CHALMERS

Distributed Simulations of Automotive Related Real-Time Systems Using the High Level Architecture Standard

Master of Science Thesis

OLA JAKOBSON

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2012

Report No. 2012:080
ISSN: 1651-4769

REPORT NO. 2012:080
ISSN: 1651-4769

Distributed Simulations of Automotive Related Real-Time Systems Using the High Level Architecture Standard

OLA JAKOBSON

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2012

Distributed Simulations of Automotive Related Real-Time Systems Using the High Level Architecture Standard
OLA JAKOBSON

Technical Report No. 2012:080
ISSN: 1651-4769
Department of Applied Information Technology
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone +46 (0)31–786 2777

Gothenburg, Sweden, 2012

Distributed Simulations of Automotive Related Real-Time Systems
Using the High Level Architecture Standard
OLA JAKOBSON
Department of Applied Information Technology
Chalmers University of Technology

ABSTRACT

In this project, the standard High Level Architecture (HLA) has been used in the area of automotive related distributed simulation systems. The purpose was to evaluate the possibility to use HLA for verification of such systems. This has been performed by building and evaluating three different prototypes. The simulation members, *federates*, of the prototypes had real-time requirements with guidelines of being able to exchange data corresponding to an amount of 10 000 bits, within 10 ms and with an update period of 10 ms. The first prototype involves federates transferring data between two Hardware-In-the-Loop (HIL)-simulators from dSPACE. The second and third involve federates executing in MATLAB and Simulink, respectively.

The results of the prototypes indicate that it most likely is possible to use HLA in such simulation systems. However, the amount of work needed to make everything function varies from case to case and sometimes tend to be relatively high. Regarding the distributed data being sent between federates, the wanted way would be to pack data into containers of small size. Doing this would result in the federates being able to only receive the data they are actually interested in. However, handling a large amount of small size containers with data within a short time period is very computationally demanding. Thus, in order to succeed with the MATLAB and Simulink prototypes with respect to the requirements, data had to be packed into larger containers. Doing that made it possible to transfer data amounts many times greater than the stated data criterion.

Another use-case that might be even more promising regarding HLA and the automotive industry came up during the thesis work. It involves running distributed Simulink models *offline* (without the real-time requirements) executed time-synchronized, with help of HLA.

Keywords: High-Level Architecture, HLA, distributed simulations, real-time systems, Run-Time Infrastructure, Hardware-In-the-Loop, HIL, automotive, Pitch Technologies, ForwardSim

PREFACE

This report will present the project that has served as the author's Master of Science thesis at Chalmers University of Technology, performed during January 2012 to June 2012. The major work has been carried out within the group Electrical Integration Environments (94920) at the Electrical Department of Volvo Cars, located in Torslanda, Gothenburg Sweden.

ACKNOWLEDGMENTS

I would like to thank Martin Nilsson, Annica Normén and the rest of the people at Volvo Cars which have helped and supported me throughout this project. Thanks are also given to Björn Fridholm who has been of great help during my thesis work. I would also like to thank Claes Strannegård from Chalmers for being my examiner.

Furthermore, thanks are given to the people working at Pitch Technologies and ForwardSim, for all technical support they have been giving me.

Last but not least, I would like to thank my girlfriend and my friends for being supportive and helpful during this thesis work. Without you, this would not have been possible.

Ola Jakobson, June 2012

CONTENTS

LIST OF FIGURES

ACRONYMS

API	Application Programming Interface
CAN	Controller Area Network
CRC	Central RTI Component
ECU	Electronic Control Unit
FDD	FOM Document Data
FOM	Federation Object Model
HIL	Hardware-In-the-Loop
HLA	High Level Architecture
LAN	Local Area Network
LRC	Local RTI Component
OMT	Object Model Template
pRTI	portable Run-Time Infrastructure
RTI	Run-Time Infrastructure
RTP	Real-Time Processor
RTT	Round-Trip Time
SOM	Simulation Object Model
TCP	Transmission Control Protocol
TRC	Variable Description File
UDP	User Datagram Protocol
XML	eXtensible Markup Language

Part I

INTRODUCTION

1 INTRODUCTION

Performing simulations on systems is an important approach when developing and verifying products, especially considering the automotive industry. With simulations one can predict and evaluate the behaviour of a system, such as the electronics in a car, in an early stage before the car is actually built. It is important to find faults and correct these as early in the development procedure as possible since they will cost multiple times more to correct in a later stage. Hardware-In-the-Loop (HIL)-simulation is a type of real-time simulation, involving interactions with real components such as several Electronic Control Units (ECUs), used in for example cars. Such simulators allow validation of both hardware and software. HIL-simulations executed in real-time are usually very computationally demanding and therefore run on dedicated real-time simulators. In the automotive industry a single HIL-system is often, for various reasons, specialised on a certain area of a car such as the motor or the electrical system. It would however be of interest to share particular information between different simulator systems. This could for example benefit from only requiring one HIL-system calculating and distributing certain repetitive data. The data could then be delivered to several other systems, instead of forcing each individual system to perform the same calculations on their own. However, in this type of *distributed simulation*, the data exchange between the different simulators need to be kept at minimum delay in order for the simulations to work properly. The delay requirement is different depending on, for example, what sort of data being exchanged, although common update intervals are in the range of 1-20 ms.

In this thesis, the High Level Architecture (HLA) standard has been evaluated in the area of automotive related distributed simulation systems. HLA can be used to share data in distributed simulation systems via the widely spread Ethernet protocol [1]. HLA originates from simulations performed during military training which could involve thousands of members sharing data. It has also been introduced to other areas, such as medical simulations [2], emergency management [3] and traffic simulations [4]. The standard is, however, not yet established in the automotive industry.

1.1 PREVIOUS WORK

There are a couple of existing protocols available in the automotive industry which can be used to distribute data between different simulator systems, such as Controller Area Network (CAN) [5] and FlexRay [6]. They are however quite limited in terms of bandwidth, and more importantly the maximum possible length of cables. FlexRay, which has the highest bandwidth of the two protocols, only supports a length of around 20 meters between connected nodes. CAN supports up to around 500 meters but when using a long distance the bandwidth becomes significantly lower. Therefore they are very unsuitable if the simulator systems are in different rooms, different buildings or even further away from each other.

There also exist brand specific solutions such as Gigalink from dSPACE [7]. Gigalink makes it possible to exchange data between several real-time simulators, with low delays. Two large drawbacks are that those systems only can be connected to other dSPACE HIL-simulators and the distance between them is limited to around 100 meters.

1.2 PURPOSE

The purpose of this thesis is to evaluate if it is possible to use HLA for verification of distributed simulations, in the area of automotive related systems.

1.3 SCOPE

Different case-studies have been prototyped and tested in order to evaluate HLA. They are described in the case-studies, Part IV. Case one, the *prototyping of HIL-simulator federates*, is considered the main focus of the thesis.

1.4 THESIS STRUCTURE

Part I gives an introduction and presents the purpose and scope of the thesis.

Part II contains background material needed for the understanding of this project. It covers different kind of computer simulations, important parts of HLA and Ethernet network related delays.

Part III describes an overall method covering the common things in the built prototypes of the case-studies. This includes general setups and how the HLA participants in the prototypes communicate during code execution. It also includes the prototypes' evaluation criteria and a list with development tools used during the project.

Part IV contains four case-studies. The first three involves building and evaluation of three different prototypes, each of them including two HLA participants. These three case-studies come with description, implementation, results, analysis and future work, for respective prototype. The fourth case-study deals with cases involving more than two HLA participants.

In *Part V* the thesis is summarized with conclusions regarding all the case-studies.

Part VI contains the appendix.

Part II

BACKGROUND

2 BACKGROUND

This chapter describes important background knowledge that the reader needs to be familiar with when reading this thesis. It covers different kind of simulations and gives a rough understanding of HLA and its components which have been used during the thesis.

2.1 COMPUTER SIMULATIONS

In general, a computer simulation is a model of a particular system, which has been implemented to run on some sort of computer, or *simulator*.

2.1.1 *Real-Time Simulations*

A real-time simulation system is a computer model of a physical model, being executed on a real-time computer [8]. The type of digital simulations in this thesis assumes simulations to run in discrete-time with constant step durations. This means that time moves forward with equal duration in every time step, known as *fixed time-step simulation*. In every step of a discrete time simulation a number of equations and functions, which represent the modeled system, are processed. The amount of real time required to perform those calculations could be shorter or longer than the simulation's time step.

A *shorter* time usually means that the simulation runs faster than real time, known as an accelerated simulation. *Longer* time corresponds to a simulation running slower than real time. These two types typically execute as fast as possible and the solving time depends primarily on the amount of available computing power and the complexity of the simulation model. The opposite of these would be a real-time simulation. Such simulator needs to compute all the calculations of the current time step in the simulation during the same length of time that its corresponding physical representation would have needed. However, when hardware is involved such as during HIL-simulations, it has to go even faster. The reason is that here the simulator also needs to perform additional tasks such as handling input and output ports of connected devices. If everything works as supposed, it should be some time left from the moment the simulator has finished everything in the current time step, before it is time to start with the next one. This time slot is called the *idle-time*. Idle-time does not exist in accelerated simulations, since those would have executed the simulation as fast as possible. However, during the idle-time in a real-time simulator, the simulator keeps waiting until the clock ticks to the next time step.

If the simulator's computational capacity is exceeded during a certain time step, an *overrun* will occur. In order to guarantee that the simulator works properly, overruns must be avoided by, for example, decreasing the work of the simulator.

2.1.1.1 *Hardware-In-the-Loop (HIL) Simulations*

HIL-simulation is a type of real-time simulation that includes real components in the simulation [9]. In the automotive industry such components are commonly different kinds of ECUs [10]. An ECU could for example be a Brake Control Module (BCM) handling the brake system of a car or a Climate Control Module (CCM) taking care of the air conditioning among other things.

The intention with a HIL-simulator is to provide all connected ECUs with proper *electrical stimulation* needed to fully exercise all ECUs. In other words, this means tricking the ECUs into thinking that they are connected to a real plant, for example a car. There are several benefits associated with HIL-simulations. A couple of automotive related examples include:

- Testing of ECUs in an early stage. Functionality can be verified without the need of a real car, which usually does not even exist in early stages.
- Completely testing and verifying ECUs is a very costly and time demanding procedure if they are to be performed in a real car. Performing these tests in a HIL-simulator would be less time demanding and more cost efficient in the long run.
- Some test cases could be potentially harmful in a real car, such as verifying that over-temperature protections works as supposed. Such tests can instead be simulated to verify that the specific ECU detects the error and reports it properly.

2.1.2 *MATLAB and Simulink*

MATLAB, which stands for MATrix LABoratory, is software developed by MathWorks [11]. It is a high-performance interactive programming environment for algorithm development, data visualization, data analysis and numerical computation [12]. As the name reveals, MATLAB is good with handling data within matrices. In MATLAB a matrix does not require dimensioning, which allows easier solving of many technical vector- and matrix related problems, in comparison with other non-interactive languages such as C and Fortran.

MATLAB features several add-ons, known as *toolboxes*. A toolbox is an application-specific solution, usually used and developed for some specialized technology. Toolboxes are available in many areas such as signal processing, control systems, wavelets and simulation.

Simulink, also developed by MathWorks, is a simulation environment integrated with MATLAB. Simulink provides a graphical environment for model-based design of dynamic and embedded simulation systems [13]. It includes a set of block libraries making it possible to design, simulate, implement and test various time-varying systems. In the same way that MATLAB has add-on toolboxes, Simulink can be extended with more application-specific block libraries.

2.1.3 *dSPACE*

dSPACE is a company with both hardware and software solutions for performing real-time HIL-simulations, among lots of other types of embedded solutions [14]. Computer models created with Simulink can, with help of dSPACE's Real-Time Interface software [15], be compiled into *applications* which can run on dSPACE hardware.

The most central part of a dSPACE HIL-simulator is the *processor board*. Communication with the processor board can be performed through a regular PC, known as the *host computer*. On this host computer, applications can be uploaded to the processor board in order for them to run on the real-time simulator. It is also usually via the host computer that test cases are executed and monitored. This can be done with the dSPACE software ControlDesk [16].

2.1.3.1 *Variable Description File (TRC)*

When a Simulink model is compiled into an application using dSPACE's Real-Time Interface software, a Variable Description File (TRC) is generated. This file contains model-specific information such as parameters, signals, data types and address locations. This information is necessary in order to observe and manipulate parameters and signals when the application runs in the simulator. The information could for example be used by ControlDesk, MLIB and CLIB.

2.1.3.2 *MLIB*

MLIB is a MATLAB to dSPACE interface library [17]. The library makes it possible to transfer data between a dSPACE application and MATLAB, running on the host computer, which the dSPACE simulator is hooked up to. MLIB has a set of ready-made functions that could be used to interact with the Real-Time Processor (RTP) running on the processor board. MLIB is based on the functions from the low-level library CLIB, described below.

2.1.3.3 *CLIB*

CLIB is a C-code to dSPACE interface library [18]. The library works in a similar way as MLIB, but function calls are instead made with C-code. CLIB is even more low-level than MLIB, which makes CLIB more promising in terms of data transfer delays, but with an increased complexity. In comparison with MLIB, it comes with less ready-made functions.

2.2 HIGH LEVEL ARCHITECTURE (HLA)

HLA has been an open standard for distributed simulation since 1996. It has been developed by the United States Department of Defence (DoD) and was initially commonly used by military forces to perform training simulations [19]. Such simulations could involve thousands of soldiers, vehicles, aircrafts etcetera, all connected together exchanging information through a simulation. In HLA, each participating member that want to take part of the simulation is called a *federate*. All information exchange is performed with help of the Run-Time Infrastructure (RTI). The information exchange follows certain federation agreements and something called Federation Object Model (FOM). The FOM contains information about all elements that are being shared among two or more federates. All federates together with the RTI and the FOM represents a *federation*.

HLA is not bound to any specific computing platforms. Whether or not a specific operative system and programming language within that system is compatible with HLA depends completely on the RTI. It is also the RTI that decides which sort of network protocols are being supported. The most commonly supported protocols are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), through Ethernet, Wi-Fi or a 3G connection.

The latest version of the HLA standard is the IEEE 1516-2010 or more commonly known as *HLA Evolved*. It offers a lot of different functionality for distributed simulation systems. However, in this thesis, only a few parts of the HLA functionalities are used. Instead, the focus lies on trying to adapt and make the prototypes of the case-studies HLA compatible and keep the data transfer delays at a minimum. Some of the most central parts of HLA have been described in the sections below.

2.2.1 *Run-Time Infrastructure (RTI)*

Federates must be connected to each other through an RTI. The RTI consists of two parts: the Central RTI Component (CRC) (also known as the *RTIexec*) and the Local RTI Component (LRC), as shown in Figure 1. It is the CRC that manages the whole federation. All fed-

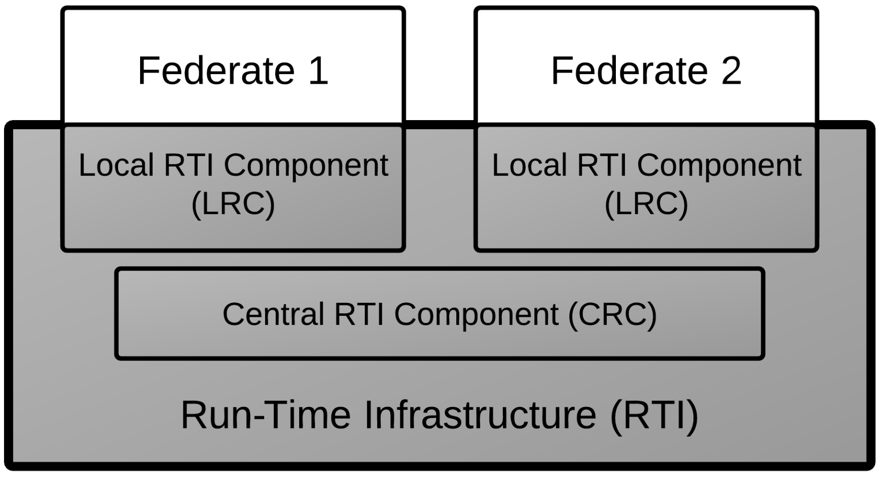
Federate 1
Federate 2
Local RTI Component (LRC)
Local RTI Component (LRC)
Central RTI Component (CRC)
Run-Time Infrastructure (RTI)

Figure 1: The grey-faded area represents the complete RTI, consisting of a CRC and one LRC for each federate (in this case two).

erates that want to take part of the federation must connect to the CRC. When a federate joins, the CRC will provide it with necessary information such as other currently joined federates and how to interact with them. Each federate is compiled and linked to an LRC. The LRC contains all methods and classes needed for the federate to be able to communicate properly with the CRC and with other federates. A federate must perform all type of communication with the rest of the federation through its own LRC. A federate makes a *call* to its LRC when it wants to communicate with the CRC or another federate, for example sending data. In a similar way, when the CRC wants to communicate with a federate, the federate's LRC will invoke a *callback*.

All parts of a federation can either run on a single computer or run distributed over several computers. For example, Federate 1 (with its LRC) can run on one computer, Federate 2 (with its LRC) on a second computer and the CRC on a third computer. Another configuration would be to let the CRC run on the same computer as one of the federates. Each federate must however be linked to its own LRC and thus be located on the same computer, the rest is up to the implementer. If the federation runs distributed over two or more computers, the communication between them goes via some sort of network. The network is usually a Local Area Network (LAN), using Ethernet.

Since HLA is an open standard, it is possible to code an RTI on your own. However, this would be a very demanding procedure, especially if the intention is to make it fully HLA compatible, support multiple federate programming languages, have low latencies etcetera. There exists a couple of open source RTIs. One of them is called Open HLA [20] developed in Java. Open HLA only supports federates written in Java and is not fully HLA compatible, among several other drawbacks. Another open source RTI is poRTIco [21]. poRTIco does have an Application Programming Interface (API) in both Java and C++ but in a relatively limited manner and does not support the latest HLA version (HLA Evolved). Since open source projects are free they are usually not state of the art, lack certain functionality and have very limited support if problems occur. These drawbacks make these RTIs unsuitable for this thesis. Instead, a complete high-performing RTI solution has been bought from the company Pitch Technologies [22].

2.2.1.1 *Pitch Technologies*

Pitch Technologies is a Swedish company that offers several HLA related products with various functions. Examples of such are software for easier development of federates, visual software for producing FOM files, software with HLA recorder utilities etc. However, the most important product is the RTI itself, named portable Run-Time Infrastructure (pRTI) [23]. The pRTI is fully compatible and certified with the latest version of the HLA standard and currently supports federates written in C++ and Java. It offers high performance and is tweakable in terms of how data packages are sent between federates. These were some of the reasons why pRTI was considered to be well-suited for this thesis.

2.2.1.2 *ForwardSim*

The pRTI does not support models written in MATLAB or Simulink. In order to convert such models into HLA federates, some sort of *gateway* is needed. This is where the Canadian company ForwardSim [24] can help. They provide two products named HLA Toolbox and HLA Blockset. HLA Toolbox extends MATLAB with a library which makes it possible to, in regular MATLAB function files, make all sort of interactions with an HLA RTI. The HLA Blockset includes specific Simulink blocks, making it possible for Simulink models to interact with an HLA RTI.

2.2.2 *Objects and Interactions*

Federates can share data with other federates using *objects* and *interactions*. An HLA *object* is something that persists over time during federation execution. Objects have one or several *attributes*. An example of an object could be a car. The car's attributes might then be data containing its current speed, position and fuel level. One federate owns the car and provides the RTI with updated values of the attributes, usually with some sort of decided frequency. Other federates have the possibility to, with help of the RTI, take part of these attributes and get updates when there is new data available.

An HLA *interaction* is a one time event that *does not* persist over time during federation execution. Interactions have one or several *parameters*. Interactions are usually used to handle instantaneous events. An example of this could be text communication between federates. Such an interaction could be implemented consisting of two parameters; one with some sort of sender identification number and the other containing a text message. Federates then have the possibility to, with help of the RTI, send messages using this interaction. When an interaction is sent, all federates that have *subscribed* to that interaction class will get those parameters.

2.2.3 *Federation Object Model (FOM) and Simulation Object Model (SOM)*

A FOM describes all information that *is shared* between two or more federates in a federation. Examples of such information are names of interactions, parameters, objects and attributes, together with their respective data types.

A Simulation Object Model (SOM) is closely related to a FOM. The difference is that a SOM is individual for each federate and describes the *possible* information which that particular federate could offer.

All FOMs and SOMs must follow the format described by the Object Model Template (OMT), which is part of the HLA standard. Following that format results in an eXtensible Markup Language (XML) file. Thus, FOM files and SOM files are always some kind of XML files.

2.2.3.1 *FOM Document Data (FDD)*

During federation run-time, the CRC needs to know certain things from the FOM file. Thus, when the federation is created, parts of the FOM file known as the FOM Document Data (FDD) is being provided to the CRC. These things include, for example, names of interactions and parameters that will be shared in the federation.

2.2.3.2 *Visual OMT*

Visual OMT [25] is software from Pitch used to create and maintain object models such as FOMs and SOMs. This is done in a graphical easy-to-use interface.

2.3 MEASURING TIME DELAYS BETWEEN HLA FEDERATES

Synchronizing the time on two regular computers to measure one-way delays is difficult and not very accurate. The easiest way of measuring elapsed time between two federates located on different computers is instead to measure the Round-Trip Time (RTT) [26]. The RTT is the amount of elapsed time it takes for one (or several) parameter(s) with data to travel from the first computer to the second one and then back again. If the computers are considered equal, both in terms of hardware and software, one can also easily obtain a feasible one-way delay by dividing the RTT value by two.

Measurements in this thesis are in overall performed equal, independent of what language the federates have been implemented in. The federates are implemented to perform the general steps below, when RTT tests are being carried out. The sending and receiving of data is performed with help of the respective federate's LRC. The different languages have, of course, their own way of measuring elapsed time.

1. Federate 1 starts the time measurement and sends a chosen amount of data to Federate 2.
2. Federate 2 receives that data and then sends an equal amount of data back to Federate 1.
3. Federate 1 receives the data sent by Federate 2 and then stops the time measurement.

The three steps above are iterated to produce a proper mean value of the specific test being run.

2.4 DELAYS ON ETHERNET NETWORKS

Ethernet networks have different sources of delays, some of which have been described briefly below.

2.4.1 *Transmission Delay*

The time it takes for sending an Ethernet package with data, known as a *frame*, between two devices (such as computers) is highly dependent on the bandwidth. The following formula can be used to calculate the transmission delay:

$$D_{Transmission} = \frac{Frame\ Size\ [bits]}{Bit\ Rate\ [bits/second]}$$

For example, the maximum sized Ethernet frame is 1500 bytes, which equals $1500 \cdot 8 = 12\,000$ bits. On a 100 Mbit/s network, the delay would be:

$$D_{Transmission} = \frac{12000}{100 \cdot 10^6} = 120\,\mu s$$

In a one gigabit network (1000 Mbit/s) the corresponding delay would only be $12\,\mu s$. However, since each frame need some overhead such as destination address, the maximum efficiency is approximately 97% (when using the maximum frame size of 1500 bytes) [27].

2.4.2 *Delays Associated with Switches*

Ethernet switches involve different kind of delays. A switch could work either using the *store-and-forward* technique or the *cut-through* technique. Using store-and-forward, the whole frame is stored in the memory of the switch before it is being forwarded on the proper out port. Here, the switch has the advantage to throw away a frame if it is damaged, but with the drawback of introducing a delay with the size of a corresponding transmission delay. Instead, by using the cut-through technique, only a small amount of the total frame needs to be read before the switch starts forwarding the frame. Here, no check is done to find out whether the frame is damaged or not but instead has the advantage of forwarding the package a lot faster.

There is an internal processing performed in switches, known as the switch fabric delay [28]. This value differs depending on the fabric of the switch, but is usually only in the order of some microseconds.

Queuing delay is a type of delay which occurs when multiple frames come to a switch in a conjunction. The frames are queued and need to be handled one at a time. The queuing delay is a non-deterministic delay because of the difficulty to exactly predict how traffic flows on a network. An easy estimation of the queuing delay would be its average, which has the following formula:

$$D_{Queuing} = Network\,Load\,(\%) \cdot D_{Transmission(max)}$$

where network load is the percentage relative full network capacity and $D_{Transmission(max)}$ corresponds to the transmission delay of a full-size frame, 1500 bytes. For example, a network with a load of 25 percent would have an average queuing delay of:

$$D_{Queuing} = 0.25 \cdot \tfrac{1500 \cdot 8}{100 \cdot 10^6} = 30\,\mu s.$$

Observe that if there is no load on the network this delay becomes zero.

2.4.3 *Wireline Delay*

Data being transmitted along a wire such as a fiber optic link travel at approximately $\frac{2}{3}$ of the speed of light. At long distance this wireline delay can have a significant impact on the total time. For example, a 100 km long cable would result in a delay of:

$$D_{Wireline} = \tfrac{1 \cdot 10^5}{0.67 \cdot 3 \cdot 10^8} \approx 500\,\mu s$$

Part III

METHOD

3 METHOD

3.1 METHOD

Four different case-studies have been carried out in order to evaluate HLA. In the first three the goal was to be able to send data between two federates, with respect to certain criteria. How the federates are created differs from case to case. However, some parts are equal and independent of how the federates are implemented. These parts have been described in sections below. The fourth case-study has been carried out with focus on when there is more than two federates in a federation.

3.1.1 *Federation Object Model (FOM) Files*

A FOM file is needed in order to create an HLA federation. Exchanging data between federates could be done with either objects with attributes or interactions with parameters, as described in Section 2.2.2 on page 14. With pRTI there is no difference in performance between objects and interactions. What differs is rather the way of modelling the communication. Since the federates in the case-studies are supposed to send data instantly between each other, an interaction class with multiple parameters was used.

A simple FOM file was built with Pitch's Visual OMT software. By performing a couple of simple steps, an interaction class was created containing one parameter with a specified data type. One also need to specify which transport protocol to use. UDP, or *HLAbestEffort*, is an unreliable protocol which does not re-send any data packages that might get lost during the transportation through a LAN. Unlike UDP, TCP (*HLAreliable*) is a reliable protocol which re-sends data if it gets lost. Since the data to be sent is updated very frequently, it would be useless to re-send lost data packages cause the data would be outdated by then. Hence, *HLAbestEffort* was chosen, which should also perform slightly better in terms of delays. The FOM was thereafter saved into an XML file.

A Java program was then created, which was able to modify the created XML file. The Java program has the possibility to generate more parameters, with names and data types. The parameter names would ideally be matched to the names of the actual data to be sent. This was however considered to be out of the scope of this thesis. Instead, the FOM file was filled with a number of "dummy" parameters with increasing numbers in their names, all with the same data type. Using the Java program, a couple of different FOM files were generated and used in the case-studies. An example of a generated FOM file with 10 parameters can be seen in Appendix A.1 on page 81.

3.1.2 *pRTI Settings*

In pRTI one can specify several different settings, making the communication between federates behave in a way that is wanted. The used settings are the same for all implemented prototypes. The LRC has been configured to run with "minimal latency". This means that the data packages are not being buffered into larger packages. Instead, they are sent directly in individual packages over Ethernet. Regarding the CRC, all standard settings have been used.

3.1.3 *Computer Setup*

All prototyped federates, including the CRC, has executed on one or two equal computers (depending on which case-study) with the following specifications:

- Operative system: Microsoft Windows XP Professional (with Service Pack 3)
- Processor: Intel Core 2 6600 (2.4 GHz)
- RAM (Random Access Memory): 2 GB

3.1.4 *The Federates' Step-By-Step Execution*

An overview of what is happening in the two federates during execution can be seen in Figure 2. Below follows a general description of what happens in each step.

Step 1: The federates connect to the CRC part of the pRTI. To connect, the federates must specify the IP-address and the port number of the computer that the CRC runs on. Federates can chose between two different connection types; *HLA_IMMEDIATE* and *HLA_EVOKED.* Using *HLA_IMMEDIATE* makes the LRC part of the pRTI automatically generate callbacks. These callbacks are executed by the federate in parallel with its other tasks, in a so called multi threaded process model. The opposite of this is the *HLA_EVOKED* connection. Here, the federate needs to manually ask the LRC if there are any new callbacks waiting to be executed. If new callbacks exist, they will thereafter be executed in a single threaded (*evoked*) process model. *HLA_IMMEDIATE* is recommended to use and will, because of its multi threaded process model, minimize delays.

Step 2: During this step, federate 1 creates a federation with a chosen name and with a specified FOM file. The FDD parts of the FOM file is used by the pRTI and contains, in this case, an interaction class with names of all the parameters that will be sent between the two federates. It also specifies that the parameters will be sent using *HLAbestEffort* (the UDP protocol). More precisely which FOM file that is specified depends on which test that should run. For example, let's say that 50 parameters of data type 64 bits integers are to be sent between the two federates. Then, a file containing 50 parameter names (with increasing numbers) of the data type *HLAinteger64BE* would be selected (generated by the Java program mentioned in Section 3.1.1). When the federation has been created, both federates are able to join it. When doing this, they also specify their names, for example *Federate1* and *Federate2* respectively.

Step 3: Both federates perform a function call to get the handle to the interaction class containing all the parameters. Next, the federates use their respective handles to make iterative calls to obtain handles to all of the parameters.

Step 4: The federates subscribes and publishes to the interaction class containing all the parameters. Publishing to an interaction class activates the possibility for a federate to send interactions within that interaction class. If an interaction is sent, all federates that have subscribed to that same class will receive that interaction (with all the parameters). Since federate 1 and 2 both should be able to send and receive interactions, they both subscribe and publish to the same interaction class.

Step 5: Here, an iterative type of RTT test is being performed, as briefly described in Section 2.3. Federate 1 starts an accurate timer to measure elapsed time and then assigns all parameters with test values. Next, the parameters are encoded into an HLA compliant bytesarray. Thereafter, federate 1 makes a function call to send them as an interaction. The parameters will at that moment be sent from federate 1 to federate 2, with help of the pRTI. If the federates are running on separate computers, the data will be transferred over Ethernet (using the UDP protocol in this case). Federate 2 will receive a callback from the RTI (automatically or manually, depending on the processing model), triggering a function named *receiveInteraction*. Federate 2 will then execute the code inside *receiveInteraction*. What will happen is that all received parameters will get mapped to their respective handles and thereafter become decoded (one by one). Then, they are all being encoded once again and sent as a new interaction, back to federate 1. Federate 1 gets a *receiveInteraction* callback, then maps and decodes all parameters. After that, the timer is stopped and the procedure is thereafter repeated from the beginning of *step 5*.

Step 6: When enough iterations have been performed, it is time to stop sending and receiving interactions. In most of the tests, at least 1000 iterations have been executed.

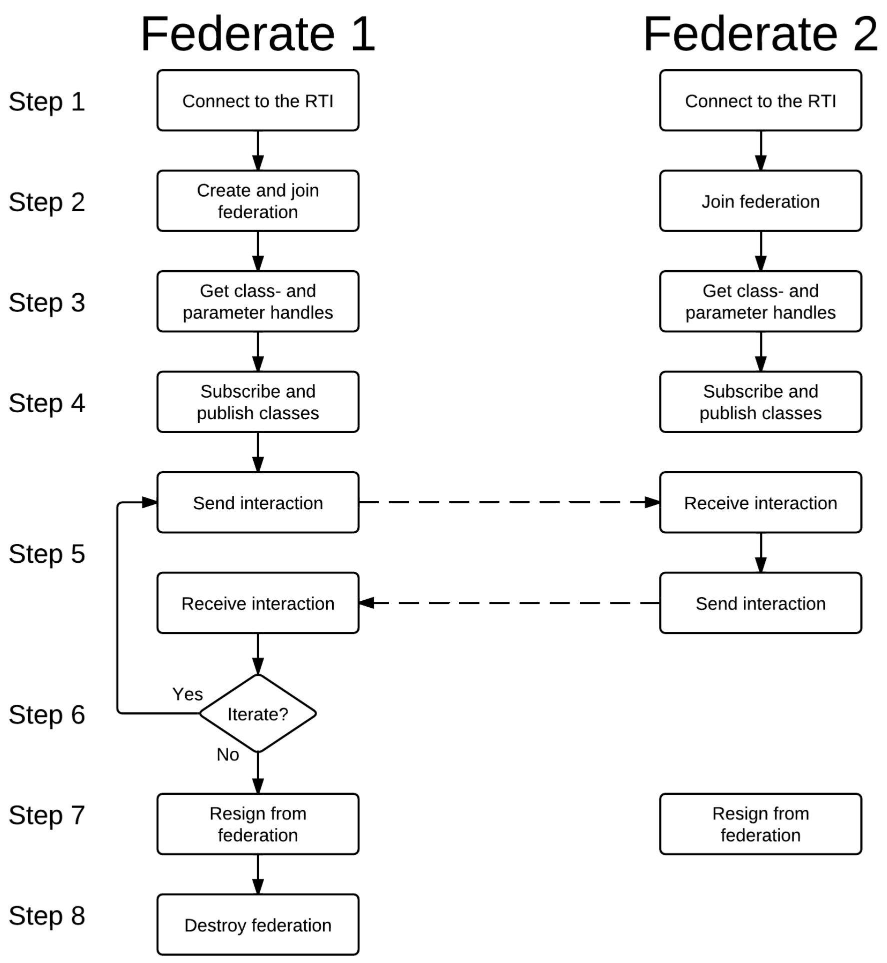

Figure 2: An overview of the actions that two federates perform.

Step 7: When the iterations have stopped, the federates will resign from the federation.

Step 8: After both federates have resigned, federate 1 makes a function call to destroy the federation.

3.2 CRITERIA

Below follow a number of criteria which have been considered when evaluating the built prototypes from the different case-studies.

3.2.1 *Delay Requirements*

Low delays are highly significant when working with these types of simulations. Data that should be shared between federates usually need to be updated very often. To have a relevant requirement to work with, the goal has been set to be able to update all data with 10 ms periods, with a maximum delay of 10 ms. The delay here corresponds to the time it takes for the data to get encoded and sent through HLA, travel from its source to its destination and then get received and decoded again.

3.2.2 *Amount of Shared Data*

The amount of shared data to be sent between federates is of course of high relevance and goes hand-in-hand with the delay requirements. A reasonable amount of data to be shared could be in the order of 2000 parameters. Each of these parameters could be everything from a few bits up to plenty, but an estimated average would be around 5 bits. This corresponds to a data amount of $2000 \cdot 5 = 10\,000$ bits. If the signals were to be packed in 64 bits blocks, the number of blocks would then be $\frac{10\,000}{64} \approx 150$. Therefore, 150 64 bits parameters is considered as an appropriate amount of data to be shared.

3.2.3 *Distance Between Federates*

The distance between federates also play an important role in terms of delays. Since HLA uses the Ethernet protocol it is possible to have federates located very far away from each other. However, the distance comes with increased delays, dependent on for example cable lengths and the number of Ethernet switches for the data to pass. One should at least be able to send the data through one switch, and make estimates on how further switches and cables affect the results.

3.2.4 *Complexity*

The amount of work needed to set up a working HLA simulation system is of importance in whether it is worth to implement or not. Considering an already set up system, it is also important to know the required level of maintenance, and the amount of knowledge needed to work with HLA.

3.2.5 *Scalability*

How HLA performs in terms of scalability would be of significance. For example, how a growing number of federates or an increased data exchange is handled.

3.2.6 *Computational Power*

It would be of interest to investigate how the computational power of the PCs in the prototypes affect the results. In some case-studies it is also of importance to not use all of the respective computer's computational capacity for handling data transfer. This must be taken into consideration during the evaluations.

3.3 DEVELOPMENT TOOLS

The following development tools have been used throughout this thesis.

- Pitch's pRTI v.4.4.0.0, providing the LRC and CRC parts of the RTI during all federation executions.
- MathWorks MATLAB R2011a and Simulink. MATLAB was used together with the MLIB library (v.4.7.2) for interactions with a dSPACE board. It was also used during measurement plots and together with ForwardSim's HLA Toolbox (v.3.0.0.539) when developing MATLAB federates. Simulink was used to develop Simulink federates together with ForwardSim's HLA Blockset (v.2.1.0.247). Simulink was also used when developing and building dSPACE applications.
- Microsoft Visual C++ 2010 Express, for developing C++ federates using Pitch's C++ API libraries. It was also used for developing C code when using the CLIB library (v.4.x) for interactions with a dSPACE board.
- Pitch's Visual OMT v.2.2.1, used to create simple FOM files.
- Eclipse IDE for Java Developers version Indigo (Service Release 1), together with JDOM (v1.1.3) to modify and extend XML formatted files (FOM files).
- dSPACE's ControlDesk Developer Version 3.7.2, for interaction with dSPACE boards such as loading of applications.

Part IV

CASE-STUDIES

4 PROTOTYPING OF HIL-SIMULATOR FEDERATES

4.1 DESCRIPTION OF PROTOTYPE

In this set-up, the goal was to send data from one model running on a dSPACE HIL-simulator, through HLA, to another model on a dSPACE HIL-simulator located somewhere else. Unfortunately, there exists no ready-made gateway for communication between a real-time dSPACE simulator and HLA. Instead, the data to be exchanged from one dSPACE simulator to another need to go through host computers that the respective dSPACE simulator is hooked up to. The host computer then has the possibility to act as a gateway between dSPACE and HLA. The intention is that the host computers should be dedicated to these tasks. Thus, it should be permitted to use all available computational power of those computers. The set-up would preferably look like Figure 3.

The implementation of this prototype has been divided into smaller parts. First, the communication between dSPACE simulator and host computer was implemented (Section 4.2.2). Then, the HLA communication between the two host computers was made (Section 4.2.3). Finally, all pieces were put together (Section 4.2.4).

4.2 IMPLEMENTATION OF PROTOTYPE

Since only one dSPACE simulator was available, the actual set-up was instead modified to look like Figure 4. The difference between this set-up and the preferred one in Figure 3 is that the data travels with HLA over the LAN twice instead of once. This can however be compensated for, by performing an RTT test between federate 1 and federate 2 (using the same amount of data), as described in Section 2.3. Since

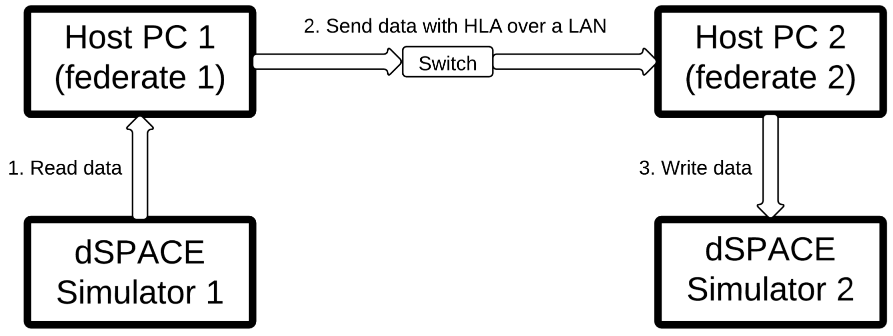

Figure 3: Preferred set-up of the dSPACE HIL-simulators prototype.

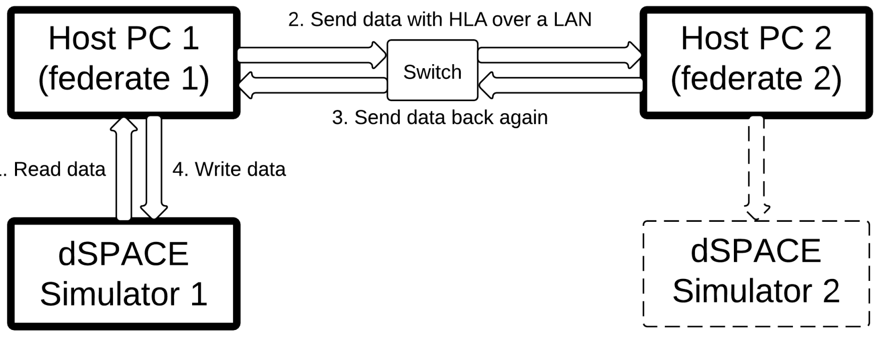

Figure 4: Actual set-up of the dSPACE HIL-simulators prototype.

host PC 1 and 2 have similar hardware and software, the RTT result can be divided by two to represent a one-way delay. This delay could then be subtracted from the total time measurement, to simulate the complete data path.

4.2.1 *Simulator Application*

First of all, in order to have some data to perform read and write operations on, an application must be running on the dSPACE simulator. Thus, a simple Simulink model was created, containing a couple of thousand constant blocks of both 8 bits and 64 bits parameters. It also includes a Real-Time Interface (RTI) block [29], acting as a link between Simulink and dSPACE hardware (not to be confused with the HLA RTI). The Simulink model was then built to a dSPACE application and uploaded to a DS1006 Processor Board [30], located in the dSPACE simulator. During the build process another important file was created, a TRC. That file contains information about the signals and parameters of the application, such as data types and address references.

4.2.2 *Communication Between dSPACE Simulator and Host Computer*

The host computer was connected to the dSPACE simulator via a bus interface, using a fibre-optic cable. This cable was attached to a dSPACE link board of type DS817 installed in the host computer, and a dSPACE link board of type DS814 on the simulator side.

dSPACE provides three different ways for transferring parameters between the host computer and the application running on the dSPACE board. The alternatives are through ControlDesk, MLIB and CLIB [31]. ControlDesk has not been evaluated since it is not intended for time-critical transfer of data. MLIB and CLIB have both been tested in order to decide which of them that is most appropriate for this prototype.

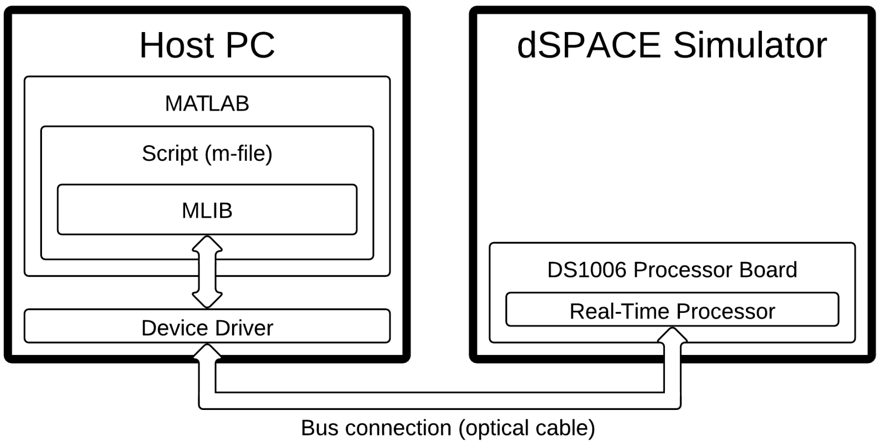

Figure 5: The communication between the host computer and the dSPACE simulator, using MLIB.

4.2.2.1 *MLIB*

With help of MLIB and ForwardSim's HLA Toolbox software, it is possible to establish a data communication channel between the dSPACE simulator and HLA. That is the reason why test code has been written for MLIB and is described below. Figure 5 shows an illustration over the communication.

The code has been written in a regular MATLAB script file (m-file). First, a processor board has to be selected. This is done by specifying the wanted board, in this case a DS1006, to the function *SelectBoard*. Then, the function *SearchTRC* is used with an input of an "expression". MLIB searches the TRC file for any parameters or signals matching the expression and generates a vector containing those matches. All the constant blocks in the created Simulink model are located directly under the model root. Hence, using the *SearchTRC* function with "Model Root/Constant*" as the expression results in a vector containing the search paths to all of the constant blocks from the model. Next, in order to read and write to these parameters, MLIB needs to know things such as address locations and data types. This is done by using the vector, containing the search paths, in the function *GetTrcVar*. The output is a vector with all information needed for MLIB to be able to read and write data to all the constant blocks. Reading the data is done by simply providing the output vector (from *GetTrcVar*) to the function *Read* and a vector with all read data is returned. The function *Write* works in a similar way, except you also have to provide the data that should be written to the parameters. Elapsed time for reading and writing operations have been measured using MATLAB's function TIC/TOC.

Results of how MLIB performs in terms of read and write delays are presented in Section 4.3.1 on page 36.

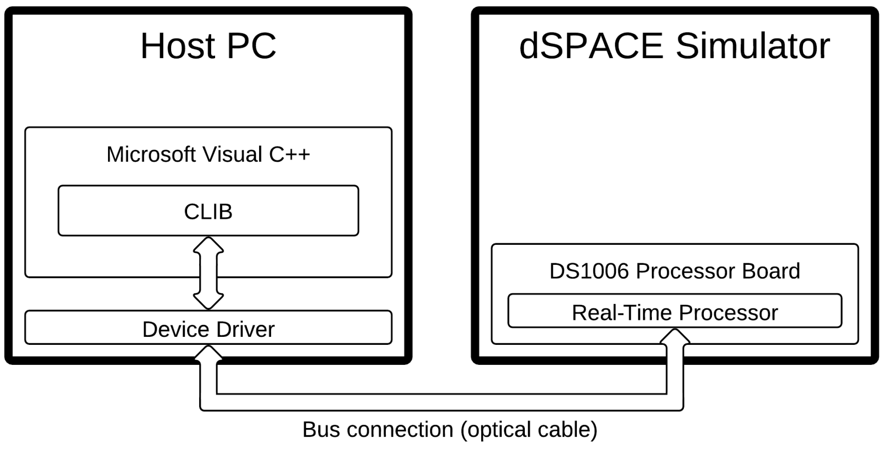

Figure 6: The communication between the host computer and the dSPACE simulator, using CLIB.

4.2.2.2 *CLIB*

The pRTI has a built-in API for C++. It is possible to combine C and C++ code in the same program. Thus, one can with CLIB read and write data and connect this to HLA. The CLIB code is described in brief below. Figure 6 shows an illustration over the communication.

The CLIB implementation has been written in regular C code. To be able to use the CLIB functions, the CLIB library was added to the project.

First, in order to get access to the DS1006 processor board, a program has to register itself to the DSP Device Driver. This is done with the function *DS_register_host_app*, which takes a name (by your choise) as input. Next, a call to *DS_board_index* is made, with the board name *DS1006* as input. The board index is returned, and is used in most function calls below in order to say which board the specific function should apply to. After that, a check is made to see that there is an application running on the board. That is done by calling *DS_is_reset*, which returns the state of whether the RTP is reset or running.

Since CLIB does not have any ready-made functions for parsing information from the TRC file, this is done in a more manual way. The TRC file in this case contains a couple of thousand parameters (all the constant blocks from the Simulink model). The parameters are sorted in an alphabetical order and are located in the processor board's memory in the same order. By providing the function *DS_get_var_addr* with the first parameter's *address name* from the TRC file, in this case "p_ManyConstantBlocks_P_real_T_0" (where "ManyConstantBlocks" comes from the name of the Simulink model), a 32 bit pointer to the memory address of that parameter is returned. By reading this 32 bit pointer address with the function *DS_read_32*, the actual memory address is returned, of where the first parameter is stored in the board's memory. This address can now be used as a reference address when performing reading and writing operations on the dSPACE board. Parameters of 64 bits data type can be read using *DS_read_64*. This read-function needs to know the reference address (from above) and how many parameters that should be read. The read values are then returned in an UInt64 vector. *DS_write_64* works in a similar way, but needs of course an UInt64 vector with values to be written. For reading and writing the 8 bits parameters, similar steps are performed, but with the functions *DS_read_8* and *DS_write_8* for reading respective writing data. When the C code is about to exit, the host application should unregister from the DSP Device Driver by calling the function *DS_unregister_host_app*.

Results of how CLIB performs in terms of read and write delays are presented in Section 4.3.1 on page 36.

4.2.3 *Communication Between the Two Host Computers*

Two HLA federates have been implemented for sending data between the two host computers. The host computers were connected to each other with a bandwidth of 100 Mbit/s over a LAN, with one Ethernet switch in between (using the cut-through technique). The results in Section 4.3.1 show that the best way, in terms of delays, is to use CLIB in this prototype. It is also the most proper solution in terms of money, since using MLIB requires licenses for both MATLAB and ForwardSim's HLA Toolbox. Hence, the two host computer federates were implemented in C++.

To perform one step at a time, the HLA federates were first implemented without involving CLIB and the dSPACE simulator.

4.2.3.1 *The Code of the Federates*

The code of the federates have been written in C++ together with necessary LRC libraries from Pitch.

An overview of what is happening during the code execution of the federates can be seen in Figure 7. Below follows a description of what happens in each step.

Step 1: The federates connect to the CRC part of the pRTI, located on the same computer as federate 1. Both federates connect using the setting *HLA_IMMEDIATE*, which makes the LRCs execute new callbacks automatically.

Step 2: During this step, federate 1 creates a federation named *MyTestFederation* and specifies a FOM file. When the federation has been created both federates join, with the names *Federate1* and *Federate2* respectively.

Step 3: Both federates perform a function call to get the handle to the interaction class named *DataParameters*. Next, the federates use their respective handles to make iterative calls to obtain handles to all of the parameters inside *DataParameters*.

Step 4: The federates subscribes and publishes to *DataParameters*.

Step 5: Here, *Federate1* begins with calling the C++ function *QueryPerformanceCounter*, used to measure elapsed time. Then, all parameters are being assigned with "dummy" values. Next, *Federate1* encode all parameters and send them as an interaction through its LRC. *Federate2* will automatically receive a callback from its LRC, triggering the function *receiveInteraction*. *Federate2* will receive all parameters and map them to their respective handles and thereafter decode them. After that, all values are slightly modified (increased by 1, for test purposes). Then, they are all being encoded once again and sent as a new interaction back to *Federate1*. Now, *Federate1* gets a *receiveInteraction* callback, maps and decodes all parameters and then again calls the function *QueryPerformanceCounter,* to be able to calculate how long time that has elapsed. The procedure is then repeated from the beginning of *step 5*.

Step 6: When 1000 iterations have been performed, sending and receiving interactions stop.

Step 7: The federates resign from the federation.

Step 8: When both federates have resigned, *Federate1* makes a function call to destroy the federation.

4.2.4 *The Complete Federates' Code*

In the complete federates' code, the scenario visualized in Figure 4 on page 30 has been realized. The CLIB code written in C (from Section 4.2.2.2) has been combined with the C++ code of federate 1 (from Section 4.2.3.1). The code of federate 2 has remained unchanged.

The final product is very similar to the steps performed in Figure 7. The major differences occur around *step 5*. Before *Federate1* starts performing *step 5*, CLIB must be initialized and ready to perform read and write operations, as described in Section 4.2.2.2. When this has been done, *step 5* is performed with a few modifications. Instead of initializing the parameters with "dummy" values, CLIB reads the corresponding parameters from the dSPACE board's memory and ports those data into the HLA parameters.

Then, *step 5* continues as regular until *Federate1* is about to call *QueryPerformanceCounter* the second time. Before doing that, the HLA parameters are ported back to proper data which CLIB writes to the dSPACE board's memory. When that has finished, the call to *QueryPerformanceCounter* is performed and the elapsed time is calculated. This modified version of *step 5* will then continue to iterate. When the iteration part has finished and *step 7* is about to run, CLIB should first unregister from the dSPACE board. The rest of the steps remain the same.

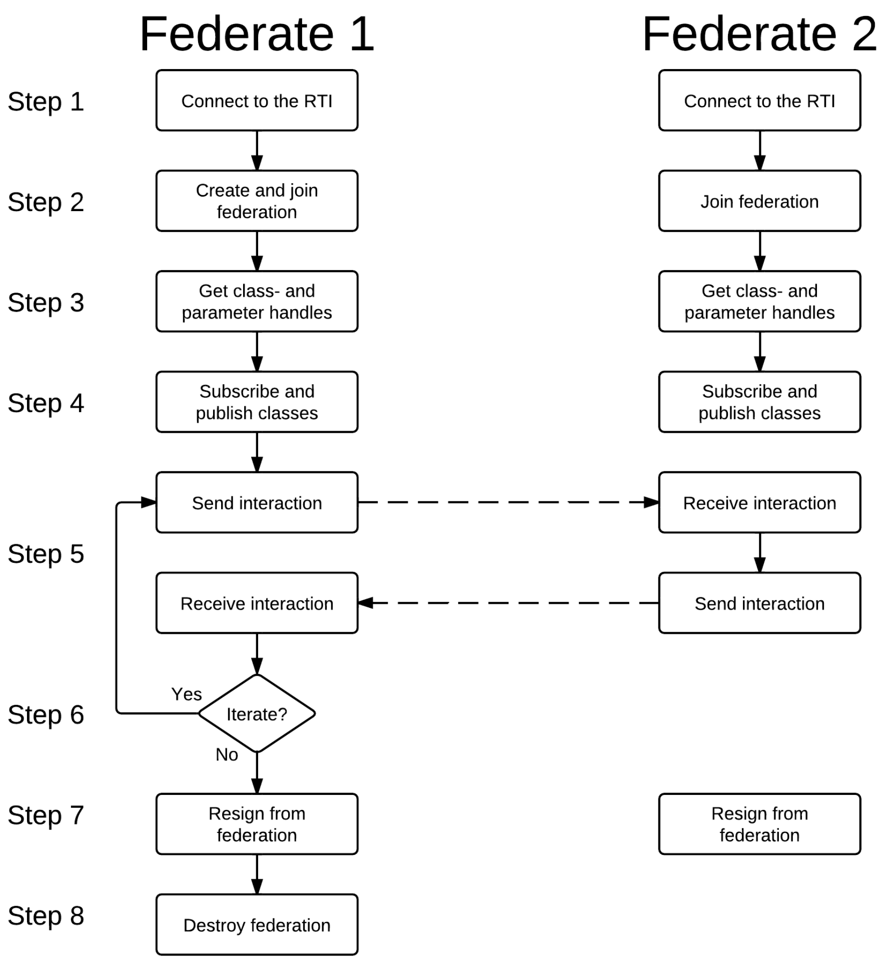

Figure 7: An overview of the actions that the two C++ federates perform.

4.3 RESULTS

4.3.1 *CLIB and MLIB - Reading and Writing of Data*

Several plots have been presented below in order to show the results regarding CLIB and MLIB.

In Figure 8 and Figure 9, one single 64 bits parameter has been read 1000 times iteratively from the memory of the DS1006 processor board, using CLIB and MLIB respectively. Observe that these two figures have been plotted with the same scale on the axes, for easier comparison. According to the figures, in most of the iterations the reading takes less than 1 ms. An interesting fact is that MLIB takes almost the double amount of time to finish all 1000 readings. Another thing that happens in both figures is that some readings take slightly more than 10 ms to execute. This is repeated with a relatively constant period. A closer look on the behaviour shows that these peaks occur around every 46-47 ms, and at that time adds an additional 10 ms to the execution time. Unfortunately, the cause of this behaviour is unknown.

To better show how long time reading values usually take, the peaks have been removed in some plots. Figure 10 and Figure 11 show what iteratively reading one 64 bits parameter looks like, without such peaks. What can be seen from those figures is that CLIB performs almost twice as good as MLIB, when reading a single parameter.

The next two figures present results of reading and writing multiple parameters. Figure 12 shows results of CLIB and Figure 13 of MLIB. Here, there is a huge difference between the results of the two figures. For example, reading 100 values with CLIB takes ~0.85 ms compared to MLIB's result of ~40 ms. The reason for this is that CLIB has the possibility to read (and write) *a block* with parameters located in consecutive memory order. What MLIB does here is that it reads each parameter as if they were located anywhere in the memory. If CLIB should have read 100 values all from spread memory locations, it would instead have taken approximately $0.35 \cdot 100 = 35$ ms.

What also can be obtained from Figure 12 is that reading takes less time than writing (when having more than a few parameters). Reading and writing additional values continue in a very linear way with approximately 5.5 ms and 8.4 ms respectively, for 1000 parameters.

In the rest of the figures below, only CLIB has been plotted. Figure 14 shows the same type of test as Figure 12, but with 8 bits parameters instead of 64 bits. Note that the scale of the y-axes are different in the mentioned figures. The 8 bits plot should ideally be 8 times faster than the 64 bits one. If one compare the figures, the result does not become a factor 8 in difference. This, however, is because of the (unavoidable) "start time", or *overhead*, of 0.35 ms which

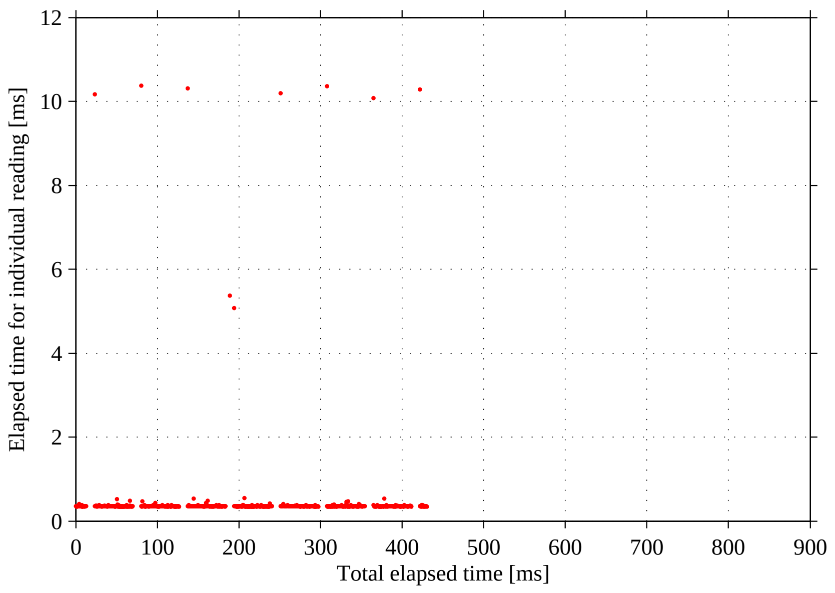

Figure 8: Using CLIB, one 64 bits parameter has been read iteratively 1000 times. Most of the readings take less than one millisecond but a couple of them peaks and goes above 10 ms. The axes of the figure are the same as in Figure 9, for easier comparison between the plots.

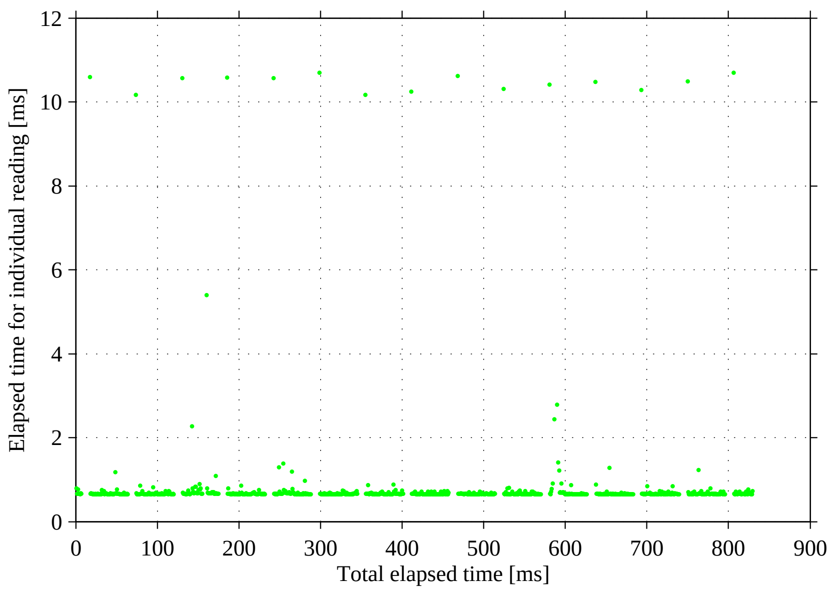

Figure 9: Using MLIB, one 64 bits parameter has been read iteratively 1000 times. Most of the readings take less than one millisecond but a couple of them peaks and goes above 10 ms. The axes of the figure are the same as in Figure 8, for easier comparison between the plots.

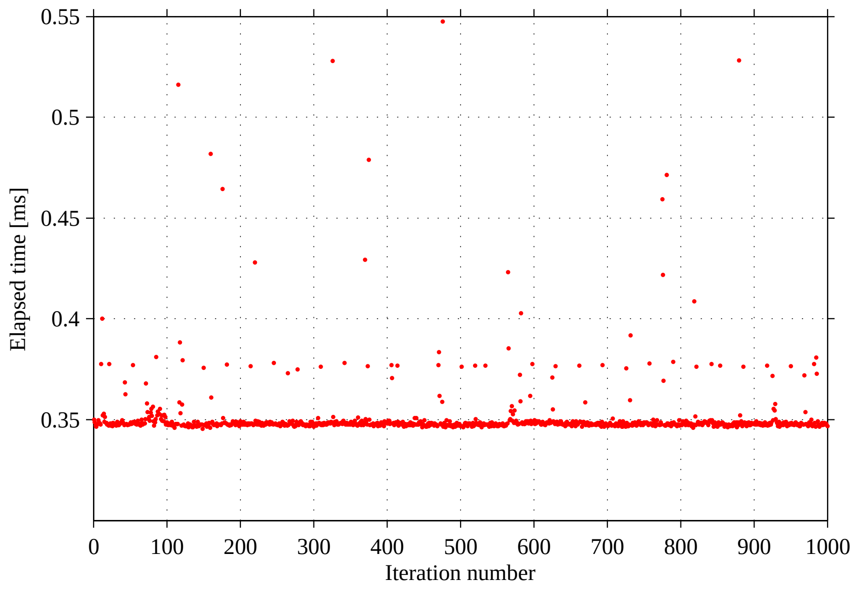

Figure 10: Using CLIB one 64 bits parameter has been read iteratively 1000 times. Here, peak values have been removed. The mean value for this plot is approximately 0.35 ms.

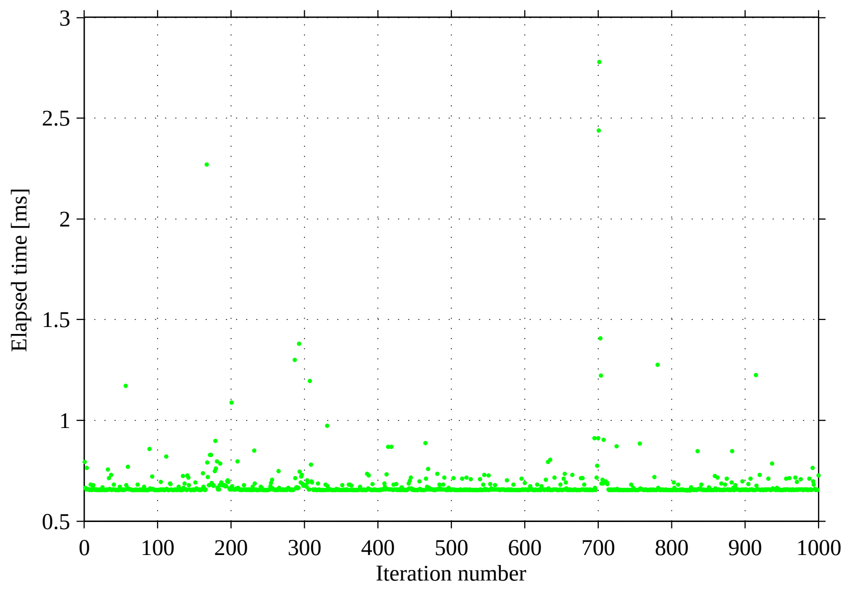

Figure 11: Using MLIB one 64 bits parameter has been read iteratively 1000 times. Here, peak values have been removed. The mean value for this plot is approximately 0.68 ms.

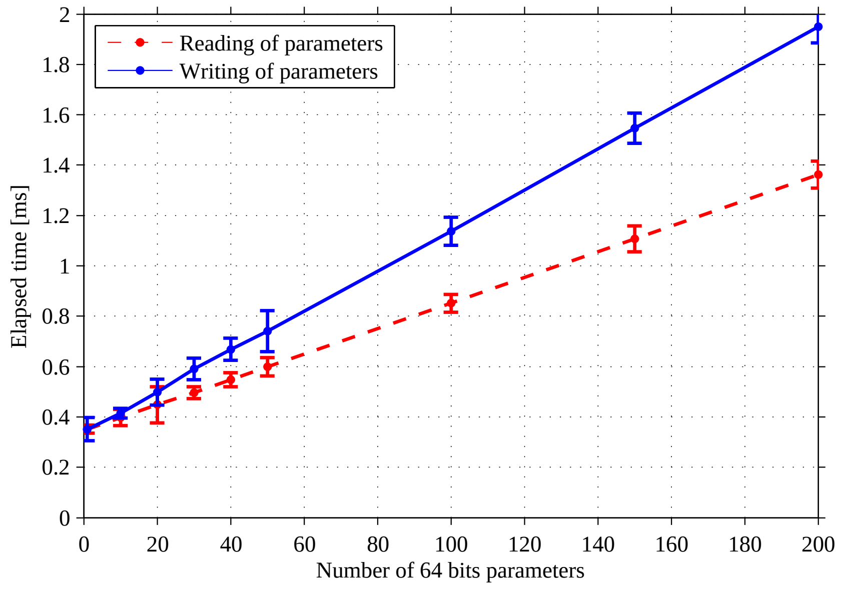

Figure 12: Using CLIB, multiple 64 bits parameters have been read (the red-dashed line) and written (the blue line). All parameters are read (respectively written) in consecutive memory address order. Each measure point in the figure corresponds to a mean value of 1000 iterations together with the standard deviation. Peak values have been removed.

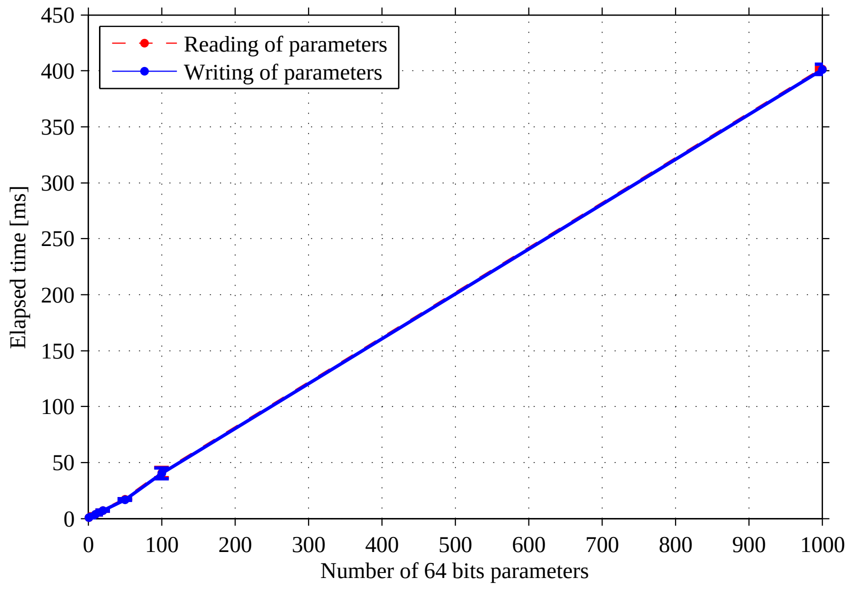

Figure 13: Using MLIB, multiple 64 bits parameters have been read and written, respectively. The two lines are almost identical, which is the reason why only one seems to appear. Each measure point in the figure corresponds to a mean value of 1000 iterations together with the standard deviation. Peak values have been removed.

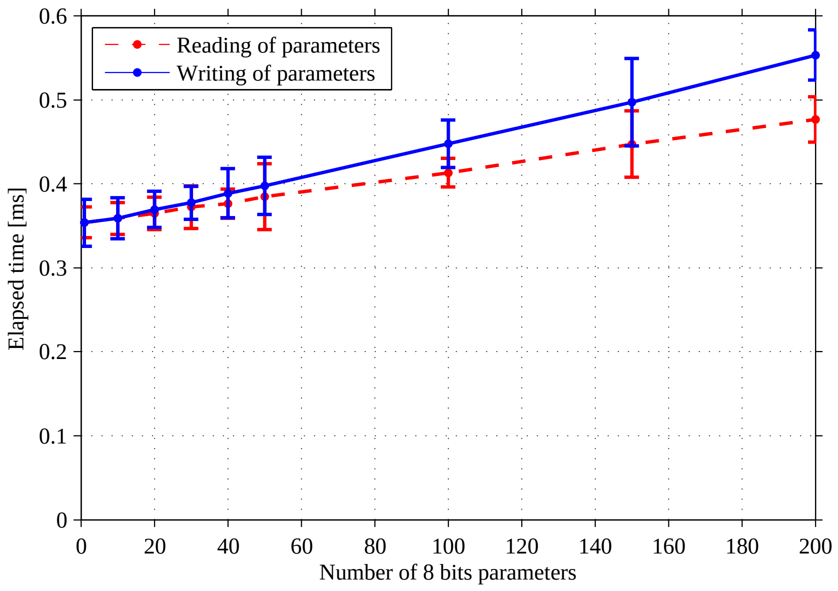

Figure 14: Using CLIB, multiple 8 bits parameters have been read (the red-dashed line) and written (the blue line). All parameters are read (respectively written) in consecutive memory address order. Each measure point in the figure corresponds to a mean value of 1000 iterations together with the standard deviation. Peak values have been removed.

stands for a large part of the elapsed time in Figure 14. If that time is removed from both figures, and they are compared, the difference is almost exactly a factor 8. For example, comparing the reading of 200 parameters gives the following: $\frac{1.36-0.35}{0.48-0.35} \approx 7.8$. When the number of parameters increase, the result is even closer to a factor 8. What this concludes is that the most significant in terms of execution time is the total amount of data to be read or written, not the data size of each individual parameter.

The Simulink model that has been compiled into an application and uploaded to the dSPACE board was very simple. The load of the RTP was therefore very low. Thus, a more demanding Simulink model was compiled and uploaded to the board, resulting in an RTP load around 50%. Figure 15 shows a plot of this, where 150 64 bits values have been read iteratively 1000 times. Figure 16 shows the same results obtained when the RTP load is low. By comparing these figures one see that in Figure 15 the values fluctuate quite much. The mean value is 1.21 ms, which compared to the mean value of Figure 16, of 1.11 ms, is not too bad. The elapsed time is only increased by in average a factor $\frac{1.21}{1.09} \approx 1.09$.

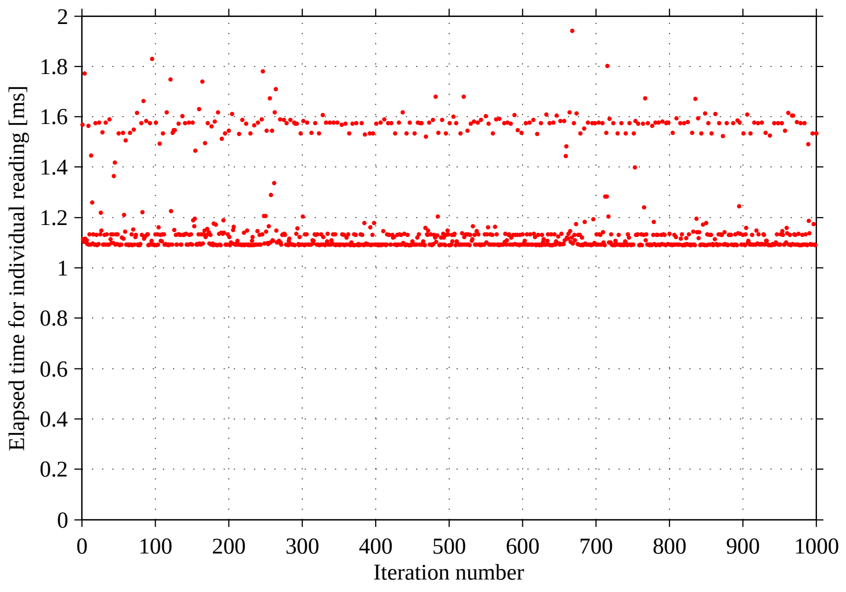

Figure 15: Using CLIB, 150 64 bits parameters have been read iteratively 1000 times. The application running on the processor board makes the RTP work on a level of around 50% load. Peaks have been removed.

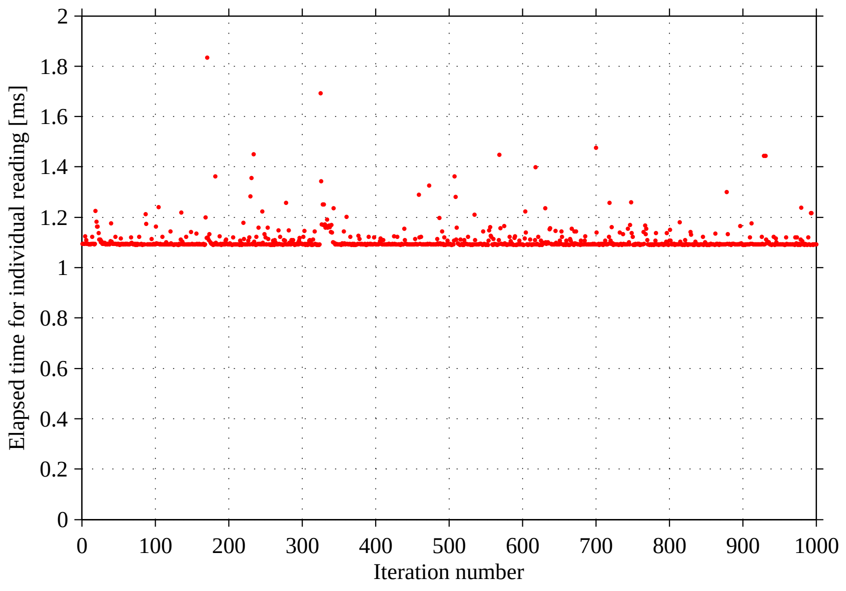

Figure 16: Using CLIB, 150 64 bits parameters have been read iteratively 1000 times. The application running on the processor board is not very demanding. Peak values have been removed.

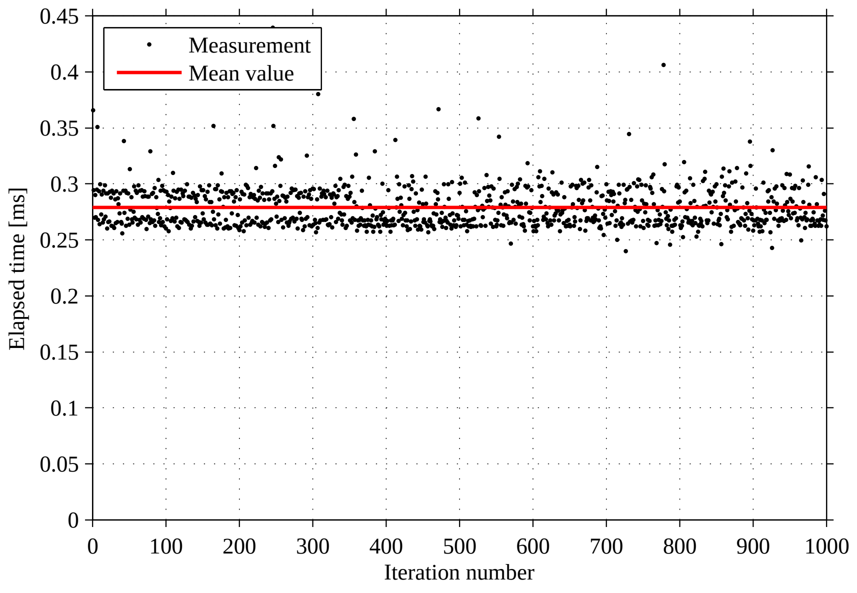

Figure 17: A single 64 bits parameter has been sent iteratively 1000 times from Federate 1 to Federate 2, using HLA. The mean value is approximately 0.28 ms.

4.3.2 *Data Exchange Between the Host Computers*

In Figure 17 and 18 the time results of sending one respective multiple 64 bits parameters from Federate 1 (on host computer 1) to Federate 2 (on host computer 2) are presented. The results originate from RTT tests, where all measurement values have been divided by two to present the corresponding one-way delays. Figure 17 tells that encoding a single parameter, sending it as an Ethernet frame on the 100 Mbit/s LAN where it passes a switch and then reaches Federate 2 where it becomes decoded again, usually takes around 0.25-0.30 ms.

As one can see in Figure 18 where multiple parameters have been sent, the difference between sending 64 bits parameters and 8 bits parameters is not big. What matters the most is the number of parameters being sent. A large part of the total time when sending multiple parameters like this occur at the receiver side. Since the parameters are sent individually, each received parameter needs to be mapped to the same parameter on the receiver side. When introducing a lot of parameters, the loop in the code for doing this increases. The 8 bits parameters do however perform a little faster. This is mainly due to the less time needed to perform encoding and decoding operations on the parameters. A computer with more computational power, such as a quicker processor, would be able to handle even more parameters during the same amount of elapsed time.

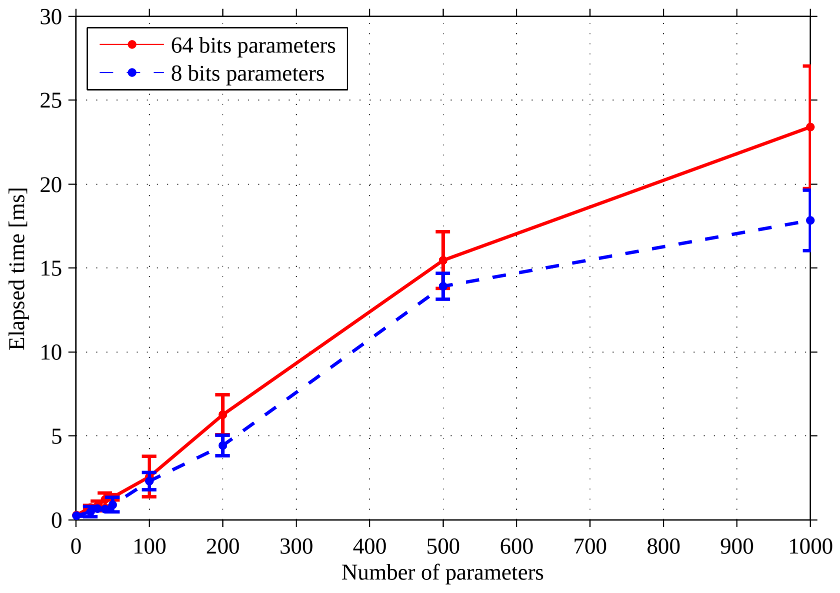

Figure 18: Multiple parameters with "dummy" values have been sent from Federate 1 to Federate 2 using HLA. The red line shows the result when transferring 64 bits parameters and the blue-dashed line the result when transferring 8 bits parameters. Each measure point in the figure corresponds to a mean value of 1000 iterations together with the standard deviation.

4.3.3 *Simulated Complete Data Path*

In Figure 19 results from the simulated complete data path have been plotted. The plot shows a relatively linear behaviour when the number of parameters increase. A more zoomed-in view of the figure can be seen in Figure 20. Observe that the peaks originating from CLIB have been removed. With peaks removed, the data requirement of being able to send 150 64 bits parameters within 10 ms is fulfilled. However, by studying Figure 21 where peaks have not been removed, the result is a bit different. That figure shows the complete data path when sending 150 64 bits parameters iteratively. Here, almost every fifth measurement includes a peak value, corresponding to a mean value around 13 ms instead of the regular value which is slightly less than 6 ms. An average of all 1000 measurements is 7.3 ms.

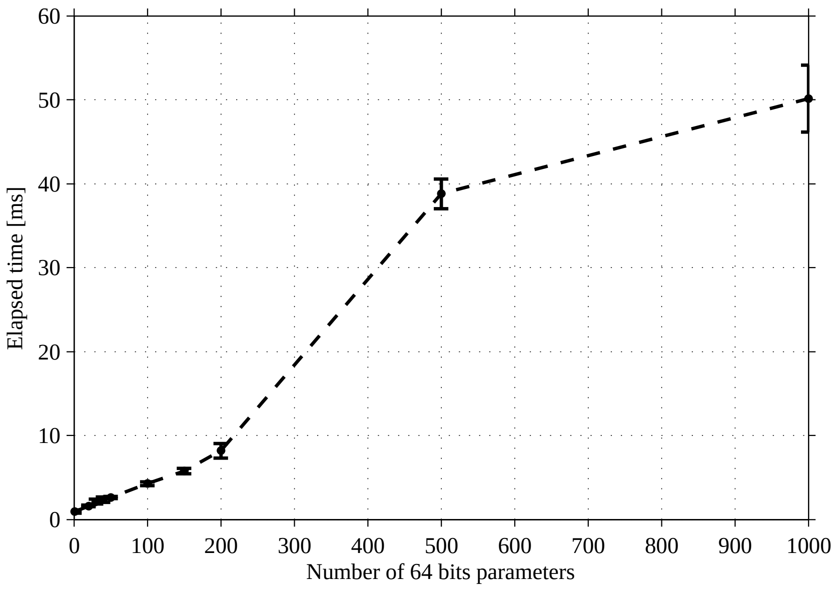

Figure 19: The simulated complete data path, with a data flow as indicated in Figure 3 on page 29. Multiple 64 bits parameters have been used and each measure point corresponds to a mean value of 1000 iterations together with the standard deviation. Observe that peak values have been removed, originating from the problem that CLIB sometimes takes additional time to perform read and write operations.

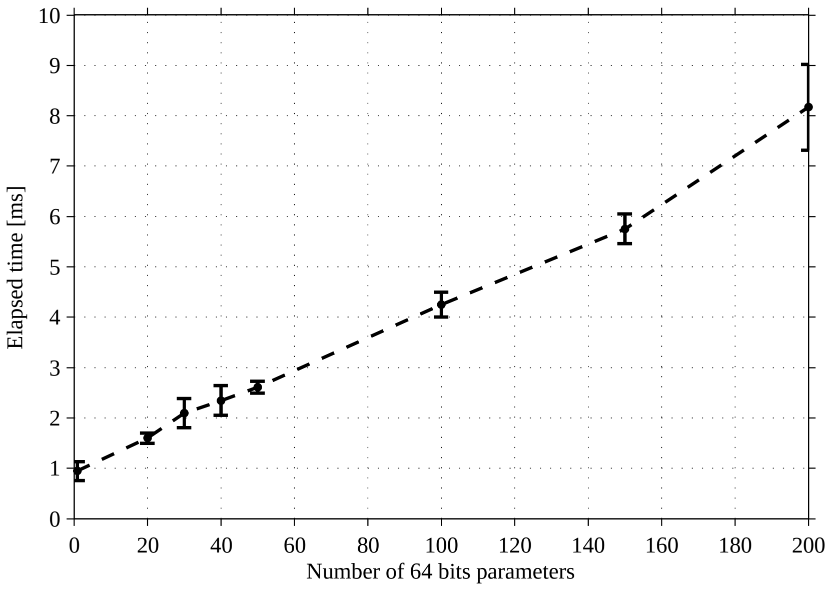

Figure 20: A zoomed-in version of Figure 19, with a maximum of 200 parameters. Observe that peak values have been removed.

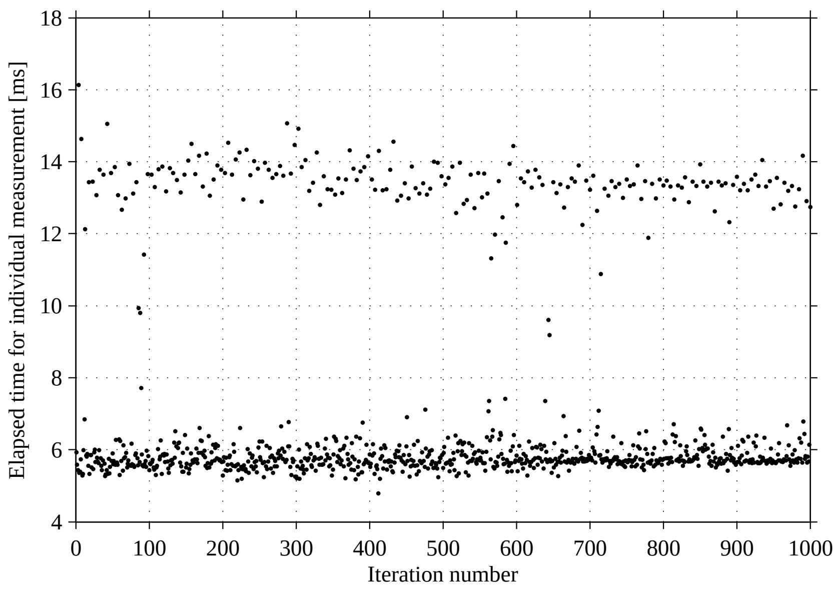

Figure 21: Here, 150 64 bits parameters have been transported the simulated complete data path, with a data flow as indicated in Figure 3 on page 29. The procedure has been iterated 1000 times. Here, the peaks, originating from the problem that CLIB sometimes takes additional time to perform read and write operations, have *not* been removed. The peak values are the dots distributed around 13 ms.

4.4 ANALYSIS

The results of the simulated complete data path look quite promising when compared with the criteria. The delay requirements of 10 ms, in combination with the distance and the amount of shared data (150 64 bits parameters) do not seem unreasonable to fulfill. The problem with the peaks that sometimes appear during data transmission between the host computer and the dSPACE board is however not solved. If those are taken into account, the delay requirements of being able to periodically (every 10th ms) update 150 64 bits parameters (within 10 ms) is during peaks exceeded by a few milliseconds, which can be seen in Figure 21. The problem could be Windows-related in combination with the bus connection to the dSPACE simulator. It could also be something happening on the dSPACE board. In summary, the problem can most likely be avoided in one way or another.

What should not be completely forgotten is that the test model is rather simple, with quite favorable conditions. For example, the test case assumes that data to be read (or written) with CLIB already has been packed in chunks with 64 bits of data. In a real model it is more likely that parameters are stored in smaller data types. If so, some sort of "conversion" would be needed to pack them into 64 bits data types. Another favorable thing with the test-case is that it assumes that all data to be read (or written) with CLIB is located in consecutive memory order. In a more complicated Simulink model, this would probably have to be arranged in some way in order to achieve such structure. If CLIB would need to read (or write) from several different memory locations, additional delays would occur. Each such delay is approximately 0.35 ms, so having all the data stored in consecutive memory order is of high importance considering delays.

The Ethernet part of this case-setup involves two Ethernet cards (one in each host computer), two cables and one Ethernet switch. If one approximate the elapsed time for sending 150 64 bits parameters between the host computers to 6 ms, obtained from Figure 20, there is another 4 ms left (according to the 10 ms criterion) which can be used for transferring the data a longer distance. The data, 150 64 bits parameters, corresponds to a total amount of $\frac{150 \cdot 64}{8} = 1200$ bytes. That amount could be packed and sent in a single Ethernet frame with help of the RTI. Let's assume perfect conditions in the meaning that there is a dedicated optical link available with no other Ethernet frames introducing any additional queuing delays. With such conditions, the longest possible distance depends primarily on the wireline delay. As calculated in Section 2.4 on page 16, the wireline delay is approximately 0.5 ms per 100 km wire. Using those calculations, the maximum possible distance using 4 ms of travel-time would then be 800 km.

The reason why MLIB seems to perform quite bad in the tests is that it is not really intended to be used for such tasks. The purpose with MLIB is usually to analyze real-time data, but not passing it on to somewhere else with low delay, as in this case-study. An example of a use-case with MLIB would be to read the values of some parameters during each time-step the simulator performs, for 1000 steps. What happens in such a case is that the RTP reads the parameters during each time-step and then stores the values in a buffer. Not until all 1000 steps have executed, all data inside the buffer is being sent to the host computer. If each time-step for example is 1 ms, it would take one second just to get all values into the buffer, and some additional transfer time before the data reaches MATLAB on the host computer. Thus, that data would be considered old and useless in this case-study, but useful in other analytical scenarios.

4.5 FUTURE WORK

A couple of suggested future improvements are presented below.

- It would be preferred to have some kind of script which produces a proper FOM file from a TRC file. As the implementation is now, the parameter names in the FOM file are not matched to the parameter names in the TRC file.
- Real data to be transferred using CLIB should preferably be packed into chunks of 64 bits data. Also, one has to make sure that the data is being stored in consecutive memory address order on the dSPACE board's memory. Both these depend on how the Simulink model is designed and also with which settings the dSPACE application is built.
- The peaks that sometimes occur when CLIB reads and writes data would be desirable to investigate, to find out what are causing them and how they are avoided.
- Finally, an improvement would be to run the federates (and the CLIB communication) on a real-time operative system, instead of Windows XP. When running the code on Windows it can get interrupted if the operative system decides to do something else, which could delay the code execution with several milliseconds. With a real-time operative system, such things can be avoided.

5 PROTOTYPING OF MATLAB FEDERATES

5.1 DESCRIPTION OF PROTOTYPE

In this test-case the goal was to evaluate how HLA performs when two MATLAB federates exchange data. To connect MATLAB with HLA, the HLA Toolbox has been used. The set-up would preferably look as in Figure 22.

5.2 IMPLEMENTATION OF PROTOTYPE

Due to license limitations with the HLA Toolbox, the implementation was modified to instead look like Figure 23. As seen in the figure, the two federates will run on the same computer, but in two different MATLAB instances. The PC's processor has two cores and the MATLAB instances will run on one core each. Also, since only one computer is used, the communication does not go with Ethernet over a LAN. However, this has been compensated for by estimating the delays associated with the LAN and taking them into account when performing tests.

As a whole, this prototype is quite similar to what has been performed in Section 4.2.3. However, instead of building the federates in C++ code using Microsoft Visual C++, they have been built in MATLAB as two functions. The HLA Toolbox's RTI library has been added to both MATLAB functions, which makes them HLA federates. The HLA Toolbox's RTI library is very similar to Pitch's C++ RTI library, which was used in the two C++ federates in Chapter 4.

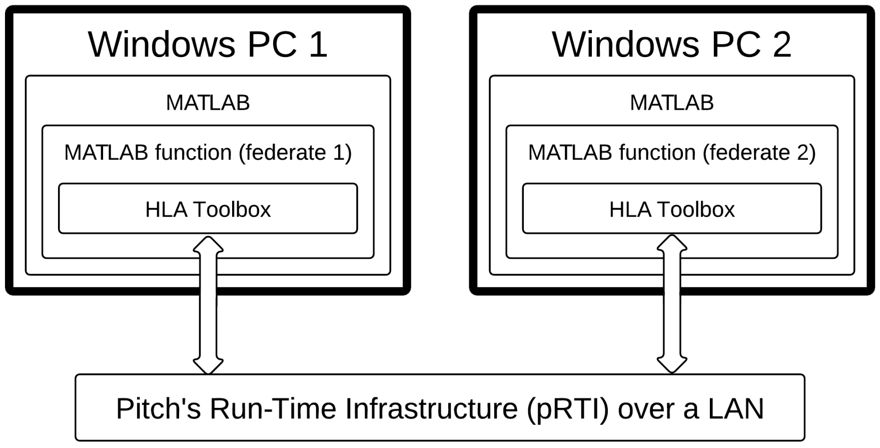
Windows PC 1
MATLAB
MATLAB function (federate 1)
HLA Toolbox
Windows PC 2
MATLAB
MATLAB function (federate 2)
HLA Toolbox
Pitch's Run-Time Infrastructure (pRTI) over a LAN

Figure 22: Preferred set-up of the MATLAB federates prototype.

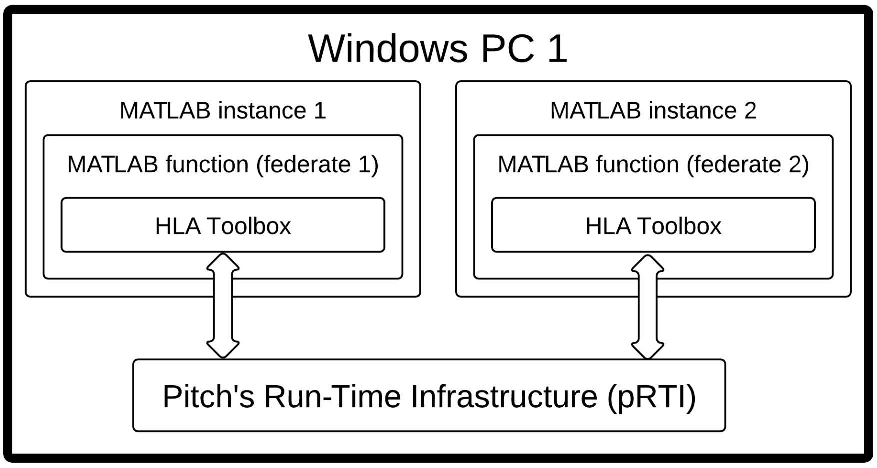
Windows PC 1
MATLAB instance 1
MATLAB function (federate 1)
HLA Toolbox
MATLAB instance 2
MATLAB function (federate 2)
HLA Toolbox
Pitch's Run-Time Infrastructure (pRTI)

Figure 23: Actual set-up of the MATLAB federates prototype.

There is one big difference in this prototype compared to the one in Chapter 4. Here, it is important to only use a certain amount of all the computational power available. The intension of a federate is normally to simulate some kind of model and exchange a certain amount of data with other federates. If all the computational power is used to send and receive data, the rest of the model will not be able to execute. It is very computationally demanding for a federate to handle many incoming parameters during a short period of time. What could be done instead is to send an arbitrarily large array with data inside one parameter, instead of putting the data in separate parameters. An array with data is a lot easier for MATLAB to handle. There are however some drawbacks doing this. If only two federates should form a federation it is relatively easy to decide what data they should send to each other, and put everything in one parameter each. However, if more federates join it is unlikely that they are interested in exactly the same data as the others publish. Let's say a third federate only is interested in a couple of values from the other two federates. Then it has to subscribe to the two large parameter arrays, receive all that data and parse out the parts it is interested in (and throw away the rest). Another opportunity would be that the other two federates publish the data that the third federate actually wants, but in that case they have to "double post" some data. A better approach is of course to only have one value in each parameter. This was unfortunately considered to be unreasonable in this prototype, with the stated criteria in Section 3.2 on page 24. Therefore this prototype has been implemented using only one parameter, containing one array with multiple values of data.

5.2.1 *The Code Implementation of the Two MATLAB Federates*

The step-by-step overview of the two MATLAB federates is shown in Figure 24. Below follows a brief description of each step.

Step 1: Both federates connect to the CRC part of the pRTI, located on the same Windows computer. Unfortunately, they must connect using the setting *HLA_EVOKED* since *HLA_IMMEDIATE* is not supported.

Step 2: During this step, federate 1 creates a federation named *MyTestFederation* and specifies a FOM file. When the federation has been created both federates join, with the names *Federate1* and *Federate2* respectively.

Step 3: Both federates perform a function call to get the handle to the interaction class named *DataParameters*. Next, the federates use their respective handles to make one call to get a handle to the single parameter array.

Step 4: The federates subscribes and publishes to *DataParameters*.

Step 5: Here, *Federate1* begins with calling the MATLAB function *TIC*, used to measure elapsed time. Then, the parameter array is filled with a number of "dummy" values. Next, *Federate1* encode the parameter and sends it as an interaction through its LRC. In the meantime, *Federate2* keeps asking its LRC for any new interactions. Since *Federate1* now has sent one, *Federate2* will find out about that and execute its *receiveInteraction* function. In that function *Federate2* will receive the parameter and decode it. After that, the same amount of "dummy" values are encoded into the parameter and sent as a new interaction, back to *Federate1*. At this moment, *Federate1* does nothing but asks its LRC for new callbacks and finally gets the one from *Federate2*. *Federate1* then executes its *receiveInteraction* and maps and decodes all parameters. When that has been done, it calls the function *TOC,* and the elapsed time is calculated. The procedure is then repeated from the beginning of *step 5*.

Step 6: When enough iterations have been performed, sending and receiving interactions stop.

Step 7: The federates resign from the federation.

Step 8: When both federates have resigned, *Federate1* makes a function call to destroy the federation.

5.3 RESULTS

Figure 25 shows the results of sending one 64 bits value inside one parameter from Federate 1 to Federate 2. The results in the figure originates from an RTT test, where all measurement values have been divided by two to present the one-way time. Since the federates are located on the same computer the Ethernet related delays are not included. However, since the amount of data is very small (64 bits) the Ethernet delay for each measurement is negligible with respect

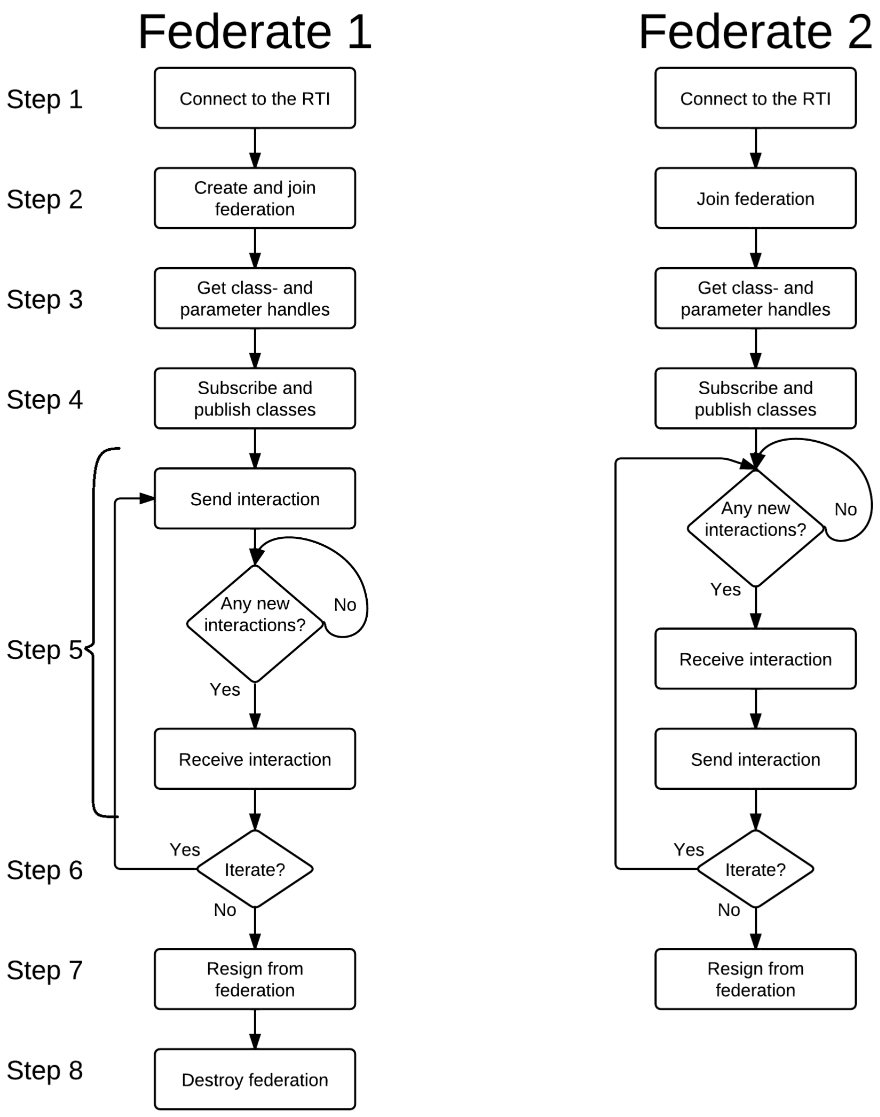

Figure 24: An overview of the actions that the two MATLAB federates perform.

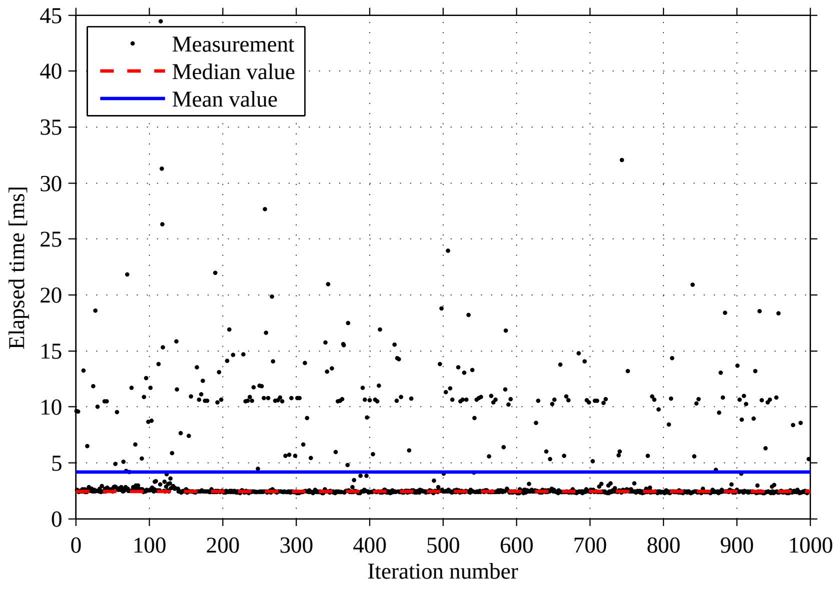

Figure 25: One 64 bit value inside a parameter has been sent iteratively 1000 times from Federate 1 to Federate 2. Both the median value (the red-dashed line) and the mean value (the blue line) have been plotted. The median and mean values are ~2.4 ms and ~4.2 ms respectively. Approximately 80% of all measurements take less than 3.0 ms.

to the total transfer time. As the figure shows, some measurements take relatively long time to perform which makes the mean value (the blue line) a bit off from the main part of the measurements. The increased execution times occur due to a couple of reasons. For example, the Windows XP computer perform many other processing tasks which could interrupt the code execution, resulting in additional delays. Also, it is not optimal to run the two federates plus the CRC part of the pRTI, all on the same computer. By doing this, the risk of obtaining measurements with additional delays is increased. In order to show a more fair view of what usual time measurements take, the median value has been plotted (the red-dashed line).

In Figure 26, multiple 64 bits values presenting one-way delays have been sent between the two federates. Independent of the number of values being sent, there is always at least a ~2.4 ms delay. The largest part of this delay is an overhead that occurs because of MATLAB. The next largest part is due to the HLA Toolbox. Since the mean values are a bit misleading, as previously discussed, the measure points in the figure show median values instead. Due to the relatively high overhead, sending 150 64 bits values only takes an additional ~0.1 ms compared to sending just one value. As the figure shows, it is usually possible to send thousands of 64 bits values within 10 ms. When the amount of data reaches these levels, the Ethernet related

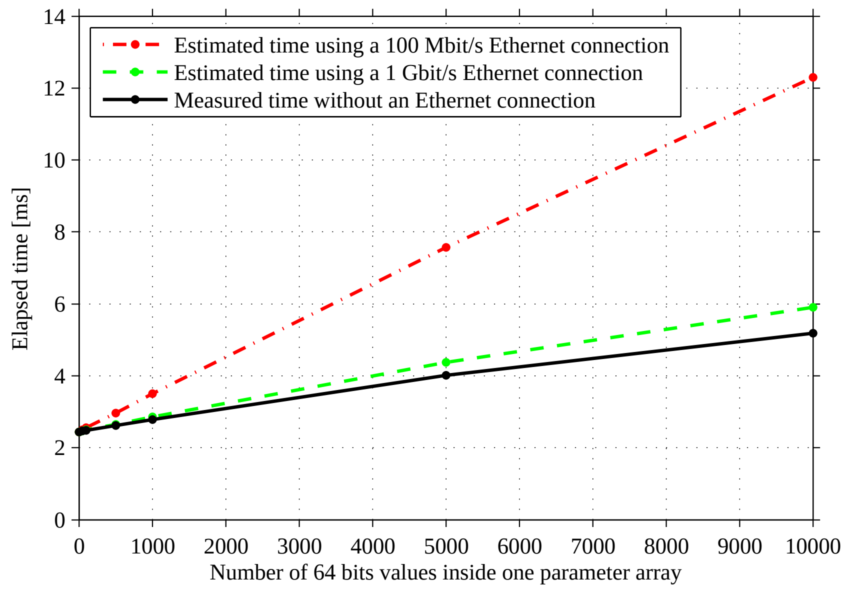

Figure 26: Multiple 64 bits values inside one parameter array have been sent from Federate 1 to Federate 2, running on the same computer. In the black line, each measure point correspond to the median value when 1000 iterations have been performed. The red (dot-dashed line) and green (dashed line) are total estimated time if the parameter array would have travelled over Ethernet, with a bandwidth of 100 Mbit/s and 1 Gbit/s respectively. An estimation of 90% efficiency has been considered during the Ethernet transportation.

delays cannot be neglected anymore, as seen in the figure. The Ethernet bandwidth plays an important role regarding the total amount of time. As can be seen, using a 100 Mbit/s connection really affects the total time whereas a 1 Gbit/s connection only affects theoretically one tenth as much.

A similar thing as Figure 26 shows, but with 16 bits values instead of 64 bits values, can be seen in Figure 27. Both figures have been plotted with the same scale on the axes, for easier comparison. Since the figure with 16 bits values only send one forth as much data in each parameter array (compared with Figure 26) the Ethernet delays are less significant. Ideally, the 16 bits figure would perform four times faster than in the 64 bits figure. By comparing the black lines when 10 000 values are sent, the 16 bits figure only performs $\frac{5.2}{4.0} = 1.3$ times faster. A similar comparison for 100 000 values becomes $\frac{41.5}{16.5} \approx 2.5$ times faster. At the latter case the overhead, which are the same in both figures, play less significance regarding the total time. Four times faster is however not reached, mainly because of the added time needed for MATLAB to handle larger matrices.

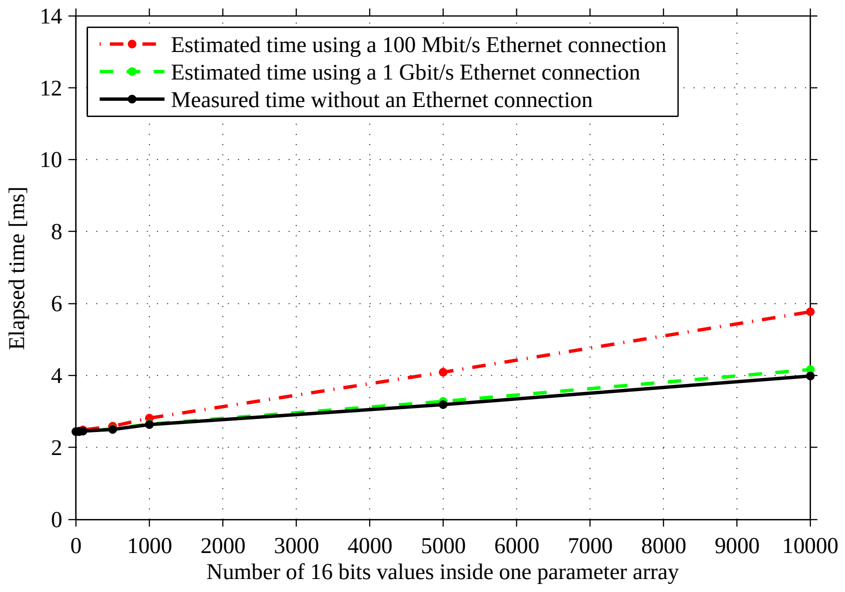

Figure 27: Multiple 16 bits values inside one parameter array have been sent from Federate 1 to Federate 2, running on the same computer. In the black line, each measure point correspond to the median value when 1000 iterations have been performed. The red (dot-dashed line) and green (dashed line) are total estimated time if the parameter array would have travelled over Ethernet, with a bandwidth of 100 Mbit/s and 1 Gbit/s respectively. An estimation of 90% efficiency has been considered during the Ethernet transportation.

5.4 ANALYSIS

The results in this case-study looks quite promising in comparison with the criteria in Section 3.2 on page 24. The amount of data that can be sent within 10 ms is many times higher than the stated criterion of 150 64 bits values (which is usually achievable in ~2.5 ms). However, there are a couple of things which have been modified. For example, all data is sent in only one parameter array instead of multiple separate parameters. This was however considered necessary in order to meet the data amount criterion and not waste all computational power for sending and receiving data. Another thing is that the 10 ms deadline is exceeded with tenths of milliseconds during some percentages of the measurements. This is due to that Windows sometimes perform other tasks which is relatively difficult to fully avoid. Thus, the 10 ms requirement cannot be strictly guaranteed, but on the other hand this is sort of expected when using Windows.

Another thing to remark is that the federates must use the *HLA_EVOKED* connection method. As the federates have been implemented now, when a federate has sent an interaction containing all data, it starts asking the LRC for new incoming interactions. In a real case all time could not be used to check for incoming interactions. The federate would also need to perform its model calculations and other such things. For example, if the federate checks for new interactions every fifth millisecond, an additional delay of 0-5 ms will be added to the total time (depending on when an interaction arrives).

Regarding Ethernet delays, the same reasoning as in Section 4.4 on page 46 would be valid. If perfect conditions are assumed, with a dedicated optical link and no queuing needed, the longest distance is primarily depending on the wireline delay. In Figure 26, for example, it takes approximately 6 ms to send 10 000 64 bits values between the two federates, using a 1 Gbit/s bandwidth. That leaves another 4 ms for the wireline delay. Since that delay is ~0.5 ms per 100 km wire, the maximum distance would in this case be 800 km.

5.5 FUTURE WORK

Suggested future improvements have been presented below.

- Better control the processor usage of the Windows operative system, so that peak values are minimized.
- In a real case, the data that should be shared between federates should match the data in the FOM file. This has not been implemented in this prototype.

6 PROTOTYPING OF SIMULINK FEDERATES

6.1 DESCRIPTION OF PROTOTYPE

In this case set-up the goal was to exchange data between two Simulink federates and evaluate the results. To make Simulink HLA compatible the HLA Blockset was used. Figure 28 shows the preferred test set-up.

6.2 IMPLEMENTATION OF PROTOTYPE

Similarly to the MATLAB case-study in Chapter 5, the set-up had to be modified due to license limitations. The federates run on the same computer but on different MATLAB instances (and different processor cores). Since they do not run over a LAN, the Ethernet related delays have been estimated and added to the results. The set-up is shown in Figure 29.

As discussed in Section 5.2 on page 49, it is very computationally demanding for a federate to handle many incoming parameters during a short time period. However, since Simulink was expected to be more efficient than the MATLAB federates, two different pairs of Simulink federates have been prototyped. In the first pair, values with data are sent in only one parameter array. The second pair was implemented using multiple separate parameters with a single data value in each parameter.

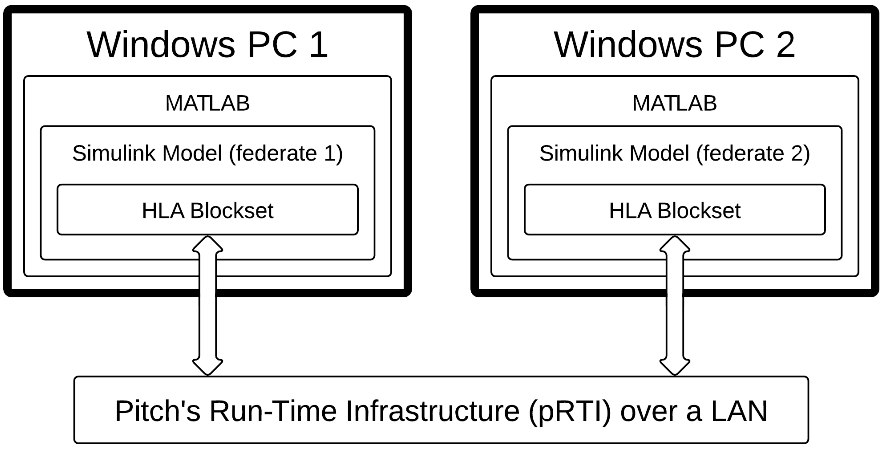
Windows PC 1
MATLAB
Simulink Model (federate 1)
HLA Blockset
Windows PC 2
MATLAB
Simulink Model (federate 2)
HLA Blockset
Pitch's Run-Time Infrastructure (pRTI) over a LAN

Figure 28: Preferred set-up of the Simulink federates prototype.

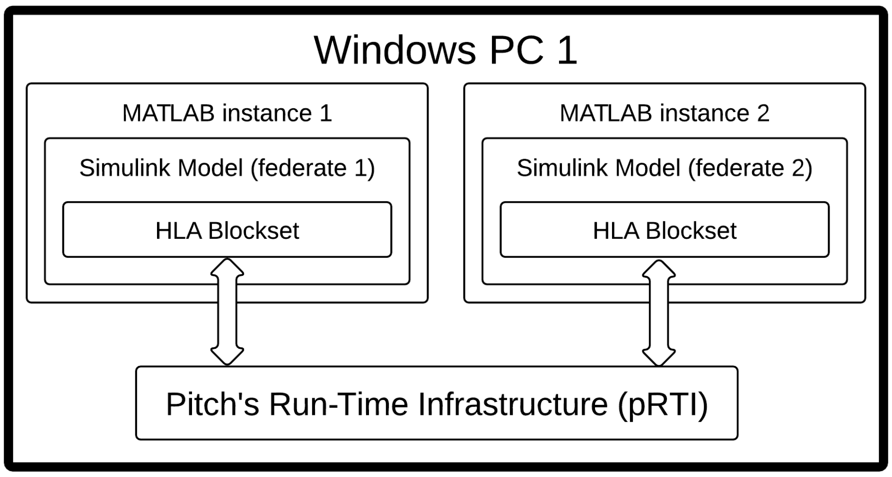
Windows PC 1
MATLAB instance 1
Simulink Model (federate 1)
HLA Blockset
MATLAB instance 2
Simulink Model (federate 2)
HLA Blockset
Pitch's Run-Time Infrastructure (pRTI)

Figure 29: Actual set-up of the Simulink federates prototype.

6.2.1 *The Model Implementations of the Simulink Federates*

Federate 1 of the first Simulink pair, where multiple values are sent in one parameter array, is shown in Figure 30 and 31. A similar representation of Federate 2 is shown in Figure 32 and 33. The models will execute as fast as possible, using a discrete time-step. Below follows a step-by-step description of the execution of the federates.

Step A: Several HLA related functions are performed in the ForwardSim *initialize* block. That block contains a graphical interface where federate specific information and several settings can be adjusted. Examples of such are federate and federation name, connection settings, selection of FOM file and which objects and interactions to include from the FOM file. When starting the executions of the models, this block makes the federates connect to the CRC part of the pRTI (located on the same Windows computer). Next, the federation is created and both federates join, using the *HLA_EVOKED* connection method (*HLA_IMMEDIATE* is not supported). Interaction class handles and subscriptions and publications of the parameter array (*Float1*) are then taken care of, all with help of the *initialize* block.

Step B: During the first time-step, the *initializing subsystem* block in Figure 30 will get triggered and thus execute. Inside that subsystem, a number of random values of data type double (64 bits values) will be encoded and sent as an interaction, in the same manner as Figure 31 shows.

Step C: In each time-step that Federate 2 executes the *initialize* block will actually ask the LRC for new callbacks, such as an incoming interaction. Since Federate 2 has subscribed to the interaction that Federate 1 recently sent, it will very soon execute its *receive interaction* block, see Figure 32. The parameter array, *Float1*, will then become decoded. However, since the data is going to be sent as a new interaction to Federate 1, it is being encoded once again. The *receive*

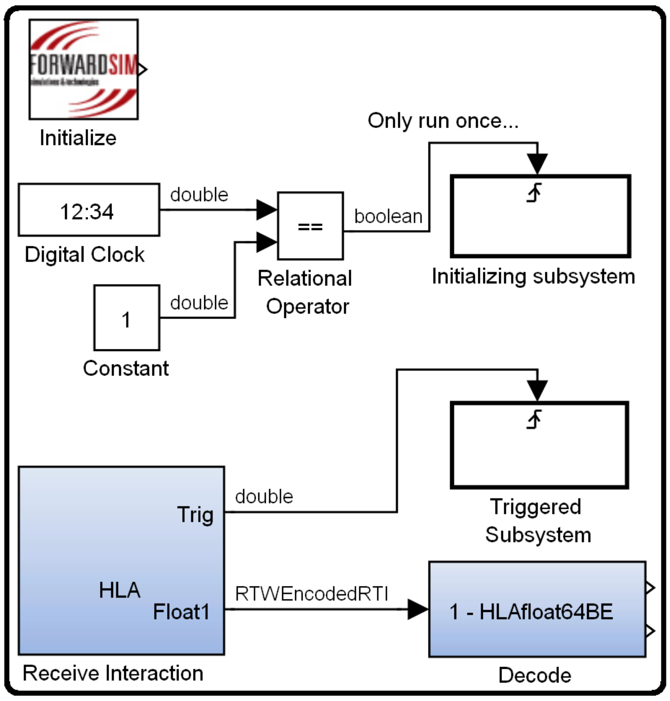

Figure 30: The graphical representation of the Simulink model of Federate 1. Here, only one parameter array is used to send multiple values with data. The *triggered subsystem* is shown in Figure 31.

interaction block has triggered the *triggered subsystem* block, which is where the encoded data will enter and be sent as an interaction, see Figure 33.

Step D: Shortly after that, Federate 1 will get noticed about the interaction and run its *receive interaction* block. The *Float1* parameter array will become decoded, but not used further. The *receive interaction* block will also trigger the *triggered subsystem* block, shown in Figure 31. There, new random double values will be encoded and sent as an interaction to Federate 2. *Step* C and *D* will then iterate until the model executions stop.

Step E: When the model executions stop, the *initialize* block will make its respective federate resign and the federation will be destroyed.

Step F: Simulink has a built-in profiler tool which has been activated on Federate 1. When model execution stops it will show the recorded information, which has been used to calculate one-way delays of the federates.

The second pair of Simulink federates, where multiple parameters have been used (with one single data value in each), is very similar to the first pair. What differs the most is that one additional *encode* and *decode* block is introduced for every parameter being added. So for example, if 100 different parameters are going to be sent then

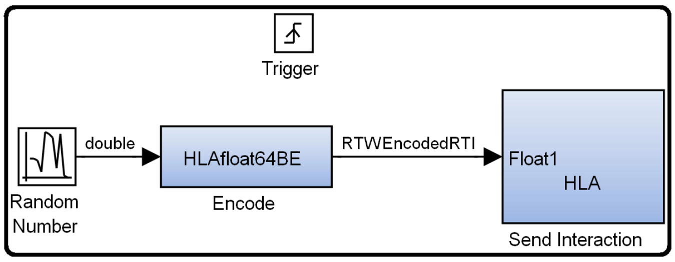

Figure 31: The graphical representation of the blocks inside the *triggered subsystem* of Federate 1.

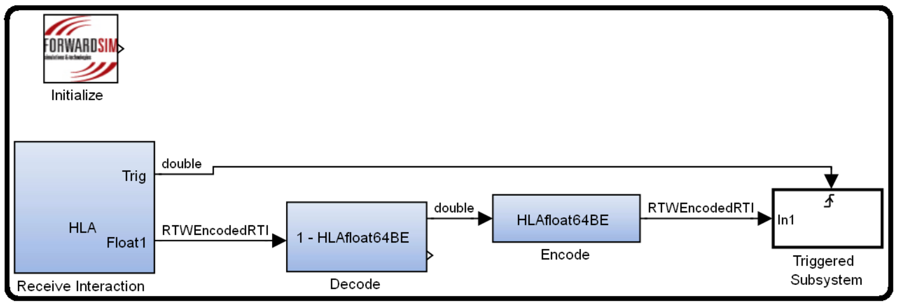

Figure 32: The graphical representation of the Simulink model of Federate 2. Here, only one parameter array is used to send multiple values with data. The *triggered subsystem* is shown in Figure 33.

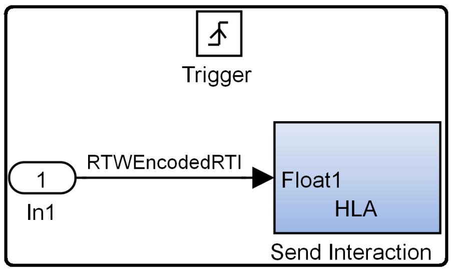

Figure 33: The graphical representation of the blocks inside the *triggered subsystem* of Federate 2.

100 encoding blocks and 100 decoding blocks have been used in each federate.

6.3 RESULTS

Figure 34 shows the one-way delays when sending multiple values inside one parameter array between the two federates. The results have been obtained by the Simulink profiler tool by dividing the total execution time with the number of times the *triggered subsystem* executed in Federate 1. Individual measurements are thus not obtained, which is why there is no standard deviation included in the figure. However, it is expected that some of the measurements take longer time than others, due to that Windows sometimes perform other tasks. So most of the measurements most likely take a little less time than shown in the figure. This can be verified with the results presented in Figure 25 on page 53, where individual measurements were performed during similar circumstances. As seen in Figure 34, sending one 64 bits value takes ~0.8 ms. Most of this time corresponds to an overhead of Simulink and the HLA Toolbox, which always occur independent of the number of values being sent in the parameter array. As shown in Figure 34 and 35, thousands of 64 bits values can usually be sent within the stated criterion of 10 ms.

In Figure 36, results are presented from the second pair of federates. Here, single data values have been sent in multiple different parameters, which is very computationally demanding. According to these results, the criterion of being able to send 150 64 bits values every 10 ms will most likely fail, since the measured mean value is ~14 ms.

6.4 ANALYSIS

Several things are similar to the analysis of the MATLAB federates, see Section 5.4 on page 56. It is usually possible, without using much computational power, to send a lot of data within 10 ms when using a single parameter array. However, since the Windows computers sometimes perform other tasks it cannot be strictly guaranteed. If instead multiple parameters are used, the computational power is more demanding and the data criterion of sending 150 64 bits values within 10 ms will most likely fail. The preferred modeling of the Simulink federates would probably be a combination using *some* parameters, with multiple values in each such parameter array.

6.5 FUTURE WORK

Suggested future improvements have been presented below.

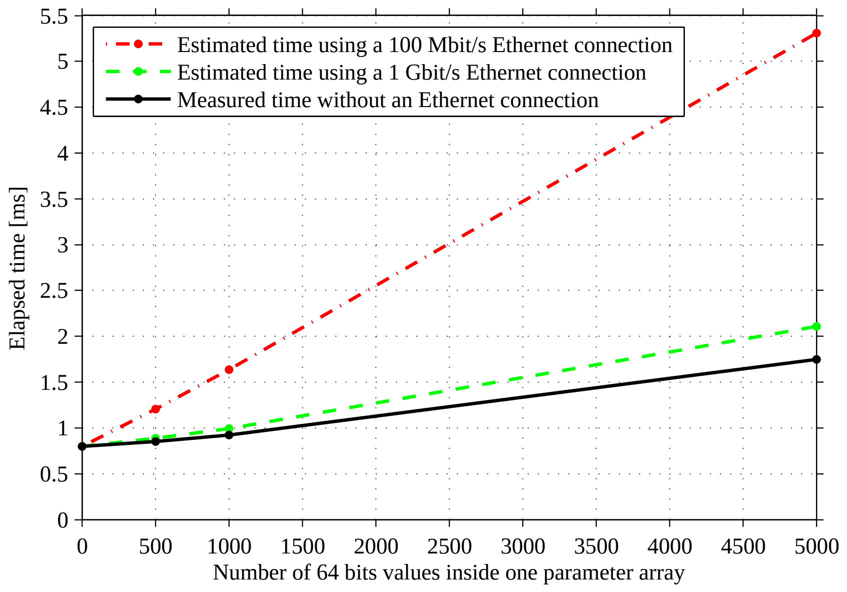

Figure 34: Multiple 64 bits values inside one parameter array have been sent from Federate 1 to Federate 2, running on the same computer. In the black line, each measure point correspond to a measured mean value of a few thousand iterations. The red (dot-dashed line) and green (dashed line) are total estimated time if the parameter array would have travelled over Ethernet, with a bandwidth of 100 Mbit/s and 1 Gbit/s respectively. An estimation of 90% efficiency has been considered during the Ethernet transportation.

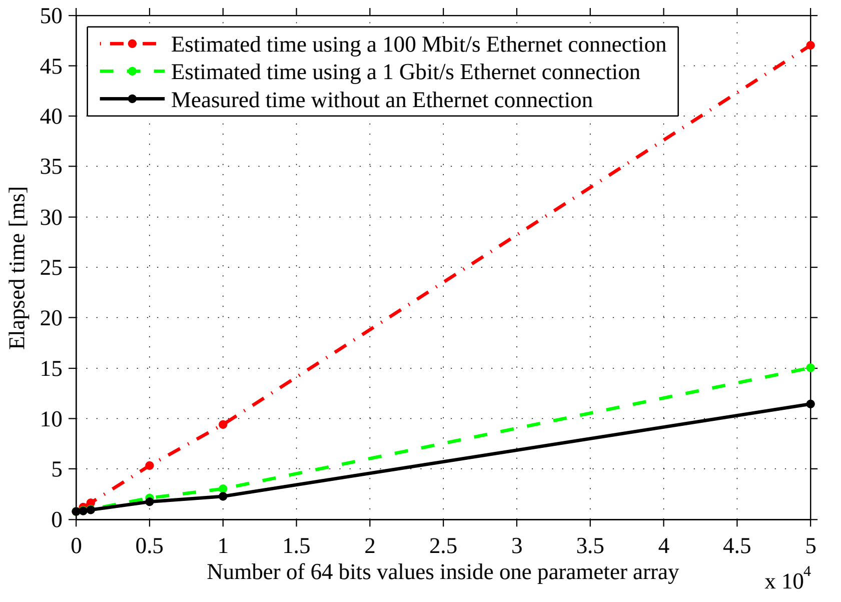

Figure 35: A zoomed-out version of Figure 34, with a maximum of 50 000 values inside one parameter array.

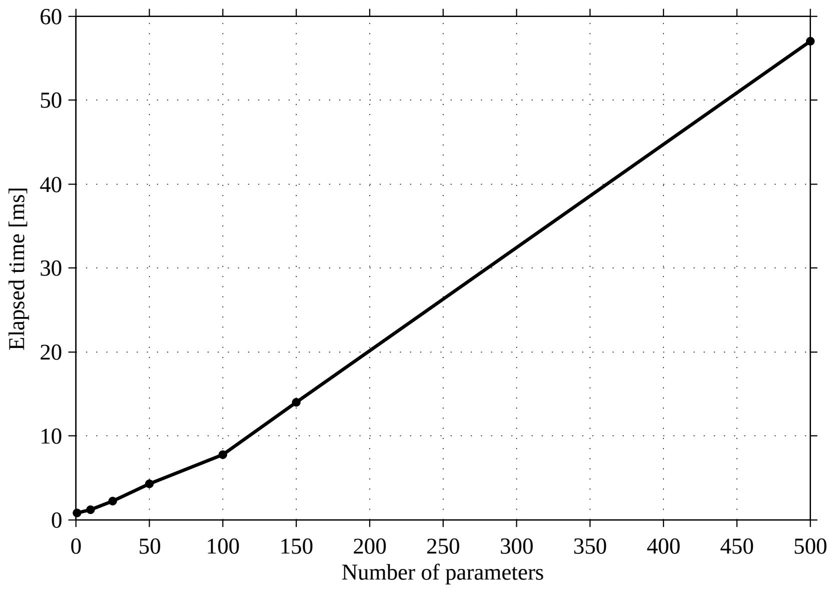

Figure 36: Multiple 64 bits parameters, each containing one value, has been sent from Federate 1 to Federate 2, running on the same computer. Each measure point correspond to a measured mean value of a few thousand iterations. The Ethernet delays have been neglected due to their small impact on the total time.

- Introduce *time management* HLA features into the models. With those features, the models can run synchronized and enter their next respective time-steps at the same time.
- Better control the processor usage of the Windows operative system, so that peak values are minimized.
- In a real case, the data that should be shared between federates should match the data in the FOM file. This has not been implemented in this prototype.

7 DISTRIBUTED SIMULATION WITH SEVERAL FEDERATES

7.1 DESCRIPTION

This chapter deals with situations involving more than two federates. Such scenarios have not been prototyped in this thesis but expected behaviors will be discussed in the analysis section below. It is expected that the reader is familiar with the previously written case-studies.

Figure 37 shows an example of a federation with several federates. Here, 10 different federates are connected, all located on different computers. There is also a computer dedicated for running the CRC part of the pRTI.

7.2 ANALYSIS

Let's assume that all federates in Figure 37 are connected to its own dSPACE HIL-simulator and that the same criteria as in Section 3.2 on page 24 applies. It is also assumed that the federates perform equal to what was obtained in Chapter 4 on page 29. For example, it would take approximately 6 ms to send 150 64 bits parameters between two federates. Here follows a discussion which is of interest regarding several federates.

- Using pRTI, the data sent between federates does usually not go through the CRC. Hence, sending lots of data between federates does not increase the work significantly for the computer running the CRC. Instead, the federates' LRCs send data directly to each other. For example: *Federate 1* publishes some data that *Federate 2*, *Federate 4* and *Federate 7* have subscribed to. Each time *Federate 1* sends an interaction it has to send that data to all the other three federates. However, it is much more computationally demanding for a federate to receive lots of data than it is for sending lots of data. Therefore, problems usually arise for federates which subscribe to many parameters being updated frequently, more seldom on the sender side.

- Since receiving data is the most demanding task, this is most likely what is going to limit the federates. Each federate must limit the number of parameters it receives to a level it manages to handle. A federate which only subscribes to parameters (thus only writes data to its dSPACE simulator) manages to pro-

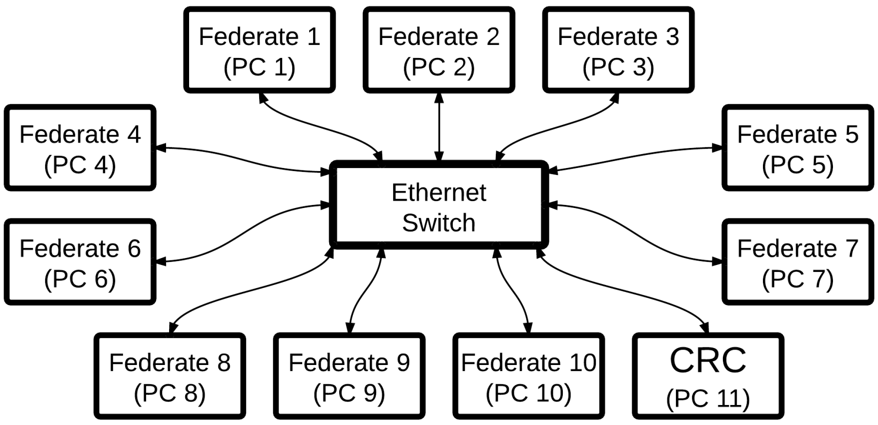

Figure 37: A federation containing 10 federates on 10 different computers. The CRC part of the pRTI runs on its own computer.

cess approximately a maximum of 250 64 bits parameters every 10 ms (obtained from the results in Chapter 4). This is during good conditions with a low Ethernet delay. As an example, it is not possible for a single federate to subscribe to 50 parameters from each federate (a total of 450 parameters), if all that data is received every 10 ms.

- If multiple federates send data in precisely the same moment, the Ethernet switch will have to put them in a queue and process them individually. This will of course add additional delays. If these delays should be a big problem one could for example, if possible, increase the bandwidth. Upgrading all network cards and the Ethernet switch from 100 Mbit/s to 1 Gbit/s will make the data travel approximately 10 times faster. This will not eliminate the queuing problem but since the data travels much faster there will be less queuing and the Ethernet delays will decrease significantly. Another thing that can solve queuing problems is to introduce more switches. If the switches use the cut-through technique they will only introduce relatively small delays and they will help unburdening the Ethernet network.

Let's now assume that the 10 federates are MATLAB or Simulink federates, or a combination of both. Let's also say that each federate publish 150 64 bits values in one single parameter array.

- It would *probably* be possible for a federate to subscribe to all 9 other parameter arrays (from the 9 other federates), considering the delay criterion of 10 ms. However, since the MATLAB and Simulink federates must connect to the CRC using *HLA_EVOKED* they can only execute in a single-threaded way. This means that they have to ask their LRCs for new incoming interactions, which they cannot do all the time in a real case (as discussed in Section 5.4 on page 56). Now, assume that Federate 10 subscribes to all 9 other parameter arrays and that it checks for new interactions every 5 ms. If one or a few of the interactions *have not* arrived to Federate 10 during the period of which it checks for new interactions, those will not be processed until, at earliest, 5 ms later. This might make the data sometimes be considered too old, with the stated 10 ms criterion. This could probably be solved by using HLA *time management* features and synchronize the federates.

- If the amount of data being sent between federates increase it is even more important to use a high bandwidth, such as 1 Gbit/s, and introduce additional Ethernet switches to unburdening the network load and thus avoid additional delays.

Part V

CONCLUSION

8 CONCLUSION

By considering the results obtained from the different case-studies, it should be possible to use HLA in automotive-related distributed systems. However, there are still some problems to be solved and work to be performed in order to be functional in a real scenario.

Regarding the *HIL-simulator federates* case, a couple of different problems arise when trying to make those simulators HLA compatible. Some of the implementation problems here are not explicitly because of HLA, instead they are related to the communication between dSPACE simulator and host computer. CLIB has its limitations and there is an unknown source which periodically causes peaking delays during communication with the dSPACE simulator. There is also work to be done regarding mapping of parameters from the application running on the dSPACE board. Parameters from the generated Simulink model (described in the TRC file) need to be mapped to proper HLA parameters in a FOM file. In order to meet the delay and data criteria, any signals containing only a few bits of data in the Simulink model need to be combined into chunks containing 64 bits of data. This is mainly due to the fact that it is very computationally demanding for the host computers (acting as federates) to handle many parameters during short time periods. The complexity level of this case-study is quite high, partially since several different parts need to work together as links in a chain.

The *MATLAB federates* prototype performs well according to the criteria about data amounts and delays, when sending lots of data in one large parameter array. However, it is more preferred to divide data into multiple parameters. Doing this opens the opportunity for other federates to subscribe to whichever parameters they are interested in, instead of receiving one large array which probably contains excess data, among other drawbacks. Sending data in multiple parameters was however not implemented here. The reason was that it would most likely have failed both in terms of the delay and data criteria and of the amount of computational power used to handle data communication. Thus, if the maximum allowed delay is very low such as in this case, the data needs to be combined in one or a couple of parameter arrays. The HLA Toolbox is relatively easy to work with. It comes with a graphical user interface which can be used to generate essential federate code which one can continue to build upon.

In the *Simulink federates* prototype, data values were sent both in one large parameter array and using multiple parameters, containing a single data value in each. Simulink did almost manage to send the stated amount of 150 64 bits parameters within the 10 ms requirement but exceeded it with a few milliseconds. Also, the amount of computational power needed to do this most likely exceeded the level reasonable to use for data communication. Sending the data in one parameter array performed even better than what the MATLAB federates achieved. However, also in Simulink the preferred way would be to combine data into arrays, instead of sending individual values in separate parameters. The graphical environment of Simulink and the HLA Blockset makes it easy to get to start with, with relatively low HLA knowledge requirements.

In all three prototype cases there is a need to build proper FOM files, describing all parameters and parameter arrays to be shared in the respective federation. This needs to be done in order to adapt existing models, usually containing a large amount of signals, into HLA federations. The most efficient way of doing this would be with some sort of script which generates a proper FOM file from a file where all signals in existing models are described. However, this could, depending on the existing models, be easier said than done.

My personal reflections are the following:

- *HIL-simulator* federates would *probably* be possible in a real scenario, considering the amount of data and delay criteria. However, it would probably demand a lot of additional work to make everything function properly. Here, several relatively complex steps are included which need to work flawless in order for the delay criterion to be strictly fulfilled. Considering these facts and the problems dicussed above, this is not really recommended.

- *MATLAB* and *Simulink* federates look relatively promising in terms of implementation complexity regarding a real scenario. If it would be considered okay to send data in a couple of parameter arrays instead of single data in each parameter, the data criterion is *definitely* fulfilled whereas the delay criterion is *usually* fulfilled. In order to better satisfy the delay criterion of 10 ms one would have to put some work into the task handling of Windows, to avoid that the deadline sometimes is exceeded. When adapting existing models, one implementation issue would probably be creating a proper FOM file.

- An even more promising use case, especially for Simulink federates, would be *offline* distributed simulations (which do not execute in real-time). An example of this would be a very computationally demanding Simulink model which is too complex or too slow to execute on a single computer. Such a model could instead be split in two or a couple of parts, running as HLA federates on several different computers. With help of HLA related *time management* functions, they can all execute in synchronized time-steps. Doing this would however also need a FOM file, for the data being shared between federates.

BIBLIOGRAPHY

[1] Cisco Systems. Ethernet Technologies, April 2012. URL http://docwiki.cisco.com/wiki/Ethernet_Technologies. (Cited on page 3.)

[2] Petty M and Windyga P. A High Level Architecture-based Medical Simulation System, 1999. URL citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.5373&rep=rep1&type=pdf. (Cited on page 3.)

[3] Klein U. Distributed Simulation for Emergency Management based on the High Level Architecture, 1998. URL citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.5702&rep=rep1&type=pdf. (Cited on page 3.)

[4] Klein U, Schulze T, and Strassburger S. Traffic Simulation Based On The High Level Architecture, 1998. URL citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.882&rep=rep1&type=pdf. (Cited on page 3.)

[5] Corrigan S. Introduction to the Controller Area Network (CAN), 2002. URL http://www.ti.com/lit/an/sloa101a/sloa101a.pdf. (Cited on page 4.)

[6] FlexRay, March 2012. URL http://www.flexray.com/. (Cited on page 4.)

[7] dSPACE - Real-Time Interface for Multiprocessor Systems (RTI-MP), April 2012. URL http://www.dspace.com/shared/data/pdf/2012/RTI_MP.pdf. (Cited on page 4.)

[8] Belanger J, Venne P, and Paquin J-N. The What, Where and Why of Real-Time Simulation, May 2012. URL http://www.opal-rt.com/sites/default/files/technical_papers/PES-GM-Tutorial_04%20-%20Real%20Time%20Simulation.pdf. (Cited on page 9.)

[9] PrecisionMBA. Hardware-in-the-loop, May 2012. URL http://www.precisionmba.com/hardware_in_the_loop.htm. (Cited on page 10.)

[10] Bortolazzi J, Hirth T, and Raith T. Specification and Design of Electronic Control Units, May 2012. URL http://www.cs.york.ac.uk/rts/docs/SIGDA-Compendium-1994-2004/papers/1996/eurdac96/pdffiles/d03_1.pdf. (Cited on page 10.)

[11] MathWorks, April 2012. URL www.mathworks.com. (Cited on page 10.)

[12] MATLAB - The Language of Technical Computing, April 2012. URL http://www.mathworks.se/tagteam/70533_91199v01_MATLABDataSheet_v9.pdf?s_cid=ML2012_bb_datasheet. (Cited on page 10.)

[13] Simulink - Simulation and Model-Based Design, April 2012. URL http://www.mathworks.se/tagteam/43815_9320v06_Simulink7_v7.pdf?s_cid=SL2012_bb_datasheet. (Cited on page 11.)

[14] dSPACE, April 2012. URL http://www.dspace.com/. (Cited on page 11.)

[15] dSPACE Real-Time Interface, April 2012. URL http://www.dspace.com/shared/data/pdf/2012/Real_Time_Interface_RTI.pdf. (Cited on page 11.)

[16] ControlDesk, April 2012. URL http://www.dspace.com/shared/data/pdf/2012/ProductBrochure_ControlDesk_4_2_E_ebook.pdf. (Cited on page 11.)

[17] dSPACE. MLIB/MTRACE - MATLAB-dSPACE Interface Libraries, May 2011. Unpublished material. (Cited on page 11.)

[18] dSPACE. CLIB - C Interface Library, May 2011. Unpublished material. (Cited on page 12.)

[19] Dahmann J, Fujimoto R, and Weatherly R. The Department of Defense High Level Architecture, 1997. URL http://www.informs-sim.org/wsc97papers/0142.PDF. (Cited on page 12.)

[20] Open HLA (oh-la), April 2012. URL http://sourceforge.net/projects/ohla/. (Cited on page 13.)

[21] The poRTIco Project, April 2012. URL http://www.porticoproject.org. (Cited on page 14.)

[22] Pitch Technologies, February 2012. URL http://www.pitch.se/. (Cited on page 14.)

[23] Pitch's pRTI, February 2012. URL http://www.pitch.se/images/files/productsheets/PitchpRTIWeb_0904.pdf. (Cited on page 14.)

[24] ForwardSim, February 2012. URL www.forwardsim.com. (Cited on page 14.)

[25] Pitch's Visual OMT, February 2012. URL http://pitch.se/products/visualomt. (Cited on page 15.)

[26] QLogic. Introduction to Ethernet Latency, August 2011. URL http://www.qlogic.com/Resources/Documents/TechnologyBriefs/Adapters/Tech_Brief_Introduction_to_Ethernet_Latency.pdf. (Cited on page 15.)

[27] Infocellar. Ethernet Frame, May 2012. URL http://www.infocellar.com/networks/ethernet/frame.htm. (Cited on page 16.)

[28] RuggedCom. Latency on a Switched Ethernet Network, April 2008. URL http://www.ruggedcom.com/pdfs/application_notes/latency_on_a_switched_ethernet_network.pdf. (Cited on page 17.)

[29] dSPACE. Real-Time Interface (RTI and RTI-MP) - Implementation Reference, May 2011. Unpublished material. (Cited on page 30.)

[30] dSPACE DS1006 Processor Board, April 2012. URL http://www.dspace.com/shared/data/pdf/2012/DS1006.pdf. (Cited on page 30.)

[31] Transferring Data from Host PC to dSPACE Board, February 2012. URL http://www.dspace.com/shared/support/faqpdf/faq027.pdf. (Cited on page 30.)

Part VI

APPENDIX

APPENDIX A

A.1 FEDERATION OBJECT MODEL (FOM) FILE EXAMPLE

Below is an example of a generated FOM file which contains one interaction class (*FloatNumbers*) with 10 parameters (named *Float1*, *Float2* ... *Float10*). All parameters are specified with the data type *HLAfloat64BE* and to use the *HLAbestEffort* (UDP) transport protocol.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectModel xmlns="http://standards.ieee.org/IEEE1516-2010"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
    schemaLocation="http://standards.ieee.org/IEEE1516-2010 http
    ://standards.ieee.org/downloads/1516/1516.2-2010/IEEE1516-DIF
    -2010.xsd">
    <modelIdentification>
        <name>New Module</name>
        <type>FOM</type>
        <version>1.0</version>
        <securityClassification>unclassified</
            securityClassification>
        <purpose />
        <applicationDomain />
        <description>Description of New Module</description>
        <useLimitation />
        <other />
    </modelIdentification>
    <objects>
        <objectClass>
            <name>HLAobjectRoot</name>
        </objectClass>
    </objects>
    <interactions>
        <interactionClass>
            <name>HLAinteractionRoot</name>
            <interactionClass>
                <name>FloatNumbers</name>
                <sharing>PublishSubscribe</sharing>
                <dimensions />
                <transportation>HLAbestEffort</transportation>
                <order>Receive</order>
                <semantics />
                <parameter>
                    <name>Float1</name>
                    <dataType>HLAfloat64BE</dataType>
                    <semantics />
                </parameter>
                <parameter>
                <name>Float2</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float3</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float4</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float5</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float6</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float7</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float8</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float9</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
            <parameter>
                <name>Float10</name>
                <dataType>HLAfloat64BE</dataType>
                <semantics />
            </parameter>
</interactions>
<switches>
    <autoProvide isEnabled="true" />
    <conveyRegionDesignatorSets isEnabled="false" />
    <conveyProducingFederate isEnabled="false" />
    <attributeScopeAdvisory isEnabled="false" />
    <attributeRelevanceAdvisory isEnabled="false" />
    <objectClassRelevanceAdvisory isEnabled="false" />
```

```
        <interactionRelevanceAdvisory isEnabled="false" />
        <serviceReporting isEnabled="false" />
        <exceptionReporting isEnabled="false" />
        <delaySubscriptionEvaluation isEnabled="false" />
        <automaticResignAction resignAction="
            CancelThenDeleteThenDivest" />
    </switches>
    <dataTypes>
        <simpleDataTypes />
        <enumeratedDataTypes />
        <arrayDataTypes />
        <fixedRecordDataTypes />
        <variantRecordDataTypes />
    </dataTypes>
    <notes />
</objectModel>
```