Universidad de las Ciencias Informáticas

Facultad 6

Título: "T-arenal v2.0: Desarrollo del back - end".

Trabajo de Diploma para Optar por el Título de Ingeniero en Ciencias Informáticas

Autores:

César Raúl García Jacas

Daniel Marino Miralles Taset

Tutor:

MSc. Longendri Aguilera Mendoza

Junio 2009

"*Nunca consideres el estudio como una obligación sino como una oportunidad para penetrar en el bello y maravilloso mundo del saber.*"

Albert Einstein

Declaración de Autoría

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo. Para que así conste firmo la presente a los _____ días del mes _____ del año _____.

César Raúl García Jacas

Firma del Autor

Daniel Marino Miralles Taset

Firma del Autor

MSc. Longendri Aguilera Mendoza

Firma del Tutor

Datos del Contacto

Autores:

César Raúl García Jacas.
Universidad de las Ciencias Informáticas.
e-mail: crjacas@estudiantes.uci.cu

Daniel Marino Miralles Taset.
Universidad de las Ciencias Informáticas.
e-mail: dmmirayes@estudiantes.uci.cu

Tutores:

Longendri Aguilera Mendoza.
Licenciado en Ciencias de la Computación.
Master en Bioinformática.
Profesor Asistente.
e-mail: loge@uci.cu

Agradecimientos

Quizás esta sea la parte más difícil de toda la tesis. Ante todo agradecer a mi familia por los consejos brindados en esta primera etapa de mi vida. A mi madre que me dio la posibilidad de vivir, que siempre espera lo mejor de mí, que ha sabido ser también un excelente padre y que gracias a ella soy un hombre de bien; siéntete nuevamente Ingeniera. A mi abuela María por sus enseñanzas desde mis primeros días. A mis hermanas por ser mi inspiración. A mi novia Ana Li por todo su amor, apoyo y comprensión. A todos los profesores que han aportado su grano de arena en mi formación profesional, en especial a Longendri por adentrarme y conducirme con sabiduría en el mágico mundo de la programación. A los profesores Aurelio y Aidacelys por ayudarme cuando los necesité. Al profesor Maikel Zúñiga por la tranquilidad brindada para el desarrollo de este trabajo. A Daniel, Rigo, Maikel, Tellez por ser excelentes compañeros con los cuales siempre pude contar.

En general a todas las personas que me han ayudado, que han compartido estos años, que supieron ver en mi valores que otros nunca quisieron, que alguna vez se acercaron y preguntaron ¿Qué tal, cómo va la tesis? A todos ustedes les estoy eternamente agradecido y de corazón les digo: muchas gracias.

César

Agradecimientos

En primer lugar, quiero agradecer a todos los dioses de todas las religiones (solo por si acaso). A toda mi familia, por educarme y por trabajar tanto para que nunca me faltase lo que necesitaba. A mi Luz Marina, por ser madre, padre, amiga y todo; por luchar tanto y sin descanso por mi, por regalarme su corazón, por quererme tanto y por hacer de mi un ser humano, un hombre y un ingeniero. A mi mamá por darme la posibilidad de vivir y por estar siempre a mi lado enseñándome a hacerlo. A mi padre por darme las fuerzas que necesitaba en todo momento y por conducirme por el angosto camino de la correctitud, gracias por tomarme de la mano, por darme las herramientas para salir adelante y por enseñarme a utilizarlas. Discúlpenme aquellos que no mencione aquí, es que se haría interminable esta sección.

Quisiera también agradecer a todos muchachos que compartieron estos maravillosos cinco años junto a mí. A César por luchar a mi lado siempre y nunca cansarse. A Giraldo, Leniecer, Enrique, Raúl y todo el resto del piquete, sin duda compartir con ustedes fue lo mejor que me pudo pasar. A Duanis y a Rigo por toda la filosofía compartida. Agradecer también a Longendri, mi tutor y guía espiritual - profesional, por todo el trabajo educativo que hizo y por la confianza que solo los grandes saben dar. A Abdel García Carreiros por enseñarme a pensar. A los profesores Aurelio, Aidacelys, Liudmila y Aliesky por todo el asesoramiento dado, profes, síentanse autores también de este trabajo. Al profesor Maikel Zúñiga por darnos el espacio y la tranquilidad. A los profesores Reynaldo Álvares y Yadira Marrero, por comprenderme. Gracias también a todos aquellos profesores y trabajadores que hicieron posible la realización de este trabajo.

De manera general, quisiera agradecer a mi eterno grupo 6, por confiar siempre en mí, por reconocerme lo que otros no tuvieron el valor o el deseo de hacer, por ayudarme a escribir las gloriosas páginas en mi libro de la vida que tienen que ver con la Universidad. A mi equipo de proyecto, por toda la obra hecha. Al grupo de Arquitectura de la facultad por ser parte escencial de mi formación.

Finalmente, mi agradecimiento infinito a todo el pueblo trabajador de mi Cuba, gracias al esfuerzo de todos ustedes pude formarme y dar mi pequeño aporte, mi compromiso, amigos míos, es con ustedes.

Daniel

Dedicatoria

A mi madre y mis hermanas por ser mi fuente de inspiración.

A mi abuela María por todo el amor brindado desde mis primeros días.

A toda mi familia por su apoyo incondicional.

César

A la luz de mi vida, a mi Luz Marina.

Daniel

Resumen

En la Universidad de las Ciencias Informáticas (UCI) se desarrolló la Plataforma de Tareas Distribuidas v1.0, alias T-arenal, que ofrece una alternativa de cómputo y que aglutina en un solo conglomerado un conjunto de estaciones de trabajo sin intentar eliminar o pretender aminorar las amplias posibilidades de aplicación de los modelos paralelos, sino de complementar todos los medios disponibles en una gran "super-computadora virtual".

Sin embargo, este sistema aún cuenta con algunas limitaciones que impiden la utilización de las miles de computadoras presentes en la UCI. Una de estas limitaciones es que sólo utiliza un servidor mediante el cual se realizan todas las funciones de gestión y peticiones de procesamiento, conllevando a un alto grado de concurrencia que tal vez el servidor no soporte debido a las bajas prestaciones que pueda tener.

En el presente trabajo se presenta la segunda versión del back-end de T-arenal, implementada con una arquitectura de varios servidores para una mejor configuración y uso más equitativo de los clientes. Esto es posible, debido a que los elementos de cómputo estarán distribuidos por diferentes servidores de acuerdo a las prestaciones que cada uno tenga, lo cual disminuye el alto grado de concurrencia que existía en la primera versión. En esta versión al igual que en la anterior el sistema será visto por el usuario final como una sola computadora, y todos los servidores serán gestionados a través de un servidor central.

Palabras Claves: Back-end, Sistema de Cómputo Distribuido, Políticas de Asignación de Tareas

Abstract

In the University of Informatics Sciences (UCI) developed the Platform for Distributed Task v1.0, which offers an alternative calculation and combining in a single cluster one set of workstations without trying to eliminate or reduce the vast opportunities of implementation of the parallel models, but to complement all available resources in one large "virtual supercomputer".

However, this system still has some limitations that prevent the use of thousands of computers in the UCI. One of these limitations is that only by using a server which performs all the functions of managing and processing requests, leading to a high degree of concurrency that perhaps the server does not support due to low benefits you may have.

The present paper presents the second version of the back-end T-arenal, implemented with an architecture of multiple servers for better configuration and more equitable use of its clients. This is possible because the computational elements are distributed throughout different servers according to the benefits that each has, which reduces the degree of concurrency that existed in the first version. In this version as in the previous system will be seen by the end user as a single computer, and all servers will be managed through a root server.

Key Words: Back-end, Distributed Computing System, Task Assignment Polices

Índice general

Índice de figuras

Introducción

La Computación tienen su origen en el cálculo, es decir, en la preocupación del ser humano por encontrar maneras de realizar operaciones matemáticas de una forma fácil y rápida. Desde la antigüedad se evidenció que se podían utilizar aparatos y máquinas para facilitar esta tarea. El primer ejemplo que se encuentra en la historia es el ábaco, que servía para agilizar las operaciones aritméticas básicas [1].

En 1833, el profesor Charles Babbage (1791-1871) de la Universidad de Cambridge, da un importante paso en la historia de la Computación al idear un nuevo dispositivo llamado la Máquina Analítica [2], que le ocupó el resto de su vida y que nunca consiguió construir ya que las técnicas de ingeniería en aquellos tiempos no estaban tan avanzadas como su imaginación. La nueva máquina representaba un profundo avance conceptual ya que estaba diseñada para realizar cualquier operación aritmética y enlazar tales operaciones entre sí, para resolver en principio cualquier problema aritmético concebible. Este proyecto no pudo realizarse ya que el mundo no estaba listo por razones económicas, tecnológicas y no lo estuvo hasta un siglo después.

Con el transcurso de los años la humanidad ha pasado por diversas generaciones de computadoras [1] y, como se puede apreciar, la computación ha evolucionado mucho desde los tiempos de Babbage. Este avance tecnológico ha permitido utilizar a estos equipos para dar solución a problemas de cálculo en distintas esferas de la ciencia y la técnica. Sin embargo, en la actualidad nos encontramos situaciones problémicas más complejas que las de años anteriores y, a pesar del desarrollo alcanzado en la computación, aún se pueden encontrar problemas cuya solución tardaría un tiempo considerablemente largo en una computadora convencional. La razón de la demora y del intenso cómputo de algunos de estos problemas es debido, en muchas ocasiones, a la gran cantidad de datos con los cuales se trabaja.

Antecedentes que fundamentan este estudio

Las investigaciones científicas que se realizan en nuestro país, principalmente en la rama de la Biotecnología, necesitan potencia de cómputo para el procesamiento y análisis de datos, que normalmente en computadoras personales demorarían días, semanas o meses para su culminación.

El aseguramiento computacional y la necesidad de incrementar la velocidad de procesamiento fueron identificadas entre otras, como líneas principales de trabajo, con el objetivo de acelerar la obtención de resultados. Además, el desarrollo acelerado de las redes de computadoras en los últimos años ha hecho reconsiderar la utilización de las costosas supercomputadoras[1], para la ejecución de aplicaciones que demanden potencia de cómputo. Una simple computadora con memoria local y procesador de capacidades moderadas no es de mucha utilidad por sí misma, pero al ser conectada a otras máquinas a través de una red de interconexión suficientemente rápida, se exalta enormemente su utilidad ya que cientos o miles de máquinas podrían trabajar como un equipo, realizando intensos cálculos para dar solución a un determinado problema.

Lo cierto es que un gran número de empresas, organizaciones y universidades de todo el mundo tienen agrupadas un conjunto de computadoras conectadas en red formando un conglomerado, comúnmente llamado clúster [3], para trabajar en la solución de problemas de intenso cómputo. En Cuba, el Ministerio de Educación Superior aprobó la creación de clúster de computadoras en las Universidades de La Habana, Las Villas y Oriente para apoyar el cálculo masivo de datos biológicos [4].

En varias instituciones científicas cubanas también se han creado clúster de computadoras homogéneas[2] de tipo Beowulf [5]. En este tipo de clúster los nodos esclavos no tienen monitores o teclados en la mayoría de los casos, y son accedidos solamente vía remota o por terminal serie. El nodo maestro controla el clúster entero, presta servicios de sistemas de archivos a los nodos esclavos y el sistema operativo en que se basan todos los nodos es GNU/Linux.

A pesar de que algunas instituciones han establecido un clúster para realizar los cálculos que procesan

[1] Se le llama supercomputadora a la computadora con capacidades de cálculo muy superiores a las comúnmente disponibles en una institución

[2] La misma arquitectura de hardware y el mismo sistema operativo, que generalmente es GNU/Linux, con los mismos componentes de software.

grandes cantidades de datos biológicos, aún no se utiliza toda la capacidad de procesamiento posible, ya que existe un número determinado de computadoras de escritorio que no forman parte de los conglomerados y no participan en los cálculos. Utilizar las computadoras que funcionan como estaciones de trabajo, puede ser de gran utilidad a la hora de encontrar una solución a un determinado problema "de gran reto" computacional, sobre todo si se aprovecha el estado ocioso de las PCs (Personal Computers), como puede ser en horarios nocturnos o no laborables.

Situación Problémica

En la Universidad de las Ciencias Informáticas (UCI) creada en Cuba en el año 2002, está presente la Bioinformática como una rama de investigación y producción de software. Varios son los proyectos que se desarrollan con centros del Polo Científico de Cuba, entre los que se destacan, el Centro de Inmunología Molecular, Centro de Ingeniería Genética y Biotecnología, Centro de Química Farmacéutica y la Facultad de Química de la Universidad de La Habana.

En realidad, son varios los proyectos bioinformáticos que lleva la UCI y que necesitan realizar una gran cantidad de cálculos que demoran un tiempo excesivamente largo en una sola computadora. Paradójicamente, la UCI es sin lugar a dudas el centro a nivel nacional con el mayor número de computadoras personales. En estos momentos cuenta con más de 6 000 PCs distribuidas en toda la red universitaria. Es válido destacar que la mayoría de estas máquinas son Pentium 4 a 2.4 GHz de velocidad, tienen gran variedad de hardware, diferentes sistemas operativos y se encuentran conectadas mediante una red local cuya velocidad de transferencia es de 100 Mbps.

Para satisfacer la necesidad de los cálculos, la Universidad dispone al igual que otros centros del país de un clúster compuesto por ocho nodos. No obstante, se está desaprovechando tiempo de procesamiento sobre todo en los horarios nocturnos o no laborables, ya que existe un número elevado de estaciones de trabajo que no forman parte del clúster para realizar los cálculos. Es por ello que se creó la Plataforma de Tareas Distribuidas v1.0, con el objetivo de unir en un solo conglomerado un conjunto de PCs distribuidas en una red LAN.

La primera versión de la plataforma ha dado buenos resultados y ha sido de gran utilidad para varios

proyectos [6], razón por la cuál se decide extender su uso y aumentar el número de computadoras que utiliza de cientos a miles. En toda la red universitaria de la UCI existen miles de computadoras de escritorio que en estos momentos no son aprovechadas eficientemente, ya que la arquitectura de la versión 1.0 del back-end de la plataforma sólo permite un único servidor con capacidad limitada de clientes. Este y otros motivos de refinamiento han provocado el desarrollo de una nueva versión, la 2.0.

La segunda versión de la plataforma, al igual que la anterior, estará separada en dos partes esenciales: front-end y back-end. El front-end es la parte del software que interactúa con los usuarios, y el back-end es la parte oculta que procesa los datos de entrada desde el front-end. El presente trabajo de diploma reutiliza el back-end de la primera versión para satisfacer las nuevas funcionalidades de la versión a desarrollar, mientras que el front-end se encuentra en desarrollo.

Problema Científico

¿Cómo lograr que la Plataforma de Tareas Distribuidas pueda utilizar una cantidad ilimitada de recursos de cómputos?

Objeto de Estudio y Campo de Acción

Para dar respuesta al problema planteado se definió lo siguiente:

Como **objeto de estudio**: Sistemas de cómputo distribuido[3].
Como **campo de acción**: Plataforma de Tareas Distribuida v1.0.

Objetivo General

Desarrollar el back-end para la Plataforma de Tareas Distribuidas v2.0.

Objetivos Específicos

- Refinar los requisitos funcionales y no funcionales del back-end.

[3]Hardware y software para el procesamiento de información

- Refinar el diseño del back-end.
- Refinar la implementación del back-end.
- Realizar pruebas exploratorias al back-end.

Tareas

1. Estudio de las principales características de la Plataforma de Tareas Distribuidas v2.0.
2. Descripción de los requisitos funcionales y no funcionales.
3. Identificación del diagrama de casos de uso del sistema.
4. Descripción de los casos de uso del sistema.
5. Descripción de la arquitectura.
6. Desarrollo del diagrama de clases del diseño.
7. Implementación de la aplicación.
8. Realización de pruebas de camino básico.

Importancia

Las computadoras de escritorio o de mesa que funcionan como estaciones de trabajo pueden superar en número, a veces considerablemente, a la cantidad de máquinas dedicadas para un clúster Beowulf, por lo que pueden servir de gran apoyo a la hora de realizar los cálculos. Como consecuencia de la heterogeneidad[4] que pueden llegar a tener las computadoras presentes en una institución, se hizo necesario contar con un sistema de cómputo para utilizar los distintos recursos computacionales distribuidos físicamente por la institución.

Debido al gran número de recursos que puede llegar a tener el sistema y la característica que tiene este de funcionar con sólo un servidor, se hace necesario el desarrollo de una nueva versión que funcione con varios servidores y permita utilizar un número mayor de computadoras para dar solución a problemas

[4]Distintos sistemas operativos y arquitectura de hardware.

que demandan de cómputo intenso. Mientras más recursos se dispongan para los cálculos, mejor será el aprovechamiento y las prestaciones de procesamiento.

Estructura del documento

- **Capítulo 1 "Fundamentación Teórica":** se presenta una reseña bibliográfica donde se hace un análisis crítico de la literatura y una presentación de la información recopilada que está estrechamente relacionada con el tema tratado; así como las herramientas y las tecnologías a emplear durante el desarrollo del trabajo.
- **Capítulo 2 "Características del Sistema":** se presenta la descripción del sistema a desarrollar, los requisitos funcionales y no funcionales, los actores y el diagrama de casos de uso del sistema de cada módulo del back - end.
- **Capítulo 3 "Diseño del sistema":** se explican los patrones de desarrollo de software usados, se presentan los diagramas de clases del diseño y los diagramas de interacción. Finalmente se muestra el diagrama de despliegue, con el objetivo de tener una idea más clara de como quedará el sistema una vez desplegado.
- **Capítulo 4 "Implementación del Sistema":** se presenta el diagrama de componentes de cada módulo que compone el back-end y el seudocódigo de los segmentos de códigos más importantes, además de los resultados de las pruebas exploratorios realizadas.

Capítulo 1

Fundamentación Teórica

En este capítulo se abordan los principales conceptos del procesamiento paralelo, así como las tecnologías y software existentes consultados en la literatura, que ayudaron a los autores a tomar una dirección en vista de dar cumplimiento a los objetivos de este trabajo.

1.1. Procesamiento paralelo

El procesamiento paralelo funciona bajo el principio de división del trabajo en subtareas, y la asignación de estas a distintos elementos computacionales. La aceptación de este procedimiento de cómputo ha sido facilitada por los siguientes desarrollos: Procesamiento Masivamente Paralelo (MPP - Massively Parallel Processing), Multiprocesamiento Simétrico (SMP - Symmetric Multiprocessing) y el Cómputo Distribuido (Distributed Computing) [3].

El procesamiento masivamente paralelo o MPP por sus siglas en inglés, es una arquitectura computacional de alto rendimiento. Estos combinan cientos y a veces miles de microprocesadores (CPUs) en un largo gabinete con cientos Giga-bytes de memoria. El principal inconveniente de estos grandes MPP, es que típicamente el costo supera los 10 millones de dólares [3].

Los SMP igualmente tienen la característica de contar con alto rendimiento para procesar gran cantidad de información que de otra manera no sería posible lograr. En estos sistemas la arquitectura de cómputo consiste en varios procesadores unidos a un bus que comparten una memoria común [3].

El otro gran acontecimiento resultó ser la computación distribuida, que es menos costosa ya que utiliza un conjunto de computadoras existentes. El uso de este tipo de procesamiento paralelo se ha extendido debido al auge que ha alcanzado el desarrollo de las redes de computadoras. En algunos casos, varios sistemas MPPs han sido combinados mediante el cómputo distribuido para producir un inigualable poder computacional [3].

Los sistemas de tipo SMP o MPP, son diseñados para solucionar "grandes" problemas en un tiempo relativamente reducido, y son conocidos como **High Performance Computing** (HPC). A diferencia de los HPC, existen otro grupo de sistemas que trabajan bajo el paradigma **High Throughput Computing** (HTC), que utilizan el tiempo ocioso de las computadoras.

En junio de 1997 HPCWire, publicó una entrevista que trataba acerca de los entornos HTC [7]. Estos están diseñados más bien para explotar la mayor cantidad de recursos computacionales posibles, y dar solución a un gran número de problemas un poco más "pequeños" para los cuales no se tiene tanta premura. Sin embargo, con el incremento de las prestaciones de hardware que están teniendo las computadoras secuenciales, son cada vez más los problemas que se están atacando con HTC [7]. Como ejemplo se pueden mencionar, los problemas de renderización de imágenes, simulaciones computacionales, procesamiento masivo de datos experimentales, entre otros.

1.1.1. Sistemas de cómputo distribuido

Un sistema distribuido es un conjunto de computadoras autónomas que aparecen integradas ante los usuarios como una única máquina para resolver determinado problema. Los componentes del sistema no son más que hardware y software que se comunican entre sí a través de una red, preferiblemente canales de alta velocidad [8] [9]. En el caso de un sistema de cómputo distribuido, el problema a resolver por lo general requiere grandes cálculos, y lo que se trata es de reducir el tiempo en obtener la respuesta haciendo uso del procesamiento paralelo, al distribuir los cómputos de forma transparente para el usuario entre varias computadoras.

Entre los sistemas de cómputo distribuido más populares se destacan algunos que han sido concebidos para dar solución a problemas específicos en distintas ramas de la ciencia [10] [11] [12], pero que no pueden ser reutilizados en otros problemas por su especificidad. Sin embargo, existen otros que han sido creados para un propósito general [13] [14] [15], y sí pueden ser usados para dar solución a cualquier problema

de intenso cómputo, que pueda ser dividido en subtareas más pequeñas para ser repartidas entre diversos procesadores. Estos últimos son los que más interesan estudiar.

1.2. Software existentes para cluster HTC

En esta sección se mencionan algunos programas que pueden ser utilizados para construir un clúster de alta capacidad de procesamiento o clúster HTC (High Throughput Computing). Estos a diferencia de los software para HPC, solamente utilizan el tiempo ocioso de las computadoras disponibles que normalmente funcionan como estaciones de trabajo [7].

1.2.1. Condor

Condor [14] es un sistema especializado para administrar trabajos de cómputo intenso y que realiza balance de carga. Condor es el resultado del proyecto de igual nombre de la Universidad de Wisconsin-Madison. El mismo se inició en 1988 y se basa en los resultados del proyecto Remote-Unix de la misma universidad.

El sistema Condor consiste en un *administrador central de recursos (central manager)*, una o más *máquinas de envíos (submit machine)*, y una o más *máquinas de ejecución (execution machine)*. El *administrador central de recursos*, maneja los detalles de vincular los trabajos con los recursos y esto implica el conocimiento de ambos. Este conocimiento es obtenido a través de *ClassAds* (ver sección 1.2.1.2). Las *máquinas de envíos* son aquellas desde las cuales es posible mandar trabajos para la fuente (pool) de Condor. Estos trabajos son realizados en *máquinas de ejecución,* las cuales están dedicadas o son máquinas de escritorio (PC) configuradas por sus usuarios para el cómputo cuando están en inactividad (idle). Los usuarios de las *máquinas de ejecución* son conocidos como *"provedores"*, y las personas que envían trabajos para el sistema son denominados *"usuarios"*.

1.2.1.1. Características

Condor es muy flexible, soporta muchas características y diferentes plataformas. Un concepto central en Condor es el denominado *"universos" (universes)*, estos son los entornos en tiempo de ejecución que ofrecen distintas características para el usuario. Condor soporta los siguientes universos: *standard*, *vanilla*, *PVM*, *MPI*, *globus*, *java* y *scheduler*.

El universo *standard* (Figura 1.1) es heredado del predecesor de Condor, Remote-Unix. Esta característica consiste en que un proceso es iniciado en la *máquina de envío*, para procesar las llamadas al sistema que son realizadas desde las *máquinas de ejecución*. Estas llamadas permiten el acceso a archivos y a los identificadores de los usuarios para referirse al ambiente original. Si una *máquina de envío* tiene varios universos *standard* de trabajos corriendo, entonces estas máquinas tienen muchos procesos y servicios que pueden afectar el rendimiento del sistema Condor. Este universo soporta aplicaciones que puedan guardar su estado, es decir, si una *máquina de ejecución* no puede atender un trabajo después de un tiempo, entonces se guarda el estado del mismo (checkpoint) y se traslada para otra *máquina de ejecución* donde se reanudará a partir del estado guardado. Los trabajos que se ejecutan según este universo requieren reenlace (relinking) de la aplicación con librerías especiales, lo cual no es posible si no se conoce el código fuente.

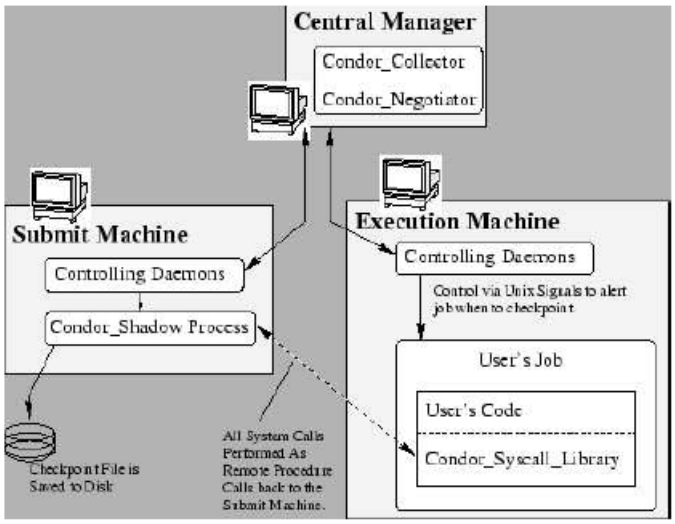

Figura 1.1: Vista general del sistema Condor usando el universo *standard*.

El universo *vanilla* ofrece menos limitaciones pero también menos características. Los trabajos que corren según este universo no pueden guardar su estado o invocar funciones remotas. La ausencia de llamadas a sistema remoto provoca que los ficheros usados por los trabajos deben estar en un sistema de archivos compartidos o deben ser explícitamente transferidos por Condor. Al no poder guardarse el estado de un trabajo, ya estos no pueden ser trasladados hacia otra *máquina de ejecución* y por lo tanto deben ser reiniciados en caso de algún error.

Los universos *PVM* y *MPI* permiten que aplicaciones escritas para estas interfaces puedan ejecutarse en

Condor. Para el uso del universo *MPI* el usuario debe seleccionar el número de nodos y la implementación específica de *MPI* a usar. Condor busca entonces un clúster Beowulf para correr el trabajo y si esto no es posible puede usar estaciones de trabajo ordinarias con este fin. El universo *PVM* soporta estilo de trabajo maestro - trabajador (master - worker), y emplea tanto clúster Beowulf y estaciones de trabajo de escritorio. Debido a la naturaleza dinámica de la *PVM*, el usuario debe especificar el mínimo y máximo número de nodos a usar cuando envía el trabajo al sistema.

El universo *globus* permite a los usuarios enviar trabajos a entornos Grid a través de Condor. El universo *java* permite que aplicaciones escritas en este lenguaje puedan también ejecutarse.

El universo *scheduler* ejecuta trabajos en una máquina local. Por lo tanto, es posible ejecutar complicados conjuntos de trabajos que dependen entre sí según Condor. Esto se realiza mediante los gestores de grafo acíclico dirigido (DAGMan, Directed Acyclic Graph Manager), el cual toma un grafo acíclico dirigido (DAG) como entrada, donde los nodos representan trabajos y las aristas las dependencias entre nodos. El DAGMan entonces trabaja como un *"meta planificador (metascheduler)"* para Condor, y envía los trabajos en el DAG en el orden correcto para preservar las dependencias.

Condor también incluye soporte heterogéneo, lo cual permite que los trabajos puedan ser subidos y ejecutados en diferentes plataformas de computadoras, si los ejecutables para cada una de estas plataformas están disponibles. Si el trabajo no está vinculado a una determinada plataforma antes de su ejecución, significa que habrá más opciones para elegir una *máquina de ejecución*. Esto supuestamente presenta un problema con la migración de los trabajos, ya que estos podrían ser cambiados hacia una plataforma diferente de la que originalmente se encontraba. Pero como el checkpoint es independiente de la aplicación, Condor asegura que el traslado de trabajos será entre máquinas de igual plataforma.

Encontrar host para trabajos en Condor, es a través de un proceso denominado *"hacer enlace" (matchmaking)*, el cual es expuesto en la siguiente sección.

1.2.1.2. Hacer Enlace (Matchmaking)

Matchmaking es una vía muy general para enlazar provedores y usuarios de un servicio. Estos enlaces se hacen a través de *"anuncios clasificados" (ClassAds, classified advertisements)*. Los provedores anuncian sus

servicios y la posibilidad de tener una conexión a ellos, mientras que los usuarios anuncian sus necesidades y deseos. Estos *ClassAds* son enviados para un servicio llamado *matchmaker*, el cual enlaza a los provedores y usuarios según sus anuncios. Los *ClassAds* tienen que estar en un formato acorde a cierto lenguaje, pero el contenido puede ser seleccionado libremente por el usuario (Figura 1.2).

```
[
Type          = "Machine";
Activity      = "Idle";
DayTime       = 36107   // current time
  // in seconds since midnight
KeyboardIdle  = 1432;   // seconds
Disk          = 323496; // kbytes
Memory        = 64;     // megabytes
State         = "Unclaimed";
LoadAvg       = 0.042969;
Mips          = 104;
Arch          = "INTEL";
OpSys         = "SOLARIS251";
KFlops        = 21893;
Name          = "leonardo.cs.wisc.edu";
ResearchGroup = {"raman", "miron",
                 "solomon", "jbasney" };
Friends       = {"tannenba", "wright" };
Untrusted     = {"rival", "riffraff" };
Rank          =
  member(other.Owner, ResearchGroup) * 10
  + member(other.Owner, Friends);
Constraint    =
  !member(other.Owner, Untrusted)
    && Rank >= 10
      ? true
    : Rank > 0
      ? LoadAvg<0.3 && KeyboardIdle>15*60
    : DayTime < 8*60*60
      || DayTime > 18*60*60;
]
```

```
[
Type           = "Job";
QDate          = 886799469;
  // Submit time secs. past 1/1/1970
CompletionDate = 0;
Owner          = "raman";
Cmd            = "run_sim";
WantRemoteSyscalls = 1;
WantCheckpoint = 1;
Iwd            = "/usr/raman/sim2";
Args           = "-Q 17 3200 10";
Memory         = 31;
Rank           =
  KFlops/1E3 + other.Memory/32;
Constraint     =
  other.Type == "Machine"
    && Arch == "INTEL"
    && OpSys == "SOLARIS251"
    && Disk >= 10000
    && other.Memory >= self.Memory;
]
```

Figura 1.2: ClassAds descritos por un provedor y un usuario respectivamente.

1.2.1.3. Planificación (Scheduling)

Condor realiza la planificación dándole prioridad a los usuarios. Cada usuario tiene una prioridad inicial de 0.5, que es la mejor prioridad que un usuario puede tener. En cada máquina que el usuario corre un trabajo incrementa su prioridad en 1, lo cual significa que las prioridades más bajas son mejores que las de mayor prioridad. La disminución de la prioridad puede ser configurado por el administrador de Condor, pero el tiempo medio de vida de una prioridad es establecido para un día por defecto, lo que significa que si un usuario deja de usar máquinas, su prioridad será reducida a la mitad por cada día que pase.

Condor no hace directamente uso de la prioridad para la planificación. Este utiliza la prioridad efectiva del usuario, la cual es la prioridad del usuario normal multiplicado por algún factor de usuario específico, permitiendo trato preferencial a usuarios individuales.

La prioridad efectiva del usuario es usada para calcular la cuota de máquinas Condor que cada usuario puede tener, lo cual es inversamente proporcional a la relación entre las prioridades de los usuarios. Esto significa que un usuario con la mitad de la prioridad de otro usuario, recibirá el doble de recursos respecto a ese usuario.

1.2.2. Berkeley Open Infrastructure for Network Computing (BOINC)

BOINC [13] es una infraestructura para la computación distribuida (Figura 1.3) desarrollada originalmente para el proyecto SETI@home [10], pero con vista a ser utilizada en otros proyectos. BOINC presenta un cliente que es compartido entre todos los proyectos. Este sistema se caracteriza por utilizar una cola central de trabajos y de tener los clientes conectados al planificador.

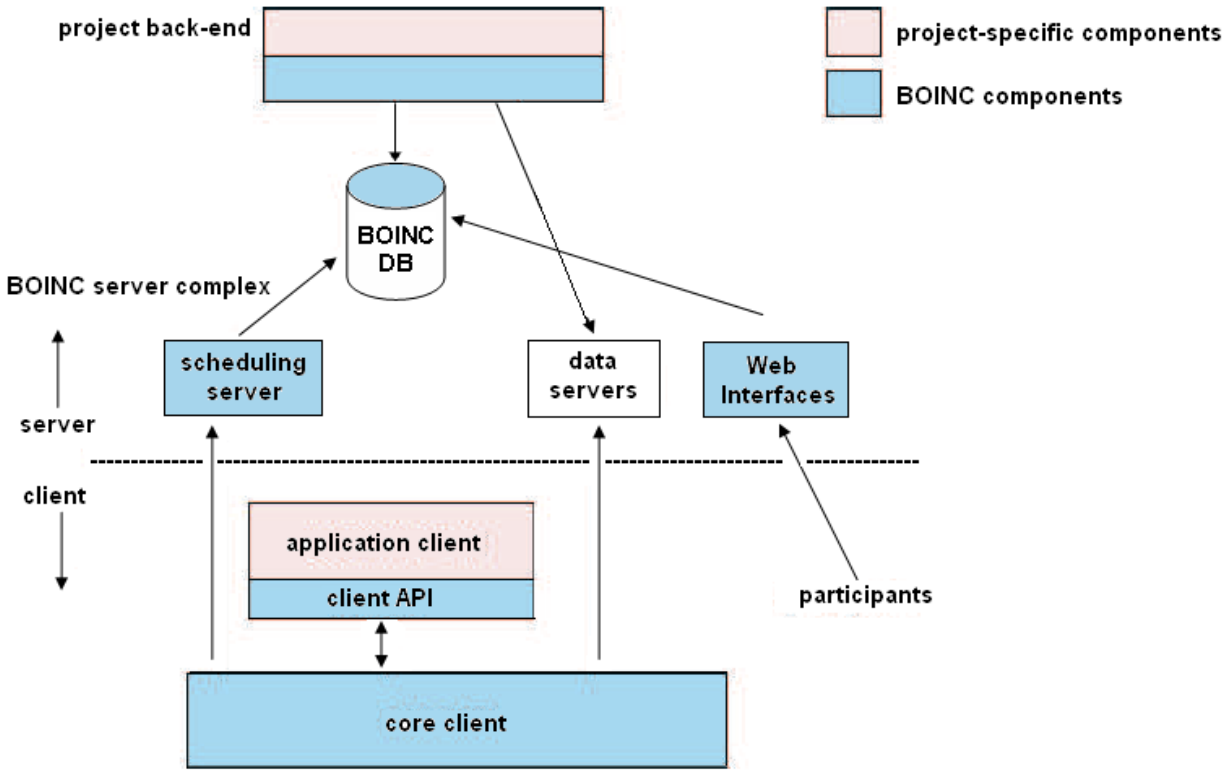

Figura 1.3: Vista general del sistema BOINC

1.2.2.1. Cliente BOINC

Un usuario que desee donar su computadora para un proyecto BOINC descarga la aplicación desde el sitio Web y posteriormente lo instala. Antes de la instalación, el usuario visita el sitio de BOINC y se suscribe. Una vez registrado, el usuario recibe un email con un identificador (ID). Este ID junto a la URL del proyecto es entrada por el usuario para firmar su cliente con un proyecto específico. Este proceso es denominado *subscripción (attaching)* del cliente a un proyecto y el proceso inverso es llamado *desconexión (detaching).*

El usuario tiene un gran control sobre el cliente según sus preferencias. Algunas de estas preferencias son aplicables a los proyectos y otras no. Entre otras cosas, el usuario puede especificar cuando el cliente puede ejecutar los trabajos, ya sea mientras el usuario está haciendo uso de la computadora o sólo usarla después de un período determinado de tiempo sin actividad por parte del usuario. Como preferencia específica de un proyecto, el usuario puede establecer el mínimo o el máximo número de trabajos que el cliente puede tener en la computadora donada para un proyecto dado. Una vez que el volumen de trabajo desciende por debajo de la cantidad mínima, el cliente contacta con el proyecto pidiendo suficiente trabajo para llegar a la cantidad máxima. Durante este período, el cliente también reporta cualquier resultado obtenido y chequea nuevas versiones de la aplicación.

La cola local de trabajos en el cliente es atendida con un comportamiento FIFO. Los clientes realizan varios trabajos a la vez según la cantidad de CPUs en el sistema, incluso si la computadora donada cuenta con NVIDIA Graphics Processing Unit (GPU), hace uso de la misma para un procesamiento más rápido. Los clientes se comunican con el servidor a través de HTTP.

1.2.2.2. Seguridad del usuario

Desde el punto de vista del usuario la seguridad está garantizada por el hashing que el proyecto realiza sobre todos los programas y datos, los cuales firma con clave privada. Esta firma se debe hacer en una computadora que no esté conectada a Internet, para minimizar el riesgo de contraer claves robadas. Esta es la única seguridad que los usuarios tienen. Los usuarios pueden estar seguros de que el código que se obtuvo desde el proyecto está firmado, pero ellos deben de tener la seguridad de que el proyecto no contiene intencionalmente código malicioso. Tampoco tienen la certeza de que alguien no haya iniciado un

proyecto falso. Por ejemplo, alguien puede montar un proyecto SET1@home, para atraer a las personas que accidentalmente visitaron la página incorrecta, en la cual se encontraba un servidor de correo distribuido para enviar mensajes spam.

1.2.2.3. Seguridad del proyecto

David Anderson, padre del proyecto BOINC, determinó que existen fundamentalmente dos maneras en que las personas pueden hackear un sistema de cómputo distribuido. Estas vías son:

- Usuarios que pueden tratar de retornar resultados erróneos con la esperanza de destruir los objetivos del proyecto.
- Usuarios que pueden tratar de retornar resultados largos con la esperanza de sobrecargar al servidor.

La primera vía está resuelta enviando el mismo cálculo para una cantidad diferentes de usuarios. Esta cantidad puede ser establecida para cada proyecto individualmente, pero en BOINC el estándar es cinco. Cuando los resultados son retornados se comparan acorde al algoritmo que el proyecto suministra. Cuando hay varios resultados iguales, entonces se escoge uno de ellos. Esto significa que si alguien devuelve un resultado incorrecto, será descartado y al usuario no se le otorgará ningún crédito.

El segundo problema se soluciona limitando el tamaño máximo de los archivos de salida que el servidor aceptará. Este es uno de los parámetros dados cuando se crea una unidad de trabajo. Cuando los clientes retornan los resultados se conectan a un programa CGI llamado *uploadhandler*. Como el nombre lo indica, controla la transferencia desde el cliente y detiene la misma si el tamaño máximo es excedido.

1.2.2.4. Servidor BOINC

El servidor BOINC consiste en un sistema de base de datos para registrar el estado de las unidades de trabajo y sus resultados, un servidor Web de datos al servicio de los clientes y que permite acceder al planificador, y cinco programas demonios ejecutándose periódicamente: *transitioner*, *validator*, *assimilator*, *file deleter* y *feeder*. Todos estos programas pueden estar en una misma PC, o pueden estar en diferentes servidores por razones de configuración. El *transitioner* se encarga de las transiciones de estado de las unidades de trabajo y los resultados, el *validator* valida los resultados, el *assimilator* hace el post procesamiento en los resultados canónicos una vez encontrado, el *file deleter* limpia el servidor de los archivos de las unidades de trabajo, y el *feeder* carga las unidades de trabajo no enviadas desde la base de datos hacia un segmento

de memoria compartida, permitiendo mejorar el rendimiento del sistema BOINC ya que limita el número de consultas.

El Planificador (Scheduler)

El planificador es un programa CGI que se ejecuta cada vez que un cliente se conecta al proyecto y pide trabajo. Esto se muestra como *scheduling server* en la figura 1.3. En lugar de consultar la base de datos, este obtiene los trabajos desde el segmento de memoria compartida cargado por el *feeder*. El *scheduler* verifica que la unidad a enviar coincida con el cliente, ya que no todos los clientes son idénticos y algunas opciones de configuración de usuario también difieren.

La Base de Datos

Una base de datos MySQL almacena información importante del servidor BOINC. Esto incluye información acerca de los usuarios registrados y sus computadoras correspondientes, sobre las aplicaciones y sus versiones, sobre los clientes y sus versiones, y por supuesto sobre las unidades de trabajo y sus resultados asociados.

1.2.3. Plataforma de Tareas Distribuidas v1.0 (t-arenal)

La Plataforma de Tareas Distribuidas [6] es un sistema de cómputo distribuido programado en Java y basado en el software **Java Based Heterogeneous Distributed Computing System** [15]. Ha sido utilizado para dar solución a varios problemas de la bioinformática, y brinda un modelo de programación de alto nivel basado en el paradigma de la POO utilizando a RMI para el paso de mensaje (Figura 1.4).

El sistema t-arenal consiste en un *servidor central (t-arenal server)* y uno o más *clientes (t-arenal client).* El *servidor central* maneja toda la información concerniente al sistema, gestiona la transferencia de ficheros vía sockets TCP, y planifica el orden en que los trabajos serán atendidos. Estos trabajos son ejecutados en los *clientes,* los cuales pueden estar dedicados o no. Las personas que envían trabajos para el sistema son denominados *"usuarios".*

En t-arenal un trabajo o ejecución no es más que una instancia de un problema previamente definido en el sistema, y del cual se pueden crear varios trabajos con el mismo o distinto juego de datos. El resultado de una ejecución constituye una solución, la cual es colocada a disposición del usuario correspondiente.

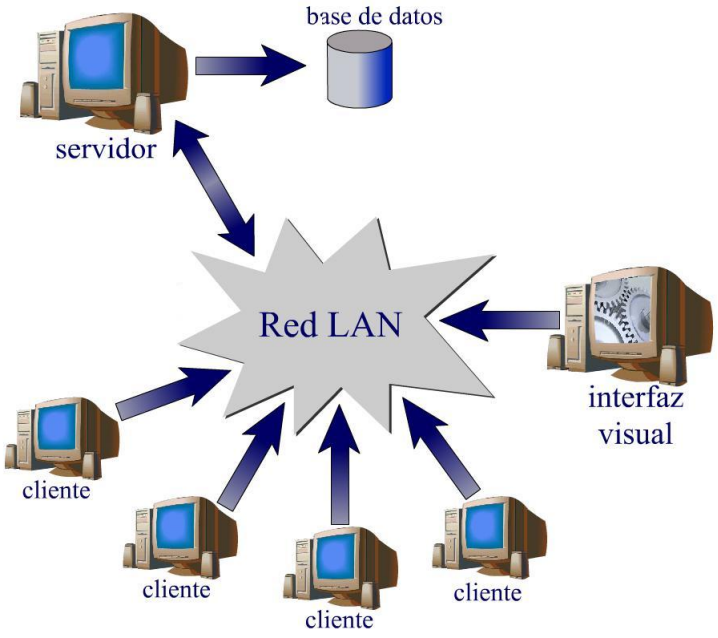

Figura 1.4: Vista general de la Plataforma de Tareas Distribuidas v1.0

1.2.3.1. Cliente t-arenal

Una persona que desee donar su computadora sólo debe instalar la aplicación *cliente*. Note que la máquina es donada al sistema, no a un proyecto en específico. Por lo tanto, un *cliente* puede trabajar en uno o varios proyectos. Sólo al *cliente* le será asignado un trabajo si el mismo se encuentra autorizado a interactuar con el sistema.

La persona que dona su máquina no presenta ningún control sobre el *cliente*. No puede decidir cuando su computadora será utilizada ni bajo que condiciones, y no puede configurar con que permisos este trabajará. El *cliente* solo atiende un trabajo a la vez y se comunica con el servidor a través de RMI.

La aplicación cliente está formada además por dos hilos (threads) que se ejecutan periódicamente: el *ClientController* y el *AlgorithmMonitorThread*. El *ClientController* esencialmente consiste en un ciclo sin límites que realiza peticiones, las procesa y retorna los resultados de las unidades de trabajo que le fueron enviadas. Si en un determinado momento no puede comunicarse con el servidor o el servidor no tiene unidades de trabajos que asignarle, entonces establece un estado de "pausa" e intentará conectarse pasado un tiempo. El *AlgorithmMonitorThread* crea, inicia y monitorea el procesamiento de una unidad de trabajo.

Si una excepción ocurre durante el procesamiento, la misma es enviada hacia el *ClientController*, quien se encargará de su tratamiento.

1.2.3.2. Servidor t-arenal

El servidor t-arenal consiste en un programa demonio: *t-arenalServer*. Este programa se encarga de realizar todas las tareas del sistema, comunicación con la base de datos, la interfaz de usuario y los *clientes*. El mismo está compuesto por varios módulos importantes: el *FileServer* y el *SocketCommunicator*, el *Timer*, y el *Scheduler*.

El propósito del *FileServer* es recibir conexiones para la transferencia de datos desde las interfaces de usuarios y los *clientes*. Cada vez que recibe una petición, crea un objeto de tipo *SocketCommunicator* quien se encargará directamente del transporte de ficheros con el solicitante. Este se ejecuta como un hilo (thread) de baja prioridad, para que la transferencia de estos archivos no afecten el rendimiento del sistema. El *Timer* chequea periódicamente las unidades pendientes y expiradas de cada trabajo que se realiza en el sistema, y se encarga además de limpiar el servidor de archivos que no se utilicen. El *Scheduler* [15] por su parte, es el encargado de la actualización dinámica de la granularidad de cada ejecución, y de la distribución eficiente del poder de procesamiento ofrecido por las máquinas donadas.

La Base de Datos

Una base de datos MySQL almacena información importante del sistema t-arenal. El sistema de gestión de base de datos puede estar en el mismo servidor que la aplicación *t-arenalServer* o en uno diferente. Esta base de datos contiene información acerca de los usuarios registrados y sus grupos, sobre los rangos de IP y los clientes que pertenecen a estos, y por supuesto sobre los problemas, las ejecuciones y soluciones obtenidas.

1.3. Sistemas de servidores distribuidos y políticas de asignación de tareas

En los últimos años los servidores distribuidos se han convertido frecuentes porque permiten una mayor potencia de procesamiento, siendo esto rentable y fácilmente escalable. Ejemplo de lo anterior tenemos:

servidores Web distribuidos, servidores de base de datos distribuidos, entre otros.

En un sistema de servidores distribuidos llegan solicitudes (tareas) que deben ser asignadas a uno de los host para su procesamiento. La regla para asignar tareas se conoce como *política de asignación de tareas (task assignment policy)*. El diseño de un sistema de servidores distribuidos a menudo se reduce a elegir *"la mejor"* política de asignación de tareas para un determinado modelo. La pregunta de cual política es *"la mejor"* es una interrogante antigua que aún sigue abierta para muchos de estos modelos.

Entre las políticas de asignación de tareas comúnmente propuestas para sistemas de servidores distribuidos se encuentran:

- **Random:** en la cual las tareas son asignadas para los host con la misma probabilidad.
- **Round - Robin:** en la cual las tareas son asignadas para los host de forma cíclica.
- **Size - Based:** en la cual todas las tareas dentro de cierto rango de tamaño son enviadas para un host particular. Esta política intenta mantener las tareas pequeñas detrás de las grandes tareas.
- **Shortest - Queue:** en la cual la tarea que llega al sistema es enviada para el host con menor cantidad de tareas en cola.
- **Least - Work - Remaining:** en la cual la tarea que llega al sistema es asignada para el host con la menor cantidad de trabajo restante según el tamaño de las tareas.

Un problema que se ha resuelto es el de la asignación de tareas en el marco de que todas son enviadas inmediatamente a un host a su llegada, y cada uno de los host atienden a sus tareas en orden FCFS (Figura 1.5). Observar sin embargo, que este modelo no excluye la posibilidad de tener una cola central de tareas, donde pueden esperar antes de ser enviadas.

Siguiendo el modelo anterior, si el tamaño de las tareas es exponencial y el proceso de arribo es Poisson, entonces la política de asignación **Shortest - Queue** es óptima [16]. En este resultado está definido como óptimo, maximizar el menor número de tareas por terminar en un tiempo t. En [17] se muestra que **Shortest - Queue** también minimiza el tiempo total para que terminen todas las tareas llegadas al sistema. En [18] se muestra que como la variabilidad de la distribución del tamaño de las tareas crece, la política **Shortest - Queue** ya no es óptima.

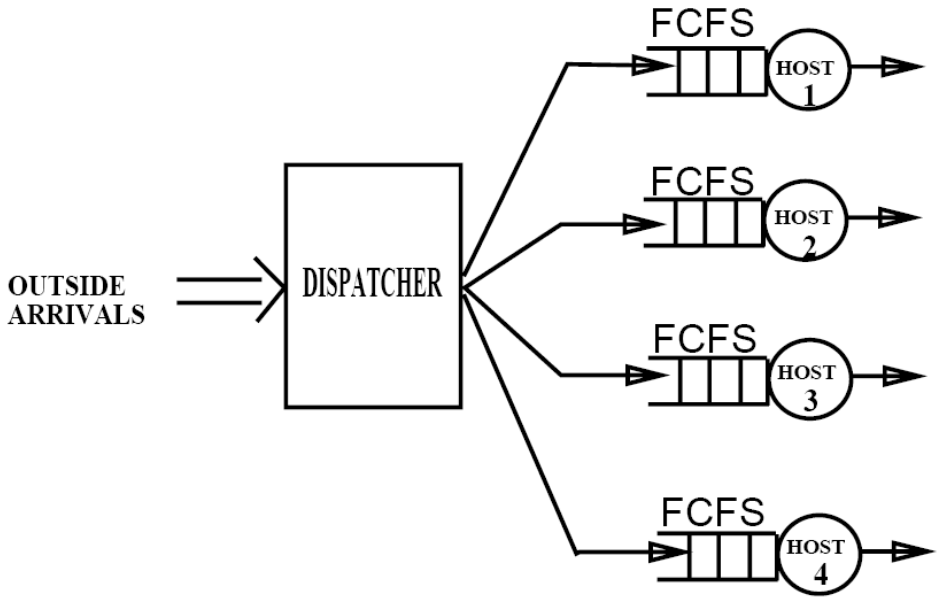

Figura 1.5: Sistema de Servidores Distribuidos

Otro escenario considerado que sigue el mismo modelo descrito con anterioridad, pero donde es conocida la edad de las tareas servidas (tiempo en servicio), es posible determinar el tiempo de espera de una tarea en cada cola. Para este escenario, en [19] se considera la regla **Shortest - Expected - Delay** la cual envía cada tarea para el host con la menor cantidad de trabajos previstos según el tiempo total de espera. En [18] se muestra que existen tamaños de tareas para los cuales esta regla no es óptima.

De la política de asignación de tareas **Least - Work - Remaining** existen dos variantes de implementación: **Least - Work - Remaining conociendo a priori el tamaño de las tareas** y **Least - Work - Remaining sin conocer a priori el tamaño de las tareas**.

En la primera variante mencionada inmediatamente que una tarea arriva al sistema, es asignada al host con menor cantidad de trabajo restante, lo cual es la suma del tamaño de las tareas en cola más los trabajos que quedan en la tarea que está siendo atendida (Figura 1.6).

En la segunda variante de la política **Least - Work - Remaining** cuando una tarea arriva al sistema, es agrupada en una cola central con un comportamiento FCFS. Sólo cuando un host está libre es que realiza una petición para atender la siguiente tarea (Figura 1.7). En [20] se demuestra que esta variante es

la mejor política de asignación de todas las políticas que no conocen de antemano el tamaño de las tareas. Este resultado es válido para cualquier distribución de tamaño de tareas y cualquier proceso de llegada al sistema.

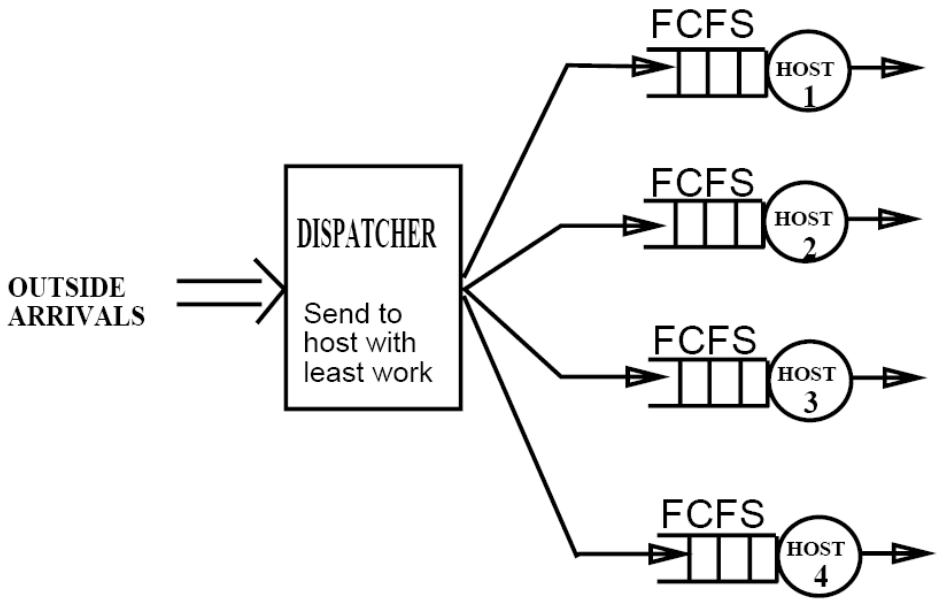

Figura 1.6: Política Least - Work - Remaining conociendo a priori el tamaño de las tareas

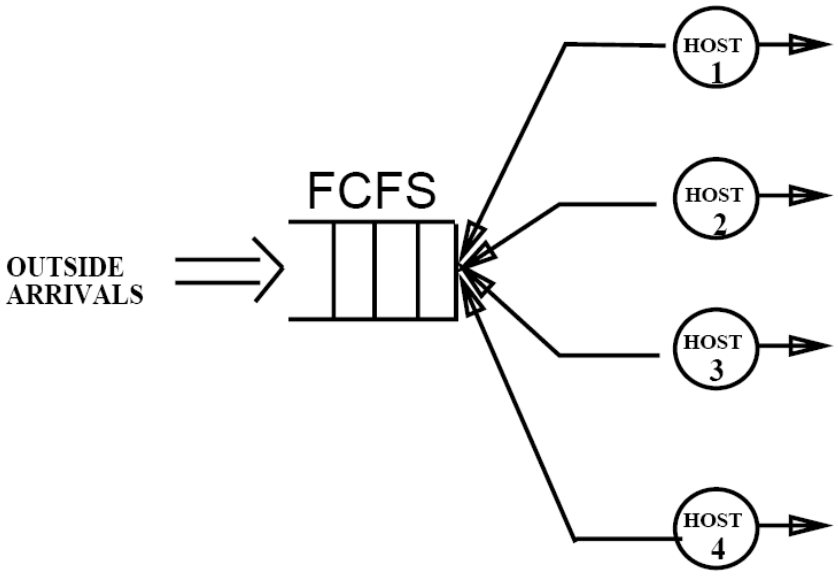

Figura 1.7: Política Least - Work - Remaining sin conocer a priori el tamaño de las tareas

1.4. Herramientas y tecnologías

Existen creencias de que un grupo de desarrollo debería organizarse en torno a las habilidades de los individuos altamente cualificados, que saben como hacer el trabajo y lo hacen bien y que raramente necesitan dirección. Esto constituye una equivocación en la mayoría de los casos, y un grave error en el desarrollo de software. Por lo tanto, es necesario un proceso que esté ampliamente disponible de forma que todos los interesados puedan comprender su papel en el desarrollo en el que se encuentran implicados. Todo esto unido a una correcta selección de las herramientas, métodos, técnicas y procedimientos que ayuden a obtener un producto de elevada calidad.

1.4.1. Metodología de desarrollo de software

Uno de los principales retos que enfrentan los desarrolladores de software es obtener un producto con calidad y de manera eficiente, lo cual supone un paso importante hacer una selección correcta de las herramientas y procesos necesarios. Con este propósito se crearon diferentes metodologías de desarrollo, las cuales son un conjunto de pasos y procedimientos que deben seguirse. Con los años y las experiencias que han ido teniendo las diferentes instituciones productoras de software, se han ido mejorando algunas metodologías y se han creado otras con el objetivo de aumentar la productividad.

Entre la familia de metodologías existen las denominadas ágiles, las cuales intentan evitar los intrincados y burocráticos caminos de las metodologías tradicionales, enfocándose en las personas y los resultados. Estas metodologías recogen ventajas de las metodologías tradicionales e incorporan características nuevas que hacen que el proceso de desarrollo sea más simple, basándose en que lo más importante en un proyecto es valorar más a los individuos que a los procesos y herramientas, al software que funciona más que a la documentación exhaustiva, a la colaboración del cliente más que a la negociación contractual, a la respuesta al cambio más que al seguimiento de un plan.

Elegir la metodología adecuada es vital para lograr un sistema de alta calidad en un tiempo razonablemente corto. Existen varias metodologías de desarrollo de software, dentro de las cuales se encuentran RUP [21], XP [22] y OpenUP [23].

Esta última es una de las metodologías ágiles de desarrollo de software. Ofrece las mejores prácticas de

una variedad de líderes en ideas sobre producción de software y comunidades de desarrollo que cubren un diverso conjunto de perspectivas y necesidades de desarrollo. Preserva las características esenciales de RUP que incluye el desarrollo interactivo, casos de uso y escenarios de conducción de desarrollo, la gestión de riesgo y el enfoque centrado en la arquitectura.

OpenUP/Basic es la forma más ágil y ligera de OpenUP, y se basa en una donación de código abierto al proceso de contenido, conocido como el Proceso Unificado Básico (BUP). La mayoría de las partes de RUP han sido excluidas de esta metodología y muchos elementos se han fusionado, siendo el resultado un proceso mucho más sencillo que coincide con los principios de RUP. OpenUP/Basic es aplicable a proyectos pequeños con grupos de 3 a 6 personas interesadas en el desarrollo rápido e interactivo.

Además de lo anterior, OpenUP/Basic define un proceso de desarrollo de software mínimo y completo. Mínimo porque solamente lo fundamental es incluido dentro del proceso, y completo porque define un conjunto de componentes que guían y definen dicho proceso de desarrollo hasta la obtención del producto. Es extensible, de manera que se pueden añadir artefactos, actividades u otro componente a la metodología, según lo requiera el sistema que se desarrolla. Por las razones antes expuestas y por decisión del Grupo de Bioinformática de la UCI, OpenUP/Basic constituye la metodología seleccionada para el desarrollo de la aplicación que se propone en el presente trabajo.

1.4.2. Lenguaje de modelado

Para dar solución al problema planteado usaremos como lenguaje de modelado UML [24] [25]. El Lenguaje Unificado de Modelado (UML) es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Captura decisiones y conocimiento sobre los sistemas que se deben construir. Se usa para entender, diseñar, hojear, configurar, mantener, y controlar la información sobre tales sistemas. Está pensado para usarse con todos los métodos de desarrollo, etapas del ciclo de vida, dominios de aplicación y medios. El lenguaje de modelado pretende unificar la experiencia pasada sobre técnicas de modelado e incorporar las mejores prácticas actuales en un acercamiento estándar. UML incluye conceptos semánticos, notación, y principios generales. Tiene partes estáticas, dinámicas, de entorno y organizativas. Está pensado para ser utilizado en herramientas interactivas de modelado visual que tengan generadores de código así como generadores de informes. La especificación de UML no define un proceso estándar pero está pensado para ser útil en un proceso de desarrollo iterativo. Pretende dar apoyo

a la mayoría de los procesos de desarrollo orientados a objetos.

1.4.3. Herramienta CASE

Las herramientas CASE (Computer Aided Software Engineering) son un conjunto de métodos, utilidades y técnicas que facilitan la automatización del ciclo de vida del desarrollo del sistema de información, completamente o en algunas fases. Como ejemplo de herramientas CASE tenemos MagicDraw, Rational Rose, Umbrello, Visual Paradigm for UML, ArgoUML y otras. Por decisión del Grupo de Bioinformática la Herramienta CASE a utilizar es Visual Paradigm.

Visual Paradigm es una herramienta CASE que soporta la última versión del Lenguaje de Modelado Unificado (UML) y la Notación del Proceso de Modelado de Negocio (BPMN), y genera código para un gran número de lenguajes de programación. Además brinda una versión libre para uso no comercial. La herramienta fue desarrollada para una amplia gama de usuarios incluyendo ingenieros de software, analistas de sistemas, analistas del negocio y arquitectos de sistemas. Permite la integración con varias herramientas de Java, y brinda además una gran interoperabilidad con otras herramientas CASE como Rational Rose.

1.4.4. IDE y lenguaje de programación

El IDE Eclipse es un entorno de desarrollo de Java que emplea módulos (en inglés plugin) para proporcionar toda su funcionalidad, a diferencia de otros entornos monolíticos donde las funcionalidades están generalmente prefijadas, las necesite el usuario o no. El mecanismo de módulos permite que el entorno de desarrollo soporte otros lenguajes de programación además de Java, así como introducir otras aplicaciones accesorias que pueden resultar útiles durante el proceso de desarrollo.

Este IDE fue seleccionado principalmente por su potente editor de código, la interfaz amigable que presenta y la existencia en el ciberespacio de una gran cantidad de plug-ins que hacen del IDE una herramienta potente. Entre otras de sus características tenemos que es un software libre y multiplataforma.

El lenguaje de programación seleccionado es Java. La razón principal de esta selección es que es independiente de plataforma y arquitectura de red. Por eso una aplicación escrita en Java, puede ejecutarse en cualquier sistema. Esta portabilidad es extremadamente importante para cualquier sistema distribuido, ya

que se espera que los clientes puedan correr en múltiples sitios en diferentes plataformas.

En los primeros días de Java, gran parte de sus críticas se originaban de su pobre rendimiento respecto a lenguajes nativos como C y Fortran. Mucho ha cambiado desde entonces. Actualmente se han producido enormes mejoras en el rendimiento de la Java Virtual Machine (JVM), principalmente atribuida a la introducción del compilador just-in-time y la tecnología hotspot [26]. Estas mejoras han dado lugar a que la ejecución de la JVM, sea comparable a la de otros lenguajes nativos [27]. Otro de los aspectos más importante de Java es el modelo de seguridad [28] [29]. Este simplifica enormemente la implementación de una estricta política de seguridad para una aplicación Java. Esto posibilita que las aplicaciones puedan cargar dinámicamente código, sin tener que preocuparse por las posibles implicaciones. Además de las características mencionadas con anterioridad, Java constituye un lenguaje *"simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico"* [30].

1.4.5. Herramientas para el desarrollo

Hibernate

Es una herramienta software libre de mapeo objeto-relacional para ambientes de desarrollo en Java. Hibernate facilita el mapeo de atributos entre una Base de Datos (BD) relacional tradicional y el modelo orientado a objetos de una aplicación. Esta potente herramienta convierte los tipos de datos utilizados por Java en los tipos de datos definidos por SQL (Structured Query Language) y viceversa. Además, es capaz de generar sentencias SQL para la persistencia y recuperación de los objetos en la BD, liberando al programador de ciertas responsabilidades y reduciendo significativamente el tiempo de desarrollo de software. También tiene la funcionalidad de crear la base de datos a partir de la información disponible de mapeo.

Gestor de Base de Datos

El gestor de base de datos utilizado es MySQL, aunque con la ayuda de Hibernate se podría cambiar de gestor sin necesidad de modificar el código ni tener que recompilar la aplicación. MySQL es un sistema de gestión de bases de datos relacional. Su diseño multihilo le permite soportar una gran carga de forma muy eficiente. Este gestor de bases de datos es probablemente el gestor más usado en el mundo del software libre, debido a su gran rapidez y facilidad de uso. Esta gran aceptación es debida en parte, a que existen

infinidad de librerías y otras herramientas que permiten su uso a través de gran cantidad de lenguajes de programación, además de su fácil instalación y configuración.

Framework Spring

Spring es un framework open source creado para ser frente a la complejidad del desarrollo de aplicaciones empresariales. Presenta un arquitectura en capas, que le permite al desarrollador decidir que componentes utilizar, además de constituir un framework coherente para el desarrollo de aplicaciones J2EE. Cada módulo o componente presente en el framework Spring puede entenderse como un ente aparte, o puede usarse combinándolo con uno o varios de los restantes. Las funcionalidades de Spring pueden ser usadas en la mayoría de los servidores J2EE. El eje central de Spring es brindar una lógica reutilizable, y objetos de acceso a datos que no estén atados a ningún servicio específico J2EE. Dichos objetos pueden ser reusados en varios entornos empresariales, aplicaciones independientes, entorno de pruebas, y muchos otros sin ningún tipo de problemas.

JUnit

JUnit permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de una clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado, si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente. En la actualidad las herramientas de desarrollo como NetBeans y Eclipse cuentan con plug-ins que permiten que la generación de las plantillas necesarias para la creación de las pruebas de una clase Java se realice de manera automática, facilitando al programador enfocarse en la prueba y el resultado esperado, y dejando a la herramienta la creación de las clases que permiten coordinar las pruebas.

1.5. Conclusiones

En un entorno HTC se debe utilizar un cluster de estaciones de trabajo con la mayor cantidad de computadoras posible para brindar una alta capacidad de procesamiento. La mayoría de los software consultados para clúster HTC son capaces de utilizar computadoras heterogéneas pero se basan en versiones compiladas para cada sistema operativo que pudieran tener las computadoras. Esto dificulta las labores de

mantenimiento y actualización del software desplegado sobre los nodos del clúster, sobre todo si se tienen muchas máquinas con una gran variedad de sistemas operativos.

Los sistemas distribuidos analizados anteriormente, están basados en el modelo cliente - servidor, donde el sistema está compuesto por un servidor y varios clientes. En la práctica este modelo ha sido utilizado con mucho éxito. Sin embargo, ya que sólo hay un servidor para todos los clientes, trae consigo un límite en el número de clientes que el sistema pueda manejar en cualquier momento, en dependencia de los recursos de red y hardware con que cuente el servidor. Una solución común es incrementar el ancho de banda de conexión y mejorar los recursos del servidor, pero esto puede ser costoso.

El software analizado en la sección 1.2.3 tiene la característica de que puede ser ejecutado independientemente de la plataforma sin necesidad de recompilar el código fuente. A este tipo de aplicación se le conoce en la literatura como "write once and run anywhere". A pesar de los cambios que se le realizaron aún presenta la limitante de no poder utilizar un gran número de PCs, por lo explicado en el párrafo anterior.

En este último sistema distribuido de las tareas que arriban al sistema no se conoce ninguna información, ya sea su tamaño o el tiempo que demoran para su culminación, por lo que se decidió por parte de los autores basar la implementación del back-end de la Plataforma de Tareas Distribuidas v2.0 sobre un modelo de servidores distribuidos, empleando para ello la segunda variante de la política de asignación de tareas **Least - Work - Remaining** descrita en la sección 1.3.

Capítulo 2

Características del Sistema

En este capítulo se describen las principales características del sistema a desarrollar, sus requisitos funcionales y no funcionales, y los actores que intervienen en el mismo. Además, se presenta el diagrama de casos de usos del sistema, así como una breve descripción de los casos de usos identificados.

2.1. Breve descripción del sistema

El sistema a desarrollar permitirá hacer un mejor uso de los elementos de cómputo con la implementación de una arquitectura de varios servidores, siguiendo la política de asignación de tareas Least - Work - Remaining. En esta versión, el back-end estará compuesto por tres módulos: *servidor central*, *servidor de peticiones* y *cliente* (Figura 2.1).

El *servidor central* fungirá como el servidor de la versión 1.0, con el único cambio de que no será quien atienda las ejecuciones. El *servidor de peticiones* solamente podrá atender una ejecución al mismo tiempo, pero además podrá colaborar con otro servidor de peticiones en la realización de una ejecución. Existirán uno o varios servidores de peticiones según las necesidades del usuario final. El *cliente* por su parte, seguirá teniendo la misma funcionalidad que el de la versión anterior.

Esta versión posibilitará además, guardar el estado de las ejecuciones periódicamente, y que la comunicación entre los componentes de la misma no sea sólo por RMI sino también por HTTP.

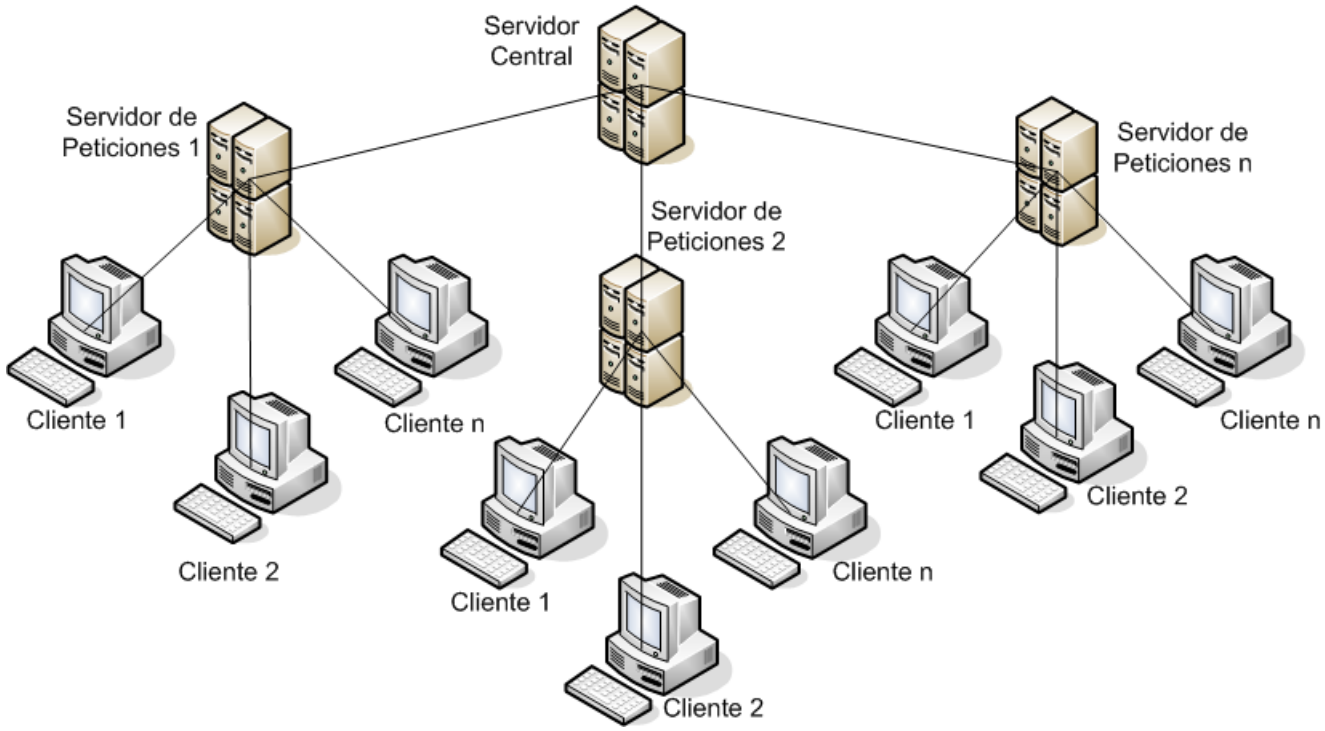

Figura 2.1: Propuesta del back-end de la Plataforma de Tareas Distribuidas v2.0

2.2. Modelo de dominio

El modelo de dominio es una representación visual de los conceptos del mundo real significativos para un problema. El diagrama de clases del modelo de dominio del presente trabajo se muestra en la figura 2.2.

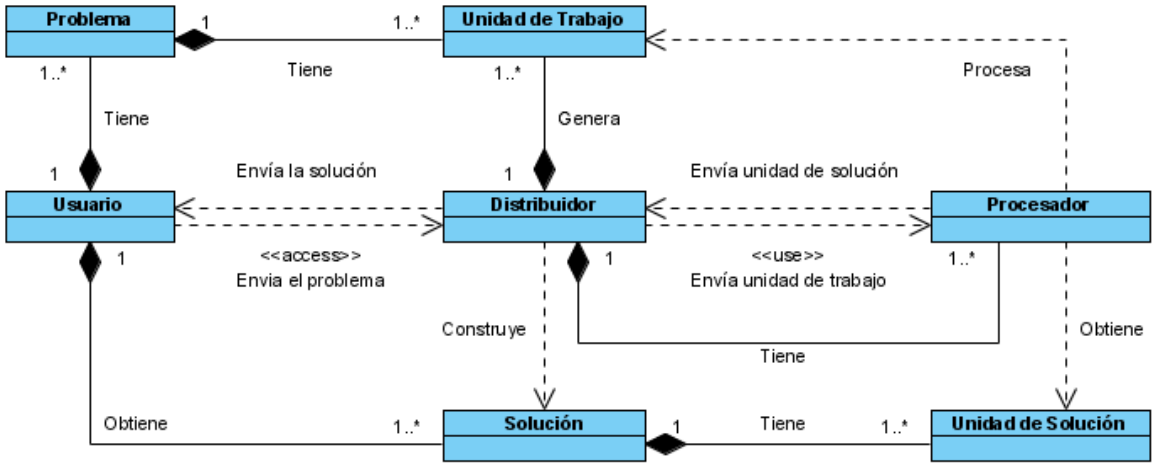

Figura 2.2: Diagrama de clases del modelo de dominio

Para una mejor comprensión, a continuación se describe cada una de las clases que intervienen en el diagrama mostrado:

- **Usuario:** representa la entidad que accede a un *problema* y desea distribuirlo.
- **Problema:** representa la entidad que es accedida por un determinado *usuario* y que se va a resolver de manera distribuida.
- **Distribuidor:** representa la entidad encargada de aceptar un *problema*, dividirlo en pequeñas *unidades de trabajo*, repartir estas unidades por los *procesadores*, construir la *solución* final y devolver esta última al *usuario*.
- **Unidad de Trabajo:** representa una porción del *problema*, cuando este es dividido en pequeñas partes.
- **Procesador:** representa la entidad encargada de procesar la *unidad de trabajo* asignada y que obtiene una *unidad de solución* correspondiente al procesamiento realizado, la cual es retornada al *distribuidor* para confeccionar la solución final del *problema*.
- **Unidad de Solución:** representa la entidad obtenida a partir del procesamiento de una *unidad de trabajo*. Es utilizada por el *distribuidor* para obtener la *solución*.
- **Solución:** es la entidad final que se le devuelve al usuario, y que es obtenida a partir de las *unidades de solución* retornadas por cada *procesador*.

2.3. Especificación de los requisitos del sistema

2.3.1. Requisitos funcionales

Los requerimientos funcionales son capacidades o condiciones que el sistema debe cumplir, y se mantienen invariables sin importar con que propiedades o cualidades se relacionen. En el presente trabajo se identificaron los siguientes:

1. Autenticar usuario.
2. Gestionar grupos.
   2.1. Adicionar un grupo.

2.2. Eliminar grupos.

2.3. Modificar datos de un grupo.

2.4. Devolver los grupos existentes en el sistema.

3. Gestionar usuarios.

3.1. Adicionar un usuario.

3.2. Eliminar usuarios.

3.3. Modificar un usuario.

3.4. Cambiar la contraseña de un usuario.

3.5. Devolver los usuarios existentes en el sistema.

4. Gestionar rangos IP de un servidor de peticiones.

4.1. Adicionar un rango IP a un servidor de peticiones.

4.2. Permitir o no un rango IP en un servidor de peticiones.

4.3. Eliminar rangos IP de un servidor de peticiones.

4.4. Devolver desde un servidor de peticiones, el rango IP cuyo intervalo de direcciones es conocido.

4.5. Devolver los rangos IP existentes en un servidor de peticiones.

5. Adicionar un cliente en un servidor de peticiones.

6. Gestionar clientes de un servidor de peticiones.

6.1. Permitir o no un cliente en un servidor de peticiones.

6.2. Eliminar clientes de un servidor de peticiones.

6.3. Devolver los clientes existentes en un servidor de peticiones.

6.4. Devolver los clientes autorizados a interactuar con un servidor de peticiones.

6.5. Devolver los clientes no autorizados a interactuar con un servidor de peticiones.

6.6. Devolver los clientes reportados a un servidor de peticiones, en un tiempo determinado.

6.7. Devolver los clientes no reportados a un servidor de peticiones, en un tiempo determinado.

7. Gestionar problemas.
    7.1. Adicionar un problema.
    7.2. Eliminar problemas.
    7.3. Devolver los problemas existentes en el sistema.
    7.4. Devolver los problemas a los que accede un usuario.
    7.5. Devolver los problemas que administra un usuario.
8. Ejecutar un problema.
9. Gestionar acceso a problemas.
    9.1. Asignar un problema a un usuario.
    9.2. Quitar a un usuario el acceso a un problema.
10. Crear una ejecución.
11. Cambiar la prioridad a una ejecución.
12. Gestionar ejecuciones.
    12.1. Detener una ejecución.
    12.2. Devolver las ejecuciones existentes en el sistema.
    12.3. Devolver las ejecuciones que pertenecen a un usuario.
13. Monitorear ejecuciones.
    13.1. Devolver el estado de una ejecución.
    13.2. Devolver los errores de una ejecución.
14. Gestionar funcionamiento de las ejecuciones.
    14.1. Establecer el estado de pausa a una ejecución.
    14.2. Quitar del estado de pausa a una ejecución.

15. Adicionar una solución.

16. Gestionar soluciones.

    16.1. Eliminar soluciones.

    16.2. Devolver las soluciones existentes en el sistema.

    16.3. Devolver las soluciones que pertenecen a un usuario.

17. Descargar una solución.

18. Adicionar un servidor de peticiones.

19. Gestionar servidores de peticiones.

    19.1. Permitir o no un servidor de peticiones.

    19.2. Eliminar servidores de peticiones.

    19.3. Devolver los servidores de peticiones existentes en el sistema.

20. Gestionar actualizaciones de clientes.

    20.1. Adicionar una actualización.

    20.2. Eliminar una actualización.

    20.3. Devolver las actualizaciones de clientes existentes en el sistema.

21. Obtener información estadística de un servidor de peticiones.

    21.1. Devolver de un servidor de peticiones la cantidad de clientes por rangos IP.

    21.2. Devolver de un servidor de peticiones la cantidad de clientes por sistemas operativos.

    21.3. Devolver de un servidor de peticiones la cantidad de clientes según la capacidad de procesamiento.

    21.4. Devolver de un servidor de peticiones la cantidad de clientes según la memoria disponible.

22. Definir la capacidad máxima del directorio de ficheros a utilizar por un cliente.

23. Gestionar la configuración de conexión a un servidor de peticiones.

    23.1. Modificar opciones de conexión a un servidor de peticiones.

23.2. Obtener opciones de conexión a un servidor de peticiones.

24. Obtener información estadística de un cliente.

24.1. Obtener de un cliente información sobre el uso del procesador.

24.2. Obtener de un cliente información sobre el uso de la memoria.

24.3. Obtener de un cliente información sobre el uso del disco duro.

25. Realizar procesamiento en un cliente.

25.1. Solicitar unidad de trabajo a un servidor de peticiones.

25.2. Procesar unidad de trabajo.

25.3. Devolver los resultados de la unidad de trabajo procesada a un servidor de peticiones.

26. Actualizar versión en un cliente.

27. Atender tarea asignada por el servidor central.

27.1. Crear una unidad de trabajo para un cliente.

27.2. Procesar el resultado de una unidad de trabajo devuelto por un cliente.

27.3. Aumentar el tiempo de atención de una unidad de trabajo.

27.4. Recibir notificación sobre algún error ocurrido durante el procesamiento de una unidad de trabajo.

28. Realizar chequeo en el servidor central.

29. Gestionar tarea en el servidor de peticiones

29.1. Solicitar tarea al servidor central.

29.2. Controlar realización de la tarea asignada por el servidor central.

30. Asignar tarea a un servidor de peticiones.

30.1. Asignar una ejecución a un servidor de peticiones.

30.2. Establecer colaboración de un servidor de peticiones con otro.

31. Recibir reporte sobre una ejecución asignada.

    31.1. Recibir reporte sobre una ejecución que es atendida por un servidor de peticiones.

    31.2. Recibir reporte sobre una ejecución que no puede ser atendida por un servidor de peticiones.

    31.3. Recibir confirmación sobre una ejecución que ha finalizado.

32. Guardar estado de una ejecución.

33. Iniciar una ejecución a partir de un estado.

2.3.2. Requisitos no funcionales

Los requerimientos no funcionales son propiedades o cualidades que el producto debe tener, y que harán del mismo un sistema confiable y seguro.

Transparencia

- El sistema ocultará la naturaleza distribuida, permitiendo que los usuarios interactúen a través de un front-end, y trabajen como si se tratara de una sola máquina.
- Los programadores o desarrolladores no tienen que encargarse de repartir el cálculo entre los nodos del sistema, solamente tienen que programar cómo dividir el problema y cómo integrar las soluciones parciales. El trabajo "sucio" de distribuir físicamente las unidades más pequeñas del problema y llevar el control de que estas se realicen queda a cargo del sistema.

Eficiencia

- La solución a los problemas se obtiene más rápidamente haciendo uso del sistema distribuido, que la respuesta que daría una simple computadora.

Flexibilidad

- Un proyecto en desarrollo como la implementación de un sistema distribuido, debe estar abierto a modificaciones que mejoren su funcionamiento. El presente trabajo es lo suficientemente flexible para que cualquier cambio a realizar no requiera la parada de todo el sistema y la recompilación de todo el código. Si se realizan cambios en el módulo del cliente, cambiarlo es tan simple como colocarlo en el

servidor, y a medida que cada elemento de cómputo realiza peticiones de trabajo, se le envía la nueva versión para su actualización automática.

Escalabilidad

- El tamaño de una red varía en dependencia de la institución que la mantenga, el sistema debe comportarse estable tanto en redes pequeñas y grandes. También debe garantizar un adecuado mantenimiento y actualización, empleando el mínimo de personal. Ampliar el sistema en cuanto a unidades de cómputo, es tan sencillo como instalar el módulo cliente en una PC y configurarle la dirección física del servidor al cual debe reportarse.

Fiabilidad

- La interacción con el sistema, estará sometida a un proceso de autenticación del usuario.
- Asegura al 100 % que el problema del usuario será resuelto, independientemente de los problemas ajenos que existan, ya sean fallos de red, electricidad, apagado de máquinas, entre otros. Si un elemento de cómputo (cliente) se desconecta, la unidad en la que el mismo estaba trabajando no se pierde, ya que se almacena una copia y luego se envía a otro cliente pasado un tiempo.
- Se garantiza comunicaciones seguras entre los componentes del sistema encriptando todo el tráfico de información.

Portabilidad

- El sistema será multiplataforma, razón por la cual se podrá ser utilizado en cualquier sistema operativo y arquitectura de hardware.

Software

- Para el uso del sistema se deberá disponer de la máquina virtual de Java 1.5 en todas las PCs, y el sistema de gestión de base de datos MySQL 5.0 solamente en aquellos ordenadores donde sean instalados los servidores.

2.4. Definición de los casos de usos del sistema

2.4.1. Actores del sistema

Los actores representan a terceros fuera del sistema que interactúan con él. En el sistema que se describe se identificaron los siguientes actores:

Actor	Descripción
Front-end del sistema	Representa un software externo que se conecta con el servidor central del sistema para intercambiar información.
Front-end del cliente	Representa un software externo que se conecta a la aplicación cliente para intercambiar información.
Cliente	Representa la aplicación encargada de realizar solicitudes de tareas a un servidor de peticiones para su procesamiento.
Reloj	Representa al actor que inicia casos de usos cada cierto período de tiempo.

2.4.2. Listado de casos de usos del sistema

La forma en que los actores usan el sistema es representada a través de los casos de usos. Estos últimos son artefactos narrativos que describen, bajo la forma de acciones y reacciones, el comportamiento del sistema desde el punto de vista del actor.

Los casos de usos identificados en el presente trabajo son enunciados a continuación:

Orden	Nombre	Prioridad	Breve descripción
1	Autenticar Usuario (ver Anexo A pág 97)	Crítico	El front-end inicia el caso de uso cuando un usuario decide interactuar con el sistema. El sistema realiza las verificaciones de los datos enviados y finalmente lo acepta o lo rechaza.

2	Gestionar Grupos	Secundario	El caso de uso inicia cuando el front-end envía una petición para adicionar, modificar, eliminar u obtener grupos. Una vez realizada alguna de estas acciones finaliza el caso de uso.
3	Gestionar Usuarios	Secundario	El caso de uso inicia cuando el front-end envía una petición para adicionar, modificar, eliminar u obtener usuarios. Una vez realizada alguna de estas acciones finaliza el caso de uso.
4	Gestionar Rangos IP de un Servidor de Peticiones	Secundario	El caso de uso inicia cuando el front-end envía una petición para adicionar, permitir, eliminar u obtener rangos IP. Se establece la comunicación con el servidor de peticiones correspondiente y se realiza la operación deseada.
5	Gestionar Clientes de un Servidor de Peticiones	Secundario	El caso de uso inicia cuando el front-end envía una petición para permitir, eliminar u obtener clientes. Se establece la comunicación con el servidor de peticiones correspondiente y se realiza la operación deseada.
6	Gestionar Problemas (ver Anexo A pág 98)	Crítico	El caso de uso inicia cuando el front-end envía una petición para adicionar, eliminar u obtener problemas. Una vez realizada alguna de estas acciones finaliza el caso de uso.
7	Ejecutar Problema (ver Anexo A pág 103)	Crítico	El caso de uso inicia cuando el front-end envía una petición para ejecutar un problema en el sistema. Luego transfiere los datos necesarios para el funcionamiento de la ejecución creada y finalmente le indica al sistema que ya puede atenderla.

8	Gestionar Acceso a Problemas	Secundario	El caso de uso inicia cuando el front-end envía una petición para asignar o quitar el acceso de un problema a un usuario. Una vez realizada alguna de estas acciones finaliza el caso de uso.
9	Gestionar Ejecuciones	Secundario	El caso de uso inicia cuando el front-end envía una petición para detener u obtener las ejecuciones existentes. Una vez realizada alguna de estas acciones finaliza el caso de uso.
10	Monitorear Ejecuciones	Secundario	El caso de uso inicia cuando el front-end envía una petición para obtener el estado en que se encuentra una ejecución o los errores que han ocurrido en la misma.
11	Gestionar Funcionamiento de las Ejecuciones	Secundario	El caso de uso inicia cuando el front-end envía una petición para pausar o reanudar una ejecución. Una vez realizada alguna de estas acciones finaliza el caso de uso.
12	Gestionar Soluciones	Secundario	El caso de uso inicia cuando el front-end envía una petición para eliminar u obtener las soluciones existentes. Una vez realizada alguna de estas acciones finaliza el caso de uso.
13	Descargar Solución	Secundario	El caso de uso inicia cuando el front-end establece la comunicación con el servidor central para descargar la solución deseada hacia la máquina desde la cual realiza la petición.
14	Gestionar Servidores de Peticiones	Secundario	El caso de uso inicia cuando el front-end envía una petición para eliminar, permitir u obtener servidores de peticiones. Una vez realizada alguna de estas acciones finaliza el caso de uso.

15	Gestionar Actualizaciones de Clientes	Secundario	El caso de uso inicia cuando el front-end envía una petición para adicionar, eliminar u obtener actualizaciones de clientes. Una vez realizada alguna de estas acciones finaliza el caso de uso.
16	Obtener Información Estadística de un Servidor de Peticiones	Secundario	El caso de uso inicia cuando el front-end envía una petición para obtener información estadística de un servidor de peticiones específico.
17	Gestionar el Acceso al Disco Duro de un Cliente	Secundario	El front-end inicia el caso de uso cuando pide la información sobre el tamaño máximo en el disco duro que se puede utilizar por el directorio de ficheros del cliente y posteriormente decidirá cambiarlo o no.
18	Gestionar Configuración de Conexión a un Servidor de Peticiones	Secundario	El front-end inicia el caso de uso cuando solicita la información de conexión al servidor. El cliente devuelve dicha información y finalmente decide si la cambiará o no.
19	Obtener Información Estadística de un Cliente	Secundario	El front-end inicia el caso de uso cuando solicita información estadística a la aplicación cliente sobre su funcionamiento.
20	Realizar Procesamiento (ver Anexo A pág 105)	Crítico	El caso de uso inicia cuando un cliente hace una solicitud de unidad de trabajo al servidor de peticiones correspondiente para luego realizar su procesamiento.
21	Atender Tarea Asignada por el Servidor Central (ver Anexo Apág 109)	Crítico	El caso de uso inicia cuando un cliente hace al servidor de peticiones correspondiente una solicitud de procesamiento, devuelve el resultado obtenido o pide aumentar el tiempo de atención de la unidad que le fue asignada.

22	Realizar Chequeo en el Servidor Central (ver Anexo A pág 114)	Crítico	El caso de uso inicia cuando después de un tiempo se ejecuta un hilo (thread) para mantener la consistencia de los datos, eliminar los ficheros y directorios que no se utilicen y controlar el estado en que se encuentran las ejecuciones asignadas.
23	Gestionar Tarea en el Servidor de Peticiones (ver Anexo A pág 115)	Crítico	El caso de uso inicia cuando después de un tiempo, el servidor de peticiones se comunica con el servidor central y solicita atender una tarea. Si la tarea asignada es una ejecución, entonces lleva el control de la misma, de lo contrario espera un tiempo e inicia nuevamente el caso de uso.
24	Asignar Tarea a un Servidor de Peticiones (ver Anexo A pág 117)	Crítico	El caso de uso inicia cuando un servidor de peticiones realiza una solicitud de trabajo. El servidor central le asigna la atención de una ejecución o le indica que colabore con otro servidor de peticiones.
25	Recibir Reporte sobre una Ejecución Asignada (ver Anexo A pág 118)	Crítico	El caso de uso inicia cuando el servidor de peticiones decide notificar al servidor central si atiende la ejecución asignada, si no la puede atender o si esta ya finalizó.

2.4.3. Diagrama de casos de usos del sistema

Los diagramas de casos de uso del sistema representan gráficamente a los procesos y su interacción con los actores.

El diagrama de casos de usos del sistema del presente trabajo fue organizado por paquetes, para una mejor comprensión y claridad, según los actores que intervienen en el mismo. Por cada paquete se muestra el diagrama de casos de usos correspondiente.

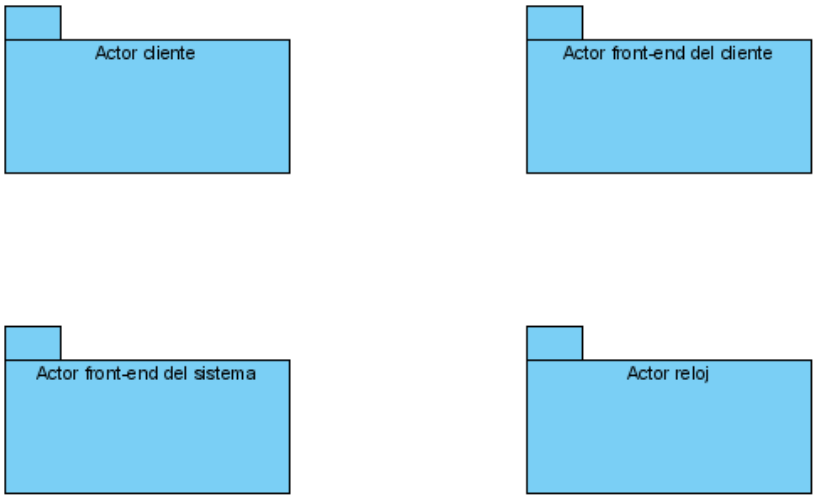

Figura 2.3: Diagrama de casos de usos del sistema.

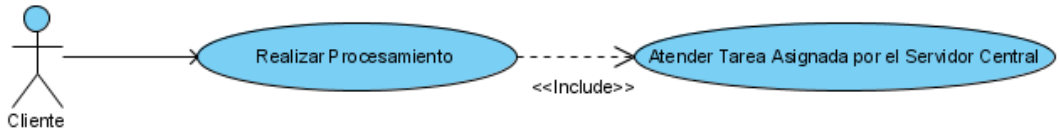

Figura 2.4: Diagrama de casos de usos del paquete *Actor cliente*.

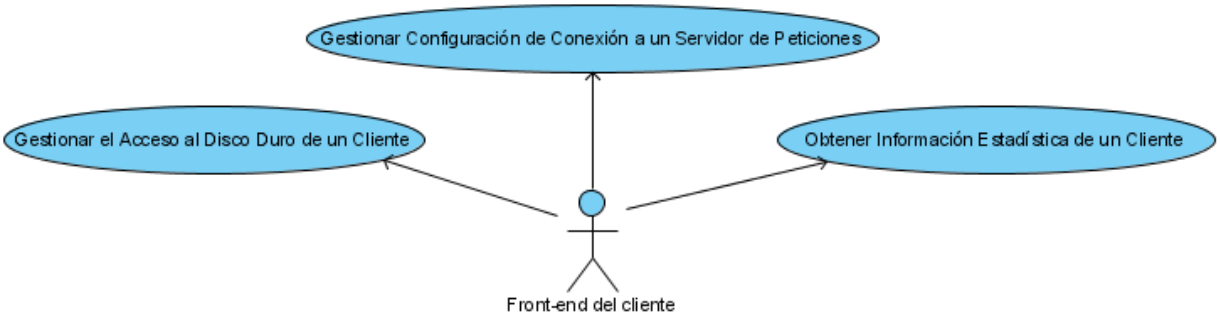

Figura 2.5: Diagrama de casos de usos del paquete *Actor front-end del cliente*.

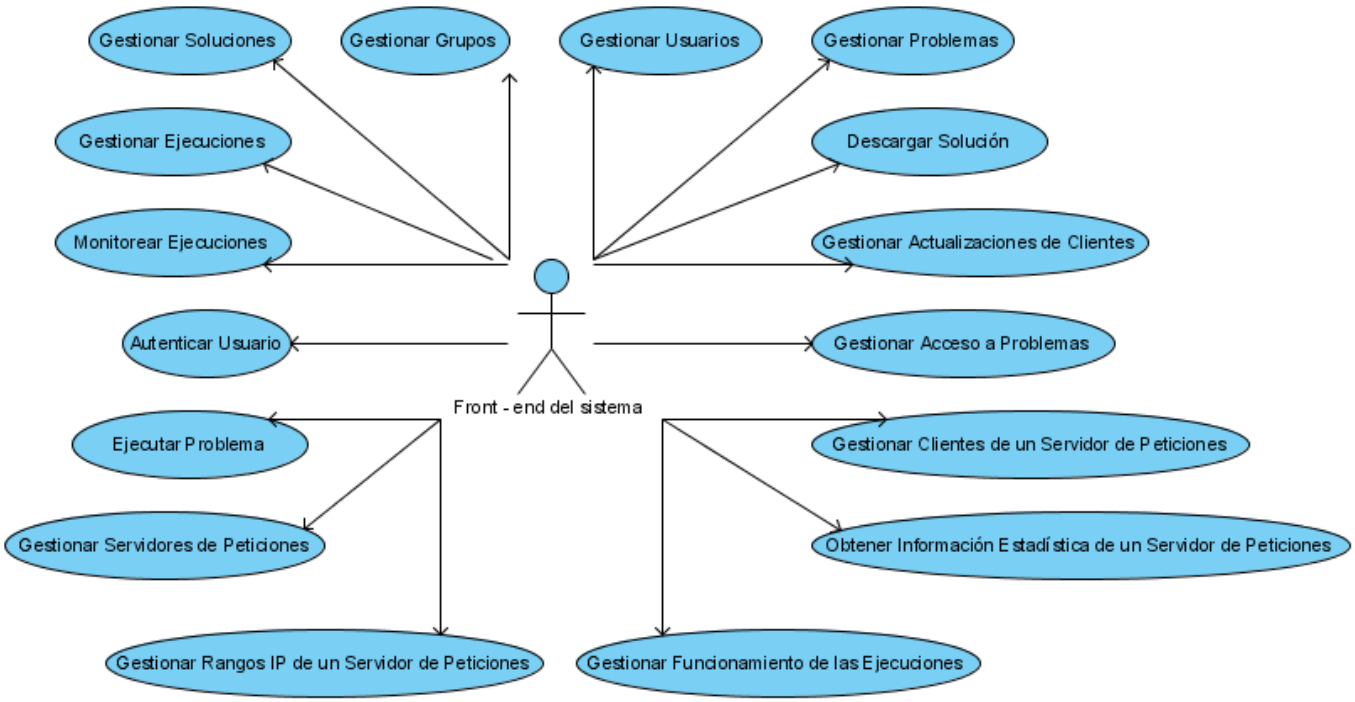

Figura 2.6: Diagrama de casos de usos del paquete *Actor front-end del sistema*.

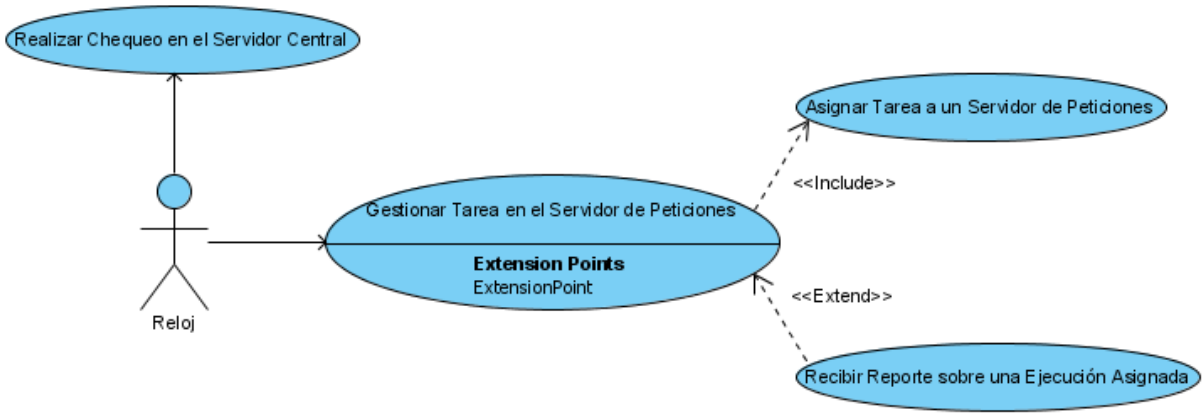

Figura 2.7: Diagrama de casos de usos del paquete *Actor reloj*.

2.4.3.1. Vista de casos de usos arquitectónicamentes significativos

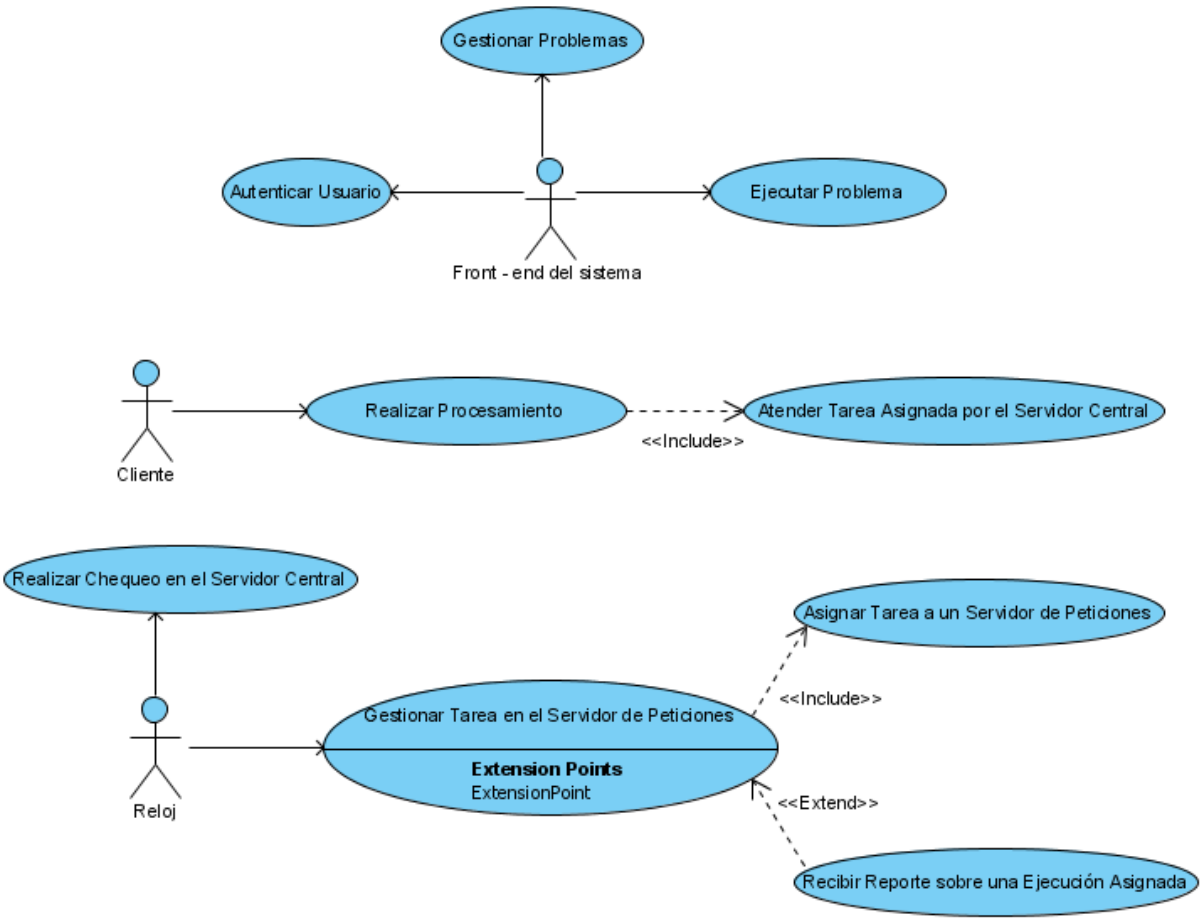

Figura 2.8: Diagrama de casos de usos arquitectónicamente significativos.

2.5. Conclusiones

En este capítulo se identificaron los requerimientos funcionales, los actores y casos de usos del sistema que sirvieron de base para realizar el diseño del producto que se presenta en este trabajo.

Capítulo 3

Diseño del Sistema

En este capítulo se describe la representación arquitectónica del sistema, haciendo énfasis en el estilo y el patrón de arquitectura utilizado, así como en los patrones de diseño más importantes. Se muestra además los diagramas de clases de los paquetes relevantes de cada subsistema, así como los diagramas de secuencias necesarios para comprender el funcionamiento del presente trabajo.

3.1. Representación arquitectónica

La misión principal de la arquitectura del sistema es mostrar una panorámica general de los entes y subsistemas que lo integran, explicando cada uno de estos en diferentes vistas. El modelo de representación arquitectónica está basado en el modelo de las 4 + 1 vistas, añadiéndole la vista de datos por la importancia que tiene para la comprensión del sistema. La vista de procesos no se describirá porque no se considera relevante la información que esta brinda. La vista de casos de uso se describió en el capítulo anterior y la vista de implementación se encuentra descrita en el siguiente capítulo.

3.1.1. Estilo de arquitectura

El estilo de arquitectura adoptado fue el *cliente – servidor* debido a la lógica distribuida del sistema (figura 3.1). Además de ser un modelo que provee usabilidad, flexibilidad, interoperabilidad y escalabilidad en las comunicaciones y que permite a los usuarios finales obtener acceso a la información en forma transparente, aún en entornos multiplataforma y de recursos heterogéneos.

En el modelo cliente - servidor, el cliente envía un mensaje solicitando un determinado servicio a un

servidor específico (hace una petición) y este envía uno o varios mensajes con la respuesta (provee el servicio) . En un sistema distribuido cada máquina puede cumplir el rol de servidor para algunas tareas y el rol de cliente para otras.

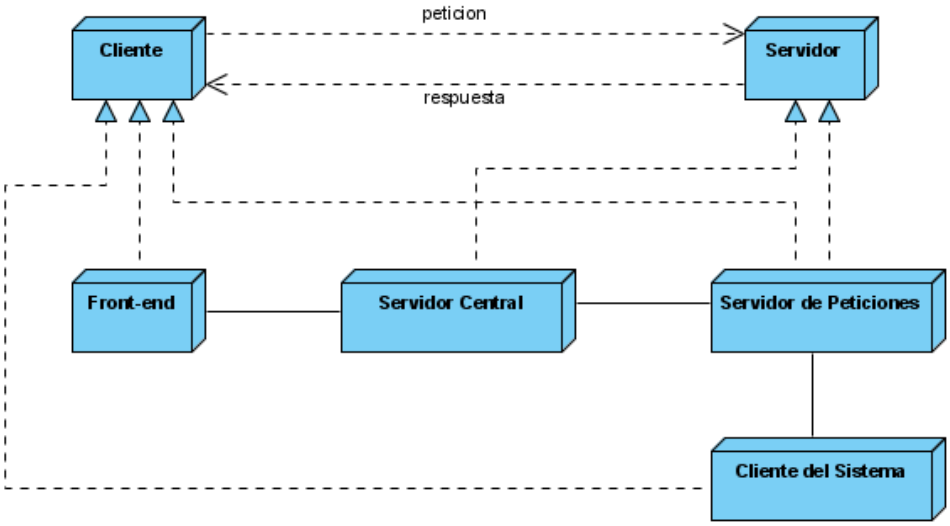

Figura 3.1: Estilo de arquitectura cliente - servidor aplicado al sistema.

Como se puede apreciar en la figura 3.1, el subsistema *Front-end* y el subsistema *Cliente del Sistema* sólo se comportan como clientes, ya que ambos solamente realizan solicitudes a los subsistemas *Servidor de Central* y *Servidor de Peticiones* respectivamente.

El subsistema *Servidor Central* siempre se comportará como servidor, en cambio el subsistema *Servidor de Peticiones* se comportará tanto de cliente como de servidor. Se comportará como cliente del *Servidor Central* para hacer peticiones de tareas, y como servidor del *Cliente del Sistema* ya que responde a las solicitudes de procesamiento que este último realiza.

3.1.2. Patrón de arquitectura

El patrón de arquitectura que sigue el sistema propuesto es Mediator [31], debido a la necesidad de que interactúen componentes independientes. Este patrón es aplicado cuando un conjunto de componentes necesitan comunicarse con el fin de trabajar en forma cooperativa; las vías de interacción aunque bien definidas se caracterizan por ser muy complejas; o bien se quieren utilizar componentes que tienen formas muy distintas de comunicación. Este patrón se caracteriza por:

- Un mediador (Mediator): que define la interfaz de comunicación entre los colegas, permitiendo la cooperación entre ellos para la realización de tareas. Este componente conoce y mantiene a todos los colegas.

- Colegas (Collegue): son aquellos que solo conocen a su mediador y es a través de este último, que únicamente pueden hacer los requerimientos.

En la figura 3.2 se muestra la interacción de los subsistemas teniendo en cuenta el patrón de arquitectura aplicado. Es importante destacar la dualidad de comportamientos que tiene el subsistema *Servidor de Peticiones* ya que es tanto mediador de las distintas entidades del *Cliente del Sistema* y al mismo tiempo colega del *Servidor Central*.

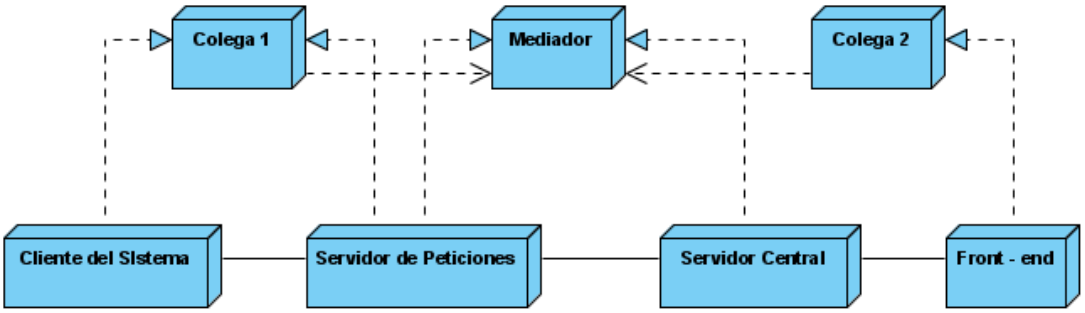

Figura 3.2: Patrón de arquitectura Mediator aplicado al sistema.

3.1.3. Patrones de diseño utilizados

Un patrón es una solución de un problema en determinado contexto [32], donde un contexto es una situación donde el patrón es aplicable y que generalmente es recurrente. A continuación se describen los patrones de diseños más importantes aplicados en el desarrollo del sistema:

3.1.3.1. Estrategia

- **Problema:** un servidor de peticiones puede atender diferentes ejecuciones de distintas naturaleza, por lo cual nunca podrá conocer a priori como particionarlas y/o procesar los resultados obtenidos a partir de cada unidad de trabajo creada.

- **Solución:** se diseña una clase abstracta *DataManager*, que define un conjunto de funciones y de la cual heredarán las clases que implementan la lógica de distribución de un problema específico. De esta

forma en el servidor de peticiones se carga dinámicamente la clase que describe ese comportamiento, obviando completamente la naturaleza del problema que atenderá.

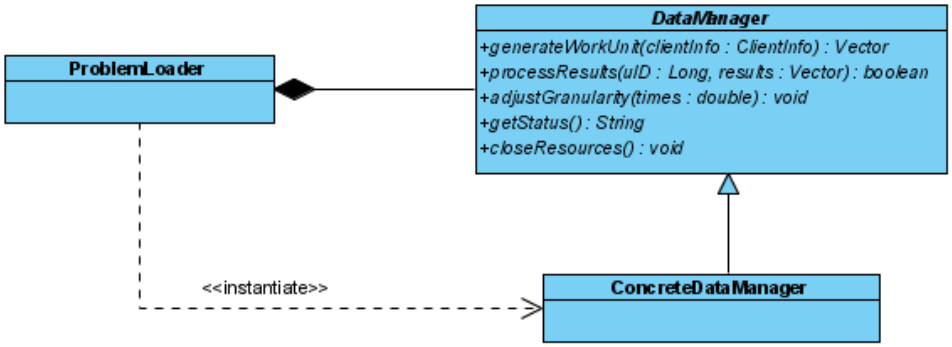

Figura 3.3: Ejemplo de la aplicación del patrón Estrategia en el sistema.

3.1.3.2. Singleton

- **Problema:** en el subsistema cliente se necesitaba que existiese una sola instancia de la clase *ConnectionManager* y que esta tuviese un único punto de acceso, de manera que varias clases pudiesen utilizar el mismo objeto.
- **Solución:** se diseña la clase *ConnectionManager* con una referencia a un objeto de ella misma y un método estático que devuelve dicho objeto.

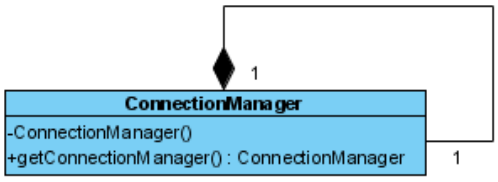

Figura 3.4: Ejemplo de la aplicación del patrón Singleton en el sistema.

3.1.3.3. Inyección de Dependencia

- **Problema:** se necesitaba lograr un bajo acoplamiento entre el sistema y el algoritmo de planificación.
- **Solución:** se definió una interfaz *SchedulerExecution* con todas las funcionalidades que debe tener el planificador de ejecuciones. La clase *Scheduler* tiene una referencia a un objeto de tipo *SchedulerExecution* pero no conoce hasta tiempo de ejecución quien será la clase concreta (*FIFOSchedulerExecution*)

que implementó dicha interfaz. Este diseño permite que en versiones futuras se pueda adicionar y cambiar dinámicamente algoritmos de planificación en el sistema.

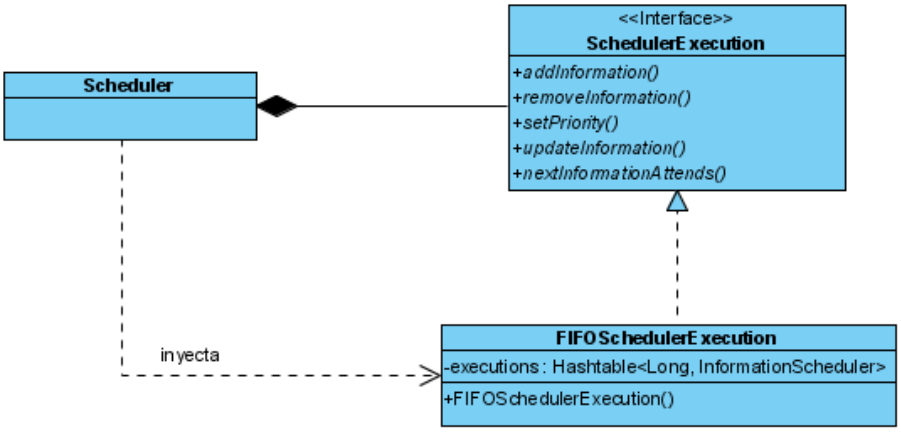

Figura 3.5: Ejemplo de la aplicación del patrón Inyección de Dependencia en el sistema.

3.1.3.4. Objeto de Acceso a Datos (DAO)

- **Problema:** se necesitaba abstraer el mecanismo de acceso a datos de la fuente de datos utilizada.
- **Solución:** se implementó por cada entidad persistente una clase DAO que es quien interactúa con la fuente de datos, y una clase Manager que es la que define las operaciones que se pueden hacer con esta entidad.

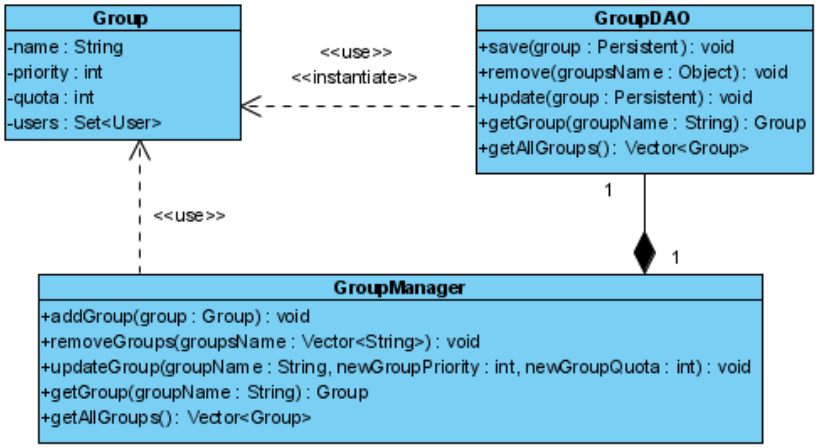

Figura 3.6: Ejemplo de la aplicación del patrón DAO en el sistema.

3.1.3.5. Factoría Abstracta

- **Problema:** se necesitaba independizar la creación de un objeto de tipo *ClientFileServer* de la clase que lo utiliza.
- **Solución:** se implementó una clase *ClientFileServerFactory* encargada de crear y retornar un objeto *ClientFileServer* según el tipo de comunicación especificada. Este diseño posibilita que en versiones futuras se puedan agregar e instanciar nuevas implementaciones de transferencia de archivos, sin necesidad de especificar la clase concreta que se utiliza en la entidad *Scheduler*.

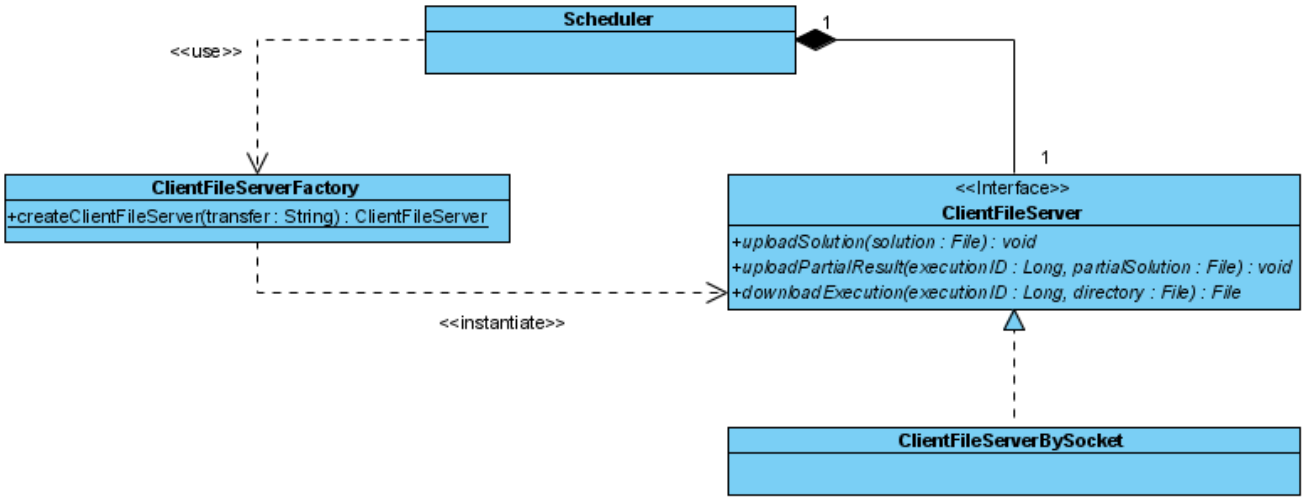

Figura 3.7: Ejemplo de la aplicación del patrón Factoría Abstracta en el sistema.

3.2. Vista lógica

En esta sección se describe las partes arquitectónicamente significativas del modelo de diseño, como son la descomposición en capas, subsistemas y paquetes.

La descripción de la vista lógica se realizará por módulos para un mejor entendimiento de las características de cada una de las partes. Se entiende por módulo cada uno de los subsistemas que componen el sistema. Se mostrará por cada módulo, los diagramas de paquetes de aquellos arquitectónicamentes significativos, teniendo en cuenta su relación y composición. Se ilustrará además, los diagramas de clases del diseño de los paquetes que se consideren necesarios, acompañados con una breve explicación de algunas de las clases o interfaces que lo componen.

3.2.1. Subsistemas

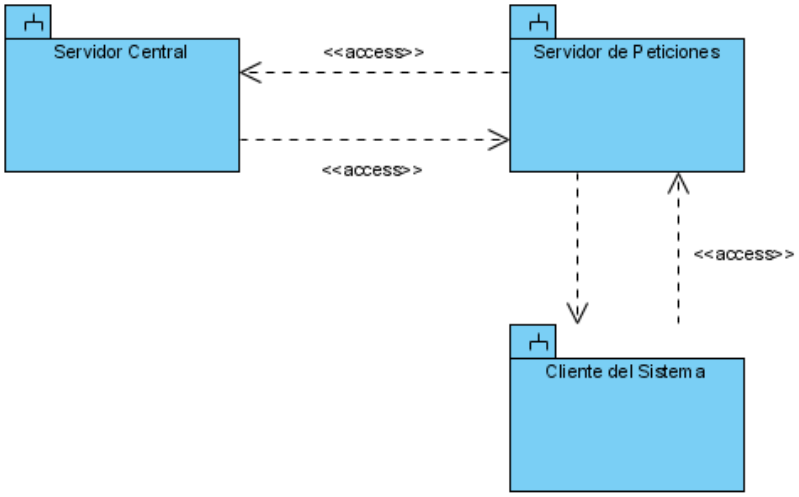

Figura 3.8: Vista general de los subsistemas y sus relaciones.

3.2.1.1. Servidor Central

Su principal función es la de chequear y planificar la asignación de las diferentes tareas a los servidores de peticiones. Además de controlar información sobre los usuarios que pueden acceder e interactuar con el sistema, así como, almacenar los problemas a partir de los cuales se crearán las diferentes ejecuciones a servir. Realiza también el seguimiento de todos los eventos que ocurren durante su funcionamiento.

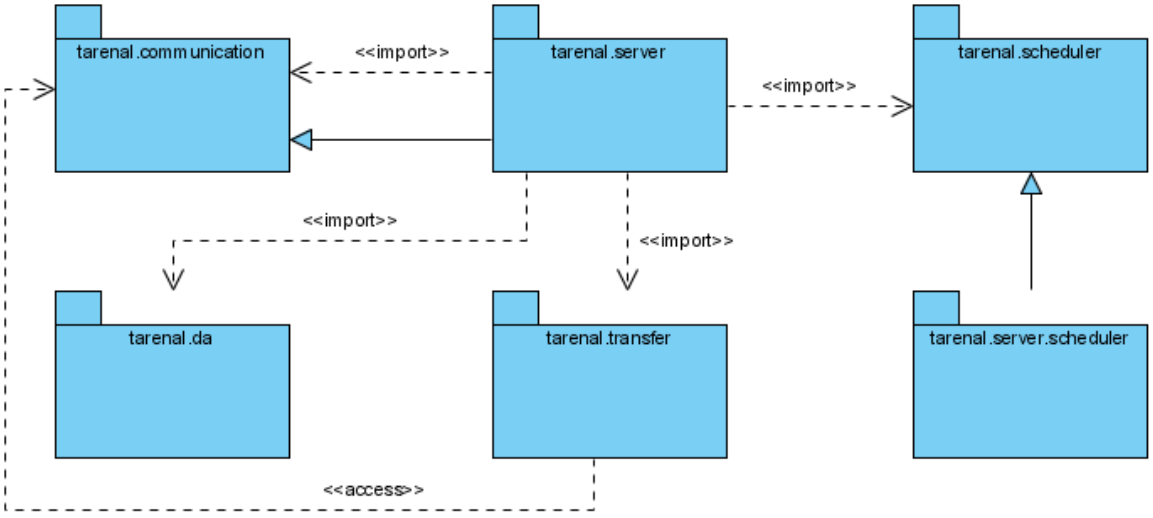

Figura 3.9: Paquetes relevantes para la arquitectura del Servidor Central. Vista de relación.

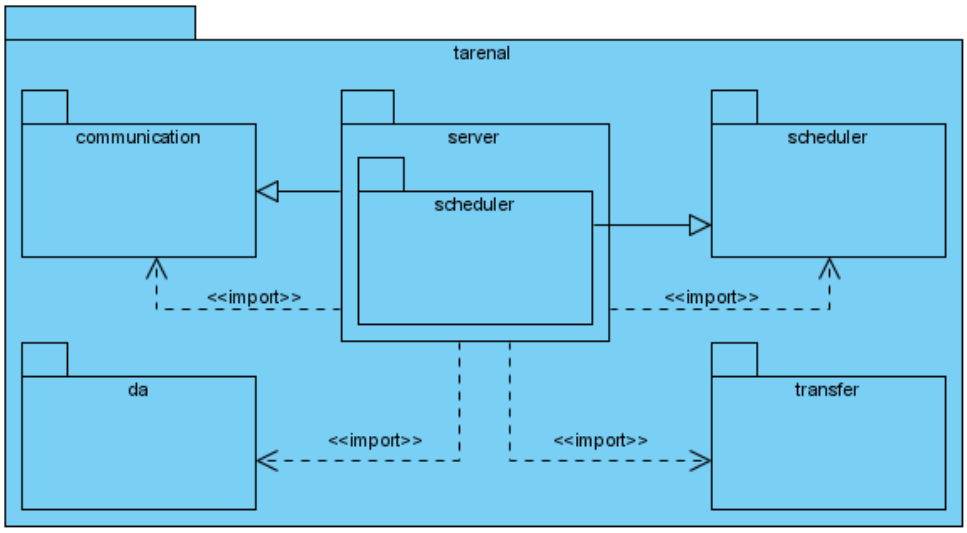

Figura 3.10: Paquetes relevantes para la arquitectura del Servidor Central. Vista de composición.

Descripción de los paquetes del Servidor Central

- **tarenal.communication:** define las interfaces de comunicación tanto para el front-end como para los servidores de peticiones.
- **tarenal.da:** define e implementa las clases de acceso a datos para la gestión de los grupos, usuarios, problemas, ejecuciones y soluciones en el sistema haciendo uso del patrón de diseño DAO.
- **tarenal.scheduler:** define la interfaz *SchedulerExecution* que provee las funcionalidades necesarias para el desarrollo de algoritmos de planificación de ejecuciones.
- **tarenal.server.scheduler:** define la clase que implementa la interfaz *SchedulerExecution* para la planificación de las ejecuciones existentes en el sistema. La clase implementada es *FIFOSchedulerExecution* y como su nombre lo indica, planifica y sirve las ejecuciones según el orden en que fueron creadas.
- **tarenal.transfer:** define e implementa las interfaces para la transferencia de archivos con los servidores de peticiones y el front-end.
- **tarenal.server:** implementa las interfaces de comunicación para que puedan realizarse todas las funcionalidades de administración e intercambio de información con los servidores de peticiones y el front-end.

Diagrama de clases del diseño del paquete *tarenal.communication* del Servidor Central

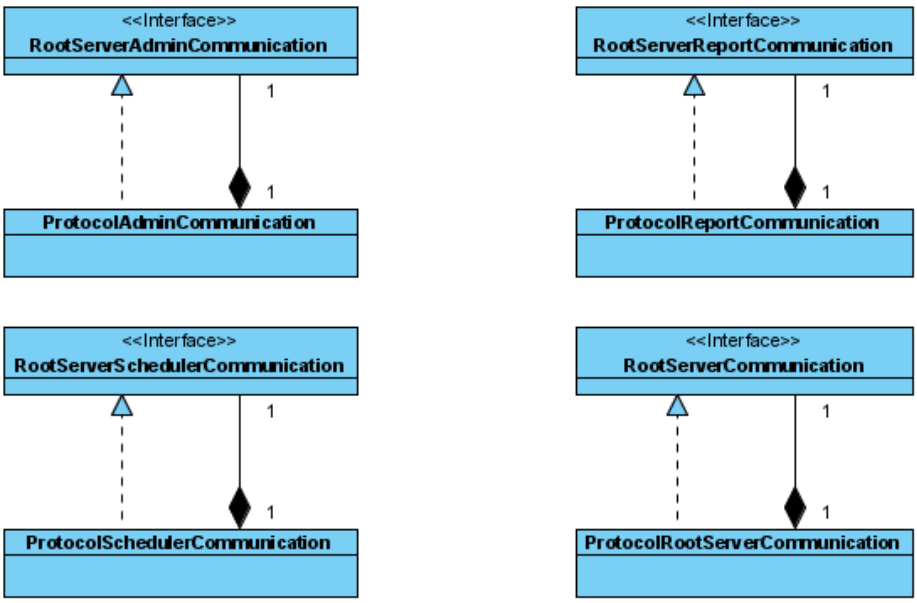

Figura 3.11: t-arenalRootServer. Diagrama de clases del diseño del paquete *tarenal.communication*.

- **RootServerCommunication:** esta interfaz es implementada por la clase cuya instancia será publicada para que los servidores de peticiones puedan intercambiar información con el servidor central del sistema.
- **RootServerAdminCommunication:** esta interfaz es implementada por la clase cuya instancia será utilizada para gestionar las diferentes entidades del sistema. Provee las funcionalidades para la gestión de los grupos, usuarios, rangos IP, clientes, problemas y soluciones. Permite además actualizar la versión de los clientes existentes.
- **RootServerReportCommunication:** esta interfaz es implementada por la clase cuya instancia será utilizada para acceder a los grupos, usuarios, rangos IP, clientes, problemas, ejecuciones, soluciones y a otras informaciones del sistema.
- **RootServerSchedulerCommunication:** esta interfaz es implementada por la clase cuya instancia será utilizada para crear, iniciar, monitorear y detener ejecuciones en el sistema.
- **ProtocolRootServerCommunication:** esta clase implementa la interfaz *RootServerCommunication* y tiene además una referencia a una instancia de dicha interfaz para realizar las funcionalidades

descritas en la misma. El objeto de esta clase es el que se publica para que los servidores de peticiones intercambien información con el servidor central.

- **ProtocolAdminCommunication:** esta clase implementa la interfaz *RootServerAdminCommunication* y tiene además una referencia a una instancia de dicha interfaz para realizar las funcionalidades descritas en la misma. El objeto de esta clase es el que se publica para la gestión remota de las entidades del sistema.

- **ProtocolReportCommunication:** esta clase implementa la interfaz *RootServerReportCommunication* y tiene además una referencia a una instancia de dicha interfaz para realizar las funcionalidades descritas en la misma. El objeto de esta clase es el que se publica para acceder a las informaciones del sistema.

- **ProtocolSchedulerCommunication:** esta clase implementa la interfaz *RootServerSchedulerCommunication* y tiene además una referencia a una instancia de dicha interfaz para realizar las funcionalidades descritas en la misma. El objeto de esta clase es el que se publica para gestionar remotamente las ejecuciones en el sistema.

Diagrama de clases del diseño del paquete *tarenal.transfer* del Servidor Central

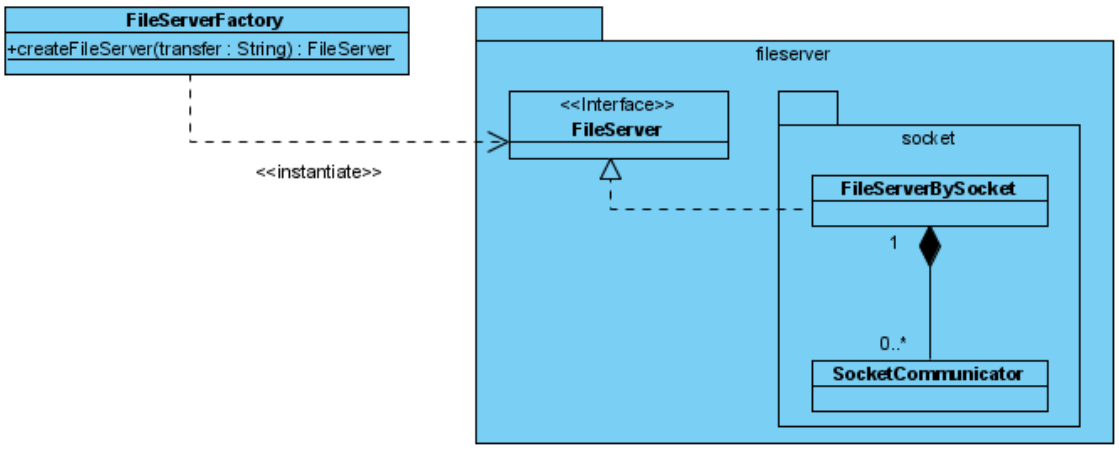

Figura 3.12: t-arenalRootServer. Diagrama de clases del diseño del paquete *tarenal.transfer*.

- **FileServerFactory:** esta clase brinda un método estático para obtener la instancia a través de la cual se gestionará la transferencia de archivos.

- **FileServer:** esta interfaz brinda los métodos para detener la descarga de una ejecución y obtener el puerto mediante el cual se realiza las transferencias de datos.
- **FileServerBySocket:** implementa la interfaz *FileServer*. El propósito de un objeto de esta clase es recibir conexiones por socket para la transferencia de archivos desde el front-end o los servidores de peticiones del sistema.
- **SocketCommunicator:** el propósito de un objeto de esta clase es realizar el envío o descarga de archivos desde el front-end o los servidores de peticiones del sistema.

Diagrama de clases del diseño del paquete *tarenal.server* del Servidor Central

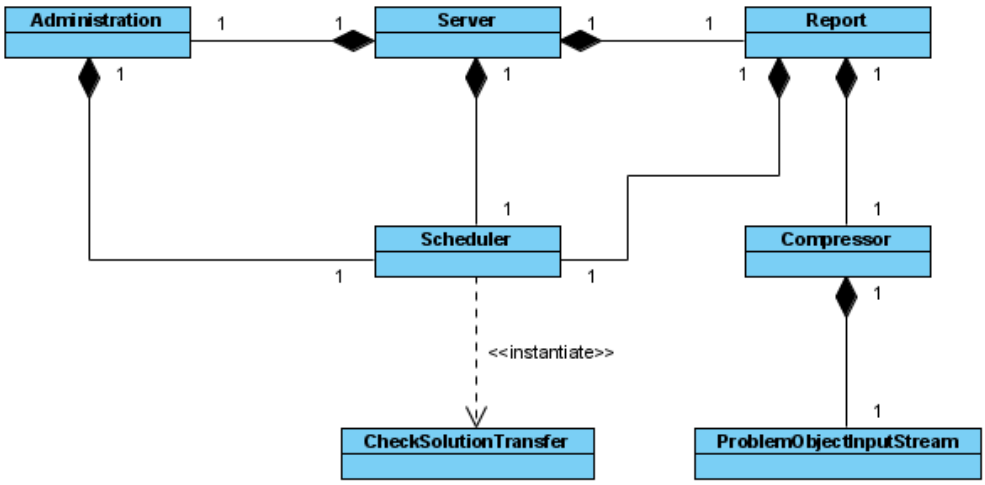

Figura 3.13: t-arenalRootServer. Diagrama de clases del diseño del paquete *tarenal.server*.

- **Server:** implementa las funcionalidades descritas en la interfaz *RootServerCommunication*. Una instancia de esta clase lee y parsea desde el archivo *server.ini* los parámetros con los que inicia el servidor central. Además de ser el objeto mediante el cual intercambia información los servidores de peticiones del sistema y al que hace referencia el objeto de tipo *ProtocolRootServerCommunication*.
- **Administration:** implementa las funcionalidades descritas en la interfaz *RootServerAdminCommunication* para la gestión de grupos, usuarios, rangos IP, clientes, problemas y soluciones en el sistema. A una instancia de esta entidad es a la que hace referencia el objeto de tipo *ProtocolAdminCommunication*.

- **Report:** implementa las funcionalidades descritas en la interfaz *RootServerReportCommunication.* A través de una instancia de esta clase se tiene acceso a las entidades e informaciones del sistema, y es a la que hace referencia el objeto de tipo *ProtocolReportCommunication.*
- **Scheduler:** implementa las funcionalidades descritas en la interfaz *RootServerSchedulerCommunication* para la gestión de las ejecuciones que existen en el sistema. Además, provee métodos para asignar las tareas a los servidores de peticiones. A una instancia de esta entidad es a la que hace referencia el objeto de tipo *ProtocolSchedulerCommunication.*

3.2.1.2. Servidor de Peticiones

Es el responsable de solicitar una tarea al servidor central, así como atender y controlar la realización de la misma. La tarea a ser atendida, será dividida en pequeñas subtareas denominadas unidades de trabajo.

El principal aspecto del servidor de peticiones es repartir las diferentes unidades de trabajos entre los clientes, siempre y cuando estos estén autorizados a interactuar con el mismo. Además de desarrollar una lógica de control de unidades pendientes de respuestas y otras que han expirado por algún error ocurrido.

El servidor de peticiones procesa el resultado de cada una de las subtareas generadas, para finalmente construir el resultado de la tarea original. Al igual que el servidor central, realiza también el seguimiento de todos los eventos que ocurren durante su funcionamiento.

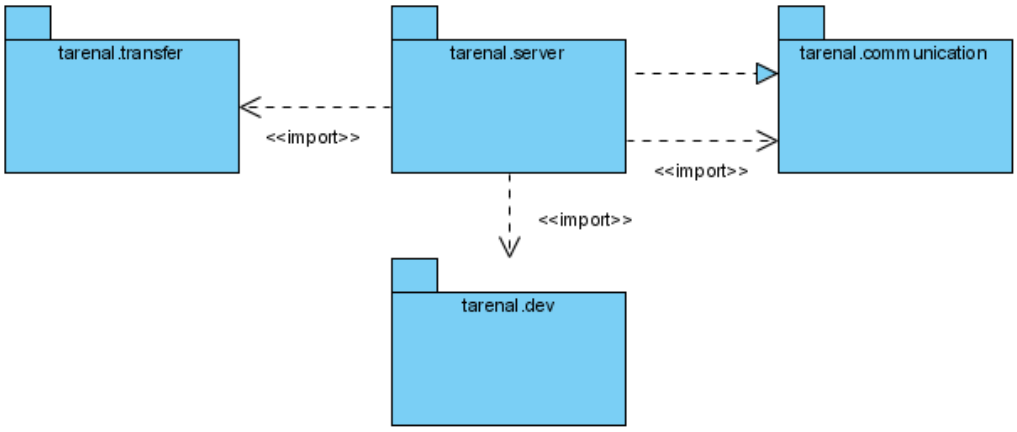

Figura 3.14: Paquetes relevantes para la arquitectura del Servidor de Peticiones. Vista de relación.

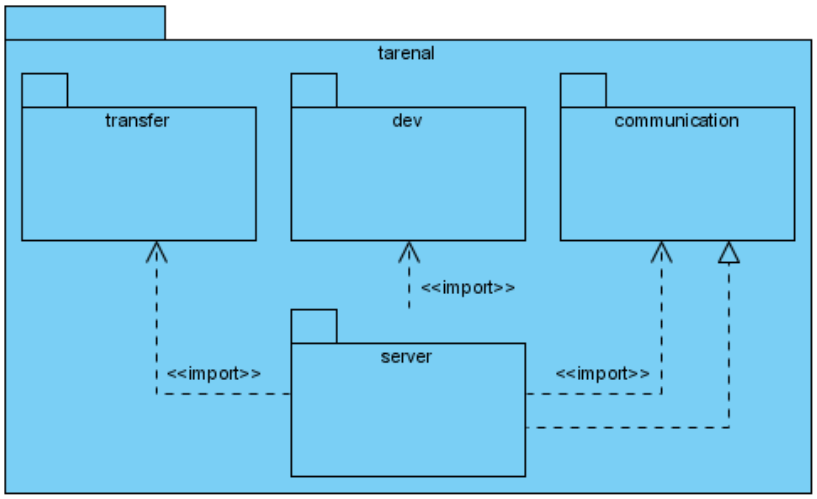

Figura 3.15: Paquetes relevantes para la arquitectura del Servidor de Peticiones. Vista de composición.

Descripción de los paquetes del Servidor de Peticiones

- **tarenal.communication:** define las interfaces de comunicación tanto para los clientes como para el servidor central.
- **tarenal.dev:** define las clases para el desarrollo de una aplicación distribuida.
- **tarenal.transfer:** define e implementa las interfaces para la transferencia de archivos.
- **tarenal.server:** implementa las interfaces de comunicación para que puedan realizarse todas las funcionalidades con el servidor central y los clientes. Implementa además, las funcionalidades relacionadas con el procesamiento de una ejecución.

Diagrama de clases del diseño del paquete *tarenal.communication* del Servidor de Peticiones

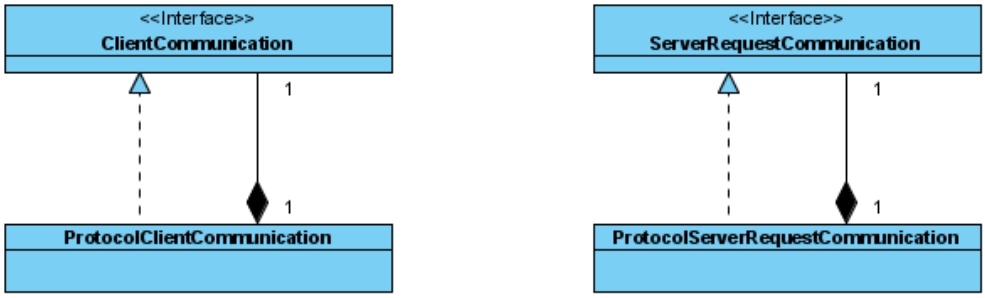

Figura 3.16: t-arenalServerRequest. Diagrama de clases del diseño del paquete *tarenal.communication*.

- **ClientCommunication:** esta interfaz es implementada por la clase cuya instancia será publicada para que los clientes interactúen con el servidor de peticiones correspondiente en la solicitud de unidades de trabajo.
- **ServerRequestCommunication:** esta interfaz es implementada por la clase cuya instancia será publicada para que el servidor central pueda intercambiar información con el o los servidores de peticiones del sistema.
- **ProtocolClientCommunication:** esta clase implementa la interfaz *ClientCommunication* y tiene además una referencia a una instancia de dicha interfaz para realizar las funcionalidades descritas en la misma. El objeto que se crea de esta clase es el que se publica para que los clientes interactúen con el sistema.
- **ProtocolServerRequestCommunication:** esta clase implementa la interfaz *ServerRequestCommunication* y tiene además una referencia a una instancia de dicha interfaz para realizar las funcionalidades descritas en la misma. El objeto que se crea de esta clase es el que se publica para que el servidor central interactúe con el o los servidores de peticiones.

Diagrama de clases del diseño del paquete *tarenal.transfer* del Servidor de Peticiones

Es importante aclarar que dentro de este paquete existe un diseño idéntico al mostrado en la figura 3.12. Sólo cambia la funcionalidad de dos clases: *FileServerBySocket* que únicamente recibe conexiones desde los clientes; y *SocketCommunicator* que solamente colabora en la transferencia de datos hacia los clientes.

En este subsistema se agregó otro diseño (figura 3.17), para la transferencia de datos hacia y desde el servidor central, abstrayéndose del protocolo a utilizar.

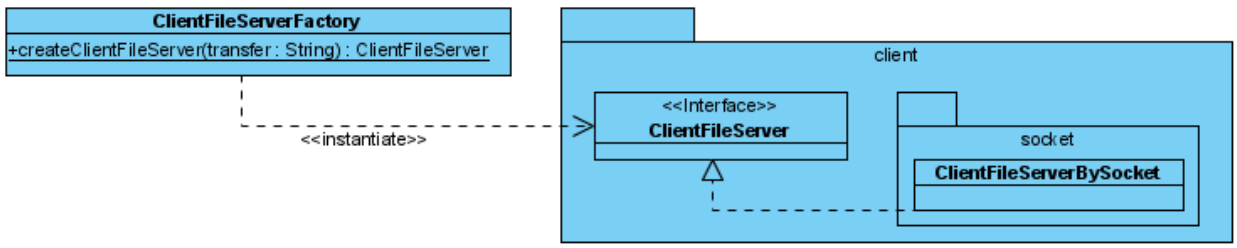

Figura 3.17: t-arenalServerRequest. Diagrama de clases del diseño del paquete *tarenal.transfer*.

- **ClientFileServerFactory:** esta clase brinda un método estático para obtener la instancia a través de la cual se gestionará la transferencia de archivos.
- **ClientFileServer:** esta interfaz brinda los métodos para descargar una ejecución y transferir hacia el servidor central la solución obtenida como resultado del procesamiento realizado.
- **ClientFileServerBySocket:** implementa la interfaz *ClientFileServer*. El propósito de un objeto de esta clase es realizar la transferencia de archivos hacia y desde el servidor central mediante socket.

Diagrama de clases del diseño del paquete *tarenal.server* del Servidor de Peticiones

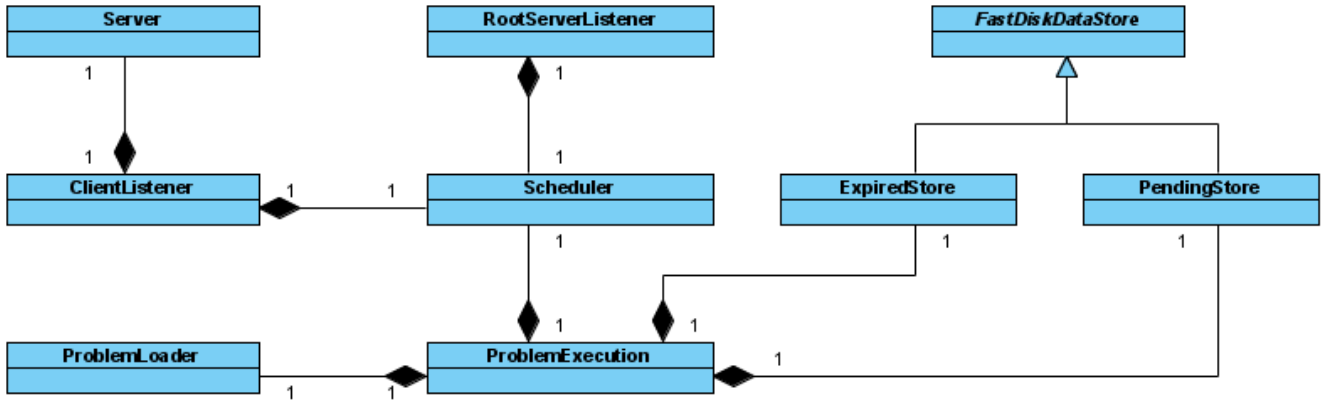

Figura 3.18: t-arenalServerRequest. Diagrama de clases del diseño del paquete *tarenal.server*.

- **Server:** una instancia de esta clase lee y parsea desde el archivo *server.ini* los parámetros con los que inicia el servidor de peticiones. Además de crear los objetos que serán publicados para el intercambio de información.
- **RootServerListener:** implementa las funcionalidades descritas en la interfaz *ServerRequestCommunication* para realizar las funcionalidades delegadas desde el servidor central. A una instancia de esta entidad es a la que hace referencia el objeto de tipo *ProtocolServerRequestCommunication*.
- **ClientListener:** implementa las funcionalidades descritas en la interfaz *ClientCommunication* para el intercambio de información con los clientes que lo soliciten. A una instancia de esta entidad es a la que hace referencia el objeto de tipo *ProtocolClientCommunication*.
- **Scheduler:** esta clase hereda de *Thread*. Una instancia de esta clase constantemente realiza, peticiones al servidor central para atender alguna tarea o controla el funcionamiento de alguna que halla sido

asignada.

- **ProblemExecution:** esta clase representa a una ejecución que está siendo atendida. Contiene información como el identificador, la prioridad, cantidad de unidades generadas, cantidad de resultados recibidos, los requerimientos computacionales de la ejecución que se atiende, entre otros.
- **ProblemLoader:** esta clase hereda de *URLClassLoader* y la instancia que se crea de esta entidad, es la responsable de cargar dinámicamente las librerías y el problema de la ejecución que será atendida.
- **PendingStore:** esta clase brinda las funcionalidades para gestionar las unidades de trabajo que se encuentran pendientes de respuestas y todavía no han expirado.
- **ExpiredStore:** esta clase brinda las funcionalidades para gestionar las unidades de trabajo que expiran por alguna causa.

3.2.1.3. Cliente del Sistema

El cliente es el subsistema que realiza una solicitud de una unidad de trabajo al servidor de peticiones correspondiente, hace el procesamiento de la misma y posteriormente retorna el resultado obtenido; y así sucesivamente. Múltiples clientes pueden realizar peticiones de trabajo al servidor.

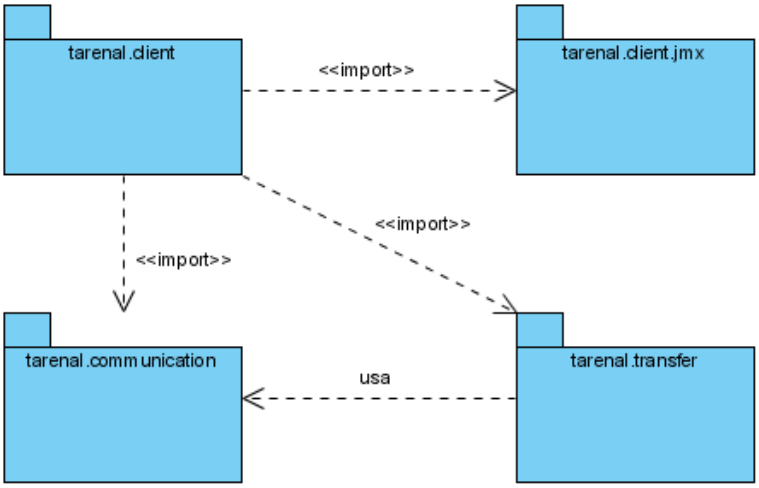

Figura 3.19: Paquetes relevantes para la arquitectura del Cliente. Vista de relación.

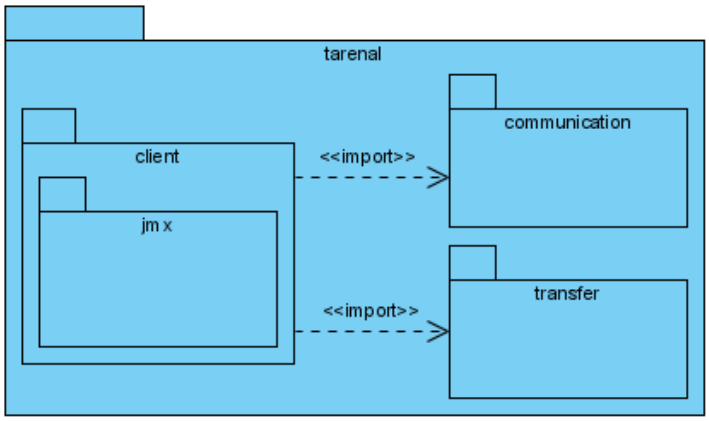

Figura 3.20: Paquetes relevantes para la arquitectura del Cliente. Vista de composición.

Descripción de los paquetes del Cliente

- **tarenal.communication:** se encuentran definidas e implementadas las entidades para gestionar la comunicación con el servidor de peticiones correspondiente.
- **tarenal.transfer:** provee las definiciones e implementaciones de clases e interfaces para la transferencia de archivos con el servidor de peticiones correspondiente.
- **tarenal.client:** implementa todas las funcionalidades para el procesamiento y devolución de los resultados de las unidades de trabajo asignadas.
- **tarenal.client.jmx:** define e implementa las interfaces de comunicación con el front-end del cliente.

Diagrama de clases del diseño del paquete *tarenal.client* del Cliente

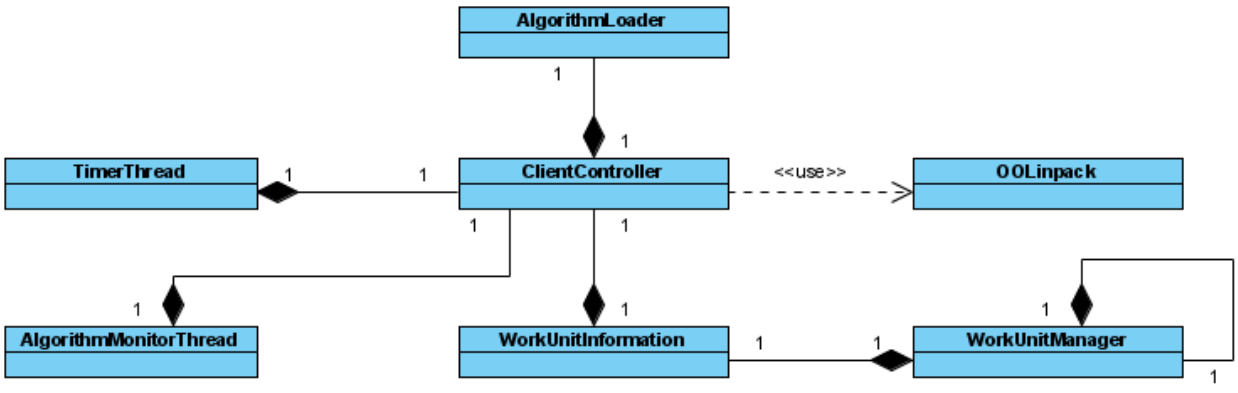

Figura 3.21: t-arenalClient. Diagrama de clases del diseño del paquete *tarenal.client*.

- **OOLinkpack:** esta clase es utilizada para calcular la potencia del procesador mediante el algoritmo Linkpack.
- **ClientController:** esta clase hereda de *Thread*. Su responsabilidad principal es gestionar todas las funcionalidades del cliente y guardar información sobre la unidad de trabajo que procesa, la información del cliente en general y determinada información de hardware.
- **AlgorithmLoader:** la responsabilidad de esta clase es cargar mediante introspección, la clase que procesará la unidad de trabajo asignada.
- **AlgorithmMonitorThread:** esta clase hereda de *Thread*, y es la encargada de monitorear el estado del procesamiento de una unidad de trabajo.
- **WorkUnitInformation:** esta clase contiene toda la información concerniente a la unidad de trabajo a procesar.
- **WorkUnitManager:** su responsabilidad principal es obtener la información de la unidad de trabajo a procesar, ya sea de la que quedó pendiente o de solicitar una nueva al servidor de peticiones correspondiente.

Diagrama de clases del diseño del paquete *tarenal.client.jmx* del Cliente

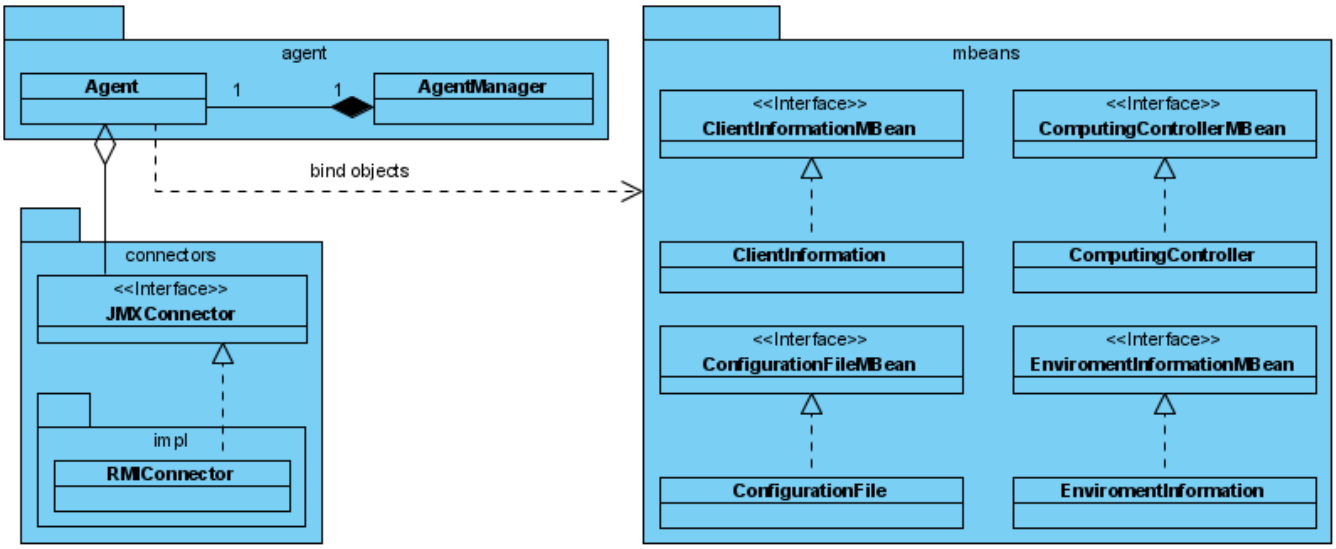

Figura 3.22: t-arenalClient. Diagrama de clases del diseño del paquete *tarenal.client.jmx*.

- **Agent:** esta clase encapsula lo referente al Agente de JMX. Su responsabilidad principal es registrar los Mbeans y adicionar e inicializar los conectores definidos.
- **AgentManager:** esta clase es la encargada de asegurar que se pueda utilizar el Agente desde varios puntos del sistema cliente.
- **JMXConnector:** esta interfaz define el comportamiento de un Conector de JMX, que es utilizado para poder acceder a los MBeans registrados por el Agente.
- **RMIConnector:** esta clase implementa la interfaz *JMXConnector* para que se puedan acceder a los MBeans registrados a través de RMI.
- **ClientInformationMBean:** esta interfaz define las funcionalidades de acceso a la información del cliente.
- **ComputingControllerMBean:** esta interfaz define las funcionalidades para configurar el tiempo en que el cliente puede solicitar unidades de trabajo para su procesamiento.
- **ConfigurationFileMBean:** esta interfaz define las funcionalidades para la configuración de los valores del cliente cuando este se inicia.
- **EnviromentInformationMBean:** esta interfaz define las funcionalidades para tener acceso a la información de hardware y sistema operativo en el cliente.
- **ClientInformation:** esta clase implementa la interfaz *ClientInformationMBean*, y la instancia creada es publicada para intercambiar información con el front-end del cliente.
- **ComputingController:** esta clase implementa la interfaz *ComputingControllerMBean*, y la instancia creada es publicada para intercambiar información con el front-end del cliente.
- **ConfigurationFile:** esta clase implementa la interfaz *ConfigurationFileMBean*, y la instancia creada es publicada para intercambiar información con el front-end del cliente.
- **EnviromentInformation:** esta clase implementa la interfaz *EnviromentInformationMBean*, y la instancia creada es publicada para intercambiar información con el front-end del cliente.

3.2.2. Diagramas de secuencia

Un diagrama de secuencia muestra las interacciones entre objetos ordenadas temporalmente. Ilustra los objetos que se encuentran en un escenario, y la secuencia de mensajes intercambiados entre ellos para llevar a cabo la funcionalidad descrita.

3.2.2.1. CU Gestionar Tarea en el Servidor de Peticiones

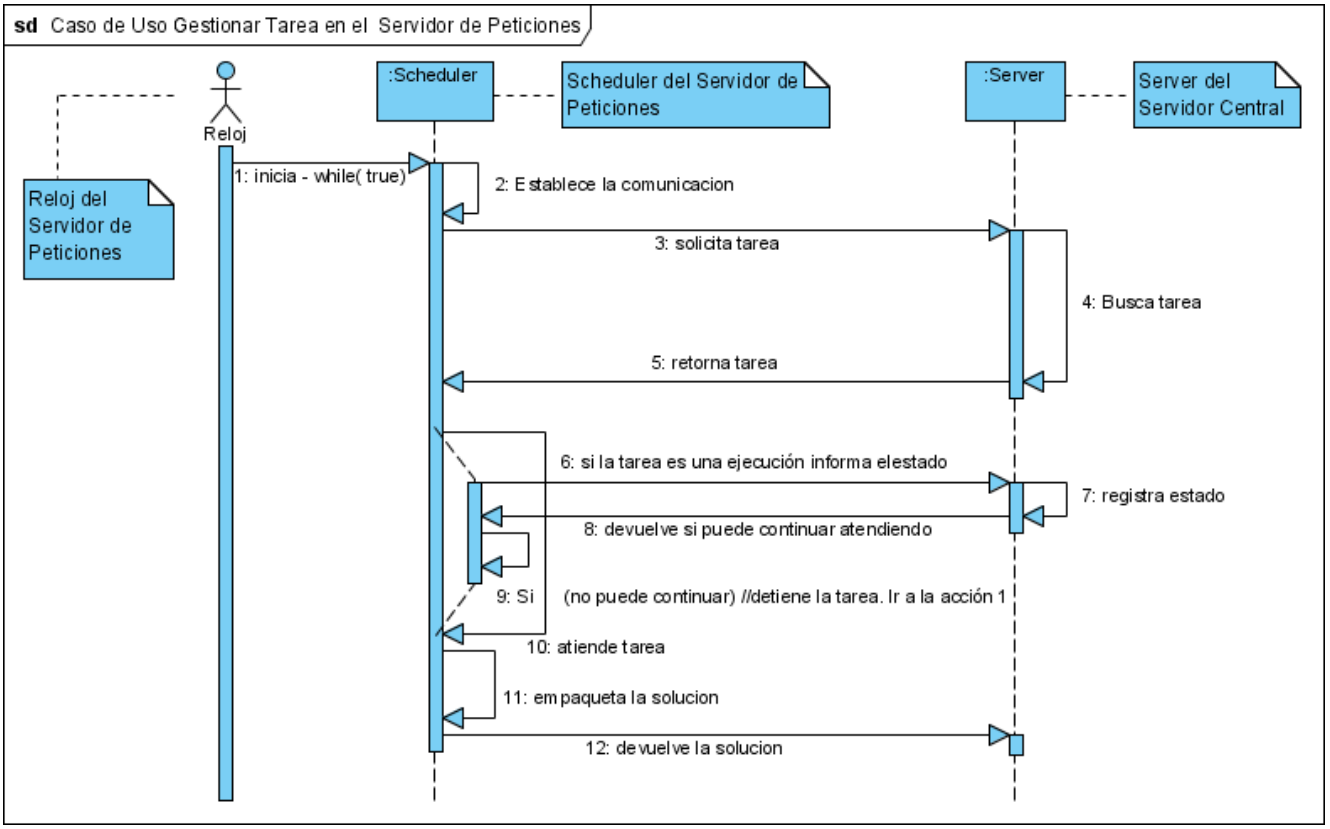

Figura 3.23: Diagrama de secuencia del CU Gestionar Tarea en el Servidor de Peticiones.

Es relevante destacar que la tarea que se envía para el servidor de peticiones puede ser una ejecución o la indicación de colaborar con otro servidor de peticiones. También es importante destacar que mientras el Scheduler del servidor de peticiones gestiona la tarea, concurrentemente y cada cierto tiempo informa al servidor central el estado de la misma. Esta acción solamente ocurre para los servidores de peticiones que están gestionando una ejecución y no para los que están colaborando.

3.2.2.2. CU Realizar Procesamiento

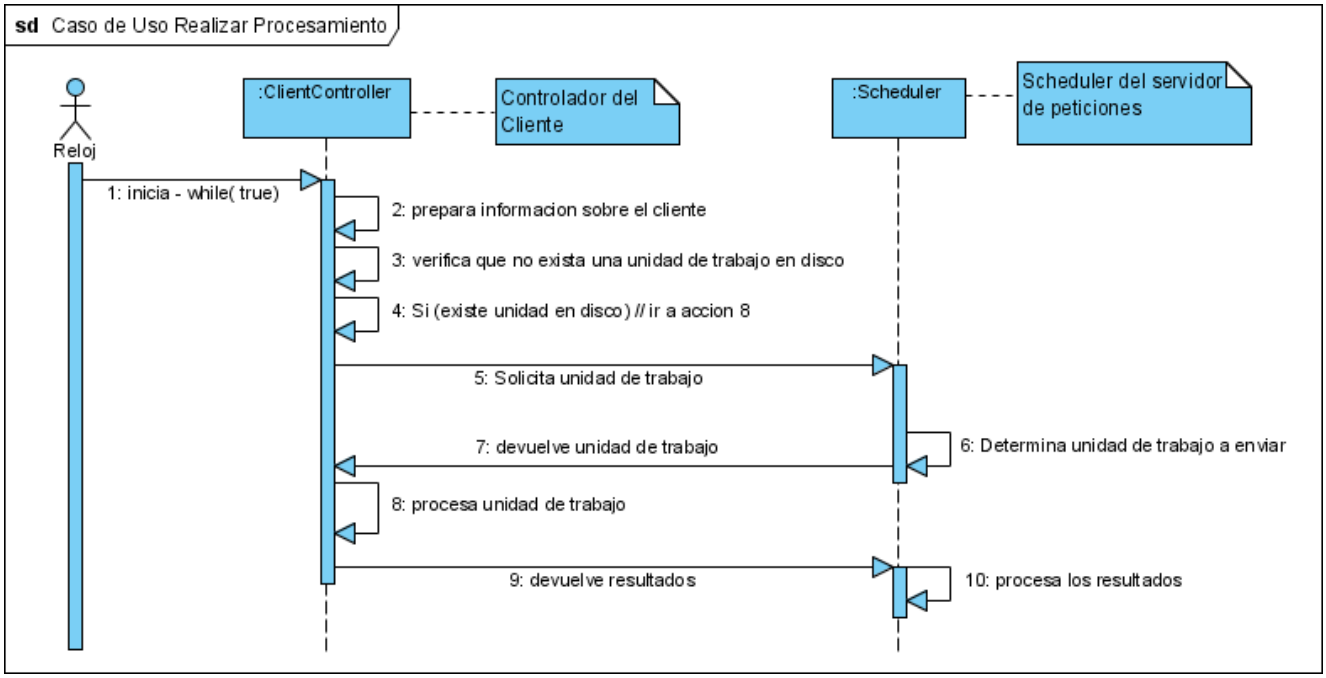

Figura 3.24: Diagrama de secuencia del CU Realizar Procesamiento.

El diagrama muestra la secuencia de acciones del controlador del cliente (*ClientController*) en conjunto con el resto del subsistema para procesar una unidad. Es importante destacar que esto se realiza mientras el proceso del cliente esté corriendo. Obsérvese también que cuando el cliente determina que existe una unidad de trabajo pendiente, le da prioridad a esta y no solicita una nueva.

3.3. Vista de datos

La base de datos no es centralizada. El servidor central solo controla información sobre los grupos, usuarios, problemas, ejecuciones y soluciones; mientras que los servidores de peticiones gestionan información sobre los rangos IP y clientes que le corresponden. Esto posibilita que cuando se atiende una tarea por parte de un servidor de peticiones, el mismo no necesita comunicarse con el servidor central para registrar la información de los clientes que realizan solicitudes, sino que es gestionada localmente.

3.3.1. Diagrama de clases persistentes

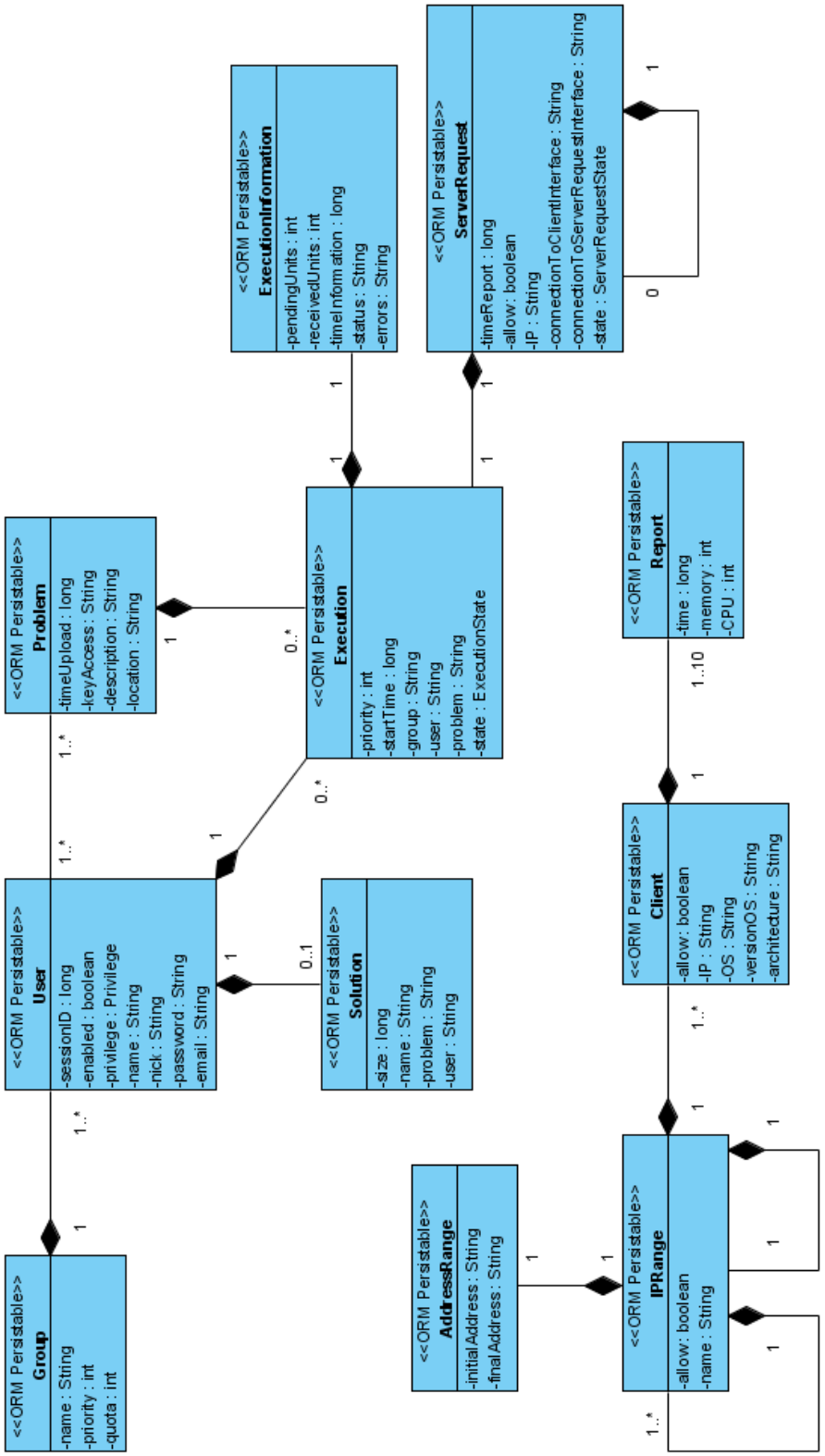

Figura 3.25: Diagrama de clases persistentes del sistema.

3.3.2. Descripción de las clases persistentes

- **Group:** representa a un grupo de usuarios del sistema. Esta clase contiene información sobre las características de un grupo, tales como: nombre, prioridad, cuota y los usuarios que pertenecen al mismo. El nombre es único para cada grupo que pertenece al sistema. Un grupo puede tener varios usuarios. La prioridad representa, la misma prioridad que tendrán las ejecuciones de los usuarios que pertenecen al grupo. La cuota, no es más que la cantidad de ejecuciones que puede tener como máximo un usuario del grupo.

- **User:** representa a un usuario del sistema. Esta clase contiene información sobre las características de un usuario, tales como: identificador de la sesión, nombre, nick, contraseña, dirección de correo electrónico y privilegio, así como los problemas, ejecuciones y soluciones que le pertenecen. El nick es único para cada usuario que pertenece al sistema. Un usuario solo puede ser miembro de un grupo, puede acceder a varios problemas y tener varias ejecuciones y soluciones.

- **Problem:** representa a un problema del sistema. Esta clase contiene información sobre las características de un problema, tales como: clave de acceso, descripción, usuarios y ejecuciones que le pertenecen. La clave de acceso es única para cada problema. Un problema puede ser accedido por muchos usuarios y un usuario puede acceder a varios problemas. Los problemas tendrán varias ejecuciones pero una ejecución solo pertenece a un problema.

- **Execution:** representa la ejecución de un problema que pertenece al sistema. Esta clase contiene información sobre una ejecución, tales como: prioridad, fecha de inicio, estado en el que se encuentra, así como el problema y el usuario al que pertenece.

- **ExecutionInformation:** representa la información de una ejecución en un instante de tiempo que es obtenida cuando es atendida. La información que almacena es la siguiente: cantidad de unidades pendientes, cantidad de unidades recibidas, fecha en que se realizó el reporte y los errores que han ocurrido.

- **Solution:** representa a una solución del sistema. Esta clase contiene información sobre las características de una solución, tales como: identificador, tamaño, así como el problema y el usuario al que pertenece.

- **IPRange:** representa a un rango IP del sistema. Los rangos de IP se utilizan para agrupar por

regiones a los clientes del sistema. De esta manera se tiene un mejor control de aquellos clientes que están autorizados o no a realizar peticiones de trabajo para su procesamiento. Esta clase contiene información sobre las características de un rango IP, tales como: nombre, si está permitido o no, rango de direcciones correspondiente (*AddressRange*), rango IP al que pertenece, así como los clientes y rangos IP que pertenecen a él. El rango de direcciones es único y no pueden existir rangos de direcciones que se solapen. Permitir un rango IP, no es más que autorizar a los clientes que pertenecen a él a interactuar con el sistema.

- **AddressRange:** representa a un intervalo de direcciones IP. Esta clase contiene información sobre las características de un rango de direcciones, estas son: dirección IP inicial y dirección IP final. El IP final nunca es menor que el IP inicial aunque sí pueden ser iguales.

- **Client:** representa a un cliente del sistema. Esta clase contiene información sobre las características de un cliente, tales como: dirección IP, arquitectura, tipo y versión del sistema operativo. Contiene además, los últimos diez reportes realizado por un cliente (ver clase Report). Los clientes pueden estar autorizados o no a interactuar con el sistema. Los que no están autorizados, no se les envían unidades de trabajo para su procesamiento. Para que un cliente sea permitido en el sistema, el rango IP (ver clase IPRange) al cual pertenece debe estar permitido también.

- **Report:** representa el reporte realizado por un cliente. Esta clase contiene la hora en que el cliente interactuó con el sistema, la cantidad de memoria y el poder de procesamiento que disponía en ese momento, para realizar el cómputo de la unidad de trabajo.

- **ServerRequest:** representa a un servidor de peticiones del sistema, contiene información sobre el momento exacto en que se reportó, si está permitido o no, la dirección IP, una referencia a la ejecución que está procesando en caso de que exista y la referencia a otro servidor de peticiones en caso que esté colaborando con él.

3.4. Vista de despliegue

Esta vista constituye la visión física del sistema, representa un grafo de nodos unidos por conexiones de comunicación.

3.4.1. Diagrama de despliegue

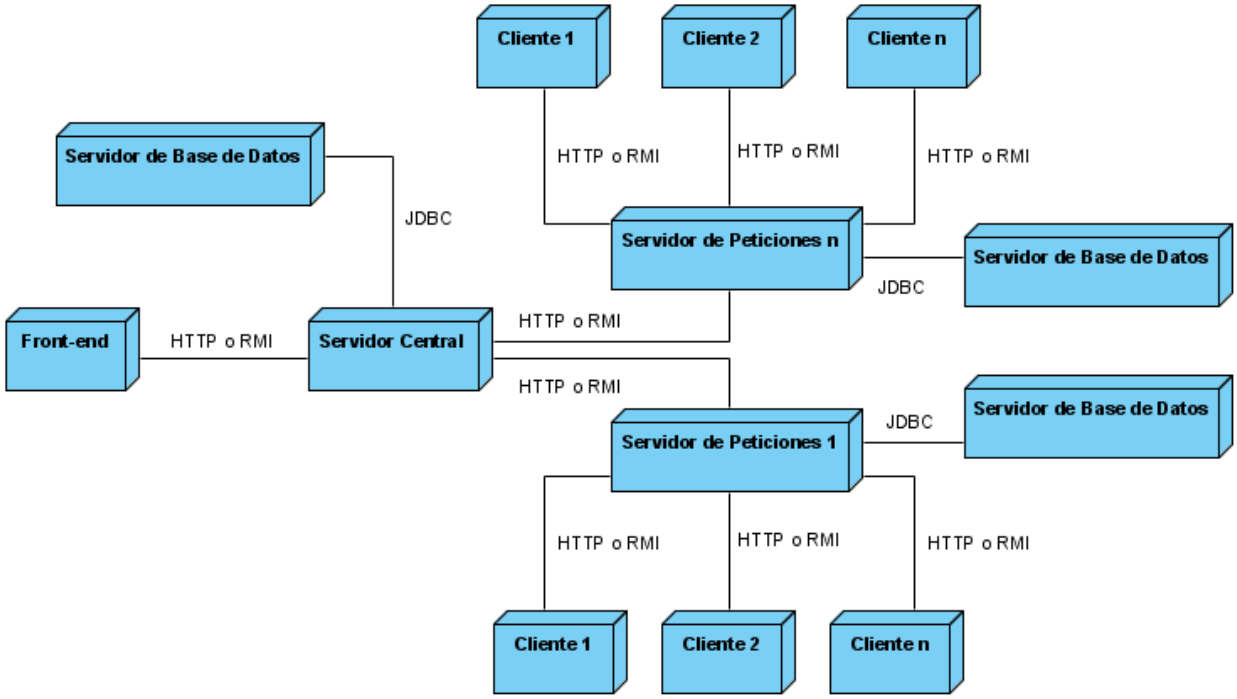

Figura 3.26: Diagrama de despliegue del sistema.

Descripción de los nodos

- **Nodo Front-end**: es el nodo donde una persona, software o componente de software externo al sistema interactúa con este.
- **Nodo Servidor Central:** es el nodo donde reside el *Servidor Central* del sistema.
- **Nodo Servidor de Peticiones**: es el nodo donde reside el *Servidor de Peticiones* del sistema.
- **Nodo Servidor de Base de Datos**: es el nodo donde reside el sistema de gestión de base de datos a utilizar por el *Servidor Central* o los *Servidores de Peticiones*.
- **Nodo Cliente:** es el nodo donde reside el *Cliente* del sistema.

Descripción de las comunicaciones

- **RMI o HTTP:** uno de estos protocolos será utilizado para realizar el intercambio de información entre los distintos módulos que integran el sistema siempre de forma transparente. Sólo el usuario que

interactúe desde el front-end debe especificar que tipo de comunicación se realizará con el *Servidor Central*.

- **JDBC:** este tipo de comunicación se establece entre los servidores y el sistema de gestión de base de datos correspondiente para la gestión de las entidades de las que son responsables.

3.5. Conclusiones

Como resultado de este capítulo se obtuvo la arquitectura del sistema sustentada en el estilo cliente - servidor. Se elaboró además el diseño del sistema desde la estructura de paquetes hasta los diagramas de clases por cada paquete, mostrando solamente los más significativos. También se muestra la vista de datos con una breve descripción de cada una de las entidades que la integran, así como la distribución física del sistema.

Capítulo 4

Implementación del Sistema

En este capítulo se describe la implementación del sistema en términos de componentes y la manera en que estos componentes serán desplegados. Además, se explica y se muestra el seudocódigo de los principales algoritmos implementados en cada módulo, así como los resultados de las pruebas realizadas.

4.1. Diagrama de componentes

El diagrama de implementación describe cómo los elementos del modelo de diseño se implementan en términos de componentes. Un diagrama de componentes muestra las organizaciones y dependencias lógicas entre componentes de software, sean estos de código fuente, binarios, archivos, bibliotecas cargadas dinámicamente o ejecutables.

En la figura 4.1 se muestran los componentes relevantes del sistema y las interfaces que estos implementan y acceden para mostrar su interacción. Existen tres componentes principales, *t-arenalRootServer*, *t-arenalServerRequest* y *t-arenalClient*. Cada uno de estos componentes estará desplegados en lugares potencialmente remotos como se muestra en la subsección 4.2.

El *t-arenalRootServer (servidor central)* es un componente ejecutable que implementa y brinda las interfaces *RootServerAdminCommunication*, *RootServerReportCommunication* y *RootServerSchedulerCommunication* para posibilitar la comunicación desde el front-end de la Plataforma. También implementa y publica la interfaz *RootServerCommunication* para la comunicación desde los servidores de peticiones.

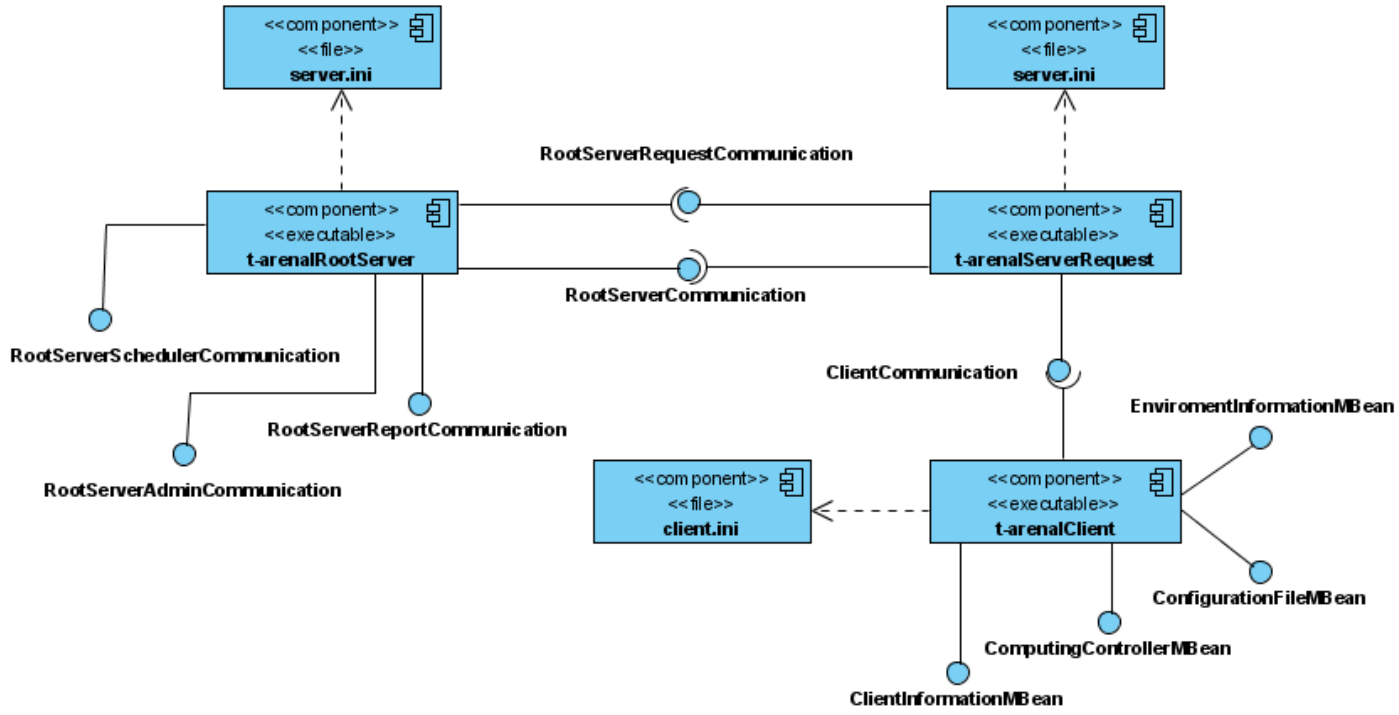

Figura 4.1: Diagrama de componentes del sistema.

El *t-arenalServerRequest (servidor de peticiones)* es también otro componente ejecutable que implementa y publica la interfaz *ClientCommunication,* para que el componente *t-arenalClient (cliente)* pueda acceder a las funcionalidades correspondientes a las solicitudes y retorno de los resultados de unidades de trabajo. Además implementa la interfaz *RequestServerCommunication* para permitir la comunicación desde el servidor central cuando este solicita realizar alguna acción.

El último componente es el *t-arenalClient*, mencionado con anterioridad, que implementa y publica cuatro interfaces: *ClientInformationMBean*, *ComputingControllerMBean, ConfigurationFileMBean* y *EnviromentInformationMBean* para la comunicación e intercambio de información con el front-end del cliente.

Cada componente tiene una dependencia con un archivo *.ini* de configuración. En el mismo se especifican las propiedades con las cuales cada componente iniciará su funcionamiento: conexión a la base de datos, tiempo de chequeo, tamaño máximo de los ficheros *.log*, configuración de la comunicación y de la transferencia de archivos, entre otros elementos.

4.2. Diagrama de despliegue de los componentes

En este diagrama se muestra la distribución de los componentes por los distintos nodos del despliegue, mostrados en la figura 3.26.

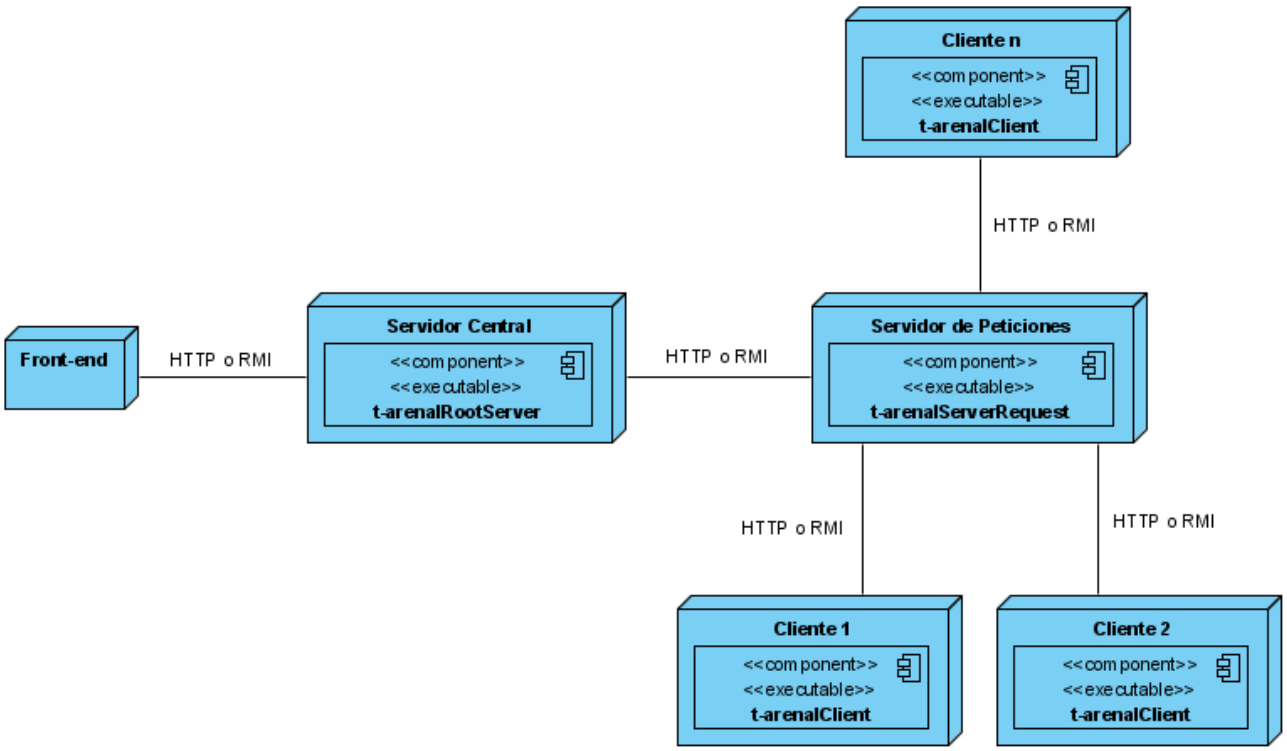

Figura 4.2: Diagrama de despliegue de los componentes.

4.3. Descripción de los algoritmos más importantes

En esta sección se describen los algoritmos implementados en cada uno de los subsistemas, para satisfacer las principales funcionalidades del presente trabajo.

4.3.1. Servidor Central

4.3.1.1. Algoritmo para asignar una tarea a un servidor de peticiones

Este algoritmo es implementado por la función *getWork()* de la clase *Scheduler*, que es invocada cuando un servidor de peticiones solicita una tarea. Los parámetros especificados son: el IP del servidor que realiza la petición, y el intervalo de tiempo en el que debe reportarse para notificar el estado de la tarea asignada

o pedir una nueva.

Primeramente se busca el servidor de peticiones con el IP especificado y se comprueba que existe en el sistema (4 - 5). Posteriormente se le actualiza el tiempo en el que debe reportarse (6) y se verifica que tenga permiso para realizar una tarea (7). Si tiene permiso, se busca la tarea y se le asigna (8 - 25), de lo contrario no se retorna nada. Recordar que la tarea puede ser una ejecución o colaborar con otro servidor de peticiones.

```
1  Entrada: ip del servidor, tiempo de reporte
2  Salida: la tarea a ser asignada
3
4  S := buscar el servidor de peticiones con el ip especificado como entrada
5  si S <> null entonces
6      actualizar el tiempo de reporte del servidor S
7      si S.permitido = cierto entonces
8          E := obtener ejecuciones en el orden en que serán atendidas
9          para i := 1 hasta n hacer
10             si S puede atender a E[i] y E[i] no está asignada entonces
11                 asignar ejecución E[i] al servidor S
12                 retornar ejecución E[i]
13             fin si
14         fin para
15
16         E := obtener todas las ejecuciones que están siendo atendidas
17         e := E[1]
18         para i := 2 hasta n hacer
19             si S puede atender a E[i] y E[i] está más atrasada que e entonces
20                 e := E[i]
21             fin si
22         fin para
23
24         establecer colaboración de S con el servidor que atiende a e
```

```
25              retornar e.servidor
26          fin si
27      fin si
28
29      retornar null
```

4.3.1.2. Algoritmo para guardar la información de una ejecución atendida por un servidor de peticiones

Este algoritmo es implementado por la función *notifyAttendsExecution()* de la clase *Scheduler*, que es invocada cuando un servidor de peticiones notifica el estado en que se encuentra una ejecución. Los parámetros especificados son: el IP del servidor que realiza la petición, el identificador y la información de la ejecución.

Primeramente se busca la ejecución con el identificador especificado y se comprueba que exista en el sistema (4 - 5). Posteriormente se obtiene el servidor de peticiones que la atiende y se comprueba que esté permitido (6 - 7). Si está permitido, se verifica que si no es el servidor que envía la notificación, entonces se actualiza la ejecución con el servidor que más adelantado esté en su atención (10 - 13). Finalmente, se actualiza la ejecución con la información enviada (19 - 20).

```
1   Entrada: ip del servidor (ipServidor), identificador de la ejecución, información de la ejecución (info)
2   Salida: cierto si el servidor puede seguir atendiendo la ejecución, de lo contrario falso
3
4   E := buscar ejecución con el identificador especificado como entrada
5   si E <> null entonces
6       S := obtener servidor de peticiones que atiende a E
7       si S.permitido = falso entonces
8           retornar falso
9       fin si
10      sino si ipServidor <> S.ip entonces
11          si info es mejor que E.info entonces
```

```
12                  cambiarle a E el servidor que la atiende por el que realiza el reporte
13              fin si
14              sino
15                  retornar falso
16              fin sino
17          fin sino si
18
19          actualizar información de E con la información enviada
20          retornar cierto
21  fin si
```

4.3.1.3. Algoritmo para verificar el estado de los servidores de peticiones con tareas asignadas

Este algoritmo es implementado por la función *verifyServersRequestResponsabilities()* de la clase *Scheduler*, y es invocada cada cierto tiempo para verificar el estado en que se encuentran los servidores de peticiones con la tarea asignada.

Primeramente se obtiene el tiempo actual del sistema (1). Luego se busca todos los servidores de peticiones existentes (3) y se verifica por cada uno que, si el lapso de tiempo transcurrido entre el último reporte y el tiempo actual del sistema, es mayor que el tiempo en el cual debería reportarse (5) entonces, si el servidor está trabajando se le quita la atención de la ejecución asignada (6 - 8); si el servidor está colaborando se le quita la responsabilidad de ayudar al servidor asignado (9 - 11).

```
1   T := obtener el tiempo actual del sistema
2
3   S := obtener todos los servidores de peticiones
4   para i := 1 hasta n hacer
5       si T − S[i].tiempoReporte > S[i].períodoReporte entonces
6           si S[i].trabajando = cierto entonces
7               quitar la atención de S[i] de la ejecución que tiene asignada
8           fin si
```

```
9           sino si S[i].colaborando = cierto entonces
10              quitar la colaboración de S[i] del servidor que tiene asignado
11          fin sino si
12      fin si
13  fin para
```

4.3.2. Servidor de peticiones

4.3.2.1. Algoritmo para calcular el tiempo de reporte al servidor central

Este algoritmo es implementado por la función *setSchedulerTimeout()* de la clase *Scheduler*, que es invocada por el servidor de peticiones antes de solicitar una tarea. Este tiempo es el pasado como parámetro a la función *getWork()* del servidor central (ver sección 4.3.1.1).

Para todos los clientes se determina cada que tiempo como promedio solicita una unidad de trabajo, tomando en cuenta las diez últimas peticiones realizadas (5 - 12). Finalmente se calcula, cada que tiempo como promedio todos los clientes analizados realizan las solicitudes, a partir de los promedios calculados a cada uno (14 - 17).

```
1   T := 0
2   P := tiempo promedio de reporte de cada cliente
3
4   C := obtener todos los clientes reportados
5   para i := 1 hasta n hacer
6       S := 0
7       R := obtener reportes del cliente C[i]
8       para j := 2 hasta m hacer
9           S := S + ( R[j] − R[j − 1] )
10      fin para
11      P[i] := S / ( m − 1 )
12  fin para
13
```

```
14 para i := 1 hasta n hacer
15     T := T + P[i]
16 fin para
17 T := T / n
```

4.3.2.2. Algoritmo para crear una unidad de trabajo para un cliente

Este algoritmo es implementado por la función *getDataUnit()* de la clase *ProblemExecution*, que es invocada cuando un elemento de cómputo realiza una solicitud de procesamiento. El parámetro especificado no es más que las informaciones y características del cliente que hace la petición.

Primeramente se verifica que existan unidades expiradas (4). Si existen entonces se busca una unidad acorde al cliente que realiza la solicitud (5 - 12), de lo contrario y si la ejecución no está pausada, se genera una nueva unidad y se le asigna (15 - 19).

```
1  Entrada: información del cliente que realiza la petición
2  Salida: la unidad de trabajo creada
3
4  si existen unidades expiradas entonces
5      obtener unidad expirada acorde a las características del cliente
6      si existe una unidad entonces
7          si el cliente puede procesar la unidad entonces
8              eliminar la unidad del almacén de unidades expiradas
9              adicionar la unidad al almacén de unidades pendientes
10             retornar unidad
11         fin si
12     fin si
13 fin si
14
15 si la ejecución no está en pausa entonces
16     generar nueva unidad de trabajo
```

```
17        adicionar la unidad creada al almacén de unidades pendientes
18        retornar unidad
19    fin si
20
21    retornar null
```

4.3.3. Pruebas al sistema

Para realizar las pruebas del presente trabajo se utilizó la *prueba de camino básico* [33] que es una técnica de prueba de caja blanca. Este método permite obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución, que permite realizar por lo menos una vez cada sentencia del programa.

Esta técnica fue aplicada en cuatro métodos del sistema, por cada uno se muestra el grafo de flujo correspondiente, la complejidad ciclomática, los caminos de pruebas, los casos de pruebas, el código que se utilizó para definir las pruebas haciendo uso de la librería JUnit y los resultados obtenidos. Con estas pruebas se validaron las principales funcionalidades que se realizan entre los subsistemas que forman parte de este trabajo, haciendo uso de las interfaces brindadas por cada uno ellos para su interacción.

4.3.3.1. Prueba del camino básico al método *login* de la clase *UserManager* del módulo *t-arenalRootServer*

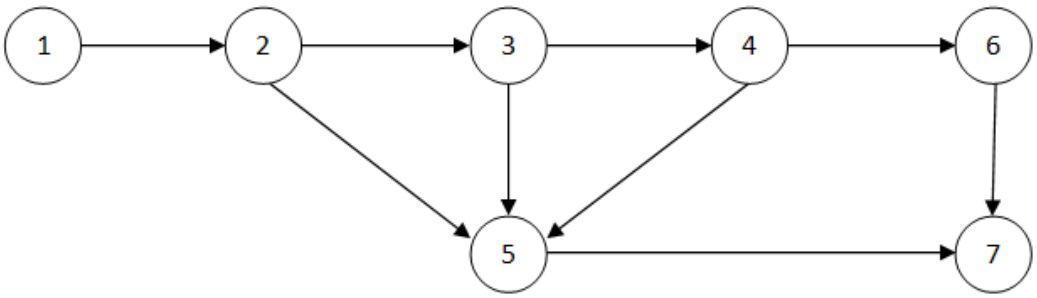

Figura 4.3: Grafo de flujo del método *login*.

Cálculo de la complejidad ciclomática

V(G) = cantidad de regiones = 4

V(G) = cantidad de aristas - cantidad de nodos + 2 = 9 - 7 + 2 = 4

V(G) = cantidad de nodos predicados + 1 = 3 + 1 = 4

Caminos de pruebas

Camino 1: 1 - 2 - 5 - 7

Camino 2: 1 - 2 - 3 - 5 - 7

Camino 3: 1 - 2 - 3 - 4 - 5 - 7

Camino 4: 1 - 2 - 3 - 4 - 6 - 7

Casos de prueba

Camino	Descripción	Entrada	Salida
1	Este camino se realiza cuando el usuario a autenticarse no existe.	**usuario:** cesar **contraseña:** cesar	Lanza una excepción de tipo *LoginException* con el mensaje *"Nick or password are incorrect or the account is not enabled"*
2	Este camino se realiza cuando la cuenta del usuario a autenticarse no está habilitada.	**usuario:** cesar **contraseña:** administrator	Lanza una excepción de tipo *LoginException* con el mensaje *"Nick or password are incorrect or the account is not enabled"*
3	Este camino se realiza cuando la contraseña del usuario a autenticarse es incorrecta.	**usuario:** root **contraseña:** root	Lanza una excepción de tipo *LoginException* con el mensaje *"Nick or password are incorrect or the account is not enabled"*
4	Este camino se realiza cuando el usuario se autentica satisfactoriamente.	**usuario:** root **contraseña:** administrator	Retorna el objeto *Login* con los datos del usuario autenticado.

Código para realizar las pruebas de los caminos básicos con el JUnit

```
1  public final void testLogin()
2  {
3      String user = "", pass = ""; // los valores están en dependencia del caso de prueba
4      Server.main( null );
5      try
6      {
7          assertTrue( Server.getInstanceAdministration().login( user, covertToMD5( pass ) ) != null );
8      }
9      catch ( LoginException e )
10     {
11         String mess = "Nick or password are incorrect or the account is not enabled";
12         assertTrue( e.getMessage().equals( mess ) );
13     }
14 }
```

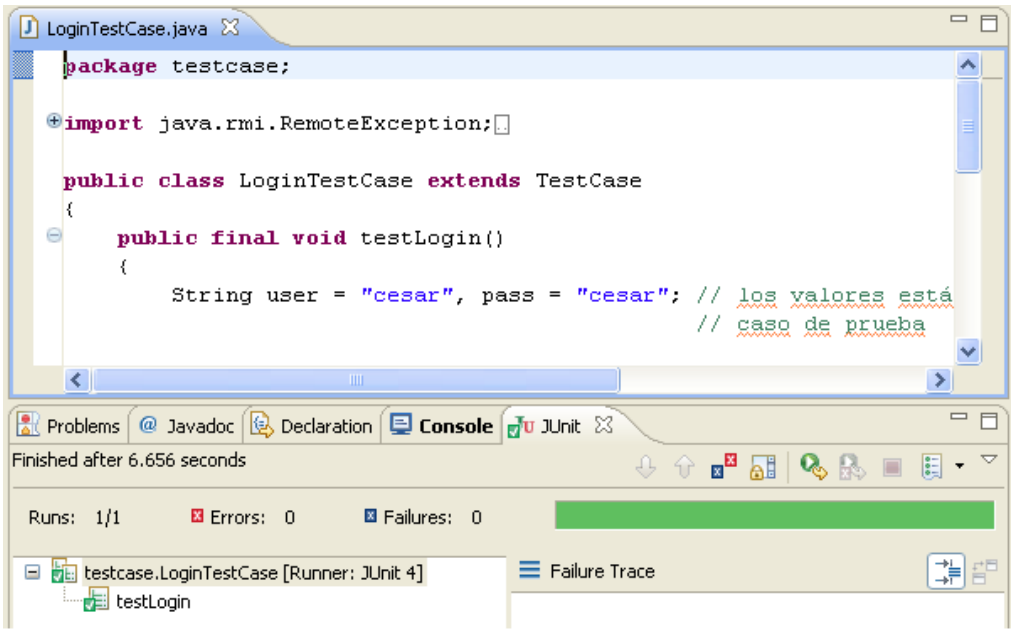

Figura 4.4: Resultado de la prueba para el camino 1 del método *login*.

4.3.3.2. Prueba del camino básico al método *getWork* de la clase *Scheduler* del módulo *t-arenalRootServer*

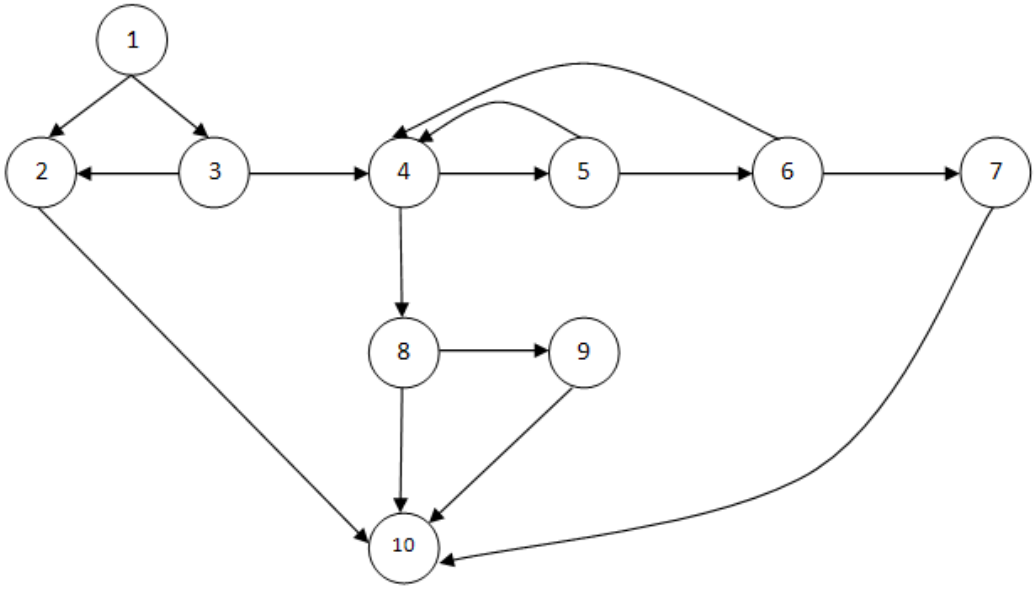

Figura 4.5: Grafo de flujo del método *getWork*.

Cálculo de la complejidad ciclomática

V(G) = cantidad de regiones = 7

V(G) = cantidad de aristas - cantidad de nodos + 2 = 15 - 10 + 2 = 7

V(G) = cantidad de nodos predicados + 1 = 6 + 1 = 7

Caminos de pruebas

Camino 1: 1 - 2 - 10

Camino 2: 1 - 3 - 2 - 10

Camino 3: 1 - 3 - 4 - 5 - 4 - ...

Camino 4: 1 - 3 - 4 - 5 - 6 - 4 - ...

Camino 5: 1 - 3 - 4 - 5 - 6 - 7 - 10

Camino 6: 1 - 3 - 4 - 8 - 10

Camino 7: 1 - 3 - 4 - 8 - 9 - 10

Casos de prueba

Camino	Descripción	Entrada	Salida
1	Este camino se realiza cuando el servidor de peticiones no existe.	**serverIP:** 10.7.19.51 **periodReport:** 60000	Retorna *null*, que indica que no se asignó una tarea.
2	Este camino se realiza cuando el servidor de peticiones existe pero no está permitido.	**serverIP:** 10.7.19.51 **periodReport:** 60000	Retorna *null*, que indica que no se asignó una tarea.
3	Este camino se realiza cuando una ejecución ya está asignada a un servidor de peticiones.	**serverIP:** 10.7.19.51 **periodReport:** 60000	No retorna nada, continúa analizando el resto de las ejecuciones.
4	Este camino se realiza cuando el servidor de peticiones no puede atender una ejecución.	**serverIP:** 10.7.19.51 **periodReport:** 60000	No retorna nada, continúa analizando el resto de las ejecuciones.
5	Este camino se realiza cuando una ejecución le es asignada al servidor de peticiones.	**serverIP:** 10.7.19.51 **periodReport:** 60000	Retorna un *java.util.Vector* con la ejecución que debe atender.
6	Este camino se realiza cuando no existe una tarea que asignar al servidor de peticiones.	**serverIP:** 10.7.19.51 **periodReport:** 60000	Retorna *null*, que indica que no se asignó una tarea.
7	Este camino se realiza cuando al servidor de peticiones se le asigna colaborar con otro servidor.	**serverIP:** 10.7.19.51 **periodReport:** 60000	Retorna un *java.util.Vector* con el servidor de peticiones con el cual debe colaborar.

Código para realizar las pruebas de los caminos básicos con el JUnit

```
1  public final void testGetWork()
2  {
3      try
4      {
5          RootServerCommunication server = (RootServerCommunication) Naming.lookup( url );
6
7          boolean test = false;
8          Vector result = server.getWork( 60000l );
9          if ( result == null ) // si no existía alguna tarea que asignar
10         {
11             test = true;
12         }
13         else // si se asignó una tarea
14         {
15             boolean execution = result.get(0) instanceof Execution;
16             boolean serverReq = result.get(0) instanceof ServerRequest;
17             if ( execution || serverReq )
18             {
19                 test = false;
20             }
21         }
22
23         assertTrue( test );
24     }
25     catch ( RemoteException e )
26     {
27         assert( true ); // no se estableció la comunicación con el servidor
28     }
29 }
```

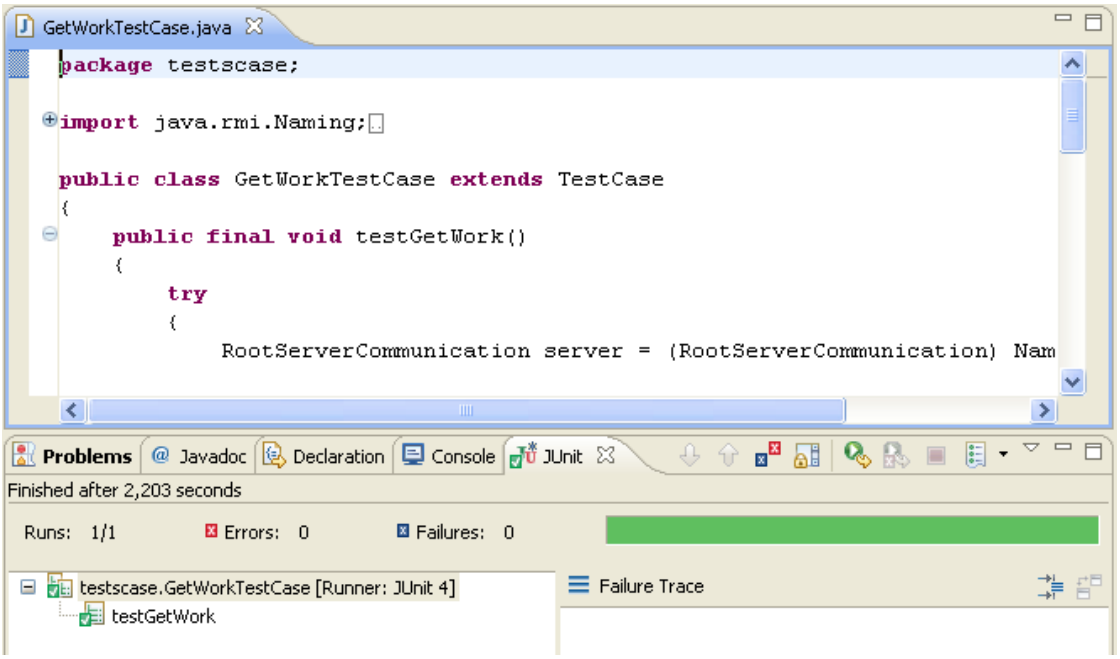

Figura 4.6: Resultado de la prueba para el camino 1 del método *getWork*.

4.3.3.3. Prueba del camino básico al método *generateDataUnit* de la clase *Scheduler* del módulo *t-arenalServerRequest*

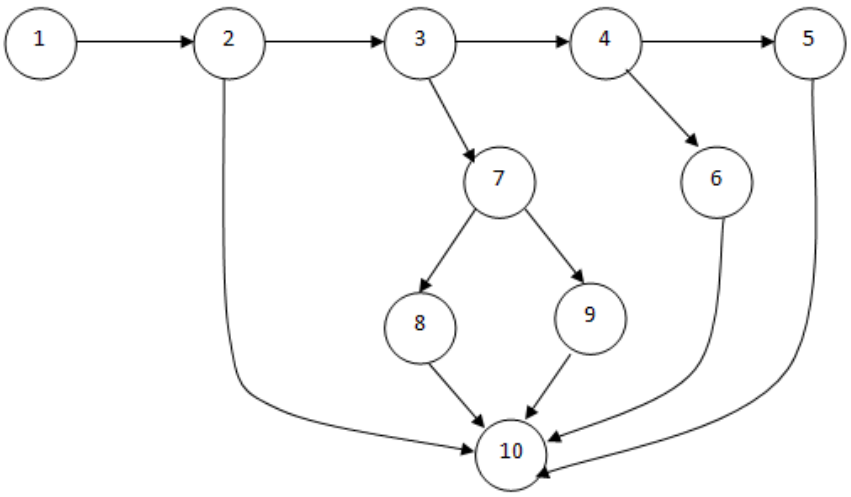

Figura 4.7: Grafo de flujo del método *generateDataUnit*.

Cálculo de la complejidad ciclomática

V(G) = cantidad de regiones = 5

V(G) = cantidad de aristas - cantidad de nodos + 2 = 13 - 10 + 2 = 5

V(G) = cantidad de nodos predicados + 1 = 4 + 1 = 5

Caminos de pruebas

Camino 1: 1 - 2 - 10

Camino 2: 1 - 2 - 3 - 4 - 5 - 10

Camino 3: 1 - 2 - 3 - 4 - 6 - 10

Camino 4: 1 - 2 - 3 - 7 - 8 - 10

Camino 5: 1 - 2 - 3 - 7 - 9 - 10

Casos de prueba

Camino	Descripción	Entrada	Salida
1	Este camino se realiza si el cliente no está permitido.	Características del cliente que realiza la solicitud.	Retorna *null*, indica que no se asignó una unidad de trabajo.
2	Este camino se realiza si existe una ejecución y no se crea una unidad de trabajo para el cliente.	Características del cliente que realiza la solicitud.	Retorna *null*, que indica que no se asignó una unidad de trabajo.
3	Este camino se realiza si existe una ejecución y se crea una unidad de trabajo para el cliente.	Características del cliente que realiza la solicitud.	Retorna la unidad creada desde la ejecución que es atendida.
4	Este camino se realiza si no existe un servidor de peticiones que ayudar.	Características del cliente que realiza la solicitud.	Retorna *null*, que indica que no se le asignó una unidad de trabajo.
5	Este camino se realiza si existe un servidor de peticiones que ayudar.	Características del cliente que realiza la solicitud.	Retorna la unidad creada desde la ejecución que es atendida por otro servidor de peticiones.

Código para realizar las pruebas de los caminos básicos con el JUnit

```
1  public final void testGenerateDataUnit()
2  {
3      try
4      {
5          ClientCommunication client = (ClientCommunication) Naming.lookup( url );
6          Vector dataUnit = client.getDataUnit( getClientProperties() );
7          if ( dataUnit == null || dataUnit != null )
8          {
9              if ( dataUnit != null )
10             {
11                 String serverIP = (String) dataUnit.elementAt( 0 );
12                 String fileServerPort = (String) dataUnit.elementAt( 1 );
13                 Long unitID = (Long) dataUnit.elementAt( 2 );
14                 Long algorithmID = (Long) dataUnit.elementAt( 3 );
15                 Long timeLimit = (Long) dataUnit.elementAt( 4 );
16                 byte[] compressedData = (byte[]) dataUnit.elementAt( 5 );
17             }
18             assertTrue( true );
19         }
20     }
21     catch ( RemoteException e )
22     {
23         assert( true ); // no se estableció la comunicación con el servidor
24     }
25     catch ( Exception e )
26     {
27         fail( e.getMessage() );
28     }
29 }
```

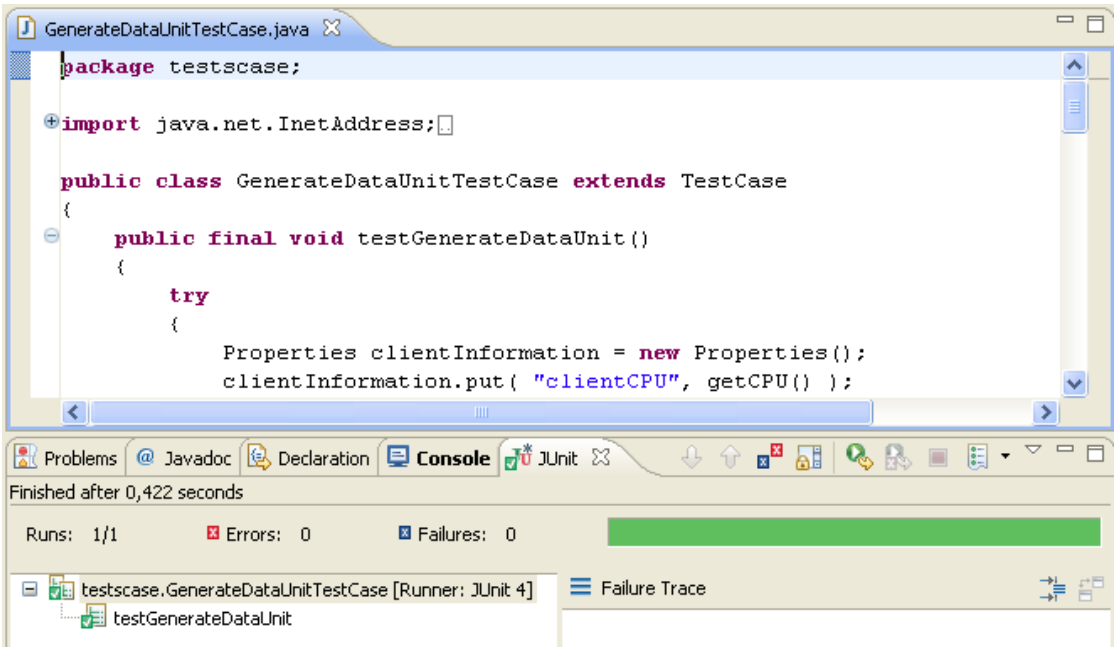

Figura 4.8: Resultado de la prueba para el camino 1 del método *generateDataUnit*.

4.3.3.4. Prueba del camino básico al método *handleResults* de la clase *Scheduler* del módulo *t-arenalServerRequest*

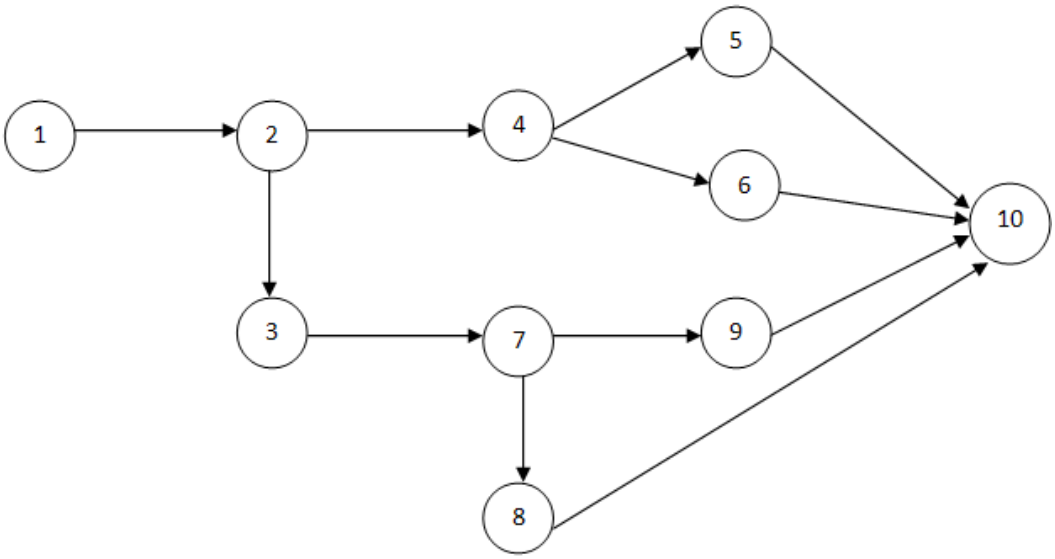

Figura 4.9: Grafo de flujo del método *handleResults*.

Cálculo de la complejidad ciclomática

V(G) = cantidad de regiones = 4

V(G) = cantidad de aristas - cantidad de nodos + 2 = 12 - 10 + 2 = 4

V(G) = cantidad de nodos predicados + 1 = 3 + 1 = 4

Caminos de pruebas

Camino 1: 1 - 2 - 3 - 7 - 8 - 10

Camino 2: 1 - 2 - 4 - 5 - 10

Camino 3: 1 - 2 - 4 - 6 - 10

Camino 4: 1 - 2 - 3 - 7 - 9 - 10

Casos de prueba

Camino	Descripción	Entrada	Salida
1	Este camino se realiza si el resultado de una unidad de trabajo no pertenece a la tarea que es atendida por el servidor de peticiones.	El resultado obtenido de una unidad de trabajo creada.	No procesa el resultado retornado.
2	Este camino se realiza si el resultado retornado pertenece a la ejecución que es atendida en ese momento, y la misma ha finalizado.	El resultado obtenido de la última unidad de trabajo creada.	Detiene la ejecución para ser adicionada como solución.
3	Este camino se realiza si el resultado retornado pertenece a la ejecución que es atendida en ese momento, y la misma no ha finalizado.	El resultado obtenido de una unidad de trabajo creada.	Actualiza el estado de la ejecución que se atiende.

4	Este camino se realiza si el resultado retornado pertenece a una ejecución que es atendida por otro servidor de peticiones.	El resultado obtenido de una unidad de trabajo creada desde otro servidor de peticiones.	

Código para realizar las pruebas de los caminos básicos con el JUnit

```
1  public final void testHandleResults()
2  {
3      try
4      {
5              Vector dataUnit = getDataUnit();
6              if ( dataUnit != null )
7              {
8                  Compressor compress = new Compressor( ClassLoader.getSystemClassLoader() );
9                  String serverIP = (String) dataUnit.elementAt( 0 );
10                 Long unitID = (Long) dataUnit.elementAt( 2 );
11                 Long executionID = (Long) dataUnit.elementAt( 3 );
12                 Vector unit = (Vector) compress.decompress( (byte[]) dataUnit.elementAt( 5 ) );
13
14                 Vector results = processUnit( unit );
15
16                 String cIP = getClientIP();
17                 Properties cInfo = getClientInformation();
18                 Vector dynInfo = getDynamicInformation();
19                 client.sendResults( cIP, serverIP, unitID, executionID, results, cInfo, dynInfo );
20                 assertTrue( true );
21             }
22         }
23         catch ( RemoteException e )
```

```
24              {
25                  assertTrue( true );
26              }
27              catch ( Throwable e )
28              {
29                  fail( e.getMessage() );
30              }
31  }
```

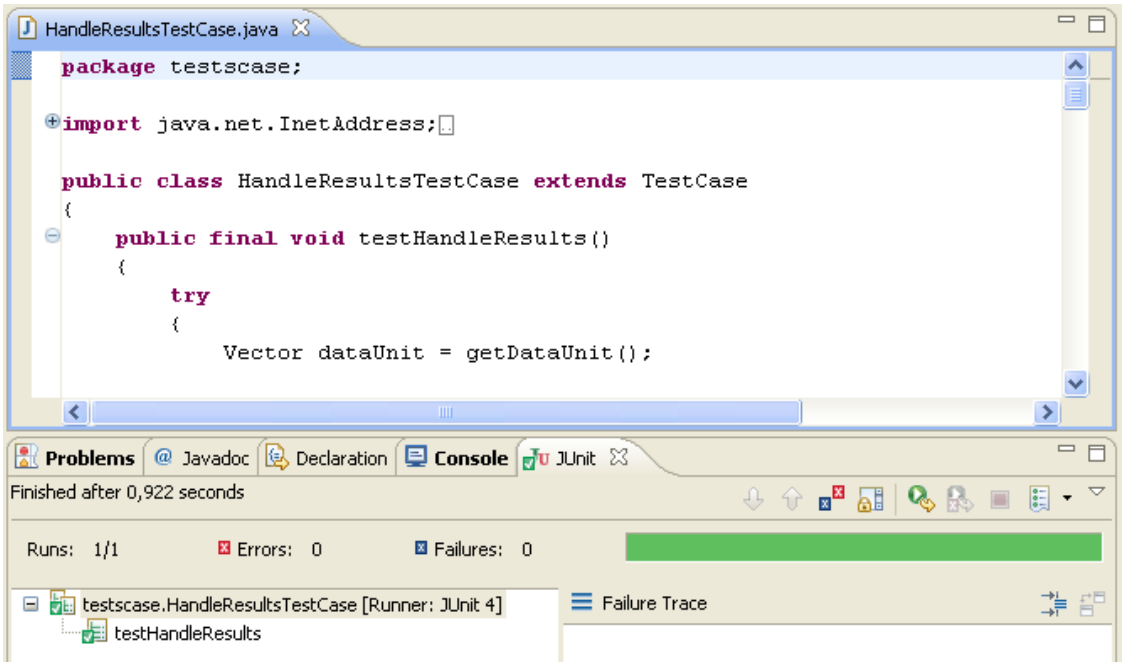

Figura 4.10: Resultado de la prueba para el camino 1 del método *handleResults*.

4.4. Conclusiones

Como resultado de este capítulo se obtuvo el diagrama de despliegue de los componentes del sistema, se hizo una breve explicación de los algoritmos más importantes que se implementaron y se mostró las pruebas de camino básico realizadas con algunos de los resultados obtenidos.

Conclusiones

1. Se identificaron las nuevas funcionalidades que cumple el back-end desarrollado y se perfeccionaron otros requisitos existentes desde la versión anterior.

2. A partir de las funcionalidades identificadas se rediseñó el back-end de la Plataforma de Tareas Distribuidas, compuesto en esta versión por un servidor central, uno o varios servidores de peticiones y por cada servidor de peticiones uno o varios clientes, haciendo uso del patrón de arquitectura Mediator.

3. Se implementó la política de asignación de tareas Least - Work - Remaining para la planificación de los trabajos que serán atendidos por el sistema.

4. Se implementó la comunicación entre los subsistemas mediante el protocolo HTTP con el fin de utilizar en un futuro la Plataforma a través de Internet.

5. Se implementó el mecanismo para realizar la persistencia de las ejecuciones que están siendo atendidas en el sistema.

6. Se realizó la evaluación de las principales funcionalidades del sistema haciendo uso de la prueba de Caja Blanca del Camino Básico.

Recomendaciones

1. Incorporar la comunicación entre el front-end y el servidor central, servidor central y servidores de peticiones, y servidores de peticiones y clientes correspondientes, a través del protocolo HTTPS.

2. Incorporar la transferencia de archivos entre el front-end y el servidor central, servidor central y servidores de peticiones, y servidores de peticiones y clientes correspondientes, a través de los protocolos FTP y FTPS.

3. Implementar el balance de carga de los clientes del sistema, de forma tal que puedan ser asignados dinámicamente según las necesidades y/o comportamiento histórico de una ejecución, o características del hardware de los servidores de peticiones.

Referencias bibliográficas

[1] Barzanallana R. Historia de la Informática; 2005. Available from: http://mundopc.net/articulos/historia-de-la-informatica/ [cited 2009 Jan 15].

[2] Bernstein J. La Máquina Analítica. Labor. Barcelona; 1981.

[3] Morrison RS. Cluster Computing; 2003.

[4] Rodríguez JPF. La bioinformática en Cuba: presente y perspectivas; 2005.

[5] Cluster Beowulf; [cited 2009 Jan 12]. Available from: http://www.beowulf.org/.

[6] Mendoza LA. Sistema de Cómputo Distribuido aplicado a la Bioinformática. Universidad de las Ciencias Informáticas; 2008.

[7] Beck A. High Throughput Computing: An Iinerview With Miron Livny. HPCwire. 1997;.

[8] Tanenbaum A, Steen MV. Distributed Systems: Principles and Paradigms. Prentice Hall, Pearson Education, USA; 2002.

[9] G Couloris JD, Kinberg T. Distributed Systems - Concepts and Design, 4th Edition. Addison-Wesley, Pearson Education, UK; 2001.

[10] SETI@HOME. Detección de señales de otras civilizaciones; 2003 [cited 2009 Jan 20]. Available from: http://seti.astroseti.org/setiathome/.

[11] The Great Internet Mersenne Prime Search; 1996. Available from: http://www.mersenne.org/ [updated 2009 Apr 10; cited 2009 Apr 15].

[12] Distributed.net;. Available from: http://www.distributed.net [updated 2009 Feb 24; cited 2009 Mar 10].

[13] Berkeley Open Infrastructure for Network Computing;. Available from: http://boinc.berkeley.edu/ [updated 2009 Jan 29; cited 2009 Feb 10].

[14] Condor. High Throughput Computing;. Available from: http://www.cs.wisc.edu/condor/ [cited 2009 Feb 10].

[15] Keane T. A General-Purpose Heterogeneous Distributed Computing System. National University of Ireland Maynooth; 2004.

[16] Winston W. Optimality of the shortest line discipline. Journal of Applied Probability. 1977;14:181–189.

[17] A Ephremides PV, Walrand J. A simple dynamic routing problem. IEEE transactions on Automatic Control. 1980;25:690–693.

[18] Whitt W. Deciding which queue to join: Some counterexamples. Operations Research. 1986 Jan/Feb;34(1):55–62.

[19] Weber RR. On the Optimal Assignment of Customers to Parallel Servers. Journal of Applied Probability. 1978 Jun;15(2):406–413.

[20] Wolff RW. An upper bound for multichannel queues. Journal of Applied Probability. 1977;14:884–888.

[21] James Rumbaugh IJ, Booch G. El Proceso Unificado de Desarrollo de Software. Addison Wesley; 1999.

[22] Extreme Programming: A gentle introduction;. Available from: http://www.extremeprogramming.org/ [updated 2006 Feb 17; cited 2009 Feb 15].

[23] Eclipse Process Framework Project;. Available from: http://www.eclipse.org/epf/ [cited 2009 Feb 20].

[24] Unified Modeling Language;. Available from: http://www.uml.org/ [cited 2009 Feb 20].

[25] James Rumbaugh IJ, Booch G. El Lenguaje Unificado de Modelado. Manual de Referencia. Segunda Edición. Addison Wesley; 2007.

[26] Armstrong E. HotSpot: A new breed of virtual machine, Javaworld;. Available from: http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html [cited 2009 Feb 15].

[27] J M Bull LP L A Smith, Freeman R. Benchmarking Java against C and Fortran for scientific applications. Journal of the ACM. 2001;.

[28] Oaks S. Java Security (2nd Edition). O'Reilly Media, Inc.; 2001.

[29] Java SE Security; 2009 Feb 10. Available from: `http://java.sun.com/security/`.

[30] Zukowski J. Programación Java 2 J2SE 1.4. vol. 1. SYBEX, Inc.; 2003.

[31] Francisca Losavio CGD. Descripción de patrones para el diseño arquitectónico utilizando RAPIDE. Acta Científica Venezolana. 2004 Dec;55(2):144–163.

[32] Elisabeth Freeman KSBB Eric Freeman. Head First Desing Patterns. O'Reilly; 2005.

[33] McCabe TJ. A Complexity Measure. IEEE Transactions on Software Engineering. 1976 Dec;2(4).

Apéndice A

Descripción de Casos de Usos Críticos

Caso de uso Autenticar Usuario

Nombre del CU	**Autenticar Usuario**
Actor	Front-end del sistema
Propósito	Autenticar un usuario en el sistema
Resumen	El front-end inicia el caso de uso cuando un usuario decide interactuar con el sistema. El sistema realiza las verificaciones de los datos enviados y finalmente lo acepta o lo rechaza.
Referencias	RF 1
Curso Normal de Eventos	
Acciones del Actor	**Respuesta del Sistema**
1. El front-end envía el nick y la contraseña del usuario que desea autenticarse.	1.1. El sistema verifica que el usuario exista
	1.2. El sistema verifica que la contraseña del usuario es correcta.
	1.3. El sistema verifica que la cuenta del usuario está habilitada
	1.4. El sistema crea y asigna una sesión al usuario

	1.5. El sistema devuelve el privilegio y la sesión del usuario autenticado, así como el puerto a utilizar por el front-end para la transferencia de archivos.
Cursos Alternativos **CA1**	
	1.1. Si el usuario no existe, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA2	
	1.2. Si la contraseña es incorrecta, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA3	
	1.3. Si la cuenta del usuario no está habilitada, el sistema emite un mensaje de error. Finaliza el caso de uso.
Prioridad	**Crítico**

Caso de uso Gestionar Problemas

Nombre del CU	**Gestionar Problemas**
Actor	Front-end del sistema
Propósito	Permitir gestionar problemas en el sistema
Resumen	El caso de uso inicia cuando el front-end envía una petición para adicionar, eliminar u obtener problemas. Una vez realizada alguna de estas acciones finaliza el caso de uso.
Referencias	RF 7.1, RF 7.2, RF 7.3, RF 7.4, RF 7.5
Curso Normal de Eventos	
Acciones del Actor	**Respuesta del Sistema**

1. El Front - end decide: ▪ Adicionar un problema. Ver sección Adicionar Problema. ▪ Eliminar un usuario. Ver sección Eliminar Problema. ▪ Obtener los problemas existentes en el sistema. Ver sección Devolver Problemas. ▪ Obtener los problemas a los que accede un usuario. Ver sección Devolver Problemas de un Usuario. ▪ Obtener los problemas que administra un usuario. Ver sección Devolver Problemas que Administra un Usuario.	
Sección Adicionar Problema	
Curso Normal de Eventos	
1. El front-end inicia la petición de adicionar un problema.	
2. Si existen archivos a transferir (ficheros que se mantienen constantes para cualquier ejecución del problema), el front-end establece la conexión con el sistema y realiza la transferencia de los mismos.	

3. El front-end envía la sesión del usuario que realiza la operación, el problema a adicionar, la tarea y el manejador de datos correspondiente, el .xml donde se describen los archivos a utilizar por una ejecución, y los .class y .jar utilizados en el desarrollo del problema.	3.1. El sistema verifica que la sesión del usuario que realiza la operación es válida.
	3.2. El sistema verifica que el usuario que realiza la operación, tenga los permisos necesarios.
	3.3. El sistema verifica que no exista otro problema con la misma clave de acceso que el problema a adicionar.
	3.4. El sistema copia los archivos enviados, adiciona el problema, y lo asigna al usuario que lo creó, y a todos los administradores del sistema. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	3.1. Si la sesión del usuario que realiza la operación no es válida, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA2	
	3.2. Si el usuario que realiza la operación no tiene los permisos necesarios, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA3	
	3.3. Si existe otro problema con la misma clave de acceso, el sistema emite un mensaje de error y no adiciona el problema. Finaliza el caso de uso.
Sección Eliminar Problemas	
Curso Normal de Eventos	

1. El front-end realiza la petición de eliminar uno o varios problemas, enviando la sesión del usuario que realiza la operación y la clave de acceso de cada problema que se desea eliminar.	1.1. El sistema verifica que la sesión del usuario es válida.
	1.2. El sistema verifica que el problema o los problemas a eliminar existan.
	1.3. El sistema verifica que el usuario que realiza la operación tenga los permisos necesarios.
	1.4. El sistema elimina el problema o los problemas deseados. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	1.1. Si la sesión del usuario no es válida, el sistema emite un mensaje de error. Finaliza el caso de uso
CA2	
	1.2. Si alguno de los problemas a eliminar no existe, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA3	
	1.3. Si el usuario que realiza la operación no tiene los permisos necesarios, el sistema emite un mensaje de error. Finaliza el caso de uso.
Sección Devolver Problemas	
Curso Normal de Eventos	
1. El front-end hace la petición de obtener todos los problemas, y envía la sesión del usuario que realiza la operación.	1.1. El sistema verifica que la sesión del usuario es válida.
	1.2. El sistema busca y retorna todos los problemas que existen en el sistema. Finaliza el caso de uso.

Cursos Alternos **CA1**	
	1.1. Si la sesión del usuario no es válida, el sistema emite un mensaje de error. Finaliza el caso de uso.
Sección Devolver Problemas de un Usuario	
Curso Normal de Eventos	
1. El front-end hace la petición de obtener los problemas a los que accede un usuario; envía la sesión del usuario que realiza la operación, y el nick del usuario del cual se desea conocer los problemas.	1.1. El sistema verifica que la sesión del usuario que realiza la operación es válida.
	1.2. El sistema verifica que exista el usuario cuyo nick fue enviado.
	1.3. El sistema retorna los problemas a los que accede el usuario cuyo nick fue enviado.
Cursos Alternos **CA1**	
	1.1. Si la sesión del usuario no es válida, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA2	
	1.2. Si el usuario cuyo nick fue enviado no existe, el sistema emite un mensaje de error. Finaliza el caso de uso.
Sección Devolver Problemas que Administra un Usuario	
Curso Normal de Eventos	

1. El front-end hace la petición de obtener los problemas que administra un usuario; envía la sesión del usuario que realiza la operación, y el nick del usuario del cual se desea conocer los problemas.	1.1. El sistema verifica que la sesión del usuario que realiza la operación es válida.
	1.2. El sistema verifica que exista el usuario cuyo nick fue enviado.
	1.3. El sistema retorna los problemas que administra el usuario cuyo nick fue enviado.
Cursos Alternos **CA1**	
	1.1. Si la sesión del usuario no es válida, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA2	
	1.2. Si el usuario cuyo nick fue enviado no existe, el sistema emite un mensaje de error. Finaliza el caso de uso.
Prioridad	**Crítico**

Caso de uso Ejecutar Problema

Nombre del CU	**Ejecutar Problema**
Actor	Front-end
Propósito	Permitir ejecutar un problema en el sistema.
Resumen	El caso de uso inicia cuando el front-end envía una petición para ejecutar un problema en el sistema. Luego transfiere los datos necesarios para el funcionamiento de la ejecución creada y finalmente le indica al sistema que ya puede atenderla.
Referencias	RF 8, RF 10
Curso Normal de Eventos	

Acciones del Actor	**Respuesta del Sistema**
1. El front-end hace una petición para ejecutar un problema, enviando la sesión del usuario que realiza la operación y la clave de acceso del problema correspondiente.	1.1. El sistema verifica que la sesión del usuario que realiza la operación es válida.
	1.2. El sistema verifica que el problema a ejecutar es accedido por el usuario
	1.3. El sistema verifica que la cantidad de ejecuciones que tendrá el usuario, tiene que ser menor que la cuota permitida.
	1.4. El sistema crea la ejecución del problema cuya clave fue enviada.
2. Si existen ficheros a transferir, el front-end establece la conexión para la transferencia de los mismos.	2.1. El sistema acepta la conexión y realiza con el front-end la transferencia de los archivos.
3. El front-end le indica al sistema que puede atender la ejecución.	3.1. El sistema adiciona la ejecución al planificador. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	1.1. Si la sesión del usuario no es válida, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA2	
	1.2. Si el problema no es accedido por el usuario, el sistema emite un mensaje de error. Finaliza el caso de uso.
CA3	
	1.3. Si la cantidad de ejecuciones que tendrá el usuario, no es menor que la cuota permitida, el sistema emite un mensaje de error. Finaliza el caso de uso.

CA4	
	1.4 Si no existen ficheros a transferir, el sistema inicia la ejecución del problema seleccionado. Finaliza el caso de uso.
CA5	
2. Si no existen ficheros a transferir, entonces el front-end realiza la actividad 3 del flujo normal de eventos.	
Prioridad	**Crítico**

Caso de uso Realizar Procesamiento

Nombre del CU	**Realizar Procesamiento**
Actor	Cliente
Propósito	Realizar la tarea que recibe desde el servidor de peticiones correspondiente.
Resumen	El caso de uso inicia cuando un cliente hace una solicitud de unidad de trabajo al servidor de peticiones correspondiente, para luego realizar su procesamiento.
CU asociado	Atender Tarea Asignada por el Servidor Central
Referencias	RF 25.1, RF 25.2, RF 25.3, RF 26
Curso Normal de Eventos	
Acciones del Actor	**Respuesta del Sistema**
1. El cliente verifica que no tiene unidad de trabajo pendiente.	
2. Si no existe una unidad de trabajo pendiente, el cliente realiza una petición al servidor de peticiones correspondiente.	2.1. El servidor de peticiones retorna la unidad de trabajo creada para el cliente *(ver caso de uso Atender Tarea Asignada por el Servidor Central. Sección Crear una Unidad de Trabajo).*

3. El cliente verifica que la versión enviada sea diferente de null, y diferente a su versión.	
4. El cliente elimina la vieja versión y copia la nueva.	
5. El cliente verifica que si tiene una unidad de trabajo a realizar, la guarda temporalmente.	
6. El cliente reinicia su funcionamiento, para iniciar con la versión actualizada. Retorna a la actividad número 1 del flujo normal de eventos.	
7. El cliente verifica que en la unidad retornada, exista una tarea a realizar.	
8. Si la unidad contiene una tarea a realizar, el cliente obtiene el identificador de la unidad, el identificador de la ejecución a la que pertenece, el tiempo límite de procesamiento y los datos requeridos por la tarea a realizar.	
9. El cliente obtiene desde el servidor de peticiones correspondiente, la tarea a realizar.	9.1. El servidor de peticiones retorna la tarea a realizar y los ficheros que serán utilizados para la realización de la misma.
10. El cliente carga la tarea y los ficheros retornados desde el servidor de peticiones.	
11. El cliente descomprime los datos recibidos en la unidad de trabajo.	
12. El cliente inicia la tarea especificada en la unidad de trabajo.	
13. El cliente inicia el hilo que controla el tiempo de procesamiento de la unidad de trabajo.	
14. El cliente realiza la tarea especificada en la unidad de trabajo.	

15. Si culminó el procesamiento, el cliente retorna el resultado obtenido hacia el sistema. Envía también sus características , el identificador de la unidad de trabajo, el identificador de la ejecución a la que pertenece la unidad de trabajo, y el tiempo total de procesamiento.	15.1. Si la ejecución existe, procesa el resultado enviado por el cliente *(ver caso de uso Atender Tarea Asignada por el Servidor Central. Sección Procesar Resultado de una Unidad de Trabajo).*
16. Después de retornado el resultado de la tarea ejecutada, el cliente retorna a la actividad número 1 del flujo normal de eventos.	
Cursos Alternativos **CA1**	
2. Si existe una unidad de trabajo pendiente, el cliente no realiza ninguna petición, y se dirige a la actividad número 8 del flujo normal de eventos.	
CA2	
3. Si la versión enviada es igual a cero, o igual a la versión existente, el cliente no inicia el proceso de actualización. El cliente se dirige a la actividad número 7 del flujo normal de eventos.	
CA3	
8. Si la unidad no contiene una tarea, el cliente esperará un tiempo, para retornar a la actividad número 1 del flujo normal de eventos.	
CA4	

12. Si al iniciar la tarea ocurre una excepción de tipo *FileNotFoundException*, el cliente descarga desde el sistema, el archivo no encontrado y necesitado para realizar la tarea. Después, retorna a la actividad número 12 del flujo normal de eventos.	
CA5	
14. Si durante la realización de la tarea: ▪ Se agota el tiempo de asignado: el cliente le pide al servidor de peticiones que le extienda el mismo *(ver caso de uso Atender Tarea Asignada por el Servidor Central. Sección Incrementar Tiempo de Atención de una Unidad de Trabajo)*. Si esto no ocurre, se detiene la ejecución de la tarea, y retorna a la actividad número 1 del flujo normal de eventos. ▪ Ocurre una excepción o error: el cliente detiene la ejecución de la tarea y reporta el error ocurrido *(ver caso de uso Atender Tarea Asignada por el Servidor Central. Sección Recibir Notificación sobre algún Error Ocurrido)*. Retorna a la actividad número 1 del flujo normal de eventos.	
CA6	
15. Si el cliente no ha culminado la tarea, retorna a la actividad número 14 del flujo normal de eventos.	

Prioridad	**Crítico**

Caso de uso Atender Tarea Asignada por el Servidor Central

Nombre del CU	**Atender Tarea Asignada por el Servidor Central**
Actor	Cliente
Propósito	Atender en el servidor de peticiones la tarea asignada por el servidor central.
Resumen	El caso de uso inicia cuando un cliente hace al servidor de peticiones correspondiente, una solicitud de procesamiento, devuelve el resultado obtenido o pide aumentar el tiempo de atención de la unidad que le fue asignada.
Referencias	RF 27.1, RF 27.2, RF 27.3, RF 27.4
Curso Normal de Eventos	
Acciones del Actor	**Respuesta del Sistema**
El cliente realiza una solicitud de: ▪ Obtener una unidad de trabajo. Ver sección Crear una Unidad de Trabajo. ▪ Enviar el resultado del procesamiento realizado. Ver sección Procesar Resultado de una Unidad de Trabajo. ▪ Aumentar tiempo de procesamiento. Ver sección Incrementar Tiempo de Atención de una Unidad de Trabajo. ▪ Enviar error ocurrido durante el procesamiento. Ver sección Recibir Notificación sobre algún Error Ocurrido.	

Sección Crear una Unidad de Trabajo	
Curso Normal de los Eventos	
1. El cliente solicita una unidad de trabajo para su procesamiento.	1.1. El servidor de peticiones adiciona o actualiza el cliente reportado con los datos enviados.
	1.2. El servidor de peticiones verifica que el cliente esté autorizado a interactuar con el mismo.
	1.3. El servidor de peticiones verifica que exista una ejecución para ser atendida.
	1.4. Si existe la ejecución, el servidor de peticiones verifica que no esté pausada.
	1.5. El servidor de peticiones crea una unidad de trabajo según las características del cliente que la solicita.
	1.6. El servidor de peticiones adiciona la unidad creada a la lista de unidades pendientes de respuesta.
	1.7. El servidor de peticiones finalmente retorna la unidad generada. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	1.2. Si el cliente no está autorizado a interactuar, el servidor de peticiones retorna solamente la actualización correspondiente. Finaliza el caso de uso.
CA2	

	1.3. Si no existe una ejecución el servidor de peticiones: ▪ Verifica que si está colaborando con otro servidor de peticiones, entonces delegada la creación de la unidad de trabajo al servidor al cual ayuda. Posteriormente retorna a la actividad número 1.6 del flujo normal de eventos. ▪ Si no colabora con ningún servidor entonces retorna solamente la actualización correspondiente. Finaliza el caso de uso.
CA3	
	1.4. Si existe la ejecución, pero la misma está pausada, el servidor de peticiones retorna solamente la actualización correspondiente. Finaliza el caso de uso.
Sección Procesar Resultado de una Unidad de Trabajo	
Curso Normal de los Eventos	
1. El cliente solicita enviar el resultado obtenido del procesamiento de una unidad de trabajo.	1.1. El servidor de peticiones verifica que la unidad retornada le pertenece.
	1.2. El servidor de peticiones verifica que exista la ejecución correspondiente a la unidad de trabajo retornada.
	1.3. El servidor de peticiones procesa el resultado retornado, y elimina la unidad de trabajo del conjunto de unidades pendientes.
	1.4. Si la ejecución finalizó el servidor inicia la compresión del directorio de trabajo asignado a la ejecución. Finaliza el caso de uso.

Cursos Alternativos **CA1**	
	1.1. Si la unidad retornada pertenece a otro servidor, el servidor de peticiones se comunica con el mismo y le envía los datos correspondientes. Finaliza el caso de uso.
CA2	
	1.2. Si no existe la ejecución, el servidor de peticiones no procesa el resultado retornado. Finaliza el caso de uso.
Sección Incrementar Tiempo de Atención de una Unidad de Trabajo	
Curso Normal de los Eventos	
1. El cliente solicita incrementar el tiempo de procesamiento de la unidad de trabajo asignada.	1.1. El servidor de peticiones verifica que la unidad de trabajo a la cual le desean aumentar su tiempo de atención le pertenezca.
	1.2. El servidor de peticiones verifica que exista la ejecución correspondiente a la unidad de trabajo.
	1.3. El servidor de peticiones verifica que la unidad de trabajo exista.
	1.4. El servidor de peticiones aumenta el tiempo de procesamiento, y lo actualiza en el conjunto de unidades pendientes.
	1.5. El servidor retorna en cuanto se aumentó el tiempo.
Cursos Alternativos **CA1**	

	1.1. Si la unidad de trabajo no le pertenece, el servidor de peticiones se comunica con el servidor al cual le corresponde y le realiza la solicitud de aumentar el tiempo de atención. Retorna a la actividad número 1.5 del flujo normal de eventos.
CA2	
	1.2. Si no existe la ejecución el servidor de peticiones retorna 0. Finaliza el caso de uso.
CA3	
	1.3. Si no existe la unidad de trabajo el servidor de peticiones retorna 0. Finaliza el caso de uso.
Sección Recibir Notificación sobre algún Error Ocurrido	
Curso Normal de los Eventos	
1. El cliente envía el error ocurrido durante el procesamiento de la unidad de trabajo asignada	1.1. El servidor de peticiones verifica que la unidad de trabajo donde ocurrió el error le pertenezca.
	1.2. El servidor de peticiones verifica que exista la ejecución correspondiente a la unidad de trabajo donde ocurrió el error.
	1.3. El servidor de peticiones registra el error ocurrido en los archivos correspondientes a la ejecución.
	1.4. Si han ocurrido más de cinco errores en una misma unidad de trabajo, el servidor de peticiones detiene la ejecución.
Cursos Alternativos **CA1**	
	1.1. Si no le pertenece la unidad de trabajo, se comunica con el servidor al cual le corresponde y entonces le notifica el error ocurrido. Finaliza el caso de uso.
CA2	

	1.2. Si no existe la ejecución el servidor no realiza nada. Finaliza el caso de uso.
Prioridad	**Crítico**

Caso de uso Realizar Chequeo en el Servidor Central

Nombre del CU	**Realizar Chequeo en el Servidor Central**
Actor	Reloj
Propósito	Mantener la consistencia de los datos en el servidor.
Resumen	El caso de uso inicia cuando después de un tiempo se ejecuta un hilo (thread), para mantener la consistencia de los datos, eliminar los ficheros y directorios que no se utilicen, y controlar el estado en que se encuentran las ejecuciones asignadas.
Referencias	RF 28
Curso Normal de Eventos	
Acciones del Actor	**Respuesta del Sistema**
1. Finaliza el tiempo de inactividad e inicia el chequeo en el servidor.	1.1. El sistema elimina los directorios no usados o vacíos, del almacén de unidades pendientes y expiradas.
	1.2. El servidor central verifica el tamaño de los archivos logs.
	1.3. El sistema verifica que los servidores de peticiones que atienden ejecuciones se hallan reportado.
Cursos Alternativos **CA1**	
	1.2. Si los servidores de peticiones que manejan ejecuciones no se han reportado, se pasa esas ejecuciones como pendientes a asignar a otro servidor.
Prioridad	**Crítico**

Caso de uso Gestionar Tarea en el Servidor de Peticiones

Nombre del CU	**Gestionar Tarea en el Servidor de Peticiones**
Actor	Reloj
Propósito	Solicitar al servidor central un tarea y mantener el control de la misma.
Resumen	El caso de uso inicia cuando después de un tiempo, el servidor de peticiones se comunica con el servidor central, y solicita atender una tarea. Si la tarea asignada es una ejecución, entonces lleva el control de la misma, de lo contrario espera un tiempo e inicia nuevamente el caso de uso.
CU asociados	Asignar Tarea a un Servidor de Peticiones, Recibir Reporte sobre una Ejecución Asignada
Referencias	RF 29.1, RF 29.2

Curso Normal de Eventos	
Acciones del Actor	**Respuesta del Sistema**
1. Finaliza el tiempo de inactividad e inicia el caso de uso.	1.1. El servidor de peticiones se comunica con el servidor central.
	1.2. El servidor de peticiones solicita una tarea al servidor central *(ver caso de uso Asignar Tarea a un Servidor de Peticiones).*
	1.3. El servidor de peticiones verifica que le haya sido asignada una tarea.
	1.4. Si la tarea asignada es una ejecución, el servidor de peticiones descarga la misma desde el servidor central, posteriormente la inicializa y la coloca para que sea atendida.
	1.5. El servidor de peticiones establece el estado de inactividad.
	1.6. Después de transcurrido el tiempo de inactividad, el servidor de peticiones comprueba que por lo menos haya sido atendida por un cliente local y no por otro servidor.

	1.7. Si la ejecución finalizó, el servidor transfiere hacia el servidor central la solución obtenida, y le notifica sobre su culminación *(ver caso de uso Recibir Reporte sobre una Ejecución Asignada. Sección Recibir Confirmación sobre una Ejecución Finalizada)*. Retorna a la actividad número 1 del flujo normal de eventos.
Cursos Alternativos **CA1**	
	1.1. Si ocurrió algún error durante la comunicación, el servidor de peticiones establece el estado de inactividad. Finaliza el caso de uso.
CA2	
	1.3. Si no le fue asignada una tarea, el servidor establece el estado de inactividad. Finaliza el caso de uso.
CA3	
	1.4. Si la tarea asignada es colaborar con otro servidor, entonces se establece la comunicación. El servidor de peticiones pasa a estado de inactividad. Finaliza el caso de uso.
CA4	
	1.6. Si no fue atendida por lo menos por un cliente local, el servidor de peticiones detiene la ejecución, compacta los resultados obtenidos hasta ese momento, y notifica al servidor central que no tiene clientes para atenderla *(ver caso de uso Recibir Reporte sobre una Ejecución Asignada. Sección Recibir Reporte sobre una Ejecución no Atendida)*. Retorna a la actividad número 1 del flujo normal de eventos.
CA5	

	1.7. El servidor de peticiones guarda los resultados parciales, realiza el control de unidades pendientes de respuestas, y notifica al servidor central sobre el estado en que se encuentra la ejecución *(ver caso de uso Recibir Reporte sobre una Ejecución Asignada. Sección Recibir Reporte sobre una Ejecución Atendida).* Retorna a la actividad número 1.5 del flujo normal de eventos.
Prioridad	**Crítico**

Caso de uso Asignar Tarea a un Servidor de Peticiones

Nombre del CU	**Asignar Tarea a un Servidor de Peticiones**
Actor	Reloj (CU Gestionar Tarea en el Servidor de Peticiones)
Propósito	Asignar una tarea a un servidor de peticiones.
Resumen	El caso de uso inicia cuando un servidor de peticiones realiza una solicitud de trabajo. El servidor central le asigna la atención de una ejecución, o le indica que colabore con otro servidor de peticiones.
Referencias	RF 30.1, RF 30.2
Curso Normal de Eventos	
Acciones del Actor	**Respuesta del Sistema**
1. Un servidor de peticiones solicita una tarea a realizar.	1.1. El servidor central comprueba la existencia del servidor de peticiones reportado.
	1.2. El servidor central actualiza el tiempo de reporte del servidor de peticiones que solicita trabajo.
	1.3. El servidor central verifica que el servidor de peticiones esté autorizado a atender tareas.
	1.4. El servidor central busca una ejecución que pueda atender el servidor de peticiones reportado.

	1.5. El servidor central retorna junto con la tarea asignada, las actualizaciones de clientes que existan. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	1.1. Si el servidor de peticiones no existe, el servidor central lo rechaza y no le asigna ninguna tarea. Finaliza el caso de uso.
CA2	
	1.3. Si el servidor de peticiones no está autorizado, el servidor central lo rechaza y no le asigna ninguna tarea. Finaliza el caso de uso.
CA3	
	1.4. Si no existe una ejecución que pueda atender el servidor de peticiones: ▪ El servidor central busca con cual servidor puede colaborar. Retorna a la actividad número 1.5 del flujo normal de eventos. ▪ Si no existe servidor con cual pueda colaborar, el servidor central retorna las actualizaciones de clientes que existan. Finaliza el caso de uso.
Prioridad	**Crítico**

Caso de uso Recibir Reporte sobre una Ejecución Asignada

Nombre del CU	**Recibir Reporte sobre una Ejecución Asignada**
Actor	Reloj (CU Gestionar Tarea en el Servidor de Peticiones)

<table>
<tr><td>Propósito</td><td colspan="2">Recibir y registrar la información enviada por el servidor de peticiones sobre una ejecución.</td></tr>
<tr><td>Resumen</td><td colspan="2">El caso de uso inicia cuando el servidor de peticiones decide notificar al servidor central si atiende la ejecución asignada, si no la puede atender, o si esta ya finalizó.</td></tr>
<tr><td>Referencias</td><td colspan="2">RF 31.1, RF 31.2, RF 31.3</td></tr>
<tr><td colspan="3">Curso Normal de Eventos</td></tr>
<tr><td>Acciones del Actor</td><td colspan="2">Respuesta del Sistema</td></tr>
<tr><td>1. El actor decide:
Enviar estado de una ejecución atendida. Ver sección Recibir Reporte sobre una Ejecución Atendida.
Notificar que una ejecución no es atendida. Ver sección Recibir Reporte sobre una Ejecución no Atendida.
Confirmar de que ha finalizado una ejecución. Ver sección Recibir Confirmación sobre una Ejecución Finalizada.</td><td colspan="2"></td></tr>
<tr><td colspan="3">Sección Recibir Reporte sobre una Ejecución Atendida</td></tr>
<tr><td colspan="3">Curso Normal de Eventos</td></tr>
<tr><td>1. Un servidor de peticiones envía el estado de la ejecución que atiende.</td><td colspan="2">1.1. El servidor central verifica que la ejecución exista.</td></tr>
<tr><td></td><td colspan="2">1.2. El servidor central verifica que el servidor de peticiones que envía el reporte, no esté registrado como que no puede atender la ejecución.</td></tr>
</table>

	1.3. El servidor central verifica que el servidor de peticiones que envía el reporte esté permitido.
	1.4. El servidor central verifica que el servidor de peticiones que envía el reporte es el que tiene asignado la ejecución.
	1.5. El servidor central guarda la información enviada por el servidor de peticiones. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	1.1. Si no existe la ejecución finaliza el caso de uso.
CA2	
	1.2. Si el servidor de peticiones está registrado como que no puede atender la ejecución, finaliza el caso de uso.
CA3	
	1.3. Si el servidor de peticiones no está permitido finaliza el caso de uso.
CA4	
	1.4. Si el servidor de peticiones que envía el reporte no es el que atiende a la ejecución, se realiza una comparación para ver quien está más adelantado, y entonces es asignado a la ejecución.
Sección Recibir Reporte sobre una Ejecución no Atendida	
Curso Normal de Eventos	
1. Un servidor de peticiones notifica que no puede atender una ejecución.	1.1. El servidor central verifica que la ejecución exista.
	1.2. El servidor central verifica que la ejecución sea atendida por el servidor de peticiones que notifica.

	1.3. El servidor central adiciona el servidor de peticiones a la lista de servidores que no pueden atender una ejecución. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	1.1. Si no existe la ejecución finaliza el caso de uso.
CA2	
	1.2. Si la ejecución no es atendida por el servidor de peticiones que notifica, finaliza el caso de uso.
Sección Recibir Confirmación sobre una Ejecución Finalizada	
Curso Normal de Eventos	
1. Un servidor de peticiones notifica que ha culminado de atender una ejecución.	1.1. El servidor central verifica que la ejecución exista.
	1.2. El servidor central elimina la ejecución y adiciona la solución al usuario que la inició. Finaliza el caso de uso.
Cursos Alternativos **CA1**	
	1.1. Si no existe la ejecución finaliza el caso de uso.
Prioridad	**Crítico**