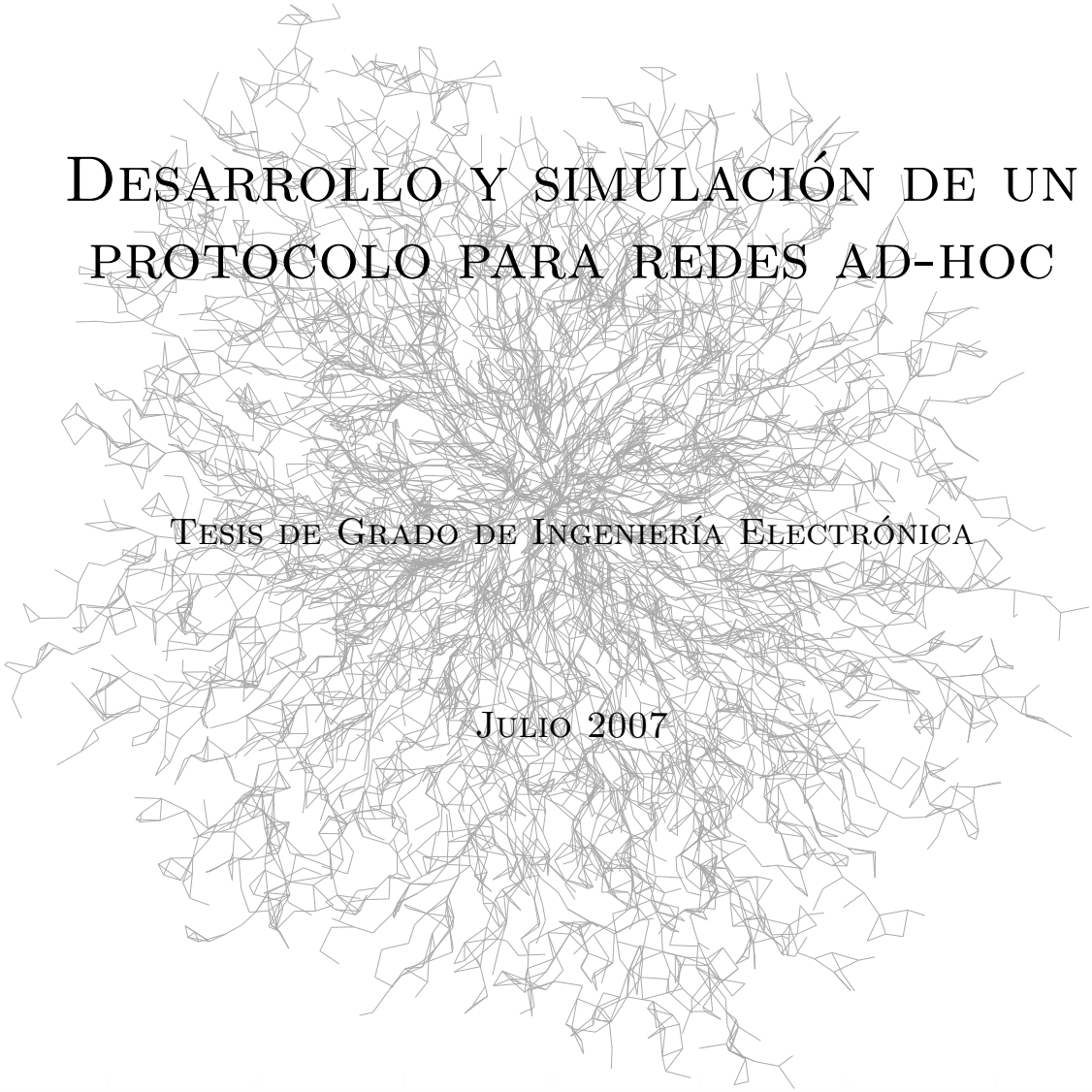
Desarrollo y simulación de un protocolo para redes ad-hoc



Tesis de Grado de Ingeniería Electrónica

Julio 2007

Autor: Alejandro Marcu (padrón 75.670)

Tutor: Dr. Ing. José Ignacio Alvarez-Hamelin

Departamento de Electrónica
Facultad de Ingeniería
Universidad de Buenos Aires

Agradecimientos

En primera instancia quiero agradecerle a José Ignacio Alvarez-Hamelin, quien durante todo el desarrollo de la tesis me guió, ayudó y enseñó con mucha dedicación y esmero.

A mi familia, Mónica, Michel y Claudia que siempre me alentaron y apoyaron en mis emprendimientos.

A Eugenia Gomez Blanco y Pablo Wolfus, con quienes compartí muchos días de trabajo en la tesis, haciendo menos solitaria la labor.

A Diego Gutnisky, con quien cursé el secundario y muchas materias de la facultad, debatiendo siempre nuestras ideas y aprendiendo el uno del otro.

A los compañeros de la facultad, con quienes cursé materias, estudié e hice trabajos prácticos, haciendo más amena la carrera; y muchos de ellos son ahora amigos, más allá de la facultad.

A mis amigos de otros ámbitos; a los que me preguntaron “cuando te recibís?” incansablemente, y que por fín les puedo dar una respuesta.

Índice general

Capítulo 1

Introducción

Los protocolos de ruteo se diseñan con el objeto de brindar servicios en una red específica. En el caso de Internet, poco se podía saber de sus actuales alcances en la década del 70, dónde se comenzaron a probar los primeros protocolos. Ciertas características de diseño evolucionaron, como ser la movilidad o el número de usuarios, y su impacto no es menor. También aparecieron nuevas posibilidades, principalmente basadas en la tecnología wifi: las redes ad-hoc. Todas estas innovaciones demandan rediseñar los protocolos para orientarlos hacia la movilidad, la autogestión y descentralización, y el crecimiento.

Contrariamente a la organización distribuida pero jerárquica (siendo por ende centralizado en las altas jerarquías) del servicio de DNS en Internet, el protocolo que se propone aquí distribuye la resolución de direcciones en toda la red, evitando cuellos de botella de tráfico y dándole mayor robustez. Si bien éste no es un servicio de nombres como el DNS, fácilmente podría cumplir además con esa función en el caso de poder asegurar que los nombres de los nodos sean únicos dentro de la red.

Otros de los puntos débiles de los protocolos actuales es la falta de correlación entre las direcciones asignadas y la topología subyacente, teniendo como consecuencia la necesidad de tablas de ruteo muy grandes para poder dirigir el tráfico. En la solución propuesta, las direcciones asignadas tienen una estrecha relación con la topología, lo que permite optimizar la elección de la ruta a la vez que se disminuye la información que se debe almacenar en las tablas de ruteo. Esto podría resolver el actual crecimiento desmesurado del tamaño de las tablas de ruteo.

Por ello, en esta tesis se propone el protocolo `ANTop` (Adjacent Network Topologies), buscando resolver las falencias mencionadas. Este protocolo fue diseñado para redes ad-hoc (normalmente son creadas en forma espontánea), y dada su naturaleza, sin un servidor o nodo central. Todos los nodos conectados a la red tienen la misma jerarquía y pueden cumplir las mismas funciones, como por ejemplo asignar direcciones a los nuevos vecinos, resolver direcciones a partir de identificadores o efectuar el ruteo de paquetes. Este modelo es completamente descentralizado y distribuido.

Los alcances de la presente tesis comprenden tres partes fundamentales. La primera es la especificación completa de `ANTop` (algoritmos y los formatos de los paquetes de datos y control). La segunda consiste en el diseño y programación de un simulador de redes con `ANTop` incorporado, que sirvió para encontrar y corregir algunas fallas, refinando el protocolo hasta lograr su correcto funcionamiento. Finalmente, en la tercera parte se realizaron diversas simulaciones del protocolo en distintas condiciones, pudiendo evaluar así su comportamiento y eficiencia.

1.1. Estado del arte

En esta sección se presentan en forma sintética otros protocolos para redes ad-hoc, analizando brevemente sus diferencias.

La base de `ANTop` puede encontrarse en los protocolos `Tribe` [8] y `PeerNet` [2]. Ambos protocolos son similares, y aunque `Tribe` es anterior cronológicamente, es más complicado de comprender por sus

abstracciones y notación. Es por ello que se describirá brevemente `PeerNet`.

Este protocolo se inspira en las aplicaciones punto a punto (*peer-to-peer*), e intenta reproducir sus ventajas en la capa de red.

La idea principal de este protocolo es separar la identidad del nodo de su dirección, contrariamente a `IP` donde el identificador es la dirección. Uno de los inconvenientes de la aproximación de `IP` en redes móviles es que si el nodo cambia de dirección, y por lo tanto de identificador, la sesión `TCP` se corta. La separación de estos dos elementos resuelve este problema y facilita el ruteo, ya que la dirección del nodo refleja la posición en la red. Esta aproximación plantea sin embargo dos nuevas necesidades: poder asignar direcciones en forma consistente con la topología, y contar con un servicio de resolución de identificadores de nodos eficiente.

Para ello, las direcciones de los nodos se asignan dinámicamente según la posición en la red, formando un árbol donde las hojas son las direcciones de los nodos. Los nodos internos del árbol de direcciones no tienen existencia física pero representan áreas de `PeerNet`, definiéndose un área como un conjunto de nodos tales que cualquier par de nodos dentro de ella pueden comunicarse utilizando un camino que no sale del área.

La capa de red del protocolo `PeerNet` se puede dividir en 3 partes:

1. Asignación de direcciones
2. Ruteo
3. Búsqueda de nodos

Para la asignación de direcciones, cuando un nuevo nodo desea conectarse, le pide una dirección a otro nodo, el cual divide su espacio de direcciones en dos y le cede una de las mitades.

El ruteo consiste en ir descendiendo por las áreas o subárboles hasta encontrar el nodo deseado. Para ello, cada uno de los nodos tiene una tabla de ruteo que indica cuál es el destino al que debe ir en caso de querer cambiar un bit determinado de la dirección. De esta forma, la tabla de ruteo tiene un tamaño igual a la cantidad de bits de la dirección, es decir que para una red de tamaño n, se necesitan $l = \log(n)$ bits para la dirección, y tablas de ruteo de tamaño l en cada nodo.

La búsqueda de nodos consiste en obtener la dirección de un nodo sabiendo su identificador primario. Esta información es almacenada de forma descentralizada en cualquiera de los nodos de la red. Para ello, se aplica una función al identificador que devuelva una dirección de la red, y este nodo es utilizado para guardar la información. Cuando un nodo desea conocer la dirección para un identificador, aplica esa misma función y envía un pedido de resolución a esa dirección, obteniendo como respuesta la dirección deseada.

Se propone para el ruteo utilizar un tamaño fijo de direcciones de 128 bits, lo que fija a su vez el tamaño de las tablas de ruteo de todos los nodos en ese valor.

`ANTop` y `PeerNet` se pueden comparar, siendo sus coincidencias y diferencias:

- Ambos son descentralizados.
- Ambos separan las direcciones de los identificadores y usan un esquema de *Rendez-Vous* distribuido para realizar el ruteo en dos fases.
- `ANTop` utiliza una estructura de hipercubo, mientras que `PeerNet` utiliza un árbol.
- `ANTop` introduce el concepto de dirección secundaria para aproximarse a un hipercubo, ampliando así la conectividad (permitiendo la existencia de bucles), sin equivalente en `PeerNet`.
- `ANTop` no fija el largo de las direcciones, pero todos los nodos de una red deben utilizar el mismo valor, cuyo máximo es 255 bits. En cambio, `PeerNet` utiliza una longitud fija de 128 bits.

- El ruteo en ambos casos busca aprovechar la información topológica que provee la dirección. Además, `ANTop` provee dos soluciones para el ruteo: una proactiva y otra reactiva. Considerando el ruteo reactivo, éste utiliza un mecanismo de detección de bucles y vuelta atrás para investigar nuevos caminos y ser robusto ante cambios en la red. El formato de las tablas de ruteo es completamente diferente.
- `PeerNet` permite el re-balanceo del árbol, cambiando las direcciones de algunos nodos para obtener una mejor distribución de las direcciones, lo que podría afectar al ruteo durante el rebalanceo de nodos. En cambio, en `ANTop`, una vez asignada la dirección, ésta no cambia.

1.2. Desarrollo de la tesis

Para el desarrollo de esta tesis, se tomó como punto de partida al trabajo *"Architectural Considerations for a Self-Configuring Routing Scheme for Spontaneous Networks"* [1], donde se propone usar hipercubos incompletos para el direccionamiento, y se dan los algoritmos de ruteo reactivo y proactivo.

A partir de esto, se buscó especificar completamente todos los algoritmos y los datos que se deben intercambiar entre los nodos para implementar el protocolo. El resultado se describe en el capítulo 2, dónde se analizan varias posibilidades y se muestran los pasos seguidos para llegar a la solución propuesta, considerando ventajas y desventajas de distintas soluciones para cada problema en particular. El protocolo se dividió en tres partes relativamente independientes: conexión y desconexión de nodos, ruteo reactivo y registro de direcciones, constando cada una de ellas con una sección dentro del capítulo.

Luego, se diseño e implementó un simulador (llamado `QUENAS`), detallado en el capitulo 3. Si bien se diseño con el objetivo de simular `ANTop`, también se buscó hacerlo flexible para que pueda ser extendido en el caso de querer hacer simulaciones adicionales o inclusive simular otros protocolos. El capitulo explica los requerimientos que se impusieron para este programa y las decisiones tomadas para poder lograrlo. Además, explica como fue implementado, entrando en detalle de las clases que componen al programa.

Utilizando `QUENAS` se corrieron dos tipos de simulaciones: una dónde se utilizan redes generadas aleatoriamente, y otra con una red de Sistemas Autónomos real. Estas simulaciones se detallan en el capítulo 4.

Los datos obtenidos por las simulaciones fueron procesados para obtener distintos aspectos del protocolo, como por ejemplo el número de conexiones por nodo, tiempo de obtención de dirección primaria, tiempo de registración, longitud de las rutas, tamaño de las tablas de ruteo y *Rendez-Vous*, etc. Todos estos datos se presentan en forma de gráficos en el capítulo mencionado, analizando las causas de los resultados obtenidos así como las implicaciones.

El capítulo 5 presenta las conclusiones obtenidas en el desarrollo de esta tesis, y sugiere además posibilidades para futuros trabajos en el mismo tema.

Capítulo 2

El protocolo

En este capítulo se explican las bases del protocolo, tomando como punto de partida al trabajo *"Architectural Considerations for a Self-Configuring Routing Scheme for Spontaneous Networks"* [1].

Con el fin de facilitar el desarrollo, se dividió al protocolo en tres partes relativamente independientes:

- Conexión y Desconexión de nodos: cómo manejar cambios en la topología de la red.
- Ruteo Reactivo: cómo guiar la información desde el origen al destino.
- Registro de Direcciones: cómo obtener la ubicación de un nodo en función de su nombre.

Para cada una de estas partes, se plantean los requerimientos que debe cumplir, para luego analizar una solución que cumpla con ellos de la mejor manera posible, detallando la información necesaria para que el protocolo pueda ser simulado o implementado.

El nombre elegido para el prtocolo es `ANTop`, proveniente de *Adjacent Network Topologies* (topologías de redes adyacentes).

2.1. Introducción al protocolo ANTop

Esta sección presenta una breve descripción de las bases del protocolo especificado en [8], explicando y ejemplificando como se construye una red de hipercubo, así como la noción de ruteo indirecto, y como encaja este protocolo dentro de la arquitectura de redes.

2.1.1. Direccionamiento basado en un hipercubo

El protocolo utiliza un direccionamiento basado en una estructura de hipercubo. Un hipercubo es la generalización de un cubo a n dimensiones, donde cada uno de los 2^n vertices es adyacente a otros n vértices. Cada uno de ellos se encuentra ubicado en las coordeandas $(x_1, x_2, ..., x_n)$, siendo $x_i = 0$ ó $x_i = 1$. De esta forma, todas las combinaciones están cubiertas, y se puede escribir la coordenada de un vértice en forma abreviada omitiendo paréntesis y comas, por ejemplo: 1000 corresponde a $(1, 0, 0, 0)$. Dos vértices son adyacentes si y sólo si difieren en una coordenada, por ejemplo 1000 y 1100 son adyacentes, pero 1000 y 0100 no lo son. En la figura 2.1 se muestra la proyección de un hipercubo de dimensión 4 [9].

Para ir de un vértice a otro recorriendo las aristas, y tomando las coordenadas como un número binario, se debe ir cambiando cada uno de los bits en los que éstos difieren en cualquier orden. Por ejemplo, para ir de 0110 a 1010, se puede recorrer pasando por 1110 para luego llegar a 1010, o pasando por 0010 y luego 1010. Siendo la distancia de un vértice a otro la cantidad de aristas recorridas, el número de aristas coincide con la cantidad de bits diferentes. Es decir, la distancia entre un vértice y otro es la cantidad de bits en que difieren sus coordenadas. En el ejemplo anterior, la distancia entre los nodos propuestos es 2.

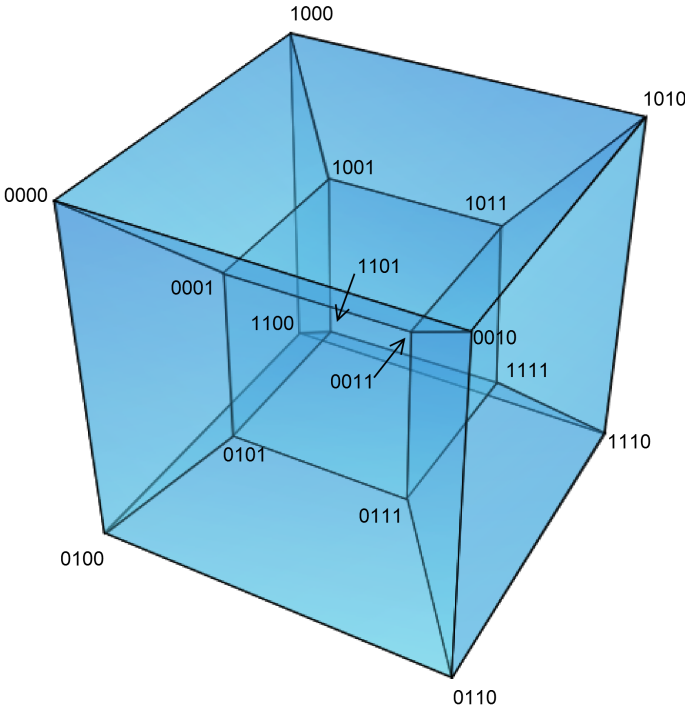


Figura 2.1: Hipercubo de dimensión 4 y sus direcciones asociadas

Aplicando esto a redes de computadoras, sería ideal que cada nodo (por ejemplo, una computadora) correspondiera con uno de los vértices. Sin embargo, esto no es factible para casos reales, y por ello se utilizan hipercubos incompletos, dónde no todos los vértices tienen un nodo asignado.

A medida que se van conectando nodos a la red, a cada uno de ellos se le asigna una dirección de hipercubo. Esta asignación se produce en forma descentralizada, obteniendo esta dirección de alguno de los nodos a los que se encuentra conectado físicamente. Para ello, un nodo no sólo tiene una dirección, llamada *dirección primaria*, sino que además cuenta con un espacio de direcciones que irá cediendo a los nuevos vecinos. Para representar este espacio de direcciones, se utiliza una máscara en la dirección primaria, que indica la cantidad de bits que son fijos en el espacio de direcciones. Por ejemplo $0100m2$ indica que la dirección primaria es 0100 y que la máscara es 2. Entonces, los dos primeros bits son siempre 01 y los otros dos toman todas las combinaciones posibles, que dado que son 2 bits, se tienen $2^2 = 4$ variaciones. Por lo tanto, este espacio de direcciones está compuesto por 0100, 0101, 0110 y 0111.

Inicialmente, el primer nodo de la red utilizará una dirección compuesta por todos ceros y una máscara 0, indicando de esta forma que controla todo el espacio de direcciones. En la figura 2.2 se muestra un ejemplo de un sólo nodo en la "red", utilizando un hipercubo de dimensión 4. Este nodo administra el espacio completo de direcciones.

Figura 2.2: Un solo nodo en una "red"

Luego, el nodo b que tiene conexión física con a desea unirse a la red, y a debe cederle una parte. El nodo a le da entonces la mitad de su espacio de direcciones, aumentando en uno el valor de su máscara, pero manteniendo la dirección primaria invariante. El nodo b recibe la otra mitad, siendo su dirección primaria 1000 y su máscara 1, como muestra la figura 2.3.

Continuando con el ejemplo, si se agregan los nodos c, d y e a la red como lo muestra la figura 2.4,

Figura 2.3: Un segundo nodo se conecta

las asignaciones se producen de la siguiente forma:

- c se conecta a a: a queda con la dirección $0000m2$, y le cede $0100m2$ a c
- d se conecta a b: b queda con la dirección $1000m2$, y le cede $1100m2$ a d
- e se conecta a d: d queda con la dirección $1100m3$, y le cede $1110m3$ a e

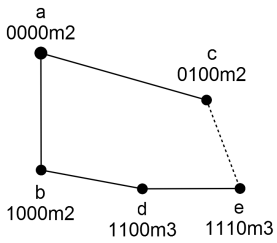


Figura 2.4: Tres nodos más se conectan a la red

La figura muestra además una linea de puntos entre c y e, representando una conexión física existente, que no está siendo aprovechada porque sus direcciones no son adyacentes y por lo tanto no se pueden conectar.

Si bien el nodo c tiene la dirección primaria 0100 que no es adyacente a la dirección 1110 de e, dentro del espacio de direcciones de c se encuentra una dirección que si lo es: 0110. Para que los nodos puedan conectarse, c puede entonces utilizar esta dirección, que pasa a ser una *dirección secundaria*. De esta forma, se gana conectividad a cambio de utilizar más de una dirección en un nodo.

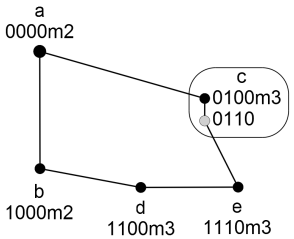


Figura 2.5: Utilización de una dirección secundaria

Si ahora desea conectarse a la red el nodo f, que tiene conexiones físicas con b y e, estos útlimos dos nodos le ofrecerán las direcciones $1010m3$ y $1111m4$. El nodo debe elegir una siguiendo ciertos criterios; uno de los cuales es elegir la menor máscara, para obtener el máximo espacio de direcciones. En ese caso, elegiría la dirección $1010m3$. Esta dirección resulta ser adyacente a 1110, y por ello puede también estar conectado lógicamente a e, aunque ninguno sea padre del otro. Es decir que a pesar de que la forma de asignación de direcciones genere en principio un árbol, se pueden establecer conexiones entre las ramas en casos como el recientemente presentado, o cuando se asignan direcciones secundarias. Entonces, el

grafo generado tiene más conexiones que si fuera un árbol, y esto le permite tener caminos alternativos que mejoran la robustez, ya que en el caso de un árbol solo las hojas se pueden eliminar sin fragmentar la red; y además se evita sobrecargar la raiz.

La figura 2.6 muestra como queda la red luego de que el nodo f se conecta.

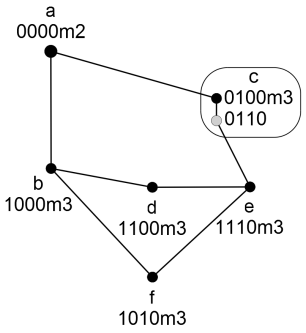


Figura 2.6: Adición del nodo f

Ahora, si se conecta el nodo g a b, adquiere la dirección $1001m4$, y b queda con $1000m4$. Luego, el nodo f se desconecta, y aparece el nuevo nodo h que está conectado a b y e, como muestra la figura 2.7.

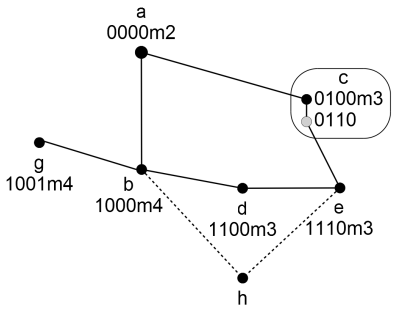


Figura 2.7: Se conecta el nodo g y se desconecta f

Cuando el nodo h pida una dirección, el nodo b no tiene espacio para cederle, y el nodo e le ofrece la dirección $1111m4$, que podría ser aceptada y conectaría al nodo. Sin embargo, el nodo f dejó un "hueco" en el espacio de direcciones de b al irse, que podría ser aprovechado por h. Es por ello que cuando un nodo se conecta, no sólo se le proponen direcciones primarias, sino que eventualmente se proponen direcciones de reconexión, que el nodo podrá utilizar si resulta conveniente.

En el ejemplo, b le propone a h la dirección de reconexión $1010m3$, que es aceptada, quedando la red como muestra la figura 2.8.

2.1.2. Ruteo Indirecto

Se vió en el apartado anterior que a cada nodo se le asigna una dirección en el hipercubo. Sin embargo, es muy posible que un nodo no sepa cuál es la dirección en el hipercubo del destino, ya que esta no está previamente acordada. Evidentemente éste debe tener alguna información del nodo con el que desea comunicarse, que se denomina dirección universal, y es única para cada posible nodo. En los ejemplos que se dieron anteriormente, podría ser el nombre que se le dió por comodidad, por ejemplo a, b, etc. Sin embargo, esta información no sirve para llevar un paquete a destino, ya que el nombre no dice nada acerca de su ubicación; para eso se hizo justamente el direccionamiento con estructura de hipercubo.

De ahí surge la necesidad de poder realizar una "traducción" entre el identificador universal, y la dirección de hipercubo. En este protocolo, esto se resuelve mediante nodos *Rendez-Vous*, que son capaces

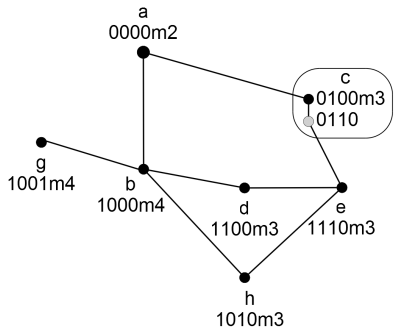


Figura 2.8: Se conecta el nodo h

de brindar este servicio. Es decir, si el nodo a desea comunicarse con e, le debe pedir a un nodo *Rendez-Vous* que le traduzca "e" a una dirección en el hipercubo, calculando la dirección de éste nodo con una función de hash a partir de la dirección universal. Luego, el nodo de *Rendez-Vous* devolvería 1110 como resultado, y con esta información, a manda el paquete a 1110.

Con este esquema, cada nodo utiliza 3 identificadores. El primero es la dirección universal, que debe ser conocida por los nodos que quieran comunicarse con él, y que es totalmente independiente de la ubicación en la red. En el ejemplo anterior, las direcciones universales son a, b, c, etc.

El segundo identificador es la dirección virtual, que es la traducción de la dirección universal en el espacio de direcciones. Para ello, se aplica una función de hash cuyo resultado se encuentra en este espacio. Por ejemplo, dada la función de hash F y siendo $F(e) = 1111$, entonces 1111 es la dirección virtual, es decir, la dirección de red del nodo *Rendez-Vous* para e. Puede ocurrir, como en este caso, que ningún nodo tenga como dirección primaria esta dirección virtual; entonces se debe hacer cargo aquel nodo que tenga esta dirección en su espacio de direcciones. Este identificador, al igual que la dirección universal, no depende de la ubicación en la red.

El tercer identificador es la dirección relativa o dirección de red, que indica dónde se encuentra el nodo en el hipercubo. Por ejemplo, la dirección de red de e es 1110.

Cuando un nodo se conecta a la red, debe enviarle al nodo de *Rendez-Vous*, cuya dirección es su propia dirección virtual, un paquete conteniendo la dirección universal y la dirección de red, para que sean almacenadas en la tabla de *Rendez-Vous*. Luego, cuando otro nodo desea comunicarse con él, computa su dirección universal utilizando la misma función de hash y le manda un pedido de resolución al *Rendez-Vous*, quien responde dando la dirección de red del nodo solicitado. Finalmente, el nodo que desea enviar un paquete cuenta con la dirección de red del destino y puede comenzar la comunicación.

En este protocolo, dado que es descentralizado, todos los nodos cumplen con la función de *Rendez-Vous*, siendo responsable cada uno de una parte del espacio de direcciones universales.

2.1.3. `ANTop` en el modelo de capas

En esta sección se muestra dónde se ubica el protocolo `ANTop` en un modelo de capas.

El diseño de la arquitectura de redes se realiza generalmente por capas, donde cada capa se construye encima de la otra. Cada capa le brinda servicios a la capa superior, haciendo uso de los servicios brindados por la capa inferior. Esto ayuda a reducir la complejidad del diseño.

Se utilizará el modelo híbrido propuesto en [7] que toma parte del modelo TCP/IP y parte del modelo OSI. En la figura 2.9 se muestran las capas del modelo.

La capa física de un nodo se comunica con la capa física de otro nodo a través de un canal (wi-fi, cable, etc). Cada una de las capas prestará servicios a las capas superiores de forma tal que puedan comunicarse con capas de igual nivel en otros nodos sin conocer los detalles subyacentes. Por ejemplo,

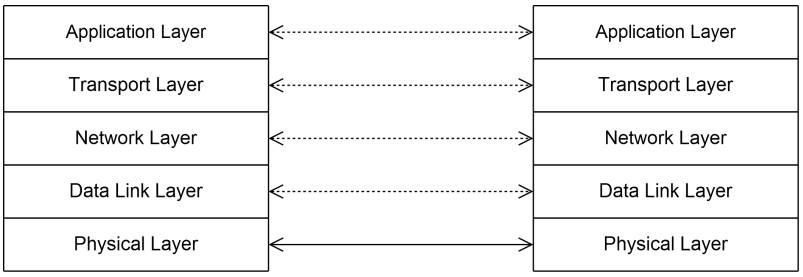


Figura 2.9: Modelo Híbrido

una aplicación en un nodo debe poder mandar información a otra aplicación en otro nodo sin tener que lidiar con los detalles de los bits transmitidos, el ruteo, confirmación y retransmisión de datos, etc. Simplemente, se basará en un servicio de la capa de transporte para enviar la información a destino.

`ANTop` es básicamente un protocolo de red; sin embargo en este trabajo se implementó una parte en la capa de aplicación para poder utilizar los servicios de conexión confiable de la capa de transporte. La figura 2.10 muestra como encaja el protocolo en el modelo híbrido.

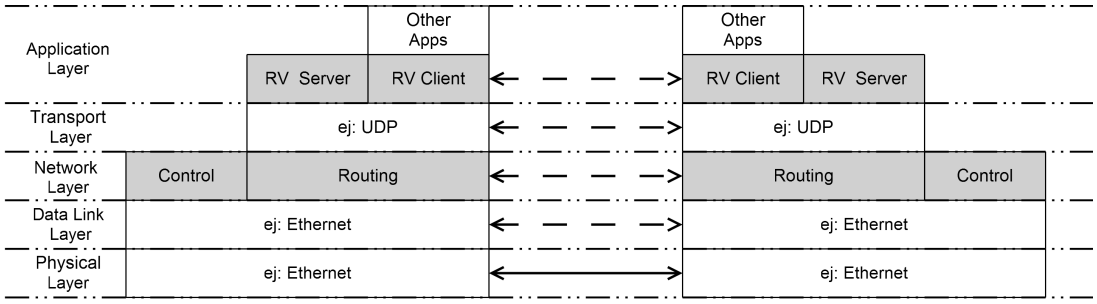


Figura 2.10: El hipercubo en el modelo híbrido

En la capa de red, se encuentra el módulo *Control*, responsable de gestionar la conexión y desconexión del nodo, ofrecer direcciones primarias y secundarias a los vecinos e indicarles que la conexión sigue activa. Las capas superiores no intervienen en el proceso de conexión, es por ello que no se dibujó la capa de transporte sobre este módulo.

También en la capa de red, el módulo *Routing* se encarga de dirigir los paquetes para que lleguen a destino. En el nodo de origen, la capa de transporte realiza una llamada a una función en este módulo indicándole los datos que se mandan y a que destino. El módulo elige cuál de los vecinos es el más conveniente para enviarle el paquete, y lo manda usando los servicios de la capa inferior (*Data Link Layer*). De ahí en más, el paquete irá pasando por distintos nodos intermedios, llegando hasta el módulo de ruteo (*Routing*) en cada caso, y como el nodo no es el destino final, el paquete se envía a un vecino, sin notificar de ninguna forma a la capa de transporte. Es decir, cuando el nodo cumple la función de ruteador, las capas de transporte y de aplicación ni siquiera advierten que esto está ocurriendo. Finalmente, cuando el paquete encuentra el nodo destino, el módulo realiza una llamada a una función en la capa de transporte pasándole la información recibida.

En la capa de aplicación se encuentran dos módulos, *Rendez-Vous Client* y *Rendez-Vous Server*, a cargo del ruteo indirecto. El primer módulo es el que se encarga de resolver las direcciones universales en direcciones de hipercubo; es por ello que las otras aplicaciones se encuentran encima de está, ya que necesitan del servicio de resolución de direcciones. El segundo módulo se encarga de gestionar la tabla de resolución y de responder a los pedidos hechos por los clientes.

2.2. Conexión y Desconexión de nodos

En el protocolo `ANTop` no hay ningún servidor central ni jerarquías, por lo tanto todos los nodos tienen la misma funcionalidad. Lo primero que necesita un nodo para conectarse es obtener una dirección primaria, ya que de otra forma no podría ser distinguido de otros nodos. Una vez obtenida, puede comenzar a enviar y recibir paquetes de otros nodos, así como a ceder su espacio de direcciones o adquirir direcciones secundarias para mejorar la conectividad.

Cuando el nodo se desconecta, es necesario también seguir un protocolo para cederles a otros nodos la información que pudieran necesitar.

En esta sección se desarrolla la parte relativa a la conexión y desconexión de los nodos, mostrando el motivo por el que fueron tomadas las decisiones, y dando como resultado un diagrama de estados y la descripción de los paquetes, suficientes para simular o implementar esta parte del protocolo.

2.2.1. Requerimientos

Las tareas que debe realizar un nodo son:

- Obtener una dirección primaria de un vecino y conectarse en cuanto la obtenga, o asignarse dirección inicial si no existen vecinos.
- Obtener direcciones secundarias.
- Ceder direcciones primarias y secundarias.
- Notificar a los vecinos a intervalos regulares que sigue conectado.
- Permitir a otros procesos en el nodo realizar sus tareas de limpieza antes de desconectarse.

2.2.2. Desarrollo de la solución

En esta parte del protocolo se pueden distinguir claramente tres etapas: establecimiento de la conexión, dirección estable y desconexión, por lo que se analizan cada una de ellas por separado.

Establecimiento de la conexión

Cuando un nodo desea conectarse a la red primero tiene que obtener una dirección primaria.
Para ello, debe notificar a sus vecinos para que éstos le ofrezcan direcciones primarias.
Esto lo hace enviando un paquete de tipo PAR (Primary Address Request) en modo broadcast, para que todos sus vecinos lo reciban y respondan a su vez con un paquete PAP (Primary Address Proposal), sugiriendo direcciones primarias.
Se deben considerar las siguientes posibilidades:

1. No hay ningún nodo en la red y por lo tanto no obtiene respuesta.
2. Uno o varios vecinos le responden cediéndole direcciones.
3. Los vecinos no tienen espacio disponible para cederle.
4. El paquete PAR o su respuesta se pierden.

Para poder resolver la primera situación, en la que el nodo es el primero en unirse a la red, no alcanza con esperar respuesta indefinidamente dado que nadie le respondería. Por ello se debe utilizar un timeout, luego del cual el nodo puede considerar que no hay otros nodos en la red.

En la segunda posibilidad, cuando se obtiene respuesta, se podría utilizar la dirección propuesta tan pronto como es ofrecida, sin embargo esto imposibilita que el nodo pueda elegir la dirección primaria más conveniente según su criterio, por ejemplo la que tiene un espacio de direcciones más grande o la que la conecta con el nodo que a su vez tiene más conexiones.

Para permitir que el nodo escoja la dirección primaria más conveniente, es necesario seguir esperando luego de obtener una respuesta. Una vez que el nodo envió el pedido de dirección, puede esperar un cierto tiempo, y una vez cumplido elegir la dirección más conveniente.

Si ninguno de los vecinos tiene espacio para cederle, el nodo no se podrá conectar a la red. Se debe distinguir esta situación de cuando el nodo se encuentra solo en la red, por lo que los vecinos no pueden simplemente no responder cuando no tienen direcciones disponibles, sino que tienen que contestar advirtiendo que no tienen espacio para ceder.

En el caso de que un paquete PAR o sus respuestas se pierdan, si el nodo obtiene alguna respuesta de otro vecino, se conectará de todas formas aunque tal vez no tenga la mejor dirección, lo cual no resulta óptimo, pero tampoco causa errores. La situación es más delicada cuando hay un solo vecino y algún paquete se pierde, o en el menos probable caso de que haya varios vecinos y todos los paquetes PAR o respuestas se pierdan, ya que esto ocasionaría que el nodo no reciba respuesta y considere que es el primer nodo en la red, produciendo más tarde una colisión de direcciones.

Para disminuir notablemente la posiblidad de que esto ocurra, en el caso de que el nodo no obtenga respuesta, se repite el procedimiento de enviar un paquete PAR y esperar respuesta. Cuantas más veces se reintente, menos probable es llegar a la situación problemática, a cambio de un mayor tiempo de conexión.

Hasta aquí, el nodo pudo elegir una dirección primaria. Sin embargo, esto no es suficiente para que le sea asignado, dado que los vecinos que enviaron propuestas no saben cuál de ellas va a ser elegida.

Por lo tanto, es necesario que el nodo confirme su elección. En principio, la confirmación podría ser enviada solamente al vecino cuya dirección fue elegida, pero es conveniente enviársela a todos como broadcast para que los que no fueron elegidos no continúen esperando una eventual respuesta. La confirmación es enviada en un paquete PAN (Primary Address Notification).

El proceso de conexión podría terminar luego del envío de este paquete. Sin embargo, si este se perdiera, el nodo estaría utilizando una dirección primaria sin que el vecino tenga conocimiento, con lo cual éste último posiblemente se la ceda a otro nodo, provacando un conflicto de direcciones.

Para evitar que pueda surgir un conflicto de direcciones por un paquete perdido, se agregó una confirmación de la notificación; es decir que el nodo que recibe el paquete PAN debe responderle al remitente con un PANC (Primary Address Notification Confirmation). El nodo comenzará a utilizar la dirección que le cedieron recién al recibir el paquete PANC. De esta forma, si se pierde el paquete PAN o PANC, el nodo no utilizará esa dirección. En el caso de que el paquete PAN se pierda, el vecino que no es notificado y por ello no cede su espacio de direcciones ni envía la confirmación. Después de un tiempo de espera, el nodo vuelve a intentar conectarse. En el caso de que se pierda el paquete PANC, el vecino cede su espacio de direcciones pero el nodo no lo utiliza y reintenta la conexión. Cuando el nodo que cedió la dirección recibe nuevamente un paquete PAR del mismo nodo, concluye que el paquete PANC se perdió y recupera el espacio que le había asignado.

El pseudo código para el establecimiento de la dirección primaria se puede ver en el Algoritmo 1. Este algoritmo utiliza los siguientes parámentros del protocolo:

- N_PAR: cantidad de veces que se envía el paquete PAR en caso de no recibir respuesta.
- T_PAP: tiempo de espera de paquetes PAP.
- T_PANC: tiempo máximo de espera de paquetes PANC.

El algoritmo tiene garantizada su finalización, ya que el bucle de los pasos 2 a 9 concluye en a lo sumo N_PAR iteraciones, y la espera del paquete PANC se limita a un tiempo T_PANC en el paso 13.

Algoritmo 1 Establecimiento de dirección primaria

```
1: i := N_PAR
2: repetir
3:    Enviar paquete PAR
4:    Esperar un tiempo T_PAP, almacenando los paquetes PAP recibidos
5:    i := i − 1
6:    si si se recibió al menos un paquete PAP con dirección disponible entonces
7:       ir a 10
8:    fin si
9: hasta que i == 0
10: si se recibió al menos un paquete PAP con dirección disponible entonces
11:    Elegir entre las direcciones propuestas una con la menor máscara
12:    Enviar un paquete PAN en broadcast notificando la dirección elegida
13:    Esperar a que llegue un paquete PANC o que transucrra un tiempo T_PANC.
14:    si se recibió un paquete PANC entonces
15:       La dirección primaria elegida ya puede ser utilizada. Fin del algoritmo.
16:    fin si
17:    ir a 1
18: si no
19:    si se recibió un paquete PAR indicando que no hay direcciones disponibles entonces
20:       No se puede conectar. Fin del algoritmo.
21:    fin si
22:    Utilizar la dirección compuesta por todos ceros
23: fin si
```

Dirección estable

Una vez que el nodo obtiene una dirección primaria, se queda en ese estado hasta que se requiera la desconexión.

Durante ese período se deberán cumplir ciertas funciones:

1. Escuchar pedidos de dirección primaria y reponder.
2. Envíar paquetes a los vecinos para informarles que el nodo sigue conectado.
3. Verificar que los vecinos envien paquetes períodicamente.
4. Ofrecer direcciones secundarias
5. Responder a pedidos de direcciones secundarias

La primera función es la que permite que los otros nodos se conecten de la misma forma que éste se conecto, dado que no hay centralización en la red y todos los nodos operan por igual.

Cuando se recibe un pedido de dirección primaria (paquete PAR), si el nodo tiene espacio disponible, debe responder ofreciéndole una dirección (paquete PAP), o diciéndole que se le agotó el espacio de direcciones en el caso de que no tuviera para ofrecer.

Una vez enviado el PAP se espera a recibir un PAN que dará a conocer la dirección elegida. Si este paquete no se recibió en un lapso de tiempo, se debe volver al modo de escucha de pedidos para reestablecer el funcionamiento normal, ya que posiblemente algún paquete se perdió.

Cuando se recibe un paquete PAN, éste puede estar indicando que la dirección elegida es la del nodo o que no es. En el primer caso, el nodo debe ceder el espacio de direcciones, cambiando su máscara y notificando a otros procesos que pudieran estar interesados en conocer este hecho (como el algoritmo de ruteo o la aplicación de *Rendez-Vous*) y responder con un paquete PANC al remitente.

Tanto si la dirección escogida es la del nodo como si no lo es, se vuelve al modo de escucha de paquetes.

El algoritmo 2 resume el ofrecimiento de direcciones primarias, siendo T_PAN el tiempo máximo que se espera respuesta.

Algoritmo 2 Ofrecimiento de direcciones primarias)

```
1: Esperar un paquete PAR
2: si hay direcciones para ceder entonces
3:    Enviar un paquete PAP ofreciendo una dirección primaria.
4:    Esperar a recibir un paquete PAN o que transcurra un tiempo T_PAN
5:    si se recibió un paquete PAN y la dirección elegida es la del nodo entonces
6:       Ceder el espacio de direcciones.
7:    fin si
8: si no
9:    Enviar un paquete PAP indicando que no se tienen direcciones.
10: fin si
11: Ir a 1
```

Este algoritmo se repite indefinidamente, y por ello debe correr en un hilo propio, que se debe crear cuando el nodo se conecta y eliminar cuando se desconecta.

La segunda tarea que el nodo debe realizar mientras está conectado consiste en enviar periódicamente paquetes a sus vecinos para informar que sigue estando conectado a la red. La ausencia repentina de estos paquetes indicaría que el nodo se desconectó sin realizar las tareas de desconexion debidas. Los paquetes enviados son de tipo HB (Heard Bit).

El algoritmo 3 muestra como se realiza esta tarea, siendo T_HB el tiempo entre envíos de paquetes HB.

Algoritmo 3 Envío de Heard Bits

```
1: Enviar un paquete HB en broadcast
2: Esperar un tiempo T_HB
3: Ir a 1
```

Este algoritmo tan solo envía paquetes HB en una frecuencia fija, repitiéndose indeterminadamente. Por ello, necesita correr en un hilo propio, que se debe crear cuando el nodo se conecta y eliminar cuando se desconecta.

Los paquetes HB enviados son recibidos en los otros nodos, y su procesamiento es el objetivo de la tarea 3.

Para ello, el nodo debe recibir paquetes durante un tiempo determinado, y todos los nodos que hayan enviado un paquete durante ese lapso serán marcados como activos. Los que permanezcan inactivos durante una cierta cantidad de iteraciones, se darán por desaparecidos, debiendo recuperar su espacio de direcciones si correspondiera. Este procesamiento se detalla en el Algoritmo 4. El parámetro del protocolo T_LHB es el tiempo durante el que se esperan paquetes, y N_HB la cantidad de veces consecutivas que se tolera no recibir un HB antes de dar al nodo por desaparecido.

Algoritmo 4 Procesamiento de Heard Bits

```
1: esperar paquetes HB durante un tiempo T_LHB
2: marcar como activos a los vecinos que enviaron paquetes HB
3: por cada vecino que permanece inactivo por N_INACT iteraciones hacer
4:    recuperar el espacio de direcciones si el vecino era un hijo
5:    eliminarlo de la tabla de vecinos
6:    notificar a otras aplicaciones de la desaparición
7: fin por
8: ir a 1
```

Este algoritmo se ejecuta indefinidamente, por lo que al igual que los algoritmos 2 y 3, debe estar en un hilo propio.

El tiempo T_LHB debe ser mayor al lapso de envío de paquetes T_LB para que siempre se reciba uno en normal funcionamiento. No es necesario que sea mucho mayor ya que en el caso de que se pierda un paquete, será marcado como inactivo en un ciclo pero tendrá posibilidades en el próximo ciclo de enviar un HB sin que sea dado por desaparecido.

En cuanto al ofrecimiento de direcciones secundarias, se debe separar de la asignación de direcciones primarias para no demorar la conexión más de lo necesario.

Se podría eventualmente enviar un pedido de direcciones secundarias y responderlo; sin embargo, dado que una vez conectados los nodos se envian periódicamente paquetes HB, al recibirlo se puede comprobar si hay alguna dirección secundaria para ofrecerle al vecino que envió el paquete. Si es posible ofrecerle una dirección, entonces se envía un paquete SAP (Secondary Address Proposal), y se espera un paquete SAN (Secondary Address Notification) en respuesta.

El algoritmo de escucha de HB (algoritmo 4) se modifica para proponer direcciones secundarias como se muestra en el algoritmo 5.

Algoritmo 5 Procesamiento de HB con propuesta de direcciones secundarias

```
1: esperar paquetes HB durante un tiempo T_LHB
2: marcar como activos a los vecinos que enviaron paquetes HB
3: por cada vecino que permanece inactivo por N_INACT iteraciones hacer
4:    recuperar el espacio de direcciones si corresponde
5:    eliminarlo de la tabla de vecinos
6:    notificar a otras aplicaciones de la desaparición
7: fin por
8: si hay algún nodo al que se le puede proponer dirección secundaria entonces
9:    enviarle un paquete SAP con la dirección propuesta.
10:   esperar un paquete SAN o que transcurra un tiempo T_SAN.
11:   si se recibió un paquete SAN entonces
12:      si se acepta la dirección propuesta entonces
13:         ceder el espacio de direcciones.
14:      si no
15:         marcar que la dirección ya fue ofrecida a ese vecino.
16:      fin si
17:   fin si
18: fin si
19: ir a 1
```

Este algoritmo agrega, luego de la verificación de nodos inactivos, una búsqueda de vecinos a los que se les pueda proponer dirección secundaria, enviándoles un paquete SAP y esperando como respuesta un paquete SAN durante un tiempo limitado por T_SAN.

Desconexión

Cuando el nodo se desconecta, es necesario enviar información a los vecinos para que puedan recuperar el espacio de direcciones facilmente. Asimismo, otras aplicaciones pueden necesitar enviar datos adicionales a otros nodos, como por ejemplo la aplicación de *Rendez-Vous* que necesita pasar las tablas a otros nodos.

Para que el protocolo no deba conocer de antemano que otras aplicaciones necesitan enviar información de desconexión, se utiliza un sistema de desconexión donde se le pregunta a otras aplicaciones si necesitan realizar alguna tarea antes de desconectarse, y en ese caso se espera a que estas finalizen.

Cuando el nodo se va a desconectar, envía un mensaje interno notificando de la desconexión, y espera durante un lapso de tiempo que le respondan quienes necesitan realizar más tareas. Luego, se espera que cada uno de los que respondió anteriormente dé por finalizadas sus tareas, y se envía un paquete DISC

(Disconnection) a los vecinos con la dirección primaria para que recuperen el espacio cedido.

El Algoritmo 6 muestra como se realiza la desconexión, siendo T_WAIT_ME el tiempo que tienen los procesos para hacer el pedido de espera y T_READY_FOR_DISC el tiempo que tienen para concluir sus tareas.

Algoritmo 6 Desconexión

```
1: enviar un mensaje interno WILL_DISCONNECT
2: esperar por un tiempo T_WAIT_ME a que lleguen mensajes WAIT_ME
3: mientras quede algún mensaje WAIT_ME o transcurrió un tiempo T_READY_FOR_DISC hacer
4:    esperar un mensaje READY_FOR_DISC
5:    eliminar el mensaje WAIT_ME correspondiente al READY_FOR_DISC recibido
6: fin mientras
7: enviar un paquete DISC a los vecinos
```

El tiempo T_READY_FOR_DISC es necesario para evitar que si un proceso funciona mal bloquee la desconexión, y debe ser suficientemente largo para que se pueda ceder toda la información necesaria a los vecinos.

2.2.3. Especificación de los estados del protocolo

El desarrollo hecho anteriormente se puede especificar mediante diagramas de estado que muestren las transiciones de un estado a otro, junto con las condiciones que desencadenan los cambios.
Para esto, se necesitaron tres diagramas de estado, mostrando cada uno tareas independientes.

- *Main State Machine*: se encarga de la conexión y desconexión del nodo, y mientras está en estado estable, de recibir los paquetes de tipo SAP y DISC.
- *Primary Address Provider*: espera pedidos de dirección primaria y se ocupa de concretar el pedido.
- *Heard Bit Listener*: esucha paquetes HB y da direcciones secundarias si es necesario.

En los diagramas, algunos estados contienen un rectángulo donde dice cuál es la acción que se efectúa tan pronto como se entra al nodo. Por ejemplo, en el estado *WaitPAP* dice *Broadcast PAR*, es decir que cada vez que se entre al estado desde otro estado o desde él mismo, se envía un paquete PAR en broadcast.

Main State Machine

En la figura 2.11 se presenta el diagrama de estados de *Main State Machine*.
Inicialmente se encuentra en el estado *Disconnected*, en el que permanece hasta que se recibe un mensaje interno *Join Network*. Cuando se recibe este mensaje, se cambia al estado *WaitPAP*, donde inicialmente se manda en modalidad de broadcast un paquete PAR (Primary Address Request). Se sale de este estado cuando se cumple un timeout, yendo a un estado en función de los paquetes PAP (Primary Address Proposal) que se hayan recibido en ese intervalo y de la cantidad de veces que se haya cumplido el timeout:

- si no hay espacio de direcciones disponibles, pasa a *Disconnected*.
- si no se obtuvo respuesta hasta la cuarta vez, entra nuevamente en *WaitPAP*.
- si se recibió al menos una respuesta, se elige la dirección primaria y pasa a *WaitPANC*.
- si no se obtuvo respuesta por quinta vez, se utiliza la dirección compuesta por todos ceros y se pasa a *Stable Address*

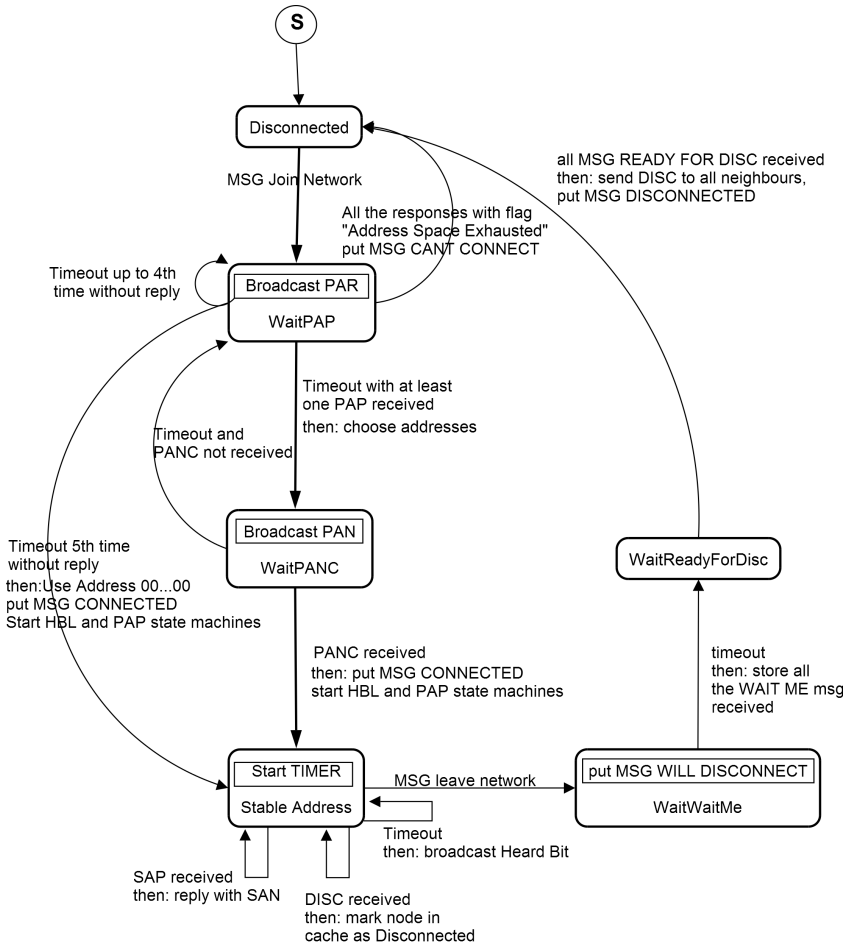


Figura 2.11: Máquina de estados "*Main State Machine*" (Algoritmos 1, 2, 3 y 6)

En el caso de tener respuesta, se entra en el estado *WaitPANC*, donde inicialmente se envía un paquete PAN (Primary Address Notification) en modalidad de broadcast para indicar la dirección elegida, y se espera recibir en respuesta un paquete PANC (Primary Address Notification Confirmation) dentro de un lapso de tiempo. Si no se recibe, se aborta la elección, volviendo al estado *WaitPAP*. Si se recibe, el nodo ya se encuentra conectado y se pasa al estado *Stable Address*.

En el estado *Stable Address* se permanece hasta recibir un mensaje interno *leave network*. Mientras tanto, desde este estado se efectúan ciertas tareas que no implican pasar por otros estados:

- si se recibe un paquete SAP (Secondary Address Proposal), se evalúa si aceptar la dirección secundaria propuesta y se responde con un paquete SAN (Seconday Address Notification)
- si se recibe un paquete DISC (Disconnected), se marca al nodo que lo envió como desconectado.
- cada cierto tiempo se envía en broadcast un paquete HB (Heard Bit), para que los vecinos sepan que el nodo sigue en funcionamiento.

Cuando se recibe el mensaje interno *leave network*, se pasa al estado *WaitWaitMe*, en el cual se le notifica a los otros procesos del nodo que se va a desconectar, mediante el mensaje interno *will disconnect*. Los procesos que requieran efectuar alguna tarea antes de la desconexión deben responder con un mensaje *wait me*. Se sale de este estado luego de un tiempo determinado, pasando al estado *WaitReadyForDisc*.

En el estado *WaitReadyForDisc* se espera a que todos los procesos que mandaron un mensaje *wait me* envíen un mensaje *ready for disc*, indicando que estan listos para desconectarse. Una vez que todos mandaron los mensajes, se pasa al estado *Disconnected*.

Heard Bit Listener

La figura 2.12 muestra el diagrama de estado para *Heard Bit Listener*.

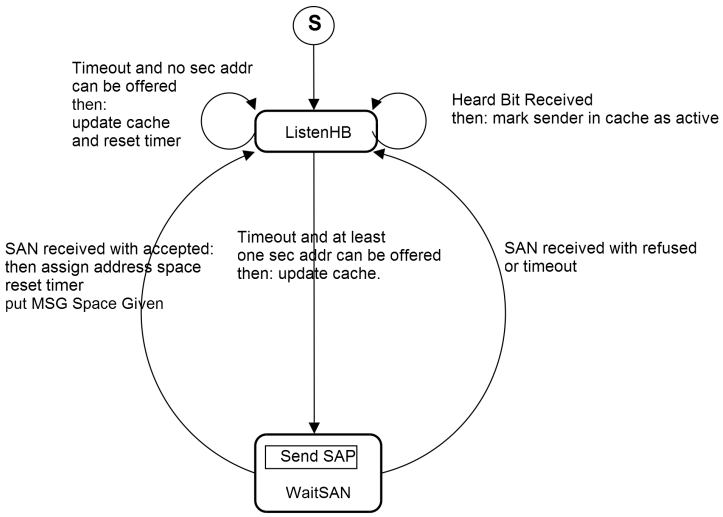


Figura 2.12: Máquina de estados "*Heard Bit Listener*" (Algoritmo 5)

Esta máquina de estados comienza a funcionar cuando en *Main State Machine* se entra al estado *StableAddress*. Inicialmente se encuentra en el estado *ListenHB* por un tiempo fijo, recibiendo paquetes HB (Heard Bit) y marcando a los nodos que enviaron estos paquetes como activos. Una vez que se termina el tiempo, si no se pueden ofrecer direcciones secundarias se vuelve al mismo estado, y si se pueden ofrecer se pasa al estado *WaitSAP*. En ambos casos los nodos que no hayan enviado un paquete HB son marcados como desaparecidos.

En el estado *WaitSAN*, primero se envía un paquete SAP (Secondary Address Proposal) y se espera en respuesta un paquete SAN. Si no se recibe luego de un intervalo de tiempo, se vuelve al estado *ListenHB* sin otorgar direcciones secundarias. En cambio, si se recibe el paquete SAN y la dirección secundaria es aceptada, se cede la dirección secundaria.

Primary Address Provider

La máquina de estados *Primary Address Provider* se muestra en la figura 2.13, y al igual que *HeardBitListener*, arranca cuando en *Main State Machine* se entra al estado *StableAddress*.

Inicialmente se encuentra en el estado *WaitPAR*, esperando paquetes PAR (Primary Address Request). Cuando se recibe uno de estos paquetes se pasa al estado *WaitPAN*.

En el estado *WaitPAN*, primero se envía un paquete PAP (Primary Address Proposal) proponiendo una dirección primaria. Si se recibe como respuesta un paquete PAN (Primary Address Notification) cuya dirección es la misma que la propuesta, entonces se responde con un paquete PANC (Primary Address Notification Confirmation) y se cede el espacio. Tanto en este caso como si no se recibe un paquete PAN en un intervalo de tiempo o si se recibe con otra dirección se vuelve al estado *WaitPAR*.

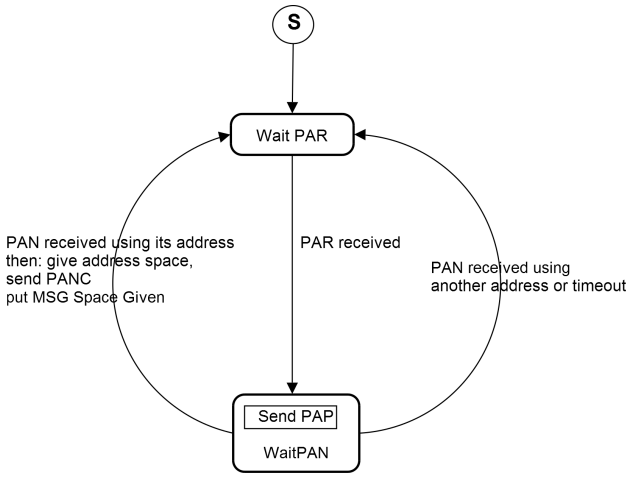


Figura 2.13: Máquina de estados "*Primary Address Provider*"

2.2.4. Estructura de los paquetes de Control

Para la conexión y desconexión de nodos de intercambian paquetes de control, que ya fueron nombrados anteriormente. Ahora ser verá con detalle como están compuestos cada uno de ellos.

En la figura 2.14 se muestra la estructura en común que tienen todos estos paquetes de control.

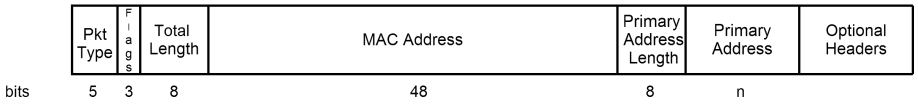


Figura 2.14: Estructura de los paquetes de control

Cada paquete contiene:

- packet type: el tipo de paquete en 5 bits, permitiendo hasta 32 tipos distintos.
- flags: 3 flags de uso general, que cada tipo de paquete puede asignar para lo que necesite.
- total length: longitud total del paquete, incluyendo encabezados opcionales.
- MAC address: dirección física del nodo que envía el paquete. Se utiliza para identificar unívocamente a los nodos cuando aún no poseen una dirección de hipercubo.
- Primary Address Length: longitud en bits de la dirección primaria (dimensión del hipercubo)
- Primary Address: dirección primaria, almacenada en n bits, siendo n el mínimo múltiplo de 8 mayor que Primary Address Length
- Optional Headers: cada paquete puede transportar encabezados opcionales.

El formato general de un encabezado opcional se muestra en la figura 2.15.

Al igual que en el formato de un paquete, se permiten 32 tipos distintos (los tipos de los paquetes y los encabezados opcionales no guardan relación) y se cuenta con 3 flags de uso general. Por el momento, el protocolo define solo un tipo de encabezado opcional, *Additional Address*, cuyo propósito es enviar más direcciones de hipercubo, siendo su significado dependiente del paquete. Este encabezado es utilizado en los paquetes de tipo `PAP`, `SAP` y y `SAN`, y su significado se detalla en la explicación de cada paquete, a continuación en esta sección. El formato de *Additional Address* se puede ver en la figura 2.16

Los valores de los campos son:

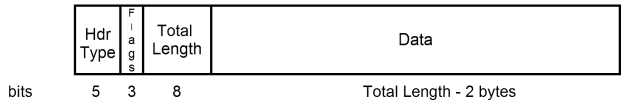


Figura 2.15: Estructura de un encabezado opcional

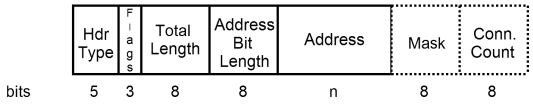


Figura 2.16: Estructura del encabezado opcional *Additional Address*

- Header Type: 1 para este encabezado opcional
- Flags:
  - flag 0: indica si el encabezado tiene el campo *Mask*
  - flag 1: indica si el encabezado tiene el campo *Connection Count*
- Total Length: longitud total del encabezado
- Address Bit Length: longitud en bits de la dirección primaria (dimensión del hipercubo)
- Address: dirección primaria, almacenada en n bits, siendo n en mínimo múltiplo de 8 mayor que Address Bit Length
- Mask: si el flag 0 está seteado, se incluye este campo que indica la máscara asociada a la dirección.
- Connection Count: si el flag 1 está seteado, se incluye este campo que indica la cantidad de conexiones que tiene el nodo con esa dirección.

Paquete PAR (Primary Address Request)

Se utilizan para hacer un pedido de dirección primaria a los vecinos. Se envían siempre en modalidad de broadcast. Se espera como respuesta un paquete PAP (Primary Address Proposal), tanto para ofrecer dirección como para informar que no se dispone del espacio necesario.

- Type: 1
- Primary Address Length: 0, para indicar que no se envía dirección primaria

Paquete PAP (Primary Address Proposal)

Se envía este paquete en respuesta a un paquete PAR para proponer direcciones de conexión

- type: 2
- flag 0: si vale 1, es porque no tiene dirección para ceder.
- Optional Headers: Si el flag 0 vale 0, entonces hay al menos 1 encabezado de tipo *Additional Address*. El primero corresponde a la dirección primaria propuesta, y si hubiera más de uno, los siguientes encabezados son direcciones de reconexión.

Paquete PAN (Primary Address Notification)

Una vez que un nodo eligió la dirección primaria con la que se va a conectar, envía este paquete en modalidad de broadcast para indicarle a sus vecinos su nueva dirección, y que su padre le ceda el espacio.

- Type: 3

Paquete PANC (Primary Address Notification Confirmation)

Lo envía como respuesta a un paquete PAN el nodo que cede el espacio de direcciones, confirmando que efectivamente recibió el paquete PAN.

- Type: 4

Paquete DISC (Disconnection)

Se envía para notificar que el nodo se está desconectando, en modalidad de broadcast.

- Type: 5

Paquete HB (Heard Bit)

Se debe enviar a intervalos regulares para notificar que el nodo sigue conectado, en modalidad de broadcast.

- Type: 6

Paquete SAP (Secondary Address Proposal)

Se envía para proponer una dirección secundaria.

- type: 7
- Optional Headers: Un encabezado del tipo *Additional Address*, conteniendo la dirección secundaria propuesta.

Paquete SAN (Secondary Address Notification)

Se envía como respuesta a un paquete SAP para indicar si la dirección secundaria fue aceptada o rechazada.

- type: 8
- flag 0: si vale 1, es porque la dirección secundaria fue aceptada.
- Optional Headers: Un encabezado del tipo *Additional Address*, conteniendo la dirección secundaria que se acepta o rechaza.

2.3. Ruteo Reactivo

El *ruteo* consiste en dirigir los paquetes que envía un nodo con destino a cualquier otro nodo en la red, debiendo para ello pasar por una serie de nodos intermedios. En general, las redes no son estáticas, sino que cambian con el tiempo, y eventualmente podrían ser muy grandes, es por ello que saber la ruta de antemano no siempre es factible o conveniente. El ruteo que cuenta con tener la ruta de antemano es el Ruteo Proactivo, conveniente para redes pequeñas y estables. En este caso, los cambios en la topología se deben propagar para que toda la red sepa cuál es la ruta para ir a cada nodo.

El ruteo se puede aplicar fácilmente en un hibercubo completo, simplemente yendo por cualquiera de los caminos que lo acerquen al destino final, lo cual es posible porque todas las aristas están presentes y cualquiera de los bits que componen la dirección puede ser cambiado. Entonces, dado que en cada paso

la distancia disminuye en uno, siempre se llegará al destino en tantos pasos como bits diferentes haya entre la dirección de origen y de destino.

Dado que en la práctica no es factible tener un hipercubo completo (ver sección 2.1.1), es necesario encontrar un método de ruteo que permita adaptarse a la topología existente a la vez que sea capaz de aprovechar las características del hipercubo.

Una alternativa es el Ruteo Reactivo, donde las rutas no son conocidas de antemano, sino que se van construyendo a medida que se necesitan, es decir que cada nodo dirige al paquete hacia el siguiente nodo intermedio, sin saber como debe seguir después, pero intentando orientarlo hacia el destino final. Esto no garantiza que el camino elegido sea óptimo en cuanto al tiempo o cantidad de nodos, pero se puede lograr una buena aproximación.

Los requerimientos que debe cumplir el protocolo de ruteo reactivo son:

1. Garantizar que si existe alguna ruta desde el origen al destino, el paquete la encuentre.
2. Minimizar el número de nodos intermedios en la ruta.
3. Evitar que un paquete pueda quedar dando vueltas indefinidamente dentro de la red.
4. Una vez guardada una ruta, mantenerla en memoria un cierto tiempo para que otros paquetes puedan aprovecharla.
5. Inspeccionar nuevas rutas periódicamente para descubrir otros caminos que pueden ser más cortos.

El primer requerimiento es una condición para que el algoritmo tenga sentido, ya que si sólo cubriera algunos casos, no sería de utilidad práctica.

Con el segundo requerimiento se busca obtener una eficiencia razonable, ya que el tiempo de envío en general crece cuando aumenta el número de nodos intermedios en la ruta. Si bien depende de otras cosas, como de la conexión entre los nodos y su carga, estos parámetros son mucho más dificiles de medir y además cambian con el tiempo.

El tercer requerimiento es necesario para evitar que alguna falla pueda traer aún más problemas sobrecargando inútilmente la red.

El siguiente requerimiento se debe a que frecuentemente dos nodos se conectan para intercambiar más de un paquete, por lo que mantener la información de las rutas por un cierto tiempo aumentará el rendimiento a costa de un pequeño espacio en memoria.

El último requerimiento es para que el algoritmo pueda aprovechar los cambios en la topología, como por ejemplo un nuevo nodo que ingresa y permite un camino más corto.

2.3.1. Solución propuesta

La ventaja de usar el hipercubo como espacio de direcciones consiste en la relación existente entre la topología y las direcciones asignadas. En principio, en un hipercubo completo, la distancia entre dos nodos es la cantidad de bits en que difieren las direcciones. Por ejemplo, la distancia entre 0010 y 1001 es 3, ya que 3 bits son diferentes. El ruteo en este caso consiste simplemente en ir cambiando de a uno los bits distintos para obtener las direcciones de los nodos intermedios, es decir que una posible ruta sería 0010, 1010, 1000, 10001.

Sin embargo, en el caso real de un hipercubo completo, no necesariamente existe esa ruta. La cantidad de bits diferentes entre origen y destino es ahora la distancia mínima. Como punto de partida del ruteo reactivo, se busca entonces que el paquete se acerque cada vez más al destino, es decir, enviarlo al nodo intermedio con la menor distancia posible al destino.

La figura 2.17 muestra un ejemplo de una pequeña red de hipercubo incompleto. Utilizando como única regla de ruteo enviar siempre al vecino que más acerque al destino final, se verán tres ejemplos.

Ejemplo 1: el nodo $0000m2(a)$ le manda un paquete a $0110m4(e)$. La distancia mínima es 2, ya que hay 2 bits distintos entre ambas direcciones. Inicialmente, el paquete se puede enviar a $1000m2(b)$ o a

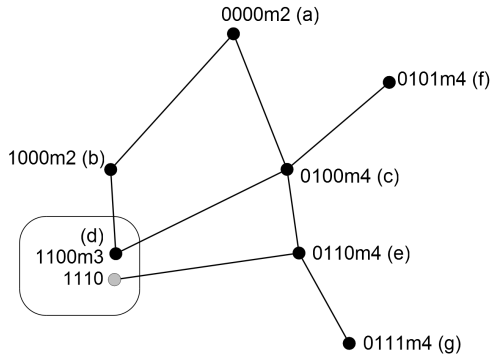


Figura 2.17: Red de Hipercubo

$0100m4(c)$, siendo las distancias entre cada uno de ellos y el destino final de 3 y 1 respectivamente. Notar que al pasar de un nodo a otro, la distancia puede o bien incrementarse o bien decrementarse en uno; nunca se mantiene constante o cambia en más de uno. El próximo destino es entonces $0100m4(c)$. Este nodo ya es vecino del destino final así que lo puede enviar directamente. En este ejemplo, se recorrió la distancia mínima dada por la diferencia de bits entre las direcciones de origen y destino.

Ejemplo 2: el nodo $0101m4(f)$ le manda un paquete a $0111m4(g)$. La distancia mínima es 1 (es decir que podrían ser vecinos), sin embargo no se encuentran conectados, por lo que el paquete debe pasar primero por $0100m4(c)$, donde la distancia al destino final es 2. De los vecinos de $0100m4(c)$, los que tienen la distancia mínima al destino son $0110m4(e)$ y $0101m4(f)$, ambos con distancia 1. En este caso no tiene sentido que el paquete vuelva por donde vino, por lo que va al nodo $0110m4(e)$ y de ahi finalmente llega a $0111m4(f)$.

Ejemplo 3: el nodo $0100m4(c)$ le manda un paquete a $0111m4(g)$. El nodo puede enviarlo tanto a $0101m4(f)$ como a $0110m4(e)$, pero no sabe de antemano que $0101m4(f)$ no conduce al destino final. Si el nodo decide mandarlo por ahi, el paquete se encontrará con que no tiene otro nodo para dónde ir, excepto volver atrás, por lo que $0101m4(f)$ lo mandará de vuelta a $0100m4(c)$. Si este nodo no reconoce que el paquete esta siendo devuelto, podría ocurrir que lo mande de nuevo a $0101m4(f)$, y el paquete quedaría yendo y viniendo entre esos dos nodos indefinidamente.

Como muestra el último ejemplo, es necesario algún mecanismo que indique cuando un paquete está siendo devuelto. Se proponen dos aproximaciones para lograr esto; una consiste en identificar en cada nodo los paquetes que pasaron para poder ver si están volviendo, y la otra es llevar esta información en el paquete. La primera técnica requiere identificar unívocamente los paquetes, así como almacenar y limpiar periódicamente información en el nodo, mientras que la segunda técnica tan sólo requiere un bit de información en el paquete, por ello se opta por esta última técnica. Este bit se lo llama flag de *returned*.

Utilizando este método, el ejemplo 3 quedaría:

Ejemplo 3 (versión 2): el nodo $0100m4(c)$ le manda un paquete a $0111m4(g)$. El nodo puede enviarlo tanto a $0101m4(f)$ como a $0110m4(e)$, y elige enviarlo al primero de ellos, dónde encuentra que no tiene otro camino que volver atrás, por lo que es enviado de regreso con el flag *returned* activado. Al reconocer que el paquete esta siendo devuelto, el nodo $0100m4(c)$ descarta $0101m4(f)$ como posible camino y lo envía por $0110m4(e)$, que a su vez lo envía al destino final.

Visited Bitmap

La solución presentada con el bit de *returned* no es suficiente aún para garantizar que el paquete llegue: por ejemplo en el caso que los vecinos que aparentemente acercan al destino final no conduzcan a él, la solución será pasar por un vecino que en principio aleja al paquete de la dirección de destino. Por lo

tanto, se necesita algún mecanismo capaz de marcar cuáles son los caminos que no conducen al destino para descartarlos y probar alternativas. Con este fin, se implementó el sistema de *Visited Bitmap*.

El *Visited Bitmap* consiste en un vector de ceros y unos, tantos como vecinos tenga el nodo, donde el cero indica que el paquete no fue ruteado por ese vecino, y el uno indica que si fue ruteado. Es necesario un *Visited Bitmap* distinto para cada destino, ya que los paquetes se rutean en función de la dirección de destino. Además, se debe almacenar un mapa de correspondencia para indicar a que vecino hace referencia cada elemento (es decir, bit) del bitmap, siendo este mapa único por nodo. En la figura 2.18 se muestra un ejemplo de *Visited Bitmap* junto con el mapa asociado.

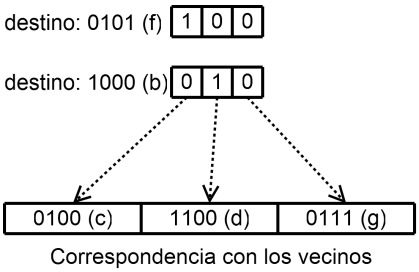


Figura 2.18: Ejemplo de *Visited Bitmap* para el nodo 0110m4 (e)

En este ejemplo, se ve el mapa de correspondecia para el nodo $0110m4(e)$, indicando que el primer bit del *Visited Bitmap* corresponde al vecino $0100(c)$, el segundo a $1100(d)$ y el tercero a $0111(g)$. Luego, hay *Visited Bitmap* para dos destinos, indicando el primero que solo $0100(c)$ fue visitado, y el segundo que $1100(d)$ fue visitado.

En el caso de que un nuevo vecino se conecte al nodo, su dirección se pone al final del mapa de correspondencia, y a todos los *Visited Bitmap* se les agrega un bit más de espacio al final con valor 0.

Cuando un nodo se va, se podría eliminar su dirección del mapa de correspondencia, así como los bits en cada uno de los *Visited Bitmap*. Sin embargo esto puede ser una operación relativamente costosa, ya que implicaría desplazar bits en todos los *Visited Bimtaps*, por lo que es preferible simplemente marcar a los vecinos desconectados, lo que se puede lograr mediante otro vector de bits, donde un cero indica que el vecino no se encuentra conectado, y un uno indica que lo está. Entonces, al desconectarse un vecino, alcanza con poner el bit correspondiente en 0. Además, se debe tener en cuenta en todos los algoritmos que ese vecino debe ser salteado.

Si se producen frecuentemente conexiones y desconexiones, quedará mucho espacio inutilizado en estos vectores, por lo que es recomendable realizar limpiezas períodicamente, lo cual consiste en eliminar del mapa de correspondencias a los nodos desconectados, así como en los *Visited Bitmap*.

Otro punto a tener en cuenta es que con la conexión de nuevos nodos podría ocurrir que un vecino que anteriormente no conducía a cierto destino pase a ser un camino válido. Es por ello que resulta conveniente limpiar el *Visited Bitmap* períodicamente, posibilitando de esta forma que se recorran nuevas rutas.

Orden de exploración de los vecinos

Como se dijo al principio, lo primero que se debe intentar es enviar el paquete por los vecinos que acerquen al paquete al destino. Si el paquete se devuelve, se deberá probar por otros vecinos, por lo que es necesario establecer un orden de exploración. Este orden consiste en probar primero hasta un número NBP (*Neighbours Before Parent*)de vecinos. Si luego de eso el paquete aún vuelve, se prueba enviarlo por el padre, y si vuelve nuevamente, se hace una búsqueda exhaustiva entre todos los vecinos restantes. Si se exploraron todos los vecinos y no se encontró camino por ninguno, entonces el paquete es enviado por la ruta inversa indicando que se devuelve. Esta forma de recorrer la red es conocida como *exploración en profundidad* o *backtracking*.

Para facilitar el desarrollo de algoritmo de exploración que se verá posteriormente, el padre del nodo siempre debe ocupar el primer lugar en el mapa de correspondecias entre vecinos y *Visited Bitmap*.

Almacenamiento de rutas

Uno de los requerimientos para el ruteo es poder almacenar durante cierto tiempo las rutas para reutilizarlas. El *Visited Bitmap* no es suficiente para tal fin, ya que no indica cuál de los vecinos se está utilizando actualmente. Entonces, se debe guardar también cuál es la ruta actual para cada destino. Toda esta información se almacena en la tabla de ruteo, que tiene el formato que se muestra en la figura 2.19.

Destino	**Ruta**	**Distancia**	**Visited Bitmap**
0101m4	0100	2	110
1000m2	1100	-	010

Figura 2.19: Ejemplo de una tabla de ruteo

La tabla de ruteo contiene además un campo de distancia al destino. Para obtener estos valores, se utilizan las *entradas inversas* en la tabla del ruteo, que consiste en agregar una entrada donde el destino es en realidad el origen del paquete, la ruta es el vecino del cuál provino el paquete, y la distancia se calcula en base al *Time To Live* del paquete, que se explicará posteriormente.

Por ejemplo, un paquete que proviene de $1000m2(b)$ y llega a través del nodo $1100m3(d)$ hasta $0110(e)$, genera una entrada inversa poniendo como destino su origen ($1000m2(b)$), como ruta 1100 y 2 en la distancia. Esta es la segunda entrada de la tabla de ruteo que se dio como ejemplo. Al utilizar entradas inversas, cuando un paquete va de un nodo a otro, no sólo queda construida la ruta en ese sentido sino que también se construye la ruta de vuelta. Dado que muy frecuentemente el nodo que recibe un paquete le responde al remitente, es bastante probable que almacenar esta información extra sea útil.

El cálculo de la distancia al origen se hace, como se mencionó anteriormente, mediante el campo *Time To Live* del paquete (ver sección 2.3.3). Este campo comienza valiendo una constante, MAX_TTL cuando el nodo es enviado, y cada vez que un nodo rutea al paquete, debe decrementar este valor, excepto en el caso de que el paquete esté siendo enviado de regreso, debiendo en este caso incrementarlo para volver al estado anterior. La distancia al origen se puede calcular en cualquier nodo como $TTL\_MAX - TTL$.

La función principal del campo *Time To Live* es en realidad evitar que algún paquete pueda quedar dando vueltas indefinidamente en la red. Si bien este caso no debería presentarse normalmente, algún nodo que no esté funcionando correctamente o cierto patrón de conexión y desconexión de nodos podrían generarlo. Para ello, si el valor de este campo llega a 0, el paquete es descartado.

Detección de bucles

Cuando un paquete está explorando rutas, podría ocurrir que llegue a un nodo por el que ya pasó. En este nodo, encontraría una ruta marcada y la seguiría, entrando de esta forma en un bucle del que no saldría en ningun momento, por lo que el paquete quedaría dando vueltas en la misma ruta hasta que el *Time To Live* llegue a 0 y sea descartado. Para no llegar a esta situación, se analizaron varias alternativas:

- Guardar la ruta en el paquete: consiste en ir guardando la dirección de cada nodo por el que se pasa en el encabezado del paquete. Se podría comprimir la información escribiendo solo el número de bit que cambió al hacer el salto de un nodo a otro. De todas formas, esto requiere enviar más información en el paquete, y cada vez que se llega a un nodo, el procesamiento adicional para saber si había pasado anteriormente o no.

- Guardar los paquetes que pasaron por cada nodo: se deben identificar los paquetes (por ejemplo con un número de secuencia), así el nodo puede almacenar una identificación para controlar después si el paquete ya había pasado. Requiere información adicional en el nodo (que se debería limpiar periódicamente), y procesamiento cada vez que un paquete llega.

- Distancia al origen: como el nodo puede conocer la distancia al origen, si posteriormente llega un paquete del mismo origen con una distancia mayor, es probablemente porque encontró un bucle.

Sin embargo, podría ocurrir que un cambio en la topología haga que el camino hasta el nodo sea más largo, detectando un bucle donde no lo hay, y, en el peor de los casos, informando que una ruta no existe cuando si existe.

- Detección en segunda vuelta: en el caso que hubiera un bucle, el paquete habrá dejado marcado en los nodos esa ruta, volviendo a recorrer el bucle una y otra vez hasta que finalice su tiempo de vida (`TTL`). Cuando el paquete da la primera vuelta, no se puede estar seguro si realmente hizo un bucle o es otro paquete (del mismo origen y al mismo destino) que vino por una ruta diferente debido a un cambio de topología. Sin embargo, si el paquete aparece nuevamente con una distancia al origen aún mayor (siendo múltiplo del primer bucle), se puede suponer con gran certeza que se encuentra en un bucle. Existe la posibilidad de que se hayan producido dos cambios de topología, incrementando en cada caso la distancia, sin embargo esto es muy poco probable, y en tal caso, las capas superiores podrían recuperar el error.

Los primeros dos métodos se descartaron principalmente por requerir mucho procesamiento de información para cada paquete. El tercer método surgió como alternativa para evitar el procesamiento y almacenamiento de información en el nodo; sin embargo presenta una vulnerabilidad ante un cambio de topología, que podría ocurrir de vez en cuando. Por ello se desarrolló el cuarto método, que busca disminuir la probabilidad de fallo a cambio de hacer que el paquete recorra una vez más el bucle.

Para implementar este método se permite almacenar dos veces una entrada para un mismo destino, cuando su distancia sea mayor que la que se tenía almacenada previamente en al menos 3, ya que el bucle más corto tiene distancia 3.

En la figura 2.20 se muestra como quedaría la tabla de ruteo si llega desde $0111m4$ un paquete destinado a $0101m4$ con distancia 9.

Destino	**Ruta**	**Distancia**	**Visited Bitmap**
0101m4	0100m4	2	110
1000m2	1100m3	-	010
0101m4	0111m4	9	110

Figura 2.20: Tabla de ruteo con dos entradas al mismo destino

La tabla de ruteo muestra que un paquete con origen $0101m4$ llegó en sólo 2 pasos, mientras que un paquete (que puede ser el mismo o no), llegó desde el mismo nodo en 9 pasos. Esto puede ser o bien porque hubo un cambio en la topología, o bien porque el paquete entró en un bucle y volvió. En principio no se supone nada y se rutea nuevamente.

Si llega otro paquete con el mismo origen pero con distancia 16, es altamente probable que haya un bucle de longitud 7 y que el paquete esté en él. En tal caso se debe cambiar el campo *Time To Live* de forma tal que su distancia figure como 2, y se debe probar enviar el paquete por un vecino distinto al que se está usando actualmente.

En cambio, si llega un paquete con el mismo origen pero con otra distancia, se supone que esto es solamente por cambio de topología, y se conservan las entradas más recientes.

Este método se implementó y simuló, pero se encontró que en ciertas situaciones, los paquetes se perdían porque un bucle no se detectaba. Esto ocurría cuando quedaba grabada una entrada antigua, y luego se entraba en un ciclo, como se muestra en la figura 2.21.

Destino	**Ruta**	**Distancia**	**Visited Bitmap**
0101m4	0100m4	6	110
0101m4	0111m4	9	110

Figura 2.21: Bucle no detectado

En este ejemplo, en algún momento se llegó al nodo con distancia 6, más tarde se llega con distancia 9, y si entra en un bucle de longitud 5, llega nuevamente con distancia 14, por lo que el bucle no es detectado. Se evaluaron distintas formas de corregir esto, como por ejemplo eliminar la entrada mínima; pero de esta forma se perdía la ruta para salir del bucle; otra posibilidad era guardar más de 2 entradas,

pero esto haría que si casualmente llegan paquetes con distancias equiespaciadas, se detecte un bucle donde no lo hay.

Finalmente, se decidió utilizar el método de distancia al origen pero comprobando que el mensaje tenga el mismo destino también. Es decir, se considera que un paquete está en un bucle cuando las tablas de ruteo contienen entradas para un paquete con el mismo origen y destino, pero con una distancia al origen menor. Al verificar el destino, se disminuye la probabilidad de falsos positivos en la detección de bucles, ya que si los destinos no coinciden, evidentemente el paquete no era el mismo. Por ejemplo, si primero llega un paquete $M(x, z)$ (siendo x el nodo de origen y z el nodo de destino) con distancia al origen de 5 saltos, y luego llega un paquete $M(x, w)$ con distancia al origen 8, no se considera que hubo un bucle. En cambio, si se consideraría un bucle si el paquete fuera $M(x, z)$ con distancia al origen 8.

Para implementar la verificación del destino, se utiliza una tabla de pares, donde se guardan los pares de nodos (origen, destino) que fueron ruteados, con punteros a las entradas en la tabla de ruteo para el origen y el destino. En la siguiente sección se presenta el algoritmo de ruteo completo.

2.3.2. Algoritmo de ruteo

El desarrollo propuesto anteriormente se puede resumir en un algoritmo de ruteo, que por conveniencia se divide en dos funciones.

La primera función es $route(M, w)$, que se muestra en el algoritmo 7. El primer parámetro, M, es el paquete que debe rutear, que se puede escribir también como $M(x, z)$, siendo x el nodo de origen y z el nodo de destino. El segundo parámetro, w, es el vecino por el cuál se recibió el paquete.

La segunda funcíon es $sendToNextNeighbour(M, w)$, que se muestra en el algoritmo 8, tomando los mismos parámetros que la primera función.

Algoritmo 7 route(M, w)

1: v recibe un mensaje $M(x, z)$ del vecino w
2: **si** M está marcado como devuelto **entonces**
3: $sendToNextNeighbour(M, w)$
4: fin del algoritmo
5: **fin si**
6: **si** se encuentra en la tabla de pares a (x, z) con distancia al origen menor que M **entonces**
7: devolver el paquete por donde vino
8: fin del algoritmo
9: **fin si**
10: buscar en la tabla de ruteo $v \rightarrow x : w, n$ (entrada inversa) y agregarla si no se encuentra
11: buscar en la tabla de ruteo $v \rightarrow z : y, m$
12: **si** se encontraron entradas **entonces**
13: enviar el paquete por la entrada con mínima distancia.
14: **si no**
15: $sendToNextNeighbour(M, w)$
16: **fin si**
17: agregar en la tabla de pares a (x, z) si no estaba anteriormente.

En los pasos 2 a 5, se verifica si el paquete está marcado como devuelto, lo cual ocurre si no se encontró ninguna ruta yendo por ese vecino. Por ello, se llama a la función *sendToNextNeighbour*, que busca enviar al paquete por otra ruta.

Los pasos 6 a 9 detectan bucles. Para ello, buscan si un paquete con el mismo origen y destino pasó previamente con una distancia al origen menor, en cuyo caso el paquete es enviado de vuelta por el vecino del que provino.

El paso 10 se agrega la entrada inversa si no estaba; es decir, se agrega la ruta para ir al nodo de origen.

El paso 11 busca la entrada directa, que indica por que vecino se debe ir hacia el destino. En los pasos 12 a 16, si existe esta entrada, se manda el paquete por el vecino especificado. Una vez que la ruta se encuentra correctamente establecida, este es el flujo de programa que siguen los paquetes subsecuentes. En cambio, si no se encuentra la ruta, se utiliza la función *sendToNextNeighbour* para que mande el paquete por el vecino más conveniente.

En el paso 17 se asocia el origen con el destino y la distancia al origen para poder detectar bucles.

Algoritmo 8 sendToNextNeighbour(M, w)

1: **si** no existe el *Visited Bitmap*. de vecinos para $v \rightarrow z$ **entonces**
2: crear el *Visited Bitmap*, y marcar al vecino w como visitado.
3: iniciar Bitmap Timer (entrada) {eliminará sólo el *Visited Bitmap* de la entrada}
4: **fin si**
5: **si** están todos los bits marcados **entonces**
6: marcar en la entrada de la tabla de ruteo que no hay ruta para z
7: buscar en la tabla de ruteo $v \rightarrow x : y, n$
8: devolver el paquete a y
9: fin del algoritmo.
10: **fin si**
11: **si** hay NBP +1 bits marcados **entonces**
12: NextHop = primer bit {padre}
13: **si no, si** el primer bit está marcado **entonces**
14: NextHop = siguiente bit no marcado
15: **si no, si** el primer bit no está marcado **entonces**
16: NextHop = bit no marcado correspondiente al vecino con mínima distancia al destino en el *Visited Bitmap*
17: **fin si**
18: iniciar Route Timer (entrada) {eliminará toda la entrada en la tabla de ruteo}
19: marcar NextHop en el *Visited Bitmap* y enviar el paquete por ese nodo.

El algoritmo 8 se utiliza para determinar por que vecino se debe enviar el paquete la primera vez que llega un paquete con ese destino, o cuando uno es devuelto porque no se encontró ruta.

En los pasos 1 a 4, si no hay *Visited Bitmap* (lo cual ocurre cuando se envía un paquete a un destino por primera vez), éste es creado y se inicializa el *timeout*.

En los pasos 5 a 10, si no se encontró ruta por ninguno de los vecinos, el paquete es devuelto por el nodo desde donde provino.

Los pasos 11 a 17 determinan cuál es el vecino a visitar, según lo explicado en 2.3.1.

El paso 18 inicializa el *timeout* para que la entrada se borre transcurrido un lapso de tiempo.

En el último paso, se marca el vecino por el que se está enviando el paquete como visitado y se envía.

Entonces, los algoritmos 7 y 8 en conjunto, y la utilización del TTL en los paquetes, cumplen con los requerimientos enunciados en 2.3.

2.3.3. Paquetes de datos

El módulo de ruteo de la capa de red le presenta un servicio de envío y recepción de segmentos a la capa de transporte. Para ello, se define un formato de paquetes de datos, siendo en realidad los datos el segmento que se transporta, cuyo contenido no le es pertinente.

La distinción entre paquetes de control y paquetes de datos se realiza en la capa de enlace de datos (*data link layer*), ya que ésta capa está basada en *Ethernet II* y por lo tanto contiene un identificador

(*Ethernet Type*) que indica cuál capa de red debe procesarlo. Los paquetes de control utilizan el identificador 1000, mientras que los paquetes de datos utilizan 1001.

Los paquetes de datos fueron definidos con el formato que se muestra en la figura 2.22.

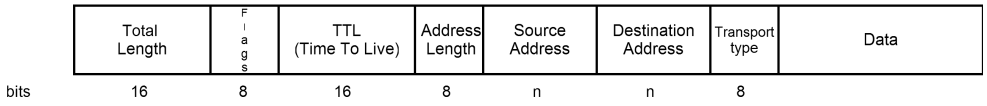


Figura 2.22: Formato de un paquete de datos

Los campos que contiene el paquete son:

- total length: longitud total incluyendo encabezado y datos
- flags: 8 bits disponibles como flags.
  - flag 0 (*isReturned*): indica si el paquete se está ruteando como devuelto.
  - flag 1(*isRendezVous*): indica si el paquete transporta datos de *Rendez-Vous*.
- TTL (Time To Live): indica cuantos saltos más puede realizar el paquete antes de ser descartado
- address length: longitud en bits de las direcciones de origen y destino
- source address: dirección de origen del paquete
- destination address: dirección de destino del paquete
- transport type: identificador del protocolo de transporte que debe procesar el segmento
- data: datos del paquete

El flag *isReturned* es usado por el algoritmo de ruteo, como se vió anteriormente. El flag *isRendezVous* es necesario para poder rutear de forma distinta los paquetes de *Rendez-Vous*, ya que el destino de un paquete *Rendez-Vous* no necesariamente existe, sino que se encuentra dentro del espacio de direcciones de otro nodo. Si bien esto produce cierto acoplamiento entre capas, lo cual no es deseable, es inevitable tener que distinguir estos paquetes.

Otro campo que merece ser comentado es *transport type*. Dado que distintas capas de transporte (UDP, TCP) pueden enviar paquetes usando los servicios provistos por el ruteo, es necesario que al llegar a destino sea procesado por una capa de transporte del mismo tipo, ya que UDP no entendería un segmento TCP y viceversa. Según el valor de este identificador se llamará a la capa de transporte que corresponda.

2.4. Registro de Direcciones (*Rendez-Vous*)

Cada nodo necesita tener un nombre para ser reconocido exteriormente, llamado identificador universal, y debe ser conocido por los nodos que se quieran comunicar con él. Este identificador puede ser una cadena de carácteres, un número, dirección IP, etc. Cuando el nodo se conecta a la red, se le asigna una dirección lógica que es la utilizada para enviar y recibir información, y en principio no tiene por qué ser conocida por los nodos que quieran contactarlo. Es por ello que surge la necesidad de un mecanismo que permita conocer la dirección de red a partir de la dirección universal, como se explicó en la sección 2.1.2.

Con este fin, el protocolo presenta un mecanismo de ruteo indirecto, en el cual, cuando un nodo desea comunicarse con otro, debe obtener su dirección lógica a partir de un nodo capaz de brindar el servicio de traducción de direcciones universales a lógicas. El protocolo usa un modelo distribuido, donde cualquiera de los nodos debe ser capaz de prestar este servicio (*DHT*, *Distributed Hashing Tables* [1]).

Para el registro de direcciones, los requisitos son:

1. los nodos se deben registrar al *Rendez-Vous* cuando se conectan y des-registrar cuando se desconectan

2. distribuir las tablas entre todos los nodos de la red

3. cuando un nodo se desconecta, debe ceder sus tablas para que no se pierdan

El primer requisito indica que cada nodo es responsable de darle su dirección al nodo de *Rendez-Vous* correspondiente y de avisar de su partida.

El segundo requisito es para que el protocolo sea descentralizado; no hay servidores centrales a cargo de resolver direcciones, sino que todos los nodos cumplen esta función.

El último requisito es fundamental para garantizar el buen funcionamiento del protocolo cuando un nodo se desconecta, ya que de otra forma, se perdería información vital para poder encontrar nodos.

No se contempla el caso de que un nodo se desconecte súbitamente, por ejemplo por un corte en las comunicaciones o de energía, quedando para estudios posteriores.

2.4.1. Análisis de la solución

Para saber cuál es el nodo responsable de traducir la dirección universal U a una dirección de red, se le aplica una función de hashing a esta dirección, debiendo estar el valor de retorno en el espacio de direcciones del hipercubo. Es decir, $E = H(U)$, siendo E la dirección del nodo de *Rendez-Vous* y H una función de hashing, que se debe diseñar de forma tal que distribuya el espacio U en E de la forma más homogénea posible para evitar sobrecargar nodos.

Cuando un nodo se conecta a la red, debe informarle a su nodo *Rendez-Vous* cuál es su dirección de red. Para ello, le envía un paquete *Register* con su dirección universal y de red. El nodo *Rendez-Vous* debe almacenar estos valores en una tabla.

Cuando el nodo se desconecta, debe mandarle un paquete *Deregister* al nodo *Rendez-Vous* con su dirección universal para que éste lo pueda eliminar de su tabla.

Para hacer una petición de resolución de dirección, el nodo debe primeramente calcular la dirección del nodo *Rendez-Vous*, para luego enviarle un paquete *Address Solve* indicando la dirección universal que necesita resolver. El nodo *Rendez-Vous* busca la dirección universal en su tabla, y envía como respuesta un paquete *Address Lookup* al remitente indicando la dirección de red o diciendo que no se encontró ningún nodo con esa direccíon universal. Cuando este mensaje le llega al remitente, este ya conoce la dirección de red del nodo que desea contactar y puede enviarle el paquete.

Si un nodo se desconecta, no puede dejar que sus tablas se pierdan, por lo que debe cedérselas a un nodo, eligiéndose el padre para favorecer las re-conexiones. En este caso, le manda un paquete *Lookup Table* al padre, dónde se encuentra la tabla de resolución entera. El padre deberá responder con un mensaje *Lookup Table Received*, para que el nodo pueda estar seguro de que fue recibida correctamente y que puede proceder a desconectarse.

2.4.2. Implementación

El registro de direcciones podría implementarse a nivel de capa de red; pero haciéndolo en la capa de aplicación se tiene la ventaja adicional de poder utilizar la capa de transporte de abajo, la cual permitiría tener una comunicación más confiable si se usa TCP. Como desventaja, las otras aplicaciones en el nodo deberán comunicarse primero con esta aplicación para resolver la dirección de red.

En realidad, se utlizan dos aplicaciones, una aplicación cliente y una servidor. La primera se encarga de hacer pedidos de direcciones y de mantenerlos en memoria un cierto tiempo, mientras que la segunda resuelve los pedidos, registra y des-registra nuevos nodos.

Cliente *Rendez-Vous*

Si el cliente de *Rendez-Vous* solamente realizara la resolución de direcciones, entonces las aplicaciones se tendrían que encargar de mandar el paquete una vez recibida la resolución. El principal inconveniente es que un pedido de resolución puede tomar un tiempo relativamente largo, por lo que muy probablemente la aplicación no quiera quedarse bloqueada esperando que llegue una respuesta para poder continuar,

aunque por supuesto que hasta que no reciba respuesta no puede enviar el paquete. Además, debería lidiar con una caché de direcciones para no volver a efectuar la consulta cada vez que sea necesario.

Por ello, es conveniente que el cliente *Rendez-Vous* se encarge de todas estas gestiones, dándole a las otras aplicaciones un medio más sencillo para que puedan enviar paquetes. Entonces, la interfaz que provee esta aplicación es la función $send(U, M)$, siendo U la dirección universal a la que se manda el paquete, y M el paquete a enviar. Esta función se muestra en el algoritmo 9.

Algoritmo 9 send(U, M)

```
1: buscar el par (U, V) en la caché
2: si se encontró (U, V) entonces
3:    enviar M a la dirección V
4: si no
5:    E = H(U)
6:    enviar un paquete Address Lookup a E pasándole la dirección U
7:    agregar (U, M) en el vector WaitQueue
8: fin si
```

Si la dirección universal se encuentra en la caché, se envía el paquete a la dirección de red correspondiente. Si no se encuentra, se manda el pedido de resolución y se agregan la dirección universal y el mensaje a un vector ($WaitQueue$) para que cuando se resuelva esta dirección, se envíe el paquete.

Cuando la aplicación recive un paquete, eventualmente se resolvió una dirección, por lo que debe agregar este valor a la caché y revisar el vector $WaitQueue$ para enviar los paquetes cuya dirección universal se acaba de resolver. Esto se muestra en el algoritmo 10.

Algoritmo 10 receive(M)

```
1: si M es un paquete de tipo Address Solve entonces
2:    si el flag solved de M vale verdadero entonces
3:       agregar en la caché (U, V)
4:       por cada par (Um, M) en WaitQueue hacer
5:          si Um == U entonces
6:             enviar M a la dirección V
7:          fin si
8:       fin por
9:    si no
10:      notificar mediante un mensaje interno que U no se puede resolver.
11:   fin si
12: fin si
```

Las aplicaciones de hipercubo deberán localilzar a la aplicación *Client Rendez-Vous* y utilizar la función $send(U, M)$ para enviar los paquetes, delegando en ella toda la responsabilidad de resolución de direcciones.

Servidor *Rendez-Vous*

El servidor de *Rendez-Vous* debe cumplir con las siguientes funciones:

- resolver pedidos de resolución
- tomar pedidos de registración y des-registración de otros nodos y reflejarlos en la tabla de resolución
- recibir tablas de resolución de otros nodos e incorporarlas
- registrar al nodo en el *Rendez-Vous* cuando éste se conecta
- des-registrar al nodo en el *Rendez-Vous* cuando se desconecta y delegar su tabla de resolución

- enviar un fragmento de la tabla de resolución cuando se cede una parte del espacio de direcciones

Los tres primeros items ocurren siempre cuando un mensaje llega a esta aplicación, es por ello que estas operaciones pueden ser efectuadas en la función *receive* de la aplicación, como se muestra en el algoritmo 11.

Algoritmo 11 receive(M)

```
1: si M es un paquete de tipo Register entonces
2:    agregar (U, V) en la tabla de resolución
3: si no, si M es un paquete de tipo Deregister entonces
4:    eliminar (U, V) de la tabla de resolución
5: si no, si M es un paquete de tipo Address Solve entonces
6:    buscar U en la tabla de resolución
7:    si se encontró U en la tabla de resolución entonces
8:       responder con un paquete Address Lookup con la dirección de red correspondiente
9:    si no
10:      responder con un paquete Address Lookup con solved = false
11:   fin si
12: si no, si M es un paquete de tipo Lookup Table entonces
13:   agregar la tabla recibida a la tabla de resolución.
14:   enviar un paquete Lookup Table Received al remitente, identificando las entradas recibidas.
15: si no, si M es un paquete de tipo Lookup Table Received entonces
16:   eliminar de la tabla de resolución las entradas indicadas.
17:   si el nodo está en proceso de desconexión entonces
18:      mandar un mensaje interno ReadyForDisc
19:   fin si
20: fin si
```

Este algoritmo realiza distintas operaciones según el tipo de paquete recibido. Ninguna de estas operaciones contiene bucles, saltos o esperas, por lo que el algoritmo debe finalizar.

Las tres últimas funciones que debe realizar el servidor *Rendez-Vous* ocurren cuando el nodo se conecta y desconecta de la red o cuando se cenden direcciones, por lo que pueden ser realizados cuando se reciben mensajes internos. El algoritmo 12 muestra estas operaciones.

Algoritmo 12 onMessageReceived(msg)

```
1: E = H(U)
2: si msg es de tipo Connected entonces
3:    enviar un paquete Register a E con (U, V)
4: si no, si msg es de tipo WillDisconnect entonces
5:    mandar un mensaje interno WaitMe
6:    enviar un paquete Lookup Table con la tabla de resolución al padre
7:    enviar un paquete Deregister a E con (U, V)
8: si no, si msg es de tipo AddressGiven entonces
9:    por cada elemento (Ur, Vr) de la tabla de resolución hacer
10:      si H(Ur) está en el espacio de direcciones cedido entonces
11:         agregar (Ur, Vr) en el vector SendTable
12:      fin si
13:   fin por
14:   enviar un paquete Lookup Table con todos los elementos de SendTable al nodo que se le cedió el
      espacio de direcciones
15: fin si
```

Este algoritmo realiza distintas operaciones según el tipo de mensaje recibido. Para el tipo *Address-Given* se realiza un bucle, pero dado que este recorre una tabla, finaliza en tantos ciclos como elementos

tenga la tabla. Las otras operaciones no tienen bucles, saltos o esperas, asegurando la finalización del algoritmo.

Ruteo *Rendez-Vous*

Para el ruteo de paquetes de *Rendez-Vous* se utilizan los algoritmos 7 y 8, pero para calcular la distancia se contepla la máscara del nodo. Por ejemplo, si el destino es 110111 y un nodo tiene dirección 010000/2, entonces la distancia utilizada en el ruteo de datos es 4, ya que hay 4 bits distintos. En cambio, en el ruteo de *Rendez-Vous*, para calcular la distancia se toman sólamente los dos primeros bits de la dirección (dado por la máscara), y por lo tanto la distancia es 1.

Esta variación en el cálculo de distancia se introdujo porque los destinos de *Rendez-Vous* no necesariamente existen, sino que pueden ser parte del espacio de direcciones de un nodo, entonces es conveniente comparar la distancia con el espacio de direcciones en vez de compararlo con la dirección en sí.

2.4.3. Formato de los paquetes de las aplicaciones de *Rendez-Vous*

Las aplicaciones de *Rendez-Vous* se comunican a nivel de capa de aplicación, es decir que sus paquetes van encapsulados dentro de segmentos de la capa de transporte, que podría ser tanto UDP como TCP. En el caso de que se usara UDP, se debería implementar un mecanismo para recuperar paquetes perdidos. Se definió un formato para estos paquetes consistente en un tipo de 5 bits y 3 bits de flags, seguido luego por datos cuyo formato depende del tipo de paquete. Esto se puede ver gráficamente en la figura 2.23.

Figura 2.23: Formato de un paquete de aplicación *Rendez-Vous*

Dentro de los datos, frecuentemente se envían direcciones universales y de red. Las direcciones universales se guardan anteponiendo un byte que indica su longitud en bytes, mientras que las de red almacenan su longitud en bits, aunque siempre ocupan un número entero de bytes.

Paquete Register

Este paquete lo manda un servidor *Rendez-Vous* a otro para indicarle que el nodo se conectó y darle su dirección universal y de red.

- type = 1
- primary address: dirección de red del nodo que se registra
- universal address: dirección universal del nodo que se registra

Paquete Deregister

Este paquete lo manda un servidor *Rendez-Vous* a otro para indicarle que el nodo se desconectó.

- type = 2
- primary address: dirección de red del nodo que se des-registra
- universal address: dirección universal del nodo que se des-registra

Address Solve

Este paquete lo manda un cliente *Rendez-Vous* a un servidor para hacer un pedido de resolución de dirección.

- type = 3
- universal address: dirección a resolver

Address Lookup

Este paquete lo manda un servidor *Rendez-Vous* a un cliente en respuesta a un paquete *Address Solve*, informando el resultado de la resolución de dirección.

- type = 4
- flag 0 (*solved*): indica si se pudo resolver o no la dirección
- primary address: si el flag 0 esta activo, se envía el valor de la dirección de red resuelta.
- universal address: dirección que se pidió resolver

Lookup Table

Este paquete se lo envía un servidor *Rendez-Vous* a otro para darle parte de sus tablas de resolución.

- type = 5
- id: identifica al fragmento de la tabla enviado
- n: cantidad de elementos en la tabla
- n veces:
  - primary address
  - universal address

Lookup Table received

Este paquete se lo envía un servidor *Rendez-Vous* a otro en respuesta a *Lookup Table* para indicarle que la tabla fue recibida correctamente.

- type = 6
- id: identificación de la tabla cuya recepción se está confirmando.

Capítulo 3

El Simulador

Una vez definido el protocolo, es necesario comprobar su correcto funcionamiento, y en caso de encontrar errores, poder corregirlos y volver a evaluarlo, depurándolo progresivamente.

Una posibilidad sería implementar el protocolo en un sistema operativo y hacer pruebas con él. Sin embargo, este enfoque presenta principalmente dos dificultades: para hacer pruebas en una red grande, se necesitarían muchas computadoras, lo cual implica altos costos y dificultad práctica de realización; la otra dificultad es monitorear lo que está ocurriendo a nivel de sistema operativo.

Es por ello que se recurre a implementar un simulador de redes, cuyo diseño se detalla en la primera parte de este capítulo.

En la segunda parte se utiliza el simulador para hacer pruebas del protocolo y se procesan los resultados obtenidos.

3.1. Arquitectura del Simulador

La realización del simulador consta de dos grandes etapas: diseño e implementación.

Durante la etapa de diseño se analizan los requerimientos del simulador, definiendo su funcionamiento y entrando progresivamente en detalle, para obtener como resultado final los diagramas de clases, los algoritmos y las decisiones sobre tecnologías a utilizar. Estos resultados son los que se presentan en esta sección.

La etapa de implementación consiste en traducir el resultado de la primera etapa en un lenguaje de programación y comprobar su correcto funcionamiento. El resultado es un programa que cumple con los requerimientos planteados. El código fuente y su documentación no se incluyen en este informe debido a su gran extensión, pero se encuentran en el CD adjunto y disponibles para bajar por internet en: http://sourceforge.net/projects/quenas

3.1.1. Diseño del simulador

Primeramente se analizaron los requerimientos que debe cumplir el simulador. Estos son:

1. Diseño flexible.
2. Capacidad de ejecución no interactiva.
3. Simulación orientada a eventos.
4. Simulación “limpia” y detallada.
5. Acciones a simular.

El primer item implica que si bien el objetivo principal del simulador es poder efectuar pruebas sobre el protocolo definido, considerando la complejidad que representa este desarrollo, se busca que no esté estrictamente limitado a este objetivo, sino que pueda ser reutilizado con diferentes pruebas u otros protocolos. Para ello, se debe pensar en términos de una plataforma (o *Framework*) de simulación sobre

la cual se desarrolla el caso particular de este protocolo.

Con el segundo item se busca poder realizar un gran número de simulaciones en forma automática. Si la ejecución del simulador requiriese intervención por parte del usuario, aunque sea simplemente para pedir un nombre de archivo, se dificultaría la automatización. Es por ello que el simulador debe utilizar la linea de comandos para obtener la información necesaria para su ejecución. Se utiliza un intérprete para ejecutar un archivo con comandos del simulador, como se describe en la sección 3.2.8.

La simulación orientada a eventos, en el tercer item, indica que el simulador genera eventos y los agenda para su posterior ejecución. Por ejemplo, cuando se envían datos por una conexión, se agenda un evento para que el destinatario reciba esos datos un cierto tiempo después. Si bien este modelo no representa exactamente la realidad dado que los eventos se ejecutan instantáneamente en la simulación, resulta natural y apropiado para este tipo de simulaciones, pudiendo generar varios eventos discretos en el caso de que fuera necesario un evento no instantáneo.

En el cuarto item, se requiere que la simulación sea "limpia", es decir, que el simulador no haga cosas imposibles de hacer en una implementación real y que tendrían un impacto significativo en los resultados. Por ejemplo, cada nodo debe comunicarse con los otros nodos a través de su conexion utilizando los protocolos, no pudiendo "hacer trampa" y comunicarse directamente.

La simulación detallada indica que se debe seguir estrictamente el protocolo, sin hacer suposiciones acerca de su funcionamiento. Por ejemplo, cuando un nodo se conecta a la red, se deben enviar todos los mensajes del protocolo y ejecutar los algoritmos propuestos. No sería válido deducir la dirección que se le asignará a partir de las direcciones de sus vecinos, ya que esto estaría suponiendo que anda bien parte de lo que se desea probar.

El último requerimiento engloba las acciones que se deben poder efectuar en una simulación:

- **Crear nodos**: los nodos se crean durante el transcurso de la simulación, especificando su dirección universal.
- **Crear conexiones**: las conexiones entre nodos son punto a punto, pudiendo especificar el ancho de banda y retardo.
- **Conexión de nodos a la red**: en cualquier momento de la simulación se puede requerir que un nodo se conecte o desconecte lógicamente de la red.
- **Envío de paquetes**: los nodos pueden enviar paquetes a otros nodos y obtener información de la ruta utilizada.
- **Adquisición de datos**: en cualquier momento de la simulación se puede requerir información de la red, los nodos, conexiones, estadísticas, etc.
- **Control de salida de datos**: se puede controlar la información que el simulador presenta a la salida para obtener solamente los datos deseados.

Una simulación consiste entonces en ejecutar una serie de acciones de los tipos mencionados aquí arriba. La forma elegida para definir la simulación es mediante un archivo de texto, dónde se escriben comandos en un lenguaje sencillo definido para este propósito.

Contando con estos requerimientos, se eligieron tecnologías apropiadas para poder cumplirlos:

- **C++**: se eligió utilizar este lenguaje de programación por su flexibilidad y velocidad de ejecución. Otros lenguajes más modernos como Java o C# podrían acelerar el tiempo de desarrollo a cambio de reducir la velocidad de ejecución del programa o la memoria que utiliza. Para que el simulador pueda trabajar con grandes redes en tiempo razonable, se priorizó la eficiencia de ejecución.
- **Extreme Programming**: es una metodología de desarrollo de software [10], de la que se utilizaron varias prácticas, principalmente los *Unit Tests* y *Acceptance Tests*. Esto consiste en escribir código

de prueba (tests) que verifica el correcto funcionamiento del programa. Estos tests se deben ejecutar frecuentemente para garantizar que a medida que evoluciona el código no se introducen errores. La diferencia entre los *Unit Tests* y los *Acceptance Tests* es que los primeros son mucho más granulares, probando cada método de cada clase, mientras que los últimos lo prueban en las condiciones que será usado el programa; por ejemplo se ejecuta una simulación y se comprueba que los resultados sean los esperados. Todos los tests deben estar automatizados para poder ejecutarlos frecuentemente y evitar errores humanos de verificación.

- **doxygen**: se utiliza este programa [13] para generar la documentación del código fuente del simulador a partir de los comentarios en el mismo. De esta forma, poniendo los comentarios en el formato apropiado, se obtiene también una valiosa documentación para desarrolladores del simulador.
- **XML**: [14] los resultados de la simulación se escriben como un archivo XML, ya que este formato permite una estructura flexible, contrariamente a lo que ocurriría utilizando tablas o archivos separados por coma.
- **XSLT**: [16] el archivo XML de salida de la simulación muchas veces no provee la información en un formato práctico para realizar tablas, graficar, etc. Se utiliza XSLT para filtrar y procesar el archivo XML obteniendo una vista de los datos que se desean obtener. Si bien XSLT no se utiliza en el simulador mismo, es recomendable su utilización para analizar los resultados.

Módulos

El simulador se divide en varios módulos, cada uno de los cuales contiene clases con un fin determinado. Cada uno de los módulos interactúa con sus pares, como muestra la figura 3.1, indicando las principales interacciones con flechas.

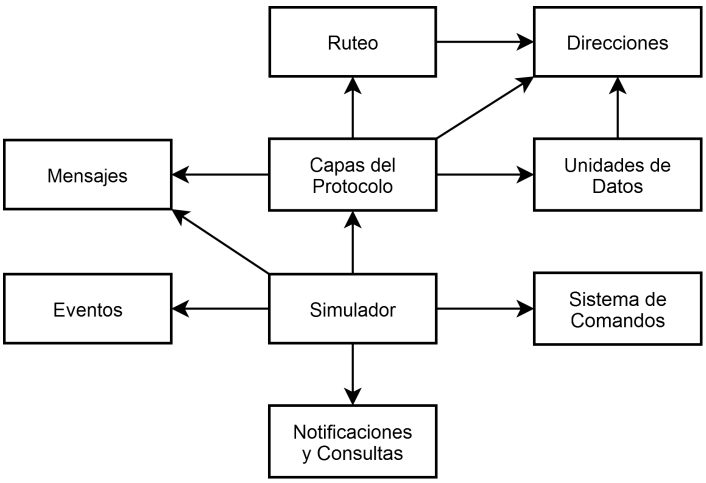


Figura 3.1: Esquema de Módulos del Simulador

Cada módulo coincide con uno o varios *namespaces* de C++. Las funcionen que cumplen los módulo son:

- **Simulador**: provee las bases del simulador, como ser el manejo de eventos, las clases que representan al tiempo, ancho de banda, redes y nodos. Es el núcleo del programa, ya que controla a todos los demás módulos.
- **Capas del Protocolo**: se definen las diferentes capas del protocolo que se implementan en un nodo: capa física, de enlace de datos, de red, de transporte y de aplicación. Además, se definen las clases necesarias para conectar las capas físicas de los distintos nodos.
- **Unidades de Datos**: cada capa del protocolo opera con una estructura de datos, que se encuentra definida en este módulo.

- **Direcciones**: define las clases que representan direcciones a distintos niveles: físico, de red y de aplicación.
- **Ruteo**: implementa el algoritmo de ruteo reactivo propuesto.
- **Mensajes**: las distintas capas del protocolo y el nodo se comunican utilizando mensajes asincrónicos, que son definidos en este módulo.
- **Eventos**: define los distintos tipos de eventos con los que trabaja el simulador.
- **Sistema de Comandos**: el usuario le dá ordenes al simulador utilizando un lenguaje de comandos, cuya interpretación y ejecución está a cargo de este módulo.
- **Notificaciones y Consultas**: el simulador produce una salida con información requerida por el usuario (consultas) o generada automáticamente cuando ocurren determinados sucesos (notificaciones), siendo este módulo el encargado de procesar, dar formato y presentar esta información.

El diseño del simulador se explica en este capítulo subdividido según estos módulos.

3.1.2. Notación

Se utiliza UML [3] para representar las clases que conforman al simulador y las relaciones entre ellas. Se explicarán brevemente los elementos de esta notación.

Para una mejor visualización del diagrama de clases de un módulo, éste se expone en varias partes, presentándose primero las clases y sus relaciones entre ellas sin los miembros que las componen, y luego en una o más figuras se detallan las clases.

La figura 3.2 muestra un ejemplo de un diagrama de relaciones entre las clases.

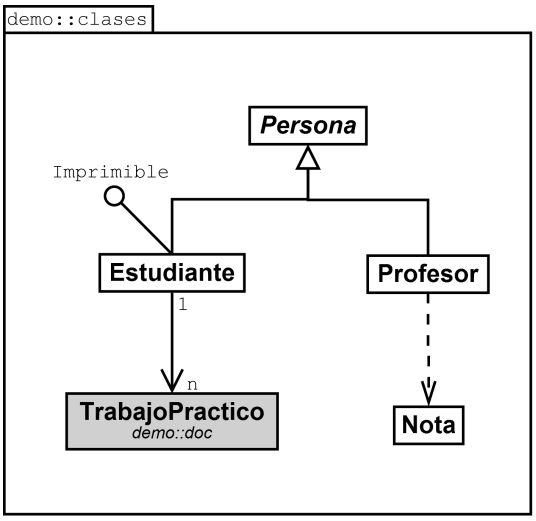


Figura 3.2: Ejemplo de diagrama de clases

El rectángulo grande representa un namespace, cuyo nombre (`demo::clases`) figura arriba a la izquierda. Cada uno de los rectángulos dentro del namespace representa una clase. Cuando el nombre está en itálica, como en `Persona`, significa que la clase es abstracta.

La flecha triangular indica herencia; en el ejemplo tanto `Estudiante` como `Profesor` heredan de `Persona`. La linea con un circulo en un extremo indica en UML que se está implementando una interface; aunque C++ no cuenta con interfaces, se utiliza esta notación cuando se cumple un rol similar a

una interface, a pesar de que se implemente como herencia múltiple. Por ejemplo, la clase `Estudiante` "implementa la interface" `Imprimible`.

La línea continua con flecha en punta indica una asociación; por ejemplo `Estudiante` y `TrabajoPractico` están asociados, es decir que la primera clase contiene instancias de la segunda. Opcionalmente se indica al costado la cardinalidad de la relación. El sombreado en la clase `TrabajoPractico` indica que pertenece a otro namespace, que se indica abajo de su nombre (`demo::doc`).

La linea punteada indica utilización, por ejemplo `Profesor` utiliza `Nota`, aunque no necesariamente guarda una instancia de ella.

Una vez presentado el diagrama global de las clases, se muestra el detalle de cada una, como en la figura 3.3. Esta clase de ejemplo busca mostrar todas las posibilidades y por ello su contenido no tiene demasiado sentido.

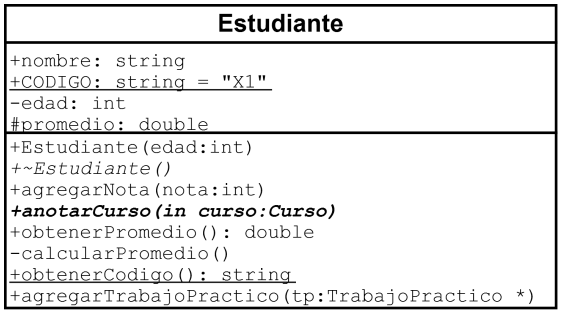


Figura 3.3: Ejemplo del detalle de una clase

En el rectángulo superior figura el nombre de la clase; en el del medio los atributos y en el inferior los métodos. El nivel de acceso se muestra utilizando "`+`" para público, "`#`" para protegido y "`-`" para privado. Cuando una variable o método es estático (es decir que se accede utilizando la clase y no una instancia), su nombre se subraya, como en el caso de `CODIGO` u `obtenerCodigo`.

Los atributos se declaran escribiendo primero su nombre, seguido de dos puntos y el tipo, y opcionalmente un valor inicial. Por ejemplo, `edad` es de tipo `int`, `CODIGO` es de tipo `string` e inicialmente vale `X1`. Se utilizan nombres en mayúsculas para indicar constantes.

En el recinto de los métodos, el o los constructores llevan el mismo nombre que la clase, y pueden llevar parámetros. El destructor es único, no puede llevar parámetros y a su nombre se le antepone el símbolo "`~`".
Cuando un método (o el destructor) es virtual, su nombre se escribe en itálica; en el ejemplo el destructor es virtual.
Los métodos en negrita itálica indican que el método es abstracto, como en el ejemplo `anotarCurso`.

Luego del nombre del método, puede seguir una lista de parámetros entre paréntesis, con igual notación que los atributos. Se utiliza `in` para indicar un parámetro constante pasado por referencia; por ejemplo "`in curso: Curso`" se implementa en C++ como "`const Curso &curso`".
En la declaración de parámetros o variables, el asterisco indica un puntero: "`tp : TrabajoPractico *`" declara a `tp` como un puntero a un `TrabajoPractico`.

3.2. Implementación

En esta sección se presentan las implementaciones de cada uno de los módulos indicados en la figura 3.1. Para cada módulo se presenta primero un diagrama donde se muestra la relación entre las clases que

lo componen, y luego se presenta el detalle de cada una de estas clases, explicando los atributos y los métodos de cada una.

3.2.1. Módulo Simulador

En este módulo se define la base para la plataforma de simulación. Las clases que lo componen se muestran en la figura 3.4.

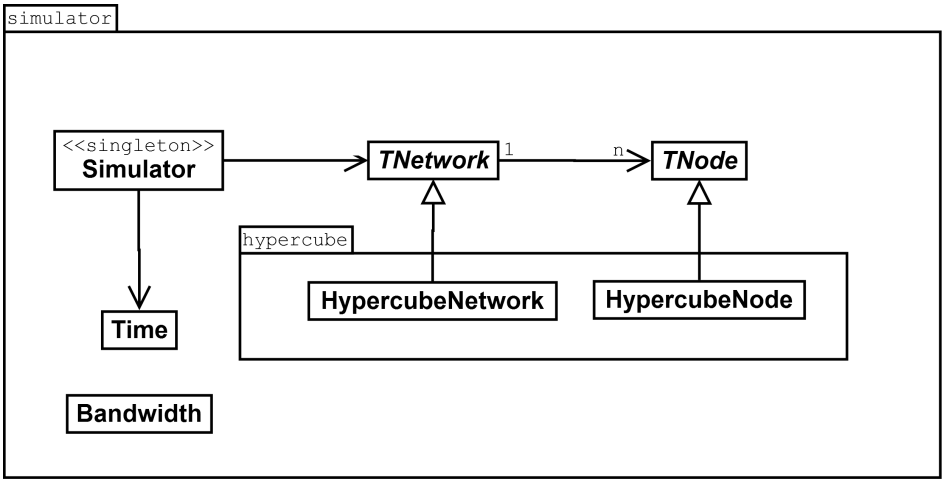


Figura 3.4: Diagrama de clases del módulo Simulador

La clase `Simulator` es el motor del simulador. Se define como *singleton*, es decir que sólo puede existir una instancia de esta clase, con el fin de que cualquier otra clase tenga acceso a sus métodos fácilmente, sin necesidad de tener un puntero al objeto.

Esta clase contiene una instancia de una red, cuya base es la clase abstracta `TNetwork`, y se provee una implementación concreta para redes de hipercubo mediante la clase `HypercubeNetwork`.
A su vez, una red está compuesta por nodos, siendo la base `TNode` y la clase para nodos de hipercubo `HypercubeNode`.

Las clases `Time` y `Bandwidth` se utilizan para representar al tiempo y ancho de banda convenientemente.

Los atributos de la clase `Simulator` (figura 3.5) son los siguientes:

- `time`: tiempo que se está simulando.
- `endTime`: tiempo en que finaliza la simulación.
- `eventQueue`: cola de prioridad dónde se guardan los eventos que se van a ejecutar, utilizando como clave de la cola el tiempo, para que cada vez que se extraiga un evento, se devuelva el próximo en orden cronológico.
- `notificator`: permite escribir las notificaciones en un archivo.
- `notifFilter`: permite filtrar las notificaciones que se van a escribir.
- `network`: red sobre la que se efectúa la simulación.
- `instance`: instancia única de la clase `Simulator`, utilizada para implementar el patrón de diseño *Singleton*.

A continuación se explican los métodos de esta clase, omitiendo los *getters* y *setters* triviales, cuya funcionalidad es consultar o darle valor a un atributo.

Figura 3.5: Clase `Simulator`

- `getInstance`: devuelve un puntero a la única instancia de esta clase. Si aún no estaba instanciada, se instancia previamente.
- `destroy`: elimina la instancia de esta clase.
- `getTime`: devuelve el tiempo que se está simulando.
- `simulateStep`: simula un paso de la simulación, es decir, ejecuta un evento.
- `simulate`: ejecuta una simulación hasta alcanzar el tiempo máximo.
- `loadFile`: carga e interpreta un archivo con comandos, agendando la ejecución de cada uno en el tiempo requerido.
- `addEvent`: agrega un evento en la cola de ejecución.
- `reset`: vuelve a 0 el tiempo de simulación.
- `notify`: las variantes de este método son llamadas para reportar una notificación. Primero, comprueban sin la notificación debe ser escrita, utilizando el filtro dado por el atributo `notifFilter`, y si así fuera, llaman al método *writeNotification*.
- `writeNotification`: escribe una notificación, utilizando para ello el objeto `notificator`.
- `exec`: ejecuta un comando sobre el objeto especificado (o sobre la red si no se especifica ninguno), y notifica el resultado de su ejecución.
- `runCommand`: ejecuta un comando sobre el objeto `Simulator`.

Las clases que representan a una red, `TNetwork` y `HypercubeNetwork`, se muestran con detalle en la figura 3.6 (ver también figura 3.4).

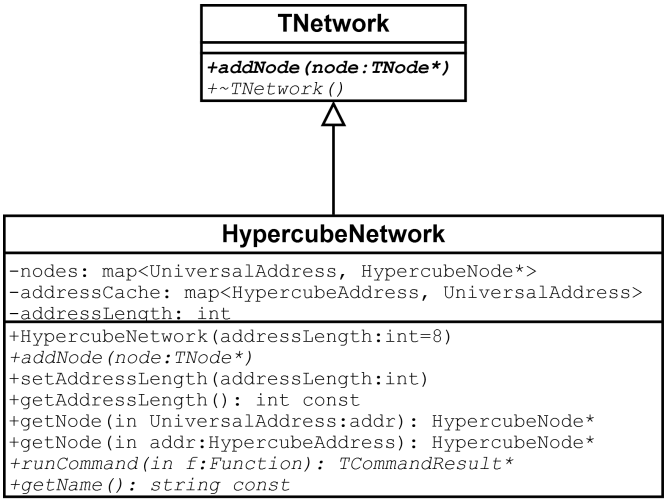


Figura 3.6: Clases `TNetwork` y `HypercubeNetwork`

`TNetwork` define la interface para la funcionalidad básica que debe tener una red, que es la capacidad de agregar nodos. Si bien esta clase no hace prácticamente nada, el simulador hace referencia a ella en vez de hacerlo directamente a la red de hipercubo, permitiendo de esta forma que si en el futuro se desea agregar otro tipo de red, se minimizen las modificaciones a realizar.

Es entonces `HypercubeNetwork` quien realiza el verdadero trabajo representando a una red de nodos de hipercubo. Esta clase guarda los nodos que se encuentran en la red en el mapa `nodes`, permitiendo además encontrar rápidamente un nodo dada su dirección universal. Para poder encontrar un nodo eficientemente sabiendo su dirección de hipercubo, se utiliza el mapa `addressCache`.
El atributo `addressLength` es la dimensión del hipercubo o longitud de la dirección, parámetro que impone el usuario para todos los nodos de la red.

En cuanto a los métodos de la clase, el constructor instancia un objeto utilizando como parámetro la longitud de dirección, u 8 si se omite. Este parámetro puede ser consultado o modificado posteriormente con los métodos `getAddressLength` y `setAddressLength`.

Los otros métodos de esta clase tienen la siguiente funcionalidad:

- `addNode`: agrega un nodo a la red.
- `getNode`: las dos versiones de este método sirven para encontrar un método, dada su dirección universal o de hipercubo.
- `runCommand`: ejecuta un comando en la red de hipercubo, como por ejemplo agregar un nodo o una conexión, cambiar la longitud de dirección, etc.
- `getName`: devuelve el nombre del objeto ("Network").

Las redes contienen nodos; siendo `TNode` la clase para representar a un nodo general, y `HypercubeNode` a un nodo de hipercubo. Estas clases se muestran en el diagrama 3.7.

La clase `TNode` provee funcionalidad para el manejo de mensajes asincrónicos.
Esto permite que un objeto se registre para obtener un tipo determinado de mensaje mediante el método `registerMessageListener`, pudiendo posteriormente des-registrarse con el método `unregisterMessageListener`. Estos métodos tan sólo reflejan sus acciones en el mapa `messageReceivers`,

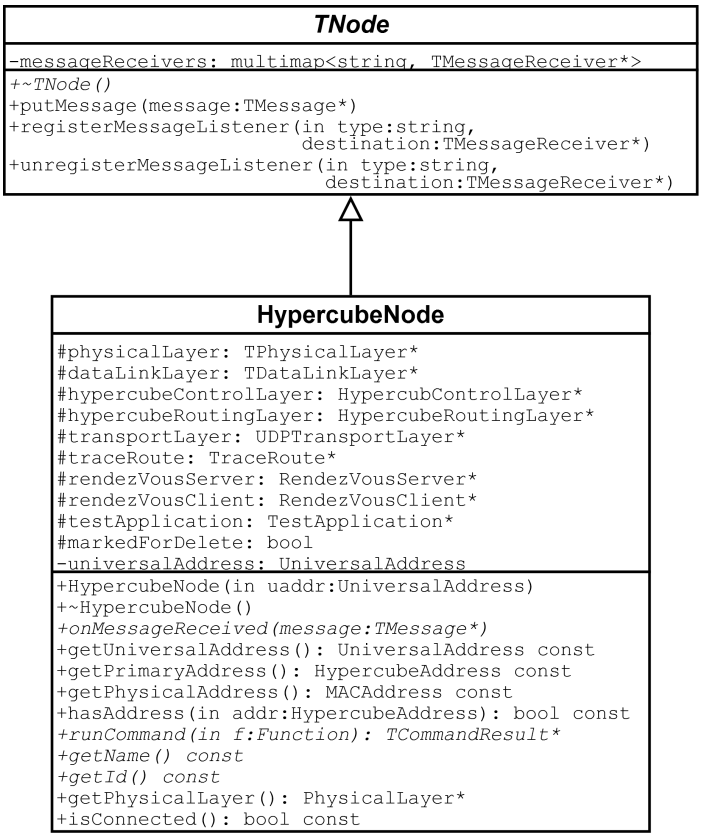


Figura 3.7: Clases `TNode` y `HypercubeNode`

indicando para cada tipo de mensaje que instancias están registradas.

Cuando en alguna parte del código se desea enviar un mensaje, se utiliza el método `putMessage`, quien busca cuales son los objetos registrados para escuchar ese mensaje y llama al método `onMessageReceived` en ellos.

La clase `HypercubeNode` hereda de `TNode` y la extiende dándole la funcionalidad requerida por un nodo de hipercubo, manejando todas las capas del protocolo.

Los atributos de esta clase son:

- `physicalLayer`: capa física.
- `dataLinkLayer`: capa de enlace de datos.
- `hypercubeControlLayer`: unidad de control que maneja la conexión y de direcciones en la capa de red.
- `hypercubeRoutingLayer`: unidad de ruteo en la capa de red.
- `transportLayer`: capa de transporte similar a UDP.
- `traceRoute`: aplicación de trazado de rutas.
- `rendezVousServer`: aplicación de *Rendez-Vous*, encargada de registrar direcciones y responder a pedidos de resolución.
- `rendezVousClient`: aplicación de *Rendez-Vous*, encargada de pedir resolución de direcciones y mantener una *caché* apropiada.

- `testApplication`: aplicación de pruebas, permite enviar paquetes de un nodo a otro.
- `markedForDelete`: indica que el nodo está marcado para ser eliminado de la red, acción que se ejecutará luego de que el nodo se haya desconectado.
- `universalAddress`: dirección universal del nodo.

El constructor de la clase toma como parámetro la dirección universal del nodo y crea todas las capas del protocolo.

Los métodos que provee esta clase son:

- `onMessageReceived`: es llamado cuando se recibe un mensaje para su procesamiento.
- `getUnivesalAddress`: devuelve la dirección universal del nodo.
- `getPrimaryAddress`: devuelve la dirección primaria de hipercubo del nodo.
- `getPhysicalAddress`: devuelve la dirección física del nodo.
- `hasAddress`: devuelve verdadero si la dirección pasada como parámetro es la dirección primaria o una de las direcciones secundarias del nodo.
- `runCommand`: ejecuta un comando en el nodo, como por ejemplo conectarse o desconectarse a la red, hacer una consulta, etc.
- `getName`: devuelve el nombre del objeto ("Node").
- `getId`: devuelve la dirección universal para ser usada como identificador del nodo.
- `getPhysicalLayer`: devuelve el objeto que representa la capa de red.
- `isConnected`: devuelve verdadero si el nodo se encuentra conectado a la red de hipercubo.

Este módulo cuenta también con las clases `Time` y `Bandwidth`, que se muestran en las figuras 3.8 y 3.9 respectivamente.

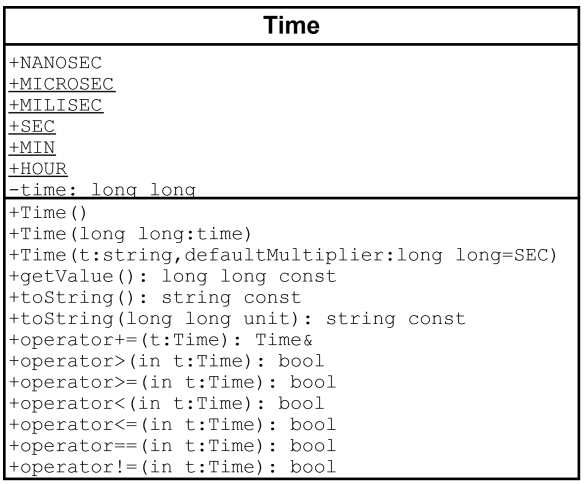


Figura 3.8: Clase `Time`

La clase `Time` se utiliza para representar el tiempo convenientemente, almacenándolo en unidades enteras de nanosegundos. También provee constantes con múltiplos de esta unidad, llegando hasta una hora.

Una de las funciones de esta clase es facilitar la conversión del tiempo desde y hacia cadenas de caracteres que incluyan las unidades. El constructor que toma una cadena como entrada se encarga de interpretarla; por ejemplo se podría ingresar “20 ms”.

Por otro lado, las funciones `toString` convierten el valor de tiempo almacenado en la instancia en una cadena de caracteres, pudiendo especificar la unidad que se desea utilizar, o dejando que el método escoja la más conveniente.

Por último, esta clase sobrecarga funciones de comparación y suma para facilitar el manejo de los tiempos.

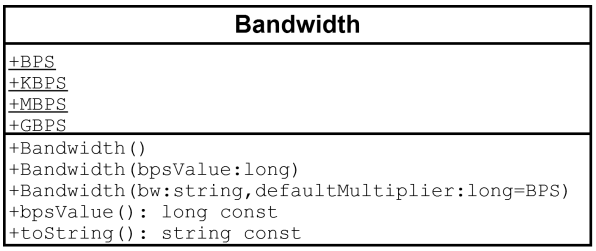


Figura 3.9: Clase `Bandwidth`

La clase `Bandwidth` es similar a `Time`, pero esta vez para ancho de banda, permitiendo utilizar las unidades `bps`, `Kbps`, `Mbps` y `Gbps`. A diferencia de la clase `Time`, esta no cuenta con sobrecarga de funciones de comparación y suma, ya que esto no es necesario.

3.2.2. Capas del Protocolo

Se implementaron en el simulador todas las capas del modelo híbrido propuesto en [7]. En la figura 3.10 se muestra un diagrama con las clases que forman parte de estas capas, a excepción de las clases que representan las máquinas de estados de la capa de red, que serán presentadas más adelante.

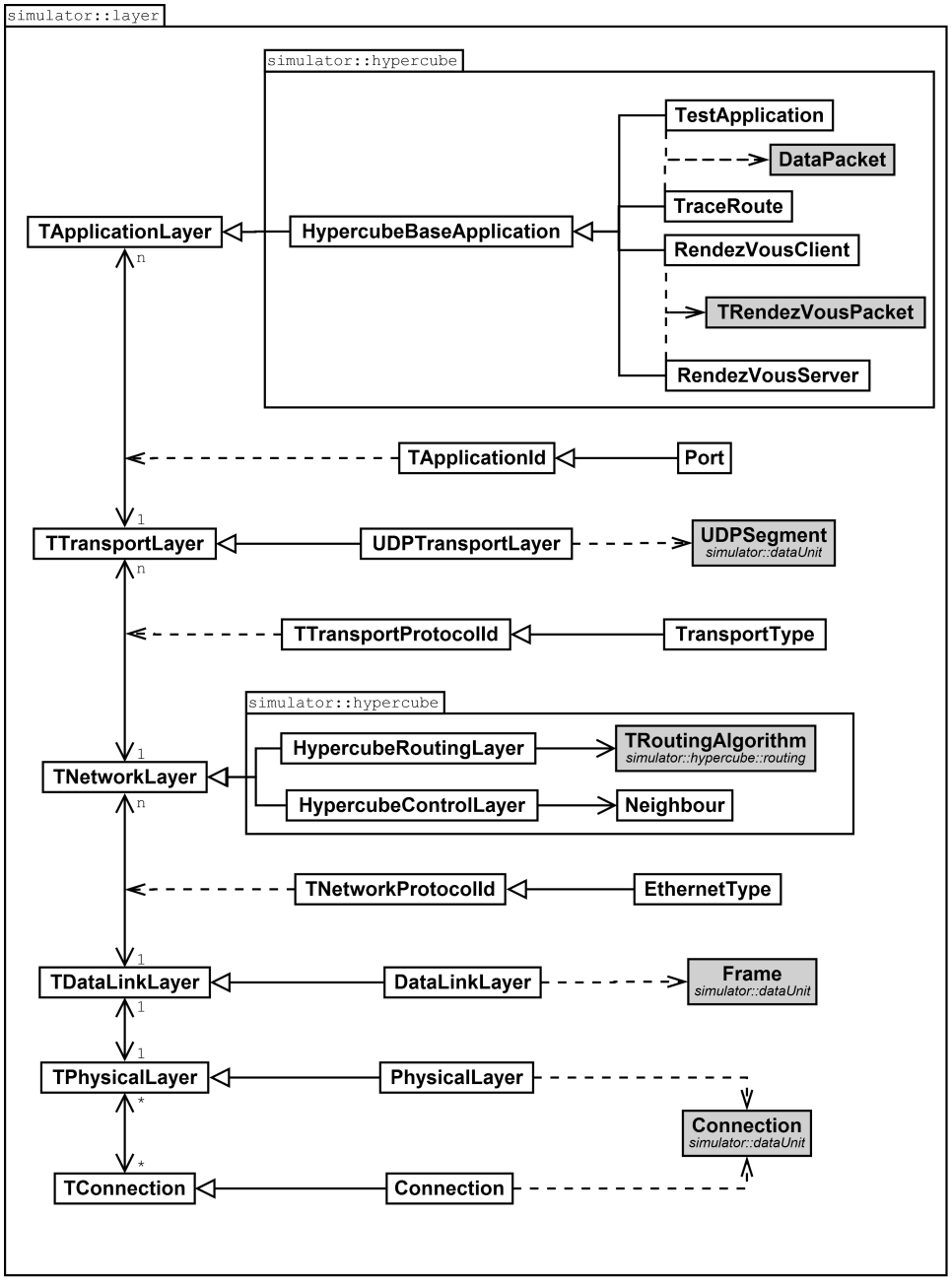


Figura 3.10: Diagrama de clases de las Capas de Protocolo

Cada una de las capas tiene una clase base abstracta, donde se define la interface que debe presentar. De cada una de estas clases, heredan una o más implementaciones. Además, cuando la relación entre una capa y su adyacente es de uno a muchos, se utiliza una identificación del protocolo. Por ejemplo, la relación entre capa de red y capa de transporte (`TNetworkLayer` y `TTransportLayer`) es de uno a muchos, por lo que la primera capa necesita un identificador para saber a que capa de transporte enviar los datos. En este caso, la clase `TTransportProtocolId` presenta la interface de estos identificadores,

mientras que `ProtocolType` es su implementación, que se utiliza en la capa de red.

Conexiones

Si bien las conexiones no son realmente una capa, por comodidad se incluyeron en esta sección. La figura 3.11 muestra en detalle las dos clases utilizadas para conexiones.

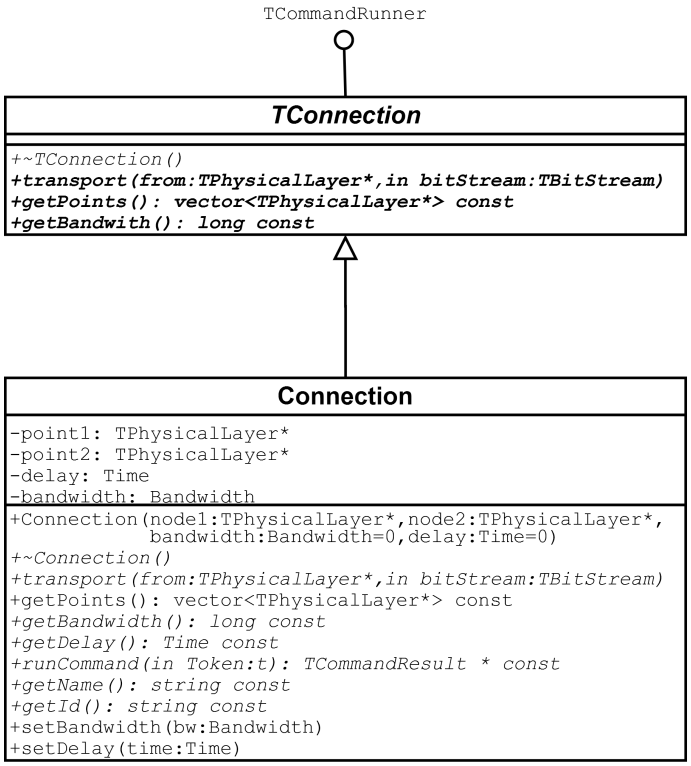


Figura 3.11: Clases de conexión

La clase abstracta `TConnection` es la clase base para las conexiones, que asume que una conexión consiste en mútiples puntos de tipo `TPhysicalLayer`, y que transporta *bit streams* desde alguno de los puntos a todos los demás, utilizando un cierto ancho de banda, que podría ser infinito.

Los tres métodos provistos por esta clase son abstractos, y las clases que hereden los deben implementar con la siguiente funcionalidad:

- `transport`: se llamará a este método para transportar a un *bit stream* desde un nodo hacia el resto, por lo que este nodo se debe encargar de hacer que las capas físicas de los otros nodos lo reciban.
- `getPoints`: debe devolver un vector con punteros a todas las capas físicas que se encuentran conectadas.
- `getBandwidth`: debe devolver el ancho de banda de la conexión, o cero para indicar infinito.

La clase `Connection` hereda de `TConnection`, implementando una conexión punto a punto, que adicionalmente provee un retardo en la transmisión.

Los dos puntos de esta conexión se almacenan en los atributos `point1` y `point2`, mientras que el retardo y ancho de banda se almacenan en `delay` y `bandwidth` respectivamente.

El constructor de la clase requiere ambos puntos de conexión, y optativamente se le puede especificar un retardo y un ancho de banda, utilizando cero e infinito respectivamente en el caso de que se omitan.

Los demás métodos cumplen con la siguiente funcionalidad:

- `transport`: envía el *bit stream* al otro punto de la conexión, creando un evento de tipo *ReceiveBitStreamEvent* que llamará al método `receive` en la capa física del nodo una vez transcurrido el tiempo de retardo. El ancho de banda no es utilizado aquí, sino en la cola de salida del nodo.
- `getPoints`: devuelve un vector con los dos puntos de la conexión.
- `getBandwidth`: devuelve el ancho de banda de la conexión.
- `getDelay`: devuelve el retardo de la conexión.
- `runCommand`: ejecuta un comando en la conexión, que puede ser una consulta (*query*) o darle un valor al ancho de banda o al retardo.
- `getName`: devuelve el nombre del objeto ("Connection").
- `getId`: devuelve una cadena de caracteres conteniendo los identificadores de ambos puntos de conexión, para generar un identificador de la conexión.
- `setBandwidth`: da un valor al ancho de banda de la conexión.
- `setDelay`: dá un valor al retardo de la conexión.

Capa física

Para representar la capa física se utilizan las clases `TPhysicalLayer` y `PhysicalLayer`, como muestra la figura 3.12.

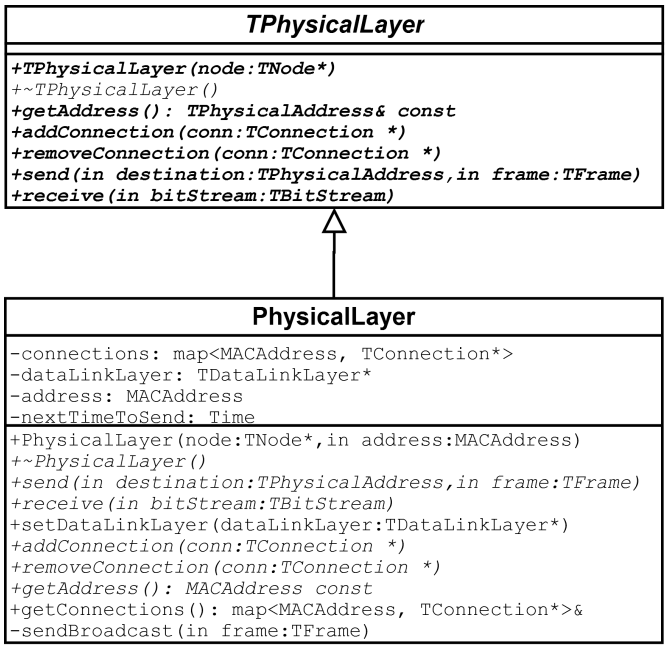


Figura 3.12: Clases de la capa física

La clase `TPhysicalLayer` declara métodos para agregar y borrar conexiones (`addConnection` y `removeConnection`), ya que la capa física debe tener registro de las conexiones establecidas. Estos métodos deben ser llamados por la conexión cuando es creada.

Además, declara los métodos para envío y recepción de datos (`send` y `receive`), y un método para obtener la dirección física de esta capa (`getAddress`).

La clase `PhysicalLayer` es una implementación sencilla de capa física que utiliza direcciones MAC. Esta dirección se guarda en el atributo `address`.
La capa física se conecta a otras capas físicas mediante conexiones, en particular, objetos `TConnection`. Todas estas conexiones se almacenan en el mapa `connections`, cuya clave es la dirección MAC del otro punto, permitiendo un rápido acceso cuando se conoce este valor.

El atributo `dataLinkLayer` es un puntero a la capa de enlace de datos en el mismo nodo, necesario para comunicarse con ella cuando se reciben datos.

Por último, el atributo `nextTimeToSend` representa el tiempo en el que se mandará el próximo paquete, para poder simular una cola de salida debido al ancho de banda. Este valor es utilizado por todas las conexiones; de esta forma, si bien las conexiones son punto a punto, se comporta como una sola conexión de medio compartido.

El constructor de la clase inicializa la referencia al nodo donde se encuentra la capa física, y da valor a su dirección.

Los métodos cumplen con las siguientes funcionalidades:

- `send`: envía un *frame* a la dirección determinada, pudiendo usarse la constante `MACAddress:BROADCAST` para enviarlo a todos los vecinos, siendo delegada la tarea en este caso al método privado `sendBroadcast`. Para efectuar el envío punto a punto, primero se calcula el retardo debido al ancho de banda y se adiciona el valor al atributo `nextTimeToSend`. Utilizando este tiempo, se agenda un evento del tipo *SendBitStreamEvent*, que se encargará de llamar al método `transport` en la conexión apropiada.
- `receive`: este método es llamado cuando se recive un *frame* desde otra capa física, y tan sólo se lo pasa a la capa de enlace de datos sin ningún procesamiento.
- `setDataLinkLayer`: da valor al atributo *dataLinkLayer*. Este método debe ser llamado antes de comenzar a utilizar el objeto.
- `addConnection`: agrega una conexión a otro nodo.
- `removeConnection`: borra una conexión a otro nodo.
- `getAddress`: devuelve la dirección MAC del nodo.
- `getConnections`: devuelve un vector con todas las conexiones.
- `sendBroadcast`: método privado para enviar un *frame* a todos los vecinos. Para ello, se itera a través de todas las conexiones y se envía a través de cada una.

Capa de enlace de datos

La capa de enlace de datos se implementa en las clases `TDataLinkLayer` y `DataLinkLayer`, como muestra la figura 3.13.

En la clase base, se define el atributo `physicalLayer` como un puntero a la capa física.
Los métodos `send` y `receive` serán llamados por las capas superiores (de red) e inferiores (física) respectivamente. Dado que una capa de enlace de datos podría estar asociada con más de una capa de red, se provee el método *registerNetworkProtocol*, que permite registrar las distintas capas de red identificándolas por un tipo que será utilizado posteriormente como discriminador.

La clase `DataLinkLayer` provee una implementación de esta capa, utilizando `frames` similares a los definidos en Ethernet II, direcciones MAC y multiplexando la capa de red mediante el identificador `EthernetType`. Los protocolos registrados se almacenan en el mapa `networkProtocol`.

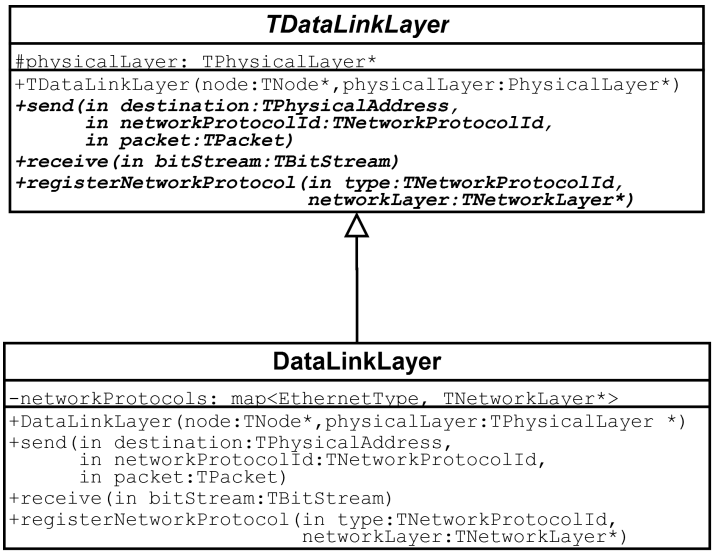


Figura 3.13: Clases `TDataLinkLayer` y `DataLinkLayer`

El constructor requiere un puntero al nodo donde se encuentra la capa de enlace de datos, así como un puntero a la capa física relacionada.

Los demás métodos cumplen con la siguiente funcionalidad:

- `send`: genera un *frame* a partir del paquete pasado y llama al método `send` en la capa física para que lo envíe.
- `receive`: es llamado por la capa física al recibir un paquete. Primero crea un paquete a partir del *frame* recibido, y luego a partir del *Ethernet Type* escoge la capa de red adecuada para llamar a su método `receive`.
- `registerNetworkProtocol`: registra una capa de red, agregándola en el mapa `networkProtocols`. Si el puntero fuera nulo, se elimina la capa de red de la registración.

Capa de Red

En la capa de red se definen dos protocolos que forman parte de `ANTop`: el control de direcciones y el ruteo. Estos se registran en la capa de enlace de datos, que al recibir *frames* se encarga de dirigírselo a quien corresponda según el EthernetType.

La clase base para ambos protocolos de red es `TNetworkLayer`, que se muestra en la figura 3.14.

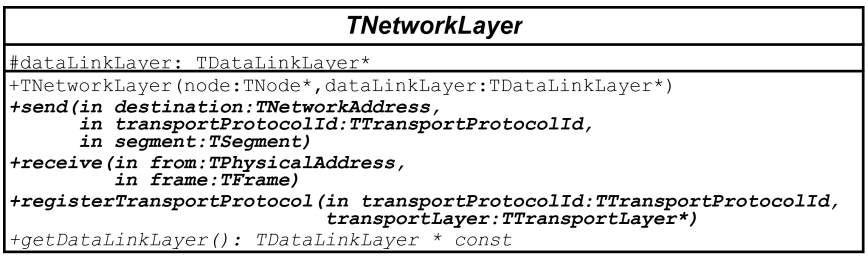


Figura 3.14: Clase `TNetworkLayer`

El atributo `dataLinkLayer` es un puntero a la capa de enlace de datos, necesario para enviar datos a través de ella. El método `getDataLinkLayer` devuelve un puntero a esta capa.

El constructor toma como parámetros el nodo donde se encuentra la capa de red, y un puntero a la capa de enlace de datos correspondiente, y guarda estos valores.

Los tres métodos restantes son abstractos, y no necesariamente tienen sentido para cualquier implementación, en cuyo caso estas deberían tirar una excepción si se llamase al método. En caso de que se implementen los métodos, deben cumplir la siguiente funcionalidad:

- `send`: este método es llamado por la capa de transporte para enviar un segmento a la dirección especificada.
- `receive`: es llamado al recibir un paquete desde la capa de enlace de datos.
- `registerTransportProtocol`: registra un protocolo de la capa de transporte, para que al recibir un paquete se pueda decidir quien se encarga de procesarlo.

Una de las clases que heredan de `TNetworkLayer` es `HypercubeRoutingLayer` (ver figura 3.15), con el propósito de relacionar a la capa de red con el algoritmo de ruteo. Es decir, el ruteo no está implementado en esta clase, lo cual permite que se pueda cambiar por otro algoritmo de ruteo (por ejemplo proactivo) con facilidad.

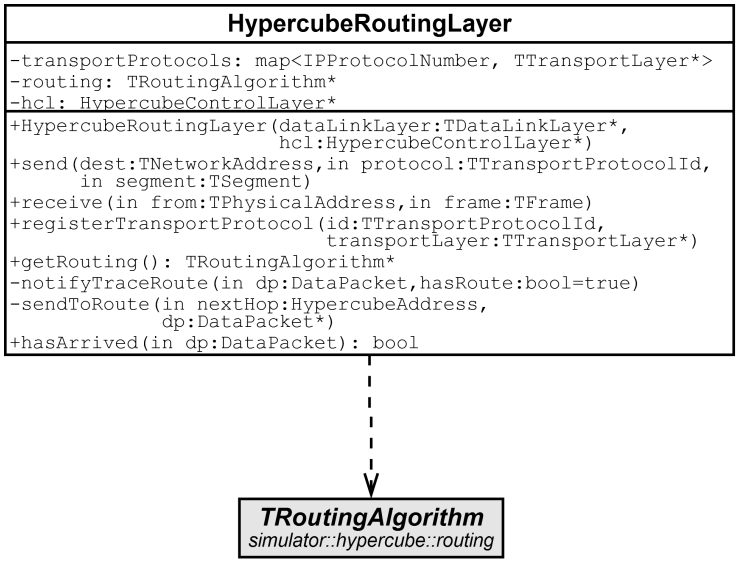


Figura 3.15: Clase `HypercubeRoutingLayer`

La clase que implementa el ruteo debe heredar de `TRoutingAlgorithm`; por ello el atributo `routing` tiene un puntero a esa clase, siendo este objeto el que realiza el verdadero trabajo de ruteo.

El atributo `hcl` es un puntero a la parte de control de direcciones dentro de la misma capa, y `transportProtocols` es un mapa que contiene las capas de transporte asociadas a sus identificadores de protocolo, lo que permitiría por ejemplo utilizar UDP y TCP dentro de un mismo nodo.

El constructor de la clase toma un puntero a la capa de enlace de datos y otro a la parte de control de direcciones dentro de la capa de red, y tan sólo guarda estos valores.

La funcionalidad de los métodos es la siguiente:

- `send`: este método utiliza al objeto `routing` para determinar cuál es el vecino al que se le debe enviar el paquete, ya que la dirección que recibe este método es una dirección de red y no necesariamente corresponde a un nodo adyacente. Una vez determinado el vecino más conveniente, el paquete es enviado a través de la capa de enlace de datos, utilizando su método `send`.

- `receive`: verifica si el paquete llegó al destino final mediante el método `hasArrived`. Si llegó, es enviado a la capa de transporte utilizando su método `receive`. Si no llegó, utiliza al objeto `routing` para determinar el próximo destino y lo reenvía.
- `registerTransportProtocol`: registra una capa de transporte, para que al recibir un paquete se pueda decidir quien se encarga de procesarlo. Para eso, el puntero se agrega en el mapa `transportProtocols`. Si el puntero fuera nulo, se elimina la capa de transporte de la registración.
- `getRouting`: devuelve un puntero al algoritmo de ruteo.
- `notifyTraceRoute`: produce una salida notificando que se completó el trazado de una ruta.
- `sendToRoute`: método auxiliar para enviar un paquete a un vecino determinado.
- `hasArrived`: devuelve verdadero si el paquete llegó a destino final; si el paquete no es de *Rendez-Vous*, la dirección de destino debe ser igual a la dirección primaria, y si lo es, también podría ser igual a una dirección secundaria o estar dentro del espacio de direcciones de alguna de ellas.

La implementación del algoritmo de ruteo se encuentra en la sección 3.2.5. Para saber si el paquette es de datos o de *Rendez-Vous*, se utiliza uno de los *flags* del paquete, *isRendezVous*, descripto en la sección 2.3.3.

En la capa de red se encuentra también el manejo de conexiones, cuya clase principal es `HypercubeControlLayer`, como muestra la figura 3.16.

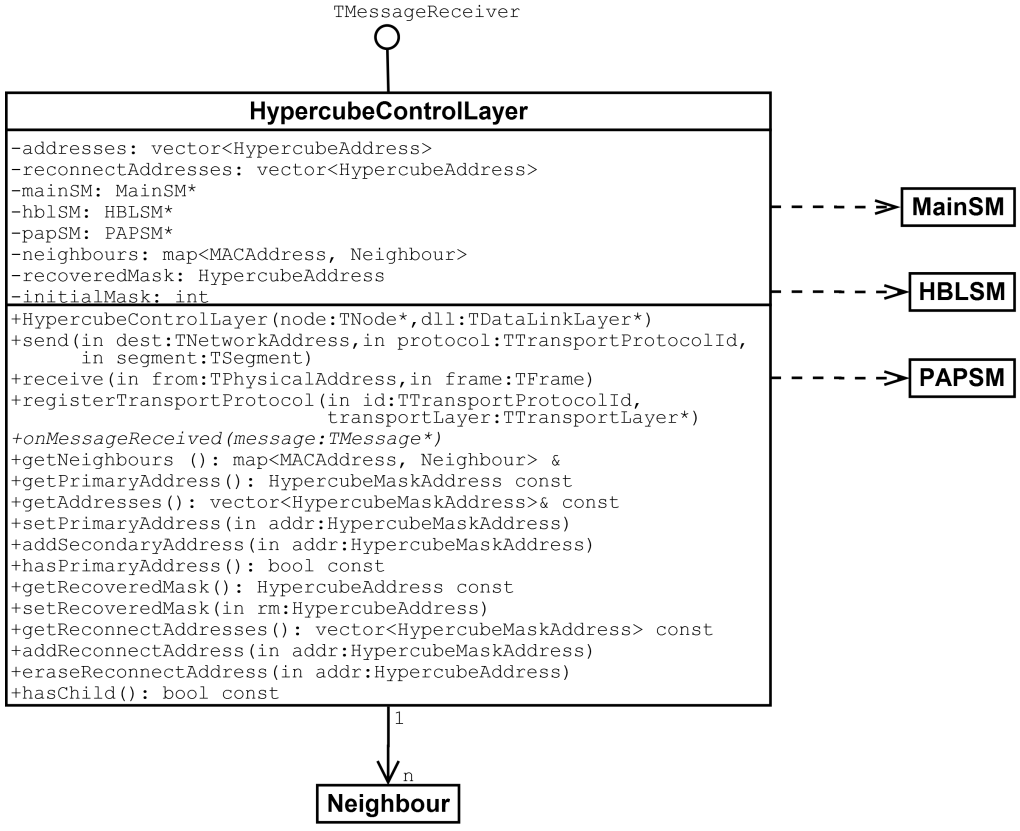


Figura 3.16: Clase `HypercubeControlLayer`

La clase implementa la conexión y desconexión de nodos utilizando 3 máquinas de estado, cuyos diagramas se presentaron en la sección 2.2.3. Primero se verá el funcionamiento de esta clase para luego

entrar en el detalle de la implementación de las máquinas de estados.

Los atributos de esta clase son:

- `addresses`: este vector contiene en el primer elemento la dirección primaria, y si hubiera más elementos, estos son direcciones secundarias.
- `reconnectAddresses`: vector con direcciones que pueden ser utilizadas para reconexión, es decir, direcciones que fueron cedidas y luego recuperadas porque el nodo se desconectó.
- `mainSM`: *Main State Machine*. Es la máquina de estados a cargo de conectarse y desconectarse de la red. (ver figura 2.11)
- `hblSM`: *Heard Bit Listener State Machine*. Esta máquina de estados escucha paquetes *Heard Bit* y dá de baja los vecinos que no lo envían durante un cierto período. Además, puede proponer direcciones secundarias como respuesta a estos paquetes. (ver figura 2.12)
- `papSM`: *Primary Address Provider State Machine*. Esta máquina de estados espera paquetes de pedido de direcciones primarias y responde a ellos. (ver figura 2.13)
- `neighbours`: contiene un mapa con todos los vecinos del nodo.
- `recoveredMask`: indica los bits del espacio de direcciones que fueron recuperados al desconectarse los vecinos.
- `initialMask`: la máscara de la dirección primaria que obtuvo el nodo al ser conectado a la red.

El constructor de la clase inicializa la máquina de estados *Main State Machine*. Las otras dos máquinas de estados permanecen inactivas hasta que se conecta a la red.
Los métodos de la clase cumplen las siguientes funciones:

- `send`: este método no debe ser llamado, dado que no tiene una capa superior. Por ello, si es llamado, tira una excepción.
- `receive`: es llamado cuando se recibe un paquete. El paquete es pasado a las máquinas de estado, que se encargarán de efectuar las acciones adecuadas.
- `registerTransportProtocol`: al igual que `send`, lanza una excepción.
- `onMessageReceived`: es llamado por el simulador cuando se recibe un mensaje de pedido de conexión o desconexión a la red por parte del usuario. Este mensaje es transmitido a las máquinas de estado.
- `getNeighbours`: devuelve un mapa con los vecinos del nodo.
- `getPrimaryAddress`: devuelve la dirección primaria del nodo.
- `getAddresses`: devuelve un vector con la dirección primaria y las direcciones secundarias si las hubiera.
- `setPrimaryAddress`: le asigna un valor a la dirección primaria del nodo.
- `addSecondaryAddress`: agrega una dirección secundaria.
- `hasPrimaryAddress`: devuelve verdadero si hay una dirección primaria asignada.
- `getRecoveredMask`: devuelve la máscara de direcciones recuperadas.
- `setRecoveredMask`: da valor a la máscara de direcciones recuperadas.
- `getReconnectAddresses`: devuelve un vector con direcciones que pueden ser utilizadas para reconexión.
- `addReconnectAddress`: agrega una dirección para reconexión.

- `eraseReconnectAddress`: borra una dirección para reconexión.
- `hasChild`: devuelve verdadero si el nodo cedió parte de su espacio de direcciones.

Los vecinos del nodo se representan mediante la clase `Neighbour`, como muestra la figura 3.17.

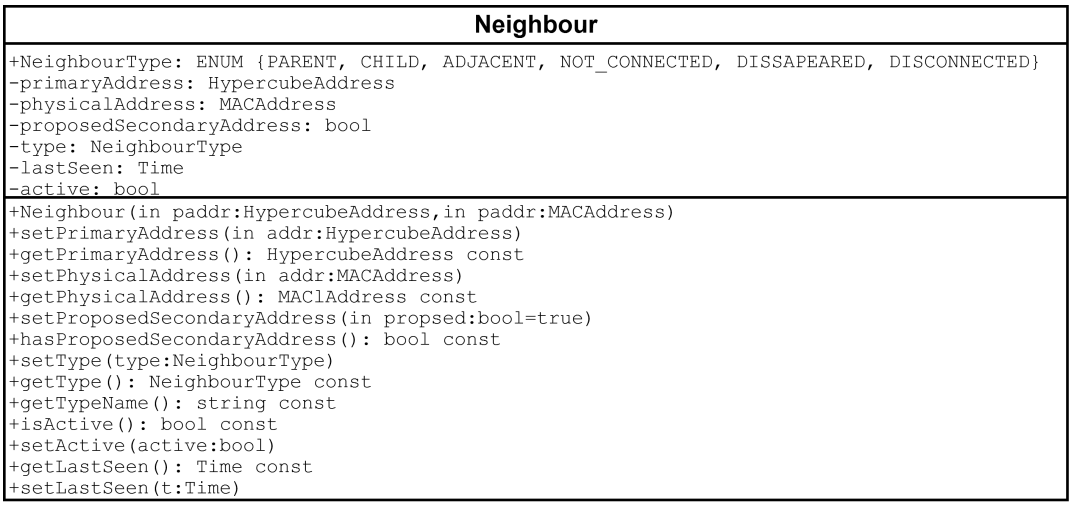


Figura 3.17: Clase `Neighbour`

La enumeración `NeighbourType` indica la relación que tiene el nodo con su vecino, pudiendo ser:

- `PARENT`: el vecino es el padre del nodo, es decir que es el que le dió la dirección primaria.
- `CHILD`: el vecino es un hijo del nodo, por lo que el vecino recibió la dirección primaria del nodo.
- `ADYACENT`: los nodos están conectados lógicamente, pero no tienen una relación de padre-hijo. Esto ocurre cuando los nodos están conectados físicamente y sus direcciones difieren en un bit, incluyendo el caso en que uno de los nodos utiliza una dirección secundaria para efecutar esta conexión.
- `NOT_CONNECTED`: hay una conexión física entre los nodos pero no se pudieron conectar lógicamente porque las direcciones no lo permiten.
- `DISSAPEARED`: el vecino estaba conectado lógicamente pero dejó de mandar *Heard Bits*, por lo que se supone que se desconectó repentinamente.
- `DISCONNECTED`: el vecino estaba conectado lógicamente y se desconectó avisando previamente.

Los otros atributos de la clase son:

- `primaryAddress`: dirección primaria del vecino.
- `physicalAddress`: dirección física de vecino.
- `proposedSecondaryAddress`: verdadero si ya se le propuso una dirección secundaria.
- `type`: tipo de vecino, según la clasificación explicada anteriormente.
- `lastSeen`: instante de tiempo en el que el vecino mandó por última vez un paquete *Heard Bit*.
- `active`: se utiliza para marcar que se recibió un paquete *Heard Bit* del vecino.

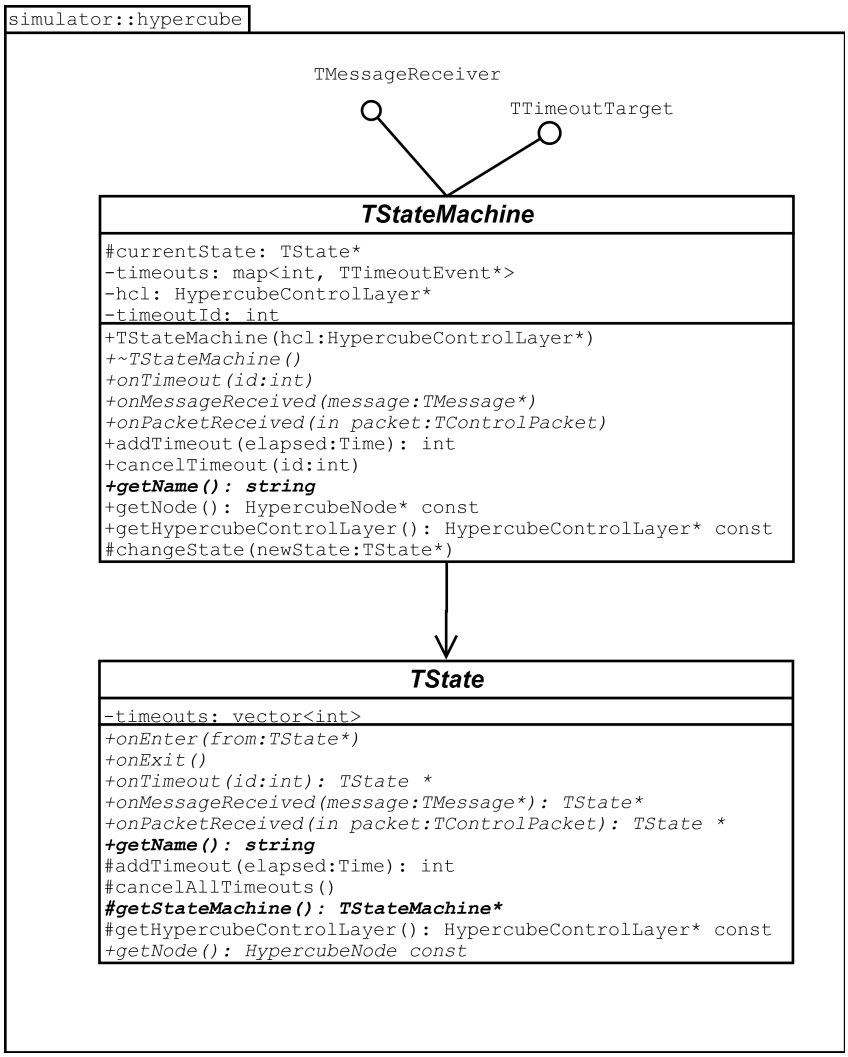


Figura 3.18: Clases `TSateMachine` y `TState`

Los métodos de esta clase son triviales; tan sólo asignan y devuelven los valores de los atributos.

Como se dijo anteriormente, la clase `HypercubeControlLayer` utiliza 3 máquinas de estados. Para implementarlas, se definen las clases `TSateMachine` y `TState`, que son las bases para las máquinas de estados y los estados respectivamente. Estas clases se muestran en la figura 3.18.

La clase `TStateMachine` representa una máquina de estados que recibe mensajes internos del nodo y paquetes desde otros nodos y los transmite al estado actual, siendo éste el encargado de determinar si se debe hacer una transición.

Esta clase provee además la funcionalidad de *timeouts* para los estados; es decir que un estado le requiere un *timeout* luego de un tiempo determinado, y ésta debe notificarle al vencerse el plazo.

Los atributos de la clase `TStateMachine` son los siguientes:

- `currentState`: puntero al estado actual de la máquina.
- `timeouts`: mapa que contiene los *timeouts* requeridos por los estados. Cada *timeout* tiene un entero que lo identifica, para que los estados puedan diferenciarlos en caso de requerir más de uno.
- `hcl`: puntero a la capa de control de hipercubo.

- `timeoutId`: id del último *timeout* requerido, utilizado para asignarle un valor al próximo pedido.

El constructor de la clase inicialia el puntero a la capa de control del hipercubo. Los métodos cumplen las siguientes funciones:

- `onTimeout`: es llamado por el simulador cuando se cumple un *timeout*, y este método llama al método del mismo nombre en el estado actual.
- `onMessageReceived`: es llamado por el nodo cuando se recibe un mensaje interno, y este método llama al método del mismo nombre en el estado actual.
- `onPacketReceived`: es llamado por el simulador cuando se recibe un paquete de otro nodo, y este método llama al método del mismo nombre en el estado actual.
- `addTimeout`: pide un *timeout* para el estado actual.
- `cancelTimeout`: cancela el *timeout* especificado.
- `getName`: método abstracto; las clases descendientes deben devolver el nombre de la máquina de estados.
- `getNode`: devuelve un puntero al nodo donde se encuentra la máquina de estados.
- `getHypercubeControlLayer`: devuelve un puntero a la capa de control de hipercubo.
- `changeState`: método auxiliar para cambiar al estado pasado como parámetro. Para ello, le indica al estado actual que va cambiar de estado mediante el método `onExit`, y luego en el nuevo llama al método `onEnter` para inicializarlo.

Los estados de una máquina se representan con la clase `TState`. El atributo `timeouts` tiene un vector con todos los *timeouts* que el estado requirió. Esto es necesario para que en el caso de que se produzca un cambio de estado antes de que se cumplan se puedan cancelar, evitando que otro estado los reciba.

Los métodos de la clase `TState` son:

- `onEnter`: es llamado cuando se entra al estado.
- `onExit`: es llamado cuando se hace una transición a otro estado. Esta implementación cancela todos los *timeouts*.
- `onTimeout`: es llamado cuando se cumple un *timeout* requerido.
- `onMessageReceived`: es llamado cuando se recibe un mensaje.
- `onPacketReceived`: es llamado cuando se recibe un paquete.
- `getName`: método abstracto; las clases descendientes deben devolver el nombre del estado.
- `addTimeout`: requiere un *timeout*.
- `cancelAllTimeouts`: cancela todos los *timeouts* requeridos.
- `getStateMachine`: método abstracto; las clases descendientes deben devolver un puntero a la máquina de estados.
- `getHypercubeControlLayer`: devuelve un puntero a la capa de control de hipercubo.
- `getNode`: devuelve un puntero al nodo donde se encuentra la máquina de estados.

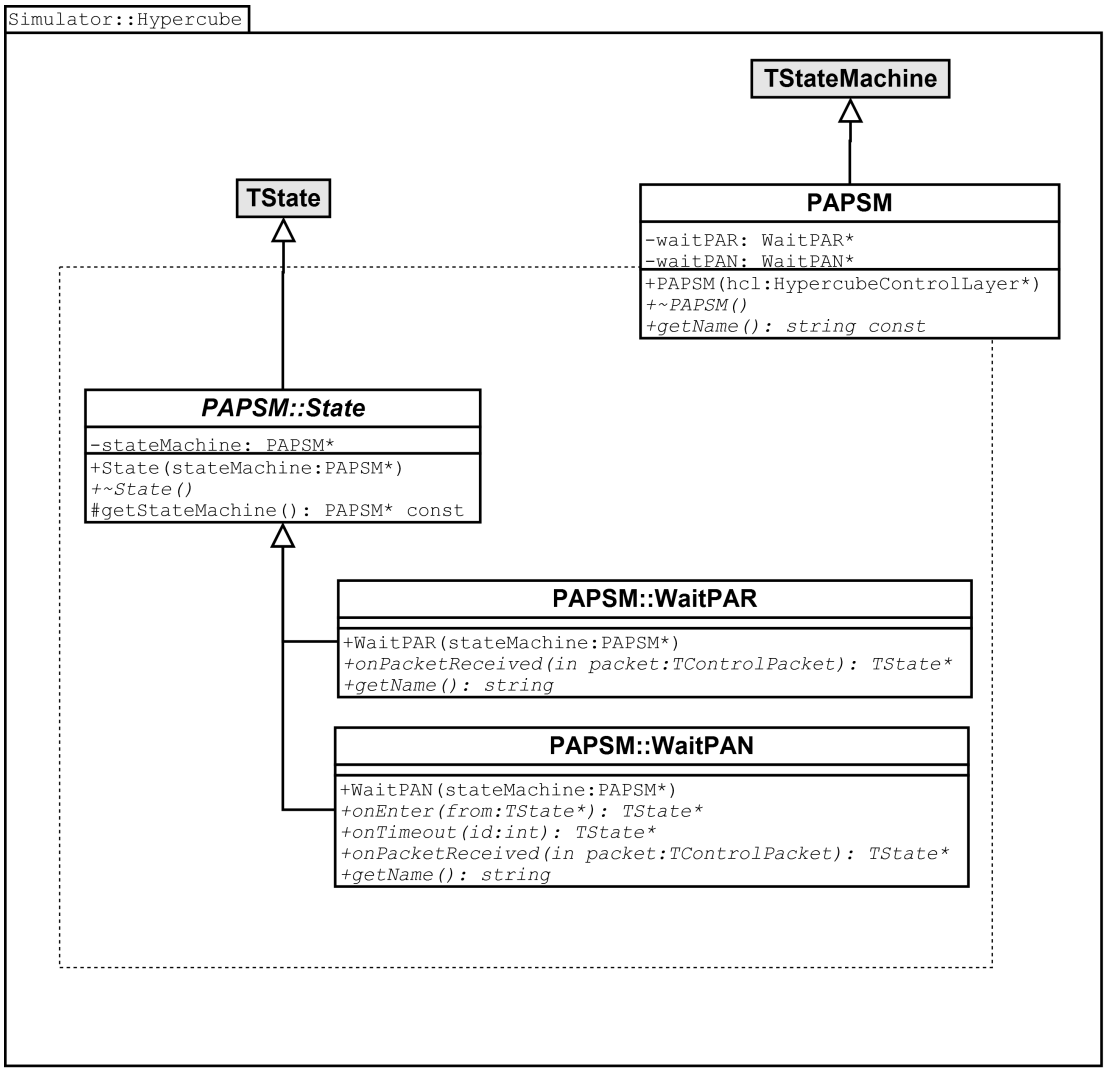


Figura 3.19: Clases para la implementación del máquina de estados `PAP`

Los métodos `onEnter`, `onTimeout`, `onMessageReceived` y `onPacketReceived` devuelven un puntero al próximo estado; a menos que se devuelva `null` se cambiará a ese estado. En esta clase, la implementación por defecto devuelve `null` en todos estos casos.

Se detallará ahora la implementación de las máquinas de estados utilizando estas clases.

La máquina de estados *Primary Address Provider* (ver figura 2.13) se implementa con las clases que muesta la figura 3.19.

La clase `PAPSM` es la implementación de la máquina de estados, siendo sus estados las clases internas `PAPSM::WaitPAR` y `PAPSM::WaitPAN`, heredando ambas de `PAPSM::State`.

Los atributos de `PAPSM` son `waitPAR` y `waitPAN`, punteros a los dos estados de esta máquina.
El constructor instancia los dos punteros e inicializa el estado en *waitPAR*. El método `getName` devuelve la cadena de caracteres "`PAP`".

`PAPSM::State` es la base para los estados, cuya única funcionalidad es proveer un puntero a la máquina de estados donde se encuentra.

La clase `PAPSM::WaitPAR` representa al estado *Wait Primary Address Request*, cuya función es escuchar paquetes a la espera de uno del tipo *PAR*, en cuyo caso se envía la respuesta (un paquete *PAP*) y se pasa al estado *WaitPAP*. Esto está implementado en el método `onPacketReceived`.
El método `getName` devuelve la cadena de caracteres "`WaitPAR`".

La clase `PAPSM::WaitPAN` representa al estado *Wait Primary Address Notification*. Los métodos de esta clase son:

- `onEnter`: requiere un *timeout*, de tal forma que si no se recibe un paquete del tipo *PAN* en un lapso de tiempo, se cancela el proceso de ceder una dirección al vecino.
- `onTimeout`: es llamado cuando se cumplió el *timeout*. Devuelve un puntero a `waitPAR`, para volver a ese estado.
- `onPacketReceived`: si el paquete recibido es de tipo *PAN*, se efectúan las tareas de delegación de espacio de direcciones y se responde con un paquete *PANC* para confirmar la recepción.
- `getName`: devuelve la cadena de caracteres "`WaitPAN`".

La máquina de estados *Heard Bit Listener* (ver figura 2.12) se implementa con las clases que muesta la figura 3.20.

La función de esta máquina de estados es esperar paquetes de tipo *HB* (*Heard Bit*) durante un lapso de tiempo; si alguno de los vecinos no lo envió, se considera que se desconectó de la red abruptamente. Además, en el caso de que se pudiera ofrecer una dirección secundaria a alguno de los vecinos, se envía un paquete con este propósito.

La clase `HBLSM` es la implementación de la máquina de estados, siendo sus estados las clases internas `HBLSM::ListenHB` y `HBLSM::WaitSAN`, heredando ambas de `HBLSM::State`.

Los atributos de `HBLSM` son `ListenHB` y `waitSAN`, punteros a los dos estados de esta máquina.
El constructor instancia los dos punteros e inicializa el estado en *ListenHB*. El método `getName` devuelve la cadena de caracteres "`HBL`".

`HBLSM::State` es la base para los estados, cuya única funcionalidad es proveer un puntero a la máquina de estados donde se encuentra.

La clase `HBLSM::ListenHB` representa al estado *ListenHB*. Sus métodos cumplen las siguientes funciones:

- `onEnter`: marca a todos los vecinos como inactivos (en el atributo `neighbours` de la clase `HypercubeNode`) y requiere un *timeout*, luego del cual todos los vecinos que no hayan mandado un paquete *HB* serán considerados como inactivos.
- `onTimeout`: se verifica si alguno de los vecinos no fue marcado como activo; en ese caso se cambia el estado del vecino a desaparecido, y se manda un mensaje interno del nodo notificando este hecho. Además, si se le puede ofrecer una dirección secundaria a alguno de los vecinos activos, se envía un paquete de tipo *SAP* con la dirección propuesta, y se pasa al estado *WaitSAN*. Si no se ofrecen direcciones secundarias, se entra nuevamente al estado *ListenHB*
- `onPacketReceived`: si el paquete recibido es de tipo *HB*, se marca al vecino que lo envió como activo.
- `getName`: devuelve la cadena de caracteres "`ListenHB`".

La clase `HBLSM::WaitSAN` representa al estado *WaitSAN*. Sus métodos cumplen las siguientes funciones:

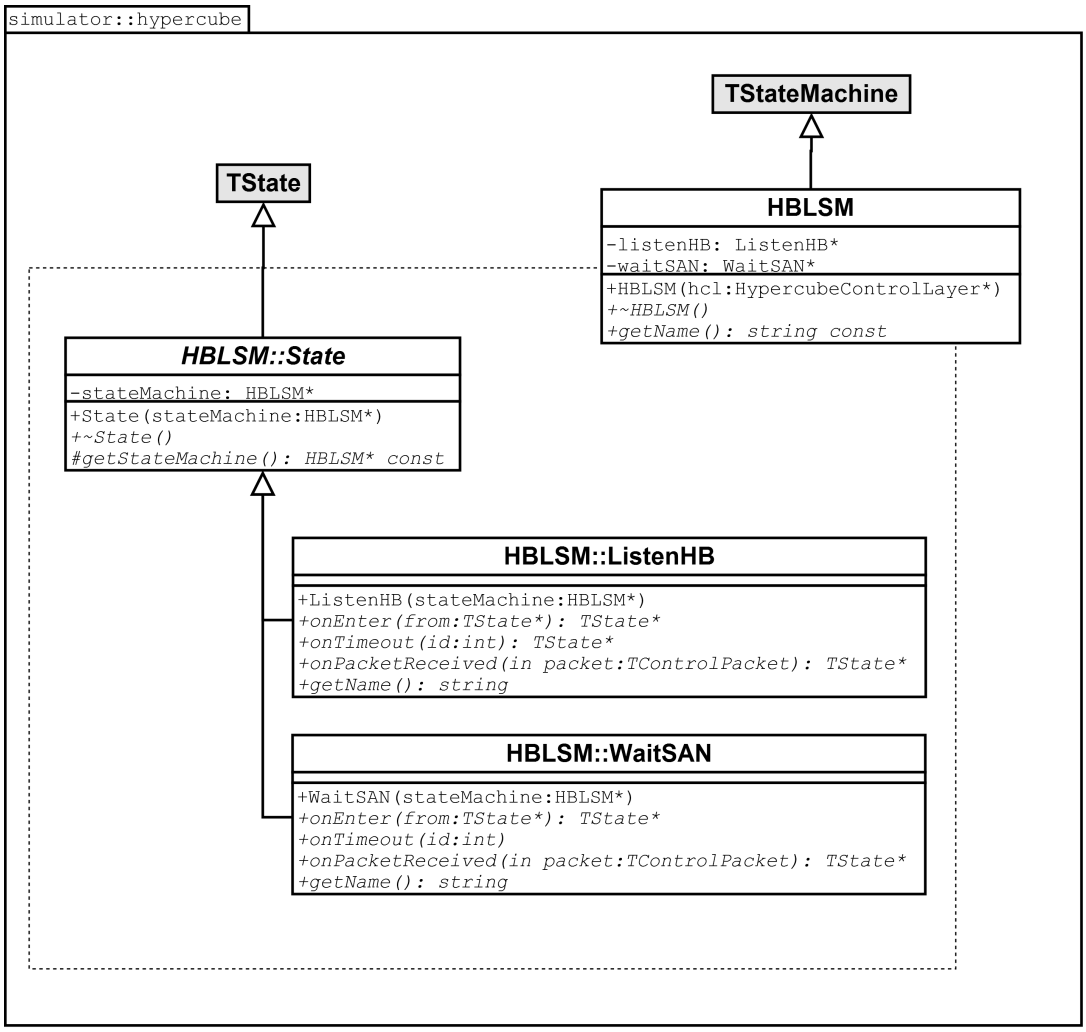


Figura 3.20: Clases para la implementación del máquina de estados `HBL`

- `onEnter`: requiere un *timeout*, transcurrido el cual se cancelará el proceso de ofrecimiento de dirección secundaria.
- `onTimeout`: el tiempo para el ofrecimiento de dirección secundaria concluyó, por lo que se cancela volviendo al estado *ListenHB*.
- `onPacketReceived`: si el paquete recibido es de tipo *SAN* (*Secondary Address Notification*), se comprueba si se aceptó la dirección, en cuyo caso se cede el espacio y se marca al vecino como adyacente. Luego se vuelve al estado *ListenHB*.
- `getName`: devuelve la cadena de caracteres "`WaitSAN`".

La máquina de estados *Main State Machine* (ver figura 2.11) se implementa con las clases que muestra la figura 3.21.

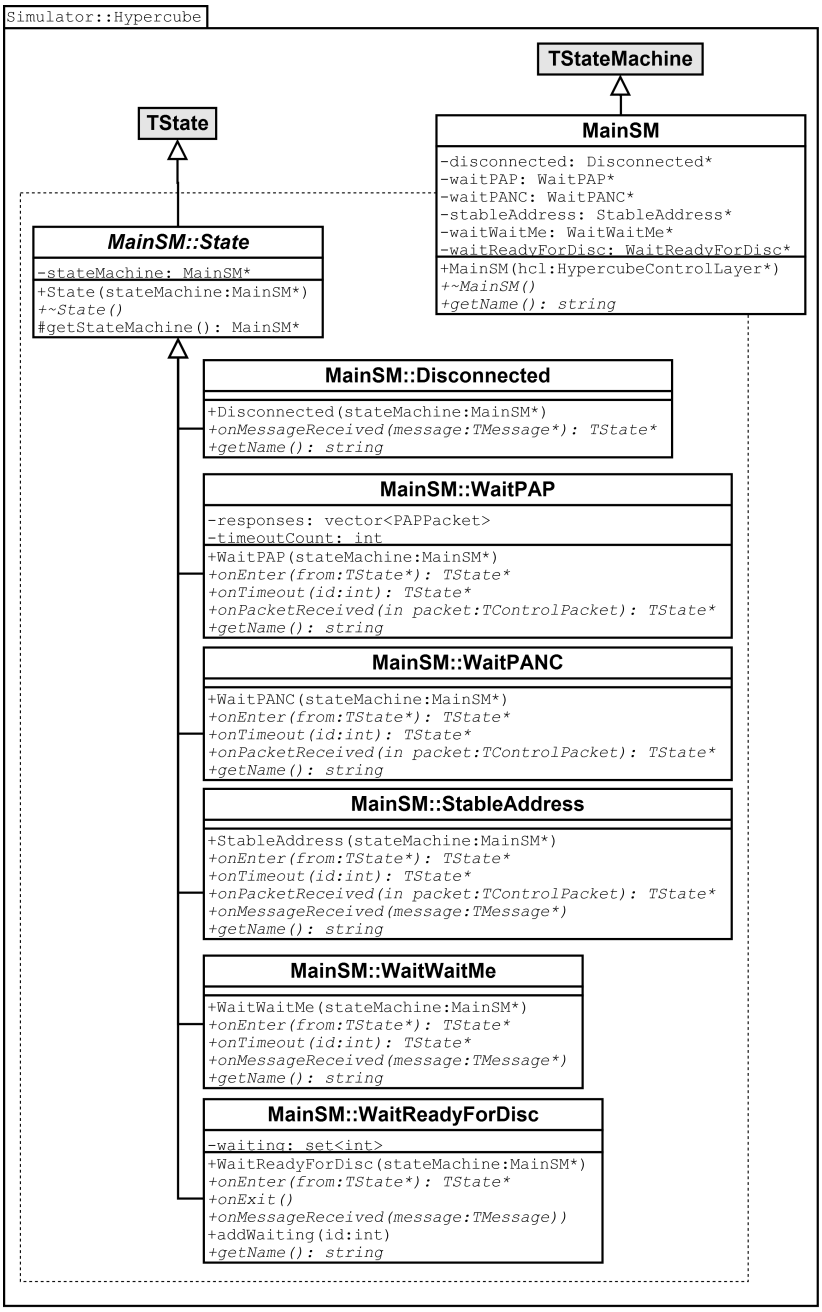


Figura 3.21: Clases para la implementación del máquina de estados `Main State Machine`

Esta máquina de estados se ocupa de conectar y desconectar el nodo a la red, implementada mediante la clase `MainSM`. Los atributos de esta clase son punteros a los estados, cuya función se explicará al detallar cada estado. El constructor instancia todos estos objetos e inicializa la máquina de estados en *disconnected*.

La clase `MainSM::State` es la clase base para los estados, cuya única funcionalidad es proveer un puntero a la máquina de estados donde se encuentra.

El estado *disconnected* (clase `MainSM::Disconnected`) indica que el nodo no está conectado a la red ni está en proceso de conexión. Cuando se recibe un mensaje interno *join_network* (es decir, el usuario requiere que el nodo se conecete a la red), se cambia al estado *WaitPAP*. Esta funcionalidad se encuentra implementada en el método `onMessageReceived`.

El método `getName` devuelve la cadena de caracteres "`Disconnected`".

En el estado *WaitPAP* (clase `MainSM::WaitPAP`) se envía un pedido de dirección primaria y se espera su respuesta. Las propuestas recibidas se guardan en el vector `responses`. El atributo `timeoutCount` contiene la cantidad de veces que se obtuvo timeout, ya que a las 5 veces que no se obtiene respuesta se considera que no hay otros nodos en la red.

Los métodos de esta clase son:

- `onEnter`: cuando se entra a este estado, se manda un paquete de tipo *PAR* (*Primary Address Request*) en modo *broadcast* a todos los vecinos.
- `onTimeout`: si algún vecino propuso una dirección primaria o dirección de reconexión, se escoge la dirección cuya máscara sea más pequeña y se pasa al estado *WaitPANC*. Si es la quinta vez que se cumple el *timeout* sin recibir respuesta, se utiliza la dirección con todos los bits en cero y se pasa al estado *WaitPANC*.
- `onPacketReceived`: los paquetes de tipo *PAP* se guardan en el vector `responses`.
- `getName`: devuelve la cadena de caracteres "`WaitPAP`".

Si se recibieron propuestas de direcciones, luego de escoger una se entra al estado *WaitPANC* (clase `MainSM::WaitPANC`), que se encarga de notificar la decisión y esperar la confirmación.

Para ello, cuenta con los siguientes métodos:

- `onEnter`: envía un paquete de tipo *PAN* (*Primary Address Notification*) en modalidad *broadcast* a todos los vecinos para notificar la dirección primaria escogida.
- `onTimeout`: se venció el tiempo de espera sin que el vecino cuya dirección fue escogida responda a la notificación; por ello se cancela la conexión con este vecino, volviendo al estado *WaitPAP*, donde volverá a comenzar el proceso de conexión.
- `onPacketReceived`: si se recibe la confirmación del vecino mediante un paquete *PANC* (*Primary Address Notification Confirmation*), se pasa al estado *StableAddres*.
- `getName`: devuelve la cadena de caracteres "`WaitPANC`".

Una vez que el nodo tiene dirección primaria, tanto porque fue dada por un vecino como porque no se encontraron otros nodos en la red, se entra al estado *StableAddress* (clase *MainSN::StableAddress*). Se permanecerá en este estado hasta que se requiera que el nodo se desconecte a la red.

1. `onEnter`: se inicializa el *timeout* para enviar *Heard Bits* a los vecinos, indicando que el nodo sigue conectado.
2. `onTimeout`: envía un paquete *HB* en modalidad *broadcast* a todos los vecinos.
3. `onPacketReceived`: si se recibe un paquete de tipo *SAP* (*Secondary Address Proposal*) proponiendo una dirección secundaria, se responde con un paquete *SAN* (*Secondary Address Notification*) indicando si se acepta o no la propuesta.

   Si se recibe un paquete de tipo *DISC*, indicando que un vecino se desconectó, se guarda su dirección para reconexión si fuera necesario y se notifica dentro del nodo que un vecino se desconectó.

4. `onMessageRececived`: si se recibe un mensaje de tipo *LeaveNetworkMessage*, se pasa al estado *WaitWaitMe*.

5. `getName`: devuelve la cadena de caracteres "`StableAddress`".

Cuando se recibe el mensaje *LeaveNetworkMessage* indicando que el usuario desea desconectarse de la red, se pasa al estado *WaitWaitMe* (clase `MainSM:WaitWaitMe`) cuya función es notificar dentro del nodo que se producirá la desconexión, para que otros procesos puedan realizar las tareas necesarias para finalizar limpiamente. Los procesos que deban realizar tareas, deben responder con un mensaje de tipo *WaitMe* utilizando un identificador único. Una vez que hayan concluido, deben enviar un mensaje de tipo *ReadyForDisc* utilizando el mismo identificador. De esta forma, cuando todos los procesos finalizaron sus tareas de limpieza, se puede proceder a la desconexión.

Los métodos de esta clase son:

- `onEnter`: envía un mensaje de tipo *WillDisconnect* dentro del nodo, y pide un *timeout* para que los otros procesos tengan tiempo de reportarse.

- `onTimeout`: finalizó el tiempo para que los procesos hagan un pedido de espera, por lo que se pasa al estado *WaitReadyForDisc*.

- `onMessageReceived`: si se recibe un mensaje de tipo *WaitWaitMe*, se utiliza el identificador único para llamar al método `addWaiting` en el objeto `MainSM::WaitReadyForDisc`, que lo agregará a la lista de identificadores en espera.

  Si el mensaje recibido es de tipo *ReadyForDisc*, se utiliza este mismo mensaje para llamar al método `onMessageReceived` en el objeto `MainSM::WaitReadyForDisc`, que eliminará el identificador de la lista. De esta forma es tenido en cuenta el caso de qu un proceso pide ser esperado y finaliza antes de que se cambie de estado.

- `getName`: devuelve la cadena de caracteres "`WaitWaitMe`".

Una vez terminado el tiempo de espera para que los procesos pidan ser esperados, se entra en el estado *WaitReadyForDisc* (clase `MainSM::WaitReadyForDisc`) en el cual se espera que todos los procesos finalizen para pasar al estado de desconexión.

El atributo *waiting* es un conjunto con los identificadores de los procesos que están en espera. Los métodos de esta clase son:

- `onEnter`: si el conjunto `waiting` está vacío, se pasa al estado *disconnected*.

- `onMessageReceived`: si el mensaje recibido es de tipo *ReadyForDisc*, se elimina el identificador del conjunto `waiting`. Si el conjunto queda vacío, se pasa al estado *disconnected*.

- `addWaiting`: se agrega el identificador al conjunto `waiting`.

- `onExit`: se envía un paquete de tipo *DISC* en modalidad *broadcast* a todos los vecinos, indicando que el nodo se está desconectando.

- `getName`: devuelve la cadena de caracteres "`WaitReadyForDisc`".

Desde este estado, se vuelve al estado inicial, *disconnected*, desde donde el proceso de conexión podrá comenzar nuevamente en caso de que el usuario lo requiera.

Capa de Transporte

La capa de transporte está compuesta por la clase base abstracta, `TTransportLayer`, y una implementación similar a UDP en la clase `UDPTransportLayer`, como muestra la figura 3.22.

La clase `TTransportLayer` contiene un puntero a la capa de red sobre la que se ubica esta capa de transporte, en el atributo `networkLayer`. El constructor proviste inicializa este puntero así como el puntero al nodo. Los métodos de la clase son abstractos, siendo la funcionalidad que deben implementar las clases descendientes:

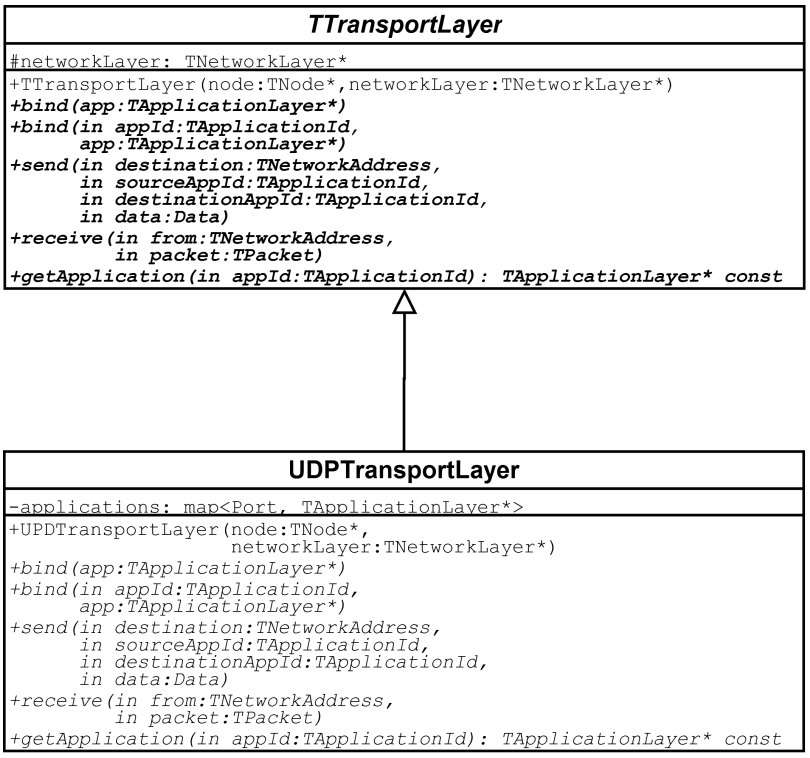


Figura 3.22: Clases de la capa de transporte

- `bind`: asigna una capa de aplicación a un identificador. Si no se especifica el identificador, se le debe asignar uno disponible.
- `send`: la capa de red llamará a este método para enviar datos. Debe construir un segmento y enviarlo a través de la capa de red.
- `receive`: es llamado desde la capa de red cuando se recibe un paquete. Debe extraer los datos del paquete, decidir a que aplicación le corresponde recibirlos según el identificador de aplicación, y llamar a un método en esta aplicación para que reciba los datos.
- `getApplication`: devuelve un puntero a la aplicación que utiliza el identificador pasado como argumento, o `null` si no existe ninguna aplicación con ese identificador.

La clase `UDPTransportLayer` implementa una capa de transporte. En el atributo `applications` se almacena un mapa de las aplicaciones registradas, utilizando como clave el puerto (*port*), que es el identificador de aplicación utilizado en UDP.

Los métodos de esta clase son:

- `bind`: agrega el puntero a la aplicación en el mapa `applications`. Si no se provee un puerto, se utiliza el primero disponible a partir de 1024.
- `send`: crea un objeto `UDPSegment` a partir de los puertos de origen y destino y de los datos, para luego enviarlo a través de la capa de red con el método `send`.
- `receive`: crea un objeto `UDPSegment` a partir del paquete recibido para extraer los datos y el puerto destino. Utilizando este puerto, busca la aplicación registrada en él mediante el mapa `applications`, y llama a su método `receive`.
- `getApplication`: devuelve la aplicación que utiliza el identificador pasado como argumento, buscándolo en el mapa `applications`.

Capa de Aplicación

En esta capa se implementaron varias aplicaciones:

- *Rendez-Vous Client*: forma parte de `ANTop`. Se encarga de resolver direcciones universales.
- *Rendez-Vous Server*: forma parte de `ANTop`. Responde a los pedidos de direcciones universales y se encarga de registrar y des-registrar al nodo.
- *Test Application*: aplicación de prueba para enviar datos a un nodo.
- *Trace Route*: permite conocer la ruta que siguen los paquetes al ser enviados.

Las clases para estas aplicaciones heredan de `HypercubeBaseApplication`, que a su vez hereda de `TApplicationLayer`. La figura 3.23 muestra estas dos clases.

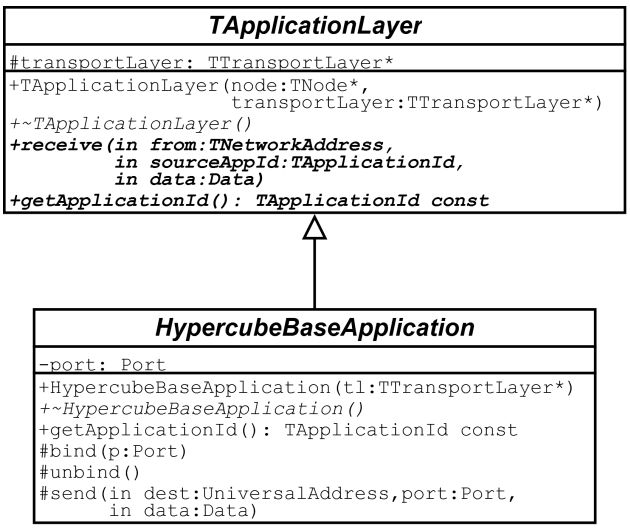


Figura 3.23: Clases `HypercubeBaseApplication` y `TApplicationLayer`

`TApplicationLayer` provee un puntero a la capa de transporte sobre la que se encuentra la aplicación. El constructor le da valor a este puntero, así como al puntero al nodo donde se encuentra.

El método abstracto `receive` es llamado por la capa de transporte cuando se reciben datos.
El otro método abstracto, `getApplicationId`, debe ser implementado para que devuelva el identificador de aplicación que está siendo utilizado.

La clase `HypercubeBaseApplication` provee métodos adicionales para las aplicaciones de hipercubo. El atributo `port` guarda el puerto UDP utilizado por la aplicación.

Los métodos cumplen con las siguientes funciones:

- `getApplicationId`: devuelve el atributo `port`.
- `bind`: llama a la capa de transporte requiriendo la utilización del puerto pasado como parámetro para la aplicación.
- `unbind`: llama a la capa de transporte para liberar el puerto utilizado.
- `send`: envía datos a oto nodo, utilizando su dirección universal y el puerto de destino. Este método utiliza la aplicación de *Rendez-Vous Client* para el envío, ya que esta aplicación es la encargada de resolver las direcciones universales.

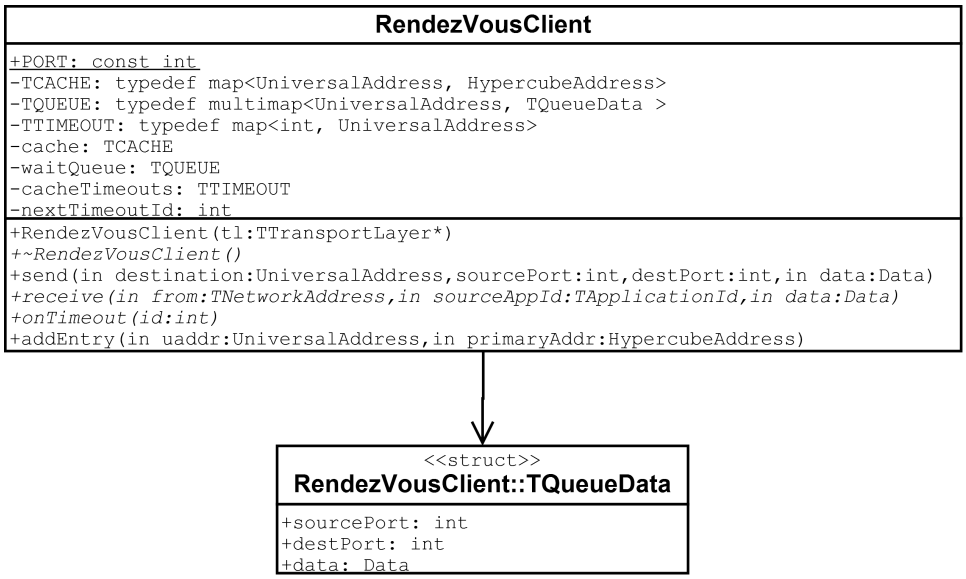


Figura 3.24: Clase `RendezVousClient`

La clase `RendezVousClient` se muestra en la figura 3.24.

Ésta permite que otras aplicaciones puedan enviar datos sin necesidad de ocuparse de la resolución de direcciones; tan sólo deben llamar a su método `send` y la clase se encarga de resolver la dirección universal y enviar los datos.

Los atributos de la clase son:

- `PORT`: esta constante indica el puerto de la aplicación; se utiliza el 9903.
- `cache`: es el mapa que almacena las resoluciones obtenidas para que se puedan volver a utilizar sin necesidad de repetir el proceso de resolución. Las entradas son eliminadas una vez que transcurre un *timeout* para evitar guardar información que no es utilizada.
- `waitQueue`: cola de espera donde se almacenan los datos que deben ser enviados y cuya dirección de hipercubo no esta aún resuelta. Se utiliza la estructura `RendezVousClient::TQueueData`, que almacena los puertos de origen y destino y los datos a enviar.
- `cacheTimeouts`: asocia los identificadores de *timeout* con las entradas que deben ser eliminadas al cumplirse el *timeout*.
- `nextTimeoutId`: valor del próximo identificador de *timeout* que será utilizado.

El constructor de la clase utiliza el método `bind` de `HypercubeBaseApplication` para que se le asigne el puerto a la aplicación. El destructor llama a *unbind* para liberar el puerto.

Los métodos cumplen con la siguiente funcionalidad:

- `send`: si la dirección universal se encuentra en la *caché*, se resuelve a través de la misma. Si no, se ponen los datos en la cola de espera, se calcula la dirección de hipercubo del *Rendez-Vous* para el destino, y se le envía un pedido de resolución. (ver algoritmo 9)
- `receive`: si se recibió una resolución de dirección universal, se agrega a la `caché`, se buscan todos los paquetes en la cola de espera que utilizan ésta dirección y se envían. (ver algoritmo 10)
- `onTimeout`: busca en el atributo `cacheTimeouts` la entrada correspondiente al identificador de *timeout*. Si la entrada no fue utilizada, es eliminada. Si fue utilizada, se marca como no utilizada y se reinicializa el *timeout*.

La clase `RendezVousServer` (figura 3.25) se encarga de resolver los pedidos del *Rendez-Vous Client*, asi como de registrar al nodo en el *Rendez-Vous* cuando se conecta y des-registrarlo cuando se desconecta.

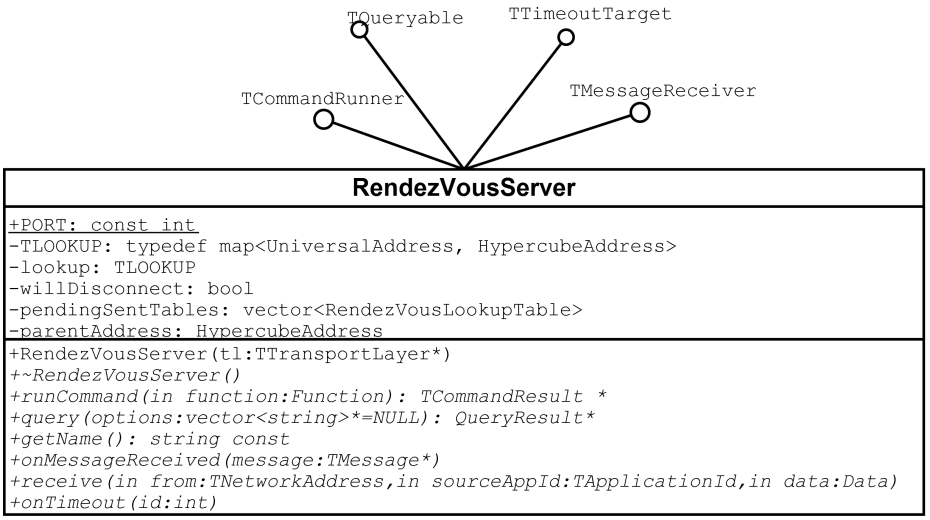


Figura 3.25: Clase `RendezVousServer`

Los atributos de la clase son los siguientes:

- `PORT`: esta constante indica el puerto de la aplicación; se utiliza el 9902.
- `lookup`: contiene la tabla de *Rendez-Vous*, es decir, la asociación entre direcciones universales y de hipercubo.
- `willDisconnect`: vale *true* si el nodo está en proceso de desconexión, en cuyo caso la tabla de *Rendez-Vous* debe ser cedida.
- `pendingSentTables`: utilizado cuando el nodo se está desconectando de la red, contiene los fragmentos de tabla que fueron cedidos pero aún se espera la confirmación.
- `parentAddress`: dirección del nodo padre.

El constructor de la clase utiliza el método `bind` de `HypercubeBaseApplication` para que se le asigne el puerto a la aplicación. El destructor llama a *unbind* para liberar el puerto.
Los métodos cumplen con la siguiente funcionalidad:

- `runCommand`: ejecuta un comando en el objeto.
- `query`: realiza una consulta de la tabla de *Rendez-Vous*.
- `getName`: devuelve la cadena de caracteres "`WaitPAN`".
- `onMessageReceived`: según el mensaje recibido hace lo siguiente:
  - *Connected*: el nodo se conectó a la red, por lo que envía un paquete de registración.
  - *WillDisconnect*: se prepara para la desconexión, enviando las tablas de *Rendez-Vous* a los vecinos.
  - *AddressGiven*: parte del espacio de direcciones fue cedido, por lo que envía las entradas que ya no le corresponden al nuevo vecino.
- `receive`: según el tipo de paquete recibido hace lo siguiente (ver algoritmo 11):

- *RendezVousRegister*: un nodo envió un pedido de registración, por lo que su dirección universal y de hipercubo se agregan en la tabla.
- *RendezVousDeregister*: un nodo envió un pedido de des-registración, por lo que se borra su entrada.
- *RendezVousAddressSolve*: pedido de resolución de dirección universal; la busca en la tabla y envía la respuesta.
- *RendezVousLookupTable*: un nodo envió un fragmento o toda su tabla de *Rendez-Vous*, por lo que las entradas se incorporan a la tabla propia y se confirma la recepción.
- *RendezVousLookupTableReceived*: se está confirmando la recepción de entradas de la tabla enviadas, por lo que son borradas de `pendingSentTables`. Si está en proceso de desconexión y no quedan más entradas por confirmar, se envía un mensaje *ReadyForDiscMessage* indicando que el proceso está listo para la desconexión.

- `onTimeout`: no hubo respuesta al envío de la tabla de *Rendez-Vous*, por lo que se fuerza la desconexión.

Una de las aplicaciones que se implementó para realizar pruebas es la que se muestra en la figura 3.26.

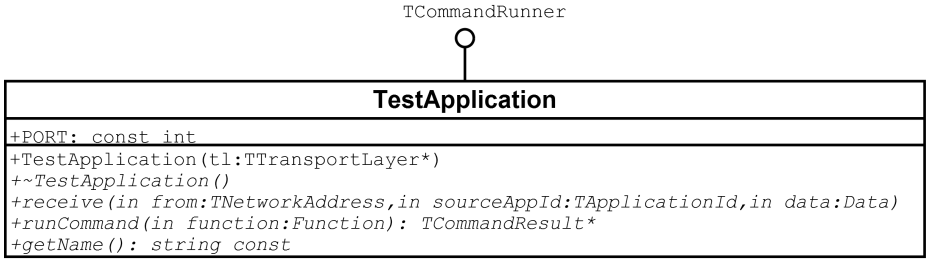


Figura 3.26: Clase `TestApplication`

Esta aplicación tan sólo envía datos a pedido del usuario a la misma aplicación en otro nodo y notifica cuando los recibe. Utiliza el puerto 9920.

El constructor de la clase utiliza el método `bind` de `HypercubeBaseApplication` para que se le asigne el puerto a la aplicación. El destructor llama a *unbind* para liberar el puerto.
Los métodos cumplen con la siguiente funcionalidad:

- `receive`: notifica que se recibieron datos, dando información de la longitud del camino y distancia.
- `runCommand`: ejecuta el comando *send*, que envía un paquete de prueba a la dirección especificada.
- `getName`: devuelve el nombre del objeto ("TestApplication").

La otra aplicación es *TraceRoute*, que se muestra en la figura 3.27. Utiliza el puerto 9901.

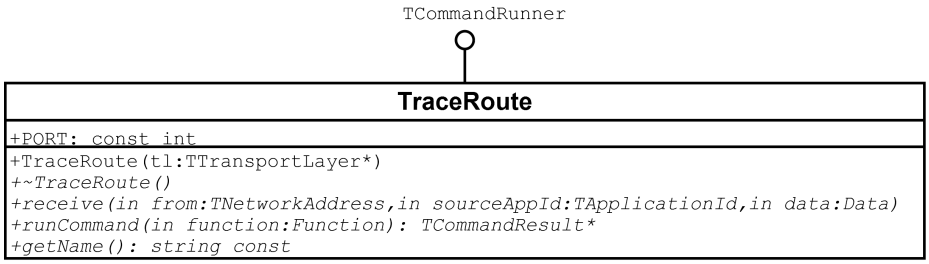


Figura 3.27: Clase `TraceRoute`

El constructor de la clase utiliza el método `bind` de `HypercubeBaseApplication` para que se le asigne el puerto a la aplicación. El destructor llama a *unbind* para liberar el puerto.

Los métodos cumplen con la siguiente funcionalidad:

- `receive`: este método no hace nada, dado que la información de ruteo no llega a la capa de aplicación. Por ello, cuando un pedido de trazado de ruta se recibe, la capa de red se encarga de notificarlo.
- `runCommand`: ejecuta los comandos de trazado, enviando un paquete a la dirección especificada.
- `getName`: devuelve el nombre del objeto ("TraceRoute").

3.2.3. Unidades de Datos

Cada capa de protocolo del nodo se comunica con la capa del mismo nivel en otro nodo utilizando un formato particular de datos. Las capas se comunicarán entre si como si estuvieran directamente conectadas y pudieran enviar y recibir ese formato de datos. Sin embargo, para que ello ocurra, las capas inferiores deben ocuparse de brindar tal servicio. Para desacoplar las capas, cada nivel debe desconocer el formato de los otros niveles. Entonces, una capa inferior toma los datos de la capa superior con todos los encabezados y datos y los encapsula en su sección de datos sin importarle el formato.

Para simular esto, el namespace `simulator::dataUnit` contiene clases generales que representan los distintos formatos de datos intercambiados entre capas. Las clases de unidades de datos para el hipercubo se encuentran en el paquete `simulator::hypercube::dataUnit`. Estas clases se ven en la figura 3.28.

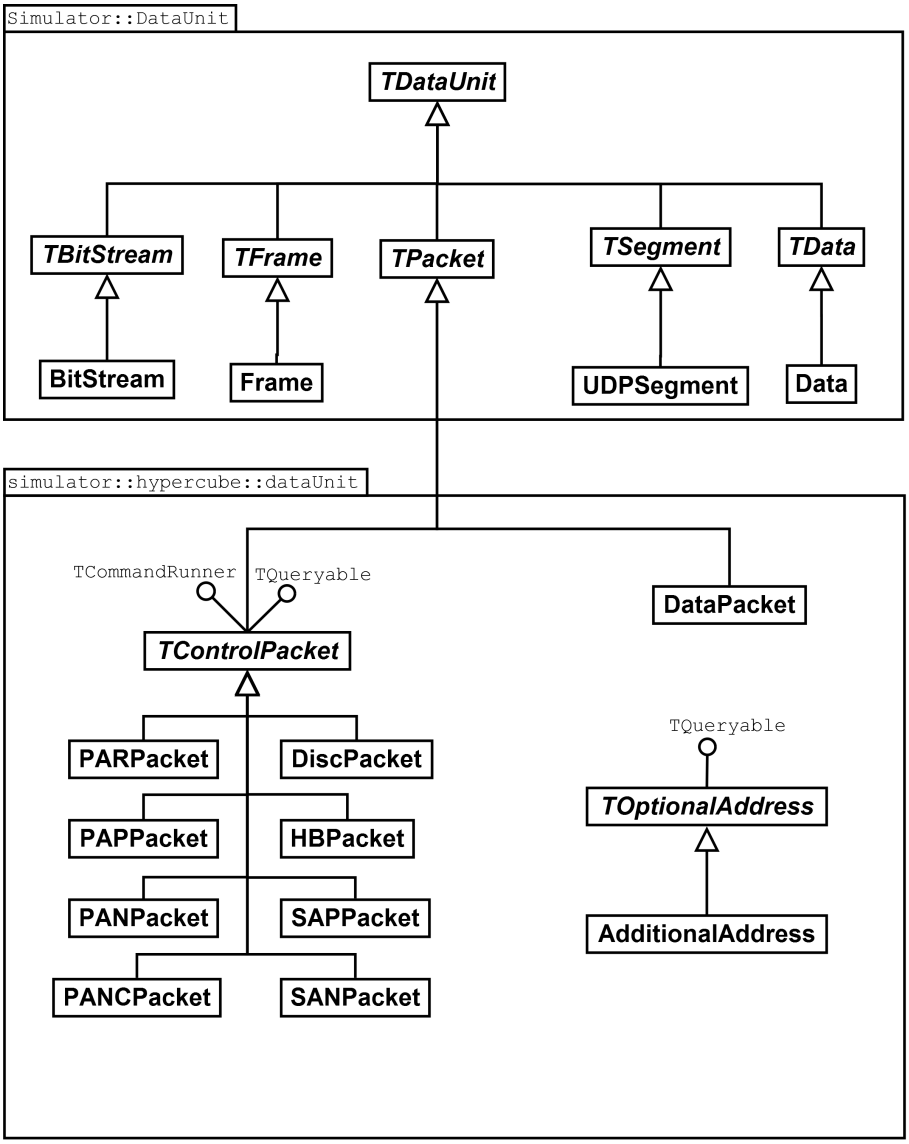


Figura 3.28: Diagrama de clases de las Unidades de Datos

La clase base, `TDataUnit` (ver figura 3.29), provee un campo donde se deben almacenar los datos que cada unidad de datos transporta (esto excluye al encabezado) y el método `getPayload` para acceder a estos datos. El método abstracto `dumpTo` debe volcar los encabezados propios del formato de datos y los datos en el iterador dado.

Heredando directamente de `TDataUnit` se encuentran las clases `TBitStream`, `TFrame`, `TPacket`, `TSegment`

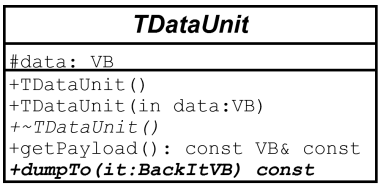


Figura 3.29: clase `TDataUnit`

y `TData`, que son las clases bases para la comunicación entre capas físicas, de enlace de datos, de red, de transporte y de aplicación respectivamente.

La finalidad de estas clases es que las distintas capas puedan utilizar como parámetro un tipo de datos de otro nivel sin necesidad de especificar que tipo exacto necesitan, permitiendo de esta forma que se puedan intercambiar las capas.

Se proveen implementaciones para los distintos tipos de datos, cada una de las cuales permite construir un tipo de datos a partir de otro de la capa inferior o de la capa superior. En el caso de construirlo a partir de los datos de la capa inferior, se extraen los datos que esta capa transporta y se obtiene el encabezado. En cambio, cuando se construye a partir de los datos de la capa superior, es necesaria información adicional, en general de las direcciones de ese nivel y la identificación del protocolo.

BitStream

Un *Bit Stream* es un flujo de bits que circula entre los nodos sin ningún formato asociado. La figura 3.30 muestra las clases para su implementación.

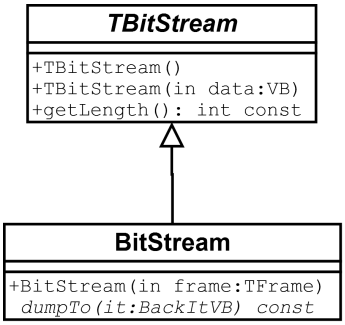


Figura 3.30: clases `TBitStream` y `BitStream`

La clase `TBitStream` es la clase base. Los datos que transporta se almacenan en el atributo `data` de su clase padre (`TDataUnit`). El método `getLength` devuelve la longitud de éste vector.

La clase `BitStream` es una implementación trivial del *Bit Stream*. No utiliza ningún tipo de encabezado.

El constructor permite instanciar el objeto a partir de un *Frame*.

El método `dumpTo` copia los datos en el vector `data` al iterador pasado.

Frame

En el nivel de capa de enlace de datos, se intercambian *frames*, cuya clase base es `TFrame`, como muestra la figura 3.31.

La clase `Frame` representa un *frame* similar a los definidos por Ethernet II. Los atributos de la clase son los campos variables que contiene el encabezado:

- `source`: dirección MAC de origen.

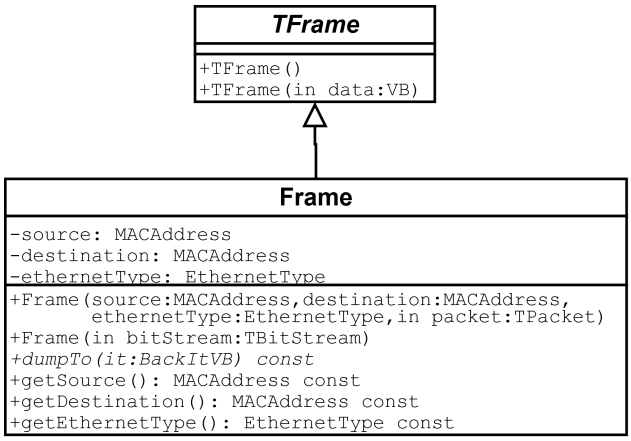


Figura 3.31: clases `TFrame` y `Frame`

- `destination`: dirección MAC de destino.
- `ethernetType`: determina el tipo de paquete que se está transportando, lo cual permite elegir la capa de red que lo debe recibir.

Uno de los constructores tiene como parámetros valores para estos tres atributos, así como un paquete. Este constructor tan sólo almacena los valores en la instancia.

El consturctor que tiene como parámetro un `TBitStream` extrae los datos del mismo y lo interpreta, almacenando los valores del encabezado y datos.

El método `dumpTo` copia los valores del encabezado y de los datos al iterador. Los otros tres métodos, `getSource`, `getDestination` y `getEthernetType` devuelven los nombres de los atributos correspondientes.

Packet

En la capa de red se intercambian paquetes, que en principio se dividen en paquetes de control (`TControlPacket`) y paquetes de datos (`DataPacket`), utilizados por `HypercubeControlLayer` y `HypercubeRoutingLayer` respectivamente.

La clase `TControlPacket` (ver figura 3.32) es la clase base para los paquetes de control.

Los atributos de esta clase son:

- `ETHERNET_TYPE`: esta constante contiene el identificador para los paquetes de control, cuyo valor es 1000.
- `totalLength`: longitud total del paquete, incluyendo encabezado y datos.
- `typeAndFlags`: los 3 bits más altos contienen *flags*; los restantes 5 bits identifican el tipo de paquete.
- `physicalAddress`: dirección MAC del nodo de origen.
- `primaryAddress`: dirección primaria de hipercubo del nodo de origen.

El constructor con parámetros le dá valores a los atributos. El constructor que no lleva parámetros sirve para que las clases descendientes puedan hacer su propia inicialización.

Los métodos de esta clase son:

- `getType`: devuelve el tipo de paquete, filtrando los 5 bits más bajos del atributo `typeAndFlags`.
- `getPhysicalAddress`: devuelve la dirección MAC del nodo de origen.

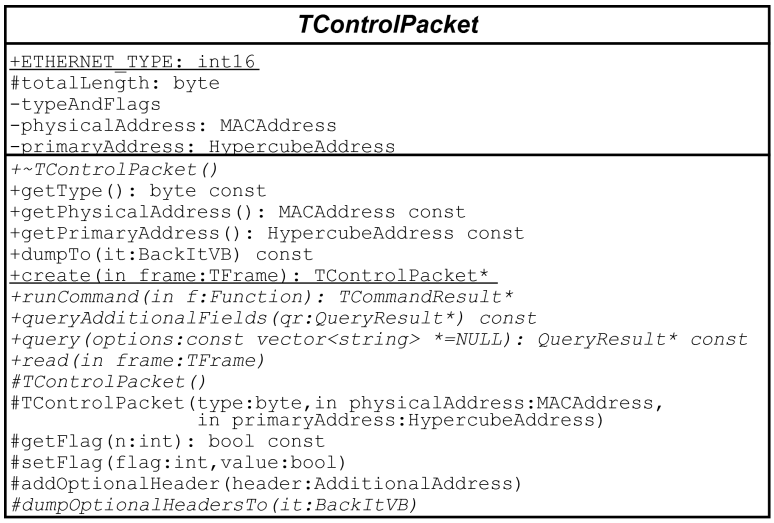


Figura 3.32: clase `TControlPacket`

- `getPrimaryAddress`: devuelve ladirección primaria de hipercubo del nodo de origen.
- `dumpTo`: copia el encabezado y los datos al iterador.
- `create`: este método estático toma un `TFrame` como parámetro, detecta cuál es el tipo de paquete que contiene el *frame* e instancia la clase correspondiente a ese tipo.
- `runCommand`: ejecuta un comando, siendo el único disponible el de consulta ("`query`").
- `queryAdditionalFields`: este método debe ser heredado por las clases que deseen proveer valores adicionales a la consulta.
- `query`: realiza una consulta del encabezado del paquete.
- `read`: llena los campos del paquete de control a partir de un *frame*.
- `getFlag`: devuelve el *flag* indicado, obtenido de los 3 primeros bits del atributo *typeAndFlags*.
- `setFlag`: da valor al *flag* indicado.
- `addOptionalHeaders`: es llamado cuando se encuentra un encabezado opcional de tipo `AdditionalAddress`, para que el paquete pueda guardarlo en sus campos.
- `dumpOptionalHeadersTo`: las clases descendientes que utilizen encabezados opcionales deben sobreescribir este método para copiarlos al iterador.

De esta clase heredan todos los paquetes de control, descriptos en 2.2.4. La figura 3.33 muestra estas clases.

Todas ellas tienen un funcionamiento similar, por lo que se explicarán los elementos en común.
Cada una tiene una constante `TYPE`, que es el identificador del paquete.
Se provee un constructor vacío que no inicializa los campos, en cuyo caso se debe llamar luego al método `read`. Luego, se provee por lo menos un constructor más que toma como parámetros los campos del encabezado, inicializando los atributos con estos valores.

El método `getName` devuelve el nombre del paquete. Los métodos que empiezan con `get` o `is` devuelven valores de los atributos, siendo la diferencia que los que utilizan `is` devuelven un valor de tipo *bool*.

Algunas de estas clases utilizan el encabezado opcional `AdditionalHeader`, que fue implementado mediante una clase del mismo nombre, descendiente de *TOptionalHeader*. Estas clases se muestran en la

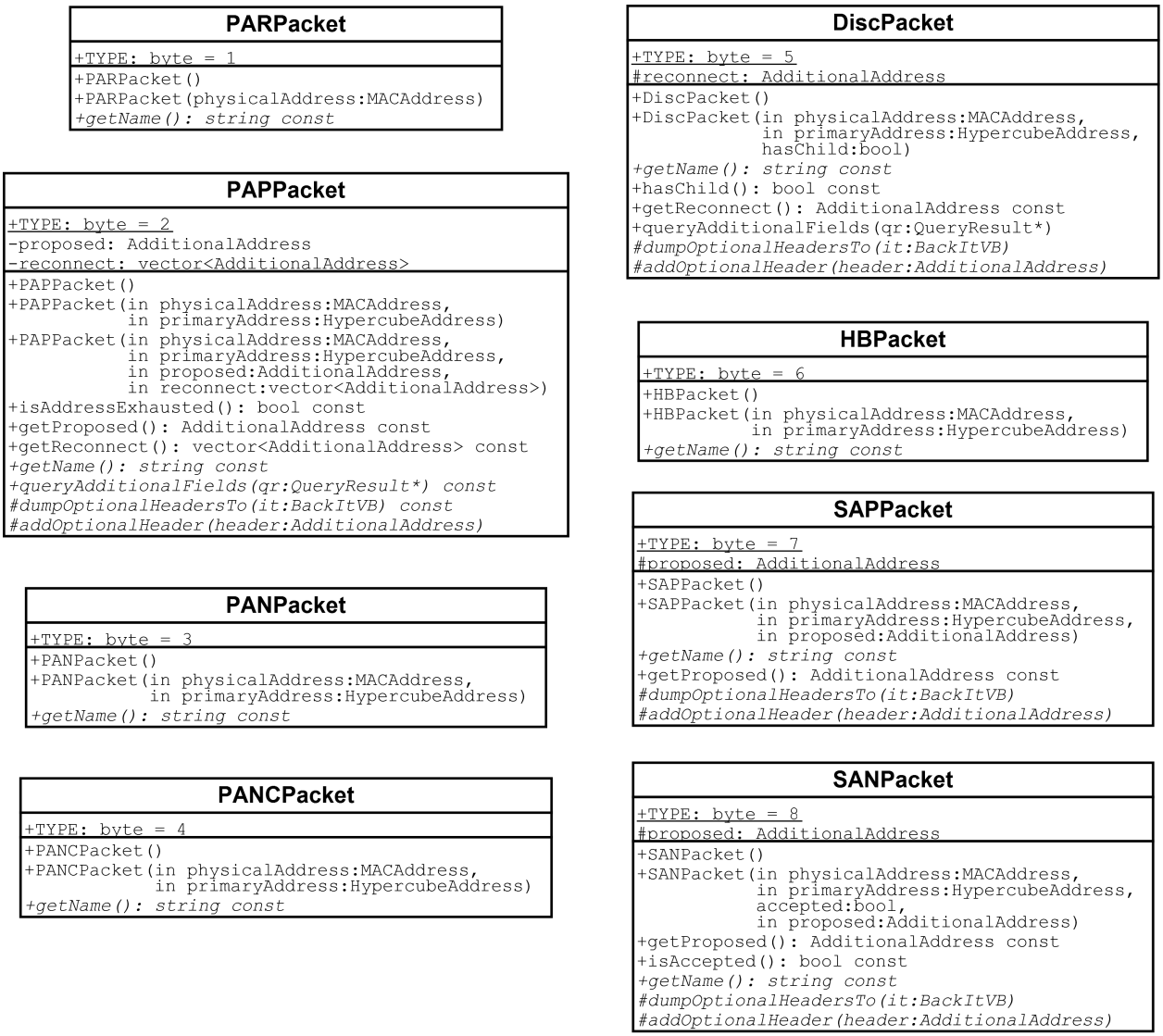


Figura 3.33: Clases para los paquetes de control

figura 3.34.

La clase `TOptionalHeader` es la clase base para los encabezados opcionales. Provee un tipo de encabezado opcional de 5 bits, permitiendo hasta 32 tipos diferentes, 3 flags y una longitud de hasta 255 bytes. Los métodos de esta clase son:

- `getHeaderType`: devuelve el tipo de encabezado, filtrando los 5 bits más bajos del atributo `typeAndFlags`.
- `getFlag`: devuelve el *flag* indicado, obtenido de los 3 primeros bits del atributo `typeAndFlags`.
- `dumpTo`: copia el encabezado opcional al iterador.
- `getType`: lee el tipo de encabezado del iterador pasado.
- `create`: método estático que crea un encabezado opcional a partir de sus datos crudos leídos mediante el iterador.
- `getLength`: devuelve la longitud del encabezado.

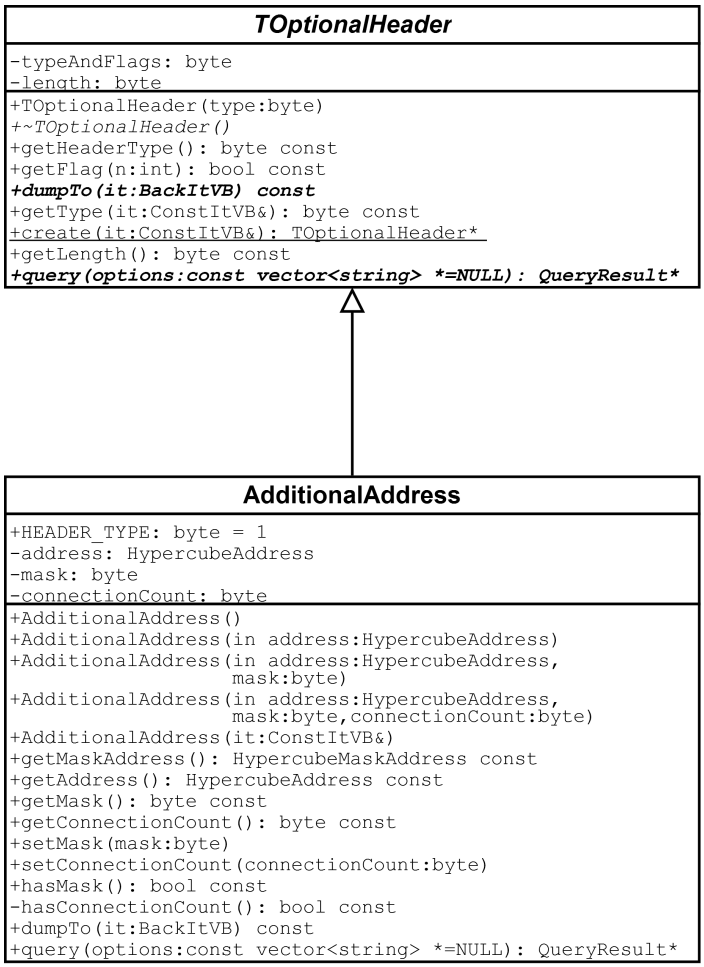


Figura 3.34: Clases para encabezados opcionales

- `query`: método abstracto que las clases descendientes deben heredar para reportar los campos del encabezado.

La clase `AdditionalAddress` implementa un encabezado opcional que permite enviar direcciones adicionales. Éstas pueden contener adicionalmente una máscara y una cuenta de conexiones, es decir, la cantidad de conexiones que tiene el nodo asociado a esa dirección.

Esta clase provee constructores para crear el encabezado utilizando diferentes parámetros, así como métodos para leer y escribir los atributos.

DataPacket

La capa de red intercambia, además de los paquetes de control vistos anteriormente, paquetes que transportan datos. Para ello, se cuenta con la clase `DataPacket`, que se muestra en la figura 3.35.

Los atributos de esta clase son:

- `ETHERNET_TYPE`: esta constante contiene el identificador para los paquetes de datos, cuyo valor es 1001.
- `MAX_TTL`: esta constante indica cuál es el valor de *Time To Live* asignado a un paquete al ser enviado.
- `totalLength`: longitud total del paquete, incluyendo encabezado y datos, hasta 65536 bytes.
- `flags`: contiene 8 *flags*.

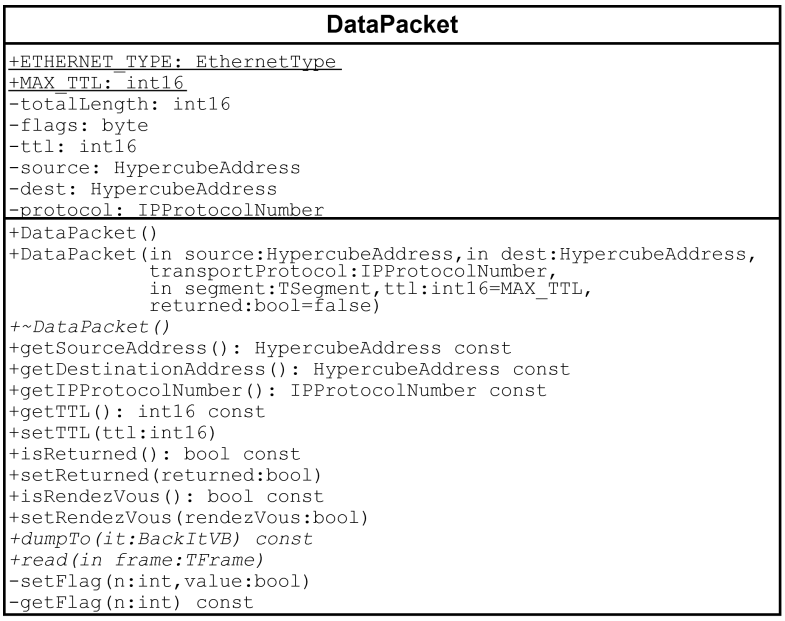


Figura 3.35: Clases `DataPacket`

- `ttl`: *Time To Live*, indica la cantidad de saltos restantes antes de que el paquete se dé por perdido y se elimine.
- `source`: dirección de hipercubo de origen del paquete.
- `dest`:dirección de hipercubo de destino del paquete.
- `protocol`: protocolo de transporte que debe recibir al paquete.

Esta clase provee dos constructores; uno que no lleva parámetros y que no realiza inicializaciones, y el otro que permite asignarle valor a todos los atributos.

Los métodos de esta clase permiten leer y escribir los valores de los atributos. El método `dumpTo` copia el encabezado y los datos a un iterador. El método `read` lee los datos de un *frame* y construye el paquete de datos incluyendo su encabezado.

UDPSegment

En el nivel de capa de transporte, se utiliza como base a `TSegment`, con la implementación `UDPSegment` (ver figura 3.36), que representa un segmento UDP.

La clase `UDPSegment` contiene los atributos `sourcePort` y `destinationPort`, representando los puertos de origen y destino respectivamente. Los puertos se utilizan para determinar cuál es la aplicación a cargo.

El constructor que toma como parámetro un paquete construye un segmento UDP leyendo sus datos. El otro constructor asigna los valores a los atributos.

Los métodos `getSourcePort` y `getDestinationPort` devuelven los puertos de origen y destino del segmento respectivamente.

Data

En la capa de aplicación, la clase base para el intercambio de datos es `TData` (ver figura 3.37).

La clase `Data` es una implementación general que hereda de `TData` y permite trabajar con los datos como un vector de bytes o como una cadena de caracteres.

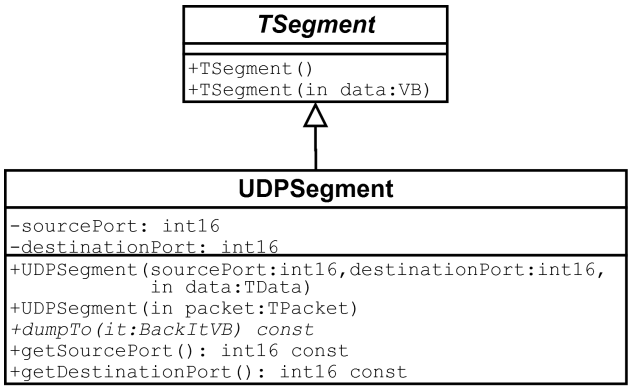


Figura 3.36: clases `TSegment` y `UDPSegment`

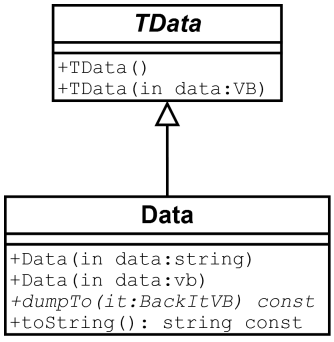


Figura 3.37: clases `TData` y `Data`

Datos de *Rendez-Vous*

Las aplicaciones *Rendez-Vous Client* y *Rendez-Vous Server* se comunican entre ellas utilizando un formato específico de datos. Este formato se explica en la sección 2.4.3, y las clases que representan los datos se muestran en la figura 3.38.

La clase `TRendezVousPacket` es la clase base para los datos de *Rendez-Vous*. El atributo `typeAndFlags` guarda el tipo en 5 bits y contiene 3 flags.

Los métodos de la clase realizan lo siguiente:

- `create`: este método estático crea una instancia de datos de *Rendez-Vous* a partir de un vector de bytes.
- `dumpTo`: las clases descendientes deben implementar este método para volcar los datos a un iterador.
- `getData`: utiliza al método `dumpTo` para obtener un volcado de los datos y los devuelve en un vector de bytes.
- `setFlag`: da valor al *flag* especificado.
- `getFlag`: devuelve el valor del *flag* especificado.

Las clases descendientes son todas similares. Todas ellas contienen una constante `TYPE` que representa su tipo.

Los constructores que reciben como parámetro `data` interpretan estos datos y almacenan los valores en sus atributos. Los constructores que contienen otros parámetros los guardan en sus atributos.

El método `dumpTo` se sobreescribe para copiar los atributos al iterador pasado como parámetro.

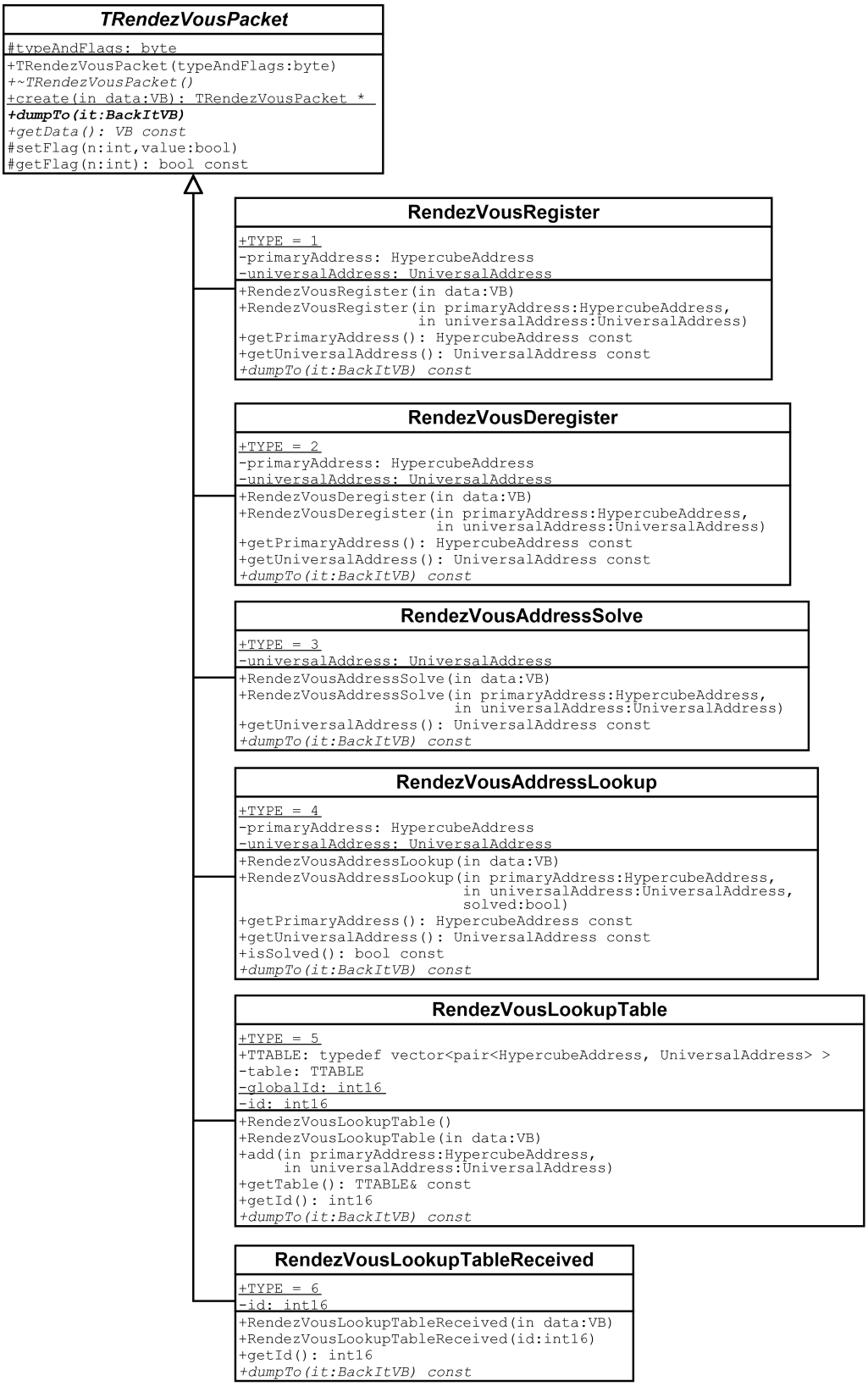


Figura 3.38: clases para los datos de *Rendez-Vous*

3.2.4. Direcciones

En el simulador se utilizan distintos tipos de direcciones para las distintas capas de protocolo.

En el namespace `simulator::address` (ver figura 3.39) se encuentran las clases que representan las direcciones.

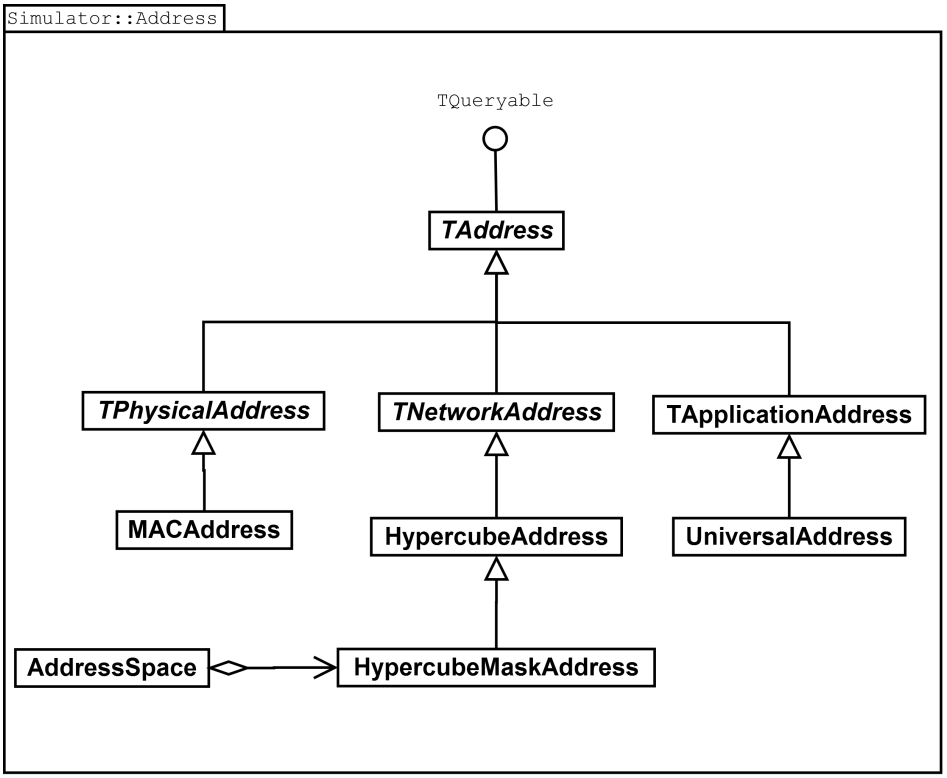


Figura 3.39: Diagrama de clases de direcciones

La clase abstracta `TAddress` (ver figura 3.40) es la base de todas las clases de direcciones. Para ello cuenta con un vector de bytes donde se almacena la dirección propiamente dicha, aunque dentro de esta clase no se tengan detalles del formato de la dirección.

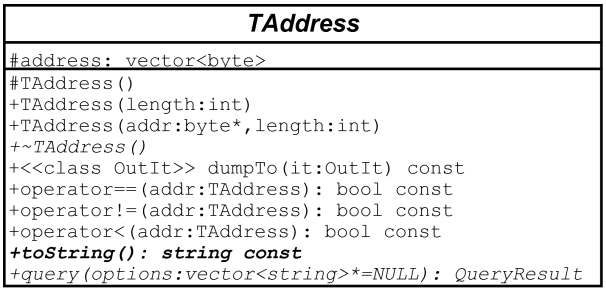


Figura 3.40: clase `TAddress`

El constructor que no lleva parámetros tiene visibilidad *protected* para que sólo pueda ser utilizado por las clases descendientes. En este caso, el atributo `address` queda con un vector sin elementos.

El constructor que toma como parámetro solamente `length`, crea una dirección con todos los bytes en 0 y de la longitud especificada, mientras que el otro constructor toma un puntero a *byte*, que es utilizado como un *array* con tantos elementos como el parámetro `length` indique.

Se sobrecargan tres operadores de comparación: "`==`", "`!=`" y "`<`", con el fin de poder utilizar descendientes de esta clase como claves en estructuras que requieren que los elementos sean comparables, como por ejemplo en un mapa o conjunto.

Los otros métodos de la clase cumplen con la siguiente funcionalidad:

- `dumpTo`: copia el vector `address` en el iterador de salida.
- `toString`: este método abstracto debe ser implementado por las clases descendientes para devolver una representación de la dirección como cadena de caracteres en un formato legible por el usuario.
- `query`: es llamado cuando se hace una consulta sobre la dirección; utiliza el método `toString` para generar la salida.

De esta clase abstracta heredean otras tres clases, también abstractas, cuyo fin es clasificar las direcciones según la capa que las utiliza: `TPhysicalAddress`, `TNetworkAddress` y `TApplicationAddress`. Esto permite que las implementaciones de las capas no dependan de un tipo particular de direcciones.

Clases para direcciones físicas

Las direcciones físicas se utilizan en la capa de red y en la de enlace de datos. La clase base para direcciones físicas es `TPhysicalAddress`, y se provee la implementación `MACAddress`, como muestra la figura 3.41.

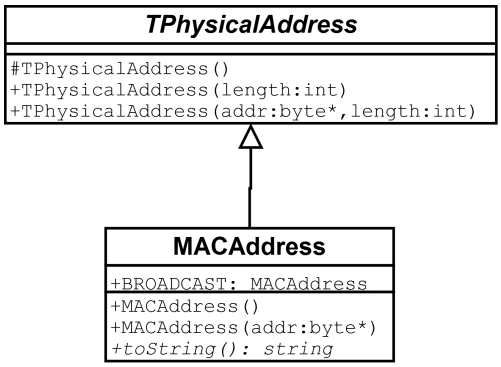


Figura 3.41: Clases `TPhysicalAddress` y `MACAddress`

La clase `TPhysicalAddress` tan sólo provee los mismos constructores que su clase padre, `TAddress`.

`MACAddress` es una implementación de direcciones MAC (*Media Access Control*), utilizadas comunmente en *Ethernet*.
Estas direcciones tienen una longitud de 48 bits. El formato de escritura para ser leído por usuarios consiste en los bytes escritos en hexadecimal separados por dos puntos, por ejemplo "23:B3:AE:C0:15:56".

Los constructores de esta clase omiten el parámetro `length` ya que es constante. El método `toString` devuelve la representación de la dirección en el formato descripto.

Clases para direcciones de red

Las direcciones de red se utilizan en la capa de red y en la de transporte. La figura 3.42 muestra las clases para estas direcciones.

La clase `TNetworkAddress` es la clase base para direcciones de red, que solamente provee los constructores de su clase padre.

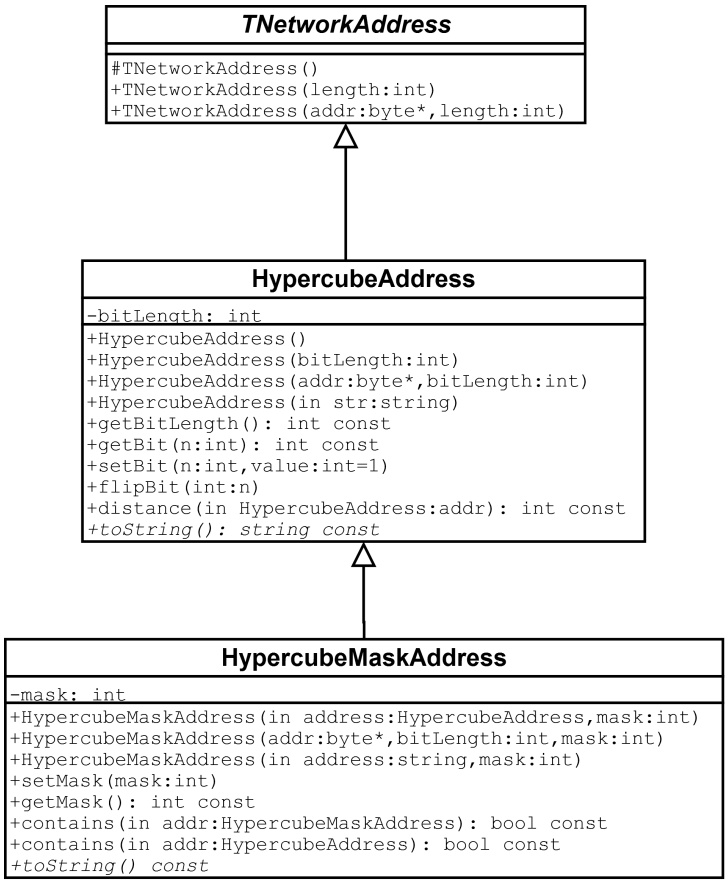


Figura 3.42: Clases `TNetworkAddress`, `HypercubeAddress` y `HypercubeMaskAddress`

La clase HypercubeAddress representa una dirección de hipercubo. Dado que la longitud de éstas se especifica en bits y no necesariamente es un múltiplo de 8, no es suficiente con conocer el tamaño del vector `address` de la clase base para saber la longitud de la dirección de hipercubo. Por eso se utiliza el atributo `bitLength`, que almacena la longitud en bits.

Asimismo, los constructores requieren la longitud en bits. Se provee también un constructor que toma como parámetro una cadena de caracteres conteniendo una representación de la dirección compuesta por "0" y "1".

Los métodos de la clase cumplen con la siguiente funcionalidad:

- `getBitLength`: devuelve la longitud en bits de la dirección de hipercubo.
- `getBit`: devuelve el valor del bit especificado.
- `setBit`: asigna un valor al bit especificado.
- `flipBit`: cambia el valor (de "0" a "1" y viceversa) del bit especificado.
- `distance`: calcula la distancia con respecto a la dirección de hipercubo pasada, siendo ésta la cantidad de bits en que ambas direcciones difieren.
- `toString`: devuelve una representación de la dirección en una cadena de caracteres utilizando "0" y "1".

Dado que es frecuente utilizar las direcciones de hipercubo con máscaras se provee la clase `HypercubeMaskAddress` que extiende la dirección de hipercubo con este fin. Esta clase está representando

en realidad un espacio de direcciones que tienen sus primeros m bits en común (siendo m la máscara) y los restantes bits toman todas las combinaciones posibles. Por ejemplo, la dirección de hipercubo "101000/4" (dónde 4 es la máscara) representa las direcciones "101000", "101001", "101010" y "101011". En este caso, los primeros 4 bits están fijos mientras que los últimos 2 pueden tomar cualquier valor.

Los constructores de esta clase son similares a los de la clase padre, `HypercubeAddress`, pero agregan un parámetro más para representar la máscara.

Los métodos de la clase son:

- `setMask`: asigna el valor de la máscara.
- `getMask`: devuelve el valor de la máscara.
- `contains`: ambas versiones de este método devuelven verdadero si la dirección o el espacio de direcciones pasado como parámetro está contenido dentro del espacio de direcciones representado por la instancia.
- `toString`: devuelve la representación de esta dirección, agregándole la máscara a la dirección de hipercubo, por ejemplo "10100101/3".

Si bien esta clase permite representar un espacio de direcciones, son acotados los espacios que se pueden representar con ella, ya que los primeros bits deben ser iguales en todo el espacio.

Para representar espacios de direcciones más complejos, se definió la clase `AddressSpace` (ver figura 3.43).

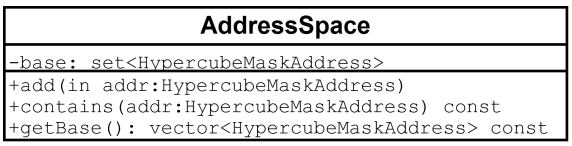


Figura 3.43: Clase `AddressSpace`

Esta clase puede representar cualquier espacio de direcciones, ya que contiene un conjunto de `HypercubeMaskAddress` (atributo `base`).

Al instanciar un objeto de esta clase, el espacio de direcciones se encuentra vacío. El método `add` agrega un espacio de direcciones al ya existente; es decir, hace la unión de ambos espacios. Además, optimiza el almacenamiento utilizando la mínima cantidad posible de direcciones sin alterar el espacio representado.

Para ello, cada vez que se llama a `add` se busca si existe algún subespacio complementario en la base, es decir que tengan la misma máscara y que coincidan en todos los bits de la máscara menos en el último, que debe ser diferente. Por ejemplo, si se agrega "101000/3", se busca "100000/3" en la base. Si se encuentra, como la unión de estos dos espacios es equivalente a "100000/2", se elimina el vector que se encontró en la base y se agrega este último en forma recursiva, de tal manera que si se encontrase un complemento de este se procedería de la misma forma.

El método `contains` devuelve verdadero si el espacio de direcciones pasado como parámetro está contenido en la instancia. Esto ocurre sólo en el caso de que esté contenido en algún elemento de la base.

Por último, `getBase` devuelve un vector con todos los elementos de la base.

Clases para direcciones de aplicación

Las direcciones de aplicación son utilizadas en la capa del mismo nombre. Las clases para estas direcciones se muestran en la figura 3.44.

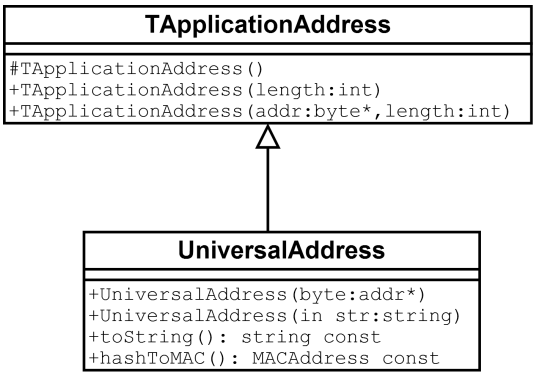


Figura 3.44: Clases `TApplicationAddress` y `UniversalAddress`

La clase `TApplicationAddress` es la base para este tipo de direcciones, que solamente provee los mismos constructores que la clase padre.

Se implementan direcciones universales con la clase `UniversalAddress`. Estas direcciones son cadenas de caracteres de cualquier longitud.

El constructor que toma como parámetro un puntero a byte crea direcciones de 6 bytes con estos elementos. El otro constructor asigna la cadena de caracteres a la dirección.

El método `toString` devuelve la cadena de caracteres de la dirección.

Por último, `hashToMAC` computa una función de *hashing* en la dirección y la adapta a una dirección MAC. Esto se utiliza para generar direcciones MAC automáticamente para los nodos a partir de la dirección universal, evitando que el usuario deba proveer esta dirección. Esto es utilizado únicamente para las simulaciones, ya que en una implementación real cada nodo tendrá una dirección MAC y no se necesitará asignar este valor.

3.2.5. Ruteo

En este módulo se implementa el algoritmo de ruteo reactivo descripto en la sección 2.3. En la figura 3.45 se muestran las clases que lo componen.

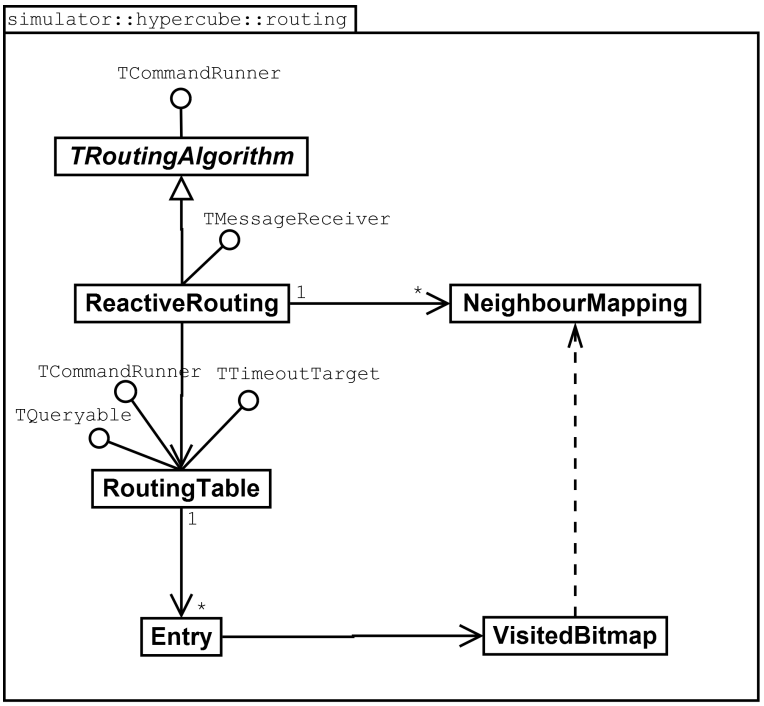


Figura 3.45: Diagrama de clases de las Capas de Ruteo

La clase `TRoutingAlgorithm` (figura 3.46) es la interface que debe presentar cualquier algoritmo de ruteo que se implemente.

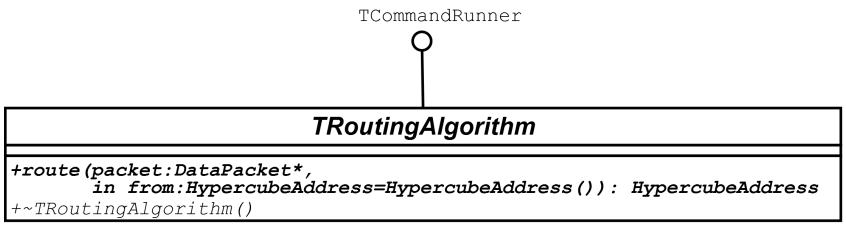


Figura 3.46: Clase `TRoutingAlgorithm`

El método abstracto `route` debe ser redefinido por las clases que lo hereden. Tiene como parámetro el paquete recibido y la dirección del nodo de donde provino. Debe devolver la dirección de hipercubo del próximo nodo.

La clase `ReactiveRouting` (figura 3.47) es la implementación provista para el ruteo reactivo (ver algoritmos 7 y 8).

Esta clase presenta los siguientes atributos:

- `node`: puntero al nodo dónde se está ejecutando este algoritmo.
- `routingTable`: tabla de ruteo del nodo.
- `mapping`: se utiliza para asociar los mapas de bits de vecinos visitados con sus direcciones.

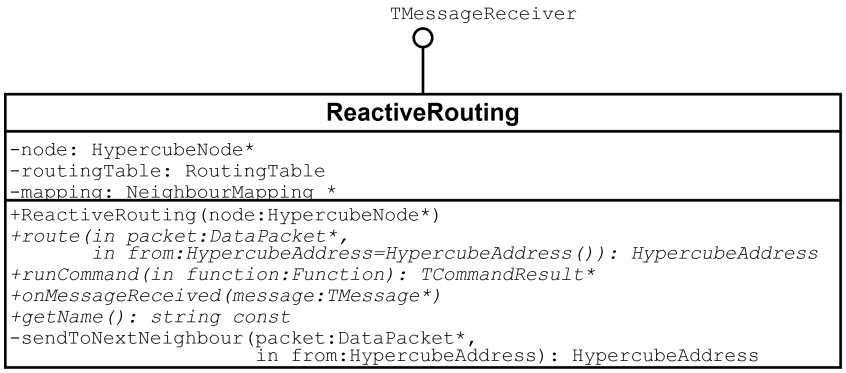


Figura 3.47: Clase `ReactiveRouting`

El constructor de la clase se registra en el nodo para recibir mensajes de tipo *Connected*, *NewRoute* y *LostRoute*. De esta forma, la clase es notificada cuando hay cambios en las conexiones con los vecinos.

Los métodos de la clase cumplen con las siguientes funciones:

- `route`: implementa el algoritmo de ruteo decripto en la sección 2.3.2. (ver algoritmo 7)
- `runCommand`: ejecuta el comando `table`, que devuelve la tabla de ruteo para que se le realicen consultas.
- `onMessageReceived`: refleja los cambios en las conexiones con los vecinos en el atributo `mapping`.
- `getName`: devuelve el nombre del objeto ("`ReactiveRouting`").
- `sendToNextNeighbour`: método auxiliar para la implementación del algoritmo de ruteo decripto en la sección 2.3.2. (ver algoritmo 8)

`ReactiveRouting` utiliza varias clases auxiliares. La tabla de ruteo se implementa en la clase `RoutingTable`, estando compuesta por entradas, representadas cada una por un objeto de tipo `Entry`. Estas dos clases se muestran en la figura 3.48.

Los atributos de la clase `RoutingTable` son:

- `entries`: almacena las entradas de la tabla de ruteo, guardadas en un mapa con la dirección de destino como clave para un acceso más rápido.
- `timeouts`: asocia los identificadores de *timeout* con la entrada que debe ser limpiada. El valor *bool* almacenado es para distinguir si se debe eliminar la entrada o sólo el *bitmap* de vecinos visitados.
- `nextTimeoutId`: valor que se utilizará para identificar el próximo *timeout*.
- `node`: puntero al nodo donde se encuentra esta tabla de ruteo.
- `pairs`: asocia un par de direcciones (origen;destino) con sus respectivas entradas.

La clase cuenta con los siguientes métodos:

- `add`: agrega la entrada a la tabla de ruteo.
- `getEntries`: devuelve todas las entradas de la tabla de ruteo cuyo destino es el especificado.
- `replace`: reemplaza una entrada por otra.
- `remove`: borra una entrada.
- `runCommand`: permite ejecutar el comando *query*, que devuelve información sobre la tabla de ruteo y sus entradas.

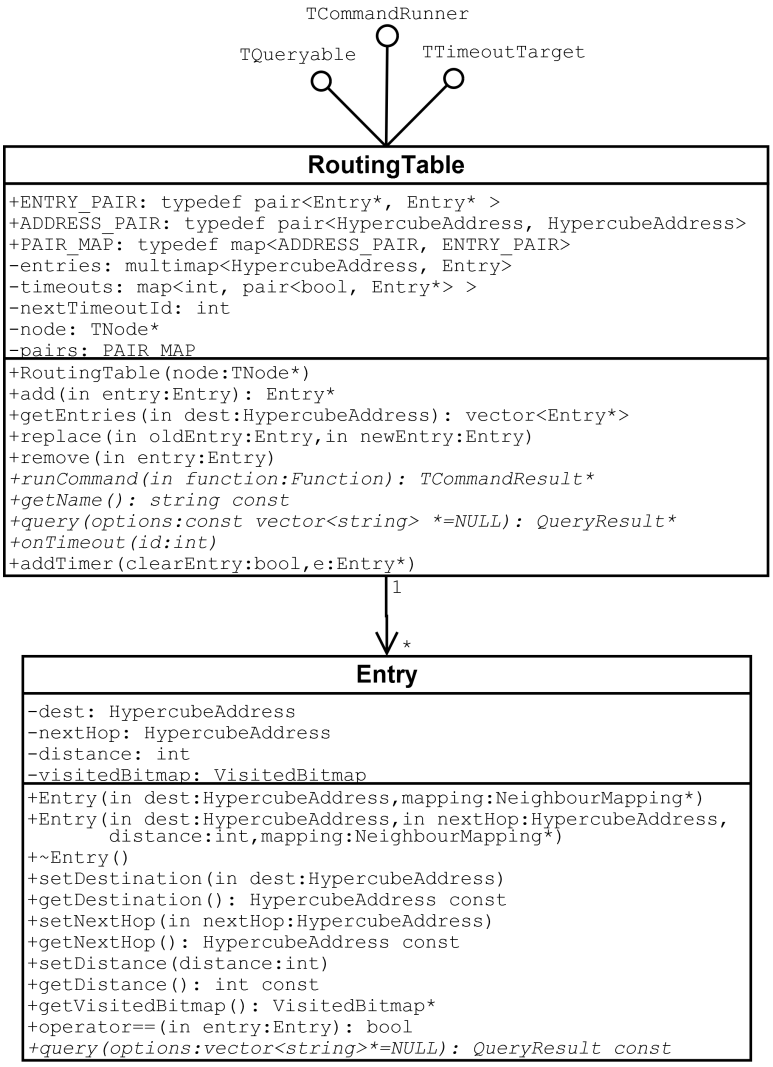


Figura 3.48: Clases `RoutingTable` y `Entry`

- `getName`: devuelve el nombre del objeto ("`RoutingTable`").
- `query`: efectúa una consulta sobre la tabla de ruteo y sus entradas.
- `onTimeout`: elimina una entrada o su *bitmap* de vecinos visitados.
- `addTimer`: agrega un *timeout* para una entrada.

Cada una de las entradas es representada por una instancia de la clase `Entry`, cuyos atributos son:

- `dest`: destino final de los paquetes.
- `nextHop`: vecino al que son mandados los paquetes para rutearlos al destino.
- `distance`: distancia al destino. Si no es conocida se utiliza la constante `MAX_TTL`.
- `visitedBitmap`: indica cuales de los vecinos fueron probados como rutas para el destino.

Los constructores de la clase permiten inicializar los atributos, que luego pueden ser cambiados o leídos a través de los métodos.

Se provee además el método `query` que devuelve un `QueryResult` con los datos de la entrada.

Se sobrecargó el operador de comparación "==" para permitir comparar objetos de este tipo.

Para saber cuales son los vecinos visitados, se utilizó un objeto de tipo `VisitedBitmap`, en conjunción con `NeighbourMapping`, que se muestran en la figura 3.49.

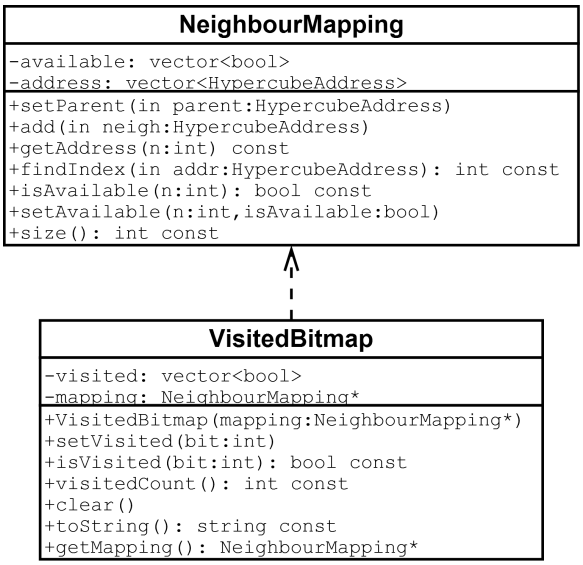


Figura 3.49: Clases `NeighbourMapping` y `VisitedBitmap`

Los vecinos se visitan en el orden descripto en la sección 2.3.1.

La clase `VisitedBitmap` contiene un vector de *bool* en el atributo `visited`, que indica cuál vecino fue visitado y cuál no. El atributo `mapping` contiene un puntero a un objeto `NeighbourMapping`, que permite asociar cada uno de los bits del vector con la dirección de un vecino.

El constructor inicializa el atributo `mapping` y marca todos los vecinos como no visitados.
Los métodos cumplen con las siguientes funciones:

- `setVisited`: marca al vecino especificado como visitado.
- `isVisited`: devuelve verdadero si el vecino especificado está marcado como visitado.
- `visitedCount`: devuelve cuantos nodos fueron visitados.
- `clear`: marca todos los nodos como no visitados.
- `toString`: devuelve una representación del objeto como cadena de caracteres, utilizando "0" y "1" para marcar los nodos no visitados y visitados respectivamente.
- `getMapping`: devuelve el atributo `mapping`.

La clase `NeighbourMapping` permite relacionar al *bitmap* de vecinos con sus direcciones, y almacena además cuales de los vecinos están disponibles, es decir, siguen contectados. El atributo `available` guarda esta información utilizando *true* cuando el vecino está disponible. Las direcciones de los vecinos se guardan en el atributo `address`, coincidiendo la posición de cada uno en el vector con las del atributo `available` del mismo objeto y `visited` de los `VisitedBitmap` asociados.

Los métodos de la clase son:

- `setParent`: da valor a la dirección del nodo padre, que se ubica en la primera posición del vector.

- `add`: agrega un vecino.
- `getAddress`: devuelve la dirección del vecino especificado.
- `findIndex`: devuelve la posición dentro del vector correspondiente al vecino con la dirección especificada.
- `isAvailable`: devuelve verdadero si el vecino especificado está disponible.
- `setAvailabe`: marca al vecino especificado como disponible o no disponible según el valor del parámetro `isAvailable`.
- `size`: devuelve la cantidad de vecinos del mapa.

3.2.6. Mensajes

Las distintas capas del protocolo intentan ser lo más independientes posible; sin embargo, lo que ocurre en un nivel puede afectar otros niveles. Por ejemplo, cuando un nodo se conecta a nivel de capa de red, la aplicación de *Rendez-Vous* debe iniciar la registración.

La forma trivial de resolver esto sería conocer las dependencias y llamar a los métodos correspondientes, pero esto dificultaría mucho cualquier cambio, ya que por ejemplo si por alguna razón se cambia la aplicación de *Rendez-Vous*, se debe cambiar la capa de red.

Para resolver esto de una forma más flexible, se utilizan mensajes internos del nodo. Cuando una capa quiere comunicar algo, pone un mensaje en una cola, y todos aquellos procesos que se encuentren registrados para escuchar el mensaje son notificados.

Las clases que quieran ser notifcadas de algún mensaje deben heredar de `TMessageReceiver` (ver figura 3.51) e implementar el método `onMessageReceived`. Además, deben llamar al método `registerMessageListener` en el nodo donde se encuentran, pasándole el tipo de mensaje al que se quiere registrar.

Cuando se desea poner un mensaje en la cola, se debe llamar al método `putMessage` del nodo, y la clase `TNode` se encargará de transmitírselo a todos los objetos que se hayan registrado.

En la figra 3.50 se muestra un diagrama con las clases que componen al namespace *simulator::message*.

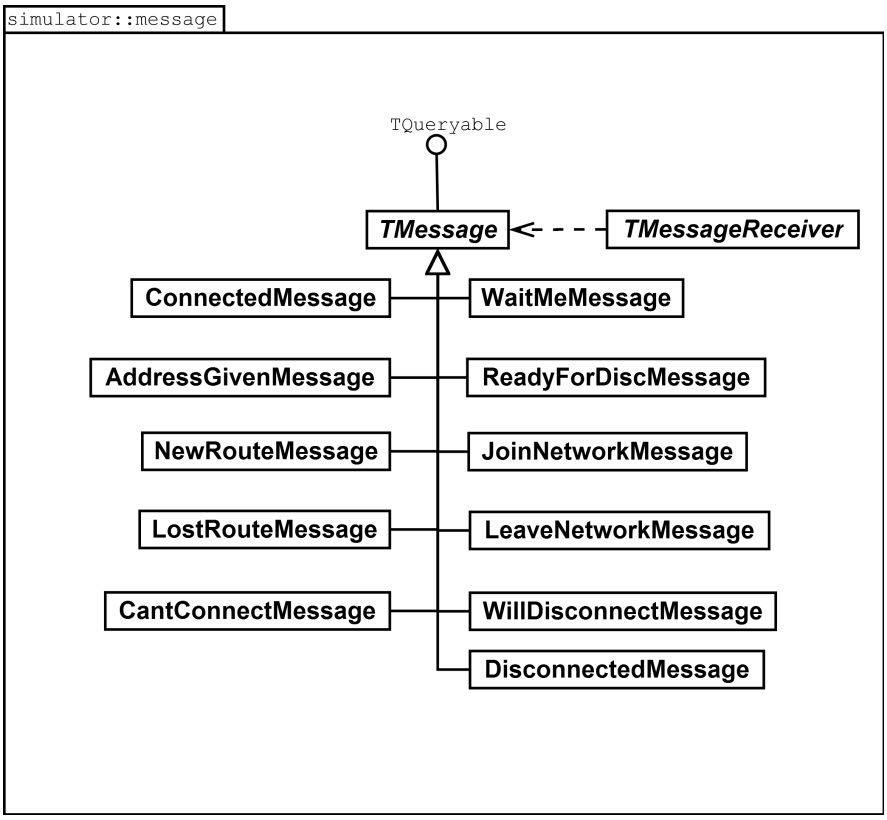


Figura 3.50: Diagrama de clases de los Mensajes

La clase `TMessage` (ver figura 3.51) es la clase base para todos los mensajes. Implementa un sistema de conteo de referencias al mensaje; de esta forma, un solo mensaje es creado y pasado a todos los receptores, y una vez que todos lo recibieron, se borra de la memoria automáticamente. El atributo `useCount` es el contador de referencia, y los métodos `incUseCount`, `decUseCount` y `getUseCount` permiten incrementarlo, decrementarlo y obtener su valor respectivamente.

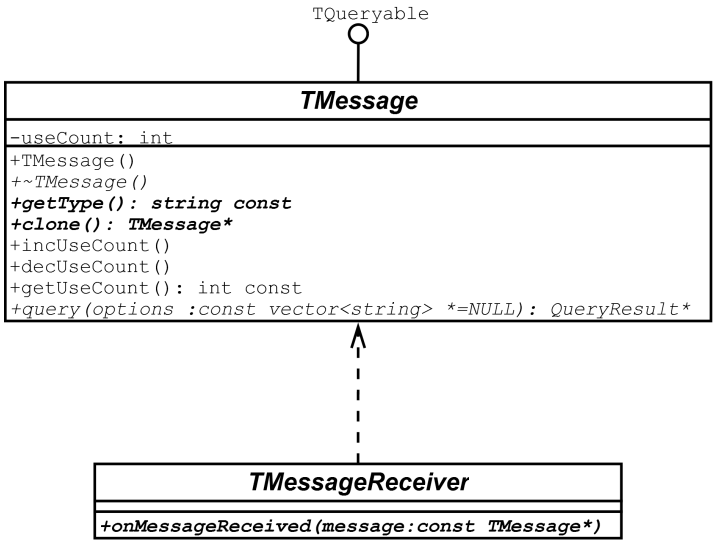


Figura 3.51: Clases `TMessageReceiver` y `TMessage`

La implementación en las clases descendientes del método abstracto `getType` debe devolver una cadena de caracteres identificando al tipo de mensaje. El método `clone` debe devolver una copia del mensaje.

Por último, el método `query` se debe sobreescribir para que devuelva un objeto `QueryResult` con información del mensaje.

Todas las clases descendientes de `TMessage` tienen una estructura similar. Utilizan la constante `TYPE` para almacenar el nombre que identifica al tipo de mensaje, que es devuelto por el método `getType`. El método `clone` se implementa creando una nueva instancia del objeto que contenga los mismos atributos.

Algunos mensajes guardan información en sus atributos, que es inicializada en el constructor y se puede obtener a través de los métodos de la clase.

A continuación, se explican cada uno de los tipos de mensaje.

El mensaje *JoinNetwork* (figura 3.52) es utilizado cuando el usuario desea conectarse a la red. Esto ocurre cuando se encuentra el comando *joinNetwork* en el archivo de simulación.

Para dar la orden de desconexión, se utiliza el mensaje *LeaveNetwork*, que es lanzado con el comando *leaveNetwork* en el archivo de simulación.

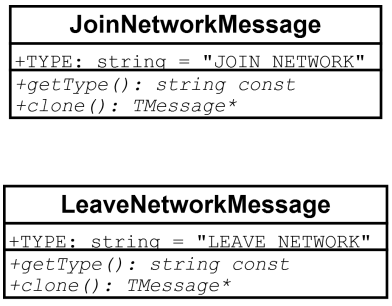


Figura 3.52: Clases `JoinNetworkMessage` y `LeaveNetworkMessage`

Una vez dada la orden de conexión, se intentará obtener una dirección primaria. Cuando esto se logra, se envía el mensaje *Connected* (figura 3.53) para que otras capas puedan efectuar sus tareas de inicialización al conectarse. Si por el contrario no se pudo conectar, por ejemplo porque no hay direcciones disponibles, entonces el mensaje enviado es *CantConnect*.

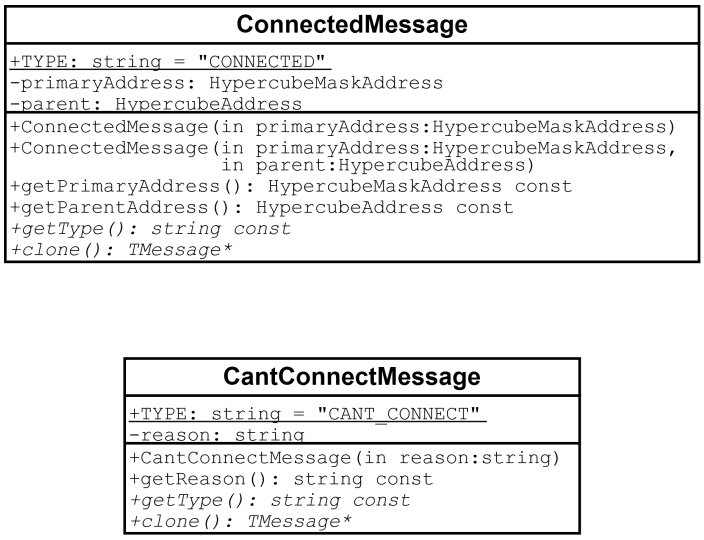


Figura 3.53: Clases `ConnectedMessage` y `CantConnectMessage`

La clase `ConnectedMessage` tiene los atributos `primaryAddress` y `parent`. El primero es la dirección primaria del nodo, y el segundo es la dirección del padre, es decir, el vecino que se la cedió.

La clase `CantConnectMessage` utiliza el atributo `reason` para almacenar una cadena de caracteres donde figura la razón por la cual no se pudo conectar a la red.

Cuando un nodo cede parte de su espacio de direcciones, envía un mensaje *AddressGiven* (figura 3.54).

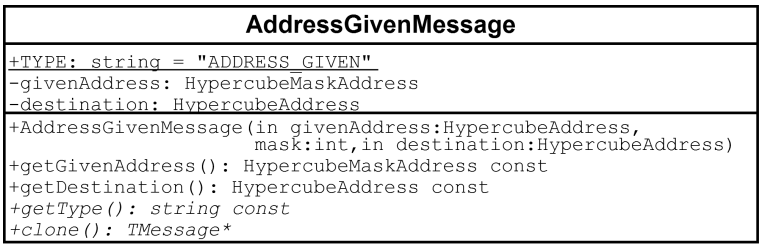


Figura 3.54: Clase `AddressGivenMessage`

Esta clase guarda en el atributo `givenAddress` el espacio de direcciones cedido, y en `destination` la dirección del nodo al que se le cedió.

Cuando un nodo tiene nuevas conexiones, o pierde alguna, avisa de estos hechos utilizando los mensajes *NewRoute* y *LostRoute* respectivamente, cuyas clases se muestran en la figura 3.55.

Ambas clases utilizan el atributo `route` para almacenar la ruta en cuestión.

Para la desconexión del nodo es necesario que todas las capas hagan sus tareas de limpieza. Es por ello que una vez que el usuario da la orden de desconexión a la red mediante el mensaje *LeaveNetwork*,

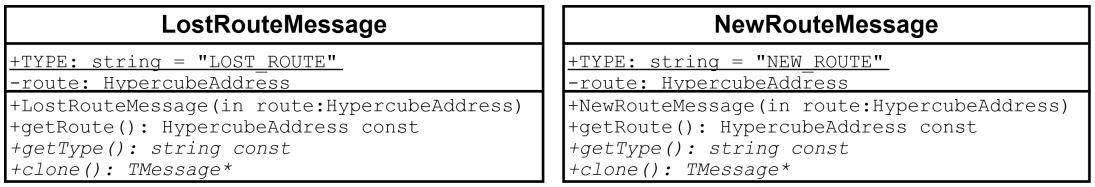


Figura 3.55: Clases `NewRouteMessage` y `LostRouteMessage`

no se procede inmediatamente, sino que primero se envía un mensaje *WillDisconnect* avisando que se desea desconectar. Todos los procesos que necesiten efectuar una tarea de limpieza, deberán responder lo antes posible con un mensaje *WaitMe*, utilizando un número de identificación único. Una vez que el proceso finaliza sus tareas de limpieza, debe enviar el mensaje *ReadyForDisc* utilizando el mismo número de identificación. Cuando todos los procesos que pidieron ser esperados mandaron este último mensaje, se procede a la desconexión y se envía el mensaje *Disconnected*. Las clases que representan estos mensajes se muestran en la figura 3.56

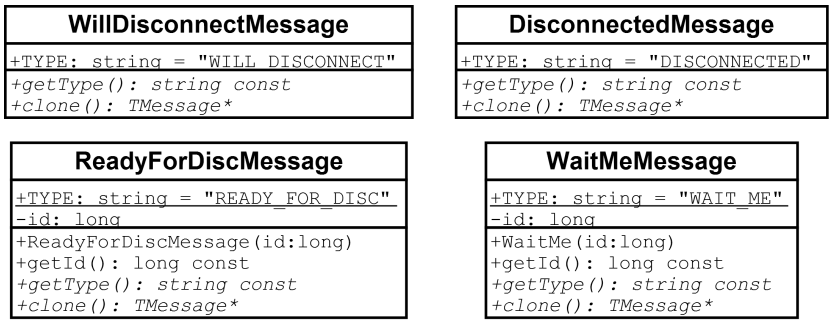


Figura 3.56: Clases para desconexión

Las clases `WaitMeMessage` y `ReadyForDiscMessage` utilizan el atributo `id` para identificar el proceso que solicita la espera y luego la da por finalizada.

3.2.7. Eventos

La simulación se realiza ejecutando eventos, que son agendados y ejecutados en orden cronológico. El namespace `simulator::event` contiene las clases que representan los distintos eventos que se ejecutan. Estas clases (ver figura 3.57) no contienen mucha funcionalidad, ya que no son más que intermediarios entre el simulador y otras clases.

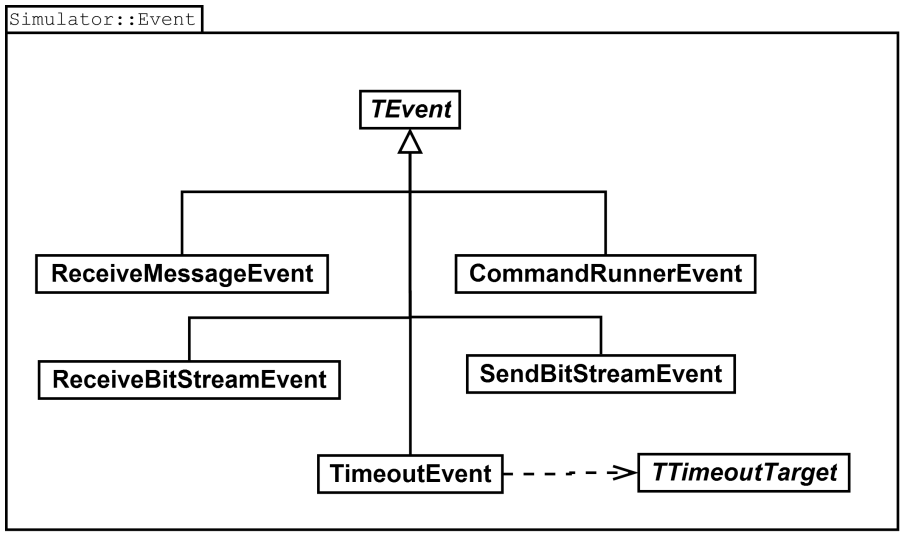


Figura 3.57: Diagrama de clases de simulator::event

La clase base para todos los eventos es `TEvent` (ver figura 3.58), que contiene la funcionalidad común a todos ellos.

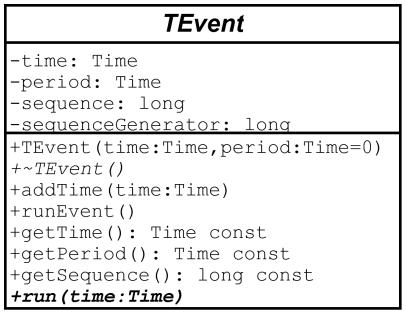


Figura 3.58: clase `TEvent`

Los atributos de esta clase son los siguientes:

- `time`: tiempo en el que se ejecutará el evento.
- `period`: período de repetición del evento. Si es cero o negativo, el evento se ejecuta una vez.
- `sequence`: número de secuencia utilizado para ordenar eventos que se ejecutan en el mismo tiempo. El que se crea primero se ejecuta primero.
- `sequenceGenerator`: próximo número de secuencia a utilizar.

El constructor requiere el tiempo de ejecución, y opcionalmente el período, que en caso de omitirse indica que se debe ejecutar una vez. El número de secuencia se asigna automáticamente.

Los métodos de esta clase son:

- `addTime`: suma el intervalo especificado al tiempo de ejecución.
- `runEvent`: este método es llamado por el simulador para ejecutar el evento. Llama al método `run` que debe ser implementado por las clases descendientes, y vuelve a agendar el evento si es periódico.
- `getTime`: devuelve el tiempo de ejecución del evento.
- `getPeriod`: devuelve el período de ejecución del evento.
- `getSequence`: devuelve el número de secuencia que se le asigno al evento.
- `run`: este método abstracto se debe implementar en las clases descendientes para ejecutar la acción correspondiente al evento.

Se detallarán ahora cada uno de los eventos.

ReceiveMessageEvent

Este evento sirve para el envío de mensajes dentro del nodo. Esto crea un sistema de colas que permite que distintas capas u objetos se comuniquen fácilmente. Por ejemplo, cuando un nodo se conecta, se envía un mensaje `CONNECTED`, y quienes estén suscriptos a este tipo de mensaje, recibirán el mensaje a través de este evento. La clase que implementa este evento se muestra en la figura 3.59.

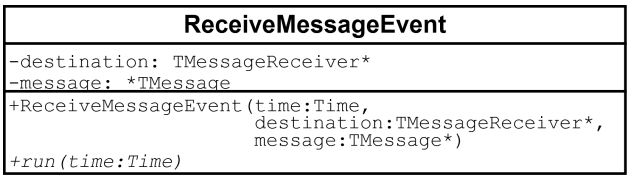


Figura 3.59: clase `ReceiveMessageEvent`

El constructor de la clase toma como parámetros, además del tiempo de ejecución, el objeto que debe recibir el mensaje (debiendo éste heredar de `TMessageReceiver`) y el mensaje a transmitir. Ambos parámetros se almacenan en los atributos con el mismo nombre.

Cuando el evento es ejecutado, se llama al método `onMessageReceived` del objeto `destination` usando `message` como parámetro. Además, se decrementa la cuenta de usos del mensaje, y cuando llega a cero, se devuelve la memoria del mensaje.

CommandRunnerEvent

Este evento permite agendar un comando para que se ejecute en un tiempo determinado. Por ejemplo, cuando se lee el archivo de entrada del simulador, los comandos son agendados mediante esta clase, que se muestra en la figura 3.60.

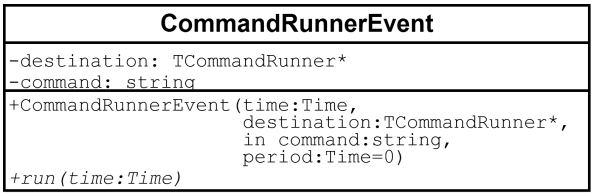


Figura 3.60: clase `CommandRunnerEvent`

El constructor de la clase toma como parámetros, además del tiempo de ejecución y el período, el objeto sobre el que se debe ejecutar el comando y una cadena de caracteres conteniendo el comando.

Ambos parámetros se almacenan en los atributos con el mismo nombre.

Cuando se ejecuta el evento, se utiliza el método `exec` de la única instancia de `Simulator` pasándole como parámetros el destino y el comando.

SendBitStreamEvent

Mediante este evento se envían *bit streams* (es decir, datos en crudo), desde la capa física a la conexión. Es necesario utilizar un evento para introducir una espera, la cual simula el retraso debido al ancho de banda finito. La clase para este evento se muestra en la figura 3.61.

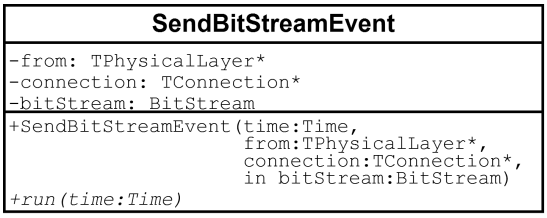


Figura 3.61: clase `SendBitStreamEvent`

En los parámetros del constuctor se incluye un puntero a la capa física del nodo que está enviando el *bit stream*, un puntero a la conexión utilizada y los datos a enviar. Estos parámetros se guardan en los atributos del objeto.

Cuando se ejecuta el evento, se llama al método `transport` en el objeto `connection`, pasándole como parámetros la capa física del nodo de origen y los datos a enviar.

ReceiveBitStreamEvent

Cuando una conexión recibe un *bit stream* (mediante el evento `SendBitStreamEvent`), lo envía al otro extremo utilizando el evento `ReceiveBitStreamEvent`, el cual permite simular la espera debido al tiempo de propagación de la conexión. La clase para este evento se muestra en la figura 3.62.

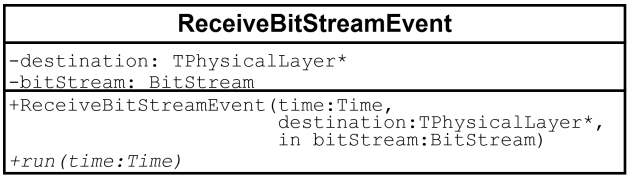


Figura 3.62: clase `ReceiveBitStreamEvent`

El constructor requiere una capa física de destino y el *bit stream* a enviar. Estos datos se guardan en los atributos del objeto.

Cuando el evento es ejecutado, se llama al método `receive` en el objeto `destination`, pasándole como parámetro el *bit stream*.

TimeoutEvent

Algunos objetos necesitan un tiempo de expiración (*timeout*) como por ejemplo, cuando se está esperando una confirmación de un paquete, si luego de un tiempo determinado no llega, se deben efectuar ciertas acciones.

Estos tiempos de expiración son implementados con la clase `TimeoutEvent`, que se muestra en la figura 3.63.

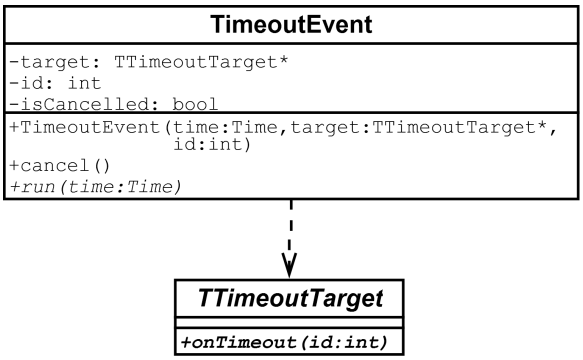


Figura 3.63: clases `TimeoutEvent` y `TTimeoutTarget`

El constructor requiere un puntero al destino del *timeout*, que debe ser una instancia de una clase descendiente de `TTimeoutTarget`, y un número de identificación que permite que un objeto pida más de un *timeout* y luego pueda saber cuál de ellos se disparó.

Los *timeouts* pueden ser cancelados. En este caso, cuando el evento es ejecutado no hace nada. Para cancelarlo, se llama al método `cancel`, que pone en verdadero al atributo `isCancelled`.

Cuando se ejecuta el evento, si `isCancelled` es falso se llama al método `onTimeout` del objeto `target`, pasándole como parámetro al identificador.

3.2.8. Sistema de Comandos

El simulador utiliza un lenguaje de comandos interpretado para que el usuario pueda realizar las simulaciones, dando órdenes y obteniendo información del estado.

El namespace `simulator::command` se encarga de interpretar y ejecutar los comandos.

Syntaxis de los comandos

Un comando se compone por una o más funciones, separadas por puntos. Por ejemplo, el siguiente comando:

```
node(a).query
```

está compuesto por las funciones `node(a)` y `query`. A su vez, una función consta de un nombre, y opcionalmente de parámetros, encerrados entre paréntesis y separados por coma. En caso de que la función no requiera parámetros, se pueden omitir los paréntesis. Por ejemplo, algunas funciones son:

```
query
query()
newConnection('a', 'b', 100 kbps)
setAddressLength(10)
```

Las funciones se ejecutan sobre un objeto, y, excepto la última función de un comando, cada una de ellas debe devolver otro objeto donde se ejecutará la siguiente función.

En el ejemplo visto anteriormente, `node(a).query`, la primera función devuelve un objeto de tipo nodo, y sobre este nodo se ejecuta la función `query`.

Esto permite navegar entre los distintos objetos que componen la red.

Una característica de este lenguaje es la posibilidad de una función de devolver un conjunto de objetos, en cuyo caso las siguientes funciones se ejecutan en todos los objetos devueltos.

Por ejemplo, en el comando:

```
allNodes.query
```

la función `allNodes` devuelve todos los nodos que están en la red, y luego la función `query` se ejecuta sobre cada uno de ellos.

También se puede encadenar más de una función que devuelva múltiples objetos:

```
allNodes.allConnections.setBandwidth(1 Mbps)
```

en este caso, para cada uno de los nodos, se obtienen todas sus conexiones, y para cada una de las conexiones, se ejecuta la función, la cual configura el ancho de banda. De esta forma, se pueden configurar los parámetros de todas las conexiones con una sola instrucción.

Las funciones se pueden clasificar según su tarea principal, que puede ser:

1. devolver uno o más objetos.
2. alterar el estado del objeto
3. efectuar una consulta

El primer tipo de funciones se utiliza para encontrar el objeto deseado, como es el caso de `node(a)`.

El segundo tipo de funciones puede configurar un valor, como en el caso de `setBandwidth(1 Mbps)`, o puede darle una instrucción a un objeto, por ejemplo `newNode`, utilizada para crear un nuevo nodo. Estas funciones no necesitan devolver un objeto, sin embargo es recomendable que devuelvan el mismo objeto sobre el que operaron, para poder encadenar funciones:

```
allNodes.allConnections.setBandwidth(1 Mbps).setDelay(10 us)
```

El último tipo de función recoge información de un objeto, en cuyo caso el comando tiene como resultado final esta información, que posiblemente sea formateada y escrita en un archivo por el simulador.

Implementación

En la figura 3.64 se muestra el diagrama de clases de los comandos.

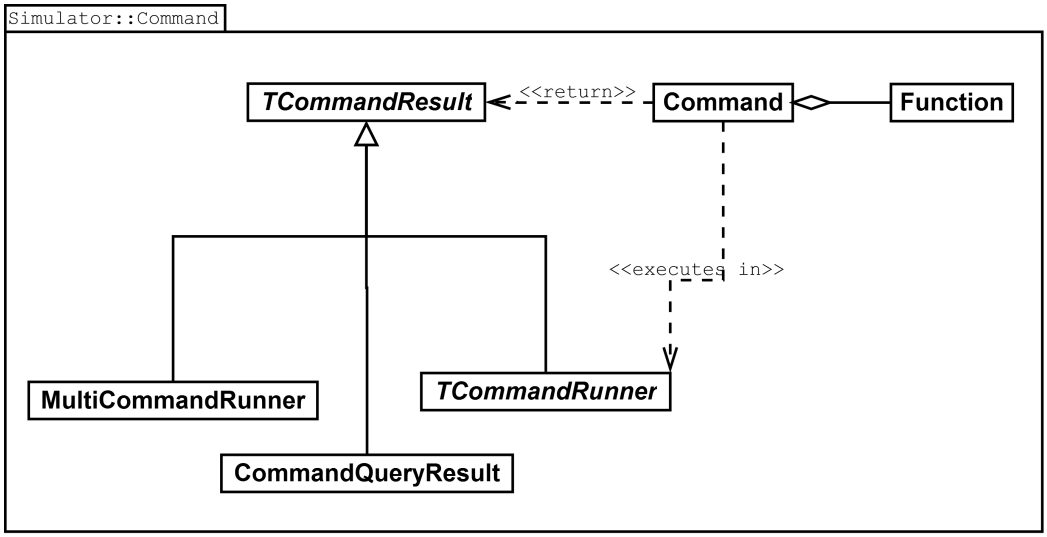


Figura 3.64: Diagrama de clases de los Comandos

Los comandos se representan mediante la clase `Command` (ver figura 3.65), la cual está compuesta por uno o más objetos del tipo `Function`.

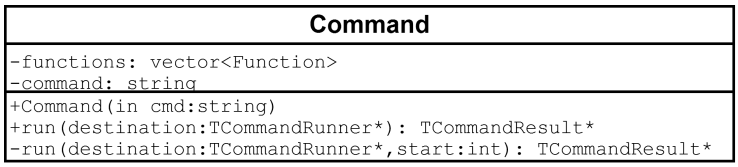


Figura 3.65: Clase Command

Los objetos de la clase `Command` se instancian pasándole el comando como una cadena de caracteres. El objeto se encarga de dividirlo en funciones y crear un objeto de este tipo para cada una de ellas.

Cuando se ejecuta el comando a través de su método `run`, este objeto se encarga de ejecutar cada función en el objeto correspondiente.

La clase `Function` (ver figura 3.66) representa una función, la cual se compone de un nombre y, opcionalmente, parámetros.

El constructor toma como parámetro una cadena de caracteres conteniendo la función, que es analizada y dividida en su nombre y parámetros, que son guardados en los atributos `name` y `params` respectivamente. La cadena de caracteres pasada como parámetro es almacenada en el atributo `originalString`.

El método `getName` devuelve el nombre de la función; `getParamCount` devuelve la cantidad de parámetros que se le pasó a la función y `toString` devuelve la cadena de caracteres pasada originalmente al constructor.

Los métodos `getBoolParam`, `getIntParam`, `getLongParam`, `getStringParam` y `getTimeParam` interpretan al parámetro `n` como el tipo de datos indicado en su nombre.

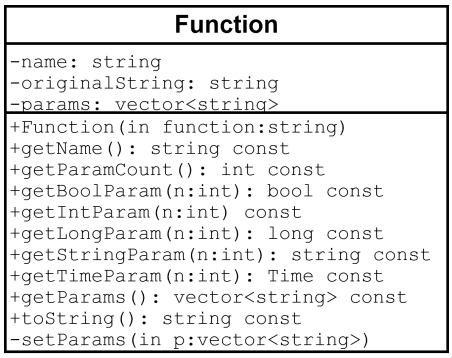


Figura 3.66: Clase Function

Cuando una función se ejecuta sobre un objeto, esta puede devolver:

1. un objeto sobre el cuál ejecutar la siguiente función (representado por `TCommandRunner`)
2. varios objetos sobre los cuales ejecutar la siguiente función (representado por `MultiCommandRunner`)
3. el resultado de una consulta (representado por `CommandQueryResult`)

Estas clases se muestran en la figura 3.67.

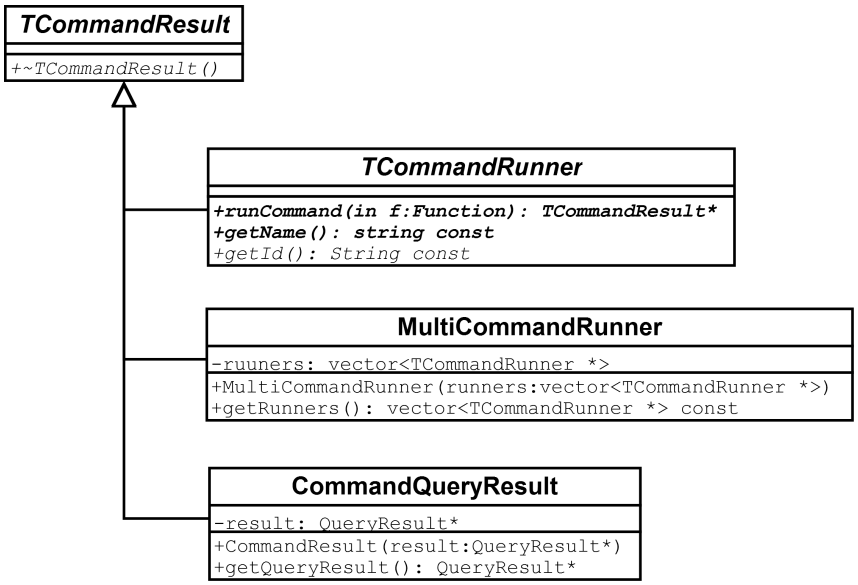


Figura 3.67: Clases de resultados de comandos

La clase `TCommandResult` tiene como propósito proveer un tipo común para estos tres tipos de resultados, siendo por ello la clase base.

Para que una clase pueda ejecutar funciones, debe heredar de `TCommandRunner` e implementar el método `runCommand`. Mediante este método, se le pasa la función que debe ejecutar y se devuelve un puntero a un objeto `TCommandResult` que es el objeto sobre el que se ejecutará la próxima función.

Los métodos `getName` y `getId` permiten que las clases descendientes den su nombre y una identificación para utilizar al generar la salida del simulador.

Cuando una función devuelve el resultado de una consulta, lo debe hacer a través de un objeto `CommandQueryResult`, el cual simplemente contiene el objeto `QueryResult` a devolver. Este valor se inicializa en el constructor y se accede posteriormente mediante el método `getQueryResult`.

En el caso de que la función devuelva varios objetos, debe devolver una instancia de `MultiCommandRunner`. Esta clase contiene el atributo `runners` que es un vector de punteros a `TCommandRunner`, que son los destinos de la ejecución de la próxima función. Este atributo es inicializado en el constructor y se accede mediante el método `getRunners`.

Ejemplo de uso del sistema de comandos

Para comprender mejor como funciona el sistema de comandos, puede ser útil ver como otras partes del simulador hacen uso de él.

A continuación, se muestra parte de la implementación de `runCommand` en el objeto `HypercubeNetwork`. Algunas de las funciones implementadas fueron omitidos en este ejemplo.

```
TCommandResult *HypercubeNetwork::runCommand(const Function &function)
{
    if (function.getName() == "query")
    {
        QueryResult *qr = new QueryResult("network");
        qr->insert("addressLength", toStr(addressLength));
        qr->insert("nodeCount", toStr(nodes.size()));
        return new CommandQueryResult(qr);
    }

    if (function.getName() == "setAddressLength")
    {
        addressLength = function.getIntParam(0);
        return this;
    }

    if (function.getName() == "node") {
       return getNode(UniversalAddress(function.getStringParam(0)));
    }

    if (function.getName() == "allNodes") {
        vector<TCommandRunner *>runners;
        map<UniversalAddress, HypercubeNode*>::iterator it;

        for (it = nodes.begin(); it!= nodes.end(); it++) {
            runners.push_back(it->second);
        }

        return new MultiCommandRunner(runners);
    }

    throw invalid_argument("Bad function: " + function.toString());
}
```

Este método compara el nombre de la función pasada como argumento con los nombres válidos de funciones. Si no es ninguno de ellos, se lanza una excepción.

La función `query` efectúa una consulta, por lo que se construye un objeto `QueryResult` con la información pedida, y luego se devuelve un `CommandQueryResult` conteniéndolo.

La función `setAddressLength` lee el primer parámetro de la función como un entero, y asigna el valor de este parámetro a un miembro del objeto. Luego, se devuelve a si mismo, por lo que si hubiera una

función a continuación, se ejecutará sobre el mismo objeto. Esto permite encadenar funciones de este tipo para hacer varias operaciones en una sola linea.

La función `node` llama a un método privado para encontrar el objeto de tipo `Node` cuya dirección universal es la especificada, y este objeto es devuelto directamente, ya que la clase `Node` hereda de `TCommandRunner`, es decir que es capaz de ejecutar comandos, y por ello puede ser devuelta por una función.

La función `allNodes` pone en un vector todos los objetos `Node` de la red y luego los devuelve por medio de un objeto `MultiCommandRunner`, lo que permitirá que la siguiente función se ejecute sobre todos los nodos de la red.

3.2.9. Notificaciones y Consultas

El simulador dá sus resultados a través de notificaciones y consultas. Las primeras se generan cuando ocurre algo particular en el simulador; por ejemplo si un nodo se conecta a la red, se genera una notificación indicando este hecho. En cambio, las consultas son pedidos de información del usuario. Por ejemplo, en un instante determinado, el usuario desea saber las direcciones de todos los nodos de la red.

Internamente, las notificaciones funcionan de forma similar a las consultas, excepto que estas las genera el simulador mismo.

Para que la salida de la simulación pueda ser procesada, se diseñó de forma tal que las notificaciones y consultas pasan por dos etapas antes de ser escritas en un archivo.

La primera etapa es el filtrado, donde se decide si la consulta va a ser escrita o no, a partir de una configuración dada por el usuario. Esto permite que sólo se escriba la información que se va a utilizar, dado que de escribir todo se generarían archivos muy grandes, impactando en el rendimiento y dificultando su procesamiento.

La segunda etapa es la que le da el formato a los datos, es decir que transforma las estructuras de datos en memoria en el formato que se va a escribir en disco. La implementación del simulador provee un formateador XML.

En la figura 3.68 se muestran las clases que componen este módulo.

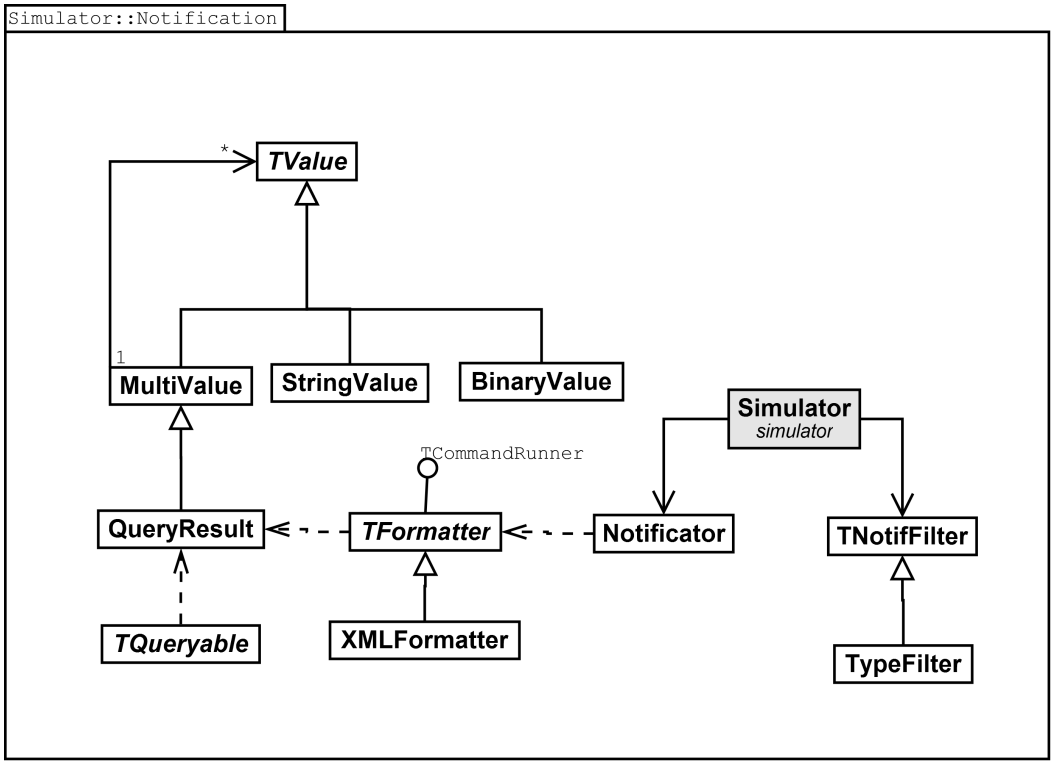


Figura 3.68: Diagrama de clases de Notificaciones y Consultas

Primeramente, las consultas se arman en memoria. Una consulta está compuesta por claves y valores, por ejemplo la clave podría ser 'address' y el valor '101011'. Pero un valor puede ser además otra colección de claves y valores, por lo que es posible armar una estructura recursiva con tantos niveles como se desee.

Las clases que se utilizan para armar estas consultas se muestran en la figura 3.69.

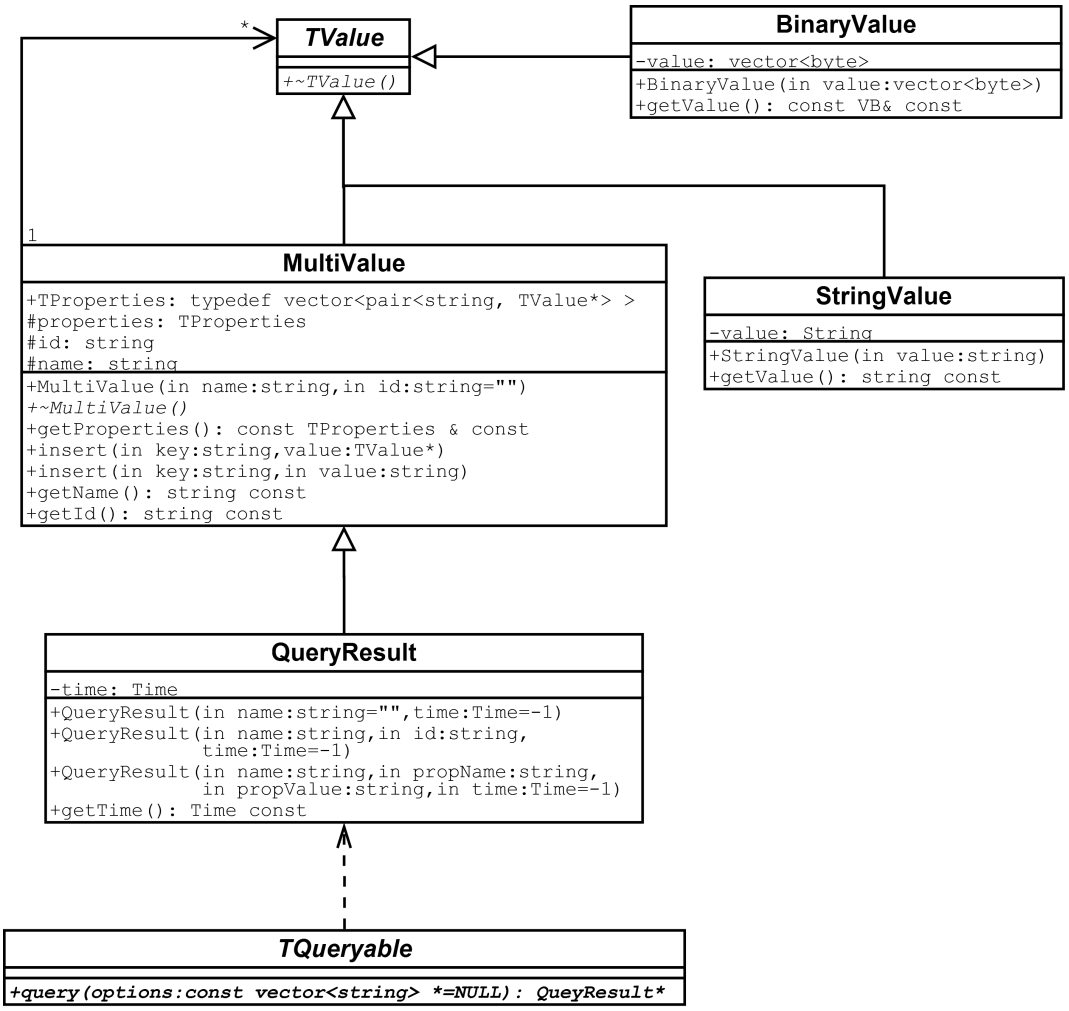


Figura 3.69: Diagrama de clases de consultas en memoria

La clase `TValue` es el ancestro común para todos los tipos de valores. Los valores de cadenas de caracteres, como ser el nombre de un nodo o una dirección, son representados con la clase `StringValue`. Los valores binarios, como por ejemplo el contenido de un paquete, se representan con la clase `BinaryValue`.

La clase `MultiValue` es un valor que a su vez contiene una colección de claves y valores en el atributo `properties`. Las propiedades se agregan mediante el método `insert`, que provee una variante para agregar una propiedad de cualquier tipo y otra que agrega una cadena de carácteres para mayor comodidad. El método `getProperties` devuelve el vector con todas las propiedades.

Esta clase además cuenta con un nombre y una identificación que se utilizan en el archivo de salida como encabezado de todas las propiedades.

De `MultiValue` hereda `QueryResult`, que le agrega el tiempo en el que fue realizada la consulta.

La clase abstracta `TQueryable` provee una interfaz que garantiza que el objeto tiene un método `query`; esto se utiliza en las notificaciones para pasar el objeto a ser consultado, y si el filtro lo acepta, se ejecuta el método para obtener el resultado.

Una vez que se tiene armada una consulta en un objeto `QueryResult`, esta deberá pasar por la etapa de filtrado, que según criterios definidos por el usuario determinará si debe ser escrita en el archivo de

salida o no. Las clases para implementar este filtrado se muestran en la figura 3.70.

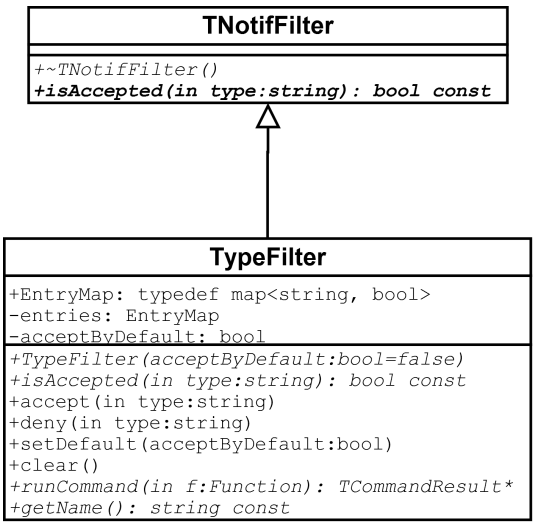


Figura 3.70: Clases `TNotifFilter` y `TypeFilter`

La clase `TNotifFilter` es la clase base, que define el método `isAccepted`. Las clases descendientes deben sobreescribir este método para devolver verdadero o falso según si la notificación debe ser escrita o no.

Se provee una implementación de filtro mediante la clase `TypeFilter`, que permite que el usuario especifique tipos aceptados y rechazados.

Los tipos se componen de varias partes, separadas por puntos. Ejemplos de tipos son "`node.message.connected`", "`node.testApplication.received`" y "`packet.discarded`". Cuando un tipo se acepta o rechaza, todos sus subtipos también. Por ejemplo, si se acepta "`node`", entonces tanto "`node.message.connected`" como "`node.testApplication.received`" son aceptados. Si luego se rechaza "`node.message`", entonces ahora el primero es rechazado y el segundo aceptado, ya que se buscan primero las coincidencias más largas.

Si un tipo no se encuentra ni aceptado ni rechazado en los criterios provistos por el usuario, entonces se utiliza el comportamiento por defecto, configurado por el usuario.

Los criterios del usuario se guardan en el atributo `entries` utilizando un mapa que asocia un tipo con un valor `bool`, indicando si es aceptado o rechazado. El atributo `acceptByDefault` indica si se deben aceptar o rechazar los tipos para los que no se definió un criterio explícito.

El constructor de la clase permite especificar el comportamiento en el caso de no encontrar un criterio para un tipo.

Los métodos de ésta clase son:

- `isAccepted`: busca entre los criterios del usuario el más largo que coincida con el tipo pasado como parámetro.
- `accept`: el tipo pasado como parámetro será aceptado.
- `deny`: el tipo pasado como parámetro será rechazado.
- `setDefault`: asigna un valor al atributo `acceptByDefault` para especificar el comportamiento en el caso de no encontrar un criterio para un tipo.
- `clear`: borra todos los criterios introducidos por el usuario.

- `runCommand`: permite ejecutar `accept`, `deny` y `setDefault` utilizando comandos del simulador.
- `getName`: devuelve la cadena de caracteres "`TypeFilter`".

Si la notificación o consulta pasa la etapa de filtrado, debe ser formateada antes de ser escrita en un archivo de salida. Esto se implementa en las clases que muestra la figura 3.71.

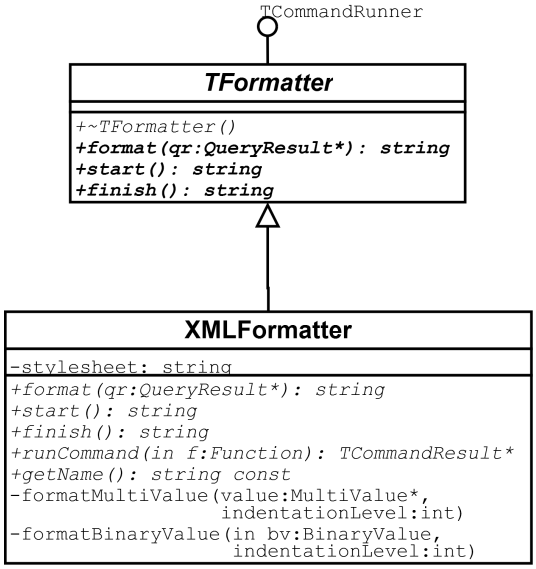


Figura 3.71: Clases `TFormatter` y `XMLFormatter`

La clase `TFormatter` provee los métodos abstractos `start` y `finish`, que son llamados cuando comienza y termina la simulación respectivamente, y el método `format`, que a partir de una consulta (`QueryResult`) genera una cadena de caracteres que será escrita en un archivo.

La clase `XMLFormatter` implementa un formateador XML. Por ejemplo, puede producir una salida como:

```
<?xml-stylesheet type="text/xsl" href="test.xsl"?>
<Simulation>
    <Notification id="simulator.exec.query">
        <time>4410.100000 s</time>
        <command>query</command>
        <network>
            <addressLength>24</addressLength>
            <nodeCount>440</nodeCount>
            <biggestMask>11</biggestMask>
        </network>
    </Notification>
</Simulation>
```

Este formateador permite que el usuario especifique un archivo de hoja de estilo (`stylesheet`) mediante el comando `setStylesheet`. El nombre de este archivo se guarda en el atributo `stylesheet` y se escribe en el archivo de salida antes de la etiqueta raíz.

Los métodos de esta clase cumplen con las siguientes funciones:

- `format`: llama a `formatMultiValue` utilizando un nivel de indentación de 1.
- `start`: devuelve la cadena de caracteres "`<Simulation>`", anteponiendo la etiqueta para el uso de hoja de estilo si la hubiera.

- `finish`: devuelve la cadena de caracteres "`<Simulation>`".
- `runCommand`: ejecuta el comando `setStylesheet`, que asigna el nombre de archivo pasado como parámetro al atributo `stylesheet`.
- `getName`: devuelve la cadena de caracteres "`XMLFormatter`".
- `formatMultiValue`: genera el XML para un objeto de `MultiValue`, haciendo llamadas recursivas si fuera necesario. El parámetro `indentationLevel` permite pasar el número de espacios que se esriben antes de las etiquetas, con el fin de que la salida XML esté tabulada para ser leída adecuadamente por un usuario.
- `formatBinaryValue`: genera el XML para un objeto de tipo `BinaryValue`, escribiéndo cada uno de sus bytes en hexadecimal.

Una vez que se obtuvo la cadena de caracteres formateada, debe ser escrita en un archivo, lo cual es realizado mendiante la clase `Notificator`, que se muestra en la figura 3.72.

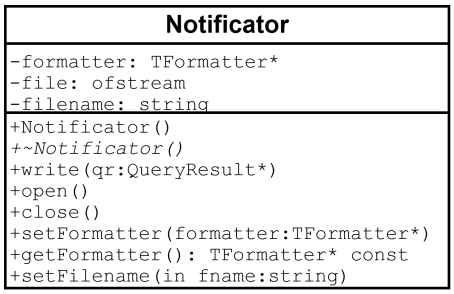


Figura 3.72: Clase `Notificator`

Esta clase tiene el atributo `formatter`, que es un puntero al objeto encargado de dar formato a la consulta antes de escribirla. El nombre del archivo de salida se guarda en `filename`, y en `file` el `handle` del archivo abierto.

Los métodos de esta clase son:

- `write`: da formato al `QueryResult` utilizando el objeto `formatter` y lo escribe en el archivo de salida.
- `open`: abre el archivo de salida.
- `close`: cierra el archivo de salida.
- `setFormatter`: asigna el objeto que será utilizado para dar formato.
- `getFormatter`: devuelve un puntero al objeto que será utilizado para dar formato.
- `setFilename`: asigna el nombre de archivo de salida.

Capítulo 4

Simulaciones

Utilizando `QUENAS`, el simulador descripto en el capítulo anterior, se diseñaron dos tipos de simulaciones: uno consistente en redes generadas aleatoriamente, y el otro utilizando Sistemas Autónomos reales.

Para efecutar estas simulaciones, se necesitaron varios pasos además de la simulación en si misma, como muestra la figura 4.1.

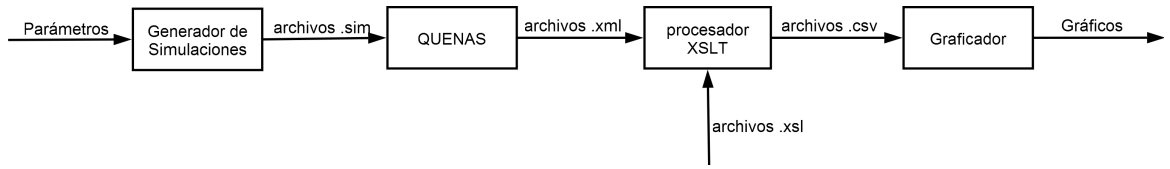


Figura 4.1: Proceso de simulación

Dado que se utilizan redes con miles de nodos, la escritura de las simulaciones a mano resulta poco factible. Por ello, se implementaron para cada caso generadores de simulaciones, que a partir de ciertos parámetros generan archivos de simulación `.sim`.

Estos archivos son tomados como entrada por el simulador, que genera como resultado archivos `.xml`.

Los archivos `.xml` contienen diversos aspectos de los resultados de la simulación. Por ello, cada archivo `.xml` es procesado con varios *templates XSLT*, cada uno de los cuales se encarga de obtener una perspectiva distinta. Por ejemplo, uno extrae los tiempos de conexión, otro los grados de los nodos y así sucesivamente, grabando los datos en un formato de tabla separado por comas (archivos `.csv`).

Los archivos `.csv` son leídos con un programa graficador, donde los datos son procesados para obtener los gráficos que se muestran en este capítulo.

4.1. Redes Aleatorias

Esta simulación se basa en una red aleatoria de nodos ubicados en un plano. Dos nodos se encuentran conectados si la distancia entre ellos es menor a un radio de cobertura fijo.

La simulación construye redes con estas características, monitoreando como se conectan los nodos, y luego envía paquetes entre distintos pares para obtener información del funcionamiento del ruteo.

Se crearon redes de distintos tamaños para evaluar el comportamiento cuando crece la red. Las cantidades de nodos utilizadas son: 10, 14, 20, 32, 50, 70, 100, 140, 200, 320, 500, 700, 1000, 1400, 2000, 3200, 5000 y 7000. Además, para disminuir el error, se efectuaron 10 series con distintas redes aleatorias para cada tamaño, totalizando 180 simulaciones.

En estos gráficos, salvo que se indique lo contrario, los valores obtenidos provienen de promediar las 10 series.

Desarrollo de la simulación

El generador de simulaciones debe armar una red que en todo momento sea conexa, dada que `ANTop` por el momento no soporta la unión de dos redes.

Se programó un generador de simulaciones en C++ que toma como parámetro el número de nodos en la red, y genera un archivo *.sim* con la simulación.

Para ello, utiliza un plano cartesiano y ubica a un primer nodo en una posición determinada. Luego, en cada paso se intenta ubicar un nuevo nodo en coordenadas aleatorias. Si este nodo se encuentra a una distancia menor al radio de cobertura de cualquier otro nodo, entonces es aceptado, y se escriben las instrucciones correspondientes en el archivo de simulación. Si no está en el radio de cobertura, se repite el paso, ubicándolo en una nueva posición hasta que esté incluido en la cobertura de otro nodo.
La figura 4.2 ejemplifica la generación hasta el cuarto nodo.

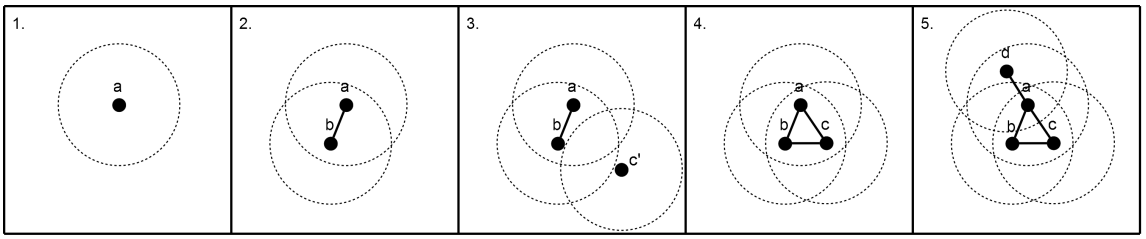


Figura 4.2: Generación de una red aleatoria. En cada paso se ubica un nodo, y si este no entra en el radio de cobertura de alguno de los existentes (como c' en el paso 3), entonces se prueba otra ubicación. Si entra en algún radio de cobertura, se incluye en la simulación.

Los nodos son considerados puntuales; sin embargo se dibujan como un círculo lleno para mayor claridad. Los círculos concéntricos mayores con lineas punteadas indican el radio de cobertura. Las lineas entre nodos marcan las conexiones.

En el paso 1 se crea el primer nodo (a) en una posición arbitraria, generando en el archivo de simulación el código para crear este nodo y conectarlo a la red (que inicialmente está vacía):

```
[0 ms] newNode(01)
node(01).joinNetwork
```

En el archivo de simulación, los nombres de los nodos son números secuenciales que utilizan ceros a la izquierda para que todos tengan la misma cantidad de dígitos; por ello el nodo a se denomina `01`. En

el paso 2, se ubica un segundo nodo que, al estar dentro del radio de cobertura del primero, genera el código:

```
[1000 ms] newNode(02)
newConnection(02,01,54 Mbps,72us)
node(02).joinNetwork
```

Esto crea a un segundo nodo y lo conecta con el primero, especificando el ancho de banda en 54 Mbps (se utiliza este valor en todas las conexiones en esta simulación) y un retardo de 72 microsegundos. Los valores de retardo son proporcionales a la distancia entre los nodos.

En el paso 3, el nodo *c'* cae fuera del área de cobertura de los nodos *a* y *b*, por lo que se descarta y se prueba en otra ubicación (paso 4) conectándose a ambos nodos mediante el código:

```
[1200 ms] newNode(03)
newConnection(03,01,54 Mbps,143us)
newConnection(03,02,54 Mbps,72us)
node(03).joinNetwork
```

Finalmente, en el paso 5, el nodo *d* se conecta con el nodo *a*, produciendo el código:

```
[1400 ms] newNode(04)
newConnection(04,01,54 Mbps,199us)
node(04).joinNetwork
```

Luego de generar todas las instrucciones de conexión, se escriben las siguientes consultas en el archivo de simulación:

- `allNodes.query()`: provee información de direcciones primarias y secundarias de cada nodo.
- `exportConnections(output/filename.csv)` genera un archivo con las conexiones establecidas.
- `allNodes.rendezVousServer.query(size)` informa el tamaño de la tabla de *Rendez Vouz* de cada nodo.
- `allNodes.query(stats)`: provee estadísticas del envío de paquetes de control.
- `query`: provee datos de la red, en particular la máxima máscara utilizada, que equivale a la dimensión necesaria para conectar la red.

Después, se utiliza la aplicación `testApplication` para efectuar envíos desde 100 nodos aleatorios a destinos también aleatorios. Cada envío consiste en realidad en 3 envíos, ya que cuando el destino recibe el paquete reponde, y al recibirlo el origen vuelve a enviarle un paquete. Esto permite comparar la longitud de los caminos de ida, vuelta, e ida una vez que las tablas de ruteo están armadas.

En esta primera etapa, los envíos se efectúan a una frecuencia tal que el siguiente envío se produzca una vez que las tablas de ruteo se borraron, para evaluar el comportamiento sin carga.

Este proceso se repite con carga; para ello primero se mandan paquetes aleatorios que cargan las tablas de ruteo, y luego se hacen envíos como los anteriores pero con una frecuencia mayor para que no se borren las tablas de ruteo.

Finalmente, se efectúa la consulta `allNodes.routing.table.query(size)`, que informa el tamaño de la tabla de ruteo de todos los nodos.

Luego de correr todas las simulaciones y procesarlas con `XSLT`, se utilizó un programa graficador para tomar estos datos y producir gráficos, que se presentan en los siguientes apartados.

Ilustración de las redes en el plano

Como se explicó anteriormente, las redes para estas simulaciones se generaron ubicando nodos en el plano y conectándolos si su distancia es menor a un radio de cobertura. Las redes obtenidas para 100 y 7000 nodos se muestran en la figura 4.3.

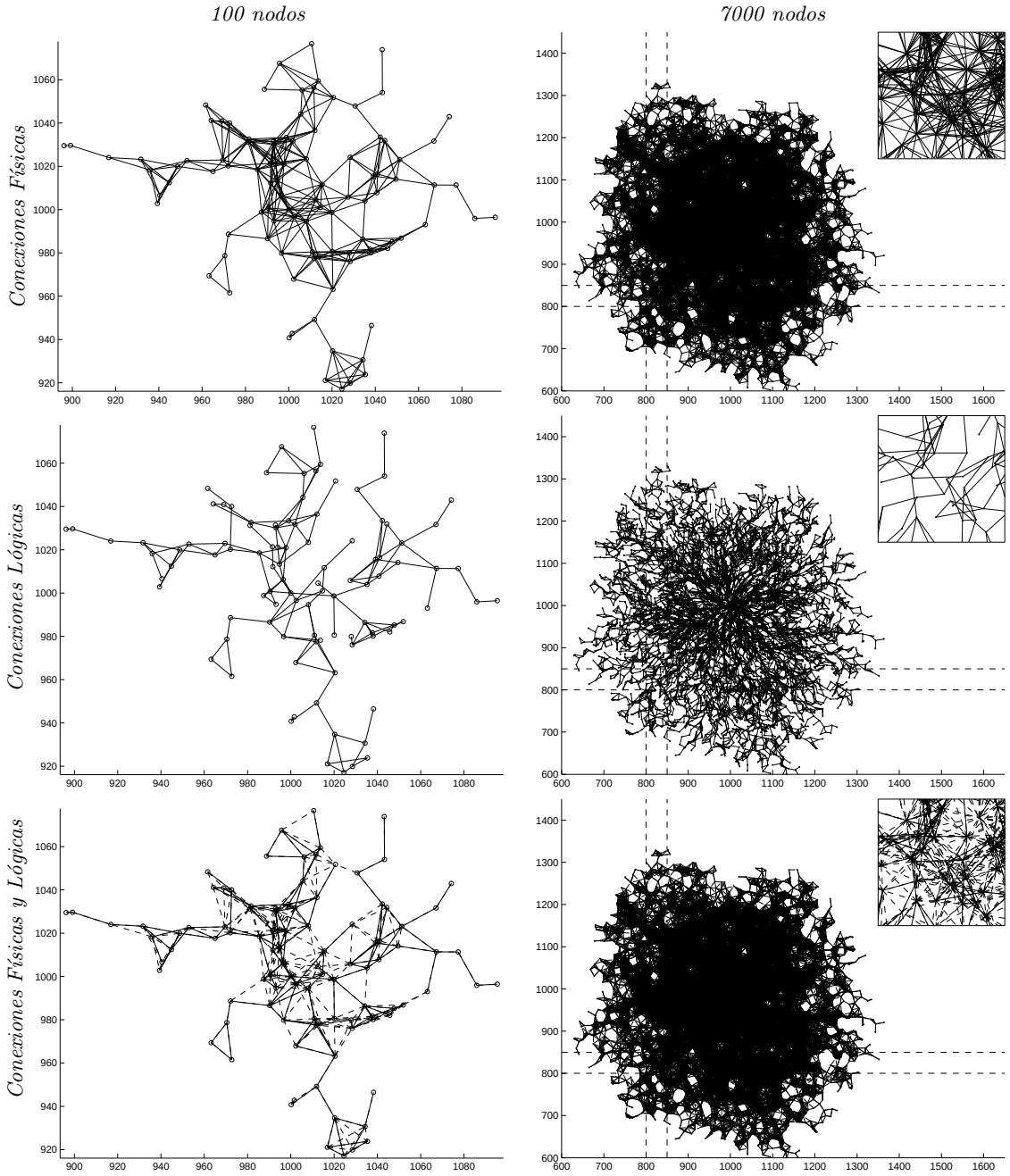


Figura 4.3: Ilustración de las redes en el plano

Las tres figuras de la izquierda son para una de las redes de 100 nodos (ya que se simularon 10 distintas) y las de la derecha para una de las redes de 7000 nodos. El recuadro en la parte superior derecha de cada figura de 7000 nodos muestra una ampliación de la zona marcada por lineas punteadas.

Los círculos representan nodos, mientras que las líneas son las conexiones entre ellos. En la primera fila se muestran las conexiones físicas, es decir, cuando los nodos están a una distancia menor al radio de cobertura, y por ello pueden comunicarse.

En la segunda fila se muestran las conexiones lógicas, que son las conexiones que el protocolo pudo establecer. Para que haya una conexion lógica debe haber una conexión física y los nodos deben tener direcciones adyacentes en el hipercubo, primarias o secundarias.

La última fila muestra las conexiones físicas y lógicas en un mismo gráfico, siendo las lineas contínuas las conexiones lógicas (y por ende también físicas) y las lineas punteadas las conexiones físicas donde no se pudo establecer una conexión lógica.

Se observa en las conexiones físicas que tiende a haber una mayor concentración de nodos y conexiones en el centro, disminuyendo hacia la periferia. Esto se debe puramente a la forma de construcción de la red.

En las conexiones lógicas se ve que la topología se asemeja más a un árbol con las raices en el centro y ramas expandiéndose hacia la periferia. Es decir, el efecto observado en las conexiones físicas se ve acentuado.

Al comparar las conexiones lógicas y físicas en la última fila, las lineas punteadas (es decir, que existe conexión física pero no lógica) representan atajos que acortarían los caminos si se pudieran aprovechar. Se ve que unos pocos de estos atajos reducirían la distancia de forma considerable, siendo estos los que permiten pasar de una zona periférica a otra adyacente sin necesidad de pasar por el centro. Sin embargo, la mayoría de los atajos ahorraría sólo 1 o 2 saltos, ya que existe un camino alternativo utilizando los nodos cercanos.

4.1.1. Parámetro `d` mínimo

En el protocolo `ANTop`, el parámetro `d` es la dimensión del hipercubo, es decir, la cantidad de bits que componen una dirección.

Resulta de interés práctico conocer la longitud mínima de dirección para poder crear una red con una determinada cantidad de nodos.

Para medir esto, lo que se hace en la simulación es utilizar siempre un parámetro `d` grande, de forma tal que todos los nodos se puedan conectar, y luego buscar cuál es la máscara más larga utilizada. Esto es equivalente al parámetro `d` mínimo, pero medirlo directamente implicaría repetir la simulación con distintos valores de `d` hasta encontrar el mínimo valor que permita conectar todos los nodos.

El resultado obtenido se muestra en la figura 4.4. Para cada tamaño de red `N` se promedia el parámetro `d` mínimo de las 10 simulaciones efectuadas,que se representa mediante la curva dibujada con línea contínua y barras de desviación estandar. Se buscó una curva que aproxime el resultado obtenido, que se traza en línea punteada, cuya función es:

$$d' = 1{,}047(\ln N)^2 - 3{,}8218(\ln N) + 9{,}1093$$

El error cuadrático medio para esta aproximación es de 1,8.

Como es de esperar, el parámetro d aumenta cuando crece la red, pudiéndose aproximar por un polinomio de segundo orden en función del logaritmo de la cantidad de nodos. Considerando que las direcciones pueden tener hasta 255 bits, una red con estas características podría tener hasta 31 millones de nodos aproximadamente.

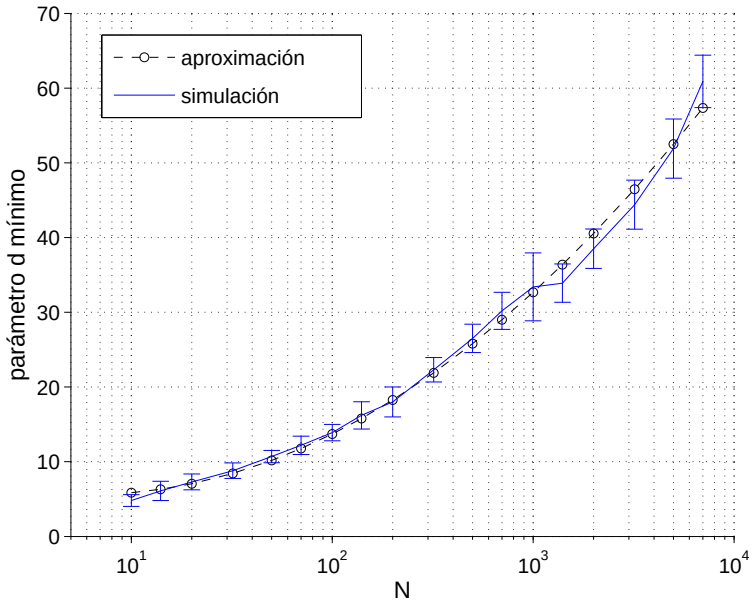


Figura 4.4: Parámetro `d` mínimo

4.1.2. Grados lógicos y físicos de los nodos

Cada nodo se encuentra conectado físicamente a una cierta cantidad de nodos, siendo esta cantidad el "grado físico" del nodo, mientras que la cantidad de conexiones que puede establecer el protocolo para un nodo es el "grado lógico".

Para cada uno de los tamaños de red, siendo `N` el número de nodos de la red, se promedian los grados físicos y lógicos de cada uno de los nodos, valores que a su vez son promediados en las 10 series, y se obtiene el gráfico de la figura 4.5.

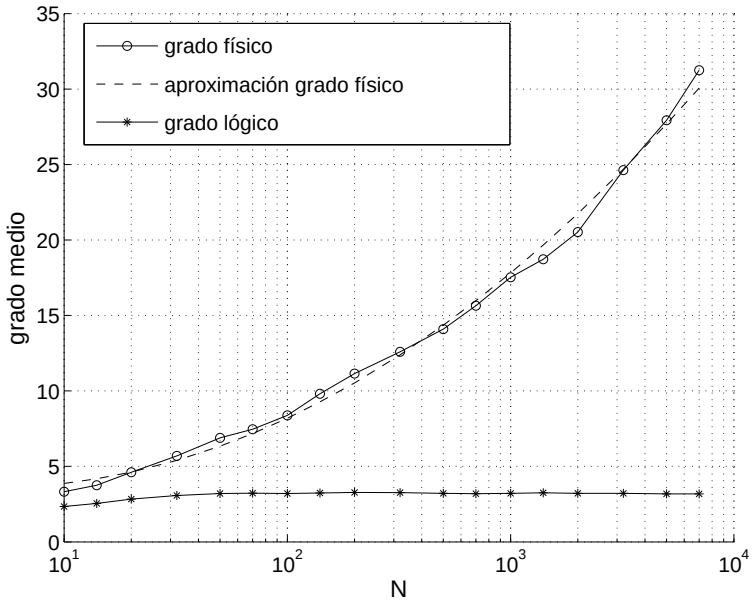


Figura 4.5: Grado medio de los nodos

Este gráfico muestra en curvas separadas los promedios de grados físicos y lógicos en función del

tamaño de red `N`, así como una curva que aproxima al grado físico mediante la función:

$$d' = 0{,}4997(\ln N)^2 - 1{,}5764(\ln N) + 4{,}8572$$

El error cuadrático medio para esta aproximación es de 0,32. En una red de 31 millones de nodos, que es la red aleatoria más grande que se estima que se puede crear utilizando 255 bits de direccionamiento, el valor medio del grado físico sería casi 130.

Se observa que el grado físico crece cuando aumenta `N`, es decir que en promedio los nodos tienen más vecinos dentro de su radio de cobertura. Sin embargo, el grado lógico es prácticamente independiente de `N`, siendo su valor aproximadamente 3,1. Esto indica que por más que las posibilidades de conexión aumentan, el protocolo no es capaz de asignar direcciones adyacentes para poder aprovecharlas.

Se vió en el gráfico anterior cuál es el grado medio de los nodos para distintos tamaños de red. Ahora se entará en el detalle de como están distribuidos los grados que conforman estos promedios.

La figura 4.6 muestra los casos de 100 y 7000 nodos, en las columnas izquierda y derecha respectivamente.

La primera fila es un histograma de la distribución de los grados lógicos en los nodos, indicando cada barra cuantos nodos hay en promedio (sobre las 10 simulaciones) con ese grado lógico, y en la segunda fila se muestra un gráfico similar pero con el grado físico. La última fila es también un histogama, pero en escala logarítmica en ambos ejes. En la abscisa se encuentra el cociente entre el grado lógico y el físico para un nodo, mientras que la ordenada es la probabilidad (dentro de las 10 series) de tener un nodo con ese cociente de grado.

Por ejemplo, para 100 nodos, la probabilidad de tener un cociente de 0,1 es de aproximadamente 0,015, lo que indica que en el 1,5 % de los nodos el grado físico es 10 veces el grado lógico.

En los histogramas de grado lógico se observa que no hay grandes diferencias para 100 y 7000 nodos; en ambos casos el grado 3 es el que tiene mayor cantidad de nodos y luego decrece rápidamente. En la red de 7000 nodos se tienen algunos pocos ejemplares con grados mayores a los de la red de 100.

La distribución de los grados físicos tiene una curvatura similar para los dos tamaños de red pero con un rango mucho mayor para la red de 7000 nodos.

Como la distribución de grado lógico no varía sustancialmente con el tamaño de la red mientras que la de grado físico si lo hace, a medida que la red crece el grado medio lógico se mantiene aproximadamente constante mientras que el grado medio físico aumenta, que es lo que se mostró previamente en la figura 4.5, donde las curvas se van separando cuando aumenta `N`.

La última fila muestra como se relacionan los histogramas de las otras dos filas. El máximo indica aproximadamente cuanto se debe reducir la curva de grado físico para que la zona más poblada coincida con la más poblada en el grado lógico.

En la red de 100 nodos, el máximo se encuentra en aproximadamente 0,45, y el pico de grado físico se encuentra en 7, entonces $7 * 0{,}45 = 3{,}15$ que es donde se encuentra la mayor concentración de grados lógicos.

En la red de 7000 nodos, el máximo se encuentra en 0,09 y el pico de grado físico se encuentra en 33, por lo que $33 * 0{,}09 = 2{,}97$ indica que en torno a 3 se encuentra la mayor concentración de grados lógicos.

En conclusión, el grado físico es mayor cuando crece la red, pudiéndose aproximar por un polinomio de segundo orden en función del logaritmo de la cantidad de nodos, mientras que el grado lógico se mantiene prácticamente constante en torno a 3.

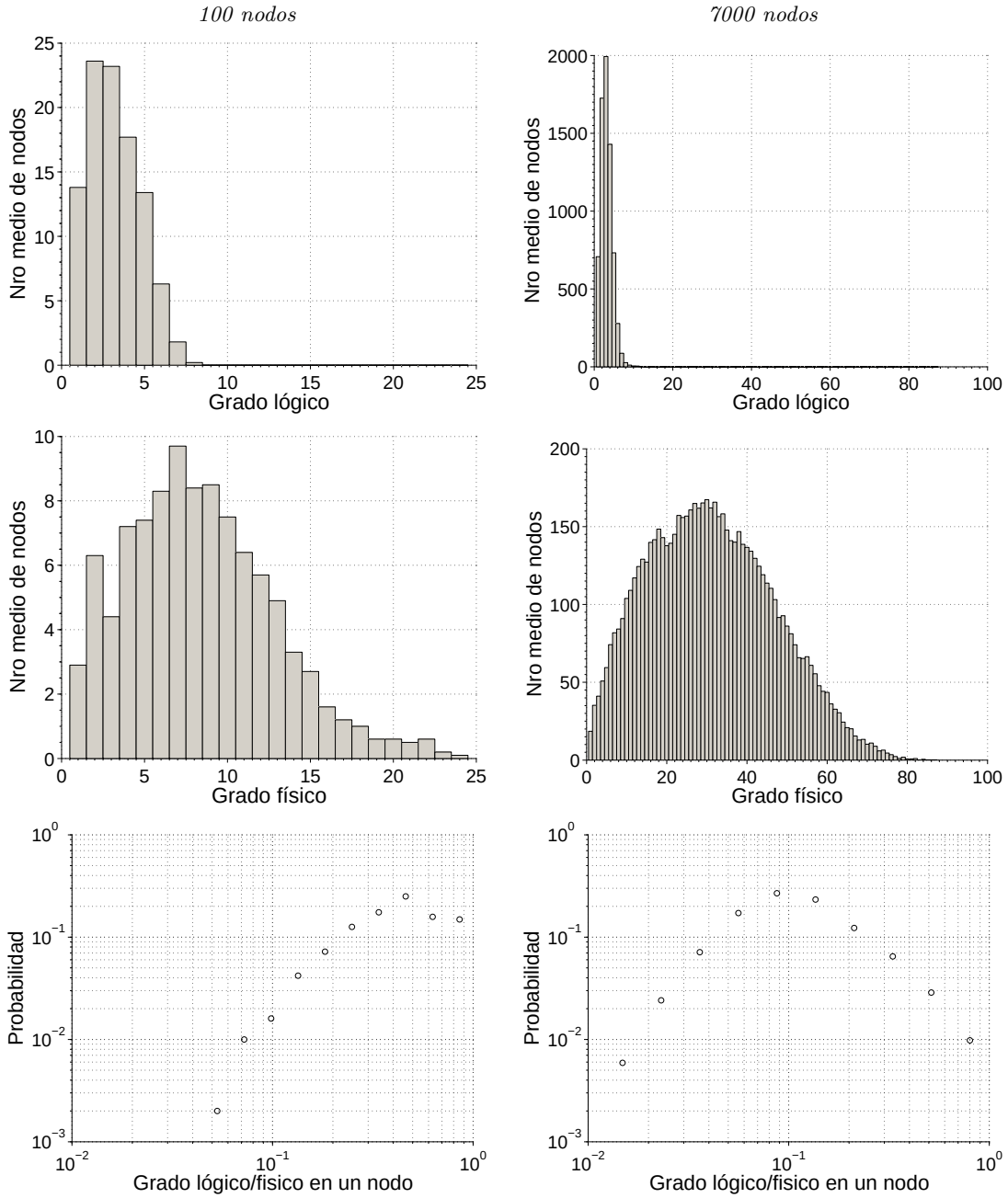


Figura 4.6: Distribución de los grados en los nodos

4.1.3. Máscaras de direcciones

Como se explicó en la sección 2.1.1, los nodos al conectarse utilizan una máscara que determina cuál es el espacio de direccionamiento manejado por el nodo. La máscara, contrariamente a la dirección, puede variar en el tiempo, creciendo si se ceden direcciones a otros nodos y disminuyendo si los hijos devuelven el espacio cedido.

La figura 4.7 muestra como se distribuyen las máscaras. La columna de la izquierda es para redes de 100 nodos, y la de la derecha para redes de 7000.

En la primera fila se muestra un histograma con la cantidad de nodos en promedio (de las 10 series) con una máscara de dirección primaria determinada. Por ejemplo, en redes de 100 nodos, el promedio de nodos que tienen direcciones primarias con máscara 8 es aproximadamente 19.

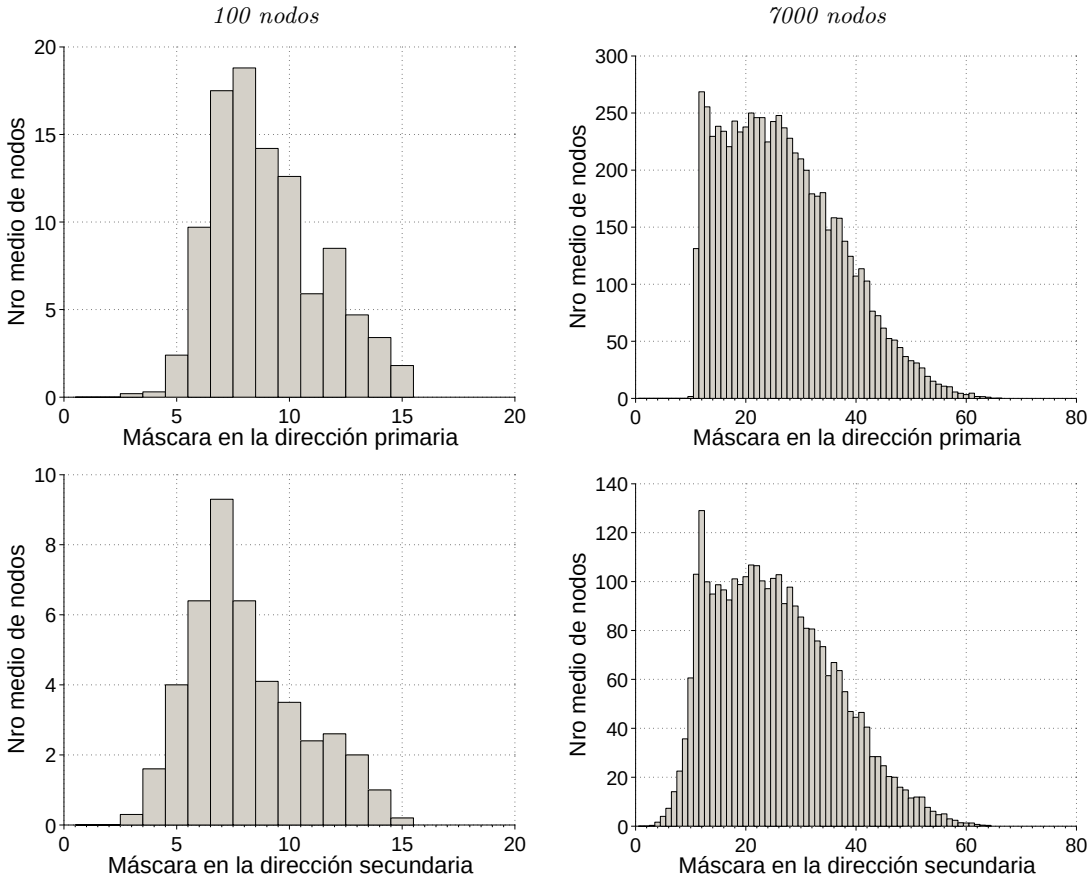


Figura 4.7: Distribución de las máscaras de direcciones. El número medio de nodos es sobre las 10 series simuladas.

En la segunda fila se muestra un histograma similar pero para las máscaras de las direcciones secundarias.

Para las direcciones primarias, se observa que la cantidad promedio de nodos con una máscara aumenta repentinamente en un valor (6 en la red de 100 nodos, 11 en la de 7000), habiendo pocos nodos con máscaras menores. Esto se debe a que cuando un nodo se conecta, elije entre todas las propuestas aquella que tiene la menor máscara, lo cual hace crecer las máscaras más pequeñas, tendiendo a emparejarlas.

En las máscaras para direcciones secundarias, los gráficos son similares a los de las direcciones primarias, pero en este caso existen máscaras más pequeñas, ya que éstas no cambian cuando parte del espacio de direcciones primario es cedido a otros nodos.

Se aprecia en estos gráficos que hay un desplazamiento de máscaras entre direcciones secundarias y primarias, debido a que estas últimas se hacen más restrictivas al ceder parte de su espacio.

4.1.4. Direcciones secundarias

Los nodos utilizan direcciones secundarias para mejorar la conectividad, ya que esto provee direcciones adicionales que al ser adyacentes a la de los vecinos, permiten establecer una conexión lógica.

La figura 4.8 muestra a la izquierda la cantidad de direcciones secundaria en promedio (de todos los nodos de cada una de las 10 series) en función del tamaño de la red. Excepto para redes menores a 70 nodos, el valor se encuentra entre 0.4 y 0.5, lo que indica que hay una dirección secundaria cada poco más de dos nodos.

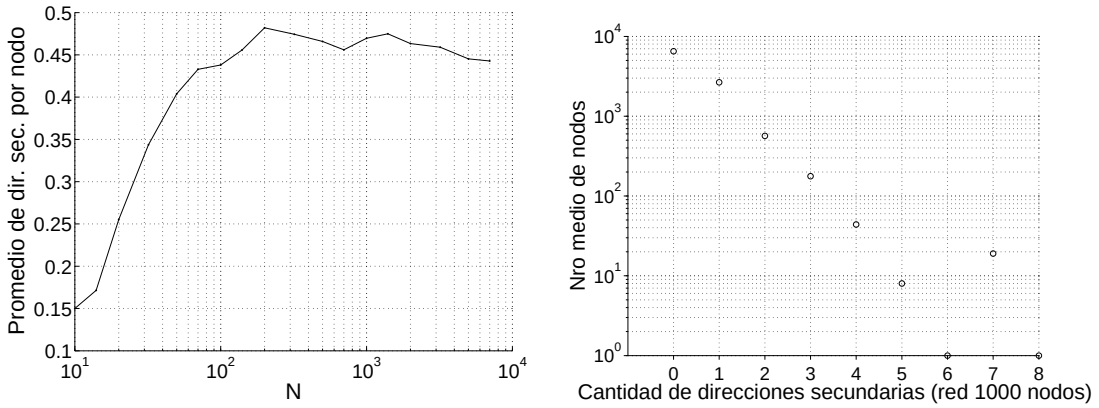


Figura 4.8: Direcciones secundarias por nodo

En la misma figura, a la derecha se muestra el caso particular de la distribución para las redes de 1000 nodos. El 64 % de los nodos no tiene dirección secundaria, el 27 % tiene una dirección secundaria, el 5 % tiene dos y el restante 4 % tiene más de dos.

Los gráficos análogos con redes de otros tamaños son similares, excepto que para redes pequeñas disminuye la cantidad de nodos con múltiples direcciones secundarias, reduciendo el valor promedio mostrado en el gráfico de la derecha.

Para redes aleatorias de más de 70 o más nodos, el número de direcciones secundarias por nodo permanece casi constante en torno a $0,45$, es decir que se otorga en promedio una dirección secundaria cada dos nodos.

4.1.5. Tiempo de obtención de la dirección primaria

Cuando un nodo desea conectarse a la red, debe enviar un pedido de dirección primaria para luego elegir una y conectarse. El tiempo que tarda desde que se ejecuta la instrucción de conexión hasta que se conecta es lo que se mide en la figura 4.9.

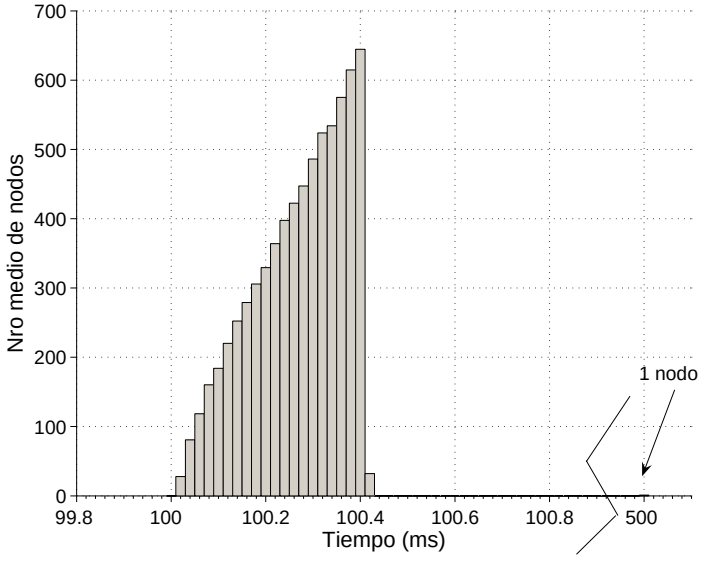


Figura 4.9: Tiempo de obtención de la dirección primaria

En este gráfico se muestra un histograma de la cantidad promedio de nodos que tardaron un cieto tiempo en conectarse.

Dado que luego de enviar el pedido de dirección primaria se esperan 100ms para recibir respuestas, no es posible conectarse en menos de ese tiempo. Una vez que se elije la dirección, se envía el mensaje `PAN` que es respondido con un `PANC` (ver sección 2.2.2). Dado que no se simula el tiempo de procesamiento en los nodos, estos tiempos de conexion están dados entoces por el tiempo que tardan los paquetes en ser enviados, lo cual depende del ancho de banda y retardo, siendo este último proporcional a la distancia.

Además, el primer nodo que intenta conectarse a la red realiza 5 veces el pedido de dirección primaria (tardando 100ms cada uno) antes de considerar que es el primero; es por ello que hay un nodo que tarda 500 ms en conectarse.

Para otros tamaños de redes, la distribución de tiempos es muy similar. Esto muestra que el tiempo de obtención de dirección primaria es independiente del tamaño de red, ya que tan solo se comunica con sus vecinos. Sin embargo, influirá el ancho de banda y retardo en la conexión con los vecinos.

4.1.6. Tiempo de registración en el *Rendez-Vous*

Una vez que el nodo obtuvo la dirección primaria, debe registrarse en el *Rendez-Vous* para asociar su dirección primaria con su dirección de red.

El gráfico 4.10 muestra el tiempo que tarda desde que se le da la instrucción de conectarse al nodo hasta que se registra y por lo tanto otros nodos pueden comunicarse con él, llamándose τ a este tiempo.

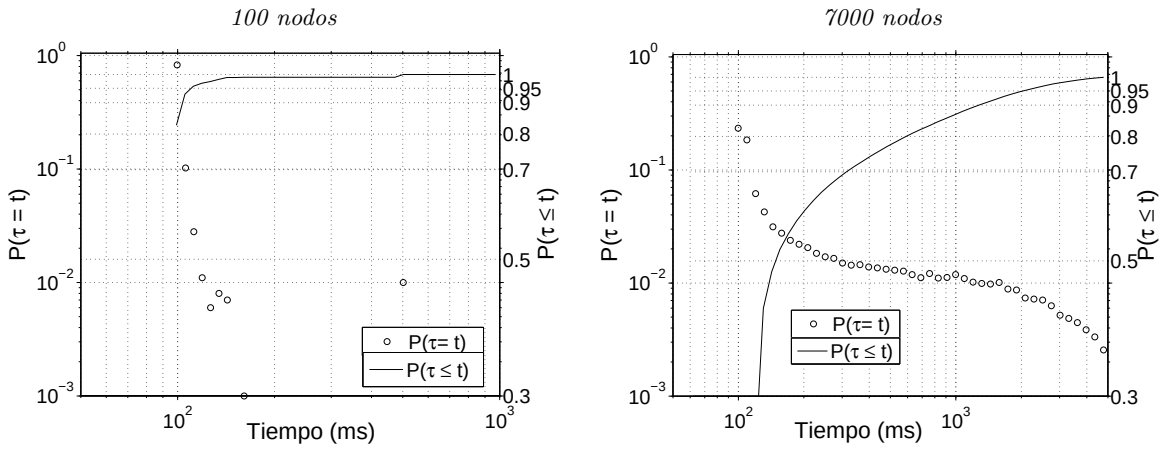


Figura 4.10: Tiempo de registración en el *Rendez-Vous*

En cada uno de estos gráficos se superpone la posibilidad de que un nodo se registre en el tiempo dado por el valor de la abscisa ($P(\tau = t)$, círculos con escala a la izquierda) con el acumulado de esta probabilidad ($P(\tau \leq t)$, línea contínua con escala a la derecha), es decir, la probabilidad de que el nodo ya esté registrado en ese instante de tiempo. El gráfico de la izquierda es para la red de 100 nodos y el de la derecha para la de 7000.

Dado que la registración recién puede comenzar cuando se obtuvo una dirección primaria, el tiempo de registración siempre es mayor o igual al de conexión (siendo igual cuando el nodo *Rendez-Vous* es el mismo nodo). Por ello, al igual que para el tiempo de obtención de dirección primaria, el mínimo es de 100 ms. Se observa que en la red de 100 nodos, poco después de los 100ms la gran mayoría de los nodos se encuentran registrados, mientras que en la red de 7000 nodos se necesitan más de 2s para obtener un alto porcentaje de nodos registrados.

Para hacer un análisis más preciso de la variación de los tiempos de registración en función del tamaño de red, en la figura 4.11 se muestra el tiempo que tardan en registrarse el 95 % de los nodos para cada uno de los tamaños de redes.

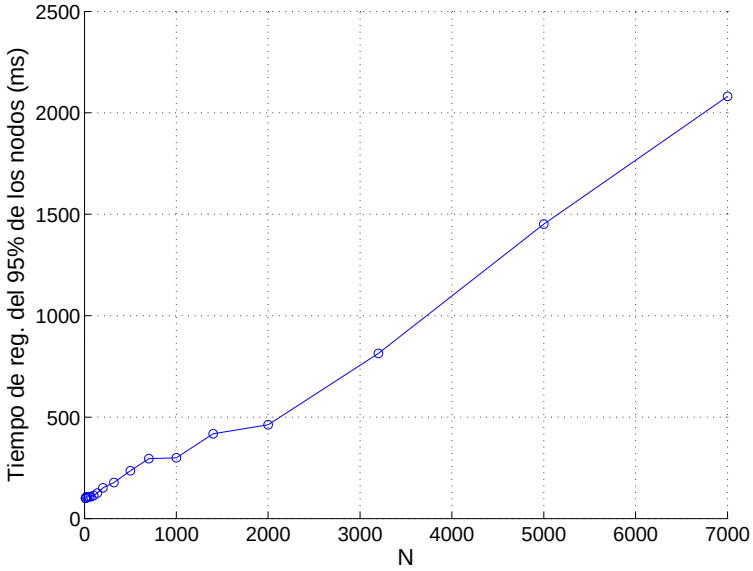


Figura 4.11: Tiempo de registración del 95 % de los nodos

El crecimiento en el tiempo de registración es aproximadamente lineal con la cantidad de nodos, aumentando aproximadamente 300ms cada 1000 nodos. Esto impone un límite práctico en el tamaño de la red menor que el impuesto por la cantidad de bits de la dirección (31 millones de nodos) ya que en una red con esta cantidad de nodos las registraciones podrían tardar más de 2 horas. Poniendo un cota máxima de 10 segundos de registración, se limita la cantidad de nodos en una red de este tipo a 30,000.

4.1.7. Distancia al *Rendez-Vous*

Se analizará ahora la distancia desde un nodo hasta el *Rendez-Vous* donde se registra. Cuando el paquete de registración se envía, es posible que la ruta no esté establecida, en cuyo caso el paquete explorará distintos caminos hasta llegar a destino. La longitud de este camino, sin tener en cuenta los retrocesos, se denomina d_r.

Es de interés comparar d_r con el camino mínimo considerando las conexiones lógicas, cuya longitud se denomina d_m.

En la figura 4.12 se comparan d_r y d_n en función del número de nodos en la red `N`. Estas dos curvas tienen la escala a la izquierda. La línea punteada muestra el cociente entre ellos, con su escala a la derecha.

La relación entre las distancias aumenta con el número de nodos. Para redes de hasta 100 nodos las distancias son muy similares, llegando en la red más grande a duplicar al camino mínimo. Es razonable que esto suceda, ya que en redes más grandes hay muchos más caminos y es más probable que el camino utilizado sea más largo.

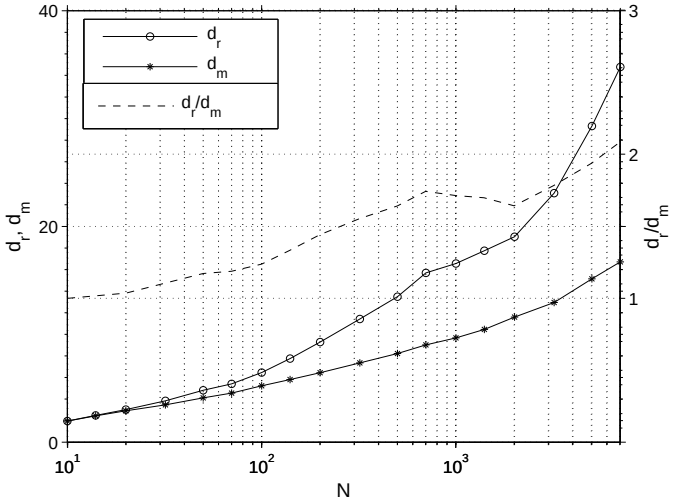


Figura 4.12: Distancia media al *Rendez-Vous*

Se analizarán a continuación los casos particulares de redes de 100 y 7000 nodos. La figura 4.13 muestra como se distribuyen los valores de d_r y d_m para estas redes. Se observa que las distribuciones de distancia recorrida tienen formas similares para 100 y 7000 nodos, y lo mismo ocurre con las distribuciones de distancia mínima.

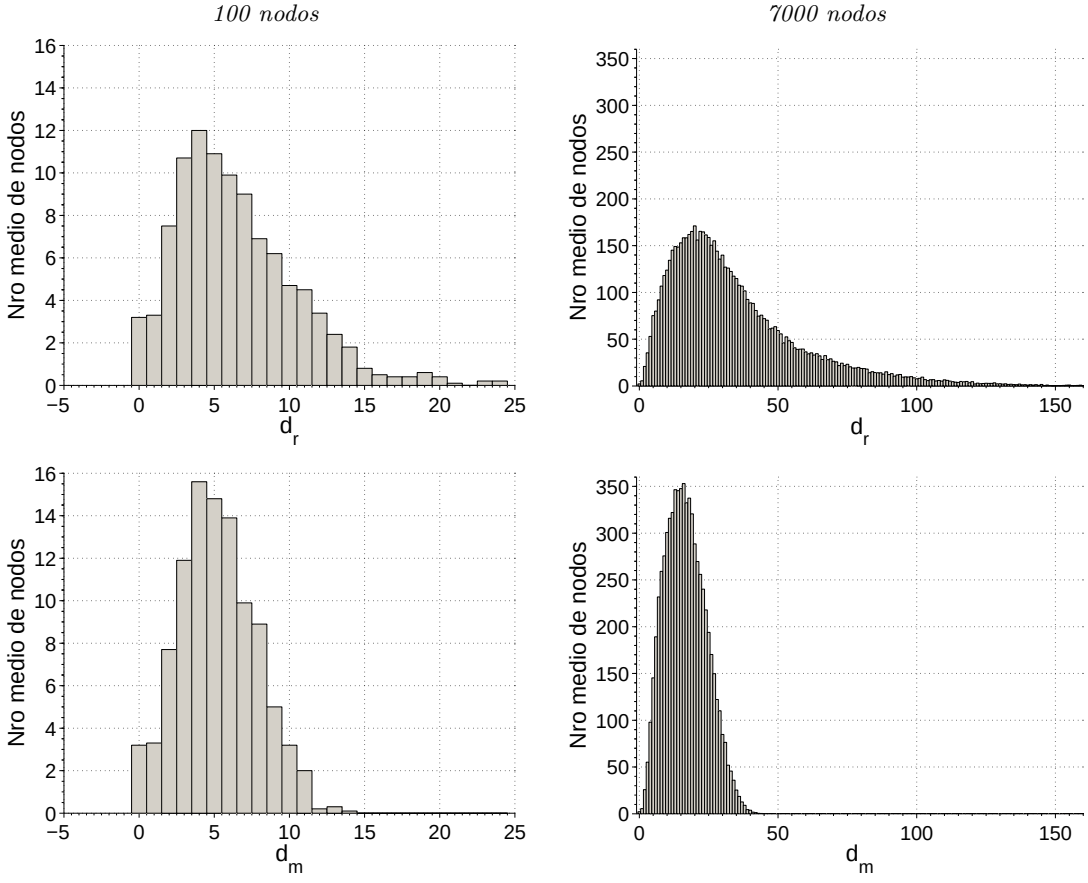


Figura 4.13: Distribución de la distancia al *Rendez-Vous*. El número medio de nodos es sobre las 10 series simuladas.

Además, entre los gráficos de los dos tipos de distancia, la principal diferencia es que para la distancia mínima la caida luego del pico es muy rápida, mientras que en la distancia recorrida cae más lentamente, produciendo una cola que llega a distancias mucho mayores, contribuyendo en el aumento del promedio de la distancia media. Se observa que el pico de la distribución se corre ligeramente hacia distancias mayores en la distancia recorrida, contribuyendo también al aumento del promedio.

Los gráficos anteriores muestran las distribuciones en distintos gráficos, por lo que no indican como se relacionan los dos tipos de distancias en cada registración.

Para eso se realizaron los gráficos de la figura 4.14, que muestra en la primera fila la frecuencia con la que se encuentra asociado cada camino mínimo y recorrido. Cuanto más oscuro es el sombreado de un cuadrado, más paquetes hay con esa relación de caminos. Por ejemplo, en el gráfico de 100 nodos, tomando 5 en el eje de abscisas se ve como se distribuyen los recorridos cuando el camino mínimo toma este valor. En este caso, la mayoría de los paquetes recorre un camino de longitud 5, y se ve además que algunos paquetes efectúan recorridos un poco más largos.

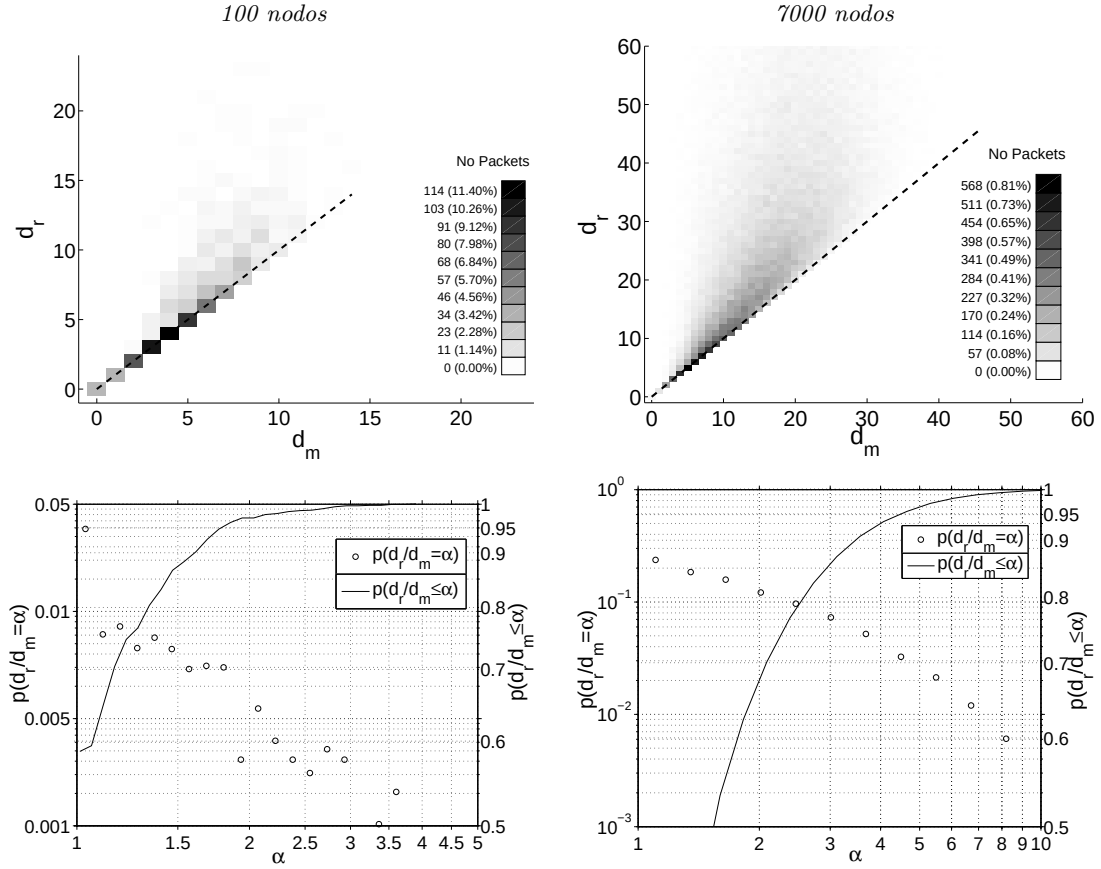


Figura 4.14: Distancia al *Rendez-Vous*

El ruteo óptimo sería si las distancias recorridas fueran iguales a las distancias mínimas, en cuyo caso todos los sombreados deberían encontrarse sobre la diagonal marcada con una línea punteada. Por abajo de ella no puede haber sombreados, ya que el camino recorrido sería menor que el mínimo. Cuanto más lejos se encuentra por arriba de la línea, menos eficiente es el ruteo.

En el gráfico para 100 nodos el sombreado no se expande muy por arriba de la linea punteada, indicando un buen rendimiento, mientras que en la red de 7000 nodos se encuentra más disperso, lo que indica que los caminos recorridos son más largos en comparación con los caminos mínimos.

Los gráficos de la fila inferior muestran como se distribuye la probabilidad del cociente entre ambas distancias. La probabilidad de que d_r/d_m sea el valor indicado en la abscisa (α) se marca con círculos.

Por ejemplo, para la red de 100 nodos, la probabilidad de que la relación sea apenas mayor que uno es de aproximadamente 60 %.

Las curvas continuas son la probabilidad acumulada, es decir, la probabilidad de que el cociente d_r/d_m sea menor o igual al valor de la abscisa. En la red de 100 nodos, aproximadamente el 87 % de las relaciones son menores a 1.5, el 96 % de las relaciones son menores a 2 y el 99 % son menores a 3. En cambio, para la red de 7000 nodos, el 65 % de las relaciones son menores a 2, el 85 % son menores a 3 y el 93 % son menores a 4.

4.1.8. Paquetes de control

El simulador lleva un conteo de los paquetes de control de cada tipo que son recibidos y enviados.

Se analizan estos valores luego de que todos los nodos se encuentran conectados, para las redes de 7000 nodos.

Cuando un nodo desea conectarse a la red, el primer paso es enviar un pedido de dirección primaria mediante un paquete de tipo `PAR` (*Primary Address Request*). Los histogramas de cantidad de paquetes enviados y recibidos se muestran en la figura 4.15.

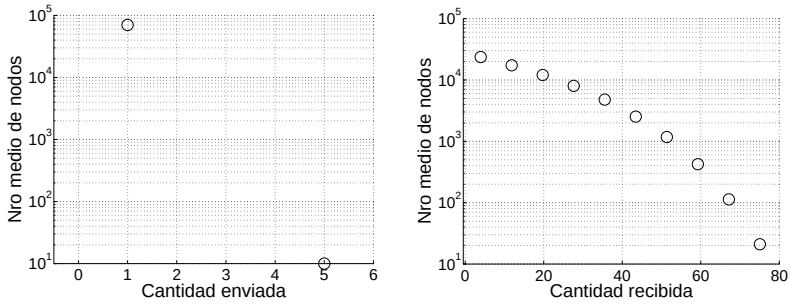


Figura 4.15: Paquetes `PAR`

Estos gráficos muestran cuantos nodos promedio (sobre las 10 series) enviaron o recibieron cada cantidad. Por ejemplo, casi 7000 nodos enviaron exactamente un paquete `PAR`. A pesar de que no se pueda ver debido a la escala, hubo un nodo que mandó 5 veces este paquete. Este es el caso del primer nodo de cada red, que intenta 5 veces encontrar vecinos, y luego se asigna la dirección compuesta por todos 0.

Como los paquetes `PAR` son enviados en modalidad broadcast, cada paquete puede ser recibido por más de un nodo. En particular, cada nodo recibe paquetes `PAR` de todos los nodos que están conectados físicamente a él y que intentan conectarse posteriormente, generando la distribución mostrada.

Al recibir un paquete `PAR`, los nodos responden con un paquete `PAP` (*Primary Address Proposal*), tengan o no dirección para proponer, e indicando si este fuera el caso. La figura 4.16 muestra la distribución para este tipo de paqutes.

Dado que son generados como respuesta al `PAR`, la cantidad enviada es igual a la cantidad de `PAR` recibida. Como el envío es de un nodo a otro, cada paquete enviado es recibido por exactamente un nodo, produciendo que el gráfico de paquetes recibidos también sea igual.

Una vez que un nodo recibe paquetes `PAR`, escoge la dirección más conveniente y la notifica utilizando un paquete `PAN` (*Primary Address Notification*) en modalidad broadcast. La figura 4.17 muestra que todos los nodos envían un paquete de este tipo. En cambio, la cantidad recibida tiene una distribución igual a la de paquetes `PAP`.

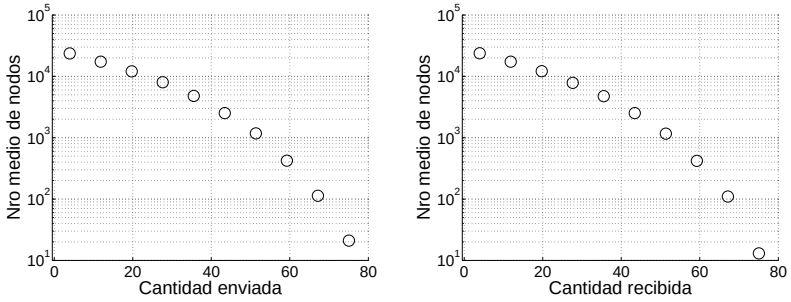


Figura 4.16: Paquetes `PAP`

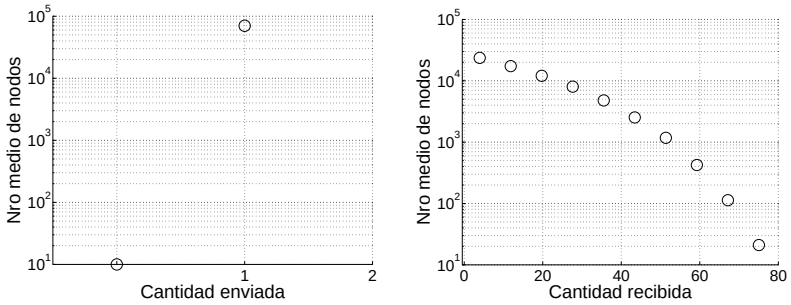


Figura 4.17: Paquetes `PAN`

Cuando el nodo que está proponiendo una dirección primaria recibe un paquete `PAN` confirmando su utilización, responde con un `PANC` (*Primary Address Notification Confirmation*). En cambio, si la dirección contenida en el paquete `PAN` no es la propuesta (es decir que se escogió otra), no se responde. Las distribuciones para paquetes `PANC` se muestran en la figura 4.18.

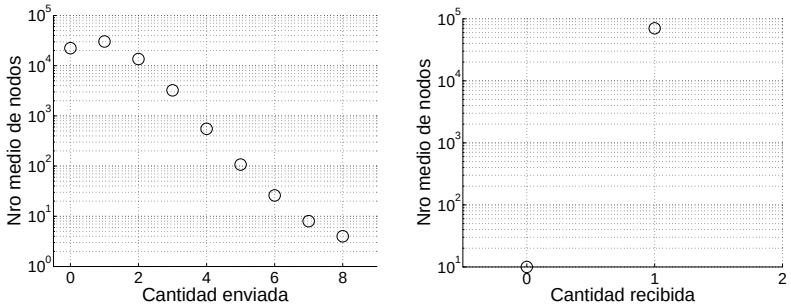


Figura 4.18: Paquetes `PANC`

Se observa que las cantidades de paquetes `PANC` enviados son mucho menores que las de `PAN` recibidos, lo que indica que en numerosas ocasiones se propone una dirección primaria que no es tomada. La cantidad de paquetes `PANC` recibidos es siempre 1, ya que sólo el nodo que está cediendo su dirección primaria debe enviarlo.

Los nodos que se encuentran conectados envían paquetes de tipo `HB` (*Heard Bit*) cada intervalos regulares en modalidad broadcast para notificar a los vecinos que siguen activos. La figura 4.19 muestra la distribución para estos paquetes.

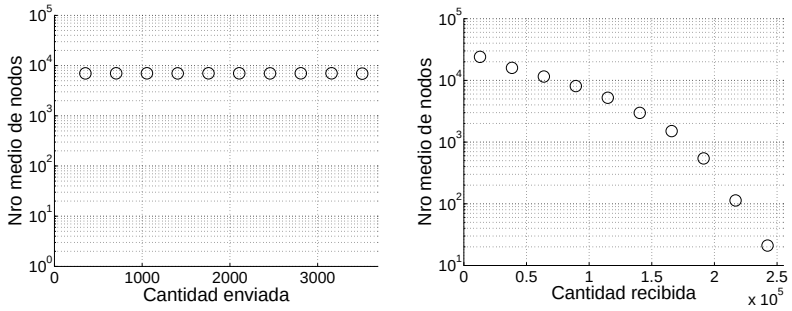


Figura 4.19: Paquetes `HB`

La cantidad de paquetes enviados tiene que ver con el tiempo que transcurre desde que se conecta el nodo hasta que se efectúa esta medición. En cambio, la cantidad recibida también tiene que ver con la cantidad de conexiones físicas del nodo, ya que cuanto más tenga, más paquetes de este tipo ser recibirán.

Los paquetes `HB` cumplen con otra funcionalidad, además de la anteriormente descripta, que consiste en dar la posibilidad de crear direcciones secundarias para conectar lógicamente a los nodos. Cuando este paquete es recibido, si los nodos no están conectados físicamente, se evalúa esta posibilidad, y de encontrar una dirección secundaria tal que el otro nodo sea adyacente, se ofrece mediante un paquete `SAP` (*Secondary Address Proposal*. La distribución para este tipo de paquetes se muestra en la figura 4.20.

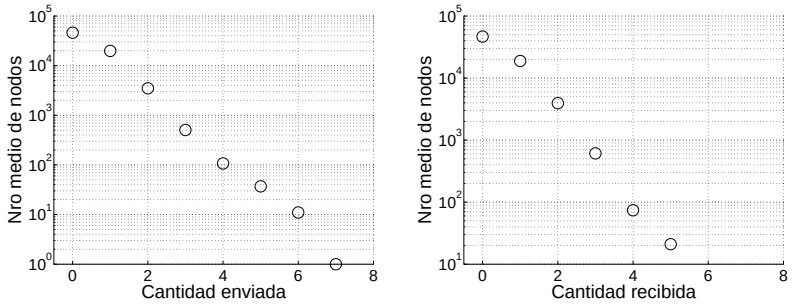


Figura 4.20: Paquetes `SAP`

Se obseva que la distribución de paquetes `SAP`, tanto enviados como recibidos, es igual a la distribución de direcciones secundarias por nodo mostrada en el apartado 4.1.4. Esto ocurre porque las direcciones secundarias son siempre aceptadas en esta implementación.

Los paquetes `SAP` se responden siempre con un paquete `SAN` (*Secondary Address Notification*), cuya distribución se muestra en la figura 4.21.
Tanto la cantidad recibida como enviada tienen la misma forma que la cantidad de paquetes `SAP` enviados.

Por último, se sumaron toods los paquetes recibidos y enviados, obteniendo las distribuciones mostradas en la figura 4.22.

Se observa que ambas distribuciones son similares a las de los paquetes `HB`, dado que este paquete es enviado muchas más veces que cualquiera del resto.

Esto implica que la energía consumida en la transmisión de paquetes está dada principalmente por el envío de estos paquetes. Si se envían con menos frecuencia, se utilizará menos energía, pero la red tardará más en adquirir direcciones secundarias y en detectar nodos que se desconectan.

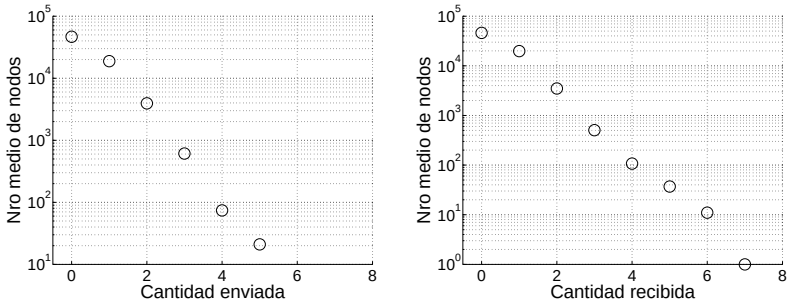


Figura 4.21: Paquetes `SAN`

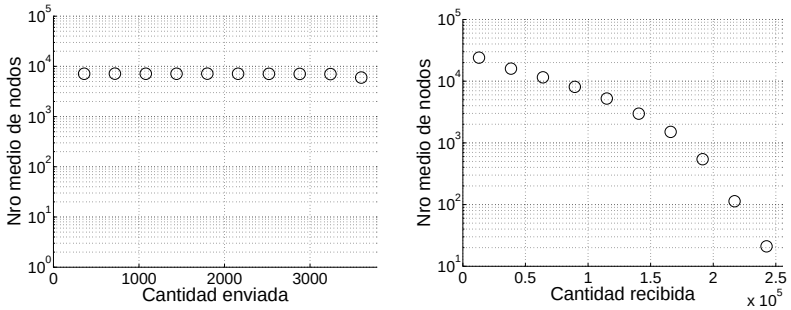


Figura 4.22: Total Paquetes

4.1.9. Envío de datos

Una vez que se encuentran todos los nodos conectados, se hacen pruebas de envío de datos. Para ello, se envían paquetes entre 1000 pares de nodos elegidos con probabilidad uniforme. Este primer envío es la etapa 1. Los nodos que reciben estos paquetes responden a quien envió el paquete, siendo esta la etapa 2, y finalmente los nodos que habían enviado los paquetes originalmente vuelven a enviar datos al mismo nodo, siendo ésta la etapa 3. Esto sirve para comparar los caminos y tiempos de ida y vuelta, y analizar que pasa al reutilizar un camino.

Además, estas pruebas de envío de datos se realizan primero sin carga, es decir, con las tablas de ruteo vacías. Entre envío y envío se deja transcurrir el tiempo suficiente para que se limpien todas las tablas. Luego se realizan estas pruebas con carga, enviando los datos con mayor frecuencia para que no se borren las tablas, con el fin de comparar ambos escenarios. Los resultados obtenidos fueron similares con y sin carga. La figura 4.23 muestra los resultados para una red de 7000 nodos con carga.

Las tres columnas representan cada una de las etapas.

En la primera fila se grafica un histograma del número medio de paquetes que recorren una cierta distancia (d_r).

Estos gráficos son muy similares para las etapas 1 y 3, indicando que la distancia recorrida en ambos casos es similar. Sin embargo, esta distancia recorrida no tiene en cuenta cuando los paquetes vuelven atrás, por lo que puede ocurrir que en la etapa 1 el paquete haga una exploración por diferentes caminos hasta encontrar uno que llega al destino, y que en la etapa 3 este camino sea utilizado. Al analizar los tiempos de estos envíos se obtendrá más información.

En la etapa 2, se observa que las distancias recorridas son un poco más cortas que en las otras dos etapas. Esto se puede deber a que al recorrer un camino, se deja rastro de la ruta inversa, indicando la distancia al origen.

La segunda fila de gráficos muestra como se distribuyen los tiempos. En este caso se observan grandes diferencias entre la etapa 1 y la etapa 3. Mientras que en la etapa 1 la probabilidad va aumentando hasta

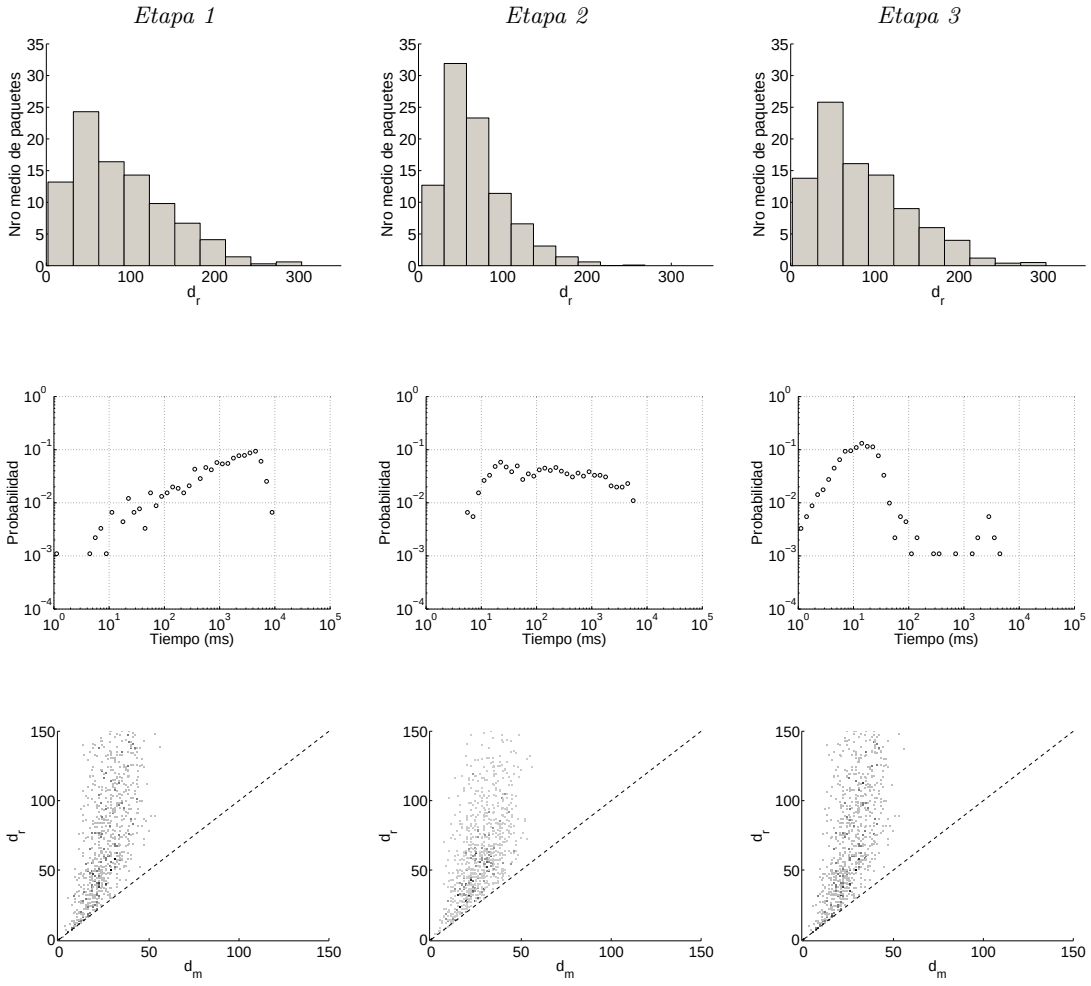


Figura 4.23: Envío de datos en las redes de 7000 nodos

3000 ms, en la etapa 3 la probabilidad aumenta hasta los 20 ms y luego disminuye rápidamente. Se puede ver que la gran mayoría de los paquetes tardan más de 100 ms en la etapa 1, pero la gran mayoría tarda menos de 100 ms en la etapa 3.

Es decir que si bien el camino recorrido tiene distancias similares para ambas etapas, los tiempos son mucho más altos cuando se se envía el paquete por primera vez, porque se exploran rutas hasta encontrar alguna que llega.

En la tercera fila se muestra la comparación de la distancia recorrida d_r con la distancia mínima d_m. La línea punteada indica la posición óptima, donde ambas distancias son iguales. Se observa que frecuentemente los caminos recorridos son 2 o 3 veces más largos que el camino mínimo.

En la figura 4.24 se grafican las distancias mínimas y recorridas cuando se efectúa la primera etapa del ruteo en función de la cantidad de nodos de la red, así como la relación entre ambas cantidades.

Se observa que para redes pequeñas las distancias son similares, siendo su relación menor a $1,5$ para redes de hasta 100 nodos, duplicando la distancia a partir de 700 nodos y triplicándola a partir de los 5000.

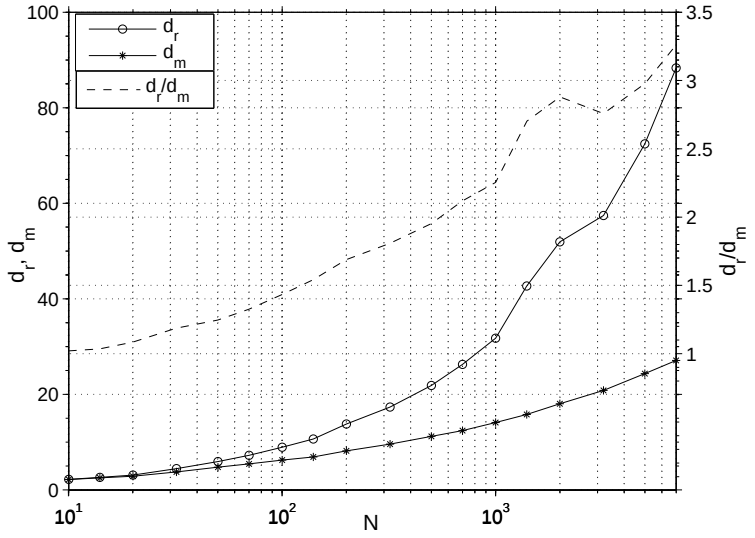


Figura 4.24: Distancia media en el ruteo

En la figura 4.25 se grafica, para el caso particular de las redes de 7000 nodos, la comparación de las distancias mínimas y recorridas y la distribución de la relación entre ellas.

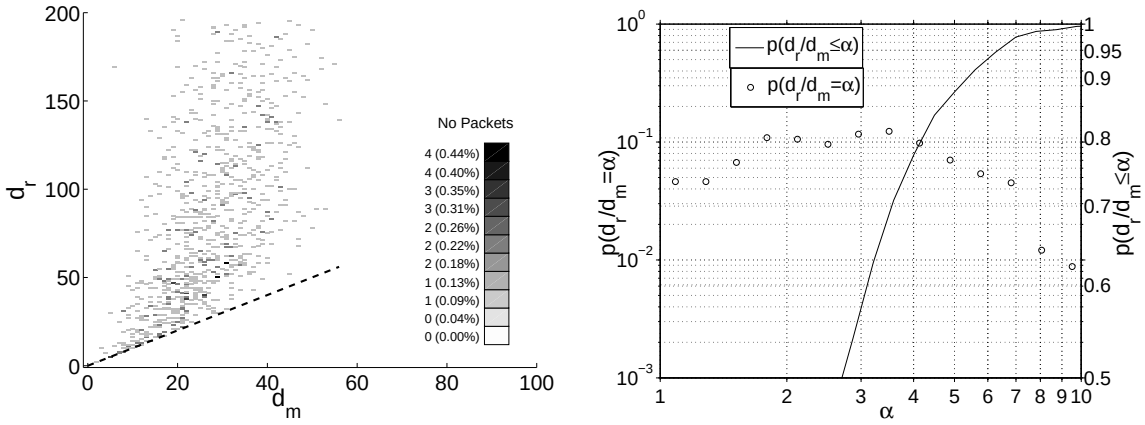


Figura 4.25: Distribución de la distancia de ruteo en una red de 7000 nodos

El gráfico de la izquierda compara las dos distancias, siendo la diagonal marcada con línea punteada el caso ideal en que ambas serían iguales. Cuanto más arriba de la diagonal se encuentre un sombreado, más grande es la relación entre las distancias.

El gráfico de la derecha muestra como se distribuye la probabilidad del cociente entre ambas distancias. La probabilidad de que d_r/d_m sea el valor indicado en la abscisa (α) se marca con círculos, con escala a la izquierda. La curva continuas es la probabilidad acumulada, es decir, la probabilidad de que el cociente d_r/d_m sea menor o igual al valor de la abscisa, encontrándose la escala a la derecha.

Aproximadamente el 60 % de los paquetes fueron ruteados con una distancia menor a tres veces la distancia mínima, y el 95 % fueron ruteados con una distancia menor a seis veces la distancia mínima.

4.1.10. Tamaño de las tablas de *Rendez-Vous*

Cada uno de los nodos almacena tablas de *Rendez-Vous* donde se asocian las direcciones universales con las direcciones de red. Dado que sólo se registran las direcciones primarias y que cada registración está almacenada en un solo *Rendez-Vous* hay tantas entradas de registraciones como nodos.

El caso ideal sería que las entradas se distribuyan equitativamente, quedando una en cada nodo. La figura 4.26 muestra la distribución obtenida para redes de 100 y 7000 nodos.

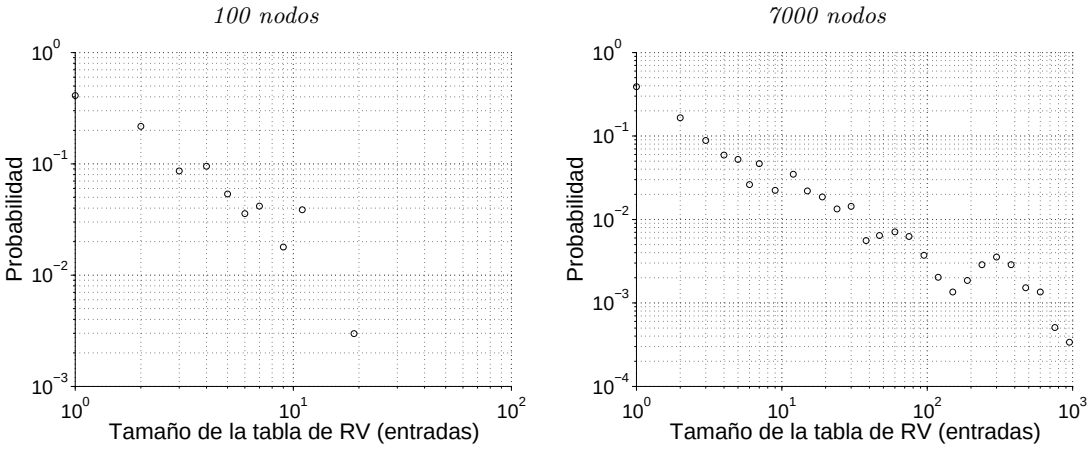


Figura 4.26: Tamaño de las tablas de *Rendez-Vous*

En ambos casos una gran cantidad de tablas son de tamaño 1, pero hay tablas de hasta 20 entradas en las redes de 100 nodos y hasta 1000 en las redes de 7000, aunque en una proporción muy pequeña.

El *Rendez-Vous* para el nodo se elije aplicando una función de hashing a la dirección universal de forma tal que el resultado sea una dirección de hipercubo. Luego, el nodo que tiene esa dirección en su espacio es el que recibe la entrada. Entonces, aquellos que tienen un espacio de direcciones más grande (máscara más corta o más direcciones secundarias) tienen mayor probabilidad de recibir entradas. Por ejemplo, un nodo con máscara 4 maneja un espacio de direcciones 64 veces más grande que uno con máscara 12 (el valor 64 se obtiene efectuando 2^{12-4}). Esto causa que los nodos con máscara más cortas tengan tablas de *Rendez-Vous* más grandes.

Este es uno de los puntos del protocolo a mejorar, buscando una forma para que el tamaño del espacio de direcciones no influya en el tamaño de la tabla de *Rendez-Vous*, pudiendo así mejorar su distribución.

4.1.11. Tamaño de las tablas de ruteo

Cuando los paquetes son ruteados se van generando entradas en las tablas de ruteo. La figura 4.27 muestra el tamaño medio de las tablas de ruteo (sobre las 10 series simuladas) luego de enviar paquetes entre 1000 pares de nodos elegidos con probabilidad uniforme, en función de la cantidad de nodos de la red N. Se muestran además las barras de desviación estandar.

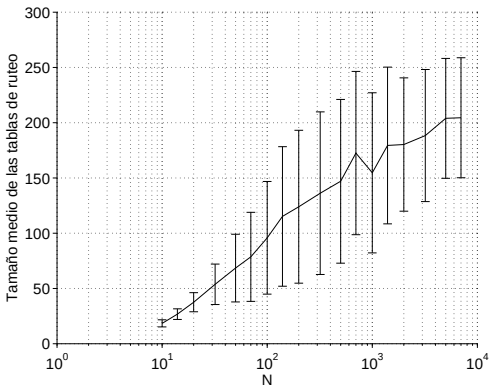


Figura 4.27: Tamaño medio de las tablas de ruteo en función del número de nodos de la red

Aunque la cantidad de paquetes enviados es igual para todas las redes, las tablas de ruteo tienen más entradas cuando las redes son más grandes, indicando que se necesita más información para efectuar el ruteo.

Tomando las redes de 100 y 7000 nodos, se agrupan aquellos nodos que tienen el mismo grado lógico y se hacen los promedios de los tamaños de la tabla de ruteo, obteniendo el resultado que muestra la figura 4.28. La primera fila muestra para cada grado cuál es el tamaño promedio de las tablas de ruteo, indicando con barras la desviación estandar, y en la segunda fila se grafica la cantidad de nodos promedio que tienen ese grado lógico.

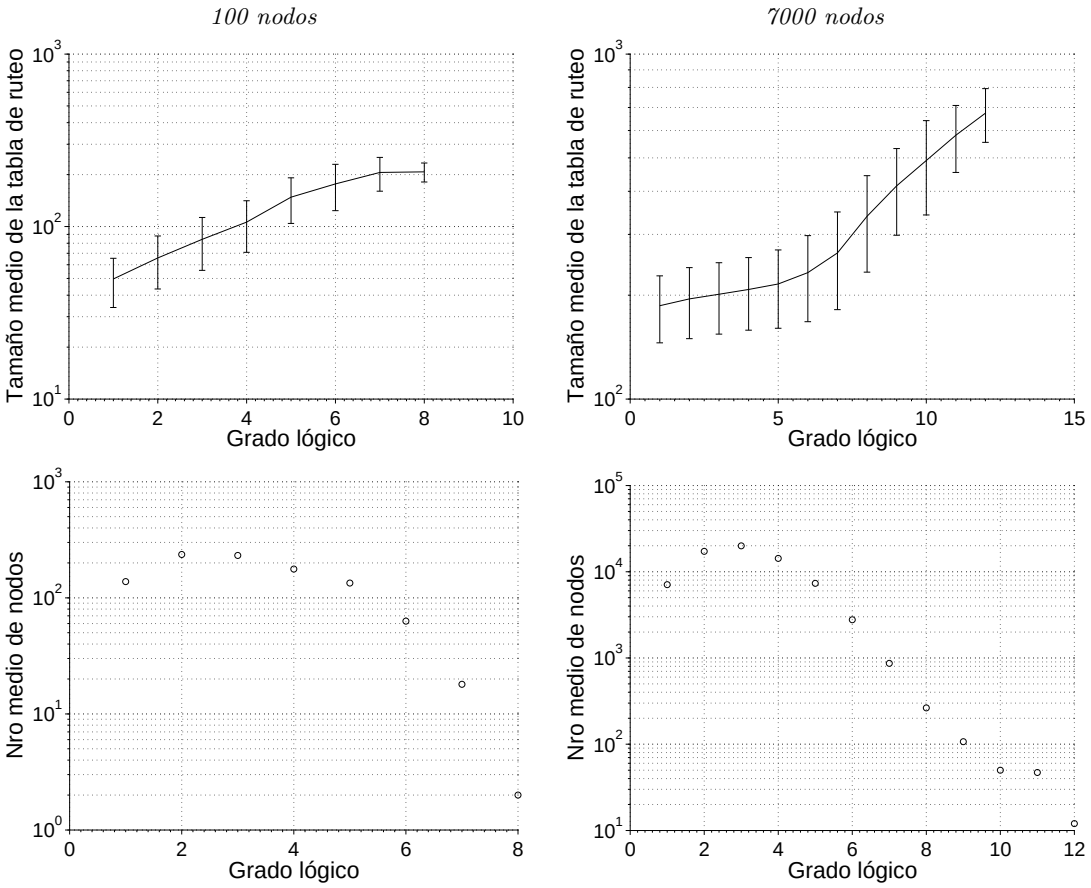


Figura 4.28: Tamaño de las tablas de ruteo por grado lógico

Las tablas de ruteo crecen cuando el grado lógico es mayor, debido a que al tener más conexiones, es más probable que los paquetes se ruteen a través del nodo.

La figura 4.29 muestra como se distribuyen los tamaños en las redes de 100 y 7000 nodos. En las redes de 100 nodos se observa un pico en las 80 entradas, habiendo una gran cantidad de nodos con tablas de ruteo de entre 40 y 100 entradas. En las redes de 7000 nodos el pico es más pronunciado, encontrándose en 200 entradas aproximadamente, y mostrando una gran concentración entre 100 y 300 entradas.

Con los tamaños de las tablas de ruteo y de *Rendez-Vous* se puede tener una estimación de la cantidad de memoria utilizada por el protocolo. En el caso de la red más grande (7000 nodos), el tamaño medio de memoria necesario para estas dos tablas es de aproximadamente 5 KB. Este valor proviene de considerar que se tienen 100 entradas en la tabla de *Rendez-Vous* y 200 en la tabla de ruteo, y que $d = 64$, por lo que cada dirección ocupa 8 bytes. También se considera que las direcciones universales ocupan en promedio 8 bytes cada una. Entonces, cada entrada de la tabla de *Rendez-Vous* tiene 8 bytes de la dirección universal y 8 bytes de la dirección de red, por lo que esta tabla ocupa $100 * (8 + 8) = 1600$ bytes.

Las entradas en la tabla de ruteo tienen dos direcciones de red de 8 bytes cada uno, más un byte de la distancia al origen y otro byte de los vecinos recibidos, totalizando entonces $200 * (8 + 8 + 1 + 1) = 3600$ bytes. Entre las dos tablas, suman poco más de 5 KB.

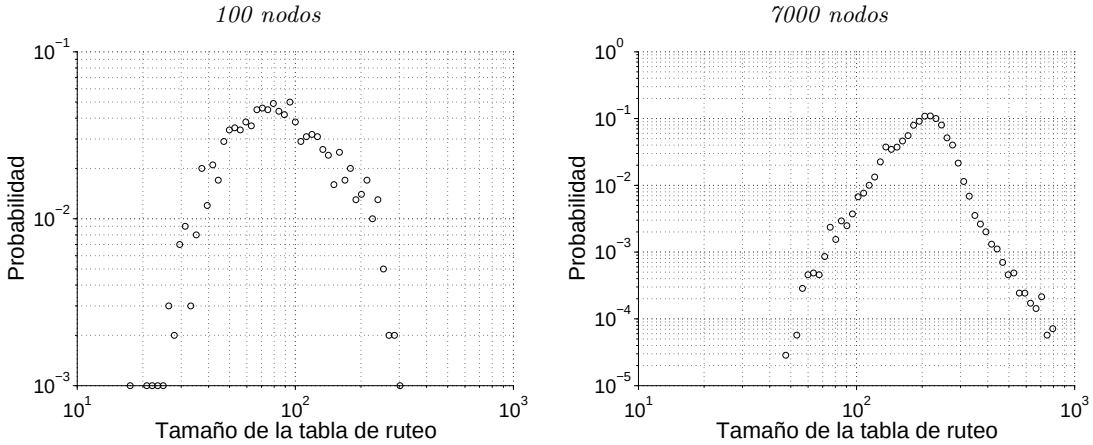


Figura 4.29: Distribución del tamaño de las tablas de ruteo

4.1.12. Conclusiones

Las simulaciones efectuadas en esta sección permitieron medir diversos aspectos del funcionamiento del protocolo en redes de entre 10 y 7000 nodos generadas aleatoriamente. Para ello se ubican nodos en posiciones distribuidas uniformemente en un plano, conectando aquellos que estén dentro de un radio predeterminado.

En todas las simulaciones efectuadas (180 en total), los nodos se pudieron conectar correctamente y se pudieron enviar paquetes entre pares aleatorios, lo cual muestra empíricamente el buen funcionamiento del protocolo.

Se encontró que la cantidad de conexiones promedio para cada nodo que logra establecer el protocolo es prácticamente independiente del tamaño de la red (aproximadamente 3), mismo para redes más grandes donde el número de vecinos conectados físicamente aumenta. Dado que dos nodos tienen que tener direcciones adyacentes para poder conectarse, no es frecuente que un nodo pueda conectarse lógicamente con una cantidad relativamente grande de vecinos. Esto abre un camino para futuras investigaciones, donde se podrían analizar otros modelos de adyacencia.

Al conectarse un nodo a la red, la obtención de dirección primaria resultó ser muy eficiente y poco dependiente del tamaño de la red, mientras que la registración en el *Rendez-Vous* necesaria para comenzar a operar ve disminuida su performance a medida que la red crece, aumentando el tiempo en forma lineal con la cantidad de nodos de la red.

En cuanto al ruteo, tanto en el caso de paquetes de *Rendez-Vous* como paquetes de datos, se comprobó que su eficiencia disminuye a medida que la red crece, recorriendo caminos más largos en relación con los caminos mínimos. Esta relación comienza siendo poco más que 1 en redes pequeñas, y con una red de 7000 nodos llega a 2 para la distancia al *Rendez-Vous* y más de 3 para el ruteo.

Además, en redes más grandes aumenta considerablemente el tiempo que necesita el primer paquete para llegar, ya que prueba más caminos. Sin embargo, una vez que se establecen las rutas, los tiempos de envío disminuyen notablemente, permitiendo un ruteo eficiente.

4.2. Redes con topologías sin escala

Un Sistema Autónomo (*Autonomous System, AS*) es un conjunto de redes y *routers* que operan bajo una única política de ruteo administrada en forma centralizada. Internet está compuesta por una red de Sistemas Autónomos conectados entre sí sin ninguna administración central salvo las especificaciones de protocolos [11]. La topología de la red de Sistemas Autónomos es un grafo de los denominados *sin escala*, es decir que la distribución de sus grados (número de vecinos) sigue una ley de potencias [4].

Las simulaciones realizadas en esta sección utilizan la topología real de los Sistemas Autónomos de Internet para saber como se desempeñaría `ANTop` en una topología sin escala. El mapa utilizado se obtuvo a partir de los datos publicados por *CAIDA* (*Cooperative Association for Internet Data Analysis)* [12], pertenecientes al mes de abril del 2005. Cada uno de los Sistemas Autónomos es considerado como un nodo, que al conectarse entre si forman la red de Sistemas Autónomos. Entre dos Sistemas Autónomos puede haber más de una conexión física, pero en el grafo se representarán mediante una arista, es decir, una sola conexión lógica.

Desarrollo de la simulación

La red de Sistemas Autónomos utilizada contiene poco más de 8500 nodos. Unos pocos tienen una gran cantidad de conexiones (varios miles), llegando hasta más de 7000 conexiones; pero muchísimos más tienen tan solo unas pocas conexiones. Más especificamente, la distribución de cantidad de conexiones sigue una ley de potencias (ver sección 4.2.1).

Para un mejor análisis de esta red se utilizan *k-cores* (ó k-nucleos), definidos de la siguiente forma: dado un grafo $G = \{V, E\}$, con un conjunto de vértices V y un conjunto de aristas E, el *k-core* se computa podando todos los vertices (con sus respecitvas aristas) con grado menor que k [5].
Es decir que para que un nodo esté en un *k-core*, debe tener al menos k vecinos que también tengan al menos k vecinos. Se denomina número de capa (*shell index*) al mayor *k-core* al que un nodo pertenece.

El *k-core* es una medida de la jerarquía de un nodo más concluyente que su cantidad de conexiones, ya que tiene en cuenta también la conectividad de los vecinos. De esta manera, se puede llamar *backbone* al *k-core* máximo, y los otros núcleos son a su vez *backbones* de menor importancia, hasta llegar al usuario final.

Los nodos de la red de Sistemas Autónomos se clasificaron según su número de capa, que varía entre 1 y 26, obteniendo mayor cantidad de nodos para los números de capa menores.

Para efectuar las simulaciones se programó un generador de simulaciones, que utiliza dos archivos de entrada: uno donde se encuentran los pares de Sistemas Autónomos que están conectados y otro que indica el número de capa de cada uno.

El generador de simulaciones comienza conectando los nodos de a uno, entregándoles sólo una dirección primaria, de forma tal que la red sea siempre conexa, conectando cada vez un nodo con el mayor número de capa posible. Esto se hace así para aprovechar mejor el espacio, ya que los nodos más conectados obtendrán máscaras más cortas y podrán ceder más direcciones, mientras que los menos conectados no desperdiciarán tanto espacio. Por ejemplo, si se utilizan direcciones de longitud 100 y el primer nodo que se conecta tiene solo una conexión, entonces el nodo solo cederá la dirección $100\ldots00/1$ y se quedará con el espacio $000\ldots00/1$, que cuenta con 2^{99} direcciones. En cambio, un nodo con 100 conexiones podría ceder todo su espacio y quedarse con solo una dirección.

Entonces, se escoge en primer lugar uno de los nodos con el número de capa máximo, se lo conecta a la red, y luego en cada paso se elige aleatoriamente un nodo con el máximo número de capa disponible que además esté conectado a uno de los nodos. Se procede de la misma forma hasta conectar todos los nodos.

Los Sistemas Autónomos utilizados tienen asignado un número que los identifica; sin embargo, para

facilitar la simulación y el análisis de resultados, se prefirió tener números correlativos. Por eso, a medida que se van conectando los nodos se les asigna un número secuencial, y se graban las equivalencias en un archivo para poder revertir el proceso.

Por ejemplo, una de las simulaciones generadas conecta los primeros nodos de la siguiente forma:

```
[100 ms] newNode(1).setHBEnabled(false)
node(1).joinNetwork()
[1100 ms] newNode(2).setHBEnabled(false)
newConnection(2,1,54 Mbps, 50us)
node(2).joinNetwork()
[2100 ms] newNode(3).setHBEnabled(false)
newConnection(3,2,54 Mbps, 50us)
newConnection(3,1,54 Mbps, 50us)
node(3).joinNetwork()
[3100 ms] newNode(4).setHBEnabled(false)
newConnection(4,3,54 Mbps, 50us)
node(4).joinNetwork()
```

Como se puede observar en este fragmento de simulación, al crearse los nodos se utiliza el comando `setHBEnabled(false)`, lo cual desactiva el envío de paquetes `HB` (*Heard Bit*), y por lo tanto no se asignan direcciones secundarias ni se descubren más adyacencias que las relaciones padre-hijo. Se debió realizar esto porque si se conectaban los nodos normalmente, las direcciones secundarias utilizaban parte del espacio y no se alcanzaba a conectar todos los nodos utilizando la máxima dimensión (255). De esta forma, se evita que se asignen direcciones secundarias al principio, logrando así conectar todos los nodos. Una vez que todos se encuentran conectados, se ejecuta el comando:

```
allNodes().setHBEnabled(true)
```

Este comando activa el envío de paquetes `HB`, que produce que se asignen direcciones secundarias y se creen conexiones lógicas cuando sea posible.

Luego de un lapso de tiempo para que se produzcan estas asignaciones, se escriben las siguientes consultas en el archivo de simulación:

- `allNodes.query()`: provee información de direcciones primarias y secundarias de cada nodo.
- `exportConnections(output/filename.csv)` genera un archivo con las conexiones establecidas.
- `allNodes.query(stats)`: provee estadísticas del envío de paquetes de control.

Dado que la registración de los nodos en el *Rendez-Vous* se produce a medida que estos se van conectando, estas ocurren cuando la red es un árbol, que como se explicó anteriormente, tiene ciertas particularidades como ser un solo camino para llegar de un nodo a otro.

Para ver como se comporta el ruteo de paquetes de registración de *Rendez-Vous* una vez que se produjeron las conexiones adicionales, se fuerza a cada uno de los nodos a enviar un paquete de registración mediante los comandos:

```
node(1).rendezVousServer.sendRegister()
node(2).rendezVousServer.sendRegister()
...
```

Cuando el *Rendez-Vous* reciba la registración, dado que esta ya existía será ignorada, pero escribirá información estadística acerca del ruteo de estos paquetes.

Luego, al igual que en la simulación de redes aleatorias, se mandan paquetes entre distintos pares de nodos, primero sin carga y luego con carga, registrando las longitudes de las rutas y los tiempos.

4.2.1. Grados lógicos y físicos de los nodos

Como se explicó en la sección 4.1.2, el grado físico de un nodo es la cantidad de conexiones que éste tiene, mientras que el grado lógico es la cantidad de conexiones que se logran aprovechar.

En la figura 4.30 se muestra en el gráfico de la izquierda como se distribuyen los grados lógicos y físicos, indicando la probabilidad de que cada nodo tenga un cierto grado. El gráfico de la derecha representa la probabilidad de que en un nodo el cociente entre el grado lógico y el físico tenga el valor dado por la abscisa.

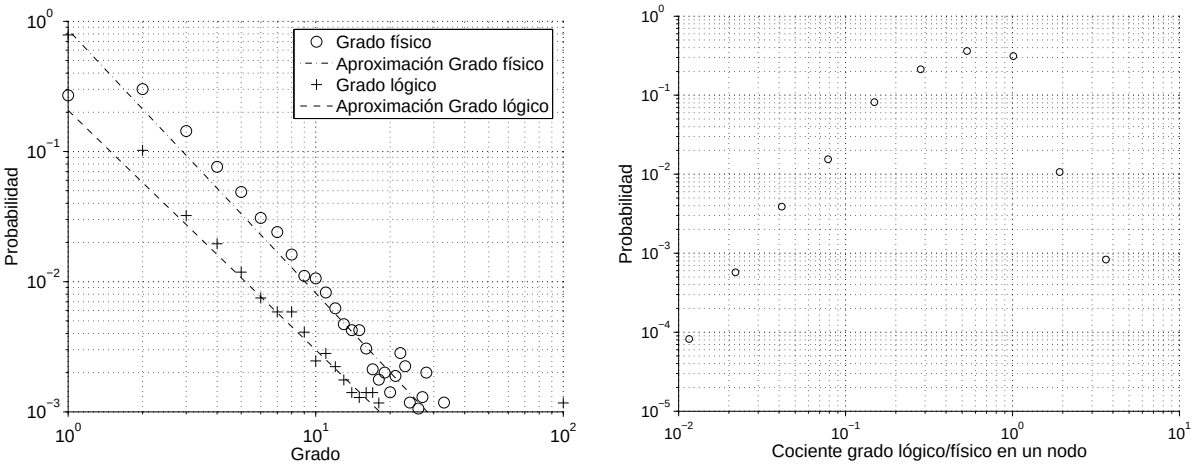


Figura 4.30: Grados lógicos y físicos

Las distribuciones de los grados, tanto físicos como lógicos, forman una recta con pendientes similares, por lo que se pueden aproximar mediante la ecuación:

$$\ln(p) = a + b\ln(g)$$

Donde p es la probabilidad, g es el grado y a y b son la ordenada al origen y pendiente de la recta respectivamente. Esta ecuación se puede reescribir como:

$$p = a' g^b$$

Siendo $a' = e^a$. Las aproximaciones obtenidas, que se muestran en lineas punteadas en el gráfico, son:

$$p = 0{,}8604 g^{-2{,}0227} \qquad \text{(grado físico)}$$

$$p = 0{,}2066 g^{-1{,}8389} \qquad \text{(grado lógico)}$$

Los errores cuadráticos medios de estas aproximaciones son 0,02 y 0,05 respectivamente. Las pendientes son similares, pero la del grado lógico decrece un poco más lento.

En cuanto al cociente entre los grados, se observa que la mayoría se encuentran en aproximadamente $0,45$, disminuyendo rápidamente a medida que decrece el cociente. Es decir que lo más común es que el grado físico sea poco más del doble que el grado lógico en un mismo nodo.

4.2.2. Máscaras de direcciones

La figura 4.31 muestra como se distribuyen las máscaras para las direcciones primarias y secundarias.

En el caso de las máscaras para direcciones primarias, se observa una gran cantidad entre 10 y 30, mientras que hay muy pocos nodos con máscaras menores a 10, y una distribución más uniforme de

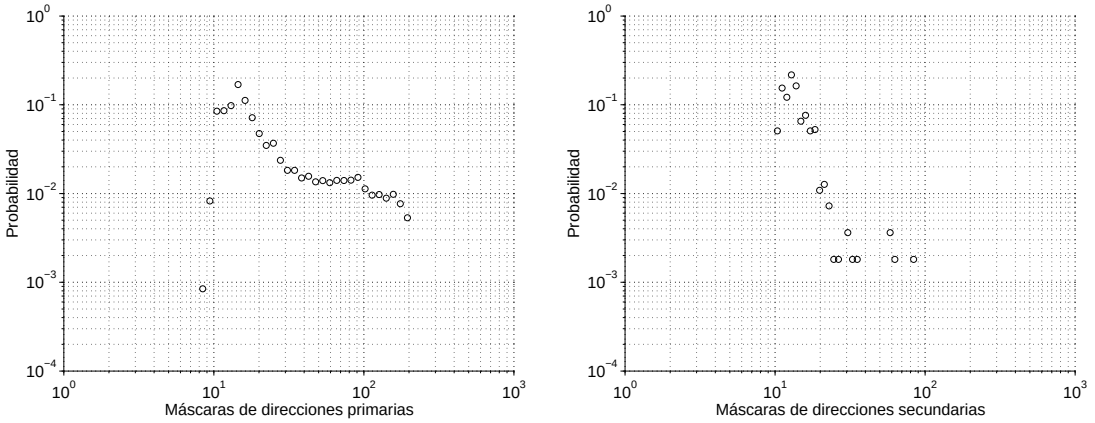


Figura 4.31: Distribución de las máscaras de direcciones

máscaras mayores a 30. Los primeros nodos en conectarse son los de número de capa más alto, y en principio obtienen las máscaras más cortas. Luego, a medida que otros nodos se van conectando, van adquiriendo direcciones de los nodos más conectados, por lo que éstos van incrementando sus máscaras. Los nodos con número de capa más bajo probablemente queden con una máscara similar a la que obtienen inicialmente, ya que ceden pocas direcciones, pero como no están muy conectados, tienen pocas posibilidades para elegir dirección.

Estos dos efectos combinados hacen que no haya una correlación importante entre el número de capa y la máscara final del nodo.

Las máscaras de direcciones secundarias muestran una distribución concentrada principalmente entre longitudes 10 y 20, lo cual es relativamente corto considerando que hay máscaras primarias de longitud 200. Esto indica que los espacios de direcciones secundarias son grandes, incrementando notablemente el espacio manejado por el nodo.

4.2.3. Registración en el *Rendez-Vous*

La figura 4.32 muestra los tiempos de registración en el *Rendez-Vous*, contando desde el momento en que se le da la orden de conexión al nodo.

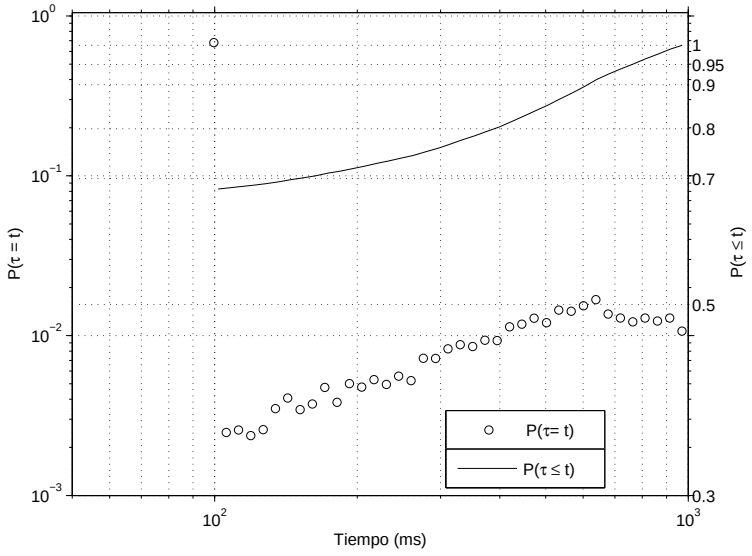


Figura 4.32: Tiempo de registración en el *Rendez-Vous*

Los círculos muestran la probabilidad de que la registración se produzca en un rango determinado de tiempo, mientras que la línea continua representa la probabildad de que la registración ya se haya producido en ese intervalo.

Una gran cantidad de registraciones se producen en aproximadamente 100ms, que es el tiempo mínimo posible. Casi el 70 % de los nodos se conectan en ese lapso, mientras que se necesitan 600ms para conectar al 90 % de los nodos y casi 1000ms para terminar de conectar a los restantes nodos.

La figura 4.33 muestra como se distribuye la distancia recorrida desde un nodo hasta el *Rendez-Vous* en el momento de la registración.

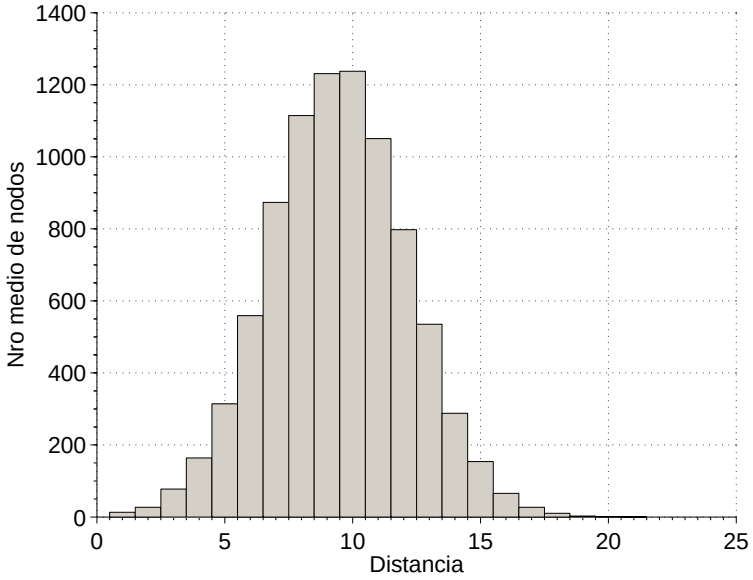


Figura 4.33: Distancia al *Rendez-Vous*

Durante el proceso de conexión de nodos los *Heard Bits* se encuentran desactivados, por lo que no se asignan direcciones secundarias ni se conectan vecinos adyacentes que no tengan una relación de padre-hijo, generando de esta forma un árbol.

Como entre dos puntos de un árbol existe sólo un camino, no tiene sentido comparar el camino encontrado por el ruteo con el camino mínimo ya que son iguales.

Sin embargo, una vez establecidas las conexiones secundarias, se forzó la realización de la registración nuevamente, para evaluar el desempeño del ruteo de *Rendez-Vous*. La figura 4.34 compara la distancia mínima d_m con la de ruteo d_r.

En el gráfico de la izquierda se muestra la cantidad de paquetes que recorrieron una cierta distancia de ruteo para cada distancia mínima, siendo el caso ideal que todos los sombreados se encuentren en la línea punteada.

Se observa una gran cantidad de paquetes que siguen caminos mucho más largos que el mínimo, y resulta llamativo el patrón de grises, donde se encuentran alternadamente colores claros y oscuros. Por ejemplo, para una distancia mínima de 10, hay muchos paquetes que la recorren con longitudes 10, 12, 14, 16, etc, mientras que con longitudes 11,13,15,etc hay muy pocos.

Esto se debe a la escasez de direcciones secundarias. En el caso límite de que no hubiera direcciones secundarias, el paso de un vecino a otro implica siempre el cambio de exactamente un bit, por lo que si se evalúa la cantidad de unos de la dirección, este valor es una vez par y otra impar alternadamente a medida que el paquete pasa de un nodo al vecino. Por ejemplo, para ir desde 0010 a 1001, se comienza teniendo una cantidad impar de unos; luego se podría ir al nodo 1010 (dirección con 2 unos, cantidad par), luego a 1000 (impar) y finalmente a 1001 (par). Es por ello que si se encuentra un camino de longitud d, es imposible encontrar uno de longitud $d+1$, ya que la paridad de la cantidad de unos en la

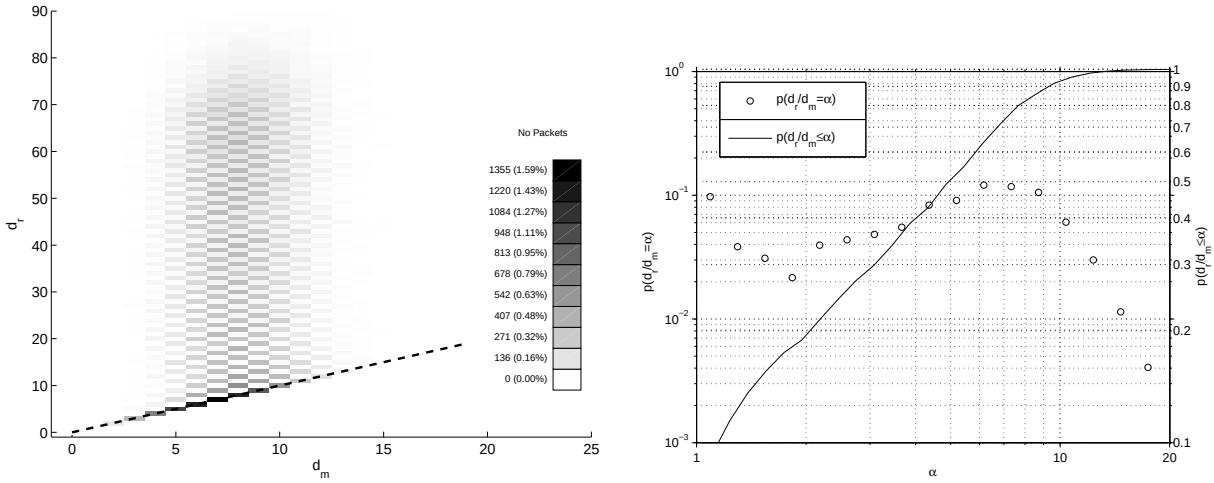


Figura 4.34: Comparación de la distancia al *Rendez-Vous*

dirección no coincide con la paridad de la dirección de destino.

En conclusión, en el caso de no utilizar direcciones secundarias, las distancias utilizando cualquier camino desde un nodo a otro son siempre pares o impares.

Para la red de Sistemas Autónomos, como hay pocas direcciones secundarias, se produce un efecto similar, aunque en ciertos casos estas direcciones son utilizadas y por ello se logra cambiar la paridad.

El gráfico de la derecha muestra la distribución de los cocientes entre la distancia de ruteo y la distancia mínima, siendo los círculos la probabilidad de que el cociente sea el valor en la abscisa, y la curva la probabilidad acumulada.

Se observa que el 50 % de los cocientes son menores que 5, y el 90 % menores que 10. Esto concuerda con el gráfico de la izquierda, donde se ve que muchos paquetes recorren distancias varias veces mayores a la distancia mínima.

4.2.4. Tamaño de las tablas de *Rendez-Vous*

La figura 4.35 muestra la distribución del tamaño de las tablas de *Rendez-Vous*.

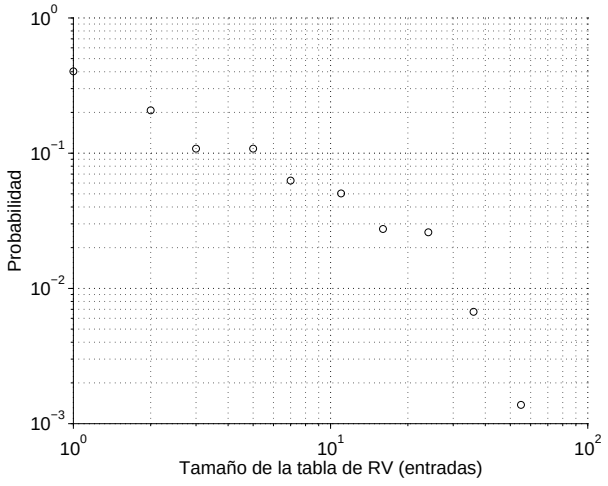


Figura 4.35: Tamaño de las tablas de *Rendez-Vous*

Se observa una gran cantidad de tablas de tamaño 1, que sería lo ideal, y que la cantidad decrece

al aumentar el tamaño, llegando a haber nodos con más de 50 entradas. Si bien esta cantidad no requiere mucha memoria o capacidad de procesamiento en el nodo, implica que habrá más resoluciones de *Rendez-Vous* en estos nodos, aumentando el tráfico.

Como se explicó en la sección 4.1.10, la causa de que haya nodos con tablas de *Rendez-Vous* mucho más grandes es que éstos manejan espacios de direcciones más grandes. Además, como las direcciones secundarias tienen máscaras cortas en estas redes, los nodos que tienen este tipo de direcciones incrementan aún más su espacio.

4.2.5. Paquetes de control

La figura 4.36 muestra la distribución de los paquetes de control enviados y recibidos para la obtención de dirección primaria.

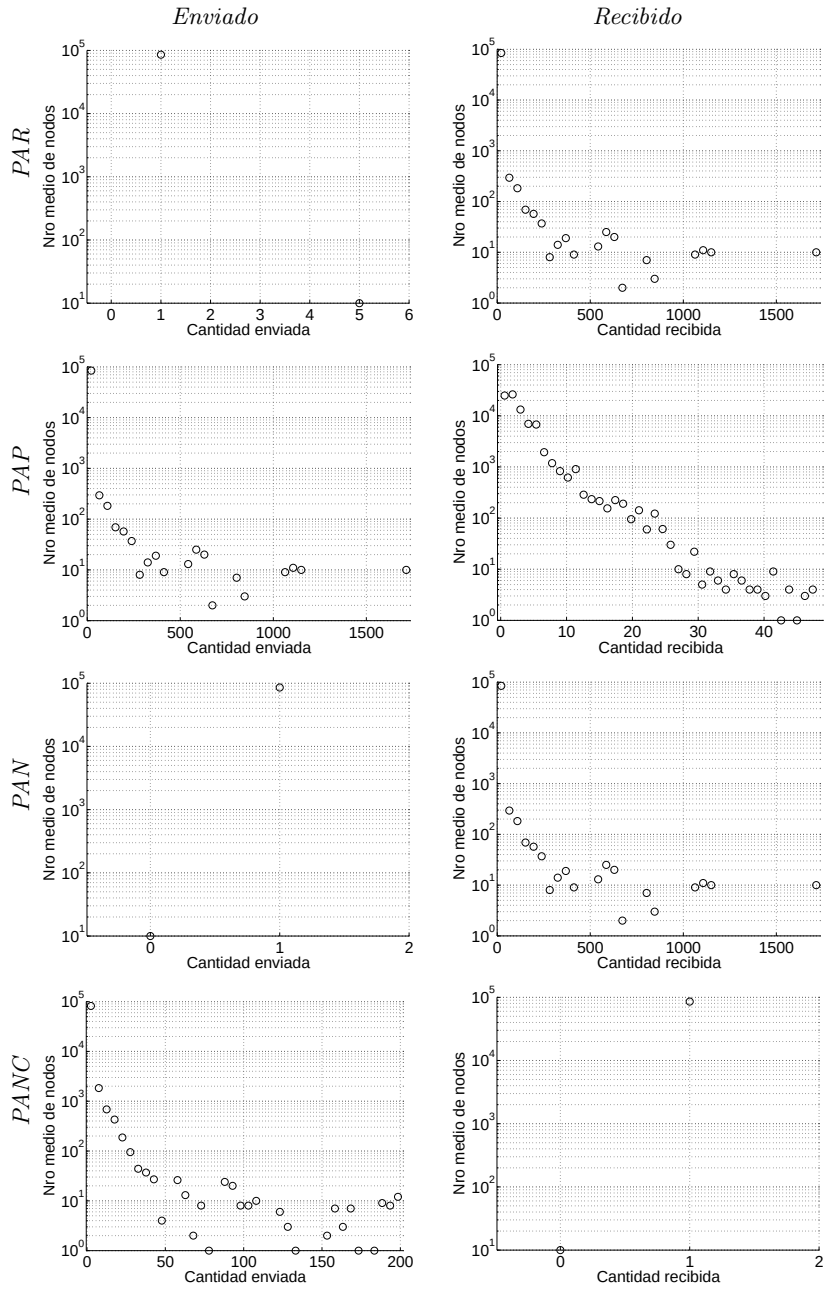


Figura 4.36: Paquetes de control para direcciones primarias

Los paquetes de tipo `PAR` son enviados una vez por cada nodo, excepto por el primero que lo envía cinco veces, luego de lo cual determina que es el primer nodo en la red. En el gráfico de recepción de paquetes `PAR`, se ve que algunos nodos reciben una gran cantidad de paquetes de este tipo, llegando algunos a recibir aproximadamente 1700. Estos son los nodos de número de capa más altos, que al conectarse primero y tener muchos de vecinos, reciben gran cantidad de pedidos.

Como cada paquete `PAR` es respondido por un paquete `PAP` proponiendo una dirección primaria o advirtiendo que no se dispone de más espacio de direcciones, la distribución de paquetes `PAP` enviados coincide con la de `PAR` recibidos.
La cantidad de paquetes `PAP` recibida indica cuantas direcciones se le propusieron a cada nodo. La mayoría tiene una sola, por lo que el nodo no tiene posibilidad de elección. Sin embargo, hay una gran cantidad de nodos que reciben 2,3,4 ó 5 propuestas, disminuyendo luego rápidamente.

Los paquetes de tipo `PAN` se envían para notificar la dirección primaria elegida, por lo que sólo un paquete de este tipo es enviado por nodo, excepto por el primero en conectarse que no envía ninguno. La distribución de paqeutes `PAN` recibidos es igual a la de `PAR` recibidos, ya que ambos se envían en modalidad broadcast en instantes de tiempo cercanos.

El último paso en la obtención de dirección primaria es el envío de un paquete `PANC` por parte del nodo que cede la dirección para indicar que la confirmación fue recibida y que se cedió la dirección. Por lo tanto, la distribución de paquetes `PANC` enviados coincide con la cantidad de direcciones primarias cedidas por cada nodo, es decir, la cantidad de hijos. Finalmente, los nodos reciben solo un paquete `PANC`, proveniente de su padre.

La figura 4.37 muestra la distribución de paquetes enviados y recibidos para obtención de direcciones secundarias.

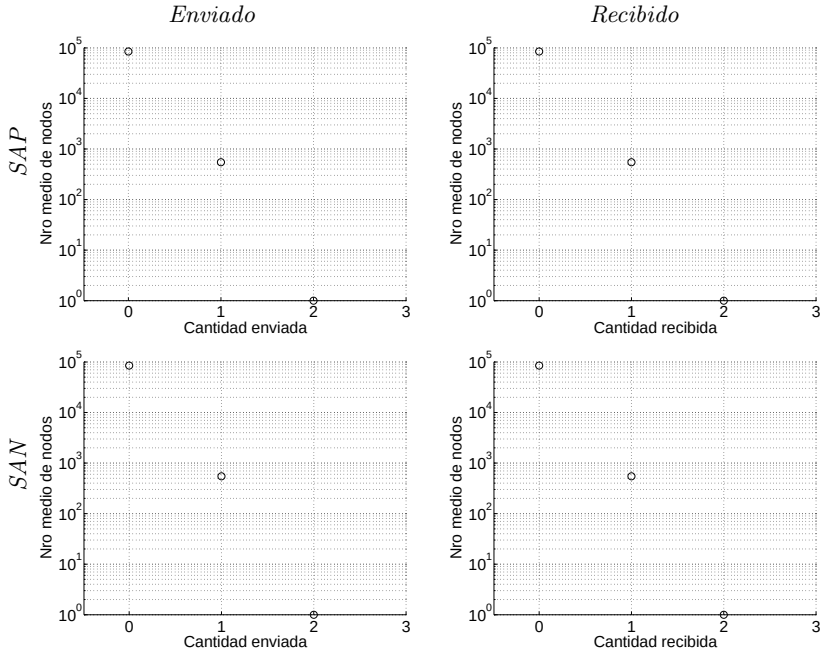


Figura 4.37: Paquetes de control para direcciones secundarias

Cuando un nodo recibe un paquete de tip `HB`, si tiene una dirección secundaria para proponer, envía un paquete `SAP`. Se observa que poco más de 500 nodos enviaron un paquete `SAP`, y muy pocos enviaron 2, generando muy pocas direcciones secundarias para una red de más de 8000 nodos. Esta misma distribución se repite para los paquetes `SAP` recibidos y para los paquetes `SAN`, tanto enviados como recibidos.

4.2.6. Envío de datos

Una vez conectados los nodos, se enviaron paquetes de prueba de la misma forma que para una red aleatorio (ver 4.1.9). La figura 4.38 muestra los resultados obtenidos para cada una de las 3 etapas (el origen envía un paquete, el destino responde y nuevamente el origen envía un paquete).

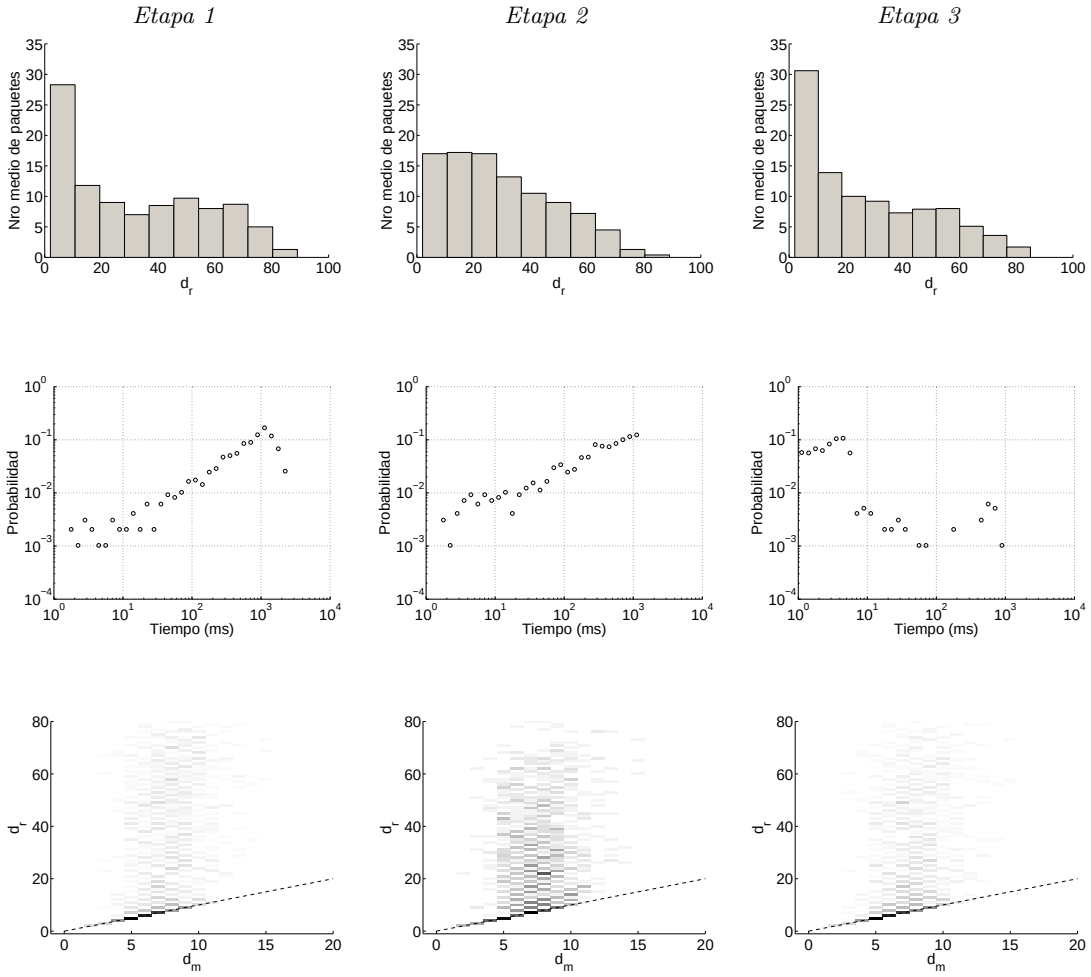


Figura 4.38: Envío de datos

La primera fila muestra como se distribuyen las distancias en cada una de las 3 etapas. En la primera y en la última etapa las distribuciones son muy similares, con una leve mejora en la última etapa, ya que las primeras dos barras indican una mayor cantidad de paquetes con distancias cortas. En la segunda etapa cambia principalmente la primera barra (longitudes entre 1 y 10 saltos), reduciendo su altura a la mitad. Esto indica que la mitad de los paquetes que van con distancias de hasta 10 saltos vuelven por un camino más largo.

En la segunda fila se muestra como se distribuye el tiempo que tardan en llegar los paquetes. Las etapas 1 y 2 son similares, aunque la primera muestra algunos paquetes que tardan más de un segundo en llegar. En cambio, la etapa 3 tiene una distribución muy diferente, que muestra que los paquetes llegan mucho más rápido una vez armada la ruta, dado que la mayoría llegan en menos de 10 ms, mientras que en las otras dos etapas se ve una gran cantidad de paquetes que tardan más de 100 ms.

La última fila compara la distancia recorrida d_r con la distancia mínima d_m. Se aprecia en todos estos gráficos un patrón alternado, que, como se explicó en 4.2.3, se debe a la escasa cantidad de direcciones secundarias. Igual que en la primera fila, las etapas 1 y 3 son similares, con una leve mejora en la etapa 3. En la etapa 2 se puede apreciar que hay más secciones sombreadas sobre la diagonal, principalmente

sobre las distancias mínimas 5 y 6, indicando que las distancias recorridas en estos casos son más largas aún que en las otras etapas.

Utilizando los valores de distancia de la primera etapa, en la figura 4.39 se graficó a la izquierda la comparación entre las distancias, y a la derecha la distribución de probabilidades de que el cociente obtenga ciertos valores, así como la probabilidad acumulada.

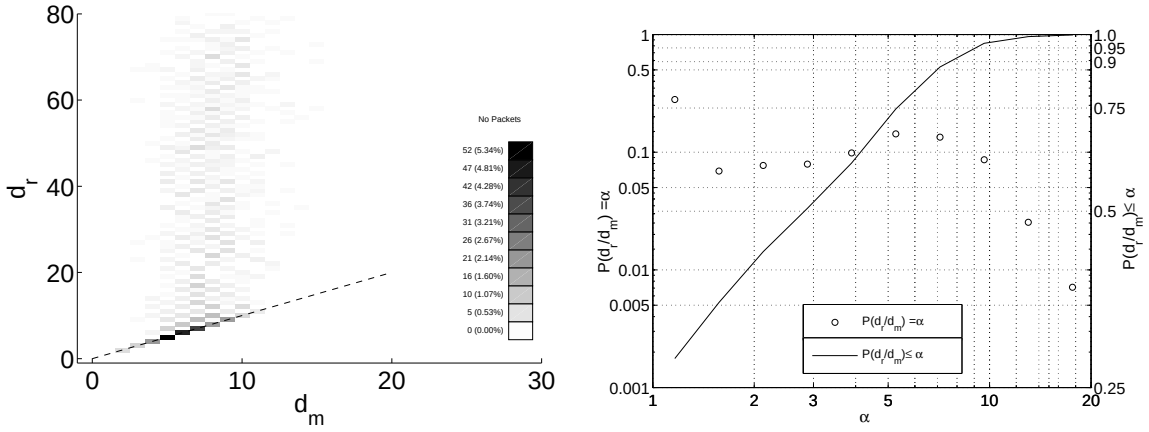


Figura 4.39: Distribución de la distancia de ruteo en una red de 7000 nodos

Se observa que poco más del 50 % de los paquetes enviados tienen una relación mayor a 3 y el 90 % tiene una relación menor a 8, quedando más del 95 % de los paquetes con una relación menor a 10.

4.2.7. Tamaño de las tablas de ruteo

La figura 4.40 muestra como se distribuyen los tamaños de las tablas de ruteo luego de enviar paquetes entre 1000 pares de nodos aleatorios.

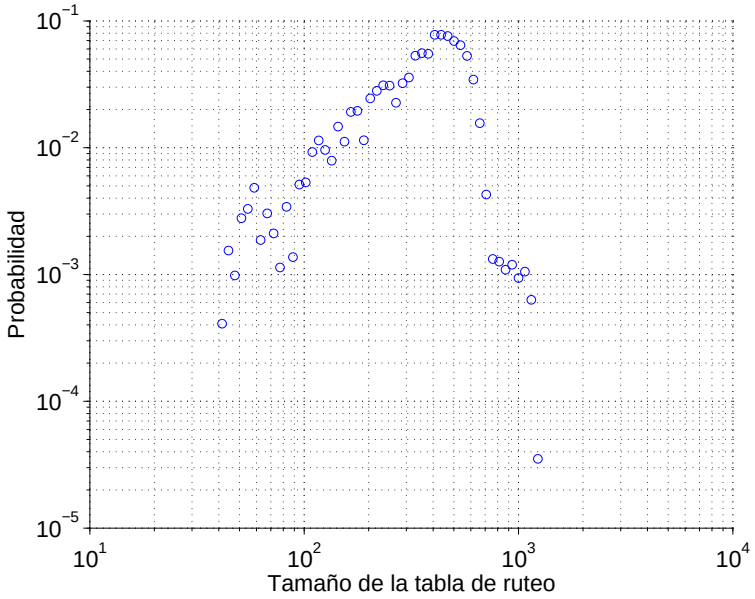


Figura 4.40: Tamaño de las tablas de ruteo

Se observa una alta concentración de nodos cuyas tablas de ruteo tienen alrededor de 500 entradas, disminuyendo luego rápidamente, y habiendo muy pocas tablas con más de 1000 entradas para un tráfico

entre 1000 pares de nodos elegidos con probabilidad uniforme.

Debido a que la red de Sistemas Autónomos tiene una estructura bastante jerárquica, donde algunos nodos tienen una gran cantidad de conexiones, es de esperar que éstos sean atravesados frecuentemente por los paquetes en su ruta, generando tablas de ruteo más grandes.

4.2.8. Conclusiones

Las redes de Sistemas Autónomos siguen una ley de potencias, es decir que pocos nodos tienen muchas conexiones y muchos nodos tienen pocas conexiones, habiendo además valores intermedios. En cambio, `ANTop` modela a la red como un hipercubo donde en el caso óptimo todos los nodos tendrían la misma cantidad de vecinos. Sin embargo, se probó su funcionamiento a pesar de que las topologías son inherentemente diferentes.

Se conectaron los nodos comenzando por aquellos con mayor cantidad de vecinos y procediendo en forma decreciente. Para poder completar la conexión de todos los nodos sin que se agotara el espacio de direcciones se realizó toda la etapa de conexión sin otorgar direcciones secundarias, evitando de esta forma emplear más espacio del necesario. Una vez finalizadas todas las conexiones se activaron las direcciones secundarias.

La distribución de conexiones lógicas obtenidas resultó ser, al igual que las conexiones físicas, una distribución de ley de potencias, aunque con grados menores.

Se observó que se asignaron relativamente pocas direcciones secundarias, debido en parte a que estas se asignaron al final de la construcción de la red y no a medida que se va conectando como es lo habitual, y en parte a la topología de este tipo de redes.

Al igual que en las redes aleatorias con muchos nodos, el ruteo, tanto de paquetes de *Rendez-Vous* como de datos, utiliza caminos varias veces más largos que los caminos mínimos, y también los tiempos de ruteo son largos en el primer envío pero disminuyen considerablemente al utilizar una ruta ya establecida.

Capítulo 5

Conclusiones

En el curso de esta tesis se partió del trabajo [1], donde se describen diversos aspectos de un protocolo de redes *ad-hoc* basadas en un hipercubo, dando los lineamientos generales. Este protocolo consiste en tres partes principales: conexión y desconexión de nodos, ruteo reactivo y registro de direcciones. Utilizando esta base, se trabajó en la especificación completa de cada una de las tres partes, obteniendo como resultado los formatos de paquetes, los diagramas de estado y los algoritmos correspondientes, denominándolo `ANTop` (*Adjacent Network Topologies*)

Luego, se programó un simulador de redes (`QUENAS`, por *Queued Event Network Automatic Simulator*), capaz de interpretar un lenguaje diseñado específicamente para este fin y producir una salida detallando lo ocurrido a lo largo de la simulación. El simulador permite crear una red compuesta por nodos y conexiones, donde cada nodo implementa las capas del modelo híbrido de capas [7] utilizando `ANTop` en la capa de red en lugar de `IP`, así como un cliente y servidor de *Rendez-Vous* en la capa de aplicación y una aplicación de envío de datos para hacer pruebas.
Si bien el desarrollo del simulador se orientó a simular `ANTop`, fue diseñado para ser flexible y poder ser reutilizado para otras simulaciones, tanto en futuras versiones de `ANTop` como para otros protocolos completamente distintos. Por ello, se puso el código fuente a disposición de la comunidad *Open Source*, pudiendo acceder libremente a él desde Internet [15].

Una vez completado el simulador, se hicieron distintas pruebas de los protocolos de ruteo (datos y *Rendez-Vous*) propuestos en [1]. Se observaron algunos problemas que provocaban la pérdida de paquetes. Analizando los problemas uno a uno se determinaron los algoritmos 7 y 8, que funcionaron correctamente, dando resultados positivos.

Luego se desarrollaron dos simulaciones para evaluar el funcionamiento del protocolo. La primera simulación consistió en redes generadas aleatoriamente, utilizando 18 tamaños de red distintos, entre 10 y 7000 nodos, y efectuando series de 10 simulaciones en cada caso. Todas estas simulaciones concluyeron exitosamente, pudiendo conectar todos los nodos con una dirección válida, y siendo capaz de enviar datos entre distintos puntos sin pérdidas. En cuanto a la eficiencia, se observó un buen rendimiento del protocolo, pudiendo efectuar las conexiones y registraciones en tiempos razonables. En el envío de paquetes se observó una disminución de la eficiencia para redes más grandes.

Según las estimaciones efectuadas, utilizando 255 bits para las direcciones, que es lo máximo que el protocolo propone, se podrían conectar aproximadamente 31 millones de nodos en una red aleatoria. Sin embargo, las estimaciones para el tiempo de registración en el *Rendez-Vous* ponen una cota mucho menor, ya que con esa cantidad de nodos, algunos de ellos tardarían varias horas en registrarse. Tomando como máximo aceptable para la registración 10 segundos, la cota disminuye a 30.000 nodos.

El espacio de almacenamiento en memoria necesario para las tablas de ruteo y *Rendez-Vous* es de aproximadamente 5 kb para la red de 7000 nodos, indicando un uso eficiente del espacio.

La segunda simulación se realizó sobre una red sin escala, basada en la topología de los Sistemas

Autónomos que componen Internet, tomando cada uno de ellos como un nodo y haciendo las interconexiones correspondientes a partir de datos reales. Esta red contiene poco más de 8500 nodos, cuyas conexiones siguen una distribución de ley de potencia, es decir que hay pocos nodos con muchas conexiones y muchos nodos con pocas conexiones. Para poder completar la simulación se debió prescindir de direcciones secundarias durante la conexión, ya que este tipo de direcciones consumía parte del espacio, agotándolo antes de que todos los nodos se pudieran conectar.

Para ambas simulaciones la gran mayoría de los paquetes de control transmitidos son del tipo *Heard Bit*, por lo que la frecuencia de transmisión de este paquete se debe ajustar para obtener un buen balance entre energía consumida por envío de paquetes y velocidad de respuesta de la red ante cambios de topología.

El resultado final de esta tesis es un protocolo completamente especificado y que funciona correctamente, así como un simulador de redes que podrá ser reutilizado para éste u otro protocolo.

5.1. Trabajos futuros

El desarrollo de `ANTop` puede ser continuado, mejorando sus prestaciones y buscando su utilización práctica.

Para ello, se proponen los siguientes puntos para continuar con este trabajo:

- *Unión y partición de redes*: por el momento, el protocolo no soporta que se creen dos redes por separado y luego se unan, ni tampoco que se desconecten dos nodos de forma tal que la red quede dividida en dos partes. El desarrollo de estos puntos no resulta trivial, pero sería de gran utilidad para redes reales.
- *Mejora en la asignación de Rendez-Vous*: debido a la forma de obtención del *Rendez-Vous*, aquellos nodos cuyo espacio de direcciones es más grande tienen tablas más grandes, que a su vez provoca que sean consultados más frecuentemente, aumentando el tráfico en ellos. Sería conveniente encontrar una forma de evitar la correlación entre el tamaño del espacio de direcciones y el de la tabla de *Rendez-Vous*.
- *Mejora de las adyacencias*: se observó en la simulación de redes aleatorias que por más que el número de conexiones físicas de los nodos aumente, las conexiones establecidas por el protocolo no aumentan. Esto se debe en parte al esquema de adyacencias, que requiere que haya a lo sumo un bit de diferencia entre dos direcciones para que el protocolo pueda establecer la conexión. Se podrían utilizar otros esquemas, como por ejemplo permitir que haya n bits de diferencia entre dos direcciones adyacentes.
- *Asignación de direcciones*: en la simulación de redes sin escala se observó que si se utilizaban las direcciones secundarias, el espacio de direcciones se agotaba antes de conectar todos los nodos. Se podría analizar un esquema de asignación de direcciones para el caso de nodos poco conectados donde estos reciban tan sólo una dirección en lugar de un espacio de direcciones, permitiendo de esta forma mejorar la distribución.
- *Ruteo proactivo*: en esta tesis se trabajó únicamente con el ruteo reactivo, pero es de interés también especificar y simular el ruteo proactivo y comparar ambos en distintas condiciones.
- *Tiempo de registración*: el tiempo de registración en el *Rendez-Vous* está limitando la cantidad de nodos, ya que este tiempo crece linealmente con la cantidad de nodos, y se hace inaceptable cuando esta cantidad supera los 30.000 nodos. Se debe optimizar el proceso de ruteo en la registración para obtener tiempos menores y poder utilizar redes más grandes.
- *Demostración de algoritmos*: demostrar formalmente el correcto funcionamiento de todos los algoritmos del protocolo.
- *Implementación real*: el protocolo puede ser implementado sobre un sistema operativo para poder utilizarlo en casos reales.

Bibliografía

[1] J. I. Alvarez-Hamelin, A.C. Viana, and M.D. de Amorim.
DHT-based Funcionalities Using Hypercubes.
In K. Al Agha, editor, *IFIP Ad-Hoc Networking*, pages 157–176. Boston: Springer, 2006.

[2] Jakob Eriksson, Michalis Faloutsos, and Srikanth Krishnamurthy.
Peernet: Pushing peer-to-peer down the stack.
In *IPTPS*, 2003.

[3] I. Jacobson G. Booch, J. Rumbaugh.
El lenguaje unificado de modelado.
Addison Wesley, 1999.

[4] R. Pastor-Satorras and A. Vespignani.
Evolution and structure of the Internet: A statistical physics approach.
Cambridge University Press, 2004.

[5] S. B. Seidman.
Network structure and minimum degree. social networks 5.

[6] Bjarne Stroustrup.
El lenguaje de programación C++.
Addison Wesley, 1998.

[7] Andrew S. Tanenbaum.
Computer Networks.
Prentice Hall Professional Technical Reference, 2003.

[8] Aline Carneiro Viana, Marcelo Dias de Amorim, Serge Fdida, and José Ferreira de Rezende.
An underlay strategy for indirect routing.
Wirel. Netw., 10(6):747–758, 2004.

[9] Ilustración de un hipercubo de dimensión 4: http://fr.wikipedia.org/wiki/image:hypercube.svg.

[10] Sitio web de *Exteme Programming*, http://www.extremeprogramming.org/.

[11] Sitio web de *RFC Editor*: http://www.rfc-editor.org/.

[12] *CAIDA, the Cooperative Association for Internet Data Analysis*, http://www.caida.org/data/.

[13] *doxygen* en *sourceforge*, http://sourceforge.net/projects/doxygen/.

[14] *Extensible Markup Language (XML)*, http://sourceforge.net/projects/quenas.

[15] *QUENAS* en *sourceforge*, http://sourceforge.net/projects/quenas.

[16] *XSL Transformations (XSLT)*, http://www.w3.org/tr/xslt.