Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Philipp Kloth

Visualisierung des Zeitverhaltens von Echtzeitnetzwerken basierend auf Gantt Charts

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Philipp Kloth

Visualisierung des Zeitverhaltens von Echtzeitnetzwerken basierend auf Gantt Charts

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik (Bachelor of Science)
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Korf
Zweitgutachter: Prof. Dr. Fohl

Eingereicht am: 21. September 2015

Philipp Kloth

Thema der Arbeit

Visualisierung des Zeitverhaltens von Echtzeitnetzwerken basierend auf Gantt Charts

Stichworte

Echtzeit-Ethernet Netzwerk, Simulation, OMNeT++, Eventlog, Gantt Charts, Eclipse Plug-in, Latenzen, Jitter

Kurzzusammenfassung

Die Bachelorarbeit befasst sich mit der Visualisierung des Zeitverhaltens von Echtzeit-Ethernet Netzwerken auf Basis von Gantt Charts. Zum Darstellen der Diagramme wird ein Eclipse Plug-in entwickelt, welches sich in die Simulationsplattform OMNeT++ integrieren lässt. Weiterhin werden Diagramm Semantiken konzipiert, welche zum Analysieren des Zeitverhaltens von Netzwerken dienen. Die resultierenden Diagramme werden anschließend im Plug-in genutzt. Es werden verschiedene Methoden zur Datenerfassung der zu visualisierenden Daten verglichen und sich für eine Methode entschieden. Am Ende werden die Ergebnisse der Evaluierung zum Plug-in und Eventlog aufgeführt.

Philipp Kloth

Title of the paper

Visualization of the timing of Real-Time networks based on Gantt charts

Keywords

Real-Time network, Simulation, OMNeT++, Eventlog, Gantt charts, Eclipse Plug-in, Latency, Jitter

Abstract

This bachelor thesis deals with the visualization of the timing of Real-Time Ethernet networks, based on Gantt charts. To illustrate the diagrams an Eclipse plug-in is developed, which can be integrated into the simulation platform OMNeT++. Furthermore diagram semantics are created, which are used to analyze the timing of networks. The resulting charts are used in the plug-in. various data collection methods for the data to be visualized,will be compared and one method will be selected. Finally the results of the evaluation of the plug-in and event log are listed.

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **3**

2.1. Echtzeit-Ethernet Netzwerk 3
2.2. OMNeT++ 6
2.3. Eclipse Plug-in 8
2.4. Gantt Charts 10

3. Analyse **12**

3.1. Anforderungsanalyse 12
3.2. Zeitenanalyse 13
3.3. Diagrammanalyse 15
3.3.1. Balkensemantik 15
3.3.2. Analyse Diagramm: Nachrichten Gruppierung 19
3.3.3. Analyse Diagramm: Ursachenfindung 22
3.4. Datenanalyse 25
3.4.1. Datenerfassung 26
3.4.2. Methoden zur Datenerfassung in OMNeT++ 27
3.4.3. Zusammenfassung und Wahl der Methode 30
3.5. Zusammenfassung der Analyse 30

4. Konzept **31**

4.1. OMNeT++ eigenes Eventlog 31
4.2. Programm 33
4.2.1. Plug-in Einstellungen, Bibliotheken und extensionpoints 33
4.2.2. GUI Design 34
4.2.3. Datenerfassung 36

5. Implementierung **37**

5.1. OMNeT++ eigenes Eventlog 37
5.2. Eclipse Plug-in 41
5.2.1. Programm Architektur 41
5.2.2. Konfiguration 42
5.2.3. Controller 44
5.2.4. Model 46
5.2.5. View 50

6. Evaluierung 53
6.1. Evaluierungskriterien 53
6.2. OMNeT++ 53
6.2.1. Simulationsgeschwindigkeit 54
6.2.2. Dateigröße 55
6.2.3. Betriebssystem Test 55
6.3. Plug-in 56

7. Zusammenfassung 58

A. Gantt Chart Timing Analyzer - Manual 60

Tabellenverzeichnis

3.1. Semantik - Latenz für eine Nachricht 16
3.2. Semantik - Latenzen als Gruppe 17
3.3. Semantik - Modullatenzen 18
3.4. Ermittelte Ende-zu-Ende Latenzen aus dem Beispiel-Netzwerk 20
3.5. Ermittelte Switch Latenzen aus dem Beispiel-Netzwerk 23
3.6. Messungen der Aufzeichnungs Methoden 27

6.1. Simulationsgeschwindigkeiten der Aufzeichnungsmethoden 54
6.2. Messungen Dateigröße der Ausgabedateien 55
6.3. Simulationsgeschwindigkeiten der Aufzeichnungsmethoden 56
6.4. Zeiten für die Sammlung der Daten 56
6.5. Testkriterien - Angewandt auf verschiedene Projekte 57

Abbildungsverzeichnis

2.1. Netzwerk Topologie 2 Module 6
2.2. Netzwerk Aufbau OMNeT++ 7
2.3. Eclipse Plattform Übersicht 9
2.4. Gantt Chart 11

3.1. Latenzen Ende-zu-Ende 15
3.2. Modullatenz 16
3.3. Beispiel-Latenz 16
3.4. Gruppenlatenz 17
3.5. Modullatenz als Gruppe Variante 1 18
3.6. Modullatenz als Gruppe Variante 2 18
3.7. Topologie Beispiel 19
3.8. Diagramm Ebene 1 - Hop Anzahl 21
3.9. Diagramm Ebene 2 - Hop Anzahl 22
3.10. Diagramm Ebene 3 - Hop Anzahl 22
3.11. Diagramm Ebene 1 - Switch 24
3.12. Diagramm Ebene 2 - Switch 25
3.13. Diagramm Ebene 2 - Switch 25

4.1. GUI Skizze: Startansicht 34
4.2. GUI Skizze: Auswahlansicht 35
4.3. GUI Skizze: Diagrammansicht 36

5.1. Projekt Aufbau 41
5.2. MANIFEST.MF Datei 44
5.3. Controller.java Klasse 44
5.4. CollectionData.java Klasse 47
5.5. TrafficClass.java Klasse 47
5.6. Message.java Klasse 48
5.7. Nassi Schneidermann Diagramm - Ablauf Nachrichtenerkennung 49
5.8. Time.java Klasse 49
5.9. View.java / Barchart.java Klassen 50
5.10. Balken Diagramm als Gantt Chart 52

6.1. Messungen Simulationsgeschwindigkeit 55
6.2. Messungen Datensammlung 56

Listings

5.1. Variablen im Header 37
5.2. Durchsuchen der .ini Datei und Setzen des Status 38
5.3. Rausschreiben der Daten in die Eventlog Datei 40
5.4. plugin.xml - command extensionpoint 42
5.5. plugin.xml - menu extensionpoint 43
5.6. setter Methoden der Controller.java Klasse 44
5.7. Auszug der Funktion searchingData() 45
5.8. Helferfunktionen der Controller.java Klasse 45
5.9. Funktion zum Berechnen von Minimum, Maximum und Durchschnitt 50

1. Einleitung

Echtzeitnetzwerke erhalten in der heutigen Zeit immer mehr Aufmerksamkeit. Sie werden in vielen Bereichen eingesetzt. Dazu gehören die Kommunikation in industriellen Automatisierungsanwendungen, in Flugzeugen und in Fahrzeugen. Da die Arbeit in der CoRE (Connection over Realtime-Ethernet) Projektgruppe der HAW (vgl. CoRE Research Group) entsteht, wird speziell auf die Kommunikation über Echtzeit-Ethernet eingegangen. Ethernet ist die meist genutzte local area networking (LAN) Technologie weltweit und in Verbindung mit höheren Protokoll-Schichten unterstützt es bereits Funktionen wie Voice-over-IP (VoIP), Audio- und Video Streaming, Echtzeit-Sicherheitskritische Steuerungen und vieles mehr. (vgl. Metcalfe u. a., 2014) Ein Echtzeitnetzwerk setzt ein klar definiertes Zeitverhalten, welches unter allen Betriebsbedingungen gewährleistet ist, voraus. In der Implementierungsphase und zur Feststellung der einwandfreien Funktionalität müssen alle aufgetretenen Latenzen eines Netzwerkes exakt überprüft werden. Damit der Nutzer die aufgetretenen Latenzen überblicken und auswerten kann, können sie visuell über Diagramme dargestellt werden. Anhand von Visualisierungen können intuitive Erkenntnisse vermittelt werden, welche zur Analyse und zur Kommunikation der dargestellten Daten dienen. (vgl. Schumann, 2013). Daher ist der Einsatz geeigneter Diagramme zum Visualisieren des Zeitverhaltens eines Echtzeitnetzwerkes ein relevantes Hilfsmittel, um bei der Analyse und Fehlersuche im Netzwerk zu unterstützen. Die CoRE Projektgruppe arbeitet an verschiedenen Simulationsmodellen für Echtzeit-Ethernet Protokolle und Feldbusse in der Simualtionsumgebung OMNeT++. Bei jeder Umstrukturierung und Weiterentwickelung des Netzwerkes muss das Zeitverhalten neu verifiziert werden. Durch die Erweiterung, das Zeitverhalten visuell betrachten zu können, entstehen Vorteile im Workflow und der Kommunikation zwischen den Mitarbeitern.

Ziel der Arbeit

Das Ziel der Arbeit setzt sich aus zwei Teilen zusammen. Zum einen müssen die relevanten Zeitdaten aus der OMNeT++ Simulationsumgebung während der Laufzeit aufgezeichnet werden. Andererseits muss ein Eclipse Plug-in entwickelt werden, welches diese Zeiten auswertet, berechnet und die resultierenden Ergebnisse in einem Gantt Chart darstellt. Für die

Darstellung der Diagramme muss eine geeignete Semantik entworfen werden, welche eine exakte Analyse der Netzwerke ermöglicht und eventuell auf Fehlerursachen hinweist. Weiterhin muss analysiert werden, welche Zeiten für die Funktionalität von Echtzeit-Netzwerken relevant sind.

Struktur der Arbeit

Kapitel 2 befasst sich mit den Grundlagen, die zum Verständnis der nachfolgenden Kapitel dienen. In diesem werden Echtzeit-Ethernet Netzwerke, die Simulationsumgebung OMNeT++, Eclipse Plug-ins und Gantt Chart näher erläutert. Danach folgt die Analyse. In der Analyse werden alle notwendigen Informationen, die zur Erstellung eines Konzepts erforderlich sind, erarbeitet. Dazu gehören eine Anforderungsanalyse, Zeitenanalyse, Datenanalyse und Diagrammanalyse. Im Anschluss folgt das Konzept der Arbeit. In Kapitel 5 wird die Implementierung des Konzepts beschrieben. Daraufhin wird das entwickelte Plug-in evaluiert und zum Schluss wird die gesamte Arbeit zusammengefasst.

2. Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, welche zum Verständnis der Arbeit beitragen. Zuerst werden die wichtigsten Anforderungen an Echtzeit-Ethernet Netzwerke erläutert und der allgemeine zu visualisierende Netzwerkaufbau gezeigt. Im Anschluss wird die genutzte OMNeT++ Simulationsumgebung näher beschrieben. Am Ende des Kapitels werden noch grundlegende Informationen zu Eclipse und Eclipse Plug-ins sowie zu Gantt Charts erläutert.

2.1. Echtzeit-Ethernet Netzwerk

Echtzeitsysteme müssen alle zeitlichen Anforderungen, welche gestellt werden, erfüllen. Dabei unterscheidet man in der Regel zwischen harter und weicher Echtzeit: (vlg. Tanenbaum, 2009).

harte Echtzeit Die Deadlines bei harter Echtzeit sind absolut und müssen über die gesamte Laufzeit eingehalten werden, da es sonst zu einer Katastrophe kommen kann. Ein gutes Beispiel bildet das Antiblockiersystem in Fahrzeugen. Die benötigten Daten werden durch einen Sensor erfasst und müssen dann weitergeleitet werden zum Auswerten. Die Zeit vom Sensor zum auswertenden Modul ist über das vorhandene System präzise festgelegt. Das bedeutet, dass die Deadlines exakt eingehalten werden müssen, damit das System entsprechend reagieren kann.

weiche Echtzeit Verletzungen von Deadlines sind nicht erwünscht, jedoch tolerierbar. Das bedeutet, dass bei Nichteinhaltung der Deadline keine Katastrophe entstehen würde. Ein Beispiel für weiche Echtzeit ist das Audiosystem im Fahrzeug. Eine nicht eingehaltene Deadline hätte hier keine kritischen Auswirkungen, jedoch würde sich die Musik seltsam anhören, wenn die Daten nicht in der vorgesehenen Zeit ankommen.

Bezogen auf Echtzeit-Ethernet Netzwerke ist es nicht nur erforderlich, dass alle Frames ankommen, sondern, dass Frames in einer bestimmten Zeit garantiert ankommen. Werden diese vorgegebenen Zeiten nicht eingehalten, sind die Frames nicht nur zu spät eingetroffen, sie können im Fall von harter Echtzeit auch unbrauchbar sein. Dadurch ist es für ein Netzwerk nicht

nur wichtig kontinuierlich (ohne Ausfälle des Netzwerkes) zu laufen, auch die Reaktionszeiten und Geschwindigkeiten spielen eine grundlegende Rolle bei der Wahl des einzusetzenden Netzwerkes. Nicht eingehaltene Deadlines am Beispiel von Fahrzeugen können fatale Folgen für die Insassen und das Fahrzeug haben.

Unterschieden werden harte und weiche Echtzeit in Ethernet-Netzwerken anhand von Prioritäten. Damit Echtzeitverhalten im Netzwerk bereitgestellt werden kann, werden IP-Pakete in unterschiedliche Klassen eingeteilt. (vgl. Holleczek und Vogel-Heuser, 2001). Diese werden im weiteren Verlauf *Traffic-Klassen* genannt. Das bedeutet, das anhand der Traffic-Klassen unterschieden werden kann, ob der aufgezeichnete Traffic Echtzeitanforderungen erfüllt oder nicht. Für die Darstellung der Diagramme ist es daher relevant, die angezeigten Zeiten über die Traffic-Klassen zu unterscheiden.

Netzwerk Aufbau

Die zu visualisierenden Switched-Ethernet Netzwerke bestehen aus Modulen (Nodes oder Gateways) und Switches, die in einer Stern-Topologie angeordnet sind. Es wird in dieser Arbeit ausschließlich von Switched-Ethernet ausgegangen. Der Sendevorgang eines Frames vom Startmodul zum Empfängermodul ist wie folgt:

1. Erstellen der Nachricht im Startmodul
2. Senden der Nachricht über den Link zum Switch
3. Empfangen der Nachricht im Switch
4. Weiterleiten der Nachricht über den Link zum Zielmodul (bei größeren Topologien zum nächsten Switch)
5. Empfangen der Nachricht im Zielmodul
6. Auswerten und Löschen der Nachricht

Die Start- bzw. Empfängermodule dienen zum Erstellen, Senden, Empfangen und Auswerten von Ethernet-Frames. Switches dienen als Verteiler der Datenpakete. Sie können die Wege anhand der MAC Adressen speichern und somit direkte Verbindungen zwischen den Nodes herstellen.

Für das Verständnis der Diagramme ist das puffernde Verhalten der Switches von Bedeutung.

Die Weiterleitungszeit von Ethernet-Frames hängt von der Netzauslastung ab, da die Frames zwischen Ein- und Ausgangsspeicher sortiert werden müssen (vgl. Bormann und Hilgenkamp, 2006). Dafür besitzen Switches Zwischenpuffer (engl. Queue). Darüber wird geregelt, dass hochprioire Echtzeit-Frames schneller weitergeleitet werden und niedrig priorisierte Frames warten können.

Die kleinste mögliche Topologie für Switched-Ethernet Netzwerke besteht aus 2 Modulen und einem Switch. (siehe Abbildung 2.1). In den zu visualisierenden Netzwerken können die Nachrichten von einem Modul zum anderen geschickt werden oder per Multicast an mehrere Empfänger. Zwischen den Modulen befinden sich Links, die je nach Hardwareeigenschaft unterschiedliche Übertragungsraten besitzen.

Jedes Modul besitzt verschiedenen Submodule. Die Hierachieebenen sind dabei unbegrenzt. Für die Erkennung der Topologie und die Sammlung von Frameinformationen, die notwendig sind zum Zusammenstellen der Daten für die Diagramme, sind bestimmte Submodule verantwortlich, die im Folgenden erklärt werden:

phy Dieses Submodul steht für die physikalischen Ports, an welchen die Kabel angeschlossen werden. Über die Nummer des Ports kann eine Verbindung zwischen zwei Switches oder einem Modul und einem Switch zugeordnet werden.

mac Das mac Modul ist wiederum ein Submodul vom phy. In diesem Submodul trifft die Nachricht im übergeordneten Modul ein und verlässt das Modul wieder.

App Das App Submodul dient zum Erstellen der Frames. Wichtig bei diesem Modul ist, dass der Name variabel gewählt werden kann. Das bedeutet, dass *App* im Namen des Moduls vorkommt, jedoch an unterschiedlichen Stellen, worauf bei der Selektion geachtet werden muss.

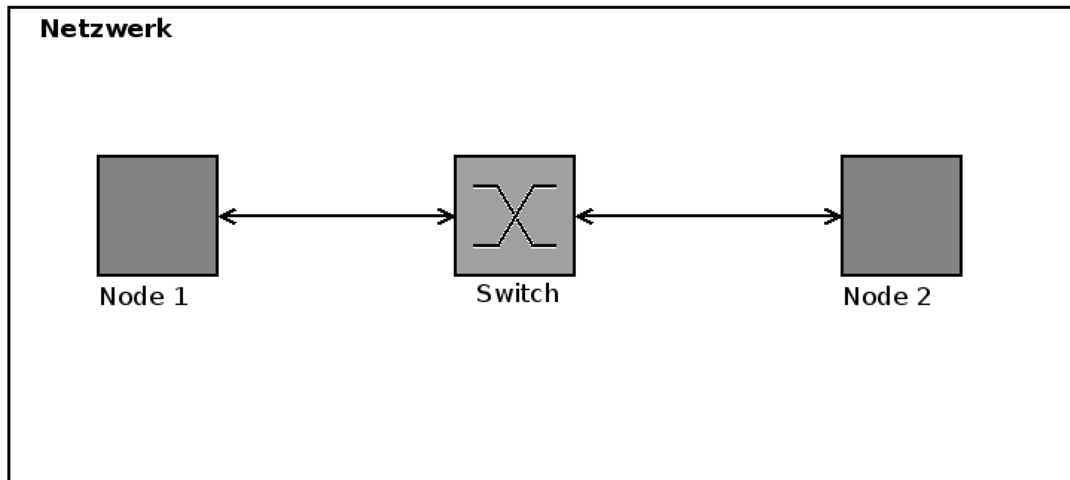


Abbildung 2.1.: Netzwerk Topologie 2 Module

2.2. OMNeT++

OMNeT++ (Objective Modular Network Testbed in C++) ist ein erweiterbares, modulares, komponentenbasiertes C++ Framework für Netzwerk Simulationen (vgl. Omnet++ Community, a). Die Simulation ist diskret Ereignis gesteuert. OMNeT++ unterstützt die Simulation beliebiger Netzwerkarchitekturen und Protokolle. Dazu werden entsprechende Module in das OMNeT++ Framework eingehangen. Es ist ein Open-Source Projekt und für wissenschaftliche Arbeiten oder private Nutzung kostenlos. Das Framework ist für die Plattformen Windows, MAC/OS und Linux verfügbar. Es ist in das Eclipse Framework eingebettet, welches als Grundlage für die Benutzeroberfläche dient (vgl. Eclipse Foundation, b) und dadurch eine Erweiterung über Eclipse Plug-ins bietet, welche in dieser Arbeit zum Einsatz kommen.
Switches und Endpunkte werden über Module definiert. Die Kommunikation zwischen den Modulen erfolgt über Nachrichten, welche als Ethernet Frames deklariert sind. Eine Nachricht beinhaltet beliebige Daten und einen Zeitstempel. Jedes Modul besitzt ein oder mehrere Input und Output Gates, zwischen denen sogenannte Connections bestehen können (siehe Abbildung 2.2). Diese Connections können gerichtete oder ungerichtete Verbindungen sein, worüber dann die Nachrichten ausgetauscht werden. Die Nachrichten verlassen das Modul durch das Output Gate und treffen im Input Gate des Ziel Modules ein.
OMNeT++ arbeitet Ereignisgesteuert und sammelt alle anfallenden Ereignisse in einem entsprechenden Eventlog. Darüber hinaus ist es auch möglich, Vector und Scalar Dateien zu erstellen um sich einen Überblick über die im Netzwerk entstandenen Daten und Übertragungs-

zeiten zu verschaffen. Eine genauere Unterscheidung zwischen diesen Logging Möglichkeiten sind in Abschnitt 3.4 zusammengefasst.

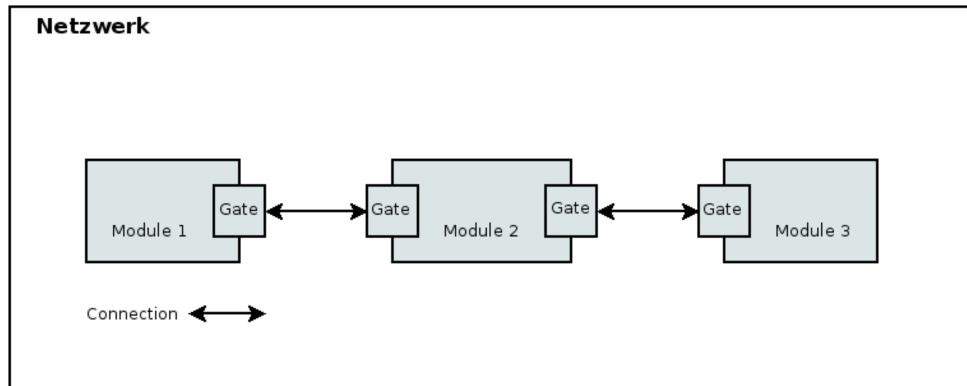


Abbildung 2.2.: Netzwerk Aufbau OMNeT++

Vorhandene CoRE Projekte

In der CoRE Projekt Gruppe an der HAW Hamburg (vgl. CoRE Research Group) sind bereits einige Echtzeit Netzwerksimulationen in OMNeT++ realisiert worden. Drei bereits implementierte Projekte (vgl. CoRE4INET), die auch zum Testen des entwickelten Plug-ins genutzt werden, sind:

- TTEthernet (AS6802) bestehend aus 3 Traffic-Klassen: Time-triggered (TT), Race-constrained (RC) und Best-effort-Traffic (BE).
- IEEE 802.1 Audio Video Bridging (AVB), ebenfalls wie TTEthernet eingeteilt in 3 Traffic-Klassen: Time-triggered (TT), AVB Stream und Best-effort-Traffic (BE).
- Signals and Gateways: Dieses Framework dient zu Erstellung von Netzwerken bestehend aus Switches, Gateways und ECU's (Steuerungseinheiten). In den Gateways treffen CAN Nachrichten aus verschiedenen Fahrzeugsteuerungen ein, diese werden dann über einen Ethernet-Frame verschickt. Im Ziel Gateway werden aus den Frames wieder CAN Nachrichten erstellt und weitergeleitet zur Steuerung. Für das Zeitverhalten wird nur das Ethernet-Netzwerk zwischen Gateways und Switches betrachtet.

2.3. Eclipse Plug-in

Damit eine gute Übersicht zu Eclipse Plug-ins geschaffen werden kann, wird zuerst einmal die Idee und der allgemeine Ansatz von Eclipse beschrieben. Danach werden die Möglichkeiten und die Herangehensweise zur Entwicklung eines Plug-ins erläutert und am Ende wird eine Liste zu den wichtigsten Punkten, die bei der Entwicklung eingehalten werden sollten, aufgeführt.

Eclipse Architektur

Eclipse ist nicht nur als IDE (integrated development environment) konzepiert, sondern als allgemeine Tool-Integrationsplattform gedacht. Es soll dem Benutzer ermöglichen, durch eine Benutzeroberfläche mit gleichem Erscheinungsbild eine große Anzahl von unterschiedlichen Tools zu verwenden. Durch diese einheitlichen Benutzerinteraktionen wird die Einarbeitungszeit erheblich verringert und die Arbeit mit neuen Erweiterungen vereinfacht. (vgl. Shavor u. a., 2004). Es gibt mittlerweile über 250 Open-Source Projekte (vgl. Eclipse Foundation, b), die auf diese Struktur aufsetzen. Eines davon ist auch das OMNeT++ Framework. Eine Eclipse IDE bietet in ihrer Ausgangsform wenige Funktionen. Alle zusätzlichen Möglichkeiten werden durch Plug-ins bereitgestellt. Der Aufbau der Eclipse Platform ist in Abbildung 2.3 veranschaulicht. Man kann erkennen, dass als Grundfunktionen nur die GUI-Werkzeuge (SWT und JFACE), die allgemeine Benutzeroberfläche (Workbench), der Workspace, Hilfe und Team bereitgestellt werden. Eigene Erweiterungen und das Java JDT sind als Plug-ins eingebunden und dienen als Erweiterung. Für den Eclipse-Nutzer ist das jedoch nicht zu erkennen.

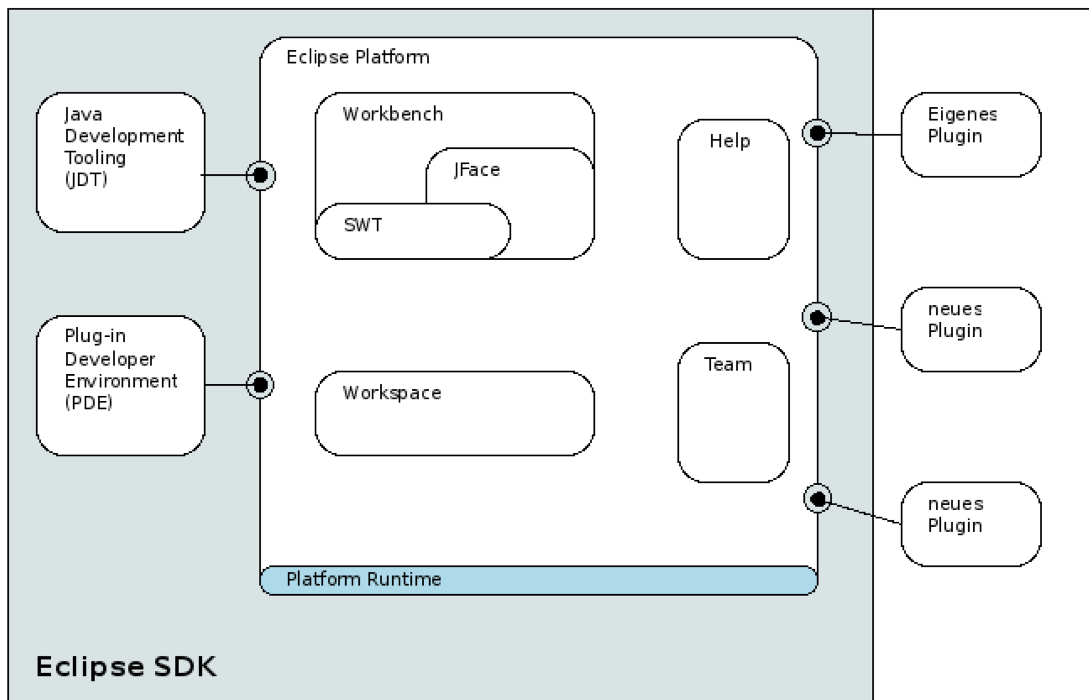


Abbildung 2.3.: Eclipse Plattform Übersicht

Grundlagen zur Plug-in Entwicklung

Ein Plug-in dient zur Erweiterung der Eclipse-IDE und verpackt den geschriebenen Code in eine modulare, erweiterbare und teilbare Einheit. (vgl. Eclipse Foundation, a). Für die Plug-in Entwicklung benötigt man zuerst das Plug-in Developer Environment (kurz PDE) von Eclipse. Diese Umgebung liefert Werkzeuge, welche zur Entwicklung erforderlich sind. Die Erweiterbarkeit einer Eclipse IDE über Plug-ins erfolgt über extensions und extensionpoints. Diese ermöglichen, dass Plug-ins jederzeit zu einem bestehenden Eclipse System hinzugefügt werden können, ohne das komplette System neu übersetzen zu müssen.

extensionpoint: Ein Plug-in kann einen Erweiterungspunkt (extensionpoint) definieren und dafür eine API bereitstellen. Dies ermöglicht, anderen Plug-ins sich mit ihrer extension dem Punkt hinzuzufügen und deren Funktionen zu nutzen. Erweiterungspunkte erhalten Code, der die bereitgestellten Aufgaben ausführen kann.

extension: Ein Plug-in bietet eine Erweiterung (extension) an. Diese baut auf den definierten Erweiterungspunkt auf. Erweiterungen können Code oder Daten sein.

Informationen zu extensions und extensionpoints sowie deren optionalen Einstellungen werden in einer plugin.xml Datei definiert. Informationen zum Plug-in, wie zum Beispiel Versionsnummer und eingebundene Bibliotheken werden in der MANIFEST.MF Datei deklariert. Für die Programmierung der grafischen Benutzeroberfläche stehen die SWT (Standard Widget Toolkit) Werkzeuge zur Verfügung. Plug-ins können exportiert und anschließend in jeder Eclipse basierten IDE eingebunden werden. Um ein Eclipse Plug-in zu implementieren, müssen folgende fünf Schritte beachtet werden (vgl. Shavor u. a., 2004, S. 221):

1. Wo soll das Plug-in in die Eclipse Plattform integriert werden? (extensionpoints finden)
2. Welche Anforderungen sollen die extensionpoints erfüllen?
3. Deklarieren der Plug-in Manifest Datei
4. Implementation der Funktionen für die Erweiterung
5. Installation des Plug-ins

Genauere Details zu diesen Schritten befinden sich im Kapitel 5.

2.4. Gantt Charts

Gantt Charts wurden in den 1910er Jahren von Henry Gantt entworfen, um Projekte übersichtlicher planen zu können. Es hat sich zur meistgenutzten Darstellungsform entwickelt, um Aktivitäten oder Ereignisse in einem zeitlichen Rahmen zu betrachten (vgl. Wilson, 2002). In Abbildung 2.4 ist ein Beispiel Gantt Chart zu sehen. In der Grundstruktur befindet sich auf der linken Seite eine Liste von Ereignissen, oben die Zeitachse und in der Mitte die Ereignisse in Balkenform. Dadurch kann der Start und das Ende sowie die Dauer eines jeden Ereignisses anhand der Balken abgelesen werden. Darüber hinaus erhält man Informationen, wann sich Ereignisse zeitlich überschneiden. Ereignisse können auch als Gruppe zusammengefasst werden, diese Gruppe wird durch einen Balken dargestellt und die einzelnen Ereignisse durch ein separates Gantt Chart, welches zum Beispiel durch Klicken auf den Gruppenbalken angezeigt werden kann.

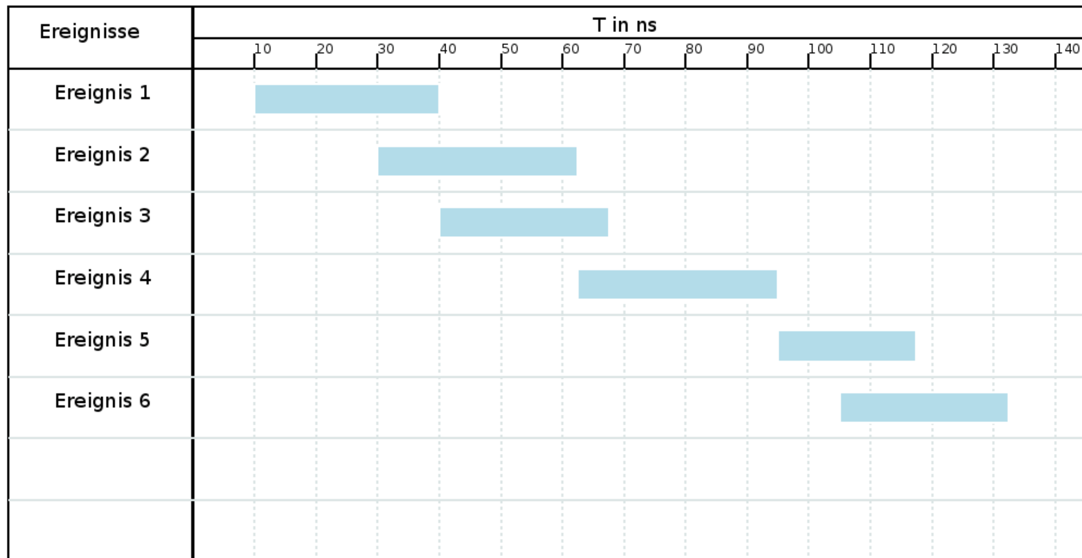


Abbildung 2.4.: Gantt Chart

3. Analyse

In diesem Kapitel entsteht eine Analyse zum Thema der Arbeit. Mit den Erfahrungen aus dem Kapitel wird dann im weiteren Verlauf das Konzept für die Arbeit erstellt. Zuerst wird untersucht, welche Analysen aufgrund der Aufgabenstellung gemacht werden müssen. Danach werden die daraus resultierenden Analysen erstellt und am Ende werden die Ergebnisse noch einmal zusammengefasst.

3.1. Anforderungsanalyse

Es soll ein Programm entwickelt werden, um das Zeitverhalten im Echtzeit-Ethernet Netzwerk visuell über Gantt Charts darzustellen. Die Simulation, die als Grundlage für diese Arbeit dient, ist ereignisgesteuert und bietet daher für jedes Ereignis einen Zeitstempel an. Es muss dementsprechend analysiert werden, welche Menge an Zeiten aus der Simulation relevant sind, um dem Nutzer einen genauen Überblick zu gewähren. Die Anforderung besteht darin, dem Nutzer relevante Daten anzuzeigen, die Schlüsse auf die Funktionsfähigkeit des Netzwerkes bieten und eine Ursachenfindung ermöglichen. Eine genaue Analyse, welche Zeiten dafür ausgewertet werden, findet sich im Abschnitt 3.2 Zeitenanalyse. Für die Anzeige der Zeiten sind Gantt Charts vorgesehen. Der Haupteinsatzort von Gantt Charts ist die Projektplanung. Daher muss geklärt werden, wie sich Gantt Charts für den Fall vom Analysieren des Zeitverhaltens im Netzwerk einsetzen lassen und welche Semantik sinnvoll ist, um die Zeiten aus der Analyse dem Nutzer übersichtlich darzustellen. Eine genaue Analyse zu Gantt Charts ist im Abschnitt Diagrammanalyse 3.3 nachzulesen. OMNeT++ bietet mehrere Methoden, um die angefallenen Ereignisse aufzuzeichnen. Somit entstehen verschiedene Datensammlungen. Es muss dementsprechend analysiert werden, welche Daten zum Darstellen benötigt werden und wie man diese Daten effizient aus der Simulation herausfiltern kann. Eine genaue Analyse zu den Daten findet sich im Abschnitt 3.4 Datenanalyse. Da die zu analysierenden Echtzeitnetzwerke verschiedene Traffic-Klassen besitzen und diese sich durch unterschiedliche Zeitanforderungen unterscheiden, muss die Auswahl der angezeigten Diagramme im Programm anhand der Traffic-Klasse gefiltert werden. Jede Traffic-Klasse kann unterschiedliche Streams besitzen. Diese sind von dem Nutzer selbst wählbar. Dem Nutzer sollte daher eine Auswahl des

Streams geboten werden oder aber eine Anzeige von allen Streams einer Traffic-Klasse. Für den Fall, dass der Nutzer nicht genau entscheiden kann, welche Zeiten für ihn wichtig sind. Eine Anzeige von allen Streams bietet auch einen Überblick über die Traffic-Klasse. Dadurch kann das Zeitverhalten für die gesamte Traffic-Klasse geprüft werden.

3.2. Zeitenanalyse

Es muss zuerst festgelegt, werden welche Zeiten dem Nutzer am Ende durch das Diagramm dargestellt werden sollen. Für die Analyse von Echtzeit-Ethernet Netzwerken gibt es verschiedene Metriken. Die bekanntesten für Echtzeit-Ethernet Netzwerke sind die Latenzzeit, der Jitter und die Paketverlustrate. Bekannt sind die drei Metriken als Quality of Service Parameter. Es ist festgelegt, dass mit diesen Metriken die Qualität des Netzwerkes nachgewiesen werden kann. Die Paketverlustrate spiegelt die Anzahl der Pakete wieder, welche auf dem Weg vom Sender zum Empfänger verloren gegangen sind und ist nicht über eine Zeit definiert. Bei der Analyse von Zeiten sind also nur die Latenzzeit und der Jitter ausschlaggebend, auf die beiden Metriken sollte daher einmal genauer eingegangen werden.

Latenz

Als Latenz bezeichnet man die zeitliche Verzögerung vom Senden des Frames bis zum Eintreffen beim Empfänger. Die Latenz ist abhängig von der Dauer des Sendevorgangs, der Signalausbreitungszeit und der Verzögerung bei Zwischenstationen (vgl. Beck, 2003). Unter der Dauer des Sendevorgangs versteht man die Abhängigkeit zwischen der Datenmenge und der Bandbreite. Die Siganalausbreitungszeit ist abhängig von der Entfernung und der Geschwindigkeit, welche wiederum durch das eingesetzte Medium bestimmt ist, jedoch höchstens Lichtgeschwindigkeit betragen kann. Diese beiden Werte sind fest durch das System definiert, welches eingesetzt wird. Die Verzögerung bei Zwischenstationen hingegen kann durch Staus in den Switches oder durch Verzögerungen beim Versenden aus dem Modul entstehen. Auf diese Größe kann der Nutzer eingreifen, indem er sein Netzwerk zum Beispiel umkonfiguriert oder die Topologie ändert. Mathematisch betrachtet ist die Latenz:

$$(T_{empfangen} - T_{senden})$$

Jitter

Als Jitter versteht man die Varianz der Laufzeit von Frames. Im Idealfall wäre die Latenz bei identischen Nachrichten konstant. In Echtzeit-Ethernet Netzwerken kommt es jedoch vor,

dass Jitter entstehen, zum Beispiel durch Paketstaus in den Switches oder Wartezeiten durch höher priorisierte Pakete. Sind in den Netzwerken geringere Latenzen erforderlich, um die Deadlines einzuhalten, können zu große Schwankungen schwere Auswirkungen auf das System haben. Mathematisch betrachtet ist der Jitter die Differenz aus zwei Latenzen der gleichen Nachrichtenklasse. Die Formel zur Berechnung ist:

$$(T_{empfangen_1} - T_{senden1}) - (T_{empfangen_2} - T_{senden2})$$

Es gilt weiterhin abzugrenzen, welche Latenzen in der Simulation von Bedeutung sind. Aufgrund der Echtzeiteigenschaften und der dadurch erforderlichen Deadlines ist die Ende-zu-Ende Latenz eine Zeit, die dem Nutzer angibt, ob das System alle Deadlines einhalten kann oder ob Verletzungen der Deadlines eingetreten sind. Durch diese Latenzen bekommt der Nutzer einen groben Überblick und kann die Funktionsfähigkeit beurteilen. Es kann jedoch noch nichts über die Ursache bei Fehlerfällen gesagt werden. Wie auf der Abbildung 3.1 zu sehen, entstehen auf der Ende-zu-Ende Route viele Zwischenlatenzen. Durch Darstellung dieser Zwischenlatenzen kann der Nutzer sehen, an welcher Station auf der Route ein Jitter aufgetreten ist. Für den Fall, dass in dem Netzwerk mehr als zwei Nachrichten auf einer Ende-zu-Ende Route geschickt werden, würde die Menge der Latenzen proportional ansteigen. Dadurch geht der Überblick verloren. Zur Evaluierung von Netzwerken spricht man daher oft von Best-Case und Worst-Case Szenerien. Umgeleitet auf den Anforderungsfall für die Latenzen würde es bei vielen Nachrichten eine Best-Case Latenz und eine Worst-Case Latenz bedeuten. Dadurch kann am Ende der größte mögliche Jitter herausgefunden werden und als Gegenstück dazu die optimale Latenz. Deshalb können für die Deadlines zwei neue Erkenntnisse gezogen werden. Erstens kann man durch die Unterscheidung zwischen Best -und Worst Case erkennen, ob die Deadline über die gesamte Simulationszeit eingehalten werden kann, auch wenn eine Nachricht eventuell verzögert ankam. Zweitens ist dadurch der absolute Jitter zu sehen, welchen man dann für ähnliche Fälle als Basis für weitere Analysen nutzen kann.

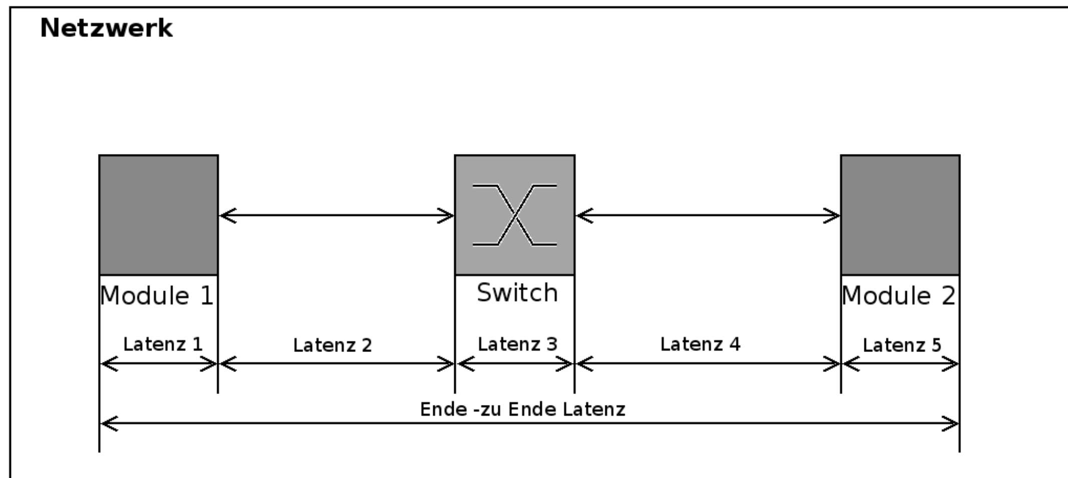


Abbildung 3.1.: Latenzen Ende-zu-Ende

Für diese Arbeit müssen also die Ende-zu-Ende Latenzen und die Zwischenlatenzen in Betracht gezogen werden. Von den ermittelten Latenzen muss das Minimum, das Maximum und der Durchschnitt berechnet werden, damit ein Best und Worst Case Fall visualisiert werden kann. Anhand des Durchschnitts aller Latenzen kann der Nutzer erkennen, ob die gesendeten Nachrichten zum Minimum oder zum Maximum tendieren. Dadurch erhält der Nutzer einen guten Einblick in die Qualität seines Netzwerkes und kann Aussagen zur Einhaltung der bei Echtzeit-Ethernet Netzwerken signifikanten Deadlines machen.

3.3. Diagrammanalyse

In diesem Abschnitt wird zuerst analysiert, wie ein Balken in dem Gantt Chart aufgebaut ist. Dann werden anhand einer Topologie verschiedene Anwendungsfälle dargestellt, um festzustellen, wie die auftretenden Zeiten am besten dargestellt werden können. Am Ende wird die verwendete Semantik festgelegt, welche später implementiert werden soll.

3.3.1. Balkensemantik

In den Gantt Charts sind die Balken definiert mit Startzeit, Dauer und Endzeitpunkt der Aufgabe (vgl. Abschnitt 2.4) . Um eine Semantik festzulegen für die analysierten Zeiten, werden zwei Fälle betrachtet. Als erstes die Modullatenz und Ende -zu Ende Latenz für eine Nachricht und zweitens die Latenzen für mehrere Nachrichten, die dann in einem Balken als Gruppe zusammengefasst werden.

Latenz für eine Nachricht

Für die Latenz einer Nachricht kann die Standarddefinition für Gantt Charts genutzt werden. In Abbildung 3.2 wird angenommen, dass die Nachricht zum Zeitpunkt 40μs in dem Modul angekommen ist und das Modul bei 110μs wieder verlassen hat. Somit ergibt sich eine Latenz laut Berechnung von Abschnitt 3.2 von 70μs. Die Länge des Balkens repräsentiert die Latenz von 70μs, der Anfang des Balkens entspricht dem Ankunftszeitpunkt und das Ende dem Zeitpunkt beim Verlassen des Moduls. Im Fall einer Ende -zu Ende Latenz entspricht der Anfang dem Startzeitpunkt der Nachricht und das Ende den Ankunftszeitpunkt im Ziel Modul. Zusammengefasst als Semantik gilt:

Latenz	Balken Anfang	Balken Ende	Balken gesamt
Modul	$T_{eintreffen}$	$T_{absenden}$	$T_{Latenz} = (T_{absenden} - T_{eintreffen})$
Ende -zu Ende	T_{start}	T_{ende}	$T_{Latenz} = (T_{ende} - T_{start})$

Tabelle 3.1.: Semantik - Latenz für eine Nachricht

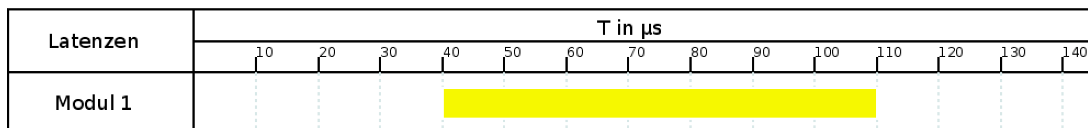


Abbildung 3.2.: Modullatenz

Ende- zu-Ende Latenzen als Gruppe

Um Latenzen als Gruppe zusammenzufassen, werden als Beispiel drei Ende-zu-Ende Latenzen betrachtet, die in einer Nachrichtenklasse aufgetreten sind. Die Beispiel-Latenzen sind in Abbildung 3.3 zu sehen. Die Semantik des Beispiels ist in Tabelle 3.2 definiert.

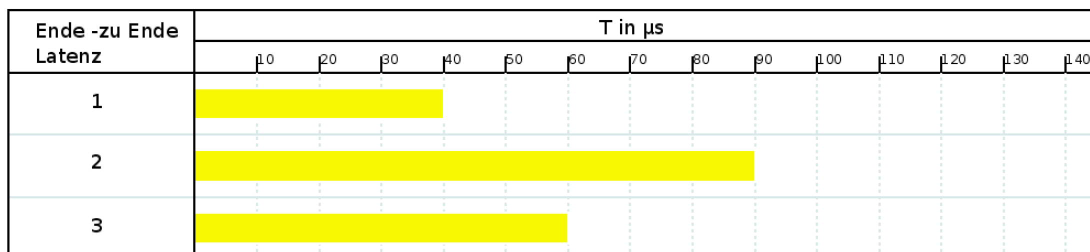


Abbildung 3.3.: Beispiel-Latenz

Um die Ende-zu-Ende Latenzen aus der Zeitenanalyse gruppiert darzustellen, würde man den gruppierten Balken laut den Vorgaben von Gantt Charts von 0μs bis 90μs erstellen. Bei

dieser Darstellung kann dann die maximale Latenz der Nachrichten am rechten Balkenende abgelesen werden. Die minimal aufgetretene Latenz ist in der Gruppenansicht nicht zu erkennen. Für den Nutzer wäre es aber von Vorteil, direkt die maximale und minimale Latenz ablesen zu können. Um beides gruppiert darzustellen, sollte nur das Intervall zwischen minimaler und maximaler Latenz betrachtet werden (vgl. HANSER automotive networks, 2013). An dem Beispiel angewandt, würde der gruppierte Balken von 40μs bis 90μs dargestellt werden. Der Vorteil besteht darin, dass somit gleich der aufgetretene Jitter abgelesen werden kann. Der Anfang des Balkens zeigt die minimale Latenz, das Ende die maximale Latenz und der gesamte Balken repräsentiert den aufgetretenen Jitter (siehe Abbildung 3.4). Somit kann der Nutzer alle relevanten Zeiten auf einen Blick wahrnehmen. Diese Art der Gruppierung wird für die Ende-zu-Ende Latenz und für die Modullatenz genutzt. Da es sich bei den zu analysierenden Netzwerken meistens um eine große Anzahl von Nachrichten handelt, die gruppiert werden, ist ein Durchschnitt von Vorteil. Dadurch ist eine Tendenz zu erkennen, wie häufig diese Latenzen aufgetreten sind. Dieser Durchschnitt wird als Linie im Balken gekennzeichnet. Zusammengefasst als Semantik gilt:

Balken Anfang	Balken Ende	Balken gesamt	Durchschnitt
$T_{Latenz\ min}$	$T_{Latenz\ max}$	$T_{Jitter} = (T_{Latenz\ max} - T_{Latenz\ min})$	$\frac{1}{n}\sum_{i=1}^{n} T_{Latenz\ i}$

Tabelle 3.2.: Semantik - Latenzen als Gruppe

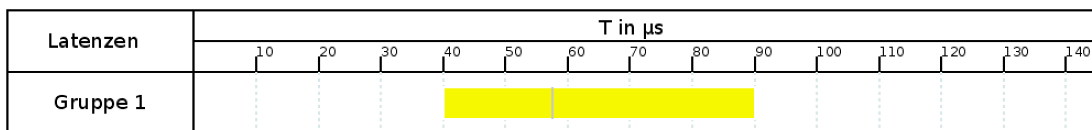


Abbildung 3.4.: Gruppenlatenz

Modullatenzen als Gruppe

Für die Betrachtung einer Menge an Modullatenzen werden zwei Varianten analysiert. Zuerst werden verschiedene Modullatenzen, wie die zuvor beschrieben Ende-zu-Ende-Latenzen, zusammengefasst. Das Ergebnis ist in Abbildung 3.5 gezeigt. Als zweite Variante wird der Best-Case, in dem Fall die minimal aufgetretene Ende-zu-Ende Latenz und der Worst-Case, die maximal aufgetretene Ende-zu-Ende Latenz untereinander dargestellt. Das Ergebnis ist in Abbildung 3.6 zu sehen. Wobei der Best-Case grün und der Worst-Case rot dargestellt ist. Vergleicht man beide Abbildungen miteinander, ist zu erkennen, dass die Darstellung durch Best-

und Worst-Case einfacher und übersichtlicher ist. Außerdem ist durch Variante eins nicht zu erkennen, welche Latenzen in den Modulen aufgetreten sind. Aus dem Grund werden zukünftig die Modullatenzen als Gruppe über den Best- und Worst-Case dargestellt. Der Anfang des Balkens zeigt den Zeitpunkt bei Eintreffen (in) der Nachricht im Modul und das Ende den Zeitpunkt beim Verlassen (out) des Moduls. Die Linklatenzen zwischen den Modulen werden nicht gesondert angezeigt, da sie durch die Länge des Links, die Materialeigenschaften und die Paketgröße festgelegt sind und sich über die Laufzeit nicht verändern. Zusammengefasst als Semantik gilt:

Balken Anfang	Balken Ende	Balken gesamt	Farben
$T_{Latenz\ in}$	$T_{Latenz\ out}$	$T_{Latenz\ out} - T_{Latenz\ in}$	grün = Best-Case / rot = Worst-Case

Tabelle 3.3.: Semantik - Modullatenzen

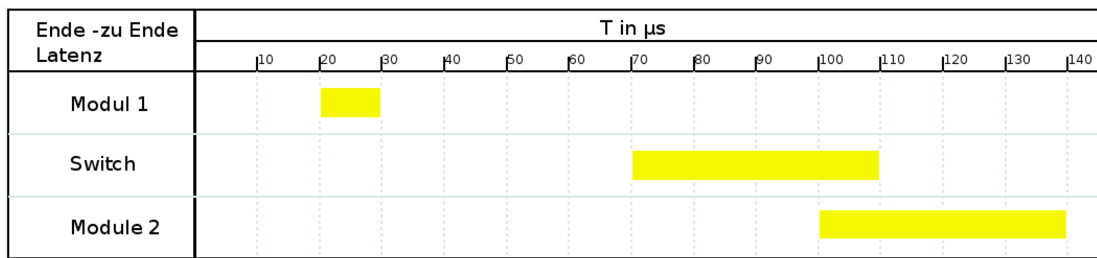


Abbildung 3.5.: Modullatenz als Gruppe Variante 1

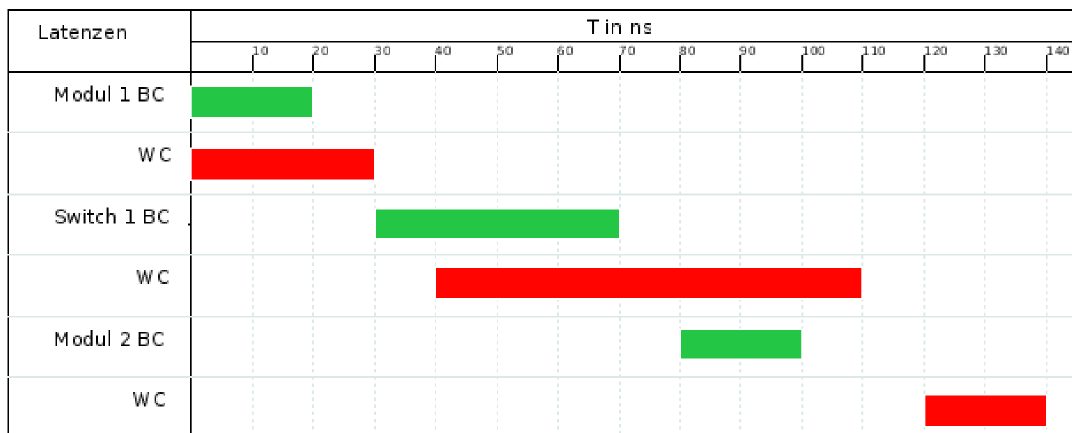


Abbildung 3.6.: Modullatenz als Gruppe Variante 2

3.3.2. Analyse Diagramm: Nachrichten Gruppierung

Im Folgenden wird analysiert, wie Nachrichten als Gruppe im Diagramm zusammengefasst werden können. Für den Benutzer ist die Darstellung der Modullatenzen über Best- und Worst Case am detailreichsten. Man erhält genaue Informationen über die Ende-zu-Ende Latenz und über die einzelnen Modullatenzen. In einem Netzwerk gibt es jedoch im häufigsten Fall verschiedene Traffic-Klassen, diese haben unterschiedliche Streams, welche dann über die Zeit eine Menge an Nachrichten verschicken. Aus dem Grund ist eine Gruppierung wichtig, um dem Nutzer bei der Entscheidung, welche Nachrichten relevant sind, zu unterstützen. Für die Analyse wird die Topologie aus Abbildung 3.7 verwendet. Um einen besseren Überblick zu bekommen, wurde das Netzwerk (AVB large network vgl. CoRE4INET) simuliert und die entstandenen Latenzen aufgezeichnet. In diesem Netzwerk gibt es drei Traffic-Klassen mit verschiedenen Streams. Für die anschließende Analyse wird die Traffic-Klasse mit Best-effort Traffic ausgewertet, welche von Node 10 über Multicast Nachrichten an alle Nodes verschickt. Die gemessenen Latenzen befinden sich in Tabelle 3.4 . In der Tabelle sind die minimalen und maximalen Ende-zu-Ende Latenzen und die Anzahl der Hops aufgelistet. Ein Hop ist definiert durch den Weg eines Netzwerkknotens zum nächsten Netzwerkknoten.

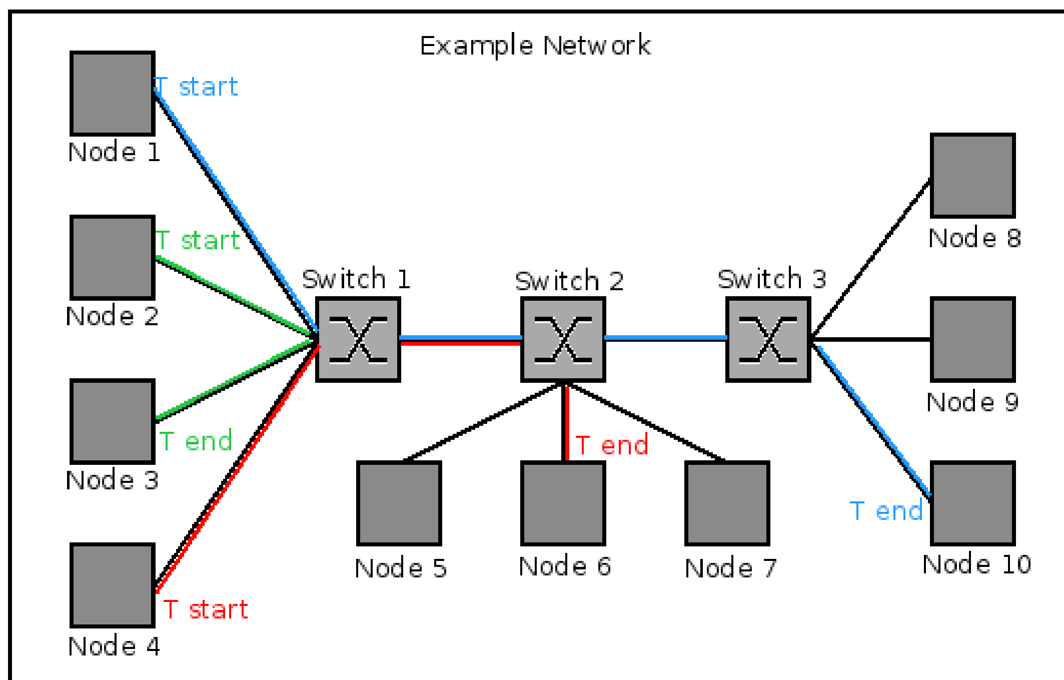


Abbildung 3.7.: Topologie Beispiel

Ende-zu-Ende Weg	Tmin	Tmax	Hop - Anzahl
Node 10 - Node 9	132,56 µs	258,43 µs	2
Node 10 - Node 8	132,56 µs	163,41 µs	2
Node 10 - Node 7	202,84 µs	328,18 µs	3
Node 10 - Node 6	202,84 µs	207,80 µs	3
Node 10 - Node 5	202,84 µs	207,80 µs	3
Node 10 - Node 4	273,12 µs	278,08 µs	4
Node 10 - Node 3	273,12 µs	278,08 µs	4
Node 10 - Node 2	273,12 µs	278,08 µs	4
Node 10 - Node 1	273,12 µs	278,08 µs	4

Tabelle 3.4.: Ermittelte Ende-zu-Ende Latenzen aus dem Beispiel-Netzwerk

Nachrichten als Gruppe

Um alle Ende-zu-Ende Latenzen in einem Diagramm darzustellen, werden die Modullatenzen summiert als Ende-zu-Ende Latenz dargestellt, ohne Best- und Worst-Case. Diese Ansicht ist gut, um einen Überblick über alle Ende-zu-Ende Latenzen von einem ausgewähltem Stream einer Nachrichten Klasse zu erhalten. Würden jetzt jedoch aus vier Nodes Nachrichten per Multicast geschickt werden, wäre die Anzahl der Ende-zu-Ende Latenzen für unser Beispiel-Netzwerk bei 36 Ende-zu-Ende Latenzen. Bei noch größeren Netzwerken entsteht dadurch wiederum eine Unübersichtlichkeit. Es gilt die Frage zu klären: "Wie können die Latenzen als weitere Gruppe zusammengefasst werden?".
Betrachtet man diese Ende-zu-Ende Latenzen, sind Unterschiede durch die Länge des Weges, also durch die Anzahl an Hops, die auf der Route zurückgelegt werden, physikalisch bestimmt. Eine Verbindung von Node 2 zu Node 3 über einen Switch (zwei Hops / grün hervorgehoben) hat im akkumulierten Best-Case immer eine geringere Latenz, als die einer Verbindung über zwei Switches von Node 4 zu Node 6 (drei Hops / rot hervorgehoben). Dadurch ist eine Zusammenfassung der Latenzen über die Hop Anzahl sinnvoll. Somit ist der Vergleich auf einer gleichen Basis möglich und es bietet die Option, den Aufbau des Netzwerkes zu analysieren. Ein Beispiel wäre, dass Frames die eine geringe Deadline benötigen, diese nicht erreichen können, aufgrund der Anzahl an Hops, die zurückgelegt wurden. Weiterhin entsteht eine Information darüber, mit welcher Anzahl von Hops die benötigte Deadline erreicht werden kann (Wenn dieser Traffic über verschiedene Ende-zu-Ende Routen verläuft).
Zusammengefasst ergeben sich drei Ebenen. Auf Ebene eins wird die Menge an Ende-zu-Ende Latenzen über die Anzahl der Hops gruppiert. In Ebene 2 werden alle Ende-zu-Ende Latenzen gefiltert, nach der zuvor ausgewählten Anzahl an Hops und in Ebene drei wird die detaillierte

Route als Best und Worst Case mit Ihren Modullatenzen dargestellt. In den Abbildungen 3.8, 3.9 und 3.10 ist das Ergebnis zu sehen. Als Hop Anzahl wurden 2 Hops ausgewählt und als Ende-zu-Ende Route die Route von Node 10 zu Node 8. Die Semantik für diese Ansichten wird wie folgt definiert:

Semantik Gantt Charts Hop Ansicht

Ebene 1

Gruppierung: Anhand der Anzahl der zurückgelegten Hops für die Menge an Ende-zu-Ende Latenzen

Balkensemantik: siehe Tabelle 3.2 im Abschnitt 3.3.1

Ebene 2

Gruppierung: Menge an Ende-zu-Ende Latenzen mit ausgewählter Hop Anzahl aus Ebene 1

Balkensemantik: siehe Tabelle 3.2 im Abschnitt 3.3.1

Ebene 3

Best-Case und Worst Case Modullatenzen für die ausgewählte Ende-zu-Ende Route aus Ebene 2

Balkensemantik: siehe Tabelle 3.4 im Abschnitt 3.3.1

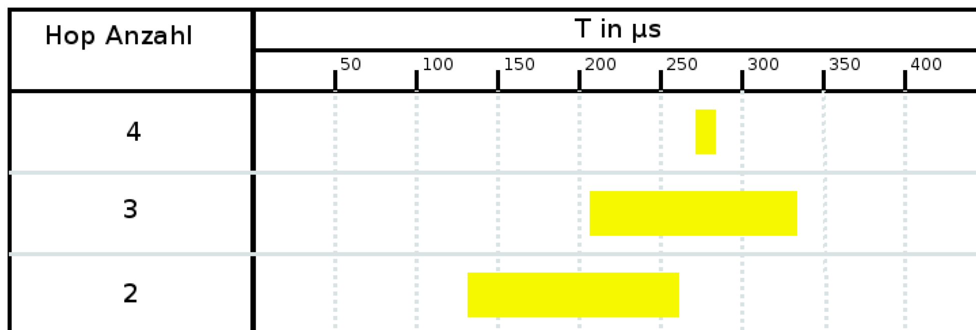


Abbildung 3.8.: Diagramm Ebene 1 - Hop Anzahl

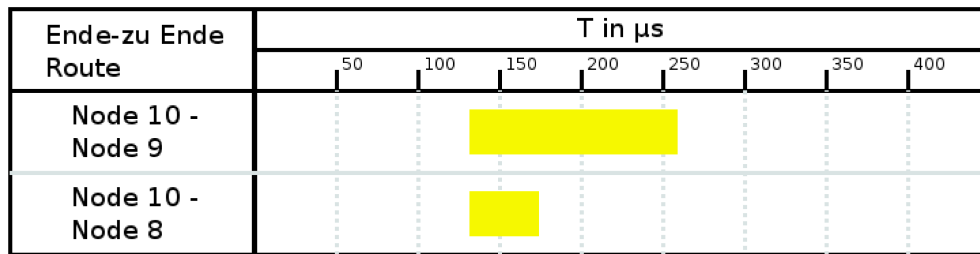


Abbildung 3.9.: Diagramm Ebene 2 - Hop Anzahl

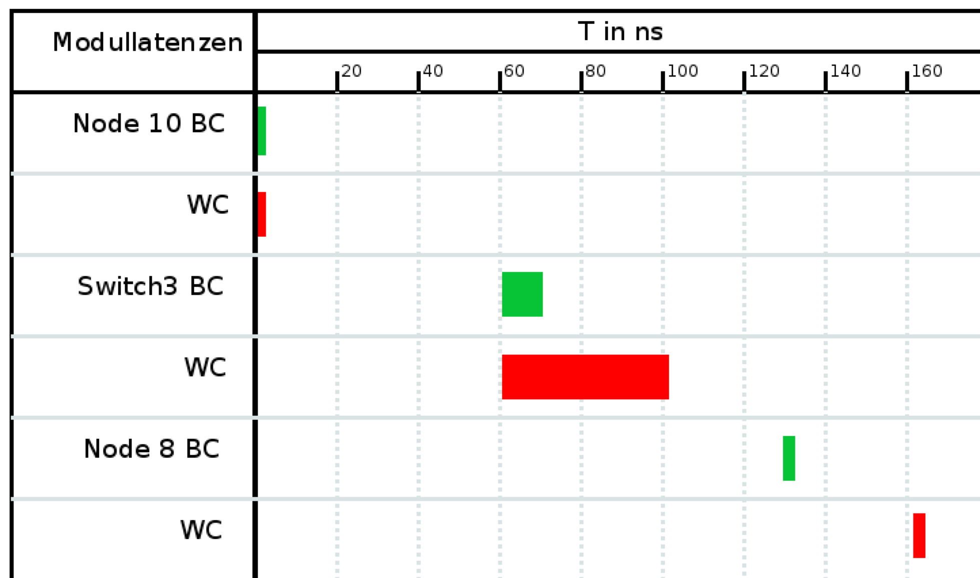


Abbildung 3.10.: Diagramm Ebene 3 - Hop Anzahl

3.3.3. Analyse Diagramm: Ursachenfindung

In diesem Absatz wird untersucht, wodurch erhöhte Latenzen auftreten können und ob eine Möglichkeit besteht, dem Nutzer über die Gantt Charts schon auf diese Ursachen aufmerksam zu machen. Durch die Einteilung der Ebenen über die Anzahl der Hops ist bereits eine Ursachenfindung Anhand der Länge, der zurückgelegten Route, möglich. Somit kann eine Analyse der Netzwerkstruktur stattfinden, welche auf eine Umstrukturierung der Topologie hinweisen kann. In Switched-Ethernet Netzwerken ist ein entscheidender Faktor für auftretende Jitter, die Modullatenz im Switch. In den End-Modulen sind keine großen Jitter zu erkennen. Es gibt jedoch die Möglichkeit, dass eine Nachricht erstellt und dann verzögert abgeschickt wird. Bei diesem Fall ist es jedoch vorprogrammiert und kein Fehler im Netzwerk.

Bei einem Switch treffen die Nachrichten ein und werden anhand von Routingtabellen weitergeleitet zum nächsten Modul. In Echtzeitnetzwerken werden darüber hinaus noch Priorisierungen der Nachrichtenklassen berücksichtigt. Switches enthalten Queues zur Pufferung von Nachrichten für den Fall, dass die Bandbreite ausgelastet ist oder weil niedrig priorisierte Nachrichten auf Grund von höher priorisierten Nachrichten warten müssen. Ist der Traffic in einem Switch hoch oder sind die Pakete groß, kommt es zu Staus, welche einen erhöhten Jitter für die wartenden Nachrichten zur Folge haben. Aus diesem Grund ist es für die Ursachenfindung relevant, die Modullatenzen der verfügbaren Switches zu vergleichen. Demzufolge kann dargestellt werden, welche Switches stärker belastet sind und es können Staus in den Switches erkannt werden. In dem Beispiel-Netzwerk sind folgende Switchlatenzen gemessen worden:

Switch	Tmin	Tmax
Switch 1	8,00 µs	39,15 µs
Switch 2	8,00 µs	133,86 µs
Switch 3	8,00 µs	134,53 µs

Tabelle 3.5.: Ermittelte Switch Latenzen aus dem Beispiel-Netzwerk

Die Übertragung der Latenzen in das Gantt Diagramm erfolgt als Gruppenlatenz, wobei der Anfang des Balkens die minimale Modullatenz und das Ende die maximale Modullatenz aufzeigt. Dadurch sind die aufgetretenen Jitter in jedem Switch über die Balken abzulesen. Damit der Nutzer eine Schnittstelle zur Hop Ansicht bekommt, kann in der nächsten Ebene die Modullatenz des ausgewählten Switches in Abhängigkeit zur Hop Anzahl der Nachrichten betrachtet werden. Um das zu verdeutlichen, werden als Beispiel alle Routen über Switch 1, die durch den Best-effort-Traffic genutzt, werden betrachtet. Die entstandenen Modullatenzen aus Switch 1 für diese vier Routen werden dann gruppiert als Hop Anzahl 4 angezeigt. Dadurch entsteht eine Filterung des aufgetretenen Traffics und eine Schnittstelle zur Hop Ansicht. Nach dem Auswählen wird dann wieder der Best- und Worst-Case angezeigt, um sich einen Gesamteindruck von der Ende-zu-Ende Route verschaffen zu können.
Es entstehen für die Analyse über dem Switch auch drei Ebenen, welche als Beispiel in den Abbildungen 3.11, 3.12 und 3.13 gezeigt werden. Beim Betrachten der Switch Latenzen wird folgendes Problem deutlich. In der letzten Ansicht bei beiden Varianten wird der Worst-Case über die maximale Ende-zu-Ende Latenz aller ausgewählten Nachrichten dargestellt. Bei einer Ende-zu-Ende Route über 3 Hops kann es passieren, dass die Worst-Case Nachricht in dem ersten Switch eine geringe Latenz hat und im zweiten Switch eine stark erhöhte Latenz. Eine andere Nachricht, welche nur eine minimal geringere Ende-zu-Ende Latenz aufweist, hat im

ersten Switch eine stark erhöhte Latenz und im zweiten Switch eine niedrige Latenz. Um das darzustellen, gibt es die Möglichkeit, einen akkumulierten Worst-Case bzw. Best-Case anzuzeigen. Dieser ist in der Simulation nicht aufgetreten, doch er bietet einen Gesamtüberblick über den Best- und Worst-Case jeder Modullatenz. Das würde für beide Varianten eine Ebene vier ermöglichen. Da die Ende-zu-Ende Latenzen jedoch nicht real, sondern virtuell sind, sollte es nur als Zusatz Chart betrachtet werden und nicht als Ebene. Zusammengefasst gilt für die drei Ebenen über die Switch Analyse und für die Zusatz Ansicht folgende Semantik:

Semantik Gantt Charts Switch Ansicht

Ebene 1

Gruppierung: Anhand der Switch Modullatenzen
Balkensemantik: siehe Tabelle 3.2 im Abschnitt 3.3.1

Ebene 2

Gruppierung: Switch Modullatenzen von dem ausgewählten Switch aus Ebene 1, für Nachrichten die auf der Route diesen Switch passiert haben, gruppiert nach der Anzahl von Hops
Balkensemantik: siehe Tabelle 3.2 im Abschnitt 3.3.1

Ebene 3

Best-Case und Worst-yCase Modullatenzen für die ausgewählte Hop Anzahl aus Ebene 2
Balkensemantik: siehe Tabelle 3.4 im Abschnitt 3.3.1

Semantik Gantt Charts akkumulierter Best- und Worst Case

Minimale und Maximale Modullatenzen von allen Nachrichten akkumuliert.
Balkensemantik: siehe Tabelle 3.4 im Abschnitt 3.3.1

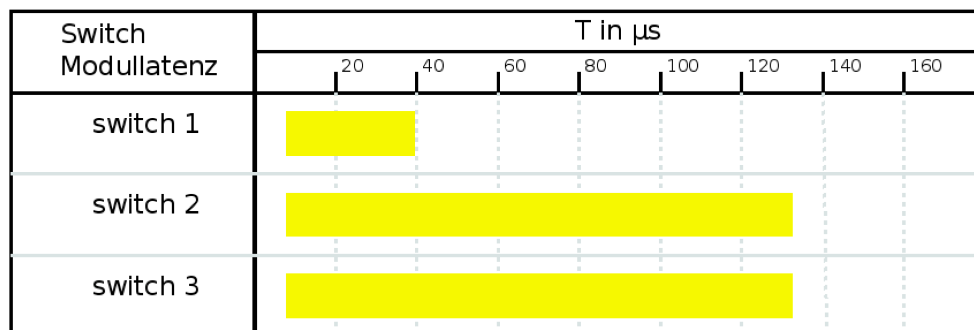


Abbildung 3.11.: Diagramm Ebene 1 - Switch

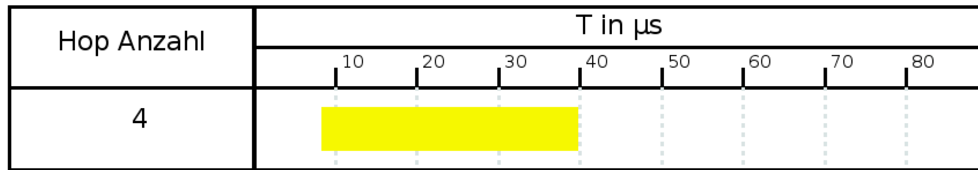


Abbildung 3.12.: Diagramm Ebene 2 - Switch

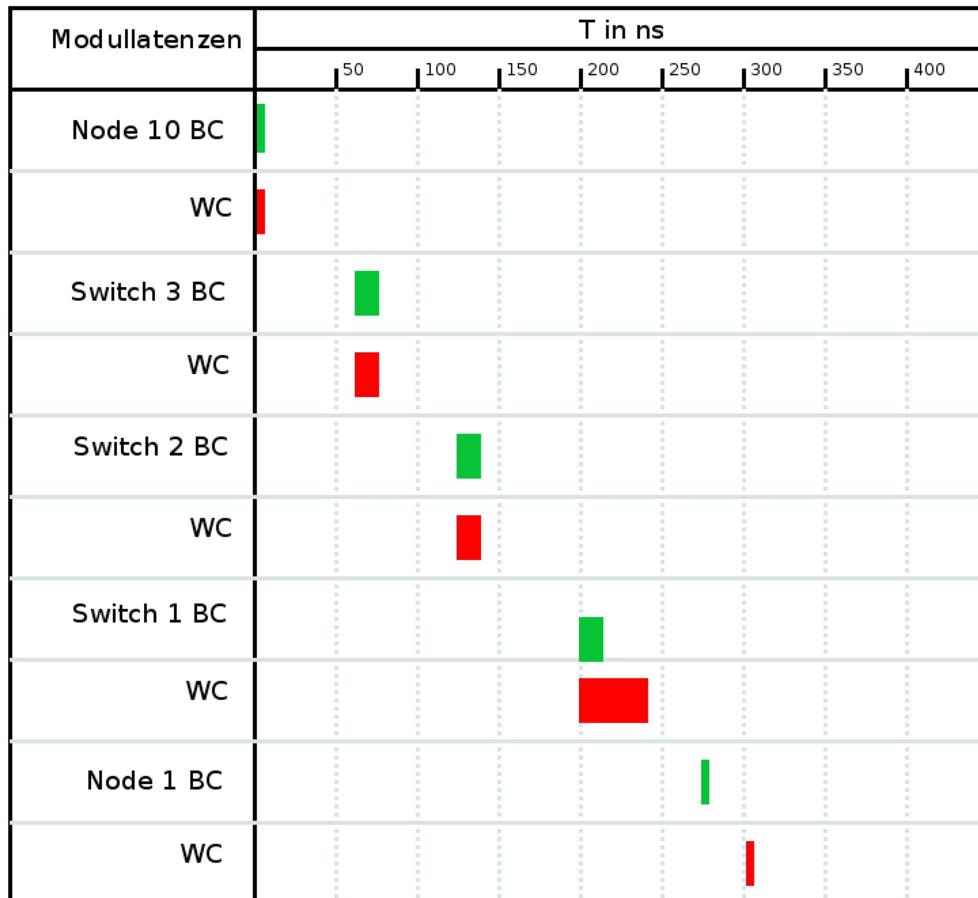


Abbildung 3.13.: Diagramm Ebene 2 - Switch

3.4. Datenanalyse

In diesem Abschnitt wird zuerst untersucht, welche Daten für die Erfassung der Zeiten aus der Analyse in Abschnitt 3.2 benötigt werden. Danach wird betrachtet, welche Methoden OMNeT++ bietet, um die Daten aus der Simulation zu sammeln. Am Ende wird die am besten

geeignete Variante ermittelt, welche dann für das Konzept und das Programm später genutzt werden soll.

3.4.1. Datenerfassung

OMNeT++ sammelt zu jedem Event, welches während der Simulation auftritt, eine Menge an Informationen. Es gilt also die Frage zu klären: "Welche Informationen oder Daten sind wichtig für die Diagramme?"
Die relevanten Daten für die Diagramme können in drei Gruppen zusammengefasst werden. Als erstes die Daten zur Identifizierung einer Nachricht, zweitens die Daten, um alle benötigten Zeiten (Sendezeitpunkt / Empfangszeitpunkt) für das Diagramm erfassen zu können und drittens die Topologie Informationen, damit die Anzahl der Hops und die Vollständigkeit eines Ende-zu-Ende Weges bestimmt werden kann. Damit eine Nachricht identifiziert werden kann, würde die von OMNeT++ zugeteilte ID ausreichen. Da der Nutzer durch diese ID nicht weiß, zu welcher Traffic-Klasse bzw. welchen Stream der Traffic-Klasse die ID gehört, müssen auch diese Daten gesammelt werden. Für die Topologie werden die Verbindungen zwischen den Switches und die Verbindungen von Module und Switch benötigt. Dafür ist die Nummer vom *phy* Module erforderlich. Somit kann der *phy* der Nachricht, mit dem *phy* der Topologieinformation verglichen werden und es können Aussagen über die Route getroffen werden. Weiterhin kann über die Topologieinformationen der aufgetretene Zeitstempel bei Multicast Nachrichten genau zugewiesen werden. Um die Zeiten für alle benötigten Diagramme zu erhalten, muss zu jedem Event der Zeitstempel gespeichert werden. Damit können alle erforderlichen Zeiten aus dem Abschnitt 3.2 berechnet werden. Zusammenfassend werden folgende Daten benötigt:

- Nachricht
  - Name der Traffic-Klasse und des Streams
  - OMNeT++ ID
  - Nummer des phys
  - Zeitstempel
- Topologie
  - Namen der Module
  - Verbindungen (Connections) zwischen den Modulen inkl. phy

3.4.2. Methoden zur Datenerfassung in OMNeT++

In OMNeT++ gibt es zwei Methoden, um Daten während der Simulation aufzuzeichnen, das Eventlog sowie Vector und Scalar Dateien. Darüber hinaus besteht die Möglichkeit, ein eigenes Eventlog zu erstellen. Für diesen Fall ist jedoch kein Mechanismus von OMNeT++ implementiert, aber es besteht die Option, direkt über den C++ Code vom Eventlog relevante Daten in eine selbst erstellte Datei zu schreiben. Um eine Auswahl zwischen den Methoden treffen zu können, wird im Folgenden jede Methode kurz beschrieben und ihre Vor- und Nachteile aufgezeigt. Im nächsten Kapitel wird dann eine Entscheidung getroffen, welche Methode am besten für das Programm geeignet ist.

Tabelle 6.1 zeigt zwei verschiedene Messungen. Die erste Messung schildert die benötigte Zeit für 10 sec Simulationszeit und die zweite Messung zeigt die Größe der resultierenden Dateien. Für die Messungen wurde das Netzwerk (AVB large network vgl. CoRE4INET) verwendet.

Aufzeichnungs Methode	Dauer bei 10 sec Simulationszeit	Dateigröße
ohne	03:16 min	-
Eventlog	09:18 min	6,1 GByte
Vector u. Scalar	03:16 min	651 Mbyte

Tabelle 3.6.: Messungen der Aufzeichnungs Methoden

Eventlog

Das Eventlog in OMNeT++ loggt alle aufgetretenen Events mit. Das sind zum Beispiel: Erstellen und Löschen von Modulen, Gates und Connections, Senden von Nachrichten etc. . Jedes Event wird mit einem Zeitstempel und einer laufenden Nummer versehen. Nach Abschluss der Simulation wird die *finish()* Methode aufgerufen. Diese Methode schließt das Eventlog ab und speichert eine .elog Datei im dafür vorgesehenen *results* Ordner. Die resultierende Datei ist eine zeilenorientierte Textdatei. Jede leere Zeile oder Zeilen beginnend mit # (Kommentare) werden ignoriert. Alle anderen Zeilen beginnen mit einem Identifier, welcher bekannt gibt, um welche Daten es sich im Folgenden handelt. Eine genaue Auflistung aller Identifier befindet sich im OMNeT++ Manual (vgl. Manual Omnet++ Community (b)). Es gibt die Möglichkeit, das Eventlog über Parameter in der .ini Datei an- bzw. auszustellen oder auf bestimmte Module zu beschränken sowie beliebige Start- und Endzeiten vorzugeben. Beim Testen einer Beschränkung auf bestimmte Module ist jedoch aufgefallen, dass die resultierende Datei größer und

die Simulationsgeschwindigkeit verringert war, im Gegensatz zum normalen Eventlog ohne Spezifikationen. Der Grund dafür konnte nicht herausgefunden werden.

Vorteile

Ein wesentlicher Vorteil von einem Eventlog ist, dass alle Informationen über den Verlauf der Simulation und über den Aufbau des Netzwerkes vorhanden sind. Es steht für jedes OMNeT++ Projekt automatisch zur Verfügung und durch die Menge an Informationen ist auch eine Ursachenfindung möglich.

Nachteile

Da das Eventlog jegliche Informationen während der Simulation aufzeichnet, steigt die resultierende Dateigröße stark an. Darüber hinaus erhöht sich auch die Simulationszeit drastisch. Ein Parsen, der für die Diagramme relevanten Daten, ist Aufgrund der Größe sehr zeitaufwendig. Wie in Tabelle 6.1 zu erkennen, hat sich die Zeit zum Simulieren mit eingeschaltetem Eventlog verdreifacht. Die entstandene Datei ist mit 6,1 GByte sehr groß. Bei längeren Simulationen kann schnell die Kapazität der Festplatte erreicht werden.

Vector und Scalar Dateien

Vectoren enthalten eine Menge von Werten, aufgezeichnet von den Modulen oder Kanälen. Diese Werte können zum Beispiel Ende-zu-Ende Latenzen oder Wartezeiten in den Queues sein. Scalare sind eine Zusammenfassung von Ergebnissen, welche während der Simulation berechnet werden und am Ende der Simulation in eine Scalar Datei übertragen werden. Die Werte werden numerisch gespeichert. Diese Zahlen sind das Ergebnis einer mathematischen Berechnung aus einer Menge an Werten, wie zum Beispiel Zähler, Minimum, Maximum oder Durchschnitt. Die Ergebnisse können über zwei Varianten aufgezeichnet werden. Zum einen über den Signal Mechanismus, welcher über *statistics* (vgl. Manual Omnet++ Community (b)) deklariert ist und zweitens direkt über den C++ Code, indem die Simulationsbibliothek verwendet wird. Beide Aufzeichnungsmethoden können ähnlich wie das Eventlog über die *.ini* Datei an- und ausgeschaltet sowie über bestimmte Parameter gefiltert werden.

Vorteile

Da alle Zeiten oder Werte in Vector und Scalar Dateien nacheinander aufgezeichnet sind, bildet es eine ideale Basis für das Parsen der Daten. Die Vector Zeiten stehen schon als fertige Ende-zu-Ende Latenzen bereit, wodurch beim Sammeln der erforderlichen Daten die Zeit für

das Berechnen entfällt. Die Simulationsgeschwindigkeit wird durch das Aufzeichnen nicht beeinträchtigt und auch die Dateigröße ist für die Menge an Informationen überschaubar.

Nachteile

Der größte Nachteil ist, dass nicht alle benötigten Daten aufgezeichnet werden. Es fehlen genaue Informationen zur Route, welche die Nachricht zurückgelegt hat und die genaue Zuordnung der Nachricht. Es gibt keine Details, von welcher Traffic-Klasse die Daten stammen. Ein weiterer Nachteil ist, dass keine Ursachenfindung über die geloggten Daten möglich ist. Bei Scalar Dateien müssen alle Ergebnisse, die berechnet werden sollen, vorher im Programmcode festgelegt werden. Das bedeutet, dass jede Änderung am Programmcode des Netzwerks eine Anpassung für die Scalar Dateien mit sich bringt.

Eigenes Eventlog

Ein selbst erstelltes Eventlog ist laut der Dokumentation von OMNeT++ nicht vorgesehen. Es besteht jedoch die Möglichkeit, durch Erweitern des C++ Codes vom Eventlog eine eigene Datei zu generieren. In der Eventlog Klasse sind für jedes Event, wie zum Beispiel Erstellen einer Verbindung zwischen zwei Gates oder das Senden einer Nachricht, einzelne Methoden implementiert. Diese Methoden sammeln die benötigten Daten und geben sie weiter zum rausschreiben in die Eventlog Datei. An dieser Stelle kann gezielt ein eigener Code hinzugefügt werden. Dadurch kann die resultierende Datei exakt auf die Anforderungen angepasst werden.

Vorteile

Ein entscheidender Vorteil ist, dass nur Daten mitgeloggt werden, welche wirklich benötigt werden. Dadurch wird die Dateigröße gering gehalten sowie die Simulationgeschwindigkeit erhöht. Ein weiterer Vorteil besteht darin, dass die Form der Ausgabedatei selbst bestimmt werden kann, wodurch die Performance beim Parsen der Daten erhöht wird.

Nachteile

Der Nachteil bei einem eigenen Eventlog liegt darin, dass die Basis für diese Art des Aufzeichnens erst programmiert werden muss. Es muss zum einen der C++ Code vom Eventlog angepasst werden und es muss eine Semantik für die Ausgabedatei festgelegt werden.

3.4.3. Zusammenfassung und Wahl der Methode

Aufgrund der Daten, die für die Diagramme benötigt werden, ist die Aufzeichnung über Vector und Scalar Dateien nicht möglich. Das OMNeT++ Eventlog und das eigene Eventlog liefern alle benötigten Daten. Beim selbst erstellten Eventlog entsteht ein Aufwand für die Implementierung, dafür bekommt man jedoch einen Zeitgewinn in der Simulationsgeschwindigkeit und die Dateigröße kann stark verringert werden. Dadurch ist auch ein Zeitgewinn bei der Zusammenstellung der Daten für die Diagramme zu erwarten. Aus diesen Gründen wird sich bei der Wahl der Methode zur Datenerfassung für das eigene Eventlog entschieden. Informationen zum Aufbau und zur Implementierung des eigenen Eventlogs befinden sich im Abschnitt 5.1.

3.5. Zusammenfassung der Analyse

Durch die Analyse konnte eine fundierte Grundlage für das Konzept erstellt werden. Es wurden die benötigten Zeiten definiert (vgl. Abschnitt 3.2). Wobei es sich um Latenzen handelt, welche Ende-zu-Ende oder im Modul gemessen werden. Dazu wird noch der resultierende Jitter berechnet. Durch die Diagrammanalyse (vgl. Abschnitt 3.3) wurde für alle benötigten Zeiten eine Semantik zur Darstellung im Diagramm festgelegt. Des Weiteren können durch die Ursachenfindungsanalyse (vgl. Abschnitt 3.3.3) nicht nur Informationen über die Latenzen und Jitter dargestellt werden, sondern es bietet eine Grundlage, um auf Fehlerstellen hinzuweisen. Da die Diagramme in verschiedene Ebenen unterteilt wurden, wird der Nutzer unterstütz bei der Wahl der relevanten Ende-zu-Ende Latenzen. Die abschließende Datenanalyse (vgl. Abschnitt 3.4) kam zu dem Ergebnis, dass ein eigenes Eventlog die ideale Lösung zur Erfassung der Daten darstellt. Es wurde weiterhin erarbeitet, welche Daten für die Diagramme relevant sind. Die Analyse war zusammenfassend erfolgreich, was bedeutet, dass alle Anforderungen erfüllt werden können.

4. Konzept

In diesem Kapitel wird ein Konzept erarbeitet, welches als Basis für das zu entwickelnde Programm dienen soll. Als erstes wird ein Konzept für das eigene Eventlog und die Struktur der Ausgabedatei festgelegt. Danach wird der Aufbau und die Struktur für das Programm entworfen. Wichtige Faktoren dabei sind das GUI Design, die Eclipse Plug-in Einstellungen, die Festlegung der benötigten Bibliotheken und extensionpoints sowie die Datenerfassung aus der eigenen Eventlog Datei. Als Basis für dieses Kapitel dienen die Analysen aus Kapitel 3.

4.1. OMNeT++ eigenes Eventlog

Das Eventlog von OMNeT++ wird über die Klassen *envirbase.h*, *eventlogfilemgr.h* und *eventlogwriter.h* erzeugt und mit Daten gefüllt. Diese Klassen befinden sich im OMNeT++ Projekt unter *omnetpp<version>/src/envir*. Während der Simulation werden aus den Modul-, Gate-, Simulation- und Messageklassen an bestimmten Stellen Methoden aus der *envirbase.h* Klasse aufgerufen. Diese Klasse gibt die Information weiter an die *evnetlogfilemgr.h* Klasse. Dort werden die benötigten Daten selektiert und weitergeleitet an die *eventlogwriter.h* Klasse. Diese schreibt dann die übermittelten Werte in das Eventlog. Öffnen und Schließen der Eventlogdateien befindet sich im *eventlogfilemgr.h*. Diese Klasse kann um die Funktion eines eigenen Eventlogs erweitert werden. In der *eventlogfilemgr.h* Klasse stellt jede Methode eine bestimmte aufgetretene Situation dar. Das kann zum Beispiel das Senden einer Nachricht, Löschen einer Nachricht, Herstellen einer Verbindung zwischen zwei Gates, Erstellen eines Modules etc. sein.

Wie in Abbildung 2.2 aus den Grundlagen zu entnehmen, kann für die Topologie die Erstellung einer Verbindung (engl. Connection) zwischen den Gates genommen werden. Die Funktion *connectionCreated(cGate *srcgate)* wird immer aufgerufen, wenn eine Verbindung zwischen zwei Gates hergestellt wird. Da es in einem Netzwerk und in einem Modul noch weitere Untermodule geben kann, muss die Auswahl auf Endknoten und Switches beschränkt werden. Für die Topologieinformationen wird der Name des Ziel Gates und des Gates der Quelle sowie die Nummer vom phy benötigt.

Die Informationen über die Nachricht können beim Senden der Nachricht abgefangen werden.

Dafür steht die Funktion *beginSend(cMessage *msg)* zur Verfügung. Dort gibt es die Möglichkeit, den Namen der Traffic-Klasse, den Namen des Streams, die ID und den Zeitstempel in die Eventlog Datei zu übertragen. Es ist auch wichtig, die Nummer des phys aufzuzeichnen, so kann im späteren Verlauf ein Abgleich mit den Daten zur Topologieinformation erfolgen, damit die richtigen Zeitstempel zum Berechnen genutzt werden. Auch an dieser Stelle muss nach Modulen gefiltert werden. Bei der Recherche über den Aufbau der Netzwerke ist bekannt geworden, dass eine Nachricht in einem App Modul erstellt und abgesendet bzw. empfangen im Mac Modul wird. Somit kann die Information auf diese Module beschränkt werden.
Eine weitere Aufgabe besteht darin, die Option herzustellen, dass das eigene Eventlog an- bzw. ausgestellt werden kann. Das reguläre Eventlog von OMNeT++ lässt sich über Parameter in der .ini Datei an- bzw. ausstellen. Es muss in der Implementierung dementsprechend eine Lösung gefunden werden, dass eigene Eventlog auch über Parameter in der .ini Datei zu steuern. Für das Eventlog von OMNeT++ wird die Dateiendung *.elog* genutzt. Für das eigene Eventlog wird die Dateiendung *.tlog* verwendet. Das selbst erstellte Eventlog wird auch text-orientiert zeilenweise beschrieben, wie es in dem OMNeT++ Eventlog der Fall ist. Die Syntax für die Ausgabedatei ist wie folgt:

<Datei> ::= <Zeile>*
<Zeile> ::= <Topologieinformation> | <Nachrichteninformation>
<Topologieinformation> ::= TOPO <Parameter>*
<Nachrichteninformation> ::= <Parameter>*
<Parameter> ::= <Attribut> <value>
<Attribut> ::= S | TC | ID | N | P | T | SM | SG | DG
<value> ::= text | integer | double

Attribut Liste

S Name des Streams
TC Name der Traffic-Klasse
ID OMNeT++ ID der Nachricht
N Name des Modules
P Nummer des phys
T Zeitstempel
SM Info ob das Event aus dem App oder Mac Modul kommt
SG Name des Source Gates
DG Name des Destination Gates

4.2. Programm

Wie bereits in der Einleitung erwähnt, soll das Konzept nach drei Faktoren erstellt werden. Beginnend mit den Plug-in Einstellungen und extensionpoints, folgt das GUI Design und am Ende die Strategie zur Datenerfassung.

4.2.1. Plug-in Einstellungen, Bibliotheken und extensionpoints

Für das Plug-in ist wichtig, wie es gestartet werden soll, ob zusätzliche Bibliotheken verwendet werden müssen und ob weitere extensionpoints benötigt werden. Eclipse enthält verschiedenen Optionen, womit das Plug-in gestartet werden kann. Es gibt die Möglichkeit, das Plug-in über die Menüleiste, die Toolbar oder über das Kontextmenü bei Auswahl mit der rechten Maustaste zu starten. Bei der Menüleiste oder der Toolbar muss zu Beginn über das Plug-in der Speicherort für die eigene Eventlog Datei bestimmt werden. Das führt zu einem erhöhten Aufwand für den Nutzer. Da die Aufzeichnungsmethoden wie zum Beispiel das Eventlog auch über die aufgezeichnete Datei direkt angesprochen werden und das selbst erstellte Eventlog als Grundlage für das Plug-in gilt, ist das Starten des Plug-ins über das Kontextmenü die am Besten geeignete Möglichkeit. Es muss also in der Implementierung festgelegt werden, dass bei einem Rechtsklick auf die Eventlog Datei das Programm gestartet werden kann. Dafür wird der extensionpoint *org.eclipse.ui.menus* verwendet. Als Ort für die Datei wird der Projekt Explorer gewählt, der bei OMNeT++ standardgemäß genutzt wird, um Dateien anzuzeigen. Weiterhin muss definiert werden, dass die Auswahl des Plug-ins nur möglich ist, wenn die Datei mit *.tlog* endet. Damit festgelegt werden kann, welche Klasse beim Start ausgeführt werden soll, wird zusätzlich der extensionpoint *org.eclipse.ui.commands* benötigt. Über diesen kann ein Handler bestimmt werden, welcher angibt, welche Klasse beim Starten genutzt werden soll.

Für die Programmierung von GUI Inhalten beinhaltet Eclipse die SWT Bibliothek. In dieser Bibliothek sind bis auf Diagramme alle benötigten GUI Funktionen vorhanden. Für die Erstellung von Diagrammen gibt es die SWTChart Bibliothek (vgl. SWTChart project Community), diese wird separat eingebunden und enthält verschiedene Funktionen zum Erstellen von Diagrammen. Eine Funktion für Gantt Charts ist jedoch nicht enthalten. Da Gantt Charts spezielle Balkendiagramme sind, können die Funktionen der Balkendiagramme aus der SWTChart Bibliothek genutzt werden. Sie müssen nur für die Diagrammtypen entsprechend angepasst werden.

Um den Pfad der Datei zu bekommen, wird die Bibliothek *org.eclipse.core.resources* benötigt. Diese Bibliothek stellt Funktionen bereit, um den Dateipfad der über einen Rechtsklick ausgewählten Datei zu bekommen. Für das Plug-in soll das MVC (Model View Controller) Pattern genutzt werden. Somit ist die Konsistenz zwischen den Daten und der Benutzerschnittstelle gesichert (vgl. Buschmann, 1998).

4.2.2. GUI Design

Für das GUI Design werden drei Ansichten benötigt. Die Startansicht zum Beginnen der Datensammlung aus der Eventlog Datei, eine Ansicht zur Auswahl der Traffic-Klasse, des Streams, der gewünschten optionalen maximalen Latenz und der gewählten Variante zur Anzeige der Diagramme sowie eine Diagrammansicht. Im Folgenden werden alle drei Ansichten beschrieben und als Design Skizze konzeptioniert.

Startansicht

Die Startansicht wird beim Starten des Plug-ins aufgerufen. Sie dient zum beginnen der Zusammenstellung aller benötigten Daten für die Diagramme. Der Aufbau ist in Abbildung 4.1 zu sehen. Das Textfeld dient für einen Einleitungstext und der Button zum Starten der Datenerfassung. Im unteren Bereich wird das Logo des Plug-ins angezeigt.

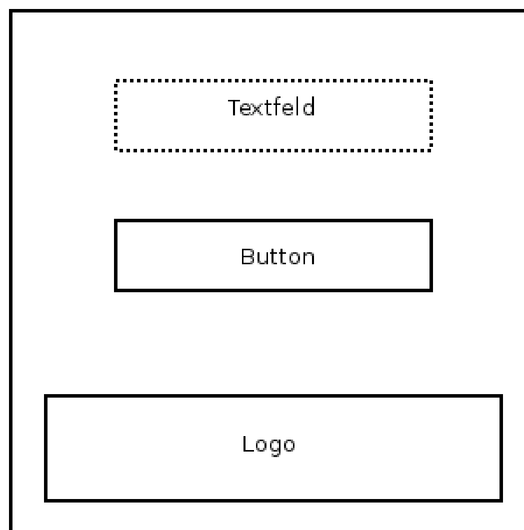


Abbildung 4.1.: GUI Skizze: Startansicht

Auswahlansicht

Über die Auswahlansicht kann der Nutzer festlegen, welche Zeiten im Diagramm dargestellt werden sollen. Dafür wird eine Auswahl der Traffic-Klassen (Auswahlreiter 1), der dazugehörigen Streams (Auswahlreiter 2), einer optionalen maximalen Latenz (Eingabe Fenster) sowie zwei Buttons zum Wählen der Diagrammansicht benötigt. Das resultierende GUI Design findet sich in Abbildung 4.2. Die Textfelder dienen zur Aufforderung einer Auswahl. Am Ende wird das Logo des Plug-ins angezeigt.

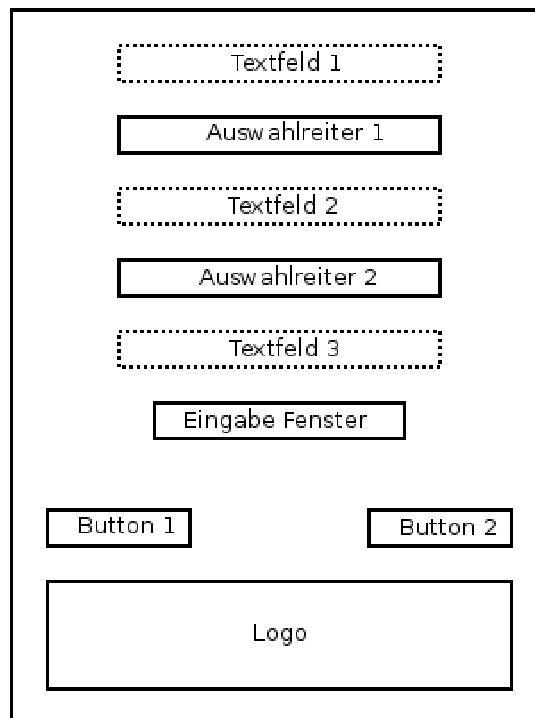


Abbildung 4.2.: GUI Skizze: Auswahlansicht

Diagrammansicht

In der Diagrammansicht müssen alle Informationen zu der getroffenen Auswahl vorhanden sein (Textfeld), das eigentliche Gantt Diagramm (Diagramm), eine Detailansicht mit Informationen zu allen Zeiten eines Balkens (Textfeld) und Buttons zum Zurückkehren zur Auswahl oder zur vorigen Diagrammebene. Zur nächsten Ebene gelangt man über einen Doppelklick auf den Balken. Ein Einfachllick auf einen Balken ändert die Detailinformationen zum Balken im Textfeld. Das GUI Design für die Diagramm Ansicht ist in Abbildung 4.3 zu sehen.

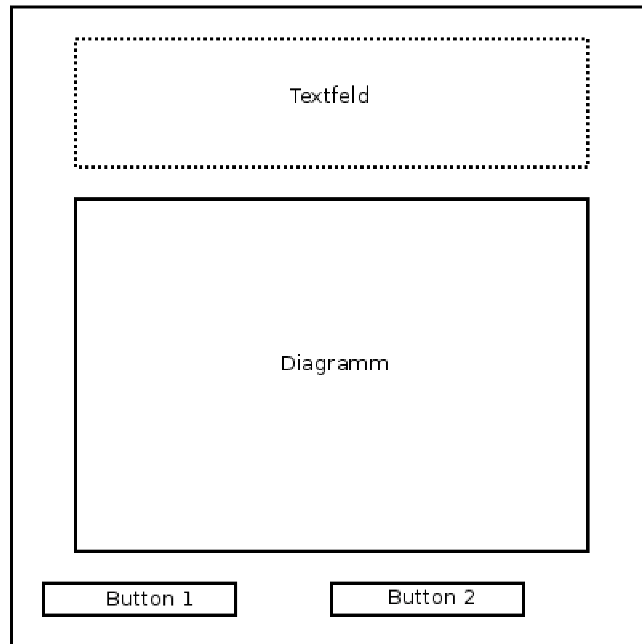


Abbildung 4.3.: GUI Skizze: Diagrammansicht

4.2.3. Datenerfassung

Relevant für die Datenerfassung ist, dass die Suche bei größeren Dateien sehr zeitaufwendig ist, der Algorithmus für die Suche sollte so funktionieren, dass das Eventlog nur einmal durchgegangen werden muss. Ein neues Durchsuchen für eine andere Auswahl einer Traffic-Klasse zum Beispiel würde die Suchzeit sonst verdoppeln. Die Weginformationen werden am Anfang gespeichert. Dafür müssen die Modulnamen und die Verbundenen Gates gespeichert werden. Da zum Zeitpunkt eines auftretenden Events von OMNeT++ nur der Ist-Zustand festgestellt werden kann, ist eine Aussage, zu welchem Modul oder von welchem Modul ein Frame kommt, nicht möglich. Jede Nachricht hat eine von OMNeT++ erstellte ID. Durch diese ID kann zwischen verschiedenen Nachrichten unterschieden werden. Die Route einer Nachricht wird durch jede folgende, identische ID erweitert und mit der Topologieinformation abgeglichen. Das ist notwendig, da die ID bei Multicast Nachrichten nicht geändert wird. Ist eine Nachricht an einem Endpunkt angelangt, werden die Zeiten für die verschiedenen Ebenen berechnet und gespeichert. Dadurch muss beim Wechseln zwischen den Ebenen kein erneutes Durchsuchen des gesamten Datensatzes erfolgen. Es werden dementsprechend Datentypen für Nachrichten, für Ergebnisse aller Ebenen, für Traffic-Klassen und für Streams benötigt.

5. Implementierung

Bei der Implementierung wird zu Beginn die Erstellung des eigenen Eventlogs beschrieben. Danach folgt eine Übersicht zum Eclipse Plug-in. Dafür wird unterschieden zwischen der Konfiguration des Plug-ins, der Programmarchitektur und der Implementierung der Model-View-Controller Klassen. Im Anhang der Arbeit befindet sich für das Plug-in noch ein englischsprachiges Manual für den Endnutzer. Dieses enthält alle relevanten Informationen zur Bedienung und Installation des entwickelten Plug-ins.

5.1. OMNeT++ eigenes Eventlog

In diesem Kapitel wird die Implementierung des eigenen Eventlogs beschrieben. Da OMNeT++ keine Funktionen für ein eigenes Eventlog bereitstellt, wurde der vorhandene C++ Code aus dem OMNeT++ Projekt erweitert. Wie im Konzept im Abschnitt 4.1 beschrieben, werden dazu die Dateien *eventlogfilemgr.cc* und *eventlogfilemgr.h* überarbeitet. Im Folgenden werden die Änderungen an beiden Dateien beschrieben. Um das eigene Eventlog später in OMNeT++ zu integrieren, genügt es, die Dateien auszutauschen und das Projekt neu zu bauen.

eventlogfilemgr.h

In der Header Datei werden zwei Variablen im private Scope hinzugefügt. Wie in Listing 5.1 zu erkennen, handelt es sich dabei um ein *FILE* für die Ausgabe Datei und einen *bool*, um die Erstellung der eigenen Eventlog Datei aktivieren bzw. deaktivieren zu können.

```
1 FILE *ftiminglog;
2 bool ganttTimingAnalyserEnabled;
```

Listing 5.1: Variablen im Header

eventlogfilemgr.cc

In der Source Datei wird das Rausschreiben der Daten in die Ausgabedatei, das Öffnen und Schließen der Datei sowie die Erkennung, ob ein eigenes Eventlog erstellt werden soll, anhand der *.ini* Datei implementiert. Das Öffnen und Schließen der Ausgabedatei wurde an den

gleichen Stellen vorgenommen, wie es beim regulären Eventlog der Fall ist. Das Öffnen in der Methode *open()* und das Schließen in der Methode *close()*.

Um festzulegen, ob das eigene Eventlog aktiviert werden soll, kann die Parameter Funktion von OMNeT++ nicht genutzt werden. Die OMNeT++ IDE erkennt die neuen Parameter nicht im Editor und es wird dann als Fehler angezeigt, was den Nutzer irritieren kann. Aus diesem Grund wurde die Option gewählt, die *.ini* Datei nach einem bestimmten Kommentar zu durchsuchen. Das Durchsuchen findet in der Methode *configure()* statt. In Listing 5.2 ist der Code Ausschnitt dafür zu finden. In Zeile 1 wird der Name der *.ini* Datei aus der Konfiguration geholt, da dieser beliebig sein kann. Danach wird die Datei Zeile für Zeile nach dem entsprechenden String durchsucht. Der String kann also in der Datei beliebig als Kommentar hinzugefügt werden.

```
1  const char *inifile = ev.getConfigEx()->getFileName();
2  ganttTimingAnalyserEnabled = false;
3  std::fstream f;
4  char cstring[256];
5  f.open(inifile, std::ios::in);
6  //Checking if Timing is enabled
7  while (!f.eof())
8  {
9      f.getline(cstring, sizeof(cstring));
10     if(strstr(cstring, "GanttTimingEnabled")){
11         ganttTimingAnalyserEnabled = true;
12     }
13 }
14 f.close();
```

Listing 5.2: Durchsuchen der .ini Datei und Setzen des Status

Das Schreiben der Daten in die Ausgabedatei erfolgt in zwei Methoden. In *beginSend(cMessage *msg)* werden die Daten zur Nachricht rausgeschrieben und in *connectionCreated(cGate *srcgate)* werden die Topologieinformationen in die eigene *.tlog* Datei übertragen. Da für die Topologieinformationen nur die Verbindungen der Gates eines Switches entscheidend sind, werden zuvor irrelevante Informationen herausgefiltert. Eine Filterung des Switches ist nur über den Namen möglich. Aus dem Grund muss für alle Projekte, welche das Plug-in nutzen, festgelegt werden, dass die Switch Module *switch* im Namen enthalten.
Für die Message Informationen muss zuvor zwischen dem Erstellen einer Nachricht oder dem Senden/Empfangen einer Nachricht in einem Gate auf der Route unterschieden werden. Der Code dafür ist in Listing 5.3 aufgeführt. Das Senden und Empfangen von Nachrichten findet im *mac* Modul statt und das Erstellen in einem *app* Modul. Somit kann über diese Namen

selektiert werden (siehe Zeile 4 und 26). Da sich die Hierarchieebenen der Module zwischen den CoRE4INET Framework und dem Signals and Gateway Framework unterscheiden und die Namen der Module auf unterschiedlichen Ebenen festgelegt sind, muss auch eine Unterscheidung zwischen diesen Frameworks stattfinden. Beim *mac* Modul ist dies möglich über das *tte* Modul, welches nur bei den mit dem Signals and Gateway Framework erstellten Netzwerken zum Einsatz kommt (siehe Zeile 7-9). Das Erstellen der Nachricht passiert in der *gatewayApp* (siehe Zeile 19) im Gateway Modul. Die gezeigten Listings der eventlogfilemgr.cc Datei gelten für OMNeT++ 4.6.

```
if (msg->isPacket() && ganttTimingAnalyserEnabled){
        cModule *mod = simulation.getContextModule();
        //Mac Module Arrive/Departure at Module
        if(mod && strcmp(mod->getFullName(), "mac") == 0)
        {
            cPacket *pkt = (cPacket *)msg;
            const char * name = mod->getParentModule()->getParentModule()->
                getFullName();
            // "tte" for Signals and Gateways
            if(strstr(name, "tte"))
            {
            fprintf(ftiminglog, "S %s TC %s ID %ld N %s P %s T %s SM \n", msg->
                getFullName(),msg->getClassName(), pkt->getEncapsulationTreeId(),
                 mod->getParentModule()->getParentModule()->getParentModule()->
                getFullName(), mod->getParentModule()->getFullName(), SIMTIME_STR(
                simulation.getSimTime()));
            }
            else
            {
             fprintf(ftiminglog, "S %s TC %s ID %ld N %s P %s T %s SM \n", msg->
                 getFullName(),msg->getClassName(), pkt->getEncapsulationTreeId(),
                   mod->getParentModule()->getParentModule()->getFullName(), mod->
                 getParentModule()->getFullName(), SIMTIME_STR(simulation.
                 getSimTime()));
            }
        }
        //Message arrives at Gateway
        if(mod && strstr(mod->getFullName(), "gatewayApp"))
        {
            cPacket *pkt = (cPacket *)msg;
            fprintf(ftiminglog, "S %s TC %s ID %ld N %s T %s SM startMsg\n", msg->
                getFullName(),msg->getClassName(), pkt->getEncapsulationTreeId(),
                mod->getParentModule()->getParentModule()->getFullName(),
                SIMTIME_STR(simulation.getSimTime()));

        }
      //Message is created
        else if(mod && (strstr(mod->getFullName(), "app") || strstr(mod->
            getFullName(), "App") || strstr(mod->getFullName(), "APP")))
        {
            cPacket *pkt = (cPacket *)msg;
            fprintf(ftiminglog, "S %s TC %s ID %ld N %s T %s SM startMsg\n", msg->
                getFullName(),msg->getClassName(), pkt->getEncapsulationTreeId(),
                mod->getParentModule() ? mod->getParentModule()->getFullName() :
                mod->getFullName(), SIMTIME_STR(simulation.getSimTime()));

        }
    }
```

Listing 5.3: Rausschreiben der Daten in die Eventlog Datei

5.2. Eclipse Plug-in

In diesem Kapitel wird die komplette Implementierung des Eclipse Plugins beschrieben. Zu Beginn gibt es einen Überblick zur Programmarchitektur. Danach wird genauer auf die Konfiguration des Plug-ins eingegangen und am Ende werden die Packages Model, View und Controller genauer erläutert.

5.2.1. Programm Architektur

Das entwickelte Plug-in besteht aus einem Projekt, welches folgende Struktur aufweist:

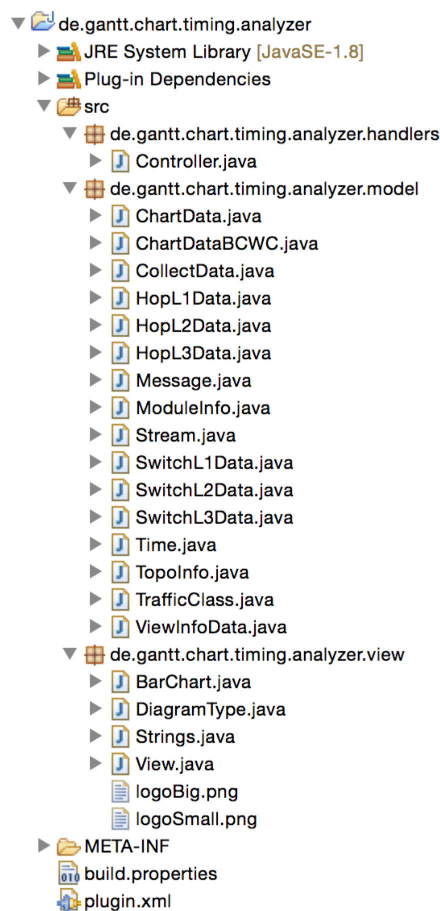


Abbildung 5.1.: Projekt Aufbau

Die Architektur nutzt das Model View Controller Pattern für eine sichere Konsistenz zwischen Daten und Benutzerschnittstelle, welches durch die Packages *de.gantt.chart.timing.model*, *de.gantt.chart.timing.view* und *de.gantt.chart.timing.handler* (Controller) repräsentiert wird. Die-

se Packages werden später genauer erläutert. Die Datei plugin.xml und der META-MF Ordner beinhaltet die Konfiguration des Plug-ins. Für die Darstellung der Klassen werden zur besseren Übersicht nur Ausschnitte aus dem Klassendiagramm gezeigt. Ein komplettes Klassendiagramm befindet sich im separaten Projektordner.

5.2.2. Konfiguration

Zur Konfiguration von Eclipse Plugins sind laut API zwei Dateien verantwortlich. Zum einen die Datei *MANIFEST.MF*, welche allgemeine Konfigurationen enthält und zum Anderen die plugin.xml Datei, welche optionale Konfigurationen zu extensions und extensionpoints zulässt. (vlg. Vogel und Milinkovich, 2015). In Listings 5.4 und 5.5 befinden sich Auszüge der implementierten plugin.xml Datei. Hier werden die genutzten zwei extensionpoints genauer beschrieben.

org.eclipse.ui.commands Über diesen extensionpoint kann das Plug-in dem Eclipse System Befehle zuordnen. Er wird genutzt, um den Namen für den Eintrag im Kontextmenü festzulegen (Listing 5.4 Zeile 11) und um einen Handler anzugeben. Über den Handler wird der Startpunkt des Plug-ins festgelegt. In diesem Fall soll die Controller Klasse als Einstiegspunkt genutzt werden. Die *execute()* Methode des Controllers wird dadurch automatisch aufgerufen. Da es bei diesem Plug-in keinen weiteren Einstiegspunkt gibt, genügt die Angabe des defaultHandlers (Listing 5.4 Zeile 9).

```
1  <extension
2          point="org.eclipse.ui.commands">
3       <category
4             id="de.gantt.chart.timing.analyzer.commands.category"
5             name="Sample Category">
6       </category>
7       <command
8             categoryId="de.gantt.chart.timing.analyzer.commands.category"
9             defaultHandler="de.gantt.chart.timing.analyzer.handlers.
                  Controller"
10            id="de.gantt.chart.timing.analyzer.commands.sampleCommand"
11            name="Gantt Chart Timing Analyzer">
12      </command>
13   </extension>
```

Listing 5.4: plugin.xml - command extensionpoint

org.eclipse.ui.menus Dieser extensionpoint wird genutzt, um bei der Auswahl mit der rechten Maustaste, einen Eintrag im Kontextmenü zu erhalten, über diesen dann das Programm gestartet werden kann. OMNeT++ nutzt den ProjectExplorer zum Anzeigen der

Projekte und Dateien. Über die locationURI (Listing 5.5 Zeile 4) wird dem System mitgeteilt, dass bei einem Rechtsklick im ProjektExplorer ein Befehl ausgelöst werden soll. Dieser Befehl veranlasst einen Menüpunkt im Kontextmenü, dieser ist in (Listing 5.5 Zeilen 6-18) genauer definiert. Da dies nur bei *.tlog* Dateien geschehen soll, wurde festgelegt, dass der Befehl nur ausgeführt wird, wenn es sich um diese Dateiendung handelt (Listing 5.5 Zeile 8-17). Eclipse stellt zum Anzeigen der Dateien noch den Package Explorer und den Navigator zur Verfügung. Diese werden im *org.eclipse.ui.menus* extensionpoint ebenfalls deklariert. Der Code dafür ist, bis auf die entsprechende *locationuri*, equivalent zu Listing 5.5.

```
1  <extension
2           point="org.eclipse.ui.menus">
3        <menuContribution
4       locationURI="popup:org.eclipse.ui.navigator.ProjectExplorer#PopupMenu">
5              <command
6                    commandId="de.gantt.chart.timing.analyzer.commands.sampleCommand"
7                    id="de.gantt.chart.timing.analyzer.menus.sampleCommand">
8                      <visibleWhen>
9                     <with variable="activeMenuSelection">
10                     <iterate
11                          ifEmpty="false">
12                      <adapt type="org.eclipse.core.resources.IResource">
13                        <test property="org.eclipse.core.resources.name" value="*.
                              tlog" />
14                      </adapt>
15                     </iterate>
16                    </with>
17               </visibleWhen>
18             </command>
19       </menuContribution>
20 </extension>
```

Listing 5.5: plugin.xml - menu extensionpoint

In der Manifest Datei sind Informationen zu den benutzten Bibliotheken, der Versionsnummer, der benötigten Java Version und der Name des Plugins hinterlegt. Die komplette MANIFEST.MF Datei ist in Abbildung 5.2 zu sehen. Die Bundle Versionen für die Bibliotheken (Zeile 7-9) wurden manuell gesetzt, da es sonst zu Inkompatibilität mit der OMNeT++ Version 4.6 auf dem Mac/OS kommt. Das liegt daran, dass die genutzten Bibliotheken in der OMNeT++ Version 4.6 , für Mac/OS eine niedrigere Versionsnummer besitzen. Wenn die erforderliche Versionsnummer einer Bibliothek höher ist als die vorhandene, kann das Plug-in nicht aktiviert werden, beim Start der IDE.

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: Gantt Chart Timing Analyzer
4 Bundle-SymbolicName: de.gantt.chart.timing.analyzer;singleton:=true
5 Bundle-Version: 1.0.10.qualifier
6 Require-Bundle: org.eclipse.ui;bundle-version="3.106.0",
7  org.eclipse.core.runtime;bundle-version="3.10.0",
8  org.eclipse.core.resources;bundle-version="3.8.0"
9 Bundle-RequiredExecutionEnvironment: JavaSE-1.8
10 Bundle-ActivationPolicy: lazy
11 Import-Package: org.swtchart
```

Abbildung 5.2.: MANIFEST.MF Datei

5.2.3. Controller

```
Controller
<<Property>> -sNumber : int
<<Property>> -SName : String
<<Property>> -maxLatency : int
<<Property>> -time : String
-data : CollectData
-view : View
+Controller()
+execute(event : ExecutionEvent) : Object
+setHopL1View(trafficClass : String, stream : String) : void
+setSwitchL1View(trafficClass : String, stream : String) : void
+setHopL2View(trafficClass : String, stream : String, number : int) : void
+setSwitchL2View(trafficClass : String, stream : String, sw : String) : void
+setHopL3L4View(trafficClass : String, stream : String, start : String, end : String, type : DiagramType) : void
+setSwitchL3L4View(trafficClass : String, stream : String, switchName : String, switchNumber : int, type : DiagramType) : void
+searchingData() : void
+backToSelectionWindow() : void
-setDefaults(val : double []) : double []
-setDefaults(val : String []) : String []
-getCategories(number : int []) : String []
+time() : String
```

Abbildung 5.3.: Controller.java Klasse

Der Controller startet die Datensammlung aus dem Model, wartet auf Eingaben vom Nutzer und gibt daraufhin die Daten an die View weiter. Der Controller leitet den Programmstart ein über die *execute()* Methode. Es wird eine Shell und eine View erzeugt. Der View wird mitgeteilt, dass die Startansicht (vgl. Abschnitt 4.2.2) angezeigt werden soll. Die setter Methoden (vgl. Listing 5.9) erhalten von der View die eingegebenen Daten, fordern den Datensatz für das angeforderte Diagramm aus dem Model an, rechnen die Daten für die Diagramme (vgl. Abschnitt 5.2.5) um, damit Sie von der View angezeigt werden können und geben dann die Daten an die View weiter.

```
1 public void setLinkL1View(String trafficClass, String stream)
```

```
2 public void setSwitchL1View(String trafficClass, String msgClass)
3 public void setLinkL2View(String trafficClass, String stream, int number)
4 public void setSwitchL2View(String trafficClass, String stream, String sw)
5 public void setLinkL3L4View(String trafficClass, String stream, String start,
      String end, DiagramType type)
6 public void setSwitchL3L4View(String trafficClass, String stream, String switchName
      , int switchNumber, DiagramType type)
```

Listing 5.6: setter Methoden der Controller.java Klasse

Die Funktion searchingData() ermittelt den Dateipfad und startet die Datensammlung aus der eigenen Eventlog Datei im Model. Für die Ermittlung des Dateipfades werden die Bibliotheken *org.eclipse.ui*, *org.eclipse.core.resources* und *org.eclipse.core.runtime* benötigt.

```
1  // get workbench window
2  IWorkbenchWindow window = PlatformUI.getWorkbench().getActiveWorkbenchWindow();
3  // set selection service
4  ISelectionService service = window.getSelectionService();
5  // set structured selection
6  IStructuredSelection structured = (IStructuredSelection) service.getSelection();
7  if (structured.getFirstElement() instanceof IFile)
8  {
9       // get the selected file
10      IFile file = (IFile) structured.getFirstElement();
11      // get the path
12      IPath path = file.getLocation();
13 }
```

Listing 5.7: Auszug der Funktion searchingData()

Listing 5.7 zeigt den Ausschnitt welcher zur Ermittlung des Dateipfades dient. In Zeile 2 erhält man vom System ein *IWorkbenchWindow*, welches alle Informationen über die Eclipse Oberfläche beinhaltet. Damit der Pfad der angeklickten Datei herausgefunden werden kann, muss ein *ISelectionService* angelegt werden. Über diesen Service wird in Zeile 6 das Objekt, welches angeklickt wurde, ermittelt. In Zeile 8 wird überprüft, ob es sich um eine Datei handelt und dann wird der Pfad der Datei gespeichert. Dieser Pfad wird dann im späteren Programmverlauf mit *toPortableString()* in einen String umgewandelt und an das Model weitergereicht.

Nachdem die Suche abgeschlossen ist, wird die View aufgefordert, die Auswahlansicht (vgl. Konzept Abschnitt 4.2.2) anzuzeigen. Die Funktionen aus Listing 5.8 dienen als Helferfunktionen für Operationen, wie beispielsweise das Umwandeln eines Integer Arrays in ein String Array.

```
1 private double[] setDefaults(double[] val)
2 private String[] setDefaults(String[] val)
```

```
3 private String[] getCategories(int[] number)
```

Listing 5.8: Helferfunktionen der Controller.java Klasse

Die Funktion *backToSelectionWindow()* wird von der View aufgerufen um zurück zur Auswahlansicht zu wechseln.

5.2.4. Model

Die Klassen aus dem Package *de.gantt.chart.timing.model* sorgen für das Laden der Daten aus der *.tlog* Datei in geeignete Datenstrukturen für die Diagramme. Es werden in dem Model alle benötigten Zeiten berechnet und es bietet Datenstrukturen für die Daten, welche die View zum Anzeigen benötigt. Die Logik zum Durchsuchen der eigenen Eventlog Datei, bestimmen der Route und abspeichern in die geeigneten Datenstrukturen, findet in den Klassen *CollectData.java*, *TrafficClass.java* und *Message.java* statt. Für die Berechnung der Latenzen dient die Klasse *Time.java*. Auf diese Klassen wird im Folgenden genauer eingegangen, alle anderen unter Abbildung 5.3 gelisteten Klassen dienen als Datenstrukturen für die Diagramme und zum Anzeigen der Informationen. In den Datenstrukturen für Diagramme werden alle Daten für die erste und zweite Ebene beider Diagrammvarianten gespeichert. Die Daten für die dritte und vierte Ebene werden direkt vor dem Anzeigen berechnet, da für diesen Fall nur noch eine geringe Anzahl von Nachrichten zur Verfügung stehen. Aus dem Grund kann die Größe der gespeicherten Daten minimiert und die Performanz der Suche gesteigert werden. In den Klassen *ChartData.java* und *ChartDataBCWC.java* werden die Daten für die Diagramme gespeichert und für die Darstellung von gestapelten Balkendiagrammen (siehe Abschnitt 5.2.5 entsprechend umgerechnet.

CollectedData.java

```
CollectData
-path : String
-trafficClasses : Map<String, TrafficClass> = new HashMap<String, TrafficClass>()
-multi : double
-topo : TopoInfo = new ArrayList<TopoInfo>()
+CollectData(path : String)
+startCollectingData(control : Controller) : void
-saveCollectedData(trafficClass : String, msgClass : String, id : int, module : String, time : double, startMsg : boolean, phy : int) : void
+streamNames() : Map<String, Set<String>>
+linkL1Data(trafficClass : String, msg : String) : Map<Integer, LinkL1Data>
+switchL1Data(trafficClass : String, msg : String) : Map<String, SwitchL1Data>
+linkL2Data(trafficClass : String, msg : String, linkNumber : int) : List<LinkL2Data>
+switchL2Data(trafficClass : String, msg : String, sw : String) : Map<Integer, SwitchL2Data>
+linkL3Data(start : String, end : String, trafficClass : String, stream : String) : List<LinkL3Data>
+switchL3Data(trafficClass : String, stream : String, switchName : String, hopCnt : int) : SwitchL3Data
+linkL4Data(start : String, end : String, trafficClass : String, msgClass : String) : List<LinkL3Data>
+switchL4Data(trafficClass : String, stream : String, switchName : String, hopCnt : int) : SwitchL3Data
+topo() : List<TopoInfo>
-deleteWrongConnections() : void
```

Abbildung 5.4.: CollectionData.java Klasse

Die Klasse CollectionData.java durchsucht die über *path* angegebene Ausgabedatei. Der Controller startet nach Betätigung des Buttons in der View über *startCollectingData()* die Suche. In CollectedData sind alle Traffic-Klassen und die Topologieinformationen hinterlegt. Somit hat sie Zugang auf alle ermittelten Zeiten. Über die **Data()* Methoden kann der Controller die benötigten Zeiten für die Diagramme anfordern. Nach dem Start der Suche wird die *.tlog* Datei über einen BufferedReader Zeile für Zeile eingelesen. Die zu Beginn aufgeführten Topologieinformationen werden in der *topo* ArrayList gespeichert. Die Informationen über die Nachrichten werden über *saveCollectedData(...)* weitergeleitet an die Traffic-Klasse. Nach Beendigung der Suche werden über *deleteWrongConnections()* alle Streams oder Traffic-Klassen, die keine vollständigen Nachrichten besitzen, gelöscht.

TrafficClass.java

```
TrafficClass
-hopMap : Map<Integer, HopL1Data>
-switchMap : Map<String, SwitchL1Data>
-streams : Map<String, Stream> = new HashMap<String, Stream>()
+TrafficClass(name : String)
+addMsgData(startMsg : boolean, modInfo : ModuleInfo, id : int, stream : String, module : String, topo : List<TopoInfo>) : void
+updateHopData(hopCount : int, modList : List<ModuleInfo>, start : String, end : String) : void
+updateSwitchData(modList : List<ModuleInfo>, start : String, end : String) : void
+hopMap() : Map<Integer, HopL1Data>
+streamMap() : Map<String, Stream>
+getAllMsg() : Map<Integer, Message>
+switchMap() : Map<String, SwitchL1Data>
```

Abbildung 5.5.: TrafficClass.java Klasse

Die TrafficClass.java Klasse besitzt Informationen über alle zugehörigen Streams, speichert Diagramm Daten für die allgemeine Auswahl aller Streams ab und erstellt die Messages für die Streams. In der Methode *addMsgData()* werden nicht vorhandene Messages neu hinzugefügt und im Fall, dass die Message existiert, werden die ersten zwei Modul Informationen gesetzt. Eine genaue Information zum Algorithmus für die Wegfindung befindet sich im folgenden Abschnitt.

Message.java

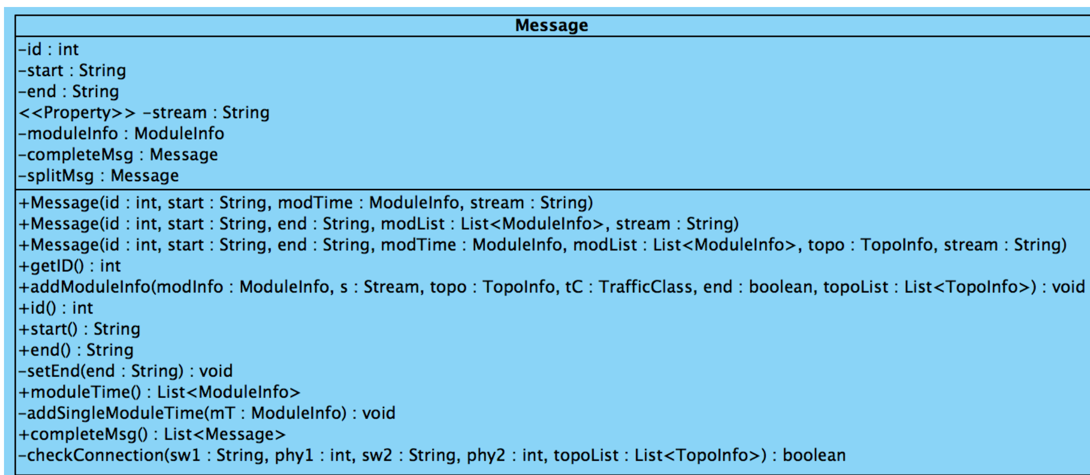


Abbildung 5.6.: Message.java Klasse

Die Klasse Message.java sammelt alle Informationen zu einer Nachricht, sie sorgt für die richtige Erkennung der Route und startet bei Ankunft im Ziel Modul, das Speichern und Berechnen aller Latenzen. Da eine Nachricht auch über Multicast gesendet werden kann und sich die OMNeT++ ID für diesen Fall nicht ändert, muss sie alle Zwischenrouten speichern. Die Erkennung, ob es sich um eine Multicast Nachricht handelt und das richtige Zuordnen der Simulationszeiten, geschieht beim Hinzufügen einer neuen Modulinformation in *addModuleInfo(...)*. Das Nassi Schneidermann Diagramm in Abbildung 5.8 zeigt den Ablauf der Zuordnungen.

Eine Modulinformation beinhaltet den Namen des Modules, seinen phy, die Simulationszeit und später die Latenz zum vorherigen Modul. Die neuen Nachrichten, die nach dem Prinzip des Nassi Schneidermann Diagrammes erstellt werden müssen, werden in der gleichen Message, in dem Array *completeMsg* gespeichert. Für einen besonderen Fall müssen auch noch *splitMsg-Nachrichten* angelegt werden. Diese werden benötigt, wenn eine Multicast Nachricht

in zwei verschiedene Richtungen verläuft, über mehr als einen Switch. Erkannt wird dieser Fall in der Abfrage, ob die Modulinformation eine Switch Verbindung ist. Ist dies eingetreten und die vorherige Modul Information ist auch eine Switch Verbindung, müssen für den weiteren Verlauf *spltiMsg-Nachrichten* angelegt werden, die die Informationen der Teilstrecken weiter sammeln. Daher wird zu Beginn der Methode geprüft, ob bereits *spltiMsg-Nachrichten* vorhanden sind und daraufhin wird mit der passenden Nachricht fortgefahren.

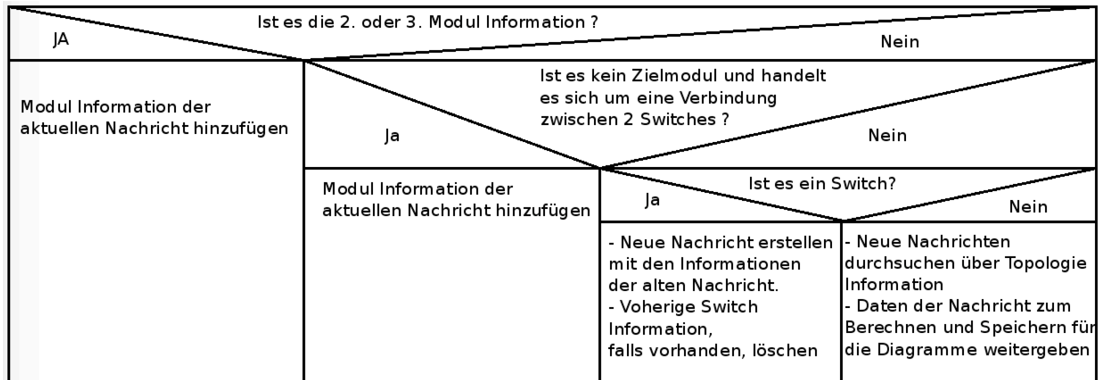


Abbildung 5.7.: Nassi Schneidermann Diagramm - Ablauf Nachrichtenerkennung

Ist die Nachricht in einem Ziel Modul angekommen werden die Informationen weitergeben an die zugehörige Traffic-Klasse und den zugehörigen Stream. Welche dann die Daten für die Diagramme sammeln und alle benötigten Latenzen berechnen.

Time.java

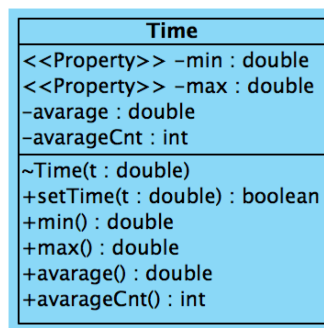


Abbildung 5.8.: Time.java Klasse

Die Time.java Klasse dient als Datenstruktur für Latenzinformationen. Sie berechnet das Minimum, das Maximum und den Durchschnitt und wird von jeder Datenstruktur der Diagramme implementiert. Die Berechnung erfolg in der Funktion *setTime(double t)*. Immer wenn

eine Nachricht eine neue Ende-zu-Ende Latenz hinzufügt, wird über die Funktion der Durchschnitt berechnet und eine Entscheidung getroffen, ob sie gegenüber früheren Latenzen das Minimum oder Maximum darstellt. Sie gibt einen boolean zurück, wodurch festgestellt werden kann, ob es sich bei der hinzugefügten Latenz um ein neues Minimum oder Maximum handelt.

```
1  public boolean setTime(double t)
2  {
3    boolean isChanged = false;
4    if(this.min > t)
5    {
6        this.min = t;
7          isChanged = true;
8      }
9      else if(this.max < t)
10     {
11         this.max = t;
12         isChanged = true;
13     }
14     this.avarage = ((this.avarage*this.avarageCnt) + t)/(this.avarageCnt+1);
15     this.avarageCnt++;
16     return isChanged;
17 }
```

Listing 5.9: Funktion zum Berechnen von Minimum, Maximum und Durchschnitt

5.2.5. View

```
View
-shell : Shell
-rowLay : RowLayout
-startSearchingBtn : Button
-label : Label
-comp : ScrolledComposite
-headline : Label
-logo : Label
-errorLabel : Label
-selectionGrp : Group
-image : Image
-width : int
-infoText : Label
-control : Controller
+View(control : Controller)
+showStartView() : void
+showSelectionView(items : Map<String, Set<String>>) : void
+showChartView(viewIData : ViewInfoData, values : ChartData, valuesBCWC : ChartDataBCWC, names : String [], type : DiagramType, maxLatency : int, l3TabInfos : String []) : void
-checkStringLength(str : String []) : String []

BarChart
-chart : Chart
-comp : ScrolledComposite
-infoText : Label
-infoTextData : String[]
-con : Controller
+BarChart(comp : ScrolledComposite, con : Controller, infotext : Label, infoTextData : String [])
+drawBarChartStrings(viewIData : ViewInfoData, values : ChartData, names : String [], shortNames : String [], type : DiagramType, maxLatency : int) : Chart
+drawBarChartLevel3(values : ChartDataBCWC, names : String [], shortNames : String []) : Chart
-getNullArray(val : double []) : double []
```

Abbildung 5.9.: View.java / Barchart.java Klassen

Im Package *de.gantt.chart.timing.view* befinden sich vier Klassen *View.java*, *BarCart.java*, *Strings.java* und *DiagramType.java. Strings.java* ist eine final Klasse, die alle Strings der GUI Elemente als Konstanten beinhaltet. Somit kann das Ändern für Strings der GUI Elemente erleichtert werden. *DiagramType.java* ist eine Enum Klasse, welche alle Diagrammtypen als Enum beinhaltet. In der *View.java* Klasse wird sie verwendet um zu unterscheiden, welches Diagramm erzeugt werden soll. Die Klasse *View.java* erzeugt die Oberfläche und die *BarChart.java* Klasse erstellt das Diagramm über die SWTChart (vlg. SWTChart project Community) Bibliothek. Auf die Klasse BarChart.java wird zum besseren Verständnis im nächsten Abschnitt detailliert eingegangen. Für die Erzeugung der drei Ansichten aus dem Konzept 4.2.2, stehen die Methoden *showStartView(...)*, *showSelectionView(...)* und *showChartView(...)* zur Verfügung (siehe Abbildung 5.9). In den Methoden wird das jeweilige Layout festgelegt, alle GUI Elemente erzeugt und platziert sowie Listener für die Buttons implementiert. Mit der Methode *checkStringLength(...)* wird die Länge der Strings, welche zum Anzeigen der Werte in der y-Achse beim Diagramm genutzt werden, geprüft und gegebenenfalls gekürzt. Die Methoden *createInfoText(..)* und *createInfoTextL3* stellen die Infotexte für die Labels zusammen, über die dann genaue Informationen zu den gewählten Balken angezeigt werden.

BarChart.java

Die Klasse BarChart.java ist für das Erstellen der Diagramme verantwortlich. Für die Diagramme wurde die SWTChart Bibliothek genutzt. Da diese Bibliothek nicht in der Eclipse PDE enthalten ist, muss sie separat eingebunden werden. Dazu werden die Projekte in den ProjectExplorer importiert und über das Manifest zugeordnet. Danach können alle Funktionen der Bibliothek für das Erstellen der Diagramme genutzt werden. Gantt Charts sind in der SWTChart Bibliothek nicht implementiert. Es gibt jedoch die Möglichkeit, Balkendiagramme zu stapeln. Daher müssen für ein Ereignis im Gantt Chart mehrere Balken erzeugt werden, welche dann gestapelt werden.

Als Beispiel ist in Abbildung 5.10 eine Gruppen Latenz zu sehen. Für diesen Fall wurden fünf Balkendiagramme erzeugt und dann gestapelt. Auf der linken Seite sind alle fünf Balken nebeneinander zu sehen und rechts gestapelt. Eine Einschränkung, die dadurch entsteht ist, dass das Gantt Chart immer bei null beginnen muss. Damit aus den Zeiten im Model die gewünschten Balken entstehen können, werden die Zeiten im Controller bereits umgerechnet. Das Diagramm wird in einem ScrolledComposite Objekt erstellt. Dieses wurde in der View.java erzeugt und dient dort als Container für das Diagramm. Beim Erstellen der Diagramme werden Listener für einen Einfach- und einen Doppelklick auf einen Balken implementiert. Durch

einen einfachen Klick werden die Werte auf dem Info Label mit den Daten des geklickten Balkens aktualisiert, über einen Doppelklick wird der Controller informiert, um welchen Balken es sich handelt, damit die Diagrammansicht gewechselt werden kann.

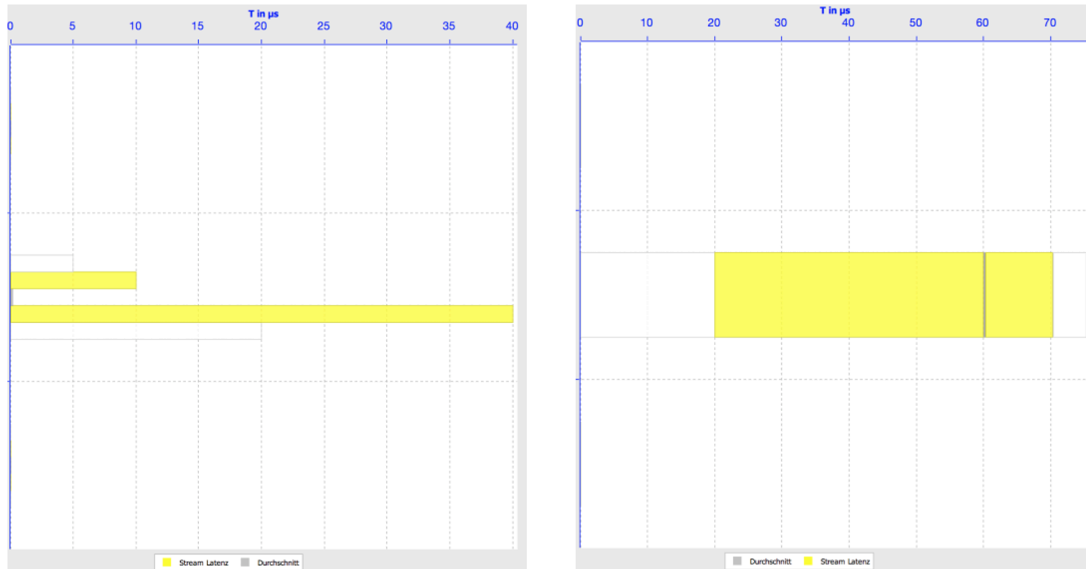


Abbildung 5.10.: Balken Diagramm als Gantt Chart

6. Evaluierung

In diesem Kapitel werden zuerst die Kriterien festgelegt, nach denen evaluiert wird. Danach werden die ermittelten Kriterien untersucht.

6.1. Evaluierungskriterien

Auf der Seite von OMNeT++ muss untersucht werden, ob die eigene Eventlog Datei eine gute Performanz aufweist und ob die entstandenen Dateien in ihrer Größe geringer sind als die vom ursprünglichen Eventlog. Des Weiteren muss geprüft werden, ob sich das Plug-in auf allen unterstützten Betriebssystemen installieren lässt. Auf der Seite von dem Plug-in soll die Dauer der Suche gemessen werden und die Funktion soll anhand von vorhandenen Projekten getestet werden. Für den Test sind folgende Fragen relevant. Damit das Programm erfolgreich genutzt werden kann, müssen alle Fragen mit ja beantwortet werden.

- Wurde das eigene Eventlog erfolgreich erstellt?
- Konnte das Plug-in gestartet werden?
- Wurden alle Traffic-Klassen und Streams erkannt?
- Werden alle Diagramme richtig dargestellt?
- Sind die Latenzen und Jitter bei dem Wechsel zur nächsten Ebene wiederzufinden?

6.2. OMNeT++

In der OMNeT++ Simulation wird die Simulationszeit und die resultierende Dateigröße gemessen. Am Ende wird getestet, ob das Plug-in auf allen unterstützten Betriebssystemen aktiviert werden kann. Als Projekt für die Messungen wurde das (AVB large network vgl. CoRE4INET) Projekt genutzt. Die OMNeT++ Version ist 4.6. Als System wird ein PC mit folgender Ausstattung genutzt.

- Betriebssystem: Windows 10
- Prozessor: AMD FX-8150 3,6Ghz
- Arbeitspeicher: 16 GB RAM 1333 MHz
- Festplatte: SSD 256 GB

Bei dem OMNeT++ Eventlog wurden keine Ausnahme Parameter gesetzt. Bei dem eigenen Eventlog werden die Module-Recordings über die .ini Datei mit dem Befehl ***.module-eventlog-recording = false* ausgeschaltet, damit nur das eigene Eventlog aufgezeichnet wird. Es werden für die Messungen Simulationszeiten von 5s, 10s, 15s und 20s gemessen.

6.2.1. Simulationsgeschwindigkeit

In Tabelle 6.1 befinden sich die gemessenen Daten. Es ist gut zu erkennen, dass die Simulationsgeschwindigkeit durch das eigene Eventlog nicht stark beeinträchtigt wird. Somit ist der Nachteil des Eventlogs, einer langsamen Simulationsgeschwindkeit, beim eigenen Eventlog nicht feststellbar. In Abbildung 6.1 sind die Messwerte dargestellt. Über die Darstellung lässt sich der Geschwindigkeitsvorteil gegenüber dem OMNeT++ Eventlog gut erkennen.

Aufzeichnungsmethode	5 sec	10 sec	15 sec	20 sec
Eventlog OMNeT++	05:02 Min	10:16 Min	15:24 Min	20:14 Min
eigenes Eventlog	02:07 Min	04:18 Min	06:23 Min	08:39 Min
ohne Methode	01:35 Min	03:11 Min	04:51 Min	06:25 Min

Tabelle 6.1.: Simulationsgeschwindigkeiten der Aufzeichnungsmethoden

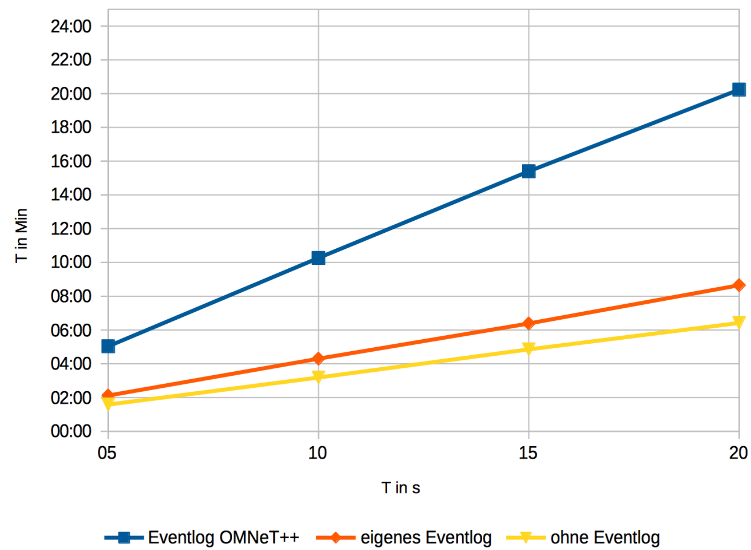


Abbildung 6.1.: Messungen Simulationsgeschwindigkeit

6.2.2. Dateigröße

Die Dateigröße vom eigenen Eventlog ist im Gegensatz zu dem Eventlog von OMNeT++ um rund 96% reduziert worden. Somit entstehen auch bei längeren Simulationen keine kritischen Datenmengen. Die gemessenen Daten sind in Tabelle 6.2 dokumentiert.

Aufzeichnungsmethode	5 sec	10 sec	15 sec	20 sec
Eventlog OMNeT++	1,9 GByte	3,8 GByte	5,7 GByte	7,7 GByte
eigenes Eventlog	79,6 MByte	159 MByte	241,9 MByte	324 MByte

Tabelle 6.2.: Messungen Dateigröße der Ausgabedateien

6.2.3. Betriebssystem Test

OMNeT++ ist verfügbar für Linux, Windows und Mac/OS. Bei diesem Test soll geprüft werden, ob sich das Plug-in auf den genannten Betriebssystemen durch Hinzufügen im vorgesehenen *dropins* Ordner aktivieren lässt. Da bei der OMNeT++ Version 4.6 auf dem Windows System die Basis Java Version 1.7 genutzt wird, wird diese auf Version 1.8 geändert. Tabelle 6.3 zeigt, dass sich das Plug-in auf allen Betriebssystemen erfolgreich installieren lässt.

Plattform	OMNeT++ 4.6
Windows	✓
Mac/OS	✓
Linux	✓

Tabelle 6.3.: Simulationsgeschwindigkeiten der Aufzeichnungsmethoden

6.3. Plug-in

Bei der Suchzeit ist es wichtig, dass die Zeit nicht exponentiell ansteigt. In Tabelle 6.4 sind die gemessen Zeiten für die Sammlung der Daten aufgeführt. Das Diagramm in Abbildung 6.2 zeigt einen proportionalen Verlauf. Somit ist die Zeit für die Datensammlung linear und steigt nicht mit steigender Simulationszeit an.

Aufzeichnungsmethode	5 sec	10 sec	15 sec	20 sec
eigenes Eventlog	5 sec	10,8 sec	14,9 sec	19,6 sec

Tabelle 6.4.: Zeiten für die Sammlung der Daten

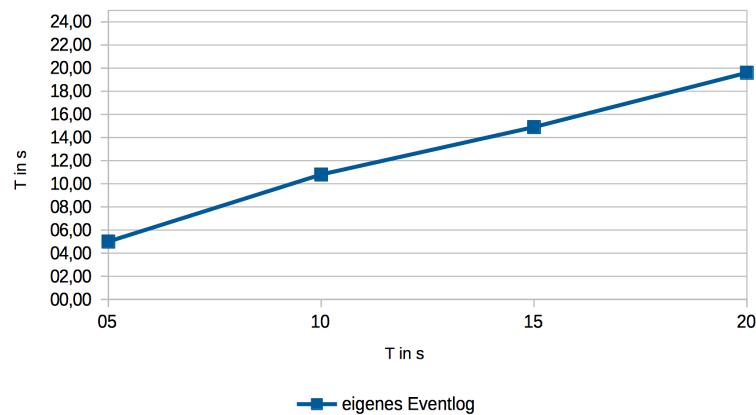


Abbildung 6.2.: Messungen Datensammlung

Beim Testen der Funktionalität des Plugins wurden folgende Projekte aus der CoRE Projekt Gruppe (vgl. CoRE Research Group) genutzt.

- AS6802 small network
- avb_AS6802 large network
- SignalsAndGateways majorNetwork (SaG)

Tabelle 6.5 zeigt alle getesteten Anforderungen. Das Plug-in konnte alle Test Projekte erfolgreich darstellen und ausführen. Somit ist die Evaluierung des Plug-ins erfolgreich.

Anforderungen	AS6802	avb_AS6802	SaG
Wurde das eigene Eventlog erfolgreich erstellt?	✓	✓	✓
Konnte das Plugin gestartet werden?	✓	✓	✓
Wurden alle Traffic Klassen und Streams erkannt?	✓	✓	✓
Werden alle Diagramme richtig dargestellt?	✓	✓	✓
Sind die Latenzen und Jitter bei dem Wechsel zur nächsten Ebene wiederzufinden?	✓	✓	✓

Tabelle 6.5.: Testkriterien - Angewandt auf verschiedene Projekte

7. Zusammenfassung

Thema der Arbeit war es, dass Zeitverhalten eines Echtzeitnetzwerkes mit Hilfe von Gantt Charts visuell darzustellen. Es hat sich herausgestellt, dass durch den Einsatz von Gantt Diagrammen nicht nur eine Aussage über die aufgetretenen Zeiten im Netzwerk getroffen werden können, sondern auch ein Blick auf die Ursachen möglich ist.

Um eine einheitliche Wissensbasis zu schaffen, wurden zu Beginn der Arbeit einige Grundlagen übermittelt. Zum einen wurden die zu analysierenden Echtzeit-Ethernet-Netzwerke erläutert und ihre wichtigen Zeitanforderungen dargestellt. Danach gab es Informationen zu der eingesetzten Simulationsumgebung OMNeT++, in denen die zur Darstellung benötigten Zeiten gemessen werden. Da OMNeT++ in das Eclipse-Framework eingebettet ist und daraus resultierend, Eclipse Plug-ins am besten geeignet sind, um eine Erweiterung in die vorhandene Plattform zu integrieren, wurden deren Aufbau und Eigenschaften zum Verständnis für die Implementierung beschrieben. Am Ende der Grundlagen wurden Gantt Charts, welche zum darstellen eingesetzt werden, definiert.

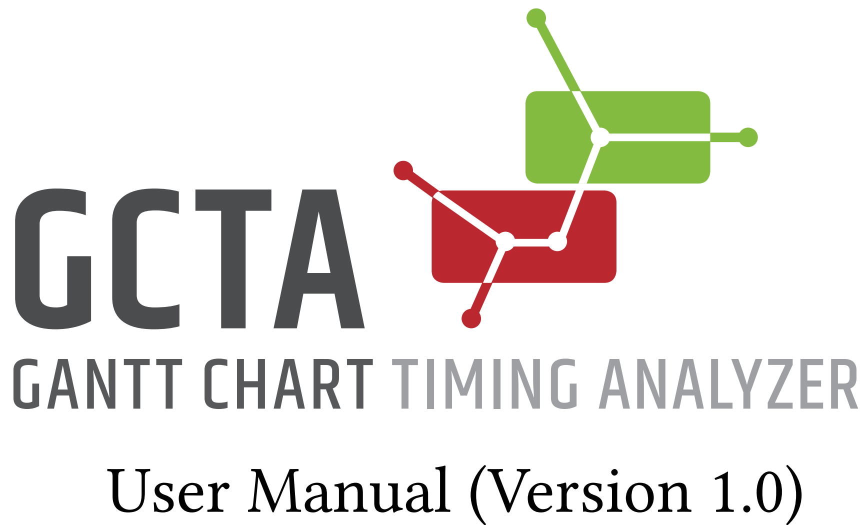
Im Hauptteil der Arbeit wurde eine umfangreiche Analyse erarbeitet, ein Konzept für das Programm entworfen und die Implementierung mit abschließender Evaluierung erläutert.
Durch die Analyse ist festgestellt worden, dass Ende-zu-Ende Latenzen, Modullatenzen und die dabei aufgetretenen Jitter für das Zeitverhalten maßgebend sind. Weiterhin wurde eine Semantik für die Gantt Charts entwickelt, welche eine übersichtliche Darstellung bietet und darüber hinaus noch eine Ursachenfindung ermöglicht. Dafür wurden verschiedene Ebenen eingesetzt, um den Nutzer bei der Wahl der relevanten Ende-zu-Ende Latenzen zu unterstützen und um die Menge der darzustellenden Latenzen besser überblicken zu können. Es wurde erkannt, dass bei Echtzeit-Ethernet Netzwerken die Anzahl der zurückgelegten Hops eines Frames, vom Sender zum Empfänger, und die Modullatenzen im Switch eine Erkenntnis über mögliche Ursachen von nicht eingehaltenen Deadlines bieten. Daher wurden zwei Varianten zur Darstellung konzeptioniert. Die Analyse der Methode zur Datenerfassung in OMNeT++, hat ergeben, dass ein eigenes Eventlog für die Performanz der Simulation, die Größe der re-

sultierenden Datei und die Performanz der anschließenden Zusammenstellung der Daten am besten geeignet ist. Anhand dieser Analyse konnte in Konzept erstellt werden, welches daraufhin erfolgreich implementiert werden konnte. Bei der Evaluierung wurden die Vorteile durch das eigene Eventlog bestätigt und beim Testen des Plug-ins wurden alle Anforderungen erfüllt.

Abschließend kann gesagt werden, dass das Ziel der Arbeit erreicht wurde. Über das Plug-in lässt sich das Zeitverhalten der Ethernet-Netzwerke visuell darstellen. Durch die Nutzung von Gantt Charts konnten Latenzen und Jitter ideal dargestellt werden und durch die Entwicklung von verschiedenen Ebenen können auch große Netzwerk Topologien einfach analysiert werden. Durch die Erstellung eines Eclipse Plug-ins konnte das Programm zum Anzeigen der Diagramme ideal in die Simulationsumgebung integriert werden.

A. Gantt Chart Timing Analyzer - Manual

In dem Anhang befindet sich das erstellte Manual. Es enthält eine Anleitung zur Bedienung des Programms, eine Installationsanleitung und eine Erläuterung zu den Diagramm. Das Manual wurde für den Endnutzer entwickelt.



User Manual (Version 1.0)

Philipp Kloth
Haw Hamburg
CoRE Research Group

Copyright © 2015 Philipp Kloth HAW Hamburg

Contents

1 Introduction **3**

2 Introduction to Gantt Charts for timing analysis **4**
2.1 Hop perspective 5
2.2 Switch perspective 9
2.3 accumulated best- and worst-case diagram 12

3 Installation **13**
3.1 OMNeT++ 13
3.2 Plug-in 14

4 Using the Tool **15**
4.1 Collecting the logging Data from your network 15
4.2 Selection Window 17
4.3 Chart View 18

1. Introduction

End-to-end latencies are significant numbers in real-time networks. Real-time networks must guarantee response within specified time constraints, often referred to as ”deadlines”. These deadlines will have the highest priority, because if a deadline is not reachable in time the data will be unusable. The Gantt Chart Timing Analyzer gives a visual overview about the end-to-end latencies and the module latencies of the simulation (see fig. 1). You can see the best and worst case latencies and are able to check if your time requirements can be met. You get these information via Gantt Charts for an easy way to analyse if your simulated network meets all deadlines in time.

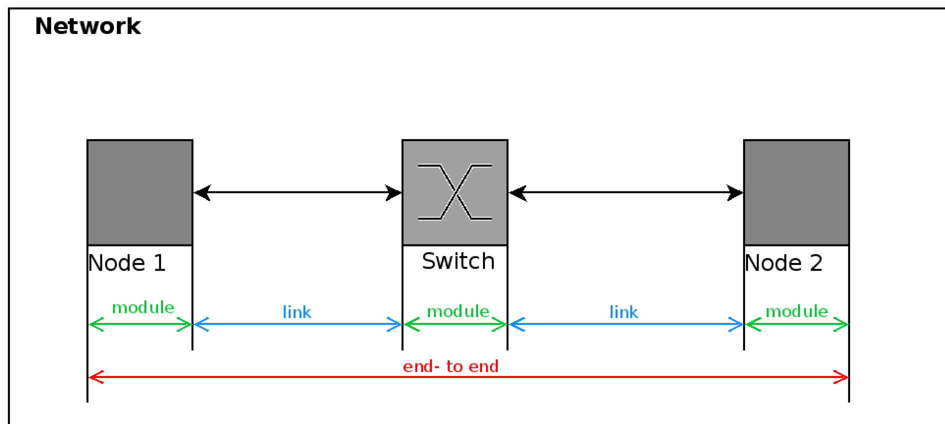


Figure 1: Network Latencies Example

2. Introduction to Gantt Charts for timing analysis

End-to-end latencies and module latencies will be displayed by the Gantt chart. To get an overview about the different Gantt Charts, the following example includes all available Gantt Charts based on the example network topology below. This network is the *large network AVB* example project, which is included in the CoRE4INET Software Package of the CoRE Project Group. You can find the topology of the network in the figure below.

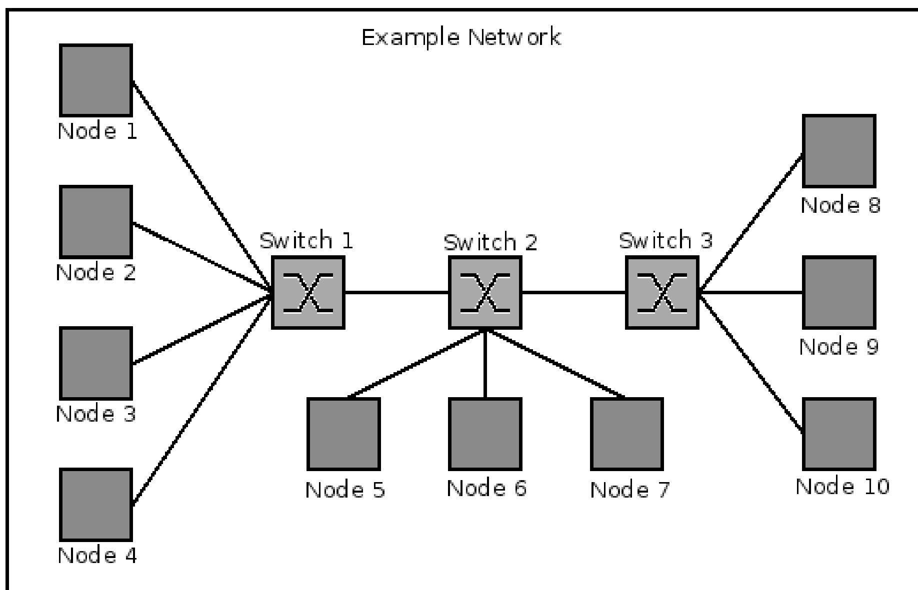


Figure 2: Example network topology

For the example the selected traffic class is *CoRE4INET::AVBFrame* and the selected stream is *–all–*. You can find the occurred end-to-end routes with their minimal- and maximal latencies and the resulting jitters in table 1.

Start	End	min Latency	max Latency	Jitter	Hop count
Node 1	Node 10	146,40 µs	467,16 µs	320,76 µs	4
Node 1	Node 8	146,40 µs	469,32 µs	322,92 µs	4
Node 2	Node 8	146,40 µs	472,44 µs	326,04 µs	4
Node 1	Node 6	107,80 µs	285,48 µs	177,68 µs	3
Node 1	Node 4	69,20 µs	132,15 µs	62,95 µs	2

Table 1: Logging data of the example network

The Gantt Charts are divided into two categories. The hop perspective and switch perspective. Both perspectives have three levels. With these diagrams the user gets an overview about the messages which doesn't reach the time requirements and a first idea of the reason.

2.1. Hop perspective

The hop perspective displays end-to-end latencies based on their hop count. Therefore the user can better compare the data, because of different hop counts be determined physically different latencies.

Level 1

The first level displays latencies based on the hop count. You can see the minimum- and maximum latencies for each occurred hop count and the resulting jitter. In the example network it will be four, three and two hops. All latencies for each hop count will be summarized in one bar. The left side of the bar shows the minimum latency of all matching messages and the right side of the bar shows the maximum latency. The bar itself displays the jitter. In Figure 3 you see the resulting chart. The maximum latency shown is defined by the user. (see section 4.2). The user must decide which maximum latency for the selected traffic class and stream must be respected. The grey line on the bar displays the average of all matching latencies. About the average score you get a tendency whether the latencies tend to maximum or minimum.

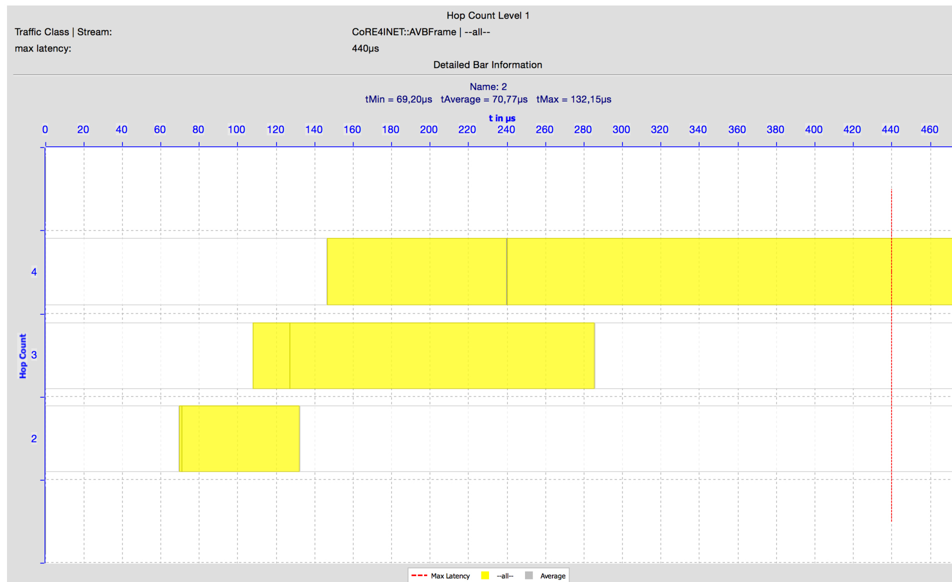


Figure 3: Hop perspective - level 1 - example network (see fig. 2)

Level 2

Level two displays all end-to-end routes with the number of hops you are interested in for your timing research. The semantic of the bars is the same as level one. The Figure below displays the resulting Gantt chart, if four hops have been selected on the higher level.

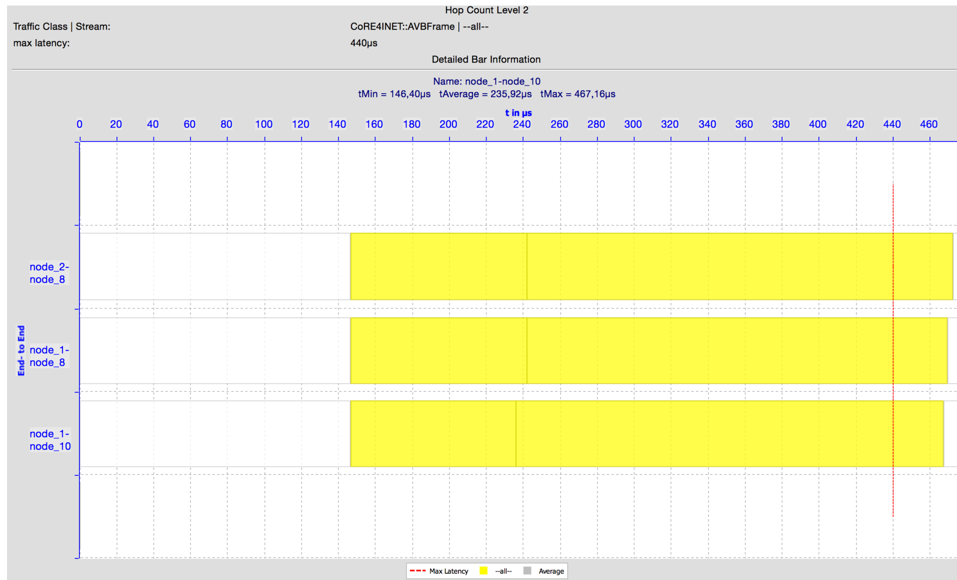


Figure 4: Hop perspective - level 2 - example network (see fig. 2)

Level 3

On the third level you will see the best- and worst-case latency for the end-to-end route in a detailed presentation. The best- and worst-case is divided into the resulting module- and link latencies. The sum of all module latencies gives the end-to-end latency. The green bars represents the best- and the red bars the worst-case. The left side of these bars represents the arrival time in the module and the right side shows the sending time to the next module. The difference between the best- and worst-case bar of one module is the resulting jitter. In Figure 5 you see the best- and worst-case latencies of the route from node 2 to node 8.

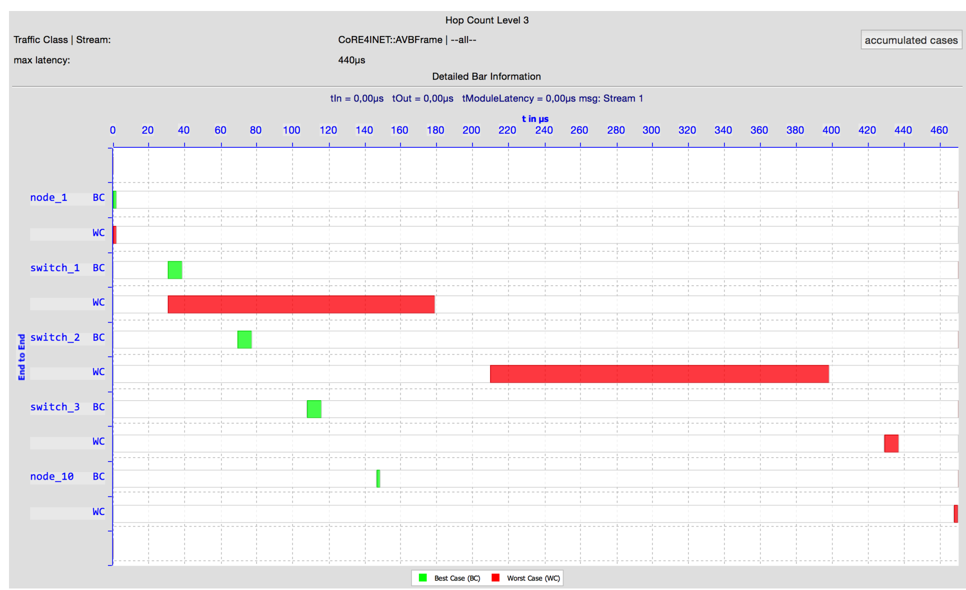


Figure 5: Hop perspective - level 3 - example network (see fig. 2)

2.2. Switch perspective

The switch perspective based on the switch latency. Switch latencies in switched Ethernet networks are critical factors for occurring jitters. Switches include queues for buffering messages. These are used if bandwidth has full capacity or because low-priority messages on the basis of higher-priority messages are waiting. Latencies from the switch give an insight whether the traffic is high or whether frames have to wait in the queues for a longer time.

Level 1

The first level displays all latencies based on the receive and send time of the message in the switch. This gives an overview of the switch latencies. If switches have high latencies, it is often the reason for too much traffic or errors in the configuration. Figure 6 displays the switch view of the example network. The maximum latency shown is defined by the user. (see section 4.2). The user must decide which maximum latency for the selected traffic class and stream must be respected.

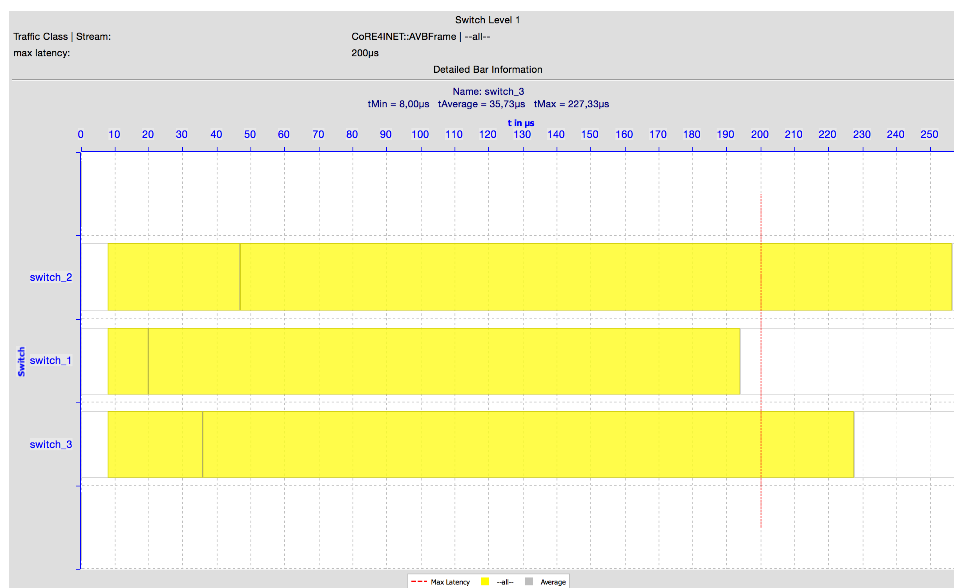


Figure 6: Switch perspective - level 1 - example network (see fig. 2)

Level 2:

The second level displays the switch latencies, depending on the number of hops that a message has covered on its end-to-end route. Figure 7 are the switch latencies based on the hop count which passed switch 2. In this example you can see that switch 2 needs less time for a low hop count then a higher hop count. For example a high network utilization on routes with 4 hops could be a reason. This level also represents a link to the hop perspective.

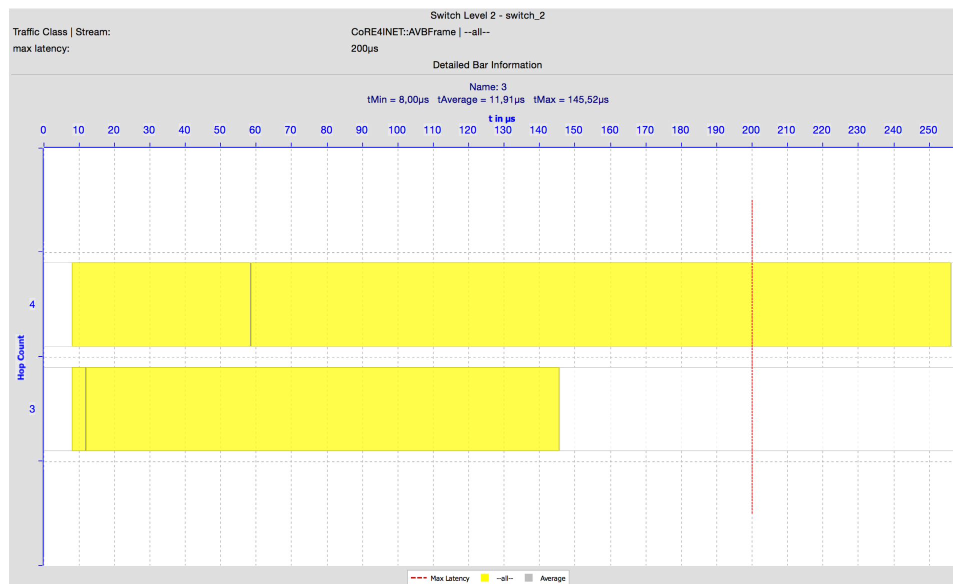


Figure 7: Switch perspective - level 2 - example network (see fig. 2)

Level 3

The bar semantic of this level is equivalent with level three in hop perspective (see section 2.3). Figure 8 displays best- and worst case latencies of messages who needed 3 hops (selected on level 2) on their route. You can see that switch 2 (selected on level 1) has the best case module latency on the route from node 1 to node 8 and the worst case latency on route from node 2 to node 8.

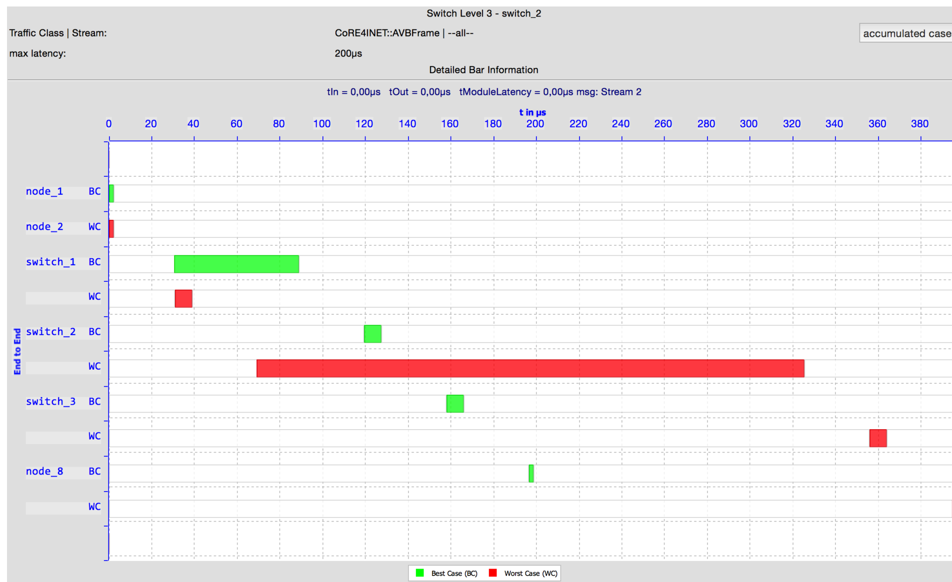


Figure 8: Switch perspective - level 3 - example network (see fig. 2)

2.3. accumulated best- and worst-case diagram

For both perspectives the third level can be displayed as an accumulated best- and worst-case. This view is virtual. That means that all displayed module latencies have occurred, but over the full simulation time and not on a single message. This offers a glance what times could occur, on the absolute best- and worst case. In fig. 9 you see that, the absolute worst case has a twice as high latency. This creates an impression which can happen, if your network is running over a long time.

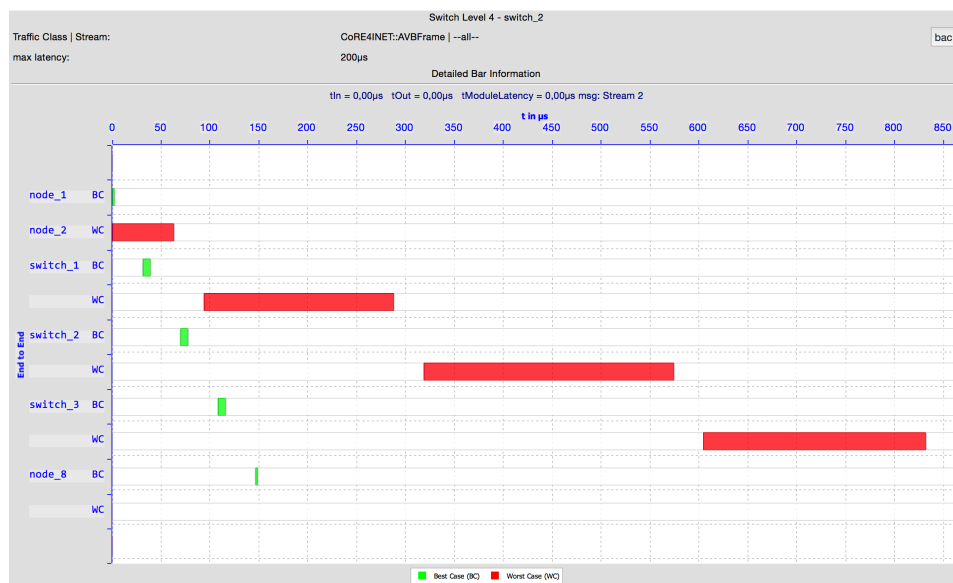


Figure 9: Accumulated best- and worst-case - switch level 3 - example network (see fig. 2)

3. Installation

In this section you get the information to integrate the plugin into your OMNeT++ Framework. The Plug-in works on OMNeT++ 4.6!

3.1. OMNeT++

Conditions

To make sure that the timing.tlog Eventlog will work correctly you have to check two module names of your network.

1. The name of the Application Module, where your message will be created. It has to contain *App* in his name.
2. The switches have to contain *switch* in his names.

Installation

You need to change some OMNeT++ files to get the timing.tlog log files after the recording. These files are important for the timing tool.

1. First you have to save the Files. Go to *<yourPath>omnetpp-"version"/src/envir* and copy *eventlogfilemgr.cc* and *eventlogfilemgr.h* to a separate Directory. After that copy the source files from *GanttChartTimingAnalyzer/src* to your OMNeT++ Directory at path *<yourPath>/omnetpp-"version"/src/envir*. Replace the old files with the new ones.

2. Open a Terminal or Console and go to your OMNeT++ Folder with
   $ cd <yourPath>/omnetpp-"version"/

   Then enter the following commands:
   $ make clean
   $ make

   The build process will take a few seconds. The timing.tlog recording files are available when you start the recording in OMNeT++.

3.2. Plug-in

Conditions

To install the Plug-in, the required Java Version under Eclipse must be 1.8 or higher. To see the one Eclipse is running under on Windows or Linux, go to Help->About OMNeT++ IDE->Installation Details->Configuration, and look for the property *eclipse.vm*. On Mac/OS go to OMNeT++ IDE->About OMNeT++ IDE->Installation Details->Configuration. For example:

eclipse.vm=C:/Program Files (x86)/java/jre1.8.0_60/bin/javaw.exe

If the Java Version is lower you can change the VM in the *omnetpp.ini* file under *<yourPath>OMNeTpp-"version"/ide/omnetpp.ini* by entering following two lines under *openFile*.

Windows
-vm
C:\jdk1.8.0\bin\javaw.exe (your exact path to javaw.exe could be different, of course)

Linux
-vm
/opt/sun-jdk-1.8.0/bin/java (your exact path to javaw.exe could be different, of course)

Mac/OS
-vm
/Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/bin/java (your exact path to javaw.exe could be different, of course)

Installation

Copy the eclipse Plug-in files from *GanttChartTimingAnalyzer/plugin* to your OMNeT++ Directory at path *<yourPath>omnetpp-"version"/ide/dropins*.
You have to restart your OMNeT++ Application and it will automatically find the plugin and activate it.

4. Using the Tool

4.1. Collecting the logging Data from your network

To start the plug-in you have to open the OMNeT++ application and do the following steps:

1. Open the *inifile* of the network you want to analyse and add the following Comment in the *[General]* section of the File:

   #GanttTimingEnabled

   This will enable the logging for the Gantt Chart Timing Analyzer.

   - To speed up the simulation you have to turn off the OMNeT++ eventlog-recording by adding the following command to the *inifile* ***.module-eventlog-recording = false*

2. To generate the required result file turn on the recording and start the simulation (see fig. 10).

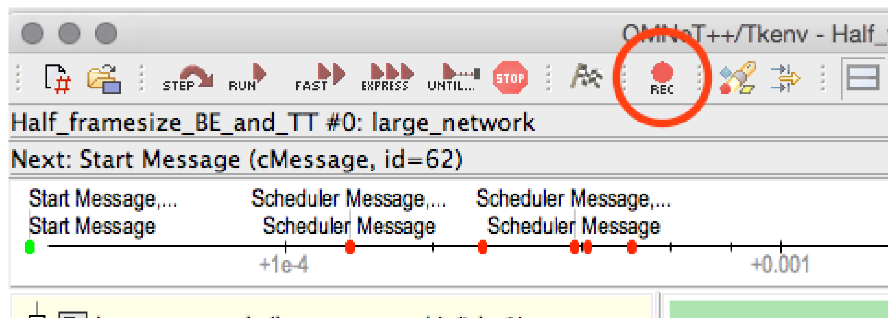


Figure 10: Activate the recording

3. At the end of the simulation you have to call the finish() method by shutting down the simulation window and confirming with yes or by clicking the finish() symbol in the IDE.

4. The *timing.tlog* result file will be restored in the results folder of your simulated project.

5. Right-click on the *timing.tlog* file and select *Gantt Chart Timing Analyzer* to start the Tool .

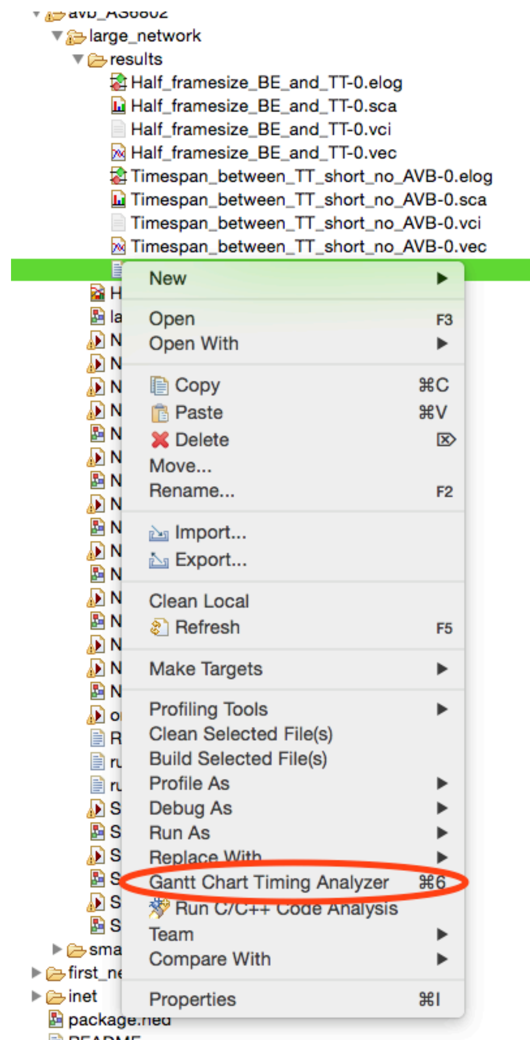


Figure 11: Start the plugin

4.2. Selection Window

If you have started the tool, select start searching. First the tool have to collect the data from the result files and calculate the times. This may take a few seconds depending on the number of messages and the size of the topology. After searching you will see the following window:

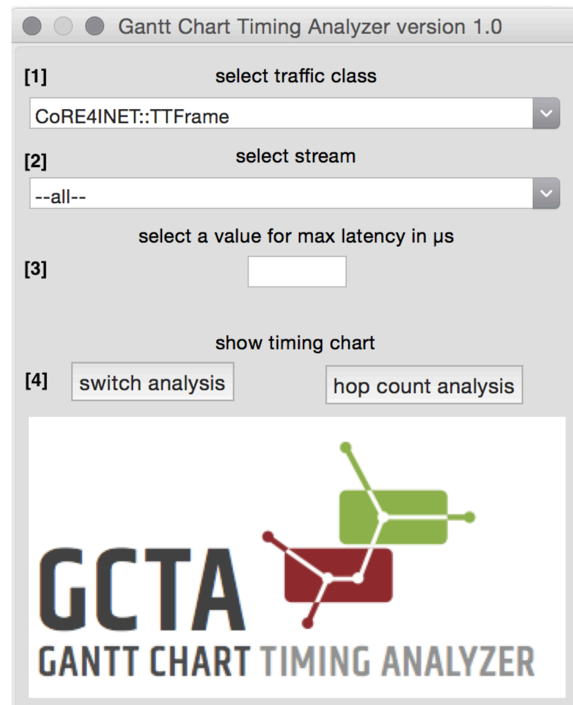


Figure 12: Selection Window

1. Select a traffic class you want to analyse.
2. Select a Stream from your Traffic Class or "–all–" if you want to see all Messages.
3. This value will be shown as your max latency in the chart. If you leave it blank no max latency will be displayed.
4. Select the chart perspective you prefer and go to the chart view.

4.3. Chart View

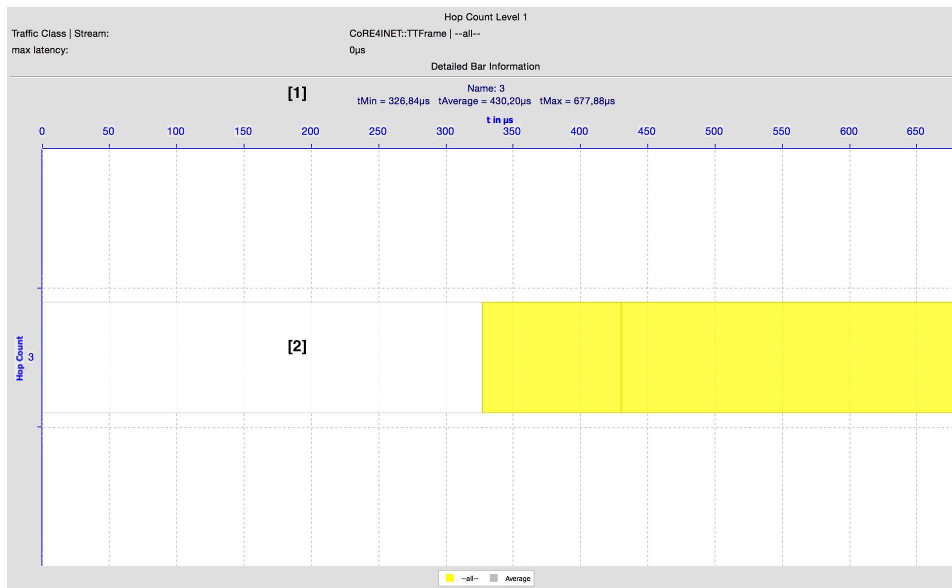


Figure 13: Chart View

1. These information text includes the timing data in numbers for the bars. You can click on the Bar with a single click, to open the information text of the bar.
2. If you want to reach the next level you have to double click on the bar.

The chart view displays the selected Gantt chart, together with all information's. You can see which Traffic Class and Stream are selected and the max Latency you have chosen. For all timing bars there will be a detailed bar Information section, with the occurred Latencies in numbers.

Literaturverzeichnis

[Beck 2003] Beck, Dr. H.: *Grundlagen der Netzwerktechnik.* url = http://www.gwdg.de/~hbeck. 2003

[Bormann und Hilgenkamp 2006] Bormann, Alexander ; Hilgenkamp, Ingo: *Industrielle Netze - Ethernet-Kommunikation für Automatisierungsanwendungen.* Hüthig Verlag, 2006. – ISBN 978-3-778-52950-8

[Buschmann 1998] Buschmann, Frank: *Pattern-orientierte Software-Architektur - ein Pattern-System.* 2. Aufl. München : Pearson Deutschland GmbH, 1998. – ISBN 978-3-827-31282-2

[CoRE Research Group] CoRE Research Group : *url = http://core.informatik.haw-hamburg.de/en/*

[CoRE4INET] CoRE4INET: *CoRE Research Group.* url = http://core4inet.realmv6.org/trac/

[Eclipse Foundation a] Eclipse Foundation: *Eclipse documentation.* url = http://help.eclipse.org/luna/index.jsp

[Eclipse Foundation b] Eclipse Foundation: *url - www.eclipse.org*

[HANSER automotive networks 2013] HANSER automotive networks: Toolkette zum Modellieren von Ethernet-Netzwerken. In: *url - www.hanser-automative.de* (2013)

[Holleczek und Vogel-Heuser 2001] Holleczek, Peter ; Vogel-Heuser, Birgit: *Echtzeitkommunikation und Ethernet/Internet - PEARL 2001 Workshop über Realzeitsysteme Fachtagung der GI-Fachgruppe 4.4.2 Echtzeitprogrammierung, PEARL Boppard, 22./23. November 2001.* Wiesbaden : Springer Berlin Heidelberg, 2001. – ISBN 978-3-540-42706-3

[Metcalfe u. a. 2014] Metcalfe, Bob ; Kozierok, Charles M. ; Correa, Colt ; Boatright, Robert B. ; Quesnelle, Jeffrey ; Holden, Matt ; Irving, Kyle: *Automotive Ethernet - the Definitive Guide* -. Intrepid Control Systems, Incorporated, 2014. – ISBN 978-0-990-53882-0

[Omnet++ Community a] Omnet++ Community: *Omnet++ 4.6.* url - www.omnetpp.org

[Omnet++ Community b] OMNET++ COMMUNITY: *Omnet++ User Manual Version 4.6*

[Schumann 2013] SCHUMANN, Heidrun: *Visualisierung - Grundlagen und allgemeine methoden.* 2000. Aufl. Berlin Heidelberg New York : Springer-Verlag, 2013. – ISBN 978-3-642-57193-0

[Shavor u. a. 2004] SHAVOR, Sherry ; FAIRBROTHER, Scott ; D'ANJOU, Jim ; KELLERMAN, John ; KEHN, Dan: *Eclipse - Anwendungen und Plug-Ins mit Java entwickeln.* 1. Aufl. München : Pearson Deutschland GmbH, 2004. – ISBN 978-3-827-32125-1

[SWTChart project Community] SWTCHART PROJECT COMMUNITY: *SWTChart light weight chart component.* url = http://www.swtchart.org

[Tanenbaum 2009] TANENBAUM, Andrew S.: *Moderne Betriebssysteme -.* 3. aktualisierte Auflage. München : Pearson Deutschland GmbH, 2009. – ISBN 978-3-827-37342-7

[Vogel und Milinkovich 2015] VOGEL, Lars ; MILINKOVICH, Mike: *Eclipse Rich Client Platform -.* 3. Aufl. Lars Vogel, 2015. – ISBN 978-3-943-74714-0

[Wilson 2002] WILSON, James M.: Gantt charts: A centenary appreciation. In: *European Journal of Operational Research* (2002)

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 21. September 2015 Philipp Kloth