# Universidad Carlos III de Madrid

*Escuela Politécnica Superior*

Ingeniería Informática

*Proyecto Fin de Carrera*

# Sistema de particionado de modelos de simulación para ejecutar simulaciones en paralelo

**Autor:** Alejandro Carrasco Gil

**Director:** Alberto Núñez Covarrubias

**Fecha:** Septiembre 2011

Dedicado a mis padres Francisco y Manuela,

a mi hermano Carlos

y a toda mi familia y amigos

# Índice de contenido

1. Introducción ..... 1

1.1. Objetivos ..... 1

1.2. Estructura del documento ..... 3

2. Análisis del problema ..... 5

2.1. Identificación de necesidades ..... 5

2.2. Estudio de viabilidad ..... 5

2.3. Ejemplo ..... 6

3. Análisis del sistema propuesto ..... 11

3.1. Definición de la arquitectura propuesta ..... 11

3.2. Descripción funcional del sistema ..... 12

3.3. Requisitos del sistema ..... 14

4. Diseño ..... 17

4.1. Diseño de componentes ..... 17

4.2. Diseño de clases ..... 20

4.2.1. Diagrama de clases ..... 20

4.2.2. Implementación ..... 27

4.3. Interfaces ..... 47

4.4. Estructuras ..... 49

4.5. Plataforma de desarrollo ..... 52

5. Algoritmos ..... 55

5.1. Proximity partition algorithm ..... 55

5.2. Simple balanced partition algorithm ..... 57

5.3. Balanced partition algorithm ..... 59

6. Pruebas ..... 63

6.1. Plan de pruebas ..... 63

6.2. Entorno de pruebas ..... 66

6.3. Resultados ..... 68

6.4. Comparativas ..... 98

7. Conclusiones ........................................................................................................ 103

Bibliografía .............................................................................................................. 107

Anexo I. Presupuesto ................................................................................................ 109

Anexo II. Manual de usuario ..................................................................................... 113

Anexo III. Glosario de términos ................................................................................ 119

# Índice de figuras

Figura 1. Modelo de simulación ........ 7

Figura 2. Ejemplo particionamiento ........ 8

Figura 3. Arquitectura de 3 capas ........ 11

Figura 4. Componente de datos ........ 12

Figura 5. Componentes de la arquitectura ........ 17

Figura 6. Componente Presentación ........ 18

Figura 7. Componente Negocio ........ 18

Figura 8. Componente Modelo ........ 19

Figura 9. Componente Entrada ........ 19

Figura 10. Componente BBDD ........ 20

Figura 11. Diagrama de clases ........ 21

Figura 12. Diagrama de clases por componente ........ 22

Figura 13. Clases componente Presentación ........ 23

Figura 14. Clases componente Entrada ........ 23

Figura 15. Clases componente Negocio ........ 24

Figura 16. Clases componente Modelo ........ 25

Figura 17. Clases componente Modelo ........ 26

Figura 18. Clases componente Modelo ........ 27

Figura 19. Interfaz capa Presentación ........ 47

Figura 20. Interfaz capa negocio ........ 48

Figura 21. Interfaz capa datos ........ 48

Figura 22. Enlace de modelo ........ 49

Figura 23. Enlace de grafo ........ 50

Figura 24. Cálculo peso enlace modelo ........ 50

Figura 25. Cálculo peso enlace grafo ........ 51

Figura 26. Ejemplo de partición ........ 52

Figura 27. Logo de Java ........ 52

Figura 28. Logo de Apple Inc. ........ 53

Figura 29. Logo de Linux ........ 53
Figura 30. Diagrama creación dominios ........ 55
Figura 31. Diagrama creación de grafos ........ 58
Figura 32. model_presentation ........ 63
Figura 33. model_1 ........ 64
Figura 34. model_2 ........ 65
Figura 35. Tabla de pruebas ........ 65
Figura 36. Logo proyecto SIMCAN ........ 66
Figura 37. Arquitectura SIMCAN ........ 67
Figura 38. Prueba 1 ........ 68
Figura 39. Prueba 2 ........ 69
Figura 40. Prueba 3 ........ 70
Figura 41. Prueba 4 ........ 71
Figura 42. Prueba 5 ........ 72
Figura 43. Prueba 6 ........ 73
Figura 44. Prueba 7 ........ 74
Figura 45. Prueba 8 ........ 75
Figura 46. Prueba 9 ........ 76
Figura 47. Prueba 10 ........ 77
Figura 48. Prueba 11 ........ 78
Figura 49. Prueba 12 ........ 79
Figura 50. Prueba 13 ........ 80
Figura 51. Prueba 14 ........ 81
Figura 52. Prueba 15 ........ 82
Figura 53. Prueba 16 ........ 83
Figura 54. Prueba 17 ........ 84
Figura 55. Prueba 18 ........ 85
Figura 56. Prueba 19 ........ 86
Figura 57. Prueba 20 ........ 87

Figura 58. Prueba 21 ........ 88

Figura 59. Prueba 22 ........ 89

Figura 60. Prueba 23 ........ 90

Figura 61. Prueba 24 ........ 91

Figura 62. Prueba 25 ........ 92

Figura 63. Prueba 26 ........ 93

Figura 64. Prueba 27 ........ 94

Figura 65. Prueba 28 ........ 95

Figura 66. Prueba 29 ........ 96

Figura 67. Prueba 30 ........ 97

Figura 68. Gráfica rendimiento *model_presentation* con *los 3 algoritmos* ........ 98

Figura 69. Gráfica speedup *model_presentation* con *los 3 algoritmos* ........ 99

Figura 70. Gráfica rendimiento *model_1* con *los 3 algoritmos* ........ 99

Figura 71. Gráfica speedup *model_1* con *los 3 algoritmos* ........ 100

Figura 72. Gráfica rendimiento *model_2* con *los 3 algoritmos* ........ 100

Figura 73. Gráfica speedup *model_2* con *los 3 algoritmos* ........ 101

Figura 74. Tabla de actividades ........ 109

Figura 75. Tabla costes por puesto ........ 109

Figura 76. Diagrama de Gantt ........ 110

Figura 77. Tabla costes personal ........ 111

Figura 78. Tabla costes recursos materiales ........ 111

Figura 79. Tabla presupuesto final ........ 112

Figura 80. Archivos de la aplicación ........ 113

Figura 81. Pantalla principal de la aplicación ........ 114

Figura 82. Botones de la aplicación ........ 114

Figura 83. Apartados de la aplicación ........ 115

Figura 84. Fichero de entrada de la aplicación ........ 116

# 1. Introducción

Desde los inicios del campo de la informática un aspecto importante y básico siempre ha sido y es el de mejorar el rendimiento tanto del hardware como del software. La mejora de este rendimiento continuamente viene dado por acortar los tiempos de ejecución de programas.

Los componentes hardware forman una parte substancial en el objetivo de conseguir optimizar el rendimiento. A principios de los 90 apareció el procesador Intel Pentium capaz de realizar 60 millones de operaciones por segundo en sus primeras versiones para terminar llegando a los 300 millones. Esto supone una mejora de más del 500% en rendimiento. En la década del 2000 el procesador Intel Pentium 4 consiguió una mejora del 300% desde sus comienzos hasta su final. Si bien, el hardware es un elemento esencial sobre el que investigar, es también un elemento en el que el margen de mejora está acotado, es decir, se llega a unos niveles en los que la curva de mejora del rendimiento empieza a disminuir para dejar de ser un crecimiento sostenido.

El proyecto está enfocado dentro del campo de los modelos de simulación. Se crearán diversos escenarios y entornos de simulación donde ejecutar el sistema para confeccionar un estudio pormenorizado sobre el impacto de cada particionado en dichos modelos.

## 1.1. Objetivos

El objetivo principal del proyecto es desarrollar una plataforma de particionado escalable y flexible.

Es por lo expuesto en el apartado anterior que el investigador ha de explorar nuevas vías para mejorar el rendimiento. Aquí es donde entran en juego conceptos como paralelización y distribución del trabajo. Lo que se pretende es conseguir que un problema grande se pueda dividir en n pequeños problemas que se ejecuten en paralelo. Las claves fundamentales para realizar lo explicado anteriormente están en usar varios procesadores y sobre todo fraccionar el trabajo en cada uno de ellos lo más eficazmente posible.

Uno de los objetivos del proyecto consiste en encontrar el particionamiento óptimo para cada modelo. Fragmentar más no siempre significa obtener mejores resultados.

El escenario de salida parte de la idea inicial de ejecutar un modelo de simulación de gran escala, lo cual en un ordenador personal de última generación es inviable por el tiempo tan largo que podría tardar. Para afrontar este inconveniente se hace uso del concepto de clúster o sistema distribuido. Esto nos va a permitir, gracias a la interconexión (red de comunicaciones)

de ordenadores, lograr un modelo distribuido. Este modelo será capaz de ejecutar el trabajo previsto en un tiempo prudencial arreglando el problema existente.

Una vez superado el primer inconveniente y definido el modelo sobre el que se ha de trabajar, entra en juego la manera en que se pone en funcionamiento dicho modelo. Para el desarrollo del proyecto se ha utilizado de la herramienta SIMCAN que permite la simulación de la ejecución de un modelo distribuido. Asimismo SIMCAN permite particionar el modelo en tantas particiones como el usuario quiera o necesite, para de esta manera ofrecer un mejor rendimiento y eficiencia. En este punto es donde este proyecto se concentra en el problema, la plataforma alcanzará un particionado óptimo del modelo a partir de una serie de algoritmos complejos. En razón de lo expuesto se obtendrá una mejora del tiempo de ejecución del trabajo y por tanto del rendimiento.

Seguidamente se explica con mayor detalle el funcionamiento de la plataforma propuesta. En primer lugar y en base a una entrada dada por el usuario se procede a examinar todos los datos. Esta entrada consistirá básicamente en un modelo y el número de particiones que se quieren adquirir. El modelo hace referencia a un sistema distribuido formado por nodos, redes, switches, etc. Se comprueba que el modelo es conexo y está bien formado. A continuación el usuario podrá elegir entre tres tipos de particionado, cada uno corresponde a un tipo de algoritmo determinado que ha sido estudiado e implementado. Por supuesto cada tipo de particionado favorece unas condiciones determinadas y dependiendo del tipo de modelo favorecerá más o menos al rendimiento. Una vez consignados todos estos datos la plataforma devolverá el resultado de cada partición, donde cada una de ellas contendrá elementos del modelo. En resumen, el objetivo final de la plataforma es obtener un modelo particionado en X particiones en base a unos algoritmos dados.

Un aspecto interesante a destacar es que los elementos con los que se ha trabajado en los algoritmos son nodos de cómputo, de almacenamiento y los elementos de interconexión (switches). En ningún momento la plataforma trabaja con información de la aplicación simulada en el entorno modelado.

Uno de los puntos fuertes por el cual se ha desarrollado esta plataforma es la mejora del rendimiento. El objetivo preferente del proyecto es el de obtener el mejor particionado posible para que el tiempo de ejecución sea mínimo. Para esto se ha trabajado con tres algoritmos diferentes, los cuales, todos consiguen mejores resultados en lugar de una ejecución sin particionar. Por extensión de lo anterior se puede decir que otras ventajas que el proyecto incluye son una mejor disponibilidad de datos y por ende una mejora de la escalabilidad. En definitiva, la plataforma podrá ejecutar trabajos de tamaño grande, cuya ejecución en ordenadores domésticos sería inasumible.

El principal inconveniente que tiene nuestra plataforma es la dependencia que tiene de la red y los elementos de comunicación, así como de la compatibilidad de todos los dispositivos. Es condición importante que estos elementos estén bien sincronizados y funcionen correctamente según sus especificaciones, lo que hace que la plataforma no presente problemas de eficiencia y pérdida de rendimiento.

Seguidamente, destacar que el proyecto ha sido implementado en el lenguaje de programación JAVA. El kit de desarrollo utilizado ha sido Java SE 6 Update 12 sobre la plataforma de desarrollo Eclipse Version 3.4.2. El que dicho lenguaje sea multiplataforma ha jugado un papel importante en su elección. Se ha considerado suficiente presentar nuestra plataforma con una interfaz textual.

## 1.2. Estructura del documento

La memoria está formada por los siguientes puntos. En el capítulo 1, se indica el marco en el que se engloba el proyecto y los objetivos que se persiguen. En el capítulo 2, se analizan las necesidades que han llevado a construir la plataforma, así como la viabilidad del proyecto y un ejemplo. En el capítulo 3, se presenta la arquitectura propuesta junto con la funcionalidad y requisitos del sistema. En el capítulo 4, se detalla el diseño de componentes y clases, además de las interfaces, estructuras y plataforma de desarrollo utilizadas. En el capítulo 5, se explican los tres algoritmos desarrollados para el sistema. En el capítulo 6, se presentan las pruebas y simulaciones realizadas sobre diferentes modelos, junto con sus resultados y gráficas comparativas. En el capítulo 7, se muestran las conclusiones a los resultados de las pruebas y las conclusiones globales del desarrollo del proyecto. El anexo I contiene el presupuesto del proyecto junto con el diagrama de Gantt. En el anexo II, se muestra el manual de usuario. Para terminar con el documento, el anexo III detalla el glosario de términos utilizados.

# 2. Análisis del problema

En esta sección se analiza el sistema de la plataforma de particionado. En primer lugar se procede a la identificación de las necesidades básicas del sistema, para hacerse una idea de lo que se va a precisar a la hora de encarar el proyecto. En base a estas primeras ideas, se realiza un estudio de viabilidad que verificará si es factible, en cuanto a costos de tiempo y dinero, hacer el sistema y ponerlo en funcionamiento. Seguidamente se pasa a hacer una breve descripción de las funciones que puede realizar nuestra aplicación, sin entrar en detalle en cada aspecto. Cosa que si se va a hacer en el último apartado, cuando se expliquen los requisitos del sistema.

## 2.1. Identificación de necesidades

Este proyecto surge a partir de la idea de mejorar las prestaciones de la plataforma SIMCAN. Se trata de una plataforma de simulación de alto rendimiento de trabajos grandes en paralelo. Nuestro sistema complementa al SIMCAN a la hora de dividir estos trabajos en varias particiones. La necesidad básica a cubrir es la de agregar nuevos algoritmos que mejoren el rendimiento de estas simulaciones, para ello el objetivo principal es el de desarrollar estos algoritmos, llevando un estudio pormenorizado de todas las variables que incluye el problema, para llegar a la mejor solución posible.

Un aspecto importante es el de la escalabilidad, será necesario que estos algoritmos estén preparados para conseguir buenos resultados tanto para problemas pequeños como para grandes problemas. Es decir, habrá que estudiar a fondo las diversas situaciones para conseguir que se pierda el menos rendimiento posible en problemas grandes.

Otra necesidad, es la de permitir una inclusión flexible de un algoritmo cualquiera. Se tendrá una interfaz gráfica sencilla e intuitiva que admita, con los menores cambios posibles, añadir en un futuro nuevos algoritmos estudiados.

## 2.2. Estudio de viabilidad

Todo proyecto requiere de un estudio de viabilidad previo para evaluar si es conveniente y productiva la realización de proyecto, o por el contrario, no es factible su elaboración por estar condenado al fracaso. En este apartado, se describirá la solución tomada a

partir de las propuestas planteadas para la realización del sistema. Se valorarán los pros y los contras, así como el impacto económico final.

Las necesidades del cliente son claras y concisas, necesita un desarrollo de tres algoritmo de partición y una interfaz gráfica para poder ejecutar dicho particionado. Se trata de un sistema que no va a ser común y por tanto los usuarios finales han de ser usuarios con ciertos conocimientos de informática. Será una aplicación específica pero que podrá ser flexible y agregar nuevos algoritmos. El cliente no nos exige ningún tipo de lenguaje de programación por lo que el equipo de trabajo lo escogerá a su libre elección. En duración del proyecto el cliente se muestra flexible y la única restricción será de una duración máxima de 1 año.

Económicamente realizar todo este montaje no supone un impacto muy grande para nuestra organización. Según las necesidades y restricciones del cliente, no sería necesario comprar equipos muy sofisticados ni contratar profesionales nuevos ya que se dispone de las personas necesarias y justas.

La organización, después de estudiar las necesidades del cliente y valorar diversas soluciones llega a la conclusión de que el proyecto es viable. Las restricciones de tiempo y funcionalidad del sistema son asumibles cien por cien, al igual que ocurre con el coste económico de todo el proyecto. Se presenta el presupuesto personalizado al cliente y a su aceptación se procede a empezar el desarrollo del sistema.

## 2.3. Ejemplo

En este apartado se va a explicar un ejemplo de aplicación de la plataforma. En primer lugar, se explicará de qué partes consta un modelo de simulación para terminar detallando un ejemplo de partición sencillo.

La figura 1 muestra la imagen de un modelo de simulación. Como se puede apreciar dispone de varios elementos interconectados que se pasan a detallar.

Los *communication devices* corresponden con elementos de interconexión de redes como pueden ser los *switches*. Estos elementos se pueden interconectar con otros *switches* o bien con *compute nodes* o *storage nodes*.

Los elementos cuyo nombre comienza por CN son *compute nodes*. Se corresponden con nodos de cómputo cuya función es la de procesar todo tipo de trabajos que reciban. A su vez, estos nodos contienen un número determinado de *cores* (32, 64 o 128). Cada núcleo representa un dispositivo de cómputo.

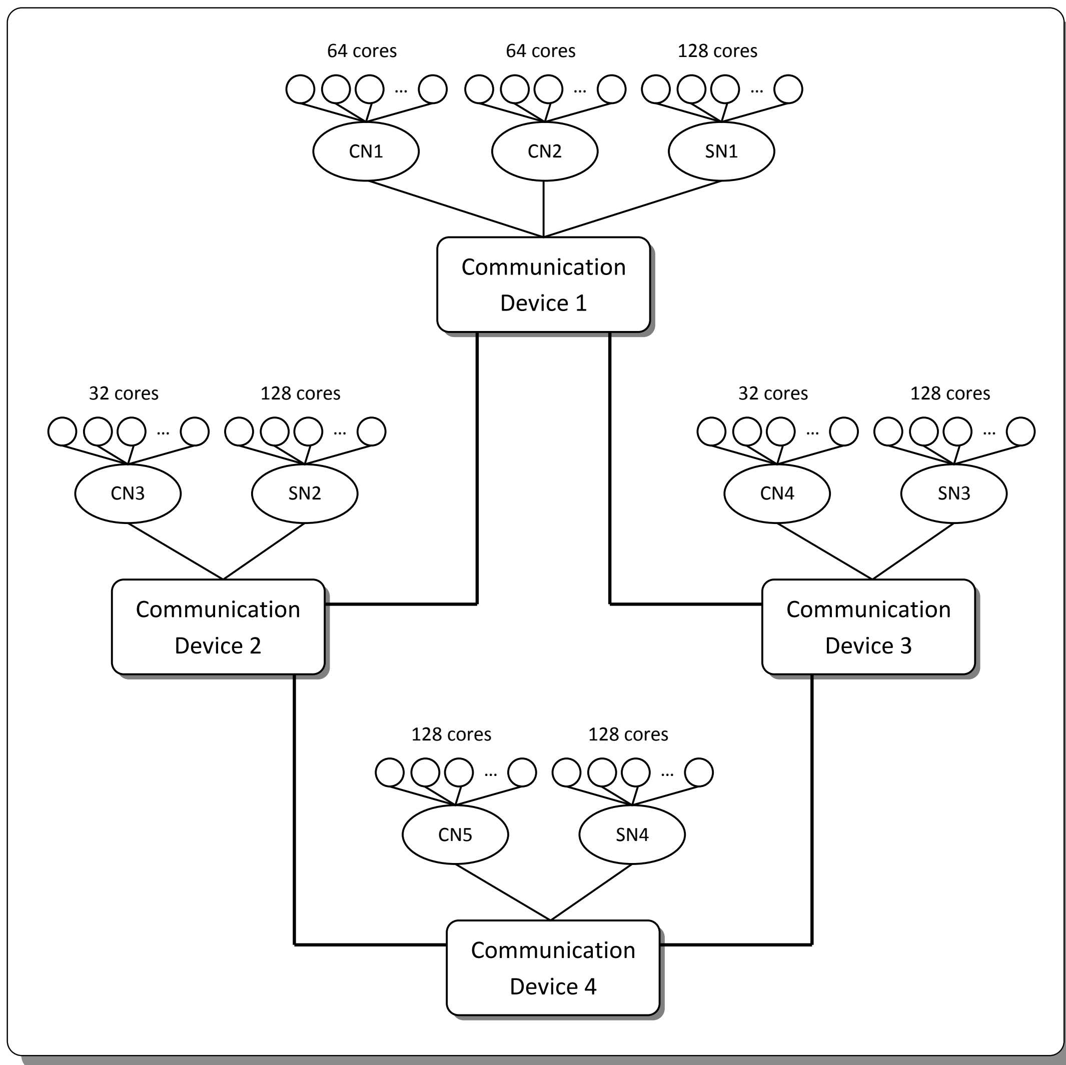


**Figura 1. Modelo de simulación**

Los elementos cuyo nombre empieza por SN son *storage nodes*. Se corresponden con nodos de almacenamiento cuya función es la de almacenar todos los datos necesarios de los trabajos en ejecución. A su vez, estos nodos contienen un número determinado de *cores* (32, 64 o 128). Cada núcleo representa un dispositivo de almacenamiento.

Después de explicar las partes de un modelo de simulación, se procede a presentar un ejemplo de particionado para el modelo propuesto en dos particiones.

La idea que utilizan los algoritmos de la plataforma es la de agrupar en dominios todos los nodos de cómputo y almacenamiento conectados a un *switch*. Una vez se tienen estos dominios se procede a la inserción de estos en cada partición según las especificaciones de cada algoritmo.

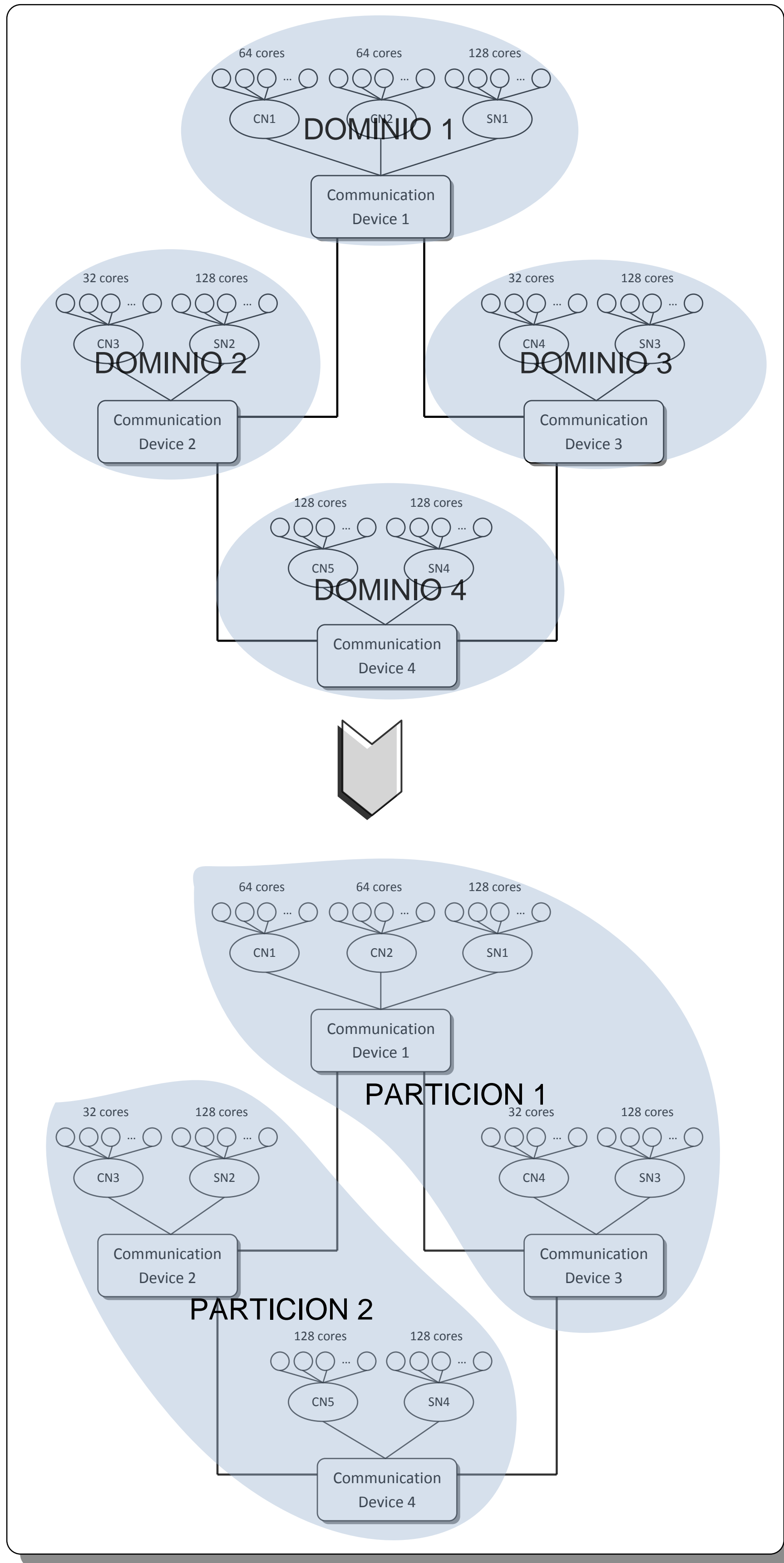


**Figura 2. Ejemplo particionamiento**

La figura 2 muestra un ejemplo de los pasos que se siguen para llegar a las particiones finales. El funcionamiento es sencillo, se escoge un dominio a insertar en una partición y según las especificaciones del algoritmo se inserta en la partición más óptima. Este paso se repite hasta que no quede ningún dominio sin asignar a alguna partición.

En el ejemplo de la figura 2, primero se inserta el Dominio 1 en la partición 1. En la siguiente iteración se escoge el Dominio 2 y se inserta en la partición 2. Seguidamente se elige el Dominio 4 y se inserta en la partición 2. Para acabar, el Dominio 3 se inserta en la partición 1.

# 3. Análisis del sistema propuesto

En este apartado se va a describir la arquitectura de la plataforma, así como la funcionalidad que ofrece el sistema y por último los requisitos necesarios de la aplicación.

## 3.1. Definición de la arquitectura propuesta

La arquitectura escogida para la plataforma ha sido la Arquitectura multicapa, en nuestro caso de 3 capas. Es una arquitectura cliente servidor, en la que el servidor se divide en dos capas distintas. Una, la capa de negocio que es el segmento que se ocupa de realizar todos los cálculos necesarios para obtener la salida del programa. Y dos, la capa de datos donde residen éstos y sus modelos lógicos en los que se estructuran y organizan. La parte del cliente se corresponde en nuestra arquitectura con la capa de presentación que se encarga de la interacción con el usuario. La figura 3 muestra un esquema de la arquitectura de la plataforma.

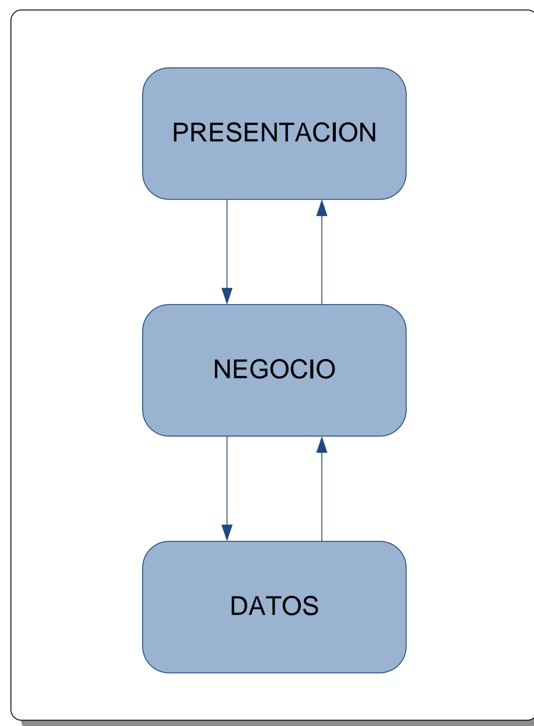


**Figura 3. Arquitectura de 3 capas**

La capa de presentación de la plataforma permitirá al usuario interactuar con el sistema, es decir, es el segmento encargado de facilitar al usuario la entrada de datos y de mostrar los resultados de las ejecuciones.

La capa de negocio será la autorizada para realizar todos los cálculos necesarios para obtener un resultado en base a la entrada dada por el usuario. El funcionamiento general de esta

sección será la de recoger los datos de entrada del usuario y en base a estos acceder a la capa de datos y poder realizar los cálculos oportunos.

Por último, la capa de datos será la que gestione las peticiones de la capa de negocios. Tendrá dos gestiones básicas, una acceder a la base de datos compuesta por un sistema de ficheros a partir de un modulo de entrada que lee los datos, los modela lógicamente y los devuelve a la capa de negocio. La otra gestión se corresponde con dar soporte lógico a la capa de negocio para crear las estructuras necesarias para el funcionamiento de la plataforma. Seguidamente en la figura 4 se muestra un croquis explicativo de la capa de datos.

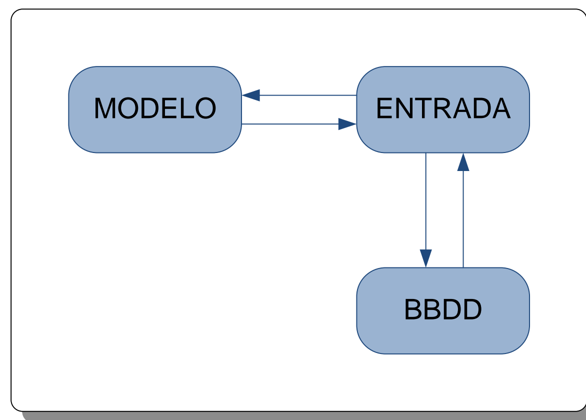


**Figura 4. Componente de datos**

## 3.2. Descripción funcional del sistema

En este apartado se va a hablar de las acciones que el usuario puede realizar en el sistema. En principio existe una funcionalidad básica que es la de realizar una ejecución sobre un algoritmo para obtener las particiones, se intentará dividir esta tarea en subtareas y detallar cada una de esta.

### *Cargar fichero*

En la parte superior izquierda de la aplicación está la sección para cargar un fichero de modelo en el sistema. Existen dos maneras de abrir el fichero, una primera que es introduciendo en un cuadro de texto la ruta donde se encuentra el fichero con el modelo. Y una segunda que es pulsando el botón de *Browse* y haciendo uso de un cuadro de dialogo de abrir ficheros elegir el fichero que se quiera. Si se elige la segunda opción, una vez se escoge el fichero automáticamente el cuadro de texto se rellena con la ruta correspondiente.

Si el usuario al escribir la ruta manualmente introdujera un nombre de fichero que no contiene un modelo, saldrá un mensaje de error en el cuadro de texto *Output*. Al igual ocurre, si un fichero de entrada tiene un modelo malformado.

### *Introducir número particiones*

Esta funcionalidad del sistema se encuentra en la parte central izquierda de la interfaz gráfica. Se provee al usuario de un cuadro de texto en el que puede introducir el número de particiones que desee. La aplicación detectará si se escribe un número o no, en caso de escribir una palabra se mostrará el mensaje de error correspondiente en el cuadro de texto *Output*.

### *Elegir algoritmo*

En la parte inferior izquierda de la interfaz, existe un botón de opción con tres elecciones que corresponden cada una a un algoritmo. El usuario no tiene más que marcar una de las tres opciones y automáticamente ya se tiene escogido el algoritmo a ejecutar. En esta funcionalidad no tienen cabida los mensajes de error.

### *Ejecutar algoritmo*

En la parte inferior central de la interfaz justo debajo del botón *Clean* se encuentra el botón *Execute*. La funcionalidad de este botón engloba el propósito principal del desarrollo del sistema. Al pulsarlo se ejecuta el algoritmo sobre el modelo y con el número de particiones escogido. Una vez termina la ejecución se muestra el resultado final en el cuadro de texto *Output*, situado en la parte derecha de la interfaz.

La manera de mostrar el resultado es la siguiente. Se muestra primeramente una línea de texto en la que aparece la frase: "================PARTITIONS================". Seguidamente se muestran las particiones, cada partición tendrá por título la frase: "Partition X" siendo X el número de partición. Una vez mostrado el título se muestra en cada línea cada uno de los elementos que pertenecen a dicha partición, solo se muestra el nombre de cada uno.

### *Limpiar cuadros de texto*

En la parte inferior central de la interfaz justo encima del botón *Execute*, se encuentra el botón *Clean*. La funcionalidad de este botón no es otra que la de limpiar y dejar vacíos de

cualquier tipo de carácter todos los cuadros de texto del sistema. Estos cuadros de texto son: el de la ruta del fichero de entrada, el cuadro de introducción del número de particiones y el cuadro de texto de salida del programa. En esta funcionalidad no tienen cabida los mensajes de error.

## 3.3. Requisitos del sistema

En este apartado se van a describir todos y cada uno de los requisitos del sistema necesarios para el desarrollo de éste.

- *Disposición de algoritmos de partición:*

  Dentro de la funcionalidad de la aplicación, es condición esencial que el usuario pueda tener la posibilidad de poder usar hasta tres tipos de algoritmo de partición.

- *Uso de modelos de entrada:*

  Es indispensable que el usuario pueda utilizar modelos de entrada para realizar las ejecuciones pertinentes. Podrá cargar ficheros que contengan dichos modelos.

- *Fijación del número de particiones:*

  Es condición imprescindible que el usuario pueda fijar como uno de los parámetros del problema el número de particiones. Serán únicamente válidos números enteros positivos.

- *Diseño escalable y flexible:*

  Es condición deseable que en un futuro se puedan agregar nuevos algoritmos a la plataforma. Es necesario por tanto, desarrollar el sistema de forma flexible de modo que si se quieren agregar nuevos algoritmos el impacto de esto sea el mínimo posible.

  A su vez es condición indispensable disponer de una plataforma escalable para no perder rendimiento ante problemas grandes, o intentar minimizar esta perdida.

- *Muestra de mensajes de error:*

  Es condición necesaria que el sistema muestre mensajes de error en los siguientes casos.

  - Fichero de entrada no encontrado.
  - Fichero de entrada no válido.
  - Número de particiones incorrecto.

- *Disponibilidad del manual de usuario:*

  Es condición indispensable que el usuario disponga del manual de usuario en todo momento para realizar cualquier tipo de consulta sobre el sistema.

- *Ejecución del sistema:*

  Es condición imprescindible activar el mecanismo correspondiente para ejecutar los algoritmos según los parámetros introducidos. Para ello el usuario dispondrá de un botón *Execute*.

- *Limpieza de cuadros de texto:*

  Es condición deseable que el usuario pueda limpiar los cuadros de texto de la aplicación de forma automática mediante un botón *Clean*. También tiene la posibilidad de limpiarlos manualmente.

- *Arquitectura del sistema:*

  Es condición indispensable implementar el sistema en base a una arquitectura de tres capas, teniendo como capas *Presentación*, *Modelo* y *Negocio*.

- *Interfaz gráfica:*

  Es condición deseable que el sistema disponga de una interfaz gráfica para introducir todos los parámetros y realizar las ejecuciones. En caso de no disponer de interfaz gráfica, es obligatorio implementar una interfaz textual.

- *Idioma interfaz:*

  Es condición deseable tener la aplicación con la opción de multi-idioma. Es obligatorio que el sistema esté en inglés o castellano.

- *División de funcionalidad en la interfaz:*

  Es condición indispensable delimitar dentro del sistema los apartados de entrada de datos y la salida al problema. En la parte izquierda de la interfaz se engloba la entrada de datos, como son cargar el modelo, introducir el número de particiones y la elección del algoritmo a ejecutar. En la parte derecha se encuentra el cuadro de texto *Output* donde se mostrará la solución al problema planteado.

- *Nivel informático de usuario:*

  Es condición deseable que el usuario tenga conocimientos medios o avanzados a nivel informático. Si bien la utilización de la aplicación es sencilla, es conveniente que aquel que la use pueda entender bien que hace cada algoritmo y sobre qué lo hace, para sacar un mejor rendimiento del sistema. Asimismo cabe destacar que no existirá ningún tipo de login al sistema ni varios tipos de usuario como administradores, clientes, etc. Solo existirá un usuario único.

- *Recursos terminal:*

  Es condición deseable que el terminal sobre el que se ejecute la aplicación disponga de un procesador de al menos 1 GHz y una memoria RAM de 256MB. Asimismo es imprescindible disponer de 5 Mb de disco duro para la instalación de la aplicación.

- *Sistema operativo:*

  Es condición indispensable disponer de un S.O. que permita el uso de la JVM. Sistemas que permiten esto son Windows, Mac Os, Linux, Android, etc.

- *Disposición de Java Virtual Machine:*

  Es condición imprescindible que el terminal donde se ejecute el sistema disponga de la JVM.

# 4. Diseño

Este apartado se va a dividir en dos partes. La primera en la que se va a detallar cada componente software del sistema y la segunda en la que se va a definir el diagrama de clases junto con todas las funciones que ofrecen.

## 4.1. Diseño de componentes

Una vez escogido el tipo de arquitectura y haber identificado las necesidades de la aplicación se procede a la parte del diseño. En primer lugar, es preciso conocer los subsistemas que van a funcionar en la aplicación. La figura 5 muestra la disposición de todos los componentes de la aplicación.

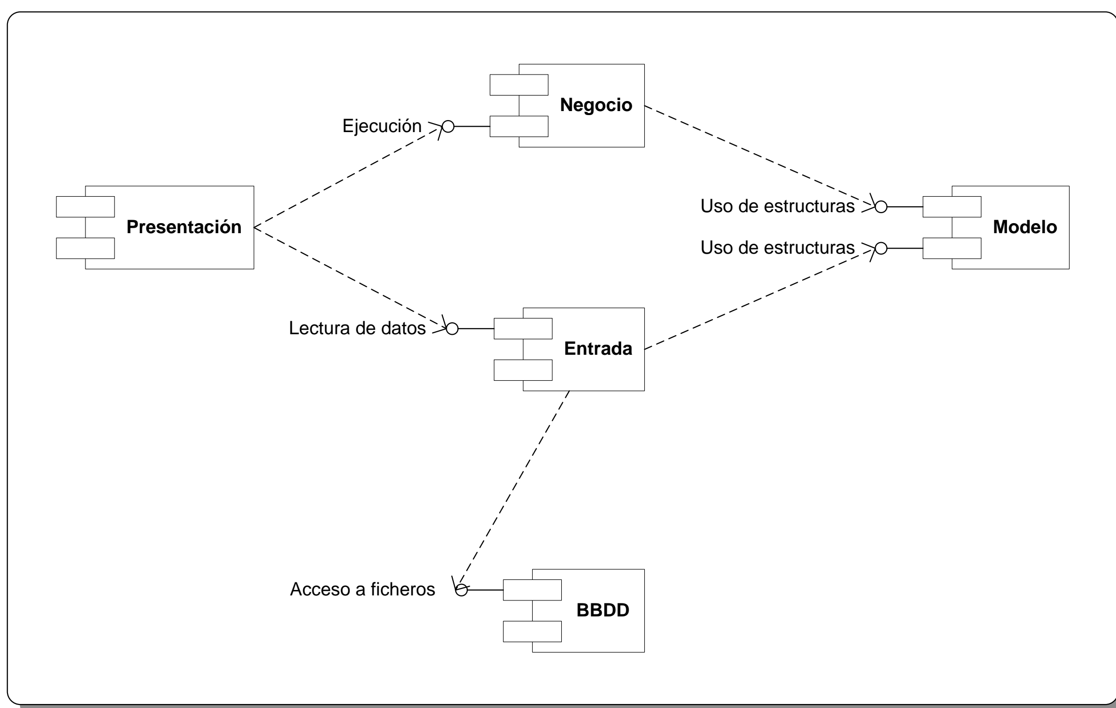


**Figura 5. Componentes de la arquitectura**

Según se ve en la figura 5, el diagrama mostrado de la aplicación dispone de cinco componentes, los cuales se van a explicar con más detalle a continuación.

El componente Presentación comprende la implementación de la interfaz gráfica del sistema, así como las llamadas necesarias a los componentes Negocio y Modelo. Del componente Negocio necesita la interfaz para la ejecución de los algoritmos de la aplicación, mientras que del componente Entrada requiere la interfaz capaz de leer los ficheros de entrada

para poder ejecutar dichos algoritmos. Dentro del componente Presentación se tiene una única clase que engloba la funcionalidad descrita anteriormente, esta clase es *ExecuteTests* mostrada en la figura 6.

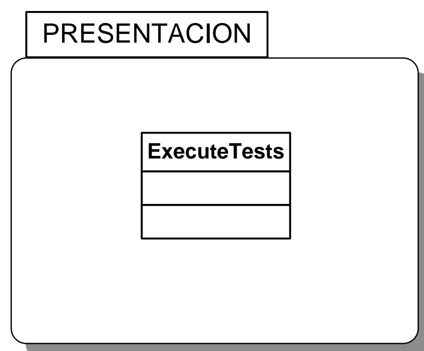


**Figura 6. Componente Presentación**

El componente Negocio incluye las clases concernientes con la funcionalidad principal del sistema. Dicho componente usa las estructuras proporcionadas por el componente Modelo para resolver el problema planteado a partir de los tres algoritmos existentes en el sistema. Además ofrece una interfaz al componente Presentación para que este ejecute el algoritmo seleccionado por el usuario. A continuación se muestra la figura 7 que comprende las clases de este componente que son: *ProximityPartitionAlgorithm, BalancedPartitionAlgorithm* y *SimpleBalancedPartitionAlgorithm*.

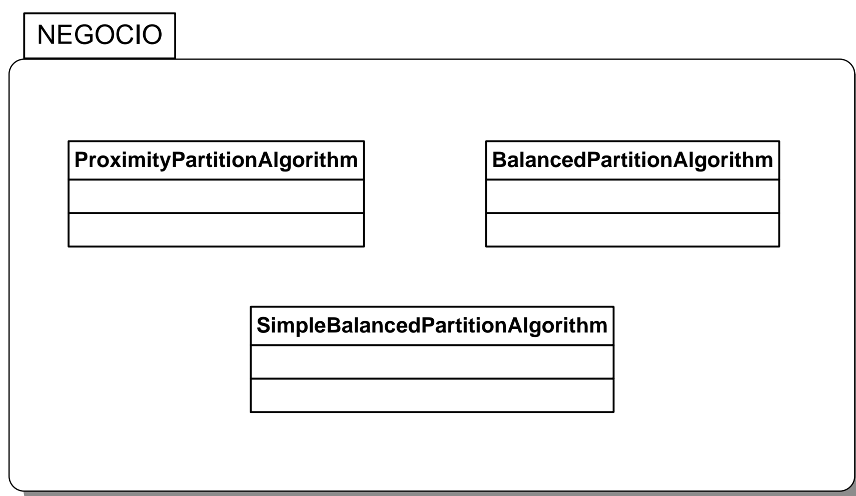


**Figura 7. Componente Negocio**

El componente Modelo contiene todas las estructuras necesarias, para que tanto el componente Negocio como el de Entrada puedan llevar a cabo sus funciones. Es, por tanto el soporte semántico sobre él se ejecutan los algoritmos. Proporciona a los componentes Negocio

y Entrada, las interfaces correspondientes a cada uno para que puedan hacer uso de estas estructuras. La figura 8 muestra las clases que lo componen.

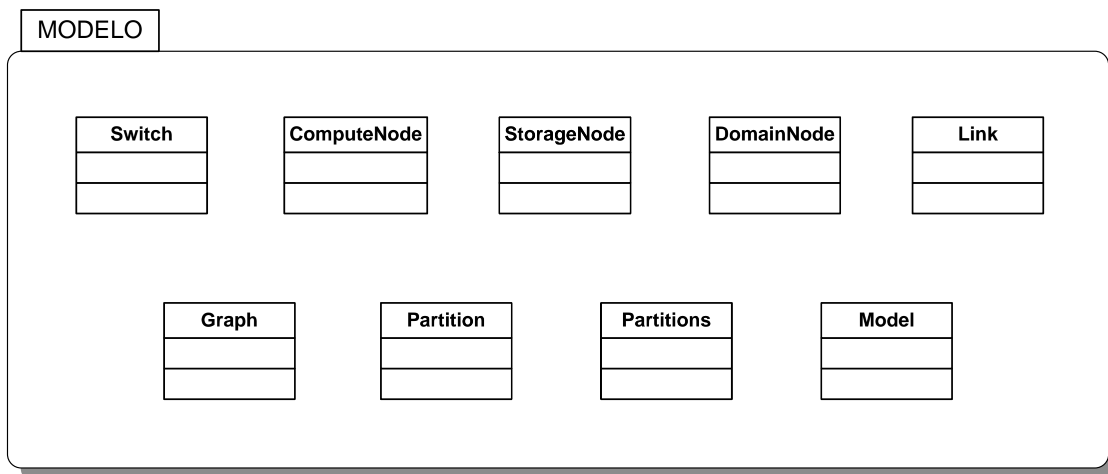


**Figura 8. Componente Modelo**

El componente Entrada contiene una única clase cuya función es la de proporcionar una interfaz de acceso a datos al componente Presentación. A su vez, necesita de las estructuras que le proporciona la interfaz de Modelo. La clase que contiene es *ParserFile*. En la figura 9 aparece el componente.

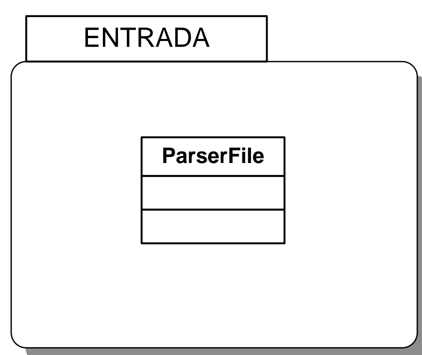


**Figura 9. Componente Entrada**

Para terminar con la descripción de componentes, se habla del componente BBDD. Este componente se basa en un conjunto de fichero dentro del sistema. Estos ficheros disponen de los datos de entrada a la aplicación, es decir, contienen los modelos que los algoritmos de la aplicación van a evaluar. Este componente proporciona una interfaz al componente Entrada con el formato de cada elemento de los modelos dentro de cada fichero. Esto se detallará en el anexo manual de usuario. La figura 10 muestra dicho componente.

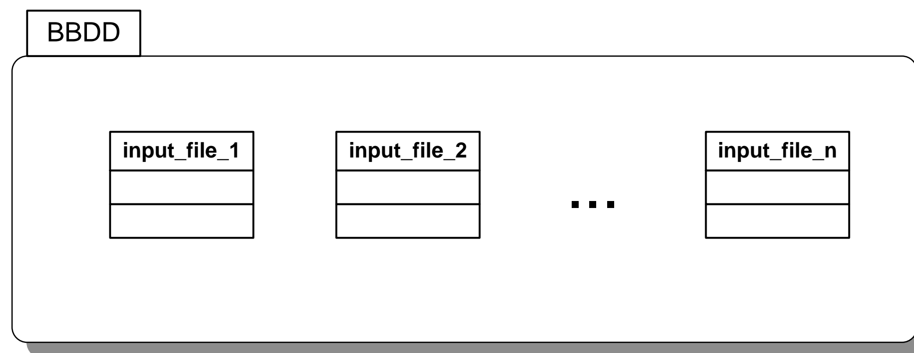


**Figura 10. Componente BBDD**

## 4.2. Diseño de clases

En esta sección se va a describir el diagrama de clases de la aplicación así como todos los métodos de cada una de las clases.

### 4.2.1. Diagrama de clases

En esta sección se va a detallar cada clase de la aplicación gráficamente, incluyendo los atributos y métodos de los que disponen. En primer lugar se representan todas las relaciones de clases dentro del sistema, para luego pasar a describir minuciosamente cada clase y sus métodos.

En la figura 11 se pueden observar todas las relaciones existentes dentro del diagrama de clases.

En primer lugar se tiene la clase *ExecuteTests* que es la encargada de abrir la interfaz gráfica del sistema. Como se puede apreciar tiene dependencias de las clases encargadas de resolver el problema (algoritmos del componente Negocio) y de la clase *ParserFile*. A partir de la interfaz gráfica el usuario puede cargar el fichero de entrada y elegir los diferentes parámetros para encontrar la solución.

Para continuar se tienen las clases de algoritmos que son las pertenecientes al modelo Negocio. Estas clases son: *SimpleBalancedPartitionAlgorithm, ProximityPartitionAlgorithm* y *BalancedPartitionAlgorithm*. En estas clases es donde se llevan a cabo todos los cálculos para llegar a la solución del problema (se explicarán con más detalle en el siguiente apartado). Tienen dependencias directas de diferentes clases del componente Modelo como se puede apreciar, estas son necesarias para dar soporte a dichos cálculos.

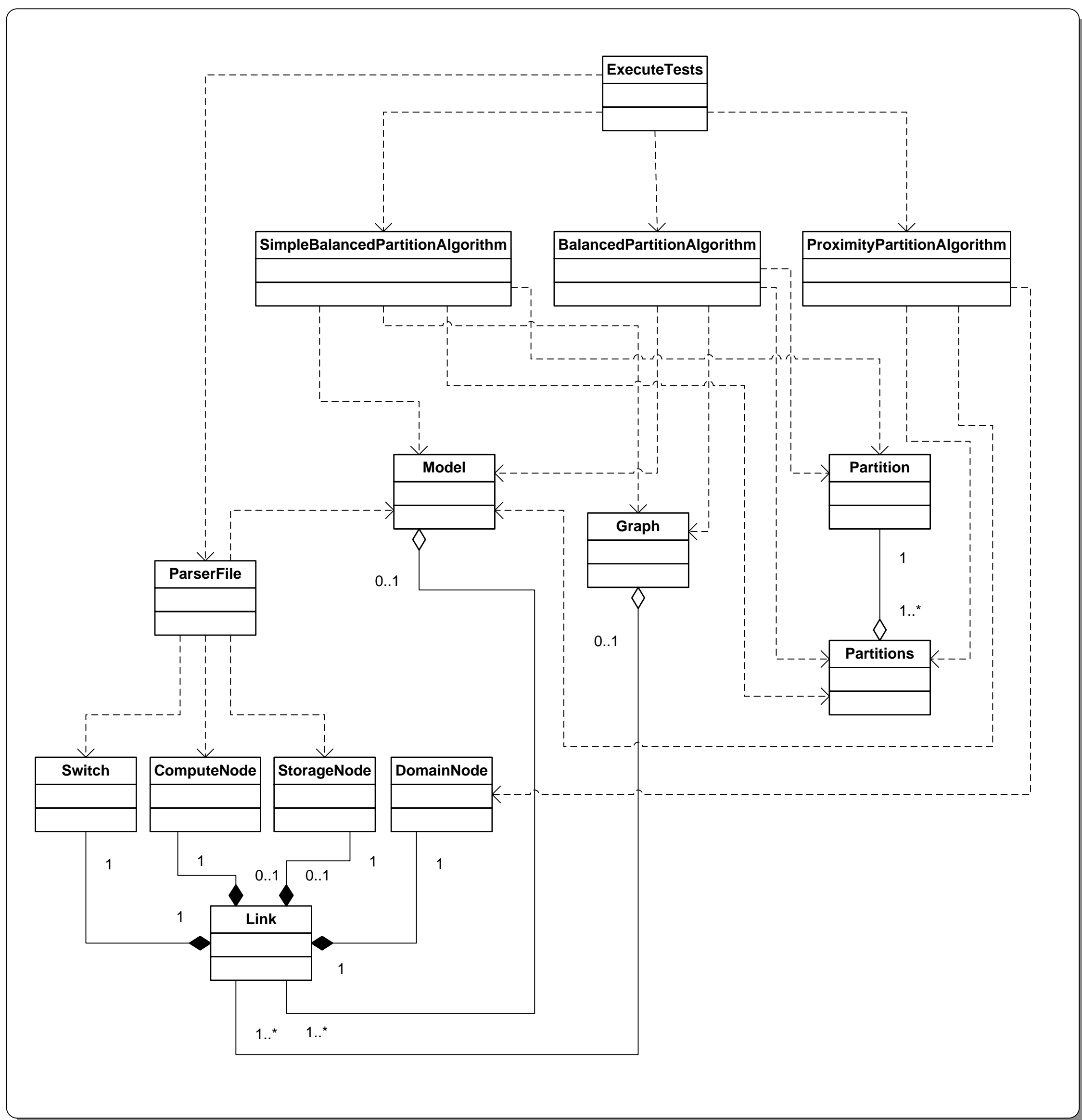


**Figura 11. Diagrama de clases**

La clase *ParserFile* es la encargada de la entrada del sistema. Tiene dependencias directas sobre diferentes clases del componente Modelo, que dan el soporte necesario para almacenar en estructuras los datos de los ficheros de entrada elegidos por el usuario.

Por último hay que hablar de las clases del componente Modelo, que como se ha comentado anteriormente en su totalidad sirven como estructuras básicas para usar en otros componentes e incluso alguna de estas clases sirven como estructuras para otras clases del mismo componente. A continuación se procede a numerar dichas clases: *Switch*, *ComputeNode*, *StorageNode*, *DomainNode*, *Link*, *Model*, *Graph*, *Partition* y *Partitions*.

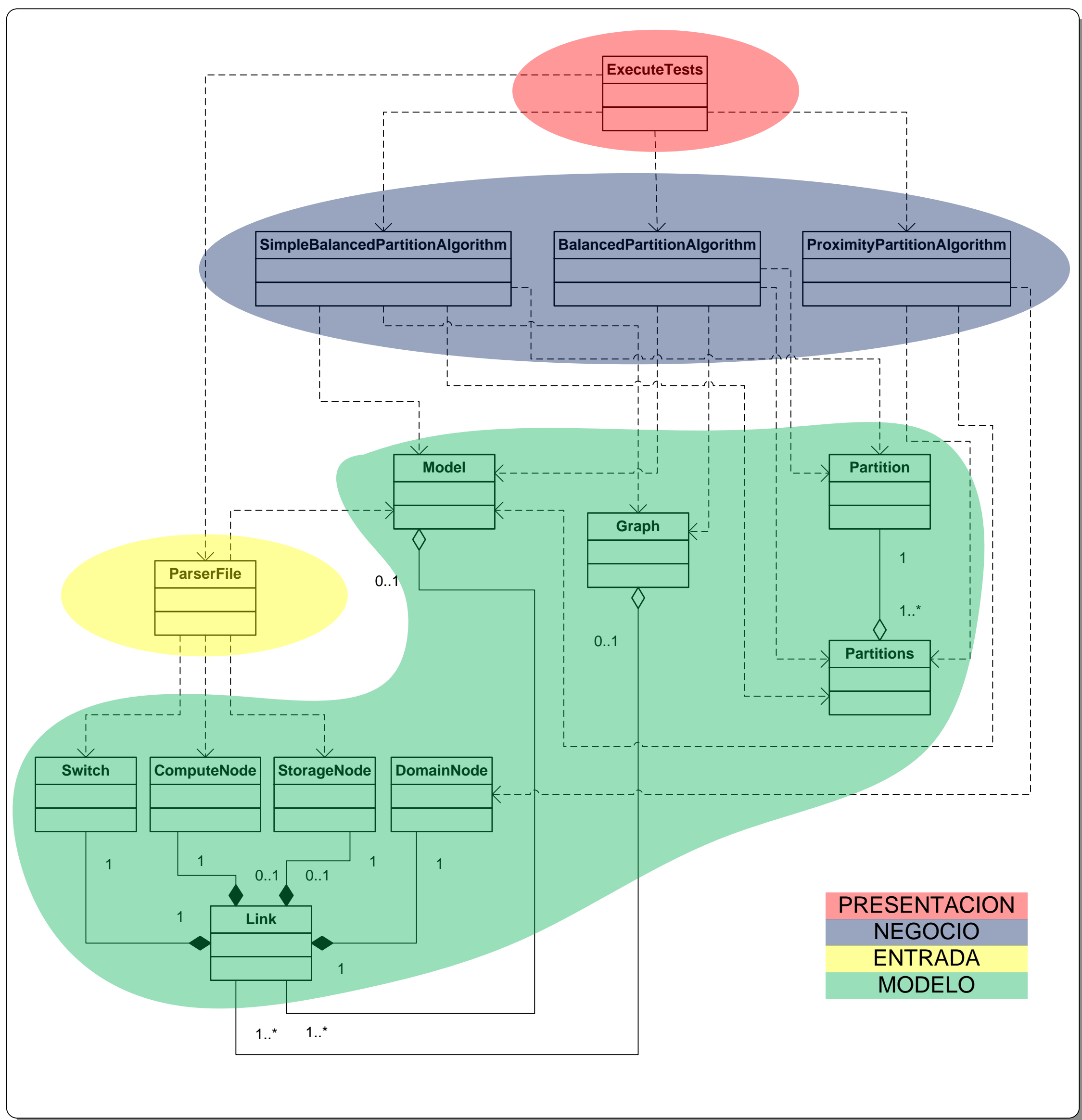


**Figura 12. Diagrama de clases por componente**

La figura 12 ilustra cada componente dentro del diagrama de clases con sus relaciones.

Una vez presentadas las relaciones existentes entre las diferentes clases del sistema, se van a explicar con más detalle una a una cada clase. Para ello, en primer lugar se va a proceder a mostrar cada clase dentro de cada componente, para posteriormente en el siguiente apartado de la memoria explicar con detalle cada método.

Del componente Presentación se muestra en la figura 13 la clase que lo forma junto con sus atributos y métodos.

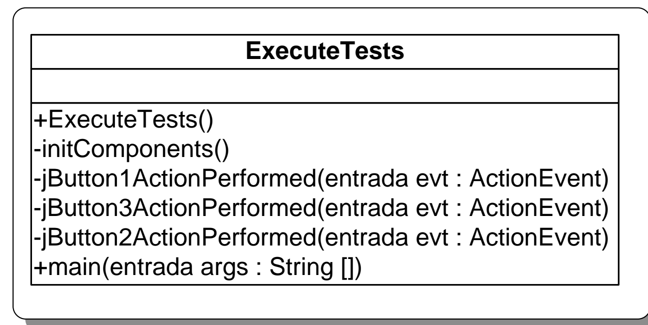


**Figura 13. Clases componente Presentación**

A continuación se muestra el componente Entrada formado por una única clase llamada *ParserFile* en la figura 14.

La figura 15 muestra el componente de Negocio con sus tres clases cada una de las cuales pertenece a un algoritmo diferente.

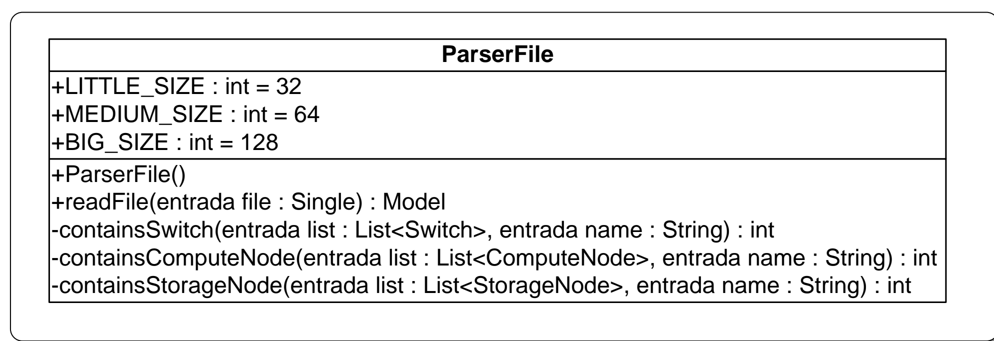


**Figura 14. Clases componente Entrada**

Para acabar con este apartado se procede a mostrar todas las clases del componente Modelo. La figura 16 muestra las clases *Switch*, *ComputeNode*, *StorageNode* y *Model*.

La figura 17 muestra las clases *Graph*, *Partition* y *Partitions*.

La figura 18 muestra las clases *Link* y *DomainNode*.

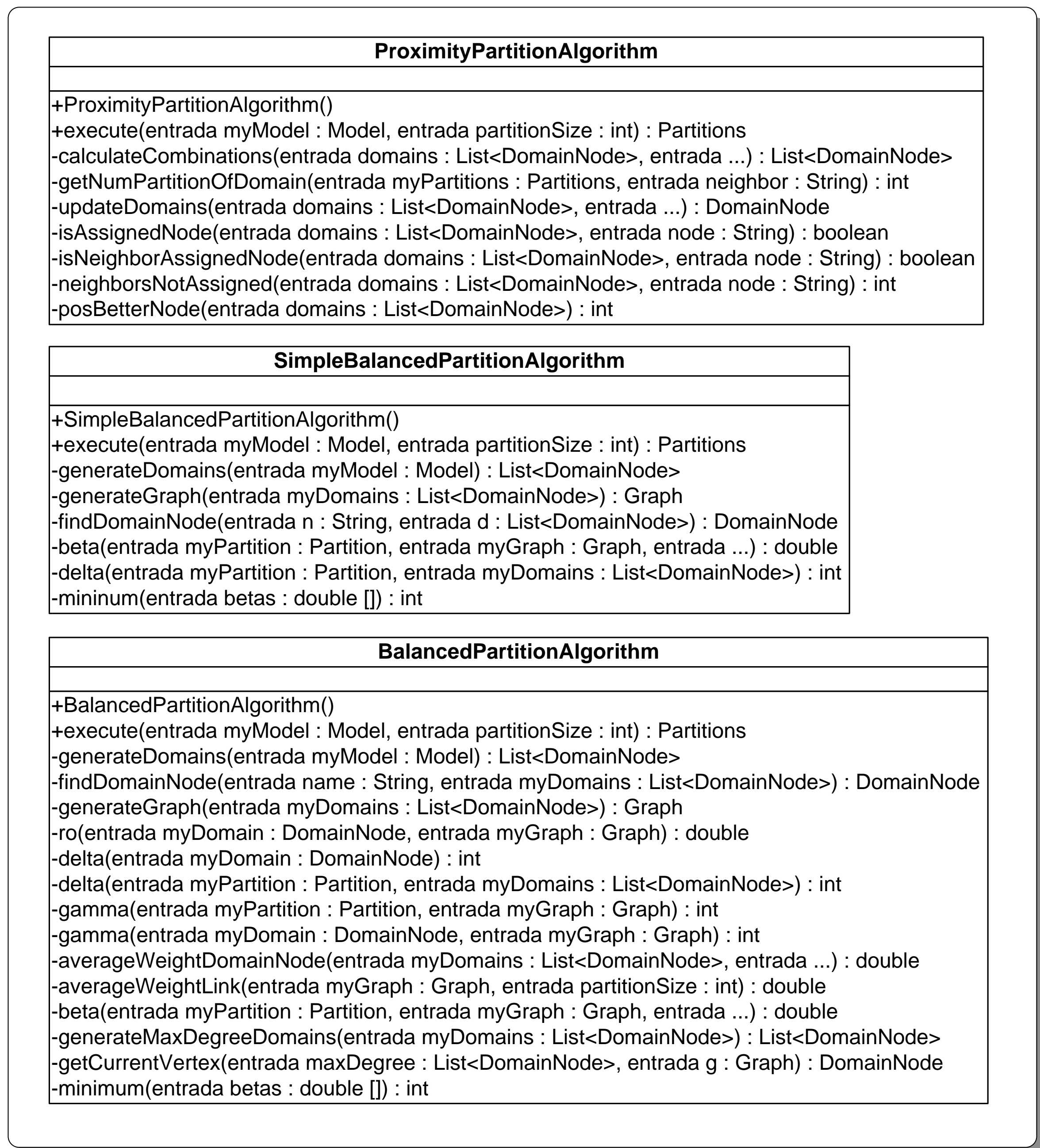


**Figura 15. Clases componente Negocio**

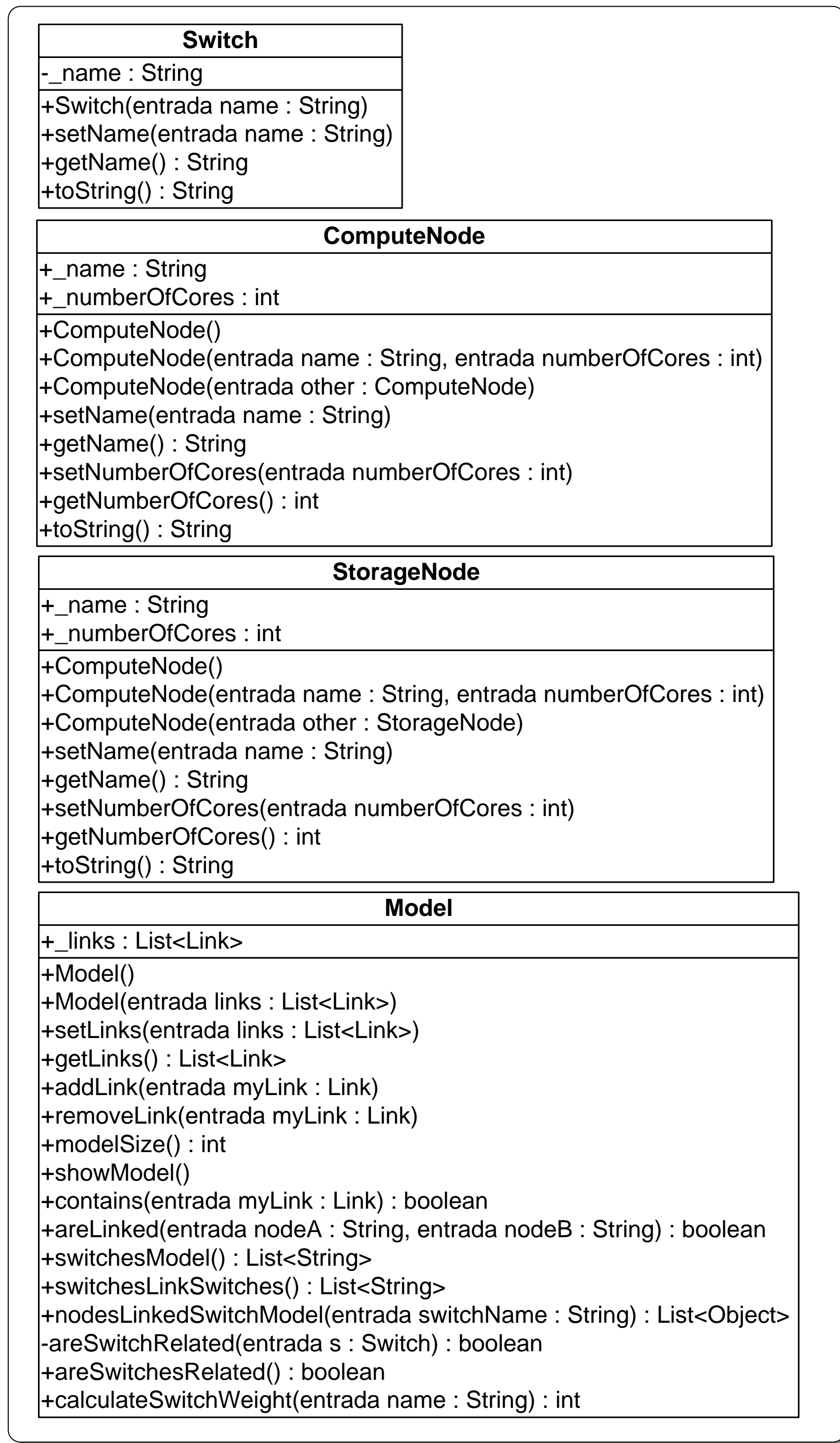


**Figura 16. Clases componente Modelo**

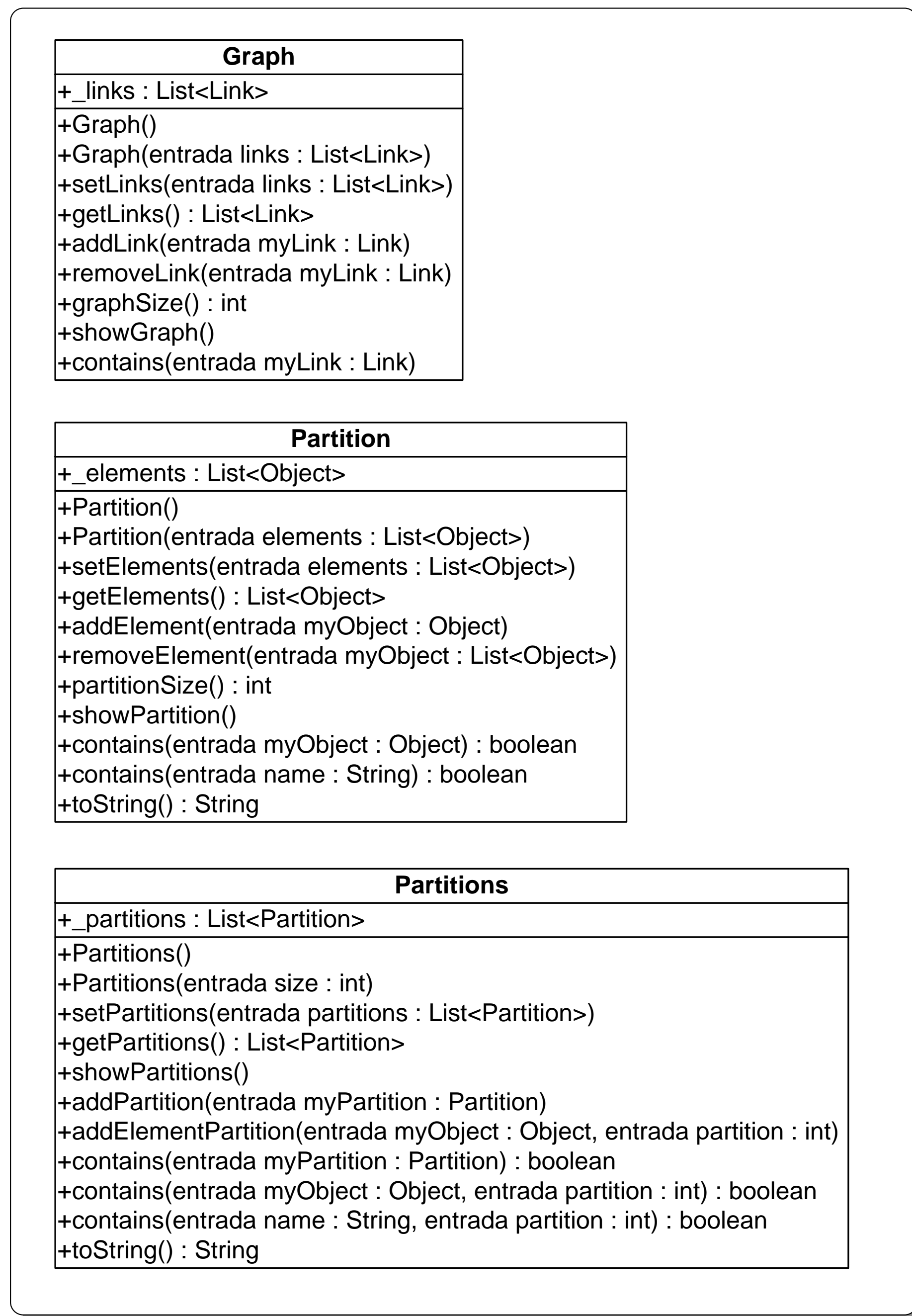


**Figura 17. Clases componente Modelo**

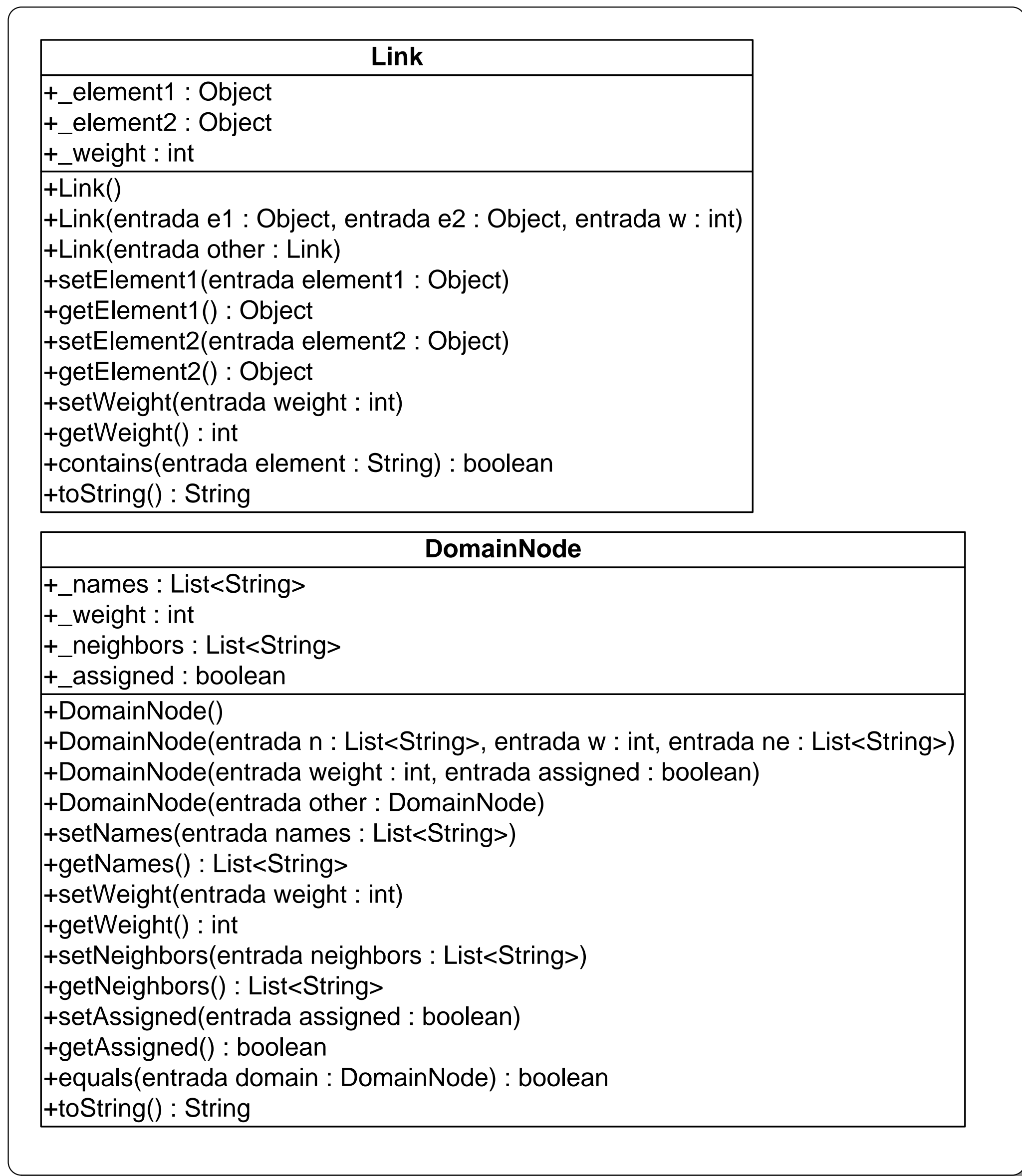


**Figura 18. Clases componente Modelo**

### 4.2.2. Implementación

#### *Switch*

*Switch* es la clase que encapsula un elemento de este tipo. Este componente encapsula la información necesaria para que los algoritmos de particionado puedan usarla y trabajen con ella. Esta información es únicamente su nombre, que para que sea único siempre empieza por la letra S seguido de un número. A continuación se describen los métodos que forman parte de la clase.

- *Switch(String name)*

  Constructor principal de la clase. Un objeto de esta clase está formado únicamente por su nombre.

- *void setName(String name)*

  Asigna un nombre dado por parámetros a un objeto de la clase.

- *String getName()*

  Devuelve el nombre de un objeto de la clase.

- *public String toString()*

  Devuelve el objeto de la clase en forma de objeto String. En este caso, éste será el nombre del elemento.

## *ComputeNode*

*ComputeNode* es la clase que modela un elemento de este tipo. Este componente encapsula la información necesaria para que los algoritmos de particionado trabajen con él. Esta información está compuesta por el nombre y el número de núcleos que tiene el nodo. El atributo nombre es el que hace al elemento único, siempre empieza por las letras "CN" seguido por un número. A continuación se describen los métodos que forman parte de la clase.

- *ComputeNode()*

  Constructor vacío.

- *ComputeNode(String name, int numberOfCores)*

  Constructor principal de la clase. Forma un objeto de la clase a partir de los parámetros nombre y número de núcleos.

- *ComputeNode(ComputeNode other)*

  Constructor secundario de la clase. Forma un objeto de la clase a partir de otro ya existente del mismo tipo. El nuevo objeto toma los atributos del objeto pasado por parámetros.

- *void setName(String name)*

  Asigna un nombre dado por parámetros a un objeto de la clase.

- *String getName()*

  Devuelve el nombre de un objeto de la clase.

- *void setNumberOfCores(int numberOfCores)*

  Asigna un número de núcleos pasado por parámetros a un objeto de la clase.

- *int getNumberOfCores()*

  Devuelve el número de núcleos de un objeto de la clase.

- *public String toString()*

  Devuelve el objeto de la clase en forma de objeto String. En este caso, éste será el nombre del elemento seguido por un guión bajo ("_") y terminado con el número de núcleos. Ej. CN1_32; CN2_128;...

## *StorageNode*

*StorageNode* es la clase que modela un elemento de este tipo. Este componente encapsula la información necesaria para que los algoritmos de particionado trabajen con él. Esta información está compuesta por el nombre y el número de núcleos que tiene el nodo. El atributo nombre es el que hace al elemento único, siempre empieza por las letras "SN" seguido por un número. A continuación se describen los métodos que forman parte de la clase.

- *StorageNode()*

  Constructor vacío.

- *StorageNode(String name, int numberOfCores)*

  Constructor principal de la clase. Forma un objeto de la clase a partir de los parámetros nombre y número de núcleos.

- *StorageNode(StorageNode other)*

  Constructor secundario de la clase. Forma un objeto de la clase a partir de otro ya existente del mismo tipo. El nuevo objeto toma los atributos del objeto pasado por parámetros.

- *void setName(String name)*

  Asigna un nombre dado por parámetros a un objeto de la clase.

- *String getName()*

  Devuelve el nombre de un objeto de la clase.

- *void setNumberOfCores(int numberOfCores)*

  Asigna un número de núcleos pasado por parámetros a un objeto de la clase.

- *int getNumberOfCores()*

  Devuelve el número de núcleos de un objeto de la clase.

- *public String toString()*

  Devuelve el objeto de la clase en forma de objeto String. En este caso, éste será el nombre del elemento seguido por un guión bajo ("_") y terminado con el número de núcleos. Ej. SN1_32; SN2_128;...

## *Model*

*Model* es la clase que modela un elemento de ese tipo. Este componente encapsula la información necesaria para que cualquier algoritmo de particionado trabaje con él. Un objeto de tipo *Model* está formado por una lista de objetos *Link*. A continuación se describen los métodos que forman parte de la clase.

- *Model()*

  Constructor principal que crea un objeto de la clase que contiene una lista vacía de enlaces.

- *Model(List<Link> links)*

  Constructor secundario que crea un objeto de la clase a partir de una lista de enlaces previa.

- *void setLinks(List<Link> links)*

  Asigna una lista de enlaces al objeto de la clase.

- *List<Link> getLinks()*

  Devuelve una lista de enlaces.

- *void addLink(Link myLink)*

  Añade un enlace a un objeto de la clase.

- *void removeLink(Link myLink)*

  Borra un enlace de un objeto de la clase.

- *int modelSize()*

  Devuelve el tamaño de un objeto de la clase. El tamaño viene dado por la longitud de la lista de enlaces.

- *void showModel()*

  Muestra un objeto de la clase por pantalla.

- *boolean contains(Link myLink)*

  Devuelve *true* si un objeto de la clase contiene el enlace pasado por parámetros en su lista de enlaces. Devuelve *false* en caso contrario.

- *boolean areLinked(String nodeA, String nodeB)*

  Devuelve *true* si dos elementos con nombres *nodeA* y *nodeB* pertenecen al modelo. Es decir, si existe un enlace con cuyos extremos tienen por nombres *nodeA* y *nodeB* devuelve true. En caso contrario devuelve *false*.

- *List<String> switchesModel()*

  Devuelve una lista de elementos de tipo *String* con los nombres de la totalidad de los elementos *Switch* que existen en el modelo.

- *List<String> switchesLinkSwitches()*

  Devuelve una lista de elementos de tipo *String* con los nombres de los elementos *Switch* que participan en al menos un enlace con otro elemento de tipo *Switch*. Esto se hace para comprobar que todo *Switch* siempre ha de tener un enlace hacia otro *Switch* por restricción del problema.

- *List<Object> nodesLinkedSwitchModel(String switchName)*

  Devuelve una lista de elementos de tipo *Object* con elementos de tipo *ComputeNode* y *StorageNode* conectados dentro del modelo a un *Switch* dado por parámetros mediante su nombre.

- *private boolean areSwitchRelated(Switch s)*

  Devuelve true si un *Switch*, dado por parámetros a partir de su nombre, tiene al menos un enlace, tanto con un *ComputeNode* como con un *StorageNode*, dentro del modelo. Esto se comprueba como restricción del problema. Devuelve *false* en caso contrario.

- *boolean areSwitchesRelated()*

  Devuelve true si todos los *Switches* del modelo están conectados con al menos un *ComputeNode* y un *StorageNode*. Devuelve *false* en caso contrario.

- *int calculateSwitchWeight(String name)*

  Devuelve el peso que tiene un elemento *Switch* dentro del modelo. El peso siempre será un valor positivo. El peso se calcula sumando los pesos de los enlaces que tiene el susodicho Switch con los nodos *ComputeNode* y *StorageNode*. Se contempla que los *StorageNode* tengan un peso 5 veces mayor que los *ComputeNode*.

### *Graph*

*Graph* es la clase que modela un elemento de ese tipo. Este componente encapsula la información necesaria para que cualquier algoritmo de particionado trabaje con él. Un objeto de tipo *Graph* está formado por una lista de enlaces de tipo de *Link*. A continuación se describen los métodos que forman parte de la clase.

- *Graph()*

  Constructor principal que crea un objeto de la clase que contiene una lista vacía de enlaces.

- *Graph(List<Link> links)*

  Constructor secundario que crea un objeto de la clase a partir de una lista de enlaces previa.

- *void setLinks(List<Link> links)*

  Asigna una lista de enlaces al objeto de la clase.

- *List<Link> getLinks()*

  Devuelve una lista de enlaces.

- *void addLink(Link myLink)*

  Añade un enlace a un objeto de la clase.

- *void removeLink(Link myLink)*

  Borra un enlace de un objeto de la clase.

- *int graphSize()*

  Devuelve el tamaño de un objeto de la clase. El tamaño viene dado por la longitud de la lista de enlaces.

- *void showGraph()*

  Muestra un objeto de la clase por pantalla.

- *boolean contains(Link myLink)*

  Devuelve *true* si un objeto de la clase contiene el enlace pasado por parámetros en su lista de enlaces. Devuelve *false* en caso contrario.

## *Partition*

*Partition* es la clase que modela un elemento de ese tipo. Este componente encapsula la información necesaria para que cualquier algoritmo de particionado trabaje con él. A continuación se describen los métodos que forman parte de la clase.

- *Partition()*

  Constructor principal que crea un objeto de la clase que contiene una lista vacía de objetos.

- *Partition(List<Object> elements)*

  Constructor secundario que crea un objeto de la clase a partir de una lista de enlaces previa.

- *void setElements(List<Object> elements)*

  Asigna una lista de objetos al objeto de la clase.

- *List<Object> getElements()*

  Devuelve una lista de objetos.

- *void addElement(Object myObject)*

  Añade un objeto a un objeto de la clase.

- *void removeElement(Object myObject)*

  Borra un objeto de un objeto de la clase.

- *int partitionSize()*

  Devuelve el tamaño de un objeto de la clase. El tamaño viene dado por la longitud de la lista de objetos.

- *void showPartition()*

  Muestra un objeto de la clase por pantalla.

- *boolean contains(Object myObject)*

  Devuelve *true* si un objeto de la clase contiene el objeto pasado por parámetros en su lista de objetos. Devuelve *false* en caso contrario.

- *boolean contains(String name)*

  Devuelve *true* si un objeto de la clase contiene el objeto cuyo nombre es pasado por parámetros en su lista de objetos. Devuelve *false* en caso contrario.

- *public String toString()*

  Devuelve el objeto de la clase en forma de objeto String. En este caso, éste será el nombre de cada elemento de la lista de objetos separados por línea. Ej.

  SN1

  CN1

  S1

  SN2

## *Partitions*

*Partitions* es la clase que modela un elemento de ese tipo. Este componente encapsula la información necesaria para que cualquier algoritmo de particionado trabaje con él. Un objeto *Partitions* dispone de una lista de objetos de tipo *Partition*, que a su vez contiene una lista de los objetos que pertenecen a esa partición. A continuación se describen los métodos que forman parte de la clase.

- *Partitions()*

  Constructor vacío.

- *Partitions(int size)*

  Constructor principal de la clase que crea una lista de particiones con tamaño *size* pasado por parámetros.

- *void setPartitions(List<Partition> partitions)*

  Asigna una lista de particiones a un objeto de la clase.

- *List<Partition> getPartitions()*

  Devuelve una lista de particiones.

- *void showPartitions()*

  Muestra todas las particiones.

- *void addPartition(Partition myPartition)*

  Añade una partición a la lista de particiones.

- *void addElementPartition(Object myObject, int partition)*

  Añade un elemento pasado por parámetros en la partición *partition* también pasada por parámetros.

- *boolean contains(Partition myPartition)*

  Devuelve *true* si la lista de particiones contiene la partición pasada por parámetros. Devuelve *false* en caso contrario.

- *boolean contains(Object myObject, int partition)*

  Devuelve *true* si la lista de particiones contiene el objeto pasado por parámetros dentro de la partición número *partition*. Devuelve *false* en caso contrario.

- *boolean contains(String name, int partition)*

  Devuelve *true* si la lista de particiones contiene el objeto cuyo nombre es pasado por parámetros dentro de la partición número *partition*. Devuelve *false* en caso contrario.

- *public String toString()*

  Devuelve el objeto de la clase en forma de objeto *String*. En este caso, éste será el objeto *String* de cada partición de la lista de particiones. Es decir, la representación del objeto será el listado de objetos de cada una de las particiones.

## *Link*

*Link* es la clase que modela un elemento de ese tipo. Este componente encapsula la información necesaria para que cualquier algoritmo de particionado trabaje con él. Dispone de dos extremos representados por 2 objetos, además del peso del propio enlace. Se puede utilizar como un enlace tanto en *modelos* como en g*rafos*. A continuación se describen los métodos que forman parte de la clase.

- *Link()*

    Constructor que crea un objeto vacío de la clase.

- *Link(Object element1, Object element2, int weight)*

    Constructor principal que crea un objeto de la clase.

- *Link (Link other)*

    Constructor secundario que crea un objeto de la clase a partir de otro dado.

- *void setElement1(Object element1)*

    Asigna el element1 a un objeto de la clase.

- *Object getElement1()*

    Devuelve el element1 del objeto de la clase.

- *void setElement2(Object element2)*

    Asigna el element2 a un objeto de la clase.

- *Object getElement2()*

    Devuelve el element2 del objeto de la clase.

- *void setWeight(int weight)*

    Asigna el peso del objeto de la clase.

- *int getWeight()*

    Devuelve el peso del objeto de la clase.

- *boolean contains(String element)*

    Devuelve *true* si el element pasado por parámetros pertenece al enlace. Devuelve *false* en caso contrario.

- *public String toString()*

    Devuelve el objeto de la clase en forma de objeto String. En este caso, éste será el nombre del primer elemento seguido por un guión medio ("-"). A su vez continua con el

nombre del segundo elemento y un nuevo guión medio y terminado con el peso del enlace. Ej. CN1-S1-80, SN1-S2-100, S1-S2-0,…

## *DomainNode*

*DomainNode* es la clase que modela un elemento de ese tipo. Este componente encapsula la información necesaria para que cualquier algoritmo de particionado trabaje con él. A continuación se describen los métodos que forman parte de la clase.

- *DomainNode()*

  Constructor que crea un objeto vacío de la clase.

- *DomainNode(List<String> names, int weight, List<String> neighbors, boolean assigned)*

  Constructor principal que crea un objeto de la clase.

- *DomainNode(int weight, boolean assigned)*

  Constructor secundario que crea un objeto vacío de la clase con peso y asignado.

- *DomainNode(DomainNode other)*

  Constructor secundario que crea un objeto a partir de otro dado.

- *void setNames(List<String> names)*

  Asigna la lista de nodos a un objeto de la clase.

- *List<String> getNames()*

  Devuelve la lista de nodos de un objeto de la clase.

- *void setWeight(int weight)*

  Asigna el peso a un objeto de la clase.

- *Int getWeight()*

  Devuelve el peso de un objeto de la clase.

- *void setNeighbors(List<String> neighbors)*

  Asigna la lista de vecinos a un objeto de la clase.

- *List<String> getNeighbors()*

  Devuelve la lista de vecinos de un objeto de la clase.

- *void setAssigned(boolean assigned)*

  Asigna el valor asignado a un objeto de la clase.

- *boolean getAssigned()*

  Devuelve el valor asignado de un objeto de la clase.

- *boolean equals(DomainNode domain)*

  Devuelve *true* si el objeto *domain* pasado por parámetros es igual, en todos sus elementos, que el objeto de la clase. Devuelve *false* en caso contrario.

- *public String toString()*

  Devuelve el objeto de la clase en forma de objeto String. En este caso, éste será un listado de los elementos del dominio, seguido del peso del nodo y terminado por un nuevo listado pero esta vez de vecinos de los elementos del dominio. Ej.

  NOMBRE DOMINIO

  S1_S2_90

  VECINOS DOMINIO

  S3

## *ParserFile*

*ParserFile* es la clase que modela un objeto de este tipo. Este componente encapsula la información necesaria para extraer un modelo a partir de un fichero de entrada. Contiene unas variables final que identifican una escala del número de núcleos que contiene cada nodo. En el fichero de entrada se ha de especificar dichas cantidades mediante dichos pseudónimos. Dicha escala es:

```
final int LITTLE_SIZE = 32;
final int MEDIUM_SIZE = 64;
final int BIG_SIZE = 128;
```

Una vez comentado lo anterior se procede a describir los métodos que forman parte de la clase.

- *ParserFile()*

  Constructor vacío de la clase.

- *Model readFile(String file) throws FileNotFoundException, IOException*

  Lee el fichero de nombre *file* y lo almacena en una estructura *Model* que es devuelta por el método.

- *private int containsSwitch(List<Switch> list, String name)*

  Devuelve la posición de un *Switch* buscado dentro de la lista de *switches* del modelo. Si el *Switch* no está en la lista devuelve -1.

- *private int containsComputeNode(List<ComputeNode> list, String name)*

  Devuelve la posición de un *ComputeNode* buscado dentro de la lista de *computeNodes* del modelo. Si el *ComputeNode* no está en la lista devuelve -1.

- *private int containsStorageNode(List<StorageNode> list, String name)*

  Devuelve la posición de un *StorageNode* buscado dentro de la lista de *storageNodes* del modelo. Si el *StorageNode* no está en la lista devuelve -1.

### *ProximityPartitionAlgorithm*

*ProximityPartitionAlgorithm* es la clase que implementa la ejecución del algoritmo de partición de proximidad.

A grandes rasgos, el algoritmo consiste en primer lugar en agrupar los nodos del modelo en nodos de dominio *DomainNode*. Después, en una primera iteración se asigna cada uno de los nodos *DomainNode* a cada una de las particiones existentes. Se asignan los nodos con mayor número de vecinos.

En las siguientes iteraciones se escogen los nodos con menos vecinos sin asignar y siempre nodos que tengan conexiones directas con otros ya asignados. El algoritmo termina cuando todos los nodos de dominio han sido asignados a una partición. Una vez comentado lo anterior se procede a describir los métodos que forman parte de la clase.

- *ProximityPartitionAlgorithm()*

  Constructor principal de la clase.

- *Partitions execute(Model myModel, int partitionSize)*

  Ejecuta el algoritmo sobre el modelo dado dividiéndolo en el número de particiones *partitionSize*. Consigue la distribución óptima para los elementos del modelo en las particiones requeridas. Devuelve una lista de particiones con el resultado del algoritmo.

- *private List<DomainNode> calculateCombinations(List<DomainNode> domains, Partitions myPartitions, int partitionSize)*

  Devuelve una nueva distribución de los componentes *DomainNode* del modelo. Además asigna un componente *DomainNode* a cada partición dando prioridad a aquellos que tienen menos vecinos por asignar.

- *private int getNumPartitionOfDomain(Partitions myPartitions, String neighbor)*

  Devuelve el número de partición donde se han de insertar elementos. Devuelve siempre un número positivo.

- *private List<DomainNode> updateDomains(List<DomainNode> domains, List<DomainNode> combinations, Partitions myPartitions)*

  Devuelve una nueva lista de componentes *DomainNode* por cada iteración del algoritmo. La actualización consiste en agregar los elementos de los *DomainNode* insertados en la actual iteración, dentro de los *DomainNode* de la iteración anterior. Actualizando tanto los nombres como los pesos y los nodos vecinos.

- *private boolean isAssigned(List<DomainNode> domains, String node)*

  Comprueba si un *DomainNode* pertenece ya a una partición cualquiera o no. Devuelve true si esto es cierto y devuelve false en caso contrario.

- *private boolean isNeighborAssignedNode(List<DomainNode> domains, String node)*

  Comprueba si un DomainNode vecino pertenece ya a una partición cualquiera o no. Devuelve true si esto es cierto y devuelve false en caso contrario.

- *private int neighborsNotAssigned(List<DomainNode> domains, String node)*

  Devuelve el número de vecinos no asignados ya a una partición cualquiera. Devuelve -1 si todos los vecinos han sido ya asignados a una partición.

- *private int posBetterNode(List<DomainNode> domains)*

  Devuelve la posición del mejor *DomainNode* sobre la cual el algoritmo ha de insertarlo en la partición correspondiente. Los mejores nodos serán aquellos que tengan más nodos vecinos libres sin asignar a ninguna partición.

### *SimpleBalancedPartitionAlgorithm*

*SimpleBalancedPartitionAlgorithm* es la clase que implementa la ejecución del algoritmo de partición de balanceo simple.

El algoritmo consiste en agrupar nodos en dominios, para luego asignar cada dominio a una partición acorde únicamente solo a los pesos de los dominios. Un punto importante en el algoritmo es que cada dominio está basado en la red que maneja un switch en el modelo. El dominio a insertar en la partición siempre es el primero de la lista de dominios, una vez insertado en la partición correspondiente éste es borrado de la lista de dominios. Una vez comentado lo anterior se procede a describir los métodos que forman parte de la clase.

- *SimpleBalancedPartitionAlgorithm()*

  Constructor principal de la clase.

- *Partitions execute(Model myModel, int partitionSize)*

  Ejecuta el algoritmo sobre el modelo dado dividiéndolo en el número de particiones *partitionSize*. Consigue la distribución óptima para los elementos del modelo en las particiones requeridas. Devuelve una lista de particiones con el resultado del algoritmo.

- *private List<DomainNode> generateDomains(Model myModel)*

  Devuelve una lista de componentes *DomainNode* a partir de un modelo dado. Añade semántica nueva que el modelo no tiene, como los pesos o los vecinos de un componente *DomainNode*.

- *private Graph generateGraph(List<DomainNode> myDomains)*

  Devuelve un objeto *Graph* a partir de un modelo dado. Añade nueva semántica como los pesos y los enlaces. Un grafo está formado por una lista de enlaces, y a su vez estos están formados por dos componentes *DomainNode*.

- *private DomainNode findDomainNode(String name, List<DomainNode> myDomains)*

  Devuelve un componente *DomainNode* de la lista de dominios. Si el componente no es encontrado devuelve *null*. El componente *DomainNode* puede contener varios nombres de nodos, con que uno de esos nombres sea el nombre dado por parámetros es suficiente para considerarlo como encontrado.

- *private double beta(Partition myPartition, Graph myGraph, int partitionSize, List<DomainNode> myDomains)*

  Devuelve un valor *double* que representa lo bien que balancea el algoritmo tras la inserción de los elementos de un componente *DomainNode* en cada partición. El mejor valor de beta es aquel más cercano a 0, que indica que la inserción en esa partición ha tenido el menor impacto de todas las inserciones en cuanto a balanceo de carga. El balanceo para este algoritmo solo se basa en el peso de los componentes *DomainNode*.

- *private int delta(Partition myPartition, List<DomainNode> myDomains)*

  Devuelve el peso de una partición dada. Se calcula como la suma de los pesos de los componentes *DomainNode* que pertenecen a dicha partición.

- *private int minimum(double [] betas)*

  Devuelve la posición del elemento con menor valor beta de la colección pasada por parámetros. Primero encuentra el valor más pequeño para luego devolver su posición.

### *BalancedPartitionAlgorithm*

BalancedPartitionAlgorithm es la clase que implementa la ejecución del algoritmo de partición de balanceo.

El algoritmo consiste en agrupar nodos en dominios, para luego asignar cada dominio a una partición acorde a los pesos de los dominios y sus enlaces. Un punto importante en el

algoritmo es que cada dominio está basado en la red que maneja un switch en el modelo. Otro aspecto esencial es la elección del dominio a insertar en una partición. Esto se hace acorde a un serie de fórmulas cuyo fin es intentar encontrar el particionado ideal para que el impacto (en términos de peso) de insertar un dominio en una partición sea el mínimo posible. Una vez comentado lo anterior se procede a describir los métodos que forman parte de la clase.

- *BalancedPartitionAlgorithm()*

  Constructor principal de la clase.

- *Partitions execute(Model myModel, int partitionSize)*

  Ejecuta el algoritmo sobre el modelo dado dividiéndolo en el número de particiones *partitionSize*. Consigue la distribución óptima para los elementos del modelo en las particiones requeridas. Devuelve una lista de particiones con el resultado del algoritmo.

- *private List<DomainNode> generateDomains(Model myModel)*

  Devuelve una lista de componentes *DomainNode* a partir de un modelo dado. Añade semántica nueva que el modelo no tiene, como los pesos o los vecinos de un componente *DomainNode*.

- *private Graph generateGraph(List<DomainNode> myDomains)*

  Devuelve un objeto *Graph* a partir de un modelo dado. Añade nueva semántica como los pesos y los enlaces. Un grafo está formado por una lista de enlaces, y a su vez estos están formados por dos componentes *DomainNode*.

- *private DomainNode findDomainNode(String name, List<DomainNode> myDomains)*

  Devuelve un componente *DomainNode* de la lista de dominios. Si el componente no es encontrado devuelve *null*. El componente *DomainNode* puede contener varios nombres de nodos, con que uno de esos nombres sea el nombre dado por parámetros es suficiente para considerarlo como encontrado.

- *private double ro(DomainNode myDomain, Graph myGraph)*

  Devuelve el ratio entre los pesos de los componentes *DomainNode* y los pesos de los enlaces del grafo. Un valor pequeño de ro indica que ese componente *DomainNode* tiene un gran impacto en las comunicaciones del modelo, de otra manera indica lo contrario.

- *private int delta(DomainNode myDomain)*

Devuelve el peso de un componente DomainNode.

- *private int delta(Partition myPartition, List<DomainNode> myDomains)*

Devuelve el peso de los nodos de una partición dada. Se calcula como la suma de los pesos de los componentes *DomainNode* que pertenecen a dicha partición.

- *private int gamma(Partition myPartition, Graph myGraph)*

Devuelve el peso de las comunicaciones de una partición dada. Se calcula como la suma de los pesos de los enlaces en los que un extremo pertenece a la partición y el otro no.

- *private int gamma(DomainNode myDomain, Graph myGraph)*

Devuelve el peso de las comunicaciones de un componente *DomainNode*. Se calcula como la suma de los pesos de los enlaces que contienen en uno de sus extremos el componente *DomainNode* pasado por parámetros.

- *private double averageWeightDomainNode(List<DomainNode> myDomains, int partitionSize)*

Devuelve el peso de nodos medio por partición de todos los componentes *DomainNode* del modelo.

- *private double averageWeightLink(Graph myGraph, int partitionSize)*

Devuelve el peso de enlaces medio por partición de todos los componentes *Link* del grafo formado a partir del modelo.

- *private double beta(Partition myPartition, Graph myGraph, int partitionSize, List<DomainNode> myDomains)*

Devuelve un valor double que representa lo bien que balancea el algoritmo tras la inserción de los elementos de un componente *DomainNode* en cada partición. El mejor valor de beta es aquel más cercano a 0, que indica que la inserción en esa partición ha tenido el menor impacto de todas las inserciones en cuanto a balanceo de carga. El balanceo para este algoritmo se basa en el peso de los componentes *DomainNode*, así como del peso de los enlaces del componente *Graph*.

- *private List<DomainNode> generateMaxDegreeDomains(List<DomainNode> myDomains)*

  Devuelve una colección de componentes *DomainNode* que comparten la característica de tener el mayor grado en sus vértices. El grado viene dado por el número de enlaces que cada vértice tiene.

- *private DomainNode getCurrentVertex(List<DomainNode> maxDegree, Graph myGraph)*

  Devuelve el componente *DomainNode* de la lista de *maxDegree* con menor valor de *ro*. *Ro* consiste en el ratio entre el número de nodos y el número de comunicaciones para un componente *DomainNode* dado, en este caso para cada uno de los componentes de la lista *maxDegree*.

- *private int minimum(double [] betas)*

  Devuelve la posición del elemento con menor valor beta de la colección pasada por parámetros. Primero encuentra el valor más pequeño para luego devolver su posición.

### *ExecuteTests*

*ExecuteTests* es la clase que realiza las ejecuciones de cada algoritmo para resolver el problema dado. Provee de una interfaz gráfica que permite muchas opcionalidades a la hora de ejecutar, tales como elegir modelos de entrada, tipos de algoritmo a ejecutar y número de particiones a crear. A continuación se describen los métodos que forman parte de la clase.

- *ExecuteTests()*

  Constructor de la clase que llama al procedimiento de iniciar los componentes de la interfaz gráfica.

- *private void initComponents()*

  Inicia los componentes de la interfaz gráfica.

- *private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)*

  Ejecuta el evento de presionar el botón de búsqueda de fichero de entrada.

- *private void jButton2ActionPerformed(java.awt.event.ActionEvent evt)*

  Ejecuta el evento de presionar el botón de limpiar los cuadros de texto y de opciones de la interfaz gráfica.

- *private void jButton3ActionPerformed(java.awt.event.ActionEvent evt)*

  Ejecuta el evento de presionar el botón de ejecución del algoritmo sobre el problema dado.

- *public static void main(String args[])*

  Programa principal del sistema que crea un nuevo objeto ExecuteTests.

## 4.3. Interfaces

En la capa de presentación se dispone de una serie de acciones que el usuario puede realizar como parte de la interfaz de usuario. Esta capa no proporciona ningún tipo de interfaz a la capa de negocio. En la figura 19 se muestra la signatura de los métodos.

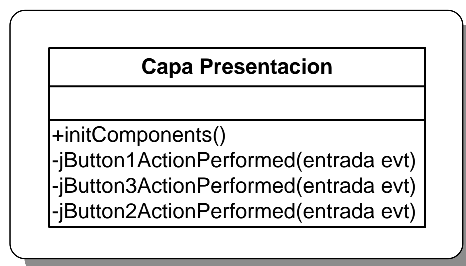


**Figura 19. Interfaz capa Presentación**

La capa de negocio proporciona a la capa de presentación la interfaz de ejecución de los diferentes algoritmos de la plataforma. A su vez contiene diversos métodos que ayudan al funcionamiento de los cálculos para conseguir la salida de la aplicación. Inmediatamente se expone la signatura de los métodos en la figura 20.

Por último la capa de datos ofrece a la capa superior la interfaz para leer los ficheros de entrada de los modelos a particionar, así como un gran elenco de acciones para manejar las diferentes estructuras que se han de crear como dominios, grafos, modelos, enlaces, etc. Posteriormente en la figura 21 se enseña la signatura de los métodos.

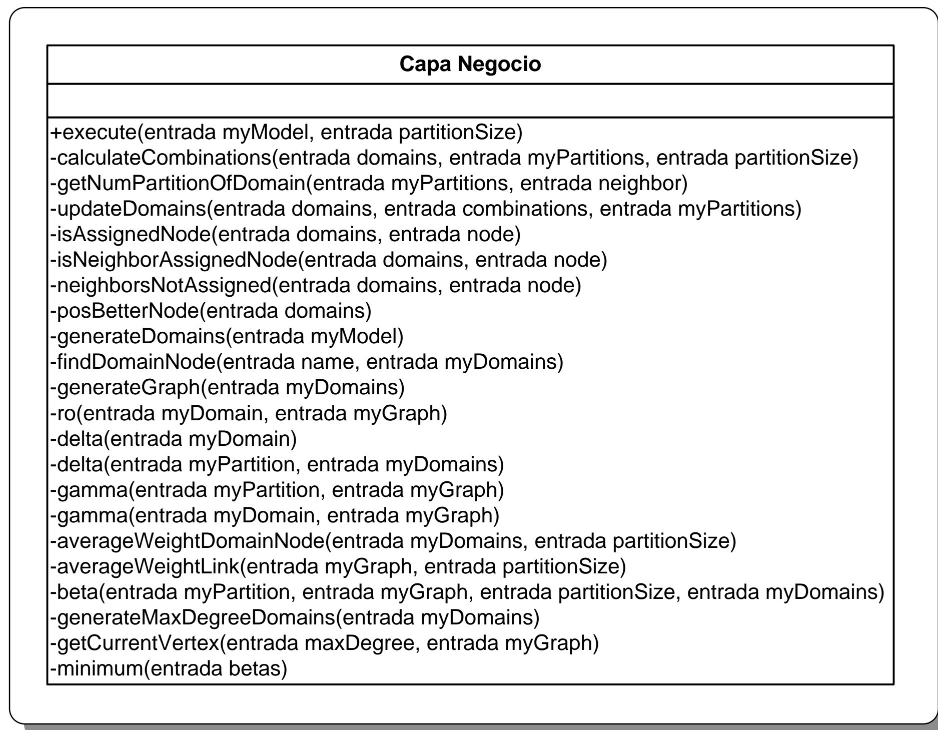


**Figura 20. Interfaz capa negocio**

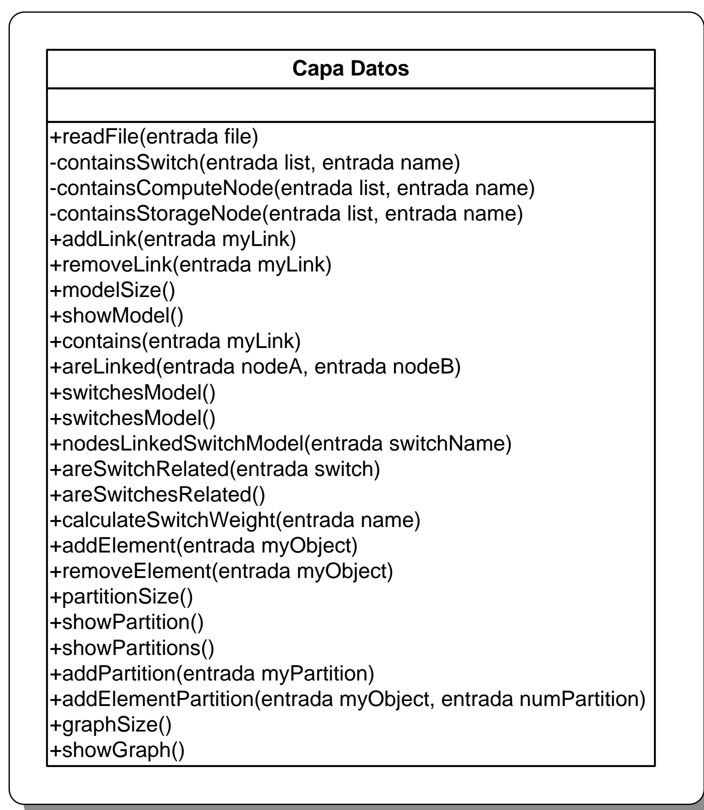


**Figura 21. Interfaz capa datos**

## 4.4. Estructuras

En este apartado se van a describir las estructuras más relevantes usadas para la generación de una solución al problema. Estas estructuras son *Link*, *Model*, *Graph* y *Partition*. Asimismo se darán detalles de otras estructuras menos importantes pero igual de necesarias.

La estructura *Link* representa un enlace entre dos elementos. Es la estructura utilizada tanto por *Model* como por *Graph* para conectar los elementos que lo forman. La estructura *Link* dispone además de un peso del enlace que dependiendo de si forma parte de un *Model* o un *Graph* será distinto, esto se explicará más adelante en esta sección. La figura 22 muestra el tipo de enlace de *Model* que conecta elementos *Switch*, *ComputeNode* y *StorageNode*.

El objeto *Switch* funciona como conector de elementos *ComputeNode* y *StorageNode* dentro de un modelo cualquiera. Un switch puede conectar cualquier tipo de elemento, es decir, *ComputeNode*, *StorageNode* e incluso otros *Switches*.

El objeto *ComputeNode* representa un nodo del modelo que contiene un número de núcleos dentro de él. Se diferencia del objeto *StorageNode* en que los núcleos se dedican únicamente a procesamiento y no almacenamiento. Estos objetos solo pueden formar parte de un enlace si en el otro extremo el objeto es un *Switch*.

El objeto *StorageNode* representa un nodo del modelo que contiene un número de núcleos dentro de él. Se diferencia del objeto *ComputeNode* en que los núcleos se dedican únicamente a almacenamiento y no a procesamiento. Estos objetos solo pueden formar parte de un enlace si en el otro extremo el objeto es un *Switch*.

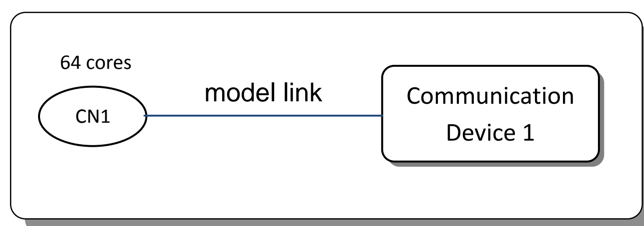


**Figura 22. Enlace de modelo**

Un objeto de tipo *DomainNode* dispone de una lista de nombres que identifican los nodos (*switches*, *computeNodes* y *storageNodes*) del grafo, así como una lista de vecinos de todos estos nodos. A su vez tiene un peso calculado como la suma de los enlaces de los nodos en el modelo y un atributo de asignado que designa si un nodo ha sido fijado a una determinada partición. En la figura 23 se muestran dos tipos de *DomainNode*.

Los enlaces de *Graph* conectan elementos de tipo *DomainNode*. Estos elementos son nodos que engloban todos los elementos conectados a un elemento de tipo *Switch* junto con él. La figura 23 muestra un enlace de *Graph*.

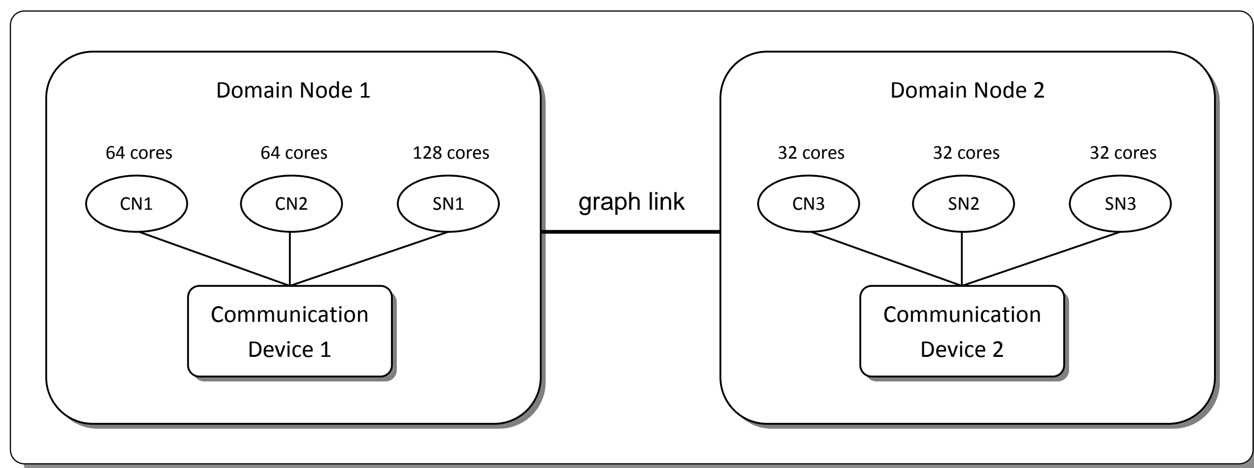


**Figura 23. Enlace de grafo**

La estructura *Model* está formada por enlaces como los de la figura 22. Cada enlace está formado por dos objetos pudiendo ser objetos de tipo *Switch*, *ComputeNode* y *StorageNode*. Puede haber tres tipos de enlaces, el primero un enlace de dos *Switches* y los restantes siendo un enlace de un *Switch* con un *ComputeNode* o un *StorageNode*. Además cada enlace tendrá un peso que tiene valores diferentes dependiendo de qué nodos formen el enlace.

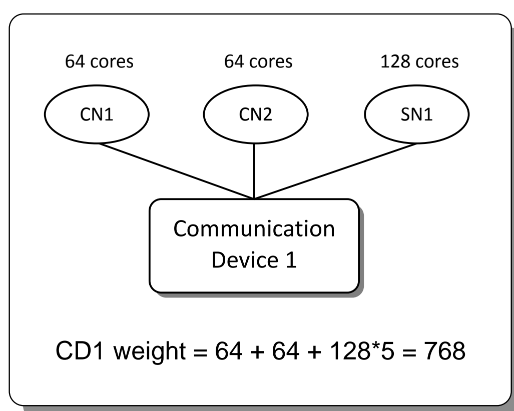


**Figura 24. Cálculo peso enlace modelo**

La forma de calcular el peso se explica a continuación. Cada enlace en el que participe el Switch con nombre name con un ComputeNode tendrá un peso que vendrá dado por el número de núcleos que tenga ese ComputeNode. Cada enlace en el que participe el Switch con

nombre name con un StorageNode tendrá un peso que vendrá dado por el número de núcleos que tenga ese StorageNode multiplicado por 5. Esto se debe a que se simula que los nodos de almacenamiento tienen una latencia 5 veces superior a los ComputeNode.

El peso total del elemento *Switch* vendrá dado por la suma de todos los pesos de cada uno de los elementos *ComputeNode* y *StorageNode* conectados al *Switch*. La figura 24 ilustra un ejemplo de cálculo de pesos de un *Switch*.

Siguiendo con la estructura *Graph*. Cada enlace de la lista está formado por dos extremos y un peso. Esos dos extremos serán objetos de tipo *DomainNode* como los de la figura 23 y el peso vendrá dado por la suma de los pesos de cada *DomainNode* que forma el enlace. La figura 25 muestra en ejemplo que ilustra la manera en la que se calculan los pesos de los enlaces de los grafos.

Para acabar con este apartado se va a proceder a explicar las estructuras *Partition* y *Partitions*. Un objeto de tipo *Partition* está formado por una lista de elementos de tipo *Object*, cada uno de estos elementos representa un nodo (*Switch*, *ComputeNode* o *StorageNode*) del modelo perteneciente a una partición.

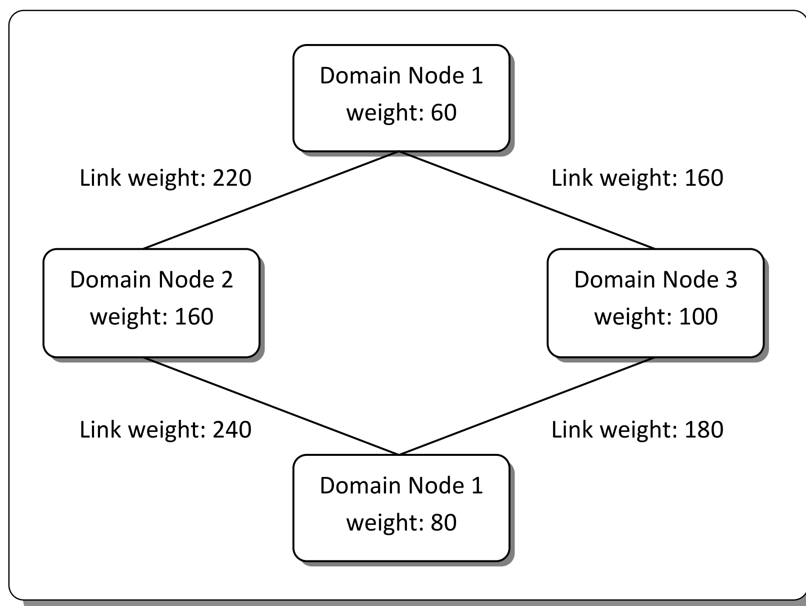


**Figura 25. Cálculo peso enlace grafo**

Un ejemplo de partición viene dado en la figura 26. A partir de los dos DomainNode de la figura 23 se ha supuesto que ambos van a ir a parar a la misma partición, siendo el resultado una fusión de todos sus elementos en una partición.

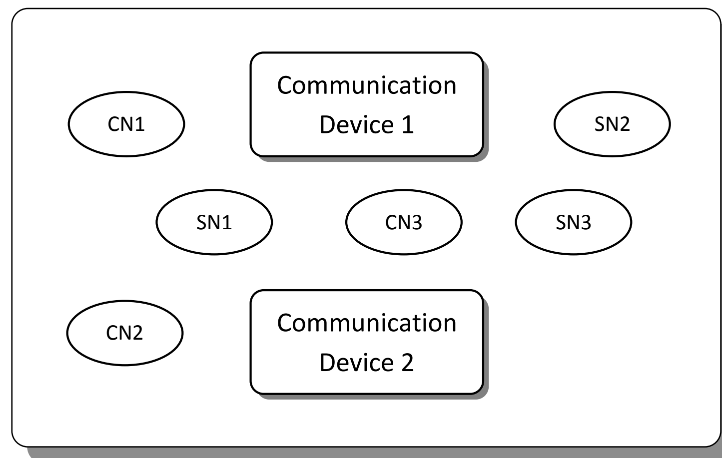


**Figura 26. Ejemplo de partición**

La estructura *Partitions* está formada por una lista de objetos *Partition*. Existirán en el sistema tantas particiones como el usuario quiera, éste podrá elegir el número de particiones y por tanto la longitud de la estructura *Partitions*.

## 4.5. Plataforma de desarrollo

El sistema desarrollado podrá funcionar en mayoría de los terminales del mercado actual, es decir, cualquier terminal que disponga de una JVM (Java Virtual Machine). La figura 27 muestra el logo de Java.

**Figura 27. Logo de Java**

Es preciso decir que la aplicación está optimizada para un tamaño de terminal medio o grande como pueden ser los ordenadores de sobremesa o portátiles. Esto incluye a todos los tipos de máquina y arquitecturas del mercado, como pueden ser los PCs y Mac. La aplicación es

compatible con la práctica totalidad de SSOO del mercado, como pueden ser Windows, Mac OS, Linux, Android, etc. Las figuras 28 y 29 muestran los logos de Apple Inc. y Linux respectivamente.

**Figura 28. Logo de Apple Inc.**

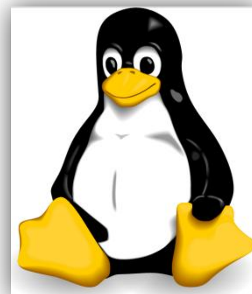

**Figura 29. Logo de Linux**

# 5. Algoritmos

En este apartado se van a presentar los algoritmos desarrollados para la plataforma del sistema. Conviene destacar que la implementación de estos algoritmos es el propósito principal del proyecto, así como tener una plataforma flexible y escalable para agregar nuevos algoritmos que repercutan el menor impacto posible. En primer lugar se detallará el algoritmo de proximidad, continuado por los algoritmos de balanceo simple y el de balanceo.

## 5.1. Proximity partition algorithm

Este apartado se va a dividir en dos secciones, una primera en la que se explica la idea base del algoritmo ayudándose de diagramas explicativos, para terminar con una segunda parte que incluye el pseudocódigo del algoritmo.

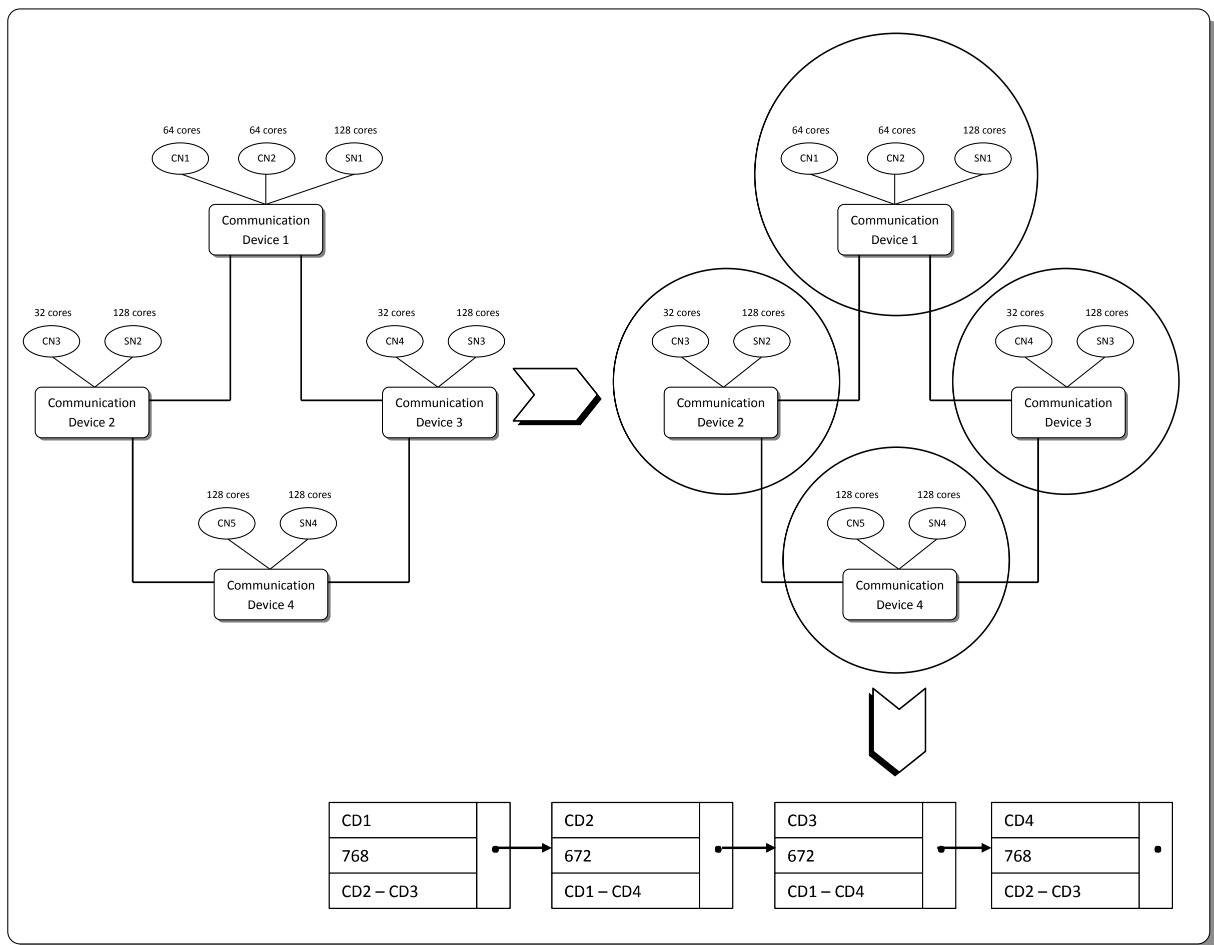


**Figura 30. Diagrama creación dominios**

La idea base del algoritmo de proximidad es la de agrupar los elementos del modelo en particiones según su proximidad. Es decir, se pretende dividir el modelo de tal forma que los elementos que estén conectados directamente participen de la misma partición siempre que sea posible y las circunstancias del problema lo permitan. Este algoritmo parte del pensamiento de que es más probable que dos nodos *Switch* del modelo que estén conectados directamente interactúen entre ambos que dos nodos que no lo estén. Desde luego, se trata de una teoría que no siempre va a ser así y por tanto, la solución al problema que dé este algoritmo será NP-incompleto.

El algoritmo crea una lista de dominios que contendrá los dominios generados a partir del modelo de entrada, tal y como se muestra en la figura 30. Cada dominio contiene una lista de nombres de *Switches*, un peso y una lista de vecinos *Switches*. Una vez dividido el modelo en dominios se procede a insertar en cada una de las particiones, en una primera iteración inserta n dominios, tantos como número de particiones. En iteraciones siguientes se inserta a cada partición aquel dominio vecino de uno de los que ya está insertado, dando preferencia a aquellos dominios con menos vecinos asignados. A continuación se muestra el pseudocódigo de lo explicado en este párrafo.

**PROXIMITY PARTITION ALGORITHM**

```
Requisitos: miModelo, tamParticion
 1: miDominio = null, misDominios = null,
 2: misParticiones = null, misCombinaciones = null
 3: iterador = 0, mejorPosicion = 0
 4: SI miModelo bien_formado && válido
 5:   SI tamParticion != 1
 6:     //se generan los dominios a partir del modelo
 6:     generar(misDominios, miModelo)
 7:     MIENTRAS iterador < tamParticion
 8:       mejorPosicion = obtener_posicion(misDominios)
 9:       //se almacena el nodo con mejor posición en auxiliar
10:       miDominio = obtener(mejorPosicion, misDominios)
11:       //se añade el dominio a la partición
12:       añadir(miDominio, misParticiones)
13:       iterador++
14:     FIN MIENTRAS
15:     MIENTRAS iterador < tamaño_lista_dominios
16:       misCombinaciones = calcular_combinaciones(misDominios,
17:                         misParticiones, tamParticion)
18:       MIENTRAS misCombinaciones != null
```

```
19:         //se añade el dominio a la partición
20:         añadir_combinacion(misCombinaciones, misParticiones)
21:         borrar_combinacion(misCombinaciones)
22:         iterador++
23:       FIN MIENTRAS
24:     FIN MIENTRAS
25:   SI NO tamParticion = 1
26:     //se añade todo el modelo a una única partición
27:     añadir(miModelo, misParticiones)
28:   FIN SI
29: FIN SI
```

## 5.2. Simple balanced partition algorithm

Este apartado se va a dividir en dos secciones, una primera en la que se explica la idea base del algoritmo ayudándose de diagramas explicativos, para terminar con una segunda parte que incluye el pseudocódigo del algoritmo.

El concepto del algoritmo tiene que ver con el balanceo de carga, es decir, el algoritmo trata de dividir la carga de todos los elementos del modelo en particiones. De este modo, una vez ejecutado el algoritmo, cada partición tendrá un peso lo más parecido posible a todas las demás. El algoritmo crea una lista de dominios que contendrá los dominios generados a partir del modelo de entrada, tal y como se muestra en la figura 31. Cada dominio contiene una lista de nombres de *Switches*, un peso y una lista de vecinos *Switches*. A su vez se genera un grafo que une cada dominio con sus respectivos enlaces.

La manera en que se define la carga en este algoritmo es la siguiente. Cada dominio tiene su propio peso, calculado como la suma de los núcleos de los nodos internos, teniendo un peso 5 veces mayor los *StorageNode* sobre los *ComputeNode* por convención. Una vez se tienen estos pesos se pasan a calcular los pesos de los enlaces del grafo, que no será más que la suma de los pesos de los dos *DomainNode* que conecte, como se puede ver en la figura 31.

La manera que tiene el algoritmo de elegir el dominio a insertar en la partición está relacionada con una función beta. Esta función lo que hace es simular el peso que tendría una partición a partir de una inserción de un dominio en ella. Es decir, para un dominio se simulan los pesos en cada una de las particiones y se inserta finalmente el dominio en la partición con menor peso de todas.

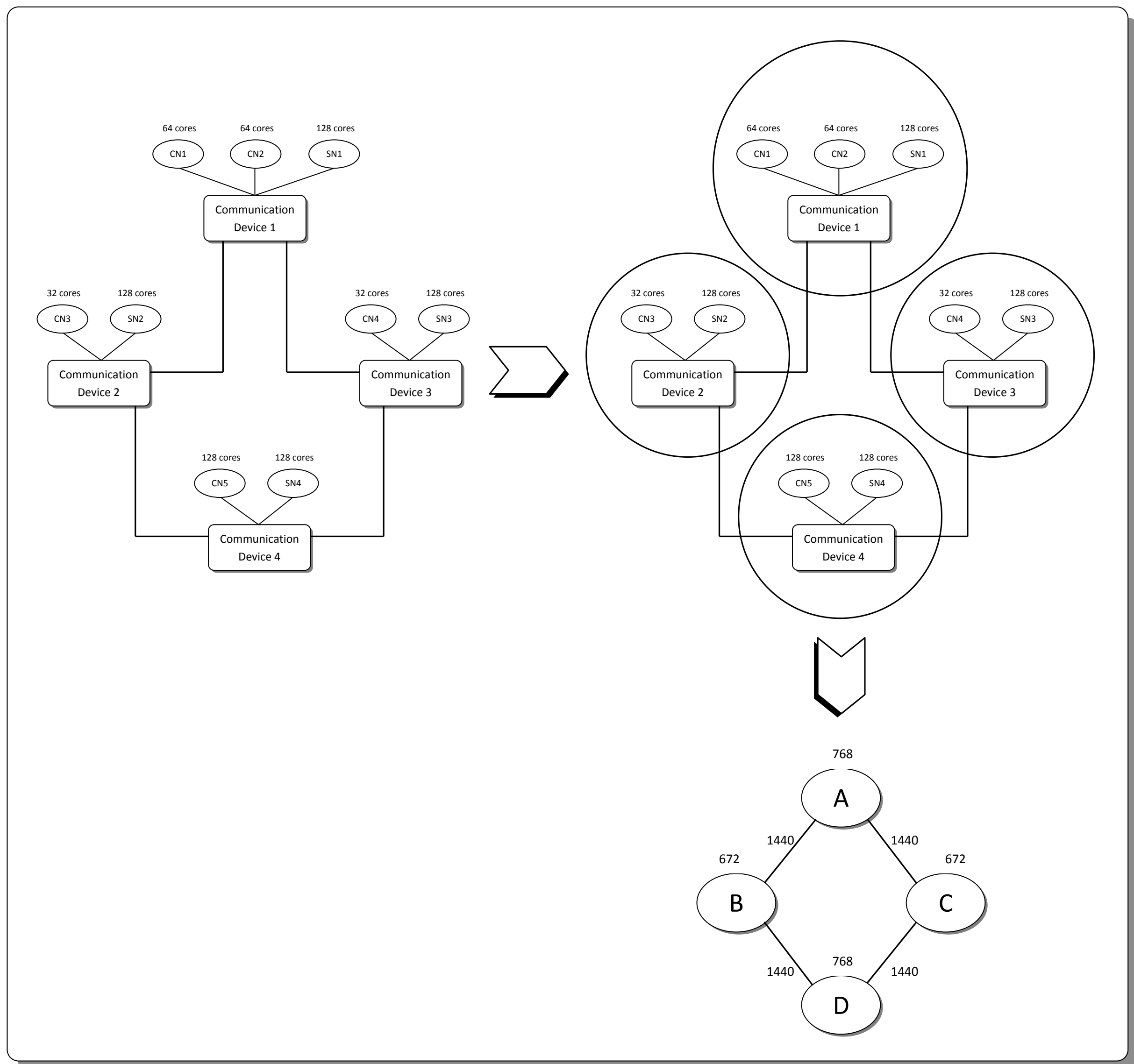


**Figura 31. Diagrama creación de grafos**

Para entenderlo mejor, si tenemos un dominio D1 con peso 100 y dos particiones P1 y P2 con pesos 150 y 200 (ya tiene dominios insertados), el algoritmo insertará D1 en P1 porque beta_D1 = 250 y beta_D2= 300. Así se consigue que en cada iteración el balanceo de carga sea lo más equilibrado posible. A continuación se muestra el pseudocódigo de lo explicado en este párrafo.

**SIMPLE BALANCED PARTITION ALGORITHM**

```
Requisitos: miModelo, tamParticion
 1: misDominios = null, misParticiones = null,
 2: misParticionesAux = null, misBetas = null,
```

```
 3: miGrafo = null, dominioActual = null, iterador = 0, j = 0
 4: SI miModelo bien_formado && válido
 5:   SI tamParticion != 1
 6:     //se generan los dominios a partir del modelo
 7:     generar(misDominios, miModelo)
 8:     generar(miGrafo, miModelo)
 9:     MIENTRAS misDominios != null
10:       //se almacena el primer nodo de la lista
11:       dominioActual = primero(misDominios)
12:       iterador = 0
13:       MIENTRAS iterador < tamParticion
14:         //se añaden los dominios a particiones auxiliares
15:         añadir(dominioActual, misParticionesAux, iterador)
16:         misBetas = beta(misParticionesAux, miGrafo, misDominios,
17:                         tamParticion)
18:         iterador++
19:       FIN MIENTRAS
20:       j = minimo(misBetas)
21:       //se añade el dominio en la partición óptima según beta
22:       añadir(dominioActual, misParticiones, j)
23:        borrar(dominioActual, misDominios)
24:     FIN MIENTRAS
25:   SI NO tamParticion = 1
26:     //se añade todo el modelo a una única partición
27:     añadir(miModelo, misParticiones)
28:   FIN SI
29: FIN SI
```

## 5.3. Balanced partition algorithm

Este apartado se va a dividir en dos secciones, una primera en la que se explica la idea base del algoritmo ayudándose de diagramas explicativos, para terminar con una segunda parte que incluye el pseudocódigo del algoritmo.

Como su nombre indica el concepto del algoritmo va a ser muy similar al anterior el *SimpleBalancedPartitionAlgorithm*. Se puede decir que es una evolución de éste último en el tema de los pesos y alguna cosa más. Se introducen dos factores nuevos que hacen mejorar el algoritmo.

El primer factor es la elección del dominio a insertar en las particiones. Si en el *SimpleBalancedPartitionAlgorithm* se insertaba según el orden del fichero de entrada del modelo, en el *BalancedPartitionAlgorithm* se da un paso adelante y se inserta el dominio óptimo. Se entiende por dominio óptimo aquel que siendo uno de los dominios con mayor grado en su vértice del grafo, además también contiene el menor valor de la función *ro*.

La función *ro*, es una función que calcula el ratio entre el peso del dominio y el peso de los enlaces del dominio. Para aclarar conceptos, si este ratio es alto quiere decir que las comunicaciones tienen poco impacto a la hora de insertar en una partición y por tanto que el peso del dominio es el factor más determinante. Si el valor de ro es bajo, las comunicaciones tendrán bastante importancia en ese dominio, y es por esto que se necesita que estos dominios sean los primeros en insertar en las particiones.

El otro factor nuevo, ya se ha comentado en el párrafo anterior y es el peso de los enlaces. Si bien, en *SimpleBalancedPartitionAlgorithm* la función beta solo tenía en cuenta pesos de dominios, en *BalancedPartitionAlgorithm* va a tener en cuenta también el peso de los enlaces del grafo formado a partir del modelo. Para entendernos, beta ahora intenta balancear teniendo en cuenta tanto el peso de los dominios como de los enlaces en forma de suma de ambos.

Un ejemplo simplificado del nuevo funcionamiento de beta. Si el *DomainNode* D1 tiene peso dominio = 100 y peso enlaces = 50 insertando en la partición P1; y D1 tiene peso dominio = 80 y peso enlaces = 100 insertando en la partición P2; el algoritmo insertará en la partición que menor valor de beta se tenga, es decir en P2 en este caso. A continuación se muestra el pseudocódigo de lo explicado en este párrafo.

**BALANCED PARTITION ALGORITHM**

```
Requisitos: miModelo, tamParticion
  1: misDominios = null, misParticiones = null,
  2: misParticionesAux = null, misBetas = null, misMayorGrado = null
  3: miGrafo = null, dominioActual = null, iterador = 0, j = 0
  4: SI miModelo bien_formado && válido
  5:   SI tamParticion != 1
  6:     //se generan los dominios a partir del modelo
  7:     generar(misDominios, miModelo)
  8:     generar(miGrafo, miModelo)
  9:     MIENTRAS misDominios != null
 10:       //se genera una lista con los dominios con mayor grado en
 11:       //sus vértices
 12:       generar(misMayorGrado, misDominios)
```

```
13:       //se almacena el mejor nodo de la lista
14:       dominioActual = minimo_ro(misMayorGrado, miGrafo)
15:       iterador = 0
16:       MIENTRAS iterador < tamParticion
17:         //se añaden los dominios a particiones auxiliares
18:         añadir(dominioActual, misParticionesAux, iterador)
19:         misBetas = beta(misParticionesAux, miGrafo, misDominios,
20:                         tamParticion)
21:         iterador++
22:       FIN MIENTRAS
23:       j = minimo(misBetas)
24:       //se añade el dominio en la partición óptima según beta
25:       añadir(dominioActual, misParticiones, j)
26:       borrar(dominioActual, misDominios)
27:     FIN MIENTRAS
28:   SI NO tamParticion = 1
29:     //se añade todo el modelo a una única partición
30:     añadir(miModelo, misParticiones)
31:   FIN SI
32: FIN SI
```

# 6. Pruebas

En esta sección se pasa a detallar las pruebas realizadas junto con sus resultados. En primera instancia se explica la manera en que se van a realizar las pruebas, para luego exponer el entorno en el que han sido ejecutadas. Por último se muestran todos los resultados y posibles comparativas surgidas a raíz de estos.

## 6.1. Plan de pruebas

En este apartado se va a describir el plan seguido a la hora de efectuar todas las pruebas y simulaciones. En primer lugar es necesario diseñar una serie de modelos, se ha considerado que con tres sería suficiente. Un primer modelo sencillo llamado *model_presentation* que incluye 4 switches, 5 nodos de cómputo y 4 nodos de almacenamiento, mostrado en la figura 32.

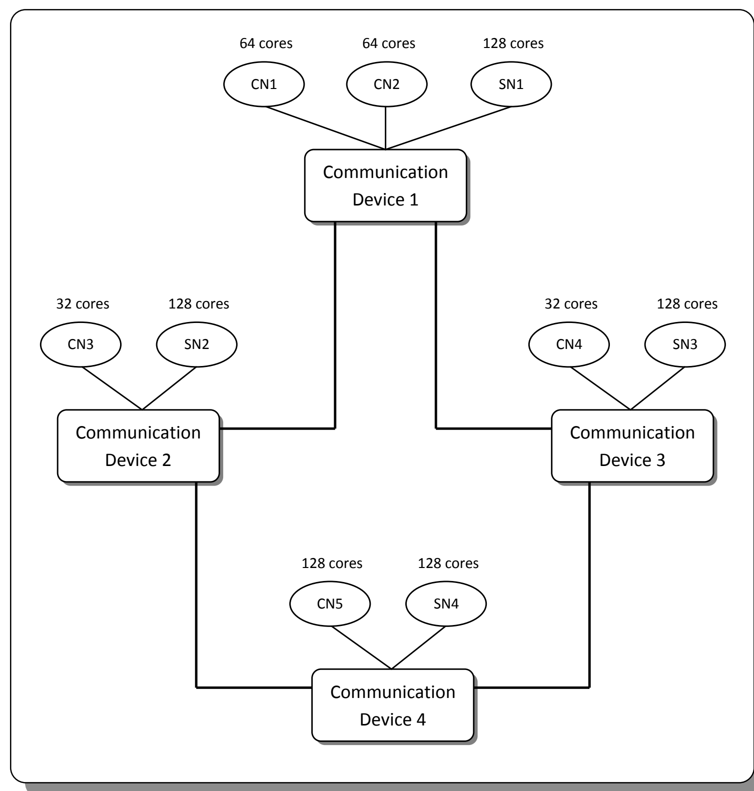


**Figura 32. model_presentation**

Un segundo modelo llamado *model_1* en base al primero añadiendo algunos elementos nuevos. Incluye 6 switches, 7 nodos de cómputo y 8 nodos de almacenamiento. Este modelo se muestra en la figura 33.

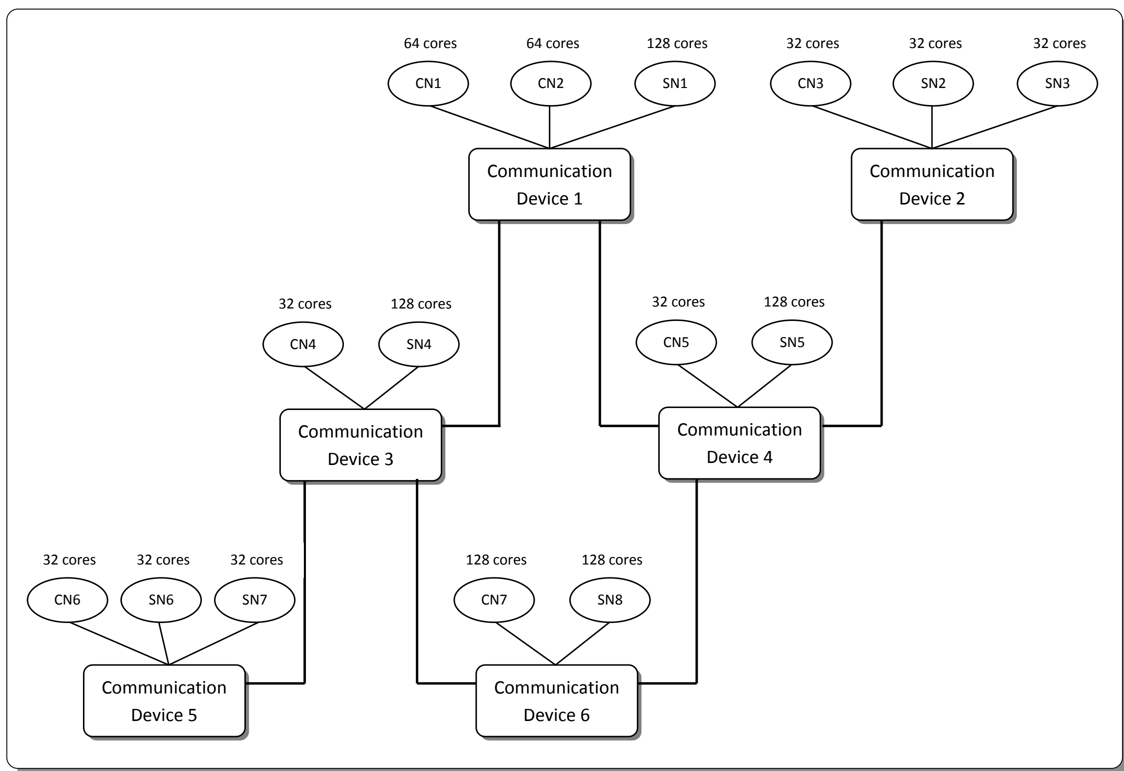


**Figura 33. model_1**

Un último modelo más completo y sofisticado llamado *model_2*. Incluye 8 switches, 14 nodos de cómputo y 14 nodos de almacenamiento. Este modelo se expone en la figura 34.

Una vez se tienen los modelos de entrada, es necesario fijar qué pruebas se van a realizar para cada uno de ellos. A continuación se exponen las pruebas a realizar para cada uno de estos, en total son 8 pruebas:

- *model_presentation* para 2 y 4 particiones.
- *model_1* para 2, 3 y 4 particiones.
- *model_2* para 2, 4 y 8 particiones.

Conviene recordar que estas 8 pruebas han de realizarse para cada uno de los tres algoritmos existentes en la plataforma de manera que el número total de pruebas será de 24. La figura 35 muestra una tabla explicativa de todas las pruebas.

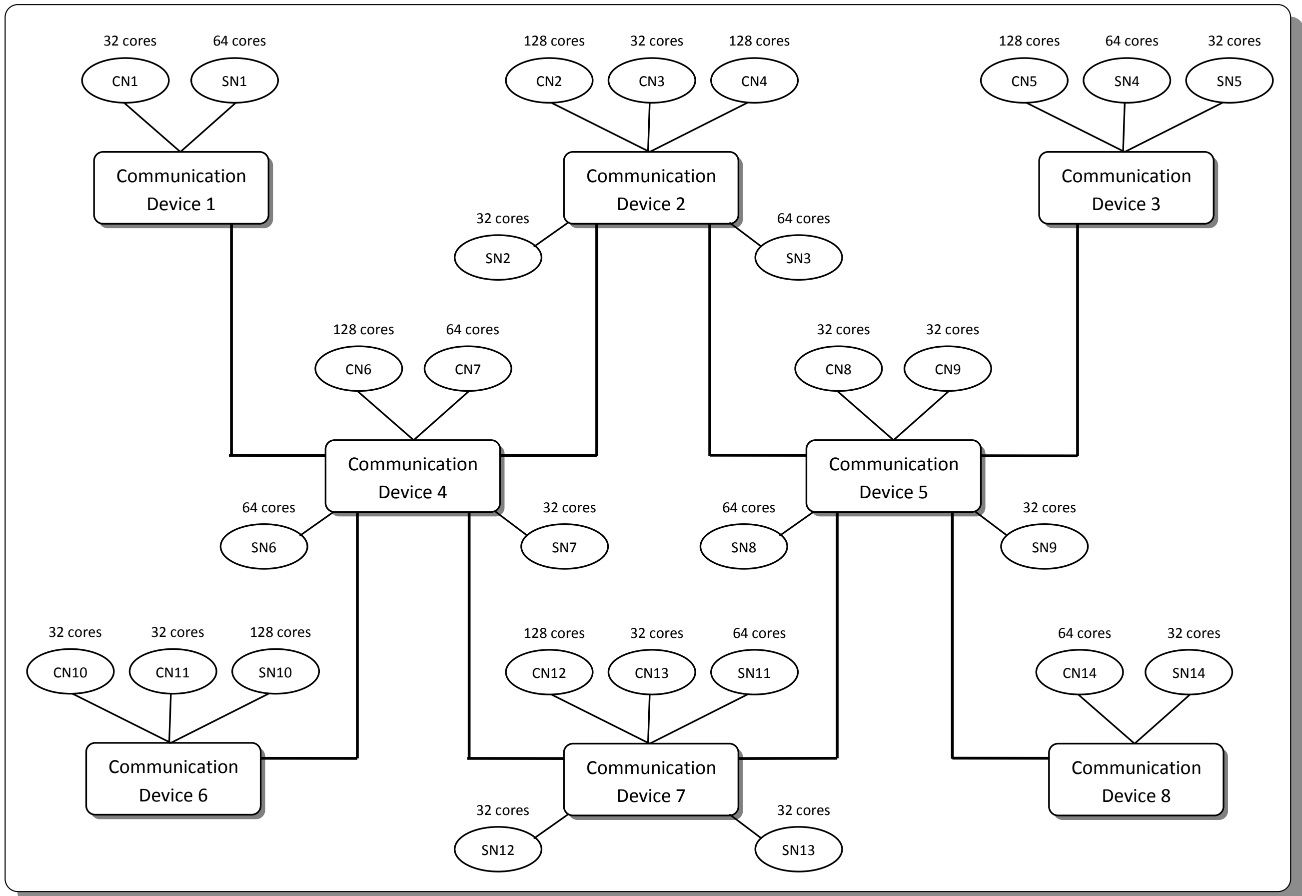


**Figura 34. model_2**

| | model_presentation | | | model_1 | | | model_2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Particiones** | ***1*** | ***2*** | ***4*** | ***1*** | ***2*** | ***4*** | ***1*** | ***2*** | ***4*** | ***8*** |
| **Proximity** | P1 | P2 | P3 | P10 | P11 | P12 | P19 | P20 | P21 | P22 |
| **Simple_Balanced** | P4 | P5 | P6 | P13 | P14 | P15 | P23 | P24 | P25 | P26 |
| **Balanced** | P7 | P8 | P9 | P16 | P17 | P18 | P27 | P28 | P29 | P30 |

**Figura 35. Tabla de pruebas**

Por último hay que reseñar que todas las ejecuciones con sus particiones resultantes se han de ejecutar en el simulador SIMCAN para obtener los datos de rendimiento.

## 6.2. Entorno de pruebas

Como bien se ha expuesto en el apartado anterior del plan de pruebas, éstas se dividen en dos fases. Una primera en la que se efectúan las ejecuciones del sistema sobre los diferentes modelos de entrada existentes con el fin de conseguir los resultados sobre las particiones obtenidas para cada algoritmo. Y una segunda fase en la que, con la ayuda del simulador SIMCAN, se realizan simulaciones para cada uno de los resultados de las ejecuciones de la primera fase con el fin de encontrar resultados sobre rendimiento. La figura 36 muestra el logo del proyecto SIMCAN.

**Figura 36. Logo proyecto SIMCAN**

El entorno para las pruebas de la primera fase se basa únicamente en un ordenador portátil Dell Vostro 1510 con sistema operativo Microsoft Windows Vista Business, procesador Intel Core Duo T5670 a 1,80 GHz y una memoria RAM de 3,00 GB.

Para las últimas pruebas o simulaciones del SIMCAN el entorno ofrecido es el propio de dicha aplicación, con procesador Intel Top Pentium XL Core 2 Quad Q8300 a 2,50 GHz y una memoria RAM de 4,00 GB.

SIMCAN ofrece la posibilidad de paralelizar la ejecución de la simulación a partir de las particiones obtenidas anteriormente. Para esto es necesario incluir dentro de la aplicación los datos referentes a nodos, switches y enlaces tal y como se han obtenido de las primeras pruebas y divididas por particiones.

Estas simulaciones se ejecutan sobre un clúster de ordenadores como el mostrado en la figura 37.

Como se puede observar dispone en su esencia de los mismos elementos que un ordenador personal, con la excepción de que diversos módulos como los discos y los procesadores están multiplicados en cantidad. Todos ellos comparten un único sistema de memoria.

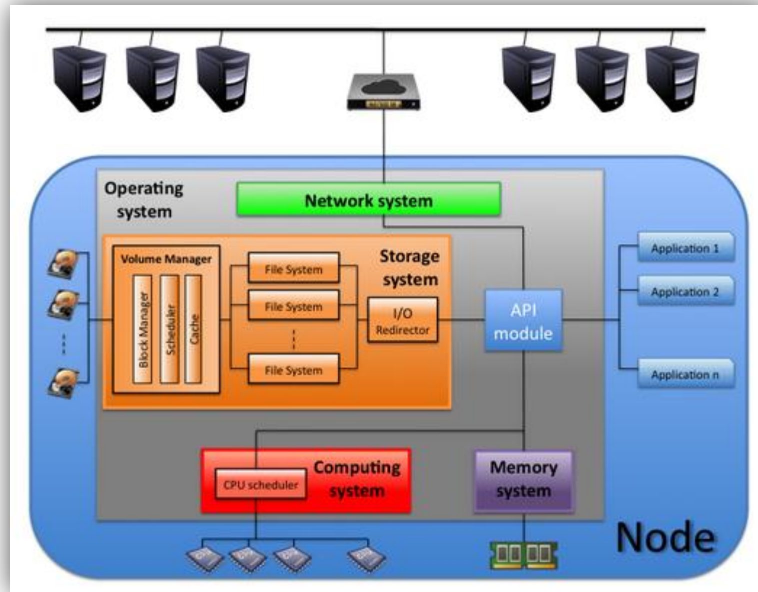


**Figura 37. Arquitectura SIMCAN**

## 6.3. Resultados

En este apartado se muestran los resultados de las pruebas realizadas con la aplicación tal y como se ha explicado en el apartado de plan de pruebas.

### *Model_presentation – 1 partición – Proximity*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – CD2 – CN4 – SN3 – CD3 – CN5 – SN4 – CD4

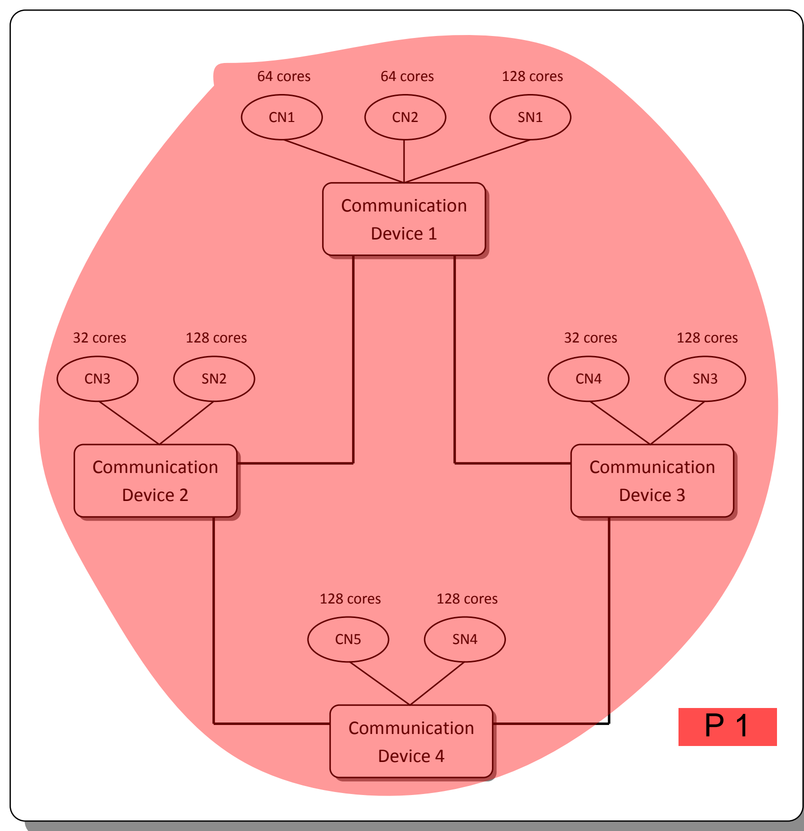


**Figura 38. Prueba 1**

### *Model_presentation – 2 particiones – Proximity*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – CD2
- ***Partición 2:*** CN4 – SN3 – CD3 – CN5 – SN4 – CD4

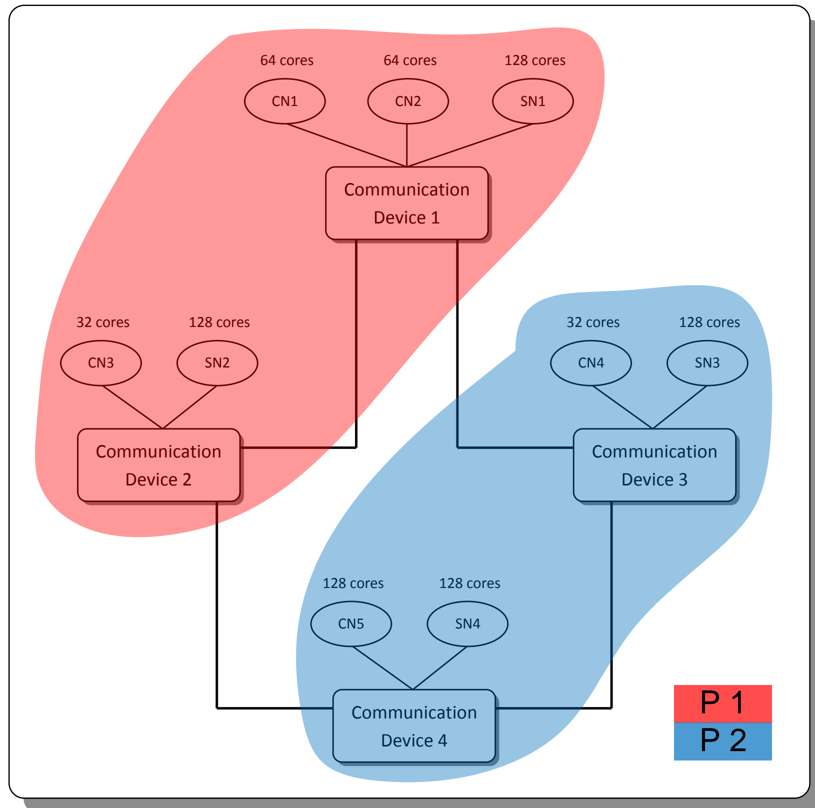


**Figura 39. Prueba 2**

## *Model_presentation – 4 particiones – Proximity*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1
- ***Partición 2:*** CN5 – SN4 – CD4
- ***Partición 3:*** CN3 – SN2 – CD2
- ***Partición 4:*** CN4 – SN3 – CD3

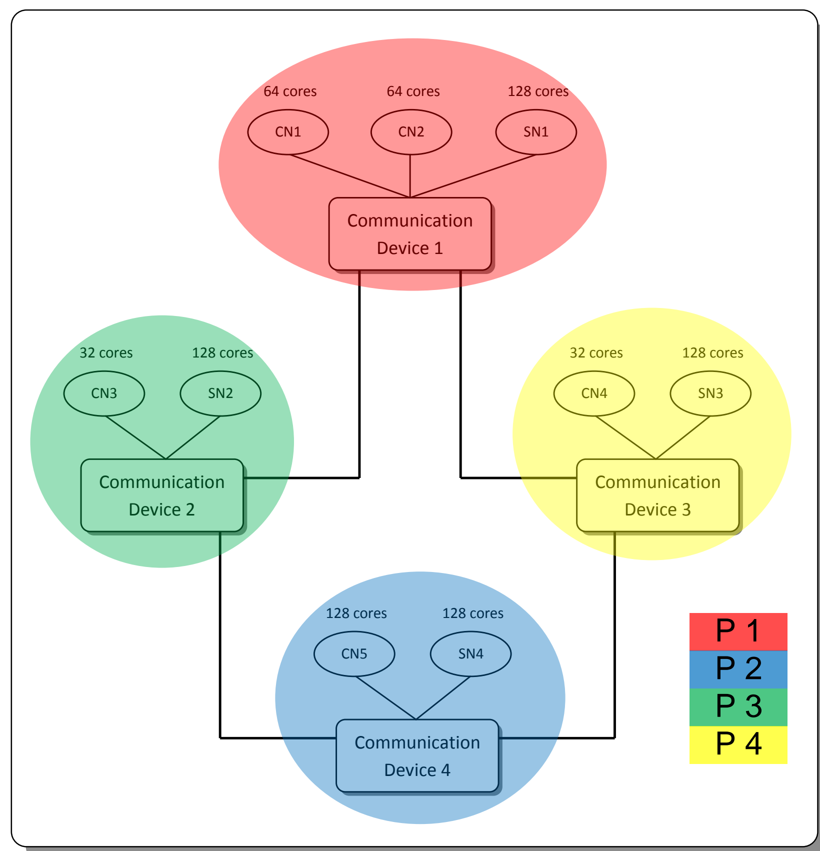


**Figura 40. Prueba 3**

## *Model_presentation – 1 partición – Simple balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – CD2 – CN4 – SN3 – CD3 – CN5 – SN4 – CD4

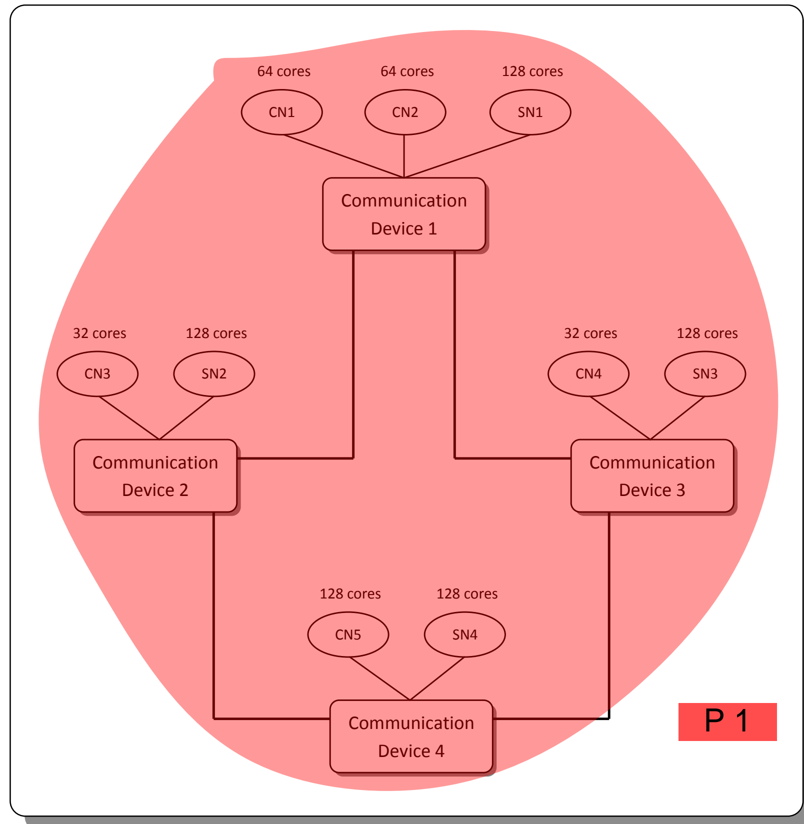


**Figura 41. Prueba 4**

## *Model_presentation – 2 particiones – Simple balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN5 – SN4 – CD4
- ***Partición 2:*** CN3 – SN2 – CD2 – CN4 – SN3 – CD3

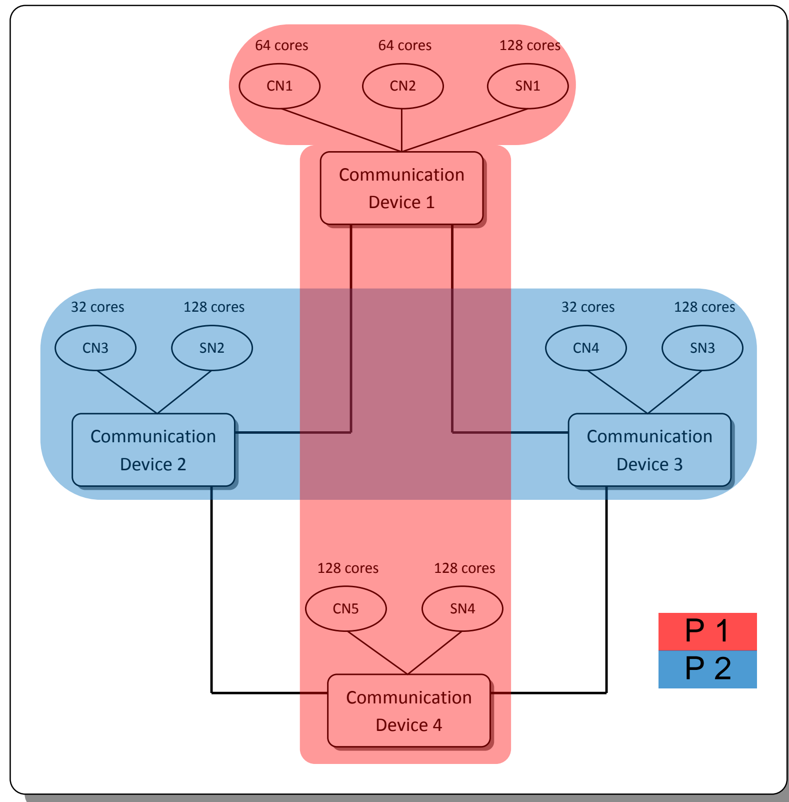


**Figura 42. Prueba 5**

### *Model_presentation – 4 particiones – Simple balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1
- ***Partición 2:*** CN3 – SN2 – CD2
- ***Partición 3:*** CN4 – SN3 – CD3
- ***Partición 4:*** CN5 – SN4 – CD4

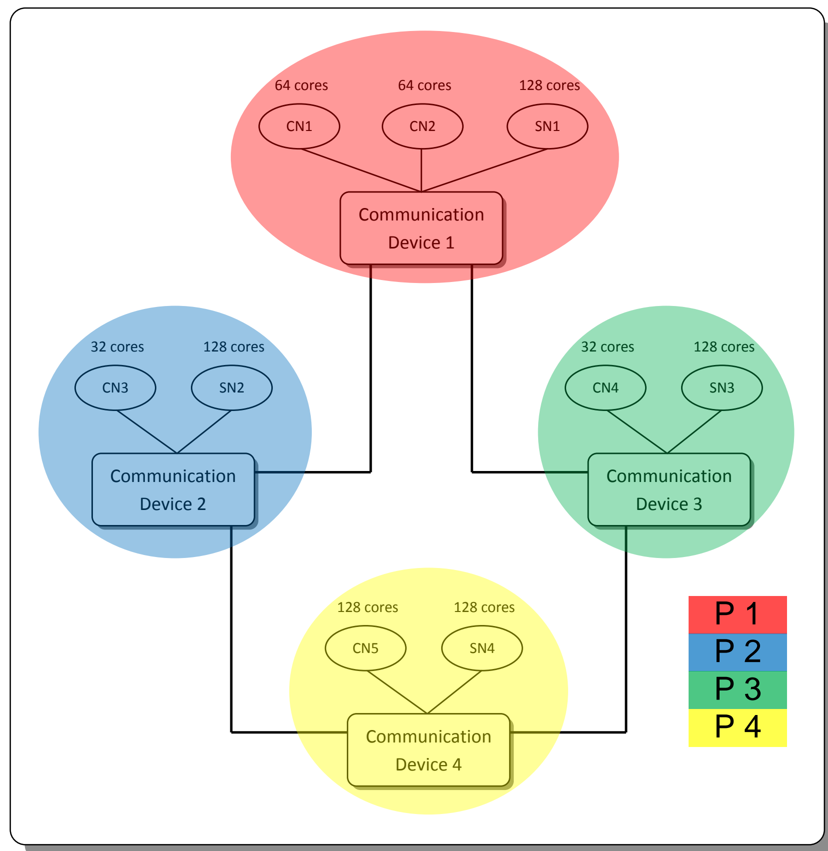


**Figura 43. Prueba 6**

## *Model_presentation – 1 partición – Balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – CD2 – CN4 – SN3 – CD3 – CN5 – SN4 – CD4

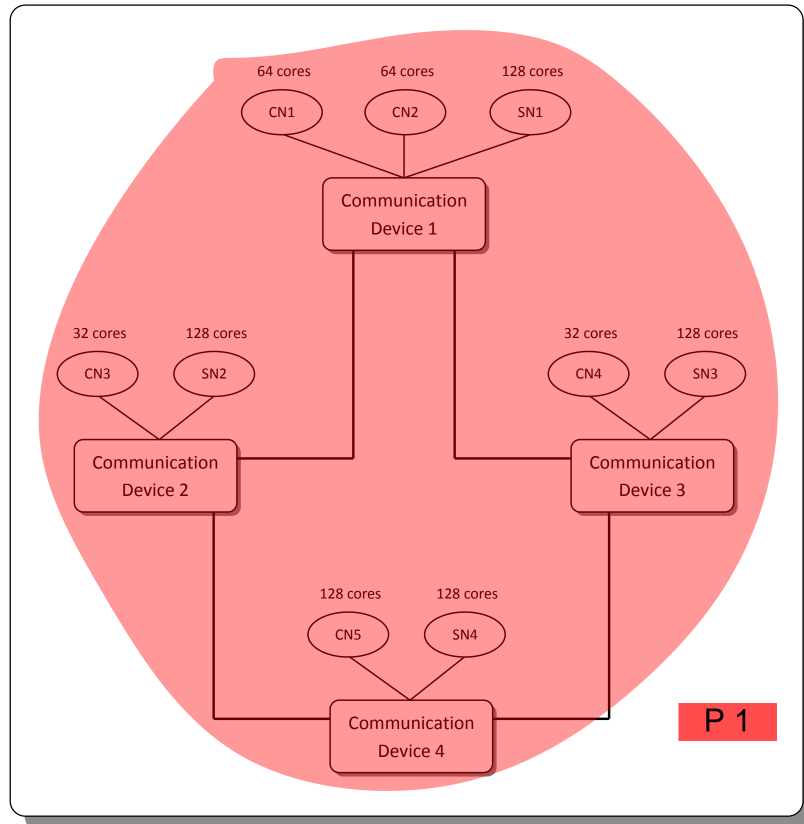


**Figura 44. Prueba 7**

## *Model_presentation – 2 particiones – Balanced*

- ***<u>Partición 1:</u>*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – CD2
- ***<u>Partición 2:</u>*** CN4 – SN3 – CD3 – CN5 – SN4 – CD4

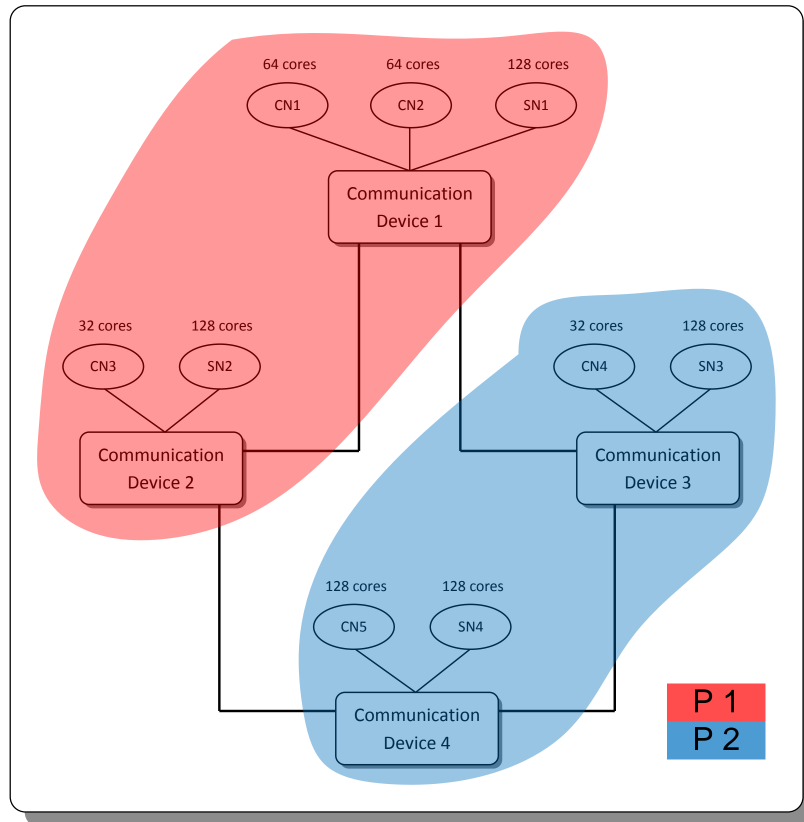


**Figura 45. Prueba 8**

## *Model_presentation – 4 particiones – Balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1
- ***Partición 2:*** CN3 – SN2 – CD2
- ***Partición 3:*** CN4 – SN3 – CD3
- ***Partición 4:*** CN5 – SN4 – CD4

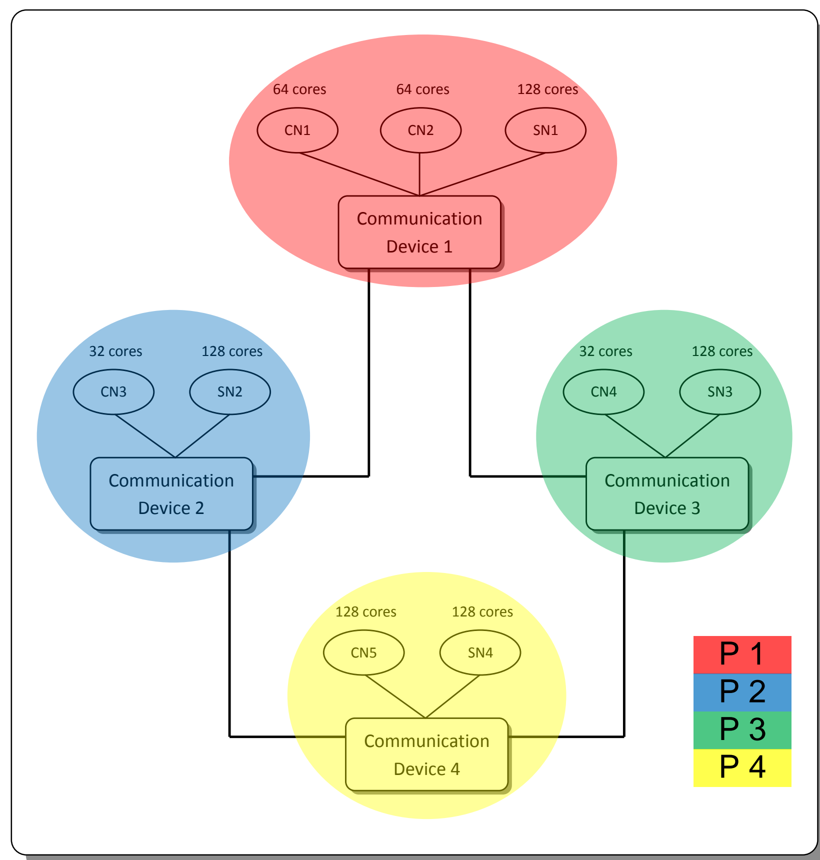


**Figura 46. Prueba 9**

## *Model_1 – 1 partición – Proximity*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – SN3 – CD2 – CN4 – SN4 – CD3 – CN5 – SN5 – CD4 – CN6 – SN6 – SN7 – CD5 – CN7 – SN8 – CD6

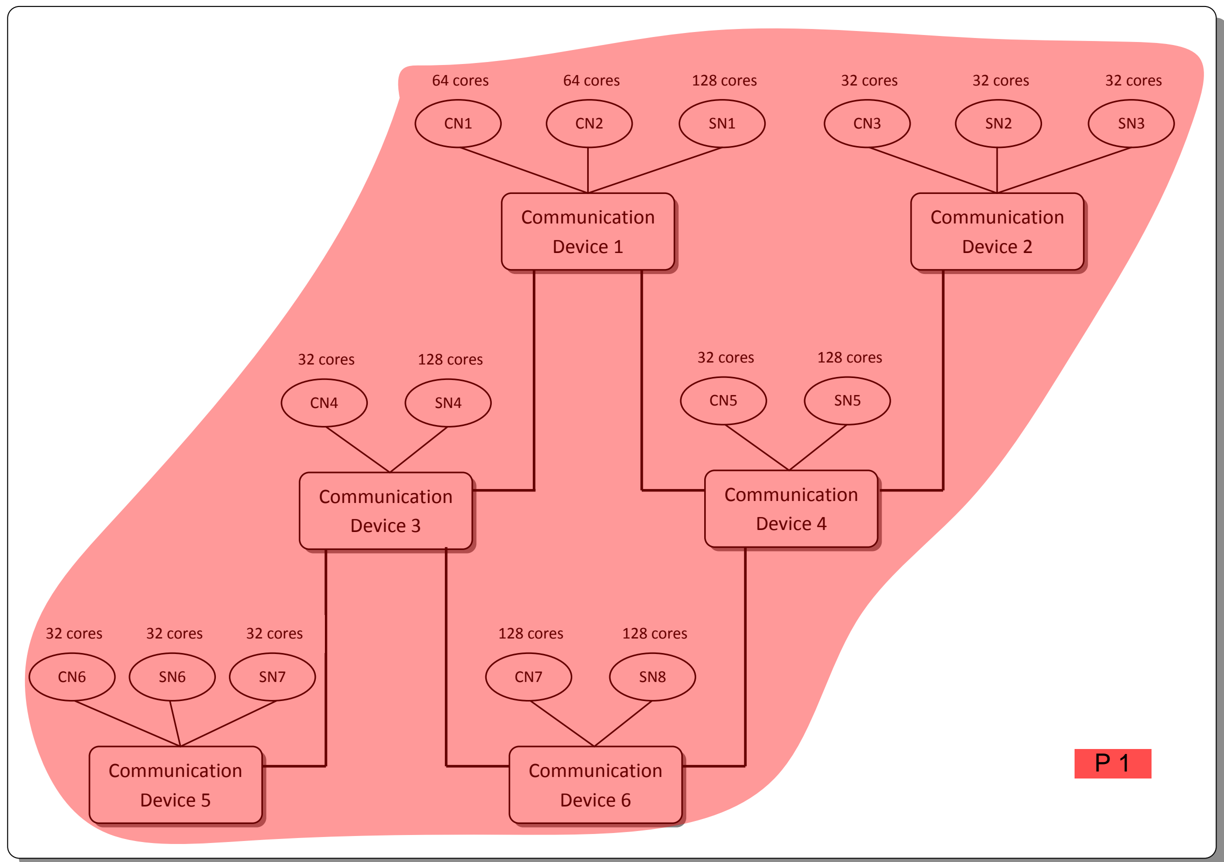


**Figura 47. Prueba 10**

## *Model_1 – 2 particiones – Proximity*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN4 – SN4 – CD3 – CN6 – SN6 – SN7 – CD5
- ***Partición 2:*** CN3 – SN2 – SN3 – CD2 – CN5 – SN5 – CD4 – CN7 – SN8 – CD6

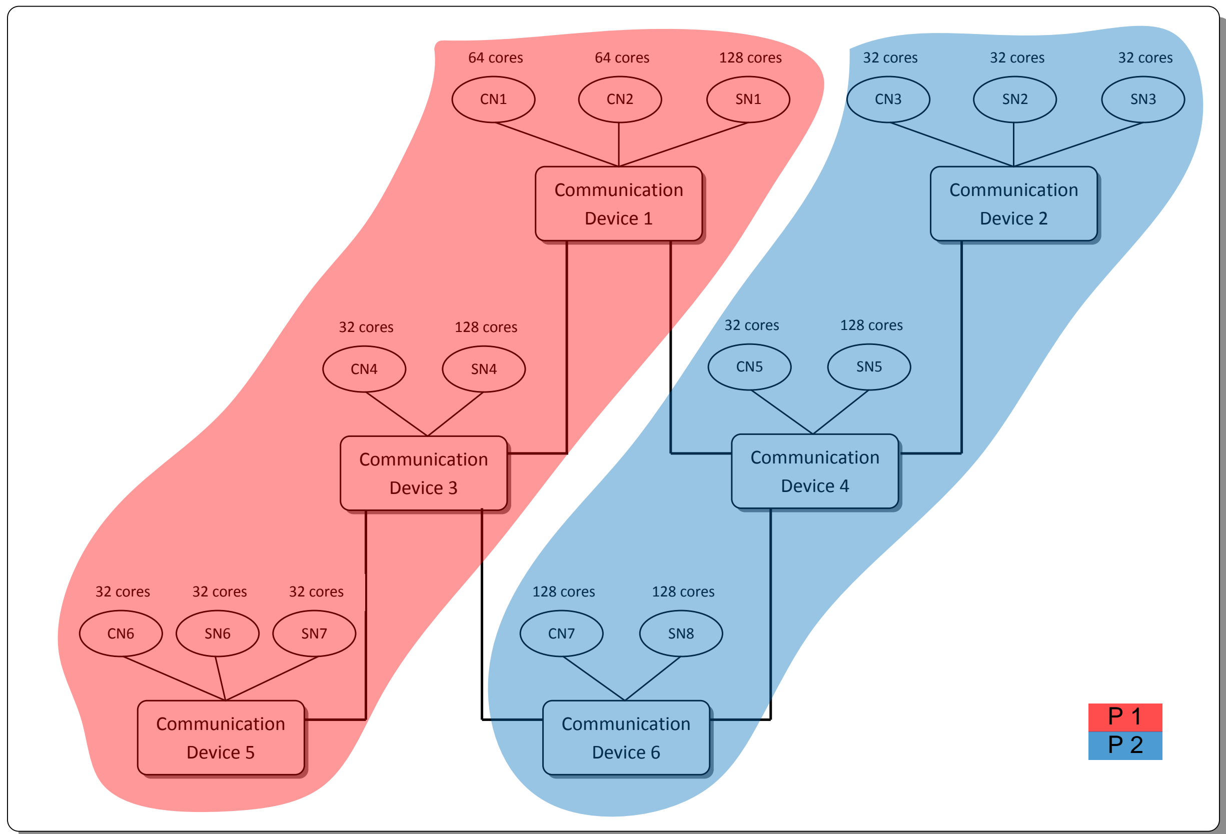


**Figura 48. Prueba 11**

## *Model_1 – 4 particiones – Proximity*

- ***Partición 1:*** CN4 – SN4 – CD3 – CN6 – SN6 – SN7 – CD5
- ***Partición 2:*** CN3 – SN2 – SN3 – CD2 – CN5 – SN5 – CD4
- ***Partición 3:*** CN1 – CN2 – SN1 – CD1
- ***Partición 4:*** CN7 – SN8 – CD6

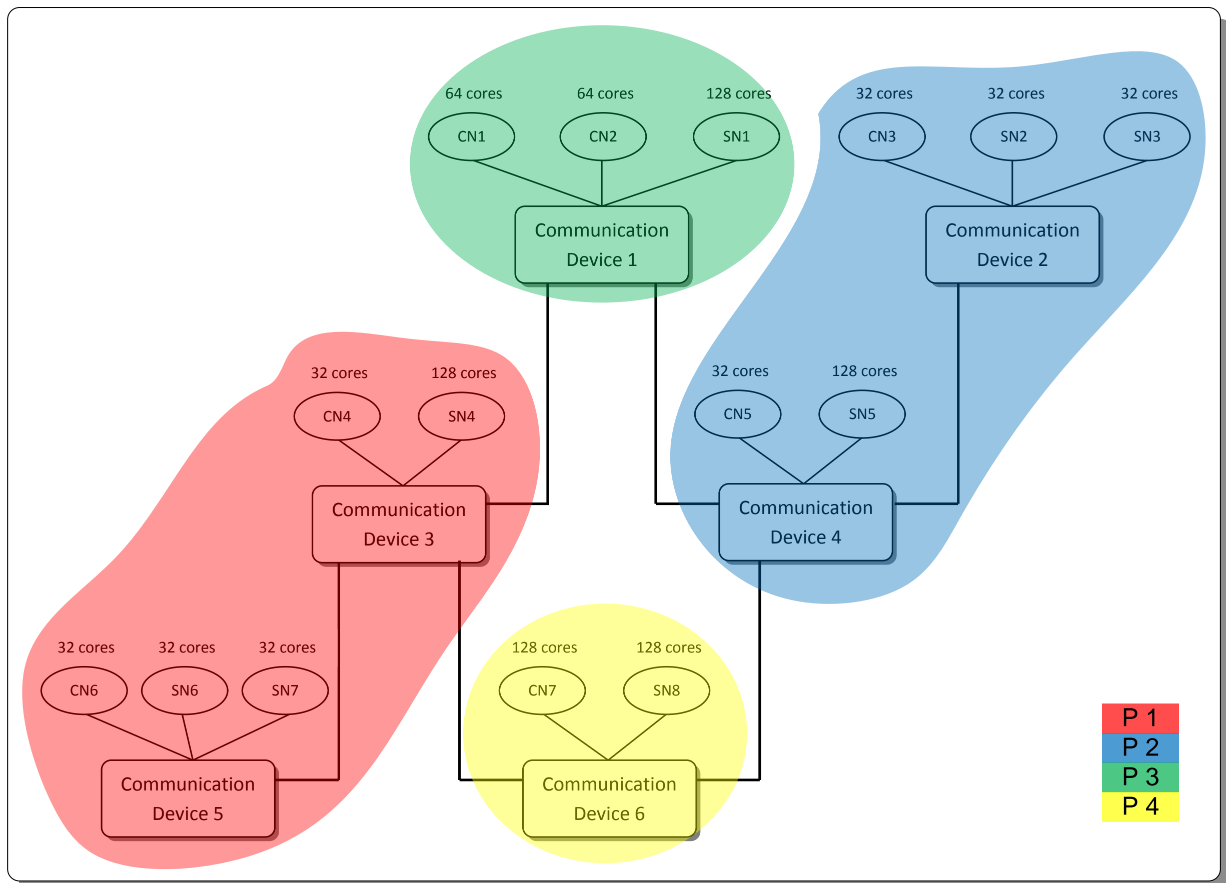


**Figura 49. Prueba 12**

## *Model_1 – 1 partición – Simple Balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – SN3 – CD2 – CN4 – SN4 – CD3 – CN5 – SN5 – CD4 – CN6 – SN6 – SN7 – CD5 – CN7 – SN8 – CD6

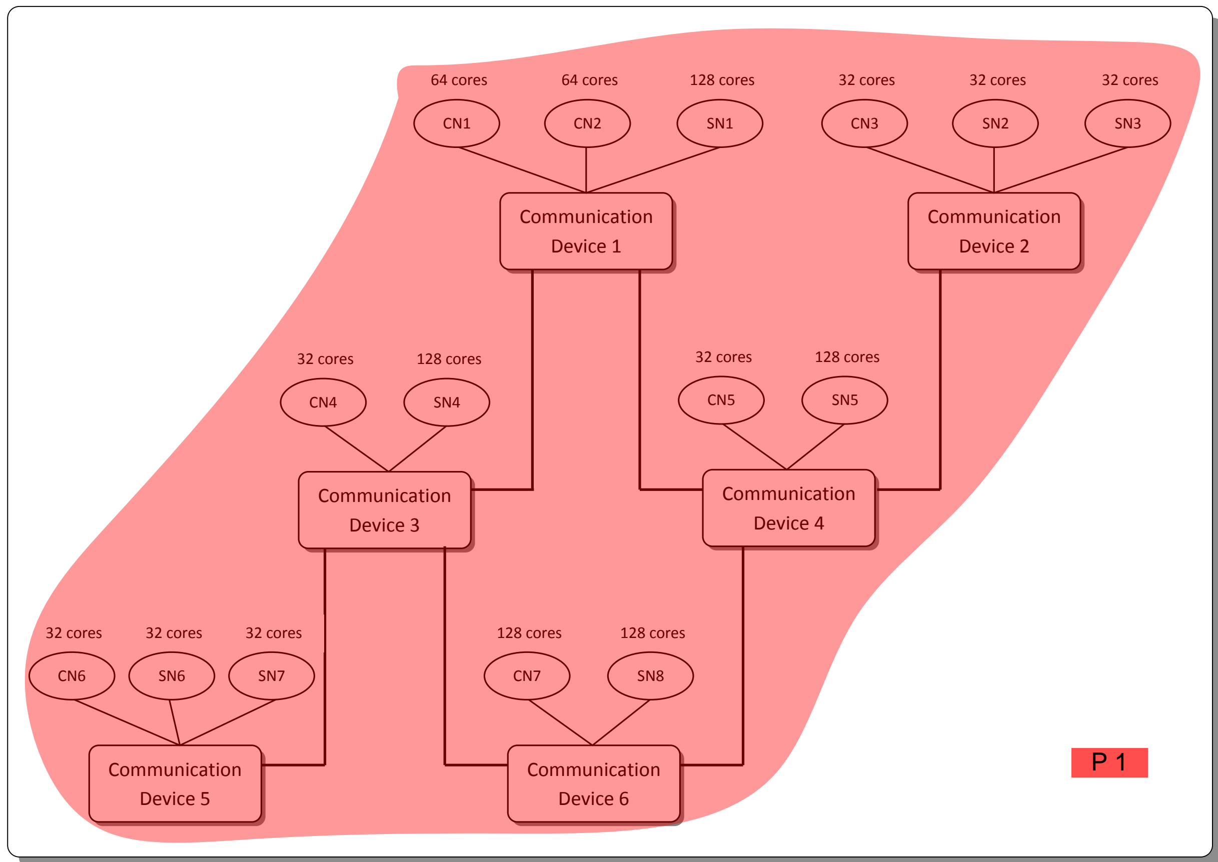


**Figura 50. Prueba 13**

## *Model_1 – 2 particiones – Simple balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN5 – SN5 – CD4
- ***Partición 2:*** CN3 – SN2 – SN3 – CD2 – CN4 – SN4 – CD3 – CN6 – SN6 – SN7 – CD5 – CN7 – SN8 – CD6

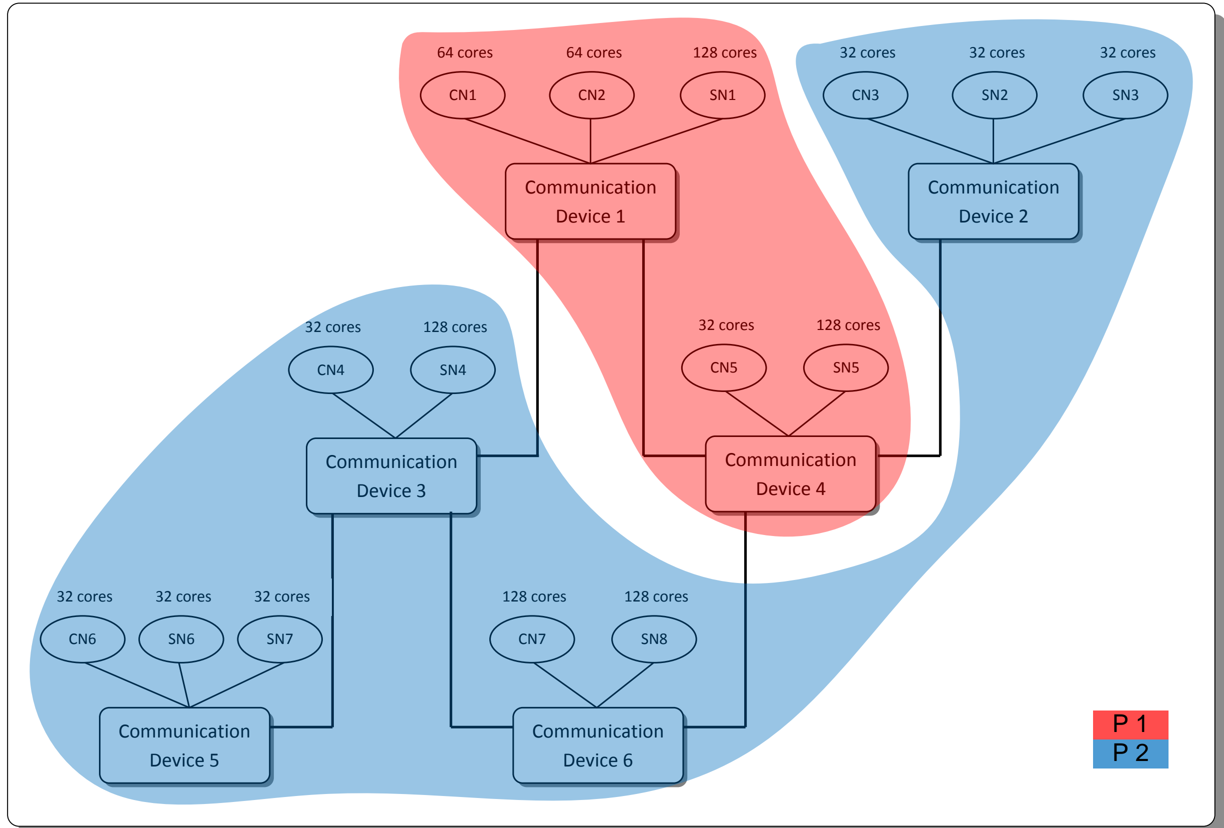


**Figura 51. Prueba 14**

## *Model_1 – 4 particiones – Simple balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1
- ***Partición 2:*** CN3 – SN2 – SN3 – CD2 – CN6 – SN6 – SN7 – CD5
- ***Partición 3:*** CN4 – SN4 – CD3 – CN7 – SN8 – CD6
- ***Partición 4:*** CN5 – SN5 – CD4

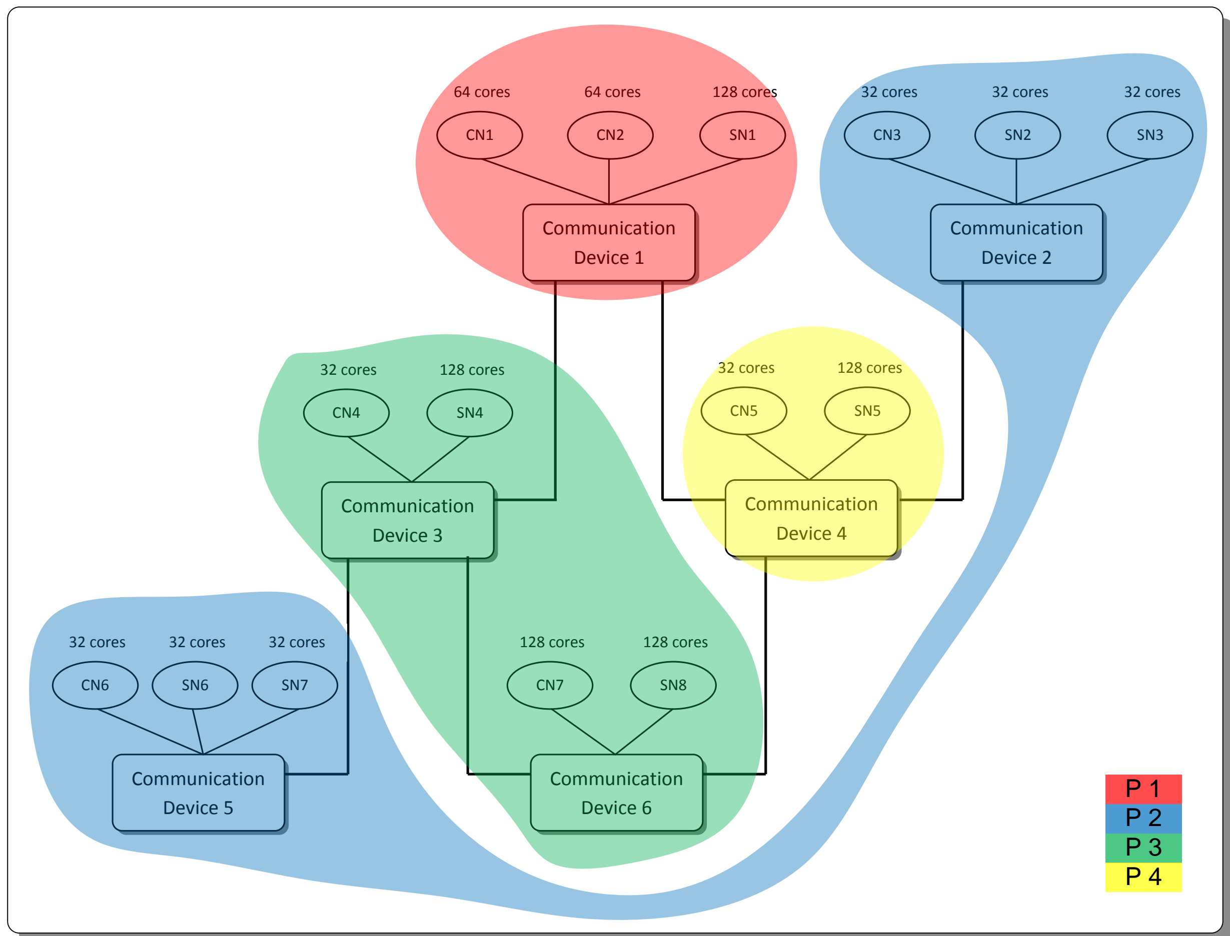


**Figura 52. Prueba 15**

## *Model_1 – 1 partición – Balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – SN3 – CD2 – CN4 – SN4 – CD3 – CN5 – SN5 – CD4 – CN6 – SN6 – SN7 – CD5 – CN7 – SN8 – CD6

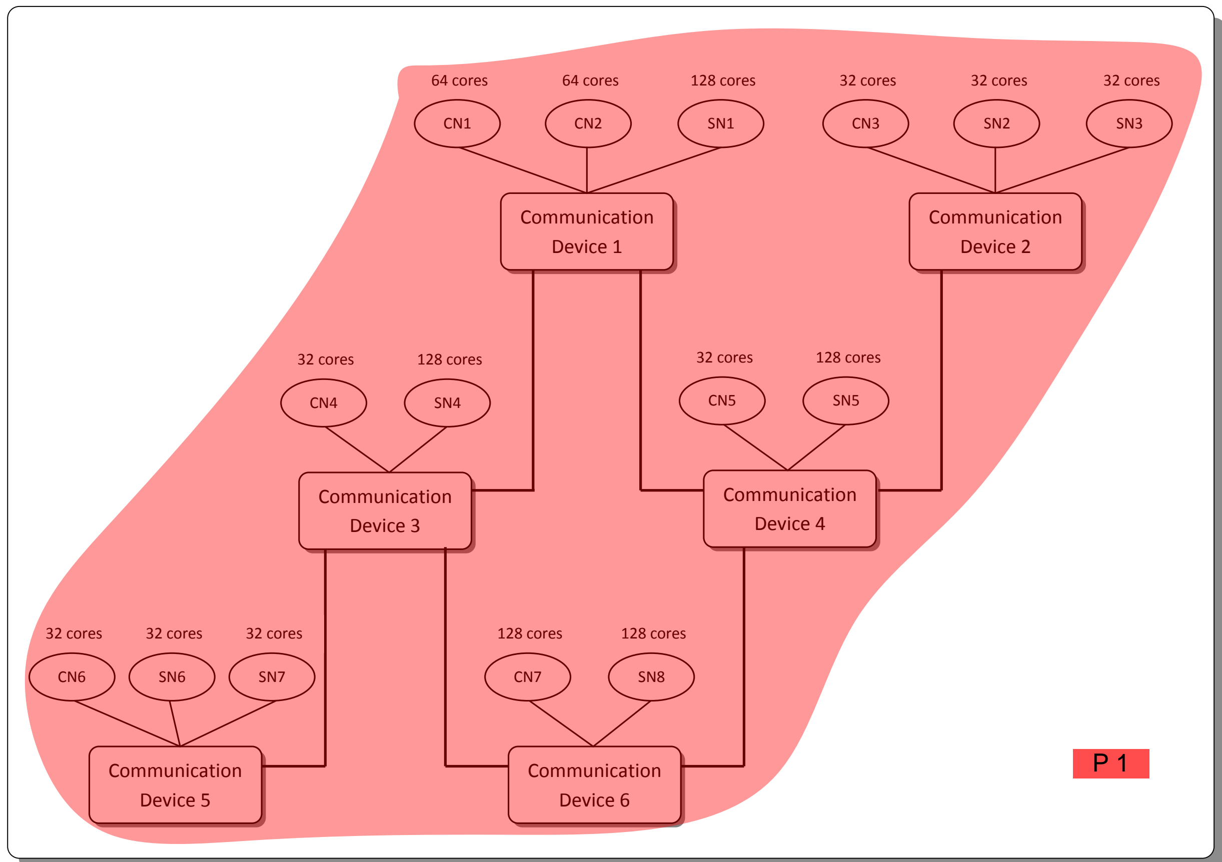


**Figura 53. Prueba 16**

## *Model_1 – 2 particiones – Balanced*

- ***Partición 1:*** CN1 – CN2 – SN1 – CD1 – CN3 – SN2 – SN3 – CD2 – CN4 – SN4 – CD3 – CN6 – SN6 – SN7 – CD5
- ***Partición 2:*** CN5 – SN5 – CD4 – CN7 – SN8 – CD6

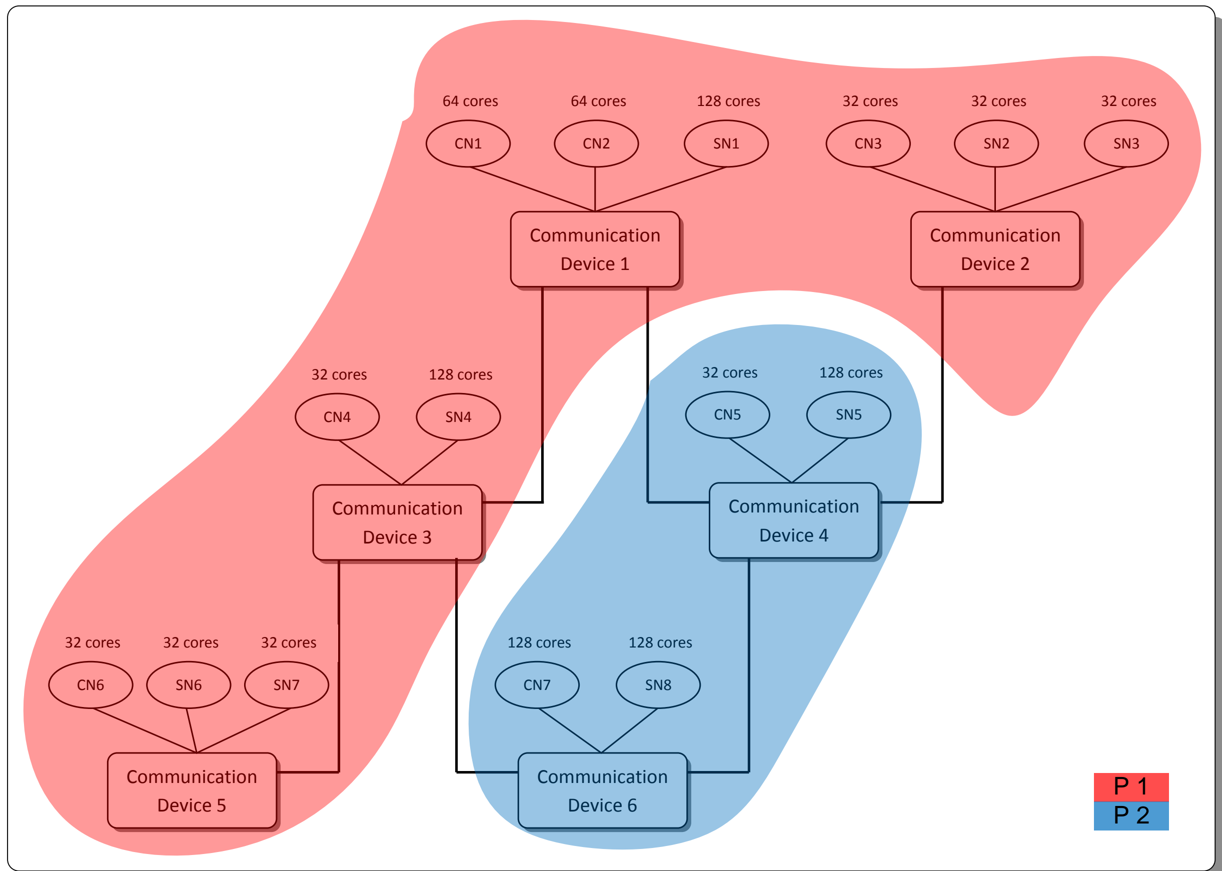


**Figura 54. Prueba 17**

## *Model_1 – 4 particiones – Balanced*

- ***Partición 1:*** CN4 – SN4 – CD3 – CN1 – CN6 – SN6 – SN7 – CD5
- ***Partición 2:*** CN3 – SN2 – SN3 – CD2 – CN5 – SN5 – CD4
- ***Partición 3:*** CN2 – SN1 – CD1
- ***Partición 4:*** CN7 – SN8 – CD6

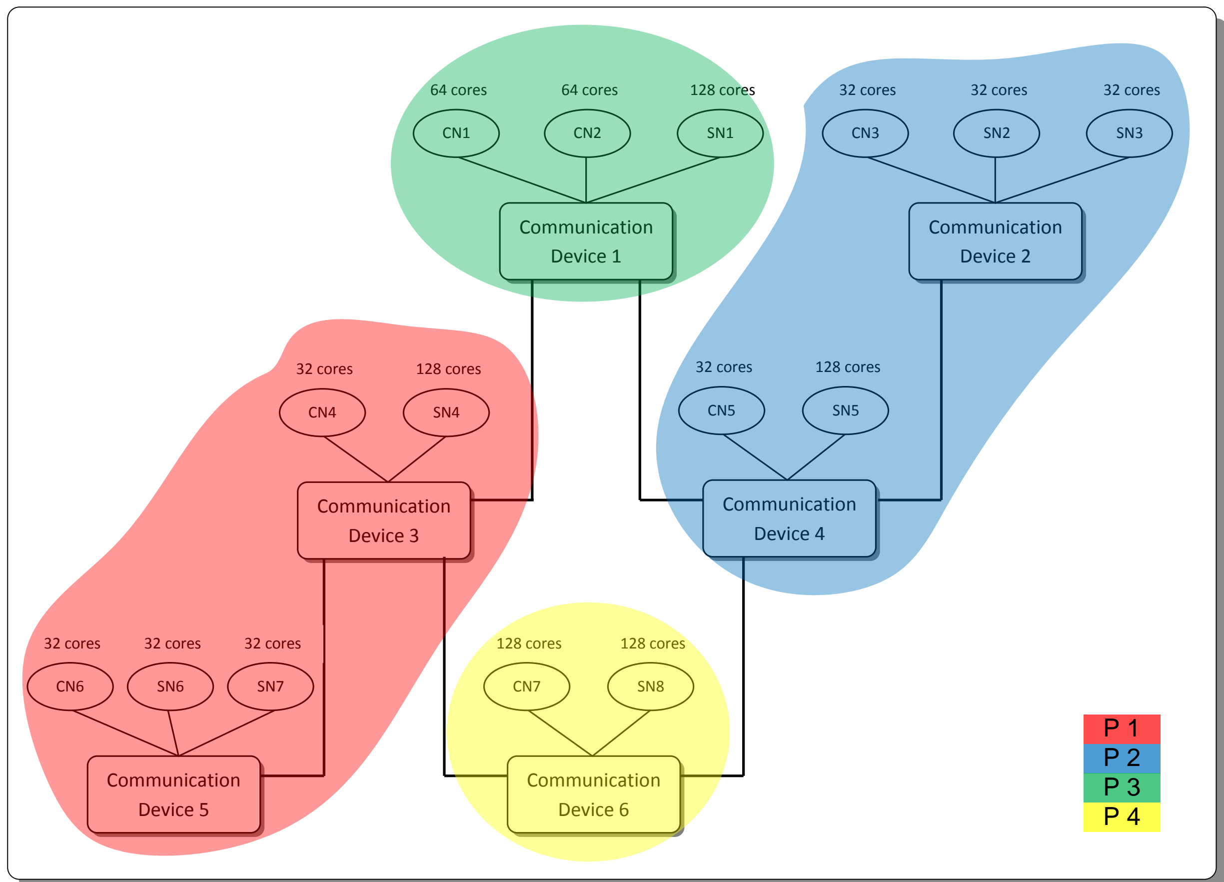


**Figura 55. Prueba 18**

## *Model_2 – 1 partición – Proximity*

- ***Partición 1:*** CN1 – SN1 – CD1 – CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN5 – SN4 – SN5 – CD3 – CN6 – CN7 – SN6 – SN7 – CD4 – CN8 – CN9 – SN8 – SN9 – CD5 – CN10 – CN11 – SN10 – CD6 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7 – CN14 – SN14 – CD8

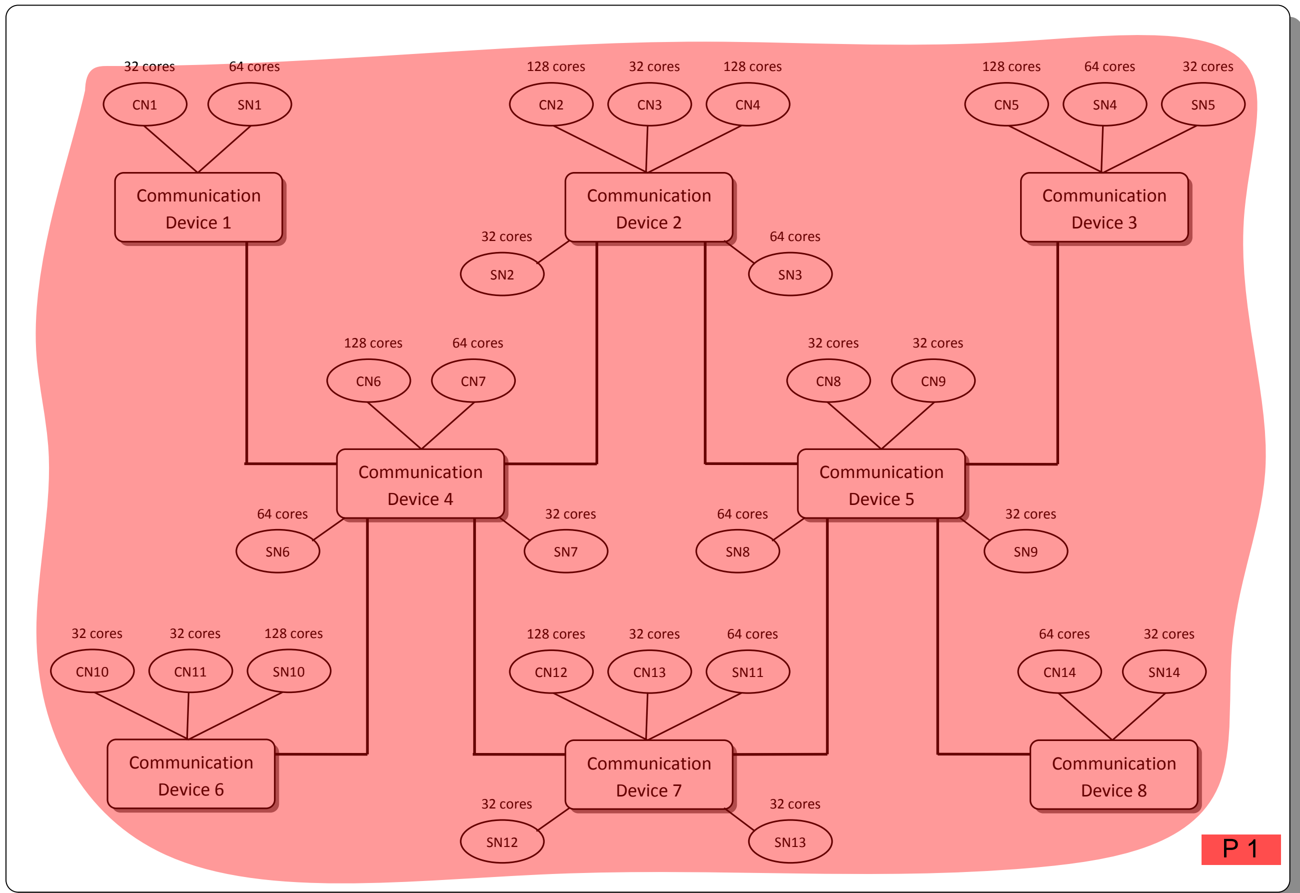


**Figura 56. Prueba 19**

## *Model_2 – 2 particiones – Proximity*

- ***Partición 1:*** CN1 – SN1 – CD1 – CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN6 – CN7 – SN6 – SN7 – CD4 – CN10 – CN11 – SN10 – CD6
- ***Partición 2:*** CN5 – SN4 – SN5 – CD3 – CN8 – CN9 – SN8 – SN9 – CD5 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7 – CN14 – SN14 – CD8

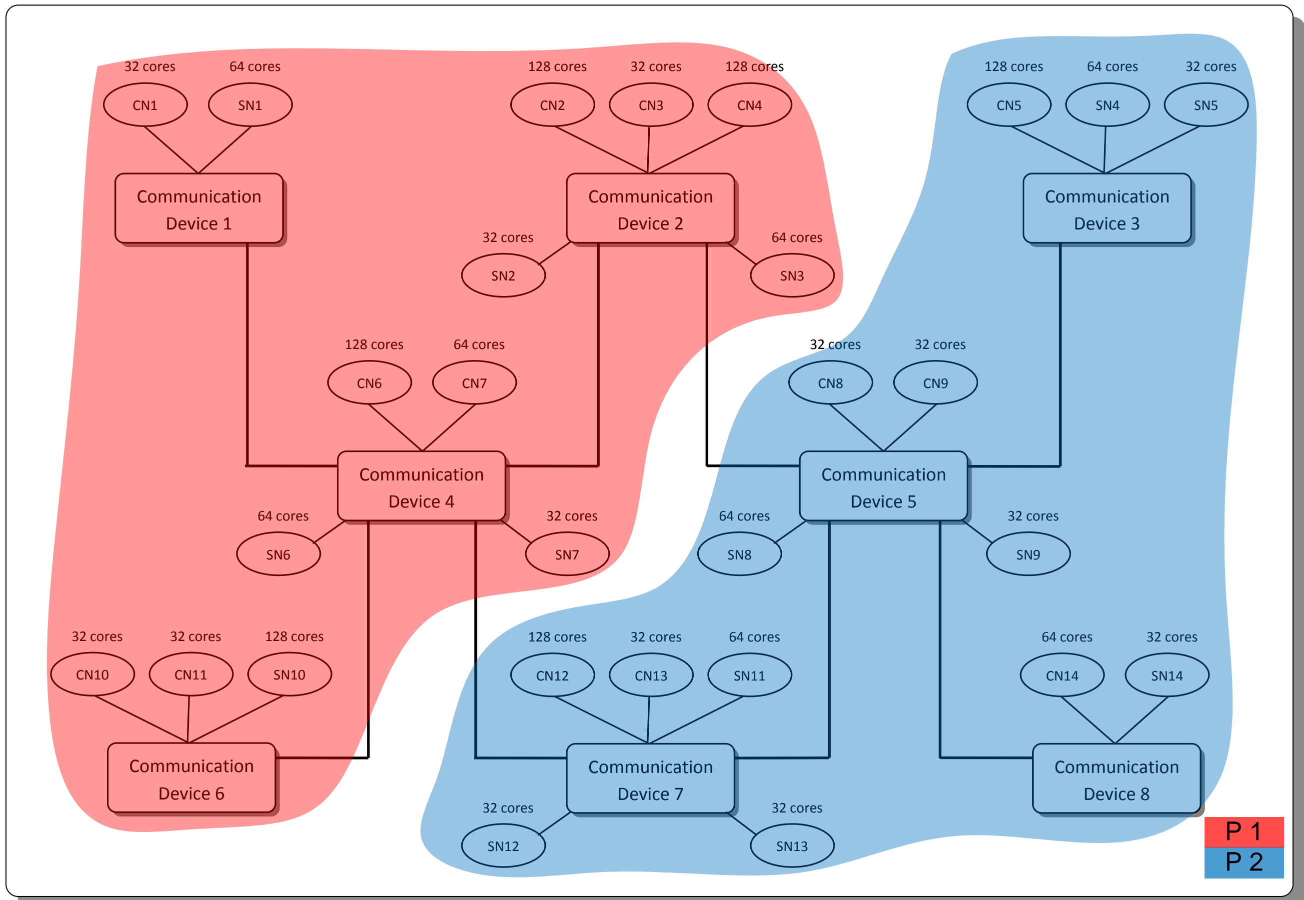


**Figura 57. Prueba 20**

## *Model_2 – 4 particiones – Proximity*

- ***Partición 1:*** CN1 – SN1 – CD1 – CN6 – CN7 – SN6 – SN7 – CD4 – CN10 – CN11 – SN10 – CD6
- ***Partición 2:*** CN5 – SN4 – SN5 – CD3 – CN8 – CN9 – SN8 – SN9 – CD5 – CN14 – SN14 – CD8
- ***Partición 3:*** CN2 – CN3 – CN4 – SN2 – SN3 – CD2
- ***Partición 4:*** CN12 – CN13 – SN11 – SN12 – SN13 – CD7

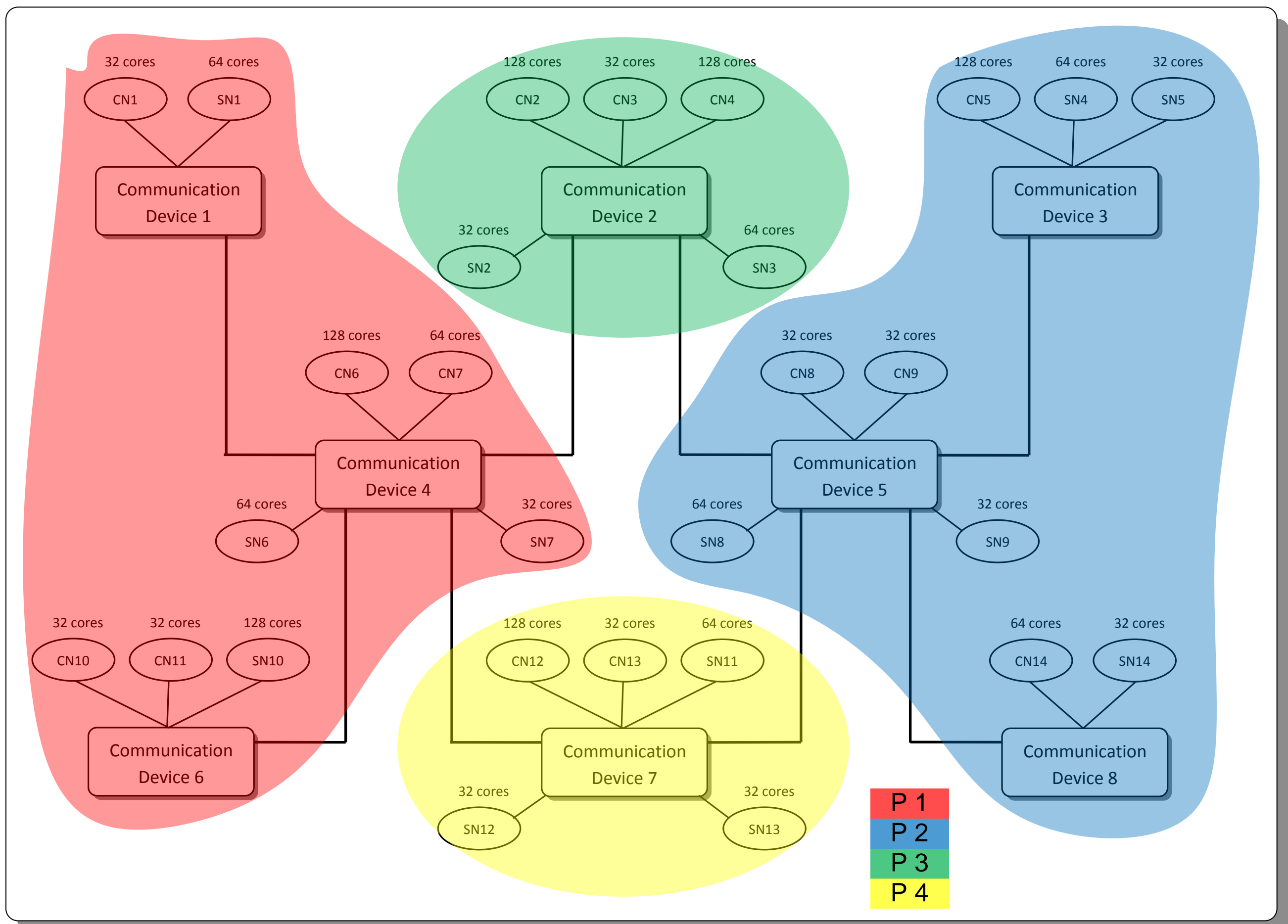


**Figura 58. Prueba 21**

## *Model_2 – 8 particiones – Proximity*

- ***Partición 1:*** CN6 – CN7 – SN6 – SN7 – CD4
- ***Partición 2:*** CN8 – CN9 – SN8 – SN9 – CD5
- ***Partición 3:*** CN2 – CN3 – CN4 – SN2 – SN3 – CD2
- ***Partición 4:*** CN12 – CN13 – SN11 – SN12 – SN13 – CD7
- ***Partición 5:*** CN1 – SN1 – CD1
- ***Partición 6:*** CN5 – SN4 – SN5 – CD3
- ***Partición 7:*** CN10 – CN11 – SN10 – CD6
- ***Partición 8:*** CN14 – SN14 – CD8

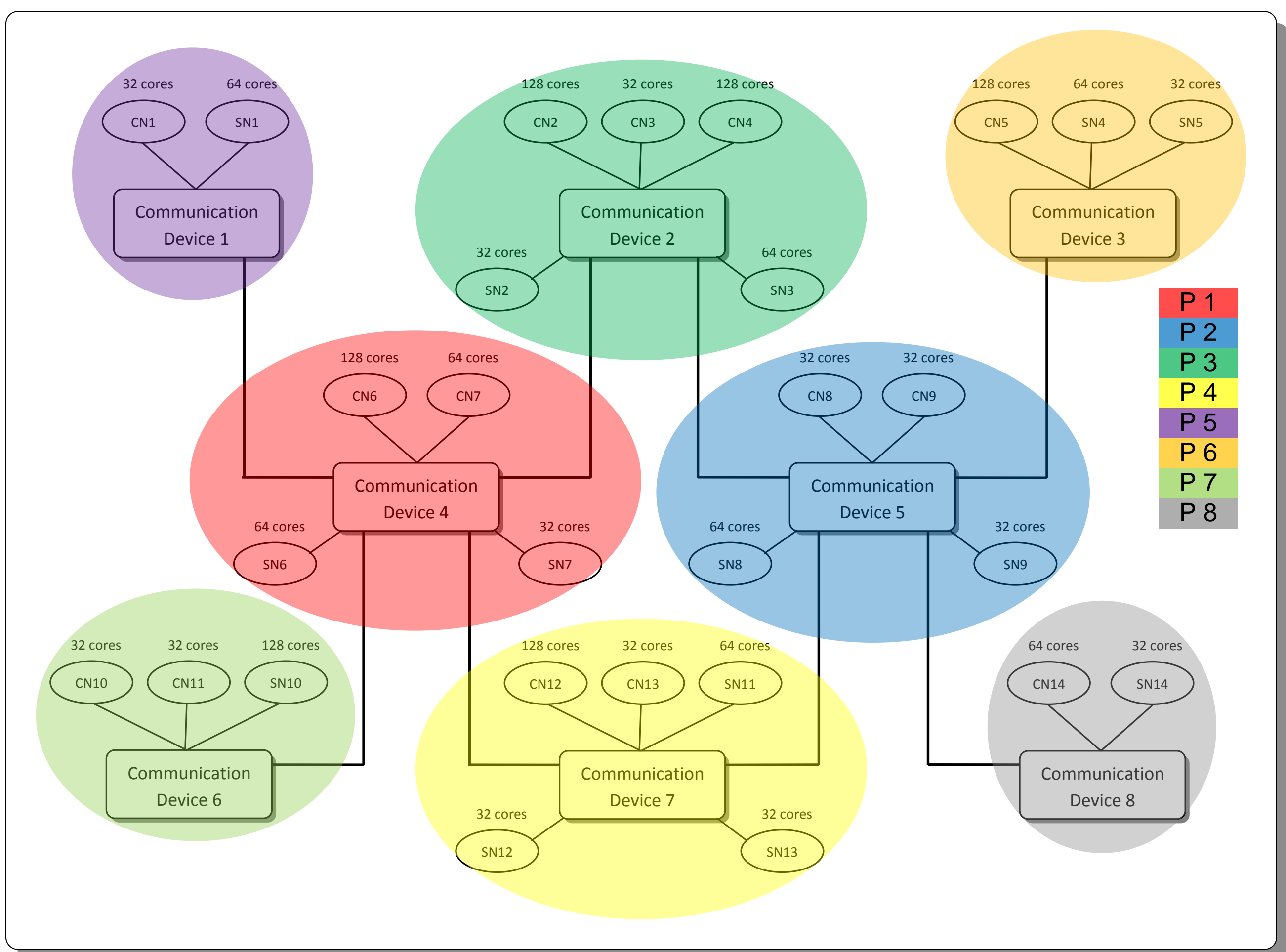


**Figura 59. Prueba 22**

### *Model_2 – 1 partición – Simple balanced*

- ***Partición 1:*** CN1 – SN1 – CD1 – CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN5 – SN4 – SN5 – CD3 – CN6 – CN7 – SN6 – SN7 – CD4 – CN8 – CN9 – SN8 – SN9 – CD5 – CN10 – CN11 – SN10 – CD6 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7 – CN14 – SN14 – CD8

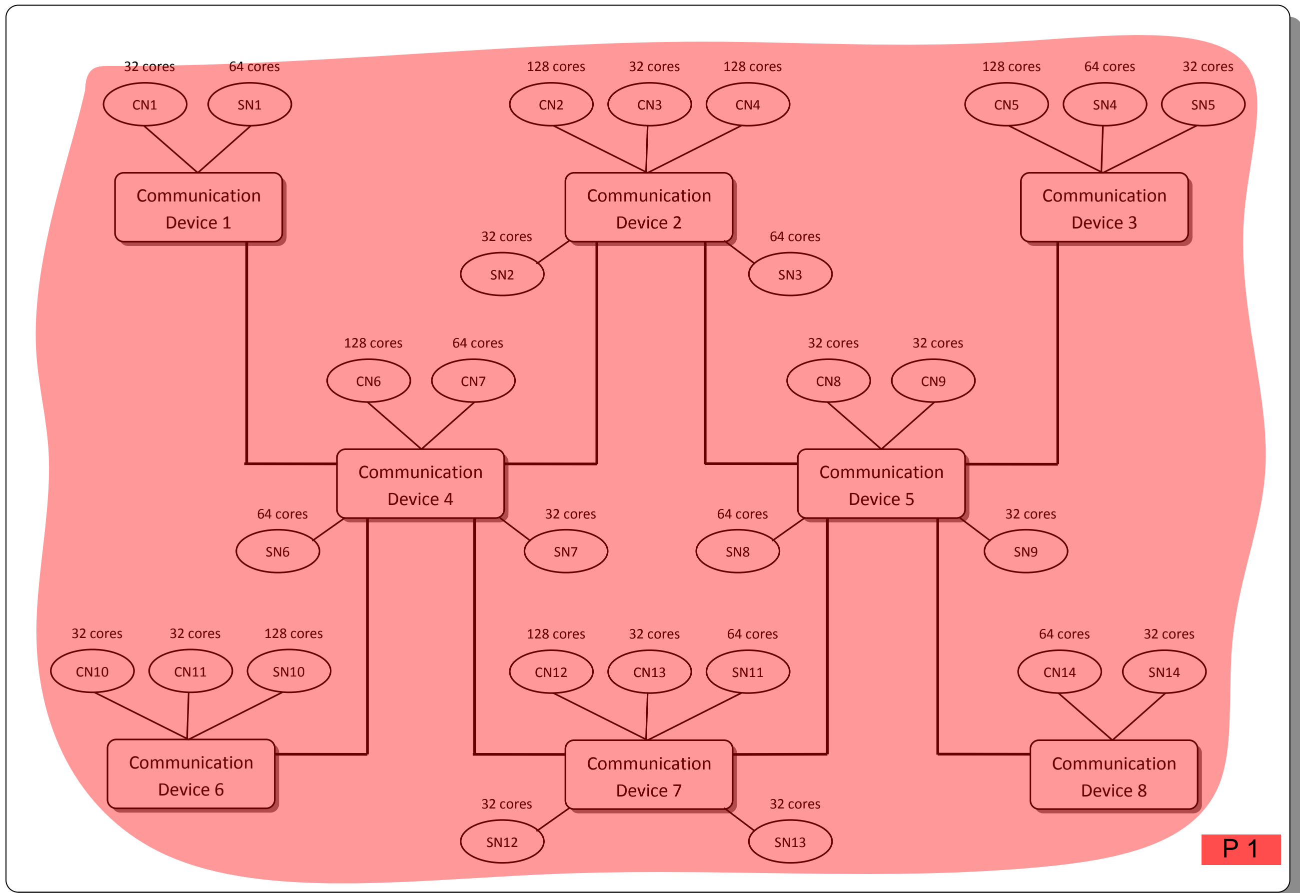


**Figura 60. Prueba 23**

## *Model_2 – 2 particiones – Simple balanced*

- ***Partición 1:*** CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN6 – CN7 – SN6 – SN7 – CD4 – CN10 – CN11 – SN10 – CD6 – CN14 – SN14 – CD8
- ***Partición 2:*** CN1 – SN1 – CD1 – CN5 – SN4 – SN5 – CD3 – CN8 – CN9 – SN8 – SN9 – CD5 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7

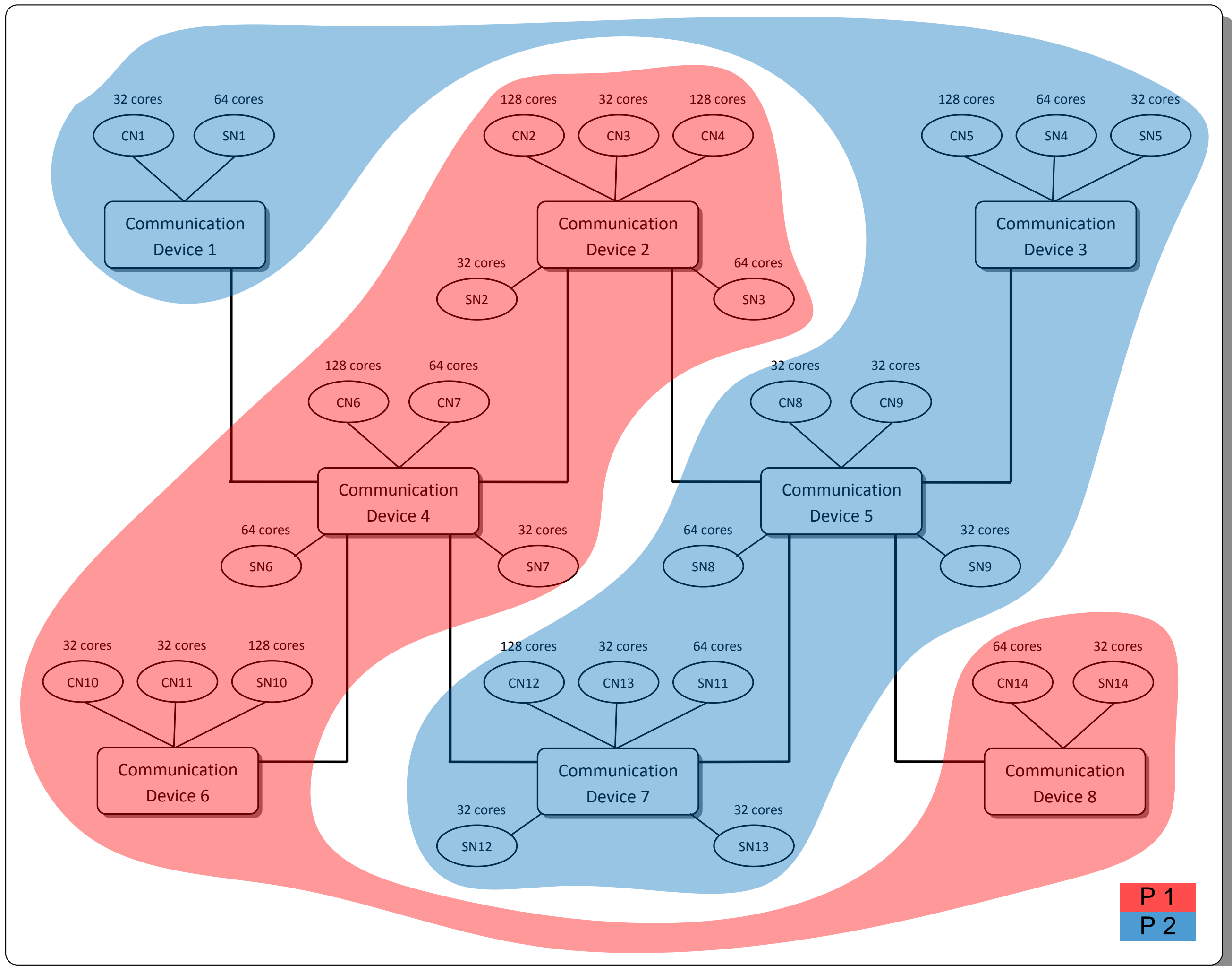


**Figura 61. Prueba 24**

## *Model_2 – 4 particiones – Simple balanced*

- ***Partición 1:*** CN1 – SN1 – CD1 – CN8 – CN9 – SN8 – SN9 – CD5
- ***Partición 2:*** CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN14 – SN14 – CD8
- ***Partición 3:*** CN5 – SN4 – SN5 – CD3 – CN10 – CN11 – SN10 – CD6
- ***Partición 4:*** CN6 – CN7 – SN6 – SN7 – CD4 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7

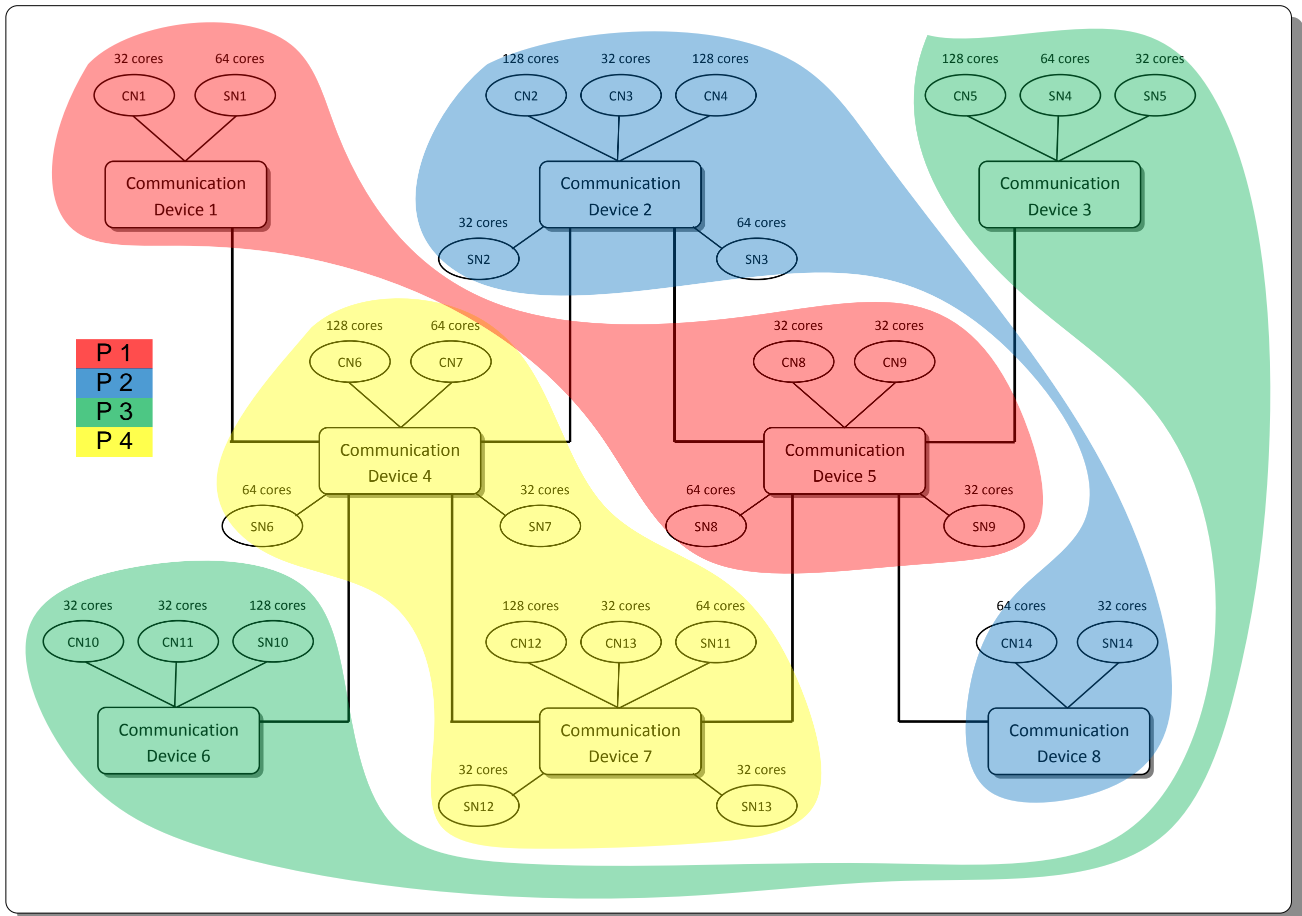


**Figura 62. Prueba 25**

## *Model_2 – 8 particiones – Simple balanced*

- ***Partición 1:*** CN1 – SN1 – CD1
- ***Partición 2:*** CN2 – CN3 – CN4 – SN2 – SN3 – CD2
- ***Partición 3:*** CN5 – SN4 – SN5 – CD3
- ***Partición 4:*** CN6 – CN7 – SN6 – SN7 – CD4
- ***Partición 5:*** CN8 – CN9 – SN8 – SN9 – CD5
- ***Partición 6:*** CN10 – CN11 – SN10 – CD6
- ***Partición 7:*** CN12 – CN13 – SN11 – SN12 – SN13 – CD7
- ***Partición 8:*** CN14 – SN14 – CD8

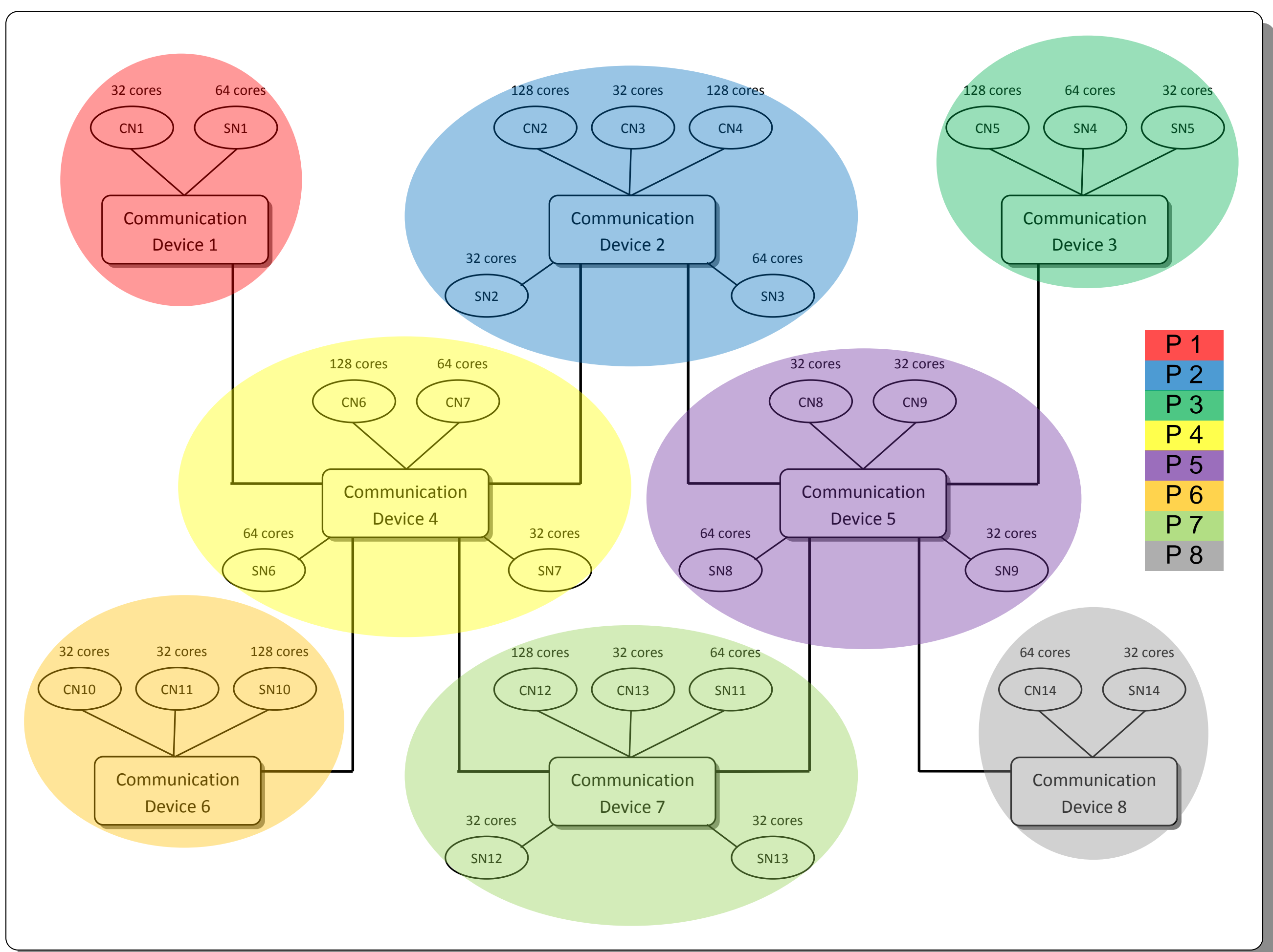


**Figura 63. Prueba 26**

## *Model_2 – 1 partición – Balanced*

- ***Partición 1:*** CN1 – SN1 – CD1 – CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN5 – SN4 – SN5 – CD3 – CN6 – CN7 – SN6 – SN7 – CD4 – CN8 – CN9 – SN8 – SN9 – CD5 – CN10 – CN11 – SN10 – CD6 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7 – CN14 – SN14 – CD8

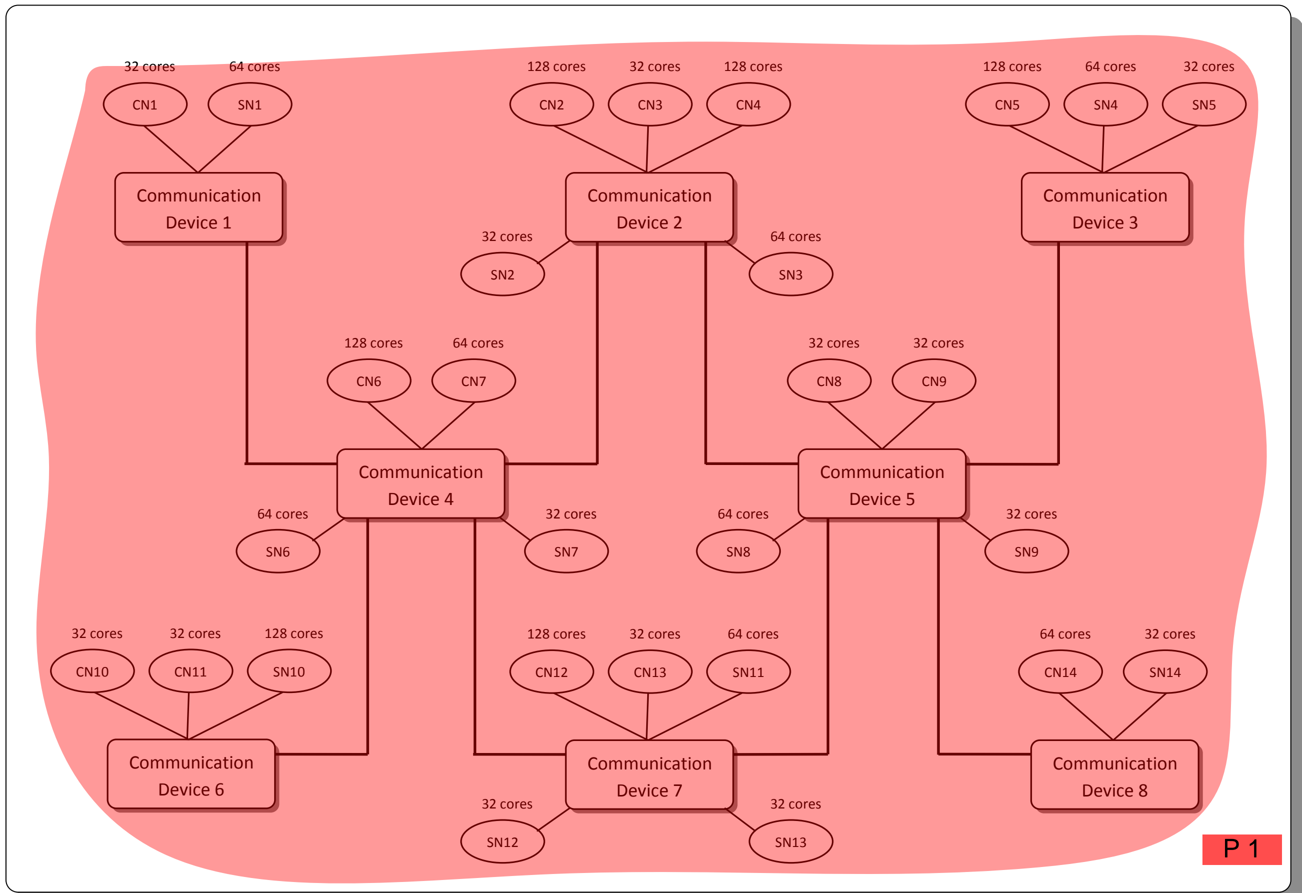


**Figura 64. Prueba 27**

## *Model_2 – 2 particiones – Balanced*

- ***Partición 1:*** CN5 – SN4 – SN5 – CD3 – CN8 – CN9 – SN8 – SN9 – CD5 – CN10 – CN11 – SN10 – CD6
- ***Partición 2:*** CN1 – SN1 – CD1 – CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN6 – CN7 – SN6 – SN7 – CD4 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7 – CN14 – SN14 – CD8

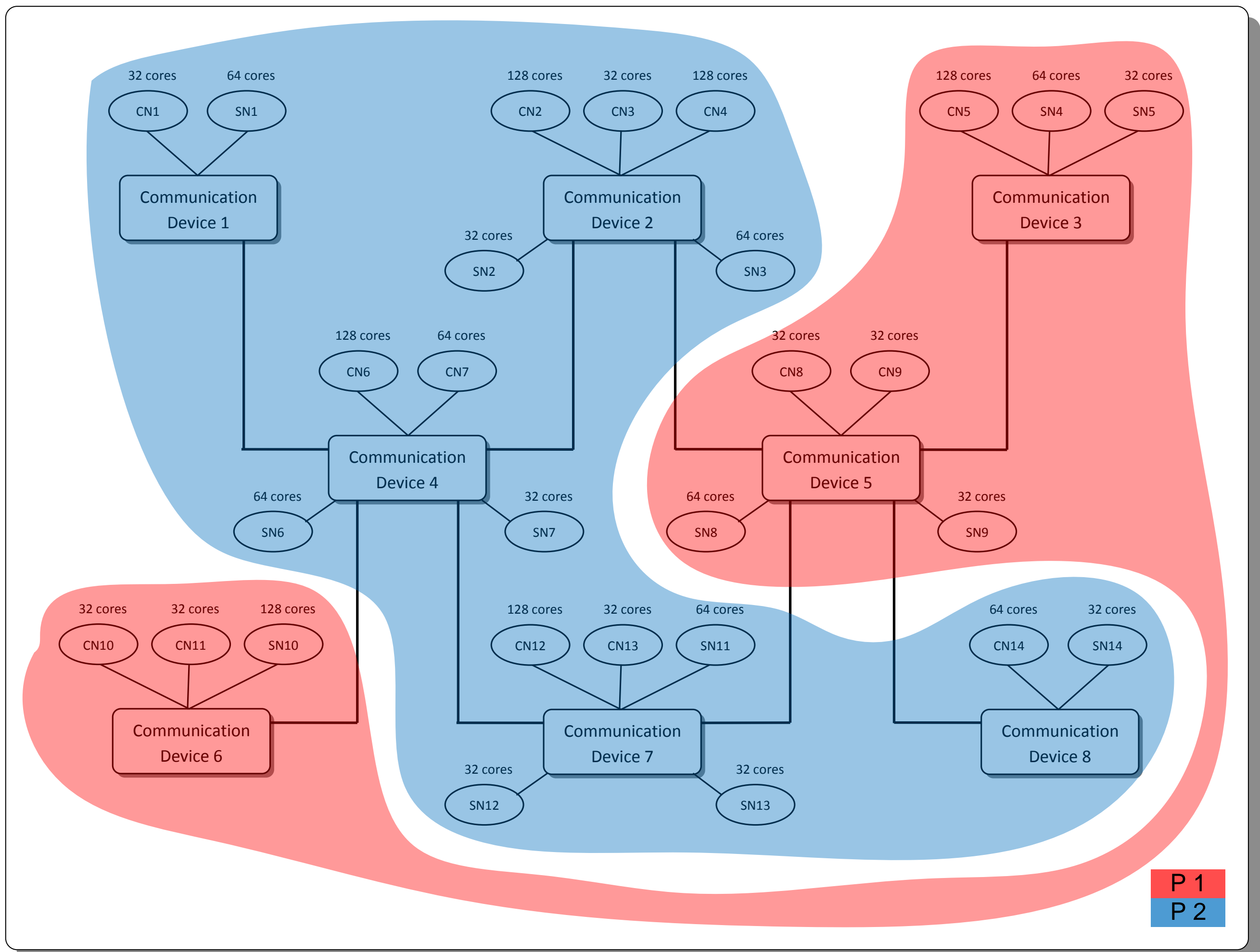


**Figura 65. Prueba 28**

## *Model_2 – 4 particiones – Balanced*

- ***Partición 1:*** CN5 – SN4 – SN5 – CD3 – CN8 – CN9 – SN8 – SN9 – CD5
- ***Partición 2:*** CN6 – CN7 – SN6 – SN7 – CD4 – CN10 – CN11 – SN10 – CD6
- ***Partición 3:*** CN2 – CN3 – CN4 – SN2 – SN3 – CD2 – CN14 – SN14 – CD8
- ***Partición 4:*** CN1 – SN1 – CD1 – CN12 – CN13 – SN11 – SN12 – SN13 – CD7

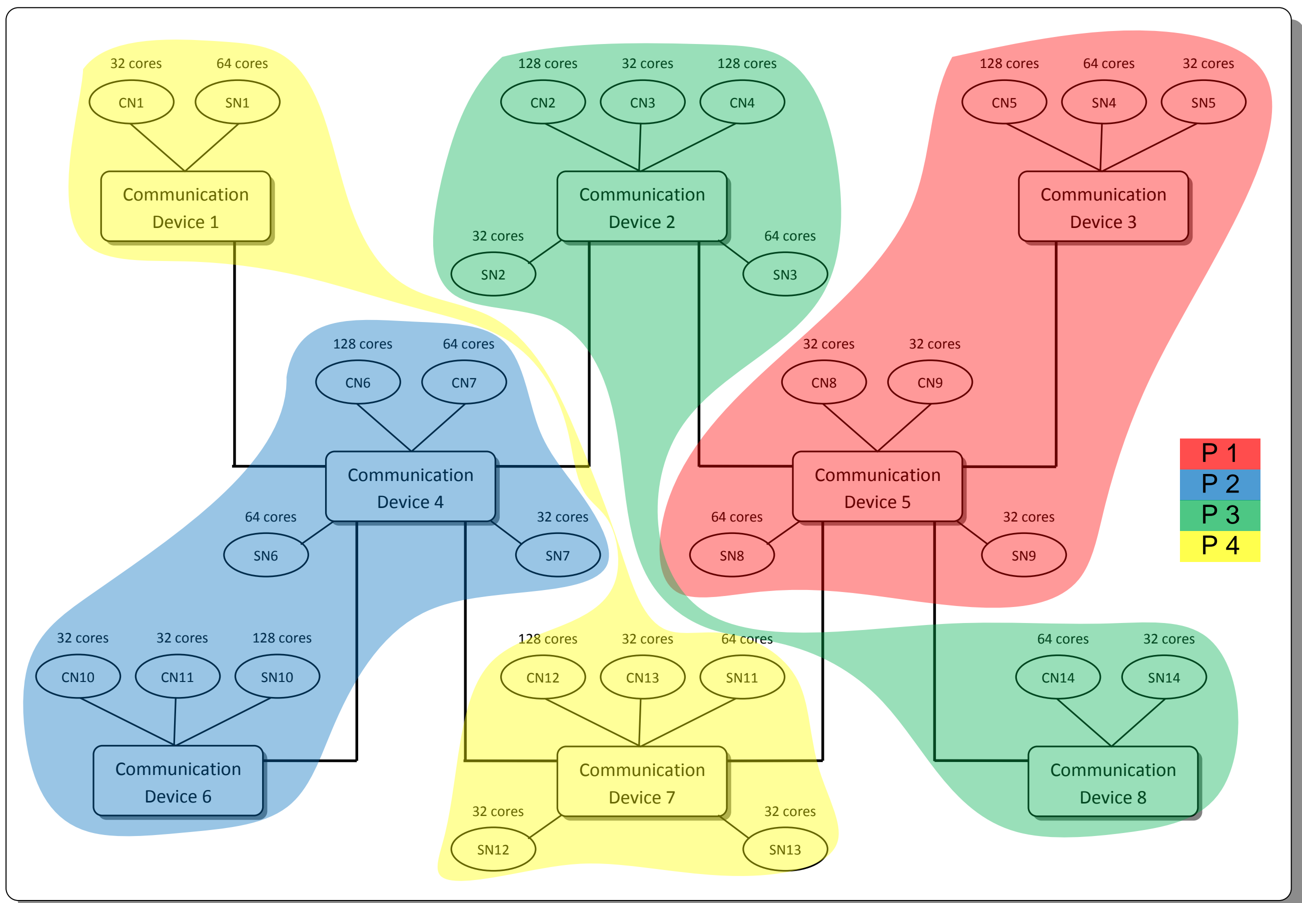


**Figura 66. Prueba 29**

## *Model_2 – 8 particiones – Balanced*

- ***Partición 1:*** CN1 – SN1 – CD1
- ***Partición 2:*** CN2 – CN3 – CN4 – SN2 – SN3 – CD2
- ***Partición 3:*** CN5 – SN4 – SN5 – CD3
- ***Partición 4:*** CN6 – CN7 – SN6 – SN7 – CD4
- ***Partición 5:*** CN8 – CN9 – SN8 – SN9 – CD5
- ***Partición 6:*** CN10 – CN11 – SN10 – CD6
- ***Partición 7:*** CN12 – CN13 – SN11 – SN12 – SN13 – CD7
- ***Partición 8:*** CN14 – SN14 – CD8

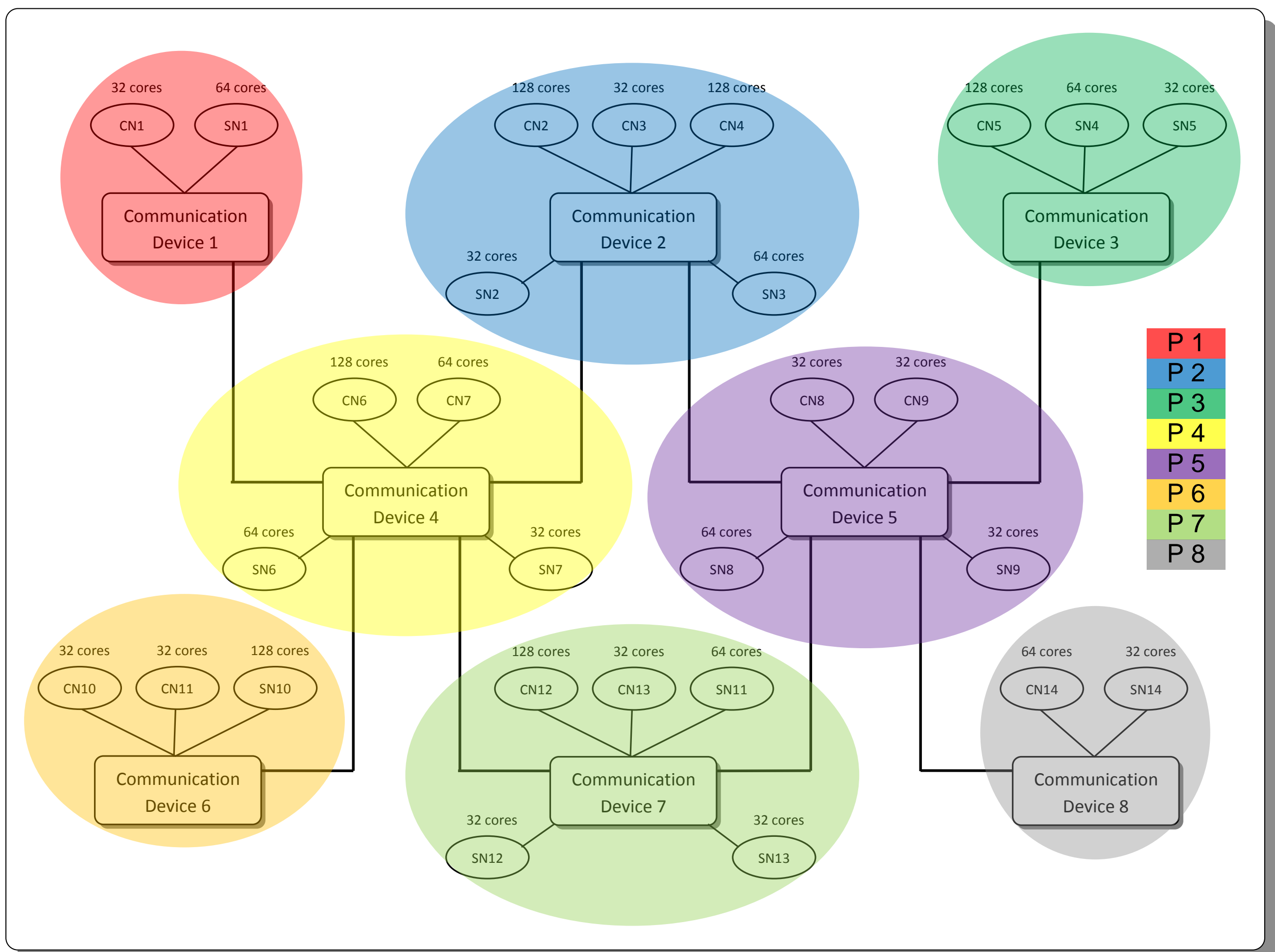


**Figura 67. Prueba 30**

## 6.4. Comparativas

En esta sección se van a mostrar diferentes comparativas junto con sus gráficas correspondientes. El objetivo es detallar los datos de rendimiento ofrecidos en la simulación de las pruebas realizadas. En primer lugar, se procede a mostrar los datos de rendimiento de cada algoritmo sobre cada modelo, dependiendo del número de particiones. En segundo lugar, se especifica el rendimiento de los algoritmos para cada modelo en un número de particiones fijo. Para terminar la investigación se centra en la escalabilidad del problema, como afecta al rendimiento el que haya modelos pequeños o grandes. En este apartado se muestran gráficas descriptivas de cómo mejora el rendimiento de la simulación, cuanto más se divide el problema en diferentes números de particiones.

### *Rendimiento Model_presentation*

La figura 68 muestra los tiempos de ejecución necesarios para la simulación de la ejecución de los tres modelos para las 1, 2 y 4 particiones. La línea del algoritmo *proximity* se ve solapada por la línea de *balanced* ya que tienen los mismos tiempos.

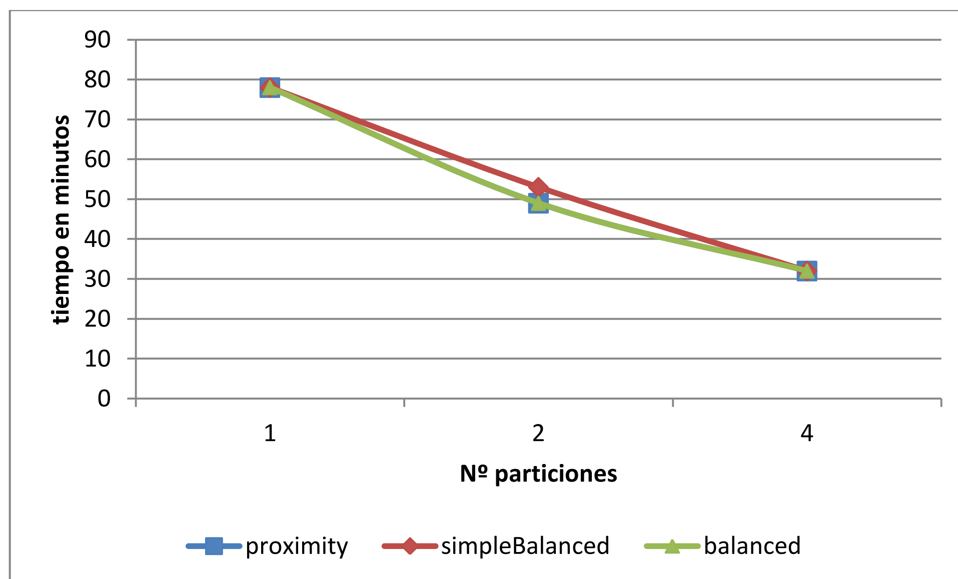


**Figura 68. Gráfica rendimiento *model_presentation* con *los 3 algoritmos***

La figura 69 muestra el speedup de los tres algoritmos para el modelo *model_presentation*. La línea del algoritmo *proximity* se ve solapada por la línea de *balanced* ya que tienen los mismos tiempos.

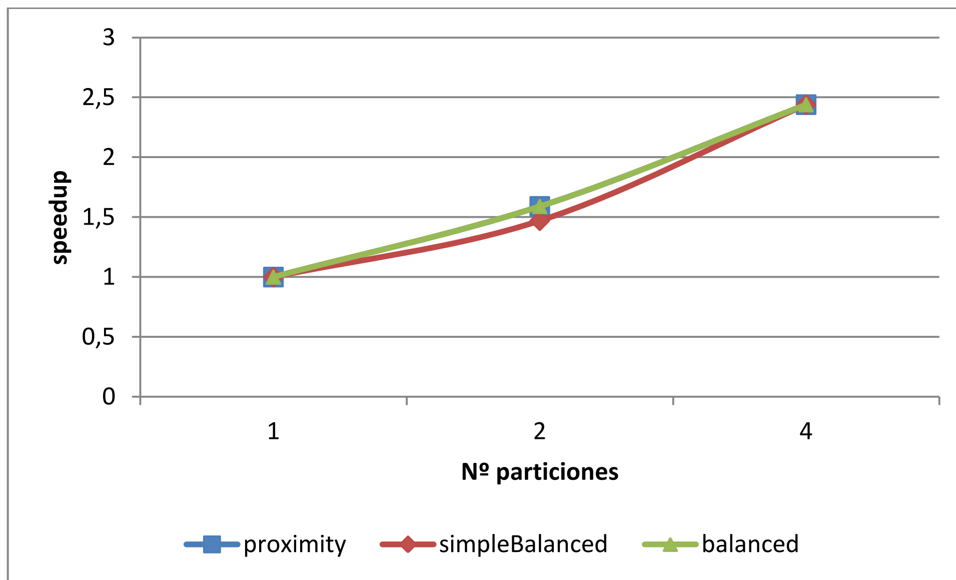


**Figura 69. Gráfica speedup *model_presentation* con *los 3 algoritmos***

## *Rendimiento Model_1*

La figura 70 muestra los tiempos de ejecución necesarios para la simulación de la ejecución de los tres modelos para las 1, 2 y 4 particiones.

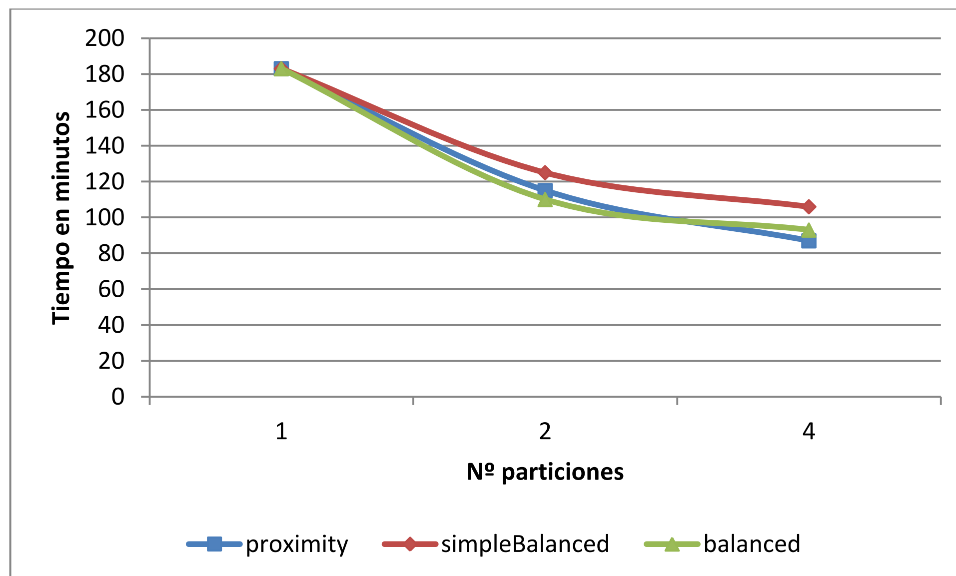


**Figura 70. Gráfica rendimiento *model_1* con *los 3 algoritmos***

La figura 71 muestra el speedup de los tres algoritmos para el modelo *model_1*.

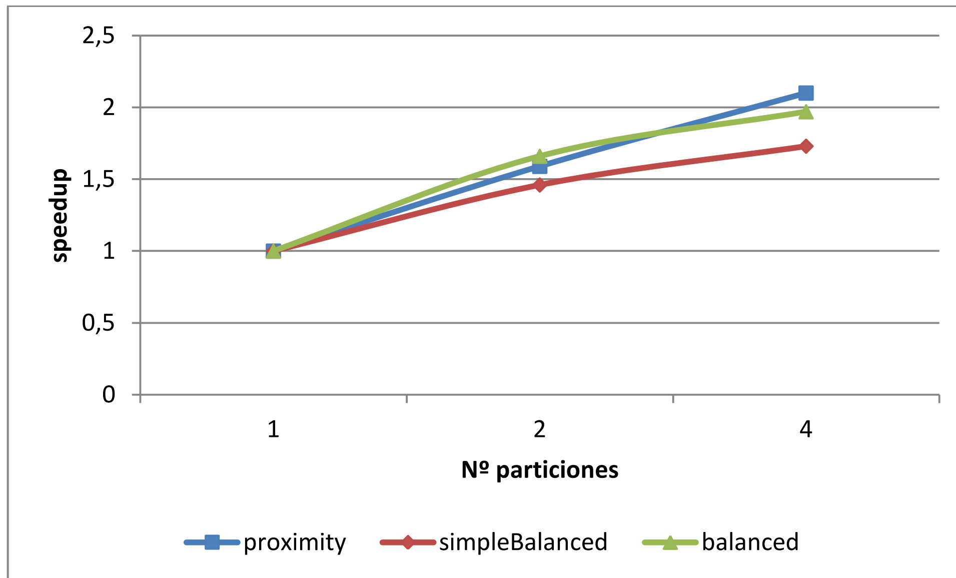


**Figura 71. Gráfica speedup *model_1* con *los 3 algoritmos***

## *Rendimiento Model_2*

La figura 72 muestra los tiempos de ejecución necesarios para la simulación de la ejecución de los tres modelos para las 1, 2, 4 y 8 particiones.

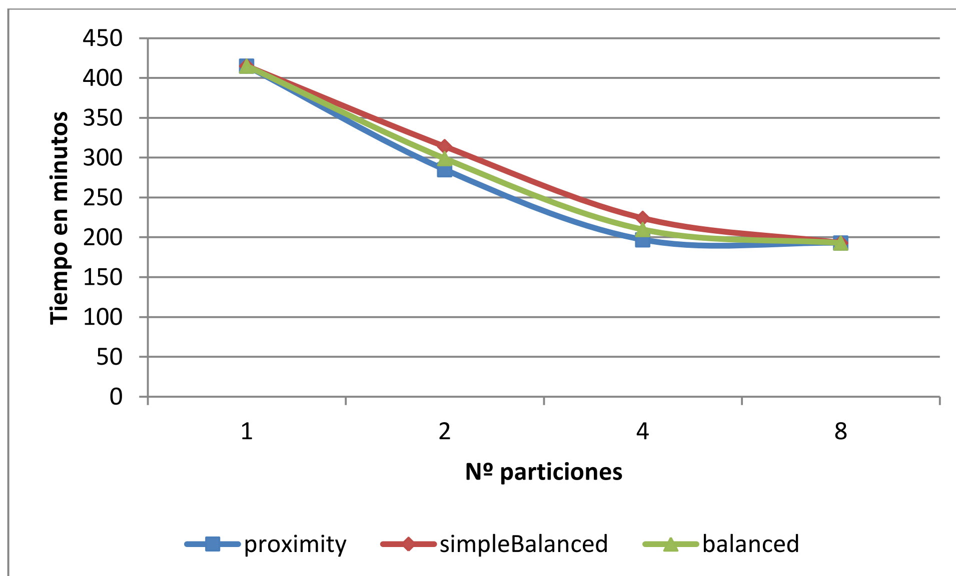


**Figura 72. Gráfica rendimiento *model_2* con *los 3 algoritmos***

La figura 73 muestra el speedup de los tres algoritmos para el modelo *model_2*.

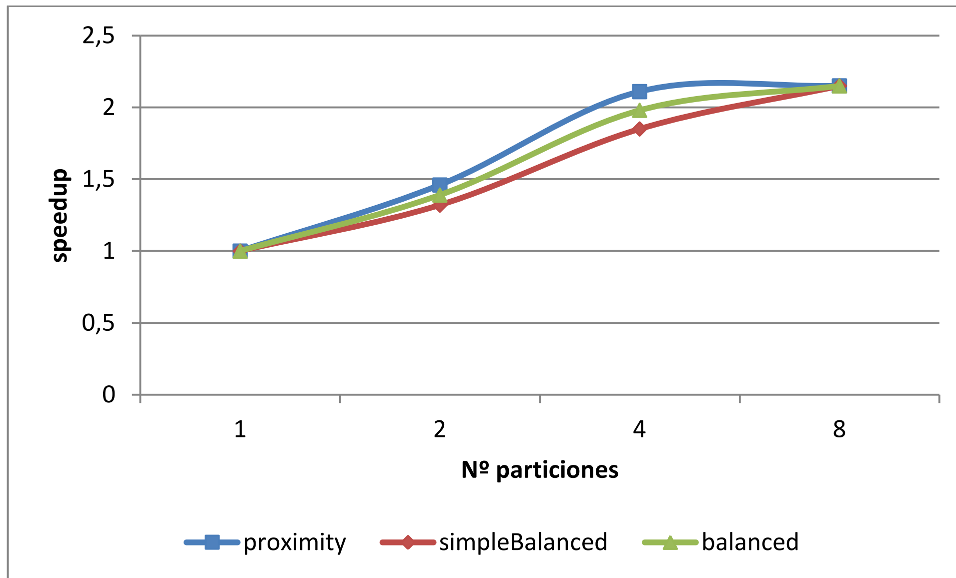


**Figura 73. Gráfica speedup *model_2* con *los 3 algoritmos***

# 7. Conclusiones

Esta sección se a dividir en dos partes, una primera en la que se sintetizarán los resultados conseguidos en las pruebas y una segunda en la que se valorará la elaboración del proyecto y posteriores mejoras a realizar en un futuro.

## *Resultados*

De las pruebas y comparativas realizadas por el sistema se llega a las siguientes conclusiones:

- En términos generales y en vista de los resultados el algoritmo ***proximity*** es el que mejor rendimiento ofrece a la simulación de los tres dados.
- El algoritmo ***balanced*** ofrece un rendimiento cercano a ***proximity*** y superior a ***simple balanced***.
- El algoritmo ***simple balanced*** ofrece el peor rendimiento de los tres algoritmos dados.
- El ***rendimiento*** ofrecido a la simulación por los tres algoritmos ***aumenta*** con el ***número de particiones***. Si bien, llegado un número máximo de particiones este rendimiento pasa a ser prácticamente insignificante.
- El ***speedup*** ofrecido a la simulación por los tres algoritmos ***desciende*** con el ***número de particiones***. Esto es debido al ***overhead*** producido por las comunicaciones realizadas en el modelo.
- El algoritmo ***proximity*** llega a ofrecer un rendimiento un ***7 %*** más ***rápido*** que el algoritmo ***balanced*** y un ***22 %*** más ***rápido*** que el algoritmo ***simple balanced***.
- El algoritmo ***balanced*** llega a ofrecer un rendimiento un ***7 %*** más ***lento*** que el algoritmo ***proximity*** y un ***14 %*** más ***rápido*** que el algoritmo ***simple balanced***.
- El algoritmo ***simple balanced*** llega a ofrecer un rendimiento un ***22 %*** más ***lento*** que el algoritmo ***proximity*** y un ***14 %*** más ***lento*** que el algoritmo ***balanced***.

## *Valoración del proyecto*

La percepción del sistema desde el comienzo al final ha sufrido diversos cambios lógicos dentro de un proyecto, llegando a las siguientes conclusiones finales.

Los principios del proyecto ha sido la parte más complicada del mismo. Si bien, la sección inicial del análisis del problema ha sido bastante intuitiva y sencilla con unos requisitos

bien definidos, la parte del diseño de algoritmos ha supuesto más tiempo del previsto debido a la dificultad del problema.

Una vez se ha tenido la especificación de requisitos y los algoritmos desarrollados, ha sido relativamente simple el realizar el diseño de la plataforma y regular un entorno y plan de pruebas para probar el sistema. En estos apartados de se han seguido los plazos correctamente.

Los resultados obtenidos se estiman como satisfactorios y acorde con la teoría estudiada durante el proyecto. El rendimiento del sistema mejora con el número de particiones, así como cabe destacar que ésta mejora tiene un límite máximo acotado, que hace que en ciertos escenarios sea mayor el esfuerzo hecho que la mejora conseguida.

El estudio de la tecnología computacional de alto rendimiento en la que se basa el proyecto ha permitido conocer mejor las ventajas e inconvenientes que supone esta tecnología. Sin duda, se puede decir que en un futuro será una parte importante del campo de la informática para la ejecución de trabajos de tamaño masivo.

Como revisión de los objetivos marcados al inicio del proyecto, se ha de comentar que se han conseguido todos en gran parte. El objetivo principal de definir una serie de algoritmos que realicen un particionado de un modelo de simulación se ha cumplido en su totalidad.

El sistema consigue obtener unos resultados fiables, tanto para problemas de tamaño pequeño como grande, cumpliendo la propiedad de escalabilidad, permitiendo ejecutar trabajos grandes en tiempos prudenciales.

Se han obtenido resultados positivos también en cuanto al objetivo de mejora de rendimiento en base a una partición de un modelo dado. Según los resultados obtenidos queda claro que la mejora existe.

Como líneas futuras dentro del proyecto se puede destacar el desarrollo de nuevos algoritmos que intenten mejorar el rendimiento de los actuales. La plataforma está preparada para incorporar cualquier algoritmo, con el menor impacto posible en coste y tiempo, que permita explorar nuevas vías de ejecución en paralelo de modelos de simulación. Se cumple así la propiedad de flexibilidad ante nuevos cambios.

Resultaría interesante para la plataforma mejorar la funcionalidad con una aplicación que generase los modelos de entrada a partir de unos datos dados por el usuario. Esto permitiría agregar de una forma más sencilla nuevos modelos y variar los existentes con el objetivo de realizar nuevos estudios que ilustren nuevos datos sobre la computación de alto rendimiento con el objetivo de obtener mejores resultados.

En un futuro sería deseable mejorar la interfaz gráfica de la aplicación con el fin de adaptarse a las nuevas tecnologías de presentación y optimizar la interacción usuario sistema.

En resumen, la valoración final del proyecto es ***positiva***. Se han adquirido nuevos conocimientos sobre la temática del problema, además de aplicar todos los conocimientos obtenidos en mi experiencia en el campo de la informática. La bibliografía analizada durante el tiempo de duración del proyecto ha sido de mucha utilidad, permitiendo explorar nuevos caminos a la hora de desarrollar los algoritmos de la plataforma. En términos generales, destacar que la complicación del proyecto ha venido dada en mayor medida por la manera de pensar en el momento de implementar los algoritmos.

# Bibliografía


- ***Introduction to Parallel Processing: Algorithms and Architectures***
  2002 Kluwer Academic Publishers
  [Behrooz Parhami]

- ***Computer Architecture: A Quantitative Approach***
  2007 Morgan Kaufmann Publishers, Inc.
  [John L. Hennessy, David A. Patterson]

- ***SIMCAN: A Highly Configurable Simulation Framework for HPC Architectures and Applications***
  [Alberto Núñez, Javier Fernández, Jesús Carretero]

- ***Wikipedia Internet Encyclopedia Project***
  [Wikimedia Foundation]

# Anexo I. Presupuesto

En esta sección se va a proceder a detallar el presupuesto total necesario para la consecución del proyecto. Este presupuesto incluye la puesta en funcionamiento y la implantación del sistema, así como la entrega de toda la documentación al cliente. El tiempo real de realización del proyecto ha sido de 9 meses.

Las actividades que forman parte del proyecto aparecen en la figura 74. Se puede apreciar como la parte de codificación y documentación han sido las más sobrecargadas en duración de tiempo. A su vez el diagrama de Gantt es mostrado en la figura 76.

| Actividad | Duración (horas) |
|---|---|
| Análisis del problema | 60 |
| Diseño | 80 |
| Codificación | 160 |
| Pruebas | 30 |
| Documentación | 120 |

**Figura 74. Tabla de actividades**

Para la realización del proyecto ha sido necesaria la colaboración de diferente personal en los diferentes apartados. La figura 75 muestra el desglose unitario del costo por cada tipo de cualificación, asimismo expone el número de personas involucradas en el proyecto.

| Puesto | Coste / hora (€) | Coste / mes (€) | Coste / año (€) |
|---|---|---|---|
| Analista | 30,00 | 2 640,00 | 39 600,00 |
| Diseñador | 35,00 | 3 080,00 | 46 200,00 |
| Programador | 25,00 | 2 200,00 | 33 000,00 |
| Ingeniero de calidad y pruebas | 30,00 | 2 640,00 | 39 600,00 |
| Responsable de documentación | 25,00 | 2 200,00 | 33 000,00 |

**Figura 75. Tabla costes por puesto**

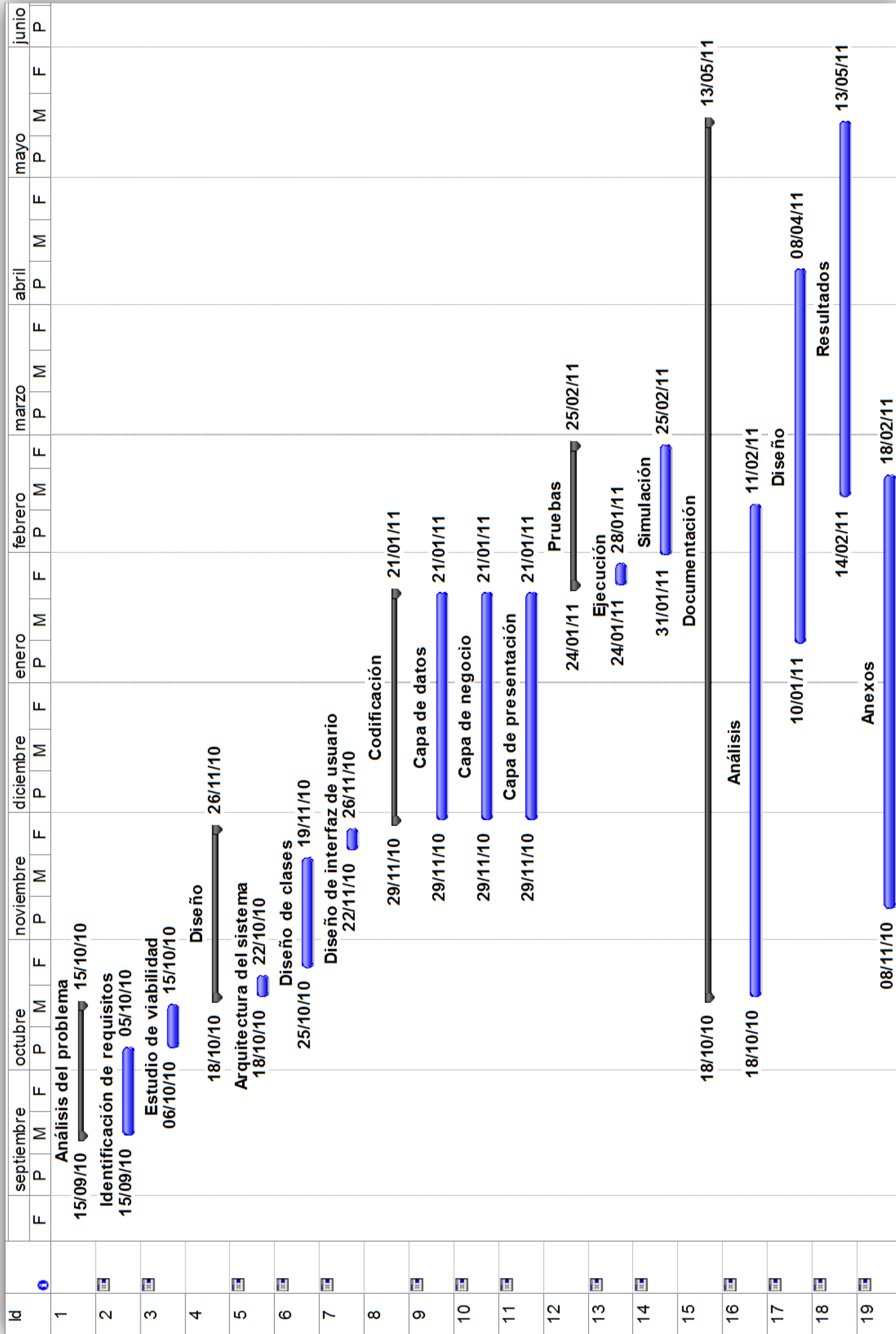


**Figura 76. Diagrama de Gantt**

Todos los costes expresados en la figura 75 son costes netos para la empresa ya después de impuestos excepto el IVA. Asimismo cabe recordar que cada empleado dispone de 15 pagas anuales, tiene la obligación de trabajar 22 días laborables mensualmente y su jornada laboral es de 4 horas.

| Puesto | Número personal | Duración / persona (horas) | Coste / hora (€) | Total (€) |
|---|---|---|---|---|
| Analista | 2 | 30 | 30,00 | 1 800,00 |
| Diseñador | 1 | 80 | 35,00 | 2 800,00 |
| Programador | 2 | 80 | 25,00 | 2 000,00 |
| Ingeniero de calidad y pruebas | 1 | 30 | 30,00 | 900,00 |
| Responsable de documentación | 1 | 120 | 25,00 | 3000,00 |
| **Total** | 7 | 450 | - | **10 500,00** |

**Figura 77. Tabla costes personal**

La figura 77 muestra el coste total por puesto imputables al proyecto, incluye el número de personal que ha participado. Asimismo en el presupuesto se incluyen los gastos del equipamiento necesario para el desarrollo del proyecto, este se muestra en la figura 78. Los gastos indirectos, tales como alquileres, luz, agua, etc. no se han incluido.

| Recurso material | Cantidad | Coste unitario (€) | Coste total (€) |
|---|---|---|---|
| Portátil Dell Vostro 1510 | 3 | 600,00 | 1 800,00 |
| Router | 1 | 50,00 | 50,00 |
| Cable RJ45 | 10 | 20,00 | 200,00 |
| Microsoft Office 2007 | 1 | 200,00 | 200,00 |
| **Total** | - | - | **2 250,00** |

**Figura 78. Tabla costes recursos materiales**

Se ha considerado necesario contemplar un margen de imprevistos de un 10% sobre el total de los gastos del presupuesto, así como un margen de beneficio para la empresa del 15%. También cabe destacar que se imputa el gasto final del IVA del 18%. En la figura 79 se muestra el presupuesto total.

| Recurso | Coste (€) |
|---|---|
| Personal relacionado con el proyecto | 10 500,00 |
| Recursos materiales | 2 250,00 |
| **Total gastos** | **12 750,00** |
| Margen de imprevistos (10%) | 1 275,00 |
| **Total gastos + imprevistos** | **14 025,00** |
| Margen de beneficio (15%) | 2 103,75 |
| **Total gastos + imprevistos + beneficios** | **16 128,75** |
| IVA (18%) | 2 903,18 |
| **Presupuesto total + IVA** | **19 031,93** |

**Figura 79. Tabla presupuesto final**

El presupuesto final es de ***diecinueve mil treinta y un euros con noventa y tres céntimos*** # **19 031,93** € #.

# Anexo II. Manual de usuario

En este apartado se introduce la práctica de la aplicación y se pretende explicar cómo debe realizarse una ejecución correcta, es decir, los pasos a seguir y la manera de operar para conseguir los resultados deseados.

Este apartado se va a dividir en tres partes. Una primera en la que se van a describir las instrucciones de instalación del programa. Una segunda en la que se va a contar las funciones e instrucciones de uso de la aplicación. Y una tercera y última en la que se detalla el formato de los ficheros de entrada al programa.

En la figura 80 se muestra los componentes de la aplicación una vez instalada. Al ser una aplicación Java todo es bastante simple. Hay que copiar la carpeta *ejecutable* del disco del programa en cualquier parte del disco de un terminal.

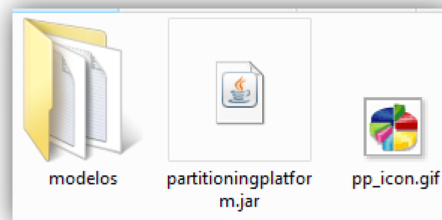


**Figura 80. Archivos de la aplicación**

Esta carpeta ejecutable contiene a su vez una carpeta *modelos* que contienen los modelos de prueba y que podrá cualquier modelo a usar. También existe el archivo *pp_icon.gif* que representa el icono de la aplicación y por último el archivo que contiene la aplicación en sí llamado *partitioningplatform.jar*.

Una vez realizada la instalación solo queda empezar a hacer funcionar la aplicación para ello si ejecutamos con doble clic el archivo *partitioningplatform.jar* debería aparecernos la pantalla de bienvenida del sistema, mostrada en la figura 81.

El funcionamiento del programa es bastante sencillo e intuitivo. La aplicación dispone de tres botones de acción *Navegar*, *Limpiar* y *Ejecutar*. El botón *Navegar* es necesario para buscar dentro del sistema de ficheros del sistema el fichero de entrada a evaluar. El botón *Limpiar* sirve para dejar en blanco los cuadros de texto y el botón de acción de la aplicación. Por último el botón de *Ejecutar*, como su nombre indica realiza una ejecución del programa y saca en el cuadro *Output* el resultado de la misma. En la figura 82 se resaltan dichos botones.

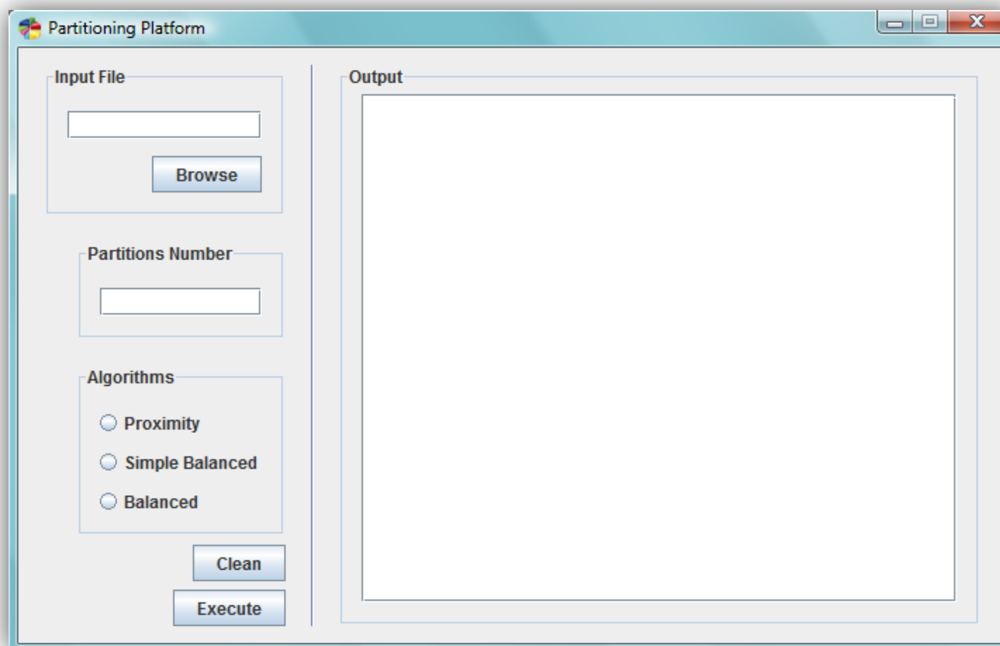


**Figura 81. Pantalla principal de la aplicación**

Para realizar una ejecución se necesita fijar tres factores. En primer lugar se debe elegir el fichero de entrada del modelo sobre el que vamos a ejecutar, esto se hace en el apartado *Input File* en la esquina superior izquierda de la interfaz.

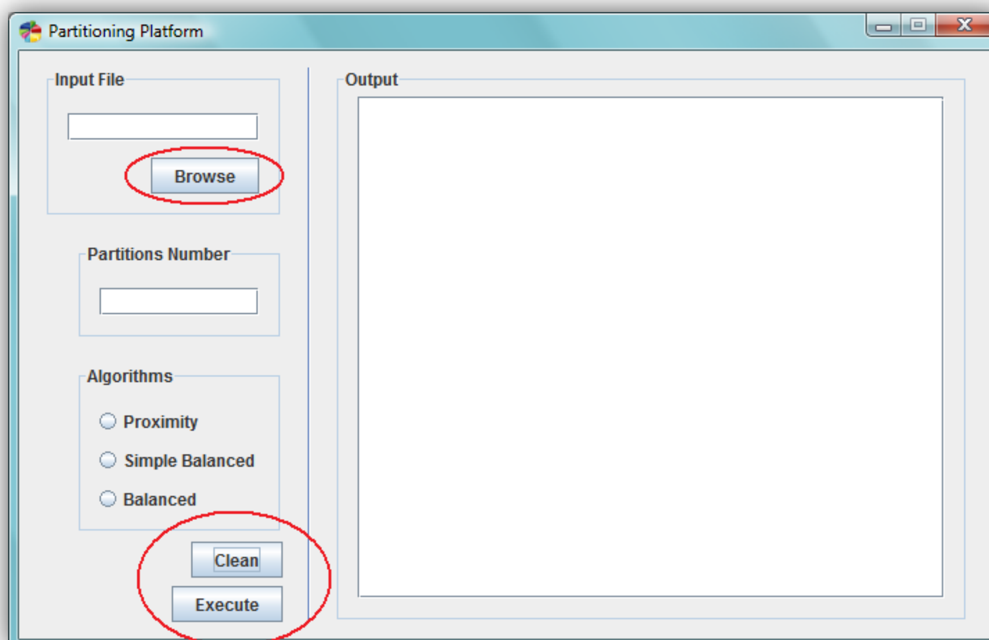


**Figura 82. Botones de la aplicación**

Para ello se tiene dos opciones, o bien escribir la ruta del fichero a mano en el cuadro de texto existente o bien pinchar en el botón *Browse* y buscar el fichero en el sistema de ficheros del terminal mediante un cuadro de dialogo de abrir ficheros. El procedimiento a realizar se muestra dentro del círculo de color rojo en la figura 83.

Una vez se tiene el modelo de entrada al programa el siguiente paso es fijar el número de particiones. Para ello se introduce en el cuadro de texto del apartado *Partitions Number* un número entero positivo. En la figura 83 se muestra dentro del círculo de color azul situado en la parte central izquierda.

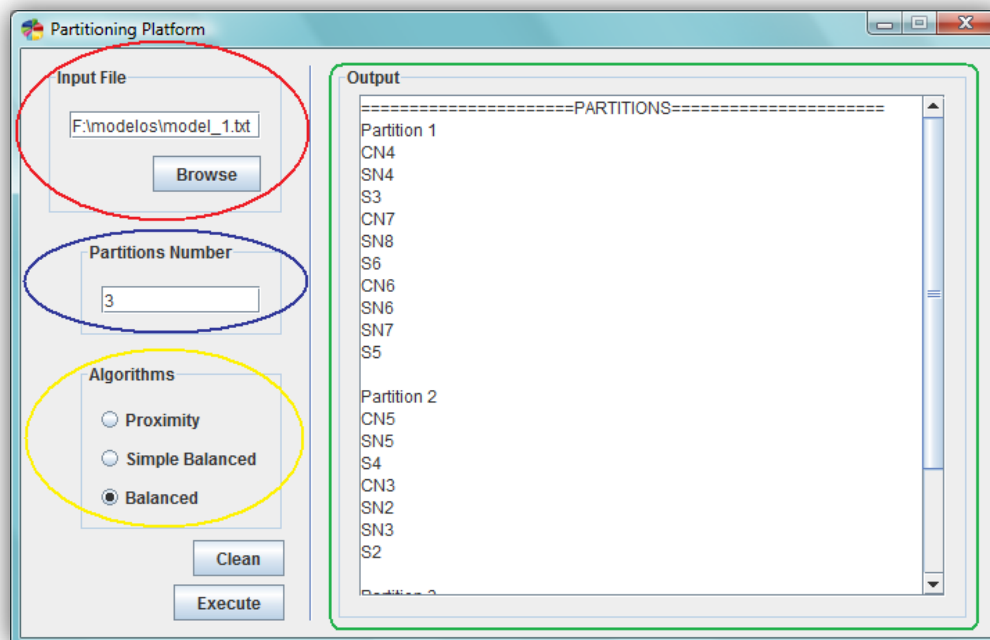


**Figura 83. Apartados de la aplicación**

El último factor a determinar es el algoritmo que se quiere utilizar en la ejecución. Para ello se tiene un botón de opción con 3 elecciones de la que solo se puede elegir una. Estas opciones son *Proximity*, *Simple Balanced* y *Balanced* que dan nombre a cada uno de los algoritmos. En la figura 83 se muestra en la parte inferior izquierda este apartado cerrado por un círculo de color amarillo.

Una vez se tienen los parámetros de entrada configurados basta con pulsar en el botón Ejecutar para que el resultado se muestre en el cuadro de texto *Output*. Mostrado en la figura 83 con un recuadro de color verde. El formato de salida que se puede apreciar en la figura 83 es bastante sencillo, muestra el nombre de cada uno de los elementos de cada una de las particiones.

Para acabar con esta sección se va a proceder a explicar el formato de los ficheros de entrada de la aplicación. Como se puede ver en la figura 84, los ficheros de entrada son ficheros de texto plano que se dividen en 4 apartados.

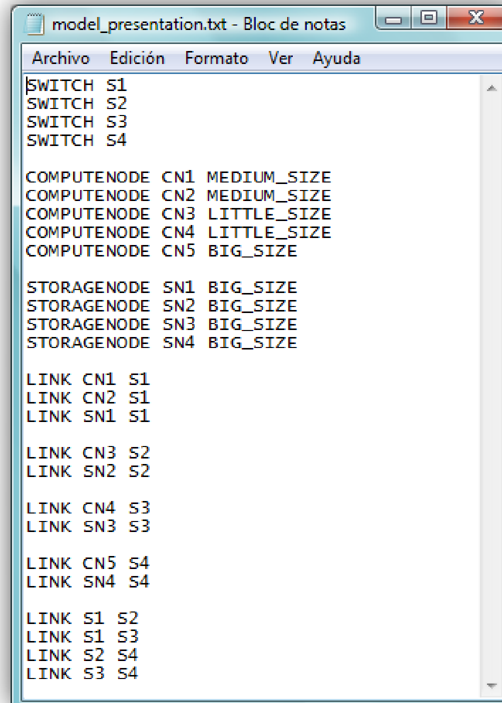


**Figura 84. Fichero de entrada de la aplicación**

Un primero que define los elementos *switch* del modelo. Para definir un *switch* basta con escribir la línea *SWITCH <nombre_nodo>*. Por convención los nombres de switch empiezan siempre con la letra S seguido de un número entero para diferenciarse entre todos.

En el segundo y tercer apartado se definen los componentes *ComputeNode* y *StorageNode* respectivamente. Para definir ambos se necesita la línea:

*[COMPUTENODE | STORAGENODE] <nombre_nodo> [LITTLE_SIZE | MEDIUM_SIZE | BIG_SIZE]*

Por convención los nombres de *ComputeNode* empiezan por CN y los de *StorageNode* por SN, seguidos ambos por un número entero. El tamaño en cuanto a número de núcleos de cada nodo se define en el último campo, se tienen tres opciones: tamaño pequeño, mediano, grande (32, 64 o 128 núcleos).

Un aspecto importante es que no se puede repetir ningún nombre de ningún tipo de nodo en la definición del fichero. Si se ponen dos nodos con el mismo nombre el programa sólo admitirá aquel que se encuentre primero en el fichero con la consiguiente pérdida de información que puede suponer tener un modelo incompleto o incluso mal formado y ni siquiera poder realizar ninguna ejecución.

La última sección comprende la definición de los enlaces del modelo de la forma:

*LINK <nombre_nodo> <nombre_nodo>*

Por restricción del programa solo serán considerados válidos dos tipos de enlaces. Un enlace entre dos *Switches*, o bien un enlace entre un *ComputeNode* o un *StorageNode* con un *Switch*.

# Anexo III. Glosario de términos

En este apartado se van a englobar todas aquellas palabras significativas dentro del proyecto, además de siglas y acrónimos. Se trata de explicar con detalle estos términos que no han podido ser expuestos en sus correspondientes secciones. Todo ello se llevará a cabo por orden alfabético ascendente.

### *A*

- ***Algoritmo:*** conjunto secuencial, definido y finito de reglas para obtener un determinado resultado en la realización de una actividad.
- ***Android:*** sistema operativo basado en Linux para dispositivos móviles, tales como smartphones o tablets.
- ***Arquitectura multicapa:*** conocida también como arquitectura de tres capas, la arquitectura multicapa es una técnica para crear aplicaciones generalmente divididas en capas de servicios de usuario, de negocios y de datos.

### *B*

- ***Beta:*** función de los algoritmos *SimpleBalancedPartitionAlgorithm* y *BalancedPartitionAlgorithm* que calcula el peso de nodos y comunicaciones de una partición cualquiera a partir de una inserción de elementos.
- ***BBDD:*** abreviatura de base de datos. Es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.
- ***BIG_SIZE:*** constante definida en la clase *ParserFile* para definir el tamaño grande de número de núcleos de un nodo cualquiera. Su valor es 128.

### *C*

- ***CN:*** Abreviación de la expresión Compute Node.

### *D*

- ***Delta:*** función de los algoritmos *SimpleBalancedPartitionAlgorithm* y *BalancedPartitionAlgorithm* que calcula el peso de los nodos de un *DomainNode* o de una partición.

## *E*

- ***Eclipse:*** es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores.
- ***Escalable:*** en telecomunicaciones y en ingeniería informática, la escalabilidad es la propiedad deseable de un sistema, una red o un proceso, que indica su habilidad para extender el margen de operaciones sin perder calidad, o bien manejar el crecimiento continuo de trabajo de manera fluida, o bien para estar preparado para hacerse más grande sin perder calidad en los servicios ofrecidos.
- ***Estudio de viabilidad:*** consiste en una iniciativa empresarial con el fin de hacerse una idea sobre la viabilidad comercial de una actividad económica.

## *F*

- ***Flexible:*** susceptible de cambios o variaciones según las circunstancias o necesidades.

## *G*

- ***Gamma:*** función de los algoritmos *SimpleBalancedPartitionAlgorithm* y *BalancedPartitionAlgorithm* que calcula el peso de las comunicaciones de un *DomainNode* o de una partición.
- ***GHz:*** el gigahercio (GHz) es un múltiplo de la unidad de medida de frecuencia hercio (Hz) y equivale a 109 (1.000.000.000) Hz.

## *H*

- ***Hardware:*** corresponde a todas las partes físicas y tangibles de una computadora: sus componentes eléctricos, electrónicos, electromecánicos y mecánicos; sus cables, gabinetes o cajas, periféricos de todo tipo y cualquier otro elemento físico involucrado.

## *I*

- ***Intel Corporation:*** es el mayor fabricante de circuitos integrados del mundo, según su cifra de negocio anual. La compañía es la creadora de la serie de procesadores x86, los procesadores más comúnmente encontrados en la mayoría de las computadoras personales.

- ***Interfaz (UML):*** es una especificación para las operaciones externas visibles de una clase, componente, u otra entidad, pero siempre sin especificar la estructura interna.
- ***IVA:*** impuesto sobre el valor añadido, es aplicado en muchos países, y generalizado en la Unión Europea.

## *J*

- ***JAVA:*** es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems a principios de los años 90.
- ***JAR:*** Java Archive, es un tipo de archivo que permite ejecutar aplicaciones escritas en lenguaje Java.
- ***JVM:*** Java Virtual Machine, es una máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el Java bytecode), el cual es generado por el compilador del lenguaje Java.

## *L*

- ***Linux:*** es un sistema operativo libre tipo Unix. Es uno de los principales ejemplos de software libre y código abierto.
- ***LITTLE_SIZE:*** constante definida en la clase *ParserFile* para definir el tamaño grande de número de núcleos de un nodo cualquiera. Su valor es 32.

## *M*

- ***Mac:*** es el nombre con el que actualmente nos referimos a cualquier computadora personal diseñada, desarrollada, construida, comercializada, y vendida por la compañía Apple Inc.
- ***Mac OS:*** Macintosh Operating System, es el nombre del sistema operativo creado por Apple para su línea de computadoras Macintosh.
- ***MB:*** el megabyte es una unidad de medida de cantidad de datos informáticos. Es un múltiplo del byte, que equivale a $10^6$ bytes.
- ***MEDIUM_SIZE:*** constante definida en la clase *ParserFile* para definir el tamaño grande de número de núcleos de un nodo cualquiera. Su valor es 64.
- ***Microsoft Corporation:*** es una empresa multinacional de origen estadounidense, fundada el 4 de abril de 1975 por Bill Gates y Paul Allen. Desarrolla, fabrica, licencia y produce

software y equipos electrónicos. Siendo sus productos más usados el sistema operativo Microsoft Windows y la suite Microsoft Office.

## N

- ***NP-completo:*** Nondeterministic Polynomial time, es el subconjunto de los problemas de decisión en NP tal que todo problema en NP se puede reducir en cada uno de los problemas de NP-completo.

## P

- ***Paralelización:*** es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente, basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo.
- ***Partición:*** es el nombre genérico que recibe cada división presente en una sola unidad física de almacenamiento de datos.
- ***PC:*** Personal Computer, es una microcomputadora diseñada en principio para ser usada por una sola persona a la vez.
- ***Pentium:*** es una gama de microprocesadores de quinta generación con arquitectura x86 producidos por Intel Corporation.

## R

- ***RAM:*** Random-Access Memory, es la memoria desde donde el procesador recibe las instrucciones y guarda los resultados.
- ***Ro:*** función de los algoritmos *SimpleBalancedPartitionAlgorithm* y *BalancedPartitionAlgorithm* que calcula el ratio entre el peso de los nodos y el peso de los enlaces o comunicaciones.

## S

- ***SIMCAN:*** una plataforma de simulación modular que puede configurarse para el modelado de una amplia gama de arquitecturas de computación de alto rendimiento.
- ***S:*** abreviación de la expresión Switch.
- ***SN:*** abreviación de la expresión Storage Node.
- ***SO:*** un sistema operativo es el programa o conjunto de programas que efectúan la gestión de los procesos básicos de un sistema informático, y permite la normal ejecución del resto de las operaciones.

- ***Software:*** es el equipamiento lógico o soporte lógico de una computadora digital. Comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas.
- ***Switch:*** es un dispositivo digital de lógica de interconexión de redes de computadores que opera en la capa de enlace de datos del modelo OSI. Su función es interconectar dos o más segmentos de red.

***U***

- ***UML:*** Unified Modeling Language, es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

***W***

- ***Windows:*** es el nombre de una familia de sistemas operativos desarrollados por Microsoft desde 1981.