Simulation and emulation of realtime communication networks

Franz Profelt

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Embedded Systems Design

in Hagenberg

im Juni 2016

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 20, 2016

Franz Profelt

Contents

Abstract

The fields of simulation and emulation are steadily gaining importance for the development of embedded systems. This is caused by the growing requirement for comprehensive testing. Various fields of application as the automotive industry demand a high level of testing.

This leads to the urge of implementing a simulation or emulation for given embedded systems or new systems under development. The field of simulation shows a big number of different systems and frameworks. Each of them provides various configuration possibilities. This variety of different possibilities and strategies for developing a system can keep developers from making a tailor made solution of a customized simulation. OMNeT++ embodies a highly flexible simulation framework for various applications.

This thesis analyzes the properties of OMNeT++ and shows the possibilities of configuration and modification for achieving specific goals. Different performance measurements are made for analyzing a test simulation in terms of runtime, processed events and real-time capabilities. An optimized design for achieving good performances is found.

The functionalities and possibilities for real-time simulation and emulation portray an interesting field for the developing and testing of embedded systems.

The Open Source implementation openPOWERLINK provides the possibility to analyze and modify an existing real-time communication system. openPOWERLINK implements the POWERLINK protocol which is widely used in the field of automation and real-time communication. Developing a simulation embedding a given system like the openPOWERLINK stack raises interesting questions regarding multiple instances. A solution is found and a POWERLINK network consisting of openPOWERLINK nodes can be simulated.

Kurzfassung

Die Themengebiete der Simulation und Emulation gewinnen stetig an Bedeutung für die Entwicklung von eingebetteten Systemen. Dies zeigt sich in der steigenden Nachfrage von umfangreichen Tests. Eine Vielzahl von Anwendungsgebieten erfordern einen hohen Grad an Tests.

Dieser Trend führt zu einem steigendem Interesse an der Entwicklung einer Simulation oder Emulation für bestehende oder neu entwickelte Systeme. Es existiert eine große Anzahl von verschiedenen Systemen zur Entwicklung von Simulationen die oftmals vielseitige Konfigurationsmöglichkeiten bieten. Dieser Überfluss an verschiedenen Möglichkeiten eine Simulation zu entwickeln, kann Entwickler von der Entscheidung abhalten, eine Simulation für ihren Nutzen zu entwickeln. OMNeT++ stellt ein sehr flexibles System zur Entwicklung von Simulationen für verschiedenste Anwendungen dar.

Diese Arbeit analysiert den Aufbau und die Eigenschaften von OMNeT++ und zeigt die Möglichkeiten zur Konfiguration und Anpassung. Es werden unterschiedliche Leistungsmessungen durchgeführt, die eine Testsimulation im Bezug auf Laufzeit, verarbeiteten Ereignissen und Echtzeiteigenschaften überprüft. Es wurde ein optimierter Aufbau der Simulation gefunden, welcher zu einem verbesserten Ergebnis der Leistung führt. Die vorhandenen Funktionen und die Möglichkeiten zur Entwicklung einer Echtzeitsimulation und Emulation stellt eine interessante Methode zum Testen und Entwickeln von eingebetteten Systemen dar. Die Open Source Implementierung openPOWERLINK bietet die Möglichkeit, ein bestehendes Echtzeitkommunikationssystem zu analysieren und für eine Simulation anzupassen. openPOWERLINK implementiert das weit verbreitete POWERLINK Protokoll, welches häufig im Bereich der Industrieautomation und Echtzeitkommunikation eingesetzt wird. Bei der Entwicklung einer Simulation, die ein gegebenes System, wie die openPOWERLINK Implementierung, einbindet, werden Fragen bezüglich Mehrfachinstanzen aufgeworfen. Es wurde eine Lösung für diese Problematik gefunden und anschließend eine Simulation entwickelt, die ein aus openPOWERLINK Knoten bestehendes POWERLINK Netzwerk nachstellt.

Chapter 1

Introduction

This thesis is intended to analyze the simulation framework *OMNeT++* [1] and the possibilities for real-time simulation and emulation of a real-time communication network. A test simulation is analyzed for determining an optimized design regarding performance and real-time capabilities. Furthermore, a simulation of the real-time communication protocol POWERLINK and its open source implementation openPOWERLINK [25] is implemented. For developing this simulation, the openPOWERLINK stack is analyzed and the platform dependencies are investigated.

1.1 Motivation

The fields of embedded systems and real-time communication entail critical requirements for timings and deterministic behavior. The development of systems meeting those requirements can be very difficult due to the lack of possibilities for sufficient testing. A variable testing environment facilitates the development and verification of such systems. The correct replication of complex scenarios and special operating conditions is not always possible. Therefore the need for simulation is gaining importance with increasing complexity of embedded systems.

Extended functionalities of simulation frameworks allow the usage as real-time simulation. By simulating a system in real time, i.e. the simulated time passes according to the real (world) time, these simulations can be used in the fields of emulation and hardware in the loop (*HiL*). This allows variable testing scenarios built with a simulated environment. These fields are necessary for testing embedded systems due to the increased possibilities of testing scenarios and setups.

Because of the raising importance of simulation and emulation. the analysis of simulation frameworks and the development of simulations for real-time communication systems represent an important field.

1.2 Content

Chapters 2 to 6 describe fundamentals regarding OMNeT++, general simulations, emulations and *HiL*, simulation designs and parallel simulation. These chapters intend to provide an extensive introduction and preparation for further analyses.

Chapter 7 discusses the analysis of two different fundamental designs and their impact on the achieved performance. This performance is measured in multiple ways to analyze the simulation regarding runtime, processed events and real-time behavior.

Subsequently, the Open Source stack openPOWERLINK is analyzed placing a special focus on the structure and the platform dependencies. As outlined in chapter 8, these dependencies are used for the following development of an OMNeT++ simulation embedding the openPOWERLINK stack. This simulation and the implementation strategies are discussed in chapter 9.

Finally the experiences and knowledge gained during this research are concluded and listed in chapter 10.

The appendix contains three chapters and comprises code snippets from the OMNeT++ framework (A), additional design measurements (B) and some implementation samples of the openPOWERLINK simulation and the integration of the openPOWERLINK stack (C).

1.3 Purpose

The purpose of this thesis is an analysis of the OMNeT++ framework and its capabilities regarding real-time simulation. Furthermore it is intended to perform an analysis of the openPOWERLINK stack focusing on the platform dependencies. This leads to the development of a simulation embedding openPOWERLINK nodes within a simulation POWERLINK network.

This thesis demonstrates the capabilities and strategies of developing a simulation for an embedded real-time communication network.

Chapter 2

OMNeT++

OMNeT++ represents an open source simulation framework written in C++. It provides an object oriented modular discrete event network simulation framework. The platform independent simulation library includes an integrated development environment (IDE), which is based on the Eclipse platform. The commercial version OMNEST provides various licensing models, whereas OMNeT++ is only available for academic or non-profit use. The intention of OMNeT++ is to provide infrastructure for writing simulations for various fields, especially those of network simulations. For this thesis the latest version of OMNeT++ 4.6 was used and analyzed.

Simulations developed with OMNeT++ are based on different components, which provide functionality for communication and topologies.

2.1 Components

Within an OMNeT++ simulation different components are used to represent the simulated system. Each component is described by a *network description* (NED) file and can be enhanced with C++ code. The *NED* file contains information about the component which is necessary for connecting and designing the simulated system, such as gates, submodules and parameters.

The different types of components and their usage are explained in the next sections.

2.1.1 Network

The outermost component is a network which consists of other components like modules and channels. The simulated topology and the connections between modules are defined within the network. A network represents a closed system which can be simulated by itself, other components must be instantiated within a network to be simulated. All instantiated modules,

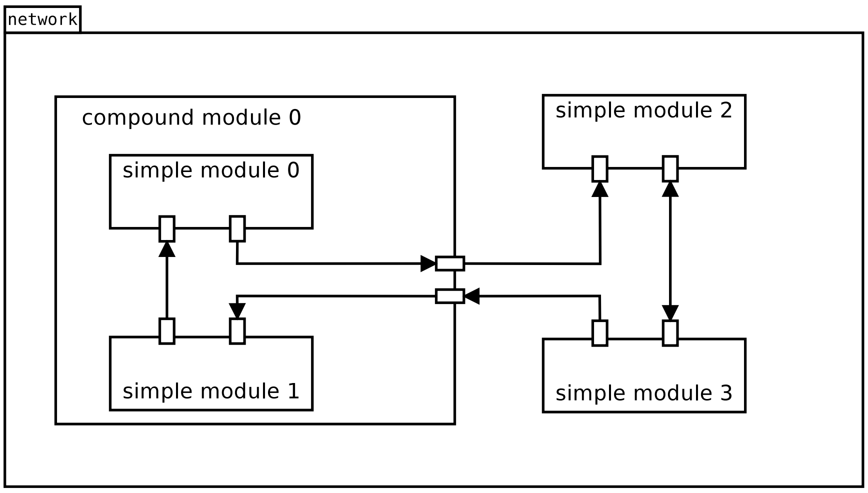

Figure 2.1: OMNeT++ components in an example network.

optional parameters and the connections between the modules are defined in the networks *NED* file. [1, section 3.2.1]

2.1.2 Modules

Modules represent functional groups of different complexities. There are two types of modules available in OMNeT++.

A simple module is the smallest part within a simulated hierarchy and represents a functional unit. For this functional unit the behavior for handling messages, the possible connections and additional parameters can be defined [1, section 3.3]. The possible connections of modules are represented by gates which can be connected to a channel or directly to other gates.

Multiple modules can be connected and combined to a compound module. Such compound modules are used in the same way as simple modules, but represent bigger functional groups. Compound modules are defined in the same way as networks, including the instantiated submodules and the connections between them [1, section 3.4]. For OMNeT++ a network and a compound module is the same component which differentiate by the value for the built-in *NED* property *@isNetwork*.

An example network including simple modules connected to a compound module is shown in Figure 2.1. Each module displayed in Figure 2.1 defines two gates which is either an input, output or bidirectional gate.

Specific functionality for custom components is implemented in the linked C++ code. The assignment of *NED* and C++ code is usually done with identical file names, but can also be done manually with the *@class NED* property. The components can embed any functionality implemented in C/C++. The usage of external libraries or language features is not limited, but must be used with care due to the effect on simulation performance. For implementing the behavior of the modules and accessing the simulation environment, for example reading the value of a defined *NED* parameter or sending a message via a gate, OMNeT++ provides different functionalities and strategies.

The functionality of a compound module is given by the submodules and their connections. A simple module is implemented by overriding specific methods following one of the two strategies below:

- Using the *handleMessage* method, a module can react to incoming messages. Drawing on this strategy, the module is only active when a message is received. The methods *send*, *scheduleAt* and *cancelEvent* can be used within *handleMessage* for sending messages, scheduling self-messages and canceling of scheduled self-messages. [1, section 4.4.1]
- The second strategy uses the method *activity* and is called *process style* strategy. The method *activity* is called by the simulation runtime as co-routine and can be implemented in the same way as a normal thread or process on an operating system. Returning from the *activity* methods equals the finishing of the modules simulation. Beside the available methods of the first strategy, additional methods can be used within *activity*. Using the method *receive* the module waits until a message is received. Waiting for a specific amount of simulation time is achieved with the *wait* method. A simple module using the *activity* method is called as co-routine of the simulation core and is scheduled non-preemptively. I.e. the activity method is not interrupted by the simulation and has to suspend by itself. This is done by waiting for a received message or waiting for a specific amount of time. [1, section 4.4.2]

2.1.3 Channels

The connections between modules can be realized in different ways. A direct connection of two gates transports the transmitted messages immediately. For applying transportation parameters (e.g. delay, latency, jitter) or implementing a custom behavior the connection can be established with a channel. The OMNeT++ framework provides three built-in channels for direct usage or sub classing.

- The *IdealChannel* represents the same behavior as a direct connection between gates and transmits each message immediately.
- The *DelayChannel* provides a *delay* parameter for configuring a constant delay for each message. Additionally, there is the possibility to disable the channel and drop all containing messages.
- The *DatarateChannel* provides the *datarate* parameter for a variable rate of transmission including a configurable bit error rate (BER) and

packet error rate (PER). These error rates can be used to mark a transmitted packet as erroneous.

These channels can be used directly or being subclassed for implementing custom functionality. [1, section 3.5]

For implementing a custom channel, three methods must be implemented:

isTransmissionChannel returns true if the channel is a transmission channel, i.e. the transmission duration is calculated and set within the packet. This type of channel only affects messages of type *cPacket* or derived types.

getTransmissionFinishTime returns the finish time of the transmission for messages of type *cPacket* or derived types.

processMessage models the behavior of the channel and its functionality. In this method the results can be stored in a provided structure, which allows a dropping of the message and a change of the resulting transmission time. The transmitted message is passed to the *processMessage* method, therefore various modifications can be made simulating the transmission through the channel. [1, section 4.8]

2.1.4 Parameters

Each component can define various parameters for configuration of their behavior. The value of these parameters can be assigned in different ways. The assignment from a *NED* file can be done directly, via inheritance, or via a compound module or network which contains the component. All parameters can also be set via a configuration file (e.g. omnetpp.ini) or interactively requested from the user. In the case of none assignment a parameter can also define a default value.

These parameters can be used by the implemented behavior by using the *par* method. [1, section 3.6]

2.1.5 Messages

Transmitted data is encapsulated in another component called message. Messages are a fundamental component of an OMNeT++ simulation as they do not only transport data but can also represent functional messages like jobs, events or tasks. The meaning of a message depends on the written simulation and the simulated system. [1, chapter 5]

These messages can also be customized for holding a specific set of data like a protocol header, checksum, or other specific data. The existing message class *cMessage* and its derived specialization *cPacket* provide different members and methods which can be used for simulations. These include control information, type information, time stamp, etc. and are included to ease the development of a simulation. Adding a few simple data fields to a message can be either done by subclassing *cMessage*, *cPacket* or using the

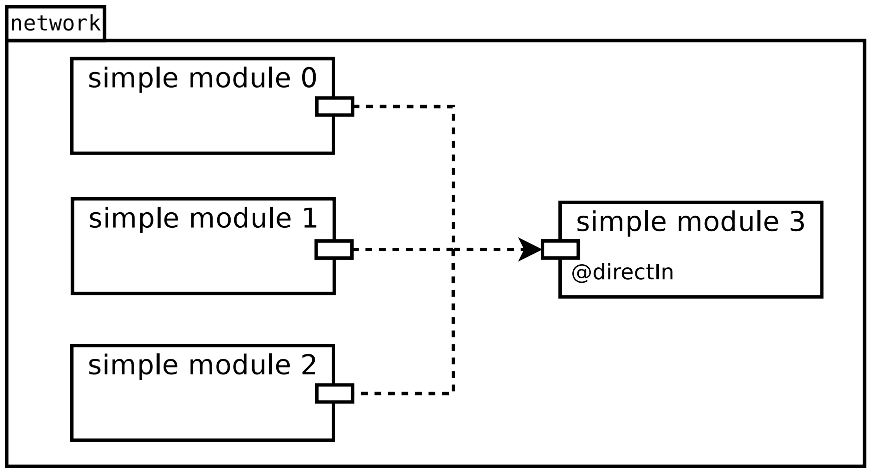

Figure 2.2: Multiple simple modules sending messages directly to simple module 3

NED language in special *.msg* files. By defining a custom message using *NED* a customized subclass will be generated by the simulation and can be used as a normal message. [1, chapter 6]

Any module can send a message via its gates by using the group of *send* methods or send it to itself as *self-message* with *scheduleAt*. The mechanism of messages is also used for implementing timeouts, timers, etc. by sending a specific message to the current module. Therefore, the method *scheduleAt* takes an absolute simulation time and a message as parameters and sends the given message at the given point of simulation time to the current module. These *self-messages* are handled by the same function as any other messages coming from other modules. For the identification of *self-messages*, the built-in method *isSelfMessage* is available. A scheduled *self-message* can be canceled via the method *cancelEvent* taking the scheduled message. [1, section 4.7.1]

The *send* method takes the message to send and the used gate as parameter. For defining the gate, the defined name, the id or the object itself can be used [1, section 4.7.2]. When the transmission of the messages should occur at a later time, there is a built-in functionality given with the *sendDelayed* method. This method work in the same way as the default *send* method but take an additional parameter which represents the delay. [1, section 4.7.6]

Messages can also be sent directly to gates of defined modules without the need of a connection to the current module. This is called *direct sending* and can be useful when multiple sender modules send to a single input gate. An example for this usage is shown in Figure 2.2.

Such a gate should be declared with the *@directIn NED* property to avoid notifications about a not connected gate. [1, section 4.7.5]

Additional handling of messages as broadcasts and retransmission requires attention to the owner of messages. By sending a message, its owner changes to both the simulation core and the receiver module. Therefore, explicit copying of a message via *dup* is necessary for sending a message to multiple modules or resending a message. [1, section 4.7.3]

Each message sent either by another module or by the current module itself represents an event for the simulation with an according time, at which this event should happen, or the message should be delivered/received. The execution and the handling of such events is done by the simulation core and defines the execution order and the performance of the simulation. The different types of simulations and the simulation core of OMNeT++ are discussed in chapter 3.

2.2 Simulation results

The simulation of different systems results in different types of outcomes. Therefore OMNeT++ provides multiple functionalities for recording and saving different results.

2.2.1 Simulation library

The traditional way of recording simulation results is the use of the simulation library functions.

The built-in type *cOutVector* provides the functionality used for recording a time series of data. By holding an instance of *cOutVector* a series of data can be recorded during simulation. During initialization of the module the according name of the recorded time series should be set. The recorded data of all *cOutVector* instances is written at the end of the simulation to a single output vector file (*.vec*). [1, section 7.9.1]

Recording single values (scalars) is done, by using the *recordScalar* method of a *cModule*. Usually this is called in the *finish* method and may record a result of statistical analysis or even a whole statistic object. All recorded scalars are written in a line based text file (*.sca*). [1, section 7.9.2]

The recording and the format of single vectors or single scalars can be defined in a configuration file as described in section 2.2.3.

This method leads to an increased dependency of result recording and the simulated system due to the hard-coded implementation. Since OMNeT++ 4.1 the newer strategies using *signals* and *statistics* are available and provide alternative methods for result recording.

2.2.2 Signals and statistics

The idea behind result recording using signals and statistics is the separation of data generation and recording. The generated data is represented and distributed via signals. The statistics access different signals for custom recording of required results. [1, section 12.1.1]

Signals provide a functionality for a communication between modules which are not directly connected via their gates. The usage of signals follows the *publisher/subscriber* principle, i.e. modules can register callback objects to a specific signal. If a new value for the signal is emitted, all registered callback objects are notified.

Signals are defined in the *NED* file of the according module or channel and can be accessed by the defined name of the signal or the resolved signal id. The name of a signal is defined globally, i.e. signals with identical names in different modules represent the same global signal. A registered listener of such a global signal receives notification of all modules which emit a new value for their signal. [1, section 4.14]

Using the *@statistic NED* property, the recording of data can be implemented using the built-in functionalities. By defining a statistic property to a signal, a filewriter listener will be registered which records all notifications of this signal. A statistic property provides multiple functionalities and options which can be defined in the *NED* file.

Declaring a simple statistic property creates a signal with the according statistic name. For separated configuration of signal and statistic, the *source* option can be used to define a signal as the source of recorded data. Built-in filters provide existing functionalities which can be directly embedded in the definition of the source signal. For example, the *count* filter can be used to count the notifications received on the defined signal. Multiple filters can be chained for manipulating the receiving data.

The *record* option defines what data shall be recorded by defining a recorder. Recorders are the final elements of the recording chain, beginning at the source signal. Between the source signal and the recorder, a variable number of filters can be set. A single statistic can define multiple recorders resulting in multiple recorded data. For distinguishing the importance of recorders, a question mark behind the recorder name can mark it as optional. Optional recorders can be skipped by default, as described in section 2.2.3. The built-in recorders like *last*, *min*, *max*, etc. result in an output scalar. The recorder *vector* results in an output vector. This generated output container represent the same functionalities as the traditional recording methods described in section 2.2.1.

There are various built-in filters and recorders available [1, section 4.15.2] and there is also the possibility for writing and using custom filters and recorders. This can be achieved by subclassing *cResultFilter* or *cResultRecorder*. [1, section 4.15.6]

The configuration of the resulting output (vector, scalar) is done independently of the used recording method.

2.2.3 Configuration

The recorded results can be individually enabled and configured. Therefore the configuration which data is recorded in which format is independent from the generated results and can be configured via a configuration file (*.ini*).

The recording using signals and statistics provides more options and possibilities and therefore requires more different configuration methods. Each statistic object can be configured individually using the *result-recording-modes* for each statistic defined by its full path within the hierarchy. This option defines the enabled recording modes (recorders described in section 2.2.2). Additionally to the available recorders, the presets *default* and *all* are available. The *default* set contains all non-optional recorders, i.e. all recorders excluding recorders marked with a question mark. The *all* set contains all non-optional and optional recorders. [1, section 12.2.1]

Statistics based recording consider a warm-up period which can be defined via *warmup-period*. Within this time beginning from the start of simulation no statistics will be recorded. When using the simulation library to manually record results (cOutVector, recordScalar), the warm-up period must be considered manually. [1, section 12.2.2]

The resulting output files can be defined independently from the used recording method via the options *output-vector-file* and *output-scalar-file*. [1, section 12.2.3]

The configuration of the recording via the simulation library (section 2.2.1) is done by *scalar-recording*, *vector-recording*, *vector-recording-interval* and *vector-record-eventnumbers*. These options allow the dis- or enabling of specific output vectors and scalars given by their full path within the hierarchy. For vector recording the definition of specific recording intervals is possible to record only interesting intervals. Also the addition of the event number to the output file can be dis- or enabled. These options also affect the recording via signals and statistics because the statistic property generates output vectors and scalars. [1, section 12.2.4, section 12.2.5]

2.3 Running an OMNeT++ simulation

The output of the OMNeT++ build process is by default an executable for starting the simulation. All generated simulation executables provide command line parameters for the configuration of specific simulation runs. The most important parameter can be passed directly or via the *-f* command and defines the used configuration files for the simulation run. The configuration files and some key options for running a simulation are described in section 2.3.1.

Within a configuration file multiple configurations can be defined and the configuration which should be used for simulation can be defined via the *-c* command line option. If no configuration to load is specified but multiple configurations are defined within the configuration file, the behavior depends on the started user interface. Starting the graphical user interface will prompt the user to choose a configuration. The command line user interface will execute the general configuration.

The configuration of the path to load the *NED* files can be defined within a configuration file as well as by a command line option (*-n*). With multiple configurations, the values are merged into a combined load path.

The selection of the user interface can be defined via the command line option *-u*. This definition overrules a configured default user interface of a configuration file. The different user interfaces and their functionalities are shown in sections 2.3.2 and 2.3.3.

OMNeT++ also provides the *opp_run* tool, which is basically an empty simulation. With this tool, a simulation model which is built as shared library can be started. With *opp_run* or any other simulation executable all available command line options and their description are acessable via *-h*.

Further configuration options of the configuration file can also be set via the command line interface by preceding the option with --. If a double configuration occurs, the values passed via command line parameter will be preferred. [1, section 10.1.1, section 10.1.2]

2.3.1 Configuration

The configuration file can be given by a command line parameter, usually the file *omnetpp.ini* is used.

Within a configuration file, multiple configurations or sections can be defined. When starting a simulation, the loaded configuration can be set, therefore multiple different simulation runs with different configurations can be defined within a single configuration file. [1, section 9.2]

The most essential configuration is the simulated network which is defined by the name of the according *NED* network.

The duration of the simulation can be defined via an ending simulated system, i.e. modules which automatically finish their execution, or a time limit is defined. A time limit can either be defined for the simulation time (*sim-time-limit*) or the cpu time (*cpu-time-limit*).

The behavior of the simulation in case of errors and the connection behavior of debuggers can be defined via various options as *debug-on-errors*, *debug-attach-on-error*, etc.. [1, section 10.1.3]

Specific configuration for the command line user interface are available for controlling the output of a running simulation (*cmdenv-express-mode*, *cmdenv-status-frequency*, *cmdenv-perormance-display*) and the possibility of user interaction (*cmdenv-interactive*). The effects of these options are shown in section 2.3.3.

Parameters of loaded components can be set individually, commonly set via wildcards, or requested from the user. Values which are directly set within the *NED* files can not be overwritten by a configuration file. Therefore a more flexible simulation model provides the majority of parameters via configurations. Parameters of modules can either be set individually by the full path within the simulated hierarchy or commonly set via wildcards. [1, section 9.3]

The configuration of module parameters also allows the definition of value ranges for so-called *parameter studies*. These *parameter studies* result in multiple simulation runs iterating over all specified parameters. Simulating a specific iteration, can be achieved via setting the run number passed by command line interface (*-r*) or defined in the configuration (*cmdenv-runs-to-execute*). [1, section 9.4]

Defined via the command line parameter or via the *user-interface* option, different user interfaces can be used to start and visualize the simulation.

2.3.2 Graphical user interface *Tkenv*

OMNeT++ provides the graphical environment *Tkenv* for running and visualizing simulations. *Tkenv* provides various different possibilities for a graphical presentation of the simulated network, processed events and simulation results.

This user interface is useful for developing and debugging a simulation during the development. The possibilities to inspect each component within the simulated system provide deep insight to the simulated system.

Different possibilities for graphical representation of the system and animation of progress/results allow a convenient presentation of a simulated system. This presentation allows for educational and presentational purposes of the simulation. [2, section 7.1]

Running complex simulations with multiple parameter studies is not recommended within Tkenv due to the increased overhead for refreshing the representation. Different run modes (*normal run*, *fast run*, *express run*) allow skipping of user interface updates. [2, section 7.3.2]

For simulations with increased complexity and longer simulation durations the command line user interface *Cmdenv* should be preferred.

2.3.3 Command line user interface *Cmdenv*

The *Cmdenv* environment represents a command line user interface. Using this environment, no graphical user interface is shown, or will be updated. This simulation method is recommended for batch simulations or running simulations with increased complexity and no need for graphical representation due to the improved performance.

During a running simulation within *Cmdenv* the output printed to the command line depends on the configuration. For debugging purposes, the *normal mode* can be used and detailed event information will be printed to the command line. When running simulations over a longer period of time, the *express mode* is recommended and only periodically status updates will be printed. The frequency of the updates and the printed details can be configured via the *cmdenv-status-frequency* and the *cmdenv-performance-display* options. [1, section 10.2.3]

Simulations defining one or more parameter studies result in multiple runs which can be defined via a command line parameter or in the configuration. For executing multiple runs, OMNeT++ provides the *opp_runall* tool which starts each run in a separate operating system process. Using multicore/multiprocessor systems *opp_runall* can execute different simulation runs on different cores/processors. [1, section 10.4.3]

Running the simulation within *Tkenv* or *Cmdenv* executes it by default sequentially. Parallel simulation imposes different requirements on the simulated systems and its design. These requirements and the functionality of parallel simulation is analyzed in chapter 6.

2.4 Simulation Core

Simulations based on OMNeT++ are designed and simulated as discrete event simulations (*DES*). The characteristics of the *DES* and the comparison to other types of simulation are discussed in the chapter 3.

Details about the different simulation properties and functionalities are outlined in section 3.4.

Chapter 3

Simulation

Simulations attempt to replicate and forecast the behavior of real world systems. Such a replication is used in various fields for testing and verifying theories and systems. The field of simulation steadily gains importance, due to the increasing complexity of systems; e.g. embedded systems and real-time systems. Especially the development and testing of real-time communication systems can essentially be improved by using simulation and emulation techniques.

Different types of simulation are applicable for different types of simulated systems and aimed results. Differences are shown in the processing of the simulated systems and in the handling of simulation time. [16, section 1.2]

3.1 Continuous simulation

Continuous simulations handle uninterrupted values over a simulated time range. This behavior may be determined by equations describing the system. For correct simulation of a continuous system, the model must be executed for the whole simulated time range. These simulations are usable for scenarios when the temporal behavior of the simulated model is of interest. [16, section 1.2.1]

Simulations and testing scenarios in the field of real-time communication are based on discrete events at specific points in time, e.g. the reception of data. The continuous processing of occasions between events is often not necessary, therefore a discrete event simulation (*DES*) is more applicable.

3.2 Discrete event simulation

This type of simulation is based on processing discrete events. During the processing of events, the simulation time does not advance and the required processing time is not considered in simulation time. The simulation time advances with multiple processed events and their defined point in simulation time. Between two consecutive events no processing is done and no changes of the system state occur. [15, chapter 1]

The assumption is that no, for the simulation relevant, events happen between two consecutive events. This exclusion is done by the implementation of the simulated model and must be concluded with care to the intention of simulation. Hence simulating a real world system requires the filtering of occasions for focusing on the simulation goal. [1, section 4.1.1]

The implementation of a *DES* can be done in various ways using different strategies or paradigms. In [15, chapter 2] Matloff introduced different paradigms to realize a *DES*. Using OMNeT++ the *Event-Oriented Paradigm* and the *Process-Oriented Paradigm* are applicable and can be achieved as following:

- The usage of the *handleMessage* method matches the *Event-Oriented Paradigm* and allows, for event based development.
- The *Process-Oriented Paradigm* can be realized by using the *activity* method and represents the *process style* strategy.

The *Activity-Oriented Paradigm* could also be implemented within OMNeT++ by using either the *activity* or the *handleMessage* method. This paradigm would require the implementation of a custom polling module which checks regularly for new events or monitors the activity of other modules. This implementation would replace and bypass the handling and scheduling of messages in OMNeT++ and is not recommended. [15, chapter 2.1]

The above discussed types of simulations belong to the group of *offline simulations*, i.e. the simulation time is not connected to the processing time. Approaching the fields of emulation and *HiL* such a connection is necessary and the behavior of *offline simulations* is unusable. These fields demand the type of real-time simulation. [5, section III.B]

3.3 Real-time simulation

Real-time simulations change the meaning of simulation time and add a connection to the real-time. The simulated events should be executed at the correct time to match the real-time. In this context, the real-time represents the real world time, cpu time, or wall time, i.e. the time which passes for the real world during the execution of the simulation. Running a real-time simulation results in processing a simulated second within an elapsed real world second. This type of simulation is not possible for every simulated system as the limits are defined by the time required to execute the operations specified by the model.

The achieved execution speeds strongly depend on the following factors:

Model The complexity of the simulated model, i.e. the functionality to process events affects the achievable execution speed. Simple functionalities can be executed faster than complex library calls or nested functions.

Host system The properties of the host system used for running the simulation define the possible execution capabilities and therefore the performance of the executed simulations. This dependency is described in section 3.5.

If the execution of the model behavior for reacting to an event takes more processing time than the simulated duration (timespan to next event), the simulation lags behind the real world behavior. Such an erroneous behavior must be corrected by the simulation core by speeding up the simulation, for example by decreasing idle times between events. The correction of such behavior results in an increased jitter (variance of event execution time). [5, section III.B]

The quality of the real-time simulation strongly depends on the simulated system and its composition. Therefore, the ideal results can be achieved by analyzing the simulated system regarding the real-time requirements and processed events. An example of such an analysis and the modifications of an OMNeT++ simulation for timing improvements are shown in [29].

The existing functionalities and properties of a simulation using the OMNeT++ framework are outlined in the next section.

3.4 Simulation with OMNeT++

By default, simulations developed with OMNeT++ are discrete event based simulations. This behavior is defined by the scheduler and the simulation core, which can be tailored by using custom components. [1, section 4.1]

Within OMNeT++ each event is represented by a message with a defined *arrival time*. Events are created by modules and then inserted in the so called future event structure (*FES*). The simulation core executes all events within the *FES* at the according simulation time.

The scheduler is the main part of the simulation core for controlling the event handling and the execution order. The scheduler accesses the *FES* and chooses the next event to be handled by the simulation. The class *cScheduler* represents the interface which is required for an event scheduler usable in OMNeT++. By default, the derived class *cSequentialScheduler* is used. This scheduler implements the default discrete event based simulation and handles the events according to their execution time, scheduling priority and scheduled time. The scheduling priority provides a mechanism for controlling the execution order of multiple events at the same time. [1, section 4.1].

An exemplary approach to realize the type of real-time simulation is implemented in the *cRealTimeScheduler*. This scheduler executes the events according to their planned arrival time. The arrival time of the next event is compared with the current real time. When the simulation is ahead of the real time, the simulation is paused for the remaining time. The *cRealTimeScheduler* waits in hard-coded 100 ms chunks for allowing a responsive simulation including graphical updates. This provided scheduler does not handle a lagging simulation in a special way and simply skips the waiting times within the method *getNextEvent*. The definition and implementation of the *cRealTimeScheduler* are shown in the appendix section A.1 or in the OMNeT++ API [18, cRealTimeScheduler].

This concept is not applicable for emulations and *HiL*, because the communication with real components does rarely contain static sleep times. The OMNeT++ sample *sockets* demonstrates this problem and a possible solution with a custom scheduler implementing the *cScheduler* class. The custom scheduler *SocketRTScheduler* and its functionality is further analyzed in section 4.1.1.

Handling a simulation which is faster than the real world system can be done in various ways as demonstrated by *cRealTimeScheduler* or the *SocketRTScheduler*. If the simulation is lagging behind the real-time, the scheduler must try to speed up the simulation and catch up with the real time. This is very difficult for complex simulations with tight timings. If the simulation lags constantly behind the real world using the *cRealTimeScheduler*, it becomes a discrete event based simulation and no real-time simulation is possible.

The sample scheduler provided by *cRealTimeScheduler* and *SocketRTScheduler* can lead to the correct strategy of implementing an optimized scheduler.

To validate a real-time simulation regarding timing quality, the performance ration can be used. This ratio represents the simulated seconds per real time seconds. A lagging simulation is defined by a performance ratio of less than one and simulation which simulates faster as the real time shows a ratio greater than one. The goal of a real time simulation is a constant ratio of one. The process of catching up of a lagging simulation to achieve a performance ratio of one can also influence the general timing behavior. Therefore the variation of delays (jitter) increases when the simulation lags temporarily. For emulations or the fields of *HiL* an increased jitter for a signal can be very critical and must be analyzed carefully.

The host machine for the simulation and its components affect the achieved simulation times. The dependencies of the host system and the results of existing research is shown in section 3.5.

Developing the simulation of a given system results in the situation of existing code. This code must be encapsulated in different modules and executed depending on incoming messages and thereby creating new messages for sending. Given systems can be designed in various hierarchies in view of the number of modules and complexity of simple modules. The different designs and their effects on real time simulation are shown in chapter 5.

3.5 System requirements

The host system for the simulation is very important regarding both speed and performance of a simulation and the achievable timings of a real-time simulation. The limitations of achievable timings of real-time simulations are defined by the execution speed of the simulated model plus the time for executing the simulation functionality around it. Assuming the RAM (random access memory) of the host systems is large enough to hold the complete simulation code and data, the limit is defined by the execution speed of the code, which is affected by the CPU (central processing unit) capabilities and the speed of connected memories. These memories include the RAM, every cache and register which is used for holding simulation code and data. The evolution of simulations and real-time simulations arises from analog simulations, to digital simulations running on supercomputers and currently common simulations running on commercial of the shelf ($CTOS$) systems and field programmable gate arrays ($FPGA$). This evolution provides more computing capabilities for lower costs. [5, section IV]

Chapter 4

Emulation and Hardware in the Loop

The field of emulation is strongly related to the field of simulation and targets the imitation of real world systems. In contrast to a simulation, the purpose of an emulation is to replicate a real world system and providing the possibility to interact with other components in the same way as the emulated system ought to. Emulation is used in various fields reaching from the replication of outdated computer systems for executing old applications, to the field of *HiL* (hardware in the loop) and the usage for verification and testing of embedded systems. [33]

The fields of emulation and *HiL* are based on real-time simulations combined with a connection to the real world. The replication of a given environment allows the developer to enclose a specific component and then execute specific scenarios. Such a component can either be a software application (*SiL* - software in the loop) or hardware component (*HiL*). This connection of simulated systems with real systems can be used for testing and verification of systems under development. An emulation system running on a host system and an enclosed system under test is shown in Figure 4.1. The shown structure is typical for various types of emulations with different enclosed systems.

Such a test method can test various different scenarios due to the flexibility of the simulated system. [14, section I]

The used simulation can be realized with any simulation technology and existing simulation frameworks, as long as the execution as real-time simulation (described in section 3.3) is possible. The requirement of real-time simulation must be met for specific given response times to allow for valid interaction with external components. In the following section, the capabilities regarding emulation and *HiL* provided by the OMNeT++ framework are discussed and analyzed.

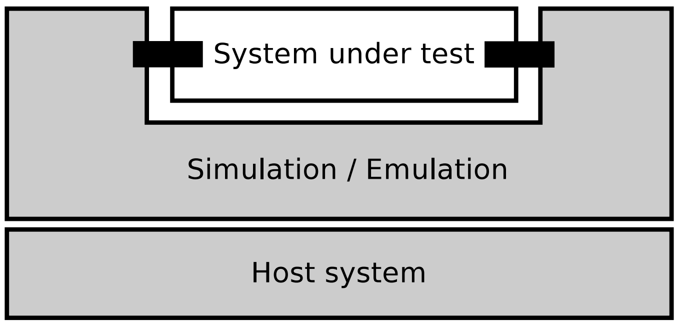

Figure 4.1: Overview of an emulation environment running on a Host system (gray) with an enclosed system under test (white) and the connections between those systems (black).

4.1 Emulation and *HiL* using OMNeT++

For the fields of emulation and *HiL* OMNeT++ provides customizable components within the simulation core and thereby allows for different strategies to implement the required behavior. The built-in functionalities and their usage shown in given examples are explained in the following section.

4.1.1 Existing functionality

The sample simulation *sockets* shows the possibilities and methods for developing an emulation. This example simulates a web server with a dynamic number of clients. The exemplary usage of an external client is also possible, which represents the connection to the real world and communicates with real requests by the user. This interaction is established via the connection to a local address and the hyper text transfer protocol (*HTTP*) GET request by a web browser. This emulation uses the custom scheduler *SocketRTScheduler*, which is implemented similarly to *cRealTimeScheduler* and derived from *cScheduler* [18]. The custom scheduler holds a TCP socket for communication with the real world. During wait times the schedule listens to the network interface and converts receiving data to simulation events. These generated events are distributed by the *ExtHttpClient* or *ExtTelnetClient*. These simple modules represent the external real world components within the simulated network. The implementation can be found within the *sockets* sample included in the OMNeT++ framework or in the appendix section A.2. This behavior is feasible for this exemplary usage, but if the timings of the simulated systems sharpen this scheduler would not allow for sufficient communication. Analyzing the implementation and the achievable timings leads to different possibilities for optimizations as described in [29].

The combination of a real-time simulation and a real world system requires a connection between them. Such a connection can be established by using various functionalities, but must always fulfill the functionality of converting occasions from the real world to simulation events and vice versa. This connection affects the achievable performance and the temporal behavior of the simulated components respectively to the real world component. In the *sockets* example the built-in functionalities for sending and receiving data via sockets are used. These built-in functionalities are implemented depending on the host platform OMNeT++ was built and therefore use either the *WinSock2* library on Windows operating systems or otherwise the Unix socket implementation. These built-in functionalities for socket communication are located in the *include/platdep* folder within the OMNeT++ installation. In section 4.2, different strategies and recommendations regarding the communication implementations are discussed.

4.2 Communication with the real world

For the fields of emulation and *HiL* the communication with the real world is very important and affects the achievable performance. By encapsulating all used communication functionalities in specific modules, the simulated model can be clearly separated. Using OMNeT++ the separated communication components can be realized with modules implementing the connection functionalities or customizing simulation components such as a custom scheduler. Recommended implementation strategies for each communication functionality and their properties are discussed in the following sections.

4.2.1 Receiving

The receiving component can be implemented as a simple module using the *process style* strategy. This strategy allows for an intuitive implementation observing the interface, then creating messages with the received data and sending them to the simulated system. Executing the simulation sequentially does not permit constant listening by such a receiver module. Thus, it must be interrupted to allow the execution of the remaining simulation. Using parallel simulation described in chapter 6, the receiving module can be implemented in blocking. Assigning the receiving module to a single processor allows for a constant listening to the communication interface. This represents an improved behavior and extended possibility for handling real time occasions and provides a potential lowered delay for the event conversion.

As shown in the *sockets* sample the customization of a simulation component, e.g. the scheduler can also make the receiving of external data without constantly blocking the simulation possible. For this execution method the scheduler *SocketRTScheduler* provides a customized scheduler implementation with a similar strategy as the built-in *cRealTimeScheduler*. This implementation strongly depends on the simulated idle times, because only during these times the reception of data is possible. The strategy of simulating a system with few idle times can lead to increased delay times until an external occasion is converted to a simulation event.

4.2.2 Sending

The sending component can be implemented as simple module either by drawing on the *process style* or the *event based* strategy. If the sending functionality inevitably blocks, this module should be implemented drawing on the *process style* strategy and be executed on a separate processor, when using parallel simulation. Running a sequential simulation, a blocking sending functionality must be used with care of the blocking behavior. Such behavior could be compensated with timeouts and retries after a defined waiting time while the simulation can proceed.

A similar approach to the receiving implementation using a customized scheduler can also be applied for a non-blocking sending. The prepared data for transmission could be buffered and sent by the scheduler during idle times. Such buffering affects the timing behavior and must be analyzed regarding achievable response times to external occasions.

For each functionality and depending on the targeted behavior, different implementation strategies and design decisions must be made. Fundamental design strategies regarding the structure of a simulation and their properties are discussed in the next chapter.

Chapter 5

Design structures

Implementing any kind of simulation or emulation can be done in various structures and designs which can be tailored for specific requirements. Given an existing system, different questions regarding the complexity of modules and the designed hierarchy arise. Such a given system can be an existing application or other given functionalities, which must be embedded within the simulation without changing the original system. The following sections discuss two fundamental design strategies and their impact on development and achievable results. Furthermore, the according functionalities and strategies within OMNeT++ are explained. An according example simulation and the resulting performances are shown and discussed in chapter 7.

5.1 Modular Design

A modular design implies a bigger number of different components communicating with each other. Different functional units from the original system are represented by the composition of multiple components and their interactions. This approach can lead to a more dynamic simulation and increase the reusability of different components and modules. Developing a complex systems can also be eased by using a modular design and therefore an increased separation in smaller functional units.

In addition a modular design can provide more insight into a given system and its executed procedures. This deeper insight can be used for educational and exemplary usages showing detailed information about the internal procedures of a functional unit. During development of the simulation or the simulated system, a modular design can also provide a clearer picture that leads to improved analyzing and debugging possibilities. The increased number of simulated components and the separation of functionality compared to a monolithic design results in increased communication between components. Designing a modular system in OMNeT++ is done by implementing a bigger number of modules connected with channels and grouped in com-

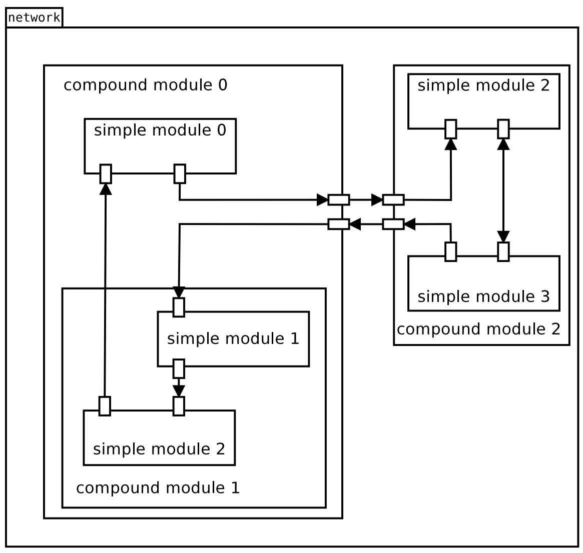

Figure 5.1: Modular design with OMNeT++

pound modules. A simulated network consisting of multiple modules and therefore implementing a modular design results in increased communication, i.e. message allocation, transmission and deallocation. These additional procedures can strongly influence the achievable performance and must be analyzed with care.

An example of a modular design using OMNeT++ components is shown in Figure 5.1. This example contains multiple compound modules consisting of enclosed simple and compound modules. The multiple connections between the modules show the required communication.

Approaching the fields of emulation and *HiL* using real-time simulation, these design decisions are important for the achievable timings. As described in section 6.3, OMNeT++ provides functionalities for running a parallelized simulation and therefore adds to an increase in the performance by parallelism. This type of execution strongly depends on the simulated system and the applied design. Due to the necessary communication between parallel simulated modules, a modular design provides more possibilities for parallelism than a monolithic design.

Developing a simulation for a given system with an existing implementation results in restraints for the applicable design. The simulation of a system which shows a modular structure can also be designed modularly by using the separations of the given system. For a modularly designed simulation, the connection of simulated system and according simulation modules must be possible. This connection can be achieved when the existing system already uses interfaces or function pointers for the connections of different components. These communication parts can be used for redirecting calls to

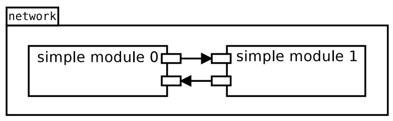

Figure 5.2: Monolithic design with OMNeT++

wrapper modules embedding the simulated system in the simulation environment. These modules handle all received messages, forward them to the enclosed component and create necessary outgoing messages. The enclosed component can be instantiated within the wrapper module and thence embedded in the simulation.

5.2 Monolithic Design

The opposite approach to the modular design discussed in the previous section is the monolithic design. Which implies a lower number of components, but an increased functionality within a single one. This decreased number of components reduces the necessary communication and a potential overhead. The reduced modularity does not allow deep insight into the system but can result in improved performance. If a simulated system is used as a single functional unit and maybe even instantiated multiple times within the simulation, a monolithic design appears favorable, especially when no detailed information about internal procedures is demanded. Within OMNeT++, a monolithic design can be achieved by condensing a compound module to a simple module with more complex functions. The execution of the single modules include more normal C++ code using simple method calls and operations. This can be done via enclosing multiple components of the given system within a single simple module. Such components can be connected directly to each other, such as the normal execution regardless of the simulation environment. The surrounding module must, similar to the implementation of modular wrapper modules, handle and forward incoming messages or create outgoing messages for the communication with the remaining simulation. The previous example network showing a modular design in Figure 5.1 can be transformed to a more monolithic design. In the example, shown in Figure 5.2, the compound modules 0 and 2 with all their submodules were replaced by two simple modules. The channels and the sent messages between the modules stay the same, but the calculation within the modules include the complete behavior of the previously included components.

As mentioned in this and the previous chapter, a parallelized simulation can provide improved behavior and benefit from specific designs. The properties of a parallel simulation and the provided functionalities of OMNeT++ are analyzed in the next chapter.

Chapter 6

Parallel Simulation

A sequential simulation is executed on a single processing unit, either on a logical processor, physical processor or a single machine within a cluster of multiple systems. Simulating complex models and a huge number of events can result in a long execution time. Increasing complexity of different systems and their simulations lead to the demand for an improved execution method. The distributed discrete event simulation (*DDES*) or parallel discrete event simulation (*PDES*) allows for the parallel execution on multiple processors. The goal of parallel execution is the improvement of simulation performance and reduction of required execution time. Within a parallel simulation, events are processed by different logical processes (*LP*). Each *LP* can be executed by a different separate processing unit and therefore executed in parallel. [4]

This distributed processing of events requires synchronization for guaranteeing the causality constraint, i.e. the processing of all events within a *LP* is done with increasing timestamps.

6.1 Synchronization

Running simulations in parallel requires a synchronization between the *LPs* to harmonize the processed simulation time. This simulation time can diverge strongly due to the different sets of processed events. With varying event intervals (i.e. the time between two consecutive events) the steps within the simulation time are differing. For the synchronization of parallel *LPs* different algorithms are available which show contrasting characteristics and effects on the achieved performance of the *PDES*. [3, chapter 2]

The two basic types of synchronization algorithms are explained in the following sections.

6.1.1 Optimistic Algorithms

Optimistic algorithms allow *LPs* to process events without guaranteeing that any new event may be scheduled at a previous point in simulation time. This behavior could lead to a violation of the causality constraint. In the case of such a violation, an event is scheduled at a point in time within the past according to the local simulation time of a *LP*. For solving this violation, a *roll back* mechanism is necessary to rewind the according procedures and allow for the correct execution of all scheduled events. A simple version of such a *roll back* mechanism is the periodical saving of the system state and restoring an old valid state in case of a causality violation. [4]

Regarding the fields of emulation and *HiL* this behavior is not applicable, because with a connection to a real world component no rewinding of the simulation time is possible. Therefore another type of synchronization is necessary.

6.1.2 Conservative Algorithms

Conservative algorithms strictly forbid the processing of events until it is guaranteed that no new event will be scheduled before the local simulation time.

An example for a conservative algorithm is the *Null Message Algorithm* (*NMA*) [3, section 2.1]. The *NMA* is transmitting so-called *Null messages* between all connected *LPs*. These *Null messages* contain information about the duration in which no event will be scheduled by the sending *LP* for the receiving *LP*. If no events are locally scheduled within this transmitted duration, the receiving *LP* can advance its simulation time to the next local event or to the end of the duration. Each *LP* also has to determine this duration for each one of its connected neighbor *LPs*, which will be transmitted to the according *LPs* and provide the same information about a guaranteed duration with no scheduled events. [12, section 3]

This algorithm is subject to different performance analysis and improvement studies [12] [28] [32]. The determination of the duration for transmission and the frequency of this procedure strongly affects the achievable performance. A badly configured *NMA* can lead to a high number of transmitted *Null messages*, which can be either condensed or skipped completely. This overhead of unnecessary transmissions results in a weakly performing simulation. Optimizations of the *NMA* can be realized by reducing the number of transmitted *Null messages* and improving the generation algorithm [6]. Such an optimization can benefit from knowledge about the simulated system and its temporal behavior. This leads to an individualized synchronization algorithm based on the *NMA*. A corresponding optimized synchronization algorithm can lead to a greatly improved performance, but is only applicable for a specific system.

Every synchronization algorithm, belonging to one of the discussed types, requires the transmission of synchronization data. This transmission demands a communication facility between *LPs*. The following section describes existing communication functionalities and their properties.

6.2 Communication

Depending on the used hardware and the type of distribution of the processing units, different communication methods can be used. Such a communication belongs to the groups of inter-thread, inter-process or inter-machine communication. Most of the existing functionalities of these fields are applicable to synchronization communication.

The choice of the communication method depends on the used simulation technology, the type of parallelization, distribution of processing units and also the used synchronization algorithm. For each possibility of every dependency, different communication methods are appropriate and result in varying performances.

A very common method for synchronization communication is a message based technology which can be applied to various underlying methods for data transmission. This method is called message passing interface (*MPI*) and uses transmitted messages via various transport technologies. On different operating systems and depending on the specific simulation setup a *MPI* can use various transport technologies. [30]

A various number of technologies for parallel simulation, synchronization and communication are available. The built-in functionalities of OMNeT++ and their characteristics are analyzed in the following section.

6.3 Parallel Simulation with OMNeT++

OMNeT++ supports the execution of simulations parallelized using different synchronization and communication methods. OMNeT++ targets the independence of the simulated model to the execution method. Therefore, the parallel execution of an existing OMNeT++ simulation should not require modifications of the implemented model. For achieving this independence, OMNeT++ uses the built-in transmission of messages between modules for the according communication between *LPs*. The message based communication between modules allows for the redirection of those messages over communication methods without changing the implementation.

This is done via the configuration files as described in section 2.3.1. The configurations regarding parallel simulation include the used synchronization and communication methods and the assignment of the simulated modules to partitions. A partition represents a *LP* and runs on a separate processing unit. Multiple modules can be assigned to one partition and thereby simulated on the same processing unit. Running a parallel simulation with OMNeT++ is done by starting multiple instances of the simulation executable with the partition to execute given by a parameter.

The configuration for a *PDES* using an OMNeT++ model uses placeholder modules and proxy gates. These components are introduced when using *PDES* features. A placeholder module is inserted instead of a module which is simulated on a different partition. This inserted placeholder module represents the replaced module. Therefore, the hierarchy and structure of the simulated module stay untouched for a *PDES*. The placeholder module provides a proxy gate for each gate of the replaced module. These proxy gates forward the received messages to the correct *LP* and also receive corresponding messages from the remotely simulated module and send them to the local network. [31, section III] [1, chapter 16]

The introduction of placeholder modules and proxy gates and the assignment of modules to specific partitions is independent of the used synchronization and communication method. The provided functionalities and possibilities for customization of synchronization and communication are analyzed in the following sections.

6.3.1 Synchronization

OMNeT++ uses a configurable synchronization algorithm for the synchronization of multiple partitions. These algorithms are represented by C++ classes derived from *cParsimSynchronizer* and can implement their algorithm using specific hook methods which are explained in the following listing. The built-in *NMA* implementation is considered for example usages of the different hook methods.

Event scheduling The implementation of the *getNextEvent* method derived from *cScheduler* provides full access to the *FES* and the next scheduled event. Using this method, the synchronization algorithm can block the simulation, remove and introduce specific messages and thereby control the executed messages. The *NMA* uses this hook for blocking the simulation during wait times and for introduction of *Null messages*.

Outgoing Messages This hook is represented by the implementation of the derived *processOutGoingMessage* method and allows for the modification of all messages which are sent to a different partition. The *NMA* uses this hook for including *Null message* information in the transmitted messages and thereby reduces the number of necessary *Null messages*.

Incoming Messages This hook is represented by the *processReceivedMessage* and *processReceivedBuffer* methods which are derived from *cParsimProtocolBase*. The *NMA* uses this hook for handling received *Null messages*.

As mentioned above, the hooks are represented by methods derived from different classes. The base class for all synchronization algorithms *cParsimSynchronizer* is derived from *cScheduler* and is used as scheduler while executing a *PDES*. The class *cParsimProtocolBase* is derived from *cParsimSynchronizer* and provides convenient methods for eased development of synchronization algorithms. Therefore, *cParsimProtocolBase* is the recommended base class for custom implementations.

OMNeT++ also provides some built-in implementations of synchronization algorithms which are listed below. [1, section 16.3.5]

cNullMessageProtocol implements the *NMA* algorithm with an enclosed class for lookahead calculation.

cIdealSimulationProtocol implements an analysis and benchmark method for given synchronization algorithms.[3, section 3].

cNoSynchronization implements an empty synchronization, i.e. all events are executed as scheduled by the model and no synchronization is happening.

Either custom implemented synchronization methods or built-in algorithms can be defined within a configuration file by the *parsim-synchronization-class* configuration. Specific implementations can also provide further configuration possibilities. [1, section 16.3.5]

6.3.2 Communication

The used communication method within a *PDES* using OMNeT++ is also configurable and customizable like the synchronization algorithm. The communication layer within OMNeT++ provides basic message transmission functionalities like send, blocking or nonblocking receive and broadcast. These methods are defined in the base class *cParsimCommunications* which must be derived for implementing an usable communication method. The used communication class is defined via the *parsim-communications-class* configuration. [1, section 16.3.5]

The implemented methods use a pointer to the class *cCommBuffer* which provides packing and unpacking functions. For specific buffer functionalities according to a custom communication the class *cCommBufferBase* is derived from *cCommBuffer* and represents the recommended base class for custom buffer. [19]

The following built-in communication methods are provided within OMNeT++ and can be used for *PDES*. [1, section 16.3.5] [19]

cFileCommunications represents the communication via textfiles. Due to the necessary file operations and the big dependency on the disk where this file is located, this communication method is more recommended for debugging purposes. This communication method can be used between all partitions which have access to a commonly reachable file system.

cNamedPipeCommunications is using named pipes for the transport of messages. The performance depends on the underlying operating system providing the named pipe functionality. Using named pipes, the communication can also be used between separate machines within a computing cluster.

cMPICommunications represents the communication using a *MPI* system. As described in section 6.2, this communication can use various transport technologies. For usage of *cMPICommunications* a *MPI* implementation must be present at the hosting system. For different operating systems, various *MPI* libraries such as Intel MPI [11], MS-MPI [17], openMPI [20] and many more can be used.

The communication between *LPs* is based on the transmitted messages among modules. This defines that among modules no direct communication using simple method calls are permitted. This restraint and the need for more requirements to run a *PDES* using OMNeT++ are explained in the following section.

6.3.3 Requirements

The execution of OMNeT++ models as *PDES* is possible when specific requirements regarding the implementation and the communication among modules are met. The following requirements are caused by the used strategy of replacing remote modules by placeholder modules with proxy gates. [31, section III.B]

1. Communication among modules is only done via message transmission over channels, i.e. no direct calls or member access is allowed. Direct sending can be used, but limitations regarding the sending to submodules must be considered.
2. No usage of global variables or public static member variables.
3. Static topologies.

Requirement one and two only apply to modules which are assigned on different partitions, because the usage of global variables, direct sending or direct method calls would bypass the message transmission. Within a single partition these functionalities can be used because no message transmission is required. Following the OMNeT++ user guide, design and implementation recommendation these requirements are already met. [31, section III.B]

Using the built-in *NMA* implementation, an additional requirement for transmission among partitions is defined. The implementation of the lookahead calculation for the *NMA* demands a non-zero delay between partitions, i.e. the channels among modules assigned to different partitions must provide a non-zero delay. [1, section 16.3.1]

The basic strategy for assigning separate modules to different partitions and therefore *LPs* affects the possibilities for parallelism depending on the structure and the used design of the simulated model. The following chapter shows the analysis of an example network inspecting the achievable performance using different designs and different execution methods.

Chapter 7

Design Evaluation

For measuring the effect of different designs on the simulation's performance, an example network was implemented using a monolithic and a modular design. These implementations represent examples of the designs discussed in chapter 5.

7.1 Simulated Example Network

The example network simulates a message queue with dispatching of different types of transmitted data (configuration, event, historical). The different types of data are processed by different parts within the network. This network includes parts for data generation and data processing. Such an exemplary network was chosen, in view of multiple similar practical applications. An overview of the simulated network is shown in Figure 7.1.

The *Generator* generates cyclic data which is transmitted via messages to the sink module. The generated data includes a field of 64 bytes and an enumeration describing the type of data.

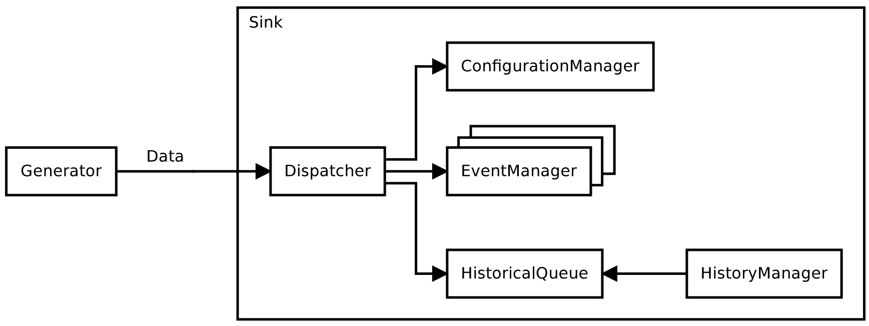

Figure 7.1: Example network including *Generator*, *Sink*, *HistoricalQueue* and the data processing modles *ConfigurationManager*, *EventManager*, and *HistoryManager*.

These messages are transmitted to a sink, which is described via the interface module *ISink*. The differently designed modules *ModularSink* and *MonolithicSink* extend the interface and represent the differently tested designs.

The first part within the sink is the *Dispatcher* which accesses the type information of the data and then forwards the packed data to the according managers or the *HistoricalQueue*. The simulated network provides a variable number of *EventManagers*, all generated events are dispatched to the *EventManagers* sequentially. The *ConfigurationManager* and the *EventManagers* are simple implementations which executes various calculations on the received data to simulate processing. Historical data are forwarded to the *HistoricalQueue* which is internally implemented by a std::queue that holds all received data until they are accessed. The *HistoryManager* accesses the *HistoricalQueue* and processes available data similar to *ConfigurationManager* and *EventManager* by executing dummy calculations. This access is initiated by the *HistoryManager* itself and provides a configurable polling interval.

The functionality of dispatching and processing the data is, as shown in Figure 7.1, included in the sink and is implemented twice with different designs.

The assumption of existing implementations for the *Dispatcher*, *HistoricalQueue*, *ConfigurationManager*, *EventManager* and *HistoryManager* is made. This assumption should connect this design test to practical scenarios, where existing code cannot be changed for the simulation. Therefore, implementing the *MonolithicSink* consists of instantiating and connecting the different parts within a single simple module. Received messages will be analyzed and the enclosed data is forwarded to the according instances. The polling is done by the *ConfigurationManager* using *self-messages* sent in an configurable interval.

Implementing the *ModularSink* requires the implementation of wrapper modules for every single part which should be represented by a separate module. These wrapper extract the transmitted data of received messages and forward them to the enclosed parts. Calls from within the enclosed parts are handled by methods of the wrappers, which are passed via function pointers (functional objects). Within those methods relevant messages are created and sent via the correct output gates. The internal structure of the *ModularSink* is shown in Figure 7.2. Within the *ModularSink* arrays of gates, connections and instances of *EventWrappers* are used for realizing a variable number of *EventManagers*.

The simulated network consists of the *Generator* instance and an instance specializing *ISink* the underlying module and therefore the tested design is configured via a variable type. The resulting network for data generation is shown in Figure 7.3.

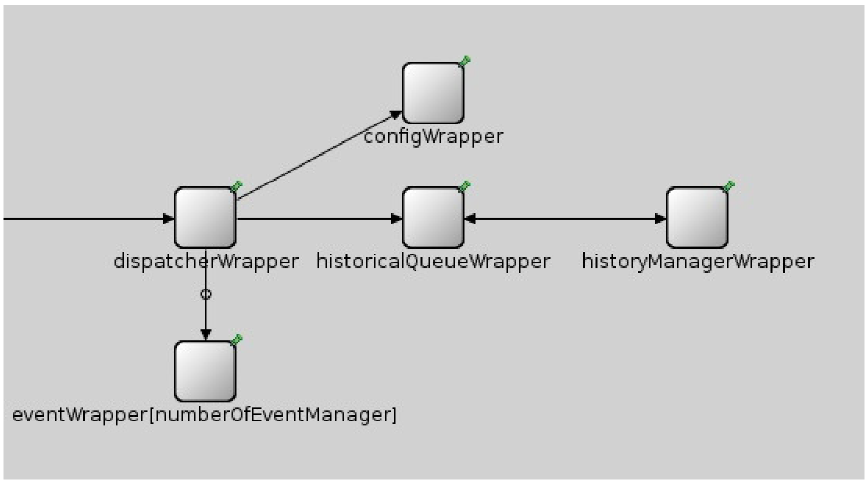

Figure 7.2: Structure of *ModularSink* showing the implemented wrapper modules and their connections.

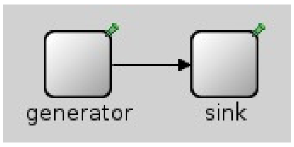

Figure 7.3: Simulated example network showing the *Generator* instance and its connection to the instance of the sink module derived from *ISink*.

7.2 Measurement Methods

For measuring the performance of the different designs, three measurement methods were implemented. The implementation of different methods was done for preventing false assumptions based on the result of a single test. These methods were implemented via different scripts for automated execution and designed dynamically allowing for the analysis of various simulations and also either sequential or parallel simulation.

7.2.1 Runtime Measurement

The performance of the simulated design can be analyzed by defining a fixed simulation time limit (*sim-time-limit*). The usage of the default (non real-time) scheduler results in a simulation which executes as fast as possible for the required number of events for reaching the simulation time limit. The required execution time for this simulation represents the performance of the simulation.

Analyzing a parallel simulation using this method does not require special attention, except the handling of multiple resulting runtime values.

7.2.2 Processed event count

By defining a fixed execution time limit (*cpu-time-limit*) and using the default scheduler, the simulation will run for a fixed time. The number of processed events within this fixed time represents the performance of the simulation.

The simulation of different designs results in contrasting event counts due to the varying number of messages using other designs with more modules and therefore more communication. For a comparison of the performance, the created number within a fixed processing time a correction factor must be considered. This factor describes the ratio of created events using the different designs. This ratio can be measured by using the configuration of the previous method 7.2.1 and analyzing the number of created events.

Analyzing a parallel simulation and comparing the performance to a sequential simulation requires further attention to the number of created events. An additional correction factor is necessary due to the varying number of sent synchronization messages. Because of this additional correction factor and the increasing uncertainty coming with it, this measurement method was not used for parallel simulation.

7.2.3 Real-time Behavior

Using the built-in real time scheduler *cRealTimeScheduler* the simulation will try to execute the simulation matching the real time. The performance output (*cmdenv-performance-display*) provides the ratio of simulated seconds per real second. As described in section 3.4, this ratio must not differ too much from one for representing a real-time simulation. The simulated network provides a configurable interval of data generation by *Generator*. Using a parameter study (described in section 2.3.1) the interval for data generation by *Generator* can be set to values form a range of intervals. With attention to the performance ratio of the different iterations, the interval limit, which still allows for real-time simulation, can be determined.

Running an OMNeT++ simulation parallelized demands the definition of a synchronization class implementing the used synchronization algorithm as described in section 6.3.1. A synchronization class used in OMNeT++ is derived from the base class *cParsimSynchronizer*. This base class implements a *cScheduler* and during a parallel simulation, it is also used as a scheduler. To achieve a parallel real-time simulation a custom synchronization class enclosing the behavior of a real-time scheduler must be implemented. Due to the lack of such a built-in synchronization class the measurement method for testing the real-time behavior was not used for testing the parallel simulation.

7.2.4 Result Recording

The results of the different measurement methods are all extracted from the resulting command line output of the simulation. The automated test scripts analyze the outputs of the executed simulation after finishing the simulation. For preventing the delay of the measurements and the simulation by writing the output to a file located on a slow peripheral, the output files are located on a *ramdisk*. A *ramdisk* represents a filesystem which is located within the *RAM* (Random access memory) and therefore provides the maximum speed for writing and analyzing outputs.

The implemented test network described in 7.1 was developed and analyzed on a Lenovo ideapad U530 with 8GB PC3-12800 DDR3 SDRAM 1600 MHz and a 4th Gen Intel® Core™ i7-4500U (1.8 GHz 200 MHz 4MB) running Kubuntu 15.10. [13]

7.3 Sequential Simulation

Each of the three developed measurement methods was executed multiple times with the example network. To eliminate a possible dependency of the resulting performance to the simulated range (e.g. period of simulation time), each method was executed with different times. The runtime and real-time method was executed for multiple values for the simulation time limit and the event method was executed for different cpu time limits. This variation of simulation time verifies the testing measurements for independence of the simulated time range and provides reference values for determining correction values for the event method 7.2.2.

By simulating multiple scenarios, three parameter studies (parameter sweeps) were executed for each testing method.

- Number of EventManager (default = 1)
- Generation interval of *Generator* (default = $100\mu s$)
- Polling interval of *HistoryManager* (default = $100\mu s$)

During the simulation with different values of a single parameter study the other values are set to the default values. For the testing scenarios, the simulation time limit for the runtime and real-time method or the cpu time limit for the event method was set to one minute arbitrarily. In the following section, the measurement result of the evaluation test (sweep of simulation or cpu time) and the results of the test scenarios are shown and analyzed.

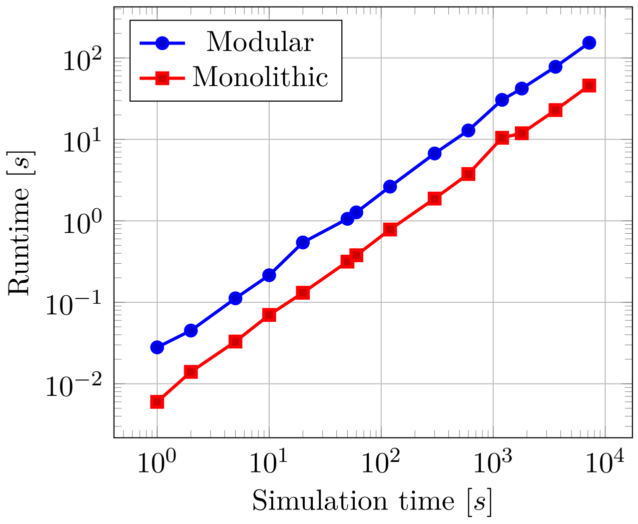

Figure 7.4: Runtime using different designs over different simulation time limits.

7.3.1 Runtime

The runtime results of the sequential simulations using both designs and given different simulation time limits are displayed in Figure 7.4. The double logarithmic axis shows two linear characteristics for modular and monolithic design. These nearly parallel linear characteristics show a constant factor between the runtimes using the two different designs. The average ratio of runtime using the modular design over the runtime using the monolithic design is 3.393.

The runtime using different designs with a varying number of *EventManagers* is shown in Figure 7.5. As expected, there is no obvious dependency of the resulting runtime on the number of included *EventManagers*. Due to the fixed simulation time, the number of created event stays the same. The only difference introduced by the increasing number of *EventManagers* is the destination of the event type data. The required time for transporting the messages and processing the data is untouched. The increasing number affects the memory usage of the simulation, which is not analyzed within this design test. The offset, visible in Figure 7.5, between the different designs represents the ratio of achieved performance regarding the runtime. The average of this ratio is 3.342.

The runtime using different designs with a varying polling interval by the *HistoryManager* is shown in Figure 7.6. These results are further displayed in a double logarithmic plot for easier recognition of both plots following

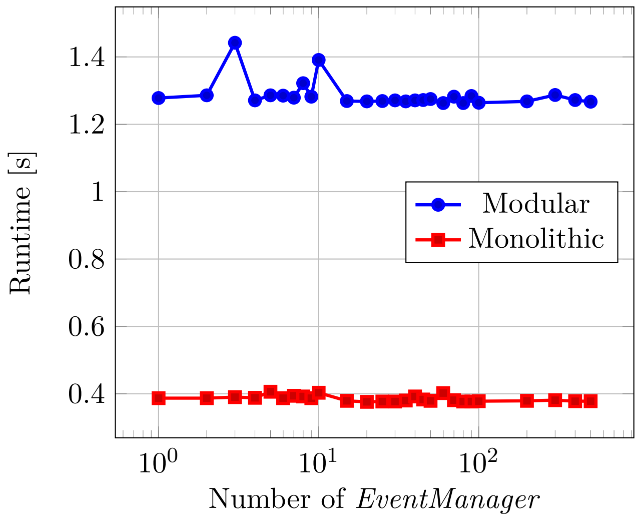

Figure 7.5: Runtime using different designs over variing number of *Event-Manager*.

a potential characteristic, which is visible as linear plots. As expected the required runtime decreases with increasing polling interval. This behavior can be explained by the sinking frequency of polling operations and thus a decreasing communication. The offset between the two plots is shown in Figure 7.6, due to the double logarithmic display, representing the ratio between the runtimes using different designs. The average ratio of runtime using the modular design over the runtime using the monolithic design is 7.809.

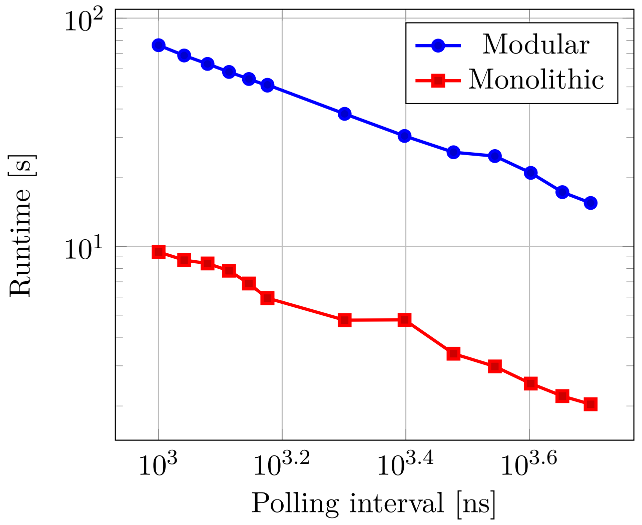

Figure 7.6: Runtime using different designs over varying polling interval by the *HistoryManager*.

The runtime using different designs with a varying generation interval by the *Generator* is shown in Figure 7.7. Similar to the previous Figure these results are also shown in a double logarithmic plot showing the potential characteristic of sinking required runtime with increasing generation interval. Again the noticeable offset between the plots is due to a ratio of resulting runtimes. The average ratio of runtime using the modular design over the runtime using the monolithic design is 1.588.

Analyzing the resulting outputs the ratio of generated messages (events) regarding the different designs can be determined. The average ratio of number of created events using the modular design over the number of created events using the monolithic design is 2.149. This ratio will be used as correction value for the next measurement method analyzing the created events within a fixed cpu time.

7.3.2 Created Events

The results of testing the simulations and analyzing the number of created events within a given cpu time are displayed in Figure 7.8. Similar to the previous section, a double logarithmic display was used for analyzing the characteristic of the plots. The courses show potential characteristics and are nearly linear and parallel, which leads to the assumption of a constant ratio between the used designs. The average ratio of the number of created events using the modular design over the number of created events using

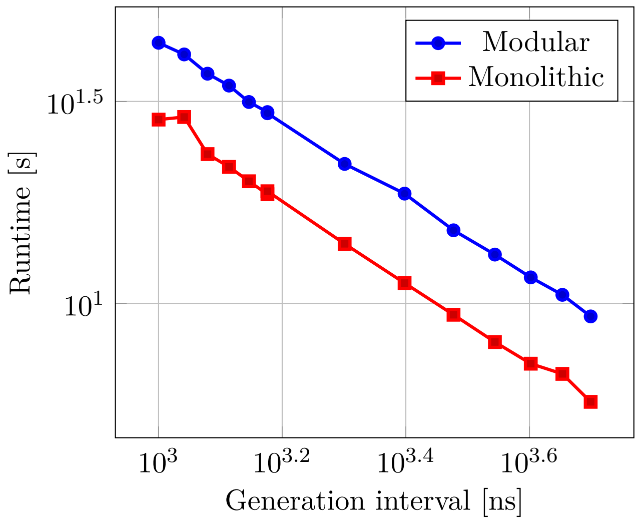

Figure 7.7: Runtime using different designs over varying generation interval by the *Generator*.

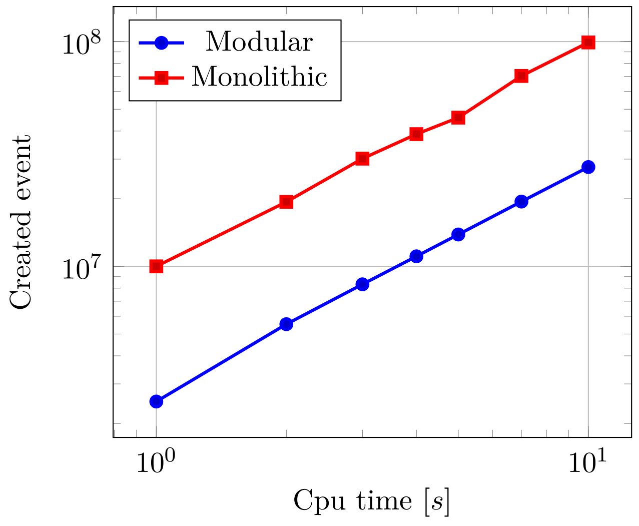

Figure 7.8: Created events for different designs over different cpu time limits.

the monolithic design is 0.284. For comparison to the previous and following test method, the reciprocal value is calculated and equals 3.521.

The number of created events using different designs and a varying number of *EventManagers* is shown in Figure 7.9. As expected and similar to the results of the runtime with a varying number of *EventManagers* there

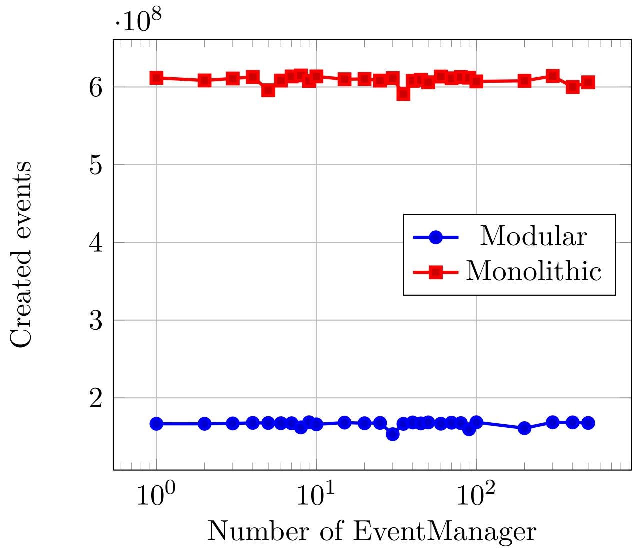

Figure 7.9: Created events using different designs over varying number of *EventManager*.

is no noticeable dependency of the created events on the number of *Event-Managers*. This presumption is due to the change of number of instantiated *EventManagers*, whereby only the destination of the transmitted events is affected and not the number of events. Using the double logarithmic scale the ratio between the different designs is visible as offset, which is shown in Figure 7.9. This ratio is defined by the number of created events using a modular design over the number of created events using a monolithic design. The average of this ratio is 0.273. In comparison between the previous and following test method the reciprocal value is calculated which equals 3.663.

The number of created events using different designs and a varying polling interval by the *HistoryManager* is shown in Figure 7.10. The expected behavior would be a sinking number of created events with increased polling interval by the *HistoryManager*. This behavior should be introduced by the decreased frequency of polling messages or self messages for timer implementations and therefore an increased number of created messages. Analyzing the characteristics shown in 7.10 displays this behavior only using the monolithic design. Due to this mismatch of both behaviors, no assumption about the dependency is made. The offset between the plots using different designs is nevertheless indicating a ratio representing the performance difference of the used designs. This ratio is defined by the number of created events using a modular design in contrast to those using a monolithic design. For this test the average of this ratio results in 0.170. In comparison to the previous and following test method the reciprocal value is calculated and

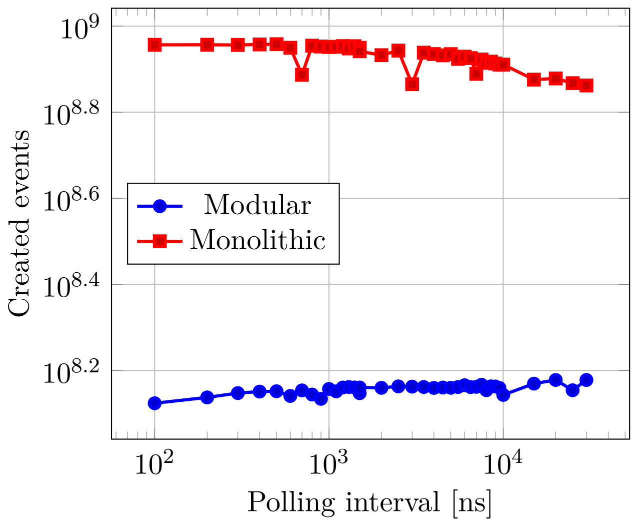

Figure 7.10: Created events using different designs over varying polling interval by the *HistoryManager*.

equals 5.882.

The number of created events using different designs and a varying generation interval by the *Generator* is shown in Figure 7.11. Similar to the polling interval of *HistoryManager* shown in 7.10 number of created events should be sinking with increased generation interval. In this scenario only the usage of the modular design shows this characteristic and thus no assumption about the dependency is made. The offset is related to the ratio between the designs and the average ratio is 0.414. In comparison to the previous and following test method the reciprocal value is calculated and equals 2.415.

7.3.3 Real-time

The real-time results of the sequential simulations using both designs and given different simulation time limits are displayed in Figure 7.12. The simulation time limit defines the runtime of a real-time simulation, but does not affect the performance, i.e. the frequency and complexity of calculations. Therefore as expected the result shows no recognizable dependency of the achievable generation interval to the simulation time. Although an offset between the different designs is visible. This offset represents the ratio of achievable generation interval using a modular design in contrast to a monolithic design. The average of this ratio is 1.717.

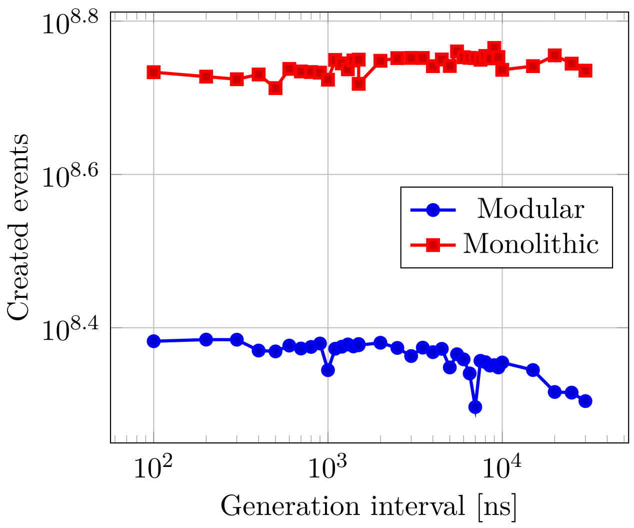

Figure 7.11: Created events using different designs over varying generation interval by the *Generator*.

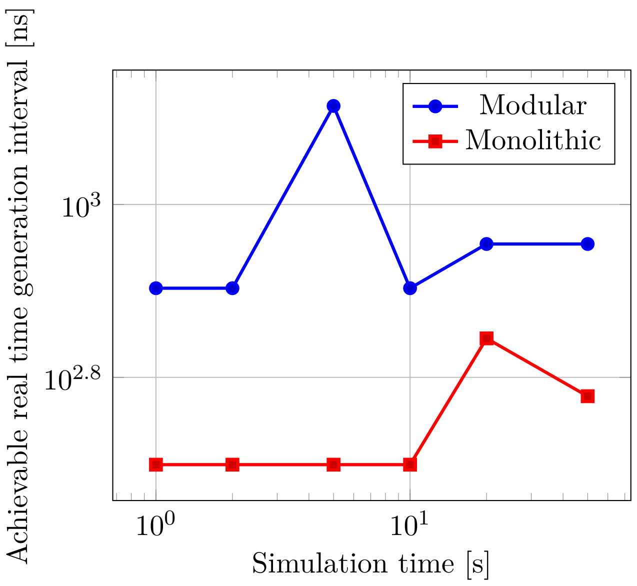

Figure 7.12: Real-time results for different designs over different simulation time limits.

The real-time results for a varying number of *EventManagers* are shown in Figure 7.13. The resulting characteristics display a wide distribution and therefore do not lead to a clear assumption about a dependency on the number of *EventManagers*. The impact of the different designs is still noticeable.

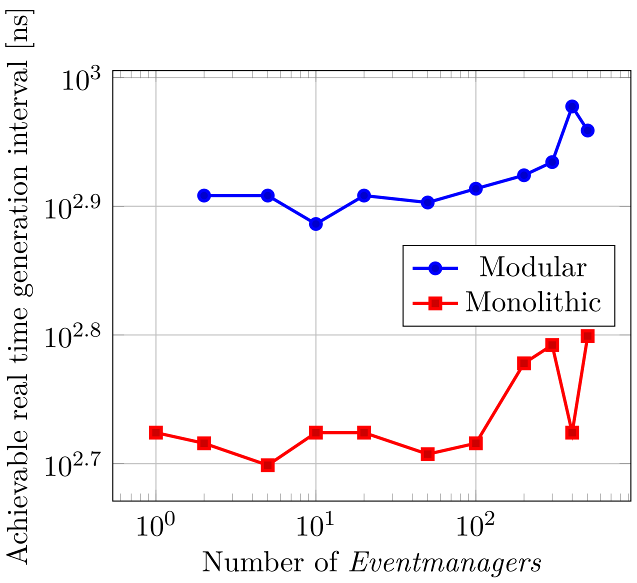

Figure 7.13: Real-time results for different designs over a varying number of *EventManagers*.

The average ratio of the achievable generation interval using different designs is 1.533.

The real-time results for a varying polling interval of *HistoryManager* is shown in Figure 7.14. Similar to the previous results there is no noticeable dependency of the achievable generation interval on the used polling interval. The performance difference is again shown in the ratio of the achieved intervals. The average of this ratio is 1.525.

The measurement method for analyzing the real-time behavior includes a parameter study of the generation interval of the *Generator*. Therefore the scenario with varying generation intervals cannot be analyzed using the real-time method.

7.3.4 Conclusion of sequential design tests

The analyzed methods were also executed on two additional host machines:

Workstation including an Intel® Core™ i3-2100 CPU @ 3.10GHz with 4GB *RAM* running Windows 7 Enterprise.

Build Server including an Intel® Core™ i7-4770 CPU @ 3.40GHz with 16GB running Linux Mint 17.

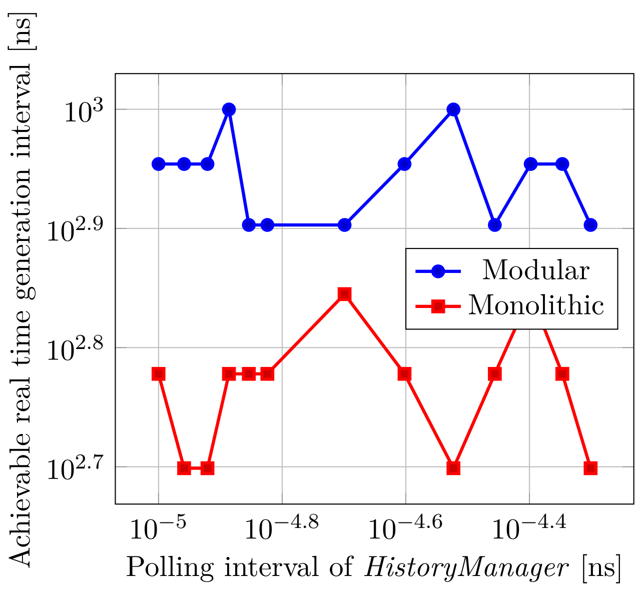

Figure 7.14: Real-time results for different designs over a varying polling interval of *HistoryManager*.

These further measurements amount to similar results and are shown in the appendix chapter B.

Analyzing the results displayed in section 7.3.1, 7.2.2 and 7.2.3 the following conclusions can be drawn:

- Using a modular design results in an increased number of created events. The average of the number of created events using the modular design in contrast to the monolithic design is 0.285. In comparison to the other ratios, the reciprocal value is calculated and equals 3.506.
- The increased number of events and the included overhead result in a decreased performance noticeable in each of the three used measurement methods.
- The average performance ratio of the required runtime using the modular design in contrast to the monolithic design is 4.033.
- The average performance ratio of achieved real-time generation interval using the modular design in contrast to the monolithic design is 1.592.
- The resulting average ratio of the combined performance is 3.044.

The overall average ratio of 3.044 represents the improved performance of the monolithic design over the modular design, which can be applied to a decreased required runtime, decreased achieved real-time interval or an increased number of created events.

By using the sequential simulation, the resulting recommendation for achieving the best performance e.g. for real-time simulations is using a monolithic design.

7.4 Parallel Simulation

For communication and synchronization, as described in section 6.3.1 and 6.3.2, the *MPI* system openMPI and the *Null Message Algorithm* were used.

An explicit synchronization of the different partitions is necessary due to multiple modules which use *self-messages* as timers. The host machine used for developing and executing the simulation provides a 4th Gen Intel® Core™ i7-4500U (1.8 GHz 200 MHz 4MB). This processor is a dual core CPU supporting hyper threading and therefore provides four logical processors distributed on two physical cores. The example network includes two autonomous modules which schedule their behavior with *self-messages*. This number of autonomous modules leads to the conclusion that two parallel partitions are most appropriate for parallel simulation.

As described in chapter 6, running a simulation distributed on parallel devices requires the mapping of the simulated modules to different parallel partitions. The used mapping assigns the *Generator* to the partition zero and all modules within the *Sink* to the partition one. This mapping is applicable for both tested designs.

7.4.1 Runtime

The measured runtime of each partition is accumulated for the comparison of different simulation runs. The resulting runtimes over varying simulation times are shown in Figure 7.15. The accumulated runtime shows, a rather constant characteristic in contrast to the potential characteristic of the sequential results. This characteristic combined with the highly increased runtime compared to the sequential simulation leads to the conclusion that the impact of the synchronization is hiding the dependency on the simulated time range. This conclusion is confirmed by the average ratio of resulting runtime using different designs of 1.074.

The resulting runtimes over a varying number of *EventManagers* are shown in Figure 7.16. In comparison to the sequential results, the characteristic of the runtime over the number of *EventManagers* remains constant. The average ratio of the achievable generation interval using different designs is 1.237.

The resulting runtimes over a varying polling interval by the *HistoryManager* are shown in Figure 7.17. Similar to the sequential simulation, the results of the runtime over the interval by the *HistoryMan-*

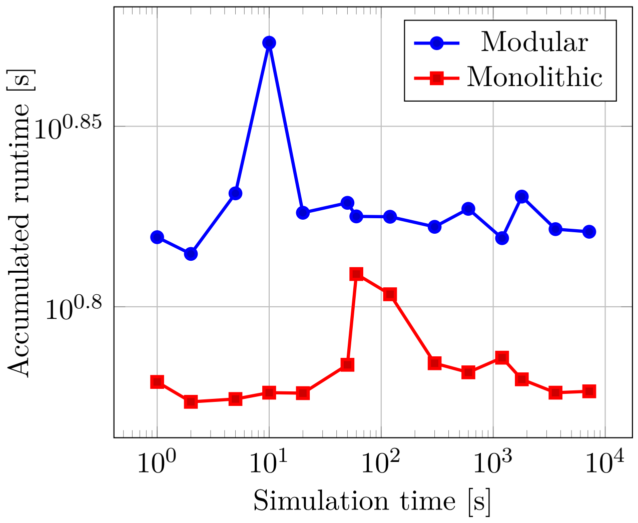

Figure 7.15: Accumulated runtime over varying simulation time limits.

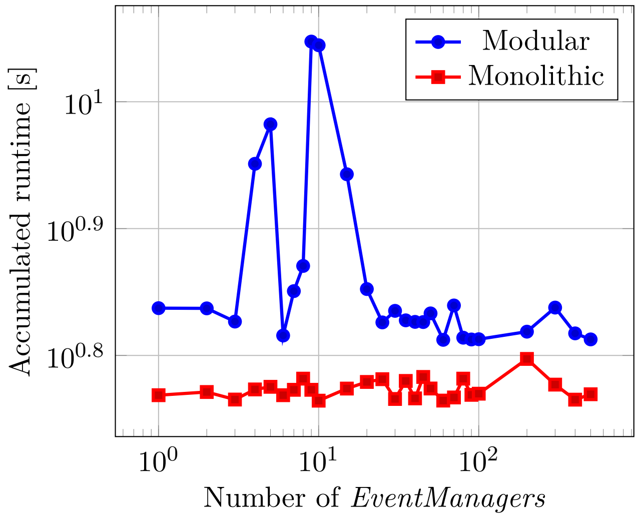

Figure 7.16: Accumulated runtime over varying number of *EventManagers*.

ager shows a potential characteristic with sinking runtime for increasing polling interval. The average ratio of the achievable generation interval using different designs is 4.916.

The resulting runtimes over a varying generation interval by the *Generator* are shown in Figure 7.17. Similar to the previous measurement, the results of the runtime over the interval by the *Generator* shows a potential characteristic with sinking runtime for increasing polling interval. The aver-

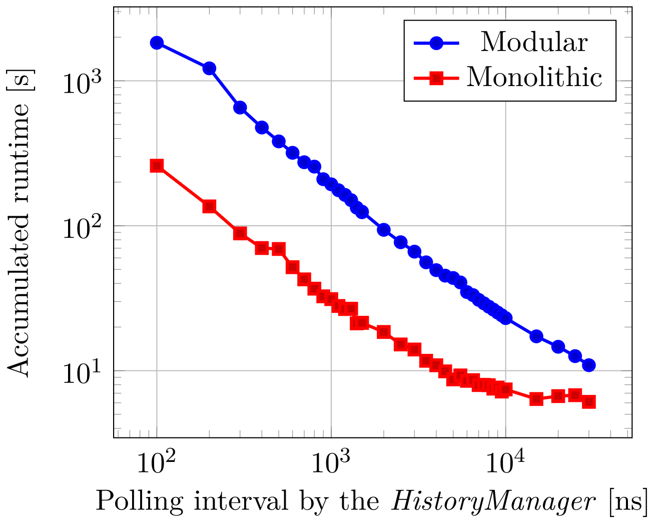

Figure 7.17: Accumulated runtime over varying polling interval by the *HistoryManager*.

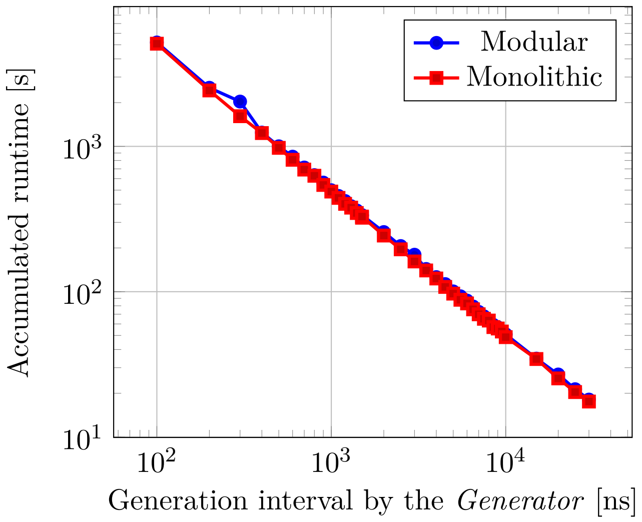

Figure 7.18: Accumulated runtime over varying generation interval by the *Generator*.

age ratio of the achievable generation interval using different designs is 1.043.

7.4.2 Conclusion of Parallel Tests

In view to the results shown in section 7.4.1, the following conclusions can be drawn:

- The performance of a parallel simulation depends strongly on the simulated model and its partitioning capabilities.
- The used synchronization highly affects the achievable performance.
- The average performance ratio calculated using the runtime results is 2.067. This average is strongly influenced by the polling interval measurement. Analyzing the other ratios and comparing them to the according sequential simulation results strengthens the assumption of high synchronization overhead.

The analyzed parallel simulation still shows a performance advantage using the monolithic over the modular design. The changed characteristics, ratios and availability compared to the sequential simulation show a decrease of the simulation performance. This can be caused by the lacking opportunities for parallelization of the example network and its rather small simulation effort.

The usage of parallel simulation could be established by implementing a custom synchronization enclosing the behavior of a real-time scheduler, which can lead to a performance improvement. For such a development the simulated system must be analyzed carefully resulting in the determination of the necessary behavior of the custom synchronization.

7.5 Conclusion

Considering the results of section 7.3 and 7.4, the modular design tends to result in an increased number of messages which causes an overhead and leads to a debased performance. Analyzing the simulation of the chosen example network, the parallel simulation was not able to achieve an improved performance in comparison to the sequential simulation and showed limited results due to increased uncertainty. The capabilities of parallel simulation, especially for the usage in the fields of real-time simulation, emulation and *HiL* must be analyzed individually for each simulated model. Based on the results simulating the defined example network the sequential simulation using a monolithic design is recommended for optimal performance and best usage for real-time simulation.

This design decision will be used for the first implementation of the openPOWERLINK simulation described in chapter 9. For developing a simulation design the structure and composition of the simulated openPOWERLINK stack must be analyzed. Information about the functionality of openPOWERLINK, its structure and hardware dependency is provided in the following chapter.

Chapter 8

openPOWERLINK

openPOWERLINK is an Open Source implementation of the POWERLINK protocol. The implementation contains the openPOWERLINK stack and different demo applications for various platforms. It is designed for a simple introduction into POWERLINK and a public available implementation. This project allows manufacturers to easily integrate POWERLINK into their products. The Open Source implementation targets at an improved integration and development of new features which may be inspired by request of manufacturers and the community. The stack is distributed under the *BSD* license and is available at Github [22] and Sourceforge [24].

The following section discusses the properties, structure and functionalities of POWERLINK.

8.1 POWERLINK

POWERLINK is an industrial real-time communication protocol based on the IEEE 802.3 standard (Fast Ethernet [10]). POWERLINK was developed by the members of the Ethernet POWERLINK Standardization Group (*EPSG*). The *EPSG* consists of different companies located in the fields of real-time communications, automation and field bus communication. [8]

One fundamental principle of POWERLINK is the usage of a common communication system for various applications. I.e POWERLINK supports the real-time transmission of data for time critical applications and a simultaneous transmission of less time critical asynchronous information.

To grant a deterministic communication, which is essential for real-time communication, collisions within a POWERLINK network are prevented by the definition of the POWERLINK cycle. Thus the features of collision detection and transmission retires of Ethernet Carrier Sense Multiple Access/Collision Detection (*CSMA/CD*) are not necessary for a POWERLINK network [10, section 4.2]. This collision prevention is done via strict slot based transmission. Each node within a POWERLINK network is only allowed to send within its assigned slot and furthermore each controlled node (*CN*) is only permitted to send a response to a request from the managing node (*MN*) [7, chapter 1]. The communication sequence of a standard POWERLINK network is shown in section 8.1.4.

The structure and components of a POWERLINK network are described in the following section.

8.1.1 Network Structure

A POWERLINK network consists of the following two different node types.

MN The managing node exist once within a normal POWERLINK network and controls the communication flow. The MN manages all registered network participants, provides a clock and defines the transmission cycle.

CN All other nodes within a normal POWERLINK network are controlled nodes and react according to the controls of the *MN*.

Within a POWERLINK network unique POWERLINK addresses (Node IDs) are assigned to each node. The address range from 1 to 239 is available for all *CNs* and can be assigned freely. The address 240 is fixed for the *MN*, each node assigned the Node ID 240 automatically performs as *MN*. When the execution of the *MN* role is not possible the assignment of the Node ID 240 is not permitted. [7, section 4.5]

A simple POWERLINK network consisting of an *MN* directly connected to one *CN* and two additional *CNs* via an Ethernet HUB is shown in Figure 8.1. POWERLINK is based on the standard IEEE 802.3 MAC layer and therefore the usage of default Ethernet hardware is possible. The different nodes can be arranged in various topologies. The integration of Ethernet HUBs into each node is recommended to allow for an easy setup of commonly used network structures, e.g. a star topology with lines of multiple nodes. [7, chapter 3]

8.1.2 Frame

The POWERLINK frame is embedded in the payload of an Ethernet 2 frame and is defined via the Ether type 0x88AB. Therefore the POWERLINK frame is preceded by the Ethernet 2 Header containing destination *MAC* Address, source *MAC* Address and Ether type. The payload of an Ethernet 2 Frame can reach up to a length of 1500 bytes succeeding with 4 bytes checksum. [10, section 3.2] [7, section 4.6.1]

In Figure 8.2, the structure of a POWERLINK frame is depicted. The POWERLINK header shown to the left contains the destination and source Node Id preceded by the message type. The length of the payload and content depends on the transmitted message and is thereby defined by the message type. [7, section 4.6.1.1]

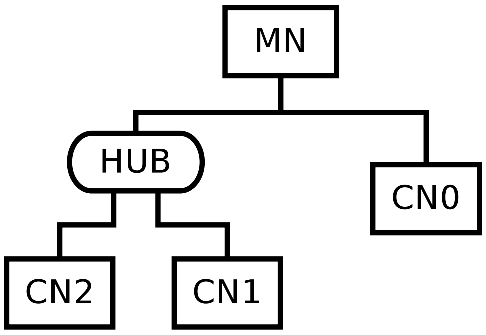

Figure 8.1: POWERLINK network consisting of an *MN* connected to one *CN* and an Ethernet HUB which is connected to two more *CNs*.

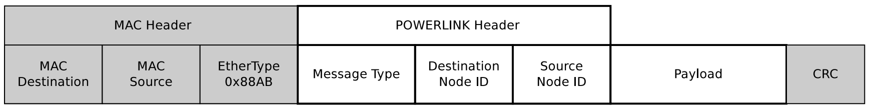

Figure 8.2: POWERLINK frame showing POWERLINK header and payload embedded in an Ethernet II frame.

Detailed information about the different structures of transmitted payloads depending on the message type can be found here [7, section 4.6.1.1.1].

8.1.3 Commands

As mentioned above, different POWERLINK messages are defined via the message type. The following commands are distinguished by the according message type.

SoC Start of cycle is sent by the *MN* as multicast and defines the start of the POWERLINK cycle and the isochronous phase.

PReq Poll request is sent by the *MN* to a specific *CN*, as unicast, transmitting data and requesting the transmission data from the *CN*.

PRes Poll response is sent by the *CN* as multicast as response to a *PReq* and contains data from the *CN*.

SoA Start of Asynchronous is sent by the *MN* as multicast and defines the end of the isochronous phase and the beginning of the asynchronous phase. Additionally, the *SoA* message contains the information about the assignment of the following asynchronous slot.

ASnd Asynchronous send is sent either by the *MN* or a *CN* as multicast and contains asynchronous data.

The transmission of an *ASnd* message is happening in the asynchronous phase, as described in the next section. *Asnd* messages can embody different services which are shown in the following listing. [7, section 4.6.1.1.6.1]

IdentResponse represents a response to a received *IdentRequest*.

StatusResponse represents a response to a received *StatusRequest*.

NMTRequest represents a response to a received *NMTRequestInvite* when an *NMT* request is pending at the local node.

NMTCommand represents a response to a received *NMTRequestInvite* when an *NMT* command is pending at the local node.

SDO represents a response to a received *UnspecifiedInvite* and signals an included *SDO* transmission.

Manufacturer specific represents manufacturer specific services and usages.

These transmissions are invoked by the preceding *SoA*, which includes a request service id. According to this request service id the scheduled node responds with the according response. The possible requested service ids are shown in the following listing. [7, section 4.6.1.1.5.1]

NoService indicates that the following asynchronous slot is unassigned.

IdentRequest represents an identification query and is used for checking the activity and accessibility of nodes within the POWERLINK network.

StatusRequest represents a query of information about a node and its status.

NMTRequestInvite represents the assignment message for a pending *NMTCommand* or *NMTRequest*.

Manufacturer specific represents manufacturer specific services and usages.

UnspecifiedInvite represents the assignment of an asynchronous slot for sending any kind of POWERLINK *ASnd* or legacy Ethernet frame.

The sequence of sent commands within a POWERLINK cycle is shown in the next section.

8.1.4 Communication Cycle

The POWERLINK communication cycle is separated in two different phases.

The isochronous phase is the first part of a POWERLINK cycle and is started by the *MN* sending an *SoC* message. Within this phase the *MN* is polling each *CN* with registered isochronous data. This polling is accomplished by sending a *PReq* message to each *CN*. This message includes information for the *CN* and also isochronous data which should be sent from the *MN* to the specific *CN*. After the reception of the *PReq* message, the *CN* sends a *PRes* message as multicast. This message includes all isochronous

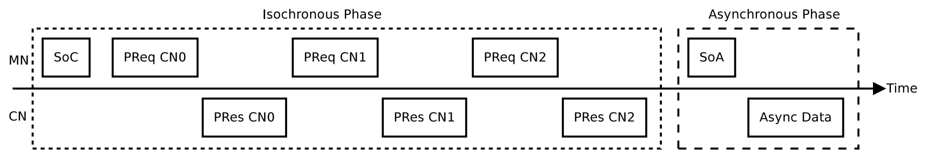

Figure 8.3: POWERLINK communication cylce containing Isochronous Phase with polling of three *CNs* and Asynchronous Phase.

data which is requested by the *MN* or any other *CN*. By multicasting this message each node which should receive a specific data set immediately receives the new values. [7, section 4.2.4.1.1]

When the last isochronous *CN* sent its *PRes* message, the isochronous phase is over. The end of this phase and the start of the asynchronous phase is marked by the sent *SoA* message by the *MN*. This message contains information about the node which is assigned to the current asynchronous slot. In each cycle, the *MN* assigns the asynchronous slot to either itself or to a *CN*. Additionally to the assigned node, a request service id is transmitted requesting a specific service as described in section 8.1.3. Is a *CN* demanding the assignment to an asynchronous slot this must be communicated to the *MN* either by the *PRes*, *IdentResponse* or *StatusResponse* message. The communicated number of pending asynchronous messages can additionally provide a priority for better scheduling. This scheduling is done by the *MN* according to the defined priorities. The messages sent within the asynchronous phase can either be a legacy Ethernet frame or an ASnd POWERLINK frame. [7, section 4.2.4.1.2]

In Figure 8.3, a simple POWERLINK communication cycle is shown. Above the horizontal time axis, all commands sent by the *MN* are depicted. Commands sent by different *CNs* are displayed below the axis. This communication cycle shows the isochronous phase marked by the dotted box to the left. This phase includes the starting command *SoC* and the sequential polling of each *CN*. This example matches the shown network in Figure 8.1 and contains three *CNs*. The different *CNs* immediately respond to the polling request.

At the right side in Figure 8.3 the asynchronous phase is marked by the dashed box. The asynchronous phase contains the *SoA* command and following asynchronous data transmission.

The transmitted data within the isochronous and the asynchronous phase and the underlying object model are described in the following section.

8.1.5 Data Transmission

The transmitted data objects are separated in two different groups.

The process data objects (*PDO*) represent time critical data which is transmitted every cycle. These transmissions happen during the isochronous phase in the POWERLINK cycle and are enclosed in the *PReq* and *PRes* messages.

When the data of a *PDO* is sent to other nodes it represents a transmit process data object (*TPDO*). Similar to the storage of received data in a *PDO*, this represents a receive process data object (*RPDO*).

The implemented application defines the number of supported *TPDOs* and *RPDOs*, but those are limited to 256 for each type. [7, section 6.4]

The counterpart to the isochronous transmitted *PDOs* are the service data objects *SDOs*. *SDOs* provide access to an object within the object dictionary (*OD*) of any node within the POWERLINK network. This access follows the client-server principle, i.e. the *SDO* client requests the value of an object located inside of the *OD* of the server. After receiving the request, the *SDO* server replies with the according data set. The transmission of these data and the according commands are located within the asynchronous phase of the POWERLINK cycle. This transmission can take place using an *ASnd* message or an User Datagram Protocol (*UDP*) connection. The granting of sending an asynchronous command is done by the *MN*.

Different functions for accessing the *OD* of the *SDO* server are provided. More information about the concrete functions for reading and writing a *SDO* can be found in [7, section 6.3.2.4.2];

The mentioned *OD* contains the definitions of all accessible objects within a POWERLINK network and is described in the next section.

8.1.6 Object Dictionary

The *OD* represents a collection of all defined objects which are transmitted via *PDO* or *SDO*, their types and additional communication objects. Within the *OD*, every object is structured in the same way and can represent either a variable, an array or a record of multiple variables. An object of type array or record contains multiple sub objects which must be of type variable. Within the *OD*, different regions contain definitions for different types and various profiles. [7, section 2.2.2]

Each object defined in the *OD* consists of the following associated values.

Index represents an identifier of the object within the *OD*.

Name is the name of the object.

Object type represents the type of the object (variable, array, record).

Data type defines the type of the contained data.

Value range defines an allowed value range for this object.

Access defines the allowed access functions for this object.

PDO mapping defines the available mapping possibilities to a *PDO*.

Default value defines the value of an uninitialized object.

Actual value represents the current value of the object.

More information about the different associated values and further information on the definitions of the *Access* and *PDO mapping* field can be found in [26].

This structure of the *OD* matches the structure of *CANopen* [9], that allows the usage of *CANopen* as application layer.

The openPOWERLINK stack implements the above described functionalities of POWERLINK. Different demo applications, drivers and the stack implementation are contained in the openPOWERLINK stack distribution. An introduction in the structure of the stack is given in the following section.

8.2 Structure of the openPOWERLINK Stack

The openPOWERLINK stack distribution package is structured in different main directories, containing demo applications, documentation, driver implementation, stack sources and many more.

For this thesis, the most important parts are the stack sources and demo applications. Furthermore, an additional simulation directory will be added and its content will be discussed in section 9.1.2.

The *apps* folder contains different demo application for embedded targets, Linux and Windows operating systems. These demo applications contain exemplary implementations of *MNs* and *CNs*. These application are analyzed for the development of demo applications within OMNeT++, as described in section 9.3.

The stack folder contains the following structure. [23, Directory Structure]

build represents the build directory for output files of the build process.

cmake contains configuration files for the build tool chain.

include/oplk represents the main include directories for all applications using the openPOWERLINK stack.

include/common represents a common internal include directory.

include/kernel represents an internal include directory for kernel modules.

include/target represents an internal include directory with target specific files.

include/user represents an internal include directory for user modules.

lib represents the installation directory for the openPOWERLINK libraries.

proj represents the directory containing different library projects.

src/arch represents a source directory with architecture specific functions.

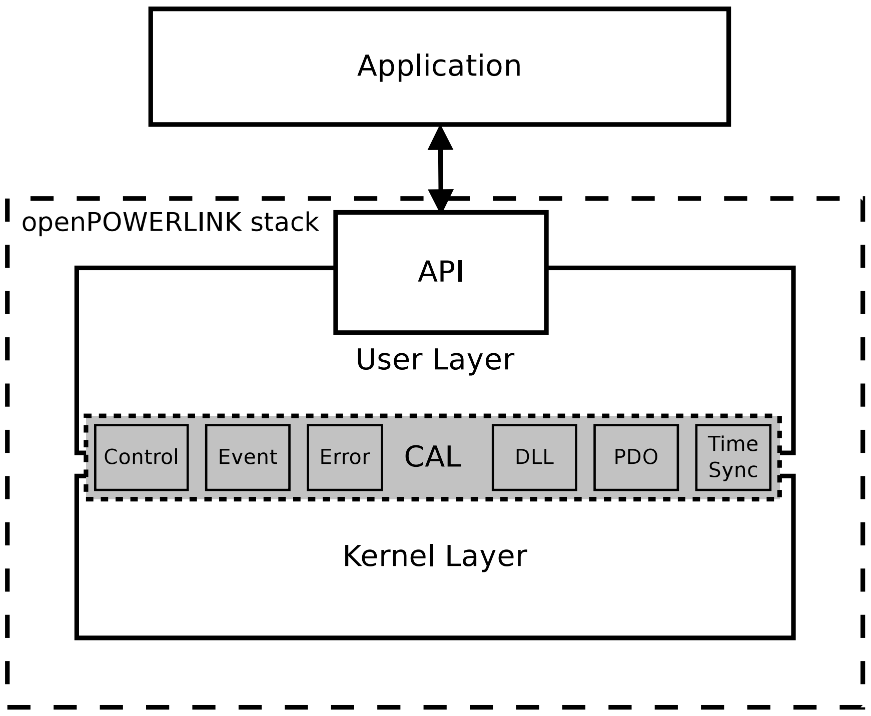

Figure 8.4: Software architecture of the openPOWERLINK stack.

src/common represents a source directory with sources for user and kernel layer.

src/user represents a source directory with sources for the user layer.

src/kernel represents a source directory with sources for the kernel layer.

This structure represents the architecture of the openPOWERLINK stack, which is explained in the following section.

8.3 Architecture

The openPOWERLINK stack is generally separated in kernel and user layer. This separation was introduced with the version 2.X of the openPOWERLINK stack and detaches the higher level user functionalities from the lower level kernel functions including time critical behavior and hardware drivers. This separation of high level functionalities and time critical procedures allows a real-time communication independent from the running application and the executed user functions.

The separation of kernel and user layer using the communication abstraction layer (*CAL*) is shown in Figure 8.4.

8.3.1 Kernel Layer

The kernel layer of the openPOWERLINK stack contains the following modules. [23, openPOWERLINK Kernel Layer]

Control module manages shutdown and startup commands for the kernel stack.

Data link layer (*DLL*) implements the *DLL* state machine and handles the creating and processing of the POWERLINK cycle.

Error handler implements the POWERLINK error handler and manages the error counters.

Event handler provides functionalities for posting events to other kernel modules and to the user layer. This handler also processes events posted by the user layer.

Led module controls the POWERLINK status and error Leds.

***PDO* module** handles the *PDOs* in the openPOWERLINK kernel layer and communicates with the *DLL* module for the transmission of the *PDO*s.

Network management (*NMT* module) implements the *NMT* state machine and provides the general functions for network management.

Virtual Ethernet driver provides a network interface for sending Ethernet packets. These packets will be transmitted during the asynchronous phase.

High resolution timer module provides high resolution timer functionalities for timing the POWERLINK cycle.

Synchronization timer module implements a timer with lower resolution than the HresTimer for non time critical usage.

Timestamp module provides helper functions for handling of timestamps.

Time synchronization module implements the time synchronization to the POWERLINK network.

Running on an operating system, e.g. Linux, the kernel layer can be built and executed separately as a kernel module of the operating system. Such a communication mode is defined via the configurations of the different stack projects (proj folder). The configurations affect the communication abstraction layer *CAL* described in section 8.3.3.

8.3.2 User Layer

The user layer includes higher level functions and provides the application programming interface (*API*) for an application using the openPOWERLINK stack. The following modules are located within the user layer. [23, openPOWERLINK User Layer]

API represents the interface to the application and provides all functions for interactions with the openPOWERLINK stack.

Control module manages shutdown and startup commands for the user stack.

Error handler handles the synchronization of error counters located in the kernel layer with the according objects in the *OD*.

Event handler provides functionalities for posting events to other user modules and to the kernel layer. This handler also processes events posted by the kernel layer.

***PDO* module** handles exchanging of process image data via the process image *API*.

***SDO* modules** provide the *SDO* stack of openPOWERLINK including command layer, sequence layer and the transmission implementation using *ASnd* or the *UDP*.

Network management (*NMT* modules) implements various *NMT* functions of the user layer including functions for *MN* and *CN* and the handling of the *Ident*-, *Status*- and *Sync*-requests.

Configuration manager module implements the handling of the *CDC* file and the configuration of connected *CNs* using *SDO* transmission.

***OD* module** implements the POWERLINK *OD*.

***OD* abstraction layer** provides an abstraction layer for accessing different POWERLINK *ODs*.

Timer module implements timer functionalities used by the user stack modules.

Time synchronization module provides a configurable notification at a specific synchronization point for the application.

8.3.3 Communication Abstraction Layer

The user and kernel layer are separated by the communication abstraction layer (*CAL*). This layer is separated in different modules, each module within the kernel and user the layer is communicating with the according opposite module. The separation of the *CAL* and placement between the kernel and user layer is shown in Figure 8.4. The gray dotted box shows the *CAL* and all embedded modules using a *CAL* related implementation.

The following modules use *CAL* modules for communication between kernel and user layer.

- control module
- event handler
- error handler
- data link layer (*DLL*)
- process data objects (*PDO*)

- time sync

For the listed modules, two *CAL* modules are necessary within the kernel and the user layer. A specific used *CAL* implementation must be configured in both layers. Different implementations of the *CAL* modules allow for various configurations and compositions of the openPOWERLINK stack using different communication methods between kernel and user layer. [23, CAL]

8.3.4 Hardware Abstraction

The openPOWERLINK stack implements the support of different operating systems and hardware platforms by the definition of the module functions in a common header. The compiled implementation represents the specification for each platform.

The platform dependency of the openPOWERLINK stack and the platform dependent modules are analyzed in section 8.5.

The dependency of the platform regarding byte endianess is handled by the abstract memory interface described in the next section.

8.3.5 Abstract Memory Interface

The abstract memory interface (*AMI*) provides multiple simple functions for the correct handling of data fields affected by the byte endianess. Similar to the *CAL* and hardware abstraction, the implementations for different platforms are defined by compiling the according implementation. [23, AMI]

The openPOWERLINK stack provides multiple configurable implementation and communication variants. These are managed within the used tool chain described in the next section.

8.4 Configuration and Build

For configuration and building of the openPOWERLINK stack the build tool *CMake* is used. The main *CMake* file *CMakeLists.txt* checks the current *CMAKE_SYSTEM_NAME* variable and uses the configuration matching the targeted platform.

The global *CMake* file loads according to the *CMAKE_SYSTEM_NAME* and *CMAKE_SYSTEM_PROCESSOR* variable the correct system specific option file. These option files include configuration for enabling and disabling parts of the openPOWERLINK stack. For example, the generation of the *CN* or *MN* library can be en- or disabled. Further options are platform-specific and provide configuration possibilities for various features available for different implementations and on different platforms.

If the targeted platform requires a custom tool chain, this must be defined by a specific tool chain file.

The *CAL*, hardware abstraction and *AMI* use specific implementations of common header files. These sets of specific sources are defined within the common *CMake* file *stackfiles.cmake*. This file includes groups defining source files for each module and each implementation specialization. The usage of such specific groups is done in project specific *CMake* files composing various stack variants for specific platforms. [23, Building openPOWERLINK]

Different projects for various systems and platforms are located in the *proj* folder as described in the following section.

8.4.1 Library Projects

Located in the *proj* folder various projects for different systems are defined. These projects are grouped in the following systems.

generic contains projects for embedded targets without underlying operating systems.

linux contains projects for Linux operating systems.

windows contains projects for Windows operating systems.

Within each system-specific folder different projects are located. These different projects use different features and configurations of the openPOWERLINK stack. Within the different project directories the *CMakeLists.txt* file defines the used implementations and sources. The groups defined in the *stackfiles.cmake* file are used and combined within the project specific *CMake* file.

Additional configurations made by preprocessor macros are set in the *oplkcfg.h* header file within each project directory.

Executing the *CMake* build tool generates the according makefile. Using this makefile either all included or specific projects can be built and installed at a defined install directory. [23, Building Stack Libraries]

8.5 Platform dependency

The openPOWERLINK uses specific implementations which realize a common header file for supporting multiple platforms. As described in the sections 8.3.3, 8.3.4 and 8.3.5, this strategy is used for the *CAL*, hardware dependencies and *AMI*.

These three categories of modules represent the platform dependency of the openPOWERLINK stack. The information about these modules and their features are taken from the naming convention of platform-specific implementations, i.e. an appended suffix with the abbreviation for the specific implementation type. The following sections refer to those suffixes as implementation types. Detailed information about the functionality of each module can be found at the header comment of each implementation file describing the designated usage and the used functions.

In the following sections, each module and its specific implementations are analyzed and the platform dependent functionalities are discussed.

8.5.1 CAL

The *CAL* includes multiple modules providing configurable implementations using different platform specific functionalities. The following listings show the different available implementations of the *CAL* modules, which is followed by the description of the different implementations and their included functionalities.

control module

- direct
- hostif
- mem
- noosdual
- pcie
- ioctl
- winioctl

event handler

- linux
- linuxkernel
- linuxioctl
- linuxpcie
- nooscircbuf
- noosdual
- nooshostif
- win32
- winkernel
- winioctl
- winpcie

error handler

- hostif
- ioctl
- local
- noosdual
- posixshm
- winioctl

DLL

- circbuff
- ioctl
- winioctl

PDO

- triplebufshm
- hostif
- linuxkernel
- local
- noosdual
- posixshm
- winkernel
- linuxmmap
- linuxpcie

time sync

- bsdsem
- hostif
- ioctl
- local
- noosdual
- winioctl
- linuxkernel
- winkernel

The different implementations with specific functionalities are often used for multiple modules. These functionalities of the different implementations are described in the following listing defined by the postfix of the implemented modules.

direct These implementations use direct calls between kernel and user layer. This is supported in configurations when both layers are located at the same instance.

hostif/nooshostif These implementations use the host interface ip core instantiated on a FPGA.

noosdual These implementations target platforms without operating system but providing a dual processor and a shared memory between them. This shared memory is used for various communications.

nooscircbuf This implementations use the circularbuffer library targeting platforms without underlying operating system.

linux These implementations use the circularbuffer library with both layer located in the Linux user space.

linuxkernel These implementations represent the interface of the Linux kernel space module which includes the kernel layer and can be accessed via ioctl.

ioctl/linuxioctl These implementations represent the counterpart for *linuxkernel* and communicate with the kernel module via ioctl.

pcie/linuxpcie These implementations use ioctl for communication with the Linux Peripheral Component Interconnect express (*PCIe*) driver and furthermore for communication with an external device connected via *PCIe* and running the kernel layer.

linuxmmap These implementations provide a memory interface with a memory mapping between kernel and user layer separated in user application and kernel module.

win32 These implementations use the circularbuffer library with both layer located in the Windows user space.

winkernel These implementations represent the interface of the Windows kernel space module which includes the kernel layer and can be accessed via ioctl.

winioctl These implementations represent the counterpart for *winkernel* and communicate with the kernel module via ioctl.

winpcie These implementations use ioctl for communication with the Windows *PCIe* driver and furthermore for communication with an external device connected via *PCIe* and running the kernel layer.

local These implementations use global variables for data storage and requires kernel and user layer running in the same domain for sharing global variables.

posixshm These implementations use posix shared memory for communication between kernel and user layer.

triplebufshm These implementations use a triple buffer implementation for synchronous reading and writing of a shared memory. The shared memory implementation can be configured due to the usage of an abstraction given by a common header file.

bsdsem These implementations use BSD semaphores for synchronization of kernel and user layer.

8.5.2 Hardware Abstraction

As described in section 8.3.4, the hardware abstraction is done in a similar way as the *CAL* by using specific implementations of common header files. The following listing includes all hardware dependent modules, their location within the stack directory, a summary of the provided functionalities and their available implementation types.

target The *target* module is located in the *arch* folder and provides target specific functions controlling the defined IP address, default gateway, global interrupt, status/error leds and a sleep function. This module is separated in different folders containing implementations for the following specific platforms and systems.

- altera-c5socarm
- altera-nio2
- linux
- linuxkernel
- windows
- winkernel
- xilinx-microblaze
- xilinx-zynqarm

These folders can also include different implementations for specific functions. The different types are similar to the listed specific implementations of the *CAL* shown in 8.5.1.

circularbuffer The *circularbuffer* module is located in the *common* folder and provides an implementation for reading and writing variable sized data segments from and to a circular buffer. This module contains different implementations regarding the location of the used memory and the handling of memory access. These different implementations use system specific functionalities for locking and synchronizing memory access. Those implementations are used by the generic *circularbuffer* implementation using the *circbuf-arch* interface. The different implementation types of this interface are shown in the following listing.

- linuxkernel
- noos
- noosdual
- nooshostif
- posixshm
- win32
- winkernel

The used functionalities are similar to the described specific implementations of the *CAL* shown in 8.5.1.

memmap The *memmap* module is located in the *common* folder and provides implementations for the handling of memory mapping used in various modules, e.g. *CAL* modules. This module contains different implementations regarding the location of the used memory and the handling of memory access. These implementations use different functionalities providing a common interface for memory mapping. The available implementation types are shown in the following listing.

- linuxpcie
- noosdual
- nooshostif
- nooslocal
- null
- winioctl

The used functionalities are similar to the described specific implementations of the *CAL* shown in 8.5.1.

edrv The *edrv* module is located in the *kernel* folder and provides implementations of various Ethernet drivers. This driver provides functionalities for Ethernet communication and settings regarding the underlying Ethernet controller. The following implementation types of Ethernet drivers supporting different types of Ethernet controller are available.

- Realtek 8111
- Realtek 8139
- Intel 8255x
- Intel 82573
- Xilinx Zynq emacps
- Intel i210
- MUX Driver in VxWorks
- Windows NDIS intermediate driver
- openMAC IP-Core
- pcap Linux
- pcap Windows

The provided versions for the different Ethernet controller are implemented for the custom behavior and usage of different controllers and libraries.

hrestimer The *hrestimer* module is located in the *kernel* folder and provides implementations for various high resolution timer drivers. These timers are necessary for timing the POWERLINK cycle and for the synchronizing to the POWERLINK network. Depending on the implementation either a connection to a hardware or software component providing the required resolution is used. The following list shows the differently supported high resolution timers.

- Intel i120
- linuxkernel
- Windows NDIS timer
- openMAC IP-Core timer
- posix
- posix-clocknanosleep
- VxWorks
- Windows
- Xilinx Zynq ttc

Those different implementations use different components as timers.

veth The *veth* module is located in the *kernel* folder and provides implementations of different virtual Ethernet driver. This driver provides the functionality of receiving a non POWERLINK Ethernet frame and passing this to the user layer. The different implementations use different functionalities of underlying systems and platforms for synchronization and communication. The available implementations are shown in the following list.

- generic
- linuxkernel
- linuxpcie
- linuxuser
- ndisintermediate

The used functionalities are similar to the described specific implementations of the *CAL* shown in 8.5.1.

sdo The *sdo* module is located in the *user* folder and provides implementations for transmitting the *SDOs* during the asynchronous phase. This transmission can be done via an *ASnd* message or a legacy Ethernet frame, as described in section 8.1.5. Performing this transmission by using legacy Ethernet frames, the *UDP* is used for data transmission based on the Internet Protocol (*IP*). The implementation of the *UDP* and *IP* frame creation can be done in various ways due to the available *UDP* and *IP* stacks on operating systems. Therefore, this module offers different implementations for different platforms providing the usage of Linux and Windows functionalities. Additionally, the open library *socketwrapper* can be configured which allows the integration of a custom *UDP* stack. The following list shows the available implementations

- linux
- windows
- socketwrapper

The implementations for Linux and Windows use the existing functionalities for creation and handling of *UDP* sockets.

timer The *timer* module is located in the *user* folder and provides different implementations for timer functionalities used by the user layer. These timer functions do not require the high resolution and accuracy of the *hrestimer* module and can therefore be implemented in a simpler and more resource-efficient way. The different implementations include a generic implementation using the global system tick and specific implementations using features of underlying operating systems. These types of implementations are shown in the following list.

- generic
- linuxkernel
- linuxuser
- vxworks

If the target system does not provide such a functionality or the requirement for a high precision user timer is not given, the usage of the generic implementation is sufficient.

targetdefs The *targetdefs* folder is located within the *include/oplk* folder and contains different header files for specific targets. The following list shows the differently defined target systems.

- c5socarm
- linux
- microblaze
- nios2
- vxworks
- wince
- windows
- winkernel
- zynqarm

These header files define preprocessor macros for common system specific attributes and small functions regarding atomic operations and synchronization functions. The main header *targetsystem.h* is located in the folder *include/oplk* and includes the according system and platform specific target file.

8.5.3 AMI

The *AMI* (described in section 8.3.5) represents also a platform dependent module and therefore provides different implementations for little- and big-endian processors. Additionally, an optimized implementation for x86 processors is provided.

None of these implementations use platform specific functionalities, but the implementation varies for achieving the correct result when handling variables consisting of multiple bytes.

The platform dependencies of the openPOWERLINK stack shown in this section represents those modules which must be customized for porting the openPOWERLINK stack to a new platform or in a new environment. The porting of the openPOWERLINK stack in a simulation environment using OMNeT++ and the development of the according simulation is shown in the following chapter.

Chapter 9

Simulation of openPOWERLINK

The development of an OMNeT++ simulation including a POWERLINK network consisting of multiple nodes is achieved by porting the openPOWERLINK stack to the OMNeT++ environment. Therefore, the platform dependencies, as discussed in section 8.5, are analyzed. Using these dependencies, the simulation is developed.

The design measurement shown in chapter 7 resulted in the recommendation of a monolithic design, which achieves an improved performance. Based on this recommendation, the platform dependencies shown in section 8.5 are analyzed and different library projects are compared. The result for the most monolithic stack configuration is the demand for the following modules to be implemented in the simulation stub.

- edrv
- hrestimer
- target
- sdoudp

Additionally to the listed mandatory modules, the *trace* module is implemented for forwarding additional trace informations to the simulation environment.

For showing the implementations and giving examples of the strategies and implementations in the following sections, the Ethernet driver module *edrv* was chosen.

The porting was designed with the intention to separate the simulation specific implementations including changes within the openPOWERLINK stack and the OMNeT++ implementations providing the simulation environment. This is achieved by the introduction of a simulation stub in the openPOWERLINK stack.

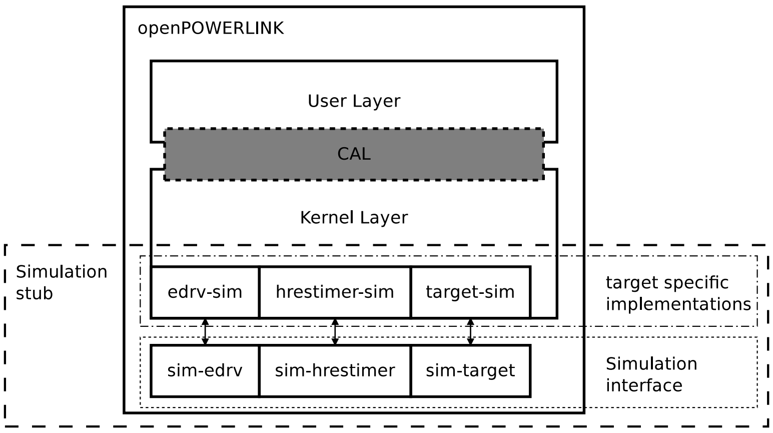

Figure 9.1: Example hierarchy of the simulation stub (dashed, left) and the included target-specific implementation (dash-dottet, right) and simulation interface (dashed, right).

9.1 Simulation Stub

The simulation stub should provide the same functions and signatures for the usage within the openPOWERLINK stack as the existing target specific implementations, but in addition should forward according function calls to the external simulation environment.

For minimizing the necessary changes in the openPOWERLINK stack the simulation stub is separated into two components, the specific implementations for the *sim* target and the simulation interface. An example hierarchy of the simulation stub is shown in Figure 9.1. The enclosed target specific implementations and the simulation interface are explained in the following sections.

9.1.1 Target-specific Implementation

Similar to the specific implementations for various modules in the openPOWERLINK stack (as described in section 8.3), the implementations for the *sim* target were added as specific implementations of the given common header files. As shown in figure 9.1 the target specific implementations are embedded in the kernel layer of the openPOWERLINK stack.

The appendix section C.1.1 shows the implementation of the *edrv_sendTxBuffer* function from the *edrv-sim* implementation.

Within the target-specific implementations of different modules only function calls to the simulation interface and small parameter conversions are implemented. The simulation interface is located in a newly introduced folder named *sim*. The content of this folder and the targeted purpose are outlined in the next section.

9.1.2 Simulation Interface

The *sim* folder contains two main folders, separating the source and include files. Each module within the openPOWERLINK stack, which should be connected to the simulation environment, has its specific simulation module within the simulation interface. The naming of the different simulation interface modules shows the common prefix *sim* separated with a hyphen from the implemented module name. The connection between the target specific implementation and the simulation interface is sketched in Figure 9.1.

The simulation interface contains all functions which are required by the ported stack module. Additionally, each simulation interface provides a *set<ModuleName>Functions* function. For all functions which should be connected to the simulation, an according function pointer is defined as typedef and all functions are grouped in a structure named *t<ModuleName>Functions*. For a common include file containing all types required for the simulation interface the typedefs for each function pointer and the structures are defined within the header file *sim.h*. This structure is passed to the *set<ModuleName>Functions* function and its content is checked for validity. Only when all of them are valid, the structure is saved in the static instance and a flag is set for valid initialization.

When a function of the simulation interface is called by the stack the stored static instance is checked and the according function is called. Therefore the external functions are called and the connection from the stack to the simulation environment is established.

For configuring the simulation interface from the simulation environment, the accessible function *set<ModuleName>Functions* is declared as an exported function for shared libraries. This is done via the preprocessor macro *OPLKDLLEXPORT* defined by the openPOWERLINK stack. This macro is defined within the target-specific header files located in *stack/include/oplk/targetdefs* as described in section 8.5.2 and marks the exported functions for the shared library.

Appendix section C.1 shows the definition and implementation of the simulation interface module *edrv-sim* regarding the initialization and the *sendTxBuffer* function as example. As shown in the appendix, an additional parameter was added to each function. This is necessary for supporting the simulation of multiple openPOWERLINK stack instances within a single simulation. The used strategy is described in section 9.2.3.

For the opposite direction of calling a stack function from the simulation environment three cases are distinguished.

1. The desired function is already an exported function, e.g. a part of the public *API*, it is resolved and called directly from the simulation environment.
2. The desired function is not exported, an according function in the simulation interface module must be provided. Within the implemen-

tation of such a function, the according stack function can be called directly.

3. The desired function is already an exported function, but specific changes to the passed parameter, e.g. function pointer, is necessary. In this case the function is defined additionally within the simulation interface providing a wrapper functionality.

In all cases, no target-specific implementation is required, because no implementation is replaced by the simulation, only additional functions are made accessible.

The connection of the openPOWERLINK stack to the simulation environment is designed independently from OMNeT++ and does not require any OMNeT++ functionalities. This independence was established for supporting various simulation environments and systems. The simulation interface can be used by every application and simulation environment which is capable of handling native shared libraries and function pointers.

For this thesis, the simulation environment OMNeT++ was chosen and the modified openPOWERLINK stack is embedded in an OMNeT++ simulation. The following sections show the implementation of the simulation and its different components.

9.2 Simulated stack

The simulated stack consists of the five implemented modules *edrv*, *hrestimer*, *target*, *sdoudp* and *trace*. These modules will also be implemented as simple modules representing the simulated structure and functional units.

The result of the successfully build process of the modified openPOWERLINK stack is a shared library exporting various functions, including the exported *API* functions and new functions for interacting with the simulation interface. The structure of the developed simulation and the enclosed components are described in the following section.

9.2.1 Simulation structure

Within the openPOWERLINK simulation the fundamental folder structure is taken from OMNeT++ recommendations and separates the simulation configuration (*simulations* folder) from the implemented model (*src* folder). For better representation of custom openPOWERLINK nodes, the *images* folder contains customized icons with an embedded openPOWERLINK logo.

The implemented model within the *src* folder is separated in different folders for the *MN*, the *CN* and a generic node implementation. These nodes and the implemented hierarchy are described in section 9.3.

Approaching the counterpart to the above-described simulation interface, the generic node contains a *stack/interface* folder enclosing the imple-

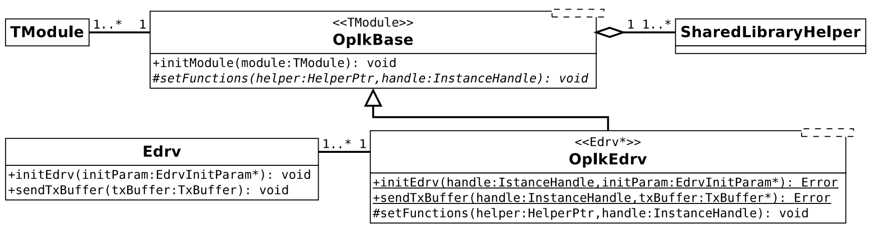
TModule
1..*
1
<<TModule>>
OplkBase
+initModule(module:TModule): void
#setFunctions(helper:HelperPtr,handle:InstanceHandle): void
1
1..*
SharedLibraryHelper
Edrv
+initEdrv(initParam:EdrvInitParam*): void
+sendTxBuffer(txBuffer:TxBuffer): void
1..*
1
<<Edrv*>>
OplkEdrv
+initEdrv(handle:IstanceHandle,initParam:EdrvInitParam*): Error
+sendTxBuffer(handle:InstanceHandle,txBuffer:TxBuffer*): Error
#setFunctions(helper:HelperPtr,handle:InstanceHandle): void

Figure 9.2: Class diagram showing the inheritance used for the stack interface.

mentation of the interface to the simulated openPOWERLINK stack. The functionalities and the underlying strategy of this interface implementation are described in the following section.

9.2.2 Interface Implementation

The implementation of the interface within the simulation is based on the usage of the following two classes, providing the basic functionalities for handling the connection to the simulation interface.

First, *OplkBase* represents a base class for every interface module connected to the simulated openPOWERLINK stack. It is implemented as a template class taking a template type for a module. This module is grouped together with an instance of the second class *SharedLibraryHelper*. The class diagram with *OplkBase*, *SharedLibraryHelper* and as example the implemented classes *Edrv* and *OplkEdrv* are shown in Figure 9.2.

Second, the class *SharedLibraryHelper* is implemented independently from the OMNeT++ framework and provides methods for the handling of a shared library. The internal implementations are defined by preprocessor macros and result in according implementations for Linux and Windows. Basically, the *SharedLibraryHelper* handles the loading of a defined shared library object and resolving of defined functions. The shared library is opened during creation of a *SharedLibraryHelper* and is closed during destruction. Thereby, this class follows the design principle of Resource Acquisition Is Initialization (RAII) [27]. This ensures the correct closing of all opened shared libraries during shutdown and prevent possible resource leaks. The resolved functions are returned as std::function objects with the requested types. The usage of functional objects instead of simple function pointer allows for more dynamic handling within the object oriented environment.

The class *OplkBase* provides an *initModule* method taking an instance of the template type as a parameter (as shown in Figure 9.2). This function creates a new instance of *SharedLibraryHelper* with the internally saved library name and stores the helper object together with the given module in an internal container.

After this creation, the helper object and the index of the currently added instances within the internal container are passed to the method *setFunctions*. This method is defined pure virtual and demands the implementation by a derived class.

All interface classes derive from *OplkBase* and contain the functionality of holding the according *SharedLibraryHelper* with an instance of the defined module type These derived classes are named according to the implemented module with *Oplk* as prefix and located within the *interface* folder and the *interface* C++ namespace. The template parameter is mostly defined a pointer to the according OMNeT++ module which is named simply by the implemented module, e.g. *Edrv* represents the OMNeT++ module for the Ethernet driver module. The described inheritance and connection are shown in Figure 9.2.

This connection of *SharedLibraryHelper* and the module instance is demanded by the requirement for multiple simulated instances of the openPOWERLINK stack within a single simulation. This requirement, its consequences, the solution and the implementation are described in the next section.

9.2.3 Multiple Instances

The openPOWERLINK stack is designed as pure ANSI C implementation structured in multiple modules. The designated usage of the openPOWERLINK stack is the execution on devices representing a single node. Because of this designated usage, no demand for multiple instances is given and the information about various states, buffer and all other data is stored within the openPOWERLINK stack as static variables. These variables exist once in the compiled application or library. Therefore, the simulation of multiple instances within a single application by simply linking to a shared or static library is not possible.

The solution for this problem is loading the same shared library multiple times in a manner that the loaded library exists multiple times within the memory. Linux supports loading of a single shared library multiple times into the memory and therefore creating multiple instances. On a negative node, Windows does not provide such a functionality. OMNeT++ supports Windows and Linux in the same way and for achieving a simulation which is usable in the identical way on either Linux or Windows another strategy has to be found. When the binary file of a shared library is copied and named differently, both files can be loaded into the memory as different libraries. This strategy does not require any special functionality and can be implemented for Windows and Linux in the same way. This handling of multiple shared libraries and the copying of the binary file on demand is implemented by the *SharedLibraryHelper*. Starting with the manually created instance, any further instances can be accessed by the method *getNextLibrary* which copies the shared library if necessary and creates a new instance with the copied library file. If a new instance is requested but the maximum number of allowed instances is exceeded, an exception is thrown. If the exception is not caught, the simulation is shut down and according informations are presented.

9.2.4 Instance Association

As mentioned in section 9.1.2 and shown in appendix section C.1, the simulation interface includes an instance handle parameter. This handle is passed when the *set<ModuleName>Functions* function of an interface module is called and stored in the static instance information within the simulation interface module. If a function call from the openPOWERLINK stack is forwarded to the simulation interface, the stored handle is passed to the called function pointer. The counterpart within the stack interface can then assign the function call to a specific instance. This assignment is done within the derived classes of *OplkBase*. The derived classes must be implemented as singleton and therefore support only a single instance within the application. This instance holds the container of modules (given via the method *initModule*) and assigned *SharedLibraryHelper* instances. The passed instance handle represents the index within this internal container and is created after inserting the instances within the *initModule* method of *OplkBase*. This structure and the quantity of the different modules is shown in Figure 9.3.

The pure virtual method *setFunctions* of *OplkBase* must be implemented by a derived class and gets the *SharedLibraryHelper* instance and the associated instance handle as shown in Figure 9.2. Within this method, each derived class must initialize the demanded functions and connections to the simulation interface. This is done by optional creation of an according structure of function pointers pointing to static methods. After successfully resolving the set*ModuleName*Functions function, this structure with the assigned handle is passed to the according simulation interface.

If a function call is forwarded from the openPOWERLINK stack via the simulation interface to the static method, the stored instance handle is passed as parameter. This handle can be used for getting the according module instance from the internal container of modules and *SharedLibraryHelper* instances. The according method can be called using the stored module instance.

For the opposite communication the *setFunctions* method can be used for resolving the required functions from the simulation interface using a specific shared library instance and saving them in the given module instance. When calling one of those stored functions, the correct shared library instance is used due to the previous resolving of all functions.

An overview of this hierarchy and the connections of different instances is shown by an exemplary composition in Figure 9.3. The shown example

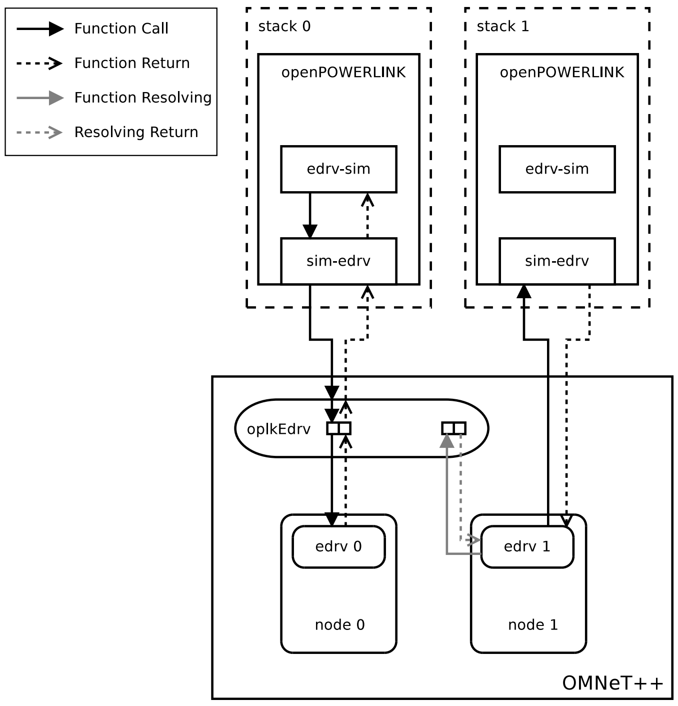

Figure 9.3: Hierarchical overview of the simulation environment, embedded modules the stack interface and simulation interface.

includes the OMNeT++ environment at the bottom containing two instantiated nodes. *node 0* and *node 1* represent separate simulated nodes within a POWERLINK network. Each node contains an *edrv* instance which is registered as a module within in the *oplkEdrv* instance. Above the OMNeT++ environment two openPOWERLINK stacks are shown, which exist independently from each other. Each openPOWERLINK stack contains the simulation interface module *sim-edrv* and the target specific implementation *edrv-sim*. The two stacks connected to the two nodes demonstrate the two directions of function calls.

The left side including *node 0* shows a call by the openPOWERLINK stack forwarding the function call from the *edrv-sim* to the simulation interface *sim-edrv*. This adds the internally stored handle to the parameters and calls the saved static method of *oplkEdrv*. Using the passed handle, the according module instance *edrv 0* is accessed and the according function is called. The shown dashed arrows mark the path of the according return values.

The right side including *node 1* requests the function object for a specific function of *sim-edrv* within stack 1. The *SharedLibraryHelper* instance saved in *oplkEdrv* returns the correct function object. This is called by *edrv 1*

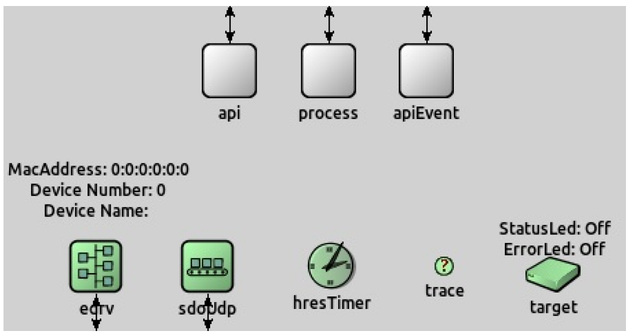

Figure 9.4: Compound module representing the openPOWERLINK stack.

and therefore the function of *sim-edrv* is directly invoked. This call is then forwarded to the openPOWERLINK stack directly within *sim-edrv*.

The implementation of *OplkBase*, *SharedLibraryHelper* and *OplkEdrv* are shown in appendix section C.2. In appendix section C.3.1 the implementation of *Edrv* is provided.

The further implementation and structure of the according OMNeT++ modules is described in the following section.

9.2.5 Stack Module

The previous mentioned OMNeT++ modules representing components within the openPOWERLINK stack are located in the *stack* folder. Additionally to the platform-dependent modules, the *Api* module was implemented providing all functions exported by the openPOWERLINK *API*. The separate modules *ApiEvent* and *ProcessSync* handle callbacks of the openPOWERLINK stack. All of these modules provide the functions which implement the required functionalities for the openPOWERLINK stack. The combination of these modules illustrate a single openPOWERLINK stack module and represent a single openPOWERLINK stack instance shown in Figure 9.4.

The different modules implement the required functionalities using OMNeT++ functionalities, e.g. scheduling self messages for implementing the timer functionality. The *Edrv* and the *Sdoudp* modules define input and output gates transmitting and receiving the according data. The *Api* also provides gates for receiving a command defining a specific *API* function which should be invoked. Occurring events and resulting return values are sent within messages by the *Api* module via the according gates.

The compound module shown in Figure 9.4 represents a single stack instance. For the usage of the stack instance, the combination with an application is necessary. This combination is implemented within the compound module *GenericNode* providing the common functionalities for *MN* and *CN*. The differently implemented nodes are described in the following section.

9.3 Simulated Nodes

Analyzing different demo applications included in the openPOWERLINK stack distribution package a general structure can be determined including the following modules.

main/demo The implementation of the main application handles the initialization, the main loop and the shutdown procedure for the demo application.

app The app module contains the handling of application specific operations, i.e. the creation of the process image and the according synchronized procedure for modifying the process image.

event The event module handles all occurring events coming from the openPOWERLINK stack.

The implementation of the nodes located within the *generic* folder should provide a basic functionality allowing a simple environment to quckly develop a demo application. Therefore, this structure was implemented within the *GenericNode* using base classes for each module as described in the following section.

9.3.1 Generic Node

All implementations regarding the generic functionalities of a simulated node are located in the *generic* folder. The compound module *GenericNode* contains an instance of the *Stack* module and modules defined by the interfaces *IDemo*, *IEvent* and *IApp*. These interfaces represent the defined structure of every node. The defined gates of each module are necessary for the base implementation and its functionalities.

This implementation is represented by the base classes *DemoBase*, *EventBase* and *AppBase*. Their implementations contain default message handling, resolving of the defined gates or parameters and statistics recording.

The designated usage of these base classes is deriving a specialized class for each specific node. These derived classes can be inserted within the *GenericNode* by setting the according parameter. The parameter *DemoType*, *EventType* and *AppType* are defined in *GenericNode* and are used for the insertion of the derived *Demo*, *Event* and *App* module.

The implementation of the *GenericNode*, the used moduleinterfaces and the base classes for each module are shown in the appendix sections C.3.2, C.3.3, C.3.4.

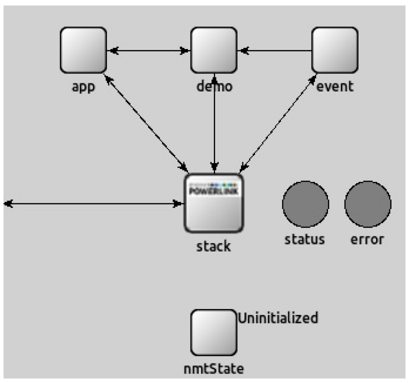

Figure 9.5: Composition of compound module representing the a generic openPOWERLINK node.

The composition of the *GenericNode* connecting the embedded *Stack* module and instances matching the different moduleinterfaces are displayed in Figure 9.5.

For a convenient display of the current state of the node the additional module *NmtState* was implemented and added to the *GenericNode*. This module does not support any communication via messages, but it subscribes to a signal of its parent module. The name of the signal can be configured via a parameter, which is set to the default value *nmtState* (defined signal of *EventBase*). If an notification for this signal is available, the received unsigned long value is casted to a NmtState and the according string is displayed.

Similar to the *NmtState* two instances of *Led* modules were added to visualize the status Leds provided by a POWERLINK node.

The *GenericNode* is designed for deriving and extending its functionality for specific nodes and applications. The functionality of an *MN* and *CN* must be implemented separately as shown in the next sections.

9.3.2 *MN* and *CN*

The implementation of the *MN* was achieved by analyzing and porting the *demo_mn_console* and the *demo_mn_embedded* to the OMNeT++ simulation. Both demo applications were analyzed for achieving the correct implementation of the simulated *MN*.

The *MN* folder within the *src* directory contains the module definition of the *MN* module and the specifically derived classes *MnDemo*, *MnEvent* and *MnApp*. These classes implement the according functions of the base classes described in section 9.3.1. This folder and the contained implementations can be used as a base project for further implementations using a simulated *MN*.

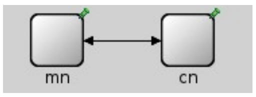

Figure 9.6: Simulated network consisting of a *MN* module directly connected to a *CN* module.

Similar to the *MN* implementation, the *CN* implementation is located in the *CN* folder containing the according module definitions and implementations. The differently implemented classes could either be replaced by alternative implementations of the moduleinterfaces or derived classes from the base classes. Another option would be that the classes are used as base classes and any further functionality is embedded. This extension is easily possible due to the virtual definition of all used methods.

The results of the developed simulation and the achieved state are described in the next section.

9.4 Results

The following sections show the achieved results in form of implemented components and designed networks.

9.4.1 Simulated Network

The implemented network for simulating a POWERLINK network consists of a *MN* module connected to a single *CN* module. This network is shown in Figure 9.6.

9.4.2 Achieved Results

For the previously described simulation the following components are completely implemented and verified by tests.

- The connection between the openPOWERLINK stack and the OMNeT++ simulation environment.
- The handling and instantiating of multiple instances.
- The definition and implementations of the necessary components for the configuration described in section 9.2.
- The composition of the *Stack* module.
- The implementation of the test network consisting of an *MN* and a *CN* as shown in section 9.4.1.

The future usage and continuing development of this simulation is described in the next chapter.

Chapter 10

Conclusion

10.1 Future Simulations

The developed OMNeT++ modules allow the simulation of a simple POWERLINK network consisting of an openPOWERLINK *MN* and a *CN*. This simulation was developed for optimal performance using a monolithic design. Alternative simulations can be implemented based on the basic strategies and reusable functionalities represented by the developed base classes, helper classes and interfaces. For a deeper insight, a modular design could be implemented by connecting the *CAL* modules with the simulation. This could be done in a similar way as handling the platform dependency by implementing according simulation specific modules redirecting the communicated calls and data into the simulation environment. Within the simulation environment even both modules of a *CAL* component could be modeled separately. This would lead to the encapsulation of the communicated data in messages. Such an implementation of *CAL* components within OMNeT++ would allow for deeper insights into the openPOWERLINK stack and the communication between kernel and user layer.

The simulated network contains a direct connection between the different nodes. This represents an optimal behavior and results in an ideal performance. For simulating the behavior of openPOWERLINK nodes in a network including real devices this connection would have to be changed. Existing functionality from the INET library regarding Ethernet and UDP could be used for converting the transmitted messages to real Ethernet frames.

10.2 Future Development

The development of the resulting simulation and the according implementations will continue to allow developers an easy access to openPOWERLINK. This easy access should be provided by a flexible simulation environment with different levels of modularity and complexity. Such an environment will provide deep insight in the openPOWERLINK stack and an possible overview for bigger networks consisting of various topologies and nodes.

This simulation will be published and continued as Open Source project available on GitHub [21] and thereby made available for every interested developer.

10.3 Emulation

The simulation mode of real-time simulation and the possibilities for emulation using OMNeT++ show a good starting point for the customized development of such applications. The built-in functionality and its usage within the examples (*sockets*) reflect the basic strategies for developing an emulation. This functionality should be used as a basis for developing more optimized and customized functionalities.

10.4 Experiences

The built-in functionalities and the corresponding possibilities of OMNeT++ are extensive. This large number of configurations and components can cause a difficult approach to the developing of a simulation. But investing more time in exploring OMNeT++ pays off well, because of the flexibility and the various fields of possible applications. Furthermore the development and testing of various systems can be eased by the benefits of an OMNeT++ simulation.

10.5 Outlook

The fields of simulation, emulation and *HiL* are steadily gaining importance owing to the growing requirement for comprehensive testing. OMNeT++ represents a dynamic framework for various applications and a high order of flexibility. This allows for the development of simulations for various fields.

The connection between simulations using OMNeT++ with embedded systems portrays a promising avenue to pursue in terms of testing and developing new systems.

Appendix A

OMNeT++ Code Snippets

A.1 cRealTimeScheduler

Listing A.1: Definition of *cRealTimeScheduler*

```
/**
 * Real-time scheduler class. When installed as scheduler using the scheduler-
   class
 * omnetpp.ini entry, it will syncronize simulation execution to real (wall
   clock)
 * time.
 *
 * Operation: a "base time" is determined when startRun() is called. Later on,
 * the scheduler object calls usleep() from getNextEvent() to synchronize the
 * simulation time to real time, that is, to wait until
 * the current time minus base time becomes equal to the simulation time.
 * Should the simulation lag behind real time, this scheduler will try to catch
   up
 * by omitting sleep calls altogether.
 *
 * Scaling is supported via the realtimescheduler-scaling omnetpp.ini entry.
 * For example, if it is set to 2.0, the simulation will try to execute twice
 * as fast as real time.
 *
 * @ingroup Internals
 */
class SIM_API cRealTimeScheduler : public cScheduler
{
  protected:
    // configuration:
    bool doScaling;
    double factor;

    // state:
    timeval baseTime;

    bool waitUntil(const timeval& targetTime);

  public:
    /**
     * Constructor.
     */
    cRealTimeScheduler() : cScheduler()  {}

    /**
     * Destructor.
     */
    virtual ~cRealTimeScheduler() {}

    /**
     * Called at the beginning of a simulation run.
     */
    virtual void startRun();

    /**
     * Called at the end of a simulation run.
     */
    virtual void endRun();

    /**
     * Recalculates "base time" from current wall clock time.
     */
    virtual void executionResumed();

    /**
     * Scheduler function -- it comes from cScheduler interface.
     * This function synchronizes to real time: it waits (usleep()) until
     * the real time reaches the time of the next simulation event.
     */
    virtual cMessage *getNextEvent();
};
```

Listing A.2: Implementation of *cRealTimeScheduler*

```
Register_Class(cRealTimeScheduler);

void cRealTimeScheduler::startRun()
{
    factor = ev.getConfig()->getAsDouble(CFGID_REALTIMESCHEDULER_SCALING);
    if (factor!=0)
        factor = 1/factor;
    doScaling = (factor!=0);

    gettimeofday(&baseTime, NULL);
}

void cRealTimeScheduler::endRun()
{
}

void cRealTimeScheduler::executionResumed()
{
    gettimeofday(&baseTime, NULL);
    baseTime = timeval_subtract(baseTime, SIMTIME_DBL(doScaling ? factor*sim->
     getSimTime() : sim->getSimTime()));
}

bool cRealTimeScheduler::waitUntil(const timeval& targetTime)
{
    // if there's more than 200ms to wait, wait in 100ms chunks
    // in order to keep UI responsiveness by invoking ev.idle()
    timeval curTime;
    gettimeofday(&curTime, NULL);
    while (targetTime.tv_sec-curTime.tv_sec >=2 ||
           timeval_diff_usec(targetTime, curTime) >= 200000)
    {
        usleep(100000); // 100ms
        if (ev.idle())
            return false;
        gettimeofday(&curTime, NULL);
    }

    // difference is now at most 100ms, do it at once
    long usec = timeval_diff_usec(targetTime, curTime);
    if (usec>0)
        usleep(usec);
    return true;
}

cMessage *cRealTimeScheduler::getNextEvent()
{
    cMessage *msg = sim->msgQueue.peekFirst();
    if (!msg)
        throw cTerminationException(eENDEDOK);

    // calculate target time
    simtime_t eventSimtime = msg->getArrivalTime();
    timeval targetTime = timeval_add(baseTime, SIMTIME_DBL(doScaling ? factor*
     eventSimtime : eventSimtime));

    // if needed, wait until that time arrives
    timeval curTime;
    gettimeofday(&curTime, NULL);
    if (timeval_greater(targetTime, curTime))
    {
        if (!waitUntil(targetTime))
            return NULL; // user break
    }
    else
    {
        // we're behind - customized versions of this class may alert
        // if we're too much behind, or modify basetime to accept the skew
    }

    // ok, return the message
    return msg;
}
```

A.2 sockets Sample

A.2.1 SocketRTScheduler

Listing A.3: Definition of *SocketRTScheduler*

```
//=========================================================================
// CSOCKETRTSCHEDULER.H - part of
//
// OMNeT++/OMNEST
// Discrete System Simulation in C++
//
// Author: Andras Varga, 2005
//
//=========================================================================

/*--------------------------------------------------------------*
  Copyright (C) 2005-2008 Andras Varga

  This file is distributed WITHOUT ANY WARRANTY. See the file
  `license' for details on this and other legal matters.
*--------------------------------------------------------------*/

#ifndef __CSOCKETRTSCHEDULER_H__
#define __CSOCKETRTSCHEDULER_H__

#include <platdep/sockets.h>
#include <platdep/timeutil.h>
#include <omnetpp.h>


/**
 * Real-time scheduler with socket-based external communication.
 *
 * \code
 * class MyInterfaceModule : public cSimpleModule
 * {
 *    cSocketRTScheduler *rtScheduler;
 *    cMessage *extEvent;
 *    char buf[4000];
 *    int numBytes;
 *    ...
 * \endcode
 *
 * \code
 * void MyInterfaceModule::initialize()
 * {
 *     extEvent = new cMessage("extEvent");
 *     rtScheduler = check_and_cast<cSocketRTScheduler *>(simulation.
    getScheduler());
 *     rtScheduler->setInterfaceModule(this, extEvent, buf, 4000, numBytes);
 * }
 * \endcode
 *
 * THIS CLASS IS JUST AN EXAMPLE -- IF YOU WANT TO DO HARDWARE-IN-THE-LOOP
 * SIMULATION, YOU WILL NEED TO WRITE YOUR OWN SCHEDULER WHICH FITS YOUR NEEDS.
 * For example, you'll probably want a different external interface than
 * a single TCP socket: maybe UDP socket, maybe raw socket to grab full Ethernet
 * frames, maybe pipe, maybe USB or other interface, etc.
 */
class cSocketRTScheduler : public cScheduler
{
  protected:
    // config
    int port;

    cModule *module;
    cMessage *notificationMsg;
    char *recvBuffer;
    int recvBufferSize;
    int *numBytesPtr;

    // state
    timeval baseTime;
    SOCKET listenerSocket;
    SOCKET connSocket;

    virtual void setupListener();
    virtual bool receiveWithTimeout(long usec);
    virtual int receiveUntil(const timeval& targetTime);

  public:
    /**
     * Constructor.
     */
    cSocketRTScheduler();

    /**
     * Destructor.
     */
    virtual ~cSocketRTScheduler();

    /**
     * Called at the beginning of a simulation run.
     */
    virtual void startRun();

    /**
     * Called at the end of a simulation run.
     */
    virtual void endRun();

    /**
     * Recalculates "base time" from current wall clock time.
     */
    virtual void executionResumed();

    /**
     * To be called from the module which wishes to receive data from the
     * socket. The method must be called from the module's initialize()
     * function.
     */
    virtual void setInterfaceModule(cModule *module, cMessage *notificationMsg,
                                    char *recvBuffer, int recvBufferSize, int *
     numBytesPtr);

    /**
     * Scheduler function -- it comes from cScheduler interface.
     */
    virtual cMessage *getNextEvent();

    /**
     * Send on the currently open connection
     */
    virtual void sendBytes(const char *buf, size_t numBytes);
};

#endif
```

Listing A.4: Implementation of *SocketRTScheduler*

```
//=========================================================================
// CSOCKETRTSCHEDULER.CC - part of
//
// OMNeT++/OMNEST
// Discrete System Simulation in C++
//
// Author: Andras Varga, 2005
//
//=========================================================================

/*--------------------------------------------------------------*
  Copyright (C) 2005-2008 Andras Varga

  This file is distributed WITHOUT ANY WARRANTY. See the file
  `license' for details on this and other legal matters.
*--------------------------------------------------------------*/


#include "SocketRTScheduler.h"

Register_Class(cSocketRTScheduler);

Register_GlobalConfigOption(CFGID_SOCKETRTSCHEDULER_PORT, "socketrtscheduler-
     port", CFG_INT, "4242", "When␣cSocketRTScheduler␣is␣selected␣as␣scheduler␣
     class:␣the␣port␣number␣cSocketRTScheduler␣listens␣on.");

inline std::ostream& operator<<(std::ostream& out, const timeval& tv)
{
    return out << (unsigned long)tv.tv_sec << "s" << tv.tv_usec << "us";
}

//--

cSocketRTScheduler::cSocketRTScheduler() : cScheduler()
{
    listenerSocket = INVALID_SOCKET;
    connSocket = INVALID_SOCKET;
}

cSocketRTScheduler::~cSocketRTScheduler()
{
}

void cSocketRTScheduler::startRun()
{
    if (initsocketlibonce()!=0)
        throw cRuntimeError("cSocketRTScheduler:␣Cannot␣initialize␣socket␣
     library");

    gettimeofday(&baseTime, NULL);

    module = NULL;
    notificationMsg = NULL;
    recvBuffer = NULL;
    recvBufferSize = 0;
    numBytesPtr = NULL;

    port = ev.getConfig()->getAsInt(CFGID_SOCKETRTSCHEDULER_PORT);
    setupListener();
}

void cSocketRTScheduler::setupListener()
{
    listenerSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (listenerSocket==INVALID_SOCKET)
        throw cRuntimeError("cSocketRTScheduler:␣cannot␣create␣socket");

    sockaddr_in sinInterface;
    sinInterface.sin_family = AF_INET;
    sinInterface.sin_addr.s_addr = INADDR_ANY;
    sinInterface.sin_port = htons(port);
    if (bind(listenerSocket, (sockaddr*)&sinInterface, sizeof(sockaddr_in))==SOCKET_ERROR)
        throw cRuntimeError("cSocketRTScheduler:␣socket␣bind()␣failed");

    listen(listenerSocket, SOMAXCONN);
}

void cSocketRTScheduler::endRun()
{
}

void cSocketRTScheduler::executionResumed()
{
    gettimeofday(&baseTime, NULL);
    baseTime = timeval_substract(baseTime, SIMTIME_DBL(simTime()));
}

void cSocketRTScheduler::setInterfaceModule(cModule *mod, cMessage *notifMsg, char *buf, int bufSize, int *nBytesPtr)
{
    if (module)
        throw cRuntimeError("cSocketRTScheduler:␣setInterfaceModule()␣already␣called");
    if (!mod || !notifMsg || !buf || !bufSize || !nBytesPtr)
        throw cRuntimeError("cSocketRTScheduler:␣setInterfaceModule():␣arguments␣must␣be␣non-NULL");

    module = mod;
    notificationMsg = notifMsg;
    recvBuffer = buf;
    recvBufferSize = bufSize;
    numBytesPtr = nBytesPtr;
    *numBytesPtr = 0;
}

bool cSocketRTScheduler::receiveWithTimeout(long usec)
{
    // prepare sets for select()
    fd_set readFDs, writeFDs, exceptFDs;
    FD_ZERO(&readFDs);
    FD_ZERO(&writeFDs);
    FD_ZERO(&exceptFDs);

    // if we’re connected, watch connSocket, otherwise accept new connections
    if (connSocket!=INVALID_SOCKET)
        FD_SET(connSocket, &readFDs);
    else
        FD_SET(listenerSocket, &readFDs);

    timeval timeout;
    timeout.tv_sec = 0;
    timeout.tv_usec = usec;

    if (select(FD_SETSIZE, &readFDs, &writeFDs, &exceptFDs, &timeout) > 0)
    {
        // Something happened on one of the sockets - handle them
        if (connSocket!=INVALID_SOCKET && FD_ISSET(connSocket, &readFDs))
        {
            // receive from connSocket
            char *bufPtr = recvBuffer + (*numBytesPtr);
            int bufLeft = recvBufferSize - (*numBytesPtr);
            if (bufLeft<=0)
                throw cRuntimeError("cSocketRTScheduler:␣interface␣module's␣recvBuffer␣is␣full");
            int nBytes = recv(connSocket, bufPtr, bufLeft, 0);
            if (nBytes==SOCKET_ERROR)
            {
                EV << "cSocketRTScheduler:␣socket␣error␣" << sock_errno() << "\n";
                closesocket(connSocket);
                connSocket = INVALID_SOCKET;
            }
            else if (nBytes == 0)
            {
                EV << "cSocketRTScheduler:␣socket␣closed␣by␣the␣client\n";
                if (shutdown(connSocket, SHUT_WR) == SOCKET_ERROR)
                    throw cRuntimeError("cSocketRTScheduler:␣shutdown()␣failed");
                closesocket(connSocket);
                connSocket = INVALID_SOCKET;
            }
            else
            {
                // schedule notificationMsg for the interface module
                EV << "cSocketRTScheduler:␣received␣" << nBytes << "␣bytes\n";
                (*numBytesPtr) += nBytes;

                timeval curTime;
                gettimeofday(&curTime, NULL);
                curTime = timeval_substract(curTime, baseTime);
                simtime_t t = curTime.tv_sec + curTime.tv_usec*1e-6;
                // TBD assert that it's somehow not smaller than previous event's time
                notificationMsg->setArrival(module,-1,t);
                simulation.msgQueue.insert(notificationMsg);
                return true;
            }
        }
        else if (FD_ISSET(listenerSocket, &readFDs))
        {
            // accept connection, and store FD in connSocket
            sockaddr_in sinRemote;
            int addrSize = sizeof(sinRemote);
            connSocket = accept(listenerSocket, (sockaddr*)&sinRemote, (socklen_t*)&addrSize);
            if (connSocket==INVALID_SOCKET)
                throw cRuntimeError("cSocketRTScheduler:␣accept()␣failed");
            EV << "cSocketRTScheduler:␣connected!\n";
        }
    }
    return false;
}

int cSocketRTScheduler::receiveUntil(const timeval& targetTime)
{
    // if there's more than 200ms to wait, wait in 100ms chunks
    // in order to keep UI responsiveness by invoking ev.idle()
    timeval curTime;
    gettimeofday(&curTime, NULL);
    while (targetTime.tv_sec-curTime.tv_sec >=2 ||
           timeval_diff_usec(targetTime, curTime) >= 200000)
    {
        if (receiveWithTimeout(100000)) // 100ms
            return 1;
        if (ev.idle())
            return -1;
        gettimeofday(&curTime, NULL);
    }

    // difference is now at most 100ms, do it at once
    long usec = timeval_diff_usec(targetTime, curTime);
    if (usec>0)
        if (receiveWithTimeout(usec))
            return 1;
    return 0;
}

cMessage *cSocketRTScheduler::getNextEvent()
{
    // assert that we've been configured
    if (!module)
        throw cRuntimeError("cSocketRTScheduler:␣setInterfaceModule()␣not␣called
     :␣it␣must␣be␣called␣from␣a␣module's␣initialize()␣function");

    // calculate target time
    timeval targetTime;
    cMessage *msg = sim->msgQueue.peekFirst();
    if (!msg)
    {
        // if there are no events, wait until something comes from outside
        // TBD: obey simtimelimit, cpu-time-limit
        targetTime.tv_sec = LONG_MAX;
        targetTime.tv_usec = 0;
    }
    else
    {
        // use time of next event
        simtime_t eventSimtime = msg->getArrivalTime();
        targetTime = timeval_add(baseTime, SIMTIME_DBL(eventSimtime));
    }

    // if needed, wait until that time arrives
    timeval curTime;
    gettimeofday(&curTime, NULL);
    if (timeval_greater(targetTime, curTime))
    {
        int status = receiveUntil(targetTime);
        if (status == -1)
            return NULL; // interrupted by user
        if (status == 1)
            msg = sim->msgQueue.peekFirst(); // received something
    }
    else
    {
        // we're behind - customized versions of this class may
        // alert if we're too much behind, whatever that means
    }

    // ok, return the message
    return msg;
}

void cSocketRTScheduler::sendBytes(const char *buf, size_t numBytes)
{
    if (connSocket==INVALID_SOCKET)
        throw cRuntimeError("cSocketRTScheduler:␣sendBytes():␣no␣connection");

    send(connSocket, buf, numBytes, 0);
    // TBD check for errors
}
```

A.2.2 ExtHttpClient

Listing A.5: Implementation of *ExtHttpClient*

```
//
// This file is part of an OMNeT++/OMNEST simulation example.
//
// Copyright (C) 1992-2008 Andras Varga
//
// This file is distributed WITHOUT ANY WARRANTY. See the file
// 'license' for details on this and other legal matters.
//


#include <platdep/sockets.h>
#include <omnetpp.h>
#include "HttpMsg_m.h"
#include "SocketRTScheduler.h"


/**
 * Model of a web browser.
 */
class ExtHTTPClient : public cSimpleModule
{
  private:
    cMessage *rtEvent;
    cSocketRTScheduler *rtScheduler;

    char recvBuffer[4000];
    int numRecvBytes;

    int addr;
    int srvAddr;

  public:
    ExtHTTPClient();
    virtual ~ExtHTTPClient();

  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    void handleSocketEvent();
    void handleReply(HTTPMsg *httpReply);
};

Define_Module(ExtHTTPClient);

ExtHTTPClient::ExtHTTPClient()
{
    rtEvent = NULL;
}

ExtHTTPClient::~ExtHTTPClient()
{
    cancelAndDelete(rtEvent);
}

void ExtHTTPClient::initialize()
{
    rtEvent = new cMessage("rtEvent");
    rtScheduler = check_and_cast<cSocketRTScheduler *>(simulation.getScheduler()
     );
    rtScheduler->setInterfaceModule(this, rtEvent, recvBuffer, 4000, &
     numRecvBytes);

    addr = par("addr");
    srvAddr = par("srvAddr");
}

void ExtHTTPClient::handleMessage(cMessage *msg)
{
    if (msg==rtEvent)
        handleSocketEvent();
    else
        handleReply(check_and_cast<HTTPMsg *>(msg));
}

void ExtHTTPClient::handleSocketEvent()
{
    // try to find a double line feed in the input - that's the end of the HTTP
     header.
    char *endHeader = NULL;
    for (char *s=recvBuffer; s<=recvBuffer+numRecvBytes-4; s++)
        if (*s=='\r' && *(s+1)=='\n' && *(s+2)=='\r' && *(s+3)=='\n')
            {endHeader = s+4; break;}

    // we don't have a complete header yet - keep on waiting
    if (!endHeader)
        return;
    std::string header = std::string(recvBuffer, endHeader-recvBuffer);
    //EV « header;

    // remove HTTP header from buffer
    if (endHeader == recvBuffer+numRecvBytes)
        numRecvBytes = 0;
    else {
        int bytesLeft = recvBuffer+numRecvBytes-endHeader;
        memmove(endHeader, recvBuffer, bytesLeft);
        numRecvBytes = bytesLeft;
    }

    // assemble and send HTTP request
    HTTPMsg *httpMsg = new HTTPMsg();
    httpMsg->setPayload(header.c_str());
    httpMsg->setDestAddress(srvAddr);
    httpMsg->setSrcAddress(addr);

    send(httpMsg,"g$o");
}

void ExtHTTPClient::handleReply(HTTPMsg *httpReply)
{
    const char *reply = httpReply->getPayload();
    rtScheduler->sendBytes(reply, strlen(reply));
    delete httpReply;
}
```

A.2.3 ExtTelnetClient

Listing A.6: Implementation of *ExtTelnetClient*

```
//
// This file is part of an OMNeT++/OMNEST simulation example.
//
// Copyright (C) 1992-2008 Andras Varga
//
// This file is distributed WITHOUT ANY WARRANTY. See the file
// 'license' for details on this and other legal matters.
//


#include <platdep/sockets.h>
#include <omnetpp.h>
#include "TelnetPkt_m.h"
#include "SocketRTScheduler.h"


/**
 * Simple model of a Telnet client.
 */
class ExtTelnetClient : public cSimpleModule
{
  private:
    cMessage *rtEvent;
    cSocketRTScheduler *rtScheduler;

    char recvBuffer[4000];
    int numRecvBytes;

    int addr;
    int srvAddr;

  public:
    ExtTelnetClient();
    virtual ~ExtTelnetClient();

  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    void handleSocketEvent();
    void handleReply(TelnetPkt *telnetReply);
};

Define_Module(ExtTelnetClient);

ExtTelnetClient::ExtTelnetClient()
{
    rtEvent = NULL;
}

ExtTelnetClient::~ExtTelnetClient()
{
    cancelAndDelete(rtEvent);
}

void ExtTelnetClient::initialize()
{
    rtEvent = new cMessage("rtEvent");
    rtScheduler = check_and_cast<cSocketRTScheduler *>(simulation.getScheduler());
    rtScheduler->setInterfaceModule(this, rtEvent, recvBuffer, 4000, &numRecvBytes);

    addr = par("addr");
    srvAddr = par("srvAddr");
}

void ExtTelnetClient::handleMessage(cMessage *msg)
{
    if (msg==rtEvent)
        handleSocketEvent();
    else
        handleReply(check_and_cast<TelnetPkt *>(msg));
}

void ExtTelnetClient::handleSocketEvent()
{
    // get data from buffer
    std::string text = std::string(recvBuffer, numRecvBytes);
    numRecvBytes = 0;

    // assemble and send Telnet packet
    TelnetPkt *telnetPkt = new TelnetPkt();
    telnetPkt->setPayload(text.c_str());
    telnetPkt->setName(text.c_str());
    telnetPkt->setDestAddress(srvAddr);
    telnetPkt->setSrcAddress(addr);

    send(telnetPkt,"g$o");
}

void ExtTelnetClient::handleReply(TelnetPkt *telnetReply)
{
    const char *reply = telnetReply->getPayload();
    rtScheduler->sendBytes(reply, strlen(reply));
    delete telnetReply;
}
```

Appendix B

Further design Measurements

This chapter includes all plots of analyzed design measurements executed on the additional host machines.

B.1 Workstation measurements

B.1.1 Runtime

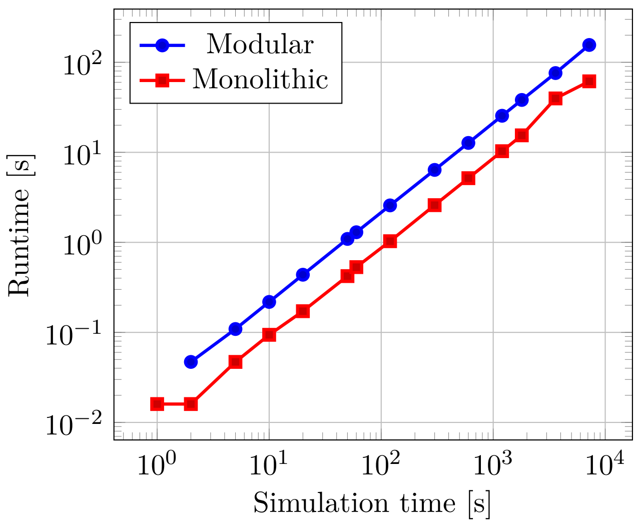

Figure B.1: Runtime results for different designs over a varying simulation time.

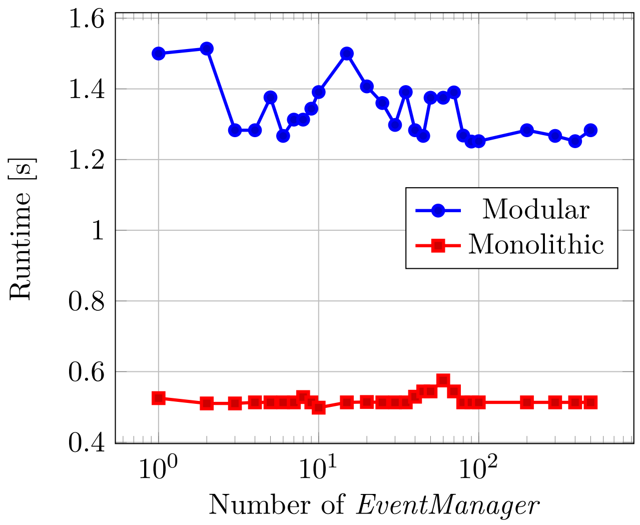

Figure B.2: Runtime using different designs over variing number of *EventManager*.

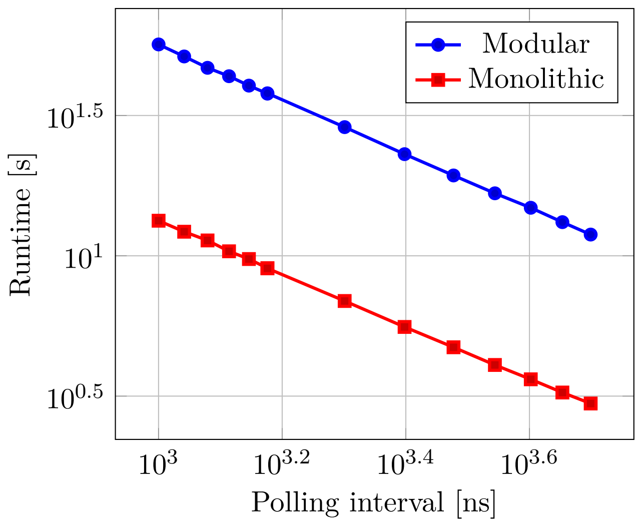

Figure B.3: Runtime using different designs over varying polling interval by the *HistoryManager*.

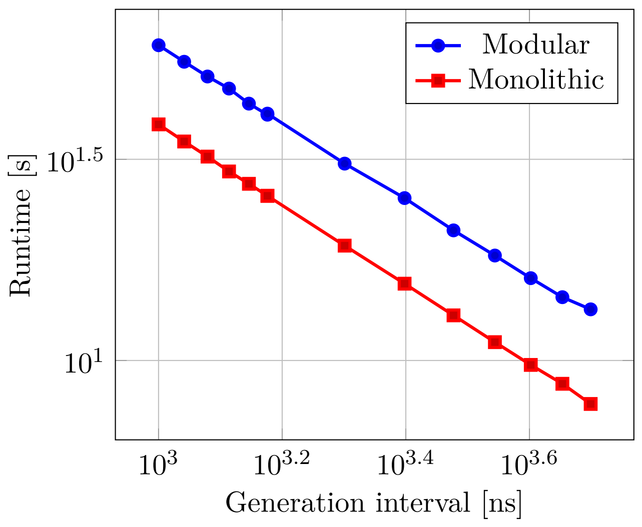

Figure B.4: Runtime using different designs over varying generation interval by the *Generator*.

B.1.2 Event

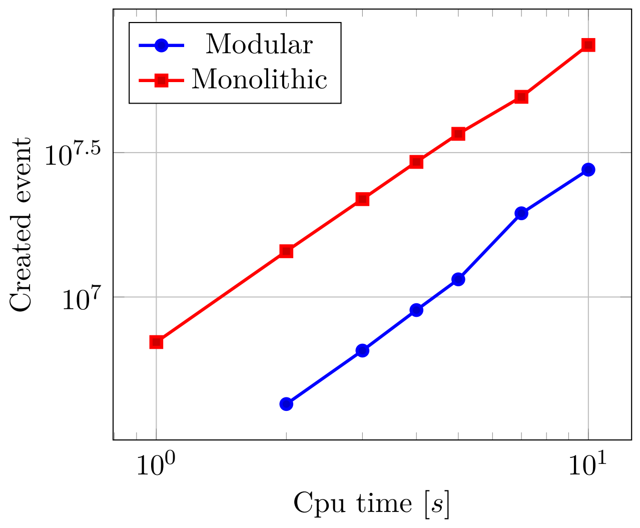

Figure B.5: Created events for different designs over different cpu time limits.

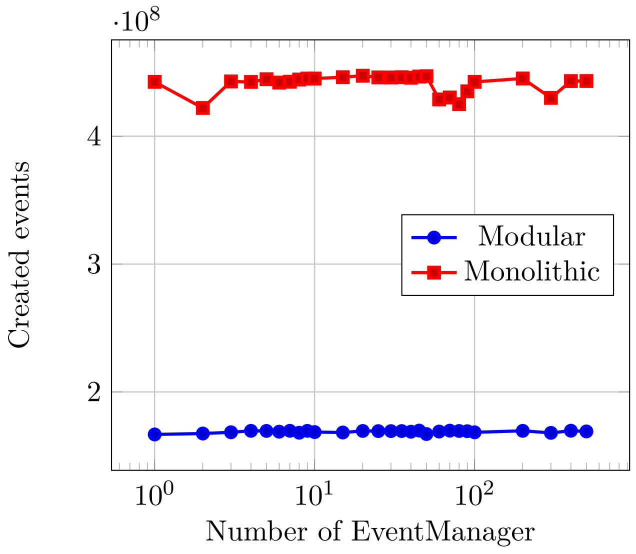

Figure B.6: Created events using different designs over varying number of *EventManager*.

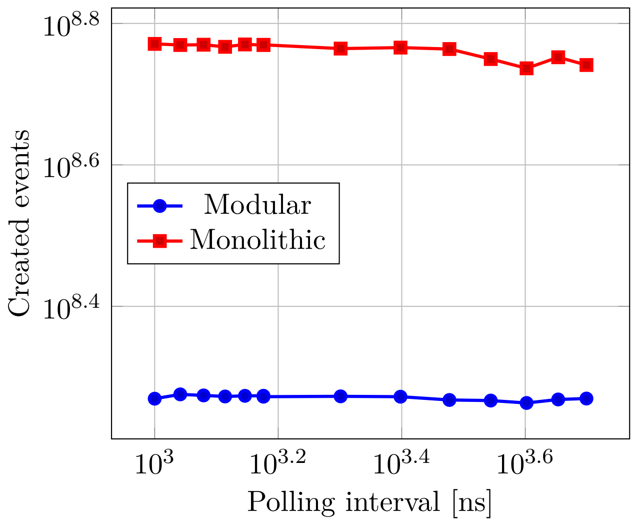

Figure B.7: Created events using different designs over varying polling interval by the *HistoryManager*.

B.2 Build server measurements

B.2.1 Runtime

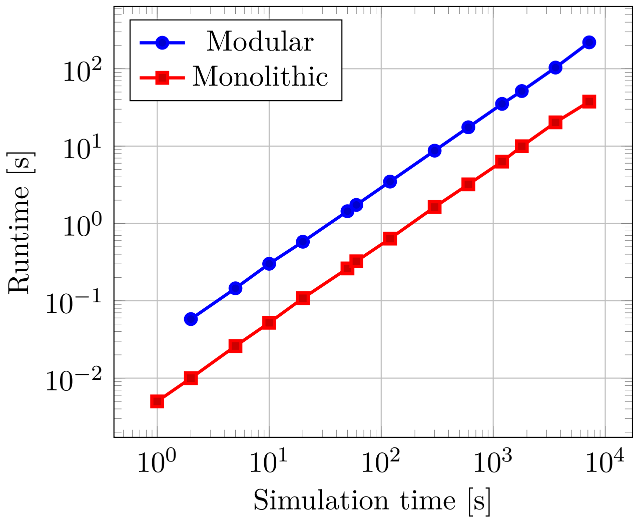

Figure B.8: Runtime results for different designs over a varying simulation time.

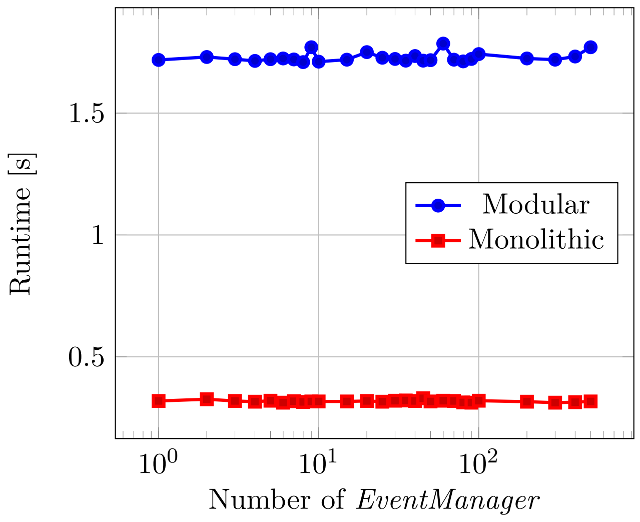

Figure B.9: Runtime using different designs over variing number of *Event-Manager*.

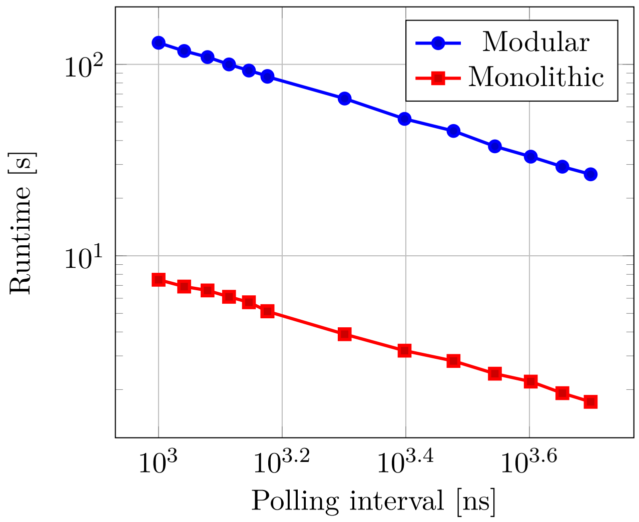

Figure B.10: Runtime using different designs over varying polling interval by the *HistoryManager*.

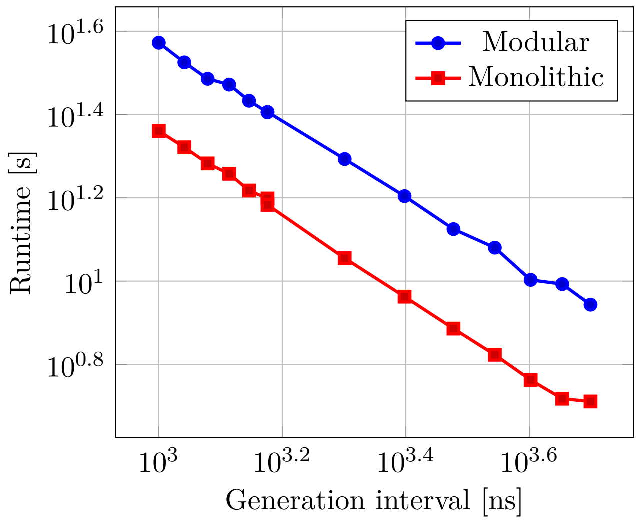

Figure B.11: Runtime using different designs over varying generation interval by the *Generator*.

B.2.2 Event

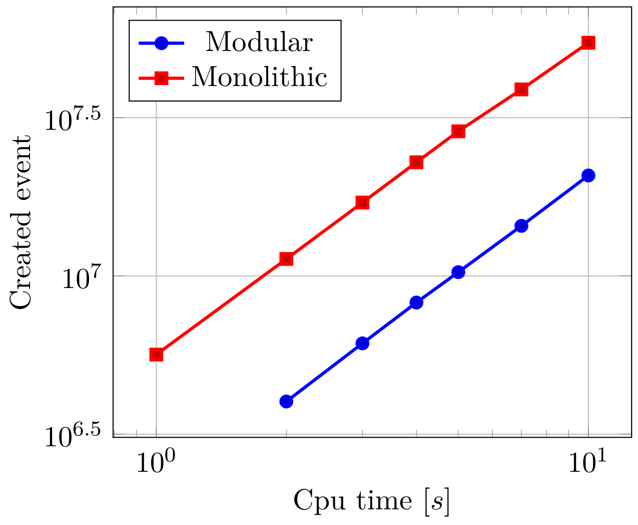

Figure B.12: Created events for different designs over different cpu time limits.

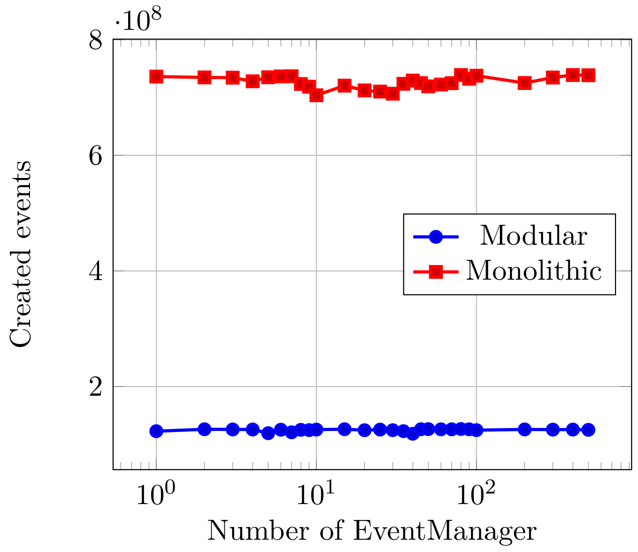

Figure B.13: Created events using different designs over varying number of *EventManager*.

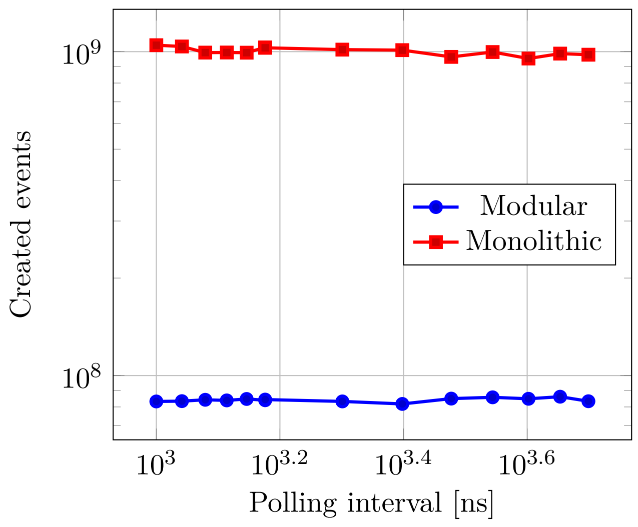

Figure B.14: Created events using different designs over varying polling interval by the *HistoryManager*.

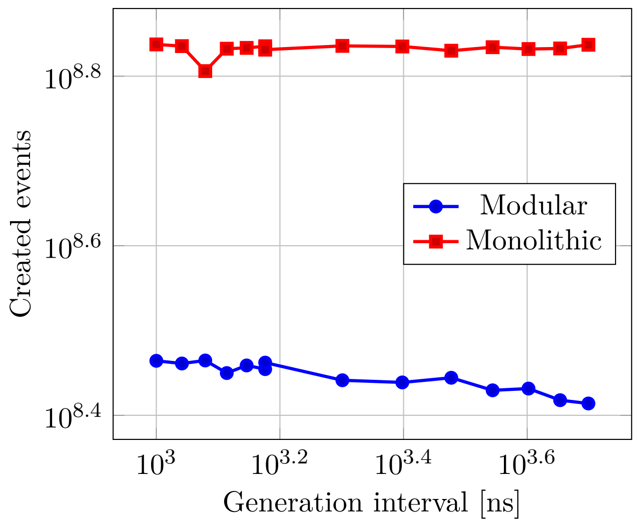

Figure B.15: Created events using different designs over varying generation interval by the *Generator*.

B.2.3 Real-time

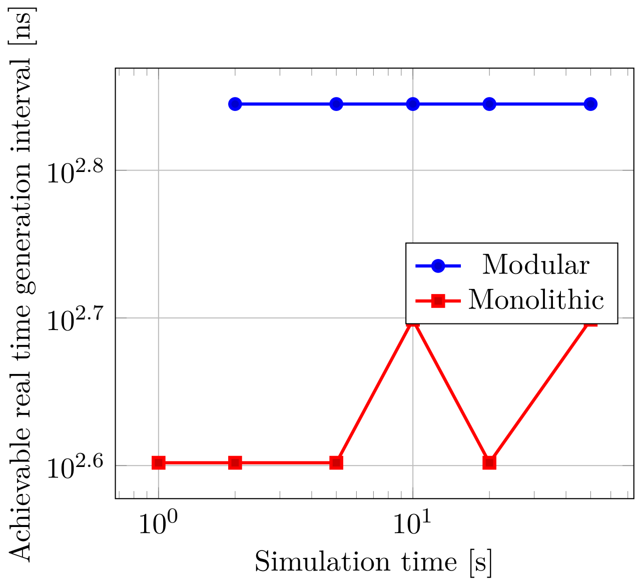

Figure B.16: Real-time results for different designs over different simulation time limits.

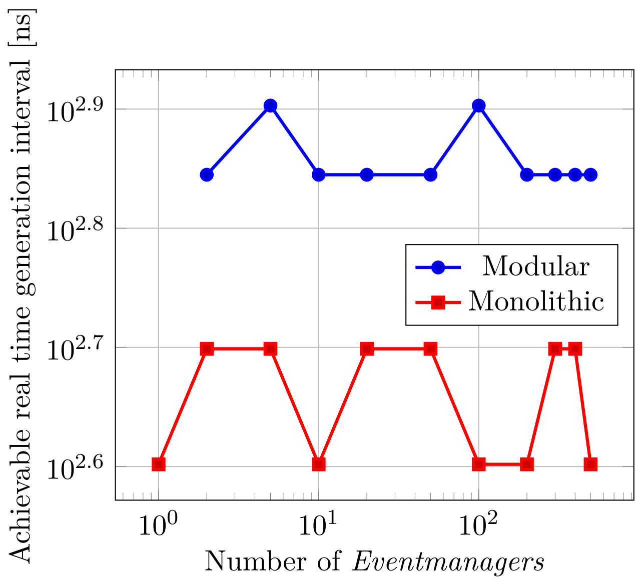

Figure B.17: Real-time results for different designs over a varying number of *EventManagers*.

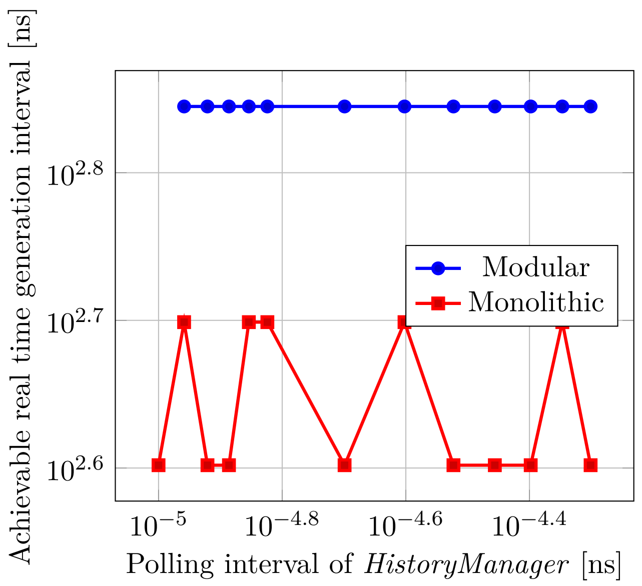

Figure B.18: Real-time results for different designs over a varying polling interval of *HistoryManager*.

Appendix C

Simulation Code snippets

C.1 Simulation interface for Ethernet driver module

C.1.1 Implementation of simulation specific module

Listing C.1: edrv-sim.c

```
//---------------------------------------
/**
\brief  Send Tx buffer

This function sends the Tx buffer.

\param  pBuffer_p           Tx buffer descriptor

\return The function returns a tOplkError error code.

\ingroup module_edrv
*/
//---------------------------------------
tOplkError edrv_sendTxBuffer(tEdrvTxBuffer* pBuffer_p)
{
    return sim_sendTxBuffer(pBuffer_p);
}
```

C.1.2 Definitions of simulation interface types

Listing C.2: sim.h

```
//---------------------------------------
// edrv types
//---------------------------------------

/**
\brief Type for initEdrv function pointer

This type defines a function pointer for the simulation interface function
 for initEdrv.

\param simInstanceHdl_p     The handle of the current simulated stack instance
\param pEdrvInitParam_p     Pointer to the edrv init parameter
```

```

\return The function returns a tOplkError error code
*/
typedef tOplkError(*tInitEdrvFunction)(tSimulationInstanceHdl simInstanceHdl_p,
                                       tEdrvInitParam *pEdrvInitParam_p);

/**
\brief Type for exitEdrv function pointer

This type defines a function pointer for the simulation interface function
 for exitEdrv.

\param simInstanceHdl_p     The handle of the current simulated stack instance

\return The function returns a tOplkError error code
*/
typedef tOplkError(*tExitEdrvFunction)(tSimulationInstanceHdl simInstanceHdl_p);

/**
\brief  Type for getMacAddr function pointer

This type defines a function pointer for the simulation interface function
 for getMacAddr.

\param simInstanceHdl_p     The handle of the current simulated stack instance

\return The function returns a pointer to the MAC address.
*/
typedef UINT8 *(*tgetMacAddrFunction)(tSimulationInstanceHdl simInstanceHdl_p);

/**
\brief   Type for Ethernet txBuffer function pointer

This type defines a function pointer for the simulation interface function
 for allocTxBuffer and freeTxBuffer.

\param simInstanceHdl_p     The handle of the current simulated stack instance
\param  pBuffer_p           Tx buffer descriptor

\return The function returns a tOplkError error code.
*/
typedef tOplkError(*tTxBufferFunction)(tSimulationInstanceHdl simInstanceHdl_p,
                                       tEdrvTxBuffer * pBuffer_p);

/**
\brief  Type for changeRxBuffer function pointer

This type defines a function pointer for the simulation interface function
 for changeRxBuffer.

\param simInstanceHdl_p     The handle of the current simulated stack instance
\param  pFilter_p           Base pointer of Rx filter array
\param  count_p             Number of Rx filter array entries
\param  entryChanged_p      Index of Rx filter entry that shall be changed
\param  changeFlags_p       Bit mask that selects the changing Rx filter
     property

\return The function returns a tOplkError error code.
*/
```

```
typedef tOplkError(*tChangeRxBufferFunction)(tSimulationInstanceHdl simInstanceHdl_p,
                                             tEdrvFilter *pFilter_p, UINT count_p,
                                             UINT entryChanged_p, UINT changeFlags_p);

/**
\brief  Type for Ethernet multicast function pointer

This type defines a function pointer for the simulation interface function
 for setRxMulticastMacAddr and clearRxMulticastMacAddr.

\param simInstanceHdl_p     The handle of the current simulated stack instance
\param  pMacAddr_p           Base pointer of Rx filter array

\return The function returns a tOplkError error code.
*/
typedef tOplkError(*tMulticastFunction)(tSimulationInstanceHdl simInstanceHdl_p,
                                        UINT8 * pMacAddr_p);

/**
\brief Edrv function pointer

This struct holds all funtion pointer to the edrv functions used in the
 simulation interface (\ref sim-edrv.g).
*/
typedef struct
{
    tInitEdrvFunction pfnInit;                          ///< Pointer to the initEdrv function
    tExitEdrvFunction pfnExit;                          ///< Pointer to the exitEdrv function
    tgetMacAddrFunction pfnGetMacAddr;                  ///< Pointer to the getMacAddr function
    tTxBufferFunction pfnSendTxBuffer;                  ///< Pointer to the sendTxBuffer function
    tTxBufferFunction pfnAllocTxBuffer;                 ///< Pointer to the allocTxBuffer function
    tTxBufferFunction pfnFreeTxBuffer;                  ///< Pointer to the freeTxBuffer function
    tChangeRxBufferFunction pfnChangeRxBufferFiler;     ///< Pointer to the changeRcBufferFilter function
    tMulticastFunction pfnSetMulticastMacAddr;          ///< Pointer to the setMulticastMacAddr function
    tMulticastFunction pfnClearMulticastMacAddr;        ///< Pointer to the clearMulticastMacAddr function
} tEdrvFunctions;
```

C.1.3 Instance information

Listing C.3: sim-edrv.c

```
/**
\brief  Instance struct for sim-edrv module

This struct contains informations about the current instance.
*/
typedef struct
```

```
{
    tEdrvFunctions edrvFunctions;   ///< Struct with all simulation interface
     functions
    tSimulationInstanceHdl simHdl;  ///< Handle to running simulation for
     multiple simulated instances
    BOOL fInitialized;              ///< Initialization flag signalling if the
     stores functions are valid
} tSimEdrvInstance;

//---------------------------------------
// local vars
//---------------------------------------

static tSimEdrvInstance instance_l = {{NULL}, 0, FALSE};
```

C.1.4 Definition and Implementation of setEdrvFunctions

Listing C.4: sim-edrv.h

```
OPLKDLLEXPORT BOOL sim_setEdrvFunctions(tSimulationInstanceHdl simHdl,
                                        tEdrvFunctions edrvFunctions_p);
```

Listing C.5: sim-edrv.c

```
/**
\brief  Setter for simulation interface functions for edrv

This function sets the simulation interface functions for edrv

\param simInstanceHdl_p     The handle of the current simulated stack instance
\param edrvFunctions_p      Structure with all simulation interface functions

\return The function returns TRUE when the all given functions are valid
    and the structure was set internally, otherwise the function returns FALSE
*/
BOOL sim_setEdrvFunctions(tSimulationInstanceHdl simHdl_p,
                          tEdrvFunctions edrvFunctions_p)
{
    if (instance_l.fInitialized != TRUE)
    {
        // check function pointer
        if ((edrvFunctions_p.pfnInit == NULL) ||
            (edrvFunctions_p.pfnExit == NULL) ||
            (edrvFunctions_p.pfnGetMacAddr == NULL) ||
            (edrvFunctions_p.pfnSendTxBuffer == NULL) ||
            (edrvFunctions_p.pfnAllocTxBuffer == NULL) ||
            (edrvFunctions_p.pfnFreeTxBuffer == NULL) ||
            (edrvFunctions_p.pfnChangeRxBufferFiler == NULL) ||
            (edrvFunctions_p.pfnSetMulticastMacAddr == NULL) ||
            (edrvFunctions_p.pfnClearMulticastMacAddr == NULL))
            return FALSE;

        instance_l.edrvFunctions = edrvFunctions_p;
        instance_l.simHdl = simHdl_p;
        instance_l.fInitialized = TRUE;

        return TRUE;
    }

    return FALSE;
}
```

C.1.5 Definition and Implementation of sendTxBuffer

Listing C.6: sim-edrv.h

```
tOplkError sim_sendTxBuffer(tEdrvTxBuffer *pBuffer_p);
```

Listing C.7: sim-edrv.c

```
/**
\brief  Send Tx buffer

This function forwards the sending call to the configured simulation interface.

\param  pBuffer_p           Tx buffer descriptor

\return The function returns a tOplkError error code.
*/
tOplkError sim_sendTxBuffer(tEdrvTxBuffer *pBuffer_p)
{
    // check if functions are initialized
    if (instance_l.fInitialized == TRUE)
    {
        return instance_l.edrvFunctions
                        .pfnSendTxBuffer(instance_l.simHdl, pBuffer_p);
    }

    return kErrorApiNotInitialized;
}
```

C.2 Stack interface for Ethernet driver module

C.2.1 SharedLibraryHelper

Listing C.8: SharedLibraryHelper.h

```
/**
    *******************************************************************************

 \file   SharedLibrary.h

 \brief  Include file for operating system specific calls for shared library
 access

 ******************************************************************************
    */

/*------------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 ------------------------------------------------------------------------------
    */

#ifndef _INC_shared_library_H_
#define _INC_shared_library_H_
```

```

//---------------------------------------
// includes
//---------------------------------------

#include <fstream>
#include <string>
#include <functional>
#include <memory>

#if defined(__linux__)
#include <dlfcn.h>
#include <link.h>
#elif defined(_WIN32)
#include <Windows.h>
#include <Winbase.h>
#endif

//---------------------------------------
// const defines
//---------------------------------------

//---------------------------------------
// typedef
//---------------------------------------

namespace interface
{

#if defined(__linux__)
    using LibraryHandle = void*;
#elif defined(_WIN32)
    using LibraryHandle = HMODULE;
#endif

    /**
     * \brief Helper class for handling shared libraries of Linux and Windows OS
     *
     * This class provides either static method for loading, unloading of
     * shared libraries and resolving of functions.
     * Using an instance the RAII principle is rrealized with loading the given
     * shared library during creation and unloading it during destruction.
     *
     * Multiple versions of the same shared library can be created as used due
     * to internal copies of thte shared library file.
     */
    class SharedLibraryHelper
    {
            // Definitions
        public:
            using InstanceHandle = unsigned int;
            using InstanceType = int;
            using HelperPtr = std::shared_ptr<SharedLibraryHelper>;

            // C-Tor / D-Tor
        public:
            SharedLibraryHelper(SharedLibraryHelper const &) = delete;
            SharedLibraryHelper(SharedLibraryHelper &&) = default;

        private:
            /**
             * \brief Constructor with given library name.
             *
             * \param libname   Name of the shared library without extension
             */
            explicit SharedLibraryHelper(std::string const & libname);

            /**
             * \brief Constructor with given library name and a maximum number
             * of allowed instances.
             *
             * \param libname                     Name of the shared library
             *                                    without extension
             * \param numberOfParallelInstances   Number of maxmimum creatable
             *                                    instances
             */
            explicit SharedLibraryHelper(std::string const & libname,
                    InstanceType numberOfParallelInstances);

            /**
             * \brief Constructor with given library name, a maximum number of
             * allowed instances and the current instance number.
             *
             * \param libname                     Name of the shared library
             *                                    without extension
             * \param numberOfParallelInstances   Number of maxmimum creatable
             *                                    instances
             * \param instanceId                  Number of the created instance
             */
            explicit SharedLibraryHelper(std::string const & libname,
                    InstanceType numberOfParallelInstances,
                    InstanceType instanceId);

        public:
            ~SharedLibraryHelper();

            // Methods
        public:
            /**
             * \brief Creates and returns the next instance of the stored shared
             * library
             *
             * \return Returns a std::shared_ptr<SharedLibraryHelper> of the
             * newly created instance
             */
            HelperPtr getNextLibrary();

            /**
             * \brief Creates and returns the next instance with the given
             * shared library name
             *
             * \param libraryName   Name of the shared library
             * \return Returns a std::shared_ptr<SharedLibraryHelper> of the
             * newly created instance
             */
            HelperPtr getNextLibrary(std::string const & libraryName);

            /**
```

```
         * \brief Resolve function with given function name and according
         * template types from saved shared library
         *
         * This method calls the static resolve function method passing the
         * saved library handle.
         *
         * \param functionName      Name of the function which should be
         * resolved
         * \return std::function object with the according template
         * parameter, or nullptr when resolving fails
         * \tparam TRet Returntype of resolved function
         * \tparam TArgs Variadic types of function arguments
         */
        template<typename TRet, typename ... TArgs>
        std::function<TRet(TArgs...)> getFunction(
                std::string const & functionName)
        {
            return SharedLibraryHelper::resolveFunction<TRet, TArgs...>(
                    mLibHandle, functionName);
        }

        // Helper Methods
    private:
        std::string getLibraryName();
        void createLibraryInstance();

        // Static Methods
    public:
        /**
         * \brief Create an instance of SharedLibraryHelper with given
         * library name.
         *
         * \param libname   Name of the shared library
         * \return Returns a std::shared_ptr<SharedLibraryHelper> of the
         * newly created instance
         */
        static HelperPtr createInstance(std::string const & libname);

        /**
         * \brief Create an instance of SharedLibraryHelper with given
         * library name and maximum number of allowed instances.
         *
         * \param libname                        Name of the shared library
         * \param numberOfParallelInstances     Maximum number of allowed
         * instances
         * \return Returns a std::shared_ptr<SharedLibraryHelper> of the
         * newly created instance
         */
        static HelperPtr createInstance(std::string const & libname,
                InstanceType numberOfParallelInstances);

        /**
         * \brief Load given shared library into memory.
         *
         * This method is platform independent and calls the according
         * dependent method
         *
         * \param libname Name of the shared Library
         * \return Handle of the loaded library
```

```
             * \throws Throws a std::runtime_error when an error occurs during
             * loading
             */
            static LibraryHandle openSharedLibrary(std::string const & libname);

            /**
             * \brief Resolve function with given function name and according
             * template types
             * from given shared library
             *
             * This method is platform independent and calls the according
             * dependent method
             *
             * \param handle            Handle of the shared Library
             * \param functionName      Name of the function to resolve
             * \return std::function object with the according template
             * parameter, or nullptr when resolving fails
             * \tparam TRet Returntype of resolved function
             * \tparam TArgs Variadic types of function arguments
             */
            template<typename TRet, typename ... TArgs>
            static std::function<TRet(TArgs...)> resolveFunction(
                    LibraryHandle handle, std::string const & functionName)
            {
                // forward to OS specific method
#if defined(__linux__)
                return SharedLibraryHelper::resolveFunctionLinux<TRet, TArgs ...>(
                        handle, functionName);
#elif defined(_WIN32)
                return SharedLibraryHelper::resolveFunctionWindows<TRet, TArgs ...>(
                        handle, functionName);
#endif
            }

            /**
             * \brief Unload given shared library from memory.
             *
             * This method is platform independent and calls the according
             * dependent method
             *
             * \param handle Handle of the shared library
             * \throws Throws a std::runtime_error when an error occurs during
             * unloading
             */
            static void closeShareLibrary(LibraryHandle handle);

            /**
             * \brief Checks if the shared library with given name is currenty
             * loaded within the application
             *
             * This method is platform independent and calls the according
             * dependent method
             *
             * \param libname Name of the shared library
             * \throws Throws a std::runtime_error when an error occurs during
             * checking
             */
            static bool isSharedLibraryLoaded(std::string const & libname);

            /**
             * \brief Getter for the recent error description occuring within
             * shared library API
             *
             * This method is platform independent and calls the according
             * dependent method
             *
             * \return Returns a error description std::string
             */
            static std::string getError();

            // Static helper methods
        private:
            static LibraryHandle openSharedLibraryLinux(
                    std::string const & libname);
            static LibraryHandle openSharedLibraryWindows(
                    std::string const & libname);

            template<typename TRet, typename ... TArgs>
            static std::function<TRet(TArgs...)> resolveFunctionLinux(
                    LibraryHandle handle, std::string const & functionName)
            {
                auto cName = functionName.c_str();

                auto rawFunc = dlsym(handle, cName);
                std::function<TRet(TArgs...)> func = reinterpret_cast<TRet (*)(
                        TArgs...)>(rawFunc);

                return func;
            }

            template<typename TRet, typename ... TArgs>
            static std::function<TRet(TArgs...)> resolveFunctionWindows(
                    LibraryHandle handle, std::string const & functionName)
            {
#if defined(_WIN32)
                std::function<TRet(TArgs...)> func;
                // TODO implement windows version
                return func;
#else
                throw std::runtime_error(
                        "error␣windows␣function␣called␣under␣different␣OS");
#endif
            }

            static void closeShareLibraryLinux(LibraryHandle handle);
            static void closeShareLibraryWindows(LibraryHandle handle);

            static bool isSharedLibraryLoadedLinux(std::string const & libname);
            static bool isSharedLibraryLoadedWindows(
                    std::string const & libname);

            static std::string getErrorLinux();
            static std::string getErrorWindows();

            static std::string getExtension();

            // Member
        private:
            std::string const cLibName;
            InstanceType mInstanceId;
            InstanceType const cMaxInstanceId;
            LibraryHandle mLibHandle;
    };
}

#endif
```

Listing C.9: SharedLibraryHelper.cc

```
/**
    *******************************************************************************
 \file   SharedLibrary.cc

 \brief  Implementation of operating system specific calls for shared library
 access

 *******************************************************************************
    */

/*-----------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 -----------------------------------------------------------------------------
    */

#include "SharedLibraryHelper.h"
#include <cstdio>
#include <iostream>

using namespace std;
using namespace interface;

#if defined(__linux__)
// Linux types
using UnknownStruct = struct unknown_struct
{
    void* pointers[3];
    struct unknown_struct* ptr;
};
using LinkMap = struct link_map;
#elif defined(_WIN32)
// Windows types
#endif

SharedLibraryHelper::SharedLibraryHelper(const std::string& libname) :
        SharedLibraryHelper(libname, 0)
{
}

SharedLibraryHelper::SharedLibraryHelper(const std::string& libname,
        InstanceType numberOfParallelInstances) :
        SharedLibraryHelper(libname, numberOfParallelInstances, 0)
{
}
```

```

SharedLibraryHelper::SharedLibraryHelper(std::string const & libname,
        InstanceType numberOfParallelInstances, InstanceType instanceId) :
        cLibName(libname), mInstanceId(instanceId), cMaxInstanceId(
                numberOfParallelInstances - 1)
{
    createLibraryInstance();

    // open shared library
    mLibHandle = SharedLibraryHelper::openSharedLibrary(getLibraryName());
}

SharedLibraryHelper::~SharedLibraryHelper()
{
    try
    {
        // close shared library
        SharedLibraryHelper::closeShareLibrary(mLibHandle);

        // check if the library was a created copy
        if (cMaxInstanceId > 0)
        {
            auto name = getLibraryName();
            // check if library is not loaded anymore (all other instances are
     already destroyed)
            if (!SharedLibraryHelper::isSharedLibraryLoaded(name))
            {
                std::cout << "Remove␣library␣" << name << std::endl;
                std::remove(name.c_str());
            }
        }
    }
    catch (exception const & e)
    {
    }
}

SharedLibraryHelper::HelperPtr SharedLibraryHelper::getNextLibrary()
{
    return getNextLibrary(cLibName);
}

SharedLibraryHelper::HelperPtr SharedLibraryHelper::getNextLibrary(
        std::string const & libname)
{
    if (mInstanceId + 1 <= cMaxInstanceId)
    {
        HelperPtr helper(
                new SharedLibraryHelper(libname, cMaxInstanceId + 1,
                        mInstanceId + 1));
        return helper;
    }
    throw runtime_error(
            "SharedLibraryHelper::getNextLibrary␣-␣maximum␣number␣of␣instances␣
     reached");
}

std::string SharedLibraryHelper::getLibraryName()
{
    return cLibName + ((cMaxInstanceId > 0) ? to_string(mInstanceId) : "")
            + getExtension();
}

void SharedLibraryHelper::createLibraryInstance()
{
    // check if multiple instances are used
    if (cMaxInstanceId > 0)
    {
        ifstream in(cLibName + getExtension(), ios::binary);
        if (!in)
            throw runtime_error(
                    "error␣copying␣library␣" + cLibName + getExtension()
                            + "␣with␣errno␣" + to_string(errno));

        auto copyName = getLibraryName();
        // check if copy already exisist
        {
            ifstream check(copyName);
            if (check.good())
                return;
        }
        ofstream out(copyName, ios::binary);
        if (!out)
            throw runtime_error("error␣opening␣copied␣library␣" + copyName);

        out << in.rdbuf();
    }
}

SharedLibraryHelper::HelperPtr SharedLibraryHelper::createInstance(
        const std::string& libname)
{
    return SharedLibraryHelper::createInstance(libname, 0);
}

SharedLibraryHelper::HelperPtr SharedLibraryHelper::createInstance(
        const std::string& libname, InstanceType numberOfParallelInstances)
{
    HelperPtr helper(
            new SharedLibraryHelper(libname, numberOfParallelInstances));
    return helper;
}

LibraryHandle SharedLibraryHelper::openSharedLibrary(const std::string& libname)
{
    // forward to OS specific method
#if defined(__linux__)
    return SharedLibraryHelper::openSharedLibraryLinux(libname);
#elif defined(_WIN32)
    return SharedLibraryHelper::openSharedLibraryWindows(libName);
#endif
}

void SharedLibraryHelper::closeShareLibrary(LibraryHandle handle)
{
    // forward to OS specific method
#if defined(__linux__)
    SharedLibraryHelper::closeShareLibraryLinux(handle);
```

```cpp
#elif defined(_WIN32)
    SharedLibraryHelper::closeShareLibraryWindows(handle);
#endif
}

std::string interface::SharedLibraryHelper::getError()
{
    // forward to OS specific method
#if defined(__linux__)
    return SharedLibraryHelper::getErrorLinux();
#elif defined(_WIN32)
    return SharedLibraryHelper::getErrorWindows();
#endif
}

LibraryHandle SharedLibraryHelper::openSharedLibraryLinux(
        const std::string& libname)
{
#if defined(__linux__)
    auto handle = dlopen(libname.c_str(), RTLD_NOW);
    if (handle == NULL)
        throw runtime_error(
                "SharedLibraryHelper::openSharedLibraryLinux␣-␣error␣loading␣library␣"
                        + libname + "␣with␣error␣" + getError());
    return handle;
#else
    throw runtime_error("error␣linux␣function␣called␣under␣different␣OS");
#endif
}

LibraryHandle SharedLibraryHelper::openSharedLibraryWindows(
        const std::string& libname)
{
#if defined(_WIN32)
    //TODO implement Windows version
    return nullptr;
#else
    throw runtime_error("error␣windows␣function␣called␣under␣different␣OS");
#endif
}

void SharedLibraryHelper::closeShareLibraryLinux(LibraryHandle handle)
{
#if defined(__linux__)
    dlclose(handle);
#else
    throw runtime_error("error␣linux␣function␣called␣under␣different␣OS");
#endif
}

void SharedLibraryHelper::closeShareLibraryWindows(LibraryHandle handle)
{
#if defined(_WIN32)
    //TODO implement Windows version
#else
    throw runtime_error("error␣windows␣function␣called␣under␣different␣OS");
#endif
}
```

```

std::string interface::SharedLibraryHelper::getErrorLinux()
{
#if defined(__linux__)
    return dlerror();
#else
    throw runtime_error("error␣linux␣function␣called␣under␣different␣OS");
#endif
}

std::string SharedLibraryHelper::getErrorWindows()
{
#if defined(_WIN32)
    //TODO implement Windows version
#else
    throw runtime_error("error␣windows␣function␣called␣under␣different␣OS");
#endif
}

std::string SharedLibraryHelper::getExtension()
{
#if defined(__linux__)
    return ".so";
#elif defined(_WIN32)
    return ".dll";
#endif
}

bool interface::SharedLibraryHelper::isSharedLibraryLoaded(
        const std::string& libname)
{
#if defined(__linux__)
    return SharedLibraryHelper::isSharedLibraryLoadedLinux(libname);
#elif defined(_WIN32)
    return SharedLibraryHelper::isSharedLibraryLoadedWindows(libname);
#endif
}

bool interface::SharedLibraryHelper::isSharedLibraryLoadedLinux(
        const std::string& libname)
{
#if defined(__linux__)
    auto* handle = dlopen(NULL, RTLD_NOW);
    auto* p = reinterpret_cast<UnknownStruct*>(handle)->ptr;
    auto* map = reinterpret_cast<LinkMap*>(p->ptr);

    for (; map; map = map->l_next)
    {
        if (libname.compare(map->l_name) == 0)
            return true;
    }

    return false;

#else
    throw runtime_error("error␣linux␣function␣called␣under␣different␣OS");
#endif
}

```

```
bool interface::SharedLibraryHelper::isSharedLibraryLoadedWindows(
        const std::string& libname)
{
#if defined(_WIN32)
    //TODO implement Windows version
#else
    throw runtime_error("error␣windows␣function␣called␣under␣different␣OS");
#endif
}
```

C.2.2 OplkBase class

Listing C.10: OplkBase.h

```
/**
    ********************************************************************************
 \file   OplkBase.h

 \brief  Include file with template base class for all interface clases

 ******************************************************************************
    */

/*------------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 ------------------------------------------------------------------------------
    */

#ifndef OPLKBASE_H_
#define OPLKBASE_H_

#include <omnetpp.h>
#include <vector>
#include <string>
#include "SharedLibraryHelper.h"
#include "oplkinc.h"

namespace interface
{
    /**
     * Template base class for each interface module.
     * This class provides handling and assiciation of shared library instances
     * with the according module instances.
     */
    template<typename TModule>
    class OplkBase
    {
            // Definitions
        public:
            /**
             * Structure with module instance and accroding shared library
    helper
             */
            struct ModuleInfo
            {
                    TModule module;
                    SharedLibraryHelper::HelperPtr helper;
            };

            using InfoCont = std::vector<ModuleInfo>;
            using InstanceHandle = SharedLibraryHelper::InstanceHandle;
            using Instance = SharedLibraryHelper::InstanceType;
            using ErrorType = OPLK::ErrorType;

            // C-Tor / D-Tor
        protected:
            /**
             * \brief Protected default constructor
             */
            OplkBase()
            {
            }

        public:
            virtual ~OplkBase()
            {
            }

            // Methods
        public:
            /**
             * \brief Initialize interface functionality with given module.
             *
             * Initializes the interface functionality saving the given module
             * with a newly created shared library helper
             *
             * \param module    Instance of module which is saved in internal
             * container
             */
            void initModule(TModule module)
            {
                if (module == nullptr)
                    throw std::invalid_argument(
                            "OplkBase::initModule␣-␣invalid␣module␣pointer");

                // create info object with correct helper instance
                SharedLibraryHelper::HelperPtr helper;

                if (mModuleInfos.empty())
                    helper = SharedLibraryHelper::createInstance(mLibName,
                            mNumberOfInstances);
                else
                    helper = mModuleInfos.back().helper->getNextLibrary(
                            mLibName);

                // add info object
                ModuleInfo info = { module, helper };
                mModuleInfos.push_back(info);

                // set function pointer of interface
                try
                {
                    setFunctions(helper, mModuleInfos.size() - 1);
                }
                catch (std::exception const & e)
                {
```

```
                // remove last module info
                mModuleInfos.pop_back();

                // rethrow exception
                throw e;
            }
        }

        /**
         * \brief Getter for the saved module instance
         *
         * This getter returns the saved module instance identified by the
         * passed handle.
         *
         * \param handle    Handle of requested module
         * \return Returns the saved instance according to the given handle.
         * \throws Throws a std::out_of_range exception when the handle is
         * invalid
         */
        TModule getModule(InstanceHandle handle)
        {
            if (handle < mModuleInfos.size())
                return mModuleInfos[handle].module;
            else
                throw std::out_of_range(
                        "OplkBase::getModule␣-␣invalid␣instance␣handle");
        }

        virtual void setFunctions(SharedLibraryHelper::HelperPtr helper,
                InstanceHandle handle) = 0;

        // static Methods
    public:
        /**
         * \brief Static setter for libary information
         *
         * Setter for static library informations, which will be passed to
         * the created shared library helper
         */
        static void setLibraryInfo(std::string const & libName,
                Instance numberOfInstances)
        {
            mLibName = libName;
            mNumberOfInstances = numberOfInstances;
        }

        // Member
    protected:
        InfoCont mModuleInfos;

        // static member
    private:
        static std::string mLibName;
        static Instance mNumberOfInstances;
};

template<typename TModule>
std::string OplkBase<TModule>::mLibName = "";
```

```
    template<typename TModule>
    typename OplkBase<TModule>::Instance OplkBase<TModule>::mNumberOfInstances =
     1;

} /* namespace interface */

#endif
```

C.2.3 OplkEdrv

Listing C.11: OplkEdrv.h

```
/*
 * OplkEdrv.h
 *
 *  Created on: Apr 28, 2016
 *      Author: franz
 */

#ifndef OPLKEDRV_H_
#define OPLKEDRV_H_

#include <OplkBase.h>

// forward declaration
class DirectEdrv;

namespace interface
{

    class OplkEdrv : public OplkBase<DirectEdrv*>
    {
            // Definitions
        public:
            using EdrvInitParamType = OPLK::tEdrvInitParam;
            using MacType = UINT8*;
            using TxBufferType = OPLK::tEdrvTxBuffer;
            using RxBufferType = OPLK::tEdrvRxBuffer;
            using FilterType = OPLK::tEdrvFilter;
            using RxCallbackType = OPLK::tEdrvRxHandler;
            using HwInfoType = OPLK::tHwParam;

            // C-Tor / D-Tor
        private:
            OplkEdrv();
        public:
            virtual ~OplkEdrv();

            // Methods
        public:
            virtual void setFunctions(SharedLibraryHelper::HelperPtr helper,
     InstanceHandle handle) override;

            // Static Methods
        public:
            static OplkEdrv& getInstance();

            static ErrorType initEdrv(InstanceHandle handle, EdrvInitParamType*
     initParam);
```

```
            static ErrorType exitEdrv(InstanceHandle handle);
            static MacType getMacAddr(InstanceHandle handle);
            static ErrorType sendTxBuffer(InstanceHandle handle, TxBufferType* txBuffer);
            static ErrorType allocTxBuffer(InstanceHandle handle, TxBufferType* txBuffer);
            static ErrorType freeTxBuffer(InstanceHandle handle, TxBufferType* txBuffer);
            static ErrorType changeRxFilter(InstanceHandle handle, FilterType* filter, UINT count, UINT entryChanged, UINT changeFlags);
            static ErrorType clearRxMulticastMacAddr(InstanceHandle handle, MacType macAddr);
            static ErrorType setRxMulticastMacAddr(InstanceHandle handle, MacType macAddr);

    };

} /* namespace interface */

#endif
```

Listing C.12: OplkEdrv.cc

```
/*
 * OplkEdrv.cc
 *
 *  Created on: Apr 28, 2016
 *      Author: franz
 */

#include <OplkEdrv.h>
#include "OplkException.h"
#include "DirectEdrv.h"

namespace interface
{

    OplkEdrv::OplkEdrv()
    {
    }

    OplkEdrv::~OplkEdrv()
    {
    }

    void OplkEdrv::setFunctions(SharedLibraryHelper::HelperPtr helper, InstanceHandle handle)
    {
        OPLK::tEdrvFunctions functions;
        functions.pfnInit = OplkEdrv::initEdrv;
        functions.pfnExit = OplkEdrv::exitEdrv;
        functions.pfnGetMacAddr = OplkEdrv::getMacAddr;
        functions.pfnSendTxBuffer = OplkEdrv::sendTxBuffer;
        functions.pfnAllocTxBuffer = OplkEdrv::allocTxBuffer;
        functions.pfnFreeTxBuffer = OplkEdrv::freeTxBuffer;
        functions.pfnChangeRxBufferFiler = OplkEdrv::changeRxFilter;
        functions.pfnSetMulticastMacAddr = OplkEdrv::setRxMulticastMacAddr;
        functions.pfnClearMulticastMacAddr = OplkEdrv::clearRxMulticastMacAddr;

        // set function pointer of interface
        auto setFunctions = helper->getFunction<OPLK::BoolType, InstanceHandle, OPLK::tEdrvFunctions>("sim_setEdrvFunctions");

        if (setFunctions == nullptr)
            throw std::runtime_error("OplkEdrv::setFunctions␣-␣unable␣to␣resolve␣setFunctions␣function");

        auto ret = setFunctions(handle,functions);

        if (ret != TRUE)
            throw std::runtime_error("OplkEdrv::setFunctions␣-␣unable␣to␣set␣function␣pointer");
    }

    OplkEdrv& OplkEdrv::getInstance()
    {
        static OplkEdrv oplkEdrv;

        return oplkEdrv;
    }

    OplkEdrv::ErrorType OplkEdrv::initEdrv(InstanceHandle handle, EdrvInitParamType* initParam)
    {
        if (initParam == nullptr)
            return OPLK::kErrorApiInvalidParam;

        try
        {
            OplkEdrv::getInstance().getModule(handle)->initEdrv(initParam);
        }
        catch (OplkException const & e)
        {
            return e.errorNumber();
        }
        catch (std::exception const &)
        {
            return OPLK::kErrorGeneralError;
        }

        return OPLK::kErrorOk;
    }

    OplkEdrv::ErrorType OplkEdrv::exitEdrv(InstanceHandle handle)
    {
        try
        {
            OplkEdrv::getInstance().getModule(handle)->exitEdrv();
        }
        catch (OplkException const & e)
        {
            return e.errorNumber();
        }
        catch (std::exception const &)
        {
            return OPLK::kErrorGeneralError;
        }

        return OPLK::kErrorOk;
    }

    OplkEdrv::MacType OplkEdrv::getMacAddr(InstanceHandle handle)
    {
        try
        {
            return OplkEdrv::getInstance().getModule(handle)->getMacAddr();
        }
        catch (std::exception const &)
        {
            return nullptr;
        }
    }

    OplkEdrv::ErrorType OplkEdrv::sendTxBuffer(InstanceHandle handle,
     TxBufferType* txBuffer)
    {
        if (txBuffer == nullptr)
            return OPLK::kErrorEdrvInvalidParam;

        try
        {
            OplkEdrv::getInstance().getModule(handle)->sendTxBuffer(txBuffer);
        }
        catch (OplkException const & e)
        {
            return e.errorNumber();
        }
        catch (std::exception const &)
        {
            return OPLK::kErrorGeneralError;
        }

        return OPLK::kErrorOk;
    }

    OplkEdrv::ErrorType OplkEdrv::allocTxBuffer(InstanceHandle handle,
     TxBufferType* txBuffer)
    {
        if (txBuffer == nullptr)
            return OPLK::kErrorEdrvInvalidParam;

        try
        {
            OplkEdrv::getInstance().getModule(handle)->allocTxBuffer(txBuffer);
        }
        catch (OplkException const & e)
        {
            return e.errorNumber();
        }
        catch (std::exception const &)
        {
            return OPLK::kErrorGeneralError;
        }

        return OPLK::kErrorOk;
    }

```

```
OplkEdrv::ErrorType OplkEdrv::freeTxBuffer(InstanceHandle handle, TxBufferType* txBuffer)
{
    if (txBuffer == nullptr)
        return OPLK::kErrorEdrvInvalidParam;

    try
    {
        OplkEdrv::getInstance().getModule(handle)->freeTxBuffer(txBuffer);
    }
    catch (OplkException const & e)
    {
        return e.errorNumber();
    }
    catch (std::exception const &)
    {
        return OPLK::kErrorGeneralError;
    }

    return OPLK::kErrorOk;
}

OplkEdrv::ErrorType OplkEdrv::changeRxFilter(InstanceHandle handle, FilterType* filter, unsigned int count,
        unsigned int entryChanged, unsigned int changeFlags)
{
    if (filter == nullptr)
        return OPLK::kErrorEdrvInvalidParam;

    try
    {
        OplkEdrv::getInstance().getModule(handle)->changeRxFilter(filter, count, entryChanged, changeFlags);
    }
    catch (OplkException const & e)
    {
        return e.errorNumber();
    }
    catch (std::exception const &)
    {
        return OPLK::kErrorGeneralError;
    }

    return OPLK::kErrorOk;
}

OplkEdrv::ErrorType OplkEdrv::clearRxMulticastMacAddr(InstanceHandle handle, MacType macAddr)
{
    try
    {
        OplkEdrv::getInstance().getModule(handle)->clearRxMulticastMacAddr(macAddr);
    }
    catch (OplkException const & e)
    {
        return e.errorNumber();
    }
    catch (std::exception const &)
```

```
        {
            return OPLK::kErrorGeneralError;
        }

        return OPLK::kErrorOk;
    }

    OplkEdrv::ErrorType OplkEdrv::setRxMulticastMacAddr(InstanceHandle handle,
     MacType macAddr)
    {
        try
        {
            OplkEdrv::getInstance().getModule(handle)->setRxMulticastMacAddr(
     macAddr);
        }
        catch (OplkException const & e)
        {
            return e.errorNumber();
        }
        catch (std::exception const &)
        {
            return OPLK::kErrorGeneralError;
        }

        return OPLK::kErrorOk;
    }

} /* namespace interface */
```

C.3 OMNeT++ modules

C.3.1 DirectEdrv

Listing C.13: DirectEdrv.h

```
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program. If not, see http://www.gnu.org/licenses/.
//

#ifndef __OPENPOWERLINK_EDRV_H_
#define __OPENPOWERLINK_EDRV_H_

#include <array>
#include <vector>
#include <memory>
#include <omnetpp.h>
#include "interface/OplkEdrv.h"
```

```

class DirectEdrv : public OPP::cSimpleModule
{
        // Definitions
    public:
        using EdrvInitParamType = interface::OplkEdrv::EdrvInitParamType;
        using MacType = interface::OplkEdrv::MacType;
        using TxBufferType = interface::OplkEdrv::TxBufferType;
        using FilterType = interface::OplkEdrv::FilterType;
        using MacCont = std::array<UINT8, 6>;
        using RxCallbackType = interface::OplkEdrv::RxCallbackType;
        using HwInfoType = interface::OplkEdrv::HwInfoType;
        using RxBuffer = interface::OplkEdrv::RxBufferType;
        using RxBufferPtr = std::shared_ptr<RxBuffer>;
        using RxBufferCont = std::vector<RxBufferPtr>;
        using Timestamp = unsigned int;
        using ReleaseRxBuffer = OPLK::eEdrvReleaseRxBuffer;

        // Methods
    protected:
        virtual void initialize();
        virtual void handleMessage(OPP::cMessage *rawMsg);

    public:
        void initEdrv(EdrvInitParamType* initParam);
        void exitEdrv();
        MacType getMacAddr();
        void sendTxBuffer(TxBufferType* txBuffer);
        void allocTxBuffer(TxBufferType* txBuffer);
        void freeTxBuffer(TxBufferType* txBuffer);
        void changeRxFilter(FilterType* filter, UINT count, UINT entryChanged,
    UINT changeFlags);
        void clearRxMulticastMacAddr(MacType macAddr);
        void setRxMulticastMacAddr(MacType macAddr);

    private:
        void refreshDisplay();
        void setInitialized(bool initialized);
        Timestamp* getCurrentTimestamp();

        // static Methods
    public:
        static void deleteRxBuffer(RxBuffer* rxBuffer);

        // Member
    private:
        MacCont mMac;
        RxCallbackType mRxCallback;
        HwInfoType mHwInfo;
        bool mInitialized;
        OPP::cGate* mEthernetOutGate;
        OPP::cGate* mEthernetInGate;
        RxBufferCont mRxBufferCont;

        OPP::simsignal_t mSentEtherTypeSignal;
        OPP::simsignal_t mReceivedEtherTypeSignal;
        OPP::simsignal_t mSentOplkTypeSignal;
        OPP::simsignal_t mReceivedOplkTypeSignal;

        size_t const cEtherTypePos = 12;
        size_t const cOplkTypePos = 14;
        unsigned short cOplKEtherType = 0xAB88;
};

#endif
```

Listing C.14: DirectEdrv.cc

```
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program. If not, see http://www.gnu.org/licenses/.
//

#include <sstream>
#include "DirectEdrv.h"
#include "interface/OplkException.h"
#include "interface/OplkEdrv.h"
#include "BufferMessage_m.h"
#include "MsgPtr.h"
#include "ApiDef.h"

using namespace std;
USING_NAMESPACE

Define_Module(DirectEdrv);

void DirectEdrv::initialize()
{
    // resolve library info parameter
    std::string libName = par("libName");
    interface::OplkEdrv::Instance numberOfInstances = par("numberOfInstances");

    // init stub
    interface::OplkEdrv::setLibraryInfo(libName, numberOfInstances);
    interface::OplkEdrv::getInstance().initModule(this);

    // resolve gates
    mEthernetOutGate = gate("ethernetOut");

    // resolve signals
    mSentEtherTypeSignal = registerSignal("sentEtherType");
    mReceivedEtherTypeSignal = registerSignal("receivedEtherType");
    mSentOplkTypeSignal = registerSignal("sentOplkMessageType");
    mReceivedOplkTypeSignal = registerSignal("receivedOplkMessageType");
}

void DirectEdrv::handleMessage(cMessage *rawMsg)
{
    MsgPtr msg(rawMsg);

    if ((msg != nullptr) && mInitialized)
    {
        // cast message
        auto buffMsg = dynamic_cast<oplkMessages::BufferMessage*>(msg.get());

        if (buffMsg != nullptr)
        {
            bubble("Frame␣received");

            // create rx buffer
            RxBufferPtr rxBuffer(new RxBuffer(), deleteRxBuffer);
            rxBuffer->rxFrameSize = buffMsg->getBufferArraySize();
            rxBuffer->pBuffer = new UINT8[rxBuffer->rxFrameSize];
            for (auto i = 0u; i < rxBuffer->rxFrameSize; i++)
                rxBuffer->pBuffer[i] = buffMsg->getBuffer(i);
            rxBuffer->pRxTimeStamp = nullptr; //getCurrentTimestamp();

            // emit signals
            auto etherType = *((unsigned short*)(&rxBuffer->pBuffer[
 cEtherTypePos]));
            emit(mReceivedEtherTypeSignal, etherType);
            if (etherType == cOplKEtherType)
                emit(mReceivedOplkTypeSignal, rxBuffer->pBuffer[cOplkTypePos]);

            bubble("Frame␣received");

            // forward received frame
            auto ret = mRxCallback(rxBuffer.get());

            // check return
            if (ret == ReleaseRxBuffer::kEdrvReleaseRxBufferLater)
            {
                // add rx buffer ptr to internal cont
                mRxBufferCont.push_back(rxBuffer);
            }
        }
    }
}

void DirectEdrv::initEdrv(EdrvInitParamType* initParam)
{
    // check parameter
    if (initParam == nullptr)
        throw interface::OplkException("invalid␣init␣param", interface::api::
 Error::kErrorEdrvInvalidParam);

    bubble("Ethernet␣driver␣initialized");
    // init member

    // copy mac addr
    mMac[0] = initParam->aMacAddr[0];

    for (auto i = 1u; i < mMac.size(); i++)
    {
        mMac[i] = initParam->aMacAddr[i];
    }

    // set rx handler
    mRxCallback = initParam->pfnRxHandler;

    // save hw info
    mHwInfo = initParam->hwParam;

    setInitialized(true);
    refreshDisplay();
}

void DirectEdrv::exitEdrv()
{
    setInitialized(false);
}

DirectEdrv::MacType DirectEdrv::getMacAddr()
{
    return mMac.data();
}

void DirectEdrv::sendTxBuffer(TxBufferType* txBuffer)
{
    // check parameter
    if (txBuffer == nullptr)
        throw interface::OplkException("invalid␣tx␣buffer", interface::api::
     Error::kErrorEdrvInvalidParam);

    bubble("Send␣Tx␣given␣buffer");

    // save simulation ctx
    auto oldCtx = simulation.getContext();
    simulation.setContext(this);

    // create buffer message
    auto msg = new oplkMessages::BufferMessage();

    // fill message
    msg->setBufferArraySize(txBuffer->txFrameSize);
    for (auto i = 0u; i < txBuffer->txFrameSize; i++)
        msg->setBuffer(i, txBuffer->pBuffer[i]);

    // emit signals
    auto etherType = *((unsigned short*)(&txBuffer->pBuffer[cEtherTypePos]));
    emit(mSentEtherTypeSignal, etherType);
    if (etherType == cOplKEtherType)
        emit(mSentOplkTypeSignal, txBuffer->pBuffer[cOplkTypePos]);

    // send message
    send(msg, mEthernetOutGate);

    // restore ctx
    simulation.setContext(oldCtx);

    // call tx handler
    if (txBuffer->pfnTxHandler != nullptr)
        txBuffer->pfnTxHandler(txBuffer);
}

void DirectEdrv::allocTxBuffer(TxBufferType* txBuffer)
```

```
{
    // check parameter
    if (txBuffer == nullptr)
        throw interface::OplkException("invalid␣tx␣buffer", interface::api::Error::kErrorEdrvInvalidParam);

    // allocate buffer with new
    txBuffer->pBuffer = new UINT8[txBuffer->maxBufferSize];
    if (txBuffer->pBuffer == nullptr)
        throw interface::OplkException("error␣allocating␣tx␣buffer␣", OPLK::kErrorEdrvNoFreeBufEntry);

    txBuffer->txBufferNumber.pArg = nullptr;
}

void DirectEdrv::freeTxBuffer(TxBufferType* txBuffer)
{
    if (txBuffer->pBuffer != nullptr)
    {
        delete[] txBuffer->pBuffer;
        txBuffer->pBuffer = nullptr;
    }
}

void DirectEdrv::changeRxFilter(FilterType* filter, unsigned int count, unsigned int entryChanged, unsigned int changeFlags)
{
    // check parameter
    if (filter == nullptr)
        throw interface::OplkException("invalid␣filter", interface::api::Error::kErrorEdrvInvalidParam);

    bubble("change␣filter");
}

void DirectEdrv::clearRxMulticastMacAddr(MacType macAddr)
{
    bubble("clear␣rx␣multicast␣mac␣addr");
}

void DirectEdrv::setRxMulticastMacAddr(MacType macAddr)
{
    bubble("set␣rx␣multicast␣mac␣addr");
}

void DirectEdrv::refreshDisplay()
{
    if (mInitialized)
    {
        stringstream strStream;
        strStream << "␣MacAddress:␣" << std::hex << (int) mMac[0];
        for (auto i = 1u; i < mMac.size(); i++)
            strStream << ":" << std::hex << (int) mMac[i];
        strStream << std::endl;
        strStream << "␣Device␣Number:␣" << mHwInfo.devNum << std::endl;
        strStream << "␣Device␣Name:␣" << mHwInfo.pDevName << std::endl;

        getDisplayString().setTagArg("t", 0, strStream.str().c_str());
        getDisplayString().setTagArg("i", 1, "green");
    }
    else
    {
        getDisplayString().setTagArg("t", 0, "Uninitialized");
        getDisplayString().setTagArg("i", 1, "red");
    }
}

void DirectEdrv::setInitialized(bool initialized)
{
    mInitialized = initialized;

    refreshDisplay();
}

DirectEdrv::Timestamp* DirectEdrv::getCurrentTimestamp()
{
    return new Timestamp(simTime().inUnit(SimTimeUnit::SIMTIME_NS));
}

void DirectEdrv::deleteRxBuffer(RxBuffer* rxBuffer)
{
    if (rxBuffer != nullptr)
    {
        if (rxBuffer->pBuffer != nullptr)
        {
            delete[] rxBuffer->pBuffer;
            rxBuffer->pBuffer = nullptr;
        }
        if (rxBuffer->pRxTimeStamp != nullptr)
        {
            delete rxBuffer->pRxTimeStamp;
            rxBuffer->pRxTimeStamp = nullptr;
        }
        delete rxBuffer;
    }
}
```

C.3.2 GenericNode

Listing C.15: GenericNode.ned

```
//
// @author Franz Profelt
//
// Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
//

package openpowerlink.generic;

import openpowerlink.generic.stack.Stack;

//
// Compound module representing a base class for every specialized openPOWERLINK
// node. Connecting the interfaces IDemo, IApp, IEvent with the Stack module.
//
module GenericNode
{
    parameters:
        @display("bgb=282,300");
        int numberOfInstances = default(1); // Number of instances
        string libName; // Name of the used shared library

        string DemoType; // Typename of the demo instance implementing IDemo
        string AppType; // Typename of the app instance implementing IApp
        string EventType; // Typename of the event instance implementing IEvent

        double mainLoopInterval @unit(ns) = default(1ms); // Interval of the
     main loop

    gates:
        input ethernetIn; // Input gate for ethernet connection
        output ethernetOut; // Output gate for ethernet connection
        input udpIn; // Input gate for UDP connetion
        output udpOut; // Output gate for UDP connection

    submodules:
        stack: Stack {
            @display("p=138,151");
            libName = libName;
            numberOfInstances = numberOfInstances;
        }
        event: <EventType> like IEvent {
            @display("p=217,51");
        }
        app: <AppType> like IApp {
            @display("p=52,51");
        }
        demo: <DemoType> like IDemo {
            @display("p=138,51");
            mainLoopInterval = mainLoopInterval;
        }
        nmtState: NmtState {
            @display("p=138,235");
        }
        status: Led {
            @display("p=197,151");
            signalName = "statusLed";
            activeColor = "green";
        }
        error: Led {
            @display("p=238,151");
            signalName = "errorLed";
            activeColor = "red";
        }
    connections:
    // Api connections
        demo.apiCall --> stack.apiCall++;
        stack.apiReturn++ --> demo.apiReturn;
        app.apiCall --> stack.apiCall++;
        stack.apiReturn++ --> app.apiReturn;

        // Process sync
        stack.processSync --> app.functionCall++;
        app.functionReturn++ --> stack.processSyncReturn;

        // Event
        stack.event --> event.event;
        event.eventReturn --> stack.eventReturn;
        event.stackShutdown --> demo.stackShutdown;

        // App call
        demo.appCall --> app.functionCall++;
        app.functionReturn++ --> demo.appReturn;

        // Network connections
        ethernetIn --> stack.ethernetIn;
        stack.ethernetOut --> ethernetOut;
        stack.udpOut --> udpOut;
        udpIn --> stack.udpIn;
}
```

C.3.3 Moduleinterfaces

Listing C.16: IDemo.ned

```
//
// @author Franz Profelt
//
// Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
//

package openpowerlink.generic;

//
// Interface for Demo modules used within a openPOWERLINK node derived from
    GenericNode
//
moduleinterface IDemo
{
    parameters:
        double startUpDelay @unit(ns); // Delay of simulation time after the
    demo should start initializing
        double mainLoopInterval @unit(ns);  // Interval of the main processing
    loop
        double shutdownTime @unit(ns); // Simulation time after the shutdown
    procedure should be initiated

    gates:
        output apiCall; // Output gate for Api calls
        input apiReturn; // Input gate for Api return messages
        output appCall; // Output gate for App calls
        input appReturn; // Input gate for App return messages
        input stackShutdown; // Input gate for shutdown message
}
```

Listing C.17: IEvent.ned

```
//
// @author Franz Profelt
//
// Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
//

package openpowerlink.generic;

//
```

```
// Interface for Event modules used within a openPOWERLINK node derived from
    GenericNode
//
moduleinterface IEvent
{
    gates:
        input event; // Input gate for occuring events
        output eventReturn;  // Output gate for return values from event
    handling
        output stackShutdown; // Output gate for occuring shutdown event
}
```

Listing C.18: IApp.ned

```
//
// @author Franz Profelt
//
// Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
//

package openpowerlink.generic;

//
// Interface for App modules used within a openPOWERLINK node derived from
    GenericNode
//
moduleinterface IApp
{
    gates:
        input functionCall[]; // Input vector for each connection using App
    functions
        output functionReturn[]; // Output vector for return values of incomming
    app functions
        output apiCall; // Output gate for Api calls
        input apiReturn; // Input gate for Api return messages
}
```

C.3.4 Base classes

Listing C.19: DemoBase.h

```
/**
    ********************************************************************************

 \file   DemoBase.h

 \brief  Include file for Base class of Demo modules.

 ******************************************************************************
    */

/*------------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 ------------------------------------------------------------------------------
    */

#ifndef __OPENPOWERLINK_DEMOBASE_H_
#define __OPENPOWERLINK_DEMOBASE_H_
```

```cpp

#include <omnetpp.h>
#include "UseApiBase.h"
#include "ReturnMessage_m.h"
#include "MessageDispatcher.h"

/**
 * \brief Base class for Demo modules implementing basic functionalities.
 *
 * This class inherits from UseApiBase and represents a cSimpleModule which
 * sends messages to the Api module and blocks the according method calls
 * until reception of the return message.
 * Additionally this class implements the handling of various states and the
 * communication to IApp and IEvent modules.
 */
class DemoBase : public UseApiBase
{
        // Definitions
    private:
        using RawMessagePtr = MessageDispatcher::RawMessagePtr;

        enum class DemoState : short
        {
            undefined,
            initializePowerlink,
            initializeApp,
            swReset,
            mainloop,
            shuttingDownApp,
            shuttingDown
        };

        // C-Tor
    public:
        /**
         * \brief Default constructor initializing the base class with the
         * name of the Api sending gate
         */
        DemoBase();

        // Methods
    protected:
        virtual void initialize();
        virtual void handleOtherMessage(MessagePtr msg) override;
        virtual void handleSelfMessage(MessagePtr msg);

        /**
         * \brief Initialize the openPOWERLINK stack
         *
         * This pure virtual method is called during initialization state.
         */
        virtual void initPowerlink() = 0;

        /**
         * \brief Shutdown the openPOWERLINK stack
         *
         * This pure virtual method is called during shutdown state.
         */
        virtual void shutdownPowerlink() = 0;

    private:
        void processAppReturn(RawMessagePtr msg);
        void processStackShutdown(RawMessagePtr msg);

        // Member
    protected:
        OPP
        ::cGate* mApiCallGate;OPP
        ::cGate* mAppCallGate;

        MessageDispatcher mDispatcher;

        simtime_t mStartUpDelay;
        simtime_t mMainLoopInterval;

        simtime_t mShutdownTime;

        DemoState mState;
};

#endif
```

Listing C.20: DemoBase.cc

```
/**
    ********************************************************************************
 \file   DemoBase.cc

 \brief  Implementation of Base class of Demo modules.

 ******************************************************************************
    */

/*------------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 ------------------------------------------------------------------------------
    */

#include <functional>
#include <string>
#include "DemoBase.h"
#include "MsgPtr.h"
#include "stack/Api.h"
#include "InitMessage_m.h"
#include "StringMessage_m.h"
#include "AppBase.h"
#include "OplkException.h"

using namespace std;
USING_NAMESPACE

DemoBase::DemoBase() :
        UseApiBase("apiCall")
{
}

```

```cpp
void DemoBase::initialize()
{
    UseApiBase::initialize();

    // resovle parameters
    mStartUpDelay = simtime_t(par("startUpDelay").doubleValue(), SimTimeUnit::SIMTIME_NS);
    mMainLoopInterval = simtime_t(par("mainLoopInterval").doubleValue(), SimTimeUnit::SIMTIME_NS);
    mShutdownTime = simtime_t(par("shutdownTime").doubleValue(), SimTimeUnit::SIMTIME_NS);

    // resolve gates
    mApiCallGate = gate("apiCall");
    mAppCallGate = gate("appCall");

    // init dispatcher
    mDispatcher.registerFunction(gate("appReturn"), std::bind(&DemoBase::processAppReturn, this, placeholders::_1));
    mDispatcher.registerFunction(gate("stackShutdown"),
            std::bind(&DemoBase::processStackShutdown, this, placeholders::_1));

    // init state
    mState = DemoState::initializePowerlink;

    // schedule init message
    scheduleAt(simTime() + mStartUpDelay, new cMessage(("Init␣demo␣-␣" + string(getParentModule()->getName())).c_str()));
}

void DemoBase::handleOtherMessage(MessagePtr msg)
{
    if (msg != nullptr)
    {
        // check if external message
        if (!msg->isSelfMessage())
        {
            // process message according to arrival gate via dispatcher
            mDispatcher.dispatch(msg.get());
        }
        else // handle self messages
        {
            handleSelfMessage(msg);
        }
    }
}

void DemoBase::handleSelfMessage(MessagePtr msg)
{
    // check sim time
    if (simTime() > mShutdownTime)
    {
        mState = DemoState::shuttingDown;
    }

    // check current state
    switch (mState)
    {
        case DemoState::initializePowerlink: {
            // init powerlink stack by derived class
            initPowerlink();

            // create and send init message for app module
            send(new cMessage("initialize␣app", static_cast<short>(AppBase::AppBaseCallType::init)), mAppCallGate);

            // advance to next init state
            mState = DemoState::initializeApp;
            break;
        }
        case DemoState::initializeApp:
            EV << "Unexpected␣message␣(" << *msg << ")␣received␣within␣initializeApp␣state" << endl;
            break;

        case DemoState::swReset: {
            // perform sw reset
            execNmtCommand(interface::api::NmtEventType::kNmtEventSwReset);

            // advance to mainloop state
            mState = DemoState::mainloop;

            // send self message
            scheduleAt(simTime() + mMainLoopInterval, new cMessage("first␣demo␣main␣loop␣message"));

            EV << "App␣initialized␣advancing␣to␣main␣loop␣state" << endl;
            break;
        }

        case DemoState::mainloop: {
            // process stack
            stackProcess();
            // schedule following main message
            scheduleAt(simTime() + mMainLoopInterval, new cMessage("demo␣main␣loop␣message"));
            break;
        }
        case DemoState::shuttingDown: {
            // shutdown powerlink
            shutdownPowerlink();

            EV << "Stack␣shutdown␣succesfully" << endl;
            mRunning = false;
            break;
        }

        case DemoState::shuttingDownApp:
            // shutdown app
            send(new cMessage("shutdown␣app", static_cast<short>(AppBase::AppBaseCallType::shutdown)), mAppCallGate);
            break;

        default:
            error("DemoBase␣-␣invalid␣demo␣state:␣%d", static_cast<short>(mState));
    }
}
```

```

void DemoBase::processAppReturn(RawMessagePtr msg)
{
    auto retMsg = dynamic_cast<oplkMessages::ReturnMessage*>(msg);
    if (retMsg != nullptr)
    {
        auto ret = retMsg->getReturnValue();

        // check if error
        if (ret != interface::api::Error::kErrorOk)
            throw interface::OplkException("Error␣in␣App␣call␣ocurred", ret);

        // check if initialization returned
        auto kind = static_cast<AppBase::AppBaseCallType>(retMsg->getKind());
        switch (kind)
        {
            case AppBase::AppBaseCallType::init: {
                // advance to sw reset state
                mState = DemoState::swReset;

                // schedule immediate self message
                scheduleAt(simTime(), new cMessage());
                break;
            }
            case AppBase::AppBaseCallType::shutdown: {
                // advance to shutdown state
                mState = DemoState::shuttingDown;

                // schedule immediate self message
                scheduleAt(simTime(), new cMessage());
                break;
            }
            default:
                error("HandleAppReturn␣-␣invalid␣message␣kind:␣%d", static_cast<
    short>(kind));
        }
    }
}

void DemoBase::processStackShutdown(RawMessagePtr msg)
{
    // advance to shutting down state
    mState = DemoState::shuttingDownApp;
}
```

Listing C.21: EventBase.h

```
/**
    ********************************************************************************
 \file   EventBase.h

 \brief  Include file for Base class of Event modules.

 ******************************************************************************
    */

/*------------------------------------------------------------------------------
```

```
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 -------------------------------------------------------------------------------
    */

#ifndef __OPENPOWERLINK_EVENTBASE_H_
#define __OPENPOWERLINK_EVENTBASE_H_

#include <omnetpp.h>
#include "ApiDef.h"

/**
 * \brief Base class for Event modules providing basic functionalities
 *
 * This class inherits form cSimpleModule and is implemented event based.
 * Dispatching to various virtual methods and output prints are implemented
 * within this class, which can either be subclassed or used directly.
 * A shutdown event results in an according message sent to the Demo module.
 */
class EventBase : public OPP::cSimpleModule
{
        // Methods
  protected:
    virtual void initialize();
    virtual void handleMessage(OPP::cMessage *msg);

    /**
     * \brief General Method for processing a received event
     *
     * This method dispatches the events according to its type to the different
     * methods.
     * Additionally this method emits the received event type as signal.
     */
    virtual interface::api::ErrorType processEvent(interface::api::ApiEventType
     eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing an Nmt state change event
     */
    virtual interface::api::ErrorType processNmtStateChangeEvent(interface::api
     ::ApiEventType eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing an error event
     */
    virtual interface::api::ErrorType processErrorEvent(interface::api::
     ApiEventType eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing a warning event
     *
     */
    virtual interface::api::ErrorType processWarningEvent(interface::api::
     ApiEventType eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing a history event
     */
    virtual interface::api::ErrorType processHistoryEvent(interface::api::
     ApiEventType eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing a node event
     */
    virtual interface::api::ErrorType processNodeEvent(interface::api::
     ApiEventType eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing a PDO change event
     */
    virtual interface::api::ErrorType processPdoChangeEvent(interface::api::
     ApiEventType eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing a Cfm progress event
     */
    virtual interface::api::ErrorType processCfmProgressEvent(interface::api::
     ApiEventType eventType, interface::api::ApiEventArg eventArg);

    /**
     * \brief Virtual method for processing a Cfm result event
     */
    virtual interface::api::ErrorType processCfmResultEvent(interface::api::
     ApiEventType eventType, interface::api::ApiEventArg eventArg);

    // Member
  protected:
    OPP::cGate* mReturnGate;
    OPP::cGate* mShutdownGate;
    OPP::simsignal_t mEventTypeSignal;
    OPP::simsignal_t mNmtStateSignal;
    bool mPrintEventType;
};

#endif
```

Listing C.22: EventBase.cc

```
/**
    ********************************************************************************
 \file   EventBase.cc

 \brief  Implementation of Base class of Event modules.

 ******************************************************************************
    */

/*------------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 ------------------------------------------------------------------------------
    */

#include "EventBase.h"
#include "ApiMessages.h"
#include "MsgPtr.h"
#include "debugstr.h"
#include "OplkException.h"
```

```

using namespace std;
USING_NAMESPACE

Define_Module(EventBase);

void EventBase::initialize()
{
    // resolve gates
    mReturnGate = gate("eventReturn");
    mShutdownGate = gate("stackShutdown");

    // register signals
    mEventTypeSignal = registerSignal("eventType");
    mNmtStateSignal = registerSignal("nmtState");

    // resolve parameter
    mPrintEventType = par("printEventType");
}

void EventBase::handleMessage(cMessage *rawMsg)
{
    MsgPtr msg(rawMsg);

    if (msg != nullptr)
    {
        // cast to EventMessage
        auto eventMsg = dynamic_cast<oplkMessages::EventMessage*>(msg.get());

        if (eventMsg != nullptr)
        {
            // process Event
            auto ret = processEvent(eventMsg->getEventType(), eventMsg->getEventArg());

            // create event return message
            auto retMsg = new oplkMessages::EventReturnMessage();
            retMsg->setName(("return␣-␣" + std::string(msg->getName())).c_str());
            retMsg->setReturnValue(ret);
            retMsg->setEventType(eventMsg->getEventType());
            retMsg->setEventArg(eventMsg->getEventArg());
            retMsg->setUserArg(eventMsg->getUserArg());

            send(retMsg, mReturnGate);
        }
    }
}

interface::api::ErrorType EventBase::processEvent(interface::api::ApiEventType eventType, interface::api::ApiEventArg eventArg)
{
    if (mPrintEventType)
    {
        EV << "Process␣event:" << endl;
        EV << "␣EventType:␣" << interface::debug::getApiEventStr(eventType) << endl;
    }

    emit(mEventTypeSignal, eventType);

    interface::api::ErrorType ret = interface::api::Error::kErrorOk;

    switch (eventType)
    {
        case interface::api::ApiEvent::kOplkApiEventNmtStateChange:
            emit(mNmtStateSignal, eventArg.nmtStateChange.newNmtState);
            ret = processNmtStateChangeEvent(eventType, eventArg);
            break;

        case interface::api::ApiEvent::kOplkApiEventCriticalError:
            ret = processErrorEvent(eventType, eventArg);
            break;

        case interface::api::ApiEvent::kOplkApiEventWarning:
            ret = processWarningEvent(eventType, eventArg);
            break;

        case interface::api::ApiEvent::kOplkApiEventHistoryEntry:
            ret = processHistoryEvent(eventType, eventArg);
            break;

        case interface::api::ApiEvent::kOplkApiEventNode:
            ret = processNodeEvent(eventType, eventArg);
            break;

        case interface::api::ApiEvent::kOplkApiEventPdoChange:
            ret = processPdoChangeEvent(eventType, eventArg);
            break;

        case interface::api::ApiEvent::kOplkApiEventCfmProgress:
            ret = processCfmProgressEvent(eventType, eventArg);
            break;

        case interface::api::ApiEvent::kOplkApiEventCfmResult:
            ret = processCfmResultEvent(eventType, eventArg);
            break;

        default:
            break;
    }
    return ret;
}

interface::api::ErrorType EventBase::processNmtStateChangeEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    EV << "NMT␣State␣change␣from␣" << interface::debug::getNmtStateStr(eventArg.nmtStateChange.oldNmtState) << "␣to␣" << interface::debug::getNmtStateStr(eventArg.nmtStateChange.newNmtState) << endl;

    // check shutdown
    if (eventArg.nmtStateChange.newNmtState == interface::api::NmtStateE::kNmtGsOff)
    {
        EV << "Stack␣received␣kNmtGsOff!" << endl;
        send(new cMessage("shutdown␣demo"), mShutdownGate);
    }

    return interface::api::Error::kErrorOk;
}

interface::api::ErrorType EventBase::processErrorEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    EV << "Error␣event:␣" ;
    EV << "␣␣Error:␣␣␣␣␣␣␣"<< interface::debug::getRetValStr(eventArg.internalError.oplkError) << endl;
    EV << "␣␣EventSource:␣"<< interface::debug::getEventSourceStr(eventArg.internalError.eventSource) << endl;
    EV << "␣␣ErrorArg:␣␣␣␣"<< eventArg.internalError.errorArg.uintArg << endl;

    send(new cMessage("shutdown␣demo"), mShutdownGate);
    //error(interface::debug::getRetValStr(eventArg.internalError.oplkError));

    //throw interface::OplkException("ErrorEvent", eventArg.internalError.oplkError);

    return eventArg.internalError.oplkError;
}

interface::api::ErrorType EventBase::processWarningEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    EV << "Warning␣event:␣" << interface::debug::getRetValStr(eventArg.internalError.oplkError) << endl;

    return eventArg.internalError.oplkError;
}

interface::api::ErrorType EventBase::processHistoryEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    auto event = eventArg.errorHistoryEntry;

    EV << std::hex << "History␣event:␣" << endl;
    EV << "␣␣␣Type:␣␣␣␣␣␣" << event.entryType << endl;
    EV << "␣␣␣ErrorCode:␣" << event.errorCode << "␣-␣" << interface::debug::getRetValStr(event.errorCode) << endl;
    EV << "␣␣␣Timestamp:␣" << event.timeStamp.sec << "␣-␣" << event.timeStamp.nsec << endl;
    EV << "␣␣␣Addr:␣␣␣␣␣␣";
    for (auto i = 0u; i < sizeof(event.aAddInfo); i++)
        EV << "␣" << static_cast<unsigned int>(event.aAddInfo[i]);
    EV << endl << std::dec;

    return interface::api::Error::kErrorOk;
}

interface::api::ErrorType EventBase::processNodeEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    auto event = eventArg.nodeEvent;

    EV << "Node␣event:␣" << endl;
    EV << "␣␣␣Type:␣␣␣␣␣␣" << interface::debug::getNmtNodeEventTypeStr(event.nodeEvent) << endl;
    EV << "␣␣␣ErrorCode:␣" << event.errorCode << endl;
    EV << "␣␣␣Mandatory:␣" << event.fMandatory << endl;
    EV << "␣␣␣NmtState:␣␣" << interface::debug::getNmtStateStr(event.nmtState) << endl;
    EV << "␣␣␣NodeId:␣␣␣␣" << event.nodeId << endl;

    return interface::api::Error::kErrorOk;
}

interface::api::ErrorType EventBase::processPdoChangeEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    auto event = eventArg.pdoChange;

    EV << "PDO␣change␣event:␣" << endl;
    EV << "␣␣␣NodeId:␣␣␣␣" << event.nodeId << endl;
    EV << "␣␣␣ParamIdx:␣␣" << event.mappParamIndex << endl;
    EV << "␣␣␣ObjCount:␣␣" << event.mappObjectCount << endl;
    EV << "␣␣␣Activated:␣" << event.fActivated << endl;
    EV << "␣␣␣Tx:␣␣␣␣␣␣␣␣" << event.fTx << endl;

    return interface::api::Error::kErrorOk;
}

interface::api::ErrorType EventBase::processCfmProgressEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    auto event = eventArg.cfmProgress;

    EV << "Cfm␣progress␣event:␣" << endl;
    EV << "␣␣␣NodeId:␣␣␣␣␣␣␣␣␣␣" << event.nodeId << endl;
    EV << "␣␣␣BytesDownloaded:␣" << event.bytesDownloaded << endl;
    EV << "␣␣␣Error:␣␣␣␣␣␣␣␣␣␣␣" << event.error << "␣-␣" << interface::debug::getRetValStr(event.error) << endl;
    EV << "␣␣␣ObjectIndex:␣␣␣␣␣" << event.objectIndex << endl;
    EV << "␣␣␣ObjectSubIndex:␣␣" << event.objectSubIndex << endl;
    EV << "␣␣␣SdoAbortCode:␣␣␣␣" << event.sdoAbortCode << "␣-␣" << interface::debug::getAbortCodeStr(event.sdoAbortCode) << endl;
    EV << "␣␣␣TotalNumOfBytes:␣" << event.totalNumberOfBytes << endl;

    return interface::api::Error::kErrorOk;
}

interface::api::ErrorType EventBase::processCfmResultEvent(interface::api::ApiEventType eventType,
        interface::api::ApiEventArg eventArg)
{
    auto event = eventArg.cfmResult;

    EV << "Cfm␣progress␣event:␣" << endl;
    EV << "␣␣␣NodeId:␣␣␣␣␣␣␣␣␣" << event.nodeId << endl;
    EV << "␣␣␣NmtNodeCommand:␣" << interface::debug::getNmtNodeCommandTypeStr(
     event.nodeCommand) << endl;

    return interface::api::Error::kErrorOk;
}
```

Listing C.23: AppBase.h

```
/**
    *****************************************************************************************
 \file   AppBase.h

 \brief  Include file for Base class of App modules.

 *********************************************************************************
    */

/*------------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 ------------------------------------------------------------------------------
    */

#ifndef __OPENPOWERLINK_APPBASE_H_
#define __OPENPOWERLINK_APPBASE_H_

#include <omnetpp.h>
#include <vector>
#include "UseApiBase.h"
#include "ReturnHandler.h"

/**
 * \brief Base class for App modules implementing basic functionalities.
 *
 * This class inherits from UseApiBase and represents a cSimpleModule which
 * sends messages to the Api module and blocks the according method calls
 * until reception of the return message.
 * Addionally this class implements the message handling for communication with
 * other Modules and the according function handling.
 */
class AppBase : public UseApiBase
{
        // Definitions
    public:
        /**
         * \brief Types of functions available via messages
         */
        enum class AppBaseCallType
            : short
            {
                init = 0, shutdown, processSync
        };

    protected:
        using GateCont = std::vector<OPP::cGate*>;
        using HandlerCont = std::vector<HandlerPtr>;
        // C-Tor
    public:
        /**
         * \brief Default constructor initializing base class
         */
        AppBase();

        // Methods
    protected:
        virtual void initialize();
        virtual void handleOtherMessage(MessagePtr msg);

        /**
         * \brief Initialize Application
         *
         * This pure virtual method is called after receiving a message with
         * messageKind AppBaseCallType::init.
         */
        virtual interface::api::ErrorType initApp() = 0;

        /**
         * \brief Execute synchronous operations
         *
         * This pure virtual method is called after receiving a message with
         * messageKind AppBaseCallType::processSync.
         */
        virtual interface::api::ErrorType processSync() = 0;

        /**
         * \brief Shutdown Application
         *
         * This pure virtual method is called after receiving a message with
         * messageKind AppBaseCallType::init.
         */
        virtual void shutdownApp() = 0;

        /**
         * \brief Handler for messages with different kinds than AppBaseCallType
     .
         */
        virtual void handleAppMessage(MessagePtr msg);

        // Member
    private:
        GateCont mReturnGates;

    protected:
        HandlerCont mReturns;
};

#endif
```

Listing C.24: AppBase.cc

```
/**

     *******************************************************************************

 \file   AppBase.cc

 \brief  Implementation of Base class of App modules.

 ******************************************************************************/

/*--------------------------------------------------------------------------------
 Copyright (c) 2016, Franz Profelt (franz.profelt@gmail.com)
 ------------------------------------------------------------------------------*/

#include <functional>
#include "AppBase.h"

USING_NAMESPACE

AppBase::AppBase() :
        UseApiBase("apiCall")
{
}

void AppBase::initialize()
{
    UseApiBase::initialize();

    for (auto i = 0; i < gateSize("functionReturn"); i++)
    {
        // resolve gates
        mReturnGates.push_back(gate("functionReturn", i));

        // create return handler with send method bound to this and the return gate
        mReturns.emplace_back(new ReturnHandler(std::bind(static_cast<int (AppBase::*)(cMessage*, cGate*)>(&AppBase::send),
                        this, std::placeholders::_1, mReturnGates[i])));
    }
}

void AppBase::handleOtherMessage(MessagePtr msg)
{
    if (msg != nullptr)
    {
        interface::api::ErrorType ret = interface::api::Error::kErrorOk;

        // check message kind
        switch (static_cast<AppBaseCallType>(msg->getKind()))
        {
            case AppBaseCallType::init:
                ret = this->initApp();
                break;
            case AppBaseCallType::processSync:
                ret = this->processSync();
                break;
            case AppBaseCallType::shutdown:
                this->shutdownApp();
                break;
            default:
                handleAppMessage(msg);
        }

        // send return message
        mReturns.at(msg->getArrivalGate()->getIndex())->sendReturnValue(ret, msg
    ->getKind());
    }
}

void AppBase::handleAppMessage(MessagePtr msg)
{
    // Empty function stub within base class
}
```

References

Literature

[1] András Varga. *OMNeT++ - Manual* (cit. on pp. 1, 4–13, 15, 16, 29, 30, 32).

[2] András Varga. *OMNet++ - UserGuide* (cit. on p. 12).

[3] R. L. Bagrodia and M. Takai. "Performance evaluation of conservative algorithms in parallel simulation languages". In: *IEEE Transactions on Parallel and Distributed Systems* 11.4 (Apr. 2000), pp. 395–411 (cit. on pp. 26, 27, 30).

[4] Rajive Bagrodia et al. "Parsec: A parallel simulation environment for complex systems". In: *Computer* 31.10 (1998), pp. 77–85. (Visited on 04/13/2016) (cit. on pp. 26, 27).

[5] J. Belanger, P. Venne, and J. N. Paquin. "The what, where and why of real-time simulation". In: *Planet RT* 1 (2010), p. 1 (cit. on pp. 15, 16, 18).

[6] Ronald C. De Vries. "Reducing null messages in Misra's distributed discrete event simulation method". In: *Software Engineering, IEEE Transactions on* 16.1 (1990), pp. 82–91. (Visited on 04/05/2016) (cit. on p. 27).

[7] EPSG. *EPSG Draft Standard 301 Ethernet POWERLINK Communication Profile Specification.* Tech. rep. V1.2.0. 2013 (cit. on pp. 52–56).

[8] *::: EPSG - About EPSG :::* http://www.ethernet-powerlink.org/en/organization/about-epsg/. (Visited on 05/15/2016) (cit. on p. 51).

[9] *::: EPSG - CANopen and POWERLINK :::* http://www.ethernet-powerlink.org/de/powerlink/technologie/canopen-and-powerlink/. (Visited on 05/20/2016) (cit. on p. 57).

[10] *IEEE Standard for Ethernet - Section 5.* No Linguistic Content. Piscataway, USA: IEEE, 2016 (cit. on pp. 51, 52).

[11] *Intel® MPI Library | Intel® Software.* https://software.intel.com/en-us/intel-mpi-library. (Visited on 05/15/2016) (cit. on p. 31).

[12] Devendra Kumar and Saad Harous. “A study of achievable speedup in distributed simulation via NULL messages”. In: *Parallel and Distributed Systems, IEEE Transactions on* 4.3 (1993), pp. 347–354. (Visited on 04/05/2016) (cit. on p. 27).

[13] *Lenovo U530 Touch | 15.6" Touch Screen Laptop | Lenovo US.* http://shop.lenovo.com/us/en/laptops/lenovo/u-series/u530-touch/#tab-tech_specs. (Visited on 03/22/2016) (cit. on p. 37).

[14] Bin Lu et al. “A Low-Cost Real-Time Hardware-in-the-Loop Testing Approach of Power Electronics Controls”. In: *IEEE Transactions on Industrial Electronics* 54.2 (Apr. 2007), pp. 919–931. (Visited on 03/17/2016) (cit. on p. 19).

[15] Norm Matloff. “Introduction to discrete-event simulation and the simpy language”. In: *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* 2 (2008), p. 2009. (Visited on 03/14/2016) (cit. on p. 15).

[16] Roger McHaney. *Understanding computer simulation.* Ventus Publishing, 2009 (cit. on p. 14).

[17] *Microsoft MPI.* https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx. (Visited on 05/15/2016) (cit. on p. 31).

[18] *OMNeT++ API Reference* (cit. on pp. 17, 20).

[19] *OMNeT++ Parallel API Reference* (cit. on p. 30).

[20] *Open MPI: Open Source High Performance Computing.* https://www.open-mpi.org/. (Visited on 04/15/2016) (cit. on p. 31).

[21] *OpenAutomationTechnologies/openPOWERLINK_omnetpp.* https://github.com/OpenAutomationTechnologies/openPOWERLINK_omnetpp. (Visited on 06/17/2016) (cit. on p. 83).

[22] *OpenAutomationTechnologies/openPOWERLINK_V2: Release 2 of the openPOWERLINK protocol stack.* https://github.com/OpenAutomationTechnologies/openPOWERLINK_V2. (Visited on 05/16/2016) (cit. on p. 51).

[23] *openPOWERLINK documentation* (cit. on pp. 57, 59, 61, 62).

[24] *openPOWERLINK download | SourceForge.net.* https://sourceforge.net/projects/openpowerlink/. (Visited on 05/16/2016) (cit. on p. 51).

[25] *openPOWERLINK :: openPOWERLINK.* http://openpowerlink.sourceforge.net/web/. (Visited on 03/29/2016) (cit. on p. 1).

[26] *openPOWERLINK :: POWERLINK/Object Dictionary :: Object.* http://openpowerlink.sourceforge.net/web/POWERLINK/Object%20Dictionary/Object.html. (Visited on 05/16/2016) (cit. on p. 57).

[27] *RAII - cppreference.com.* http://en.cppreference.com/w/cpp/language/raii. (Visited on 05/20/2016) (cit. on p. 74).

[28] Syed S. Rizvi, Khaled M. Elleithy, and Aasia Riasat. “Reducing null message traffic in large parallel and distributed systems”. In: *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on.* IEEE, 2008, pp. 1115–1121. (Visited on 04/05/2016) (cit. on p. 27).

[29] Artur Austregesilo Scussel et al. “Improvements in OMNeT++/INET Real-Time Scheduler for Emulation Mode”. In: *arXiv preprint arXiv:1509.03105* (2015). (Visited on 12/17/2015) (cit. on pp. 16, 20).

[30] The MPI Forum. *MPI: A Message-Passing Interface Standard 3.1.* Tech. rep. 2015 (cit. on p. 28).

[31] Andras Varga and Ahmet Y. Dekercioglu. “Parallel Simulation Made Easy With OMNeT+”. In: (2003) (cit. on pp. 29, 31).

[32] András Varga, Ahmet Y. Şekercioğlu, and Gregory K. Egan. “A Practical Efficiency Criterion for the Null-Message-Algorithm”. In: *In Proc. of European Simulation Symposium.* 2003 (cit. on p. 27).

[33] *What is emulation? | Koninklijke Bibliotheek.* https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation. (Visited on 05/12/2016) (cit. on p. 19).