UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Yafrid-NG: Mejora de un sistema Grid Computacional para el render de escenas 3D

José Ángel Mateos Ramos

Diciembre, 2007

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Informática

Yafrid-NG: Mejora de un sistema Grid Computacional para el render de escenas 3D

Autor: José Ángel Mateos Ramos
Director: Carlos González Morcillo

Diciembre, 2007.

TRIBUNAL:

Presidente:
Vocal:
Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	Fdo.:	Fdo.:

© José Ángel Mateos Ramos. Se permite la copia, distribución y/o modificación de este documento bajo los términos de la licencia de documentación libre GNU, versión 1.1 o cualquier versión posterior publicada por la *Free Software Foundation*, sin secciones invariantes. Puede consultar esta licencia en http://www.gnu.org.

Este documento fue compuesto con LaTeX. Imágenes generadas con OpenOffice.

Resumen

El proceso de render es el último paso en la síntesis de imágenes 3D. En este proceso se genera una imagen bidimensional a partir de la especificación tridimensional de una escena. Durante el renderizado se decide el color de cada pixel que forma parte de la imagen resultado en función de una serie de parámetros como son la colocación de la cámara, las fuentes de luz, la geometría del objeto y otros aspectos relevantes.

La etapa de render requiere una gran capacidad computacional, es por ésto que se considera el cuello de botella en la generación de gráficos por computador.

El presente proyecto pretende reducir los tiempos empleados por el proceso de renderizado haciendo uso de un conjunto de computadores heterogéneos, distribuídos en internet, que ceden parte de sus ciclos de CPU para renderizar escenas tridimensionales.

Abstract

Rendering is the last step of 3D image synthesis process. During this process a 2D image is generated from a 3D specification of a scene. Rendering phase consists in deciding which color will be assigned to each pixel that compose the final image. This value depends on a set of parameters such as the place where camera is, sources of light, geometry of the object and some other important aspects.

Rendering stage is a computer intensive task and this is why it is considered the bottleneck in computer graphics.

This project tries to reduce the time spent by the rendering stage using a set of heterogeneous computers, distributed over the Internet that will supply some of their resources for rendering 3D scenes.

Con todo mi cariño para mis padres
y mi hermana, por permitirme llegar
hasta donde estoy. Os quiero.

Agradecimientos

Ha sido mucha la gente que ha participado en este proyecto. Seguro que alguien se queda en el tintero, espero que sepa perdonarme.

A Carlos Gonzalez Morcillo por su ayuda, su apoyo y su amistad. Es un verdadero lujo trabajar contigo. A la gente del grupo Oreto por cederme un sitio en su laboratorio. Sois unos compañeros perfectos y todavía tendréis que soportarme un poquito más. A Loren por el camino recorrido, todo un placer. Seguimos adelante. A Nacho, Inés, Violeta e Iñaki por compartir su mesa y sus sonrisas conmigo. A Chema por la revisión del documento y su inminente visita.

Muchas Gracias a todos. Os llevo en mi corazón.

Índice general

Índice de figuras

Índice de cuadros

Índice de listados

Índice de algoritmos

Capítulo 1

Introducción

En la actualidad, las grandes productoras cinematográficas hacen uso de los computadores para obtener espectaculares efectos especiales e incluso películas completas de animación. Los gráficos por computador también se utilizan en la industria de los videojuegos. Son muchos los campos en los que los gráficos 3D juegan un papel importante. Por lo tanto conviene estudiar el proceso de producción 3D y analizar las posibles limitaciones que se presentan para poder mejorarlas.

Para el proceso de producción de gráficos y animaciones 3D se pueden adoptar diferentes estrategias, pero siempre hay que tener en cuenta las siguientes etapas básicas[36](ver figura 1.1):

1. **Pre-producción**: Durante esta etapa se lleva a cabo el desarrollo de los guiones, el casting y los *storyboads*. La pre-producción es el inicio de un proyecto. El realizar una pre-producción errónea se puede traducir en un retraso del proyecto, superación de costes y escasez creativa.

2. **Producción**: Durante el proceso de producción, lo primero que se hace es el modelado de los personajes, los objetos y los escenarios que se van a utilizar. Una vez que se ha modelado todo lo necesario el siguiente paso es el de añadir esqueletos y animar los objetos deseados. Finalmente el proceso de render consiste en representar visualmente los

modelos animados con la ayuda de una cámara. Este proceso es especialmente costoso ya que se aplican algoritmos complejos que requieren una gran capacidad de cómputo y consumen una cantidad considerable de recursos.

3. **Post-producción**: Durante esta etapa se aplicarán los retoques finales, como la composición final del proyecto, la modificación de algunos aspectos de la animación... etc.

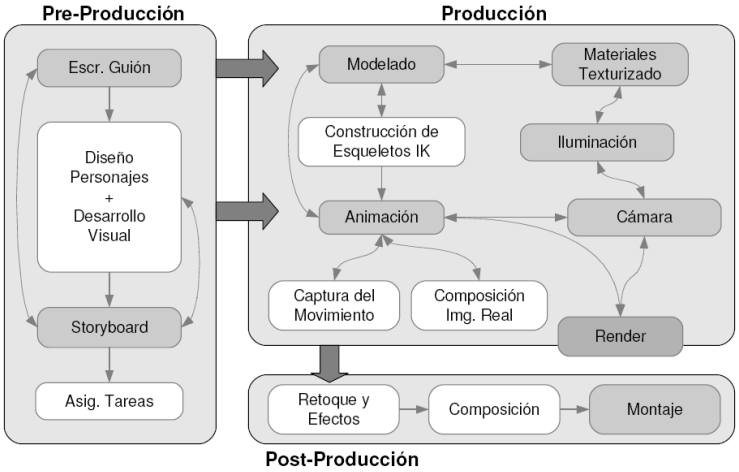

Figura 1.1: Proceso de Síntesis 3D

Dentro de la etapa de producción, el proceso de render requiere grandes cantidades de tiempo y de recursos. Se utilizan algoritmos complejos cuya función es la simulación de la luz física para determinar el color de cada uno de los píxeles que forman parte de la imagen resultado.

La etapa de renderizado supone un cuello de botella en el proceso de producción de gráficos 3D. Debido a esto son muchas las iniciativas que intentan reducir los tiempos de renderizado. Según Chalmers[16] se siguen tres grandes líneas de investigación:

- **Optimizaciones vía hardware**: Algunos desarrolladores utilizan GPUs (Graphics Processing Units) programables como una forma de optimizar los tiempos de render. Éste es el caso de GELATO que es un sistema de renderizado híbrido (basado en hardware y en software) desarrollado por Nvidia. Otra opción es la de utilizar hardware diseñado especialmente para tareas de renderizado. El problema de esta aproximación es que es específica para cada arquitectura hardware.

- **Mejoras en los algoritmos**: Los algoritmos de render han sido refinados continuamente. En la actualidad con este tipo de optimizaciones es dificil obtener una mejora en el tiempo de renderizado.

- **Uso de computación distribuida**: Muchos estudios de cine utilizan las llamadas render farms (granjas de renderizado) para acelerar la producción de los fotogramas, pero ésta es una solución que no está al alcance de todo el mundo, además se trata de computadores homogéneos y suelen pertenecer a una sola organización.

Como se puede observar, son muchas las iniciativas que tratan de reducir los tiempos de renderizado, esto nos indica que la etapa de renderizado es un cuello de botella en el proceso de producción de gráficos 3D. Sería interesante aplicar otros enfoques para reducir los tiempos de renderizado.

1.1. Justificación del trabajo

Con el propósito de reducir los tiempos empleados en la etapa de renderizado, en el presente proyecto se propone una arquitectura grid para realizar tareas de render. Este grid estará compuesto por un conjunto de máquinas heterogéneas conectadas a internet, cada una de las cuales realizará una porción del trabajo. Basándose en el término de *Volunteer Computing*, Yafrid-NG permitirá a una serie de usuarios que cedan sus ciclos de CPU para renderizar escenas enviadas por otros usuarios y al mismo tiempo utilizarán el grid para renderizar sus propias escenas.

1.2. Estructura del documento

El documento actual se estructura en seis capítulos y varios anexos. A continuación se describe el contenido de cada capítulo de una forma detallada.

- **Capítulo 1: Introducción**: Se trata del presente capitulo. En éste se presenta una breve introducción al problema del renderizado, las aproximaciones que se han seguido

para reducir los tiempos de render y la justificación del presente trabajo. También se presentan los objetivos perseguidos y la estructura del documento.

- **Capítulo 2: Objetivos del proyecto**: En este capítulo se explicarán, de una forma más detallada, los objetivos del presente proyecto.
- **Capítulo 3: Antecedentes, estado de la cuestión**: En este capítulo se realizará un estudio de las diferentes áreas que guardan relación con el proyecto así como algunas de las tecnologías existentes para su implementación.
- **Capítulo 4: Metodología de trabajo**: En este capítulo se abordarán los objetivos perseguidos, explicando de forma detallada las hipótesis de trabajo.
- **Capítulo 5: Resultados**: En este capítulo se presentarán los resultados obtenidos con las distintas pruebas realizadas.
- **Capítulo 6: Conclusiones**: En este capítulo se presentarán las conclusiones obtenidas con la realización de este trabajo. Además también se propondrán posibles mejoras que se pueden aplicar a Yafrid-NG.

Capítulo 2

Objetivos del proyecto

El objetivo principal de este proyecto es el de construir un sistema que permita utilizar un **grid** de computadores para el renderizado de escenas en 3D. El grid debe manejarse como si se tratase de un recurso local, abstrayendo al usuario de la gestión y reserva de recursos, la división del trabajo, la localización de los recursos, la composición de la escena y otros aspectos que tienen que ver con el problema del renderizado distribuido (ver figura 2.1).

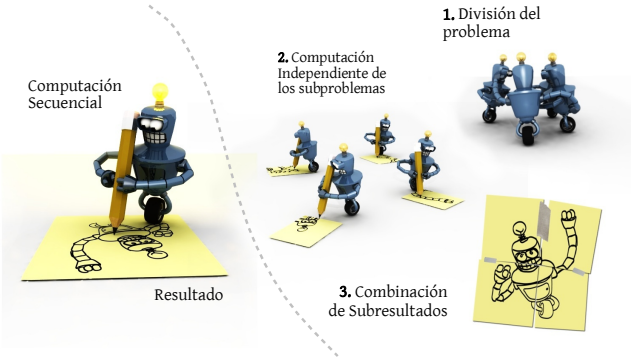

Figura 2.1: Diferencias entre computación secuencial y computación distribuida

Yafrid-NG está basado en una versión previa, llamada Yafrid[17], con el propósito de mejorarlo. El sistema se diseñará desde el principio teniendo en cuenta una serie de aspectos clave como son la descentralización, la seguridad, la implementación de un mecanismo eficiente para el paso de archivos y el uso de algunos de los servicios proporcionados por ICE para la implementación de sistemas distribuídos, *IceGrid* y *Glacier2* (ver sección 3.2.6.3).

Partiendo de este objetivo principal, se definen una serie de requisitos que han de cum-

plirse necesariamente para su ejecución. Estos requisitos son:

- Proporcionar una arquitectura tan descentralizada como sea posible. Todo el proceso de renderizado debe ser dirigido por el cliente que realiza la tarea, eliminando así la mayoría de los servicios de los que dependería el sistema si fuese centralizado. Además, para el paso de archivos se utilizará un sistema tipo peer-to-peer(p2p), haciendo que cada nodo obtenga el archivo utilizado a partir del resto de nodos que van a realizar el trabajo.

- Proporcionar un sistema eficiente para el paso de los archivos necesarios. Para evitar que el cliente envíe los archivos necesarios al conjunto de computadores que van a realizar la tarea se implementará un sistema de paso de archivos. El cliente dividirá el archivo en fragmentos más pequeños que distribuirá a los nodos y serán estos quienes se encargarán de obtener el archivo completo solicitando las piezas que faltan al resto de nodos.

- Proporcionar un entorno seguro de ejecución. Es necesario limitar el uso del grid a los clientes autorizados, para evitar que usuarios malintencionados dañen el correcto funcionamiento del grid. Se aplicarán políticas de seguridad a la hora de reservar los recursos del grid para realizar el renderizado.

- Permitir la gestión dinámica de recursos en el grid. De esta forma el número de recursos de los que dispone el grid puede variar de forma dinámica. Se pueden eliminar y agregar recursos del grid sin demasiada dificultad.

- Proporcionar una adecuada gestión de los recursos del grid. Es importante gestionar los recursos proporcionados, evitando que clientes defectuosos reserven todos los recursos que forman parte del grid, o que los reserven de forma indefinida, impidiendo que otros usuarios hagan uso de éstos.

- Proporcionar una sencilla instalación del software. Es importante desvincular al usuario de las dependencias del sistema. Se creará un instalador que realizará todos los pasos necesarios para que un usuario no tenga que preocuparse de buscar las dependencias

para el funcionamiento de Yafrid-NG. Al menos se realizará un instalador para los principales sistemas operativos del mercado de ordenadores personales.

- Permitir el renderizado tanto de frames estáticos como de animaciones. Esto implica diferentes granularidades en el sistema. Cuando un usuario desea renderizar un frame estático, la escena se dividirá en porciones más pequeñas y cada recurso realizará una porción del trabajo. Si lo que se desea renderizar es una animación la división del trabajo se realizará a nivel de frame, repartiendo los frames de los que consta la animación a los diferentes recursos, cada uno de estos renderizará uno o varios frames, devolviendo los resultados para que el usuario componga la animación.

- Proporcionar un sistema multiplataforma. El sistema debe desarrollarse de tal forma que se permita su ejecución en distintos sistemas operativos y distinto hardware. Esto nos permitirá la puesta en funcionamiento de un grid heterogeneo en el que cualquier máquina con conexión a internet podrá participar en el proceso de renderizado de una escena.

- Proporcionar un sistema basado en estándares abiertos. Así se asegura la portabilidad entre las distintas arquitecturas y sistemas operativos.

- Permitir la independencia del motor de render. En principio el sistema funcionará para los motores de render Blender y Yafray. Pero no debe ser costoso el hecho de adaptarlo a otros motores diferentes.

- Desarrollar el sistema utilizando tecnologías libres. Las partes principales de Yafrid-NG se desarrollará utilizando herramientas de código abierto para garantizar la continuidad del proyecto por la comunidad de usuarios y desarrolladores, el proyecto empleará tecnologías GPL y su distribución se realizará bajo la licencia GNU Public License.

Capítulo 3

Antecedentes, estado de la cuestión

3.1. Computación Grid

3.1.1. Introducción

En los últimos años se ha observado un incremento en la capacidad de los computadores y en el rendimiento de las redes. Sin embargo, todavía existen problemas que no pueden ser tratados con la nueva generación de supercomputadores. Estos problemas requieren de una gran cantidad de recursos, que no pueden ser proporcionados por una simple máquina. Se han realizado múltiples desarrollos que han dado lugar a una tecnología que nos permite utilizar una serie de recursos geográficamente distribuidos como si se tratase de un solo computador, esta tecnología recibe el nombre de **Grid**.

A la hora de realizar procesamiento paralelo existen dos opciones claramente diferenciadas:

- **Cluster de computadores**: Un cluster es un conjunto de computadores típicamente conectados mediante una red de tipo Fast Ethernet. Los recursos del cluster se administran de forma centralizada y todos los computadores pertenecen a la misma organización.
- **Grid de computadores**: Se trata de un conjunto de computadores heterogéneos, tipicamente conectados a través de internet, que colaboran para resolver una tarea de forma paralela. En un Grid los recursos están distribuidos en distintas ubicaciones. A la hora de utilizar un Grid, son varios los aspectos que hay que tener en cuenta debido a su naturaleza intrínseca. Por ejemplo, en un Grid no todos los nodos son confiables, por lo que se tiene que definir algún mecanismo de seguridad para controlar el acceso a los recursos.

El nombre de Grid viene de la similitud de este tipo de sistemas y la red eléctrica. En ambos el usuario desconoce de dónde proviene el servicio, simplemente lo utiliza. La computación Grid es un campo que está actualmente en auge y su principal objetivo es el de resolver un problema de una forma más rápida.

En la sección actual se presentan las características de este tipo de sistemas.

3.1.2. Definición de Grid

Actualmente existen varias definiciones de lo que es un sistema Grid. Una de ellas fue proporcionada por los autores Foster y Kesselman[19].Estos autores definen un Grid como:*"Un sistema que coordina recursos distribuidos usando protocolos e interfaces estandar, abiertos y de propósito general para proporcionar calidades de servicio no triviales."*

Los términos clave de la anterior definición son los siguientes:

- **Coordina recursos distribuídos**: Un Grid integra y coordina recursos y usuarios que se encuentran dentro de diferentes dominios de control. Además se encarga de asuntos como la seguridad, políticas, gestión de miembros...

- **Usando protocolos e interfaces estándar, abiertos y de propósito general**: Un Grid se construye sobre protocolos multipropósito y de interfaces que gestionan temas como la autenticación, la autorización, el descubrimiento de recursos y el acceso a dichos recursos. Es importante que estos protocolos e interfaces sean estándar y abiertos para facilitar la cooperación en un ambiente multiplataforma.

- **Proporcionan calidades de servicio no triviales**: Un Grid permite que sus recursos se usen de una forma coordinada para proporcionar diferentes calidades de servicio, relacionados con el tiempo de respuesta, disponibilidad y seguridad. Además nos permite la combinación de múltiples tipos de recursos para satisfacer demandas complejas por parte del usuario. De esta forma, la utilidad del sistema combinado es significativamente mayor que la suma de sus partes.

Otra definición más moderna de Buyya y Srikumar[13] identifica los sistemas Grid como: *Un tipo de sistema paralelo y distribuido que permite compartir, seleccionar y agregar recursos autónomos geográficamente distribuidos en tiempo de ejecución dependiendo de su disponibilidad, capacidad, rendimiento, coste y los requisitos de calidad de servicio del usuario.*

Según estas definiciones se puede concluir que el termino de computación Grid está asociado con los conceptos de colaboración y de trabajo conjunto.

3.1.3. Clasificación de Sistemas Grid

Son varias las clasificaciones que se pueden hacer de los sistemas Grid según varios aspectos. A continuación se presentan dos de ellas.

Los sistemas Grid pueden estar formados desde un pequeño conjunto de procesadores hasta un red global. *Minoli*[30] hace la siguiente clasificación según el número de computadores y su disposición:

- **Local Grid**: También llamado cluster. Está formado por un conjunto de computadores homogéneos (misma arquitectura, mismo sistema operativo) que pertenecen a la misma organización o departamento. En general, este tipo de Grids no tienen fuertes requisitos en lo referente a la seguridad ya que todos los nodos son confiables.

- **IntraGrid (Enterprise Grids)**: A un nivel mayor de heterogeneidad, este tipo de Grids están formados por procesadores distintos, normalmente localizados en diferentes departamentos de una misma organización. Estos Grids suelen tener una topología jerárquica. La red utilizada en este tipo de Grids suele ser la Intranet de la organización o mediante redes privadas virtuales (VPNs) para conectar nodos remotos de la organización.

- **InterGrid (Grid puro)**: Este tipo de sistemas están constituidos por computadores que no forman parte de la misma organización. Dada su naturaleza completamente heterogénea, se requieren los mayores niveles de seguridad.

Según su funcionalidad, podemos encontrar diferentes clasificaciones. Una de ellas, propuesta por *Jacob*[26], distingue entre las siguientes categorías:

- **Grid Computacional**: Se utiliza para gestionar recursos, especialmente ciclos de CPU y memoria. En muchos de estos grids existen servidores de alto rendimiento dedicados. Los procesadores se suelen denominar nodos, recursos, clientes...

  Por lo general, el modelo computacional de grid utiliza una cola en la que se van colocando los trabajos a ser procesados. De esta forma, el grid está constantemente trabajando y el resultado se puede obtener transcurridas unas horas o unos días después de colocar el trabajo en la cola.

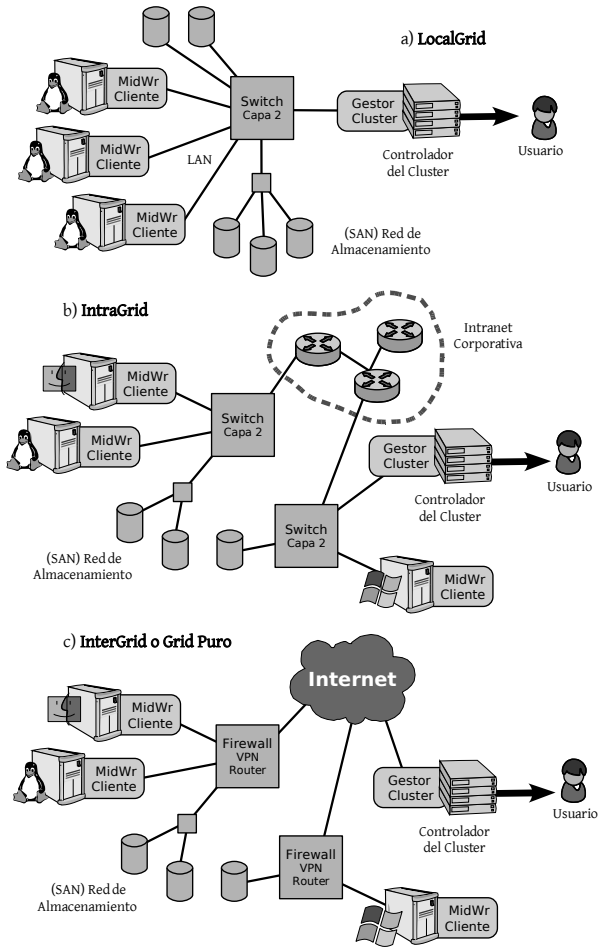

Figura 3.1: Clasificación de Grid según su complejidad

- **Grid de Recolección Computacional**: Este grid se emplea para utilizar y localizar ciclos de CPU. Los nodos de este grid suelen ser PCs de escritorio cuyos usuarios ceden voluntariamente para participar en el grid. Suelen estar formados por un número muy alto de procesadores.

- **Grid de Datos**: Empleado para almacenar datos y proporcionar acceso a múltiples organizaciones. Este tipo de grid no está dedicado a hacer ningún tipo de cálculo sino a almacenar información en varios equipos. Con esta aproximación se busca, o bien mejorar la eficiencia de la recuperación de la información o bien incrementar el espacio de almacenamiento potencial.

Desde el punto de vista del usuario, este tipo de grid se utiliza como si de un solo recurso se tratase. El usuario envía al grid un documento para ser almacenado y el grid escoge una máquina determinada o un grupo de ellas para almacenar esta información.

3.1.4. Seguridad en el Grid

Uno de los aspectos clave a la hora de utilizar un grid de computadores es el de la seguridad. Debido a que no todos los nodos que forman parte del grid son confiables, hay que establecer algún tipo de mecanismo que nos permita decidir si un usuario está o no habilitado para utilizar los recursos que el Grid proporciona. También hay que evitar que los datos que se intercambian entre los nodos sean interceptados por terceros malintencionados.

Así, se pueden definir unos aspectos clave de la seguridad en el grid[15]. Estos aspectos son:

- **Privacidad**: Hace referencia a que los datos intercambiados no puede ser observados por terceros. Sólo el emisor y el receptor tienen que ser capaces de entender los mensajes que están intercambiando. Si alguien más intercepta la comunicación debe ser incapaz de obtener ninguna información util. Esto se consigue, generalmente, con algoritmos de encriptación/desencriptación.

- **Integridad**: La integridad nos asegura que los datos no han sido manipulados de manera fraudulenta durante la comunicación. Aunque una tercera parte no sea capaz de obtener información de las comunicaciones, debido a la confidencialidad, si que podría modificar los mensajes intercambiados. Esto confundiría a la parte receptora, que podría pensar que ha habido un error en la comunicación. Los algoritmos de encriptación de clave pública protegen contra este tipo de ataques, ya que el receptor tiene forma de saber si el mensaje recibido es el que el emisor envió.

- **No-repudiación**: El emisor no puede decir que no envió los datos.

- **Autenticación**: Se trata de la verificación de la identidad de un usuario, recurso, servicio... Normalmente la autenticación es bidireccional, es decir, ambas partes saben con certeza que se están comunicando con quien deberían hacerlo.

- **Autorización**: Cada uno de los recursos o usuarios solo deben usar los servicios para los que ha sido autorizado. La autorización está relacionada con la autenticación, ya que hay que asegurarse de que un usuario es quien dice ser, para poder decidir si está capacitado a realizar una acción determinada.

En esta sección se realizará un estudio acerca de como se consiguen los objetivos anteriormente propuestos.

3.1.4.1. Esquemas de Encriptación

Para conseguir la privacidad en las comunicaciones se hace uso de un algoritmo de encriptación.

Para transmitir un mensaje M entre dos nodos, se aplica un algoritmo de encriptación sobre dicho mensaje obteniendo un mensaje C que es el que se va a transmitir. Cuando el receptor obtiene C aplica un algoritmo de desencriptación sobre C obteniendo nuevamente M (ver figura 3.2).

Para encriptar un mensaje es necesario hacer uso de una clave. Llegados a este punto podemos distinguir dos aproximaciones para la encriptación del mensaje:

- **Encriptación simétrica**: Se utiliza la misma clave para la encriptación y desencriptación del mensaje. El inconveniente de este tipo de encriptación es que es necesario el intercambio de la clave utilizada. Dicha clave puede ser interceptada permitiendo así la desencriptación de los mensajes por parte de un tercero.

  Algunos ejemplos de algoritmos de encriptación simétrica son: ROT13, DES, CAST...

- **Encriptación asimétrica**: Existen dos claves diferentes, una pública y una privada, que son complementarias. Una de estas claves se utiliza para encriptar el mensaje y la otra para desencriptarlo. Es un método más seguro para la encriptación de los datos. Los inconvenientes que presenta este tipo de encriptación es que el cifrado es más lento que con los algoritmos de clave simétrica y que pueden aparecer patrones que faciliten su criptoanálisis.

La clave pública debe ser conocida por todos los nodos que forman parte del sistema. La clave privada solamente la conoce su propietario. Existe una relación entre ambas claves, lo que se encripta con la clave pública se desencripta con la clave privada y viceversa.

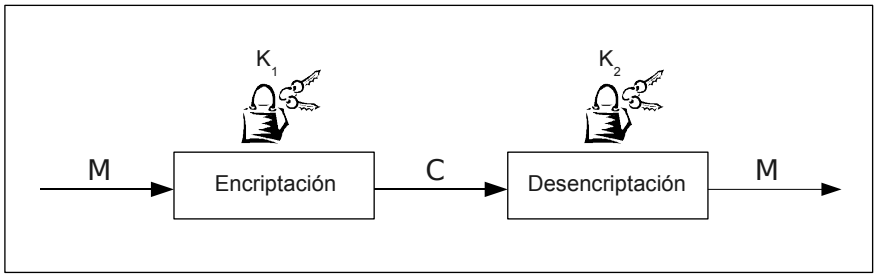

Figura 3.2: Cifrado de un mensaje. M es el mensaje original y C es el mensaje encriptado. Para encriptación simétrica: K_1 = K_2. Para encriptación asimétrica: K_1 != K_2.

Existen una serie de beneficios si se escoge la encriptación asimétrica para proporcionar privacidad. Mientras que utilizando la encriptación simétrica sólo se proporciona privacidad, usando encriptación asimétrica se consigue, además de privacidad, integridad y autenticación. Otra de las ventajas es que el conocimiento de la clave pública no compromete de ninguna forma la clave privada.

3.1.4.2. Public Key Infraestructure (PKI)

Una infraestructura que proporciona autenticación y cifrado basado en clave pública recibe el nombre de Public Key Infraestructure. Cada uno de los usuarios que forme parte de este tipo de infraestructura debe contener una clave pública y otra clave privada. En la base de este tipo de infraestructuras tenemos los **certificados**, que se utilizan para validar al usuario y la clave pública asociada con el usuario, y la **Autoridad de Certificados (CA)** que se encarga de tramitar los certificados.

Los **certificados** son credenciales para un determinado usuario, contienen los datos asociados al usuario y son firmados por un CA. Existen varios tipos de certificados. El formato más popular es el X.509. Un certificado de este tipo contiene la siguiente información:

- La versión de X.509 que se está utilizando.
- La información sobre el usuario y el CA correspondiente.
- El algoritmo utilizado para cifrar la firma del certificado.
- El sujeto cuya clave pública ha sido cifrada.
- La validez del certificado, indicando la fecha hasta la que éste es válido.
- La información acerca de la clave pública.
- El campo de la firma digital. Este campo se crea aplicando una función hash a toda la información anterior firmada con la clave privada del CA.
- Además existen algunos campos que permiten la adaptación de los certificados X.509.

Las **autoridades de certificado (CA)** son entidades en las que confían diferentes sitios. Los CAs son responsables de firmar los certificados de los usuarios que se suscriben al CA. Existen varios modelos de confianza en los sistemas de PKI, estos modelos son:

- **Modelo Monopoly**: En este modelo de confianza sólo existe un CA en el cual confían todas las entidades que obtienen los certificados de este CA. Es un modelo muy sencillo pero tiene problemas de escalabilidad, especialmente para sistemas grandes.
- **Modelo Monopoly más RA**: Similar al anterior, solo que ahora existen RAs (Registry Authorities) que se encargan de la verificación de las claves públicas. Las RAs comunican esta información a las CAs.
- **CA Delegado**: En este modelo, el CA confiable puede emitir certificados para otros CAs que automáticamente pasan a ser confiables también. Éstos reciben el nombre de CAs delegados. El usuario puede obtener certificados de un CA delegado.
- **Oligarquía**: Este sistema se suele utilizar en navegadores. En este sistema los productos tienen un sola clave configurada con muchos CAs. Cualquier certificado tramitado por alguna de estas CAs es aceptado.

- **Otros**: Además de los modelos expuestos anteriormente existen algunos más como son la *Anarquía*, *Restricción de nombres*...

Uno de los problemas de utilizar certificados es que alguno de ellos haya sido robado. Aunque tienen un periodo de validez si algún usuario malintencionado se hace con uno de estos certificados puede ser peligroso. Para evitar esta situación los CAs emiten, periodicamente, una lista de certificados revocados que es una lista de certificados que aún no han caducado, pero que ya no son válidos. El sistema de autenticación debe consultar esta lista antes de aceptar la identidad de un usuario como válida.

3.1.4.3. Secure Socket Layer (SSL)

Secure Socket Layer es uno de los protocolos más utilizados para asegurar la capa de transporte. La versión más reciente se llama Transport Layer Security (TLS). La version 2 de SSL fue desarrollada por Netscape en el año 1995. El protocolo funciona de la siguiente forma:

- El cliente contacta con el servidor para iniciar una sesión SSL/TLS. En este paso el cliente no se autentifica, pero si indica una serie de algoritmos de encriptación que soporta. Además el cliente envía un número aleatorio, R_c, que se utilizará para crear la clave de la sesión.

- El servidor responde enviando su certificado al cliente. También envía otro número aleatorio, R_s que también formará parte de la clave de la sesión.

- Después el cliente verifica el certificado y extrae la clave pública del servidor y elige un número aleatorio S. Además el cliente calcula K que se obtiene a partir de una función de R_c, R_s y S.

- El cliente envía S y la clave K encriptada mediante una función hash y la clave pública del servidor.

- A partir de este punto, toda la información enviada a través del canal SSL/TLS se encripta con la clave de sesión K.

Hay que destacar que con los pasos anteriormente descritos, el cliente es capaz de autenticar al servidor pero el servidor no puede autenticar al cliente. SSL/TLS permite la autenticación mutua, en la que el servidor puede autenticar al cliente si éste contiene el certificado requerido.

3.1.5. Redes P2P

Según Schoder y Fischback[33] el término **Peer-to-Peer(P2P)** se refiere a una arquitectura de red donde todos los nodos son iguales(peers), de tal forma que dos o más nodos son capaces de colaborar espontáneamente sin necesidad de una coordinación central.

Schollmeier[34] enumera las características que definen las redes Peer-to-peer:

1. **Permiten compartir recursos y servicios distribuidos.** En una red P2P cada nodo puede proporcionar funcionalidad de cliente y de servidor; es decir, pueden proporcionar o utilizar recursos como capacidad de almacenamiento, ancho de banda o ciclos de CPU.

2. **Son descentralizados.** No existe un elemento central de organización de la red o en el uso de recursos y comunicación entre nodos. Además ningún nodo puede tener control sobre otro nodo, realizándose la comunicación entre nodos directamente. Existe una distinción propuesta por *Minar*[29] que distingue entre **redes P2P puras** y **redes P2P híbridas**. Las redes de este segundo tipo suelen dejar el indexado de información y la autentificación en manos de una entidad coordinadora, combinando así los principios de las redes P2P y el modelo cliente/servidor.

3. **Autónomos.** Cada nodo de la red P2P puede decidir individualmente qué recursos quiere compartir con otros nodos y cuándo quiere que estén disponibles.

Según Subramanian y Goodman[35], los sistemas de tipo P2P se han clasificado en tres categorías; mensajería instantánea, compartición de ficheros y grid computing. En su trabajo, proponen las categorías de:

- **Información y Gestión Documental**: En este grupo estarían los sistemas que proporcionan información sobre qué nodos y qué recursos están disponibles en la red P2P,

la gestión, almacenamiento y uso de documentos y edición colaborativa síncrona y asíncrona de documentos.

- **Compartición de ficheros**: Probablemente es el tipo de aplicación P2P con más uso en la actualidad. Se estima que más del 70 % del tráfico de Internet se debe a este tipo de aplicaciones. Algunos protocolos y aplicaciones famosas en esta categoría son Napster (2000) que utilizaba un modelo de directorio centralizado para realizar las búsquedas sobre los contenidos de la red (red P2P híbrida según *Minar*[29]), Gnutella (2001) que no utiliza ningún elemento central de coordinación y todos los nodos son iguales. Un modelo novedoso de aplicación P2P es Freenet(2003) donde los ficheros a compartir no residen en el disco duro de los nodos que los comparten, sino que están localizados en otros sitios de la red. Este planteamiento permite que el almacenamiento y el acceso a los datos se realice de forma totalmente anónima.

- **Ancho de banda**: Se centra en el uso efectivo del ancho de banda, balanceo de carga, etc. Este tipo de aplicaciones P2P se utilizan en el ámbito de streaming de audio y vídeo(PeerCast, P2P-Radio, SCVI.net...). Para la transmisión de ficheros muy grandes, se emplea un esquema de partición del fichero en partes que son distribuidas entre distintos nodos de la red que se transmiten de forma independiente. Este tipo de planteamiento es el utilizado por BitTorrent (2003).

- **Almacenamiento**: Presentan soluciones alternativas a las redes de almacenamiento (Storage Area Networks) de las organizaciones. Con este tipo de aplicaciones como OceanStore (2000) y Fairsite (2002), cada nodo de la red utiliza parte de su espacio de almacenamiento para ser utilizado por la aplicación P2P, utilizando un esquema de clave pública y clave privada para la identificación del nodo y los datos.

- **Uso de ciclos de CPU**: El empleo de aplicaciones P2P para el uso de ciclos de CPU permite alcanzar mayor potencial de computación que el supercomputador más caro que existe en la actualidad. En realidad el término de *Scavenging Computational Grid* se refiere a este tipo de aplicaciones P2P. Uno de los proyecto más famosos de este tipo es SETI@HOME. El objetivo de este proyecto es el de detectar señales de radio

emitidas por civilizaciones inteligentes desde fuera de la tierra. Para ello, un telescopio en Puerto Rico recoge parte del espectro electromagnético del espacio y envía los datos al servidor central. Este servidor divide los datos en unidades más pequeñas y se las envía a los millones de coputadores que ceden sus ciclos de CPU al proyecto. En este proyecto participan alrededor de un millón de usuarios, lo que proporciona una capacidad de cómputo de unos 60 TeraFLOPS(superando la capacidad de cualquiera de los spuercomputadores existentes actualmente). Aunque este proyecto tiene algunas características propias de las redes P2P, no puede considerarse como tal. Se parece más a una aplicación cliente/servidor debido a que el servidor central coordina todos los nodos de la red (que unicamente reciben los datos y devuelven sus resultados). Además, no existe comunicación de ningún tipo entre los nodos. Este tipo de computación distribuida se denomina *Volunteer Computing*.

Yafrid-NG se basa en gran parte en este tipo de arquitecturas, intentando proporcionar un mecanismo de *Volunteer Computing* para permitir el renderizado de escenas 3D.

3.1.6. Métodos de Descomposición en Tareas

Para resolver un trabajo utilizando un Grid es necesario descomponer este trabajo en tareas más pequeñas, de tal forma que cada uno de los nodos realice una porción del trabajo de forma paralela a los demás. Dentro de los métodos de descomposición podemos distinguir aquellos que se basan en la descomposición del algoritmo y los que se basan en la descomposición del dominio.

- **Métodos de descomposición del algoritmo**: Este método de descomposición consiste en realizar un estudio del algoritmo y ver que instrucciones se pueden ejecutar de forma paralela. De esta forma cada nodo contiene una parte del algoritmo. El inconveniente de esta aproximación es que se suele producir una alta carga de mensajes en la red de comunicaciones.

- **Métodos de descomposición del dominio**: Se estudia la forma de particionar los datos, de tal manera que el problema pueda resolverse en paralelo por varios procesadores idénticos.

En los métodos de descomposición del dominio se tiene una serie de nodos que ejecutan el mismo algoritmo secuencial a diferentes conjuntos de datos del dominio del problema. La unidad mínima que computa cada nodo se denomina *tarea* o *unidad de trabajo*.

3.1.6.1. Modelos computacionales

El modelo computacional va a determinar la forma en la que los trabajos se distribuyen entre los nodos del sistema. El modelo computacional debe tratar que todos los nodos cuenten con la misma carga de trabajo en cualquier instante de tiempo, evitando los procesadores ociosos.

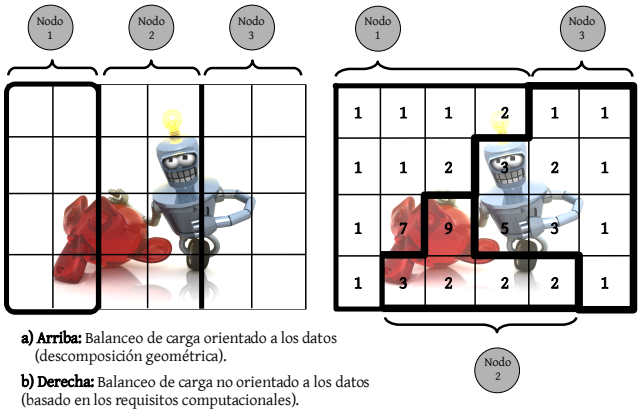

Figura 3.3: Balanceo de carga en modelos computacionales orientados a datos

Existen distintos modelos computacionales. A continuación se presenta una clasificación de éstos:

- **Orientados a Datos**: Las tareas son asignadas antes de que empiece la ejecución del sistema. Los nodos saben antes de comenzar a trabajar las porciones del trabajo que van a tener que realizar. Existen dos formas de realizar el balanceo de carga (ver figura 3.3):

  1. **Balanceo orientado a los datos**: Las tareas se dividen en igual número entre las unidades de procesamiento. Este tipo de distribución sólo resulta interesante si se tarda el mismo tiempo en ejecutar cada una de las tareas.

2. **Balanceo orientado a los requisitos computacionales**: A veces es posible realizar un procesamiento sobre las tareas a realizar y así estimar el tiempo que va a llevar la ejecución de cada una. Esto permite realizar una distribución de las tareas más equitativa. Una vez que se ha realizado la estimación, las tareas se reparten de tal forma que cada nodo tenga la misma estimación de carga computacional. El analisis previo del trabajo para la estimación de cada una de las tareas no debe suponer una carga de tiempo grande en relación con el tiempo de procesamiento de cada tarea.

- **Bajo Demanda**: Los modelos vistos anteriormente dependen de los requisitos computacionales de las tareas. Para la distribución de las tareas, utilizando los modelos previos, es necesario un análisis de las tareas que a veces puede ser impreciso o costoso. En estas ocasiones es mejor utilizar un modelo computacional bajo demanda (ver figura 3.4).

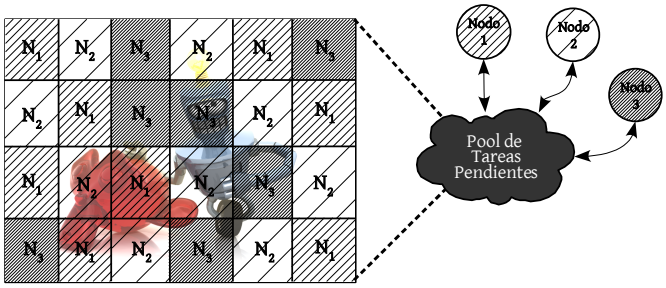

Figura 3.4: Distribución bajo demanda

Con esta aproximación los trabajos se van asignando de forma dinámica. Cuando un nodo termina con la ejecución de una tarea se le asigna otra y así sucesivamente hasta que se realizan todas. Utilizando esta técnica se puede obtener un balanceo de carga poco eficiente, ya que no se tiene en cuenta la complejidad de las unidades de trabajo. Si la tarea más costosa se dejase para el final, esta retrasaría la conclusión del trabajo y algunos de los nodos permanecerían ociosos. Este problema se podría resolver haciendo una estimación de las tareas más complejas para que sean asignadas antes que las tareas más simples. Así se minimiza el tiempo que los nodos esperan ociosos mientras finaliza

la ejecución del trabajo.

- **Híbridos**: Muchas veces la mejor forma de asignar las tareas es utilizando una aproximación híbrida de las dos aproximaciones anteriores. Este modelo computacional es ideal cuando se puede realizar un análisis del tiempo de ejecución de cada tarea, pero sin mucha exactitud.

3.2. Middlewares

3.2.1. Introducción

Se puede definir un *Middleware* como un software de conectividad consistente en una serie de servicios que permiten que multiples procesos, ejecutándose en una o varias máquinas, interactuen a través de una red de comunicaciones (ver figura 3.5).

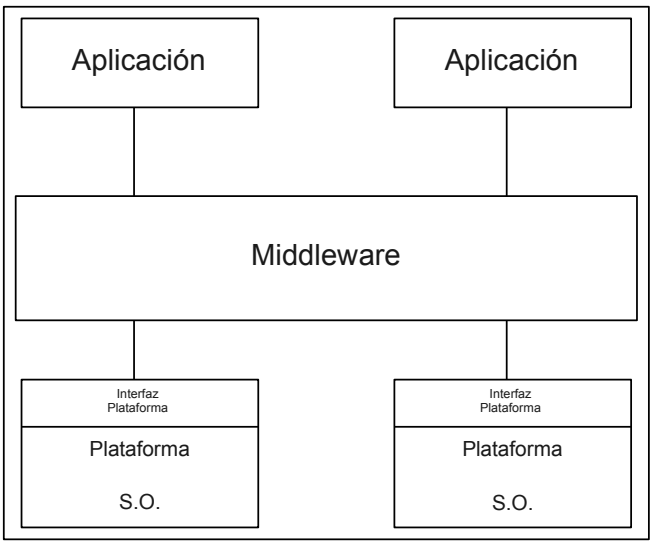

Figura 3.5: Esquema general de un *Middleware*

Un *Middleware* debe:

- Ocultar la heterogeneidad de los componentes hardware, sistemas operativos y protocolos de comunicación.
- Proporcionar interfaces de alto nivel, uniformes y estandar, a los desarrolladores para que las aplicaciones sean compuestas, reusables, portables e interoperables de una forma sencilla.
- Proporcionar una serie de servicios de propósito general para facilitar la colaboración entre aplicaciones.
- Aislar a los programadores de la localización de los recursos.

En la sección actual se analizarán distintos *Middlewares* haciendo especial hincapie en *Ice (Internet Communication Engine)* por ser la opción elegida para la implementación de Yafrid-NG.

3.2.2. CORBA

En 1989 el *Object Management Group*(OMG) se formó para resolver el problema que suponía desarrollar aplicaciones distribuidas portables para sistemas heterogéneos. La primera de las especificaciones clave propuesta por el OMG fue la *Object Management Architecture* (OMA) y su nucleo, la especificación CORBA. Éstos proporcionan una arquitectura que es lo suficientemente rica y flexible como para adaptarse a una amplia variedad de sistemas distribuidos.

CORBA[7] *(Common Object Request Broker Architecture)* es un estandar que permite que varios componenetes software, escritos en distintos lenguajes y ejecutados en diferentes máquinas, trabajen de forma conjunta. CORBA utiliza un paradigma de orientación a objetos y facilita la ejecución de métodos remotos.

La especificación de CORBA indica que la interacción entre los objetos debe realizarse a traves de un ORB *(Object Request Broker)*. Un ORB es el mediador entre los clientes y los objetos. Está encargado de garantizar la interoperabilidad.

Como todas las tecnologías, CORBA tiene una terminología única asociada. Algunos de estos terminos son similares a los de otras tecnologías y otros son específicos de CORBA. A continuación se presenta una lista con los terminos más importantes[23]:

- **Objeto CORBA**: Un objeto CORBA es una entidad "virtual" capaz de ser alojada por un ORB y atender las invocaciones de los clientes. Es virtual porque realmente no existe hasta que no se realiza una implementación concreta en un lenguaje de programación.

- **Cliente**: Un cliente es una entidad que realiza invocaciones sobre un objeto CORBA. Un cliente puede existir en un espacio de direcciones que está completamente separado del objeto CORBA, o el cliente y el objeto CORBA pueden existir dentro de la misma aplicación. El término cliente solo es significativo dentro del contexto de una invocación

determinada, ya que la aplicación que actua como cliente para una petición puede actuar como servidor de otras peticiones.

- **Servidor**: Un servidor es una aplicación en la que uno o más objetos CORBA existen. De la misma forma que con los clientes este término es solo significativo en el contexto de una petición particular.

- **Petición**: Una petición es una invocación de una operación sobre un objeto CORBA por parte de un cliente. La petición fluye desde el cliente al objeto objetivo en el servidor. Este objeto enviará los resultados si se requiere alguna respuesta.

- **Referencia a objeto**: Una referencia a objeto es un manejador utilizado para identificar y localizar un objeto CORBA. Para los clientes, las referencias a objetos son entidades opacas. Un cliente utiliza las referencias a los objetos para realizar peticiones, pero no está capacitado para crear sus propias referencias a objetos ni puede acceder o modificar los contenidos de una referencia a objeto. Una referencia a objeto se refiere aun solo objeto CORBA.

- **Sirviente**: Un sirviente es una entidad en un lenguaje de programación que implementa uno o más objetos CORBA. Se dice que los sirvientes *encarnan* objetos CORBA porque proporcionan implementaciones para esos objetos.

Uno de los componentes más destacables de CORBA es el POA *(Portable Object Adapter)*. El POA es el componente encargado de dirigir las invocaciones a un sirviente local, o para el balanceo de carga, a otro servidor diferente. El POA está compuesto por tres componentes:

- La referencia al objeto.

- El mecanismo de conexión entre peticiones.

- El código asociado que realiza la operación.

Un objeto CORBA define su interfaz mediante un lenguaje denominado IDL *(Interface Definition Language)*. Esta interfaz especifica el contrato que el servidor ofrece al cliente.

Cuando un cliente quiera realizar una petición utilizará las operaciones definidas en la interfaz, realizando el *marshalling* de los datos para su envio. En el lado del servidor se realizará el *unmarshalling* de los argumentos, se procesará la petición y, de forma similar a la descrita, se enviarán los resultados al cliente (ver figura 3.6).

La interfaz IDL se especificará con una sintaxis determinada, independiente de cualquier lenguaje de programación empleado por el cliente o el servidor. Después se generará código para el lenguaje deseado. El objetivo es el de separar la interfaz de la implementación del objeto.

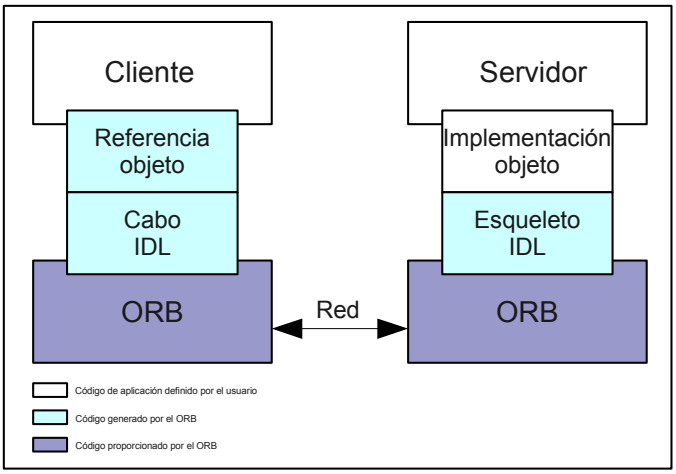

Figura 3.6: Infraestructura de CORBA

Un ejemplo de interfaz IDL podría ser:

```
module Demo{
  interface Hello{
    void puts(in string arg);
  };
};
```

En esta interfaz se define una operación llamada puts. Esta operación tiene un argumento de entrada, arg, y no devuelve ningun resultado. Al generar el código para esta interfaz, entre otros, se generarán los cabos para el cliente y el esqueleto para el servidor. Después de la generación de código el usuario deberá implementar el comportamiento de la operación puts.

Una implementación de la operación puts en java podría ser la siguiente:

```
class Hello_Impl extends Demo.HelloPOA
{
  public void puts(String arg)
  {
    System.out.println(arg);
  }
}
```

El siguiente paso después de implementar la operación puts sería la implementación de los procesos cliente y servidor. En la aplicación servidor se instancia un objeto de la clase definida anteriormente. En el cliente se crea una referencia al objeto remoto y se invoca la operación puts.

El problema mas importante de CORBA es la complejidad de su arquitectura y de sus interfaces. Además también presenta problemas con el mapping al lenguaje C++, sobre todo en lo referente al manejo de excepciones, la seguridad en el manejo de hilos y la gestión de la memoria.

3.2.3. RMI

Java RMI (Remote Method Invocation)[8] permite la creación de aplicaciones distribuidas basadas en Java, en las cuales los métodos de los objetos remotos se pueden invocar desde otras máquinas virtuales, que posiblemente se encuentren en distintos host. RMI utiliza la serialización de los objetos para realizar el *marshalling* y el *unmarshalling* de los parámetros sin truncar los tipos, de tal manera que haya un soporte verdadero del polimorfismo.

El mecanismo utilizado por java RMI es similar al utilizado por CORBA y otros middlewares. Es necesario especificar una interfaz para el objeto remoto. Dicha interfaz está escrita también en Java. Un ejemplo de interfaz sería:

```
public interface Hello extends java.rmi.Remote {
void puts (String str) throws java.rmi.RemoteException;
}
```

Como se puede observar en la interfaz expuesta anteriormente, es necesario heredar de la interfaz *java.rmi.Remote*. Para la compilación de la interfaz es necesario hacer uso de un

compilador de proxies. Se utilizará un esqueleto como base, en el lado del servidor, y un proxy para realizar las invocaciones remotas, en el lado del cliente.

Para implementar un objeto remoto hay que crear una clase que herede del esqueleto generado e implemente el comportamiento de las operaciones especificadas. Por último, para poder acceder a un objeto remoto es necesario incluirlo en un *registry* de objetos.

Java RMI soporta características más avanzadas como la activación implicita, pero tiene claras restricciones. La más significativa es que cliente y servidor deben estar implementados en Java, lo que supone una clara limitación a la hora de desarrollar aplicaciones distribuidas.

3.2.4. Web Services/SOAP

El W3C[9] define un **Web Service** como un sistema software diseñado para permitir la interacción entre dos máquinas sobre una red de comunicaciones. Para que esta interacción sea posible se asume que existe una descripción de las operaciones soportadas por el servidor. Esta descripción está descrita en un formato procesable a nivel máquina denominado **WSDL** (**Web Services Description Language**). De esta forma el cliente puede conocer los métodos ofrecidos por el servidor, sus parámetros y los tipos de los parámetros sólo con consultar el correspondiente WSDL.

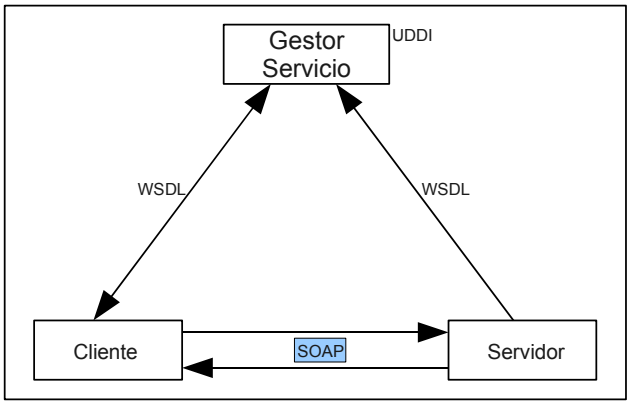

Figura 3.7: Arquitectura de los *Web Services*

Para interactuar con el servidor se utilizan mensajes **SOAP** (**Simple Object Adapter Protocol**). **SOAP** es un protocolo para el intercambio de mensajes XML, normalmente utilizando

HTTP/HTTPS. SOAP forma los cimientos de los Web Services, proporcionando una capa de intercambio de mensajes sobre la que se pueden construir servicios más abstractos. Existen varios patrones de mensajes para SOAP, pero el más común es el de RPC (Remote Procedure Call) en el que un nodo (cliente) envía una petición a otro nodo (servidor), éste la procesa y devuelve los resultados.

El **UDDI** (**Universal Description Discovery and Integration**) es un protocolo para publicación y descubrimiento de metadatos acerca de los *Web Services* que permite que las aplicaciones los localicen, ya sea durante la etapa de diseño o la de ejecución.

A continuación se presenta un ejemplo de una aplicación descrita con WSDL:

```
<?xml version="1.0"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
 xmlns:tns="http://example.com/stockquote.wsdl"
 xmlns:xsd1="http://example.com/stockquote.xsd"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
 <schema targetNamespace="http://example.com/stockquote.xsd"
  xmlns="http://www.w3.org/2000/10/XMLSchema">
   <element name="TradePriceRequest">
    <complexType>
     <all>
      <element name="tickerSymbol" type="string"/>
     </all>
    </complexType>
  </element>
  <element name="TradePrice">
   <complexType>
    <all>
     <element name="price" type="float"/>
    </all>
   </complexType>
  </element>
 </schema>
</types>
<message name="GetLastTradePriceInput">
 <part name="body" element="xsd1:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
 <part name="body" element="xsd1:TradePrice"/>
```

```
</message>
<portType name="StockQuotePortType">
 <operation name="GetLastTradePrice">
  <input message="tns:GetLastTradePriceInput"/>
  <output message="tns:GetLastTradePriceOutput"/>
 </operation>
</portType>
<binding name="StockQuoteSoapBinding"
type="tns:StockQuotePortType">
 <soap:binding style="document"
 transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="GetLastTradePrice">
  <soap:operation
  soapAction="http://example.com/GetLastTradePrice"/>
   <input>
    <soap:body use="literal"/>
   </input>
   <output>
    <soap:body use="literal"/>
   </output>
 </operation>
</binding>
<service name="StockQuoteService">
 <documentation>My first service</documentation>
  <port name="StockQuotePort"
  binding="tns:StockQuoteBinding">
   <soap:address location="http://example.com/stockquote"/>
  </port>
</service>
</definitions>
```

El principal inconveniente de los *Web Services* es que la especificación de las interfaces es bastante tediosa y compleja. Además se contemplan aspectos que nada tienen que ver con la especificación de la interfaz. Si se compara con la especificación de la misma interfaz utilizando SLICE se puede ver la gran diferencia existente[24]:

```
interface StockQuoteService
{
float GetLastTracePrice(string tickerSymbol);
};
```

3.2.5. .NET Remoting

.NET Remoting[6] es una API de programación desarrollada por Microsoft en la versión 1.0 del *Framework* .NET. Se trata de una tecnología similar a CORBA y a Java RMI. .NET Remoting permite que un proceso cliente envíe un mensaje a un proceso servidor y obtenga una respuesta con independencia de si ambos procesos se están ejecutando en distintos dominios de aplicación, en distintos procesos o en distintas máquinas. El objetivo del framework .NET es el de proporcionar una plataforma de desarrollo de software que sea independiente de la plataforma empleada. Actualmente existen dos implementaciones libres de .NET que son: *Mono* y *DotGNU*.

.NET Remoting utiliza *proxies* para permitir la invocación remota. Si el cliente está configurado correctamente, sólo es necesario crear una instancia al objeto remoto utilizando **new** (o la función de instanciación del lenguaje utilizado). Cuando un cliente crea una instancia de un objeto remoto, la infraestructura de .NET crea un objeto proxy con la misma interfaz que el objeto remoto. Cuando el cliente invoca una operación sobre este *proxy* y el sistema remoto recibe la invocación, la direcciona al proceso servidor, realiza la invocación sobre el objeto servidor y devuelve el valor al *proxy* del cliente el cual devuelve el valor al propio cliente.

El *runtime* de .NET Remoting almacena el *listener* que atiende las peticiones sobre el objeto remoto en el dominio de aplicación del proceso servidor. En el lado del cliente, todas las peticiones sobre un objeto remoto son gestionadas por el *runtime* de .NET Remoting a través de objetos de tipo *Channel*, los cuales encapsulan el protocolo de transporte y se se encargan de la serialización y el *marshalling* de las peticiones a través del cliente y el dominio de aplicación del servidor. El comportamiento de un objeto de tipo *Channel* es similar al del ORB en la arquitectura CORBA.

Para especificar la interfaz, .NET permite utilizar cualquiera de los lenguajes soportados por .NET. No hace falta compilar dicha interfaz y no hay que tener en cuenta convenios especiales de nombrado.

3.2.6. ZeroC ICE

3.2.6.1. Introducción

ICE (Internet Communication Engine)[25] es un middleware, desarrollado por **ZeroC**[11], orientado a objetos. Basicamente ésto significa que ICE proporciona herramientas, API's y soporte de librerías para construir aplicaciones cliente/servidor orientadas a objetos. Las aplicaciones ICE se pueden ejecutar en ambientes heterogéneos: cliente y servidor pueden estar implementados en distintos lenguajes, pueden ejecutarse sobre diferentes sistemas operativos y arquitecturas, y pueden comunicarse usando una variedad de tecnologías de red. El código fuente de estas aplicaciones puede portarse de manera independiente al entorno de desarrollo. Los principales objetivos de ICE son los siguientes:

- Proporcionar una plataforma middleware orientada a objetos adecuada para su uso en entornos heterogéneos.
- Proporcionar un conjunto de prestaciones que permitan el desarrollo de aplicaciones distribuidas reales en una amplia variedad de dominios.
- Evitar la complejidad innecesaria, haciendo que la plataforma sea facil de aprender y de usar.
- Proporcionar una implementación que proporciona seguridad intrinseca, haciendola adecuada para su uso sobre redes públicas inseguras.

Abreviando, se puede decir que el objetivo de ICE es el de proporcionar una plataforma tan potente como CORBA pero evitando complejidad innecesaria y los errores cometidos con CORBA.

Como la gran mayoría de middlewares ICE también tiene su propia terminología, bastante común al resto de las terminologías mencionadas anteriormente.

El término **cliente** designa una entidad activa. Los cliente solicitan servicios a los servidores. El término **servidor** designa una entidad pasiva. Los servidores proporcionan servicios en respuesta a las peticiones por parte de los clientes. Normalmente, en las aplicaciones los servidores no son puramente servidores ya que pueden funcionar como servidores atendiendo

peticiones por parte de los clientes, pero también pueden ser clientes de otro servicio para satisfacer la petición de los clientes. De forma análoga, ocurre lo mismo con los clientes. Por ejemplo, un cliente puede realizar una operación que requiere cierta cantidad de tiempo, pasando un *objeto callback* al servidor que se utilizará para notificar el fin de la operación al cliente. En este caso particular el cliente actua como cliente cuando realiza la petición y como servidor cuando se le notifica que la operación ha concluido.

Otro de los conceptos importantes es el de **objeto ICE**. Un **objeto ICE** es una entidad conceptual o abstracción. Se caracteriza por los siguientes aspectos:

- Un objeto ICE es una entidad en espacio de direcciones local o remoto que puede responder a las peticiones de los clientes.
- Un solo objeto puede ser instanciado en un solo servidor o, de forma redundante, en varios servidores.
- Cada objeto tiene una o varias interfaces. Una interfaz es una colección de operaciones que un objeto proporciona. Los clientes realizan peticiones invocando estas operaciones.
- Una operación tiene cero o más parámetros y puede, o no, tener un valor de retorno. Los parámetros y el valor de retorno tienen un tipo específico. Los parámetros tienen un nombre y una dirección. Los parámetros de entrada se inicializan en el cliente y se le pasan al servidor. Los parámetros de salida se inicializan en el servidor y se le pasan al cliente (el valor devuelto es un parámetro especial de salida).
- Un objeto ICE tiene una interfaz principal. Además un objeto Ice puede proporcionar cero o más interfaces alternativas, conocidas como *facets*.
- Cada objeto ICE tiene una única *identidad de objeto*. La identidad de un objeto es un valor identificativo que lo distingue del resto de objetos.

Después de definir lo que es un objeto Ice es importante saber lo que es un **Proxy**. Un **Proxy** es un artefacto local al espacio de direcciones del cliente. Representa el (posiblemente remoto) objeto Ice para el cliente. Un proxy actúa como el *embajador* local del objeto ICE.

Un proxy encapsula toda la información necesaria para permitir la invocación de operaciones remotas. Existen diferentes tipos de proxy:

- **Stringfied proxies**: La información contenida en un proxy se puede expresar en forma de cadena. Por ejemplo: *SimplePrinter:tcp -h ¡ip¿ -p ¡puerto¿*. El runtime de Ice proporciona operaciones que permiten convertir un proxy a su forma de cadena y viceversa.

- **Direct proxies**: Un **direct proxy** es un proxy que contiene la identidad de un objeto junto con la dirección el la que se ejecuta su servidor. La dirección se especifica con el identificador de un protocolo(como TCP/IP o UDP) y una dirección específica (como el *hostname* y el número de puerto). Para contactar con un objeto, ICE utiliza la información contenida en el proxy.

- **Indirect proxies**: Un proxy indirecto tiene dos formas. Puede proporcionar sólo la identidad del objeto (*SimplePrinter*) o la identidad junto con el identificador del *adaptador de objetos* (*SimplePrinter@PrinterAdapter*). Un objeto que es accesible usando solo su identidad se llama *objeto well-known*.

Otro de los conceptos importantes en ICE es el de replicación. En ICE la replicación consiste en hacer que los *adaptadores de objetos* (y sus objetos) estén disponibles en multiples direcciones. El objetivo es el de proporcionar redundancia, ejecutando el mismo servidor en varios computadores. Si alguno de estos computadores falla, el servidor sigue estando disponible en cualquiera de los demás servidores. Una forma de conseguir la replicación es la de indicar múltiples direcciones para un objeto. Por ejemplo:

```
SimplePrinter:tcp -h server1 -p 10001: tcp -h server2 -p 10002
```

Aquí se indica que el objeto con identidad *SimplePrinter* está disponible en dos direcciones, una en el host *server1* y otra en el host *server2*. Cuando se invoca una operación sobre *SimplePrinter* el runtime de ICE elegirá uno de ellos de forma aleatoria para el primer intento de conexión, si la conexión falla se intentará con el resto de direcciones.

Otra forma de proporcionar **replicación** es utilizar **replica groups**. Este método requiere el uso de un servicio de localización.

Un **replica group** tiene un único identificador y consiste en un número de adaptadores de objetos. Un adaptador de objetos puede ser miembro, como máximo, de un **replica group**. El comportamiento del servicio de localización que contiene un **replica group** es un detalle de implementación. Por ejemplo, el servicio de localización podría decidir devolver la dirección de todos los adaptadores de objetos en el grupo y el cliente podría elegir uno de ellos de forma aleatoria o utilizando alguna heurística.

Una de las características importantes de ICE es que permite programación asíncrona. Cuando se realiza una invocación remota, el hilo que realiza la invocación se bloquea hasta que se reciben los resultados de la operación. Algunas veces esto no conviene. Imaginemos que desde una interfaz gráfica se realiza una operación que requiere una gran capacidad de cómputo. Mientras esta operación se resuelve la interfaz se quedaría bloqueada. Para solucionar este problema ICE permite la programación asíncrona.

Asyncronous Method Invocation(AMI) es el término que se utiliza para describir el soporte, en el lado del cliente, para la programación asíncrona. Usando AMI una invocación remota no bloquea el hilo que la realiza. El hilo sigue con su ejecución de forma normal y la aplicación es informada por el ICE *runtime* cuando la respuesta se recibe.

Asyncronous Method Dispatch(AMD) es el equivalente en el lado del servidor de *AMI*. El número de peticiones síncronas que un servidor puede atender depende del modelo de concurrencia utilizado. Haciendo uso de *AMD*, un servidor suspende el procesamiento de una petición para liberar el hilo que despacha las peticiones tan pronto como sea posible. Cuando la petición se ha realizado el servidor envía la respuesta de forma explicita. Tipicamente, una operación AMD encola las peticiones, para procesarlas posteriormente utilizando un hilo de la aplicación. Así, el servidor minimiza el uso de los hilos que despachan las peticiones y puede atender miles de clientes simultáneos.

3.2.6.2. SLICE

SLICE *(Specification Language for ICE)* es el mecanismo de abstracción, proporcionado por ICE, para separar la interfaz de los objetos de su implementación. **SLICE** establece un contrato entre el cliente y el servidor que describe los tipos y las interfaces utilizadas por la aplicación. Esta descripción es independiente del lenguaje de implementación, por lo que no

importa si el cliente y el servidor están escritos en el mismo lenguaje.

Las definiciones en SLICE son compiladas para un lenguaje de implementación particular(ver Figura 3.8). El compilador traduce las definiciones independientes del lenguaje en definicioes especificas del lenguaje y APIs. Los algoritmos de traducción para varios lenguajes de implementación se conocen como *language mappings*. Actualmente ICE define *language mappings* para C++, Java, C#, Visual Basic .NET, Python, Ruby y PHP.

A continuación se presenta un ejemplo de interfaz escrita en SLICE. Esta interfaz se compilaría para cualquiera de los lenguajes que soporta ICE para su posterior implementación:

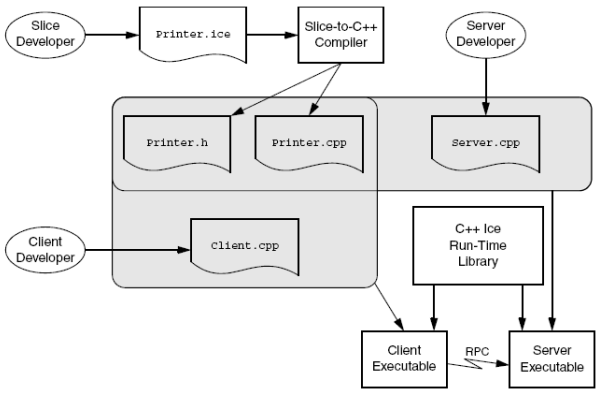

Figura 3.8: Proceso de desarrollo si cliente y servidor están implementados utilizando el mismo lenguaje

SLICE es un lenguaje puramente declarativo ya que solo permite la definición de interfaces y tipos (pero no su implementación). No hay ninguna forma de escribir sentencias ejecutables en SLICE.

3.2.6.3. Servicios Avanzandos de ICE

El núcleo ICE proporciona una plataforma cliente-servidor para el desarrollo de aplicaciones distribuidas. Pero las aplicaciones distribuidas demandan unos servicios como pueden ser la activación bajo demanda, la distribución de proxies, configuración de aplicaciones, comunicaciones seguras etc... ICE proporciona una serie de servicios que proporcionan muchos de estas características. A continuación se comentan **IceGrid**, **Glacier2** y **IceSSL** que son los que usa Yafrid-NG.

IceGrid es el servicio de localización y activación de ICE. Las características de **IceGrid** son:

- **Servicio de localización**: IceGrid permite que los clientes accedan de forma indirecta a los servidores.
- **Activación de los servidores bajo demanda**: IceGrid permite activar los servicios bajo demanda, esto es, cuando un cliente accede a un objeto alojado por el servidor. Esta activación se realiza de forma transparente al usuario.
- **Distribución de la aplicación**: IceGrid permite la distribución de la aplicación a un conjunto de computadores de una forma sencilla.
- **Replicación y balanceo de carga**: IceGrid permite la replicación agrupando los *adaptadores de objetos* de varios servidores en un *adaptador de objetos* virtual. Además, IceGrid monitoriza la carga de cada computadore y utiliza esa información para decidir que endpoints devolver al cliente.
- **Sesiones y reserva de recursos**: IceGrid establece una sesión para permitir la reserva de los recursos. IceGrid evita que otros clientes utilicen recursos ya reservados hasta que el cliente los libera o expira la sesión.
- **Recuperación automática de errores**: ICE soporta reintentos automáticos y recuperación de errores en cualquier proxy que contenga varios endpoints.
- **Consultas dinámicas**: Además de la localización indirecta, las aplicaciones pueden interactuar directamente con IceGrid para localizar objetos de varias formas.
- **Monitorización del estado**: IceGrid soporta interfaces Slice que permiten a las aplicaciones monitorizar sus actividades y recibir notificaciones acerca de eventos significativos.
- **Administración**: IceGrid incluye herramientas de administración por línea de comandos y con interfaz gráfica.

- **Despliegue**: Utilizando archivos XML se pueden describir los servidores que van a ser desplegados en cada computador. Utilizando plantillas se simplifica la descripción de servidores iguales.

Un dominio IceGrid consiste en un *registry* y cualquier número de nodos (ver figura 3.9. Juntos cooperan para gestionar la información y los procesos servidor que componen la aplicación. Cada aplicación asigna servidores a nodos particulares. El registry mantiene un registro con esta información, mientras que los nodos son los responsables de iniciar y monitorizar sus procesos servidores. En una configuración típica un nodo se ejecuta en cada una de las computadores que alojan un servidor ICE. El *registry* no consume muchos recursos, por lo que normalmente se ejecuta en la misma máquina que un nodo.

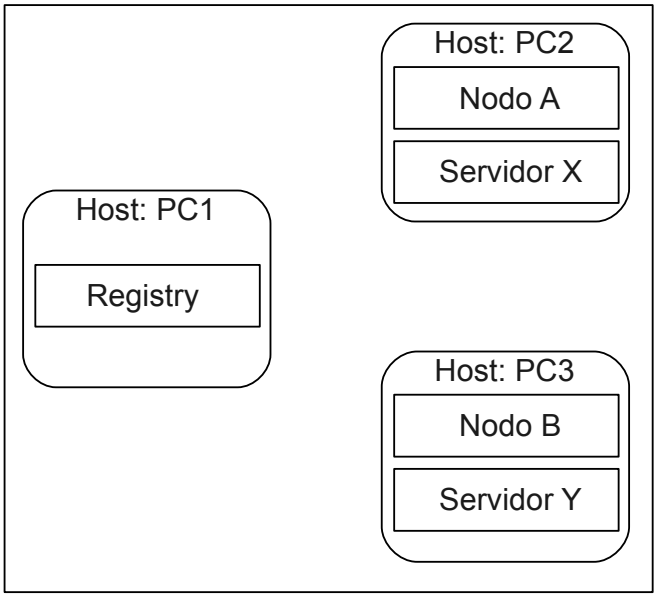

Figura 3.9: Arquitectura de una aplicación simple con IceGrid

En IceGrid el *despliegue* es el proceso mediante el cual se describe la aplicación al *registry*. En esta descripción se puede incluir diversa información como puede ser los *replica groups*, nodos, servidores, adaptadores de objetos, objetos *well known*... que intervienen en la aplicación. IceGrid utiliza *descriptores* para el despliegue de la aplicación. Los *descriptores* están escritos en XML y se pueden generar de varias formas, ya sea escribiendo el XML con algún editor de texto, creando el descriptor de la aplicación utilizando una interfaz gráfica

de administración o de utilizando la interfaz administrativa de programación que proporciona IceGrid.

Un ejemplo de descriptor de aplicación:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
             <server id="EncoderServer"
             exe="/opt/ripper/bin/server"
             activation="on-demand">
                 <adapter name="EncoderAdapter"
                 id="EncoderAdapter"
                 register-process="true"
                 endpoints="tcp"/>
             </server>
        </node>
    </application>
</icegrid>
```

En el ejemplo anterior se presenta una aplicación que se llama *Ripper* y que se compone de un solo nodo llamado *Node1*. En este nodo se aloja un servidor cuyo identificador es *EncoderServer* y que se activa bajo demanda, es decir, cuando alguno de los clientes realice una invocación sobre él. El servidor utiliza un adaptador de objetos llamado *EncoderAdapter* y que utiliza el protocolo TCP. Como se puede observar no es necesario especificar el puerto ni la dirección IP si se utiliza IceGrid. El atributo *register-process* permite que IceGrid desactive el servidor cuando sea necesario.

El servicio **Glacier2** es una solución cortafuegos para las aplicaciones ICE. Se puede ver Glacier2 como el cortafuegos utilizado por parte del servidor. Glacier2 nos permite manejar situaciones como la presentada en la figura 3.10 sin demasiadas consecuencias en las partes cliente y servidor.

Las ventajas de utilizar Glacier2 son las siguientes:

- Los clientes requieren unos cambios mínimos para utilizar Glacier2.
- Solo es necesario un puerto *front-end* en el router para proporcionar cualquier número de servicios. El router Glacier2 se encarga de atender las conexiones que el router permite.

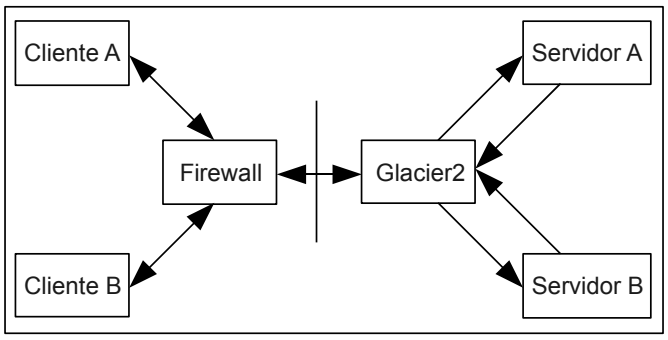

Figura 3.10: Escenario creado por Glacier2

- El número de conexiones a los servidores se reduce. Glacier2 actua como un concentrador estableciendo una sola conexión con cada servidor para atender peticiones de cualquier número de clientes.
- Los servidores no son conscientes de la existencia de Glacier2. Para un servidor, Glacier2 no es nada más que otro cliente.
- Se soportan las retrollamadas, evitando así que se tengan que abrir nuevas conexiones desde los servidores a los clientes.
- Glacier2 es independiente de la especificación Slice de la aplicación. Por esto es muy eficiente, enruta los mensajes sin realizar el *unmarshalling* de su contenido.
- Además Glacier2 proporciona soporte para la gestión de sesiones definida por el usuario, autenticación y tiempos de inactividad.

Para utilizar Glacier2 no son necesarios muchos cambios en la aplicación. Simplemente hay que escribir un archivo de configuración, decidir si utilizar el gestor de sesiones interno o proporcionar uno customizado, arrancar el router, modificar el cliente para que utilice el router y establezca una sesión con éste.

IceSSL es un *plug-in* incluido por ICE que nos permite añadir seguridad a las aplicaciones distribuídas utilizando el protocolo SSL (ver subsección 3.1.4.3). Para el desarrollo de aplicaciones seguras es importante proteger la información, asegurar su integridad y verificar la identidad de las dos partes que intervienen en la comunicación.

Para integrar IceSSL en una aplicación, normalmente, no es necesario modificar el código fuente, pero implica las siguientes tareas administrativas:

- Crear una Infraestructura de clave pública (ver subsección 3.1.4.2).
- Configurar el *plug-in* IceSSL.
- Modificar la configuración de la aplicación para que instale el *plug-in* IceSSL y utilice conexiones seguras.

ICE proporciona un mecanismo que permite instalar extensiones (como IceSSL) dinamicamente sin necesidad de cambiar el código de la aplicación. Además, ICE incluye un script en python, **iceca**, que nos abstrae de la complejidad de OpenSSL y nos permite realizar con relativa facilidad las siguientes tareas:

- Inicializar una Autoridad de Certificados (CA) raíz.
- Generar peticiones de nuevos certificados.
- Firmar las peticiones de certificados para convertirlas en un certificado valido.
- Convertir los certificados para su uso en plataformas con requerimientos específicos.

3.2.6.4. Justificación del uso de ICE

Una vez estudiadas las características principales de ICE se puede justificar su uso para implementar la parte de comunicaciones de Yafrid-NG:

- Se distribuye bajo una licencia GPL.
- Se trata de una arquitectura facil de aprender y de usar.
- Proporciona una serie de servicios que facilitan el desarrollo de aplicaciones distribuídas.
- Permite separar interfaz de implementación.
- Es una arquitectura multilenguaje y multiplataforma.

- Existe una comunidad de desarrollo muy activa.
- Se ofrece mucho soporte, ya sea en los foros de ZeroC[11] o mediante su revista *Connections*[10] publicada periódicamente.

Además de las ventajas expuestas anteriormente, existen otros aspectos que hacen de ICE un middleware más apropiado, que por ejemplo CORBA, para los objetivos perseguidos. Entre éstos podemos destacar:

- Ice permite las llamadas asíncronas del lado del servidor (AMD).
- Al contrario que CORBA, ICE proporciona *mappings* para lenguajes como PHP, Visual Basic y C#.
- La complejidad de CORBA es mucho mayor que la ICE por lo que la curva de aprendizaje es más pronunciada.

3.3. Síntesis de Imagen Realista

3.3.1. Introducción al proceso de síntesis

En los últimos años los gráficos por computador han ido tomando más importancia en diferentes disciplinas. Han pasado de ser utilizados por la comunidad científica a otros campos como son el mercado cinematográfico, los videojuegos, la visualización médica y los sistemas de diseño asistido por computador. Se define **síntesis de imagen realista** al área que se encarga de la representación de las imágenes de objetos tridimensionales, cuya descripción matemática está almacenada en la memoria de un computador, con el mayor realismo posible de forma que un observador humano no pueda diferenciar si se trata de una fotografía o de una imagen generada mediante un motor de render.

A la hora de construir una imagen interviene un proceso de selección, abstracción y aproximación de las propiedades del objeto a representar. Según [18] se pueden identificar tres niveles de realismo en gráficos por computador, diferenciados en la aproximación, abstracción u omisión de las propiedades de los objetos que forman la escena a representar:

- **Realismo físico**: La imagen proporciona la misma estimulación visual que la escena a representar. Por lo tanto, el modelo matemático debe ser completo y preciso, conteniendo todos los datos referentes a geometría, materiales, iluminación, etc. Además el motor de render debe ser capaz de simular de forma precisa la interacción de la luz en este entorno virtual. Raras veces se alcanza este grado de realismo, debido a las limitaciones de los dispositivos de visualización actuales y a que los métodos son computacionalmente intensivos. Además, el sistema de visión humano no es capaz de percibir toda la información que estos métodos pueden aportar.

- **Fotorrealismo**: La imagen generada debe proporcionar la misma *respuesta visual* que la escena a representar, aunque la energía física que proviene de la imagen pueda ser diferente que la percibida en la escena. Así se pueden tener en cuenta las limitaciones del sistema de visión humano para simplificar el proceso de generación de imagenes por computador. Aunque obtener imágenes fotorrealistas es menos costoso que sus equivalentes físicamente realistas, sigue siendo un proceso muy costoso computacionalmente

y se continúan estudiando alternativas de optimización.

- **Realismo funcional**: La escena generada debe tener la misma *información visual* que la escena a representar. Información se refiere a las propiedades de tamaño, forma, posición y materiales. Algunas veces es preferible el realismo funcional al fotorrealismo debido a que es computacionalmente más sencillo, por ejemplo, en los simuladores de vuelo la información presentada al piloto coincide con la información real de vuelo.

A la hora de construir imágenes fotorrealistas es importante que además de la perspectiva, se comprenda y se simule el comportamiento de la luz y las características del sistema visual humano. Se han desarrollado modelos matemáticos que estudian la interacción de la luz en las superficies. Después de la aparición del microprocesador, los ordenadores tuvieron suficiente potencia como para simular estas interacciones. El nivel de realismo está asociado a la precisión en esta simulación, a mayor precisión se obtendrá un mayor grado de realismo.

Existen algunas limitaciones importantes. La principal es el excesivo coste computacional de los métodos más realistas. Es necesario establecer un nivel de detalle en la escena que se quiere simular. Es importante el uso de algoritmos y métodos que incorporen conocimiento para la ayuda a la evaluación de una escena. Así es posible decidir en que zonas hay que dedicar un mayor esfuerzo computacional.

Los distintos métodos de render existentes tratan de dar solución a la ecuación propuesta por Kajiya[28] en 1986, dicha ecuación(3.1) describe el flujo de energía luminosa en una escena. Se basa en las leyes físicas de la luz y proporciona resultados perfectos mientras que los métodos de render ofrecen aproximaciones a estos resultados ideales.

$$L_o(x, \overrightarrow{w}) = L_e(x, \overrightarrow{w}) + \int_\Omega f_r(x, \overrightarrow{w}', \overrightarrow{w}) L_i(x, \overrightarrow{w}')(\overrightarrow{w}' \cdot \overrightarrow{n}) d\overrightarrow{w}' \tag{3.1}$$

Esta ecuación se puede interpretar como: “dada una posición x y con una dirección $\overrightarrow{w}$ determinada, el valor de iluminación saliente L_o es el resultado de sumar la iluminación emitida L_e y la luz reflejada. La luz reflejada(el segundo término de la ecuación) viene dada por la suma de luz entrante L_i desde todas las direcciones multiplicado por la reflexión de la superficie y el ángulo de incidencia”.

En la siguiente sección se hará una breve introducción a los métodos de render que tratan de aproximar la ecuación3.1 descrita.

3.3.2. Métodos de Render

El proceso de render es el encargado de convertir la descripción de una escena tridimensional en una imagen bidimensional. Durante dicho proceso, se determinará el color de cada uno de los píxeles que forman parte de la imagen resultado.

A continuación se proporcionará una breve introducción a los métodos de renderizado utilizados.

3.3.2.1. ScanLine

Introducido por Bouknight en 1970[12], se trata de uno de los algoritmos de renderizado más básico. Es un algoritmo para la representación punto a punto de una imagen.

Algorithm 1 Bucle general de ScanLine

```
for all pixel de la imagen do
    línea ← Trazar una línea desde la cámara al pixel
    color ← Scanline(línea)                          ▷ Algoritmo 2
end for
```

Algorithm 2 Procedimiento Scanline (línea)

```
punto ← encontrar el punto de intersección más cercano
color ← color de fondo
for all fuente de luz do
    color ← color + iluminación directa
end for
Devolver(color)
```

Una de sus principales ventajas erradica en la rapidez del algoritmo. Permite la simulación de cualquier tipo de sombreado funcionando correctamente con texturas. El principal inconveniente es que no simula de forma realista reflexiones y refracciones de la luz.

3.3.2.2. RayTracing

El método de trazado de rayos, propouesto por Whitted[37], permite calcular de una forma unificada la reflexión y la refracción de la luz, sombras, eliminación de superficies ocultas y otros efectos necesarios para conseguir escenas fotorrealistas.

La idea básica del trazado de rayos es seguir el camino de la luz desde las fuentes emisoras de fotonoes hasta que llegan a la posición del observador (*forward Raytracing*). El problema de esta aproximación es que la mayoría de los rayos nunca llegan al observador. Para evitar trazar rayos que no llegarán al plano imagen se emplea el trazado de rayos hacia atrás (*backward Raytracing*) donde los rayos parten del observador hasta alcanzar los objetos de la escena.

A continuación se muestra el algoritmo de un trazador de rayos básico en pseudocódigo:

Algorithm 3 Bucle general del Raytracing.

```
for all píxel de la imagen do
    rayo ← trazar un rayo desde la cámara al píxel
    color ← Raytracing (rayo)                          ▷ Algoritmo 4
end for
```

Algorithm 4 Procedimiento RayTracing (rayo)

```
punto ← encontrar el punto de intersección más cercano
color ← color de fondo
for all fuente de luz do
    rayo_sombra ← trazar un rayo desde (punto) hasta la fuente de luz
    if el rayo no choca con ningún objeto then
        color ← color + iluminación directa
        if el material tiene propiedad de reflexión then
            color ← color + Raytracing (rayo_reflejado)
        end if
        if el material tiene propiedad de refracción then
            color ← color + Raytracing (rayo_transmitido)
        end if
    else
        color ← negro
    end if
end for
Devolver(color)
```

El metodo de RayTracing calcula, de forma recursiva, la contribución de la luz debido a

la reflexión y refracción que se produce en ciertas superficies. Se definen cuatro tipo de rayos (ver figura 3.11:

- **Rayos primarios o Visuales**: Rayos que parten de la camara. Para cada elemento de la escena se comprueba si el rayo intersecta con alguno de ellos, quedándonos con el punto de intersección más cercano de toda la lista de objetos.
- **Rayos de sombra**: Parten del punto de intersección con el objeto y tienen dirección hacia las fuentes de luz. Se vuelve a comprobar si colisiona con algún objeto, en este caso el punto de origen del rayo estaría en sombra.
- **Rayos reflejados**: Si el objeto en el que intersectó el rayo tiene propiedades de reflexión de tipo espejo, se generará un nuevo rayo reflejado en este punto. Dicho rayo se construirá en un procedimiento recursivo, pasando a comportarse como un rayo primario en la siguiente iteración del algoritmo.
- **Rayos Transmitidos**: En caso de objetos transparentes, y de forma análoga al tratamiento para los rayos reflejados, se generará un rayo transmitido. Este rayo se comportará como primario en la siguiente iteración del algoritmo.

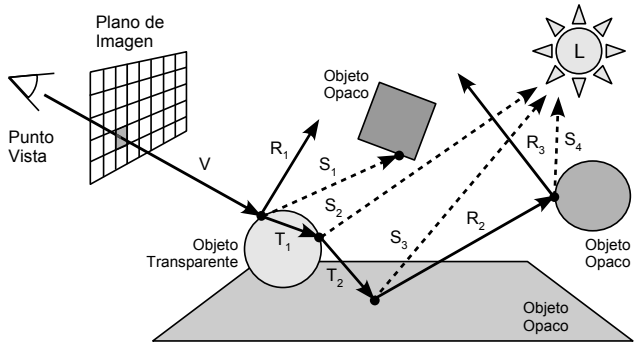

Figura 3.11: Tipos de rayos en RayTracing

Este método es computacionalmente costoso debido a la necesidad de calcular la intersección más cercana de cada rayo con los objetos que hay en la escena. Existen técnicas que

permiten reducir el número de cálculos realizados, reduciendo así el tiempo empleado. Algunas de estas técnicas intentan reducir el número de rayos lanzados analizando la escena y prestando una mayor atención a las zonas que lo requieran, porque las diferencias entre píxeles vecinos son importantes o bien deciden cuándo es conveniente no descender más en el trazado recursivo de rayos. Otras técnicas tratan de encontrar intersecciónes más rápidas entre el rayo y el objeto, éstas son las que más optimizan el tiempo de cómputo. Por último están las técnicas que parten de un volumen que agrupa todos los elementos de la escena y progresivamente van dividiendo el volumen en partes de menor tamaño, agrupando los objetos en esos volúmenes. Ésto nos permite centrarnos en la zona que nos interesa en lugar de contemplar la escena completa.

El método de RayTracing permite simular reflexiones especulares y refracciones pero no está pensado para simular iluminaciones indirectas o sombras difusas.

Ambient Occlusion

La técnica de *ambient occlusion* fue propuesta inicialmente por Zhurov[38] como alternativa a las técnicas de radiosidad para aplicaciones interactivas por su bajo coste computacional. Esta técnica es un caso particular del uso de pruebas de oclusión en entornos con iluminación local para determinar los efectos difusos de iluminación. Se puede definir la ocultación de un punto de una superficie como:

$$W(P) = \frac{1}{\pi} \int_{w \in \Omega} \rho(d(P,w)) cos\theta dw \tag{3.2}$$

Donde $d(P,w)$ es la distancia entre P y la primera intersección con algún objeto de la escena en la dirección de w. El término $\rho(d(P,w))$ es una función con valores entre 0 y 1 que indica la magnitud de iluminación ambiental que viene en la dirección de w. Por último, θ es el ángulo formado entre la normal en P y la dirección de w.

La principal ventaja de esta técnica es que es bastante más rápida que las técnicas que realizan un cálculo correcto de la iluminación indirecta, obtiene resultados suficientemente buenos y con menos ruido. El principal inconveniente es que no es un método de iluminación global y no puede simular efectos complejos como caústicas o contribuciones de luz entre

superficies con reflexión difusa.

Radiosidad

Propuesta inicialmente por Goral y Greenberg[22] esta técnica calcula el intercambio de luz entre superficies. Esta técnica calcula una solución independiente del punto de vista, pero el cálculo de esta solución es muy costoso en tiempo y en espacio de almacenamiento. El cálculo del intercambio de luz entre superficies se hace mediante el factor de forma, calculado mediante la siguiente ecuación:

$$F_{i,j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\theta_i \, cos\theta_j}{\pi r^2} H_{ij} \, dA_j \, dA_i \tag{3.3}$$

Siendo F_{ij} el factor de forma de la superficie i a la superficie j y $cos\theta_i \, cos\theta_j$ es el ángulo entre las normales de los planos de cada parche. pir^2 mide la distancia entre los parches y H_{ij} es el factor de visibilidad (con valores entre 0 y 1). Por último, el término dA_x corresponde al área de la superficie x.

La principal ventaja de la técnica de radiosidad es que ofrece muy buenos resultados para superficies difusas. Además al ofrecer una solución independiente del punto de vista, cuando la iluminación es calculada, puede utilizarse para renderizar la escena desde diferentes ángulos. Por el contrario, si se utiliza para modelos muy complejos se trata de una técnica especialmente costosa. También tiene a suavizar las zonas de penumbra, lo cual no siempre es deseable.

PathTracing

El algoritmo de PathTracing fue propuesto por Kajiya[28] como solución a la ecuación de renderizado (en el mismo artículo en el que fue propuesta) basandose en las ideas de Cook[14]. El método *Distributed RayTracing* traza rayos distribuidos aleatoriamente para conseguir sombras suaves, motion blur y profundidad de campo. El método de PathTracing utiliza esta idea para calcular todos los posibles caminos de la luz.

Este algoritmo se basa en un mecanismo que consiste en trazar rayos para calcular todos los posibles caminos por los que puede venir la luz en un pixel. Este método de resolución

se basa en la técnica de integracion de Monte Carlo que consigue crear rayos uniformemente para todos los caminos. El PathTracing sólo utiliza un rayo reflejado para calcular la iluminación indirecta. Cuando una rayo impacta contra una superficie difusa, sólo se realizará una llamada con un nuevo rayo de cálculo de iluminación indirecta, cuya dirección será aleatoria dentro del dominio de definición de la superficie.

La principal ventaja obtenida utilizando este método es que si la iluminación varía poco (como en una escena exterior) se obtienen resultados sin ruído con pocas muestras.

Photon Mapping

Método de renderizado propuesto por Jensen[27] y que se basa en desacoplar la representación de la iluminación de la geometría. Se realiza en dos pasos, primero se construye la estructura del mapa de fotones (trazado de fotones) desde las fuentes de luz al modelo. En una segunda etapa de render se utiliza la información del mapa de fotones para realizar el renderizado de manera más eficiente.

Cuando se emite un fotón, éste es trazado a través de la escena de igual forma que se lanzan los rayos en el método de RayTracing. Cuando un fotón choca contra una superficie, éste puede ser reflejado, transmitido o absorvido en función de las propiedades del material.

Los fotones son almacenados cuando impactan sobre superficies no especulares. Este almacenamiento se realiza sobre una estructura de datos que está desacoplada de la geometría del modelo y se utilizará en la etapa de render para obtener información sobre qué impactos de fotones están más cerca del punto del que queremos calcular su valor de iluminación.

Figura 3.12: Escena renderizada con varios métodos de render. a) ScanLine b) RayTracing c) Ambient Occlussion d) PathTracing (Skydome) e) PathTracing (Un foco de iluminación directa) f) PathTracing con HDRI

3.4. Sistemas relacionados

En esta sección se comentarán algunos sistemas que utilizan una aproximación de computación paralela para renderizar escenas en 3D.

3.4.1. Dr Queue

Dr Queue[4] es una herramienta que distribuye una tarea en una granja de computadores conectados mediante una red. Los frames repartidos se renderizan por dichos computadores para después juntarlos y obtener el resultado final del proceso. Dr Queue soporta varios motores de renderizado incluyendo: Blender, Maya, Lightwave... También permite la posibilidad de escribir algún script si se necesita algún otro motor de render.

Dentro de Dr Queue se pueden distinguir diferentes roles. Estos roles son: *Master*, *Slave* y *Client*. El *Master* es el gestor del sistema. Contiene toda la información acerca de todos los *slaves* disponibles y las tareas a ejecutar. Los *slaves* renderizarán los frames que serán asignados por el master. Los *slaves* mantienen su configuración local y periodicamente se la remiten al *master*. Los *clients* son programas que pueden realizar algunas operaciones sobre el estado del sistema. Son capaces de recibir información acerca de la cola de trabajo, manipular

esta cola para cambiar el orden en el que los frames se van a renderizar, habilitar nuevos *slaves*... Normalmente estas tareas se realizan mediante herramientas por linea de comandos que cambian el estado actual del sistema. No hay restricciones en el tipo de peticiones que un cliente puede realizar. Sin embargo, algunas de las operaciones permitidas pueden conducir a resultados inesperados e incorrectos.

Durante el proceso de renderizado un *slave* necesita conocer la localización de los recursos así como el script que tiene que ejecutar. Es por esto que Dr Queue necesita un sistema de almacenamiento compartido. Así, todos los *slaves* son capaces de acceder a los recursos compartidos y realizar las tareas que se les asignan.

El hecho de usar Dr Queue implica que todos los recursos están centralizados y el usuario tiene control total sobre el sistema. La siguiente aproximación utiliza un grid de computadores para resolver el mismo problema. Esto implica la descentralización y el anonimato de los usuarios que se involucran en el proyecto.

3.4.2. BURP

BURP[3] es el acrónimo de *Berkeley Ugly Rendering Project*. BURP es un sistema público que utiliza los sistemas distribuidos para renderizar escenas en 3D. Está basado en **BOINC**[2] (**Berkeley Open Infraestructure for Network Computing**) y actualmente se encuentra en la fase de pruebas, por lo que hoy en día BURP no está trabajando al 100 %.

BOINC hace uso del *Volunteer Computing* para resolver tareas complejas. Si un usuario quiere donar sus ciclos de CPU y sus recursos se tiene que instalar el software proporcionado por BOINC y unirse al proyecto BURP. Una vez que todos los pasos se han realizado correctamente el sistema funciona de la siguiente forma:

1. El PC cliente obtiene algunas instrucciones del servidor. Estas instrucciones dependerán de la máquina cliente, por lo que se apdaptarán a la memoria y los ciclos de CPU disponibles.

2. Después, el cliente se descarga todos los archivos y ejecutables necesarios y realiza las operaciones.

3. Una vez que se tienen los resultados se envían al servidor que los procesa y obtiene la solución requerida.

4. El cliente solicita nuevas instrucciones y el proceso comienza de nuevo.

Cada unidad de trabajo se envía, al menos, a dos computadores. Cuando el usuario remite los resultados el servidor comparará ambos y si son correctos una serie de créditos serán asignados a cada uno de los clientes. Estos créditos se utilizarán para realizar una clasificación de todos los usuarios suscritos al proyecto. Este sistema de créditos se utilizará posteriormente para obtener algunas estadísticas que pueden ser consultadas por los usuarios para comprobar su participación total.

Capítulo 4

Metodología de trabajo

4.1. Introducción

En este capítulo se comentará con un alto nivel de detalle la construcción de Yafrid-NG, prestando especial atención a las partes más destacables del sistema.

En el primer punto se expondrá la arquitectura del sistema, indicando cuales son las partes de las que se compone Yafrid-NG. Posteriormente, se se explicará el flujo de trabajo desde que un cliente solicita la ejecución de una tarea de render hasta que obtiene los resultados en su máquina. La ejecución de un render involucra los siguientes pasos (ver figura 4.1):

- Obtención de información acerca de la escena a renderizar y cálculo de las unidades de trabajo.
- Establecimiento de sesión con el grid.
- Reserva de recursos para la ejecución de la tarea.
- Paso de archivos necesarios.
- Etapa de renderizado.
- Obtención de los resultados por parte del cliente.
- Liberación de los recursos utilizados.

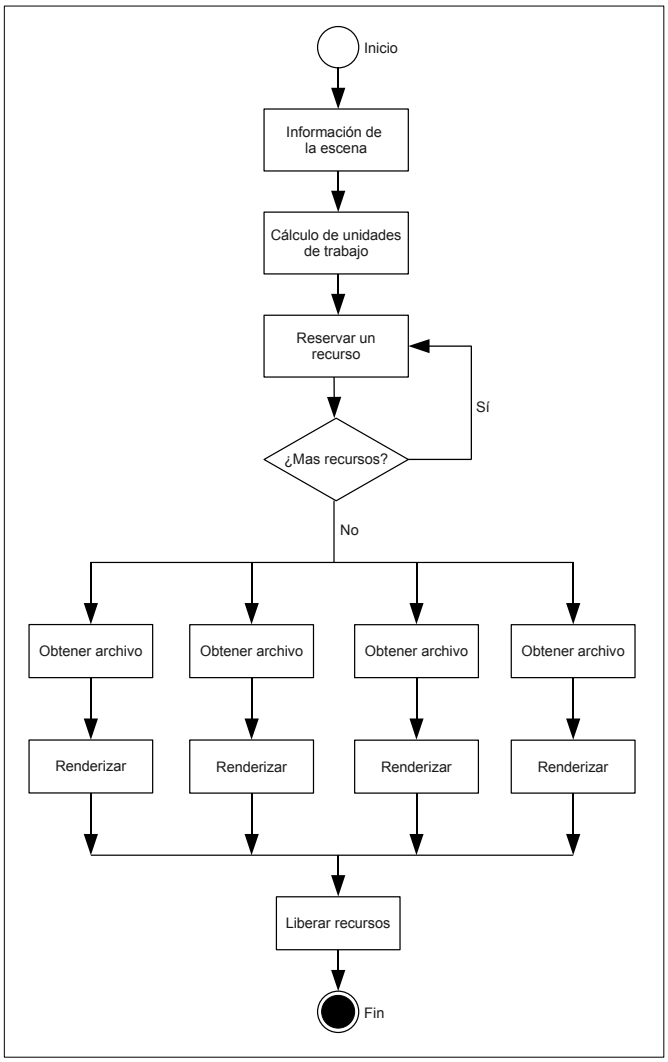

Figura 4.1: Diagrama de flujo: Renderizado de una escena

Para el desarrollo de Yafrid-NG se ha seguido una metodología iterativa. Se han identificado las partes fundamentales del sistema y se han ido implementado una sobre otra, obteniendo cada vez una aproximación más cercana a la solución buscada.

4.2. Tecnologías utilizadas

La implementación del sistema se ha realizado utilizando el lenguaje de programación C++ debido a su facil portabilidad entre distintas plataformas, eficiencia y rapidez. Como

consecuencia de que la API de Blender[1] está implementada en Python, será necesaria la programación de algunos scripts en este lenguaje para realizar el análisis y el renderizado de la escena.

Para el tratamiento de imágenes se ha escogido la librería ImageMagick[5] por tener una licencia compatible con la GPL, proporcionar un API suficientemente extensa para los requerimientos de este proyecto y dar soporte a múltiples lenguajes.

En cuanto al *middleware* utilizado para la parte de comunicaciones se ha escogido ZeroC ICE básicamente porque se trata de un middleware multiplataforma, fácil de utilizar y distribuido bajo una licencia GPL.

El entorno de desarrollo empleado es bajo el sistema operativo GNU/Linux, pero debido al uso de tecnologías multiplataforma, será posible la compilación para otros sistemas operativos sin demasiadas dificultades.

4.3. Arquitectura y funcionamiento del sistema

4.3.1. Arquitectura

El uso de un grid computacional implica la disponibilidad de una serie de computadores que proporcionan ciclos de CPU para la ejecución de tareas especialmente costosas. Una vez que tenemos los recursos disponibles, un cliente envía un trabajo al grid y obtiene los resultados como si se ejecutase de forma local. En Yafrid-NG se pueden distinguir dos roles principales, nodos o proveedores y clientes.

Los **nodos** o **proveedores** son los componentes del sistema que van a ceder sus ciclos de CPU para el renderizado de escenas, bajo la demanda de algún usuario.

Un **cliente** remitirá sus trabajos para que sean renderizados por el grid. Con el propósito de que el sistema sea tan descentralizado posible, es el propio cliente el que se va a encargar de realizar las labores de gestión para la correcta ejecución de la tarea. Estas labores implican:

- Coordinar a los nodos para la obtención de la escena que se va a renderizar. El cliente hará el papel de *broker* indicando a cada uno de los nodos dónde puede encontrar la porción de archivo demandada.

- Coordinar a los nodos a la hora de renderizar una escena. Es el propio cliente el que le indicará a cada proveedor la unidad de trabajo que debe realizar.

Para conseguir estos objetivos el cliente proporciona dos servicios llamados **P2PBroker** y **RenderManager**. El primero se utilizará para gestionar el paso de archivos y el segundo para gestionar la asignación de unidades de trabajo, evitando que alguno de ellos se ejecute más de una vez, si no es necesario.

Uno de los aspectos clave a la hora de planificar un grid de computadores es el de la gestión de los recursos. Es decir, cómo se realiza el descubrimiento, la reserva, la ejecución de la tarea y la liberación de los recursos. Para esto se utilizan dos servicios básicos, al que tienen que tener acceso todos los componentes del sistema. Estos servicios son el **gestor de sesiones** y el **gestor de nodos** de Yafrid-NG. La función del gestor de sesiones es la de gestionar las sesiones creadas por parte de los usuarios. Éstas se hacen necesarias para la correcta asignación y liberación de los recursos. La función del gestor de nodos es la de tener constancia de los recursos que forman parte del grid y permitir la agregación de nuevos recursos, de forma dinámica. Además también debe permitir la reserva de recursos y su posterior liberación. El gestor de sesiones y el gestor de nodos se utilizan de forma conjunta para proporcionar y gestionar el acceso a los recursos del grid.

El acceso al gestor de sesiones y el gestor de nodos estará protegido con *glacier2* (ver sección 3.2.6.3). Para obtener una serie de recursos encargados de la ejecución de la tarea, el usuario debe establecer una sesión con *glacier2*, proporcionando certificados válidos para su autenticación. En caso contrario el usuario no podrá hacer uso del grid.

En la figura 4.2 se presenta un esquema aproximado de la arquitectura de Yafrid-NG.

4.3.2. Flujo de trabajo general

En esta sección se explicará, a grandes rasgos, el funcionamiento del sistema. En las secciones posteriores se explicará cada uno de los pasos con un mayor nivel de detalle.

Para añadir recursos al grid, es necesario que los proveedores establezcan una sesión con Yafrid-NG y se registren en el servicio de gestión de nodos. Un proveedor debe proporcionar un proxy a un objeto de tipo factoría. Los objetos factoría implementan la siguiente interfaz:

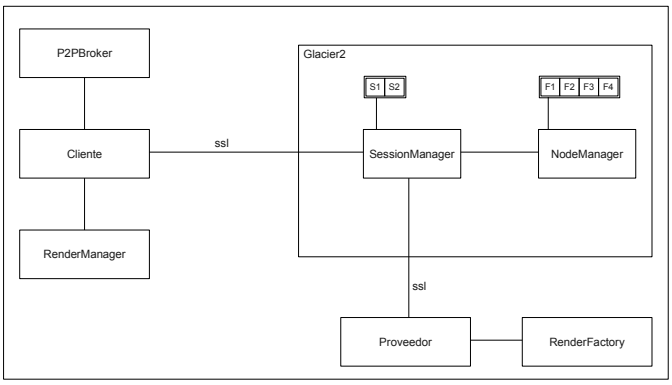

Figura 4.2: Arquitectura de Yafrid-NG

```
interface YafridngRendererFactory
{
  YafridngRenderer* createRenderer(int RenderEngine,
  string scene);
};
```

Sólo proporcionan una función, *createRenderer*. Ésta devuelve un proxy a un objeto del tipo *YafridngRenderer*. El propósito de la factoría es el de la inicialización del objeto. Los argumentos proporcionados se corresponden con un entero que representa el motor de render y una cadena que se corresponde con el nombre del archivo que contiene la descripción de la escena. Los objetos *YafridngRenderer* implementan la siguiente interfaz:

```
interface YafridngRenderer
{
  void obtainFile(Pieces piecesList, int piece, int fileSize,
  File* f, P2PBroker* broker);
  void render(TZones zones, YafridngRenderManager* manager)
  throws RenderingFailedException;
  void flush();
};
```

A continuación se describen las operaciones proporcionadas:

- **obtainFile**: Cuando se invoca la operación *obtainFile* comienza el proceso de intercambio de archivos (ver sección 4.8). Cuando termine la ejecución de la función el archivo se habrá transferido completamente. Los argumentos proporcionados son: una lista que

contiene la información de las piezas en las que se ha dividio el archivo, el número de pieza que se le ha asignado, el tamaño total del archivo, un proxy a un objeto de tipo *File* para la obtención del fragmento asignado y un proxy a un objeto de tipo *P2PBroker* que se utilizará para obtener información acerca de las piezas restantes.

Una pieza de archivo viene determinada por la siguiente información:

1. **offset**: indica la posición del archivo en la que comienza la pieza.
2. **pieceSize** indica el tamaño, en bytes, de la pieza.

- **render**: Cuando un nodo ha obtenido el archivo se invoca la función *render* para dar comienzo al trabajo. El nodo solicitará una unidad de trabajo al cliente y comenzará con su renderizado (ver sección 4.9). Los argumentos proporcionados son: una secuencia de zonas que indican los parámetros necesarios para la ejecución de los trabajos y un proxy a un objeto de tipo *RenderManager* que gestionará la ejecución de la tarea.

- **flush**: La operación *flush* se ejecuta cuando ha finalizado la ejecución de la tarea. Con esta operación se liberarán los recursos utilizados por el proveedor y se eliminará el objeto *YafridngRenderer* del adaptador de objetos.

Cuando un cliente quiere realizar un trabajo utilizando el grid, deberá establecer una sesión con el sistema y reservar algunos de los nodos disponibles. Un cliente se encarga de calcular las unidades de trabajo en las que se va a dividir la tarea. Después obtendrá una lista de recursos, invocará la función *obtainFile* sobre cada uno de ellos, después la función *render* para dar lugar a la ejecución de la tarea y finalmente la función *flush*. El cliente será el encargado de gestionar todo el proceso de paso de archivos y de renderizado.

4.4. Procesamiento de la escena

4.4.1. Obtención de información de la escena

Para llevar a cabo el renderizado de una escena es necesaria una previa recopilación acerca de algunos datos sobre la misma. En concreto, es necesario conocer si se trata de una

Valor	Motor de render
0	Blender
1	Yafray

Cuadro 4.1: Valores soportados para el motor de render en la versión actual del sistema.

animación o de un frame estático, la resolución de la escena y el motor de render utilizado. En función de estos parámetros el sistema actuará de distinto modo.

El usuario será el encargado de indicar si se trata de un frame estático o de una animación. Para el primero no hará falta ninguna información adicional, pero si se trata de una animación será necesaria la especificación, por parte del usuario, del frame inicial y del frame final, de esta forma el sistema será "consciente" de los frames que tiene que procesar.

Para obtener información acerca de la resolución de la escena y el motor de render utilizado se ha implementado un sencillo script implementado en Python.

Los resultados obtenidos por este script se almacenarán en un archivo llamado *scene.txt*. Su contenido será similar al siguiente:

```
0
720
576
```

El primero de los valores hace referencia al motor de renderizado(ver cuadro 4.1) y puede tomar dos valores diferentes en función del motor utilizado (Blender o Yafray).

Los siguientes valores indican la resolución de la escena. El primero de los valores hace referencia al ancho y el segundo al alto. Es necesario conocer estos valores para poder dividir la escena en unidades de trabajo más pequeñas.

Una vez que ha finalizado la ejecución del script, el programa cliente accederá al contenido del archivo *scene.txt* y obtendrá la información necesaria para el correcto cálculo de las unidades de trabajo.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import Blender
from Blender import *
from Blender.Scene import Render
import sys, os

scn = Scene.GetCurrent()
context = scn.getRenderingContext()

#Obtaining resolution
x = context.imageSizeX()
y = context.imageSizeY()

#Writing output file
fp = file('scene.txt', 'w')
fp.write(str(context.renderer) + '\n' + str(x) + '\n' + str(y))
fp.close()
```

Listado 4.1: Propiedades de una escena Blender por medio de Python

4.4.2. Calculando las unidades de trabajo

En Yafrid-NG se utilizan dos estrategias diferentes para calcular las unidades de trabajo. Se pueden distinguir dos tipos diferentes de granularidad (ver figura 4.3):

- **Granularidad fina**: El frame se divide en fragmentos más pequeños, cada uno de los cuales forma una unidad de trabajo. También se conoce como granularidad *intraframe*.
- **Granularidad Gruesa**: Las unidades de trabajo están compuestas por frames diferentes. También se conoce como granularidad *interframe*.

Una vez leída la información acerca de la escena hay que dividir el trabajo a realizar. Dependiendo del tipo de escena que se está manipulando, se utilizará una granularidad u otra para el cálculo de las *unidades de trabajo*. Las aproximaciones posibles son:

- **Frame estático**: Si se va a renderizar un frame estático se utilizará granularidad fina, de tal forma que cada uno de los nodos renderizará una o varias porciones de la escena.
- **Animación**: Si se va a renderizar una animación, se utilizará granularidad gruesa. El cálculo de unidades de trabajo se simplifica, ya que cada uno de los nodos renderi-

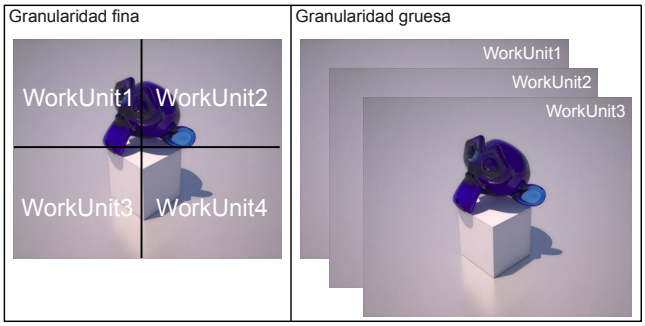

Figura 4.3: Diferentes granularidades

zará uno o varios frames. Es necesario conocer el número de frames para una correcta división del trabajo.

A la hora de calcular las tareas en el caso de un frame estático hay que tener en cuenta la naturaleza aleatoria de algunos de los métodos de renderizado. Debido a esta naturaleza aleatoria se puede percibir cierta diferencia entre las unidades de trabajo una vez compuesto el resultado. Para evitar que al componer la escena se perciban estos errores de muestreo se utilizará una **zona de interpolación** (ver figura 4.4). La zona de interpolación es una porción de imagen común entre dos unidades de trabajo. La porción de la imagen comprendida entre la banda de interpolación se renderizará dos veces, una por cada unidad. A la hora de componer la imagen estas zonas comunes se superpondrán una sobre la otra haciendo imperceptible el cambio entre una unidad de trabajo y sus vecinas. La composición de la imagen se explicará más adelante en este mismo capítulo (ver sección 4.11).

Para almacenar la información acerca de las unidades de trabajo se utilizará una estructura que consta de los siguientes campos:

- **id**: identificador de la zona.
- **x1**: coordenada x de la esquina superior izquierda de la unidad de trabajo.
- **y1**: coordenada y de la esquina superior izquierda de la unidad de trabajo.
- **x2**: coordenada x de la esquina inferior derecha de la unidad de trabajo.

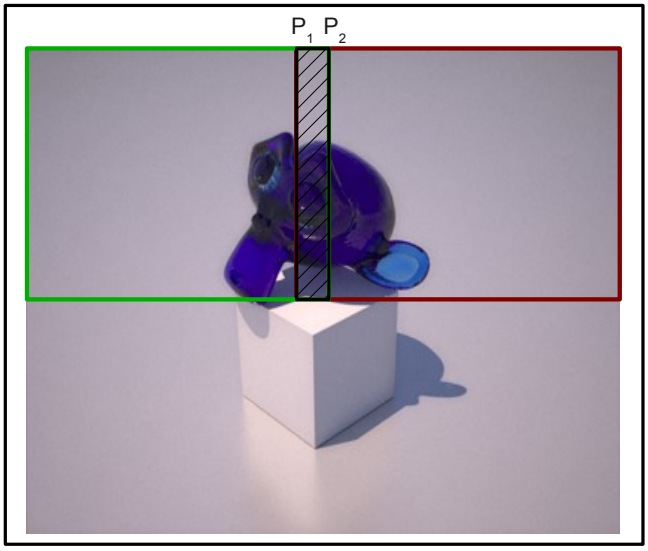

Figura 4.4: Banda de interpolación entre dos zonas

- **y2**: coordenada y de la esquina inferior derecha de la unidad de trabajo.
- **frame**: número de frame que le corresponde a la unidad de trabajo.

Obviamente, si la tarea a realizar consiste en un frame estático, el campo frame siempre tendrá el valor 1 y los valores de las variables *x1, y1, x2 e y2* irán cambiando. En el caso de una animación los valores de *x1 e y1* serán igual a 0 (esquina superior izquierda de la imagen), los valores *x2 e y2* serán el ancho y el alto de la resolución de la escena (esquina inferior derecha) y el campo *frame* almacenará el número de frame que se corresponde con la unidad de trabajo.

4.5. Gestión de Sesiones con Yafrid-NG

4.5.1. Problemática

Cuando un cliente desea realizar el renderizado de una escena con Yafrid-NG debe utilizar una serie de recursos proporcionados por el sistema. Existen varios problemas inherentes al utilizar esta estrategia:

1. ¿Qué ocurre si un cliente reserva todos los recursos de los que dispone el grid?

2. ¿Qué ocurre si un cliente, debido a un mal funcionamiento, no libera los recursos que ha reservado?

La respuesta a estas preguntas es que el resto de los usuarios no estarán capacitados a utilizar dichos recursos. Si alguno de los clientes reserva todos los recursos proporcionados, los demás no podrán realizar ninguna tarea mientras estos recursos no se liberen. Si alguno de los clientes no libera los recursos reservados previamente, éstos no volverán a estar disponibles para el resto y serán inservibles.

Para solucionar el problema se restringe el uso de los recursos proporcionados a través de una sesión[31]. Para poder utilizar el grid, un cliente deberá establecer una sesión con *Glacier2* y utilizar ésta para la adquisición de los recursos. La sesión creada limitará el numero de recursos que un cliente puede utilizar y también guardará constancia de los que se le han asignado, para que en caso de que no sean liberados, encargarse de hacerlo.

4.5.2. Sesión de renderizado

Cuando un usuario crea una sesión con *Glacier2* se le devolverá un proxy a un objeto del tipo *RenderSession*. La interfaz en *Slice* de una sesión de renderizado es la siguiente:

```
interface RenderSession extends Glacier2::Session
{
  void keepAlive();
  YafridngRenderer* create(int renderEngine, string scene)
  throws NoMoreNodesException;
  void addNode(YafridngRendererFactory* node);
};
```

Las operaciones implementadas por una sesión de renderizado son:

- **keepAlive**: El propósito de esta operación es el de mantener la sesión activa mientras se está ejecutando una tarea. Es posible que durante el renderizado de una escena la sesión esté inactiva durante una cantidad indeterminada de tiempo. Para evitar que se cierre prematuramente el cliente tiene que hacer llamadas, de forma periódica, a la función keepAlive.

- **create**: La operación *create* permite que un usuario reserve un nodo del grid para la ejecución de un renderizado. Cuando un recurso es reservado, ningún otro cliente lo podrá utilizar hasta que se libere. La operación *create* recibe como argumentos el motor de render (*renderEngine*) y el nombre del archivo a renderizar (*scene*), ambos valores son necesarios para inicializar el recurso utilizado. Esta función devuelve un proxy a un objeto de tipo *YafridngRenderer* cuyo funcionamiento se ha explicado en la sección 4.3.2. Los recursos serán liberados cuando se destruya la sesión. Si un cliente intenta reservar un recurso, pero se ha superado el límite de recursos reservados o ya no hay más disponibles en el grid se lanzará una excepción del tipo *NoMoreNodesException*.

- **addNode**: Con esta función se añadirá un nuevo recurso al grid. Éste podrá ser utilizado posteriormente por los clientes para renderizar sus escenas. Como argumento es necesario proporcionar un proxy a un objeto del tipo *YafridngRendererFactory* detallado en la sección 4.3.2.

Para limitar el número de recursos utilizados por un cliente se utiliza una variable de tipo entero. Cada vez que el cliente invoque la operación *create* esta variable incrementará su valor en uno. Cuando se alcance un máximo predefinido la operación *create* lanzará una excepción del tipo *NoMoreNodesException* indicando que no se pueden reservar más nodos.

El objeto *RenderSession* almacena todos los recursos utilizados por el cliente correspondiente en dos listas (ver figura 4.5). Una de ellas se utiliza para almacenar las factorías utilizadas. La otra se emplea para almacenar los *renderer* creados. Cuando la sesión termina, bien porque el cliente lo solicita o bien porque caduca, todos los recursos utilizados serán liberados automáticamente. Para esto es necesario implementar la operación *destroy*. Este metodo se invocará cuando se destruya la sesión, así nos aseguramos de que todos los recursos son liberados y están disponibles para su uso por parte de otros clientes.

4.5.3. Gestor de sesiones de Yafrid-NG

Para proporcionar la funcionalidad descrita anteriormente es necesaria la implementación de un gestor de sesiones. La misión de dicho gestor será la de crear una nueva instancia del objeto *RenderSession* y devolversela al cliente que creó la sesión.

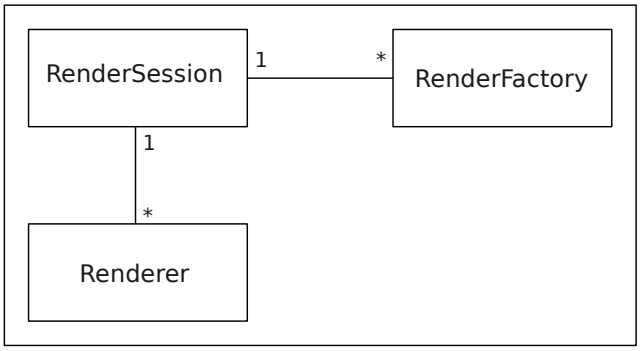

Figura 4.5: Diagrama de una sesión de renderizado

El gestor de sesiones mantendrá una lista de las sesiones existentes y cada vez que una nueva sea creada, comprobará que el resto de las sesiones existentes estén activas, en caso contrario eliminará las que sean necesarias.

Para la implementación del objeto *SessionManager* hay que crear una clase que herede de la clase *Glacier2::SessionManager*, si se utiliza autenticación mediante password, y la clase *Glacier2::SSLSessionManager*, si se utiliza autenticación mediante certificados. En Yafrid-NG se utiliza *Glacier2::SSLSessionManager* por lo que es necesaria la implementación de la siguiente interfaz:

```
module Glacier2 {
  exception CannotCreateSessionException {
    string reason;
  };

  interface SSLSessionManager {
    Session* create(SSLInfo info, SessionControl* control)
    throws CannotCreateSessionException;
  };
};
```

Cuando un cliente invoca la operación *createSession* sobre la interfaz del router, el router valida la identidad del usuario e invoca la operación *SessionManager::create*. La función *create* debe devolver un proxy a un nuevo objeto de sesión o lanzar una excepción de tipo *CannotCreateSessionException* (ver figura 4.6).

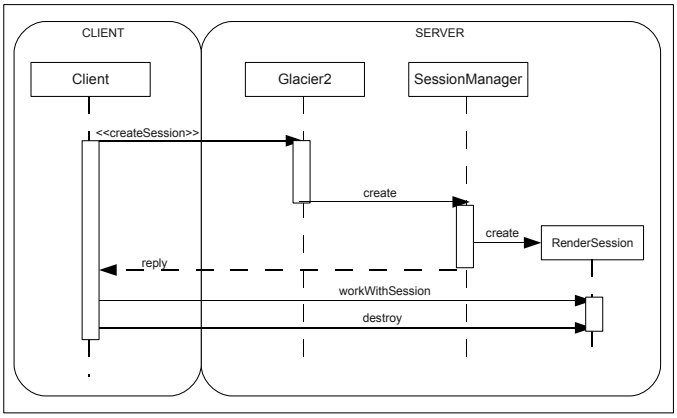

Figura 4.6: Diagrama de interacción: Usando una sesión de renderizado

Glacier2 invoca la operación *destroy* de una sesión cuando ésta expira. La función *destroy* está implementada en la clase *RenderSession*, de esta manera nos aseguramos de la liberación de los recursos.

4.5.4. Manteniendo la sesión activa

El uso de sesiones nos permite gestionar los recursos del grid de una forma adecuada. Para que esta técnica funcione correctamente es necesario establecer un tiempo de *timeout*. Este tiempo establece el número de segundos que una sesión puede estar inactiva hasta que expire.

Puede ocurrir que después de establecer una sesión, el cliente esté inactivo durante un periodo de tiempo y la sesión expire prematuramente imposibilitando, de esta forma, operaciones posteriores. Para evitar esta situación el cliente lanzará un hilo cuya única misión será la de realizar llamadas a la función *keepAlive* de forma periódica, para mantener la sesión activa. Es posible conocer el *timeout* establecido utilizando la función *getSessionTimeout* proporcionada por *Glacier2*. Conocido este tiempo podemos lanzar un hilo que "dormirá" un tiempo determinado y después invocará *keepAlive*, así indefinidamente hasta que se destruya dicho hilo (ver algoritmo 5).

Algorithm 5 Hilo que mantiene activa una sesión

```
while not destroyed do
    sleep(timeout)
    session→keepAlive()
end while
```

4.6. Gestión de Nodos

4.6.1. Problemática

A la hora de desplegar una sesión es necesaria la especificación de los recursos que forman parte del grid. Estos recursos permanecen fijos mientras el sistema se esté ejecutando.

Uno de los objetivos de Yafrid-NG es el de permitir la gestión dinámica de los recursos que constitutyen el grid, por lo que se hace necesario el registro de nuevos nodos en tiempo de ejecución. Para esto se ha implementado un servicio, llamado *NodeManager*, que permite la adición de nuevos recursos al grid. Además *NodeManager* también se encarga de reservalos, asegurando que si alguno de los nodos está siendo utilizado no será asignado a otro cliente, hasta que se libere.

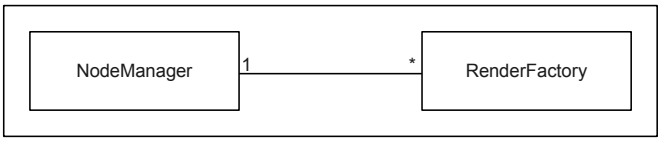

Figura 4.7: Diagrama del Servicio Gestor de Nodos

El *NodeManager* es un servicio interno al que sólo se tendrá acceso a través de una sesión de renderizado. El usuario realizará la petición de recursos utilizando la sesión y ésta acuará de intermediaria entre el cliente y el servicio *NodeManager*.

4.6.2. El servicio NodeManager

La interfaz en *Slice* del servicio *NodeManager* es la siguiente:

```
interface YafridngNodeManager
{
```

```
  void addNode(YafridngRendererFactory* factory)
  throws NodeAlreadyExistsException;
  YafridngRendererFactory* allocateNode()
  throws NoMoreNodesException;
  void releaseNode(YafridngRendererFactory* factory);
};
```

Las operaciones implementadas por el servicio gestor de nodos son las siguientes:

- **addNode**: Esta operación permite la agregación de nuevos nodos para su uso por parte de los clientes. Como argumento se proporciona un proxy a un objeto de tipo *YafridngRendererFactory*. La función *addNode* comprueba que no existe ningún recurso igual que el proporcionado como argumento. Si esto se cumple se añadirá un nuevo recurso al grid. Si ya existe el recurso se lanzará una excepción del tipo *NodeAlreadyExistsException* notificando de esta forma que el recurso ya existía previamente.

- **allocateNode**: Esta operación permite la reserva de un nodo. Cuando se reserva uno de los recursos, no estará disponible para ninguno de los otros clientes hasta que no se libere. Si no existe ningún nodo disponible se lanzará una excepción del tipo *NoMoreNodesException*.

- **releaseNode**: Con esta operación se libera un recurso previamente reservado. Como argumento se proporciona un proxy a un objeto de tipo *YafridngRendererFactory* que se localizará en la lista de nodos disponibles y se liberará, permitiendo su posterior uso por parte de otro cliente.

El servicio *NodeManager* almacena una lista con los recursos disponibles. Cuando se invoca la operación *addNode* se añade una nueva entrada a dicha lista. Esta operación debe ser invocada por un nodo. Las otras operaciones, *allocateNode* y *releaseNode* son invocadas por un cliente, a través de una sesión. Cuando se llama a la función *allocateNode* se devuelve una factoría y se marca como utilizada. La operación *releaseNode* libera el recurso indicado.

Combinando el servicio gestor de nodos con las sesiones de Yafrid-NG se ha conseguido proporcionar una gestión de recursos adecuada, uno de los aspectos a tener en cuenta a la hora de utilizar un grid de computadores para la realización de una tarea (ver figura 4.8).

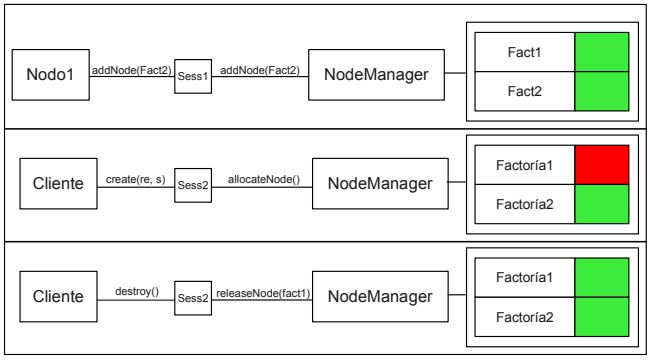

Figura 4.8: Operaciones proporcionadas por el gestor de nodos

4.7. Obtención de recursos para el renderizado

4.7.1. Protocolo de reserva de Nodos

Cuando un cliente desea realizar un trabajo en el grid, necesita reservar una serie de recursos para la ejecución de dicho trabajo. Estos recursos se proporcionarán a través de la sesión creada. Para reservar un recurso hay que invocar la función *allocateNode* proporcionada por la interfaz *RenderSession*.

Un cliente reservará nodos hasta que obtenga tantos como partes en las que se ha dividido la escena o hasta que reciba una excepción de tipo *NoMoreNodesException*. Esta excepción nos indica que ya no hay más nodos disponibles en el grid o que el cliente ha alcanzado el máximo de nodos permitidos para un mismo usuario.

4.7.2. El objeto Renderer y la cola de Render

El objeto *Renderer* es un objeto que almacena toda la información necesaria para la realización de un trabajo de render. Esta información consiste en:

- El nombre del archivo a renderizar.
- La lista de las unidades de trabajo.
- Un proxy a un recurso de los que se han reservado.

- Una pieza del archivo a renderizar.
- La información de todas las piezas en las que se ha dividido el archivo.
- El tamaño del archivo.
- Un proxy a un objeto de tipo *File* para el paso del archivo.
- Un proxy a un objeto de tipo *P2PBroker*.
- Un proxy a un objeto de tipo *YafridngRenderManager*.

Además el objeto *Renderer* implementa una operación llamada *run* cuyo propósito es el de invocar dos operaciones sobre el nodo asignado.

La primera de ellas es *obtainFile*. Al invocar esta función sobre un nodo comenzará la transferencia del archivo que contiene la descripción de la escena. Cuando termine la ejecución de esta función el nodo tendrá el archivo completo y se comenzará con la ejecución de las diferentes unidades de trabajo.

La segunda operación es *render*. Al invocar esta función sobre un nodo, éste comenzará a solicitar unidades de trabajo y a realizarlas utilizando los recursos locales.

Se creará un objeto de este tipo para cada uno de los recursos obtenidos.

Además se va a utilizar una cola en la que se van a almacenar los objetos de tipo *Renderer*. Para cada uno de estos objetos se lanza un hilo que ejecuta la función *run* de dicho objeto. Así se consigue que todos los recursos se utilicen de forma paralela. Los recursos obtendrán el archivo de y cuando lo tengan completo comenzarán con la tarea de renderizado (ver figura 4.9).

4.8. Intercambio de archivos

4.8.1. Problemática

A la hora de llevar a cabo un renderizado es necesario proporcionar los archivos requeridos a todos los nodos que van a ejecutar la tarea. Para esto se pueden aplicar varias estrategias (ver figura 4.10), dos de ellas son:

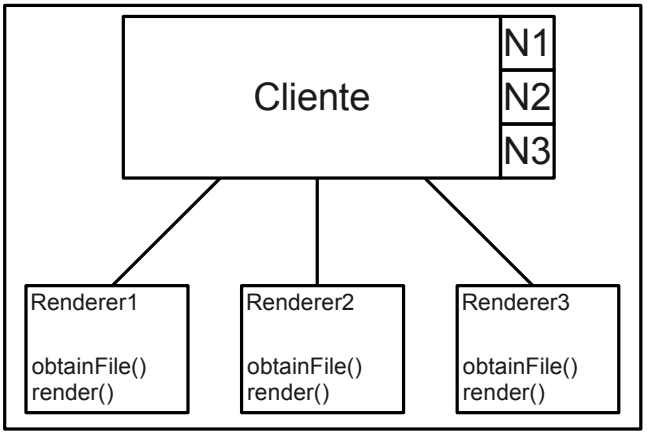

Figura 4.9: Hilos lanzados para tres nodos

- **Paso de archivos a través de un repositorio**: Con esta aproximación el cliente subirá el archivo a un repositorio. Cuando el archivo esté disponible, los nodos accederán a ese repositorio y obtendrán el archivo. El inconveniente de esta aproximación es que el repositorio tiene que servir el archivo a todos los *renderer* y esto se traduce en una sobrecarga en el servicio. Además el paso del archivo desde el cliente al repositorio requiere un tiempo que hay que añadir al tiempo total de renderizado.

- **Obtener el archivo del cliente**: Otra opción es que mediante algún mecanismo, los nodos accedan directamente al archivo en el cliente. De esta forma nos ahorramos el tiempo que supone pasar el archivo a un repositorio pero seguimos teniendo el problema de la sobrecarga. Si todos los nodos obtienen el archivo del cliente de forma simultánea, éste se sobrecargará.

Supongamos que tenemos un archivo de 20MB que contiene la descripción de la escena que se desea renderizar. El cliente transferirá el archivo a un repositorio intermedio y después los nodos lo obtendrán de ahí. Si se utilizan cinco nodos para el renderizado de la escena, el repositorio deberá transferir 100MB en total.

El método utilizado por Yafrid-NG para el paso de archivos pretende evitar esta situación haciendo que cada uno de los nodos que van a realizar la tarea transfiera una porción del archivo. El cliente dividirá el archivo en tantas partes como nodos participen en el render y

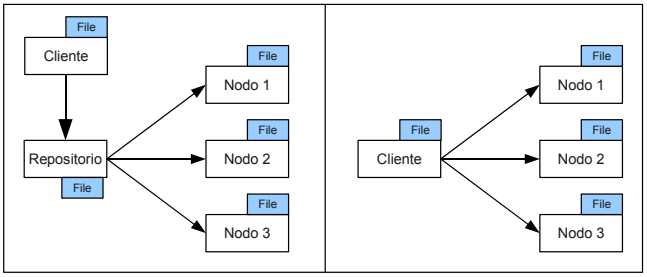

Figura 4.10: Estrategias para el paso de archivos

actuará como *broker* durante el intercambio del archivo. Los nodos le preguntarán sobre las partes del archivo que necesitan y el cliente les indicará donde las pueden obtener.

4.8.2. Paso de un archivo

Para el paso de archivos, Yafrid-NG utiliza una técnica que permite optimizar la transferencia de archivos[32] usando las carácterísticas del ICE. A continuación se describirá en detalle dicha técnica.

Típicamente, para el paso de un archivo se utiliza una interfaz similar a la siguiente:

```
sequence<byte> ByteSeq;
interface FileStore
{
  ByteSeq get(string name);
  void put(string name, ByteSeq bytes);
};
```

Utilizando esta interfaz el archivo se transmite de golpe, con una llamada a la función *get*. Aunque esta aproximación funciona en una red de área local, si el tamaño del archivo es considerable, se recomienda segmentarlo y transferirlo utilizando varias llamadas remotas en lugar de una.

Para la segmentación del archivo y su transferencia en fragmentos se propone la siguiente interfaz:

```
interface FileStore
{
```

```
  ByteSeq read(string name, int offset, int num);
  void write(string name, int offset, ByteSeq bytes);
};
```

La operación *read* solicita un numero de bytes (*num*) a partir de una posición determinada (*offset*). Cuando la operación *read* devuelva una secuencia de bytes vacía se habrá llegado al final del archivo. Esta operación supone un problema de rendimiento, ya que lo que con la anterior interfaz se realizaba mediante una llamada a la función *get* con la nueva interfaz propuesta se utilizan varias llamadas, por lo que se incrementa la latencia.

4.8.2.1. Cliente

A continuación se presenta el código mediante el cual un cliente leería un archivo del servidor:

Algorithm 6 Lectura de un archivo I

```
offset ← 0
len ← 1000*1024
name ← fileName
fp ← openFile(name)
for not finished do
    data ← prx→read(name, offset, len)
    if data.empty() then
        break
    end if
    write(data)
    offset←offset + data.size()
end for
close(fp)
```

Este algoritmo es bastante sencillo. El cliente simplemente leerá distintos fragmentos del archivo hasta que reciba una secuencia de bytes vacía, entonces se habrá llegado al final del archivo y finaliza la ejecución del bucle.

Esta implementación tiene dos problemas: la latencia de la red y el segundo y más importante es el tiempo empleado en escribir los datos al archivo, durante el cual cesa toda la actividad en la red. Lo ideal sería que el servidor leyese el siguiente fragmento de datos mientras se escribe el contenido en el archivo. Para conseguir este paralelismo es posible utilizar

las operaciones asíncronas en el lado del cliente (ver sección 3.2.6.1), de esta forma efectuaríamos la petición de un nuevo fragmento del archivo y mientras el servidor prepara los datos el cliente escribiría el fragmento leido anteriormente.

Para poder hacer uso de esta característica proporcionada por ICE es necesario especificarlo en la interfaz slice. La especificación de la operación *read* es la siguiente:

```
interface FileStore
{
  ["ami"] ByteSeq read(string name, int offset, int num);
};
```

Al añadir la directiva *["ami"]* en la interfaz se generará el siguiente código:

- Una clase abstracta, llamada *callback*, la cual será utilizada por el runtime de ICE para notificar a la aplicación el final de la operación. Esta clase consta de dos métodos:

  1. **void ice_response(params)**: Si se ejecuta esta función la operación ha terminado correctamente. Los parámetros se corresponden con el valor devuelto y los valores de salida.
  2. **void ice_exception(const Ice::Exception)**: Indica que ha ocurrido una excepción durante la ejecución de la operación.

- Un método llamado *nombre_de_la_operación_async*. El primer parámetro es una instancia de la clase *callback* definida anteriormente. Los siguientes parámetros son los parámetros de entrada de la operación.

Para poder utilizar la invocación asíncrona es necesario implementar las dos operaciones descritas anteriormente. Además se implementa una operación, perteneciente al *callback*, llamada *getData* que bloquea el hilo hasta que se reciben los datos del fragmento de archivo solicitado. Para un mayor detalle de estas implementaciones ver anexo B.2.

Una vez que se ha implementado el *callback* se puede leer el archivo utilizando el algoritmo 7.

Llegados a este punto, no se puede reducir el tiempo que se tarda en enviar la petición con *read_async* ni tampoco la cantidad de tiempo que se tarda en escribir en el disco. Pero

Algorithm 7 Lectura de un archivo II

```
offset ← 0
len ← 1000*1024
name ← fileName
cb ← new readCallBack
fp ← openFile(name)
prx→read_async(cb, name, offset, len)
for not finished do
    cb→getData(data)
    if data.empty() then
        break
    end if
    offset←offset + data.size()
    prx→read_async(cb, name, offset, len)
    write(data)
end for
close(fp)
```

sí se puede reducir el tiempo que se bloquea el hilo principal cuando se llama a la función *getData*. Para esto, se pueden reducir los efectos de la latencia de la red teniendo más de una llamada asíncrona activa. Con la primera de ellas obtendremos el fragmento actual esperado y la segunda solicita la siguiente porción de archivo. Así el servidor estará más ocupado, ya que cuando devuelva el resultado de la primera invocación la segunda ya habrá llegado y estará esperando a ser despachada.

El nuevo algoritmo para la recuperación del archivo se puede consultar en 8.

Ésta es la última versión del algoritmo, en el lado del cliente, para recuperar un fichero. Todavía se puede optimizar más pero esta optimización ya no tiene nada que ver con el algoritmo del cliente sino con la implementación de la función *ice_response()* (ver listado 4.2).

Como se puede observar en el listado, la información devuelta por la función *read* se copia desde el buffer *bytes* a *_bytes*. Además el *runtime* de ICE copia los datos del buffer de *unmarshalling* a *bytes*. Podemos evitar la copia desde el buffer de *unmarshalling* al vector *bytes*. Para ello es necesario añadir un metadato a la interfaz Slice. Por tanto la operación *read* quedaría de la siguiente forma:

```
interface FileStore
{
```

Algorithm 8 Lectura de un archivo III

```
offset ← 0
finished ← false
len ← 1000*1024
name ← fileName
readCallBack curr, next
fp ← openFile(name)
for  do
    if not curr then
        curr ← new readCallBack
        next ← new readCallBack
        prx →read_async(curr, name, offset, len)
    else
        swap(curr, next)
    end if
    prx →read_async(next, name, offset + len, len)
    curr→getData(data)
    if data.empty() then
        break
    end if
    write(data)
    offset ← offset + data.size()
end for
close(fp)
```

```
void
File_nextI :: ice_response(const Ice::ByteSeq& bytes)
{
  Lock sync(*this);
  _bytes = bytes;
  _done = true;
  notify();
}
```

Listado 4.2: Implementación de la función ice_response I

```
void
File_nextI :: ice_response
(const ::std::pair<const ::Ice::Byte*, const ::Ice::Byte*>& bytes)
{
  Lock sync(*this);
  Ice::ByteSeq(bytes.first, bytes.second).swap(_bytes);
  _done = true;
  notify();
}
```

Listado 4.3: Implementación de la función ice_response II

```
  ["ami", "cpp:array"] ByteSeq read(string name, int offset,
  int num);
};
```

Con este metadato el *runtime* de ICE pasa un par de punteros como argumentos de la función que apuntan al inicio y al final de la secuencia de bytes respectivamente. Estos punteros hacen referencia directamente al buffer de *marshalling* por lo que evitamos la copia anteriormente citada. Hay que adaptar el código de la función *ice_response* (ver listado 4.3).

4.8.2.2. Servidor

La implementación de la operación *read* es bastante directa. El servidor tiene que abrir el archivo solicitado, colocar el puntero en la posición indicada y leer el número de bytes solicitado. Si se ha llegado al final del archivo se devuelve una secuencia vacía (ver algoritmo 9).

Una de las desventajas de esta implementación es que en momentos puntuales requiere el doble de la memoria necesaria para la copia de los datos en buffer de *marshalling*. Para evitar la copia de estos datos se pueden usar llamadas asíncronas en el lado del servidor (*AMD*).

Algorithm 9 Operación read. Servidor

```
fp ← openFile(name)
if seek(fp, offset) == 0 then
    return emptyData
end if
read(bytes, num, fp)
close(fp)
return bytes
```

Para esto hay que añadir la directiva *["amd"]* en la interfaz de slice y cambiar la forma en la que se devuelven los resultados.

Para la lectura de los datos es necesaria la creación de un vector de bytes. Esto requiere su inicialización con ceros. Para evitar esta inicialización se puede utilizar la funcionalidad proporcionada por el metadato *["cpp:array"]* expuesto previamente. Ahora en lugar de pasar un vector al *callback AMD* se le pasan dos punteros, uno que apunta al principio de los datos y otro que apunta al final. Esto requiere unos mínimos cambios en la implementación de la función *read* (consultar código fuente). La interfaz en Slice de la función *read* queda de la siguiente forma:

```
interface FileStore
{
  ["ami", "amd","cpp:array"] ByteSeq read(string name,
  int offset, int num);
};
```

Todavía se pueden añadir algunas mejoras a la lectura de archivos. Cada vez que un cliente lee un fragmento del archivo el servidor tiene que:

- Abrir y cerrar el archivo.
- Buscar la posición adecuada en el archivo.
- Dedicar un buffer de lectura para cada petición.

Se puede evitar el tener que abrir y cerrar el archivo cada vez que se realiza una lectura. También se puede evitar el posicionamiento adecuado dentro del contenido del archivo, ya que se realizará una lectura secuencial. Para esto es necesario añadir una nueva interfaz que

se encargará de abrir el archivo cuando el cliente lo solicite y devolver un proxy a un objeto que nos va a permitir leer el contenido del archivo hasta que se alcance su final. Finalmente la interfaz en Slice queda de la siguiente forma:

```
interface File
{
  ["ami", "amd", "cpp:array"] Ice::ByteSeq next();
};

interface FileStore
{
  File* read(string name, int num)
  throws FileAccessException;
};
```

La operación *read* de la interfaz *FileStore* abre un archivo para su lectura y devuelve un proxy a un objeto de tipo *File* que es el que nos va a permitir leer el archivo. Cuando se alcance el final se devolverá una secuencia vacía, se cerrará el archivo y se eliminará el objeto *File* del adaptador de objetos.

4.8.2.3. Lectura de porciones del archivo

Actualmente se tienen dos interfaces que nos permiten leer un archivo de principio a fin de una forma optima. El objetivo de Yafrid-NG es el de proporcionar un sistema de intercambio de archivos en el que el cliente que ejecuta la tarea parta el archivo en un número de piezas más pequeñas, de tal forma que cada uno de los nodos que van a realizar el trabajo reciban una de ellas, haciendo así que entre todos los nodos tengan el archivo completo.

Una vez que cada nodo tiene su porción de archivo, obtendrá el resto a partir de los demás nodos o del cliente original, si no tiene otra opción. Para ello es necesario realizar un último cambio en la interfaz *FileStore*. Al utilizar la función *read* proporcionada por *FileStore* para la apertura de un archivo será necesario proporcionar como argumentos dónde comienza la porción de archivo que se quiere leer (*offset*) y cuantos bytes la forman (*PieceSize*). Quedando de la siguiente forma:

```
interface FileStore
{
```

```
  File* read(string name, int num, int offset, int pieceSize)
  throws FileAccessException;
};
```

Al invocar la función *read* se buscará la posición adecuada dentro del archivo utilizando *offset*. El argumento *pieceSize* se utiliza para saber cuando se tiene que parar de leer. La función *next* de la interfaz *File* comenzará a leer fragmentos a partir de *offset* y hasta que haya leído *pieceSize* bytes.

4.8.2.4. Partición y distribución del archivo

Para realizar el renderizado de una escena es necesario que todos los nodos tengan el archivo que contiene la descripción de la escena. A continuación se explicará la técnica utilizada para el paso del archivo a los nodos.

Para la distribución del archivo el cliente utilizará un objeto de tipo *P2PBroker*. La interfaz de Slice de este objeto es la siguiente:

```
interface P2PBroker{
  void notifyPiece(int piece, FileStore* fileSt);
  File* getPieceKeeper(int piece);
};
```

Este objeto proporciona dos operaciones. Estas operaciones son:

- **notifyPiece**: Esta invocación es utilizada por un nodo para indicar que ya ha obtenido alguna de las piezas. Como argumento se le pasa el número de pieza y un proxy a un objeto de tipo *FileStore* que se utilizará para que el fragmento de archivo pueda ser leído por otro nodo.

- **getPieceKeeper**: Esta función es utilizada por un nodo para saber dónde puede obtener una de las piezas que le faltan. El argumento indica el número de pieza solicitado. El valor devuelto se corresponde con un proxy de tipo *File* que permitirá que el nodo lea la pieza que solicita. Si el cliente todavía no tiene constancia de que ningún nodo posea esa pieza él será el encargado de proporcionar la porción de archivo solicitada.

Por tanto es el cliente que solicita la ejecución del trabajo el que se encarga de coordinar la distribución del archivo, eliminando así la dependencia de cualquier servicio.

Una vez que se han reservado los recursos para el renderizado el cliente tiene que transferir el archivo a dichos nodos. Lo primero que hace es calcular el tamaño de cada uno de los trozos que se van a distribuir. Teniendo presente el número de nodos con los que cuenta para la realización del trabajo y el tamaño del archivo, se calculará el tamaño de cada uno de los trozos y se repartirán por todos los nodos utilizando la operación *obtainFile* proporcionada por la interfaz *YafridngRenderer* (ver imagen 4.11).

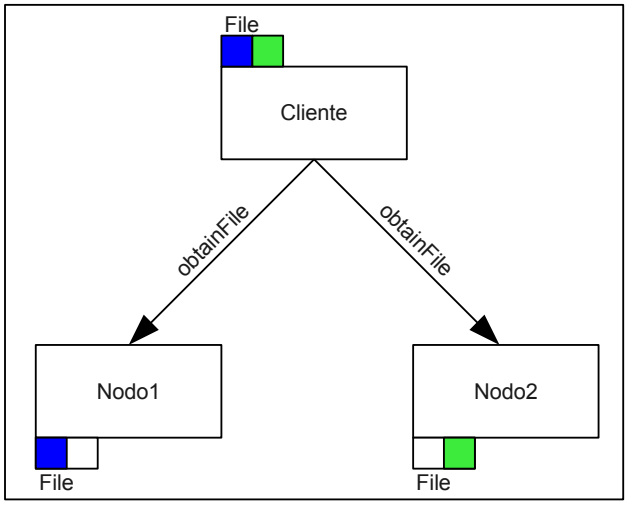

Figura 4.11: Paso de archivos I

Cuando un nodo haya terminado de escribir el fragmento que se le ha asignado, lo notificará a través de la operación *notifyPiece* proporcionando el número de pieza que ha obtenido y un proxy a un objeto *FileStore* que permitirá que el cliente obtenga proxies de tipo *File*, se los pase a otros nodos para que lean dicha pieza.

Después de notificar que la pieza asignada ya ha sido leída, el nodo empezará a solicitar el resto de piezas que tiene que conseguir para obtener el archivo entero (ver figura 4.12). Una vez que esto ocurra finalizará la ejecución de la función *obtainFile* y se podrá proceder con el renderizado de la escena.

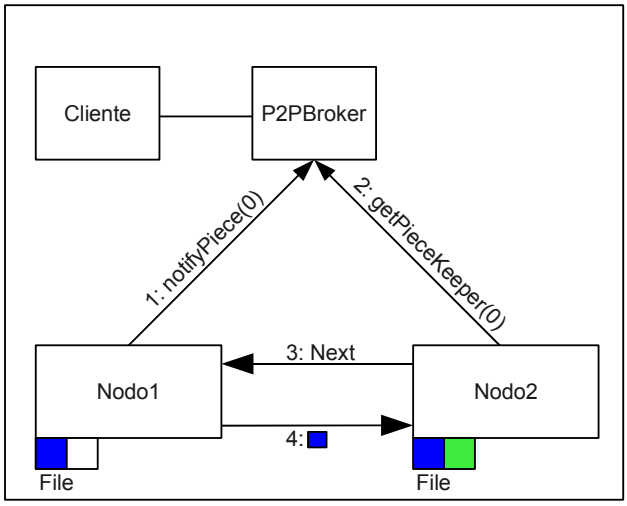

Figura 4.12: Paso de archivos II

4.8.3. La clase FileManager

El propósito de la clase *FileManager* es el de proporcionar cierto grado de abstracción a la hora de recuperar un fichero. Para crear una instancia de esta clase se hace necesaria la siguiente información:

- El nombre del archivo que se desea recuperar.
- Los flags que se utilizarán para abrir el archivo. El valor de los flags dependerá de si el archivo ya existe y se va a abrir para completar partes que faltan o si el archivo no existe y hay que crearlo.
- Un proxy de tipo *File* que se utilizará para recuperar el archivo.
- Un offset. Este argumento es opcional, si no se proporciona ningún offset se utilizará el valor 0. El offset indica la posición a partir de la cual se va a escribir en el archivo.

La clase *FileManager* proporciona la siguiente función:

- **getFile**: Una vez que se ha creado la instancia de la clase, esta función nos permite recuperar el archivo especificado con los parámetros de inicialización. Este archivo se

obtendrá del servidor adecuado y se escribirá en el disco duro local según los valores proporcionados.

4.9. El proceso de Renderizado

4.9.1. La operación render

Cuando termina la ejecución de la operación *obtainFile* el nodo ya posee el archivo necesario para la tarea de render. Inmediatamente después, el cliente invocará la operación *render* del nodo, pasándole la lista de zonas y un proxy al objeto *YafridngRenderManager* (ver subsección 4.9.2).

El proposito de la función *render* es el de solicitar y renderizar unidades de trabajo hasta que se complete el trabajo. El funcionamiento de la operación *render* se puede resumir en los siguientes pasos:

- Solicitud de una nueva zona a renderizar al objeto *YafridngRenderManager* proporcionado por el cliente.
- Preparar el comando de ejecución de render con los argumentos adecuados para la zona que se ha asignado.
- Ejecutar el renderizado.
- Indicar que la zona ya ha sido renderizada, para la recuperación y la composición de los resultados por parte del cliente.

Esta secuencia de pasos se ejecutará hasta que el cliente indique que la tarea ha llegado a su final. Cuando esto ocurra el cliente se encargará de invocar la función *flush* del objeto *YafridngRenderer* para liberar los recursos utilizados (ver figura 4.13).

Para renderizar una unidad de trabajo es necesario el uso de un script escrito en python. Este script renderizará una porción de escena según los parámetros proporcionados.

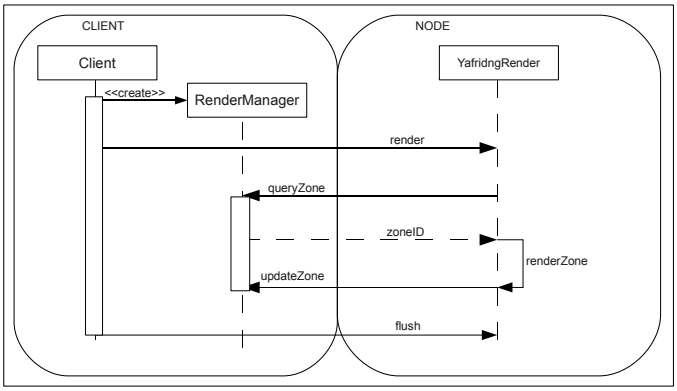

Figura 4.13: Diagrama de Interacción: Proceso de renderizado

4.9.2. El objeto RenderManager

Las unidades de trabajo en Yafrid-NG se asignarán bajo demanda por parte de los nodos. El objeto encargado de gestionar este proceso es el *RenderManager*, descrito a continuación.

El objeto *RenderManager*, instanciado por el cliente, se encargará de coordinar los nodos en la tarea de renderizado. Este objeto es el que va a asignar las unidades de trabajo a cada uno de los nodos, y al que se van a remitir los resultados. La interfaz proporcionada por *RenderManager* es:

```
interface YafridngRenderManager
{
  int queryZone(string nodeId);
  ["ami", "amd"] void updateZone(int zone, string fileName,
  File* f);
};
```

Este objeto proporciona dos operaciones. Estas operaciones son:

- **queryZone**: La operación *queryZone* devolverá el identificador de una unidad de trabajo que todavía no se ha asignado. Como argumento recibe el identificador del nodo que va a realizar el trabajo. El identificador se asociará a la zona asignada. El valor devuelto es un entero que indicará la unidad de trabajo que se tiene que realizar.

- **updateZone**: La operación *updateZone* será invocada por parte de los nodos para remitir los resultados obtenidos con la ejecución de una unidad de trabajo. Como argu-

mentos recibe un entero que se corresponde con el identificador de la unidad de trabajo realizada, una cadena que se corresponde con el nombre del archivo que contiene el resultado y un proxy a un objeto *File* que se utilizará para la recuperación de dicho archivo.

Para llevar el control de las zonas que se han renderizado, las que se están renderizando y las que quedan pendientes se guardará una lista en la que se indica el estado en el que se encuentra cada unidad de trabajo. Existen tres estados posibles para un trabajo, estos estados son:

- **NOT RENDERED**: Es el estado correspondiente a una unidad de trabajo que todavía no ha sido ni asignada, ni realizada. En el momento inicial todas las unidades tendrán este estado.
- **RENDERING**: Es el estado en el que se encuentra una unidad de trabajo que ha sido asignada pero cuya ejecución todavía no ha finalizado.
- **RENDERED**: Es el estado correspondiente a una unidad de trabajo que ya ha sido realizada.

Cuando un nodo realice la petición de una unidad de trabajo mediante la función *queryZone* se le asignará la primera que se encuentre en estado *NOT RENDERED*. El estado de la unidad asignada se cambiará a *RENDERING* y no se volverá a asignar, a no ser que ocurra algún error en el renderizado de dicha unidad.

Cuando un nodo indique que ya ha terminado con la unidad asignada mediante la operación *updateZone* se localizará la unidad de trabajo correspondiente y se le asignará el estado *RENDERED* (ver figura 4.14).

4.9.3. Gestión de errores

Es posible que alguno de los nodos se desconecte mientras está renderizando alguna unidad de trabajo. Si esto ocurre, dicha tarea permanecería indefinidamente en el estado *RENDERING* y nunca se concluiría el trabajo. Para evitar esta situación Yafrid-NG incluye un

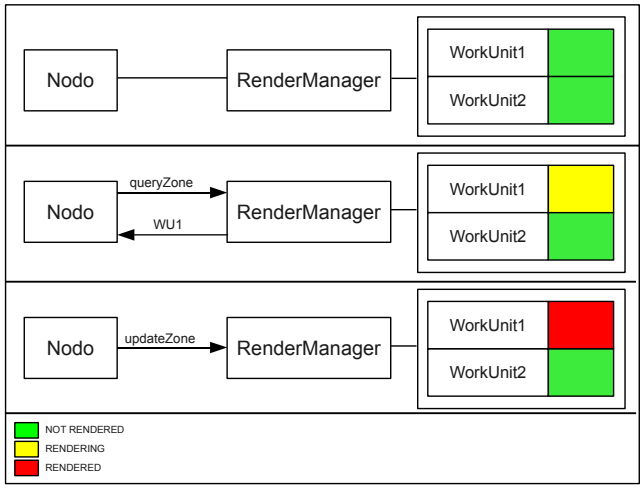

Figura 4.14: Operaciones del objeto RenderManager

mecanismo que permite detectar si un nodo se ha desconectado mientras realizaba un trabajo y reasignar este trabajo a otro nodo funcional.

Además de las operaciones descritas en la sección 4.9.2, el objeto *RenderManager* proporciona otra función *pública* pero que no se puede invocar de forma remota, la operación *cancelRender*. Esta operación nos permite cancelar una zona que esta siendo renderizada por alguno de los nodos. Como argumento recibe el identificador del nodo que ha fallado. La función localizará la zona que está siendo renderizada por dicho nodo y cambiará su estado de *RENDERING* a *NOT RENDERED*. Así, la próxima vez que un nodo solicite la asignación de una tarea mediante *queryZone* esta unidad de trabajo le será asignada.

Para controlar si alguno de los recursos se ha desconectado del sistema mientras estaba realizando un trabajo Yafrid-NG lanza un hilo que, de forma periódica, hará un *ping* a cada uno de los nodos que ha reservado el cliente, utilizando la operación *ice_ping*. Si alguno de estos se ha desconectado, al invocar esta operación se lanzará una excepción, ya que dicho nodo no está disponible. Si se captura esta excepción se invocará la función *cancelZone*.

Para el correcto funcionamiento de esta técnica es necesario que mientras queden unidades de trabajo por realizar no se liberen los recursos. Por tanto, si ya no se pueden asignar más unidades de trabajo pero todavía se están realizando algunas, cada vez que un nodo solicite un trabajo se le devolverá un valor que indica que ya no hay mas unidades disponibles pero que

todavía se están ejecutando algunas. Este nodo *dormirá* durante un cierto tiempo y volverá a solicitar un trabajo. Cuando todos los trabajos hayan finalizado se podrán liberar los recursos.

4.10. Recuperación de resultados

4.10.1. La operación updateZone

El proceso de recuperación de resultados consiste en obtener el archivo con la imagen producida al realizar una zona de trabajo. Recuperar la imagen puede tomar un tiempo considerable debido a las limitaciones de la red. Para evitar que esto afecte al nodo, la operación *updateZone* es asíncrona. El nodo invocará la operación y seguirá con su trabajo sin necesidad de esperar a que esta se ejecute.

El número de peticiones que puede atender el objeto *RenderManager* depende del modelo de concurrencia utilizado. Debido a que la operación *updateZone* puede requerir cierto tiempo es posible que se reciban más peticiones que hilos soporta el servidor. Para evitar que algunas peticiones sean descartadas, esta operación se declara como asíncrona en el lado del servidor.

Cuando se invoca la operación *updateZone*, los argumentos proporcionados se almacenarán en una cola, para su posterior procesado por parte de un hilo independiente. Los resultados se obtendrán de forma secuencial, por orden de llegada.

4.10.2. El objeto Retrieve

Cuando un nodo invoque la operación *updateZone* se creará un objeto de tipo *Retrieve* con todos los datos necesarios para la recuperación del archivo resultado y la composición con la escena final. Estos objetos ser irán almacenando en una cola para su procesado posterior.

La clase *Retrieve* proporciona dos operaciones. Estas operaciones son:

- **perform**: Al invocar la operación *perform* se llevarán a cabo las acciones necesarias para la obtención de los resultados y la composición del resultado final. Si se trata de un frame estático la imagen obtenida se compondrá con el la imagen local. Si se trata de una animación se recuperará el archivo correspondiente al frame y se colocará en el directorio correspondiente.

- **cancel**: La operación cancel simplemente lanzará una excepción y cancelará el trabajo.

4.10.3. La cola ResultManager

El objeto *ResultManager* implementa una cola en la que se van a ir almacenando los objetos de tipo *Retrieve*. La clase *ResultManager* proporciona varias operaciones que nos permiten gestionar la cola. Estas operaciones son:

- **add**: Permite añadir un objeto de tipo *Retrieve* a la cola. Los argumentos proporcionados son un callback para reportar los resultados, la resolución de la imagen final, la zona con la que se corresponde el resultado, el nombre del archivo y un proxy a un objeto de tipo *FileManager* para la obtención del archivo resultado.

- **perform**: La operación *perform* ejecutará el primero de los elementos de la cola. Una vez ejecutado lo eliminará. Esta operación no recibe ningún argumento. La operación *perform* devuelve un valor booleano que indica si se ha terminado de procesar todas las tareas.

- **lastZone**: La operación *lastZone* se utiliza para notificar que ya no se van a agregar más zonas a la cola. Esto nos permite finalizar con el hilo que se encarga de ejecutar los trabajos que se encuentran en la cola. Esta operación no recibe ningún argumento.

- **destroy**: La operación *destroy* cancela todos los trabajos que se encuentran en la cola. Esta operación no recibe ningún argumento.

- **getDestroy**: La operación *getDestroy* permite la consulta del estado de la cola. Se devuelve un valor booleano.

El programa cliente lanzará un hilo cuyo objetivo será el de invocar la operación *perform* mientras no se hayan procesado todos los resultados (ver figura 4.15). El hilo finalizará cuando se hayan recibido todos los resultados esperados o cuando ninguno de los nodos reservados esté funcionando.

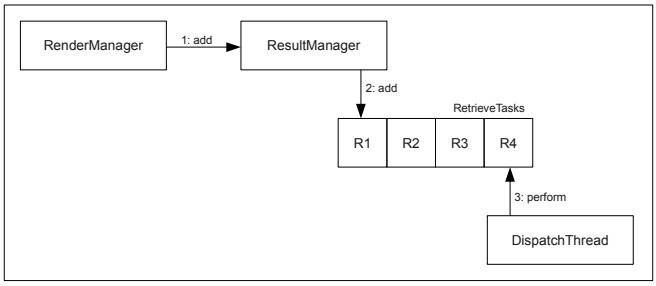

Figura 4.15: Recuperación de resultados

4.11. Composición de la Imagen

Si el trabajo consiste en un frame estático es necesaria la composición de las unidades de trabajo. Para esto es necesario conocer la resolución del resultado final y las coordenadas que se corresponden con la unidad de trabajo dentro del resultado.

Lo primero que hay que hacer, una vez recuperado el archivo, es generar la máscara que se va a utilizar. Para ello se crea una imagen consistente en un recuadro negro sobre otro blanco, utilizando las coordenadas que se corresponden con la unidad de trabajo para que la composición sea la adecuada. Después se aplica un filtro gaussiano que fusiona los bordes para poder calcular el nivel de transparencia de cada uno de los píxeles de la imagen que vamos a componer (ver figura 4.16).

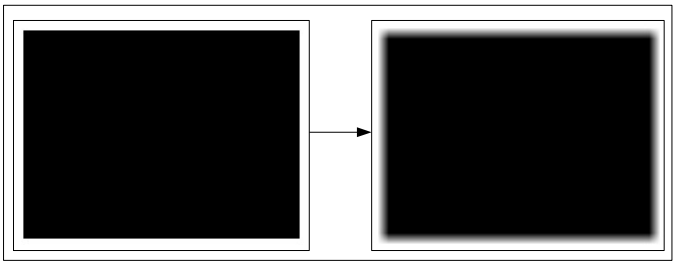

Figura 4.16: Máscara para el cálculo de la transparencia. Izquierda: Máscara calculada. Derecha: Aplicación de filtro Gaussiano

Después se calcula el grado de transparencia de cada uno de los píxeles utilizando la máscara generada.

El siguiente paso consiste en aplicar la máscara a la imagen recuperada del nodo, para su posterior composición con la imagen resultado. Para ello cargamos el archivo obtenido y seleccionamos la sección que contiene la imagen util utilizando las funciones *chop* y *crop* proporcionadas por la libería ImageMagick [5] (ver figura 4.17).

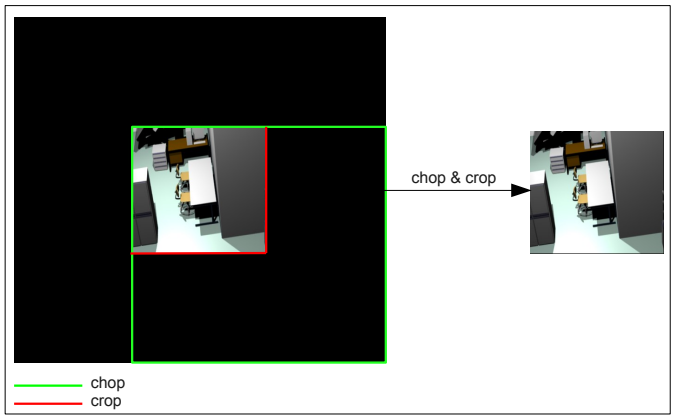

Figura 4.17: Uso de chop y crop sobre la imagen obtenida

Una vez que se han obtenido la máscara y la porción de imagen util, se componen ambas, copiando el nivel de transparencia de la máscara sobre la imagen, para después añadirla a la imagen resultado. Para esto basta con componer la imagen obtenida después de aplicar la máscara y el archivo de resultado.

4.12. Seguridad en Yafrid-NG

Es necesario aplicar una política de seguridad que nos permita autenticar a los clientes que hacen uso del grid y proteger el grid contra el acceso sin autorización. Un usuario se puede autenticar mediante un nombre de usuario y un password o proporcionando los certificados correspondientes. Además un cliente debe estar seguro de que se está comunicando con el servicio adecuado.

Para proteger a Yafrid-NG se hace uso conjunto del servicio de router proporcionado por ICE, *Glacier2* (ver sección 3.2.6.3, y el plugin *IceSSL*. Este plugin proporciona dos funcionalidades que son:

- **Encriptación**: Este aspecto protege las comunicaciones de la interceptación de los paquetes.

- **Autenticación**: Establece la identidad digital de una de las dos partes o de ambas.

En Yafrid-NG los servicios de gestión de sesiones y de gestión de nodos se ejecutarán detrás de *glacier2*. Para añadir un nuevo nodo al sistema, el nodo deberá autenticarse vía *Glacier2*, para establecer una sesión y poder proporcionar sus recursos. Para renderizar una escena, el cliente deberá autenticarse vía *Glacier2* para establecer una sesión y reservar los nodos necesarios para la ejecución del trabajo.

Para proporcionar esta configuración ha sido necesaria la creación de una autoridad de certificado (CA) utilizando la herramienta *iceca*, proporcionada por ICE. Se han generado certificados para el cliente y para los nodos. También ha sido necesario establecer las relaciones de confianza utilizando la propiedad de *IceSSL, TrustOnly*. De esta forma podemos establecer reglas de confianza de la siguiente forma:

```
IceSSL.TrustOnly=CN="Ice Server Glacier2 Router"
```

Indicando que solo se confíe en el servicio que proporcione unos certificados cuyo CN sea “Ice Server Glacier2 Router”.

Para generar certificados es necesario:

- Realizar una solicitud de certificado, proporcionando los datos requeridos por la utilidad de generación.

- Obtener el certificado firmado por la autoridad de certificado correspondiente.

Con *iceca* se pueden generar certificados utilizando el argumento *request*. Para firmar un certificado después de haberlo generado se utiliza el argumento *sign*. Se ha generado un certificado para *Glacier2*, un certificado para todos los proveedores y un certificado para el cliente.

Capítulo 5

Resultados obtenidos

5.1. Introducción

En este capítulo se realizará una comparativa entre los tiempos obtenidos al renderizar una escena de manera tradicional y los tiempos obtenidos utilizando Yafrid-NG. A la hora de realizar las pruebas utilizando Yafrid-NG se utilizarán distinto número de nodos y distinto número de unidades de trabajo. Los resultados relevantes son el tiempo transcurrido hasta que todos los nodos tienen el archivo y el tiempo total, incluyendo el tiempo transcurrido en el intercambio del archivo, el tiempo transcurrido durante el renderizado y el tiempo transcurrido durante la recuperación de los resultados y la composición de la imagen final.

Las secciones en las que se ha divido este capítulo son:

- **Renderizado tradicional**: Se comentarán los resultados obtenidos al renderizar una escena utilizando una máquina. También se hará referencia a las características de la máquina. Posteriormente se realizarán una serie de pruebas utilizando un número variable de máquinas con las mismas características.

Propiedad	Escena I	Escena II
Nombre de la escena	dragon_paraBlender	LC5_paraBlender
Tamaño del archivo	3,5 MB	17,3 MB
Motor de render	Blender	Blender
Resolución	640x480	800x600

Cuadro 5.1: Propiedades de la escena utilizada para las pruebas de Yafrid-NG

- **Renderizado con Yafrid-NG. Distinto número de nodos**: En esta sección se realizarán varias pruebas utilizando Yafrid-NG. El número de unidades de trabajo en todos los casos será de 9. El número de renderers será de 4, 9 y 16 respectivamente. Se presentará una tabla con los resultados obtenidos.

- **Renderizado con Yafrid-NG. Distinto número de unidades de trabajo**: En esta sección se realizarán varias pruebas utilizando Yafrid-NG. El número de unidades de procesamiento (renderers a partir de ahora) será de 16. El número de unidades de trabajo será variable, la escena se dividirá en 9, 16 y 25 unidades de trabajo. Se presentará una tabla con los resultados obtenidos.

- **Comparativa de los resultados**: En esta sección se presentarán una serie de gráficas realizadas a partir de los resultados obtenidos y se comentarán las conclusiones a las que han llevado las pruebas realizadas.

Las pruebas se han realizado con un total de 16 máquinas pertenecientes al laboratorio LD8 de la ESI, las características de estos computadores se puede consultar en la tabla 5.2, a pesar de que el número de *renderers* ha sido variables.

Además de las pruebas propuestas en esta sección también se han hecho experimentos cambiando el tamaño de la unidad de transferencia. Los valores testeados han sido 512 KB y 1MB. Los resultados obtenidos son muy similares para ambos casos, por lo que se han desestimado estas pruebas.

Las propiedades de las escenas utilizadas están descritas en el cuadro 5.1.

Figura 5.1: Escenas renderizadas con el motor de blender. Izquierda: dragon_paraBlender. Derecha: LC5_paraBlender

Propiedad	Valor
Procesador	Intel P4 2.66 GHz
Memoria principal	512 MB
Sistema operativo	Windows XP

Cuadro 5.2: Características de la máquina utilizada para el renderizado de las escenas

5.2. Renderizado tradicional

Para renderizar una escena se ha utilizado una máquina con las características que aparecen en la tabla 5.2.

Al renderizar la escena del dragón utilizando el motor de render proporcionado por blender se ha obtenido un tiempo de 16 minutos y 11 segundos.

Al renderizar la escena del laboratorio LC5 utilizando el motor de render proporcionado por blender se ha obtenido un tiempo de 12 minutos y 4 segundos.

A continuación se realizarán varios renders utilizando diferentes números de máquinas y de unidades de trabajo.

Renderers	Unidades de trabajo	Paso del archivo	Tiempo total
4	9	00:00:02	00:05:27
9	9	00:00:04	00:04:24
16	9	00:00:08	00:04:31

Cuadro 5.3: Resultados obtenidos con Yafrid-NG. Número de renderers variable. Escena Dragón.

Renderers	Unidades de trabajo	Paso del archivo	Tiempo total
4	9	00:00:12	00:04:01
9	9	00:00:20	00:03:07
16	9	00:00:37	00:03:23

Cuadro 5.4: Resultados obtenidos con Yafrid-NG. Número de renderers variable. Escena LC5.

5.3. Renderizado con Yafrid-NG. Distinto numero de renderers

Para estas pruebas las escenas se han dividido en un total de 9 unidades de trabajo. El número de máquinas utilizado renderizarlas ha sido de 4, 9 y 16. En las tablas 5.3 y 5.4 se muestran los resultados obtenidos.

5.4. Renderizado con Yafrid-NG. Distintas unidades de trabajo

Para estas pruebas se han utilizado un total de 16 máquinas para renderizar las escenas. El número de unidades de trabajo ha sido de 9, 16 y 25. En las tablas 5.5 y 5.6 se muestran los resultados obtenidos.

Renderers	Unidades de trabajo	Paso del archivo	Tiempo total
16	9	00:00:08	00:04:31
16	16	00:00:08	00:04:40
16	25	00:00:08	00:03:20

Cuadro 5.5: Resultados obtenidos con Yafrid-NG. Unidades de trabajo variables. Escena Dragón.

Renderers	Unidades de trabajo	Paso del archivo	Tiempo total
16	9	00:00:37	00:03:23
16	16	00:00:39	00:02:34
16	25	00:00:38	00:02:43

Cuadro 5.6: Resultados obtenidos con Yafrid-NG. Unidades de trabajo variables. Escena LC5.

5.5. Comparativa de los resultados

En esta sección se analizarán los resultados expuestos en las tablas anteriores, comentando los resultados obtenidos. También se explicará el por qué de los resultados obtenidos.

En la figura 5.2 se muestra la relación de los tiempos empleados para renderizar cada una de las escenas utilizando un número de unidades de trabajo fijas y un número de *renderers* variable. La primera conclusión que se puede obtener analizando los resultados es que conforme aumenta el número de renderers también aumenta el tiempo empleado en la obtención del archivo. Esto tiene sentido ya que cuantos más renderers haya más fragmentos se hará del archivo y más invocaciones remotas serán necesarias para su obtención. Es por esto que al aumentar el número de renderers aumenta el tiempo requerido para el paso del archivo. También señalar que al utilizar la estrategia del paso de archivos comentada en la sección 4.8 se reduce el tiempo empleado en la transferencia del archivo. Si se utilizase un servidor centralizado habría que transferir los datos a éste y después las unidades de procesamiento tendrían que obtenerlo del mismo. Con la nueva aproximación nos ahorramos la transferencia al servidor y liberamos al cliente del paso del archivo al total de los *renderers*.

En las gráficas también se puede comprobar que para un total de 9 unidades de trabajo el número optimo de renderers es 9. Si se utilizan 4 renderers cada uno renderizará más de una zona y serán necesarias varias llamadas remotas para la solicitud de las zonas, para este caso se ha obtenido el mayor tiempo en la ejecución de la tarea. Si se utilizan 16 renderers habrá 7 de ellos ociosos que también tendrán que ser gestionados, por lo que el tiempo también aumenta pero de una forma menos pronunciada.

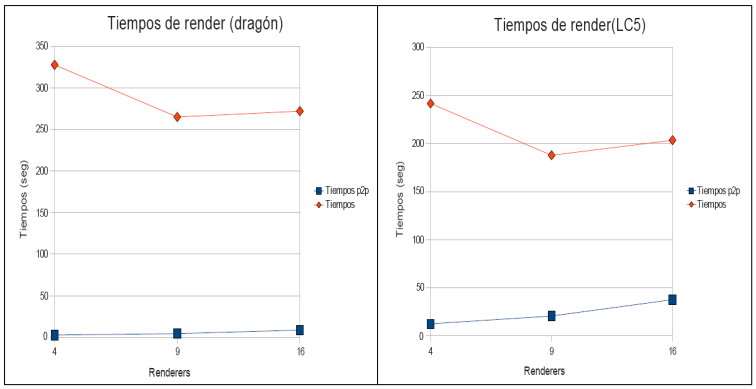

Figura 5.2: Gráfica de resultados. Numero de renderers variable

En la figura 5.3 se muestra la relación de los tiempos empleados para renderizar las escenas utilizando un número de unidades de trabajo variable y un número de *renderers* fijo. Para ambas escenas se puede observar que al conservar el número de renderers fijo el tiempo de paso del archivo permanece constante. Según las conclusiones obtenidas anteriormente, teniendo un total de 16 unidades de trabajo el tiempo óptimo debería obtenerse utilizando 16 renderers. Pero como se puede observar en la gráfica que se corresponde con la escena del dragón, se tarda menos tiempo en renderizar la escena dividiendola en 9 y 25 unidades de trabajo. Esto se debe a que al hacer la división de la escena en 16 fragmentos la parte más compleja requiere un mayor tiempo que cualquiera de las partes obtenidas al dividir la escena en 9 y 25 unidades. En concreto, la unidad de trabajo más compleja al dividir la escena en 9 unidades de trabajo se renderiza en 257 segundos, la parte más compleja al dividir la escena en 25 unidades de trabajo se renderiza en 185 segundos mientras que para renderizar la parte más compleja al dividir la escena en 16 unidades de trabaja se emplean 266 segundos.

En el caso de la escena del laboratorio LC5 el resultado es el esperado ya que la comple-

jidad de las unidades de trabajo es más homogénea.

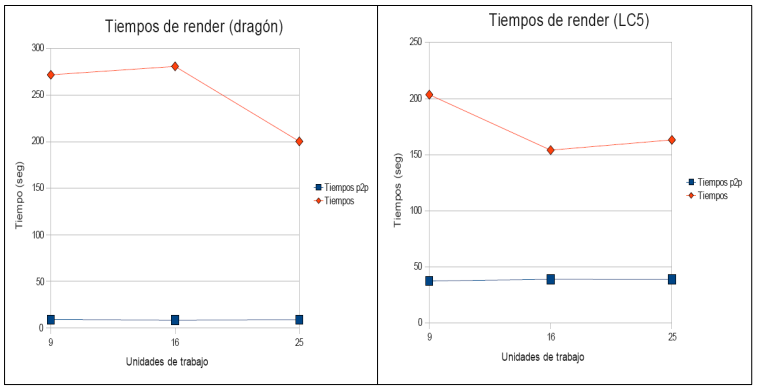

Figura 5.3: Gráfica de resultados. Numero de unidades de trabajo variable

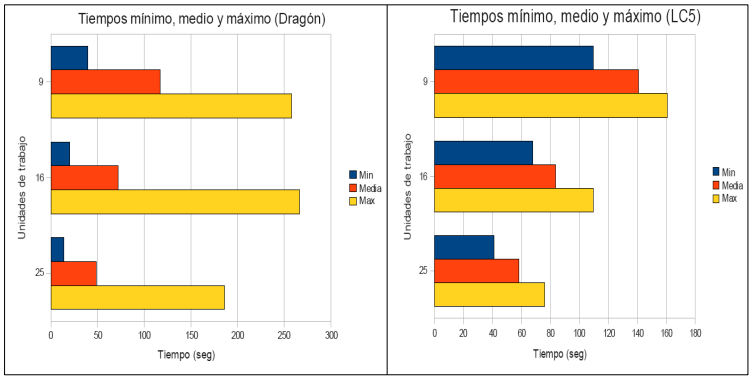

Figura 5.4: Gráfica de barras. Relaciones entre tiempos mínimos, medios y máximos

Para ilustrar mejor esta situación se ha estudiado el tiempo que se tarda en renderizar la unidad de trabajo más sencilla, la unidad de trabajo más compleja y la media de los tiempos empleados (ver figura 5.4). Como se puede observar en la gráfica, la diferencia entre el tiempo medio y el máximo es mucho más pronunciada en la escena del dragón que en la escena del laboratorio. Esto influye negativamente en el tiempo de renderizado, ya que la unidad más compleja retrasa la ejecución de la tarea.

Esto nos lleva a la conclusión de que el tiempo de renderizado no depende solamente del número de renderers utilizados y el número de unidades de trabajo sino también de la complejidad de estas unidades de trabajo. Por tanto resultaría conveniente realizar una partición

más adecuada de la escena para optimizar los tiempos empleados.

Por último señalar que para la realización un trabajo es necesario encontrar una relación adecuada entre unidades de trabajo y renderers utilizados. Por lo que la asignación de los nodos debería hacerse en función de la complejidad de la escena y las unidades de trabajo en las que se ha dividido. Las unidades más complejas deberían subdividirse para hacer una partición equitativa y nivelar el tiempo utilizado para renderizar cada unidad de trabajo.

Capítulo 6

Conclusiones y propuestas

6.1. Conclusiones

Con la elaboración de este trabajo se ha obtenido un sistema grid para el renderizado de escenas en 3D. Los objetivos alcanzados han sido:

1. Proporcionar una infraestructura para el renderizado distribuido utilizando un conjunto de máquinas independientes y heterogéneas.

2. Proporcionar un sistema lo más descentralizado posible. Los únicos servicios necesarios son el gestor de sesiones y el gestor de nodos. Una vez que el cliente ha reservado los recursos necesarios será el encargado de gestionar todo el proceso de renderizado, eliminando así la dependencia de otros servicios que no sean el gestor de sesiones y el gestor de nodos.

3. Proporcionar un sistema para el intercambio de archivos en el que cada uno de los nodos transferirá una porción del archivo, reduciendo la carga que supondría para el cliente esta tarea. El cliente, en lugar de enviar el archivo completo, dividirá el archivo en función del numero de nodos reservados para la realización del trabajo. Después de repartir los fragmentos del archivo será el encargado de coordinar a los nodos para la obtención de resto de fragmentos. Con esto se libera al cliente de parte de la carga que supone transferir el archivo completo a los distintos nodos.

4. Proporcionar una gestión adecuada de los recursos del grid, limitando el número de nodos que un cliente puede utilizar y liberando los recursos si se produce algún fallo en el cliente. Para conseguir este objetivo se ha hecho uso de las sesiones de renderizado. A través de estas sesiones un cliente puede reservar los recursos para la ejecución de los trabajos. Las sesiones también permiten la limitación de los recursos que puede reservar un cliente así como la liberación de dichos recursos si el cliente no los libera.

5. Proporcionar una infraestructura de seguridad para el acceso al grid. Tanto para la agregación de recursos como para la obtención de recursos del grid será necesario establecer una sesión con *Glacier2*. Para establecer dicha sesión habrá que proporcionar certificados válidos, en caso contrario *Glacier2* rechazará la petición de establecimiento de sesión.

6. Permitir la gestión dinámica de los recursos del grid. A priori no es posible conocer el número de recursos que van a constituir el grid. Para la agregación dinámica de recursos se proporciona el servicio *YafridngNodeManager* que almacenará una lista de los recursos disponibles. Esta lista será dinámica, es decir, podrá crecer y decrecer según se añadan o eliminen nuevos recursos al grid.

7. Permitir tanto el renderizado de frames estáticos como el renderizado de animaciones. Para alcanzar este objetivo se han definido dos granularidades distintas. La granularidad fina utilizada para los frames estáticos y la granularidad gruesa utilizada para las animaciones. De esta forma si la tarea consiste en un frame estático cada *renderer* renderizará un pequeño fragmento del total. Si la tarea consiste en una animación cada

renderer renderizará uno o varios frames de los que componen la animación final.

8. Proporcionar un sistema multiplataforma. Una vez desarrollado el sistema se ha compilado para los dos sistemas operativos principales del mercado (Windows y Linux). Debido a que se han utilizado librerías estandares no debe suponer demasidad dificultad compilar Yafrid-NG para otras arquitecturas u otros sistemas operativos.

9. Permitir la independencia del motor de render. Para utilizar otros motores de render sólo sería necesaria la implementación de un script de renderizado para el motor deseado y la modificación de una pequeña parte del código de proveedor de servicio.

El interfaz de acceso al grid para el usuario es muy sencillo. Un cliente sólo tiene que indicar el archivo que contiene la descripción de la escena, si se trata de un frame estático o de una animación y en el último de los casos, el frame inicial y el frame final. El usuario no será consciente de que se están utilizando varios computadores para el renderizado de la escena. Para él es como si el grid fuese un recurso local. No es necesario un conocimiento adicional para el uso de Yafrid-NG.

Con el uso de Yafrid-NG se reduce el tiempo invertido en el renderizado de una escena. El uso de un grid de computadores implica una serie de pasos que no son necesarios si se utiliza una sola máquina. Estos pasos incluyen el intercambio de archivos entre el cliente y los nodos, la solicitud de las unidades de trabajo por parte de los nodos y la recuperación de los resultados. Estas tareas incrementan el tiempo total de renderizado. Debido a las razones expuestas el uso de Yafrid-NG tiene sentido si la complejidad de la escena renderizada es mediana o grande. No merece la pena utilizar el grid para renderizar una escena sencilla, ya que probablemente se obtendría un tiempo mayor que al renderizarla de forma local.

El tiempo que requerido para la ejecución de un trabajo utilizando Yafrid-NG viene dado por la siguiente expresión:

$$T_{total} = T_{trans} + T_{renderDist} + T_0 \tag{6.1}$$

En la que T_{trans} es el tiempo requerido para el transporte del archivo, $T_{renderDist}$ es el tiempo requerido para renderizar la escena de forma distribuida y T_0 es el tiempo requerido para recuperar los resultados y componer la escena.

En una situación ideal en la que todas las unidades de trabajo fuesen de la misma complejidad y tuviesemos tantos nodos como unidades de trabajo el tiempo de renderizado se dividiría equitativamente entre el número de nodos utilizados, dando lugar a la ecuación 6.2.

$$T_{renderDist} = \frac{T_{renderLocal}}{N_{renderers}} \tag{6.2}$$

En la que $T_{renderLocal}$ es el tiempo empleado en renderizar la escena de forma local y $N_{renderers}$ es el número de recursos que van a realizar la tarea.

Por lo tanto, la desigualdad que se tiene que cumplir para que el uso del grid esté justificado es la siguiente:

$$T_{renderLocal} > T_{total} \tag{6.3}$$

Desarrolando esta inecuación llegamos a la siguiente conclusión:

$$T_{renderLocal} > \frac{(T_{trans} + T_0) \times N_{renderers}}{N_{renderers} - 1} \tag{6.4}$$

Además es necesario tener en cuenta el tiempo que se invierte en el renderizado de las bandas de interpolación, por lo que se añadirá un elemento χ a la desigualdad. Si el tiempo de render local es mucho mayor que el tiempo de renderizado, este factor se puede despreciar. Finalmente la desigualdad queda de la siguiente forma:

$$T_{renderLocal} > \frac{(T_{trans} + T_0) \times N_{renderers}}{N_{renderers} - 1} + \chi \tag{6.5}$$

Utilizando esta desigualdad se puede saber cuando es conveniente utilizar Yafrid-NG para el renderizado de una escena y cuando no.

6.2. Líneas de investigación futuras

A continuación se presentan algunas propuestas de mejoras en Yafrid-NG como el análisis previo de la escena, para obtener una orden adecuado para la ejecución de las tareas, la replicación de los servicios básicos o la implementación de un verificador de permisos que aumente el nivel de seguridad del grid.

6.2.1. Análisis previo de la escena

En la implementación actual de Yafrid-NG la escena se divide en partes iguales, sin tener en cuenta la complejidad de cada una de las unidades de trabajo. Al hacer este particionado sin un análisis previo es posible que las unidades más complejas se ejecuten al final, retrasando así la ejecución del trabajo. Suponiendo una escena dividida en cuatro unidades de trabajo y dos nodos que la renderizarán, puede ser que la unidad más compleja se asigne en último lugar, influyendo negativamente en el tiempo de render (ver figura 6.1).

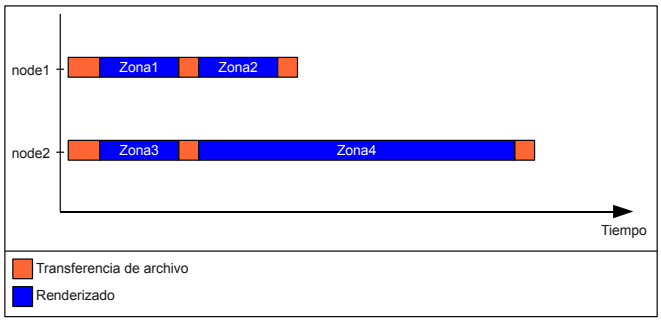

Figura 6.1: Gráfica de tiempos sin una ordenación previa

Si se realiza un analisis de la complejidad de cada una de las unidades de trabajo, es posible asignar las tareas más complejas en primer lugar, eliminando así el problema expuesto anteriormente (ver figura 6.2).

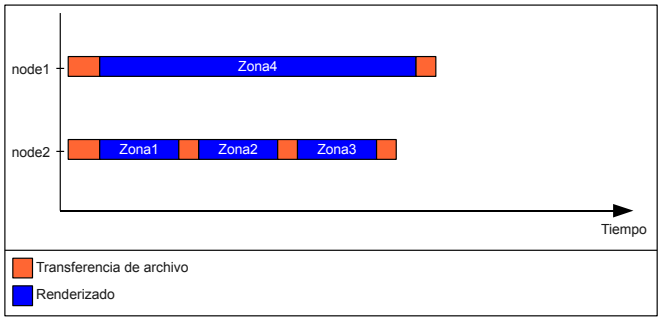

Figura 6.2: Gráfica de tiempos con ordenación previa

Además de la ordenación de las unidades de trabajo, el realizar un análisis de la escena

también permitiría dividir en unidades de trabajo más pequeñas las zonas más complejas, realizando así un particionado más equitativo de la escena (ver figura 6.3).

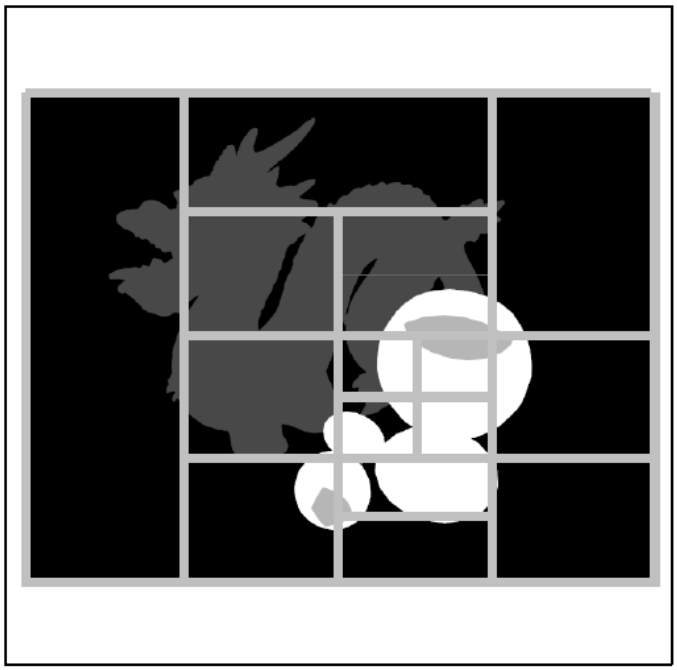

Figura 6.3: Particionado inteligente de la escena

Para llevar a cabo este particionado habrá que realizar un análisis previo para estimar la complejidad de la escena. Una forma de realizarlo es empleando los llamados "mapas de importancia"[21]. Estos mapas son una imagen en escala de grises, en la que las zonas más complejas tendrán colores cercanos al blanco mientras que las menos complejas tendrán colores cercanos al negro.

El primero de los pasos consistiría en hacer una partición inicial básica. Después, para cada zona se analizaría la complejidad de las zonas vecinas, fusionando las que tengan un nivel de complejidad parecida. Por último, para realizar una partición equitativa, se dividirían las zonas cuyo *ratio* complejidad-tamaño sea elevado en porciones mas reducidas.

Con esta mejora se obtendrían mejores tiempos de renderizado debido a que el tiempo requerido para la realización de los trabajos sería más o menos uniforme por lo que no existiría ninguna unidad de trabajo que retrasase la ejecución de la tarea de forma significativa.

6.2.2. Mejoras en el paso de archivos

En el mecanismo para el paso de archivos, cuando un nodo escribe el fragmento que se le ha asignado, lo notifica al cliente. Cuando otro nodo solicite alguna porción del archivo, el cliente comprobará si dicha porción ha sido notificada y proporcionará un proxy para la recuperación del fragmento, sin contemplar otros aspectos como pueden ser la sobrecarga de la red.

En la versión actual, cada nodo notifica al cliente cuándo ha terminado de escribir en disco local la porción del archivo asignada al inicio del proceso. De esta forma el cliente tendrá un candidato para cada una de los fragmentos que componen el archivo completo, por lo que no se podrá elegir entre una serie de candidatos para la obtención de un fragmento del archivo.

Una posible mejora es que los nodos notifiquen cada una de las piezas que van obteniendo. Así, cuando alguno de los nodos solicite un fragmento del archivo, el cliente podrá elegir, de una lista de nodos candidatos, el que potencialmente proporcionará el fragmento solicitado de una forma más rápida.

Para obtener dicho nodo bastaría con hacer una prueba previa a cada uno de los disponibles. Se podría lanzar un *ping* y medir el tiempo que tarda cada uno de los nodos en responder. El que menos tarde será el nodo elegido para el paso de la porción solicitada.

El coste para la realización de esta mejora no sería muy alto. Bastaría con hacer unas ligeras modificaciones en las operaciones del interfaz *P2PBroker* y en la operación *obtainFile* de la interfaz *YafridngRenderer*.

Realizando esta mejora se reducirían los tiempos empleados para el paso de archivos.

6.2.3. Mejoras al proceso de renderizado

El proceso de renderizado depende de una gran cantidad de parámetros. Los valores de estos parámetros pueden influir negativamente en el tiempo de renderizado sin proporcionar cambios en la calidad de la escena que el usuario pueda percibir.

Sería posible el uso de conocimiento experto en cada uno de los nodos que renderizan las unidades de trabajo. Con el uso de un conjunto de reglas difusas se podrían optimizar los valores de esos parámetros, reduciendo el tiempo de renderizado y obteniendo calidades

similares a la que se obtendría utilizando los parámetros iniciales.

Con esta mejora se reducirían aún más los tiempos de render. Ya que los parámetros de cada unidad de trabajo se modificarían con el objetivo de reducir el tiempo empleado para su ejecución sin modificar de forma perceptible la calidad final de la escena.

6.2.4. Generación de estadísticas

Para llevar un control del trabajo realizado por cada uno de los nodos, se podrían generar una serie de estadísticas para cada uno de los *renderer*. También se podría utilizar un sistema de créditos, similar al utilizado por BOINC[2], para establecer un ranking de usuarios en función del trabajo realizado.

La información se almacenarían en una base de datos y se podrían visualizar de forma esquemática vía web. Así cada uno de los usuarios podría consultar su grado de participación con respecto al de otros. Esto motivaría a los usuarios a proporcionar sus ciclos de CPU para el renderizado de escenas con Yafrid-NG.

Para llevar a cabo esta mejora se podría proporcionar acceso a Yafrid-NG mediante una interfaz web. Usando esta interfaz un usuario de Yafrid-NG se loguearía en el sistema obteniendo sus estadísticas y actualizándolas cada vez que se realizase un trabajo. Utilizando un lenguaje como PHP se podrían calcular las estadísticas para cada usuario y proporcionar una visualización adecuada en forma de gráficas, comparativas con las estadísticas de otros usuarios...

6.2.5. Replicación de los servicios

Con el propósito de proporcionar tolerancia a fallos sería deseable que los servicios de gestión de sesiones y gestión de nodos estuviesen replicados en varias máquinas. En este caso, exisitiría un servidor principal y varios secundarios. Los servidores secundarios mantienen una copia del estado del servidor principal. Así si el servidor principal falla, los clientes pueden acceder al servidor secundario para obtener los recursos necesarios para el trabajo.

Para proporcionar replicación se pueden utilizar las facilidades proporcionadas por *IceGrid*. Hay diferentes formas de proporcionar replicación de servicios utilizando ICE[20]:

- **Proxies directos**: Un proxy directo puede tener multiples *endpoints* y puede hacer referencia a múltiples réplicas del mismo servicio.

- **Proxies indirectos y grupos de replica**: Un proxy indirecto puede hacer referencia a un grupo de replica, por ejemplo *hello@MyReplicaGroup*. El localizador de ICE proporcionará los endpoints de una o más réplicas.

Como los servicios de Yafrid-NG necesitan guardar el estado, proporcionar replicación requiere de la implementación de algún mecanismo para que el estado de las diferentes réplicas sea consistente. Para esto sería necesario hacer uso de un patrón observador y una sesión entre cada uno de los servicios esclavos y el servicio master. Los servicios esclavos crean una sesión con el servicio master para recibir los cambios realizados en éste.

Cuando un servicio esclavo establezca una sesión con el maestro proporcionará un proxy a un objeto de tipo observador. Lo primero que se hace al establecer la sesión es sincronizar ambos servicios. Mediante una función el maestro notificaría su estado al esclavo. Después cada uno de los cambios que se realicen sobre el servicio maestro provocará una notificación a todos los esclavos que han establecido la sesión correspondiente. Las interfaces que habría que implementar sería similares a las siguientes:

```
interface NodeManagerObserver
{
  void init(YafridngRendererFactorySeq factories);
  void added(YafridngRendererFactory factory);
  void removed(string id);
}

interface NodeManagerSession
{
  void keepAlive();
  void destroy();
}

interface NodeManager extends Yafridng::NodeManager
{
  NodeManagerSession* createSession(NodeManagerObserver* obsvr);
}
```

Así los servicios esclavos implementarían la interfaz *NodeManagerObserver* para recibir

los cambios realizados en el maestro. El maestro invocará la función *init* cuando se establezca una sesión y las funciones *added* o *removed* según se añadan o se eliminen factorías. Las interfaces *NodeManagerSession* y *NodeManager* son implementadas por el maestro y utilizadas por los esclavos para la creación de sesiones. A la hora de crear una sesión el esclavo debe proporcionar un proxy a un objeto de tipo observador para la notificación de los cambios.

Otra característica deseable es que en lugar de existir un solo gestor de nodos y un solo gestor de sesiones para todos los usuarios del sistema, existieran varios, gestionando cada uno de ellos una parte del total de los usuarios que forman parte del grid.

6.2.6. Mejoras en la seguridad

Actualmente, Yafrid-NG proporciona acceso a los recursos del grid utilizando el protocolo ssl y proporcionando los certificados adecuados. Para el correcto funcionamiento de este sistema se hace necesaria la implementación de un verificador de permisos (*permissionsVerifier*). La implementación actual del verificador de permisos permite que cualquier cliente que proporcione un certificado válido acceda al grid.

Una posible mejora sería la implementación del verificador de permisos haciendo que, en lugar de comprobar el certificado proporcionado, utilice un servicio de consulta de directorio (como LDAP) para permitir que un cliente establezca una sesión.

Además, los clientes y los proveedores sólo utilizan ssl para las comunicaciones con *Glacier2*. Otra posible mejora para la seguridad sería el uso de ssl para las comunicaciones entre clientes y proveedores. Para esto sería necesaria la especificación de las reglas de confianza de forma cuidadosa, de tal manera que cada elemento del sistema sólo confíe en el resto de elementos que se tienen que comunicar con el y rechace cualquier otra conexión.

Apéndice A

Referencia de Yafrid-NG

A.1. Vista general

`module Yafridng`

El modulo Yafridng contiene la definicion de todas las estructuras, las excepciones y las interfaces utilizadas por Yafrid-NG.

A.1.1. Indice de interfaces

- **File**: La interfaz File permite la lectura de un archivo por parte de un cliente.
- **FileStore**: La interfaz FileStore define la operacion que nos permite abrir un archivo para su posterior lectura.
- **P2PBroker**: La interfaz P2PBroker define las operaciones proporcionadas por el gestor de intercambio de archivos.
- **RenderSession**: La interfaz RenderSession define las operaciones que se pueden ejecutar a traves de una sesion.
- **YafridngNodeManager**: La interfaz YafridngNodeManager define las operaciones necesarias para la gestion de los nodos que proporcionan sus ciclos de cpu.

- **YafridngRenderManager**: La interfaz YafridngRenderManager define las operaciones proporcionadas por el gestor de renderizado.
- **YafridngRenderer**: La interfaz YafridngRenderer define las operaciones proporcionadas por un nodo.
- **YafridngRendererFactory**: La interfaz YafridngRendererFactory se utilizara para crear e inicializar objetos del tipo YafridngRenderer.

A.1.2. Índice de excepciones

- **FileAccessException**: Excepcion que se lanza si hay algun problema al acceder a un archivo local.
- **NoMoreNodesException**: Excepcion que se lanza si ya no se pueden reservar mas nodos.
- **NodeAlreadyExistsException**: Excepcion que se lanza si se intenta añadir un nodo que ya existia previamente.
- **RenderingFailedException**: Excepcion que se lanza si hay algun problema al renderizar una unidad de trabajo.
- **YafridngException**: Excepcion basica de Yafridng.

A.1.3. Índice de estructuras

- **Piece**: La estructura Piece contiene los datos necesarios para especificar un fragmento de archivo.

  Está compuesta por los siguientes campos:

  - **offset**: El campo offset contiene la posicion inicial dentro del archivo.
  - **pieceSize**: El campo pieceSize contiene el tamaño del fragmento de archivo.

- **TZone**: La estructura TZone contiene los datos necesarios para especificar una unidad de trabajo.

  Está compuesta por los siguientes campos:

  - **id**: El campo id contiene el identificador de una zona.
  - **x1**: El campo x1 contiene la coordenada x de la esquina superior izquierda de la unidad de trabajo.
  - **y1**: El campo y1 contiene la coordenada y de la esquina superior izquierda de la unidad de trabajo.
  - **x2**: El campo x2 contiene la coordenada x de la esquina inferior derecha de la unidad de trabajo.
  - **y2**: El campo y2 contiene la coordenada y de la esquina inferior derecha de la unidad de trabajo.
  - **frame**: El campo frame contiene el numero de frame que se corresponde con la unidad de trabajo.

A.1.4. Indice de secuencias

- **Pieces**: Pieces es una secuencia de elementos de tipo Piece.
- **TZones**: TZones es una secuencia de elementos de tipo TZone.

A.2. Yafridng::File

A.2.1. Vista general

```
interface File
```

La interfaz File permite la lectura de un archivo por parte de un cliente.

A.2.2. Índice de operaciones

- **next**: La operacion next devuelve la siguiente secuencia esperada.

```
[ "ami", "amd", "cpp:array" ]
::Ice::ByteSeq next();
```

A.3. Yafridng::FileStore

A.3.1. Vista general

```
interface FileStore
```

La interfaz FileStore define la operacion que nos permite abrir un archivo para su posterior lectura. Para abrir un archivo es necesario utilizar la operacion read.

A.3.2. Índice de operaciones

- **read**: La operacion read abre un archivo en el servidor y nos devuelve un proxy File para su posterior lectura.

```
read
File* read(string name,
int num,
int offset,
int pieceSize)
throws
FileAccessException;
```

Parámetros proporcionados:

1. **name**: indica el nombre del archivo que se desea leer.
2. **num**: indica el numero de bytes que se transferiran con cada llamada a la funcion next.
3. **offset**: indica la posicion a partir de la cual se va a leer en el archivo.
4. **pieceSize**: indica el numero de bytes totales que se van a leer.

Valor devuelto: Un proxy a un objeto de tipo File.

Excepciones lanzadas:

- **FileAccessException**: si ocurre algun error a la hora de abrir el archivo.

A.4. Yafridng::P2PBroker

A.4.1. Vista general

```
interface P2PBroker
```

La interfaz P2PBroker define las operaciones proporcionadas por el gestor de intercambio de archivos. Sera utilizado por los nodos para notificar las piezas escritas y para solicitar las piezas restantes.

A.4.2. Índice de operaciones

- **notifyPiece**: La operación notifyPiece se utiliza por parte de los nodos para indicar que ha concluido con la escritura de una pieza de archivo en disco.

  ```
  notifyPiece
  void notifyPiece(int piece,
  FileStore* fileSt);
  ```

  Parámetros proporcionados:

  1. **piece**: indica el numero de la pieza que se ha escrito en disco.
  2. **FileStore**: es un proxy que permite la creacion de proxies de tipo File para la posterior lectura del archivo.

- **getPieceKeeper**: La operacion getFileKeeper se utiliza para obtener una porcion del archivo intercambiado. Un nodo solicita una pieza y el cliente devolvera un proxy a un objeto de tipo File mediante el cual se podra leer la pieza solicitada.

  ```
  File* getPieceKeeper(int piece);
  ```

Parámetros proporcionados:

1. **piece**: indica el numero de pieza solicitada.

Valor devuelto: Un proxy a un objeto de tipo File.

A.5. Yafridng::RenderSession

A.5.1. Vista general

```
interface RenderSession extends ::Glacier2::Session
```

La interfaz RenderSession define las operaciones que se pueden ejecutar a traves de una sesion. Es necesario establecer una sesion para hacer uso de Yafrid-NG. La sesion hara de intermediario entre los usuarios de Yafrid-NG y el sistema.

A.5.2. Índice de operaciones

- **keepAlive**: La operacion keepAlive se invocara de forma periodica para mantener la sesion activa y no destruirla prematuramente.

  ```
  void keepAlive();
  ```

- **create**: La funcion create se invocara, por parte del cliente, para obtener un recurso computacional.

  ```
  YafridngRenderer* create(int renderEngine,
  string scene)
  throws NoMoreNodesException;
  ```

  Parámetros proporcionados:

  1. **renderEngine**: indica el motor de render que se va a utilizar.
  2. **scene**: indica el nombre de la escena que se va a generar.

Valor devuelto: un proxy a un objeto de tipo YafridngRenderer que sera utilizado para la ejecucion de un trabajo.

Excepciones lanzadas:

- **NoMoreNodesException**: si el cliente ya ha reservado los recursos maximos permitidos.

- **addNode**: La operacion addNode permite la agregacion de un nuevo recurso computacional a Yafrid-NG.

```
void addNode(YafridngRendererFactory* node);
```

Parámetros proporcionados:

1. **node**: indica la factoria proporcionada por el nodo para la creacion de objetos de tipo YafridngRenderer.

A.6. Yafridng::YafridngNodeManager

A.6.1. Vista general

```
interface YafridngNodeManager
```

La interfaz YafridngNodeManager define las operaciones necesarias para la gestion de los nodos que proporcionan sus ciclos de cpu.

A.6.2. Índice de operaciones

- **addNode**: La operacion addNode permite la adicion de un nodo al sistema.

```
void addNode(YafridngRendererFactory* factory)
throws NodeAlreadyExistsException;
```

Parámetros proporcionados:

1. **factory**: es un proxy a un objeto de tipo YafridngRendererFactory. Existira un objeto de este tipo por cada nodo, permitiendo la posterior creacion de objetos de tipo YafridngRenderer.

Excepciones lanzadas:

- **NodeAlreadyExistsException**: si ya existe un nodo con la misma identidad en el sistema.

- **allocateNode**: La operacion allocateNode permite la reserva de uno de los nodos para la ejecucion de un trabajo.

```
YafridngRendererFactory* allocateNode()
throws NoMoreNodesException;
```

Valor devuelto: un proxy a un objeto de tipo YafridngRendererFactory que sera utilizado para la creacion de objetos de tipo YafridngRenderer.

Excepciones lanzadas:

- **NoMoreNodesException**: si ya no hay más recursos disponibles.

- **releaseNode**: La operacion releaseNode permite la liberacion de un recurso previamente reservado.

```
void releaseNode(YafridngRendererFactory* factory);
```

Parámetros proporcionados:

- **factory**: indica el recurso que se ha de liberar.

A.7. Yafridng::YafridngRenderManager

A.7.1. Vista general

```
interface YafridngRenderManager
```

La interfaz YafridngRenderManager define las operaciones proporcionadas por el gestor de renderizado. Su funcion sera la de asignar las unidades de trabajo pendientes de ejecucion y la recuperacion de los resultados.

A.7.2. Índice de operaciones

- **queryZone**: La operacion queryZone se utiliza para solicitar una unidad de trabajo por parte de un nodo.

  ```
  int queryZone(string nodeId);
  ```

  Parámetros proporcionados:

  1. **nodeId**: es el identificador del nodo que realiza la peticion.

  Valor devuelto: un entero que se corresponde con la unidad de trabajo asignada.

- **updateZone**: La operacion updateZone se utiliza para indicar la consecucion de una unidad de trabajo por parte de un nodo.

  ```
  [ "ami", "amd" ]
  void updateZone(int zone,
  string fileName,
  File* f);
  ```

  Parámetros proporcionados:

  1. **zone**: indica el numero de la unidad de trabajo que se ha ejecutado.
  2. **fileName**: indica el nombre del archivo que contiene el resultado.
  3. **f**: es un proxy a un objeto de tipo File para la recuperacion del resultado.

A.8. Yafridng::YafridngRenderer

A.8.1. Vista general

```
interface YafridngRenderer
```

La interfaz YafridngRenderer define las operaciones proporcionadas por un nodo. El cliente las utilizara para renderizar una escena.

A.8.2. Indice de operaciones

- **obtainFile**: La operacion obtainFile comienza el proceso de obtencion del archivo que contiene la escena. Cuando finalice su ejecucion el nodo tendra el archivo completo.

  ```
  void obtainFile(Pieces piecesList,
  int piece,
  int fileSize,
  File* f,
  P2PBroker* broker);
  ```

  Parámetros proporcionados:

  1. **piecesList**: es una secuencia de obtainFile. Contiene la informacion de todas las piezas en las que se ha dividido el archivo.
  2. **piece**: indica el numero de pieza asignada al nodo.
  3. **fileSize**: indica el tamaño total del archivo que contiene la descripcion de la escena.
  4. **f**: es un proxy a un objeto de tipo File que se utilizara para obtener la pieza asignada.
  5. **broker**: es un proxy a un objeto de tipo P2PBroker que se utilizara para la localizacion de las piezas restantes.

- **render**. La operacion render comienza el proceso de renderizado de la escena. El nodo ira solicitando la asignacion de unidades de trabajo hasta que el gestor de renderizadole indique que se ha renderizado la escena completa.

```
void render(TZones zones,
YafridngRenderManager* manager)
throws RenderingFailedException;
```

Parámetros proporcionados:

1. **zones**: es una secuencia de TZone. Contiene la informacion acerca de las unidades de trabajo.
2. **manager**: es un proxy a un objeto de tipo YafridngRenderManager. El nodo lo utilizara para obtener nuevas tareas y remitir los resultados de las ya realizadas.

- **flush**: La operacion flush libera los recursos utilizados por un renderer. Se invocara cuando finalice el renderizado de la escena.

A.9. Yafridng::YafridngRendererFactory

A.9.1. Vista general

```
interface YafridngRendererFactory
```

La interfaz YafridngRendererFactory se utilizara para crear e inicializar objetos del tipo YafridngRenderer.

A.9.2. Índice de operaciones

- **createRenderer**: La opearcion createRenderer crea un objeto de tipo YafridngRenderer y devuelve un proxy para su manipulacion.

```
YafridngRenderer* createRenderer(int RenderEngine,
string scene);
```

Parámetros proporcionados:

1. **RenderEngine**: indica el motor de render utilizado. Puede tomar dos valores, 0 para Blender y 1 para Yafray.

2. **scene**: contiene el nombre de la escena que se va a renderizar.

Valor devuelto: Un proxy al objeto creado. El cliente utilizara este proxy para la manipulacion del objeto.

Apéndice B

Código Fuente

Debido a la extensión general del código fuente, en el presente anexo se presentan los fragmentos más relevantes de código fuente de Yafrid-NG. En primer lugar se mostrará la interfaz en Slice y seguirán los algoritmos más destacables.

B.1. Yafridng.ice

```
#include <Ice/BuiltinSequences.ice>
#include <Glacier2/Session.ice>

/**
El modulo Yafridng contiene la definicion de todas las
estructuras, las excepciones y las interfaces utilizadas
por Yafrid-NG.
**/

module Yafridng{

  /**
  La estructura Piece contiene los datos necesarios
  para especificar un fragmento de archivo.
  **/
  struct Piece
  {
    /**
    El campo offset contiene la posicion inicial
    dentro del archivo.
    **/
```

```
  int offset;
  /**
  El campo pieceSize contiene el tamaño del
  fragmento de archivo.
  **/
  int pieceSize;
};

/**
La estructura TZone contiene los datos necesarios
para especificar una unidad de trabajo.
**/
struct TZone
{
  /**
  El campo id contiene el identificador de una zona.
  **/
  int id;
  /**
  El campo x1 contiene la coordenada x de la esquina
  superior izquierda de la unidad de trabajo.
  **/
  int x1;
  /**
  El campo y1 contiene la coordenada y de la esquina
  superior izquierda de la unidad de trabajo.
  **/
  int y1;
  /**
  El campo x2 contiene la coordenada x de la esquina
  inferior derecha de la unidad de trabajo.
  **/
  int x2;
  /**
  El campo y2 contiene la coordenada y de la esquina
  inferior derecha de la unidad de trabajo.
  **/
  int y2;
  /**
  El campo frame contiene el numero de frame que se
  corresponde con la unidad de trabajo.
  **/
  int frame;
};
```

```
/**
Pieces es una secuencia de elementos de tipo [Piece].
Se trata de una lista de estructuras de tipo [Piece].

@see Piece
**/
sequence<Piece> Pieces;
/**
TZones es una secuencia de elementos de tipo [TZone].
Se trata de una lista de estructuras de tipo [TZone].

@see TZone
**/
sequence<TZone> TZones;

/**
Excepcion basica de Yafridng. Todas las excepciones heredan
de YafridngException.
**/
exception YafridngException
{
  /**
  Cadena que contiene la razon por la que se ha lanzado
  la excepcion.
  **/
  string reason;
};

/**
Excepcion que se lanza si hay algun problema al renderizar
una unidad de trabajo.
**/
exception RenderingFailedException extends YafridngException
{
};

/**
Excepcion que se lanza si hay algun problema al acceder a
un archivo local.
**/
exception FileAccessException extends YafridngException
{
};

/**
```

```
Excepcion que se lanza si se intenta añadir un nodo que ya
existia previamente.
**/
exception NodeAlreadyExistsException extends YafridngException
{
};

/**
Excepcion que se lanza si ya no se pueden reservar
mas nodos. Bien porque se ha llegao al limite permitido
o bien porque ya no hay mas nodos disponibles.
**/
exception NoMoreNodesException extends YafridngException
{
};

/**
La interfaz File permite la lectura de un archivo por parte
de un cliente.
**/
interface File
{
  /**
  La operacion next devuelve la siguiente secuencia esperada.

  @return Una secuencia de bytes que se corresponde con los
  bytes leidos.

  @see Ice::ByteSeq
  **/
  ["ami", "amd", "cpp:array"] Ice::ByteSeq next();
};

/**
La interfaz FileStore define la operacion que nos permite
abrir un archivo para su posterior lectura. Para abrir
un archivo es necesario utilizar la operacion read.
**/
interface FileStore
{
  /**
  La operacion read abre un archivo en el servidor y nos
  devuelve un proxy File para su posterior lectura.

  @param name indica el nombre del archivo que se desea leer.
```

```
    @param num indica el numero de bytes que se transferiran
    con cada llamada a la funcion [next].
    @param offset indica la posicion a partir de la cual se
    va a leer en el archivo.
    @param pieceSize indica el numero de bytes totales que se
    van a leer.

    @throws [FileAccessException] si ocurre algun error a la
    hora de abrir el archivo.

    @return Un proxy a un objeto de tipo [File].

    @see FileAccessException
    @see File
    **/
    File* read(string name, int num, int offset, int pieceSize)
    throws FileAccessException;
};

/**
La interfaz P2PBroker define las operaciones proporcionadas
por el gestor de intercambio de archivos. Sera utilizado por
los nodos para notificar las piezas escritas y para solicitar
las piezas restantes.
**/
interface P2PBroker{
    /**
    La operación notifyPiece se utiliza por parte de los nodos
    para indicar que ha concluido con la escritura de una pieza
    de archivo en disco.

    @param piece indica el numero de la pieza que se ha escrito
    en disco.
    @param FileStore es un proxy que permite la creacion de
    proxies de tipo [File] para la posterior lectura del
    archivo.

    @see FileStore
    **/
    void notifyPiece(int piece, FileStore* fileSt);

    /**
    La operacion getFileKeeper se utiliza para obtener una
    porcion del archivo intercambiado. Un nodo solicita una
    pieza y el cliente devolvera un proxy a un objeto de tipo
```

```
    [File] mediante el cual se podra leer la pieza solicitada.

    @param piece indica el numero de pieza solicitada.

    @see File
    **/
    File* getPieceKeeper(int piece);
};

/**
La interfaz YafridngRenderManager define las operaciones
proporcionadas por el gestor de renderizado. Su funcion
sera la de asignar las unidades de trabajo pendientes de
ejecucion y la recuperacion de los resultados.
**/
interface YafridngRenderManager
{
    /**
    La operacion queryZone se utiliza para solicitar una
    unidad de trabajo por parte de un nodo.

    @param nodeId es el identificador del nodo que
    realiza la peticion.

    @return un entero que se corresponde con la unidad de
    trabajo asignada.
    **/
    int queryZone(string nodeId);

    /**
    La operacion updateZone se utiliza para indicar la
    consecucion de una unidad de trabajo por parte de un
    nodo.

    @param zone indica el numero de la unidad de trabajo
    que se ha ejecutado.
    @param fileName indica el nombre del archivo que contiene
    el resultado.
    @param f es un proxy a un objeto de tipo [File] para la
    recuperacion del resultado.

    @see File
    **/
    ["ami", "amd"] void updateZone(int zone, string fileName,
    File* f);
```

```
};

/**
La interfaz YafridngRenderer define las operaciones
proporcionadas por un nodo. El cliente las utilizara
para renderizar una escena.
**/
interface YafridngRenderer
{
  /**
  La operacion obtainFile comienza el proceso de obtencion
  del archivo que contiene la escena. Cuando finalice su
  ejecucion el nodo tendra el archivo completo.

  @param piecesList es una secuencia de [Piece]. Contiene
  la informacion de todas las piezas en las que se ha
  dividido el archivo.
  @param piece indica el numero de pieza asignada al nodo.
  @param fileSize indica el tamaño total del archivo que
  contiene la descripcion de la escena.
  @param f es un proxy a un objeto de tipo [File] que se
  utilizara para obtener la pieza asignada.
  @param broker es un proxy a un objeto de tipo [P2PBroker]
  que se utilizara para la localizacion de las piezas
  restantes.

  @see Pieces
  @see File
  @see P2PBroker
  **/
  void obtainFile(Pieces piecesList, int piece, int fileSize,
  File* f, P2PBroker* broker);

  /**
  La operacion render comienza el proceso de renderizado
  de la escena. El nodo ira solicitando la asignacion de
  unidades de trabajo hasta que el gestor de renderizado
  le indique que se ha renderizado la escena completa.

  @param zones es una secuencia de [TZone]. Contiene la
  informacion acerca de las unidades de trabajo.
  @param manager es un proxy a un objeto de tipo
  [YafridngRenderManager]. El nodo lo utilizara para
  obtener nuevas tareas y remitir los resultados de las ya
  realizadas.
```

```
    @throws [RenderingFailedException] si ocurre algun error
    durante el renderizado.

    @see TZones
    @see YafridngRenderManager
    **/
    void render(TZones zones, YafridngRenderManager* manager)
    throws RenderingFailedException;

    /**
    La operacion flush libera los recursos utilizados por
    un renderer. Se invocara cuando finalice el renderizado
    de la escena.
    **/
    void flush();
};

/**
La interfaz YafridngRendererFactory se utilizara para crear
e inicializar objetos del tipo [YafridngRenderer].
**/
interface YafridngRendererFactory
{
    /**
    La opearcion createRenderer crea un objeto de tipo
    [YafridngRenderer] y devuelve un proxy para su
    manipulacion.

    @param RenderEngine indica el motor de render utilizado.
    Puede tomar dos valores, 0 para Blender y 1 para Yafray.
    @param scene contiene el nombre de la escena que se va a
    renderizar.

    @return Un proxy al objeto creado. El cliente utilizara
    este proxy para la manipulacion del objeto.
    **/
    YafridngRenderer* createRenderer(int RenderEngine,
    string scene);
};

/**
La interfaz YafridngNodeManager define las operaciones
necesarias para la gestion de los nodos que proporcionan
sus ciclos de cpu.
```

```
**/
interface YafridngNodeManager
{
  /**
  La operacion addNode permite la adicion de un nodo al
  sistema.

  @param factory es un proxy a un objeto de tipo
  [YafridngRendererFactory].
  Existira        un objeto de este tipo por cada nodo,
  permitiendo la posterior creacion de objetos de tipo
  [YafridngRenderer].

  @throws NodeAlreadyExistsException si ya existe un nodo
  con la misma identidad en el sistema.

  @see YafridngRendererFactory
  @see NodeAlreadyExistsException
  **/
  void addNode(YafridngRendererFactory* factory)
  throws NodeAlreadyExistsException;

  /**
  La operacion allocateNode permite la reserva de uno de
  los nodos para la ejecucion de un trabajo.

  @return un proxy a un objeto de tipo
  [YafridngRendererFactory] que sera utilizado para la
  creacion de objetos de tipo [YafridngRenderer].

  @throws [NoMoreNodesException] si ya no hay mas
  recursos disponibles.

  @see YafridngRendererFactory
  **/
  YafridngRendererFactory* allocateNode()
  throws NoMoreNodesException;

  /**
  La operacion releaseNode permite la liberacion de un
  recurso previamente reservado.

  @param factory indica el recurso que se ha de liberar.
  **/
  void releaseNode(YafridngRendererFactory* factory);
```

```
};

/**
La interfaz RenderSession define las operaciones que se
pueden ejecutar a traves de una sesion. Es necesario
establecer una sesion para hacer uso de Yafrid-NG. La
sesion hara de intermediario entre los usuarios de
Yafrid-NG y el sistema.

@see Glacier2::Session
**/
interface RenderSession extends Glacier2::Session
{
  /**
  La operacion keepAlive se invocara de forma periodica
  para mantener la sesion activa y no destruirla
  prematuramente.
  **/
  void keepAlive();

  /**
  La funcion create se invocara, por parte del cliente,
  para obtener un recurso computacional.

  @param renderEngine indica el motor de render que se
  va a utilizar.
  @param scene indica el nombre de la escena que se va
  a generar.

  @return un proxy a un objeto de tipo [YafridngRenderer]
  que sera utilizado para la ejecucion de un trabajo.

  @throws [NoMoreNodesException] si el cliente ya ha
  reservado los recursos maximos permitidos.

  @see YafridngRenderer
  @see NoMoreNodesException
  **/
  YafridngRenderer* create(int renderEngine, string scene)
  throws NoMoreNodesException;

  /**
  La operacion addNode permite la agregacion de un nuevo
  recurso computacional a Yafrid-NG.
```

```
      @param node indica la factoria proporcionada por el nodo
      para la creacion de objetos de tipo [YafridngRenderer].

      @see YafridngRendererFactory
      **/
      void addNode(YafridngRendererFactory* node);
    };
  };
```

B.2. Callback de la función next

```
void
File_nextI :: ice_response(const ::std::pair<const ::Ice::Byte*,
const ::Ice::Byte*>& bytes)
{
  Lock sync(*this);
  Ice::ByteSeq(bytes.first, bytes.second).swap(_bytes);
  _done = true;
  notify();
}

void
File_nextI :: ice_exception(const ::Ice::Exception& ex)
{
  Lock sync(*this);
  _exception.reset(ex.ice_clone());
  _done = true;
  notify();
}

void
File_nextI :: getData(::Ice::ByteSeq& bytes)
{
  Lock sync(*this);
  while (!_done) {
    wait();
  }
  _done = false;
  if (_exception.get()) {
    auto_ptr<Ice::Exception> ex = _exception;
    _bytes.clear();
    ex->ice_throw();
  }
```

```
  bytes.swap(_bytes);
}
```

B.3. Algoritmo: recuperación de un archivo

```
int FileManager::getFile()
{
  FILE* fp = fopen(_scene.c_str(), _flags.c_str());
  int len = 1000 * 1024;
  int rc = EXIT_SUCCESS;
  int offset = 0;
  IceUtil::Time start = IceUtil::Time::now();
  File_nextIPtr curr, next;
  Ice::ByteSeq bytes;

  fseek(fp, _offset, SEEK_SET);

  for (;;) {
    if (!curr) {
      curr = new File_nextI;
      next = new File_nextI;
      _file->next_async(curr);
    }
    else {
      swap(curr, next);
    }
    _file->next_async(next);
    curr->getData(bytes);
    if (bytes.empty()) {
       break;
    }
    if (fwrite(&bytes[0], 1, bytes.size(), fp)
        != bytes.size()) {
      cerr << "error writing: " << strerror(errno) << endl;
      rc = EXIT_FAILURE;
    }
    else
    offset += bytes.size();
  }

  IceUtil::Time tm = IceUtil::Time::now() - start;
  cout << "Time employed: " << tm.toSecondsDouble() << endl;
  fclose(fp);
```

```
  return rc;
}
```

B.4. Algoritmo: composición de un frame estático

```
void Retrieve::perform()
{
  TZone finalZone;
  int posx = 0, posy = 0;
  int ib = 10;

  //Recovering File
  _file.getFile();

  //Composing the image, if it is a static frame
  if(this->_sceneType == 0){
    try{
      finalZone = getFinalDimensions(_zone, ib, _width,
      _height);

      //Obtaining Mask for transparency
      Image filterWhite(Geometry(finalZone.x2 - finalZone.x1,
      finalZone.y2 - finalZone.y1), "white");
      Image filterBlack(Geometry(_zone.x2 - _zone.x1,
      _zone.y2 - _zone.y1), "black");

      if(finalZone.x1 != _zone.x1)
        posx += ib;
      if(finalZone.y1 != _zone.y1)
        posy += ib;

      //Composing mask and Applying gaussian filter
      filterWhite.composite(filterBlack, posx, posy,
      OverCompositeOp);
      filterWhite.gaussianBlur(ib/2, 20);
      Image source(_file.getFileName());

      //Obtaining transparency values
      for(int i = 0; i < finalZone.x2 - finalZone.x1; i++)
        for(int j = 0; j < finalZone.y2 - finalZone.y1; j++){
          Color pix = filterWhite.pixelColor(i, j);
```

```
          filterWhite.pixelColor(i, j, Color(pix.redQuantum(),
          pix.greenQuantum(), pix.blueQuantum(),
          pix.redQuantum()));
        }

      //Chopping and cropping the image
      source.chop(Geometry(finalZone.x1, finalZone.y1));
      source.crop(Geometry(finalZone.x2 - finalZone.x1,
      finalZone.y2 - finalZone.y1));

      //Copying opacicity
      source.composite(filterWhite, 0, 0, CopyOpacityCompositeOp);

      Image result(_image);
      result.modifyImage();

      //Composing and writing the final result
      result.composite(source, finalZone.x1, finalZone.y1,
      PlusCompositeOp);
      result.write(_image);
    }
    catch(Exception& err)
      {
        cout << "Exception composing Image: " << err.what();
      }
  }
  _cb->ice_response();
}
```

Apéndice C

Manual de usuario

C.1. Arrancando Yafrid-NG

A continuación se explicará como poner en funcionamiento Yafrid-NG en una máquina con acceso a internet. Junto con Yafrid-NG se proporcionan tres scripts. El primero nos permite compilar Yafrid-NG, el segundo nos permite arrancar el sistema y el tercero para pararlo.

El código fuente se encuentra en la carpeta *src*. En el nivel superior hay un script llamado *initYafrid.sh* que cambiará el directorio, compilará Yafrid-NG y volverá al directorio actual. El contenido de *initYafrid.sh* es:

```
!#/bin/sh

echo ""
echo "Script that initializes Yafrid-NG"
echo ""

cd src
echo "Cleaning previous compilations..."
make clean
echo "Compiling source code..."
make yafridng
cd ..
```

Para arrancar el sistema basta con ejecutar el script *startYafrid.sh* que se encargará de arrancar el nodo y el registry, desplegar la aplicación y poner en funcionamiento el servicio

de *Glacier2*. A continuación se muestra el contenido del archivo *startYafrid.sh*:

```
#!/bin/sh

echo ""
echo "Script to start Yafrid-NG"
echo ""

echo "Deploying the node and the registry..."

icegridnode --Ice.Config=config.icegrid --daemon --nochdir
sleep 3

echo ""
echo "Deploying Yafrid-NG..."
icegridadmin --Ice.Config=config.icegrid
    -e "application add yafridng.xml"

#arrancando el router
icegridadmin --Ice.Config=config.icegrid
    -e "server start Yafridng.Glacier2"
```

Para detener el sistema basta con ejecutar el script *stopYafrid.sh* cuyo contenido es el siguiente:

```
!#/bin/sh

echo ""
echo "Script to stop Yafrid-NG"
echo ""

echo "Removing Yafrid-NG application..."
icegridadmin --Ice.Config=config.icegrid
    -e "application remove Yafridng"

echo "Shutting down Node1"
icegridadmin --Ice.Config=config.icegrid
    -e "node shutdown Node1"
```

Para la ejecución de Yafrid-NG en una máquina se hace necesario seguir unos sencillos pasos de configuración. A continuación se detallan estos pasos uno por uno para un facil seguimiento por parte del usuario.

Al compilar Yafrid-NG se generán dos librerías dinámicas llamadas *libSessionManagerServiceI.so* y *libYafridngNodeManager.so* que se corresponden con el gestor de sesiones y el gestor de nodos, respectivamente. Es necesario que la variable de entorno **LD_LIBRARY_PATH** contenga la ruta en la que se encuentran dichos archivos. Normalmente se encontrarán dentro del directorio *lib* o del directorio *src*.

Por defecto, al ejecutar el script *startYafrid.sh* todos los servicios aceptarán conexiones desde la máquina local (*localhost*). Si se quiere proporcionar acceso a otras máquinas es necesario realizar un pequeño cambio en el descriptor de la aplicación, *icegrid/yafridng.xml*.

```
<server-instance template="Glacier2"
client-endpoints="ssl -h <dir_ip> -p <port>"
server-endpoints="tcp"
session-timeout="30">
```

Es necesario proporcionar valores adecuados para el campo *<dir_ip>*, correspondiente a la dirección ip del equipo en el que se ejecuta el sistema, y el campo *<port>*, correspondiente al puerto utilizado.

C.2. Ejecutando un proveedor

Los archivos necesarios para ejecutar el software proveedor se encuentran dentro de la carpeta *provider*. Para proporcionar ciclos de CPU y participar en tareas de renderizado basta con ejecutar el archivo *YafridngProvider*. El comando que hay que ejecutar es:

```
YafridngProvider --Ice.Config=config.node
```

El archivo *config.node* contiene la información necesaria para la correcta inicialización del *runtime* de ICE. El contenido de este archivo es el siguiente:

```
Ice.Default.Router=Yafridng.Glacier2/router:
   ssl -h <dir_ip> -p <port>
Ice.ThreadPool.Server.Size=6
Ice.ThreadPool.Server.SizeMax=10
Ice.ThreadPool.Client.Size=6
Ice.ThreadPool.Client.SizeMax=10
YafridngNodeAdapter.Endpoints=tcp
YafridngResultAdapter.Endpoints=tcp
```

```
YafridngResultAdapter.ThreadPerConnection=1

#Ice.Trace.Network=1

Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.CertAuthFile=ca_cert.pem
IceSSL.CertFile=user_cert.pem
IceSSL.KeyFile=user_key.pem
IceSSL.Password=yafridng
IceSSL.DefaultDir=certs
IceSSL.TrustOnly=CN="Ice Server Glacier2 Router"
```

Es necesario editar este archivo y proporcionar la dirección ip y el puerto en los que escucha el servicio *Glacier2*.

Bibliografía

[1] Blender's official site. http://www.blender.org.

[2] Boinc's official site. http://boinc.berkeley.edu.

[3] Burp's official site. http://burp.boinc.dk.

[4] Dr queue's official site. http://www.drqueue.org.

[5] Image magick's official site. http://www.imagemagick.org.

[6] .net remoting architecture. http://msdn2.microsoft.com/en-us/.

[7] Omg's corba site. http://www.corba.org/.

[8] Remote method invocation. http://www.java.sun.com/products/jdk/rmi/.

[9] Word wide web consortium site. http://www.w3.org.

[10] Zeroc's newsletter for the ice community. http://www.zeroc.com/newsletter.

[11] Zeroc's official site. http://www.zeroc.com/.

[12] W. J. Bouknight. A procedure for generation of three-dimensional half-toned computer graphics. *Communications of the ACM*, 1970.

[13] R. Buyya and V. Srikumar. A gentle introduction to grid computing and technologies. *CSI Communications*, 29:9–19, July 2005.

[14] L. Carpenter, R.L. Cook, and T. Porter. Distributed ray tracing. *Computer Graphics*, 1986.

[15] A. Chakrabarti. *Grid Computing Security*. Springer, 2007.

[16] A. Chalmers, T.A. Davis, and E. Reinhard. *Practical Parallel Rendering*. AK Peters, 2002.

[17] J.A. Fernandez Sorribes. *Sistema Grid para el render de escenas 3D distribuido: Yafrid*. 2006.

[18] J. A. Ferwerda. Three varieties of realism in computer graphics. Technical report, Cornell University, 2003.

[19] I. Foster and C. Kesselman. *The GRID: Blueprint for a new computing infraestructure*. Morgan Kaufman, 2004.

[20] B. Foucher. Master-slave replication with ice. *Connections*, 2007.

[21] C. González Morcillo, G. Weiss, L. Jiménez, D. Vallejo, and J. Albusac. A multiagent system for physically based rendering optimization. 2007.

[22] C. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. *Proceedings of SIGGRAPH*, 1984.

[23] M. Henning. *Advanced CORBA programming with C++*. Addisson Wesley, 1999.

[24] M. Henning. To slice or not to slice. *Connections*, 2005.

[25] M. Henning and M. Spruiell. *Distributed Programming with ICE*. 2007.

[26] B. Jacob. Grid computing: What are the key components? taking advantage of grid computing for application enablement. Technical report, IBM Developerworks Grid Library, 2003.

[27] H. W. Jensen. Global illumination using photon maps. *Eurographics Rendering Workshop*, 1996.

[28] J. T. Kajiya. The rendering equation. *Computer Graphics*, 1986.

[29] N. Minar. Distributed systems topologies. http://www.open-p2p.com/, 2001.

[30] Daniel Minoli. *A Networking Approach to Grid Computing*. Wiley-Interscience, 2004.

[31] M.Ñewhook. Custom sessions and icegrid. *Connections*, 2006.

[32] M.Ñewhook. Optimizing performance of file transfers. *Connections*, 2006.

[33] Detlef Schoder and Kai Fischbach. Peer-to-peer prospects. *Commun. ACM*, 46(2), 2003.

[34] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *P2P '01: Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, Washington, DC, USA, 2001. IEEE Computer Society.

[35] R. Subramanian and B. D. Goodman. *Peer to Peer Computing: The Evolution of a Disruptive Technology*. Idea Group Publishing, 2005.

[36] I. Victor Kerlow. *The art of 3D computer animation and imaging*. John Wiley & Sons, 2000.

[37] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 33(6):343–349, June 1980.

[38] S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. *Eurographics Rendering Workshop*, 1998.