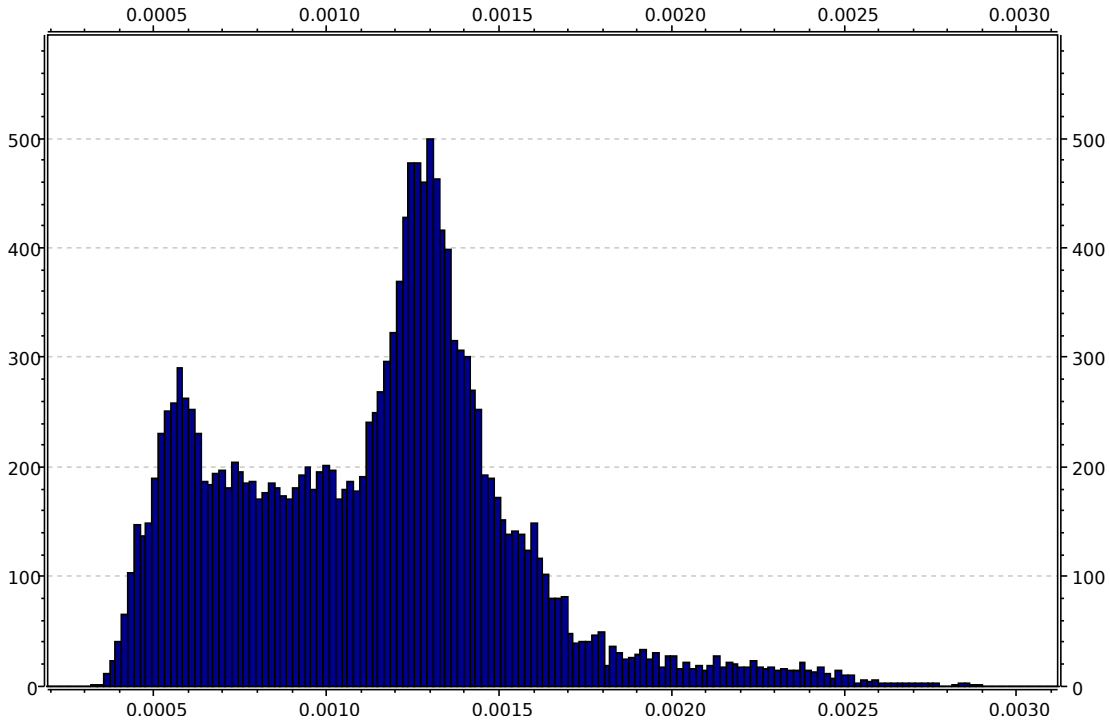


Abbildung 4.32.: Szenario 1 / Simulation 1.6 / Übertragungszeiten von High-Frames

Der Zeitschlitzmechanismus ist hier aufgrund der grossen Verteilung erkennbar. Hier gibt es High-Frames, die bis maximal ca. $3000\mu s$ benötigen, wobei bei den vorherigen Simulationen maximal ca. $1000\mu s$ für den Frame-Versand aufgewendet werden.

Frames mit der Priorität Low

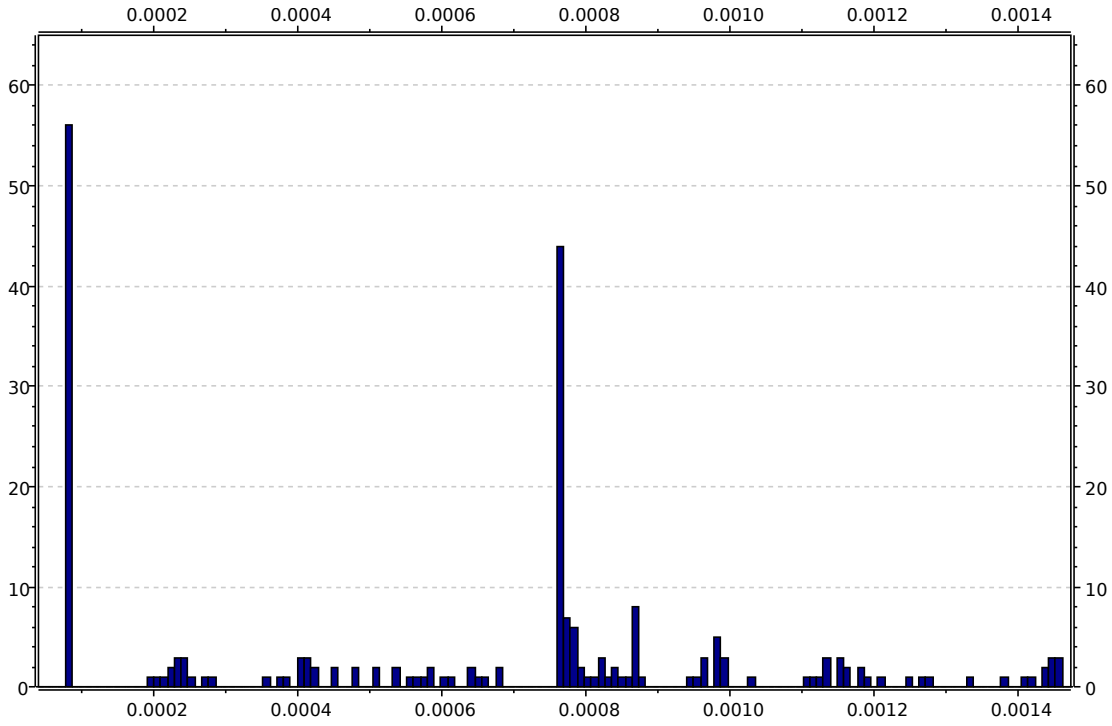


Abbildung 4.33.: Szenario 1 / Simulation 1.6 / Übertragungszeiten von Low-Frames

Dadurch, dass die High-Frames zurückgehalten werden, entsteht auch mehr Platz für das Übertragen von Low-Frames. Jedoch wird dieser Platz auch von Express-Frames beansprucht, weshalb sich die Frames länger als bei den vorherigen Simulationen ohne Zuflusslimitierung verzögern. Des Weiteren entstehen die längeren Verzögerungszeiten auch dadurch, dass die High-Frames prioritär behandelt werden.

4.1.6.2. Queues

Frames mit der Priorität High

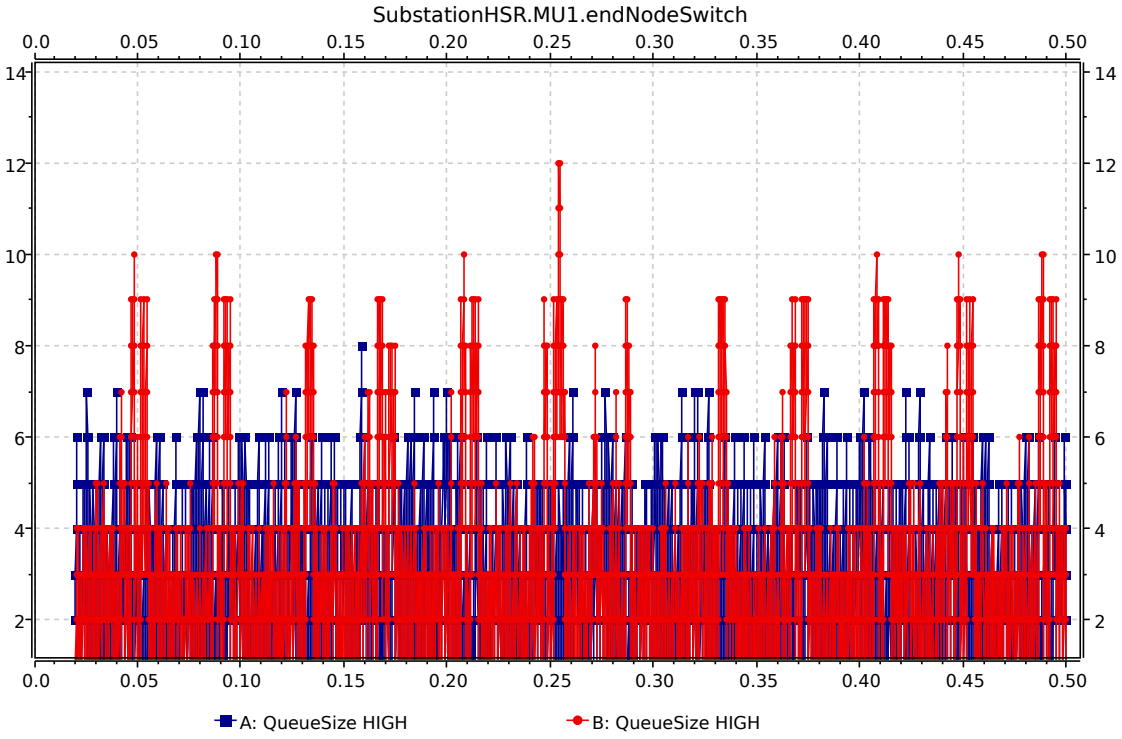


Abbildung 4.34.: Szenario 1 / Simulation 1.6 / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen

Es ist ein stärkerer Rückstau vorhanden als bei den bisherigen Simulationen, da in einer Rotphase keine High-Frames versendet werden können.

Frames mit der Priorität Low

Aufgrund der (verglichen mit den Frames anderer Prioritäten) niedriger Erzeugungsrate stauten sich bei den Frames mit der Priorität Low keine bei einem Ausgang eines Knotens an.

4.1.6.3. Anzahl Unterbrechungen (Preemptions)

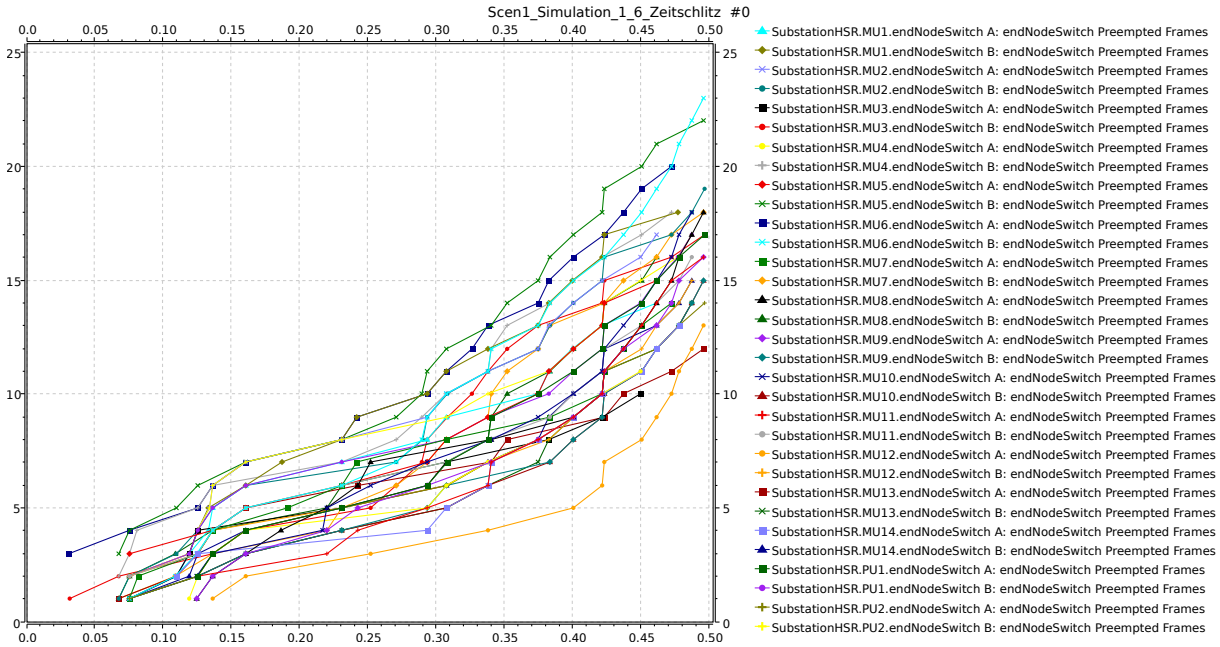


Abbildung 4.35.: Szenario 1 / Simulation 1.6: Anzahl Unterbrechungen durch Express-Frames

4.1.7. Simulation 1.7: Maximale Auslastung

Anstelle von 14 werden 17 MUs eingesetzt, um das Netz stärker auszulasten.

4.1.7.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	74	37098	0
Mittelwert:	$355.84\mu s$	$469.46\mu s$	–

Tabelle 4.10.: Szenario 1 / Simulation 1.7 / Datenangabe

Express-Frames

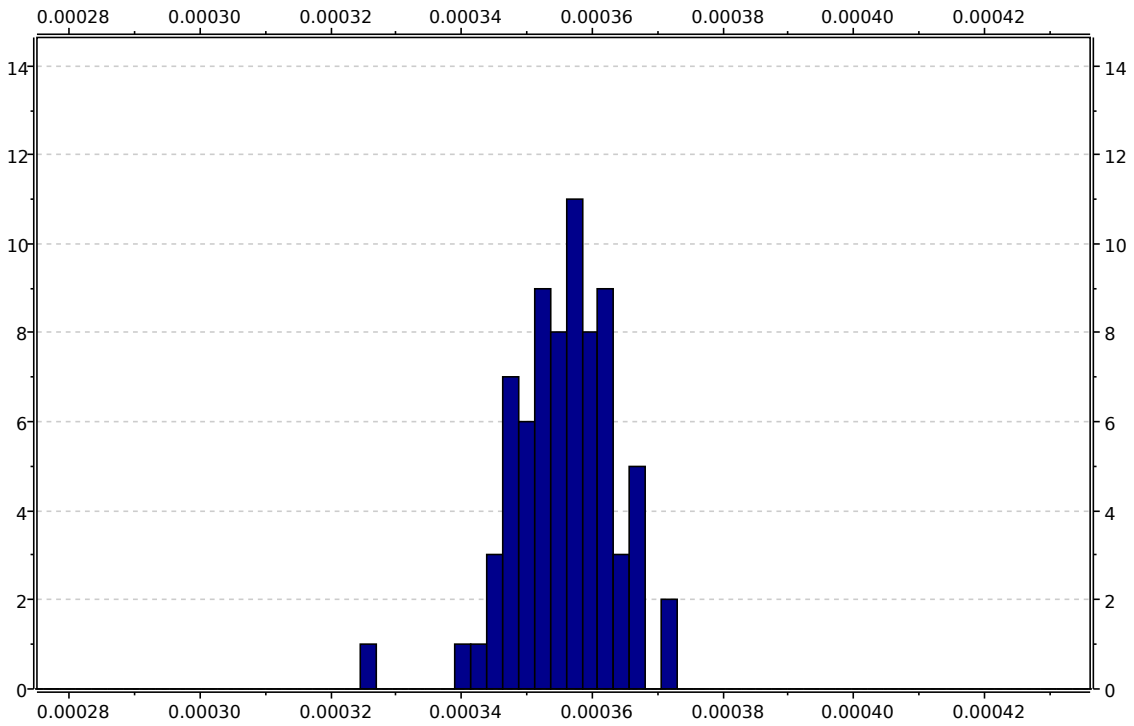


Abbildung 4.36.: Szenario 1 / Simulation 1.7 / Übertragungszeiten von Express-Frames

Da in dieser Simulation mehr Knoten vorhanden sind, durch die die Frames weitergeleitet werden müssen, entstehen automatisch längere Übertragungszeiten.

Frames mit der Priorität High

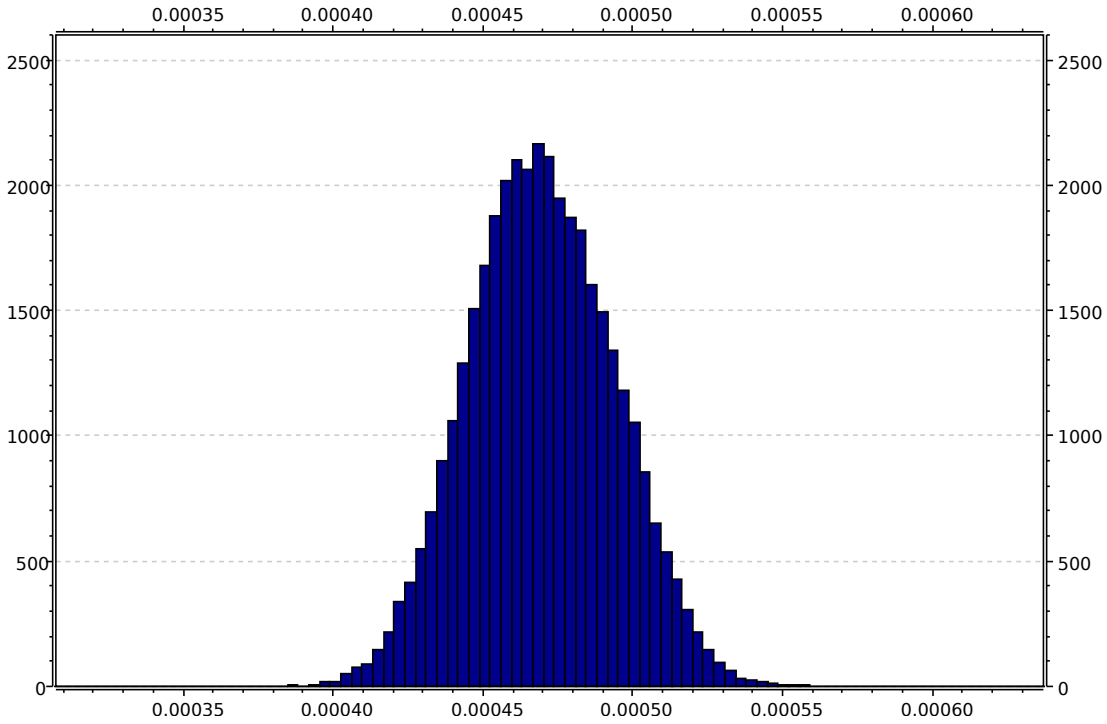


Abbildung 4.37.: Szenario 1 / Simulation 1.7 / Übertragungszeiten von High-Frames

Folgendes lässt sich über dieses und das vorherige Histogramm aussagen:

Aufgrund der hohen Netzauslastung haben nicht mehr die meisten Frames die geringste Übertragungszeit (siehe vorherige Simulationen ohne Zeitschlitzverfahren). Es fällt auf, dass die Werte in diesen beiden Histogrammen normalverteilt sind.

4.1.7.2. Queues

Frames mit der Priorität High

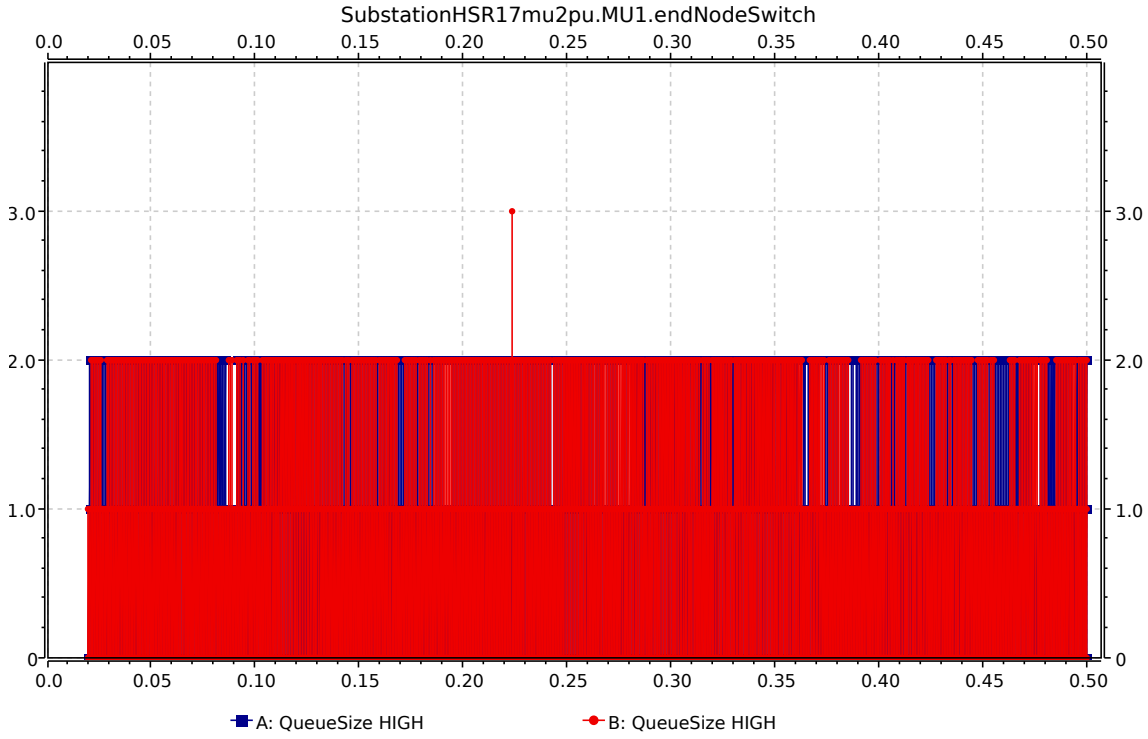


Abbildung 4.38.: Szenario 1 / Simulation 1.7 / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen

Hier lässt sich feststellen, dass das Netzwerk sich nahe an der Maximalauslastung befindet, da sich die Queues nie ganz abbauen, aber auch nicht sonderlich anstauen.

4.1.7.3. Anzahl Unterbrechungen (Preemptions)

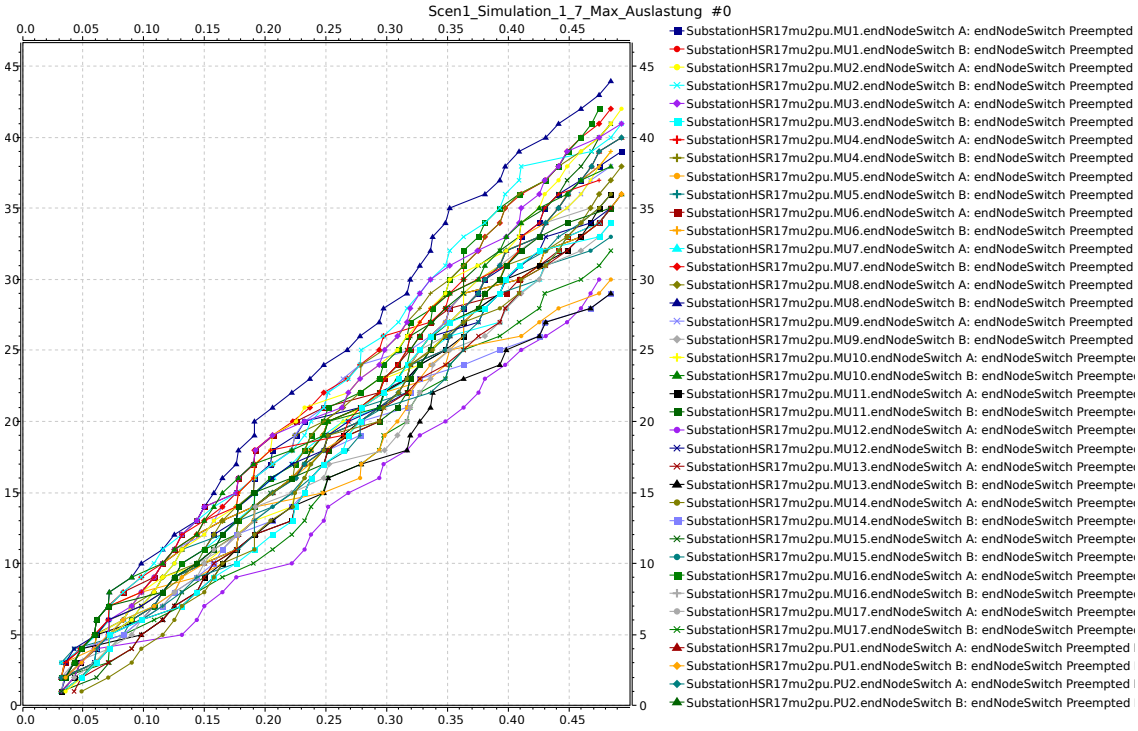


Abbildung 4.39.: Szenario 1 / Simulation 1.7: Anzahl Unterbrechungen durch Express-Frames

4.1.8. Simulation 1.8: Maximale Auslastung mit TCP-ähnlichem Verkehr

4.1.8.1. Übertragungszeiten

	Express	High	Low
Anzahl Frames:	74	37098	1118
Mittelwert:	$347.92\mu s$	$309.07\mu s$	$97475.52\mu s$

Tabelle 4.11.: Szenario 1 / Simulation 1.8 / Datenangabe

Express-Frames

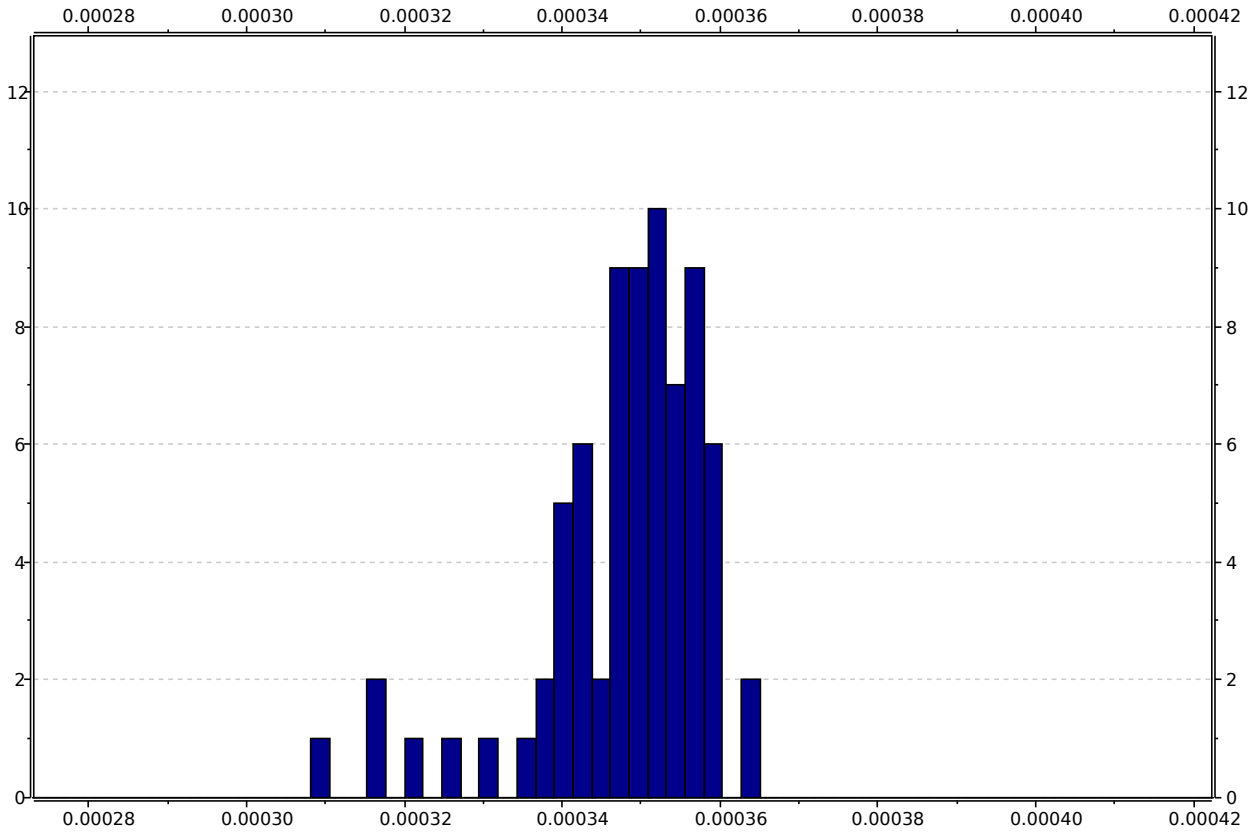


Abbildung 4.40.: Szenario 1 / Simulation 1.8 / Übertragungszeiten von Express-Frames

Frames mit der Priorität High

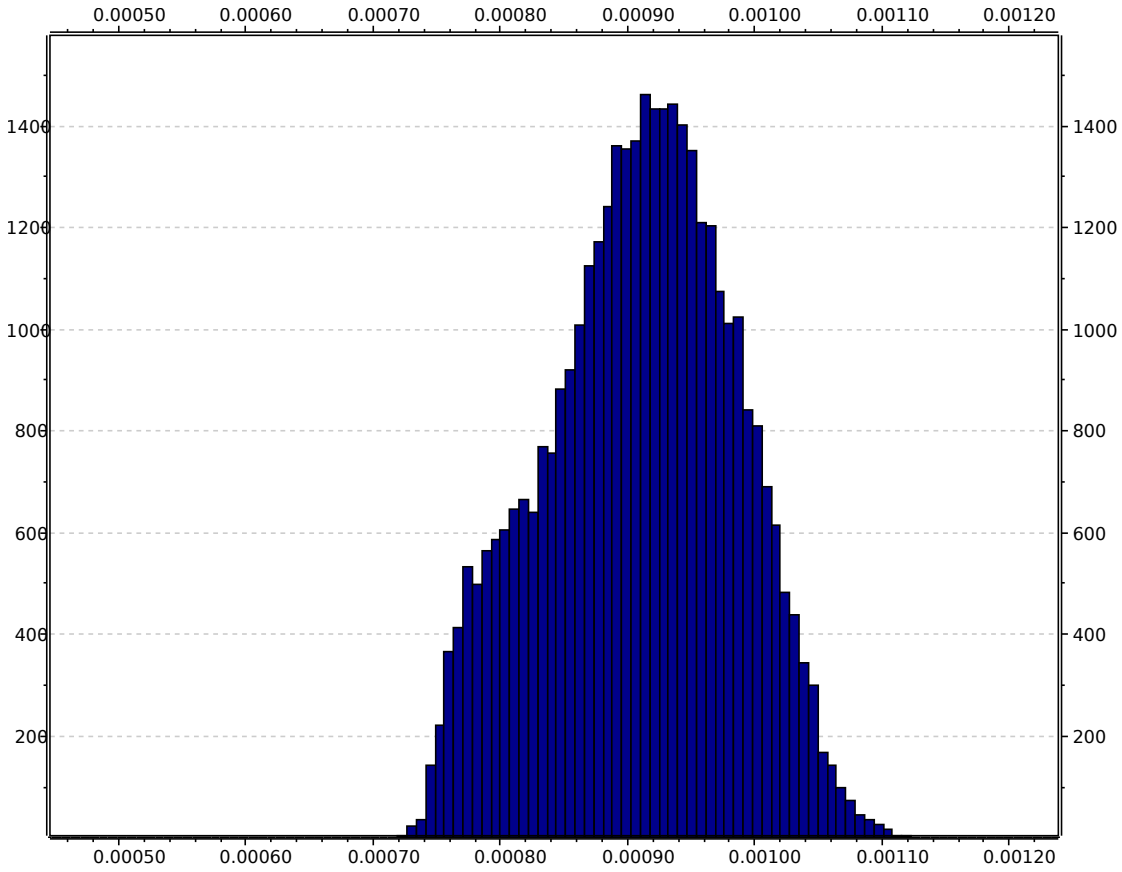


Abbildung 4.41.: Szenario 1 / Simulation 1.8 / Übertragungszeiten von High-Frames

Folgendes lässt sich über dieses und das vorherige Histogramm aussagen:

Da diese Simulation grundsätzlich dieselbe wie die vorherige ist, aber zusätzlich TCP-ähnlicher Traffic generiert wird, kann man erkennen, dass die Normalverteilung, wie sie in der Abbildung 4.37 auf Seite 91 zu sehen ist, hier eine grössere Abweichung aufweist.

Frames mit der Priorität Low

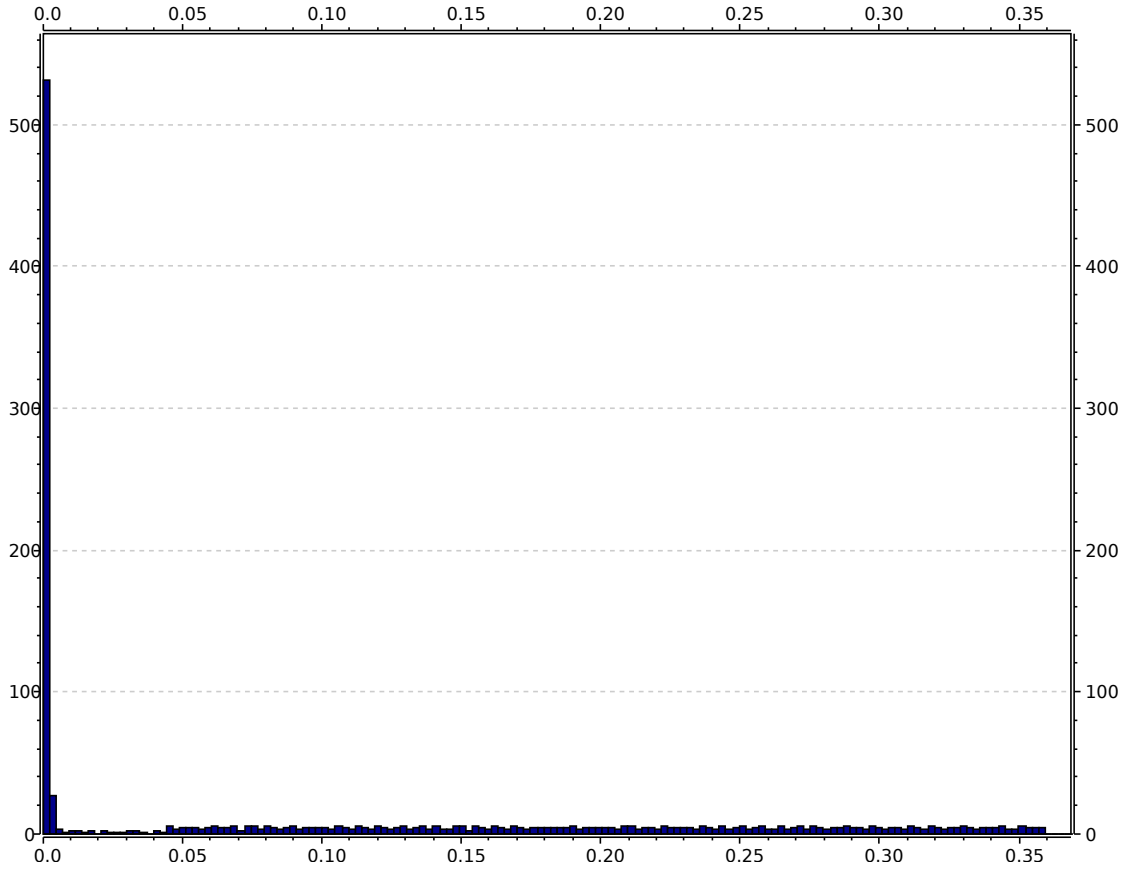


Abbildung 4.42.: Szenario 1 / Simulation 1.8 / Übertragungszeiten von Low-Frames

Die meisten Frames weisen eine Übertragungszeit zwischen 0 und $2200\mu s$ (Breite des ersten Balkens im Histogramm) auf. Da es jedoch aufgrund der Netzwerkauslastung eine sehr breite Streuung der Übertragungszeiten gibt, sind die Klassenbreiten des Histogramms gegenüber den beiden vorhergehenden Histogrammen vergleichsweise hoch.

4.1.8.2. Queues

Frames mit der Priorität High

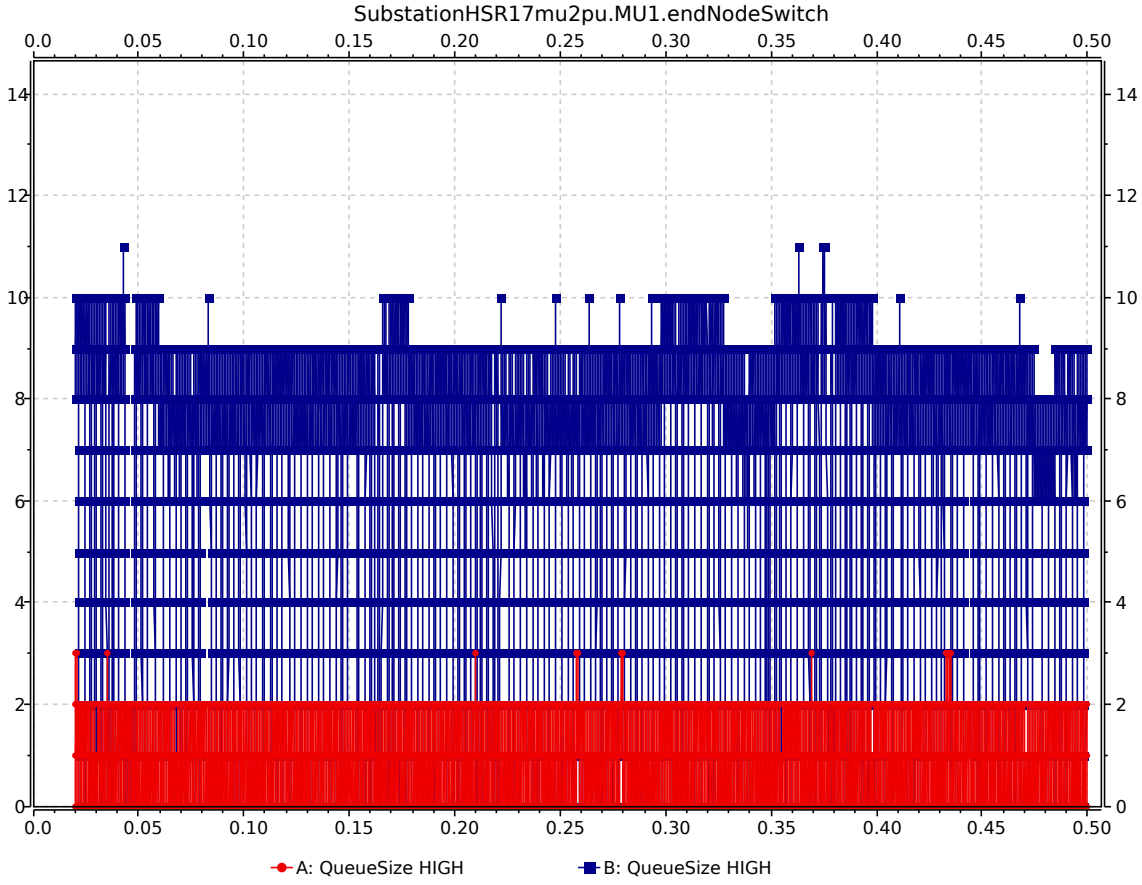


Abbildung 4.43.: Szenario 1 / Simulation 1.8 / Knoten «MU1»: Grösse der High-Queues bei beiden Ausgängen

Hier lässt sich feststellen, dass das Netzwerk sich nahe an der Maximalauslastung befindet, da sich die Queues nie ganz abbauen. Im Vergleich zur Simulation «Maximale Auslastung» ist hier erkennbar, dass sich ein grösserer Rückstau in den Queues bildet. Dies ist auf den zusätzlichen TCP-ähnlichen Traffic zurückzuführen, der bei dieser Simulation generiert wird.

Frames mit der Priorität Low

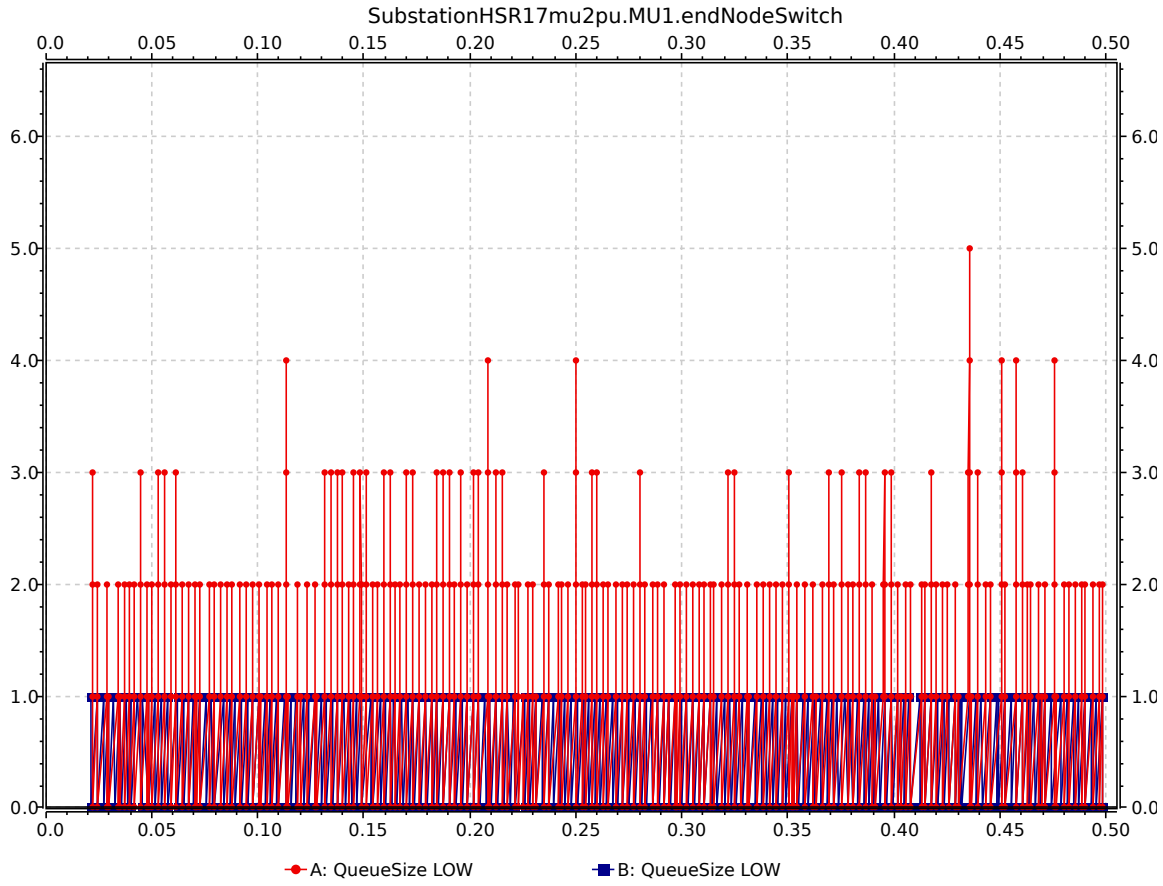


Abbildung 4.44.: Szenario 1 / Simulation 1.8 / Knoten «MU1»: Grösse der Low-Queues bei beiden Ausgängen

Auch hier ist die hohe Netzwerkauslastung erkennbar. Allerdings kann die Queue zeitweise ganz abgebaut werden. Dies ist so aufgrund der Häufigkeit, in der TCP-ähnlicher Traffic generiert wird.

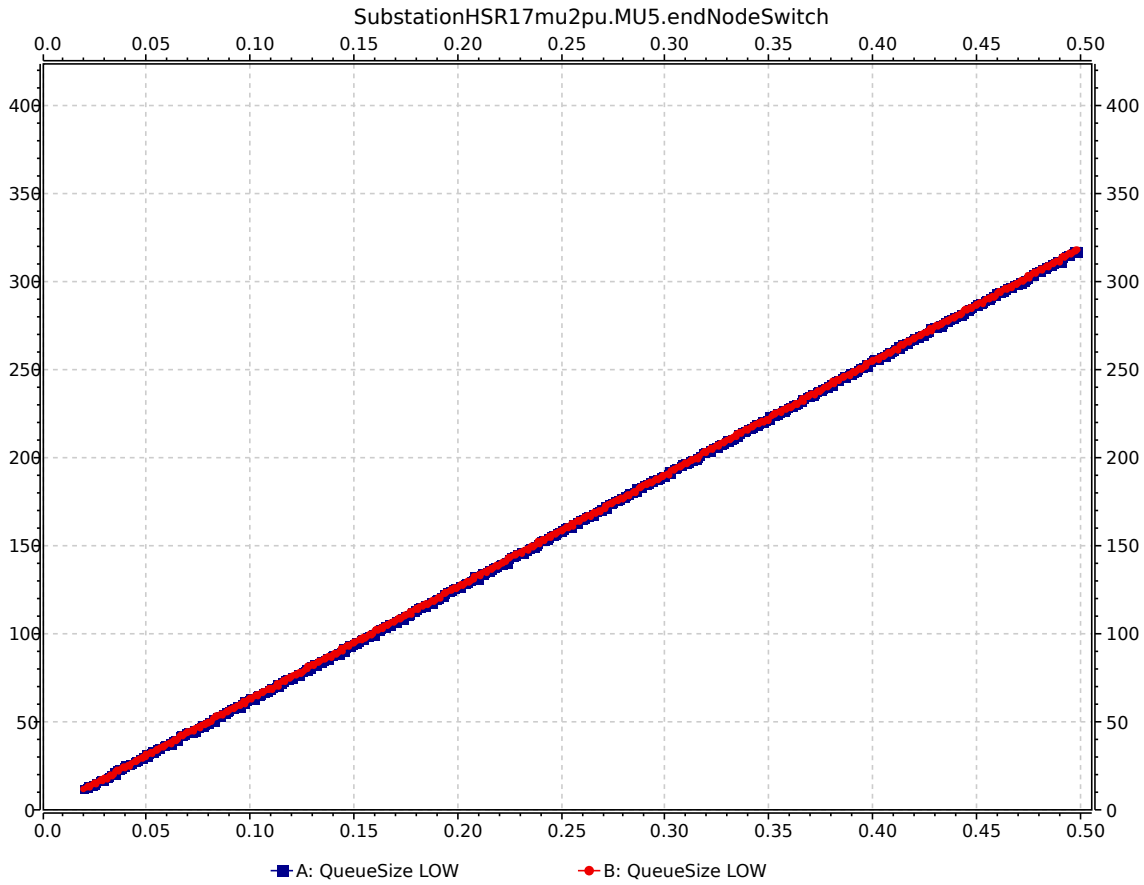


Abbildung 4.45.: Szenario 1 / Simulation 1.8 / Knoten «MU5»: Grösse der Low-Queues bei beiden Ausgängen

«MU5» generiert den TCP-ähnlichen Traffic. Die Low-Frames stauen sich in der Queue an, weil auf dem Ring ein enormes Datenaufkommen herrscht, das durch die höher priorisierten Frames der anderen Knoten verursacht wird.

4.1.8.3. Anzahl Unterbrechungen (Preemptions)

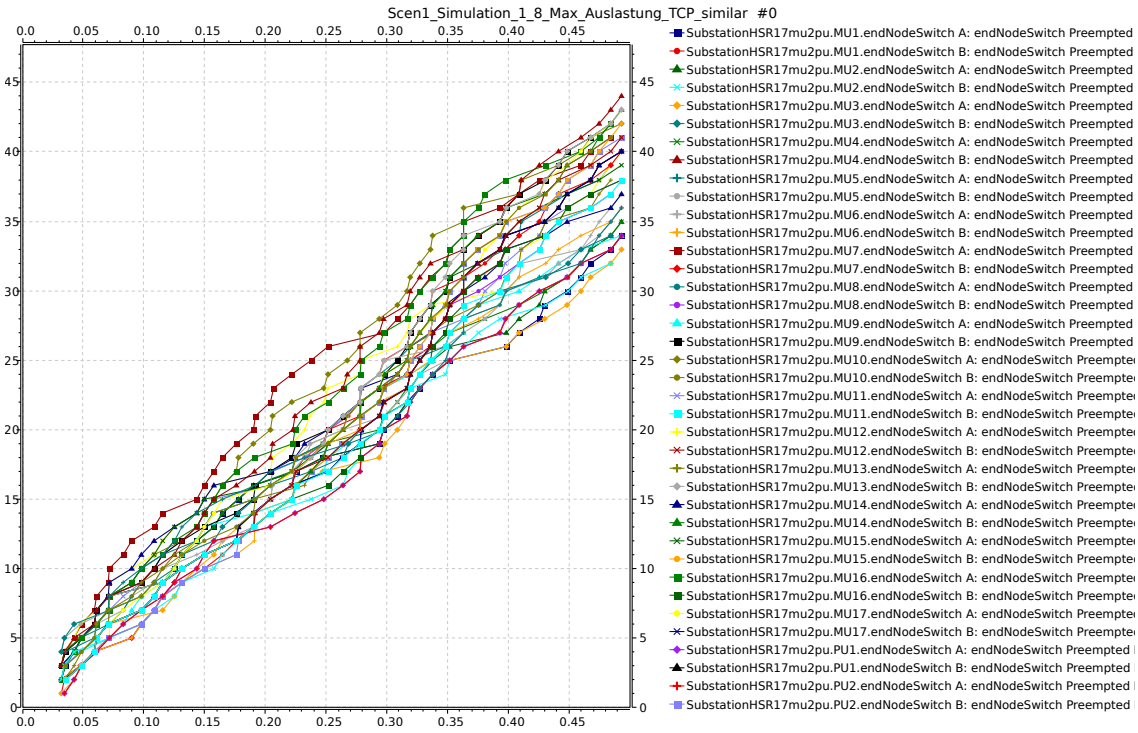


Abbildung 4.46.: Szenario 1 / Simulation 1.8: Anzahl Unterbrechungen durch Express-Frames

4.1.9. Fazit

Es wurden die durchschnittlichen Übertragungszeiten der Frames nach ihrer Priorität zusammengeführt und verglichen, um die bestmögliche Konfiguration zu erörtern. Dazu werden die Werte pro Priorität sortiert und dem kleinsten Wert die höchste Punktzahl (8 Punkte) vergeben. Befinden sich zwei unterschiedliche Werte in der selben Mikrosekunde, so erhalten sie die selbe Punktzahl.

Die Punkte werden je nach Priorität gewichtet und folgendermassen zusammengezählt:

$$E = Punkte\,für\,ExpressFrames$$
$$H = Punkte\,für\,Frames\,der\,Priorität\,High$$
$$L = Punkte\,für\,Frames\,der\,Priorität\,Low$$
$$Total\,Punkte = 3 * E + 2 * H + L$$

Sim #	Mittelwert $[\mu s]$ Express	Punkte E	Mittelwert $[\mu s]$ High	Punkte H	Mittelwert $[\mu s]$ Low	Punkte L	Total Punkte
1.1	272.92	6	368.83	6	445.25	8	38
1.2	274.13	4	369.06	5	446.28	7	29
1.3	273.02	5	368.48	6	445.25	8	35
1.4	269.93	7	347.18	7	12417.16	5	40
1.5	273.02	5	368.59	6	445.25	8	35
1.6	263.68	8	1131.17	3	613.94	6	36
1.7	355.84	2	469.46	4	-	3	17
1.8	347.92	3	309.07	8	97475.52	4	29

Tabelle 4.12.: Ranking der Simulationen nach ihren Mittelwerten (Übertragungszeiten)

Aus dieser Tabelle kann man entnehmen, dass im Gesamten die **Simulation 1.4 «Vortritt für Frames von Aussen mit Zuflusslimitierung von 88 Kbit/s»** (siehe Kapitel 4.1.4 auf Seite 71) die tiefsten durchschnittlichen Übertragungszeiten aufweist. Dies lässt sich dadurch erklären, dass der für unseren Anwendungsfall unwichtige Traffic (mit der Priorität Low) begrenzt wird und somit die höher priorisierten Frames mehr Platz auf dem Ethernet haben.

Bezüglich der durchschnittlichen Übertragungsdauer bei Express-Frames schneidet zwar die Simulation 1.6 «Zeitschlitzverfahren» (siehe Kapitel 4.1.6 auf Seite 85) besser ab, jedoch nur weil der Versand von High-Frames durch das Zeitschlitzverfahren eingeschränkt wird und somit diese nur in der Grün-Phase gesendet werden können. Wenn weniger High-Frames versendet werden, können die Express-Frames eher versendet werden, womit diese schneller ankommen. Da die High-Frames für unseren Anwendungsfall aber wichtige Daten beinhalten, ist dieser Mechanismus suboptimal für das Versenden von Messwerten.

Die durchschnittliche Übertragungsdauer der High-Frames ist gegenüber der Simulation 1.4 bei der Simulation 1.8 «Maximale Auslastung mit TCP-ähnlichem Verkehr» (siehe Kapitel 4.1.8 auf Seite 94) schneller, dies aber nur weil dort einiges mehr an High-Frames versendet wird und

diese somit den Express-Frames weniger Platz für die Fragmentierung lassen (siehe «Mittelwert Express» in der Tabelle 4.12 auf der vorherigen Seite).

Je nach Bedürfnis kann man also folgende Konfigurationen empfehlen:

- Wer auf den unwichtigen Traffic (TCP-ähnlicher Verkehr / Frames mit Priorität Low) verzichten kann, dem wird Simulation 1.4 «Vortritt für Frames von Aussen mit Zuflusslimitierung» empfohlen.

- Ist der Low-Traffic nicht ganz unwichtig (wenn u.a. Protokolle des Application Layers gebraucht werden und schnell reagieren müssen (z.B. Management-Konsole)), dann wird eine Konfiguration ohne Zuflusslimitierung wie auch ohne Zeitschlitzverfahren empfohlen.

5. Diskussion und Ausblick

5.1. Besprechung der Ergebnisse

Die umfangreichen Ergebnisse sind aufgrund der ausführlichen Überprüfung der Implementation (siehe Kapitel 3.2 auf Seite 35) aussagekräftig. Des Weiteren traten auch erwartete Ergebnisse ein, wie zum Beispiel, dass mit 17 MUs das Netzwerk an seine Grenzen stösst (siehe Simulation in Kapitel 3.3.1.7 auf Seite 52). Die Simulation von IET funktioniert wie erwartet und die Preemption von Frames der Priorität high und low kann zeitlich simuliert werden.

Anhand der aus den Simulationen resultierenden Graphen lassen sich Annahmen bestätigen, weisen einen verständlichen Ablauf auf und lassen auf klare Aussagen schliessen. Mittels dem vom unserem Betreuer vorgegebenen Anwendungsfall (siehe Besprechung in Kapitel 12.2.8 auf Seite 130) erlangen die Simulationen eine praxisnahe Relevanz. Es können Vorgaben zum Netzwerk, den Mechanismen und dem Lastprofil parametrisiert werden, um den Optimierungsbedarf heraus zu kristallisieren.

Mit den Daten, die der Arbeit auf einem USB-Stick beigelegt sind, können die Simulationen selbst durchgeführt und verifiziert werden.

5.2. Erfüllung der Aufgabenstellung

5.2.1. HSR-Knoten

5.2.1.1. Zwei Prioritäten

Soll	Der Knoten soll zwei Prioritäten unterstützen, d.h. zwei Warteschlangen pro Interface bewirtschaften.
Ist	Wenn zwei Frames mit unterschiedlichen Prioritäten (Low & High) zur gleichen Zeit eintreffen, wird das Frame mit der Priorität High als Erstes weitergesendet.
Nachweis	Die Frames werden in unterschiedliche Queues (Warteschlangen) gesetzt und diese werden der Reihe nach abgearbeitet (absteigend nach Priorität). Es existieren pro Scheduler 6 Queues, dessen Nummerierungen in einem Enum gespeichert sind:

```
1  typedef enum {
2      Express_RING,
3      Express_INTERNAL,
4      High_RING,
5      High_INTERNAL,
6      Low_RING,
7      Low_INTERNAL
8  } queueName;
```

Listing 5.1: hsrDefines.h - Enum mit Queuenummerierung

Der Switch analysiert das erhaltene Frame nach Priorität und Herkunft und teilt dem Scheduler mit, in welcher Queue er das Frame einordnen soll. Ein Ausschnitt dieser Analyse und Zuteilung (Herkunft des Frames ist im Beispiel von der CPU und wird an beide Ausgänge zum Ring zugeteilt) ist in Kapitel 15.1 auf Seite 145 einsehbar.

Nach der Zuteilung werden die Queues je nach Mechanismus im Scheduler abgearbeitet.

Tabelle 5.1.: Nachweis: Knoten unterstützt zwei Prioritäten

5.2.1.2. IET

Soll	Der Knoten soll Interspersing Express Traffic (IET) unterstützen, d.h. Express Frames können die aktuell ablaufende Übertragung eines Frames unterbrechen.
Ist	Die Unterbrechung findet nicht wirklich, sondern nur in der Zeitberechnung statt. Sobald ein Express-Frame ankommt wird es zu dem Zeitpunkt, zu dem ein Fragment des zu unterbrechenden Frames übertragen wurde, mit Interframe Gap (IFG), versendet. Danach wird die Zeit, die für die Übertragung des Express-Frames plus IFG und Grösse der neu entstandenen Felder der weiteren Fragmente verwendet wurde, dem «fragmentierten» Frame hinzugefügt. Dies hat für die Auswertung den selben Effekt wie wenn eine Fragmentierung stattgefunden hätte (Express-Frame wird während normalem Frame versandt, normales Frame kommt später an).
Nachweis	Der folgende Text bezieht sich auf den Codeausschnitt aus Kapitel 15.2 auf Seite 147: Der Scheduler prüft beim Abarbeiten seiner Queues («processOneQueue()»), ob derzeit normale Frames oder Express-Frames versendet werden (anhand der «transmitLock»-Variablen, werden mit den «unlock()»-Funktionen nach dem Senden eines Frames wieder aufgehoben). Wird derzeit nichts versendet, kann das Frame normal versendet werden. Wird derzeit ein normales Frame versendet und es steht der Versand eines Express-Frames an, wird anhand dem normalen Frame berechnet, wann das Express-Frame versendet werden kann («getExpressSendTime()»). Kann das normale Frame nicht fragmentiert werden, gibt «getExpressSendTime()» einen ungültigen Wert zurück, der von «processOneQueue()» abgefangen wird (dann wird das Express-Frame nachdem das normale Frame fertig versendet wurde bzw. beim nächsten Abarbeiten versendet). Gibt «getExpressSendTime()» einen gültigen Wert zurück, der besagt, wann das Express-Frame versendet werden kann, so wird ein Event erstellt, der zum berechneten Zeitpunkt das Express-Frame versendet. Das normale Frame wird entsprechend verlängert (siehe «Fverlängert» in Kapitel 3.1.5.2 auf Seite 30).

Tabelle 5.2.: Nachweis: Knoten unterstützt IET

5.2.1.3. Limitierung Ringzufluss

Soll	Der in den Ring einfliessende Traffic kann limitiert werden.
Ist	Es wurde der Mechanismus implementiert, dass Frames erst versendet werden können, sobald eine gewisse Anzahl an Tokens generiert wurde (es gibt ein Maximum an Tokens). Beim Versand werden diese Tokens verbraucht (ein Token pro Byte). Ein weiteres Frame kann erst wieder versandt werden, sobald genügend Tokens vorhanden sind. Für genauere Informationen zum Mechanismus siehe Kapitel 3.1.2.2 auf Seite 23.
Nachweis	In der Datei «omnetpp.ini» kann das Maximum an Tokens gesetzt werden. Wird kein Maximum gesetzt, wird der Traffic nicht limitiert.

```
1  ...
2  # Set a maximum of x Byte/s of generated low-traffic
3  **.endNodeSwitch.framebyteLimit = 12500
4  ...
```

Listing 5.2: omnetpp.ini - Definition Zuflusslimitierung

Der Codeausschnitt, in welchem die Zuflusslimitierung umgesetzt ist, ist in Kapitel 15.3 auf Seite 149 aufzufinden. In diesem Codeausschnitt befinden sich zwei Methoden, «containerHasEnoughBytes()» und «subtractFromByteContainer()» .

«containerHasEnoughBytes()» wird vor jedem Versand aufgerufen und gibt «false» zurück, wenn ein Low-Frame als Parameter mitgegeben wird und dessen Grösse die vorhandene Anzahl Tokens überschreitet. Zudem ist diese Funktion für die Token-Generierung zuständig.

In «subtractFromByteContainer()» wird bei der vorhandenen Anzahl Tokens die Anzahl abgezogen, die der Grösse des mitgegebenen Frames in Bytes entspricht. Abgezogen wird jedoch nur, wenn es sich um ein Low-Frame handelt. Diese Funktion wird beim Versenden aufgeführt und wird nur dann ausgeführt, wenn genügend Tokens vorhanden sind.

Tabelle 5.3.: Nachweis: Knoten kann Zufluss limitieren

5.2.1.4. Variierung der Vortrittsregeln

Soll	Die Vortrittsregeln bezüglich der im Ring zirkulierenden Frames und den Frames, die in den Ring einfliessen, können variiert werden (z.B. „zirkulierende Frames haben immer Vortritt“ oder „minimaler Zufluss wird garantiert“).
Ist	Bezüglich der Herkunft gibt es zwei Arten von Verkehr: Vom Ring und von Aussen auf den Ring. Es sind drei Arten von Vortrittsregeln implementiert: • Frames vom Ring haben Vortritt («zirkulierende Frames haben immer Vortritt») • Frames von Aussen haben Vortritt («minimaler Zufluss wird garantiert») • Reissverschluss (Der Vortritt für Frames vom Ring und von Aussen wechselt sich ab: Wenn ein Frame vom Ring Vortritt hatte, hat als Nächstes das Frame von Aussen Vortritt)

Nachweis	Der Mechanismus ist in der Datei «omnetpp.ini» definiert und wird vom Switch bei der Initialisierung in dessen Schedulern gesetzt.
	`...`
	`# Define Schedulermode ("FCFS", "RING_FIRST", "INTERNAL_FIRST", "ZIPPER")`
	`**.endNodeSwitch.schedulerMode = "ZIPPER"`
	
	`# Define Schedulermode for one device, for example EndNode "MU1"`
	`**.MU1.endNodeSwitch.schedulerMode = "FCFS"`
	`...`
	Listing 5.3: omnetpp.ini - Definition Vortritt-Mechanismus im Switch
	Ein Ausschnitt, bei dem aufgezeigt wird, wie die Vortritts-Mechanismen ablaufen, ist in Kapitel 15.4 auf Seite 151 einsehbar.

Tabelle 5.4.: Nachweis: Knoten verfügt über unterschiedliche Vortrittsregeln

5.2.1.5. Zeitschlitzverfahren

Soll	Der Knoten implementiert ein Zeitschlitzverfahren, welches dem zeitkritischen Traffic und dem Bulk Traffic je eine Phase zuordnet.
Ist	Es ist ein Mechanismus implementiert, welcher ein Intervall in zwei Phasen aufteilt. In der ersten Phase werden so viele Frames mit der Priorität High (zeitkritischer Traffic) versendet wie möglich. Wenn in dieser Phase noch Platz frei ist, es jedoch keine Frames mit der Priorität High mehr gibt, dann können Frames mit der Priorität Low (Bulk Traffic) auch innerhalb dieser Phase versendet werden. In der zweiten Phase können lediglich Frames der Priorität Low versendet werden. Express-Frames (gehören auch zum zeitkritischen Traffic) können zu jeder Zeit versendet werden.
Nachweis	Der Scheduler überprüft beim Abarbeiten seiner Queues, ob er sich in einer Phase befindet, in der er senden kann. Dazu ruft er die Funktion «timeslotIsValid()» (siehe Kapitel 15.5 auf Seite 153) mit der derzeitigen Queue als Parameter auf und sendet nur, wenn diese «true» zurück gibt. Diese Funktion gibt bei Express- und Low-Frames immer «true» zurück, da Express-Frames immer und Low-Frames auch in jeder Phase gesendet werden können, da diese nur gesendet werden, wenn kein High-Frame in den Queues existiert oder gesendet werden kann. Wird der Funktion eine Queue mit High-Frames übergeben, überprüft sie anhand der derzeitigen Zeit, ob das Frame gesendet werden kann oder nicht. Dazu wird folgende Rechnung angewandt: $$simtime = Derzeitige\,Simulationszeit\,(Jetztiger\,Zeitpunkt)\,[s]$$ $$phase = Konfigurierte\,Phasengrösse\,[s]$$ $$ret = Rückgabewert\,[Boolean]$$ $$ret = \begin{cases} simtime\%(2 * phase) < phase & true \\ simtime\%(2 * phase) \geq phase & false \end{cases}$$ «ret» enthält dabei den Rückgabewert der Funktion «timeslotIsValid()». Wenn «ret» = «true» ist, dann befindet sich die Simulation in der Grün-Phase und kann High-Frames versenden.

Tabelle 5.5.: Nachweis: Knoten kann nach Zeitschlitzverfahren arbeiten

5.2.2. Lastgenerator

Wie sich die Lastgenerierung eines einzelnen HSR-Nodes verhält, wird in der Konfigurations-XML-Datei festgelgt. Die Struktur dieser XML-Datei wird in Kapitel 3.1.4 auf Seite 27 genau erläutert. Diese XML-Konfiguration wird zu Beginn eines Simulationslaufs geladen und in einem CPU-Objekt gespeichert.

Das CPU-Modul eines HSR-Knotens (siehe Kapitel 3.2 auf Seite 26) löst die Generierung einzelner Frames je nach Konfigurationsparameter mittels sogenannter «Selfmessages» aus.

Wenn ein neues Frame erstellt werden muss, wird die Methode «generateOnePacket()» des CPU-Moduls aufgerufen. Diese Methode kann im Kapitel 15.6 auf Seite 154 eingesehen werden.

Im Rahmen dieser Arbeit wurden bei den Simulationen nur Lastgeneratoren verwendet, die eine konstante Last generieren. Das heisst, es wird nur die Konfiguration verwendet, welche Frames nach einem vorgegebenen Zeitintervall generiert. Der Zufallsanteil kann dabei über einen «epsilon»-Parameter definiert werden.

5.2.2.1. Konstante Framerate

Soll	Der Lastgenerator generiert für die Simulation ein Verkehrsaufkommen mit konstanter Frame-Rate.
Ist	Es lässt sich ein Strom mit konstanter Framerate generieren. Mittels Konfigurationsdatei kann man die Rate (in welchen Zeitabständen ein neues Frame generiert werden soll) bestimmen. Zudem kann über den Parameter «epsilon» (siehe Konfigurations-XML 3.1.4 auf Seite 27) eine maximale Abweichung des herkömmlichen Intervalls definiert werden. Somit wird auf das definierte Intervall ein Wert aufsummiert, der sich zwischen 0 und dem epsilon-Wert bewegt und mit einer Zufallsfunktion zustande kommt.
Nachweis	Der folgende Code-Ausschnitt zeigt auf, wie die Intervall-Zeit der Konstanten Lastgenerierung berechnet wird. Diese Zeit gibt an, wann das nächste Frame versendet werden muss.

```
1  ...
2  case BEHAVIOR_CONSTANT_LOADGEN:
3  {
4    if( delayedMessage->getSendData().last == 0 )
5    {
6      interval = delayedMessage->getSendData().interval;
7      if( delayedMessage->getSendData().epsilon > 0.0 )
8      {
9        epsilon = delayedMessage->getSendData().epsilon;
10       interval = uniform( interval, ( interval + epsilon ) );
11     }
12   }
13   else
14   {
15     randomLast = delayedMessage->getSendData().last;
16   }
17 }
18 ...
```

Listing 5.4: CPU.cc Intervall-Berechnung für Konstante Lastgenerierung

Wenn der epsilon-Wert = 0.0 konfiguriert wird, wird der Zufallsanteil ignoriert. Somit lässt sich eine Last generieren, die feste Intervall-Zeiten aufweist.

Tabelle 5.6.: Nachweis: Lastgenerator kann Frames mit konstanter Rate erzeugen

5.2.2.2. Zufällige zeitliche Verteilung

Soll	Der Lastgenerator generiert für die Simulation ein Verkehrsaufkommen mit einer zufälligen zeitlichen Verteilung der Frames.
Ist	Es kann ein Verkehrsaufkommen generiert werden, bei dem die Zeitabstände zwischen den Frames zufällig gewählt werden. Dies wird erreicht, durch setzen eines Intervalls und einem epsilon-Wert.
Nachweis	Siehe Nachweis in Kapitel 5.2.2.1 auf der vorherigen Seite.

Tabelle 5.7.: Nachweis: Lastgenerator kann Frames mit zufälliger zeitlicher Verteilung erzeugen

5.2.2.3. Spontane Einzelmeldungen

Soll	Der Lastgenerator generiert für die Simulation ein Verkehrsaufkommen mit spontanen Einzelmeldungen.
Ist	Spontane Einzelmeldungen in Form von Express-Frames können generiert werden. Diese werden per Zufall versandt.
Nachweis	Die spontanen Einzelmeldungen werden mit einem Zeitintervall und einem verhältnismässig grossen «epsilon»-Wert parametrisiert. So kann das Verhalten spontaner Einzelmeldungen simuliert werden. Dieser Mechanismus wird in den Simulationen verwendet, um Express-Frames zu versenden, damit diese mit einer zufällig gewählten Zeit spontan versendet werden (siehe 3.14 auf Seite 50).

Tabelle 5.8.: Nachweis: Lastgenerator kann spontane Einzelmeldungen versenden

5.2.3. Durchführbarkeit der Simulationen und Interpretation der Resultate

Soll	Das Zeitverhalten der verschiedenen Weiterleitungsvarianten soll durch entsprechende Simulationsläufe ermittelt werden. Die Resultate sind zu vergleichen und zu interpretieren.
Ist	Verschiedene Weiterleitungsvarianten können simuliert und analysiert werden. Statistiken können zudem generiert werden, was den Vergleich der Varianten untereinander vereinfacht. Die Weiterleitungsvarianten sind in Kapitel 3.1.2 auf Seite 22 aufgeführt. Auf dem Datenträger, der dieser Dokumentation beiliegt, sind die Konfigurationen der Simulationen enthalten. Die Simulationen können somit erneut, oder aber auch mit neuen modifizierten Parametern durchgeführt werden.
Nachweis	Die Konfigurationen der einzelnen Simulationen können dem Kapitel 3.3 auf Seite 49 entnommen werden. Deren Resultate und Interpretationen sind ausführlich dokumentiert in Kapitel 4 auf Seite 54.

Tabelle 5.9.: Nachweis: Simulationen durchführbar und interpretierbar

5.3. Rückblick

Alle Anforderungen der Aufgabenstellung wurden aus unserer Sicht erreicht und überprüft (siehe Kapitel 5.2 auf Seite 104).

Nach einigen Anlaufschwierigkeiten (Code-Refactoring und fehlendem Quellcode bei der vorherigen Arbeit), ging die Einarbeitungsphase in OMNeT++ und IET gut von Statten. Auch die Struktur der Dokumentation war schnell aufgebaut. Ein sehr grosser Vorteil bei unserem Vorgehen war, dass die Dokumentation parallel zu den Programmierarbeiten fortlaufend gepflegt wurde und somit keine Informationen verloren gingen.

Was den Implementationsstand bei Drucklegung dieser Arbeit betrifft, ist die Modellierung eines HSR-Knotens gemäss Aufgabenstellung so weit abgeschlossen. Was derzeit noch nicht fertig implementiert ist, ist die Modellierung einer RedBox mit Interlink-Ports. Die Implementation wurde teilweise parallel zum HSR-Knoten erstellt, ist jedoch aus Zeitmangel nicht abgeschlossen. Wenn versucht wird eine RedBox in eine Simulation einzubauen wird derzeit noch eine Exception geworfen.

Neben der fortlaufenden Pflege der Dokumentation erstellten wir für das Projekt ein Wiki, in dem die neusten Erkenntnisse und Besprechungen festgehalten wurden und das unser Betreuer jederzeit einsehen konnte. Für die Entwicklung der Implementation und der Dokumentation verwendeten wir ein Versionsverwaltungssystem («Git» mit «GitHub» als Frontend). So wussten alle Projektbeteiligten zu jeder Zeit was zu tun ist und was noch getan werden muss.

Die Besprechungen mit unserem Betreuer erwiesen sich als interessant und lehrreich. Des Weiteren wurde die Arbeit zwischen Studenten und Dozenten auf einer äusserst kooperativen Ebene umgesetzt und diskutiert. Schade ist, dass solche Arbeiten während eines ereignisreichen Semesters durchgeführt werden müssen und somit zu wenig Zeit für allfällige Detailpflege und weitere Simulationen übrig bleibt.

Gerne würden wir eine weitere Arbeit in diesem Themengebiet und mit den Teilnehmern dieses Projektes in Anlauf nehmen.

Zusammenfassend kann ein positives Fazit gezogen werden: Das Projekt wurde mit grossem Interesse erarbeitet und wurde unserer Meinung nach erfolgreich umgesetzt.

5.4. Ausblick

Mittels der implementierten Simulationsumgebung ist es möglich, IET in HSR-Netzwerken zu simulieren, wobei es sich bei IET um einen Standard handelt, der bei Drucklegung dieser Arbeit noch immer im Entwicklungsstadium ist.

Die Implementierung würde sich beliebig erweitern lassen. Wir sehen Potenzial in der Umsetzung von Verbindungen zwischen HSR- und Nicht-HSR-Netzwerken sowie der Kopplung verschiedener HSR-Ringe.

Des Weiteren könnten zusätzliche Mechanismen entwickelt und in Simulationen beobachtet werden. Zum Beispiel wäre die Umsetzung eines Schedulers mit einem ähnlichen Verfahren zum «Completely Fair Scheduler (CFS)» innerhalb der gleichpriorisierten Queues denkbar.

Mit der hier entwickelten Implementation könnte man selbstverständlich weitere Szenarien und Simulationen konfigurieren, durchführen und analysieren. Dank der Daten, die dieser Arbeit beiliegen, ist dies problemlos möglich.

Teil III.

Verzeichnisse

6. Literaturverzeichnis

[1] Gemperli, Alfred: *Vertiefungsarbeit: HSR Simulation mit OMNeT++*. Technischer Bericht, Institute of Embedded Systems, ZHAW School of Engineering, Juli 2011.

[2] IEEE 802.3 DMLT Study Group: *Distinguished Minimum Latency Traffic in a Converged Traffic Environment (DMLT) @http://www.ieee802.org/3/DMLT/*, Dezember 2013.

[3] IEEE Computer Society, LAN/MAN Standards Committee of the: *IEEE P802.3br/D1.0 Draft Standard for Ethernet Amendment: Specification and Management Parameters for Interspersing Express Traffic*, Dezember 2014.

[4] IEEE P802.3br IET Task Force: *Interspersing Express Traffic (IET) Project Authorization Request (PAR) @http://www.ieee802.org/3/br/P802d3br_PAR.pdf*, September 2013.

[5] IEEE P802.3br IET Task Force: *Interspersing Express Traffic (IET) Baseline @http://www.ieee802.org/3/br/Baseline/8023-IET-TF-1405_Winkel-iet-Baseline-r3.pdf*, Juni 2014.

[6] IEEE P802.3br IET Task Force: *Interspersing Express Traffic (IET) Schedule @http://www.ieee802.org/3/br/8023-IET-TF-1401_IET_schedule.pdf*, Januar 2014.

[7] Institute of Embedded Systems: *HSR-Konzept @http://ines.zhaw.ch/de/engineering/institute-zentren/ines/forschung-und-entwicklung/high-availability/hsr.html*, September 2014.

[8] Kirrmann,Hubert: *HSR – High Availability Seamless Redundancy @http://lamspeople.epfl.ch/kirrmann/Pubs/IEC_62439/IEC_61439-3/IEC_62439-3.5_HSR_Kirrmann.ppt*, August 2010.

[9] NetModule AG: *Applying PRP and HSR Protocol for Redundant Industrial Ethernet @http://www.netmodule.com/en/technologies/interfaces_networks/IEC62439*, September 2014.

[10] OMNeT++ Community: *INET Framework for OMNeT++ Manual @http://inet.omnetpp.org/doc/INET/inet-manual-draft.pdf*, Juni 2012.

[11] OMNeT++ Community: *OMNeT++ User Manual @http://www.omnetpp.org/doc/omnetpp/manual/usman.html*, November 2014.

[12] OMNeT++ Community: *OMNeT++ Wiki - Setting up Parallel Simulations @http://www.omnetpp.org/pmwiki/index.php?n=Main.SettingUpParallelDistributedSimulations*, November 2014.

[13] Weibel, Hans: *PA14_wlan_1: Projektarbeit im Fachgebiet Kommunikation*. Aufgabenstellung, September 2014.

7. Glossar

EthernetII Frame Definiert eine Struktur, um Nutzdaten über ein Ethernet-Netzwerk zu versenden. Es ist die kleinste adressierbare Einheit auf dem Link Layer. Die Struktur eines EthernetII-Frames gliedert sich wie folgt: Zieladresse, Quelladresse, VLAN-Tag (optional), Ethertype (Ethernet II), Nutzdatenanteil, Frame Check Sequence (CRC).

HSR **High Availability Seamless Redundancy**. Redundanzprotokoll für Ethernet basierte Netzwerke. HSR ist für redundant gekoppelte Ringtopologien ausgelegt. Die Datenübermittlung innerhalb eines HSR-Rings ist im Fehlerfall gewährleistet, wenn eine Netzwerkschnittstelle ausfallen sollte.

IET **Interspersed Express Traffic**. Erlaubt es Frames während des Sendevorgangs zu unterbrechen, um sogenannte Express-Frames so schnell wie möglich versenden zu können.

IFG **Interframe Gap**. Minimaler zeitlicher Abstand zwischen dem Versand zweier Frames.

Link Layer Stellt sicher, dass Daten über eine physikalische Schnittstelle von und zu einem Netzwerk transferiert werden können.

LyX LyX ist ein Front-End-Editor für das Textsatz-System LaTeX. Das freie Softwarepaket kann unter `http://www.lyx.org` heruntergeladen werden und ist für die Betriebssysteme Linux, Windows und Mac OS X verfügbar.

mFrame MAC Merge Frame ist ein Format eines Ethernet Frames. Im Gegensatz zu herkömmlichen EthernetII Frames, stellt das mFrame Format Parameter und Strukturen zur Verfügung, die es ermöglichen, das Ethernet Frame zu fragmentieren.

Preemption Unter Preemption versteht man den Vorgang, welcher das Unterbrechen der Frames im IET ausführt. Frames, die keine Express-Priorität haben, können während ihres Sendevorgangs von Express-Frames unterbrochen werden. Die Preemption wird im Rahmen dieser Arbeit lediglich mittels Zeitrechung vollzogen, das heisst, es werden keine Frames effektiv unterbrochen, jedoch deren Übertragungszeiten künstlich verlängert. Das Ergebnis ist dasselbe wie bei einer effektiven Fragmentierung.

PRP **Parallel Redundancy Protocol**. Hat ein ähnliches Funktionsprinzip wie HSR, aber auch Unterschiede wie z.B. das Frameformat. PRP- und HSR-Netze können jedoch über RedBoxen kommunizieren.

Selfmessage In OMNeT++ kann ein Modul sich selber eine Selfmessage senden. Diese Selfmessage nimmt als Parameter eine Verzögerungszeit. Nach dieser Verzögerungszeit wird im selben Modul ein Event ausgelöst um die Selfmessage zu verarbeiten. Diese Methode wird beispielsweise verwendet, um immer wiederkehrende Ereignisse in Modulen auszulösen.

Steady-State Eingeschwungener, stabiler Zustand der Simulation. Damit keine verfälschten Simulationsresultate auftreten, werden die Messwerte in einem Zeitintervall aufgenommen, welches einige Zeit nach dem Simulationsstart beginnt und einige Zeit vor der Generierung der letzten Frames endet.

8. Abbildungsverzeichnis

9. Tabellenverzeichnis

10. Listingverzeichnis

Teil IV.

Anhang

11. Offizielle Aufgabenstellung

1 Ausgangslage In Anlagen für die Automatisierung der elektrischen Energieversorgung hat sich Ethernet gut etabliert. Ein Anwendungsfeld ist jedoch noch mit Unsicherheiten behaftet: der Prozessbus von Unterstationen. Bei dieser Anwendung werden extrem viele Messdaten erfasst und übertragen. Gleichzeitig soll das Netzwerk Steuerbefehle (z.B. für Notabschaltung) mit sehr geringer Verzögerung übertragen können.

Um höchste Verfügbarkeit zu garantieren wird das Ethernet in einer Ringtopologie betrieben. Das Redundanzverfahren heisst HSR (High-availability Seamless Redundancy) und arbeitet verlustfrei, d.h. es übersteht den Ausfall einer Komponente oder eines Links, ohne dass Frames verloren gehen.

Es gibt verschiedene Ansätze, die Verzögerung kritischer Frames zu garantieren.

a) Die Erhöhung der Datenrate (in diesem Fall von 100 MBit/s auf 1 GBit/s) ist naheliegend. Damit kann das Problem aber nicht prinzipiell gelöst, sondern lediglich auf ein anderes Niveau verschoben werden. Diesen "Brute Force"- Ansatz möchte man wegen den damit verbundenen sehr viel höheren Anforderungen an die Hardware wenn möglich vermeiden und stattdessen lieber einen effizienten Algorithmus verwenden.

b) Wenn es um sehr zeitsensitive Anwendungen geht, hat Ethernet generell das Problem, dass ein langes Frame, dessen Aussendung schon begonnen hat, die Aussendung eines hoch priorisierten Frames verzögert. Das Zeitverhalten könnte mit einem Pre-Emption-Mechanismus verzögert werden, welcher es erlaubt, das Versenden eines langen Frames zu unterbrechen und später wieder aufzunehmen. In der Standardisierung gibt es Bestrebungen, einen solchen Mechanismus einzuführen.

c) Durch ein zeitgesteuertes Scheduling kann man Zeitfenster für kritische Kommunikation reservieren und somit Verzögerungszeiten garantieren.

2 Aufgabenstellung In der Arbeit soll untersucht werden, welchen Effekt die zur Diskussion stehenden Massnahmen für einen konkreten Anwendungsfall bringen. Das umfasst folgende Tätigkeiten:

2.1 Modell für HSR-Knoten erweitern Das betrachtete Netzwerk ist ein HSR-Ring. Die bestehende Simulationsumgebung soll so erweitert bzw. angepasst werden, dass folgende Funktionen/Mechanismen simuliert werden können:

a) Der Knoten soll zwei Prioritäten unterstützen, d.h. zwei Warteschlangen pro Interface bewirtschaften.

b) Der Knoten soll Interspersing Express Traffic (IET) unterstützen, d.h. Express Frames können die aktuell ablaufende Übertragung eines Frames unterbrechen.

c) Der in den Ring einfliessende Traffic kann limitiert werden.

d) Die Vortrittsregeln bezüglich der im Ring zirkulierenden Frames und den Frames, die in den Ring einfliessen, können variiert werden (z.B. „zirkulierende Frames haben immer Vortritt“ oder „minimaler Zufluss wird garantiert“).

e) Der Knoten implementiert ein Zeitschlitzverfahren, welches dem zeitkritischen Traffic und dem Bulk Traffic je eine Phase zuordnet.

2.2 Lastmodell beschreiben und implementieren
Das durch die Anwendung generierte Verkehrsaufkommen ist zu studieren und zu beschreiben. Lastgeneratoren sollen implementiert werden, die das Verkehrsaufkommen für die Simulation generieren durch die Überlagerung von Strömen mit folgender Charakteristik:

a) konstante Framerate

b) zufällige zeitliche Verteilung der Frames

c) spontane Einzelmeldungen

2.3 Simulationen durchführen und Resultate interpretieren
Das Zeitverhalten der verschiedenen Weiterleitungsvarianten soll durch entsprechende Simulationsläufe ermittelt werden. Die Resultate sind zu vergleichen und zu interpretieren.

3 Ziele

- Es liegt eine lauffähige und ausreichend dokumentierte Simulationsumgebung vor, welche
  - die verschiedenen Weiterleitungsvarianten implementiert,
  - Traffic unterschiedlicher Charakteristik generieren kann,
  - die Laufzeit der einzelnen Frames misst und geeignet visualisiert.
- Das zeitliche Verhalten einiger Konfigurationen ist für verschiedene Lastprofile simuliert. Die Resultate sind visualisiert, interpretiert und kommentiert.

12. Projektmanagement

12.1. Präzisierung der Aufgabenstellung

12.2. Besprechungsprotokolle

Die Besprechungsprotokolle wurden Stichwortartig in einem eigenen Wiki festgehalten. Der Inhalt dieser Protokolle lautet wie folgt:

12.2.1. Kalenderwoche 38: 17.09.2014

- Allgemeine Info, um was geht es:
  - Echtzeit und Hochverfügbarkeit
  - HSR-Ring-Netzwerk-Aufbau
  - Diverse Mechanismen (Welches Frame hat Vortritt?)
- Organisatorisches:
  - Wöchentlicher Rapport via E-Mail vor der Besprechung
  - Wöchentliche Besprechung jeden Donnerstag um 12:00
  - Nächste Besprechung fällt aus

12.2.2. Kalenderwoche 40: 02.10.2014

- Express-Priorität nicht anhand VLAN-Tag festlegen / feststellen
  - Z.B. im EtherType-Feld definieren
- IET spezifiziert, dass bereits gesendetes Fragment ankommen muss (Normales Frame muss bei einem Express-Frame unterbrochen und darf nicht verworfen werden)
  - Kein Fragment $<$ 64 Bytes (min. Size Ethernet Frame)
  - Zu Fragmentierendes Frame in richtige Teile aufsplitten
- In unserem Fall ist das Netzwerk fehlerfrei
- Express-Frames werden nicht fragmentiert
- PHY Layer soll von Ganzem nicht merken
- Jeder Switch in einem Gerät hat Scheduler

12.2.3. Kalenderwoche 41: 09.10.2014

- Anhand Zeitpunkt Frameversand und dessen Grösse ist die Dauer berechenbar
- Zeitberechnung
  - Frames müssen nicht fragmentiert werden wie in der Realität
  - Wie lange hat ein normales Frame, wann ein Express-Frame dazwischenkommt?
  - Express-Frame zu welchem Zeitpunkt senden? (Wann wäre Frame-Fragment versendet worden?)
- Frame mehrmals unterbrechbar
- Queuegrösse ist auch ein Faktor (Per INI-Datei der Simulation definierbar)

12.2.4. Kalenderwoche 42: 16.10.2014

- Traffic-Pattern für Testing (wird noch gesandt)
  - Mix konstanter Bitraten (Erzeugt Frame mit bestimmten Grösse jeden bestimmten Zeitabstand, es können sich auch mehrere überlagern)
  - Random (Background-Traffic)
  - Ab und zu Express-Frame (Alarm)
  - Lastgenerator in Node, für jeden Knoten spezifisch definierbar
- Besprechung Zeitschlitzverfahren
  - Zeitschlitze für High und Low, Express kann immer
  - Intervall wird vereinbart
    - In Zeitschlitz muss man maximales Frame durchbringen
    - Zeitschlitz hat Grün-Phase, in der High Frames (zyklische Messwerte) gesendet werden können, Rest der Zeit für Sendevorgang oder Low

12.2.5. Kalenderwoche 43: 23.10.2014

- Gerät soll sich selber benachrichtigen, sobald neues Frame zu versenden ist
- Lastgenerator Random Prioritäten, z.B. 10% Express, 20% High, 70% Low
- Gibt viel Multicast, wie senden?
- Demnächst Simulation im kleinen Stil mit First Come First Serve zum Laufen bringen

12.2.6. Kalenderwoche 44: 30.10.2014

- Simulation läuft, aber stürzt nach 2 Frames ab
  - Beinhaltet 3 DANH-Geräte, 2 verschiedene senden zeitgleich an anderen
- Scheduler pro Port implementieren, bis jetzt ist Scheduler pro Gerät implementiert
  - Macht wenig Sinn, Switch kann auf freien Ports gleichzeitig versenden
  - Frame kommt an, wo muss es hin? Entsprechendem Scheduler zuweisen
- Express: klein und selten, konstante Bitrate
- High: normaler Verkehr (Monitoring, Netzwerkmanagement, ...)
- Low: Background-Verkehr (z.B. TCP), oft und klein & gross

12.2.7. Kalenderwoche 45: 06.11.2014

- Gedanken zu IET (Zeitberechnung):
  - Sender behält normales Frame, um zu sagen wann Express-Frame versendet werden soll
  - Empfänger verlängert «fragmentiertes Frame» um die Dauer des dazwischen versendeten Express-Frames + IFG
    - Empf. überprüft, ob Sendezeit des Express-Frames zwischen Sende- und Ankunftszeit des normalen Frames ist. Wenn ja, wird das normale Frame verlängert
  - Zweiter Kanal für Express Frame?
    - Wenn Channel busy ist, kann Express-Frame gesendet werden?
    - Zweiter Kanal nur für Express Frames
    - Keine wirkliche Fragmentierung in Simulation vornehmen, Zeitrechnung reicht aus
  - FragCount sagt nicht max. 5 Fragmente, sondern ist ein Modulo-4-Counter
  - Frame abbrechen
    - Nur wenn Rest genug lang ist
    - Express muss Fragmentlänge + IFG abwarten
    - Normal Frame Ankunftszeit um Express + IFG verlängern

12.2.8. Kalenderwoche 46: 13.11.2014

- Anwendungsfall: Substation Automation
  - Schutz von Schaltungen und Leitungen sicherstellen
  - Merging Units (MUs) erfassen Messwerte
    - 7 Messwerte (4x Spannung, 3x Strom) in einem Frame plus Zeitstempel und sonst. Steuerinfos
      - High-Priority Frame von total 160 Bytes
      - Konstant 4000x pro Sekunde pro MU
    - Spontane Einzelmeldungen (Express-Frame, selten und kurz, ca. 100 Bytes)

            - Wird zufällig mit bestimmter Wahrscheinlichkeit versendet, z.B. bei 20% Wahrscheinlichkeit wenn Randomwert von 0 bis 1 zwischen 0 und 0.2 liegt

        - Frames werden via Multicast verteilt (Publisher / Subscriber Modell)
        - Ziel sind Protection Units (PUs), welche anhand der erhaltenen Werte Entscheidungen treffen und doppelt vorhanden sind
        - Max. Anzahl an MUs: 19 Stk., schauen wie es z.B. mit 10 ist
    - Background-Traffic (TCP, nicht wirklich simulierbar, Unicast, meistens mit Gerät ausserhalb Ring)
        - TCP-ähnlichen Traffic in 2 Knoten generieren
            - 200 Frames à 1500 Bytes pro Sekunde von A nach B
            - 200 Frames à 64 Bytes pro Sekunde von B nach A
- Was Intern generiert wird wird gleich behandelt wie das was von Aussen kommt
- Queue-Limit spielt keine allzu grosse Rolle
    - Wie lange ein Frame warten muss oder wie viele Frames verloren gehen sind äquivalente Aussagen
- Man muss am Schluss etwas zur Delay-Charakteristik von Strömen und Express-Frames sagen können
    - Wann generiert und wann Ankunft? Differenz ist Übermittlungszeit
    - Ankunftszeit dort feststellen, wo das Frame vom Netz entfernt wird
    - Nicht besseren, sondern schlechteren Fall anschauen
        - Duplikat anschauen (Was als 2tes ankommt)
        - Best-/Worst-Case spielt nur bei Unicast eine Rolle, bei Multi-/Broadcast macht es wahrscheinlich keinen grossen Unterschied
- Duplikaterkennung bei allen Ports implementieren
    - Kein Port soll dasselbe Frame mehrmals versenden

12.2.9. Kalenderwoche 47: 20.11.2014

- Testfälle für Überprüfung der Implementation
    - Nicht fragmentierbare Frames
    - Versenden eines langen, normalen Frames und 2 darauf folgende Express-Frames

12.2.10. Kalenderwoche 48: 27.11.2014

- Start der Simulation für Belastung nicht sehr interessant, ab dem Zustand nach der «Initialisierung» interessant
  - Ab «Steady State» aufnehmen
- Testing
  - Mit Multicast-Frames testen, weil diese beim Sender gelöscht werden (Bei einem Defekt würde die Dauer dieses Frames etwa der Dauer eines normalen Frames entsprechen
  - Express-Frames mit Multicast versenden
  - Mit 17 MUs (siehe Kapitel 12.2.8 auf Seite 130), die permanent Messwerte via Multicast versenden sollte so gut wie kein Platz mehr auf dem Ethernet frei sein.
  - Intervall, in dem die Messwerte versendet werden, variiert von MU zu MU (Phase beginnt nicht überall zur exakt selben Zeit)
    - Beim Intervall eine Zufallszeit dazu- oder abzählen, sodass der Intervall nicht konstant ist
    - Delayunterschied zu Fall, an dem die MUs gleichzeitig und mit exakt selbem Intervall senden untersuchen
  - Fall: 14, 15 MUs
    - Einfluss TCP (High-Frame wird angestaut, wenn gerade ein Low-Frame gesendet wird)
    - Was wenn TCP Lücken zwischen High-Frames schamlos nutzt?
      - Zufluss TCP limitieren?
  - 100Mbps für die Verbindung verwenden
    - Untersuchen wie viele Frames pro Sekunde noch übertragbar sind (bevor sich die Queues nicht mehr leeren). Wie nahe kann man an diese Grenze gehen?
    - Frame mit Messwerten beträgt 160 Bytes: ca. 69444 solcher Frames über 100Mbps pro Sekunde
      $\frac{10^8}{(160+Präamble+IFG)*8} = \frac{10^8}{180*8} = 69444.\overline{4}$

- Visualisierung der Resultate
  - Histogramm (Wie viele haben Transmissiontime x?)
    - Klassenbreite (Transmissiontime auf ms/us genau)
  - Gesamtkapazität < Summe von allem MU- plus TCP-Traffic
  - Verifizierung Verhalten mit Frameverlauf in Simulation
- Anleitung
  - Ab CD wie starten
    - Möglich: Exe-Datei starten und Config editieren (VM auf CD?)
  - Wie Simulation laufen lassen
  - Wie Konfigurieren (in XML- und INI-Datei)
  - Wie bestimmter Fall simulieren

12.2.11. Kalenderwoche 49: 04.12.2014

- Was bedeuten in der Simulation die verschiedenen Farben der Verbindungen?
  - In Anleitung festhalten
- Interner Delay (im Knoten) von 6us implementieren
  - Zeit bis der Teil des Frames gelesen werde, um entscheiden zu können, was damit gemacht wird (Cut-Through-Switching)
- Für die Verbindung zwischen den Endknoten 20m 100Mbps Kabel verwenden
  - Delay von 5ns pro 1m Kabel
  - Delay 20m Kabel: 100ns
- Zuflusslimitierung bei TCP-ähnlichem Traffic auf z.B. 1Mbit/s setzen
- Notieren, wo Aufgabenstellung in welchem Code erfüllt wurde und wie man zu diesem Modul gekommen ist
- Testfälle
  - Nur ein Frame auf leeren Ring senden -> schnellster Übertragungswert
  - Frame bei t1, Express bei t2 (bisschen später als t1) versenden -> Wie sieht es mit der Preemption aus?
    - Was wenn das Frame nicht fragmentierbar ist?
    - Was wenn es fragmentierbar ist?
    - Kann ein langes Frame von mehreren Express-Frames unterbrochen werden?

12.2.12. Kalenderwoche 50: 11.12.2014

- Zuflusslimitierung nur für Frames der Priorität Low
  - Express-Frames gibt es «by design» wenige
  - Zweck der Limitierung: Nicht Überlebenswichtiges limitieren
- Steady State
  - Werte in leerem Ring uninteressant
  - Start und Stop uninteressant
  - Fenstergrösse, in der die Werte aufgezeichnet werden, definieren, paar 10tel Sekunden
  - Kann in omnetpp.ini definiert werden
- Dies war die letzte Besprechung

13. Anleitung zur Simulationsumgebung und -Durchführung

Hierbei handelt es sich um eine grobe Anleitung, wie man die Simulationsumgebung mit den bei gelieferten Daten ausführen, konfigurieren und dessen Resultate analysieren kann. Der genaue Umgang mit OMNeT++ ist in dessen offiziellen User Manual beschrieben [11].

13.1. Starten der virtuellen Maschine

Auf dem mitgelieferten USB-Stick befindet sich eine VDI-Datei. Diese Datei beinhaltet eine virtuelle Maschine (VM), auf der das 32Bit Linux-Betriebssystem Debian inklusive der OMNeT++-Umgebung und einer englischen Anleitung eingerichtet ist. Eine deutsche Anleitung wird in diesem Kapitel aufgeführt.

Damit diese virtuelle Maschine ausgeführt werden kann wird die Software VirtualBox benötigt. Diese kann unter `www.virtualbox.org` heruntergeladen werden.

Um die VDI-Datei einzubinden erstellt man in VirtualBox eine neue virtuelle Maschine und navigiert durch den Assistenten (Typ: «Linux», Version: «Debian (32 bit)»). Die virtuelle Maschine wurde mit einem zugewiesenen Arbeitsspeicher von 512MB getestet (es wird empfohlen 1GB oder mehr als Arbeitsspeicher auszuwählen). Anstelle einer neuen Festplatte zu erstellen wird nun die bestehende VDI-Datei eingebunden. Wurde die virtuelle Maschine erstellt, kann sie gestartet werden.

Für den Betrieb der virtuellen Maschine sind keine Login-Daten notwendig. Es wird automatisch eine Desktop-Umgebung gestartet, die in etwa folgendermassen aussieht (Desktop-Icons können variieren):

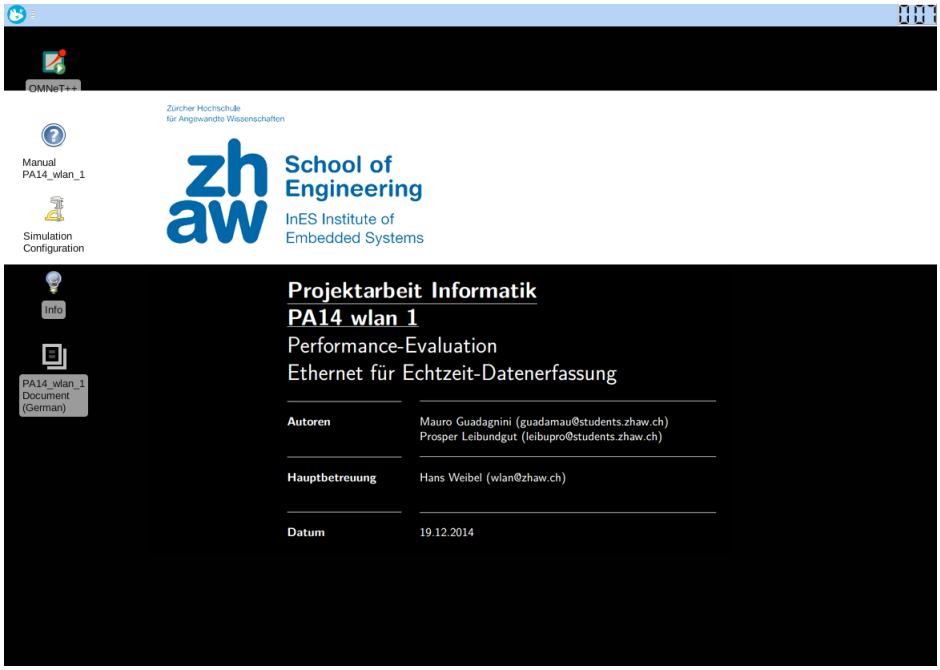


Abbildung 13.1.: Screenshot der gerade gestarteten virtuellen Maschine

Auf dem Desktop und dem Startmenü (links oben im Screenshot) werden alle benötigten Verknüpfungen zur Verfügung gestellt. Im geöffneten Startmenü gibt es rechts unten die Möglichkeit, die virtuelle Maschine herunter zu fahren und des Weiteren auch die Möglichkeit, nach Applikationen / Verknüpfungen zu suchen.

Sollte trotzdem ein Login benötigt werden, ist dieser in der VM unter dem Punkt «Info» zu finden.

13.2. Simulation konfigurieren

Für die Konfiguration einer Simulation kann man OMNeT++ oder einen normalen Texteditor verwenden. In dieser Anleitung werden die Dateien mit OMNeT++ editiert. OMNeT++ kann man über die Desktopverknüpfung starten.

Jede Simulation ist in der Datei «omnetpp.ini» (in OMNeT++ in Projekt «PA14_wlan_1» im Ordner «simulations») aufgeführt. Wer die Dateien mittels Texteditor editieren möchte, findet die Dateien über die Desktopverknüpfung «Simulation Configuration».

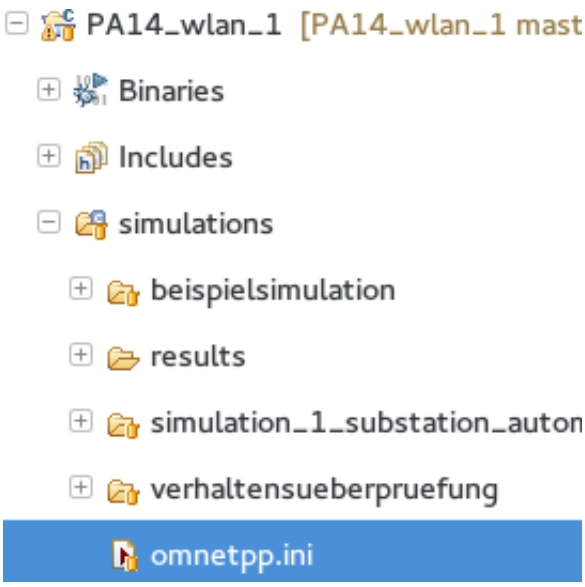


Abbildung 13.2.: Standort der Datei «omnetpp.ini» in OMNeT++

```
1  [Config Beispielsimulation]
2  network = beispielsimulation.SubstationHSR
3  **.cpu.xmlparam = xmldoc("beispielsimulation/config.xml")
4  **.endNodeSwitch.schedulerMode = "FCFS"
5  **.PU1.cpu.multicastListener = 1
6  **.PU2.cpu.multicastListener = 1
7  **.vector-recording-intervals = 0.02..0.5
8  # **.endNodeSwitch.framebyteLimit = 12500
9  # **.endNodeSwitch.timeslotPhaseSize = 10000
```

Listing 13.1: omnetpp.ini - Beispiel eines Simulationseintrags

Dieser Eintrag zeigt, welche Dateien zur Konfiguration editiert werden können. Hier können zudem weitere Eigenschaften der Simulation definiert werden (wie z.B. der anzuwendende Mechanismus), die am Anfang der Datei «omnetpp.ini» vermerkt sind. Bei den Zeilen mit einem #-Symbol am Anfang wird die dort gesetzte Konfiguration nicht angewandt (derzeit als Kommentar gesetzt).

13.2.1. Netzwerkaufbau

Für den Netzwerkaufbau (Zeile mit «network») wird die Datei «beispielsimulation/SubstationHSR.ned» verwendet. In dieser Datei sind die einzelnen Knoten definiert und miteinander Verbunden.

```
1  MU1: HsrEndNode {
2    macAddress = "00-15-12-14-88-01";
3    @display("i=devices/mu;p=759,53");
4  }
```

Listing 13.2: SubstationHSR.ned - Beispiel eines Knoteneintrags

Beim Editieren eines solchen Knoten reicht es, wenn man den Namen (hier «MU1»), die MAC-Adresse und die Koordinaten des Knotens in der Netzwerkdarstellung («p=x,y») editiert. Erstellt man einen neuen Knoten, sind das auch die einzigen Angaben, die zu denen anderer Knoten abweichen müssen.

Die MAC-Adresse, die hier definiert wird, wird später in der Konfiguration des Lastgenerators verwendet.

Mehr über NED-Dateien ist im OMNeT++ User Manual aufzufinden [11].

13.2.2. Lastgenerator

Die XML-Datei, die zum Konfigurieren des Lastgenerators verwendet wird, ist im Beispiel des Simulationseintrags (erstes Listing am Anfang des Kapitels 13.2 auf der vorherigen Seite) aufgeführt. Hier wird die Datei «beispielsimulation/config.xml» verwendet.

Wie die XML-Datei in etwa aussieht ist in Kapitel 3.1.4 auf Seite 27 aufzufinden. Ein Eintrag (source-Tag) beschreibt die Lastgenerierung eines bestimmten Frametyps für einen bestimmten Knoten (mittels MAC-Adresse). Für einen Knoten können mehrere solcher Einträge existieren, sodass er z.B. ständig High- und Low-Frames generieren könnte. Es können auch Knoten existieren, die keinen Eintrag in der XML-Datei haben.

In der XML-Datei ist des Weiteren aufgeführt, für was welcher Inhalt editiert werden muss.

13.3. Simulation starten

Falls bereits eine neu konfigurierte Simulation zuvor schon gestartet wurde, muss diese geschlossen und neu geöffnet werden, damit die aktuelle Konfiguration übernommen wird.

Sofern OMNeT++ noch nicht geöffnet wurde, muss es jetzt über die Desktopverknüpfung gestartet werden. Ist die Applikation geöffnet, klickt man oben auf «Run» -> «Run History» -> «PA14_wlan_1» oder klickt auf den im folgenden Bild rot umrandeten Pfeil und dann auf «PA14_wlan_1».

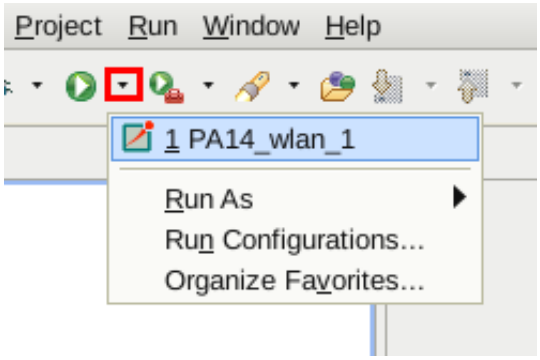


Abbildung 13.3.: Starten der Simulation in OMNeT++

Ist die Simulationsumgebung gestartet, kann man eine Konfiguration/Simulation auswählen und klickt auf «OK». Zur Auswahl stehen alle in dieser Arbeit durchgeführten Simulationen (Der Konfigurationsname ist bei der jeweiligen Simulation in der ersten Zeile aufgeführt) oder die vorhin konfigurierte Beispielsimulation.

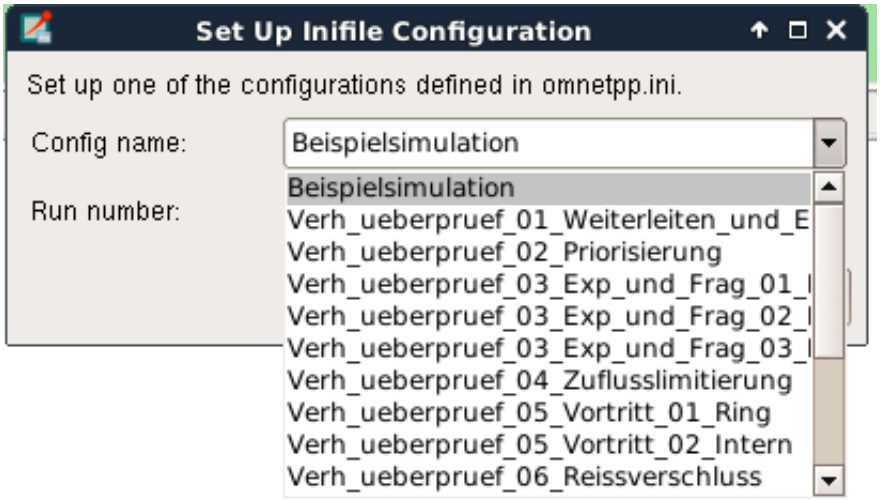


Abbildung 13.4.: Konfigurations-/Simulationsauswahl in OMNeT++

Wenn man «Beispielsimulation» auswählt, sieht nach dem Klick auf «OK» die Umgebung in etwa wie folgt aus:

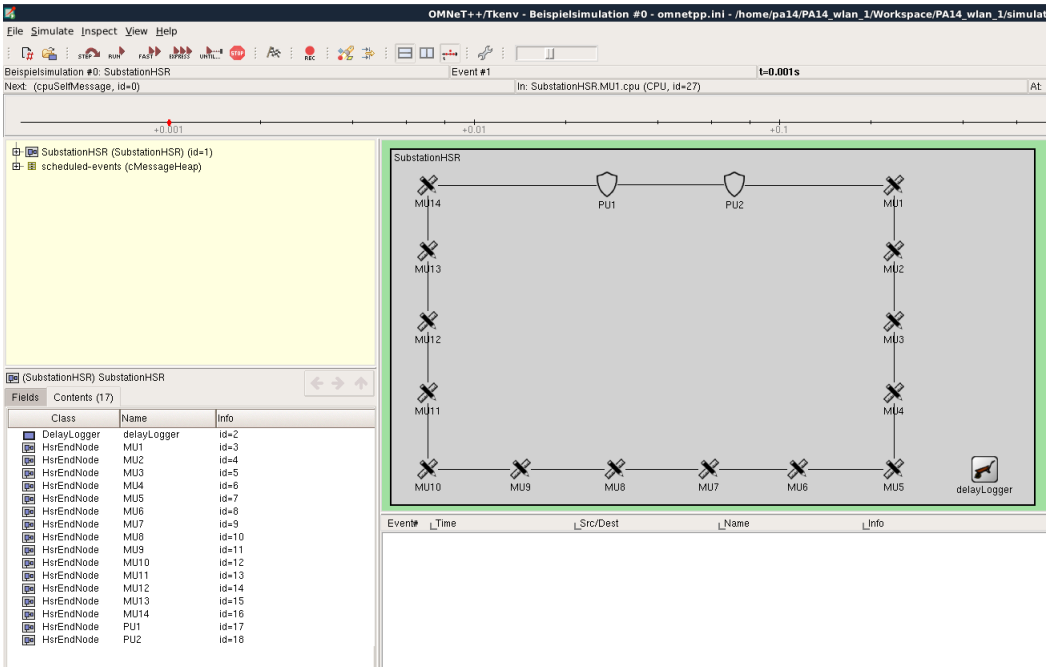


Abbildung 13.5.: Simulationsumgebung in OMNeT++

Die Fensterbereiche kann man sich nach Belieben zu recht ziehen. Im unteren Bereich ist der Nachrichtenverlauf aufgeführt.

Folgende Buttons links oben in der Simulationsumgebung können unter Anderem für die Simulation verwendet werden:

- **Step:** Tätigt ein einzelnes Event, muss kein Frameversand sein.
- **Run:** Lässt die Simulation in einer verfolgbaren Geschwindigkeit laufen. Es werden alle Animationen dargestellt. Die Geschwindigkeit lässt sich mittels Schiebefeld oben rechts neben dem Schraubenschlüssel-Button genauer definieren.
- **Fast:** Die Simulation läuft viel schneller, der Frameversand wird nicht mehr animiert, jedoch ist anhand der Verbindungsfarben noch ersichtlich auf welcher Leitung gerade gesendet wird (Gelbe Verbindung).
- **Express:** Die Simulation läuft auf der höchsten Geschwindigkeit und zeigt keine Animationen mehr. Der Nachrichtenverlauf wird zudem nicht mehr nachgeführt.
- **Until:** Führt die Simulation im Express-Modus bis zum angegebenen Zeitpunkt aus.
- **Stop:** Haltet die Simulation zum derzeitigen Zeitpunkt an. Man kann ab diesem Punkt die Simulation auch weiterführen lassen.
- **Finish:** Schliesst die Simulation ab. Klickt man danach nochmals auf «Step», «Run», «Fast», «Express» oder «Until» kann man die Simulation nochmals von Vorne ausführen.

Mehr zur Simulationsumgebung ist im OMNeT++ User Manual im Kapitel «The Tkenv Graphical Runtime Environment» aufgeführt [11].

Ist die Simulation beendet (ein Fenster mit dem Inhalt «no more events» erscheint), wird automatisch der sogenannte «Finish»-Befehl ausgeführt, welcher die aufgezeichneten Resultate abspeichert. Interessieren nur die Werte bis zum jetzigen Zeitpunkt, kann man die Simulation mit einem Klick auf «Finish» abschliessen. Nun kann man die Resultate der Simulation ansehen.

13.4. Resultate ansehen

Die Resultate können in der OMNeT++ Applikation (nicht die Simulationsumgebung) angesehen werden. Die Resultate der jeweiligen Simulation werden als ANF-Datei mit dem Konfigurationsnamen als Dateinamen im selben Ordner wie dessen XML-Datei (siehe Kapitel 13.2.2 auf Seite 138) aufgelistet. Die Rohdaten der Simulation befinden sich in OMNeT++ in Projekt «PA14_wlan_1» im Ordner «simulations/results», haben den Namen der Konfiguration und unterschiedliche Dateiendungen.

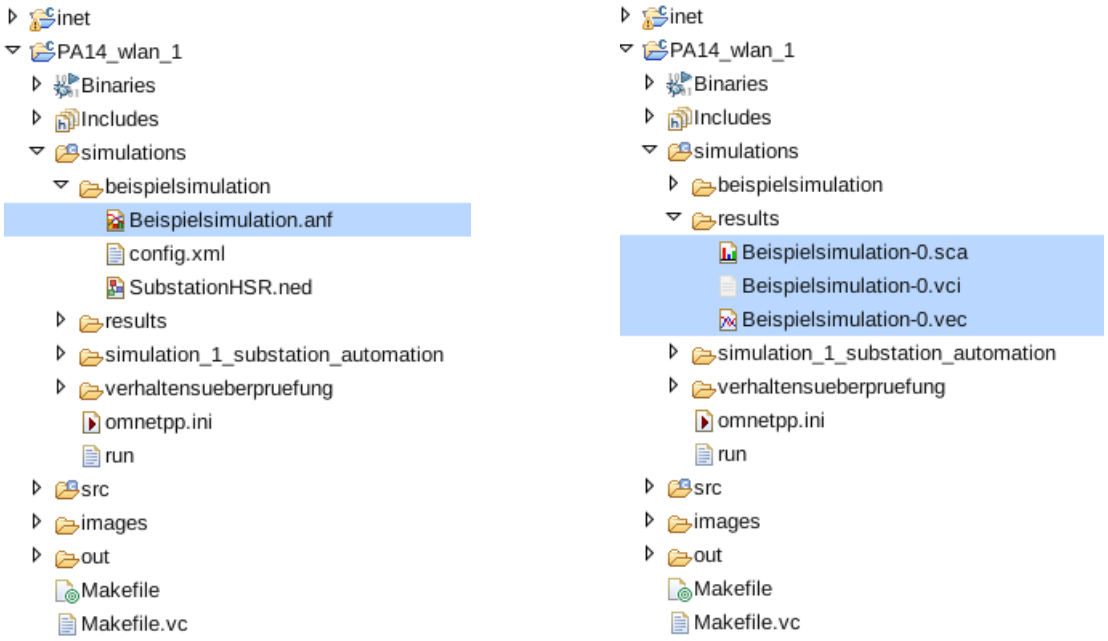


Abbildung 13.6.: Standorte der Resultate (ANF-Datei und Rohdaten) der Simulation «Beispielsimulation»

Nun kann man mit einem Doppelklick auf die entsprechende ANF-Datei klicken, um die Resultate zu öffnen. Unterhalb des mittleren Fensters kann man auf «Browse Data» klicken, um die Resultate genau analysieren zu können. In dieser Arbeit wurden spezifisch folgende Daten aufgezeichnet:

Vectors: Ausgang: QueueSize nach Priorität und Herkunft (From Ring / (Internal))
Beschreibt, zu welchem Zeitpunkt wie viele Frames bei einem Ausgang eines Knotens in einer Queue eingereiht sind.

Vectors: Ausgang: endNodeSwitch Preempted Frames
Beschreibt, zu welchem Zeitpunkt wie viele Express-Frames bei einem Ausgang eines Knotens andere Frames unterbrochen haben.

Histograms: Express, High & Low Prio Transmission Time
Zeigt, wie viele Frames welche Übertragungsdauer hatten.

Die genauere Bedeutung der Resultate ist in Kapitel 4 auf Seite 54 aufgeführt. Diese Resultate können einzeln oder kombiniert (mehrere mit Ctrl-/Strg-Taste markieren) per Rechtsklick -> «Plot» ausgegeben werden.

Der genauere Umgang bei der Auswertung der Resultate kann im OMNeT++ User Manual im Kapitel «Analyzing the Results» betrachtet werden [11].

14. USB-Stick & SourceCode

Dieser Arbeit wird ein USB-Stick mit dem SourceCode und folgenden Eigenschaften mitgegeben:

Marke & Modell	SanDisk Extreme USB 3.0 Flash Drive
Kapazität	16 GB
Name	PA14_wlan_1
Dateisystem	NTFS

Tabelle 14.1.: USB-Stick Eigenschaften

Auf dem USB-Stick befinden sich folgende Dateien und Verzeichnisse:

Documentation	Beinhaltet diese Dokumentation als PDF-Datei und als Quellcode im LyX-Format
PA14_wlan_1.vdi	Virtuelle Maschine (siehe Kapitel 13.1 auf Seite 135)
SourceCode_Simulation	Beinhaltet den Source-Code dieses Projekts als C++-Code und OMNeT++-Files Folgende Ordner befinden sich innerhalb dieses Verzeichnisses: • **omnetpp-4.5**: OMNeT++Version 4.5 • **Workspace:** Workspace zur Integration in OMNeT++ – **inet**: INET Framework von OMNeT++ – **PA14_wlan_1**: Programmierte Projektarbeit von Mauro Guadagnini und Prosper Leibundgut

Tabelle 14.2.: USB-Stick Inhalt

15. Codeausschnitte

15.1. Switch: Zuteilung Frame von CPU an beide Ports nach Aussen

```
1   ...
2   void EndNodeSwitch::handleMessage( cMessage* msg ) {
3   ...
4    /* Schedulers */
5       Scheduler* schedGateAOut = HsrSwitch::getSchedGateAOut();
6       Scheduler* schedGateBOut = HsrSwitch::getSchedGateBOut();
7       Scheduler* schedGateCpuOut = HsrSwitch::getSchedGateCpuOut();
8   ...
9       /* Gates */
10      cGate* gateAIn = HsrSwitch::getGateAIn();
11      cGate* gateBIn = HsrSwitch::getGateBIn();
12      cGate* gateCpuIn = HsrSwitch::getGateCpuIn();
13
14      cGate* gateAInExp = HsrSwitch::getGateAInExp();
15      cGate* gateBInExp = HsrSwitch::getGateBInExp();
16      cGate* gateCpuInExp = HsrSwitch::getGateCpuInExp();
17  ...
18      /* Make a clone of the frame and take ownership.
19       * Then we have to downcast the Message as an ethernet frame. */
20      cMessage* switchesMsg = msg;
21      this->take( switchesMsg );
22  ...
23      /* Arrival Gate */
24      cGate* arrivalGate = switchesMsg->getArrivalGate();
25
26      /* Switch mac address. */
27      MACAddress switchMacAddress = *( HsrSwitch::getMacAddress() );
28
29      EthernetIIFrame* ethernetFrame = check_and_cast<EthernetIIFrame*>( switchesMsg );
30
31      /* Source and destination mac addresses */
32      MACAddress frameDestination = ethernetFrame->getDest();
33      MACAddress frameSource = ethernetFrame->getSrc();
34
35      EthernetIIFrame* ethTag = NULL;
36      vlanMessage* vlanTag = NULL;
37      hsrMessage* hsrTag = NULL;
38      dataMessage* messageData = NULL;
39
40      EthernetIIFrame* frameToDeliver = NULL;
41      EthernetIIFrame* frameToDeliverClone = NULL;
42
43      framePriority frameprio = LOW;
44
45      MessagePacker::decapsulateMessage( &ethernetFrame, &vlanTag, &hsrTag, &messageData );
46
47      /* After decapsulating the whole ethernet frame becomes a ethernet tag
48       * (only the header of an ethernet frame)
49       * ethTag variable just there for a better understanding. */
50      ethTag = ethernetFrame;
51
52      /* determine package prio and set enum */
53      if( vlanTag != NULL )
54      {
55          if( vlanTag->getUser_priority() == EXPRESS )
56          {
57              frameprio = EXPRESS;
58          }
59          else if( vlanTag->getUser_priority() == HIGH )
60          {
61              frameprio = HIGH;
62          }
63      }
64  ...
65          if( arrivalGate == gateCpuIn || arrivalGate == gateCpuInExp )
66          {
67              frameToDeliver = MessagePacker::generateEthMessage( ethTag, vlanTag, hsrTag, messageData );
68              frameToDeliverClone = frameToDeliver->dup();
69
70              switch( frameprio )
```

```
71              {
72                  case EXPRESS:
73                  {
74                      schedGateAOut->enqueueMessage( frameToDeliver, EXPRESS_INTERNAL );
75                      schedGateBOut->enqueueMessage( frameToDeliverClone, EXPRESS_INTERNAL );
76                      break;
77                  }
78                  case HIGH:
79                  {
80                      schedGateAOut->enqueueMessage( frameToDeliver, HIGH_INTERNAL );
81                      schedGateBOut->enqueueMessage( frameToDeliverClone, HIGH_INTERNAL );
82                      break;
83                  }
84                  default:
85                  {
86                      schedGateAOut->enqueueMessage( frameToDeliver, LOW_INTERNAL );
87                      schedGateBOut->enqueueMessage( frameToDeliverClone, LOW_INTERNAL );
88                      break;
89                  }
90              }
91              scheduleProcessQueues('A');
92              scheduleProcessQueues('B');
93          }
94  ...
95  }
96  ...
```

Listing 15.1: switch/EndNodeSwitch.cc - Zuteiltung Frame von CPU an beide Ports nach Aussen

15.2. Scheduler: Zeitberechnung und Versand IET

```
1   ...
2   void Scheduler::processOneQueue( cQueue* currentQueue, queueName currentQueueName )
3   {
4       if( !currentQueue->isEmpty() )
5       {
6     ...
7           /* check if the transmit state of the nic is idle. */
8           if( ( transmitLock == 0 && transmitLockExp == 0 &&
9                 timeslotIsValid( currentQueueName ) && enoughBytesAvailableToSend )
10              || schedID == 'C' )
11          {
12              cMessage* msg = check_and_cast<cMessage*>( currentQueue->pop() );
13
14              if( !(currentQueueName == EXPRESS_RING) && !(currentQueueName == EXPRESS_INTERNAL) )
15              {
16                  transmitLock = 1;
17                  /*
18                   * Only attach Frames that are not from the "Express-Ring"- or "Express-Internal"-queue
19                   */
20                  sendingStatus->attachFrame( msg );
21                  sendMessage( msg, schedOutGate );
22      ...
23              }
24              /*
25               * Express-Frames
26               */
27              else
28              {
29                  transmitLockExp = 1;
30                  sendMessage( msg, schedOutGateExp );
31      ...
32              }
33          }
34          else
35          {
36              if( ( currentQueueName == EXPRESS_RING || currentQueueName == EXPRESS_INTERNAL ) &&
37                      transmitLock == 1 &&
38                      transmitLockExp == 0 &&
39                      enoughBytesAvailableToSend )
40              {
41                  /*
42                   * EXPRESS HANDLING (PREEMPTION)
43                   */
44                  expSendTime = getExpressSendTime();
45
46                  /* return SIMTIME_ZERO means current frame is not fragmentable */
47                  if( expSendTime != SIMTIME_ZERO )
48                  {
49                      transmitLockExp = 1;
50                      sendExpressFrame( currentQueue, currentQueueName, expSendTime );
51                  }
52              }
53              else
54              {
55          ...
56              }
57          }
58      }
59  }
60
61  simtime_t Scheduler::getExpressSendTime( void )
62  {
63      /* Algorithm from documentation PA14_wlan_1 */
64      int64_t allBytesOfSendingFrame = sendingStatus->getMessageSize();
65      simtime_t calcExpSendTime;
66
67      if( allBytesOfSendingFrame >= 128 )
68      {
69
70          simtime_t simTimeNow = simTime();
71          simtime_t currentPreemptionDelay = sendingStatus->getPreemptionDelay();
72
73          simtime_t sendTimeOfFrame = sendingStatus->getSendingTime();
74
75          int64_t bytesAlreadySent = ( int64_t )floor(
76                      ( ( simTimeNow.dbl() - sendTimeOfFrame.dbl() ) * datarate ) / 8.0
77      + ( (currentPreemptionDelay.dbl() * datarate) / 8.0 )
78                  );
79          int64_t bytesAlreadySentData = bytesAlreadySent - 12 - (INTERFRAME_GAP_BITS / 8);
80          if( bytesAlreadySentData < 0 )
81          {
82              bytesAlreadySentData = 0;
83          }
84
85
86          int64_t bytesNotYetSent = allBytesOfSendingFrame - bytesAlreadySent + 12;
87
88
89          if( ( bytesAlreadySent >= ( 72 + ( INTERFRAME_GAP_BITS / 8 ) ) &&
90              bytesAlreadySentData % 8 == 0 &&
```

```
91              bytesNotYetSent >= 64 ) || nextUpAnotherExp == 1 )
92          {
93              /* can send express frame now. */
94              calcExpSendTime = simTimeNow;
95          }
96          else if( bytesAlreadySentData % 8 != 0 &&
97                   bytesNotYetSent >= 64 )
98          {
99              if( bytesAlreadySent >= ( 72 + ( INTERFRAME_GAP_BITS / 8 ) ) )
100             {
101                 calcExpSendTime = simTimeNow + ( ( ( 8 - ( bytesAlreadySentData % 8 ) + (
                        INTERFRAME_GAP_BITS / 8 ) ) * 8 ) / datarate );
102             }
103             else
104             {
105                 calcExpSendTime = simTimeNow + ( ( ( 72 + ( INTERFRAME_GAP_BITS / 8 ) - bytesAlreadySent +
                        ( 8 - ( bytesAlreadySentData % 8 ) ) ) * 8 ) / datarate );
106             }
107         }
108         else if( bytesAlreadySentData % 8 == 0 &&
109                  bytesNotYetSent >= 64 )
110         {
111             if( bytesAlreadySent >= ( 72 + ( INTERFRAME_GAP_BITS / 8 ) ) )
112             {
113                 calcExpSendTime = simTimeNow + ( ( ( ( INTERFRAME_GAP_BITS / 8 ) ) * 8 ) / datarate );
114             }
115             else
116             {
117                 calcExpSendTime = simTimeNow + ( ( ( 72 + ( INTERFRAME_GAP_BITS / 8 ) - bytesAlreadySent )
                        * 8 ) / datarate );
118             }
119         }
120         else
121         {
122             calcExpSendTime = SIMTIME_ZERO;
123         }
124     }
125     else
126     {
127         calcExpSendTime = SIMTIME_ZERO;
128     }
129
130     return calcExpSendTime;
131 }
132
133 void Scheduler::sendExpressFrame( cQueue* currentQueue, queueName currentQueueName, simtime_t expSendTime )
134 {
135     cMessage* msgExp = check_and_cast<cMessage*>( currentQueue->pop() );
136     EthernetIIFrame* msgethExp = check_and_cast<EthernetIIFrame*>( msgExp->dup() );
137     parentSwitch->sendDelayed(msgExp, (expSendTime-simTime()), schedOutGateExp);
138
139     ...
140
141     EthernetIIFrame* fcsMsg = new EthernetIIFrame();
142     fcsMsg->setBitLength( 4 * 8 );
143     EthernetIIFrame* ifgMsg = new EthernetIIFrame();
144     ifgMsg->setBitLength( INTERFRAME_GAP_BITS );
145     EthernetIIFrame* preambleMsg = new EthernetIIFrame();
146     preambleMsg->setBitLength( 8 * 8 );
147
148     cMessage* currMsg = sendingStatus->getMessage();
149
150     simtime_t timeMfcsFirstFragment = schedNicExp->getTransmissionChannel()->calculateDuration( fcsMsg );
151     simtime_t timeIfg = schedNicExp->getTransmissionChannel()->calculateDuration( ifgMsg );
152     simtime_t timePreamble = schedNicExp->getTransmissionChannel()->calculateDuration( preambleMsg );
153     simtime_t timeMsgExp = schedNicExp->getTransmissionChannel()->calculateDuration( msgExp );
154     simtime_t timeMsgDelayWhenFragmented = timeMfcsFirstFragment + timeIfg + timePreamble + timeMsgExp +
            timeIfg + timePreamble;
155
156     addPreemptionDelay( currMsg, timeMsgDelayWhenFragmented );
157
158     ...
159 }
160 ...
```

Listing 15.2: switch/Scheduler.cc - Zeitberechnung und Versand IET

15.3. Scheduler: Zuflusslimitierung

```
...
unsigned char Scheduler::containerHasEnoughBytes( cMessage* msg )
{
    unsigned char retVal = 0x00;
    /*
     * The tokenizer has only impact on low priority frames.
     * This method always returns true when the priority is not low.
     * */
    EthernetIIFrame* ethTag = NULL;
    vlanMessage* vlanTag = NULL;
    hsrMessage* hsrTag = NULL;
    dataMessage* ethPayload = NULL;

    ethTag = check_and_cast<EthernetIIFrame*>( msg->dup() );

    MessagePacker::decapsulateMessage( &ethTag, &vlanTag, &hsrTag, &ethPayload );

    framePriority framePrio = static_cast<framePriority>( vlanTag->getUser_priority() );

...

    if( framePrio == LOW )
    {
        /* Time elapsed since last sent low priority frame. */
        simtime_t timeElapsed = simTime() - lowSendTime;
        double creditBytes = timeElapsed.dbl() * ( framebyteLimit / 8.0 );
        int64_t creditBytesFloored = ( int64_t )floor( creditBytes );

        /* in case of an overflow the result of ( framebytecontainer + creditBytes ) is negative */
        if( ( framebytecontainer + creditBytesFloored ) > framebyteLimit || ( framebytecontainer +
             creditBytesFloored ) < -1 )
        {
            framebytecontainer = framebyteLimit;
        }
        else
        {
            framebytecontainer = framebytecontainer + creditBytesFloored;
        }


        cGate* arrivalGate = msg->getArrivalGate();
        if( arrivalGate == parentSwitch->getGateCpuIn() || arrivalGate == parentSwitch->getGateCpuInExp() )
        {
            EthernetIIFrame* checkmsg = check_and_cast<EthernetIIFrame*>( msg->dup() );
            int64_t lengthFrame = checkmsg->getByteLength();
            delete checkmsg;

      /* value of framebytelimit is -1 if the tokenizer is disabled -> return true */
            if( framebyteLimit == -1 )
            {
                retVal = 0x01;
            }
            else if( framebytecontainer - lengthFrame >= 0 )
            {
                /* in this case the low priority frame can be sent at this time. */
                retVal = 0x01;
                lowSendTime = simTime() - (creditBytes-creditBytesFloored)/framebyteLimit;
            }
        }
        else
        {
            retVal = 0x01;
        }
    }
    else
    {
        retVal = 0x01;
    }

    MessagePacker::deleteMessage( &ethTag, &vlanTag, &hsrTag, &ethPayload );

    return retVal;
}

...

void Scheduler::subtractFromByteContainer( cMessage* msg )
{
    /*
     * The tokenizer has only impact on low priority frames.
     * This method does nothing when the priority is not low.
     * */
    EthernetIIFrame* ethTag = NULL;
    vlanMessage* vlanTag = NULL;
    hsrMessage* hsrTag = NULL;
    dataMessage* ethPayload = NULL;

    ethTag = check_and_cast<EthernetIIFrame*>( msg->dup() );

    MessagePacker::decapsulateMessage( &ethTag, &vlanTag, &hsrTag, &ethPayload );
```

```
90
91      framePriority framePrio = static_cast<framePriority>( vlanTag->getUser_priority() );
92
93  ...
94
95      MessagePacker::deleteMessage( &ethTag, &vlanTag, &hsrTag, &ethPayload );
96
97      if( framePrio == LOW )
98      {
99          /*
100          * Only subtract newly generated frames (from the cpu)
101          */
102         cGate* arrivalGate = msg->getArrivalGate();
103         if( arrivalGate == parentSwitch->getGateCpuIn() || arrivalGate == parentSwitch->getGateCpuInExp() )
104         {
105             EthernetIIFrame* checkmsg = check_and_cast<EthernetIIFrame*>( msg->dup() );
106             int64_t lengthFrame = checkmsg->getByteLength();
107             delete checkmsg;
108
109             if( framebytecontainer - lengthFrame >= 0 )
110             {
111                 framebytecontainer = framebytecontainer - lengthFrame;
112             }
113             else if ( framebyteLimit != -1 )
114             {
115                 throw cRuntimeError( "[ Scheduler ]: Error in traffic limitation" );
116                 parentSwitch->endSimulation();
117             }
118         }
119     }
120 }
121 ...
```

Listing 15.3: switch/Scheduler.cc - Zuflusslimitierung

15.4. Scheduler: Mechanismen zur Queueverwaltung

```
1   ...
2   void Scheduler::processQueues( void )
3   {
4       queueName* currentSortOrder;
5
6
7       switch( schedmode )
8       {
9           /*
10           * FCFS: we have no distinction between Ring- and Internal frames.
11           * See the function above "enqueueMessage". Frames ordered to the ring queues
12           * are redirected to the internal queues.
13           * Due to the above description, the behaviour FCFS and RING_FIRST have not to
14           * be distinguished here.
15           *
16           * The Ring queues of each level are processed first,
17           * which is part of the RING_FIRST policy.
18           *
19           * In case of FCFS-scheduling-mode the ring queues are just always empty,
20           * see enqueueMessage-function.
21           *
22           * The processing order of the queues is as follows:
23           *
24           * EXPRESS_RING, EXPRESS_INTERNAL, HIGH_RING, HIGH_INTERNAL, LOW_RING, LOW_INTERNAL
25           * */
26          case FCFS:
27          case RING_FIRST:
28          {
29              currentSortOrder = ringFirstSortOrder;
30              loopQueues( currentSortOrder );
31              break;
32          }
33
34          /*
35           * The Internal queues of each level are processed first,
36           * which is part of the INTERNAL_FIRST policy.
37           */
38          case INTERNAL_FIRST:
39          {
40              currentSortOrder = internalFirstSortOrder;
41              loopQueues( currentSortOrder );
42              break;
43          }
44
45          /*
46           * ZIPPER-Policy
47           * Alternates the queues between ring and internal.
48           *
49           * Processing order of the queues:
50           *
51           * [2n] even
52           * EXPRESS_RING, EXPRESS_INTERNAL, HIGH_RING, HIGH_INTERNAL, LOW_RING, LOW_INTERNAL
53           *
54           * [2n-1] odd
55           * EXPRESS_INTERNAL, EXPRESS_RING, HIGH_INTERNAL, HIGH_RING, LOW_INTERNAL, LOW_RING
56           * */
57          case ZIPPER:
58          {
59              switch( curQueueState )
60              {
61                  case RING:
62                  {
63                      currentSortOrder = ringFirstSortOrder;
64                      loopQueues( currentSortOrder );
65                      curQueueState = INTERNAL;
66                      break;
67                  }
68                  case INTERNAL:
69                  {
70                      currentSortOrder = internalFirstSortOrder;
71                      loopQueues( currentSortOrder );
72                      curQueueState = RING;
73                      break;
74                  }
75                  default:
76                  {
77                      break;
78                  }
79              }
80              /* break to case ZIPPER */
81              break;
82          }
83  ...
84      }
85  }
86
87  ...
88
89  void Scheduler::loopQueues( queueName* currentSortOrder )
90  {
```

```
91      cQueue* currentQueue = NULL;
92      queueName currentQueueName;
93
94      for( unsigned char i = 0; i < QUEUES_COUNT; i++ )
95      {
96          currentQueueName = currentSortOrder[ i ];
97          currentQueue = check_and_cast<cQueue*>( queues->get( currentQueueName ) );
98
99          if( currentQueue != NULL )
100         {
101             processOneQueue( currentQueue, currentQueueName );
102         }
103         else
104         {
105             throw cRuntimeError( " [ PANIC ] : current queue to process is NULL! exiting ... \n" );
106         }
107     }
108 }
109 ...
```

Listing 15.4: switch/Scheduler.cc - Mechanismen zur Queueverwaltung

15.5. Scheduler: Zeitschlitzverfahren

```
1  ...
2  unsigned char Scheduler::timeslotIsValid( queueName currentQueueName )
3  {
4      simtime_t now = simTime();
5      simtime_t phase = ( timeslotPhaseSize * 8.0 ) / datarate;
6      /*
7       * The interval contains two phases, one for high, one for low
8       * An express-frame can always be sent
9       */
10     simtime_t interval = phase * 2;
11
12     /*
13      * Return true if the timeslot is disabled (size = 0)
14      * An express-frame can always be sent
15      *
16      * A low frame can also always be sent, because if there are no high frames and only low frames,
17      * they can be sent in the phase for the high frames and high frames always come before low frames
18      */
19     if( timeslotPhaseSize == 0 || currentQueueName == EXPRESS_RING || currentQueueName == EXPRESS_INTERNAL
            ||
20             currentQueueName == LOW_RING || currentQueueName == LOW_INTERNAL)
21     {
22         /*
23          * If size is 0 the timeslot-mechanism is disabled
24          */
25         return 0x01;
26     }
27     else if( currentQueueName == HIGH_RING || currentQueueName == HIGH_INTERNAL )
28     {
29         /*
30    * if ( time modulo (2 * phasesize) ) < phasesize
31    * the high-frame can be sent
32    */
33         if( fmod( now.dbl(), interval.dbl() ) < phase.dbl() )
34         {
35             return 0x01;
36         }
37         else
38         {
39             return 0x00;
40         }
41     }
42     else
43     {
44         return 0x00;
45     }
46 }
47 ...
```

Listing 15.5: switch/Scheduler.cc - Zeitschlitzverfahren

15.6. CPU: Lastgenerator

```
1  ...
2  EthernetIIFrame* CPU::generateOnePacket(SendData sendData)
3  {
4      const char* framePrioArr[] = {"EXPRESS", "HIGH", "LOW"};
5
6      EthernetIIFrame *result_ethTag = NULL;
7
8      const char* ethFrameName = "Uni";
9
10     if(sendData.destination.isBroadcast()) {
11         ethFrameName = "Broad";
12     }
13     else if(sendData.destination.isMulticast()) {
14         ethFrameName = "Multi";
15     }
16
17     /*
18      * Display Seq# in GUI
19      */
20     std::stringstream ss;
21     ss << ethFrameName << " From: " << getParentModule()->getFullName() << " (" << framePrioArr[sendData.
           frameprio] <<") Seq#" << sequenceNumber;
22     ethFrameName = ss.str().c_str();
23
24     result_ethTag = MessagePacker::createETHTag( ethFrameName, sendData.destination, macAddress );
25
26     hsrMessage *result_hsrTag = NULL;
27     result_hsrTag = MessagePacker::createHSRTag( "HSR", 1, sequenceNumber );
28
29     vlanMessage *result_vlanTag = NULL;
30     result_vlanTag = MessagePacker::createVLANTag( "vlan", sendData.frameprio, 0, 0, 0 );
31
32     dataMessage* result_dataMessage = NULL;
33     result_dataMessage = MessagePacker::createDataMessage( "stdData", sendData.paketgroesse, messageCount++
           );
34
35     EthernetIIFrame* result_ethFrame = NULL;
36     result_ethFrame = MessagePacker::generateEthMessage( result_ethTag, result_vlanTag, result_hsrTag,
           result_dataMessage );
37
38
39     MessagePacker::deleteMessage(&result_ethTag, &result_vlanTag, &result_hsrTag, &result_dataMessage);
40
41     EV << "[ OK ] " << sendData.frameprio << " Message created at: " << result_ethFrame->getCreationTime()
           << " created." << "  |  Seq#: " << sequenceNumber << endl;
42     sequenceNumber++;
43
44     return result_ethFrame;
45 }
46 ...
```

Listing 15.6: cpu/CPU.cc - Lastgenerator