UNIVERSITY CARLOS III OF MADRID

BACHELOR IN TELECOMMUNICATION TECHNOLOGY ENGINEERING

BACHELOR THESIS

# VIRTUAL DISTRIBUTED ENVIRONMENTS FOR SYSTEMS WITH TIME REQUIREMENTS

Author: Carlos Antonio Perea Gómez
Tutor: Marisol García Valls

June 2016

## Acknowledgments

To my supportive family, specially those who will not be able to share this moment, and my *Bicho*, for having to deal and yet never quit believing in *Carlos's Style*.

Not to forget those friends who supported me during my whole bachelor, a time full of happiness but also some frustrating moments. Unforgettable chapter of my life

Also to Marisol, my tutor, that despite the abnormal situation decided and committed to supervise my project throughout its duration.

# Abstract

Virtualization is widely propagating technology that is used to run multiple virtual machines on the same computational unit by means of a piece of firmware, hardware or software called a hypervisor.

Despite having been used since the 60âs, the current indisputable need for fast reliable communication may put this technology to question. This project analyzes the amount of impact the virtualization has on the transmission times. In the first part, the Xen hypervisor, configured with different virtual environments, simulating complex scenarios, will be evaluated to determine the size of the impact. As a bridge between the multiple virtual machines, middleware Ice, will be used.

Furthermore lower in the scale, for embedded systems, the XtratuM hypervisor was designed to support real-time systems. The second part is dedicated to evaluating whether the communication maintains the real time property of these systems. Bare boned virtualization will be implemented in this second part of the project.

***Keywords:*** *virtualization, hypervisor, middleware, Ice, XtratuM, Xen, transmission time*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Project Motivation

Technology must keep up with the needs of the new innovative optimized systems. If the automotive or aeronautic technology field is taken as a reference, an increasing tendency of using pure electronically systems to handle complete systems can be observed. In modern cars, barely anything is still driven manually in a mechanic manner.

For instance traditional steering or throttle has been modified to electronically driven systems. In airplanes, fly-by-wire technology has been developed to optimize the old fashioned hydraulic systems. Stability, precision and stand-alone ability are some of the properties of modern systems.

All these electronic systems need a computational circuit to perform all the arithmetic calculations as well as logic, I/O and control operations. Furthermore additional complex hardware components are required to run the systems. Due to the large amount of different electronic systems found inside an aircraft multiple computational units may be required. This high demand of units has a negative impact on the cost effectiveness of the aircraft or cars. In addition physical available weight and space are affected.

Making use of complete systems, consisting of multiple computational units to achieve the desired functionality, from a technological point of view may give the impression of being outdated. A different approach must be taken when designing and implementing the system if an optimum trade-off between performance and cost effectiveness is to be achieved.

The possibility of making use of virtual environments offers a advantages such as improving the execution of specific applications. Furthermore, physical space reduction is perhaps the most obvious advantage being able to run all systems from the same computational unit.

Closely related with the space reduction is the weight reduction. From the aerospace industry it is know that carrying weight on board causes an increase in fuel consumption, hence the decrease of such weight will perhaps leave space for some further equipment, or simply just save fuel hence money. Additionally this would be contributing to the environment as emissions are reduced.

However the possibility of turning multiple machines into a single one, keeping up the performance of the different virtual machines is not as straight forwarded. There are many factors that must be examined in depth before knowing whether the system is capable of running in a virtual environment.

One of the more critical aspect, regardless of whether the virtual CPUs, or the shared storage are good enough for the system, is the connectivity between the different systems.

Inside an automobile or an aircraft, the systems are interconnected with each other, as complex data cross reference operations are made to calculate millions of instructions, and outcomes. This requires a perfect connectivity pattern among the different systems. And here lies the crucial aspect of virtualization. Is the transmission using this technology fast enough? Is it as reliable as traditional communication systems?

Time is a critical factor, milliseconds, even microseconds may change the course of outcome in a specific situation. Taking the ESP system as an example, the continuous monitoring and comparison of two data samples, the intended and the actual moving direction, helps keep the car on the track under any circumstance. In the event of an icy patch, the system reacts and corrects the trajectory to keep the car safe on track.

If the communication under harsh circumstances would fail, the system would turn useless and the car and its passengers may stumble into a unfavorable situation, which may lead to an accident, causing minor or major personal and material damages.

As communication is not entirely instantaneous, there always exist a transmission delay. These systems are designed to cope with a time delay tolerance. This delay inevitably is affected when operate within a virtual layer.

The study of time in the systems and how it behaves under certain circumstances is hence the key for properly assessing virtual systems.

For real time systems, it is crucial to know the temporal working execution limits. Therefore it is necessary to use techniques that offer deterministic executions and the correspondent support tools, such as OSes of virtualization software among others.

## 1.2 Objectives

Answering whether the communication between virtual machines is predictable and reliable for the systems to run properly is a complex matter that requires to carry out a responsible study.

The aim of this project is to contribute towards the analysis of temporal behaviour of virtualized systems by means of performing structured performance tests on two largely different virtual environments and capturing delivery times.

On the one hand, a virtual environment is set up with **Xen**, and on the other hand a virtual environment is set up with **XtratuM**. Two different hypervisors. XtratuM is used for embedded real time systems, and the opensource Xen hypervisor offers a large compatibility with many different Unix and Linux distributions. [6]

The first part of the project, related to Xen will aim to determine the time cost of the virtual implementation of a complex system composed of multiple virtual machines running separate OSes. Analyzing delay times with different system loads. Determining this way speed and reliability these virtual systems can have under a high system load.

In order to obtain clear and usable results from the trials four different environments will be developed. The system will be tested making use of a middleware, an **IceStorm** service application will be developed. ***Subscriber*** and ***Publisher*** will be programmed and executed in the multiple environments.

In order to achieve the communication for the IceStorm service, an additional **IceBox** service must run in the system, in order to deliver, receive and forward the different set of data to the respective data owner.

On the second part of the project, focused on a simple bare-boned system, XtratuM will be put under analysis. The deployment of the hypervisor and the inter-partition communication will be used to analyze the effectiveness of the real time hypervisor, and prove whether the communication among virtual machines (partitions) is indeed without any delay.

Merging these two parts of the project the overall objective is to analyze the responsiveness and time cost in the transmissions within virtual environments when it comes to communication. With this analysis, an assessment can be made for, or against the use of virtual systems.

## 1.3 Contents

This document is branched into the following chapters:

- **Chapter 1**: Introduces the related topic by means of the project motivation and objectives.
- **Chapter 2**: Holds latest virtual environments and virtual technologies, the state of art, that will be used and analyzed throughout the project.
- **Chapter 3**: The development of Xen, the first part of the project, is described. Requirements and architectures among other information can be found here
- **Chapter 4**: Trials of the Xen development with the results and conclusions.
- **Chapter 5**: The development of XtratuM, the second part of the project, is described. Requirements and architectures among other information can be found here.
- **Chapter 6**: Trials of XtratuM development with the results and conclusions.

- **Chapter 7**: Holds general information about the project such as the project history, encountered obstacles, and a short legal, economic and environmental sphere analysis.
- **Chapter 8**: The conclusions to the main objectives and details about future works are described in this last section

# Chapter 2

# State of Art

## 2.1 Real Time Systems

A real time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period of time, i.e. hardware and software subject to time constraint. This constraint is usually known as “deadline” and can be understood in the orders of milliseconds or microseconds depending on the user’s specification.

It is important to note that only the combination of the correct logic output of this system together with the exact delivery time will result in a successful result. Any correct logic output at a non-desired time is beyond acceptance. The differentiation between a wrong logic output and an off time result is nonexistent. [1][2]

Examples of these systems can be found in the automobile sector, for example ABS, ESP or TC systems, as well as in spacecraft sector using the Eurofighter Typhoon as an example of a fly-by-wire aircraft.

Some requisites of a RTS are the small physical size, predictable quick response to triggers and high processor utilization in order to avoid system over-sizing.

Real Time Systems can be classified under three different categories depending on the consequence of missing a deadline:

- ***Hard Real Time System***: The consequence of overrunning a given deadline results in a potential complete system failure.

- ***Firm Real Time System***: A system failure is discarded if the deadline is not met, however the result will be obsolete and of no further use.

- ***Soft Real Time System***: The consequences of overrunning a given deadline are not as critical to the system as with the Hard Real Systems, nevertheless the continuous overrunning of deadlines will affect the QoS.

Furthermore a fourth category can be used when describing a system where only n out of k deadlines have to be meet in order to be fully functional. These systems can be classified as *Weakly Hard Real Time Systems.*

As for the RTSs we can differentiate between Event Triggered Systems, an event initiates the correspondent activities, and Time Triggered Systems, execution of one or more sets of tasks according to a predetermined task schedule and Cyclic Systems. [3]

This can be implemented in a Task Model with Periodic Tasks (time triggered), Aperiodic Tasks (event Triggered) and Sporadic Tasks, Aperiodic Tasks with known minimum inter-arrival times. Together with the definition of Task Constraints such as Deadline, Resource and Precedence constraints, can be used to perform Schedulability and Worst Case Execution Time (WCET) Analysis.

Even though multiprocessor and multithread capability is not a mandatory requirement, it is desired to improve performance, throughput, fault tolerance and reliability of a RTS.[3]

Different HW configuration can be used to obtain Symmetric Multithreading (SMT), Symmetric Multiprocessor (SMP), Asymmetric Multiprocessor and Distributed Systems.

## 2.2 Virtual Environments

**KVM** (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, which provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko.

Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc.[4]

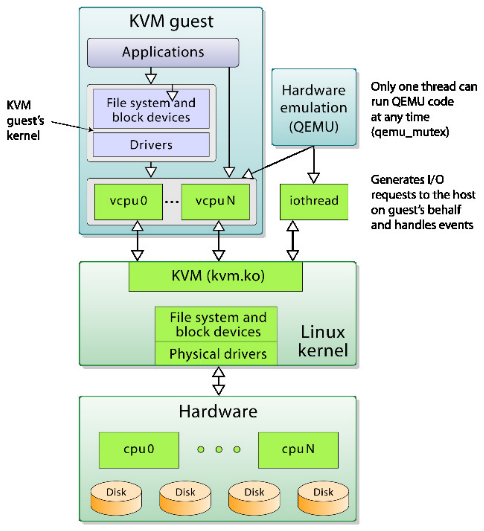

KVM guest
Applications
File system and block devices
Drivers
KVM guest's kernel
vcpu0
vcpuN
Hardware emulation (QEMU)
Only one thread can run QEMU code at any time (qemu_mutex)
iothread
Generates I/O requests to the host on guest's behalf and handles events
KVM (kvm.ko)
File system and block devices
Physical drivers
Linux kernel
Hardware
cpu0
cpuN
Disk
Disk
Disk
Disk

Figure 2.1: KVM Architecture
Source: *https://upload.wikimedia.org/wikipedia/commons/4/40/Kernel-based_Virtual_Machine.svg*

**QEMU** is an emulator and virtualization machine that allows you to run a complete operating system as just another task on your desktop. It can be very useful for trying out different operating systems, testing software, and running applications that won't run on your desktops native platform.

QEMU runs on x86 systems running Linux, Microsoft Windows, and some UNIX platforms, and can host target systems from a range of different microprocessors.[5]

It is important to note that KVM is a fork of Qemu executable. A system can run Qemu by itself and it will handle all the virtual machine resources as well as all the virtual HW and CPU. The only drawback is the extreme slow communication between the host and guests CPU's. This aspect can be optimized using KVM on top, which only focuses on the acceleration of the CPU.

If we define the hypervisor or Virtual Machine Monitor (VMM) as a system that creates and runs VM then we can affirm that QEMU is a fully standalone hypervisor whereas KVM is just an accelerator that uses processor extensions.

## 2.3 Xen

Now called Xen Project is a native hypervisor (Type 1) which makes it possible to run various instances or rather different operating systems on a single machine. Some of the applications Xen is used for are virtualization of servers and desktops or IaaS applications.

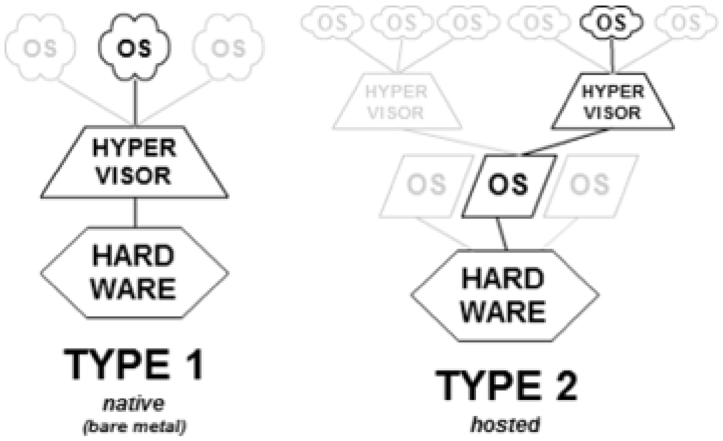


Figure 2.2: Hypervisor Types
Source: *https://upload.wikimedia.org/wikipedia/commons/e/e1/Hyperviseur.png*

Due to the microkernel design, Xen only generates a small memory footprint (around 1MB) and restricts a limited interface to the guests. This enforces the robustness and security of the hypervisor.

Additionally it provides us with driver isolation. This guarantees that a driver's fault within a VM does not affect the rest of the system. And last but not least one of the most important and advantageous features of Xen is the Paravirtualization. It will be described further on in more detail.[6]

Looking into the architecture of Xen, first we must define some key components for a better comprehension.

*Domain*: Running instance of a virtual machine

*Domain 0*: contains drivers for the hardware, as well as the toolstack to control VMs

*Toolstack*: section that covers various toolstack front-ends available as part of the Xen Project Stack and the implication of using each

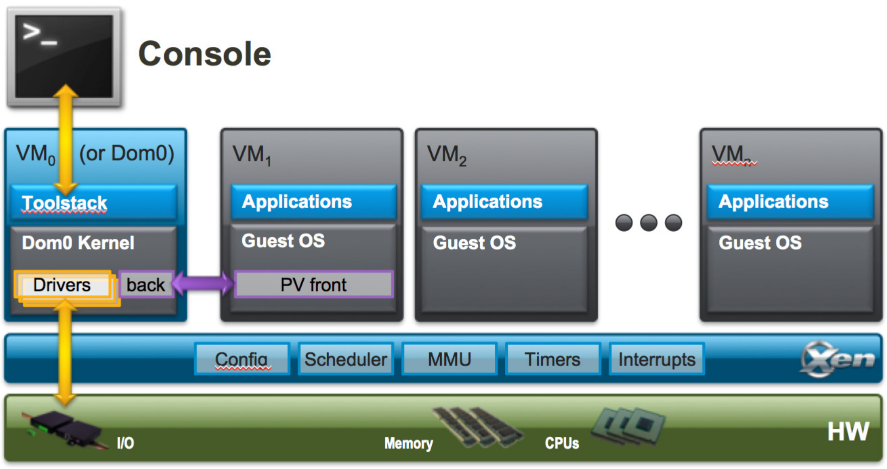


Figure 2.3: Xen Architecture
Source: *http://wiki.xen.org/wiki/Xen_Project_Software_Overview*

As pictured it can see that the hypervisor runs directly on the HW and is responsible for managing the memory, the CPU and diverse Input/Output interrupts. As mentioned before the domain 0 is a special domain that apart from containing the drivers it will control and manage creation, destruction and configuration of further virtual machines of the system.[6]

Mentioned above, Xen offers Paravirtualization (PV), but this is not the only approach that Xen offers when running the guest operating systems, Hardware Assisted Virtualization (HVM) is also supported. From these two, up to five different modes can be depicted:

- **PV**: efficient and lightweight technique that does not require to emulate the full set of hardware and firmware services. Guest operating systems are aware of the hypervisor and run more efficiently without the emulation nor the virtualization of HW, i.e. no explicit virtualization support. However PV-Kernel and PV-drivers are essential.

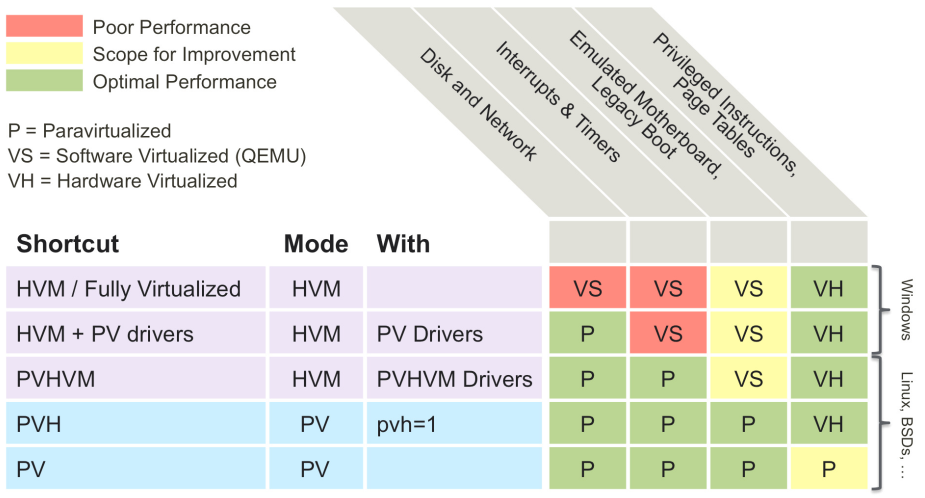
- **HVM**: also known as full virtualization, makes use of virtualization extensions from the host CPU to achieve guest virtualization. Qemu is used to emulate HW, adapters and the BIOS, The extensions are used for performance boost purposes. Kernel support is not required.

- **HVM with PV Drivers**: Based on HVM, the utilization of PV drivers for I/O speeds up the performance of HVM.

- **PVHVM**: also known as PV-on-HVM drivers instead of the default PV drivers. These drivers bypass the emulation for disk and network IO. As a result performance is boosted.

- **PVH**: PV guest OS using the default PV drivers for boot and I/O, for the remaining drivers HW virtualization extensions is used. (without emulation)

All of this previous configurations can be summarized in the following figure:



🟥 Poor Performance
🟨 Scope for Improvement
🟩 Optimal Performance

P = Paravirtualized
VS = Software Virtualized (QEMU)
VH = Hardware Virtualized

| Shortcut | Mode | With | Disk and Network | Interrupts & Timers | Emulated Motherboard, Legacy Boot | Privileged Instructions, Page Tables | |
|---|---|---|---|---|---|---|---|
| HVM / Fully Virtualized | HVM | | 🟥 VS | 🟥 VS | 🟨 VS | 🟩 VH | Windows |
| HVM + PV drivers | HVM | PV Drivers | 🟩 P | 🟥 VS | 🟨 VS | 🟩 VH | |
| PVHVM | HVM | PVHVM Drivers | 🟩 P | 🟩 P | 🟨 VS | 🟩 VH | Linux, BSDs, ... |
| PVH | PV | pvh=1 | 🟩 P | 🟩 P | 🟩 P | 🟩 VH | |
| PV | PV | | 🟩 P | 🟩 P | 🟩 P | 🟨 P | |

Figure 2.4: Virtualization Modes' Properties [5]

Another advantage when using Xen is the ability to live migrate the guest operating systems between physical hosts across a network without the loss of availability.

Even though x86 as well as ARM architectures are supported by Xen, the difference of scalability of the systems is remarkable. The following table represents data for the latest Xen 4.6 version[6]:

| | | ***x-86 Architectures*** | ***ARM Architectures*** |
|---|---|---|---|
| **Host** | Physical CPUs | 4095 | 8/128 |
| | Physical RAM | 16 TB | 16 GB/ 5 TB |
| **Guest** | Virtual CPUs | 512 (PV) | 8/128 |
| | Virtual RAM | 512 (PV) | 1 TB |

Table 2.1: Xen 4.6 Specifications for Different Architectures

## 2.4 Xtratum

Xtratum is real time hypervisor (Type 1) that provides a framework to run several operating systems (or real-time executives) in a robust partitioned environment. It can be used to build partitioned systems. It meets safety critical real-time requirements and was designed as a nanokernel, i.e. it virtualizes the essential HW devices for the execution of several concurrent OSes, being at least one a Real Time Operating System.

Furthermore in order to reduce the design complexity and increase the reliability, XtratuM was designed as a monolithic, nonpreembale kernel. It is important to note that it is independent of Linux and bootable. In order to describe XtratuM we first need to clarify the main concept of this technology, the Partitioning Concept.[8]

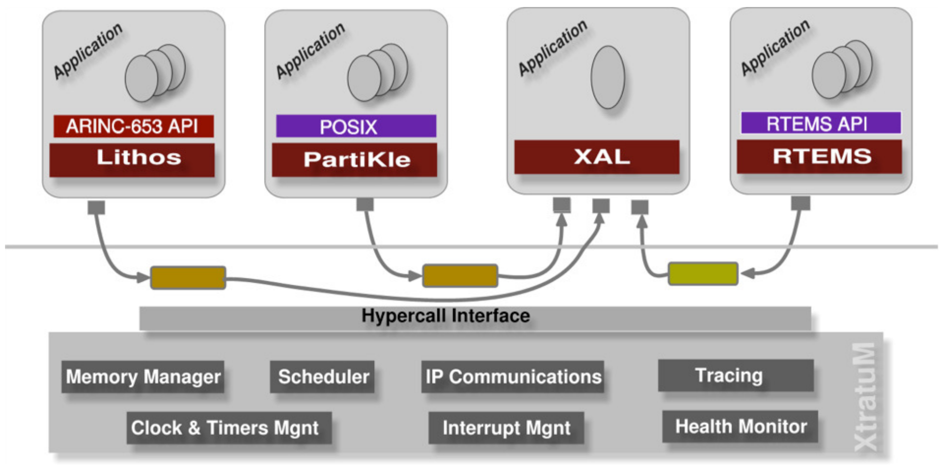


Figure 2.5: Xtratum Architecture [7]

## Architecture

XtratuM is in charge of the virtualization of the services to the partitions. CPU, memory interrupts and some I/O ports are virtualized. The lower layer of the diagram pictures the internal architecture of the hypervisor. Furthermore three layers can be identified[9][7]:

- **Hardware-dependent layer**: Implantation of drivers required for strictly necessary hardware such as CPU. HAL (Hardware Abstraction Layer) isolates this layer.
- **Internal-service layer**: Provides the strictly necessary C-functions such as *strcpy*, and data structures. This layer is not accessible by any partition.
- **Virtualization-service layer**: Provides para-virtualization services, via hypercall mechanism.

The internal architecture of the hypervisor includes:

- **Scheduling** A cyclic scheduling policy is used.
- **Memory Management** Spatial isolation using hardware mechanisms.
- **Interrupt Management** XtratuM handles the interrupts and propagates these to the different partitions if necessary.
- **Clock and Timer Management**
- **IP Communication** The communication between partitions is modeled using ports. XtratuM is in charge of creating the ports, and the correspondent channels for data flow.
- **Health Monitor** Detects and reacts to any anomaly that may arise during the session.
- **Tracing** XtratuM provides a mechanism for tracing.

#### Partitioned System

A partition is defined as an execution environment managed by the hypervisor, analogous to a VM. Each partition consists of one or multiple processes implemented by the guest OSes. These partitions need to be virtualized in order to be able to be executed on top of the hypervisor.

The development on top of XtratuM requires to write the code in the corresponding partition. Additionally XtratuM takes control of the system at boot time and initializes the HW, after this the partition code is executed. The partition code can be classified under three categories:

- An application compiled to be executed on a bare-machine
- A real time operation system and its applications
- A general purpose operating system and its applications

The resulting execution of the partitions in each case can be once again classified as:

- **Bare Application**: Application designed to run directly on the HW, being aware of it.

- **Operating System Application**: The OS deals with the virtualization and needs to be virtualized, implies that the application is not virtualized as it runs on top of a real time OS and uses its services.

Further relevant features of XtratuM are among others the use of PV techniques, strong temporal isolation via a fixed cyclic scheduler, which makes it impossible for an application to run in parallel with another application; strong spatial isolation, meaning that partitions do not share memory, each partition has its own allocated memory and can only be accessed by the correspondent owners. It also provides a strong robust communication mechanism. [14]

XtratuM can be seen as a spatial and temporal isolation layer between the HW and the various guest OSes. [10]

Recapping the robust communication mechanism between partitions, it is crucial to underline that it is a port-based communication. XtratuM implements the channel between at least two ports. This channel can be access by the different partitions using the access points i.e. ports. Two modes are provided: sampling and queuing. It is the hypervisor responsibility to encapsulate and transport the messages.

Xtratum also provides us with a Fault Management Model. By fault we understand the event of a system trap, including HW and SW interrupts and/or processor interrupts, or an event triggered by the hypervisor. The faults are at first detected and handled by the hypervisor itself, then propagated to the corresponding partitions. Furthermore a Health Monitor is part of the hypervisor and is in charge of detecting and reacting to anomalous events or states.

As a result of the isolation of the different partitions subsystems will not be affected by a faulty partition.

### 2.4.1 System Operation

XtratuM uses a cyclic scheduler to run the various partitions, therefore the system must consist of states and transitions.

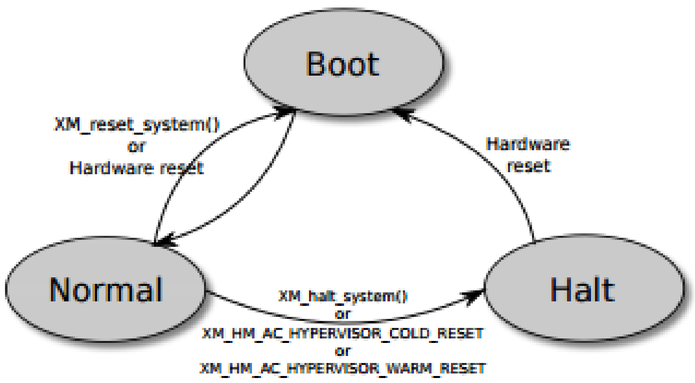


Figure 2.6: System States and Transitions [12]

The general schema of how the system behaves can be found in fig. 2.6. At *Boot*, the software loads XtratuM in main memory. During the boot time the partition scheduler is not initialized yet and as a consequence the partitions are not yet started.[12]

When the system boots successfully, the system reaches the normal state. Here the normal execution of XtratuM takes place. The scheduler is initialized and partitions start to function. Note that no input is required to get from *Boot* to *Normal* state, the transition is made automatically.

The last stage that the system is made of, is the *Halt* state. This state can be reached from *Normal* state if an error is encountered or if the system invokes the halt call.

As mentioned, during the *Normal* state the partitions work under the scheduler configuration. Again this procedure behaves accordingly to a state transition schema.
During the initialization the partitions find themselves in the *Boot* state. There the preparation of the virtual machine and the standard execution environment, consisting of communication, I/O, interrupt ports etc., occurs. After this the partition changes to *Normal* state. Again this transition happens automatically, as it is the case during the system initialization. [12]

These first two states are indifferent for the hypervisor, as both happen during the fixed time slot provided. However for better studying of the complete process the states are differentiated.

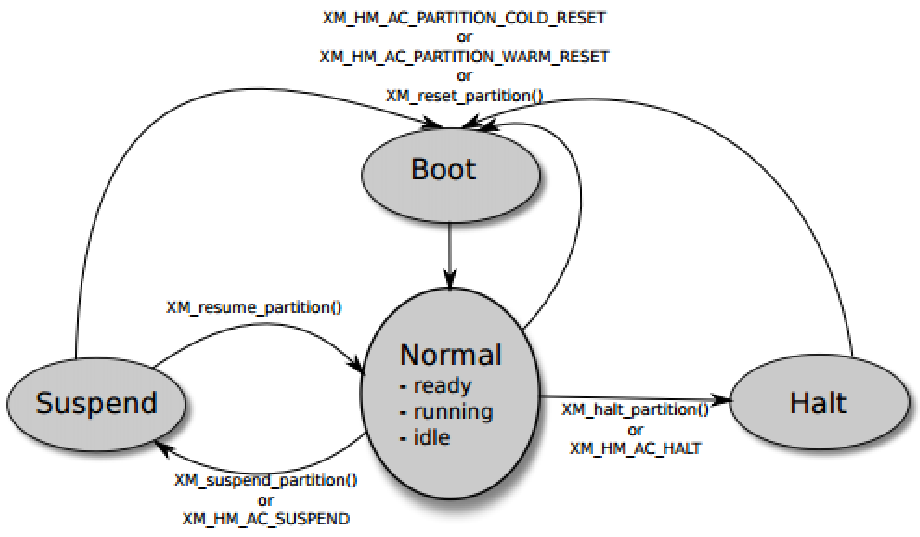


Figure 2.7: Partition States and transitions [12]

During *Normal* state, the partition is executed as programed. Three sub-states are differentiated during this state:

- *Ready* Stand-by state. The partition can be executed, but not in the own time slot. Hence has to wait for the correct time.
- *Running* the partition is being executed.
- *Idle* This state can be access using the idle call. The partition is currently in the time slot to be executed, however it does not want to execute anything, and relinquishes the processor, until the next time slot, or the appropriate interrupt.

From the *Normal* state the last two states can be reached. If an error occurs or the partition sends the appropriate call, the partition may be halted, and reaches the *Halt* state. All the resources associated are hence released, and the partition will have to boot again, as the *Normal* state is unreachable from this state.

The other state, is the *Suspend* state. The partition can be suspended and resumed by the system. During the suspension all interrupts will be buffered and are left pending.

### 2.4.2 Development Process Overview

The simplest scenario for developing XtratuM is composed of two different actors, a partition developer (PD) and an integrator (I). The interaction of these two will allow the establishment of a correct working environment for XtratuM.

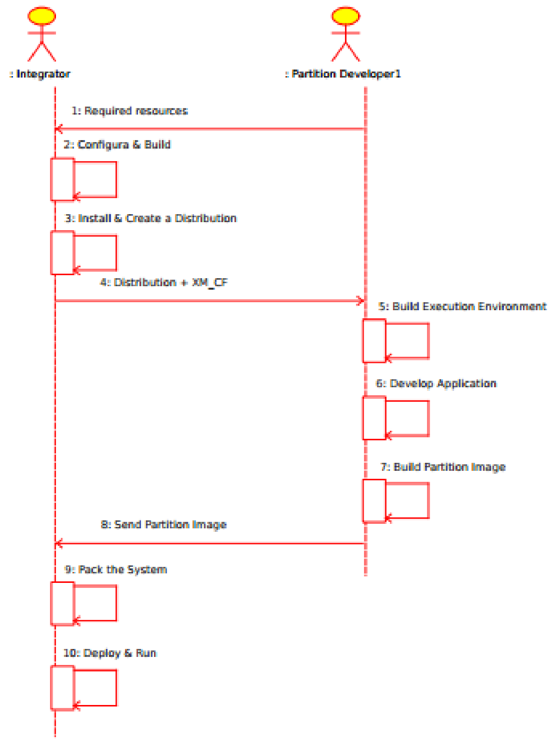


Figure 2.8: Interaction between Integrator and Partition Developer [12]

The above figure shows the task's flow and interaction of both parties. Firstly the PD defines the required resources and sends this data to the integrator. With this information the integrator configures the XtratuM source code, adapting it to the requirements, and builds the hypervisor binary, user libraries and tools (*menuconfig*).

Additionally the integrator then allocates the available system resources to the partitions. This is the creating the *XM CF* configuration file, where memory areas, scheduling plans, port and channel creation among other resources are detailed described for further processing.

These resulting binaries are then sent to the PDs. These are unique and if more than one partition developer is in action, all have to receive and use the same binaries. With such the PD can develop and build the execution environment.

Once the environment is built, the PD can move on to creating the partition application. An image of this is created and sent back to the integrator, which will pack the gathered image together with the resident software, the binaries, partition and configuration files, so that the system can be deployed and run.

Even though the idea behind this procedure may result easy to understand, each step described, includes multiple sub-steps and the use of the extensive XtratuM tool's library. More detailed procedure can be found in the section 5.3.

## 2.5 Middleware

The middleware can be described in its wide sense as software glue. It lies between the OSes and applications and enables multiple components of a system to communicate and share data.

### 2.5.1 Internet Communication Engine

Ice is an object oriented middleware platform. It provides tools, APIs and library support to be used in heterogeneous environments, being unnecessary that both emitter and receiver are written in the same programming languages. It can be run on different OSes and machine architectures, communicating using various networking technologies.[15]

Each Ice object contains an interface with a certain number of operations. These operations, as well as interfaces and data types are exchanged between both ends using the Slice language.

This custom language fulfills the heterogeneous environment support by defining a client-server contract, which is compiled for a specific programming language correspondent to a specific interface defined, the so called

Slice.

The translation of Slice into the different programming languages is called Language Mapping. Currently Ice supports C++, C#, Java, JavaScript, Python, Objective-C, PHP and Ruby. This language is purely declarative and no statement for execution can be written in Slice.

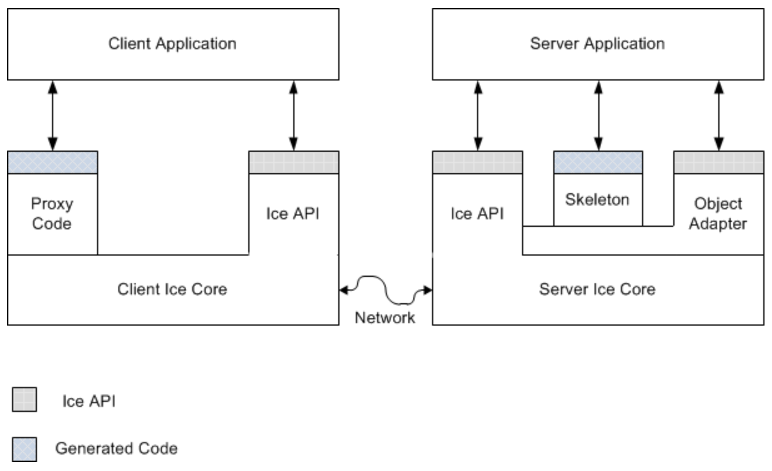


Figure 2.9: Ice [15]

Taking a deeper look into the Client-Server Structure, we can conclude that both consist of a mixture of application, and Slice generated code.

- The Ice core provided as a number of libraries, contains the runtime support for remote communication.
- The part that is independent of the specific types defined in Slice is called the generic part of the Ice core, and is accessed through the Ice API. This is used to take care of administrative chores such as initializing and finalizing the Ice runtime.
- The object adapter is part of the Ice API for the server side. It has 3 major functions. It is responsible for the creation of proxies. It maps incoming requests to specific objects. And it can be associated with more than one transport endpoint (for example with TCP and UDP).
- The proxy code is generated from the Slice definition. It's two major functions are on the one hand providing a down call interface for the

client which ends up in sending an RPC message that invokes the corresponding function on the target object in the server; and on the other hand providing *marshalling* and *unmarshalling*[1] code.

- The skeleton code is generated from the Slice definition. It is equivalent to the proxy code, but on the server side. It provides an up-call interface that permits to transfer the thread of control to the application code.

As the client sends an RPC message, it is to underline that Ice provides an RCP protocol that can use indistinctly TCP or UDP as the underlying transport. Hence no real time?. SSL for encryption is also supported. Briefly the Ice protocol defines a number of message types, a protocol state machine that determines the sequence of message exchange, encoding rules, and a header. Compression with a configuration parameter is supported to conserve bandwidth, as well as bidirectional operation is supported.

Ice ships with various services that provides us with the sophisticated platform for distributed application development. This services are worth mentioning, Freeze and FreezeScrip, IceGrid, IceBox, IceStorm, IcePatch2 and Glacier2. IceStorm, IceBox and Glacier2 will be used to perform further development.

This platform provides several benefits for developers. Support for multiple interfaces, as well as synchronous and asynchronous messaging. Machine, language and transport independence and security among others.

### 2.5.2 IceStorm

IceStorm is an efficient publish subscribe service for Ice applications in need of multicasting or even broadcasting information to multiple destinations. For a better understanding of the concept behind publish and subscription we can take a look at the following example.[15]

Suppose we take a pitot tube monitoring application. Measurements such as wind speed and temperature are collected periodically from a control center. This information must then be transmitted to various monitors, for further processing of such. The main performance disadvantage of this

[1] “Marshaling is the process of serializing a complex data structure, such as a sequence or a dictionary, for transmission on the wire. The marshaling code converts data into a form that is standardized for transmission and independent of the endian-ness and padding rules of the local machine. Unmarshaling is the reverse of marshaling.”

scenario and implementation is clear: the collector must not only collect the data, bus must then redirect the data accordingly to specific parameters, to the correct destinations. Hence, data collection, monitor registration, data delivery, and error recovery must be handled by the collector.

Using IceStorm as a gateway between the various monitors, allows the collector to get rid of the extra duties, and ensures just a single collector functionality.

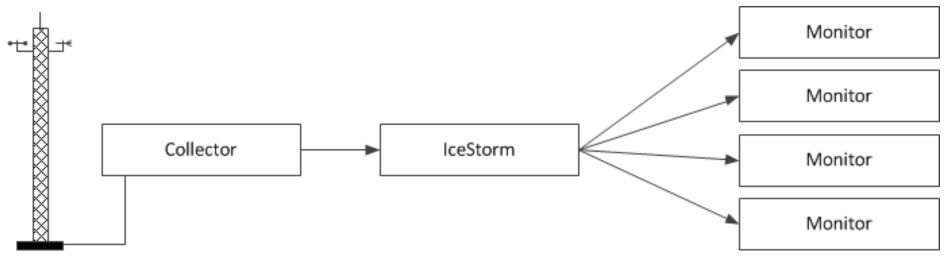


Figure 2.10: IceStorm [15]

The IceStorm Service simplifies substantially the implementation of the collector. It acts as a mediator between the collector and the monitors, publisher and subscribers respectively. Various advantages are offered by this service:

- Collector makes only a single request on the IceStorm server when a new set of data is ready to be transmitted. The IceStorm server takes over all pertinent actions to forward this data to the subscribers.

- Subscribers only communicate with the IceStorm server for subscribing and unsubscribing tasks. Enlightening off administrative tasks from the collector.

- Smooth simple actions must be taken when deploying IceStorm in both, publisher and subscriber.

In order to fully understand the IceStorm's capabilities we can discuss some fundamental concepts:

- Message: It is represented by an invocation of a Slice operation. The operation consists of a name, and some parameters. The name defines the type of operation whereas the parameters define the message content. If we invoke such operation on an IceStorm proxy, the message is then published. Analogous the subscriber receives the message as a servant up call. It is important to note that IceStorm uses a push model for the delivery of messages. Polling is not supported.

  - IceStorm Messages are strictly unidirectional. They have a void return type and hence cannot contain out parameters nor raise user exceptions. The Publisher cannot receive messages.

  These messages are automatically discarded once they are published to the correspondent subscribers. If an error occurs during the transmission a copy of the message will not be queued. Additional IceStorm will automatically remove the subscription from the topic.

- IceStorm Topic: is essentially equivalent to an application-defined Slice Interface. In order to understand better this concept we can go back to a traditional client server style. This application interface is hence the contract/protocol between the server and the client. If an application is interested in a specific type of message it can subscribe to such topic. The publisher will then send the message and the subscriber must hence implement the topic interface in order to obtain that type of message. It is important to underline that IceStorm forwards each message to multiple subscribers. Furthermore IceStorm does not support a verification of the compatible interfaces on both ends.

- Federation: Also known as topic graphs. A Federation or topic graph is formed by creating a unidirectional link from one topic to another. Each link carries a cost. If the message cost of exceeds the link cost, the message will not be published.

The following figure represents an example of what a Federation could look like.

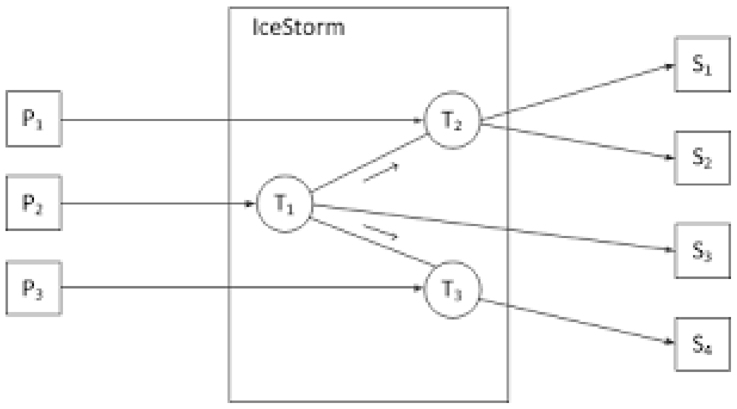


Figure 2.11: Ice Federation [15]

This example combines a collection of three different topics, T1, T2, T3, with their correspondent publishers, P1, P2, P3, and a total of four subscribers. We can see that T1 is linked to T2, and T3 respectively depicted with the arrows. S1 and S2, as depicted, are subscribed to T2, but due to the link, they will receive the messages from T2 as well as T1. The same occurs with S4, it receives the messages from T3 and T1 even though it is only subscribed to T3. Moreover S3, only subscribed to T1, will only receive the message from that topic.

From this example we can extract two pieces of additional information. Firstly the publication of the messages to the different destinations happens hop by hop, i.e. T1 will first publish the message to the S3, and then it will publish it to its links. This implies a delay in the transmission that was introduced as link cost. Secondly IceStorm is not optimized and will send duplicate messages from different topics, depending on the topic federation, to the same subscriber. In our example S5 will receive the message from T1 twice as it is subscribed to T2 and T3.

IceStorm supports two different modes of behavior, the persistent mode, and the transient mode. By default IceStorm operates in persistent mode. This mode implies a database where IceStorm stores all the information related to the topics, links and subscribers. Replication, which provides higher availability, is supported in this mode.

Alternatively IceStorm can run without a database, in transient mode. Same operational behavior but without support of replication.

#### Highly Available IceStorm

Furthermore, IceStorm provides a highly available mode. It uses the replication of master-slave with automatic recovery in case the master would fail. The idea behind this mode is elaborated by using the Garcia-Molina "Invitation Election Algorithm".

This algorithm sets a priority to each replica and sorts it into a replica group. Inside that group the replica with the highest priority becomes the master or coordinator, turning all other replicas into slaves of it. All these replicas are configured in a static way containing information about the rest of the replicas. This way when overcoming an error, the replicas can group themselves back together, knowing the information of each other.

In addition, the masters will periodically be looking out for replicas and replica groups pursuing the aim of forming larger, but fewer groups. Moreover, the slaves will also be periodically contacting their corresponding master to check whether it is the correct master of the group. The replica will turn its state into error otherwise, performing hence the recovery procedure.

The only limitation of this algorithm while used with IceStorm is majority. A group of replicas must contain the majority of the replicas, implying a minimum of three replicas. This is necessary if network partitioning wants to be avoided. During the full system startup, the replication starts only when every replica in each group participates. When a majority group is formed, the databases contain the states are compared. The latest one is hence transferred to all replicas and replication can begin to function properly.

The IceStorm replicas can have four different states which can be divided into node states, or group states. Nodes can either be *inactive*, when awaiting an election, or *election* when electing a master or coordinator. On the other hand, the groups can have the state of *reorganization*, or the *normal* state which implies a normal active replicating status.

### 2.5.3 Data Distribution Service (DDS)

The DDS for RTS is a data communication standard managed by Object Management Group (OMG) that provides scalable, low-latency, high performance, interoperable data exchange for distributed applications. It is suitable for real time and near real time systems.

The DDS Standard includes a well defined API that allows the creation of portable code. The DDS standard references the Real Time Publish Subscribe (RTPS) Wire Protocol standard which defines the wire protocol for DDS communications. Generally speaking DDS is a p2p communication model requiring no gateway, server nor daemons that have to be running nor configured.

For developing purpose I have chosen OpenDDS, which is an open-source version supporting the capability defined in the DDS 1.2 Specification and version 2.2 of The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS). However the term DDS will be used instead of OpenDDS for simplicity reasons.

A very basic conceptual view of the mentioned Publish Subscribe Architecture can be seen in the following picture.

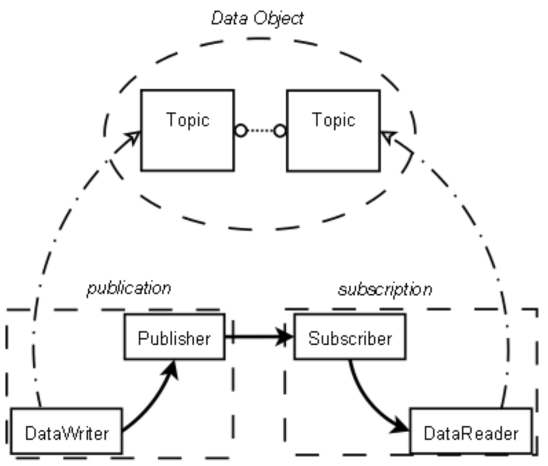


Figure 2.12: DDS [12]

The basic components are: Topic, Publisher, Subscriber, DataWriter and DataReader.

Topics contain information about a single data type, the distribution and availability. They can have more than one DataReaders and DataWriters.

Publishers apply control and restrictions to flow of data from DataWriters. Analogously Subscribers apply control and restrictions to flow of data from DataReaders. The DataWriters create and DataReaders receive Samples (Data values) of a single application data type. Both can only have a single Topic.

A Publication or a Subscription can have many associated Subscriptions or Publications receptively.

It is important to note that an application can be either a Publisher a Subscriber or eventually both. Hence the data flow can be easily understood by following the arrows in the conceptual diagram. The publishing side initiates the flow of data by writing a Sample into the DataWriter. It gets published, and sends the Samples to the associated subscribers. These input the received Sample into the DataReaders. The flow ends when the subscribing applications retrieves the data from the DataReaders. The flow of data is controlled by QoS Policies. Aspects of QoS can be Real-time deivery, Bandwidth Redundancy or Persistance.

The association of a publication and a subscription exists only when a DataReader's Topic is compatible with a DataWriter's Topic. Then the Publication and Subscription become associated and data is published between them.

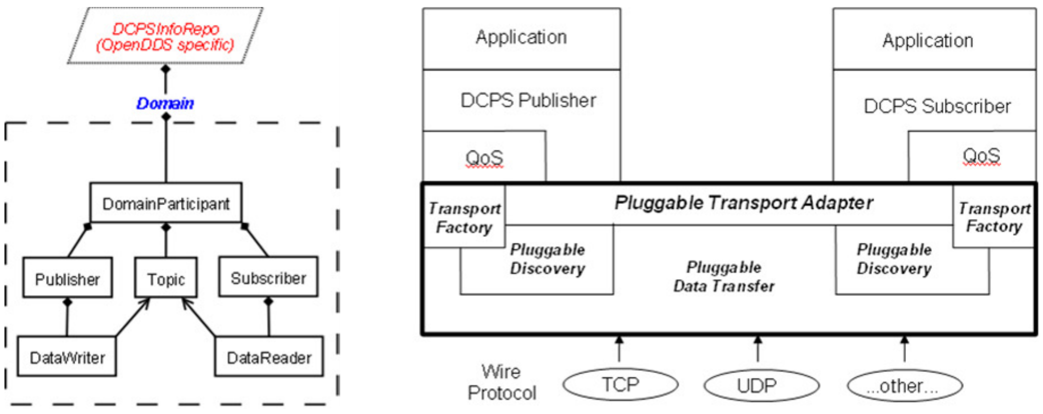


Figure 2.13: DDS [12]

However if we look at the actual components in a system we can find two new specific components. The domain which is a conceptual container of the system. The communication can only occur between components within the same domain. Note that all components inside a domain are called Entities.

The DCPSInfoRepo which is an OpenDDS object that detects when association can occur and notifies both parties to make the association.

The specification to enable interoperability amongst DDS implementations was initially called RTPS, now it is know as DDS Interoperability Protocol (DDSI). In our case, OpenDDS supports unicast as well as multicast. The transport design is particular to actual domains. Usually some of the already existent transports are combined with higher lever protocol and must be accommodated for the specific use.

Again in our case OpenDDS separates the transport from higher level protocols by means of Extensible Transport Framework (ETF) [11]

## 2.6 Research Context

The presented bachelor thesis is framed in the research context of real time distributed systems and cyber-physics. More precisely it is based on the challenges, tasks and solutions identified in [16] for virtualization technologies in predictable cloud computing, and in [19] for middleware technologies in cyber-physic systems.

The middleware Ice, specially, was improved in various contributions proposed by a group who has developed a project [30] in which different parameters were adjusted to improve performance. Moreover, in the project [31] a centralized architecture for real time distributed systems was made to support a variable greater number of clients. Proposals for online verification of cyber-physic systems connected as in [33], [45] and [34] as well as middleware with augmented logic in order to support changes in the dynamic structure of real time distributed systems in service [17], [36], [35], [37].

This project is enclosed within the efficient management of the execution platform resources, not only in the participating nodes but also in the interaction and communication between the nodes by means of the improved middleware technology. Likewise, it is enclosed within the dynamic management which requires augmented logic [46].

With the management of the resources in the participating nodes, a better exploitation of the computational resources (CPU, memory, energy, etc.) is obtained using priority and temporal planning managers as in [?], [28], [29], [26], [27], [?], [20], [21] or [43]. In the management of the communication between the nodes based on middleware, for a system with time requirements, or in general for real time systems it is necessary to analyze the reliability of the platform, its performance and temporal stability. In this context the following projects have been developed. An application for rail traffic and its use in laboratories is orchestrated in [38]. In [41] a surveillance video application, activated by sensors and supporting structural changes in real time was design. The design and implementation of a system for the composition of services based on Jini, Java technology, can be found in [42]. In [44] a study about DDS for paravirtualization in virtualBox can be found. A bridge for adapting middleware to different communication paradigms is presented in [39]. And lastly, augmented interconnection between iLand and the distributed annex Ada (Ada DSA) is presented in [40].

This bachelor thesis focuses to an extend with these contributions, which establish the research environment developed throughout this project.

Firstly, it is related with the performance analysis of middleware for distributed environments with time requirements. Secondly, it is also strongly related with the operating system, as it is fundamental to know certain details about this component in order to be able to add pertinent improvements to the middleware. This latter, analyzes the CPU consumption affected by the physical memory in order to obtain the load limits of the machines, above which anomalies in the middleware are encountered.

And lastly it is related with the dynamic management of the structure and reconfiguration of safe distributed systems with a brief insight into quality software management [48], [47].

# Chapter 3

# Project Development: Xen

The first step of the development is to install and configure the hypervisor Xen. This chapter explains the whole procedure that must take place in order to achieve a clean correct installation of a distributed system. Detailed descriptions of the different configurations can be found in the previous sections.

The performance analysis will then be executed on this configured platformed. By creating a unique and fully tailored system, achieving a perfect and complete overview of what the configurations, any external alterations that may influence the results, are minimized.

## 3.1 System Requirements

In order to successfully achieve the aims presented, minimum software and hardware requirements set by Xen must be met. Table 3.1 contains these requirements.

### 3.1.1 Software Requirements

Starting off with the software requirements, it is clear that the use of Xen is a popular election when making use of virtual technology. Additionally a UNIX distribution is required to run the configuration domain as well as the guest domains. This way a proper paravirtualized system can be implemented.[15]

**IceStorm Requirements**

In order to be able to use Ice‘s Service IceStorm, a system with either UNIX or Windows Operating System is required. For this service simple sockets in the network layer will be used. Moreover no different requirements than the installation of the Zeroc Ice SDK, and the Ice's services is necessary.

In this case the C++ SDK will be installed, and hence an appropriate compiler is necessary in order to program,, build and execute the code in matter. The friendly GNU C++, *g++* compiler is used in this case.

### 3.1.2 Hardware Requirements

The hardware that is needed to run such software is depicted in the table 3.1 where the two columns hold the correspondent described hardware.

On the one hand Ubuntu requires a standard processor with at least 700 MHz of clock-speed. 512 MB of RAM and 5 GB of free HD space.

On the other hand Xen does not restrict the clock-speed but rather the architecture of processor. Only x_86 and ARM processors are supported for the virtualization. As far as the physical memory, it's requirement doubles the one from the Ubuntu distribution and rises to 1024 MB. [6]

| | Ubuntu | Xen |
|---|---|---|
| CPU | 700 MHz | x_86, ARM |
| RAM | 512 MB | 1024 MB |
| HDD | 5 GB | - |
| Graphic | VGA 1024 x 768 | Intel/AMD-V |

Table 3.1: Minimum System Requirements

As the domains will run Ubuntu, its specifications must be met within the correspondent virtual machines. Additionally the requirements for Xen has to be taken into account. If a top-down implementation approach is taken, all together the System must have at least *1024 Mb of RAM*, run under at least *700 Mhz CPU* and reserve at least *5 Gb of free disk storage* the for every domain planned to be installed. The system should have Intel/AMD Virtualization Technology for Directed I/O support.

A physical network card, even though a basic component found in almost any system nowadays, is essential to perform the desired tests. Without a network card, the network virtualization among the different virtual machines could be affected in terms of could be achieved, that is no virtual link could be establish

## 3.2 Design

In order to achieve a close and deep interaction with the virtual environments, two working nodes with different virtualization layers have been implemented. The main working station, named "Computer A" onwards, is based on an HP EliteBook 6930p laptop, whereas the second working station, named "Computer B" onwards, is an up to date modern Lenovo E460 laptop. The hardware specifications of both working nodes can be analyzed in the following table. Clearly they meet the minimum required specifications stated in the previous section.

| | Computer A | Computer B |
|---|---|---|
| | Intel Core 2 Duo P8700 | Intel i5-6200U |
| CPU | 2x2.6 GHz | 2x2.8 GHz |
| RAM | 1.5 GB | 16 GB |
| NIC | Intel 82567 | Intel AC-3160 |
| Max. Speed | 1 Gbps | 150 Mbps |

Table 3.2: Hardware Specification of Working Nodes

Further on in the *Trials section* the different configurations of the different environments will be explained. Note that the choice of using two different working stations connected with each other allows to have all possible different scenarios when it comes to performance testing. Summing up, in basic terms, the final distribution of the design is observed in the following figure:

## 3.3 Architecture: Computer A

As mentioned above, in order to achieve a wide interaction with the virtual environments, this working station consists of the deployment of a Xen 4.4 hypervisor, which acts as the bonding layer between the hardware

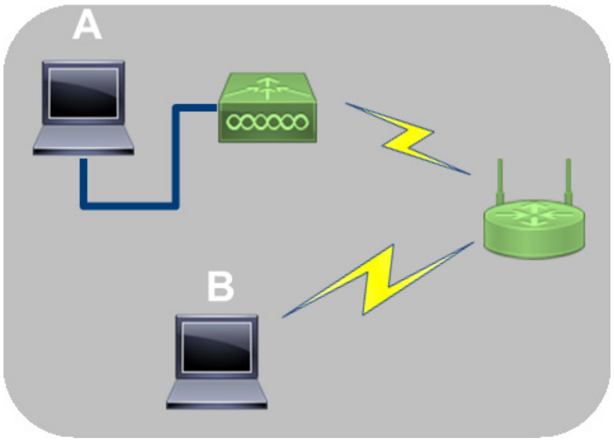
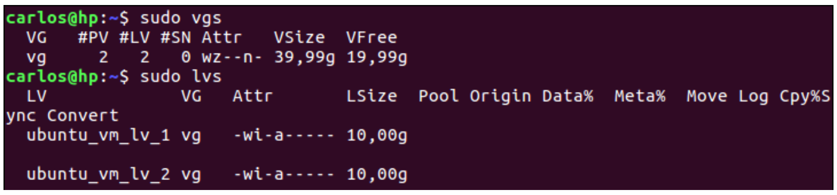


Figure 3.1: Basic Design

and the different virtual machines, depicted in Fig. 3.3.

Computer A is platformed with the latest Ubuntu Xenial Xerus distribution (version 16.04). As the Xen hypervisor is included in the standard Ubuntu repository, it can be simply downloaded and installed as if it would be a regular software. In this case the 64-bit hypervisor was installed, and even though the dom0 works only with a 32-bit kernel, this version will allow to run 64-bit guest machines.

As each different domain requires at least 5 GB of storage capacity, two different 10 GB partitions of the physical HDD are made in order to hold the different domains.



```
carlos@hp:~$ sudo vgs
  VG   #PV #LV #SN Attr   VSize  VFree
  vg     2   2   0 wz--n- 39,99g 19,99g
carlos@hp:~$ sudo lvs
  LV             VG   Attr       LSize  Pool Origin Data%  Meta%  Move Log Cpy%S
ync Convert
  ubuntu_vm_lv_1 vg   -wi-a----- 10,00g

  ubuntu_vm_lv_2 vg   -wi-a----- 10,00g
```

Figure 3.2: HDD Partition Table

Moreover in order to achieve connectivity between the different guest machines and host, a further network configuration has to be implemented in order to guarantee such. This can be achieved using the *bridge-utils*. See section 3.5 for further information.

For this purpose, two additional virtual machines have been configured and installed. The installation procedure of these guests is described in section 3.6.

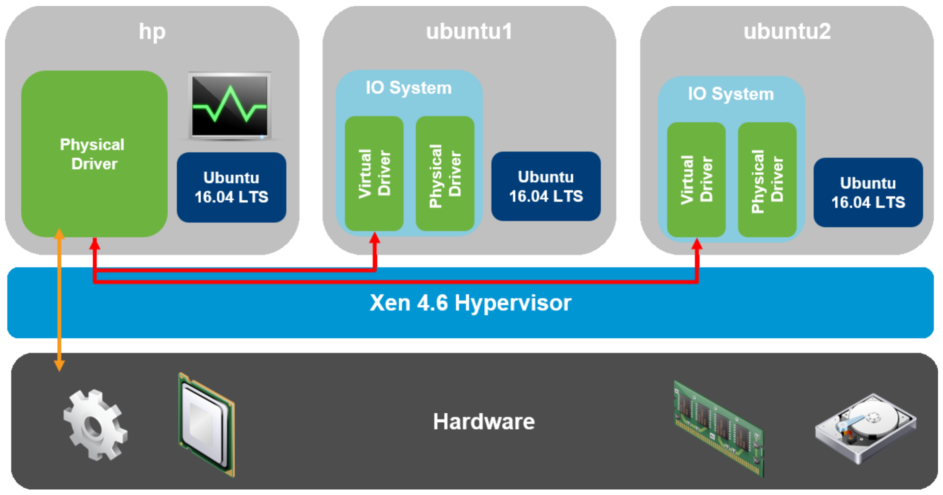


Figure 3.3: Final Architecture Computer A

The figure above shows the exact final architecture of Computer A, a bottom layer of pure hardware contained in the EliteBook. The middle block is the hypervisor which supports the dom0 and two further guest machines. All three of these virtual machines hold the same OS.

## 3.4 Architecture: Computer B

This working station named Computer B, is perhaps more complicated and complex from a virtual point of view. The base system is a normal Windows 10 Education edition, installed on a Lenovo E460 laptop.

Inside Computer B, Oracle VM Virtual Box 5.0. is installed in order to be able to have an UNIX environment. In this case Ubuntu 14.04 LTS is installed as a virtual machine using the KVM virtualization offered by Oracle VM VirtualBox.

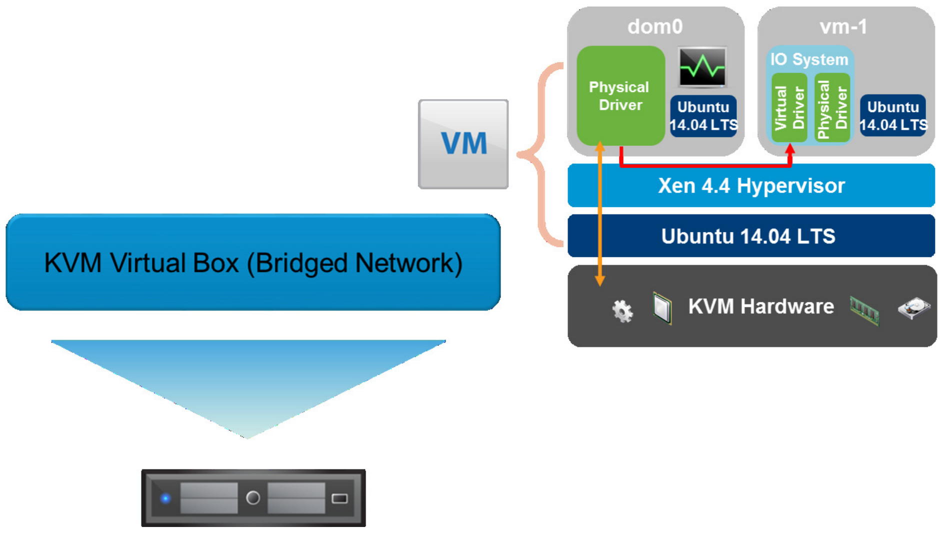


Figure 3.4: Final Architecture Computer B

The network configuration of the VirtualBox virtual machine is setup in the *Bridge* mode so that no further configuration has to be made in the adapters.

From this point onwards in the configuration, the same procedure was followed as in Computer A to install Xen with the control domain *dom0* and, in this case, one guest domain *vm-1*.

## 3.5 Network

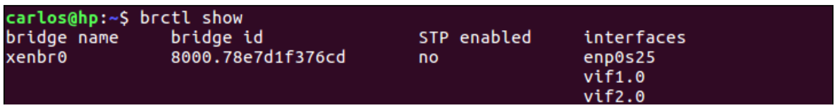
Unfortunately one of the constrains associated to the usage of the Xen hypervisor, is the difficulty encountered when virtualising a wireless connections. Therefore in order to avoid additional hurdles with the configuration, wired connections are used.

As the guest domains have been configured following paravirtualization, they have access to their virtual network interfaces. Rather than emulating the real network devices, these virtual interfaces, *vifx.y*, provide a much faster and more efficient network communication for the domains.

A paravirtualised network device consist of a frontend and a backend network device:

- The **backend** network device resides in the domain *dom0* and it contains the domain ID and the index of the device, x and y respectively.
- The **frontend** network device is located in the guest domain and acts basically as any normal network interface. It is bounded to the *xen-netfront* driver and creates the traditional ethN network device.

These devices are interconnected via a virtual link. Furthermore a connection between the virtual devices and the real device must be established. This can be achieved using bridging.

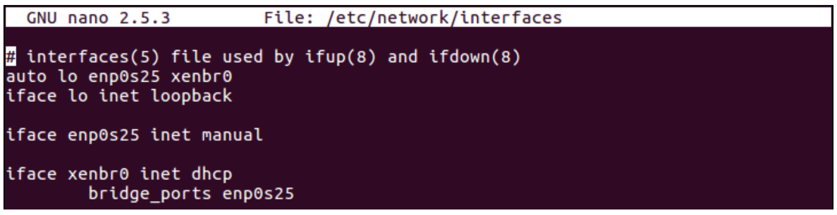


```
carlos@hp:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
xenbr0          8000.78e7d1f376cd       no              enp0s25
                                                        vif1.0
                                                        vif2.0
```

Figure 3.5: *$ brctl show* Output

Bridging the backend domain will allow the frontend domains to become independent members of the network.

In order to use this configuration a software bridge, acting as a virtual switch, is created so that it links the backend devices with the physical device again via a virtual channel. To do so the *interfaces* configuration file found under */etc/network/* must be modified adding the bridge interface and setting up the bridging.



```
  GNU nano 2.5.3           File: /etc/network/interfaces

# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo enp0s25 xenbr0
iface lo inet loopback

iface enp0s25 inet manual

iface xenbr0 inet dhcp
        bridge_ports enp0s25
```

Figure 3.6: */etc/network/interfaces* configuration file

This bridge is then set up in the guest configuration file to create the guest domain. As it is set to DHCP, the guest domain will be provided with an IP address within the same network as the host.

For this specific case depicted in Fig. 3.1 both machines A and B were configured in such a way so that each domain can be reached independently of the other domains. A schematic of the internal network devices follow:

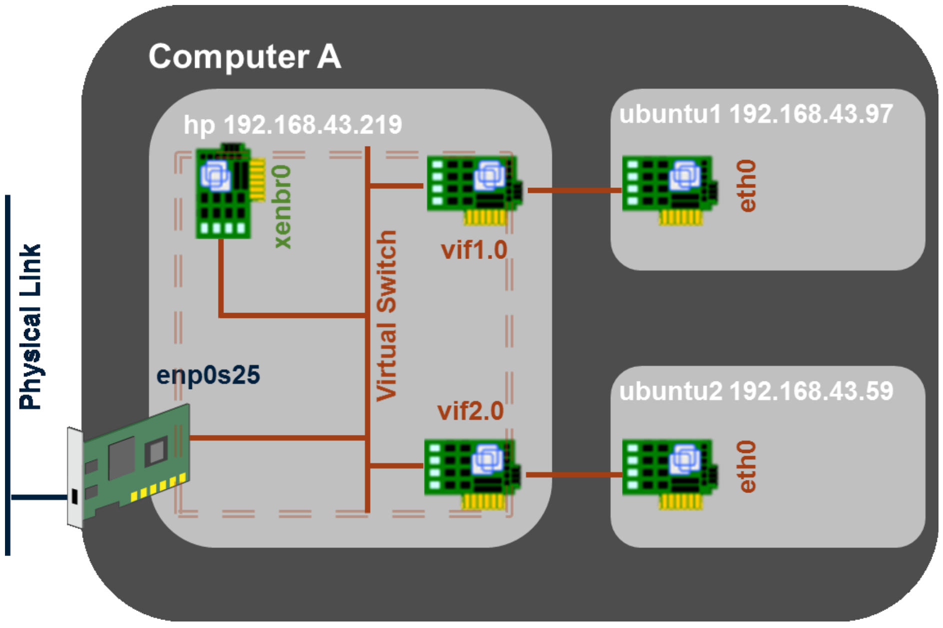


Figure 3.7: Computer A: Network Devices Structure

As explained above, each paravirtualized connection is made up of the two different virtual devices, all which are then interconnected via a virtual link. For computer A where there are two different guest domains, the bridge is connected to both virtual interfaces. The figure above clearly shows the final setup of the network layer.

Below, analogously to computer A, the configuration of computer B is depicted. In this case only one guest domain is being used, hence we basically have the same configuration as computer A, just with one paravirtualized connection less.

It is important to note that for simplicity purposes all incoming, outgoing and forwarding connections in the respective firewall have been enabled to prevent conflict while doing the trials.

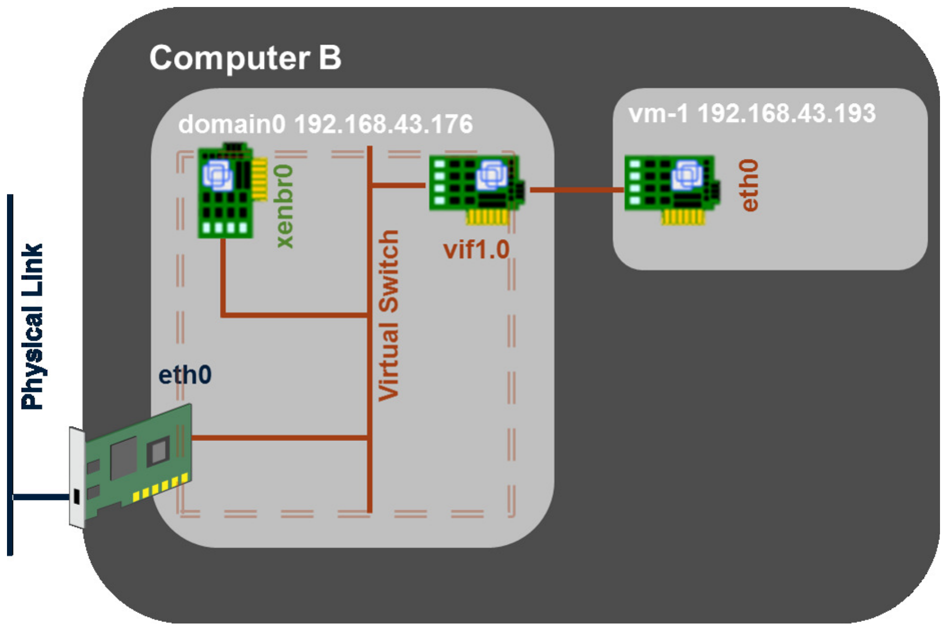


Figure 3.8: Computer B: Network Devices Structure

## 3.6 Xen: Domain Creation

The Xen hypervisor can be installed through the UNIX terminal console without any further difficulties. Note that the virtualization must be supported by the system's chip, and accordingly enabled in the BIOS. If any errors during the installation would occur, the installation process will exit returning the correspondent error.

If the hardware specifications meet the minimum requirements for Xen. section 3.1.2, no further problems should be encountered during the installation process.

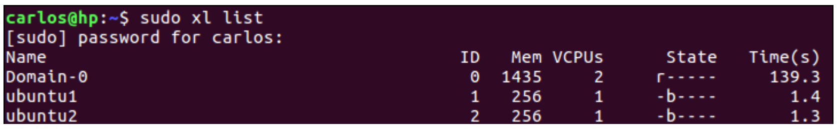
### 3.6.1 Control Domain: Dom0

To install the Control Domain, simply using the following command should suffice:

```
$ sudo apt-get install xen-hypervisor-amd64
```

Once the hypervisor is installed the GRUB will automatically boot the system into the host machine, known as control domain, *dom0*. After a reboot the correct configuration of the system can be checked with the *xl list* command. Now the system is ready to accept the installation of guest virtual machines.

It is an easy procedure as no further configuration steps are required to achieve the installation of the control domain. The following subsection explains in detail how to install the guest domains. After the successful installation of such, for Computer A the following output to the *xl list* command is observed:



```
carlos@hp:~$ sudo xl list
[sudo] password for carlos:
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0  1435     2     r-----     139.3
ubuntu1                                      1   256     1     -b----       1.4
ubuntu2                                      2   256     1     -b----       1.3
```

Figure 3.9: *xl list* Computer A

### 3.6.2 Guest Domains: DomU

According to the community documentation there are two different ways creating a Virtual Machine within Xen:

- **Automatically**: pre-build guest images can be downloaded from project based sources
- **Manually**: A set of tools such as virt-builder, which is part of libguestfs; virt-manager, belonging to libvirt; and xen-tools allows to genereate and build a customized VM.

This later procedure can be divided into two different virtualization modes that Xen supports: PV and HVM.

Knowing the differences in these two virtualization principles, it is worth mentioning that HVM will be more complex in the setup phase. Devices such as Ethernet or ATA/SATA have to be emulated while the CPU and memory has to be virtualized using hardware if good performance is to be achieved.

Moreover it is suggested to use PV drivers within the HVM domain as the default emulated devices tend to be very slow. In the sight of the above the main focus of this matter sets back to Guest PV. The process of creating a PV virtual machine can be summarized in the following chart:

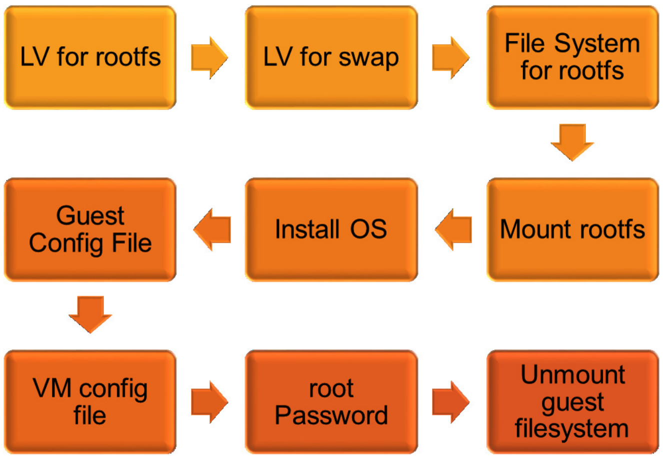


Figure 3.10: Guest Domain Installation Flowchart

*xl-utils* tool stack simplifies the whole procedure and by simply writing out the correct configuration file a guest domain can be created in matter of minutes.

Starting off by getting the Netboot Images for the desired distribution and saving them on the system. With this information the guest domain configuration file can be created, for demonstration purposes, the same file as the one used in the configuration of one of the guest images in Computer A architecture is used.

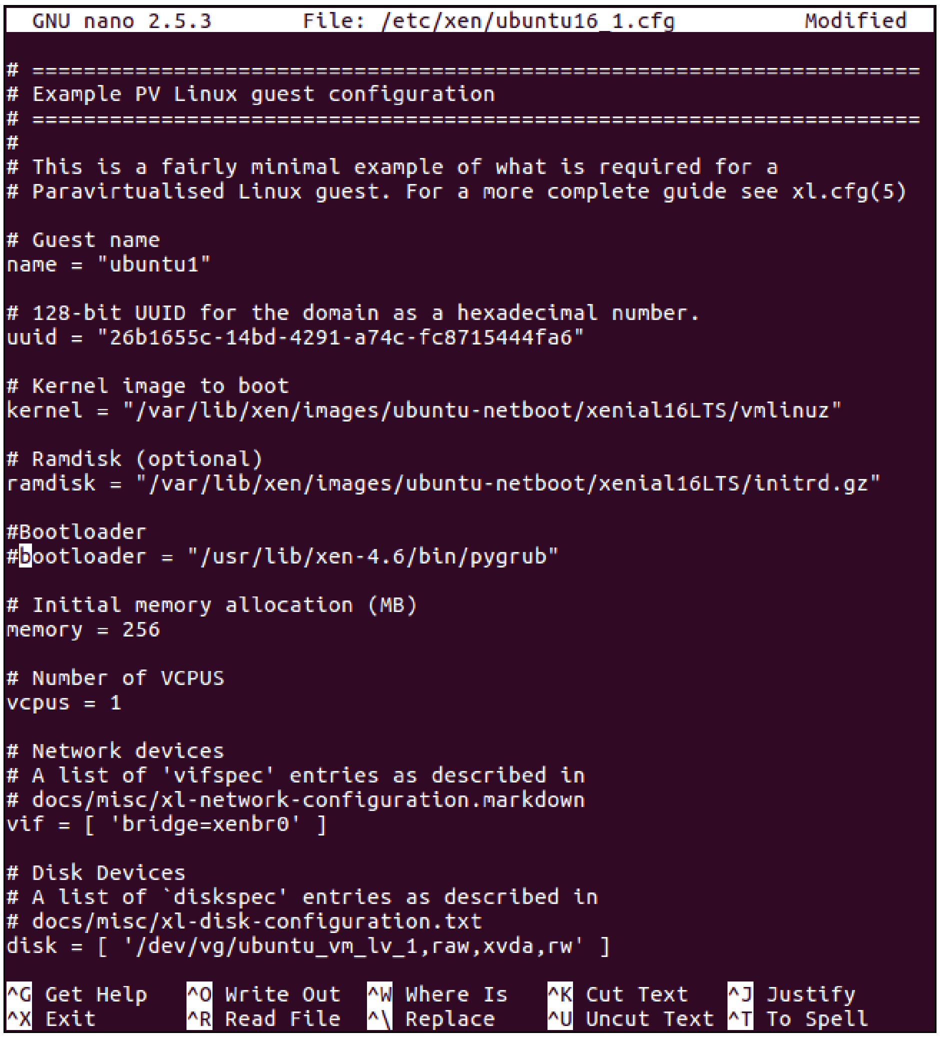
In this configuration file, several information about the guest domain must be introduced. Name, UUID for the domain are essential pieces of information, but more important is initially to configure the kernel image to boot together with the ramdisk path (previously downloaded for the desired distribution).

Hardware specs such as the initial RAM allocation and number of virtual CPUs, together with the network and disk device must be represented in the configuration file.

Using the simple *$ xl create -c name.conf* command, Xen parses the configuration file and starts off the installation procedure in a normal manner.

Once the installation is completed without any errors, the configuration file must once more be modified in order to point to the hypervisor the path to the bootloader and to remove the kernel and ramdisk lines to prevent the domain to boot into installation mode ever again.

This simpler procedure could be achieved without the help of the Xen utilities. Nevertheless it can be concluded that the existence of a high error risk, points towards the use of these tool utilities which simplifies and guarantees a quick and simple install.



```
  GNU nano 2.5.3          File: /etc/xen/ubuntu16_1.cfg          Modified

# ======================================================================
# Example PV Linux guest configuration
# ======================================================================
#
# This is a fairly minimal example of what is required for a
# Paravirtualised Linux guest. For a more complete guide see xl.cfg(5)

# Guest name
name = "ubuntu1"

# 128-bit UUID for the domain as a hexadecimal number.
uuid = "26b1655c-14bd-4291-a74c-fc8715444fa6"

# Kernel image to boot
kernel = "/var/lib/xen/images/ubuntu-netboot/xenial16LTS/vmlinuz"

# Ramdisk (optional)
ramdisk = "/var/lib/xen/images/ubuntu-netboot/xenial16LTS/initrd.gz"

#Bootloader
#bootloader = "/usr/lib/xen-4.6/bin/pygrub"

# Initial memory allocation (MB)
memory = 256

# Number of VCPUS
vcpus = 1

# Network devices
# A list of 'vifspec' entries as described in
# docs/misc/xl-network-configuration.markdown
vif = [ 'bridge=xenbr0' ]

# Disk Devices
# A list of `diskspec' entries as described in
# docs/misc/xl-disk-configuration.txt
disk = [ '/dev/vg/ubuntu_vm_lv_1,raw,xvda,rw' ]

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell
```

Figure 3.11: *ubuntu16_1.conf* Guest Domain Configuration File

# Chapter 4

# Trials: Xen

A series of performance test has been carried out to obtain the various delays that may incur when working with virtual environments. As mentioned previously four different environments were used during the trials.

As the application used for the trials is the IceStorm service provided by Zeroc Ice, needs at least one *Publisher*, one *Subscriber* and the local *Icebox* service to run, these services will run on different locations to determine the delay times in the different configurations.

In all four configuration scenarios, the same procedure takes place:

- **IceBox** server will be running from one of the domains, and will allow the interconnection between the publisher and the subscribers.
- **Publisher** instance will be running from one of the domains. It will broadcast every 1 second a message with a specific topic name and desired data size in Bytes. Additionally the system publication time will be sent as part of the message.
- **Subscriber** instance will be running from one of the domains. It will subscribe to the subscriber's topic, and hence receive the messages, a simple operation will extract the time sent by the publisher and calculate how long it has taken to transmit the message.

To standardize the trials, in each scenario the same performance tests will be performed. Publishing 1, 1024 and 6144 bytes of data under a normal CPU load, and while the system is undergoing a stress situation. A further 69 Bytes of data containing time values, for further calculations, is also sent, but is tared to achieve standard results for the amount of data described.

The stress on the system will be performed with the *stress* workload generator. It imposes a configurable amount of CPU, memory, I/O, and disk stress on the system.

*$ stress –cpu 8 –io 4 –vm 2 –vm-bytes 128M –timeout 10s* as an example.

In order to keep the analysis consistent, *stress* was used with different configuration parameters so that the CPU is 100% loaded and the physical memory reaches a utilization quota of around 77%.

The analysed period of time records about half a minute. During this time, ten time samples have been taken into account and averaged. Each time sample measures the transmission time a topic suffers since it is published until it reaches the subscriber. From this data, the most significant figures, such as the average, maximum and minimum has been recorded for further analysis. The detailed table with the exact times can be found in the Annex A.1.

Represented in the graph figures of this chapter are the average time measurements obtained as mentioned above. In the legend of the figures *normal* represents the transmitted time under minimum system load. *Loaded* on the other hand measures the transmissions times when the system undergoes the call of the *stress* function.

## 4.1 Configuration A

In this configuration only *Computer A* is used. As we can see below both instances of the IceStorm service as well as the IceBox service run within the same domain, the control domain *hp*.
This configuration will resemble the best case scenario, as no connection with other guest domains or machines is required. It will be used as a reference for making conclusions of further analysis.

### 4.1.1 Results: Configuration A

The resulting time delays can be observed in the following bar graph. The transmission delay of 1 Byte under normal circumstances hols a time

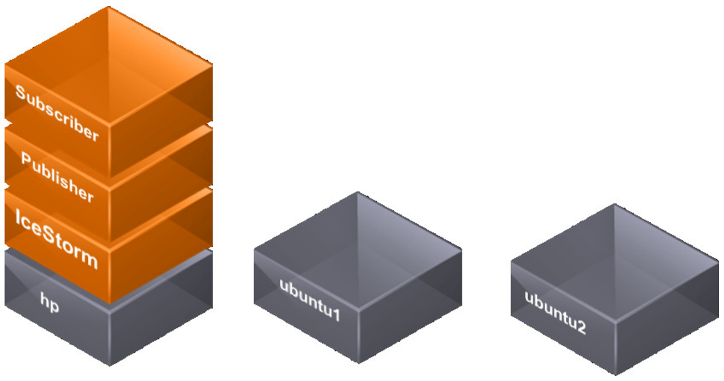


Figure 4.1: Configuration A

of 1,237 ms. For 1 Mb and 6 Mb the time rises to just below 2 ms 1,941 ms and 1947 ms respectively.

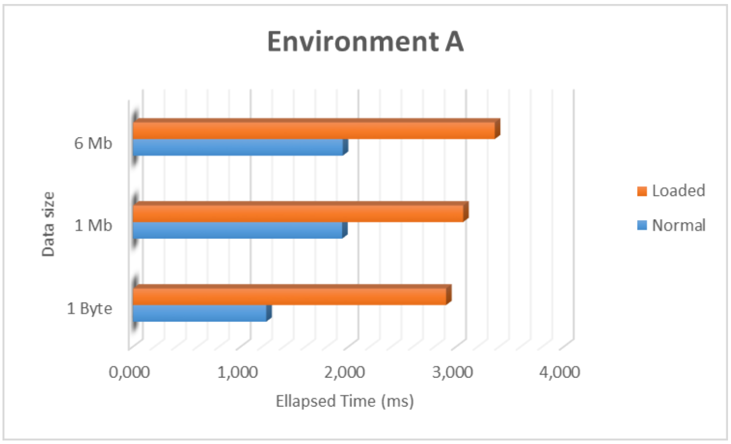


Figure 4.2: Time Results Configuration A

With an overloaded system, these times increased up to *134,95 %*. An important increase which translates into doubling the transmission delay. The delay is significant compared to the small transmission times under normal circumstances. This system would not be usable for the transmission of small sets of data.

## 4.2 Configuration B

For this configuration again only the *Computer A* is used. The only difference from *Configuration A* is the placement of the *Subscriber*. For this configuration the subscriber instance will run from a guest domain, in this case *ubuntu1*.

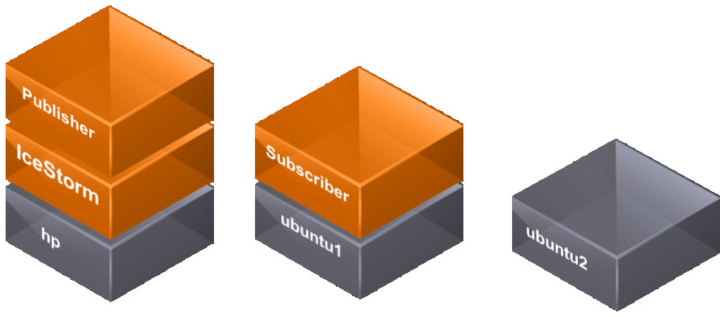


Figure 4.3: Configuration B

### 4.2.1 Results: Configuration B

Initially the prediction of this configuration is an increase in the transmission time as the *Subscriber* now doesn't reside within the same domain, but on a guest domain. Additionally as the *hp* domain will have one less duty during the stress period, it's performance should be noticeable.

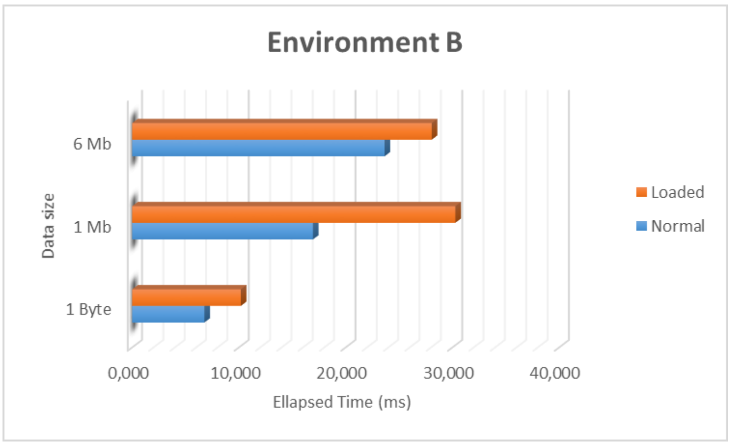


Figure 4.4: Time Results Configuration B

Indeed the transmission times in a relaxed state have increased noticeable, and also as predicted the increase in time during the stressed state has been proportionally decreased.

The fact that now the work is shared among two different domains counteracts in the increase of time during the stressed state. Only on average, the transmission was delayed by 50 %, half the time compared to *Configuration A*.

Talking about a relaxed time of 6,7 ms, 17 ms and 23,6 ms respectively for the different amount of data sent, the delay has increased on average up to 7 ms depending on how much data was sent by the *Publisher*. Again a more detailed results can be found in Annex A.1

Evaluating these results in aspects of percentile efficiency, again the higher the volume of data sent the better the performance.

## 4.3 Configuration C

This configuration is the last one to use just one workstation. In this latter configuration each instance of the application is deployed in individual domains, so the IceBox service is run from the control domain and subscriber and publisher run from the domains *ubuntu1* and *ubuntu2* respectively.

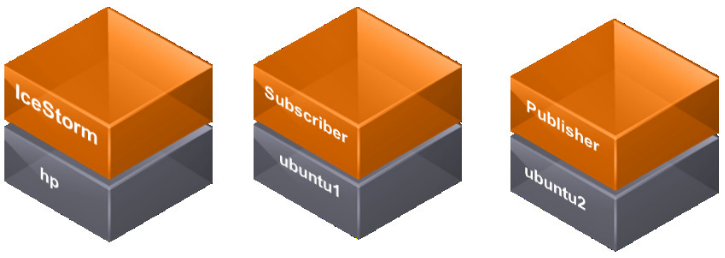


Figure 4.5: Configuration C

### 4.3.1 Results: Configuration C

What is expected is an even lower delay time while the system is stressed, as each domain only has to cope with one duty. The delay incurred due to the fact that now three domains are being used, should affect the times considerably.

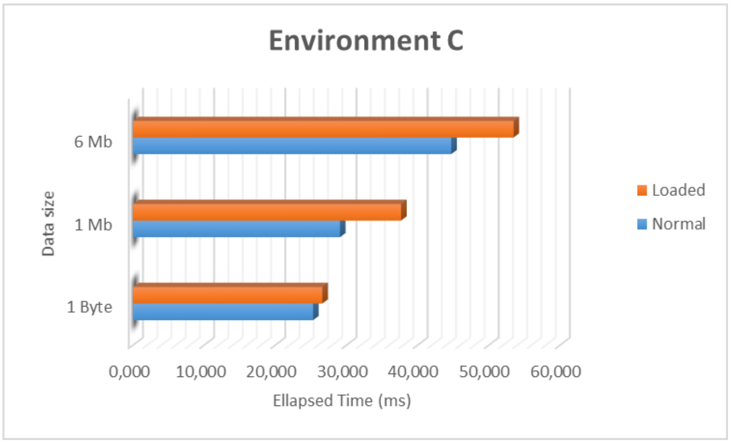


Figure 4.6: Time Results Configuration C

In fact the delay while using a stressed system varies from 5% to 29,5%, so far the best results achieved.

Dividing the duties among domains, clearly favours the transmission times during a stressed system. However the transmission times under normal CPU load are significantly increased when compared to Configuration A and B. The delay proportion may be low, but the overall delay compared to the simple configuration is remarkable high, and should be taken seriously into account.

The times raise from 1,2 ms in Configuration A up to 25,2 ms in this configuration when using a relaxed system and 1 Byte of data is sent.

## 4.4 Configuration D

This is the last configuration adopted for the performance trials. This configuration involves both computers. Again only one instance of the application is run on a single domain, as in the previous configuration. This time around the publisher will run within *Computer B* on the *vm-1* domain. On *ubuntu2* host no applicatino will run.

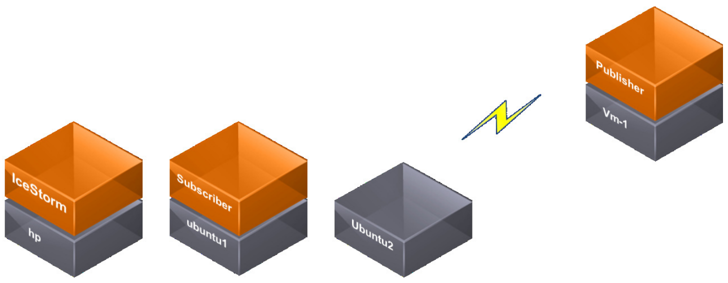


Figure 4.7: Configuration D

### 4.4.1 Results: Configuration D

Predicted during this configuration are large transmission times during a relaxed state compared to the basic configuration, but with very similar delays in a stress situation of the systems. The times measured are from the second subscriber located in the *vm-1* domain.

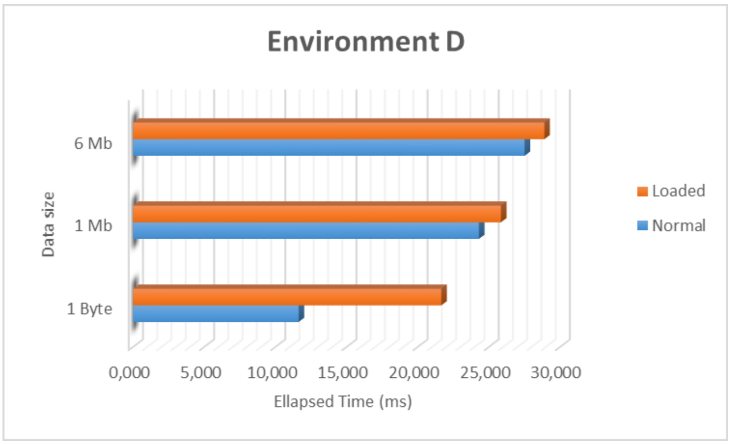


Figure 4.8: Time Results Configuration D

These results are very similar when it comes to delay time, in ms, of the loaded system, compared to the basic configuration. However, from a percentile point of view, this configuration holds the best results of the trials.

The low transmission delay achieved from *Computer A* to *Computer B* is due to the fact that Computer B is a much more powerful unit, and the processing time is reduced to a minimum leaving the actual transmission time from one end to another neglectable.

Surprisingly the resulting delays are lower than in the previous configuration. In a relaxed state 11,6 ms 24,2 ms and 27, 5 respectively. Resulting in 14 ms, 5 ms and 17 ms faster transmission times.This can only be due to the same reason mentioned a few lines above. The processing time is much better on *Computer B* and hence the times. From these results further conclusions can be drawn, and are explained in the upcoming section.

## 4.5 Conclusions

As there are no given baseline time values to compare these results with, further analysis of the resulting data is required, the times simply by themselves do not hold a decision of whether the development and the trials have been done correctly or not.

Figure 4.9 shows the transmission delay of the four different scenarios. This is the time difference calculated when the system works under a high load. The consecutive bars represent 1 Byte, 1 Mb and 6 Mb consecutively. Unfortunately no fundamental conclusion can be drawn from this results.

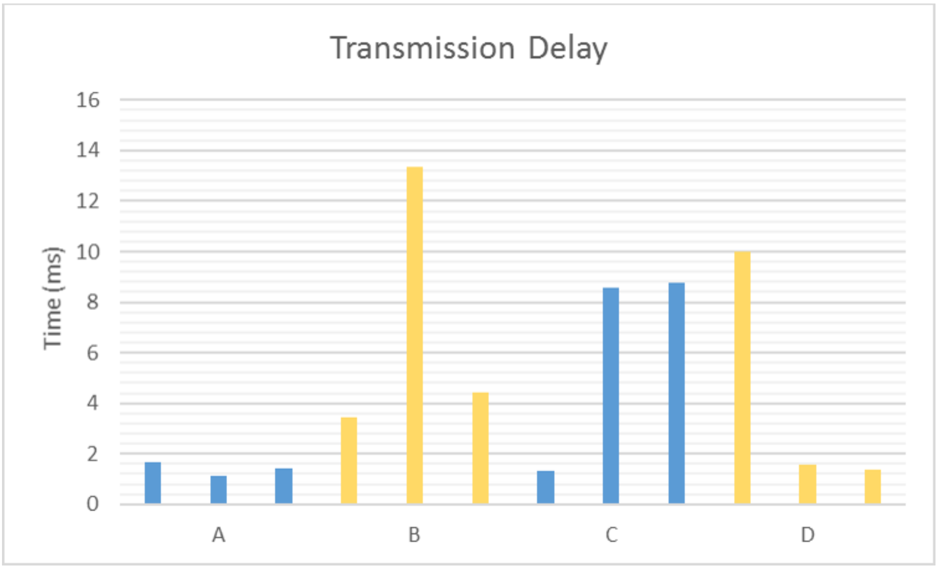


Figure 4.9: Transmission delay

However if the transmission delay times are compared to the normal transmission times in the different scenarios, conclusions can be drawn.

$$TxDelay(\%) = \frac{StressedTxTime - RelaxedTxTime}{100}$$

The relationship between the delay in time increase calculated in natural units and in percentage is important result to highlight. An extend delay in milliseconds does not necessarily imply an extreme delay in time compared to the should value of the transmission time.

This idea is reflected if the most extreme results from *Configuration A* and *Configuration D* are compared. The delay of 1.7 ms with a normal transmission time of 1.23 ms, corresponds to approximately 134% of delay in the transmission, where as, in the latter configuration 1.38 ms of delay only translates into 5%, as the normal transmission time is above 27 ms.

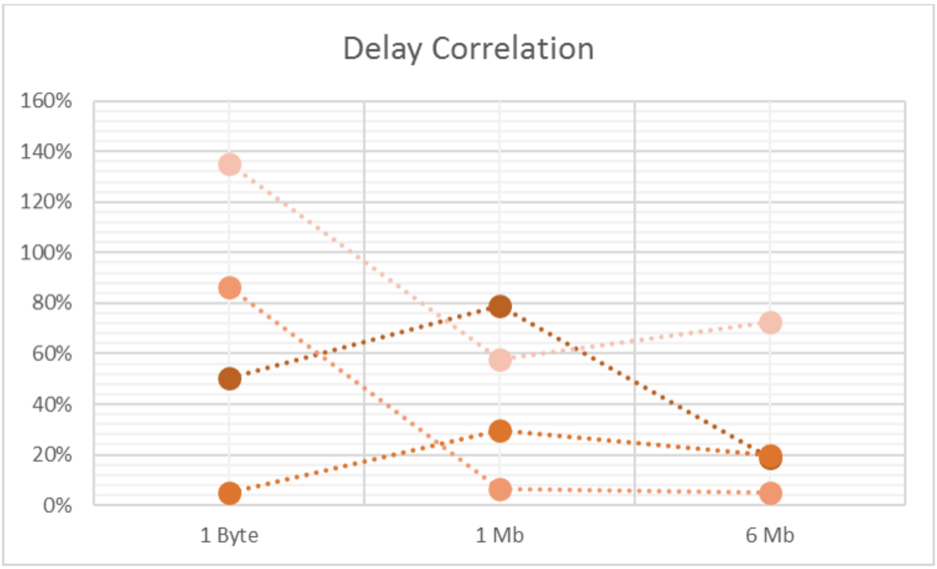


Figure 4.10: Correlation between Percentile Delay and Data Size

Moreover from the results it can also be concluded that the size of the transmitted data affects the results. Initially it could be predicted that the larger the packet sent, the larger the percentile delay would be, however that is not always the case.

Analyzing and relating the percentile delays with the amount of data sent, yields a negative linear correlation. The percentile delay in the transmission tends to be lower when the amount of data sent increases. Making this system more optimized for larger data transfers. This idea is depicted in figure 4.10

Additionally this finding reveals that the larger the size of the transmitted data, the higher the transmission time is. A straightforward conclusion that matches the theory of communications:

$$PacketTransmissionTime = \frac{PacketSize}{BitRate}$$

The above equation states that if the bit rate is set constant, the higher the packet size the higher the transmission time will be. Again, this behaviour can be observed in the collected transmission times of the trials performed.

Taking into account that the data has been extracted from over 300 records, and that both, the negative correlation and the higher transmission time for larger sized data, matches the theoretical background a correct performance can be assumed.

It is important to highlight that the results obtained, during the loaded stages, are distant to be comparable to real time values. On average the absolute delay is just below 5 ms, and although the time delay tolerance may vary from system to system, these results do not make Xen and Ice much of use for real time critical systems.

# Chapter 5

# Project Development: XtratuM

During this second part part of the project, the XtratuM hypervisor was used, to determine the efficiency and eventually the time criticality of the communication among virtual machines, in this case partitions.

As seen in the previous sections, in order to run and execute XtratuM and the partitioned system, ten complex steps must be taken, Fig. 2.8. These steps require specific requirements in order to be completed.

## 5.1 System Requirements

XtratuM is independent of Linux and self-bootable. However Linux is used as the bare OS in order to ease the development of the complete system. Several packages need to be installed in the Linux environment. The table below shows the required packages that need to be installed in the Linux system. [12]

| Pacakge | Linux Package Name | | Purpose |
|---|---|---|---|
| host gcc | gcc-4.4 | req | Build host utilities |
| make | make | req | Core |
| libncurses | libncurses5-dev | req | Configuration source code |
| binutils | binutils | req | Core |
| x86-toolchain | gcc-4.4 | req | Core |
| libxml2 | libxml2-dev | req | Configuration parser |
| qemu | qemu | req | Simulated run |

Table 5.1: Required Packages

As far as the hardware is regarded, the only restriction is made with the processor. So far from the Sparc v8 architectures, the LEON2, LEON3 and the multicore LEON4 processors are supported, in addition to generic x86 and ARM processors.

## 5.2 Architecture & Design

For the development of the environment XtratuM *xmvm-x86-2.4* was used. This is the only available version to download from the official website. As suggested by the official guide, the bottom OS layer is a Debian Linux distribution, to be precise version 3.2. The reason for using such old version is simply by suggestion of the official XtratuM documentation.

The actual hardware was fully virtualized using VMware, for the purpose of keeping it as clean and simple as possible. The following architecture was design. Note that once the XtratuM system was deployed it detects *2397.982 MHz* processor speed.

| | |
|---|---|
| CPU Architecture | i686 |
| CPU mode | 32-bit & 64-bit |
| Clock Speed | 2400.001 MHz |
| Number of CPUs | 1 |
| Core/socket | 1 |
| RAM | 1 GB |

Table 5.2: Hardware Configuration VMware

On top of the hypervisor layer two different partitions will be deployed as seen on the following figure.

## 5.3 Building XtratuM

With the configured hardware and all the previous requirements met, the next step ought to be building XtratuM. The first step is to compile the hypervisor, but just before that a deep clean may be performed in order to remove any configurations that may interfere with the eventual system.

As described in the theory section, section 2.4.2, the XtratuM sources have to be compiled and executed. In order to accomplish a correct output

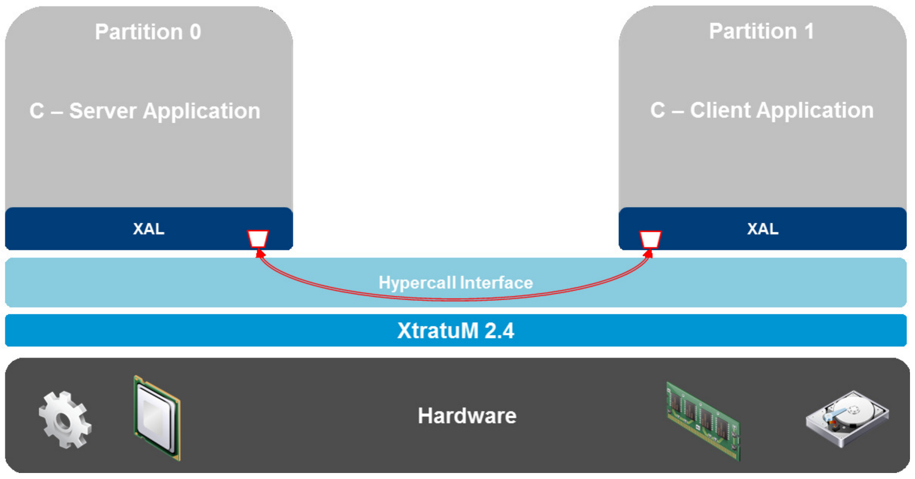


Figure 5.1: Architecture for XtratuM

the *PATH* variables of the source file *xmconfig* must be filled in accordingly to with the path of the required locations, as for example the root folder of the XtratuM files.

During this configuration a configuration wizard will pop up, and assist the user during the configuration. Most importantly is the selection of the correct processor from the *menuconfig*.

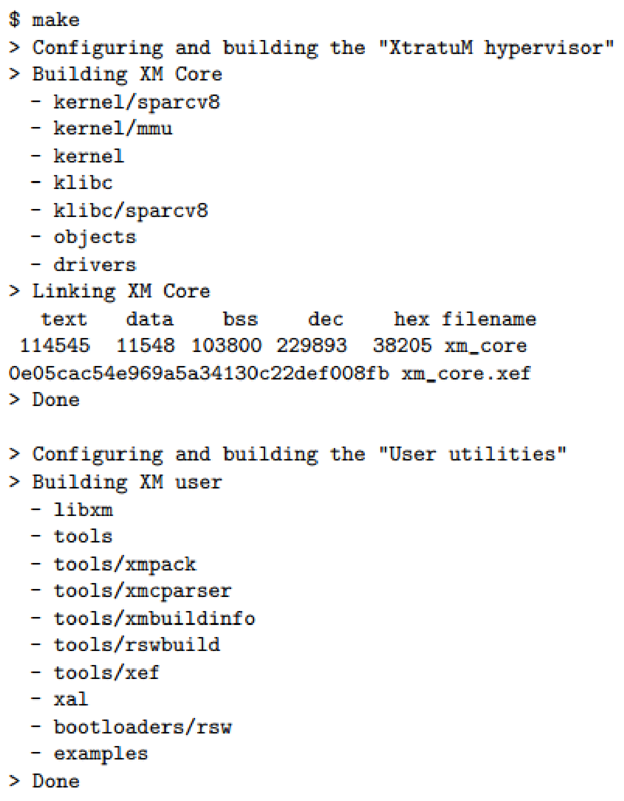
Once the configuration is finished, the next step is to compile the sources combined with the created configuration files. Using the simple command *$ make* will automatically compile and build all necessary files. The output of this step can be observed in fig. 5.2

When done, binaries for distributing among partitions are required. This can be simply achieved with a self-extracting installer invoking *$ make distro-run*. This will generate *xtratum-2.4.run* file.

This file is then simply to be executed in order to install and deploy the final system. If all steps are done correctly a "*Installation completed.*" should print on the terminal.

## 5.4 Building Partitions

As mentioned, a partition is an execution environment. It can be a bare-application, a real-time OS or a general purpose OS with its applications.



```
$ make
> Configuring and building the "XtratuM hypervisor"
> Building XM Core
  - kernel/sparcv8
  - kernel/mmu
  - kernel
  - klibc
  - klibc/sparcv8
  - objects
  - drivers
> Linking XM Core
   text    data     bss     dec     hex filename
 114545   11548  103800  229893   38205 xm_core
0e05cac54e969a5a34130c22def008fb xm_core.xef
> Done

> Configuring and building the "User utilities"
> Building XM user
  - libxm
  - tools
  - tools/xmpack
  - tools/xmcparser
  - tools/xmbuildinfo
  - tools/rswbuild
  - tools/xef
  - xal
  - bootloaders/rsw
  - examples
> Done
```

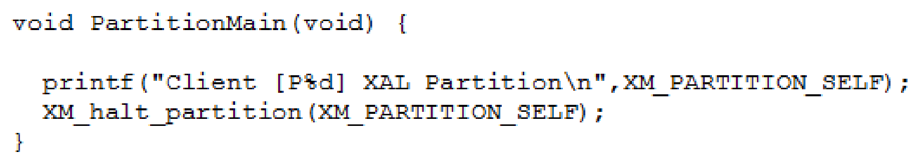
Figure 5.2: XtratuM Compilation

For the purpose of this project, two different bare-application partitions will be created. This partitions are known as XAL, are minimal developing bare C applications.

As the performance of the communication between partitions is to be analyzed, one client and one server have been programmed into the two different XAL applications, with the purpose of establishing a connection between them and send data from the server to the client.

This bare C applications or XAL partitions have to have a special structure. The *xm* header for hypercalls, and the standard *stdio* header may be used for normal C functions. Furthermore there must be a *void PartitionMain(void)* function where the executing code can be found.

The above partition will simply print out to the terminal the status and then the partition will be halted.



```
void PartitionMain(void) {

  printf("Client [P%d] XAL Partition\n",XM_PARTITION_SELF);
  XM_halt_partition(XM_PARTITION_SELF);
}
```

Figure 5.3: XAL Example File

Additionally to the actual XAL code which is in charge of running the desired functions, the configuration file must be tailored to meet the required criteria. This file called *xm cf.ia32.xml* is the core for a satisfactory configuration of the system.

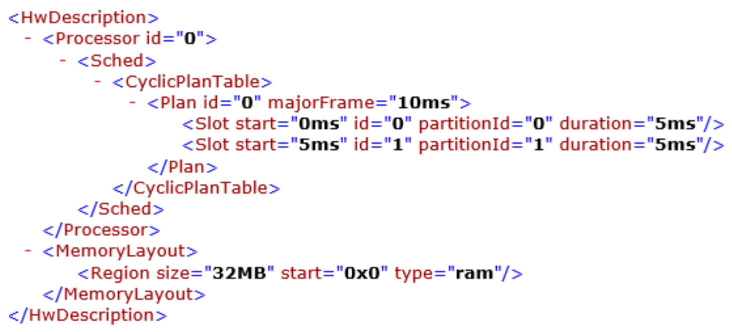
The file may contain various amount of information and properties of the desired partitioning configuration. The most relevant configuration properties that are relevant to the project and that can be found in the file are:

- Physical Memory Area
- Hardware Description
  - Processor Selection
  - Cyclic Plan Table
  - Memory Layout
- Partition Table, i.e. properties of the two partitions
- Channel information

Taking a look at the actual xml file, we can observe the Hardware Description section. The complete configuration file can be found in Annex B.

### 5.4.1 Application: Partition 0

Partition with Id. **0** contains a simple XAL application that behaves like a casting server. It will use the inter-partition communication techniques to



```
<HwDescription>
 - <Processor id="0">
   - <Sched>
     - <CyclicPlanTable>
       - <Plan id="0" majorFrame="10ms">
           <Slot start="0ms" id="0" partitionId="0" duration="5ms"/>
           <Slot start="5ms" id="1" partitionId="1" duration="5ms"/>
         </Plan>
       </CyclicPlanTable>
     </Sched>
   </Processor>
 - <MemoryLayout>
     <Region size="32MB" start="0x0" type="ram"/>
   </MemoryLayout>
</HwDescription>
```

Figure 5.4: Hardware Description XM_CF

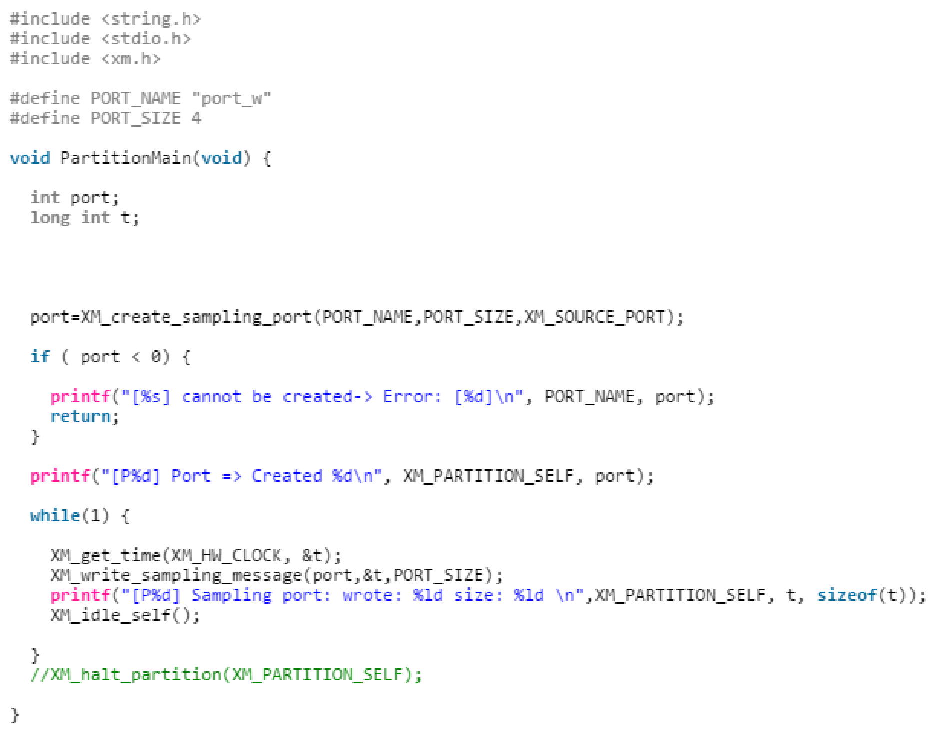
send data to the other partition, the client. In this case *sampling* ports will be used.

The correct configuration is reflected in the *XM_CF*. With this, the port can be created inside the XAL application and the desired data can be written into it. Both these operations can be accomplished with *XM_create_sampling_port(char *portName,xm_u32_t maxMsgSize, xm_u32_t direction)* and *XM_write_sampling_message(xm_s32_t portDesc, void *msg-Ptr, xm_u32_t size)* respectively. The appropriate parameters are chosen to achieve the correct implementation.

In order to analyze the performance, the transmitted data will be a *long* variable, which contains the value of the time elapsed since the last hardware reset. This can be obtained using the *XM_get_time(xm_u32_t clockId, *xmTime_t time)*. As clockId the *XM_HW_CLOCK* will be used.

The partition will only write one data sample to the port in each execution period. The reason for this is to avoid data loss and assure the correct transmission of the data. The different time configurations of the partition will be explained in more detail in the Trial section. As for the physical memory are, this partition will have 1 MB available to use.

As it is a bare C application no much resources will be needed, and the allocated memory should suffice.



```
#include <string.h>
#include <stdio.h>
#include <xm.h>

#define PORT_NAME "port_w"
#define PORT_SIZE 4

void PartitionMain(void) {

  int port;
  long int t;



  port=XM_create_sampling_port(PORT_NAME,PORT_SIZE,XM_SOURCE_PORT);

  if ( port < 0) {

    printf("[%s] cannot be created-> Error: [%d]\n", PORT_NAME, port);
    return;
  }

  printf("[P%d] Port => Created %d\n", XM_PARTITION_SELF, port);

  while(1) {

    XM_get_time(XM_HW_CLOCK, &t);
    XM_write_sampling_message(port,&t,PORT_SIZE);
    printf("[P%d] Sampling port: wrote: %ld size: %ld \n",XM_PARTITION_SELF, t, sizeof(t));
    XM_idle_self();

  }
  //XM_halt_partition(XM_PARTITION_SELF);

}
```

Figure 5.5: XAL: Partition 0

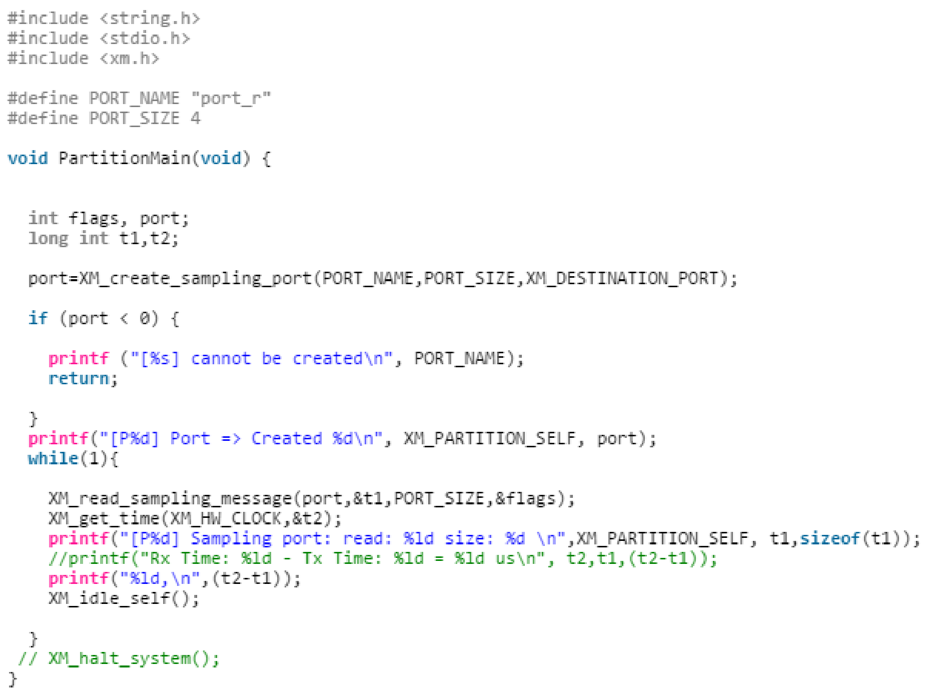
### 5.4.2 Application: Partition 1

On the other side, the partition with Id. **1** will resembles a simple straightforward client. Again based on a XAL application, this partition will create another sampling port, in order to read the data.

After the port for reading is created, the application will read the data inside that port. To do so the *XM_read_sampling_message(xm_s32_t portDesc, void *msgPtr, xm_u32_t size, xm_u32_t *flags)*. This function will write to the pointed variable the contents of the port. In the case of this application it will read the elapsed time, at the point in time just before the data was sent from the server partition.

Yet again this client calculates the elapsed time on this partition since the last hardware reset, right after the data was received correctly. Naming this last obtained elapsed time *t_client* and the received elapsed time *t_server*, simple arithmetic will allow to calculate the delay time since, the data was written in the sampling port, until it was read again from the sampling port.

$$TransmissionTime = t\_client - t\_server$$



```
#include <string.h>
#include <stdio.h>
#include <xm.h>

#define PORT_NAME "port_r"
#define PORT_SIZE 4

void PartitionMain(void) {


  int flags, port;
  long int t1,t2;

  port=XM_create_sampling_port(PORT_NAME,PORT_SIZE,XM_DESTINATION_PORT);

  if (port < 0) {

    printf ("[%s] cannot be created\n", PORT_NAME);
    return;

  }
  printf("[P%d] Port => Created %d\n", XM_PARTITION_SELF, port);
  while(1){

    XM_read_sampling_message(port,&t1,PORT_SIZE,&flags);
    XM_get_time(XM_HW_CLOCK,&t2);
    printf("[P%d] Sampling port: read: %ld size: %d \n",XM_PARTITION_SELF, t1,sizeof(t1));
    //printf("Rx Time: %ld - Tx Time: %ld = %ld us\n", t2,t1,(t2-t1));
    printf("%ld,\n",(t2-t1));
    XM_idle_self();

  }
 // XM_halt_system();
}
```

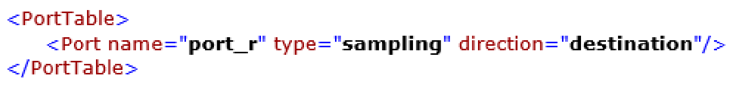
Figure 5.6: XAL: Partition 1

## 5.5 Networking

The correct configuration of the network is essential for the project. This communication between the different partitions is port based, and fully virtualized.

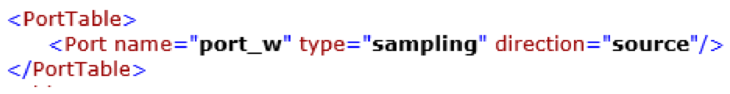
In order for these ports to be utilized by the XAL applications, they have to appear on the configuration file according with the requirements. For this project, two different ports are going to be utilized. One for writing and another one for reading.

The actual configuration information of the port, name, type and destination, can be found within the actual *Partition* section.



```xml
<PortTable>
    <Port name="port_r" type="sampling" direction="destination"/>
</PortTable>
```

Figure 5.7: Port Configuration



```xml
<PortTable>
    <Port name="port_w" type="sampling" direction="source"/>
</PortTable>
```

Figure 5.8: Port Configuration

Additionally the virtual channel that will interconnect the ports with each other has to be captured in the configuration file. There will only be a single channel that connects the writing port with the reading port.

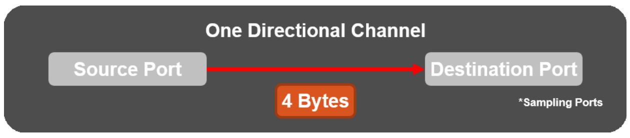


Figure 5.9: Port Configuration

With the correct configuration special hypercalls inside the XAL application can be used in order to create, the configured ports, send and receive the desired data.

# Chapter 6

# Trials: XtratuM

During this chapter various results obtained from performing different tests will be carefully analyzed. The first three sections have been completed with a 5:1 time ratio between the allocated time for the server and for the client. The last section, D, will have a ratio of 1:1. During all performance tests, the same amount of data has been transferred, 4 Bytes, the changing variable will be the scheduling plans.

Around 20-30 samples of the delayed times have been recorded. With these samples, the average and standard deviation were calculated. This is used to generate 20.000 statistical samples following a normal distribution. With this large amount of data a histogram has been created and hence the Gauss bell diagram represent the probability of having a certain delay time. *By delay the total transmission time is understood.* This is because if the system is considered real time, any elapsed time can be considered as an actual delay.

It is worth to note that even tough a theoretical wide range time values are represented in the diagrams, the practically obtained correspondent maximum transmission times and minimum transmission times are to be seen in the correspondent tables. No practical results were obtained outside the *max-min* boundary.

## 6.1 Scheduling Plan A

Given the following scheduling table the system was configured. In this case, and for the rest of the upcoming scenarios, the Major Time Frame is a complete cycle of both partitions, this way the system is optimized, no waiting times occur and no inter-exchange in the order of the partitions will occur. The server partition will always run first and the client partition will run second.

| MF: 600 ms | Start | Duration |
|---|---|---|
| Server | 0 ms | 500 ms |
| Client | 500 ms | 100 ms |

Table 6.1: Scheduling Plan A

The calculated average resulted to be $499532\mu s$. This result compared to the Server's schedule times, coincides almost to the $\mu s$ with the duration of such. From this comparison, it can be stated that the transmission delay is simply neglectable. The time taken is just the duration of the partition.

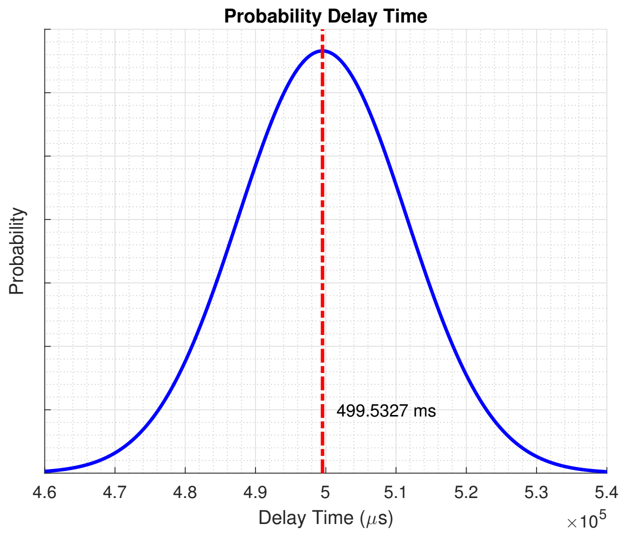


Figure 6.1: Results 500 ms

| Max Time | Min Time |
|---|---|
| $500200\mu s$ | $493279\mu s$ |

Table 6.2: Practical Range of Results A

## 6.2 Scheduling Plan B

In this case the calculated average resulted to be just some $\mu s$ above the duration of the Server's duration, $50030\mu s$. Again the average equals the duration of the Server's partition. However $30\mu s$ of overtime can be observed. This implies only a 0.06% of increase in the time. This overtime can be considered as neglectable, as the appreciation of $30\mu s$ is close to zero time.

| MF: 60 ms | Start | Duration |
|---|---|---|
| Server | 0 ms | 50 ms |
| Client | 50 ms | 10 ms |

Table 6.3: Scheduling Plan B

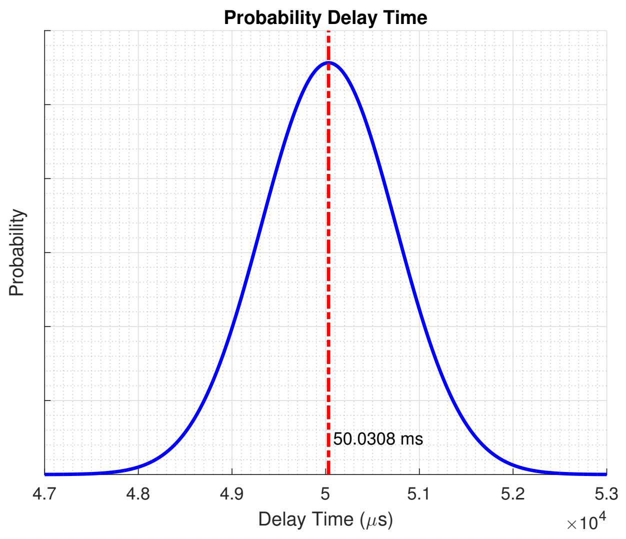


Figure 6.2: Results 50 ms

| Max Time | Min Time |
|---|---|
| $50945\mu s$ | $43642\mu s$ |

Table 6.4: Practical Range of Results B

## 6.3 Scheduling Plan C

This is the last scenario where the ratio 5:1 is used. Again it can be observed that the average transmission time is very similar to the duration of the Server's partition. In this case the average holds $5085\mu s$.
Again in this scenario the average exceeds the duration of the Server's partition. In this scenario $85\mu s$ which correspond to 1.7%.

| MF: 6 ms | Start | Duration |
|---|---|---|
| Server | 0 ms | 5 ms |
| Client | 5 ms | 1 ms |

Table 6.5: Scheduling Plan C

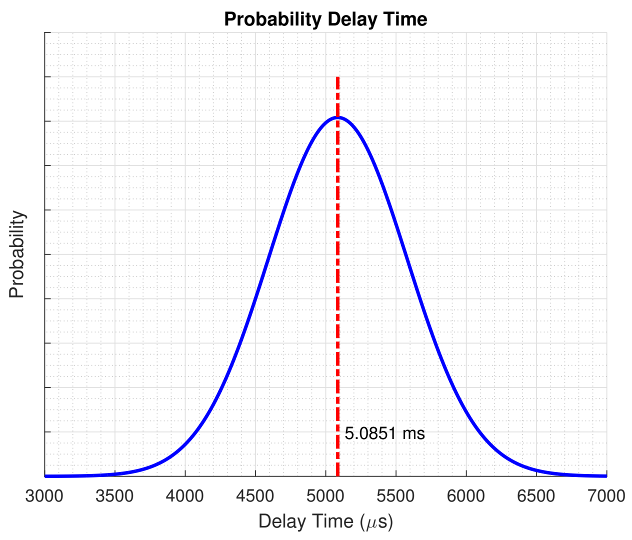


Figure 6.3: Results 5 ms

| Max Time | Min Time |
|---|---|
| $50945\mu s$ | $43642\mu s$ |

Table 6.6: Practical Range of Results C

During the trials with this scheduling plan, there occurred some anomalies. Values doubling the average time measurement as for example $11060\mu s$

were found. An explanation to this is presented in the conclusion section.

## 6.4 Scheduling Plan D

On this latter scenario, the scheduling plan has been modified so that it holds a 1:1 ratio regarding the partition's durations. The average now is $4894\mu s$. It can be observed that this time the average lies below the duration of the server's partition duration. The average lies 2.12 % below the time duration.

This resembles a significant improvement in performance if compared to the previous scenario.

| MF: 600 ms | Start | Duration |
|---|---|---|
| Server | 0 ms | 5 ms |
| Client | 5 ms | 5 ms |

Table 6.7: Scheduling Plan D

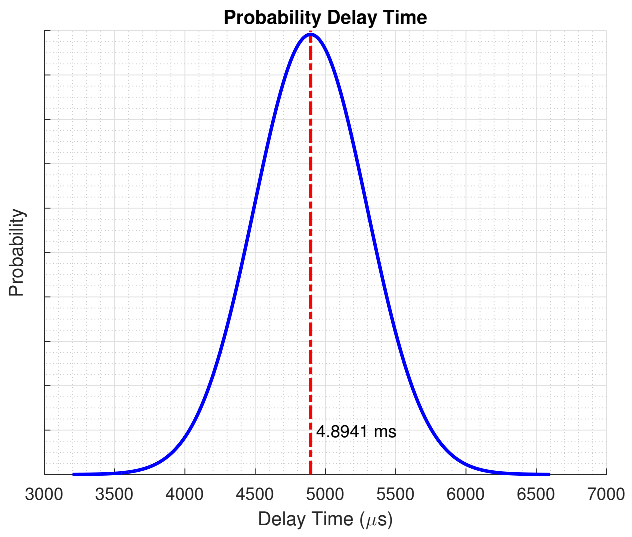


Figure 6.4: Results 5 ms (1:1 Ratio)

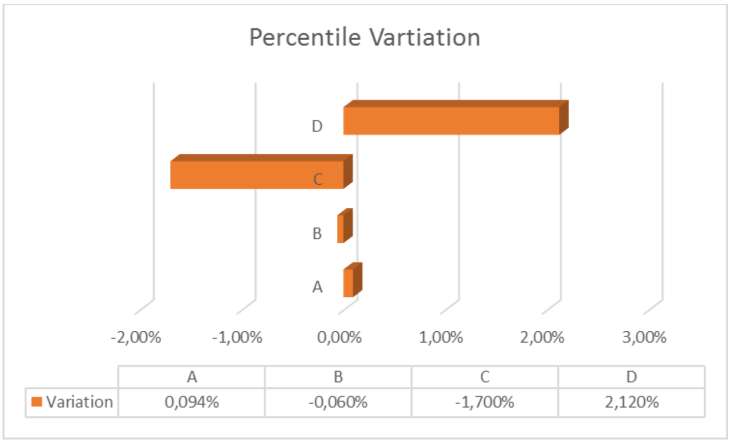
| Max Time | Min Time |
|---|---|
| $50940\mu s$ | $3839\mu s$ |

Table 6.8: Practical Range of Results D

## 6.5 Conclusion

Analyzing the obtained results the conclusion that can be made is clear. The relationship between the Server's partition and the time taken to deliver the data does not oscillate more than 2.1% with respect to that time. A time variation of just a few $\mu s$ does not necessary mean a failure in the application or in the system.

Hence the system does actually communicate without any noticeable delay.



|  | A | B | C | D |
|---|---|---|---|---|
| 🟧 Variation | 0,094% | -0,060% | -1,700% | 2,120% |

Figure 6.5: Percentile Variation XtratuM

The actual transmission time of the data in addition to the execution time that the application needs for creating the ports and writing or reading from them will from now onwards described as the neto time.

The measured times during the trials do not correspond entirely to just the neto time. Additionally to the neto time, a partition's content switch overhead time must be taken into consideration.

As pictured in the figure 6.6a XtratuM saves and loads the partition's context, here the optimum operation is pictured.

Since the optimum case can not be guaranteed all the time, XtratuM will try to adjust as much as possible the beginning of the transition. This can be observed in 6.6b.

In the case where the partition execution time reaches beyond the planned time slot, it may be the case if a hypercall is executed just at the end of the time slot, the context switch will delay the beginning of the next partition. This can be seen in 6.6c.

The delay added to the neto time results in the final measured time. This explains why there are transmission times that may take longer than the partition duration.

This case clearly occurred in scenario C, as anomalous results were obtained. Here the context switch was offsetting the starting times with every cycle.

As no time margins were pre-configured, the slot times of the partitions left for execution of the applications suffered a cyclic reduction. Therefore every *n* cycles, the client partition had been reduced up to an extend where it could not run the application satisfactory and had to wait for the next cycle to properly read the data. Note that that unread data sample is lost and will never reach the destination.

This latter delay, correspondent to the context switch, can be removed if the worst case execution time of the context switch is introduced as margin between the partitions. Represented in 6.6d. Nevertheless deeper research into how that time is calculated is required.

Concluding, regardless of the fact that the measured times take into account the execution time of creating/writing/reading the ports and the eventual context switching times, these transmission times are simply not exceeded by a factor for them not to be considered real time communication.

The data will just take as long to reach the destination, as the duration of the sending partition is. Hence the main and one factor that will affect this transmission time is simply the duration of the transmitting partition.

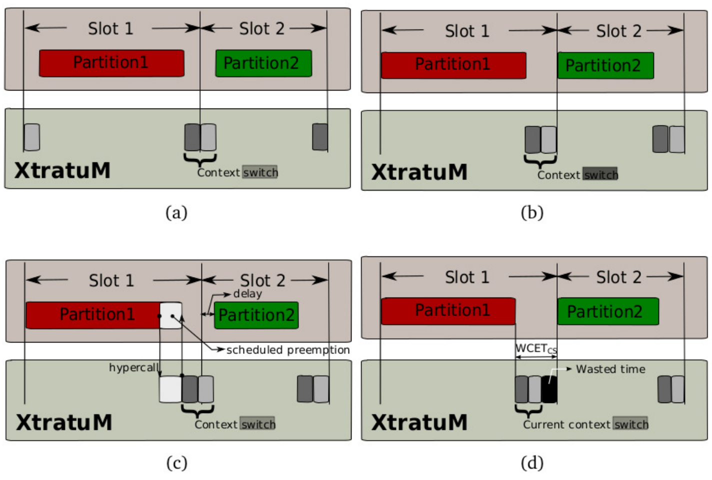


Figure 6.6: XtratuM Context Switch [12]

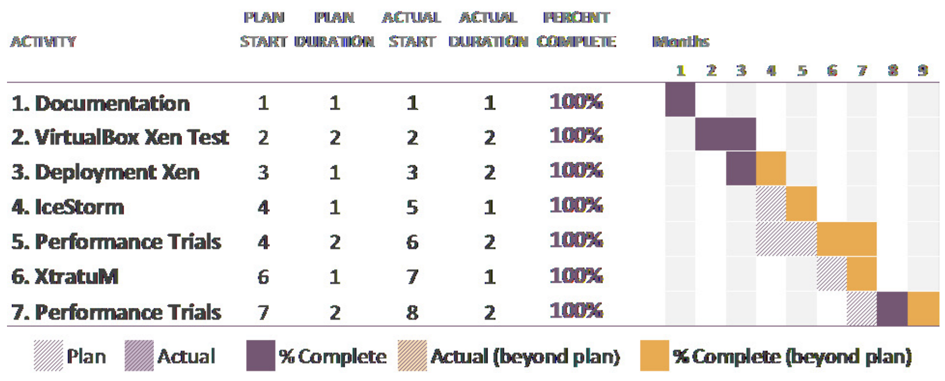
The transmitting partition's duration must be long enough for the it to create and read the port. Otherwise no transmission will occur as the idle state will be reached before achieving the successful transmission. Analogously, the receiving partition must have enough time to create and read the port.

# Chapter 7

# Project's General Information

## 7.1 Project's History

This project started back in October 2015 and took place until June 2016. During those nine months diverse targets were set and multiple stages created with the aim of achieving a structured and time efficient work.



| ACTIVITY | PLAN START | PLAN DURATION | ACTUAL START | ACTUAL DURATION | PERCENT COMPLETE |
|---|---|---|---|---|---|
| 1. Documentation | 1 | 1 | 1 | 1 | 100% |
| 2. VirtualBox Xen Test | 2 | 2 | 2 | 2 | 100% |
| 3. Deployment Xen | 3 | 1 | 3 | 2 | 100% |
| 4. IceStorm | 4 | 1 | 5 | 1 | 100% |
| 5. Performance Trials | 4 | 2 | 6 | 2 | 100% |
| 6. XtratuM | 6 | 1 | 7 | 1 | 100% |
| 7. Performance Trials | 7 | 2 | 8 | 2 | 100% |

Figure 7.1: Project Development Gantt Chart

The first stage, taking place in October, consisted mainly in collecting information and documentation about the virtual environments and real time based systems, with high detail to the Project Xen, XtratuM and both the Ice and DDS middleware. A deep analysis and insight was gained during this period of time.

With all the information it had to be determined what type of environments would be implemented. Regarding the middleware the decision was taken to use Ice. Simply the available resources, examples and guides over matched those of DDS. Another advantage of this choice was the possibility of developing this application in any of the programming languages, Java, C++, C#, php, Ruby etc. However C++ SDK was finally chosen.

As the actual real time of the system was going to be evaluated and put under performance tests, both Xen and XtratuM hypervisors were going to be installed in order to have notice the difference between what is supposed to work in real time, XtratuM, and a normal type 1, bare bone, hypervisor Xen.

During the second stage the installation and system test of Xen, Xtratum and the different Ice Services was put under test in the Oracle VM VirtualBox environment. This way, it is very easy to install and test multiple configurations of host operating systems and network configurations, with the possibility to backup working states of the machine, and restoring them in the case of future failure.

After taking into account the user experience and stability of the diverse configurations the decision of using Ubuntu 14.04 LTS was taken. Later this OS would be upgraded to the Xenial Xeros version.

The third stage, the configuration was intended to be implemented in a Lenovo E460 Laptop, but unfortunately this laptop was an out of the box developers edition yet, that did not have vitalization capabilities enable through the BIOS. An update that was eventually to come had to be installed. Two weeks of intensive research to find a workaround solution were spent, but without success. Finally an old laptop had to be taken in order to start of with the development of the system. At this stage the project got behind schedule.

Finally the Xen hypervisor was installed, and afterwards the correspondent guest domains. Now the system was ready to start off with the developing of the Ice service.

During the fourth stage the *Subscriber* and *Publisher* were programmed to performed the desired task of being able to measure the time taken since the publisher publishes into a topic until the subscriber receives from that topic. Additionally a performance test protocol was established for the future testing.

After failing to deploy the IceStorm service, it came obvious that an extra Ice service was necessary, IceBox. A few days of research were necessary as it was completely new to the project. However no further problems were encountered and no delay was caused in the project's schedule. The operational and user tests turned out successfully.

Stage number five was the most time consuming stage from all the previous stages. The performance trials were performed during this period of time. Repetitive tests, saving logs, and ordering the collecting data. As mentioned over 300 records of date were stored for further analysis.

Once the trials had been performed, and made sure that the results were coherent, consistent and accurate, XtratuM was installed on the machine.

Lastly the same trials as in stage 5 were performed, once again very repetitive and time consuming stage of the project.

## 7.2 Encountered Problems and Mended Errors

During the eight-nine months of the project no big incidentals occurred. Perhaps the most significant one right at the beginning, when the BIOS of the new laptop did not accept the installation of Xen, was one of the major delays of the project.

Moreover, as no wireless connections could be virtualized the use of a wireless access point to convert the connection to a physical cable one was necessary. Just a couple of days until the delivery of the AP were necessary, from then onwards the AP did no interfere at all in any way with the execution of the project.

During the XtratuM part of the project, a big amount of time was invested searching for information. The official company, a spin off of the polytechnic school of Valencia, offers an academic version. However the version provided is out-dated and various examples do not compile. This makes the procedure more complex.

Additionally, no previous analysis like this has been done previously and a lot of self try and error had to be invested to reach the final application environment.

Summing up, one hardware issue and one software issue tried to set back the project, but in the end the project was finished successfully within the planned time, and fully in line with the milestones expected.

## 7.3 Legal Framework

During the development of the project numerous pieces of software were used. No legal restrictions, further than complying with the license agreements, have to be taken into account if these systems are to be used.

In the following paragraphs the different licence agreement of the software used is described in detail.

Ubuntu is a collection of thousands of different programmes and documents created by numerous individual people companies and teams. These may come under different licenses depending on the authors wish. However all the different programs used during this project inside Ubuntu, *stress*, and the *g++ compiler* are both free software licensed under the **GPLv2**.

GPLv2 can be summarized as allowing copying and distribution of the verbatim copies, but the modification of them is not permitted.

Moving on to the commercial software used for this project. Surprisingly it was the case to be that all the additional software used during the project had open-source license for developing and research purposes.

Project Xen was released under the terms of the GPLv2 and hence no further End User License is required. The only restriction that comes while using Xen is the trademark policy to ensure that all the different branded hypervisors are actually fully compatible and will run the available virtual machines. Note that Project Xen is owned by Citrix, and so is its trademark.

As for the second hypervisor used, XtratuM, the academic version under the GPL was used, there exist however a professional version which contact with the company has to be made in order to get a pricing and licensing information.

With regard to the middleware, Ice, belonging to the Zeroc company, puts their software under the two different open-source license: GPLv2 and BSD 3-Clause. This latter license offers the possibility to modify the binaries and source code files, but the enclosure of the copyright notice is mandatory, and cannot be used to promote a final product without specific authorization of the owner.

However for commercial purposes of the Ice application an economic agreement with the owner company must be met prior to the deployment of such.

## 7.4 Environmental-Economic Sphere

Currently the world is undergoing a global warming phase that has been caused by the excess of mistreat of the earth's environment. High emissions of pollutants are contributing to the destruction of the planet. And even though the society is aware of this reality, few people take counter-measures to reduce the personal impact on the environment.

A simple electronic device unfortunately has a negative impact on the environment throughout its life-cycle.

Starting off with the manufacturing of such devices, composed mainly by semiconductors and plastic, later on while the device is used, the energy upkeep, generates a pollution footprint. And finally when the device is not usable anymore, the dumping of such device again generates a negative impact on the environment.

The objective of this project was to asses virtual systems to determine the degree of usability of these virtual environments studying the possibility of substituting multiple traditional systems into a single virtual systems.

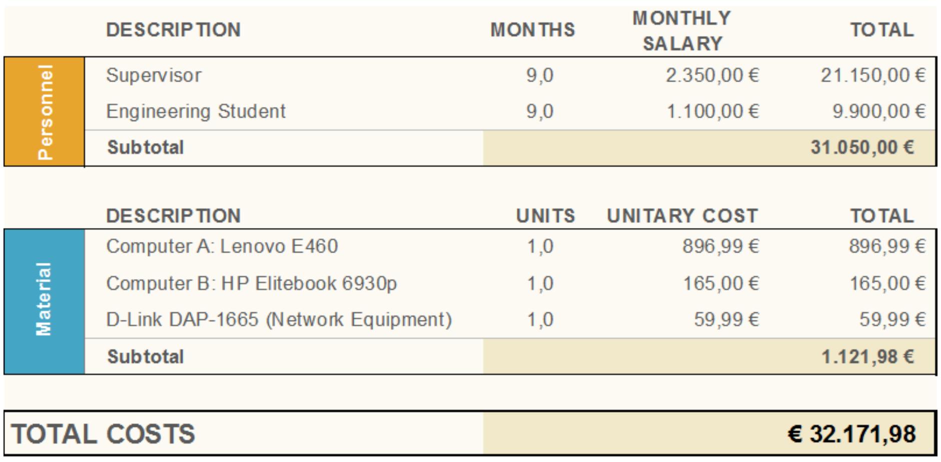
Hence if the virtualization is imposed and virtual systems are deployed a positive impact to the environment will be made, as the amount of devices manufactured is reduced.

Taking a first hand experience, at the University Carlos III of Madrid, the computer labs contain two different CPU towers with all components. One runs a Linux distribution and the other one is based on Windows. With a single tower holding both virtual environments half of the CPU towers could be spared.

The reduction of the manufacturing does not just mean a positive impact on the environment, but also on the economy. To compare the price difference of a RAM component. An 8 GB HyperX Fury RAM module has a price on the market of 49, 37¬ where the same model but 2x4 GB modules cost 76, 68¬ around 55 % more than the one module. This cost reduction for the different companies, that make use of this technology, could be translated into a higher profit, better investment or a lower price on for the consumer.

Summarizing, virtual systems have a positive impact on the environment, as they reduces the amount of electronic devices. This as well will mean a cost reduction for the owner of the systems. The large amount of scalability that a virtual environment offers with minimum effort is to be taken into account.

## 7.5 Budget



| | DESCRIPTION | MONTHS | MONTHLY SALARY | TOTAL |
|---|---|---|---|---|
| Personnel | Supervisor | 9,0 | 2.350,00 € | 21.150,00 € |
| | Engineering Student | 9,0 | 1.100,00 € | 9.900,00 € |
| | **Subtotal** | | | **31.050,00 €** |

| | DESCRIPTION | UNITS | UNITARY COST | TOTAL |
|---|---|---|---|---|
| Material | Computer A: Lenovo E460 | 1,0 | 896,99 € | 896,99 € |
| | Computer B: HP Elitebook 6930p | 1,0 | 165,00 € | 165,00 € |
| | D-Link DAP-1665 (Network Equipment) | 1,0 | 59,99 € | 59,99 € |
| | **Subtotal** | | | **1.121,98 €** |

| | |
|---|---|
| TOTAL COSTS | **€ 32.171,98** |

Figure 7.2: Project Budget

As seen in the figure the total budget of the project was **32.171,98 EUR**.

# Chapter 8

# Conclusions and Future Works

## 8.1 Conclusions of the Project

The conclusions of the two parts are highlighted in more detail in the correspondent conclusion sections. However an assessment of the overall general objective of the project follows.

The objective, was to analyze the responsiveness of the transmissions while using the virtual environments, to asses the possibility of virtualization.

From the results in the first part, the conclusion is clear. There exists a noticeable proportional delay when transmitting under a stress situation. Overall the measured delay are factors of milliseconds, and hence this virtual distributed environment can not be considered for hard real time systems.

Additionally the lower the amount of data sent the higher the proportional delay will be, another drawback encountered during this first part.

It would depend on what the time specifications for the individual systems actually are, but if up to 15 ms delay are regarded as acceptable, theses complex virtual system are absolutely safe for reproduction, as no data was lost during any of the transmissions

Hence the implementation of a virtual environment can be recommended for soft real time systems that transmit megabytes of data.

Moving on to the second part, which focuses on bare-bone virtualization. The conclusion that is taken from analyzing these results is that the transmission time matches the partition duration under normal operation. XtratuM handles the systems with extreme accuracy, and the real time property is respected.

However there exists a limitation when working with XratuM and it is the complex deployment of the system. Previous calculations must be taken into account in order to avoid switch context delays. If a correct configuration and deployment of the system is made, the transmission occurs with no delay, else the switch overhead will cause to one of the partitions to not operate every *n* cycles. In this project the loss of a transmitted data "packet" was the result of this effect.

XtratuM should be used for systems where just bytes of data are sent. Best practice in embedded systems, such as controlling units.

Overall analyzing the results and applying these in a more general context, the virtual environments should be used to replace multiple separated units. This represents no legal restrictions more than complying with the license agreements. Furthermore it has a positive impact on earth's environment and may contribute to saving substantial amounts of money.

With this project it is proven that XtratuM behaves indeed in a responsive manner, with almost neglectable time delay, but with the possibility of encountering anomalies under one circumstance. On the other hand Xen was fully reliable during the trials and the only limitation of this hypervisor is the considerable delay, to bear in mind, while experience a high system load.

## 8.2 Future Works

During this project only a small portion of the virtualization world has been analyzed. The decision was made to use two different hypervisors. However the market holds many more commercial hypervisors, such as Red Hat Enterprise Virtualization, or VmWare vSphere, that could be analyzed as well.

XtratuM offers so many different functions that makes it possible for it to be tailored to meet the specific individual needs. More insight into this exquisite hypervisor could be done to find the perfect optimum configuration.

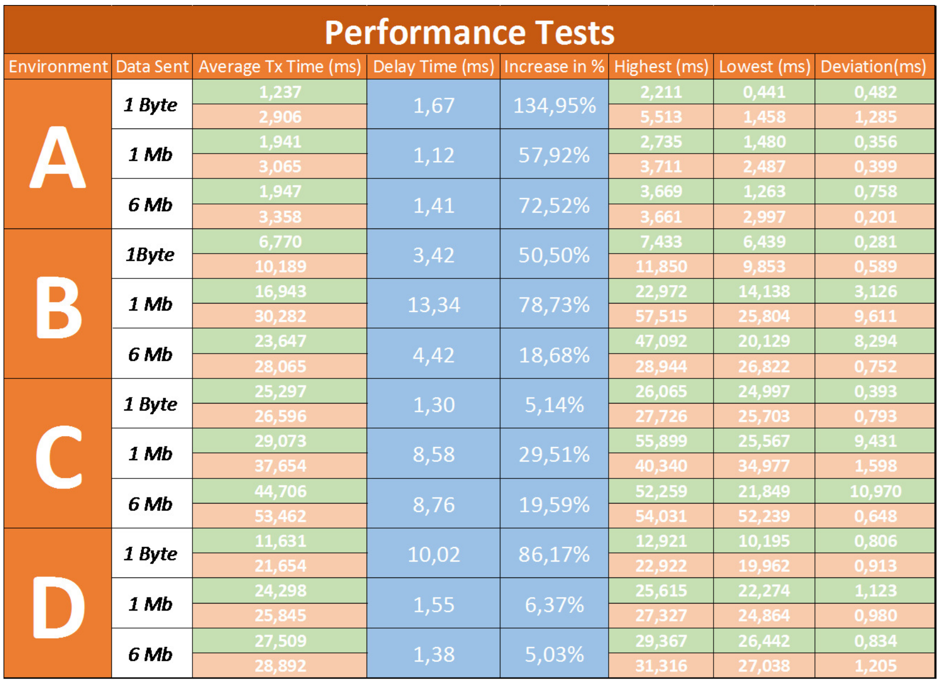
Despite the fact that the sources of XtratuM did not contain much information, further research of just this hyervisor could be made. More complex virtual environments with real time operating systems can be put under tests to see how they behave.

From the environmental impact aspect of the virtualization, analysis of pollution emissions can be elaborated to obtain actual real values of such factor. Perhaps an economic analysis of savings can be computed to asses the cost-effectiveness of using virtual environments.

# Appendix A

# Time Measurements

## A.1 Xen Time Measurements

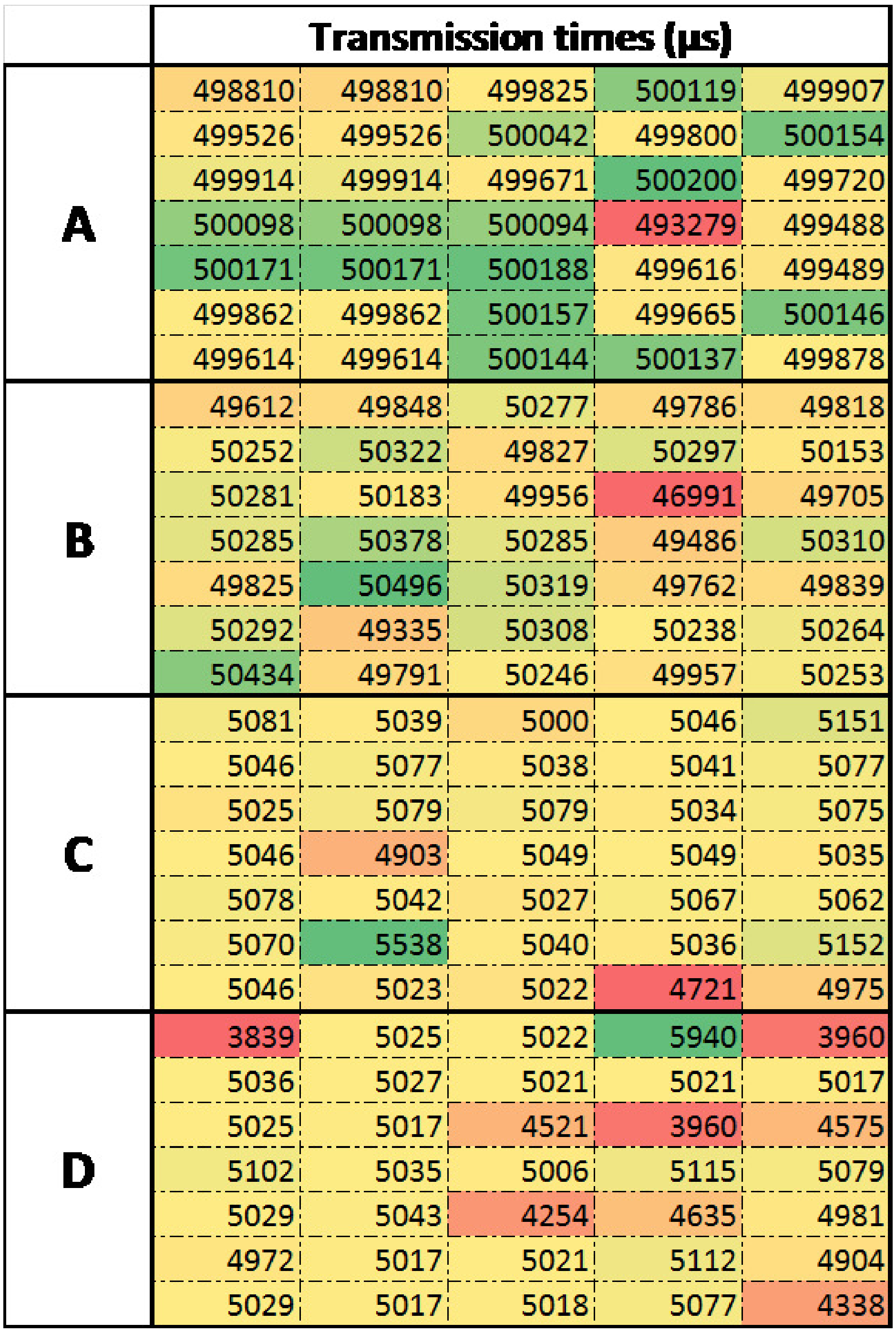


**Performance Tests**

| Environment | Data Sent | Average Tx Time (ms) | Delay Time (ms) | Increase in % | Highest (ms) | Lowest (ms) | Deviation(ms) |
|---|---|---|---|---|---|---|---|
| A | *1 Byte* | 1,237 | 1,67 | 134,95% | 2,211 | 0,441 | 0,482 |
| | | 2,906 | | | 5,513 | 1,458 | 1,285 |
| | *1 Mb* | 1,941 | 1,12 | 57,92% | 2,735 | 1,480 | 0,356 |
| | | 3,065 | | | 3,711 | 2,487 | 0,399 |
| | *6 Mb* | 1,947 | 1,41 | 72,52% | 3,669 | 1,263 | 0,758 |
| | | 3,358 | | | 3,661 | 2,997 | 0,201 |
| B | *1Byte* | 6,770 | 3,42 | 50,50% | 7,433 | 6,439 | 0,281 |
| | | 10,189 | | | 11,850 | 9,853 | 0,589 |
| | *1 Mb* | 16,943 | 13,34 | 78,73% | 22,972 | 14,138 | 3,126 |
| | | 30,282 | | | 57,515 | 25,804 | 9,611 |
| | *6 Mb* | 23,647 | 4,42 | 18,68% | 47,092 | 20,129 | 8,294 |
| | | 28,065 | | | 28,944 | 26,822 | 0,752 |
| C | *1 Byte* | 25,297 | 1,30 | 5,14% | 26,065 | 24,997 | 0,393 |
| | | 26,596 | | | 27,726 | 25,703 | 0,793 |
| | *1 Mb* | 29,073 | 8,58 | 29,51% | 55,899 | 25,567 | 9,431 |
| | | 37,654 | | | 40,340 | 34,977 | 1,598 |
| | *6 Mb* | 44,706 | 8,76 | 19,59% | 52,259 | 21,849 | 10,970 |
| | | 53,462 | | | 54,031 | 52,239 | 0,648 |
| D | *1 Byte* | 11,631 | 10,02 | 86,17% | 12,921 | 10,195 | 0,806 |
| | | 21,654 | | | 22,922 | 19,962 | 0,913 |
| | *1 Mb* | 24,298 | 1,55 | 6,37% | 25,615 | 22,274 | 1,123 |
| | | 25,845 | | | 27,327 | 24,864 | 0,980 |
| | *6 Mb* | 27,509 | 1,38 | 5,03% | 29,367 | 26,442 | 0,834 |
| | | 28,892 | | | 31,316 | 27,038 | 1,205 |

Figure A.1: Xen Table of Time Measurements

## A.2 XtratuM Time Measurements

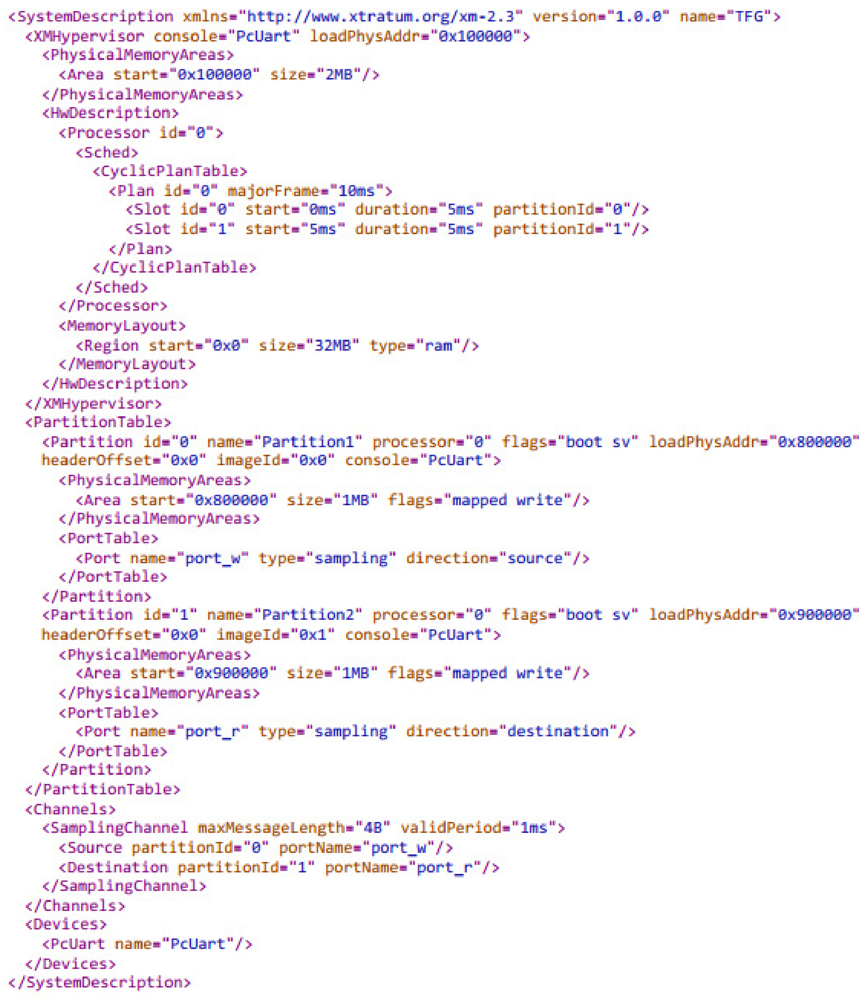


|  | Transmission times (µs) |  |  |  |  |
|---|---|---|---|---|---|
| A | 498810 | 498810 | 499825 | 500119 | 499907 |
|  | 499526 | 499526 | 500042 | 499800 | 500154 |
|  | 499914 | 499914 | 499671 | 500200 | 499720 |
|  | 500098 | 500098 | 500094 | 493279 | 499488 |
|  | 500171 | 500171 | 500188 | 499616 | 499489 |
|  | 499862 | 499862 | 500157 | 499665 | 500146 |
|  | 499614 | 499614 | 500144 | 500137 | 499878 |
| B | 49612 | 49848 | 50277 | 49786 | 49818 |
|  | 50252 | 50322 | 49827 | 50297 | 50153 |
|  | 50281 | 50183 | 49956 | 46991 | 49705 |
|  | 50285 | 50378 | 50285 | 49486 | 50310 |
|  | 49825 | 50496 | 50319 | 49762 | 49839 |
|  | 50292 | 49335 | 50308 | 50238 | 50264 |
|  | 50434 | 49791 | 50246 | 49957 | 50253 |
| C | 5081 | 5039 | 5000 | 5046 | 5151 |
|  | 5046 | 5077 | 5038 | 5041 | 5077 |
|  | 5025 | 5079 | 5079 | 5034 | 5075 |
|  | 5046 | 4903 | 5049 | 5049 | 5035 |
|  | 5078 | 5042 | 5027 | 5067 | 5062 |
|  | 5070 | 5538 | 5040 | 5036 | 5152 |
|  | 5046 | 5023 | 5022 | 4721 | 4975 |
| D | 3839 | 5025 | 5022 | 5940 | 3960 |
|  | 5036 | 5027 | 5021 | 5021 | 5017 |
|  | 5025 | 5017 | 4521 | 3960 | 4575 |
|  | 5102 | 5035 | 5006 | 5115 | 5079 |
|  | 5029 | 5043 | 4254 | 4635 | 4981 |
|  | 4972 | 5017 | 5021 | 5112 | 4904 |
|  | 5029 | 5017 | 5018 | 5077 | 4338 |

Figure A.2: XtratuM Table of Time Measurements

# Appendix B

# Configaration File XtratuM - *cf.ia32.xml*



```
<SystemDescription xmlns="http://www.xtratum.org/xm-2.3" version="1.0.0" name="TFG">
  <XMHypervisor console="PcUart" loadPhysAddr="0x100000">
    <PhysicalMemoryAreas>
      <Area start="0x100000" size="2MB"/>
    </PhysicalMemoryAreas>
    <HwDescription>
      <Processor id="0">
        <Sched>
          <CyclicPlanTable>
            <Plan id="0" majorFrame="10ms">
              <Slot id="0" start="0ms" duration="5ms" partitionId="0"/>
              <Slot id="1" start="5ms" duration="5ms" partitionId="1"/>
            </Plan>
          </CyclicPlanTable>
        </Sched>
      </Processor>
      <MemoryLayout>
        <Region start="0x0" size="32MB" type="ram"/>
      </MemoryLayout>
    </HwDescription>
  </XMHypervisor>
  <PartitionTable>
    <Partition id="0" name="Partition1" processor="0" flags="boot sv" loadPhysAddr="0x800000"
    headerOffset="0x0" imageId="0x0" console="PcUart">
      <PhysicalMemoryAreas>
        <Area start="0x800000" size="1MB" flags="mapped write"/>
      </PhysicalMemoryAreas>
      <PortTable>
        <Port name="port_w" type="sampling" direction="source"/>
      </PortTable>
    </Partition>
    <Partition id="1" name="Partition2" processor="0" flags="boot sv" loadPhysAddr="0x900000"
    headerOffset="0x0" imageId="0x1" console="PcUart">
      <PhysicalMemoryAreas>
        <Area start="0x900000" size="1MB" flags="mapped write"/>
      </PhysicalMemoryAreas>
      <PortTable>
        <Port name="port_r" type="sampling" direction="destination"/>
      </PortTable>
    </Partition>
  </PartitionTable>
  <Channels>
    <SamplingChannel maxMessageLength="4B" validPeriod="1ms">
      <Source partitionId="0" portName="port_w"/>
      <Destination partitionId="1" portName="port_r"/>
    </SamplingChannel>
  </Channels>
  <Devices>
    <PcUart name="PcUart"/>
  </Devices>
</SystemDescription>
```

Figure B.1: Xen Table of Time Measurements

# Appendix C

# Summary

Technology must keep up with the needs of the new innovative optimized systems. If the automotive or aeronautic technology field is taken as a reference, an increasing tendency of using pure electronically systems to handle complete systems can be observed. In modern cars, barely anything is still driven manually in a mechanic manner.

All electronic systems need a computational circuit to perform all the arithmetic calculations as well as logic, I/O and control operations. Furthermore additional complex hardware components are required to run the systems. Due to the large amount of different electronic systems found multiple computational units may be required. This high demand of units has a negative impact on the cost effectiveness of the aircraft or cars. In addition physical available weight and space are affected.

The possibility of making use of virtual environments offers a advantages such as improving the execution of specific applications. Furthermore, physical space reduction is perhaps the most obvious advantage being able to run all systems from the same computational unit.

However this possibility of turning multiple machines into a single one, keeping up the performance of the different virtual machines is not as straight forwarded. There are many factors that must be examined in depth before knowing whether the system is capable of running in a virtual environment.

One of the more critical aspect, regardless of whether the virtual CPUs, or the shared storage are good enough for the system, is the connectivity between the different systems.

Complex data cross reference operations are made to calculate millions of instructions, and outcomes. This requires a perfect connectivity pattern among the different systems. And here lies the crucial aspect of virtualization. Is the transmission using this technology fast enough? Is it as reliable as traditional communication systems? If the communication would fail, the system would turn useless.

As communication is not entirely instantaneous, there always exist a transmission delay. These systems are designed to cope with a time delay tolerance. This delay inevitably is affected when operate within a virtual layer.

The aim of this project is to contribute towards the analysis of temporal behaviour of virtualized systems by means of performing structured performance tests on two largely different virtual environments and capturing delivery times.

On the one hand, a virtual environment is set up with **Xen**, and on the other hand a virtual environment is set up with **XtratuM**. Two different hypervisors. XtratuM is used for embedded real time systems, and the opensource Xen hypervisor offers a large compatibility with many different Unix and Linux distributions. [6]

The first part of the project, related to Xen will aim to determine the time cost of the virtual implementation of a complex system composed of multiple virtual machines running separate OSes.

In order to obtain clear and usable results from the trials four different environments will be developed. The system will be tested making use of a middleware, an **IceStorm** service application will be developed. ***Subscriber*** and ***Publisher*** will be programmed and executed in the multiple environments.

On the second part of the project, focused on a simple bare-boned system, XtratuM will be put under analysis. The deployment of the hypervisor and the inter-partition communication will be used to analyze the effectiveness of the real time hypervisor, and prove whether the communication among virtual machines (partitions) is indeed without any delay.

Merging these two parts of the project the overall objective is to analyze the responsiveness and time cost in the transmissions within virtual environments when it comes to communication. With this analysis, an assessment can be made for, or against the use of virtual systems.

As there are no given baseline time values to compare these results with, the obtained times simply by themselves do not hold a decision of whether the development and the trials have been done correctly or not.

The time difference calculated when the system works under a high load is one of the obtained results from the first part of the project.Unfortunately no fundamental conclusion can be drawn from these results.

However if the transmission delay times are compared to the normal transmission times in the different scenarios, conclusions can be drawn.

The relationship between the delay in time increase calculated in natural units and in percentage is important result to highlight. An extend delay in milliseconds does not necessarily imply an extreme delay in time compared to the should value of the transmission time.

Moreover from the results, it can also be concluded that the size of the transmitted data affects the results. Initially it could be predicted that the larger the packet sent, the larger the percentile delay would be, however that is not always the case.

Analyzing and relating the percentile delays with the amount of data sent, yields a negative linear correlation. The percentile delay in the transmission tends to be lower when the amount of data sent increases. Making this system more optimized for larger data transfers. This idea is depicted in figure 4.10

Additionally this finding reveals that the larger the size of the transmitted data, the higher the transmission time is. A straightforward conclusion that matches the theory of communications.

Taking into account that the data has been extracted from over 300 records, and that both, the negative correlation and the higher transmission time for larger sized data, matches the theoretical background a correct performance can be assumed.

It is important to highlight that the results obtained, during the loaded stages, are distant to be comparable to real time values. On average the absolute delay is just below 5 ms, and although the time delay tolerance may vary from system to system, these results do not make Xen and Ice much of use for real time critical systems.

During the second part of the project Xtratum was analyzed. The obtained measurements resulted in a clear conclusion. The relationship between the Server's partition and the time taken to deliver the data does not oscillate more than 2.1% with respect to that time. A time variation of just a few $\mu s$ does not necessary mean a failure in the application or in the system.

Hence the system does actually communicate without any noticeable delay.

In the case where the partition execution time reaches beyond the planned time slot, it may be the case if a hypercall is executed just at the end of the time slot, the context switch will delay the beginning of the next partition. This can be seen in 6.6c. This explains why there are transmission times that may take longer than the partition duration.

Anomalous results were obtained during this second part. Here the context switch was offsetting the starting times with every cycle.And as no time margins were pre-configured, the slot times of the partitions left for execution of the applications suffered a cyclic reduction. Therefore every $n$ cycles, the client partition had been reduced up to an extend where it could not run the application satisfactory and had to wait for the next cycle to properly read the data. Note that that unread data sample is lost and will never reach the destination.

The data will just take as long to reach the destination, as the duration of the sending partition is. Hence the main and one factor that will affect this transmission time is simply the duration of the transmitting partition. However the partition's duration must be long enough for the it to create and read the port. Otherwise no transmission will occur as the idle state will be reached before achieving the successful transmission. Analogously, the receiving partition must have enough time to create and read the port.

Overall analyzing the results and applying these in a more general context, the virtual environments should be used to replace multiple separated units. This represents no legal restrictions more than complying with the license agreements. Furthermore it has a positive impact on earth's environment and may contribute to saving substantial amounts of money.

With this project it is proven that XtratuM behaves indeed in a responsive manner, with almost neglectable time delay, but with the possibility of encountering anomalies under one circumstance. On the other hand Xen was fully reliable during the trials and the only limitation of this hypervisor is the considerable delay, to bear in mind, while experience a high system load.

# Bibliography

[1] Stefan M. Petters. Real-Time Systems. 2007
*http://www.cse.unsw.edu.au/~cs9242/08/lectures/09-realtimex2.pdf.*
Addison-Wesley, Reading, Massachusetts, 1993.

[2] Ben-Ari, M., "Principles of Concurrent and Distributed Programming", Prentice Hall,1990. Ch 16

[3] Gerhard Fohler. Time Triggered and Event Triggered, Off-line Scheduling. 2004
`http://www.win.tue.nl/ johanl/educ/2IN20/TT-ET+offline.pdf`

[4] Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems", Addison-Wesley/ ACM Press. ISBN 0-201-331381

[5] Linux-KVM
`http://www.linux-kvm.org/page/MainPage`

[6] Xen Project Wiki. Support and Documentation Resources
`http://wiki.xen.org/wiki/XenOverview`

[7] XtratuM Hypervisor
`http://www.xtratum.org/`

[8] M. Masmano, I. Ripoll, A. Crespo, J.J. Metge, and P. Arberet. Xtratum: An opensource hypervisor for tsp embedded systems in aerospace. In DASIA 2009. DAta SystemsIn Aerospace., May. Istanbul 2009.

[9] S. Peiro, A. Crespo, I. Ripoll, M. Masmano ,Partitioned Embedded Architecture basedon Hypervisor: the XtratuM approach. Eighth European Dependable Computing Con-ference, EDCC-8 2010, Valencia, Spain, 28-30 April 2010. IEEE Computer Society 2010,ISBN 978-0-7695-4007-8

[10] XtratuM-Real time Nanokernel for Linux Nicholas Mc Guire Distributed & EmbeddedSystems Lab, Lanzhou Univeristy, China

[11] Open DDS Overview
`http://www.opendds.org/about.html`

[12] FENTISS /UPVLC. XtratuM Hypervisor for INTEL x86. Volume 3: User Manual. September 2012.

[13] FENTISS /UPVLC. XtratuM Hypervisor for INTEL x86. Volume 3: Reference Manual. September 2012.

[14] M. Masmano, J. Coronel, fentISS, Valencia, Spain, P. Balbastre, A. Crespo, J. Simo, S. Peiro Universitat Politecncia de Valencia, Spain "XtratuM hypervisor for mixed-criticality systems"

[15] Zeroc Inc. Ice 3.6.1 Documentation Manual
`https://download.zeroc.com/Ice/3.6/Ice-3.6.1.pdf`

[16] M. García-Valls, T. Cucinotta, C. Lu. *Challenges in real-time virtualization and predictable cloud computing.* Journal of Systems Architecture 60(9), pp. 726–740. 2014.

[17] M. García-Valls, L. Fernández Villar, I. Rodríguez López. *iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems* Transactions on Industrial Informatics 9(1), pp. 228-236. 2013.

[18] M. García-Valls, A. Alonso, J. Ruiz, A. Groba. *An architecture for a quality of service resource manager middleware for flexible multimedia embedded systems* Proc. $3^{rd}$ Int'l Conference on Software Engineering and Middleware (SEM). LNCS, vol. 2596, pp. 36–55. 2003.

[19] M. García Valls, R. Baldoni. *Adaptive middleware design for CPS: Considerations on the OS, resource managers, and the network run-time.* Proc. $14^{th}$ Workshop on Adaptive and Reflective Middleware (ARM). Co-located to ACM ACM/IFIP/USENIX Middleware. Vancouver, Canada. December 2015.

[20] J. Cano, M. García-Valls. *Scheduling component replacement for timely execution in dynamic systems.* Software: Practice and Experience, vol. 44(8), pp. 889-910. January 2013.

[21] J. Cano, M. GarcíaâValls, P. BasantaâVal. *Component framework for supporting safe and dynamic replacement in realâtime systems.* RIAI â Revista Iberoamericana de Automática e Informática Industrial, vol. 11(1), pp. 98â108. 2014.

[22] J. Duenas, A. Alonso, W. Lopes Oliveira, M. Garcia, G. Leon. *Software architecture assessment.* In: *Software architecture for product families: principles and practice.* Addison-Wesley. 2000.

[23] B. Bouyssounouse, et al. *Programming languages and real-time systems.* In: *Embedded systems design: the ARTIST roadmap for research and development.* Springer, 2005.

[24] B. Bouyssounouse, et al. *QoS Management.* In: *Embedded systems design: the ARTIST roadmap for research and development.* Springer, 2005.

[25] B. Bouyssounouse, et al. *Adaptive real-time systems development.* In: *Embedded systems design: the ARTIST roadmap for research and development.* Springer, 2005.

[26] M. García-Valls, A. Alonso, J. A. de la Puente. *A dual priority assignment mechanism for dynamic QoS resource management.* Future Generation Computer Systems, vol. 28(6), pp.902-911. June 2012.

[27] C. M. Otero Pérez, L. Steffens, P. van der Stok, S. van Loo, A. Alonso, J. Ruíz, R. J. Bril, M. García Valls. *QoS-Based Resource Management for Ambient Intelligence.* In: *Ambient Intelligence: Impact on Embedded Sytem Design*, pp. 159–182. Kluwer Academic Publishers. 2003.

[28] M. García-Valls. *Calidad de servicio en sistemas multimedia empotrados mediante gestión dinámica de recursos.* Universidad Politécnica de Madrid. (2001)

[29] M. García-Valls, A. Alonso, J.A. de la Puente. *Mode change protocols for predictable contract-based resource management in embedded multimedia systems.* In Proc. of IEEE Int'l Conference on Embedded Software and Systems (ICESS), pp. 221-230. May 2009.

[30] M. García-Valls, C. Calva-Urrego, A. Alonso, J.A. de la Puente. *Adjusting middleware knobs to suit CPS domains.* Proc. of 31$^{st}$ ACM/SIGAPP Symposium on Applied Computing (SAC), pp. 2027-2030. Pisa, Italy. April 2016.

[31] M. García-Valls. *A proposal for cost-effective server usage in CPS in the presence of dynamic client requests.* Proc. of 19th IEEE International Symposium on Real-time Distributed Computing (ISORC). York, UK. May 2016.

[32] A. Alonso, M. García-Valls, J. A. de la Puente. *Assessment of timing properties of family products.* In: ARES Workshop – Development and Evolution of Software Architectures for Product Families. LNCS, vol. 1429, pp. 161–169. Springer. 1998.

[33] M. García-Valls, D. Perez-Palacin, R. Mirandola. *Time sensitive adaptation in CPS through run-time configuration generation and verification.* Proc. of 38th IEEE Annual Computer Software and Applications Conference (COMPSAC), pp. 332–337. 2014

[34] M. M. Bersani, M. García-Valls. *The cost of formal verification in adaptive CPS. An example of a virtualized server node.* Proc. of 17th IEEE High Assurance Systems Engineering Symposium (HASE). 2016.

[35] M. García-Valls, P. Basanta-Val. *A real-time perspective of service composition: key concepts and some contributions.* Journal of Systems Architecture, vol. 59(10), pp. 1414–1423. November 2013.

[36] M. García-Valls, P. Basanta-Val. *Low complexity reconfiguration for real-time data-intensive service-oriented applications.* Future Generation Computer Systems, vol. 37, pp. 191-200. July 2014.

[37] M. García-Valls, P. Basanta -Val. *Comparative Analysis of different middleware approaches to real-time reconfiguration.* Journal of Systems Architecture, vol. 60(2), pp. 221-233. February 2014.

[38] M. García Valls, P. Basanta Val. *Usage of DDS data-centric paradigm for remote monitoring and control laboratories.* IEEE Transactions on Industrial Informatics, vol. 9(1), pp. 567-574. February 2013.

[39] I. Rodríguez-López, M. García-Valls. *Architecting a Common Bridge Abstraction over Different Middleware Paradigms.* Ada-Europe 2011, pp. 132-146. Edimburgh, UK. June 2011.

[40] M. García-Valls, F. Ibánez-Vázquez. *Integrating Middleware for Timely Reconfiguration of Distributed Soft Real-Time Systems with Ada DSA.* Ada-Europe 2012, pp. 35-48. Stockholm, Sweden. July 2012.

[41] M. García-Valls, P. Basanta-Val, I. Estévez-Ayres. *Adaptive real-time video transmission over DDS*. Proc. of 8th IEEE International Conference on Industrial Informatics,. pp. 130–135. Osaka, Japan. July 2010.

[42] M. García-Valls, I. Estévez-Ayres, P. Basanta-Val. *CoSeRT: a framework for composing service-based real-time applications*. Proc. of Business Management Workshops. Lecture Notes in Computer Science, vol. 3812, pp. 329-341. 2005.

[43] M. García-Valls, A. Crespo, J. Vila. *Resource management for mobile operating systems based on the active object model.* International Journal of Computer Systems Science & Engineering, vol. 28(4), 195–205. 2013.

[44] R. Serrano-Torres, M. García-Valls, P. Basanta-Val. *Virtualizing DDS middleware: performance challenges and measurements*. Proc. of 11th IEEE International Conference on Industrial Informatics (INDIN). July 2013.

[45] M. García-Valls, D. Perez-Palacin, R. Mirandola. *Time-sensitive adaptation in CPS through run-time configuration generation and verification.* Proc. of 38th IEEE Computer Software and Applications Conference (COMPSAC). Vasteras, Sweden. July 2014.

[46] M. García-Valls, F Gómez-Molinero. *Real-time reconfiguration in complex embedded systems: A vision and its reality.* Proc. of 9th IEEE International Conference on Industrial Informatics (INDIN). Lisbon, Portugal. July 2011.

[47] J. García-Muñoz, M. García-Valls, J. Escribano-Barreno: *Improved Metrics Handling in SonarQube for Software Quality Monitoring.* Proc. of 13th International Conference Distributed Computing and Artificial Intelligence (DCAI). Sevilla, Spain. 2016.

[48] J. Escribano-Barreno, J. García-Muñoz, M-.García-Valls, M.: *Integrated metrics handling in open source software quality management platforms.* ITNG. (2016)

[49] L. Cappa-Banda, M. García-Valls. *Experimenting with a load-aware communication middleware for CPS domain.* ITNG. (2016)