UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

TRIFS. UN SISTEMA NOVEDOSO DE DISTRIBUCION DE VIDEO CON P2P.

T E S I S

QUE PARA OBTENER EL TITULO DE:

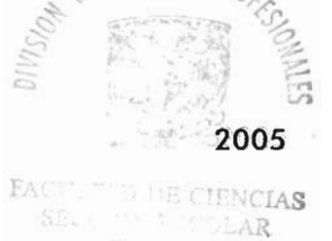
LICENCIADO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

Y U R I V A S I L E V S K I

FACULTAD DE CIENCIAS
UNAM

DIRECTOR DE TESIS: DR. SERGIO RAJSBAUM GORODEZKI



2005

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"Trifs, un sistema novedoso de distribución de video con P2P."

realizado por Yuri Vasilevski

con número de cuenta 09852571-1 , quien cubrió los créditos de la carrera de:

Licenciado en Ciencias de la Computación.

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	Dr. Sergio Rajsbaum Gorodezki	
Propietario	M. en C. Javier García García	
Propietario	M. en C. José de Jesús Galaviz Casas	
Suplente	Dr. Ricardo Marcelin Jiménez	
Suplente	Lic. En C. C. Manuel Sugawara Muro	

Consejo Departamental de Matemáticas

Dr. Francisco Hernández Quiroz

CONSEJO DEPARTAMENTAL DE

Trifs, un novedoso sistema de distribución de video basado en tecnología P2P

Yuri Vasilevski

A la Internet ...

Tabla de Contenido

Índice de figuras

Sinopsis

En esta tesis se diseña un sistema de distribución de contenido multimedia, especialmente distribución de video. Éste está basado en la idea de las redes *Peer2Peer* híbridas y usa la Internet como medio de distribución. Ofrece las siguientes ventajas: buena escalabilidad, robustez (tanto en condiciones de operación normales, como durante picos en la red), garantías de calidad de servicio y costos de infraestructura prácticamente nulos. Esto se obtiene particionando el protocolo en tres capas, cada una de las cuales es muy sencilla y completamente enfocada a proveer una funcionalidad concreta. La primera (capa de *peers*) es la encargada de la transferencia del contenido en sí y usa la idea de aglomeramiento de archivos (*file swarming*) para optimizar su eficiencia. Las segunda (de llave) y tercera (de índice) capas sirven como un directorio del contenido manejado en la primera, y también se encargan de asegurar la disponibilidad y replicación de éste. Aparte, la tercera capa también se encarga de arbitrar la agregación del contenido. Otra parte importante de este sistema es que regula los flujos de su contenido, con lo que puede aprovechar de manera más eficiente los recursos que sus integrantes aporten.

Con esto se crea un sistema de distribución de contenido capaz de funcionar en el mundo real (y no sólo un modelo con propiedades teóricas interesantes) que posee los requerimientos de infraestructura y robustez dignos de un buen sistema P2P y garantías de calidad de servicio que se acercan a las ofrecidas en los sistemas cliente-servidor.

Introducción

0.1. Distribución de Contenido

Distribución de Contenido es un término muy general, usado para describir el modelo de distribución **de uno a muchos**. En general este contenido puede ser cualquier tipo de información digital y lo que es muy importante es que generalmente no son cruciales los tiempos de distribución de éste. Para solucionar el problema de Distribución de Contenido existen varias alternativas. Las basadas en el paradigma de cliente-servidor que presentan grandes problemas de escalabilidad, aparte de poseer un punto de fallo único (que es el servidor). También existen las basadas en el paradigma *Peer2Peer*, cuyo mayor problema es la prácticamente total imposibilidad de tener garantías de calidad de servicio en la distribución del contenido. Muchos P2P como `Napster` [5], `Gnutella` [4], `Freenet` [3] no son escalables, ya sea debido a la necesidad de servidores de índice centralizados (`Napster`) o debido a los algoritmos de ruteo de búsquedas que usan (`Gnutella` [15] y `Freenet`).

0.1.1. Redes de Distribución de Contenido

Las **Redes de Distribución de Contenido** o **CDN** (por sus siglas en inglés), también utilizan el enfoque basado en servidores, pero la idea de éstas es poner una serie de servidores replicados "al borde de la Internet"[1], así los clientes pueden conectarse al servidor

geográficamente más cercano y de este modo esquivar los cuellos de botella de Internet. Esto en especial facilita la distribución en tiempo real y las garantías de calidad de servicio necesarias para ésta, por lo que son muy usadas para la distribución de señales multimedia en Internet. El problema de los CDN es que son muy costosos de poner y mantener en operación, además debido al gran dinamismo en los posibles clientes es muy difícil tener un balanceo de carga real. De esta manera es inevitable la creación de cuellos de botella en algunos servidores, mientras que otros están subutilizados en un tiempo dado.

0.1.2. Redes *Peer2Peer*

La idea de las redes ***Peer2Peer*** o **P2P** en el contexto de distribución de contenido, es la de usar los recursos (principalmente ancho de banda) de los clientes interesados en el contenido para la distribución de ese contenido. Esto se realiza organizando los nodos de los clientes a forma de que una vez que un nodo obtenga el contenido en el que esta interesado, done alguna parte de su ancho de banda para que otros nodos interesados en ese contenido lo puedan obtener de él. Debido a que de antemano no se puede saber exactamente cuales van a ser las capacidades ni la disponibilidad de los nodos integrantes de una red de esta naturaleza, al diseñar la red se asume que todos los nodos van a ser iguales entre si y van a desempeñar potencialmente las mismas funciones. De este modo, gracias a que en las redes P2P la distribución del contenido se realiza utilizando los recursos de los nodos interesados en el contenido, se requiere muy poca infraestructura del lado del productor del contenido, lo que reduce significativamente los costos. Aunque, ya que no se tiene ningún control sobre las capacidades o el comportamiento de los nodos (tambien llamados ***peers***) de una red P2P, es muy difícil poder dar estimaciones del ancho de banda disponible o de la disponibilidad del contenido, por lo que es el peor caso para el ofrecimiento de garantías

de calidad de servicio.

0.1.3. Aglomeramiento (*Swarming*)

Con la aparición del llamado "Efecto *Slashdot*" [13] al final de la década de los 90's se demostró la gran debilidad del paradigma cliente-servidor. Este efecto se manifiesta cuando una desproporcionalmente gran cantidad de clientes se interesa por un servicio en un intervalo de tiempo reducido. Es muy difícil y costoso atender este tipo de situaciones, ya que el tráfico pico se diferencía enormemente del tráfico promedio. Esto generalmente causa la interrupción total del servicio, o en el mejor de los casos, una degradación significativa en la calidad del mismo. Cabe también mencionar, que los sistemas P2P tradicionales también sufren de este problema. Debido a esto hace varios años surgió la idea de aglomeramiento en los sistemas P2P que consiste en dividir el contenido en pequeños pedazos, así conforme un cliente obtiene una parte del contenido la pone a disposición de los demás clientes sin esperar a tener el contenido completo. Esto permite que el ancho de banda y las fuentes disponibles crezcan linealmente con respecto a la cantidad de *peers* y por ello los sistemas con aglomeramiento no sufren del efecto *slashdot*. El aglomeramiento actualmente es usado en muchos protocolos P2P como `eDonkey2000` [2, 8] y `Overnet` [2] desarrollados por Jed McCaleb y `BitTorrent` [1] creado por Bram Cohen.

0.2. Este trabajo

Esta tesis presenta un nuevo protocolo de una red de distribución de contenido multimedia basado en el paradigma *Peer2Peer*. En este protocolo se usa la idea de aglomeramiento y se le quita la decisión a los usuarios de los *peers* respecto a qúe es lo que van a compartir. Esto ayuda a tener ciertas "garantías" de calidad de servicio y de distribución de contenido

en tiempo real. De este modo se obtienen los beneficios tanto de los CDN como de las redes P2P.

0.3. Trabajos relacionados

Como premisa de este trabajo se hizo la investigación de otros trabajos que se han publicado en el área:

- El trabajo [17] presenta otra alternativa de distribución de contenido multimedia mediante el paradigma P2P. A diferencia del presentado en esta tesis, obtiene las "garantías" de calidad de servicio necesarias para este tipo de aplicación dividiendo la topología de la red en grupos geográficos. Esto dificulta el mantenimiento de la red y aumenta significativamente el costo de las búsquedas del contenido si éste no está disponible en el grupo local del nodo que está buscando. Así mismo queda la duda de la escalabilidad de la red propuesta en este artículo.

- Los trabajos [10] y [12] proponen el uso de un protocolo basado en `BitTorrent` (esta basado en el aglomeramiento de archivos) para la distribución de contenido multimedia. Este enfoque tiene la gran ventaja de estar basado en un protocolo tan ampliamente estudiado y probado como es el `BitTorrent`, pero hereda varias de las desventajas para el rubro de distribución de contenido del mismo. En el artículo [10] se estudia el caso en el que el productor está transmitiendo el contenido en tiempo real, por lo que es aceptable la necesidad de un *tracker* centralizado que mantendría el mismo productor. Sin embargo la transmisión en tiempo real es un caso muy particular y esto reduce mucho el uso de dicho protocolo. A su vez en [12] se incorporan los *trackers* de `BitTorrent` en el sistema P2P que está creando, de este modo eli-

mina el punto de falla único que tiene el `BitTorrent`, pero no propone ninguna garantía de calidad de servicio fuera de las ofrecidas por la técnica de aglomeramiento. Esto en la práctica, se ve claramente con la red P2P `eDonkey2000` que no puede proveer ninguna garantía de calidad de servicio cuando el espacio de búsqueda crece. En efecto, un usuario que desee obtener un contenido de baja demanda, con alta probabilidad deberá esperar un largo tiempo, puesto que los recursos de los pocos nodos que tienen el contenido deseado se destinarán mayormente a la distribución de los contenidos populares.

A pesar de esto, [12] es el trabajo más cercano al protocolo propuesto en esta tesis, sólo que el protocolo aquí expuesto permite controlar la disponibilidad de las partes inpopulares con lo que puede ofrecer garantías de calidad de servicio más fuertes que [12].

0.4. Requerimientos de Diseño

La idea de este trabajo es crear `Trifs`, una red de distribución de contenido que minimice la infraestructura que ésta requiera, hasta el punto en que cualquier persona pueda crear y mantener un canal de distribución de contenido sin invertir prácticamente nada en infraestructura y tiempo.

Para esto la red se va a dividir en una serie de **canales** (llamados así a semejanza de los canales de televisión) y cada canal va ser una comunidad (grupo de suscriptores/usuarios), la cual va a estar interesada en los **programas** (llamados así a semejanza de los programas en los canales de televisión) que ofrece el canal. Estos programas van a ser el contenido que ofrece el canal. A cambio del acceso al contenido, los usuarios van a proveer cierto espacio de almacenamiento y ancho de banda al canal para su propagación y distribución y

éste va a ser agregado por (algunos de) los usuarios del canal.

Como toda red para distribución de video, la correspondiente a esta propuesta también tiene ciertos requerimientos.

Primero que nada los requerimientos comunes entre cualquier red de distribución de video:

- La suscripción a canales debe ser controlada por los usuarios. Esto significa que los usuarios controlarán completamente que canales van a ver, sin los "canales de bono" que a los proveedores de televisión tanto les encantan. Esto también significa que los canales deben ser independientes entre sí y no interferir unos con otros.

- La calidad de señal debe ser controlada. Esto significa que para cada canal al cual un usuario se vaya a suscribir, éste canal debe proveer información respecto a cuál es la calidad de audio y video del contenido que se puede esperar de ese canal. Además, el canal debe tener una manera de forzar que la calidad del contenido que acepte esté en los rangos de calidad esperados.

- La señal debe ser ininterrumpible. Esto significa que tienen que existir ciertas garantías respecto a la distribución de la señal para asegurar que un programa puede ser visto sin fallas ni interrupciones.

En segundo lugar la propuesta de esta tesis añade los siguientes requerimientos específicos a esta red de distribución de video:

- Descentralización. Todos los usuarios de un canal dado deben donar de manera voluntaria los recursos requeridos por este canal. Estos deben ser utilizados por el canal de manera que las funciones de éste sean distribuidas entre todos los nodos y que el

canal pueda sobrevivir fallas aisladas en cualquiera de sus nodos (i.e. no debe tener punto único de fallo).

- Infraestructura *Ad-hoc*. Los suscriptores deben poder entrar y salir de la red sin comprometer la funcionalidad un canal.

- Escalabilidad. El sistema debe poder mantener tanto canales exageradamente populares con muchos miles de suscriptores, como canales muy pequeños con tan sólo los suscriptores suficientes para que el canal subsista.

- Bajo mantenimiento. Un canal prácticamente no debe requerir la intervención de usuarios para su existencia, esto inclusive en ambientes sumamente dinámicos.

Aparte de estos requerimientos, este sistema fue diseñado para proveer las siguientes funcionalidades que lo harán separarse de las hasta ahora conocidas redes de distribución de video.

Con respecto a cómo ver los programas:

- La posibilidad de ver contenido en vivo, i.e. en tiempo real con respecto a la programación.

- La posibilidad de ver cualquier programa de la programación, en cualquier momento.

- La posibilidad de almacenar programas de la programación para verse en un futuro.

- La posibilidad de detener, volver a ver algún pedazo, saltarse pedazos y en general, navegar cualquier programa a la hora de verlo.

- Tener un sistema de etiquetas bien definido para la información de los programas y canales.

Con respecto al manejo de un canal:

- Los canales son especializables a una audiencia particular, se logra validando el contenido que acepten sobre las etiquetas asociadas a los programas.
- La agregación de contenido puede ser restringida, ya sea a un conjunto de personas, o de forma aún mas restringida con licencias por cada acción.
- La persona que agregó un programa no tiene ninguna obligación de estar conectado a la red mientras su programa esta en la programación.

0.5. Organización de la Tesis

Esta tesis se organiza de la siguiente manera:

Capítulo 0: Introducción. El capítulo que actualmente está leyendo. Su función es desarrollar las motivaciones para el diseño de un sistema como el descrito en esta tesis. Así como, ubicar al lector en un contexto histórico con respecto al problema que esta tesis intenta resolver.

Capítulo 1: Diseño del Sistema. En este capítulo primero se especifican el contexto y las bases con las que vamos a armar el sistema, después se ve que éste esta compuesto de tres módulos funcionalmente distintos entre sí (el módulo de red, el módulo de seguridad y el módulo de usuario) y se explica el comportamiento y las desiciones de diseño de estos módulos.

Capítulo 2: Módulo de Red. Éste es es el módulo principal del sistema y como tal es al que se le brinda más atención. En este capítulo primero se describe a detalle el funcionamiento y la organización de este módulo, el que se compone de tres capas. La primera capa, es la capa en la que sucenden todas las transferencias y manejo del contenido de un canal. Mientras que en las segunda y tercera capas se organiza toda la información sobre la localización de este contenido a forma de un directorio, lo que permite búsquedas muy rápidas sobre el contenido almacenado en la primera capa.

Después se especifica la realización de todos los servicios y funcionalidades que provee el protocolo de red `Trifs` como la creación de un canal, la agregación del contenido y las distintas formas de la obtención de éste.

Por último, se detallan todos los procesos de estabilización que tienen que seguir los nodos de un canal para garantizar su mantención y sano funcionamiento independientemente de la entrada/salida de otros nodos y la agregación/expiración del contenido.

Capítulo 3: Módulo de Seguridad. En este capítulo se detallan las dos funciones del módulo de seguridad y la organización del mismo. La primera de estas funciones es la de manejar y organizar el contenido de un canal. Para esto toda la información de un canal se divide en bloques pequeños según las especificaciones de este módulo y se organiza en un árbol jerárquico a forma de un sistema de archivos. La segunda función, es la de filtrar tanto el contenido que se agrega al canal para evitar abusos a este canal, como el contenido que se le presenta a los usuarios de los nodos para evitarles ver contenido ideseable, ofensivo o penoso.

Capítulo 4: Módulo de Usuario. Este capítulo cubre el último módulo del sistema y a diferencia de los demás módulos, al este módulo tratar sobre la integración de los usuarios

con sus nodos, es irrelevante su organización a las especificaciones de `Trifs` mientras que provea la posibilidad al usuario de interactuar con el sistema y obtener contenido de él. Por lo que en este capítulo sólo se especifica que funcionalidades debería proveer un módulo de usuario sin especificar la realización de éstas y se da un ejemplo de cómo se podría organizar e implementar uno.

Capítulo 5: Uniendo Todo. Aquí se describe el funcionamiento de un canal a lo largo de su existencia y la forma en la que interactúan los diversos módulos de cada nodo en cada una de las situaciones que pueden aconteserle a un canal. Para esto, la vida de un canal se divide en tres estapas; una etapa de infancia, en la que un canal se crea y trata de llegar a un estado funcional completo; la etapa adulta, en la que el canal va a pasar el mayor tiempo y la única en la que el canal es completamente funcional y capaz de satisfacer todos los requerimientos del sistema; y por último, la etapa de vejez, en la que un canal manifiesta funcionamieto incompleto o erróneo. Por último, es este capítulo se hacen mediciones de desempeño de un canal `Trifs` tanto en condiciones de funcionamiento normal como durante la manifestación de efecto *slashdot* y éstas se comparan con los desempeños de las diversas alternativas que hay actualmente.

Capítulo 6: Conclusiones. En este capítulo vienen las reflecciones de lo que se cree que se logró con el protocolo propuesto en esta tesis. Se comentan los que el autor de esta tesis cree que son los puntos fuertes y las debilidades que presenta el sistema. Por último, se plantea el camino a seguir para futuro desarrollo y mejoras tanto de este protocolo como de protocolos basados en él.

Apéndice A: Comandos del Módulo de Red En este apéndice se especifican formalmente todos los comandos (mensajes) válidos de la comunicación entre los módulos de red de

los nodos de un canal.

Apéndice B: Esquemas de XML Aquí se especifica la organización de la metainformación del sistema en esquemas de XML, tal y como estaría almacenada en los meta bloques.

[1] Se refiere a poner servidores en sitios que se encuentran a muy pocos saltos (*hops*) de distancia de los usuarios finales, esto generalmente significa poner estos servidores directamente en las redes internas de los Provedores de Servicios de Internet (ISPs) con más clientes.

Capítulo 1

Diseño del Sistema

Como se plantea en la introducción nuestra red de distribución de contenido va a dividir la red en **canales**. Por su parte, el contenido que se distribuye en los canales y se compone de entidades indivisibles que llamaremos **programas**. La manera más intuitiva de relacionarlos es definiendo un canal como una secuencia potencialmente infinita de programas.

Por otro lado, es altamente improbable que al momento de la creación de un canal dado, la vida potencialmente infinita de éste se conozca con la precisión necesaria para conocer cada programa que en algún momento incluirá. Pero, la buena noticia es que, en la gran mayoría de los casos, eso es absolutamente innecesario, ya que es suficiente conocer la programación a un futuro cercano en cada momento dado. Así que tomando esto en cuenta, para nosotros, un **canal** es una ventana de alguna longitud constante moviéndose sobre la potencialmente infinita secuencia de programas de este canal.

Una vez definida cómo es la información que tiene que manejar el sistema, podemos dar una descripción de muy alto nivel de su funcionamiento.

Cada persona que esté interesada en formar parte de algún canal, tiene que crear un **nodo** dedicado para ese canal. En el caso de que el canal todavia no exsista, el primer nodo

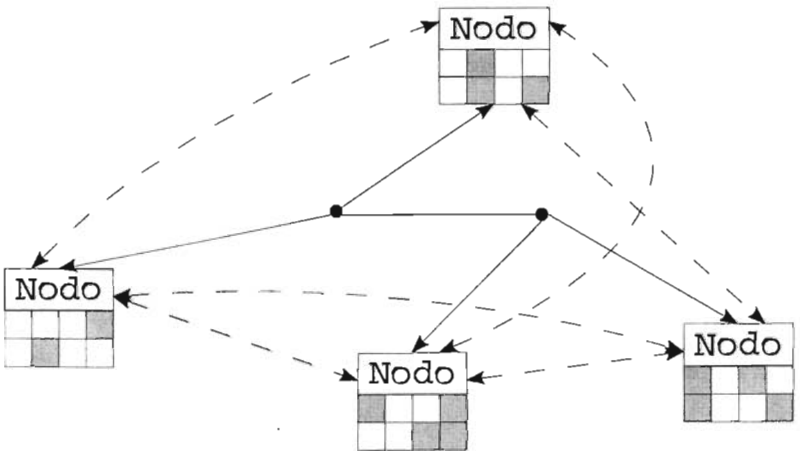


Figura 1.1: Topología de un canal.

dedicado al canal, lo crea. Este nodo le va a servir a su usuario para acceder al contenido almacenado en este canal y también al canal dejando que `Trifs` lo administre (en cuanto a que material debe proveer este nodo) según crea que es mejor. Es por ello, que las personas que deseen formar parte de algún canal tienen que estar dispuestas a seguir las estipulaciones de ese canal y a permitir el almacenamiento de alguna parte de su contenido, este se hará en la **capa de peers**.

Aparte de esto, para que la información almacenada en un canal sea accesible, es necesario agregar una capa que provea de cierta estructura (o hasta un directorio) sobre el contenido almacenado en un canal. Para ello, los dueños de los nodos pueden (y es muy aconsejable que lo hagan) proponerse para almacenar la información referente a la localización del contenido en toda la red (formada por los nodos pertenecientes a un canal) y poder servir de guías cuando otro nodo requiere localizar algún **bloque**[1]. A su vez, para facilitar su funcionamiento, esta capa se divide en dos subcapas de manera jerárquica. La primer capa, que llamaremos **capa de llave** va a proveer información respecto a donde es

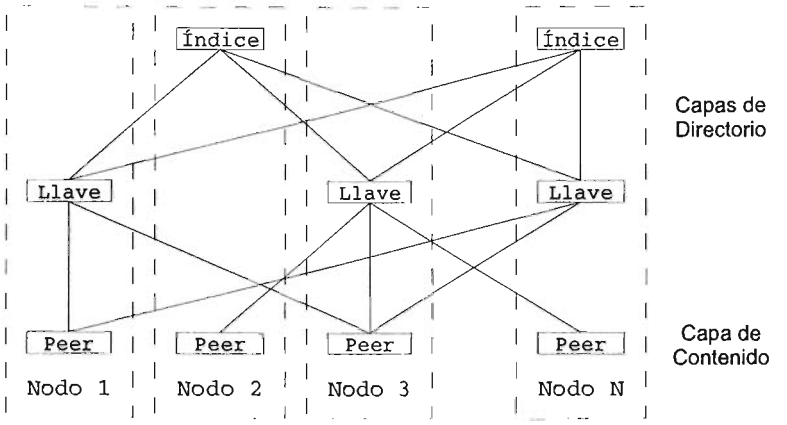


Figura 1.2: Organización de las capas

localizable alguna parte concreta del contenido total del canal. La segunda capa, que llamaremos **capa de índice** proveerá información sobre a que miembro de la primer capa se le debe preguntar respecto a la información de alguna parte del contenido del canal (ver figura 1.2).

De este modo, debido de que tenemos nodos en jerarquías distintas, usando esta estructura no podemos crear una red **Peer2Peer pura**[2]. Pero tenemos la gran ventaja de que al tener nodos con un panorama global es mucho más fácil estudiar la dinámica de una red como la que se plantea en esta tesis. Así a los nodos les es mucho más sencillo saber cuando hay algún problema con el canal y asegurar una buena disponibilidad del contenido almacenado en él. Como se verá más adelante, en la práctica, la introducción de esta estructura altamente jerárquica no impone irregularidades sobre los nodos participantes en la red más allá de las que se presentan en el mundo disímil en el que vivimos. Por lo que a lo largo de esta tesis podremos construir una red **P2P híbrida**[3] usando esta estructura con sus beneficios.

Este sistema se compone de varios módulos independientes entre ellos. Cada uno de ellos provee una función específica, cuya interoperación permitirá al sistema satisfacer los requerimientos impuestos sobre el diseño del sistema y proveer las funcionalidades esperadas.

El módulo más interesante, por mucho, es el **módulo de red**. Este módulo es el responsable de la distribución del contenido, del mantenimiento de la consistencia de la red y de proveer el acceso a la información almacenada en esta red. Como ya antes se vió, este módulo se va a componer de tres capas: capar de *peers*, capa de nodos de llave y capa de nodos de índice.

El **módulo de seguridad** es el encargado, tanto de la autenticidad de los usuarios, como la del contenido, y como tal, su principal función es la de regular las modificaciones que sufra el canal.

Por último el **módulo de usuario** que integra la interacción del usuario con el sistema completo. Su meta es proveer una interface limpia para interconectar los servicios que el sistema provee con otros programas que se encarguen de manejar el sistema, o bien, en algún futuro, de interconectar al sistema con otro módulo que interactúe directamente con usuarios humanos.

Con este diseño tenemos módulos independientes y autocontenibles, lo que simplifica mucho la realización de una implementación robusta del protocolo. Así mismo, el tener esta separación entre el protocolo y la interface permite una gran flexibilidad para su implantación y en la integración de `Trifs` con otros sistemas.

En este sistema la información es propagada por el módulo de red, en una red de tipo *Peer2Peer*. Esto se hace tratando de asegurar la disponibilidad del contenido, independientemente de que los usuarios estén entrando y saliendo de la red de manera continua e incontrolable. En especial, esta red P2P fue diseñada bajo la idea de que las personas son

insignificantes seres egoístas e ignorantes, por lo que es responsabilidad de la red la propagación y replicación de la información para asegurar su disponibilidad en un ambiente tan dinámico.

[1]Pedazo de información, ya sea de un programa o sobre el canal.

[2]Una red P2P es pura si asume y requiere que todos sus nodos integrantes sean exactamente iguales entre si.

[3]Una red P2P es híbrida si requiere que algunos de los nodos que la integran realicen alguna labor (generalmente de servidor) distinta a otros nodos.

Capítulo 2

Módulo de Red

2.1. Descripción

El módulo de red, es el principal módulo del sistema. Es el responsable de toda la comunicación y transferencia de datos en la red, así como de la propagación y del aseguramiento de la disponibilidad de la información a lo largo del tiempo de vida del sistema.

Para hacer esto, el módulo va a implementar una red por capas *quasi* P2P con el siguiente funcionamiento.

La capa base va a ser integrada por los nodos a los que llamaremos ***peers*** y precisamente en esta capa se van a hacer todas las transferencias de contenido y éste va a ser almacenado. Para esto, los programas van a ser divididos en **bloques** pequeños (del orden de centenares de *kilobytes*) para asegurar una rápida propagación y balanceo de carga. Después cada bloque va a tener una **llave** asociada a él; esta llave es obtenida mediante la aplicación de una función *hash*, criptográficamente segura[1], al contenido de cada bloque. También habrá un bloque distinguido: el **bloque maestro**[3]. Este bloque va a ser la raíz de toda la información almacenada: tanto sobre el canal como sobre los programas y su programación.

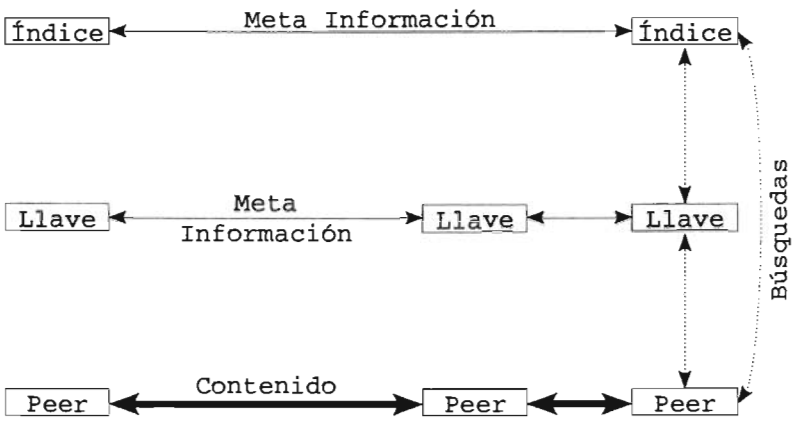


Figura 2.1: Capas del Módulo de Red.

La llave de este bloque va a ser el *hash* de la llave pública del canal y como tal, también va a funcionar como el nombre del canal.

Los nodos de la siguiente capa de la red van a ser los responsables de mantener la información asociada a una llave específica. Estos sabrán qué *peers* poseen el bloque asociado a esta llave y también recolectarán reportes de calidad de servicio (qué tan fácil/difícil fue la obtención del bloque) de los *peers*. Basándose en eso, replicarán el bloque a otros *peers* en caso de que ello sea necesario. Estos nodos van a ser llamados **nodos de llave**.

Finalmente la última capa de la red proveerá un índice/directorio con información de en dónde se puede obtener información respecto a una llave dada. Por ello, los nodos pertenecientes a esta capa son llamados **nodos de índice**. Su funcionalidad básica será asociar llaves a nodos de llave, de una manera muy similar a como los servidores DNS asocian nombres de dominio a direcciones IP. También van a obtener reportes provenientes de nodos de llave respecto a calidad de servicio y en caso de que consideren la situación crítica, van a tratar de convencer a *peers* para que bajen esos bloques críticos y así tratara

de asegurar su disponibilidad en la red.

Otra función muy importante de los nodos de índice va a ser la de aceptar programas nuevos al canal, basándose en las políticas que se tengan especificadas en el canal. Para después propagar los bloques del programa recién agregado y una vez que lo considere estable en el canal, actualizar los bloques de programación.

Es muy importante mencionar que cada nodo en la red va a ser un *peer*, y después, si el nodo así lo quiere, puede también empezar a ser un nodo de llave o de índice.

2.2. El Protocolo de Red en Operación

En esta sección se presentan los procedimientos para realizar las distintas tareas que permite realizar el protocolo. Estos procedimientos tanto en su explicación, como en el pseudocódigo presentado, hacen uso de los distintos comandos disponibles para la comunicación entre los módulos de red de los distintos nodos de un canal y especificados por `Trifs` cuya sintáxis y significado exacto se puede ver en el apéndice A.

2.2.1. Haciendo *Bootstrap* a un canal

Para hacer *bootstrap*[2] a un canal el creador debe tener una pareja de llaves privada/pública, la definición[3] del canal que está creando y debe permanecer en línea mientras que el canal en cuestión entra en una etapa funcional autosostenible.

Primero, el creador empieza como *peer*, nodo de llave y nodo de índice para todos los bloques que contengan la definición del canal, y a lo mejor, algunos programas que el creador desee agregar desde la creación del canal. Después, hace de conocimiento público (de alguna manera) su llave pública (al mismo tiempo el nombre del canal) y su dirección IP.

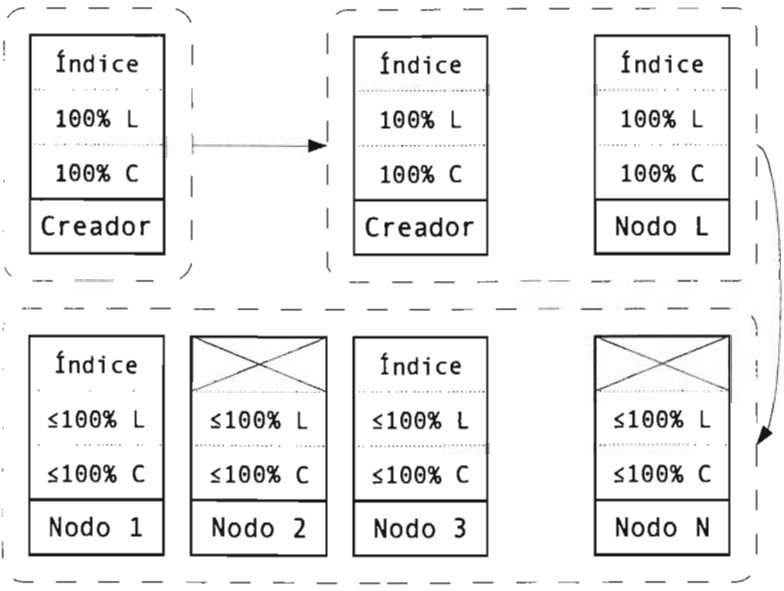


Figura 2.2: Proceso de *bootstrap*.

A partir de este momento, mientras que otros nodos entran a la red, va a empezar la replicación de bloques y la clonación de nodos de llave/índice por el funcionamiento innato del sistema. Cuando haya suficientes copias de cada bloque y de los nodos de llave/índice, la existencia del canal se va a independizar de la disponibilidad del nodo creador del canal.

2.2.2. Entrando al Canal

Si un nodo quiere entrar al canal todo lo que tiene que conocer es la dirección de un nodo perteneciente al canal y el nombre (la llave del bloque maestro) del canal al que quiere entrar. Una vez que tenga esta información, puede contactar al nodo que conoce del canal para preguntarle la lista de direcciones de nodos de índice del canal (con SERVERS). Después, este nodo ya es considerado parte del canal y puede obtener información de él.

Una vez en el canal, el nodo deberá agregar los bloques que tiene almacenados a los bloques disponibles en el canal. Para ello, primero tiene que bajar la programación actual y ver cuáles de los bloques que tiene almacenados siguen siendo vigentes y después notificar a los nodos de llave sobre el hecho de que está almacenando los bloques que siguen siendo vigentes.

También es muy adecuado, que el nodo se proponga para la lista de nodos potenciales de llave y/o índice.

Figura 2.3: Entrada al canal.

2.2.3. Obteniendo Bloques del Canal

Para obtener un bloque el *peer* **A** debe formar parte del canal y conocer la llave KEY del bloque que él desea obtener.

Primero, el *peer* **A** le pregunta a un nodo de índice (aleatoriamente escogido) su lista de nodos de llave responsables de la llave KEY (manda INFO KEY). Después le pregunta a un nodo de llave **L** (aleatoriamente escogido de entre la lista que le acaba de mandar el nodo de índice) qué *peers* tienen el bloque con la llave KEY (manda FIND KEY). Posteriormente, **A** entra en la cola de espera de su nodo favorito (recordando quiénes han sido buenos nodos para él o escogiendo aleatoriamente) **B** (manda QUEUE KEY) y espera. Mientras tanto, **A** puede monitorear su progreso en la cola (manda PING) y en caso de que **B** lo quite de la cola o la espera sea demasiado prolongada, **A** puede entrar en la cola de otro nodo.

Una vez que **A** recibe el comando `CONNECT KEY PORT` tiene que tratar de abrir una conexión TCP al puerto `PORT` del nodo (vamos a asumir que) **B**. Y si no logra conectarse, le pide a **B** que intente abrir la conexión (manda `CONNECT KEY PORT`). Esto puede ser necesario si el nodo **B** está atrás de un *firewall* o una máscara de red.

Una vez establecida la conexión TCP, el *peer* **A** simplemente le pide el bloque en cuestión (manda `GET KEY 0`) y una vez que lo haya bajado, cierra la conexión (manda `BYE`).

Para terminar, **A** tiene que informar a varios (de 3 a 5) nodos de llave, de la lista recibida del nodo de índice, que él también tiene el bloque asociado a la llave `KEY`.

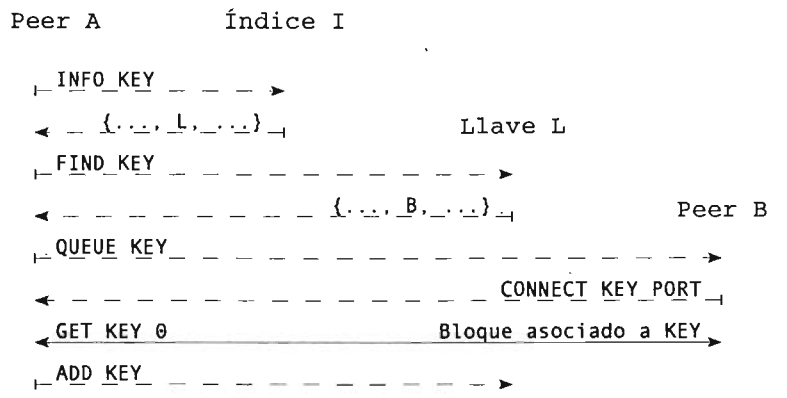


Figura 2.4: Obteniendo bloques.

2.2.4. Mandando Bloques a *Peers*

El proceso comienza cuando un *peer* **B** es contactado por otro *peer* **A** para entrar en su cola de espera (recibe `QUEUE KEY`). Posteriormente **B** atiende a otros nodos hasta que llega el tiempo de mandarle información a **A**. Entonces **B** le manda `CONNECT KEY PORT` a **A** y espera una conexión TCP a su puerto `PORT` de **A** u otro comando `CONNECT KEY PORT` de

A que le indique que él debe de intentar iniciar la conexión TCP.

Una vez iniciada la conexión TCP entre **A** y **B**, **B** espera recibir sobre la recién creada conexión el comando `GET KEY OFFSET`, en cuyo caso empieza a mandar el contenido del bloque asociado a la llave `KEY` a partir del *byte* `OFFSET`; o bien el comando `BYE`, en cuyo caso cierra la conexión TCP a **A**.

2.2.5. Obteniendo Bloques en Tiempo Real

Este es el caso cuando el nodo **A** necesita obtener bloques de un programa que la persona detrás del nodo **A** está viendo. En este caso es crucial poder obtener bloques, al menos en tiempo real con respecto a la tasa de información (*data rate*) del programa, o preferiblemente, tan rápido como **A** pueda hacerlo. Por lo tanto este tipo de transferencia de información entre *peers* recibe la mayor prioridad posible.

Para lograrlo, cuando el nodo **A** solicita el bloque, manda `QUEUE KEY REALTIME` en vez de `QUEUE KEY` para indicar su impaciencia y lo solicita a varios nodos al unísono.

Cada nodo que manda información (cada *peer*) también debe tener una bahía (*slot*) especial para mandar únicamente bloques de impaciencia `REALTIME`.

2.2.6. Obteniendo Bloques para Ver Después

Este es el caso, cuando un *peer* **A** necesita obtener los bloques de un programa que la persona detrás de **A** ha marcado para verlo después. En esta situación opuesta a la de tiempo real, es de menor exigencia sobre los tiempos de transferencia. Así que, es el caso de menor prioridad para transferencias entre *peers*.

Cuando el *peer* **A** solicite bloques, manda el comando `QUEUE KEY SCHEDULED` en lugar del comando `QUEUE KEY` para indicar el propósito de estas transacciones.

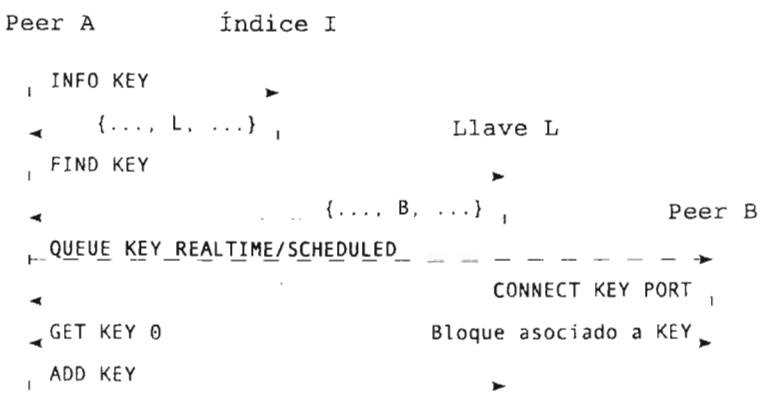


Figura 2.5: Obteniendo bloques en tiempo real/para ver después.

Cada nodo que envíe información (cada *peer*) debe de mandar los bloques de impaciencia SCHEDULED a través de bahías (*slots*) de propósito general, junto con bloques de impaciencia REALTIME, PROPAGATION y bloques sin impaciencia señalada.

2.2.7. Atendiendo una Solicitud para Bajar un Bloque

Esto ocurre cuando un nodo de llave o uno de índice considera necesario propagar un bloque dado, comenzando cuando un *peer* **A** recibe el comando GET KEY de un nodo de llave/índice o el comando GET KEY IP de un nodo de índice. Entonces **A** trata de bajar el bloque asociado a la llave KEY siguiendo el algoritmo estándar para obtener un bloque o contactando al *peer* con dirección IP directamente. En ambos casos, **A** debe mandar el comando QUEUE KEY PROPAGATION en vez del comando QUEUE KEY para indicar su propósito.

Cuando el *peer* **A** haya bajado el bloque asociado a la llave **KEY**, si la solicitud provino especificando la dirección IP del *peer* de dónde bajarla, en vez de notificar a los nodos de

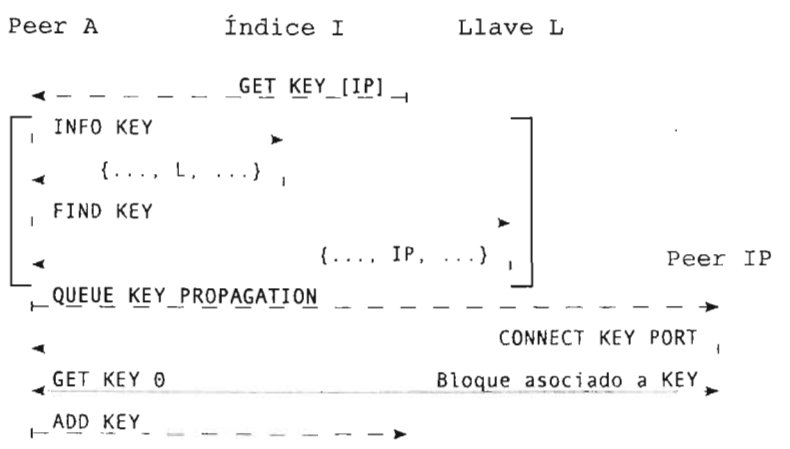


Figura 2.6: Propagación solicitada de bloques.

llave responsables de la llave KEY, **A** notifica al nodo de índice del que provino la solicitud (manda ADD KEY).

Como el éxito de este tipo de transferencias es crucial para el sano funcionamiento del canal, cada nodo que mande información (cada *peer*) debe de tener una bahía (slot) especial para mandar únicamente bloques de impaciencia PROPAGATION.

2.2.8. Agregando Programas al Canal

Este proceso comienza cuando un *peer* **B** le solicita a un nodo de índice **I** que agregue un programa (**I** recibe ADD KEY). Entonces **I** obtiene el bloque KEY del *peer* **B** y verifica que sea un bloque descriptor de programa[3] sintácticamente válido de un programa semánticamente aceptable[4].

En el caso de que el nodo de índice **I** decida aceptar el programa, **I** empieza a pedir a otros *peers* que bajen bloques del programa cuyo descriptor de programa es el bloque KEY (manda GET KEY IP). Conforme va obteniendo respuestas de qué nodos ya bajaron

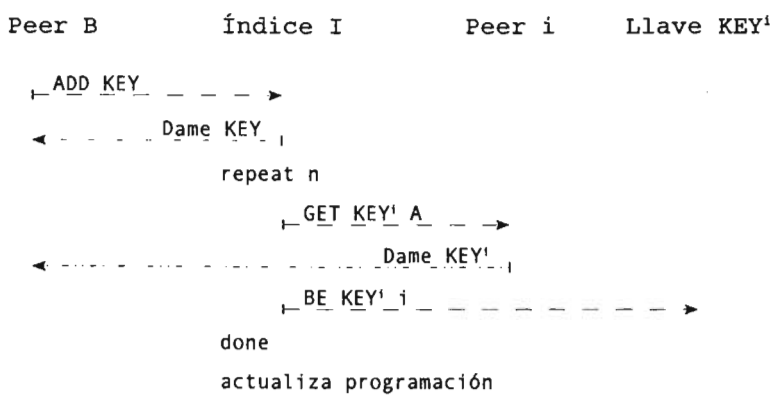


Figura 2.7: Agregando bloques al canal.

qué bloques (recibe ADD KEY), va pidiendo a otros nodos que obtengan los bloques recién propagados de sus nuevas fuentes. Este proceso se repite, hasta el punto en el que cada bloque asociado al programa es considerado estable (tiene suficientes réplicas) en el canal.

Después le pide a su módulo de seguridad que regenere los bloques de programación[3] y el bloque maestro[3], replicándolos exactamente de la misma manera hasta que son considerados estables.

Después, para cada llave nueva KEY (asociada a cada bloque recién agregado al canal), le pide a un conjunto pequeño de nodos, de entre su lista de nodos potenciales de llave, que se responsabilize de almacenar la información acerca de esa llave KEY (mandándoles BE KEY IP1 [IP2 [...]]).

Finalmente, el nodo de índice **I** le informa a sus réplicas acerca de las nuevas llaves que él agregó (mandando ADD KEY IP). Después bloquea el bloque maestro[3] en sus réplicas (manda LOCK, recibe LOCKACK); modifica los nodos de llave responsables del bloque maestro (manda ADD KEY IP) y desbloquea el bloque maestro en sus réplicas (manda UNLOCK).

2.3. Estabilización de la Red

Hasta aqui hemos construido una red completamente funcional, pero debido a el enfoque altamente disconexo que se usa en la comunicación entre los nodos, en ella todavía hay algunos agujeros que en un escenario muy dinámico pueden crear incongruencias en la información que poceen los distintos nodos, que si no son corregidas inclusive pueden destruir el canal. Para prevenir eso, es necesario introducir algún tipo de protocolo de estabilización que verifique si hay alguna incongruencia en la información que poseen los nodos de la red y en caso de encontar alguna, la corrija.

Como todos los procesos de estabilización, éste también tiene que ser ejecutado por cada nodo de manera periódica a lo largo de toda su existencia. Así, cada uno de los procesos de estabilización descritos abajo sólo cubren una ejecución de todo el proceso de estabilización que cada nodo tiene que hacer.

2.3.1. Estabilización de *Peers*

Debido a que cada *peer* puede donar sólo una capacidad finita de almacenamiento, es necesario realizar la limpiesa periódica de los bloques cuyo uso ya expiró en el canal. Para hacerlo los *peers* van a seguir el siguiente procedimiento:

Cada *peer* va a bajar la programación actual y a registrar todos sus bloques nuevos que aparecen en la programación y por algún motivo aún no estan registrados. Mientras tanto, borra todos los bloques que tiene guardados que ni aparecen en la programación ni se los ha pedido almacenar un nodo de índice. Para los bloques que se le ha pedido almacenar, si después de un muy largo tiempo de espera estos bloques no aparecen en la programación, deben ser eliminados, ya que en este caso el *peer* debe asumir que hubo algún error en el proceso de agregación de un programa al canal.

2.3.2. Estabilización de Nodos de Llave

La estabilización de los nodos de llave tiene que rectificar dos situaciones posiblemente erróneas. En primer lugar, la existencia misma del nodo de llave, que pudo haber expirado junto con el bloque de cuya llave era responsable. Después, una vez que verifica la validez de su existencia tiene que completar la información que posee respecto a la llave de la que es responsable para poder brindar mejor información a los nodos que se la soliciten. Esto lo hace realizando el siguiente procedimiento:

Cada nodo de llave también verifica la programación (obtenida por su capa de *peer*) para ver si tiene algún sentido su existencia y si no la tiene, se eliminará, a menos que haya sido recientemente contactado por un nodo de índice para ser un nuevo nodo de llave.

En el último caso, su existencia también va a estar sujeta a un muy largo período de espera. Si después de este período de espera no aparece en la programación, va a asumir que hubo algún problema en el proceso de agregación de algún programa.

En el caso especial en que el nodo de llave sea responsable por el bloque de la llave maestra, su capa de *peer* tiene que bajar la programación usando las nuevas fuentes del bloque maestro (si es que hay tales) y si nota algún cambio va a tirar todas las fuentes anteriores que tienen el bloque atrasado.

Cada nodo de llave también tiene que preguntarle a algún nodo de índice cuáles deberían ser sus (del nodo de llave) réplicas (mandando `INFO KEY`), para después verificar cuáles de sus réplicas siguen vivas y solicitar la información que sus réplicas poseen respecto a la llave de la que él es responsable, junto con los nodos que ellos consideran sus (de las réplicas de los nodos de llave) réplicas (mandando `SYNC KEY`).

2.3.3. Estabilización del Nodo de Índice

Los nodos de índice tienen que actualizar la información referente a la localización de los nodos de llave que almacenan. Esto lo hacen desechando la información asociada a las llaves que recien expiraron y completando la información sobre las que siguen vigentes.

Cada nodo de índice también va a consultar la programación (obtenida en su capa de *peer*) para ver si tiene algún sentido seguir almacenando información de cada llave que conoce. En caso de que la respuesta sea negativa, borra la información que tiene almacenada respecto a esa llave, a menos que sea la llave de un bloque asociado a un programa que está tratando de agregar al canal.

En el último caso, el mantenimiento de la información respecto a esa llave está sujeto a la disponibilidad de todos los bloques asociados al programa del que forma parte el bloque con esta llave en la red. Con esto es todavía posible completar la agregación del programa al canal.

Además cada nodo de índice revisará cuáles de sus réplicas siguen estando vivas y les preguntará qué llaves tienen almacenadas, cuáles son los nodos de llave asociados a esas llaves y quiénes son los que ellos consideran sus réplicas (mandando `SYNC` y `SYNC KEY`).

[1]Esto es nesesario para poder asegurar la inviolabilidad del contenido. Ver Módulo de Seguridad para más información.

[2]Bootstrap: término ocupado para definir el proceso por el que pasa una red desde su creación hasta que llega a la etapa funcional.

[3]Por ahora sólo otro bloque. Ver Módulo de Seguridad para más información.

[4]Por ahora sólo obtiene una respuesta de tipo verdadero/falso basado en las etiquetas del programa. Ver Módulo de Seguridad para más información.

Capítulo 3

Módulo de Seguridad

3.1. Descripción

Al igual que el módulo de red, el llamado módulo de seguridad va a operar en cada uno de los nodos que compongan un canal, proveyendo servicios a estos nodos de interpretación y validación del contenido. Más específicamente, sus funciones consisten en la realización de varias tareas, de entre ellas destacan:

- Interpretar los bloques que están almacenados en el canal; leer el bloque maestro, entender su estructura y proseguir leyendo la información almacenada en los bloques que están ligados desde el bloque maestro o algún otro de los bloques ya leídos y entendidos, que pueden ser tanto *bloques de contenido* (programas), como *bloques de meta información* (horario, información sobre programas, etc.). Esto lo va a hacer de una manera muy similar a la que usan los sistemas de archivos jerárquicos para interpretar archivos y directorios, a partir de un conjunto de bloques almacenados en un medio de almacenamiento sólido como un disco duro.

- Una vez que la información almacenada en el canal sea accesible a los nodos, el módulo de seguridad va a regir ciertas decisiones que los nodos tengan que tomar basándose en la información que interprete de los bloques. Estas decisiones, generalmente van a servir para proteger a los usuarios del sistema, en el sentido de evitarles la pena de ver material ofensivo y/o de calidad inaceptable, o bien para proteger al canal de abusos por parte de los nodos que quieran agregar información al canal.

Es muy importante hacer notar desde el principio que para la interoperabilidad entre los nodos que compongan un canal, es igual de importante que haya una especificación de qué función *hash* y esquema de firmas digitales deben usar los nodos, que el protocolo de red mismo. Por lo que, nuestra propuesta es usar SHA-1[6] y RSA[7] como función *hash* y esquema de firmas digitales respectivamente, debido a la gran seguridad que han demostrado a lo largo de sus respectivos tiempos de vida. Con esto, cualquier mención de un valor *hash* a lo largo de este capítulo se refiere específicamente a los valores *hash* obtenidos mediante la función *hash* SHA-1. Del mismo modo, cualquier mención de llaves públicas y firmas digitales se refieren específicamente a las llaves públicas y firmas digitales que provee el algoritmo de llave pública RSA.

3.2. Estructura de Bloques

Como se ha visto antes, la información almacenada por los nodos de una canal no es más que un conjunto de bloques; cada uno de los cuales posee un identificador que llamamos **llave**. Teniendo este tipo de información, para satisfacer la necesidad de búsquedas sobre el contenido e interpretación del mismo, vamos a organizar los bloques a modo de un sistema de archivos jerárquico. Para lo que a continuación daremos una calsificación de los posibles tipos de bloques y en 3.3 veremos como se organiza toda la información de un

canal usando esos tipos de bloques.

Todos los bloques (ya sean de contenido o de meta datos) que se almacenen en el canal representarán simplemente una secuencia reducida de *bytes* (de no más de 1MB para facilitar su transferencia). La llave asociada con cada bloque va a ser la obtenida sacando el *hash* del contenido de todo el bloque, para todos los bloques con excepción del bloque maestro que va a ser identificado con el *hash* de la llave pública del canal (probablemente la llave pública de su creador) para que su nombre sea fijo a lo largo de la existencia del canal.

3.2.1. Bloques de Contenido

Un **bloque de contenido** es simplemente la secuencia sin ningún tipo de estructura (con respecto del sistema) de *bytes* que corresponde a la parte del programa que le toca almacenar.

3.2.2. Meta Bloques

Los **meta bloques** (bloques que contienen meta información) son documentos válidos de `XML` con información referente a los datos de un programa o alguna otra parte del canal, o bien con ligas a otros bloques (representadas con llaves de bloques a los que apuntan) que forman la estructura jerárquica en la que se va a almacenar toda la información del canal. Con esto se va a crear un árbol jerárquico muy parecido al sistema de archivos `UNIX File System V7`.

Los meta bloques se dividen en los siguientes tipos:

Bloque Maestro

El bloque maestro es la raíz de toda la información almacenada en el canal y sobre el canal y está compuesto de:

1. Llave pública del canal
2. Liga a un bloque de permisos (la llave asociada a un bloque de permisos)
3. Liga a un bloque descriptor del canal
4. Firma digital de los campos (1) al (3)
5. Liga a un bloque de programación
6. *Hash* de todo el bloque con la parte donde va este campo sustituída por `0x00`s

Los campos (1) al (3) sólo tienen que ser modificados por el creador del canal (o cualquiera con la llave privada del canal o el derecho moral y permiso[1] para modificar sus metadatos), por lo que (4) no perjudica en absoluto la naturaleza dinámica del canal mientras que garantiza la autenticidad de los campos (1) al (3).

Pero desafortunadamente, el campo (5) debe ser modificado con la adición de cada programa por lo que el creador del canal no puede firmarlo y su autenticidad dependerá de qué tan adversos son los nodos de índice. Finalmente, el campo (6) nos asegura que al menos el bloque maestro no fue accidentalmente modificado debido a una falla en el medio de almacenamiento o transmisión.

En el apéndice B.1 se puede ver como se organizaría esta información en un esquema de `XML`.

Bloque de Permisos

El bloque de permisos contiene la información acerca de quién y cómo puede controlar el canal, es decir, quién pude modificar la meta información del canal o de los programas y quién puede agregar los programas en sí.

Este bloque contiene la siguiente información:

1. Llave pública del canal.
2. Reglas acerca de quién tiene permiso para modificar la información del canal y qué puede modificar de esta información. Esto puede ser alguna de las siguientes opciones:
   - Cualquiera
   - Por Certificado
   - Con un conjunto de llaves conocidas.
3. Reglas acerca de quién tiene permiso de modificar la programación del canal y qué puede modificar de esta programación. Esto puede ser alguna de las siguientes opciones:
   - Cualquiera
   - Por Certificado
   - Con un conjunto de llaves conocidas.
4. Firma digital del bloque completo por el creador del canal.

En el apéndice B.2 se puede ver como se organizaría esta información en un esquema de XML.

Bloque Descriptor de Canal

Este bloque contiene la declaración de intenciones del canal respecto a qué tipo de contenido espera proveer.

Más precisamente, contiene la siguiente información:

1. Nombre del Canal
2. Breve descripción del Canal
3. Palabras clave asociadas al Canal
4. Etiquetas generales acerca del contenido que se espera encontrar en el canal. Estas pueden incluir información acerca de la audiencia destinada, los tipos/generos de programas esperados en el canal, los idiomas en los que se espera vengan los programas en el canal o la calidad de señal (imagen y sonido) que se espera tengan los programas del canal.
5. Datos técnicos del canal, estos principalmente incluyen información acerca de cómo maneja un canal su contenido e incluyen información acerca de:
   - Parámetros de la ventana de programación
   - Restricciones sobre etiquetas[2] de programas.

En el apéndice B.3 se puede ver como se organizaría esta información en un esquema de XML.

Bloque de Programación

El bloque de programación es el responsable de almacenar la información acerca de qué programas se van a pasar y cuándo van a pasar. En caso de que un bloque de progra-

mación crezca demasiado, éste puede ser dividido en dos y los bloques pueden ampliarse para formar una especie de lista ligada.

Estos bloques forman una lista de entradas cada una con la siguiente información, que es la mínima necesaria para poder servir como una guía de programación:

- Hora de inicio programada
- Hora de terminación programada
- Nombre del Programa
- Liga al descriptor[2] del programa

En el apéndice B.4 se puede ver como se organizaría esta información en un esquema de XML.

Bloque Descriptor de Programa

Este bloque es para un programa lo que el bloque descriptor de canal es para un canal. Contiene la declaración de los parámetros tanto técnicos como semánticos del programa que está describiendo, así como la lista de los bloques de contenido que componen al programa y posee la siguiente estructura:

1. Nombre del Programa
2. Clasificación
3. Palabras clave asociadas al programa
4. Algunos datos de carácter semántico sobre el programa, estos pueden incluir información acerca de:

- Año en el que se realizó
- País de Origen
- Género/Categoría
- Director/Realizador
- Elenco

5. Algunos datos de carácter técnico sobre el programa, estos deben incuir:
   - Duración
   - Idiomas de pistas de audio, si es que hay alguna
   - Idiomas de pistas de subtítulos, si es que hay alguna
   - Especificaciones de la señal de video
   - Especificaciones de las señales de audio
6. Una lista ordenada de los bloques que componen el programa

En el apéndice B.5 se puede ver como se organizaría esta información en un esquema de XML.

3.3. Organización de la Información en un Canal

Teniendo este conjunto de bloques podemos crear un árbol conformado como un sistema de archivos jerárquico. La raíz de este árbol sería el bloque maestro y los demás bloques, sus demás nodos y hojas. Esto lo podemos ilustrar con el diagrama 3.1.

De este modo un Programa se compone de su encabezado (bloque descriptor de programa) y una lista de bloques que contienen la señal del programa en cuestión (bloques de contenido).

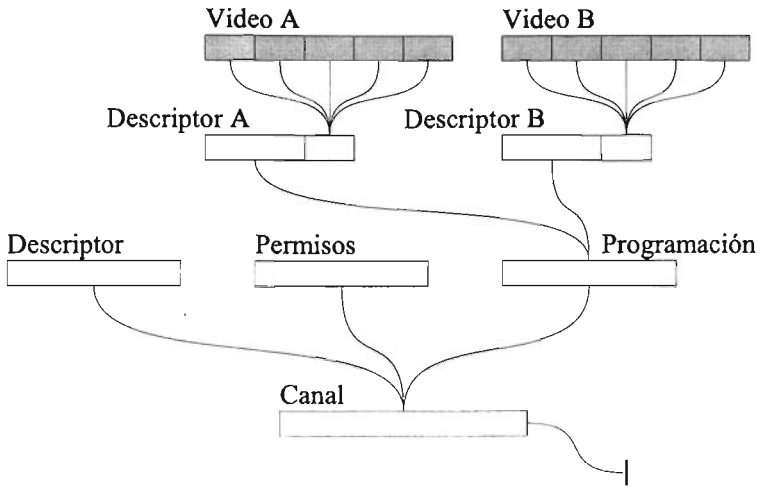


Figura 3.1: Organización de la información en un canal.

Los programas a su vez forman la programación del canal que está compuesta de una lista ligada de bloques de programación.

El canal se compone de su descriptor (bloque descriptor de canal), sus especificaciones de seguridad (bloque de permisos) y la programación en curso (la lista ligada de bloques de programación).

Gracias a que en esta estructura la llave ligada a todos los bloques a excepción del bloque maestro es obtenida por el *hash* de su contenido, podemos estar seguros de su integridad tanto conciderando la posibilidad de errores accidentales en la transmisión o almacenamiento de los bloques, como la excistencia de nodos malignos en el sistema. De hecho debido a que sólo los nodos de índice pueden agregar contenido y modificar la programacion, son los únicos que podrían agregar contenido corrupto al sistema, pero gracias a que necesitamos muy pocos nodos de índice, y cuando necesitamos agregar uno lo escogemos de una lista de nodos bien portados, esto hace difícil para un nodo mal intencionado

corromper el contenido de un canal `Trifs`.

3.4. Seguridad de Contenido

Esta es la parte por la que el Módulo de Seguridad recibe su nombre y es la más importante de sus funciones.

Aquí cuando hablamos de seguridad de contenido nos referimos a dos cosas: primero, asegurar que el canal no reciba contenido no apto según sus especificaciones. Esto es, tanto a nivel semántico (que no pongan pornografía en un canal infantil, ni La Sirenita en uno pornográfico), como a nivel técnico (que no pongan video en formato DV-1 en un canal diseñado para usuarios de *modem*). Segundo, cuidar al usuario final filtrando el contenido del canal según las especificaciones de cada usuario del sistema. Esto último se logra validando el contenido contra las especificaciones de los usuarios en sus propios nodos.

3.4.1. Seguridad de Usuarios

De esta parte de la seguridad se encarga en módulo de seguridad asociado a cada nodo de la red y su función es filtrar la información que el dueño de este nodo considere inapropiada para él.

Esto funciona mediante la aplicación de ciertas restricciones sobre los programas, basándonos en cualquier subconjunto de las etiquetas asociadas a dichos programas.

Debido a la flexibilidad del sistema de etiquetas asociadas a los programas, podemos personalizar cualquier aspecto de la programación que vamos a obtener; desde opciones tan detalladas como ver sólo los cortometrajes realizados en los años 1978 y del 1946 al 1952 en Tuvalú y que vengan con calidad DVD y audio de 5.1 canales, hasta simplemente

decir que no nos gustan las películas de terror o que el cine silente es aburrido.

Para lograr esto, cada nodo va a mantener una lista de parejas de expresión regular y un identificador PASS/REJECT, ésta será evaluada en orden empezando por el primer elemento de la lista hasta encontrar la primera entrada cuya expresión regular aplique con lo que tomará la decisión asociada a esa entrada. En caso de que ninguna entrada de la lista explícita compagine, suponemos que hay una última que dice * PASS.

3.4.2. Seguridad del Canal

Al igual que la seguridad de usuarios esta parte se va a encargar de filtrar el contenido que alguien intente agregar al canal. Como los encargados de aceptar programas son los nodos de índice, es en ellos en los que va a funcionar esta parte del módulo de seguridad.

Su trabajo, a su vez se va a dividir en dos funciones: primero, identificar y validar a usuarios que quieran agregar contenido al canal; y segundo, validar el contenido que se quiera agregar para que cumpla con las especificaciones del canal. Esto último se hará de exactamente la misma manera como funciona la seguridad de usuarios, sólo que basándose en reglas especificadas por el creador del canal en el bloque descriptor del canal.

La validación de usuarios va a seguir el siguiente procedimiento: en caso de que cualquiera pueda agregar contenido, todos los usuarios van a ser válidos; en caso de que la validación sea por certificados, el usuario tiene que presentar un certificado válido expedido y firmado por alguna de las llaves conocidas[3] por el canal (la llave del creador u otras en el bloque de seguridad); en caso de que sea por llave conocida, tiene que firmar su propio certificado con algunas de las llaves conocidas, esto para reducirlo al caso de autenticación por certificados.

De este modo podemos restringir que sólo personas confiables agreguen contenido al

canal, por lo que un nodo de índice puede estar seguro que los parametros asociados al programa que se está agregando son confiables y siendo que después el contenido no puede ser modificado, también los demás nodos podrán tomar deciciones sobre los parámetros asociados a los programas y filtar los programas en base a estas decisiones.

[1] Ver bloque de permisos.
[2] Ver bloque descriptor de programa.
[3] Llave almacenada en el bloque de permisos del canal (*well known keys*).

Capítulo 4

Módulo Usuario

4.1. Descripción del Módulo de Usuario

La funcionalidad del módulo de usuario reside en proveer una abstracción del protocolo desarrollado en la presente tesis para los usuarios del sistema que implementen dicho protocolo. Como tal es irrelevante al protocolo en sí. Por lo que su especificación se deja a los desarrolladores del sistema que implanten el protocolo desarrollado aquí.

Pero hablando en términos generales, su funcionalidad debe incluir la comunicación entre el módulo de red y el usuario, filtrando la información a través del módulo de seguridad para obtener el contenido del canal y la posibilidad de agregar contenido al módulo de red para poder iniciar el procedimiento de la agregación del contenido al canal.

Una posible instanciación de un módulo de usuario que cumpla estas funciones podría dividir el modulo de usuario en dos partes, la funcional y la visual. Este diseño del módulo de usuario también se podría extender a usar un entorno de cliente-servidor adentro de una red local para que haya sólo un nodo `Trifs` usando la conexión a internet independientemente de si hay varias personas interesadas en ver contenido de canales `Trifs`. Esta

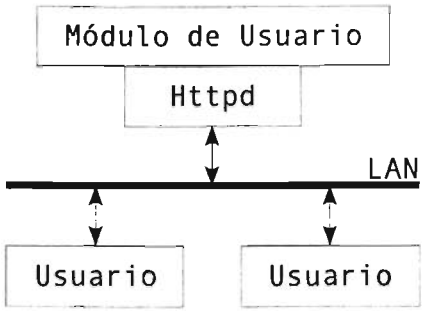


Figura 4.1: Propuesta para un Módulo de Usuario.

propuesta se ilustra en la figura 4.1.

Esta propuesta se podría realizar usando un servidor `http` del lado del servidor del módulo de usuario que podría presentar la programación, opciones y seleccion de programa para ver mediante una pagina `html` con un diseño tipo las guías de programación estándar como se muestra en la figura 4.2. Así mismo, la transmisión del contenido en sí podría realizarce usando el mismo servidor de `http`, ya que en una red local disponemos de suficiente ancho de banda para que eso sea un modo de distribución sumamente razonable, por lo que simpleme se tendría que lanzar cualquier reproductor de video desde la misma página de la programación.

Como ya se mencionó antes, el diseño e implementación del módulo de usuario es un tema fuera del alcance de esta tesis, por lo que la propuesta arriba presentada no es más que la idea no definitiva del autor de esta tesis respecto de cómo instanciar este módulo. Sin embargo, esta propuesta tiene ciertas ventajas. El uso del paradigma cliente-servidor permite un mayor aprovechamiento del enlace a Internet ya que habría sólo un nodo `Trifs` centralizado, por lo que no se desperdiciaría ancho de banda ni del enlace ni de la red `Trifs` en sí transmitiendo repetidas veces varios bloques (al menos los que contienen la progra-

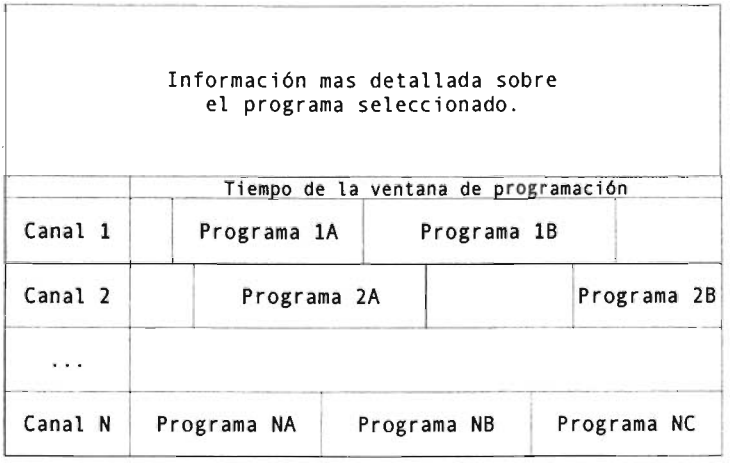


Figura 4.2: Propuesta de Interface del Módulo de Usuario.

mación). Otra ventaja innegable, es que gran parte de lo que necesitamos para realizar esta propuesta ya está hecho, del lado de los clientes no tenemos que agregar nada y sólo del lado del servidor tenemos que integrar las funcionalidades que debe proveer el módulo de usuario de `Trifs` con algun servidor `http`. Eso último provee la gran ventaja del gran ahorro de tiempo y la estabilidad y seguridad implicada por el uso de tecnologías/componentes tan probados como los que se propone utilizar.

Capítulo 5

Uniendo Todo

Ahora que tenemos especificados todos los módulos por separado podemos explicar el funcionamiento de `Trifs` como un sistema completo. En la figura 5.1 tenemos un nodo `Trifs` que se compone de sus módulos de red, seguridad y usuario. Este nodo esta conectado (en el sentido de poder mandar información a otros nodos si es nesesario) con otros nodos `Trifs` usando la Internet a través de su módulo de red. A su vez, el módulo de red intercambia información con el módulo de seguridad para regir algunos aspectos de su funcionamiento, por ejemplo el almacenamiento del contenido de la red `Trifs`; y el módulo de usuario al cual le sirve el contenido solicitado por éste. En este esquema se está asumiendo una organización del módulo de usuario como la planteada en el capítulo anterior, por lo que podemos tener varios usuarios usando el mismo nodo `Trifs` y obteniendo contenido de él.

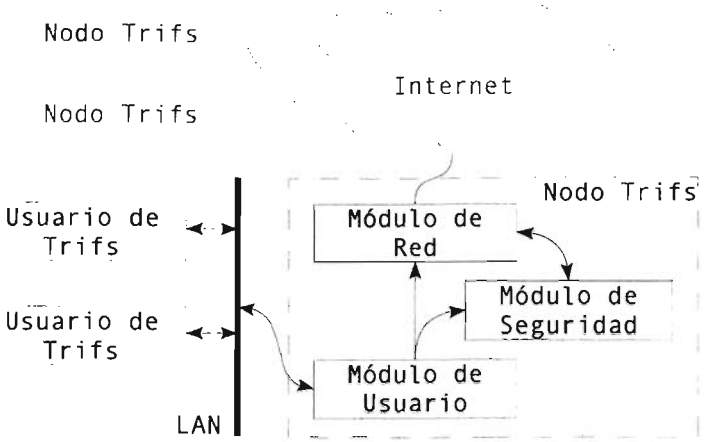


Figura 5.1: Un nodo `Trifs` completo.

5.1. La Vida de un Canal

Veamos a continuación cómo interoperan todas las partes que cubrimos en los capítulos anteriores para asegurar el buen funcionamiento de un canal a lo largo de todas las fases de su existencia.

5.1.1. Infancia

Por infancia de un canal entenderemos las etapas por las que pasa desde que su creador diseña sus especificaciones y requerimientos, hasta el punto en el que su existencia se independiza de la accesibilidad del nodo del creador del canal. En ella el canal atraviesa por las siguientes etapas:

- El creador del canal decide el crear un canal bajo los requerimientos que a él le parezcan convenientes.

- El creador genera las llaves pública y privada del canal y especifica los requerimientos decididos arriba usando el sistema de etiquetas de `Trifs`.

- El creador levanta su nodo con la información especificada. En este momento su nodo va a fungir tanto de nodo *peer* para los bloques especificados, como de nodo de llave/índice para la información de los bloques que almacena.

- El creador posiblemente agrega algún programa al canal usando su módulo de usuario. Esto con el fin de hacer más atrayente para otros el entrar a su canal con lo que se acorta la etapa de infancia.

- Conforme otros participantes van entrando al canal, el nodo del creador y posteriormente los nodos que ya entraron al canal le solicitan a estos participantes que se conviertan en nodos de llave/índice y/o almacenen bloques del canal.

- El creador mantiene vivo su nodo al menos hasta que su canal llega a la etapa de adulto.

Notas:

Aquí el único detalle técnico delicado es el referente al último punto. Es el cómo puede el creador saber si su canal ya entró a la etapa de adulto o si todavía sigue en la infancia. Pero gracias a que el nodo del creador, va a ser un nodo índice, este va a poseer una visión global de toda la red que conforma su canal, así mismo, va a recibir los informes de calidad de servicio y solicitudes de ayuda en la propagación de los nodos de llave. Y tomando esto en cuenta, es razonablemente sencillo decidir que tan cerca de ser considerado adulto es un canal.

5.1.2. Etapa adulta

En esta etapa es en la que va a pasar la mayor parte de su vida un canal. También es la etapa en la cual va a sufrir la mayor parte de sus adaptaciones. Esto, ya sea en cuanto a la topología de la red que lo conforma, o en cuanto a modificaciones de su contenido y/o especificaciones. A diferencia de la infancia, en la que trataba de superarla lo antes posible, en esta etapa se tiene que intentar quedar el mayor tiempo posible.

A lo largo de esta etapa pueden pasar los siguientes procesos:

- Al entrar un nodo al canal, el módulo de red se encarga de integrarlo tanto a nivel de almacenamiento, como a nivel participativo.
- Al salir un nodo del canal, los protocolos de estabilización de los nodos para los que su salida fue relevante restablecen con ayuda de otros nodos la funcionalidad que el nodo que se salió proveía.
- Se agrega contenido al canal mediante el módulo de usuario, el que primero almacena la información en la capa de *peer* del módulo de red. Y después le dice al módulo de red que inicie las negociaciones con algún nodo índice para la aceptación del contenido al canal.
- Los acuerdos de qué nodos van a cumplir qué funciones, se deciden en el módulo de red y sin ninguna intervención tanto de otros módulos como de usuarios de los nodos.
- Las modificaciones a las especificaciones del canal, también se hacen a través del módulo de usuario. Este prepara los bloques actualizados y las credenciales necesarias para la transacción, las almacena en la capa de *peer* del módulo de red, y le dice

a éste que comience las negociaciones con algún nodo de índice para llevar acabo las modificaciones.

- La eliminación de contenido va a acontecer por si sola al expirar de la programación los bloques que conformen la información y contenido del canal. De esto se encargan los protocolos de estabilización de la capa de red.

Notas:

Debido que a la topología tan flexible que usa esta red, no puede haber ningún problema con entradas simultáneas de nodos. Por lo que nuestra única preocupación son las salidas repentinas y masivas que puedan surgir. Pero debido a la aleatoriedad en el proceso de elección de los clones de un nodo de llave/índice y de los nodos que almacenen un bloque dado, se espera que inherentemente tiendan a distribuirse bien en el sentido geográfico, por lo que el umbral de la cantidad de réplicas necesarias no debe ser muy elevado.

5.1.3. Vejez

A diferencia de las dos etapas anteriores, en las que hay alguien (el creador del canal) quien decide que se entre en esas etapas, y que todo el canal a la vez entra a cada una de ellas. En esta etapa solo se puede entrar cuando se acaba la vida útil del canal, esto ya sea por fallas o por falta de interés/participación. Por lo que cada uno de los nodos que siga interesado en el canal, al detectar que no logra obtener la información que debería poder obtener del canal si estuviera en su sano funcionamiento, decide degradarlo al estado de viejo y en el caso de que después de algún tiempo de vejez no se rejuvenezca, simple y sencillamente matarlo (en el sentido de eliminar su nodo permanentemente del canal).

Debido a que en este estado el canal no es funcional (ya sea parcial o totalmente), los

eventos que pueden pasar en el son muy limitados.

- Un nodo declara viejo el canal en el que está, si desde su conocimiento local del canal no logra considerarlo funcional. Lo que puede ser provocado por imposibilidades de búsquedas o aislamiento de bloques importantes de la información del canal.
- Un nodo rejuvenece el canal en el que está, si después de algún tiempo de espera la funcionalidad del canal regresa.
- Un nodo declara muerto un canal cuando pierde la esperanza de que el canal rejuvenezca.

Notas:

Al igual que con las cuestiones de enfermedades y decesos clínicos, es sumamente delicado el especificar los parámetros para esta etapa de la vida de un canal. En general, se espera que un nodo no va a presenciar esta etapa para los canales que le sean interesantes. Y en caso de que le toque presenciarla, lo razonable sería que declare sus umbrales de sanidad y muerte proporcionalmente al interés que tiene en el contenido que el canal le pueda ofrecer. Pero tampoco es aconsejable el rendirse muy pronto debido a que una falla local en los canales de transferencia de datos, localmente pueda hacer parecer catastrófico el estado de salud de un canal, mientras que el canal se encuentre en perfecto estado desde cualquier otro punto de la red.

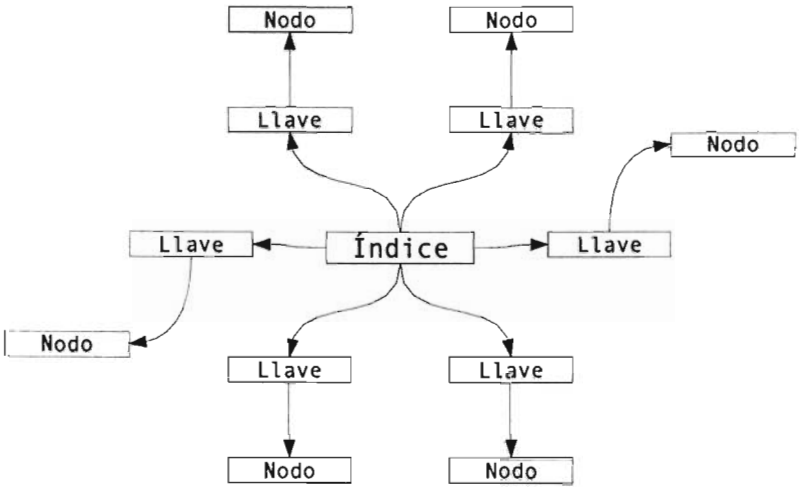


Figura 5.2: Topología sin replicación

5.2. Desempeño

Debido a la gran flexibilidad y naturaleza disconexa del protocolo propuesto en esta tesis, es muy difícil estudiar el rendimiento de una red `Trifs` directamente.

5.2.1. Modelo para la Topología de `Trifs`

El modelo de la figura 5.2 cuenta con una topología de estrella en cuyo centro se encuentra un servidor de índice conectado con los L servidores de llave, los que a su vez están conectados con los N nodos que almacenan los bloques (información almacenada en la red) asociados a cada llave.

Este modelo con $N = L$ es claramente equivalente a una red `Trifs` sin ningún tipo de replicación y en el que cada nodo de `Trifs` hará las funciones de al menos un nodo de los propuestos por la topología de la figura 5.2.

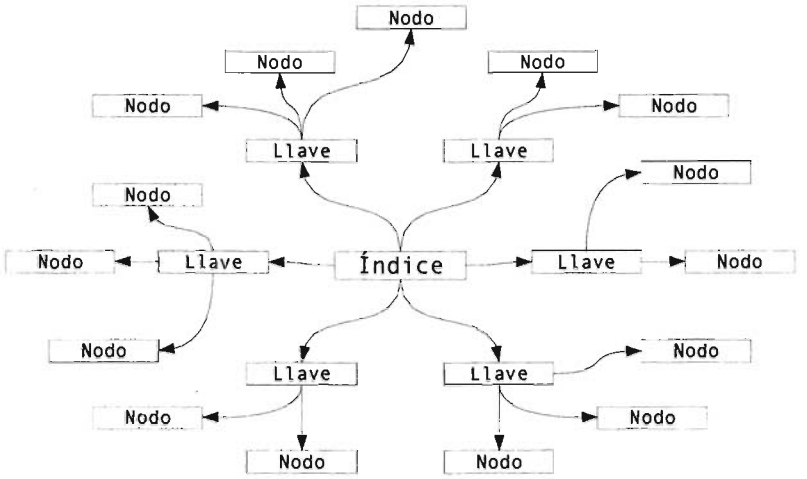


Figura 5.3: Topología con replicación de información

Teniendo este modelo, lo podemos extender al presentado en la figura 5.3, en el que introducimos la replicación de la información almacenada en el canal (los bloques) replicada entre los N nodos (con $N \geq L$).

Este caso es claramente igual a la red `Trifs` sin replicación de nodos de metainformación (i.e. nodos de llave y nodos de índice), en la que cada nodo de `Trifs` va a desempeñar las funciones de varios nodos propuestos por la topología de la figura 5.3.

A su vez, podemos extender el modelo representado en la figura 5.3 para que incluya la replicación de la metainformación. Para ello en el centro de la gráfica generada por nuestra topología vamos a tener varios nodos de índice con exactamente la misma información, en vez de sólo uno. De la misma manera, para cada bloque vamos a tener varios nodos de llave en lugar de uno, con exactamente la misma información. Esto se muestra en la figura 5.4.

Esta topología (figura 5.4) es un excelente modelo de una red `Trifs` "desenrollada" en la que cada nodo de `Trifs` va a desempeñar las funciones de varios nodos propuestos por

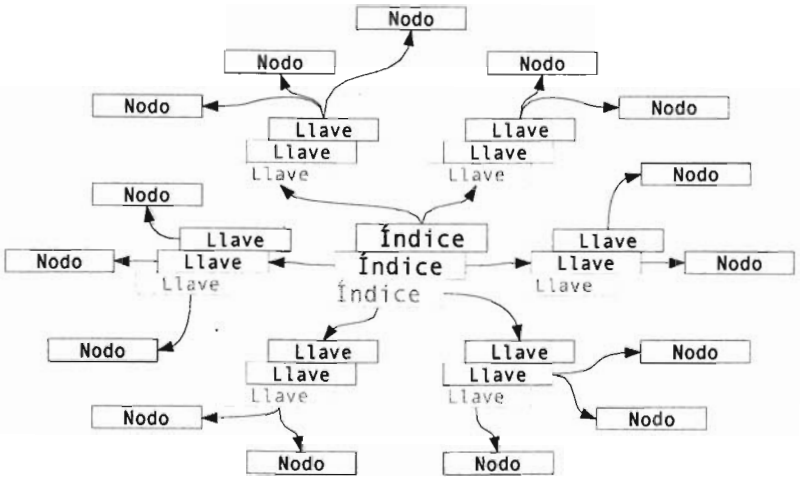


Figura 5.4: Topología con replicación completa (información y metainformación)

la topología de la figura 5.4.

Con esto, podemos formalizar el siguiente modelo para una red `Trifs`:

Un modelo para una red `Trifs` con parámetro de replicación d y K bloques de información, es una gráfica con la topología asociada al modelo de la figura 5.4 con $I \geq d$ servidores de índice (nodos de índice), $L \geq K \times d$ servidores de llave (nodos de llave) y $N \geq K \times d$ nodos (*peers*).

5.2.2. Desempeño de `Trifs`

A continuación estudiaremos el desempeño de `Trifs` en varios escenarios y usos posibles.

Búsquedas

Sabiendo la llave del bloque que uno quiera obtener, una búsqueda de un bloque en `Trifs` consiste en preguntarle a algún nodo de índice qué nodos de llave poseen información sobre esa llave, después preguntarle a alguno de entre esos nodos de llave donde es donde está esa información. Esto evidentemente nos lleva $O(1)$ pasos.

Mientras que los algoritmos P2P "puros", en el mejor caso conocido (`Chord` [9], `Kademlia` [11]), nos llevan $O(log(N))$ y en ocasiones (`Gnutella`[4]) $O(N)$.

A su vez, los P2P híbridos (`FastTrack` [14], `Gnutella2`, `eDonkey2000`) requieren de un paso para la búsqueda de contenido popular mientas que son $O(I)$ (recordemos que I es el número de servidores de índice). Este caso se estudia en [16] con detalle.

Escalabilidad

En cuanto a escalabilidad, el único problema que puede presentársenos es debido a la casi centralización que tenemos en los nodos de índice. Así que, estudiemos la utilización promedio de estos.

Supongamos que tenemos un canal `Trifs` en el que sus integrantes obtienen un promedio de B bloques por segundo. En este caso cada nodo de índice va a obtener un promedio de $\frac{B \times N}{I}$ peticiones y una utilización de red de $\frac{B \times N}{I} \times 50$ bytes/segundo[1].

Poniendo un ejemplo, esto se traduce a 10,4 peticiones por segundo y 520 bytes/segundo por nodo de índice. Esto en una canal con $N = 10000$ usuarios, $I = 10$ nodos de índice, contenido de 1 mbps y en el que cada usuario ve dos horas de contenido al día.

A continuación lo comparamos con otras redes P2P; en el caso de las P2P "puras", las modernas (`Chord`, `Kademlia`) suelen ser altamente escalables ya que que cada nodo guarda una muy pequeña parte de la topología y organización de la red, por lo que no es relevante

qué tan grande es la red en su totalidad. A su vez, las redes P2P híbridas, suelen presentar una escalabilidad semejante a `Trifs` ya que también están basadas en la idea de tener un conjunto pequeño de nodos que mantengan la información global de la red. Éstas tienen escalabilidad lineal con respecto del protocolo aquí propuesto, siendo éste de los mejores en esta clase debido a que la cantidad de información que manejan los nodos de índice es la mínima cantidad de información posible.

Efecto *Slashdot*

Para comprobar el desempeño de nuestro sistema con respecto al efecto *slashdot* es importante notar dos cosas: el comportamiento y disponibilidad de los nodos de llave/índice y la posibilidad de obtener información durante la manifestación de dicho efecto. Para eso, supongamos que el efecto dura s segundos, durante los cuales se crea un interés por un programa que consta de m bloques.

Como por cada petición a un nodo de llave se tiene que hacer una consulta previa a un nodo de índice, nos es suficiente ver la carga que sufrirán los nodos de índice durante la manifestación del efecto. Ésta será de no mayor a $\frac{N \times m}{I \times s}$ peticiones por segundo y $\frac{N \times m}{I \times s} \times 50$ bytes/segundo de transferencia[1].

Esto se traduciría, siguiendo nuestro ejemplo anterior en 125 peticiones por segundo y una utilización de 6,1 kilobytes/segundo para cualquier nodo de índice, suponiendo que el efecto dura las dos horas de la transmisión de un programa a 1 mbps (900 partes de 1 megabyte) de dos horas.

A su vez, la disponibilidad de los bloques en el canal crece linealmente con respecto al interés que hay en ellos, si los bloques son de tamaño t y la velocidad promedio de los canales de comunicación es v, entonces el canal puede duplicar la disponibilidad de cualquier bloque dado cada $\frac{t}{v}$ tiempo.

Eso en nuestro ejemplo equivaldría a que cualquiera de los m bloques participantes en el efecto *slashdot* duplica su disponibilidad cada 64 segundos, si suponemos que la velocidad promedio de los nodos es $\frac{1}{8}$ del ancho de banda del programa que están tratando de ver (i.e. $v = 128$ kbps).

Este aumento lineal en la disponibilidad de la información conforme a su popularidad nos asegura que no habrá un aumento significativo en las colas de espera de las personas que poseen alguno de los m bloques participantes en el efecto *slashdot*, por lo que no se ve afectada la disponibilidad de los demás bloques de la red.

Esto último, es debido a que si el tamaño de las colas de espera es q y n es la cantidad de réplicas que hay de un bloque dado, es necesario que en un instante dado al menos $q \times n$ personas se interesen en este bloque para duplicar el tamaño de las colas de espera de las n personas que tienen el bloque. Mientras que si este bloque es uno de los m participantes en el efecto *slashdot*, tiene que pasar $[log_2(N) - log_2(q \times n)] \times \frac{t}{v}$ segundos para que las colas de espera promedio de los nodos que almacenan bloques participantes en el efecto *slashdot*, lleguen a ser no mayores que dos veces el tamaño promedio de las colas de espera en condiciones normales del canal.

Regresando a nuestro ejemplo, esto se traduce en una espera de $8{,}29 \times 64 \approx 530$ segundos en el caso de que los 10000 integrantes decidan obtener alguno de los m bloques al unísono (donde unísono = un lapso menor a 10 minutos). Esto suponiendo que la longitud de cola promedio es $q = 4$ y hay tan pocas como $n = 8$ réplicas de cada bloque al iniciar el efecto.

En lo que respecta a los demás sistemas P2P, la única manera de superar el efecto *slashdot* es asegurando que el crecimiento en la disponibilidad (el número de fuentes) de la información se asemeje al crecimiento en la demanda (cantidad de peticiones en un intervalo de tiempo) de dicha información, por lo que los únicos sistemas que sí superan este

efecto, son los basados en el paradigma de aglomeramiento de archivos (`BitTorrent`[1], `eDoneky2000`[2]).

[1]Se esta considerando que un mensaje promedio va a ser de 50 *bytes* de longitud, 24 *bytes* para su encabezado y 26 *bytes* para su cuerpo.

Capítulo 6

Conclusiones

A lo largo de esta tesis se han construido las especificaciones para un sistema de distribución de contenido multimedia sobre una red *quasi* P2P.

En su diseño se trató de juntar varias de las ventajas que ofrecen distintos enfoques para la realización de distribución de contenido multimedia. Con esto se logró un sistema con una escalabilidad mucho mayor que la que ofrecen los basados en el paradigma Cliente-Servidor, pero manteniendo la posibilidad de ofrecer ciertas garantías de calidad de servicio. A pesar de su gran escalabilidad, este sistema también ofrece búsquedas de información muy rápidas (de complejidad constante, ver sección 5.2 para el análisis de desempeño), lo que representa un gran problema para los sistemas P2P puros. Para la distribución de la información usa los métodos introducidos por la idea del aglomeramiento de archivos, dándonos las ventajas que esta técnica posee (como rápida propagación de la información y resistencia al efecto *slashdot*), pero de una manera mucho más flexible para el manejo de la información (gracias a que directamente `Trifs` se encarga de manejar la información) y esquivando la centralización y punto único de fallo que el aglomeramiento de archivos suele presentar.

En cuanto a los requerimientos de infraestructura necesarios para poner en operación un canal `Trifs`, es importante notar el hecho de que este protocolo sí supone la homogeneidad en la mayoría de sus integrantes a pesar de que no es un P2P puro. Esto se logra debido a la granularidad de los bloques (unidad de información), por lo que se espera que haya más llaves que nodos físicos. De éste modo en promedio cada nodo va a ser tanto *peer*, como nodo de llave para algunas llaves. Por lo tanto a este nivel (tanto de *peers* como de nodos de llave), tenemos un P2P puro y sólo necesitamos un conjunto reducido de nodos que funjan de nodos índice que se van a separar del resto de los nodos. Considerando la no homogeneidad entre las capacidades de los nodos físicos, no es descabellado asumir que va a haber un conjunto reducido de nodos que puedan y estén dispuestos a desempeñar una función un poco más difícil que la de los demás nodos. Por lo que la infraestructura necesaria para la creación de un canal `Trifs` es similar a la necesaria para la creación de una red P2P pura, pero sin la necesidad de explícitamente pensar en los problemas que causa el hecho de que en el mundo real no todos los nodos tienen las mismas capacidades.

De hecho este diseño nos ayuda a controlar mucho mejor (de manera más natural) la carga que va a tener cada nodo, ya que aparte de decidir que tanto contenido es capaz de almacenar, puede decidir que tantos nodos de llave va a mantener.

También es importante considerar, que el protocolo propuesto aquí probablemente no es el óptimo para la tarea que se le impone, pero para su diseño se consideraron mucho más importantes la posibilidad de trabajar en un ambiente tan extremadamente caótico como lo es la Internet actual, y ademas brindarnos información respecto del funcionamiento de su red en un ambiente real. Lo primero es muy importante para su implantación e indispensable para lo segundo, que permite comprender realmente la dinámica de un sistema como el que se está creando y con ello tener una mucho mejor visión de todos los parámetros tanto para protocolos complementarios como para los que van a convertir en obsoleto al

especificado en esta tesis.

6.1. Desarrollo a Futuro

La idea a corto y mediano plazo, es primero que nada hacer la implementación de un sistema basado en las especificaciones, recomendaciones y el protocolo presentados en este documento. Además, crear algunos canales de prueba para adquirir cierta masa crítica y poder recolectar conocimiento de los parámetros con los que funciona el diseño aquí creado, en el mundo real, sus fallas, limitaciones y las mejoras que se pueden hacer en una posible versión futura, tanto del protocolo como de la implementación del mismo.

Una vez que se tenga recolectada esta información, se pretende desarrollar el sucesor de este protocolo, preservando todos los beneficios que tiene (i.e. distribución de carga, autocontrolable, etc.) y agregando los que se vea son necesarios o deseables, pero esta vez ya tratando que sea un verdadero Peer2Peer y tratando de corregir los *hacks* que requirió el módulo de seguridad de esta versión.

La idea a más largo plazo es desarrollar un sistema de archivos colaborativo P2P real basándonos en lo aprendido del desarrollo de esta red, que a pesar de que resuelva un problema real e importante aún es muy limitada y controlada debido a las especificaciones del problema. Este sistema de archivos deberá permitir todas las operaciones que se esperan de un sistema de archivos, proveer la seguridad que se espera de un sistema de archivos que esté implantado en un ambiente posiblemente hostil, pero que a su vez, dar ciertas garantías de accesibilidad a la información y tiempos de respuesta en las operaciones.

Apéndice A

Comandos del Módulo de Red

A.1. Comandos en la capa de *Peers*

Cada *peer* va a almacenar la siguiente información: un conjunto de nodos de índice (esto es necesario para estar conectado a la red, conectado en el sentido de poder obtener información sobre otros nodos y bajar contenido de la red), un conjunto de parejas (llave, bloque) de los bloques que tiene almacenado, y una cola de espera para *peers* que esperan obtener bloques de él.

La comunicación entre *peers* va a usar UDP *Datagrams* para todo, exceptuando la transferencia del bloque en sí. Esta última va a usar conexiones TCP sobre demanda.

El protocolo consiste de los siguientes comandos:

`SERVERS`

El nodo receptor del comando envía al nodo solicitante su lista de nodos de índice conocidos.

`QUEUE KEY [PRIO]`

El *peer* que envía el comando entra en la cola de espera del *peer* que lo recibe. El que

es solicitado que mande el bloque correspondiente a la llave KEY con nivel de impaciencia PRIO. Donde PRIO puede ser REALTIME, lo que significa que el *peer* que mandó el comando está mostrando el programa en tiempo real; SCHEDULED, lo que significa que el solicitante quiere obtener el programa para poder mostrarlo en algún futuro; o PROPAGATION, lo que significa que el solicitante fue pedido almacenar el bloque para asegurar su disponibilidad en la red.

CONNECT KEY PORT

El que manda este comando le avisa al *peer* que lo recibe que está listo para recibir una conexión TCP al puerto PORT y empezar a transferir el bloque con llave KEY. Este comando puede ser contestado con otro comando CONNECT KEY PORT indicando que es el otro lado el que debe de iniciar la conexión TCP.

HAS [KEY]

El *peer* que recibe el comando va a responder al que lo mandó con la lista que contiene únicamente la llave KEY en el caso de que tenga el bloque asociado con esta llave o una lista vacía en el caso contrario. Si el parámetro KEY fue omitido, va a mandar la lista con las llaves asociadas a todos los bloques que tiene almacenados.

MESSAGE TEXT

El *peer* que manda el comando, le está mandando el mensaje TEXT con propósitos informativos/vitácoras/QoS al nodo que lo recibe.

PING

Este comando se usa para ver si un nodo sigue vivo. En caso de que el *peer* que manda el comando esté en la lista de espera del que lo recibe, éste también le responde su posición POS en la cola.

Respuesta:

`PONG [POS]`

Como se mencionó arriba, los *peers* usan conexión TCP sobre demanda para la transferencia de bloques; y una vez establecida la conexión, ejecutan el siguiente protocolo:

`GET KEY OFFSET`

El bloque que recibe este comando empieza a mandar el bloque con la llave `KEY` empezando desde el *byte* `OFFSET`.

`BYE`

El *peer* que recibe este comando cierra la conexión TCP.

A.2. Comandos en la capa de Nodos de Llave

Cada nodo de llave va a almacenar la siguiente información: un conjunto de *peers* que almacenan el bloque asociado a su llave, la fecha de expiración del bloque asociado con su llave y un conjunto de sus réplicas. Aquí es importante mencionar que los nodos de llave asociados a una llave no van a ser el principal/maestro y clones/esclavos, si no que van a ser iguales entre sí. De este modo, se garantiza que si uno de los nodos de llave, independientemente de cuál sea, falla o muere, los demás nodos asociados a esta llave van a contener exactamente la misma información que tenía el nodo fallido y van a continuar proveyendo el servicio que se espera de ellos.

Igual que en la comunicación *Peer - Peer*, los nodos en esta capa van a comunicarse entre sí usando UDP *datagrams*.

`ADD KEY IP`

El nodo que recibe este comando va a agregar la dirección `IP` a su conjunto de *peers* que tienen almacenado el bloque asociado a la llave `KEY`. Esto, después de una confirmación[1]

del *peer* con dirección IP.

`DEL KEY IP`

El nodo que recibe este comando va a sacar la dirección IP de su conjunto de *peers* que tiene almacenado el bloque asociado a la llave KEY. Esto, después de una confirmación[1] del *peer* con dirección IP.

`SYNC KEY`

El nodo que manda este comando solicita al nodo que lo recibe le mande su lista de los que él considera sus réplicas, seguida de su lista de todos los *peers* que tienen almacenado el bloque asociado a la llave KEY. El nodo que recibe va a responder con esas listas.

A.3. Comandos en la capa de Nodos de Índice

Similarmente a los nodos de llave, los nodos de índice van a almacenar un conjunto de parejas de la forma (KEY,IPs) que asocien una llave dada al conjunto de nodos de llave responsables de la información acerca de esta llave y uno de sus réplicas. Al igual que con los nodos de llaves, todas las réplicas van a ser iguales en funcionalidad entre sí, así que la pérdida o fallo de alguno de los nodos de índice no van a perjudicar la disponibilidad de la información en la red. Como los nodos de más alto nivel y como tales con la mejor vista de la red, los nodos de índice van a mantener una lista de los nodos potenciales de llave y los nodos potenciales de índice. Así cuando se necesita un nodo de llave/índice, éste puede ser tomado de la lista de nodos observados por algún tiempo y esperemos bien comportados (una medida de buen comportamiento podría ser su disponibilidad).

Igual que con las otras dos capas, ésta también va a usar UDP *datagrams* para toda la comunicación que va a suceder a este nivel.

[1] Ver comando HAS en la capa de *peers*.

`ADD KEY IP`

El nodo que recibe este comando va a agregar la dirección `IP` a su lista de los nodos de llave que poseen información sobre la llave `KEY`. Esto, después de una confirmación[1] del nodo con dirección `IP`.

`DEL KEY IP`

El nodo que recibe este comando va a sacar la dirección `IP` de su lista de los nodos de llave que poseen información sobre la llave `KEY`. Esto, después de una confirmación[1] del nodo con dirección `IP`.

`SYNC [KEY]`

En caso de que el parámetro `KEY` esté ausente, el nodo que recibe este comando va a responder con la lista de todas sus réplicas, seguida de la lista de todas la llaves que conoce. Cuando el parámetro `KEY` esté presente, el nodo que recibe este comando va a responder con la lista de todos los nodos de llave conocidos (por él) que posean información sobre la llave `KEY`.

`LOCK`

El nodo que recibe el comando va a bloquear el bloque maestro contra modificaciones hasta que el bloqueo es levantado o expira. Tiene que responder informando que acepta el bloqueo al nodo que manda el comando.

Respuesta:

`LOCKACK`

`UNLOCK`

El nodo que recibe este comando va a levantar su bloqueo contra modificaciones sobre el nodo maestro.

[1] Ver Capitulo de Comunicación Entre Capas.

A.4. Comunicación Entre Capas

Hasta ahora, con todas estas capas como entes independientes podemos tener una red totalmente estática con todo el contenido que queremos que tenga, pero aún no es posible encontrar el contenido que necesitamos, ni mantener ese contenido en la red en caso de algún cambio en su topología. Por eso, para hacer que esta red funcione necesitamos hacer que las capas cooperen entre sí.

Aquí también, toda la comunicación entre nodos va a ser usando UDP *datagrams*.

A.5. Comunicación *Peer* - Llave

`PEERS KEY`

El nodo que recibe este comando (que debe ser un nodo de llave) va a responder con la lista de sus réplicas, las que almacenan información sobre la llave `KEY`.

`FIND KEY`

El nodo que recibe este comando (que debe ser un nodo de llave) va a responder con la lista de *peers* conocidos (por él) que almacenan el bloque asociado a la llave `KEY`.

`ADD KEY`

El nodo que recibe este comando (que debe ser un nodo de llave) va a agregar la dirección IP del *peer* que manda el comando a su lista de *peers* que tienen almacenado el bloque asociado a la llave `KEY`. Esto, después de una confirmación del *peer*.

`DEL KEY`

El nodo que recibe este comando (que debe ser un nodo de llave) va a sacar la dirección IP del *peer* que manda el comando de su lista de *peers* que tienen almacenado el bloque asociado a la llave `KEY`. Esto, después de una confirmación del *peer*.

`GET KEY`

El nodo que manda este comando (que debe ser un nodo de llave) está pidiendo al nodo que recibe el comando almacenar el bloque asociado a la llave `KEY`.

A.6. Comunicación *Peer* - Índice

`PEERS`

El nodo que recibe este comando (que debe ser un nodo de índice) responderá al nodo que manda este comando con la lista de sus réplicas.

`INFO KEY`

El nodo que recibe este comando (que debe ser un nodo de índice) responderá con la lista de todos los nodos de llave conocidos (por él), responsables de la llave `KEY`.

`WANNABE KEY|INDEX`

El nodo que recibe este comando (que debe ser un nodo de índice) va a agregar al nodo que manda el comando a su lista de nodos potenciales de llave/indice (esto dependiendo de si el parámetro es `KEY/INDEX`).

`BE KEY|INDEX IP1 [IP2 [...]]`

El nodo que manda el comando (que debe ser un nodo de índice) le está ofreciendo al nodo que recibe el comando, ya sea ser un nodo de índice y tomar como sus réplicas a los nodos con direcciones `IP1`, `IP2`, `...`; o bien, un nodo de llave responsable de mantener información acerca de la llave `KEY` y tomar como sus réplicas a los nodos con direcciones `IP1`, `IP2`, `...` . En el último caso, `IP1` puede ser `0.0.0.0` lo que significa que `IP2`, `...` son las direcciones de los *peers* que almacenan el bloque con llave `KEY`.

`GET KEY [IP]`

El nodo que manda este comando (que debe ser un nodo de índice) solicita al nodo que recibe el comando que almacene el bloque asociado a la llave `KEY` bajándolo del *peer* con dirección `IP`. El caso cuando no se manda el parámetro `IP` es idéntico al comando `GET KEY` de la sección Comunicación *Peer* - Llave.

`ADD KEY`

Al nodo que recibe este comando (que debe ser un nodo de índice) se le pide que agregue al canal, el programa con la llave `KEY`. Del nodo que manda este comando se espera que tenga en su capa de *peer* un bloque descriptor de programa válido asociado a la llave `KEY` junto con todos los bloques que compongan al programa.

A.7. Comunicación Llave - Índice

`NEEDPEER KEY`

El nodo que manda este comando (que debe ser un nodo de llave) le está solicitando al nodo que recibe este comando (que debe ser un nodo de índice) que convenza a algún *peer* para que sea su réplica.

`NEEDHELP KEY`

El nodo que manda este comando (que debe ser un nodo de llave) le está pidiendo al nodo que lo recibe (que debe ser un nodo de índice) que convenza a algún *peer* para que almacene el bloque asociado a la llave `KEY`.

`ADD KEY IP`

El nodo que recibe este comando (que debe ser un nodo de llave) va a agregar a su conjunto de nodos que tienen el bloque asociado a la llave `KEY`, al *peer* con dirección `IP` después de

una confirmación de dicho *peer*.

Apéndice B

Esquemas de XML

Este apéndice contiene las recomendaciones para los esquemas de XML de los diversos metabloques que el sistema usa.

En cada uno de estos esquemas se pueden usar varias veces los elementos para los que esto tenga sentido. Los atributos Lang pueden ser usados o no en cualquier subatributo del atributo en el que están en los esquemas abajo escritos, incluyendo el atributo en si. Y en caso de que Lang no sea heredado de algún elemento anterior, se considera desconocido.

B.1. Bloque Maestro

```
<?xml version='1.0' encoding='UTF-8'?>
<Master>
  <MasterKey>HEX Llave publica del canal</MasterKey>
  <Security>HEX Llave del bloque de permisos</Security>
  <Channel>HEX Llave del bloque descriptor de canal</Channel>
  <Signature>HEX Firma digital de los anteriores</Signature>
```

```
  <Schedule>HEX Llave de la programaci'on</Schedule>
  <Hash>HEX Hash del bloque completo</Hash>
</Master>
```

B.2. Bloque de Permisos

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<Security>
  <MaterKey>HEX La llave p'ublica del canal</MaterKey>
<!--Si el sub'arbol Channel no est'a,
significa que cualquiera puede modificar el canal-->
  <Channel>
    <Key CanCertify="BOOL" >HEX Llave p'ublica de alguien</Key>
  </Channel>
<!--Si el sub'arbol Scedule no esta,
significa que cualquiera puede modificar la programaci'on-->
  <Scedule>
    <Key CanCertify="BOOL" >HEX Llave p'ublica de alguien</Key>
  </Scedule>
  <Signature>HEX Firma por la llave maestra o alguna
    de las llaves del canal</Signature>
</Security>
```

B.3. Bloque Descriptor de Canal

```
<Channel>
  <ContentsTags Lang="Formato bibliogr'afico de 3
    letras ISO-639-2 (como fre para franc'es)" >
    <Title>STRING T'itulo del canal</Title>
    <SubTitle>STRING Subt'itulo/T'itulo alternativo
      del canal</SubTitle>
    <URL>STRING URL correspondiente al canal</URL>
    <Genre>STRING G'eneros del canal (classical,
      sci-fi, drama, etc)</Genre>
    <Mood>STRING Humores que refleja el canal
      (Romantic, Sad, etc)</Mood>
    <ContentType>STRING Tipos de programas del
      canal (Documentary, Feature Film, Cartoon,
      etc)</ContentType>
    <Subject>STRING Describe el tema del canal</Subject>
    <Description>STRING Breve descripci'on del contenido
      del canal</Description>
    <KeyWords>STRING Palabras claves asociadas al canal
      </KeyWords>
    <Rating>STRING Clasificaci'on del canal (A, B-15, C,
      ...)</Rating>
  </ContentsTags>
  <ThecnicalTags>
```

```
<Lenght>INT Duraci'on preferida en segundos</Lenght>
<Video>
  <Resolution>INTxINT Resoluci'on preferida de la se˜nal
    de video XxY</Resolution>
  <FPS>FLOAT El n'umero preferido de cuadros por
    segundo del video</FPS>
  <DataRate>INT Ancho de banda preferido de la
    compresi'on del video en bps</DataRate>
</Video>
<Audio>
  <Stream ID="INT Identificador de la se˜nal de audio" >
    <Language>STRING Lenguajes preferidos de la se˜nal
      de audio</Language>
    <Channels>INT No. de canales preferido (1=mono,
      2=setero, 6=5.1, ...)</Channels>
    <Frecuensy>INT Frecuencia preferida de la se˜nal en Hz
      (44100, 48000, 22050, ...)</Frecuensy>
    <DataRate>INT Ancho de banda preferido de la
      compresi'on de audio en bps</DataRate>
  </Stream>
</Audio>
<SubPicture>
  <Stream ID="INT Identificador de la se˜nal de subtitulos" >
    <Language>STRING Lenguajes preferidos de la
      se˜nal de subt'itulos</Language>
```

```
      </Stream>
    </SubPicture>
  </ThecnicalTags>
  <ChannelSpecs>
    <Window>
      <Length>INT Duraci'on en segundos de la ventana
        de programaci'on</Length>
      <StartAt>INT Duraci'on en segundos de cuanto va a
        mantener anterior al tiempo real</StartAt>
      <ProgramAfter>INT Duraci'on en segundos de a partir
        de que lugar de la ventana se pueden agregar
        programas</ProgramAfter>
    </Window>
    <Filter Accept="BOOL" ID="INT Numero de la regla" >
      UTF-8 Expresi'on regular que indique que se acepta
      al canal</Filter>
  </ChannelSpecs>
</Channel>
```

B.4. Bloque de Programación

```
<?xml version='1.0' encoding='UTF-8'?>
<Schedule Next="HEX siguiente bloque de programacion" >
  <Program Key="HEX llave del bloque descriptor de
    programa" Lenght="INT" Start="TIME" >UTF-8 Nombre
```

```
    del programa</Program>
</Schedule>
```

B.5. Bloque Descriptor de Programa

```
<Program>
  <ContentsTags Lang="Formato bibliogr'afico de 3
    letras ISO-639-2 (como fre para franc'es)" >
    <Title>STRING T'itulo del programa</Title>
    <SubTitle>STRING Subt'itulo/T'itulo alternativo
      del programa</SubTitle>
    <URL>STRING URL correspondiente al programa</URL>
    <Genre>STRING G'enero del programa (classical,
      sci-fi, drama, etc)</Genre>
    <Mood>STRING Humor que refleja el programa
      (Romantic, Sad, etc)</Mood>
    <ContentType>STRING Tipo del programa (Documentary,
      Feature Film, Cartoon, etc)</ContentType>
    <Subject>STRING Describe el tema del programa, como:
      Aerial view of Seattle.</Subject>
    <Description>STRING Breve descripci'on del contenido
      del programa</Description>
    <Synopsis>STRING Descripci'on de la historia del
      programa</Synopsis>
    <KeyWords>STRING Palabras claves asociadas al
```

```
    programa</KeyWords>
  <Rating>STRING Clasificaci'on del programa (A, B-15,
    C, ...)</Rating>
  <Date>STRING Fecha de la creaci'on/liberaci'on del
    programa</Date>
  <Country>Formato ISO-639-2 indicando el pa'is de
    origen del programa</Country>
  <Director>STRING Nombre del director del programa
    </Director>
  <Writer>STRING Nombre del guionista/escritor</Writer>
  <Composer>STRING Nombre del compositor</Composer>
  <Performer>STRING Nombre del actor/interprete
    <Role>STRING Nombre del personaje/papel interpretado
      </Role>
  </Performer>
</ContentsTags>
<ThecnicalTags>
  <Lenght>INT Duraci'on en milisegundos</Lenght>
  <Video>
    <Resolution>INTxINT Resoluci'on de la se˜nal de video
      XxY</Resolution>
    <AspectRatio>FLOAT Proporciones para el caso de que
      los pixeles no sean cuadrados</AspectRatio>
    <FPS>FLOAT El n'umero de cuadros por segundo del video
    </FPS>
```

```
    <DataRate>INT Ancho de banda de la compresi'on del
      video en bps</DataRate>
  </Video>
  <Audio>
    <Stream ID="INT Identificador de la se~nal de audio" >
      <Language>STRING Lenguaje o nombre de la se~nal de
        audio</Language>
      <Channels>INT No. de canales (1=mono, 2=stereo,
        6=5.1, ...)</Channels>
      <Frecuensy>INT Frecuencia de la se~nal en Hz (44100,
        48000, 22050, ...)</Frecuensy>
      <DataRate>INT Ancho de banda de la compresi'on de
        audio en bps</DataRate>
    </Stream>
  </Audio>
  <SubPicture>
    <Stream ID="INT Identificador de la se~nal de
      subtitulos" >
      <Language>STRING Lenguaje o nombre de la se~nal de
        subt'itulos</Language>
      <Impaired>BOOL Indica si los subt'itulos son
        especiales para personas con problemas auditivos
      </Impaired>
    </Stream>
  </SubPicture>
```

```
  </ThecnicalTags>
  <Parts>
    <TotalParts>INT Cantidad total de partes</TotalParts>
    <Part Pos="INT N'umero de esta parte" >HEX Llave del
      bloque</Part>
  </Parts>
</Program>
```

Bibliografía

[1] [online] bittorrent homepage. *http://bittorrent.com/.*

[2] [online] edonkey2000 and overnet homepage. *http://www.edonkey2000.com/.*

[3] [online] freenet homepage and protocol. *http://freenet.sourceforge.net/.*

[4] [online] gnutella protocol. *http://www.rixsoft.com/Knowbuddy/gnutellafaq.html.*

[5] [online] napster homepage. *http://www.napster.com/.*

[6] Secure hash standard, sha-1. *Federal Information Processing Standards Publication 180-1 (FIPS PUB 180-1)*, 1993.

[7] Beneh Dan. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS No. 46, pp. 203 - 213*, 1999.

[8] Oliver Heckmann and Axel Bock. The edonkey2000 protocol. Technical Report KOM-TR-08-2002, Multimedia Communications Lab, Darmstadt University of Technology, December 2002.

[9] D. Karger M. F. Kaashoek I. Stoica, R. Morris and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking, vol. 11, no. 1, pp. 17 - 32*, 2003.

[10] Thomas Plagemann Karl-André Skevik, Vera Goebel. Analysis of bittorrent and its use for the design of a p2p based streaming protocol for a hybrid cdn. *Technical Report, June, 2004.*

[11] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Processings of the IPTPS, Cambridge, MA, USA, February 2002, pp. 53 - 65.*

[12] Yuan Shen Edmund Wong Nirav Dave, Albert Huang. Content distribution using an enhanced bittorrent system. *http://people.csail.mit.edu/ndave/Academics/*, 2004.

[13] Adler S. The slashdot effect, an analisys of three internet publications. 1999.

[14] Ashish Sharma. The fasttrack network. *PC Quest, http://www.pcquest.com/content/p2p/102091205.asp.*

[15] Kunwadee Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. *Presentado en O'Reilly's www.openp2p.com, February 2001.*

[16] Beverly Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. Technical report, Stanford University, February 2001.

[17] Wenwu Zhu Zhensheng Zhang Zhe Xiang, Qian Zhang and Ya-Qin Zhang. Peer-to-peer based multimedia distribution service. *IEEE TRANSACTIONS ON MULTIMEDIA, VOL. 6, NO. 2, APRIL 2004.*