ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Propuesta para una plataforma Peer to Peer basada en agentes Java/CORBA

AUTOR: Ana Belén Mateos García
DIRECTOR: Javier Vales Alonso

Julio **/2005**

Autor	Ana Belén Mateos García
E-mail del Autor	ANIKABE1@GMAIL.COM
Director(es)	Javier Vales Alonso
E-mail del Director	Javier.Vales@upct.es
Codirector(es)	
Título del PFC	Propuesta para una plataforma Peer to Peer basada en agentes Java/CORBA
Descriptores	Sistema distribuido, Java, CORBA

Resumen

El objetivo de este proyecto ha sido diseñar e implementar una arquitectura P2P para la obtención distribuida de recursos basada en agentes. Estos agentes interactúan entre si para ofrecer una infraestructura de transporte de información y de selección de pares. Asimismo, los agentes tienen la misión de controlar la admisión a sus servicios por parte de otros agentes. Para ello, establecen un mecanismo básico de gestión y creación de contratos. En la arquitectura desarrollada, los contratos son implementados a través de un mecanismo de control de flujo del tipo *Leaky Bucket*. De este modo, durante la fase de establecimiento los nodos negocian los parámetros del *Leaky Bucket* y el nodo proveedor admite peticiones sólo al ritmo negociado.

El sistema desarrollado está orientado a ofrecer servicios generales, como por ejemplo cesión de ancho de banda entre equipos de una *intranet*, cesión de CPU, etc. No obstante, en el proyecto se ha probado sólo una aplicación típica de redes P2P: la compartición de ficheros.

Asimismo, las pruebas desarrolladas han tenido únicamente en cuenta el caso de compartición "altruista", donde los agentes cedían siempre sus recursos si estos estaban disponibles. Finalmente, para la implementación se ha escogido el sistema de *middleware* CORBA, junto al lenguaje de programación Java. A través de los servicios de CORBA, se han conseguido desarrollar fácilmente los agentes de la red P2P.

Titulación	Ingeniería Técnica de Telecomunicaciones
Intensificación	Telemática
Departamento	Tecnología de la Información y las Comunicaciones (TIC)
Fecha de Presentación	Julio- 2005

TABLA DE CONTENIDOS

CAPÍTULO 1 – INTRODUCCIÓN

CAPÍTULO 2 - TRABAJOS RELACIONADOS

CAPÍTULO 3 - ARQUITECTURA

CAPÍTULO 4 - DESARROLLO

CAPÍTULO 5 – PRUEBAS

CAPÍTULO 6 - CONCLUSIONES Y LÍNEAS FUTURAS

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE GRÁFICAS

CAPÍTULO 1

INTRODUCCIÓN

1.1-Resumen

El presente proyecto denominado "*Propuesta para una plataforma Peer to Peer basada en agentes Java/CORBA"* tiene como objetivo diseñar e implementar una arquitectura *Peer to Peer* para obtener recursos de un modo distribuido basada en agentes.

Un agente es un proceso que realiza básicamente tres funciones: pedir, servir y limitar el acceso a sus servicios por parte de otros agentes de la red. Para desarrollar todas estas funciones se dispone de una serie de mecanismos de control, los contratos, basados en un algoritmo conocido como *leaky bucket*. Éstos son negociados entre los distintos agentes registrados en un servidor.

Todo agente que llega a la red, es registrado en un servidor denominado gestor, donde recibirá un identificador único y podrá acceder a información a cerca de todos los agentes pertenecientes a la red. Esta información, será utilizada posteriormente para poder hacer negociaciones, es decir para establecer contratos con otros pares de la red.

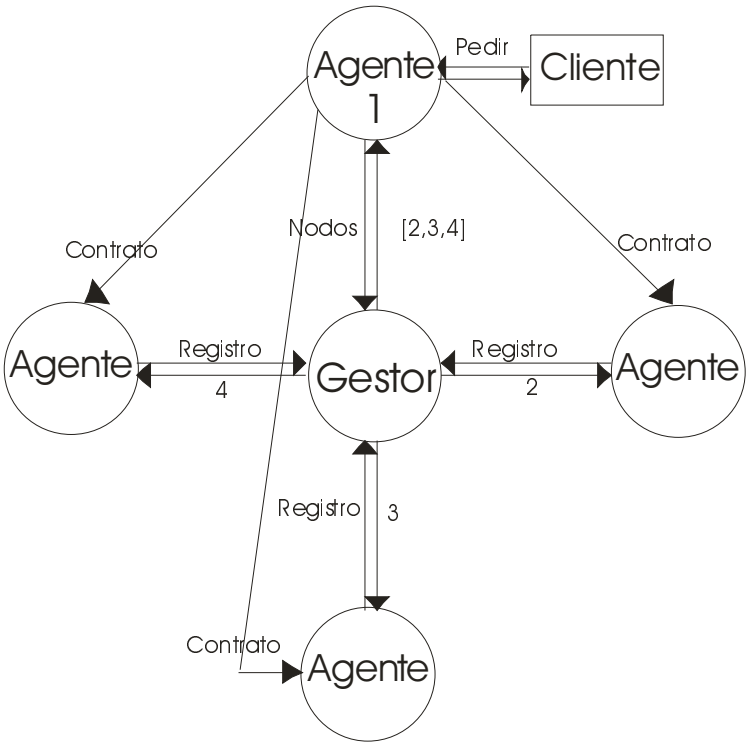


FIGURA 1.1.1.- Funcionamiento del programa

Al establecer un contrato, el agente negociará los parámetros del *leaky bucket* y sólo transmitirá peticiones al ritmo establecido en dicho contrato. De este modo, cada agente se asegura no dedicar más ancho de banda del establecido, y así poder dedicar el resto a otros cometidos. Otra razón por la que se establecen contratos es la de evitar que los recursos de los agentes queden congestionados.

La arquitectura P*eer to Peer* de este proyecto, está ideada para ofrecer cualquier tipo de servicio, ya sea la cesión de ancho de banda entre los distintos equipos de una intranet, la cesión de CPU, etc. En este caso, ha sido desarrollado de tal modo que se puedan compartir archivos entre distintas máquinas con la única limitación descrita en los contratos.

La aplicación desarrollada en el proyecto esta formada un sistema híbrido, es decir, consta de partes centralizadas y no centralizadas. El sistema esta formado por agentes registrados en un servidor.

La acción de servir ficheros se realiza entre diversos agentes, es decir, cada agente que ha establecido un contrato con el agente que hizo la petición, irá transmitiendo distintos fragmentos, hasta completar el archivo pedido. De este modo es posible obtener el archivo deseado en un menor periodo de tiempo.

En este tipo de arquitectura se puede jugar con distintos parámetros para dar servicio, es decir, se pueden establecer múltiples condiciones para la negociación de los contratos. Se puede disponer de un sistema dotado con nodos de "distintas personalidades", disponiendo así de un modo bastante realista. En el caso de este proyecto, se tiene una única personalidad, que es la altruista, de modo que los agentes no controlan que otros hayan compartido o hayan negado servicios.

Del mismo modo, también se podría establecer como condición la cantidad de servicio que tiene que haber ofrecido un agente para poder recibir servicios de otros agentes. Como se puede observar se puede tener una amplia gama de condiciones en este tipo de arquitecturas.

Si observamos la figura 1.1.1. se puede ver de modo gráfico el funcionamiento de nuestro sistema. Los agentes de la izquierda, derecha y la parte inferior hacen una llamada al gestor denominada "*Registro"* cuyo objetivo no es otro que el de registrarse en el servidor gestor, obteniendo como respuesta un identificador único. A continuación, el agente de la parte superior de la figura, registrado previamente, hace una petición de un archivo a través de una consola denominada cliente, que es controlada por la persona que maneja la aplicación. Lo primero que sucede, es que este agente pide y recibe información al gestor sobre los distintos agentes registrados en él. Una vez obtenida dicha información, se procede a la de uno o más agentes registrados en la red con posesión del archivo que quiere obtener el agente local. En el caso de encontrar agentes con dicho fichero, se procede a establecer un contrato con ellos y seguidamente a comenzar con la transmisión de éste. Finalmente, el agente local dispondría del archivo deseado.

1.2 Organización del proyecto fin de carrera en capítulos

A continuación se puede ver una breve descripción sobre los capítulos que contiene este documento, así como una breve descripción del contenido de cada uno de ellos.

Capítulo 2.- Trabajos Relacionados

Este capítulo consta de un estudio a cerca de las redes *Peer to Peer*. Se puede encontrar su definición, los tipos de redes que existen, las distintas aplicaciones que se le pueden dar, los programas más utilizados en la actualidad y su modo de actuar. Explica también la historia de este tipo de redes desde su aparición hasta los problemas que existen en la actualidad con el ámbito musical. También se puede encontrar una relación con los lenguajes de programación más utilizados para implementar este tipo de redes, así como una breve descripción de cada uno.

Capítulo 3.- Arquitectura

Podemos ver una descripción a cerca de la arquitectura del proyecto, explicando las diferentes partes que de las que se compone, modelo de contratos, algoritmos utilizados, etc.

Capítulo 4.- Desarrollo

Aquí se puede encontrar toda la información a cerca del código del proyecto. Consta de dos apartados. En el primero se puede encontrar una explicación resumida de partes de las que consta el proyecto y en el segundo una explicación más detallada de cada método y su funcionamiento.

Capítulo 5.- Pruebas

En este capítulo se puede encontrar una relación de pruebas realizadas en el laboratorio. Consta de diversos escenarios, donde cada uno describe una situación distinta. Aquí se pueden ver también los resultados obtenidos de las distintas ejecuciones que nos hacen una descripción del funcionamiento global del proyecto.

Capítulo 6.- Conclusiones y líneas futuras

Consta de las conclusiones obtenidas a lo largo del desarrollo del proyecto, así como un conjunto de ideas para futuras mejoras y ampliaciones de éste.

Anexo 1.- Modo de Ejecución del *software*

Este anexo es una pequeña guía a cerca de cómo ejecutar el proyecto. Contiene todas las instrucciones necesarias para ejecutar cada componente como el servidor de nombres de CORBA, el gestor donde se registrará cada agente y los agentes.

CAPÍTULO 2

TRABAJOS RELACIONADOS

2.1-Redes *Peer to Peer*

2.1.1. Definición

Una red *Peer to Peer* (entre iguales o P2P) es una red que no tiene clientes y servidores fijos, sino una serie de nodos que se comportan a la vez como clientes y como servidores de los demás nodos de la red. Este modelo de red contrasta con el modelo cliente-servidor. Cualquier nodo puede iniciar o completar una transacción compatible. Los nodos pueden diferir en configuración local, velocidad de proceso, ancho de banda de su conexión a la red y capacidad de almacenamiento.

Debido a que la mayoría de los ordenadores domésticos no tienen una IP fija, sino que le es asignada por el proveedor (ISP) en el momento de conectarse a Internet, no pueden conectarse entre sí porque no saben las direcciones que han de usar de antemano. La solución habitual es realizar una conexión a un servidor (o servidores) con dirección conocida (normalmente IP fija), que se encarga de mantener la relación de direcciones IP de los clientes de la red, de los demás servidores y habitualmente información adicional, como un índice de la información de que disponen los clientes. Tras esto, los clientes ya tienen información sobre el resto de la red, y pueden intercambiar información entre sí, ya sin intervención de los servidores. (Ver Referencias [2.1]).

2.1.2. Tipos de redes Peer to Peer

Los programas P2P consisten esencialmente en un motor de transferencia de archivos entre usuarios corrientes conectados entre sí. El programa gestiona los intercambios entre *peers* (usuarios, nodos) usando los recursos de sus propios ordenadores (Ver Referencias [2.2]).

Hay tres tipos de redes, según tengan éstas, o no, un servidor central que gestione las transacciones:

- **Las redes centralizadas**. Los nodos están conectados a un servidor central donde publican información sobre los contenidos que ofertan para compartir. Cuando un nodo realiza una petición, el servidor central le indicará el nodo al cual debe conectarse para iniciar la transmisión. Es lógico pensar que una red gestionada a través de una sola máquina tiene un índice de vulnerabilidad alto ya que cualquier ataque que se produzca a dicho servidor supone la anulación de todas las operaciones. Además este modelo necesita una amplia infraestructura de direccionamiento con toda la información de los *host* que forman la red. Este problema provoca límites de escalabilidad debido a que requiere un servidor con una capacidad cada vez mayor cuando aumente la cantidad de nodos en la red. [FIGURA 2.1.2.1]

Las redes descentralizadas Se crearon para evitar el problema de la vulnerabilidad de las redes con un servidor central, entre las cuales destacan *Gnutella y Freenet*. Estas redes no utilizan servidor central y por tanto son mucho menos susceptibles a ataques pero, en cambio, la gestión de las operaciones de búsqueda y transferencia es mucho menos eficiente. Cada nodo se puede comunicar de manera equitativa, igualitaria con otro nodo. Cuando un nodo (A) se conecta a una red descentralizada, se conecta a un nodo (B) para avisarle que está vivo. El nodo (B) a su vez informa a toda la comunidad a la que está conectado (C, D, E, F, etc.) de la presencia de (A). Se repite el patrón con el resto de la comunidad. Una vez que (A) ha anunciado que está vivo, puede enviar un requerimiento de búsqueda a (B) que lo traspasa al resto de nodos. Si por ejemplo (C) tiene una copia del archivo requerido por (A), transmite a (B) una respuesta quien la pasa a (A) quien finalmente puede abrir una conexión directa con (C) y bajar el archivo. Esta propagación suele estar controlada por un TTL (*Time To Live)*. Este sistema tiene la ventaja que ya no existe un único servidor y que es mucho más difícil matar los múltiples servidores. Es un sistema más lento que el centralizado, y no garantiza encontrar un archivo incluso si existe en la red, porque puede estar mucho muy lejos como para que un requerimiento de búsqueda alcance a un nodo que lo tenga antes de cumplir su TTL. (Ver la [FIGURA 2.1.2.2]). (Ver Referencias [2.10]).

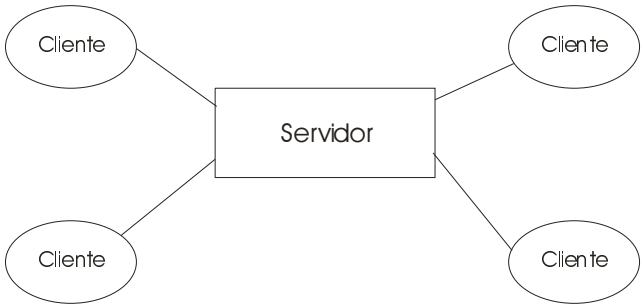


FIGURA 2.1.2.1.- Tipos de redes P2P. P2P centralizada (Napster, OpenNap)

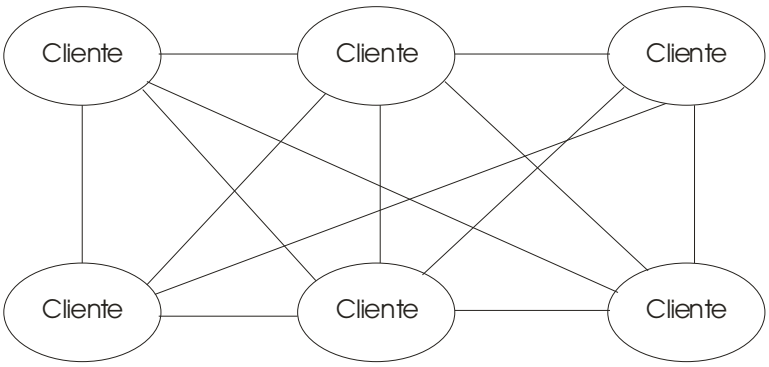


FIGURA 2.1.2.2.- Tipos de redes P2P. P2P descentralizada (Gnutella, Freenet)

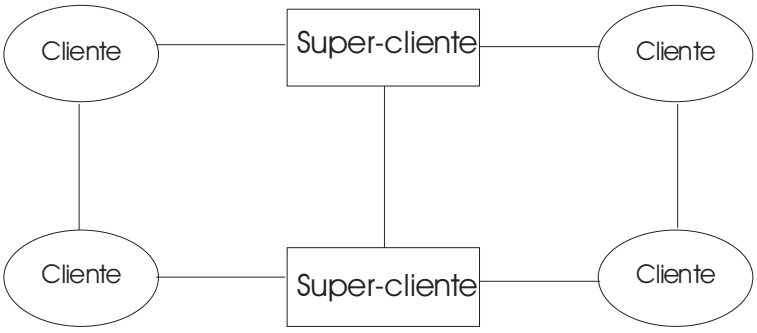


FIGURA 2.1.2.3.- Tipos de redes P2P. P2P híbrida (*Fast Track, eDonkey, WinMX*)

- **Las redes híbridas**. Son las que actualmente gozan de una mayor representación. Sus creadores han sabido aprovechar las ventajas del sistema centralizado y del descentralizado consiguiendo, así, un equilibrio entre resistencia a ataques y eficiencia. Las operaciones de búsqueda y transferencia son gestionadas en este caso por los denominados *súper-peers* (súper-clientes) que actúan como nodos activos y agilizan el funcionamiento de la red. Explicado de otro modo, los usuarios hacen las búsquedas de archivos en un servidor centralizado. Éste localiza la ubicación de los archivos y posteriormente y los usuarios lo descargan directamente del nodo que contenga dicho archivo. Por esta razón, este sistema se denomina híbrido. Actualmente este tipo de sistemas suelen ser más eficaces que los centralizados porque algunas tareas como la búsqueda se pueden hacer de un modo más eficiente. *Kazaa y eDonkey* son un buen ejemplo de gestión eficiente y optimización de los recursos. (Ver la [FIGURA 2.1.2.3]). (Ver Referencias [2.9]).

En este proyecto se podría clasificar la red *Peer to Peer* como híbrida debido a que se tienen partes centralizadas y partes descentralizadas. El servidor de nombres de CORBA y el servidor gestor están centralizados, es decir, corren sobre una sola máquina. Con el servidor de nombres se pueden localizar los distintos elementos distribuidos de la red, mientras que en el servidor gestor están registrados todos los nodos que hay en ésta. Por otro lado está la parte descentralizada, que se corresponde con los distintos nodos de la red y los archivos para compartir. De este modo cuando un nodo quiera un archivo, se buscarán en el gestor los nodos que hay registrados (centralizados) y luego se buscará la ubicación de dicho archivo en los nodos (descentralizado).

2.1.3. Aplicaciones

Existen cinco categorías en las que se pueden clasificar las aplicaciones de las redes *Peer to Peer* (Ver Referencias [2.3]).

- Colaboración
- Servicios de alta tecnología
- Servicios de Agentes Inteligentes
- Compartición de Archivos
- Computación Distribuida

Los colaboradores de un proyecto pueden comunicarse y conducir el trabajo juntos vía tecnología P2P. P2P además provee servicios de elite a sus clientes usuarios para, en esencia, mover o almacenar repositorios de datos más cerca de Internet. Y con los servicios de P2P basados en agentes inteligentes, agentes en diferentes computadoras se comunican directamente para realizar tareas.

Un ejemplo de aplicación de computación distribuida podría ser un sistema que necesite hacer una cantidad muy elevada de operaciones. Este sistema puede utilizar centenares de ordenadores diferentes dispersos en institutos y centros de investigación de muchos países para hace una misma tarea, participar en el computo, de forma que cada uno almacene una parte del cómputo.

Un sistema *Peer to Peer* con servicio de agentes inteligentes denominado *DIAMS* tiene la función de ayudar a los usuarios a acceder, administrar, compartir e intercambiar información relevante para ellos en la Web. El esquema de colaboración de los agentes en DIAMS incluye agentes personales que se encargan de presentar, organizar y administrar las colecciones de información de los usuarios. Esos agentes interactúan con otros dos tipos de agentes, que son los agentes de asociación y los agentes de conocimiento. Los primeros facilitan la colaboración entre agentes personales, pues se ocupan de obtener conocimiento de otros usuarios, y de comunicar los que presentan intereses más cercanos; los segundos contienen conocimiento especializado que ponen al servicio de los agentes personales. (Ver Referencias [2.11]).

Otro ejemplo de aplicación de redes *Peer to Peer* sería la aplicación desarrollada en el ámbito del proyecto IST europeo SWAP (*Semantic Web & Peer-to-Peer*), mediante el cual se ha implantado la tecnología P2P en distintas organizaciones que operan en el sector turístico de las Islas Baleares. La red P2P posibilita la compartición de conocimiento entre dichas organizaciones, y constituye una útil herramienta de soporte a la toma de decisiones para la gestión global de las Islas Baleares como destino turístico. (Ver Referencias [2.12]).

2.1.4. Breve historia de los programas Peer to Peer

La historia de los sistemas P2P se remonta a finales de la década de 1970 (Ver Referencias [2.2]). Las redes *Usenet* (1979) y *Fidonet* (1984) son consideradas sus madres. En un principio, fueron desarrolladas como redes de intercambio de noticias entre varios campus universitarios de los EE.UU. Desde principios de los años 90 muchas grandes corporaciones internacionales como Intel y Boeing empezaron a usar redes P2P para realizar operaciones con gran volumen de cálculos, utilizando miles de ordenadores de todo el mundo al mismo tiempo. Este tipo de iniciativas se extendió a proyectos científicos que operaban con gran cantidad de datos, lo que les permitió prescindir de los engorrosos y costosos superordenadores.

No fue hasta mayo de 1999 que el uso de las redes P2P se hizo masivo. Shawn Fanning y Sean Parker, estudiantes de la Northeastern University (Boston, EE.UU.), crearon entonces *Napster,* una red cuyo objetivo principal era el intercambio de archivos musicales. La red se valía de un cliente (programa) que se podía descargar desde cualquier parte del mundo a través de la *World Wide Web*. El programa se hizo inmediatamente famoso entre los

ínter nautas, dado el gran abanico de posibilidades que proporcionaba. Con *Napster*, cualquier usuario podía conectarse y descargar el último disco de su artista favorito antes, incluso, de que éste saliera al mercado en su país. También ofrecía la posibilidad de encontrar canciones inéditas de artistas poco conocidos, versiones raras o limitadas de determinados LPs o música que, simplemente, no se podía conseguir sin ir específicamente a un determinado país.

A modo de paréntesis, sin embargo, no debemos olvidar que la popularización de las grabadoras de CD para ordenador y la caída espectacular de los precios que éstas experimentaron entre 1999 y 2001, contribuyeron en gran medida a fomentar el aumento espectacular del uso de los P2P.

Las grandes discográficas norteamericanas no tardaron en ser conscientes de la magnitud del fenómeno. La organización que las agrupa, la *Recording Industry Association of America* (R.I.A.A), denunció inmediatamente el desafío que estos programas suponían, tanto a los derechos de producción como a los derechos de autor. A finales del mismo año, cuando los miembros de la red ya superaban el millón, la R.I.A.A interpuso la primera querella ante una Corte Federal de San Francisco por violación de los derechos de autor. Entre abril de 2000 y julio del mismo año se libró una batalla judicial en la que intervinieron, además de la R.I.A.A., ciertos artistas de renombre, como el grupo Metallica. Esta batalla finalizó el 26 de julio, con el decreto de suspensión de todas las actividades de *Napster* por parte de una juez del distrito de Washington.

Parecía así haberse llegado al punto final de la aventura del intercambio gratuito de archivos en la red. Nada más lejos de la realidad. El cierre de *Napste*r supuso un punto de inflexión en la reciente historia de las P2P. El programa no tardó en mutar en otros que mejoraron algunas de las deficiencias de su predecesor, entre ellas, la imposibilidad de prescindir de un servidor central. Ante los indicios de que el tráfico de archivos no disminuía sino que, por el contrario, aumentaba espectacularmente, las grandes compañías discográficas se enrolaron en una batalla legal que, todavía hoy, parece muy lejos de llegar a su final.

2.1.5 Programas *Peer to Peer* más conocidos

Los programas *Peer to Peer* más conocidos actualmente, son aquellos que permiten a los usuarios poder obtener ficheros mayormente multimedia como pueden ser juegos, música, videos, etc. Algunos de los más conocidos son:

- eMule
- eDonkey
- Overnet
- Bittorrent

EMule (Ver Referencias [2.4]) es un programa de código abierto "*open source*", que trabaja en la red eDonkey (Ver Referencias [2.5]) que es un desarrollo propietario de Jed McCaleb.

EMule, eDonkey y Overnet además de permitir el intercambio directo entre clientes sin necesidad de un servidor central, cuentan con la propiedad de trocear el fichero en trozos de 10MB por lo que en caso de fallo en la descarga de un trozo basta con repetir la descarga del trozo corrupto. Además permite continuar la descarga desde otro lugar en caso de que se interrumpa la conexión con el lugar desde donde estábamos haciendo la descarga, sincronizándose al último trozo descargado. También permite descargas en paralelo de trozos de un mismo fichero desde varios sitios simultáneamente y sobre todo algo que hace que la red *eDonkey* sea muy rápida y viva es que un trozo descargado inmediatamente está disponible para que otros puedan descargarlo desde nuestro PC.

EDonkey (eD2k) y *eMule* usan la misma red. Sin embargo *eMule* tiene un interfaz gráfico más potente. *EMule* fuerza el upload y lo premia mediante un sistema de créditos, con el fin de evitar comportamientos egoístas (recibir muchos ficheros pero no enviar ninguno), con *eDonkey* esto era opcional.

eMule trabaja con bloques de ficheros de 5 MB en lugar de los 10MB que usa eDonkey para dotar a la red de mayor velocidad, dado que si un fichero está corrupto es necesario descargarlo entero de nuevo.

2.1.6 Variantes de la tecnología *Peer to Peer*

"*Peer2Mail*" (P2M)

Creado por el programador israelí Ran Geva, *Peer2Mail* (P2M) es producto de la evolución de una serie de herramientas que comenzó en URLBlaze y que ha concluido, al menos de momento, en esta combinación de *'Peer to Peer'* y correo electrónico llamada a hacerse un hueco entre los 'gigantes' de las descargas desde un punto de vista diferente.

"Tan fácil como enviar un 'e-mail'", asegura su eslogan. Y es así: quien sepa acceder a una cuenta de correo puede utilizar P2M sin dificultades para 'bajarse' a su ordenador el material en el que esté interesado, con el ojo puesto únicamente en la mayor o menor velocidad que alcance su conexión.

Y siempre que alguien lo haya 'subido' previamente a la Red, claro. Porque la piedra angular de P2M, como la de todos los demás sistemas P2P, está en compartir. Si los usuarios quieren descargar un archivo, es necesario que otra persona los haya colocado antes en una cuenta gratuita de *'e-mail'*. (Ver Referencias [2.6]).

2.2-Lenguajes de Programación en los que se han programado aplicaciones Peer to Peer

Una aplicación se puede realizar con multitud de lenguajes de programación. Algunos de éstos son los siguientes:

2.2.1 JAVA

Java es una plataforma de software desarrollada por *Sun Microsystems*, de tal manera que los programas creados en ella puedan ejecutarse de la misma forma en diferentes tipos de arquitecturas y dispositivos computacionales. Como lenguaje es simple, orientado a objetos, distribuido, interpretado, robusto, seguro, neutral con respecto a la arquitectura, portable, de alta perfomance, multithreaded y dinámico.

A la hora de crear un programa *Peer to Peer* con Java pueden utilizarse varias opciones:

- Sockets: Mediante la clase Socket incluida en el paquete java.net podemos crear conexiones de flujo, que son los que utilizan el protocolo TCP, entre dos ordenadores. Los Sockets son conexiones entre dos ordenadores remotos con una comunicación continúa que finaliza cuando uno de los dos ordenadores cierra su conexión.

- CORBA (*Common Object Request Broker Architecture*): Es una arquitectura independiente del lenguaje de programación que proporciona mecanismos uniformes para invocar objetos remotos de un modo transparente. Sirve para automatizar tareas de sistemas distribuidos como pueden ser el registro, localización y activación de objetos.

- RMI (*Remote Method Invocation*) Al igual que CORBA, es un mecanismo que sirve para invocar y ejecutar procedimientos remotos, pero es una tecnología para el desarrollo de aplicaciones Java únicamente.

- JXTA es un conjunto de protocolos abiertos para construir aplicaciones Peer to Peer. Permite interconectar distintos tipos de dispositivos de red, desde teléfonos celulares, hasta PDAs. JXTA crea una red virtual donde cada nodo puede interactuar con los demás.

2.2.2 Microsoft .NET

Microsoft .Net es un modelo de computación distribuida, y provee los cimientos para la nueva generación de software. Utiliza los Servicios *Web* como un medio para poder interoperar a distintas tecnologías. Permite conectar distintos sistemas operativos, dispositivos físicos, información y usuarios. Les da a los desarrolladores las herramientas y tecnologías para hacer rápidamente soluciones de negocios que involucran distintas aplicaciones, dispositivos físicos y organizaciones. La idea central detrás de la plataforma .NET es la de servicio. Más concretamente software como servicio y de cómo construir, instalar, consumir, integrar o agregar (en federaciones) estos servicios para que puedan ser accedidos mediante Internet.

2.2.3 C#

C# es un lenguaje de propósito general orientado a objetos creado por Microsoft para la plataforma .NET. Es el conjunto de nuevas tecnologías en las que Microsoft ha estado trabajando para mejorar tanto su sistema operativo como su modelo de componentes (COM) para obtener una plataforma con la que sea sencillo el desarrollo de software en forma de servicios *Web*. Éstos son un novedoso tipo de componentes software que se caracterizan a la hora de trabajar por su total independencia respecto a su ubicación física real, la plataforma sobre la que corre, el lenguaje de programación con el que hayan sido desarrollados o el modelo de componentes utilizado para ello.

C# combina los mejores elementos de múltiples lenguajes de amplia difusión como C++, Java, Visual Basic o Delphi. La idea principal detrás del lenguaje es combinar la potencia de lenguajes como C++ con la sencillez de lenguajes como Visual Basic, y que además la migración a este lenguaje por los programadores de C/C++/Java sea lo más inmediata posible.

2.2.4 Phyton

Python es un lenguaje de programación script, interpretado, interactivo y orientado a objetos. Se le compara con Tcl, Perl, Scheme o Java. Destaca en una sintaxis muy sencilla y limpia pero con gran potencia. Contiene módulos, clases, tipos de datos de alto nivel y escritura dinámica. Tiene interfaces para diversos sistemas y librerías. También puede utilizarse como un lenguaje de extensión para aplicaciones que necesitan una interfaz programable. Otra ventaja es su portabilidad, funcionando en sistemas Unix y derivados, Windows, Dos, Mac y otros. No hay necesidad de compilar código en Python antes de ejecutarlo, razón que lo convierte en un lenguaje de script.

Con este lenguaje de programación se creo una aplicación Peer to Peer que en tan solo quince líneas permite compartir miles de ficheros. (Ver Referencias [2.7]) TinyP2P fue escrito por un profesor de Ciencias de la Computación de la Universidad de Princeston (Edward Felten) para demostrar que la prohibición P2P es virtualmente imposible. El código se puede encontrar en la página Web:

http://www.freedom-to-tinker.com/tinyp2p.py

2.2.5 Perl

Perl (*Practical Extraction and Report Language*) es un lenguaje de programación desarrollado por Larry Wall inspirado en otras herramientas de UNIX como son: sed, grep, awk, c-shell, para la administración de tareas propias de sistemas UNIX. Es similar a C, y esta muy extendido en labores de administración de sistemas y programación CGI(*Common Gateway Interface*) en los servidores Web.

A partir de Perl se creó un programa *P2P, MoleSter 0.0.4* ocupa 6 líneas de código, 466 bytes. Esto se hizo después de que el código del programa comentado en el punto 2.2.2, *TinyP2P* fuera publicado. (Ver Referencias [2.8]). Se puede encontrar el código fuente junto con los comentarios en la siguiente dirección de Internet:

http://ansuz.sooke.bc.ca/software/molester/molester

CAPÍTULO 3

ARQUITECTURA

3.1. Introducción

El objetivo de este proyecto es estudiar con un modelo sencillo, pero suficiente, el comportamiento de una arquitectura *Peer to Peer* basada en agentes para obtener recursos de un modo distribuido.

La arquitectura de este proyecto esta pensada para ofrecer cualquier tipo de servicio, pero se ha particularizado de tal modo que los distintos agentes que componen una red puedan compartir ficheros entre distintas máquinas.

La aplicación desarrollada en el proyecto está formada por un sistema híbrido, ya que combina partes distribuidas y centralizadas. Para su implementación se ha escogido el sistema *middleware CORBA,* junto al lenguaje de programación Java. A través de los servicios de *CORBA,* se han conseguido desarrollar fácilmente los agentes de la red *Peer to Peer.*

CORBA (Common *Object Request Broker Architecture*) es una de las tecnologías *middleware* de mayor difusión. Da soporte a aplicaciones con las siguientes características:

- **Distribución:**
  - Los objetos pueden existir en diferentes ordenadores

- **Heterogeneidad:**
  - Los ordenadores pueden estar ejecutando sistemas operativos diferentes
  - Los objetos pueden estar implementados en diferentes lenguajes de programación, en nuestro caso Java.

3.2 Elementos del Sistema

Como hemos dicho anteriormente, la aplicación desarrollada en el proyecto está formada por un sistema híbrido, ya que combina elementos distribuidos y centralizados. Los elementos centralizados del sistema son el servidor de nombres de CORBA y el servidor gestor y los elementos distribuidos son los agentes de la red.

3.2.1 SERVIDOR DE NOMBRES DE CORBA (*Naming Service)*

El servidor de nombres es un servicio de búsqueda que permite localizar e invocar objetos persistentes en el entorno CORBA de un modo transparente.

Se ha utilizado el servidor de nombres *ORBD* (*Object Request Broker Daemon*). Se lanza en una sola máquina indicando la dirección IP de ésta y el puerto donde se lanza el servicio para su futura localización por parte de otros elementos del sistema:

orbd –ORBInitialPort NumeroDePuerto –ORBInitialHost NombreDelHost

3.2.2. SERVIDOR GESTOR

Es el servidor que contiene toda la información de la red referente a los agentes existentes en ella. Cuando un nuevo agente llega a la red, éste es registrado de inmediato en el gestor, y obtiene un identificador único en toda la red. El gestor tiene también la función de entregar información sobre la red a los agentes que lo pidan.

3.2.3. AGENTES

Un agente es un proceso que realiza básicamente tres funciones: pedir, servir y limitar el acceso a sus servicios por parte de otros agentes de la red. Para desarrollar todas estas funciones se dispone de una serie de mecanismos de control, los contratos, basados en un algoritmo conocido como *leaky bucket*. Éstos son negociados entre los distintos agentes registrados en un servidor.

Al establecer un contrato, el agente negociará los parámetros del *leaky bucket* y sólo transmitirá peticiones al ritmo establecido en dicho contrato. De este modo, cada agente se asegura no dedicar más ancho de banda del establecido, y así poder dedicar el resto a otros cometidos. Otra razón por la que se establecen contratos es la de evitar que los recursos de los agentes queden congestionados.

Para hacer una petición de un fichero, el agente utiliza información procedente del gestor para ver los agentes disponibles en la red y la identidad de los agentes que poseen el fichero deseado.

La acción de servir ficheros se realiza entre diversos agentes, es decir, cada agente que ha establecido un contrato con el agente que hizo la petición, irá transmitiendo distintos fragmentos hasta completar el archivo pedido. De este modo es posible obtener el archivo deseado en un menor periodo de tiempo.

3.3. Estructura del funcionamiento del sistema

Como se ha dicho anteriormente el sistema esta compuesto por varias partes como son el servidor de nombres, el servidor gestor y los agentes. Cuando un agente llegue a la red, tendrá que registrarse en el servidor gestor para obtener un identificador único y poder pedir y transmitir ficheros dentro de la red.

Si observamos la figura 3.3.1. se puede ver un pequeño resumen de cómo funciona la petición y transmisión de recursos (ficheros) en la red. Los pasos serían los siguientes:

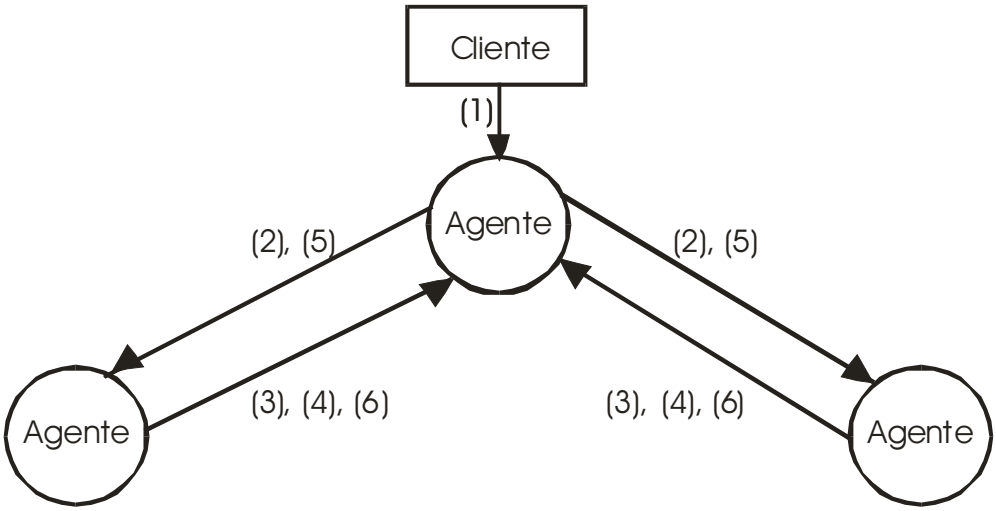


FIGURA 3.3.1.- Funcionamiento del sistema

(1) Solicitud del fichero por parte del consumidor local

El agente local hace una petición del archivo que quiere obtener por medio de otros agentes remotos.

(2) Petición del contrato

El agente local busca a otros agentes que contengan el archivo deseado y establece un contrato con ellos.

(3) Respuesta de aceptación del contrato

Cada agente remoto en función de si posee el archivo pedido aceptará o rechazará el contrato.

(4) Entrega de recurso al agente local

Cada agente que haya aceptado un contrato con el agente local, le irá sirviendo partes del fichero pedido.

(5) Guardar recurso recibido

El agente local va recibiendo partes de fichero, y las va guardando. Al final obtendrá el fichero pedido, que será la unión de todas las partes recibidas.

(6) Ruptura del contrato

Al finalizar la transmisión del fichero, se romperán los contratos formalizados al inicio de la transmisión.

3.4. Sistema de establecimiento de contratos

El establecimiento de contratos entre los distintos agentes, significa la aceptación de una serie de condiciones por parte de dos agentes. En el caso del presente proyecto significa que el agente remoto posee el fichero pedido por el agente local y está dispuesto a compartirlo con él. Además, el hecho de establecer el contrato, condiciona la capacidad de la línea que se va a usar para la transmisión de datos entre los dos agentes. Esto quiere decir, que el agente remoto limitará el ancho de banda dedicado al agente remoto. Este objetivo se consigue aplicando el algoritmo *Leaky Bucket,* explicado en este capítulo.

Las condiciones de establecimiento de contrato podrían haber sido otras. Se podría jugar con variables y porcentajes obtenidos en transmisiones anteriores que condicionarían la aceptación o rechazo de dichos contratos. Es decir, un agente podría aceptar un contrato, si tiene el fichero que se pide y además el agente local que lo pide ha transmitido con anterioridad cierta cantidad de información. De este modo se puede tener un sistema diferente, condicionando la forma de actuar de ciertos nodos. Se podría incluso obtener un sistema con agentes que tengan "personalidades propias", es decir, que existan agentes "altruistas", cuya única condición a cumplir para aceptar el contrato sea la de poseer el fichero pedido. Un agente "egoísta" sería el que aceptara un contrato si el agente local cumple una sería de condiciones en su historial de transmisiones, etc.

3.5. Modelo de Consumo. Algoritmo Leaky Bucket

3.5.1 Introducción y definición

Leaky Bucket es un algoritmo de control de flujo. Es básicamente un contador. Consiste en un cubo al que le van llegando testigos a una velocidad de Cir testigos/segundo. La cantidad de testigos que hay en un momento dado es de Tc testigos, y su capacidad máxima es de Cbs testigos.

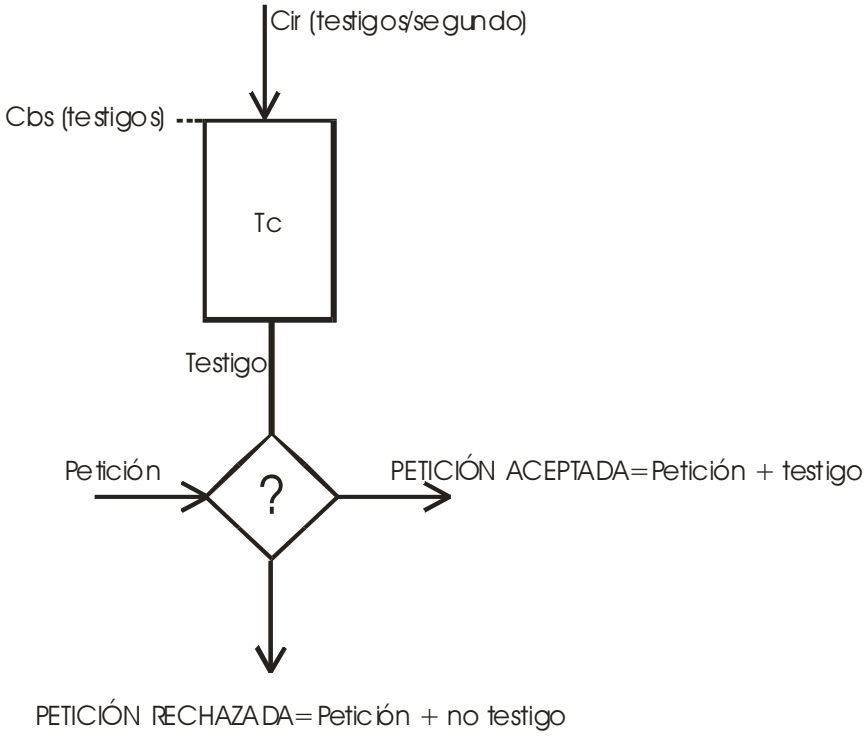


FIGURA A2.1.- Leaky Bucket

Al este sistema de control de flujo van llegado peticiones. Cuando hay una petición y en el cubo hay un testigo disponible, la petición será marcada como aceptada, en caso contrario, será marcada como rechazada. [Figura A2.1]

Con *Leaky Bucket*, se ha conseguido limitar el ancho de banda que un agente dedica a servir peticiones remotas. Se fija una tasa máxima, y por encima de ella se rechazan las peticiones.

3.5.2. Modo de calcular los parámetros según el porcentaje de la capacidad que se quiera compartir

Dada la capacidad máxima de una línea, se puede configurar el *Leaky Bucket* de modo que opere utilizando una capacidad menor, es decir, haciendo que el sistema dedique menos ancho de banda para ciertas actividades.

Para calcular los parámetros utilizamos la siguiente fórmula:

$$\mathbf{Cir = C \cdot \alpha / T}$$

- **Cir** es la velocidad a la que se generan los testigos, se mide en testigos por segundo.
- **C** es la capacidad máxima del sistema
- **α** es el porcentaje que vamos a usar de la red
- **T** es el tamaño de paquete usado

Para esta operación, hay que tener en cuenta el cambio de unidades siguiente:

- 1 Kbit = 1024 bits
- 1 Mbit = 1024 Kbits

A continuación se muestra un ejemplo, donde buscamos un 20% de la capacidad de la red. Los datos son los siguientes:

- C = 4 Mbps = 4 · 1024 = 4096 kbps
- $\alpha = 0.20$
- T = 64 Kbits

Tras operar obtenemos un Cir de 12.8 testigos por segundo, es decir, el *Leaky Bucket* tiene que generar un testigo cada 1/12.8 segundos (0.078125 segundos).

CAPÍTULO 4

DESARROLLO

4.1-Introducción

Antes de proceder a la explicación de la implementación de los archivos del proyecto, hay que decir que éste está compuesto por diversos archivos que se describen brevemente a continuación:

- ***gestor.idl:*** Es el interfaz CORBA. En este archivo es donde se especifican todas las operaciones y los tipos que soportan las aplicaciones principales del proyecto, es decir, el gestor y los agentes. (Ver Glosario [G.4.1], [G.4.2]).

- ***ServGestor.java:*** Es un programa que actúa como un servidor. A través del éste se van registrando los distintos agentes de la red, y se pueden consultar los que existan.

- ***Lb.java:*** Es una aplicación que implementa el algoritmo *Leaky Bucket*. Se usa en los agentes (Ver capítulo 3).

- ***Uno.java:*** Es el programa que representa la parte gráfica de los agentes. A través de éste, un agente puede hacer peticiones de archivos.

- ***ServAgente.java:*** Es un programa que actúa a la vez como cliente y servidor. En la parte cliente, se comunica con el gestor para registrar al agente que define. Como servidor consta de varios métodos para poder comunicarse con otros agentes y negociar peticiones de ficheros.

4.2.- Implementación

4.2.1.- Implementación de "*gestor.idl*"

El archivo *"gestor.idl"* es donde se especifican todas las operaciones y tipos que soportan ciertos objetos. Contiene dos interfaces, una para el gestor denominada "interfazgestor" y otra para los agentes "interfazagente". En cada interfaz se definen los métodos y tipos de variables que éstos utilizan, que se explicarán a continuación, en los apartados 4.2.2 y 4.3.3.

4.2.2.- Implementación de "*ServGestor.java*"

El gestor se encarga de dar de alta los diferentes agentes que entran en la red, y de proporcionar información sobre éstos a otros agentes.

"ServGestor.java" consta de dos clases, "gestorImplementation" donde se desarrollan los métodos del gestor, y "SerGestor" donde se inicializan y se crean objetos para poder utilizar CORBA y donde se escucharán las futuras peticiones de algún cliente, es decir, de algún agente que se quiera registrar o pedir información.

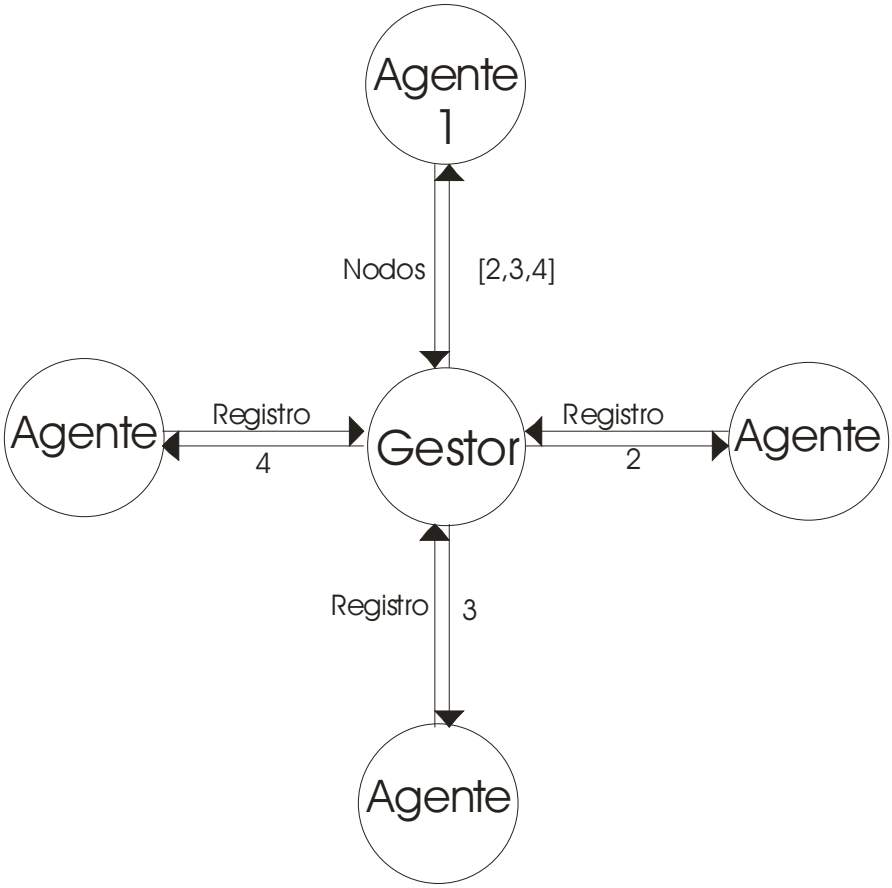


FIGURA 4.2.2.1.- Funcionamiento del Gestor

Dentro de la clase "gestorImplementation" tenemos dos métodos, que estaban definidos en la interfaz denominada "interfazgestor" de la siguiente manera:

```
interface interfazgestor {
    long Registro(in interfazagente identidad);
        info Nodos();
};
```

Estos métodos serán invocados por los agentes, que actúan como clientes de este programa servidor, el gestor. Explicado gráficamente se podría ver observando la FIGURA 4.2.2.1.

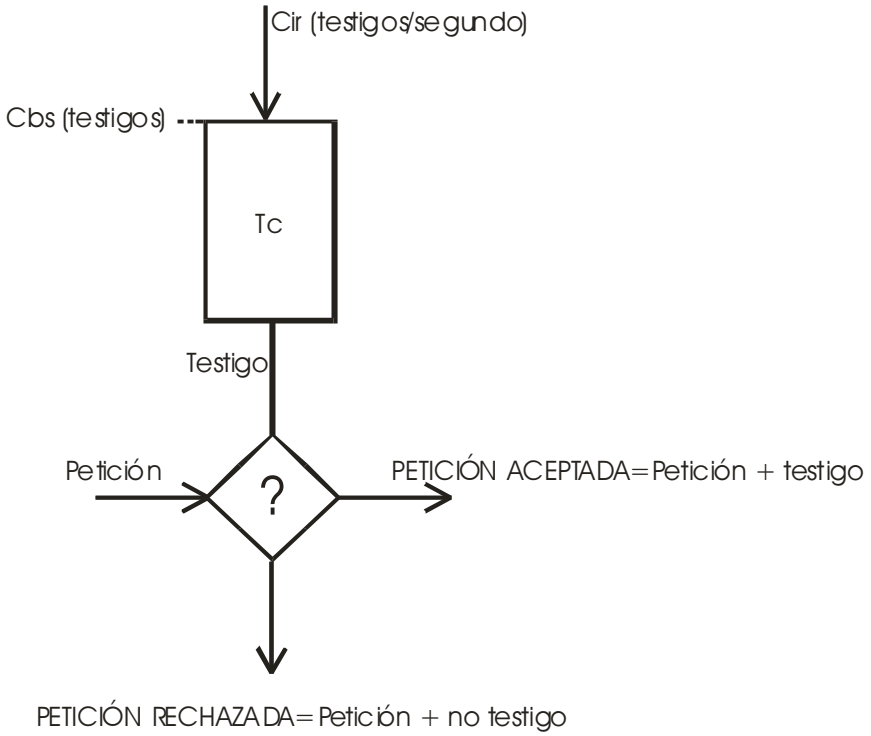


FIGURA A2.1.- Leaky Bucket

- **public int Registro(interfazagente ident):** Este método se usa para registrar agentes nuevos en la red con un identificador único. Como parámetro de entrada se le pasa un objeto de tipo interfazagente.
Al llamar a este método, se va recorriendo un *array* que será donde se irán guardando los distintos agentes de la red. Por defecto se ha definido que el número máximo de agentes en la red será de 10, por lo tanto, si entra un agente nuevo, y existe una posición vacía dentro del array, se procederá a registrar al agente, y se devolverá un entero con su identificador.

- **public informacion[] Nodos():** Este método devuelve un array con los agentes que hay registrados en el gestor. Es el mismo array que se recorre en el método anterior a la hora de hacer el registro.

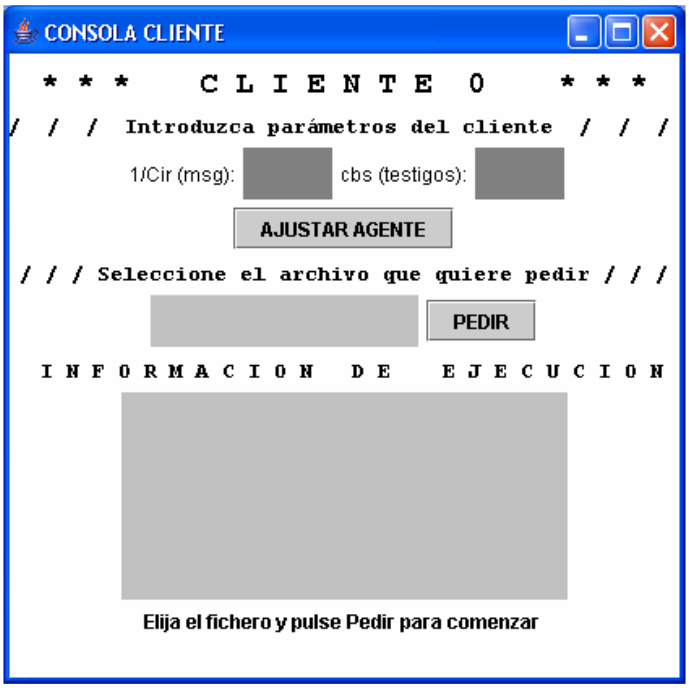


FIGURA 4.2.4.1.- Consola de un agente

4.2.3.- Implementación de "*Lb.java*".

Como se ha comentado antes, este archivo contiene la implementación del algoritmo de control de flujo *Leaky Bucket* (Ver capítulo 3).

"Lb.java" se ha realizado de tal modo que cuando creemos un objeto de este tipo le tengamos que pasar como argumentos dos parámetros imprescindibles para su funcionamiento, **cir** y **cbs**, que se corresponden con el número de testigos que se generaran por segundo, y el numero máximo de testigos que puede contener el cubo (*Leaky Bucket*). A la hora de crear el programa se ha decidido que cuando se tenga que introducir el valor de cir, se indique directamente el valor de 1/cir en milisegundos.

Consta de tres métodos:

- **public void go():** Este método es el más importante y sirve para que el *Leaky Bucket* comience a funcionar. Lo que hace es lanzar un *thread* que cada 1/cir milisegundos genere un testigo nuevo si el tamaño del cubo es inferior al limite máximo establecido al crear el *Leaky Bucket*.

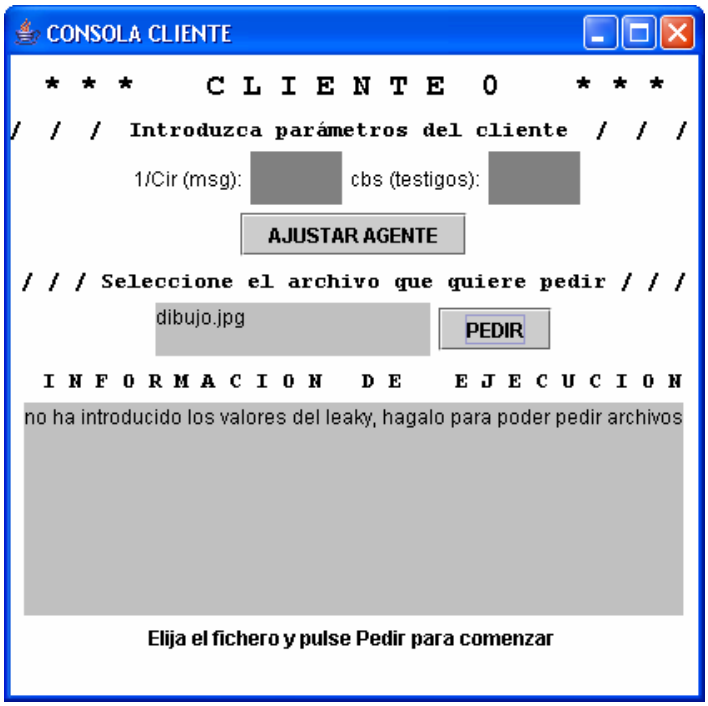


FIGURA 4.2.4.2.- Consola de un agente sin parámetros

- **public void info():** Este método simplemente indica información sobre los parámetros **cir** y **cbs** por pantalla.

- **public bolean Dispon():** Con este método se puede conocer si hay o no testigos en el Leaky Bucket en el momento de realizar una petición. Por tanto, si se hace una llamada a Dispon() y devuelve true, significará que había un testigo disponible, y que se puede cursar una petición. En caso contrario, no se podría cursar nada, sería un indicio de la limitación de ancho de banda de un agente, marcado previamente en los parámetros del *Leaky Bucket*. Para entender la implementación realizada observar la figura [FIGURA A2.1.]

4.2.4.- Implementación de "***Uno.java***"

Este fichero contiene la implementación del entorno gráfico de la aplicación desarrollada en el proyecto. Se trata de una consola donde se seleccionan tanto los parámetros del *Leaky Bucket* de cada agente, como el fichero que se quiere pedir.

En cada ordenador donde se ejecute el programa, habrá una consola de este tipo, denominada cliente, mediante la cual se podrán ir viendo los resultados obtenidos de las ejecuciones realizadas.

Al lanzar un agente a la red, éste se registra en el gestor, obteniendo un identificador único. Seguidamente pide información sobre los agentes de la red, y llama al programa "Uno.java" pasándole como argumentos el identificador, el *array* con información sobre los agentes, y un objeto del tipo del gestor. Se puede observar entonces que se abre una consola, *o frame*, (el cliente) donde el usuario podrá seleccionar todo lo que se ha comentado anteriormente [FIGURA 4.2.4.1.]

Este programa esta diseñado de modo que lo primero que se debe introducir para poder seguir adelante, son los valores de configuración de *Leaky Bucket*. Es decir, el valor de **cir,** que se introduce el valor de 1/cir en milisegundos, que se corresponde con el tiempo que tiene que pasar para que se genere un testigo. También hay que introducir el valor de **cbs,** que se corresponde con el número máximo de testigos en el cubo del *Leaky Bucket*.

Por lo tanto, si no se introducen estos parámetros, y se procede a hacer una petición, se anunciará que antes se deben cumplimentar los datos anteriormente indicados (Ver [FIGURA 4.2.4.2.]).

Una vez introducidos los parámetros, el cliente puede proceder a escribir el nombre del archivo que desea obtener de otros agentes. Hay que tener en cuenta que una vez establecidos los parámetros no van a poder ser cambiados.

Cuando se ha hecho la petición del archivo, en la implementación el siguiente paso ha sido buscar datos actualizados de los agentes que hay en la red, utilizando para ello el objeto de tipo gestor que se pasa como parámetro al llamar a *Uno*. Finalmente se llama a la función del agente denominada *Pedir* y de este modo comenzarán los trámites para la transferencia del archivo.

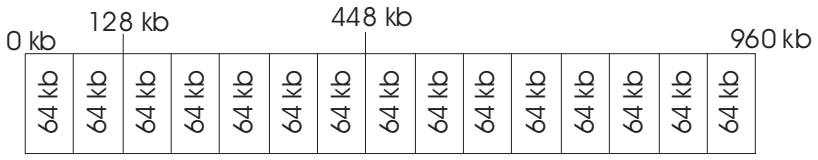


FIGURA 4.2.5.1.- División de un archivo en partes

4.2.5.- Implementación de "***ServAgente.java***"

"ServAgente.java" realiza dos funciones principalmente, una sería la de registrarse en el gestor al ser lanzado, es decir, actuar como un cliente del servidor gestor. Esta implementación está realizada en la clase *"ServiAgente"*, que también es donde se inicializan y se crean objetos para poder utilizar CORBA. La otra función sería la de compartir un fichero seleccionado por el usuario implementada en la clase "*agenteimplementation"*.

Cada ordenador tiene una consola desde la cual el usuario puede pedir un archivo a través de su agente (local). Al realizar la petición, el agente local se pone en contacto con otros agentes de la red que contengan el archivo deseado, y establece unas negociaciones con éstos para que se lo compartan entre todos. El resultado es el archivo deseado, compuesto por diversos trozos, procedentes de diversos agentes.

En la figura 4.2.5.1 podemos ver un archivo de 960 Kb, dividido en fragmentos de 64 Kb. (Ver figura [FIGURA 4.2.5.1.]).

Para entender bien el funcionamiento de *"ServAgente.java"* se puede observar el siguiente esquema:

1. El agente local pide un archivo utilizando el método *Pedir*

2. El agente local busca a otros agentes que contengan el archivo deseado, y establece un contrato con ellos mediante el método *PedirRemoto*

3. Una vez que ha obtenido contratos de X agentes (como máximo hemos establecido cinco), hace una petición a uno de ellos con el método *Tam*, para así establecer el tamaño del archivo destino.

4. El archivo a compartir se divide en trozos de un tamaño fijo. Se informa de la cantidad de partes con el método *DefineBloque*

5. Se lanza un hilo por cada agente con contrato que exista. Este hilo se encargará de realizar peticiones para ir tramitando todo el archivo del siguiente modo:

   a. Se pide el identificador de un trozo de fichero que no se haya obtenido ya (método *PedirBloque)*. Este trozo puede ser o no el último, si lo es la única diferencia que existirá, es que el tamaño puede ser inferior al de un trozo normal.

   b. El agente remoto leerla posición indicada en su archivo, y devuelve la información obtenida (método *ConsumirRemto)*.

   c. El agente local mediante el método *Escribir* guarda dicha información en la misma posición en el archivo destino.

   d. Se llama al método *Contar* para saber cuantas peticiones se han realizado

   e. A partir de aquí se vuelve a proceder de igual modo que en el punto a, tan solo que puede suceder que no queden más trozos de fichero porque ya se hayan transmitido todos. En este momento, se llama al método *Completo* para concluir con la transmisión del archivo, es decir, se verifica que se ha completado la transmisión sin errores, se cierra el archivo destino, se rompen los contratos con los agentes remotos y se devuelve información del proceso.

A continuación se procede a la explicación de los métodos que dispone *"ServAgente.java"* dentro de la clase *"agenteimplementation"* . Los métodos están previamente definidos en la interfaz idl de CORBA denominada *interfazagente* de la siguiente manera:

interface interfazagente {

long long ***Tam****(in string fileorigen);*

boolean ***LeakyB****(in long numero,in long icir, in long icbs);*

largo* ***Pedir**(in long numero, in info infor, in string fileorigen, in string agconectados);*

void ***DefineBloque****(in long trozos);*

*long **PedirBloque**();*

*long **Bloques**();*

*boolean **PedirRemoto**(in long numeroLocal,in long numeroRemoto, in string archivo);*

*void **RomperContrato**(in long numeroLocal,in long numeroRemoto);*

*datos **ConsumirRemoto**(in long numLocal, in long numRemoto, in long long*

apuntador, in string Rin, in long tambuf);

*boolean **Completo**();*

*void **Escribir**(in long long apuntador, in OctetArray buf);*

*void **Contar**(in long ie);*

};

* largo es un tipo de dato definido en el interfaz

- **public int[] Pedir(int numero, informacion[] infor, String fileorigen, String filedestino)**

Este es el método principal, a partir del cual se le hacen las llamadas a los otros métodos y se inicia el proceso de compartición.

Es llamado desde la consola del cliente para hacer la petición del archivo deseado y devuelve cierta información sobre la petición como el tamaño del archivo pedido, el número de partes en las que se divide, etc.

Comienza buscando agentes que contengan el archivo pedido e intentando establecer un contrato con ellos mediante el método *PedirRemoto.* Un contrato no es más que un acuerdo establecido entre el agente local y remoto para poder compartir temporalmente información.

El siguiente paso consiste en definir el tamaño del archivo destino. Para ello se pregunta a un agente remoto el tamaño con el método *Tam*. Esto se hace evitar solapamiento de información cuando se estén guardando las diferentes partes del archivo.

Una vez que se sabe ya el tamaño del archivo, se procede a calcular el número de secciones en las que hay que dividirlo. Se llama al método *DefineBloque* para definir el puntero de inicio de cada trozo.

Por último en este método, se lanzan tantos hilos como contratos establecidos haya, pidiendo las peticiones a los distintos agentes remotos.

- **public boolean LeakyB(int nume, int icir, int icbs)**

Este método se utiliza para definir los parámetros del algoritmo *Leaky Bucket* introducidos en la consola por el usuario. Devolverá el valor booleano *true*, si ambos parámetros son correctos.

- **public void DefineBloque(int trozos)**

El objetivo de este método, es el de definir los punteros de cada trozo de información en los que va a estar dividido el archivo pedido. Consta de tres *arrays* del tamaño del número de segmentos obtenidos.

El *array* partes tiene en cada posición el número del bit por el cual comienza cada trozo. Si cada trozo es de tamaño X, la primera posición empezará en 0, la segunda en 1·X, la tercera en 2·X y así consecutivamente.

Los otros dos *arrays* booleanos, usado y pedido, nos sirven en otros métodos para marcar si un trozo ha sido pedido ya.

- **public synchronized int PedirBloque()**

Este método es llamado por cada agente remoto cada vez que quiere hacer una petición. Devuelve el identificador del puntero que tiene que pedir cada agente al hacer la petición. Es un método sincronizado, para que sólo un agente a la vez pueda pedir dicha información y así evitar los duplicados.

- **public synchronized int Bloques()**

La función de este método es contar el número de peticiones que quedan por hacer. Sirve principalmente para informar de cuando es la última petición que suele ser de un tamaño distinto a las otras peticiones.

- **public boolean PedirRemoto(int numeroLocal,int numeroRemoto, String archivo)**

Este método sirve para poder establecer contratos con agentes remotos. Lo primero que hace es comprobar que el agente remoto tiene el archivo pedido por el usuario. Si lo tiene, entonces se procede a aceptar el contrato, y a lanzar el *Leaky Bucket* para que empiece a funcionar. Esto sería la condición que impone el agente remoto, pues con el *Leaky Bucket*, le está limitando la capacidad de la línea que le va a dedicar al agente local para la transmisión de la información.

- **public void RomperContrato(int numeroLocal,int numeroRemoto)**

Con este método simplemente se rompe el contrato establecido entre el agente local y el remoto.

- **public long Tam(String fileorigen)**

Con esta función se obtiene el tamaño del archivo que se esta pidiendo. Sirve para definir el tamaño del archivo destino, para que así a la hora de ir guardando la información que se recibe no haya solapamientos.

- **public synchronized byte[] ConsumirRemoto(int numeroLocal, int numeroRemoto, long apuntador, String archivo, int tambuf)**

En este método cada agente remoto accede al archivo solicitado, lee una la parte de información que le corresponde transmitir, y la envía al agente local.

Como se ha dicho anteriormente, cada petición va condicionada la tasa de transmisión de *Leaky Bucket*, y nunca se transmitirá más rápido de lo permitido por el agente remoto.

- **public synchronized boolean Completo()**

Es un método para finalizar la petición del archivo y proporcionar información sobre como ha ido la transmisión. Este método es llamado cada vez que se transmite una parte de la información, por ello tiene un contador que sirve para controlar el número de trozos que han sido transmitidas.

Cuando se llega al límite máximo de trozos, se cierra el archivo que se creó y abrió para guardar los datos que se iban recibiendo de los distintos agentes remotos. A partir de ese momento se considera la transferencia terminada y se devuelve información a cerca de todo el proceso, como el tiempo que se ha tardado, porcentaje transmitido por cada agente, etc. También se rompen todos los contratos existentes con el agente local.

- **public synchronized void Escribir(long apuntador, byte[] b)**

Este es el método a partir del cual se guardan las partes de archivo recibidos de los agentes remotos, en el archivo destino. Es un método sincronizado para evitar solapamiento de la información. De este modo solo un hilo cada vez puede escribir en el fichero. Cuando se recibe una llamada a este método, se posiciona el puntero de escritura en la posición indicada, y se vuelcan los datos recibidos en esa posición, obteniendo poco a poco el archivo completo.

- **public void Contar(int i)**

Esta función es utilizada para llevar un control de los trozos que transmite cada agente. Es llamada por el método Completo para poder luego dar información y estadísticas de todo el proceso de transmisión del archivo.

CAPÍTULO 5

PRUEBAS

5.1-Introducción

A la hora de realizar las pruebas del proyecto, se han montado varios escenarios para poder comprobar de este modo, distintas características del mismo. En todo momento se han hecho las pruebas en una red de ordenadores y bajo el sistema operativo *Linux*.

5.2.- Escenarios

5.2.1.-ESCENARIO 1

Tenemos un sistema formado por dos ordenadores conectados entre si por sus tarjetas de red. La velocidad de la red que forman es de 100 Mbps. En este escenario se simula el acceso a Internet, mediante un *servidor ISP*.(Ver Glosario [G5.1]).

Un cliente pedirá un archivo a su agente, el cual se pondrá en contacto con otro agente de la red, y le transmitirá el archivo al cliente como si estuviera conectado a Internet. (Ver figura [FIGURA 4.2.1.1]).

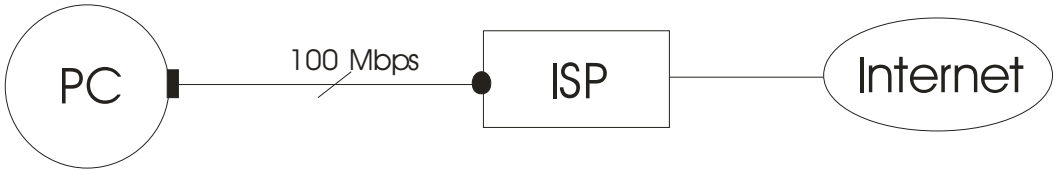


FIGURA 5.2.1.1.- Escenario 1

Se han configurado los parámetros del *Leaky Bucket* de tal modo que el agente que simula ser el servidor ISP curse todas las peticiones que le lleguen, es decir, que no ofrezca ninguna restricción a la hora de usar su ancho de banda con otros agentes. Estos parámetros son: cir=10^6 testigos/sg, cbs=10^8 testigos. El tamaño de cada paquete transmitido es de 64 Kbits.

Se han realizado varias pruebas con este modelo, haciendo peticiones de varios archivos de distintos tamaños, y los resultados que hemos obtenido han sido los que se pueden observar en la tabla [Tabla 5.2.1.1] y en la gráfica [Gráfica 5.2.1.1].

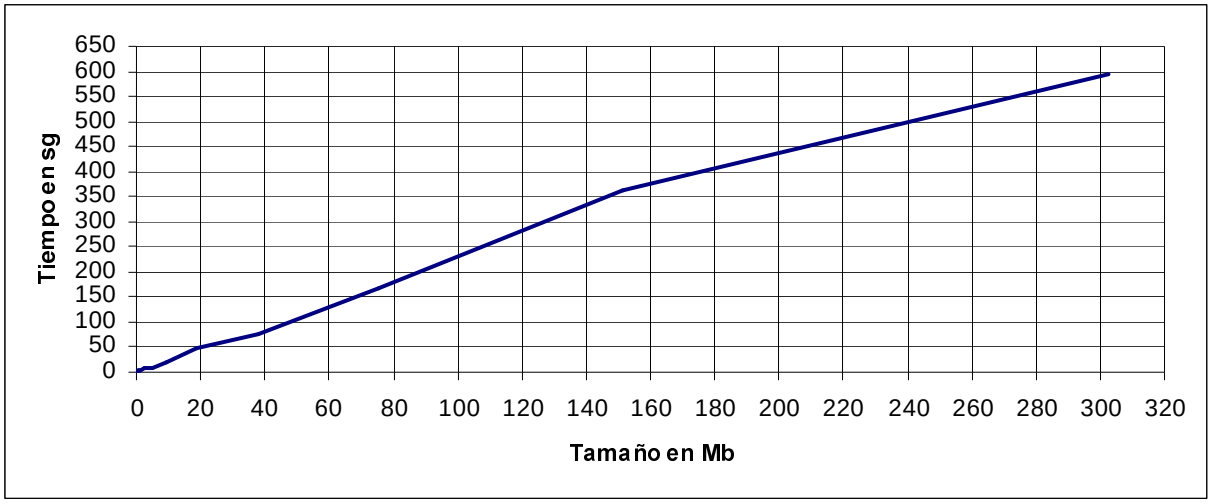
La conclusión más inmediata que se puede obtener de los datos obtenidos, es que la velocidad máxima de transferencia de la red es de 4 Mbps, un valor muy bajo, si se compara con los 100 Mbps que definen a la red donde se han realizados dichas pruebas. Los motivos de este valor no están suficientemente claros, puede deberse a que la red esta siendo utilizada para otras aplicaciones y por otros usuarios en el laboratorio. También puede deberse a alguna característica del funcionamiento de CORBA. (Ver gráfica [Gráfica 5.2.1.2])

Por tanto, a partir de los resultados obtenidos en este escenario, se tomará la velocidad de transferencia máxima de la red como 4 Mbps.

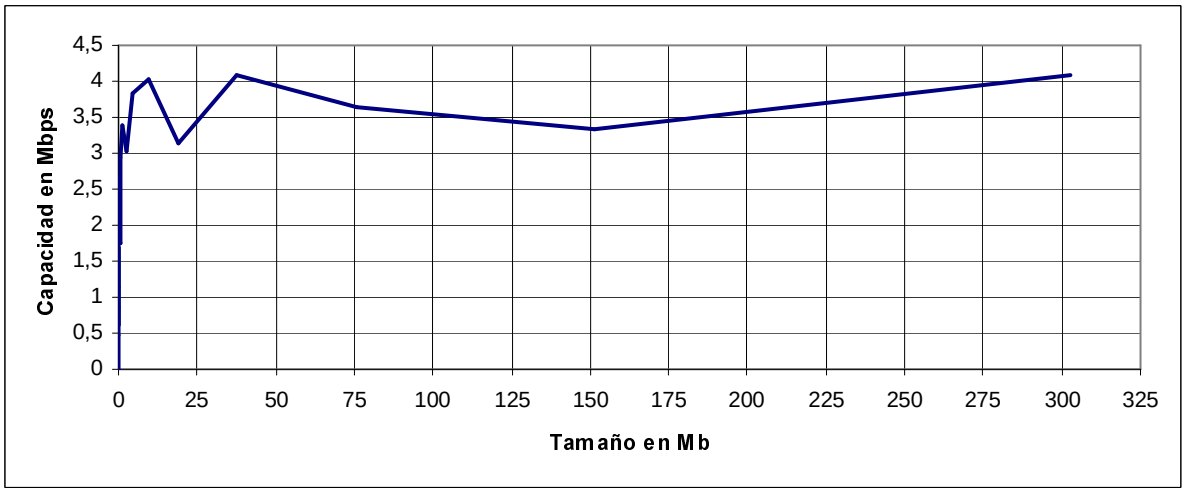
Numero de Agentes	Tamaño Archivo	Trozos de 64Kb	Tiempo (sg)	Capacidad (Mbps)
2 (sólo 1 sirve)	155 bytes	1	0.128	0.009
2 (sólo 1 sirve)	12359 bytes	1	0.198	0.470
2 (sólo 1 sirve)	17693 bytes	1	0.08	1.687
2 (sólo 1 sirve)	40960 bytes	1	0.359	0.8704
2 (sólo 1 sirve)	61440 bytes	1	0.161	0.621
2 (sólo 1 sirve)	112640 bytes	2	0.754	1.139
2 (sólo 1 sirve)	163840 bytes	3	0.469	2.665
2 (sólo 1 sirve)	286720 bytes	5	1.024	1.8168
2 (sólo 1 sirve)	460800 bytes	8	1.286	1.7369
2 (sólo 1 sirve)	757760 bytes	12	2.028	2.85
2 (sólo 1 sirve)	1228800 bytes	19	2.757	3.4
2 (sólo 1 sirve)	2467840 bytes	38	6.218	3.028
2 (sólo 1 sirve)	4945920 bytes	76	9.826	3.84
2 (sólo 1 sirve)	9902080 bytes	152	18.75	4.02
2 (sólo 1 sirve)	19814400 bytes	303	47.885	3.15
2 (sólo 1 sirve)	39639040 bytes	605	73.881	4.09
2 (sólo 1 sirve)	79288320 bytes	1210	166.231	3.639
2 (sólo 1 sirve)	158586880 bytes	2420	361.921	3.34
2 (sólo 1 sirve)	317184000 bytes	4840	592.854	4.08

Tabla 5.2.1.1.- Resultado de tiempos del escenario 1

A continuación podemos ver dos gráficas con el comportamiento del escenario uno:



Gráfica 5.2.1.1.- Resultado de tiempos de transmisión de ficheros de distintos tamaños procedentes de la Tabla 5.2.1.1. Modelo 1



Gráfica 5.2.1.2.- Capacidad usada de la línea

5.2.2.-ESCENARIO 2

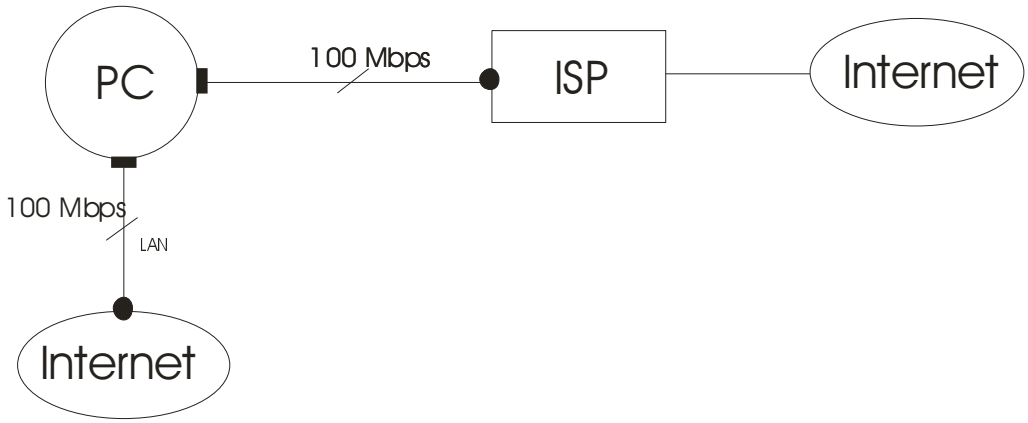


FIGURA 5.2.2.1.- Escenario 2

Tenemos un sistema formado por tres ordenadores conectados entre si por sus tarjetas de red. Se han montado dos redes, de tal modo que el ordenador que hace las peticiones dispone de dos tarjetas de red, y los ordenadores que cursan las peticiones de archivos disponen de una única tarjeta de red. La velocidad de las redes que forman es de 100 Mbps pero dados los resultados obtenidos en el escenario anterior se tomará como 4 Mbps.

Un cliente pedirá un archivo a su agente, el cual se pondrá en contacto con otros agentes de la red. Uno de los agentes simulará ser un servidor ISP (como en el escenario anterior) y otro será un ordenador de una red de área local, (LAN) con conexión a Internet. (Ver figura [Figura 4.2.2.1]).

Capacidad Máx (Mbps)	%	Cir (testigos/segundo)	1/Cir (milisegundos)
4	**100**	**64**	**15**
4	**50**	**32**	**31**
4	**30**	**19.2**	**52**
4	**20**	**12.8**	**78**
4	**10**	**6.4**	**156**
4	**5**	**3.2**	**312**
4	**4**	**2.56**	**390**
4	**3**	**1.92**	**520**
4	**2**	**1.28**	**781**
4	**1**	**0.64**	**1562**

Tabla 5.2.2.1.- Parámetros del Leaky Bucket para distintos porcentajes de la capacidad de la línea

En este escenario se han configurado los parámetros del *Leaky Bucket* de varios modos. El agente que actúa como servidor ISP tiene configurado el *Leaky Bucket* de modo que curse todas las peticiones que le lleguen, es decir, que no ofrezca ninguna restricción a la hora de usar su ancho de banda con otros agentes (100% de su capacidad). Estos parámetros son: cir=64 testigos/sg, cbs=64 testigos.

El otro agente se irá configurando de varios modos en las diferentes pruebas, de tal modo que opere con distintos porcentajes de su capacidad total. [Tabla 5.2.2.1 – Parámetros del *Leaky Bucket* para distintos porcentajes de la capacidad]. Para ver como se han calculado estos parámetros ver capítulo 3. El tamaño de cada paquete transmitido es de 64 Kbits.

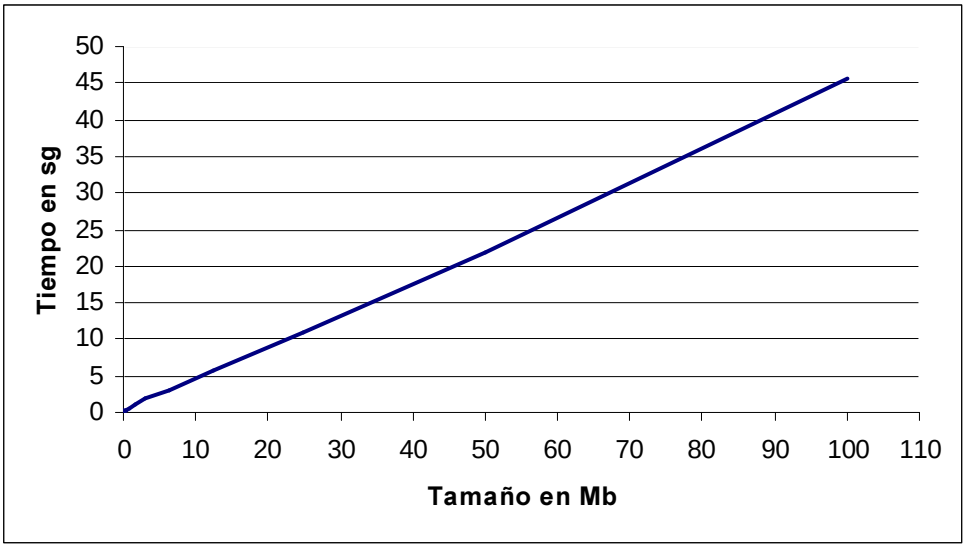
Agentes	Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A2 ¿rech LB?	Tiempo (sg)
3	17693	1	1 N	- N	0,117
3	40960	1	1 N	-N	0,122
3	92160	2	1 N	1 N	0,223
3	194560	3	2 N	1 N	0,298
3	399360	7	4 N	3 S	0,308
3	808960	13	8 N	5 S	0,528
3	1628160	25	19 N	6 S	0,958
3	3266560	50	40 N	10 S	1,833
3	6543360	100	86 N	14 S	2,881
3	13096960	200	171N	29 S	5,686
3	26204160	400	348 N	52 S	10,81
3	52418560	800	680 N	120 S	21,785
3	104847360	1600	1331 N	269 S	45,667

Tabla 5.2.2.2.- Resultado de tiempos del escenario 2

Agente 1 (A1) : C =100%, Cbs =64 testigos, Cir =64 testigos/segundo, 1/Cir=15 milisegundos

Agente 2 (A2) : C =10%, Cbs =5 testigos, Cir =6.4 testigos/segundo, 1/Cir=156 milisegundos

Se puede observar en esta tabla que el *Leaky Bucket* del agente dos comienza a rechazar peticiones a partir de un tamaño superior a 194560 bytes teniendo la capacidad del canal configurada de tal modo que se curse al 10% de la capacidad de ésta. El cbs del agente dos está configurado para que haya 5 testigos como máximo dentro del cubo.



Gráfica 5.2.2.2.- Resultado de tiempos de transmisión de ficheros de distintos tamaños procedentes de la Tabla 5.2.2.2. Modelo 2

En esta gráfica podemos ver reflejados los datos obtenidos en la tabla 5.2.2.2. Podemos ver que el modelo sigue una actuación normal debido a la linealidad de la gráfica, es decir, a mayor tamaño de fichero, mayor tiempo de transmisión.

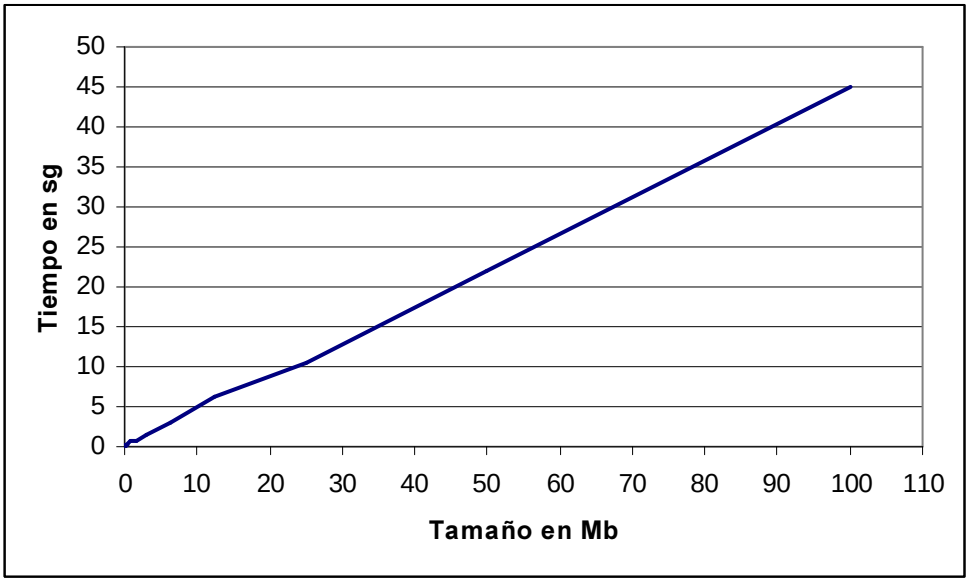
Agentes	Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A2 ¿rech LB?	Tiempo (sg)
3	17693	1	1 N	- N	0,231
3	40960	1	1 N	- N	0,084
3	92160	2	1 N	1 N	0,098
3	194560	3	2 N	1 N	0,209
3	399360	7	5 N	2 S	0,25
3	808960	13	7 N	6 S	0,668
3	1628160	25	20 N	5 S	0,702
3	3266560	50	42 N	8 S	1,542
3	6543360	100	84 N	16 S	2,958
3	13096960	200	168 N	32 S	6,196
3	26204160	400	348 N	52 S	10,396
3	52418560	800	679 N	121 S	22,009
3	104847360	1600	1347 N	253 S	45,086

Tabla 5.2.2.3.- Resultado de tiempos del escenario 2

Agente 1 (A1) : C =100%, Cbs =64 testigos, Cir =64 testigos/segundo, 1/Cir=15 milisegundos

Agente 2 (A2) : C =10%, Cbs 2= testigos, Cir =6.4 testigos/segundo, 1/Cir=156 milisegundos

Al igual que en la tabla anterior, vemos que el *Leaky Bucket* comienza a rechazar con peticiones mayores que 194560 bytes. En el agente dos, el cbs está configurado para que haya 2 testigos como máximo dentro del cubo, y para que la capacidad máxima de transmisión del canal sea del 10%.



Gráfica 5.2.2.3.- Resultado de tiempos de transmisión de ficheros de distintos tamaños procedentes de la Tabla 5.2.2.3. Modelo 2

Vemos que los resultados son similares a la tabla anterior, teniendo el mismo porcentaje de utilidad del canal y distintos tamaños del cubo 2 ahora y 5 en el caso anterior.

En esta gráfica podemos ver reflejados los datos obtenidos en la tabla 5.2.2.3. El comportamiento sigue siendo lineal.

Agentes	Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A2 ¿rech LB?	Tiempo (sg)
3	17693	1	1 N	0 N	0,072
3	40960	1	1 N	0 N	0,194
3	92160	2	1 N	1 N	0,089
3	194560	3	1 N	2 N	0,164
3	399360	7	5 N	2 N	0,332
3	808960	13	9 N	4 S	0,464
3	1628160	25	20 N	5 S	0,9
3	3266560	50	43 N	7 S	1,831
3	6543360	100	90 N	10 S	2,905
3	13096960	200	184 N	16 S	5,7
3	26204160	400	370 N	30 S	11,299
3	52418560	800	734 N	66 S	22,546
3	104847360	1600	1467 N	131 S	44,331

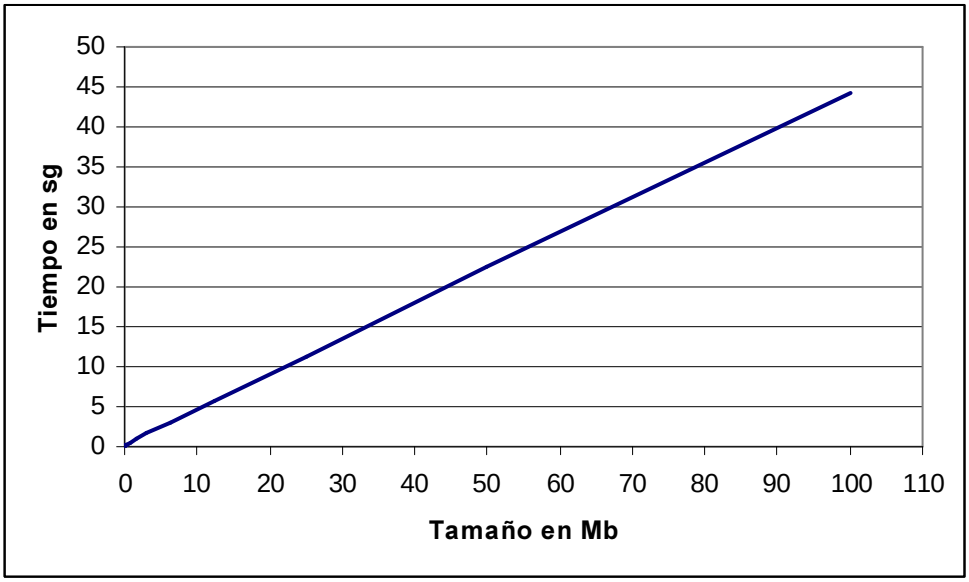
Tabla 5.2.2.4.- Resultado de tiempos del escenario 2

Agente 1 (A1) : C =100%, Cbs =64 testigos, Cir =64 testigos/segundo, 1/Cir=15 milisegundos

Agente 2 (A2) : C =5%, Cbs =2 testigos, Cir =3.2 testigos/segundo, 1/Cir=312 milisegundos

Los datos obtenidos en esta tabla se han conseguido configurando el agente dos de tal modo que se use el 5% del canal y con un cubo de tamaño máximo 2 testigos. En comparación con las tablas 5.2.1.2 y 5.2.1.3, en esta tabla, se obtienen tiempos de transferencia similares. Por lo tanto no hay diferencias a destacar entre tener un agente utilizando el 5% y el 10% de su capacidad.

.



Gráfica 5.2.2.4.- Resultado de tiempos de transmisión de ficheros de distintos tamaños procedentes de la Tabla 5.2.2.4. Modelo 2

En este caso el *Leaky Bucket* comienza a rechazar peticiones para ficheros de más de 7 trozos, para tamaños mayores que en las dos tablas anteriores que rechazaba a partir de ficheros mayores de 3 trozos.

En esta gráfica podemos ver reflejados los datos obtenidos en la tabla 5.2.2.4.

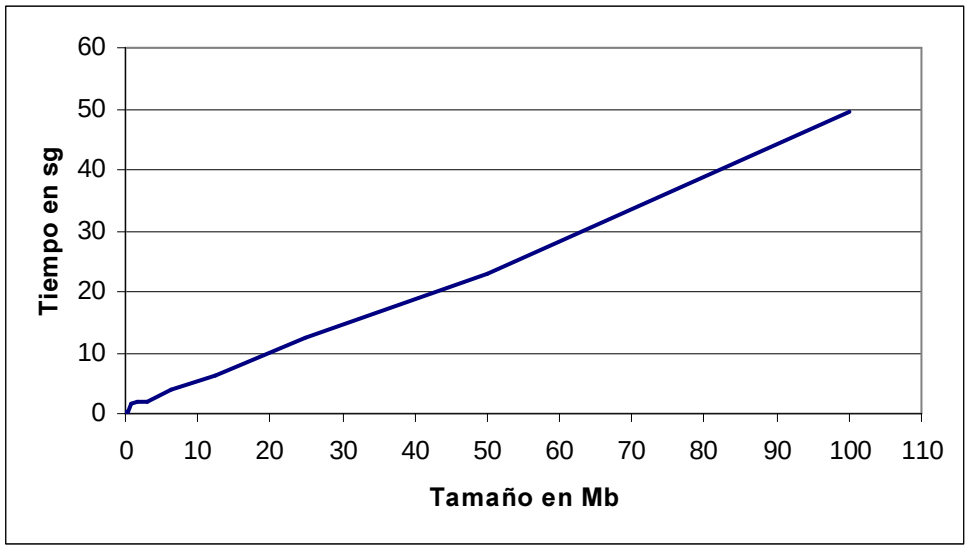
Agentes	Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A2 ¿rech LB?	Tiempo (sg)
3	17693	1	1 N	- N	0,083
3	40960	1	1 N	0 N	0,234
3	92160	2	1 N	1 N	0,093
3	194560	3	1 N	2 N	0,264
3	399360	7	3 N	4 N	0,152
3	808960	13	6 N	7 N	1,716
3	1628160	25	18 N	7 S	2,025
3	3266560	50	43 N	7 S	2,021
3	6543360	100	92 N	8 S	3,926
3	13096960	200	191 N	9 S	6,09
3	26204160	400	388 N	12 S	12,335
3	52418560	800	781 N	19S	22,954
3	104847360	1600	1569 N	31S	49,518

Tabla 5.2.2.5.- Resultado de tiempos del escenario 2

Agente 1 (A1) : C =100%, Cbs =64 testigos, Cir =64 testigos/segundo, 1/Cir=15 milisegundos

Agente 2 (A2) : C =1%, Cbs =5 testigos, Cir =0.64 testigos/segundo, 1/Cir=1562 milisegundos

En esta tabla se en el agente dos se cursa a un 1% de la capacidad del canal, con un cbs de tamaño 5. Los datos obtenidos son un poco más elevados en con porcentajes de 5% y de 10%. Como conclusión se puede obtener que con estos datos, cuanto mayor sea el tamaño de fichero a compartir, mayores serán las diferencias obtenidas entre los distintos porcentajes de canal.



Gráfica 5.2.2.5.- Resultado de tiempos de transmisión de ficheros de distintos tamaños procedentes de la Tabla 5.2.2.2. Modelo 2

Se puede observar en esta tabla que el *Leaky Bucket* del agente dos comienza a rechazar peticiones a partir de un tamaño superior a 13 trozos.

En la gráfica 5.2.2.5 podemos ver reflejados los datos obtenidos en la tabla 5.2.2.5.

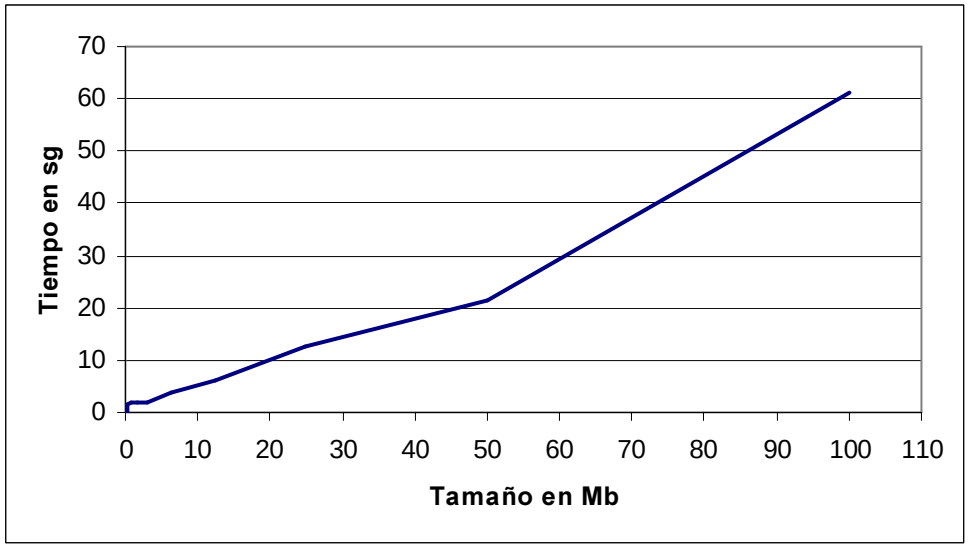
Agentes	Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A2 ¿rech LB?	Tiempo (sg)
3	17693	1	1 N	- N	0,117
3	40960	1	1 N	- N	0,126
3	92160	2	1 N	1 N	0,137
3	194560	3	2 N	1 N	0,183
3	399360	7	3 N	4 N	1,697
3	808960	13	9 N	4 N	1,792
3	1628160	25	21 N	4 N	1,875
3	3266560	50	46 N	4 N	1,961
3	6543360	100	95 N	5 N	3,883
3	13096960	200	194 N	6 N	6,008
3	26204160	400	390 N	10 N	12,803
3	52418560	800	785 N	15 S	21,259
3	104847360	1600	1562 N	38 S	61,152

Tabla 5.2.2.6.- Resultado de tiempos del escenario 2

Agente 1 (A1) : C =100%, Cbs =64 testigos, Cir =64 testigos/segundo, 1/Cir=15 milisegundos

Agente 2 (A2) : C =1%, Cbs =2 testigos, Cir =0.64 testigos/segundo, 1/Cir=1562 milisegundos

En esta última tabla, donde tenemos la misma capacidad que en el caso anterior 1 % de la capacidad del canal, vemos que los datos varían pero para los ficheros más grandes. En este caso, el tamaño del cbs es más pequeño, de tamaño 2 frente al tamaño 5 que se tenían en el caso anterior. Se puede ver, que al tener un cubo más pequeño se rechazan más peticiones y se obtienen tiempos mayores, pero sólo en archivos grandes.



Gráfica 5.2.2.6.- Resultado de tiempos de transmisión de ficheros de distintos tamaños procedentes de la Tabla 5.2.2.6. Modelo 2

En esta gráfica podemos ver reflejados los datos obtenidos en la tabla 5.2.2.6.

Como conclusión global de todos los casos del escenario dos, podemos decir que el tiempo varía siempre en mayor cantidad a la hora de transmitir el fichero más grande con el que se han hecho las pruebas (1600 trozos). En el resto de casos los tiempos suelen ser similares. Por lo tanto, el hecho de restringir la capacidad del canal afectará en mayor medida a ficheros grandes.

Sin embargo también se ha podido observar que al tener dos agentes, el tiempo de transmisión se ha reducido bastante.

5.2.2.-ESCENARIO 3

Este escenario es físicamente similar al escenario 2, tenemos un sistema formado por tres ordenadores conectados entre si por sus tarjetas de red. Se han montado dos redes, de tal modo que el ordenador que hace las peticiones dispone de dos tarjetas de red, y los ordenadores que cursan las peticiones de archivos disponen de una única tarjeta de red. La velocidad de las redes que forman es de 100 Mbps pero dados los resultados obtenidos en el escenario anterior se tomará como 4 Mbps.

La diferencia que existe entre los dos escenarios, es que este último va a tener una mayor cantidad de agentes.

El cliente situado en el Pc1 pedirá un archivo a su agente y éste se pondrá en contacto con otros agentes para que le sirvan. El PC2 tiene un agente que se corresponde con un ordenador de un red de área local, y el PC3 tendrá varios agentes. Como se puede observar (Ver figura [Figura 4.2.3.1]), en este último PC hay más de un agente, es decir, varios procesos agentes corriendo sobre la misma máquina. Esto simularía una red de área local con varios ordenadores que comparten un ancho de banda, en este caso, la capacidad total de este conjunto de agentes (PC3) sería de 100 Mbps.

Para la configuración de algunos parámetros del Leaky Bucket en este escenario hay que tener en cuenta una cosa, y es que la suma de porcentajes de las capacidades de los agentes del PC3 nunca podrá ser mayor de 100%. Esto se debe a que todos los agentes comparten una línea, y nunca se podrá superar la capacidad de ésta. Para la configuración del agente del PC1, vamos a poder poner los valores que deseemos según el uso de la capacidad que se quiera tener.

A la hora de tomar los datos en este escenario, se han ido variando los distintos porcentajes de la capacidad que se deseaba compartir (Ver tabla [Tabla 5.2.2.1 – Parámetros del *Leaky Bucket* para distintos porcentajes de la capacidad]).

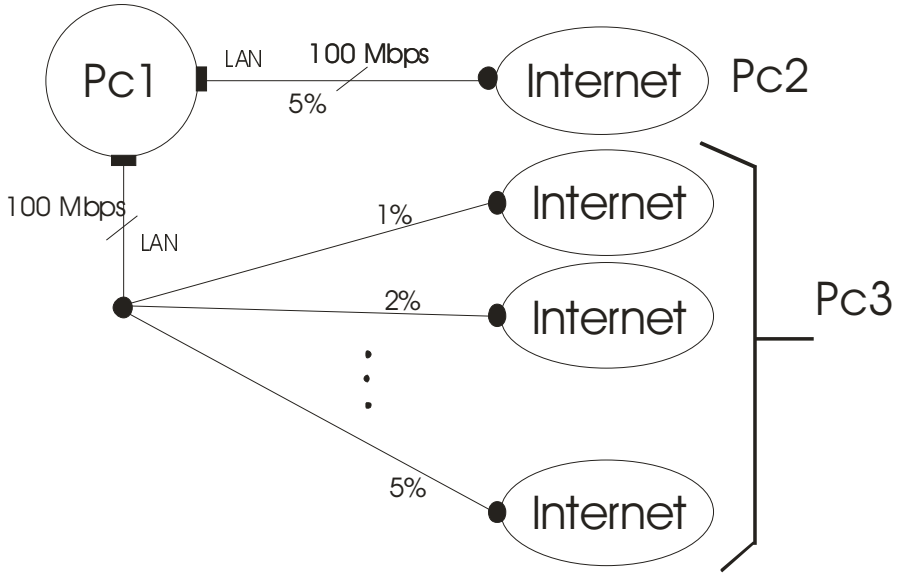


FIGURA 5.2.3.1.- Escenario 3

Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A22 ¿rech LB?	Tiempo (sg)
17693	1	1 N	- N	- N	- N	0,25
40960	1	1 N	- N	- N	- N	0,201
92160	2	1 N	1 N	- N	- N	0,28
194560	3	1 N	- N	1 N	1 N	0,255
399360	7	3 N	1 N	2 N	1 N	0,386
808960	13	4 N	4 S	3 N	2 N	4,202
1628160	25	13 S	4 S	4 S	4 S	5,47
3266560	50	28 S	8 S	7 S	7 S	7,204
6543360	100	50 S	20 S	17 S	13 S	11,006
13096960	200	96 S	43 S	34 S	27 S	19,125
26204160	400	184 S	89 S	72 S	55 S	32,374
52418560	800	374 S	176 S	142 S	108 S	59,987
104847360	1600	719 S	373 S	288 S	220 S	118,148

Tabla 5.2.3.1.- Resultado de tiempos del escenario 3

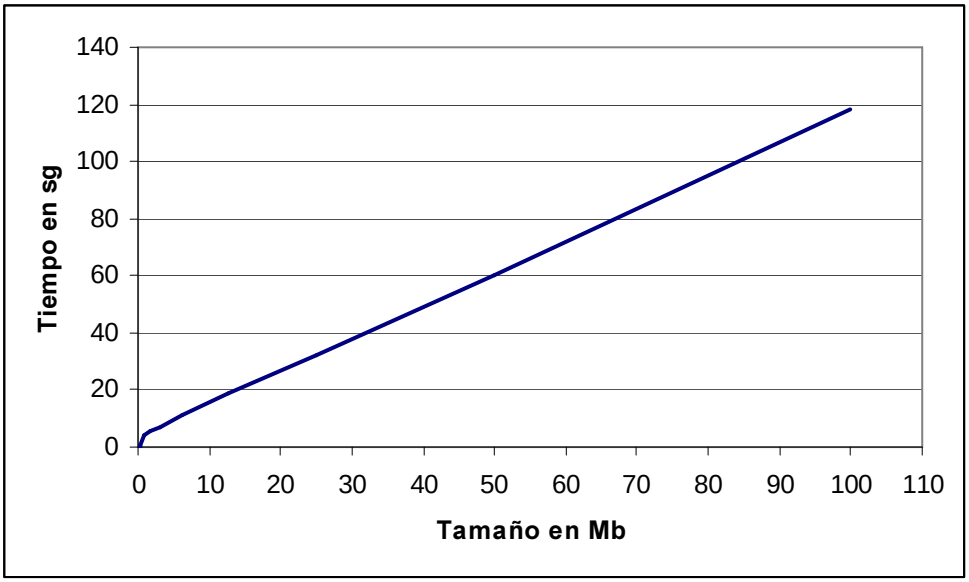
Agente 1 (A1) : C =10%, Cbs =5 testigos, Cir =6.4 testigos/segundo, 1/Cir=156 milisegundos

Agente 2.1 (A21) : C =5%, Cbs =2 testigos, Cir =3.2 testigos/segundo, 1/Cir=312 milisegundos

Agente 2.2 (A21) : C =4%, Cbs =2 testigos, Cir =2.56 testigos/segundo, 1/Cir=390 milisegundos

Agente 2.3 (A21) : C =3%, Cbs =2 testigos, Cir =1.92 testigos/segundo, 1/Cir=520 milisegundos

En esta tabla se puede observar que los tiempos son superiores a los del escenario dos, debido a que no se está utilizando al 100% la capacidad de ningún canal. La diferencia se empieza a notar para ficheros de tamaño mayores de 7 trozos. A partir de este tamaño el *Leaky Bucket* también empieza a rechazar en los distintos agentes.



Gráfica 5.2.3.1 Resultado de tiempos de transmisión de ficheros de distintos tamaños.

En esta gráfica podemos ver representados los datos de la tabla 5.2.3.1. Se puede apreciar que tiene un carácter lineal.

Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A22 ¿rech LB?	Tiempo (sg)
17693	1	- N	- N	- N	1 N	0,199
40960	1	1 N	- N	- N	- N	0,238
92160	2	1 N	- N	- N	1 N	0,254
194560	3	1 N	1 N	1 N	- N	0,437
399360	7	3 N	1 N	1 N	2 N	0,62
808960	13	4 S	3 N	3 N	3 N	2,625
1628160	25	10 S	5 S	5 S	5 S	5,199
3266560	50	19 S	11 S	10 S	10 S	8,959
6543360	100	31 S	28 S	23 S	18 S	12,773
13096960	200	62 S	56 S	46 S	36 S	22,531
26204160	400	121 S	115 S	93 S	71 S	40,54
52418560	800	236 S	234 S	188 S	142 S	77,687
104847360	1600	469 S	470 S	379 S	282 S	149,141

Tabla 5.2.3.2.- Resultado de tiempos del escenario 3

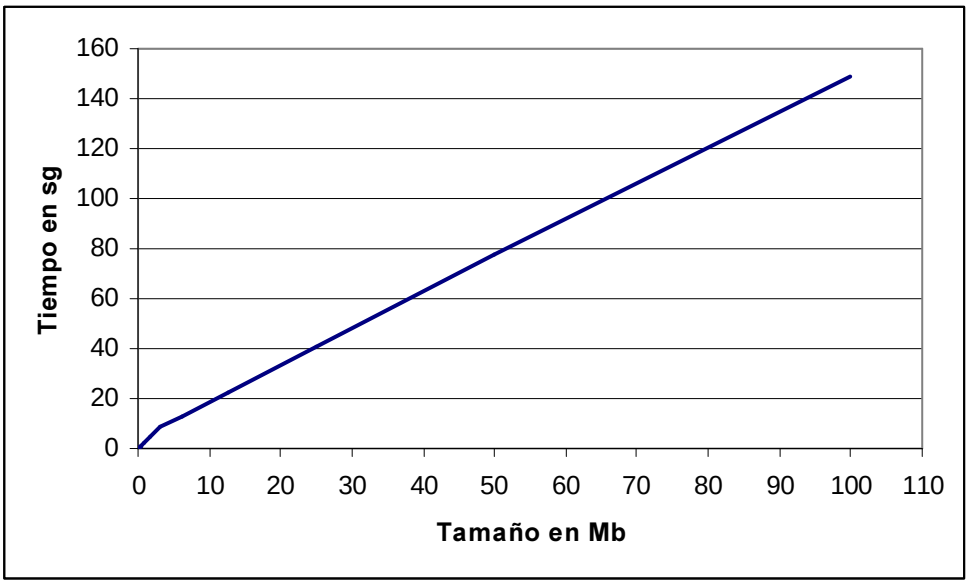
Agente 1 (A1) : C =5%, Cbs =2 testigos, Cir =3.2 testigos/segundo, 1/Cir=312 milisegundos

Agente 2.1 (A21) : C =5%, Cbs =2 testigos, Cir =3.2 testigos/segundo, 1/Cir=312 milisegundos

Agente 2.2 (A22) : C =4%, Cbs =2 testigos, Cir =2.56 testigos/segundo, 1/Cir=390 milisegundos

Agente 2.3 (A23) : C =3%, Cbs =2 testigos, Cir =1.92 testigos/segundo, 1/Cir=520 milisegundos

Podemos observar que los tiempos siguen incrementando. Esto se debe a que las capacidad del agente uno ha sido reducida a la mitad, es decir, ha pasado de 10% al 5%. El *Leaky Bucket* sigue rechazando para ficheros mayores de 7 trozos.



Gráfica 5.2.3.2 Resultado de tiempos de transmisión de ficheros de distintos tamaños.

En esta gráfica podemos ver representados los datos de la tabla 5.2.3.2.

Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A22 ¿rech LB?	Tiempo (sg)
17693	1	1 N	- N	- N	- N	0,183
40960	1	1 N	- N	- N	-N	0,245
92160	2	1 N	1 N	- N	- N	0,22
194560	3	1 N	1 N	1 N	- N	0,199
399360	7	2 N	1 N	2 N	2 N	0,332
808960	13	4 S	4 S	3 N	2 N	3,992
1628160	25	9 S	5 S	5 S	6 S	5,058
3266560	50	19 S	13 S	10 S	8 S	8,455
6543360	100	42 S	26 S	19 S	13 S	16,421
13096960	200	88 S	53 S	37 S	22 S	31,079
26204160	400	179 S	108 S	73 S	40 S	59,728
52418560	800	363 S	216 S	145 S	76 S	114,976
104847360	1600	723 S	438 S	291 S	148 S	228,516

Tabla 5.2.3.3.- Resultado de tiempos del escenario 3

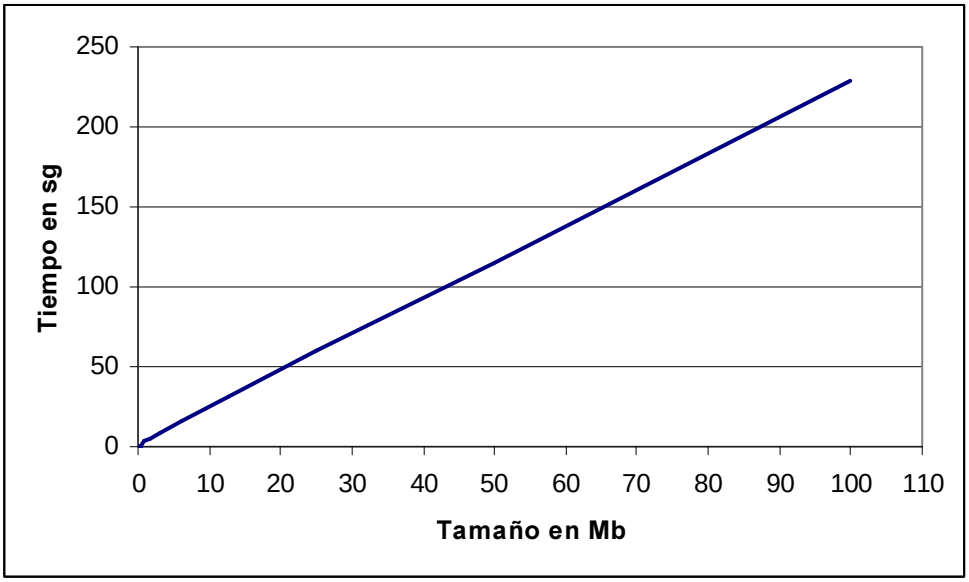
Agente 1 (A1) : C =5%, Cbs =2 testigos, Cir =3.2 testigos/segundo, 1/Cir=312 milisegundos

Agente 2.3 (A21) : C =3%, Cbs =2 testigos, Cir =1.92 testigos/segundo, 1/Cir=520 milisegundos

Agente 2.2 (A22) : C =2%, Cbs =2 testigos, Cir =1.28 testigos/segundo, 1/Cir=781 milisegundos

Agente 2.3 (A23) : C =1%, Cbs =2 testigos, Cir =0.64 testigos/segundo, 1/Cir=1562 milisegundos

En esta tabla obtenemos las mismas conclusiones que en la anterior, a menores capacidades de canal utilizadas, mayores tiempos de transmisión.



Gráfica 5.2.3.3 Resultado de tiempos de transmisión de ficheros de distintos tamaños.

En esta gráfica podemos ver representados los datos de la tabla 5.2.3.3

Tamaño (bytes)	Trozos	Trozos A1 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A21 ¿rech LB?	Trozos A22 ¿rech LB?	Tiempo (sg)
17693	1	- N	1 N	- N	- N	0,162
40960	1	1 N	- N	- N	- N	0,138
92160	2	- N	1 N	1 N	- N	0,256
194560	3	1 N	1 N	1 N	- N	0,317
399360	7	2 N	2 N	2 S	1 S	0,218
808960	13	4 S	4 S	2 S	3 S	2,061
1628160	25	10 S	5 S	5 S	5 S	3,437
3266560	50	26 S	8 S	8 S	8 S	9,125
6543360	100	58 S	14 S	14 S	14 S	19,207
13096960	200	119 S	27 S	27 S	27 S	37,604
26204160	400	244 S	52 S	52 S	52 S	77,604
52418560	800	494 S	102 S	102 S	102 S	156,266
104847360	1600	994 S	202 S	202 S	202 S	315,479

Tabla 5.2.3.4.- Resultado de tiempos del escenario 3

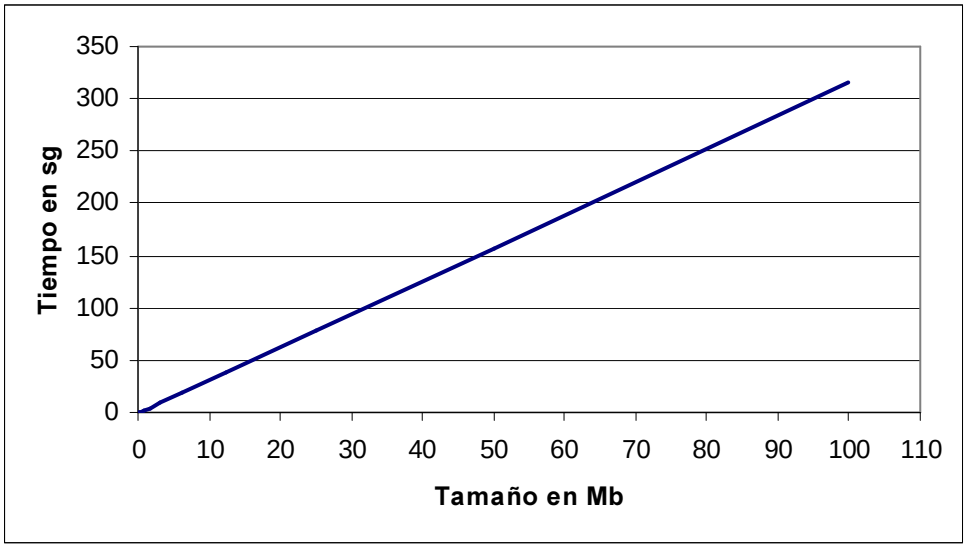
Agente 1 (A1) : C =5%, Cbs =2 testigos, Cir =3.2 testigos/segundo, 1/Cir=312 milisegundos

Agente 2.1 (A21) : C =1%, Cbs =2 testigos, Cir =0.64 testigos/segundo, 1/Cir=1562 milisegundos

Agente 2.2 (A22) : C =1%, Cbs =2 testigos, Cir =0.64 testigos/segundo, 1/Cir=1562 milisegundos

Agente 2.3 (A23) : C =1%, Cbs =2 testigos, Cir =0.64 testigos/segundo, 1/Cir=1562 milisegundos

En este escenario se han obtenido tiempos mayores que en el escenario dos debido a que ninguno de los agentes de este caso funcionan al 100% de su capacidad. Si se hicieran las pruebas con la totalidad de la capacidad de los canales, se obtendrían tamaños inferiores que en los escenarios anteriores.



Gráfica 5.2.3.4 Resultado de tiempos de transmisión de ficheros de distintos tamaños.

En esta gráfica podemos ver representados los datos de la tabla 5.2.3.1.

Como se ha podido ver, en el escenario dos las diferencias de tiempos entre los distintos casos del escenario es notable a partir del fichero de tamaño mayor (1600 trozos).

CAPÍTULO 6

CONCLUSIONES Y LINEAS FUTURAS

6.1-Conclusiones

El objetivo de este proyecto ha sido estudiar con un modelo sencillo, pero suficiente, el comportamiento de una arquitectura *Peer to Peer* basada en agentes para obtener recursos de un modo distribuido.

El resultado ha sido una aplicación específica *Peer to Peer* para compartir ficheros entre los diferentes agentes registrados en un servidor. Cada agente dispondrá de un identificador único que será útil para que otros agentes contacten con el y establezcan negociaciones de contratos.

La aplicación desarrollada en el proyecto está formada por un sistema híbrido, ya que combina partes distribuidas y centralizadas. Para su implementación se ha escogido el sistema *middleware CORBA,* junto al lenguaje de programación Java. A través de los

servicios de *CORBA,* se han conseguido desarrollar fácilmente los agentes de la red *Peer to Peer.*

Como elementos distribuidos tenemos los distintos agentes de la red. Cada uno puede estar ubicado en una máquina distinta. Por otro lado, los elementos centralizados serían el servidor de nombres de CORBA y el servidor gestor donde se registrarán los distintos agentes.

Se ha comprobado el funcionamiento del sistema. La gestión de los contratos se ha realizado con éxito. Se ha puesto a prueba el algoritmo *Leaky Bucket* en cada uno de los agentes, comprobando de este modo que se puede ajustar el ritmo de transmisión de servicio con otros agentes. Éste ha sido un factor muy importante, ya que esto hace posible que cada agente destine sólo el porcentaje de capacidad deseado para la transmisión de ficheros con otros agentes. También se han tomado datos para comprobar que cuantos más agentes participan en la transmisión de un fichero, se tarda menos tiempo en recibirlo.

Se han obtenido los resultados esperados de este proyecto. El único inconveniente ha sido que el tamaño de los ficheros a compartir ha estado limitado debido a restricciones del laboratorio donde se han realizado las pruebas. El espacio libre en disco duro que quedaba no dejaba hacer pruebas con ficheros mayores a los utilizados.

6.2-Líneas Futuras

Como se ha venido comentando, el objetivo del proyecto es el de implementar una arquitectura *Peer to Peer* para obtener recursos distribuidos. En este caso, se ha desarrollado para poder compartir archivos entre distintas máquinas con la única limitación indicada en los contratos.

Una de las cosas que se podrían hacer para comprobar mejor las características de este proyecto sería probarlo para ficheros grandes, que ha quedado sin probar debido a la limitación de la capacidad del disco duro de las máquinas con las que se han realizado las pruebas. De este modo el funcionamiento del algoritmo de control *Leaky Bucket* quedaría comprobado de un modo más óptimo y se podrían tener mejores conclusiones sobre el sistema.

Hay una serie de opciones que se pueden probar en un futuro aprovechando la arquitectura del software desarrollado en este proyecto como probarlo con servicios tales como la cesión de ancho de banda entre los distintos equipos de una intranet, la cesión de cpu entre distintas máquinas, etc.

Dentro de lo que es el software desarrollado en este proyecto, se podrían añadir más aplicaciones para disponer de una mayor comunicación entre los usuarios que manipulan cada agente, es decir, se podría añadir una aplicación de comunicación entre dos agentes mediante mensajes, un Chat. Éste sería útil para preguntar por información sobre posibles ficheros existentes en la red. Por ejemplo si este programa se usara en una empresa privada, nos serviría para pedir información a cerca de algunos catálogos, de información de gestión, etc.

Otra posible ampliación sería la de insertar un reproductor multimedia, un editor de textos y otras aplicaciones en la consola de casa cliente, para de este modo poder abrir los ficheros recibidos sin tener que abrirlos desde otro programa del ordenador.

Durante el proyecto también se ha hablado del establecimiento de contratos entre los distintos agentes de la red. En este caso, se establecía un contrato cuando a la hora de pedir un fichero había agentes que disponían de él y no se había establecido un número máximo de contratos.

Una posible mejora futura del proyecto sería aplicar los contratos con otras condiciones. Podemos regular el proyecto creando un algoritmo de creación de contratos de tal modo que para poder recibir archivos se tenga que haber cumplido ciertas condiciones dependiendo a las personalidades que pueda poseer cada agente.

Al hablar de personalidades se hace alusión a los distintos modos de actuar de los agentes atendiendo a una serie de condiciones. De este modo el sistema podría tener agentes con personalidades que podrían quedar bien definidas con nombres como “altruistas”, “egoístas”, etc.

Un agente “altruista” sería aquel que independientemente de si otros agentes le compartieran o no archivos, acepta un contrato con ellos para de esa forma transmitirles ficheros.

Para que un agente “egoísta” transmita archivos con un agente éste tendría que haberle compartido con anterioridad.

REFERENCIAS

REFERENCIAS ESPECÍFICAS UTILIZADAS

[2.1] http://es.wikipedia.org/wiki/P2P

[2.2] http://www.ub.es/geocrit/sn/sn-170-54.htm

[2.3] Clasificación según Pat Gelsinger de Intel

[2.4] http://www.emule-project.net

[2.5] http://www.edonkey2000.com/

[2.6] http://www.internautas.org/html/1/2644.html

[2.7] http://www.noticiasdot.com/publicaciones/2005/0105/1701/
noticias170105/noticias170105-09.htm

[2.8] http://ansuz.sooke.bc.ca/software/molester/

[2.9] http://www-db.stanford.edu/~byang/pubs/hybridp2p_med.pdf

[2.10] http://www.dcc.uchile.cl/~mmarin/revista-sccc/sccc-web/Vol4/ecc2.pdf

[2.11] http://lizt.mty.itesm.mx/catedra/decic-brena.pdf

[2.12] http://www.turismo.uma.es/turitec/turitec2004/docs/actas_turitec_pdf/11.pdf

[4.1] http://www.geocities.com/atomeua/GlosarioCorba.html

REFERENCIAS GENERALES

- **Client/Server Programming with Java and CORBA, Second Edition, by Robert Orfali, Dan Harkey**
- **Deitel & Deitel, "Cómo programar en Java", Ed. Prentice-Hall.**
- **http://java.sun.com/j2se/1.4.2/docs/api**

ANEXO 1

MODO DE EJECUCIÓN DEL SOFTWARE

Introducción

En este anexo se explica el modo de ejecutar el presente PFC. Se ha ejecutado bajo el sistema operativo *LINUX*, en una red de área local (*LAN*). Cada ordenador tiene instalada la versión j2sdk1.4.2 de Java 2. (*Software Development Kit*, kit de desarrollo de software de la plataforma Java 2).

Ejecución del Servidor de Nombres de CORBA

Hemos utilizado el servidor de nombres ***ORBD*** (*Object Request Broker Daemon*) que se usa para permitir a los clientes localizar e invocar transparentemente objetos persistentes en servidores en el entorno CORBA.

Se ejecutara únicamente en un ordenador siguiendo el siguiente esquema:

orbd –ORBInitialPort NumeroDePuerto –ORBInitialHost NombreDelHost

Imaginemos que hemos decidido que se ejecute en el puerto 1500 y en el puesto cuya dirección IP es 192.168.6.1, entonces lo ejecutaríamos del siguiente modo:

orbd –ORBInitialPort 1500 –ORBInitialHost 192.168.6.1

Ejecución del Servidor Gestor

Este servidor, que lo usaremos para que se puedan registrar los agentes del sistema, se ejecutara en una sola máquina, por ejemplo, en la que hemos lanzado el servidor de nombres.

Para proceder a la ejecución escribiremos lo siguiente:

java –cp gestor gestor.SerGestor -ORBInitialPort 1500

Si no ejecutáramos el servidor gestor en el mismo ordenador que el servidor de nombres tendríamos que especificar el host donde este ejecutándose éste mediante el parámetro *–ORBInitialHost* seguido del nombre del host.

Ejecución de los clientes y agentes

Por último nos falta ejecutar los clientes y los agentes. En cada ordenador tiene que haber un cliente y un agente. El cliente se encarga de elegir el archivo que quiere y hacerle la petición a su agente local. Éste se pondrá en contacto con otros agentes y se encargará de realizar todo el proceso.

La manera de ejecutarlos es el siguiente:

java –cp gestor gestor.SeriAgente –ORBInitialHost 192.168.6.1 –ORBInitialPort 1500

GLOSARIO

G4.1- IDL

(IDL: *Interface Definition Lenguaje*) Es un lenguaje que permite expresar tipos de datos, atributos, operaciones, interfaces, etc. Es independiente de los lenguajes de programación y se sitúa en un metanivel con respecto a ellos. Su sintaxis es parecida a la de Java o C++. IDL se aplica a los lenguajes de programación, mediante un compilador de IDL. (Ver Referencias [3.1]).

G4.2- Compilador IDL

Es una herramienta que toma por entrada un fichero .idl y genera como salida una serie de ficheros que juntos conforman parte de la arquitectura CORBA. En Java, es *idltojava* y proporciona como salida los ficheros *stub y skeleton*, dos ficheros de soporte (*Helper y Holder*) y una representación .java de la interfaz .idl. A partir de J2SE v1.4 el compilador es idlj y en lugar de generar el *skeleton* del servidor, genera un POA.

G5.1-Servidor ISP

(ISP *Internet Service Provider*) Servidor que provee servicios para Internet.

GA2.1- Control de flujo

Tráfico punto a punto entre un transmisor y un receptor. Evita que un transmisor rápido sature a un receptor lento. El control de flujo es una técnica más de control de congestión.

GA2.2- Control de congestión

Intenta asegurar que la subred sea capaz de transportar el tráfico ofrecido.

Sistemas Distribuidos

Un *Sistema Distribuido* es el que ofrece servicios implementados sobre una red de computadoras, como si se tratara de un único sistema. Es transparente al usuario o al sistema con respecto a localización, migración, réplica, concurrencia y paralelismo.

Este tipo de sistemas pueden estar basados en:

- Un sistema operativo distribuido: *Chorus, Mach, Amoeba, V Kernel, 2K, etc.* Tiene la propiedad de implementar más transparencias de distribución y ser más eficiente. Requiere acuerdo entre todos los sistemas para usar el mismo sistema operativo.

- En una plataforma de servicios de distribución (*middleware*): CORBA, DCOM, DCE, RMI. Generalmente soporta la heterogeneidad de sistemas. Tiene la propiedad de poseer una gran flexibilidad y de dar una solución global.

Cliente

Un cliente esta informalmente definido como una entidad (nodo, programa, módulo, etc.) que realiza peticiones pero no es capaz de servirlas. Si el cliente también es capaz de servir entonces juega el papel de servidor.

Servidor

Un servidor esta informalmente definido como una entidad que sirve peticiones de otras entidades. Un servidor no inicia nunca la petición realizada por un cliente, y si lo hiciera, jugaría el papel de cliente. Normalmente suele haber un mayor número de clientes que de servidores.