Fikret Bašić, BSc

SimPar - A Concurrent Simulation System for Evaluation of Power-Aware Smart Sensors

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Institute for Technical Informatics

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Advisers

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Thomas Wolfgang Pieber

Graz, December 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

We currently live in a time where rapid development, integration of the Industry 4.0, hardware/software co-design, and accurate model predictions are becoming more prevalent in most companies. An accurate simulation rises the chances of having a good product in the end. The thesis focuses on analysing and researching the interaction between an external simulation and a SystemC model. Primarily, the focus is set on designing, implementing and finally evaluating a framework for enabling a concurrent interaction between the different simulation models. To achieve the concurrency, a model based on the client-server network model is proposed. The Server handles the communication protocol and simulation evaluation, while the client runs the actual simulation and presents its results. This system framework solution is named "SimPar", a name derived from the *simulation* and *parallel* features. Extensibility, security, and energy efficiency play a major role. To test the created framework, a power-aware smart sensor model has been implemented in SystemC. It has been evaluated on functionality, accuracy, and energy consumption. The final SimPar simulation product offers a state-of-the-art SystemC simulation environment and provides reusable design principles and solutions.

Kurzfassung

Wir leben jetzt in einer Zeit, in der schnelle Entwicklung, Integration von Industrie 4.0, Hardware/Software Co-Design und genaue Modellvorhersagen immer wichtiger in Firmen werden. Eine gute Simulation erhöht die Chancen, am Ende ein gutes Produkt zu erhalten. Meine Arbeit fokussiert sich auf das Analysieren und Erforschen der Interaktionen zwischen einer externen Simulation und einem SystemC Modell. Dabei liegt der Schwerpunkt vor allem auf dem Designen, Implementieren und schlussendlich Evaluieren einer Framework, die eine parallele Interaktion mit verschiedenen Simulationsmodellen ermöglicht. Um die Nebenläufigkeit zu erreichen, wurde ein Modell vorgeschlagen, das auf einer Client-Server Netzwerktopologie basiert. Der Server übernimmt das Kommunikationsprotokoll und die Evaluierung der Simulation, während der Client die Simulation selbst übernimmt und die Ergebnisse veröffentlicht. Diese System-Rahmenstruktur wird "SimPar" genannt, ein Name, der von den Simulationseigenschaften und der Parallelität abgeleitet ist. Erweiterbarkeit, Datensicherheit und Energieeffizienz spielen dabei eine große Rolle. Zur Evaluierung des erstellten Systems wurde ein SystemC Modell eines energiebewussten Smart Sensors entwickelt. Es werden die funktionale Richtigkeit und der Energieverbrauch evaluiert. Das SimPar Simulationsprodukt stellt eine SystemC-Simulationsumgebung mit wiederverwendbare Designprinzipien und Lösungen zur Verfügung.

Acknowledgements

I would like to give a special thanks to my supervisor, mentor and professor, Prof. Dr.techn. Christian Steger for giving me the opportunity to work this past year with a wonderful team on the IoSense project. I am very grateful for all the advices and guidance that I received during the course of that work period and for allowing me to extend the work of the project by defining it as my master thesis.
I would like to thank my other thesis committee members for their own guidance and finding the time to attend to my exam when it was the most crucial for me.

I would like to thank my fellow co-workers and researches at the Institute of Technical Informatics and for all the great advices that you gave me during the course of this thesis.
I especially want to give thanks to my co-mentor DI Thomas Wolfgang Pieber for support, advices and fast corrections. Without you I would have been lost on many occasions.

And finally, I would like to give a special thanks to my parents who were always there for me, supported me and believed that I can and will achieve this milestone. It was you who first achieved the title of Dipl.-Ing. and gave me the inspiration to follow in these footsteps.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CRC	Cycle Redundancy Check
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIR	Finite Impulse Response
FPGA	Field-programmable Gate Array
FRAM	Ferroelectric Random Access Memory
FSM	Finite State Machine
FSM	Finite State Machine
GoF	Gang of Four
GUI	Graphical User Interface
HDL	Hardware Description Language
HTML	HyperText Markup Language
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IP	Internet Protocol
LAN	Local Area Network
LSB	Least Significant Bit
MCU	Micro Controller Unit
MPSoC	Multi-Processor System on Chip
NFC	Near-field Communication
OS	Operating System
POSA2	Pattern-Oriented Software Architecture
RAM	Random Access Memory
RF	Radio Frequency
SMP	Symmetric Multiprocessing

SOC	System on a Chip
TCP	Transmission Control Protocol
TLM-DT	Transaction Level Modelling with Distributed Time
TLM	Transaction Level Modelling
UI	User Interface
ULP	Ultra Low Power
UML	Unified Modelling Language
VCD	Value Change Dump
VHDL	VHSIC Hardware Description Language
XML	EXtensible Markup Language

1 Introduction

With the advancements in the field of Hardware-Software development it is becoming increasingly apparent that the need arises for faster and more accurate development of models used in simulations. As it was defined by Elshamy and Elssamadisy [13], rather than focusing on a large problem by dividing it into smaller and independent parts with the use of "*Divide and Conquer*" methodology, it is necessary to approach the problem from an analytical standpoint. The thesis primarily focuses on investigating the methods and principles on which an efficient simulation system can be set up for simulating hardware devices in a virtual environment. It then proposes its own model for simulation, as well as a hardware model which should fulfil the requirements set by modern industry. The collective solution of methods, principles and technologies used in the aforementioned project is named SimPar. The name is derived from the "parallel simulation" aspect and from the SystemC being the primary tool used for modelling and simulation.

This section will give a brief overview of the main motivation behind this thesis, will go in depth of the problem source, and it will give proposals for the theoretical solutions. The thesis was done as a part of a larger IoSense project[1] funded by the European Union and co-worked at the Institute of Technical Informatics - Hardware/Software Codesign research group of the Technical University of Graz. The project is aimed at the development and state-of-the-art research in new appliances and technologies related to sensor communication, security and energy efficiency, among other elements. More details can be seen on the official site of the project at IoSense [22]. Theoretical and scientific hypothesis for the parallel execution of the SystemC simulations with Gazebo system in the background were based on the work done by Pieber, Ulz, and Steger [34].

[1] http://www.iosense.eu/

1.1 Motivation

Development of hardware devices was historically generally done independently from the software development. One of the reasons was that both the knowledge and experience of the designers that worked on the modelling of products were either aimed at hardware or software parts. The requirement set by the professions influenced such divisions and there was a certain layer of miscommunication and misunderstanding between the parties which lead to the situation where the work would generally remain independent until the need to finalize and test the product. As stated by Teich [43], the other reason is that the modelling standards in the industry were not catching up with the changes in technology and tools, still relying on old, but at least proven methods of development.

Originally, there were not many devices, especially small and portable ones, which offered or required special need of software attention. That changed in 70's of the 20th century when first microcomputers arrived at the market [27]. Microcomputers generally consisted of a microprocessor, a term that is sometimes used indistinguishable from the microcomputers. Further developments resulted in microcontrollers which greatly influenced the ability to customize own code and functions for portable and smaller hardware devices, but at the same time still resulted in a more divided community of developers. Finally, the development of FPGA (Field-programmable Gate Array) and the creation of the general concept of SOC (System on a Chip) lead to the need of creating software tools for the hardware developers. Today these tools are generally known as the HDL (Hardware Description Language)s and are used extensively by both the hardware and software developers [15]. One of the most well know HDL are VHDL (VHSIC Hardware Description Language) and Verilog.

The objective of the aforementioned HDLs was to keep the balance of abstractions to be understandable to both the hardware and software developers [26]. The balance shifted in the favour of software developers with the development of *SystemC*. Unlike other HDLs, SystemC focuses more on modelling and simulating a resulting hardware product rather than making a synthesizable model of it [7]. These models were generally faster and easier to develop and test, but the simulations can potentially be slow, especially if

programmed on a lower layer of abstraction. That, and the fact that SystemC is only intended to be used on a single-core CPU (Central Processing Unit) meant that increasing the performance by splitting the task was difficult. Once a simulation has started, its instance cannot be tempered with until it is finished. This lead to problems, one of them being that if everything is run on one machine there should be enough storage to save the trace data for long-running simulations.

Teich [43] states that to remedy the problem it is necessary to try and develop a system for parallel simulation which would not only be applicable for one model, but rather to allow an easy integration of any subsequent model insertion and control. But the hardware developers are not only satisfied with having a simulation system.
It is necessary to provide a communication protocol between the model and the simulation system, along with a blueprint for developing efficient SystemC models for more accurate simulations. These models should be easy to understand and apply. It is necessary for them to be coherent for both hardware and software developers. The technical models should also try to provide simulation results on elements like energy consumption, functional efficiency and processing time.

1.2 Problem Analysis

The design problem rose from a simulation system which was in development for the IoSense project by using the traditional approach with the SystemC simulation. The approach of the design, as well as the problem description, were based on the work by Pieber, Ulz, and Steger [34].
The system is divided in two parts:

- *Gazebo model* - a model used for simulating a robot and its interaction with the environment
- *Smart sensor model* - a SystemC model which describes a complex hardware element composed of a microcontroller and sensors

The gazebo model (the robot) would interact with the smart sensor model by issuing commands. These commands were part of a larger communication

protocol devised on a XML-like schema, where the data necessary for the simulation would be transmitted. Each time a new message packet is sent the simulation has to stop before the packet can be processed. Only after the new packet is processed, the simulation can carry on from the previously stopped timestamp. This process is very time consuming and can potentially create a bottleneck if there are many messages already residing in the buffer. Figure 1.1 presents this process graphically.

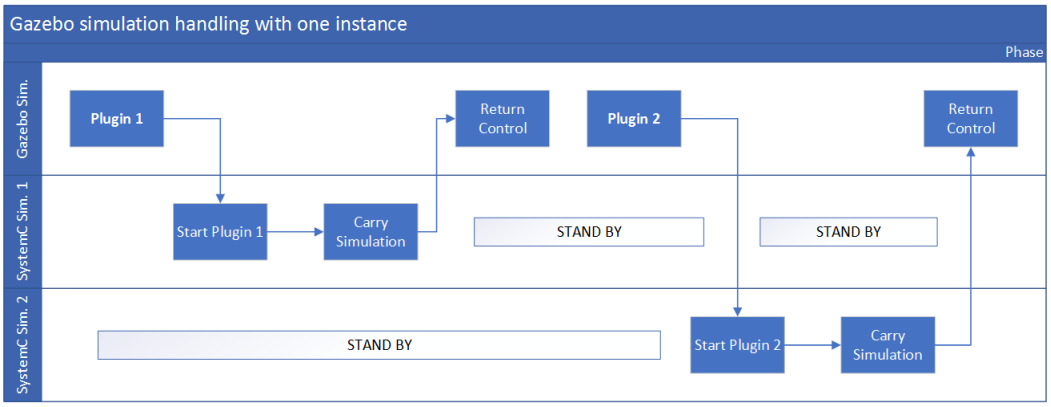

Figure 1.1: Swimline model representation of the original Gazebo - SystemC interaction concept [34]

The following improvements have been proposed to improve the overall model:

- Create a new communication model - redefine a communication system with the use of network capabilities
- Separation between the simulation models - add abstraction between the SystemC and the interactive (Gazebo) model
- Redefine a new Smart Sensor SystemC model - model a new smart sensor with focus on energy efficiency

With the proposed suggestions the design steps would remain focused on the flexibility and extensibility of the model. It was suggested by Lapalme et al. [25], that this approach would lead to the creation of an environment which potentially helps both the hardware-software developers in modelling a separate sensor models and developers focusing on creating an interactive command system. That environment would be composed of the collection

of individual solutions which would eventually lead to the creation of an unified methodology named “SimPar”.

1.2.1 Concurrent Simulation System

A proposition was made, to try and create a system which would take the advantage of running a simulation model with multiple PCs. Emerging use of a large number of computer devices without direct need of knowledge or interaction with them, for the purpose of fulfilling a web service task, is called *cloud computing*. As stated by Keane [23], this form of web service is becoming ever more prevalent in use of modelling and simulations, since it allows for more and better computational possibilities. But, the implementation of these systems, by using standard cloud approaches, can be difficult. That comes from the fact that each simulation must be identifiable for it to be possible to be evaluated. The identification can be ensured with the use of a proper communication protocol and master-slave hierarchy, where client-server topology is one of the most prevalent ones. Such systems are rarely seen, especially in the domain of using SystemC for handling simulations. Most approaches tend to either change the core specification of SystemC to run on multiple cores by force or to simply use a stronger computer for running one simulation. These approaches are not always reliable and they might not work the same for every model.

The idea is to finally offer SystemC simulation developers a reliable and model-independent system for running more complex and parameter-specific simulations. An example of such simulation can be a model which tries to emulate a complex sensor network for interaction with a dynamic device, like a robot through Gazebo toolkit [18]. The robot can change its behaviour based on the readings from the surrounding environment and from the sensors. To simulate different behaviour based on the data for a certain time step, multiple simulations are needed. This can be very resource and time consuming. The proposed simulation system would then be seen as a framework which can allow for multiple models to be easily added and controlled based on the needs of the developer.

1.2.2 SystemC Overview

Development of new hardware products, which can range from small NFC (Near-field Communication) Antenna to complete Smart Sensors, is handled through diverse modelling and testing. For more portable, extensible, easier and complete models, it was decided to use SystemC as the hardware-description language for any further hardware modelling and simulation.

With SystemC it is possible to create a semi-abstract hardware-software development framework intended to imitate hardware programming with software approach. What it actually consists of is a set of different C and C++ based structures and classes, templates and macros used to create an event-driven simulation interface. Each structure or class in SystemC is viewed as a "module". These modules consist mainly of input and output ports, inner signals and processes. The processes can either be thread or method based. They are implemented like functions with a special trigger signal which indicates when the functions are being called.

Unlike programs written in standard C or C++ where a call to a function is blocking, in SystemC multiple functions, or threads, can be run in parallel, as each execution is set to be done in one delta cycle [7]. The number of delta cycles in a function can be controlled by inputting event or time based variables. A usual programming approach, which is also applied in this project, is to define functions as threads and set each thread to run for one cycle, where different steps of a function are controlled through states by an FSM (Finite State Machine).

While SystemC models are not intended to be used for direct synthesis (i.e. FPGA hardware mapping), it is possible to do so with a set of specially made software and hardware tools. A SystemC model should primarily be used to ease the interaction and communication between the software and hardware development departments, through fast modelling and simulation, as it allows for an easier understanding between both parties [24]. It can however be used together with other HDL for a full development cycle, most notably VHDL or Verilog. This is shown in Figure 1.2.

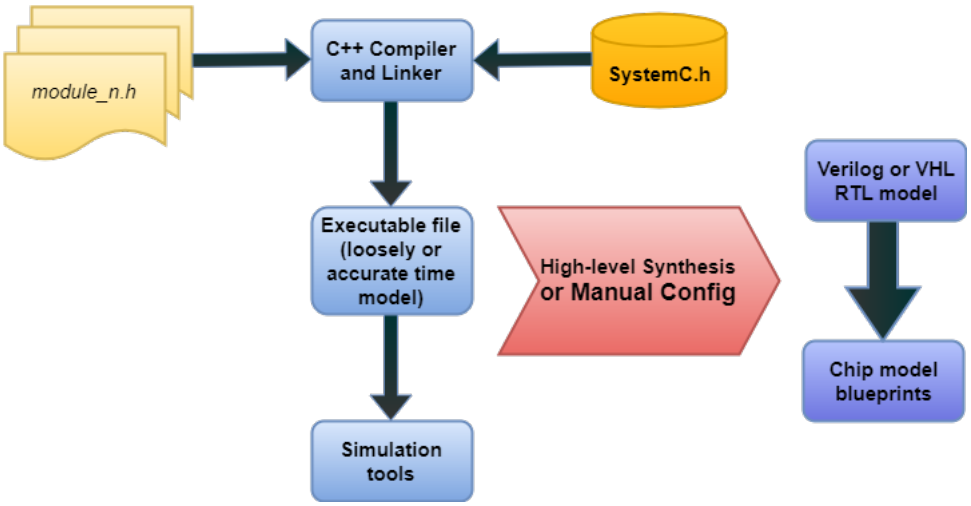

Figure 1.2: Modelling and development using SystemC and other HDLs

1.2.3 Client-Server Communication

The main goal behind SimPar is the development of the Server-Client system model for handling multiple SystemC simulations. Each SystemC simulation is to be run on a separate Client machine. The server has two tasks:

- handles the connection with the Clients
- provides necessary commands

These commands come from some other simulation, where in the current testing case, that is the Gazebo model.
The communication between the Client and the Server is handled with Unix-based Socket communication. Since SystemC is a utility library for the C++ programming language, the whole program was developed primarily in C++ with the Socket communication being done in a separate C file. Python was used for some additional smaller scripting purposes. The program behind the communication is developed like a framework, allowing for a simple insertion of additional different SystemC models, as well as classes.

Figure 1.3 shows the graphical representation of the communication and callbacks between server and clients. The system designed in this way helps in removing the mutex call usually set on the whole runtime execution to

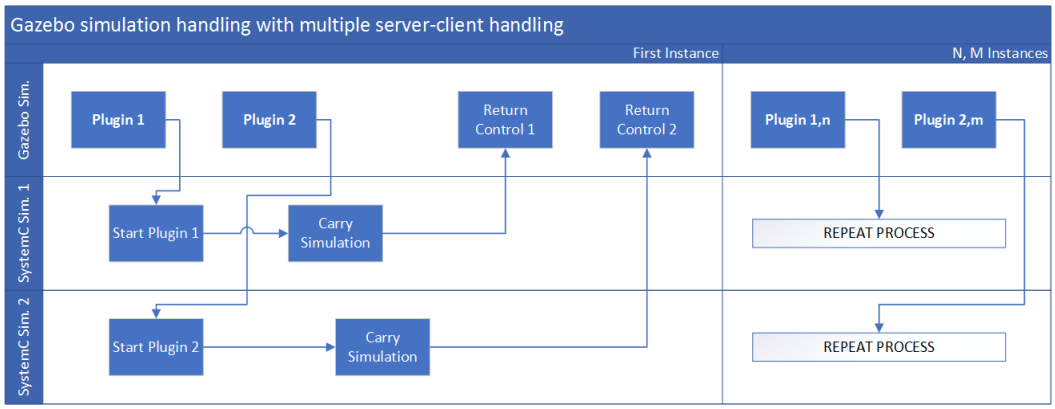

Figure 1.3: Swimline model representation of the network Gazebo - SystemC interaction concept

allow for receiving and interpreting the commands between the interactive device (i.e. Gazebo) and the sensor. Where previously the whole system had to be put on stop, the new model expends on this by allowing each sensor communication to be its own separate entity. That helps not only if the program is run on multiple machines, but also if it is run from the same computer, since all different socket connections between the main server and client will be done through a separate thread, rather than a forked process, therefore saving time and memory [5].

1.2.4 Smart Sensor Modelling

The second part of the SimPar project implementation required an evaluation of a smart sensor hardware device. The development was divided in two parts:

1. Modelling and Simulation
2. Hardware implementation

The hardware implementation and real-world energy consumption evaluation was done separately as its own element in the overall IoSense project. The element which was implemented as part of the SimPar evaluation model was the modelling and simulation of the hardware (sensor system with a

microcontroller). It essentially consists of developing a SystemC model and evaluating it through simulation.

Since SystemC allows for a verbose modelling through different layers of interaction, it was necessary to define which key simulation aspects are to be handled. It was decided to base the whole model foremost on the three following aspects:

- *Energy consumption* - keep track of used energy by summing up the power output of individual modules over the time of the simulation
- *Time accuracy* - retain the number of hardware cycles needed to process certain function
- *Functionality accuracy* - aim for a robust and correct model representation

Main source of material for the development of individual modules consisted primarily of the hardware documentation and related sources. Since it was very difficult to retain the model accuracy of one key aspect compared to the other, it was generally aimed to try and balance out the settings of each individual module to fulfil the correct end results, i.e. if certain functions could not be fully implemented, change the time and energy consumption accordingly. Work done by Ulz et al. [44] served as one of the main sources of inspiration for the simulation development of the NFC and security based smart sensor.

Energy Efficiency and Measurements

An important factor in the development of hardware devices is power management. Power dissipation depends on individual dissipations of its modules, being influenced both from the hardware components and software functionality design.
Before finishing the hardware prototype design it is recommended to test its potential energy consumption. This is of great importance while working on sensor or portable systems, since having a system which is dependent on battery requires it to be better optimized to effectively prolong the life-duration and cost of the device.

Black et al. [7] state that since SystemC was primarily developed as a general-purpose HDL focusing on transitional and behavioural system models, for it to be able to capture the energy consumption it is necessary to implement a mathematical model. The calculation formulas are based on a simple design. Generally, every module is described through a certain set of *power states*. Each power state outputs the typical current consumption defined by the real-world measurements and is generally taken either directly from the documentation or based on the behaviour of other, similar hardware elements. The current consumptions are then summed and multiplied with the given voltage. Energy is then calculated directly as the integration of power dissipation through time:

$$i(t) = \sum_{k=1}^{pStates} i_k(t) \tag{1.1}$$

$$E = \int u(t) * i(t) dt \tag{1.2}$$

During the design of the Smart sensor as part of the SimPar evaluation it was necessary to pick low-consumption hardware elements. These components are generally found today under the advertising name "*ultra-low power*". They are characterized by a low power consumption which generally ranges between a few milliamperes to even micro-amperes, and feature one or several additional low-power (sleep) states.

While most components generally consume energy, some other actively participate in the process known as "energy harvesting". During this action an additional power is drawn to power the device. El-Sayed et al. [35] mention that this process is usually done through a special hardware element, in this case that being an NFC chip. Sometimes the drawn power is sufficient to operate the whole device instead of a battery.

1.3 Thesis Contribution

SimPar was developed because of several needs found both in modelling and simulation of sensor systems. Because of that, it can generally be viewed as a collection of solutions rather than a single solution. The fields in which these solutions are applicable are SystemC behavioural modelling and development of a system for an environmental simulation of hardware models (e.g. sensors) and actuators (e.g. robots). Some other ideas found in this thesis can also be used for general HDL modelling using some other language, for building a simulation-based client-server platform, for model creation, energy monitoring, parameter extraction, and efficiency optimisation.
Even though SystemC is becoming increasingly popular, and is being used as a HDL for fast modelling and testing, there have not been many standards and research work done in this field aimed at integrating models created in SystemC with other models in a larger simulation environment. SimPar tries to change that by offering a simple to learn framework and communication protocol which allows for an easy integration and configuration. Engineers wanting to use this approach can easily model an independent SystemC model. Afterwards it is only necessary to adapt this model to the present communication protocol and to define the interactive model of the environment, which can be run from any other simulation platform. Since SimPar is build using the client-server topology, it can easily be defined how many instances of the SystemC models can be run, where each instance can be independently configured and monitored. This offers a better flexibility and resource management than most on-the-go[2] programmed systems.

1.4 Thesis Structure

For an easier development and handling of the SimPar methodology, the overall project was split into two main phases as seen in Figure 1.4. The first phase titled "*Server-Client Model*" is concerned with the development of the template framework for SystemC multi-threaded simulation handling.

[2]simulation systems build for a specific purpose rather than being general

The second phase "*SystemC Model*" focuses on the modelling and simulation of an actual SystemC model which is then evaluated for the energy consumption and which results are then compared with the real-world measurements.

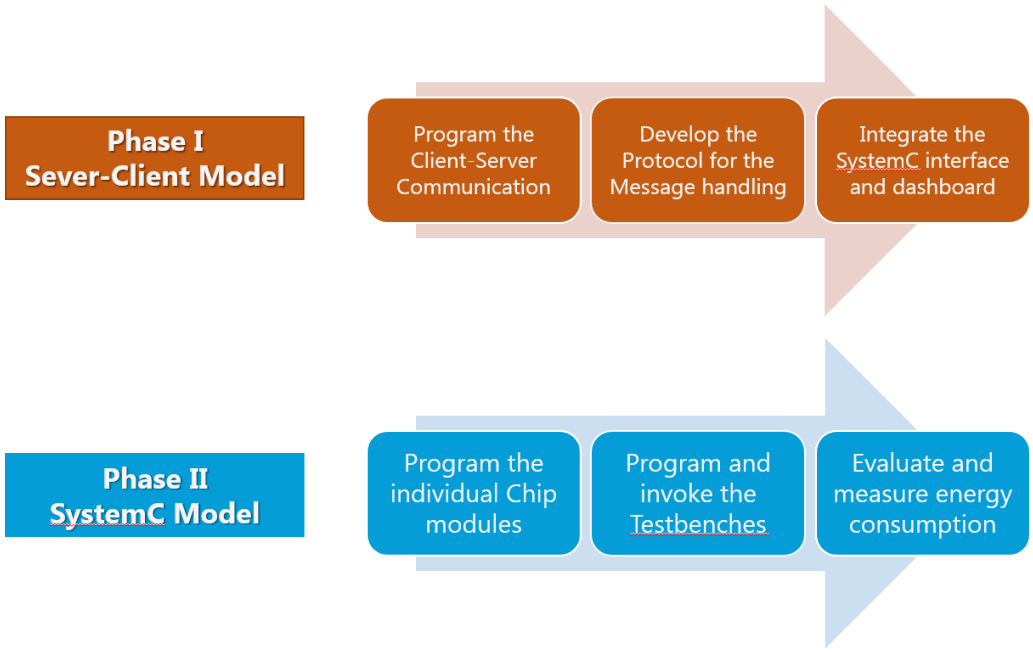

Figure 1.4: Main strategic plan for SimPar development

The thesis consists of the following chapters:

1. *Introduction* - current chapter, theoretical background and directions
2. *Related Work* - analysis of similar research work in the field
3. *Design* - theoretical and practical background behind the SimPar (algorithms, directions, tools, etc.)
4. *Implementation* - explanation how the end product was finally implemented by using various software tools and techniques
5. *Evaluation* - analysis of the end results, measurements, etc.
6. *Conclusion* - final word and future work

While the "Introduction" and "Related Work" are mostly concerned with the overall theoretical background of the thesis, the other three chapters are focused on the features of SimPar.

2 Related Work

This chapter inspects the related research work on the elements which are covered in this thesis. The primary purpose of the related work was to help in devising a design and subsequent implementation, as well as to follow the current trends in the *state-of-the-art* research. Some of these breakthroughs in SystemC, energy efficiency and smart sensors fields have been carefully analysed and partially implemented while others have been used as a guiding model to be build upon.

Most of the related work that has been analysed is indirectly correlated with the system behind this thesis - SimPar. That is mostly due to the nature of the SimPar methodology, where the central topic is based upon new approaches of handling SystemC simulation as its own entity on a higher abstraction layer in a networked system, something that has been not much researched on. As seen in the research papers, they are mostly focused on creating application-specific models and simulation systems based on the time and place of the research. Because of that, only the most relevant papers are going to be extracted and elucidated in this section. They are divided in three subsections, analysing concurrent and parallel running SystemC models, energy tracking techniques and smart sensor modelling.

2.1 Concurrent SystemC Models and Simulations

In modern times, a big focus in regards to the modelling and simulation is placed on the concurrent simulation of models. Depending on the practice, there exist three common approaches to these simulations:

1. *Multi-instance one-simulation* - instances of different elements in a model are run parallel inside one occurrence of a simulation

2. *One-instance multi-simulation* - there is only one instance of interest represented in a model which can be run parallel in multiple simulations
3. *Multi-instance multi-simulation* - multiple instances of a model can be run parallel together with multiple simulations

As it can be seen from the list, there are also different levels of complexity depending on the concurrent type of simulation run.
The example of the first type can be seen in the research paper titled "Concurrent Simulation Platform for Energy-Aware Smart Metering Systems" from Park et al. [33]. In this work, a proposal and implementation was made in which a smart house was designed as a model where each house appliance of interest (i.e. some object, fan or TV) was defined as a separate instance inside the model which could be run concurrently. SystemC was used in modelling and running the overall simulation. The goal was primarily tasked with handling energy consumption by monitoring each individual instance, as can be seen in the Figure 2.1.

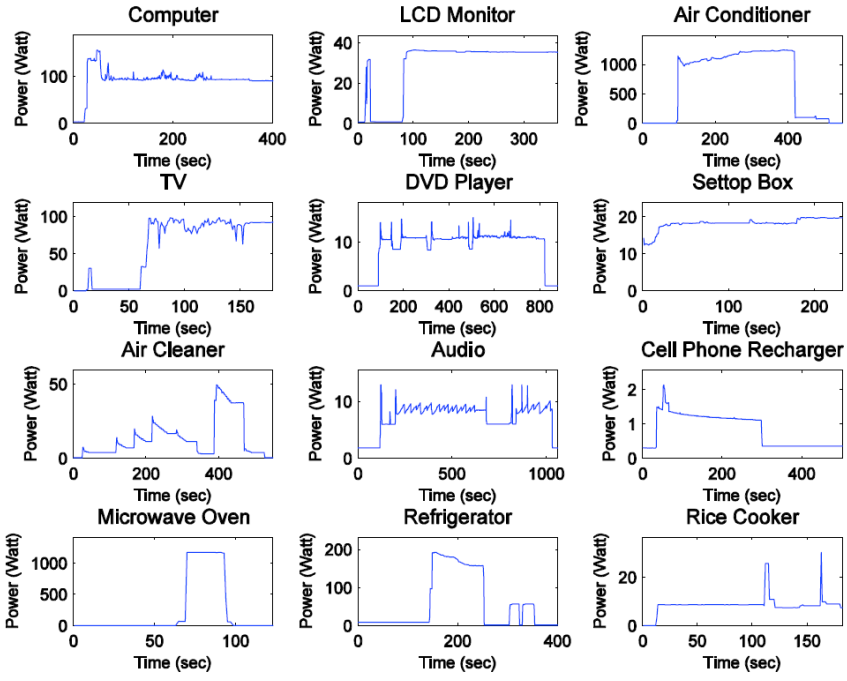

Figure 2.1: Various instances of monitoring power dissipation of home appliances (Park et al. [33])

While this method proved to be effective in the task that was given, it still did not provide a solution for running multiple instances of a SystemC simulation which is regarded as one of the main problems behind this simulation framework, e.g. defining different smart houses with diverse power model parameters.

With the development of SoC (System on Chip) devices, so came the development of the MPSoC (Multi-Processor System on Chip) which greatly influenced the research in the field of parallel modelling and simulation. Schumacher et al. [39] described one of the examples of an architecture that is developed for the SystemC simulation of MPSoC is *parSC*. The architecture uses different cores of a computer to try and synchronise the processing behaviour seen by the MPSoC devices. It results in a speed-up up to 4.4 compared to the classic SystemC implementations with the use of four cores.

The other important works in this field are work on parallel simulating a model using both the GPU (Graphics Processing Unit) and CPU interaction, by Sinha, Prakash, and Patel [41], and work by Huang et al. [21], where a general solution was proposed to the idea of developing SystemC models which allow for a parallel simulation execution in embedded systems. The second article in particular is important, since it defines a foundation on which many other works are based on. It does that by defining a state machine approach for handling different SystemC simulation instances.

2.1.1 Concurrency in Relation with SMP Machines

As it is apparent, many conventional SystemC researches have not dwelt too much into the concurrency in relation to the multi-instance multi-simulation field. On the contrary, there exist several papers that try to exploit parallel simulation based on the SystemC models on the SMP (Symmetric Multiprocessing) machines. They do this by trying to exploit the CPUs present in the SMP machines for running multiple threads. In this subsection two of those research papers are mentioned.

First custom implementations date back to 2009 from Ezudheen et al. [14], when three researchers from the Intel Corporation published a paper on

parallelizing SystemC kernel on SMP machines. This work focuses primarily on changing the kernel of the SystemC process handling by programming techniques and leverage for the parallel execution of multi-core machines. The changes in the design are presented to be similar to the previously researched "Distribution Library" which is primarily aimed at TLM (Transaction Level Modelling) rather than the cycle-accurate modelling. The changes for the low-level handling can be generally exploited by changing the behaviour on how the "scheduler" interacts with the running processes.

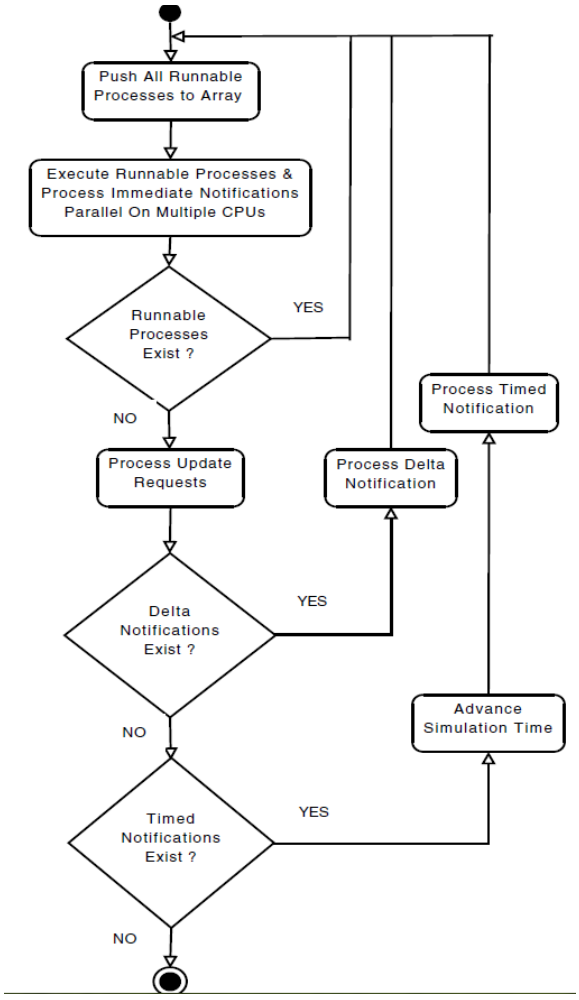

Figure 2.2: Proposed SystemC Parallel scheduler by Intel (Ezudheen et al. [14])

In the Figure 2.2 it can be seen how the scheduler behaves. It queues all the requests as it normally would in the sequential kernel. Afterwards, it creates the multiple execution environment, which has the information related to the currently executable threads and processes. The threads are assigned to each particular CPU (Central Processing Unit), sometimes grouping similar processes (threads) inside one CPU to allow for better cache and memory handling. After all runnable processes are executed and finished, the process proceeds to the *update* step which is identical to the sequential kernel.
Issues can arise during the last step by updating the process handler. Namely, the update queue can only accept one request at the time. To counter these inconsistencies, an implementation using the linked lists was suggested.

The overall results show that it is possible to get much better performance than with the sequential kernels, ideally going up to the times-n ("n" being the number of CPUs) faster execution time, but in reality, it is usually smaller due to the synchronisation between processes. Several methods have been proposed which help in synchronising threads, one of them being *Manual Grouping* where each group of processes is assigned separately. This method shows better overall performance, but when the simulations are too large (have many processes) and there are many CPUs presented by the SMP machine, then it is necessary to use the automatic groupings.

The second researcher paper of interest on the SMP machines, done by Aline et al. [1] from the Laboratorie d'Informatique de Paris 6, takes a similar approach by presenting their own scheduler engine build upon TLM2.0.
Their scheduler is named "*SystemC-SMP*" and is build upon the previous works on the TLM-DT (Transaction Level Modelling with Distributed Time). The paper claims that they managed to achieve 1.8 speed-up compared to the sequential simulation.

The architecture takes advantage of the aforementioned TLM-DT for providing a virtual platform (user defined) for running individual *SC THREADs*. The SystemC-SMP works as a kernel scheduler for assigning specific SC THREADs to the *POSIX Threads*. The abstract representation of the system can be seen in the Figure 2.3.
For *synchronisation* purposes, the architecture takes advantage of the events from the core SystemC commands. These events cycle through three states:

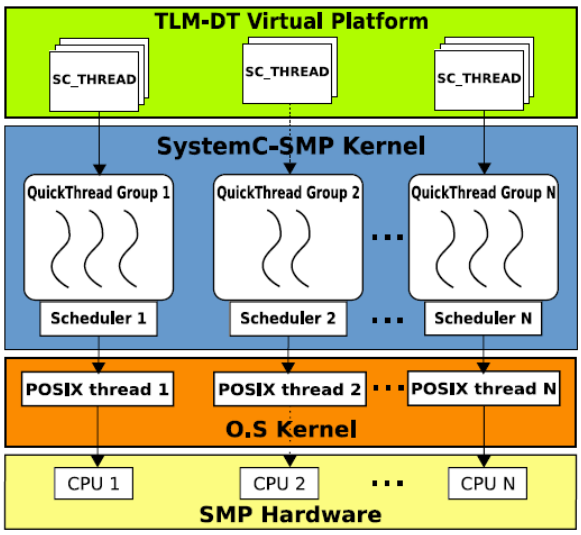

Figure 2.3: SMP Parallel SystemC Architecture (Aline et al. [1])

IDLE, *WAITING* and *NOTIFIED*. They are controlled with the *wait()* and *event.notify()* commands.

2.2 Energy Consumption and Power Estimation

Energy optimisation is a topic of interest found in many research papers, especially those focused on hardware simulation development. The reason comes from the fact that a precisely-developed simulation model can help in reducing cost and time during the development of real hardware in terms of predicting the energy consumption and optimizing it beforehand. As it is a strong point in industry, so it is also in this thesis.

Power measurement is used in the evaluation during the testing of the concurrent simulation platform, researched by Park et al. [33]. In the paper a simple, yet effective power model, was used where each device is described trough certain set of states. Each state needs a specific amount of power to use during the operation. During the operations the consumed energy is

calculated as sum between the products of power and time in found states. An example from the paper can be seen in the Figure 2.4.

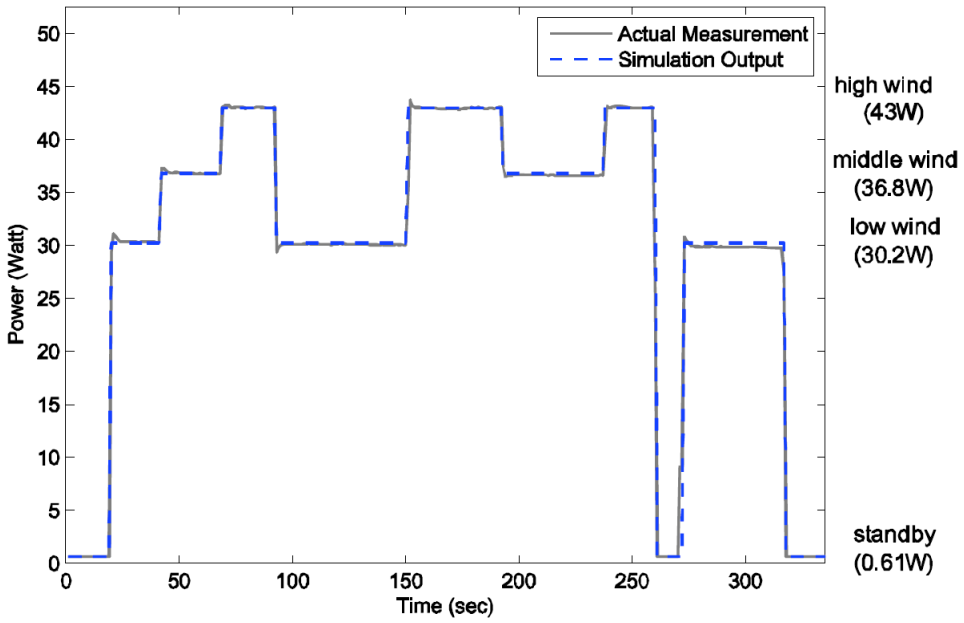

Figure 2.4: Power values during different states of a fan (Aline et al. [1])

The previously mentioned model is effective, as it is described in the paper, because it captures most of the power that such a device would use. In that process it mainly covers the main operational and standby states. There exist always additional power leakage which is hard to predict and which was addressed in the paper. The model resulted in the correlation of 0.973 compared to the real world measurements.
As it is observed, it can be seen that generally, research papers regarding the SystemC simulation mostly fall in either the TLM, behavioural, or system level category. Since TLM is of a higher abstraction level, which puts focus on the communication between modules rather than interworking of ones, there are not many quality papers that show a good energy-consumption model. One of the papers that seems to be focused in this field of TLM power estimation is titled "A Power Estimation Methodology for SystemC Transaction Level Models", by Nagu, Ing-Chao, and Vijay [30]. The paper however goes more into the detail on how the energy model is build around

constructing a hierarchical representation of a transaction level data, and a power model interface for mapping augmented power information, rather than how it would be possible to make a standard approach to energy estimation with TLM SystemC models.

A common idea among the designers of power models, especially among the researchers of sensor networks, is the use of power states. As already mentioned, power states are easy to implement and they generally result in good accuracy of the results. In the article submitted by Schmidt et al. [38], power states are separated between the inner states of modules and transition states. Transition states are the ones associated with transition between power states for wireless communication.
The formula used for the computation of energy measurement is:

$$E = \sum_{state} P_{state} * t_{state} + \sum_{trans} P_{trans} * t_{trans} \tag{2.1}$$

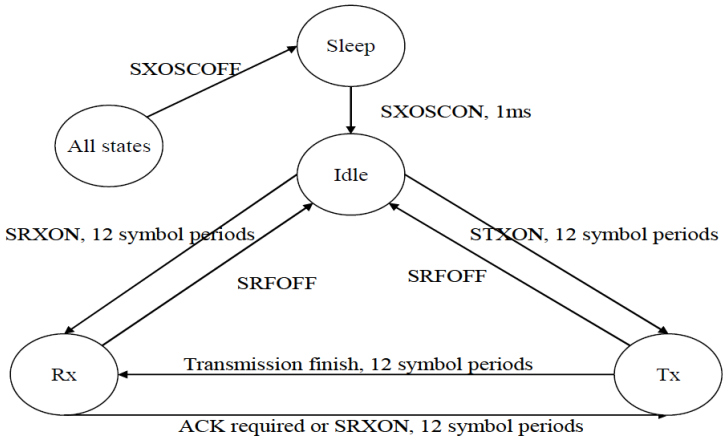

Figure 2.5: FSM model of a transceiver (Du, Mieyeville, and Navarro [12])

The use of power states is generally controlled through a FSM (Finite State Machine). In the work of Du, Mieyeville, and Navarro [12] a thorough explanation was given on the inner states and their dynamic, seen in Figure 2.5. Each given state has different current consumption, where they also tried to model the different consumption for the transceiver based on the dBm values, presented in Table 2.1.

Table 2.1: Current consumption values of the Transceiver model (Du, Mieyeville, and Navarro [12])

Mode	Consumption
Sleep	20 µA
Idle	426 µA
Rx	18.8mA
Tx(0dBm)	17.4 mA
Tx(-5dBm)	13.9 mA
Tx(-10dBm)	11.2 mA

2.3 Breakthroughs in Sensor Modelling

The idea behind modelling sensors is made on the aspect that the virtual sensors should maintain indistinguishable features compared to the real world counterparts. There exist different features which can be subjected to analysis from a sensor model. The main features are the ones directly related to the functionality of a sensor. As modern sensors themselves can also be seen as small computer-based machines, a question is stated, can sensors pass the Turing test? This aspect of sensors is discussed and analysed by the work of Siegel [40].
In the mentioned work, software agents are proposed which try to fulfil conditions that the sensor has low level adaptation, analog-to-digital conversion, linearisation and calibration, as well as network communication management. These aspects can be difficult to implement, due to the hardware and software constraints and developers time. The scientific work on Virtual Sensor, done by G. Hoekstra et al. [16], proposes the concept of modelling sensors to be done at the highest level of abstraction, so that it enables a sufficiently accurate characterization of the total system behaviour. The virtual sensors in a system should have an option to be validated.
While the research work concerning this thesis was written with using SystemC as the main modelling tool, other popular tools include *VHDL*

and *Verilog*. A VHDL model was specifically built for a Smart Sensor and described in the paper "VHDL Model of Smart Sensor" by Bagade and Limkar [4]. This work presents both the economical importance of developing smart sensors, as well as the VHDL approach for a potential FPGA synthesis. The model and simulation itself are fairly simple, mostly being focused on the analog-digital converter and the use of the FIR (Finite Impulse Response) filter. They also focus on providing the parallel simulation for the VHDL model, which showed similarity to the presented SystemC Smart Sensor model.

A more focused work on specific hardware component was placed on investigating how to model and simulate RF (Radio Frequency) devices, like NFC. Several works exist in this field, among others those by Navarro, Migliato-Marega, and Carrel [31]. Here, a NFC module was built in a special simulator by using a C# programming interface and tests were conducted by analysing if the auxiliary devices (microcontroller and a sensor) are able to fully operate given a certain power input rate from the energy harvesting. It was concluded that with the presently selected sensor and microcontoller, it is not possible to fully power the device by a small margin. This is very important since it shows how significant it is to provide a good model and simulation results during the development of a smart sensor.

2.4 Relations and Improvements

In the previous sections, short descriptions were given for the individual related work, many of which can be considered as state-of-the-art in the field. In this section, a brief overview of the relations and improvements is given between the SimPar solution presented in this thesis and the ones that have been researched and analysed.
Table 2.2 showcases general similarities and improvements compared between the SimPar and the related work listed and described in previous sections. "Similarities" lists elements which can be found both in the presented work and the related work, while "Improvements" primarily lists elements that are implemented or found in the presented work, but not in the work of the related research paper.

Table 2.2: Comparison in relation to the related work

Scientific paper	Similarities	Improvements
"*Scalably Distributed SystemC Simulation for Embedded Applications*", Huang et al. [21]	Have a master controller and slaves (sim. instances) controlled with a protocol	Proposed a network-based approach as opposed to the local one
"*parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures*", Schumacher et al. [39]	Linear speed-up compared to number of cores (or threads)	Possible use with only one CPU independent from the OS
"*Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations*", Aline et al. [1]	Both use the POSIX Threads	Different approach, focusing on the Behaviour and Cycle-accurate model rather than TLM
"*Battery-less Near Field Communication Sensor Tag Energy Study with ContactLess Simulation*", Navarro, Migliato-Marega, and Carrel [31]	Similar technology and the NFC chip is from the same company	Reduced focus on the analogue RF behaviour, while more put on the protocol and digital handling
"*Energy modelling in sensor networks*", Schmidt et al. [38]	Power states and calculation methods	Inclusion of more power states
"*Modelling Energy Consumption of Wireless Sensor Networks by SystemC*", Du, Mieyeville, and Navarro [12]	Use of power states, modelling similar hardware	Using newer energy consumption measurement approaches

3 Design

For every successful practical implementation it is necessary to research the theory behind its structure. In this chapter a strong notion is set on the scientific background behind both the first and the second phase of SimPar. Algorithms, individual methods, design patterns and methodologies, which were used and proposed, are analysed.

3.1 Client-Server Model Design

The first phase of the design is focused on envisioning and finally creating the design plan for the implementation of two applications:

- Server application - for handling the simulations
- Client application - for running the simulations

In general, both applications will contain the same network core, carefully constructed APIs (Application Programming Interfaces) which stay hidden behind a programming *façade*. The client application should provide a clear interface for integration with the SystemC library. The mentioned core library is not needed on the server side, since the server should operate on handling instructions through plain messages rather than having a direct input into the simulation work-flow. In other words, the server can change and influence which parameters define a particular SystemC instance, when the simulation starts and when it should end. The client side interprets the received commands and handles the simulation accordingly.

The application design will follow the principles of program design which includes class and sequential process design of the UML (Unified Modelling Language) and the network protocol behaviour.

3.1.1 Server-side Application Design

The server in SimPar behaves essentially as a stand-alone application designed to handle and instruct clients on how the simulation should be handled. Because of that, the server application should be able to be either run on the same machine, together with the client application instances, or on a separate computer. The server in this case fulfils the basic definition of a host machine which is able to handle request-response messages from the clients, as stated by Andrew [2].
The server has two main tasks with the following functionalities:

1. *Handle client connections* – initial, continuous thread
    a) Set up an initial configuration
    b) Wait and then connect an incoming client
    c) Handle client information
    d) Disconnect the client when the connection ends
2. *Handle client communication* – additional thread, for each client
    a) Keep track of the protocol steps
    b) Send and receive messages
    c) Interpret incoming messages and create own commands

During runtime, the server instance should enter an *infinite loop* with one thread or sub-process reserved for connecting and disconnecting hosts. Generally, it is aimed to make the server application as error-tolerant as possible, since it is advisable that the application should continue running even if a problem occurs with one of the client instances.

Handling multiple clients simultaneously

During the run-time of the application, the server keeps the information related to each individual client. That is done through keeping tab of the specific state of the client, from opening to finally closing the connection. Table 3.1 showcases what data is kept in the table for each individual client. "Client ID" represents the unique ID (Identification) given to each individual applicable client machine. "Ip address" and "port number" are unique identifiers for a *socket* and only one of those can be open at any given time.

"Status" is the current active state. "Simulation" is the identifier and label of the simulation which is currently tied to a client [1], while "Command" is the current command in the communication protocol.
Each client is uniquely identifiable with the specific socket properties (IP (Internet Protocol) address and port), but also with the ID number.

Table 3.1: Client data on the Server side

Client ID	IP Addr.	Port Num.	Status	Simulation	Command
1234	127.0.0.1	44380	State 1	Counter	sim. output
4652	192.168.3.1	44381	State 3	Smart Sensor	sim. start
2564	112.35.4.1	44382	State 2	NFC	read comd.
4658	192.168.3.1	44386	State 3	Smart Sensor	sim. stop
...	...	...	...	...	...

The communication between a client and the server follows the standard procedure described by Andrew [2], set by the protocol descriptions in the TCP/IP (Transmission Control Protocol / Internet Protocol) layer. After the initial configuration, the server will keep its listening port open and will wait for a packet which signalizes the start of the handshake procedure. The problem arises from the need to keep the program going for both the handling of the client connection and each individual simulation.
To make it possible for the server to, at the same time, listen to the new connections, but also handles the already established ones, it was necessary to use one of the following methods:

- Forking - by creating *sub-processes* for each new client
- Threading - by creating new *threads* for each new client
- Select - by using an interrupt handler and handling each individual client as a separate file descriptor

[1] One client can have multiple SystemC simulation models, but needs to create a new connection for each one of them

Forking can be applied by allowing in design to create a new sub-process, from the main process, each time a new client wants to connect. This new *child* process will enter the protocol loop, that is, the part of the code intended to cover the standard communication between the server and the specific client. The main process is essentially a loop which is blocking and is interrupted once a new request for the connection arrives. An advantage is that this method is easier to implement than other ones, but it can be a bit difficult to manage, since all created child processes retain the previous allocated memory during the initial stages of the server, even though they are not used on later.

Threading is similar to forking in sense that it creates a parallel and completely independent part of the program. The difference here is that the inner loop, which handles the new upcoming connections, is actually handled in a thread and thus is not a direct element of the main process. All new connections are also maintained through threads instead of child processes.

Select is different from both forking and threading in that it enables handling of multiple clients through the services offered with the C socket API. It ultimately sets a service which is blocking and which waits for an action from the socket, it being to read or to write. New clients are received by creating new sockets and each socket is labelled with an unique identifier. When a request is received, a function should be put into place which handles the checking of the unique identifier and handles the response accordingly.

Each of the described methods has its advantages and disadvantages, but generally it was decided to stick to the threads for the following reasons:

- Better portability and support between the OS
- No significant performance difference between forking and threading, Hauser et al. [20]
- More flexible than forking, and especially, than the *select* procedure

Both threading and forking are at an advantage by making it easier to construct fault-tolerant responses to sudden or planned disconnections, both by the server and client application.

Protocol for the communication between Server and a Client

Each potential client connected to the server is viewed as an individual entity. The main goal of each clients is to successfully connect to the server and to start a simulation session. But to start this procedure it is necessary to define a certain *communication protocol*. As stated by Bonaventure [8], a protocol defines rules for the communication between two hosts. It helps in determining the current state[2] during the communication, any potential errors, as well as by adding flexibility in moving between the current, previous and future states.
The flowchart which presents the communication protocol designed to be used in SimPar can be seen in the Figure 3.1. The cases written (Case 0, Case 1 ...) encapsulate the entity cases of the execution, but also the switch cases designed to be used in the actual program implementation.

The message protocol is made of three main steps:

1. *Handshake*
2. *Commands and parameters exchange*
3. *Simulation handling*

The communication session between the server and a client starts with the server accepting the client request for the connection through the *handshake* procedure. To establish the connection it is necessary for the client to send an appropriate *connection code*. This code is used to identify the client, but also to set an appropriate simulation command framework. If the received code is wrong[3], the protocol will immediately terminate. This step also serves as a security measure.
The connection codes are build upon the following pattern: 1xxx, 2xxx, ...
The first number in the connection code dictates the type of the simulation. Examples: 1231, 2564, ...

The second phase of the protocol is focused on exchanging messages between the client and the server concerning various commands and parameters. Each packet contains a certain command message. The commands

[2] a state refers to both the status of the current message to be anticipated and the one to be sent

[3] not beforehand registered on the server

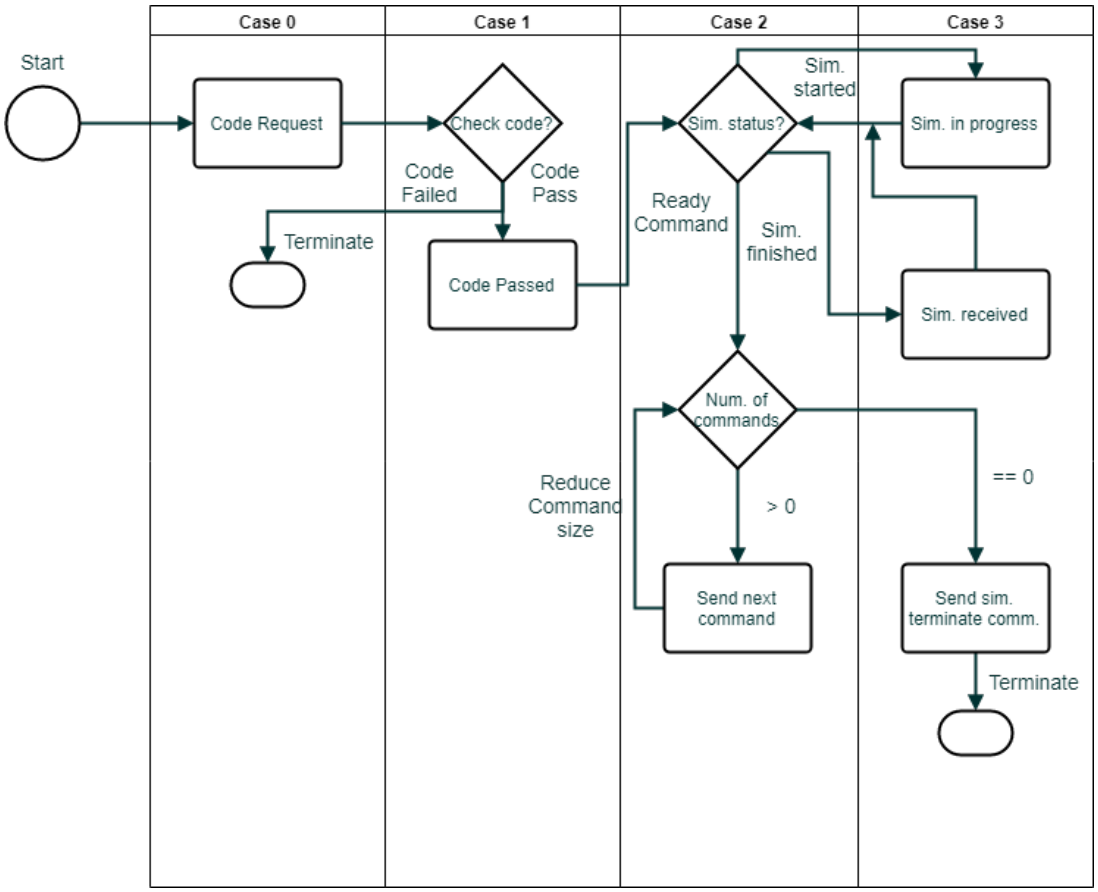

Figure 3.1: Message protocol used between the server and a client in the SimPar solution

depend on the simulation model and have to be defined by the designer, but more than often they consist of various values used for the parameter definitions. This step consists of *Case 2* and will continue until a special command is given. For each iteration, the client sends a command, the server interprets it, and sends back an acknowledgement message.

Finally, after the initial parameters and other command inputs have been received, the client will send a special command to start the simulation. This is indicated through the *Case 3* as seen in Figure 3.1. Any command received during the run-time of the simulation will not be registered. Only when the simulation is finished will the server notify the client. Afterwards the client can either repeat the second state of the protocol by sending additional commands or it can send a special command to terminate the communication, ending the protocol session.

3.1.2 Client-side Application Design

The Client is designed as a host which handles the simulation. It contains one (or several) SystemC model(s) which runs and configures the simulation based on the commands that was received on input.
The Simulations are independent and handled by the *factory method* based classes during run-time. That is done through the standardized approach of configuring the simulation and the trace file (signal tracing). Each Client needs first to connect to a running Server with the correct IP address, port number and its own connection code. After the connection is successfully established, the protocol handles the communication and simulation running on both sides. Figure 3.2 illustrates that graphically.

The focus was kept on developing the same interface for all simulations. That includes setting up the connection, reading commands and passing them to the simulation. The framework, around which the client application is designed, is based on a similar principle like the server framework, with some additional simplifications and changes.

The client application would consist of implementation classes grouped in three categories: socket communication, model and simulation, and extra tools. The socket communication classes would be very similar, some

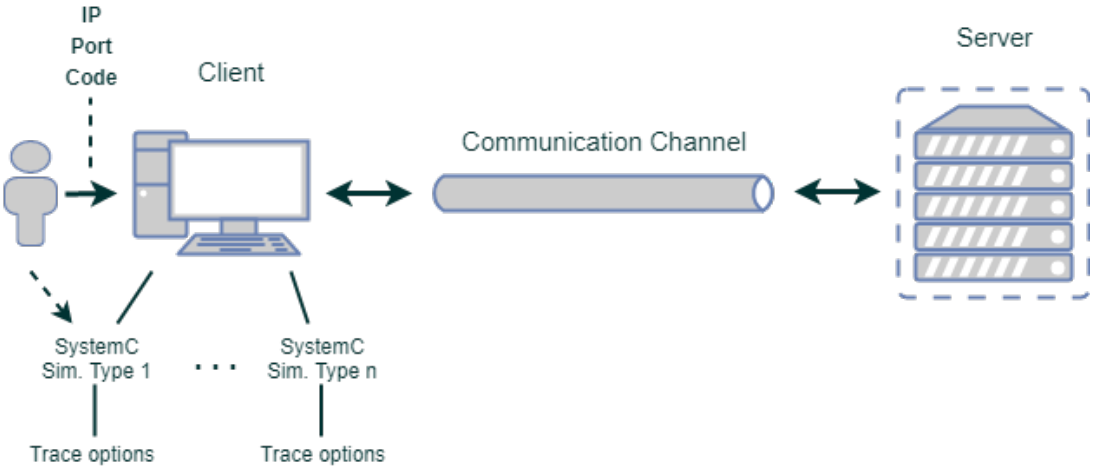

Figure 3.2: Client host initialization and communication

identical, to the server application classes, since they use the same template. The main difference would be in how the client behaves by setting up a connection.
The following steps are taken when setting up a new client host process:

1. Set up a network address through the initial configuration
2. Define the socket interface by connecting to the already known server address (IP address and the port number)
3. Call the "socket handler", which is essentially a class which offers the services for setting up and maintaining the connectivity to the server

Before the network initialization commences, the client should set up the appropriate simulation. That is already predefined by the input code given to the start of the client application. Based on this code, the server will also automatically known which current SystemC model is being run on the client side.
After the simulation ends (SystemC model ends its session), both the simulation objects (object model, trace files, etc.), as well as the network objects should be handled and closed accordingly[4].

[4]Necessary both for memory deallocation since it is a C++ application, but also for safely saving the results

SystemC model integration handling

The handling of individual SystemC models has been designed in the following way. Each separate model should be easy to include. All the necessary simulation files would be contained inside one source folder. Inside this folder it should be possible to insert any number of additional sub-folders, each dedicated to one individual SystemC model. A model can be developed independently from the client application and when it is done and ready, it can easily be integrated into the whole framework.
The integration of new SystemC models should heed the following steps:

1. Create a new sub-folder inside the main "simulation" folder and copy the whole source SystemC model code inside the newly created sub-folder
2. Create a new C++ "Handler" class inside the predefined "sim_handler" sub-folder which should inherit the general simulation virtual class for implementing the necessary functions
3. (optional) Add any additional functions or signals to the newly created simulation class. The class should consist of the signals and macro-modules which are able to be interacted with during the client-server session for parameter modification and changes
4. Create the "Top" class for calling and handling the SystemC simulation. This class inserts a virtual class named "Simulation" for defining the necessary members and functions
5. Extend the factory simulation function by adding a new code for the newly created simulation identification, as well as the protocol for message handling and control

Adding new simulation handlers is simple, but it creates a new layer of complexity by introducing new designs and protocol definitions for individual simulations every time a new simulation is introduced. That has to be done, since SystemC models can be developed in various ways and introducing one standard can be very limiting. It is possible however to develop a SystemC model which would follow the guidelines of the simulation handling from one of the already integrated models and thus reducing the time and effort needed to adapt new simulations.

3.1.3 Design Patterns

The program build behind the client-server interaction follows a specific coding standard. Many of the elements were adapted to be easily read and further extended if the need arises. That was done for the purpose of allowing other developers to use the given template of SimPar and apply it to their own models.
To fulfil this goal, the design of the system follows two main principles:

- Adding extensibility to the system through the use of *design patterns*
- Making the system portable by handling the cases of different *operating systems*

Since the system optimisation for making it runnable under different OS is seen more as an implementation problem, in design only the specifications of the used design patterns are going to be mentioned.
What are design patterns and why were they used? There exist many definitions, but in general they represent a blueprint used as an over-layer for a general construction model. They can be used in many different fields, from psychology to architecture, but in this thesis they are generally associated with the software. As stated in the work of Gamma et al. [17], each design pattern consists of four elements: *pattern name*, *problem*, *solution* and *consequences*.
In this section, each design pattern applied during the development of the client and server applications of the SimPar system are explained.

Wrapper Façade

The program was designed and implemented using extensively the wrapper façade design pattern. First described and standardised in POSA2 (Pattern-Oriented Software Architecture) in Schmidt et al. [37], this pattern is used to describe the encapsulation of the functions and data provided by existing non-object-oriented APIs (Application-Programming Interface) with a more object-oriented, extensible, robust, portable and maintainable interface.
Since both the server and the client side use the sockets for in-between communication, the actual development of socket functionalities follow

Unix-based C functions. Because the programming language C is not object-oriented, to make the use of the functions more robust and open to multiple classes and uses, an approach was needed to make this development easier. General guidelines were followed set in the paper Schmidt [36], which describes the implementation of the wrapper façade in the general C++ object-oriented environment.

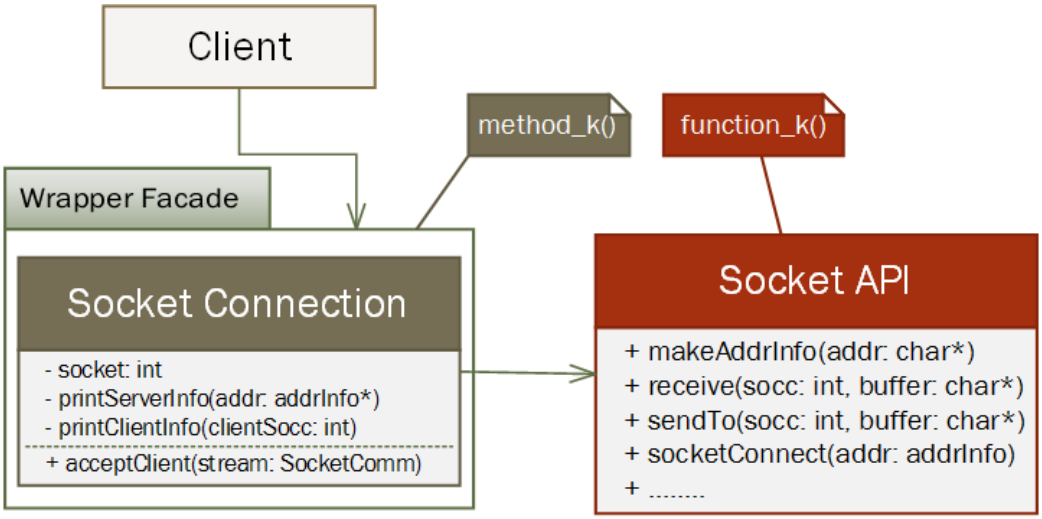

Figure 3.3: Initializing socket connection through a wrapper façade design

The way how this design pattern is used can be best seen in the Figure 3.3, which is an abstract representation of an example design of the used Socket class and the C API interaction. Essentially, C Socket functions are all placed and implemented in a separate file. This file can be manipulated and updated with new functions independent from the rest of the application. The functions which are implemented follow the C programming guidelines. Separately, individual classes are implemented in C++. Each of these classes can be used to fulfil a different Socket protocol role, from establishing the connection to sending and receiving data. Classes also consist of own functions which in turn call the C API functions. By following this approach, the user who uses the classes to initiate the objects in his code does not need to know, or even make direct interaction with the C API classes. That allows for an easier development and understanding of the code, as well as inner-correlation between individual programs written in different programming

languages (in this case those being C and C++).

Wrapper façade was also used to enable certain classes to be independent from the operating system in use. The examples are threads and classes which provide functions that are independent of the programming platform. In this way, an user extending his own program with program written for different os can easily call a function or initiate an object without explicitly stating how the code should be called and run. That is done by specifying different code segments of the code which are dependent on the os that they are run and compiled during the compile-time. An example on how this approach is defined, which has also been used during the development of SimPar, can be seen in the work of Schmidt [36].

Singleton

First stated in the primer book of the design patterns commonly known as GoF (Gang of Four, Gamma et al. [17]), singleton is described as a *creational* design pattern.
It is a simple pattern used primarily for two purposes:

- Ensure that a class only has one instance and a global access to it
- The defined instance should be initialized on the first use

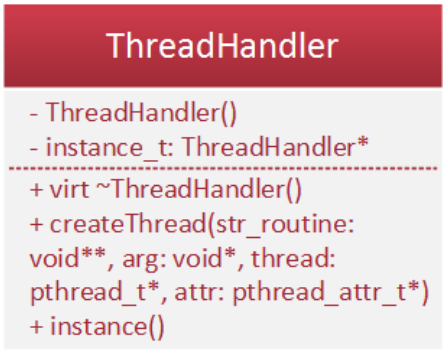

Figure 3.4: Thread Handler class designed as a singleton class

In terms of the overall design, singleton was designed to be used on several occasions, primarily to allow global access of functions and variables used

by all different dependent and independent classes. Singleton was also heavily used to enable the use of the global functionality of service and tools. Table 3.2 shows different use-cases of the potential singleton classes and for which application side (server or client) they were designed, while Figure 3.4 presents a concrete design solution used for a class designed to handle thread creation based on different operating systems[5].

Table 3.2: Design of the singleton design pattern in SimPar

Class	**Description**	**Application side**
Message handler	Used for checking client connection code and message protocol	Server
Stylist	Terminal colour log printing (multi-platform)	Client & Server
Thread handler	For thread creation, only one instance allowed	Server
Hash Map handler	Creation and handling of the hash maps used for client connections	Server

While singleton design pattern offers a good quick solution for otherwise closed object-oriented problems, it also comes with its own critiques and issues. Many considered this design pattern to be actually an anti-pattern. As described in Bhardwaj [6], singleton can be easily misused, since global initialization is worse on the performance and unit testing, and program flow control can be more difficult to set.

Strategy Pattern

To further increase the extensibility and make an easier use on the user side of the application, a design proposal was made to incorporate strategy

[5]Between Windows and Linux solution and Solaris solution

pattern as well. This design pattern is build on the principle which allows the user to call and use a specific algorithm, but with hiding the implementation details. Programs set in this way are open for extension, but not for modifications. As stated through the general definitions found in the work by Gamma et al. [17], the strategy design pattern is used when there exist closely based implementations of the same general action. In this case the user is only concerned with running an appropriate function and not with the type and implementation of that function on the other side of the program.

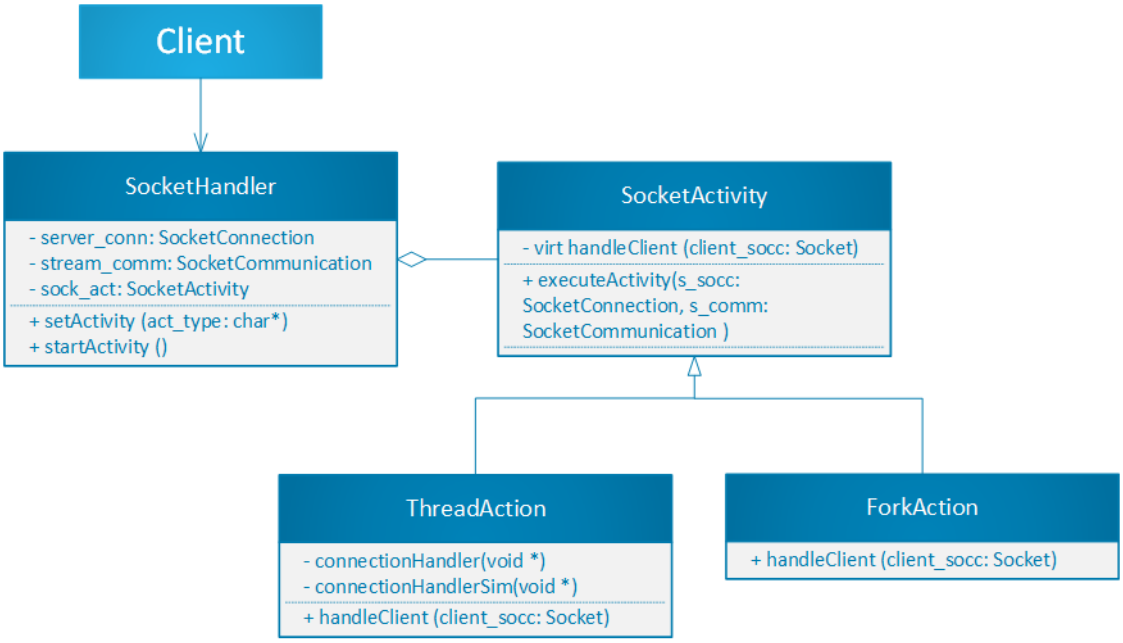

Figure 3.5: UML class diagram of the strategy design pattern used for the design of server client handler

In the context of the SimPar project, the strategy pattern was applied during the design of the network socket handling of users on the server-side application. To allow for a parallel and simultaneously multiple-handling of clients, several of the implementations techniques were open to be implemented. As mentioned previously, two main techniques were implemented and those were forking and threading.

To allow the program to be independent of the os [6] and to allow for a different performance evaluation, a central function for handling the client was implemented, where the approach used is determined during the run-time. This is done generally through the command line and can be changed any time the server application is run again. By using this approach, a certain level of flexibility and usability is added, where any changes done to the main user-handling module are independent of the handling techniques being used (in this case, forking of threading). Figure 3.5 shows the design of the class diagram that was used in the project.

Factory Method

The design of both the server-side and client-side applications were aimed at creating a development environment which would allow for an easier use of different functions and their instantiation through objects. One of the design patterns aimed at fulfilling these requirements is the *factory method*.

Listing 3.1: Initiating correct simulation object using the factory method function

```
simc::Simulation *sim = factoryMethodSimulation(connCode);
sim->initializeSimulation();
sim->setTraceName(connCode);
sim->processTraceOptions();
```

The factory pattern is used in several occasions, mostly to provide different object handling, e.g. different SystemC model instantiation. Client-side application uses the benefits of the factory method in its design. Each simulation model has its own handler and a different set of messages used in its protocol. The instantiation of the simulation object happens during the run-time, where it is only necessary to provide the code for the specific simulation. In fact, based on the simulation code, the application will automatically initialize the correct object and assign all the required functionalities to it. A snippet of that code can be seen in Listing 3.1. The full class design is presented in Figure 3.6. Here, two different simulation

[6] Primarily Windows, Linux or some other Unix-based system

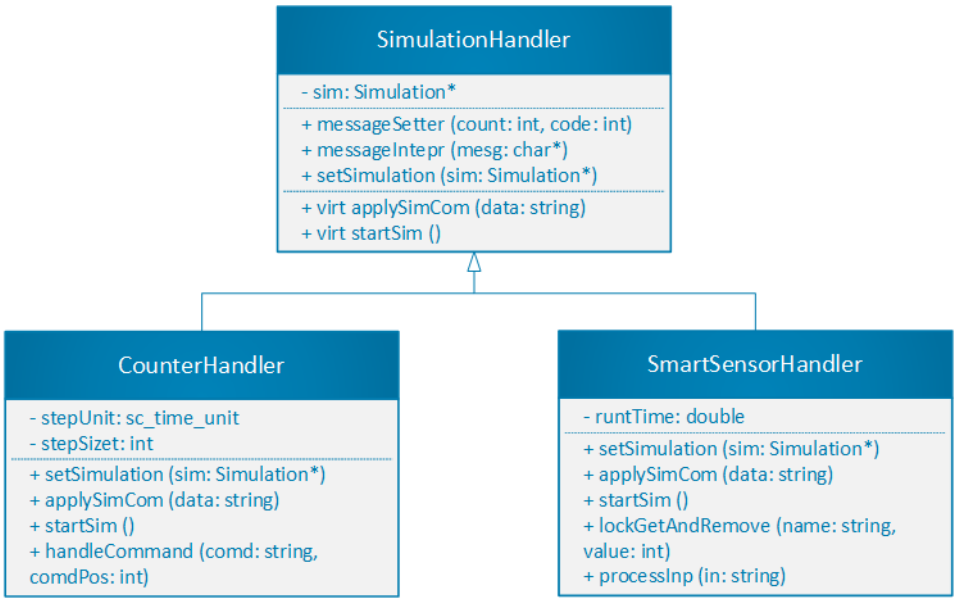

Figure 3.6: UML class diagram of the factory method design pattern used for the handler of different simulations

handlers were initialized, one for the "*Counter*" and one for the "*Smart Sensor*" simulation model. Due to the flexible nature of the design, it is possible to easily add additional SystemC models by attaching new handlers to the factory method class named "*Simulation Handler*".

Scoped Locking Idiom

Scoped locking is a software idiom primarily used to ensure that the allocated resources set for a specific action during the run-time (like mutex) that is synchronisation-dependant, are also released once the hold on that action ends[7].
While the definition of the context, problem and solution is similar to the concept of design patterns, idioms are different in the way that they are confined to a smaller context, usually under certain constrains and more than often, tied to a specific programming language. The definition of idioms

[7] i.e. once the mutex is released

was first introduced in the GoF book (Gamma et al. [17]) where examples are presented with the use of the C++ programming languages.
An extension to the Scoped Locking Idiom is called the "*Guard idiom*". As stated in Crahen [9], this idiom is very similar to the scoped locking in nature and definition with the main difference being that the guard idiom is designed to allow for "Guards"[8], to cooperate with one another by introducing more flexibility. The flexibility is achieved by allowing locking scopes to not only be tied with the creation and destruction of an single object, but also trough other means.

Listing 3.2: Guard class inside the Server-side application

```
template <class LOCK>
class Guard
{
public:
        Guard (LOCK &lck): lock (lck) {
                lock.acquire();
        }
        ~Guard() {
                lock.release();
        }

private:
        LOCK &lock;
};
```

Scoped locking idiom, or more precisely, a simpler design of the Guard idiom was designed to be used in conjunction with the threads for an easier maintenance of the mutex allocations. The design code template which is used in the server-side application is shown in the listing 3.2. Guard idiom adds a certain layer of flexibility, where the mutex lock is automatically released when the object associated with it goes out of scope. By using this approach, it is not necessary to always implicitly specify when the lock needs to be released, but rather, that is done automatically.

[8] classes and objects designed with the Guard idiom

3.2 SystemC Smart Sensor Model Design

The implementation and construction of a real-world hardware device is generally being preceded with a hardware model design. For the purpose of showcasing the *rapid* development introduced by the idea of SimPar, a "*Smart Sensor*" has been designed. A smart sensor represents a solution for the new sensor designs used in industry, designed to provide multi-functional, energy-efficient and autonomous working environment. With these goals set, it is not a simple matter to assemble such a device due to the manufacturing cost. As stated by Yong et al. [45], it is generally advisable to also provide a software model for simulation purposes, where different ideas and modifications can be put into place.
This chapter will go more into the detail on the proposed design of a smart sensor which is composed of multiple parts and hardware elements.

As part of the project, the proposal for the general model design has been made which suggest that the designed smart sensor should fulfil the following conditions:

- **Energy-efficiency** - all hardware elements should either adhere to the low-power, or to the more modern, *ultra* low-power energy consumption design
- **Security** - the smart sensor should provide security functions for handling data
- **Wireless communication** - provided with the NFC technology
- **Port extensions** - allow for the option to add custom sensors through the extensible ports

By respecting the constraints given from the projects, a design was drafted consisting of the following main hardware components: a microcontroller, NFC tag, security chip, FRAM (Ferroelectric Random Access Memory) chip, and two extension ports. The interconnection between individual parts can be seen in Figure 3.7.
The microcontroller was added to ensure the programmable support for the smart sensor device, while the FRAM is there for further support in case a large amount of data is needed to be stored. Currently, it has been

proposed that all devices should communicate internally by using the I2C (Inter-Integrated Circuit) *bus technology*. Additionally, since the smart sensor is being designed to offer service independent of the actual sensor in use, potential devices connected to the external ports will not be actually designed.

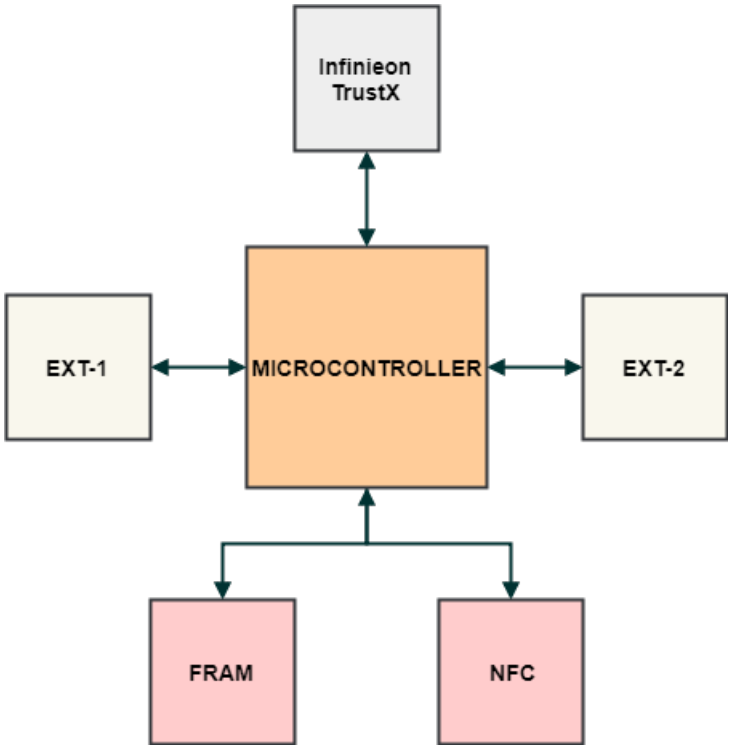

Figure 3.7: Overview of the hardware elements of the smart sensor and their interconnections

Table 3.3 shows the list of the hardware elements and chips decided to be used as a *reference model* for the SystemC model. By using concrete elements with technical documentations, it is possible to make easier predictions for the overall system behaviour and energy consumption. For the most part, the SystemC model should be independent of the construction components. They are only used to provide a general validation for the functional accuracy and real-world device behaviour.

To fulfil the quality and the functional accuracy of the model, a set of *test*

Table 3.3: Hardware components used as a reference during the modelling of the smart sensor

Hardware	Reference Component	Description
Microcontroller	Apollo2	One of the best MCU (Micro Controller Unit) in terms of the ULP (Ultra Low Power) capabilities
FRAM	MB85RC1MT	Non-volatile memory, low power consumption, fast, I2C
NFC Tag	ST25DV64K	Inner memory, I2C, low power consumption
Security Chip	Optiga TrustX	Wide range of security functions, easy to integrate

benches was proposed to be designed and implemented. The test benches should be designed in a similar approach like the corresponding interactive module for the specific design module. An example would be the test bench for the NFC tag module. This test bench tries to mimic both the complementary parts of the microcontroller, which uses the I2C (Inter-Integrated Circuit) technology, and with an external NFC card reader device for the RF communication. Each test bench would include a number of specific *test scripts* for different evaluations, but primarily for the functional-accuracy, time-accuracy and energy consumption.
Test benches are implemented for each individual macro module. More on them, as well as on the particular tests, can be read in the "Evaluation" chapter.

3.2.1 Design Methodologies

This subsection is devised as a counterpart to the "Design Patterns" subsection of the previous phase I design analysis. Since SystemC is described as a HDL utility, many of the design patterns used generally with the programming languages can also be applied here. There are however differences and the only design pattern used, which is also found in the "Gang of Four" book, is the *singleton* design pattern. Other approaches used in defining the design of the SystemC program can be described as common methodologies used in this field.
Two main methodologies used in designing the SystemC model in SimPar are:

- Hierarchical module definition
- Defining the complex functionalities at a more abstract layer

Hierarchical module connections

This approach is used to represent the modules of a SystemC model as a tree. The root of the tree is essentially a module known as the "Top" module, while the nodes are main modules. The children of these nodes are sub-modules of the according parent module, a notion described by De Graaf [10]. This topology helps in tracking down potential issues that can arise during the development and testing phases, and in helping extending the SystemC model, especially once it gets more complex.
Figure 3.8 shows how the modules are interconnected in the SimPar SystemC model. Each macro-module (the module which comes directly after the top module) is described as its own chip and can easily be developed separately compared to the other modules.

As already described, the top module is defined as the root of the tree representation. Usually, this module only has one port which is the input *clock* signal. This clock signal, which originates from the calling *sc_main* function, represents the reference clock signal on which other input signals are defined. The other elements found inside the top module are primitive signal channels and instances of modules which are connected in between. The modules represented at the top level are called "macro modules" since

they are the main construction blocks of the SystemC model. Each sub-module that is defined inside the macro modules is a child to its parent. These leafs can contain additional sub-modules which would, also in this case, only be dependant on the upper parent module.

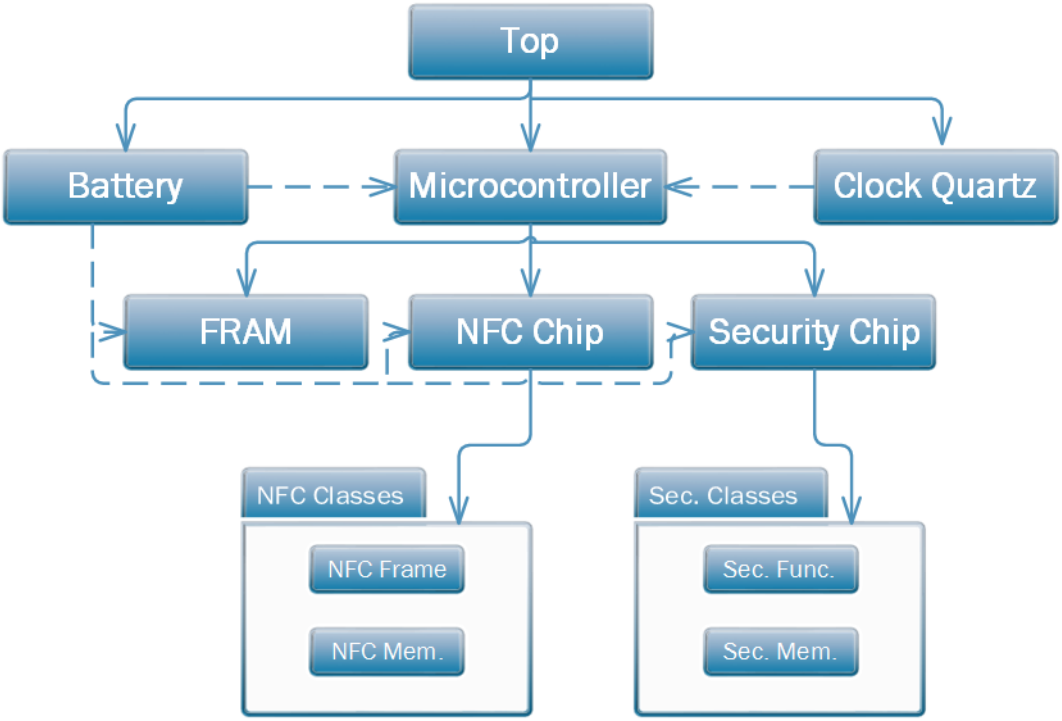

Figure 3.8: Design of the module hierarchical dependencies using a tree-like structure

There are generally no drawbacks in using the hierarchical approach compared to some other approaches, except for the fact that additional complexity can arise when one or more changes are needed to be done on the macro-level. Additionally, there is a delay of one delta cycle through the signal transmission between the modules which has to be taken into the consideration during the development.
Adding or removing a macro-modules is not difficult, but making changes either with the connections (signals) or functions can lead to the subsequent change for adaptation to every child module.

Function distribution among the different layers of abstraction

Each SystemC module is a representation of a real-world hardware element. Many of these elements are already grouped into one structure known as chip. Hardware chips can contain both a complex design and a complex functionality. Integrating all the different functions and behaviours of a chip inside a SystemC module can prove to be difficult. It is almost necessary to divide the functions among those which can potentially be emulated and those which are only going to be represented through a black box, by providing input values and expecting specific outputs.
How the black box is represented depends on the overall design. SystemC allows for an easy extension through standard C++ classes and structures. Each module can have a specific class dedicated with supplying different functionality options. Input values would be inserted through the parameters of function, while the function would return values either through the standard return value or through references.
As mentioned in Doucet and K. Gupta [11], by following this design approach, a SystemC module which describes a complex function would only need to call and manipulate the function on the right execution step, while still maintaining the important simulation parameters (execution time and energy consumption among others) and the overall hardware structure.

The implementation of individual functionalities during the development of SimPar was considered as part of the macro-modules. Since the FRAM module is modelled as a full hardware emulation, it was not necessary to make any additional functionality extensions. On the other hand, the NFC module was designed with an additional memory class to hold and work with memory data on the pure application abstraction layer. Similar changes have also be done for the microcontroller module, and especially the *Security chip* module since it itself had a very abstract documentation.
The security module should host a separate interface to the functional part and the memory part. With this approach, it makes it easier to divide between the data that is saved and that is in operation (being worked on). Further, it offers extensibility by presenting an easier access to individual code modifications in case they are necessary to be implemented in the future. The microcontroller follows a similar principle, but the main focus is still kept on the SystemC elements, rather than the C++ extension classes.

Table 3.4: Functionality extensions of the abstract layer in SimPar design

Macro-module	Application
FRAM	No direct use
NFC Tag	Memory handling & Frame operations
Security Chip	Memory handling & Security operations
Microcontroller	Program representation & modules interaction

Table 3.4 shows an overview of where individual application functionality extensions have been applied. A more detailed analysis of individual elements can be found in following subsections of the "Design" and module-focused "Implementation" sections.

3.2.2 Memory Management

For the purpose of storing data over longer periods of time, a permanent memory service was proposed. This is achieved by using FRAM technology. This type of memory is non-volatile, meaning that it makes it possible to store data even when the power is turned off from the energy source[9]. The design is based on the "MB85RC1MT" chip, where the specifications taken for building the SystemC model are explained in the Table 3.5.
The module is to be implemented on the system level, meaning that it is aimed to provide time/cycle accuracy, as well as the hardware functional accuracy. That is primarily done through the I2C function, which is to be completely implemented as a SystemC thread and controlled via the input SCL signal. Each state changes on the clock change and usually consists of an operation on the bit level. That includes sending, receiving data bits, and special ACK bits as well.

[9] i.e. the power coming out from the battery or other sources

Table 3.5: Listing of the target features for the FRAM SystemC module

Feature	Values	Handling
Operating frequency	3.4MHz, 1MHz, 100kHz	Operating clock for the read/write functions of the I2C protocol
Bit configuration	131,072 words x 8 bits	Defined array of fixed length for storing data
Communication	I2C with SCL & SDA pins	As thread functions defined on a multi FSM principle
Operating voltage	1.8V to 3.6V	An "enable" signal which controls the on/off state of the module
Current consumption	0.71mA to 1.2mA	Consumption during the active state

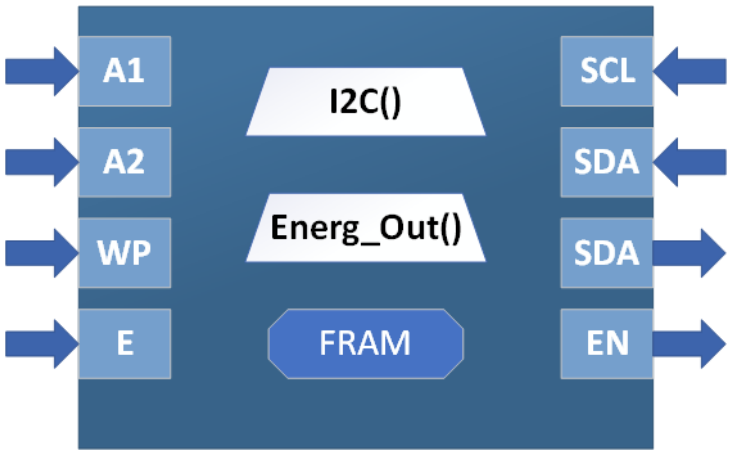

Figure 3.9: FRAM design of the SystemC module

Figure 3.9 indicates which signals are to be implemented in the FRAM design, as well which functions (i.e. threads) should handle the functional part of the module throughout the signal interactions.

The FRAM module consists of the following input/output signals:

- □ A1 - First bit used for the identification of the unique FRAM chip
- □ A2 - Second bit used for the identification of the unique FRAM chip
- □ WP - Control bit for the writing process
- □ E - Enable signal, for operation control
- □ SDA - Input signal from the master device of the I2C process
- □ SCL - Clock input signal of the I2C process
- ■ SDA - Output signal to the master device of the I2C process
- ■ EN - Current energy output, for measurements

Threads/functions designed to be run as part of the FRAM module:

- **I2C()** - main I2C process which emulates and offers read/write capabilities of between the master (microcontroller) and slave (FRAM)
- **Energ_Out()** - calculates and outputs through the EN signal the current output based on the ongoing energy state

FRAM is primarily used in conjunction with the microcontroller module. The connection is maintained through a separate I2C protocol, where the microcontroller plays the role of the master and the FRAM module of the slave. The current consumption is calculated through the energy states which are defined as part of an abstract functionality design layer.

3.2.3 NFC Communication Device

The NFC design is based on the specifications set by the energy-efficient NFC chip tag from STMicroelectronics, as described from ST25DV64K [42]. It is capable of communicating with other devices using the RF communication based on the ISO (International Standard for Organization) 15693 standard, as well as the communication using the I2C protocol. In both cases they include the behaviour of the NFC tag as a slave, where the communication would be performed with the microcontrollers through the use of the I2C protocol and RF with any other device (to be designed separately). Closer inspection of the specifications can be seen in Table 3.6; where the proposed SystemC model is seen in Figure 3.10.

Table 3.6: Listing of the target features for the NFC SystemC module

Feature	Values	Handling
Operating frequency I2C	1MHz, 100kHz	Operating clock for the read/write functions of the I2C protocol
Operating rate RF	up to 53KBit/s	Data rate for the read/write functions of the RF
EEPROM	64KBit (8KBit for imp.)	Memory which can be used for storing user data
Write time	5ms	time needed to write new data per block or byte
Mailbox	256bytes	used for the fast data transfer between I2C and myacroRF interfaces
Energy Harvesting	Pin based	Used to supply analogue voltage and current
Operating voltage	1.8V to 5.5V	An "enable" signal which controls the on/off state of the module
Current consumption	0.1uA to 0.5uA	Consumption during the active state

NFC module consists out of the following input/output signals:

- □ SDA - Input signal from the master device of the I2C process
- □ SCL - Clock input signal of the I2C process
- □ E - Enable signal for operation control
- □ RFI - Input clock signal for the incoming radio frequency data rate
- □ RFO - Input clock signal for the outgoing radio frequency data rate
- □ RFE - Enable signal for RF communication
- □ ANI - Input data in bits from the RF antenna
- ■ SDA - Output signal to the master device of the I2C process
- ■ GPO - Special interrupt signal

- VEH - Voltage output towards the connected device[10] during energy harvesting
- RFE - RF enable signal during the process of sending data
- ANO - Output data in bits from the RF antenna
- EN1 - Current energy output for measurements, from the I2C based processes
- EN2 - Current energy output for measurements, from the RF based processes

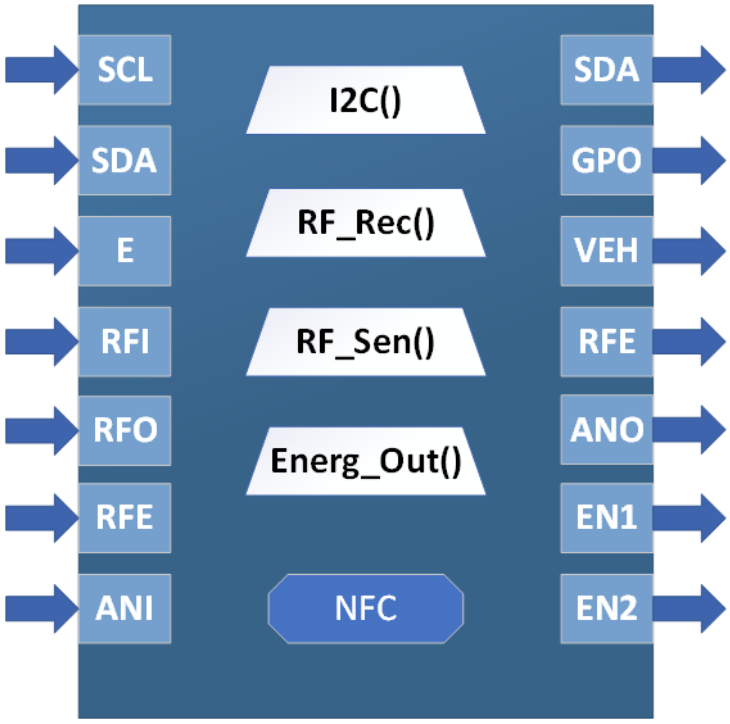

Figure 3.10: NFC design of the SystemC module

Threads/functions designed to be run as part of the NFC module:

- **I2C()** - main I2C process which emulates and offers read/write capabilities between the master (microcontroller) and slave (NFC)
- **RF_Rec()** - RF handling thread processed on the incoming frame

[10] in the overall design, a microcontroller

- **RF_Sen()** - myacroRF handling thread processed on the outgoing frame
- **Energ_Out()** - calculates and outputs through the EN signal the current output based on the ongoing energy state

The design, similar to that of the FRAM chip, is based on the *system* behaviour of the model. That means that a certain level of emulation is introduced into the model design by trying to replicate the time and functional accuracy. This is done through both the accurate representation of the I2C protocol handling and the RF handling, by using correct transmission cycles, time delays between messages, and to a certain degree, accurate representation of the inner data processing time.
The module is supposed to offer the following main functionalities and services with:

1. *Direct transfer* - Though the use of the Mailbox between the devices using I2C and RF communication, up to 256 bytes
2. *Memory capabilities* - Saving data as the user memory for a certain period of time
3. *Energy harvesting* - Supply both itself and optionally other devices through the energy harvesting principals using the NFC technology

Since the NFC module is capable of providing different memory capabilities (mailbox, user memory, system and dynamic memory), it is necessary to provide a clear design based on the functional abstract programming approach during its implementation. The same principal is to be upheld as well during the care of the RF communication protocol, since it contains special rules of handling.

3.2.4 Security Handling

The proposed Smart Sensor design was build upon an idea of offering fast, reliable, energy-efficient, and among others, *secure* solution for the data processing. In modern hardware design, security is primarily handled by a separate security chip. A chip like that can offer several security functionalities, it being independent of the data and process for what the overall hardware is designed for.

The design which is to be implemented using SystemC is based on the proposed characteristics of the "Optiga TrustX" security chip, described in Optiga manual [32]. Table 3.7 shows main adapted features. Since the chip presents a complex hardware design, one that is difficult to implement using a system approach, it has been suggested to design the security module to be cycle-accurate on the communication level, while the time and functional accuracy of the security processes should be kept at an abstract level. The memory capabilities of the chip are also to be designed on an abstract level, offering only basic functionality capabilities through a C++ class extension.

Table 3.7: Listing of the target features for the Security Chip SystemC module

Feature	**Values**	**Handling**
Operating frequency	1MHz, 100kHz	Operating clock for the read/write functions of the I2C protocol
User memory	10kBits	Handling permanent and non-permanent data
Communication	I2C with SCL & SDA pins	As thread functions defined on a multi FSM principle
Security functions	Crypto toolbox	Offers different security functions based on hashing, signatures, etc.
Operating voltage	1.6V to 5.5V	An "enable" signal which controls the on/off state of the module
Current consumption	20mA	Typical active state

Since the documentation provided for the specified hardware chip does not clarify how the security aspect of the functions is being handled, a special protocol is proposed to be implemented. That protocol should offer command interpretation from the data received through the I2C protocol. A special process should handle the command and the security aspect. Representation of the signals and inner processes can be seen in Figure 3.11.

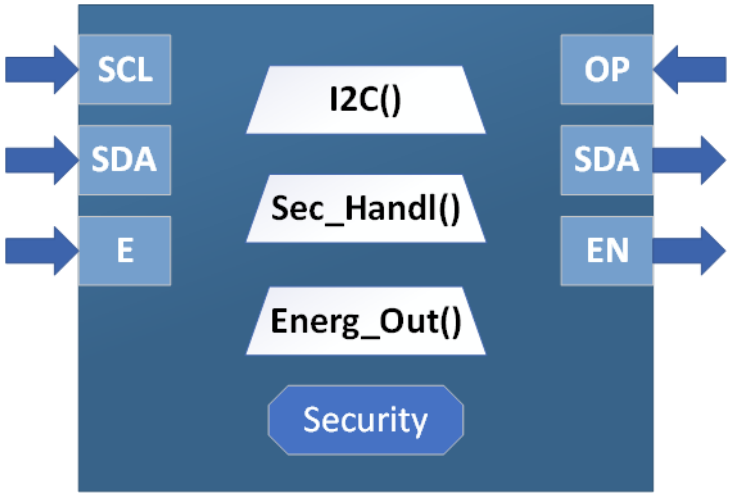

Figure 3.11: Security Chip design of the SystemC module

The following input/output signals are defined:

- SDA - Input signal from the master device of the I2C process
- SCL - Clock input signal of the I2C process
- E - Enable signal, for operation control
- OP - Control signal, for signalling if there is enough power for the device to operate
- SDA - Output signal to the master device of the I2C process
- EN - Current energy output, for measurements

Threads/functions designed to be run as part of the Security chip module:

- **I2C()** - main I2C process which emulates and offers read/write capabilities between master (microcontroller) and slave (security chip)
- **Sec_Handl()** - Functional handling of the security functions
- **Energ_Out()** - calculates and outputs through the EN signal the current output based on the ongoing energy state

The security chip offers thirteen different operations ranging from simple hashing to signature creation and verification. These operations are mostly of no concern for the target simulation and they are to be implemented in a simplistic way. That is primarily to be handled though a black box,

where the output is independent from the input and is only composed of a random stream of values.

3.2.5 Main Control Unit

The central element of the Smart Sensor is a MCU. For the design of an ultra-low power MCU, a MCU from the "Ambiq Micro" company was chosen, named Apollo2 [3]. The key features integrated in the design can be seen in the table 3.8.

Table 3.8: Listing of the target features for the Microcontroller SystemC module

Feature	Values	Handling
Operating frequency	48MHz, 1MHz	Operating clock for general MCU purposes
SRAM	1MB(1KB)	Handling volatile process data
Communication	I2C with SCL & SDA pins	As thread functions defined on a multi FSM principle
Operating voltage	1.1V to 5.5V	An "enable" signal which controls the on/off state of the module
Current consumption	10uA	Typical active state

The MCU is designed to contain three main I2C threads/processes, one for each of the individual connected module. In every case, the respective modules function as slaves[11], while the MCU takes the role of the master. Other processes designed to be implemented are:

- **SCL_Gen()** - Generates the SCL signal for the individual modules in the I2C communication
- **Prog_Contr()** - FSM implementation and control of the MCU program
- **Energ_Out()** - used current in the MCU used for the energy calculation

[11]Only masters can initiate and control the communication, slaves only respond

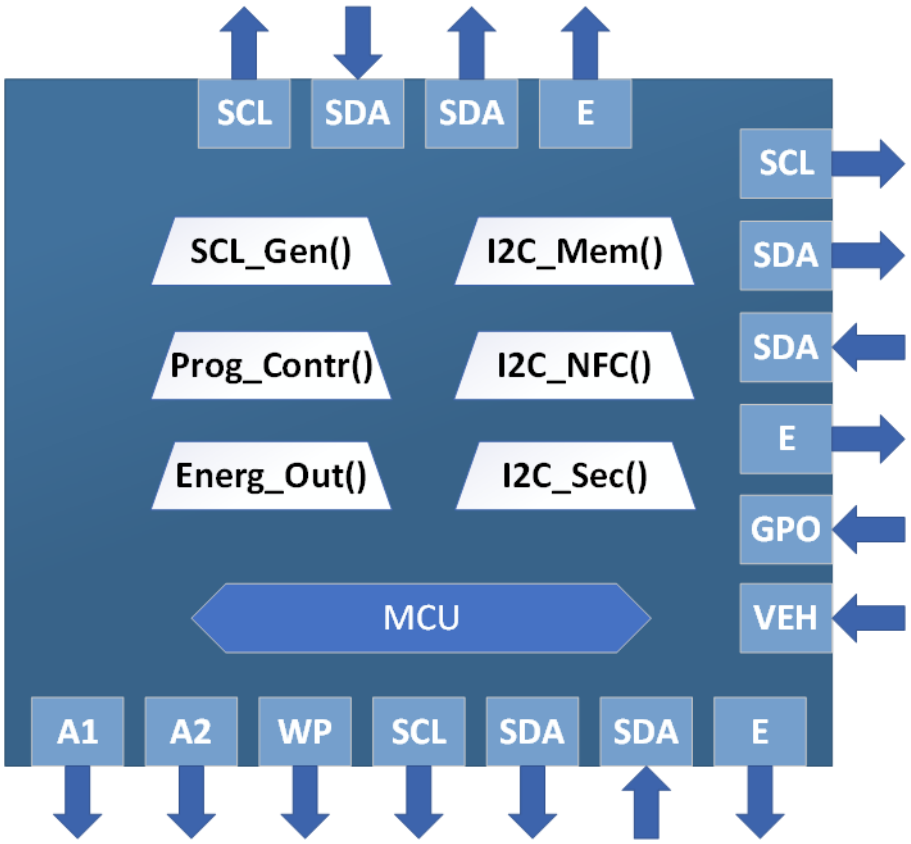

Figure 3.12: MCU design of the SystemC module

The microcontroller is designed to contain only one signal directly associated with this module, and that is the *main clock* signal. This signal has the highest frequency among different clock signals and is used for dividing and controlling other I2C SCL signals. Additional signals showcased in the Figure 3.12 are also present and separated based to which other module they are connected to.
There are three groups of signals:

- Security chip group - positioned on the upper part in the Figure, it composes primarily only the I2C used for that particular communication
- FRAM group - positioned on the lower part in the Figure, it consists out of identification and control signals, as well the I2C communication signals
- NFC group - on the right side in the Figure, it contains the GPO interrupt signals, VEH energy harvesting analogue output and other I2C communication signals

Since a MCU hardware device is very complex, the designed model is made to follow the principles of a finite-state machine. Essentially, the communication processes run with the I2C interface are kept on the system level, while the main program will be implemented as an abstract behavioural system. The program run should offer the possibility to read the data and interact with the NFC module, interact through the security module and to write the end results to the FRAM. Data which is being processed with the MCU has to be saved and read from the inner RAM (Random Access Memory).

3.2.6 Clock Signal Control

Smart Sensor is made of individual modules, each operating at a different frequency. To control the various frequencies, a set of modules was designed with main intention of functioning as the "frequency dividers". They are named *Quartz* modules and they operate on a simple formula as seen and defined by Miller [28]:

$$f_out = \frac{f_in}{n} \tag{3.1}$$

The input frequency is usually the main input simulation frequency. It is the highest frequency based on which the other smaller frequencies are divided from and defined. In the case of equation, the value “n” is an integer finite value. The Modules in SystemC are designed in a similar way, with the main thread/process being used to count the passed cycles for each calling of the main input frequency and when a certain count is reached, the output frequency (the divided one) changes its period. A practical showcase of the signals used for the system quartz can be seen in Figure 3.13.

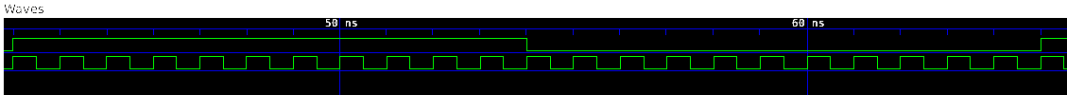

Figure 3.13: Signal wave of the input and output signals of the system quartz module

The main input clock signal for the simulation can vary, but it is by default set to be 1GHz. That means that any other clock frequency is essentially smaller and will be controlled through this main frequency. The designed quartz systems to be used in the model are:

- *System Quartz* - for defining the main clock signal of the MCU. For 48MHz, the input signal will have to tick about 21 times before the period of the microcontroller signal can change.
- *NFC RF Quartz* - used for the RF communication, more accurately defined as a data rate.
- *I2C Quartz* - defines the frequency of the main operating SCL signal used in the I2C communication

Each module is connected as a macro-module being part of the top-module. They consist of the input frequency signal, which is the input signal to the specified top-model. Additionally, they consist of an enable signal which controls the operation and an output signal, which in this case is the reduced clock signal. There exists an additional process inside the microcontroller used to construct the SCL for the I2C with individual modules (FRAM, NFC and security module).

3.2.7 Measurement Approach for the Energy Consumption

Energy measurement is derived from the sum of the individual energy consumptions at module level. It is calculated through first calculating the *power consumption* and then multiplying it with the passed simulation time. Since power is not consistent and it changes with the voltage and current value, all power is first sampled on each individual clock step, later averaged and then multiplied with the total time.

Each module operates with different power states. They are mainly idle and active states with few in-between. The current values of each individual state and their trigger conditions are taken directly from the documentation of individual modules.

The power states and current values are to be implemented as following[12]:

- FRAM power states
  - active standard freq. = 0.04mA
  - active fast freq. = 0.24mA
  - active high op. freq. = 0.71mA
  - idle = 0.015mA
  - sleep = 0.004mA
- NFC power states
  - active e2 read I2C = 0.22mA
  - active mb read I2C = 0.22mA
  - active e2 write I2C = 0.11mA
  - active mb write I2C = 0.28mA
  - idle RF = 0.02mA
  - idle I2C = 0.08mA
  - active RF = 2.5mA
- Security chip power states
  - active state = 20mA
  - idle state = 20mA
  - sleep state = 0.07mA

[12] All elec. current values are displayed based on the 3.3V supply voltage

- Microcontroller power states
  - active state = 0.0128mA/MHz
  - sleep state = 0.101mA
  - deep sleep state = 0.0064mA

Additionally, for certain modules *dynamic power change* was designed to be implemented. This functionality offers the change of the current consumption based on the change of the supply voltage.
For the microcontroller, the following formulas were derived, where "X" represents the current voltage and "curr" the calculated current:

$$active = (3.3/X) * curr - [1.31 * (3.3 - X)] \tag{3.2}$$

$$sleep = (3.3/X) * 0.101 - [0.0388 * (3.3 - X)] \tag{3.3}$$

$$deepSleep = (3.3/X) * 0.0064 - [0.00402 * (3.3 - X)] \tag{3.4}$$

It is to be expected that a more precise model results in giving more accurate energy consumption readings. This assumption is based on the derivation of more exact time readings. Essentially, it is safe to assume that proportionally, a higher error rate on number of cycles needed for a specific process will result in a higher error on the final energy calculation. Because of that, special care is taken when implementing individual models, aiming at lowering abstraction and increasing the functional and time accuracy.

4 Implementation

The design of every system is followed by its test implementation. For a better understanding how a system works in a near/full real-world environment, it is necessary to provide a practical model. A program like that can be used to simulate the system, improve upon the input data, parameters, and to give a good overview on what will the final cost and performance look like.

4.1 Development Environment

The SimPar project was devised to be implemented in two parts, but three applications:

1. Client - server system
    - Server C++ application
    - Client C++ (with SystemC) application
2. Smart sensor simulation
    - Smart sensor SystemC model

The applications were developed using C and C++ programming languages. The Smart sensor was implemented with using the SystemC utility library, but the core programming is still that of the C++. All three applications were developed using a Debian Virtual Machine, i.e. it was done on a Linux OS. Partial development was also done on Windows 10, but that was mainly to test if the corresponding functions and implementations also work under the Windows OS in the way that they were devised. *Eclipse* was the main IDE (Integrated Development Environment) used during the development.

One of the criteria set for the implementation of SimPar was the backwards capability and simplicity. By following this principle, the result code would be easily maintainable and portable. For that reason, only the official technical standard of *C++03* was used for the programming definition. Inclusion of external libraries was kept at a minimum. All libraries which have been used are standard libraries, except for the "SystemC" and "Pthread" library. Other libraries which are heavily used in the development of both the server and client applications, as well as the smart sensor, are:

- iostream
- stdio & stdint
- string (and string-related libraries, like sstream)
- ctime
- iterator
- cstddef
- vector & queue

In addition to the listed libraries, there are also others used to provide the socket toolbox, both for the Linux and the Windows systems, but these are listed and explained in a separate section. It should also be noted that the inclusion of the SystemC library automatically includes additional libraries and namespaces as well. E.g., the "std" namespace is included by default. That can potentially be dangerous, since the inclusion of namespaces with the same name can make conflicts, as well as inclusion of libraries which were already previously included. For those reasons, a special care was taken during the development of the applications to avoid any further errors.

4.1.1 Application Structure

All three individual applications share a similar file structure. It was important to make the file and folder structure readable and understandable. That is achieved by using a hierarchical representation of folders, clear naming conventions and reducing unnecessary files, but also separating bundled up functions. The following are listings of the organised structures of most files (mainly C++ classes) and folders of the finished programs.

Server application "src" folder structure:

- Socket_Classes
  - strategy
    - FORKAction.h & FORKAction.cpp
    - MessageHandler.h & MessageHandler.cpp
    - SOCKETActivity.h & SOCKETActivity.cpp
    - THREADAction.h & THREADAction.cpp
  - NETAddr.h & NETAddr.cpp
  - SOCKETHandler.h & SOCKETHandler.cpp
  - SOCKETConnection.h & SOCKETConnection.cpp
  - SOCKETCommunication.h & SOCKETCommunication.cpp
  - SocketAPI.h & SocketAPI.cpp
- tools
  - HelperFunctions.h & HelperFunctions.cpp
  - SimplePacket.h & SimplePacket.cpp
  - Stylist.h & Stylist.cpp
  - HashMap.h & HashMap.cpp
  - HashMapHandler.h & HashMapHandler.cpp
- Server_Service.cpp
- Guard.h
- MutexThread.h & MutexThread.cpp
- ThreadHandler.h & ThreadHandler.cpp
- Documentation.h
- Includes.h

The main execution file is "Server_Service.cpp". It contains the *main()* function trough which other objects are instantiated. There is a certain scope transparency. That means that certain classes are only visible through other selected classes, to make the program more readable. The "tools" folder contains general singleton classes for services used throughout the program.

Client application "src" folder structure:

- Socket_Classes
  - NETAddr.h & NETAddr.cpp
  - SOCKETHandler.h & SOCKETHandler.cpp
  - SOCKETLine.h & SOCKETLine.cpp
  - SocketAPI.h & SocketAPI.cpp
- simulation
  - counter
    - *SystemC simulation model files*
  - sim_handler
    - CounterHandler.h & CounterHandler.cpp
    - SmartSensorHandler.h & SmartSensorHandler.cpp
    - SimulationHandler.h & SimulationHandler.cpp
  - smart_sensor
    - *SystemC simulation model files*
  - CounterTop.h & CounterTop.cpp
  - Simulation.h & Simulation.cpp
  - SmartSensorTop.h & SmartSensorTop.cpp
- tools
  - HelperFunctions.h & HelperFunctions.cpp
  - SimplePacket.h & SimplePacket.cpp
  - Stylist.h & Stylist.cpp
- Client_Service.cpp
- Documentation.h
- Includes.h

The client application is build in a similar manner like the server. Socket classes are similar, but the connection and communication classes of the server were replicated with a single "Socket Line" class. Additionally, a folder was added in which the SystemC models are kept, along with the simulation handlers.

Smart Sensor application "src" folder structure:

- general
  - *Contains mainly C++ singleton service and configuration classes*
- modules
  - battery
  - fram
  - microcontroller
  - nfc
  - quartz
  - rf_scanner
  - security
  - *Contains also Include file and Top modules*
- Project_Smart_Sensor.cpp

The list of the Smart Sensor application has mostly been dismissed for the reason that each folder contains modules, test benches, sub-modules, C++ functional classes, etc. Listing of each individual class would take too much space.

Most classes of all three applications include the specially constructed "Include" header file. This header file offers the following services:

- Include global and often used libraries
- Offer access to constant and static variables
- Service functions which do not belong to any class

The client and the server applications also contain a "Documentation.h" file which contains general text used by the *Doxygen*[1] documentation tool. The programming files (C++ classes) of these applications are also commented out accordingly based on the Doxygen documentation. The SystemC smart sensor model is documented as well, but without the use of the Doxygen platform.

[1]Doxygen is a tool and standard used for documenting and generating code documentation, with an HTML (HyperText Markup Language) option

4.2 Integration and Implementation of the Client and Server Applications

The system is constructed with two programs. One handles the client and the other the server side. Both are independent, but they do share some similarities in the implementation. A graphical representation of the main functionalities can be seen in Figure 4.1. This figure showcases the different functionalities with the use of a Venn diagram, where it can easily be seen which parts of the applications potentially share similar or even identical code. The parts on which the server and client application are analogous are primarily around the socket handling.

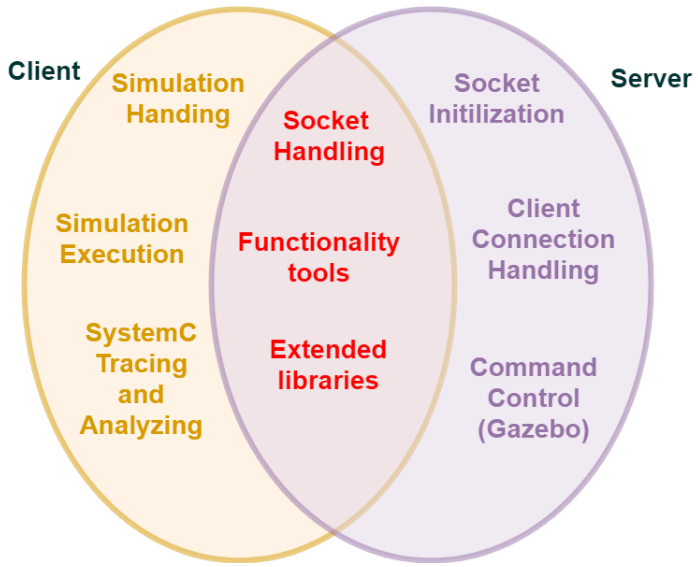

Figure 4.1: Venn diagram showcasing server and client functionalities

Since the Socket functions belong to both the client and the Server side, one shareable Socket API is implemented which is then copied and used on both the client and the server application. Apart from that detail, the overall folder structure and programming style is kept for both applications. The style follows the agenda of design patterns by focusing program handling on objects, leaving the code and operations in the *main* function to be simple and understandable.

4.2.1 Implementation of a Simple Socket Communication

The socket communication was implemented based on the standard UNIX socket programming principles. The style is based on the famous Beej's guide to network programming defined in Hall [19]. Since most of the guides imply that the socket programming is to be done in C, a certain level of flexibility is lost when developing larger applications. For that reason, C functions are programmed in their own independent API and C++ is used to develop classes to handle individual aspects and steps of the socket programming.

4.2.2 Server Communication Handling

The server first establishes its own configuration (host, port, type handling...) and then continues with the following steps:

1. Server is waiting and afterwards, accepting a connection
2. Server is waiting for the response of the client to set the connection
3. It then sends a request for an unique "code"
4. It receives the code from the client and checks. If it is wrong, end the connection. If it is good, proceed with following actions
5. Client is then connected to a specific command vector and thread (in a larger environment, also to channels)
6. Server proceeds with sending the messages with commands, each time waiting for a correct status message from the Client telling the Server that it is ready again to receive a new message
7. The previous step is repeated (step 6) until a message to start the simulation is sent
8. Server is waiting for the simulation to end on the client side and confirms by receiving a message of confirmation
9. Server can ask for results or proceed with new commands or end the simulation altogether. If it repeats, it jumps back to the step 6
10. In case the simulation cycle is completely done by the server, the server sends a message to end the connection to the client and then it ends its own. The client also has the ability to end the connection at any point

The server handles each client individually, concurrently. The way in which that is done is set through the command line when calling the program as the third parameter. Both *forking* and *threading* is implemented, but the work focuses on threads, as threading is the only approach that works on *Windows*.
Figure 4.2 shows the sequence diagram of the communication between the server and a client for a particular simulation session. To note here is that no actions are showed which include premature disconnection from the client side, as well as other errors or alternative actions. Some smaller steps and messages transmissions have also been omitted to save on the visual space representation.

4.2.3 Control of the Client SystemC Simulation

The idea on the client side was to develop the same interface for all simulations. That includes setting up the connection, reading commands and passing them to the simulation. The framework around which client is build is set around the server framework with specific simplifications and changes. For the development purposes, generally two SystemC models have been integrated:

1. *Counter simulation* - A one simple SystemC module described as a counter; Done primarily to test the work-flow of the communication and to gain the understanding of the primary (universal) message parameters
2. *Proto Smart Sensor simulation* - Run a previously implemented Smart Sensor SystemC simulation with already predefined commands for testing

The so-called Proto Smart Sensor model is essentially a model build previously from other team members of the IoSense project and only adapted to be run on this client site. Because of that, the actual implementation of both the Counter and the Proto Smart Sensor module will not be presented in this work, rather the description of the implementation will remain more focused on the SimPar version of the Smart Sensor. The approach which is introduced can also be applied to any subsequent SystemC simulations.

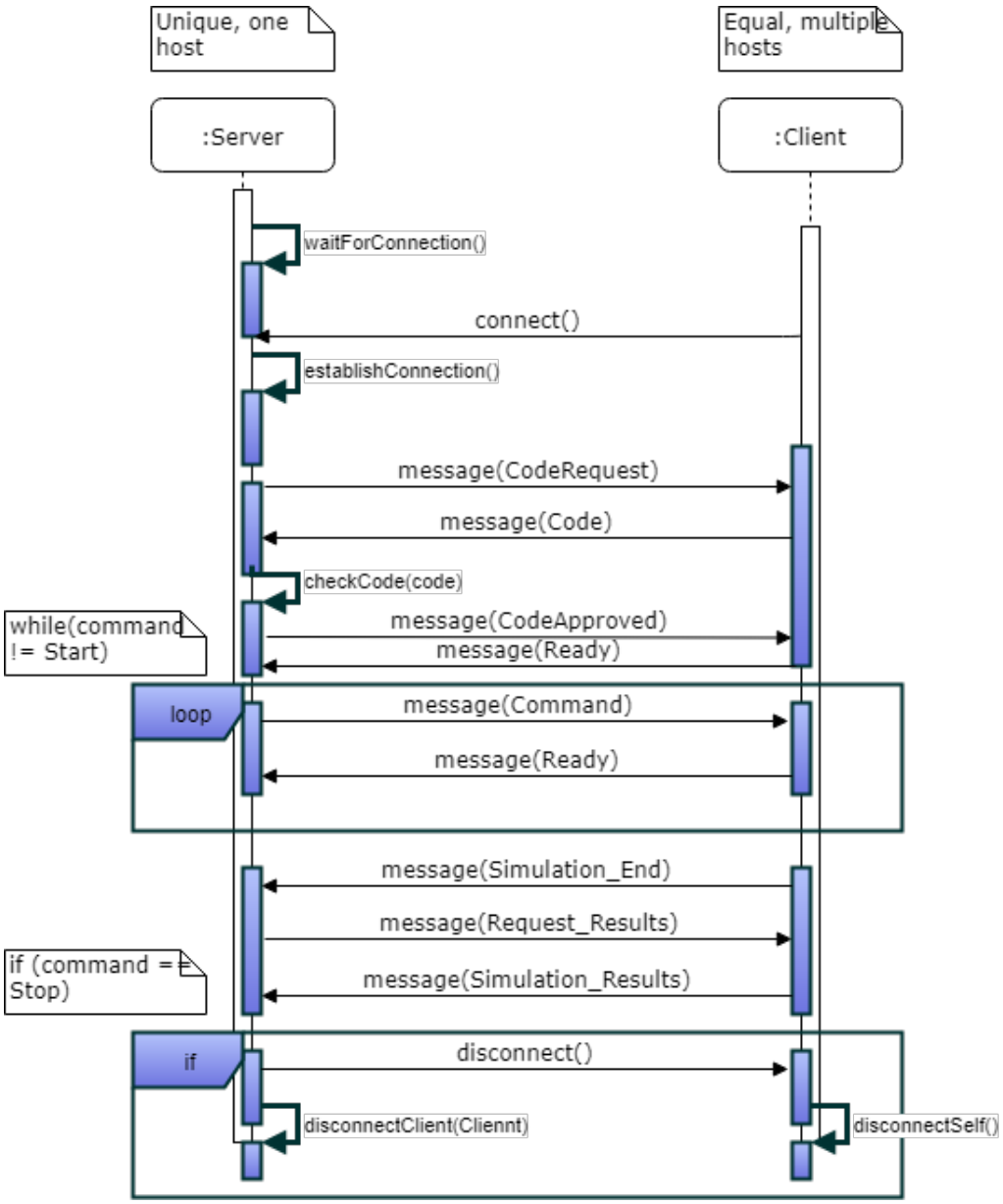

Figure 4.2: Sequence diagram of the communication between Server and Client

4.2.4 Communication Packet Composition

Currently, for simplicity purposes, a system composed of a simple packet structure is proposed and implemented. To streamline it additionally, it is not necessary to create structures and then send them via a binary representation, but rather it can also be done though a character array (string).
The packets are represented in an XML (EXtensible Markup Language) format with two main elements, header (type) and body (data), set with:

- <t></t>: type of the message; can be status, code, command, etc.
- <d></d>: data inside the message; depends on the type

Primary commands used in the communication are described in Table 4.1. The commands are universal among different simulation models and each SystemC module implemented on the client side should also adjust its simulation run-time based on the pre-defined protocol. That should not be difficult, since the main difference between the simulations should only be in the commands that are sent during the initialization step of the simulation. The handshake step, as well as the step concerned with the simulation handling is to remain the same for all the models as to reduce the complexity.
It should be noted that the packet structure follows the XML text structure. The commands sent inside the data part (<d></d>) also adhere to this style.

To synchronize the communication of a session between Server and Client an instruction counter is proposed. The reason is that the server will always wait for the message from the client to respond with an appropriate action (request – response method). In case the client also waits for the next step (is currently idle) it will simply send a message where it says that it is ready and then wait for a response from the server. Initially, the instruction counter was thought to be set through the client, but for the communication purposes, it was decided to be set on the server. The instruction counter follows the cases set by the design in Figure 3.1, where the overall communication goes by principle seen from the sequence diagram shown in Figure 4.2.

Table 4.1: Main commands used in the packet structure of the communication protocol

Command	Description	Case
<t>status</t> <d>ready</d>	General client status signal stating that it is ready to accept the next message in the protocol	0-2
<t>status</t> <d>code_request</d>	Response from the server asking for the code validation	0
<t>code</t> <d>*some_code*</d>	Code sent from the client side	1
<t>status</t> <d>code_passed</d>	Server response if the code has succeeded	1
<t>status</t> <d>code_failed</d>	Server response if the code has failed	1
<t>command</t> <d>*command*</d>	Command sent from server to the client	2
<t>status</t> <d>waiting</d>	After the commands have been sent, server waits for a response from the client	2
<t>status</t> <d>sim_started</d>	Client responds to server that the simulation has started	2
<t>status</t> <d>sim_status</d>	Server asks client for the status of the simulation	2
<t>status</t> <d>sim_finished</d>	Client responds that the simulation has finished	2
<t>status</t> <d>sim_received</d>	Server acknowledges the end of the current simulation run	2
<t>command</t> <d>0</d>	Server responds with closing the complete simulation session	3

Configuring the packet handling inside the application

Handling the packet content on the client side is done through the special "Handler" classes. For each added SystemC simulation, a separate class needs to be constructed. These classes are located inside the "sim_handler" folder. In this folder, a special class is located called the *SimulationHandler*. It essentially servers as a template for providing access to the already implemented functions, which are associated with the simulation handler, and to set the need to implement specific *virtual* functions[2].

Listing 4.1: Simulation Handler Super Class Header

```
namespace simc {
        class SimulationHandler
        {
        public:
        SimulationHandler();
        virtual ~SimulationHandler();

        const char* messageSetter(const int instCounter
        , const uint16_t connCode = 0);
        int messageInterpreter(char* message);
        void setSimulation(Simulation *sim);

        virtual int applySimulationCommands(std::string data) = 0;
        virtual void startSimulation() = 0;

        Simulation *sim;
        };
}
```

Code listing 4.1 displays how the header file of the main simulation handler super-class is implemented. Two important virtual functions: "applySimulationCommands" and "startSimulation" should be implemented by the user. These functions specify how the simulation commands are interpreted

[2]virtual functions must be implemented, serving as a good program control

and how the simulation is started respectively. User is also able to add any additional custom functions exclusive to his simulation, which was also the case in the SimPar test cases with the two simulation instances of the *Counter* and the *Smart Sensor* simulations.

SystemC simulations are highly dependent on the server control, in which case, the client cannot start or stop a simulation before the approval of the server. This relationship can be described as a master/slave communication.

4.2.5 Cross-platform Support

Client and server applications were developed in mind to handle both Linux and Windows operating systems. Some additional changes on the Linux system have been made to support other UNIX based systems (e.g. Solaris). The end result is that the server side is partially able to be run also on Windows, but with certain constrains that have not been cleared until the end of the project (could be left for the future work). The client application is able to be run on both operating systems, but the actual implementation depends on the additional code, which is the SystemC simulation. If the handling of the simulation uses some Linux-specific libraries, it can create problems while trying to run the application on Windows. Runnable application is handled independently from the system and is only defined accordingly via a compiler.

During the implementation, special care was taken when defining how the socket communication should be integrated. While in terms of the naming conventions and even the functions in general, there is very little difference between Windows and Linux, except for exclusive libraries which have been used. This was handled inside the "Include.h" file where all external libraries are being included. The proof action is done using the **#if defined(...) ... #else ...** statement. This approach was also used to control the code before the compilation process for other critical statements. An additional example can be found with the *mutex* handler, which is differently done on all three os of interest[3].

[3]Linux, Windows and Solaris

4.3 Implementation of the SystemC Smart Sensor

The implementation of the main simulation model was performed through different phases. For each phase, a separate module was implemented and viewed as an individual component. Afterwards, a test bench is set to test the functionality and accuracy of the developed module, such that the module can safely be inserted in conjunction with other simulation parts.
Since the smart sensor model is very large and consists of several elements with thousands of lines of code defining them, only the most significant algorithms and techniques applied will be presented in this section.

4.3.1 Memory Representation

The memory module is implemented as the FRAM chip, previously specified in the design step by Figure 3.9. Two main functions are constantly being run in a process loop, the communication and working functions named I2C and the function/process dedicated to calculating the output current (energy consumption).

The memory module was used as a building block for setting the I2C communication protocol used also by the other modules. The principle on which the I2C communication is defined is the same as the one used for implementing the communication framework inside the NFC and the security module, with small exceptions.
Figure 4.3 presents the diagram of steps taken during the standard session of an I2C interaction. Each communication step begins by first analysing if the received device address is correct. If it is, then the second step can proceed in which the address is first being read and then the appropriate operation is conducted. The operation can either be *read* or *write*, depending what was defined in the first 8 bits sent as the device address [4]. The I2C communication protocol has special *start*, *stop* and *ack/noack* sequences. Since they require special properties of the analogue signals, they have been actually implemented with the use of an additional *enable* signal.

[4] the LSB (Least Significant Bit) in the device address byte determines this operation

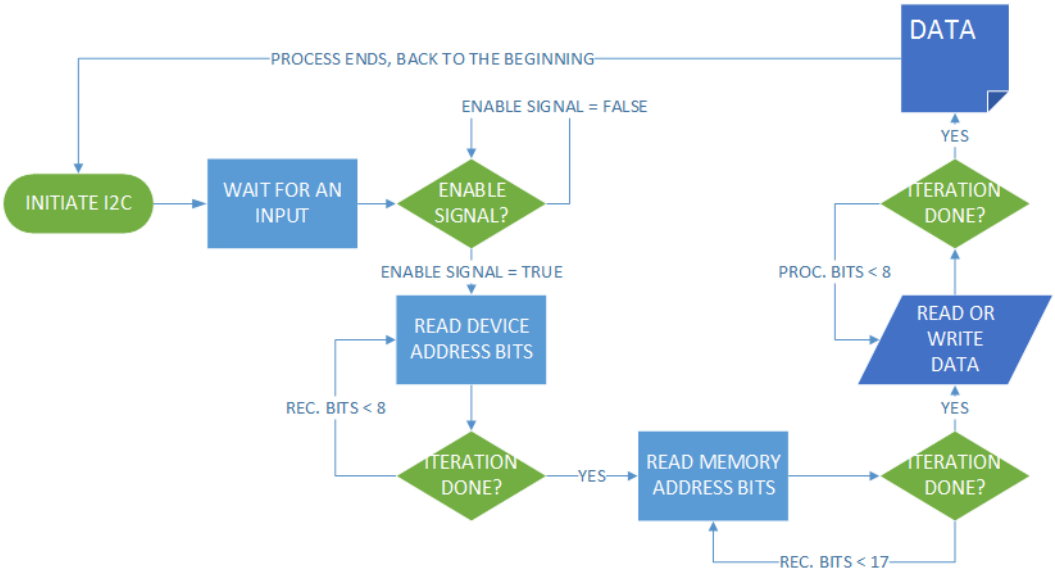

Figure 4.3: I2C process diagram used as a main template for this type of communication

The memory module works in three different states. While it waits for an enable signal it remains in the *idle* state. Upon starting the I2C session it will go into the active state. A special sequence of command has to be sent (partially through the device address) from master to memory (slave) for it to be able to go to the *sleep* state.The memory can only go from sleep back to idle through an another interrupt by the master system.

4.3.2 NFC Communication Handling

The NFC tag has seen an emulated implementation (similar to the memory module) through very complex communication and functionality handling. The NFC tag on its own has a memory sub-class which contains separately the user memory, dynamic and system memory. They are all implemented as static arrays, but the interaction with writing and reading from memory positions is independent. All functional handling is being done through a careful interaction with the registers, which are usually either associated to the dynamic or the static memory. These functionalities usually range from

sending and saving a value to the NFC user memory, using the *mailbox* to transfer data between RF and the I2C or to activate the energy harvesting capability. Since this communication algorithms plays such a vital role, its implementation is inspected separately.

I2C Communication

The I2C communication protocol by the NFC module follows the same principles set by Section 3.2.3, and is also found with the memory module. There are some minor differences unique to this module:

- The sleep state is triggered internally, rather by an instruction from the master
- The device address is different, with a special bit being used to separate memory interaction between dynamic, and user from the static memory
- The I2C protocol is two stepped, meaning that it is first necessary to send a write command during the reading process, with the actual reading being followed during the second *start* signal

RF Communication

While the I2C communication generally works on frequencies around 1MHz, the transfer rate for the RF is much smaller, being opted at 26KBits. Aside from that, the RF interface communicates with frames rather than with simple bytes. Each frame consists of several bytes, each indicating a different purpose. A showcase of a used frame can be seen in Figure 4.4.

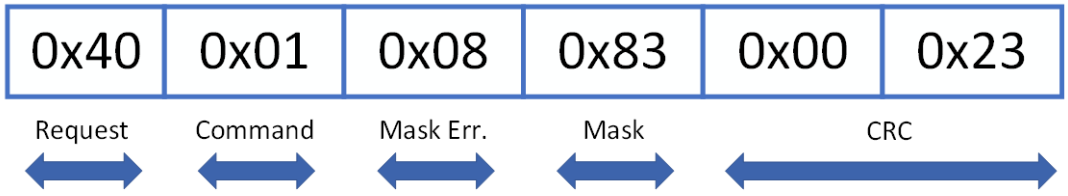

Figure 4.4: RF request frame format

The RF communication is made of both similar, yet different, request and response frames. For each frame a command is sent, next to additional information and an obligatory CRC (Cycle Redundancy Check) 16-bit value. Since every command requires a different and complex handling, a set of inheritance classes was designed. Each class represents a different command, with the main request class "RFFrameRequest" being the super class with general methods. All request classes are located inside the "request_data" folder and they include all the commands that the referenced NFC tag also includes. The principle of implementation is based on the strategy pattern.

Mailbox Fast Transfer

One of the main functionalities and purposes of the NFC tag in this case is the use of the "mailbox" function. This function allows a very fast transfer between an RF external device and some other end device through the NFC tag. One of the steps (step 5) is shown in code listing 4.2.
This process was implemented in the following way:

1. Set the MB_MODE (static register 0x0D) to 1
2. Get the b3 and b2 values of the EH_CTRL_DYN (Dynamic 0x02) and check if the condition matches (VCC and RF are ON)
3. Read MB_CTRL_DYN (Dynamic 0x06) and check that the value should be 0x00 (mailbox is reset)
4. Read first message length and then the message (both should be 0)
5. Write in MB_CTRL_DYN to enable the mailbox
6. Send the command to write to the mailbox and update the corresponding registers (status and length registers)

Listing 4.2: Simulation Handler Super Class Header

```
NFCFrame::instance()->writeDynamicCommand(rfValues,
                    0x40, 0x06, 0x01);
rfWrite = true;
waitForTheResponse = true;
waitForAction();
```

4.3.3 Security Module Operation Execution

The security module design is very abstract as it is not well defined how the inner workings of the logic is made concerning the handling of the various security functions. Because of that reason, the implementation of the security module follows an improvised approach based on the experience from modelling other modules and simulations, as well as taking notes from the implementation of some other security chips.
The security module uses a specific execution protocol, that is exclusively developed and then implemented for this module. A single session follows these steps:

1. Indicate that the process to write in the working memory is activated
2. Start sending data from master to the security module (standard I2C protocol)
3. Check if there were any errors during the process. Resend the data up to 3 times in case an error is detected
4. Send indication to stop writing inside the working memory
5. Inform that the module should start a security process
6. The microcontroller (the master) checks periodically (every 20ms) if the process is finished
7. After the process is finished, the microcontroller gets notified and then an error check is forwarded
8. The microcontroller indicates to work with the working memory
9. The security module sends the newly created data to the microcontroller (after applying a security function)

The implementation of the security module differs from the one of the FRAM chip and the NFC tag in terms of the abstraction implementation layers. Where the two mentioned modules were designed to fulfil both the functional and the execution-time constraints, the security module makes assumptions when it comes to the accurate timing. The functions implemented are also focused only in giving the random value output, rather than doing the actual security process. This is done for the reason that the purpose of each function is not important in the overall energy consumption analysis of the smart sensor, since the security module must keep a constant consumption regardless which security function is used.

4.3.4 Tracking the Energy Consumption and Harvesting

The consumed power of the whole smart sensor is calculated by first acquiring the total current consumption and then multiplying it with the present battery output voltage level. Each module outputs one electrical current value on each clock trigger, where the signal is usually named and defined in the following convention: *sc_out<double>currentOutI2C*. Only the NFC module consists of two different current outputs, from the I2C and from the RF element. The setting of the appropriate power states and current value is done from a separate thread/process for all modules. The used naming convention for individual modules is the following: *void outputCurrentI2C()*.

Table 4.2: Energy harvesting current output in mA based on the magnetic field strength

Modulation	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**	**5.5**
10%	0.1	0.3	0.7	0.9	1.1	1.3	1.5	2.1	3.3	4.5	4.9
100%	0.7	0.7	0.7	0.7	0.9	0.9	1.3	1.7	1.9	2.3	2.7

Energy Harvesting has been implemented as a feature to be used through the NFC module. It gives the possibility to power other devices, but also to charge the battery. The amount of current and voltage drawn depends on the proximity of the RF device which powers it, as well as the strength of the signal. Table 4.2 displays different input current values from the energy harvesting process. They are set in a ordering position and dependent on the *H value*[5], which in this case ranges from 0.5 to 5.5, and the new value is taken with the step of 0.5. Appropriate functions have been implemented to carry both the energy harvesting control process and the current drawn in a separate process.
To active the energy harvesting process it is necessary to set the special register value of the EH_MODE to be 0x00 of the 0x02 dynamic address. The NFC module contains a special running process which periodically checks if that value has been changed.

[5]Magnetic field strength

Handling Dynamic Energy Consumption

The energy Consumption is handled dynamically with two approaches:

1. Calculating the power output with the dynamic voltage
2. Using the power states to change the current consumption

Battery dynamic voltage is a direct result of the battery model. This model is build on a simple principle of calculating the current charge of the battery and determining the current voltage output, the same one used to power the rest of the devices.
The Battery charge formulas are displayed with 4.1 and 4.2 , and discharge with 4.3 , respectively:

$$Voltage = Charge\_Coeff * Max\_Battery\_Voltage \tag{4.1}$$

$$Max\_Current = \frac{(Charge\_Coeff * Max\_Battery\_Voltage) - (\frac{Charge}{Battery\_Capacity})}{Battery\_Resistance} \tag{4.2}$$

$$Voltage = (\frac{Charge}{Battery\_Capacity}) - Current\_Load * Battery\_Resistance \tag{4.3}$$

The values “Battery Capacity” and “Battery Resistance” are given at the compile time and they are constant. Essentially, the are used to describe the main parameters of the specific battery. The voltage is dynamic, as well as the current load which is read. Equation 4.2 displays the maximum current that the battery can take in case the load is higher during the charging process. The value of the battery voltage is used by each cycle step to calculate the power. That is done by multiplying it with the captured current at a specific time input.

Energy states are usually triggered and therefore changed by a certain set of conditions. One of these conditions is usually setting the energy state to be proportional to the working inner state of the module, or in case of idle and sleep states, changing to this state in case of a longer inactivity.

5 Evaluation

To better understand and improve a system, it is necessary to perform tests intended for performance, functionality, and fault-tolerance evaluations. From the implementation's perspective, the SimPar system can be evaluated on the following points:

- Server and client SystemC simulation model handling
  - server response by interacting with multiple clients in terms of handling faults (sudden errors), functionality precision and performance (execution time)
  - client response to specific communication messages, handling of the commands relative to the simulation and performance impact
  - network stability and conduction from both the server and client applications
- Smart Sensor SystemC simulation
  - performance of the simulation, run-time and resource handling
  - actual simulation parameters influencing the time and functional accuracy of individual modules
  - fault tolerance (signal errors, delays, wrong commands, etc.)
  - energy consumption measurement and analysis of the energy efficiency (including the energy harvesting capabilities)

Since the client and the server applications have been developed to function as a framework rather than an end solution for the desired simulation use, the main evaluation points will be focused on the smart sensor application.

5.1 Server and Client Application Evaluation

The system can be executed through two compiled programs, where each program can be run on a separate machine. Considering how the system was implemented, the server can be run on a Linux[1] and a Windows machine. The client depends on the SystemC models with which it interacts. That is for the reason that specific models used in the testing phase use exclusive Linux commands and therefore, could not be run on a Windows machine. A simple UI (User Interface) was implemented to handle initial configuration [2] on both the server and the client side. Additional help windows are also provided to guide the user in executing the program. Most of the actions are done automatically and independently which was one of the main goals of the developed model.

A stylized terminal interface is provided if the program is run through the command terminal, both for Windows and Linux, with legacy support[3]. Figure 5.1 shows the program execution run on the same machine with the server first running the initial setup and then accepting and maintaining the connection with two clients. Even if the client applications are being run in parallel (in two different processes), the server manages to display the messages and to handle them accordingly.

The results of several test-runs show no issues with handling a large number of clients, due to the optimization principles which were applied. Fault tolerance was also tested, in which case, even if the communication is abruptly broken between one client and the server, the process which runs on other clients, continues normally. Sending and handling commands showed no issues as well. The client tracing works as expected. Small issues are detected with the protocol handling in case of failures and random messages, with the solutions being currently designed and planned for the future work.
The system was not tested on a real network, but rather on a small interconnected LAN (Local Area Network). The application is not intended to be used as it is on a larger network, because the server IP address must be known in advance.

[1] Potentially also some other UNIX-based OS, such as Solaris
[2] setting up the IP address, port number, handling type, simulation type, etc.
[3] support for the previous versions of the CMD, i.e. it changed with Windows 10

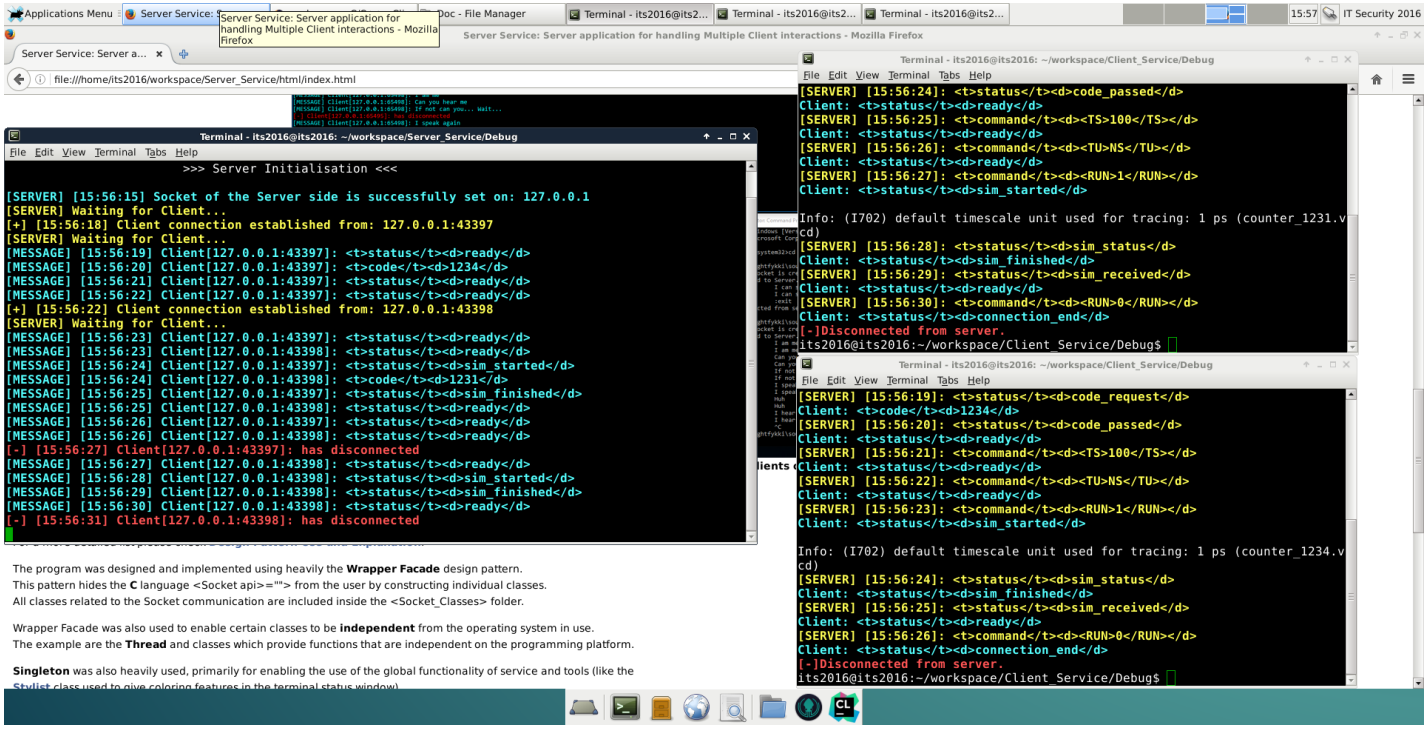

Figure 5.1: Execution of the server application and two client instances on one machine

5.2 Smart Sensor Model Evaluation

The main focus of the evaluation is set on the smart sensor. The model of the smart sensor is easier to analyse and adjust. This is unlike the server and client applications, where the framework is not adjustable to affect the changes to the performance of the system. Rather, the performance of the system is mainly dependant on the SystemC simulation which is being run. Several critical points have been taken into the consideration while developing and implementing the model of the smart sensor to enable an easier evaluation at a later period in time. It was also necessary to provide an effective framework where the parameters of the SystemC model can be changed during the execution time, rather than the compile time.

Alongside the developed application a *configuration file* was created named "config.cfg" and placed inside the root of the application structure. This configuration file is essential in running the simulation since it gives instructions and values for each individual model parameters. Each parameter has a trailing comment providing the explanation and the default values. It is absolutely necessary to follow the instructions since a falsely given value can sometimes result in the simulation behaving unintentionally, or it might become completely defective, as not all faults are caught[4].
The power and energy values are saved in a column/row style, with two intended columns. One column is dedicated to the time-stamp for when the value is taken, and the other is for the power taken at that time point. The energy consumption is calculated afterwards from these values using the standard formula. Since the main clock measures the power on every nanosecond, the size of the output evaluation file can result in several gigabytes. Because of that reason, *aggregation* is used by calculating an arithmetic mean for every one thousand values (or for every millisecond).

The chapter is organized in two main sections. The first section deals with evaluating the functional accuracy of each individual model while at the same time also presenting a rough estimation of the consumed energy. The second section is more focused on the full-program execution including the devised microcontroller programs, displaying the power and energy overviews.

[4]to most parameters a default value will be assigned in case of unexpected inputs

5.2.1 Evaluation of Functionality accuracy and Energy Consumption for Individual Modules

For each individual module, a *test bench* was implemented. These test benches are modelled as separate modules, containing the functions and signals used to control and work with the assigned main working modules. In other words, they try to replicate the master side of the communication, in this case that being the microcontroller[5]. Next to the standard functions which are used for the communication and protocol handling, the test benches also contain a special function which lists the corresponding test instances. Each instance contains a list of commands and steps used to test a specific functionality of the corresponding module.

The tests which are being run are controlled through the *configuration file*. Each session of the simulation is only able to run one test. Most of the tests also contain some form of *standard output*, thanks to which the log information can be read and analysed to see, if the module functions properly. The power consumption is also saved in a power trace file in a "[time stamp][ts metric], [power][pow. metric]" format. Additionally, a *signal trace file*(VCD, Value Change Dump) is created. The signals, which are traced, are set inside the aforementioned configuration file.
The next sub-sections give a brief summary of each individually created tests, mean power and energy consumption for the duration of the 100ms, which is used for every test simulation.

FRAM Test Bench

The first developed module was the FRAM. A total of five tests were realized. Two are focused on the writing capabilities, one only on read of multiple values, one for testing the *sleep* functionality and one interconnects all the functions that this memory module provides.
The test cases can be seen in the table 5.1. It should be noted that since the execution of the specific tasks can be relatively fast, the modules mostly remain in the sleep state. Also, the mean power which is displayed alongside

[5]since the microcontroller is the central element which communicates with all others

the calculated energy consumption, is derived from other modules as well and not only from the FRAM chip. Since these modules are not active, they mostly remain in either the idle or the sleep state.

Table 5.1: Memory module test cases

Test Case	Brief Test Explanation	Power	Energy
Write 1 val.	Send and write a single value	0.643mW	0.064mJ
Write n val.	Send and write 3500 inc. values	0.877mW	0.087mJ
Read n val.	Send a request and read 3500 values from the memory	0.877mW	0.087mJ
Set sleep	Set the memory module to the sleep state for 50ms, then write 3500 values	0.859mW	0.085mJ
Complex func.	Set the sleep state for 25ms, write 1500 values, set again to sleep for 25ms, read written values	0.827mW	0.082mJ

From the presented power and energy values it can be seen that the energy dissipated during the process of reading and of writing is the same. That is due to the nature of the FRAM technology, and theoretically, there is no difference between them. In a real world environment however, some other elements could influence the change in the results.

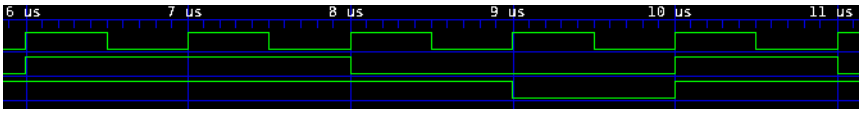

Figure 5.2: FRAM trace VCD during the writing process

The output results are also saved in a VCD file showcasing the signals. One of those results of the process can be seen in Figure 5.2. The first signal represents the I2C SCL signal, the second input SDA signal and the third SDA output signal.

NFC Test Bench

The NFC tag module received the most elaborate and diverse collection of test cases, mainly for the reason of its complex implementation. Since the module consists of two types of separate operation modes, I2C and RF communication, test cases have been divided in three main groups: I2C based test cases (Table 5.2), RF based test cases (Table 5.3) and the inter-combination of two modes (Table 5.4).

Table 5.2: NFC tag module I2C test cases

Test Case	Brief Test Explanation	Power	Energy
Write 1 val.	Send and write a single value	0.643mW	0.064mJ
Write n val.	Send and write 3500 inc. values	0.877mW	0.087mJ
Write & Read 1	Write and then read 1500 values	0.719mW	0.071mJ
Write & Read 2	Complex set of write and read func.	0.643mW	0.064mJ
System mem.	Write and read from the system memory	0.643mW	0.064mJ
Dynamic mem.	Write and read from the dynamic memory	0.643mW	0.064mJ

It can be noticed from the analysis presented in Table 5.2 that the values of the power and energy consumption are identical to the ones presented in the FRAM analysis in Table 5.1. This is the result of the similar power-aware technology used for the construction of the respective I2C protocol. However, there are differences between these two modules and tests over longer periods of time would eventually lead to more significant distinctions. The tests focused on reading and writing to the dynamic memory show equal results to the write command of the user memory values. The reason for this behaviour is that for the system/dynamic memory tests, only the changes in one value have been tracked. The tests were focused more on the functionality accuracy rather than the energy consumption analysis.

Table 5.3: NFC tag module RF test cases

Inventory	Command to check the *inventory* status	0.851mW	0.084mJ
Read a block	Command to read one block (4 bytes)	0.821mW	0.082mJ
Write a block	Command to write one block (4 bytes)	0.936mW	0.093mJ
Read n blocks	Command to read multiple blocks (n*4 bytes)	0.851mW	0.084mJ
Write n blocks	Command to write multiple blocks (n*4 bytes)	1.077mW	0.107mJ
Read config	Command to read a specific configuration	0.827mW	0.082mJ
Write config	Command to write to a specific configuration	0.879mW	0.087mJ
Read dynamic	Command to read a specific dynamic mem. pos.	0.827mW	0.082mJ
Write dynamic	Command to write to a specific dynamic mem. pos.	0.879mW	0.087mJ
Write message	Command to use the fast message system	2.214mW	0.220mJ
Read msg. length	Command to read the current message length	0.827mW	0.082mJ
Read message	Command to read the current message	0.827mW	0.082mJ

Table 5.4: NFC tag module RF and I2C test cases

Mailbox 1	Series of commands for sending the message from a RF to the I2C system	5.235mW	0.522mJ
Mailbox 2	Series of commands for sending the message from a I2C to a RF system	1.646mW	0.164mJ
Energ. harvesting	Turn on the energ. harv. followed by the mailbox 2 test	-19.2mW	-1.91mJ

The implemented NFC module offers the use of the standard NFC RF set functions which can be seen listed in the Table 5.3. These were for the simulation the most critical functions, among others, which have been implemented and thereafter tested. RF commands generally take much longer time for a similar execution than the I2C counterparts for the reason of the much slower bit transfer. This type of the operation modes also draws more energy. There are also some other more significant differences, one of them being that the writing operation takes more time mainly because of the so-called *write cycle* which takes additional time compared to the read operation.
Table 5.4 lists the three most complex tests regarding the NFC tag module, since all three tests are run by using both the I2C and the RF operation modes. In both cases of the mailbox tests, a sequence of 32 values was sent. The energy harvesting test is focused first on declaring a set of sequence of commands for turning on the energy harvesting mode, followed with the "Mailbox 2" test case. The negative values for the power and energy results indicate that during this operation, a part of the energy was used to charge the battery.

Security Chip Test Bench

Unlike the tests previously made with the other modules, the security chip is set differently in that it is implemented on a higher abstraction layer. That can also be seen with the config. values from the security handling seen in Table 3.7, where the current in active state is very high compared to other

processes. The same current consumption is also kept in idle state for the security reasons. Table 5.5 displays nine different tests, each separated on writing and reading from the user memory, system memory, and running the general security tests from the main functions.

Table 5.5: Security chip module test cases

Test Case	**Brief Test Explanation**	**Power**	**Energy**
Write 1 val.	Send and write a single value	0.668mW	0.066mJ
Write n val.	Send and write 4 random values	0.685mW	0.068mJ
Read n val.	Read 5 specific values	0.700mW	0.069mJ
Write & Read	Complex set of write and read func.	0.845mW	0.084mJ
System mem. 1	Write changes to the data in the system memory placed on the position 0x01	0.722mW	0.072mJ
System mem. 2	Tries to access memory address not existing, operation fails	0.722mW	0.072mJ
Security 1	Tests the security function 1 (Mutual Authentication using DTLS)	20.88mW	2.084mJ
Security 2	Tests the security function 2 (One Way Authentication) first, followed by 3 (Crypto Toolbox)	53.58mW	5.356mJ
Security 3	Tries to access a non-existing security function	1.201mW	0.120mJ

The test cases named “Security 1” and “Security 2” show considerable higher energy input than other tests. That is mainly due to how the security algorithm works. It takes a fixed amount of time for one session to end, where each different session can take somewhere between 30 and 70 ms. The final test, “Security 3”, tests the function error-handling.

5.2.2 Microcontroller Program Testing

The last element which was designed and then integrated into the overall system was the microcontroller. This module functions as a combination step between all different modules and is used in controlling the whole process. Because of that reason, most of the processes implemented inside the microcontroller are actually based on the control elements from the test benches. The main difference is that the microcontroller rises the complexity of integration. Additionally, programs were put into place for actually running the individual processes. Two main programs were developed:

1. Basic NFC ->FRAM - Data is received from a RF card, send to the microcontroller and then saved permanently inside the FRAM
2. Advanced NFC ->Security ->FRAM - Similar as before, the main difference being that a security function is called before the data is saved to the FRAM

The testing of the individual programs had three goals; to test how the smart sensor functions as a finished simulation model, does each individual module fulfils the basic modelled functions and how does the power and energy consumption behave under the set parameters.

Microcontroller Program Definition

One process/thread was defined inside the microcontroller with two additional functions. The thread is used to control the values and the execution of the individual microcontroller programs. These programs are defined as the functions and are called on each clock step. Each of the two programs is implemented through the use of the FSM principle. The microcontroller will be denoted as UC for the rest of this section.

Basic program

1. RF scanner card starts a protocol of sending arbitrary data to the NFC tag mailbox
2. UC receives the signal that new data is present in the NFC tag
3. UC checks the size of the message inside the mailbox
4. Using the I2C communication, the UC reads the data from the NFC

5. The data is saved inside the UC RAM
6. UC signals the FRAM and starts sending the data to be saved
7. Program ends and reset is initialized

Advanced program

1. Steps 1 - 5 are repeated from the basic program
2. The UC starts the security protocol
   - □ Prepare the data from RAM to be sent
   - □ Set the appropriate security function
   - □ Wait and respond on corresponding steps
3. After the security function is done, the chip sends back the new data to the UC
4. Steps 5 - 7 are repeated from the basic program

Energy Consumption

The following section showcases the analysis of the energy consumption of the two aforementioned programs. Both programs have been individually evaluated. From the parameter list it was set that for the both programs a total number of 32 values is to be worked with (from the RF scanner tag to all other processes). Additionally, the second advanced program used the first function (Mutual Authentication using DTLS) as the security basis. Both simulations run for a total of 100ms and only one session of data transmission is initialized.
The power and energy consumption of the basic program are shown in Figures 5.3 and 5.4, and from the advanced program in Figure 5.5 and 5.6 respectively. By the basic program, most power is spent during the RF first step, up until 37ms, afterwards until 42ms very little power is used. From 42ms onwards the whole device is in sleep state. The advanced program shows similar results except for action in-between 38ms and 65ms. There is a large increase in power due to the high current consumption of the security chip active state. The total energy consumption of the basic program has been measured at 0.34mJ, while advanced program showed a much larger consumption of 2.43mJ.

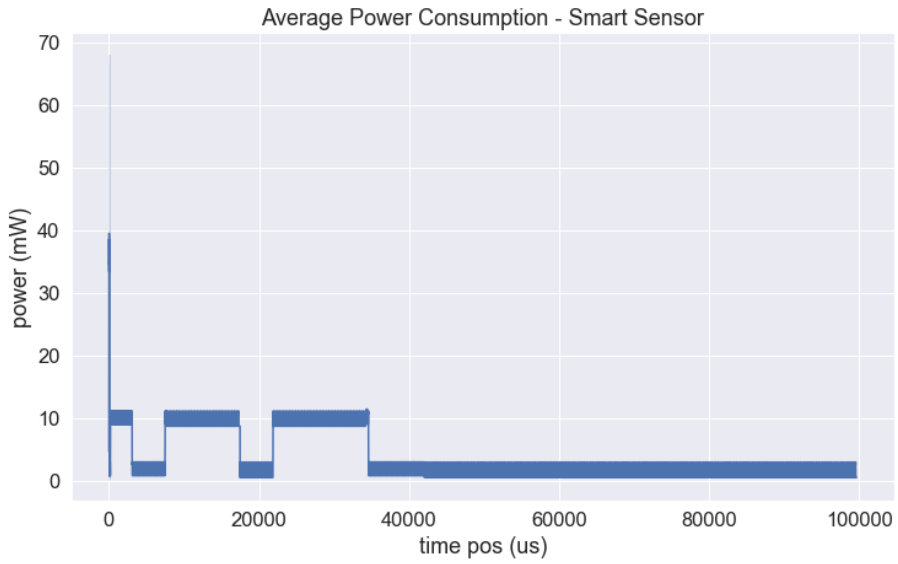

Figure 5.3: Power consumption during the run of the basic microcontroller application

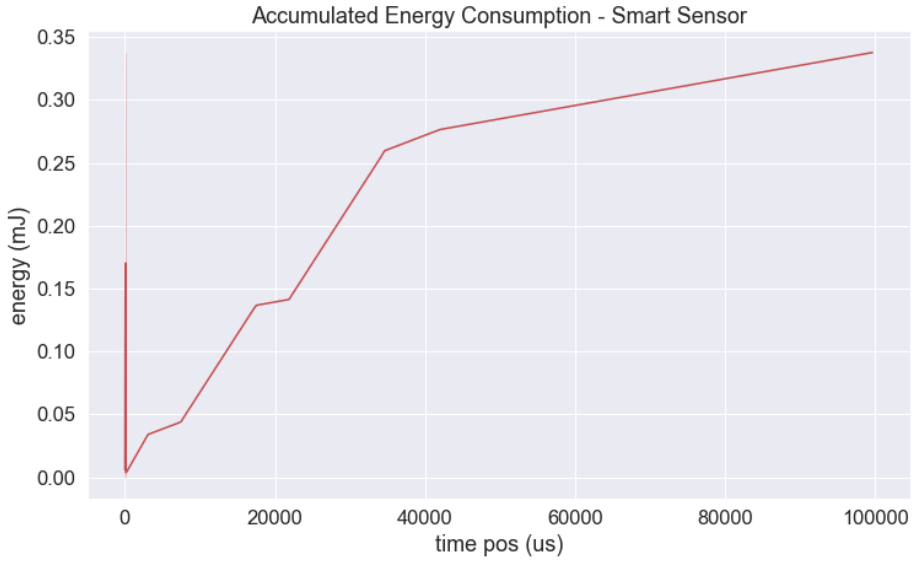

Figure 5.4: Energy consumption traced for the basic microcontroller application

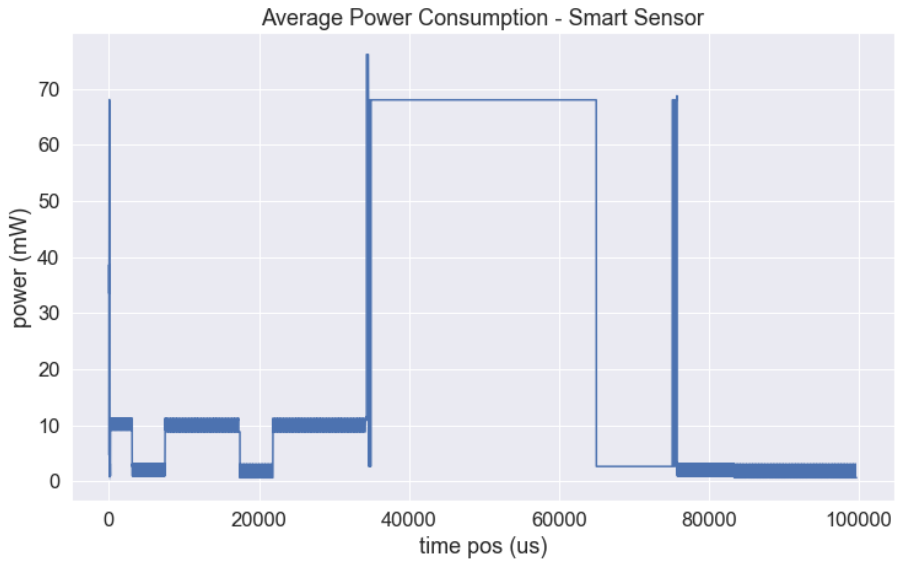

Figure 5.5: Power consumption during the run of the advanced microcontroller application

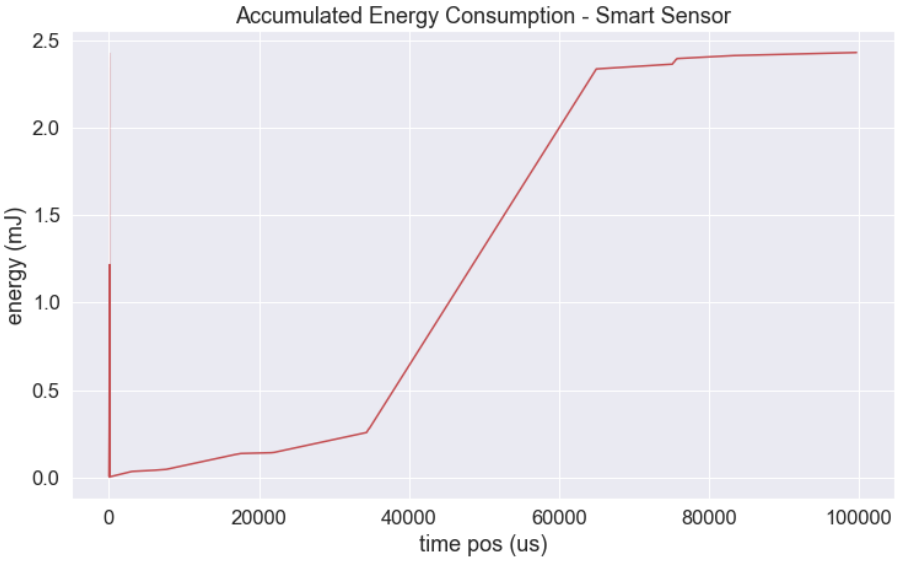

Figure 5.6: Energy consumption traced for the advanced microcontroller application

6 Conclusion and Future Work

In this thesis a solution titled "SimPar" was discussed. The main principle is envisioned as a set of tools and frameworks designed to help in modelling SystemC simulations to be run in a parallel environment. For that reason, a client-server methodology was recommended. The system is primarily intended to work for simulations by which runtime is highly dependant on the input parameters. These simulations generally include modern smart sensors. To that extent, the second phase of the thesis discusses a full design and implementation of one of the smart sensors. The key points established are security, dependability, and power awareness.
The goal which was set during the realization of this project was to create an usable concept which could be applied on independent SystemC simulations. The main challenge was in creating a protocol which can be easily used and implemented from the side of different developers. That was partially fulfilled, but the question remains how approachable this design is and should it be geared more in the direction of reducing the design or the implementation time of a potential simulation.
As a guideline for a possible simulation model, a smart sensor was developed and evaluated on the energy performance. The SystemC design methodology was aimed at functional and energy accuracy. The described approach is accurate, but time consuming. The development also followed a real-world physical implementation. In current time and especially in the near future, design approaches aimed more towards a higher abstract representation of the hardware (i.e. TML design) will be more and more prevalent. In this scenario, it would also be necessary to ensure that the parallel system model of the SimPar can also support different simulation designs.

6.1 Future Work

Since SimPar was developed as a project which could be viewed independently from the point of the two-phase development (simulation handling and the smart-sensor model), so are the revisions generally seen as improvements for either the client-server system, the smart-sensor model or the interconnection between these two phases.

Proposals for the improvement of the client-server system model:

- Add a GUI (Graphical User Interface) for real-time observations
- Make the model more dynamic by adding user or admin interaction during the run-time of the simulation
- Adapt the system with a better protocol and simulation handling

On the other hand, some proposals for the smart-sensor SystemC model include:

- Add a higher and lower abstraction layer representation
- Change the simulation environment for an easier interface for testing and running purposes
- Implement real tests for SystemC model interaction with an environment model (e.g. Gazebo model)

Some of the mentioned proposals for the future work have already been analysed and are being researched. These ideas are further more discussed in their own respective sub-chapters.

6.1.1 Real-time Signal Plotting using a GUI

The main goal of the proposal is to enable "real-time" simulation data reading (from specific signals of a SystemC simulation) during its run-time. The idea behind comes from the fact that the data from a SystemC simulation is only able to be read when the simulation ends its run-time cycle. Depending on the time needed, sometimes it is of use that the data is read during its execution, to make for an easier analysis. Since the main system is programmed primarily in C++ (with SystemC) and partially C language, graphical options are limited and hard to exploit. For that reason

Python has been recommended for use.
The main reasons why Python is recommended to be used above other languages are:

- It allows for a good portability - between different os
- It is easy to use and maintain - code is simple to write and combine
- It has a good support for inner workings with C-based programming languages
- It has support for many graph and GUI oriented libraries

The main technical challenge is trying to find the best approach for the communication between C++ main program and Python script, by still maintaining good performance, time accuracy and reliability. Two main approaches regarding the communication have been proposed. They can be tested using a separate program with a simple SystemC model and using the strategy design pattern. The ideas are:

1. *Define through file*
   - \+ Python updates its script as an animation ("read-time" update definition)
   - \+ Stable and extensible [1]
   - \+ Able to store data and track them individually
   - \+ Portable [2]
   - − Less efficient in execution than standard output communication (uses files)
   - − Possible issues (not tested) with providing the setup to the GUI program in Python, time synchronization, or setting multiple runs
2. *Define with "Named Pipes"* by Munagekar [29]
   - \+ More conventional and efficient approach (use a memory location and redirect the output, no need for saving the data on disk)

[1] by adding additional signals, make changes independent on the operating system, ...
[2] between Unix systems and Windows

+ Easier synchronization between C++ and Python (better than conventional pipes, since one program will wait until the other comes to a specific command)
- Not cross-platform [3]
- Does not directly save values

It is possible that for the GUI extension, other possible solutions might be suggested, among which is the use of QT library for the C++ or even Node.js for a higher abstract programming.

6.1.2 Handle Terminal User Interaction

This improvement is mainly proposed on the client-server system side, since it concerns how the environmental simulation supervision is to be handled. Currently, the system is implemented to handle the client handshake request and subsequent simulation requests automatically. While that proves to handle most of the tasks efficiently and also helps in reducing the direct user involvement, it does not leave much room open for improvements. It would also be helpful if there was a better error-handling schema with status monitoring from both the server and partially from the client side.

The work in this field is suggested to be done with the implementation of additional terminal commands and connections. The following is the proposal for which the server would be able to directly send commands to a client via the *user interaction* (as a precursors to the "Dynamic Simulations"):

- Create a "special client" which works as a Telnet protocol system, taking inputs and giving outputs from that terminal.
- The server would receive this messages in a separate thread from the special client and would put them in a queue in a global variable. The reason for a global variable is that it is one of the ways to enable the threads to take values from the main process.

[3] UNIX-based, but there are ways to make it on Windows

- Variables are still separate vectors of strings, hence currently, there is no additional need for semaphores or mutex (although depending on the performance issues, it could be rearranged).
- The special client would also be able to use commands to list all the connected clients and similar options to allow him for an easier control. Before sending a command, an appropriate "*code*" would have to be written in front of the message. That code is used by the Server to find the correct client.

A potential issue with this proposal is that the simulation clients work on a protocol principle and always send their messages first. It can be blocking in regards to server handling [4]. A solution for this problem would include *timeouts*, allowing for a double-sided connection and changing the protocol to be controlled on the server side rather than on the client side.

6.1.3 Smart Sensor SystemC Model Improvements

The developed smart sensor simulation model fulfils most of the set requirements, but it is always open for improvements. One of those include a more accurate handling of the clock signal. While the model was developed to handle the different frequencies of the I2C protocol, an issue eventually was shown where it was difficult to accurately synchronise different processes without impacting the overall simulation by a large margin. Currently, all I2C clock signals run at 1MHz, but some modules support also a smaller frequency for a higher energy efficiency.
Additionally, a better battery module can be implemented with more accurate readings and statuses. This is especially evident with the charge routine, which currently automatically charges the battery.
An important feature would be the extension of the error and the status handling by the RF functionality of the RF Scanner and Tag, to make it a proper emulated NFC chip. While this feature would not give better energy readings or the end results, the mentioned potential implementation would effectively cover most of the hardware functional options on the software level, rounding up the emulation design approach.

[4] "recv function" of the socket functions is blocking

Bibliography

[1] Mello Aline et al. "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '10. Dresden, Germany: European Design and Automation Association, 2010, pp. 606–609. ISBN: 978-3-9810801-6-2. URL: http://dl.acm.org/citation.cfm?id=1870926.1871069 (cit. on pp. 17–19, 23).

[2] Tanenbaum Andrew. *Computer Networks*. 4th. Prentice Hall Professional Technical Reference, 2002. ISBN: 0130661023 (cit. on pp. 25, 26).

[3] Apollo2. *Apollo2 Official Documentation*. June 2017. URL: https://ambiqmicro.com/apollo-ultra-low-power-mcus/apollo2-mcu/ (visited on 10/26/2018) (cit. on p. 55).

[4] Vaishali M. Bagade and M. B. Limkar. "VHDL Model of Smart Sensor." In: *International Journal of Science, Engineering and Technology* 2 (2014), pp. 817–826. ISSN: 2348-4098. URL: http://ijset.in/wp-content/uploads/2014/07/IJSET.0720140230.1011.1807_Vaishali_817-826.pdf (cit. on p. 22).

[5] Gilad Ben-Yossef. *On Threads, Processes and Co-Processes*. CELF ELC Europe 2009. 2009. URL: https://elinux.org/images/1/1c/Ben-Yossef-GoodBadUgly.pdf (cit. on p. 8).

[6] Kapil Bhardwaj. *When Singleton Becomes an Anti-Pattern*. Oct. 2017. URL: https://dzone.com/articles/singleton-anti-pattern (visited on 09/15/2018) (cit. on p. 36).

[7] David C. Black et al. *SystemC: From the Ground Up, Second Edition. The Science of Microfabrication*. Vol. 2. Springer US, 2010. ISBN: 978-0-387-69958-5. DOI: 10.1007/978-0-387-69958-5 (cit. on pp. 2, 6, 10).

[8] Olivier Bonaventure. *Computer Networking: Principles, Protocols and Practice*. Boston, MA, USA: The Saylor foundation, 2011. URL: https://www.saylor.org/site/wp-content/uploads/2012/02/Computer-Networking-Principles-Bonaventure-1-30-31-OTC1.pdf (visited on 09/25/2018) (cit. on p. 28).

[9] Eric D. Crahen. *The Guard Idiom: Enhancing the Scoped Locking Idiom*. May 2003. URL: http://www.drdobbs.com/cpp/the-guard-idiom-enhancing-the-scoped-loc/184401644 (visited on 09/20/2018) (cit. on p. 40).

[10] Alexander De Graaf. *SystemC: an overview*. ET 4351. Sept. 2014. URL: http://bd.eduweb.hhs.nl/es/systemc/SystemC-14v1.pdf (cit. on p. 44).

[11] Frederic Doucet and Rajesh K. Gupta. "Microelectronic System-on-Chip Modeling using Objects and their Relationships." In: *IEEE D and T of Computer*. 2000. URL: http://mesl.ucsd.edu/mesl-website/pubs/doucet-osee1.pdf (cit. on p. 46).

[12] W. Du, F. Mieyeville, and D. Navarro. "Modeling Energy Consumption of Wireless Sensor Networks by SystemC." In: *2010 Fifth International Conference on Systems and Networks Communications*. Aug. 2010, pp. 94–98. DOI: 10.1109/ICSNC.2010.20 (cit. on pp. 20, 21, 23).

[13] Ahmed Elshamy and Amr Elssamadisy. "Divide After You Conquer: An Agile Software Development Practice for Large Projects." In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 164–168. ISBN: 978-3-540-35095-8 (cit. on p. 1).

[14] P Ezudheen et al. "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines." In: *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*.

PADS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 80–87. ISBN: 978-0-7695-3713-9. DOI: 10.1109/PADS.2009.25. URL: http://dx.doi.org/10.1109/PADS.2009.25 (cit. on pp. 15, 16).

[15] Charles Fulks. *FPGA Familiarization*. The International System Safety Society - Tennessee Valley Chapter. May 2016. URL: https://system-safety.org/issc2016/T07_FPGA.pdf (cit. on p. 2).

[16] Alfons G. Hoekstra et al. "Modelling and Simulation of Automatic Debiting Systems for Electronic Toll Collection on Motor Highways." In: 1997 (cit. on p. 21).

[17] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (cit. on pp. 33, 35, 37, 40).

[18] GazeboSim. 2018. URL: http://gazebosim.org (visited on 08/06/2018) (cit. on p. 5).

[19] Brian Hall. *Beej's Guide to Network Programming Using Internet Sockets*. Beej's blog web site. June 2016. URL: https://beej.us/guide/bgnet/pdf/bgnet_USLetter.pdf (visited on 11/02/2018) (cit. on p. 67).

[20] Carl Hauser et al. "Using Threads in Interactive Systems: A Case Stud." In: Dec. 1993, pp. 94–105 (cit. on p. 27).

[21] K. Huang et al. "Scalably distributed SystemC simulation for embedded applications." In: *2008 International Symposium on Industrial Embedded Systems*. June 2008, pp. 271–274. DOI: 10.1109/SIES.2008.4577715 (cit. on pp. 15, 23).

[22] IoSense. 2016. URL: http://www.iosense.eu/ (visited on 08/15/2018) (cit. on p. 1).

[23] Phillip Keane. Apr. 2018. URL: https://www.engineering.com/DesignSoftware/DesignSoftwareArticles/ArticleID/16724/Cloud-Simulation-Where-Are-We-Now.aspx (visited on 08/06/2018) (cit. on p. 5).

[24] Dave Kelf. "Linking high-level synthesis with formal verification." In: *Tech Design Forum* (2015). URL: http://www.techdesignforums.com/

practice/technique/onespin-systemc-hls-formal-verification/ (cit. on p. 6).

[25] James Lapalme et al. "Separating Modeling and Simulation Aspects in Hardware/Software System Design." In: *Microelectronics, 2006. ICM '06* (Jan. 2007), pp. 202–205 (cit. on p. 4).

[26] Mehran Massoumi. *Hardware Description*. HDL Research & Development, Averant Inc., USA. 2001. URL: https://www.eolss.net/sample-chapters/C15/E6-45-02-11.pdf (cit. on p. 2).

[27] S. Mazor. "The history of the microcomputer-invention and evolution." In: *Proceedings of the IEEE* 83.12 (Dec. 1995), pp. 1601–1608. ISSN: 0018-9219. DOI: 10.1109/5.476077 (cit. on p. 2).

[28] R. L. Miller. "Fractional-Frequency Generators Utilizing Regenerative Modulation." In: *Proceedings of the IRE* 27.7 (June 1939), pp. 446–457. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1939.228513 (cit. on p. 57).

[29] Abhishek Munagekar. *Write an IPC program using pipe using C++ and Python*. Aug. 2016. URL: https://prgwonders.blogspot.com/2016/08/write-ipc-program-using-pipe-using-c.html (visited on 09/29/2018) (cit. on p. 97).

[30] Dhanwada Nagu, Lin Ing-Chao, and Narayanan Vijay. "A Power Estimation Methodology for systemC Transaction Level Models." In: *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '05. Jersey City, NJ, USA: ACM, 2005, pp. 142–147. ISBN: 1-59593-161-9. DOI: 10.1145/1084834.1084874. URL: http://doi.acm.org/10.1145/1084834.1084874 (cit. on p. 19).

[31] D. Navarro, G. Migliato-Marega, and L. Carrel. "Battery-less Near Field Communication Sensor Tag Energy Study with ContactLess Simulator." In: *ICWNC 2017m The Thirteenth International Conference on Wireless and Mobile Communications* 2 (2017), pp. 63–66. ISSN: 2308-4219. URL: https://www.thinkmind.org/index.php?view=article&articleid=icwmc_2017_3_40_28008 (cit. on pp. 22, 23).

[32] TrustX Optiga. *Optiga TrustX Official Documentation*. Feb. 2018. URL: https://www.infineon.com/cms/en/product/security-smart-

card-solutions/optiga-embedded-security-solutions/optiga-trust/optiga-trust-x-sls-32aia/ (visited on 10/25/2018) (cit. on p. 53).

[33] S. Park et al. "Concurrent simulation platform for energy-aware smart metering systems." In: *IEEE Transactions on Consumer Electronics* 56.3 (Aug. 2010), pp. 1918–1926. ISSN: 0098-3063. DOI: 10.1109/TCE.2010.5606347 (cit. on pp. 14, 18).

[34] W. Thomas Pieber, Thomas Ulz, and Christian Steger. "SystemC Test Case Generation with the Gazebo Simulator." In: (Jan. 2017), pp. 65–72. DOI: 10.5220/0006404800650072 (cit. on pp. 1, 3, 4).

[35] A. El-Sayed et al. "A survey on recent energy harvesting mechanisms." In: *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. May 2016, pp. 1–5. DOI: 10.1109/CCECE.2016.7726698 (cit. on p. 10).

[36] Douglas Schmidt. "Wrapper Facade - A Structural Pattern for Encapsulating Functions within Classes." In: *C++ Report Magazine* (Jan. 1999) (cit. on pp. 34, 35).

[37] Douglas C. Schmidt et al. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Chichester, UK: Wiley, 2000. ISBN: 978-0-471-60695-6. URL: https://www.safaribooksonline.com/library/view/pattern-oriented-software-architecture/9781118725177/ (cit. on p. 33).

[38] D. Schmidt et al. "Energy modelling in sensor networks." In: *Advances in Radio Science* 5 (2007), pp. 347–351. DOI: 10.5194/ars-5-347-2007. URL: https://www.adv-radio-sci.net/5/347/2007/ (cit. on pp. 20, 23).

[39] C. Schumacher et al. "parSC: Synchronous parallel SystemC simulation on multi-core host architectures." In: *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Oct. 2010, pp. 241–246. DOI: 10.1145/1878961.1879005 (cit. on pp. 15, 23).

[40] Mel Siegel. "Sensor Modeling and Simulation: Can it pass the Turing Test?" In: 2001 (cit. on p. 21).

[41] R. Sinha, A. Prakash, and H. D. Patel. "Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs." In: *17th Asia and South Pacific Design Automation Conference*. Jan. 2012, pp. 455–460. DOI: 10.1109/ASPDAC.2012.6164991 (cit. on p. 15).

[42] ST25DV64K. *ST25DV64K Official Documentation*. Dec. 2017. URL: https://www.st.com/en/nfc/st25dv64k.html (visited on 10/25/2018) (cit. on p. 49).

[43] J. Teich. "Hardware/Software Codesign: The Past, the Present, and Predicting the Future." In: *Proceedings of the IEEE* 100.Special Centennial Issue (May 2012), pp. 1411–1430. ISSN: 0018-9219. DOI: 10.1109/JPROC.2011.2182009 (cit. on pp. 2, 3).

[44] T. Ulz et al. "SECURECONFIG: NFC and QR-code based hybrid approach for smart sensor configuration." In: *2017 IEEE International Conference on RFID (RFID)*. May 2017, pp. 41–46. DOI: 10.1109/RFID.2017.7945585 (cit. on p. 9).

[45] Zhang Yong et al. "Progress of smart sensor and smart sensor networks." In: *Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788)*. Vol. 4. May 2004, 3600–3606 Vol.4. DOI: 10.1109/WCICA.2004.1343265 (cit. on p. 41).