INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

Sincronização de Alto Desempenho para um Ambiente de Co-Simulação

NUNO FILIPE DA SILVA OLIVEIRA
novembro de 2019

POLITÉCNICO
DO PORTO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Sincronização de Alto Desempenho para um Ambiente de Co-Simulação

NUNO FILIPE DA SILVA OLIVEIRA

Outubro de 2019

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Performant Synchronization for a Co-Simulation Environment

Nuno Filipe da Silva Oliveira

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientador: Professor Doutor Luís Nogueira

Co-orientador: Engenheiro João Monteiro

Porto, 06 de Outubro de 2019

Resumo

A Co-Simulação oferece grandes benefícios na integração de sistemas ciber-físicos e na análise de comportamento de código, permite que os vários subsistemas e a sua cooperação sejam testados durante o desenvolvimento através de um processo controlado de simulação. No entanto, ainda existem desafios com ambientes de co-simulação, como a obtenção de resultados de simulação de forma expedita e a eficaz sincronização dos referidos sub-sistemas. Geralmente existe um compromisso entre os dois. De forma geral, é possível distinguir dois métodos de sincronização, sendo eles sincronização por tempo ou por eventos, tendo cada um deles as suas vantagens e desvantagens para cenários de simulação diferentes. A Co-Simulação pode ser considerada uma simulação distribuída e, por essa razão, herda problemas inerentes ao campo de computação distribuída, entre os quais a sincronização de relógios. A empresa que se propôs a desenvolver o presente tema em parceria com o ISEP, Critical TechWorks, possui uma implementação funcional de um ambiente de co-simulação com um método de sincronização baseado em tempo. O desafio principal desta tese consiste em explorar a implementação outros métodos com vista a compreender e comparar as suas vantagens e desvantagens, e se possível obter um desempenho melhor do que o atual em termos de tempo de computação e produção de resultados.

Os trabalhos de Leslie Lamport ampliaram a visão de como o conceito de eventos pode ser útil para sincronizar relógios em ambientes distribuídos. Ainda assim esses trabalhos não oferecem por si só uma solução para capturar causalidade. Para solucionar esse problema surgiram relógios de vetor, que usam como base os trabalhos de Lamport e que permitem de facto capturar causalidade entre eventos.

Aproveitando esse conhecimento científico, a presente tese propõe uma solução de sincronização baseada em eventos para um ambiente de simulação. A solução apresentada reduz a sobrecarga gerada pelo método de sincronização atual. Nos cenários de teste realizados a solução implementada tende a produzir resultados em menor tempo para os mesmos cenários, comparativamente com a solução existente.

Palavras-Chave: Co-Simulação, Sincronização, Sincronização por Tempo, Sincronização por Eventos, Desempenho

Abstract

Co-Simulation offers great benefits in the integration of cyber-physical systems and code behavior analysis, allowing the various subsystems and their cooperation to be tested during development through a controlled simulation process. However, challenges still exist with co-simulation environments, such as obtaining simulation results expeditiously and synchronizing said subsystems. There is usually a compromise between the two. In general, it is possible to distinguish two synchronization methods, either time or event synchronization, each having its advantages and disadvantages for different simulation scenarios. Co-Simulation can be considered a distributed simulation and, therefore, inherits problems inherent in the field of distributed computing, including clock synchronization. The company that proposed the development of this theme in partnership with ISEP, Critical TechWorks, has a functional implementation of a co-simulation environment with a time-based synchronization method. The main challenge of this thesis is to explore the implementation of other methods in order to understand and compare their advantages and disadvantages, and if possible, to achieve better than current performance in terms of computing time and results.

Leslie Lamport's work has broadened the vision of how the concept of events can be useful for synchronizing clocks in distributed environments. Still, these jobs do not in themselves provide a solution for capturing causality. To solve this problem arose vector clocks, which are based on Lamport's works and that allow in fact to capture the causality between events.

Building on this scientific knowledge, this thesis proposes an event-based synchronization solution for a simulation environment. The solution presented reduces the overhead generated by the current synchronization method. In the test scenarios performed, the implemented solution tends to produce results in less time for the same scenarios, compared to the existing solution.

Keywords: Co-Simulation, Synchronization, Time-Driven, Event-Driven, Performance

Acknowledgments

The elaboration of this thesis would not have been possible without the collaboration, encouragement and commitment of several people. I would therefore like to express my gratitude and appreciation to all those who have contributed to making this a reality. I would like to express my sincere thanks to all.

First of all, to my teacher, Professor Dr. Luís Nogueira, for the cordiality with which he always received me and for the freedom of action that allowed me. It opened me horizons and was fundamental in the transmission of experiences, in the creation and solidification of knowledge and in my successes.

To Critical TechWorks for allowing and supporting me on the undertaking of this thesis.

To my adviser, Engineer João Monteiro for his availability and support over the course of this thesis.

To my friends who were never absent, I thank for friendship and affection that they always had.

To my girlfriend, Filipa Silva, for her unconditional support and for the availability she had for me.

Nuno Oliveira

Table of Contents

List of Figures

List of Snippets

List of Tables

Notation and Glossary

ACPI	Advanced Configuration and Power Interface
API	Application Programmers Interface
APIC	Advanced Programmable Interrupt Controller
CDR	Common Data Representation
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CTS	Composite State Transfer
ECU	Engine Control Unit
FMI	Functional Mock-Up Interface
FMU	Functional Mock-Up Unit
HPET	High Precision Event Timer
HRT	High-Resolution Time
Hz	Hertz
IB	Integration Bus
ICR	Interrupt Command Register
IPI	Inter-Processor Interrupt
IRQ	Interrupt Request
IRR	Interrupt Request Register
MIPS	Millions of Instructions Per Second
MSR	Model Specific Register
NTP	Network Time Protocol
OMG	Object Management Group
OS	Operating System
OSI	Open Systems Interconnection Model
PIT	Programmable Interrupt Timer
PPM	Parts Per Million
PTP	Precision Time Protocol
QIBA	QEMU Integration Bus Adapter
QEMU	Quick Emulator
RAM	Random Access Memory
RT	Real-Time
RTC	Real Time Clock
SMP	Symmetric Multiprocessing
TCP	Transmission Control Protocol
UDP	User Datagram Packet
UTC	Coordinated Universal Time

1 Introduction

1.1 Context and Overview

Complex systems that integrate hardware, software and networking are increasing on an unprecedented scale (Nielsen et al. 2015). Current market needs make the development of such systems to be divided between different teams and/or external suppliers (Gomes et al. 2017). Each team develops a different part of the system that needs to integrate with the other sub-systems, making integration testing a necessity very early on the development process to provide optimal information regarding future problems.

Co-Simulation provides support to perform integration testing from the start of development by allowing each sub-system to be simulated, thus creating a simulation environment that represents the whole system where not only integration testing is done but also requirement compliancy checks, regression and acceptance tests (Gomes et al. 2017). Thus, co-simulation provides a solution to test and simulate systems that are distributed but also dependent on each other, hence a promising approach for interoperable systems development by allowing the development of software and/or hardware to be done independently (Quaglia et al. 2011).

When sub-systems are dependent on one another this generates a problem commonly referred to as a *coupling problem*, which is defined by the inability to test a single sub-subsystem in isolation. Co-Simulation mitigate this because it isolates each of the sub-systems allowing communication to happen trough a data sharing mechanism. Thus, a single sub-system can be tested in isolation, with the rest of them being mocked. But while it helps to mitigate the coupling between sub-systems during developments, co-simulation's main asset is that it allows for complete system testing while simulating the production environment that the software will run on.

1.2 Problem, Motivation, and Goals

Co-simulation approaches the coupling problem by allowing the software to run independently with inter-process communication being provided by some form of data sharing mechanisms, such as a bus or a shared memory structure. Thus, the simulation of software and its couplings is treated as a distributed system.

So, by using a distributed architecture to solve the coupling between sub-systems, co-simulation inherits challenges that are inherent to distributed systems. Namely, clock synchronization and the ordering of events.

These are well-known problems in the field of distributed systems and have been studied for a number of years, at least since (Lamport 1978a). Since then, multiple solutions have emerged, such as Christians' algorithm (Sampath and Tripti 2012) or the Berkely algorithm (Sampath and Tripti 2012). While these solutions work, they do not take into account simulation time; only synchronizing clocks and events using real-time as a reference (Sampath and Tripti 2012). In a simulation, time depends on the program execution progress (Cornell 2006). This measure is referred to as *simulation time* and is intrinsically linked to how fast a program can run in a simulated environment (Cornell 2006).

If the simulation is not supported by a real-time environment, the simulation time differs from real time, thus generating a difference between the two, usually referred to as a delta or offset. This delta/offset is either cumulative or self-correcting. In non-real-time scenarios, certain sections of a program may run faster than others, which tends to make the delta self-correct with respect to real-time, meaning that the delta will belong to a closed value interval. But if a program has exponential complexity, the delta is ever increasing in difference with real time.

The various systems on a co-simulator can exhibit individual skew. Besides this, co-simulation brings two additional challenges: it often does not run directly on top of the OS (Operating System), hence running on a virtual machine (VM); and the virtual machine that it may run on will sometimes simulate the hardware. This means that the architecture of the guest virtual machine and host might differ, this generates a problem that the host's clock source might not be available in the VM. In such cases a virtual clock is used (VMware 2011). This tends to happen with custom software, e.g. software that is built for very specific hardware (Gomes et al. 2017). These challenges make accurate time keeping difficult in co-simulation scenarios when compared against a traditional distributed system.

The main problem considered here is how to synchronize simulation time and guarantee event order across a co-simulation environment that simulates the behavior of multiple electronic components of an automobile as a distributed system for a given environment.

As stated above, several solutions already exist for typical distributed systems, including one already implemented in the target framework of the current work. However, still uses a central coordinator to dictate the simulations pace, which is believed to create a significant overhead. Each individual simulator synchronizes its clock to that of the central controller at each simulation tick, creating the said overhead, since systems must stop at the end of each tick.

Thus, the challenge taken herein takes the path of searching for an alternative solution to the use of a central simulation coordinator that at the same time can guarantee sub-system synchronization. By not using a central controller and thus not making simulators synchronize without need, the performance, i.e. the real time that is spent advancing the simulation is expected to be optimized.

In summary, the goal of this thesis is to explore and compare optimization and synchronization techniques in co-simulation environments aiming to improve the performance of a co-simulator platform. For this, measurements such as how much simulation time has elapsed for a given simulator and how it equates to real-time will be considered, along with a guarantee that events occur in the order they are supposed to, in an effort to obtain the maximum performance possible.

The conclusions will be supported by a comparative study, exposing differences, advantages and disadvantages of the considered solutions.

1.3 Document Structure

This document has the following chapter structure: Chapter 2 focuses on the relevant state of the art for this thesis namely clock synchronization algorithms, clock sources, and co-simulation; Chapter 3 focuses on the business aspect, analyzing the benefits and sacrifices for customers; Chapter 4 details the existing solution, it details how the framework that support the simulation is constructed, the current synchronization method and the network concept implemented. Chapter 5 analyzes several methods reviewed in the state of the art, describing the pros and cons of each and proposing one to be implemented sustained with a justification; Chapter 6 deals with the implementation details of the chosen solution; Chapter 7 contains the comparative study where the method for obtaining results is explained and the those of each synchronization method are compared. Chapter 8 presents the conclusion off this thesis.

2 State of the Art

This chapter presents some relevant state-of-art information related to time sources, algorithms for clock synchronization and event-based synchronization. It provides information on how operating systems are managing time, how processes are scheduled and assigned to a given CPU (Central Processing Unit) core. Moreover, it explains relevant co-simulation platforms in detail and introduces techniques used for synchronization in co-simulation environments.

2.1 Time Keeping

Time keeping on simulators depends on the underlying host system clock (VMware 2011b). The host system uses one or several clock sources to maintain accurate time, such as NTP (Network Time Protocol), PTP (Precision Time Protocol), RTC (Real Time Clock), HPET (High Precision Event Timer), among others. PTP and NTP use the network stack to keep time accurate, while RTC (also referred to as the BIOS clock) and HPET are native to the underlying hardware. Independently of the clock source, almost all clocks in a computer system are based on piezoelectric crystal oscillators. They provide time by oscillating at a manufacture defined frequency and with each oscillation, they either increment a counter or request an interrupt, or both. The counter can be stored on an CPU MSR (Model Specific Register), the RAM (Random Access Memory) or some form of non-volatile memory. The counter will wrap-around when it reaches its max size (i.e. 16 bits, 32 bits,…) (Intel 2004).

Computer clocks are almost exclusively based on hardware oscillators and they suffer from two phenomenon that affect their ability to keep time. The first, called skew, is a phenomena where the same sourced clock signal arrives at different components at different times (Harris 1999) and is often measured in PPM (Parts Per Million) and its maximum value is manufacturer defined.

The skew value will tend to fluctuate because of temperatures variations, aging, voltage instability or the material from which the crystal in the oscillator is made.

The second phenomenon is called jitter and is defined as a short term variation of the clock edge from its ideal location (Smilkstein 2007), it is typically generated by things such as ripples on the 50/60Hz (Hertz) power lines, interference for nearby circuits (Rhee et al. 2009), etc.

2.1.1 **Clock Properties**

Clocks on host computer systems can be have several different properties, such as:

- Interrupt Generation - The ability of a clock to generate interrupts. When the clock ticks, it propagates an Interrupt Request (IRQ) which is then handled by the associated interrupt handler(Bovet and Cesati 2002).
- Frequency - A measure of the number of events that occur in each time unit. In case of a clock, the events are cycles, often called ticks by some authors (Thompson 2009) . When provided at a constant rate, cycles can be counted during a period (e.g. 1 second) to calculate the frequency which is measured in Hertz (Hz).

  $$f = \frac{cycles}{\Delta time} \tag{1}$$

  Equation (1) is used to provide the frequency *f* given *n* cycles over a time interval.
- Period - Defined by equation

  $$t = \frac{1}{f} \tag{2}$$

  is the reciprocal of the frequency and is expressed in International System of Units (SI) time unit (second, millisecond, microsecond, or nanosecond).
- Resolution - The smallest increment between two consecutive time unit measurements. In practice this means that it’s the smallest value a clock can measure. The resolution of a clock can be classified as low or high depending on the size of the increment (Microsoft 2018).
- Offset – Defined by the time difference reported by two clock sources. The offset relative to an absolute time *t* can be expressed as:

  $$Clock_A(t) - t \tag{3}$$

and the offset of off $Clock_A$relative to $Clock_B$ can be expressed by:

$$Clock_A(t) - Clock_B(t) \tag{4}$$

(Rhee et al. 2009).

- Skew – A source of cumulative clock error. It is caused by the same sourced clock signal arriving at different components at different times (Harris 1999), meaning that the same clock can diverge from itself depending on where the measurement is being taken from. A clock is said to be working within specification if its within is manufacturer defined PPM threshold (Sundararaman, Buy, and Kshemkalyani 2005).

- Jitter – It is a source of clock error and is not cumulative. Jitter consist of noise that affects bus lines and tends to oscillate around zero, so, it compensates for the errors it caused earlier (rtsj.org 2018).

- Monotonicity – A monotonic clock is a clock that increases at a constant rate and the following equation

$$Clock_A(1) > Clock_A(0) \tag{5}$$

must always be true, where $Clock_A(1)$ is the second measurement and $Clock_A(0)$ is the first.

- Access Time – Time used to read the clock source. Depending on the source the time taken to read it, varies significantly.

2.1.2 Clock Sources

This chapter introduces and defines the most commonly used clock sources by operating systems. The clock source is always dependent on the architecture of the hardware, except for the last two below clock sources. Most if not all of the below clock sources should be present in any machine (OSDev 2017).

- RTC – The Real Time Clock is based on a configurable frequency; it can generate interrupts and provides a read-write register and a resolution of 1 second. The RTC keeps time in a human readable table with representation for seconds, minutes, hours, days, months and years. Depending on the model, the years may be limited to 100. Its 1 second resolution categorizes the RTC as a low-resolution clock, thus, it's inadequate for use by time sensitive applications. It's also a battery backed clock, meaning that its primary function is to keep track of time when the system is turned off (STMicroelectronics 2004).

- PIT – the Programmable Interval Timer is driven by a 1.193182 MHz oscillator, and provides three separate 16-bit counters. Of these, only channel 0 is interesting for timekeeping, since channel 1 might not exist and channel 2 is connected to a PC (Personal Computer) speaker (OSDev 2017).

  The counter on channel 0 is set by the BIOS during boot to 0xFFFF with an output frequency of 18.2065 Hz meaning that it generates and IRQ (Interrupt Request) every ~54 ms. With each IRQ the counter is decremented at a rate of 1.193182 MHz. The channel maybe set in several modes (OSDev 2017), such as:

  - Mode 0 – Interrupt on terminal count, the PIT generates interrupts until the counter reaches the reload value, when this happens the counter wraps around to 0xFFFF. The reload value can be set at any time and must be 16 bits in size.
  - Mode 2 – Rate generator, the PIT generates one interrupt and sets the counter's reload value and starts counting down again.

  The channel 0 counter can be directly read without problems, it can also be latched by issuing a command to the PIT, when the counter is latched the current value of the counter is copied to an auxiliary register until it's read. The PIT has a skew value of around 30 PPM (OSDev 2017).

- ACPI-PM – the Advanced Configuration Power Interface Power Management timer is a monotonically increasing read-only counter. It runs at 3.579545 MHz and is usually connected to the same physical oscillator as the PIT, because of this, its skew value is usually the same as the PIT. It also has interrupt generation capabilities, these can be setup to generate an interrupt when the high-order bit of the 24/32-bit counter changes. This can be helpful in tracking counter wraparounds (Intel 2005). It was designed to be able to cope with system sleep state transitions, to do this the Operating System's saves and restores the state of the counter using the Power Interface (Intel 2005).

- HPET – the High Precision Event Timer was designed by a partnership between Intel and Microsoft to replace the PIT and RTC. It either runs a 32-bit or a 64-bit main counter running at a system-dependent frequency of at least 10 MHz (Intel 2004), however it is usually connected to a 14.31818 MHz oscillator, running directly at this frequency. The firmware maps a 1024-byte physical memory range to the HPET. Within the 1024-byte address space there are various configuration and status registers that specify the HPET properties, such as the frequency, the mode, either 64-bit or 32-bit mode, the main monotonically counter register, and up to 32 individual timers. Each of these individual timers, contains a comparator register, a capability and a configuration register. Also, they support the one-shot mode of operation (Intel 2004), this mode generates an interrupt when the main counter reaches the same value as the comparator register. This value of this register can be set by software. While all individual timers support the one-shot mode, the periodic mode is only supported by some (Intel 2004). When using this mode, the period is set by first writing to the comparator register, the next write to it defines the period, e.g. waiting 1 second between the first and the subsequent write would define a period of 1 second. When the main counter reaches the value of the comparator, the latter is incremented by the value of the period. Furthermore, each timer also has interrupt generation capabilities that can be set individually for each of the timers, and the HPET can emulate the periodic interrupts of the PIT or the RTC and by using the ACPI power management interface, it can also be saved and restored to its original state when a sleep state transition occurs (Intel 2004). The specification sets that the skew should not be larger than 500 ppm for intervals longer than 1 millisecond (Intel 2004).

- TSC – the Time Stamp Counter is a 64-bit processor register counting the processor clock cycles originally and was introduced in the Intel Pentium processor family to provide a fast-high-resolution clock primarily intended for profiling and benchmarking (Paoloni and Intel 2010). It's internal to the CPU core meaning that it ticks at the CPU clock frequency, so in a sense it has perfect resolution because it has the smallest granularity possible in a computer. It is also fast to read, taking only a few clock cycles to do so, making it the fastest counter available on the x86 platform (Paoloni and Intel 2010). The TSC can be read in one instruction (atomically) using the RDTSC or RDTSCP instructions, the first is a non-serializing instruction, which means that instructions after it, can be executed before it, when they are executed in a processor that supports out-of-order execution. The RDTSCP is the serializing variant, this means that every instruction that precedes an RDTSCP call must be executed before, so out-of-order execution does not affect RDTSCP instruction up until the point it is executed, however subsequent instructions may begin execution before the read operation is performed (Paoloni and Intel 2010). The TSC itself also as 4 variants, the constant TSC, the period does not change with CPU frequency scaling, however it does change on C state transitions, the invariant, runs at a constant rate and it’s not affected by power saving state changes, the nonstop, which has the properties of both constant and invariant and the undefined, that changes with every state transition (Iordache 2019) . A multi-core CPU has several TSC counters, that depending on the variant may or not be synchronized with each other, even if they are, a program can only read the TSC of the CPU core that the program is physically being executed on (Paoloni and Intel 2010). Not all CPU vendors that implement the IA32_64 instruction set have TSC or implement the RDTSC instruction, so portability when using TSC is a concern (Iordache 2019) .

2.2 Clock Synchronization

Clock synchronization is an important part on any modern system. Time is usually perceived as monotonically increasing and always increasing by the same value. The former is generally true, time advances forward, even if the systems are physically separated, time should keep increasing. But the latter is almost never true: different systems will have different clock sources and thus, different resolutions, which leads to clocks being out of synch between each other, because the value by which they advance is different for each one.

Messages traded between these systems, will have a timestamp that appear as if is jumping forwards and backwards which is a non-desirable effect (ntp.org 2019). For example, an E-Mail message might have a sent timestamp with a value stating that it has sent after it was received, even on a single isolated system some applications might have trouble with out of synch time, for example a database trying to recover to the last good state (ntp.org 2019). Because of this several protocols were elaborated to solve time inconsistencies. Below is an introduction to the most used protocols for clock synchronization between systems.

2.2.1 **Network Time Protocol**

The Network Time Protocol (NTP) is used to synchronize time among distributed time servers and clients. It uses the User Datagram Protocol (UDP) using port 123 as both the source and destination, which in turn runs over IP based networks (Lesch 2005). The NTP network gets its time from an authoritative time source, such as a radio clock or an atomic clock attached to the time server ("Network Time Protocol: Best Practices White Paper" 2008). To retrieve the current time, an NTP client makes requests at regular intervals(from 64 seconds to 1024 seconds (Mills et al. 2010)). However this interval is not static, NTP uses an intricate heuristic algorithm to automatically control the poll interval for maximum accuracy while keeping the network overhead at a minimum (Delaware 2014).

NTP employs the concept of a stratum to describe how many NTP hops away a machine is from a trusted time source. The stratum 1 is typically the server or servers with the attached time sources, time is sent up the stratum, from stratum 1 to stratum 2 and so on ("Network Time Protocol: Best Practices White Paper" 2008).

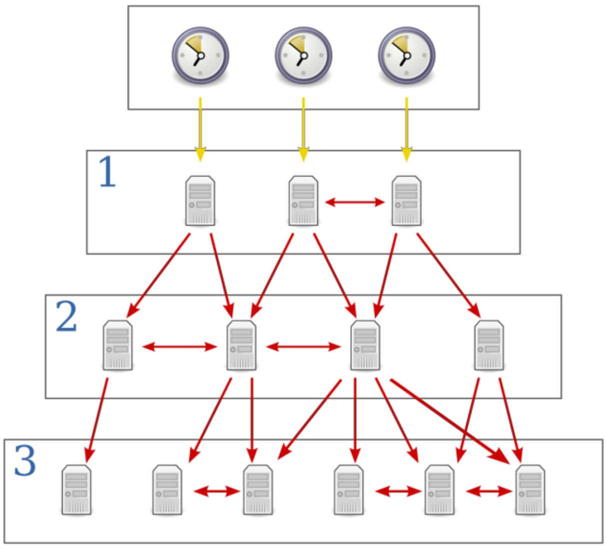


Figure 1 – NTP Stratum

Figure 1 shows how the stratum is divided and how a machine running NTP broadcasts time within and across stratums. Any machine running NTP will automatically choose the machine with the lowest stratum number and communicates with it via NTP, effectively using it as it's time source. This enable the creation of a self-organizing tree between different machines and stratums using NTP ("Network Time Protocol: Best Practices White Paper" 2008). Each machine will tell the other its current time and subtract the network delay.

Typically, the communication between different machines using NTP is statically configured, meaning that each machine has the IP of other machines within is stratum or above it, thus a network of associations is formed, and it's this same network that provides timekeeping by using NTP messages that are exchanged between each pair of machines within an association ("Network Time Protocol: Best Practices White Paper" 2008). Each message contains a timestamp that is either 64 or 32 bits long for seconds (max 136 years) and 32 bits for fractions of a second (up to 0.25us), and the scale used is always Universal Time Coordinated (UTC) (Mills et al. 2010).

In a Local Area Network (LAN) environment, NTP can be configured to use IP broadcast messages instead. This reduces the complexity because each machine can be configured to send or receive broadcast messages ("Network Time Protocol: Best Practices White Paper" 2008).

NTP has a maximum accuracy of 10ms on Wide Area Networks(WAN), and 10us on a LAN (Laird 2012).

2.2.2 Precision Time Protocol

The Precision Time Protocol (PTP) is defined in the IEEE 1588-2008 standard. PTP, as defined in the standard is a network protocol that is used to precisely and accurately synchronize time between different machines distributed across a network. It is used when the machines and the network behavior are relatively stable, and it should only be used across a LAN with the machines involved having clearly defined tasks. It supports both multicast and unicast and can use both at the same time (Montini et al. 2017).

It is intended to measurement and control systems, such as, industrial automation, test environments or simulation environments (Eidson 2008). It was made to be simple and have minimalist resource requirements on networks and host components.

The protocol enables heterogeneous machines with different types of clocks of various levels precision and resolution to synchronize to a grandmaster clock. The latter can also be synchronized to an external time source. The standard defines the states of a clock, the allowed transitions between states, network messages, fields, semantics and describes the datasets that each clock must maintain (Montini et al. 2017). It also defines the available actions and timing of messages across the network.

PTP is designed as a master-slave protocol, where there is one master clock with the other machines being slaves. To synchronize the slaves the master clock sends a synchronization message and a follow up at a preset interval ("Precision Time Protocol Software Configuration Guide for IE 2000U and Connected Grid Switches" 2018).

During the synchronization, two parameters are estimated:

- Offset: Represents the time difference between two clocks.
- Network Delay: The time that a message takes to reach its destiny.

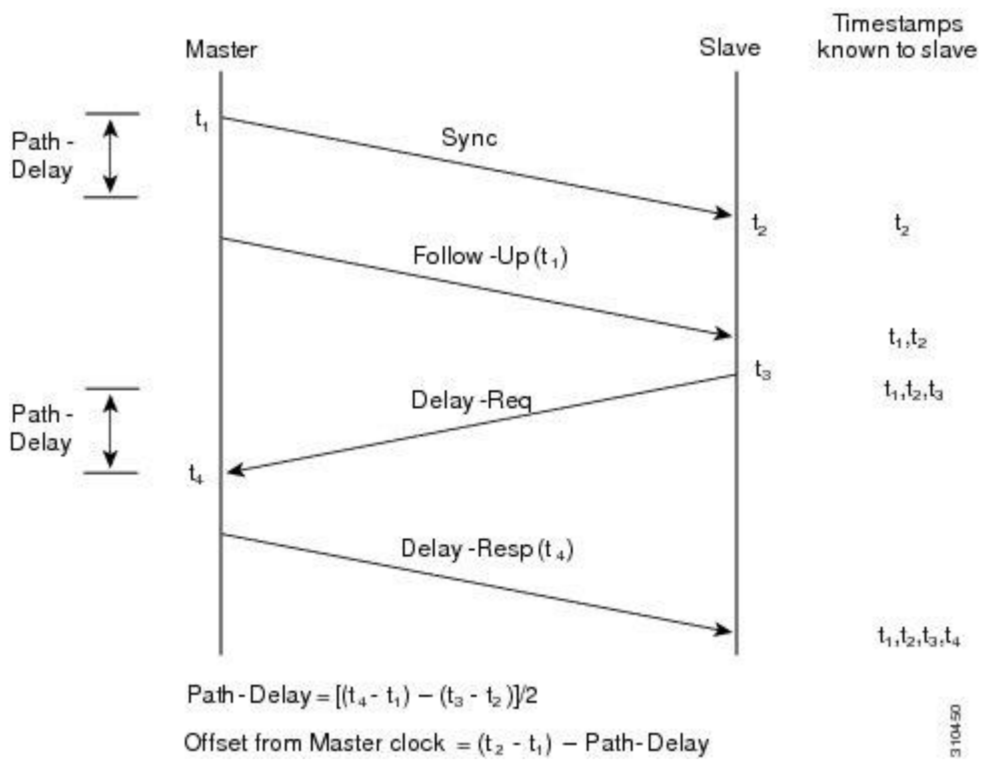


Figure 2 – PTP Network Messages
("Precision Time Protocol Software Configuration Guide for IE 2000U and Connected Grid Switches" 2018)

Figure 2 demonstrates the synchronization procedure between the master and a slave and the network messages involved. Periodically, the master sends a multicast sync message, and since it's a multicast packet, every slave receives it.

The sync message is just an empty packet, before sending the message the master saves time t_1, it then sends the synch message, when the slave receives it, it saves the current time it has on its clock, time t_2, after this, the master sends a follow up message that contains the time t_2, the slave then calculates the delay using

$$Delay = t_2 - t_1 \tag{6}$$

and sends it back to the master, saving time t_3, the master then notes the time $t4$ and sends it back to the slave. After this exchange, the slave possesses all four timestamps and uses them to compute the offset of its clock relative to the master ("Precision Time Protocol Software Configuration Guide for IE 2000U and Connected Grid Switches" 2018).

2.2.3 Reference Broadcast Synchronization

The Reference Broadcast Synchronization (RBS) unlike the previous protocols uses a receiver to receiver synchronization method instead of depending on a central clock and works best on small networks. It can be expanded but the network must support broadcast.

However, it still uses the concept of a central server to broadcast reference beacons. These beacons do not contain any time information, they are empty datagrams, used by the receivers as a reference to calculate the phase offset between one-another. Upon receiving a packet, the receivers compare their clocks with one another to calculate their relative offset. Thus, using RBS, time is relative between all network nodes and dependent on when the node receives the reference beacon (Elson, Girod, and Estrin 2002).

RBS can be used in its most simple form, with one transmitter and two receivers with a beacon being sent at a regular interval or used with n receivers. When using more than two receivers the broadcast rate must be increased to ensure precision, and, since RBS works with User Data Protocol (UDP) which does not mitigate packet loss, the number of broadcasts is increased to minimize packet loss (Elson, Girod, and Estrin 2002).

Ideally every node would receive the reference beacon at the exact same time, but there is always latency in a network, thus, propagation and receive time are still uncertain, but by removing the central clock, and using UDP the propagation time is negligible in networks where the range is relatively small (Elson, Girod, and Estrin 2002).

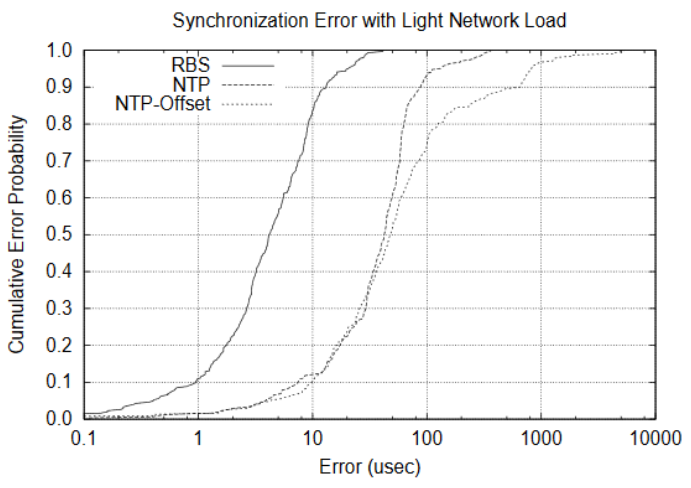


Figure 3 - RBS vs NTP on small Networks
(Elson, Girod, and Estrin 2002)

Figure 3 shows RBS being compared against NTP, it shows that RBS will maintain a low error while NTP and its variant NTP-Offset have a higher error. The cumulative error probability increases at the same rate in all the protocols, neither of them display any major differences when the error probability is taken onto account. But, RBS produces minor errors, from (Elson, Girod, and Estrin 2002) RBS performed more than 8 times better than NTP—an average of 6:45μ sec error, compared to 53:30μ sec for NTP.

2.3 Event Based Synchronization

Unlike clock synchronization, event-based synchronization states that systems do not have to agree on time, but that they agree on the order in which events occur. Furthermore, if two systems do not interact then their clocks do not need to be synchronized absolutely since it would not be observable and thus not cause problems (Lamport 1978a).

2.3.1 **Lamport Clocks**

In a Lamport clock, each system maintains a single Lamport timestamp counter for tagging each event with the value of this counter. This counter is incremented before the event timestamp is assigned (Krzyzanowski 2017).

In Lamport clocks, $a \rightarrow b$ defines a relationship called *happens before*, such that a happens before b, it is a transitive property, meaning if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. This leads to a situation where all events happen to be completely ordered. That is, $a \rightarrow b$, then $Clock(a) < Clock(b)$. However, Lamport clocks do not capture causality (Lamport 1978).

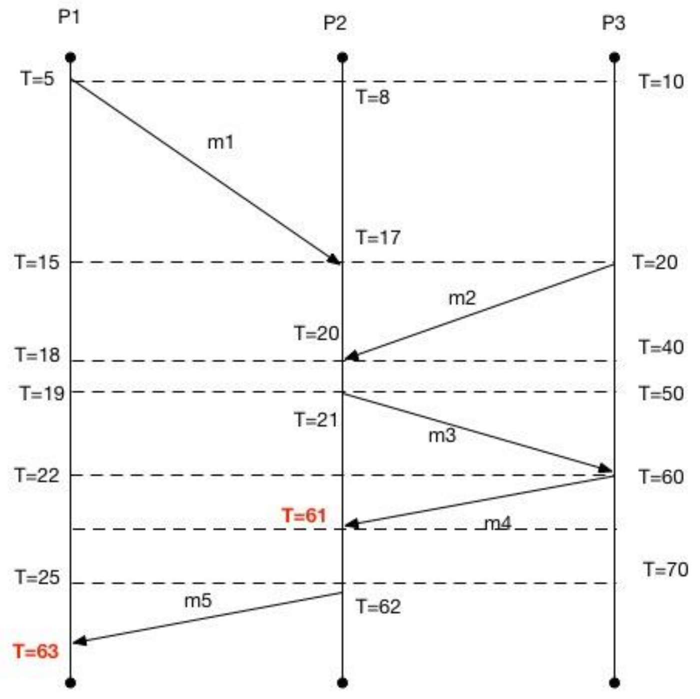


Figure 4 - Lamport Clock Diagram
(Thiruvathukal and Kaylor 2013)

Figure 4 demonstrates how Lamport clocks work. P_n denotes a sub-system n, and all of them form a distributed system. In it P1 sends message m1 to P2 with a Lamport timestamp of 5, when P2 receives m1 it sees that 5 is less than its current timestamp and does not update its counter. However, in the case of messages m4 and m5 the counter is updated since the Lamport timestamp is ahead of the local one, so system P2 updates its stamp when it receives m4 and P1 updates when it receives m5. This behaviour demonstrates how Lamport Timestamps work, when a system sends a message, this message is stamped with the current value of the individual counter of the system plus an increment, usually 1. The receiving system then checks if the stamp matches the last know value, if it's equal or greater than the last value of the counter, it will increment it's counter by one unit (Krzyzanowski 2017).

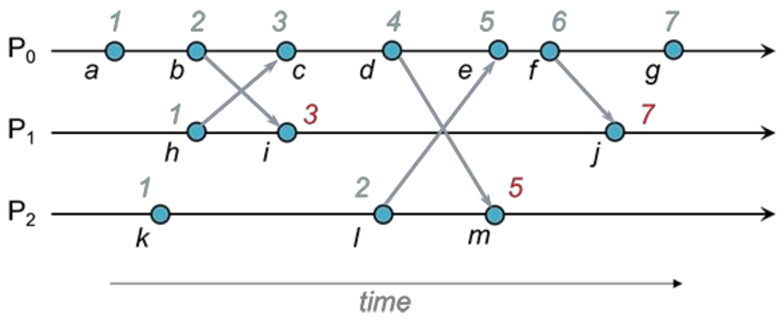


Figure 5 – Event Timestamp Causality

(Krzyzanowski 2017)

Figure 5 shows how Lamport clocks do not capture causality. It's possible to see that the timestamp for event k is less than event i, but it's not possible to state that $k \to i$ or $i \to k$ since k is a local event in $P2$. So, the actual time cannot be captured in terms of real time nor can we state what event caused the other (Thiruvathukal and Kaylor 2013).

2.3.2 **Vector Clocks**

Vector clocks allow for causality to be captured. They work in similar fashion to Lamport clocks but instead of a timestamp, each system has a vector of numbers, with each index in the vector corresponding to a system in the domain (Thiruvathukal and Kaylor 2013).

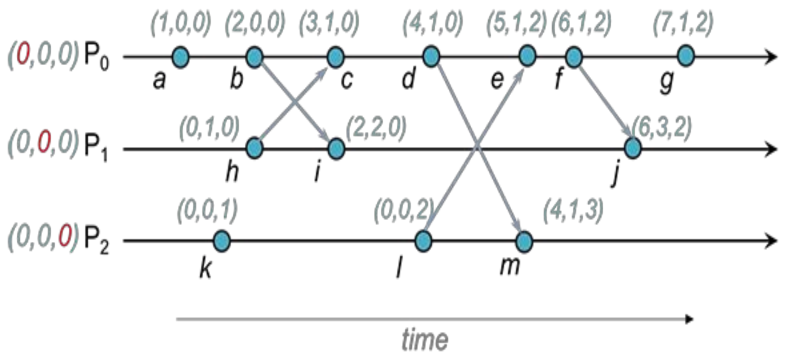


Figure 6 – Vector Clocks
(Thiruvathukal and Kaylor 2013)

Figure 6 demonstrates how each system P_i keeps a vector of length n, each counter inside the vector correspond to a different system, every time a system sends a message to another the recipient systems increments the counter at the vector index corresponding to the sender. Thus, it shows how causality can be captured, like Figure 5, Figure 6 shows the events k and i, but instead of a simple counter, systems now keep a vector, and since the vector for $P2$ at event k is $(0,0,1)$ and the vector for $P1$ at event i is $(2,2,0)$, it's possible to say that $k \to i$ since k has the value of 0 for the first two vector positions and a value of 1 for the last, this means that from the perspective of P2 no other event as occurred in the entire system, while i has the values of 2 for the first two positions and 0 for the last, so two events occurred at P0 and P1 respectively and since the last position is zero no event from P2 has reached P1, so k can only precede i.

2.4 Data Distribution Services

With the advent of the internet, applications, now more than ever, rely on TCP/UDP stack has the underlying framework to communicate. The adoption of these protocols by every major OS manufacture attest to their success and power. However, these protocols reside in the transport layer of the Open Systems Interconnection (OSI) model (Saxena 2014), making them too low level to be used directly. Because of this, protocols like HTTP, FTP, RTP, and SOAP have emerged, they work on the application layer, and are easier to use. Each one has a different design goal and purpose, thus, each one provides different functionality for applications domains. In this context appears the Data Distribution Service (DDS).

A DDS is a networking middleware that has the objective of simplifying the complex network programming involved in communications. A typical DDS will implement a publish–subscribe pattern for sending and receiving data, events and state (Corsaro and C. 2012). By using this pattern, a DDS provides key abstractions to the upper layers, it handles the networking, the memory size and usage, delivery, flow control and manages the data marshalling and remarshaling, thus, it reduces the complex network programming required and provides an abstraction layer. In a DDS there are a set of nodes, called publishers, that create *"topics"*. These can be anything and relate to anything within the system. Once a publisher, publishes something to its topics, the transport layer of the DDS is then responsible to deliver the produced information to the systems that registered (subscribers) to receive updates on the topic. It's possible for any node to be a publisher, subscriber or both.

2.4.1 **Real Time Publish-Subscribe**

The Real-Time Publish-Subscribe (RTPS) protocol is defined and governed by the Object Management Group (OMG) and uses the CDR (Common Data Representation) also defined and governed by the OMG to represent all basic data and structures.

The RTPS protocol was designed to run over a transport layer without any Quality-of-Service (QOS) features, such as UDP/IP, the protocol itself implements reliability in the transfer of issues and state by taking advantage of the multicast capabilities of the transport (UDP). It is designed to promote determinism of the underlying communication mechanism (Thiebaut 2002).

It has two main communication models: the publisher-subscriber and the Composite State Transfer (CTS). The publish-subscriber transfer data between publishers and subscribers, while the CTS transfers state.

According to the specification from (Thiebaut 2002), the RTPS protocol offers the following design features:

- Automatic discovery of new applications.
- Reliable, real time publish subscriber pattern for communications over standard IP networks.
- QOS and fault tolerance properties to enable best-effort for delivery of data and to allow the creation of networks without single points of failure.
- Configurability to allow balancing the requirements for reliability and timeliness for each data delivery.
- Modular approach allows new systems to implement only a subset of the protocol and still able to communicate with full implementation systems.
- Scalability, to enable systems to scale to large networks.
- Extensibility to allow the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- Type-safety to prevent application programming errors from compromising the operation of remote nodes.

The protocol runs in a Domain of several Domain Participants. Each Domain Participant provides local endpoints to be used to send and receive information using the protocol. The endpoints might be readers or writers. The writers provide local data by publishing it to endpoint topic. The information is then relayed to the readers via the RTPS protocol. There are two classes of writers, these are Publishers and CTS writers (Thiebaut 2002).

Publishers provide information to one or more subscribers using a publisher-subscriber patter. Both Publishers and Subscribers are Domain Participants, the presence of a publication is what differentiates the two, when a publication is present in a Domain Participant it indicates that the Domain Participant is willing to publish issues to matching subscriptions on the Domain. The attributes of the publication describe the contents (the topic) that is published onto the domain (Thiebaut 2002).

The CST writer is used as a communication endpoint of the CTS to transfer state to CST readers (Thiebaut 2002). As in the case of writers, there are two classes of readers, the CST reader and the Subscriber, both are Domain Participants.

The protocol delivers two kinds of functionality: the data distribution, where the protocol specifies the message formats and communication that support both the publish subscriber and CTS, and the administration, where the protocol defines the use of the CTS to provide functionality that enables Domain Participants to fetch information about all the other Participants and CTS endpoints in the Domain (Thiebaut 2002). This enables Participants to obtain a detailed picture of the Domain, in which every writer and reader are known, thus enabling Participants to end the data to the right locations and to intercept incoming packets (Thiebaut 2002).

2.5 Operating System Process Management

Operating system (OS) process management refers to how processes are managed under an OS. Particularly, how processes are moved to the CPU and how it can be allocated to serve a specific process. This chapter describes process scheduling and affinity from a Linux perspective, but most of the points discussed hereafter are valid on other operating systems.

2.5.1 Process Scheduling

The process scheduler is a component of the operating system that is responsible for determining which process is in the ready state and should be moved to the running state. That is, decide which process should run on the CPU. Schedulers are either preemptive, non-preemptive or cooperative (Krzyzanowski 2015).

Preemptive schedulers allow the OS to context switch any process. This means that the OS can suspend any executing process and order the execution of another. On the downside, it exhibits more overhead than a non-preemptive scheduler, because it must deal with the context switching of processes alongside with selecting which process should execute next (Krzyzanowski 2015). To make this decision, the scheduler considers 4 events:

- The current task goes from *running* to *waiting* state because it issues an I/O request or some operating system request that cannot be satisfied immediately;
- The current task terminates;

- An interrupt request (IRQ) causes the scheduler to run, and then it must check if a task has run for its allotted time;
- An I/O operation is complete for a task that requested it and the process now moves from *waiting* to *ready* state. The scheduler may then decide to preempt the current task and allow the old one enter *running* state.

By allowing context switches, it inherently allows for higher degrees of concurrency and better interactive performance (Krzyzanowski 2015), but unlike a non-preemptive scheduler execution times might be harder to predict.

Non-preemptive schedulers do not allow for context switching to occur, that is, once a process starts executing, it will not stop until it is finished. However, if an Interrupt Request (IRQ) is thrown the executing process might stop, even so the way interrupts are handled is system dependent, and in some cases an interrupt might not be handled until the current process is executed to completion.

By not allowing context switching, non-preemption, has one advantage: the current process gets all the CPU to itself, making executions times of a process to become fairly predictable (Thekkilakattil, Dobrin, and Punnekkat 2012).

Cooperative schedulers are similar to the non-preemptive, the operating system still cannot context switch a process, but a process might yield control before it finishes execution. An example of this happens when the process is waiting for an input/output operation, thus it’s possible for another process to be executed and take control of the CPU before the completion of the previous process. In short, processes cooperate with one another to minimize CPU idle time (Saewong and Rajkumar 1999).

2.5.2 **Affinity**

Affinity refers to the ability of the operating system to bind a process to a CPU core. This allows for a process to always be executed in the selected core (“Linux System Programming, 2nd Edition [Book]” 2013). The scheduler is responsible for allocating the process to the core and making sure the process only runs on the allowed processors core. Without any type of affinity or a poor one, the scheduler can bounce processes around multiple cores each time they are rescheduled, which leads to a ping pong affect, degrading cache performance.

Multi-core-based computer systems are designed to keep the core's caches valid for as long as possible, keeping the data in its independent cache. Every time a process is rescheduled, the CPU must make sure that the cache is up to date with the RAM. When the cache and the RAM don't match, it is said they are out of sync. The CPU must then invalidate the cache, which is a costly operation. This happens more often if a process is re-assigned between cores because each core will have its copy of data on cache, and each core will have different data, causing more invalidations and thus more cache misses.

CPU affinity is used to optimize cache performance: it works by scheduling threads that have shared data in the same CPU core. Doing so helps stop contention issues over data and lower cache misses' rate. Also, if the application is time-sensitive, setting the CPU affinity ensures that the application receives the full attention of the core it was allocated at.

There are two types of affinity, soft and hard. Soft affinity is essentially the tendency of the scheduler to keep the process on the same core until completion. It is highly dependent on how the scheduler is implemented. For example, pre-2.5 Linux kernel has very poor affinity. Since 2.5 with the introduction of the O(1) scheduler soft affinity got an much better improvement, with the scheduler trying to keep the process assigned to the core it executed on first ("Linux System Programming, 2nd Edition [Book]" 2013)

Hard affinity, on the other hand, is the ability to define it through a system call. Using hard affinity makes sure that for example, a process bound to CPU core 0 will only run in that core.

2.6 Simulators and Emulators

A simulation runs on a simulator, which is an environment that models/mimics the activity of the *thing* being simulated and is used for analysis and study of the *"thing"* under different scenarios (Banks 2001). The simulation environment will control the state and the model of the *thing*, but it will not perform of the real *thing.* For example, a car simulator may model how a car behaves and its different states, like running, breaking, but it can't physically perform any of those actions, thus they are simulated. It's not meant to be a substitute of the real *"thing"*, it is only intended to appear like it (Banks 2001). Thus, a simulator will appear as the real thing but cannot be a substitute for it.

Emulation on the other hand, is for actual usage as a substitute of the *thing*, as in it does (or appears to the user as doing) the same (Bibliotheek 2019). For example, an x86 target architecture emulator will emulate this same architecture and give the exact same output as the real *thing* and can be used as a substitute for an X86 CPU.

The following sub-chapter introduces the Quick Emulator (QEMU), since it is the emulator that is used in this thesis because of its hardware emulation capabilities and open-source nature. Other emulators exist for this same (or more specific) tasks, however a comparative analysis is out of the scope of the current thesis; first, because of the existing dependency on QEMU in Critical TechWork's virtualization projects and secondly because it is not the aim of this thesis to explore what emulators perform better at hardware emulation.

2.6.1 **Quick Emulator**

Quick Emulator (QEMU) was developed to be a fast, cross platform machine emulator. QEMU runs on top of an OS and has the capability to emulate several different hardware architectures (Weil 2019). It can emulate a processor architecture such as, but not limited to X86, ARM and PowerPC and uses binary translation to translate the machine code of the binaries being emulated to the target CPU Host. Binary translation in turn has its drawbacks; while it's very flexible and allows for QEMU to translate machine code between different architectures, it introduces a significant overhead. For example, running an instance of (classic) QEMU to emulate an x86 architecture even if the host architecture is x86, will still translate all the machine's instructions (Weil 2019).

To optimize this, QEMU can work in conjunction with Kernel Virtual Machine (KVM) or XEN. KVM is a hypervisor that offers support for (multiple-running) virtual CPUs powered by hardware management resources such as Intel's VT-X extension. It can only be used if the target architecture matches that of the host. KVM is integrated into Linux as a kernel module and is now part of every major Linux distribution. Because it's a Linux kernel module it only works on Linux or Unix distributions, and the CPU of the host must support virtualization extensions (Graziano 2011). XEN is like KVM, as it's an hypervisor that takes advantage of the underlying CPU virtualization extensions and cannot be run on Windows or macOS. Unlike KVM however, it does not ship natively with Linux. Thus, when using KVM or XEN, QEMU does not and can't perform any type of binary translation because KVM or XEN will take advantage of the hardware acceleration provided by virtualization extensions (Graziano 2011).

QEMU can also be used for debugging and simulation purposes: it provides capabilities to stop the emulator, inspect it's state, save it and restore it (Weil 2019). It can also emulate, hardware devices such as graphics cards, network adapters, hard drives and input/output devices (Weil 2019).

2.7 Co-Simulation

Co-Simulation can be defined as a cooperative simulation where several simulators, each modeling a sub-system, compose the distributed system model being simulated. The systems exchange data between them during the simulation scenario, thus making it possible to verify how all the components of the system perform when combined (Gomes et al. 2017).

Unlike traditional simulation, co-simulation brings additional overhead. Given its distributed nature, systems are independent and must exchange data through some form of transport layer while requiring the simulation to be synchronized across the several sub-systems (Gomes et al. 2017).

The main challenges with co-simulation are to achieve faster simulation results and better synchronization and timing accuracy (Dohyung Kim, Youngmin Yi, and Soonhoi Ha 2005). Synchronization between the several sub-systems creates most of the overhead in the co-simulation environment, thus decreasing simulation speed.

2.7.1 Time Driven Co-Simulation

Time driven co-simulation is a technique used to synchronize sub-systems in a simulation environment. This technique involves assigning the same simulation time step to each sub-system such that they run independently and synchronize at the end of each time step, named after synchronization point. Communication is limited to the synchronization point, i.e., simulators only receive their messages at the synch point making communication limited to the predefined synchronization point (Munawar et al. 2013).

This approach becomes inefficient when events are very irregular over time, because as proved in (Lamport 1978a), if two distributed systems don’t communicate then there is no need to keep their clocks explicitly synchronized. So, with very disperse events the simulators spend unnecessary time synchronizing, resulting in an overhead. Choosing the time step size is a difficult task because of the accuracy/speed trade-off; smaller time steps increase the overall accuracy but reduce execution speed whereas larger time steps improve execution speed but reduce accuracy (Munawar et al. 2013).

2.7.2 Classic Event Driven Co-Simulation

Event-driven co-simulation is another technique used to synchronize systems in a simulation environment, but instead of using time steps to synchronize clocks, it uses events as time markers, meaning that the simulators involved only synchronize with each other when an event is about to be sent (Munawar et al. 2013).

An event scheduler is used to record the system time and to maintain an ordered event list. This list is usually a queue that stores events in their chronological order. Just before the simulation starts the event scheduler receives events from the simulators to add to the list.

When the simulation starts, the event scheduler sends the first event from the list to the target simulators and if this event generates further events, these are added to the list and sorted. The event that was first sent is then removed. The scheduler then jumps to the next timestamp and sends the next event. This process lasts until the whole simulation stops meaning that are no more events in the queue or when a system reaches a certain state (Munawar et al. 2013).

2.8 Co-Simulation Synchronization Techniques

This section details aspects of the general synchronization techniques for co-simulation found in scientific literature and used by Critical Techworks. All the bellow strategies share common traits; they are either time driven or event driven, with most being event driven.

2.8.1 BMW Virtual Validation

The BMW Virtual Validation (VV) project to which Critical Techworks is a contributor, is defined as a co-simulation environment targeted to simulate and test the behavior and cooperation of multiple Electronic Control Units (ECUs) that are part of an automobile under multiple test scenarios.

The VV environment uses RTPS as described in section 2.4.1 as its networking middleware. Because of this, the term Participant from RTPS is adopted and is used to refer to systems involved in the simulation process.

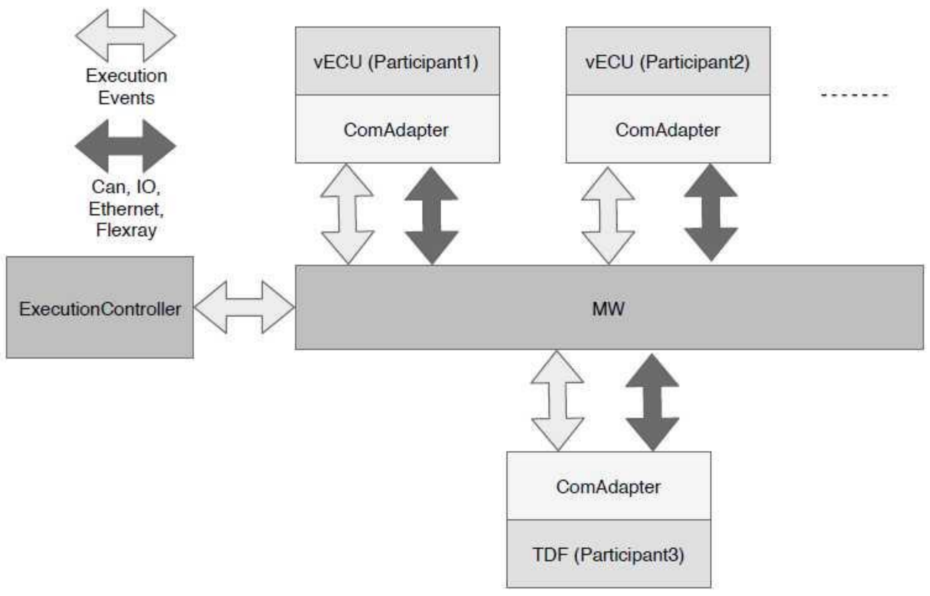


Figure 7 – BMW Virtual Validation Architecture Overview
Courtesy of BMW and Critical TechWorks

Figure 7 shows the current architecture of the VV platform from a high-level perspective. The represented MW component is the data transport middleware responsible for the communication between the different sub-systems, the virtual ECU's (vECU) and the execution controller. The ComAdapter is a networking component responsible for routing the communication between the MW and the virtualized platforms, thus acting as an indirection layer.

For the time being the Virtual Validation environment uses only one strategy to keep time synchronized between the simulation participants. Following is a description of the already-employed Tick and Wait and the Free Run strategies, being the latter devised but not implemented.

2.8.1.1 TickTick Done

TickTick Done is the name of the current strategy used to keep time synchronized across the several sub-systems being simulated. It uses a (centralized) Execution Controller component responsible for dictating the simulation time step. It acts as the central coordinator for the whole simulation.

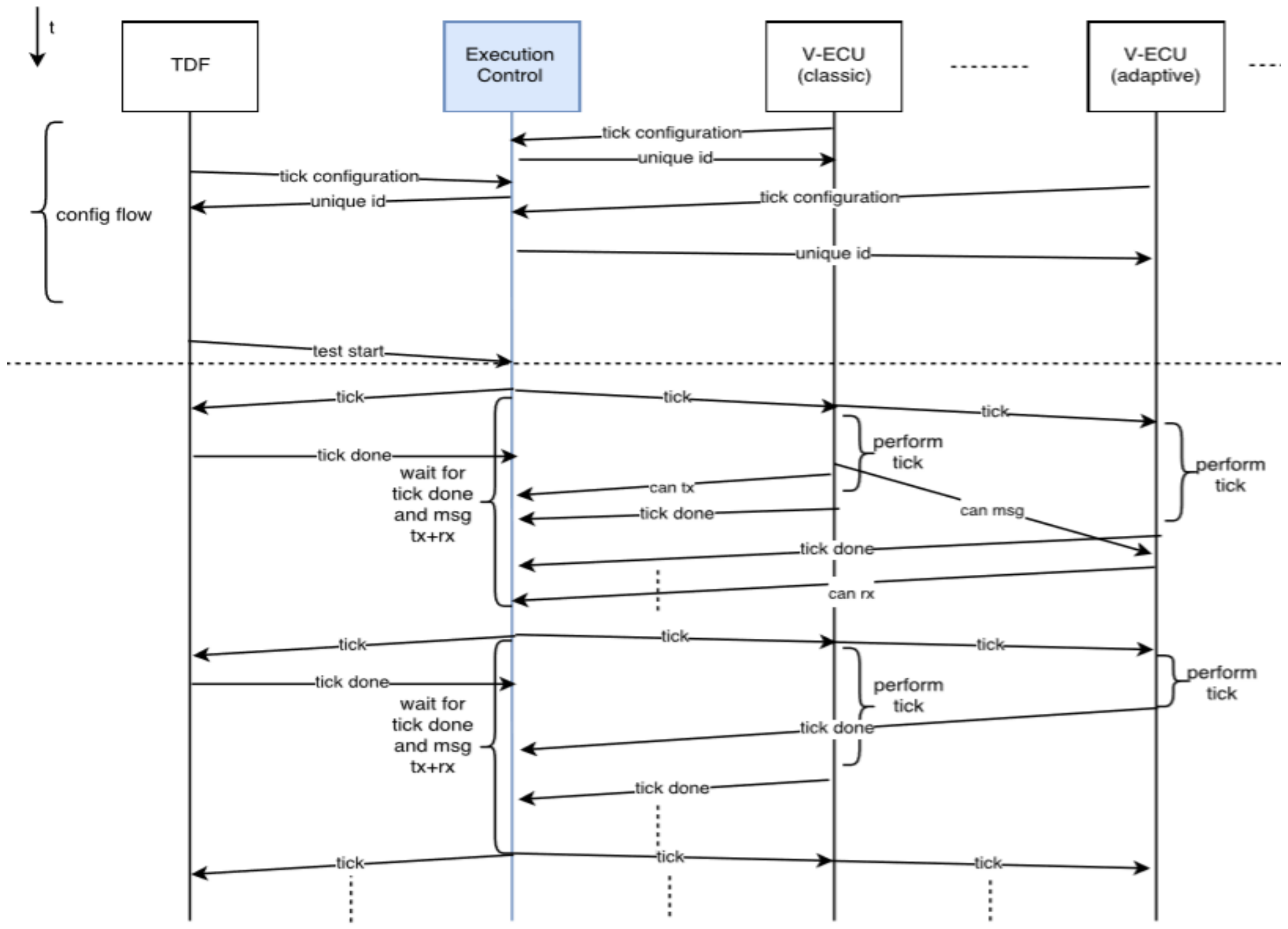


Figure 8 – System Sequence Diagram TickTick Done
Courtesy of BMW and Critical TechWorks

Figure 8 depicts the operation of the TickTick Done method. As the figure shows, the Execution Controller has the responsibility of keeping the simulation in sync and serving as a central clock. It does this by broadcasting a *tick* packet which tells the sub-system simulators to advance its simulated time by one tick, the exact time by which they are advancing is the tick resolution and was set during the tick configuration. When a sub-system simulator completes the tick execution it sends back a *tick_done* packet to the Execution Controller and waits for the next tick. So, the Execution Controller is effectively controlling the progress of all the sub-system simulators. This approach works, but it comes at a cost, the tick flow communication adds an overhead to those sub-systems that execute the current tick quicker than that of the worst-case hence these simulators are kept idle for (eventually) long periods of time until they receive the next tick, leading to a sub-optimal use of computation resources.

2.8.1.2 Free Run

Free Run is another strategy to synchronize the simulation time between participants.

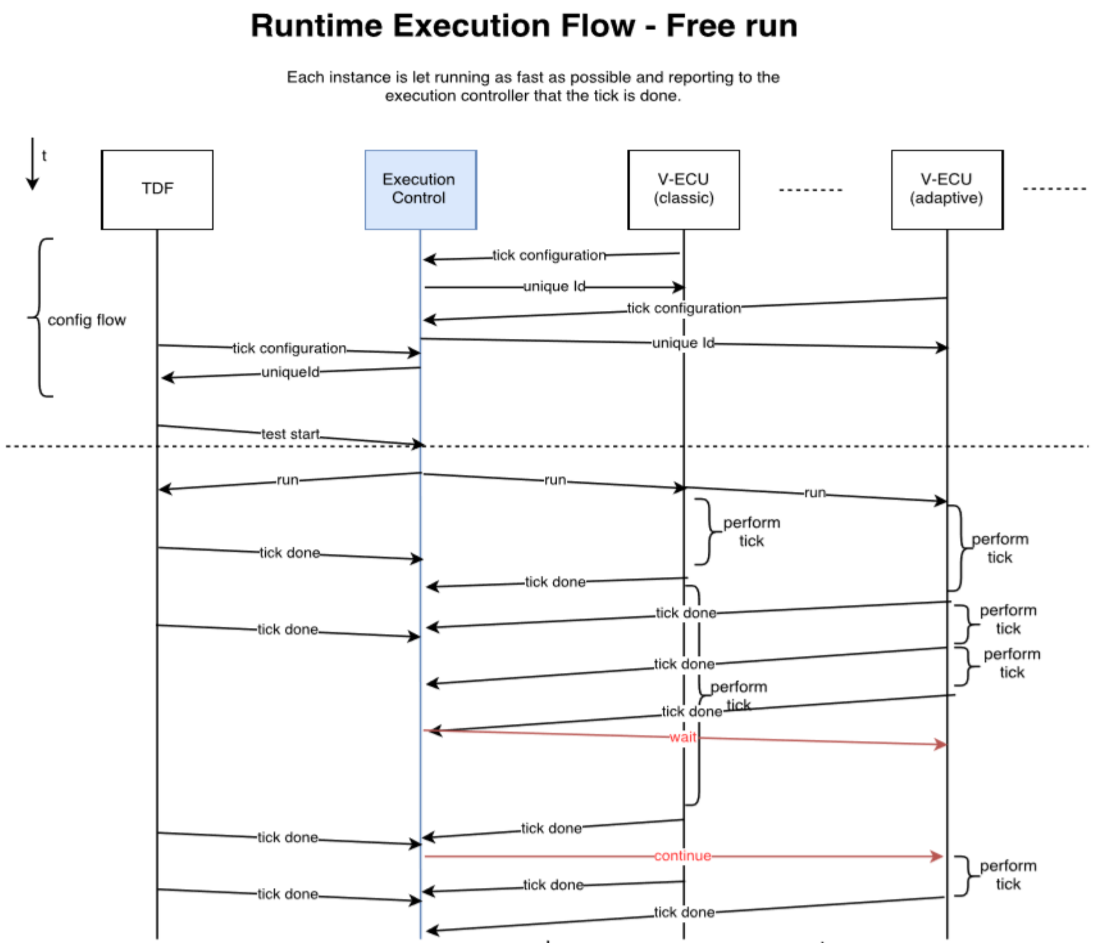

Runtime Execution Flow - Free run
Each instance is let running as fast as possible and reporting to the execution controller that the tick is done.
t
TDF
Execution Control
V-ECU (classic)
V-ECU (adaptive)
config flow
tick configuration
unique Id
uniqueId
test start
run
perform tick
tick done
wait
continue

Figure 9 – System Sequence Diagram Free Run
Courtesy of BMW and Critical TechWorks

Figure 9 demonstrates the behaviour of the Free Run strategy. It is somewhat similar to TickTick Done, with the exception that in this strategy the Execution Controller does not send tick messages. Each Participant performs the *tick* by itself without intervention from other components, sending only the *tick_done* to the Execution Controller so that it can detect when a simulation is moving too far apart from the other.

When that happens, it sends a Wait message to the sub-systems that are ahead causing them to stop. Meanwhile the sub-system or sub-systems that were behind continue performing ticks, and once they reach the same number of *ticks* of the stopped systems a *Continue* message is sent to all sub-systems so that the simulation can resume execution.

This strategy is only thought of, it is not implemented yet.

2.8.2 **KronoSim**

KronoSim is a testing platform design to test complex cyber-physical systems in a closed-loop. It follows a modular and extensible architecture and is usable in multiple domains. It features real-time control allowing the integration of simulation models with physical components. Through the concept of modularity it promotes reuse and reconfiguration, allowing the simulation of simple or complex systems. The later are tightly coupled and communicate with one-another (Chaves et al. 2018).

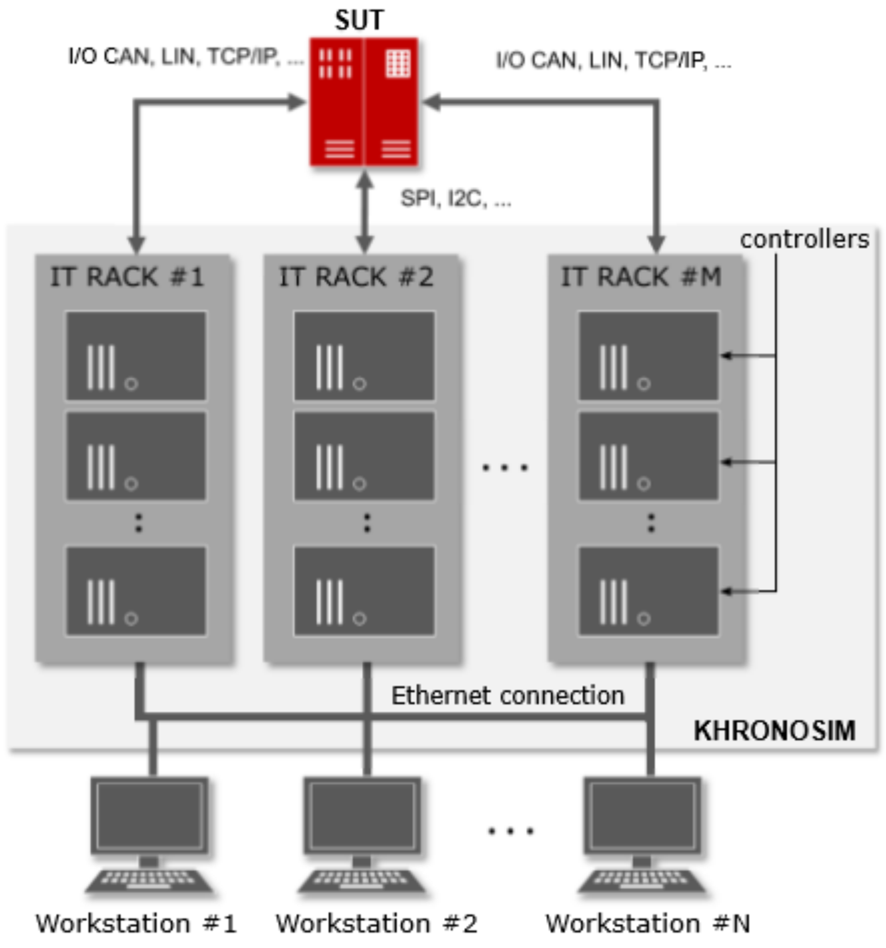


Figure 10 – KronoSim High Level Architecture
(Chaves et al. 2018)

Figure 10 shows the high-level hardware architecture of KronoSim and its modular approach. Each unit in the figure is named Information Technology Rack (ITR) and each one is composed of several real-time controllers. ITR's can be scaled to adapt to the test's needs. The communication between the workstations and the ITRs is made through Ethernet networking, while communication between the ITRs and the Systems Under Test (SUT) can be performed across a wide range of protocols and mediums, such as digital and analogue I/O's, Controller Area Network (CAN), automotive dedicated standards such as LIN (Local Interconnect Network) or Flex Ray, serial connections such as RS232, RS485, or ARINC 429 (Aeronautical Radio, Incorporated 429), Ethernet connections such as TCP (Transmission Control Protocol) (Chaves et al. 2018).

KronoSim uses PTP as its clock synchronization protocol. It was chosen due to its ability to perform measurements and control systems with accuracies in the sub-microsecond range. Also, PTP runs over bare Ethernet networks, Profinet, and Devicenet (Chaves et al. 2018) and is a simple solution to adopt: only a network connection is needed between ITR's (Chaves et al. 2018).

2.8.3 Optimized timed hardware/software co-simulation without roll-back

(Wonyong Sung and Soonhoi Ha 1998) proposed an event driven technique. In it, all simulators notify each other of their next event time, in order to know this time, simulators use lookahead to predict it, meaning they look through the instructions loaded into memory until a specific instruction is found and then, using an heuristic they calculate how much time it takes to reach that instruction. the time of the event. Each simulator continues its execution advancing its local clock freely until its local time matches the smallest event time received. After reaching that stopping point, it synchronizes its local clock with the global one and stops. Once every simulator has completed this last step, the backplane, which acts like a central controlling unit, sends a dummy message to awake the recipient of the event. Upon completing the processing of this message, the recipient sends back the reply to the backplane and stops. The backplane then sends the response to the simulator that originated the event. Since one event, can potentially generate more events as a response, the backplane will keep repeating the process until all events have been processed. When this ends, simulators resynchronize with the global clock once more, and continue to execute freely until the next event. However, this approach requires that the system(s) under simulation allow the retrieving of the next event time using lookahead.

2.8.4 Optimistic distributed timed co-simulation based on thread simulation model

(Yoo 1998) proposes a technique where each simulator runs freely, advancing its local clock unrestricted assuming that no event will arrive. If this does not hold true, and the local clock of the simulator receiving the event is relatively ahead, it rolls back its local clock to the timestamp of the event. Dismissing every computation and events that it may have processed after that timestamp. If a simulator is behind the timestamp of the event it will continue to increment its local until it reaches it, however if the communication between sender and recipient is asynchronous, the sender will continue to run, incrementing its clock. When it receives a reply to its generated message it must roll back its own clock. While this technique can be a good demonstrator of optimistic synchronization, it comes with its natural fallbacks: the more events being exchanged the worse the performance, thus, it does not scale well because the more sub-system simulators are added, the worse the overall performance. However, if the number of event exchanges is a known (or estimated) variable with a low value, this method can provide a good performance and greatly reduces synchronization overhead when compared with time driven solutions.

2.8.5 Trace-driven HW/SW co-simulation using virtual synchronization technique

(Dohyung Kim, Youngmin Yi, and Soonhoi Ha 2005) propose a trace driven simulation. This method is divided into two parts. First it captures execution traces by executing cycle accurate tasks without considering any delay on the systems being simulated, storing them upon completion of the traces. It then evaluates different communication architectures very fast with an almost zero synchronization overhead by simulating dynamic behavior from the architecture using memory traces. However, trace-driven simulation uses a large amount of storage and cannot handle dynamic behavior because of the predetermined trace order. The simulation accuracy is also highly dependent on the OS model and channel model.

2.9 Simulation Modeling

Simulation modeling is employed to create either physical or digital prototypes to analyze their behavior under simulated conditions that mimics the real world. It is a helpful methodology that helps developers understand how the systems they design will behave under different conditions (Maria 1997).

2.9.1 Functional Mock-Up Interface

The Functional Mock-up Interface (FMI) is a standard used for the interface of simulation components in the automotive industry. It defines an interface to develop cyber-physical systems using the functional mock-up unit (FMU). Models built with the FMU under active simulation, are instantiated by simulator calls to the FMI and parsed to the simulator to start executing (Munawar et al. 2013). An FMU may be linked to other components, in that case the simulation environment is a co-simulation one, or maybe standalone. External tools can also be used via co-simulation (Munawar et al. 2013).

2.9.2 SystemC

SystemC is written in C++ as it comprises a set of classes to facilitate the design of simulation processes. SystemC is used to perform system-level modeling, integration testing and software development. It mimics the hardware description language VHDL but is more widely used to model systems (“IEEE Standard for Standard SystemC Language Reference Manual” 2012). Models do not need to be built with SystemC directly, because of its reliance as a super-set of C++, any C++ program is automatically a valid SystemC program.

3 Business Value Analysis

This chapter describes this thesis from a business perspective. It details the concept development, describes and analyses how value is added by the implementation, what it brings to the end customer and exposes the evaluated pros and cons from his perspective

3.1 Peter Koen's 5 Elements

According to Peter Koen the new concept development model (NCD) is made of three essential parts depicted in Figure 11.

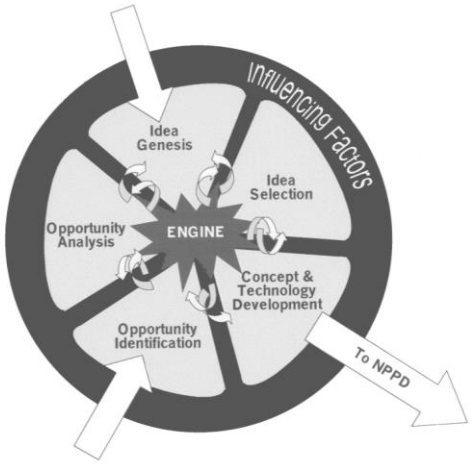


Figure 11 – The New Concept Development Model
(Koen et al. 2001)

The figure shows the new concept development model (NCD) and its three concepts. According to (Koen et al. 2001) the three concepts are referred to as *"Engine"*, the influencing factors and the five key elements. The *"Engine"* represents the management, culture and strategy of the organization and powers the five key elements: Opportunity Identification, Opportunity Analysis, Idea Genesis, Idea Selection and the Concept development.

The influencing factors are something that the organization can only control up to a certain degree. They influence the entire process, from concept inception up until commercialization and consist of the company's organizational abilities, clients, competition, political influence, technologies, etc. In short, anything can be factor if it impacts the concept development.

The analysis of the five key elements in the scope of the work of this thesis results in:

Opportunity Identification – Virtual validation methods are employed in the automotive industry to test several components. Vehicle electronic systems are becoming increasingly complex due to new concept demands that range from increased vehicle comfort and security to autonomous driving. The development of these components requires specific tests from an early development process to ensure and improve sub-system integration.

Opportunity Analysis - To deal with increasing complexity and to reduce the time-to-market, automotive makers are moving towards co-simulation to test the integration of multiple components. However, existing techniques suffer from sub-optimal performance-synchronization trade-offs with great margin for improvement. Mitigating this problem has the potential of making the development processes of (at least) the automotive industry more efficient.

Idea Genesis – The idea of analyzing the current co-simulation synchronization methods came up mainly to produce a performant synchronization method that allows for sub-systems to keep in synch and sacrifice the less performance possible, ultimately leading to faster availability of results

Idea Selection - To select the idea, several synchronization techniques were researched to understand their pros and cons in depth, serving as basis for producing the most adequate solution to the problem in hands. With this, the idea was selected

Concept Development – The concept of the solution to be developed consists in a performant method that allows for distributed simulations to maintain synchronization.

3.2 Value Proposition

The value proposition is a key asset when defining the business strategy for the product or service, because it clearly defines its objective in a precise and specific manner. Ideally it should describe a proposition that creates value in the perception of the client. Therefore, the definition of the value proposition has direct implications on the customers decision to choose a service or product. Based on these considerations the following value proposition was elaborated.

<u>A new synchronization method that offers its customers less overhead and increased simulation speed</u>. Leading to customers spending less time performing simulations to verify the systems they are developing conform to the requirements. It will not only increase simulation speed but also be configurable, so that the developer can chose the way he wants to run the simulation. Together with the benefits of co-simulation, the service offers rapid prototyping capabilities and eases the development process allowing customers to test their systems early in the development cycle, thus reducing latter debugging and refactoring and reducing time to market.

3.3 Value Analysis

Value analysis provides a methodology to help identity the key resources and functions necessary to establish the product or service.

In the context of this thesis, the service is a co-simulation environment used in product development process in the scope of (at least) the automobile industry to develop and test complex control systems, while providing an efficient and performance-driven simulation environment. In this context performance means that simulation time converges towards to real time.

Other solutions do exist such as hardware-in-the-loop testing racks that provide the same functionality as a simulator but unlike it, it requires actual hardware to be present in the loop.

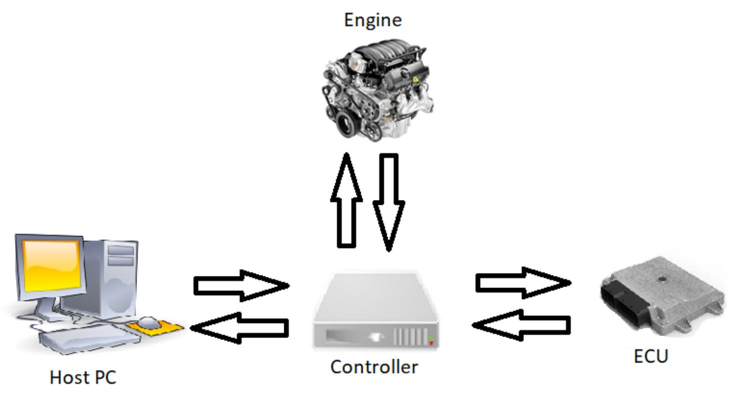


Figure 12 – Hardware in the Loop

Figure 12 shows a hardware-in-the-loop configuration. It is evident that while it provides the benefit of a simulation scenario as close as possible to the real thing, it also has some drawbacks, only one developer at a time can use it, requires physical maintenance and is also more expensive.

When compared against hardware-in-the-loop, a co-simulation environment capable of simulating an automobile's components, allows for each developer to test and develop their code, does not use actual hardware and as such has a lower price tag. However, is not as close to the real thing but it compensates by allowing for faster software iteration cycles.

While the focus of this thesis is the automotive industry, the work developed can be applied to other areas; mainly those that need to synchronize systems in a relative fashion, i.e., the clocks on the systems don't need to be synchronized explicitly.

3.3.1 **Value Creation**

When creating a new service or product what is being offer is value. This value must be quantified and analyzed so it's benefits can be studied (Nicola 2018).

There are three main concepts to value creation:

- Value – It refers to the trade of a product or service, being it tangible or otherwise. The value is dependent of how willing the customer is to pay for the product/service.
- Value for the customer – An evaluation made by the customer(s) between the observable benefits and costs. However a customer not only uses a product or service for the benefit is provides, but also because of his/her self-needs.

- Perceived Value – The quality of a service/product is intrinsically linked to customer perception. When pricing a service/product, the customer is normally kept in the dark of what factors influenced the pricing, so, a customer relies on the emotion appeal and the benefits they believe will receive (Kenton 2018).

In accordance with the service and considering these three concepts the following are the perceived benefits for the service:

- Higher Productivity – Allows developers to test integration during the development process by simulating the hardware and thus, lowers the quantity of bugs found during the end stages of development.
- Cost – Lowers the cost of the testing infrastructure.
- Availability – Unlike hardware-in-the-loop, developers don't need to wait for the testing rack to be available, the software can run directly on the developer's computer.
- Early Feedback – Provides a much earlier feedback to the developer about the system being developed and its integration with other systems, allowing the developer to adjust much earlier in the development process.
- Abstraction – Allows for the study of a problem at different levels of abstraction. A developer may wish to test at a higher level of abstraction to reduce the overall complexity of system, which facilitates rapid prototyping.

The sacrifices for the costumer are the following:

- Slower Testing – A simulation is slow, it does not run in real time, instead running in what's called simulation time, real time is typically greater than simulation time. Simulation time is a representation of how much time has elapsed if the system would run in the actual hardware, while real time, is the clock time measured by a reference clock, that has elapsed since the simulation began. In a perfect co-simulation environment, simulation time would equal real time, but this is never the case, instead real time advances at a much faster rate than simulation time. So, testing becomes inevitably slower.
- System Specific – The co-simulation environment developed only targets the Linux family, so operating systems like macOS or Windows do not work with the co-simulation environment.

- Accuracy – Not as accurate as the real *"thing"*

While slower testing is indeed a sacrifice for the costumer, availability compensates for this and this thesis aims to minimize as much as possible the offset between simulation time and real time.

According to benefits and sacrifices are summarized in Table 1 -Benefits and Sacrifices:

Benefits	**Sacrifices**
Higher Productivity	Slower Testing
Cost	System Specific
Availability	Accuracy
Early Feedback	-
Abstraction	-

Table 1 -Benefits and Sacrifices

3.3.2 Canvas Model

The canvas model is a tool that allows for a quick overview of the business. It allows for a high grain visualization of the aspects surrounding the product/service being proposed.

Figure 13 represents the canvas model for this thesis adjusted to the service specification.

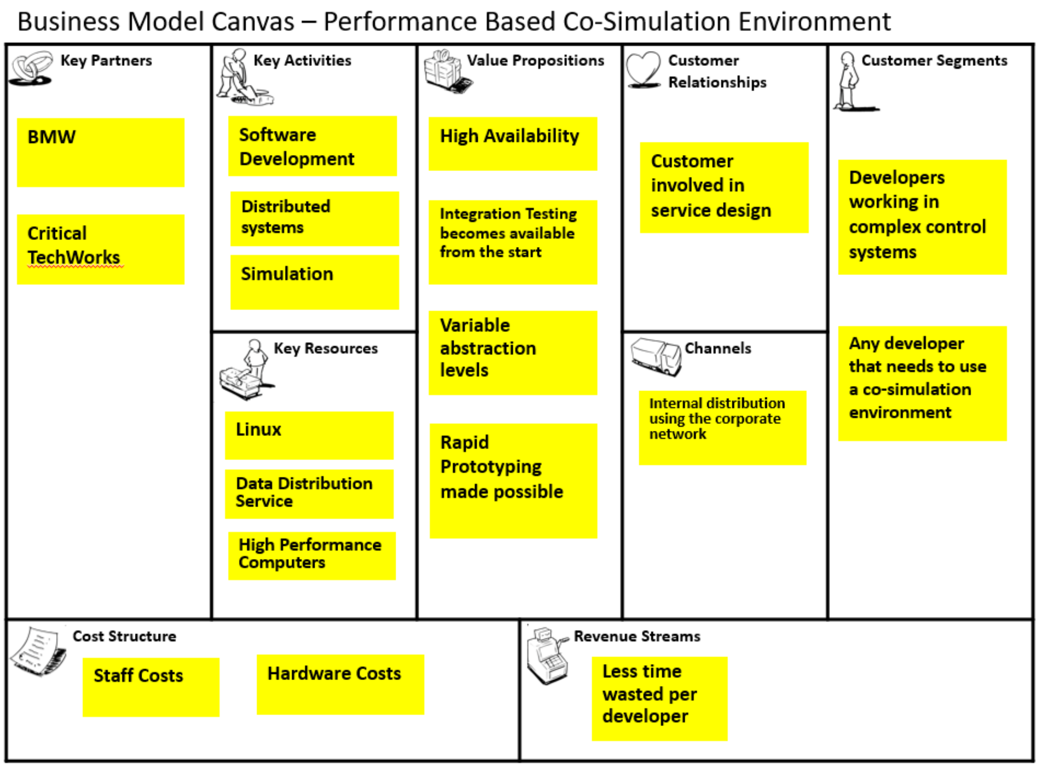

Business Model Canvas – Performance Based Co-Simulation Environment
Key Partners
BMW
Critical TechWorks
Key Activities
Software Development
Distributed systems
Simulation
Key Resources
Linux
Data Distribution Service
High Performance Computers
Value Propositions
High Availability
Integration Testing becomes available from the start
Variable abstraction levels
Rapid Prototyping made possible
Customer Relationships
Customer involved in service design
Channels
Internal distribution using the corporate network
Customer Segments
Developers working in complex control systems
Any developer that needs to use a co-simulation environment
Cost Structure
Staff Costs
Hardware Costs
Revenue Streams
Less time wasted per developer

Figure 13 – Canvas Model

Key Partners – As explained before, this thesis is being conduct under a real business setting in conjunction with Critical TechWorks and BMW. Thus, BMW and Critical TechWorks are its only business partners suited, furthermore the service is for internal use only.

Cost Structure – The cost structure for the service is mainly derived from costs staff for implementing the service, the hardware needed to do so, and the hardware-in-the-loop testing racks to insure the co-simulation environment produces the correct results.

Revenue Streams – The target customer does not have to pay any value. However, development times will go down, making the developers more productive and in turn generating more profit for the company, therefore justifying the investment. So, it's not direct value but rather indirect, in the form of time.

Channels – Only internal channels inside the company will be used to deliver the product to its intendent target. As said, it's for internal use only and it will not be available to the public.

Key Resources – The key resources for this service are the use of Linux since some crucial components of the implementation rely on its features, QEMU as an emulation/simulation platform, Fast RTPS, an implementation of RTPS that acts as a networking middleware and high-performance computers with high random-access memory (RAM) values that can handle several simulators concurrently.

Key Activities – The key activities to satisfy the needs of the customer revolve around software development. Particularly, simulation environments development and distributed systems development.

Customer Segments – The service is aimed at developers that needs an environment capable of simulating several components that are distributed. It targets primarily developers working in embedded systems in the automobile industry, however the market is not the entire auto industry, only those developers working for the company.

Customer Relationships – A textbook definition of a service, is that is impossible to stock it and therefore services are made on demand according to customer specification. This exact definition is being applied to this service, the customer(s) is involved directly in the specification and the iterating life-cycle of the whole software development, with the aim to achieve a service that meets all the costumers needs.

Value Proposition – It's the aim of the service to develop a synchronization method with less overhead for a co-simulation platform, with the aim to increase simulation speed. A new synchronization method not only reduces overhead, but in conjunction with co-simulation leads to higher availability meaning that the service will always be available and can be used by several customers at the same time and being a co-simulation environment, it offers rapid prototyping capabilities, variable levels of abstraction and eases the development process by allowing testing from the start of it.

4 Virtual Validation Framework

This chapter introduces and explores the architecture of the Integration Bus (IB), a MW part of the BMW VV that provides programmers and testers an API that supplies different synchronization methods for the management of a co-simulation, message handling, packet emulation and state management.

Most of the implementation work done in the course of this thesis focuses on the IB, specifically for implementing a new synchronization method that is expected to be useful in coordinating participants (sub-systems) in a co-simulation without the need for a central controller, also named after Sync Master in the VV framework. The framework already provided some synchronization methods, but all of them relied on the central controller. The next few chapters explain some of the most important details about the said framework.

4.1 Integration Bus

The Integration Bus (IB) is a middleware which acts as a network bus, keeps synchronization across participants in the co-simulation environment and supports controls for the different states a Participant can be in. Its communication mechanism relies on FastRTPS, which is an open source implementation of the Real Time Publish-Subscribe standard introduced in chapter 2.4.1. The synchronization is done through its internal methods and the participant state is managed by an integrated state machine. This section address how it creates participants, handles their state and synchronization and how it provides its data bus capabilities.

4.1.1 Participant Instantiation

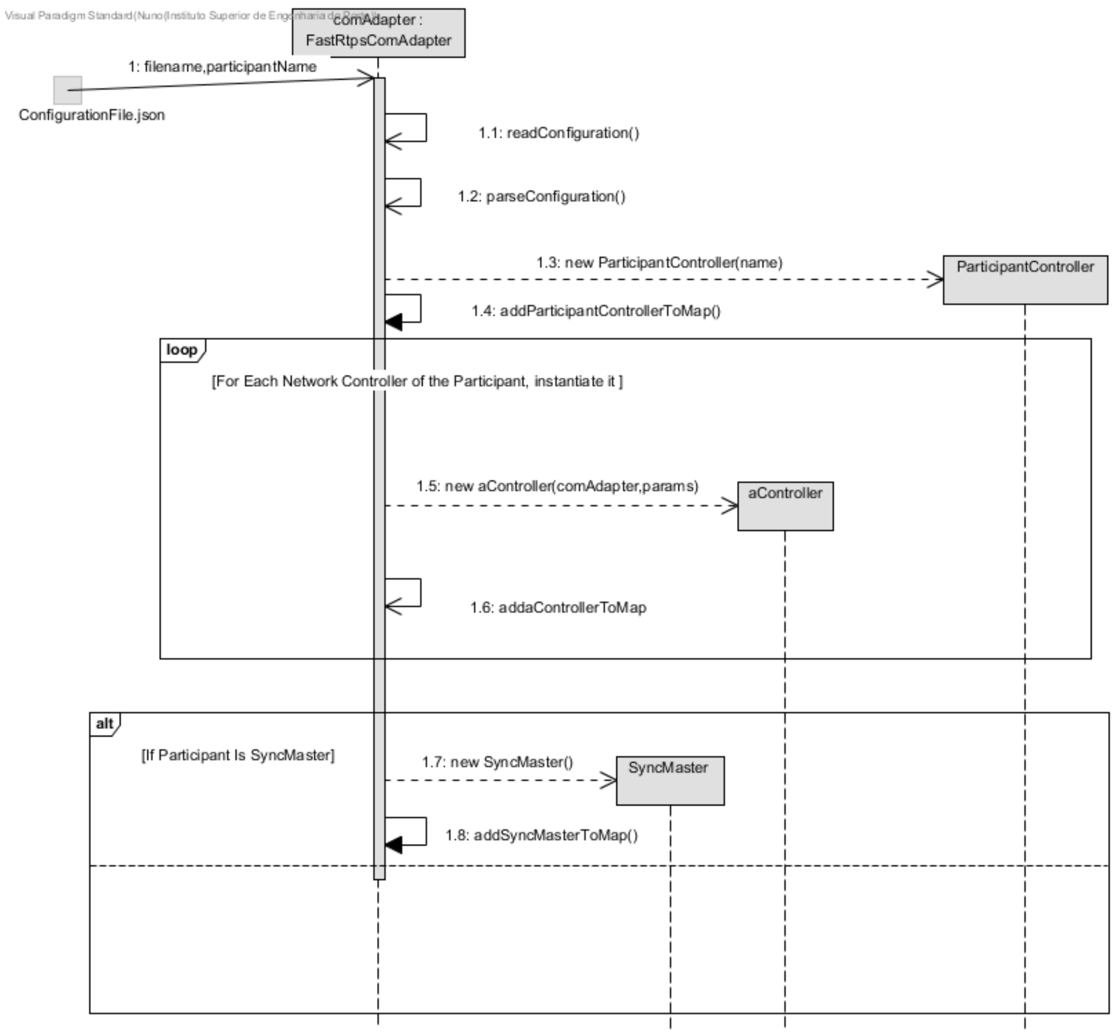
In order for a participant to be instantiated the IB executable must be run as an executable and supplied with a Json configuration file, this file must contain a series of parameters including the configuration for a participant. The following figure shows a snippet of this configuration.

```
"Participants": [
    {
        "Name": "Participant1",
        "Description": "Random Participant” ,
        "IsSyncMaster": true,

        "CanControllers": [ "CAN1", "CAN2" ],
        "LinControllers": [ "LIN1", "LIN2"],
        "FlexRayControllers": [ "FlexRay1", "FlexRay2"],
        "EthernetControllers": [
            {
                "Name": "ETH0",
                "MacAddr": "F7:04:68:71:AA:A0"
            },
            {
                "Name": "ETH2",
                "MacAddr": "F7:04:68:71:AA:A1"
            }
        ]
    },
    {
        "Name": "Participant2",
        "Description": "Random Participant ",

        "CanControllers": [ "CAN1", "CAN2"],
        "LinControllers": [ "LIN1", "LIN2"],
        "FlexRayControllers": [ "FlexRay1", "FlexRay2"],
        "EthernetControllers": [
            {
                "Name": "ETH0",
                "MacAddr": "F7:04:68:71:AA:B0"
            }
        ]
    }}]

"TimeSync": {
    "SyncType": "TickTickDone",
    "TickPeriodNs": 100000
}
```

Snippet 1 – Json Participant Configuration Snippet

Snippet 1 shows an example configuration file containing all the participants involved in the simulation (two in this case). It also contains details of each participant such as their name, which is used to identify them, a description, that should contain a small description about the participant and the different network controllers.

When instantiating a participant, the Integration Bus will match the supplied name with the Participants names declared in the file and from there it will instantiate all its specified network resources need by the participant. For the case of Snippet 1, if the supplied name was Participant 1 it would instantiate two Ethernet Controllers, two Can Controllers, two Lin Controllers and two FlexRay Controllers. The configuration also indicates that Participant 1 is the Sync Master. The latter will be further explained in chapter 4.1.4.



Visual Paradigm Standard(Nuno(Instituto Superior de Engenharia do Porto))
comAdapter : FastRtpsComAdapter
1: filename,participantName
ConfigurationFile.json
1.1: readConfiguration()
1.2: parseConfiguration()
1.3: new ParticipantController(name)
ParticipantController
1.4: addParticipantControllerToMap()
loop
[For Each Network Controller of the Participant, instantiate it]
1.5: new aController(comAdapter,params)
aController
1.6: addaControllerToMap
alt
[If Participant Is SyncMaster]
1.7: new SyncMaster()
SyncMaster
1.8: addSyncMasterToMap()

Figure 14 – Participant Controller’s Instantiation Sequence Diagram

Figure 14 demonstrates how a participant is created. The *FastRtpsComAdapter* (hereafter denoted as *ComAdapter*) is responsible for the instantiation of every controller that a participant needs. The *ComAdapter* aggregates every controller, and it does so through an internal map data structure.

To decide what type of controllers a participant requires, it reads and parses a Json configuration file similar to the above Snippet 1. Given that the configuration is valid and the supplied Participant name exists, it will instantiate a *ParticipantController* add it to its map and proceed to instantiate the necessary network controllers. Figure 14 provides a broad picture of this procedure.

One aspect that should be highlighted is that the *ComAdapter* aggregates the controllers but also injects itself into them. This is because of its responsibilities. To have adequate responsibility segregation, the controllers don't send messages, delegating this responsibility to the ComAdapter that is passed by dependency injection to each network controller. The network controller's job is to emulate packets, Ethernet, FlexRay, CAN and LIN. With each controller type being responsible for one type of packet, e.g. the *CANController* is responsible for emulating CAN packets how to serialize and deserialize them, but it does not send messages from one Participant to the others. That job falls on the Com Adapter, which is the class that handles communication with the underlying *FastRtps* module.

4.1.2 **Message Transmission**

Message transmission between Participants is done through *links,* which can be understood as subnets of a typical Ethernet network. While the host system might only support Ethernet, the Integration Bus has the capability of handling different packets by emulating them.

The IB communication stack emulates Ethernet, CAN, LIN and FlexRay packets. To do this, it implements is own version of these packets and builds the packet as if it was the real thing, using the host to do the actual transport. So, a CAN, LIN or FlexRay packet might be transported over Ethernet without any of the simulation participants knowing about it, all they see is the actual packet, so the host's transport protocol is transparent to them.

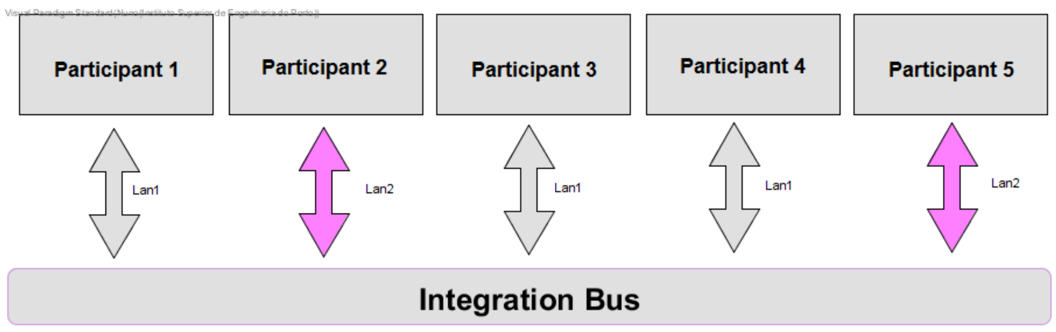


Figure 15 – Integration Bus Middleware

Figure 15 depicts an abstract view of how the Integration Bus handles the communication between participants and how different participants are connected to different *links*. In this example participants 1, 3 and 4 use LAN1 and Participants 2 and 5 are connected to LAN2. Since the Integration Bus uses the Real Time Publish-Subscribe standard it happens to work as broadcast publisher subscriber. So Participants 1, 3, 4 are publishers/subscribers onto the link Lan1 making them to either publish or subscribe messages exclusively for this specific link. Similarly, participants 2 and 5 only publish and/or subscribe messages to *link* Lan2. So, the Integration Bus uses the *link* to define what is commonly known as a topic in a publisher-subscriber system.

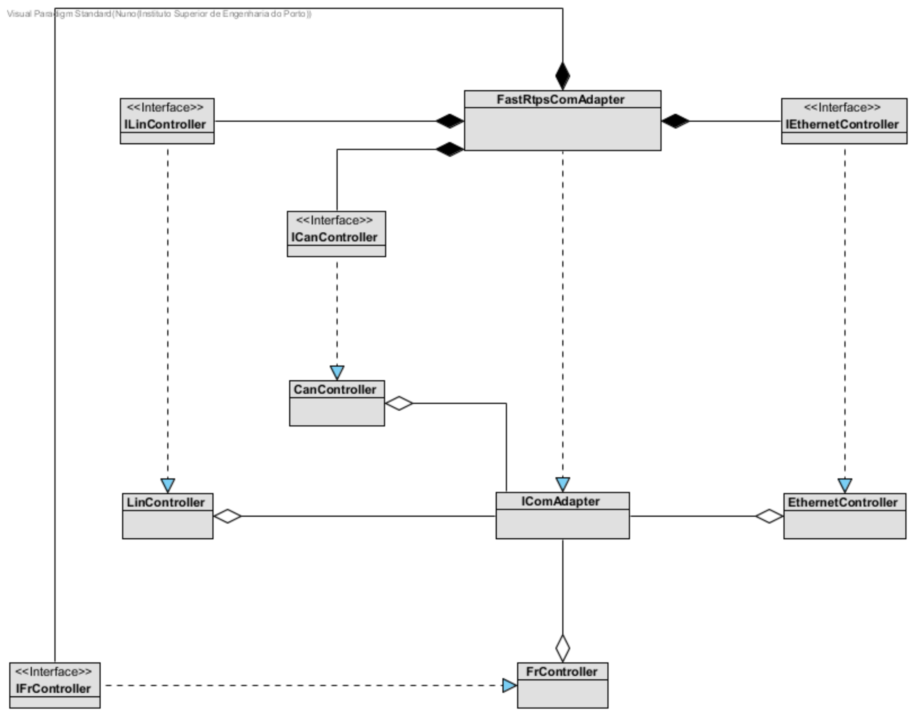


Figure 16 – Messaging Classes

Figure 16 represents the messaging class structure in the IB. As explained above, not all of these will be instantiated by the *ComAdapter*. A Participant can have several network controllers of the same type or different types, and the *ComAdapter* will create instances of the controllers defined in the configuration. Each controller represents a different communication standard and is used to simulate it.

```
struct EthMessage
{
    EthTxId transmitId;
    std::chrono::nanoseconds timestamp;
    EthFrame ethFrame;

};
```

Snippet 2 - EthMessage

Snippet 2 shows the definition of an Ethernet message *EthMessage* that's created by the *EthernetController*. The actual transmit operation of the message is first handled by the *ComAdapter* which delivers it to the *FastRtps* which in turn pushes it into the physical or virtual network stack of the underlying host. All message types work under the same operation principle, being that each specific controller instantiates the message and then sends the message to the Com Adapter for further handling.

```
/***Can Messages***/
            void SendIbMessage(EndpointAddress from, const 
sim::can::CanMessage& msg) override;
            void SendIbMessage(EndpointAddress from, const 
sim::can::CanTransmitAcknowledge& msg) override;

/***Ethernet Messages**/
            void SendIbMessage(EndpointAddress from, const 
sim::eth::EthMessage& msg) override;
            void SendIbMessage(EndpointAddress from, const 
sim::eth::EthTransmitAcknowledge& msg) override;

/***FlexRay Messages***/
            void SendIbMessage(EndpointAddress from, const 
sim::fr::FrMessage& msg) override;
            void SendIbMessage(EndpointAddress from, const 
sim::fr::FrMessageAck& msg) override;

/***Lin Messages***/
            void SendIbMessage(EndpointAddress from, const 
sim::lin::RxRequest& msg) override;
            void SendIbMessage(EndpointAddress from, const 
sim::lin::TxAcknowledge& msg) override;

/***Synchronization Messages***/
            void SendIbMessage(EndpointAddress from, const sync::Tick& msg) 
override;
            void SendIbMessage(EndpointAddress from, const sync::TickDone& 
msg) override;

/***Commands***/
            void SendIbMessage(EndpointAddress from, const 
sync::ParticipantCommand& msg) override;
            void SendIbMessage(EndpointAddress from, const 
sync::SystemCommand& msg) override;
```

Snippet 3 – ComAdapter message functions

Snippet 3 shows the prototype of some of the messages that are handled by the *ComAdapter*. As it can be seen every message must include the *EndpointAddress*, which in this case identifies the sending Participant.

```
struct EndpointAddress {
    ParticipantId participant;
    EndpointId endpoint;
    LinkName linkName;
};
```

Snippet 4 – Endpoint Address Struct

Snippet 4 shows the data associated with an endpoint address. This address is calculated from data in the configuration file. The participant id is the index of the participant in the configuration file, the endpoint id is the index of the participants network controller and the link name, is name of the link where the network controller is connected. A participant must have at least one endpoint address per link.

```
"Links": [ {
"Name": "LAN1",
"Endpoints": [
"Participant1/ETH0",
"Participant2/ETH0"
]}]
```

Snippet 5 – Configuration File Link Example

Snippet 5 shows an example of a link inside the configuration file. The link, named LAN1 has two participants each with a network controller. "Participant1/ETH0" refers to Participant1 and is network controller ETH0 while "Participant2/ETH0" refers to Participant2 and is network controller ETH0. When a configuration file like the example above is read, the IB, creates a topic named LAN1, and registers the endpoints as publishers and subscribers to it. Any message that originates with the topic LAN1 is routed to the other subscribers that are part of the topic. Even if the participant has other network controllers these will be not be able to receive any messages from the LAN1 topic unless they are a part of it. Every network controller regardless of the type must have an associated link.

4.1.3 State Handling

The IB also includes a state machine that is managed in two separate ways. It can be controlled by a test engineer that creates an instance of the *SystemController*, serving as a tool to send commands over the integration bus, or it can be managed internally by the executed application.

The *SystemController* is responsible for managing the state of each participant and only one instance of it exists within the entire co-simulation scenario, even if each of the individual simulators are physically distributed.

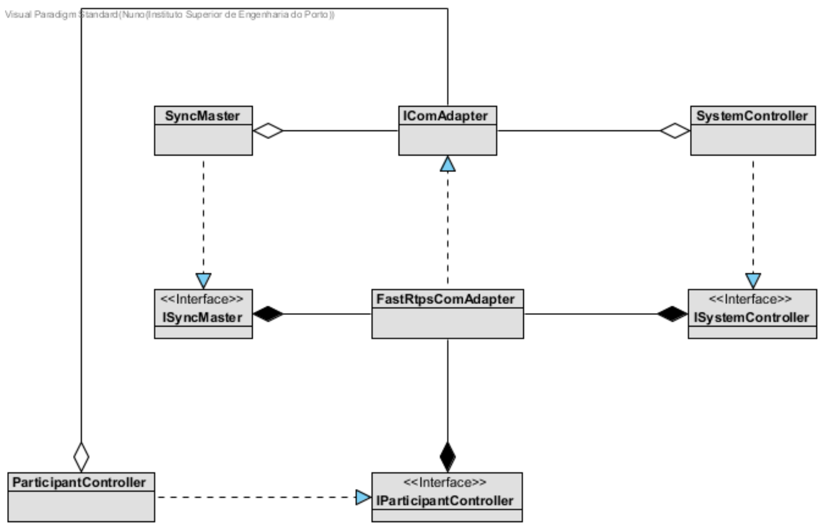


Figure 17 – State Handling Classes

Figure 17 depicts the main classes that are part of the IB state handling. The FastRtpsComAdapter does not send any commands, it acts as a message dispatcher as written above. The SystemController sends the following commands to the ParticipantController instances:

- Participant Commands
- System Commands

ParticipantCommands affect only a specific participant and are used to manage the operation of specific participants. *SystemCommands* affect every participant. These are used to start, stop, or shutdown the system simulation. The *SystemController* will manage all of the state transitions by sending commands to the *ParticipantController*, which acts upon the received messages. In its essence, the *SystemController* acts as the simulation state manager.

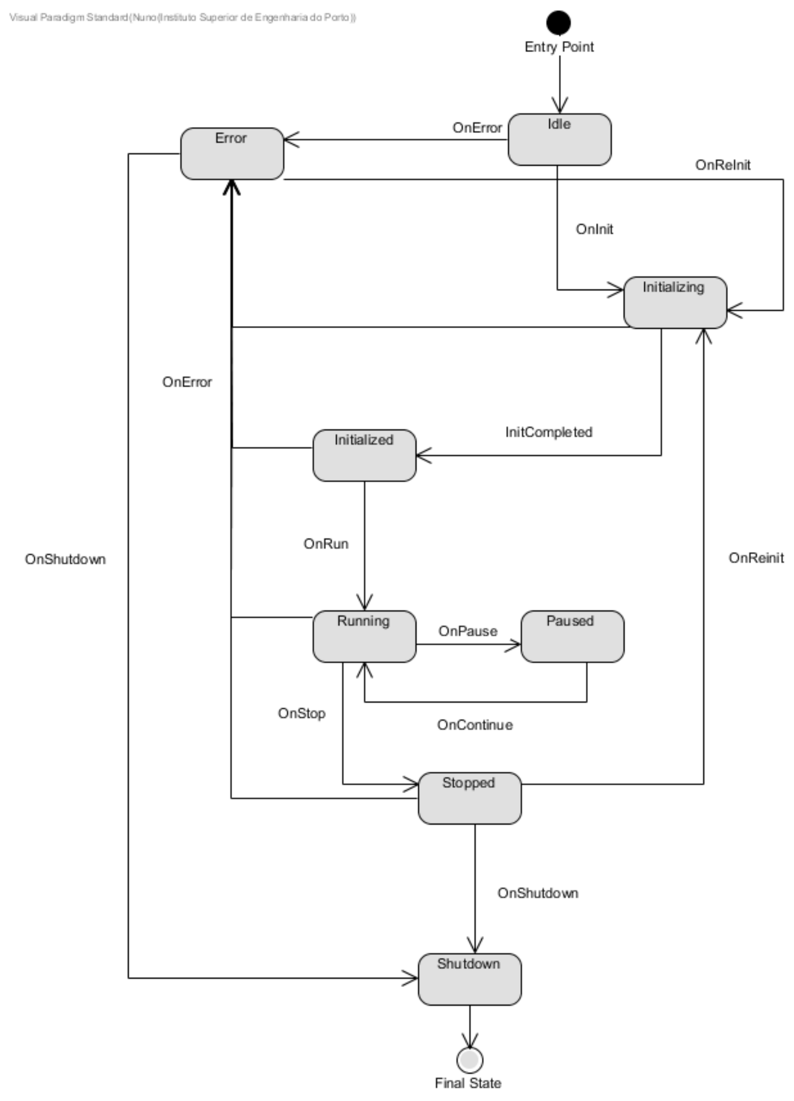


Figure 18 – Participant State Diagram

Figure 18 – Participant State Diagram depicts the State Diagram that depicts every possible state and valid state transitions. There is a total of 8 states that a Participant can be in.

- Idle – The participant and all its network controllers have been instantiated. After this, the simulation task can be set.
- Initializing – Puts the participant in an initialization state and checks if all necessary callbacks needed by the participant have been set.
- Initialized – If no errors are found the participant transitions to this state, where it waits for a Run Command.
- Running – The participant is executing the simulation.
- Paused – The participant pauses the currently running simulation.

- Stopped – The participant stops running the simulation task and does not execute any of its callbacks.
- Shutdown – The Participants exits.
- Error – Used to flag an error in the simulation process.

```
ChangeState(ParticipantState newState, std::string reason)
```

Snippet 6 – State Change

Snippet 6 – State Change shows the function call located in the *ParticipantController* class that a participant uses to change state. Note that a string containing a reason must be supplied. This is useful for when errors occur since the cause of the error can be supplied as the reason which in turn can be read on console logs.

4.1.4 Synchronization

Synchronization is kept by using one of two methods that work very similarly two each other. Both rely on the notion of a time period that a participant can run, also known as *tick time*. This time period is defined in the input configuration file. It is introduced in the simulation by the *SyncMaster* which is a Participant with the extra responsibility of synchronizing every other Participant dictating the pace of the simulation, thus acting as a centralized server.

The two synchronization methods supported by the *SyncMaster* are named *TickTick Done* and *TimeQuantum*, respectively. The major difference between them is that while the *Tick* message consists of a time period and is always accepted, the TimeQuantum message has a status field, that has three possible values, *Granted*, that indicates if the time period has been granted, meaning it has been accepted, *Invalid* in case of a runtime error like an exception being raised for an invalid numeric conversion for the time period, and the last state is the *Rejected* one, which indicates that the participant is in an invalid state to receive a grant.

Figure 8 in chapter 2.8.1.1 depicts how the TickTick Done method works. The TimeQuantum works very similarly, with the major difference that the Participant receiving the grant checks its own internal state to see if it can accept the grant and then checks the state that was sent with the TimeQuantum message. If the internal state and the message status field is of type *Accepted* and the comparison between them yields TRUE then the Participant will run for the specified time period.

4.2 QEMU Integration Bus Adapter

The QEMU Integration Bus Adapter (QIBA) is a component that is mainly used to interconnect the Integration Bus and a QEMU instance. It is responsible for the network connectivity between QEMU and IB and synchronizes QEMU with an external simulation clock.

The following chapters present the two main aspects of QIBA, the synchronization of QEMU with the external simulation clock and the network communication.

4.2.1 External Clock Synchronization

To synchronize QEMU with an external clock, QEMU must first be patched since it does not support this functionality out of the box. An already-available patch named *qqq* (Nutaro 2016) is used to provide it. The latter provides an interface for pacing the execution of QEMU allowing it to be used as a sub-system of a (larger) simulation system. It works by allowing a file descriptor to be open between the QEMU guest and the host so that an integer value can be written to it. This integer referred to as request advance corresponds to the number of microseconds QEMU is allowed run, corresponding to a progress step of its simulation time.

At the end of the time step execution, QIBA reads back from QEMU the actual simulated time value and according to the patch notes:

"*The value read will be the actual number of virtual microseconds by which QEMU has advanced its virtual clock. This will be greater than or equal to the requested advance.*" (Nutaro 2016)

QIBA then increments its simulation clock and repeat the process until the end of the simulation.

The above approach can potentially turn QEMU into a deterministic simulator if it is launched with the *icount* flag. This allows QEMU to keep track of the number of executed instructions and associate a time value to each instruction. With this, a pre-defined time-step interval always correspond to a specific number of instructions to run but given that the value read might be greater that the request one it is not fully deterministic, thus soft real time.

If QEMU is launched with the Kernel Virtual Machine (KVM) flag the execution will be faster but the soft real time properties will be lost because all emulated processor instructions will be offloaded to the host CPU.

4.2.2 Network Routing

QIBA's networking approach consist of the use of Linux Network Namespaces that are used to create an isolated network environment for each of the QEMU's instances. This is used to enforce exchange of information exclusively through the IB. Devices in different namespaces are unreachable and as such, QIBA uses virtual network interfaces mechanisms to expose the internal namespace networks to applications running on the host namespace.

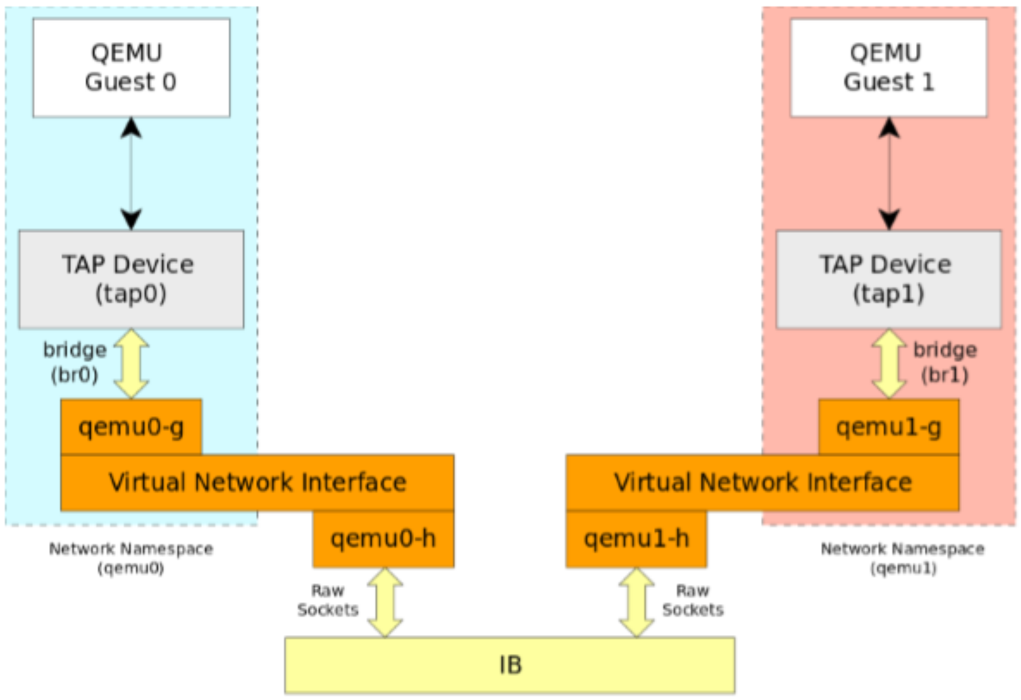


Figure 19 – QIBA Network Isolation Diagram
Courtesy of BMW and Critical TechWorks

Figure 19 illustrates the network isolation between each guest. There are two Linux network namespaces, qemu0 and qemu1. The namespaces contain a tap device which is used by QEMU's network stack to provide a connection to the host. On startup QEMU attaches itself of the tap interface which is connected to a virtual bridge to the virtual ethernet *veth* devices. The *veth* devices are virtual Ethernet devices and act as a tunnel between the network namespaces to create a bridge to a physical network device in the host.

Each pair is named after the qemuX-g and qemuX-h adapters, where X can be any number. So, each guest happens to have its network stack isolated from the host and other guests. To communicate with the host or other guests a QEMU instance must route its traffic through the qemuX-h interface, being this the interface that exposes each guest outside of its namespace.

Figure 19 also depicts the use of raw sockets to communicate with the IB. In the simulation environment all the data exchanged between the Participants needs to be routed through the IB, and to achieve this QIBA uses libPCap. It is a packet capture library that allows QIBA to capture packets from either side and inject them to the appropriate interface. It works by biding to the specified qemuX-h interface. It then captures incoming and outgoing packets and stores them in a FIFO data structure, only injecting them into the target guest when the current time step completes. So, messages are not routed immediately upon capture, but instead delivered in burst on the next time step, instead of the time step they were received in.

5 Comparison of Potential Solutions

The synchronization method implemented in the existing solution is time driven with fixed time steps. As mentioned in chapter 2.7 smaller time steps result in better accuracy but lower performance. A larger time step has the opposite effect. Given that the simulator is always stopping to re-synchronize if the systems under simulation only produce irregular events (sparse over time), then there is a lot of synchronization overhead.

Lamport proved that if two systems are not communicating then their clocks do not need to be absolutely synchronized. So, if events are sparse, the current solution is synchronizing more that it must, and this creates an overhead, because Participants are constantly interrupting their simulation task to resynchronize.

The motivation exists to explore an alternative, optimized methods. The following sub-chapters present the potentials solutions sustained on the research conducted in the chapter 2.7.

5.1 Potential Solutions

While the research for the state of the art did not (and should not) consider the architecture of the IB framework, the following potential solutions take it into account in order to converge towards Critical Techworks use-case. As such, the chosen solution is the one that fills the requirements and integrates better in the IB framework.

The current implemented methods are time driven and suffer from problems naturally associated with them, for example, when one sub-system finishes is *tick* it remains idle until it receives the next one, so in order to avoid this, and event driven approach will be used. An event in this context is defined as the sending of a network packet. The following sub-sections present three potential solutions to the problem.

5.1.1 Vector Clocks

Leslie Lamport states in his paper (Lamport 1978a) that a distributed system does not need to be synchronized with another unless they are actively exchanging messages and that time does not needed to be in absolute synchrony. Leslie presents in his paper an alternative: instead of using time as our order variable for messages we use what Leslie called *logical time*. This was introduced in chapter 2.3.1 on Lamport Clocks.

However, the work of (Lamport 1978) alone is not enough to provide a solution to the problem presented in chapter 1.2, because it only guarantees event order and does not provide any causality information. However, expanding on the work of Lamport, Vector clocks were introduced to be able to capture causality.

5.1.1.1 Applying Vector Clocks to the VV framework

A proposition to introduce vector clocks in the VV Framework is now introduced.

Each sub-system simulator holds its own set of virtual clocks, one for each network controller, with each network controller kept in a in memory map, with the map key being the *EndpointAddress* because one is already generated for each network controller, so this can be leveraged to provide some benefit.

When a message is sent in the scope of a given controller, the Participant fetches the vector clock from the map structure and increments its own position in the vector by one. When the message is received by the other Participants the clock is retrieved from the map and the vector position of the Participant that sent the message is incremented.

This solution is easy to implement and at the same time leverages the architecture of the already existing framework by extending the functionality of the existing classes and allowing to eliminate the use of a central controller by making each Participant manage its own logical time.

Vector clocks detect sources of desynchronization but do not mitigate them, meaning that it is possible to identify if the Participants that went out-of-synch with each other, but cannot resolve the issue. Has such, mitigation techniques must be implemented on the side since vector clocks do not provide them.

5.1.2 Event Based with Rollback

Event based with rollback also relies on vector clocks but provides an active mitigation technique against desynchronization by the means of rollback. What rollback does is that when a desynchronization is detected all the simulators are forced to go back in time, so, it rolls back the simulators in time. This is done by having each simulator take snapshots of its stack and heap, store them and when a rollback must occur, the simulator searches a data structure looking for the state that is less or equal to the timestamp that it must go to.

There are however a few problems with this approach. If the system under simulation is very event intensive then there may be many resynchronization events, probably resulting in worse performance when compared to the existing solution (chapter 2.8.1.1); When a rollback occurs all the computations that were done after the rollbacked timestamp are lost and also because it must take regular execution snapshots it requires huge amounts of memory.

But, main problem for this solution is the incompatibility with the IB framework. Most of the time each Participant will be running inside a QEMU instance and the IB on the host.

So, the Participant will be running on QEMU and the IB on the host and it's the Integration Bus that provides the bus and the synchronization. The IB is transparent to the Participant, since the latter is not aware that it is being synchronized or how its messages are being routed. This creates a problem with taking snapshots, the vector clocks are on the host but the Participant is on the QEMU guest, so that means that two sets of snapshots have to be taken, one for the guest and another for the host, and also, they must be paired, on snapshot of the guest must correspond to a host snapshot.

Taking two snapshots creates a problem of its own, since there are no guarantees that the snapshot pair would be taken exactly at the same time.

5.1.3 **Event Lookahead**

Event lookahead works by inference, meaning that it infers when the next event (message) is going to occur and jumps all the clocks of the simulators to that timestamp. Each simulator will look ahead to predict the minimum timestamp of its next event, compute the offset between its virtual clock and the timestamp and send the resulting offset to the Sync Master.

The Sync Master will then determine which one of the received offsets is the lowest, followed by, incrementing its current clock by adding the offset value and sending the resulting calculation to the other simulators, which will then compare their own clock timestamp with the received one. If the subtraction of their own clock timestamp and the received one results in zero, it means that the simulator can proceed with the sending of its message. If the result is different than zero, the current timestamp on the clock is substituted by the received one.

The problem with lookahead is that it may depend on variable values unknown until the event is processed and in that case it is impossible to compute the lookahead timestamp with any degree of certainty.

There is also another very important consideration: what's being simulated isn't known a-priory in the scope of the Critical Techwork's use-case. The platform is supposed to simulate a multitude of different systems, each implies the simulated sub-systems is abstract to the underlying simulation platform, making lookahead very difficult or outright impossible. (Lungeanu and Shi 1999) in their paper state this same consideration. This certainly applies to the VV framework since what the participant will be simulating is not known, the framework itself was written under the principle that the simulation task is agnostic to the framework.

Still, eventually considering a use-case in which the VV framework supported lookahead, this method provides time jumps instead of letting simulators runs freely, eventually leading to the already-mentioned sub-optimal computation usage.

5.2 Chosen Solution

Event lookahead can be discarded since it does not comply with the requirements given that what’s being simulated isn’t known. It is then therefore excluded as a solution. The remaining options are rollback and vector clocks. Both have their advantages and disadvantages: event rollback needs large amounts of memory so it can take snapshots of both the stack and the queue, but its main problem is that it needs to take snapshots not only of the IB but also of the QEMU Participant.

Given this, it would be very hard for rollback to be implemented onto the current framework, since the Participant’s simulation task is not known and the framework itself was built with the idea that each Participant can be running anything.

Considering all the points made, the chosen solution adds a new synchronization method that synchronizes the co-simulation Participants without any type of centralized controller, and it is based on Vector Clocks, which in turn are dependent on Lamport Timestamps and both are categorized as logical clocks. It is already compatible with the current framework architecture, does not use a central controller, detects concurrent messages and message losses. The only disadvantage is that is does not mitigate desynchronization, however these are implemented using function call-backs, so to allow more freedom to the end user, as he can introduce its own mitigation algorithms if the default is not enough.

6 Implementation

This chapter focuses on the design and implementation of the proposed solution. The implementation is based on Logical clocks that were introduced in (Lamport 1978) and consist of a Lamport Timestamp or Vector Clock, and are designed to allow for the global ordering of events from each of the different systems.

The technical definition of an *event* is not stated in Leslie's paper (Lamport 1978). The definition chosen for this implementation is:

- Event: any transmitted message from a simulation Participant containing a data payload.

In order to implement the proposed synchronization method, the first task was to implement the Lamport Timestamps, followed by the Vector Timestamp and lastly the Vector Clock structure.

The next few chapters detail the design considerations and the implementation of the synchronization method which is named *LogicalVectorClock* by me.

6.1 Overview

The figure bellow shows a class diagram representing the classes that make up the Vector Clock implementation. These are the main classes of the implementation and are the ones that provide the new synchronization method.

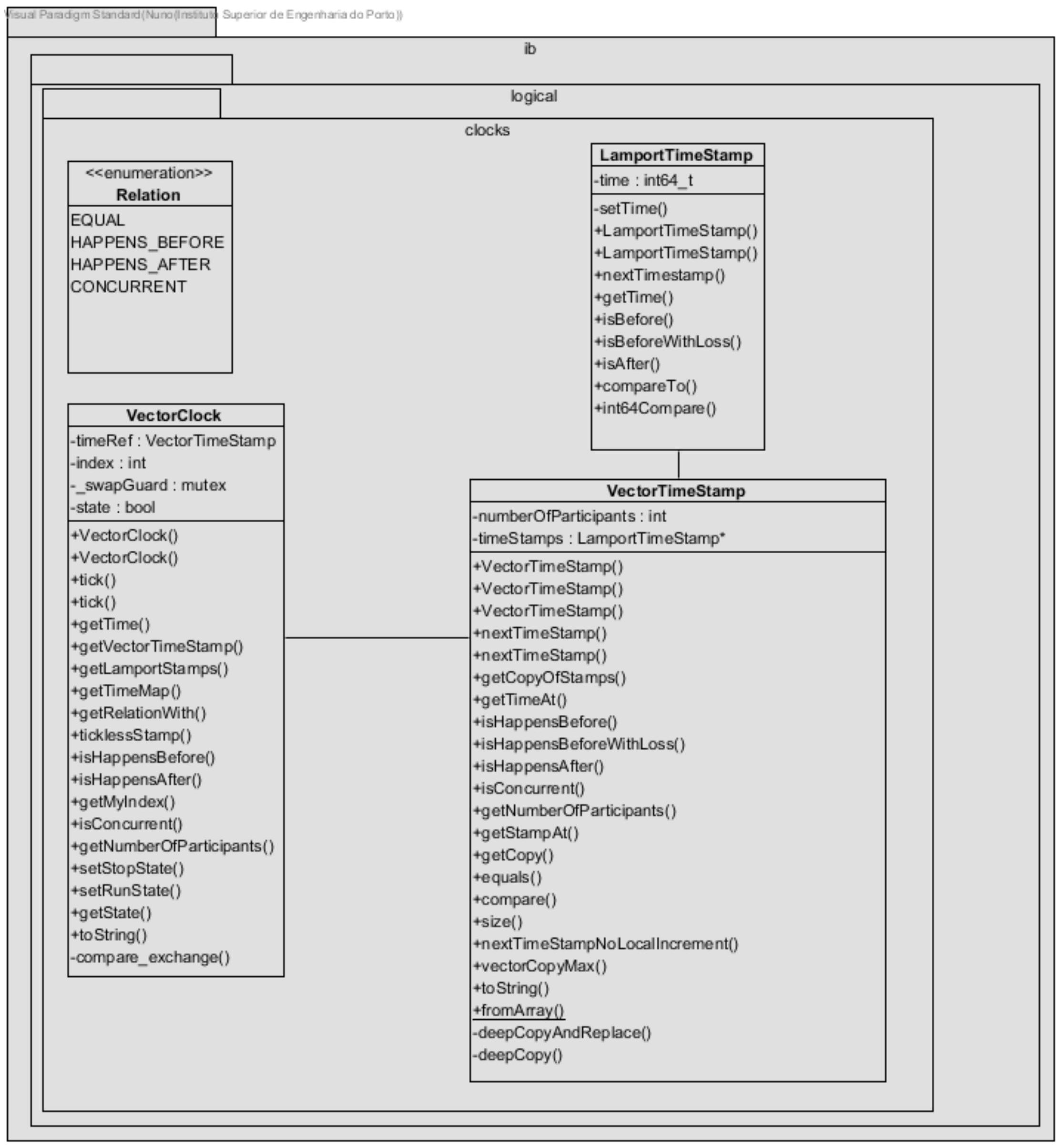


Figure 20 – Vector Clock Class Diagram

Figure 20 shows the class diagram for the Vector Clock implementation. It consists of three main classes *VectorClock*, *VectorTimeStamp* and *LamportTimeStamp*. The *LamportTimestamp* is aggregated by the *VectorTimestamp* class and the latter is aggregated by the *VectorClock* class. The *LamportTimestamp* class will be explained in chapter 6.2, the *VectorTimeStamp* will be explained in chapter 6.3 and th*e VectorClock* in chapter 6.3.1.

The enumeration *Relation* is shown with the following definitions:

- EQUAL: The local Vector Timestamp is equal to the one received, meaning that the arrays of Lamport Timestamps are the same length and each index of it contains the same value

- HAPPENS_BEFORE: The local Vector Timestamp is less than the one received, meaning that the arrays of Lamport Timestamps are the same length and at least one index of the recipients Lamport Timestamp array is less than the senders Vector Timestamp

- HAPPENS_AFTER: The local Vector Timestamp is greater than the one received, meaning that the arrays of Lamport Timestamps are the same length and at least one index of the recipients Lamport Timestamp array is greater than the senders Vector Timestamp

- CONCURRENT: The local Vector Timestamp is concurrent with the received one, meaning that given $LV_i\ RV_i\ LV_j\ RV_J$ with $LV_i\ LV_j$ being the two values of the recipients (local) Lamport Timestamp at index i and j respectively and $RV_i\ RV_j$ being the two values of the senders Lamport Timestamp (the received one) at index i and j respectively they are concurrent if $LV_i > RV_i\ \&\&\ LV_j < RV_j$ or $LV_i < RV_i\ \&\&\ LV_j > RV_j$

6.2 LamportTimestamp

The *LamportTimeStamp* class as represented in Figure 20 is the most underlying class. All other classes are dependent on its behavior. It has a single attribute, *time,* which is a 64-bit integer, and works as the monotonic clock counter variable.

The class implementation was done using a semantic that is close to functional programming: once an instance of the *LamportTimeStamp* is created it is effectively final, meaning that the counter value cannot be changed. Thus, the class is inherently thread safe. While this approach was good to assure thread safety it generated the following problem:

The main responsibility of the class is to maintain an integer counter that corresponds to the logical time that the participant's network controller is at, given that the instance is immutable this counter cannot be incremented. Instead, an operation is provided that returns a new instance.

```
LamportTimeStamp* nextTimestamp();
bool isBefore(LamportTimeStamp timeStamp);
bool isBeforeWithLoss(LamportTimeStamp timeStamp);
bool isAfter(LamportTimeStamp timeStamp);
```

Snippet 7 – LamportTimeStamp Operations

Snippet 7 shows the most important methods of the *LamportTimeStamp* class. The *nextTimestamp()*, discussed just above, returns a new instance of the class. When called, it will call the class constructor and parse as a parameter its counter value plus one and then return the newly created instance with the incremented counter value, thus maintain its thread safety and allowing the counter to be incremented.

The remaining methods are used to compare *LamportTimeStamps*. They were made self-explanatory: *isBefore()*will check if the parsed parameter counter value is less than the calling instance counter value and the *isAfter* checks if value is more than the calling instance counter value.

The *isBeforeWithLoss* is a special case method in the sense that it is used to check if the difference between the counter values is bigger than one. If this is true, it means that a loss of a number of messages equal to the difference of the counter values has occurred. This loss is determined using the following equation:

$$numberOfMessagesLost = abs(this.time - timeStamp.time) \qquad (7)$$

6.2.1 **Unit Tests**

To test the correctness of the class implementation unit tests were used to guarantee the requirements for each function.

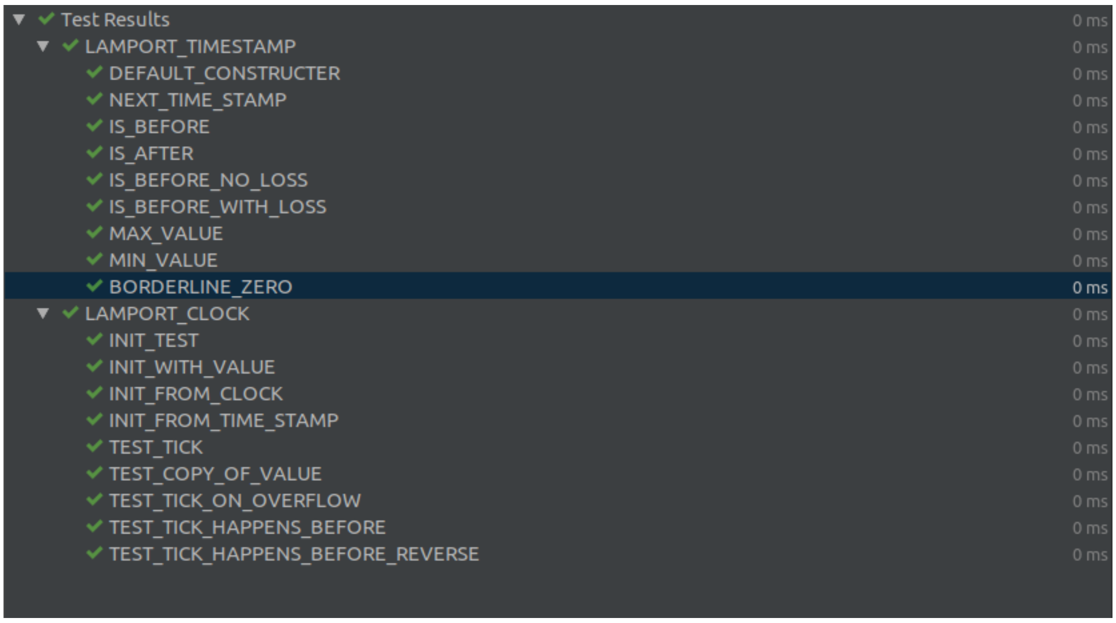

```
▼ ✓ Test Results                                   0 ms
  ▼ ✓ LAMPORT_TIMESTAMP                            0 ms
      ✓ DEFAULT_CONSTRUCTER                        0 ms
      ✓ NEXT_TIME_STAMP                            0 ms
      ✓ IS_BEFORE                                  0 ms
      ✓ IS_AFTER                                   0 ms
      ✓ IS_BEFORE_NO_LOSS                          0 ms
      ✓ IS_BEFORE_WITH_LOSS                        0 ms
      ✓ MAX_VALUE                                  0 ms
      ✓ MIN_VALUE                                  0 ms
      ✓ BORDERLINE_ZERO                            0 ms
  ▼ ✓ LAMPORT_CLOCK                                0 ms
      ✓ INIT_TEST                                  0 ms
      ✓ INIT_WITH_VALUE                            0 ms
      ✓ INIT_FROM_CLOCK                            0 ms
      ✓ INIT_FROM_TIME_STAMP                       0 ms
      ✓ TEST_TICK                                  0 ms
      ✓ TEST_COPY_OF_VALUE                         0 ms
      ✓ TEST_TICK_ON_OVERFLOW                      0 ms
      ✓ TEST_TICK_HAPPENS_BEFORE                   0 ms
      ✓ TEST_TICK_HAPPENS_BEFORE_REVERSE           0 ms
```

Figure 21 – LamportTimeStamp Unit Tests

Figure 21 shows the number and name of the tests developed to test the implementation of the *LamportTimeStamp* class. As the figure shows the tests are split in two, the *LAMPORT_TIMESTAMP* and the *LAMPORT_CLOCK*. The reason for this is, the former tests only the class implementation in isolation meaning it consist of only unit tests, while the latter tests its integration with a mock class named *LogicalClock*.

The *LamportTimeStamp* was designed work with its instance variable aggregated by another class, meaning that once an instance of it is created all its internal properties are final and cannot be changed, thus it's an immutable class and so every time that a counter update occurs the actual counter is not updated, but instead a new copy is created with an incremented counter value, thus the reason why it need to be aggregated by another class, the current instance variable must be replaced by the copy on the class aggregating the instance. Naturally, in the clock implementation the aggregator class is the *VectorTimeStamp*, but for testing purposes a *LogicalClock* class was introduced to provide similar functionally.

```
TEST(LAMPORT_CLOCK, TEST_TICK_HAPPENS_BEFORE){
    using namespace ::ib::logical_clocks;
    LamportTimeStamp happensBefore = LamportTimeStamp(10);
    LamportTimeStamp initStamp = LamportTimeStamp(100);
    LogicalClock clock = LogicalClock(initStamp);
    LamportTimeStamp stamp = clock.tick(happensBefore); //increment by
1
    ASSERT_TRUE(stamp.isAfter(happensBefore));
    ASSERT_TRUE(happensBefore.isBefore(stamp));
    ASSERT_TRUE(stamp.isAfter(initStamp));
    ASSERT_TRUE(initStamp.isBefore(stamp));
}
```

Snippet 8 – LamportTimeStamp Integration Test Example

Note the use of the class *LogicalClock* in Snippet 8. This is the mock class that was just discussed above. An instance of *LamportTimeStamp* class can be created but it effectively final to provide thread safety, this means that none of its attribute values can be changed as written before. So, a mock class had to be introduced so some tests could be conducted.

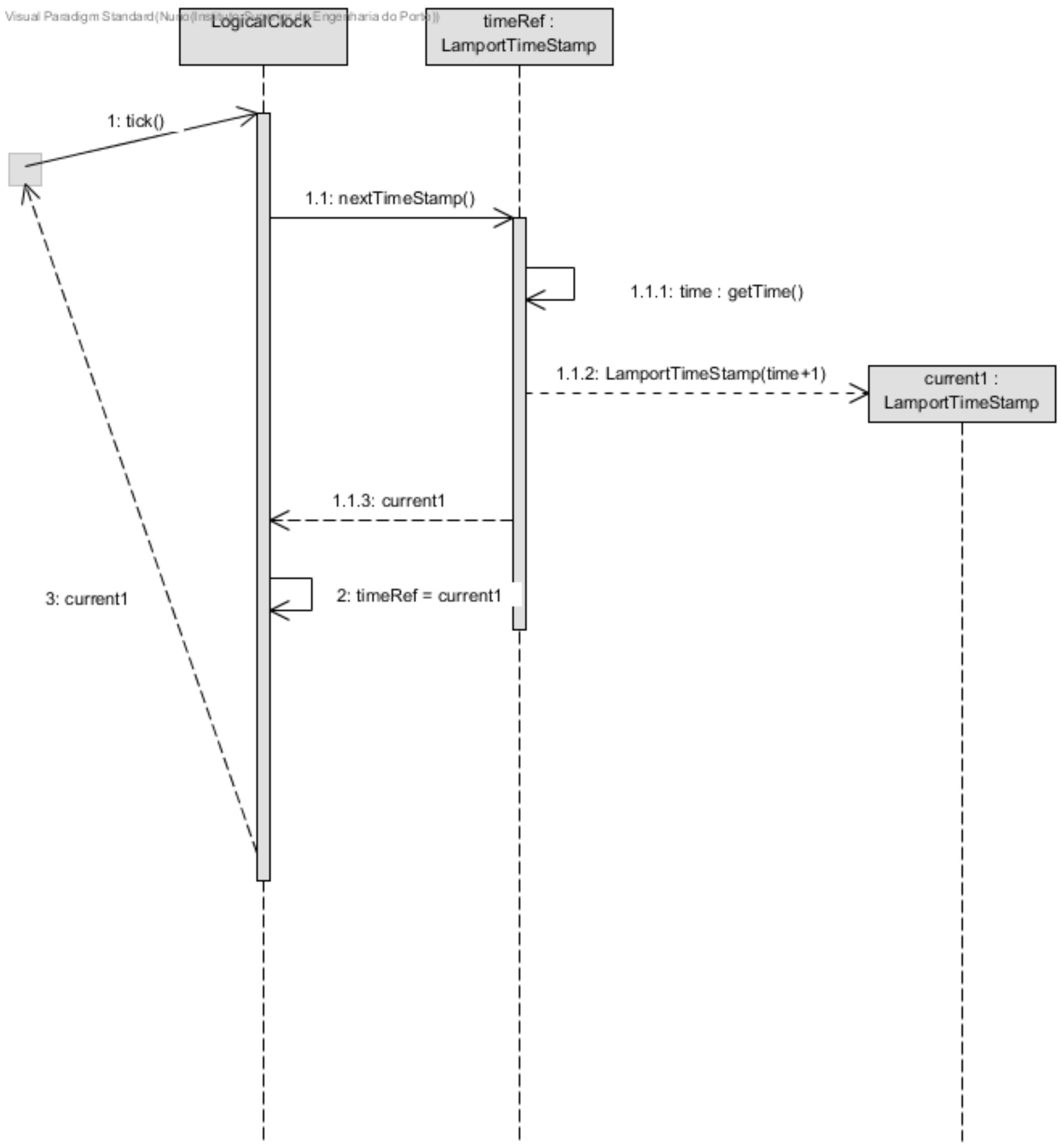

Visual Paradigm Standard(Nuno(Instituto Superior de Engenharia do Porto))
LogicalClock
timeRef : LamportTimeStamp
1: tick()
1.1: nextTimeStamp()
1.1.1: time : getTime()
1.1.2: LamportTimeStamp(time+1)
current1 : LamportTimeStamp
1.1.3: current1
2: timeRef = current1
3: current1

Figure 22 – LogicalClock Tick System Sequence Diagram

The Figure 22 depicts a sequence diagram demonstrating the test shown in Snippet 8. It shows that a new LamportTimeStamp instance is created when the *tick* function is called which is turn calls the *nextTimeStamp* function. The latter is the function that creates a new *LamportTimeStamp* with the counter increased. After this, the new instance is returned and substitutes the old in the *LogicalClock* class.

6.3 VectorTimestamp

This class was designed to encapsulate and manage an array of *LamportTimeStamp.* It provides an interface so the underlying timestamps can be managed. It manages every instance and is very similar to the mock class *LogicalClock* mentioned above, with the difference that the mock class only managed one instance of a *LamportTimeStamp*.

Both classes have the *TimeStamp* in their name, however the actual time stamp is the *LamportTimeStamp* which holds the counter. *VectorTimeStamp* acts more as a manager for the several instances, but it is also a time stamp. The difference is, that while a Lamport time stamp can be looked at by just comparing two single integers, in a Vector time stamp the entire array of integers must be compared before any conclusion can be taken.

The next two arrays provide an example array of a *VectorTimeStamp timeRef* attribute. This attribute is a pointer array of *LamportTimeStamp*, where each Lamport stamp is associated with a Participant and the counter represents the events that the Participant has sent.

$$[\ 0, 1,5, 3\ ] \tag{8}$$

$$[\ 0, 2, 5, 3\ ] \tag{9}$$

The above arrays show two arrays of randomly chosen integers where each array is a *VectorTimeStamp* and each index of the array is a *LamportTimeStamp.*

Leslie Lamport defined a relationship for his Lamport Time Stamps called happens before (already covered in chapter 2.3.1). Given that definition, if one would look at $index_0$ of both arrays (8) and (9) we would say that they are equal. However, considering $index_1$ they are not. In fact, according to the definition of the relationship array (8) is said to occur before (9). This is a problem because some indexes are equal while some are different.

Vector Time Stamps tackle this problem by looking at the whole array instead of each individual index. For example, the Lamport time stamps at $index_0$ of equation's (8) and (9) are equal, but the same is not true for $index_1$, so the relation is said to be (9) happens after (8) and (8) happens before (9), depending on which one is compared to what.

The *VectorTimeStamp* was designed to handle this exact situation, to verify the relation between the received vector and the vector of the Participant.

```
Relation VectorTimeStamp::compare(VectorTimeStamp stamp) {
    if (this->numberOfParticipants != stamp.numberOfParticipants) {
        throw std::invalid_argument("Vectors do not match in length");
    }
    Relation rel = Relation::EQUAL;
    for (int i = 0; i < stamp.numberOfParticipants; i++) {
        if (this->timeStamps[i].isBefore(stamp.timeStamps[i])) {
            if (rel == Relation::HAPPENS_AFTER) {
                return Relation::CONCURRENT;
            }
            rel = Relation::HAPPENS_BEFORE;
        } else if (this->timeStamps[i].isAfter(stamp.timeStamps[i])) {
            if (rel == Relation::HAPPENS_BEFORE) {
                return Relation::CONCURRENT;
            }
            rel = Relation::HAPPENS_AFTER;
        }
    }
    return rel;
}
```

Snippet 9 – Comparison Function

Snippet 9 shows the C++ code used to determine the relation between two *VectorTimeStamp*. It shows that the two vectors must be equal in size and exhibits the relations that were previously discussed namely, *equal*, *happens before* and *happens after*.

A relationship was introduced, named concurrent; this relation happens when one of the indexes is 'in the future' and the other is 'in the past' with respect to each other; The following arrays show an example of this type of relation.

$$[\, 0, 5, 9, 15 \,] \tag{10}$$

$$[\, 0, 3, 9, 20 \,] \tag{11}$$

Comparing either (10) with (11) or (11) with (10) is irrelevant. The result returned by the function depicted in Snippet 9 would be the same, i.e. concurrent. This is because one of the indexes is ahead while the other is behind regarding logical time.

In other words, when comparing those two arrays they are both in the past and future, so this is said to be a concurrent relation. The arrays are only an example, it's not something that supposed to happen; at least not in this implementation, because the IB broadcasts messages, so a difference so large as the one at $index_3$ should not happen. If it does, it indicates a packet loss as opposed to a concurrent relation, this will be discussed in a further chapter where the mitigation method will also be presented.

While the above two arrays are in fact in a concurrent relation as per Leslie's paper, they are not the typical concurrent relation in this thesis. A typical concurrent is caught much earlier, as defined by the following condition:

$$abs(V1_n - V2_n) = 1 \;\&\&\; abs(V1_i - V2_i) = 1 \tag{12}$$

Where $V1$ and $V2$ are both vector timestamps, n and i are the vector indexes as shown the difference at two separate indexes is one. This is typical the case has a bigger the difference would indicate not a concurrent relation but a message/packet loss.

Still, there is one crucial detail missing: what does exactly a concurrent relation show? Well, this can be better demonstrated by the following two arrays:

$$[\,0, 1\,] \tag{13}$$

$$[\,1, 0\,] \tag{14}$$

These two arrays exemplify two Participants that are sending a message at the exact same logical time. When one of the Participants receives the message, it will compare its clock against its own Vector clock and according to the example, determine that the relation is concurrent.

This means that a *happens before* relation cannot be inferred and so it is not possible to infer which event was issued first, (13) or (14). A strategy to handle concurrent messages was devised and is introduced in Chapter 6.4.

In summary, the *VectorTimeStamp* class is responsible for checking the relation between an instance of its class and another *VectorTimeStamp*. It is also responsible for managing the underlying *LamportTimeStamps*. While it manages all the stamps it is only allowed to increment one of them, the one that corresponds to the index of its corresponding Participant. So, if a hypothetical Participant 1 has an index 0, it can only increment that index. Consequently, its *nextTimeStamp* function will only apply to that index, leaving the rest unchanged.

When a *VectorTimeStamp* is received as part of an event/message from another Participant, the local *VectorTimeStamp* will evaluate the relation between both Vector Clocks. If a *happens before* is inferred with the calculated difference at the senders index equal to one it will substitute its internal *LamportTimeStamps* for the received ones.

6.3.1 **VectorClock**

The *VectorClock* is the highest-level class of those discussed. It delegates most of its functionality to the *VectorTimeStamp* proving only the high-level API and concurrency handling. It provides the latter to ensure that operations on the *VectorTimeStamp* are atomic.

```
std::unordered_map<std::string,std::shared_ptr<ib::logical::clocks::Ve
ctorClock>> controllerClocks;
```

Snippet 10 – Vector Clock Map

A Participant can have multiple Vector Clocks as Snippet 10 indicates. Each is saved in a map data structure where the key is the network *link* name that is provided in the JSON configuration file. Each clock is responsible to keep time only for the network that it has been assigned to. For example, a clock for link *LAN1* will only keep time for that network.

When a message is received by a network controller the synchronization method is checked. If it is of type *LogicalVectorClock* the message will be sent to the Participant controller for further processing. This message passing is done by using an Observer pattern, where the Participant controller is the Observer and the network controller the observable.

When a message arrives at a network controller, the Participant controller is notified through its update method as shown in snippet 11.

```
ib::mw::sync::LogicalClockSyncTypeAction
ParticipantController::update(const ib::mw::EndpointAddress &from,
const ib::sim::eth::EthMessage &msg) {
    return this->ParseMsg(msg.stamp, msg.stamp.size(), from);
}
```

Snippet 11 –Participant Controller Observer Update

Snippet 11 shows that the message stamp which is a *VectorTimeStamp* from the Participant that sent the message, is parsed to the *ParseMsg* method along with the message size and the endpoint address.

```
LogicalClockSyncTypeAction
ParticipantController::ParseMsg(std::vector<int64_t> msgStamp, int
size,
                                                             const
ib::mw::EndpointAddress &from) {
    if (this->controllerClocks.count(from.linkName) == 1) {
        try {
            logical::clocks::VectorTimeStamp stamp =
logical::clocks::VectorTimeStamp::fromArray(msgStamp,

size);
            auto clock = this->controllerClocks[from.linkName];
            return this->HandleMessage(stamp, clock, from);
        }
        catch (std::exception e) {

        }
    } else {
        return LogicalClockSyncTypeAction::Process_Message;
    }
}
```

Snippet 12 – Vector Clock Retrieval

Snippet 12 shows how the vector clock is retrieved from the array. It shown that the link name is used to retrieve it from the map. After that, the received *VectorTimeStamp,* clock and the endpoint address are parsed to the *HandleMessage* method for further processing.

```
LogicalClockSyncTypeAction
ParticipantController::HandleMessage(logical::clocks::VectorTimeStamp
stamp,

std::shared_ptr<logical::clocks::VectorClock> clock,
                                                                const
ib::mw::EndpointAddress &from) {

    auto currentStamps = clock->getVectorTimeStamp();
    if(currentStamps.isHappensAfter(stamp)){
        return ib::mw::sync::Process_Message;
    }
    else if (currentStamps.isHappensBeforeWithLoss(stamp)) {
        auto reply = this->ExecutePacketLossHandler(stamp,
currentStamps, from, _endpointAddress,
                                                    clock->getState(),
clock->getMyIndex());
        if (reply == Process_Message_Increment_Clock) {
            IncrementClock(stamp, clock);
            return ib::mw::sync::Process_Message;
        } else if (reply == Discard_Message_Increment_Clock) {
            IncrementClock(stamp, clock);
            return reply;
        }
        return reply;
    } else if (currentStamps.isConcurrent(stamp)) {
        auto reply = this->ExecuteConcurrentRelationHandler(stamp,
currentStamps, from, _endpointAddress,
                                                            clock-
>getState(), clock->getMyIndex());
        if (reply == Process_Message_Increment_Clock) {
            IncrementClock(stamp, clock);
            return ib::mw::sync::Process_Message;
        } else if (reply == Discard_Message_Increment_Clock) {
            IncrementClock(stamp, clock);
            return reply;
        }
        return reply;
    } else if (stamp.isHappensAfter(currentStamps)) {
        IncrementClock(stamp, clock);
        return ib::mw::sync::Process_Message;
    }
    return ib::mw::sync::Process_Message;
}
```

Snippet 13 – Clock Handling

Snippet 13 shows how the received VectorTimeStamp is processed and that the received vector time stamp is compared against the current stamp and depending on the relation between the two, certain actions can happen.

Certain handlers are called when the order of events is not the correct one. For example, if the current stamps are behind with a loss, meaning that an index of the current stamp where the difference is bigger than one when compared to the same index of the received VectorTimeStamp: when this happens the packet loss handler is called. If the relation between them is deemed to be concurrent then the concurrent handler is executed.

Whatever the handler that it is executed, all of them will return a reply of type *LogicalClockSyncTypeAction.* This indicates the action that the *handleMessage* function should do. There are 4 types of actions in total:

- Process_Message – It is used to indicate that the message should be processed, however the Vector Clock is not incremented. Normally this type of action is used to signal the network controller that received the message that it should process it. In short, the *handleMessage* function will not do anything and will just instruct the network controller to continue and process the message.

- Discard_Message – It is used to indicate that the message should be discarded. As in the message above, it is used to signal the network controller. It tells it to not process the received message, thus, ignoring the contents of the message and any affect that the message might have on the Participant.

- Process_Message_Increment_Clock – It is used to indicate that the message should be processed, but with this type of action the clock is also increment. The *handleMessage* function will increment the clock and then reply with the *Process_Message* to instruct the network to process it.

- Discard_Message_Increment_Clock – It is used to indicate that the message should be discard but the clock should still be incremented. It has the same flow as the action above, but the *handleMessage* function will tell the network controller to discard the message.

All the actions above are returned from the concurrent or packet loss handlers to the *handleMessage* function of the *ParticipantController* which in turn replies with either *Process_Message or Discard_Message* to the network controller.

If no handler is executed, an action is still returned to the network controller. Since no handler is executed, this means that the order of events is correct, thus, and the recipient's clock will be incremented and the *Process_Message* will be sent to the network controller. This is the default flow and unlike the handlers that are implemented as callbacks, cannot be changed.

The handlers are implemented by the test engineer responsible for building the simulation setup, thus, achieving a higher degree of flexibility as proposed in 5.1.1.1.

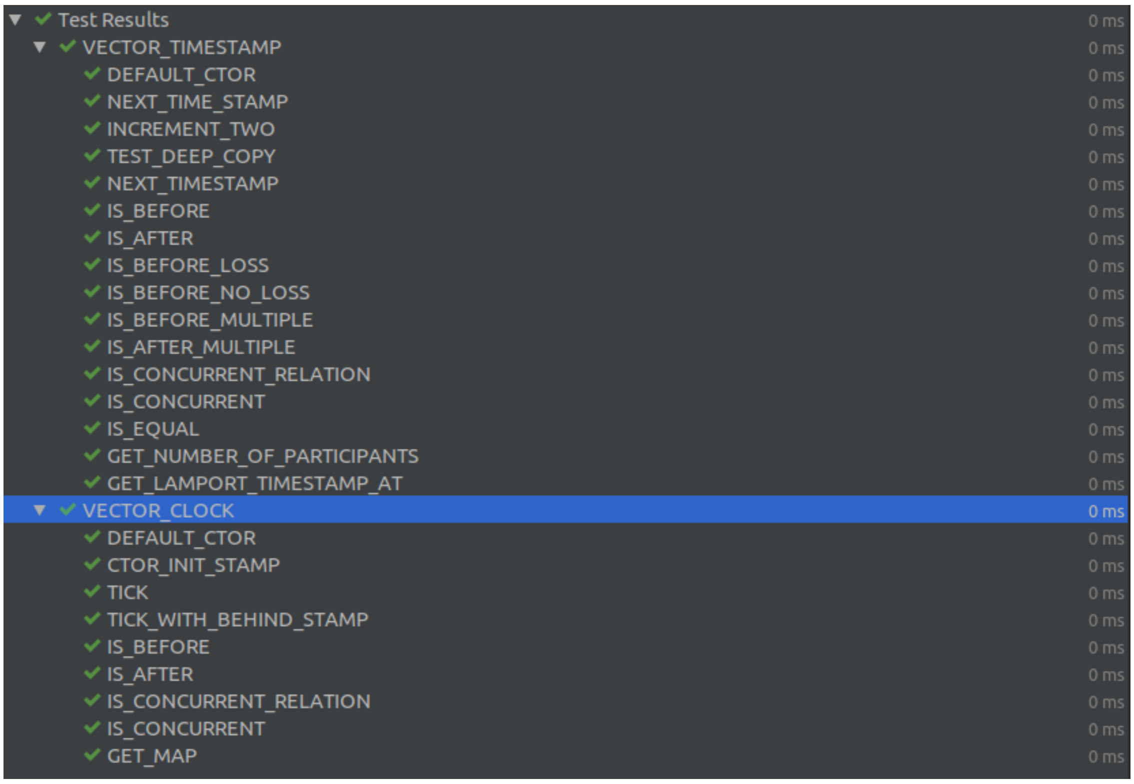
```
pCtrl-
>setPacketLossHandler([&ethMsgCtrlMap](ib::logical::clocks::VectorTime
Stamp receivedStamps, ib::logical::clocks::VectorTimeStamp
currentStamps,
ib::mw::EndpointAddress from, ib::mw::EndpointAddress to, bool state,
int myIndex) -> ib::mw::sync::LogicalClockSyncTypeAction {
    utils::pout(PINFO,_caller) << "Packet Loss Detected";
    for(int i =0;i<receivedStamps.size();i++){
        auto res = abs(receivedStamps.getTimeAt(i)-
currentStamps.getTimeAt(i));
        if(res>=2){
            ethMsgCtrlMap["ETH0"].ethCtrlPtr->SendSigStop();
            utils::pout(PINFO,_caller) << "SigStop Sent";
            for(int
j=currentStamps.getTimeAt(i)+1;j<=receivedStamps.getTimeAt(i);j++){
                auto msg = ib::mw::sync::RetransmitMessageRequest();
                msg.participantIndex = i;
                msg.vectorValueAtIndex = j;
                utils::pout(PINFO,_caller) << "Sending Retransmit
Request";
                ethMsgCtrlMap["ETH0"].ethCtrlPtr-
>SendRetransmitRequest(msg);
                utils::pout(PINFO,_caller) << "Sent Retransmit
Request";
                std::this_thread::sleep_for(1s);
            }
            utils::pout(PINFO,_caller) << "SigCont Sent";
            ethMsgCtrlMap["ETH0"].ethCtrlPtr->SendSigCont();
            return ib::mw::sync::Discard_Message_Increment_Clock;
        }
    }
    return ib::mw::sync::Discard_Message;
});
```

Snippet 14 – Packet Loss Handler Callback

Snippet 14 shows the packet loss handler implemented for the system tests and it is shown here to demonstrate the flexibility previously discussed. The logic of the shown implementation is to implement behaviour to make possible to recover packets. Although this was implementation for the system tests it is also the default implementation provided. This implementation is described in depth in 6.4.1.

6.3.2 **Unit Tests**

To test the implementation several tests were implemented. A part of the system tests discussed, there were also unit and integration tests.



```
▼ ✓ Test Results                         0 ms
  ▼ ✓ VECTOR_TIMESTAMP                   0 ms
      ✓ DEFAULT_CTOR                     0 ms
      ✓ NEXT_TIME_STAMP                  0 ms
      ✓ INCREMENT_TWO                    0 ms
      ✓ TEST_DEEP_COPY                   0 ms
      ✓ NEXT_TIMESTAMP                   0 ms
      ✓ IS_BEFORE                        0 ms
      ✓ IS_AFTER                         0 ms
      ✓ IS_BEFORE_LOSS                   0 ms
      ✓ IS_BEFORE_NO_LOSS                0 ms
      ✓ IS_BEFORE_MULTIPLE               0 ms
      ✓ IS_AFTER_MULTIPLE                0 ms
      ✓ IS_CONCURRENT_RELATION           0 ms
      ✓ IS_CONCURRENT                    0 ms
      ✓ IS_EQUAL                         0 ms
      ✓ GET_NUMBER_OF_PARTICIPANTS       0 ms
      ✓ GET_LAMPORT_TIMESTAMP_AT         0 ms
  ▼ ✓ VECTOR_CLOCK                       0 ms
      ✓ DEFAULT_CTOR                     0 ms
      ✓ CTOR_INIT_STAMP                  0 ms
      ✓ TICK                             0 ms
      ✓ TICK_WITH_BEHIND_STAMP           0 ms
      ✓ IS_BEFORE                        0 ms
      ✓ IS_AFTER                         0 ms
      ✓ IS_CONCURRENT_RELATION           0 ms
      ✓ IS_CONCURRENT                    0 ms
      ✓ GET_MAP                          0 ms
```

Figure 23 – Vector Clock Unit and Integration Tests

Figure 23 shows the unit and integration tests implemented to test the implementation of the Vector Clock class. All of them are used to ensure that the implementation of the class is according to specification, meaning that it works as the Vector Clock described in (Fidge 1988).

6.4 Mitigation Strategies

The following chapter presents the mitigation strategies implemented to handle the problems described with the Vector Clock proposal described in chapter 5.1.1.1. These include handling event concurrency, which happens when two or more systems are sending events at the same logical time. As explained, this does not mean at the exact same time, only that the Vector

Clock checks the receives a timestamp and determines that, one or more timestamps are in the “past” and others in the “future”. It is also presented the packet loss recovery strategy.

6.4.1 Packet Loss Recovery

When a Participant sends an event it is cached in a circular buffer. The circular buffer has a default capacity of 100 messages. This number can be increased by the test engineers.

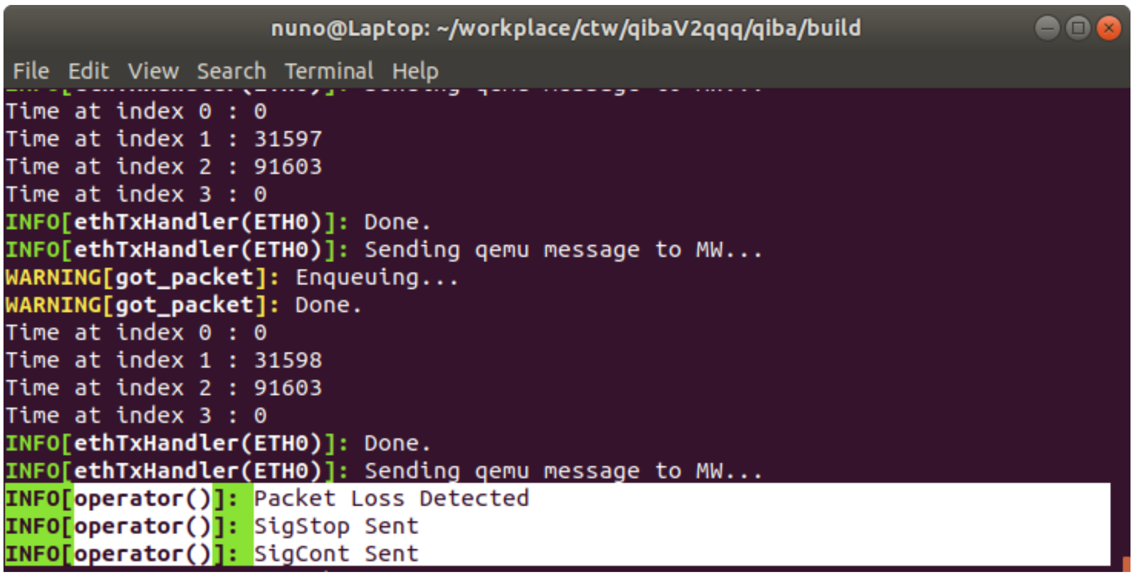
When a Participant receives an event and detects that it is missing several packets it will execute the packet loss handler. When started, the handler first broadcasts a **SIGSTOP** to every Participant. Consequently, each of the **SIGSTOP** recipients will stop executing, but the IB instance will continue to run, meaning that each of the stopped Participants can still reply to certain messages.

After the **SIGSTOP** is sent, the Participant with the missing packets will broadcast a *Retransmission* request message, which is composed by a *VectorTimeStamp* and an integer. The *VectorTimeStamp* indicates which message the Participant is requesting. This is necessary because, in the whole co-simulation scenario every message has a unique *VectorTimeStamp*, so the only way for two messages to have the same time stamp if when a concurrent relation happens, but this is mitigated immediately has shown above.

The integer is the participant vector clock index that the message is meant for. Since the IB only works by broadcast it must identify which participant does it want the message from. This integer will always be equal to the ID of the Participant that sent the message that trigger a packet loss detection on the recipient Participant.

When a Retransmission request is received the recipient will check if its ID is equal to the specified one, if it is, it will check if it has the request message in its cache, sending it to the requester if it does, if it does not, it will sent a *MessageNotFound* reply to the sender.

For each missing message, the Participant that is executing the packet loss handler will continue to send *Retransmission* request messages, one per missing message. When all the requests have been sent, and either the missing events or a not found reply are received, the Participant will increment its clock to match the one that gave rise to the packet loss handler being executed. Note, that each of the missing messages that are received are processed in the order that should have been processed if the loss had never occurred. After this process finishes a **SIGCONT** is sent and the co-simulation is resumed.



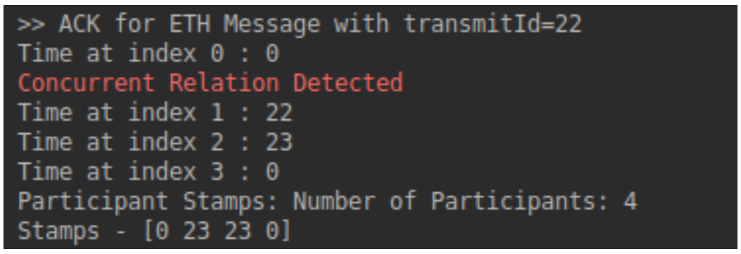
```
nuno@Laptop: ~/workplace/ctw/qibaV2qqq/qiba/build
File  Edit  View  Search  Terminal  Help
Time at index 0 : 0
Time at index 1 : 31597
Time at index 2 : 91603
Time at index 3 : 0
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
WARNING[got_packet]: Enqueuing...
WARNING[got_packet]: Done.
Time at index 0 : 0
Time at index 1 : 31598
Time at index 2 : 91603
Time at index 3 : 0
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[operator()]: Packet Loss Detected
INFO[operator()]: SigStop Sent
INFO[operator()]: SigCont Sent
```

Figure 24 – SigStop and SigCont Console Log

Figure 24 shows the console log for a Participant for the process described above. In it, a packet loss was detected, a SIGSTOP was sent and then a SIGCONT. It does not show the *Retransmission* requests since those are not logged.

6.4.2 **Concurrent messages**

The definition of a concurrent message was already presented in chapter 6.3 without introducing the mitigation technique. When a concurrent relation happens, this will be detected by one or more Participants.



```
>> ACK for ETH Message with transmitId=22
Time at index 0 : 0
Concurrent Relation Detected
Time at index 1 : 22
Time at index 2 : 23
Time at index 3 : 0
Participant Stamps: Number of Participants: 4
Stamps - [0 23 23 0]
```

Figure 25 – Concurrent Relation Console Log

Figure 25 shows the console log of a Participant that detected a concurrent relation. When this happens the concurrent relation, callback is executed. It has the same the degree of flexibility as the packet loss callback described in Snippet 14.

The default implementation will simply tell the Participant to apply the action *Discard_Message,* since it is enough to fulfil the requirements. When the latter action is returned by the callback, the *handleMessage* function will not increment the clock, thus creating a difference bigger that one on one of the indexes of the vector, thus the packet loss callback will be called when the next event is received. Because of this, the order of events will eventually be restored, since the default implementation of the packet loss handler guarantees that the events are sent and processed by the order they were originally meant to.

This seems a bit of a contradiction since a concurrent relation implies that the events are to be processed at the same time, which can have nefarious effects as discussed along this thesis. But a small trade off does exists in the default implementation, since it ignores the messages of the Participants, it relies on the packet loss handler to restore the order, but the first Participant to have the correct order will be the one that receives the next event first, which might not be what was intended. This is the reason that the implementation for the mitigation strategies was done through callbacks, this way if the default is not enough, the test engineer can supply is own implementation.

6.5 Integration of the Proposed Solution with IB

The new synchronization method called *LogicalVectorClock* had to be integrated onto the IB framework. First there was the implementation of the Vector Clock was described above, then there was a need to integrate the Vector clock onto the existing classes. There were several classes that had to be extended, the network controllers and the Participant Controller. A strategy to instantiate the new method had to be implemented.

The application of the new strategy is selected by the synchronization method provided in the configuration file. If the synchronization method is not of the type *LogicalVectorClock* there is no need to instantiate the vector clock, otherwise the vector clock must be created and shared between the network controllers and the Participant Controller.

Since it is the Participant Controller that knows what type of synchronization is in use and because the network controller must increment the clock when it sends a message, the clock must be injected onto the network controller, but first the Participant Controller must be checked to see what synchronization method is being used. To achieve this, a visitor pattern was used.

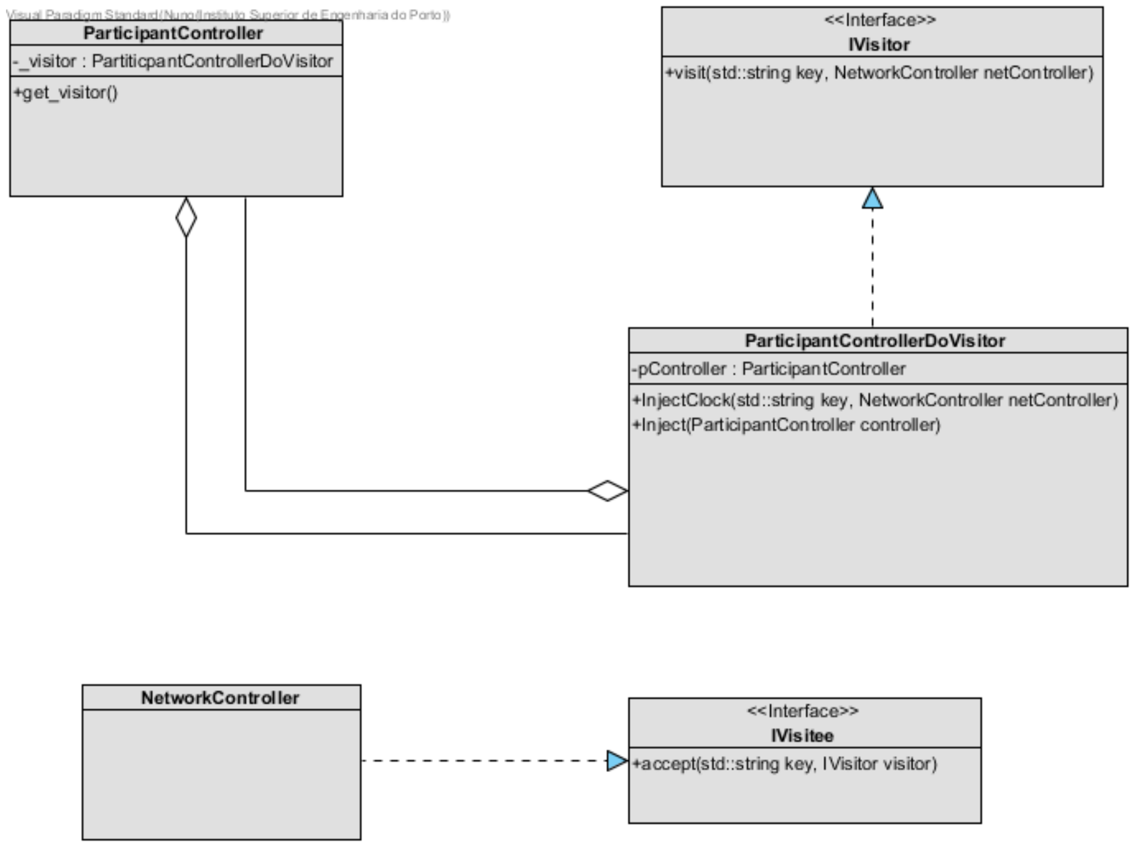

Visual Paradigm Standard(Nuno(Instituto Superior de Engenharia do Porto))
ParticipantController
-_visitor : PartiticpantControllerDoVisitor
+get_visitor()
<<Interface>>
IVisitor
+visit(std::string key, NetworkController netController)
ParticipantControllerDoVisitor
-pController : ParticipantController
+InjectClock(std::string key, NetworkController netController)
+Inject(ParticipantController controller)
NetworkController
<<Interface>>
IVisitee
+accept(std::string key, IVisitor visitor)

Figure 26 – Visitor Classes

Figure 26 shows the classes implemented to achieve what was described above. Create the Vector Clock and inject it onto the network Controller if the type of synchronization method is *Logical Clock*. The figure shows two interfaces, these are a part of the Visitor-Visitee pattern, that allows for a degree of indirection between the Participant and network controllers, thus making the design more loosely coupled. The class *ParticipantControllerDoVisitor* acts as an indirection layer to allow for the separation.

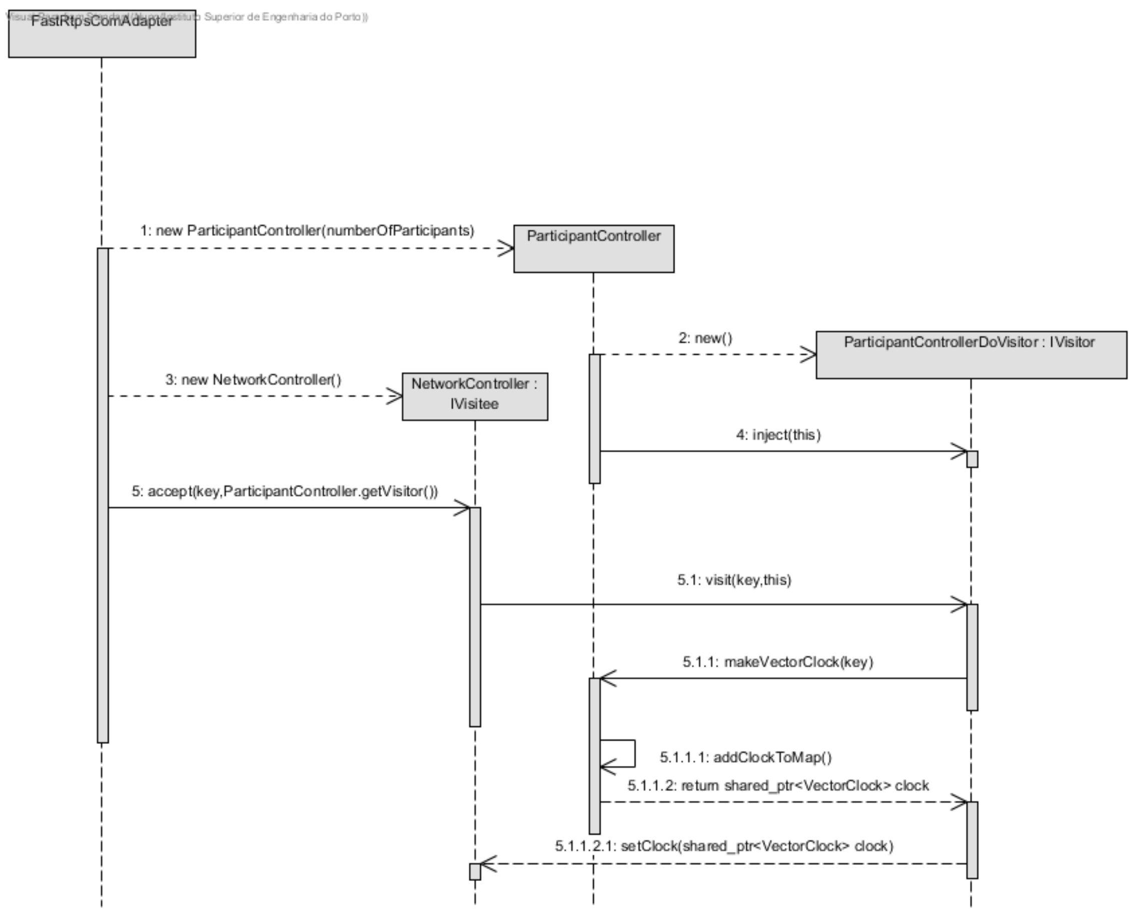

FastRtpsComAdapter
1: new ParticipantController(numberOfParticipants)
ParticipantController
2: new()
ParticipantControllerDoVisitor : IVisitor
3: new NetworkController()
NetworkController : IVisitee
4: inject(this)
5: accept(key,ParticipantController.getVisitor())
5.1: visit(key,this)
5.1.1: makeVectorClock(key)
5.1.1.1: addClockToMap()
5.1.1.2: return shared_ptr<VectorClock> clock
5.1.1.2.1: setClock(shared_ptr<VectorClock> clock)

Figure 27 - Visitor System Sequence Diagram

Figure 27 shows the system sequence diagram for the implementation of the Visitor-Visitee pattern. The *FastRtpsComAdapter* is responsible for the creation of both the network and the Participant controllers. When a network controller is created, the *accept* function, which in turn calls the *visit* method of the *ParticipantControllerDoVisitor*, which contains the instance of the Participant controller. This instance is used to make the Vector Clock by calling the method *makeVectorClock*. This method check if the synchronization method is of type Logical Clock and if it is it will create the clock add it to a map, where each clock is associated with one link type which corresponds to one network controller, thus there is one clock per network controller.

After this, the clock is returned in a shared pointer, so changes to the clock object are shared between the Participant and network controllers. The clock is returned to the *ParticipantControllerDoVisitor* which then sets it to the network controller. If synchronization method is different from the one specified, then the call to the *makeVectorClock* never happens, and the method returns immediately.

There was still the need to communicate the messages received by the network controller to the Participant Controller. To do this, the Participant controller first registers itself as an observer when the network controller visits it, thus creating one more layer of indirection making the Participant Controller and the network controller decoupled. This second layer is needed because the network controller is the one that receives the message, but it is the Participant Controller that does the processing of the received *VectorTimeStamp.*

7 Comparative Analysis

Evaluation of results is crucial to understand the advantages and disadvantages of the proposed solution and how it compares to the existing one. To address the need for evaluation and to understand if the solution is suitable, simulations are performed with the IB framework running with the existing time driven synchronization method (*TickTickDone*). Simulations are also performed for the solution proposed (*LogicalVectorClock*).

Co-simulations are run with simulated components design to have the no dynamic memory allocations; this is to ensure that each run of the simulation is consistent. Both the existing synchronization method and the proposed one are used to collect data for comparison.

While simulation and real time were collected from the simulator and the OS respectively. An empirical analysis is used to test the hypothesis that the new synchronization method and the underlying components supporting it are faster that the current time driven solution.

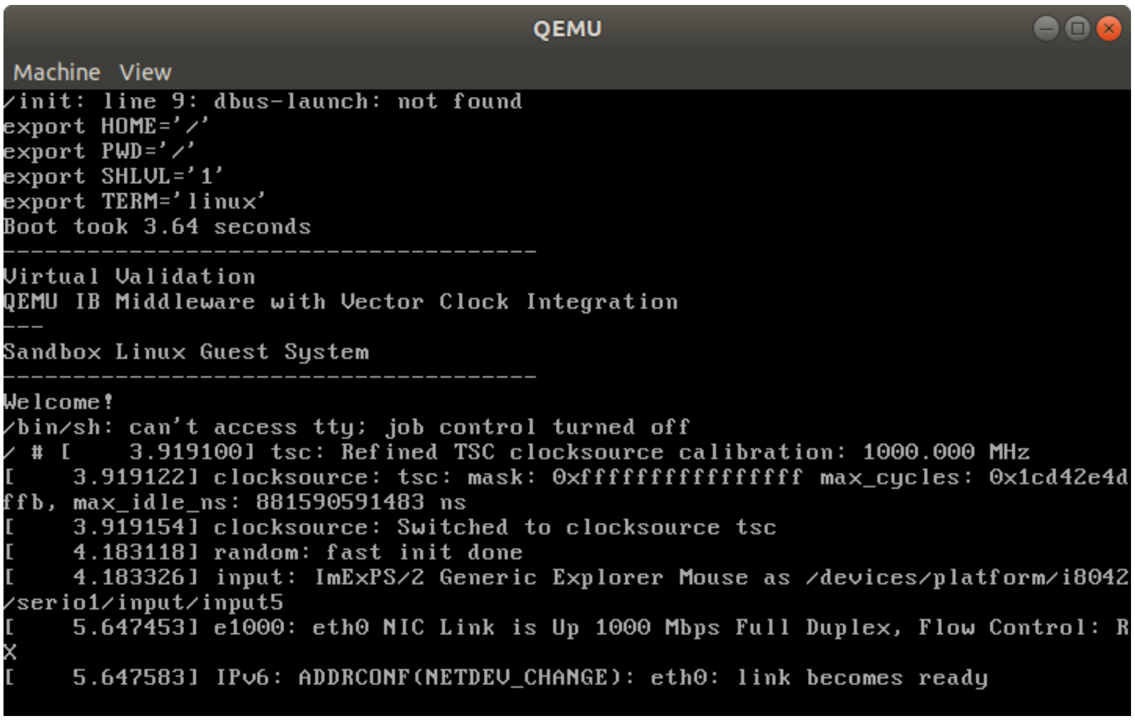
7.1 Data Gathering Method

To gather data, the IB is instantiated along with two participants running on a QEMU that is patched with qqq. The QEMU instances are started using the following configuration:

```
{ "qemuConfig": [ {
"executable"   : "../../sandbox/qemu_qqq/qemu/build/x86_64-softmmu/qemu-system-
x86_64",
"netNamespace"  : "qemu0",
"netIfName"    : ["qemu0-h"],
"arguments"    : { "clock"    : "-icount 1,sleep=off -rtc clock=vm -m 4G",
                   "kernel"   : "-kernel ../../sandbox/linux/build/arch/x86_64/boot/bzImage",
                   "initrd"   : "-initrd
/home/nuno/workplace/ctw/qibaV2/qiba/configuration/rootfs0.cpio.gz -device virtio-mouse-
pci -device virtio-keyboard-pci",
"network"   : {if0"   : {"tap"   : "-netdev tap,id=mynet0,ifname=tap0,script=no,downscript=no",
                          "dev"   : "-device e1000,netdev=mynet0,mac=",
                          "mac"   : "52:54:00:12:34:57"
                         } }}
```

Snippet 15 – QEMU JSON Configuration File

Snippet 15 – QEMU JSON Configuration File the configuration for an instance of QEMU. Only three configuration values are changed for each participant, these are, the *netNamespace,* the *netIfName, mac* and *initrd*. All of the parameters are same for every participant. The snippet also shows that QEMU is booted with the *icount* flag set to TRUE and the clock as being *vm*, indicating that the clock is the internal QEMU clock.



```
/init: line 9: dbus-launch: not found
export HOME='/'
export PWD='/'
export SHLVL='1'
export TERM='linux'
Boot took 3.64 seconds
------------------------------------
Virtual Validation
QEMU IB Middleware with Vector Clock Integration
---
Sandbox Linux Guest System
------------------------------------
Welcome!
/bin/sh: can't access tty; job control turned off
/ # [    3.919100] tsc: Refined TSC clocksource calibration: 1000.000 MHz
[    3.919122] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x1cd42e4d
ffb, max_idle_ns: 881590591483 ns
[    3.919154] clocksource: Switched to clocksource tsc
[    4.183118] random: fast init done
[    4.183326] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input5
[    5.647453] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: R
X
[    5.647583] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
```

Figure 28 – QEMU Clock Source

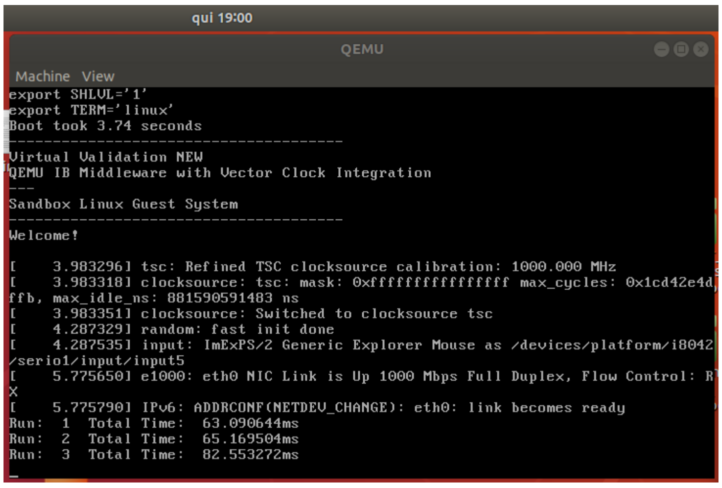
Figure 28 – QEMU Clock Source shows the internal clock source used by QEMU and the frequency of it, that in this case it is 1000 MHz. The clock source showed is TSC and since it is the internal QEMU clock it is independent of the host system.

To obtain measurements to allow the evaluation of the solution the following data points were collected:

- Simulation time
- Real time

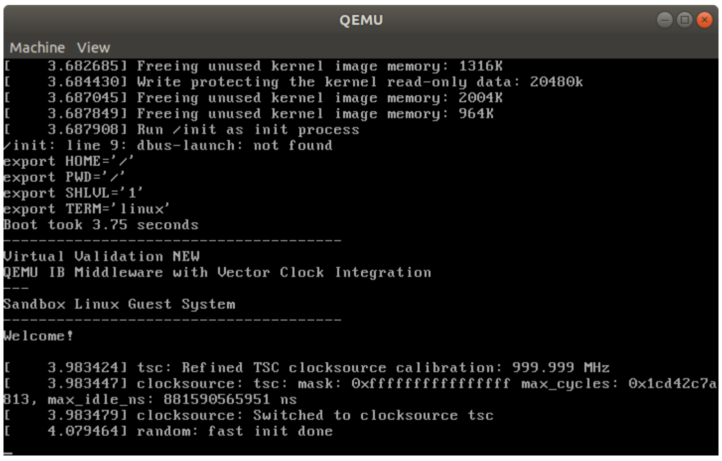
The simulation time is measured from the QEMU instance while the real time is measured from the host clock.

While real time is the only measure that matters for performance measurement, during some preliminary tests I observed that the boot time was inconsistent between start-ups, as depicted by the following images:



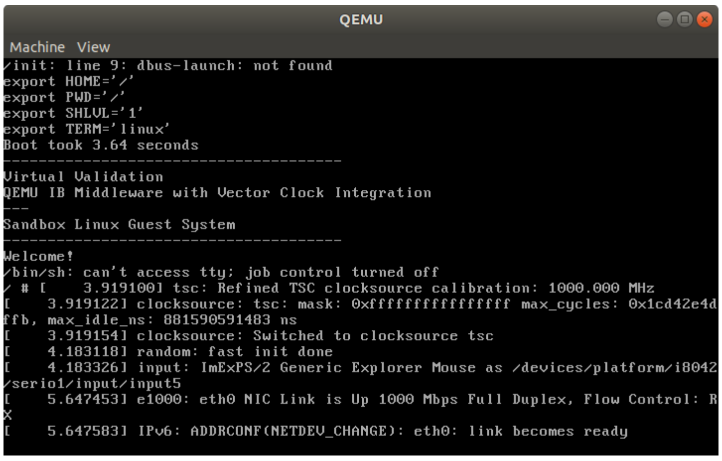
```
export SHLVL='1'
export TERM='linux'
Boot took 3.74 seconds
---------------------------------------
Virtual Validation NEW
QEMU IB Middleware with Vector Clock Integration
---
Sandbox Linux Guest System
---------------------------------------
Welcome!

[    3.983296] tsc: Refined TSC clocksource calibration: 1000.000 MHz
[    3.983318] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x1cd42e4d
ffb, max_idle_ns: 881590591483 ns
[    3.983351] clocksource: Switched to clocksource tsc
[    4.287329] random: fast init done
[    4.287535] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input5
[    5.775650] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: R
X
[    5.775790] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
Run:  1  Total Time:  63.090644ms
Run:  2  Total Time:  65.169504ms
Run:  3  Total Time:  82.553272ms
_
```

Figure 29 – QEMU Booting 1



```
[    3.682685] Freeing unused kernel image memory: 1316K
[    3.684430] Write protecting the kernel read-only data: 20480k
[    3.687045] Freeing unused kernel image memory: 2004K
[    3.687849] Freeing unused kernel image memory: 964K
[    3.687908] Run /init as init process
/init: line 9: dbus-launch: not found
export HOME='/'
export PWD='/'
export SHLVL='1'
export TERM='linux'
Boot took 3.75 seconds
---------------------------------------
Virtual Validation NEW
QEMU IB Middleware with Vector Clock Integration
---
Sandbox Linux Guest System
---------------------------------------
Welcome!

[    3.983424] tsc: Refined TSC clocksource calibration: 999.999 MHz
[    3.983447] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x1cd42c7a
813, max_idle_ns: 881590565951 ns
[    3.983479] clocksource: Switched to clocksource tsc
[    4.079464] random: fast init done
_
```

Figure 30 – QEMU Booting 2



```
/init: line 9: dbus-launch: not found
export HOME='/'
export PWD='/'
export SHLVL='1'
export TERM='linux'
Boot took 3.64 seconds
---------------------------------------
Virtual Validation
QEMU IB Middleware with Vector Clock Integration
---
Sandbox Linux Guest System
---------------------------------------
Welcome!
/bin/sh: can't access tty; job control turned off
/ # [    3.919100] tsc: Refined TSC clocksource calibration: 1000.000 MHz
[    3.919122] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x1cd42e4d
ffb, max_idle_ns: 881590591483 ns
[    3.919154] clocksource: Switched to clocksource tsc
[    4.183118] random: fast init done
[    4.183326] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input5
[    5.647453] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: R
X
[    5.647583] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
```

Figure 31 – QEMU Booting 3

Given the inconsistent boot times depicted by Figure 29, Figure 30 and Figure 31 with times of 3.75, 3.74 and 3.64 seconds respectively the simulation time was included to see it's variance with the tests.

```
void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet)
{
    msgHdlSt * msgHdlPtr{reinterpret_cast<msgHdlSt *>((void *)args)};
    // get raw message
    auto payload = std::vector<uint8_t>(packet,packet+header->len);
    // enqueue raw message to Tx fifo.
    utils::pout(PWARNING,_caller) << "Enqueuing...";
    (msgHdlPtr->ethG)->ethTxFifo.enqueue(payload);
    utils::pout(PWARNING,_caller) << "Done.";
    if(!msgHdlPtr->gSys.init){
        msgHdlPtr->gSys.wctTickStart =
boost::posix_time::microsec_clock::local_time();
        msgHdlPtr->gSys.init = true;
    }
    if(msgHdlPtr->gSys.runNs(simResNs.count())==-1){
        boost::posix_time::ptime
wctTickEnd(boost::posix_time::microsec_clock::local_time());
        msgHdlPtr-
>gSys.realTimeS=boost::posix_time::time_duration(wctTickEnd -
gSys.wctTickStart).total_nanoseconds();
        std::cout<<"Total Execution Time: "<<gSys.realTimeS/1000000<<"
Miliseconds"<<std::endl;
        msgHdlPtr->pCtrl->Stop("QEMU DONE");
    }
}
```

Snippet 16 – Real Time Computation

Snippet 16 shows the code used to measure the elapsed real (wall-clock) time during a simulation run. The code presented is part of QIBA and works by checking when the first packet is received, only then is the start time initiated. If the start time was initialized when the QIBA instance is created, then the QEMU boot overhead would be included in the real time measurement, since QIBA launches the QEMU instances.

While the boot overhead was eliminated a small overhead might still be encountered if it the first packet is not from the software, this was a compromise that had to be made otherwise the only solution to get an accurate 1:1 simulation time to real time would be to write a kernel module to be bundle with the kernel that QEMU uses so that time from the host OS could be read from user space, this would be in essence a new system call that would have to read memory from the host OS.

Simulation time is collected directly from the QEMU instance clock source, it is independent of the host system time and does not include the boot overhead. More specifically, the simulation time is collected by the software written to accomplish the tests.

Two types of tests were done, and two pieces of software were written to accomplish each of the types. One was designed to be network bound and another to be CPU bound. Both use the following snippets to calculate simulation time.

```
func Now() Time
```

Snippet 17 – Golang Now function

The function shown in Snippet 17 calls the system clock, that in this case the TSC counter as detailed above.

```
startTime := time.Now()
.
.
.
.Do the actual processing and sending of the network packets
.
.
endTime := time.Now()
timeItTookToRun  := endTime - startTime
```

Snippet 18 – Delta Time

Snippet 18 shows how the delta time which is the effective simulation time that is considered for this study is calculated. When the software starts to run, it first initializes every variable so that memory allocation overhead remains constant, so for CPU bound tasks the overhead should be the same on every run, but for network bound tasks the same cannot be said as will be showed latter.

After that is done, the start time is collected, the processing and sending of the network packets is done and the end time is collected. The start time is then subtracted to the end time giving exactly of much time the software took to run without the boot up overhead.

The two types were done so it could be seen if there were differences between different types of task, in this case a network intensive task and a CPU intensive one.

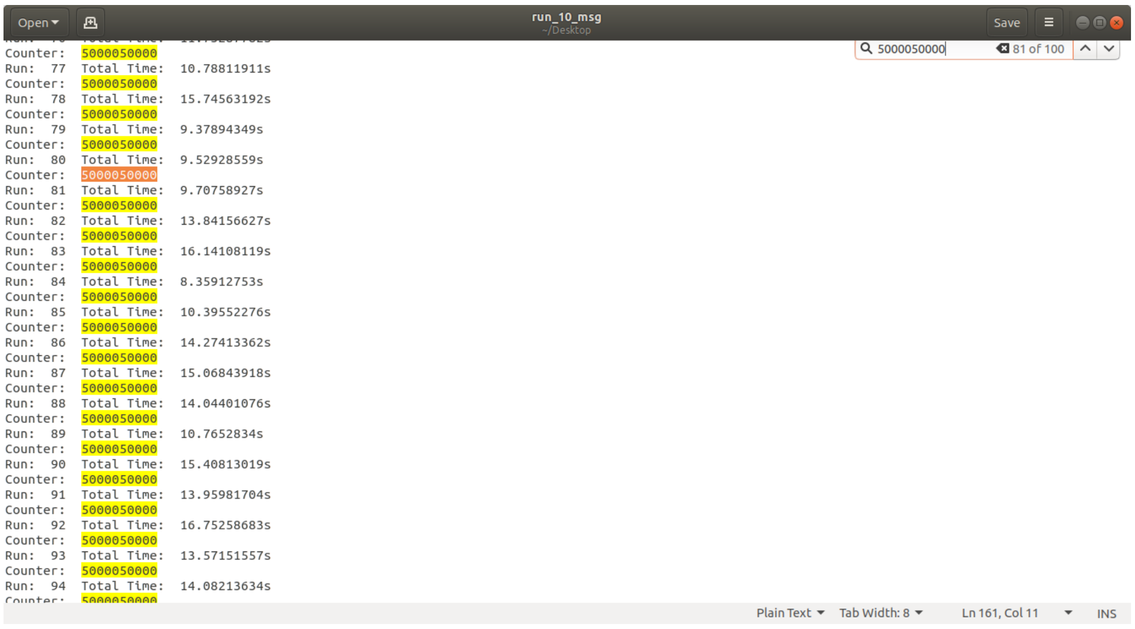
7.1.1 **Network Bound**

To run the network bound test, a piece of software was written to put pressure onto the network component, thus being network bound.

It is able to run as a server or client. When initialized as a server, it sends network packets as fast as it can, each packet data field contains a single integer. These integers sent start at 1 and end at 100000, thus excluding SYNC, ACK and SYNC-ACK packets, 100000 packets are sent. The client on its hand computes the sum of the received integers. The result of this sum is then compared with the following formula:

$$\frac{n(n+1)}{2} \tag{15}$$

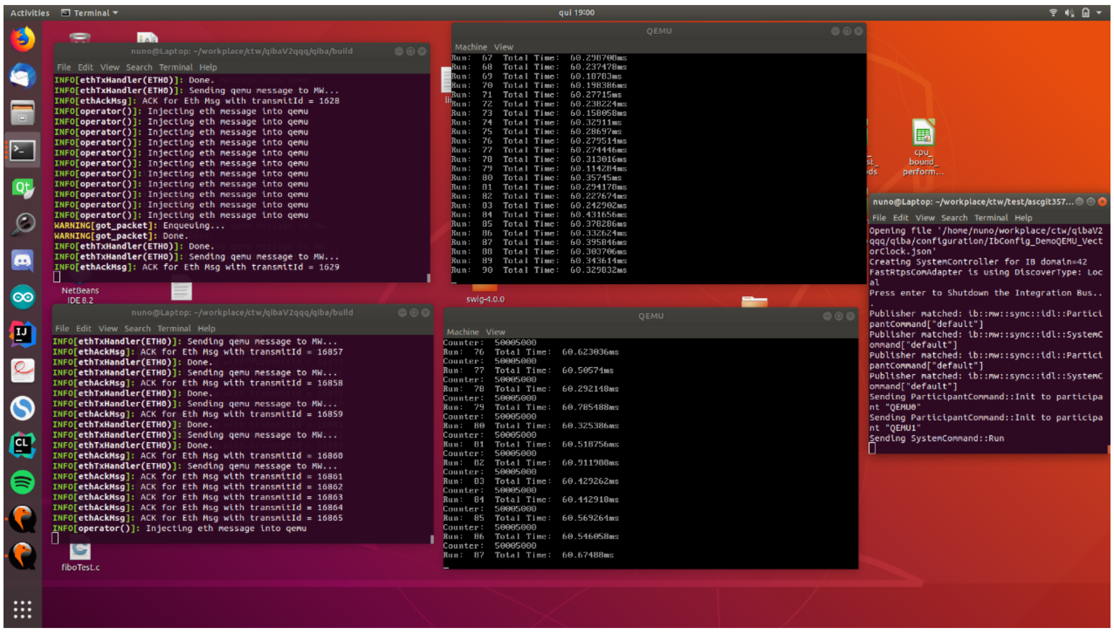
Equation (16) gives the result of the sum of the first n natural numbers, thus if the client receives all the packets, the sum of the numbers that were received must match the computation of the above formula.



```
Counter:  5000050000
Run:  77  Total Time:  10.78811911s
Counter:  5000050000
Run:  78  Total Time:  15.74563192s
Counter:  5000050000
Run:  79  Total Time:  9.37894349s
Counter:  5000050000
Run:  80  Total Time:  9.52928559s
Counter:  5000050000
Run:  81  Total Time:  9.70758927s
Counter:  5000050000
Run:  82  Total Time:  13.84156627s
Counter:  5000050000
Run:  83  Total Time:  16.14108119s
Counter:  5000050000
Run:  84  Total Time:  8.35912753s
Counter:  5000050000
Run:  85  Total Time:  10.39552276s
Counter:  5000050000
Run:  86  Total Time:  14.27413362s
Counter:  5000050000
Run:  87  Total Time:  15.06843918s
Counter:  5000050000
Run:  88  Total Time:  14.04401076s
Counter:  5000050000
Run:  89  Total Time:  10.7652834s
Counter:  5000050000
Run:  90  Total Time:  15.40813019s
Counter:  5000050000
Run:  91  Total Time:  13.95981704s
Counter:  5000050000
Run:  92  Total Time:  16.75258683s
Counter:  5000050000
Run:  93  Total Time:  13.57151557s
Counter:  5000050000
Run:  94  Total Time:  14.08213634s
Counter:  5000050000
```

Figure 32 – Client Log File

Figure 33 shows the client log file after 100 runs. For each run the results of the sum are logged and as the figure shows the counter value appears 100 times with the value of 5000050000 which is correct and can be verified with the above formula for $n = 100000$.



```
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 1628
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
INFO[operator()]: Injecting eth message into qemu
WARNING[got_packet]: Enqueuing...
WARNING[got_packet]: Done.
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 1629
```

```
Run:  67  Total Time:  60.298708ms
Run:  68  Total Time:  60.237478ms
Run:  69  Total Time:  60.18783ms
Run:  70  Total Time:  60.198386ms
Run:  71  Total Time:  60.27715ms
Run:  72  Total Time:  60.238224ms
Run:  73  Total Time:  60.158058ms
Run:  74  Total Time:  60.32911ms
Run:  75  Total Time:  60.28697ms
Run:  76  Total Time:  60.279514ms
Run:  77  Total Time:  60.274446ms
Run:  78  Total Time:  60.313016ms
Run:  79  Total Time:  60.114284ms
Run:  80  Total Time:  60.35745ms
Run:  81  Total Time:  60.294178ms
Run:  82  Total Time:  60.227674ms
Run:  83  Total Time:  60.242902ms
Run:  84  Total Time:  60.431656ms
Run:  85  Total Time:  60.378286ms
Run:  86  Total Time:  60.332624ms
Run:  87  Total Time:  60.395846ms
Run:  88  Total Time:  60.303706ms
Run:  89  Total Time:  60.343614ms
Run:  90  Total Time:  60.329832ms
```

```
Opening file '/home/nuno/workplace/ctw/qibaV2
qqq/qiba/configuration/IbConfig_DemoQEMU_Vect
orClock.json'
Creating SystemController for IB domain=42
FastRtpsComAdapter is using DiscoverType: Loc
al
Press enter to Shutdown the Integration Bus..
.
Publisher matched: ib::mw::sync::idl::Partici
pantCommand["default"]
Publisher matched: ib::mw::sync::idl::SystemC
ommand["default"]
Publisher matched: ib::mw::sync::idl::Partici
pantCommand["default"]
Publisher matched: ib::mw::sync::idl::SystemC
ommand["default"]
Sending ParticipantCommand::Init to participa
nt "QEMU0"
Sending ParticipantCommand::Init to participa
nt "QEMU1"
Sending SystemCommand::Run
```

```
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16857
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16858
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16859
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[ethTxHandler(ETH0)]: Done.
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16860
INFO[ethTxHandler(ETH0)]: Sending qemu message to MW...
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16861
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16862
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16863
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16864
INFO[ethAckMsg]: ACK for Eth Msg with transmitId = 16865
INFO[operator()]: Injecting eth message into qemu
```

```
Counter:  50005000
Run:  76  Total Time:  60.623036ms
Counter:  50005000
Run:  77  Total Time:  60.50574ms
Counter:  50005000
Run:  78  Total Time:  60.292148ms
Counter:  50005000
Run:  79  Total Time:  60.785488ms
Counter:  50005000
Run:  80  Total Time:  60.325386ms
Counter:  50005000
Run:  81  Total Time:  60.518756ms
Counter:  50005000
Run:  82  Total Time:  60.911988ms
Counter:  50005000
Run:  83  Total Time:  60.429262ms
Counter:  50005000
Run:  84  Total Time:  60.442918ms
Counter:  50005000
Run:  85  Total Time:  60.569264ms
Counter:  50005000
Run:  86  Total Time:  60.546058ms
Counter:  50005000
Run:  87  Total Time:  60.67488ms
```

Figure 33 – Data Gathering

Figure 33 shows the data being gathered with instances of QEMU and two instances of QIBA. The QIBA instances are responsible for routing the network traffic between the two instances of QEMU. The IB framework runs under QIBA and controls the synchronization between the participants as well as being responsible for packet loss recovery and detecting and solving concurrency problems, such as two messages being sent at the same time. The packet loss and concurrent message detection only applies if the synchronization method is *LogicalVectorClock*, else the IB only handles the synchronization aspect.

7.1.1.1 Network Bound Results

To compare the two synchronization methods both the simulation and real time were collected for each of the synchronization methods, namely the *LogicalVectorClock* which is the focus of this thesis and the *TickTickDone* synchronization method which was a method that was already implemented.

The tables bellow shows the data collect for each of these synchronization methods.

RUNS	SIM TIME	REAL TIME
RUN 1	63.064864ms	8231ms
RUN 2	62.593752ms	6453ms
RUN 3	62.069577ms	9895ms
RUN 4	63.378258ms	7169ms
RUN 5	63.387269ms	5703ms

RUN 6	62.702491ms	6224ms
RUN 7	62.216232ms	5557ms
RUN 8	62.456953ms	7376ms
RUN 9	63.437735ms	9303ms
RUN 10	64.178821ms	7472ms
RUN 11	62.307312ms	8466ms
RUN 12	62.010343ms	6617ms
RUN 13	62.626813ms	6209ms
RUN 14	62.653734ms	8602ms
RUN 15	62.289818ms	6926ms
RUN 16	60.998718ms	8449ms
RUN 17	61.551394ms	7171ms
RUN 18	61.873603ms	8792ms
RUN 19	62.386471ms	8411ms
RUN 20	63.124404ms	9437ms
VARIANCE	0.532467704	1645044,345
STANDARD DEVIATION	0.729703847ms	1,34064E+11ms

Table 2 – *LogicalVectorClock* Synchronization Method Test Results

RUNS	SIM TIME	REAL TIME
RUN 1	69.607588ms	11599ms
RUN 2	78.617844ms	12240ms
RUN 3	79.806824ms	12065ms
RUN 4	79.860814ms	11620ms
RUN 5	79.939568ms	10959ms
RUN 6	79.75925ms	12951ms
RUN 7	79.980172ms	14559ms
RUN 8	79.933388ms	14479ms
RUN 9	79.91568ms	13929ms
RUN 10	79.783158ms	14551ms
RUN 11	79.83057ms	12521ms
RUN 12	79.79455ms	12492ms
RUN 13	79.99288ms	12823ms
RUN 14	78.49678ms	9742ms
RUN 15	67.816554ms	14683ms
RUN 16	64.51793ms	11276ms
RUN 17	62.062884ms	10510ms
RUN 18	66.89646ms	14079ms
RUN 19	66.62906ms	13826ms
RUN 20	67.377748ms	13121ms

VARIANCE	44.11975379	2136208,513
STANDARD DEVIATION	6.642270228ms	2,25454E+11ms

Table 3 - *TickTickDone* Synchronization Method Test Results

Both tables show the data collect using the network bound piece of software. Both tables show that the simulation time differs between runs for the same synchronization method.

Since QEMU is running with the *icount* and the *qqq* patch this might be confusing, since it should theoretically be more or less consistent, but this behaviour was expected given the patch notes presented in chapter 4.2.1 and the boot inconsistency presented in 7.1.

The test was designed to put pressure on the network, which is routed through the host and check if the simulation time presented any deviation given the fact that the host is not a hard real time system and from the results that exact thing happened.

There is also the fact that the Linux Kernel also does not do a good job handling network packets, since, each network namespace contains a private linked list that is used to store packets in, each time a packet is routed to another namespace it must be fully copied. Creating a memory and performance overhead. This behaviour is documented in the Linux Foundation wiki (Foundation 2016).

The following figure shows the different types of network overhead:

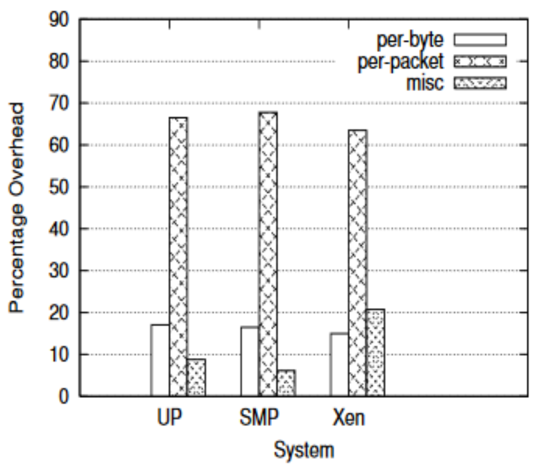


Figure 34 - Per-byte vs. Per-packet Overhead in Uniprocessor,

Multiprocessor and Virtualized Systems from (Menon and Zwaenepoel, n.d.)

Figure 34 shows and compared three types of network overhead using TCP packets. The per byte overhead measures the impact that the size of the packets have on the network, the per packet the impact of the number of packets have on the network and the misc which measures the overhead that is unrelated to receive processing, and are not strictly per-packet or per-byte (Menon and Zwaenepoel, n.d.).

7.1.2 **CPU Bound**

A new piece of software was written to see if QEMU could run deterministically if the network pressure was reduced and the CPU usage was increased. The software computes the first 40 Fibonacci numbers using the worst possible approach in order to maximize CPU and memory usage.

```
func computeAndSend(n int){
    for i := 1; i < n; i++ {
        int res = FibonacciRecursion(i);
        send(res);
    }
}

func FibonacciRecursion(n int) int {
    if n <= 1 {
        return n
    }
    return FibonacciRecursion(n-1) + FibonacciRecursion(n-2)
}
```

Snippet 19 – Fibonacci Golang Implementation

Snippet 19 shows the implementation of the Fibonacci sequence used. That implementation is guaranteed to put pressure on the CPU and memory since there is no tail recursion or memoization, thus a new stack trace is open every time and most of the call stack is repeated and computed again. After the computation the number is sent over the network to the other QEMU instance.

7.1.2.1 CPU Bound Results

Below is the results table for the CPU bound software for both synchronization methods, *LogicalVectorClock* and *TickTickDone*.

	LogicalVectorClock	**TickTickDone**

	Simulation Time (ms)	Simulation Time (ms)
Run 1	462,91	462,224691
Run 2	462,226901	462,224691
Run 3	462,224041	462,224691
Run 4	462,342787	462,224691
Run 5	462,243884	462,224691
Run 6	462,222025	462,224691
Run 7	462,235543	462,224691
Run 8	462,315065	462,224691
Run 9	462,245187	462,224691
Run 10	462,223739	462,224691
Run 11	462,23346	462,224691
Run 12	462,224451	462,224691
Run 13	462,336005	462,224691
Run 14	462,222023	462,224691
Run 15	462,237417	462,224691
Run 16	462,220649	462,224691
Run 17	462,337335	462,224691
Run 18	462,23482	462,224691
Run 19	462,229773	462,224691
Run 20	462,234895	462,224691
Variance	0,023162551	3,40124E-27
Standard Deviation	0,152192481	5,83201E-14

Table 4 – CPU Bound Results

As Table 4 shows the standard deviation for both synchronization methods. The *TickTickDone* synchronization method present one that is negligible, to the point of being considered zero, while the *LogicalVectorClock* shows a much greater standard deviation of around 0,153 milliseconds or 153 microseconds.

So, QEMU was performing deterministically at least using the software detailed in 7.1.2 but the *LogicalVectorClock* presented a standard deviation that might not be acceptable is some test cases that require microsecond resolution.

7.2 Performance Comparison

Table 2 it is possible to see that the variance values are much higher than zero for the simulation time on both tables, this means that when a task is network bound both methods start differing from the mean with the results of the *TickTickDone* showing 6.642270228 millisecond standard deviation against the 0.729703847 milliseconds of the *LogicalVectorClock*, thus for process that are quite network intensive the *LogicalVectorClock* method can maintain a more accurate simulation time.

In order to compare the actual performance real time is used, since the CPU bound task does not present this measure only the results from the network bound test are considered.

The following chart figure uses the real time data from Table 2 and Table 3 to better show the difference in real time between the two methods.

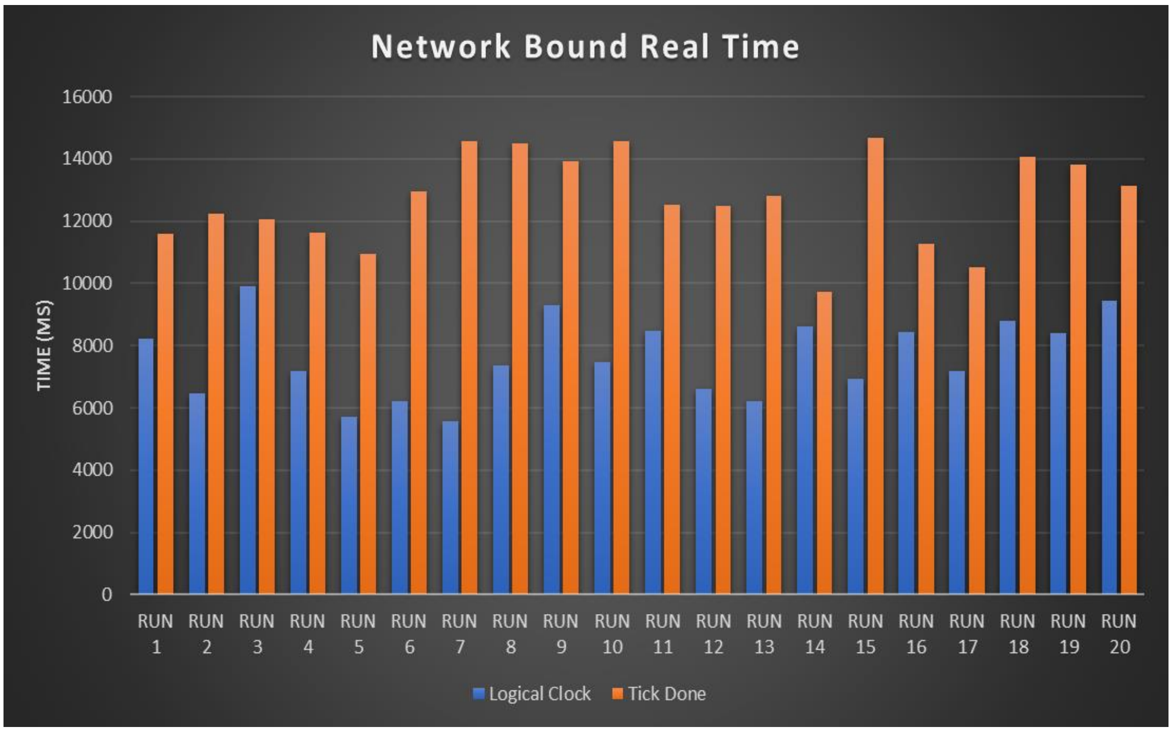


Figure 35 – Real Time Comparison Chart

Figure 35 shows that the measured real time is always higher for the *TickTickDone* method for each run. The following table shows the calculated average of all the runs:

REAL TIME AVERAGE

Logical Clock	Tick Done
7623.15MS	12701.25ms

Table 5 – Real Time Average

Table 5 shows the real time average calculated using the real time data from Table 2 and Table 3. From looking at the average values the Logical Clock as a lower average, meaning that it runs faster to completion that the *TickTickDone* method, thus having on average a higher performance on the runs of the test scenario. To calculate the performance increase of the *LogicalVectorClock* the following formulas are used:

$$\frac{Average_{LogicalClock} - Average_{TickDone}}{Average_{TickDone}} * 100 \tag{16}$$

$$\frac{Average_{LogicalClock}}{Average_{TickDone}} * 100 \tag{17}$$

Using equation (16) a value of -39.98% is obtained, meaning that the new synchronization method, *LogicalVectorClock*, is 39.98% smaller, thus 39.98% faster in respect to real time since it is the speed that it takes to run from start to finish of the test scenario that it is being measured.

With equation (17) a value of 60.02% is obtained, meaning that the average of the Logical method is just 60.02% of the average of the Tick Done, indicating that the Logical Clock synchronization method takes just 60.02% of the time to, compared to the Tick Done.

In conclusion the new synchronization method takes on average just 60.02% of the time of the old method showing and improvement of 39.98% in performance under the network bound test scenario.

8 Conclusion

The aim of this thesis was to find a new synchronization method that was faster thus more performant than the old method. Most of the effort was spent on researching the literature and implementing the synchronization method according to the works of (Lamport 1978) and (Fidge 1988).

Both are based around the notion of that time does not matter relying instead on the notion that if events are order and a relation can be established between them implicit synchronization is achieved. Leslie Lamport showed that using clocks was not needed, a single integer is enough to keep systems synchronized has long has every system understands what that number means and acts accordingly.

This thesis showed that both works can be applied for use in actual production systems, and given the results found the performance is generally better than time driven methods at least when it comes to network driven processes.

There is still space for further improvement, as discussed in 7.1 a custom kernel with a new system call could be implemented, leading to a 1 to 1 relationship between simulation time and real time, thus eliminating the trade-offs made. Also, using a deterministic kernel as the host would potentially stop the deviation seen in the simulation time.

Furthermore, while the CPU bound results shows that QEMU kept the simulation time without any standard deviation for the TickTickDone method, further tests should be done to see if this holds true for other computational task, mainly those using floating points.

References

Banks, Jerry, ed. 2001. *Discrete-Event System Simulation*. 3rd ed. Prentice-Hall International Series in Industrial and Systems Engineering. Upper Saddle River, NJ: Prentice Hall.

Bibliotheek, Koninklijke. 2019. "What Is Emulation? | Koninklijke Bibliotheek." February 8, 2019. https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation.

Bovet, Daniel Pierre, and Marco Cesati. 2002. *Understanding the Linux Kernel*. O'Reilly Media, Inc.

Chaves, Alcidney, Ricardo Maia, Carlos Belchio, Rui Araujo, and Goncalo Gouveia. 2018. "KhronoSim: A Platform for Complex Systems Simulation and Testing." In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 131–38. Turin: IEEE. https://doi.org/10.1109/ETFA.2018.8502602.

Cornell. 2006. "Simulation Time." January 18, 2006. http://jist.ece.cornell.edu/jist-user/node5.html.

Corsaro, Angelo, and Douglas C. 2012. "The Data Distribution Service – The Communication Middleware Fabric for Scalable and Extensible Systems-of-Systems." In *System of Systems*, edited by Adrian V. Gheorghe. InTech. https://doi.org/10.5772/30322.

Delaware, University. 2014. "Poll Interval Management." January 31, 2014. https://www.eecis.udel.edu/~mills/ntp/html/assoc.html#poll.

Dohyung Kim, Youngmin Yi, and Soonhoi Ha. 2005. "Trace-Driven HW/SW Cosimulation Using Virtual Synchronization Technique." In *Proceedings. 42nd Design Automation Conference, 2005.*, 345–48. Anaheim, CA, USA: IEEE. https://doi.org/10.1109/DAC.2005.193830.

Eidson, John. 2008. "IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," March, 94.

Elson, Jeremy, Lewis Girod, and Deborah Estrin. 2002. "Fine-Grained Network Time Synchronization Using Reference Broadcasts," 17.

Fidge, Colin J. 1988. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering." In .

Foundation, Linux. 2016. "Networking:Sk_buff [Wiki]." July 19, 2016. https://wiki.linuxfoundation.org/networking/sk_buff.

Gomes, Cláudio, Casper Thule, David Broman, Peter Larsen, and Hans Vangheluwe. 2017. "Co-Simulation: State of the Art," February.

Graziano, Charles David. 2011. "A Performance Analysis of Xen and KVM Hypervisors for Hosting the Xen Worlds Project," 50.

Harris, David. 1999. "Skew-Tolerant Circuit Design." PhD Thesis, Stanford University.

"IEEE Standard for Standard SystemC Language Reference Manual." 2012. IEEE. https://doi.org/10.1109/IEEESTD.2012.6134619.

Intel. 2004. "HPET Spec." https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf.

———. 2005. "Advanced Configuration and Power Interface Specification." https://www.intel.com/content/dam/www/public/us/en/documents/articles/acpi-config-power-interface-spec.pdf.

Iordache, Sergiu. 2019. "TSC Resynchronization - The Chromium Projects." January 25, 2019. http://www.chromium.org/chromium-os/how-tos-and-troubleshooting/tsc-resynchronization.

Kenton, Will. 2018. “Perceived Value.” Investopedia. March 30, 2018. https://www.investopedia.com/terms/p/perceived-value.asp.

Koen, Peter, Greg Ajamian, Robert Burkart, Allen Clamen, Jeffrey Davidson, Robb D’Amore, Claudia Elkins, et al. 2001. “Providing Clarity and A Common Language to the ‘Fuzzy Front End.’” *Research-Technology Management* 44 (2): 46–55. https://doi.org/10.1080/08956308.2001.11671418.

Krzyzanowski, Paul. 2015. “Process Scheduling.” Process Scheduling. February 28, 2015. https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html.

———. 2017. “Assigning Lamport & Vector Timestamps.” September 29, 2017. https://www.cs.rutgers.edu/~pxk/417/notes/clocks/index.html.

Laird, Jeff. 2012. “PTP Background and Overview,” July, 9.

Lamport, Leslie. 1978a. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM* 21 (7): 558–65. https://doi.org/10.1145/359545.359563.

———. 1978b. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM* 21 (7): 558–65. https://doi.org/10.1145/359545.359563.

Lesch, Robert. 2005. “U. S. Air Force Network Time Service,” May, 23.

“Linux System Programming, 2nd Edition [Book].” 2013. May 2013. https://learning.oreilly.com/library/view/linux-system-programming/9781449341527/.

Maria, Anu. 1997. “Introduction to Modeling and Simulation,” 7.

Menon, Aravind, and Willy Zwaenepoel. n.d. “Optimizing TCP Receive Performance,” 14.

Microsoft, Microsoft. 2018. “Acquiring High-Resolution Time Stamps.” May 31, 2018. https://docs.microsoft.com/en-us/windows/desktop/sysinfo/acquiring-high-resolution-time-stamps.

Mills, D., J. Martin, J. Burbank, and W. Kasch. 2010. “Network Time Protocol Version 4: Protocol and Algorithms Specification.” RFC5905. RFC Editor. https://doi.org/10.17487/rfc5905.

Montini, Laurent, Tim Frost, Greg Dowd, and Vinay Shankarkumar. 2017. “Precision Time Protocol Version 2 (PTPv2) Management Information Base.” June 2017. https://tools.ietf.org/html/rfc8173.

Munawar, Asim, Takeo Yoshizawa, Tatsuya Ishikawa, and Shuichi Shimizu. 2013. “On-Time Data Exchange in Fully-Parallelized Co-Simulation with Conservative Synchronization.” In *2013 Winter Simulations Conference (WSC)*, 2127–38. Washington, DC, USA: IEEE. https://doi.org/10.1109/WSC.2013.6721590.

“Network Time Protocol: Best Practices White Paper.” 2008. Cisco. December 17, 2008. https://www.cisco.com/c/en/us/support/docs/availability/high-availability/19643-ntpm.html.

Nicola, Susana. 2018. “Análise de Valor de Negócio.”

Nielsen, Claus Ballegaard, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. 2015. “Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions.” *ACM Computing Surveys* 48 (2): 1–41. https://doi.org/10.1145/2794381.

ntp.org. 2019. “What Is NTP?” 2019. http://www.ntp.org/ntpfaq/NTP-s-def.htm#AEN1259.

Nutaro, James. 2016. “Qqq: Module for Synchronizing with a Simulation Clock - Patchwork.” February 6, 2016. https://patchwork.kernel.org/patch/8235161/.

OSDev. 2017. “Programmable Interval Timer - OSDev Wiki.” May 24, 2017. https://wiki.osdev.org/Programmable_Interval_Timer.

Paoloni, Gabriel, and Intel. 2010. “Benchmark Code Execution Times on Intel ® IA-32 and IA - 64 Instruction Set Architectures.”

https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

"Precision Time Protocol Software Configuration Guide for IE 2000U and Connected Grid Switches." 2018. Cisco. August 30, 2018. https://www.cisco.com/c/en/us/td/docs/switches/connectedgrid/cg-switch-sw-master/software/configuration/guide/ptp/b_ptp_ie2ku.html.

Quaglia, Davide, Franco Fummi, Maurizio Macrina, and Saul Saggin. 2011. "Timing Aspects in QEMU/SystemC Synchronization," January, 4.

Rhee, Ill-Keun, Jaehan Lee, Jangsub Kim, Erchin Serpedin, and Yik-Chung Wu. 2009. "Clock Synchronization in Wireless Sensor Networks: An Overview." *Sensors* 9 (1): 56–85. https://doi.org/10.3390/s90100056.

rtsj.org. 2018. "Overview (Clocks and Timers)." 2018. http://www.rtsj.org/specjavadoc/timers_overview-summary.html.

Saewong, S., and R. Rajkumar. 1999. "Cooperative Scheduling of Multiple Resources." In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, 90–101. https://doi.org/10.1109/REAL.1999.818831.

Sampath, Amritha, and C. Tripti. 2012. "Synchronization in Distributed Systems." In *Advances in Computing and Information Technology*, edited by Natarajan Meghanathan, Dhinaharan Nagamalai, and Nabendu Chaki, 417–24. Advances in Intelligent Systems and Computing. Springer Berlin Heidelberg.

Saxena, Piyush. 2014. "OSI Reference Model – A Seven Layered Architecture of OSI Model" 1 (10): 12.

Smilkstein, Tina Harriet. 2007. *Jitter Reduction on High-Speed Clock Signals*. Citeseer.

STMicroelectronics. 2004. "Real Time Clock Application Example," 9.

Sundararaman, Bharath, Ugo Buy, and Ajay D. Kshemkalyani. 2005. "Clock Synchronization for Wireless Sensor Networks: A Survey." *Ad Hoc Networks* 3 (3): 281–323. https://doi.org/10.1016/j.adhoc.2005.01.002.

Thekkilakattil, A., R. Dobrin, and S. Punnekkat. 2012. "Probabilistic Preemption Control Using Frequency Scaling for Sporadic Real-Time Tasks." In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, 158–65. https://doi.org/10.1109/SIES.2012.6356581.

Thiebaut, Stefaan. 2002. "Real-Time Publish Subscribe (RTPS) Wire Protocol Specification." February 2002. https://tools.ietf.org/html/draft-thiebaut-rtps-wps-00.

Thiruvathukal, George, and Joe Kaylor. 2013. "Clocks and Synchronization — Distributed Systems Alpha Documentation." 2013. http://books.cs.luc.edu/distributedsystems/clocks.html.

Thompson, Robert Bruce Thompson, Barbara Fritchman. 2009. *PC Hardware in a Nutshell*. http://shop.oreilly.com/product/9780596005139.do.

VMware. 2011a. "Timekeeping in VMware Virtual Machines," 32.

———. 2011b. "Timekeeping in VMware Virtual Machines," 32.

Weil, Stefan. 2019. "QEMU Version 3.1.0 User Documentation." February 8, 2019. https://qemu.weilnetz.de/doc/qemu-doc.html.

Wonyong Sung, and Soonhoi Ha. 1998. "Optimized Timed Hardware Software Cosimulation without Roll-Back." In *Proceedings Design, Automation and Test in Europe*, 945–46. Paris, France: IEEE Comput. Soc. https://doi.org/10.1109/DATE.1998.655981.

Yoo, Sungjoo. 1998. "Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model." In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE'98)*, 71–75. https://doi.org/10.1109/HSC.1998.666240.