TESIS
II 2004
H43

Universidad Católica Andrés Bello.

Facultad de Ingeniería

Escuela de Ingeniería Informática

MEJORAMIENTO DEL DESEMPEÑO EN LAS COMUNICACIONES DE LOS SERVICIOS DE EJECUCIÓN DE UN GRID COMPUTACIONAL

Este jurado; una vez realizado el examen del presente trabajo ha evaluado su contenido con el resultado: Diez y Nueve (19) y Diez y Ocho (18).

JURADO EXAMINADOR

Evelenir Barreto

José Antonio González

Yudith C. Cardinale

Realizado por: **Jossman A. Hernández R.**
Rodolfo J. Montilla P.

Profesor Guía: **Yudith C. Cardinale.**

Fecha: **Octubre 2004.**

Índice General

AGRADECIMIENTOS i

DEDICATORIA ii

ÍNDICE DE FIGURAS vi

ÍNDICE DE TABLAS vii

RESUMEN viii

CAPÍTULO I. Introducción 1

1. - Motivación y Justificación 1

2. - Planteamiento del Problema 2

3. - Objetivos 4

3.1. - Objetivo General 4

3.2. - Objetivos Específicos 4

4. - Alcances y Limitaciones 5

CAPÍTULO II. Marco Teórico 7

1. - *Grids* Computacionales 7

2. - Precarga de Clases 9

2.1. - Formato de un Archivo *.Class* 11

3. - *Buffering* de Datos 14

4. - CORBA (*Common Object Request Broker Architecture*) 15

5. - SUMA: un *grid* computacional para ejecución de *bytecode* de Java 16

5.1. - Objetivos de SUMA 16
5.2. - Características de SUMA 17
5.3. - Estructura Básica de SUMA 18
5.4. - Componentes de SUMA 19
5.5. - Procesos de Invocación y Ejecución 23
5.5.1. - *Execute* 24
5.5.2. - *Submit* 25
5.6. - Carga Dinámica de Clases en SUMA 26
5.7. - *Buffering* en SUMA 27

CAPÍTULO III. Metodología 28

1. - Metodología RUP (*Rational Unified Process*) 28

1.1. - Fase de Inicio 29
1.2. - Fase de Elaboración 30
1.3. - Fase de Construcción 31
1.4. - Fase de Transición 32

CAPÍTULO IV. Desarrollo del Módulo de Precarga de Clases Para un *Grid* Computacional 33

1. - Fase de Inicio 33

2. - Fase de Elaboración 37

3. - Fase de Construcción 42

4. - Fase de Transición 45

CAPÍTULO V. Desarrollo del *Buffering* Para un *Gris* Computacional 47

1. - Fase de Inicio 47

2. - Fase de Elaboración 49

3. - Fase de Construcción 52

3.1 - Propuesta: Recuperación de la Excepción 54
3.2 - Propuesta: Diseño de un Buffer Optimista - Por Tablas 55

4. - Fase de Transición 56

CAPÍTULO VI. Elaboración de Pruebas 57

1. - Ambientes de Pruebas 57

1.1. - Hardware y Software 57
1.2. - Escenarios 57

2. - Tipos de Pruebas 58

2.1. - Pruebas de Funcionalidad 58
2.2. - Pruebas de Validación y Desempeño 60

3. - Resultados de las Pruebas de Desempeño 61

CAPÍTULO VII. Conclusiones 65

CAPÍTULO VIII. Recomendaciones 67

BIBLIOGRAFÍA 68

GLOSARIO DE TÉRMINOS 71

APÉNDICES 73

Índice De Figuras

Figura II.1. Estructura de SUMA 18

Figura II.2. Secuencia de
Invocaciones a Métodos SUMA 24

Figura III.1. Metodología RUP 28

Figura IV.1. Funcionamiento Actual de SUMA 34

Figura IV.2. Diagramas de Caso de
Uso Servicios *Submit* y *Execute* Modificados 36

Figura IV.3. Leyenda de Diagramas de Clases 37

Figura IV.4. Diagrama de Clases del
Cliente Interno Modificado 38

Figura IV.5. Diagrama de Clases del
Agente de Ejecución Modificado 40

Figura IV.6. Diagrama de Secuencia
Módulo de Precarga 45

Figura V.1. Diagrama de Casos de
Uso Módulo de *Buffering* 48

Figura V.2. Diagrama de Clases del Agente
de Ejecución Módulo de *Buffering* 50

Figura V.3. Diagrama de Secuencia
Módulo de *Buffering* 53

Índice De Tablas

Tabla IV.1. Proceso de Ejecución
del Módulo de Precarga 46

Tabla VI.1. Tiempos de la Prueba de
Desempeño en Ambiente Local 62

Tabla VI.2. Tiempos de Invocación
de la Clase JGFAllSizeA 63

Tabla VI.3. Tiempos de Invocación de
la Clase JGFArithBench 64

Resumen

SUMA es una plataforma distribuida de ejecución de aplicaciones en *bytecode* de Java, cuyo objetivo principal es aprovechar eficientemente las capacidades de los recursos con que se cuenta en una red local o global, poniendo al alcance de todos los usuarios la velocidad de plataformas de cómputo intensivo.

Para mejorar el rendimiento y confiabilidad de la plataforma SUMA se desarrollaron e incorporaron dos nuevos módulos. El primero consistió en desarrollar una precarga de clases necesarias para la ejecución de un *bytecode,* de tal manera de poder disminuir las interrupciones durante el proceso de ejecución para transferir las clases necesarias ubicadas del lado del cliente. El segundo de estos módulos permitió garantizar la integridad de los datos que son extraídos del *buffer,* verificando si un archivo ha sido modificado durante el proceso de lectura.

La realización de este Trabajo Especial de Grado estuvo guiada por la metodología RUP (*Rational Unified Process*), la cual se basa en el lenguaje unificado de modelo (UML), Esta metodología permitió dividir el trabajo en 4 ciclos o fases, en donde cada una de estas presenta sus propios objetivos, actividades y entregables, permitiendo así tener un mayor control sobre el trabajo realizado.

Capítulo I. Introducción

1. - Motivación y Justificación

En la actualidad, a la hora de diseñar y crear programas no sólo se busca que éstos sean eficaces sino también eficientes, en vista de que la tendencia es obtener la información solicitada en tiempo real. Por esta razón cualquier segundo de retraso en una aplicación puede ocasionar grandes pérdidas no sólo a nivel informativo, sino incluso a nivel económico.

Muchas empresas invierten miles de dólares constantemente en la búsqueda de soluciones que minimicen los tiempos de respuesta de sus aplicaciones, sin importar que estas mejoras sólo brinden pequeños cambios en los tiempos esperados. Estas soluciones pueden venir dadas por muchos aspectos, desde la integración de un nuevo módulo hasta la sustitución del software o hardware utilizado por uno nuevo.

En el caso particular de plataformas distribuidas, a través de los cuales se acceden a recursos ubicados en diferentes localizaciones geográficas, es importante hacer optimizaciones a nivel de las comunicaciones. Estas optimizaciones son de total interés en los casos de ejecuciones y acceso a datos remotos.

La tecnología *Grid* surge del nuevo paradigma de computación distribuida propuesto por Ian Foster y Carl Kesselman[12] a mediados de los 90. Se basa fundamentalmente en el acceso remoto a recursos computacionales, y su tecnología estándar es el *Globus Toolkit*[1]. Su objetivo es permitir gestionar y distribuir la potencia de cálculo disponible, de tal forma que los

[1] Globus Toolkit: Software de código abierto usado para construir Grids

usuarios se beneficien de la potencia de computadores infrautilizados que se encuentran dispersos geográficamente. Éstos ceden parte de sus recursos para procesar aplicaciones con funciones de cálculo avanzadas.

Determinar por adelantado qué clases se necesitarían para la ejecución remota de un programa, o diseñar un mejor manejo de un *buffer* de datos para una buena manipulación de los datos, no sólo representaría una disminución en los tiempos de respuesta sino también permitiría ejecutar sus servicios de una manera más eficiente. Estas cualidades son altamente anheladas especialmente en sistemas para el soporte de computación de alto desempeño, como es el caso de los *Grids* computacionales.

2. - Planteamiento del Problema

SUMA (*Scientific Ubiquitous Metacomputing Architecture*) es un *Grid* computacional para la ejecución de programas Java, tanto secuenciales como paralelos. Se encarga de la administración de recursos distribuidos geográficamente y de la ejecución de *bytecodes* de Java, permitiendo a los usuarios una transparencia de localidad de las aplicaciones que sean ejecutadas en el sistema.

Cuando un usuario desea ejecutar un programa, éste solicita un servicio de ejecución de SUMA *(Execute o Submit)* a través de clientes por comandos, clientes gráficos o vía Web. En el momento que SUMA recibe una solicitud de un cliente, localiza una plataforma de ejecución apropiada y envía un mensaje a la misma. Un agente de ejecución en la plataforma designada

comienza la ejecución del programa. Para el servicio *Execute*, el usuario sólo tiene que especificar la clase principal *(main)*. Las otras clases y los archivos de datos se leen desde el cliente en tiempo de ejecución, por demanda del método que se está ejecutando, y la salida es enviada de nuevo al cliente.

Tanto el servicio *Submit* como el *Execute* requieren la carga dinámica de clases y datos desde el cliente hacia el servidor, desde el momento en que se inicia la ejecución del programa. Este proceso implica ciertos retardos en la comunicación; especialmente cuando existe un gran número de clases adicionales y un gran conjunto de datos a ser procesados; debido a que la ejecución se detiene mientras se cargan las clases y datos desde el cliente hasta la memoria del servidor.

Como Trabajo Especial de Grado, se plantea la incorporación a SUMA de mecanismos que permitan la carga de clases y datos por anticipado, para poder aprovechar la concurrencia que pueda existir entre los servidores de ejecución y la transmisión de datos, y de esta manera poder lograr un mejor desempeño en la comunicación de los Servicios de Ejecución de SUMA, debido a que cualquier mejora que se pueda brindar a esta plataforma que asegure una disminución en los tiempos de respuesta, garantizaría un mejor servicio a los usuarios.

3. - Objetivos

3.1. - Objetivo General

Incorporar técnicas de precarga de clases, *buffering* y *caching* de datos en la ejecución de aplicaciones de usuarios, en ambientes de *Grids* Computacionales basados en Java, con la finalidad de reducir los retardos en la comunicación.

3.2. - Objetivos Específicos

- Desarrollar módulos de análisis de *bytecode* para determinar cuáles son las clases que serán llamadas durante la ejecución del programa para que sean cargadas en la memoria del servidor por anticipado.
- Realizar un estudio comparativo de las posibles técnicas de precarga de clases del lado del servidor de ejecución, de manera de administrar eficientemente las clases precargadas.
- Implantar la mejor técnica de precarga de clases del lado del servidor de ejecución de acuerdo al estudio comparativo propuesto en el objetivo anterior.
- Evaluar los tiempos de respuesta de los servicios *execute* y *submit* de SUMA antes y después de haber desarrollado la precarga de clases, con la finalidad de determinar el nivel de mejora de los mismos.

- Estudiar los mecanismos básicos que posee la plataforma SUMA con respecto al manejo del *buffering* y *caching* de datos.

- Proponer e implantar mejoras en el uso de *buffering* y *caching* de datos en la plataforma SUMA

- Evaluar los tiempos de respuestas para obtener cuantitativamente las mejoras con respecto a la versión actual.

4. - Limitaciones y Alcances

- El caso de estudio que se empleó fue SUMA: un *Grid* de cómputo para la ejecución de programas Java tanto secuenciales como paralelos.

- Por ser este Trabajo Especial de Grado una nueva funcionalidad de la plataforma SUMA, el desarrollo del mismo se llevó a cabo en un principio en las instalaciones de su nodo matriz ubicado en la Universidad Simón Bolívar.

- Debido a que la plataforma SUMA se encuentra actualmente en funcionamiento, todas las pruebas que fueron realizadas se aplicaron bajo la supervisión del personal del laboratorio de SUMA.

- Por razones de presupuesto el software utilizado fue de código abierto y de distribución gratuita.

- El Trabajo Especial de Grado sólo estuvo enfocado en la mejora del rendimiento de las comunicaciones entre un cliente y el servidor de ejecuciones, la cual se logró a través de la implementación de la precarga de clases y del mejoramiento del *buffering* y *caching* de datos que existe actualmente en SUMA.

Debido a que el desarrollo de este trabajo especial de grado está basado fundamentalmente en un ambiente distribuido, es importante tener claros algunos conocimientos en lo que se refiere a sistemas de este tipo.

1. - *Grids* Computacionales

La posibilidad de compartir recursos computacionales costosos a lo largo de un dominio extendido de usuarios y oferentes de recursos se ha implementado a través de sistemas basados en sistemas ya existentes, lo que ha sido llamado "meta-sistemas" o "meta-computadores", y más recientemente *Grids* computacionales.

Un *Grid* computacional es distinguible de una simple colección de computadores por un nivel de software, generalmente llamada *middleware*, que transforma un conjunto de recursos independientes en una única y coherente máquina virtual. Idealmente, esta máquina debería ser tan simple de usar como la máquina que el usuario tiene en su escritorio y debería permitir fácil colaboración entre colegas localizados en cualquier parte del mundo. Es responsabilidad del *Grid* computacional ofrecer la metáfora de una única máquina, administrando transparentemente los movimientos de datos, *caching* y conversión; detectando y administrando fallas; asegurando que los recursos físicos y de datos de los usuarios son protegidos en forma adecuada y planificando y despachando los componentes de las aplicaciones sobre los recursos disponibles al usuario.

Algunos de los beneficios potenciales de un *Grid* computacional para la comunidad de científicos y organizaciones que necesitan ejecutar aplicaciones de alto rendimiento son:

- Los recursos de super-computación son muy costosos como para ser replicados. Alternativamente una aplicación puede requerir recursos que no podrían ser adquiridos porque tal configuración particular se requiere sólo raras veces. Finalmente, ciertos recursos únicos (como bases de datos especializadas) no pueden ser replicados.

- Muchas organizaciones disponen de equipos informáticos que no se usan durante el 100% del tiempo. En este sentido, es importante desarrollar algoritmos que permitan aprovechar equipos informáticos durante el tiempo de "ocio" en que la organización propietaria no los utilice, ofreciendo al mismo tiempo niveles de seguridad satisfactorios a la organización propietaria.

- Muchas organizaciones requieren disponibilidad de poder de cómputo en ambientes geográficamente dispersos y hostiles donde no es posible trasladar material de cómputo. Tales ambientes requieren un terminal móvil, muy ligero, de muy bajo poder de cómputo, que disponga de conexión a equipos distantes. Estos terminales servirían para recopilar datos que serían enviados a computadores ubicados en sitios fijos pero geográficamente dispersos, y probablemente realizar cálculos a tiempo real, que requieren de la selección de una plataforma que se encuentre ociosa.

2. - Precarga de Clases

Con el pasar de los años la computación se ha hecho más compleja e innovadora; nuevas herramientas de programación, mejoras en las técnicas de procesamiento de información y almacenamiento de la misma, manejo de archivos; son unos de los tantos ejemplos que existen en la actualidad.

Quizás una de las más significativas soluciones o avances en la historia de la computación fue la ejecución de programas de manera remota, la cual permitió procesar información desde cualquier parte donde se encuentren los usuarios siempre y cuando se tenga alguna conexión entre la máquina o dispositivo en donde se ejecutarán los archivos y donde se encuentren almacenados los mismos.

Según estudios realizados por el Departamento de Ciencias de la Computación e Ingeniería de la Universidad de California en San Diego, en conjunto con el Departamento de Ciencias de la Computación de la Universidad de California en Santa Bárbara [9], se plantea que "la proliferación de Internet está motivando al desarrollo de ambientes de computación móvil en donde código portable puede ser ejecutado en sitios remotos". Estos ambientes de desarrollo resultaron ser algo novedosos, por un lado brindaban la facilidad de procesamiento desde cualquier parte, pero por otro lado pueden generar retrasos en la ejecución de los programas. En dicho estudio, además se plantea que "el desempeño de los programas móviles depende tanto de la latencia de la red y de la disponibilidad del ancho de banda, así como también de la velocidad de procesamiento durante la ejecución". "El retraso en los tiempos de respuesta

de un sistema perturba la ejecución de los procesos de un usuario", es por ello que desarrolladores se han dado a la tarea de buscar la manera de aminorar los tiempos en las transferencias de archivos y así obtener un mayor rendimiento de los programas.

Entre algunas de las mejoras que han implementado se encuentran [10]:

- *Splitting* o "División de Clases".
- *Prefetching* o "Precarga de Clases".
- *Non-Strict Execution* o "Ejecución no Estricta de Clases"

La precarga de clases es una técnica empleada para reducir el retardo en la ejecución de programas de manera remota. Esta técnica oculta el retraso de transferencia solapando la misma con ejecución, por ejemplo, un archivo *.class* es transferido por la red, mientras se va ejecutando, eliminando en un caso óptimo dicho retraso. Una precarga efectiva requiere primero de una política para determinar en qué punto de la ejecución se debería realizar cada petición de carga y así maximizar el solapamiento. Segundo, debería presentar un mecanismo de activación de la carga de clases para ejecutar la precarga.

Los retrasos por transferencias de clases y archivos; necesarios para la ejecución del programa; ocurren porque la ejecución del programa debe realizar una interrupción cada vez que necesita una clase y la misma no se encuentra disponible. Primero la clase o archivo es buscado localmente en los dispositivos de almacenamiento, y si éste no se encuentra se realiza la petición de transferencia de la clase. Con la precarga de clases se busca mantener

a disposición del programa de una manera local todas aquellas clases mucho antes que el mismo las necesite, reduciendo en gran número la pérdida de tiempo por transferencias.

"La precarga mas efectiva es aquella que es semánticamente invisible al programa y sin causar ninguna falta de memoria" [1]. La idea de una precarga es disminuir el retardo por transferencias, por lo tanto, la misma debe trabajar de una manera casi imperceptible y natural. Con o sin precarga, la JVM trabajará de la misma manera, las clases que se necesiten serán cargadas en el momento en que sean solicitadas y ejecutadas según la estructura del programa.

2.1. - Formato de un Archivo *.Class*

Dado que el caso de estudio es SUMA, el cual es un sistema que ofrece servicios de ejecución de *bytecodes* de Java es necesario conocer la estructura que posee un archivo *.class*.

Cada archivo *.class* contiene una sencilla definición de una clase o de una interfaz. Cada archivo consiste en un *stream* de 8-*bits* de *bytes*, en donde todas las cantidades de 16-*bits*, 32-*bits* y 64-*bits*, son el resultado de leer en 2, 4 u 8 consecutivos 8-*bits* de *bytes*. Elementos de diferentes tamaños son almacenados siguiendo un orden *big-endian*,[2] en donde los de mayor tamaño se encuentran de primero. En plataformas Java y Java2, este formato es

[2] Big-Endian: Adjetivo que se emplea para indicar cuales bytes son mas significativos en un tipo de dato multi-bytes. En un sistema big-endian el valor más significativo va almacenado en las posiciones mas bajas. http://www.webopedia.com/TERM/b/big_endian.html

soportado por interfaces *Java.io.DataInput* y *Java.io.DataOutput*, y clases como *Java.io.DataInputStream* y *Java.io.DataOutputStream*.

Un archivo *.class* presenta la siguiente estructura [19]:

- *magic*
- *minor_version*
- *major_version*
- *constant_pool_count*
- *constant_pool[constant_pool_count-1]*
- *access_flags*
- *this_class*
- *super_class*
- *interfaces_count*
- *interfaces[interfaces_count]*
- *fields_count*
- *field_info fields[fields_count]*
- *methods_count*
- *method_info methods[methods_count]*
- *attributes_count*
- *attribute_info attributes[attributes_count]*

El valor de *magic* consiste en un número de 4 *bytes*, cuyo valor es siempre 0xCAFEBABE. Este valor le permite conocer a la JVM si en verdad está ejecutando un archivo creado por alguna otra JVM.

Los valores *minor_version* y *major_version* permiten determinar la versión del archivo *.class*, y a su vez sirven de referencia a las JVM para determinar si son capaces de soportar dicha versión.

El *constant_pool_count* contiene la cantidad de elementos que presenta el *constant_pool[ ]* (arreglo de constantes). El *constant_pool[ ]* consiste en una tabla de estructuras que contiene valores de constantes de *string*, nombres de clases e interfaces, nombres de archivos, y otras constantes.

El *access_flag* tiene la finalidad de denotar los permisos de acceso a dicha interfaz o clase, tomando valores como público, privado entre otros.

Los valores *this_class* y *super_class* contienen un índice válido en el "pool de constantes", la cual apunta a una constante del tipo clase y que representa a su vez el nombre de la clase y la superclase directa respectivamente, definida por el *.class*.

El *interface_count* representa el número de elementos presentes en el *interfaces[ ]* (Arreglo de Interfaces). Cada valor en el arreglo *interfaces[ ]* debe tener un índice valido el "pool de constantes", en donde dicho valor representa cuáles interfaces "implementa" o "extiende" la clase principal.

El *fields_count* representa el número de elementos presentes en el *fields[ ]*. La estructura *fields[ ]* contiene información sobre todos los *fields* que han sido declarados en la clase o interfaz pero no sobre aquellos que han sido heredados a través de una superclase.

El *methods_count* representa el número de elementos presentes en el *methods[ ]*. La estructura *methods[ ]* contiene la información sobre todas aquellas instancias de métodos, métodos estáticos e instancias de inicialización de métodos.

El *attributes_count* representa el número de elementos presentes en el *attributes[ ]*. La estructura *attributes[ ]* contiene la información de aquellos atributos que pertenezcan al archivo fuente, y aquellos atributos que puedan ser removidos.

3. - *Buffering* de Datos

Un *buffer* es una región de la memoria donde se almacenan y manipulan datos, los cuales luego serán copiados definitivamente en el disco o transmitidos a través de una red según sea el caso.

El *buffering* es una técnica que da suficiente poder de control de accesos en ambientes multiusuario y que ayuda a reducir el tiempo de lectura de los datos, ya que una vez que estos se encuentran en el *buffer* es más rápido tener acceso a ellos que si estuvieran en el disco o en un lugar remoto.

Entre las técnicas que se utilizan para implementar el *buffering* se encuentran cuatro que son:

- **Modo Optimista:** de esta forma ningún registro se bloquea cuando comienza la edición, es decir, cuando se intenta actualizar en el disco se produce el bloqueo del

registro y éste se libera una vez finalizada su modificación. Esto maximiza la disponibilidad del registro.

- **Modo Pesimista:** cuando se produce una edición se intenta bloquear un registro. Si esto es posible, la edición se permite y el registro es liberado. De lo contrario se produce un error y se rechaza la edición.

- **Modo Fila:** este modo permite editar un solo registro del *buffer* y cuando el puntero se mueve de registro se guardan las actualizaciones que se han hecho; a esto se le conoce como actualización implícita.

- **Modo Tabla:** este modo permite que sean múltiples registros los que sean modificados en el *buffer* del cursor, lo que implica que no se produzca la actualización implícita si éste se mueve.

4. - CORBA (*Common Object Request Broker Architecture*)

CORBA definido por el *Object Management Group* (ONG) [11] provee una arquitectura independiente a la plataforma y del lenguaje de programación para escribir aplicaciones orientadas a objetos y distribuidas. Los objetos de CORBA pueden residir en la misma máquina del proceso cliente o en cualquier otra máquina conectada a través de una red o de Internet, superando la heterogeneidad de los sistemas que contienen los objetos que

interactúan. Se podría asegurar entonces, que la combinación CORBA-Java se presenta como una herramienta conveniente en el desarrollo de *Grid* computacionales.

5. - SUMA: un *grid* computacional para ejecución de *bytecode* de Java.

SUMA es una plataforma distribuida de ejecución de aplicaciones en *bytecode* de Java, tanto aplicaciones multi-hilos (en un único espacio de memoria) como paralelas, comunicándose a través de paso de mensajes.

SUMA se implementa como un sistema de objetos distribuidos y tiene entre sus objetivos principales la transparencia de localidad a las aplicaciones que sean ejecutadas en el sistema. En principio, la plataforma SUMA no está limitada a Java, porque otros lenguajes pueden ser compilados a *bytecode* de la Máquina Virtual Java [13]. Adicionalmente, SUMA provee acceso a librerías nativas de alto desempeño para permitir el desarrollo de aplicaciones más eficientes.

5.1 - Objetivos de SUMA

El objetivo principal de SUMA es ofrecer acceso a recursos distribuidos de alto desempeño para ejecución de *bytecode* de Java, incluyendo código paralelo con comunicación a través de *mpiJava*. Las plataformas de ejecución de *bytecode* pueden contener envoltorios *(wrappers)* de funciones científicas nativas, optimizadas para la plataforma de hardware sobre la que se ejecuta, con el propósito de compensar la baja velocidad de ejecución que ofrecen muchos interpretadores de *bytecode*. Además se busca cumplir con otros objetivos:

- Aprovechar eficientemente las capacidades de los recursos con que se cuenta en una red local o global, poniendo al alcance de todos los usuarios la velocidad de plataformas de cómputo intensivo.

- Compartir recursos computacionales bajo convenios de cooperación entre universidades y centros de investigación.

- Aprovechar el tiempo ocioso de la capacidad computacional instalada.

5.2 - Características de SUMA

- Capacidad de ejecución de aplicaciones compiladas a *bytecode* de Java, sobre un conjunto de recursos distribuidos.

- Transparencia en la ejecución de programas con una interfaz sencilla, en donde la visión de máquina virtual presentada al usuario es la de una máquina poderosa en su escritorio.

- Servicios implementados en modo nativo, por ejemplo librerías de álgebra lineal como *PLapack*.

- Adicionalmente, SUMA provee herramientas para el perfilado de desempeño *(performance profiling)* de aplicaciones, así como de monitoreo y administración de la máquina virtual.

5.3 - Estructura Básica de SUMA

De una manera transparente para el usuario, SUMA está implementado como un conjunto de servicios CORBA. Los usuarios ejecutan sus programas a través de procesos cliente que invocan servicios de SUMA, los cuales se implementan como facilidades CORBA. En la Figura II.1 se puede apreciar la estructura de la plataforma SUMA.

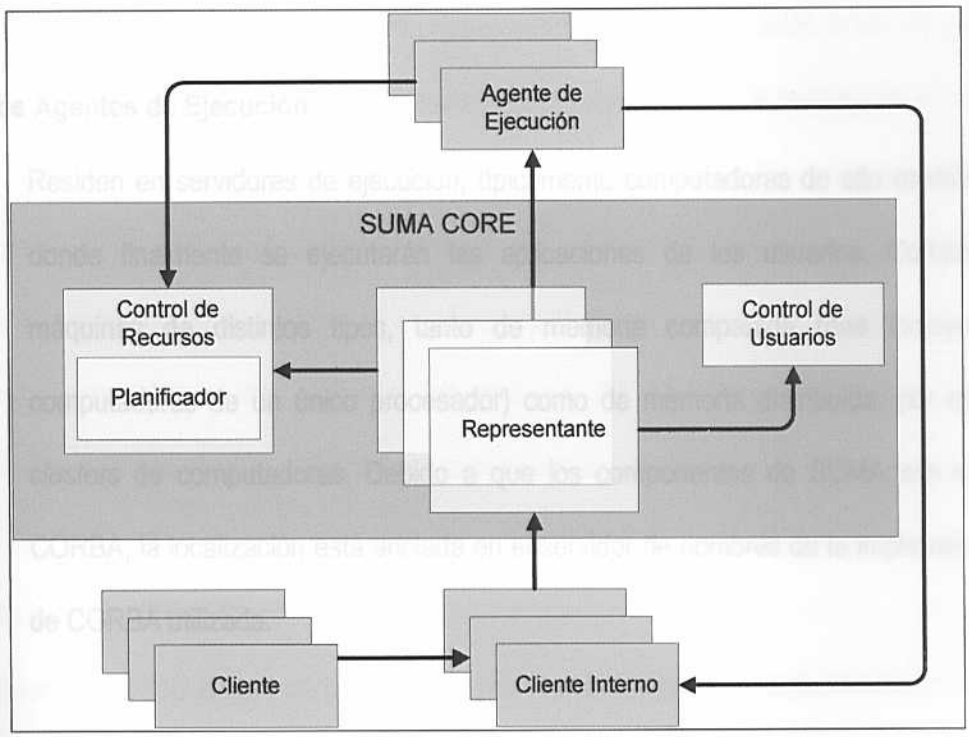

Figura II.1. Estructura de SUMA
Fuente: CAR[2]

Los Clientes

- Son aplicaciones de diversos tipos y funcionalidades que se comunican con el sistema de transporte, entregando la información necesaria para solicitar la ejecución de una aplicación.

El Sistema de Transporte o Núcleo

- Se encarga de recibir las peticiones y encontrar un servidor de aplicaciones adecuado para la ejecución de cada petición, monitoreando el estado de los servidores y garantizando la ejecución de las aplicaciones.

Los Agentes de Ejecución

- Residen en servidores de ejecución, típicamente computadoras de alto rendimiento, donde finalmente se ejecutarán las aplicaciones de los usuarios. Comprenden máquinas de distintos tipos, tanto de memoria compartida (que incluyen las computadoras de un único procesador) como de memoria distribuida, por ejemplo *clusters* de computadoras. Debido a que los componentes de SUMA son objetos CORBA, la localización está anclada en el servidor de nombres de la implementación de CORBA utilizada.

5.4 - Componentes de SUMA

Los principales componentes de la plataforma SUMA son:

Agentes de ejecución (*Execution Agents*)

Los agentes de ejecución proveen un método único llamado *execute*. Un Agente de Ejecución recibe una orden de un representante para ejecutar una aplicación, entonces comienza a cargar clases y archivos dinámicamente. Para las plataformas paralelas, el agente de ejecución juega el rol de puerta de entrada. El método *execute* tiene cuatro parámetros:

RU *execute* (*in* EU, *in ExecAgentConf*, *in AbstractClient*, *inout ProfilerConf*)

El método *execute* devuelve una estructura de datos (RU o *Results Unit*) que contiene información acerca de la ejecución del programa, tal como las características del nodo de ejecución. La estructura de datos de entrada EU (Ex*ecution Unit*) contiene información relacionada con las clases y archivos de datos que necesita la aplicación a ser cargada. El parámetro *ExecAgentConf* contiene información adicional acerca de la solicitud de ejecución, como el número de nodos, si el servicio de puntos de reanudación fue requerido, etc. Este parámetro es parte del *Configuration Unit* (CU), una estructura de datos que el representante recibe. El parámetro *AbstractClient* es la referencia al cliente interno, usada por el agente de ejecución para obtener las clases y los archivos. El parámetro *ProfilerConf* contiene información sobre perfilación, cuando este servicio sea solicitado.

Cliente interno (*Client Stub*)

El software de SUMA incluye un paquete que permite a los clientes acceder a los servicios de SUMA. Este paquete contiene una clase (el cliente interno) que atiende las llamadas desde el Agente de Ejecución (*callbacks*) para cargar las clases y archivos de datos dinámicamente. Un método importante que provee el cliente interno es:

bytearray findClass (*in string classname*);

Este método devuelve la clase requerida por la aplicación. El Cliente Interno contiene métodos que envuelven las clases Java relacionadas con el manejo de archivos. Estas clases son: *File, FileDescriptor, FileInputStream, InputStream* (para *System.in*) y *OutputStream* (para *System.out* y *System.err*). Todos los métodos definidos en el Cliente Interno son invocados por las aplicaciones que se ejecutan en los nodos remotos.

Representante (*Proxy*)

Un representante tiene básicamente dos funciones. Primero, conecta un cliente con un agente de ejecución, de tal modo que estos componentes se comuniquen directamente entre sí. La segunda función de un representante es implementar el servicio *submit*. Si un programa termina anormalmente, el representante solicita otro nodo y reinicia la ejecución. Los métodos principales del representante son: *execute, submit, getResults, getStatus* y *cancel*. Estos métodos son llamados desde los clientes.

El método *execute* provee el servicio de ejecución interactiva. Los métodos *submit, getResults, getStatus* y *cancel* son usados para las ejecuciones fuera de línea. Un *submit* instancia un cliente interno virtual en el representante, luego pasa su referencia a un *execute* de un Agente de Ejecución. Retorna *Submit_ID*, que identifica la tarea dentro de SUMA. El método *getResults* es invocado por clientes de SUMA y devuelve los resultados (en el parámetro *Results*) de un *submit* finalizado exitosamente. Este método necesita el identificador de tarea (*Submit_ID*) e información sobre el usuario (*UsrInfo*) para autenticación.

El método *getStatus* devuelve el estado de una tarea que ha sido enviada a SUMA con el comando *submit (executing, waiting, terminated*). Finalmente, *cancel* permita desencolar una tarea enviada que todavía esté esperando por ejecución.

Planificador (*Scheduler*)

El planificador implementa métodos para la selección del representante y el agente de ejecución. Esta selección se basa en los requerimientos de la aplicación y en información de estado obtenida de las plataformas de ejecución. El planificador mantiene información acerca de los recursos disponibles en SUMA, como por ejemplo la descripción de las plataformas de ejecución en términos de su tipo (*clusters*, redes de estaciones de trabajo, PC's, etc.), poder relativo, tamaño de la memoria, librerías disponibles y carga promedio. Los representantes y los agentes de ejecución se registran y se retiran ellos mismos en el planificador, proveyendo la información necesaria para que éste tome decisiones de asignación de recursos.

Control de usuarios (*User Control*)

Se encarga del registro y la autenticación de usuarios. Maneja capacidades para controlar el acceso a los recursos. El método principal del componente control de usuarios es:

boolean checkAccess(in UserInfo, in ExecAgentConf)

Este método devuelve *true* si una ejecución, representada por *ExecAgentConf*, y solicitada por el usuario descrito en *UserInfo*, puede ser concedida; retorna *false* en caso contrario. El control de usuarios también incluye métodos para tareas administrativas, como por ejemplo agregar y eliminar usuarios.

5.5 - Proceso de invocación y ejecución

Una vez que SUMA recibe la solicitud de una máquina cliente, localiza una plataforma de *ejecución transparentemente y envía un mensaje a dicha plataforma. Un agente de ejecución* en la plataforma designada comienza la ejecución del programa. Para el servicio *execute*, el *usuario sólo tiene que especificar la clase principal (main). Las otras clases y los archivos de* datos se leen desde el cliente en tiempo de ejecución, por demanda del método que se está ejecutando, y la salida es enviada de nuevo al cliente. Para el servicio *submit*, la máquina del cliente envía todas las clases y archivos de entrada a priori; la salida se mantiene en SUMA hasta que el usuario la solicite. Se puede introducir varios parámetros adicionales junto con el programa, como restricciones de planificación y el tipo de plataforma de ejecución.

La figura II.2 muestra la secuencia de invocaciones a métodos dentro de SUMA para el *execute* (líneas delgadas) y *submit* (líneas gruesas). Las cajas en dicha figura representan objetos CORBA y las flechas representan invocaciones de estos métodos.

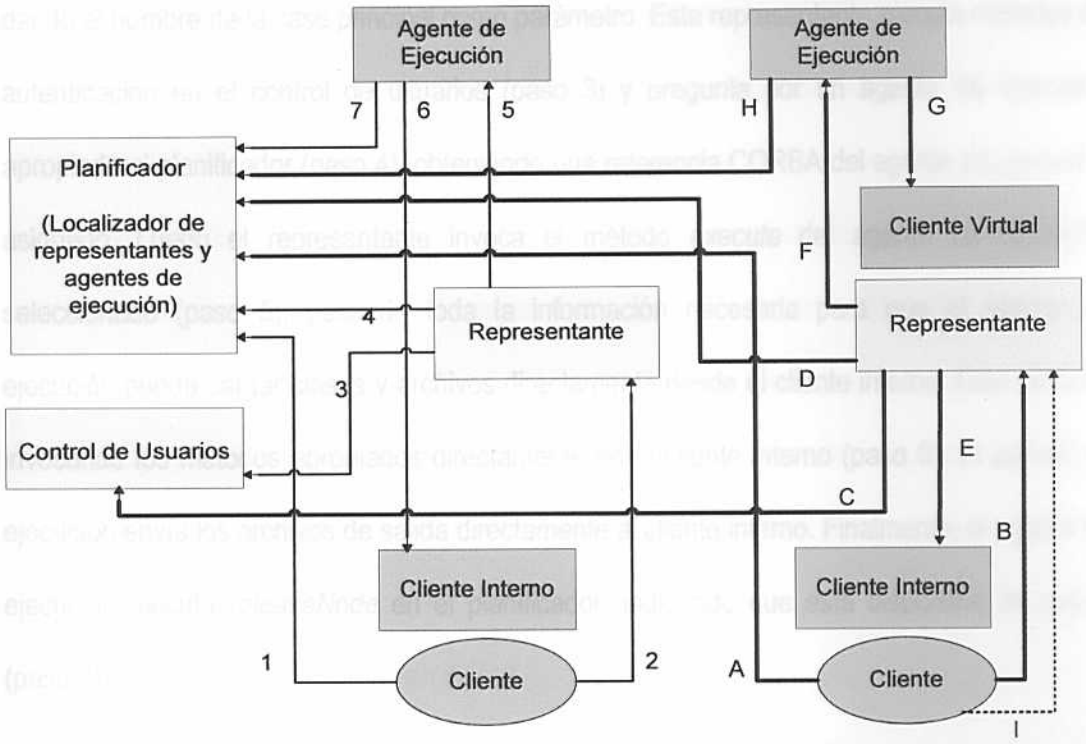

Figura II.2. Secuencia de invocaciones a métodos en SUMA
Fuente: CAR[2]

Antes de invocar los métodos *execute* o *submit*, algunos objetos esenciales de SUMA deben registrarse en el servidor de nombres de CORBA. Adicionalmente, los agentes de ejecución y los representantes deben registrarse en el planificador. A continuación se explican las secuencias de invocaciones para el *execute* y el *submit*.

5.5.1 - *Execute*

Cuando un cliente desea solicitar una ejecución en SUMA, primero debe encontrar un representante, invocando el método *findProxy* en el planificador. El planificador encuentra un representante apropiado y retorna una referencia CORBA al cliente (paso 1 en la figura II.2). El cliente puede entonces invocar el método *execute* en el representante asignado (paso 2),

dando el nombre de la case principal como parámetro. Este representante ejecuta métodos de autenticación en el control de usuarios (paso 3) y pregunta por un agente de ejecución apropiado al planificador (paso 4), obteniendo una referencia CORBA del agente de ejecución asignado. Luego el representante invoca el método *execute* del agente de ejecución seleccionado (paso 5), pasando toda la información necesaria para que el agente de ejecución pueda cargar clases y archivos directamente desde el cliente interno. Esto se hace invocando los métodos apropiados directamente en el cliente interno (paso 6). El agente de ejecución envía los archivos de salida directamente al cliente interno. Finalmente, el agente de ejecución ejecuta *releaseNode* en el planificador, indicando que está disponible de nuevo (paso 7).

5.5.2 - *Submit*

Si un cliente desea ejecutar un *submit* en SUMA, debe invocar el método *findProxy* en el planificador. El planificador encuentra un representante apropiado (debe tener la capacidad para actuar como un cliente virtual) y retorna una referencia CORBA al cliente (paso A en la figura II.2). Luego el cliente invoca el método *submit* en el representante asignado (paso B), pasando los nombres de todas las clases y archivos necesarios para ejecución como parámetros. El representante seleccionado ejecuta métodos de autentificación en el control de usuarios (paso C) y pregunta por un agente de ejecución adecuado al planificador (paso D), obteniendo una referencia CORBA para el agente de ejecución. El representante obtiene todas las clases y archivos especificados en el comando *submit* (paso E). En este momento, el cliente puede desconectarse de SUMA. El representante invoca el método *execute* en el

agente de ejecución seleccionado (paso F), pasando toda la información necesaria al agente de ejecución para que éste pueda cargar las clases y los archivos por demanda desde el cliente interno virtual. Esto se hace llamando a los métodos apropiados en el cliente virtual asociado (paso G). Finalmente, el agente de ejecución ejecuta un *releaseNode*, indicando que está disponible de nuevo (paso H). El representante debe tener capacidad de almacenamiento para mantener los archivos de salida hasta que el cliente se conecte de nuevo (si se desconectó), e invoque el método *getResults* en el representante (flecha punteada I). Note que para obtener los resultados el cliente se debe conectar al mismo representante.

5.6 - Carga Dinámica de Clases en SUMA

El uso de un *classloader* independiente para cada aplicación, permite la carga de clases referenciadas por la clase principal de la aplicación a ejecutar, desde el cliente hasta el agente en tiempo de ejecución. Al momento de iniciar la ejecución de una petición en un esclavo de un agente de ejecución, se crea una instancia de un *SUMAClassLoader* para cargar las clases desde el cliente interno que solicitó el servicio de SUMA. En la implementación actual sólo se cargan desde el cliente las clases que no pertenecen a los paquetes propios del JDK, de la implementación CORBA o de SUMA. Una función importante del *classloader* de SUMA es no permitir a las aplicaciones cargar clases propias de SUMA, para evitar que puedan realizar modificaciones al sistema.

5.7 - *Buffering* en SUMA

La carga remota de archivos se realiza utilizando un sistema de *buffering* que reduce significativamente el número de *callbacks* CORBA de lectura al cliente, adelantándose al requerimiento de lectura de la aplicación. A través de esta técnica se reduce el sobre-tiempo *(overhead)* introducido por la ejecución remota. El tamaño del buffer de entrada y salida es parametrizable por el cliente, así podrá adaptarse a los requerimientos de su aplicación.

1. - Metodología RUP (*Rational Unified Process*)

Para el desarrollo del Trabajo Especial de Grado se empleó la metodología RUP creada por Rational Software Corporation. Se basa en el lenguaje unificado de modelado (UML) y su objetivo principal es el de asegurar la producción de software de calidad dentro de plazos y presupuestos predecibles [2]. Dirigido por casos de uso, centrado en la arquitectura, iterativo (mini-proyectos) e incremental (versiones).

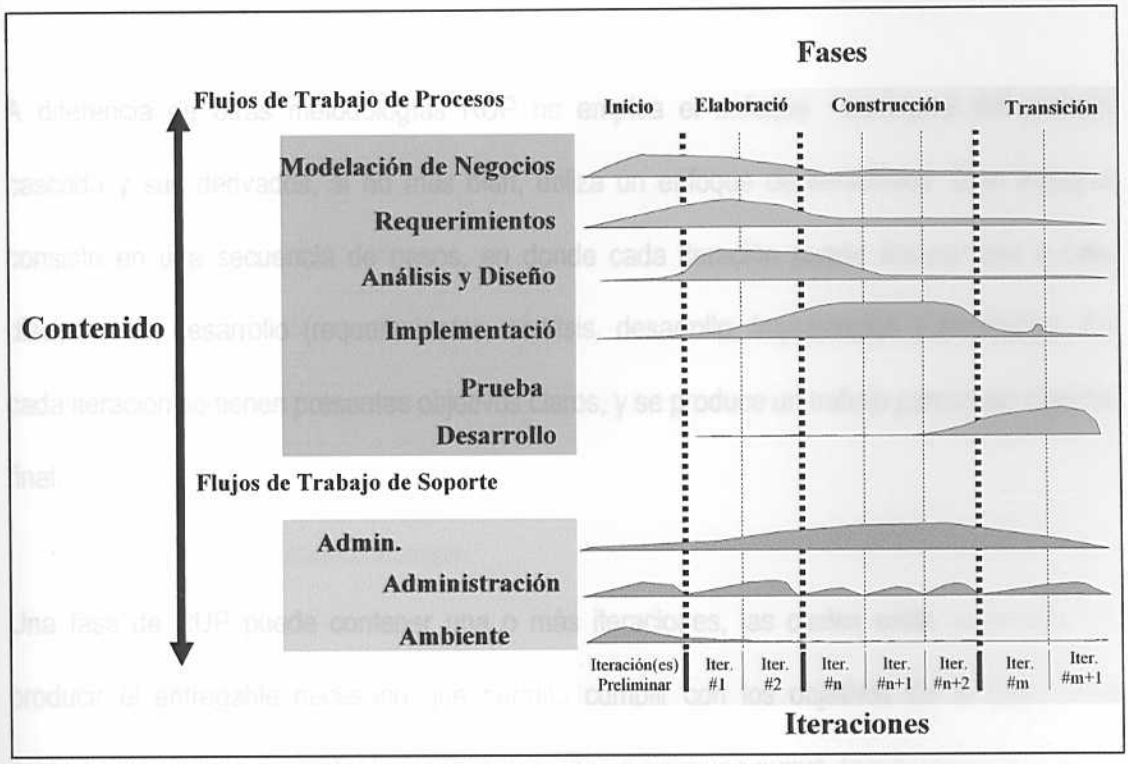

Figura III.1. Metodología RUP
Fuente: The Rational Unified Process Made Easy[2]

RUP divide el proceso de desarrollo en ciclos [2]; en donde cada ciclo consta de 4 fases; teniendo un producto al final de cada ciclo. Cada fase concluye con un hito bien definido donde deben tomarse ciertas decisiones. Las fases de un ciclo son:

- Inicio.
- Elaboración.
- Construcción.
- Transición.

A diferencia de otras metodologías RUP no emplea el enfoque tradicional del método cascada y sus derivados, si no más bien, utiliza un enfoque de iteraciones. Este enfoque consiste en una secuencia de pasos, en donde cada iteración puede abarcar una o más disciplina de desarrollo (requerimientos, análisis, desarrollo, implantación y transición). En cada iteración se tienen presentes objetivos claros, y se produce un trabajo parcial del sistema final.

Una fase de RUP puede contener una o más iteraciones, las cuales están enfocadas en producir el entregable necesario que permita cumplir con los objetivos de la fase. Si la iteración termina sin concretar la fase, se debe dar inicio a una nueva, la cual dará inicio, justo en el punto de culminación de la iteración anterior.

1.1 - Fase de Inicio

La fase de inicio está conformada por los siguientes objetivos:

- Establecer la oportunidad y alcance el proyecto. La oportunidad del proyecto incluye:
  - Criterios de éxito
  - Identificación de riesgos
  - Estimación de recursos necesarios
- Identificar todas las entidades externas con las que se trata (actores) y definir la interacción a un alto nivel de abstracción:
  - Identificar todos los casos de uso
  - Describir algunos en detalle

Los entregables de esta fase son:

- Un documento de visión general.
- Modelo inicial de casos de uso (10% a 20 % listos).
- Criterios de éxito
- Identificación inicial de riesgos.

1.2 - Fase de Elaboración

Los objetivos de esta fase son:

- Analizar el dominio del problema.
- Establecer una arquitectura base sólida.
- Desarrollar un plan de proyecto.

- Eliminar los elementos de mayor riesgo para el desarrollo exitoso del proyecto.

Los entregables de esta fase son:

- Modelo de casos de uso (80% completo) con descripciones detalladas.
- Otros requerimientos no funcionales o no asociados a casos de uso.
- Descripción de la Arquitectura del Software.
- Un prototipo ejecutable de la arquitectura.
- Plan de desarrollo para el resto del proyecto.

1.3 - Fase de Construcción

Su objetivo principal es alcanzar la capacidad operacional del producto. En esta fase todos los componentes restantes se desarrollan e incorporan al producto. Todo es probado en profundidad. El énfasis está en la producción eficiente y no en la creación intelectual. Puede hacerse construcción en paralelo, pero esto exige una planificación detallada y una arquitectura muy estable.

Los entregables de esta fase son:

- El producto de software integrado y corriendo en la plataforma adecuada.
- Manuales de usuario.
- Una descripción del *release* actual.

1.4 - Fase de Transición

El objetivo de esta fase es traspasar el software desarrollado a la comunidad de usuarios. Una vez instalado surgirán nuevos elementos que implicarán nuevos desarrollos (ciclos).

Los entregables de esta fase son:

- Pruebas Beta para validar el producto con las expectativas del cliente.
- Ejecución paralela con sistemas antiguos.
- Conversión de datos.
- Entrenamiento de usuarios.
- Distribución del producto.

Capítulo IV. Desarrollo del Módulo de Precarga de Clases Para un *Grid* Computacional

Siguiendo el esquema de la metodología escogida, el trabajo fue dividido en varias iteraciones, cada una con un conjunto de entregables bien definidos, que permitieron el cumplimiento satisfactorio de los objetivos planteados. Como se puede apreciar en el plan de trabajo en la tabla 1 del apéndice A, estas iteraciones incluyen una o más disciplinas de desarrollo, las cuales pueden variar el nivel de intensidad a tal grado que podrá aparecer o no en dicha iteración, dependiendo de la fase en que se encuentre comprendida.

1. - Fase de Inicio

Durante esta fase de inicio se lograron despejar incógnitas referentes al "¿Qué?" y "¿Cómo?" del trabajo que se desempeñaría a lo largo de todas las fases, así como también, poder identificar alcances y posibles riesgos del proyecto.

Al comienzo de esta fase se realizaron un conjunto de actividades enfocadas en el levantamiento de información, análisis de requerimientos y visualización de posibles riesgos, los cuales permitieron tener suficiente información sobre el alcance y los objetivos, para luego poder confirmar si era factible o no seguir con el proyecto, para luego trazar que procedimientos había que seguir y sobre todo qué se tenía que construir.

Las primeras actividades consistieron en levantamiento de información por medio de entrevistas, y búsqueda de material referente al tema de "precarga de clases". Las entrevistas fueron llevadas a cabo con personal de la plataforma SUMA, en especial con docentes que

han participado en este gran proyecto. En estas entrevistas se trató todo lo referente al funcionamiento actual de la plataforma SUMA, sus principales componentes, y áreas de posible riesgo a la hora de realizar alguna modificación.

Por medio de estas primeras actividades se logró tener una visión general del comportamiento actual de los servicios *execute* y *submit* de la plataforma SUMA, que se puede apreciar en la figura IV.1.

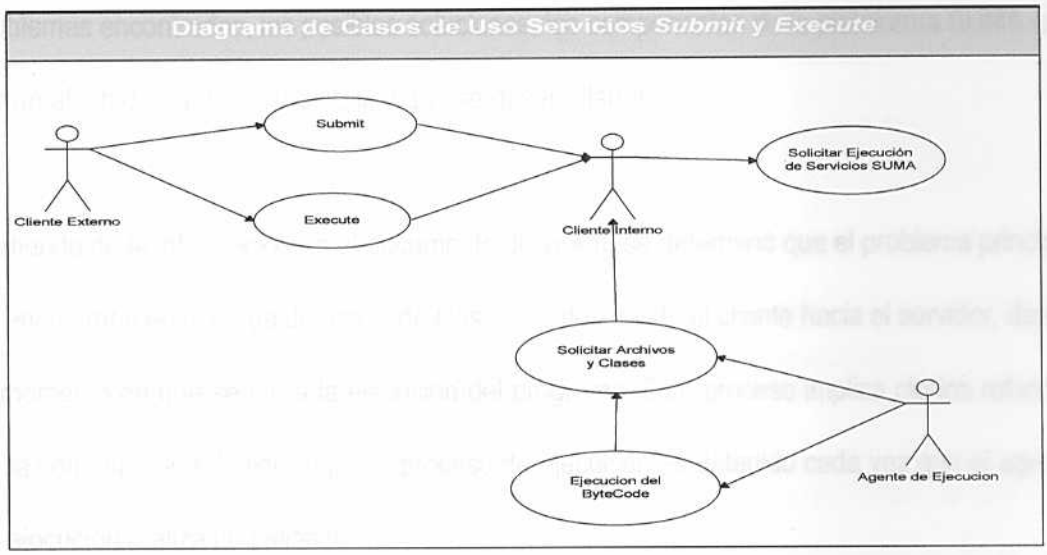

Figura IV.1. Funcionamiento Actual de SUMA
Fuente: Elaboración propia.

Además en esta fase se realizó un estudio del código fuente de la plataforma SUMA, a manera de determinar la dependencia entre las clases que la conforman. Esta dependencia se puede apreciar en el apéndice B.

Otro punto que se tomó en consideración fue estimar todos aquellos recursos que serían necesarios a lo largo de todo el proyecto tanto a nivel de software como hardware. Estos requerimientos se pueden apreciar con mayor detalle en el apéndice C.

En base a esta primera información, se realizó un documento de visión (ver tabla 1 apéndice D), con el cual se pudo tener una aproximación inicial de las necesidades y posibles soluciones que se aplicarían a la plataforma SUMA. En este documento se destacan los problemas encontrados, las posibles soluciones, los componentes de la plataforma SUMA que serían afectados, y los casos de uso que se desarrollarían.

Partiendo de la información en el documento de visión, se determinó que el problema principal se encontraba en la carga dinámica de clases y datos desde el cliente hacia el servidor, desde el momento en que se inicia la ejecución del programa. Este proceso implica ciertos retardos en la comunicación debido a que el proceso de ejecución es detenido cada vez que el agente de ejecución realiza un *callback*.

Para conseguir la solución a estos problemas se diseñó un nuevo diagrama de casos de uso, en donde se aprecian las nuevas incorporaciones al modelo original (ver figura IV.2). Los óvalos en fondo blanco representan los casos de uso actuales que presenta SUMA, mientras los que poseen fondo gris representan los casos de uso propuestos para la inclusión del

servicio de precarga. Para tener una idea más detallada sobre la descripción de los actores y de los casos de uso incorporados ver apéndice E.

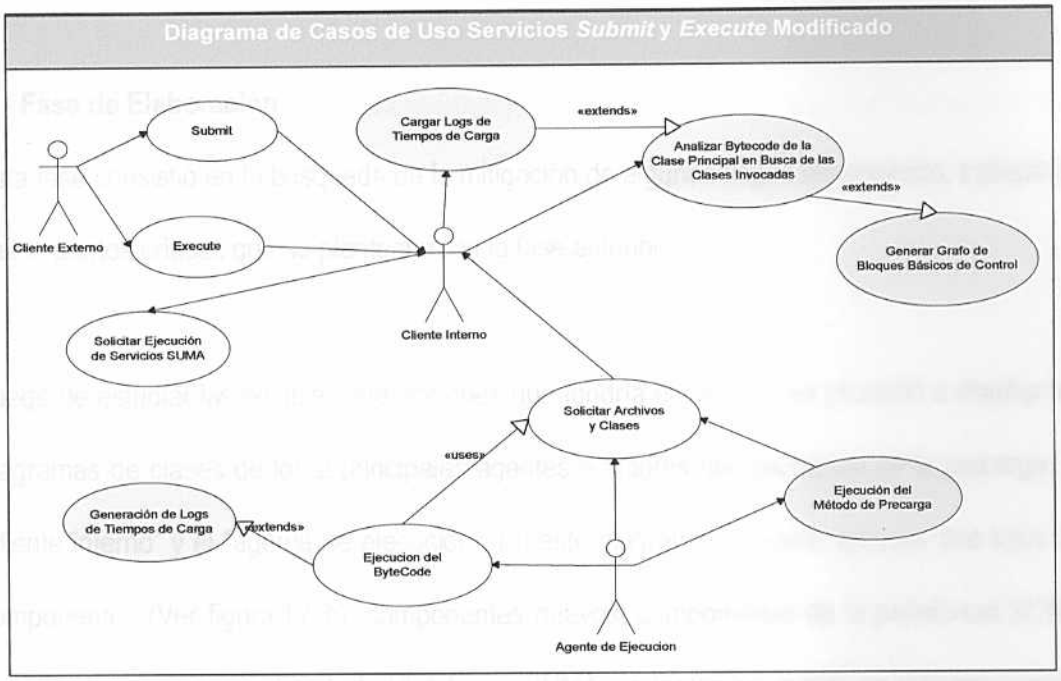

Figura IV.2. Diagrama de Casos de Uso Servicios *Submit y Execute* Modificados
Fuente: Elaboración propia.

Una vez desarrollado el documento visión y diseñado el diagrama de casos de uso, se decidió en conjunto con el Tutor cuál seria el caso de uso crítico. Este caso de uso representaría, el pilar fundamental del proyecto, por lo que tenía que ser planteado y desarrollado antes que cualquier otro. El caso de uso escogido fue "analizar b*ytecode* de la clase principal en busca de las clases invocadas". (Ver tabla 2 apéndice E).

Otro punto de gran importancia que se tomó en consideración durante esta fase fue lo referente a los riesgos que se podían correr a la hora de implementar la precarga de clases y

algunas posibles ideas de como mitigarlos. Estas consideraciones fueron estudiadas y plasmadas en la tabla 1 del apéndice F.

2. - Fase de Elaboración

Esta fase consistió en la búsqueda de la mitigación de algunos riegos del proyecto, trabajando en los puntos críticos que se plantearon en la fase anterior.

Luego de estudiar las posibles interacciones que tendría el sistema, se procedió a diseñar los diagramas de clases de los 2 principales agentes o actores que participan en la precarga; el "cliente interno" y el "agente de ejecución". En este diagrama se puede apreciar tres tipos de componentes (Ver figura IV.3.): componentes nuevos, componentes de la plataforma SUMA modificados y componentes de la Plataforma SUMA originales los cuales no sufrieron ningún cambio.

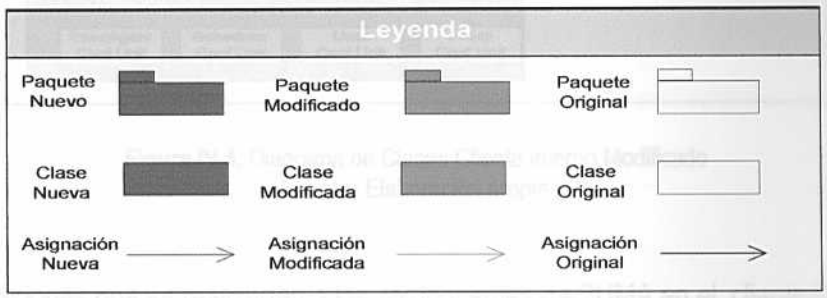

Figura IV.3. Leyenda Diagrama de Clases
Fuente: Elaboración propia.

En la figura IV.4 se pueden apreciar los nuevos componentes y las modificaciones que serían agregados en el cliente interno. Entre los nuevos componentes se destacan:

- El paquete *Prefetching*, donde estarían contenidas todas aquellas clases necesarias para extraer información de un *bytecode*.

- La estructura PG (*Prefetching Graph*), en donde se almacenaría información referente a la precarga, como nombre de la clase, tiempo de transferencia y tiempo de invocación.

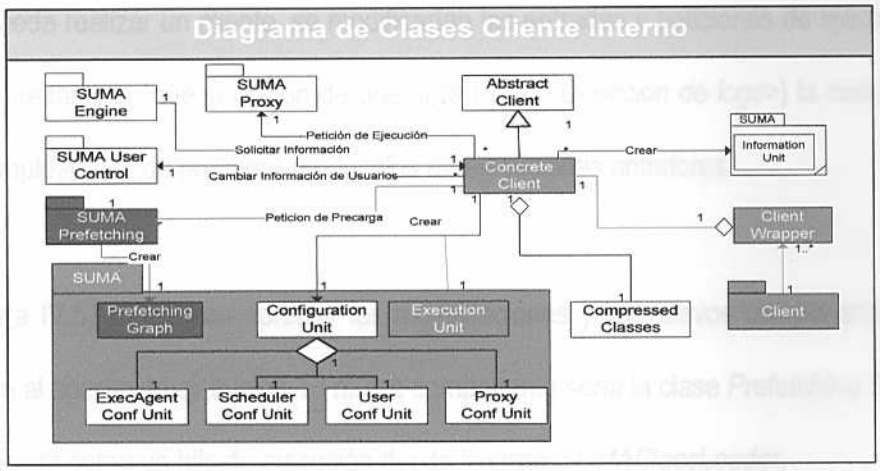

Figura IV.4. Diagrama de Clases Cliente Interno Modificado
Fuente: Elaboración propia.

Las modificaciones que se realizarían a los componentes de SUMA en el cliente interno son:

- En la clase *ConcreteClient*, donde se encuentra definido el comportamiento del cliente interno, se agregaría la invocación al paquete de *Prefetching* para extraer la

información de la clase principal que se desea ejecutar. Esta información sería almacenada en la estructura *Execution Unit* para ser enviada al agente de ejecución.

- En la estructura "EU" del paquete SUMA, en donde se encuentran los datos necesarios para realizar la ejecución de una aplicación o programa, se agregaría un nuevo atributo para almacenar el grafo de precarga.

- En el paquete *Client*, en donde están definidas las instrucciones o comandos que puede realizar un cliente, se modificarían las entradas y peticiones de ejecución para ingresar una nueva opción de precarga (*-pref <Direccion de logs>*) la cual permitiría emplear *logs* de precarga producidos en ejecuciones anteriores.

En la figura IV.5, se pueden apreciar las modificaciones y los nuevos componentes que se agregarían al agente de ejecución. El nuevo componente sería la clase *Prefetching*. Esta clase sería invocada como un hilo de ejecución desde la clase *SUMAClassLoader*.

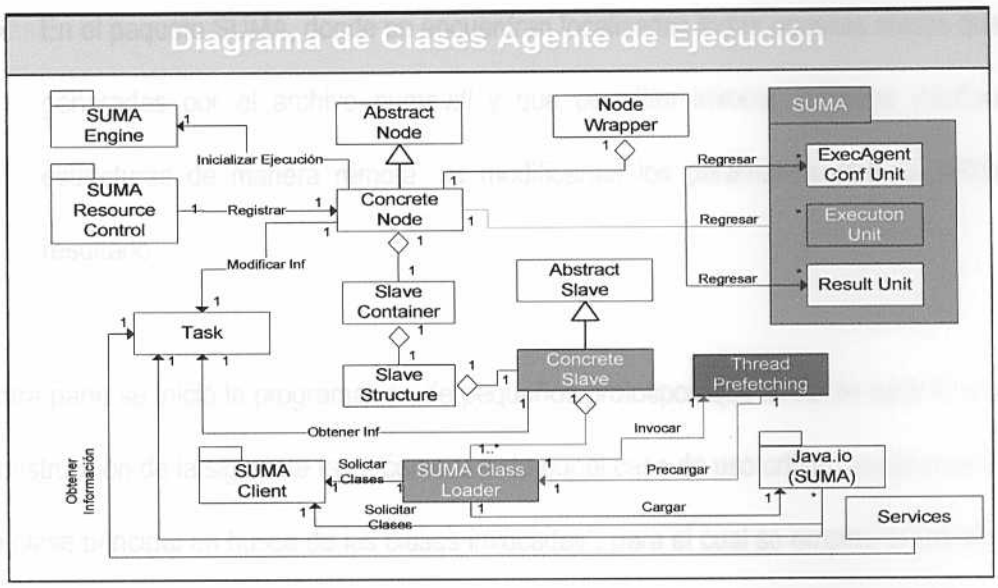

Figura IV.5. Diagrama de Clases Agente de Ejecución Modificado
Fuente: Elaboración propia.

Por otra parte las modificaciones que se realizarían en el agente de ejecución son:

- En la clase *ConcreteSlave*, donde se encuentra definido el comportamiento de un agente de ejecución en modo secuencial, se modificaría la invocación al *classloader*, para agregar como nuevo parámetro el grafo con las clases precargadas.

- En la clase *SUMAClassLoader*, donde se realiza la carga de todas aquellas clases y archivos necesarios para la ejecución de una aplicación, se invocaría un hilo de ejecución que estaría en comunicación constante con el *classloader*, a modo de precargar las clases que se encuentren el grafo.

- En el paquete SUMA, donde se encuentran localizadas todas aquellas clases que son generadas por el archivo *suma.idl* y que permiten invocar métodos interfaces y estructuras de manera remota, se modificarían los parámetros para la salida del resultado.

Por otra parte se inició la programación de pequeños prototipos que sirvieran para el proceso de construcción de la siguiente fase, comenzando por el caso de uso crítico "analizar b*ytecode* de la clase principal en busca de las clases invocadas", para el cual se empleó la herramienta "*Espresso*"[3]. Para conocer con más detalle los pasos a seguir para la construcción de prototipos ver tabla 1 apéndice G.

Adicionalmente se desarrollaron algunos programas de prueba que sirvieron para despejar algunas dudas sobre la manera en que serían precargadas las clases. En vista de que se quería que la precarga corriera paralelamente a la ejecución normal de la plataforma SUMA, se crearon varios programas que trabajasen con diferentes hilos de ejecución que pudieran hacer invocaciones y métodos de la clase principal. Durante estas ejecuciones se encontraron algunos problemas con relación a la disponibilidad de algunos atributos y funciones, ya que la mayoría de los mismos se encontraban definidos como privados y al momento de ser invocados desde el hilo de ejecución generaban un error. En definitiva, estas pruebas ayudaron entonar la implementación.

[3] Espresso: Herramienta que permite analizar e interpretar un *bytecode* de Java. [16]

3. - Fase de Construcción

En esta fase se tomaron los prototipos construidos en la fase anterior con respecto al cliente interno y al agente de ejecución y se modificaron de tal manera de poder adaptarlos a la plataforma SUMA.

En primer lugar la herramienta "e*spresso*" fue modificada y se agregó como un nuevo paquete a SUMA, denominado *sumaPrefetching*. Este paquete es el que permitirá realizar el análisis de *bytecode* de Java. A este paquete se le incorporo la clase *Prefetching*, la cual consta de métodos destinados al procesamiento, generación, y almacenamiento de la información obtenida de un *bytecode* específico. Gracias a estos métodos, se puede obtener una estructura con la información de las clases invocadas en un *bytecode*. Dicha estructura es almacenada de acuerdo al orden de invocación de las clases y es llenada con valores iniciales referentes a los tiempos de precarga.

Por otra parte, para el manejo de las estructuras necesarias en todo el proceso de precarga, se creó dentro del archivo *idl* de suma, la estructura PG (*Prefetching Graph*), en la que se almacena información referente a la precarga. Adicionalmente la estructura EU (e*xecution unit*) fue modificada agregando un nuevo atributo, en donde se almacenaría el grafo de las clases precargadas generado en la clase *Prefetching*.

Dentro de la clase *concrteClient*, se realizaron las respectivas modificaciones para ejecutar la precarga de clases. Estos cambios consistieron en: creación de nuevos métodos para interactuar con la clase *Prefetching*, inicialización y carga de estructuras de control, carga de *logs* de precarga partiendo de una dirección dada por el cliente, así como también el envío del grafo de precarga al agente de ejecución para su interpretación y posterior ejecución.

Los *logs* de precarga contienen información referente a los tiempos de transferencia y de invocación de las clases que participaron en la ejecución pasada de alguna aplicación o programa. De existir estos *logs*, y de ser suministrado por el cliente externo la dirección de los mismos, la precarga de clases no se realizaría, sino por el contrarió, se cargaría el grafo de precarga con la información contenida en el archivo.

En lo que se refiere a los cambios del lado del agente de ejecución, todas aquellas clases que invocaban a la clase *SUMAClassLoader* fueron modificadas debido a la necesidad de enviar un nuevo parámetro (grafo de bloques básicos de la precarga) a través de una invocación o llamada. Dentro de la clase *SUMAClassLoader* se creó una nueva clase denotada *ThreadPrefetching*, la cual funcionaría como un hilo de ejecución y estaría encargada de realizar llamadas al método *loadClass()* (perteneciente a la clase *SUMAClassLoader*) con la finalidad de cargar las clases almacenadas dentro del grafo de precarga.

Por otra parte se tomaron medidas de los tiempos antes y después de cargar una clase desde el cliente, de tal manera de poder calcular los tiempos en que tarda una clase en ser transferida (tiempo de transferencia) y el tiempo en que tarda una clase en ser llamada (tiempo de invocación). Estos valores se almacenaron dentro del mismo grafo donde era extraída la información, de tal manera de poder retornarlo al cliente.

Una vez retornado el resultado al cliente interno, se procede a almacenar la información contenida el grafo de precarga en el *log* correspondiente para la clase principal de la aplicación o programa que se ejecuto. Seguidamente, el resultado de la ejecución es regresado al cliente externo.

En la figura IV.6 se muestra el diagrama de secuencia que representa la ejecución paso a paso del módulo de precarga luego de la construcción de sus componentes. Para poder tener una descripción con mas detalle de dicha ejecución ver la tabla IV.1.

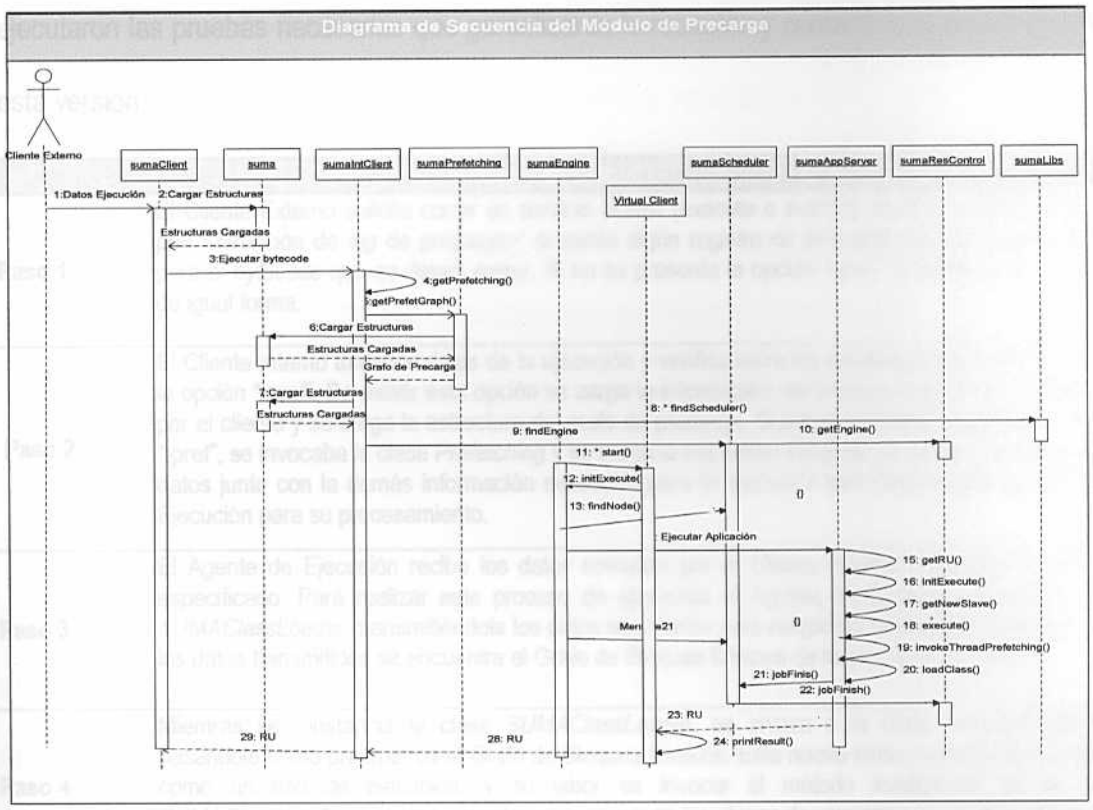

Figura IV.6. Diagrama de Secuencia Módulo de Precarga
Fuente: Elaboración propia.

4. - Fase de Transición

En esta última fase de la metodología se procedió a implantar en la Universidad Simón Bolívar la nueva versión de la plataforma SUMA desarrollada en las fases anteriores, Dicha versión no ocasionaba ningún conflicto con la que se esta ejecutando actualmente, debido a que se instaló en una cuenta de usuario independiente creada para tal fin .

Luego de modificar los archivos de configuración (*myprofile, profile, .orbacusproperties*); para adaptar la nueva versión a las condiciones operativas presentes en dicho ambiente; se

ejecutaron las pruebas necesarias que garantizaran la robustez y correcto funcionamiento de esta versión.

Proceso de Ejecución	
Paso 1	El Cliente Externo solicita correr un servicio SUMA (*execute* o *submit*), especificando la opción "-pref <dirección de *log* de precarga>" si existe algún registro de precarga utilizado anteriormente para el *bytecode* que se desea correr. Si no se presenta la opción "-pref" la precarga se realizará de igual forma.
Paso 2	El Cliente Interno toma los datos de la ejecución y verifica entre los parámetros enviados, si existe la opción "-pref". De existir esta opción se carga la información del archivo ubicado en la ruta dada por el cliente y se carga la estructura del grafo de precarga. Si por el contrario no existía la opción "-pref", se invocaba la clase *Prefetching* y se carga la estructura del grafo de precarga. Luego estos datos junto con la demás información necesaria para la ejecución son transmitidas al Agente de Ejecución para su procesamiento.
Paso 3	El Agente de Ejecución recibe los datos enviados por el Cliente Interno y ejecuta el *bytecode* especificado. Para realizar este proceso de ejecución el Agente de Ejecución invoca la clase *SUMAClassLoader*, transmitiéndole los datos necesarios para cargar las clases que necesite. Entre los datos transmitidos se encuentra el Grafo de Bloques Básicos de la precarga de clases.
Paso 4	Mientras se instancia la clase *SUMAClassLoader*, se invoca a la clase *ThreadPrefetching* pasándole como parámetros el Grafo de Bloques Básicos. Esta nueva clase invocada se comporta como un hilo de ejecución, y su labor es invocar el método *loadClass()* de la clase *SUMAClassLoader* de tal manera de ir precargando las clases de una manera paralela al proceso normal.
Paso 5	Para invocar el método *loadClass* se recorre el Grafo de Bloques Básicos y dependiendo de los valores de los tiempos de transferencia y de invocación de una clase, se decide si se realiza la precarga o no, de dicha clase. La regla empleada para determinar si se precarga una clase o no, es determinar si el tiempo de invocación de una clase es menor al tiempo de transferencia de la misma se procede a realizar la precarga. Si por el contrario, el tiempo de invocación de una clase es mayor al tiempo de transferencia, no se realiza la precarga, y la clase será carga según el proceso normal.
Paso 6	Durante el proceso de carga de una clase te toman medidas de los tiempos antes y después de realizar la transmisión de una clase desde el cliente hasta el agente de ejecución, de tal manera de poder calcular nuevos tiempos de transferencia y de invocación de cada clase. Seguidamente el Grafo de Bloques Básicos es actualizado con los nuevos valores y el mismo es retornado en una de las estructuras de control al Cliente Interno.
Paso 7	Una vez retornado el resultado de la ejecución, el Cliente Interno, toma el Grafo de Bloques Básicos y lo almacena en un archivo de *logs* de precarga. Si el archivo ya existía se sobrescribe la información.
Paso 8	Por ultimo el Cliente Interno retorna los resultados obtenidos al cliente.

Tabla IV.1. Proceso de Ejecución del Módulo de Precarga
Fuente: Elaboración propia.

Capítulo V. Desarrollo del Módulo de *Buffering* Para un *Grid* Computacional

Para el desarrollo del módulo de *buffering* y *caching* de datos se realizó un estudio siguiendo la metodología RUP, con el cual se pudiese determinar algunas de las técnicas mas usadas en la actualidad, para luego dejar planteada una propuesta para el mejoramiento de esta sección. Paralelo a la realización de la propuesta, se implantaría en la plataforma SUMA una de las mejoras estudiadas. El plan de trabajo que se siguió a través del desarrollo de este módulo se puede apreciar en la tabla 2 del apéndice A.

1. - Fase de Inicio

Las primeras actividades consistieron en levantamiento de información por medio de entrevistas, y búsqueda de material referente al tema de *buffering* de datos. Las entrevistas fueron llevadas a cabo de la misma forma que en la fase de inicio del desarrollo del módulo de precarga y gracias a ese trabajo sólo hubo que profundizar con lo referente al manejo de *buffering* de de la plataforma SUMA; sus características, funciones, estructura y sobre todo su alcance.

Al igual que en el desarrollo anterior, los recursos de software y hardware necesarios para desarrollar el módulo de *buffering* pueden ser visualizados en el apéndice C.

En base a esta primera información, se realizó un documento de visión (ver tabla 2 apéndice D), con el cual se pudo tener una aproximación inicial de las necesidades y posibles soluciones que se aplicarían a la plataforma SUMA. En este documento se destacan los

problemas encontrados, las posibles soluciones, los componentes de la plataforma SUMA que serian afectados, y los casos de uso que se desarrollarían.

Tomando como referencia el documento de visión, se pudo notar que durante el proceso de lectura del *buffer* de datos de la plataforma SUMA no se detecta si un archivo ha sido modificado, por lo que se podría estar leyendo información desactualizada e incluso errónea. Para tratar de corregir esta situación se diseñó un diagrama de casos de uso, con las posibles soluciones a este problema, el cual se puede apreciar en la figura V.1. En este diagrama se identificaron los actores involucrados, y las funciones o tareas que estos desempeñan. Los óvalos en fondo blanco representan los casos de uso actuales que presenta SUMA, mientras los que poseen fondo gris representan los casos de uso propuestos. Para tener una idea mas detallada sobre descripción de los actores y de los casos de uso incorporados ver apéndice H.

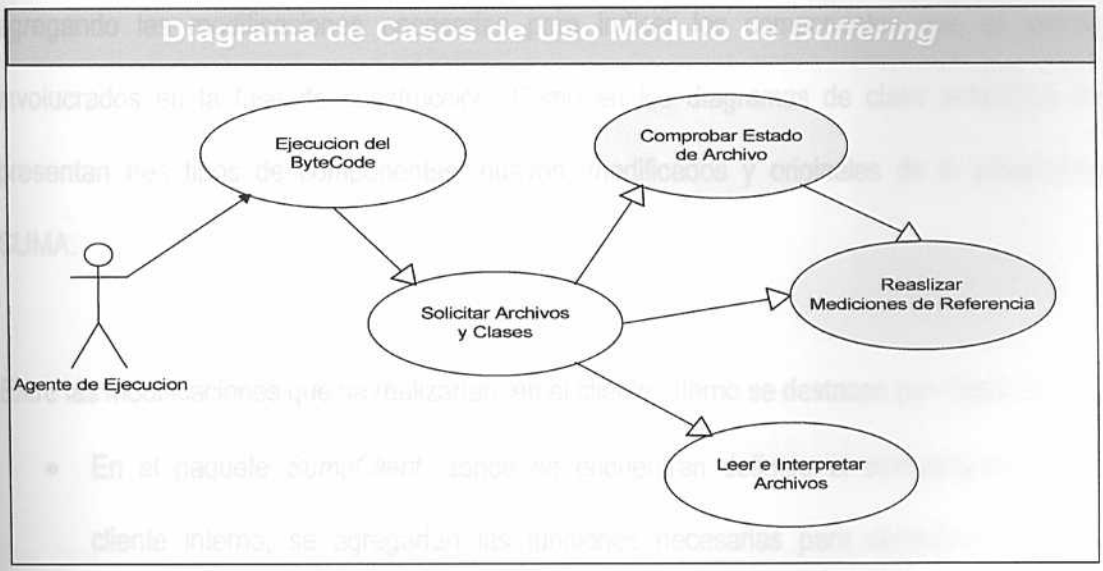

Figura V.1. Diagrama de Casos de Uso Módulo de *Buffering*
Fuente: Elaboración Propia.

El caso de uso crítico escogido para ser desarrollado en primer lugar fue "realizar mediciones de referencia". A través de este caso de uso se definió la estrategia que se emplearía durante el desarrollo de este módulo. Ver tabla 2 del apéndice H.

Por otra parte, se determinaron algunos riesgos por los que se podría atravesar si se llegara a implantar el modulo de *buffering* y *caching* de datos, además de las posibles acciones que se podrían tomar para mitigar estos riesgos. En la tabla 2 del apéndice F se presentan los principales riesgos y mitigaciones encontrados.

2. - Fase de Elaboración

Durante esta fase de elaboración se desarrollo el diagrama de clases del agente de ejecución tomando como referencia el diagrama realizado durante el desarrollo de la precarga de clases, agregando las modificaciones necesarias para indicar los componentes que se verían involucrados en la fase de construcción. Como en los diagramas de clase anteriores se presentan tres tipos de componentes, nuevos, modificados y originales de la plataforma SUMA.

Entre las modificaciones que se realizarían en el cliente interno se destacan (ver figura V.2):

- En el paquete *SumaClient*, donde se encuentran definido el comportamiento del cliente interno, se agregarían las funciones necesarias para determinar algunas

propiedades de un archivo, en particular el tiempo de la última actualización y el tamaño del mismo.

- En el paquete SUMA; donde se encuentran localizadas todas aquellas clases que son generadas por el *IDL* de SUMA y que sirven para invocar métodos, interfaces y estructuras de manera remota; se encontrarían los métodos agregados en el paquete anterior.

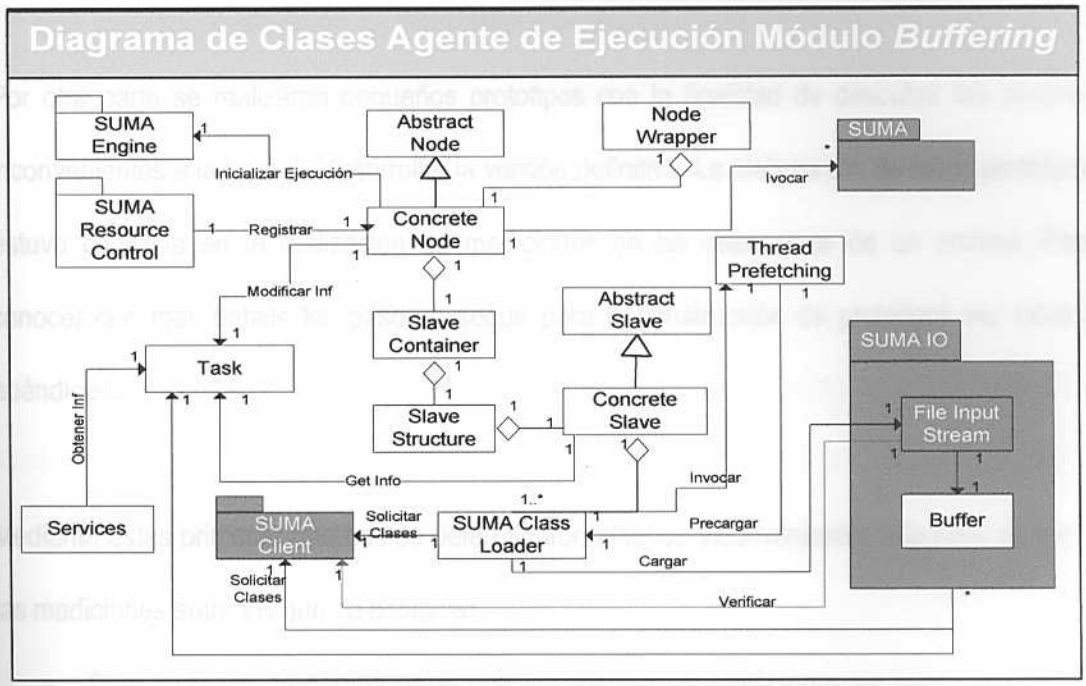

Figura V.2. Diagrama de Clases Agente de Ejecución Módulo *Buffering*
Fuente: Elaboración propia

Por otra parte las modificaciones que se realizarían a los componentes del agente de ejecución son:

- En el paquete *SumaIO*, que resulta de una redefinición del paquete *java.io*, se realizarían las invocaciones de las funciones desarrolladas en el paquete *SumaClient*, que permitirían detener la ejecución de la aplicación en el caso de realizarse una lectura de datos erróneos.

- En la clase *File*, donde se definen todos aquellos métodos que son aplicables a un archivo, se agregarían las implementaciones de aquellas clases definidas en el paquete SUMA.

Por otra parte se realizaron pequeños prototipos con la finalidad de descubrir las posibles inconvenientes a la hora de desarrollar la versión definitiva. La elaboración de estos prototipos estuvo centrada en la realización de mediciones de las referencias de un archivo. Para conocer con más detalle los pasos a seguir para la construcción de prototipos ver tabla 2 apéndice G.

Mediante estas primeras pruebas se determinaron algunos inconvenientes a la hora de hacer las mediciones entre los que se destacan:

- Si el intervalo de tiempo en el que se realizan las mediciones es muy corto, no es posible determinar la fecha de la última actualización.

- Si las modificaciones que se realizan en el archivo no alteran el tamaño original, no es posible determinar si hubo modificaciones en el archivo.

Gracias a las pruebas realizadas se determinó practicar mediciones basándose en ambas características (tiempo de última modificación y tamaño del archivo), a manera de garantizar la presencia de una modificación efectiva.

3. - Fase de Construcción

En esta fase de construcción se desarrolló una mejora del *buffer* de la plataforma SUMA, modificando algunos de sus principales componentes y agregando nuevas funcionalidades.

En primer lugar se desarrollaron los métodos necesarios para realizar las mediciones y comprobaciones necesarias que determinaran si un archivo había sido modificado. Estos métodos fueron declarados dentro del archivo *suma.idl*, específicamente en la interfaz *AbstractRemoteClient* de tal manera de que pudieran ser invocados de manera remota. El desarrollo de estos métodos fue agregado dentro de la clase *ConcreteClient*, manteniendo la estructura de otros métodos dentro de la misma clase.

Dentro del constructor de la clase *SUMAInputStream* se agregaron nuevas variables referentes al tiempo de la última modificación de un archivo y su tamaño original, obtenidos mediante los métodos creados en el paso anterior. Dentro de esta misma clase se agregaron

invocaciones al método *isModified*; que se encuentra desarrollado en la clase *File*; antes de realizar la llamada al *buffer* de lectura. Si el archivo había sido modificado durante el proceso de lectura, se lanza una excepción indicando que se puede estar trabajando sobre un archivo corrupto o erróneo.

En la figura V.3 se puede apreciar la secuencia final a través del tiempo y la interacción entre los componentes que forman parte del módulo de *buffering* de datos.

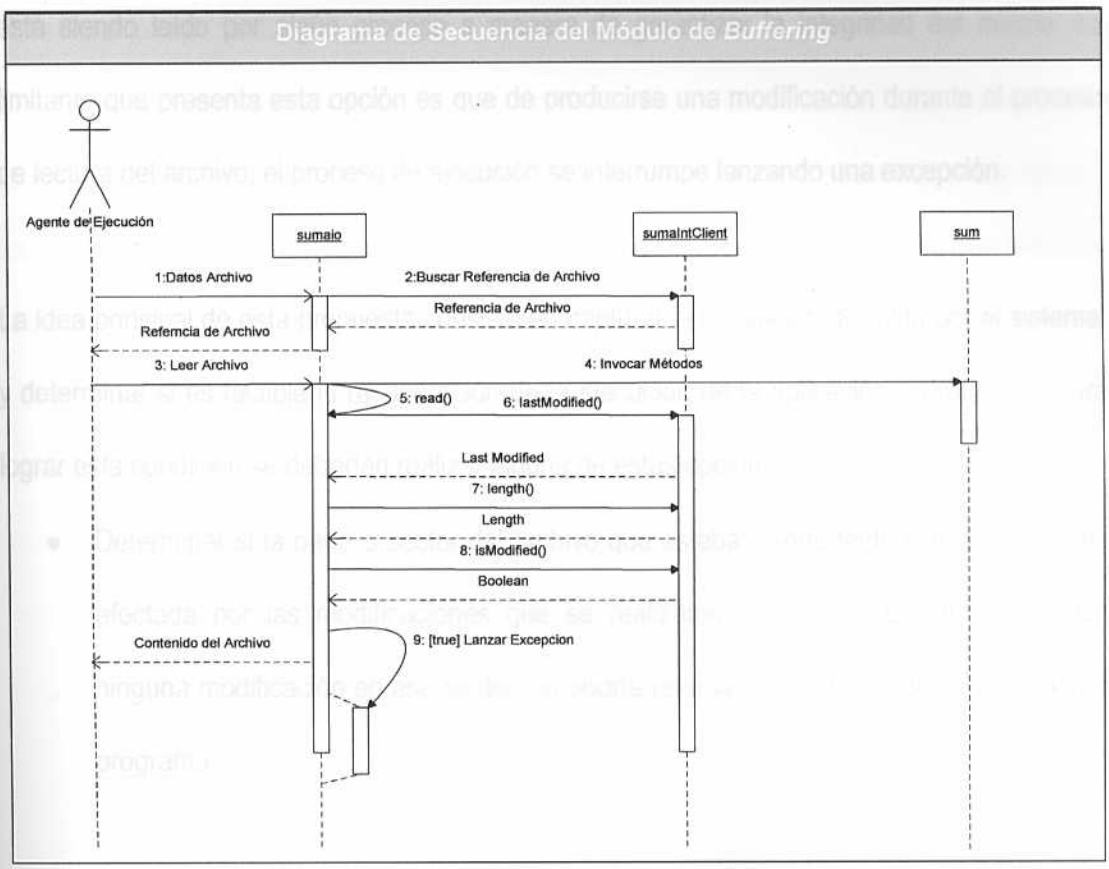

Figura V.3. Diagrama de Secuencia Módulo *Buffering*
Fuente: Elaboración propia.

Para finalizar con esta fase de construcción se decidió elaborar una propuesta para un mejor manejo del *buffering* y *caching* de datos la cual brinde un mejor rendimiento a la plataforma SUMA. Entre las principales ideas que se podrían implementar se encuentran las siguientes:

3.1 - Propuesta: Recuperación de la excepción

Tomando como referencia el módulo de *buffering* desarrollado en la iteración anterior, se presenta una opción en donde se verifica el estado actual de un archivo mientras el mismo está siendo leído por algún proceso a manera de garantizar la integridad del mismo. La limitante que presenta esta opción es que de producirse una modificación durante el proceso de lectura del archivo, el proceso de ejecución se interrumpe lanzando una excepción.

La idea principal de esta propuesta consiste en capturar la excepción arrojada por el sistema, y determinar si es factible la recuperación de la ejecución de la aplicación o programa. Para lograr esta condición se deberían realizar alguna de estas opciones:

- Determinar si la parte o sector del archivo que estaba siendo leído estuvo realmente afectada por las modificaciones que se realizaron al archivo. De no encontrase ninguna modificación en ese sector, se podría retomar la ejecución de la aplicación o programa.

- Si el sector del archivo que se está leyendo resulta alterado por algún otro proceso, se podría deshacer o borrar el *buffer* que se tenga almacenado hasta ese momento y retornar la lectura desde un sector anterior que se encuentre libre de modificaciones.
- Una opción más drástica podría ser deshacer o vaciar todo lo que se ha leído hasta el momento de producirse la excepción y volver a leer e interpretar el archivo.

3.2 - Propuesta: Diseño de un buffer optimista - por tablas

En esta segunda propuesta se plantea implantar un nuevo diseño de *buffering* de datos, en esta oportunidad basándose en dos técnicas conocidas de *buffering*, "optimista" y "por tablas". La idea es aprovechar las ventajas de ambas técnicas y así obtener un híbrido que presente sus características y funcionalidades. El diseño de este *buffer* estaría regido por lo siguiente:

- Todos los registros ubicados en el *buffer* se encuentran habilitados.
- Se permite la modificación simultánea de múltiples registros ubicados en el *buffer*.
- El bloqueo se produce sólo cuando efectivamente se van a guardar las modificaciones en el disco y no en el *buffer*.

4. - Fase de Transición

En esta fase de transición se implanto la nueva versión de la plataforma SUMA que consta del módulo de *buffering* de datos y que fue desarrollado durante las fases anteriores.

Seguidamente se realizaron las pruebas correspondientes y se dieron las indicaciones necesarias para su buen uso y mantenimiento. Esta nueva versión de la plataforma SUMA desarrollada durante las fases anteriores con las nuevas implementaciones sobre el *buffering* y *caching* de datos se implantó en una cuenta de usuario independiente, la cual no ocasionaba ningún conflicto con la versión que estaba funcionando en esos momentos.

Para comenzar a realizar las pruebas se modificaron los archivos *.orbacusproperties, profile y myprofile,* que eran necesarios para adaptar la nueva versión a las condiciones operativas al nuevo ambiente.

Luego de constatar que la versión se ejecutaba correctamente, se entregó al personal de SUMA las especificaciones de la misma así como la propuesta para el desarrollo de un mejor módulo de *buffering* y *caching* de datos.

Capítulo VI. Elaboración de Pruebas

Durante el desarrollo de este trabajo especial de grado se realizaron diversas pruebas con la finalidad de determinar si las implementaciones y modificaciones que se practicaron funcionaban con normalidad arrojando los resultados esperados.

1. - Ambientes de pruebas

Las pruebas fueron realizadas en diferentes ambientes, dependiendo de la magnitud de la prueba y de los resultados que se querían obtener. Dependiendo del ambiente en donde se encontrara, las modificaciones necesarias y aspectos a tomar en cuenta variaban.

1.1. - Hardware y Software

El equipo donde se realizaron las pruebas fue *pentium* 4 de 1.7*Mhz*, 256*MB* de RAM y con un disco duro de 40*GB*. El Sistema Operativo era *Suse 9.0*.

1.2. - Escenarios

Con la finalidad de ver el comportamiento del sistema con los nuevos módulos desarrollados, fue necesario llevar a cabo las pruebas en tres escenarios distintos que fueron:

- **Aplicación Local**

  En las pruebas de aplicación local la plataforma SUMA así como el resto de componentes y herramientas se encontraban en una misma máquina, por lo que los archivos de configuración debían ser modificados para que apuntaran a la misma máquina.

- **Aplicación Distribuida**

  Las pruebas de aplicación distribuida consistieron en colocar algunos de los componentes de la plataforma SUMA en diferentes máquinas conectadas en red, logrando así un ambiente distribuido.

- **Aplicación Remota**

  Para este tipo de prueba los componentes de la plataforma SUMA eran implantados en una misma máquina, mientras que el usuario realizaba la conexión desde otra, por lo que los archivos de configuración debían ser modificados para que apuntaran a la máquina en donde se encontraba implantada la versión a probar.

2. - Tipos de pruebas

Dependiendo de la fase de la metodología se practicaron diferentes tipos de pruebas, las cuales realizaban actividades diferentes pero buscando cumplir el mismo objetivo, el de garantizar la eficiencia de lo que se había desarrollado e implantado.

2.1. - Pruebas de funcionalidad

Las pruebas de funcionalidad se realizaron con la finalidad de verificar si el código desarrollado era correcto, tanto a nivel de prototipos como a nivel de la nueva plataforma creada. La mayor parte de estas pruebas fueron realizadas basándose en un ambiente de

aplicación local y utilizando aplicaciones sintéticas, debido a la facilidad para realizar cualquier modificación en caso de no obtener el resultado deseado.

El mayor uso de estas pruebas durante el desarrollo del módulo de precarga estuvo en:

- Determinar las clases contenidas en un *bytecode*.
- Formar un grafo de clases precargadas siguiendo el orden de invocación de las mismas.
- Transmitir el grafo desde el cliente interno al agente de ejecución.
- Manejar hilos de ejecución que emplearan propiedades y métodos de otras clases.
- Crear hilos de ejecución para extraer la información del grafo de clases para realizar la precarga.
- Crear, inicializar, llenar y transmitir la estructura de datos con información referente a la precarga de clases. Dichas estructuras poseen el nombre de la clase y los tiempos de transferencia e invocación.
- Calcular y almacenar tiempos de transferencia y de invocación para cada clase.
- Cargar y almacenar información sobre el resultado de la precarga de clase.

Con respecto al módulo de *buffering* se realizaron las siguientes pruebas:

- Comprobar características que demuestren la ocurrencia de algún cambio en la información contenida en un archivo.
- Invocar métodos desde el agente de ejecución, ubicados en el cliente interno.

- Lanzar excepciones luego de producirse alguna modificación sobre un archivo que esta siendo leído.

2.2. - Pruebas de validación y desempeño

En estas pruebas de validación y diseño se buscaba determinar si lo que se había desarrollado se ejecutaba de una manera eficaz, es decir, si se cumplían los resultados esperados de la mejor manera posible, y dentro del rango de valores esperado.

Para este tipo de pruebas se utilizó un *benchmark*, que se encuentra en el foro de Java llamado *Java Grande Forum* [22]. Con la ejecución de dicho *benchmark* se pudo medir el desempeño del nuevo módulo de precarga incorporado a SUMA.

Las pruebas de validación y desempeño dentro del módulo de precarga de clases estuvieron centradas en:

- Verificar la correcta ejecución de la precarga de clases.
- Determinar si los tiempos de ejecución habían disminuido con respecto a las fases anteriores.
- Comprobar el buen desempeño en diferentes ambientes de prueba.

Con respecto al módulo de *buffering* se realizaron las siguientes pruebas:

- Verificar la correcta ejecución del proceso de *buffering* de datos.
- Comprobar el buen desempeño en varios ambientes de pruebas.

3. - Resultados de las pruebas de desempeño

Para poder medir el desempeño del modulo de precarga de clases se tomaron medidas de los tiempos de ejecución de la nueva versión de SUMA con *prefetching* contra la versión anterior. Estas pruebas fueron realizadas en ambiente local, y con la ayuda del *benchmark* de "Java Grande".

En la tabla VI.1 se pueden apreciar los tiempos arrojados en dichas pruebas, así como también el porcentaje de mejora por cada clase *benchmark* ejecutado, encontrando diferencias en los porcentajes de mejora según el número de clases de tipo cliente (que no sean de Java o de SUMA) que emplee dicho *benchmark*.

Según los resultados obtenidos se puede notar que los *benchmark* que presentan pocas clases de tipo cliente tienden a tener un bajo incremento porcentual de mejora, e incluso en algunos casos se presentan resultados contrarios a los esperados, incrementándose el tiempo de ejecución en la versión de SUMA con precarga (con o sin el uso de *logs*), y llegando a obtener valores negativos de un 3,58%. Sin embargo, el porcentaje de mejora la versión de SUMA con precarga y *logs* resultó superior en todas las mediciones alcanzando su máximo en un 31,5%, lo cual indica que el uso de registros minimiza los tiempos de ejecución de los programas o aplicaciones que se desean procesar.

Clase Ejecutada	Nº de Clases Tipo Cliente	Tiempos sin *Precarga (mseg)*	Tiempos con *Precarga* (mseg)	*% de Mejora*	Tiempos con *Precarga y Logs* (mseg)	*% de Mejora*
PruebaAA	6	2562	1881	26,58	1755	31,5
JGFCryptBenchSizeA	6	5226	5401	-3,35	5201	0,48
JGFFFTBenchSizeA	6	21905	21426	2,19	21058	3,87
JGFHeapSortBenchSizeA	6	3260	3163	2,98	3047	6,53
JGFLUFactBenchSizaA	6	2140	2185	-2,1	2153	-0,61
JGFSeriesBenchSizeA	6	45862	47502	-3,58	46953	-2,38
JGFSORBenchsizeA	6	5383	5236	2,73	5197	3,46
JGFSparseMatmultBench SizeA	6	15773	15740	0,21	15641	0,84
JGFAllSizeA	18	120949	109201	9,71	97849	19,1
JGFArithBench	4	365224	377025	-3,23	361426	1,04
JGFAssignBench	5	288775	265783	7,96	257096	10,97
JGFCastBench	4	116464	103257	11,34	101246	13,07
JGFCreateBench	14	1210957	1123608	7,21	1072499	11,43
Total	93	2204480	2081408	5,58	1991121	9,68

Tabla VI.1. Tiempos de la Prueba de Desempeño en Ambiente Local
Fuente: Elaboración Propia

En la tabla VI.2 se puede apreciar un análisis más detallado de uno de los *becnhmark* que posee más clases de tipo cliente (18 clases en total). En dicha tabla se observan las diferencias que existen en los tiempos de invocación de cada una de las clases que serán empleadas por el *benchmark* tanto en las versiones de SUMA con y sin precarga. Los resultados obtenidos en dicho análisis .evidencian una notable disminución del tiempo en que

son invocadas las clases, obteniendo una mejora de 96,56% y 99,65% en las versiones de SUMA con precarga y la versión con precarga y *logs* respectivamente.

Clases de *JGFAllSizeA*	Tiempo Invocación sin *Precarga* (mseg)	Tiempo invocación con *Precarga* (mseg)	% de *Mejora*	Tiempos invocación con *Precarga y Logs* (mseg)	% de *Mejora*
JGFAllSizeA	0	0	0	0	0
jgfutil.JGFInstrumentor	55	14	74,55	23	58,18
series.JGFSeriesBench	31	30	3,23	16	48,39
series.SeriesTest	15	11	26,67	16	-6,67
jgfutil.JGFSection2	11	7	36,36	9	18,18
jgfutil.JGFTimer	16	40	-150	20	-25
lufact.JGFLUFactBench	48036	30	99,94	21	99,96
lufact.Linpack	13	19	-46,15	20	-53,85
heapsort.JGFHeapSortBench	851	16	98,12	20	97,65
heapsort.NumericSortTest	13	8	38,46	11	15,38
crypt.JGFCryptBench	1787	12	99,33	11	99,38
crypt.IDEATest	9	15	-66,67	11	-22,22
fft.JGFFFTBench	4013	22	99,45	14	99,65
fft.FFT	13	9	30,77	10	23,08
sor.JGFSORBench	21266	27	99,87	16	99,92
sor.SOR	11	38	-245,45	16	-45,45
sparsematmult.JGFSparseMatmultBench	4020	24	99,4	19	99,53
sparsematmult.SparseMatmult	16	33	-106,25	24	-50
Total	80176	355	99,56	277	99,65

Tabla VI.2. Tiempos de Invocación de la Clase *JGFAllSizeA*
Fuente: Elaboración Propia

En la tabla VI.3 se analizan los tiempos de invocación del *benchmark* que presentaba menos clases de tipo cliente (4 clases en total), obteniendo un resultado similar al análisis anterior, en donde los tiempos en que se invocan las clases disminuyeron en la versión de SUMA con precarga y por consiguiente se obtuvo una mejora porcentual, alcanzando valores de un 55,56%.

Clases de *JGFArithBench*	Tiempo Invocación sin *Precarga (mseg)*	Tiempo invocación con *Precarga* (mseg)	*% de Mejora*	Tiempos invocación con *Precarga y Logs* (mseg)	*% de Mejora*
JGFArithBench	0	0	0	0	0
jgfutil.JGFSection1	79	31	60,76	30	62,03
jgfutil.JGFInstrumentor	16	17	-6,25	12	25
jgfutil.JGFTimer	31	20	35,48	14	54,84
Total	126	68	46,03	56	55,56

Tabla VI.3. Tiempos de Invocación de la Clase *JGFArithBench*
Fuente: Elaboración Propia

Luego de estas pruebas se pudo notar que existe una diferencia a la hora de aplicar la precarga de clases, dependiendo del número de clases que posea la aplicación o programa que se desea ejecutar, obteniendo mejores resultados en aquellos en donde se cuenta con un mayor número de clases para ser cargadas desde el cliente hasta el agente de ejecución.

Capítulo VII. Conclusiones

1. Una manera de obtener una potencia mayor de cálculo; sin necesidad de adquirir equipos costosos de última tecnología; es mediante el uso de G*rids* computacionales, los cuales permiten que los usuarios se beneficien de la potencia de ordenadores infrautilizados que se encuentran dispersos geográficamente.

2. El rendimiento de un programa o aplicación depende de muchos factores tanto internos como externos. Este rendimiento es traducido por muchas empresas en términos de ganancias o pérdidas, por lo que se ven en la necesidad de invertir grandes cantidades de dinero en la búsqueda de nuevas técnicas y estrategias que garanticen un mayor desempeño en sus aplicaciones.

3. La plataforma SUMA es un *Grid* computacional creada para las ejecuciones de programas Java, tanto secuenciales como paralelas. Se encarga de la administración de recursos distribuidos geográficamente y de la ejecución de *bytecodes* de Java, permitiendo a los usuarios una transparencia de localidad de las aplicaciones que sean ejecutadas en el sistema.

4. Mediante la metodología RUP; cuyo objetivo principal es el de asegurar la producción de software de calidad dentro de plazos y presupuestos predecibles; se desarrollaron dos nuevos módulos (precarga y b*uffering*) para ser integrados a la plataforma SUMA con la finalidad de obtener un mejor desempeño de la misma.

5. A través del módulo de precarga se logró transferir por adelantado todas aquellas clases que eran necesarias para la ejecución de un *bytecode,* minimizando las detenciones del proceso de ejecución por falta de alguna clase procedente del cliente, y disminuyendo los tiempos de ejecución de la plataforma SUMA.

6. La precarga de clases posee un mejor rendimiento cuando el programa o aplicación que se desea ejecutar presenta un mayor número de clases, mostrando poca disminución de los tiempos de ejecución e inclusive un aumento de los mismos cuando el número de clases a cargar es muy pequeño.

7. Mediante el uso de historiales de precarga o *logs* a la hora de ejecutar la precarga de clases se obtiene un mejor rendimiento en los tiempos de ejecución, que en el mejor de los casos de prueba llegaron a ser superiores al 90%, debido a que no es necesario realizar un análisis de *bytecode* para extraer las clases que se desean precargar.

8. Por medio del módulo de *Buffering* se logró garantizar la integridad de los datos interpretados durante la ejecución de un *bytecode*, mejorando así la confiabilidad de la plataforma SUMA.

Capítulo VIII. Recomendaciones

A continuación se presentan una serie de recomendaciones las cuales podrían ser de utilidad para desarrollos futuros:

- Integrar las diferentes versiones e implementaciones de la plataforma SUMA antes de dar inicio a futuros desarrollos sobre esta plataforma, de tal manera de mantener una consistencia de versiones.

- Realizar un estudio detallado de todas las fortalezas y debilidades que presenta actualmente la plataforma SUMA, que sirva de base para la realización de nuevos proyectos que permitan aumentar el rendimiento de esta plataforma y así brindar un mejor servicio.

- Implementar una nueva versión del *buffer* de lectura y escritura de la plataforma SUMA; pudiendo tomar como referencias algunas de las propuestas realizadas en la fase de construcción del capitulo V de este Trabajo Especial de Grado; a manera de lograr un mejor desempeño del mismo.

Bibliografía

Libros

[1] Hennessy, Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann. May 2002.

[2] P. Kroll, P.Kruchten, *The Rational Unified Process Made Easy. A Practitioner's Guide to the RUP*. Addison Wesly.April 2003.

[3] F. Berman, A. Hey, G. Fox .*Grid Computing Making the Global Infrastucture a Reality*. Willey Series. 2003.

[4] David West, *Planning a Project with the Rational Unified Process*. Rational. 2002.

[5] B. Cohoon, K. Mckinley. *Tolerating Latency by Prefetching Java Objects. Workshop on Hardware Support for Objects and Microarchitectures for Java,* Autin. October, 1999.

[6] Gary. Pollice. *Ussing the Rational Unified Process for Small Projetcs: Expanding Upon Extreme Programing*. Rational. 2001.

[7] Jain Raj. *Art of Computer Systems Performance Analysis Techniques for Experimental Design Measurements Simulation and Modeling*. Willey Computer. January 1991.

[8] Andrew Appel, Jens Palsberg. *Moderm Compiler Implementation in Java*, Second Eition. Cambridge University Press. 2002.

Papers

[9] C. Krintz, B. Calder, H. Lee, B. Zorn. Overlapping execution with transfer using non-strict execution for mobile programs. In *Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[10] C. Krintz, B Calder, U. Hölzle. Reducing transfer delay using java class file splitting and prefetching. In *proceedings of 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming System, Languages and Applications (OOP-SLA)*, November 1999.

[11] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. IEE Communications Magazine, 14(2), February 1997.

Referencias Electrónicas

[12] Universidad Politécnica de Madrid. Tutorial de *Grid*.
http://www.catedravodafone.etsit.upm.es/formacion/seminarios/grid.html

[13] Programming Languages for the Java Virtual Machine.
http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html

[14] Java Class File Format.
http://jakarta.apache.org/bcel/manual.html

[15] Api de Java versión 1.4.2
http://java.sun.com/j2se/1.4.2/docs/api/

[16] Herramienta *Espresso* para el Análisis de un *Bytecode* de Java
http://types.bu.edu/Espresso/HTML/espresso.Espresso.html#_top_.

[17] Página Personal de la tutora Yudith Cardinale.
http://www.ldc.usb.ve/~yudith/investigacion/

[18] Página Principal de la Plataforma SUMA.
http://suma.ldc.usb.ve/disgeneral/corba.html

[19] The Java Class File Format.
http://www.murrayc.com/learning/java/java_classfileformat.shtml

[20] The Class File Format.
http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html

[21] Conceptos Básicos de un Diagrama de Secuencias.
http://web.cs.ualberta.ca/~pfiguero/soo/uml/secuencia01.html

[22] Java Grande Forum Benchmark Suite - Version 2.0
http://www.javagrande.org

Glosario de Términos

- ***Aplicación Sintética:*** programa utilizado para realizar pruebas de funcionalidad a una aplicación.
- ***Benchmark:*** Aplicación para medir el rendimiento de un sistema.
- ***Bytecode*:** Es una clase de código intermedio, el cual es mas abstracto que el código de maquina y es producido por compiladores e interpretado por compiladores.
- ***Buffering*:** Técnica utilizada para reducir los tiempos de lectura de datos, ya que mantiene los datos en un buffer y no en el disco.
- ***Caching:*** Tecnología basada en subsistemas de memoria, con la finalidad de acelerar el computador, pero a un costo bajo.
- ***.Class*:** *Bytecode* de java generado por el JVM.
- ***Callbacks:*** Es el proceso de comunicación entre un Agente de Ejecución y un Cliente, en donde el primero solicita alguna información.
- ***Execute:*** Servicio de de la plataforma SUMA para ejecutar una aplicación o programa de un cliente, especificando la clase principal.
- **Globus Toolkit:** Software de código abierto usado para construir Grids
- ***IDL:*** Es un lenguaje para la definición de interfaces, el cual es independiente del lenguaje de programación.
- ***JDK:*** Es un producto de *SUN Microsystems* apuntado al desarrollo de aplicaciones en lenguaje Java.
- ***JVM:*** Máquina Virtual de Java, la cual es una herramienta que permite ejecutar e interpretar un *bytecode*. En cada plataforma existe un JVM.

- ***mpiJava:*** Librería o conjunto de clases Java que permiten ejecutar aplicaciones en paralelo.
- ***Prefetching:*** Técnica que se basa en la transferencia de archivos por adelantado. Es empleada para reducir el retardo en la ejecución de programas de manera remota.
- ***Submit:*** Servicio de de la plataforma SUMA para ejecutar una aplicación o programa de un cliente especificando la clase principal, sin necesidad que el cliente este conectado durante la ejecución.

Apéndice A. Planes de Trabajo

Plan de Trabajo Módulo de Precarga	
Inicio	**Iteración I:** - Levantar información (entrevistas, búsqueda de material, requerimientos). - Crear plan de trabajo. - Realizar documento visión. - Crear diagrama de casos de uso. **Iteración II:** - Desarrollar casos de uso de mayor importancia - Realizar análisis de riesgo. - Desarrollar nuevos casos de uso.
Elaboración	**Iteración III:** - Desarrollar nuevos casos de uso - Diseñar diagramas de clase. - Diseñar diagramas de secuencia. **Iteración IV:** - Desarrollar casos de uso restantes - Crear algunos prototipos.
Construcción	**Iteración V:** - Creación de componentes del cliente interno **Iteración VI:** - Creación de componentes del agente de ejecución.
Transición	**Iteración VII:** - Traslado de la versión desarrollada al Ambiente de Producción. - Realizar pruebas.

Tabla A.1. Plan de Trabajo Módulo Precarga
Fuente: Elaboración propia.

Plan de Trabajo Módulo *Buffering*	
Inicio	**Iteración I:** - Levantar información (entrevistas, búsqueda de material, requerimientos). - Crear plan de trabajo. - Realizar documento visión. - Realizar estudio de técnicas de *buffering*
Elaboración	**Iteración II:** - Desarrollar caso de uso restante. - Diseñar diagramas de clase. - Diseñar diagramas de secuencia. **Iteración III:** - Desarrollo de prototipos.
Construcción	**Iteración IV:** - Creación de componentes del módulo de *buffering* **Iteración IV:** - Creación de propuesta sobre *buffering*.
Transición	**Iteración V:** - Traslado de la versión desarrollada al Ambiente de Producción. - Realizar pruebas.

Tabla A.2. Plan de Trabajo Módulo de *Buffering*
Fuente: Elaboración propia.

Apéndice B. Diagrama de Clases de SUMA

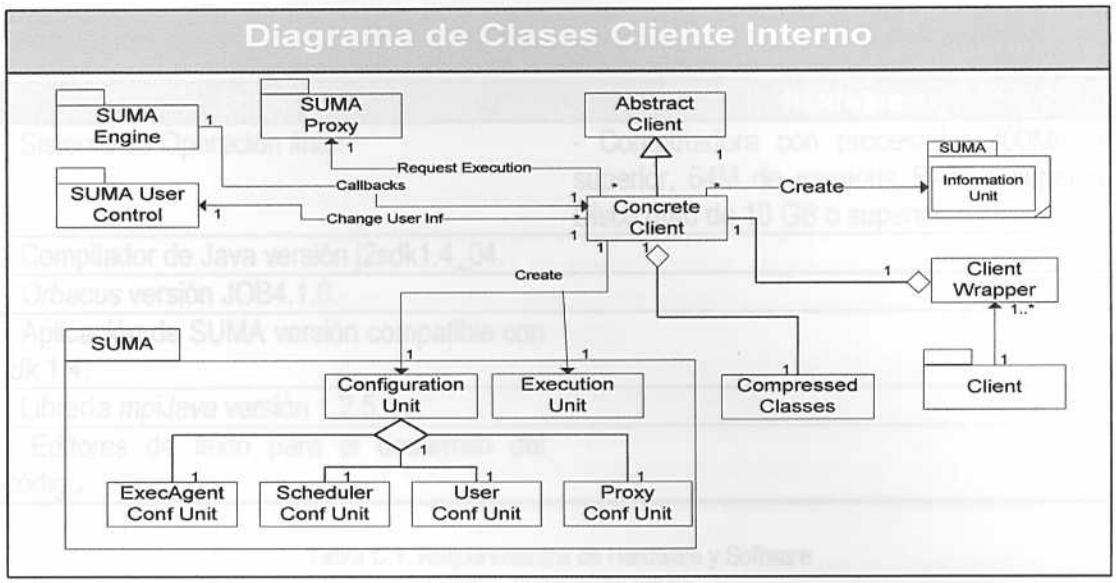

Figura B.1. Diagrama de Clases del Cliente Interno de SUMA
Fuente: Elaboración propia.

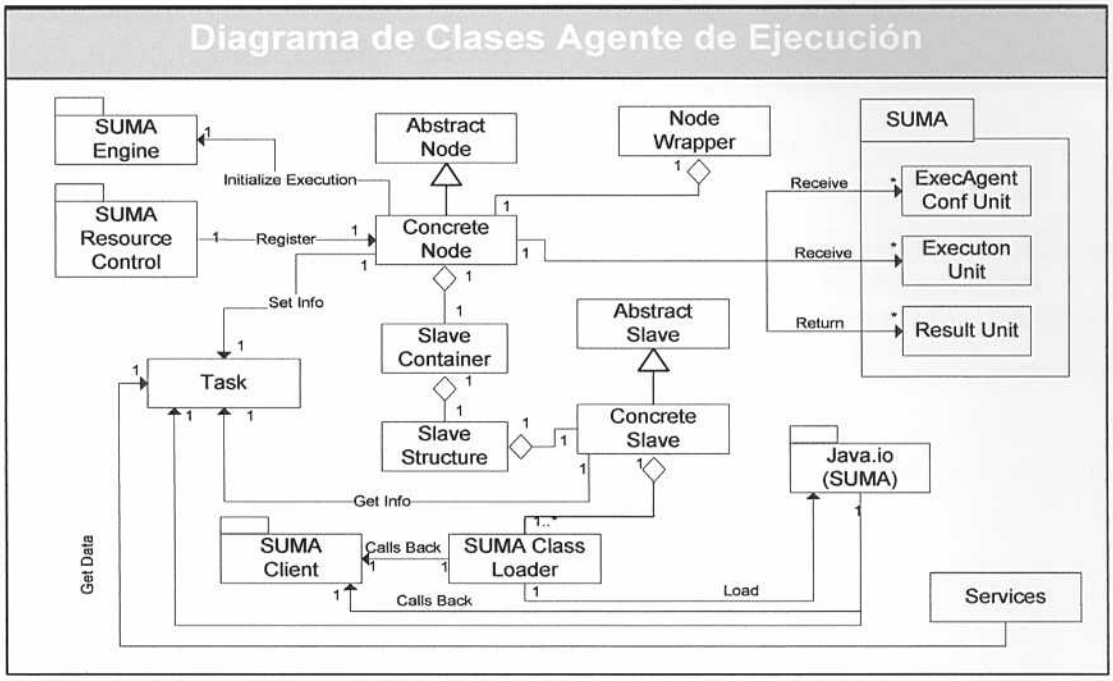

Figura B.1. Diagrama de Clases del Agente de Ejecución de SUMA
Fuente: Elaboración propia.

Apéndice C. Requerimientos de Hardware y Software

Software	Hardware
- Sistema de Operación *linux*	- Computadora con procesador 300Mhz o superior, 64M de memoria RAM o superior, Disco duro de 10 GB o superior.
- Compilador de Java versión j2sdk1.4_04.	
- *Orbacus* versión JOB4.1.0.	
- Aplicación de SUMA versión compatible con *jdk* 1.4.	
- Librería *mpiJava* versión 1.2.5.	
- Editores de texto para el desarrollo del código.	

Tabla C.1. Requerimientos de Hardware y Software
Fuente: Elaboración propia

Apéndice D. Documentos de Visión

Precarga De Clases: Visión	
Problemas	SUMA es una plataforma distribuida, creada con la finalidad de aprovechar al máximo los recursos no utilizados de varios computadores, colocándolos a la disposición de todas aquellas personas que deseen ejecutar un *bytecode* de Java sin importar en donde se encuentre. Sin embargo por el simple hecho de ejecutar código de manera remota, se presentan algunas situaciones: - Cada vez que el proceso de ejecución requiere una clase procedente del cliente, el agente de ejecución debe detener el proceso y transferir la clase requerida para poder continuar con la ejecución, ocasionando retrasos en todo el proceso. - Si un cliente desea ejecutar la misma aplicación en varias oportunidades, el proceso se realiza de la misma manera.
Posibles Soluciones	Para solventar los problemas planteados se podría realizar lo siguiente: - Crear un modelo de precarga de clases, el cual permita reducir el número de peticiones por parte del agente de ejecución hacia el cliente en el momento de ejecutar el *bytecode*. - Mantener un registro de todas aquellas clases involucradas en el proceso de precarga, el cual contenga información sobre tiempos de transferencia e invocación de clases los cuales puedan ser usados en futuras corridas parar minimizar tiempos de ejecución.
Componentes Involucrados	Los componentes en donde se realizarán modificaciones e inclusión de nuevos módulos son: - Cliente interno. - Agente de ejecución.
Casos de Uso	- Analizar *bytecode* de la clase principal en busca de las clases invocadas. - Carga de *logs* de precarga. - Generar grafo de bloques básicos de control. - Ejecución del método de precarga. - Generación de *logs* de precarga.
Criterios de Éxito	- La implementación de la precarga de clases reduce los tiempos de ejecución de las aplicaciones o programas de los clientes externos. - Se generan registros de luego de cada corrida los cuales pueden ser empleados en un futuro para minimizar tiempos de ejecución.

Tabla D.1. Documento de Visión Módulo de Precarga
Fuente: Elaboración propia

Buffering de Datos: Visión	
Problemas	La plataforma SUMA presenta un manejo del *buffering* de datos sencillo pero eficaz, el cual le permite reducir significativamente el número de *callbacks* de lectura al cliente. Entre los aspectos en los que se podrían desarrollar algunas mejoras se destacan: - A medida que el agente de ejecución va solicitando la información al cliente, la misma va siendo almacenada en el buffer para que luego pueda ser leída por otros procesos. El problema puede presentarse si estos procesos modifican el archivo de origen que esta siendo leído por el agente de ejecución, ya que se podría estar leyendo información desactualizada o errónea. - El tamaño de buffer puede ser parametrizable por el cliente, sin embargo el tamaño predeterminado es de 1024B. Por cada lectura que se realiza se pide un bloque completo de información, pero si la información que se debe leer es mucho menor al bloque solicitado se puede caer en una mala administración de los recursos.
Posibles Soluciones	Entre las posibles soluciones para mejorar la eficiencia del *buffering* de datos se destacan: - Implantar nuevos procesos que verifiquen constantemente el estado del archivo que esta siendo leído por el agente de ejecución. Si el archivo ha sido modificado y el proceso de lectura no ha finalizado, se interrumpiría el proceso de lectura por medio de una excepción. - Implementar alguna técnica de bloqueo de registros, la cual permita restringir el acceso a un registro específico cuando el mismo esta siendo actualizado. - Diseñar una nueva implementación del *buffering* de lectura de datos la cual permitiría solicitar bloques de memoria de un tamaño variable, de acuerdo a las solicitudes realizadas al cliente. De esta manera se daría un mejor aprovechamiento de los recursos del sistema.
Componentes Involucrados	Los componentes en donde se realizarán modificaciones e inclusión de nuevos módulos son: - Cliente interno. Paquete *ConcreteClient*. - Agente de ejecución. Paquete *sumaio*.
Casos de Uso	- Comprobar estado de archivo. - Realizar mediciones de referencia.
Criterios de Éxito	- La lectura de datos por parte del agente de ejecución se realiza de una manera eficiente y sin riesgos de trabajar con información errónea. - Los recursos del sistema son aprovechados de una manera más eficiente, lo que aumentaría el rendimiento.

Tabla D.2 Documento de Visión Módulo de *Buffering*
Fuente: Elaboración propia

Apéndice E. Casos de Uso del Módulo de Precarga

Descripción de Actores	
Cliente Externo	Es módulo que desea utilizar el sistema para obtener un beneficio, solicitando al mismo la ejecución de programas de manera remota por medio de dos métodos: *submit* o *execute*.
Cliente Interno	Es el encargado de establecer y mantener la comunicación entre el cliente externo y el sistema (En el caso del método *submit* la comunicación será con el cliente virtual, mientras que con el método *execute* será directamente con el agente de ejecución). El cliente interno es el que atiende las llamadas desde el agente de ejecución (*callbacks*) para cargar las clases y archivos de datos dinámicamente.
Agente de Ejecución	Es el encargado de procesar las solicitudes de los clientes externos. El mismo recibe una orden de un representante para ejecutar una aplicación, por lo que comienza a cargar clases y archivos dinámicamente

Tabla E.1. Descripción de Actores
Fuente: Elaboración Propia

Nombre	*Carga de logs de precarga*	
Disparador	El cliente interno desea ejecutar una aplicación o programa.	
Actor primario	Cliente interno.	
Precondición	El cliente externo solicita ejecutar una aplicación con la opción "*-pref* <dirección del *log*>".	
Condición final de éxito	Los *logs* de precarga han sido cargados exitosamente.	
Condición final de fallo	Los *logs* de precarga no pudieron ser cargados.	
Escenario	Actor	Sistema
	1. - Solicita cargar los *logs* de precarga de la clase principal que se desea ejecutar.	2. - Busca el archivo en la dirección dada por el cliente externo.
		3. - Lee el archivo de *logs* y extrae la información almacenada.
		4. - Copia la información del archivo en una estructura y la almacena en forma de grafo de bloques básicos.
		5. - Retorna el grafo de bloques básicos.
Extensiones	2.a - Si no existe un archivo de *logs*: 1.a.1 -Finaliza el caso e uso.	
	3.a - Si no se puede leer el archivo o si posee información errónea: 3.a.1 - Finaliza el caso de uso.	

Tabla E.2. Caso de Uso Carga de *Logs* de Precarga
Fuente: Elaboración Propia

Nombre	*Analizar bytecode de la clase principal en busca de las clases invocadas*	
Disparador	El cliente interno solicita analizar la clase o *bytecode* en busca de las clases invocadas.	
Actor primario	Cliente interno.	
Precondición	El cliente externo debe haber invocado el método *submit* o *execute sin la opción "-pref <dirección del log>"*.	
Condición final de éxito	Se obtienen las los nombres de las clases que son invocadas en l clase principal dada.	
Condición final de fallo	No se pudo obtener los nombres de las clases invocadas.	
Escenario	Actor	Sistema
	1. - Invoca el proceso que genera el analizar *bytecode*.	2. - Analiza el *bytecode* de la clase principal en busca de todas aquellas clases que son invocadas.
		3. - Extrae la información contenida dentro del repositorio del *bytecode*.
		4. - Almacena las clases invocadas en el *bytecode* que no pertenecen a los paquetes de java ni a los de SUMA en un repositorio.
Extensiones		

Tabla E.3. Caso de Uso Analizar *Bytecode* de la Clase Principal en Busca de las Clases Invocadas
Fuente: Elaboración Propia

Nombre	*Generar grafo de bloques básicos de control*	
Disparador	El cliente externo extrae la información de las clases contenidas dentro del *bytecode*.	
Actor primario	Cliente interno.	
Precondición	El cliente interno debe haber analizado y extraído la información del *bytecode*.	
Condición final de éxito	El grafo de bloques básicos es generado.	
Condición final de fallo	No se pudo generar el grafo de bloques básicos.	
Escenario	Actor	Sistema
	1. - Invoca el proceso que genera el grafo de bloques básicos, transfiriéndole la clase principal que se desea procesar.	2. - Por cada una de las clases encontradas durante el análisis se crean valores iniciales de tiempos de transferencia e invocación y se almacenan en una estructura.
		3. - Construye el grafo de forma recursiva basándose en la estructura obtenida
		4. - Retorna el grafo de estructuras obtenido.
Extensiones		

Tabla E.4. Caso de Uso Generar Grafo de Bloques Básicos de Control
Fuente: Elaboración Propia

Nombre	*Generación de logs de precarga*	
Disparador	El agente de ejecución solicita estimar los tiempos de transferencia de cada clase.	
Actor primario	Agente de ejecución.	
Precondición	La carga y ejecución de clases debe haber iniciado y no debe existir ningún archivo de *logs*.	
Condición final de éxito	Los *logs* de precarga han sido generados exitosamente.	
Condición final de fallo	Los *logs* de precarga no pudieron ser generados.	
Escenario	Actor	Sistema
	1. - Solicita estimar el tiempo de carga de cada clase.	2. - Toma los tiempos antes y después de cargar una clase.
		3. - Calcula el tiempo de transferencia y de invocación de cada clase.
		4. - Almacena en la estructura de cada clase los tiempos calculados.
		5. - Registra en un archivo las estructuras formadas.
Extensiones		

Tabla E.5. Caso de Uso Generación de *Logs* de Precarga
Fuente: Elaboración Propia

Apéndice E. Casos de Uso del Módulo de Precarga

Nombre	*Ejecutar método de precarga*	
Disparador	El agente de ejecución invoca método que transfiere por adelantado los archivos y métodos.	
Actor primario	Agente de ejecución.	
Precondición	Se debe haber invocado el proceso de ejecución del *bytecode*.	
Condición final de éxito	Los archivos fueron transferidos exitosamente.	
Condición final de fallo	Los archivos no pudieron transferirse.	
Escenario	Actor	Sistema
	1. - Invoca método de precarga de clases, dando como parámetro el grafo de estructura de clases.	2. - Recorre el grafo en busca de los bloques básicos que va a transferir.
		3. - Transfiere cada clase del grafo tomando en cuenta los tiempos de transferencia e invocación de las mismas.
Extensiones	3.a - Si el tiempo de carga de una clase es menor al tiempo de ejecución: 3.a.1 - No se realiza la precarga de dicha clase.	

Tabla E.6. Casos de Uso Ejecutar Método de Precarga
Fuente: Elaboración Propia

Apéndice F. Riesgos y Mitigaciones

Riesgos	Posibles Soluciones
- La implantación de la precarga de clases del lado del Agente de Ejecución no reduzca los tiempos de ejecución, y por el contrario los aumente.	- Modificar el orden de invocación de las clases. - Tratar de reducir el número de instrucciones que se ejecutan normalmente de tal manera de reducir los tiempos de ejecución.
- La ejecución de la precarga de clases interrumpa el proceso de ejecución.	- Que el módulo de precarga trabaje de una manera independiente al proceso de ejecución, de tal manera de no interrumpirlo si se presenta algún problema.
- Los registros empleados para cargar la información contengan información no actual o errónea.	- Que los registros sean renovados cada vez que se ejecute alguna aplicación o programa. - Verificar a través del cliente o por cualquier otro método, si la aplicación o programa que se desea ejecutar fue alterado desde la última corrida.

Tabla F.1. Riesgos y Mitigaciones Módulo Precarga
Fuente: Elaboración Propia

Riesgos	Posibles Soluciones
- Realizar modificaciones basadas en una mala política de *buffering* de datos podría ocasionar retrasos en los tiempos de ejecución de alguna aplicación o programa, disminuyendo el rendimiento de la plataforma SUMA	- Buscar antecedentes de las posibles técnicas de *buffering* a implantar, y comparar los resultados obtenidos a manera de determinar la factibilidad de la implantación de dicha técnica.
- La interrupción innecesaria de una aplicación o programa al determinar que un archivo ha sufrido alguna modificación durante su lectura e interpretación.	- Definir buenos parámetros de comparación que permitan determinar con mayor precisión si un archivo ha sido modificado mientras esta siendo leído por el sistema.

Tabla F.2. Riesgos y Mitigaciones Módulo *Buffering*
Fuente: Elaboración Propia

Apéndice G. Plan de Elaboración de Prototipos

	Pasos para la Elaboración de Prototipos
Paso 1	Recolectar información sobre la estructura de un archivo *.class* de Java, determinar cuáles eran sus componentes, tipos de datos almacenados, y sobre todo cómo poder extraer información de éstos.
Paso 2	Encontrar una manera de extraer la información contenida en un *bytecode* referente a las clases invocadas por el mismo
Paso 3	Crear un programa que analizara un archivo *.class*, extrayendo todas aquellas clases que eran invocadas desde este archivo. Este programa recibía como parámetros de entrada el nombre de la clase principal que se quería analizar y retornaba un vector con todas las clases encontradas en el mismo orden en que serían ejecutadas.

Tabla G.1. Plan Elaboración de Prototipos Módulo Precarga
Fuente: Elaboración propia

	Pasos para la Elaboración de Prototipos
Paso 1	Realizar un estudio referente a los criterios que podrían definir si un archivo ha sido modificado.
Paso 2	Realizar pruebas en donde se trabajase con diferentes hilos de ejecución los cuales deberán ejecutar lecturas y escrituras sobre un mismo archivo, de tal manera poder determinar si el mismo fue modificado
Paso 3	Crear un programa que analizara un archivo *.class*, extrayendo todas aquellas clases que eran invocadas desde este archivo. Este programa recibía como parámetros de entrada el nombre de la clase principal que se quería analizar y retornaba un vector con todas las clases encontradas en el mismo orden en que serían ejecutadas.

Tabla G.2. Plan Elaboración de Prototipos Módulo *Buffering*
Fuente: Elaboración propia

Apéndice H. Casos de Uso del Módulo de *Buffering*

Descripción de Actores	
Agente de Ejecución	*El agente de ejecución es el encargado de procesar las solicitudes* de los clientes externos. Al momento de realizar la ejecución de la aplicación o programa, el agente de ejecución solicita cargar las clases y archivos necesarios para su procesamiento.

Tabla H.1. Descripción de Actores
Fuente: Elaboración Propia

Nombre	*Realizar mediciones de referencia.*	
Disparador	El agente de ejecución solicita leer e interpretar algún archivo.	
Actor primario	Agente de ejecución	
Precondición	Debe existir una referencia valida del archivo que se analizará.	
Condición final de éxito	Se logró tomar con éxito la fecha de la última modificación del archivo y su tamaño actual.	
Condición final de fallo	Las medidas de referencia no pudieron ser tomadas.	
Escenario	Actor	Sistema
	1. - Solicita el archivo o clase que desea interpretar.	2. - Busca la referencia del archivo solicitado y lo devuelve al agente de ejecución.
	3. - Solicita ir leyendo el archivo con la referencia obtenida mientras el mismo contenga información almacenada.	4. - Toma la fecha de la ultima modificación realizada sobre el archivo así como su tamaño actual.
		5. - Retorna los valores obtenidos.
Extensiones	2.a. - No existe referencia al archivo solicitado. 2.a.1. - El sistema lanza una excepción y el proceso de ejecución es interrumpido.	
	4.a. - No es posible tomar la fecha de la ultima actualización del archivo. 4.a.1. - Finaliza el proceso. 4.b. - No es posible tomar el tamaño actual del archivo especificado. 4.b.1. - Finaliza el proceso.	

Tabla H.2. Caso de Uso Realizar Mediciones de Referencia
Fuente: Elaboración Propia

Nombre	*Comprobar estado de archivo.*	
Disparador	El agente de ejecución solicita comprobar el estado de un archivo para determinar si sufrió alguna modificación.	
Actor primario	Agente de ejecución.	
Precondición	Se han tomado los valores de referencia del archivo al que se desea comprobar su estado.	
Condición final de éxito	Se ha comprobado el estado actual del archivo con éxito.	
Condición final de fallo	No se comprobó el estado actual del archivo.	
Escenario	Actor	Sistema
	1. - Solicita comprobar el estado actual del archivo.	2. - Recibe los valores de las medidas de referencia del archivo y los almacena.
		3. - Solicita tomar nuevos valores de referencia del archivo.
		4. - Compara los valores de referencia almacenados con los nuevos valores.
Extensiones	4.a. - Los valores de referencia almacenados coinciden con los nuevos valores. 4.a.1. - El archivo no ha sufrido modificaciones, por lo que se retorna el valor "*false*". 4.b. - Los valores de referencia almacenados no coinciden con los nuevos valores. 4.b.1. - El archivo ha sufrido alguna modificación, por lo que se lanza una excepción.	

Tabla H.3. Caso de Uso Comprobar estado de Archivo
Fuente: Elaboración Propia