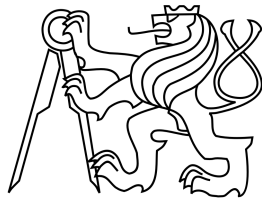


Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering



Master's thesis

**Modelling of Profinet
communication**

Bc. Jan Prášek

May 2015

Thesis supervisor: **Ing. Pavel Burget, Ph.D.**

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Jan Prášek**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **Modelling of Profinet communication**

Guidelines:

1. Create model of a basic Profinet network with switches, real-time cyclic and acyclic communication, and non-real-time communication.
2. Parametrise the model according to an existing network, i.e. take into account the topology and existing communication relations.
3. Compare the modelled and physical behaviour of the network and use the results to improve the model.
4. Use the model for the diagnostics of the physical network based on offline and online traffic measurement.

Bibliography/Sources:

- [1] Industrial communication networks - Fieldbus specifications - Part 5-10: Application layer service definition - Type 10 elements. IEC 61158-5-10 ed.3. 08/2014.
- [2] Industrial communication networks - Fieldbus specifications - Part 6-10: Application layer protocol specification - Type 10 elements. IEC 61158-6-10 ed.3. 08/2014.
- [3] OMNeT++ User Manual. Version 4.6.

Diploma Thesis Supervisor: Ing. Pavel Burget, Ph.D.

Valid until the summer semester 2015/2016



Prague, February 20, 2015

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 11 května 2015

Měněl
.....

Poděkování

Rád bych poděkoval především panu Ing. Pavlu Burgetovi, Ph.D. za skvělé vedení a panu Ondřeji Fialovi za cenné rady a připomínky. Dále také děkuji své rodině a přátelům za významnou podporu během celých studií.

Abstrakt

Tato práce využívá simulační nástroj OMNeT++ k simulaci sítě Profinet. Výsledné řešení modelů jednotlivých síťových zařízení je vysoce parametrizovatelné a připravené pro možnost budoucího rozšíření. Dále se práce zabývá také samotnou parametrizací sítě, jednotlivých zařízení a probíhající komunikace. Pro možnost automatizace simulací byla také v rámci této práce navržena jednoduchá API knihovna. Výsledný model byl následně ověřen měřeními a také bylo naznačeno jeho možné použití pro diagnostiku modelovaných sítí.

Klíčová slova: Profinet, modelování sítě, OMNeT++, JSON

Abstract

This thesis uses OMNeT++ simulation tool for the Profinet network modelling. The introduced modules which are used to model the real network devices are implemented in a highly customizable way with a possibility of future extensions. Thesis also introduces a parametrization structure for the defined modules as well as for the network topologies themselves. A simple API support for the implemented solution is also provided to automate the simulation process. Several experiments were conducted to show the possible uses of the Profinet simulation model and to verify it.

Keywords: Profinet, network modelling, OMNeT++, JSON

Contents

1	Introduction	1
1.1	Assignment overview	1
1.2	Proposed solution	1
2	Implementation	3
2.1	OMNeT++	3
2.1.1	Framework overview	3
2.1.2	Message EthFrame	6
2.1.3	Channel ethernet_100Mb	7
2.1.4	Simple module EndStation	8
2.1.4.1	NED definition	9
2.1.4.2	C++ class	10
2.1.4.3	Events	13
2.1.5	Simple module SimpleSwitch	16
2.1.5.1	NED definition	16
2.1.5.2	C++ class	18
2.1.6	Compound module GenericDevice	20
2.1.6.1	NED definition	20
2.1.7	Derived modules	21
2.1.7.1	Simple module creation	22
2.1.7.2	Compound module creation	25
2.1.7.3	Compilation	26
2.2	JSON definition	26
2.2.1	JSON format overview	26
2.2.2	Simulation parameters	27
2.2.3	Custom events	28
2.2.4	Network definition	29
2.2.4.1	Nodes	29
2.2.4.2	Links	31
2.3	Profinet simulation API	32
2.3.1	API implementation	32
2.3.2	API usage	34
2.3.3	Console application	36
3	Modelling a network	37
3.1	Physical network	37
3.2	Setting the parameters	38
3.2.1	Switching parameters	38
3.2.2	Communication parameters	41
3.3	Running the simulation	43

3.3.1	Using the OMNeT++ IDE	43
3.4	Simulation results	44
3.4.1	Experiments	44
3.4.1.1	Experiment 1	45
3.4.1.2	Experiment 2	48
3.4.1.3	Experiment 3	50
3.4.1.4	Experiment 4	51
3.4.1.5	Experiment 5	52
4	Conclusion	55
	Appendices	57
A	Support analysis tools	57
A.1	Matlab script vectorAnalysis.m	57
A.2	Tool packetAnalyser	57
	References	59
	Attached CD contents	61

List of Figures

1	OMNeT++ structure.	4
2	Inheritance diagram for class <i>EthFrame</i>	4
3	Inheritance diagram for classes <i>EndStation</i> and <i>SimpleSwitch</i>	5
4	Explicitly tagged VLAN Ethernet frame.	7
5	EndStation simple module.	8
6	Events sequence diagram.	13
7	SimpleSwitch simple module.	17
8	GenericDevice compound module.	21
9	Overall class diagram.	22
10	Adding API shared library to the linker.	35
11	Test network topology.	37
12	Measurement scheme for the Switch 1 device.	39
13	Measurement scheme for the Switch 1 device with additional traffic.	40
14	Switching delay dependency on additional traffic of Switch 1.	41
15	Switching delay dependency on additional traffic of Switch 2.	41
16	Browsing results in the OMNeT++ IDE.	44
17	Experiment 1 simulation error on link <i>Controller</i> \rightarrow <i>Switch 1</i>	46
18	Experiment 1 simulation error on link <i>Controller</i> \leftarrow <i>Switch 1</i>	46
19	Experiment 1 simulation error on link <i>Controller</i> \leftrightarrow <i>Switch 1</i> after the compensation.	47
20	Experiment 2 simulation error on link <i>Controller</i> \rightarrow <i>Switch 1</i>	48
21	Experiment 2 simulation error on link <i>Controller</i> \leftarrow <i>Switch 1</i>	49
22	Experiment 2 simulation error on link <i>Controller</i> \leftrightarrow <i>Switch 1</i> after the compensation.	49
23	Experiment 3 simulation error on link <i>Controller</i> \rightarrow <i>Switch 1</i>	50
24	Experiment 3 simulation error on link <i>Controller</i> \leftarrow <i>Switch 1</i>	51
25	Experiment 3 simulation error on link <i>Controller</i> \leftrightarrow <i>Switch 1</i> after the compensation.	51
26	Experiment 4 simulation error on link <i>Controller</i> \leftrightarrow <i>Switch 1</i> after a network failure.	52
27	Experiment 5 simulation error on link <i>Controller</i> \leftrightarrow <i>Switch 1</i> with an error in HW configuration.	53

List of Tables

- 1 Priority queue mapping based on number of queues and frame priority. 16

Listings

1	Adding an action example.	11
2	Action handle method declaration example.	15
3	Getting parameter value examples.	15
4	<i>DerivedEndStation</i> NED file definition	23
5	<i>DerivedEndStation</i> header file definition	24
6	<i>DerivedEndStation</i> source file definition	24
7	<i>DerivedDevice</i> NED file definition.	25
8	JSON structure example	27
9	Custom events example.	29
10	Node's JSON definition example.	31
11	Link's JSON definition example.	32
12	API example usage.	35
13	API console application usage.	36

Part 1

Introduction

1.1 Assignment overview

The main task of this thesis was to use the OMNeT++ simulation tool for the Profinet network modelling. The designed tool was used to model a real physical Profinet network. The acquired model was then used for network diagnostics.

1.2 Proposed solution

Profinet is a widely used industrial Ethernet standard[1]. The designed simulation model therefore operates on the link layer introduced by the ISO/OSI reference model[2] and aims to model basic devices typical for Profinet - end nodes and switches. The designed model introduces several simplifications that are typical for Ethernet industrial use. Only full-duplex communication within a loop-free network with static topology was considered. This means that no MAC[3] (medium access control) and loop resolution mechanisms[3] have to be modelled. Also only 100Mbit Ethernet standard[3] is considered since it is the only one supported by the Profinet[1]. However on the other hand, priority mechanics using VLAN priority tagging[3] are included in the model's behaviour to distinguish real-time and non real-time communication.

The simulation environment OMNeT++[4] was chosen by the thesis assignment for the simulation implementation. The OMNeT++ framework is well supported and frequently updated. It also contains an open source library *INET*[4] that models various network layers and their protocols, including link layer and Ethernet. Usage of some of the *INET*'s defined parts was considered, yet it turned out to be too complex to adjust for our needs. Therefore the proposed Profinet simulation model was implemented.

Apart from the simulation model itself, means of parametrization had to be defined. Two mostly used data container formats were considered - JSON[5] and XML[6]. In the end the JSON format was chosen since it is well supported, easily parsed and better readable by humans.

To support the usage of the designed simulation model for external applications an API was defined in a form of a shared pre-compiled library.

All the implementation was done under the Ubuntu Linux distribution to be compatible with already existing tools developed in our department.

1.2 PROPOSED SOLUTION

Part 2

Implementation

2.1 OMNeT++

In this chapter, a basic description of the OMNeT++ simulation framework and the description of all of the implemented parts in the introduced Profinet simulation model are given. These are:

- Message *EthFrame* in section 2.1.2.
- Channel *ethernet_100Mb* in section 2.1.3
- Simple module *EndStation* in section 2.1.4.
- Simple module *SimpleSwitch* in section 2.1.5.
- Compound module *GenericDevice* in section 2.1.6.
- Possibility of deriving user modules to extend the proposed functionality in section 2.1.7.

2.1.1 Framework overview

OMNeT++ is a discrete event simulator designed primarily for building network simulators[4]. It is implemented as a modular component based C++ library and it introduces a network description (NED) language extension operating above the C++ core[7]. An Eclipse based IDE with the NED language support is also provided. In addition, NED files can be viewed and edited in two modes - source code and graphical[7]. Typical OMNeT++ simulator consists of several layers (see figure 1):

- **Messages** — All communication between connected network modules is done using messages. Messages are implemented as a C++ class that can contain user data that are to be transported through the network. Another way to utilize messages is to use them as timed self-messages to construct timed events within a given module. The core library provides user with two generic types of messages implemented in C++ classes *cMessage* and *cPacket* (derived from *cMessage*, provides more functionality). User can define custom messages that extend one of the generic classes using a simplified C++ format supported by the OMNeT++ framework. Only the message type and variables that the message should hold are defined using this format. The corresponding C++ class is automatically built by

2.1 OMNET++

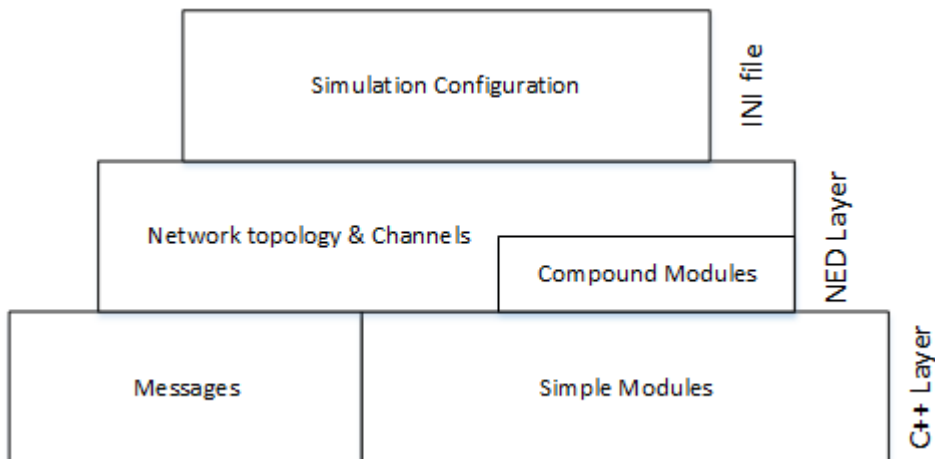


Figure 1: OMNeT++ structure.

the OMNeT++ framework. The class also automatically implements setters and getters for all of the defined variables. In the case of the Profinet simulation model, message *EthFrame* (extending *cPacket*) was designed (more in section 2.1.2), its inheritance diagram is shown on fig. 2.

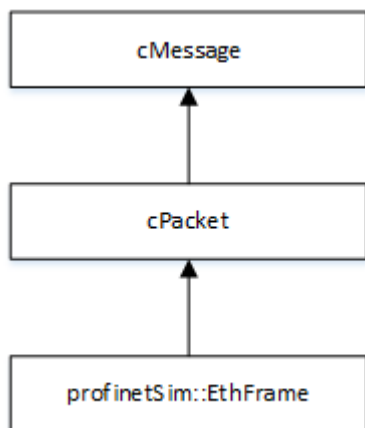


Figure 2: Inheritance diagram for class *EthFrame*.

- **Channels** — All connections between modules in a network topology (and within compound modules as well) are done using channels. There are 3 basic channel types defined in the library core of the OMNeT++ - ideal channel, datarate channel and delay channel. In our simulation model, mainly datarate channel is used to accurately simulate connections with desired bit rates. A custom channel *ethernet_100Mb* derived from the datarate channel has been designed using the NED language (see section 2.1.3).
- **Modules** — Devices in the network topology are modelled using modules. There are two types of modules user can define:

- **Simple module** — This module consists of two parts. First part is a C++ class derived from a blank simple module class *cSimpleModule* that specifies how the module behaves (for example how it reacts on incoming messages). Second part is a NED file describing module’s parameters, statistical signal definitions and connections to the outer world (gates). The parameters are automatically available to the tied C++ implementation. In the designed tool, there are two simple modules defined - *EndStation* (see section 2.1.4) and *SimpleSwitch* (see section 2.1.5). Inheritance diagram is shown on figure 3.
- **Compound module** — This is a complex module that can combine multiple simple modules and/or another compound modules into one. It is defined only with a single NED file which specifies interconnections, gates or parameter assignments. One compound module combining one *EndStation* and one *SimpleSwitch* together has been designed - *GenericDevice* (see section 2.1.6).

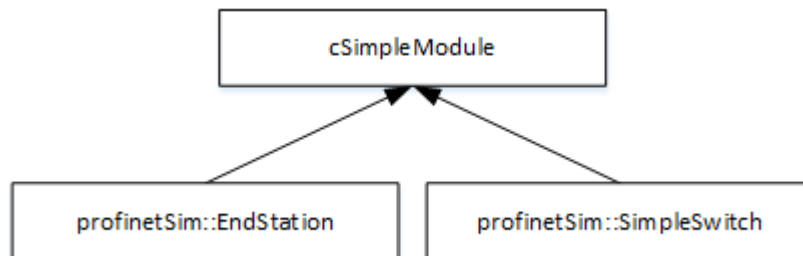


Figure 3: Inheritance diagram for classes *EndStation* and *SimpleSwitch*.

- **Network topology** — Network topology is designed using the NED language in a similar way as a compound module is - all of the modules and their interconnections are listed. Also, modules’ parameters are assigned. Typically, there are several networks defined within a project which may only differ in modules’ parameters or it may define a completely different topology.

Note, that the designed Profinet simulation model uses a JSON structure to capture the network topology (discussed in section 2.2). The NED file containing the topology is created automatically by the designed API (introduced in section 2.3) based on the JSON file provided.

- **Simulation configuration** — The configuration is done within an *.INI file that uses a special syntax (Refer to[7]). Various simulation parameters (duration, source network, etc.) can be assigned as well as remaining unassigned modules’ parameters. A single configuration file can be shared by multiple networks.

Note, that the designed Profinet simulation model uses a JSON structure to parametrize the simulation (discussed in section 2.2). The INI file containing the simulation parameters is created automatically by the designed API (introduced in section 2.3) based on the JSON file provided.

2.1 OMNET++

Special part of the framework are statistical signals. These are defined in simple modules and are recorded by the simulation to produce simulation results. The framework provides basic statistical functions (such as mean, variance, max value, min value, etc.) that can operate on the defined signals. There are two types of signals:

- **Scalar signals** — After the simulation completion only a single number is recorded per statistical signal. This type of signal saves disk space as well as the result evaluation time.
- **Vector signals** — Output of this type of signals is a sequence of all recorded event occurrences with their corresponding timestamps. Especially for longer simulations, the vector signals produce large-sized files.

Before any simulation can be run, all the source C++ files needs to be compiled. From the user point of view, this has already been taken care of and the executable has been created. However, should a user define additional simple modules or messages, the simulation model has to be recompiled. This option is discussed in section 2.1.7. All the NED files are loaded dynamically and do not require compilation, therefore user can create INI files, network topologies and even compound modules without any limitation while using a single executable.

OMNeT++ provides user with two simulation environments - Tkenv graphical runtime environment and Cmdenv command-line environment.

2.1.2 Message EthFrame

For the purpose of an Ethernet communication simulator an appropriate message type had to be defined - the *EthFrame* message. It extends the *cPacket* class from the OMNeT++ core library and is defined in a file */src/ethFrame.msg* using the simplified C++ format. The framework automatically creates */src/ethFrame_m.h* and */src/ethFrame_m.cc* files before every compilation (if they are missing or the message definition has changed). In these C++ source/header files, the class *EthFrame* is properly defined based on the simplified notation given by */ethFrame.msg* file. The following variables are defined in the *EthFrame* message to model the actual Ethernet frame with an explicit VLAN tagging format specified by the IEEE 802.3 (Ethernet frame) and IEEE 802.1q (VLAN tagging) standards (see figure 4)[3]:

- (int64) **destinationAddress** — Destination MAC address. Although the address is in fact 6 bytes long it is stored in a 8 byte type (int64). The frame physical data length is defined independently of any defined variables, therefore the difference of the data sizes has no effect on the transmission times throughout the datarate channel.
- (int64) **sourceAddress** — Source MAC address.
- (short) **typeLength** — Type/length encapsulation.

- (short) **vlanProtocolId** — VLAN protocol identifier. It has a fixed value of 0x8100.
- (short) **vlanTagCtrl** — VLAN control tag. It holds information about VLAN identifier and VLAN frame priority.
- (char) **data[]** — User data. The length of the array is specified after the object is created. Application may use this array freely to store user or application specific data.
- (int) **fcs** — Frame check sequence.

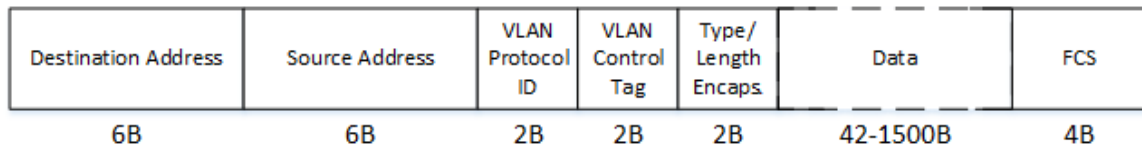


Figure 4: Explicitly tagged VLAN Ethernet frame.

The current implementation of the simulation model does not utilize all of the defined variables, yet they were included anyway to maintain the frame's structure and to be available for the future use.

Setters and getters are automatically implemented for all of the defined variables. Also, the parent class *cPacket* provides the *EthFrame* message object with two similar useful functions - *setByteLength(int64 byte_size)* and *setBitLength(int64 bit_size)*. These define the actual data size of the *EthFrame* and affect the propagation time through the datarate channels. If the actual *EthFrame* data size is not specified, the propagation delay through any channel is zero.

2.1.3 Channel ethernet_100Mb

This custom channel is defined in a NED file *ethernet_100Mb.ned* and it is located in the root folder of the simulation model. The designed channel extends OMNeT's datarate channel with the following parameters assigned:

- **datarate** — Data rate speed value. Set to industrial standard of 100Mbit/s.
- **disabled** — False (default value). The channel is enabled.
- **delay** — Cable length delay is currently neglected, the parameter is set to 0 (default value).
- **ber** — Bit error rate. The simulation model currently presumes error-free communication. Value is set to 0 (default value).
- **per** — Packet error rate. The simulation model currently presumes error-free communication. Value is set to 0 (default value).

2.1.4 Simple module EndStation

This simple module models the behaviour of a basic one port addressable end point device. It is capable of receiving and sending Ethernet frames (modelled as packets *EthFrame* - see section 2.1.2) and contains both input and output queues. As we don't assign the frames any particular meaning, when receiving a frame its statistics are recorded and after that the frame is discarded. The mechanics behind frame creation and frame sending will be discussed further in section 2.1.4.3. As every simple module does, the definition consists of two parts - a C++ class and a NED definition. A function scheme is depicted on fig 5.

Module *EndStation* can be connected to any other module defined within the project (even with another *EndStation*).

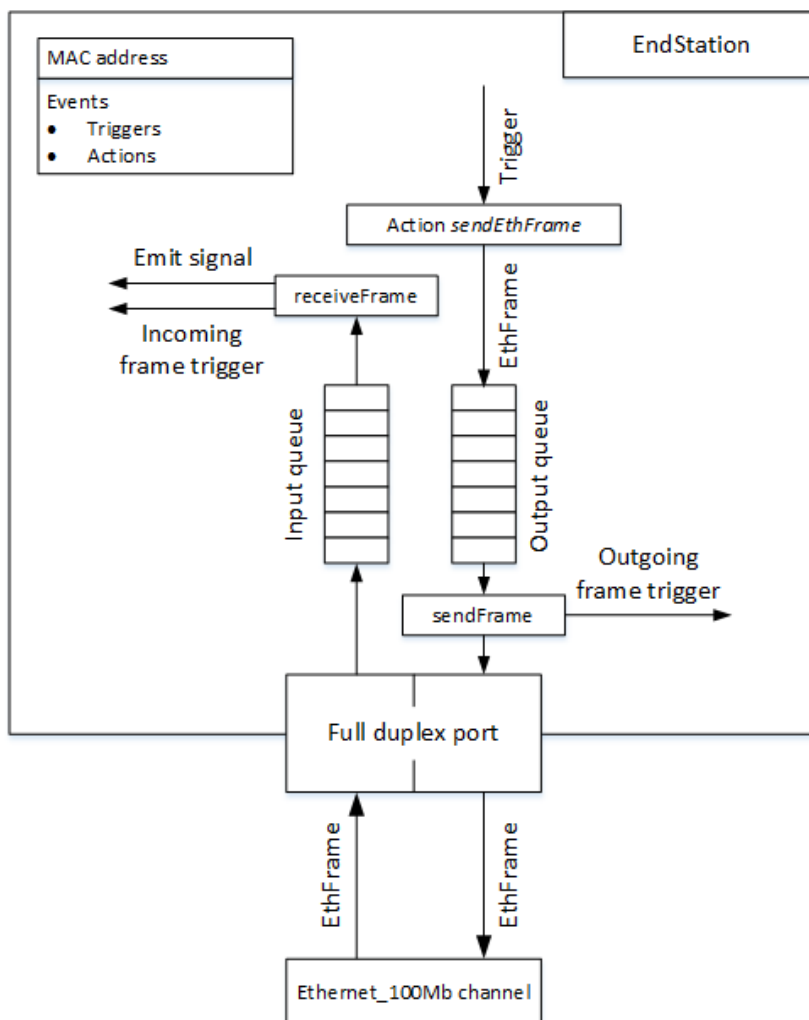


Figure 5: EndStation simple module.

2.1.4.1 NED definition

The NED file *endStation.ned* defining this simple module is located in the root folder of the project. The module is connected to the outer world by a single full-duplex port (gate in the NED terminology).

Parameters Following parameters have been defined:

- (double) **minFrameSpacingTime** @unit(s) — Inter-frame gap. It is the minimal time gap between two consecutive frames. This parameter has a default value of $0.96 \mu s$ which corresponds to the 100Mbit Ethernet standard [3]. The special tag *@unit(s)* signs that the expected value is entered using a time unit (framework takes care of any conversions).
- (int) **macAddressOui** — Upper 3 bytes of MAC address (organizationally unique identifier part). Default value 0x400000 (unicast, locally unique).
- (int) **macAddressOap** — Lower 3 bytes of MAC address (organizationally assigned portion part).
- (string) **sourceJson** — Serialized JSON string containing given *endStation*'s parameters and events (including custom events). This string is automatically generated by the API and should not be altered by the user. The default value is an empty string which corresponds to a blank module without any events.

Note: All parameters with defined default value don't have to be specified when creating network, but can be overwritten if desired. The NED parameters cannot use integral type with more than 4 bytes, therefore the MAC address had to be split into two parts.

Statistical signals Last important part present in the NED file is a statistical signals definition. There is one statistical signal defined which records the propagation delay times of arriving frames for each priority level - **frameDelayByPriority_***. The wildcard '*' stands for given priority number. All VLAN priority levels are considered (0-7 inclusive). The signal contains the following statistical template:

- **mean** — Scalar measurement that captures the mean value of propagation delay of received frames throughout the network at given priority level.
- **minimum** — Scalar measurement that records the minimal value of propagation delay of received frames throughout the network at given priority level.
- **maximum** — Scalar measurement that records the maximal value of propagation delay of received frames throughout the network at given priority level.
- **count** — Scalar signal that records the total number of received frames at given priority level.

2.1 OMNET++

- **vector** — Vector measurement that records propagation delay times of all received frames at given priority level with their timestamps.

2.1.4.2 C++ class

The class *EndStation* is defined in files *endStation.h* and *endStation.cc* located in the *src* folder. It is derived from a generic simple module class *cSimpleModule* and overrides several of its methods. The class structure is designed to support polymorphism to allow user an easy definition of derived classes that alter this module's behaviour.

Public methods Brief overview of this module's public methods:

- *public* **EndStation()** — Class constructor. Registers actions supported by this class.
- *public* **~EndStation()** — Class destructor. Frees dynamically allocated memory and cancels any pending timed events.

Protected methods

- *protected virtual void* **initialize()** — This method is automatically called after the class constructor. It initializes class variables, loads NED parameters, parses JSON string specified in the NED definition using *parseJson* method and creates statistical signals based on the template defined in the NED file. Periodic event *sendFrameEvent* that is responsible for sending generated *EthFrames* is also created.
- *protected void* **registerAction**(std::string action_identifier, ActionHandlerType action_handler) — This method registers an user-defined action handle method and binds it with given string identifier. It is called typically in the class constructor. More on actions in section 2.1.4.3. An example of this method's usage is shown in listing 1.

Input parameters:

- (std::string) **action_identifier** — String identifier tied to the given action. When this identifier is entered in the JSON file and the corresponding event is triggered, the action tied to this ID is executed. Note: If there is already an entry stored with the given string ID, the action handle function is overwritten.
- (ActionHandlerType) **action_handler** — The type of this parameter is actually *std::function <void(cMessage*)>* - a function wrapper that stores a function pointer of the given function. **Important:** It is necessary that the action handle function has return type *void* and one input parameter of type *cMessage**.

```

1 std::string action_id = ACTION_SEND_ETH_FRAME; //string ID
2 ActionHandlerType action_handler = std::bind( &EndStation::
        sendEthFrame, this, std::placeholders::_1);
3
4 registerAction( action_id, action_handler );

```

Listing 1: Adding an action example.

The code in listing 1 assigns the function *sendEthFrame* of the class *EndStation* to a string ID defined as a constant string *ACTION_SEND_ETH_FRAME*.

- *protected virtual void handleMessage*(*cMessage *msg*) — This method overwrites the implementation of its parent class *cSimpleModule*. It is called whenever a self-message time elapses or a message arrives on the input port. Three possible variants may occur:
 - There is an *EthFrame* message (which inherits from *cMessage* - see section 2.1.2) arriving on the input port. In this case, the input *cMessage* is retyped to *EthFrame* and is received by calling function *receiveFrame* upon it.
 - A time-triggered event defined in the JSON file has triggered (more on events and triggers in section 2.1.4.3). In this case *triggerHandle* routine is called upon the input message which contains the trigger information.
 - *SendFrameEvent* self-message that takes care of the timing of sending frames has been triggered. This means that there are frames ready in the output buffer queue and the inter-frame gap after previous sending has elapsed. Function *sendFrame* is called.

Input parameters:

- (*cMessage**) **msg** — Pointer to the triggering message.
- *protected virtual void receiveFrameRoutine*(*EthFrame *frame*) — This method is automatically called upon every received frame within the private function *receiveFrame*. It gives the user a possibility to implement additional operations upon the arriving frame (ie. for derived classes). It is left blank in the current build. **Important:** Do not delete the frame in this method.

Input parameters:

- (*EthFrame**) **frame** — Pointer to the received frame.
- *protected int insertFrameToOutputQueue*(*EthFrame *frame*) — Purpose of this method is to allow the user to insert frames into the output queue (which is a private class variable) when designing custom actions.

Input parameters:

- (*EthFrame**) **frame** — Pointer to the frame that is to be sent.

2.1 OMNET++

Return value:

- SUCCESS when successful. Note: No other return value is present in the current build, but it may change in the future.
- *protected void* **sendEthFrame**(cMessage *action_msg) — This is an action handle routine that is registered in the class' constructor using the *registerAction* method. Its string identifier is a constant *ACTION_SEND_ETH_FRAME* which equals to "sendEthFrame". In this function an *EthFrame* is created and put into the output queue based on the parameters attached to the input *action_msg* parameter.

Input parameters:

- (cMessage*) **action_msg** — Pointer to a message containing given action's parameters loaded from the JSON file. An additional description is given in section 2.1.4.3.

Private methods

- *private void* **parseJson**() — This method is called in the *initialize* method and attempts to parse the JSON string specified by the *sourceJson* NED parameter. If the JSON structure is invalid, the given module will be blank without any events. Otherwise, information about this module's events are stored and all the necessary time triggers are scheduled. The JSON parser used is introduced in[8].
- *private void* **triggerHandler**(cMessage *trigger_msg) — This method is called whenever any event that is loaded from JSON file has triggered (both timed and non-timed triggers are included). Based on the information attached to the input parameter *trigger_msg*, corresponding actions are performed (using method *performAction*). If the trigger was time-based, it is rescheduled.

Input parameters:

- (cMessage*) **trigger_msg** — Pointer to a message containing information about the trigger.
- *private int* **performAction**(cMessage *action_msg) — This method looks up the desired action specified within the input parameter *action_msg* in registered actions. If a match is found, the action is executed.

Input parameters:

- (cMessage*) **action_msg** — Pointer to a message containing information about the action that is to be performed.

Return value:

- ACTION_UNKNOWN — No action matching the required one was found.
- ACTION_PERFORMED — Action has been performed.

- *private void* **sendFrame()** — This method sends the first frame in the output queue (FIFO) and searches for any events that may be triggered by this outgoing frame. If such trigger is found, method *triggerHandle* is called upon it. Also, if there are any frames left to be sent in the output buffer, next sending is scheduled.
- *private void* **receiveFrame**(EthFrame *frame) — This method receives given frame, emits statistical signals and searches for any events that may be triggered by this incoming frame. If such trigger is found, method *triggerHandle* is called upon it.

Input parameters:

- (EthFrame*) **frame** — Pointer to a frame that is being received.

2.1.4.3 Events

One of the most important mechanics introduced is the event system. Events are defined using a JSON structure (described in section 2.2) and are loaded in the *initialize* method in every *EndStation* simple module (or any derived one) present in the simulation network. Events are capable of parametrizing module's behaviour and are closely tied to custom actions implemented in the C++ class representing the given simple module (*EndStation* or any module derived from it). Event sequence diagram is depicted on figure 6. Each event consists of two parts:

- **Trigger** — A condition to be met for actions to execute.
- **Actions** — Methods to be executed when the event is triggered.

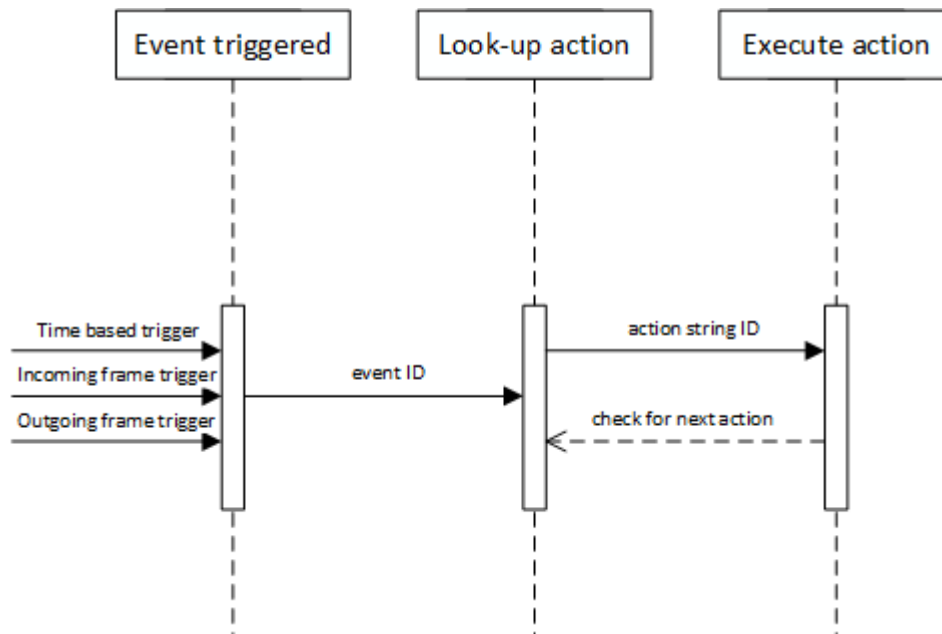


Figure 6: Events sequence diagram.

2.1 OMNET++

Trigger is responsible for execution of specified actions at defined conditions. At the moment there are 4 trigger types hard-coded in the simple module *EndStation* implementation (and its derivatives):

- **cyclic** — Timed trigger that repeats with a period given by a normal distribution $\mathcal{N}_t(\mu_t, \sigma_t^2)$. In the JSON structure of this trigger, user must specify two additional **<type-specific-parameters>** parameters:
 - (double) **mean** — The mean value μ_t of the normal distribution $\mathcal{N}_t(\mu_t, \sigma_t^2)$ [*ms*].
 - (double) **stdDev** — The standard deviation value σ_t of the normal distribution $\mathcal{N}_t(\mu_t, \sigma_t^2)$ [*ms*].
 - (double) **initPhase** [*optional*] — Time of the first trigger regardless of the period setting [*ms*]. If not specified, the trigger triggers after the first period given by the previous two parameters elapses.
- **acyclic** — Timed trigger that repeats with a period given by an uniform distribution $\mathcal{U}_t(a, b)$. In the JSON structure of this trigger, user must specify two additional **<type-specific-parameters>** parameters:
 - (double) **lowLimit** — The lower bound value a of the uniform distribution $\mathcal{U}_t(a, b)$ [*ms*].
 - (double) **highLimit** — The upper bound value b of the uniform distribution $\mathcal{U}_t(a, b)$ [*ms*].
 - (double) **initPhase** [*optional*] — Time of the first trigger regardless of the period setting [*ms*]. If not specified, the trigger triggers after the first period given by the previous two parameters elapses.
- **incomingFrame** — This trigger type is triggered by every incoming frame with the specified event ID that generated that frame. In the JSON structure of this trigger, user must specify one additional **<type-specific-parameters>** parameter:
 - (int) **refID** — Sensitive event ID of the incoming frame.
- **outgoingFrame** — This trigger type is triggered by every outgoing frame with the specified event ID that generated that frame. In the JSON structure of this trigger, user must specify one additional **<type-specific-parameters>** parameter:
 - (int) **refID** — Sensitive event ID of the outgoing frame.

Unlike actions, triggers are hard-coded and there is currently no mechanic to alter them or add new ones.

Actions are user defined methods that are performed whenever a trigger occurs. User-defined new actions are implemented using C++ in classes derived from the *EndStation* in a standardized way - the action handle method has a *void* return type and has one input parameter of the type (*cMessage**). An example of the action handle method declaration is shown in listing 2. Also, the user must register these new functions in the class constructor and assign a string ID to it (that serves as the *type* field in the JSON action definition). An example of the registration is shown in listing 1.

```
1 void anotherAction( cMessage *action_msg );
```

Listing 2: Action handle method declaration example.

The input parameter *action_msg* has parameters attached to it using data containers of *cPar* type (pairs string ID - value). The parameters are loaded from the JSON *actions* structure defined in section 2.2.3 as the `<type-specific-parameters>` parameters. Two additional parameters are always available - *PAR_EVENT_ID* which stores the event ID specified in the JSON file and *PAR_ACTION_ID* which is a unique number identifier of an action for given event ID (starting from 0). An example of how to get parameter value from the given action message is shown in listing 3. If the action is creating *EthFrames*, it is necessary to add to them a *cPar* parameter *PAR_EVENT_ID* so that they will trigger non-timed triggers.

```
1 long eventID = action_msg->par( PAR_EVENT_ID ).longValue();
2 double doubleVal = action_msg->par( "doubleValuePar" ).doubleValue();
```

Listing 3: Getting parameter value examples.

Whenever a new action is added, it is stored in a private variable (user cannot interact with it directly), which is in fact a map of pairs – handle function pointer and its string ID descriptor. When an event is triggered, the correct pair is looked up based on the given string ID (loaded from the JSON file) and the tied handle function is executed. User can specify unlimited amount of parameters tied to given action in the JSON configuration file. All these parameters will also be available within the action handle function.

There is one action implemented and registered in the *EndStation* simple module – **sendEthFrame** (briefly discussed in section 2.1.4.2). In the JSON structure, this action requires one following `<type-specific-parameters>` parameter:

- (int) **destination** — Identifier (ID) of the destination device.
- (int) **priority** [*optional*] — Priority of the frame. Valid values are 0-7 (inclusive). When not specified or an invalid value is entered, the default priority 0 is used.
- (int) **dataLength** [*optional*] — User data byte length of the frame. Valid values are 42-1500 (inclusive). When not specified or an invalid value is entered, a default value 46 bytes is used.

2.1.5 Simple module SimpleSwitch

This simple module models the behaviour of a simplified VLAN aware Ethernet switch. The module has a selectable number (theoretically unlimited) of ports and a selectable number of output priority queues for each of the ports (up to 8). This setting is done globally for all of the ports. Frames from the highest priority level queue are always sent first. There is a hard-coded table (see table 1) defined in the module that puts frames to the correct output priority queue based on the number of the priority queues present and the priority of the given frame itself. This mapping is recommended by the IEEE 802.1p standard[3]. A function scheme is depicted on fig 7.

The switch also implements a self-learning MAC address lookup table. The table is static and does not implement any decay due to the assumption of a static topology. If the destination address is not yet listed in the table, the switch floods frames to all the ports except for the one on which they arrived.

Switching (forwarding) delay of the frame propagation through the switch from input to output queue is modelled using normal distribution $\mathcal{N}_{SD}(\mu, \sigma^2)$.

User can connect any module defined within the project to the *SimpleSwitch* module.

Frame priority	Number of output queues present							
	1	2	3	4	5	6	7	8
0 (default)	0	0	0	1	1	1	1	2
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1
3	0	0	0	1	1	2	2	3
4	0	1	1	2	2	3	3	4
5	0	1	1	2	3	4	4	5
6	0	1	2	3	4	5	5	6
7	0	1	2	3	4	5	6	7

Table 1: Priority queue mapping based on number of queues and frame priority.

2.1.5.1 NED definition

The NED file *simpleSwitch.ned* defining this simple module is located in the root folder of the project. Given module is connected to the outer world by an array of full-duplex ports (gates in the NED terminology).

Parameters The following parameters have been defined:

- (double) **minFrameSpacingTime** @unit(s) — Inter-frame gap. It is the minimal time gap between two consecutive frames. This parameter has a default value of

0.96 μs which corresponds to the 100Mbit Ethernet standard[3]. The special tag *@unit(s)* signs that the expected value is entered using a time unit (framework takes care of any conversions).

- (double) **switchingDelayMean** @unit(s) — Mean value μ of the normal distribution $\mathcal{N}_{SD}(\mu, \sigma^2)$. Default value $10\mu s$.
- (double) **switchingDelayStdDev** @unit(s) — Standard deviation value σ of the normal distribution $\mathcal{N}_{SD}(\mu, \sigma^2)$. Default value $1\mu s$.
- (int) **numPorts** — Number of ports available to connections. Default value 8.
- (int) **numPrioQueues** — Number of output priority queues on each port. Valid values are 1-8 inclusive. Default value 4 queues.
- (int) **inQueueSize** — Size (number of frames it can hold) of the input queue on each port. Default value 256 frames.
- (int) **outQueueSize** — Size (number of frames it can hold) of each of the output priority queues on each port. Default value 256 frames.

Note: All parameters with defined default value don't have to be specified when creating network, but can be overwritten if desired.

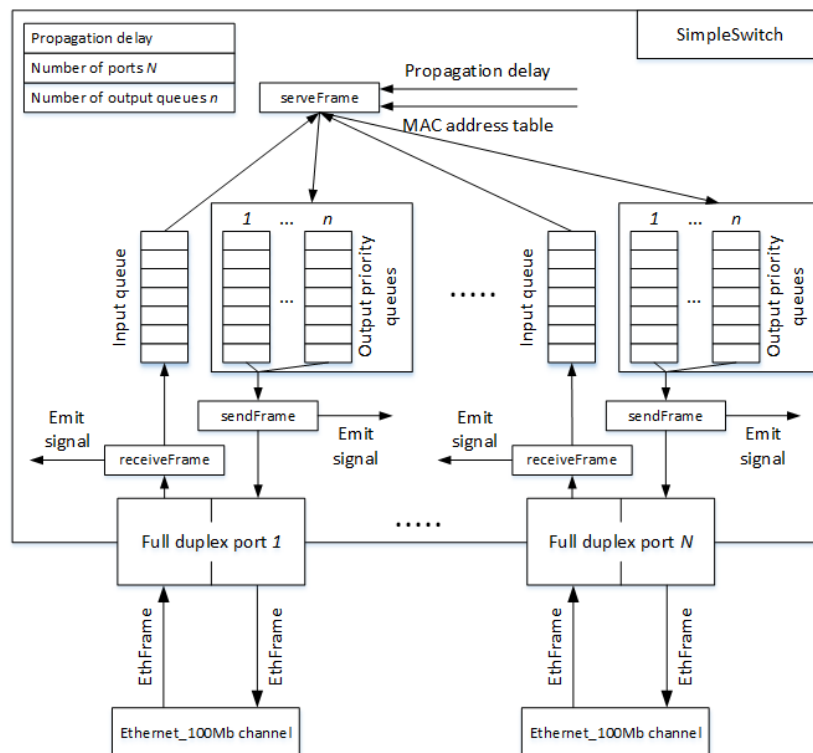


Figure 7: SimpleSwitch simple module.

2.1 OMNET++

Statistical signals There are three statistical signals defined which record different traffic on each port:

- **portTrafficIn_*** — frame size of the arriving frames on given port.
- **portTrafficOut_*** — frame size of the outgoing frames on given port.
- **portTrafficTotal_*** — frame size of both outgoing frames and arriving frames on given port.

The wild-card '*' stands for given port number. Only used ports' statistics are recorded. Port numbering begins at 0. For each statistical signal, there is a statistic template with the following scalar measurements:

- **mean** — Mean value of the frame data size.
- **sum** — Total frame data size processed.
- **count** — Number of frames processed.

Statistical signal **portTrafficTotal_*** records also one vector measurements:

- **vector(count)** — Vector signal that records the total number of processed frames on given port in time (with timestamps).

Note: Only one statistical signal uses a vector recording. This is to reduce output file size. User can easily alter the recorded measurements in the NED file - no compilation is needed.

2.1.5.2 C++ class

The class *SimpleSwitch* is defined in files *simpleSwitch.h* and *simpleSwitch.cc* located in the *src* folder. It is derived from a generic simple module class *cSimpleModule* and overrides several of its methods.

Public methods

- **SimpleSwitch()** — Class constructor. Initializes the MAC address lookup table.
- **~SimpleSwitch()** — Class destructor. Cleans up dynamically allocated queues.

Protected methods

- *protected virtual void* **initialize()** — This method is automatically called after the class constructor. It initializes class variables, loads NED parameters and creates statistical signals with defined statistical templates defined in the NED file.
- *protected virtual void* **handleMessage(cMessage *msg)** — This method overwrites the implementation of its parent class *cSimpleModule*. It is called whenever

a self-message time elapses or message arrives on the input port. Three possible variants may occur:

- There is an *EthFrame* message (which inherits from *cMessage* - see section 2.1.2) arriving on the input port. In this case, the input *cMessage* is retyped to *EthFrame* and is received by calling function *receiveFrame* upon it.
- *serveInputFrame* self message that takes care of the timing of frame propagation and forwarding has been triggered for given port. This means that there are frames ready in the input queue on some port and the propagation (bridge) delay after previous forwarding for given port has elapsed. Function *serveFrame* is called.
- *SendFrame* self message that takes care of the timing of sending frames has been triggered. This means that there are frames ready in some of the output priority queue on some port and the inter-frame gap after previous sending for given port has elapsed. Function *sendFrame* is called.

Input parameters:

- (cMessage*) **trigger_msg** — Pointer to the triggering message.

Private methods

- *private void* **sendFrame**(int port, cMessage *msg) — This method sends the first frame in the highest level non-empty priority output queue (FIFO) for given port. If there are any frames left to be sent in any of the the output queues, next sending for the given port is scheduled. Also, the statistical signals are emitted.

Input parameters:

- (int) **port** — Number of the sending port.
- (cMessage*) **msg** — Pointer to a *sendFrame* self message to be rescheduled if there are more frames to be sent.
- *private void* **receiveFrame**(EthFrame *frame) — This method receives given frame by placing it in the input queue of the port on which it arrived, emits statistical signals and schedules self message *serveInputFrame* to model the bridge delay (if not already scheduled). If the input queue is full, the frame is discarded.

Input parameters:

- (EthFrame*) **frame** — Pointer to a frame that is being received.
- *private void* **serveFrame**(int port) — This method forwards the first frame in the given port's input queue. The destination port is determined using the MAC address lookup table. If the destination address cannot be found within the table or the destination is a multicast address, all the ports except the one on which

2.1 OMNET++

the frame arrived, are flooded. Also, the frame's source address is added as a new table entry with the corresponding port tied to it (if not already present in the table). The frame is then placed into one of the output priority queues based on the frame's VLAN priority and the total number of output priority queues present in the switch (as seen in table 1). If the given priority output queue is full, the frame is discarded.

At the simulation start, the MAC address table is empty, therefore the first few communication cycles will generate some extra traffic, because of the flooding. This effect also occurs in real switches. Since we usually consider steady state of the communication this effect can be neglected in a long term simulations or avoided completely with a *warm-up* period. The *warm-up* period specifies time before which no statistical signals are recorded. It is set in the *simulation* part of the JSON definition described in section 2.2.2.

Input parameters:

- (int) **port** — Number of the port being served.

2.1.6 Compound module **GenericDevice**

This is a complex module that combines two already defined simple modules - one *EndStation* and one *SimpleSwitch*. The functionalities of both simple modules are preserved and merged. This applies to the C++ implementation as well as to the NED definition. These modules are connected together using an ideal channel (unlike any other connection that is modelled by the *ethernet_100Mb* channel) with an unlimited data rate. This models a fast bus that will be used in a real device. The connection consumes the *EndStation*'s only port and one "hidden" port (port 0) of the *SimpleSwitch*. Module's scheme is depicted on figure 8.

GenericDevice compound module aims to model a generic addressable device with multiple ports and switching capabilities. It is capable of substituting both individual simple modules making them obsolete within the network topology design, yet the module can be connected to any other module defined within the project.

Compound modules are defined solely using a single NED file and cannot implement any additional C++ functionality.

2.1.6.1 NED definition

The NED file *genericDevice.ned* defining this compound module is located in the root folder of the project. Given module is connected to the outer world by an array of full-duplex ports (gates in the NED terminology).

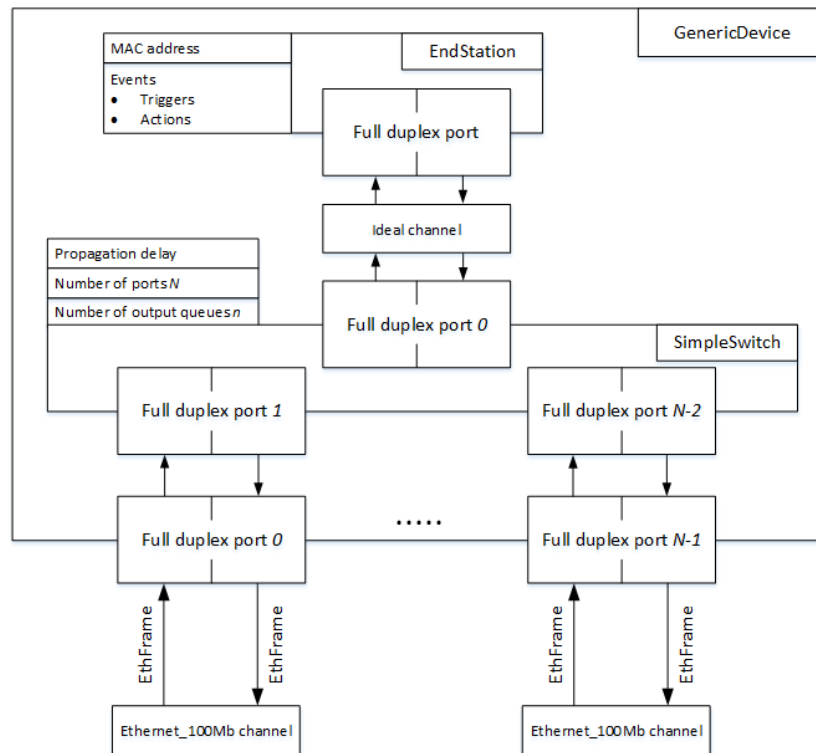


Figure 8: GenericDevice compound module.

Parameters All parameters of both sub-modules must be specified when using this compound module. Module *GenericDevice* defines only one "own" parameter:

- (int) **numPorts** — Number of free ports that can be used to connect other modules. The number does not include the hidden port that connects the sub-modules. This parameter sets the **numPorts** parameter of inner *SimpleSwitch* module incremented by 1 (to compensated the hidden port). Default value 2.

For parameters defined in the simple stations refer to corresponding sections (*EndStation* - section 2.1.4.1, *SimpleSwitch* - section 2.1.5.1).

Statistical signals All statistical signals defined within each sub-module are merged as well. For their definition refer to corresponding sections (*EndStation* - section 2.1.4.1, *SimpleSwitch* - section 2.1.5.1).

2.1.7 Derived modules

In this section, the process behind the creation of a new derived module using the OMNeT++ IDE is described. Also, an example simple module *DerivedEndStation* and compound module *DerivedDevice* are created and provided in the Profinet simulation model.

2.1 OMNET++

Currently, only the simple module *EndStation* is adjusted to support user-created simple modules derived from it. When such a module is created, it is also possible to form a compound module combining it with the *SimpleSwitch* module. The process is described in the following steps:

- Creating the derived simple module *DerivedEndStation* - section 2.1.7.1.
- Combining the created module with a *SimpleSwitch* module to create the compound module *DerivedDevice* - section 2.1.7.2.
- Compilation - section 2.1.7.3.

A class diagram including the *DerivedEndStation* module is depicted on figure 9.

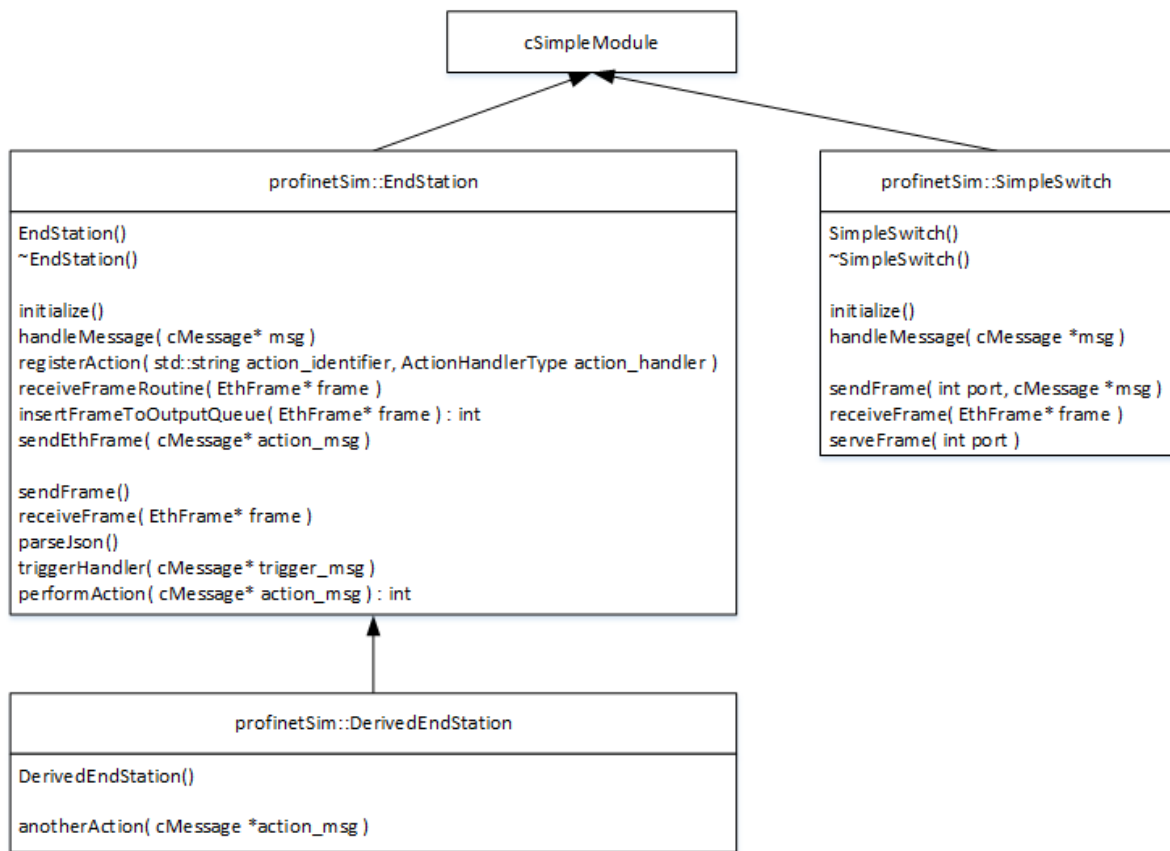


Figure 9: Overall class diagram.

2.1.7.1 Simple module creation

To create a new blank simple module, right click on the *profinetSim* project in the *Project Explorer* window, which is usually located on the top left corner of the OM-NeT++ IDE, and select *New* → *Simple Module*. In the pop-up window enter the desired module name (in our example *DerivedEndStation*) and press *Next*. After that, select *A simple module* option and press *Finish*.

Total of 3 files are automatically created in the root folder - NED file *derivedEndStation.ned*, C++ header file *derivedEndStation.h* and C++ source file *derivedEndStation.cc*. It is advised to move the C++ files to the *src* folder located in the project root folder to preserve established file structure. Note: User may choose to create the files manually, but the final code should look the same.

NED definition Firstly, let's deal with the NED file *derivedEndStation.ned*. To ensure compatibility with the parent simple module *EndStation*, the same structure (parameters, gates, statistical signals) must be defined. User may add new parameters or statistical signals freely, yet those will not be counted with by the designed API (described in section 2.3). The most convenient way is to simply copy the *EndStation*'s NED file definition and change the module's name. The default edit mode of the IDE is the graphical one. To switch to *Source* view, select the *Source* tab at the bottom left corner of the main window. The NED file code is shown in listing 4 (comments have been omitted).

```

1 package profinetSim;
2
3 simple DerivedEndStation
4 {
5     parameters:
6         double minFrameSpacingTime @unit(s) = default(0.96us);
7         int macAddressOui = default(0x400000);
8         int macAddressOap;
9         string sourceJson = default("");
10
11         @signal[frameDelayByPriority_*](type= simtime_t);
12         @statisticTemplate[frameDelayByPriority](record=mean, min, max
13         , count, vector);
14     gates:
15         inout port;
16 }
```

Listing 4: *DerivedEndStation* NED file definition

C++ header file By default, the parent class of a new simple module is the class *cSimpleModule*. To change the parent class, the *endStation.h* header file has to be included and the *cSimpleModule* class name has to be changed to *EndStation*. Also, to preserve the *EndStation*'s behaviour, the automatically added functions *handleMessage* and *initialize* has to be deleted. If the purpose of this derived module is to add additional actions, the only methods to declare (and implement) are the constructor and the action handle routine (method). The action handle routines have a fixed format (see listing 2) that is described in section 2.1.4.3. An example of such defined header file is shown in listing 5. User may also define any other methods or override existing ones if desired.

2.1 OMNET++

```
1 #ifndef __PROFINETSIM_DERIVEDENDSTATION_H_
2 #define __PROFINETSIM_DERIVEDENDSTATION_H_
3
4 #include <omnetpp.h>
5 #include "endStation.h"
6
7 namespace profinetSim {
8
9 class DerivedEndStation : public EndStation
10 {
11     public:
12         DerivedEndStation();
13
14     protected:
15         void anotherAction( cMessage *action_msg );
16 };
17
18 }
19
20 #endif
```

Listing 5: *DerivedEndStation* header file definition

```
1 namespace profinetSim {
2
3 Define_Module(DerivedEndStation);
4
5 DerivedEndStation::DerivedEndStation(){
6     std::string action_id = "anotherAction";
7     ActionHandlerType action_handler = std::bind( &DerivedEndStation::
8         anotherAction, this, std::placeholders::_1);
9
10     registerAction( action_id, action_handler );
11 }
12
13 void DerivedEndStation::anotherAction( cMessage *action_msg ){
14     //example of parameter load
15     int intPar = (int) action_msg->par( "intField" ).longValue();
16     double doublePar = action_msg->par( "doubleField" ).doubleValue();
17     std::string strPar = action_msg->par("strField").stringValue();
18
19     //create blank frame and put it to the output queue
20     EthFrame *frame = new EthFrame("frame");
21     frame->setByteLength( 64 + FIXED_FRAME_PART_SIZE );
22     frame->setSrcAdress( 0 );
23     frame->setDestAdress( 1 );
24     insertFrameToOutputQueue( frame );
25 }
```

Listing 6: *DerivedEndStation* source file definition

C++ source file This file implements the methods declared in the header file. In the case of our example module, the action handler method *anotherAction* has been implemented and also registered in the constructor with a string identifier "anotherAction". The example code is shown in listing 6. Note, that all of the parent's *public* and *protected* methods are available for use (see section 2.1.4.2 for the *EndStation* C++ implementation).

2.1.7.2 Compound module creation

To combine the designed derived simple module *DerivedEndStation* with a *SimpleSwitch* simple module only a single NED file is needed.

To create a new blank compound module, right click on the *profinetSim* project in the *Project Explorer* window, which is usually located on the top left corner of the OMNeT++ IDE, and select *New* → *Compound Module*. At the pop-up window enter the desired module name (in our example case *DerivedDevice*) and press *Next*. After that, select *An empty compound module* option and press *Finish*.

If no NED parameters were added in the NED definition of the *DerivedDevice* module, simply copy the source code of the *GenericDevice* compound module and substitute *EndStation* with *DerivedEndStation*. The NED code of the *DerivedEndStation* is shown in listing 7.

```

1 package profinetSim;
2
3 module DerivedDevice
4 {
5     parameters:
6         int numPorts = default(2);
7
8     gates:
9         inout port[numPorts];
10    submodules:
11        simpleSwitch_i: SimpleSwitch {
12            numPorts = sizeof(port)+1;
13        }
14        endStation_i: DerivedEndStation {
15        }
16    connections allowunconnected:
17        EndStation_i.port <--> simpleSwitch_i.port[0];
18        for i=0..sizeof(port)-1 {
19            simpleSwitch_i.port[i+1] <--> port[i];
20        }
21 }

```

Listing 7: *DerivedDevice* NED file definition.

2.2 JSON DEFINITION

2.1.7.3 Compilation

When the derived modules are defined, the Profinet simulation model project has to be recompiled. To start the compilation, right click on the *profinetSim* project in the *Project Explorer* window, which is usually located on the top left corner of the OMNeT++ IDE, and select *Build Project*.

The compilation text output is printed in a *Console* tab located under the main editing window. If the compilation is successful, the tool is ready to be used.

2.2 JSON definition

In order to provide the user with a clear and comprehensive way to capture network topologies and to enter the required parameters, a JSON structure has been designed. The JSON file contains information about the simulation parameters, the network topology definition and the device parameters (with events). Optionally, another JSON file can be used to define custom events. This JSON file shares the same structure as an "inline" definition of custom events introduced in section 2.2.3. Also, a brief JSON format overview is described in section 2.2.1.

The highest level of the main JSON file's structure contains three main parts:

- Simulation parameters - section 2.2.2.
- Custom events - section 2.2.3 [*optional*].
- Network definition - section 2.2.4.
 - Nodes - section 2.2.4.1.
 - Links - section 2.2.4.2.

2.2.1 JSON format overview

JSON is a lightweight data-interchange format[5] that can easily be read by humans and parsed independently by various programming languages (in our case C++). User data can be inserted in two types of containers:

- **Objects** — Objects are wrapped by braces "`{}`" and contain data with a following notation: "`name`": *value*. Where *name* is a string identifier. Multiple entries are separated by commas `,`.
- **Arrays** — Array is a construction wrapped by brackets "`[]`" that can hold multiple items that usually share the same value type. Entries are separated by commas `,`.

Valid value types are: *object*, *array*, *decimal number* and *string*. Also several special values may be entered: *null*, *true* and *false*. It is clear, that values of type *object* and *array* may be arbitrarily nested. The highest level value type is an *object* without any string identifier (name).

An example code demonstrating the JSON structure is listed in the code listing 8.

```

1 {
2   "stringPar" : "hello world",
3   "numberPar" : 1.58,
4   "arrayPar" : [{ "anotherPar" : 1,
5                   "objectPar" : { "type" : "generic",
6                                   "value" : 1.11
7                                   },
8                   "lastPar" : "bye"
9                 },
10  { "anotherPar" : 2,
11    "objectPar" : { "type" : "special",
12                  "value" : 8.11
13                  },
14    "lastPar" : "byebye"
15  }
16 ]
17 }

```

Listing 8: JSON structure example

2.2.2 Simulation parameters

Simulation parameters are given within the top level JSON object under a string identifier *simulation*. The parameters are (the expected data type for the C++ parser is stated within the parentheses before every parameter):

- (string) **name** — Name of the network. This parameter is used as a name of a network topology NED file generated automatically by the defined API.
- (double) **duration** — Desired duration of the simulation given in $[ms]$. It is valid up to three decimal digits.
- (double) **warmUp** *[optional]* — Warm-up period of the simulation given in $[ms]$. It is valid up to three decimal digits. This parameter specifies the time at the beginning of the simulation in which no statistical signals are recorded. Note that this will effectively shorten the simulation duration (result-wise). The first recorded signal will have a timestamp greater or equal to the *warmUp* period. If not specified, default value 0 is used.
- (string) **inputs** — Path to a folder containing the designed OMNeT++ simulation model.

2.2 JSON DEFINITION

- (string) **customEvents** — Location of custom events definition. Custom events allow to define events that may be used by multiple devices or to simplify the nodes' definitions. There are two possible entries - either a keyword "inline" meaning that custom events are defined within the main JSON file under an object labelled *customEvents* (see section 2.2.3) or a generic string that is represented as a path to an external JSON file defining the custom events. The custom events definition is optional and user may choose to define all the events within the respective nodes. In this case, keyword "inline" has to be used (the object *customEvents* in the main JSON file can be left out blank or not present at all). If the external JSON file option is selected, it is important to make sure that the specified file path is valid from the perspective of the API application invoking the simulation when using relative pathing.
- (string) **output** — Output file name containing simulation results.

2.2.3 Custom events

This part introduces the JSON structure of events. The structure is common for "inline" custom events, external JSON file custom events as well as for events defined within each node (device) - these are simple labelled *events* (see section 2.2.4.1). Object named *customEvents* is located at the top level object of the main JSON file (for "inline" notation) and contains an *array* of the following values:

- (int) **id** — An unique event identifier. It is used as a reference ID to link given custom event to a certain node (device). Also serves as a reference ID for incoming and outgoing frame triggers.
- (string) **name** [*optional*] — Name of the event. This parameter is not used in any way and serves only for user clarity purpose.
- **trigger** — Condition on which the given event executes actions tied to it. Triggers are introduced in section 2.1.4.3. It is an *object* value containing the following values:
 - (string) **type** — Trigger type. Four trigger types have been defined - *cyclic*, *acyclic*, *outgoingFrame* and *incomingFrame*.
 - **<type-specific-parameters>** — Additional parameters required by the given type of the trigger.
- **actions** [*optional*] — *Array* of actions that will be performed each time this event is triggered. Although this parameter is optional, it does not make sense not to specify any actions since the event would be "blank". If multiple actions are specified, they are executed in the same order as defined here. Actions are introduced in section 2.1.4.3. Each item (*object*) of the *array* must contain the following parameters:

- (string) **type** — Action type. Actions supported will differ across the defined modules. Currently only one action is defined in the *EndStation* simple module - *sendEthFrame*.
- **<type-specific-parameters>** — Additional parameters required by the given type of the action.

```

1 {
2   ...
3   "customEvents" : [{ "id"       : 0,
4                       "name"     : "PN_IO communcation",
5                       "trigger"  : { "type"       : "cyclic",
6                                     "mean"       : 2,
7                                     "stdDev"    : 0.01,
8                                     "initPhase"  : 0.5
9                                     },
10                      "actions"  : [{ "type" : "sendEthFrame",
11                                      "destination" : 1
12                                      },
13                                      { "type" : "sendEthFrame",
14                                      "destination" : 2
15                                      }
16                                      ]
17                      }
18   ]
19   ...
20 }

```

Listing 9: Custom events example.

2.2.4 Network definition

JSON *Object* named *network* is located at the top level object of the main JSON file and contains two *arrays* to define the simulation network - *nodes* and *links*.

2.2.4.1 Nodes

Each item of the *nodes array* represents a single device to be modelled in the network. It captures all of the necessary parameters of the respective NED modules using the following structure:

- (int) **id** — An unique node identifier. Serves as the lower 3 bytes of the MAC address for addressable devices (the NED parameter *macAddressOap*). For NED parameters concerning the MAC address refer to *EndStation*'s NED definition in section 2.1.4.1.
- (string) **type** — Type of the node (device). The value of this parameter must match (case-sensitively) one of the defined modules (*EndStation*, *SimpleSwitch*,

2.2 JSON DEFINITION

GenericDevice or any other user defined modules) to determine the node's behaviour and supported actions.

- (int) **macAddressOui** [*optional*] Upper 3 bytes of the MAC address. If not specified, the default value is provided by the corresponding NED parameter. For NED parameters concerning the MAC address refer to *EndStation*'s NED definition in section 2.1.4.1.
- (string) **name** [*optional*] — Node's name. If defined, it is used as the node's name within a NED file describing the network when it is generated using API functions.
- **switch** [*optional*] — Contains parameters regarding the switch part of the device (if present in the specified module). If not specified, the default values of corresponding NED parameters are used (refer to NED file definition of *SimpleSwitch* module in section 2.1.5.1). It is a JSON *object* value containing the following:
 - (int) **numPorts** — Number of free ports available for connection. Does not include the inner port used in compound module *GenericDevice*.
 - (int) **numPrioQueues** — Number of the output priority queues on each port.
 - **switchDelay** — The switch (bridge/propagation) delay is modelled using a normal distribution of probability $\mathcal{N}_{SD}(\mu, \sigma^2)$ - this JSON *object* contains a pair of parameters describing this normal distribution.
 - * (double) **mean** — Mean (expected) value μ of the given normal distribution [*ms*].
 - * (double) **stdDev** — Value of the standard deviation σ of the given normal distribution [*ms*].
- **endStation** [*optional*] — Contains parameters regarding the end point part of the device (if present in the specified module). At the moment, this JSON *object* contains only events definition:
 - **events** [*optional*] — *Array* of the event definitions. There are two possible type of entries:
 - * **regular** — The event is fully described here. The structure is the same as introduced under the *customEvents* *object* in section 2.2.3.
 - * **custom** — The event has been defined as a custom event (either using "inline" notation or an external JSON file). In this case a different structure of an event item is required:
 - (int) **refID** — Custom event reference ID as stated in the filed *id* of the custom event's definition.

- **actions** *[optional]* — An *array* of additional actions to be added to given custom event. The JSON structure of actions has been introduced in the section 2.2.3.

An example of nodes *array* is shown in listing 10.

```

1 "nodes": [
2     {
3         "id": 0,
4         "type": "GenericDevice",
5         "name": "PLC",
6         "switch": {
7             "numPorts": 2,
8             "switchDelay": {
9                 "mean": 0.005796,
10                "stdDev": 0.000046
11            }
12        },
13        "endStation": {
14            "events": [
15                {
16                    "refID": 45,
17                    "actions": [
18                        {
19                            "type": "sendEthFrame",
20                            "destination": 1
21                        }
22                    ]
23                }
24            ]
25        }
26    }
27 ]

```

Listing 10: Node's JSON definition example.

2.2.4.2 Links

Device connections are captured using the JSON *array* named *links*. Each connection is full duplex and has to be listed only once (the end points are interchangeable). The structure of each *link object* is given as:

- (double) **source** — Start point of the connection. The value must obey a special notation: **<module-id>.<port-number>**.
- (double) **target** — Endpoint of the connection. The value must obey a special notation: **<module-id>.<port-number>**.

2.3 PROFINET SIMULATION API

```
1 "links": [  
2     {  
3         "source": 0.1,  
4         "target": 1.1  
5     }  
6 ]
```

Listing 11: Link's JSON definition example.

Note, that the port numbering starts at 1. An example of links *array* is shown in listing 11.

2.3 Profinet simulation API

A C++ pre-compiled shared library has been designed to provide an API support to the designed Profinet simulation model. The API contains methods to parse the JSON file and to automatically create files needed for the simulation execution based on the loaded topology and parameters. Also, a method to execute prepared simulation is implemented.

The API is currently available under Linux operating systems only, which may be subject to change in future builds.

2.3.1 API implementation

The API is implemented as a C++ class *ProfinetSimApi* within a defined namespace *profinet_sim_api*. In this section a brief overview of the class' methods is described. An example usage of the API is discussed in section 2.3.2 and shown in listing 12.

Public methods

- *public* **ProfinetSimApi**() — Class constructor. Defines default values of the simulation parameters.
- *public int* **loadInputFile**(std::string inputFileName) — This method loads the given JSON file and creates a network NED file based on the parsed topology. Also, the method loads the simulation parameters, stores them within the object and creates an INI file corresponding to the generated NED file. Both of the created files are stored within the simulation model's folder specified by the *inputs* simulation parameter value of the source JSON file. The files are named according to the *name* simulation parameter value of the source JSON file with an appropriate suffix (<name>.ned for the network topology NED file and <name>.ini for the simulation initialization file). Simulation parameters of the JSON file are

discussed in section 2.2.2. If all is successful, the simulation is ready to be executed. User can use the *isSimulationReady* method to check if the simulation is ready. The JSON parser used is introduced in[8].

Input parameters:

- (std::string) **inputFileName** — File name of the input JSON file.

Return value:

- STATUS_OK — JSON parsing and files creation was successful.
- ERR_IO — A file could not be opened/created.
- ERR_PARSE — JSON file structure is not valid.
- *public int* **executeSimulation()** — This method runs the loaded simulation (if loaded successfully) using *system* function by executing automatically created shell command. The simulation is done using a *Cmdenv* (command-line environment, see OMNeT++ manual[7] for further information) which takes over of the user program until it is finished. The simulation progress is printed out in the standard output roughly every 2 seconds by the *Cmdenv*. After the simulation is done, the recorded statistical signals are saved to the output files located in *results* folder which is created in the root folder of the Profinet simulation model.

Return value:

- STATUS_OK — Simulation was successful and the output files have been created.
- ERR_SIM_NOT_RDY — Simulation was not ready, it must be properly loaded using the *loadInputFile* method first.
- other — Other error codes that were returned by the *system* call. Refer to OMNeT++ documentation for further information[7].
- *public std::string* **getNetworkName()** — Returns the loaded value of the *name* simulation parameter.

Return value:

- (std::string) — The simulation network name.

- *public std::string* **getInputsFolderPath()** — Returns the loaded value of the *inputs* simulation parameter.

Return value:

- (std::string) — The location (path) of the Profinet simulation model.

- *public double* **getSimulationDuration()** — Returns the loaded value of the *duration* simulation parameter.

Return value:

2.3 PROFINET SIMULATION API

– (double) — The duration of the simulation [*ms*].

- *public double* **getWarmUpPeriod()** — Returns the loaded value of the *warmUp* simulation parameter.

Return value:

– (double) — The warm-up period of the simulation [*ms*].

- *public bool* **isSimulationReady()** — Returns if a simulation is ready to be executed.

Return value:

– true — The simulation has been loaded and is ready for execution.

– false — The simulation is not ready for execution.

- *public int* **getStatus()** — Returns the last error code that occurred in any of the API functions. After this function is called, the status is reset to STATUS_OK.

Return value:

– STATUS_OK — No error since the last call of this method.

– ERR_IO — I/O error occurred (a file could not be read/created).

– ERR_PARSE — JSON parse error - bad structure.

– ERR_SIM_NOT_RDY — There was an attempt to run the simulation that was not properly loaded.

2.3.2 API usage

To use the API, the pre-compiled shared library *libprofinetSimApi.so* file has to be added to the user project's linker libraries. The provided header file *ProfinetSimApi.h* has to be added to the compiler includes and also has to be included in the source files where the API functions are called. An example C++ client program using the API has been implemented using Eclipse IDE and GCC toolchain under the Ubuntu 14.04 Linux distribution (*profinetSimApiConsoleApp*).

To add the shared library to the linker libraries in the Eclipse IDE, right click your project and select *Properties*. In the *Properties* window select *C/C++ Build* → *Settings* → *GCC C++ Linker* → *Libraries* tab and add the folder containing the shared object *libprofinetSimApi.so* as is depicted on figure 10.

Similarly, to add the include path containing the header file, select *C/C++ Build* → *Settings* → *GCC C++ Compiler* → *Includes* in the *Properties* window.

When the environment is set, a simple example of API usage is shown in the listing 12.

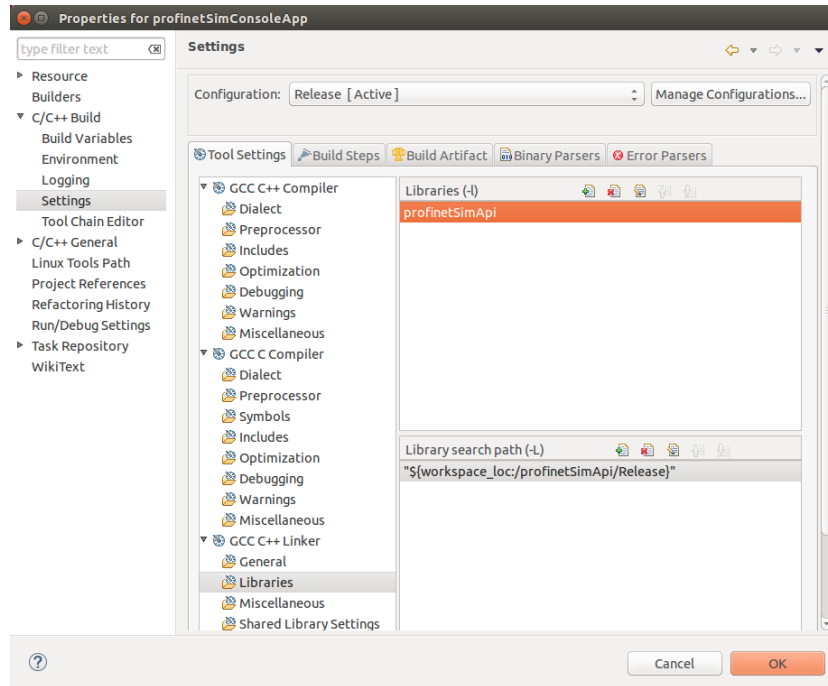


Figure 10: Adding API shared library to the linker.

```

1 #include "ProfinetSimApi.h"
2
3 using namespace std;
4 using namespace profinet_sim_api;
5
6 /*
7  * Example of omnet API library usage
8  */
9 int main() {
10     string inputFile = "/home/user/workspace/omnetApi_client/data/
input/network.json";
11     ProfinetSimApi *api = new ProfinetSimApi();
12
13     api->loadInputFile( inputFile );
14     if( api->isSimulationReady() ){
15         api->executeSimulation();
16     }
17
18     printf( "\n\nExit code: %d", api->getStatus() );
19
20     return 0;
21 }

```

Listing 12: API example usage.

2.3 PROFINET SIMULATION API

2.3.3 Console application

A console application *profinetSimApiConsoleApp* has been implemented under the Ubuntu Linux distribution. It takes one input when called - a string representing the path to the main JSON source file. The application will attempt to parse the given JSON, create the network topology NED file as well as the INI file. If the parsing is successful it will also run the prepared simulation. API's exit code is printed out in the standard output. An example of use is shown in listing 13.

```
1 ./profinetSimApiConsoleApp /home/user/workspace/myJson.json
```

Listing 13: API console application usage.

Important: User has to make sure the shared library *libprofinetSimApi.so* is properly loaded within the system. One possible way of doing this is to create a *.conf file in the */etc/ld.so.conf.d* system folder containing a file path to the folder containing the shared library and then bind it using the command *ldconfig* as a superuser.

Part 3

Modelling a network

3.1 Physical network

A physical Profinet network was available for testing and model verification. The network consists of the following devices (the topology is depicted on figure 11):

- PN IO Controller - Siemens CPU 315-2 PN/DP
- Switch 1 (PN IO Device) - Harting FTS 3082-ASFP
- 3x PN IO Device - Siemens IM151
- Switch 2 (PN IO Device) - Siemens Scalance XF204

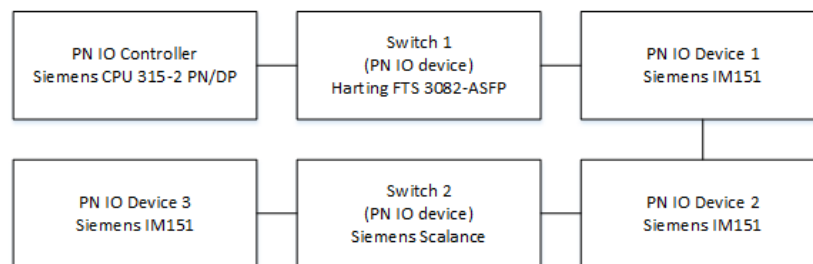


Figure 11: Test network topology.

The project hardware configuration sets the following Profinet real-time communication among the devices:

- Cyclic communication in both directions between the Controller and Siemens IM151 IO devices every 2 *ms* using a producer-consumer system.
- Cyclic communication in both directions between the Controller and both switches every 128 *ms* using a producer-consumer system.

Apart from the defined PN communication which uses a PNIO protocol with VLAN-tagged frames with priority 6, other protocols are also present (at steady state) - LLDP, STP and ARP. These protocols formed about ~0,2% of the total communication in the test topology and their frames are not explicitly VLAN-tagged which means that they use the default priority value of 0.

3.2 Setting the parameters

In order to simulate the network, the source JSON file has to be set up. Apart from the known characteristics such as link connections or the number of the devices' ports, this includes the switching parameters of the devices as well as their communication parameters that form the events definitions. Both of these are discussed in this section.

All of the devices have been modelled using the *GenericDevice* module defined in section 2.1.6. This means, that they have to specify both the switching parameters and the events definitions.

3.2.1 Switching parameters

To determine accurate switching behaviour, the switching (forwarding) delay must be set. As discussed in the section 2.1.5, this delay is modelled using the normal distribution $\mathcal{N}_{SD}(\mu, \sigma^2)$. To acquire the parameters, a measurement has been designed.

To measure the delay, two identical and transparent Ethernet taps were connected between the measured device and the adjoining devices in the network topology. The full duplex traffic flowing through both taps was recorded using a Siemens BANY device and send to PC to be stored in a **.pcap* file. All the recorded frames are stored with their corresponding timestamps with a time precision of *ns*. The measuring network scheme for the *Switch 1* device measurement is shown on figure 12. To ensure enough data to accurately compute the desired parameters, 5 minute measurements were recorded. This translates to several hundreds of thousands recorded frames depending on the selected measured device.

For data analysis a standalone program *packetAnalyser* was implemented in C++ using a libpcap[9] library under the Ubuntu Linux distribution. The analysis program is not included in the Profinet simulation model and is briefly discussed in the Appendix A. The algorithm itself is fairly simple - timestamps of two closest consecutive frame recordings sharing the same source and destination addresses as well as identical program counters are subtracted to acquire the switching delay time of the given frame. The two frames are in fact recordings of one single frame recorded by the respective taps as the frame propagates through the network. Both directions are considered to ensure the measurement is valid and that the two taps are in fact as close to identical as possible. Generally speaking, the measurements showed that the difference between results from both directions can be neglected. The timestamp differences were stored for all of the frames propagating through the given device to form a sample for statistical analysis. The sample mean and sample variance (and standard deviation) were computed and used in the JSON file capturing the test network.

All of the non-end point devices in the test topology were measured using this method. The end point devices' (*PN IO Controller* and *PN IO Device 3*) parameters were estimated based on the rest of the measurements to preserve the topology and hardware

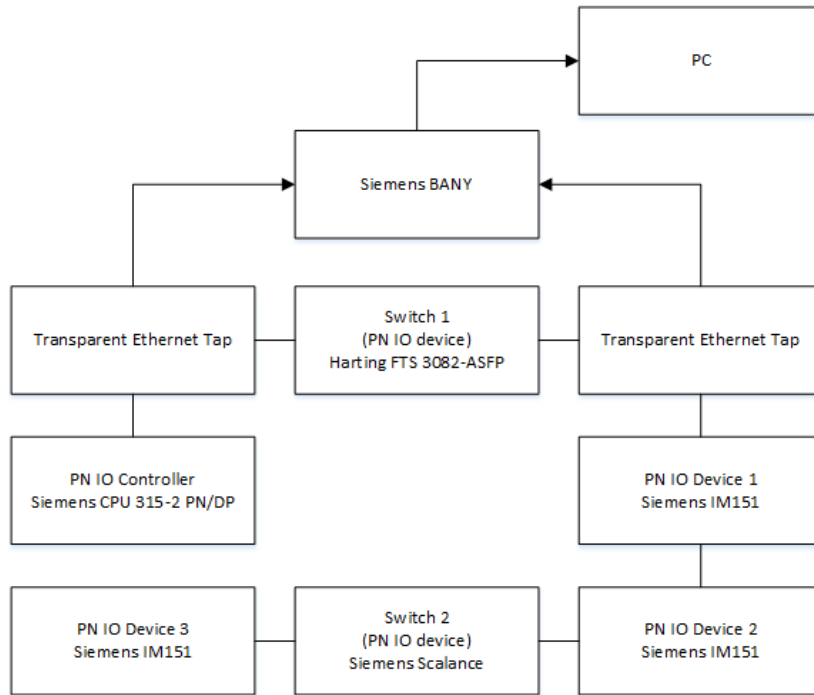


Figure 12: Measurement scheme for the Switch 1 device.

configuration. The *PN IO Device 3* is identical to *PN IO Device 1* and *PN IO Device 2* which were both measured. Both of those measurements yielded almost identical parameters therefore the same values have been used for the last, non measured device *PN IO Device 3* without introducing any error. The same values have been used for the *PN IO Controller* as well since the devices share a common manufacturer. A minor offset error may have been introduced by this extrapolation, yet it would affect all of the PNIO frames (which form $\sim 99.8\%$ of all of the steady state communication) in both directions since all the PNIO traffic either originates or ends in the controller. Therefore when comparing the frame propagation delays throughout the network for various devices or priority levels, the error is the same in all cases and can be dealt with easily.

The final parameter values of the normal distribution $\mathcal{N}_{SD}(\mu, \sigma^2)$, both measured and estimated, assigned to the network devices are as following:

- PN IO Controller: $\mu = 5.796\mu s, \sigma = 0.046\mu s$
- Switch 1 (Harting): $\mu = 4.689\mu s, \sigma = 0.096\mu s$
- PN IO Device 1: $\mu = 5.796\mu s, \sigma = 0.046\mu s$
- PN IO Device 2: $\mu = 5.796\mu s, \sigma = 0.042\mu s$
- Switch 2 (Scalance): $\mu = 8.508\mu s, \sigma = 0.294\mu s$
- PN IO Device 3: $\mu = 5.796\mu s, \sigma = 0.046\mu s$

3.2 SETTING THE PARAMETERS

Also, an addition independent traffic flow dependency on the switching delay (considering only the mean value) for devices with more than 2 ports was examined. All devices but the switches have only 2 ports and physically cannot accept any additional traffic except for what runs in the established communication. Therefore only the two switches were included in this measurement. An experiment was designed using two identical Ethernet taps and Siemens BANY as depicted on figure 13. The BANY is able to generate an external traffic with a variable load and customizable frames. The BANY has a total of 4 ports that are able to transmit or record frames. In this case two ports are used as frame generators and two are used to record the frames. In order to be able to record frames propagated through the measured switch, only one direction of communication can be recorded at one time. Each of the BANY ports assigned for frame generation in fact simulate one virtual device. These virtual devices send frames only to each other, the generated traffic is therefore independent and not forwarded to the ports used by the Profinet application. Using the variable additional traffic load (up to 100% of the 100Mbit/s channel), the measurement showed that the switching delays of the watched Profinet communication are influenced only in a minor way by the additional independent traffic.

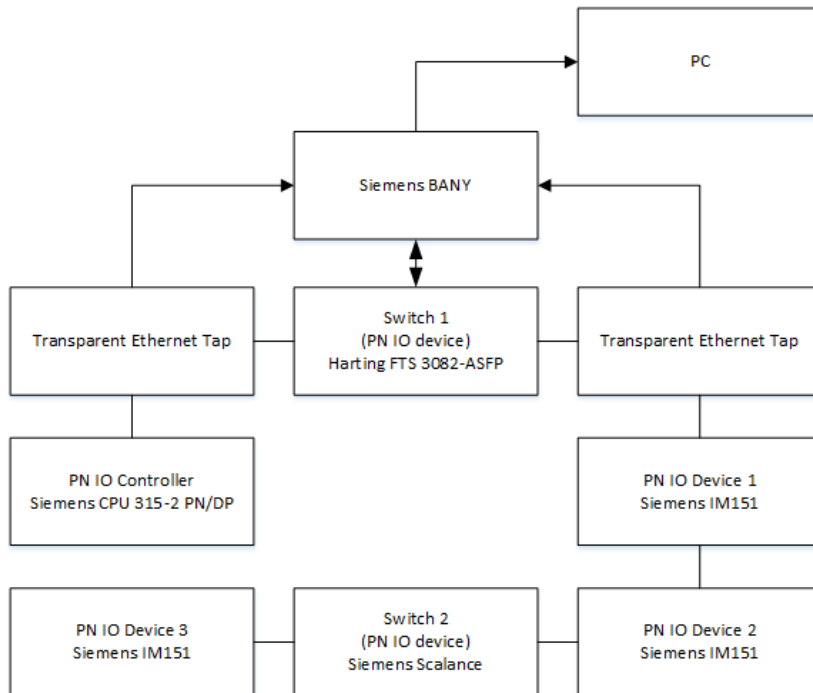


Figure 13: Measurement scheme for the Switch 1 device with additional traffic.

The graphs for both measurements are shown on figure 14 for the *Switch 1* (Harting) and on figure 15 for the *Switch 2* (Scalance). Since the influence is very light, only a few (4) values of the traffic load were tested [0%, 20%, 80%, 100%].

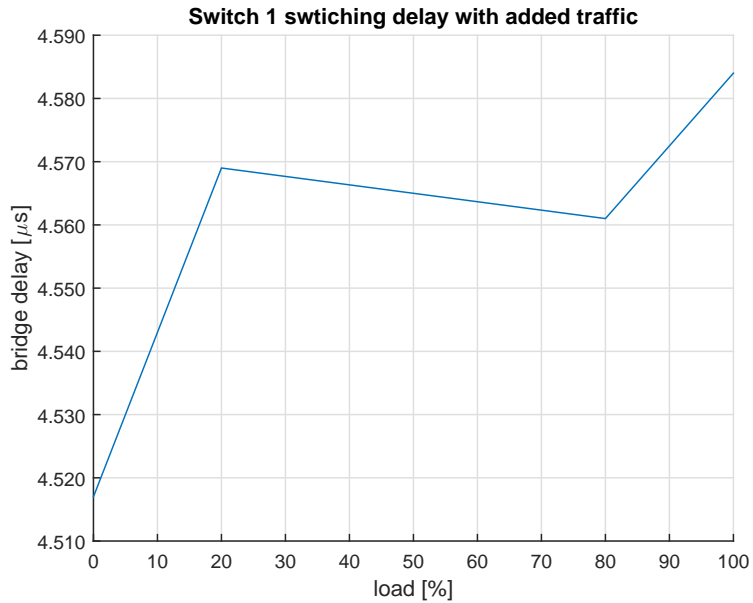


Figure 14: Switching delay dependency on additional traffic of Switch 1.

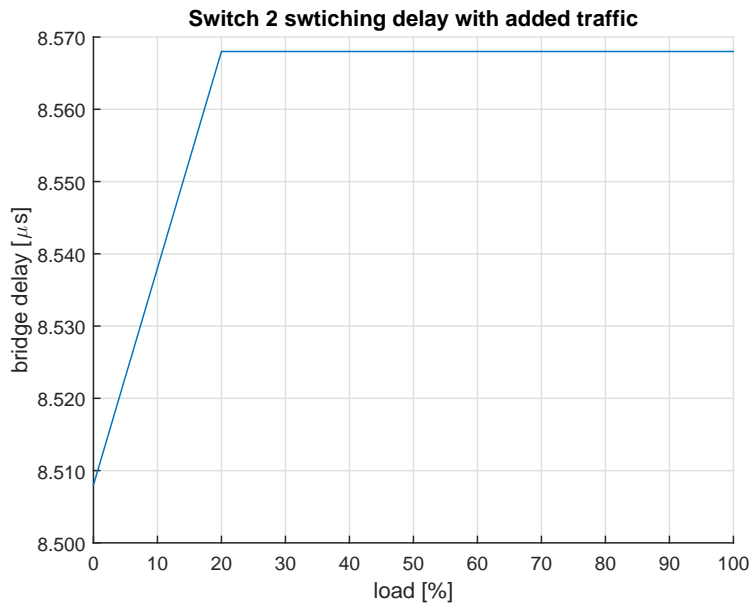


Figure 15: Switching delay dependency on additional traffic of Switch 2.

3.2.2 Communication parameters

The communication is modelled using the events system introduced in section 2.1.4.3. Each communication channel (meaning pair source - destination for given protocol) is modelled using one event with the *cyclic* trigger (modelled using normal distribution) and one *sendEthFrame* action. The *cyclic* time trigger has been chosen since we ap-

3.2 SETTING THE PARAMETERS

proach each non PNIO communication protocol and channel independently with no prior knowledge. The PNIO is given as *cyclic* by the network hardware configuration.

In order to accurately set all of the communication channels, a measurement was conducted to capture all of the traffic on each link of the network. The traffic was captured using a single Ethernet tap and recorded by a Siemens BANY that also stored it as a *.pcap file. The traffic was analysed using a standalone program *packetAnalyser* implemented in C++ using the libpcap library[9] This support program is not included in the Profinet simulation model and is briefly discussed in the Appendix A. In addition a network analysis tool Wireshark[10] was used to view, filter and count recorded frames. The measurement showed that the following link layer protocols are present in the communication - LLDP, ARP, STP and PN DCP. These form ~0.2% of all the traffic (rest is PNIO communication) on the link *Controller - Switch 1*.

The link between *Controller* and *Switch 1* has been chosen to demonstrate the simulation model since it covers the heaviest PNIO traffic (the results are discussed in section 3.4). The priorly known parameters for PNIO communication introduced in section 3.1 were also measured to verify their values and to be used in some of the conducted experiments described in section 3.4.1. The following communication protocols and their parameters for cyclically triggered events were measured on the chosen link (t_0 stands for the *initPhase* JSON parameter):

PNIO protocol

- Controller \rightarrow Switch 1: $\mu = 127.999ms$, $\sigma = 0.041\mu s$, $t_0 = 118.833ms$
- Controller \rightarrow IO device 1: $\mu = 2.000ms$, $\sigma = 0.046\mu s$, $t_0 = 1.827ms$
- Controller \rightarrow IO device 2: $\mu = 2.000ms$, $\sigma = 0.043\mu s$, $t_0 = 0.827ms$
- Controller \rightarrow Switch 2: $\mu = 127.999ms$, $\sigma = 0.089\mu s$, $t_0 = 120.833ms$
- Controller \rightarrow IO device 3: $\mu = 2.000ms$, $\sigma = 0.046\mu s$, $t_0 = 1.834ms$
- Controller \leftarrow Switch 1: $\mu = 128.740ms$, $\sigma = 17.213\mu s$, $t_0 = 6.234ms$
- Controller \leftarrow IO device 1: $\mu = 2.000ms$, $\sigma = 0.147\mu s$, $t_0 = 0ms$
- Controller \leftarrow IO device 2: $\mu = 2.000ms$, $\sigma = 0.117\mu s$, $t_0 = 1.392ms$
- Controller \leftarrow Switch 2: $\mu = 127.991ms$, $\sigma = 5.302\mu s$, $t_0 = 28.009ms$
- Controller \leftarrow IO device 3: $\mu = 2.000ms$, $\sigma = 0.366\mu s$, $t_0 = 0.895ms$

PN DCP protocol

- Controller \leftarrow Switch 2: $\mu = 59996.052ms$, $\sigma = 0.368ms$, $t_0 = 43876.632ms$

LLDP protocol

- Controller \rightarrow Switch 1: $\mu = 414.269ms$, $\sigma = 10.958ms$, $t_0 = 75.103ms$
- Controller \leftarrow Switch 1: $\mu = 2524.466ms$, $\sigma = 26.949ms$, $t_0 = 818.366ms$
- Controller \leftarrow IO device 1: $\mu = 1189.247ms$, $\sigma = 18.586ms$, $t_0 = 75.030ms$

STP protocol

- Controller \leftarrow Switch 1: $\mu = 999.439ms$, $\sigma = 17.030ms$, $t_0 = 1199.709ms$

ARP protocol

- Controller \leftarrow Switch 2: $\mu = 59996.175ms$, $\sigma = 0.432ms$, $t_0 = 13878.932ms$

3.3 Running the simulation

When all the parameters needed for the JSON structure are set, the simulation can be run. There are several possible ways how to run the simulation:

- Use the designed API library in a user application. This is discussed in section 2.3.2.
- Use a provided console application that uses the API. See section 2.3.3.
- Manually run the simulation in the OMNeT++ IDE. This option is described in section 3.3.1.

Note that the simulation requires a valid JSON file to be successfully run.

3.3.1 Using the OMNeT++ IDE

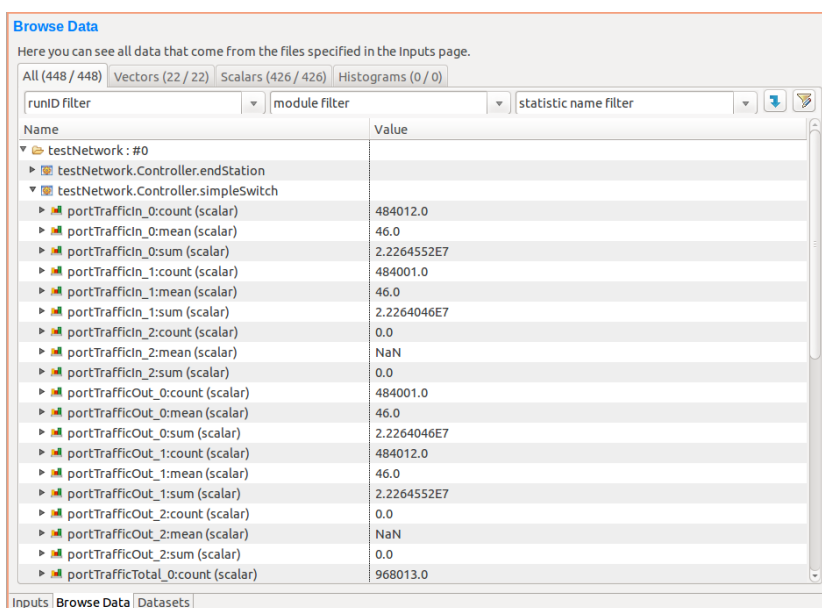
In order to run the simulation manually, user has to be provided with a NED file describing the given network and an INI file configuring the simulation. If this is not the case, these files has to be created. They are automatically created by the API function *loadInputFile* and placed in the project root (which is discussed in section 2.3). User may also choose to create those files manually corresponding to the provided JSON file, in this case please refer to OMNeT++ documentation[7].

When the two files are present in the root folder of the simulation model, to run the simulation click on the desired INI file in the *Project Explorer* window and select *Project* \rightarrow *Run*. The progress is printed in the *Console* window located at the bottom of the IDE.

3.4 SIMULATION RESULTS

3.4 Simulation results

After a successful simulation, the result files containing all the defined statistical signals are created and stored in the *results* folder located in the root folder of the simulation model. Two files are created - $\langle outputName \rangle.sca$ containing the scalar statistical signals and $\langle outputName \rangle.vec$ containing the vector statistical signals. These can be analysed directly in the OMNeT++ IDE by double clicking the given file in the *Project Explorer* window and then creating an *Analysis File*. When prompted, simply press finish to create the file. To view the results open the analysis file and switch to *Browse data* tab in the bottom left corner of the IDE's main window as can be seen in figure 16. Note that the created *Analysis File* is created in the root folder not in the *results* folder.



The screenshot shows the 'Browse Data' window in the OMNeT++ IDE. It displays a table of simulation results for a specific run. The table has two columns: 'Name' and 'Value'. The data is organized into a tree structure under the path 'testNetwork : #0'. The table lists various traffic statistics for three ports (0, 1, and 2), including counts, means, and sums for both incoming and outgoing traffic. The total incoming traffic count is 968013.0.

Name	Value
testNetwork : #0	
testNetwork.Controller.endStation	
testNetwork.Controller.simpleSwitch	
portTrafficIn_0:count (scalar)	484012.0
portTrafficIn_0:mean (scalar)	46.0
portTrafficIn_0:sum (scalar)	2.2264552E7
portTrafficIn_1:count (scalar)	484001.0
portTrafficIn_1:mean (scalar)	46.0
portTrafficIn_1:sum (scalar)	2.2264046E7
portTrafficIn_2:count (scalar)	0.0
portTrafficIn_2:mean (scalar)	NaN
portTrafficIn_2:sum (scalar)	0.0
portTrafficOut_0:count (scalar)	484001.0
portTrafficOut_0:mean (scalar)	46.0
portTrafficOut_0:sum (scalar)	2.2264046E7
portTrafficOut_1:count (scalar)	484012.0
portTrafficOut_1:mean (scalar)	46.0
portTrafficOut_1:sum (scalar)	2.2264552E7
portTrafficOut_2:count (scalar)	0.0
portTrafficOut_2:mean (scalar)	NaN
portTrafficOut_2:sum (scalar)	0.0
portTrafficTotal_0:count (scalar)	968013.0

Figure 16: Browsing results in the OMNeT++ IDE.

The result files are in fact text files that can be viewed in or parsed by various programs. Refer to [7] for their syntax definitions. To analyse the resulting vector file and to compare these results with real data captured in a *.pcap file, a MATLAB script *vectorAnalysis.m* has been created. The real data reference was created based on the captured *.pcap file in a standalone C++ program *packetAnalyser* which uses the libpcap library[9]. Both support tools are not included in the Profinet simulation model and are briefly discussed in the Appendix A.

3.4.1 Experiments

Several experiments were conducted on the chosen link *Controller - Switch 1* to verify and use the simulation model. The real data sample captured on this link is 319.349s long which determines the simulations' duration. This duration translates to over 9.6 ·

10^5 frames. Both of the data samples (real and simulated) have been re-sampled to unify them for comparison. The sampling period is set to $T_S = 1s$ therefore the comparison graphs ends at the time $t_F = 319s$. Note that when reading the graphs shown, negative simulation error means the simulation predicted less frames in given time than there actually were in the real data.

3.4.1.1 Experiment 1

In this experiment, we modelled only the PNIO communication based on the prior knowledge of the communication cycles as they were introduced in section 3.1. Based on the frame difference between the simulation and the measured data we altered the model to best fit the real data and again checked the frame difference.

Figures 17 and 18 depict a simulation error for each direction of the communication on the chosen link using the initial model. Each direction of communication has been altered as following:

- *Controller* \rightarrow *Switch 1*: The frame difference at the final time $t_F = 319s$ is $n_{diff} = -771$ frames (see fig. 17). To compensate this error, one event with the *cyclic* trigger and one *sendEthFrame* action destined to the *Switch 1* was added to the *Controller* JSON device description. The trigger's parameters are: $\mu = 414.285ms$, $\sigma = 0ms$ and *initPhase* = $0ms$. The σ parameter would only add more noise to the data (decreasing readability) and was not considered. Although the normal distribution is not defined for $\sigma = 0$, in this case the constant value μ is used instead. The μ parameter was obtained as follows:

$$\mu = \frac{t_F}{|n_{diff}| - 1} [s] \quad (1)$$

Note that the denominator is not directly number of the frame difference n_{diff} but rather the number of frame periods between them.

- *Controller* \leftarrow *Switch 1*: The frame difference at the final time $t_F = 319s$ is $n_{diff} = -713$ frames (see fig. 18). To compensate this error, one event with the *cyclic* trigger and one *sendEthFrame* action destined to the *Controller* was added to the *Switch 1* JSON device description. The trigger's parameters are: $\mu = 448.033ms$, $\sigma = 0ms$ and *initPhase* = $0ms$. The parameter μ was obtained using eq. 1.

3.4 SIMULATION RESULTS

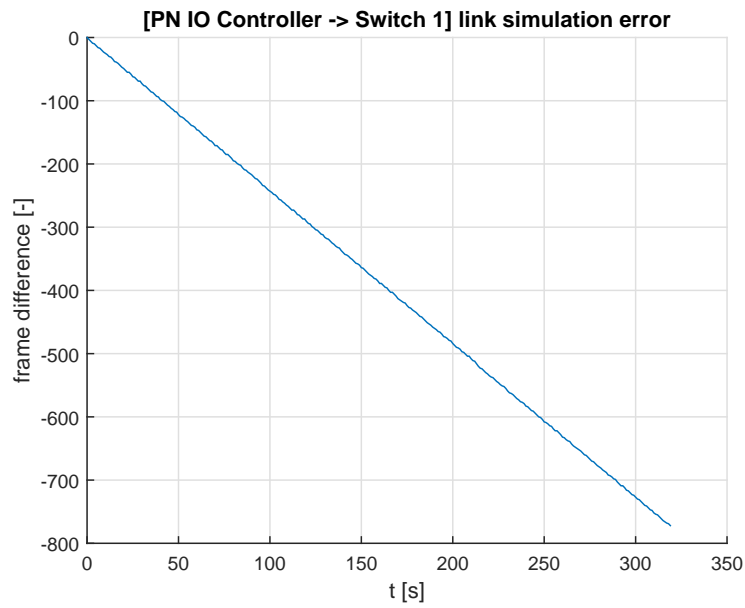


Figure 17: Experiment 1 simulation error on link *Controller* \rightarrow *Switch 1*.

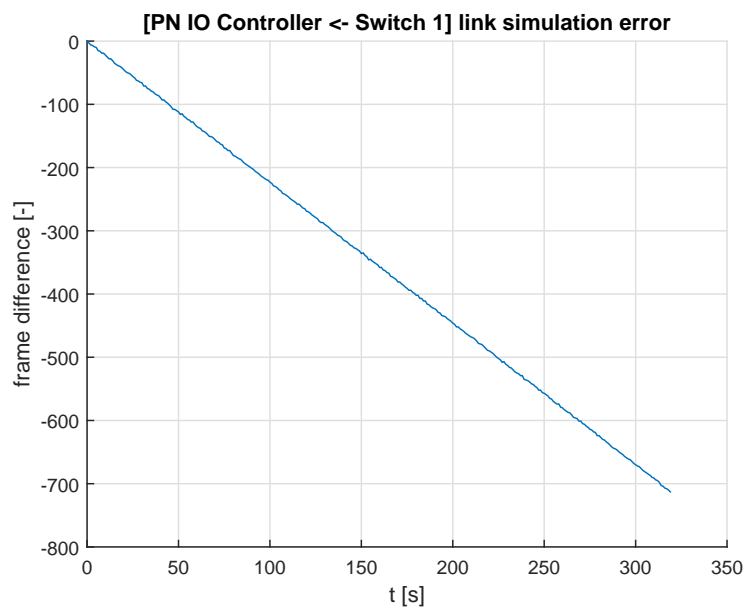


Figure 18: Experiment 1 simulation error on link *Controller* \leftarrow *Switch 1*.

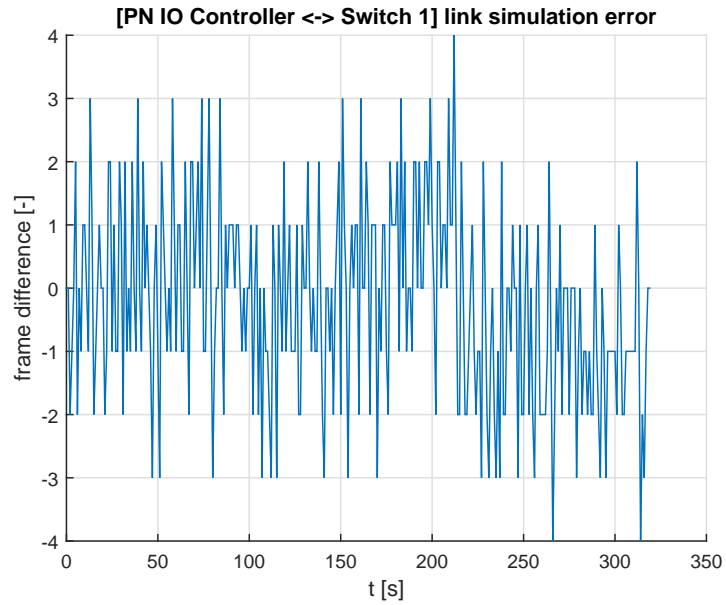


Figure 19: Experiment 1 simulation error on link *Controller* \leftrightarrow *Switch 1* after the compensation.

The total simulation error after the compensation is shown on figure 19. If we consider that the total number of frames is $\sim 9.6 \cdot 10^5$, the estimation error is in the order of 10^{-5} . Both of the JSON files (describing original and altered model) are attached on CD in folder *experiments/ex1*.

3.4 SIMULATION RESULTS

3.4.1.2 Experiment 2

In this experiment, we modelled the PNIO communication based on the prior knowledge of the communication cycles as they were introduced in section 3.1 and all of the remaining protocols' traffic based on the measurement. Based on the frame difference between the simulation and the measured data we altered the model to best fit the real data and again checked the frame difference.

Figures 20 and 21 depict a simulation error for each direction of the communication on the chosen link using the initial model. Each direction of communication has been altered as following:

- *Controller* \rightarrow *Switch 1*: The frame difference at the final time $t_F = 319s$ is $n_{diff} = -2$ frames (see fig. 20). To compensate this error, one event with the *cyclic* trigger and one *sendEthFrame* action destined to the *Switch 1* was added to the *Controller* JSON device description. The trigger's parameters are: $\mu = 319000ms$, $\sigma = 0ms$ and $initPhase = 0ms$. The parameter μ was obtained using eq. 1.

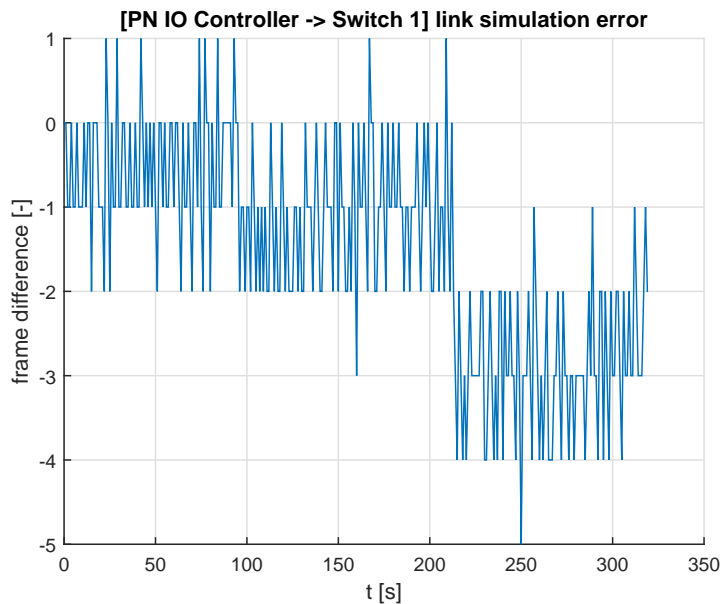


Figure 20: Experiment 2 simulation error on link *Controller* \rightarrow *Switch 1*.

- *Controller* \leftarrow *Switch 1*: The frame difference at the final time $t_F = 319s$ is $n_{diff} = 11$ frames (see fig. 21). This means that the simulation is ahead of the real data. This behaviour is clear if we look at the measured values of the PNIO protocol originating from both of the switches - they don't match the expected values entered in the prior set-up of the network. Refer to section 3.1 for the expected values and section 3.2.2 for the measured values. To account this deviation, the values were set according to the measurement.

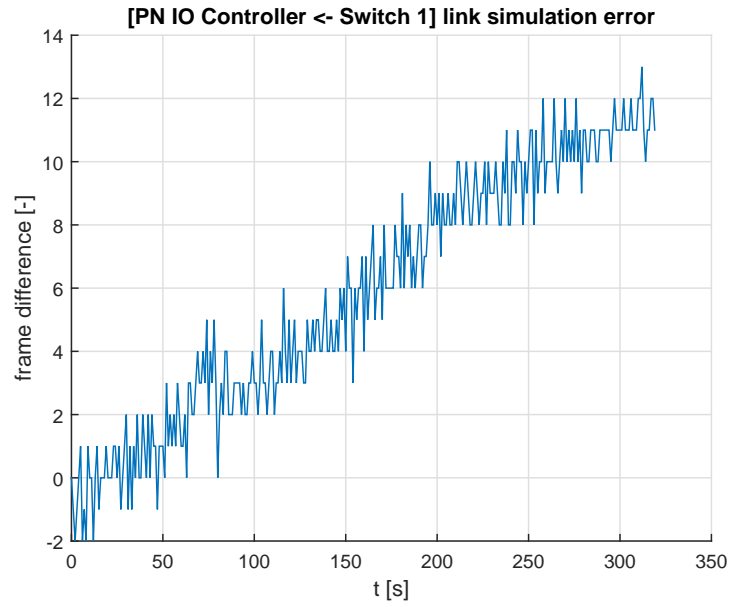


Figure 21: Experiment 2 simulation error on link *Controller* \leftarrow *Switch 1*.

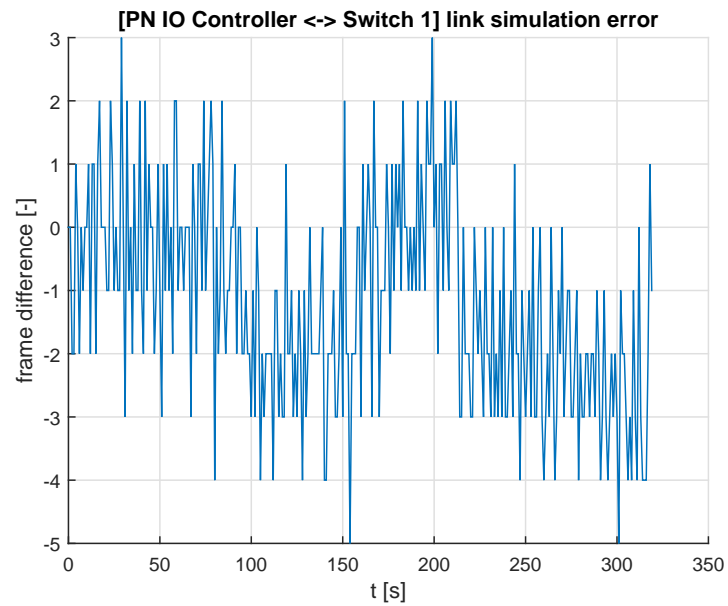


Figure 22: Experiment 2 simulation error on link *Controller* \leftrightarrow *Switch 1* after the compensation.

The total simulation error after the compensation is shown on figure 22. If we consider that the total number of frames is $\sim 9.6 \cdot 10^5$, the estimation error is in the order of 10^{-5} . Both of the JSON files (describing original and altered model) are attached on CD in folder *experiments/ex2*.

3.4 SIMULATION RESULTS

3.4.1.3 Experiment 3

In this experiment, we modelled all of the traffic based on the measured values (see section 3.2.2). Based on the frame difference between the simulation and the measured data we altered the model to best fit the real data and again checked the frame difference.

Figures 23 and 24 depict a simulation error for each direction of the communication on the chosen link using the initial model. Each direction of communication has been altered as following:

- *Controller* \rightarrow *Switch 1*: The frame difference at the final time $t_F = 319s$ is $n_{diff} = -3$ frames (see fig. 23). To compensate this error, one event with the *cyclic* trigger and one *sendEthFrame* action destined to the *Switch 1* was added to the *Controller* JSON device description. The trigger's parameters are: $\mu = 159500ms$, $\sigma = 0ms$ and $initPhase = 0ms$. The parameter μ was obtained using eq. 1.

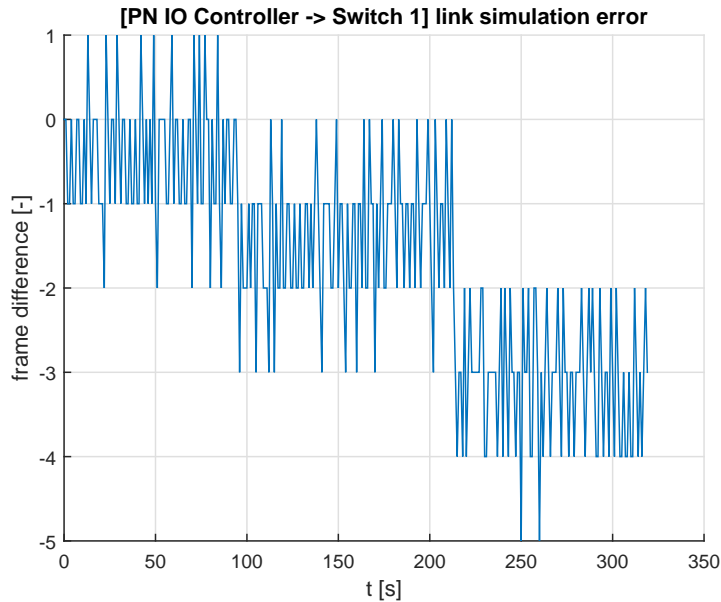


Figure 23: Experiment 3 simulation error on link *Controller* \rightarrow *Switch 1*.

- *Controller* \leftarrow *Switch 1*: The frame difference at the final time $t_F = 319s$ is $n_{diff} = -2$ frames (see fig. 24). To compensate this error, one event with the *cyclic* trigger and one *sendEthFrame* action destined to the *Controller* was added to the *Switch 1* JSON device description. The trigger's parameters are: $\mu = 319000ms$, $\sigma = 0ms$ and $initPhase = 0ms$. The parameter μ was obtained using eq. 1.

The total simulation error after the compensation is shown on figure 25. If we consider that the total number of frames is $\sim 9.6 \cdot 10^5$, the estimation error is in the order of 10^{-5} . Both of the JSON files (describing original and altered model) are attached on CD in folder *experiments/ex3*.

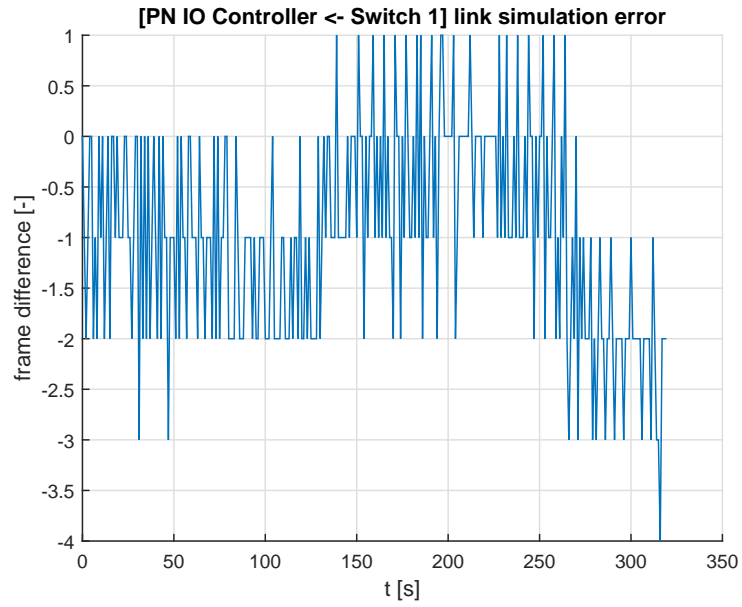


Figure 24: Experiment 3 simulation error on link *Controller* \leftarrow *Switch 1*.

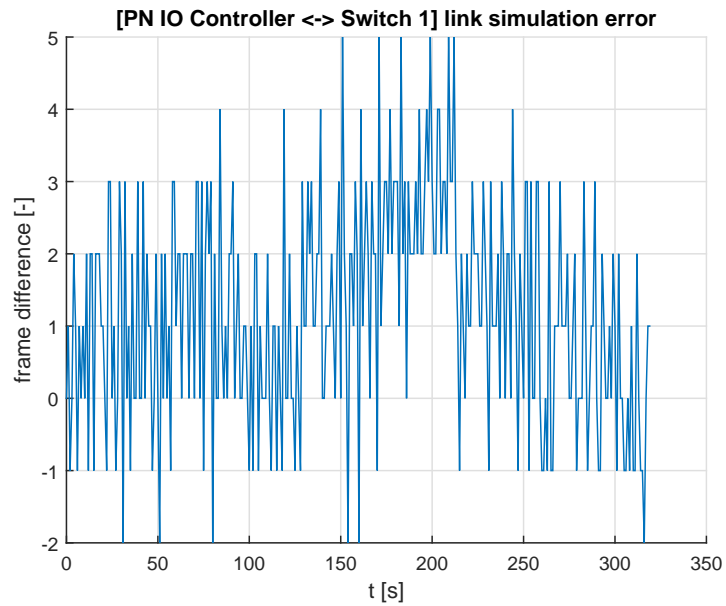


Figure 25: Experiment 3 simulation error on link *Controller* \leftrightarrow *Switch 1* after the compensation.

3.4.1.4 Experiment 4

In this experiment we use the simulation to detect a network failure. The final model created in experiment 1 (see section 3.4.1.1) was used as a reference to the real data measurement. The real data were measured on the chosen link *Controller 1 - Switch 1*.

3.4 SIMULATION RESULTS

During the real data capture, the *PN IO Device 3* has been disconnected from the network at time $t_D = 256s$. Figure 26 shows the frame difference between the simulation and the captured real data. Note that the real data sample is only 300s long.

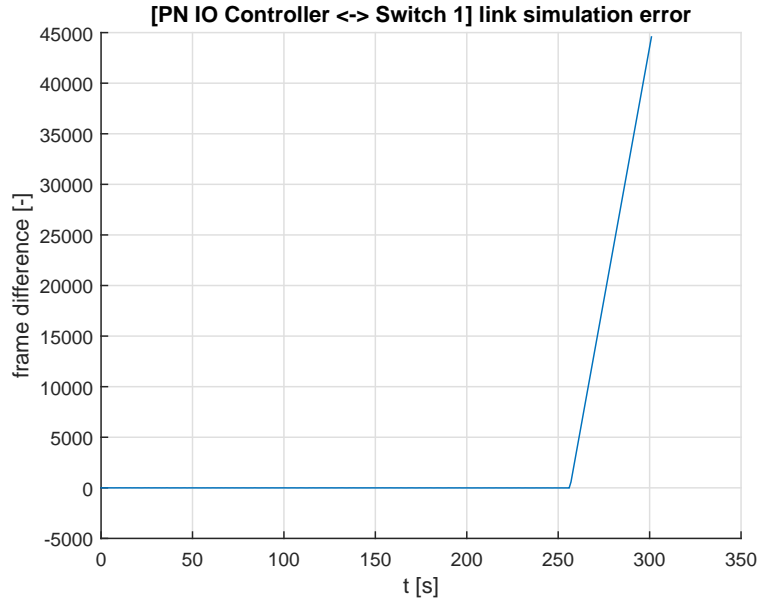


Figure 26: Experiment 4 simulation error on link *Controller* \leftrightarrow *Switch 1* after a network failure.

3.4.1.5 Experiment 5

In this experiment we use the simulation to detect an error in the network hardware configuration. The final model created in experiment 1 (see section 3.4.1.1) was used as a reference to real data measurement. The real data were measured on the chosen link *Controller 1 - Switch 1*. In this case, an intentional error was introduced in the PNIO communication between the *Controller* and the *PN IO Device 3* - the communication period was set to 4ms instead of 2ms. Figure 27 shows the frame difference between the simulation and the captured real data. Note that the real data sample is only 300s long.

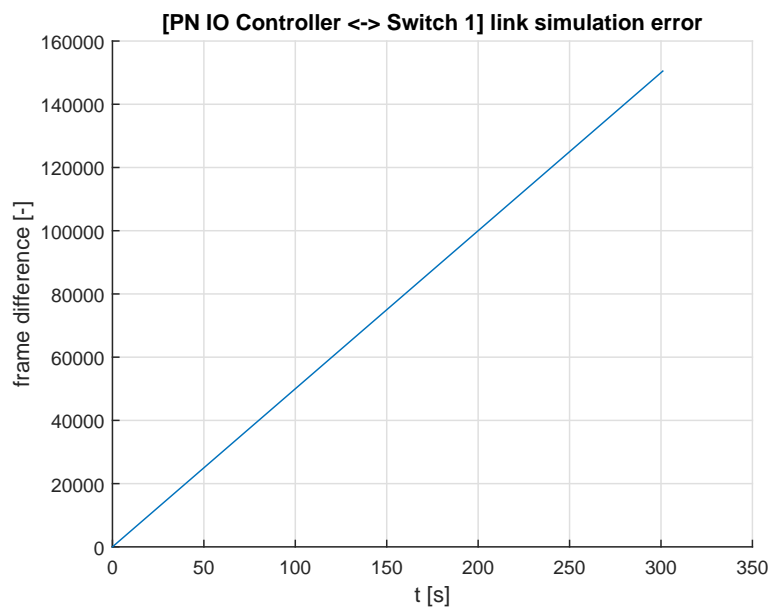


Figure 27: Experiment 5 simulation error on link *Controller* \leftrightarrow *Switch 1* with an error in HW configuration.

3.4 SIMULATION RESULTS

Part 4

Conclusion

A Profinet simulation model has been implemented using the OMNeT++ simulation framework. The model is able to capture generic devices, switches or endpoints and their traffic such as cyclic or acyclic communication on a link layer in a parametrized way using the designed JSON structure. The model also supports deriving of user-defined modules to model more complex devices. Also, the introduced *ProfinetSimApi* API has been implemented to allow an automated simulation set-up and execution.

A physical test network has been modelled using the created simulation model to verify it and to demonstrate its possible uses in network diagnosis. The simulation error (frame difference) achieved on over 5 minutes long data samples was in the order of 10^{-5} .

Future improvements may include adding additional core modules implementing unique and more advance actions, adding additional trigger types or even enabling user-defined triggers. Another area of improvement might be to add a compatibility support for the Ethernet modules and frames of the INET library present in the OMNeT++ framework. Also, additional platform support might be considered for the API.

4.0 CONCLUSION

Appendices

A Support analysis tools

Apart from the simulation model itself, two supporting tools have been implemented to automate the simulation result analysis and the parameter acquisition.

A.1 Matlab script `vectorAnalysis.m`

This Matlab script is used for parsing the resulting *.vec files containing vector statistical signals. It can also parse text files generated by the `createRealDataVector` function of the `packetAnalyser` tool described below. If both data files are loaded, the script can re-sample them and plot a frame difference graph. More information is provided via commentary in the script itself.

A.2 Tool `packetAnalyser`

A developer tool `packetAnalyser` that performs various task using a libpcap library[9] was implemented in C++ under the Ubuntu Linux distribution. Only source files are provided since the functions' parameters have to be modified manually as well as the desired function calls or other modifications. Following functions have been implemented:

- `int createRealDataVector()` — This method is used to generate real data vectors based on a specified *.pcap file and device. The *.pcap file should only contain measurement from one link (one way or full-duplex). Three text files are created - `packetsVecOut.txt` containing frames outgoing from specified device, `packetsVecIn.txt` containing frames incoming to specified device and `packetsVecTotal.txt` for all frames present in the *.pcap file. The output format of all of the generated files is as following: `<timestamp><number-of-frames>` separated by a single space " " (each entry is on a new line). The method always returns 0.
- `int delayMeasurement()` — This method is used to analyse the switching delay measurements described in section 3.2.1 based on a *.pcap file. The results (μ and σ values) are printed in the standard output. The method always returns 0.

A SUPPORT ANALYSIS TOOLS

- *int* **trafficParameters()** — This method is used to analyse a communication channel measurements between two devices described in section 3.2.2 based on a *.pcap file. The results (μ and σ values) are printed in the standard output. The method always returns 0.

References

- [1] PROFIBUS & PROFINET International. Profinet. <http://www.profibus.com/technology/profinet/>, 2015.
- [2] Itu.int. X.200:information technology - open systems interconnection - basic reference model: The basic model. <http://www.itu.int/rec/T-REC-X.200/en/>, 2015.
- [3] Rich Seifert. *The switch book: the complete guide to LAN switching technology*. John Wiley & Sons, Inc., 2000.
- [4] Omnetpp.org. Omnet++ discrete event simulator. <http://http://www.omnetpp.org/>, 2015.
- [5] Json.org. Json. <http://http://www.json.org/>, 2015.
- [6] W3.org. Extensible markup language (xml). <http://http://www.w3.org/XML/>, 2015.
- [7] Omnetpp.org. *OMNeT++ User Manual*, 2015. Version 4.6.
- [8] Supereasyjson. <http://sourceforge.net/projects/supereasyjson/>, 2014.
- [9] Libpcap-1.7.3. <http://www.linuxfromscratch.org/blfs/view/svn/basicnet/libpcap.html>, 2015.
- [10] Wireshark.org. Wireshark. <https://www.wireshark.org/>, 2015.

Attached CD contents

The attached CD contains a pdf version of this thesis, the implemented tools and example JSON definitions. The folder structure is as follows:

- *dip_prasekjan.pdf* — pdf version of this thesis.
- */api/* — contains the *profinetSimApi* source files, the compiled library and the Console application
- */experiments/* — contains JSON definitions for the conducted experiments.
- */fig/* — contains all the figures used in their original resolution.
- */profinetSim/* — contains the designed Profinet simulation tool OMNeT++ project.
- */supportTools/* — contains the implemented support tools discussed in Appendix A.