UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Telemática

TRABAJO FIN DE GRADO:

ESTUDIO DEL COMPORTAMIENTO TEMPORAL DE ENTORNOS DISTRIBUIDOS VIRTUALIZADOS

Autor: Jorge Domínguez Poblete

Tutora:
Marisol García Valls

Agradecimientos

En primer lugar, agradecer a mi tutora Marisol García Valls por sus rápidas respuestas, buenas ideas y mejores orientaciones. También agradecerle el haber confiado en mí para llevar a cabo este Trabajo de Fin de Grado de investigación.

Dar las gracias a mi madre, a mi padre y a mi hermana por haberme apoyado durante la carrera, y aguantado en épocas de prácticas y exámenes (casi todo el tiempo).

Dar las gracias a la *CRD* de Telemática, es decir, a Cristian, Raúl y Dani *con vosotros empezó todo*. Porque a pesar de lo mucho que nos hemos quejado, puede (o no) que algún día echemos de menos esas tardes en los que nos volvíamos locos intentando acabar prácticas, estudiando los exámenes por *Skype*, jugando a los cabezones y de cervezas después de los exámenes.

Por último, y no menos importante, dar las gracias a mis amigos por estar en los momentos buenos, malos y regulares. No pongo nombres porque, por suerte, me faltaría papel. Sois unos grandes.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introduction

In this chapter the research context of the work, the social context and the motivation that has driven the realization of this Final Degree Work, its objectives and finally the structure of the document are described.

1.1. Social context and motivation

The use of distributed systems with temporal requirements is becoming more common in our society. Also, the improvements to the capacity and reliability of communication networks make possible the increasing usage of networks in real-time domains. Examples are car systems that contain hundreds of ECUs connected by different network segments, ranging from CAN (for the most critical systems) to Ethernet for less reliable parts.

The middleware provides communication services which enables transmission of data in distributed systems and increases development productivity. This is due to the fact that middleware abstracts the low-level details of the network and operating system of each machine. Another important contribution is that it allows applications to focus purely on the application-level logic.

Ice and DDS are two middleware technologies that can be used in distributed embedded applications. Ice offers a very efficient and simple programming model in an essentially client-server paradigm. It offers an efficient and small footprint implementation that is a natural Corba evolution. On another hand, DDS has a powerful asynchronous communication paradigm, offering the possibility of fine tuning the communication with QoS (quality of service) parameters.

Currently, virtualization technology is flooding the market as it favors server consolidation, reducing maintenance of the infrastructure. Such a technology has also evolved to the time-sensitive and real-time domains as explained in [21].Therefore, it is evident that the usage of virtualization technology will introduce im-

portant benefits to the economy. However, the effect of such a technology in the performance and temporal behaviour of applications has to be carefully studied in order to determine the appropriateness of a virtualized deployment.

An example of these savings can be seen in the study conducted by *Southwestern Illinois College*. In this study it is analysed the costs of adding 35 servers using virtualization and without using it. As we can see in the picture below 1.1, the cost savings are significant. It should also be noted that the energy savings from using virtualization is also beneficial to the environment by saving on electricity consumption and reducing emissions of CO_2. [1]

3 Year Total Cost of Ownership

	Without VMware	With VMware	Savings
Direct Costs			
VMware Services	$ -	$ 17,000	$ (17,000)
VMware Software & Support	$ -	$ 38,938	$ (38,938)
Third Party Software & Support	$ -	$ -	$ -
Server Hardware	$ 229,500	$ 27,000	$ 202,500
Network Costs	$ 49,500	$ 18,000	$ 31,500
SAN Costs	$ -	$ 30,000	$ (30,000)
Total Direct Costs	$ 279,000	$ 130,938	$ 148,063
Indirect Costs			
Data Center	$ 136,823	$ 16,965	$ 119,858
Server Provisioning	$ 11,745	$ 1,980	$ 9,765
Server Administration	$ 50,760	$ 55,080	$ (4,320)
Procurement	$ 8,750	$ 750	$ 8,000
Total Indirect Costs	$ 208,078	$ 74,775	$ 133,303
Total Cost of Ownership	$ 487,078	$ 205,712	$ 281,366

Figura 1.1: Savings through the use of virtualization

The main focus of this Final Degree Work on the analysis of the temporal behaviour of distributed applications that have temporal requirements when they are implemented using middleware technologies (essentially Ice and DDS) both over a bare machine and over a virtualized environment. Both settings are compared to extract the actual costs incurred by the technology over different scenarios with and without significant load and for different exchanged data sizes. Moreover, the work uses a base application for monitoring environmental data and for its transmission to a server and it explains how it would be implemented on a cloud

platform.

1.2. Aims

The main objectives of this Final Degree Work are as the listed below:

- Studies on virtualization software, cloud platforms, communications middleware and Linux scheduler tasks.
- Development of a test code and installation tools to perform the test.
- Running the test and conclusions.
- Application development and integration in OpenStack.
- Conclusions and possible improvements.

1.3. Structure of Final Degree Work

Below you can find a brief description of the chapters of the Final Degree Work:

- **Chapter 1. Introduction:** where the reasons which prompted the realization of the Final Degree Work, context and objectives are exposed, written in English.
- **Chapter 2. Introduction:** where the reasons which prompted the realization of the Final Degree Work, context and objectives are exposed.
- **Chapter 3. Estado del arte:** it is described the technological context used in the Final Degree Work as virtualization software, cloud platforms, communications Middleware and Linux scheduler.
- **Chapter 4. Análisis de las prestaciones de Ice:** the temporal behaviour obtained in the various scenarios studied by using Ice.
- **Chapter 5. Análisis de las prestaciones de DDS:** the temporal behaviour obtained in various scenarios studied by using DDS.
- **Chapter 6. Creación de una aplicación DDS y su integración a la nube:** application development and integration of DSS into a cloud platform.
- **Chapter 7. Conclusions and future works:** the results and possible extensions of the Final Degree Work are described in English.

- **Chapter 8. Conclusiones y trabajos futuros:** it is described the results and possible extensions of the Final Degree Work.
- **Chapter 9. Anexos:** it is described detailed planning, the budget, KVM installation and configuration of a public bridge.

Capítulo 2

Introducción

En este capítulo se describe el contexto de investigación en el que se enmarca el trabajo, contexto social y la motivación que ha impulsado a la realización de este Trabajo de Fin de Grado, sus objetivos y, finalmente, la estructura del documento.

2.1. Contexto social y motivación

El uso de sistemas distribuidos cada vez es más común en nuestra sociedad. Además, la mejora de las redes de comunicaciones que permiten comunicaciones más rápidas y fiables han hecho posible esta proliferación de las comunicaciones con requisitos de tiempo real. Ejemplos de ello son los sistemas del automóvil, que contienen cientos de centralitas conectadas por segmentos de red diferentes, que van desde la CAN (para los sistemas más críticos), a Ethernet para las partes menos fiables.

El middleware de comunicaciones hace posible la transmisión de datos en sistemas distribuidos y aumenta la productividad de su desarrollo, ya que abstrae los detalles de bajo nivel de la red y del sistema operativo de cada máquina, otra aportación importante es que permite a las aplicaciones centrarse sólo en la lógica a nivel de aplicación.

El middleware de comunicaciones que se va a usar para la investigación es Ice y DDS. El motivo de la selección de Ice es porque es simple de usar y muy eficiente para entornos cliente-servidor. Por otro lado, DDS porque tiene un paradigma de comunicación muy potente, desacoplado y, además, permite usar ciertos parámetros de calidad de servicio.

Ice y DDS son dos tecnologías middleware distintas que pueden ser usadas en aplicaciones distribuidas empotradas. Ice ofrece un eficiente y simple modelo de programación en un paradigma cliente-servidor. Lo que ofrece una eficiente implementación de una evolución de Corba. Por otro lado, DDS tiene un paradig-

ma asíncrono de comunicación, que ofrece la posibilidad de modificar de forma precisa la comunicación con parámetros de calidad de servicio.

Actualmente, la tecnología de virtualización está inundando el mercado favoreciendo su consolidación en entornos de servidor reduciendo el coste de mantenimiento de la infraestructura. Esta tecnología ha evolucionado en las áreas de tiempo real como se explicó en [21]. Además, evidentemente el uso de tecnología de virtualización puede introducir importantes beneficios en las cuentas de una empresa. Sin embargo, el uso de esa tecnología y su comportamiento en las aplicaciones debe ser cuidadosamente estudiado para determinar si el despliegue de la virtualización es apropiado.

Además, la virtualización ofrece múltiples ventajas: proporciona sistemas flexibles y fácilmente modificables, ahorro de costes debido al uso más eficiente del hardware, facilita la instalación de máquinas nuevas y su migración. Todos estos beneficios de la virtualización hacen posible el despliegue de sistemas distribuidos sobre entornos virtualizados. Un ejemplo de este ahorro lo podemos extraer de un estudio llevado a cabo por *Southwestern Illinois College*. En este estudio se analizaron los costes de añadir 35 servidores mediante el uso de la virtualización y sin él. Como podemos observar en la imagen inferior 2.1, el ahorro económico es significativo, también hay que destacar que el ahorro energético del uso de virtualización también es beneficioso para el medioambiente al ahorrar en consumo eléctrico y disminuir las emisiones de CO_2. [1]

El objetivo principal de este Trabajo de Fin de Grado consiste en el análisis del comportamiento temporal de aplicaciones distribuidas que tienen requerimientos temporales cuando son implementadas usando tecnologías middleware (principalmente Ice y DDS) sobre máquina física y en entornos virtualizados. Los dos escenarios son comparados para comprobar el coste incurrido por el uso de esta tecnología sobre diferentes escenarios con y sin carga sintética y mediante el intercambio de mensajes de diferente tamaño. Además, el trabajo usa una aplicación base para monitorizar datos de tipo ambiental y su transmisión a un servidor, se explica su integración en una plataforma en la nube.

2.2. Objetivos

Los principales objetivos son:

- Estudio de software de virtualización, plataformas en la nube, middleware de comunicaciones y planificador de tareas Linux.
- Desarrollo del software e instalación de herramientas realización de los test.
- Realización de los test y conclusiones.

3 Year Total Cost of Ownership

	Without VMware	With VMware	Savings
Direct Costs			
VMware Services	$ -	$ 17,000	$ (17,000)
VMware Software & Support	$ -	$ 38,938	$ (38,938)
Third Party Software & Support	$ -	$ -	$ -
Server Hardware	$ 229,500	$ 27,000	$ 202,500
Network Costs	$ 49,500	$ 18,000	$ 31,500
SAN Costs	$ -	$ 30,000	$ (30,000)
Total Direct Costs	$ 279,000	$ 130,938	$ 148,063
Indirect Costs			
Data Center	$ 136,823	$ 16,965	$ 119,858
Server Provisioning	$ 11,745	$ 1,980	$ 9,765
Server Administration	$ 50,760	$ 55,080	$ (4,320)
Procurement	$ 8,750	$ 750	$ 8,000
Total Indirect Costs	$ 208,078	$ 74,775	$ 133,303
Total Cost of Ownership	$ 487,078	$ 205,712	$ 281,366

Figura 2.1: Ahorro gracias a virtualización

- Desarrollo de código de la aplicación e integración en OpenStack.
- Conclusiones y elaboración de posibles líneas de mejora.

2.3. Estructura del Trabajo de Fin de Grado

A continuación, se aporta una breve descripción de los capítulos que componen el Trabajo de Fin de Grado:

- **Capítulo 1. Introduction:** en donde se exponen los motivos que han impulsado la realización del Trabajo de Fin de Grado, contexto y objetivos, en inglés.
- **Capítulo 2. Introducción:** en donde se exponen los motivos que han impulsado la realización del Trabajo de Fin de Grado, contexto y objetivos.
- **Capítulo 3. Estado del arte:** se describe de forma teórica el contexto tecnológico usado en el Trabajo de Fin de Grado como Software de virtualiza-

ción, plataformas en la nube, Middleware de comunicaciones y el planificador de tareas de Linux.

- **Capítulo 4. Análisis de las prestaciones de Ice:** se estudia el comportamiento temporal obtenido en los diversos escenarios mediante el uso de Ice.
- **Capítulo 5. Análisis de las prestaciones de DDS:** se estudia el comportamiento temporal obtenido en diversos escenarios mediante el uso de DDS.
- **Capítulo 6. Creación de una aplicación DDS y su integración a la nube:** desarrollo de una aplicación de DDS y su integración a una plataforma en la nube.
- **Capítulo 7. Conclusions and future works:** se expone de forma razonada los resultados obtenidos y posibles extensiones sobre el Trabajo de Fin de Grado, en inglés.
- **Capítulo 8. Conclusiones y trabajos futuros:** se expone de forma razonada los resultados obtenidos y posibles extensiones sobre el Trabajo de Fin de Grado.
- **Capítulo 9. Anexos:** se detalla la instalación de KVM, configuración de un bridge público, planificación y presupuesto.

Capítulo 3

Estado del arte y herramientas software utilizadas

3.1. Software de virtualización

La *virtualización* consiste en la acción de simular mediante software distintos elementos, que, generalmente forman parte de un ordenador como pueden ser componentes hardware tal y como tarjeta gráfica, memoria RAM, procesador y, también un sistema operativo o distintas configuraciones de red. [6]

La forma más típica de virtualizar es mediante el uso de las llamadas *máquinas virtuales*. Una *máquina virtual* normalmente es un proceso más en el sistema operativo, con la peculiaridad de que esta trata de abstraer los componentes hardware de los que dispone la máquina física y el sistema operativo que corre en ese momento, y poder crear una con configuración con los componentes, tanto hardware como software, que se adapte a los intereses específicos para los que se crea necesario. Por tanto, permite reproducir el funcionamiento de una plataforma con unas características determinadas y se comporta como un programa más que se ejecuta en nuestro ordenador.

Una *máquina virtual* puede tener distintos fines, uno de ellos puede ser el de simular múltiples máquinas físicas sobre el mismo hardware y, de esta manera poder hacer un uso más eficiente y permitir un considerable ahorro monetario. Además, permite probar un sistema o plataforma con los parámetros deseados antes de llevar a cabo un despliegue físico. Igualmente permite ejecutar software que no puede ser ejecutado en el sistema operativo de la máquina física, por ejemplo, Microsoft Office en Linux. [7]

Por todo ello, las *máquinas virtuales* siempre han sido muy interesantes en el entorno de empresas tecnológicas ya que permiten crear sistemas flexibles adaptando su configuración. También favorecen la escalabilidad de un sistema de forma sostenible económicamente. Algunos ejemplos de *software de virtualización* más conocidos según su entorno son:

- *Virtual Box*, usado en entornos de PC de sobremesa.
- Por otro lado, son conocidos para el entorno de servidor tanto *VMWare* como *KVM*.

Según su funcionamiento se puede clasificar las *máquinas virtuales* en dos grandes grupos [8]:

- **Máquinas virtuales de proceso o hypervisor de tipo 1.** Permiten que se ejecuten varias *máquinas virtuales* en una misma máquina física. Esto se lleva a cabo mediante el *hypervisor*, que es una capa de software que puede ejecutar directamente sobre el hardware o bien sobre un sistema operativo. Gracias a este tipo de *máquinas virtuales* podemos hacer que varios sistemas operativos convivan con independencia unos de otros, pero, compartiendo el hardware de la máquina física sobre la cual están ejecutando. Puede ser útil, en el caso de querer tener varios servidores en la misma máquina física. Figura 3.1.

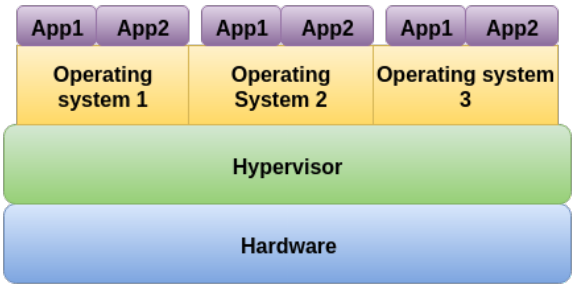

Figura 3.1: Hypervisor sobre hardware

- **Máquinas virtuales de sistema o hypervisor de tipo 2.** Este tipo de *máquina virtual* soporta únicamente un proceso y, trata de proporcionar un espacio multiplataforma e independiente. Un ejemplo de este tipo de *máquina virtual* es la *máquina virtual* de Java. Figura 3.2.

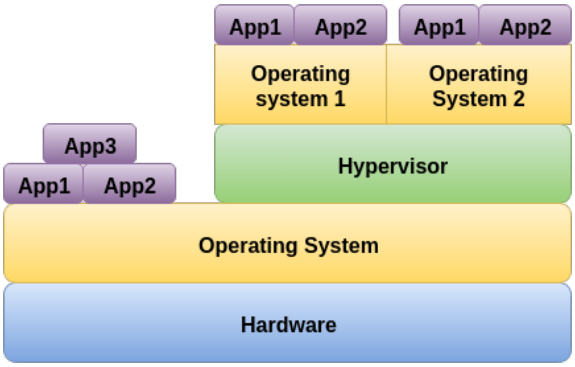

Figura 3.2: Hypervisor sobre sistema operativo

3.1.1. KVM

KVM proporciona *virtualización* completa para Linux, siempre que el procesador de la máquina física permita *virtualización* como lo hacen AMD-V o Intel-VT. KVM se trata de un *hypervisor* de tipo 2 y es software de código abierto y, permite ejecutar múltiples sistemas operativos o virtualizar hardware. Si la CPU no soporta *virtualización*, KVM usa QEMU para traducir las instrucciones de CPU virtual en la máquina física. Adicionalmente, gracias a QEMU conseguimos mejorar el rendimiento general de KVM.

A pesar de tratarse de software código abierto, KVM compite con Virtual Box y VMWare las cuales son de código propietario gracias a su fácil escalabilidad y gran rendimiento todo ello de forma libre. Viene integrado por defecto desde el kernel 2.6.20 de Linux. Al tratarse de un software de código abierto cuenta con una gran comunidad de usuarios detrás y, además, cuenta con una muy buena documentación. Por otro lado, KVM es fácilmente integrable en la nube. El proyecto de KVM es respaldado por grandes compañías como Red Hat, IBM, HP, Intel y algunas otras.

Es muy interesante el uso de KVM para sistemas de tiempo real. Estos sistemas se caracterizan por tener que garantizar una respuesta en un tiempo determinista. Normalmente reciben información, la procesan, y dan una respuesta en un tiempo pequeño (entre milisegundos y microsegundos). Los sistemas de tiempo real son usados en aplicaciones de aviónica, en el mercado de valores, coches inteligentes, entre otros muchos ejemplos. [9]

KVM puede ser utilizado para implementar sistemas de tiempo real, de esta manera se puede utilizar una máquina física para reproducir varios sistemas de

tempo real, independientes entre sí, mediante el uso de KVM. Para ello se hace uso de Control Groups mediante los cuales se puede asignar una serie de tareas a una serie de recursos como memoria, ancho de banda o prioridad de asignación de recursos, y de esta manera, asegurar el buen funcionamiento de KVM. Gracias a herramientas como Control Groups, se puede asegurar la eficiencia del sistema y así hacer posible el buen funcionamiento como un sistema de tiempo real.

Además, me parece importante destacar la función de *libvirt*. *Libvirt* es un conjunto de herramientas software mediante el cual nos permite que las funciones de los *hypervisor* (Virtualbox, KVM) se lleven a cabo de forma correcta. Algunas de las funciones llevadas a cabo por libvirt son: control de redes virtualizadas, control de la ejecución de las *máquinas virtuales*, realizar soporte remoto de las *máquinas virtuales*, etc. [12]

Una vez que ya ha instalado KVM, ya se puede crear la primera *máquina virtual* en la cual podemos modificar multitud de parámetros para adaptarlos a necesidades específicas. KVM puede virtualizar una gran cantidad de parámetros hardware: procesador, RAM, tarjeta de vídeo y formas de conectividad red.

Una de las características de KVM es usar *overcommit* lo que se traduce en poder usar más recursos de forma virtual que los disponibles de forma física. Es decir, se puede usar más núcleos que los disponibles en el procesador entre otras cosas, también se puede modificar hebras por núcleo, número de zócalos y modelo de CPU. Respecto a la memoria RAM se puede asignar hasta un poco más de memoria que la disponible en el host físico, esto se hace gracias a que podemos aumentarla mediante una porción de memoria de swap. [10]

También hay que mencionar la configuración de la red en KVM, para proceder a su explicación se van a mencionar dos de las muchas posibles configuraciones:

- **Bridge privado.** Es útil cuando lo único que queremos es comunicar *máquinas virtuales* y dentro de un mismo host físico. Por defecto, KVM crea una subred virtual privada en la cual está el host físico y el resto de *máquinas virtuales*. El punto negativo es, que no podremos acceder desde fuera de esta subred virtual a las *máquinas virtuales*.

- **Bridge público.** Nos permite tener una IP pública en el rango de nuestra subred, de manera que los paquetes que recibe y envía por la red lo hace con una IP distinta a la del host físico. Utiliza una interfaz del host físico para enviar y recibir los paquetes de la red. De esta manera nuestra *máquina*

virtual puede comunicarse con otros dispositivos de nuestra red y viceversa, con independencia de su host físico.

En ambas configuraciones en el caso de coexistir varias *máquinas virtuales* cada una de ellas tendrá una IP distinta.

La configuración de un bridge público en KVM se explica de forma detallada en nuestro apéndice Configurando un bridge público en KVM

La instalación de KVM se explica de forma detallada en el apéndice Instalación KVM

3.1.2. Xen

Se trata de un *hypervisor* que se ejecuta directamente sobre el hardware de código abierto, inicialmente desarrollado por la universidad de Cambridge en 2003, en la actualidad toda una comunidad de desarrolladores trata cada día de implementar mejoras. Algunas de las empresas colaboradoras son ARM, Citrix, Verizon, Intel, Google, AMD, Oracle, entre otras.

Xen posee características similares al resto de *hypervisores* y, su objetivo principal es el de poder abstraer las características hardware y software de un ordenador. Xen es usado en múltiples aplicaciones como *virtualización* de servidores, plataformas en la nube, *virtualización* de escritorio, aplicaciones de seguridad, sistemas empotrados, etc.

Algunas de las características de Xen son las siguientes [**?**]:

- **Código abierto.** De esta manera es una gran comunidad de usuarios y empresas colaboradoras la que se encarga del desarrollo del proyecto, y no una única empresa de forma exclusiva.
- **Paravirtualización.** Se trata de una técnica mediante la cual la *máquina virtual* puede alcanzar altos rendimientos frente a otras técnicas usadas para la *virtualización*.
- **Escalabilidad.** Beneficio derivado de la *virtualización* que es la posibilidad de aumentar la infraestructura de forma eficiente en cuanto a costes.

- **Rendimiento.** Distintas pruebas de rendimiento avalan a Xen como una de las mejores soluciones en cuanto *virtualización* frente a otras como KVM y VirtualBox.
- **Seguridad.** Otra de las principales razones del uso de *máquinas virtuales* es la seguridad ya que separa al sistema anfitrión de la *máquina virtual*. Xen tiene una estricta política en cuanto a seguridad y trata de resolver lo más rápido posible eventuales vulnerabilidades.
- **Flexibilidad.** Xen proporciona una gran cantidad de opciones para la *virtualización*, y se acomoda en función de las características del equipo anfitrión. También nos permite realizar múltiples modificaciones a nuestro gusto.
- **Modularidad.** Gracias a esta propiedad, Xen puede ser más robusto, escalable y seguro.
- **Migración de máquinas virtuales.** De esta forma podemos realizar balances de carga entre múltiples servidores y conseguir un rendimiento más eficiente.

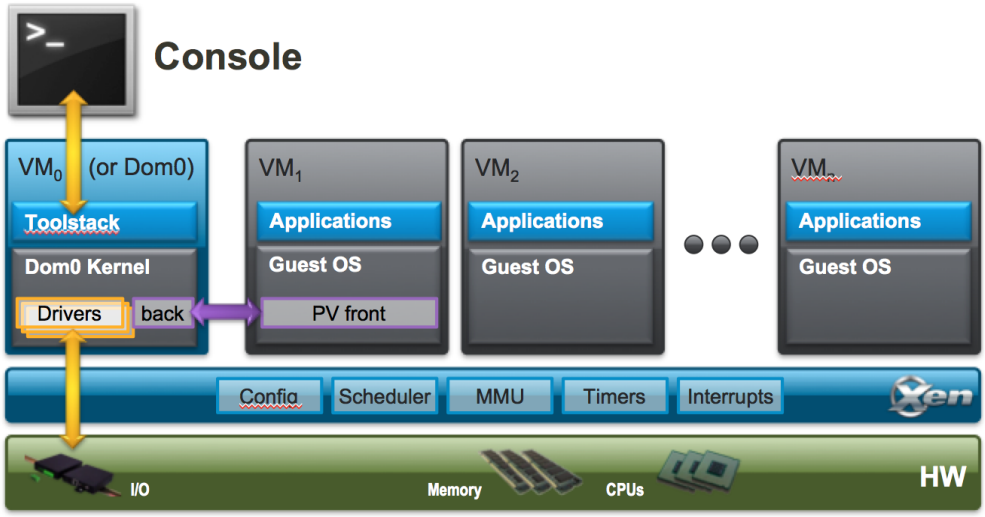

Figura 3.3: Esquema de la arquitectura de Xen (fuente `http://wiki.xen.org/`)

Además, Xen es usado para la *virtualización* de plataformas en la nube, como por ejemplo en el servicio EC2 (Elastic Compute Cloud) de computación en la

nube proporcionado por Amazon. Xen ha desarrollado una API para el desarrollo que entre otras cosas facilita el desarrollo de *virtualización* en la nube.

3.1.3. VMWare

VMWare es una empresa norteamericana encargada de proporcionar software orientado a la *virtualización*, para ello ofrece software sobre la nube y distintos tipos de *hypervisor* en función del resultado que se quiera proporcionar.

Se van a destacar las funciones de *virtualización* de VMWare mediante *máquinas virtuales*. Como hemos comentado anteriormente, gracias a una *máquina virtual*, por medio de una capa de software, virtualizar elementos hardware y, por ejemplo, usar un sistema operativo distinto al instalado en la máquina física.

Algunas de las características que nos proporciona VMWare son:

- Simultaneidad, poder ejecutar varias *máquinas virtuales* al mismo tiempo en una misma máquina física.
- Independencia entre *máquinas virtuales*, de esta manera si una de ellas falla no afecta al resto.
- Simular un completo entorno tecnológico con sus características hardware y software mediante la ejecución de un programa.

Estos conceptos anteriores pueden ser aplicados a la creación de entornos más eficientes y flexibles. Un buen ejemplo es la *virtualización* de servidores en una máquina física, de esta manera se puede ahorrar en costes al necesitar menos máquinas físicas y, al mismo tiempo aumentar la eficiencia de las máquinas de las que disponemos y su utilización.

Algunas de las otras soluciones de VMWare orientadas a la nube son, VMWare vSphere el cual permite la *virtualización* y acceso a nube de carácter privado con el objetivo de proporcionar a las empresas una herramienta que permita, entre otras cosas, la *virtualización* de centros de datos y su integración en la nube. Otro de los productos es vCloud Air, el cual permite la ejecución de terceras aplicaciones desde la nube y, abre un nuevo campo en el desarrollo de aplicaciones enfocadas a ser usadas desde un entorno nube. [13]

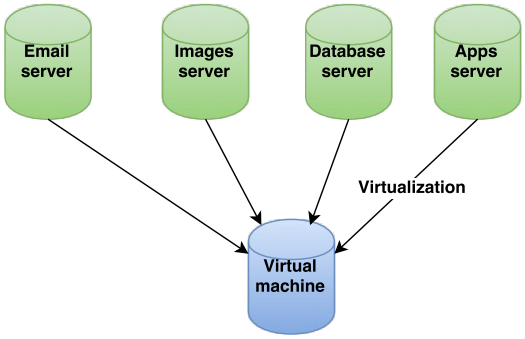

Figura 3.4: Ejemplo de virtualización de servidores

3.1.4. Virtual Box

VirtualBox es un *hypervisor* y actualmente es desarrollado por Oracle, de igual forma que otras *máquinas virtuales* permite correr otros sistemas operativos como Windows, Linux, OS X, Solaris y OpenSolaris sobre el SO que ejecute sobre la máquina física. De forma similar a como permiten otras *máquinas virtuales*, permite correr múltiples sistemas operativos de forma simultánea e independientes entre sí. Recomendado para usuarios de PC de sobremesa, aunque también puede usarse en servidores. [14]

Virtual Box permite múltiples configuraciones del software y hardware de la máquina física como almacenamiento disponible, memoria RAM, audio y otros periféricos.

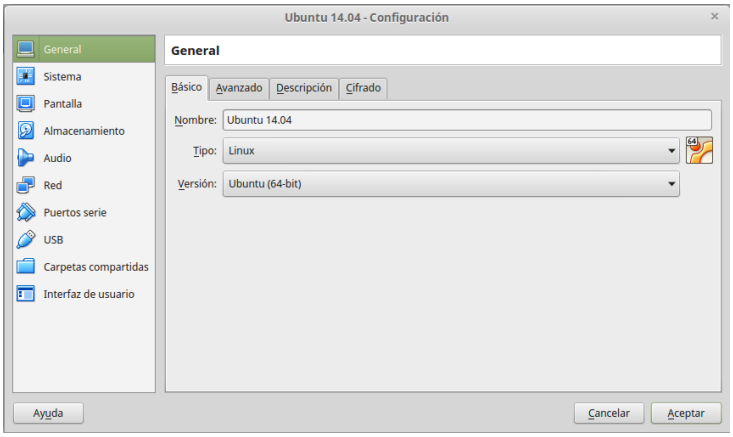

Figura 3.5: Configuración de Virtual Box

También permite modificaciones sobre la configuración de red, como acceso NAT mediante un bridge privado o mediante un bridge público o una modalidad mixta entre un bridge público y uno virtual llamado adaptador solo-anfitrión.

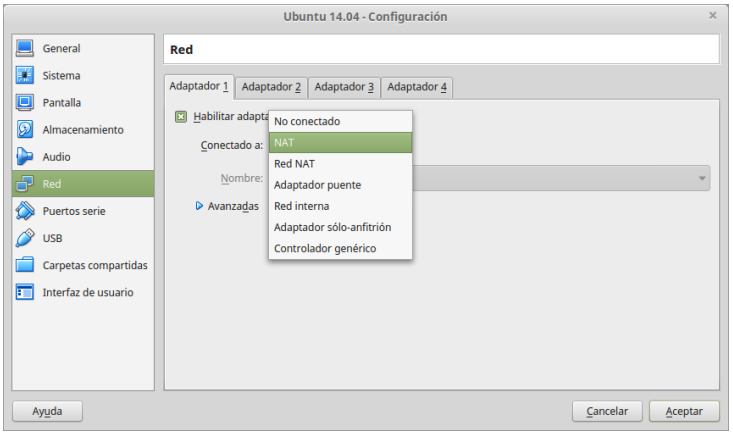

Figura 3.6: Configuración de red en Virtual Box

3.2. Plataformas en la nube

En primer lugar, se explica de lo que trata la conocida como computación en la nube, se trata de un moderno enfoque por el cual los usuarios pueden acceder a diversos servicios como almacenamiento, bases de datos y otros servicios a través de internet.

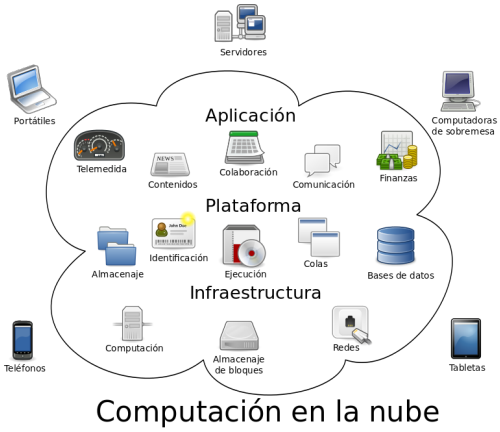

Figura 3.7: Servicios en la nube (fuente `https://commons.wikimedia.org/wiki/File:Cloud_computing-es.svg`)

De esta forma se mejora la flexibilidad, escalabilidad y seguridad para el acceso a estas aplicaciones y a la vez se ahorra en costes y se permite el uso de los servicios desde cualquier punto del planeta. Típicamente los proveedores de estos servicios tienen distribuidos distintos servidores de forma geográfica.

Los tipos de computación en la nube pueden clasificarse en tres distintos:

1. **Plataforma como servicio.** Permite a los usuarios el desarrollo, ejecución y control de aplicaciones sin tener que preocuparse de la infraestructura sobre la que operar. Se puede diferenciar entre públicas y privadas.

2. **Software como servicio,** se trata de programas accesible a través de Internet mediante el navegador. Es usado en programas de todo tipo como desarrollo de software, *virtualización*, antivirus, contabilidad, etc.

3. **Infraestructura como servicio,** consiste en toda la infraestructura por la

cual se soportan los distintos servicios en la nube como almacenamiento y configuraciones de red.

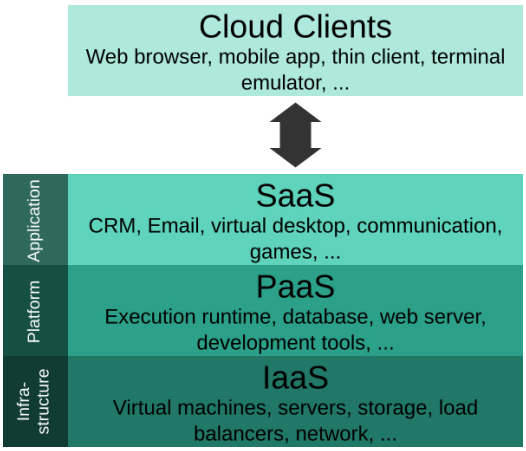

Figura 3.8: Modelos de nube (fuente `https://commons.wikimedia.org/wiki/File:Cloud_computing_layers.svg`)

Respecto a las plataformas en la nube existen soluciones de código abierto y código privativo. Las plataformas en la nube de código abierto, tienen una comunidad más amplia y unas actualizaciones más rápidas. Las de código privativo, aseguran un producto más robusto y fiable. Algunas de las plataformas en la nube de código abierto más conocidas son *OpenStack* y *CloudStack*.

3.2.1. OpenStack

OpenStack se creó en junio de 2010, cuando la NASA y la empresa Rackspace deciden iniciar un proyecto orientado a la nube de código abierto. La NASA liberó software de tipo Infraestructura como servicio y, por su parte Rackspace liberó software también de tipo Infraestructura como servicio enfocado al almacenamiento.

Tiene el objetivo de crear una plataforma de computación en la nube, de código abierto y que sirva tanto para desplegar nubes públicas como privadas. Trata de ser sencilla de implementar, fácilmente escalable y con múltiples prestaciones.

Tiene un gran respaldo de empresas del sector como Dell, IBM, Red Hat, VMWare, KVM, Cisco, HP, y así hasta unas 200 empresas involucradas en el

proyecto. Cada seis meses se libera una versión nueva de *OpenStack* y, también cada 6 meses hay reuniones de diseño de cara al desarrollo de la siguiente versión.

OpenStack está compuesto de forma modular. De esta manera cada componente cumple una función. Algunas de las funciones son almacenamiento, computación o gestión de redes. A continuación, se muestran algunos de los componentes y sus principales funciones.

- **Nova.** Controlador de estructura de computación en la nube de forma masivamente escalable.
- **Cinder.** Almacenamiento a nivel de bloque, el cuál es virtualizado, accesible a través de Nova de forma transparente al usuario.
- **Swift.** Sistema de almacenamiento escalable distribuido que puede ser usado por organizaciones para almacena datos de forma eficiente, segura y barata.
- **Neutron.** Permite la gestión de redes que permite la comunicación entre distintos componentes.
- **Horizon.** Proporciona interfaz gráfica en forma de web para poder controlar de forma más sencilla e intuitiva los distintos módulos.

El uso de *OpenStack* permite tener una arquitectura flexible y, de esta forma adaptarla a necesidades específicas. *OpenStack* también proporciona un conjunto de APIs, lo que agiliza el desarrollo y además permite ahorrar costes ya que es código libre. [15]

Este tipo de arquitectura modular trabaja de forma conjunta para poder crear una infraestructura que se acomode requisitos específicos. En la siguiente imagen 3.9, se puede apreciar como los distintos módulos interactúan entre sí.

Hay que destacar que el *hypervisor* más usado por *OpenStack* para realizar la *virtualización* es KVM usado por un 62 % para soluciones desarrolladas por *OpenStack* (datos de noviembre 2013). Otras opciones usadas para la *virtualización* son Xen y otras tecnologías relacionadas como vSphere de VMWare. [**?**]

En relación al anterior párrafo, el uso de instancias en *OpenStack* es básicamente una *máquina virtual*. Una instancia se trata de una imagen de un sistema

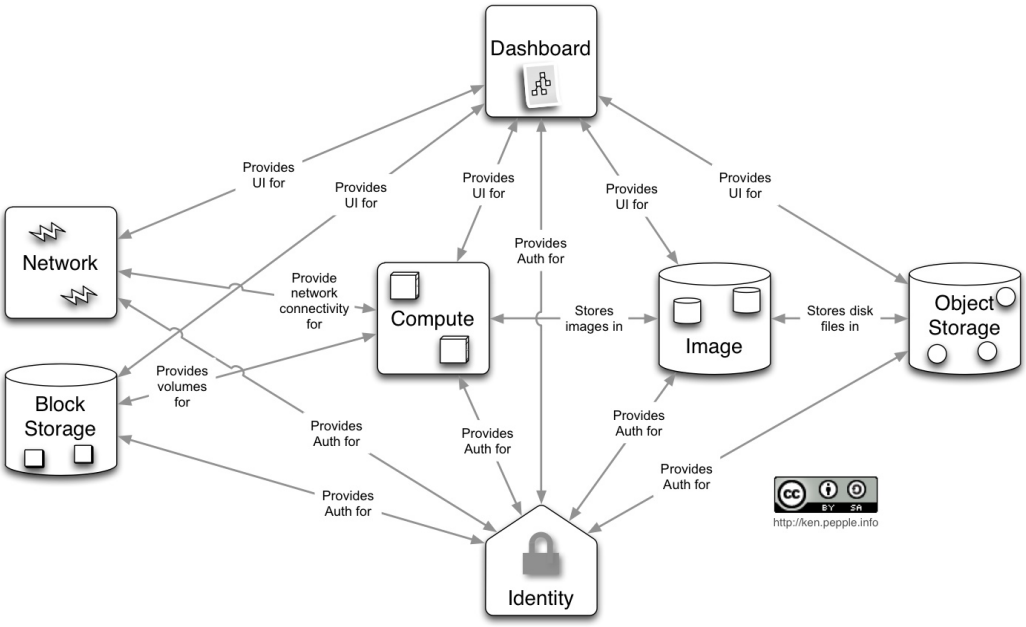

Figura 3.9: Módulos de OpenStack (fuente `https://commons.wikimedia.org/wiki/File:Openstack-conceptual-arch-folsom.jpg`)

operativo el cual puede estar configurado en base a intereses específicos. Existen distintos tipos de instancias que varían según características hardware como procesador, memoria RAM o almacenamiento. Una instancia puede tener dos tipos de IP:

- **IP privada.** La cual se usa para poder establecer comunicaciones entre instancias, emula un comportamiento similar a un NAT.
- **IP flotante.** Se trata de una IP pública de manera que permite la comunicación desde fuera.

3.2.2. CloudStack

Se trata de un proyecto de código abierto que trata de desplegar Infraestructura como servicio(IaaS) en la nube tanto pública como privada. Fue inicialmente desarrollado por Cloud.com y Citrix. De forma similar a como hace *OpenStack*, *CloudStack* usa *hypervisores* como KVM, XEN o VMWare vSphere. [16]

La finalidad de *CloudStack* es la de proveer una plataforma en la nube que sea flexible y abierta y además sea segura y escalable.

Algunas de las características principales de *CloudStack* son:

- **Interfaz de usuario fácil de manejar.** Está basada en una interfaz web para controlar la plataforma en la nube de manera sencilla desde cualquier dispositivo.
- **Escalabilidad.** Uno de los principales alicientes de la computación en la nube, es la sencillez para ampliar la infraestructura.
- **Código abierto y minuciosa documentación.** Al ser código abierto tiene una gran comunidad de personas detrás que adapta el código en función de sus necesidades. Además, se dispone de abundante documentación.
- **Soporte de múltiples hypervisor.** *CloudStack* soporta múltiples *hypervisor* como KVM, Xen, Hyper-V, Xen, etc.
- **API REST.** Dispone de una API REST propia para la comunicación entre cliente y servidor mediante HTTP.

3.3. Middleware de comunicaciones

3.3.1. Ice

Ice (*Internet Comunication Engine*) es un framework de código abierto para sistemas distribuidos basados en un protocolo *RPC* (*Llamada a procedimiento remoto*) creado y desarrollado por la empresa ZeroC. Ice soporta múltiples lenguajes como Java, C++, C#, PHP, Python, Ruby, y algunos más, también puede ser ejecutado en varios sistemas operativos como Linux, Windows, OS X, Android, Solaris, iOS.

Un framework se trata de una estructura y usa herramientas como bibliotecas mediante las cuales se puede facilitar el desarrollo de un software.

Por otra parte, el objetivo de los protocolos *RPC*, típicamente usados en comunicaciones del tipo cliente-servidor, es el de ejecutar funciones de forma remota en otros ordenadores y devolver un resultado. Estos protocolos pueden hacer uso de lenguajes llamados *IDL* (*Lenguaje de Descripción de Interfaz*).

El funcionamiento de *Ice* es el siguiente: El cliente llama al *stub*. *Stub* son librerías instaladas tanto en el cliente como en el servidor, que permite convertir los parámetros usados en una función y también los resultados devueltos por el cliente. A convertir los parámetros de las funciones en el lado del cliente se le llama marshalling, y a reconvertir los parámetros y los resultados devueltos por el servidor unmarshalling.

Algunas de las características de *Ice* son:

- **Seguridad.** La seguridad es un elemento clave en las comunicaciones para poder transmitir información de forma fiable. *Ice* proporciona seguridad en los mensajes mediante protocolos de encriptación de mensajes y autenticar conexiones como SSL/TLS proporcionado por IceSSL el cuál hace uso de herramientas como certificados digitales, códigos de autenticación, y uso de llaves para encriptación.
- **Rapidez.** Uno de los principales objetivos en la creación de *Ice* fue el de intentar crear un software ligero de forma que fuera fácilmente escalable. Para ello, hace un uso muy eficiente de los recursos y demanda poca memoria y uso de CPU. También hace uso de protocolos de compresión muy

útiles cuando el ancho de banda es limitado

- **Confiable.** *Ice* es tolerante a fallos, permite que si una máquina falla las otras puedan seguir funcionando como si nada hubiera ocurrido. Además, permite hacer balanceo de carga en diferentes servidores favoreciendo factores tan importantes como la escalabilidad.

- **Flexible.** Permite hacer invocaciones de forma síncrona y también asíncrona mediante el uso de protocolos como TCP, UDP, SSL/TLS. Además, permite el uso de una conexión de tipo bidireccional de esta forma permite que el cliente y el servidor puedan comunicarse en las dos direcciones sobre una misma conexión. También hay que destacar la flexibilidad en el uso de distintos lenguajes gracias al compilador *Slice*.

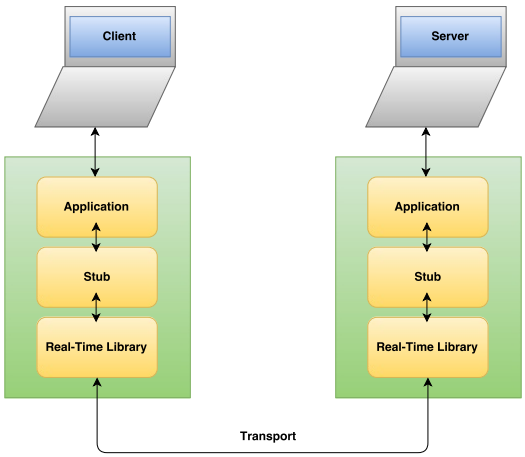

Figura 3.10: Esquema funcionamiento de RPC

Otros servicios de *Ice*:

- **IceGrid.** Este servicio posibilita el hecho de actuar de forma parecida a un DNS de forma que se le pone un nombre a un servidor *Ice* y es el propio IceGrid el encargado de devolver parámetros como dirección IP y puerto asociado. Entre otras funciones permite activar un servidor cuando se quiere hacer uso de él y no está funcionando, replicar un servidor y hacer balanceo de carga, y proporciona de forma fácil administración.

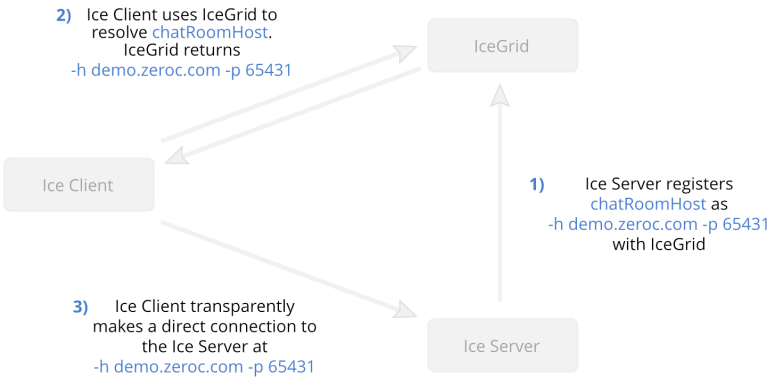

Figura 3.11: IceGrid (fuente `https://zeroc.com/products/Ice`)

- **Glacier2.** Permite establecer comunicaciones de manera segura tanto en redes privadas como públicas a través de un firewall, para ello solo se necesita abrir un puerto en el firewall y de esta forma poder crear comunicaciones entre múltiples servidores y clientes. Otra de las funcionalidades que aporta es la de poder establecer comunicaciones bidireccionales y así los servidores pueden hacer llamadas al cliente sobre una misma conexión.

Figura 3.12: Glacier2 (fuente `https://zeroc.com/products/Ice`)

- **IceStorm.** Se trata de un sistema middleware de tipo publicación-suscripción mediante el uso de eventos. El funcionamiento consiste en la suscripción por parte de un cliente a un *topic*. El *topic* se crea dinámicamente en el servidor y se distingue mediante un identificador único.

Figura 3.13: IceStorm (fuente `https://zeroc.com/products/Ice`)

- **Freeze.** Permite almacenar de forma persistente objetos en las bases de datos de Berkeley y hacer uso de la base datos con funciones como indexar. La base datos de Berkeley y Freeze usados de manera unificada permite que no haga falta instalación, ni administración.

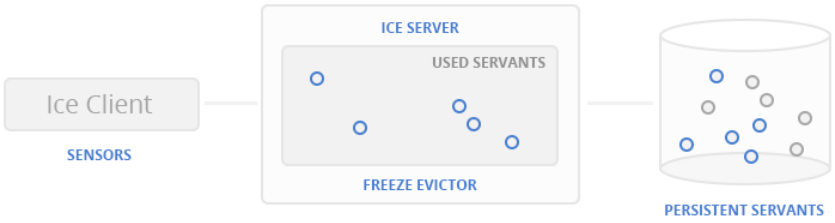

Figura 3.14: Freeze (fuente `https://zeroc.com/products/Ice`)

3.3.2. Ice Storm

Ice Storm dota a *Ice* de un paradigma de comunicación unidireccional del tipo publicación-suscripción. Este patrón permite que los mensajes se clasifiquen por *topics*, existen nodos a los que se les permite suscribirse a un *topic*. De manera que, cuando un publicador manda mensajes de un *topic* los difunde a todos los nodos suscriptores de ese *topic*.

Los *topics* son creados de forma dinámica y se identifican mediante nombres únicos. Un *topic* es similar a las operaciones y tipos de mensajes definidos por una interfaz de Slice. El publicador posee un módulo para mandar mensajes a través de un proxy y el suscriptor los recibe. Tanto el suscriptor como el publicador deben asegurarse de que están usando interfaces compatibles.

Ice Storm permite que la difusión de mensajes sea más fácil para el publicador de la información, ya que de esta forma es el módulo de *Ice Storm* el encargado de manejar los suscriptores, la recuperación ante fallos y el envío. De esta forma, a través del módulo de *Ice Storm* se consigue que el publicador haga una única llamada a este módulo para difundir mensajes y es el encargado de la retransmisión del mensaje. También lleva a cabo las tareas del control de los suscriptores tanto de apuntarse, como de borrarse del *topic* suscrito.

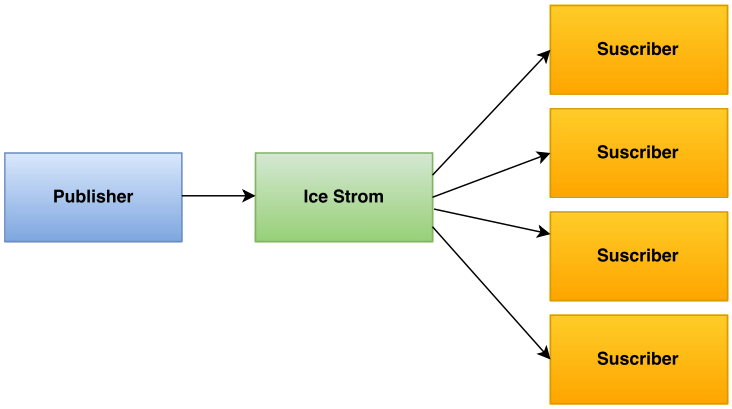

Figura 3.15: Ice Storm

Además, *Ice Storm* permite que cada suscriptor establezca sus propios parámetros de calidad de servicio.

3.3.3. DDS

DDS (*Data Distribution Service*) es un protocolo middleware y API que nace de la mano de OMG (*Object Management Group*) con el objetivo de crear un estándar del tipo publicador-suscriptor y datos centralizados.

Se basa en el mismo paradigma que *Ice Storm* de publicación-suscripción, de esta forma se simplifica las comunicaciones entre un servidor y varios clientes a nivel de programación. Mediante el uso de un módulo de *Ice Storm* se encarga de la difusión de mensajes y llevar a cabo diversas tareas de control de la comunicación. Este módulo también se encarga del registro de suscriptores a los diversos *topics* que pueden existir. [3]

Algunas de las características de *DDS* son:

- **Datos centralizados**. *DDS* se define como un middleware de datos centralizados ya que, de esta forma, se incluye toda la información contextual que necesita una aplicación cuando recibe datos. Favorece los sistemas y aplicaciones que se comunican mediante el uso de middleware. Gracias a que *DDS* es un sistema de datos centralizado, se permite especificar qué datos almacena y diversas propiedades qué determinan cómo y cuándo compartir esos valores de forma fácil y segura.
- **Acceso global.** *DDS* permite a las aplicaciones acceso desde cualquier sitio, simulando el almacenamiento local. *DDS* se encarga de almacenar los datos solo por el tiempo necesario y permite comunicaciones a través de sistemas móviles, empotrados, en la nube, con independencia del lenguaje o sistema y con una baja latencia.
- **Calidad de Servicio.** Algunos de los parámetros de calidad de servicio son confiabilidad, tiempo de vida del sistema y seguridad. Además, *DDS* se adapta múltiples situaciones, como, por ejemplo, de forma dinámica conoce dónde enviar los datos, aunque haya cambios en el sistema e informa a los componentes de la comunicación de los cambios. Cuando el tamaño de los datos es muy grande, *DDS* se encarga de enviar solo la información necesaria a los componentes imprescindibles. En materia de seguridad, *DDS* lleva a cabo control de acceso mediante certificados de autenticación(CA), *DDS* Domains y encriptación mediante AES128 y AES256.
- **Descubrimiento dinámico.** *DDS* permite que los publicadores y suscriptores puedan ser descubiertos mediante *DDS* de forma dinámica en cualquier

momento, ya sea compilación o ejecución. Además, *DDS* también descubre si este punto de comunicación está publicando datos o suscrito a un *topic* o ambas. También este descubrimiento dinámico permite conocer las características de comunicación de los distintos suscriptores y publicadores. Todo ello con independencia de si sucede en una misma máquina o en la red.

- **Escalabilidad.** *DDS* está diseñado para ser escalable a lo largo de distintas plataformas desde una pequeña red local a enormes sistemas sobre la nube. De esta forma *DDS* potencia el Internet de las cosas, ya que permite una gran cantidad de participantes (incluso millones), comunicándose a gran velocidad y con una gran seguridad y fiabilidad.

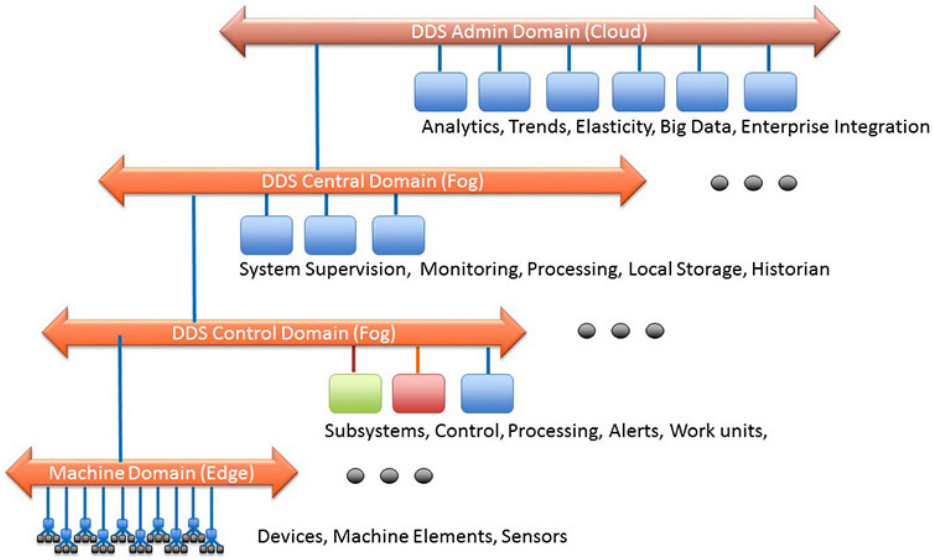

Figura 3.16: Escalabilidad de DDS (fuente `http://portals.omg.org/dds/what-is-dds-3/`)

Además, hay que destacar el uso de RTPS (*Real-Time Publish-Suscribe Protocol*), se usa sobre una capa de transporte no confiable como TCP o UDP y, cuyo objetivo es proporcionar una estructura de DCPS (*Data-centric publish-subscribe*). Gracias a este protocolo, se puede modificar la aplicación para que: se adapte a un modelo de calidad de servicio propuesto para el envío y recepción de datos, modelado de datos, como acceder a los datos desde la aplicación, etc. [4]

Lo que persigue el uso de este protocolo es:

- Asegurar que las aplicaciones de tipo publicador-suscriptor tengan un buen

rendimiento y sean confiables, y que se haga uso de calidad de servicio sobre redes IP.

- Fácil escalabilidad en la red y, que se permita una gran escalabilidad.
- Facilitar que los dispositivos puedan ser de tipo enchufar y usar, que las aplicaciones y servicios se descubran automáticamente y se puedan unir a la plataforma de forma sencilla.
- Fácilmente configurable para fijar parámetros de fiabilidad y temporales.
- Hace uso de mensajes de estado para la comunicación entre publicador y suscriptor.

RTPS opera dentro de un Dominio de Participantes en el cual hay publicadores y suscriptores, dentro de este Dominio de Participantes, el cual contiene Puntos Finales de Comunicaciones (escritores y lectores), y mediante este se puede enviar y recibir información mediante RTPS. En este caso, los escritores envían datos y los lectores acceden a ellos. Un publicador en un Dominio de Participantes mandará datos que sean solicitados por los suscriptores, algunos atributos de la publicación son el *topic*, el tipo de publicación y la calidad que esta debe tener en función de los parámetros de QoS. De forma similar ocurre con los suscriptores, que tiene características muy similares, solo que en este caso son los encargados de recibir las publicaciones.

3.4. El planificador de tareas de Linux

La función principal del planificador de tareas es la de maximizar el uso del procesador, es decir, es el encargado de decidir qué proceso se ejecuta en el procesador y cuál tiene que esperar. Se trata de intercalar los procesos de manera que parezca que múltiples procesos se ejecutan de manera concurrente y simultánea.

El planificador se encarga de tareas como: cuánto tiene que esperar un proceso para ser ejecutado, cuándo tiene que ser ejecutado y durante cuánto tiempo. El realizar de la mejor manera estas funciones va a determinar en gran medida la calidad del planificador. [5]

Pueden existir dos casos en los que actúa un planificador de tareas:

- En el caso de que nos encontremos con un sistema operativo que permite multitarea pero el procesador cuenta con un único núcleo. En este caso, el planificador tratará de intercalar procesos haciendo creer que estos se ejecutan de manera simultánea.
- En el caso de encontrarnos con un sistema operativo que permite multitarea y procesador cuenta con múltiples núcleos. En este escenario, se podrán ejecutar múltiples tareas de manera simultánea.

Existen dos tipos de realizar multitarea de forma cooperativa o de manera preferente:

- En la forma cooperativa es el propio proceso el que decide cuando renunciar al procesador y el sistema operativo no puede forzar esta situación. Esto no es ideal ya que algunos programas podrían funcionar de manera incorrecta provocando fallos en el sistema.
- De forma preferente es el propio procesador el que elige qué proceso debe ser ejecutado en cada momento y que porción de tiempo debe usar cada proceso. Este es el método usado por Linux.

En ambos casos el procesador al expulsar el proceso guarda el contexto del proceso que se estaba ejecutando, para que cuando se quiera reanudar ese proceso se cargue su contexto, y de esta manera continuar ejecutando el programa.

3.4.1. Sistemas de tiempo real

El soporte de Sistemas de Tiempo Real de Linux se incorporó con el estándar POSIX, que describen múltiples políticas para planificar tareas. Desde entonces, Linux ha sufrido múltiples mejoras que le han permitido mejorar sus requisitos temporales y, de esta manera comportarse mejor para sistemas de tiempo real y sistemas embebidos.

Uno de los medios por los cuales Linux permite modificar el planificador es mediante la asignación de prioridad estática. Esta prioridad estática comprende valores entre 1 y 99, se ejecuta el proceso con un número mayor, a los procesos sin características de tiempo real se les asigna un 0. Si hay un proceso ejecutando con prioridad estática 35 y llega un con prioridad estática 34, el proceso con prioridad

estática 35 seguirá ejecutando hasta que termine o se bloquee. Por otro lado, si hay un proceso ejecutando con prioridad estática 35 y llega un con prioridad estática 36, el procesador parará el proceso con prioridad estática 35 y se ejecutará el de 36.

Algunos otros tipos de políticas de planificación de tareas son:

- FIFO (*First in, First Out*), mediante esta política el proceso con mayor prioridad se ejecutará hasta que no llegue otra con mayor prioridad, termine o renuncie.
- Round robin, mediante esta política se asigna una porción de tiempo a cada proceso y cuando termine se pone el último en la lista de procesos. Hay que tener en cuenta que solo resulta útil en procesos con la misma prioridad.
- Política normal, en este caso se trata de procesos sin necesidad de ejecución en tiempo real por lo que la prioridad estática es de 0 y los procesos con políticas FIFO o Round Robin siempre tendrán prioridad sobre este.

3.5. Marco regulador

Para el desarrollo del Trabajo de Fin de Grado se propuso que el gasto en software debía de ser mínimo y, en caso de que fuera posible, software de código abierto. La mayoría de herramientas software usadas han sido de código abierto, con excepción de *DDS Connext* de RTI.

Aunque *DDS Connext* sea software de código propietario, dispone de licencias gratuitas para fomentar la investigación de sistemas distribuidos en entornos académicos como universidades. Además, para conseguir la licencia para universidades se debían cumplir una serie de condiciones como:

- Que la organización a la que se presta la licencia no tenga un estatus lucrativo.
- El uso de la licencia tendrá fines de investigación o educativos.
- El proyecto debe tener una mínima o inexistente financiación comercial.

El proyecto desarrollado cumplía las anteriores condiciones, por lo que se nos facilitó la licencia para universidades por parte de RTI. Por otro lado, señalar que se ha respetado el *Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia*. Por ello, en el apartado de referencias se señalan las distintas fuentes consultadas para el desarrollo de este Trabajo de Fin de Grado. [2]

3.6. Contexto de investigación en el que se enmarca el trabajo

El presente Trabajo Fin de Grado se enmarca en el contexto de las líneas de investigación de sistemas distribuidos de tiempo real y ciber-físicos, concretamente se basa en los retos, problemas y soluciones identificados en [21] para tecnologías de virtualización para entornos de computación predecible en la nube y en [24] para tecnologías middleware para sistemas ciber-físicos. Concretamente el middleware Ice ha sido mejorado en varias arquitecturas propuestas por el grupo de

investigación como [35] en el que se ajustan diferentes parámetros para mejorar el rendimiento; [36] en el que se realiza una propuesta de arquitectura centralizada para sistemas distribuidos de tiempo real que puedan soportar un número elevado y variable de clientes; las propuestas para verificación on-line de sistemas ciber-físicos conectados como [38], [50] y [39]; middleware con lógica aumentada para soportar cambios de estructura dinámica sistemas distribuidos de tiempo real basados en servicios [22], [41], [40], [42].

Este trabajo se encuadra dentro de la gestión eficiente de recursos de la plataforma de ejecución tanto en los nodos participantes como en la comunicación e interacción entre los nodos a través de la mejora de la tecnología middleware. De igual forma se encuadran dentro de la gestión dinámica que requiere lógica aumentada [51].

A nivel de gestión de recursos dentro de los nodos participantes, se consigue un mayor aprovechamiento de los recursos computacionales (CPU, memoria, energía, etc.) utilizando gestores de recursos con capacidades de priorización y de planificación temporal como [23], [33], [34], [31], [32], [25], [26] o [48]. Muchas de las técnicas de gestión y diseño de estas plataformas están descritas en [27], [30], [28], y [29].

En el lado de las comunicaciones entre nodos basados en middleware, para un sistema con requisitos de tiempo es necesario analizar la confiabilidad de la plataforma, su rendimiento y estabilidad temporal. En este sentido, diversos trabajos preceden el contexto de este proyecto como se enumera a continuación. En [43] se orquestó una aplicación de monitorización del tráfico ferroviario y su uso en laboratorios. En [46] se diseñó una aplicación de video vigilancia activada por sensores y con soporte para cambios de estructura en tiempo real. En [47] se diseñó e implementó un sistema para la composición de servicios basado en tecnología Java, con Jini. En [49] se realiza un estudio del middleware DDS para virtualización sobre VirtualBox. En [44] se presenta un puente para adaptación de middleware a diferentes paradigmas de comunicación y en [45] se presenta la interconexión aumentada entre iLand y el anexo distribuido de Ada (Ada DSA).

El presente Trabajo de Fin de Grado entronca en mayor o menor medida con todos estos trabajos. En primer lugar, se relaciona con los trabajos de estudio de rendimiento de middleware para entornos distribuidos con requisitos de tiempo. En segundo lugar, se relaciona fuertemente también con el nivel del sistema operativo, componente que es fundamental conocer con cierto detalle para poder añadir las mejoras pertinentes al middleware. En esta segunda parte se relaciona con los trabajos de gestión de recursos: primeramente, para analizar el consumo de CPU

y en segundo lugar con un entorno de red cargado para obtener los límites de carga en las máquinas a partir de los cuales la utilización del middleware presenta anomalías. En último lugar se relaciona con la gestión dinámica de la estructura y soporte a la reconfiguración segura en sistemas distribuidos. Los entornos de gestión de la calidad del software [53], [52] han sido también tocados de forma superficial.

Capítulo 4

Análisis de las prestaciones de ICE

4.1. Entorno hardware y software del experimento

Las distintas pruebas fueron efectuadas entre una o dos (dependiendo del escenario) máquinas con el hardware y software que se describe a continuación. Los distintos escenarios fueron:

- Misma máquina física.
- Distintas máquinas físicas.
- Misma máquina virtual.
- Dos máquinas virtuales misma máquina física.
- Distinta máquina virtual en máquina distinta física.

Además, hay que señalar que en todos los escenarios las pruebas se hicieron de dos maneras. En una de ellas únicamente se ejecutaba el cliente o servidor y en otro caso se ejecutaba un pequeño programa. Con este programa se generaba una carga sintética mediante la cual, el porcentaje de uso de la CPU alcanzaba valores cercanos al 100 %.

Escenario hardware y software de la máquina virtual:

- **Hardware:** QEMU Virtual CPU Versión 2.0.0 de 2 núcleos y 1024 MB de RAM

- **Software:** Versión de KVM asociada al kernel 3.19.0-49-generic #55 14.04.1-Ubuntu SMP
  Ubuntu 12.04.5 LTS de 32 bits
  Librerías de Ice (Internet Communication Engine) versión 3.4 y librerías asociadas a ICE para soportar distintos lenguajes, en nuestro caso C++.
  Versión 4.6.3 del compilador g++.
  Distintos módulos de Ice como IceStorm, IceBox, IceFreeze en su versión 3.4 todos ellos.

Escenario hardware y software de la máquina física o host:

- **Hardware:** Intel Celeron CPU E3400 de doble núcleo a 2.6 Ghz con 1971 MB de RAM

- **Software:** Ubuntu 14.04.3 LTS de 32 bits
  Librerías de Ice (Internet Communication Engine) versión 3.5 y librerías asociadas a ICE para soportar distintos lenguajes, en nuestro caso C++.
  Versión 4.8.4 del compilador g++.
  Distintos módulos de Ice como IceStorm, IceBox, IceFreeze en su versión 3.5 todos ellos.

4.1.1. Descripción del experimento

El experimento que se va a llevar a cabo fue, el de enviar distintos bloques de bytes en distintas formas de información como un número, caracteres y cadenas de texto de distintos tamaños (1 carácter, 6 caracteres, 1024 caracteres, 2048 caracteres y 4096 caracteres). A continuación, se compararon los tiempos obtenidos en los distintos escenarios con y sin uso de carga sintética. El tamaño de los datos en nuestra máquina un char ocupa 1 byte y un número de tipo int 4 bytes.

Para ello se usó un esquema de tipo cliente-servidor y el cliente se conectó 100 veces al servidor y realizó 100 peticiones, de esta manera podemos asegurar que los resultados obtenidos son más cercanos a la realidad. El tiempo se medió desde que el cliente hace la petición hasta que le llega la contestación del servidor al cliente. En este caso, la contestación del servidor será un mensaje sin contenido.

El código usado en el cliente es el siguiente:

```
try{
  ic=Ice::initialize(argc, argv);

  Ice::ObjectPrx base= ic->
     stringToProxy("StringService:default -h localhost -p
     10000");
  StringServicePrx remoteService=
     StringServicePrx::checkedCast(base);

  Ice::ObjectPrx base1= ic->
     stringToProxy("ArithmeticService:default -h localhost
     -p 10001");
  ArithmeticServicePrx remoteService1=
     ArithmeticServicePrx::checkedCast(base1);

  if(!remoteService)
    throw "Invalid proxy";

  if(!remoteService1)
    throw "Invalid proxy";

  for(int i=0;i<100;i++){
  auto begin = std::chrono::high_resolution_clock::now();
  remoteService1->addIntegers(2);
  auto end = std::chrono::high_resolution_clock::now();
  //remoteService->stringSize("a");
  //remoteService->stringSize(cadena1024);
  //remoteService->stringSize(cadena4096);
  //remoteService->stringSize(cadena2048);
  //remoteService->stringSize("abcdef");
  }
```

En primer lugar, se inicializa la ejecución Ice. A continuación, se crean dos proxis en los cuales se indican parámetros como el servicio el cuál queremos solicitar al servidor, la dirección IP del servidor y el puerto a utilizar, por defecto, para el protocolo TCP/IP.

El bucle de tipo *for* nos sirvió para hacer 100 peticiones al mismo servidor, de esta manera se fueron ejecutando las diversas pruebas para la realización del experimento.

El código usado en el servidor es el siguiente:

```
try{
     ic=Ice::initialize(argc,argv);
     Ice::ObjectAdapterPtr adapter1=
        ic->createObjectAdapterWithEndpoints("asii_adapter1",
        " default -p 10000");
     Ice::ObjectPtr object1= new StringServiceI;
     adapter1->add(object1,
        ic->stringToIdentity("StringService"));
     adapter1->activate();

     Ice::ObjectAdapterPtr adapter2=
        ic->createObjectAdapterWithEndpoints("asii_adapter2",
        " default -p 10001");
     Ice::ObjectPtr object2= new ArithmeticServiceI;
     adapter2->add(object2,
        ic->stringToIdentity("ArithmeticService"));
     adapter2->activate();

     ic->waitForShutdown();

   } catch (const Ice::Exception& e){
     cerr << e << endl;
     status=1;
   } catch(const char* msg){
     cerr << msg << endl;
     status=1;
   }
```

De manera similar al cliente se inicializa la ejecución de Ice. A continuación, se crean dos objetos, donde escuchará el cliente, en donde se indican los parámetros como: el servicio el cuál va a prestar y el puerto donde va a escuchar para el protocolo TCP/IP.

Después, se crean dos objetos de tipo *Servant* los cuales van a ser instanciados por *StringServiceI* y *ArithmeticServiceI*. Finalmente, se añade a los objetos adaptadores *StringServiceI* y *ArithmeticServiceI* y se activan.

Una vez descrito tanto el cliente como el servidor, se va a explicar como se generó la carga sintética en el procesador. Para ello se usa el siguiente código:

```
long double c = 3.54e32;
long double b = 2.12e20;
int a = 0;
float resultado;

int main(){

   for(a=0;a<1000000000;a++){
      resultado=c/b;
      b=b+2;
   }
   return 0;
}
```

Para sobrecargar el sistema nos hemos decantado por el uso de un bucle de operaciones de coma flotante. Como sabemos las operaciones de coma flotante requieren una elevada potencia de cálculo por parte del procesador. Para sobrecargar ambos núcleos del sistema se tenía que ejecutar de forma simúltanea dos veces dicho programa. Acto seguido, se adjuntan las imágenes que muestran la carga del sistema antes 4.1 y después de la ejecución del programa 4.2.

Este búcle se ejecutó tanto en el cliente como en el servidor, en el caso de que estos se ejecutaran en máquinas físicas distintas. En el caso en el que se ejecutaban dos máquinas virtuales dentro de un mismo host físico, solo se ejecutó el bucle en el cliente, ya que la máquina física no soporta la carga de las dos máquinas virtuales. Por lo que no podemos añadir más carga sin provocar el cierre de la VM por parte de KVM.

Se puede apreciar que con este pequeño programa se fuerza a la CPU y se alcanzan picos cercanos al 100 % de rendimiento de la máquina. También hay que señalar que, como es lógico, el tiempo el cuál tarda en ejecutarse el programa para la generación de la carga es mayor que el de ejecución de nuestras pruebas cliente-servidor en Ice.

Desde el punto de vista de la red, las conexiones se realizan de forma que nuestro cliente se conectaba al servidor y aquí entra en juego Ice el cuál se encarga de autenticar la conexión, de realizar la petición del cliente al servidor y llevar la respuesta de vuelta al cliente. En la imagen 4.3 se muestra el comportamiento en la red de nuestra aplicación Ice para una única ejecución, desde el lado del cliente.

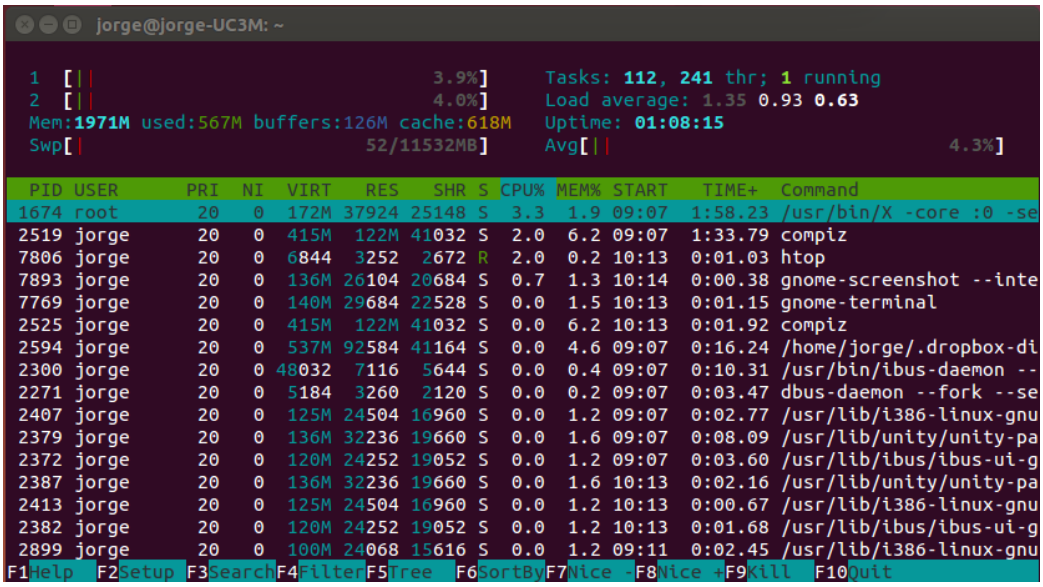

Figura 4.1: Estado sin carga sintética

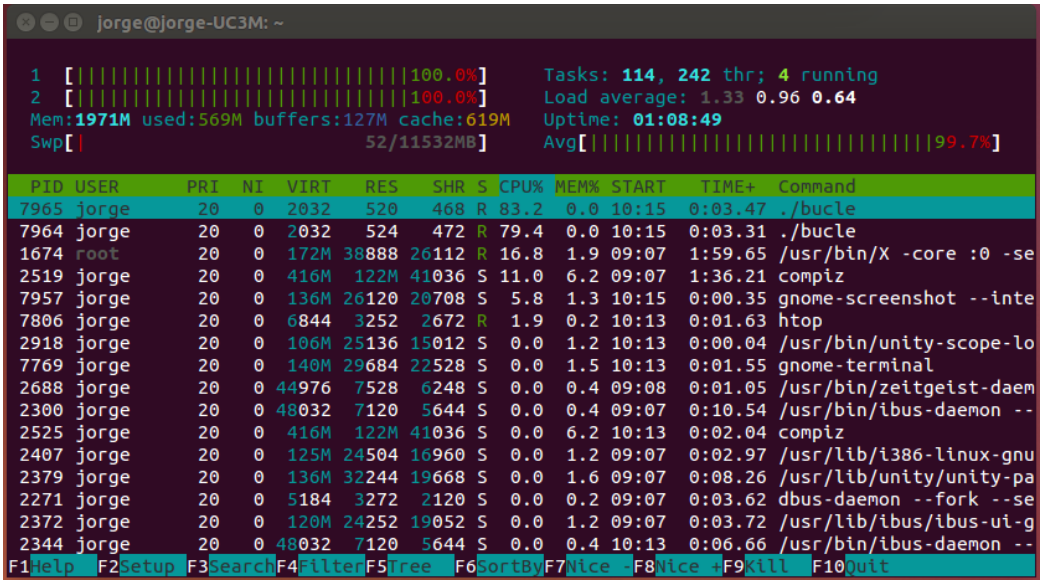

Figura 4.2: Estado con carga sintética

La figura 4.4 representa el diagrama de flujo cliente-servidor, visto desde el cliente. En primer lugar, se establece una conexión TCP entre el servidor y el cliente. A continuación, el servidor por medio de Ice valida la conexión, después el cliente hace dos peticiones al servidor y este le contesta. Finalmente se cierra la conexión.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	163.117.141.51	163.117.141.64	TCP	74	41912 > ndmp [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=10
2	0.000217000	163.117.141.64	163.117.141.51	TCP	74	ndmp > 41912 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM
3	0.000276000	163.117.141.51	163.117.141.64	TCP	66	41912 > ndmp [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=103583 TSecr=8061
4	0.000663000	163.117.141.64	163.117.141.51	ICEP	80	Validate connection
5	0.000711000	163.117.141.51	163.117.141.64	TCP	66	41912 > ndmp [ACK] Seq=1 Ack=15 Win=29312 Len=0 TSval=103583 TSecr=806
6	0.000857000	163.117.141.51	163.117.141.64	ICEP	138	Request(1): StringService.ice_isA()
7	0.001055000	163.117.141.64	163.117.141.51	TCP	66	ndmp > 41912 [ACK] Seq=15 Ack=73 Win=29056 Len=0 TSval=806125 TSecr=10
8	0.001124000	163.117.141.64	163.117.141.51	ICEP	92	Reply(1): Success
9	0.003028000	163.117.141.51	163.117.141.64	ICEP	121	Request(2): StringService.stringSize()
10	0.003282000	163.117.141.64	163.117.141.51	ICEP	95	Reply(2): Success
11	0.004594000	163.117.141.51	163.117.141.64	ICEP	80	Close connection
12	0.004853000	163.117.141.64	163.117.141.51	TCP	66	ndmp > 41912 [FIN, ACK] Seq=70 Ack=142 Win=29056 Len=0 TSval=806126 TS
13	0.005815000	163.117.141.51	163.117.141.64	TCP	66	41912 > ndmp [FIN, ACK] Seq=142 Ack=71 Win=29312 Len=0 TSval=103584 TS
14	0.006013000	163.117.141.64	163.117.141.51	TCP	66	ndmp > 41912 [ACK] Seq=71 Ack=143 Win=29056 Len=0 TSval=806126 TSecr=1

Figura 4.3: Paquetes en la red durante ejecución del programa

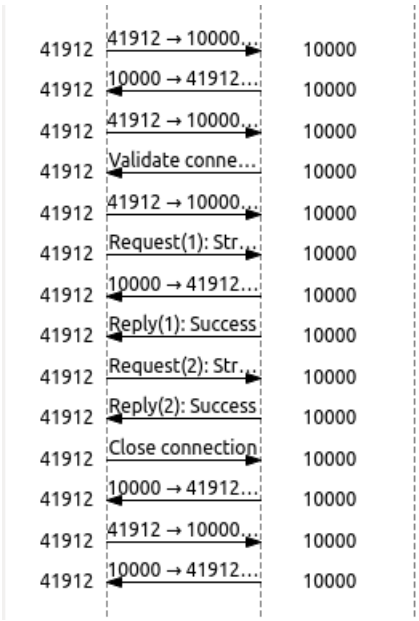

Figura 4.4: Diagrama de flujo de Wireshark

4.2. Tablas de resultados y gráficas

4.2.1. Caso de sistema sin carga sintética

En esta subseción, se aportarán los resultados obtenidos de las pruebas anteriormente comentadas. En primer lugar, se va a presentar una tabla con los tiempos obtenidos para los distintos escenarios y el caso en el cuál no se añade una carga sintética. Los tiempos obtenidos están en unidades de nanosegundos.

	Same host	Different host	Same virtual machine	Two virtual machines in the same host	Two differents virtual machines in two differents hosts
Send a char	77492,01	303641,46	232418,00	680339,90	619126,30
Send a number	75744,89	301594,68	191647,30	491871,00	628828,00
Send a chain of 6 chars	77863,90	305353,16	202483,40	551559,30	641570,30
Send a chain of 1024 chars	80079,28	594221,65	202147,70	540960,70	948085,80
Send a chain of 2048 chars	83973,69	752386,99	210523,60	509094,30	1051946,00
Send a chain of 4096 chars	85469,23	957346,24	218869,20	1707000,40	1269192,30

Tabla 4.1: Tabla de tiempos obtenidos para los distintos escenarios sin carga sintética, tiempos en nanosegundos y plataforma ICE

Gráficas obtenidas:

Se puede afirmar que, de forma general, cuanto mayor es el número de bytes enviados mayor es el tiempo que se tarda en procesar y transmitir la información.

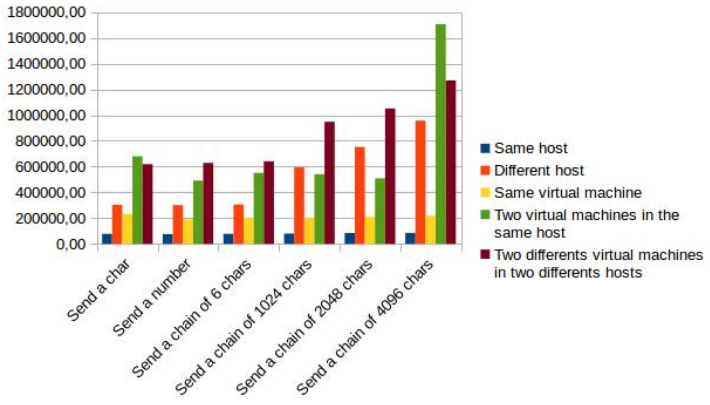

Figura 4.5: Diagrama de tiempos de los distintos escenarios, sin carga

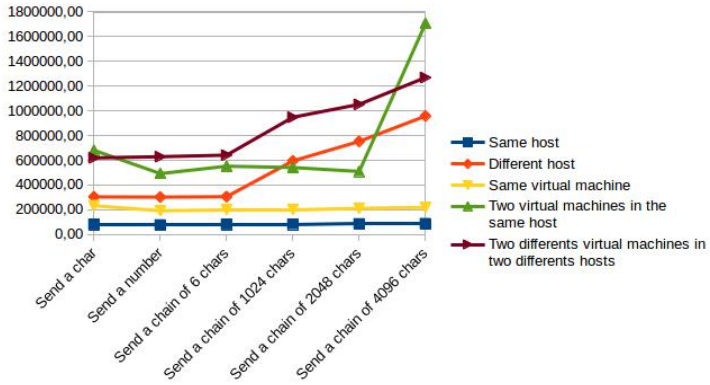

Figura 4.6: Gráfica de tiempos de los distintos escenarios, sin carga

Pero, se puede destacar los escenarios en los que la aplicación se ejecuta en la misma máquina física o en la misma máquina virtual ya que apenas hay diferencia de tiempos en función del mensaje que se envíe.

Se puede decir que el tiempo se acorta si la comunicación se realiza en la misma máquina física con respecto a si se realiza en máquinas físicas distintas. Se observa que cuando se comparan los tiempos de misma máquina física y misma máquina virtual con diferente máquina física. De forma similar ocurre en el caso de dos máquinas virtuales en la misma máquina física con respecto al escenario de dos máquinas virtuales en distintas máquinas físicas.

4.2.2. Caso de sistema con carga sintética

A continuación, se exponen los datos para el caso en el cuál la máquina se encuentra saturada.

	Same host	Different host	Same virtual machine	Two virtual machines in the same host	Two differents virtual machines in two differents hosts
Send a char	124817,40	250198,74	134209,10	492515,90	493810,90
Send a number	93981,87	266641,70	151197,20	479487,80	473504,80
Send a chain of 6 chars	103461,21	251249,69	156464,70	469764,80	510323,60
Send a chain of 1024 chars	112419,64	535313,69	138508,10	854604,10	449432,30
Send a chain of 2048 chars	94666,40	719895,42	1101671,60	774319,80	1326440,00
Send a chain of 4096 chars	108954,35	909553,37	133802,40	540983,60	2203225,10

Tabla 4.2: Tabla de tiempos obtenidos para los distintos escenarios con carga sintética, tiempos en nanosegundos y plataforma ICE

Gráficas obtenidas:

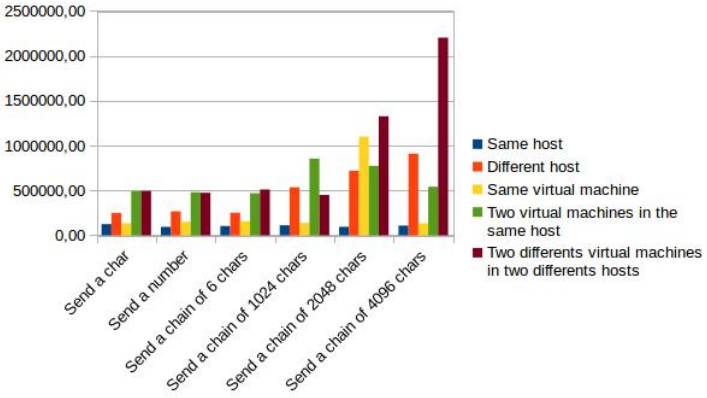

Figura 4.7: Diagrama de tiempos de los distintos escenarios, con carga

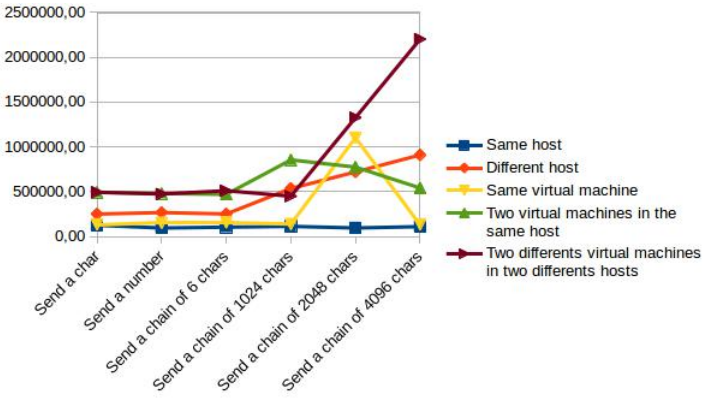

Figura 4.8: Gráfica de tiempos de los distintos escenarios, con carga

Con respecto al caso anterior, en el cuál no existía carga sintética podemos apreciar que la mayoría de afirmaciones se siguen respetando, en este caso los tiempos en los escenarios de distinta máquina física y de dos máquinas virtuales en distintas máquinas físicas acusan esta saturación del sistema.

Capítulo 5

Análisis de las prestaciones de DDS

5.1. Objetivo de los análisis

En este capítulo, las distintas pruebas se efectuaron con el objetivo de comprobar el rendimiento de *DDS* en distintos escenarios y condiciones. Con ello, se trató de determinar cuáles son las condiciones óptimas de uso de *DDS* y, en cuáles no resulta tan provechoso.

Los test se realizaron en distintos escenarios y condiciones como: uso de máquinas físicas y uso de máquinas virtuales, también se hicieron pruebas de carga progresiva de la máquina con carga sintética y aumento de carga en la red.

En las siguientes subsecciones se explicará con más detalle: el entorno hardware y software usado, herramienta usada para obtener los datos, cómo se ha generado la carga sintética en el procesador, cómo se ha generado la carga de red, y los datos obtenidos.

5.2. Entorno hardware y software del experimento

Las distintas pruebas fueron efectuadas entre una o dos (dependiendo del escenario) máquinas con el hardware y software que se describe a continuación. Los distintos escenarios fueron:

- Misma máquina física.

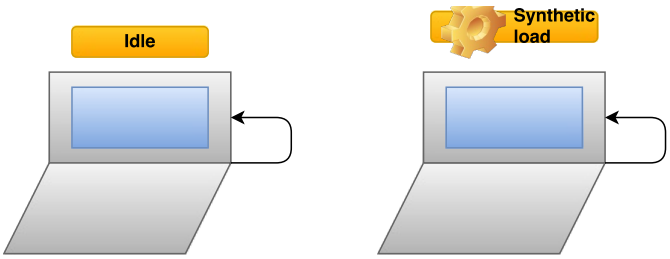

Figura 5.1: Sin carga (izda.) y con carga sintética (dcha.)

- Distintas máquinas físicas.

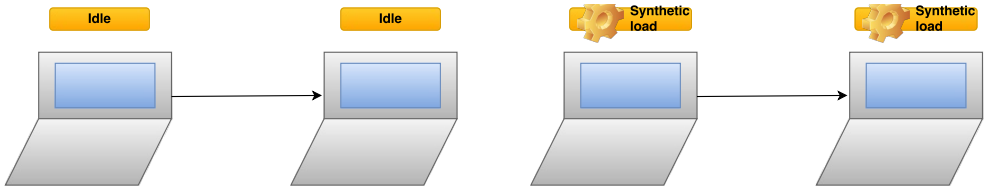

Figura 5.2: Sin carga (izda.) y con carga sintética (dcha.)

- Misma máquina virtual.

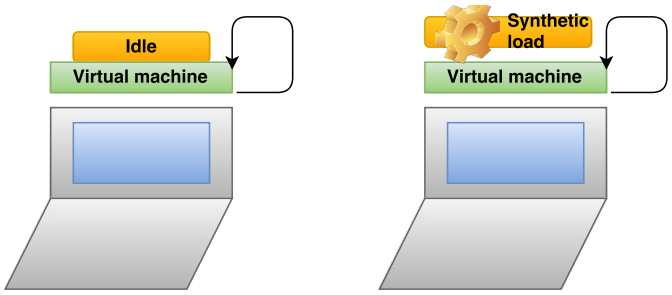

Figura 5.3: Sin carga (izda.) y con carga sintética (dcha.)

- Dos máquinas virtuales misma máquina física.

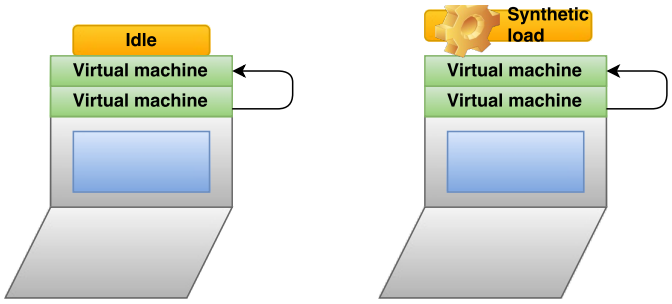

Figura 5.4: Sin carga (izda.) y con carga sintética (dcha.)

- Distinta máquina virtual en máquina distinta física.

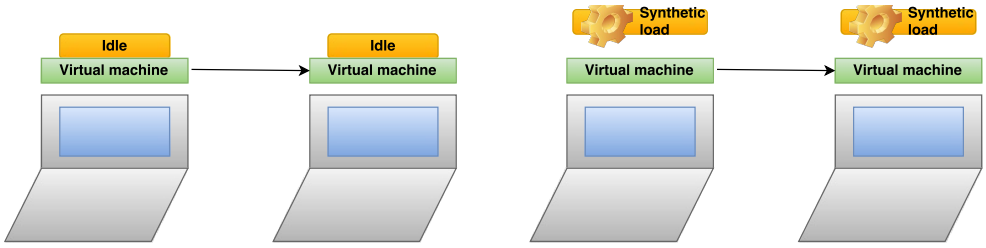

Figura 5.5: Sin carga (izda.) y con carga sintética (dcha.)

Además, hay que señalar que en todos los escenarios las pruebas se hicieron de dos maneras, en una de ellas únicamente se ejecutó el cliente o servidor y, en otro caso, se ejecutaron junto a un pequeño programa. Con este programa se

generaba una carga sintética, mediante la cual el porcentaje de uso de la CPU alcanzó valores progresivos entre el 0 y el 100 % como se explicará más adelante.

Escenario hardware y software de la máquina virtual:

- **Hardware:** QEMU Virtual CPU Versión 2.0.0 de 2 núcleos y 1024 MB de RAM
- **Software:** Versión de KVM asociada al kernel 3.19.0-49-generic #55 14.04.1-Ubuntu SMP
  Ubuntu 12.04.5 LTS de 32 bits
  RTI Connext DDS 5.2.0

Escenario hardware y software de la máquina física o host:

- **Hardware:** Intel Celeron CPU E3400 de doble núcleo a 2.6 Ghz con 1971 MB de RAM
- **Software:** Ubuntu 14.04.3 LTS de 32 bits
  RTI Connext DDS 5.2.0

5.2.1. Descripción de las pruebas

Para comprobar el rendimiento de *DDS* en distintos escenarios se decidió hacer diferentes pruebas de envío de mensajes de distinto tamaño y en distintas condiciones. El tamaño de los mensajes: 1 byte, 4 bytes, 1024 bytes, 2048 bytes, 4096 bytes.

Debido a la naturaleza de *DDS* el intercambio de mensajes siguió un esquema de tipo publicador-suscriptor y el publicador envió 10000 muestras del mismo *topic* al que se ha suscrito el suscriptor. De esta manera podemos asegurar que los resultados obtenidos son más cercanos a la realidad.

Prueba 1, sin carga sintética. En primer lugar, se lleva a cabo el envío de mensajes con cargas de entre 1 y 4096 bytes. En esta situación, en la cual, tanto publicador como suscriptor estaban libres de carga, es decir, la única aplicación corriendo en el sistema en primer plano fue el cliente o servidor o ambos dependiendo del escenario.

Prueba 2, con carga sintética progresiva de 0 % a 100 %. La segunda prueba consistió, también, en el envío de mensajes con cargas de entre 1 y 4096 bytes, pero en este caso el procesador de cliente y servidor fueron cargados de forma sintética y progresiva mediante un programa llamado *lookbusy* como se explicará en la siguiente subsección. Se midió el rendimiento de *DDS* para porcentajes de carga de la CPU de 25 %, 50 %, 75 % y 100 %.

Lookbusy se encargó de generar carga sintética tanto en el publicador, como en el suscriptor en el caso de que estos se ejecutan en máquinas físicas distintas. En el caso en el que se ejecutaran dos máquinas virtuales dentro de un mismo host físico, solo se ejecutó *lookbusy* en el cliente, ya que, la máquina física no soporta la carga de las dos máquinas virtuales por lo que no podemos añadir más carga sin provocar el cierre de la VM por parte de KVM.

La última prueba consistió en comprobar el comportamiento de *DDS* en caso de carga de la red. Para ello, se carga al publicador con la recepción por parte de este de un gran paquete de datos desde un tercer nodo, mientras el publicador enviaba mensajes con cargas de entre 1 y 4096 bytes.

5.2.2. RTIPerf

La herramienta que se usó para comprobar el rendimiento de *DDS* fue *RTIPerf*, la elección de esta herramienta para los test no ha sido casualidad, ya que después de barajar distintas opciones fue la más completa. *RTIPerf* permite seleccionar un gran número de opciones que permiten modificar el test de acuerdo a las características deseadas.

El modo de funcionamiento de la herramienta consiste en simular el paradigma de publicación-suscripción. El publicador envía muestras de datos lo más rápido posible y, después del envío de al menos 100 muestras, el suscriptor envía un mensaje de eco mediante el cual se calcularon distintas estadísticas.

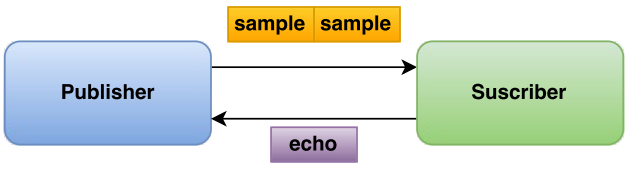

Figura 5.6: Esquema de funcionamiento de RTIPerf

Una de las mejores características de *RTIPerf*, a parte de su alto grado de personalización de los test, es su portabilidad entre plataformas ya que puede usarse en los tres principales sistemas operativos como son Windows, Linux y MAC OS.

Algunos de los parámetros que pueden ser modificados para tanto publicador como suscriptor son:

- *bestEffort*, hace uso de bestEffort este parámetro está por defecto desactivado.
- *dataLen* (bytes), especifica el payload del mensaje y permite el envío de hasta 63000 bytes.
- *domain (ID)*, identifica el dominio de la comunicación para que se pueda realizar la comunicación tanto el publicador como el suscriptor deben de estar en el mismo dominio.
- *enableSharedMemory*, permite el uso de memoria compartida.
- *enableTcpOnly*, establece que todas las comunicaciones de la aplicación deben de usar TCP, por defecto desactivado.
- *multicast*, usa multicast para recibir datos, no está habilitado por defecto.
- *qosprofile (filename)*, se especifica el directorio en el cuál se encuentra el XML con los parámetros de QoS deseados.

Algunos de los parámetros que pueden ser modificados para únicamente por el suscriptor son:

- *numPublishers (count)*, número de publicadores que el suscriptor esperará para empezar.

Algunos de los parámetros que pueden ser modificados para únicamente por el publicador son:

- *latencyCount (count)*, número de muestras que se enviarán antes de enviar un paquete de ping para determinar la latencia.
- *numIter (count)*, número de muestras a enviar.

- *pubRate*, número de muestras por segundo hasta 10.000.000
- *numSubscribers (count)*, número de suscriptores que el publicador esperará para empezar.
- *sleep (millisec)*, tiempo en milisegundos en el que publicador "duerme" entre envíos.

5.2.3. Lookbusy

Se trata de un pequeño programa mediante el cual podemos alterar diversos parámetros de ejecución para simular las diferentes condiciones de carga:

- Carga de la CPU:
  - Porcentaje de utilización de la CPU deseado se puede especificar tanto un intervalo de porcentaje de uso como un valor fijo.
  - Número de núcleos de la CPU a ser utilizado.
  - Modo de carga de la CPU, puede ser lineal(fijo) o en forma de curva (aumento progresivo)
  - Duración del tiempo de porcentaje de pico máximo dentro del modo curva.
- Carga de memoria RAM:
  - Cantidad de memoria a usar, se puede indicar en bytes, kilobytes, megabytes, gigabytes.
  - Tiempo en el que se entra en suspensión entre iteraciones, 1 ms por defecto.
- Uso del disco duro:
  - Cantidad de disco duro a usar, se puede indicar en bytes, kilobytes, megabytes, gigabytes.
  - Tiempo en el que se entra en suspensión entre iteraciones, 1 ms por defecto.
  - Tamaño de los bloques de memoria a usar para operaciones de entrada y salida.

Además, trata de compensar otras cargas que pueda haber en el sistema de manera que se mantengan las características que se desean.

A continuación, se muestra una captura de las opciones que permite la aplicación de *lookbusy*.

Como podemos observar ofrece una gran cantidad de opciones para ajustar de manera óptima las condiciones de carga que se desea que *lookbusy* genere en nuestro ordenador. Por ejemplo:

- Porcentaje de uso de la CPU
- Núcleos en uso
- Cantidad de memoria en uso
- Tamaño de disco en uso

Durante las pruebas, de forma paralela al uso de *lookbusy*, se usó la herramienta *htop* para monitorizar el estado de la carga de la máquina y comprobar que *lookbusy* estaba funcionando de manera correcta.

5.3. Tablas de resultados y gráficas

En esta sección se va a mostrar y analizar los resultados obtenidos de las diferentes pruebas en los distintos escenarios.

5.3.1. Prueba 1. Sin carga sintética

En primer lugar, se va a analizar los resultados obtenidos para el envío de mensajes de tamaño variable (1 byte, 4 bytes, 6 bytes, 1024 bytes, 2048 bytes y 4096 bytes) en un escenario en el que el procesador tiene una carga baja de procesamiento (entre el 2 % y el 6 %), ya que el único proceso que se ejecutó en primer plano fue el publicador o suscriptor o ambos, dependiendo del escenario. A continuación, se muestran los tiempos obtenidos, los cuales están en microsegundos:

	Same host	Different host	Same virtual machine	Two virtual machines in the same host	Two differents virtual machines in two differents hosts
Send a char	174,00	237,00	731,00	1348,00	1455,00
Send a number	137,00	240,00	708,00	2165,00	1828,00
Send a chain of 6 chars	155,00	246,00	941,00	3183,00	1995,00
Send a chain of 1024 chars	166,00	562,00	850,00	3592,00	1568,00
Send a chain of 2048 chars	178,00	726,00	707,00	2156,00	1850,00
Send a chain of 4096 chars	291,00	918,00	930,00	4936,00	1822,00

Tabla 5.1: Tabla de tiempos obtenidos para los distintos escenarios sin carga sintética, tiempos en microsegundos y plataforma DDS

De forma general, cuanto mayor es el número de bytes enviados mayor es el tiempo que se tarda en procesar y transmitir la información. También podemos observar que en la ejecución de la plataforma de publicador-suscriptor cuando se realiza en máquinas virtuales el tiempo aumenta de manera considerable.

En el caso de la comunicación en dos máquinas virtuales en la misma máquina física y dos máquinas virtuales en dos máquinas físicas distintas, en el primer caso los tiempos son bastante mayores y esto puede deberse a que la máquina física que hospeda a las dos máquinas virtuales se encuentra bastante sobrecargada.

A continuación, se muestran las gráficas estadísticas obtenidas para facilitar la visualización y comparación de los datos:

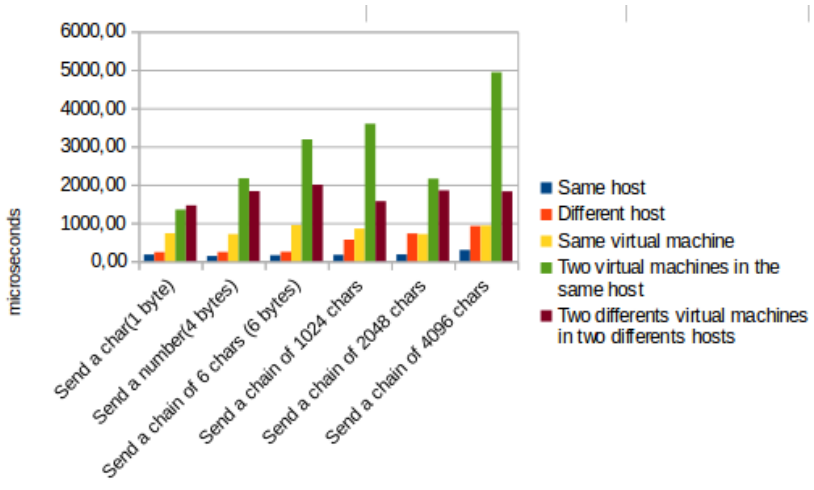

Figura 5.7: Diagrama de tiempos de los distintos escenarios en DDS, sin carga

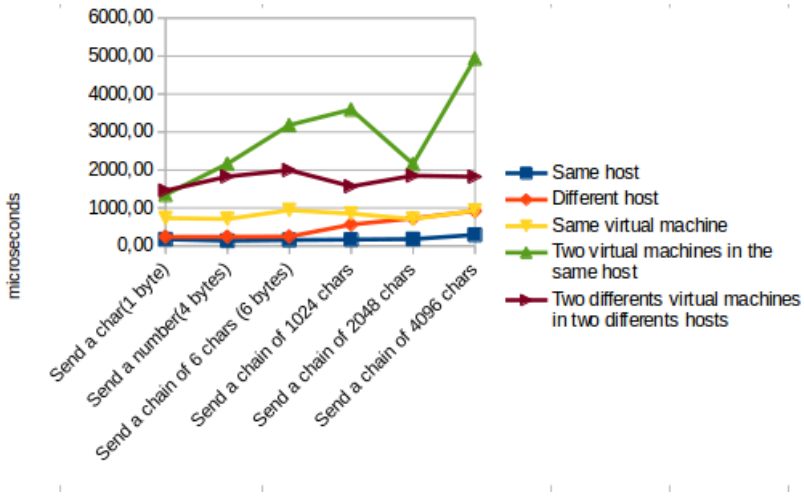

Figura 5.8: Gráfica de tiempos de los distintos escenarios en DDS, con carga

5.3.2. Prueba 2. Carga progresiva

A continuación, se va a mostrar y analizar los datos que se obtuvieron para la prueba de envío de mensajes de 1024 bytes el cuál se considera una carga estándar y para un escenario en el cuál el procesador está cargado a diferentes porcentajes (25 %, 50 %, 75 %, 100 %).

	Same host	Different host	Same virtual machine	Two virtual machines in the same host	Two differents virtual machines in two differents hosts
25 %	112,00	668,00	295,00	2565,00	1577,00
50 %	139,00	639,00	365,00	2364,00	2494,00
75 %	158,00	648,00	486,00	2396,00	1930,00
100 %	146,00	621,00	2179,00	6408,00	5348,00

Tabla 5.2: Tabla de tiempos obtenidos para los distintos escenarios de carga progresiva, tiempos en microsegundos y plataforma DDS

A continuación, se muestran las gráficas estadísticas obtenidas para favorecer la visualización y comparación de los datos:

Se puede apreciar que se sigue respetando en gran medida lo que ocurrió en el escenario anterior sin carga, ya que el rendimiento de la comunicación entre máquinas físicas es mejor que entre máquinas virtuales.

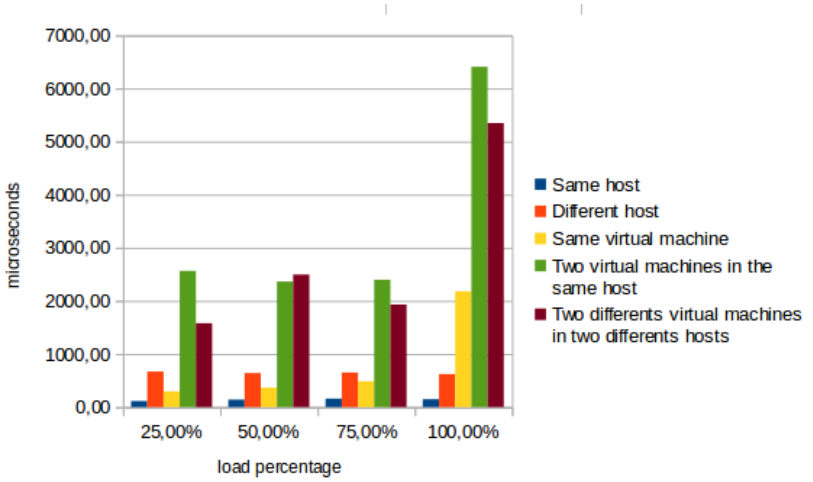

Figura 5.9: Diagrama de tiempos de los distintos escenarios en DDS, carga progresiva

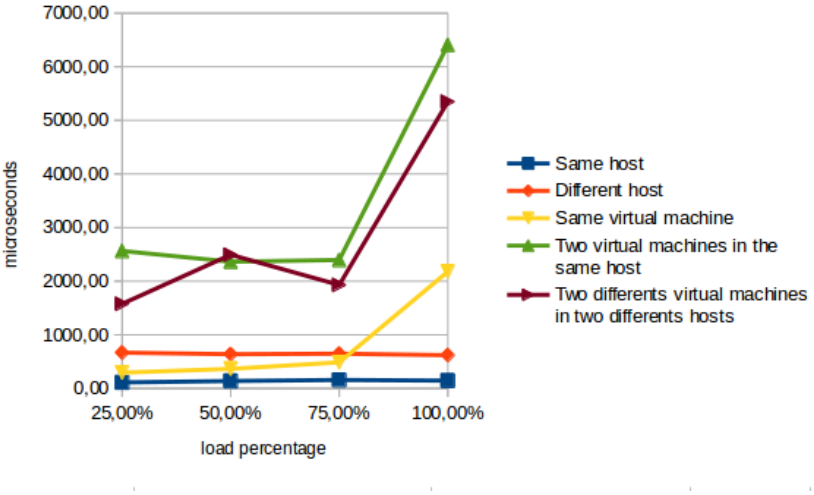

Figura 5.10: Gráfica de tiempos de los distintos escenarios en DDS, con carga

5.3.3. Caso de sistema con carga de red

Por último, se mostrarán los datos obtenidos para el escenario en el cuál se saturó de carga de red las máquinas para así comprobar el efecto que este produce. Se enviaron mensajes de distinta carga (1 byte, 4 bytes, 6 bytes, 1024 bytes, 2048 bytes, 4096 bytes).

A continuación, se muestran las gráficas estadísticas obtenidas para favorecer

	Same host	Different host	Same virtual machine	Two virtual machines in the same host	Two differents virtual machines in two differents hosts
Send a char	247,00	2160,00	320,00	2754,00	5847,00
Send a number	246,00	2105,00	340,00	2636,00	5822,00
Send a chain of 6 chars	241,00	2055,00	296,00	2431,00	4482,00
Send a chain of 1024 chars	246,00	2456,00	241,00	3152,00	4959,00
Send a chain of 2048 chars	274,00	2677,00	289,00	3987,00	4343,00
Send a chain of 4096 chars	718,00	3091,00	375,00	4524,00	5080,00

Tabla 5.3: Tabla de tiempos obtenidos para los distintos escenarios de carga de red, tiempos en microsegundos y plataforma DDS

la visualización y comparación de los datos:

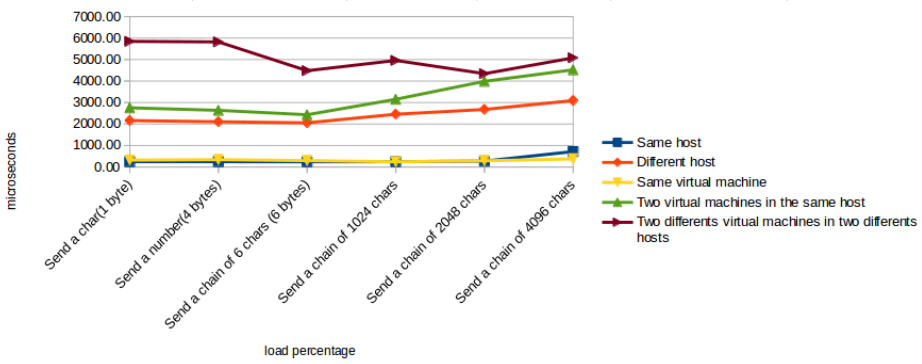

Figura 5.11: Diagrama de tiempos de los distintos escenarios en DDS, carga progresiva

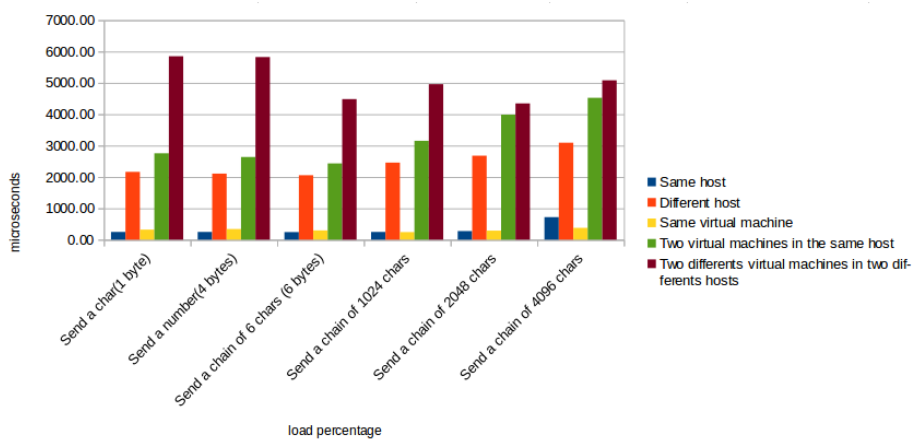

Figura 5.12: Gráfica de tiempos de los distintos escenarios en DDS, con carga

En este escenario podemos apreciar cómo cambia la situación respecto a los anteriores, ya que la carga de red afecta en mayor medida, como es lógico, a la

comunicación entre máquinas físicas distintas y máquinas virtuales hospedadas en distintas máquinas físicas.

5.4. Pruebas con servidor

Para las pruebas realizadas con anterioridad se ha usado un hardware muy limitado, lo cuál en cierta manera ayudaba a simular un sistema empotrado muy usado en sistemas de tiempo real.

En las siguientes pruebas se usa un equipo servidor con gran potencia para que actuó como publicador, con el objetivo de comprobar si se puede mejorar el comportamiento temporal dado el hardware limitado de las máquinas con el que se realizaron las pruebas anteriores.

Como en las pruebas anteriores los escenarios fueron los siguientes:

- Misma máquina física.
- Distintas máquinas físicas.
- Misma máquina virtual.
- Dos máquinas virtuales misma máquina física.
- Distinta máquina virtual en máquina distinta física.

Se llevaron a cabo las pruebas 1 y 2 descritas en la subsección Descripción de las pruebas. Estas pruebas fueron realizadas con parámetros QoS, que aseguraban una cierta calidad en la comunicación.

5.4.1. Parámetros QoS en RTIPerftest

Una de las características más importantes de *DDS* es la de permitir establecer parámetros QoS, ya que, de esta forma *DDS* se puede adaptar a los requisitos temporales y de calidad que pueda requerir una aplicación distribuida. En este

capítulo las pruebas realizadas sobre *DDS* tenían parámetros *DDS* algunos de los cuáles se va a especificar y detallar a continuación.

DDS opera sobre RTPS la cuál es una capa que asegura fiabilidad mediante un mecanismo de asentimiento con mensajes de tipo Heartbeat y ACKNACK con los cuales mediante un número de secuencia y una pequeña caché en la se puede saber que mensajes han llegado al suscriptor y cuáles no. En la siguiente imagen se puede apreciar un pequeño esquema de funcionamiento.

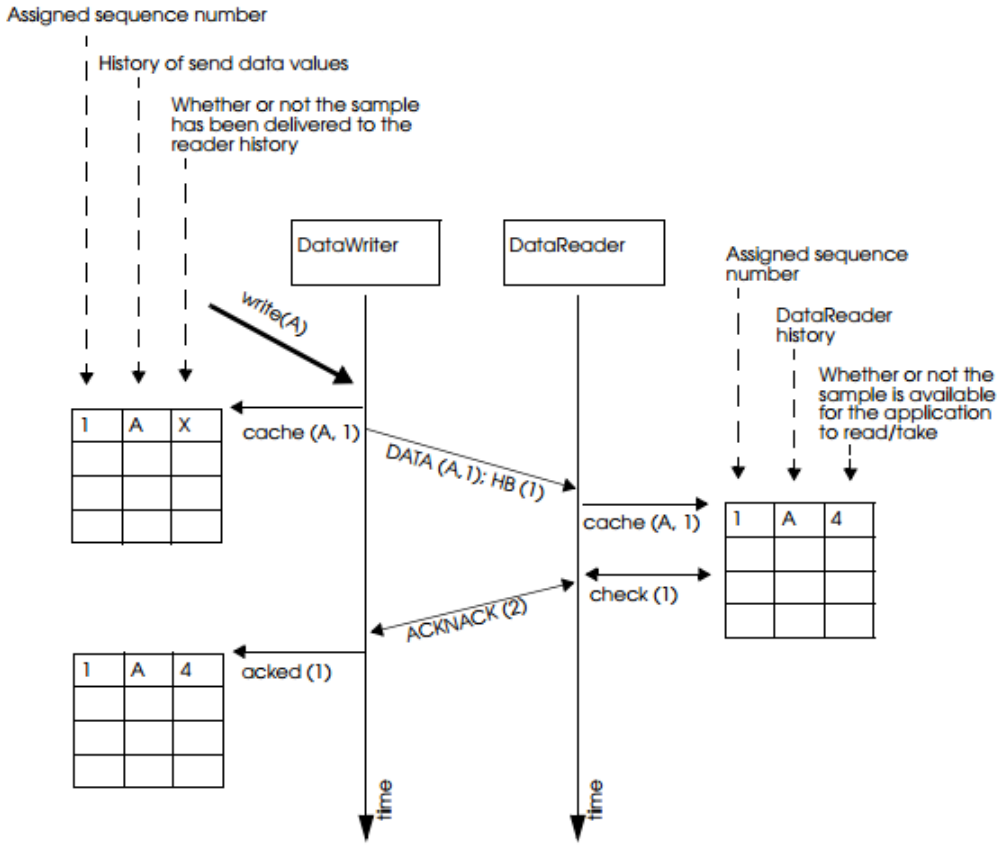

Figura 5.13: Esquema de RTPS, fuente `https://community.rti.com/rti-doc/510/ndds.5.1.0/doc/pdf/RTI_CoreLibrariesAndUtilities_UsersManual.pdf`

En *RTIPerftest* los parámetros de QoS se controlan tanto en el código fuente como en un *XML*. A continuación, se comentará algunos de los más importantes de los cuales *RTIPerftest* hace uso.

Uno de ellos es la desactivación de los ACKs, el publicador va a considerar

que las muestras se han asentido positivamente después de 0.1 milisegundos esto está especificado en el archivo XML que define algunos de los parámetros QoS de la aplicación *RTIPerftest*

```
<protocol>
        <rtps_reliable_writer>
          <disable_positive_acks_min_sample_keep_duration>
            <sec>DURATION_ZERO_SEC</sec>
            <nanosec>100000</nanosec>
          </disable_positive_acks_min_sample_keep_duration>
        </rtps_reliable_writer>
      </protocol>able_writer>
</protocol>
```

Las muestras enviadas por el publicador al suscriptor pueden no estar disponibles si en alguna de las anteriores hay algún error. Esto puede ser ignorado si se activa el parámetro -bestEffort, mediante el cual no tiene en cuenta la recepción por el suscriptor de muestras anteriores. Por ello, se decidió llevar a cabo pruebas en nuestro servidor desactivando el envío de ACK positivos y activando el modo bestEffort y comprobar si el comportamiento temporal de la aplicación *RTIPerftest* de *DDS* mejoraba.

5.4.2. Entorno hardware y software de publicador y suscriptor

Escenario hardware y software de la máquina virtual en el suscriptor:

- **Hardware:** QEMU Virtual CPU Versión 2.0.0 de 2 núcleos y 1024 MB de RAM
- **Software:** Versión de KVM asociada al kernel 3.19.0-49-generic #55 14.04.1-Ubuntu SMP
  Ubuntu 12.04.5 LTS de 32 bits
  RTI Connext DDS 5.2.0

Escenario hardware y software de la máquina física o host en el suscriptor:

- **Hardware:** Intel Celeron CPU E3400 de doble núcleo a 2.6 Ghz con 1971 MB de RAM

- **Software:** Ubuntu 14.04.3 LTS de 32 bits
  RTI Connext DDS 5.2.0

Escenario hardware y software de la máquina virtual en el publicador:

- **Hardware:** QEMU Virtual CPU Versión 2.0.0 de 2 núcleos y 1024 MB de RAM
- **Software:** Versión de KVM asociada al kernel 3.19.0-49-generic #55 14.04.1-Ubuntu SMP
  Ubuntu 12.04.5 LTS de 32 bits
  RTI Connext DDS 5.2.0

Escenario hardware y software de la máquina física o host en el suscriptor:

- **Hardware:** Intel Celeron CPU E3400 de doble núcleo a 2.6 Ghz con 1971 MB de RAM
- **Software:** Ubuntu 14.04.3 LTS de 32 bits
  RTI Connext DDS 5.2.0

5.4.3. Tablas de resultados y gráficas

A continuación, se presentan los resultados obtenidos para las pruebas descritas con anterioridad.

5.4.3.1. Prueba 1. Sin carga sintética

En primer lugar, se va a analizar los resultados obtenidos para el envío de mensajes de tamaño variable (1 byte, 4 bytes, 6 bytes, 1024 bytes, 2048 bytes y 4096 bytes) en un escenario en el que el procesador tiene una carga baja de procesamiento (entre el 2 % y el 6 %), ya que el único proceso que se ejecutaron en primer plano fue el publicador o suscriptor o ambos, dependiendo del escenario. El único cambio con respecto a la prueba 1 anterior, es que el servidor tomará el rol del publicador. A continuación, se muestra con los tiempos obtenidos los cuales están en microsegundos:

Misma máquina física:

	Average	Standard deviation	Minimum value	Maximum value
Send a char	85,00	41,4	28,00	211,00
Send a number	95,00	43,5	25,00	252,00
Send a chain of 6 chars	89,00	43,6	23,00	249,00
Send a chain of 1024 chars	98,00	45,1	26,00	996,00
Send a chain of 2048 chars	95,00	44,1	27,00	329,00
Send a chain of 4096 chars	90,00	43,3	29,00	364,00

Tabla 5.4: Tabla de tiempos obtenidos el escenario de misma máquina física sin carga sintética, tiempos en microsegundos y plataforma DDS

Distinta máquina física:

	Average	Standard deviation	Minimum value	Maximum value
Send a char	248,00	47,50	171,00	1374,00
Send a number	256,00	40,10	173,00	918,00
Send a chain of 6 chars	247,00	14,00	157,00	1167,00
Send a chain of 1024 chars	508,00	50,90	426,00	1872,00
Send a chain of 2048 chars	643,00	47,20	576,00	1690,00
Send a chain of 4096 chars	809,00	98,20	743,00	4311,00

Tabla 5.5: Tabla de tiempos obtenidos el escenario de distinta máquina física sin carga sintética, tiempos en microsegundos y plataforma DDS

Misma máquina virtual:

	Average	Standard deviation	Minimum value	Maximum value
Send a char	47,00	34,70	21,00	1238,00
Send a number	53,00	35,70	19,00	433,00
Send a chain of 6 chars	59,00	40,60	21,00	600,00
Send a chain of 1024 chars	58,00	39,40	20,00	451,00
Send a chain of 2048 chars	49,00	32,10	22,00	498,00
Send a chain of 4096 chars	50,00	35,10	22,00	371,00

Tabla 5.6: Tabla de tiempos obtenidos el escenario de misma máquina virtual sin carga sintética, tiempos en microsegundos y plataforma DDS

Dos máquinas virtuales en el mismo host:

	Average	Standard deviation	Minimum value	Maximum value
Send a char	201,00	117,30	91,00	1765,00
Send a number	207,00	130,60	100,00	1916,00
Send a chain of 6 chars	216,00	146,20	94,00	2364,00
Send a chain of 1024 chars	207,00	117,20	100,00	1538,00
Send a chain of 2048 chars	229,00	136,40	114,00	2093,00
Send a chain of 4096 chars	243,00	132,60	124,00	3662,00

Tabla 5.7: Tabla de tiempos obtenidos el escenario de dos máquinas virtuales en el mismo host sin carga sintética, tiempos en microsegundos y plataforma DDS

Dos máquinas virtuales en distinto host:

	Average	Standard deviation	Minimum value	Maximum value
Send a char	410,00	310,70	245,00	7286,00
Send a number	396,00	321,80	238,00	12045,00
Send a chain of 6 chars	400,00	296,50	248,00	7765,00
Send a chain of 1024 chars	685,00	330,00	507,00	7291,00
Send a chain of 2048 chars	800,00	304,10	643,00	6413,00
Send a chain of 4096 chars	980,00	302,50	810,00	10776,00

Tabla 5.8: Tabla de tiempos obtenidos el escenario de dos máquinas virtuales en distinto host sin carga sintética, tiempos en microsegundos y plataforma DDS

En estas gráficas se puede observar a modo de síntesis los tiempos medios obtenidos para los distintos escenarios:

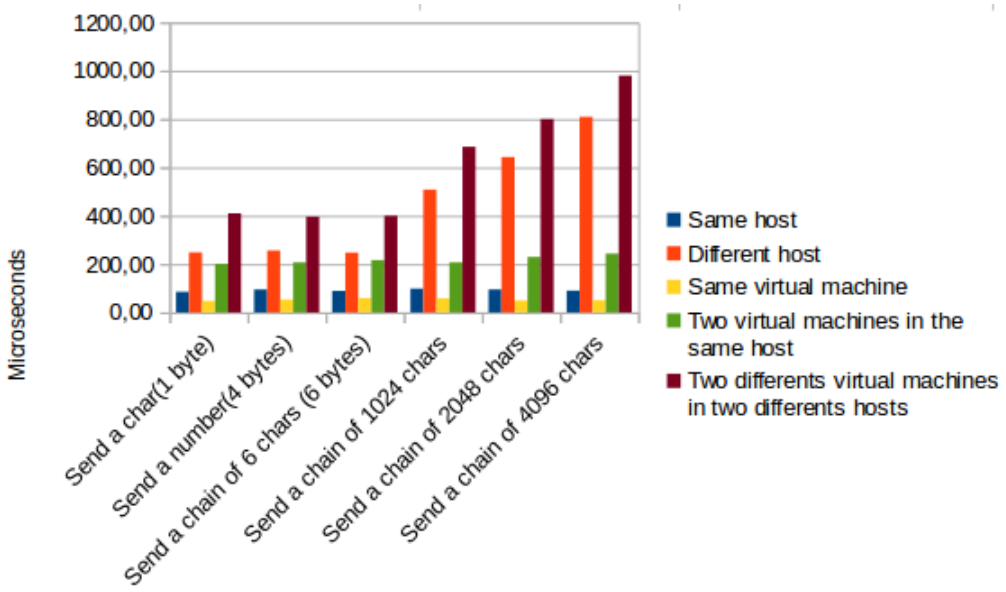

Figura 5.14: Diagrama de funcionamiento de DDS sin carga sintética, medido desde máquina servidor

Se puede apreciar que los tiempos obtenidos en todos los escenarios son mejores respecto a las pruebas anteriores. Esto se debe al uso de la nueva máquina física más potente que la máquina física usada para las pruebas anteriores como publicador. El escenario en el que mejor se puede apreciar esta mejora, es el de

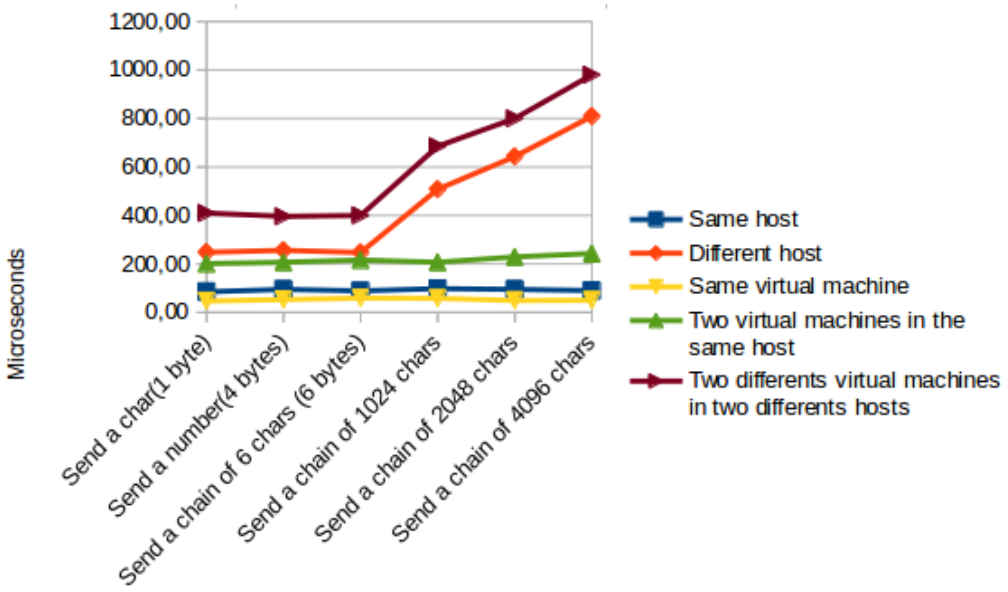

Figura 5.15: Gráfica de funcionamiento de DDS sin carga sintética, medido desde máquina servidor

dos máquinas virtuales hospedadas en una misma máquina física. Hay que destacar que, cuando la comunicación se realiza entre dos máquinas físicas distintas, los tiempos son peores respecto a los escenarios en los que la comunicación se realiza en la misma máquina física.

5.4.3.2. Prueba 2. Con carga sintética

A continuación, se va a mostrar y analizar los datos que se obtuvieron para la prueba de envío de mensajes de 1024 bytes el cual se considera una carga estándar y para un escenario en el cuál el procesador está cargado a diferentes porcentajes (25 %, 50 %, 75 %, 100 %). El único cambio con respecto a la prueba 2 anterior, es que el servidor tomará el rol del publicador. A continuación, se muestran los tiempos obtenidos, los cuales están en microsegundos:

Misma máquina física:

	Average	Standard deviation	Minimum value	Maximum value
25 %	44,00	19,80	23,00	1445,00
50 %	38,00	21,40	22,00	1682,00
75 %	34,00	12,50	22,00	489,00
100 %	78,00	39,40	25,00	209,00

Tabla 5.9: Tabla de tiempos obtenidos el escenario de misma máquina física con carga sintética, tiempos en microsegundos y plataforma DDS

Distinta máquina física:

	Average	Standard deviation	Minimum value	Maximum value
25 %	476,00	123,20	405,00	5416,00
50 %	461,00	63,60	399,00	2567,00
75 %	453,00	112,70	397,00	7466,00
100 %	508,00	47,20	412,00	2062,00

Tabla 5.10: Tabla de tiempos obtenidos el escenario de distinta máquina física con carga sintética, tiempos en microsegundos y plataforma DDS

Misma máquina virtual:

	Average	Standard deviation	Minimum value	Maximum value
25 %	52,00	67,10	21,00	5181,00
50 %	63,00	81,00	20,00	4618,00
75 %	67,00	103,40	21,00	4133,00
100 %	57,00	38,70	21,00	561,00

Tabla 5.11: Tabla de tiempos obtenidos el escenario de misma máquina virtual con carga sintética, tiempos en microsegundos y plataforma DDS

Dos máquinas virtuales en el mismo host:

	Average	Standard deviation	Minimum value	Maximum value
25 %	720,00	674,50	470,00	29812,00
50 %	813,00	873,50	458,00	20414,00
75 %	1100,00	1366,80	465,00	30140,00
100 %	678,00	276,80	507,00	5870,00

Tabla 5.12: Tabla de tiempos obtenidos el escenario de dos máquinas virtuales en el mismo host con carga sintética, tiempos en microsegundos y plataforma DDS

Dos máquinas virtuales en distinto host:

	Average	Standard deviation	Minimum value	Maximum value
25 %	212,00	258,20	80,00	7111,00
50 %	362,00	776,80	84,00	14251,00
75 %	707,00	1403,70	82,00	17716,00
100 %	206,00	107,40	87,00	1594,00

Tabla 5.13: Tabla de tiempos obtenidos el escenario de dos máquinas virtuales en distinto host con carga sintética, tiempos en microsegundos y plataforma DDS

En estas gráficas se puede observar a modo de síntesis los tiempos medios obtenidos para los distintos escenarios:

Se puede decir que, de forma general, los tiempos de todos los escenarios son mejores respecto a las pruebas anteriores, en las que no se contaba con la máquina física servidor que actúa como publicador. En este caso, sí que parece que la carga afecta más al escenario de dos máquinas virtuales en una misma máquina física. De la misma manera, no produce tanto perjuicio en los tiempos del escenario de dos máquinas virtuales hospedadas en distinta máquina física.

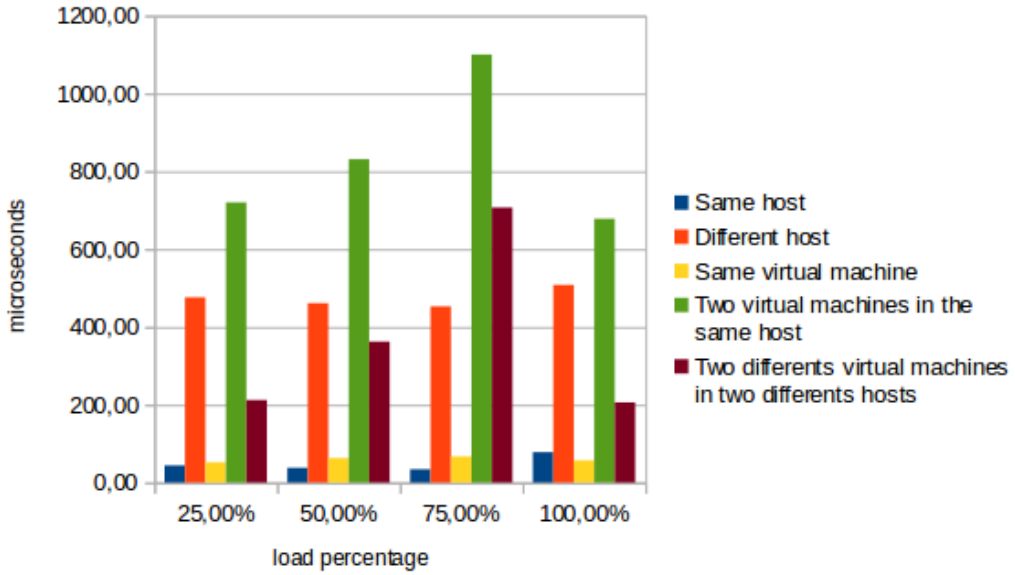

Figura 5.16: Diagrama de funcionamiento de DDS con carga sintética, medido desde máquina servidor

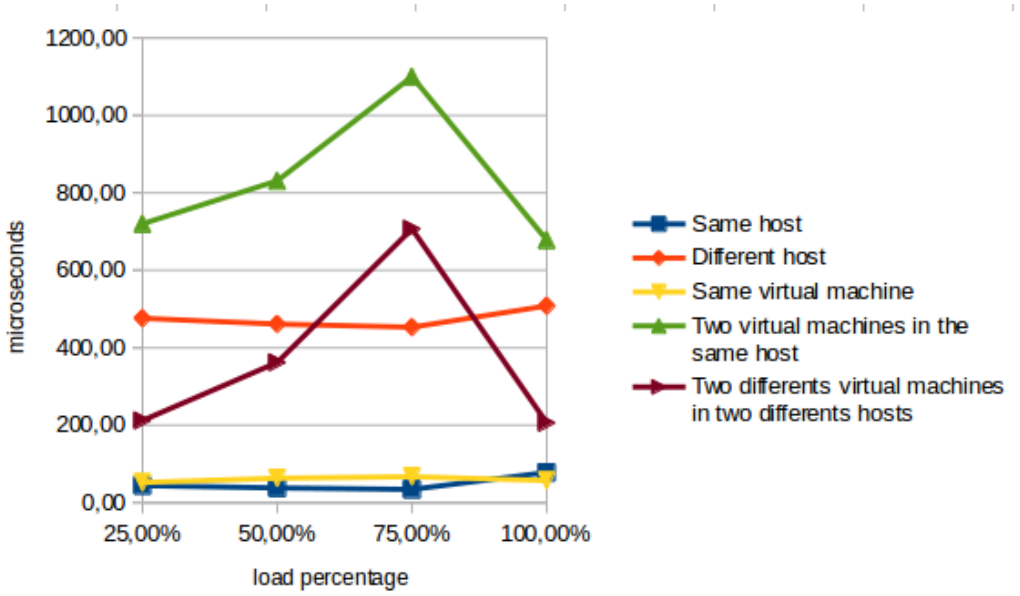

Figura 5.17: Gráfica de funcionamiento de DDS sin carga sintética, medido desde máquina servidor

5.4.3.3. Prueba 3. Con carga sintética y Best Effort

A continuación, se va a mostrar y analizar los datos que se obtuvieron para la prueba de envío de mensajes de 1024 bytes el cuál se considera una carga estándar y para un escenario en el cuál el procesador está cargado a diferentes porcentajes (25 %, 50 %, 75 %, 100 %). El único cambio con respecto a la prueba anterior es que se desactiva el envío de ACKs y se activa el modo *Best Effort*. A continuación, se muestran los tiempos obtenidos, los cuales están en microsegundos:

Misma máquina física:

	Average	Standard deviation	Minimum value	Maximum value
25 %	41,00	7,60	22,00	84,00
50 %	35,00	7,90	18,00	67,00
75 %	30,00	5,10	21,00	65,00
100 %	148,00	21,30	37,00	186,00

Tabla 5.14: Tabla de tiempos obtenidos el escenario de misma máquina física con carga sintética y Best Effort, tiempos en microsegundos y plataforma DDS

Distinta máquina física:

	Average	Standard deviation	Minimum value	Maximum value
25 %	465,00	34,50	426,00	1937,00
50 %	452,00	42,50	413,00	2904,00
75 %	443,00	36,80	405,00	3403,00
100 %	550,00	13,50	500,00	1302,00

Tabla 5.15: Tabla de tiempos obtenidos el escenario de distinta máquina física con carga sintética, tiempos en microsegundos y plataforma DDS

Misma máquina virtual:

	Average	Standard deviation	Minimum value	Maximum value
25 %	30,00	19,70	16,00	725,00
50 %	30,00	20,50	16,00	862,00
75 %	59,00	65,90	17,00	3323,00
100 %	32,00	82,70	17,00	8042,00

Tabla 5.16: Tabla de tiempos obtenidos el escenario de misma máquina virtual con carga sintética, tiempos en microsegundos y plataforma DDS

Dos máquinas virtuales en el mismo host:

	Average	Standard deviation	Minimum value	Maximum value
25 %	125,00	47,70	84,00	3802,00
50 %	131,00	215,90	80,00	7131,00
75 %	152,00	384,10	80,00	9871,00
100 %	206,00	119,60	82,00	1220,00

Tabla 5.17: Tabla de tiempos obtenidos el escenario de dos máquinas virtuales en el mismo host con carga sintética, tiempos en microsegundos y plataforma DDS

Dos máquinas virtuales en distinto host:

	Average	Standard deviation	Minimum value	Maximum value
25 %	597,00	363,20	485,00	16799,00
50 %	608,00	487,60	481,00	14475,00
75 %	640,00	690,30	476,00	25692,00
100 %	636,00	128,90	506,00	4992,00

Tabla 5.18: Tabla de tiempos obtenidos el escenario de dos máquinas virtuales en distinto host con carga sintética, tiempos en microsegundos y plataforma DDS

En estas gráficas se puede observar a modo de síntesis los tiempos medios obtenidos para los distintos escenarios:

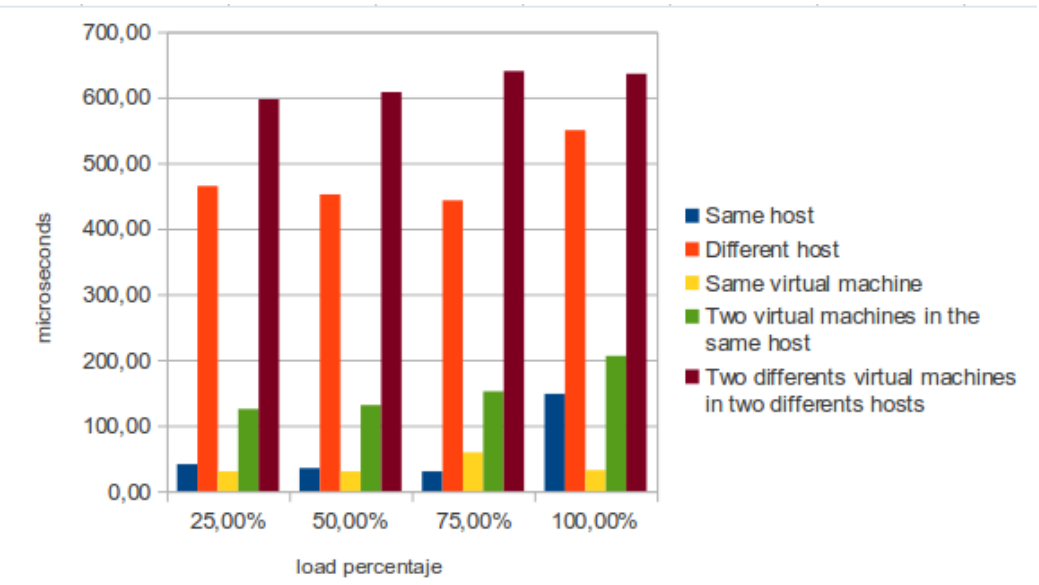

Figura 5.18: Diagrama de funcionamiento de DDS con carga sintética y best effort, medido desde máquina servidor

Esta prueba se compara con la anterior para poder apreciar el efecto de *Best Effort* en comparación con la anterior prueba que soportaba parámetros de calidad de servicio. Se puede observar que el escenario en el que más efecto ha hecho el uso de *Best Effort* ha sido el de dos máquinas virtuales en misma máquina física, que mejora significativamente. En el resto de escenarios no se aprecia una

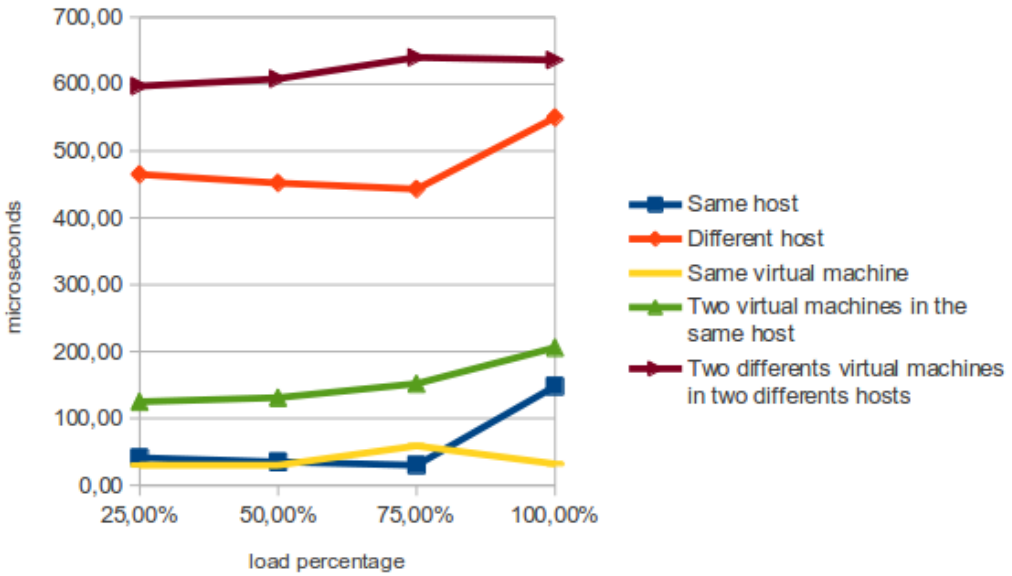

Figura 5.19: Gráfica de funcionamiento de DDS sin carga sintética y best effort, medido desde máquina servidor

llamativa variación de los tiempos, disminuye en todos los escenarios ligeramente, con excepción del escenario de dos máquinas virtuales en distinta máquina física que incrementa ligeramente los tiempos obtenidos.

Capítulo 6

Creación de una aplicación DDS y su integración a la nube

6.1. Introducción

Durante este capítulo se va a explicar cómo se ha desarrollado una aplicación basada en middleware de comunicaciones de tipo publicador-suscriptor, se usa *RTI DDS* para ello y, además, se integra en un entorno en la nube para ello se utiliza *OpenStack*.

De esta forma, se pretende acercar a un plano más práctico lo desarrollado a lo largo de esta memoria. Adicionalmente, se explica con detalle el despliegue de la nube y distintos aspectos de la programación de una aplicación de middleware de comunicaciones como, por ejemplo, calidad de servicio.

El objetivo de la aplicación es la comunicación entre nodos haciendo uso de middleware de comunicaciones de tipo publicador-suscriptor. Haciendo uso de *OpenStack* y, para así sacar partido de las características que nos aporta el uso de una plataforma de estas características.

6.2. Arquitectura y descripción de la aplicación

La aplicación consistirá en crear una aplicación que recoge distintos datos, de tipo meteorológico, a través de unos sensores y su posterior procesado y configuración. La aplicación se estructura en *DDS* con un modelo publicador-suscriptor y datos centralizados, en la cuál se va a poder enviar mensajes con distintos parámetros modificables.

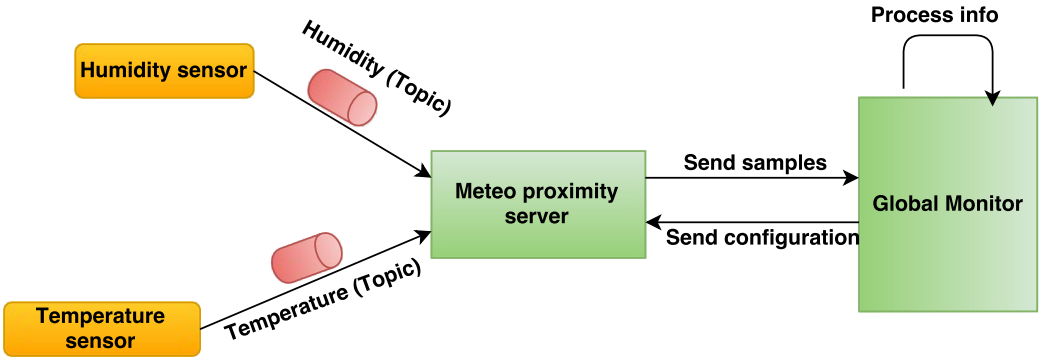

Figura 6.1: Esquema de funcionamiento de la aplicación

Esta aplicación se basa en una disposición genérica, que puede usarse para el envío de mensajes de forma independiente del contenido y se pueden modificar parámetros importantes como el nombre del *topic* y número de muestras a utilizar.

Para el desarrollo de la aplicación se hace uso de un *topic* dinámico, del cual se puede modificar su nombre, la diferencia con respecto a un *topic* normal es que se puede cambiar su estructura en función de las necesidades del usuario.

Como podemos observar en la imagen 6.1 la aplicación consta de:

- Dos sensores meteorológicos, los cuáles se encargan de recoger muestras del entorno y enviarlas al nodo suscriptor.
- Un nodo monitor, que actúa con el rol de suscriptor que monitoriza y envía los datos al nodo configurador.
- Nodo configurador, encargado de realizar cálculos estadísticos a partir de las muestras recibidas y enviar modificaciones de la configuración a los nodos sensores.

Las comunicaciones entre los distintos nodos de la aplicación se llevarán a cabo mediante *topics* y haciendo uso del protocolo de transporte DCPS. En la aplicación se hará uso de tres *topics* "Humidity", "Temperature", "Configuration". En la figura 6.2 se muestra el esquema de comunicaciones entre las distintas entidades de la aplicación.

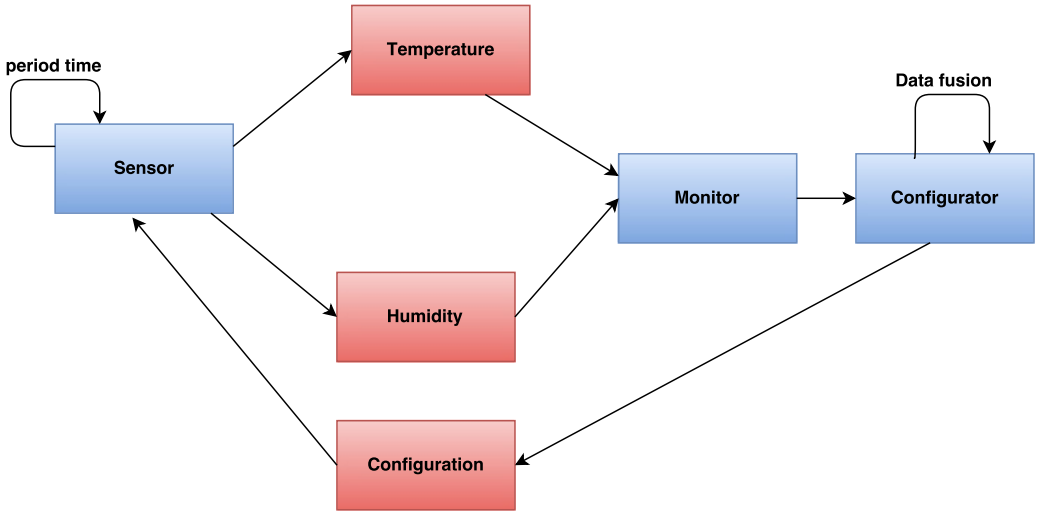

Figura 6.2: Esquema de comunicaciones de la aplicación

6.2.1. Uso de un *topic* dinámico

Usualmente las comunicaciones mediante *DDS* hacen uso de *topics* y estos están asociados a un tipo de datos definido por el lenguaje de programación que estemos usando.

En este caso, se va a definir un *topic* dinámico, se lleva a cabo mediante la creación de una clase en la cual determinar los parámetros requeridos por la aplicación. La estructura consta de un nombre para el *topic*, un número de secuencia y un payload. La clase en la cual se define el *topic* dinámico contiene un método para su creación y otro para su borrado.

```
struct HelloWorld {
   string<TOPIC_NAME> prefix;
   long sampleId;
   sequence<octet, TOPIC_MAX_PAYLOAD_SIZE> payload;
};
```

Mediante el uso de un *topic* dinámico se puede modificar parámetros tan importantes del *topic* como su nombre o payload, lo que resulta muy útil para que la plataforma desarrollada sea fácilmente portable y modificable.

http://www.omg.org/spec/DDS-XTypes/1.1/PDF/

6.2.2. Análisis de la aplicación

La aplicación se ha desarrollado de forma que permite su escalabilidad y flexibilidad gracias a su fácil portabilidad. También es importante destacar que se ha hecho uso de parámetros QoS para poder adaptar de una forma más precisa a las necesidades de la aplicación.

Los parámetros QoS de una aplicación deben ajustarse las necesidades de fiabilidad de respuesta requeridos por esta. En el caso de la aplicación descrita hay que tener en cuenta las siguientes características:

- El tamaño medio de los mensajes es de 1024 Kb.
- El escenario usado se trata de la comunicación entre dos máquinas físicas.
- La carga media de ejecución de la aplicación es del 25 %.

Se van a usar parámetros QoS que aseguren cierta fiabilidad:

- *History*, en este caso usaremos *KEEP_ALL*, con el cuál se especifica que las muestras se mantendrán hasta que sean procesadas por el suscriptor.
- Otros parámetros como *Availability, Durability, Durability Service, DataWriterProtocol* y *DataReaderProtocol*, lo cuáles van a asegurar un envío ordenado, almacenar en caché, uso de ACK y NACK y su frecuencia de envío.

Como se puede apreciar en la figura inferior la aplicación puede crecer en número de dispositivos de manera sencilla y sin necesidad de hacer modificaciones en la aplicación. De esta manera se podría añadir de forma sencilla más sensores y monitores al sistema.

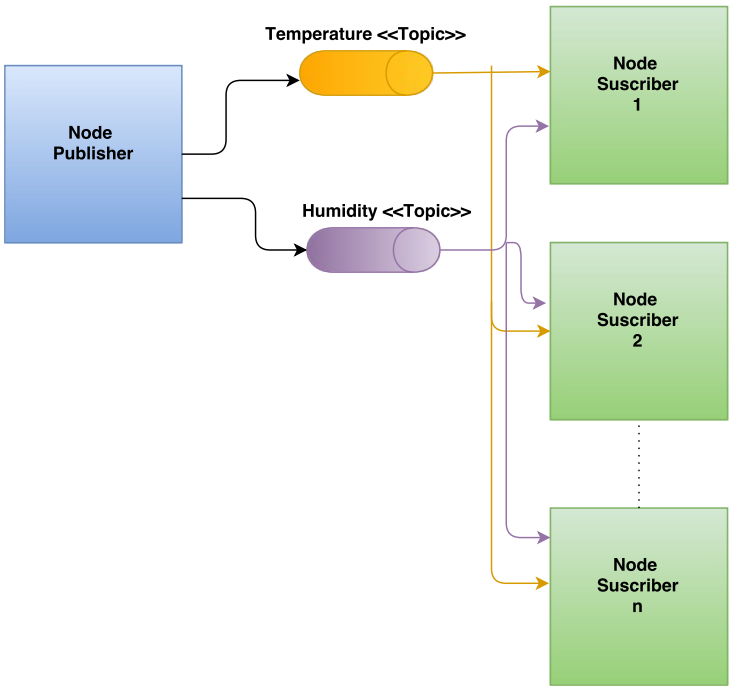

Figura 6.3: Ejemplo de escalado de nodos monitor

La aplicación puede usarse tanto en máquinas físicas como en máquinas virtuales. Como se sabe en el caso de usa de máquinas virtuales aporta múltiples ventajas:

- Adaptar mediante la virtualización nuestras necesidades de hardware.
- Ejecutar en una máquina física mediante máquinas virtuales dos o más nodos de nuestra aplicación de forma independiente entre sí, es una forma de conseguir un ahorro monetario.
- Facilita la escalabilidad y la migración entre máquinas virtuales.

6.3. ¿Por qué usar la nube para nuestra aplicación?

En la anterior sección hemos explicado las características más importantes acerca de nuestra aplicación basada en el uso de middleware de comunicaciones en concreto *RTI DDS*. [19]

En esta sección se va a desarrollar de forma sencilla como podría integrarse esa aplicación en la nube. Una de las dudas que podría surgir al lector es cómo puede mejorar nuestra aplicación mediante su integración en la nube. Algunas de las razones son estas:

- **Mejoras en escalabilidad.** Esto es debido a que la nube se basa en el uso de sistemas distribuidos lo que permite que se pueda escalar fácilmente y sin un límite tan marcado como puede tener los sistemas tradicionales de almacenamiento.

- **Acceso desde cualquier lugar.** El sistema de almacenamiento tradicional es limitado a la hora de permitir su acceso ya que en muchas ocasiones solo es posible su acceso de forma local. Mediante el uso de la nube podemos acceder desde cualquier lugar gracias a HTTP.

- **Mejoras en flexibilidad por el uso de la virtualización.** Esto es porque gracias a la naturaleza de la nube disponemos de un pool de recursos y nosotros podemos adaptarlo de forma sencilla a nuestras necesidades, y posteriormente modificarlo en caso de que nuestros requisitos cambien.

- **Reducción de costes.** Se estima que podría cortarse el gasto en almacenamiento hasta en 10 veces, esto es debido al gran ahorro que se producirá en electricidad e incluso en personal que antes se encargaba de gestión de almacenamiento en local. Todo ello sin renunciar a disponibilidad y fiabilidad.

- **Mejor aprovechamiento de los recursos.** Múltiples estudios sugieren que usualmente solo se hace uso de un 5 %-20 % del hardware disponible en las instalaciones.

6.3.1. Despliegue de OpenStack mediante Devstack

Para desplegar *OpenStack* se hizo uso de Devstack, se trata de un script con el cual crear de forma rápida un entorno *OpenStack*. Es una forma rápida y sencilla de empezar a trabajar con *OpenStack*, además nos permite realizar múltiples ajustes de configuración. *OpenStack* ofrece una plataforma en la nube la cual cuenta con múltiples módulos que podemos usar o no en función de nuestras necesidades. [20]

Devstack carga la configuración de un archivo llamado *local.conf* en el cuál se especifican entre otros:

- Contraseñas de administrador, base de datos, servicios, etc.
- Módulos *OpenStack* a activar.
- Rango de IPs usadas en la aplicación.
- Plugins.

Este podría ser un ejemplo del fichero local.conf :

```
#cloud-config

users:
  - default
  - name: stack
    lock_passwd: False
    sudo: ["ALL=(ALL) NOPASSWD:ALL\nDefaults:stack
       !requiretty"]
    shell: /bin/bash

write_files:
  - content: |
       #!/bin/sh
       DEBIAN_FRONTEND=noninteractive sudo apt-get -qqy
          update || sudo yum update -qy
       DEBIAN_FRONTEND=noninteractive sudo apt-get install
          -qqy git || sudo yum install -qy git
       sudo chown stack:stack /home/stack
       cd /home/stack
```

```
      git clone
         https://git.openstack.org/openstack-dev/devstack
      cd devstack
      echo '[[local|localrc]]' > local.conf
      echo ADMIN_PASSWORD=password >> local.conf
      echo DATABASE_PASSWORD=password >> local.conf
      echo RABBIT_PASSWORD=password >> local.conf
      echo SERVICE_PASSWORD=password >> local.conf
      ./stack.sh
   path: /home/stack/start.sh
   permissions: 0755

runcmd:
  - su -l stack ./start.sh
```

Simplemente faltaría arrancar el script llamado *stack.sh* para comenzar el despliegue de *OpenStack*. DevStack se puede ejecutar en múltiples escenarios como:

- Despliegue de *OpenStack* en una única máquina virtual.
- Despliegue de *OpenStack* en una única máquina física.
- Despliegue de *OpenStack* en plataforma multinodo.

6.3.2. Puesta en funcionamiento de la aplicación en OpenStack

Cómo se ha descrito con anterioridad, los beneficios de integrar la aplicación en *OpenStack* son múltiples como: escalabilidad, flexibilidad, uso eficiente de recursos, etc. En este apartado del capítulo se dan algunas recomendaciones para la adaptación de la aplicación a una estructura en la nube.

OpenStack está constituido de forma modular y cada módulo aporta una funcionalidad a nuestra arquitectura. En el apartado teórico de *OpenStack* en el capítulo del Estado del Arte ya se describieron algunos de los módulos de *OpenStack*, a modo de recordatorio los principales son:

- **Nova.** Controlador de estructura de computación en la nube de forma masivamente escalable.

- **Cinder.** Almacenamiento a nivel de bloque, el cuál es virtualizado, accesible a través de Nova de forma transparente al usuario.
- **Swift.** Sistema de almacenamiento escalable distribuido que puede ser usado por organizaciones para almacena datos de forma eficiente, segura y barata.
- **Neutron.** Permite la gestión de redes que permite la comunicación entre distintos componentes.
- **Horizon.** Proporciona interfaz gráfica en forma de web para poder controlar de forma más sencilla e intuitiva los distintos módulos.

De entre estos módulos hay que destacar *Nova*. Mediante *Nova*, o también *Horizon*, el usuario puede interactuar con *OpenStack* para ejecutar una instancia (máquina virtual configurada en base a intereses específicos) para ello previamente se autenticará al usuario. A continuación, Nova-scheduler determinará el nodo en el cuál se ejecutará la instancia, y Nova-compute llevará a cabo la ejecución sobre el nodo. Mediante *Nova* o *Neutron* se puede establecer la configuración de red.

En función de los requisitos de la aplicación, se necesita un número variable de nodos y en cada nodo se ejecutan algunos de los módulos de *OpenStack*. [19]

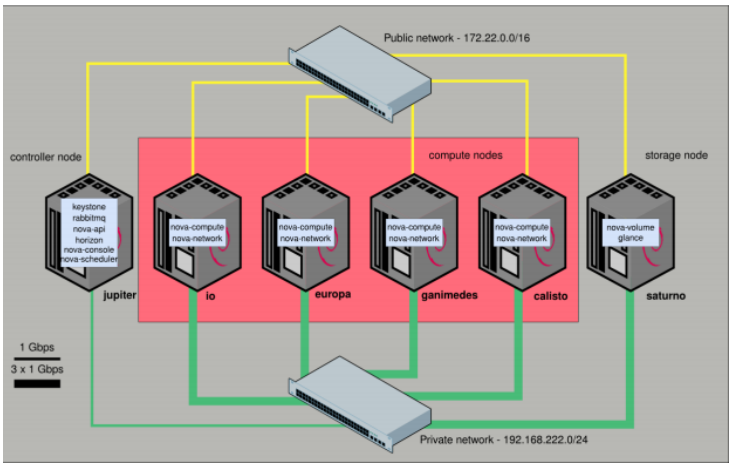

Figura 6.4: Ejemplo de estructura de nodos en OpenStack, imagen de IES Gonzalo Nazareno

Como se puede apreciar en la imagen hay tres tipos de nodos:

- Nodo de computación, en el cuál se ejecutan las instancias y, por tanto, necesita una gran cantidad de memoria RAM y buenos procesadores.
- Nodo controlador, el cuál se encarga de tareas como la gestión, autenticación y planificación. No es necesario un equipo potente.
- Nodo de almacenamiento, en este nodo se almacenan las imágenes de las máquinas virtuales y sirven como almacenamiento para las instancias en ejecución. Se necesitan equipos con gran capacidad de disco duro.

Por último, hay que destacar la gestión de la red en nuestra plataforma sobre *OpenStack*. La configuración de red de los nodos puede ser pública o privada.

- Configuración de red pública, de esta forma la aplicación será accesible por cualquier equipo desde Internet. Si se tratase de una estructura de la nube privado, la aplicación será accesible sólo desde dentro de la red local en la que se encuentre.
- Configuración de red privada, mediante la cual se realiza la mayor parte de comunicaciones entre nodos. De esta forma se aumenta la seguridad de la nube.

Capítulo 7

Conclusions and future work

In this chapter the conclusions reached are presented following the development of this Final Degree Project with regard to the objectives initially presented. In addition, some ideas are introduced with the aim of future improvements on this project.

7.1. Overall conclusions

The main objective of this project has been to analyse the temporal behaviour of middleware platforms in different scenarios in synthetic environments with CPU load and network load.

First, the study of the temporal behaviour of the Ice platform was carried out. To this end, an application which sent messages of different sizes was developed. It was measured the time elapsed from the client to receive a response from the server. The test was performed with synthetic CPU load and without it.

Regarding the results of the Ice tests, the scenarios in which virtual machines were used led to worse results regarding the implementation of middleware on physical machines. Also is noteworthy when messaging is more than 1024 bytes in size, times of execution start to increase significantly. There are no notable differences in the temporal behaviour of the middleware execution over different scenarios with synthetic load regarding their behaviour without synthetic load.

The DDS study was made by using the tool RTIPerf, in order to simplify the test. Moreover, you could enable and disable quality of service parameters easily. Thus we analyse the behaviour of DDS in different scenarios. CPU tests were performed unloaded, with synthetic CPU load, network load and also one of the machines was replaced by another more powerful to see how much the times obtained could be improved. Testing quality of service parameters and use of Best Effort was also made.

After analysing the data from DDS tests, the conclusions are similar to those reached in the tests performed to evaluate the performance of Ice. The scenarios in which virtual machines are used get worse times than in those where the test is made on physical machines. In addition, it should be noted that the scenarios with virtual machines accuse way more synthetic CPU load and network load scenarios than physical machines. Later on, the tests were performed by replacing one machine with another powerful machine to evaluate how much could the new machine improve the performance. The results of the tests using this powerful machine improved the test times, later tests were performed with Best Effort (the previous test used QOS parameters) and the results are significantly better. In the tests it was noted that times achieved were bad in most scenarios, comprising the ones from 1024 bytes and a CPU load of more than 75 %.

Finally, an application that simulated a distributed environment similar to a weather station was developed. In this chapter, aspects of development of DDS-based applications are detailed.Then, their integration in the cloud is analyzed.

In conclusion, the development of distributed applications using virtualization is possible, though you have some limitations and currently it does not have a good performance as running on physical machines. These limitations, as we have seen, can be overcome if we use powerful computers and by using quality of service parameters if it is strictly necessary.

7.2. Future work

Listed below you can find a few ideas that could be made for a more complete analysis of the temporal behaviour of middleware software:

- Analysis of middleware using transport protocols apart from TCP, and UDP, SSL / TLS or WebSockets.

- Analysis on a platform middleware cloud like OpenStack.
- Analysis of middleware using other operating systems.
- Analysis of the behaviour of other middleware as Ice Storm.
- Development of a distributed application with real-time requirements using virtualization.
- Expand the study on the quality of service parameters and their effects on the temporal behaviour of the middleware.

Capítulo 8

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones obtenidas tras el desarrollo de este Trabajo de Fin de Grado respecto a los objetivos que inicialmente se presentaron. Adicionalmente, se introducen ideas con la finalidad de poder introducir futuras mejoras sobre este proyecto.

8.1. Conclusiones generales

El objetivo principal de este proyecto fue el de analizar el comportamiento temporal de plataformas middleware en distintos escenarios y en entornos de carga sintética de la CPU y carga de red.

En primer lugar, se procedió a realizar el estudio del comportamiento temporal en la plataforma Ice. Para ello se desarrolló una simple aplicación, la cual enviaba mensajes de distinto tamaño. Se calculaba el tiempo que tardaba el cliente en recibir una respuesta del servidor desde que se enviaba el mensaje. Se realizaron pruebas con carga sintética de la CPU y sin ella.

Respecto a los resultados obtenidos de las pruebas de *Ice*, se puede decir que en los escenarios en los cuales se usaron máquinas virtuales se obtuvieron resultados algo peores respecto a la ejecución del middleware en máquina física. También hay que destacar que, es a partir del envío de mensajes de un tamaño superior a los 1024 bytes cuando los tiempos empiezan a aumentar de forma no-

table. No existen diferencias destacables en el comportamiento temporal de la ejecución del middleware de los distintos escenarios con carga sintética respecto a su comportamiento sin carga sintética.

El estudio de *DDS* esta vez se hizo uso de la herramienta *RTIPerf* ya que nos permitía la realización de los test de una manera fácil. Por otra parte, se podía habilitar y deshabilitar parámetros de calidad de servicio de manera sencilla. De esta manera se realiza un análisis del comportamiento de *DDS* en los distintos escenarios. Se realizaron pruebas con la CPU sin carga, con carga sintética de la CPU, con carga de red y, además, se sustituyó una de las máquinas por otra más potente para comprobar cuánto podía mejorar los tiempos obtenidos. También se realizaron pruebas con parámetros de calidad de servicio y con uso de Best Effort.

Después de analizar los datos extraídos de las pruebas de *DDS*, las conclusiones son semejantes a las que se llegó en las pruebas realizadas para evaluar el rendimiento de *Ice*. Los escenarios en los cuales se usan máquinas virtuales obtienen peores tiempos en los test que los realizados en máquinas físicas. Adicionalmente hay que señalar, que los escenarios con máquinas virtuales acusan de mayor manera la carga sintética de CPU y la carga de red que los escenarios de máquinas físicas. Posteriormente, se realizaron pruebas sustituyendo una de las máquinas por otra más potente para evaluar cuánto podría mejorar el rendimiento. El resultado de las pruebas con el uso de esta máquina de mayor potencia mejoró de forma notable los tiempos, posteriormente se realizaron pruebas con Best Effort (las anteriores tenían parámetros de calidad de servicio) y los resultados obtenidos son apreciablemente mejores. En las pruebas se apreció que, en la mayoría de escenarios, a partir de 1024 bytes y también de una carga de CPU superior al 75 % los tiempos obtenidos empeoraban en gran medida.

Finalmente, se desarrolló una aplicación que simulaba un entorno distribuido similar a una estación meteorológica. En este capítulo se detallaron aspectos de desarrollo de aplicaciones basadas en *DDS*, y posteriormente se analizó su integración en la nube.

La conclusion general del trabajo es que las aplicaciones distribuidas son generalmente adecuadas para su despliegue en escenarios virtualizados. Sin embargo, aplicaciones con requerimientos temporales, tienen algunas limitaciones que deben ser consideradas mediante un cuidadoso análisis, especialmente para entornos en los que los recursos son bastante limitados.

8.2. Líneas futuras

A continuación, se muestran algunas ideas que se podrían realizar para conseguir un análisis del comportamiento temporal del middleware más completo:

- Análisis del middleware haciendo uso de protocolos de transporte distintos a TCP, como UDP, SSL/TLS o WebSockets.
- Análisis del middleware sobre una plataforma en la nube como OpenStack.
- Análisis del middleware haciendo uso de otros Sistemas Operativos.
- Análisis del comportamiento de otros middlewares como Ice Storm.
- Desarrollo de una aplicación distribuida con requisitos de tiempo real haciendo uso de virtualización.
- Ampliar el estudio sobre los parámetros de calidad de servicio y sus efectos en el comportamiento temporal del middleware.

Apéndice A

Planificación

A continuación, se expondrá la planificación del proyecto en sus distintas etapas y se detallará la duración y fechas en las que tuvieron lugar.

Tarea	Inicio	Duración	Fin
Reuniones iniciales	1/10/2015	2	2/10/2015
Tema a investigar	1/10/2016	1	1/10/2016
Definición de primeras tareas	2/10/2016	1	2/10/2016
Investigación y documentación	5/10/2015	27	10/11/2015
Estudio de máquinas virtuales	5/10/2015	7	13/10/2015
Estudio de ICE y DDS	14/10/2015	13	30/10/2015
Estudio de OpenStack	2/11/2015	5	6/11/2015
Estudio de planificadores	9/11/2015	2	10/11/2015
Análisis y test para ICE	12/11/2015	33	15/1/2015
Parámetros a analizar de ICE	12/11/2015	5	18/11/2015
Desarrollo de código para test	19/11/2015	15	9/12/2015
Realización de test	10/12/2015	8	21/12/2015
Análisis de los resultados	11/1/2016	5	15/1/2016
Análisis y test para DDS	1/2/2016	32	15/3/2016
Parámetros a analizar de DDS	1/2/2016	5	5/2/2016
Instalación de herramientas test	16/11/2015	6	23/11/2015
Realización de test	16/2/2016	15	7/3/2016
Análisis de los resultados	8/3/2016	6	15/3/2016
Desarrollo de aplicación	16/3/2016	31	27/4/2016
Análisis de requisitos	16/3/2016	5	22/3/2016
Modelado de aplicación	23/3/2015	4	28/3/2016
Desarrollo de código	29/3/2016	9	8/4/2016
Realización de pruebas	11/4/2016	5	15/4/2016
Integración en OpenStack	18/4/2015	8	27/4/2016
Memoria TFG	28/4/2016	25	1/6/2016
Escritura de memoria	28/4/2016	20	25/5/2016
Corrección y adaptación de memoria	26/5/2016	5	1/6/2015

Tabla A.1: Tabla de datos para el diagrama de Gantt

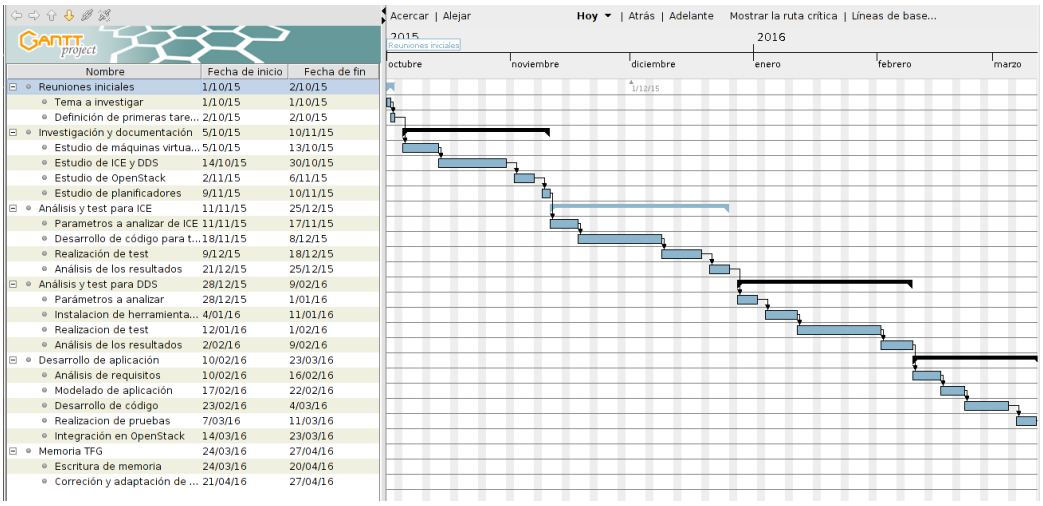

Figura A.1: Diagrama de Gant

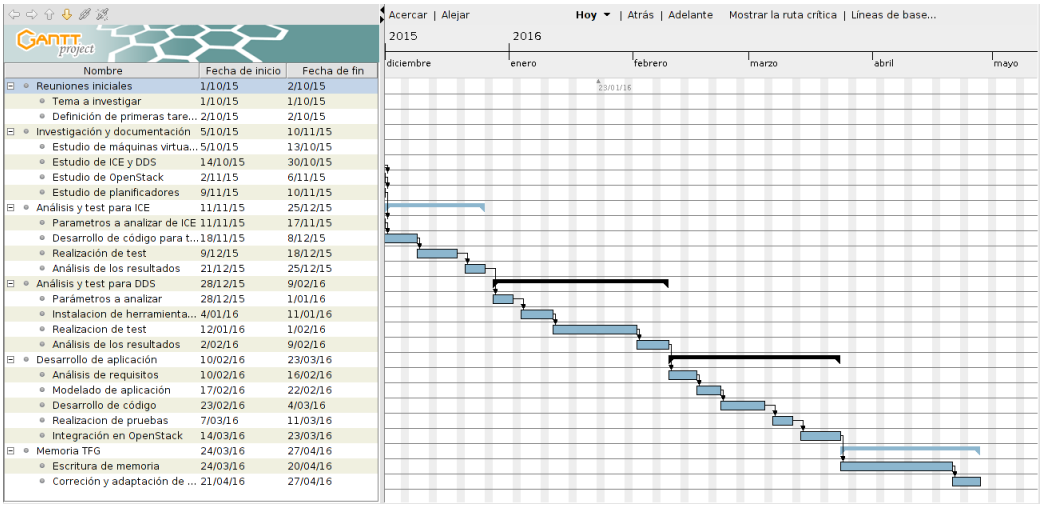

Figura A.2: Diagrama de Gant 2

Apéndice B

Presupuesto

En este capítulo del anexo se van a especificar los costes que se han producido para la realización del proyecto. Este presupuesto se va a clasificar por: **coste de personal**, referido al importe que debe ser abonado a las distintas personas que han colaborado en la realización del proyecto; **recursos materiales**, referido al coste asociado al uso de hardware y software para la realización del proyecto; **otros gastos**, como el acceso a internet.

- **Autor:** Jorge Domínguez Poblete
- **Departamento de Ingeniería Telemática:**
- **Descripción del proyecto:**
  - **Título:** Estudio del comportamiento temporal de entornos distribuidos virtualizados
  - **Duración:** 7 meses
  - **Tasa de costes indirectos:** 20 %
- **Presupuesto total del proyecto:** Consultar tabla B.1
- **Subcontratación de tareas:** No hay subcontratación de tareas
- **Otros costes indirectos:** No existen otros costes indirectos

En la siguiente tabla se desglosa de forma detallada los costes descritos:

Concepto	Cantidad (€)	Coste (€)	%Proyecto	Dedicación (meses)	Depreciación (meses)	Total (€)
Coste personal						
Graduado en Ingeniería Telemática	1	2.694,39	-	7	-	18.860,73
Ingeniero Senior	1	4.289,54	-	0.7	-	3.002,68
Subtotal	-	-	-	-	-	21.863,41
Coste material						
PC laboratorio	2	400	100	8	60	150
Servidor	1	800	100	2	60	26,67
Subtotal	-	-	-	-	-	176,67
Otros costes						
Conexión a Internet	1	30	-	9	-	270
Subtotal	-	-	-	-	-	270
Total	-	-	-	-	-	22.310,08

Tabla B.1: Tabla de presupuesto

Apéndice C

Instalación KVM

Este anexo explica la instalación de KVM sobre un ordenador sobremesa con procesador que permite virtualización.

1. Para empezar, debemos abrir una terminal y dar permisos de superusuario por ello escribiremos sudo previamente al comando que queremos usar.

2. En primer lugar, se comprueba si la CPU soporta virtualización. Para ello se usa el siguiente comando:

   ```
   sudo grep -c "svm\|vmx" /proc/cpuinfo
   ```

   Si el resultado obtenido es 0, desgraciadamente, la CPU no soporta virtualización, si obtenemos 1 o más la CPU soporta virtualización.

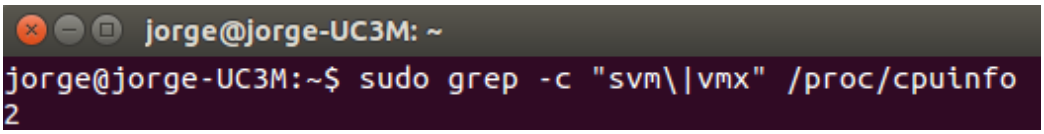

Figura C.1: Instalando KVM paso 2

3. A continuación, se instala una serie de librerías y aplicaciones que permite entre otras cosas tener una interfaz gráfica y poder modificar distintos parámetros de la configuración de red. Para proceder a la instalación se usa el siguiente comando:

```
sudo apt-get install qemu-kvm libvirt-bin
   bridge-utils virt-manager
```

4. Ahora hay que crear un usuario y añadirlo al grupo de libvirtd para poder usar KVM. Se usaran los siguientes comandos:

```
sudo adduser jorge
sudo adduser jorge libvirtd
```

5. Ahora se comprueba que todo funciona de manera correcta. Se va a usar este comando:

```
virsh -c qemu:///system list
```

   Debe aparecer de la siguiente manera:

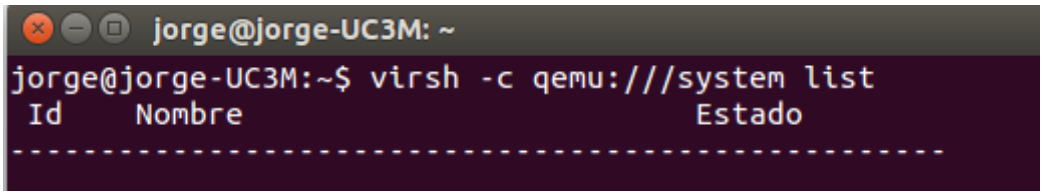

Figura C.2: Instalando KVM paso 5

6. ¡Enhorabuena ha completado la instalación de KVM!

Apéndice D

Configurando un bridge público en KVM

Este anexo explica de forma general cómo configurar un bridge público de dos formas distintas:

- **Modificando fichero interfaces.** Se accede al fichero de texto interfaces que se encuentra situado en el directorio /etc/network. Hay que tener permisos root para modificarlo. Se reemplaza la interfaz eth0 por una nueva interfaz llamada br0 que actuará como bridge. Se configuran distintos parámetros como su IP que será asignada por DHCP; los puertos que se usaran para realizar el bridge; si usar o no Spanning Tree Protocol; máximo tiempo de espera del sistema para configuración del bridge; tiempo de espera de las interfaces para unirse al bridge. Por último, hay que reiniciar la configuración de las interfaces de red para que cargue el nuevo fichero de configuración. Se usa el comando:

  ```
  /etc/init.d/networking restart
  ```

  En versiones recientes de Ubuntu como la 14.04 hay que usar:

  ```
  sudo service network-manager restart
  ```

- **Usando interfaz de KVM.** Para ello, hay que modificar la forma de conectividad de la máquina virtual desde KVM. Se accede desde el menú principal

de KVM a Editar>Detalles de la conexión. Hay que seguir los siguientes pasos:

1. En primer lugar, se añade una nueva interfaz, para ello se pincha en el botón con la cruz verde como se puede ver en la siguiente figura.

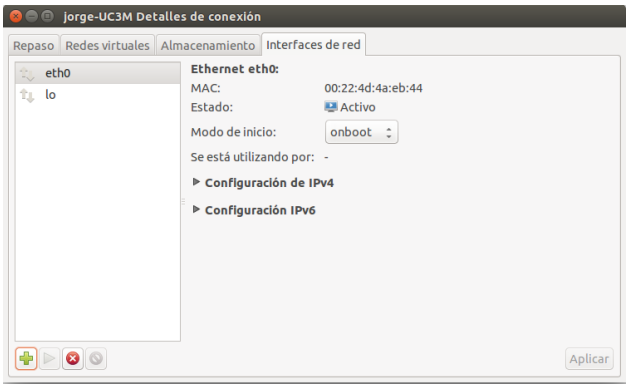

Figura D.1: Configurando un bridge público I

2. A continuación, se selecciona el tipo de interfaz que debe ser un puente.

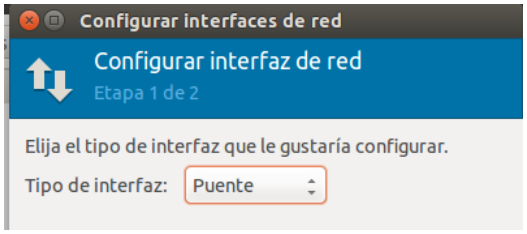

Figura D.2: Configurando un bridge público II

3. Después configuramos la interfaz como se detalla en la siguiente imagen.

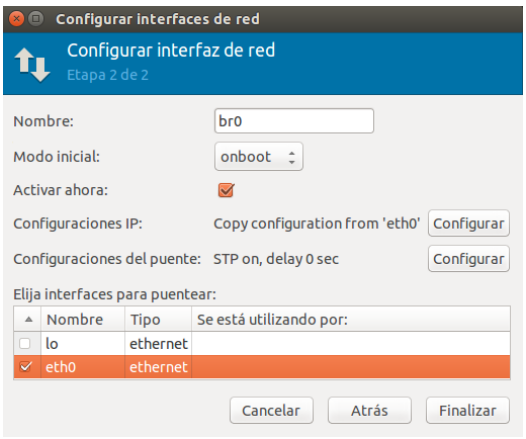

Figura D.3: Configurando un bridge público III

4. Una vez realizado esto debemos comprobar mediante el comando (ifconfig) que la configuración de red queda de la siguiente manera en la máquina física y en la máquina virtual.

```
jorge@jorge-UC3M:~$ ifconfig
br0       Link encap:Ethernet  direcciónHW 00:22:4d:4a:eb:44
          Direc. inet:163.117.141.49  Difus.:163.117.141.255  Másc:255.255.255.0
          Dirección inet6: 2001:720:410:1030:c1fb:9ada:502e:aed4/64 Alcance:Global
          Dirección inet6: 2001:720:410:1030:222:4dff:fe4a:eb44/64 Alcance:Global
          Dirección inet6: fe80::222:4dff:fe4a:eb44/64 Alcance:Enlace
          ACTIVO DIFUSIÓN FUNCIONANDO MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:5788 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:3682 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:0
          Bytes RX:6266806 (6.2 MB)  TX bytes:856342 (856.3 KB)

eth0      Link encap:Ethernet  direcciónHW 00:22:4d:4a:eb:44
          ACTIVO DIFUSIÓN FUNCIONANDO MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:5843 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:3843 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:1000
          Bytes RX:6375578 (6.3 MB)  TX bytes:885738 (885.7 KB)
          Interrupción:16

lo        Link encap:Bucle local
          Direc. inet:127.0.0.1  Másc:255.0.0.0
          Dirección inet6: ::1/128 Alcance:Anfitrión
          ACTIVO BUCLE FUNCIONANDO  MTU:65536  Métrica:1
          Paquetes RX:3620 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:3620 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:0
          Bytes RX:8572612 (8.5 MB)  TX bytes:8572612 (8.5 MB)

virbr0    Link encap:Ethernet  direcciónHW d6:1e:ad:d9:f9:b5
          Direc. inet:192.168.122.1  Difus.:192.168.122.255  Másc:255.255.255.0
          ACTIVO DIFUSIÓN MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:0 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:0 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:0
          Bytes RX:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Figura D.4: Configuración en máquina física

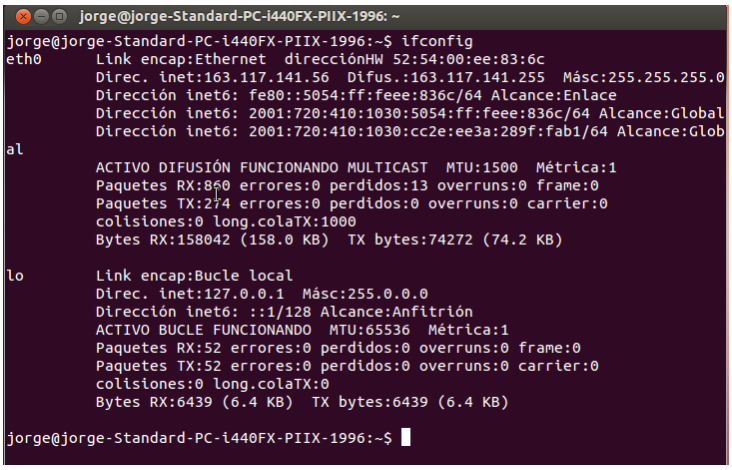

Figura D.5: Configuración en máquina virtual

Bibliografía

[1] Christine Leja. *Implementing Server Virtualization At Southwestern Illinois College*. (2010)

[2] Licencia de RTI Connext DDS. *https://www.rti.com/resources/research-programs.html*

[3] Estándar DDS definido por OMG *http://portals.omg.org/dds/*

[4] Protocolo RTPS *https://wiki.wireshark.org/Protocols/rtps*

[5] Planificador de tareas. *Linux Sytem Programming*. o'Reilly, Robert Love

[6] Virtualización. *Documentation*. https://en.wikipedia.org/wiki/Virtualization

[7] Máquina virtual. *Documentation*. https://en.wikipedia.org/wiki/Virtualization

[8] Tipos de hypervisor. *Documentation*. http://www.virtzone.net/the-difference-between-a-type-2-hypervisor-and-a-type-1-hypervisor/

[9] Sistemas de tiempo real. *Documentation*. https://en.wikipedia.org/wiki/Real-time_computing

[10] Overcommit en KVM. *Documentation*. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Administration_Guide/chap-Virtualization-Tips_and_tricks-Overcommitting_with_KVM.html

[11] Libvirt. *Documentation*. https://wiki.archlinux.org/index.php/Libvirt

[12] Libvirt. *Documentation*. http://www.xenproject.org/

[13] VMWare. *Documentation*. http://www.vmware.com/es

[14] VirtualBox. *Documentation*. https://www.virtualbox.org/

[15] OpenStack. *Documentation*. http://www.gonzalonazareno.org/cloud/material/intro-openstack.pdf

[16] CloudStack. *Documentation*. https://cloudstack.apache.org/about.html

[17] Ice. *Documentation*. https://zeroc.com/products/Ice

[18] Amar Kapadia, Sreedhar Varma, Kris Rajana: *Implementing Computing Storage with OpenStack Swift*.

[19] Alberto Molina Coballes, Jesús Moreno León, José Domingo Muñoz Rodríguez, Cayetano Reinaldos Duarte, Germán Cervantes Abad, Juan Pedro López Luna, Miguel Ángel Montero Navarro, Alejandro Roca Alhama, Miguel Ángel Ibáñez Mompeán, Carmelo Molina Castro, José Antonio Bravo López: *Implantación y puesta a punto de la infraestructura de un cloud computing privado para el despliegue de servicios en la nube*.

[20] DevStack. *Documentation*. http://docs.openstack.org/developer/devstack/guides/multinode-lab.html

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] J.Cano, M.García-Valls, P. Bastanta-Val. *Component framwork for supporting safe and dynamic replacement in real-time systems.* RIAI - Revista Iberoamericana de Automática e Informática Industrial, vol. 11(1), pp. 98-108. 2014.

[27] 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.

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

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

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

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

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

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

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

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

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

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

[54] L. Cappa-Banda, M. García-Valls. *Experimenting with a load-aware communication middleware for CPS domain*. ITNG. (2016) ——————- Referencias de otras tecnologías de middleware ————

[55] ZeroC Inc.: The Internet Communications Engine. http://www.zeroc.com/ice.html (2003)

[56] Object Management Group. *A Data Distribution Service for Real-time Systems Version 1.2*. 2007.

[57] Sun Microsystems: Java^{TM} Remote Method Invocation API (2016) http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/

[58] Object Management Group: The Common Object Request Broker. Architecture and Specification, Version 3.3 (November 2012) http://www.omg.org/spec/CORBA/3.3

[59] ISO/IEC Information Technology Task Force (ITTF). *OASIS AMQP1.0 – Advanced Message Queuing Protocol (AMQP), v1.0 specification*. I(SO/IEC 19464:2014). 2014.