Diseño y desarrollo de un sistema para gestionar y ejecutar simulaciones en clusters de Raspberry Pi

Design and development of a system to manage and run simulations in Raspberry Pi clusters

Trabajo Fin de Grado

Grado en Ingeniería de Computadores

Curso 2020-2021

Pablo Román Morer Olmos

Miguel Pérez de la Rubia

Dirigido por:

Alberto Núñez Covarrubias

Pablo Cerro Cañizares

AGRADECIMIENTOS

Ambos queremos agradecer el trabajo de Alberto y Pablo, por su inquebrantable paciencia y sus consejos, no siempre debidamente aplicados.
Pablo & Miguel

Pablo Morer Olmos
A mi familia, por su apoyo incondicional durante toda mi vida académica.
A mis amigos, por hacer que estos años de carrera hayan sido inmejorables.
Festejando los momentos buenos y amenizando los no tan buenos.

Miguel Pérez de la Rubia
A mi familia, por darme la posibilidad de llegar aquí.
A mi pareja, por aguantarme pese a que todo suena a bip bop bop bip.
A Dani y Aitor, por despejarme y ayudarme a resolver los bugs a horas intempestivas.
A los Amijos, que durante tanto tiempo nos hemos contado las alegrías y las penas y han sabido escucharme.
A Yeste, por sacarme de casa a respirar.

Resumen

Diseño y desarrollo de un sistema para gestionar y ejecutar simulaciones en clústers de Raspberry Pi

Este proyecto tiene como objetivo diseñar y desarrollar una metodología que permita analizar el rendimiento de aplicaciones MPI en entornos distribuidos con distintas configuraciones.

Para lograr este objetivo, una misma aplicación MPI se procesa en dos fases. La primera fase consiste en desarrollar una aplicación distribuida que genera la traza de una aplicación MPI ejecutada sobre un *cluster* de placas *Raspberry Pi*. La traza contiene las llamadas a funciones que se realizan. Seguidamente, se representa gráficamente el resultado. La segunda fase, una vez obtenida la traza de ejecución, consiste en la generación de escenarios de simulación en SIMCAN, los cuales serán utilizados para simular la ejecución de la traza en distintas configuraciones de entornos distribuidos. Estas simulaciones se realizan de forma remota en un *cluster* de placas *Raspberry Pi*.

Palabras clave

Clúster, MPI, Raspberry Pi, Pruebas, Rendimiento, Ejecución, Distribuido, Simulación, Traza.

Abstract

Design and development of a system to manage and run simulations in Raspberry Pi clusters

This project aims at designing and developing a new methodology that allows studying the efficiency of MPI applications in distributed environments using different configurations.

In order to achieve this objective, the MPI application under study is processed in two different phases. The first one consists in developing a distributed application that generates the trace of an MPI application executed in a Raspberry Pi cluster. This trace contains the calls invoked in the application, which is graphically displayed. The second phase consists in generating simulation scenarios - using SIMCAN - that represent the execution of the trace in different distributed environments. These simulations are remotely executed in a *Raspberry Pi* cluster.

Keywords

Cluster, MPI, Raspberry Pi, Tests, Efficiency, Implementation, Distributed, Simulation, Trace

ÍNDICE DE CONTENIDOS

1 - Introducción

En este capítulo se detallan los principales objetivos del proyecto, su alcance y su motivación para llevarlo a cabo. Asimismo, se expone el planteamiento del trabajo establecido para su realización.

1.1 Objetivos

El objetivo principal de este Trabajo de Fin de Grado es diseñar e implementar una metodología que permita analizar el rendimiento de aplicaciones MPI en entornos distribuidos. En este Trabajo de fin de grado han participado dos personas, por ello aún teniendo el objetivo común cada uno de los integrantes ha desarrollado una serie de objetivos individuales para cada parte del sistema.

Objetivos realizados en común:

- Configuración de distintos *clusters,* formados por placas Raspberry PI 4, con el objetivo de ejecutar tanto las aplicaciones MPI bajo estudio como las simulaciones.

Objetivos realizados por Pablo R. Morer Olmos:

- Estudio y modificación de la biblioteca TraceLib.
- Diseño e implementación de una aplicación que permite la ejecución de un programa MPI en un entorno distribuido, formado por un conjunto de Raspberry Pi, y la consecuente representación gráfica de los resultados mediante trazas, que muestra una información de cada llamada MPI y Entrada/Salida.
- Diseño e implementación de una aplicación que permite capturar y registrar llamadas a funciones MPI y Entrada/Salida en un entorno distribuido real.

Objetivos realizados por Miguel Pérez de la Rubia:

- Diseño e implementación del sistema de gestión y simulación de entornos distribuidos.
- Sistema de monitorización de los nodos de cómputo activos en el *cluster*.

- Diseño de un planificador para optimizar el uso de recursos y aprovechar el paralelismo a la hora de lanzar simulaciones en el *cluster*.

1.2 Alcance y motivación

El alcance de este Trabajo de Fin de Grado es meramente académico. Se ha realizado para facilitar el estudio de aplicaciones MPI de una manera más accesible al alumno.

La motivación principal de este proyecto reside en acercar, de una manera más visual, el funcionamiento de un programa MPI en un entorno distribuido, así como mostrar cómo afectan los cambios en un sistema distribuido de manera más ágil, optimizando los tiempos de ejecución de las simulaciones. De esta forma se presenta la utilidad y la versatilidad de un sistema distribuido.

Actualmente, por problemas de seguridad, no se permite realizar ejecuciones de MPI en los laboratorios de la Facultad de Informática. Mediante la utilización de un *cluster* de Raspberry Pi similar al propuesto(figura 1) se puede resolver este problema. Sin embargo, la configuración y gestión de un *cluster* puede resultar compleja debido al manejo de permisos por distintos usuarios. Con este trabajo se permite facilitar este proceso.

1.3 Plan de Trabajo

Debido a que el trabajo ha sido realizado de manera independiente, en las figuras 2 y 3 se muestra el plan de trabajo que ha seguido cada uno de los integrantes del proyecto mediante diagramas de Gantt. En ambos casos, el plan de trabajo se ha completado en 44 semanas comprendidas entre octubre de 2020 y septiembre de 2021, donde se han llevado a cabo distintas etapas:

- Investigación y recogida de información.
- Planteamiento y diseño del proyecto.
- Instalación y configuración de entornos.
- Desarrollo e implementación.

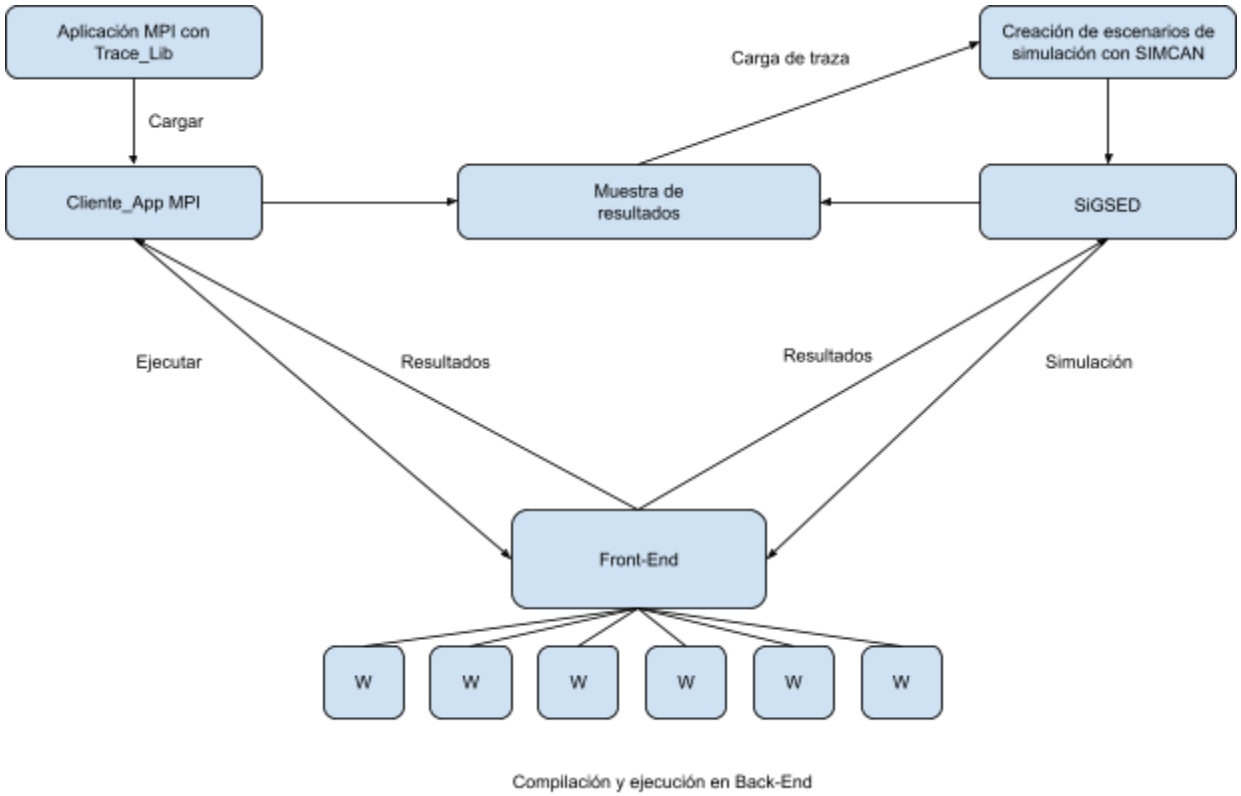


Figura 1. Esquema de la propuesta

Plan de trabajo de Pablo Morer

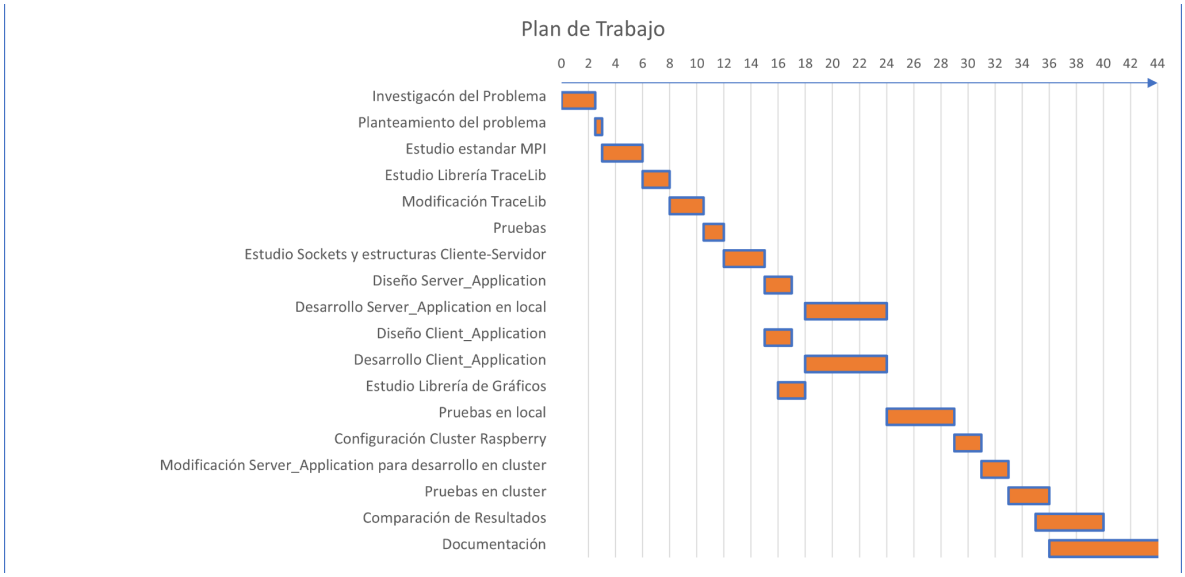


Figura 2. Plan de trabajo de Pablo Román Morer Olmos

Plan de trabajo de Miguel Pérez

Figura 3. Plan de trabajo de Miguel Pérez

1.4 Estructura del plan

Como se ha indicado anteriormente, el trabajo ha sido realizado por dos personas. Esto se ha visto reflejado no solo en el plan de trabajo sino también en el desarrollo de los capítulos. Los capítulos del 1 al 3 pertenecen a la parte común, y han sido escritos conjuntamente. Los capítulos del 4 al 6 han sido escritos y presentan el trabajo y la implementación de Pablo Morer, y los capítulos del 7 al 9 han sido escritos y muestran el trabajo y la implementación de Miguel Pérez.

Estos nueve capítulos se tratan siguiendo la siguiente estructura:

- Capítulos comunes:
    1. En el primer capítulo se presenta la introducción al proyecto, comentando los objetivos, la motivación, el alcance, la planificación y la estructura empleada.

2. El segundo capítulo describe los elementos utilizados para la implementación del sistema.
3. El tercer capítulo detalla los pasos a seguir para la configuración del *cluster* de Raspberry Pi.

- Capítulos de Pablo Morer:
    4. El cuarto capítulo contiene toda la información sobre la biblioteca para generar trazas de aplicaciones MPI y sus modificaciones.
    5. El quinto capítulo detalla el desarrollo de la parte del cliente en la aplicación de generación de trazas.
    6. El sexto capítulo detalla el desarrollo de la parte del servidor en la aplicación de generación de trazas.
- Capítulos de Miguel Pérez:
    7. El séptimo capítulo detalla la herramienta de simulación utilizada y la arquitectura diseñada para la gestión y simulación de trazas.
    8. El octavo capítulo detalla el funcionamiento y las interfaces de la parte cliente del sistema de gestión y simulación de entornos distribuidos.
    9. El noveno capítulo detalla el funcionamiento y las interfaces de la parte servidor del sistema de gestión y simulación de entornos distribuidos

1 - Introduction

This chapter details the project's main objectives, its scope, and the motivation for carrying it out. It also sets out the approach to the work involved for its realization.

1.1 - Objectives

The main objective of this Bachelor's thesis is to design and implement a methodology to analyze the performance of MPI applications executed in distributed environments. In order to achieve this objective, we propose the following secondary objectives.

Common Objectives:

- Configuration of two different Raspberry Pi clusters, where the MPI applications under study, and the simulation of these applications, are executed in the cluster.

Objectives made by Pablo R. Morer Olmos:

- Study and modification of the TraceLib library.
- Designing and implementing an application to generate traces of an MPI application in a distributed environment.
- Designing and implementing an application that allows the execution of an MPI program in a distributed environment and the graphical representation of the results obtained traces.

Objectives made by Miguel Pérez de la Rubia:

- Designing and implementing the system of management and simulation of distributed environments.
- The monitoring system in the cluster for the active computing nodes.
- Design of a scheduler to optimise available resources and take advantage when launching simulations.

The scope of this Bachelor's thesis is purely academic. It has been done to ease the study of MPI applications and make them more accessible for the student.

1.1 Scope and Motivation

The main motivation of this project is to design and implement a methodology that allows to study the performance of MPI applications in a distributed system, as well as show how the changes affect a distributed system in a more agile and efficient way, optimizing the execution time of the simulations. Thus, the usefulness and the versatility of a distributed system are presented.

Currently, due to security issues, MPI executions in the laboratories of the Computer Science Faculty are not allowed. By using a cluster of Raspberry Pi (see Figure 1), this problem can be alleviated. However, the configuration and management of the cluster can be complex due to the handling of permissions by different users. This project allows us to facilitate this process.

1.2 Work Plan

Since the work has been carried out independently, figures 2 and 3 show the work plan followed by each of the members of the project using Gantt charts. In both cases, the work plan was completed in 44 weeks between October 2020 and September 2021. Different stages have been carried out:

- Research and information gathering.
- Project approach and design.
- Installation and configuration of environments.
- Development and implementation.

Pablo Morer's work plan is depicted in figure 2, and Miguel Pérez's work plan is illustrated in figure 3.

1.3 Plan structure

As indicated in the previous point, the work has been carried out by two students, and this has been reflected not only in the work plan but also in the writing this final thesis.

Chapters 1-3 belong to the common part, and have been written jointly, chapters 4-6 have been written and drafted by Pablo Morer, and chapters 7-9 have been written and drafted by Miguel Pérez.

This work is structured as follows:

- Common Chapters:
    1. Chapter 1 details the introduction to the project, describing the objectives, motivation, scope, planning and structure used.
    2. Chapter 2 details the elements used for the implementation of the system.
    3. Chapter 3 details the steps to follow for the configuration of the Raspberry Pi cluster.
- Chapters by Pablo Morer:
    4. Chapter 6 details all the information about the library for generating MPI traces and their modifications.
    5. Chapter 5 details the development of the client-side of the MPI trace generation application.
    6. Chapter 6 details the development of the server-side of the MPI trace generation application.
- Chapters by Miguel Pérez:
    7. Chapter 7 details the simulation tool used and the architecture designed for trace management and simulation.
    8. Chapter 8 details the operation and interfaces of the client-side of the distributed environment management and simulation system.
    9. Chapter 9 details the operation and interfaces of the server-side of the distributed environment management and simulation system.

2 - Elementos del sistema

En este capítulo se describen los elementos que hay disponibles en el mercado que pueden ser utilizados para la configuración y despliegue del *cluster*, así como los que se han usado para ejecutar tanto las aplicaciones MPI como las simulaciones de las mismas. La figura 4 ilustra la arquitectura del *cluster,* donde se encuentra el tipo de Raspberry PI utilizado, además del switch encargado de la comunicación entre los dispositivos.

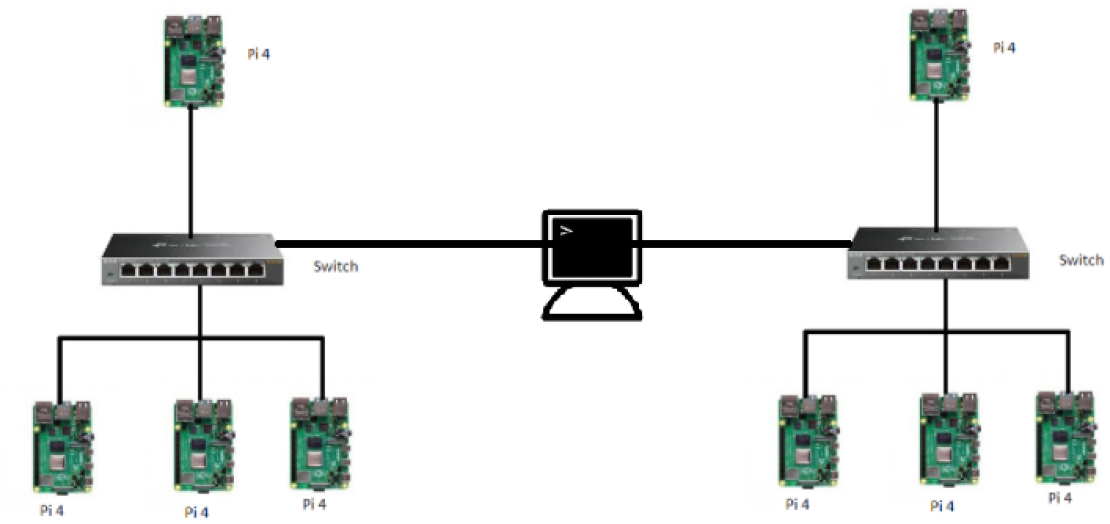


Figura 4. Esquema de la arquitectura del *cluster*

2.1 - Placas Raspberry Pi

El *cluster* está formado por un conjunto de placas Raspberry Pi, encargadas del trabajo de cómputo y del reparto de carga. La Raspberry Pi encargada del reparto de trabajo será única y en adelante será referida como front-end. Raspberry Pi ha contado con distintos modelos desde su aparición. A continuación se describen, cronológicamente a su orden de salida al mercado, las distintas características de cada uno de ellas:

Raspberry Pi 1: **Model A+ y Model B+**[1]. Es la primera versión de Raspberry Pi, lanzada al mercado en 2012. En la primera versión, modelo A+, la placa cuenta con 26 puertos GPIO, puertos de salida de video HDMI y RCA. Además, cuenta con una memoria RAM

[1] https://raspberry-projects.com/pi/category/pi-hardware/raspberry-pi-model-b-plus

de 256 Mb, así como un procesador Single-Core de 700 MHz junto con una gráfica Broadcom Videocore IV y slot para tarjetas SD. El modelo posterior, model B+, cuenta con 4 puertos USB, además, se le añade puerto Ethernet junto con un aumento de su memoria RAM a 512Mb. En este modelo se sustituye el slot para tarjetas SD por un soporte para tarjetas microSD, cambio que se mantendrá en las sucesivas versiones.

Raspberry Pi 2: **Model B**[2]. Esta es la segunda versión de Raspberry Pi, lanzada al mercado en 2014. Se encuentran diversos cambios con respecto a la Raspberry Pi 1. Entre los distintos cambios encontramos la adición de 40 pines GPIO, la supresión del puerto RCA, 1Gb de memoria RAM compartida con la gráfica y un nuevo procesador Quad Core de 900 MHz de frecuencia.

Raspberry Pi 3: Model A+[3]**, Model B**[4] **y Model B+**[5]. Esta es la tercera versión de Raspberry Pi, lanzada al mercado entre los años 2016 y 2018. De esta generación cabe destacar los modelos B y B+ por la mejora en los procesadores, llegando a los 1.4 GHz de frecuencia con un procesador Quad Core. En estos modelos también se progresa en cuanto a la conexión, incorporando 5GHz de conectividad inalámbrica y 300Mbits/s mediante el puerto Ethernet

Raspberry Pi 4: Model B[6]. Esta es la cuarta versión de Raspberry Pi, lanzada al mercado en el año 2019. Esta placa tiene varias versiones. Estas varían su memoria RAM disponible entre 2Gb, 4Gb y 8Gb. Al igual que en el modelo anterior, cuenta con puerto Gigabit Ethernet, el cual elimina la limitación de 300Mbps, y se actualizan los puertos USB por USB 3.0. Esta versión posee una mejora de procesador también, pasando a un procesador Quad-Core de 1,5 GHz.

Raspberry Pi: Pico . Anunciada en 2021, esta pequeña placa viene proporcionada de un procesador dual core ARM Cortex M0+ de 133 MHz, 2MB de almacenamiento integrado y 264KB de RAM. Su mayor característica es su reducido tamaño, 51x21mm.

[2] https://raspberry-projects.com/pi/category/pi-hardware/raspberry-pi-2-model-b
[3] https://raspberry-projects.com/pi/category/pi-hardware/raspberry-pi-3-model-a
[4] https://raspberry-projects.com/pi/category/pi-hardware/raspberry-pi-3-model-b
[5] https://raspberry-projects.com/pi/category/pi-hardware/raspberry-pi-3-model-b-pi-hardware
[6] https://raspberry-projects.com/pi/category/pi-hardware/raspberry-pi-4

Las placas usadas para el desarrollo del *cluster* han sido únicamente del modelo Raspberry Pi 4 Model B, puesto que son las proporcionadas por la Facultad de Informática.

2.2 - Placas utilizadas en el montaje del *cluster*

Puesto que para el desarrollo del *cluster* sólo se ha utilizado la Raspberry Pi 4 Model B (figura 5), el *cluster* formado es un *cluster* homogéneo. Para este proyecto se cuenta con 12 de estas placas en la configuración, que poseen 4GB de RAM.

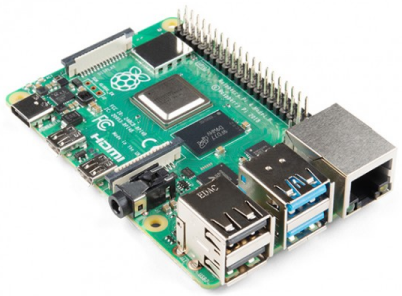


Figura 5. Raspberry Pi 4 Model B

Las especificaciones técnicas de este modelo son las siguientes:

- SoC: Broadcom BCM2711.
- CPU: 1.5GHz 64-bit quad-core Cortex-A72.
- RAM: 4 GB LPDDR4 SDRAM.
- Ethernet Gigabit Ethernet.
- 802.11ac LAN inalámbrica y Bluetooth 5.0.
- USB 2 x Conector USB 2.0.
- USB 2 x Conector USB 3.0.

2.2.1 - Montaje de las placas

Las placas se encuentran organizadas en dos carcasas en vertical, una con capacidad para 8 Raspberry Pi y la otra con capacidad para 4, consiguiendo así

facilitar el transporte de estas. El modelo de carcasas utilizado es "Joy-It Tower-Case para Raspberry Pi" con capacidad para 4 y 8 placas Raspberry Pi (figura 6).

Figura 6. Carcasa Joy-It Tower-Case para Raspberry Pi

2.3 - Switch

El switch se encarga de la conexión entre las distintas placas. Contamos con un switch de ocho puertos para cada *cluster*. Los modelos utilizados son el Switch TP-Link TL-SG108E (figura 7) y el 8-Port Gigabit Desktop Switch DGS-1008D (figura 8).

Figura 7. Switch TP-Link TL-SG108E

Figura 8. 8-Port Gigabit Desktop Switch DGS-1008D

2.4 - Almacenamiento

Como se menciona en la sección 2.1, las placas Raspberry Pi Model 4 carecen de almacenamiento integrado. Por ello, se utilizan tarjetas microSD de 16Gb para desplegar el sistema operativo Raspberry Pi OS Lite en cada placa del *cluster* (figura 9).

Figura 9. Tarjeta microSD de 16GB

2.5 - Alimentación

Para la alimentación de las Raspberry Pi 4 han sido utilizados los cargadores oficiales USB tipo C de 5,1V, 3A(figura 10).

Figura 10. Alimentación Raspberry Pi 4

3 - Configuración del *cluster*

En este capítulo describen las distintas configuraciones del *cluster* para el desarrollo del proyecto. Una parte del *cluster* está dirigida a la obtención de trazas de las aplicaciones y la otra a la gestión y simulación de los distintos entornos.

3.1 - Configuración General

Aunque en el *cluster* va a tener dos objetivos diferentes, hay una configuración común que se realiza en ambos.

3.1.1 - Configuración front-end

Debido a la necesidad de tener una instancia para la comunicación y acceso al *cluster,* se ha decidido emplear una de las placas para realizar el rol de front-end del sistema. Desde esta placa se produce un intercambio de datos con la parte cliente y se realiza el reparto de trabajo entre los distintos nodos de cómputo.

Para el uso del front-end es necesaria la instalación de un sistema operativo, que en este caso será Raspberry Pi OS Lite(32 bit)[7], un sistema operativo basado en Debian. Este sistema operativo carece de interfaz gráfica, al ser la versión Lite. Para la instalación de Raspberry Pi OS Lite(32 bit) se ha utilizado el software *Raspberry Pi Image*[8] proporcionado por Raspberry Pi para la creación de imágenes de sistemas operativos sobre dispositivos de almacenamiento.

3.1.2 - Instalación Java

El primer paso para llevar a cabo la configuración del front-end será instalar la última versión de Java disponible. Se ha optado por utilizar Java en el desarrollo del proyecto debido a que facilita el uso de bibliotecas como la de sockets y threads, necesarias para la comunicación con el cliente. A su vez, también se ha tenido en cuenta la portabilidad del lenguaje, ya que es un lenguaje multiplataforma.

Para la instalación de Java en la Raspberry Pi se han de seguir los siguientes pasos:

[7] https://www.raspberrypi.org/software/operating-systems/
[8] https://www.raspberrypi.org/software/

```
sudo apt-get update
sudo apt-get install default-jdk
```

Para comprobar que la instalación se ha realizado correctamente se ejecuta:

```
java -version
```

La salida de este comando debería ser similar a la siguiente:

```
openjdk version "11.0.5" 2019-10-15
 OpenJDK Runtime Environment (build
11.0.5+10-post-Raspbian-1deb10u1)
 OpenJDK Server VM (build 11.0.5+10-post-Raspbian-1deb10u1, mixed
mode)
```

3.1.3 - Configuración de Red

Para la configuración de red, se ha decidido utilizar direcciones IP estáticas, ya que esto facilita el acceso al servidor. Además, las IPs también serán estáticas dentro del *cluster,* así será más fácil la asignación de trabajo a cada uno de los nodos de cómputo.

Al front-end se le asignan dos IP, una IP será asignada para la conexión con el exterior del *cluster* y una IP asignada para la identificación dentro del *cluster*. La IP usada para la identificación con el exterior del *cluster* es la 192.168.0.105 y la usada dentro del *cluster* es la 169.254.12.1 Para ello editamos el fichero /etc/network/interfaces, configurándolo de la siguiente forma:

```
auto lo
iface lo inet loopback

auto eht0
iface eth0 inet static
address 192.168.0.105
netmask 255.255.255.0
network 192.168.0.1
```

```
auto eth0:0
iface eth0:0 inet static
address 169.254.12.1
netmask 255.255.255.0
network 169.254.12.0
```

Para asegurarnos del correcto funcionamiento, reiniciamos el dispositivo con el comando `reboot` o con `/etc/init.d/networking restart`. Tras el reinicio, se debe ejecutar el comando `ifconfig`, que indicará los datos de las interfaces de red mostrando la IP establecida en el fichero.

3.1.4 - Configuración de Nodos

Para la configuración de los nodos se instalará en las tarjetas SD respectivas a cada placa el sistema operativo Operativo Raspberry Pi OS Lite. De la misma forma que se ha realizado en el front-end, sección 3.1.1.

Una vez que se haya instalado el sistema operativo será necesario configurar cada nodo con su propia IP, como se especifica en el apartado 3.1.3. Puesto que el front-end emplea la IP 169.254.12.1 para los nodos se utilizarán las direcciones consecutivas, dentro del rango 169.254.12.2-169.254.12.13.

3.1.5 - Instalación de MPI

La primera herramienta necesaria para el funcionamiento de la metodología desarrollada es MPI[9], que corresponde con las siglas *Message Passing Interface*. MPI es un estándar que se utiliza en la paralelización de programas, por lo que nos permitirá utilizar los nodos del cluster de manera simultánea durante la ejecución de las aplicaciones MPI. Para esta tarea se ha seleccionado la distribución de MPICH[10], una implementación estándar MPI de código abierto. En este caso la versión que se ha instalado es la 3.3.

Para instalación de MPICH hay ejecutar los siguientes comandos:

[9] https://es.wikipedia.org/wiki/Interfaz_de_Paso_de_Mensajes
[10] https://www.mpich.org/

```
sudo apt-get update
sudo apt-get install mpich
```

3.1.6 - Arranque sin HDMI

El sistema operativo instalado por defecto no permite el arranque sin que se detecte un cable HDMI conectado. Para solventar este problema y poder arrancar las Raspberry Pi sin un dispositivo de salida gráfica hay que modificar el archivo `/boot/config.txt` en cada una de ellas, añadiendo la siguiente línea:

```
hdmi_force_hotplug = 1
```

3.2 - Instalación de software adicional

En esta sección, se describen de forma detallada las distintas configuraciones del *cluster* utilizadas para el desarrollo del sistema de gestión y simulación de entornos distribuidos, en adelante *SiGSED*. Una parte se dedica a las especificaciones de la Raspberry Pi utilizada como front-end y otra a las especificaciones de las Raspberry Pi utilizadas como nodos de cómputo. Por otro lado, se describe la interacción dentro del *cluster* con el front-end para llevar a cabo la simulación de los distintos escenarios recibidos.

3.2.1 - Instalación de RSH en el front-end

RSH se corresponde con las siglas en inglés de Remote Shell. Para poder ejecutar las simulaciones correctamente en cada una de las Raspberry Pi hay que instalar en el front-end la funcionalidad de rsh-server. La funcionalidad principal es poder ejecutar comandos de manera remota. De esta forma, se pueden ejecutar instrucciones remotamente desde el front-end.

Para la instalación de RSH se ejecuta el siguiente comando:

```
sudo apt-get install rsh-server
```

A continuación se modifica el archivo `/etc/hosts.equiv` para permitir que los otros dispositivos de la red puedan conectarse. Se añadirá un `+` al final del archivo para indicar que todos las máquinas de la red pueden conectarse.

3.2.2 - Instalación de RSH en los workers

Para poder utilizar correctamente RSH se necesita instalar en los nodos de cómputo rsh-client.

En primer lugar se ha de descargar el cliente RSH ejecutando el siguiente comando en cada uno de los nodos:

```
sudo apt-get install rsh-client
```

A continuación se tiene que modificar el archivo /etc/hosts.equiv de cada uno de los nodos, añadiendo al final del archivo un + para permitir el acceso al dispositivo a los otros nodos de la red.

3.2.3 - Instalación de Simcan en el cluster

Para el funcionamiento del SiGSED es necesaria la instalación del simulador de entornos distribuidos Simcan[11]. Para su instalación es necesario tener OMNeT++[12] e INET[13] disponibles en todas las máquinas.

OMNeT++ es una biblioteca de simulación modular, desarrollada en C++, principalmente utilizada para desarrollo de simuladores de red. INET es una biblioteca de modelos de código abierto para el entorno de simulación OMNeT ++ . Proporciona protocolos, agentes y otros modelos para investigadores y estudiantes que trabajan con redes de comunicación.

3.2.3.1 - Instalación de OMNeT++

El primer paso para la instalación de Simcan en el *cluster* será descargar OMNeT++ y descomprimirlo mediante las instrucciones:

[11] http://antares.sip.ucm.es/cana/simcan/about.html
[12] https://omnetpp.org/
[13] https://inet.omnetpp.org/

```
wget
https://github.com/omnetpp/omnetpp/releases/download/omnetpp-6.0pre
11/omnetpp-6.0pre11-src-linux.tgz
$ tar xvfz omnetpp-6.0pre11-src-linux.tg
```

A continuación se instalan las dependencias de python:

```
python3 -m pip install --user --upgrade numpy pandas matplotlib
scipy seaborn posix_ipc
```

Añadimos la ruta donde se encuentra el OMNeT++ al path de la máquina. Hay dos formas de hacerlo:

- Solo en la sesión

```
$ export PATH=$HOME/omnetpp-6.0pre11/bin:$PATH
```

- De manera permanente, editando el archivo .bashrc:

```
$ PATH=$HOME/omnetpp-6.0pre11/bin:$PATH
```

Para que OMNeT++ 6 pueda ser compilado hay que tener una versión de C++ mayor o igual a la 14, lo que significa que la versión del gcc debe ser superior o igual a 9. Se utilizará la versión 10.1.0 para el desarrollo y los pasos a seguir son los siguientes:

```
sudo apt install git
git                                                              clone
https://bitbucket.org/sol_prog/raspberry-pi-gcc-binary.git
cd raspberry-pi-gcc-binary
tar -xjvf gcc-10.1.0-armhf-raspbian.tar.bz2
sudo mv gcc-10.1.0 /opt
cd ..
rm -rf raspberry-pi-gcc-binary
echo 'export PATH=/opt/gcc-10.1.0/bin:$PATH' >> ~/.bashrc
echo  'export  LD_LIBRARY_PATH=/opt/gcc-10.1.0/lib:$LD_LIBRARY_PATH'
>> ~/.bashrc
. ~/.bashrc
sudo ln -s /usr/include/arm-linux-gnueabihf/sys /usr/include/sys
sudo ln -s /usr/include/arm-linux-gnueabihf/bits /usr/include/bits
sudo ln -s /usr/include/arm-linux-gnueabihf/gnu /usr/include/gnu
```

```
sudo ln -s /usr/include/arm-linux-gnueabihf/asm /usr/include/asm
sudo ln -s /usr/lib/arm-linux-gnueabihf/crti.o /usr/lib/crti.o
sudo ln -s /usr/lib/arm-linux-gnueabihf/crt1.o /usr/lib/crt1.o
sudo ln -s /usr/lib/arm-linux-gnueabihf/crtn.o /usr/lib/crtn.o
```

Para comprobar que se ha instalado correctamente se puede ejecutar `gcc-10.1 --version`

Después de esto se modifica el archivo `configure.user` en el directorio de OMNeT++ para configurar los flags WITH_QTENV, WITH_OSG y WITH_OSGEARTH para su ejecución sin interfaz gráfica

```
WITH_QTENV=no
WITH_OSG=no
WITH_OSGEARTH=no
```

A continuación simplemente se ejecuta el comando de configuración y de compilación:

```
 ./configure
make
```

3.2.3.2 - Instalación de INET

El primer paso para la instalación de INET es descargar la versión 4.3.2

```
wget
https://github.com/inet-framework/inet/releases/download/v4.3.2/ine
t-4.3.2-src.tgz
tar -xvf inet-4.3.2-src.tgz
```

Seguidamente se compila el INET:

```
which omnetpp || echo "Setup Omnet++ environment first"
# Comprobar que Omnet está en el path
source setenv
make makefiles
make -j $(nproc) MODE=debug
```

3.2.3.3 - Instalación de SIMCAN en el cluster

La instalación de SIMCAN en el *cluster* se hará sin interfaz gráfica para las simulaciones, ya que esta no es necesaria.

Antes de compilar SIMCAN hay que copiar la carpeta en el directorio que se vaya a instalar y después hacer un clean del proyecto.

```
make cleanall
```

A continuación es necesario añadir la ruta a INET en la variable INET_PROJ, encargada de guardar la ruta en la que se encuentra instalado INET, en el Makefile:

```
nano Makefile
INET_PROJ=~/inet4.3/
make -j $(nproc) MODE=debug
```

3.2.4 - Ejecución de comandos entre el front-end y los workers

Para la ejecución de comandos remotos e intercambio de ficheros entre los nodos del *cluster* se utiliza RSH, explicada en el punto 3.1.2, y SCP.

SCP se corresponde con las siglas en inglés Secure Copy Protocol y es un medio de transferencia segura de archivos informáticos entre un host local y otro remoto o entre dos hosts remotos mediante el uso del protocolo Secure Shell (SSH). Para su uso, solo es necesario la activación del protocolo SSH en las máquinas.

Se utilizará RSH desde el front-end hacia los nodos de cómputo para lanzar las instrucciones y scripts correspondientes. En general, serán instrucciones para conocer el estado de cada nodo de cómputo (detallado en la sección 8.3), o para lanzar a ejecución los archivos del simulador mediante un script (detallado en la sección 8.4).

La estructura de un comando RSH es la siguiente:

```
rsh <options> -l <user> <ip_dest/hostname> command
```

SCP se utilizará para el intercambio de ficheros de simulación entre el front-end y el nodo de cómputo al que corresponda en ese momento los ficheros necesarios para la ejecución del simulador.

La estructura de un comando SCP es la siguiente:

```
scp [source files] [user]@[host]:[path]
```

4 - TraceLib: Biblioteca para la generación de trazas de aplicaciones MPI

En este capítulo se presenta la biblioteca *TraceLib*, cuya finalidad es registrar llamadas MPI. Cabe remarcar que la librería fue diseñada y desarrollada por el alumno de la Facultad de Informática de la UCM, Bryan Raúl Vaca Vargas para su Trabajo de Fin de Grado[14].

Con el fin de implementar mejoras necesarias para el desarrollo del Trabajo de Fin de Grado, ha sido de vital importancia analizar la biblioteca *TraceLib* de forma detallada y realizar distintas modificaciones que consiguen eliminar ciertas limitaciones que se indican en este capítulo.

4.1 - Biblioteca *TraceLib*

La biblioteca *TraceLib* se encarga de reconocer y registrar un conjunto de llamadas a funciones MPI y de Entrada/Salida invocadas por cualquier aplicación MPI.

El uso de esta biblioteca permite obtener una lista de las llamadas a funciones invocadas en una aplicación MPI ordenadas cronológicamente, así como los tiempos en los cuales fueron invocadas. Esta funcionalidad potencia la reproducibilidad de los experimentos, permitiendo reproducir el comportamiento de una aplicación ejecutando las instrucciones de una traza generada.

4.2 - Estructura *taskInfo* para el registro de tareas

Entendemos por tarea cada una de las llamadas a funciones, tanto MPI como de E/S, que realiza una aplicación MPI en su ejecución.

Podemos representar una tarea en base a las marcas temporales de su inicio y su final. Para ello, se ha utilizado una estructura que contiene toda la información necesaria para su registro.

[14] https://eprints.ucm.es/id/eprint/50220/1/027.pdf

```
struct taskInfo{
        char name[9];
        char *pathname;
        double timeStamp;
        int data_size;
        int offset;
        int my_rank;
        int rank_dst_src;
        int id;
        struct taskInfo *next;
};
```

Donde:

- *name* representa el nombre de la tarea.
- *pathname* es la ruta del fichero a utilizar en las llamadas de E/S.
- *timeStamp* es la marca temporal en la que ocurrió el inicio, o el final, de una tarea.
- *data_size* representa el tamaño de los datos leídos, escritos o enviados por la llamada actual.
- *offset* representa el desplazamiento en el fichero cuando la tarea corresponde a una llamada a *pwrite()* o *pread()*.
- *my_rank* corresponde al *rank del proceso que invoca la llamada.*
- *rank_dst_src* corresponde al *rank* del proceso origen o destino, según la llamada MPI.
- *id* es el identificador único asociado a cada par de *taskInfo* que forman el inicio y el final de una misma tarea.
- *next* representa el puntero a la siguiente *tarea.*

4.3 - Funciones implementadas en la biblioteca TraceLib

La biblioteca sigue una estructura uniforme para describir las llamadas a las distintas funciones. Principalmente, recordamos que se registran todas las llamadas MPI y E/S que se puedan utilizar en cualquier aplicación MPI. Todas las llamadas a las funciones instanciadas en la biblioteca *TraceLib* siguen la misma sintaxis:

```
typeReturn nameFunction_trace([parámetros_de_entrada], const int
```

```
Rank);
```

Donde:

- *typeReturn* es el tipo de valor devuelto por la función. Coincide con el tipo de la llamada MPI o E/S original.
- *nameFunction_trace* es el nombre de la llamada de la biblioteca. El nombre se forma a partir del nombre de la función MPI o E/S original y el sufijo *_trace*.
- *parametros_de_entrada* son los diversos parámetros que se utilizan en la llamada original MPI o E/S invocada.

Un ejemplo de esta sintaxis es:

```
int creat_trace(const char *path, mode_t mode, const int myRank);
```

En función del tipo de llamada que realicen las funciones, podemos agruparlos en tres clases: funciones de gestión de llamadas, llamadas de E/S, llamadas MPI.

4.3.1 - Funciones de gestión de llamadas

Esta sección muestra las funciones relativas a la gestión de llamadas que realizan los distintos procesos. En esencia, estas funciones se encargan de la información necesaria para generar la traza.

```
void addTaskInfo(char *name, struct timeval *tvIni, struct timeval
*tvEnd, int rankSource, int rankDest, int dataSize);

void addTaskFileInfo(const char *name, const char *path, struct
timeval *tvIni, struct timeval *tvEnd, int my_rank, int
rank_dst_src, int dataSize, int offset);
```

Por cada llamada invocada de la biblioteca *TraceLib*, se produce una llamada a *addTaskInfo()* o una llamada a *addTaskFileInfo()* (dependiendo si gestiona un fichero o no). De esta forma, mediante los distintos parámetros de entrada se crean dos estructuras *taskInfo()* (las cuales se detallan en la Sección 4.2), una encargada de guardar la información correspondiente al inicio de la tarea, y otra encargada de guardar la información correspondiente a la finalización de la tarea.

```
void traceInit();
```

La función *traceInit()* se invoca con la ejecución de la llamada a *MPI_Init_trace()*. Inicializa y reserva el espacio necesario para las distintas estructuras de datos pertenecientes a una tarea, inicializa el contador de identificadores para las tareas y registra la marca temporal inicial del proceso que la ejecuta.

```
void traceEnd();
void writeListTaskInfo(FILE *file);
void showListTaskInfo(struct taskInfo *fun);
```

La función *traceEnd()* se invoca con la ejecución de la llamada a *MPI_Finalize_trace()*. Genera un fichero de registro donde se muestran todas las llamadas asociadas a un mismo *rank* capturadas previamente. *TraceEnd()* invoca la función *writeListTaskInfo(),* que se encarga de recopilar la lista de tareas y volcarlas al fichero de registro. Para la depuración se utiliza la función *showListTaskInfo(),* que muestra por pantalla las tareas registradas en cada proceso.

```
char * obtainFileName(const int fd);
```

La función *obtainFileName()* obtiene el nombre de un archivo a partir de su descriptor de fichero. Esta función se detalla en la sección 4.5 .

4.3.2 - *Llamadas de E/S*

Esta sección muestra las funciones utilizadas para capturar y gestionar llamadas de E/S. Todas las funciones que se muestran a continuación invocan sus funciones respectivas de la biblioteca estándar de C.

```
int creat_trace(const char *path, mode_t mode, const int myRank);
```

La función *creat_trace()* crea un nuevo fichero o reescribe uno ya existente según el modo indicado. Esta función se ejecuta cada invocación a la llamada *create()* en la aplicación MPI.

```
int open_trace(const char *pathn, int flags, const int myRank);
```

La función *open_trace()* abre un fichero. Esta función se ejecuta cada invocación a la llamada *open()* en la aplicación MPI.

```
FILE *fopen_trace(const char *path, const char *mode, const int
myRank);
```

La función *fopen_trace()* abre un fichero con el modo indicado. Esta función se ejecuta cada invocación a la llamada *fopen()* en la aplicación MPI.

```
int close_trace(int fd, const int myRank);
```

La función *close_trace()* cierra el descriptor de fichero indicado. Esta función se ejecuta cada invocación a la llamada *close()* en la aplicación MPI.

```
int fclose_trace(FILE *stream, const int myRank);
```

La función *fclose_trace()* cierra el fichero indicado por el puntero a fichero. Esta función se ejecuta cada invocación a la llamada *fclose()* en la aplicación MPI.

```
ssize_t read_trace(int fd, void *buf, size_t count, const int
myRank);
```

La función *read_trace()* lee un número de bytes desde un descriptor de fichero y lo guarda en el buffer indicado. Esta función se ejecuta cada invocación a la llamada *read()* en la aplicación MPI.

```
ssize_t pread_trace(int fd, void *buf, size_t count, off_t offset,
const int myRank);
```

La función *pread_trace()* lee desde una posición indicada un número de bytes desde un descriptor de fichero y lo guarda en el buffer indicado. Esta función se ejecuta cada invocación a la llamada *pread()* en la aplicación MPI.

```
ssize_t fread_trace(void *ptr, size_t size, size_t nmemb, FILE
*stream, const int myRank);
```

La función *fread_trace()* lee los datos de un fichero dado en el array apuntado por un descriptor de fichero. Esta función se ejecuta cada invocación a la llamada *fread()* en la aplicación MPI.

```
ssize_t write_trace(int fd, void *buffer, size_t count, const int
myRank);
```

La función *write_trace()* escribe un número de bytes desde un buffer y lo guarda en el descriptor de fichero indicado. Esta función se ejecuta cada invocación a la llamada *write()* en la aplicación MPI.

```
ssize_t fwrite_trace(const void *ptr, size_t size, size_t nmemb, FILE
*stream, const int myRank);
```

La función *fwrite_trace()* escribe los datos de un buffer dado en el array apuntado por un descriptor de fichero. Esta función se ejecuta cada invocación a la llamada *fwrite()* en la aplicación MPI.

4.3.3 - Llamadas MPI

Esta sección muestra las funciones necesarias para capturar y gestionar un subconjunto de llamadas del estándar MPI. Todas las funciones que se muestran a continuación invocan sus funciones respectivas de la biblioteca MPI.

```
int MPI_Init_trace(int *argc, char ***argv);
```

La función *MPI_Init_trace()* inicializa un entorno de ejecución MPI. Esta función se ejecuta cada invocación a la llamada *MPI_Init()* en la aplicación MPI.

```
int MPI_Comm_size_trace(MPI_Comm comm, int * size);
```

La función *MPI_Comm_size_trace()* determina el tamaño de un grupo de procesos asociados a un comunicador. Esta función se ejecuta cada invocación a la llamada *MPI_Comm_size()* en la aplicación MPI.

```
int MPI_Comm_rank_trace(MPI_Comm comm, int * rank);
```

La función *MPI_Comm_rank_trace()* se encarga de asociar un número *rank* único a cada proceso asociado a un comunicador. Esta función se ejecuta cada invocación a la llamada *MPI_Comm_rank()* en la aplicación MPI.

```
int MPI_Finalize_trace();
```

La función *MPI_Finalize_trace()* finaliza un entorno de ejecución MPI. Esta función se ejecuta cada invocación a la llamada *MPI_Finalize()* en la aplicación MPI.

```
int MPI_Send_trace(const void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, const int myRank);
```

La función *MPI_Send_trace()* se encarga de realizar un mensaje bloqueante de un proceso a otro proceso. Esta función se ejecuta cada invocación a la llamada *MPI_Send()* en la aplicación MPI.

```
int MPI_Recv_trace(void *buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Status *status, const int myRank)
```

La función *MPI_Recv_trace()* se encarga de recibir un mensaje bloqueante de un proceso a otro proceso. Esta función se ejecuta cada invocación a la llamada *MPI_Recv()* en la aplicación MPI.

```
int MPI_Scatter_trace(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm, const int myRank);
```

La función *MPI_Scatter_trace()* se encarga de enviar datos desde un proceso a todos los demás procesos de un mismo comunicador. Esta función se ejecuta cada invocación a la llamada *MPI_Scatter()* en la aplicación MPI.

```
int MPI_Gather_trace(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm, const int myRank);
```

La función *M.PI_Gather_trace()* se encarga de agrupar un conjunto de valores de un grupo de procesos. Esta función se ejecuta cada invocación a la llamada *MPI_Gather()* en la aplicación MPI.

```
int MPI_Bcast_trace(void *buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm, const int myRank);
```

La función *MPI_Bcast_trace()* se encarga de transmitir un mensaje desde un proceso con el rank *root* a todos los demás procesos con el mismo comunicador. Esta función se ejecuta cada invocación a la llamada *MPI_Bcast()* en la aplicación MPI.

4.4 - Modificación de la biblioteca TraceLib

La biblioteca *TraceLib* inicialmente se desarrolló bajo el sistema operativo macOS. Al no disponer de ningún dispositivo con este sistema operativo, se produjeron ciertas limitaciones a la hora de compilar y ejecutar funciones pertenecientes a esta biblioteca.

En una primera instancia, las desarrollo en local del proyecto se iba a desarrollar bajo el sistema operativo Windows 10, cuya arquitectura basada en el núcleo NT 10.0 es distinta a la arquitectura del sistema operativo macOS basada en el núcleo UNIX.

Esta distinción en núcleos desencadenó una serie de inconvenientes en las llamadas a sistema que se realizan en la biblioteca *TraceLib*, lo que no permite la correcta compilación de la biblioteca, y con ello, la imposibilidad de hacer pruebas en un entorno local.

Este problema de diferencias entre Windows 10 y macOS se intentó solucionar instalando Cygwin64[15], un entorno que proporciona compatibilidad a los sistemas Unix en Microsoft Windows. Cygwin64 consiste en emular una colección de herramientas que aportan una apariencia similar a Linux.

Una vez terminada la instalación y la configuración de Cygwin64, el rendimiento y la eficiencia brindada por este entorno a la hora de realizar ciertas operaciones básicas del sistema (como la creación de ficheros o la navegación entre directorios), no eran suficientes para el desarrollo de este proyecto. Por ello, se decidió por descartar Cygwin64 como entorno de desarrollo.

[15] https://cygwin.com/install.html

Una segunda solución fue utilizar un sistema operativo libre que comparta el núcleo basado en UNIX. Se optó por el sistema operativo Ubuntu, una distribución de Linux que puede utilizarse tanto en ordenadores como en servidores.

Aun teniendo en común un núcleo basado en el sistema UNIX, entre macOS y Ubuntu existe una diferencia en el sistema de ficheros, donde UNIX utiliza UFS frente al sistema APFS de macOS.

Este inconveniente entre sistemas de ficheros se presentó al compilar funciones como *read_trace()* o *write_trace(),* comentadas anteriormente (sección 4.3.2). Tras comparar el manual de llamadas a sistemas de Linux y el manual de llamadas a sistemas de macOs, se confirmó que, aún compartiendo núcleo basado en UNIX, no comparten todas sus funcionalidades y directivas.

El principal problema se encontraba a la hora de invocar a la llamada al sistema *fnctl()*, la cual mediante la directiva *F_GETPATH* obtiene la ruta al descriptor de fichero.

La utilización de esta llamada es indispensable para obtener el nombre de un fichero a partir de su descriptor de fichero, y así, poder realizar operaciones de lectura y escritura de ficheros. Pero, la directiva *F_GETPATH del sistema operativo macOS* no se encuentra en el sistema UNIX, por lo que la modificación realizada en esta biblioteca consiste en la implementación de una función capaz de reemplazar la directiva *F_GETPATH*.

Esta modificación permite una mejora a la hora de utilizar la biblioteca *TraceLib* en distintas plataformas del sistema UNIX. La implementación y el uso de esta nueva función que llamaremos *obtainFileName()* se describe en las secciones 4.5 y 4.6 .

Además, al ser el sistema operativo de las Raspberry Pi otra distribución de Linux, esta versión de la biblioteca será la utilizada por el servidor en el entorno distribuido.

4.5 - Función obtainFileName

Esta nueva función forma parte de la biblioteca que gestiona las llamadas de los procesos a cada una de las funciones MPI y E/S. La función *obtainFileName()* obtiene el nombre de un archivo a partir de su descriptor de fichero, donde *fd* es el descriptor del fichero del que se obtiene el nombre.

4.5.1 - *Implementación obtainFileName: macOS*

La implementación de la función *obtainFileName* en macOS es la siguiente:

```
char * obtainFileName(const int fd){
    char *path,*file, *aux;
    path = malloc (100 * sizeof(char));
    aux = malloc (100 * sizeof(char));
    file = malloc (100 * sizeof(char));
    if(fcntl(fd,F_GETPATH,path) != -1){
      while( (aux = strsep(&path, "/")) != NULL){
            strcpy(file,aux);
      }
    }
    return file;
}
```

La implementación de *obtainFileName()* para el sistema operativo macOS utiliza la función *fcntl(),* que proporciona control sobre un descriptor de fichero. Pasando por parámetro la primitiva *F_GETPATH,* se consigue la ruta hacia el fichero buscado. Seguidamente, mediante la función estándar de C *strcpy(),* se obtiene el nombre del fichero.

4.5.2 - *Implementación obtainFileName: Ubuntu 20.04*

La implementación de la función *obtainFileName* en Ubuntu 20.04 es la siguiente:

```
char * obtainFileName(const int fd){
     char * linkname;
     ssize_t r;
     linkname = malloc(100 * sizeof(char));
      char fdS[5];
     if(linkname == NULL)
          exit(EXIT_FAILURE);
     sprintf(fdS,"%d",fd);
     char path[100] = "/proc/self/fd/";
     strcat(path,fdS);
     r = readlink(path,linkname, 100*sizeof(char));
     if(r < 0)
         exit(EXIT_FAILURE);
    return linkname;
 }
```

Esta función es la encargada de obtener el nombre de fichero a partir de un descriptor de fichero recibido por parámetro de entrada de tipo int. Este parámetro se convierte a una cadena de caracteres mediante la función *sprintf()*.

Puesto que los descriptores de ficheros abiertos en un sistema Linux se encuentran en la ruta "*/proc/self/fd/*", únicamente hay que unir esta cadena de caracteres con la obtenida anteriormente, que determina el descriptor de fichero correspondiente al nombre de fichero que se busca obtener. Para ello se utiliza la función interna de Linux *readlink()*.

La función *readlink()* coloca el enlace simbólico que se encuentra en una ruta dada en un buffer de tipo *char*. Este buffer será el nombre del fichero que buscamos. Una vez completada con éxito, *readlink()* devolverá el número de bytes que ocupa el buffer. La signatura de la función *readlink()* se presenta a continuación.

```
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
```

Donde:

- *path* es la ruta al descriptor de fichero abierto del cual queremos obtener el nombre.
- *buf es el buffer donde vamos a guardar el nombre de fichero.*
- *bufsiz es el tamaño del buffer donde vamos a guardar el nombre del fichero.*

4.6 - Uso de la función obtainFileName

Una vez hallada la forma de sustituir de forma efectiva la función *fcntl()* que contiene la directiva *F_GETPATH,* mediante la función *obtainFileName()*, es necesario reemplazarlas dentro del código de la biblioteca *TraceLib*.

Como se menciona al principio del capítulo, la función *obtainFileName()* se usa en funciones que usen las llamadas de lectura o escritura. Estas funciones se pueden dividir en dos tipos según el argumento que reciba por parámetro. Este argumento puede ser un puntero a fichero (funciones *fread_trace() y fwrite_trace())* o el descriptor de fichero (funciones *read_trace() y write_trace())*.

4.6.1 - Paso de parámetro: puntero a fichero

Las funciones que reciben un puntero a fichero por parámetro son aquellas que utilizan la función estándar de C *fileno* para invocar la llamada a *obtainFileName()*. La función *fileno* permite acceder al descriptor de fichero de un fichero dado y así, poder usar este descriptor de fichero para invocar la llamada *obtainFileName()*. Por ejemplo en funciones como *fread_trace()* o *fwrite_trace()*.

```
ssize_t fread_trace(void *ptr, size_t size, size_t nmemb, FILE
*stream, const int myRank){
      struct timeval tvIni,tvEnd;
      gettimeofday(&tvIni, NULL);
       ssize_t sz = fread(ptr,size,nmemb,stream);
      gettimeofday(&tvEnd, NULL);
      char *file;
      file = malloc (100 * sizeof(char));
      file = obtainFileName(fileno(stream));
      addTaskFileInfo(FRE,file,&tvIni,&tvEnd,myRank,-1,(int) size,-1);
      return sz;
 }
```

4.6.2 - Paso de parámetro: descriptor de fichero

Las funciones que reciben un descriptor de fichero por parámetro son aquellas que utilizan ese descriptor de fichero para invocar la a llamada *obtainFileName()*. Por ejemplo las funciones *read_trace()* y *write_trace()*.

```
ssize_t read_trace(int fd, void *buf, size_t count, const int myRank){
      struct timeval tvIni,tvEnd;
      gettimeofday(&tvIni, NULL);
      ssize_t sz = read(fd,buf,count);
      gettimeofday(&tvEnd, NULL);
      char *file;
      file = malloc (100 * sizeof(char));
      file = obtainFileName(fd);
      addTaskFileInfo(READ,file,&tvIni,&tvEnd,myRank,-1,(int)count,-1);
      return sz;
}
```

5 - Obtención de trazas de aplicaciones MPI en entornos distribuidos. Parte Cliente.

Este capítulo describe el funcionamiento de la aplicación llamada *Client_Application*.

La aplicación *Client_Application* pretende facilitar el estudio del rendimiento y la eficacia de aplicaciones MPI ejecutadas en un cluster de Raspberry Pi. Este objetivo se consigue mediante gráficas que representan el proceso de ejecución.

Client_Application es una aplicación distribuida que se encarga de recopilar la información necesaria para procesar una aplicación MPI, conectarse al servidor de Raspberries, enviar la información necesaria, recibir la traza de ejecución de la aplicación MPI por parte del servidor y, por último, mostrarla en forma de gráfica.

La interfaz gráfica (figura 11) de *Client_Application* ha sido desarrollada en Java mediante el patrón Modelo-Vista-Controlador (MVC), donde se separa la lógica de la aplicación de la interfaz de usuario. *Client_Application* permite al usuario obtener de forma gráfica una trazabilidad de la aplicación MPI que escoja, de manera sencilla y rápida, siguiendo muy pocos pasos y guiándose únicamente de botones y simples popUps. La aplicación *Client_Application* cuenta con dos secciones bien diferenciadas.

La sección superior se encarga de la interacción con el usuario mediante el uso de botones. Los botones permiten la recopilación de información y la conexión con el servidor.

La sección inferior representa gráficamente la traza generada mediante el registro obtenido por parte del servidor. Esta gráfica en forma de barras indica el tiempo de ejecución de cada de la aplicación MPI enviada al servidor. De esta forma permite distinguir visualmente de forma muy clara qué tareas MPI o de E/S se han ejecutado en cada instante del tiempo, ya que a cada función MPI se le asigna un color único. Por último, en esta sección se incluyen una serie de botones que permiten desplazarse por el gráfico de manera horizontal.

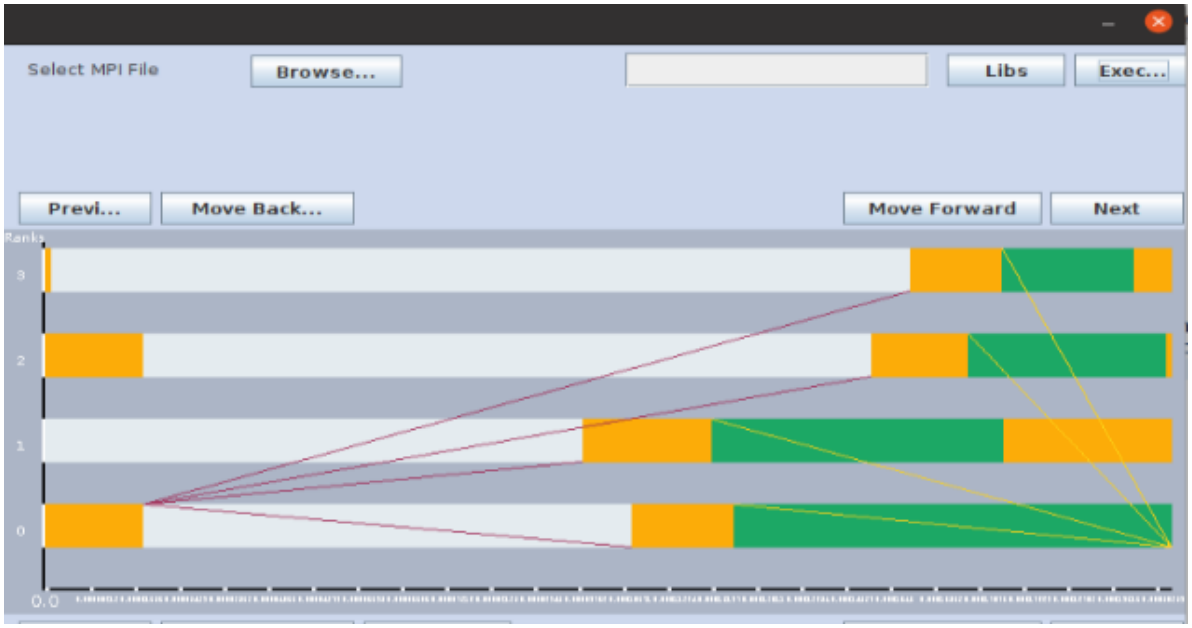


Figura 11. Interfaz de *Client_Application*

La funcionalidad de la aplicación *Client_Application* se divide en 3 fases (figura 12).

- Fase 1, recopilar información
- Fase 2, ejecución del cliente
- Fase 3, representación gráfica de la traza MPI

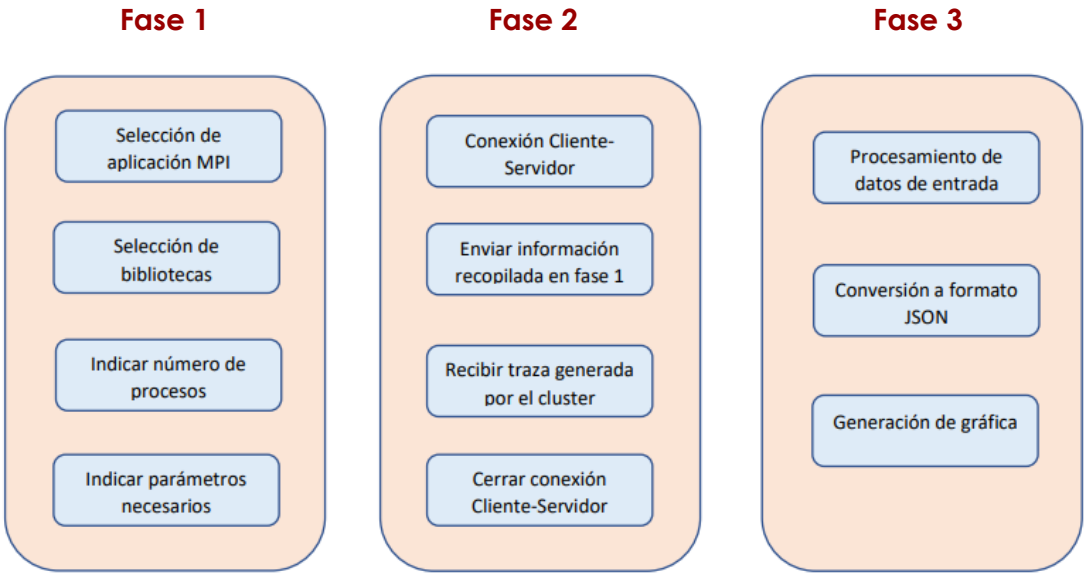


Figura 12. Representación Arquitectura de *Client_Application*

5.1 - Fase 1: recopilación de información

En esta fase el usuario introduce información para llevar a cabo el proceso de la obtención de la traza. Para ello, se solicita al usuario la siguiente información:

- Aplicación MPI a ejecutar.
- Bibliotecas necesarias para la compilación de la aplicación MPI.
- Número de procesos para la ejecución de la aplicación MPI.
- Parámetros necesarios para la ejecución de la aplicación MPI.

5.1.1 - Selección de Aplicación MPI

Al ejecutar la GUI aparecen diversos botones implementados mediante Java Swing (figura 13), que permiten la interacción entre la aplicación y el usuario. El primer botón *browse* que aparece permite seleccionar la aplicación MPI que queremos ejecutar en lenguaje C, mediante una ventana emergente lanzada por *chooseFile()*, (figura 14).

Si el control de errores no muestra ningún error al comprobar que la extensión del programa es ".c", la ruta al fichero seleccionado se envía al Controlador. A su vez se habilitan dos botones antes bloqueados; el botón *Libs* y el botón *Send*.

Si existe algún error, se muestra una ventana emergente con el motivo del error.

Figura 13. Vista para recopilar Información

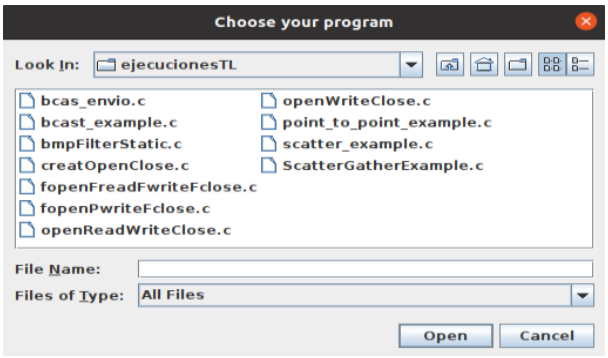


Figura 14.Selección de programa MPI

5.1.2 - Selección de Bibliotecas

El nuevo botón habilitado *Libs* (figura 13) permite seleccionar, en caso de ser necesario, un conjunto de bibliotecas para la compilación de la aplicación MPI seleccionada anteriormente mediante otra ventana emergente lanzada por *chooseFile()*. Estas bibliotecas se pasan al controlador del patrón MVC mediante un array de Strings, donde cada String representa la ruta a cada fichero. Este mismo array se utiliza para enviar al controlador el número de bibliotecas seleccionadas usando su método length().

Todos los ficheros seleccionados pasan un control de errores, donde se comprueba que las extensiones de las bibliotecas son ".h". En caso contrario, aparece una ventana emergente con el motivo del error.

5.1.3 - Selección de configuración de la ejecución

Por último, el botón *Send* se encarga de preguntar al usuario la configuración con la que quiere ejecutar el programa MPI mediante dos ventanas emergentes.

La primera ventana emergente necesaria para la ejecución del programa solicita el número de procesos con los que se va a lanzar la ejecución (figura 15). Una vez el usuario introduce el número deseado, este se envía al controlador en forma de String. La segunda ventana emergente permite al usuario introducir los parámetros necesarios para la ejecución de la aplicación MPI (figura 16).

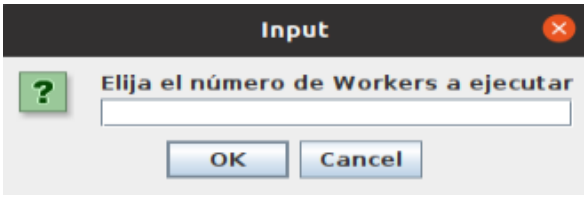


Figura 15.Solicitud número de workers

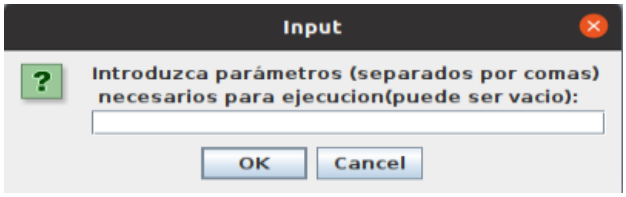


Figura 16.Solicitud de parámetros

5.2 - Fase 2: Ejecución del cliente

Una vez el controlador cuenta con todos los parámetros y la configuración para ejecutar la aplicación MPI deseada, llega la ejecución del cliente.

El controlador crea un objeto de la clase Cliente, que permite realizar conexiones a través de la red. El constructor del objeto obtiene la siguiente información:

- Ruta al programa MPI
- Número de bibliotecas
- Array de rutas a las distintas bibliotecas
- Número de Workers
- Parámetros para la ejecución del programa MPI

5.2.1 - Conexión Cliente-Servidor

La conexión Cliente-Servidor crea una comunicación entre la aplicación *Client_Application* y la aplicación instalada en el nodo front-end del servidor llamada *Server_Application* (detallada en el capítulo 6).

Los recursos que se mandan del cliente al servidor son los comentados en el punto anterior (sección 5.2) , mientras que los recursos que se envían del servidor al cliente consisten en un fichero que contiene la traza de ejecución de una aplicación MPI. La conexión es mediante sockets.

Para una implementación de la parte del Cliente en un sistema Cliente-Servidor correcto se han seguido los siguientes puntos:

1. Creación del Socket
2. Conexión Cliente-Servidor
3. Envío de información recopilada
4. Obtención de la traza MPI
5. Cierre de conexión Cliente-Servidor

5.2.1.1 - Crear Socket

Para crear el Socket del cliente utilizaremos la llamada a la función *socket()* del paquete java.net.

Este método intenta conectarse al servidor especificado, en el puerto especificado. Si este constructor no lanza una excepción, la conexión es exitosa y el cliente se conecta al servidor.

La dirección a la que intentará conectarse será la configurada en la Raspberry seleccionada como front-end.

El objeto creado de la clase Socket nos proporciona los métodos *getInputStream()* y *getOutputStream()*. Gracias a estos métodos accedemos a los dos *streams* asociados a un socket: el outputStream y el inputStream. El outputStream nos va a permitir crear un objeto de la clase DataOutputStream (*dos*). Mediante el *dos* se configura el flujo de salida, utiliza un buffer para escribir los datos de tipo primitivo de salida. El objeto DataInputStream *(dis)* lo utilizaremos para leer los datos de tipo primitivo que el servidor nos envíe a través de un flujo de datos. Se pueden observar las variables descritas en el siguiente código.

```
//Creación del socket
Socket socket = new Socket("192.168.56.12" , 5000);
DataOutputStream dos = new
                        DataOutputStream(socket.getOutputStream());
DataInputStream dis = new DataInputStream(socket.getInputStream());
```

5.2.1.2 - Realizar conexión Cliente-Servidor

La conexión se realiza utilizando el protocolo TCP. A la hora de realizar la llamada a la función *socket* hay que asegurarse de que el servidor esté conectado y en espera. De otra forma, se mostrará un error.

La comunicación entre el Cliente y el Servidor es parte fundamental de este trabajo, se trata del intercambio de información entre ambas partes. Vamos a distinguir dos partes de la comunicación: escribir (enviar datos) y leer (recibir datos).

5.2.1.3 - Enviar datos

Con respecto al envío de datos, principalmente utilizaremos la llamada a función *writeUTF()* de la clase DataOutputStream mediante el objeto *dos*. Esta función escribe una cadena de caracteres en el flujo de salida en formato UTF-8 que permite conocer el tamaño de la cadena enviada. Utilizaremos esta función para enviar todos los requisitos para que el Servidor pueda ejecutar la aplicación MPI.

De esta forma, enviamos al servidor mediante el flujo de datos el nombre del fichero MPI, el número de procesos que va a ejecutar el programa MPI y sus correspondientes parámetros. Si el programa no requiere parámetros, se envía un String vacío. Seguidamente, se envían el número de bibliotecas necesarias, si al menos hay una se procede a enviar el nombre y el contenido (por medio de *sendBuffer()*) de cada biblioteca.

Por último, utilizando la función implementada *sendBuffer()* se envía el contenido del fichero ".c" de la aplicación MPI.

```
//Send MPI file name
dos.writeUTF(nameMPI);
//Send number of processors
dos.writeUTF(nProc);
//Send params
dos.writeUTF(params);
//Send number of libs
dos.writeUTF(String.valueOf(numLibsFiles));

if(numLibsFiles > 0) {
      //Send libs name
      for(int i = 0; i < numLibsFiles; i++) {
            dos.writeUTF(nameLibs[i]);
      }
     //Send libs file
      for(int i = 0; i < numLibsFiles ; i++) {
            sendBuffer(libFiles[i],dos);
      }
}
//Send MPI file
sendBuffer(mpiFile, dos);
```

```
void sendBuffer(File file, DataOutputStream dos) throws IOException {
          byte[] bytes = new byte[1024];
          InputStream in = new FileInputStream(file);
          int count;
          while((count = in.read(bytes))> 0) {
               dos.write(bytes, 0, count);
          }
          dos.flush();
          in.close();
     }
```

La función *sendBuffer()* tiene como objetivo enviar el contenido de un fichero al front-end del servidor, donde:

- *file* es el fichero a enviar. Se trata del programa MPI o las bibliotecas necesarias para la ejecución de este.
- *dos* es el objeto DataOutputStream que permite el paso de mensajes hacia el Servidor.

Para esta ocasión, en lugar de invocar a la llamada *writeUTF(),* se invoca a la llamada *write(),* la cual escribe en el flujo de datos un número de bytes especificado por parámetro a partir de un byte del fichero indicado también por parámetro. Esta operación se hace mientras haya bytes disponibles para leer en el fichero. La lectura del fichero se realiza mediante un objeto de la clase InputStream que permite el acceso a un fichero.

5.2.1.4 - Recibir datos

A continuación, pasa a explicarse cómo el cliente obtiene los datos generados por el servidor. Esta información es: el nombre de la traza generada y su contenido.

Con respecto a la recepción de datos, utilizaremos la llamada a función *read()* de la clase DataInputStream mediante el objeto *dis.* Esta función guarda una cadena de caracteres del flujo de entrada en formato UTF-8 que permite conocer el tamaño de la cadena recibida. Utilizaremos esta función para recibir el nombre de la traza generada por el Servidor.

```
//Receive number of processors
this.nameTrace = dis.readUTF();
//Receive text contain
getBuffer(nameTrace, dis, controller.getDestTrace());
```

De esta forma, recibimos del servidor el nombre de la traza generada. A continuación se invoca a la función implementada *getBuffer()*.

```
void getBuffer( String nameFile, DataInputStream dis,
String dest ) throws IOException {
   OutputStream out = new FileOutputStream( dest + nameFile);
   byte[] bytes = new byte[1024];
   int count;
   while((count = dis.read(bytes)) > 0) {
           out.write(bytes,0,count);
           if(count < 1024) {
                 out.flush();
                 break;
           }
     }
     out.close();
}
```

Esta función *getBuffer()* recibe desde el servidor el contenido de un fichero. Donde:

- *nameFile* es el nombre del fichero a recibir.
- *dis* es el objeto DataInputStream que permite el paso de mensajes desde el servidor.
- *dest* es la ruta donde se va a guardar el fichero a recibir.

Para la implementación de esta función, inicialmente se crea un objeto de la clase OutputStream para crear un nuevo fichero con el nombre y la ruta recibidos por parámetro. Utilizando este mismo objeto se escribe el número de bytes extraidos del flujo de datos conectado al servidor utilizando el objeto DataInputStream. Cuando el número de bytes leídos sean menor que los esperados, se utiliza el método *flush()* encargado de vaciar el flujo de salida y hacer que se escriban los bytes de salida almacenados.

5.2.1.5 - Cerrar Conexión

El último punto correspondiente al buen funcionamiento del cliente en un sistema Cliente-Servidor hace referencia al cerrado de los flujos de datos abiertos y accediendo al controlador para actualizar la ruta de la traza generada por el Servidor.

```
controller.setTraceLib(controller.getDestTrace() + this.nameTrace);
dis.close();
dos.close();
```

5.3 - Fase 3: Representación gráfica de aplicaciones MPI

Por último, una vez recibida la traza devuelta por el servidor, se consigue un gráfico de barras que representa la ejecución de una aplicación MPI. Donde a cada proceso le corresponde una de las barras horizontales que salen del eje vertical. Las barras cambian de color en función de la tarea que se ejecute entre dos instantes de tiempo, estos dos instantes representan el inicio y el fin de la tarea. Con respeto al eje horizontal, representa el tiempo de ejecución del programa MPI completo.

La principal idea de representar la ejecución de esta forma es, a parte de ver gráficamente la ejecución, obtener una visión clara de posibles problemas de rendimiento como cuellos de botella, y obtener distintas perspectivas de una misma aplicación MPI al cambiar su configuración de ejecución, por ejemplo, variando el número de procesos. De esta forma, comparando distintas gráficas de una misma aplicación MPI podemos encontrar posibles mejoras en el rendimiento.

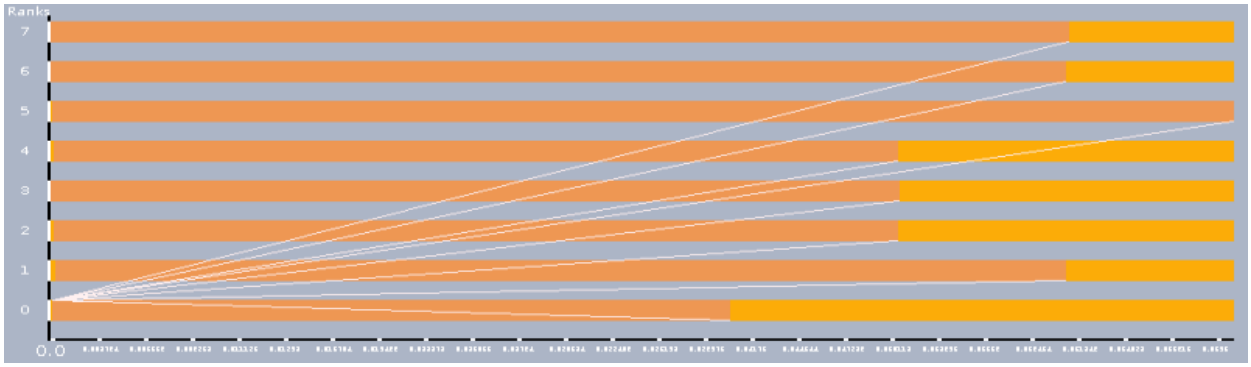


Figura 17. Representación gráfica de función MPI_Bcast con 8 procesos

La figura 17 representa la ejecución de un programa MPI con 8 procesos donde cada uno de ellos invoca la función *MPI_Bcast*(). El máster (proceso 0) envía un valor que todos los procesos lo reciben. El color naranja representa el tiempo que el proceso está esperando a recibir el valor enviado por el proceso master, el color amarillo indica que ya no se encuentra ejecutando ninguna función MPI o E/S, sino que se están ejecutando operaciones de cómputo, y por último, la línea desde el proceso master hasta cada proceso indica el envío del dato.

En el ejemplo de la figura 17, podemos observar claramente que el instante en el que llega la línea dirigida desde el proceso master a otro proceso, prácticamente coincide con el instante que termina la ejecución de la tarea *MPI_Bcast*(). Esto es debido a que la función está implementada de tal manera que permanece ejecutándose en espera hasta que recibe un valor.

La siguiente figura (figura 18), muestra gráficamente la ejecución de una aplicación MPI que invoca a varias llamadas MPI. El objetivo de esta función era calcular la media de los elementos de un array de forma paralela entre varios procesos usando *MPI_Scatter()* y *MPI_Gatter()*, detalladas en la sección 4.3.3 .

Podemos observar claramente que, las secciones de color blanco representan el tiempo que un proceso está ejecutando la función *MPI_Scatter()*, las secciones de color verde representan el tiempo que un proceso está ejecutando la función *MPI_Gatter()*, y por último, las secciones de color amarillo representan las funciones de cómputo que no pertenecen a llamadas MPI, en este caso, las franjas amarillas que se encuentran entra las llamadas *MPI_Scatter()* y *MPI_Gatter()* significan cálculos para obtener la media de un conjunto de valores.

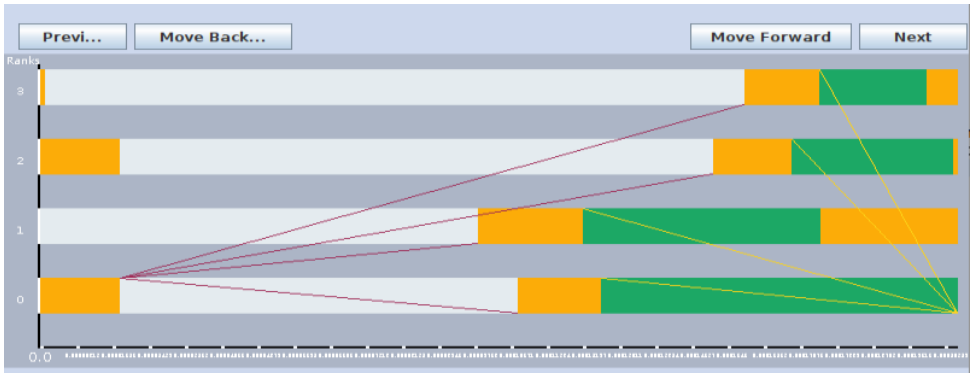


Figura 18. Representación gráfica de Aplicación MPI para calcular la media.

La primera vista que se obtiene del gráfico muestra una vista final tras la ejecución de todas las tareas. Para poder retroceder o avanzar en el orden de ejecución de las tareas utilizamos los botones *Previous* y *Next*. Mientras que, los botones *Move Backward* y *Move Forward* permiten retroceder o avanzar en el eje horizontal del gráfico.

La transformación de la traza de una aplicación MPI recibida a una gráfica es posible mediante la conversión previa de esta traza a un formato JSON[16], un formato de texto sencillo que principalmente se utiliza para el intercambio de datos.

Los objetos de este formato son colecciones que se representan mediante pares de la forma <clave>:<valor>, separadas por comas y puestas entre llaves, como se muestra en la figura 19. Gracias a este formato podemos representar cada ejecución como un objeto de forma sencilla.

Figura 19. Conversión de traza de ejecución MPI a formato JSON

Por último, es necesario comentar que para el desarrollo de la gráfica se ha reutilizado el código realizado por Bryan Raúl Bryan Vergas, adaptándolo a la implementación de este trabajo.

[16] https://www.json.org/json-es.html

6 - Obtención de trazas de aplicaciones MPI en entornos distribuidos. Parte Servidor.

Este capítulo describe el funcionamiento de la aplicación *Server_Application* desplegada en el *cluster* de Raspberry Pi.

Server_Application es una aplicación distribuida que se encarga de generar trazas de aplicaciones MPI ejecutadas en un cluster de Raspberry Pi.

La aplicación *Server_Application* se instala y se ejecuta en el front-end del *cluster* de Raspberry Pi. Tras esto, se pone a la espera de una petición de conexión mediante sockets por parte de *Client_Application* (detallada en el capítulo 5).

Realizada esta conexión, el front-end recibe la información necesaria para poder compilar y ejecutar una aplicación MPI, con el objetivo final de generar sus trazas de ejecución. Esta información es, como se ha explicado en la sección 5.1, la siguiente:

- Aplicación MPI a ejecutar.
- Bibliotecas necesarias para la compilación de la aplicación MPI.
- Número de procesos para la ejecución de la aplicación MPI.
- Parámetros necesarios para la ejecución de la aplicación MPI.

Tras evaluar que la obtención de recursos ha sido correcta, el front-end se encarga de compilar y ejecutar la aplicación MPI mediante una serie de scripts. Y, gracias a la librería *TraceLib* comentada en capítulo 1, la aplicación MPI ejecutada genera el registro de trazas con la información de cada proceso.

Terminada la ejecución, el front-end envía a *Client_Application* el nombre de la traza generada y su contenido mediante la misma conexión abierta anteriormente. Una vez terminada la transmisión de datos, esta conexión se cierra. Por último, el front-end del cluster de Raspberry Pi vuelve a ponerse a la escucha esperando una nueva conexión.

6.1 - Ejecución del Servidor

Antes de nada es necesario instalar la aplicación Server_Application en nuestro nodo elegido como front-end e instalación de nodos Back-End en el resto.

6.1.1 - Configuración del nodo front-end

Para una ejecución efectiva y libre de fallos hay que configurar el entorno en el nodo seleccionado como front-end. Esto conlleva la creación de ciertos directorios y ficheros indispensables para el desarrollo del programa. De tal manera que el árbol de directorios debe quedar de la siguiente forma:

```
/ Directorio_Servidor/
 |→ archivos_MPI/
 |→ ejecutables_MPI/
 |→ TraceTxt/
 |→ Machines/
    |→ machines.txt
 |→ Scripts/
    |→ executeMPI.sh
    |→ sortTrace.sh
 |→ TraceLib/
     |→ traceLib.c
     |→ traceLib.h
|→ config.properties
|→ Servidor.jar
```

Figura 20. Árbol de directorios en nodo front-end

Tanto los ficheros, como los scripts mencionados en la figura 20, se explican detalladamente más adelante, en la sección 6.4.

Para que el programa sea portable y se pueda instalar en diversos *clusters* de forma sencilla y rápida, se ha diseñado un Script de instalación que automatiza la creación de carpetas y ficheros.

6.2 - Conexión Cliente-Servidor

La primera interacción del servidor es la creación de un socket asociado a un puerto especificado (que el cliente debe conocer). Una vez creado el socket se pone a la escucha esperando que el cliente se conecte mediante el método *accept*.

De la misma manera que el cliente, se crean objetos de la clase DataInputStream y DataOutputStream.

```
ServerSocket server = new ServerSocket(5000);
 while(true){
  Socket socket = server.accept();
  DataInputStream dis = new DataInputStream(socket.getInputStream());
  DataOutputStream dos = new
                          DataOutputStream(socket.getOutputStream());
                                  … …
```

6.3 - Recopilación de información

Toda la información enviada por *Client_Application* se debe recoger de manera sincronizada a cómo se envían (sección 5.2.1.3). Por ello, el código implementado es muy similar al código del cliente.

Pero en vez de realizar llamadas dos.writeUTF(), se realizan llamadas a dis.readUTF(). Por último, el contenido de los ficheros se adquieren mediante la función getBuffer() explicada anteriormente.

```
//MPI File name received
mpiFilename = dis.readUTF();
//number of workers received
nProp = dis.readUTF();
//Params received
params = dis.readUTF();

//numLibs received
numLibs = Integer.parseInt(dis.readUTF());
if(numLibs > 0) {
     nLibs = new String[numLibs];
     for(int i = 0; i < numLibs; i++){
           //Library name received
           nLibs[i] = dis.readUTF();
     }
     for(int i = 0; i < numLibs; i++){
           //Library content received
           getBuffer(socket, properties,nLibs[i], dis);
   }
 }
//MPI File content received
getBuffer(socket, properties,mpiFilename, dis);
```

6.4 - Ejecución Programa MPI

Una vez el front-end tiene todos los requisitos para compilar y procesar un programa MPI, se invoca a la función *compileMPIFile()*.

6.4.1 - Compilar biblioteca TraceLib

Para obtener el registro de trazas, primeramente, hay que crear un objeto *traceLib.o* a partir de la biblioteca *TraceLib*.

El objeto *traceLib.o* enlazado con la aplicación MPI recibida por parte de *Client_Application* va a permitir la generación de trazas por parte del nodo Front-End. Su creación se realiza compilando diferentes archivos mediante el compilador *gcc* que permite compilar ficheros c. A continuación se muestra el comando de compilación que se ejecuta mediante la propia clase de Java *Process*. Donde:

- *pathLib* representa la ruta donde se guardará el objeto *TraceLib.o* .
- *nameLibO* corresponde al nombre que tendrá el objeto *TraceLib.o* .
- *pathIncludeMPI es la ruta donde se encuentra el fichero MPI necesario para la compilación*
- *nameLibC* es el nombre del archivo *TraceLib.c* .

Los *-I<path>*[17] son para indicar dónde están los ficheros de cabecera necesarios para la compilación, mientras que, los *-c<path>* indica el fichero fuente escrito en lenguaje c.

```
"gcc -o " +  properties.getProperty("pathLib")
         +  properties.getProperty("nameLibO")
         + " -I" + properties.getProperty("pathIncludeMpi")
         + " -c "+  properties.getProperty("pathLib")
          + properties.getProperty("nameLibC")
```

Se ha decidido simplificar el acceso a las diferentes rutas y nombres mediante la clase estándar *properties*[18]. Esta clase permite modificar archivos que se almacenan de forma permanente una serie de valores emparejados. De esta forma, si es necesario

[17] http://www.chuidiang.org/clinux/herramientas/librerias.php

[18] https://javiergarciaescobedo.es/programacion-en-java/15-ficheros/358-archivo-de-propiedades-properties

actualizar la ruta o el nombre de algún fichero, solo hay que modificar el valor correspondiente al nombre.

6.4.2 - *Compilación y Ejecución de programa MPI*

En segundo lugar, se procede a la compilación y ejecución del programa MPI mediante el siguiente script de Bash. Para la ejecución del Script también se ha utilizado la clase *Process*.

```
#!/bin/bash

export PATH=$PATH:/home/rpimpi/mpi-install/bin

file=$1 #File .c to compile
libTrace=$2 #Path/File library needed to compile
where=$3 #Path/File executable generated
numProc=$4
machines=$5
dest=$6
params=$7

cd $dest

mpicc -o $where -I/home/rpimpi/mpi-install/include  $file $libTrace

mpiexec -hostfile $machines -np $numProc $where $params
```

Este script, en primera instancia, se encarga de compilar y enlazar todas las bibliotecas necesarias para el compilado mediante el comando *mpicc*. A parte de enlazar la biblioteca *TraceLib* se enlaza también toda biblioteca necesaria para la aplicación MPI donde:

- *file* es la aplicación MPI a compilar y ejecutar.
- *libTrace* corresponde al objeto *libTrace.o* creado en la sección 6.4.1 .
- *where* representa la ruta al objeto MPI que se va a crear si el compilado es correcto

Si el compilado no ha dado error, se invoca el comando *mpiexec*. Este comando permite ejecutar una aplicación MPI. Donde:

- *machines* corresponde al fichero que contiene las direcciones de las máquinas que ejecutarán el programa MPI
- *numProc* representa el número de procesos.
- *where* es el objeto MPI obtenido en el compilado mediante *mpicc*.
- *params* son los argumentos necesarios para la ejecución del programa MPI

En este punto es conveniente mencionar y mostrar un ejemplo del fichero *machines*. Este fichero es necesario para conocer en qué Raspberry Pi del *cluster* se despliegan los procesos de la aplicación MPI.

Como podemos observar, en este ejemplo se despliegan cuatro Raspberry Pi. Cada una se identifica con su correspondiente dirección de red. El hecho de que el *master* realice cómputo o no, depende de de la aplicación MPI que se ejecute.

```
master:192.168.54.12
computo-1:192.168.54.9
computo-2:192.168.54.3
computo-3:192.168.54.15
```

Figura 21. Representación fichero *machines*

6.5 - Generación y envío de Traza_MPI

Esta sección abarca la forma que tiene el servidor de generar un informe que representa la traza de ejecución de una aplicación MPI y su envío al cliente.

6.5.1 - Generación de la traza

Tras la ejecución del programa MPI enlazado con la biblioteca *TraceLib*, se obtienen diversos ficheros de trazas correspondientes a la ejecución de cada proceso. Cada proceso genera su propia traza para evitar desvirtuar los tiempos de ejecución. Ya que, si todos los procesos acceden al mismo fichero de manera concurrente, se intercalan escrituras que pueden provocar la pérdida de registros. Existen técnicas de

exclusión mútua que solventan este problema, pero su implementación implicaría alterar el tiempo de ejecución.

Una vez finalizada la ejecución de la aplicación MPI, se constituye una única traza juntando todas las trazas generadas por cada proceso. Estas trazas se ordenan por orden de ejecución en función del tiempo. Para ello, las marcas de *ini* y *end* deben de estar separadas. A continuación podemos observar un ejemplo(figura 22):

```
2.536734 Ini_bcas  0 3 1 0
2.536849 End_bcas  0 3 1 0
```

Figura 22. Representación traza MPI generada

Donde:

- El primer parámetro indica la marca de tiempo asociada a la tarea
- El segundo parámetro indica el nombre de la tarea
- El tercer parámetro indica el identificador de la tarea
- El cuarto parámetro es el identificador del proceso que invoca la llamada
- El quinto parámetro indica el número de datos transferidos
- El sexto parámetro indica el identificador del proceso que envía los datos a los procesos receptores

El hecho de generar diversas trazas independientes por cada proceso y luego juntarlas en una única traza_MPI puede dar lugar a generar un desorden en los tiempos de ejecución. Por ello se ha implementado el siguiente Script que se invoca tras la ejecución del Script de compilación y ejecución.

```
if [ -f trace_complete.txt ]
then
   rm trace_complete.txt
fi

echo Generate Trace
cat *simulation_mpi_* | sort > trace_complete.txt #Order by timeStamp
echo Deleting Traces simulation_mpi_
rm *simulation_mpi_* #Delete traces of each rank
```

Este Script asegura que no exista ningún fichero con el mismo nombre que el fichero a generar. Seguidamente forma un único fichero y ordena por orden numérico y alfabético cada línea de la traza mediante el comando *sort*.

6.5.2 - Envío de la traza

Por último, hay que enviar *Traza_MPI* a *Client_Application* mediante el socket comentado anteriormente (sección 6.2). Para ello, de nuevo reutilizamos el código utilizado en la parte del *Client_Application*. Si la compilación y ejecución ha sido correcta se enviará el nombre de la traza seguido de su contenido. En caso contrario se enviará un 0. Una vez terminada la ejecución, se procede a la desconexión del socket y la creación de otro que espera nuevas peticiones de conexión (sección 6.2).

6.5.3 - Compilación y ejecución del servidor

Antes de comenzar con la configuración del único nodo que intercambiará información con el cliente, es necesario enlazar y compilar *Server_Application*.

Este programa, que permite la ejecución de un Servidor capaz de conectarse y atender las peticiones de la aplicación *Client_Application*, se ha desarrollado en java, por lo que los pasos a seguir para crear un archivo ejecutable *.jar* son los siguientes:

Lo primero es posicionarse en el directorio donde se encuentran las distintas clases que componen el programa.

```
cd /home/pi/TFG/Servidor/src/
```

Una vez en este directorio hay que crear un archivo *.class* por cada una de estas clases mediante el compilador *javac*

```
1 javac ExecuteCommands/Command.java
2 javac GetProperties/GetProperties.java
3 javac Logger/Logger.java
4 javac Server/main.java
```

Lo siguiente es indicar cuál de los archivos anteriores contiene la función *main*. Para ello es necesario crear un archivo MANIFEST.MF en la misma ruta comentada anteriormente, en este archivo hay que añadir la siguiente línea

```
Main-Class: Server.main
```

Finalmente ya podemos crear el archivo *.jar* con los elementos obtenidos anteriormente mediante el comando *jar* y ejecutándose mediante el comando *java -jar*

```
1 jar cvmf MANIFEST.MF server.jar ExecuteCommands/Command.class
  GetProperties/GetProperties.class Logger/Logger.class
  Server/main.class

2 java -jar server.jar
```

7 - Arquitectura general de la reproducción de trazas en el simulador

SiGSED es una aplicación desarrollada enteramente en Java, la cual, mediante sockets, permite comunicación entre cliente-servidor (figura 23).

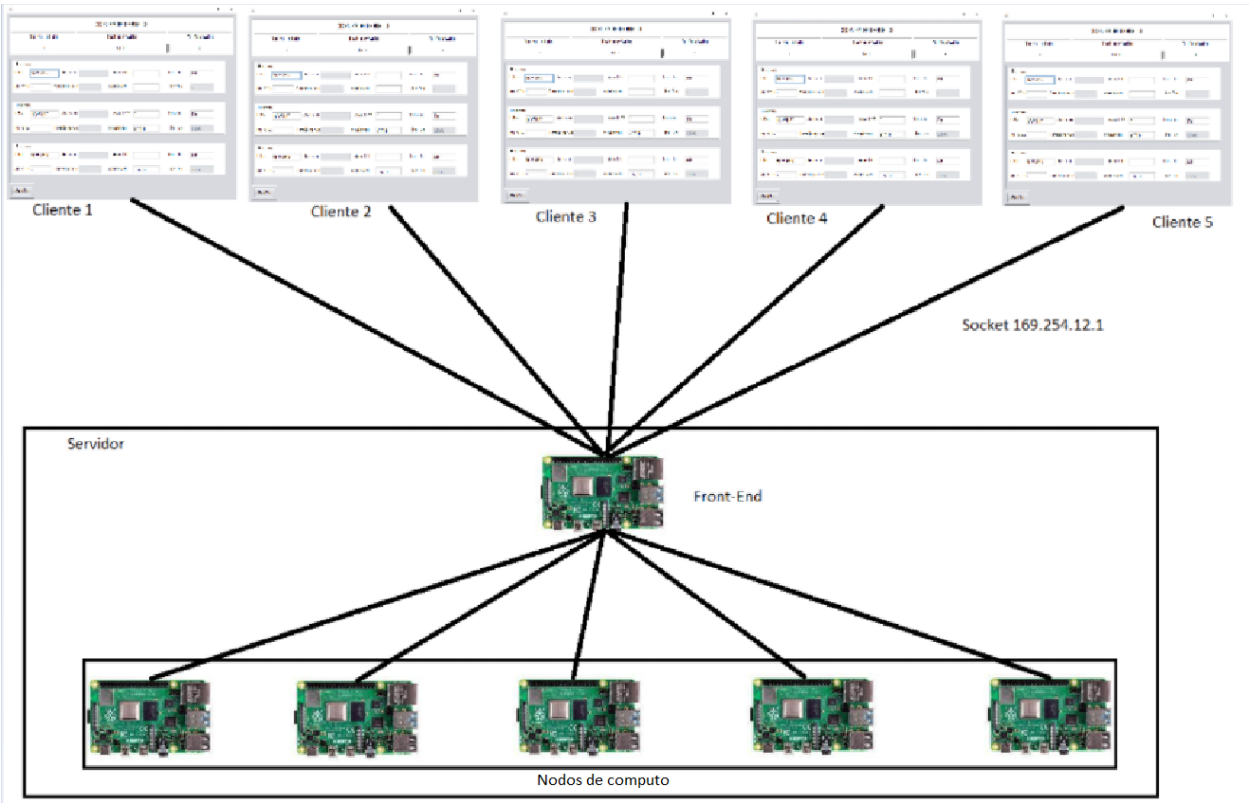


Figura 23. Arquitectura general de la simulación de trazas

En la parte cliente se encuentra la interfaz gráfica de la aplicación, con la que interactúa el usuario a la hora de enviar los ficheros de configuración, y conocer el estado del *cluster*. Además, se encuentra SIMCAN-GUI que es la interfaz de usuario en la que se permite a los usuarios configurar y modelar entornos distribuidos para el simulador SIMCAN.

El lado del servidor estará dividido en dos partes: el front-end y los nodos de cómputo. El front-end es el encargado del reparto de trabajo entre los distintos nodos de

cómputo, y siendo estos últimos los encargados de realizar las simulaciones utilizando SIMCAN. Además, el front-end se encarga de la obtención de los datos de estado de cada uno de los nodos de cómputo, necesarios para el cliente para conocer el estado del *cluster*.

7.1 - SIMCAN

SIMCAN es una plataforma de simulación para modelado y simulación tanto de sistemas como de aplicaciones distribuidas. Actualmente el simulador es de código abierto. SIMCAN ha sido desarrollado en C++ utilizando OMNeT++, una biblioteca de simulación centrada en el desarrollo de simuladores de redes. La aplicación se desarrolló enfocada a la investigación, aunque actualmente ha sido adaptada para ser utilizada en la docencia.

Uno de los objetivos que posee SIMCAN, y que beneficia el desarrollo de este trabajo, es el alto nivel de flexibilidad y escalabilidad, que permite a los usuarios modelar una amplia gama de configuraciones de sistema distribuidos. Esta flexibilidad que ofrece SIMCAN se basa en su repositorio, el cual contiene una colección de modelos que representan los componentes más relevantes de un sistema distribuido, como son la CPU, los discos y las redes de comunicación. Por ejemplo, una CPU puede ser modelada como si fuera Single-Core o como si fuera Quad-core.

SIMCAN proporciona el mismo método de agrupación que el utilizado en los sistemas distribuidos reales, es decir, una rack que contiene varias placas, donde cada placa contiene varios nodos de cómputo. El tamaño de estos racks y placas es completamente configurable. Por lo tanto, los sistemas grandes se pueden implementar fácilmente utilizando este método.

Por último, SIMCAN proporciona APIs, de las siglas en inglés Application Programming Interfaces, para el desarrollo de sistemas distribuidos. Estas APIs están basadas en APIs reales, que facilitan la programación. Por ejemplo, SIMCAN ofrece una API similar a POSIX[19], que contiene las principales funciones para la gestión de ficheros, memoria y servicios de red. Así mismo, SIMCAN posee un conjunto de llamadas MPI para la ejecución de aplicaciones MPI en escenarios de simulación utilizando SIMCAN.

[19] http://get.posixcertified.ieee.org/

7.2 - SIMCAN-GUI

SIMCAN-GUI es una aplicación, programada en Java, que permite a los usuarios configurar y modelar entornos distribuidos para SIMCAN.

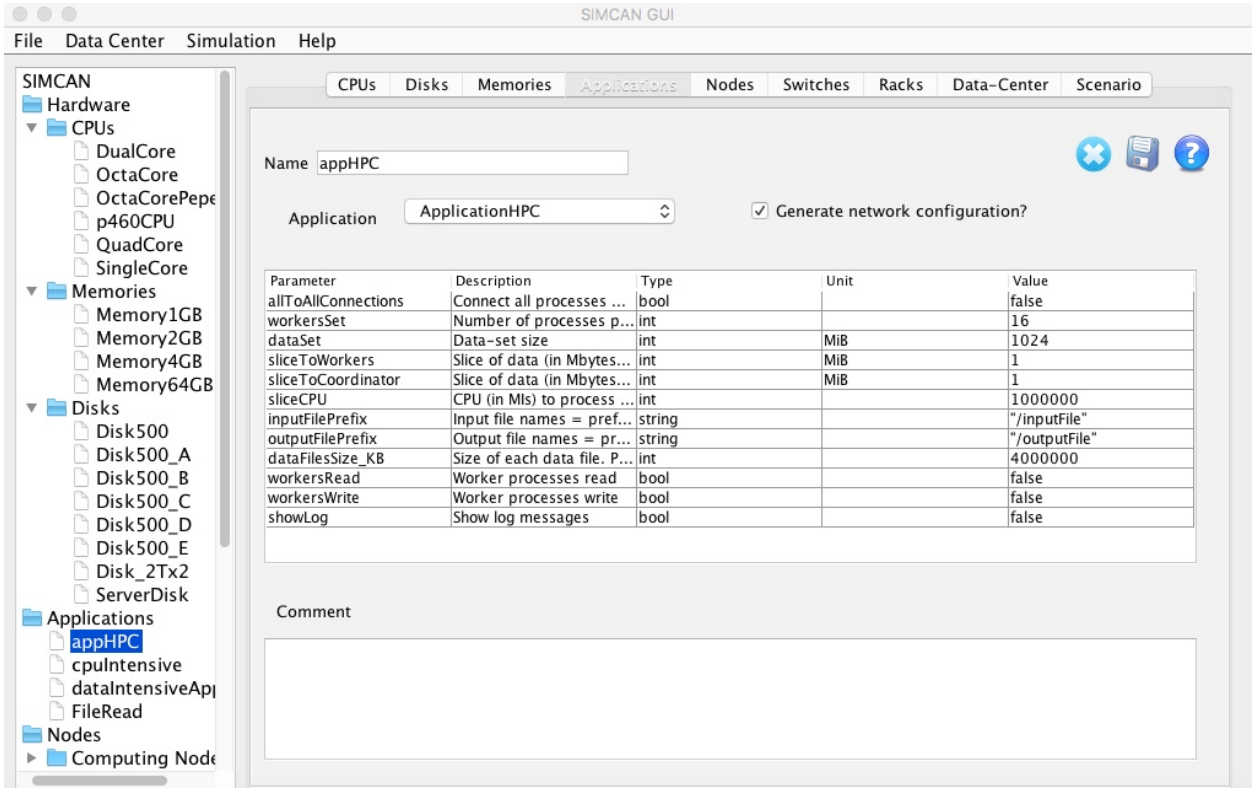


Figura 24. Configuración de una aplicación distribuida en SIMCAN-GUI

Como se ha mencionado en el apartado anterior, una de las particularidades que ofrece SIMCAN es el repositorio, a través del cual ofrece distintos modelos de componentes para facilitar al usuario el diseño de sistemas distribuidos. Este repositorio es accesible mediante SIMCAN-GUI (figura 24) y muestra un repositorio de componentes, en la parte izquierda de la interfaz, y sus especificaciones, en el cuadro principal.

En la parte superior de la interfaz, en una misma fila, se muestran los distintos componentes que se pueden configurar, en el caso de la figura 24 es una aplicación, sin embargo en el caso de la figura 25 es un nodo.

Los nodos se pueden personalizar completamente utilizando los módulos del repositorio, es decir, discos, CPU, memorias, entre otros, como vemos en la figura 24. Analizando la figura se puede observar que es posible realizar cambios en la configuración, tales como cambiar el nombre del nodo, el tipo de CPU que utiliza, qué tipo de planificador de CPU utiliza, la capacidad de la memoria, el tipo de disco de almacenamiento y las aplicaciones que va a simular.

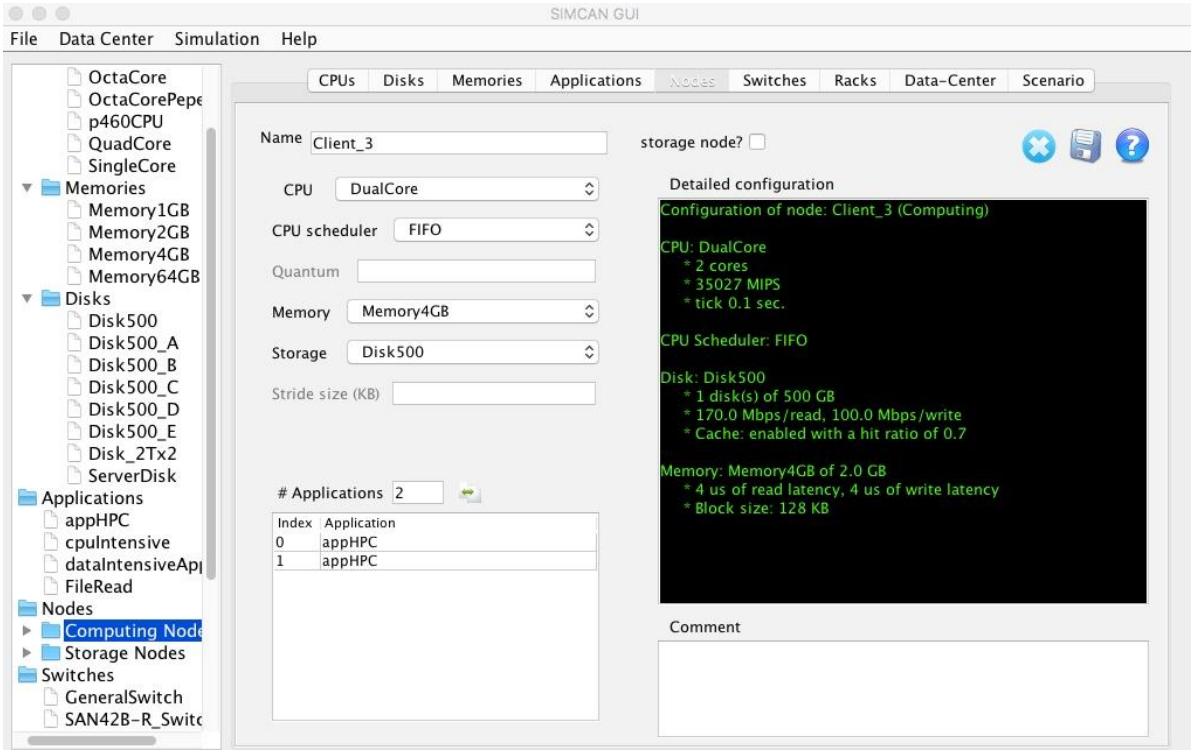


Figura 25. Configuración de un nodo en SIMCAN-GUI

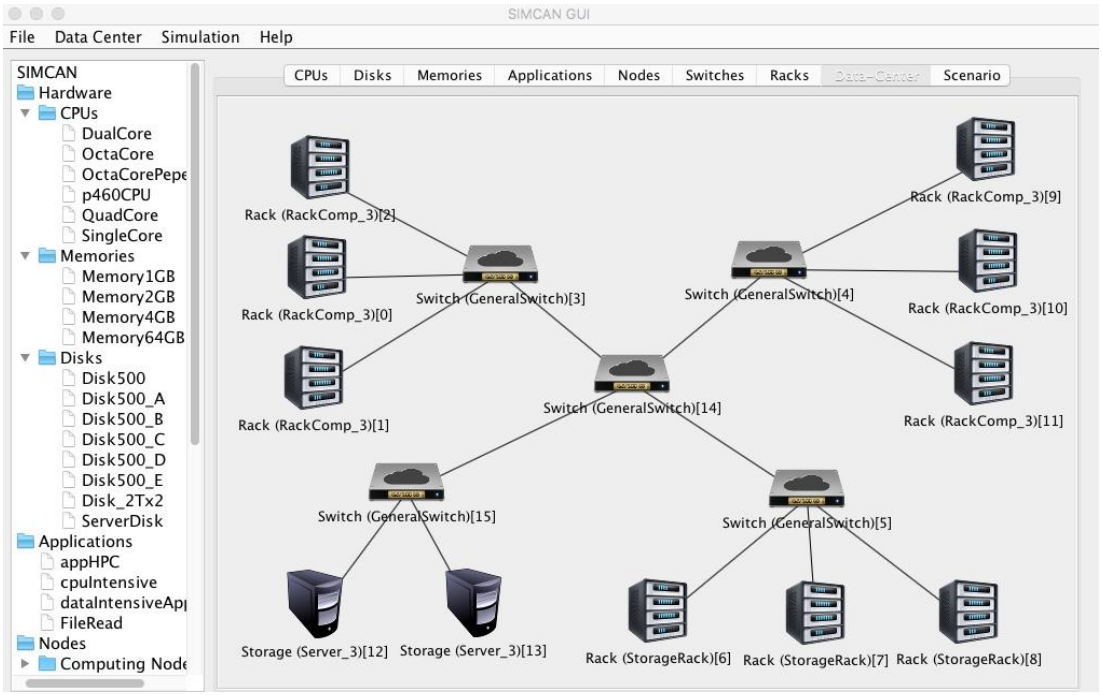
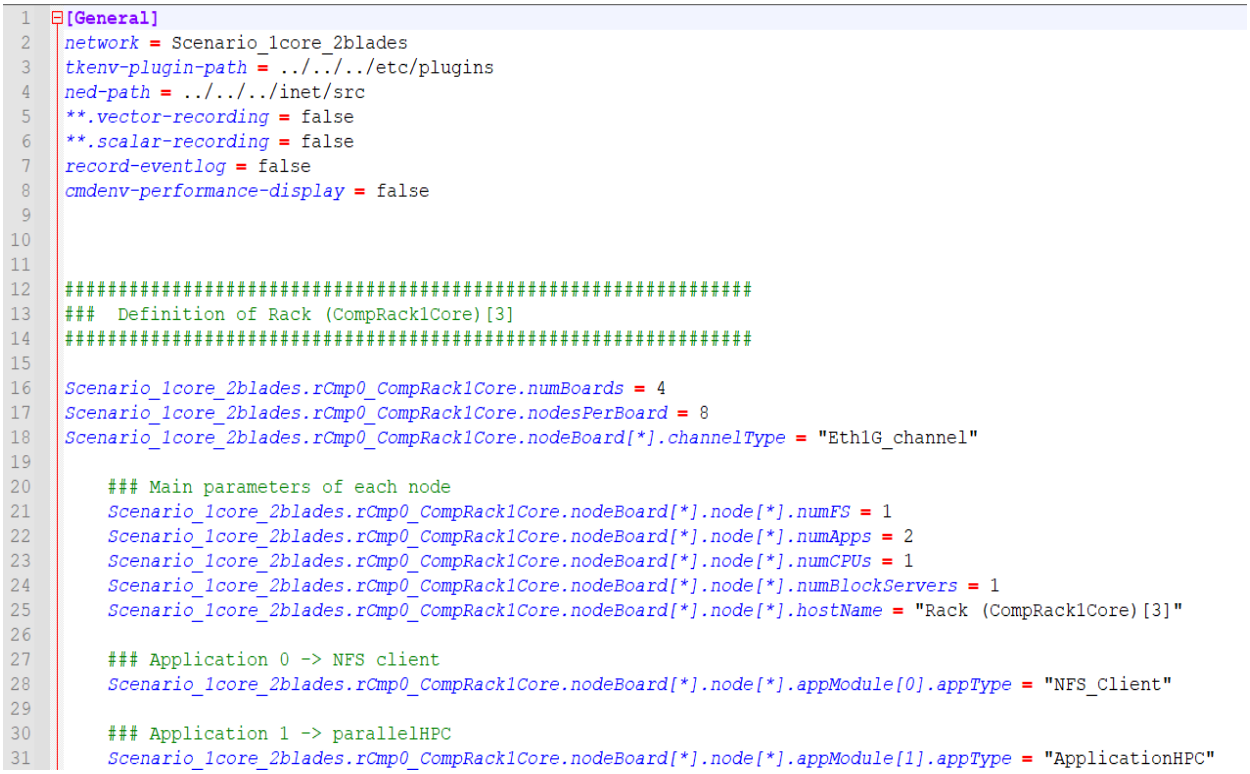


Figura 26. Configuración de scenarios en SIMCAN-GUI

Por último, en la figura 26 se puede ver la representación de un data-center en SIMCAN, diseñado desde SIMCAN-GUI. En la figura se puede observar que el data-center está compuesto por 12 nodos de cómputo, 3 de almacenamiento, y 5 switches. Además, tanto los componentes como las redes pueden ser configuradas de manera individual. De esta forma, se permite generar redes heterogéneas, es decir, redes cuyos enlaces entre los nodos tengan distintas velocidades.

Una vez que terminada la configuración de todos los componentes, en la parte de *scenario* se solicita el directorio en el que se quiere guardar el proyecto además de un nombre con el que guardarlo. Tras seleccionar el nombre se generan los ficheros *.ini, .ned, run* y la carpeta *config*. Los ficheros .ini y .ned poseen la configuración del sistema que se quiere simular.

En el fichero *.ini* (figura 27) se encuentra la información del escenario a simular como es el tipo de racks de cómputo, racks de almacenamiento o tipo de aplicación a ejecutar.



```
[General]
network = Scenario_1core_2blades
tkenv-plugin-path = ../../../etc/plugins
ned-path = ../../../inet/src
**.vector-recording = false
**.scalar-recording = false
record-eventlog = false
cmdenv-performance-display = false



############################################################
###  Definition of Rack (CompRack1Core)[3]
############################################################

Scenario_1core_2blades.rCmp0_CompRack1Core.numBoards = 4
Scenario_1core_2blades.rCmp0_CompRack1Core.nodesPerBoard = 8
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].channelType = "Eth1G_channel"

    ### Main parameters of each node
    Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].numFS = 1
    Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].numApps = 2
    Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].numCPUs = 1
    Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].numBlockServers = 1
    Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].hostName = "Rack (CompRack1Core)[3]"

    ### Application 0 -> NFS client
    Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].appModule[0].appType = "NFS_Client"

    ### Application 1 -> parallelHPC
    Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].appModule[1].appType = "ApplicationHPC"
```

Figura 27. Ejemplo de Fichero .ini

En la parte superior, líneas de la 1 a la 8, se muestra la configuración general del sistema. Entre los elementos más importantes cabe destacar el identificador de la red, *Scenario_1core_2blades.*

A continuación, las líneas 16 a la 18, muestran la configuración de un rack de cómputo, especificando el número de placas, de nodos por placa y el tipo de conexión.

```
############################################################
###  Definition of Rack (CompRack1Core)[3]
############################################################
Scenario_1core_2blades.rCmp0_CompRack1Core.numBoards = 4
Scenario_1core_2blades.rCmp0_CompRack1Core.nodesPerBoard = 8
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].channelType
= "Eth1G_channel"
```

Seguidamente, las líneas de la 21 a la 25, se definen los parámetros principales de cada rack, que son el número de aplicaciones a ejecutar, número de CPUs, número de servidores y el nombre de que se le asignará a cada nodo.

```
### Main parameters of each node
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].num
FS = 1
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].num
Apps = 2
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].num
CPUs = 1
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].num
BlockServers = 1
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].hos
tName = "Rack (CompRack1Core)[3]"
```

En la parte inferior, las líneas 28 y 31, son el tipo de aplicaciones que se van a ejecutar en el nodo.

```
### Application 0 -> NFS client
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].app
Module[0].appType = "NFS_Client"
### Application 1 -> parallelHPC
Scenario_1core_2blades.rCmp0_CompRack1Core.nodeBoard[*].node[*].app
Module[1].appType = "ApplicationHPC"
```

Por otro lado en el fichero *.ned,* a continuación se muestra un fragmento, contiene la descripción de la red en un formato desarrollado para OMNeT++. En él se definen los componentes que forman la red, es decir, switches, racks, el tipo de red que se utiliza, en este caso IPv4, y cómo se conectan cada uno de ellos.

```
package SIMCAN.simulations.Scenario_1core_2blades;

import ned.DatarateChannel;
import SIMCAN.Nodes.Racks.*;
import SIMCAN.Nodes.Nodes.*;
import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
import inet.nodes.ethernet.EtherSwitch;
```

```
// ------------------------------------------------------------
//   Definition of the scenario
// ------------------------------------------------------------
network Scenario_1core_2blades{

   // ---------------------------------------------------------
   //   Definition of main modules
   // ---------------------------------------------------------
   submodules:
      //   Definition of switches
      //
       switch_0:EtherSwitch;

      //
-----------------------------------------------------------------
      //   Definition of Racks
      //
-----------------------------------------------------------------
      // Rack (CompRack1Core)[3]
      rCmp0_CompRack1Core:Rack;

      // Rack (CompRack1Core)[4]
      rCmp1_CompRack1Core:Rack;

      // Rack (storageRack2blades)[5]
      rSto2_storageRack2blades:Rack;
```

Por último, el fichero *run* es un script que lanzará la ejecución una vez que estos ficheros se hayan generado.

7.3 - Introducción al entorno Cliente

En esta sección se lleva a cabo una breve explicación de la parte cliente y será desarrollada en profundidad en el capítulo 8.

En el cliente se encontrará instalado SIMCAN-GUI y la interfaz gráfica de la aplicación desarrollada, SiGSED. Así, una vez que el usuario haya terminado de configurar los escenarios, lanzará la aplicación. Una vez que SiGSED esté en ejecución, se mostrará la interfaz principal de la aplicación. Esta se encargará de mostrar el estado de los recursos del servidor refrescando la información de manera periódica. Además, el usuario podrá enviar al servidor, bajo demanda, los ficheros de la configuración que quiera ejecutar, pulsando el botón para enviar ficheros, añadiendo la ruta a los

ficheros seleccionados para que el servidor realice la simulación y presionando por último el botón de confirmar.

7.4 - Introducción al entorno Servidor

En esta sección se hará una breve explicación de la parte servidor y será desarrollada en profundidad en el capítulo 9.

En el servidor encontraremos dos partes a su vez: front-end y nodos de cómputo.

El front-end posee la parte de la aplicación correspondiente al servidor, mientras que los nodos de cómputo, dedicados a la simulación, tienen instalado SIMCAN. El front-end se encuentra en una de las Raspberry Pi del *cluster*, y no estará destinada a realizar simulaciones, únicamente se encargará del reparto de trabajos y obtención de datos y resultados. El front-end no dispone de interfaz gráfica, pues no es necesaria para su uso y en caso de que hubiera algún error, este se mostraría por la consola de la propia Raspberry Pi. Además, el front-end podrá atender las peticiones de distintos clientes, pues es multihilo.

Los nodos de cómputo están plenamente dedicados a la ejecución de simulaciones mediante SIMCAN y únicamente se comunican con el front-end, así el front-end puede controlar mejor qué trabajo está haciendo cada nodo de cómputo

8 - Diseño de la parte cliente del Sistema de Gestión y Simulación de Entornos Distribuidos

En este capítulo se detalla el desarrollo del sistema SiGSED en la parte cliente.

El cliente estará desplegado en una máquina externa al *cluster*, en este caso un pc de sobremesa. Este se encargará de mandar los ficheros de simulación, comentados previamente en la sección 7.2, al servidor para que simule los entornos proporcionados, procese los resultados y los envíe al cliente. Además, cabe destacar que el cliente dispone de un sistema de monitorización de recursos del servidor para conocer el estado de cada una de las máquinas que se encuentran en el *cluster*.

8.1 - Comunicación con el servidor

Para llevar a cabo la comunicación entre cliente y servidor se establece una conexión mediante sockets. Esta comunicación no es constante, pues esto podría saturar el front-end con peticiones muy costosas, así que el cliente únicamente establecerá conexión con el servidor bajo demanda, mediante un mecanismo de polling configurable. Este mecanismo permite regular la carga de peticiones enviadas del cliente al servidor para evitar así su saturación.

El cliente únicamente conocerá la IP y el puerto de escucha del front-end del servidor, el resto de comunicaciones serán orquestadas de manera interna por el front-end.

8.2 - Desarrollo de la GUI

La parte del cliente es la única parte del SiGSED que posee una GUI. En esta se mostrará el estado del servidor, además de proporcionar los mecanismos necesarios para enviar los ficheros al front-end. La GUI está desarrollada utilizando la plataforma NetBeans, ya que los mecanismo proporcionados por su IDE facilitan la generación de una GUI.

Una primera pantalla que permite el inicio de sesión en el sistema, para el que se solicita un usuario y una contraseña (figura 28). En la segunda ventana si la contraseña se ha introducido correctamente, se muestra la información de las Raspberry Pi (figura

29), además de un botón en la esquina inferior izquierda. Una vez presionado este botón, el programa se encarga de solicitar la ruta a los ficheros de simulación (figura 30), esta ventana de la interfaz es explicada en profundidad en la sección 8.3. Una vez añadida la ruta se habilita la opción de enviar los ficheros de simulación que se han obtenido previamente, como se explica en el capítulo anterior. En el caso en el que se haya producido un error aparecerá una ventana que proporciona información relacionada al respecto (figura 31).

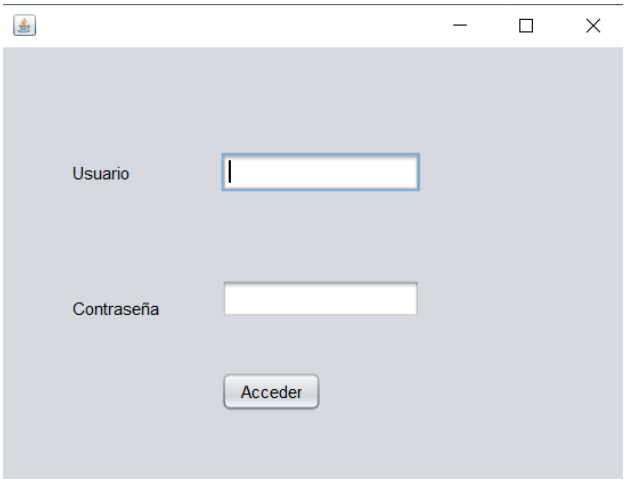


Figura 28. Acceso al sistema

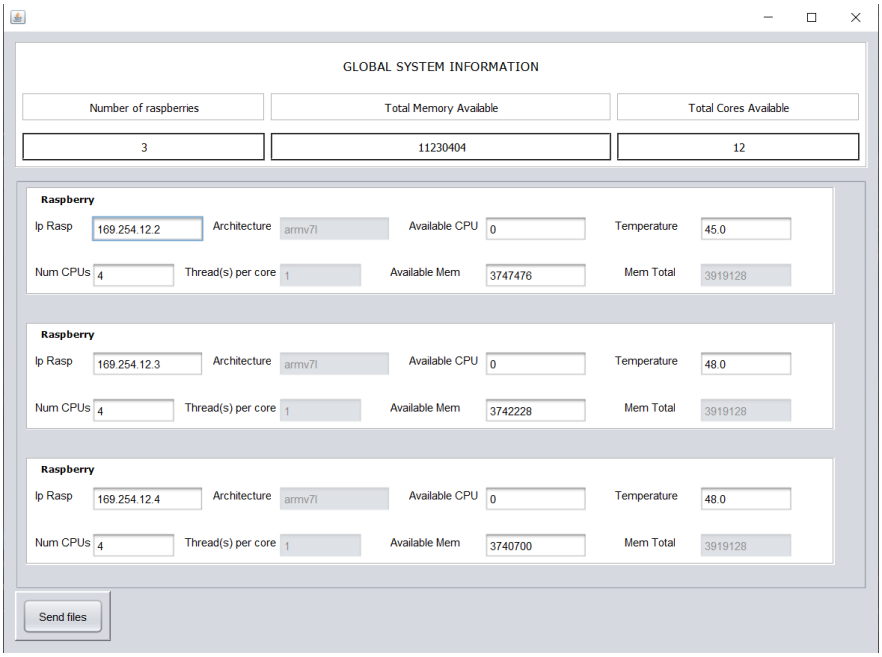


Figura 29. Monitorización de recursos

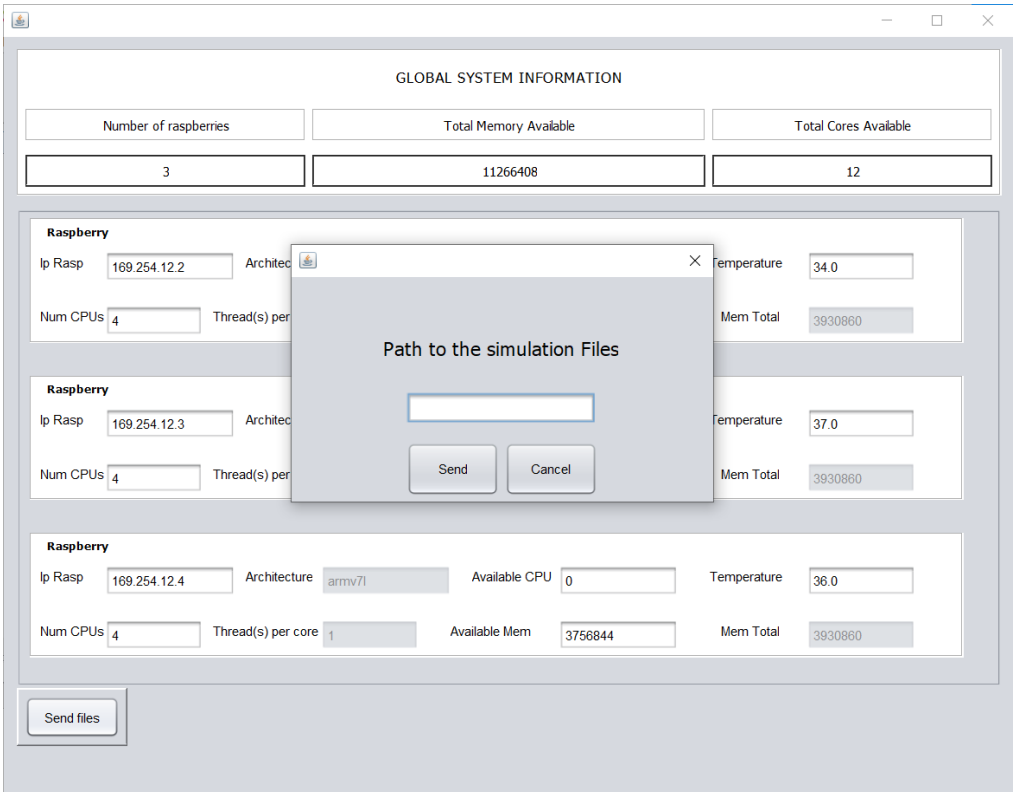


Figura 30. Solicitud de ruta a los ficheros de simulación y envío

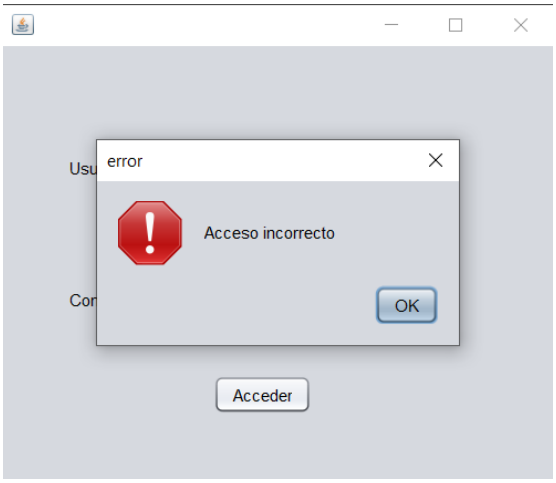


Figura 31. Fallo en el registro al acceder al sistema

8.3 - Monitorización de recursos

Uno de los aspectos necesarios para la correcta gestión y utilización del *cluster* de computación es conocer sus recursos disponibles. Para ello la monitorización de los recursos del sistema es fundamental. Al establecerse la conexión mediante sockets con el servidor, la parte cliente, solicita al front-end la información que posea del resto de nodos, en ese momento.

Mediante un array, la información relacionada con los recursos disponibles del *cluster* llegan al cliente a través del front-end. Este array está compuesto de estructuras de datos que se denominan *RaspInfo*. Cada *RaspInfo* contiene la información de uno de los nodos del *cluster*. Al recibir el array con los datos estos se cargan en la interfaz mediante el jPanel *RaspInfoPanel* (figura 32).

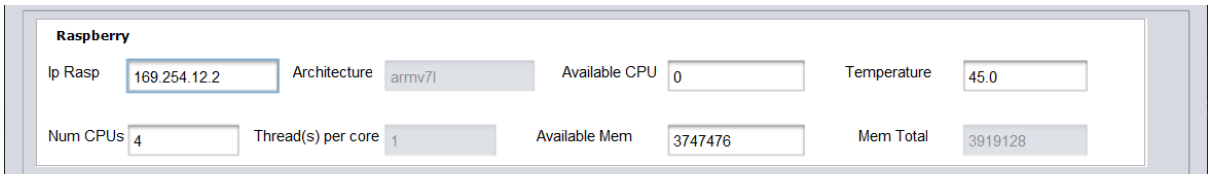


Figura 32. Rasp InfoPanel

Para mantener los datos actualizados y así conocer los cambios que se produzcan en el *cluster*, en este caso, después de un periodo de tiempo se envía una petición al front-end, para que envíe el estado del *cluster* en ese momento. Una vez recibida la información se cierra la conexión con el servidor. De esto se encarga el método run(), el cual crea la tarea que invoca al método *consultaEstado()* cada TIME_TO_WAIT milisegundos. Esta variable está inicializada a 6000 milisegundos.

El método *consultaEstado()* se encarga de enviar al servidor la petición de consulta del estado de los nodos de cómputo. Cuando el servidor envíe los datos del *cluster* al cliente estos se reciben como un *Object* y seguidamente son convertidos a *ListRaspInfo* que es una clase que contiene una lista con la información referente a los nodos de cómputo del Servidor.

```
public void run() {
    java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        Timer timer;
        timer = new Timer();

        TimerTask task;
        task = new TimerTask() {
            int tic = 0;

            @Override
            public void run() {
                try {
                    ListRaspInfo rasp;
                    rasp = client.consultaEstado();
                    raspList = rasp;
                    actualizaInfor();
                } catch (IOException | ClassNotFoundException ex) {

Logger.getLogger(MainIface.class.getName()).log(Level.SEVERE, null, ex);
                }
            }
        };
        // Empezamos dentro de 6000ms y luego lanzamos la tarea cada 6000ms
        timer.schedule(task, TIME_TO_WAIT, TIME_TO_WAIT);
    }
);
```

```
public static ListRaspInfo consultaEstado() throws IOException,
ClassNotFoundException {
        sc = new Socket(HOST, puerto);
        DataOutputStream out = new DataOutputStream(sc.getOutputStream());
        out.writeUTF("2");

        ObjectInputStream raspEnt = new
ObjectInputStream(sc.getInputStream());
        //Recibimos el array de raspberrys
        Object riList = (Object) raspEnt.readObject();
        ListRaspInfo rasp = (ListRaspInfo) riList;

        for (int i = 0; i < rasp.getList().size(); i++) {
            RaspInfo get = rasp.getList().get(i);
            String aux = get.getArchi().replaceAll("Architecture:",
"").replace(" ", "");
            get.setArchi(aux);
        }
        sc.close();
        return rasp;
    }
```

Además, en la GUI también se encuentra un apartado que nos mostrará los recursos totales del sistema (figura 33), en el cual se encuentra el número de Raspberry Pi que hay en el *cluster* activas, la memoria total disponible y el número total de cores que hay en el *cluster* disponibles. Este apartado se actualiza a la vez que el *RaspInfoPanel* pues se calculan los datos con la suma de los campos correspondientes a cada una de las Raspberry Pi. *ActualizaInfor()* se encarga de actualizar los datos que se han recibido en cada uno de los *jPanel* de la interfaz de las Raspberry Pi así como los datos totales del sistema.

```
public void actualizaInfor() {
        int memSistema, numRaspis, numCores;
        memSistema = numCores = 0;
        numRaspis = index;
        for (int i = 0; i < index; i++) {
            RaspInfoPanel raspi = infoList.get(i);
            raspi.setTxtCPUDispRasp("" +
raspList.getList().get(i).getCpuDisp());
            raspi.setTxtIpRsp(raspList.getList().get(i).getIpRasp());
            raspi.setTxtMemDispRasp("" +
raspList.getList().get(i).getMemDisp());
            memSistema += raspList.getList().get(i).getMemDisp();
            raspi.setTxtTempRasp("" +
raspList.getList().get(i).getTemperatura());
            raspi.setTxtMemTotal("" +
raspList.getList().get(i).getMemTotal());
            raspi.setTxtThreads("" +
raspList.getList().get(i).getNumThreads());

            raspi.setTxtNumCPUs("" + raspList.getList().get(i).getNumCpus());
            numCores += raspList.getList().get(i).getNumCpus();
            raspi.setTxtArchi(raspList.getList().get(i).getArchi());
        }
        panelNumRaspTXTInfoGlobal.setText("" + numRaspis);
        panelMemDispTXTInfoGlobal.setText("" + memSistema);
        panelCoresDispTXTInfoGlobal.setText("" + numCores);
        panelInfoGlobal.updateUI();
        panelRaspis.updateUI();
    }
```

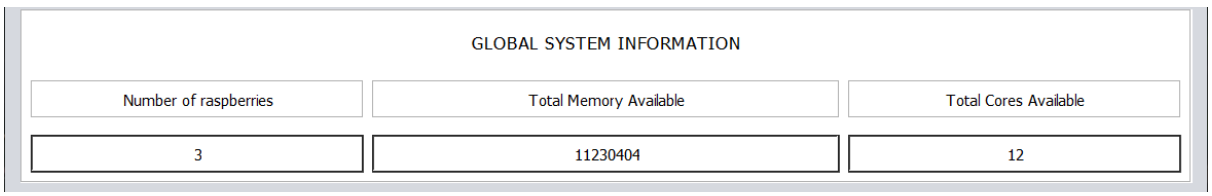


Figura 33. Información global del sistema

8.4 - Envío de ficheros: entornos de simulación

Para poder simular el funcionamiento de la aplicación MPI en entornos de simulación, el usuario debe proporcionar una serie de ficheros que contengan la descripción de los entornos. En este caso, se deben proporcionar los ficheros con extensión *.ini* y *.ned*, que contienen la configuración y la topología de los entornos distribuidos donde la aplicación va a ser simulada. Estos archivos se pueden encontrar en cualquier directorio del sistema.

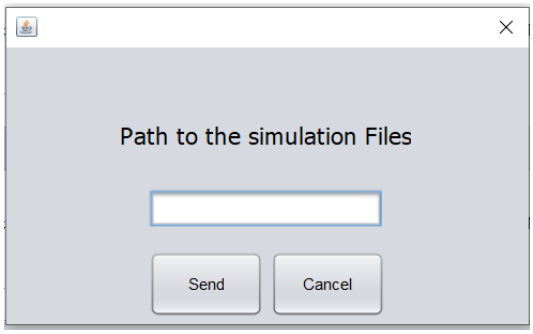


Figura 34. Envío de ficheros al Servidor

Una vez que se pulse el botón de enviar, mencionado en la figura 29 aparece una nueva ventana. Esta ventana (figura 34) solicita la ruta correspondiente a los dos archivos de configuración requeridos. Debajo del panel de texto a rellenar con la ruta hay dos botones. El botón de la izquierda, llamado *send,* se encarga de que el programa comprima los ficheros en una nueva carpeta, que tiene por nombre la dirección IP del cliente con extensión *.zip,* envíe estos dos archivos al servidor y una vez terminada la tarea cierre la ventana y vuelva la interfaz principal de la aplicación (figura 29). El botón de la derecha, llamado *cancel,* simplemente cierra la ventana para volver a la interfaz principal de la aplicación, sin realizar ninguna operación de envío.

Tras el envío de la carpeta comprimida se cerrará el socket por el cual se ha establecido la conexión. Seguidamente, el cliente abrirá un nuevo socket que permanecerá a la espera de que el servidor devuelva los resultados de la simulación mediante una nueva conexión.

El método *sendComprimido()* se encarga de recibir el path a través de la ventana mostrada en la figura 34 y mandar la carpeta comprimida que contiene los ficheros de configuración al servidor. En caso de que la ruta no exista, aparecerá un mensaje indicando que la ruta especificada no es válida.

```
public static boolean sendComprimido(String path) throws IOException {
        sc = new Socket(HOST, puerto);
        int entry;
        byte[] byteArray;
        BufferedInputStream bis;
        BufferedOutputStream bos;
        DataOutputStream out;
        DataInputStream in;
        File localFile;

        String filename = comprimir(path);

        if(filename.equalsIgnoreCase("")) return false;
        localFile = new File(filename);
        bis = new BufferedInputStream(new FileInputStream(localFile));
        bos = new BufferedOutputStream(sc.getOutputStream());
        out = new DataOutputStream(sc.getOutputStream());
        in = new DataInputStream(sc.getInputStream());

        out.writeUTF("1");
        while (!in.readUTF().equalsIgnoreCase("accepted"));
        //Enviamos el nombre del fichero y el fichero
        out.writeUTF(localFile.getName());
        byteArray = new byte[8192];
        while ((entry = bis.read(byteArray)) != -1) {
            bos.write(byteArray, 0, entry);
        }
        bis.close();
        bos.close();
        sc.close();
        return true;
    }
```

El método *comprimir(path)*, invocado por *sendComprimido()* es el encargado de comprimir los ficheros *.ini* y *.ned* de la ruta especificada en la carpeta con nombre la IP del cliente que lo está realizando. Para que el sistema operativo no de errores en la carpeta generada se cambia el caracter "." por el caracter "_". Si se ejecuta correctamente devuelve la ruta al fichero recién comprimido, en caso de no existir la ruta se devolverá una ruta vacía indicando que no existe.

```
private static String comprimir(String path) throws FileNotFoundException,
IOException {
        // ruta completa donde están los archivos a comprimir
        File carpetaComprimir = new File(path);
        String ret = "";
        // valida si existe el directorio
        if (carpetaComprimir.exists()) {
            // lista los archivos que hay dentro del directorio
            File[] ficheros = carpetaComprimir.listFiles();
        // crea un buffer temporal para ir poniendo los archivos a comprimir
            ZipOutputStream zous;
            InetAddress direccion = InetAddress.getLocalHost();
            String IP_local = direccion.getHostAddress().replace(".","_");
            FileOutputStream fos = new
FileOutputStream("..\\comprimir\\"+IP_local+".zip");
            zous = new ZipOutputStream(fos);
            ret = "..\\comprimir\\"+IP_local+".zip";

            // ciclo para recorrer todos los archivos a comprimir
            for (int i = 0; i < ficheros.length; i++) {
                if (ficheros[i].getName().endsWith(".ini")||
                    ficheros[i].getName().endsWith(".ned")) {

                    ZipEntry entrada = new ZipEntry(ficheros[i].getName());
                    zous.putNextEntry(entrada);

                    //obtiene el archivo para comprimirlo
                    FileInputStream fis = new FileInputStream(directorioZip +
entrada.getName());
                    int leer;
                    byte[] buffer = new byte[1024];
                    while (0 < (leer = fis.read(buffer))) {
                        zous.write(buffer, 0, leer);
                    }
                    fis.close();
                }
            }
            zous.closeEntry();
            zous.close();
            fos.close();
            System.out.println("Directorio de salida: " + ret);
        } else {
            System.out.println("No se encontró el directorio..");
        }
        return ret;
    }
```

8.5 - Recepción de resultados

Tras enviar los ficheros de simulación al servidor es necesario mantener un medio de comunicación abierto.

Para ello el cliente abre un socket que permanece a la espera de una conexión con el servidor. Una vez que el servidor se conecte a este socket se recibirán los resultados de la simulación en una carpeta comprimida y una vez que se terminen de recibir los datos, se notifica mediante un pop-up (figura 35) que los resultados han sido recibidos satisfactoriamente.

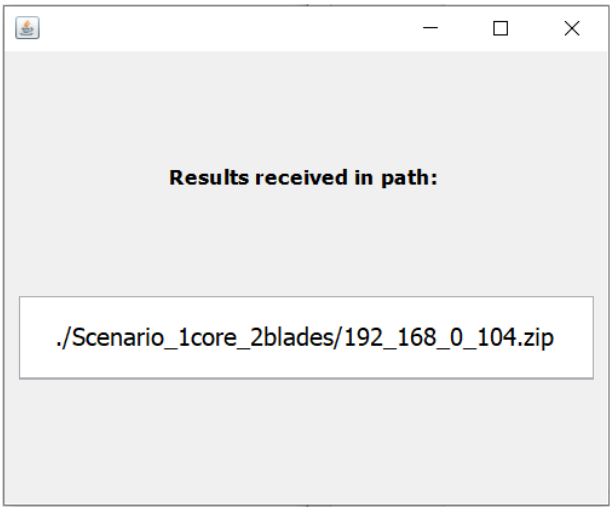


Figura 35. Ruta a los resultados recibidos del servidor

9 - Diseño de la parte servidor Sistema de Gestión y Simulación de Entornos Distribuidos

En este capítulo se detalla el funcionamiento del servidor. De la misma manera que en el capítulo 8 se describe la parte del cliente, en la parte servidor encontramos dos estructuras de funcionamiento distintas. Una primera que da soporte a la monitorización del *cluster* y otra que será la encargada de llevar a cabo la simulación. Esta última parte se encarga del reparto de ficheros con extensión *.ini* y *.ned*, mencionados en el capítulo 7.2, entre los demás nodos del *cluster* y de la ejecución de los escenarios una vez recibidos estos ficheros.

9.1 - Monitorización del *cluster*

En esta sección se detalla el proceso que sigue el servidor para obtener la información de cada uno de los nodos que componen el *cluster*.

9.1.1 - Conexión con el cliente

Una vez que el servidor ha recibido una petición del cliente para que le proporcione la información de cada nodo del *cluster*, el front-end invoca el método *consultEstado()*:

```
private void consultarEstado() throws IOException, InterruptedException {
        byte[] byteArray;
        DataOutputStream dos = new DataOutputStream(sc.getOutputStream());

        runScript(dos);
        System.out.println("Tam de claslist: " + raspis.getList().size());
        ObjectOutputStream raspOut = new
ObjectOutputStream(sc.getOutputStream());
        raspOut.writeObject((Object) raspis);
    }
```

En él se invoca al método *runScript(dos)*, explicado en el apartado 9.1.2, y una vez obtenidos los resultados se envía el objeto *raspis*, que es de tipo *ListRaspInfo*, que contiene la información de los nodos de cómputo solicitada por el cliente.

9.1.2 - *Obtención de información de cada worker*

Para la obtención de la información de cada uno de los nodos worker del *cluster* se ejecuta el método *runScript(dos)*.

```
private void runScript(DataOutputStream dos) throws IOException,
InterruptedException {
        Process proc = null;
        DataInputStream dis = new DataInputStream(sc.getInputStream());

        for (int i = 0; i < NUMNODOS; i++) {
            File fichero = new File("./filesNeeded/", "Salida"+ i +".txt");
            try {
                if (fichero.createNewFile()) {
                    System.out.println("fichero creado correctamente número:"
+ i);
                    proc = Runtime.getRuntime().exec("sudo chmod 777
./filesNeeded/Salida" + i + ".txt");

                }
                else {
                    System.out.println("El fichero ya existe");
                }
            }
            catch (IOException e) {
                e.printStackTrace();
            }
            proc = Runtime.getRuntime().exec("sudo chmod 777
./filesNeeded/Salida" + i + ".txt");

        }

        proc = Runtime.getRuntime().exec("./rshScript.sh");
        try {
            readFile();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
```

Este método se encarga de la creación de los ficheros donde se almacenará la información obtenida de cada nodo de cómputo. Una vez creados tantos ficheros como nodos de cómputo haya en el *cluster* se ejecuta la línea:

```
proc = Runtime.getRuntime().exec("./rshScript.sh");
```

Esta línea se encarga de ejecutar el script *rshScript*, el cual debe obtener mediante RSH la información de cada nodo de cómputo y almacenarla en los ficheros creados previamente.

```
#!/bin/bash
v=0
for i in $(cat ./ipFile.txt); do
    echo "IpRasp" > ./filesNeeded/Salida$v.txt
    echo $i >> ./filesNeeded/Salida$v.txt
    rsh -l pi $i cat /proc/meminfo >> ./filesNeeded/Salida$v.txt
    rsh -l pi $i lscpu >> ./filesNeeded/Salida$v.txt
    echo "IpRasp" > ./filesNeeded/Salida$v.txt
    rsh -l pi $i cat /sys/class/thermal/thermal_zone0/temp >>
./filesNeeded/Salida$v.txt
    v = v+1
done
```

Donde:

- *ipFile.txt* es un fichero donde se encuentran las ips de los distintos nodos de cómputo del *cluster*.
- *./filesNeeded/Salida$v.txt* es un subdirectorio en el cual se encuentran los ficheros que contienen los datos de los nodos.
- */proc/meminfo* es un directorio en el que se encuentra la información de memoria del dispositivo en el que se vaya a consultar.
- `lscpu` es un comando que muestra la información de la cpu.
- */sys/class/thermal/thermal_zone0/temp* es un directorio en el que se encuentra la temperatura de la Raspberry Pi.

Una vez terminada la ejecución del script, es necesario filtrar parte de la información obtenida. Esto es debido a que se pretende enviar únicamente la información que necesita el cliente para no sobrecargar el sistema. Se invoca al método *readFile()*, encargado de rellenar la estructura *raspis*, que será la que se envíe al cliente con los datos necesarios.

```
private void readFile() throws FileNotFoundException {
        FileReader fr = null;
        BufferedReader br;
        File[] ficheros = (new File("./filesNeeded/")).listFiles();
        for (int i = 0; i < ficheros.length; i++) {
            if (ficheros[i].getName().contains("Salida") &&
                ficheros[i].getName().endsWith(".txt")) {
```

```
                try {
                    br = new BufferedReader(new FileReader(ficheros[i]));
                    RaspInfo aux = new RaspInfo();
                    System.out.println(ficheros[i].getName());
                    System.out.println(ficheros[i].length());
                    String linea;
                    while ((linea = br.readLine()) != null) {
                        if (linea.contains("IPrasp")) {
                            linea = br.readLine();
                            aux.setIpRasp(linea);
                        } else if (linea.contains("MemTotal")) {
                            int res = (new Scanner(linea)).
                               useDelimiter("\\D+").nextInt();
                            aux.setMemTotal(res);
                        } else if (linea.contains("MemFree")) {
                            int res = (new Scanner(linea)).
                               useDelimiter("\\D+").nextInt();
                            aux.setMemFree(res);
                        } else if (linea.contains("MemAvailable")) {
                            int res = (new Scanner(linea)).
                               useDelimiter("\\D+").nextInt();
                           aux.setMemDisp(res);
                        } else if (linea.contains("Architecture")) {
                            aux.setArchi(get.getArchi().replaceAll(
                               "Architecture:","").replace(" ", ""));
                        } else if (linea.contains("CPU(s)")) {
                            int res = (new Scanner(linea)).
                               useDelimiter("\\D+").nextInt();
                            aux.setNumCPUs(res);
                        } else if (linea.contains("Thread(s) per core")) {
                            int res = (new Scanner(linea)).
                               useDelimiter("\\D+").nextInt();
                            aux.setNumThreads(res);
                        } else if (linea.contains("Temp")) {
                            linea = br.readLine();
                            int res = new Scanner(linea).nextInt();
                            aux.setTemperatura(res / 1000);
                        }
                    }
                    raspis.setList(aux);
                } catch (IOException ex) {
                    Logger.getLogger(ServerThread.class.getName()).
                    log(Level.SEVERE, null, ex);
                } finally {
                    try {
                        if (fr != null) {
                            fr.close();
                        }
                    } catch (Exception e2) {
                        e2.printStackTrace();
                    }
                }
            }
        }
}
```

9.2 - Ejecución de escenarios en el *cluster*

En esta sección se detallan los pasos necesarios para la ejecución de los escenarios en el *cluster*.

9.2.1 - Recepción de ficheros en el front-end

Tras establecer la conexión el cliente con el servidor, se reciben en el front-end los archivos *.ini* y *.ned,* necesarios para la simulación. Para disminuir el tráfico en la red de comunicaciones cabe destacar que los archivos han sido comprimidos por el cliente antes de ser enviados. La carpeta recibida se descomprime en el subdirectorio */filesNeeded/*, mencionado en el apartado 9.1, mantiene como nombre la IP del cliente para así poder identificar a qué máquina hay que enviar los resultados de la simulación una vez obtenidos.

De esta tarea se encargan los métodos *descomprimirArchivo()* y *descomprimir()*. Ambos métodos funcionan de manera inversa a los ya explicados en el capítulo anterior *sendComprimido()* y *comprimir()*.

```
  private void descomprimirArchivo() throws FileNotFoundException, IOException
{
        BufferedInputStream bis;
        BufferedOutputStream bos;
        byte[] receivedData;
        int in;
        String file;
        bis = new BufferedInputStream(sc.getInputStream());
        receivedData = new byte[1024];
        DataInputStream dis = new DataInputStream(sc.getInputStream());

        //Recibimos el nombre del fichero
        file = dis.readUTF();
        file = file.substring(file.indexOf('\\') + 1, file.length());

        //Para guardar fichero recibido
        FileOutputStream fos = new
FileOutputStream("../descomprimir/"+sc.getRemoteSocketAddress()+".zip");
        bos = new BufferedOutputStream(fos);
        while ((in = bis.read(receivedData)) != -1) {
            bos.write(receivedData, 0, in);
        }
        bos.close();
        dis.close();
```

```
        descomprimir(file);
        repartirFicheros(file);
    }
```

```
private void descomprimir(String path) {
        //cadena que contiene la ruta donde están los archivos .zip
        String directorioZip = path;
        //ruta donde están los archivos .zip
        File carpetaExtraer = new File(directorioZip);

        //valida si existe el directorio
        if (carpetaExtraer.exists()) {
            //lista los archivos que hay dentro  del directorio
            File[] ficheros = carpetaExtraer.listFiles();

            //ciclo para recorrer todos los archivos .zip
            for (int i = 0; i < ficheros.length; i++) {
                try {
                    //crea un buffer temporal para el archivo que se va
descomprimir
                    ZipInputStream zis = new ZipInputStream(new
FileInputStream(directorioZip + ficheros[i].getName()));

                    ZipEntry salida;
                    //recorre todo el buffer extrayendo uno a uno cada
archivo.zip y creándolos de nuevo en su archivo original
                    while (null != (salida = zis.getNextEntry())) {
                        FileOutputStream fos = new
FileOutputStream(directorioZip + salida.getName());
                        int leer;
                        byte[] buffer = new byte[1024];
                        while (0 < (leer = zis.read(buffer))) {
                            fos.write(buffer, 0, leer);
                        }
                        fos.close();
                        zis.closeEntry();
                    }
                } catch (FileNotFoundException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Directorio de salida: " + directorioZip);
        } else {
            System.out.println("No se encontró el directorio..");
        }
    }
```

Una vez que se han descomprimido correctamente los ficheros, el método *repartirFicheros()* se encarga de seleccionar el nodo de cómputo al que enviar los ficheros, la ejecución del script encargado de lanzar la simulación. Este script también se encarga de enviar los resultados obtenidos al cliente.

El nodo de cómputo es seleccionado mediante un algoritmo de Round Robin[20]. Es de esto de lo que se encarga la variable *indice*, la cual irá incrementando cada vez que se ejecute el método. Esta variable no puede ser mayor que el número de nodos de cómputo del *cluster*, por ello, cuando el valor sea igual índice volverá a valer cero.

```
private void repartirFicheros(String file) throws IOException {
        Process proc = null;
        String worker = workersIP[indice];
        proc = Runtime.getRuntime().exec("./sendSimulations" + worker + " " +
file);
        }
```

9.2.2 - *Gestión de la simulación*

Para simplificar la explicación de cómo se ejecuta la simulación, dividimos este proceso en tres apartados: envío de ficheros, ejecución de la simulación y obtención de los resultados.

Estos tres apartados se llevan a cabo en el siguiente script, el cuál explicaremos paso a paso en los siguientes puntos. En el apartado 9.2.3 se explica el envío de los ficheros de configuración al nodo, en el apartado 9.2.4 se explica cómo se lleva a cabo la simulación y en el apartado 9.2.5 se explica cómo se lleva a cabo la obtención de resultados en el front-end. Por último, en el apartado 9.3 se explica cómo se envían los resultados de la simulación al cliente que la solicitó.

```
#!/bin/bash

scp -p frontend@169.254.12.1: $HOME/SocketsServer/filesNeeded/$1/
worker@$0: $HOME/Simcan/simulations/

rsh -l pi $0 runSimulation
```

[20] https://es.wikipedia.org/wiki/Planificaci%C3%B3n_Round-robin

```
scp -p worker@$0: $HOME/Simcan/Simulation/$1/ frontend@169.254.12.1
$HOME/SocketsServer/filesNeeded/$1/

java sendResults $1
```

Donde `$0` es la IP del nodo destino que va a ejecutar la simulación la cual se le pasa por parámetro al script y `$1` es la carpeta de ficheros que se tienen que mandar al worker.

9.2.3 - Envío de ficheros a los nodos

Una vez que se ha descomprimido la carpeta se deben enviar los ficheros a un nodo worker para llevar a cabo su simulación. El nodo de cómputo al que se envía la información ha sido seleccionado anteriormente, como se explica al final del apartado 9.2.1. Los ficheros se envían al nodo mediante SCP al ejecutar la siguiente línea del script. Donde la primera ruta corresponde al origen de los ficheros en el front-end y la segunda ruta es la ruta destino en el worker.

```
scp -p frontend@169.254.12.1: $HOME/SocketsServer/filesNeeded/$1/
worker@$0: $HOME/Simcan/simulations/
```

9.2.4 - Ejecución de la simulación

Una vez que se han recibido los ficheros, la siguiente línea está destinada a la ejecución de la simulación. Esta ejecutará el script `runSimulation` que se encuentra en cada nodo de cómputo y que ejecutará la simulación con los ficheros que han sido enviados.

```
rsh -l pi $0 runSimulation $1
```

A continuación se muestra el script `runSimulation` encargado de configurar el path correcto al proyecto que se quiere simular y lanzar a ejecución el proyecto de simulación.

```
export OMNETPP_HOME=$HOME/omnetpp-5.2.1
```

```
export INET_HOME=$OMNETPP_HOME/projects/inet
export SIMCAN_HOME=$OMNETPP_HOME/projects/simcan/$0
./run_release -u Cmdenv
```

9.2.5 - *Obtención de resultados*

Una vez que ha terminado la ejecución de la simulación se ejecuta la línea encargada de obtener los resultados de la simulación en el front-end para así enviarlo de nuevo al cliente.

```
scp -p worker@$0: $HOME/Simcan/Simulation/$1/ frontend@169.254.12.1
$HOME/SocketsServer/filesNeeded/$1/
```

9.3 - Envío de resultados al cliente

Finalmente, la última línea del script, se encarga de ejecutar un programa en Java, el cuál recibe por parámetro la ip del cliente para saber qué carpeta ha de comprimir y a qué cliente ha de enviar los resultados obtenidos.

```
java sendResults $1
```

La clase `sendResults` tiene una funcionalidad similar a la del Cliente a la hora de gestionar el envío de ficheros. Comprime los archivos con los resultados, en este caso los que tengan extensión *.rca,* y una vez comprimido los envía mediante un socket al cliente con la IP que le ha llegado por parámetro.

```
public static boolean sendComprimido(Socket sc, String path, String ip)
throws IOException {
        sc = new Socket(ip, puerto);
        int entry;
        byte[] byteArray;
        BufferedInputStream bis;
        BufferedOutputStream bos;
        DataOutputStream out;
        DataInputStream in;
        File localFile;

        String filename = comprimir(path);
        if(ret.equalsIgnoreCase(""))
           return false;
        localFile = new File(filename);
```

```
        bis = new BufferedInputStream(new FileInputStream(localFile));
        bos = new BufferedOutputStream(sc.getOutputStream());
        out = new DataOutputStream(sc.getOutputStream());
        in = new DataInputStream(sc.getInputStream());

        while (!in.readUTF().equalsIgnoreCase("accepted"));
        //Enviamos el nombre del fichero y el fichero
        out.writeUTF(localFile.getName());
        byteArray = new byte[8192];
        while ((entry = bis.read(byteArray)) != -1) {
            bos.write(byteArray, 0, entry);
        }
        bis.close();
        bos.close();
        sc.close();
        return true;
    }
```

10 - Conclusiones y trabajo futuro

En este capítulo se desarrollan las conclusiones y las líneas de trabajo futuro de este proyecto.

10.1 - Conclusiones

En esta sección se detallan las conclusiones, tanto generales, como individuales, de cada participante del proyecto.

10.1.1 - Conclusiones Generales

Este proyecto ha supuesto un reto tanto a nivel académico como a nivel personal. Se han puesto a prueba múltiples conocimientos obtenidos en el grado de Ingeniería de Computadores, dando como resultado el desarrollo de este Trabajo de Fin de Grado.

Los integrantes de este trabajo no hemos tenido la oportunidad de utilizar una Raspberry Pi antes de este proyecto. Familiarizarse con el entorno y configurar un sistema desde cero para un fin en concreto ha sido una tarea importante. Esto nos ha permitido asegurar que el estudio y la investigación previa a la ejecución de cualquier avance, por pequeño que sea, es una parte necesaria del desarrollo.

10.1.2 - Conclusiones por Pablo Román Morer Olmos

La utilización de gráficas permite entender de forma intuitiva el comportamiento de distintos procesos ejecutados en un entorno distribuido. De esta forma, el usuario que lo desee puede utilizar esta herramienta con cualquier aplicación MPI, empleando distintas configuraciones y, así, obtener sus propias conclusiones.

Por otra parte, los problemas imprevistos que han ido surgiendo a lo largo del desarrollo del proyecto también se han solventado de manera exitosa. Por ejemplo, la mejora de la biblioteca *TraceLib*, aumentando su compatibilidad en distintos entornos.

Como conclusión final, el objetivo de desarrollar una herramienta para el estudio del rendimiento de aplicaciones MPI en sistemas distribuidos se ha cumplido de forma satisfactoria, demostrando que el software desarrollado se puede llevar a las aulas

para dar mayor visibilidad a las aplicaciones distribuidas, pudiendo así, facilitar su comprensión y entendimiento.

10.1.3 - Conclusiones Miguel Pérez de la Rubia

Algo que me ha motivado a realizar este proyecto ha sido la visión de que esto sirva para futuras generaciones de alumnos, ayudando al estudio y comprensión de un sistema distribuido. Aún siendo a pequeña escala, puede visualizarse el funcionamiento y la aplicación de las asignaturas cursadas.

El primer reto que enfrenté fue la configuración de las placas Raspberry Pi. Puesto que desconocía el usuario y contraseña para acceder al sistema y no sabía si había sido modificado, tuve que reinstalar el sistema operativo perdiendo la configuración previa.

A continuación comenzó el desarrollo de la aplicación en Java. El siguiente problema al que me enfrenté apareció aquí, al generarse un error del que no era capaz de hallar el origen. Finalmente se solucionó el problema, tras varios días probando distintas configuraciones, cambiando en el cliente la ruta de los ficheros *.java*, los cuales se tenían que encontrar en el la carpeta generada por defecto por NetBeans y no en una subcarpeta.

Por último, el mayor reto que he enfrentado ha sido la simulación de los entornos generados. Debido a problemas con las dependencias de librerías de Python para el uso de OMNeT++ y Raspberry Pi OS Lite no ha sido posible terminar esta parte del proyecto. Siendo aquí donde más tiempo se ha empleado probando los distintos sistemas operativos proporcionados por Raspberry Pi y las distintas versiones de OMNeT++ para intentar solucionarlo pero sin ser capaz de conseguir una solución.

Aún no habiendo podido terminar satisfactoriamente esta parte del proyecto, sí que considero que este trabajo me ha servido para aprender sobre el desarrollo de software de manera independiente y no bajo un guión de prácticas.

10.2 - Trabajo Futuro

Para este proyecto se pueden realizar diferentes ampliaciones que mejorarían su utilidad en un futuro próximo, por ejemplo:

- Mejorar la conectividad entre las aplicaciones cliente y el servidor utilizando un intercambio de paquetes menos pesado que permita la portabilidad del cliente.
- Modificar la biblioteca *TraceLib* para que no sea necesario cambiar la sintaxis de los nombres de las funciones MPI y E/S originales. Esta mejora aportará comodidad a la hora de seleccionar las aplicaciones MPI, ya que el usuario no tendrá que añadir el sufijo *_trace* en las distintas llamadas a estas funciones.
- Solucionar los problemas encontrados entre OMNeT++ y Raspberry Pi OS Lite.
- Implementar distintos planificadores de carga de trabajo para la aplicación de gestión de simulaciones como FIFO y Shortest Process Next, entre otros.
- Desarrollo de una red segura bajo mensajes cifrados para el acceso al servidor y mejora del acceso a la parte cliente.
- Modificar el sistema de ficheros de cliente para generar una red local de archivos.

10- Conclusions and future work

This section presents the conclusions and some lines of future work.

10.1- Conclusions

This section details the overall and individual conclusions of each participant in the project.

10.1.1- General Conclusions

This project has been challenging both academically and personally. A great deal of knowledge obtained in the Computer Engineering degree has been put to test, resulting in the development of this Bachelor's thesis.

The members of this project did not had the time to use a Raspberry Pi before. Familiarising ourselves with the environment and configuring a system from scratch to a specific purpose have been important. This has allowed us to settle the study and the previous research to any advance, no matter how small, is a necessary part of the development.

10.1.2- Conclusion by Pablo Román Morer Olmos

The use of graphics allows us to intuitively understand the behaviour of different processes executed in a distributed system. In this way, the final users can use the application of their choice, using different configurations for the same MPI application and thus obtain their own findings.

On the other hand, the unintended problems that have arisen through the development of the project have also been successfully solved. For example, the improvement of the *TraceLib* library, increasing the compatibility with different environments.

As a final conclusion, the goal of developing a tool for studying the performance of MPI applications has been successfully achieved. By achieving this goal, it is shown that the

developed software can be used in classrooms, facilitating the understanding and comprehension of distributed applications.

10.1.3- Conclusions by Miguel Pérez de la Rubia

Something that has motivated me to carry out this project has been the vision that it will be useful for future generations of students, helping them to study and understand a distributed system. Even though it is on a small scale, the operation and application of the different subjects studied can be seen in this project.

The first challenge I faced was the configuration of the Raspberry Pi. Since I did not know the username and password to access the system or they had been modified, I had to reinstall the operating system, losing the previous configuration.

Then, the development of the Java application started. The next problem I dealt with appeared here when an error was generated and I couldn't find the source. Finally the problem was solved, after several days of trying different options, by changing the path of the *.java* files in the client, which had to be found in the default NetBeans auto generated folder and not in a subfolder.

Finally, the biggest challenge I faced was the simulation of the generated environments. Due to problems between Python library dependencies, for the use of OMNeT++, and Raspberry Pi OS Lite it has not been possible to finish this part of the project. Being here where more time has been spent testing the different operating systems provided by Raspberry Pi and the different versions of OMNeT++ to try to solve it but without being able to find a solution.

Although I have not been able to finish this part of the project satisfactorily, I consider that this work has helped me to learn about software development in an independent way and not under any practical guidelines.

10.2- Future work

For this project, we found several extensions that can be done to improve its usefulness in the future. Some of these extensions can be:

- Improve the connectivity between the client and server applications by using a faster connection protocol to use less bandwidth.
- Modify the TraceLib library so it is not necessary to change the syntax of the original MPI and I/O function names. This improvement will provide convenience when selecting MPI applications, as the user will not have to add the suffix _trace in the different calls to these functions.
- Fix the incompatibility between OMNeT++ and Raspberry Pi OS Lite.
- Implement different schedulers for the simulation management such as FIFO, Shortest Process Next, among others.
- Development of a secure network under encrypted messages for access to the server and improved access to the client-side.
- Use a local network file system in the cluster to reduce the network traffic.

Apéndice A

Aportación común de los miembros

En conjunto se ha realizado el capítulo 1 de introducción y el capítulo 2 de elementos del sistema, siendo Pablo Morer el que facilitó el modelo para el desarrollo de las figura 2 y 3, y el que generó la figura 1 del esquema general de la propuesta. Así mismo, el trabajo de revisión y corrección ha sido llevado a cabo también de manera conjunta.

Aportación individual de cada miembro

A.1 Pablo R. Morer Olmos

Durante la fase de configuración del sistema distribuido, sobre 4 Raspberry Pi, se ha encargado de:

- Instalación y configuración de la herramienta MPI en el front-end para ejecutar aplicaciones en paralelo.
- Instalación y configuración de Java para la compilación y ejecución de aplicaciones MPI.
- Configuración de los nodos de cómputo.

En cuanto a la fase de desarrollo software:

- Estudio y análisis de la biblioteca *TraceLib*.
- Modificaciones en la biblioteca *TraceLib* que llevan a cabo su correcta implementación en un sistema Linux.
- Diseño e implementación de las aplicaciones *Client_Applicatión* y *Server_Application*.
- Creación en un script de bash para la instalación de la aplicación *Server_Application* en cualquier Raspberry Pi.
- Realización de pruebas y experimentos necesarios para asegurar que la parte *Client_Application* y *Server_Application* funcionan de forma correcta.

En cuanto a la fase de documentación, ha desarrollado:

- Guía de instalación Java en una Raspberry Pi

Junto a este trabajo se han desarrollado los capítulos 4, 5 y 6. Estos aportación contiene:

- Explicación del funcionamiento de la biblioteca *TraceLib*.
- Documentación de las modificaciones realizadas en la biblioteca *TraceLib*.
- Documentación de la aplicación *Client_Application*.
- Documentación y guía de instalación de la aplicación *Server_Application*.
- Documentación gráfica de pruebas realizadas sobre aplicaciones MPI en un entorno distribuido.

A.2 Miguel Pérez de la Rubia

Durante la fase de configuración del sistema distribuido, sobre cuatro Raspberry Pi, se ha encargado de:

- Realizar la instalación del sistema operativo Raspberry PI OS lite en las cuatro Raspberry Pi dedicadas a la simulación de los entornos generados.
- Configuración de IPs estáticas.
- Instalación de Java sobre el front-end usado para el desarrollo de la aplicación SiGSED.
- Instalación de RSH y la activación de SCP,en las Raspberry Pi usadas por SiGSED, incluyendo en esta instalación, la configuración de los archivos de red pertinentes para el correcto funcionamiento.
- Modificación del sistema de arranque del sistema operativo para el arranque de la máquina sin salida gráfica.

Debido a los problemas encontrados en la instalación de OMNeT++ para el uso SIMCAN en los nodos de cómputo, ha realizado pruebas sobre los distintos sistemas operativos que ofrece la compañía Raspberry Pi para comprobar si era posible la instalación sobre alguno de estos.

Durante la fase de desarrollo software ha implementado:

- Cliente y Servidor de la aplicación SiGSED.

- Scripts usados en el *cluster* para la ejecución de simulaciones.

Durante la fase de documentación ha desarrollado:

- Guía de configuración sobre:
    - IPs estáticas en las placas Raspberry Pi.
    - Guía de instalación de RSH y activación de SCP en el front-end y en los nodos de cómputo usados para el desarrollo de la aplicación SiGSED.
    - Guías de instalación de OMNeT++, INET++, SIMCAN.

Unido a este trabajo se encuentra también el desarrollo de los capítulos 7, 8 y 9. Esta aportación implica los siguientes puntos:

- Introducción al simulador SIMCAN, explicando características del mismo y en qué parte del sistema se aplica cada una.
- Documentación de la parte cliente de la aplicación SiGSED para su funcionamiento junto con SIMCAN-GUI.
- Documentación de la parte servidor de la aplicación SiGSED, junto con los scripts utilizados para el funcionamiento de SIMCAN.

Bibliografía

Información acerca de Raspberry para su instalación y configuración:

https://www.raspberrypi.org/software/

https://raspberry-projects.com/pi/category/pi-hardware

https://www.domocasainteligente.com/ip-estatica-en-raspberry-pi-raspbian/

Información acerca SIMCAN para su instalación

http://antares.sip.ucm.es/cana/simcan

Información acerca RSH para su instalación

https://www.mkssoftware.com/docs/man1/rsh.1.asp

Información acerca de SCP para su instalación

https://linux.die.net/man/1/scp

Información acerca de la librería *TraceLib* y generación de gráficas para aplicaciones MPI por Bryan Raúl Vaca Vargas

https://eprints.ucm.es/id/eprint/50220/1/027.pdf

Información acerca de Java para desarrollo de aplicaciones

https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

https://javiergarciaescobedo.es/programacion-en-java/15-ficheros/358-archivo-de-propiedades-properties

https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

Información acerca de JSON

https://www.json.org/json-es.html

Información acerca de Linux

https://man7.org/linux/man-pages/man2/readlink.2.html

http://www.chuidiang.org/clinux/herramientas/librerias.php

https://linux.die.net/man

Información acerca de sistemas operativos para conocer sus núcleos y sistemas de ficheros

https://es.wikipedia.org/wiki/Unix_File_System

https://es.wikipedia.org/wiki/Apple_File_System

https://cygwin.com/install.html

https://es.wikipedia.org/wiki/Windows_NT

Información acerca de MPI

https://www.open-mpi.org/

https://es.wikipedia.org/wiki/Interfaz_de_Paso_de_Mensajes

https://www.mpich.org