Especificación y análisis del protocolo Chord en Maude

SARA MANCHADO ILLÁN

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Programación y Tecnología Software

10 de Septiembre de 2012

Director: José Alberto Verdejo López
Colaboradora: María Isabel Pita Andreu

Autorización de Difusión

SARA MANCHADO ILLÁN

10 de Septiembre 2012

La abajo firmante, matriculada en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Especificación y análisis del protocolo Chord en Maude", realizado durante el curso académico 2011-2012 bajo la dirección de Jose Alberto Verdejo López y de María Isabel Pita Andreu en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

Los sistemas *peer-to-peer* estructurados basan su funcionamiento en la implementación de una Tabla Hash Distribuida (THD), entre las cuales destacan: Kademlia, Chord, CAN y Pastry. Este documento presenta la especificación en Maude de un sistema *peer-to-peer* implementado con la Tabla Hash Distribuida Chord. Chord organiza su espacio de identificadores utilizando un anillo, donde cada nodo es responsable de un conjunto de claves las cuales se reasignan cuando un nodo entra o sale a la red. Además, cada nodo mantiene una tabla de encaminamiento, denominada tabla *finger*, utilizada para llevar a cabo el algoritmo de búsqueda. La especificación implementada ha sido diseñada para la compartición de archivos, de forma que los nodos puedan conectarse de la red, desconectarse de la red, publicar un fichero o buscar un fichero en un máximo de log *n* saltos, siendo *n* el número de nodos que conforman la red.

Palabras clave

Chord, Maude, especificación formal, sistemas distribuidos, *peer-to-peer,* Tabla Hash Distribuida.

Abstract

Distributed Hash Tables (DHT) is an essential component in most peer-to-peer systems. A large number of DHTs have been studied through theoretical simulations and analyses over the last years, such as: Kademlia, Chord, CAN, and Pastry. This paper presents the distributed specification of a peer-to-peer network that uses the Chord DHT in the formal specification language Maude. Identifiers are ordered on the Chord ring. A key is assigned to the first node whose identifier is equal to or follows the key in the identifier space. Each node maintains a routing table, called the *finger* table, which is used by the lookup algorithm. The contribution of this paper is a specification for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures, which guarantees that the lookup algorithm takes no longer than log n steps.

Keywords

Chord, Maude, formal analysis, distributed Systems, peer-to-peer, distributed hash table.

Índice de contenidos

Autorización de Difusión iii

Resumen iv

Palabras clave iv

Abstract v

Keywords v

Índice de contenidos 1

Índice de figuras 5

Índice de tablas 7

Agradecimientos 8

Capítulo 1 - Introducción 9

Capítulo 2 - Sistemas peer-to-peer 12

2.1 Sistemas *peer-to-peer* no-estructurados 13

2.1.1 Modelo peer-to-peer centralizado *13*

2.1.2 Modelo peer-to-peer puro o totalmente descentralizado *14*

2.1.3 Modelo peer-to-peer híbrido o semicentralizado *14*

2.2 Sistemas *peer-to-peer* estructurados 15

2.2.1 Ataques en los sistemas peer-to-peer estructurados *17*

Capítulo 3 - Tablas Hash Distribuidas 20

3.1 Kademlia 20

3.1.1 Asignación de claves a nodos *20*

3.1.2 Tabla de encaminamiento *21*

3.1.3 Protocolo Kademlia *23*

3.1.4 Algoritmo de búsqueda *24*

3.1.5 Almacenar un par <clave, valor> *28*

3.1.6 Incorporar un nodo a la tabla *28*

3.1.7 Dar de baja un nodo de la tabla *28*

3.1.8 Aplicaciones ... *29*
3.1.9 Ataques ... *30*
3.1.10 Conclusión ... *31*
3.2 Chord ... 31
3.2.1 Asignación de claves a nodos ... *31*
3.2.2 Tabla de encaminamiento ... *33*
3.2.3 Lista de sucesores y predecesores ... *34*
3.2.4 Protocolo Chord ... *34*
3.2.5 Algoritmo de búsqueda ... *35*
3.2.6 Protocolo de estabilización ... *36*
3.2.7 Incorporar un nodo a la red ... *37*
3.2.8 Dar de baja un nodo de la red ... *38*
3.2.9 Aplicaciones ... *38*
3.2.10 Ataques ... *39*
3.2.11 Conclusión ... *39*
3.3 Pastry ... 40
3.3.1 Asignación de claves a nodos ... *40*
3.3.2 Tabla de encaminamiento ... *41*
3.3.3 Lista de nodos vecinos ... *42*
3.3.4 Lista de nodos hoja ... *42*
3.3.5 Algoritmo de búsqueda ... *42*
3.3.6 Incorporar un nodo a la tabla ... *44*
3.3.7 Dar de baja un nodo de la tabla ... *45*
3.3.8 Protocolo Pastry ... *45*
3.3.9 Aplicaciones ... *46*
3.3.10 Conclusión ... *47*
3.4 Content-Addressable Network (CAN) ... 48
3.4.1 Asignación de claves a nodos ... *48*
3.4.2 Tabla de encaminamiento ... *49*
3.4.3 Algoritmo de búsqueda ... *50*
3.4.4 Incorporar un nodo a la tabla ... *51*
3.4.5 Dar de baja un nodo de la tabla ... *52*
3.4.6 Mejoras del sistema ... *53*

3.4.7 Aplicaciones *54*
3.4.8 Conclusión *54*
3.5 Otras THDs 54
3.5.1 EpiChord *55*
3.5.2 Viceroy *56*
3.5.3 Kelips *57*
3.6 Comparativa de las Tablas Hash Distribuidas estudiadas 59
Capítulo 4 - Especificación del protocolo Chord en Maude 61
4.1 El lenguaje de especificación Maude 61
4.2 Detalles de la especificación 65
4.3 Módulos de la especificación 68
4.3.1 BitString *68*
4.3.2 Node-ID *71*
4.3.3 Node-IP *71*
4.3.4 Info-Node *71*
4.3.5 KeysTable *72*
4.3.6 FingerTable *73*
4.3.7 SuccessorList *74*
4.3.8 PredecessorList *75*
4.3.9 PeerOperations *75*
4.3.10 Peer *77*
4.3.11 Init *82*
4.4 Procesos básicos entre nodos 82
4.4.1 Incorporar un nodo a la red *83*
4.4.2 Eliminar un nodo de la red *86*
4.4.3 Proceso de estabilización *89*
4.4.4 Publicar un fichero en la red *91*
4.4.5 Eliminar la clave de un fichero *92*
4.4.6 Búsqueda de una clave *94*
4.4.7 Nodo comprueba su predecesor *96*
Capítulo 5 - Análisis del protocolo 98
5.1 Configuración inicial 98

5.2 Resultado de los procesos 104
5.2.1 Incorporar un nodo a la red *106*
5.2.2 Eliminar un nodo de la red *108*
5.2.3 Estabilización de la red *112*
5.2.4 Publicación de un fichero *116*
5.2.5 Eliminar la clave de un fichero *118*
5.2.6 Búsqueda de una clave *119*
5.2.7 Nodo verifica su predecesor *125*
5.2.8 Paralelismo entre procesos *130*
5.2.9 Comando search *142*
Capítulo 6 - Conclusiones **149**
6.1 Reflexión 150
Bibliografía **151**

Índice de figuras

Figura 2.1 Clasificación de los sistemas *peer-to-peer* ... 12
Figura 3.1 Evolución de los *buckets* de la tabla de encaminamiento de un nodo 00… ... 22
Figura 3.2 Búsqueda del nodo más cercano a la clave 1110… ... 25
Figura 3.3 Tabla de encaminamiento del nodo 0011… ... 27
Figura 3.4 Tabla de encaminamiento del nodo 101… ... 27
Figura 3.5 Representación de los identificadores de los nodos en Chord ... 32
Figura 3.6 Representación de los nodos en CAN en un espacio 2-dimensional ... 49
Figura 3.7 Representación de la ruta para encaminar un mensaje desde el nodo C hasta el nodo E ... 50
Figura 3.8 Estado del espacio de coordenadas bidimensional de una red CAN antes de la incorporación del nodo 4 ... 51
Figura 3.9 Estado del espacio de coordenadas bidimensional de una red CAN después de la incorporación del nodo 4 ... 52
Figura 3.10 Entradas iniciales de la memoria caché del nodo *n* ... 56
Figura 3.11 Representación del estado del nodo 23 en una red Kelips compuesta por 10 grupos de afinidad ... 58
Figura 5.1 Representación del estado inicial del anillo ... 99
Figura 5.2 Comando para ejecutar Maude ... 105
Figura 5.3 Comando que carga la especificación de la Tabla Hash Distribuida Chord ... 105
Figura 5.4 Resultado de insertar el nodo con identificador 20 a la red Chord ... 109
Figura 5.5 Resultado de eliminar el nodo con identificador 15 de la red Chord ... 111
Figura 5.6 Estado de la red antes de ejecutar el proceso de estabilización por el nodo con identificador 15 ... 113
Figura 5.7 Resultado tras ejecutar el proceso de estabilización por el nodo con identificador 15 ... 115
Figura 5.8 Resultado tras publicar en la red el fichero 10111110 por el nodo con identificador 15 ... 117
Figura 5.9 Resultado tras eliminar el fichero 10000010 de la red por el nodo con identificador 15 ... 120

Figura 5.10 Resultado de buscar la clave 00011110 por el nodo con identificador 1 122
Figura 5.11 Resultado de buscar la clave 00000101 por el nodo con identificador 1 124
Figura 5.12 Estado de la red antes de verificar el predecesor del nodo con identificador 30..... 126
Figura 5.13 Resultado tras verificar el predecesor del nodo con identificador 30 128
Figura 5.14 Resultado tras chequear de nuevo el predecesor del nodo con identificador 30..... 129
Figura 5.15 Resultado tras ejecutar tres acciones paralelas (abandono de un nodo e incorporación de dos nodos a la red).. 132
Figura 5.16 Resultado tras ejecutar dos acciones paralelas (proceso de estabilización y abandono de un nodo de la red).. 134
Figura 5.17 Resultado tras ejecutar dos acciones paralelas (incorporación y abandono de un nodo de la red) 136
Figura 5.18 Resultado tras ejecutar dos acciones paralelas (búsqueda y publicación de una clave) 137
Figura 5.19 Resultado tras ejecutar dos acciones paralelas (incorporación de un nodo y publicación de un fichero) 139
Figura 5.20 Resultado tras ejecutar dos búsquedas simultáneamente. 141
Figura 5.21 Solución 1 del comando *Search* con condición < 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : empty , SL : S:SuccessorList, PL : nilPredecessor , KT : empty> 143
Figura 5.22 Solución 2 del comando *Search* con condición < 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : empty , SL : S:SuccessorList, PL : nilPredecessor , KT : empty > 144
Figura 5.23 Solución 3 del comando *Search* con condición < 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : empty , SL : S:SuccessorList, PL : nilPredecessor , KT : empty > 145
Figura 5.24 Resultado del comando *Search* con condición mensaje FILE-FOUND. 147
Figura 5.25 Resultado del comando *Search* con condición mensaje FILE-NOT-FOUND........ 148
Figura B.1 Representación del estado inicial del anillo... 156
Figura B.2 Resultado de buscar la clave 01011010 por el nodo con identificador 223 168
Figura B.3 Resultado tras la incorporación del nodo con identificador 150 a la red................. 171

Índice de tablas

Tabla 3.1 Entradas de las tablas *finger* de los nodos N16 y N65 correspondientes a la figura 3.5 33
Tabla 3.2 Ejemplo de una tabla de encaminamiento Pastry del nodo 1234 con b=4 41
Tabla 3.3 Ejemplo de una tabla de encaminamiento Pastry del nodo 135A..... 44
Tabla 3.4 Resumen de las características generales de las Tablas Hash Distribuidas estudiadas 60
Tabla 5.1 Ficheros que comparte cada uno de los 5 nodos del ejemplo..... 98
Tabla B.1 Ficheros que comparte cada uno de los 15 nodos de la red 157

Agradecimientos

En primer lugar quisiera agradecer a María Isabel la oportunidad que me ha brindado para poder realizar este proyecto. En su calidad de colaboradora del trabajo de fin de Máster me ha guiado con su experiencia en todo momento, y me ha facilitado todo tipo de información y ayuda durante su desarrollo. Por todo esto y mucho más, gracias María Isabel.

Por último agradecer el apoyo de mis familiares y amigos por haber estado a mi lado siempre que lo he necesitado.

Capítulo 1 - Introducción

A partir del año 2000, las redes de ordenadores distribuidas denominadas *peer-to-peer* o P2P, se han convertido en un foco importante de investigación. Estas arquitecturas han sido diseñadas para compartir recursos informáticos mediante la comunicación directa entre los propios equipos que conforman la red sin la intervención directa de un servidor o autoridad central.

Las principales ventajas que ofrecen los sistemas *peer-to-peer* son: tolerancia a fallos, escalabilidad, disponibilidad y rendimiento. Estos sistemas deben garantizar la disponibilidad de los datos y que estos son correctos.

El gran número de usuarios implicados en las redes y la ausencia de una autoridad central implica que el sistema debe ser capaz de operar a pesar de que algunos de los participantes puedan ser maliciosos. Algunos de los ataques más comunes a los sistemas *peer-to-peer* son: el ataque Sybil, el ataque Eclipse y los ataques de encaminamiento y almacenamiento.

Los sistemas *peer-to-peer* pueden ser clasificados según diversas características, en este documento son clasificados por el sistema utilizado para encontrar la información en la red. Así pues, tenemos los sistemas *peer-to-peer estructurados* y los sistemas *peer-to-peer no-estructurados*. Los sistemas *peer-to-peer estructurados* surgieron para intentar cubrir los problemas de escalabilidad de los sistemas *peer-to-peer no estructurados*. Las sistemas *peer-to-peer estructurados* superan las limitaciones de los no estructurados manteniendo una Tabla Hash Distribuida (THD) y permitiendo que cada nodo sea responsable de una parte específica del contenido en la red. Algunas de las Tablas Hash Distribuidas más conocidas son: Kademlia [21], Chord [34], CAN [27], Pastry [28], EpiChord [18], Viceroy [20], y Kelips [15].

En este trabajo se presenta una especificación formal en Maude [8] de la Tabla Hash Distribuida Chord. Maude es un lenguaje de especificación para especificaciones formales basado en lógica de reescritura [23]. Las especificaciones permiten describir el comportamiento

de un sistema en base a la especificación de los tipos de datos que lo componen y las operaciones.

Las especificaciones en Maude presentan principalmente dos ventajas. En primer lugar, al estar basadas en lógica de reescritura es posible razonar sobre los sistemas especificados y demostrar propiedades sobre ellos. Y en segundo lugar, es posible construir prototipos sobre los sistemas especificados basados en la reducción de los términos mediante las ecuaciones y las reglas de reescritura. Estos prototipos consisten en la reducción de términos hasta conseguir sus formas normales.

El uso de métodos formales para describir el comportamiento de la Tabla Hash Distribuida Chord ayuda a entender la descripción informal realizada en [34]. En particular, haciendo uso del lenguaje Maude, nos da la oportunidad de ejecutar y analizar la especificación como si fuese un prototipo.

Se conocen pocos estudios que utilicen técnicas de especificación formal para describir las distintas THD. En particular, existe una descripción formal del protocolo Chord para verificar formalmente el algoritmo de estabilización utilizando π-cálculo [2]. En [25] se presenta una especificación formal en el lenguaje Maude de la THD Kademlia, en [26] se amplia dicha especificación utilizando *sockets* para modelar el comportamiento distribuido del sistema. Por último, en [19] se presenta un modelo de la THD Pastry en el lenguaje de especificación TLA+ y se prueban propiedades de correción y consistencia utilizando su *model checking*.

El resto del documento sigue una estructura sencilla basada en el proceso seguido a lo largo del estudio del trabajo. El capítulo 2 introduce los sistemas *peer-to-peer* con el fin de contextualizar al lector. A continuación, el capítulo 3, presenta las Tablas Hash Distribuidas estudiadas, así como las distintas aplicaciones y ataques que las caracterizan. En el capítulo 4, se aborda el grueso de la parte técnica donde se describe la especificación del protocolo Chord en Maude que satisface los objetivos establecidos. Más adelante, en el capítulo 5 se presenta un análisis de la especificación sobre un conjunto de casos de prueba que contienen algunas de las reducciones que se han llevado a cabo para explicar los procesos y los mensajes que

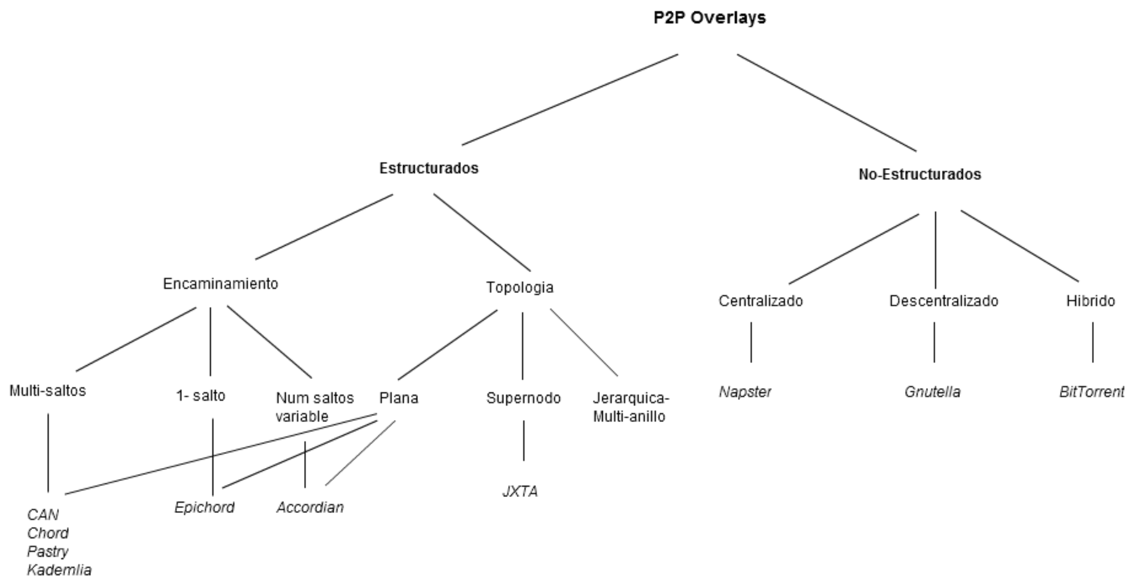
intercambian los nodos sobre un ejemplo. Por último, en el capítulo 6, se proporciona una sección de conclusiones, discusión de las principales aportaciones del proyecto así como una reflexión y bibliografía.

Capítulo 2 - Sistemas peer-to-peer

Los sistemas *peer-to-peer* se definen como redes que tienen la finalidad de compartir recursos; normalmente ficheros. Estos sistemas están compuestos por nodos que se conectan entre sí, usando redes IP formando una red virtual a nivel de aplicación llamada *red superpuesta*. Esta red se utiliza para encaminar mensajes entre los nodos con el objetivo de buscar información. Por ello, debe soportar búsquedas flexibles, eficientes, tolerantes a fallos, y ofrecer garantías de que todo lo que existe puede ser encontrado.

Dependiendo del diseño utilizado para conectar los nodos de la red, podemos clasificar los sistemas *peer-to-peer* en *estructurados* o *no-estructurados* [33]. La figura 2.1 muestra gráficamente la clasificación.

Figura 2.1 Clasificación de los sistemas *peer-to-peer*



2.1 Sistemas *peer-to-peer* no-estructurados

Los sistemas *peer-to-peer no-estructurados* están clasificados a su vez en tres modelos: el modelo *peer-to-peer* centralizado, el modelo *peer-to-peer* puro o totalmente descentralizado y el modelo *peer-to-peer* híbrido o semicentralizado.

2.1.1 Modelo peer-to-peer centralizado

El modelo centralizado es una estructura de red cliente-servidor. El servidor central mantiene una base de datos con información de los ficheros proporcionados por cada nodo. Cada vez que un cliente se conecta o desconecta de la red, la base de datos se actualiza. Los mensajes de búsqueda y control son enviados al servidor centralizado. Este modelo proporciona un rendimiento muy elevado a la hora de localizar recursos.

Del mismo modo, todo el coste y responsabilidad recae sobre una sola máquina y por lo tanto presenta problemas de escalabilidad y seguridad. Por ello se considera que dicho sistema es costoso y difícil de mantener. Además puede ocurrir que el servidor sufra algún ataque o se convierta en un cuello de botella al verse saturado por las peticiones de los múltiples usuarios, en cuyo caso la red dejaría de funcionar.

Napster es uno de ejemplos más representativos en este tipo de arquitecturas. Napster [24] utiliza un servidor central que almacena los índices de todos los archivos disponibles en un grupo de usuarios. Para recuperar un archivo, un usuario consulta el servidor central utilizando el nombre del archivo conocido y obtiene la dirección IP de la máquina que almacena el archivo solicitado. Aunque Napster utiliza un modelo de comunicación *peer-to-peer* para la transferencia del archivo, el proceso de localizar un archivo todavía está muy centralizado, lo cual caracteriza a Napster como un sistema costoso a nivel de escalabilidad y vulnerable a fallos.

2.1.2 Modelo peer-to-peer puro o totalmente descentralizado

En el modelo *peer-to-peer* puro o totalmente descentralizado no existe un servidor central para ayudar a los clientes a buscar la información requerida y el nodo que la posee. Esta responsabilidad recae sobre el conjunto de nodos que conforman la red. Se utiliza un esquema de inundación (*flooding*) para encontrar un determinado recurso enviando mensajes de búsqueda a cada uno de los nodos a los que está conectado. Estos, a su vez, realizan la misma operación, de forma que la búsqueda se expande por todos los nodos desde el origen.

Esta arquitectura es más robusta y económica que la explicada en el apartado anterior, sin embargo, sobrecarga el ancho de banda, requiere más tiempo y la localización de los recursos no está garantizada. Además, el mecanismo de inundación es claramente no escalable.

Un ejemplo de este modelo es Gnutella. La red Gnutella [14] se basa en interconexiones de nodos que se intercambian información del estado de los otros nodos de la red. Las búsquedas se realizan mediante el mecanismo de inundación.

2.1.3 Modelo peer-to-peer híbrido o semicentralizado

El modelo *peer-to-peer* híbrido o semicentralizado selecciona un conjunto de nodos que serán los encargados de realizar el encaminamiento de la red. Esta topología permite reducir o aumentar el número de nodos involucrados en el encaminamiento dependiendo del volumen de tráfico de la red.

La red BitTorrent [3] es un ejemplo de este tipo de modelos la cual fue diseñada para el intercambio de ficheros por el programador estadounidense Bram Cohen y se estrenó en la conferencia CodeCon en el año 2002. BitTorrent permite descargar el contenido de un archivo a varios usuarios sin que unas descargas ralenticen a otras. Para lograr esto el protocolo fuerza a todos los nodos a compartir las partes que tienen de los ficheros con los otros nodos de la red, de modo que cada miembro contribuye a la distribución del contenido.

2.2 Sistemas *peer-to-peer* estructurados

Los sistemas *peer-to-peer estructurados* basan su funcionamiento en la implementación de una Tabla Hash Distribuida (THD) [5].

Cada uno de los nodos que forma parte de una Tabla Hash Distribuida almacena una parte de la información que se comparte en la red en pares <claves, valor>. A cada nodo se le asigna un identificador perteneciente al espacio de claves. Esto permite dividir el espacio de claves en *n* fragmentos, asignando cada fragmento al nodo cuyo identificador este contenido en dicho fragmento. Así, se asegura que cada par <clave, valor> es mantenido por un nodo de la red, pudiéndose encontrar en varios de ellos si se almacena información duplicada por motivos de seguridad.

Una Tabla Hash Distribuida permite localizar de forma eficiente la información compartida en la red. Para ello cada nodo de la red mantiene información sobre la localización de otros nodos, en una estructura de datos llamada *tabla de encaminamiento.* De esta forma, cuando un nodo desea enviar o encaminar un cierto mensaje a un nodo identificado por la clave *k*, envía el mensaje a aquel nodo de los nodos que se encuentran en su tabla de encaminamiento que más cerca se encuentre al destino del mensaje. Dicha propiedad de cercanía entre un nodo y una clave de la Tabla Hash Distribuida varía entre los distintos tipos de tablas, ya que depende de la forma de organizar el espacio de claves y del algoritmo de encaminamiento utilizado. Sin embargo, todas las tablas de encaminamiento ofrecen dos funciones básicas: la inserción-actualización de claves *put* (clave, valor) y el acceso a las mismas *get* (clave). Con estas funciones se llevan a cabo sus operaciones básicas: inicializar la tabla de encaminamiento cuando un nodo se une al sistema, actualizar la tabla de encaminamiento y reorganizar las entradas de los índices para los distintos nodos cuando un nodo se une o abandona el sistema. Este fenómeno es llamado transitoriedad (*churn*).

El tipo de encaminamiento utilizado para buscar una información en la red puede ser de dos tipos:

1. Encaminamiento iterativo. El nodo que inicia una consulta controla todo el proceso de búsqueda. En primer lugar, el nodo inicial manda mensajes a los nodos más cercanos a la clave buscada en su tabla de encaminamiento. Estos le responden con la mejor información que tienen. A continuación, el nodo inicial manda nuevos mensajes a los nodos más cercanos a la clave que busca de entre los que le han enviado, los cuales a su vez le informan de los nodos más cercanos de los que tienen conocimiento. Así, el nodo inicial manda mensajes a los nuevos nodos de los que va teniendo conocimiento hasta que encuentra la información buscada. Este tipo de encaminamiento se utiliza principalmente cuando se realizan búsquedas en paralelo, esto es cuando se envían los mensajes a varios nodos en paralelo para agilizar la búsqueda. El encaminamiento iterativo evita en estos casos contactos redundantes. Además, el encaminamiento iterativo permite actualizar la tabla de encaminamiento del nodo que realiza la búsqueda al mismo tiempo que se realiza esta. Este tipo de encaminamiento es el que se realiza en las THD Kademlia y EpiChord.

2. Encaminamiento recursivo. El nodo inicial contacta con otro nodo, más cercano a la clave buscada, para pedirle información y este segundo nodo es el encargado de contactar a su vez con otro nodo más próximo. Un ejemplo de uso de este tipo de encaminamiento es el utilizado en la THD Chord. Se ha demostrado en [35] que este tipo de encaminamiento mejora el tiempo de búsqueda, es decir, tiene una latencia más baja con respecto al encaminamiento iterativo. Sin embargo el encaminamiento iterativo proporciona otras propiedades de gran utilidad como son un fácil proceso de depuración o un fácil mantenimiento del estado de la red.

La estructura interna de las Tablas Hash Distribuidas es estructurada, comúnmente como anillo o árbol, lo cual satisface la condición de que el número de pasos necesarios para encontrar un determinado valor en la Tabla Hash Distribuida es generalmente del O(log n), donde n es el número total de nodos en el sistema. Respecto a los algoritmos de encaminamiento tienen dos objetivos principales: un fácil mantenimiento de la tabla de encaminamiento y un número pequeño de pasos para encaminar un mensaje. Como solución a satisfacer estas condiciones

tenemos, o bien pequeñas tablas con un orden de pasos logarítmico O(log n), ejemplos existentes de este tipo de sistemas son CAN [27], Chord [34], Kademlia [21] y Pastry [28], o como segunda opción tenemos una tabla grande y uno o dos pasos, siendo Kelips [15], Epichord [18] y Viceroy [20] los ejemplos más significativos.

Las Tablas Hash Distribuidas permiten implementar sistemas completamente descentralizados constituidos por nodos que son iguales entre ellos y altamente escalables. Además, destacan por ser fáciles de programar y por proporcionar un alto rendimiento.

2.2.1 Ataques en los sistemas peer-to-peer estructurados

Los sistemas *peer-to-peer* presentan una alta probabilidad de sufrir un ataque informático con el fin de causar algún tipo de daño al sistema. Los ataques más conocidos son: el ataque Sybil, el ataque Eclipse y los ataques de encaminamiento y de almacenamiento.

2.2.1.1 El ataque Sybil

El ataque Sybil, como describe Douceur [11], consiste en la creación de un gran número de nodos malignos llamados "identidades" y colocarlos de una forma estratégica con el objetivo de tomar el control de una parte de la Tabla Hash Distribuida. Este ataque no destruye una Tabla Hash Distribuida por sí solo. Sin embargo puede ser utilizado para generar un gran número de nodos maliciosos en la red con el fin de facilitar la ejecución de otros ataques. Así pues, dada una red superpuesta, cuantos más nodos maliciosos haya en el sistema más fácil será contaminar las tablas de encaminamiento de los nodos honestos.

Las estrategias de defensa diseñadas para proteger a las Tablas Hash Distribuidas de este ataque tienen como objetivo limitar su actuación, ya que garantizar una total protección es imposible debido a que se requiere una entidad física central para validar la identificación de los nodos de la red.

2.2.1.2 El ataque Eclipse

Para poder realizar este ataque, el atacante debe tener el control de un número de nodos honestos de la red. Para ello divide la red en distintas subredes, de forma que cuando un nodo desea comunicarse con un nodo de otra subred, el atacante encaminará el mensaje a través de un nodo malicioso. Así el atacante "eclipsa" una subred de otra.

Sit y Morris [32] fueron los primeros en estudiar este ataque enfocado en las Tablas Hash Distribuidas. Afirman que los sistemas en los que los nodos vecinos no tienen requisitos especiales de verificación son los más vulnerables a este tipo de ataque. La forma más fácil de explotar esta debilidad es a través de las actualizaciones erróneas en las tablas de encaminamiento.

En el modelo más común del ataque Eclipse los nodos maliciosos se unen y tratan de maximizar el envenenamiento de las tablas de encaminamiento de todos los nodos honestos. Para ello, se usa información maliciosa durante los protocolos de actualización de las tablas de encaminamiento. Sin embargo, este no es el único escenario posible. Por ejemplo, el adversario puede tratar de atacar sólo un pequeño subconjunto de los nodos, una clave específica o filas específicas de las tablas de encaminamiento. El atacante también puede tratar de difundir la intoxicación de forma lenta, atacando los nodos de forma secuencial y comportándose correctamente la mayor parte del tiempo.

Muchas soluciones de defensa han sido propuestas. Sin embargo, la defensa contra este tipo de ataques sigue siendo un problema abierto.

2.2.1.3 Los ataques de encaminamiento y almacenamiento

Los ataques Sybil y Eclipse son ataques que no perturban directamente las Tablas Hash Distribuidas. Estos simplemente crean nodos maliciosos (ataque Sybil) o aíslan a los nodos honestos de la red (ataque Eclipse). Sin embargo, una vez que los nodos maliciosos están

instalados en la red y los nodos honestos están eclipsados, se puede realizar un daño en las peticiones de búsquedas de las Tablas Hash Distribuidas mediante el envío de mensajes a los nodos maliciosos, además de devolver resultados falsos. Estos ataques suelen ser utilizados para facilitar otros ataques, especialmente los ataques de encaminamiento que perturban el proceso de encaminamiento, y los ataques de almacenamiento que devuelven una información errónea a las solicitudes de búsqueda.

Estas son algunas de las amenazas de seguridad más conocidas a las que se enfrentan las Tablas Hash Distribuidas. La gran variedad de estrategias de defensa para su detección o protección muestra la dificultad de diseñar una Tabla Hash Distribuida libre de ataques. Podemos concluir que para mejorar la seguridad de las Tablas Hash Distribuidas, estas requieren una fuerte identificación de los nodos, una autoridad central de confianza y un algoritmo de encaminamiento fiable y eficiente.

En el siguiente capítulo se detallan las tablas hash distribuidas Kademlia, Chord, Pastry y CAN.

Capítulo 3 - Tablas Hash Distribuidas

El objetivo primordial de este capítulo es dar al lector una introducción teórica acerca de algunas de las Tablas Hash Distribuidas más conocidas: Chord, Kademlia, CAN, Pastry, EpiChord, Kelips y Viceroy.

3.1 Kademlia

Kademlia es una Tabla Hash Distribuida usada para la implementación de redes *peer-to-peer* estructuradas. Kademlia especifica la información que debe almacenarse en cada nodo de la red, referente a los otros nodos, así como, la información de los ficheros que se comparten. Además, regula la comunicación entre nodos y el intercambio de información. Al igual que en otras THDs la complejidad de la operación de búsqueda es del orden de O(log n), siendo n el número de nodos que forman parte de la tabla. Fue desarrollada en el año 2002 por Petar Maymounkov y David Mazieres y presentada en la Universidad de Nueva York [21].

3.1.1 Asignación de claves a nodos

Kademlia asigna a cada nodo y clave de fichero un identificador en binario, de igual longitud para nodos y claves, generando el "espacio de claves o nombres". Normalmente, las cadenas de bits son de 160 bits generadas mediante la función hash SHA-1.

Cada nodo tiene una clave única asociada, la cual lo identifica en la red de forma inequívoca. La red guarda información en forma de pares <clave, valor>. La clave es el identificador de un fichero y el valor la localización del nodo que lo comparte. Cada par se guarda en el nodo conectado a la red cuyo identificador es el más cercano entre los conectados a la clave del par. Kademlia utiliza la función XOR para calcular la distancia entre dos identificadores de claves. La ventaja principal de esta función es la simetría, de tal forma que un nodo conoce a todos los identificadores de los nodos que le conocen.

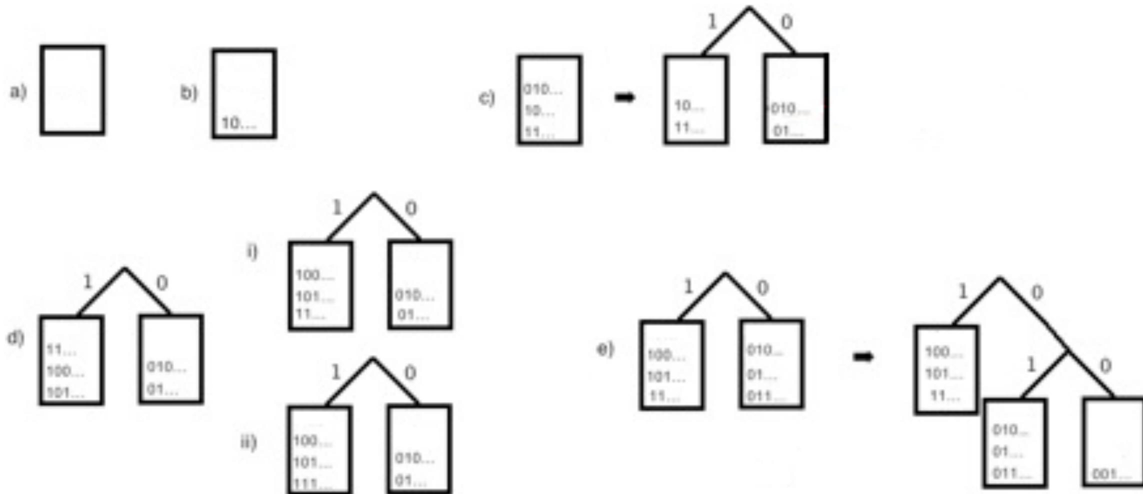
3.1.2 Tabla de encaminamiento

Cada nodo de la Tabla Hash Distribuida almacena una tabla de encaminamiento con información sobre la localización de los nodos red que se encuentren a una distancia entre 2^i y 2^{i+1} de sí mismo, donde i toma el valor desde 1 hasta la longitud de los identificadores. En Kademlia, esta tabla de encaminamiento se puede representar como un árbol binario en cuyas hojas se ubican unas listas de nodos que comparten el mismo prefijo en sus identificadores y en los nodos internos están los prefijos comunes de dichas listas. Estas listas se denominan *k-buckets*.

Cada *bucket* está compuesto por *k* nodos como máximo. El valor de *k* viene determinado por el número de nodos que tienen una alta probabilidad de fallo en una hora. Para cada nodo se almacena su identificador, el puerto UDP y la dirección IP. Puede que algunos de los *buckets* que pertenecen a los rangos bajos en el árbol estén vacíos, esto es debido a que el intervalo de los identificadores es pequeño. Cada *bucket* está ordenado de forma que el primer elemento de la lista es el nodo que hace más tiempo que hemos contactado y al final de la lista se encuentran los nodos con los que hemos contactado últimamente. Esta política da preferencia a los nodos más antiguos del *bucket* siguiendo las conclusiones del estudio realizado en [30], donde se razona que los nodos que llevan más tiempo conectados tienen más probabilidades de seguir conectados que los nuevos nodos que se incorporan al sistema.

El número de *buckets* de la tabla de encaminamiento de un nodo viene dado por el número de veces que se haya dividido que es la longitud del prefijo común de los contactos del último *bucket*. La figura 3.1 muestra la evolución de los *buckets* de la tabla de encaminamiento del nodo 00… para *k-buckets* de longitud 3 (*k*=3).

Figura 3.1 Evolución de los *buckets* de la tabla de encaminamiento de un nodo 00…



donde cada fase consiste en:

a. Inicialmente sólo hay un *bucket* en la tabla de encaminamiento del nodo 00… Este *bucket* está vacío.
b. A medida que se conocen nuevos contactos estos se van introduciendo en el *bucket* hasta que quede completo.
c. El *bucket* está lleno. Para insertar un nuevo contacto con prefijo 01… se contacta con el primer contacto del *bucket*. Se comprueba si dicho contacto está conectado. Si está conectado y el prefijo del *bucket* coincide con el prefijo del identificador del nodo propietario de la tabla el *bucket* se divide y se inserta el nuevo contacto en el *bucket* que le corresponda. En este caso, el prefijo del identificador del nuevo nodo coincide con el prefijo del nodo propietario por lo que el *bucket* se divide en dos *buckets*. Nótese que el nodo propietario de la tabla de encaminamiento siempre corresponde al *bucket* situado más a la derecha de la tabla, que es el *bucket* que se puede dividir. Se siguen añadiendo contactos en los

buckets hasta que estos están llenos, entonces dependiendo de quién es el nodo propietario se toman diferentes caminos.

d. El *bucket* con prefijo 1 está lleno y se quiere insertar un nuevo contacto cuyo prefijo es 111….El *bucket* no puede dividirse ya que su prefijo no coincide con el prefijo del identificador del nodo propietario. Por tanto, se contacta con el primer contacto del *bucket*. Pueden darse dos situaciones:

   i. Si el primer contacto del *bucket* está vivo, este pasa a ocupar la última posición de la lista y el nuevo contacto a insertar se descarta.
   ii. Si el primer contacto de la lista está desconectado, este se descarta y el nuevo contacto se inserta en la última posición del *bucket*.

e. El *bucket* con prefijo 0 está lleno y se quiere insertar un nuevo contacto con prefijo 001.... El *bucket* se vuelve a dividir en dos ya que su prefijo coincide con el prefijo del identificador del nodo propietario de la tabla de encaminamiento.

De esta forma la tabla de encaminamiento del nodo 00.. va creciendo y almacenando nuevos contactos, generando un árbol binario degenerado.

3.1.3 Protocolo Kademlia

El protocolo Kademlia proporciona cuatro *Remote Procedure Calls* (RPC) básicas implementadas con segmentos UDP.

- PING. Comprueba si un nodo está conectado a la red.
- STORE. Ordena a un nodo que almacene un par<clave, valor> en su memoria.
- FIND_NODE. Toma como argumento el identificador de un nodo. Obtiene los α nodos con identificador más cercano al identificador dado como parámetro que se encuentran en la tabla de encaminamiento del nodo que recibe la petición.

- FIND_VALUE. Toma como argumento la clave de la información a buscar. Si el nodo que recibe la RPC tiene información sobre la clave devuelve el valor asociado, si no tiene la información devuelve los α nodos más cercanos a la clave que conozca, como ocurre en la operación FIND_NODE.

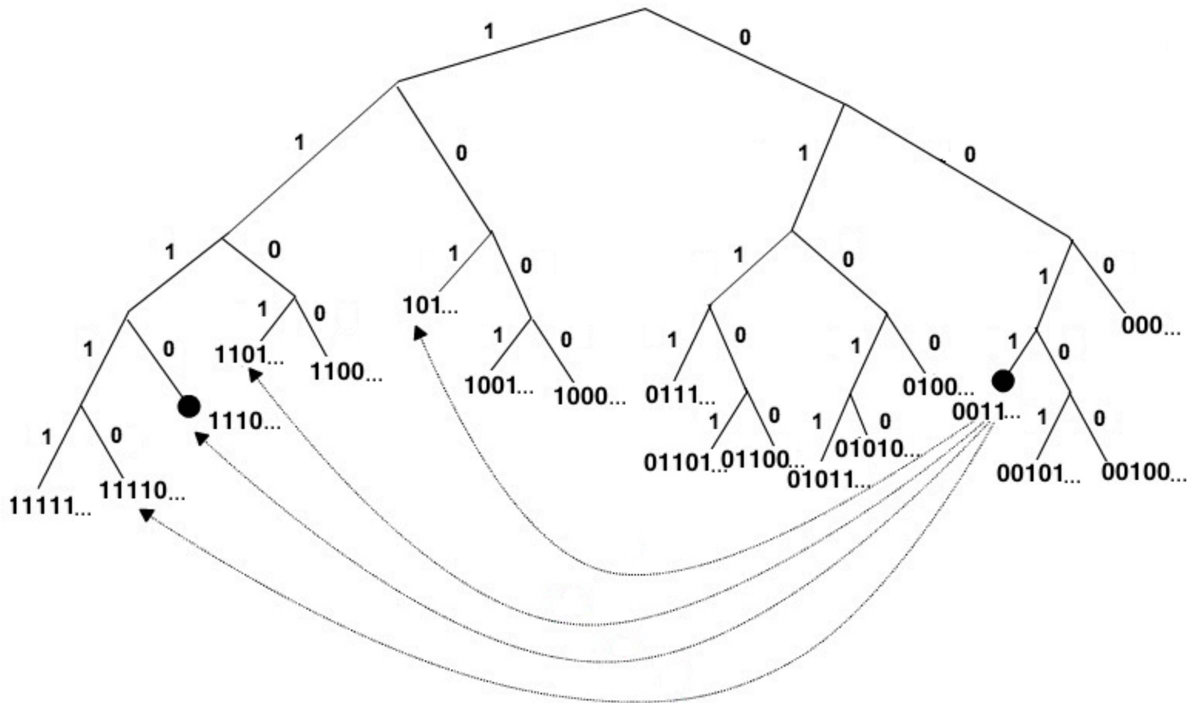
En Kademlia se sigue un enfoque dinámico para la detección de fallos y el mantenimiento del estado de la red, ya que estas acciones dependen del mecanismo de búsqueda y del tráfico de la red. Esto se debe a que al enviar una consulta se adjunta la información necesaria para actualizar los *k-buckets* de los nodos que la reciban. Cuando el tráfico de mensajes es considerable, los *k-buckets* se mantienen actualizados. Sin embargo, puede ocurrir que el tráfico de la red disminuya, en cuyo caso no se garantiza que los *k-buckets* estén actualizados. Para evitar esta situación, los nodos de Kademlia refrescan sus *k-buckets* cada hora. Este proceso consiste en elegir aleatoriamente una entrada del *k-bucket* para comprobar que dicho contacto sigue vivo, realizando una búsqueda a través del identificador del nodo. Los fallos se detectan cuando no se obtiene respuesta de alguno de los nodos contenidos en los *k-buckets*. Cuando esto ocurre se elimina la entrada.

3.1.4 Algoritmo de búsqueda

En Kademlia las búsquedas se realizan siguiendo un mecanismo asíncrono paralelo y para ello se emplean las referencias contenidas en los *k-buckets*. Cuando un nodo n quiere realizar una consulta por un recurso con clave k, obtiene los α pares más próximos con respecto a la medida XOR de su tabla de encaminamiento y les envía la consulta por k. Cuando los α pares reciben la consulta por k, si poseen la referencia a k se la hacen llegar a n, en caso contrario consultan su tabla de encaminamiento y le mandan los x nodos más cercanos que conocen. En este caso la búsqueda se realiza de manera iterativa y en cada iteración, la proximidad entre las claves se reduce al menos en un medio. Además cada vez que se recibe un mensaje se actualiza el nodo en el *bucket* correspondiente. Esto hace que los nodos de la tabla de encaminamiento se mantengan actualizados en función del intercambio de mensajes.

En la figura 3.2, el nodo cuyo identificador tiene el prefijo 0011... busca información asociada a una clave con prefijo 1110....

Figura 3.2 Búsqueda del nodo más cercano a la clave 1110…



Los pasos a seguir son:

1. El nodo con prefijo 0011... busca los α nodos con identificador más cercano a la clave 1110.... de su tabla de encaminamiento. Estos contactos se encuentran en el *bucket* situado más a la izquierda de la figura 3.3 cuyo primer bit es un 1.
2. Suponiendo que α es 1 y que el contacto más cercano en la tabla de encaminamiento del nodo 0011.. es el 101.... El nodo con prefijo 0011… manda la petición de búsqueda al nodo 101….
3. El nodo 101… le contesta con el nodo más cercano a la clave 1110… de su tabla de encaminamiento. La figura 3.4 muestra la tabla de encaminamiento del nodo 101…. Se observa que todos los contactos del *bucket* situado más a la izquierda de la tabla de encaminamiento del nodo 101 comienzan con el bit cero. Además,

el nodo 101... tiene contactos en su tabla de encaminamiento que empiezan por 11…, por lo que se fijan dos bits.

4. El nodo 101...le envía al nodo 0011… el contacto más cercano que conoce, el nodo 1101….
5. El nodo 0011… envía la consulta al nodo 1101....
6. El nodo 1101...busca en su tabla de encaminamiento el nodo más cercano. Su tabla de encaminamiento contiene contactos que empiezan por 111, por lo que quedan fijados tres bits. El nodo 1101... envía al nodo 0011… el contacto más cercano a la clave de búsqueda, el cual es el nodo 11110…
7. El nodo 0011… envía la consulta al nodo 11110…, el cual en su tabla de encaminamiento contiene nodos cuyo prefijo empieza por 1110, quedando fijado cuatro bits que corresponden con el identificador del nodo buscado.
8. El nodo 11110… envía al nodo 0011… la dirección del nodo 1110….
9. El nodo inicial envía un mensaje al nodo 1110… para obtener la información asociada a la clave. Si el nodo 1110 tiene la información asociada a la clave, entonces envía un mensaje al nodo inicial y la búsqueda se dará por finalizada. En caso de que el nodo no contenga dicha información devolverá los α nodos más cercanos a la clave que conozca, como ocurre durante la búsqueda de un nodo.

Figura 3.3 Tabla de encaminamiento del nodo 0011…

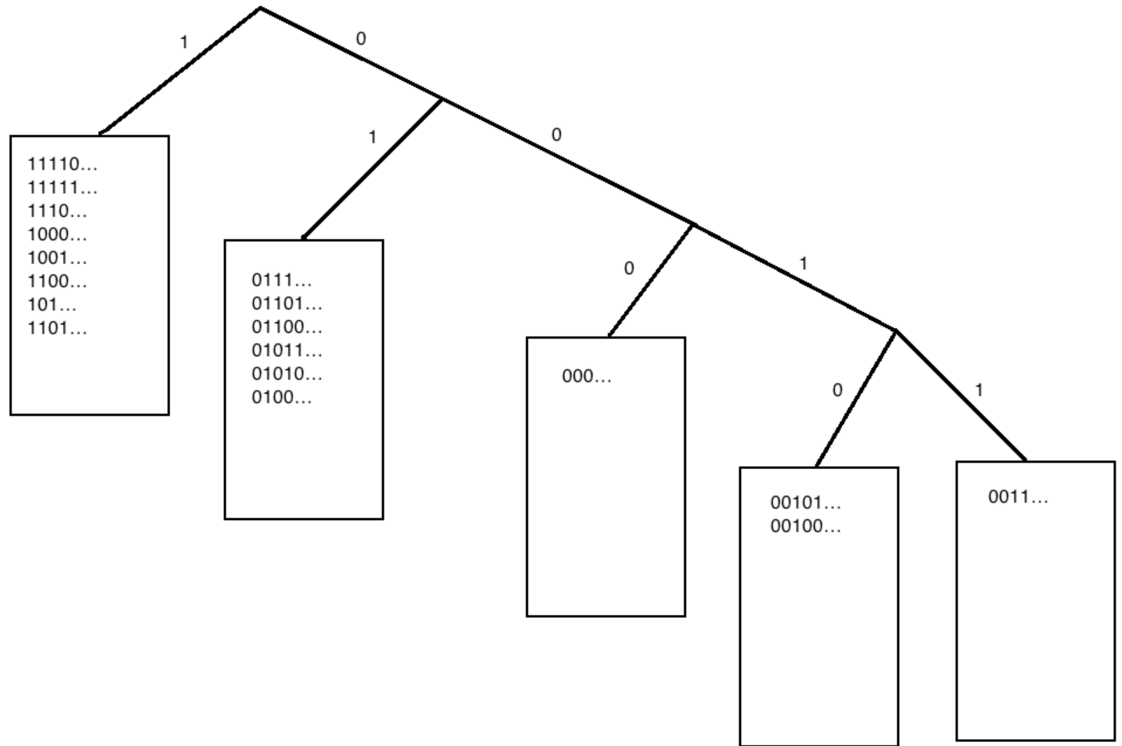
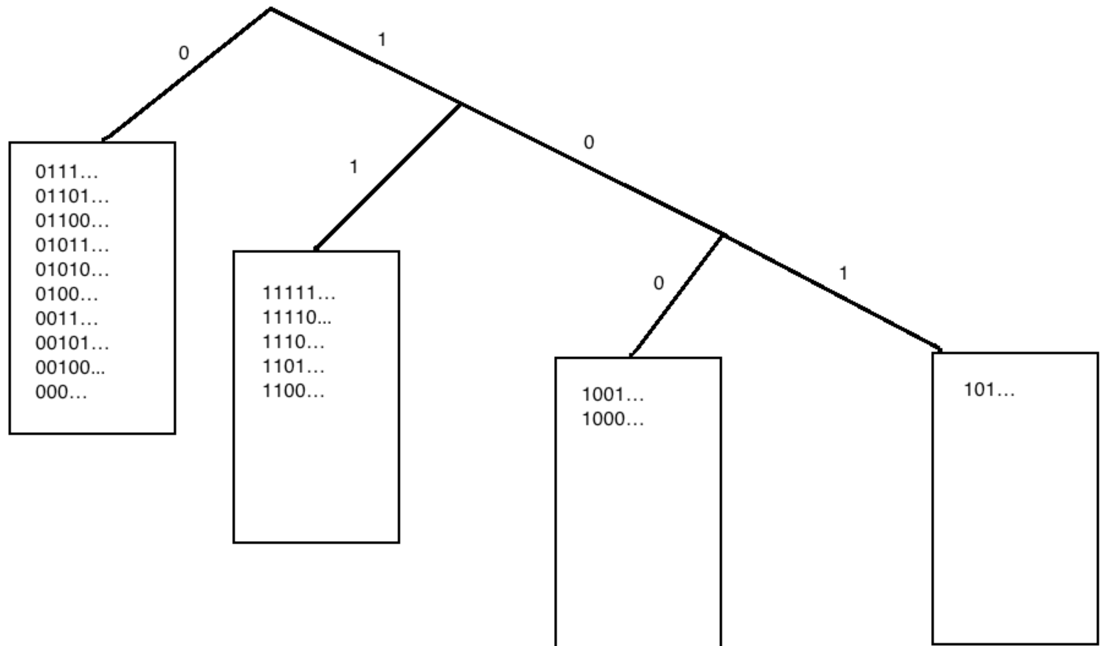


Figura 3.4 Tabla de encaminamiento del nodo 101…



El parámetro α representa el grado de paralelismo en las consultas de la red. Estudios realizados en [35] demuestran que cuando el valor de α es tres se alcanza la máxima eficiencia durante las búsquedas en la red. Para valores mayores de 3, se pierde eficiencia y se incrementa la carga de la red sin mejorar apenas el rendimiento.

3.1.5 Almacenar un par <clave, valor>

Como se explicó en la sección 3.1.1, cada nodo de la red almacena información sobre las claves más cercanas a su identificador. Así pues, el nodo que comparte la información localiza a los α nodos más cercanos a la clave utilizando el algoritmo de búsqueda explicado anteriormente y les mandará mediante el RPC STORE almacenar dicha información.

3.1.6 Incorporar un nodo a la tabla

Cuando un nodo quiere unirse a la tabla debe conocer un nodo de contacto. El nuevo nodo añade el nodo conocido a su tabla de encaminamiento. A continuación, ejecuta el algoritmo de búsqueda preguntando por su propio identificador consiguiendo contactos entre el identificador conocido y el suyo propio. Para completar su tabla de encaminamiento con contactos más lejanos que el contacto inicial conocido, el nodo envía peticiones de búsqueda para otras claves aleatorias con los prefijos que necesite para completar su tabla. Estas peticiones son también enviadas cuando el nodo refresca contactos de los *buckets* que hace mucho tiempo que no modifica.

3.1.7 Dar de baja un nodo de la tabla

Cuando un nodo desea abandonar la red no debe realizar ninguna operación en especial. Es el propio sistema el que lo elimina de las tablas de encaminamiento de los nodos activos cuando éste no responda a los mensajes.

3.1.8 Aplicaciones

Kademlia ha sido implementada en muchas aplicaciones, tales como eMule[13], aMule[1] y BitTorrent[3], las cuales son usadas por millones de usuarios cada día.

eMule es una de las aplicaciones más utilizadas para compartir contenido digital a través de los protocolos de las redes *peer-to-peer*. La razón principal es que permite a sus usuarios acceder a contenido, asegurando la integridad de los ficheros disponibles, evitando la corrupción de archivos y la propagación de malware.

eMule trabaja con dos redes simultáneamente. Por un lado la red eD2k, caracterizada por una arquitectura clásica que utiliza los servidores eD2k. Cada uno de los servidores pone a disposición una cantidad determinada de nodos donde se encuentran las partes de los archivos. Por otro lado, está la denominada red Kad [22], basada en la Tabla Hash Distribuida Kademlia, cuya característica principal reside en ser totalmente descentralizada, es decir que en su funcionamiento todos los nodos que forman parte de la red poseen los mismos valores y las mismas características, logrando de esta manera que la función de eMule pueda seguir su curso independientemente de la caída ocasional de uno de los nodos de la red de servidores.

Cuando utilizamos la red Kad, debemos conocer la dirección IP de un nodo de contacto, las cuales pueden ser obtenidas a través de la red de servidores.

aMule es un programa multiplataforma de intercambio de archivos, similar a eMule que funciona tanto con la red eDonkey como con la red Kad.

BitTorrent, al igual que las anteriores, es un sistema *peer-to-peer* que permite transmitir y descargar archivos pero con la peculiaridad de que dichos archivos suelen tener un tamaño grande. El protocolo BitTorrent clásico fue diseñado con un servidor central (*central tracker*), el cual manejaba el estado de la red, con el fin de ayudar a los usuarios interesados en el mismo

archivo a que encontrarán entre sí la manera de facilitar la descarga. Con el tiempo, observaron que estos servidores de seguimiento no proporcionaban robustez al sistema, ya que cada vez que se caía un servidor (*tracker)*, se caía toda la red. Sin embargo, gracias a la Tabla Hash Distribuida Kademlia, el protocolo BitTorrent soporta un servidor distribuido (*distributed tracker)* proporcionando los mismos servicios que el servidor central pero solventando dicho problema [9].

3.1.9 Ataques

Hoy en día, Kademlia es la Tabla Hash Distribuida más utilizada en los sistemas *peer-to-peer*. Esto hace que una red basada en Kademlia despierte el interés del atacante. Por ello, es importante que Kademlia proporcione un buen sistema de seguridad para sus usuarios.

El ataque de Sybil es uno de los ataques que más afecta a la red Kademlia. Por ejemplo, para eliminar un valor de una clave de la red, los atacantes podrían agruparse en torno a la clave, aceptando cualquier intento de almacenar el par <clave, valor>, pero nunca devolver el valor cuando se consulta la clave.

Para entender completamente los problemas de seguridad que afectan a la Tabla Hash Distribuida Kademlia, se encuentra un análisis basado en la simulación de ataques en [36]. Haciendo uso de simulaciones se puede realizar una buena evaluación de la efectividad y consecuencias del ataque, con ello mejoraremos el diseño de las tácticas defensivas. El impacto del ataque depende principalmente de tres factores: el número de nodos de ataque, la posibilidad de que el atacante elija un ID de forma arbitraria y la posibilidad de iniciar el ataque antes de que el recurso se difunda. Como resultado de las simulaciones, las características de la seguridad de una red Kademlia generan varias cuestiones. La introducción de un servicio de certificación en una Tabla Hash Distribuida puede ser utilizada para negar o limitar todos los ataques realizados contra un sistema *peer-to-peer* estructurado como es la red Kademia o la red Kad.

3.1.10 Conclusión

A modo resumen en el diseño de Kademlia:

- Topología de la red: árbol binario.
- Medida de proximidad: función XOR, la cercanía entre dos identificadores de claves viene definida por el resultado de la función lógica.
- Detección de inconsistencias:
    - Consistencia: en el estado se valora los nodos que llevan mayor tiempo de sesión.
    - Estabilización: los *k-buckets* se refrescan cada cierto intervalo de tiempo. Los mensajes contienen la información necesaria para actualizar el estado de los nodos.

3.2 Chord

Chord es una Tabla Hash Distribuida utilizada para la implementación de redes *peer-to-peer* estructuradas. Chord especifica la información que cada nodo de la red debe almacenar, referente a otros nodos, así como, la información referente a los ficheros que es compartida entre nodos. Además regula la comunicación y el intercambio de mensajes entre estos. Como en otras THDs la complejidad de la operación de búsqueda es del orden de O(log n), siendo *n* el número de nodos que forman parte de la tabla. Chord fue desarrollado en el *Laboratory for Computer Science* del *Instituto Tecnológico de Massachusset* por Ion Stoica, Robert Morris y David Liben-Nowell [34].

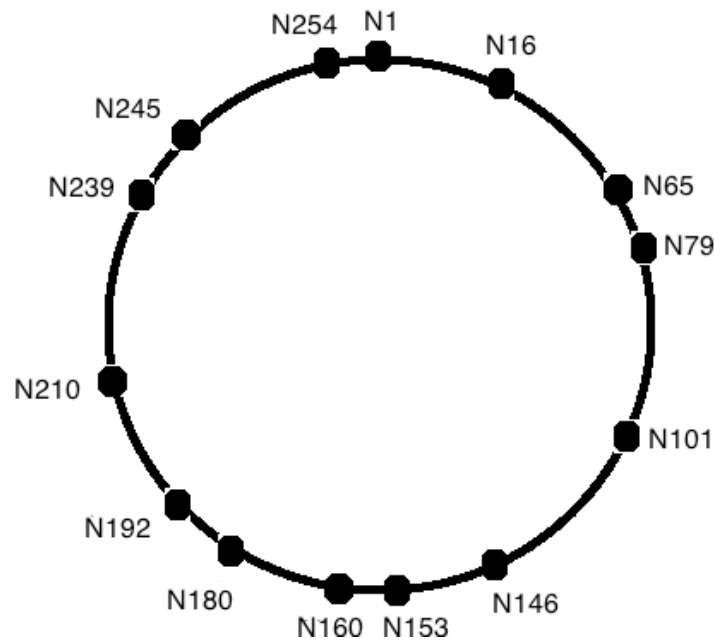
3.2.1 Asignación de claves a nodos

Los nodos y las claves de los ficheros compartidos tienen asociado un identificador binario de *m* bits. El valor de *m* se fija dependiendo del número de nodos de la red. La asignación se realiza mediante la función hash SHA-1.

Los identificadores de los nodos se representan en un anillo de identificadores módulo 2^m. El rango de los valores de los identificadores de los nodos y claves oscila desde 0 hasta 2^m - *1*. En los ejemplos siguientes, identificamos los nodos con el valor numérico de la cadena de bits de su identificador.

Cada nodo tiene un nodo sucesor y un nodo predecesor. El sucesor de un nodo es el siguiente nodo en el anillo en sentido de las agujas del reloj, mientras que el predecesor de un nodo es el siguiente nodo en el anillo en el sentido contrario. En la figura 3.5 se representa un anillo Chord con 256 identificadores (*m*=8) y 14 nodos. El sucesor del nodo N153 es el nodo N160 y el predecesor del nodo N160 es el nodo N153 .

Figura 3.5 Representación de los identificadores de los nodos en Chord



En Chord, cada nodo es responsable de las claves situadas entre él mismo y su predecesor. Por ejemplo, en la figura 3.5, el nodo N16 es responsable de las claves cuyos identificadores sean desde 2 hasta 16 inclusive. Esto significa que el nodo almacena la información necesaria para localizar la información asociada a las claves de las que es responsable. De forma que, una clave *k* es asignada al primer nodo activo cuyo identificador es

igual o es el siguiente a *k* en el espacio de identificadores. Este se conoce como nodo sucesor de la clave *k*, y se denota como *successor(k)*. Por ejemplo, volviendo a la figura 3.5, para insertar la clave 17 hay que localizar en el anillo el primer nodo activo en sentido de las agujas del reloj partiendo desde 17, successor(17), que es el nodo N65. El nodo N65 será el responsable de almacenar la información asociada a la clave con identificador 17.

3.2.2 Tabla de encaminamiento

Un nodo Chord sólo necesita almacenar información sobre unos pocos nodos de la red para localizar la clave que busca. Esta información se almacena en una tabla de encaminamiento, llamada tabla *finger*. Esta tabla tiene *m* entradas, donde *m* es la longitud de bits de los identificadores del anillo. Cada entrada 2^i, donde $0 <= i <= m-1$, de la tabla *finger* de un nodo con identificador *n* está formada por el identificador del nodo sucesor de $(n + 2^i)$ y su dirección IP. Nótese que la primera entrada en la tabla es el sucesor del nodo en el anillo.

La tabla 3.1 muestra las tablas *finger* de los nodos N16 y N65 correspondientes al anillo de la figura 3.5.

Tabla 3.1 Entradas de las tablas *finger* de los nodos N16 y N65 correspondientes a la figura 3.5

Nodo 16		
Índice	**Id Nodo**	**Dirección IP**
2^0 = 1	N65	127.0.0.65
2^1 = 2	N65	127.0.0.65
2^2 = 4	N65	127.0.0.65
2^3 = 8	N65	127.0.0.65
2^4 = 16	N65	127.0.0.65
2^5 = 32	N65	127.0.0.65
2^6 = 64	N101	127.0.0.101
2^7 = 128	N146	127.0.0.146

Nodo 65		
Índice	**Id Nodo**	**Dirección IP**
2^0 = 1	N79	127.0.0.79
2^1 = 2	N79	127.0.0.79
2^2 = 4	N79	127.0.0.79
2^3 = 8	N79	127.0.0.79
2^4 = 16	N101	127.0.0.101
2^5 = 32	N101	127.0.0.101
2^6 = 64	N146	127.0.0.146
2^7 = 128	N210	127.0.0.210

Usaremos un ejemplo para explicar cómo un nodo busca en su tabla *finger* el nodo responsable de una determinada clave. Tomando como referencia la figura 3.5, imaginemos que queremos buscar el nodo que almacena la información de la clave 73 en la tabla *finger* del nodo N16. Para ello, se resta el valor de la clave menos el identificador del nodo de la tabla *finger*, en este caso, 73 menos 16 que da como resultado 57. A continuación, se busca la potencia de dos que más se acerque, sin sobrepasar a 57, en este caso es 5, ya que 2^5 es 32. Por lo tanto, en la entrada 2^5 se almacena el nodo más cercano al nodo responsable de almacenar la información que conoce el nodo con identificador 16, el cual es el nodo N65.

3.2.3 Lista de sucesores y predecesores

Para aumentar la robustez de la Tabla Hash Distribuida, cada nodo mantiene una lista de sucesores y predecesores de tamaño *r*, con los primeros *r* sucesores y *r* predecesores respectivamente. Si el sucesor inmediato de un nodo no responde, el nodo es sustituido por la segunda entrada de su lista de sucesores. Del mismo modo, si el predecesor inmediato de la lista de predecesores de un nodo no responde, es sustituido por la segunda entrada de la lista de predecesores. Los *r* sucesores y los *r* predecesores tendrían que fallar simultáneamente para poder interrumpir el anillo Chord. Una situación así es bastante improbable para valores elevados de *r*. Asumiendo que cada nodo falla independientemente con probabilidad *p*, la probabilidad de que los *r* sucesores y los *r* predecesores fallen simultáneamente es de $p*r*r$. Aumentando *r* se hace el sistema más robusto aunque también resultará más caro de mantener.

3.2.4 Protocolo Chord

Chord soporta cuatro operaciones básicas:

- Búsqueda de una clave.
- Estabilización de la red.
- Incorporación de un nodo a la red.
- Abandono de un nodo la red.

3.2.5 Algoritmo de búsqueda

El algoritmo de búsqueda en Chord consiste en:

Un nodo *n* busca la información de una clave *k*. Inicialmente comprueba si la clave que busca está entre su identificador y su predecesor.

1. La clave está entre su identificador y su predecesor. Por lo tanto, el nodo *n* es el responsable de la clave. Consulta su tabla de claves y se finaliza la búsqueda.
2. La clave no está entre su identificador y su predecesor. El nodo *n* pasa a buscar el nodo responsable de la clave en su tabla *finger* siguiendo los pasos explicados en la sección 3.2.2. Dos situaciones posibles:
    a. Encuentra el nodo responsable de *k* en su tabla *finger*. El nodo *n* envía la petición de búsqueda directamente a ese nodo y se finaliza la búsqueda.
    b. No encuentra el nodo responsable de la clave *k* en su tabla *finger*. El nodo *n* busca en su tabla *finger* un nodo *j* cuyo identificador sea más cercano a la clave *k* que él mismo, ya que ese nodo sabe más que el propio nodo *n* sobre la región del anillo en la que se encuentra *k*. El nodo *n* manda la petición de búsqueda de la clave *k* a ese nodo *j*. Repitiendo este proceso, los nodos van descubriendo otros nodos cuyos identificadores están cada vez más cerca de la clave *k*.

Veamos un ejemplo práctico del algoritmo de búsqueda usando el anillo Chord de la figura 3.5 y las tablas *fingers* de los nodos N16 y N65 de la tabla 3.1. El nodo N16 envía una petición de búsqueda de la clave 73. En primer lugar, el nodo N16 comprueba si la clave está entre su identificador y su predecesor, N1. Como la clave 73 no está comprendida en ese intervalo de claves, el nodo N16 busca en su tabla *finger* el nodo responsable de la clave 73. En la tabla *finger* no se encuentra el nodo responsable de la clave. Por tanto, se busca un nodo cuyo identificador sea más cercano a la clave 73. El resultado es el nodo N65. El nodo 16 reenvía la petición de búsqueda de la clave 73 al nodo N65. El nodo N65 comprueba si la clave 73 está comprendida entre su identificador y su predecesor. La clave 73 no está dentro de ese intervalo

de claves. Por tanto, se busca en su tabla *finger* el nodo responsable de la clave 73. El resultado de la búsqueda es el nodo N79. Así pues, el nodo N65 reenvía la búsqueda al nodo N79. El nodo N79 comprueba si la clave está comprendida entre su identificador y su predecesor. Como es así, el nodo N79 envía un mensaje al nodo invocador, N16, indicándole que es el nodo responsable de la clave 73 y con ello se da por finalizada la búsqueda.

Para que las búsquedas se puedan realizar de forma rápida, es conveniente que las tablas *finger* se mantengan actualizadas. Además, Chord es una red dinámica, lo que significa que los nodos pueden unirse y abandonar la red en cualquier momento. Sin embargo, los nodos deben poder localizar cualquier clave en la red. Para conseguirlo, Chord necesita preservar dos condiciones:

- El sucesor y el predecesor de un nodo tiene que mantenerse correctamente.
- Para cada clave *k*, el sucesor de *k* es su responsable.

Cada nodo ejecuta periódicamente el protocolo de estabilización con el fin de mantener actualizadas su lista de sucesores, su lista de predecesores y su tabla *finger*.

3.2.6 Protocolo de estabilización

El protocolo de estabilización es un algoritmo que se ejecuta en cada uno de los nodos para mantener actualizada la información de las listas de sucesores y predecesores y de la tabla *finger*. Es necesario ejecutar este algoritmo frecuentemente debido al carácter dinámico de estas redes, en las que los nodos se incorporan y abandonan la red muy rápidamente. Su funcionamiento consiste en:

Un nodo *n* inicia el proceso de estabilización cada cierto tiempo *x*. En primer lugar, el nodo *n* solicita a su sucesor *s* que le envíe su predecesor inmediato *p*.

a. El nodo *p* es el sucesor del nodo *n*, lo que significa que el nodo *n* tiene un sucesor inmediato válido.
b. El nodo *p* no es el sucesor inmediato del nodo *n*, lo que significa que se ha incorporado recientemente un nuevo nodo al sistema, el nodo *p*. El nodo *n* actualiza su lista de sucesores, predecesores y su tabla *finger* incorporando al nuevo nodo *p*. Ahora el nodo *n* tendrá como sucesor inmediato *p*. El nodo *n* notifica al nodo *p* la existencia de sí mismo, *n*, con el fin de que el nodo *p* actualice su predecesor inmediato a *n*.

Por último, el nodo *n* envía un mensaje de estabilización a la red para su sucesor con el fin de propagar el proceso de estabilización a través de la red. Este proceso se repite hasta que se alcanza el nodo que inicio el proceso de estabilización, *n*. Tras esto, la información contenida en la lista de sucesores, predecesores y las tablas *finger* de cada nodo participante de la red será una información válida y coherente con el estado actual del anillo.

Rana Bakhshi y Dilian Gurov verifican formalmente en [2] la robustez del algoritmo de estabilización utilizando álgebras de procesos.

3.2.7 Incorporar un nodo a la red

Un nodo *n* se incorpora a la red Chord a través de un nodo contacto en el anillo Chord. En primer lugar, el nodo *n* manda un mensaje a su nodo contacto con el identificador con el que desea incorporarse al anillo. El nodo contacto debe añadir al nodo *n* en la zona adecuada del anillo entre el predecesor y el sucesor de su identificador. Una vez situado el nodo *n* en el anillo, se le asignan las claves que se encuentran entre su identificador y su predecesor, claves que antes correspondían al que ahora es su sucesor. Por ejemplo, tomando como referencia la figura 3.5, si el nodo N5 se incorpora a la red, este se sitúa entre el nodo N1 y el N16 y se le asignan las claves 2, 3, 4 y 5.

3.2.8 Dar de baja un nodo de la red

Cuando un nodo *n* decide abandonar la red, informa a su nodo predecesor *p* y a su nodo sucesor *s* su partida, entregando las claves de las que es responsable a su nodo sucesor. El nodo *p* y el nodo *s* actualizan su lista de predecesores y sucesores eliminando el nodo *n*. El nodo s incorpora las claves entregadas por el nodo *n* a su tabla de claves. El resto de nodos que forman parte de la red eliminan de su lista de atributos al nodo que abandona la red a través del proceso de estabilización.

Cuando falla un nodo, surge un problema pues su predecesor ya no tiene un sucesor válido. Para solucionarlo, cada nodo Chord hace uso de su lista de sucesores. Si un nodo se da cuenta que su sucesor ha fallado, simplemente toma al siguiente nodo en la lista como su sucesor. No obstante, pasado un tiempo, a través del proceso de estabilización se actualiza su lista de atributos eliminando al nodo fallido.

3.2.9 Aplicaciones

Chord es conocida como una Tabla Hash Distribuida, sin embargo, Chord no especifica ningún mecanismo para almacenar los datos. Por ello, en la capa superior de encaminamiento de Chord se ha implementado el sistema de almacenamiento Dhash.

Dhash proporciona una interfaz para la recuperación y obtención de los datos en los nodos participantes de la red. Dhash utiliza técnicas como la replicación. Los datos almacenados en el sistema son inmutables e identificados por su contenido. DHash ha sido implementado en sistemas de copias de seguridad, en sistemas con múltiples archivos (CFS) y en el servidor *Usenet News.*

CFS [10] es un sistema de archivos construido encima de la red superpuesta de Chord. Ofrece los servicios básicos de una Tabla Hash Distribuida, y mantiene réplicas y caches. CFS está orientado al almacenamiento de bloques y encima de él se construye un sistema

convencional de ficheros similar al sistema UNIX. Cada bloque es guardado simultáneamente en varios nodos Chord. No ofrece otros servicios como podrían ser un motor de búsqueda, aunque sí que utiliza un sistema de autentificación de clave pública.

UsenetDHT [31] permite a los servidores Usenet compartir almacenamiento. Estos sistemas aprovechan las ventajas proporcionadas por Chord y Dhash para distribuir un gran nivel de carga de datos a un buen rendimiento. DHash abstrae el proceso de replicación y mantenimiento de datos usando nodos fallidos y reunión de nodos al sistema.

3.2.10 Ataques

Chord tiene una alta vulnerabilidad a los ataques de encaminamiento y almacenamiento. En este tipo de ataques los atacantes, generalmente, utilizan falsa información, tales como: índices de falsos archivos, falsas direcciones IP o falsas tablas de encaminamiento, cuya finalidad es dar una visión incorrecta del estado del anillo Chord. Asumiendo que la información almacenada en una red Chord es usada para autentificar la localización de los nodos participantes. Una posible estrategia para comprobar si la red está siendo víctima de un ataque podría ser: por cada nodo n que forma parte de la red se solicita a los otros nodos participantes de la red que busquen al nodo n . Si el resultado de la búsqueda es fallido significa que la red está siendo víctima de un ataque ya que los nodos no tienen una visión consistente del estado del anillo Chord.

3.2.11 Conclusión

A modo resumen del diseño de Chord:

- Topología de la red: anillo.
- Medida de proximidad: sucesores, nodos cuyo identificador es inmediatamente superior y predecesores, nodos cuyo identificador es inmediatamente inferior.
- Detección de inconsistencias:

- Protocolo de estabilización: actualiza la lista de sucesores, la lista de predecesores y las tablas *finger* de los nodos que forman el anillo.

3.3 Pastry

Pastry es una Tabla Hash Distribuida utilizada para la implementación de redes *peer-to-peer* estructuradas similar a Chord. Pastry especifica la información que cada nodo de la red debe almacenar, referente a otros nodos, así como, la información de los ficheros que es compartida entre nodos. Además regula la comunicación y el intercambio de mensajes entre estos. Como en otras THDs la complejidad de la operación de búsqueda es del orden de O(log n), siendo n el número de nodos que forman parte de la red. Pastry fue desarrollado en *Microsoft Research* por Antony Rowstron y en la Universidad Rice de Houston por Peter Druschel [28].

3.3.1 Asignación de claves a nodos

A cada nodo y cada clave de fichero se le asigna un identificador único e intransferible de 128 bits generado mediante un generador de números aleatorios de alta precisión o a través de la función hash SHA–1 o mediante Global Unique IDentifier (GUID) en [28], estos identificadores son números en base 16 (hexadecimal). Los identificadores de los nodos se representan en un anillo de identificadores módulo 2^{128}. El rango de los valores de los identificadores de los nodos oscila desde 0 hasta 2^{128}-1.

El identificador de un nodo está dividido en varios niveles, donde cada nivel representa un dominio, el cual está representado por b bits consecutivos del identificador. Los bits desde las posiciones $b*l$ a $b*(l+1)-1$ especifican el dominio en el nivel l. Así pues, los b bits más significativos del identificador representan el dominio del nodo al nivel 0, y así sucesivamente. Existen 2^b dominios en cada nivel, numerados desde 0 hasta 2^b - 1.

Cada nodo contiene, además de su identificador, una lista de nodos vecinos, una lista de nodos hoja y una tabla de encaminamiento. Los nodos hoja son los nodos más próximos al identificador del nodo de la lista.

Cada clave de fichero es almacenada en los *p* nodos cuyos identificadores son los más cercanos al identificador de la clave. Pastry garantiza que la búsqueda de una clave es encaminada por el nodo cuyo identificador es el más cercano al identificador de la clave.

3.3.2 Tabla de encaminamiento

La tabla de encaminamiento de un nodo tiene $\log_{2^b} n$ niveles. Cada nivel *l* contiene 2^b -1 entradas. Cada entrada contiene la dirección IP de los nodos que comparten el mismo prefijo que el identificador del nodo hasta el nivel *l*-1. El tamaño de la tabla de encaminamiento es, aproximadamente, $[\log_{2^b} n] * (2^b - 1)$, donde *n* es el número de nodos que forman la red.

La tabla 3.2 muestra las primeras filas de la tabla de enrutamiento de un nodo cuyos primeros dígitos de su identificador son 1234.

Tabla 3.2 Ejemplo de una tabla de encaminamiento Pastry del nodo 1234 con b=4

Fila	Tabla de encaminamiento									
0	0	1	2	3	4	5	6	7	8	…	F
1	10	11	12	13	14	15	16	17	18	…	1F
2	120	121	122	123	124	125	126	127	128	…	12F
3	1230	1231	1232	1233	1234	1235	1236	1237	1238	…	123F

En las celdas de cada fila se hace variar el último dígito, conservando los anteriores que corresponden al nodo al que pertenece la tabla. Así pues, en la fila cero, se tiene una variación de 0 a F, en la fila uno, todas las celdas empiezan con 1 por lo que variamos el segundo digito, en la fila 2 todas las celdas empiezan por 12 y variamos el tercer digito y en la fila 3 todas las celdas

empiezan por 123 por lo que variamos el cuarto dígito, y así sucesivamente hasta completar la tabla.

3.3.3 Lista de nodos vecinos

La lista de nodos vecinos contiene los *m* nodos más próximos según las métricas usadas en la red. Las métricas de encaminamiento son proporcionadas por programas externos según la dirección IP del nodo, lo que permite cambiar fácilmente de métrica para buscar las rutas entre nodos con menor latencia o mayor ancho de banda.

Esta lista no se utiliza durante el algoritmo de búsqueda. Su utilidad es mantener la tabla de encaminamiento. El tamaño de la lista es de 2^b.

3.3.4 Lista de nodos hoja

La lista de nodos hoja contiene el conjunto de nodos más próximos al identificador del nodo en las dos direcciones del círculo, normalmente los ocho mayores y los ocho menores. Esta lista es utilizada durante el algoritmo de búsqueda con el fin de acortar las búsquedas. El tamaño de la lista es de $2*2^b$.

3.3.5 Algoritmo de búsqueda

En Pastry se envían mensajes de un nodo a otro mediante el algoritmo de búsqueda para encontrar una clave *k*.

Un nodo *n* recibe un mensaje de búsqueda de un nodo *d*. El nodo *n* comprueba si el identificador del nodo destino *d* está dentro de la lista de nodos hoja.

1. Si el nodo *d* está dentro de su lista de nodos hoja, el mensaje se reenviará directamente al nodo *d*.

2. Si el nodo *d* no está dentro de su lista de nodos hoja, entonces se utiliza la tabla de encaminamiento. La búsqueda del nodo al que se le debe reenviar la búsqueda se realiza comparando, hasta donde sea posible, dígito por dígito entre los identificadores del nodo *n* y el de *d* en la tabla de encaminamiento, acercándose así a él cada vez más, si no se puede continuar recorriendo la tabla de encaminamiento del nodo *n* y no se ha encontrado el identificador exacto del destino, se envía la búsqueda al nodo que se encuentre en la celda con el prefijo común más largo.

   Por ejemplo, tomando como referencia la tabla 3.2, el nodo 1234 recibe un mensaje con nodo destino 134D. El nodo 1234 busca en su tabla de encaminamiento la celda con el identificador más cercano al 134D. Como podemos ver en la tabla, no es posible que la búsqueda pase de la fila 1, ya que los nodos comparten solamente un digito de sus identificador. Así pues, la celda 13 contiene la información del nodo más cercano al nodo 134D. Supongamos que la celda 13 contiene la dirección IP del nodo 135A. La tabla 3.3 muestra la tabla de encaminamiento del nodo 135A. Buscando la celda con el identificador más cercano a 134D llegamos a la celda 134 y en ella encontramos la IP del nodo 134D dando por finalizada la búsqueda. Sin embargo podría darse el caso de que la dirección IP encontrada en la celda fuese otra, como por ejemplo 134E, en cuyo caso continuaríamos el proceso hasta encontrar la dirección IP del nodo 134D.

Tabla 3.3 Ejemplo de una tabla de encaminamiento Pastry del nodo 135A

Fila	Tabla de encaminamiento									
0	0	1	2	3	4	5	6	7	8	…	F
1	10	11	12	13	14	15	16	17	18	…	1F
2	130	131	132	133	134	135	136	137	138	…	13F
3	1350	1351	1352	1353	1354	1355	1356	1357	1358	…	135F

La complejidad del algoritmo de búsqueda es del orden de O(log *n*) siendo *n* el número de nodos que forman parte de la red Pastry.

3.3.6 Incorporar un nodo a la tabla

Cuando un nodo *n* quiere formar parte de la red:

1. El nodo *n* envía un mensaje de unión a uno de sus nodos vecinos de su lista de acuerdo a la medida de proximidad de la red, indicándole el identificador con el que desea incorporarse al sistema.
2. El nodo vecino reenvía el mensaje a través de los nodos n_1, n_2,… hasta el nodo n_i, donde n_i es el nodo cuyo identificador es el más cercano al identificador del nodo *n.* A continuación, el nodo *n* recibirá información para completar su tabla de encaminamiento y sus listas como sigue:

    i. El nodo *n* recibe del nodo vecino su lista de nodos vecinos y la fila 0 de su tabla de encaminamiento.
    ii. El nodo *n* recibe la fila 1 de la tabla de encaminamiento del nodo n_1, ya que el nodo *n* y el nodo n_1 deben tener un dígito en común en su prefijo. El

nodo *n* recibe la fila 2 de la tabla de encaminamiento del nodo n_2 y así sucesivamente hasta completar la fila *i* con la información del nodo n_i.

iii. El nodo n_i envía al nodo *n* su lista de nodos hoja, ya que n_i y n son numéricamente próximos.

iv. Por último, el nodo *n* informa a los nodos de su tabla de encaminamiento y de sus listas de su incorporación en la red para que tengan constancia de su existencia. Con ello, el nodo *n* quedaría integrado en la red Pastry.

3.3.7 Dar de baja un nodo de la tabla

Cuando un nodo desea abandonar la red, este notificará voluntariamente a los demás nodos para que estos actualicen la información de sus tablas y listas.

Cada nodo envía periódicamente un mensaje a sus nodos vecinos para notificarles que sigue activo. Si un nodo no responde a dichos mensajes durante un periodo de tiempo *t*, se asume que dicho nodo ha fallado y se activa un protocolo para actualizar las listas. El nodo que descubre que ha fallado un nodo, localiza al nodo con identificador más cercano al identificador del nodo fallido y lo reemplaza. Ese nuevo nodo enviará un mensaje a todos sus nodos más cercanos para que actualicen su estado.

3.3.8 Protocolo Pastry

Algunas de las operaciones que presenta la interfaz de Pastry para interactuar con la aplicación son:

- **nodoId = pastryInit(Credentials).** Incorpora un nodo a la red Pastry. El parámetro *Credentials* es proporcionado por la aplicación y contiene toda la información necesaria para incorporar al nodo a la red de forma segura. Como resultado de la operación el nodo quedará integrado en la red.

- **route(msg,destId).** Encamina un mensaje hacia el nodo cuyo identificador es numéricamente más cercano al identificador del nodo destino.
- **deliver(msg,destId).** Recibe un mensaje cuyo identificador del nodo local es numéricamente el más cercano al identificador del nodo destino.
- **forward(msg,destId,nextId).** Reenvía un mensaje a un nodo cuyo identificador es el mismo que el identificador del nodo destino. Si el identificador del nodo destino es NULL significa que el mensaje va dirigido al nodo local.
- **newLeafs(leafSet).** Este método se utiliza cuando hay un cambio en la lista de nodos hoja. Gracias a este método la aplicación tiene la oportunidad de actualizar el vector de nodos hoja frente a posibles incoherencias de la red.

3.3.9 Aplicaciones

El sistema Pastry ha sido utilizado en una amplia variedad de aplicaciones, entre ellas destacan: Scribe [7, 29], SplitStream [6], Squirrel [16] y PAST [12].

Scribe es un sistema para la distribución multipunto desde el nivel de aplicación, desarrollado por los creadores de Pastry. Pastry ayuda a Scribe en la creación de grupos de nodos, en la gestión de los nodos en el grupo y, principalmente, permite a Scribe generar árboles de distribución multipunto con una sobrecarga mínima en cuanto a mensajes intercambiados se refiere. Pastry y Scribe están totalmente descentralizados, es decir, todas las decisiones se realizan con información local y cada nodo posee capacidades idénticas.

SplitStream es un sistema para la distribución/transmisión de contenido que requiere un alto ancho de banda. Dicho sistema está basado en Pastry. La idea clave de SplitStream es que la carga de reenvió se distribuye entre todos los participantes, aumentando la robustez del sistema cuando un nodo falla. Además, es capaz de asignar a cada nodo participante un ancho de banda diferente.

Squirrel es un sistema totalmente descentralizado basado en la idea de permitir a los navegadores web situados en máquinas locales compartir sus caches locales. El resultado es una cache web eficiente, altamente escalable y tolerante a fallos sin necesidad de un hardware específico o con un alto coste de administración. Este proporciona un mayor rendimiento, en términos de ancho de banda y latencia, en comparación con una cache web centralizada. Sin embargo, implica la imposición de una sobrecarga mínima en los nodos participantes.

PAST es un sistema *peer-to-peer* que almacena archivos de gran escala proporcionándolos escalabilidad, disponibilidad, seguridad y comparación de los recursos. Los archivos en PAST son inmutables y pueden ser compartidos discretamente por su propietario. PAST sigue una filosofía parecida a CFS, pero usando la red superpuesta Pastry. Intenta mejorar el sistema de CFS en almacenamiento de ficheros y *caching*, ayudándose de los servicios que ofrece Pastry, como son una mayor eficiencia y proximidad de las claves. Ofrece un sistema de ficheros de certificados para las operaciones básicas.

3.3.10 Conclusión

A modo resumen del diseño de Pastry:

- Topología de la red: anillo.
- Medida de proximidad: igualdad de prefijo, cuantos más dígitos tengan en común dos claves o identificadores, más próximos están.
- Detección de inconsistencias: el estado de un nodo se actualiza cuando se detecta que un nodo de la red no responde.

Tanto Pastry como Chord usan un espacio de identificadores circular y la complejidad del algoritmo de encaminamiento es O(log n). Los procesos que utiliza Chord cuando un nodo se incorpora a la red o falla son menos complejos que los que utiliza Pastry. El proceso de unión de un nodo en una red Pastry presenta una complejidad alta debido a que la tabla de

encaminamiento del nuevo nodo se rellena con los nodos que intervienen en los mensajes que se envían por la red para incorporar al nuevo nodo.

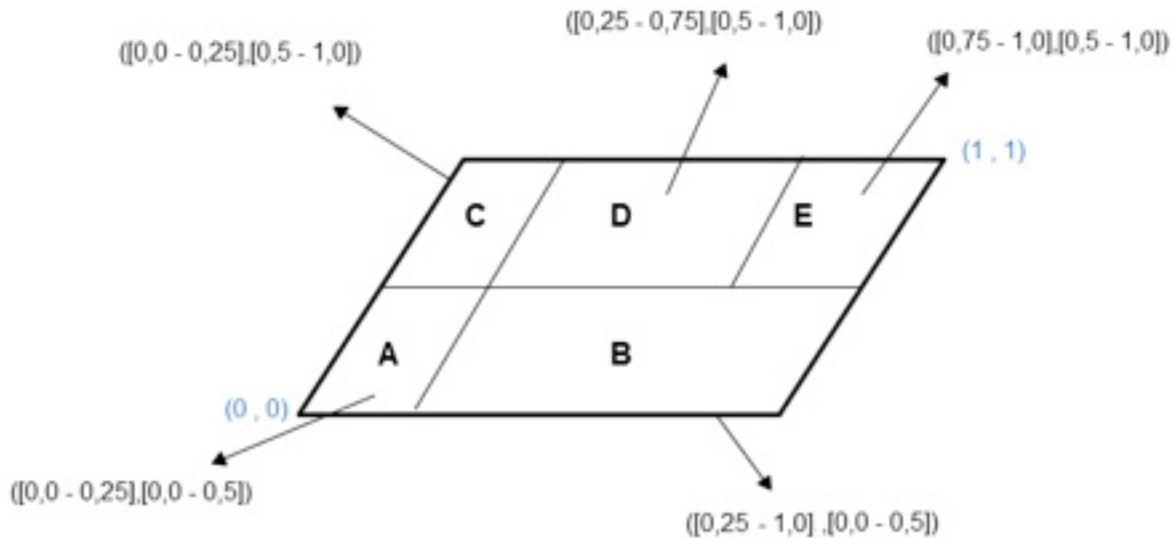
3.4 Content-Addressable Network (CAN)

CAN es una Tabla Hash Distribuida utilizada para la implementación de redes *peer-to-peer* estructuradas. CAN especifica la información que cada nodo de la red debe almacenar, referente a otros nodos, así como, la información de los ficheros que es compartida entre nodos. Además regula la comunicación y el intercambio de mensajes entre estos. CAN fue desarrollado en la Universidad de California por Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp y Scott Shenker [27].

3.4.1 Asignación de claves a nodos

Los nodos de una red CAN forman un espacio *d*-dimensional de coordenadas Cartesianas, donde *d* indica la dimensión del espacio. A cada nodo se le asigna una partición del espacio de coordenadas, de tal forma que a cada nodo le corresponde una zona única y diferenciada. Nótese, que las coordenadas de un nodo en el espacio virtual no están relacionadas con su posición física en la red. La figura 3.6 muestra un espacio de coordenadas 2-dimensional [0,1] x [0,1] compuesta por 5 nodos.

Figura 3.6 Representación de los nodos en CAN en un espacio 2-dimensional



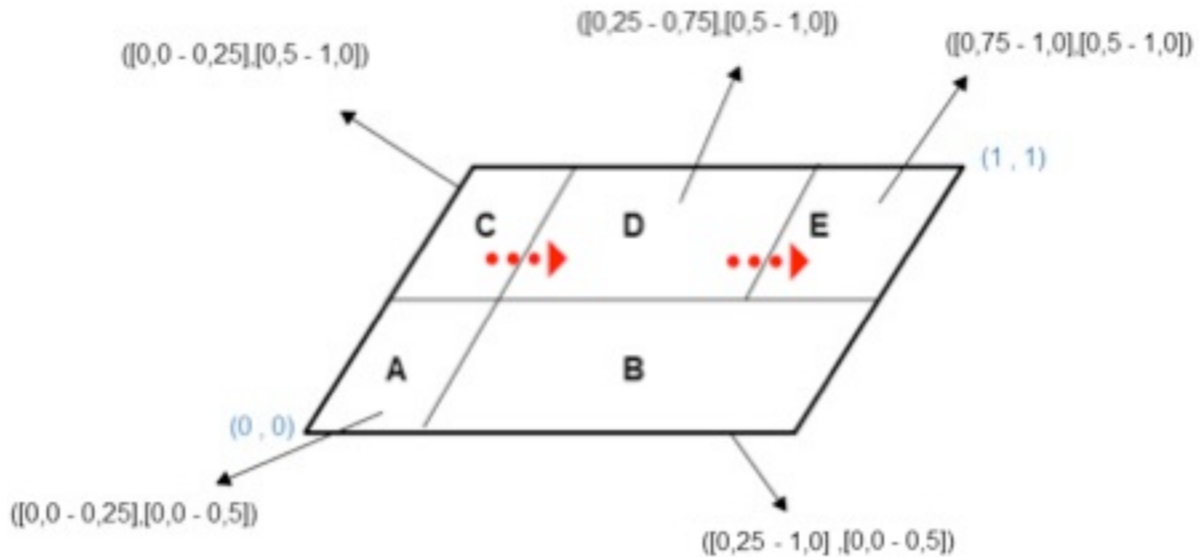
A cada clave *k* se le asigna una zona *z* del espacio de coordenadas mediante una función hash. La zona *z* corresponde a un nodo, el cual será el nodo que almacene la clave.

3.4.2 Tabla de encaminamiento

Cada nodo que forma parte de la red CAN mantiene una tabla de encaminamiento que almacena la información de las direcciones IP de sus nodos vecinos, así como, las zonas de coordenadas de las que son responsables. Dos nodos son vecinos si de entre sus *d* coordenadas coinciden todas menos una. Por ejemplo, en la red representada en la figura 3.6, el nodo A es vecino de los nodos C y B pero no del nodo D.

Todos los mensajes que se envían por la red CAN contienen el nodo destino final al que va dirigido el mensaje, además del nodo al que se envía el mensaje. Cuando un nodo recibe un mensaje cuyo nodo destino no coincide con sí mismo, éste lo reenvía a aquel nodo vecino cuyas coordinadas sean las más cercanas a las coordinadas del nodo destino. Por ejemplo, en la figura 3.7, el nodo C encamina el mensaje cuyo destino es el nodo E a través del nodo vecino D.

Figura 3.7 Representación de la ruta para encaminar un mensaje desde el nodo C hasta el nodo E



3.4.3 Algoritmo de búsqueda

Cuando se desea buscar una clave *k*, el identificador de la clave determina la zona del espacio de coordenadas donde se encuentra almacenada.

Un nodo desea buscar la información de la clave *k*. El nodo comprueba si el identificador de la clave coincide con la zona de la que es responsable el nodo. Si no coinciden, este reenvía la petición de búsqueda al nodo vecino cuyas coordenadas sean las más cercanas a las coordinadas del identificador de la clave hasta llegar al nodo responsable del área que contiene la zona buscada.

La complejidad del algoritmo de búsqueda es del orden de $O(d*n^{1/d})$, siendo *n* el número de nodos que forman parte de la tabla y *d* la dimensión del espacio de coordenadas.

3.4.4 Incorporar un nodo a la tabla

Cuando un nodo *n* quiere incorporarse a la red debe seguir tres pasos:

1. El nodo *n* debe contactar con un nodo que forme parte de la red CAN.
2. El nodo *n* debe encontrar una zona en la red CAN. Para ello, el nodo *n* elige una zona aleatoria *z* del espacio de coordenadas. A continuación, envía un mensaje *join* a la zona *z*. Este mensaje será reenviado por la red según el protocolo de encaminamiento hasta llegar al nodo responsable de la zona *z*.
   El nodo responsable de la zona z deberá dividir su zona en dos partes iguales asignando una de ellas al nodo *n*.
3. Por último, se notificará a los vecinos de la zona dividida de la existencia del nuevo nodo *n*. El nodo *n* aprende las direcciones IPs del nodo que dividió su zona en dos.

Las figuras 3.8 y 3.9 muestran el espacio de coordenadas bidimensional de una red CAN antes y después de que el nodo 4 se incorpore a la red. El nodo 4 elige aleatoriamente la zona 2 como espacio de coordenadas.

Figura 3.8 Estado del espacio de coordenadas bidimensional de una red CAN antes de la incorporación del nodo 4

Figura 3.9 Estado del espacio de coordenadas bidimensional de una red CAN después de la incorporación del nodo 4

3.4.5 Dar de baja un nodo de la tabla

Cuando un nodo abandona la red CAN entrega su zona y la información de su tabla de encaminamiento a uno de sus nodos vecinos, generando una zona válida si la zona del nodo que abandona es posible combinarla con la zona del nodo vecino. Sin embargo, en ocasiones, no es posible combinar ambas zonas, por lo que se selecciona al nodo vecino con la zona más pequeña. Este nodo será temporalmente el responsable de ambas zonas.

El problema surge cuando falla un nodo generando una zona inalcanzable. Para evitar esta situación, cada nodo envía periódicamente un mensaje a sus nodos vecinos para notificarles que sigue activo. Si un nodo no responde a dichos mensajes durante un cierto periodo de tiempo T, se asume que dicho nodo ha fallado y se activa el protocolo de reestructuración. El protocolo de reestructuración (*takeover*) asegura que uno de los nodos vecinos del nodo fallido tomará el control de su zona. Reasignada la zona a un nodo vecino, éste informa a sus vecinos del cambio para que refresquen la información de su tabla de encaminamiento.

3.4.6 Mejoras del sistema

Existen diferentes técnicas cuyo principal objetivo es reducir la latencia de encaminamiento [27]. Muchas de estas técnicas ofrecen robustez tanto en el encaminamiento como en la disponibilidad de los datos.

Una de las mejoras consiste en incrementar el número de dimensiones del espacio de coordenadas, *d*. Con esto, se consigue reducir la longitud de la ruta de encaminamiento y con ello, la latencia de ruta. Incrementar el número de dimensiones implica que un nodo tiene más vecinos, y con ello incrementa el número de saltos disponibles a elegir para generar la ruta de encaminamiento del mensaje.

Otra forma sencilla de mejorar el encaminamiento es disponer de múltiples espacios de coordenadas, *r*, cada uno de los cuales se llama realidad (*reality*). Cada nodo posee unas coordenadas diferentes y una lista de vecinos totalmente distinta para cada una de las realidades del sistema. A cada clave se le asigna una zona en cada una de las realidades, por lo que está almacenada por *r* nodos distintos. Con múltiples realidades se mejora la tolerancia a fallo durante el encaminamiento, debido a que en caso de que se produzca un fallo durante el encaminamiento en una realidad, los mensajes podrían continuar su encaminamiento haciendo uso de otra de las realidades.

En el diseño original cada zona del espacio de coordenadas es asignada a un único nodo del sistema. Una posible mejora sería permitir que una zona sea asignada a múltiples nodos, donde *maxpeers* indica el número de nodos máximos por zona. Las ventajas que conlleva esta mejora son: reducir el número de saltos en el encaminamiento, reducir la latencia por salto y mejorar la tolerancia a fallos.

La incorporación de estas técnicas mejoraría significativamente el diseño de la Tabla Hash Distribuida CAN. Sin embargo, también acarrean un incremento del coste y de la complejidad. Por tanto, antes de modificar el diseño original de la tabla hash es conveniente hacer un estudio de los requerimientos de la aplicación para la cual va a ser utilizada.

3.4.7 Aplicaciones

Sistemas de gran escala de gestión de almacenamiento tales como OceanStore [17], Farsite [4] y Publius [37] utilizan la Tabla Hash Distribuida CAN. Todos estos sistemas requieren una eficiente inserción y recuperación del contenido.

3.4.8 Conclusión

A modo resumen del diseño de CAN:

- Topología de la red*:* espacio de coordenadas Cartesianas multidimensional.
- Medida de proximidad: dos nodos son vecinos si de entre sus d coordenadas se solapan todas menos una. Esta medida de proximidad es usada para el encaminamiento de mensajes.
- Detección de inconsistencias: algoritmo de reestructuración (*takeover*) garantiza que cuando un nodo no responde alguno de los nodos vecinos del nodo toma el control de la zona.

3.5 Otras THDs

En los últimos años han sido presentadas una amplia gama de Tablas Hash Distribuidas. Ejemplos existentes de este tipo de sistemas han sido presentadas previamente (Kademlia [21], Chord [34], Pastry [28] y CAN [27]). Existen en la literatura muchos más diseños de Tablas Hash Distribuidas, la mayoría de ellos basados sobre alguna de las Tablas Hash Distribuidas explicadas con el fin de mejorar su eficiencia bajo condiciones concretas de uso. En esta sección se introducen las Tablas Hash Distribuidas: Epichord [18], Viceroy [20] y Kelips [15].

3.5.1 EpiChord

EpiChord es una Tabla Hash Distribuida desarrollada por Ben Leong, Barbara Liskov y Erik D. Demaine [18]; diseñada para mejorar el rendimiento de la Tabla Hash Distribuida Chord.

Epichord, al igual que Chord, está organizado en un espacio de direcciones unidimensional y circular donde a cada nodo se le asigna un identificador único generado mediante la función hash SHA-1.

Una de las principales diferencias entre Epichord y Chord es el algoritmo de encaminamiento. Epichord implementa un protocolo de encaminamiento llamado *reactive routing,* que utiliza una memoria caché de tamaño indefinido en cada nodo cuyas entradas tienen asociado un tiempo. Las entradas de la memoria caché se aprenden de dos formas:

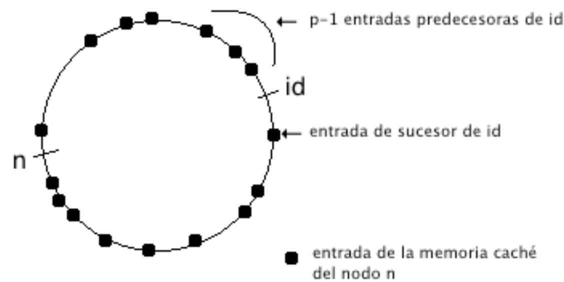
- Cuando un nodo se une a la red, obtiene la cache completa de su predecesor y de su sucesor.
- Cuando un nodo recibe una consulta, comprueba si el nodo solicitante está en la memoria caché. Si no es así, lo añade a la memoria y establece un tiempo a dicha entrada con la hora local que tiene el nodo solicitante.

Este protocolo de encaminamiento permite mantener el estado de las tablas de encaminamiento actualizadas mediante la observación del tráfico de la red sin necesidad de un proceso de estabilización como ocurre en Chord.

El algoritmo de búsqueda de Epichord está basado en búsquedas en paralelo. Cuando un nodo n desea buscar el nodo con identificador *id*, envía p consultas en paralelo a las entradas de su memoria caché que apuntan al sucesor del nodo con identificador *id* y a los p-1 nodos que preceden al nodo con identificador *id.* A continuación, los nodos que reciben la consulta envían al nodo n los "mejores" nodos que tienen en su memoria caché para que el nodo n pueda continuar la búsqueda. Nótese que en caso de que los nodos contengan en su caché el nodo con identificador *id*, estos se lo mandarán al nodo n. El número de saltos necesarios para localizar un

identificador es de uno o dos saltos en la mayoría de los casos. La figura 3.10 muestra las entradas iniciales que tiene la memoria caché del nodo *n* para realizar la búsqueda del nodo con identificador *id*.

Figura 3.10 Entradas iniciales de la memoria caché del nodo *n*



3.5.2 Viceroy

Viceroy se caracteriza por proporcionar un servicio de búsqueda completamente distribuido y escalable para las redes *peer-to-peer*. Viceroy fue desarrollado en la Universidad de California y en la Universidad Kebrew por Dahlia Malkshi, Moni Naor y David Ratajczak [20].

A cada nodo participante se le asocian dos valores que determinan su conexión con el resto del sistema: un identificador, cuyos valores oscilan entre 0 y 1; y un nivel *l,* determinado por un entero positivo. Cada clave se almacenada en el nodo cuyo identificador sea el más próximo al identificador de la clave.

Una de las principales diferencias entre Viceroy y el resto de Tablas Hash Distribuidas es la topología que utiliza para organizar su espacio de direcciones. Los identificadores que forman la red Viceroy se representan con una red mariposa (*butterfly*) formada por tres clases de uniones: un anillo general, donde cada nodo es conectado con su sucesor y su predecesor; anillos de niveles, donde nodos del mismo nivel están conectados unos con otros formando un anillo; y

la mariposa donde cada nodo hoja del nivel l señala a dos nodos "abajo" del nivel l+1 y cada nodo perteneciente al nivel l >1 señala al nodo "arriba" de los nodos del nivel l-1.

El proceso de encaminamiento está dividido en tres fases distintas:

1. Escalar ascendentemente por la red hacia el nivel l-1 del nodo.
2. Descender por la red, del nivel l al nivel l+1, hasta llegar a un nodo sin enlaces hacia debajo de la red, el cual supuestamente se encuentra en las proximidades del objetivo.
3. Por último, se realiza la búsqueda utilizando el anillo y el anillo de niveles hasta que se alcanza el objetivo.

La complejidad de la operación de búsqueda es del orden de O(log n), siendo n el número de nodos que forman parte de la tabla [20].

Viceroy está diseñado para ayudar a las redes extremamente escalables, dinámicas y con un alto nivel de tráfico.

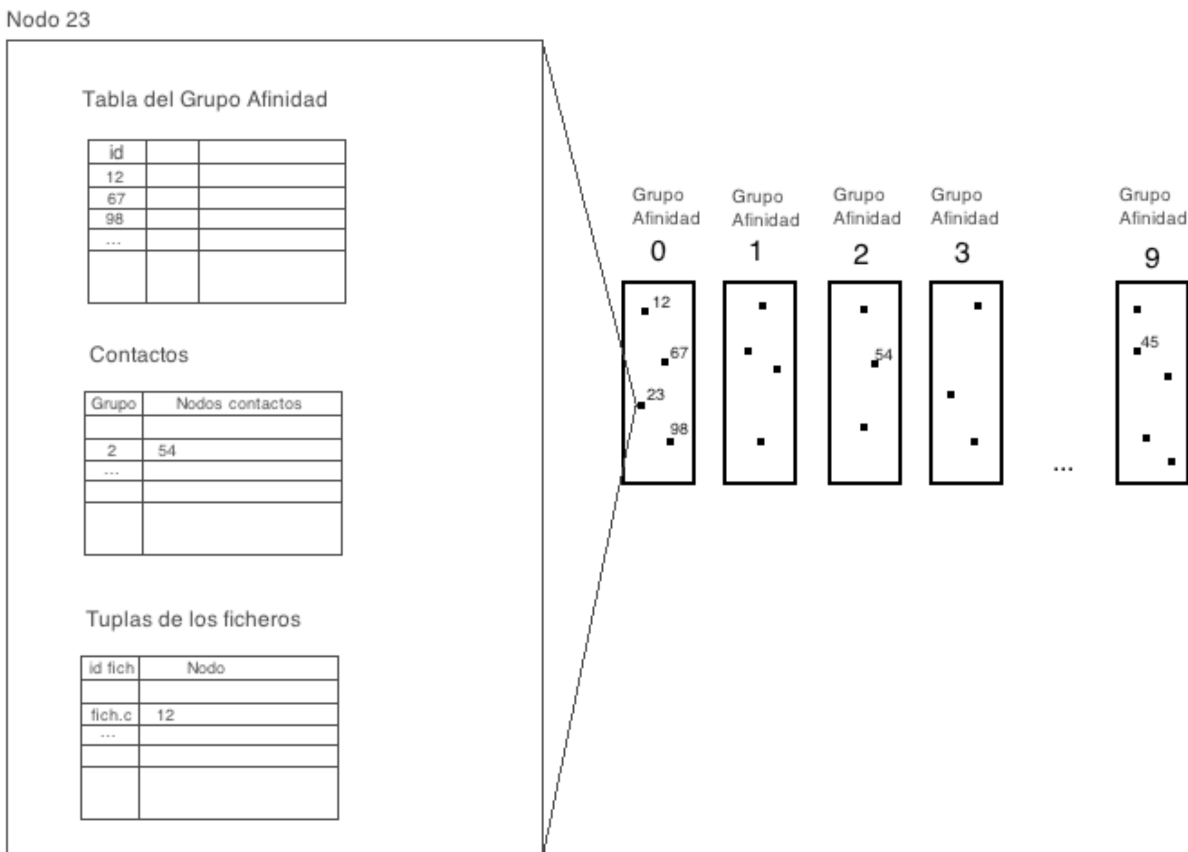
3.5.3 Kelips

Kelips fue desarrollado en la Universidad Cornell de Nueva York por Indranil Gupta, Ken Birman, Prakash Linga, Al Demers y Robbert van Renesse [15].

Kelips usa un espacio de direcciones de O($\sqrt{n}$) por nodo, siendo n es el número de nodos en el sistema, el cual está dividido en p grupos de afinidad. Cada nodo pertenece a un grupo de afinidad y su estado está formado por: la tabla del grupo de afinidad, que contiene los nodos que pertenecen a su grupo de afinidad; contactos, tabla que almacena los nodos contacto de los otros grupos de afinidad; y tuplas de los ficheros, tabla que almacena el nombre de los ficheros y las direcciones IP de los nodos que los almacenan. En la figura 3.11 se muestra el estado del nodo 23 que forma parte de una red Kelips compuesta por 10 grupos de afinidad. Al igual que en otras

Tablas Hash Distribuidas, los identificadores de nodos son cadenas de bits generadas por la función hash SHA-1 cuyo rango de valores oscila desde 0 hasta *p*-1.

Figura 3.11 Representación del estado del nodo 23 en una red Kelips compuesta por 10 grupos de afinidad



Cuando un nodo *n* desea buscar un archivo *f* , busca el grupo de afinidad al que pertenece *f*. A continuación, envía una petición de búsqueda de *f* al nodo contacto más cercano topológicamente que conoce de ese grupo de afinidad. Este busca a *f* en su tabla de tuplas de los ficheros y envía un mensaje al nodo que inicio la búsqueda con la dirección del nodo que almacena *f*.

Kelips reduce el tiempo y la complejidad de las búsquedas a un único salto. Kelips proporciona un buen equilibrio de la carga del sistema en comparación con las otras tablas estudiadas.

Cuando un nodo quiere insertar un archivo *f*, busca el grupo de afinidad al que pertenece *f*. A continuación, envía una petición de inserción al nodo contacto más cercano topológicamente que conoce de ese grupo de afinidad. Este elige aleatoriamente un nodo *x* de su grupo de afinidad y le reenvía la petición de inserción. El nodo *x* es ahora el nodo que almacena el archivo. Así pues, la complejidad en tiempo y mensajes de la operación de inserción es del orden de O(1).

Kelips destaca por tener una estructura ligera, y por su alta tolerancia a fallos. Sin embargo, para lograr este rendimiento incrementa de forma significativa el uso de memoria y la comunicación en segundo plano. Además, la flexibilidad del estado del nodo permite búsquedas e inserciones satisfactorias.

3.6 Comparativa de las Tablas Hash Distribuidas estudiadas

A continuación se muestra una tabla comparativa de las Tablas Hash Distribuidas estudiadas.

Tabla 3.4 Resumen de las características generales de las Tablas Hash Distribuidas estudiadas

	Kademlia	**Chord**	**Pastry**	**CAN**	**Epichord**	**Viceroy**	**Kelips**
Topología de la red	Árbol binario	Anillo	Anillo	Espacio de coordenadas Cartesianas multidimensional	Anillo	Mariposa	Multicapas
Encaminamiento	O(log n)	O(log n)	O(log n)	$O(d*n^{1/d})$	O(1)	O(log n)	O(1)
Tamaño de la Tabla de encaminamiento	O(log n)	O(log n)	O(log n)	O(d)	O(log n)	7	$O(\sqrt{n})$
Aplicaciones	Emule, aMule y BitTorrent	DhaFS, UsenetDHT y OverCite	Scribe, SplitStream, Squirrel y PAST	DNS, OceanStore, Publius y Peer-to-Peer file sharing systems.	---	---	Kache

Capítulo 4 - Especificación del protocolo Chord en Maude

La estructura de este capítulo es la siguiente. En primer lugar, la sección 4.1 sirve como una breve introducción del lenguaje Maude. A continuación, la sección 4.2 aborda ciertos detalles de la especificación con el fin de facilitar al lector su comprensión. En la sección 4.3 se describen los distintos módulos implementados para la especificación. Por último, la sección 4.4 explica los procesos básicos que se llevan a cabo entre los nodos que conforman el anillo.

4.1 El lenguaje de especificación Maude

Maude es un lenguaje de especificación formal desarrollado a finales de los años noventa por un grupo de investigadores dirigidos por J. Meseguer en el *Computer Science Laboratory* de *SRI International* (EE.UU.) [23].

El lenguaje Maude está basado en la lógica de reescritura. Esta lógica es una lógica para representar sistemas concurrentes que tienen estados y evolucionan por medio de transiciones. Los estados se definen como estructuras de datos en la lógica ecuacional de pertenencia subyacente a la lógica de reescritura y las transiciones se especifican mediante reglas de reescritura.

Maude integra la programación orientada a objetos dentro de la programación funcional por medio del módulo predefinido CONFIGURATION, en el cual se declaran los tipos necesarios para definir los objetos y los mensajes. Maude permite especificar tanto el paso de mensajes asíncrono, como diversos patrones de sincronización de objetos y mensajes.

El lenguaje básico proporcionado por Core Maude ha sido ampliado mediante módulos definidos en el propio lenguaje al lenguaje Full Maude. Este lenguaje proporciona muchas facilidades, entre ellas, una sintaxis amigable para la programación orientada a objetos. Sin

embargo en este trabajo no hemos utilizado esta ampliación. La especificación está realizada íntegramente en Core Maude.

Existen otras ampliaciones del lenguaje como la proporcionada por el módulo REAL-TIME-MAUDE que permite especificar la noción del tiempo, o el módulo MODEL-CHECKER que permite probar propiedades temporales de los sistemas mediante técnicas de *model checking*. El uso de estas facilidades del lenguaje se deja como trabajo futuro.

En Maude, las unidades básicas de especificación son llamadas módulos. Existen dos tipos de módulos: módulos funcionales (fmod) y módulos de sistema (mod). Además Full Maude proporciona un tercer tipo: módulos orientados a objetos (omod).

Los módulos funcionales en Maude definen tipos de datos y operaciones sobre ellos por medio de teorías ecuacionales.

Para reducir un término hasta su forma canónica se usa la instrucción:

red *TérminoAReducir.*

Un módulo funcional se declara usando las siguientes palabras claves:

fmod *<Nombre del módulo>* ***is***
<Definiciones y declaraciones>
endfm

En Maude, la definición de los tipos se realiza con la palabra reservada *sort* seguida del nombre del tipo un espacio en blanco y un punto. Se pueden realizar declaraciones de subtipo con la palabra reservada *subsort* y declaraciones de pertenencia mediante la palabra reservada *mb* seguida de la definición de un término, dos puntos y el tipo al que pertenecen. Las declaraciones de pertenencia pueden ser condicionales, declarándose en este caso con la palabra reservada *cmb* e indicando la condición mediante la palabra reservada *if*.

La definición de una operación empieza por la palabra reservada *op*, seguida del nombre de la operación, dos puntos, la lista de los tipos de los parámetros separados por espacios en blanco, el símbolo ->, el tipo del resultado y un punto. Todas las operaciones tienen que devolver un valor y sólo uno, pero no es necesario que tenga parámetros. Las constantes se definen como operaciones que no tienen ningún argumento. Las operaciones se pueden definir utilizando notación infija en la cual se utiliza el símbolo del subrayado para indicar la posición del parámetro o con notación prefija. El comportamiento de las operaciones se define por medio de ecuaciones, que comienzan con la palabra reservada *eq* y pueden ser condicionales, en cuyo caso comienzan con la palabra reservada *ceq*. Las ecuaciones pueden utilizar variables cuyo tipo se puede declarar mediante cláusulas que comienzan con la palabra reservada *var* o en las mismas ecuaciones añadiendo el tipo a la variable precedido de dos puntos.

Los módulos funcionales terminan con la palabra reservada *fmod*. Un módulo puede incluir a otros. Existen tres formas de incluir los módulos para las que se utilizan las palabras reservadas *protecting*, *extending* e *including*.

Los módulos de sistema representan una teoría de la lógica de reescritura. Además de las declaraciones y definiciones utilizadas en los módulos funcionales, en estos módulos se definen las reglas de reescritura, las cuales pueden ser condicionales o no condicionales.

Las reglas especifican las transiciones locales concurrentes que puede tener lugar en un sistema, si el patrón en la regla de la izquierda coincide con un fragmento del estado del sistema y el estado de la condición se cumple. En ese caso, la transición especificada por la regla puede darse, y el fragmento combinado del estado se transforma en la instancia correspondiente de la parte derecha.

Un módulo de sistema se declara usando las siguientes palabras claves:

mod *<Nombre del módulo>* ***is***

<Definiciones y declaraciones>

endm

La representación de reglas incondicionales se declaran con la siguiente estructura:

rl *[(Etiqueta)]* ***:*** *(Term-1)* **=>** *(Term-2) .*

La representación de reglas condicionales se declaran con la siguiente estructura:

crl *[(Etiqueta)]* ***:*** *(Term-1)* **=>** *(Term-2)*
if *(Condición-1) /\ ... /\ (Condición-k) .*

Los módulos funcionales y de sistema pueden ser parametrizados. Los requisitos del parámetro se especifican por medio de teorías. Estas tienen la forma:

fth *<Nombre de la teoría>* is
<Declaraciones y definiciones>
endfth

El sistema proporciona en el archivo *prelude.maude* la definición de las teorías más comunes.

El módulo se instancia mediante la definición de una vista, la cual relaciona el parámetro abstracto con el parámetro real. La definición de una vista es de la forma:

view *<Nombre de la vista>* ***from*** *<Nombre de la teoría>* ***to*** *<Nombre del módulo>* ***is***
<Declaraciones>
endv

Los módulos parametrizados definen el parámetro a continuación del nombre del módulo, indicando la teoría que lo define:

fmod *<Nombre del módulo>* ***{****X* ***::*** *<Nombre de la teoría>****} is***

<*Definiciones y declaraciones*>

endfm

El parámetro se representa en el módulo mediante el nombre dado en la cabecera seguido del símbolo del dólar seguido del nombre del tipo en la teoría. Por ejemplo, la teoría TRIV que define un tipo Elt sin ningún requisito. Para referenciar el parámetro en el módulo utilizamos el término X$Elt.

Para instanciar un módulo parametrizado nombramos el módulo y a continuación entre paréntesis el nombre de la vista que lo instancia.

En esta sección hemos presentado aquellos elementos del lenguaje Maude necesarios para comprender la especificación realizada. Para más información sobre la sintaxis u otros aspectos del lenguaje Maude referirse a [8].

4.2 Detalles de la especificación

Una vez descrito como es el funcionamiento teórico del protocolo Chord y las nociones básicas del lenguaje Maude, pasaremos a presentar la especificación de la Tabla Hash Distribuida Chord en Maude. La especificación se realiza sobre el protocolo descrito de forma informal en [34]. El código de la especificación se encuentra disponible en http://maude.sip.ucm.es/chord.

En esta sección se abordan ciertos aspectos generales de la especificación. Más detalles sobre los tipos de datos, clases y mensajes del protocolo se pueden encontrar en las siguientes secciones de este capítulo.

El protocolo Chord está especificado en Maude como una configuración de objetos y mensajes. Los nodos son objetos de clase *peer*, definida como:

```
subsort Info-Node < Oid .

op Peer : -> Cid .

op FT :_ : FingerTable -> Attribute [ctor  gather(&)] .
op KT :_ : KeysTable -> Attribute [ctor  gather(&)] .
op SL :_ : SuccessorList -> Attribute [ctor  gather(&)] .
op PL :_ : PredecessorList -> Attribute [ctor  gather(&)] .
```

donde cada objeto está identificado por su identificador del tipo *Node-ID* y su dirección IP del tipo *Node-IP*, dando lugar al tipo *Info-Node*.

Cada nodo tiene una lista de atributos, los cuales son:

- FingerTable. Almacena en una tabla la información de la tabla *finger* del nodo.
- KeysTable. Almacena en una tabla la información de las claves de las que es responsable el nodo.
- SuccessorList. Almacena en una lista la información de los sucesores del nodo.
- PredecessorList. Almacena en una lista la información de los predecesores del nodo.

Es muy importante mantener la estructura del anillo para un correcto funcionamiento del protocolo Chord, lo cual significa que la información de las tablas y de las listas debe contener en todo momento una información válida. Para ello, los nodos se envían y reciben información mediante el paso de mensajes. Los mensajes se declaran en Maude como operaciones del tipo *msg*.

Los mensajes definidos en esta especificación han sido implementados siguiendo una política: el primer parámetro del mensaje indica el nodo que recibe el mensaje, y el segundo parámetro del mensaje indica el nodo que envía el mensaje. Por ejemplo, en el mensaje FIND-SUCCESSOR definido como sigue:

op FIND-SUCCESSOR : Info-Node Info-Node -> Msg .

el primer parámetro del tipo *Info-Node* es el nodo que recibe el mensaje FIND-SUCCESSOR y el segundo parámetro del tipo *Info-Node* es el nodo que envía el mensaje para encontrar su sucesor.

El uso de este tipo de mensajes permite simular una comunicación asíncrona entre los nodos. En la versión de la especificación no se tiene en cuenta el tiempo transcurrido en el envió del mensaje.

Las reglas de reescritura especifican de forma declarativa el comportamiento asociado a los mensajes describiendo los procesos básicos que se pueden dar entre los nodos de una red Chord.

Para la realización de la especificación ha sido necesario ampliar dos módulos del sistema MAP y LIST predefinidos por Maude en el fichero *prelude.maude*.

El módulo MAP se amplía en el módulo MAP+. Las operaciones añadidas son:

- op deleteEntryMap : X$Elt Map{X,Y} -> Map{X,Y} . Dados como parámetros de entrada una entrada de la tabla y la tabla, la operación *deleteEntryMap* devuelve como resultado la tabla sin la entrada. Nótese que si la entrada no está en la tabla, se devuelve como resultado la tabla sin modificaciones.
- op appendMap : Map{X,Y} Map{X,Y} -> Map{X,Y} . Dados como parámetros de entrada dos tablas, la operación *appendMap* devuelve el resultado de concatenar las dos tablas. Las tablas de entrada deben de ser disjuntas.
- op deleteMap : Map{X,Y} Map{X,Y} -> Map{X,Y} . Dados como parámetros de entrada dos tablas, la operación *deleteMap* devuelve el resultado de eliminar en la segunda tabla las entradas de la primera. Las entradas de la primera tabla que no se encuentran en la segunda no se consideran.

El módulo LIST se amplía en el módulo LIST+. Las operaciones añadidas son:

- op deleteList : X$Elt List{X} -> List{X} . Dados como parámetros de entrada un elemento de la lista y la lista, la operación *deleteList* devuelve la lista sin el elemento. Nótese que si el elemento no está en la lista, se devuelve como resultado la lista sin modificaciones, y si el elemento está varias veces en la lista se eliminan todas ellas.

El módulo NAT se amplía en el módulo NAT+. La operación añadida es:

- op exp : Nat -> Nat . La operación *exp* recibe como parámetro un número natural y devuelve como resultado la potencia de dos elevada al número natural pasado como parámetro de entrada.

4.3 Módulos de la especificación

A continuación se explican cada uno de los módulos definidos en la especificación de la Tabla Hash Distribuida Chord en Maude.

4.3.1 BitString

El módulo BITSTRING es un módulo funcional que define el tipo *Bit* y el tipo *BitString*. El tipo *Bit* es un dígito cuyo posible valor es 0 ó 1. El tipo *BitString* es una cadena de bits separados por punto y coma. La cadena vacía se representa con la constante *null*. A continuación, se presentan las definiciones de los tipos *Bit* y *BitString* junto con sus operaciones constructoras.

```
sort Bit .
sort BitString .
subsort Bit < BitString .
```

```
op 0 : -> Bit [ctor] .
op 1 : -> Bit [ctor] .
op _;_ : BitString Bit -> BitString [ctor] .
op null : -> BitString [ctor] .
```

Las operaciones definidas que actúan sobre las cadenas de bits son:

- op successor : BitString -> BitString . La operación *successor* obtiene la cadena de bits que representa al sucesor del valor de entrada.
- op predecessor : BitString -> BitString . La operación *predecessor* obtiene la cadena de bits que representa al predecesor del valor de entrada.
- op length : BitString -> Nat . La operación *length* recibe como parámetro una cadena de bits y devuelve como resultado el número de bits de dicha cadena. Nótese que los bits a cero a la izquierda de la cadena también forman parte de ella.
- op decToBin : Nat Nat -> BitString . La operación *decToBin* recibe como parámetros dos números naturales. El primero indica el número expresado en base decimal que se quiere pasar a binario y el segundo parámetro indica la longitud que tendrá la cadena de bits resultante. Dicha longitud debe ser suficiente para representar el número en base binaria. En caso contrario la función no está definida.
- op _<_ : BitString BitString -> Bool .
  op _>_ : BitString BitString -> Bool .
  op _>=_ : BitString BitString -> Bool .
  op _<=_ : BitString BitString -> Bool .
  Cada uno de los operadores relacionales <, >, <= y >= recibe como parámetros dos cadenas de bits del tipo *BitString* devolviendo el resultado de comparar la primera cadena con la segunda.
- op first : BitString -> Bit . La operación *first* devuelve como resultado el bit más significativo de la cadena.
- op elimFirst : BitString -> BitString . La operación *elimFirst* devuelve como resultado la cadena de bits de entrada sin el bit más significativo.

- op elimZeros : BitString -> BitString . La operación *elimZeros* devuelve como resultado la cadena de bits de entrada sin ceros por la izquierda.
- op zerosIzq : BitString -> Bool . La operación *zerosIzq* devuelve como resultado verdadero si existen ceros por la izquierda en la cadena de bits de entrada y falso en caso contrario.
- op resta : BitString BitString -> BitString . La operación *resta* devuelve como resultado la cadena de bits resultante de restar los valores representados por las cadenas de bits de entrada.
- op suma : BitString BitString -> BitString . La operación *suma* devuelve como resultado la cadena de bits resultante de sumar los valores representados por las cadenas de bits de entrada.
- op C2 : BitString -> BitString . La operación *C2* devuelve como resultado el complemento a dos del valor representado por la cadena de bits de entrada.
- op ChangeBits : BitString -> BitString . La operación *changeBits* devuelve como resultado la cadena de bits de entrada con los unos cambiados por ceros y los ceros cambiados por unos.
- op firstBit1 : BitString Nat -> Nat . La operación *firstBit1* devuelve el número natural que se obtiene de calcular la potencia de dos elevada a la posición en la que se encuentra el primer bit a uno empezando por el bit más significativo de la cadena de bits de entrada. El número natural de los parámetros de entrada indica la longitud de la cadena de bits de entrada.
- op BitStringToDec : BitString Nat -> Nat . La operación *BitStringToDec* devuelve como resultado el número natural que se obtiene al convertir la cadena de caracteres de bits a un número decimal. El número natural de los parámetros de entrada indica la longitud de la cadena de bits de entrada.
- op posBit1 : BitString Nat -> Nat . La operación *posBit1* devuelve como resultado un número natural que indica la posición en la que se encuentra el primer bit a uno empezando por el bit más significativo de la cadena de bits de entrada. El número natural de los parámetros de entrada indica la longitud de la cadena de bits de entrada.
- op modulo : BitString Nat -> BitString . La operación *modulo* devuelve como resultado el módulo 2^{Nat} de la cadena de bits de entrada.

4.3.2 Node-ID

El módulo NODE-ID es un módulo funcional que define el identificador de un nodo. El Node-ID es una cadena de bits del tipo *BitString* de una determinada longitud. La longitud de la cadena viene definida por la constante IDBits. En una red Chord real, esta constante tendrá el valor 128. En los ejemplos realizados en este trabajo se utiliza un valor de 8 bits para facilitar la legibilidad de las pruebas. A continuación se muestra la definición del tipo Node-ID y de la constante IDBits en Maude.

```
sort Node-ID .
subsort Node-ID < BitString .
op IDBits : -> Nat .
eq IDBits = 8 .
```

4.3.3 Node-IP

El módulo NODE-IP es un módulo funcional que define la dirección IP de un nodo. En los ejemplos realizados las direcciones IP de los nodos están definidas por cadenas de bits del tipo 127.0.0.*x* donde *x* es el valor en decimal del identificador del nodo del tipo Node-ID. Sin embargo, cualquier otra cadena también sería válida. A continuación se muestra la definición de la clase Node-IP en Maude.

```
sort Node-IP .
subsort String < Node-IP .
```

4.3.4 Info-Node

El módulo INFO-NODE es un módulo funcional que define el tipo información de un nodo. La información de un nodo está compuesta por su identificador del tipo Node-ID y su dirección

IP del tipo Node-IP, los cuales están separados por el carácter @. A continuación se muestra la definición del tipo Info-Node y de su operación constructora en Maude.

```
sort Info-Node .
op _@_ : Node-ID Node-IP -> Info-Node [ctor] .
```

Las operaciones definidas que actúan sobre la información de un nodo son:

- op getNodeID : Info-Node -> Node-ID . La operación *getNodeID* devuelve como resultado el identificador del nodo.
- op getNodeIP : Info-Node -> Node-IP . La operación *getNodeIP* devuelve como resultado la dirección IP del nodo.

4.3.5 KeysTable

El módulo KEYSTABLE es un módulo funcional que define la tabla de claves de un nodo. Para ello definimos dos vistas (vBITSTRING y vINFO-NODE) que utilizan la teoría TRIV predefinida en el fichero *preludio.maude*. La especificación de la vista vBITSTRING en que se relaciona el tipo BitString con la teoría TRIV es la siguiente:

```
view vBitString from TRIV to BITSTRING is
  sort Elt to BitString .
endv
```

La especificación de la vista vINFO-NODE en que se relaciona el tipo Info-Node con la teoría TRIV es la siguiente:

```
view vInfo-Node from TRIV to INFO-NODE is
  sort Elt to Info-Node .
endv
```

Por último se instancia el módulo parametrizado MAP+ con los nombres dados a las vistas:

```
including MAP+{vBitString,vInfo-Node} *
                                        (sort Map{vBitString,vInfo-Node} to KeysTable,
                                         op insert to insertKT,
                                         op deleteMap to deleteKT,
                                         op deleteEntryMap to deleteEntryKT,
                                         op appendMap to appendKT ) .
```

dando como resultado tablas de claves donde cada entrada está formada por la clave del fichero, la cual es una cadena de bits del tipo BitString, y la información del nodo que publicó el fichero a la red del tipo Info-Node.

Las operaciones se renombran para poder diferenciarlas de las operaciones sobre la tabla de claves.

4.3.6 FingerTable

El módulo FINGERTABLE es un módulo funcional que define la tabla *finger* de un nodo. Para ello se instancia el módulo MAP+ con dos vistas (NAT e vINFO-NODE), donde la vista NAT está predefinida en el fichero *prelude.maude*.

```
including MAP+{Nat,vInfo-Node} * (sort Map{Nat,vInfo-Node} to FingerTable,
                                  op undefined to undefinedFT,
                                  op insert to insertFT,
                                  op deleteMap to deleteFT,
                                  op deleteEntryMap to deleteEntryFT,
                                  op appendMap to appendFT ) .
```

Las operaciones se renombran para poder diferenciarlas de las operaciones sobre la tabla *finger*.

El número de entradas de la tabla *finger* viene determinado por la longitud del identificador del nodo (IDBits). Cada una de las entradas está formada por un numero natural, 2^i con $0 <= i <=$ IDBits-1 , y la información del nodo (successor(Node-ID + $2^{(i-1)}$)) que es del tipo Info-Node .

4.3.7 SuccessorList

El módulo SUCCESSORLIST es un módulo funcional que define la lista de sucesores de un nodo. Para su especificación se utiliza el módulo LIST+.

```
including LIST+{vInfo-Node} * (sort List{vInfo-Node} to SuccessorList,
                                sort NeList{vInfo-Node} to NeListSuccessor,
                                op nil to nilSuccessor,
                                op append to appendSuccessor,
                                op head to headSuccessor,
                                op tail to tailSuccessor,
                                op last to lastSuccessor,
                                op front to frontSuccessor,
                                op occurs to occursSuccessor,
                                op reverse to reverseSuccessor,
                                op $reverse to $reverseSuccessor,
                                op size to sizeSuccessor,
                                op $size to $sizeSuccessor,
                                op deleteList to deleteListSucc,
                                op __ to _&_ ) .
```

Las operaciones se renombran para poder diferenciarlas de las operaciones definidas en la lista de sucesores.

4.3.8 PredecessorList

El módulo PREDECESSORLIST es un módulo funcional que define la lista de predecesores de un nodo. Para su especificación se utiliza el módulo LIST+.

```
including LIST+{vInfo-Node} * (sort List{vInfo-Node} to PredecessorList,
                               sort NeList{vInfo-Node} to NeListPredecessor,
                               op nil to nilPredecessor,

                               op append to appendPredecessor,
                               op head to headPredecessor,
                               op tail to tailPredecessor,
                               op last to lastPredecessor,
                               op front to frontPredecessor,
                               op occurs to occursPredecessor,
                               op reverse to reversePredecessor,
                               op $reverse to $reversePredecessor,
                               op size to sizePredecessor,
                               op $size to $sizePredecessor,
                               op deleteList to deleteListPred,
                               op __ to _||_ ) .
```

Las operaciones se renombran para poder diferenciarlas de las operaciones definidas en la lista de predecesores.

4.3.9 PeerOperations

El módulo PEEROPERATIONS es un módulo funcional en el cual se definen las operaciones que interactúan con los atributos de los nodos (tabla de claves, tabla *finger*, lista de sucesores y lista de predecesores). Estas operaciones son:

- op createNode : Nat -> Info-Node . La operación *createNode* genera la información relacionada con un nuevo nodo tomando como identificador del nodo el número natural que le pasan como parámetro de entrada.
- op closestPrecedingNode : Info-Node Info-Node FingerTable Nat -> Info-Node . Dados dos nodos como parámetros de entrada; N1 y N2 respectivamente, la operación *closestPrecedingNode* obtiene el nodo más cercano a N1 que hay en la tabla *finger* de N2. El número natural indica la entrada de la tabla *finger* que estamos explorando. El valor con el que debe llamarse a la función es con el valor que tiene la última entrada de la tabla *finger*, para nuestro ejemplo 128.
- op closestNodeKey : Info-Node FingerTable BitString -> Info-Node . La operación *closestNodeKey* tiene como parámetros: la información de un nodo, su tabla *finger* y la cadena de bits de una clave. La operación devuelve el nodo más cercano a la clave que hay en la tabla *finger*. Para ello, se siguen tres pasos:
    i. Restar el valor de la clave menos el identificador del nodo.
    ii. Buscar la potencia de dos más cercana al resultado de la resta.
    iii. Devolver la información del nodo que está almacenado en la entrada de la tabla.
- op buildTF : Info-Node SuccessorList FingerTable Nat -> FingerTable . La operación *buildTF* tiene como parámetros: la información de un nodo, su lista de sucesores, su tabla *finger* y un número natural. El número natural indica la entrada de la tabla *finger* que estamos generado. El valor que debe tener cuando se llama a la función es 1. Esta operación genera la tabla *finger* del nodo que le pasan como parámetro de entrada haciendo uso de su lista de sucesores.
- op closestNodeLS : Info-Node SuccessorList Info-Node -> Info-Node . La operación *closestNodeLS* tiene como parámetros: la información de un nodo, del cual se quiere buscar su sucesor, la lista de sucesores y la información del nodo que tiene que buscar el nodo más cercano al nodo dado. Esta operación da como resultado el nodo más cercano en la lista de sucesores.
- op firstNodeRing : SuccessorList Info-Node -> Info-Node . La operación *firstNodeRing* devuelve la información del nodo cuyo identificador es el menor en el anillo usando la lista de sucesores que le pasan como parámetro de entrada.

- op lastNodeRing : SuccessorList Info-Node -> Info-Node . La operación *lastNodeRing* devuelve la información del nodo cuyo identificador es el mayor en el anillo usando la lista de sucesores que le pasan como parámetro de entrada.
- op buildKT : Info-Node Info-Node KeysTable KeysTable -> KeysTable . La operación *buildKT* obtiene la tabla de claves de un nodo a partir de la tabla de claves de su nodo sucesor. El primer Info-Node es el nodo para el cual se va a construir la tabla de claves y el segundo Info-Node es el nodo sucesor del cual se utiliza su tabla de claves. La primera KeysTable es la tabla de claves del nodo sucesor y la segunda KeysTable es la tabla de claves que estamos generando.

4.3.10 Peer

El módulo PEER es un módulo de sistema en el cual se define la estructura de los nodos.

Los nodos que conforman la red Chord pertenecen a la clase *Peer* la cual hace uso de la clase *Cid* predefinida en Maude:

```
subsort Info-Node < Oid .
op Peer : -> Cid .
```

Cada uno de los nodos tiene una lista de atributos, la cual está formada por una tabla *finger*, una lista de sucesores, una lista de predecesores y una tabla de claves. Las operaciones que definen los atributos son:

```
op FT :_ : FingerTable -> Attribute [ctor  gather(&)] .
op KT :_ : KeysTable -> Attribute [ctor  gather(&)] .
op SL :_ : SuccessorList -> Attribute [ctor  gather(&)] .
op PL :_ : PredecessorList -> Attribute [ctor  gather(&)] .
```

El término que representa un nodo tiene una estructura de la forma:

```
<  N : Peer | FT : ft , SL : sl , PL: pl , KT : kt >
```

donde N es la información del nodo, y ft, sl, pl y kt son los valores actuales de sus atributos.

El módulo PEER es el encargado de definir las operaciones que especifican los mensajes y, las reglas de reescritura. A continuación se presentan las operaciones que definen cada uno de los mensajes:

- op JOIN-NODE : Info-Node Nat -> Msg . La operación *JOIN-NODE* define el mensaje que un nodo genera cuando desea incorporarse a la red. Los parámetros de esta operación hacen referencia al nodo contacto, es decir, al nodo al que se le envía el mensaje y un número natural que indica el identificador del nuevo nodo.
- op FIND-SUCCESSOR : Info-Node Info-Node -> Msg . La operación *FIND-SUCCESSOR* define el mensaje que un nodo envía cuando desea encontrar su sucesor.
- op BUILD-FINGERS : Info-Node Info-Node FingerTable SuccessorList PredecessorList KeysTable -> Msg . La operación *BUILD-FINGERS* define el mensaje que un nodo envía cuando quiere generar su tabla *finger*.
- op GET-KT-SUCC : Info-Node Info-Node -> Msg . La operación *GET-KT-SUCC* define el mensaje que un nodo envía cuando quiere solicitar la tabla de claves de su nodo sucesor.
- op UPDATE-PRED : Info-Node Info-Node -> Msg . La operación *UPDATE-PRED* define el mensaje que un nodo envía a otro nodo para que actualice su predecesor.
- op UPDATE-LIST-SUCC-JOIN : Info-Node Info-Node Info-Node -> Msg . La operación *UPDATE-LIST-SUCC-JOIN* define el mensaje que un nodo, que se ha incorporado al sistema, envía a otro nodo para que actualice su lista de sucesores.
- op UPDATE-LIST-PRED-JOIN : Info-Node Info-Node Info-Node -> Msg . La operación *UPDATE-LIST-PRED-JOIN* define el mensaje que un nodo, que se ha incorporado al sistema, envía a otro nodo para que actualice su lista de predecesores.
- op GET-LIST-SUCC-JOIN : Info-Node Info-Node -> Msg . La operación *GET-LIST-SUCC-JOIN* define el mensaje que un nodo, que se ha incorporado al sistema, envía a su nodo sucesor para que le envíe su lista de sucesores y su lista de predecesores.

- op UPDATE-LIST-JOIN : Info-Node Info-Node SuccessorList PredecessorList -> Msg . La operación *UPDATE-LIST-JOIN* define el mensaje que un nodo envía a otro nodo para que actualice su lista de sucesores y su lista de predecesores.
- op UPDATE-KT : Info-Node Info-Node -> Msg . La operación *UPDATE-KT* define el mensaje que un nodo envía a otro nodo para que actualice su tabla de claves.
- op GET-KT : Info-Node Info-Node -> Msg . La operación *GET-KT* define el mensaje que un nodo envía a otro nodo para que le envíe su tabla de claves.
- op STABILIZE : Info-Node -> Msg . La operación *STABILIZE* define el mensaje que un nodo genera cuando desea iniciar el proceso de estabilización.
- op STABILIZE-GET-PRED : Info-Node Info-Node -> Msg . La operación *STABILIZE-GET-PRED* define el mensaje que un nodo envía a su nodo sucesor para que le envíe su predecesor.
- op STABILIZE-RES : Info-Node Info-Node Info-Node -> Msg . La operación *STABILIZE-RES* define el mensaje que un nodo envía a su nodo sucesor para que compruebe si su sucesor inmediato es correcto. De esta forma se propagará el proceso de estabilización por la red.
- op NOTIFY : Info-Node Info-Node -> Msg . La operación *NOTIFY* define el mensaje que un nodo envía a su nodo sucesor para que compruebe si su predecesor inmediato es válido.
- op LEAVE-GROUP : Info-Node -> Msg . La operación *LEAVE-GROUP* define el mensaje que un nodo genera cuando desea abandonar la red.
- op NOTIFY-LEAVE-PRED : Info-Node Info-Node -> Msg . La operación *NOTIFY-LEAVE-PRED* define el mensaje que un nodo envía a su nodo predecesor para notificarle que abandona la red.
- op NOTIFY-LEAVE-SUCC : Info-Node Info-Node -> Msg . La operación *NOTIFY-LEAVE-SUCC* define el mensaje que un nodo envía a su nodo sucesor para notificarle que abandona la red.
- op UPDATE-LIST-SUCC-LEAVE : Info-Node Info-Node Info-Node -> Msg . La operación *UPDATE-LIST-SUCC-LEAVE* define el mensaje que un nodo, que abandona la red, envía a otro nodo para que actualice su lista de sucesores.

- op UPDATE-LIST-PRED-LEAVE : Info-Node Info-Node Info-Node -> Msg . La operación *UPDATE-LIST-PRED-LEAVE* define el mensaje que un nodo, que abandona la red, envía a otro nodo para que actualice su lista de predecesores.
- op INIT-LOOKFOR : Info-Node BitString -> Msg . La operación *INIT-LOOKFOR* define el mensaje que un nodo genera cuando desea buscar una clave en la red. Los parámetros de esta operación hacen referencia al nodo al que se le envía la petición de búsqueda y una cadena de caracteres que indica el identificador de la clave a buscar.
- op LOOKFOR-KEY : Info-Node Info-Node BitString -> Msg . La operación *LOOKFOR-KEY* define el mensaje que un nodo envía a otro nodo para que éste busque la clave que se le pasa como parámetro.
- op FILE-FOUND : Info-Node Info-Node BitString -> Msg . La operación *FILE-FOUND* define el mensaje que un nodo envía al nodo que solicitó la búsqueda cuando la clave a buscar ha sido encontrada.
- op FILE-NOT-FOUND : Info-Node Info-Node BitString -> Msg . La operación *FILE-NOT-FOUND* define el mensaje que un nodo envía al nodo que solicitó la búsqueda cuando la clave a buscar no está compartida en la red.
- op CHECK-PRED : Info-Node -> Msg . La operación *CHECK-PRED* define el mensaje que un nodo genera cuando desea comprobar si su predecesor inmediato es válido.
- op GET-PRED : Info-Node Info-Node -> Msg . La operación *GET-PRED* define el mensaje que un nodo envía a su nodo predecesor para obtener su nodo sucesor.
- op CHECK-PRED-SUCC : Info-Node Info-Node PredecessorList Info-Node -> Msg . La operación *CHECK-PRED-SUCC* define el mensaje que comprueba si el nodo sucesor del nodo predecesor coincide con el nodo que inicio el proceso de verificación. Si no coincide se actualiza la lista de predecesores del nodo que inicio el proceso. En caso contrario, la lista no se modifica.
- op FIX-FINGERS : Info-Node -> Msg . La operación *FIX-FINGERS* define el mensaje que un nodo genera cuando desea refrescar las entradas de su tabla *finger*.
- op GET-FT-SUCC : Info-Node Info-Node -> Msg . La operación *GET-FT-SUCC* define el mensaje que un nodo envía a su nodo sucesor para que le envíe su tabla *finger*.

- op UPDATE-FINGERS : Info-Node Info-Node FingerTable -> Msg . La operación *UPDATE-FINGERS* define el mensaje que actualiza la tabla *finger* de un nodo usando la tabla *finger* de su nodo sucesor.
- op FIX-FINGERS-NEXT : Info-Node Info-Node -> Msg . La operación *FIX-FINGERS* define el mensaje que un nodo envía a otro nodo para que este actualice su tabla *finger*.
- op INIT-ADD-KEY : Info-Node BitString -> Msg . La operación *INIT-ADD-KEY* define el mensaje que un nodo genera cuando desea subir a la red un fichero. Los parámetros de esta operación hacen referencia al nodo que publica el fichero y una cadena de caracteres que indica el identificador de la clave del fichero que se va a compartir en la red.
- op INIT-DELETE-KEY : Info-Node BitString -> Msg . La operación *INIT-DELETE-KEY* define el mensaje que un nodo genera cuando desea eliminar de la red un fichero. Los parámetros de esta operación hacen referencia al nodo que publicó el fichero y una cadena de caracteres que indica el identificador de la clave del fichero que se va a eliminar de la red.
- op ADD-KEY-NET : Info-Node Info-Node BitString -> Msg . La operación *ADD-KEY-NET* define el mensaje que un nodo envía al nodo responsable de almacenar la clave del fichero para que la añada en su tabla de claves.
- op DELETE-KEY-NET : Info-Node Info-Node BitString -> Msg . La operación *DELETE-KEY-NET* define el mensaje que un nodo envía al nodo responsable de almacenar la clave para que elimine de su tabla de claves la clave del fichero.
- op KEYS-ADD : Info-Node Info-Node KeysTable -> Msg . La operación *KEYS-ADD* define el mensaje que un nodo envía a otro nodo para que éste incorpore en su tabla de claves las entradas de su tabla de claves.
- op KEYS-DELETE : Info-Node KeysTable -> Msg . La operación *KEYS-DELETE* define el mensaje que un nodo envía a otro nodo para que éste elimine de su tabla de claves las entradas de la tabla de claves que le pasan como parámetro.

La sintaxis y explicación de las reglas de reescritura está abordada en la sección 4.3 mediante los procesos básicos entre nodos.

4.3.11 Init

El módulo INIT es un módulo de sistema encargado de definir el estado inicial del anillo denominado *init-state*. Init-state está definido del tipo predefinido *Configuration* y está compuesto por nodos y mensajes que evolucionan por reglas de reescritura que describen el efecto de los eventos de comunicación entre nodos y mensajes.

La red de cinco nodos cuyos identificadores son: 1, 15, 30, 48 y 63, definida en la sección 5.1 se representa mediante el siguiente término inicial:

```
op init-state : -> Configuration .
eq init-state =
 < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : FT15 ,SL : SL15 ,PL : PL15 ,KT :KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : FT30 ,SL : SL30 ,PL : PL30 ,KT :KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : FT48 ,SL : SL48 ,PL : PL48 ,KT :KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : FT63 ,SL : SL63,PL : PL63 ,KT : KT63 >
STABILIZE(N15) .
```

donde la estado inicial de la red simula una situación en la que el nodo con identificador 15 va a iniciar su proceso de estabilización.

4.4 Procesos básicos entre nodos

En esta sección se explica el funcionamiento de cada uno de los procesos posibles en una Tabla Hash Distribuida Chord: incorporar un nodo a la red, eliminar un nodo de la red, estabilización de la red, publicación de un fichero, eliminar la clave de un fichero, búsqueda de una clave o chequear el predecesor.

El comportamiento de cada proceso se especifica mediante las reglas de reescritura definidas en el módulo PEER. Dependiendo de cada proceso será necesario realizar un determinado número de reglas.

4.4.1 Incorporar un nodo a la red

El mensaje JOIN-NODE en el término que representa la red indica que un nodo se incorpora a la red. Cuando un nodo se incorpora a la red debe hacerlo a través de un nodo contacto. En el mensaje se indica el nodo contacto y el identificador del nuevo nodo. El nodo contacto genera la información del nodo nuevo e inicia el proceso de búsqueda del nodo sucesor del nodo nuevo:

```
rl [join-node1] :
  < S : Peer | FT : ft , SL : sl , PL : pl , KT : kt >
    JOIN-NODE(S, i)
=> < S : Peer | FT : ft , SL : sl , PL : pl , KT : kt >
     < createNode(i) : Peer | FT : empty , SL : S , PL : nilPredecessor , KT : empty >
     FIND-SUCCESSOR(S, createNode(i)) .
```

El nodo contacto comprueba si el Node-ID del nodo nuevo está entre su identificador y su sucesor. Si es así, el nodo contacto es su predecesor y el sucesor del nodo contacto es el sucesor del nodo nuevo:

```
crl [findSuccesor1] :
    < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
    FIND-SUCCESSOR(N2, N1)
  => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
    GET-FT-SUCC(headSuccessor(sl2),N1)
    if (getNodeID(N1) > getNodeID(N2) and (getNodeID(N1) <= getNodeID(headSuccessor(sl2))))
        or (getNodeID(headSuccessor(sl2)) < getNodeID(N2) and getNodeID(N1) <= LAST-NODE) .

crl [findSuccesor1] :
    < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
```

```
    FIND-SUCCESSOR(N2, N1)
  => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
    GET-FT-SUCC(N2,N1)
    if (getNodeID(N2) > getNodeID(N1) and getNodeID(N1) < getNodeID(headPredecessor(pl2))) .
```

En caso contrario, el nodo contacto busca en su tabla *finger* el nodo cuyo Node-ID se aproxime más al Node-ID del nodo nuevo y se le envía un mensaje FIND-SUCCESOR del nodo nuevo.

```
crl [findSuccesor2] :
    < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
    FIND-SUCCESSOR(N2, N1)
  => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
    FIND-SUCCESSOR(closestPrecedingNode(N2,N1,ft2,exp(sd(IDBits,1))),N1)
    if not (getNodeID(N1) > getNodeID(N2) and
      getNodeID(N1) <= getNodeID(headSuccessor(sl2))) or
       not (getNodeID(headSuccessor(sl2)) < getNodeID(N2) and
      getNodeID(N1) <= LAST-NODE) or
       not (getNodeID(N2) > getNodeID(N1) and
      getNodeID(N1) < getNodeID(headPredecessor(pl2))) .
```

El mensaje GET-FT-SUCC obtiene la tabla *finger* del nodo sucesor para construir la tabla *finger* del nuevo nodo.

```
rl [getFTSucc] :
  < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
  GET-FT-SUCC(N2,N1)
=> < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
  BUILD-FINGERS(N1, N2, ft2 , sl2, pl2, kt2) .
```

Una vez encontrado el nodo sucesor del nodo nuevo, el nodo nuevo envía un mensaje BUILD-FINGERS a su nodo sucesor con la información necesaria para generar su tabla *finger*, su tabla de claves y su lista de sucesores y predecesores:

```
crl[buildFingers] :
```

```
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    BUILD-FINGERS(N1, N2, ft2, sl2,pl2,kt2)
=> < N1 : Peer | FT : buildTF(N1,sl1,ft1,1),
                 SL : appendSuccessor(N2,appendSuccessor(sl2,nilSuccessor)) ,
                 PL : pl3 , KT : kt3 >
    UPDATE-PRED(N2,N1)
    KEYS-DELETE(N2,kt3)
    STABILIZE(headPredecessor(pl3))
    if kt3 := buildKT(N1,N2,kt2,kt1)
      /\ pl3 := appendPredecessor(pl2,appendPredecessor(N2,nilPredecessor)) .
```

A continuación, el mensaje UPDATE-PRED actualiza la lista de sucesores y predecesores del nodo sucesor del nuevo nodo, incorporando a este.

```
rl [updatePred] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     UPDATE-PRED(N1,N2)
  => < N1 : Peer | FT : ft1 , SL : appendSuccessor(sl1,appendSuccessor(N2,nilSuccessor)) ,
               PL : appendPredecessor(N2,pl1) , KT : kt1 >
```

Más adelante, el mensaje KEYS-DELETE elimina de la tabla de claves del nodo sucesor aquellas claves que ahora le pertenecen al nodo nuevo.

```
rl [keysDelete1] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     KEYS-DELETE(N1,kt2)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : deleteKT(kt2,kt1) > .
```

Por último, el nodo predecesor inicia el proceso de estabilización para que todos los nodos de la red sean notificados de la incorporación del nodo nuevo y actualicen sus tablas y listas garantizando que la información que contienen es válida.

4.4.2 Eliminar un nodo de la red

El mensaje LEAVE-GROUP en el término que representa la red indica que un nodo abandona la red. Nótese que este mensaje se refiere a cuando un nodo abandona la red de forma ordenada y no a cuando un nodo falla.

```
rl [leaveGroup1] :
   < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
   LEAVE-GROUP(N2)
=> KEYS-ADD(headSuccessor(sl2), N2, kt2)
     NOTIFY-LEAVE-PRED(headPredecessor(pl2),N2)
     NOTIFY-LEAVE-SUCC(headSuccessor(sl2),N2)
     FIX-FINGERS(headPredecessor(pl2))
     FIX-FINGERS-NEXT(headSuccessor(sl2),headSuccessor(sl2)) .
```

El mensaje KEYS-ADD, envía su tabla de claves a su nodo sucesor para que este las añada a su tabla de claves.

```
rl [keys1] :
      < N2 : Peer | FT : ft , SL : sl , PL : pl , KT : kt >
      KEYS-ADD(N2,N1,kt2)
  => < N2 : Peer | FT : ft , SL : sl , PL : pl , KT : appendKT(kt2, kt) > .
```

El siguiente paso es notificar a su nodo sucesor y a su nodo predecesor que abandona la red con el fin de que estos eliminen al nodo que abandona de su lista de sucesores y predecesores respectivamente. Para ello, envía los mensajes NOTIFY-LEAVE-SUCC y NOTIFY-LEAVE-PRED.

```
rl [NotifyLeavePred1] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     NOTIFY-LEAVE-PRED(N1,N2)
  => < N1 : Peer | FT : ft1 , SL : deleteListSucc(N2,sl1) , PL : pl1 , KT : kt1 >
     UPDATE-LIST-SUCC-LEAVE(headSuccessor(deleteListSucc(N2,sl1)),N1,N2) .
```

```
rl [NotifyLeaveSucc1] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    NOTIFY-LEAVE-SUCC(N1,N2)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : deleteListPred(N2,pl1) , KT : kt1 >
    UPDATE-LIST-PRED-LEAVE(headPredecessor(deleteListPred(N2,pl1)),N1,N2) .

crl [updateListSuccLeave] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     UPDATE-LIST-SUCC-LEAVE(N1,N3,N2)
  => < N1 : Peer | FT : ft1 , SL : deleteListSucc(N2,sl1) , PL : pl1 , KT : kt1 >
     UPDATE-LIST-SUCC-LEAVE(headSuccessor(sl1),N3,N2)
     if (headSuccessor(sl1) =/= N3 ) .

  crl [updateListSuccLeave2] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     UPDATE-LIST-SUCC-LEAVE(N1,N3,N2)
  => < N1 : Peer | FT : ft1 , SL : deleteListSucc(N2,sl1) , PL : pl1 , KT : kt1 >
     if (headSuccessor(sl1) == N3 ) .

  crl [updateListPredLeave] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     UPDATE-LIST-PRED-LEAVE(N1,N3,N2)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : deleteListPred(N2,pl1) , KT : kt1 >
     UPDATE-LIST-PRED-LEAVE(headPredecessor(pl1),N3,N2)
     if (headPredecessor(pl1) =/= N3 ) .

  crl [updateListPred2Leave] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     UPDATE-LIST-PRED-LEAVE(N1,N3,N2)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : deleteListPred(N2,pl1) , KT : kt1 >
     if (headPredecessor(pl1) == N3 ) .
```

Por último, hay que actualizar las tablas *finger* de los nodos de la red. Para ello, envía un mensaje FIX-FINGERS al nodo predecesor, para que actualice las entradas de su tabla *finger* y un mensaje FIX-FINGERS-NEXT, para que el resto de nodos que conforman la red también actualicen sus tablas *finger*:

```
rl [fixFingers] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     FIX-FINGERS(N1)
   => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     GET-FT-SUCC(headSuccessor(sl1),N1) .

rl [getFTSucc] :
   < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
     GET-FT-SUCC(N2,N1)
  => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
     UPDATE-FINGERS(N1,N2,ft2) .

rl [updateFingers] :
   < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    UPDATE-FINGERS(N1,N2,ft2)
    => < N1 : Peer | FT : buildTF(N1,sl1,ft1,1) , SL : sl1 , PL : pl1 , KT : kt1 > .

crl [fixFingersNext] :
     < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
      FIX-FINGERS-NEXT(N2,N1)
    => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
      FIX-FINGERS(N2)
      FIX-FINGERS-NEXT(headPredecessor(pl2),N1)
      if (headPredecessor(pl2) =/= N1) .

crl [fixFingersNext2] :
    < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
     FIX-FINGERS-NEXT(N2,N1)
   => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
     FIX-FINGERS(N2)
     if (headPredecessor(pl2) == N1) .
```

Nótese que las claves de los ficheros que publicó cuando estaba activo permanecerán almacenadas en las tablas de claves de sus nodos responsables.

4.4.3 Proceso de estabilización

Un nodo debe actualizar su lista de atributos cada cierto tiempo. Como nuestra especificación no tiene tiempo simulamos el comienzo del proceso en la red con el mensaje STABILIZE. La finalidad de ejecutar este proceso es verificar que el nodo sucesor inmediato que tiene es válido. A continuación, se solicita a su nodo sucesor su predecesor mediante el mensaje STABILIZE-GET-PRED:

```
rl [stabilize1] :
  < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  STABILIZE(N1)
 => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  STABILIZE-GET-PRED(headSuccessor(sl1), N1) .
```

El sucesor le envía su predecesor mediante el mensaje STABILIZE-RES:

```
rl [stabilizeGetPred1] :
   < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
   STABILIZE-GET-PRED(N2,N1)
=> < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
   STABILIZE-RES(N1, N2, headPredecessor(pl2)) .
```

El nodo que inició el proceso de estabilización comprueba si el nodo sucesor que tiene actualmente no coincide con el nodo predecesor devuelto por su sucesor. Esta situación se dará cuando el nodo predecesor devuelto por su sucesor se ha unido recientemente al sistema. Si esta situación se produce, es decir, el nodo sucesor inmediato no coincide con el nodo predecesor de su sucesor, el nodo que inicio la estabilización actualizará su lista de sucesores y predecesores añadiendo el nodo predecesor devuelto por su nodo sucesor. Además, con el mensaje NOTIFY se comprueba que el nodo nuevo tiene un predecesor válido. Por último, se envía un mensaje FIX-FINGER y FIX-FINGER-NEXT para actualizar las tablas *finger* del resto de los nodos que

conforman la red, y un mensaje UPDATE-LIST-SUCC-JOIN para incorporar al nuevo nodo en las listas de sucesores y predecesores de algunos de los otros nodos de la red.

```
crl [stabilizeRes1] :
  < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  STABILIZE-RES(N1, N2, PRED)
=> < N1 : Peer | FT : ft1 ,
                 SL : appendSuccessor(PRED,sl1) ,
                 PL : appendPredecessor(pl1,appendPredecessor(PRED,nilPredecessor)) ,
                 KT : kt1 >
   NOTIFY(PRED,N1)
   FIX-FINGERS(N1)
   FIX-FINGERS-NEXT(headPredecessor(pl1),N1)
   UPDATE-LIST-SUCC-JOIN(headPredecessor(pl1),PRED,PRED)
   if getNodeID(PRED) > getNodeID(N1) and
      getNodeID(PRED) > getNodeID(headSuccessor(sl1)) or
      (getNodeID(N1) <= LAST-NODE and getNodeID(N2) > getNodeID(PRED)) .
```

En caso de que su nodo sucesor coincide con el nodo predecesor devuelto por su sucesor, significa que tiene un sucesor inmediato válido y le enviará un mensaje NOTIFY para que verifique si su nodo predecesor es válido.

```
crl [stabilizeRes2] :
  < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  STABILIZE-RES(N1,N2,PRED)
=> < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  NOTIFY(headSuccessor(sl1),N1)
  FIX-FINGERS(N1)
  if not (getNodeID(PRED) > getNodeID(N1) and
     getNodeID(PRED) > getNodeID(headSuccessor(sl1))) or
     not (getNodeID(N1) <= LAST-NODE and  getNodeID(N2) > getNodeID(PRED)) .


crl [Notify1] :
     < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
     NOTIFY(N2,N1)
```

```
  => < N2 : Peer | FT : ft2 , SL : sl2 , PL : appendPredecessor(N1,pl2) , KT : kt2 >
     if getNodeID(N1) > getNodeID(headPredecessor(pl2)) and getNodeID(N1) < getNodeID(N2) .

crl [Notify2] :
     < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
     NOTIFY(N2,N1)
  => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
     if not (getNodeID(N1) > getNodeID(headPredecessor(pl2)) and getNodeID(N1) < getNodeID(N2))
     and (pl2 =/= nilPredecessor) .

 rl [Notify3] :
     < N2 : Peer | FT : ft2 , SL : sl2 , PL : nilPredecessor , KT : kt2 >
     NOTIFY(N2,N1)
  => < N2 : Peer | FT : ft2 , SL : sl2 , PL : appendPredecessor(N1,nilPredecessor) , KT : kt2 > .
```

4.4.4 Publicar un fichero en la red

El mensaje INIT-ADD-KEY en el término que representa la red indica que un nodo desea publicar un fichero en la red. En el mensaje se indica el identificador del nodo que publica el fichero y la clave del fichero. En primer lugar, busca el nodo que debe almacenar la clave del fichero. Una vez localizado el nodo, se le envía un mensaje ADD-KEY-NET indicándole el Info-Node del nodo que publica el fichero y la clave del fichero para que este lo inserte en su tabla de claves .

```
rl [init-addKey] :
  < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  INIT-ADD-KEY(N1,K)
=> < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  ADD-KEY-NET(closestNodeKey(N1,ft1,K),N1,K) .

crl [addKeyNode1] :
  < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  ADD-KEY-NET(N1,N2,K)
=> < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : insertKT(K,N2,kt1) >
```

```
  if K <= getNodeID(N1) and K > getNodeID(headPredecessor(pl1)) or
     getNodeID(N1) < getNodeID(headPredecessor(pl1)) and
     K > getNodeID(headPredecessor(pl1)) or
    getNodeID(N1) < getNodeID(headPredecessor(pl1)) and
    K <= getNodeID(N1) .
```

Debemos tener en cuenta que estamos considerando que cada identificador de fichero lo guarda un solo nodo, lo cual permite que la operación *insertKT* funcione correctamente. Sin embargo, en una implementación real, un fichero puede ser almacenado por varios nodos para evitar pérdida de información en caso de fallo de los nodos. En este trabajo nos restringimos a la especificación del protocolo según indica el artículo [34].

4.4.5 Eliminar la clave de un fichero

El mensaje INIT-DELETE-KEY en el término que representa la red indica que un nodo desea eliminar un fichero de la red. En el mensaje se indica el identificador del nodo que desea eliminar el fichero y la clave del fichero. En primer lugar, comprueba si la clave está dentro de su tabla de claves. Si es así, la elimina y finaliza el proceso:

```
crl [init-deleteKey2] :
  < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  INIT-DELETE-KEY(N2,K)
=> < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : deleteKT(K |-> N2,kt1) >
   if $hasMapping(kt1, K) .
```

En caso contrario, busca en su tabla *finger* el nodo que contiene la clave o aquel cuyo identificador se aproxime más a la clave. Si el nodo responsable de la clave no se encuentra en su tabla *finger* se sigue buscando en los otros nodos más cercanos a la clave. Una vez localizado el nodo que contiene la clave, se le envía el mensaje DELETE-KEY-NET para que la elimine de su tabla de claves.

```
crl [init-deleteKey1] :
```

```
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    INIT-DELETE-KEY(N2,K)
 => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    DELETE-KEY-NET(closestNodeKey(N1,ft1,K),N2,K)
    if not $hasMapping(kt1, K) .

crl [init-deleteKey2] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    INIT-DELETE-KEY(N2,K)
 => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : deleteKT(K |-> N2,kt1) >
    if $hasMapping(kt1, K) .

crl [deleteKeyNode1] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     DELETE-KEY-NET(N1,N2,K)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : deleteKT(K |-> N2,kt1) >
    if K <= getNodeID(N1) and K > getNodeID(headPredecessor(pl1))
      or getNodeID(N1) < getNodeID(headPredecessor(pl1)) and K > getNodeID(headPredecessor(pl1))
      or getNodeID(N1) < getNodeID(headPredecessor(pl1)) and K <= getNodeID(N1) .

crl [deleteKeyNode2] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     DELETE-KEY-NET(N1,N2,K)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     DELETE-KEY-NET(headSuccessor(sl1),N2,K)
     if not (K <= getNodeID(N1) and K > getNodeID(headPredecessor(pl1)))
       or not (getNodeID(N1) < getNodeID(headPredecessor(pl1))
       and K > getNodeID(headPredecessor(pl1)))
       or not (getNodeID(N1) < getNodeID(headPredecessor(pl1)) and K <= getNodeID(N1)) .
```

Como las claves se publican en un único nodo no es necesario realizar ninguna otra acción.

4.4.6 Búsqueda de una clave

El mensaje INIT-LOOKFOR en el término que representa la red indica que un nodo desea buscar un fichero en la red. En el mensaje se indica el identificador del nodo que desea buscar el fichero y la clave del fichero a buscar. Este proceso consiste en buscar al nodo responsable de almacenar la clave. Una vez encontrado el nodo, si la clave se encuentra en su tabla de claves, se envía un mensaje de éxito al nodo que inicio la búsqueda. En caso contrario, se envía un mensaje que indica que la clave no se encuentra compartida en la red.

En primer lugar, el nodo comprueba si la clave está dentro de su tabla de claves. Si es así, él es el nodo responsable de almacenarla y envía un mensaje FILE-FOUND indicando que el fichero ha sido encontrado con el nodo que lo ha publicado, y se da por finalizado el proceso de búsqueda:

```
crl [init-lookfor2] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    INIT-LOOKFOR(N1, K)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    FILE-FOUND(N1, N1, K)
    if $hasMapping(kt1, K) .
```

En caso contrario, se procede a buscar el nodo responsable de la clave del fichero. El nodo busca en su tabla *finger* el nodo cuyo identificador se aproxime más a la clave de búsqueda. Una vez encontrado, se le envía un mensaje LOOKFOR-KEY para que continúe el proceso de búsqueda.

```
crl [init-lookfor1] :
  < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  INIT-LOOKFOR(N1, K)
=> < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
  LOOKFOR-KEY(closestNodeKey(N1,ft1,K), N1, K)
  if not $hasMapping(kt1, K) .
```

En primer lugar, comprueba si la clave de búsqueda está en su tabla de claves. Si es así, envía un mensaje FILE-FOUND al nodo que inicio la búsqueda indicando que la clave ha sido encontrada.

```
crl [lookfor-Key1] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    LOOKFOR-KEY(N1, N2, K)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    FILE-FOUND(N1, N2, K)
    if $hasMapping(kt1, K) .
```

En caso contrario, buscará en su tabla *finger* el nodo cuyo identificador se aproxime más a la clave de búsqueda y se le enviará el mensaje LOOKFOR-KEY con la clave a buscar.

```
crl [lookfor-Key3] :
     < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     LOOKFOR-KEY(N1,N2,K)
    => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     LOOKFOR-KEY(closestNodeKey(N1,ft1,K),N2,K)
     if not $hasMapping(kt1, K) and
       not (K <= getNodeID(N1) and K > getNodeID(headPredecessor(pl1))) .
```

Este proceso termina cuando se encuentra el nodo responsable de almacenar la clave de búsqueda. Si este contiene la clave en su tabla de claves se envía el mensaje FIND-FOUND al nodo que inicio la búsqueda dando por finalizada con éxito el proceso de búsqueda. En caso contrario, envía un mensaje FILE-NOT-FOUND al nodo que inicio la búsqueda indicando que el archivo no está almacenado en la red:

```
crl [lookfor-Key1] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    LOOKFOR-KEY(N1, N2, K)
  => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    FILE-FOUND(N2, N1, K)
    if $hasMapping(kt1, K) .
```

```
crl [lookfor-Key2] :
      < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
      LOOKFOR-KEY(N1, N2, K)
   => < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
      FILE-NOT-FOUND(N2, N1, K)
      if not $hasMapping(kt1, K) and (K <= getNodeID(N1) and K > getNodeID(headPredecessor(pl1))) .
```

Puede ocurrir que el nodo realice una búsqueda de una clave inmediatamente después de que el nodo responsable de almacenarla ha abandonado la red. Por lo que en ese momento la clave del fichero está almacenada en su nodo sucesor. Sin embargo, el nodo sigue teniendo referencia al nodo que ha abandonado la red, por lo que la búsqueda se realiza sobre dicho nodo, el cual ya no es el responsable de la clave, por lo que se recibe un mensaje NOT-FILE-FOUND. Sin embargo, la clave si está en la red pero almacenada por su nodo sucesor. Esto no supone un problema, ya que el proceso de estabilización se ejecuta periódicamente en cada nodo. Por lo que el nodo actualizará su lista de atributos, eliminando al nodo que abandonó la red, y cuando decida realizar de nuevo la búsqueda de la clave recibirá un mensaje FILE-FOUND.

4.4.7 Nodo comprueba su predecesor

Un nodo comprueba cada cierto tiempo si su nodo predecesor inmediato es válido. Como nuestra especificación no tiene tiempo simulamos el comienzo del proceso en la red con el mensaje CHECK-PRED. En el mensaje se indica el identificador del nodo que inicia el proceso.

En primer lugar, a través del mensaje GET-PRED, el nodo que inició el proceso solicita a su nodo predecesor su sucesor.

```
rl [checkPredecessor1] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    CHECK-PRED(N1)
=> < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     GET-PRED(headPredecessor(pl1),N1) .
```

```
rl [GetPred] :
   < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
   GET-PRED(N2,N1)
 => < N2 : Peer | FT : ft2 , SL : sl2 , PL : pl2 , KT : kt2 >
    CHECK-PRED-SUCC(N1, N2, pl2, headSuccessor(sl2)) .
```

Conocido el nodo sucesor, el mensaje CHECK-PRED-SUCC comprueba si el nodo sucesor coincide con el nodo que inicio el proceso. Si son iguales, significa que el nodo que inicio el proceso tiene un predecesor inmediato válido. En caso contrario, se actualiza su lista de predecesores.

```
crl [CheckPredSucc1] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
    CHECK-PRED-SUCC(N1, N2, pl2, SUCC)
=> < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     if (getNodeID(SUCC) == getNodeID(N1)) .

crl [CheckPredSucc1] :
    < N1 : Peer | FT : ft1 , SL : sl1 , PL : pl1 , KT : kt1 >
     CHECK-PRED-SUCC(N1, N2, pl2, SUCC)
  => < N1 : Peer | FT : ft1 , SL : sl1 ,
 PL : appendPredecessor(SUCC,appendPredecessor(N2,deleteListPred(SUCC,deleteListPred(N1,pl2)))),
  KT : kt1 >
        if not (getNodeID(SUCC) == getNodeID(N1)) .
```

Capítulo 5 - Análisis del protocolo

Una vez descrito el funcionamiento teórico de la Tabla Hash Distribuida Chord presentamos los resultados que proporciona la especificación sobre una red de 5 nodos en los diferentes procesos. El capítulo está dividido en dos partes. En la primera parte se define la configuración inicial de la red sobre el cual se han ejecutado las pruebas y en la segunda parte se presentan los resultados obtenidos al realizar cada uno de los procesos.

5.1 Configuración inicial

Para ilustrar la ejecución del protocolo necesitamos comenzar con una red concreta. Para ello, se ha elaborado un anillo Chord de 8 bits cuyo estado inicial está formado por 5 nodos cuyos identificadores son: 1, 15, 30, 48 y 63. En este ejemplo se utilizan solo 8 bits en los identificadores de los ficheros y de los nodos, en lugar de los 128 que utiliza el protocolo real, y un anillo con pocos nodos para poder mostrar con más facilidad los resultados de los procesos llevados a cabo. El apéndice B muestra la red de un anillo Chord de 8 bits formado por 15 nodos activos.

En la tabla 5.1 se muestran los ficheros que comparte cada uno de los nodos que forma la red.

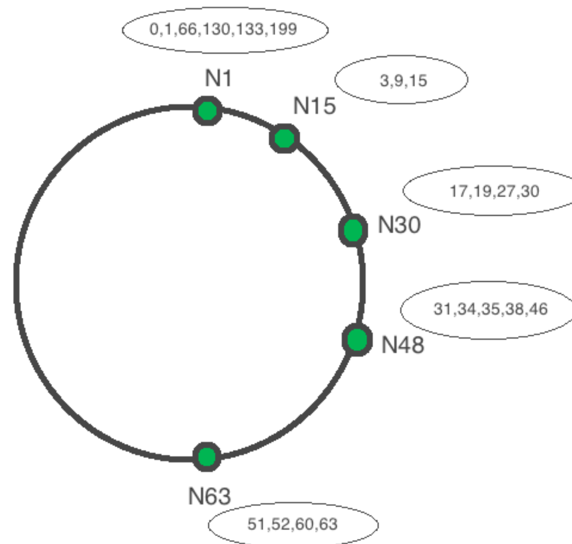
Tabla 5.1 Ficheros que comparte cada uno de los 5 nodos del ejemplo

Identificador del nodo	Identificador del fichero
1	3, 17, 51, 52
15	19, 27, 30, 31, 66, 130
30	199
48	0, 15, 38, 46, 60, 133
63	1, 9, 34, 35, 63

Nótese que el nodo que publica el fichero no tiene porqué ser el nodo encargado de almacenarlo en la red. Una clave *k* es asignada al primer nodo activo cuyo identificador es igual o es el siguiente a *k* en el espacio de identificadores. Por ejemplo, el nodo con identificador 1 publica la clave 3. El nodo responsable de almacenar la clave es el nodo con identificador 15.

La figura 5.1 muestra gráficamente el anillo con los nodos que lo conforman y las claves de los ficheros de las que son responsables cada uno de ellos.

Figura 5.1 Representación del estado inicial del anillo



El estado inicial del anillo y la longitud de bits de los identificadores se especifica en Maude como sigue:

```
eq IDBits = 8 .

eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : FT15 , SL : SL15 , PL : PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : FT30 , SL : SL30 , PL : PL30 , KT : KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : FT48 , SL : SL48 , PL : PL48 , KT : KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : FT63 , SL : SL63, PL : PL63, KT : KT63 > .
```

donde el identificador del objeto es una tupla cuya primera componente es el valor del identificador del nodo representado en 8 bits y la segunda componente es la dirección IP, que para facilitar la legibilidad de los términos en Maude denotamos con el valor en decimal del identificador del nodo.

Cada uno de los objetos tiene una lista de atributos: tabla *finger* (FT), tabla de claves (KT), lista de sucesores (SL) y lista de predecesores (PL).

La tabla *finger* de un nodo con identificador n (FTn) tiene 8 entradas, donde 8 es la longitud de bits de los identificadores del anillo. Para cada entrada 2^i de la tabla, donde $0 <= i <= 7$, se almacena el identificador del nodo sucesor de ($n + 2^i$) y su dirección IP. Nótese que la primera entrada en la tabla es el sucesor del nodo n en el anillo.

La tabla de claves de un nodo con identificador n (KTn) está formada por los identificadores de las claves de los ficheros de los que el nodo n es el responsable. Cada una de las entradas de la tabla es una tupla cuya primera componente es el valor del identificador de la clave representado en 8 bits y la segunda componente es identificador del nodo que comparte el fichero en la red.

La lista de sucesores de un nodo con identificador n (SLn) está formada por los identificadores de sus nodos sucesores en el anillo. No es necesario almacenar todos los nodos sucesores, sino sólo aquellos que sean necesarios para garantizar que no se pierden datos durante la transitoriedad de los nodos en la red. En este ejemplo, la lista de sucesores almacena todos los nodos sucesores del nodo con identificador n debido a que la red está formada por muy pocos nodos.

La lista de predecesores de un nodo con identificador n (PLn) está formada por los identificadores de sus nodos predecesores en el anillo. Al igual que en la lista de sucesores, no es necesario almacenar todos los nodos predecesores del nodo con identificador n.

A continuación se muestran los valores de la lista de atributos de cada uno de los 5 nodos que forman la red.

Nodo 1:

- Tabla *finger*

```
eq FT1 = 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
        16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
        32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
        64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
       128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" .
```

- Tabla de claves:

```
eq KT1 = 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
         0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" .
```

- Lista de sucesores:

```
eq SL1 = N15 & N30 & N48 & N63 .
```

- Lista de predecesores:

```
eq PL1 = N63 || N48 || N30 || N15 .
```

Nodo 15:

- Tabla *finger*:

```
eq FT15 = 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
          2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
          4 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
          8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
         16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
```

```
32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" .
```

- Tabla de claves:

```
eq KT15 = 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
          0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" .
```

- Lista de sucesores:

```
eq SL15 = N30 & N48 & N63 & N1 .
```

- Lista de predecesores:

```
eq PL15 = N1 || N63 || N48 || N30 .
```

Nodo 30:

- Tabla *finger*:

```
eq FT30 = 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
          2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
          4 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
          8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
         64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
        128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" .
```

- Tabla de claves:

```
eq KT30 = 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15",
          0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15",
          0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" .
```

- Lista de sucesores:

```
eq SL30 = N48 & N63 & N1 & N15 .
```

- Lista de predecesores:

```
eq PL30 = N15 || N1 || N63 || N48 .
```

Nodo 48:

- Tabla *finger*:

```
eq FT48 = 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
          2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
          4 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
          8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
         16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
         32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
         64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
        128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" .
```

- Tabla de claves:

```
eq KT48 = 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
          0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
          0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48" ,
          0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48" ,
          0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" .
```

- Lista de sucesores:

```
eq SL48 = N63 & N1 & N15 & N30 .
```

- Lista de predecesores:

```
eq PL48 = N30 || N15 || N1 || N63 .
```

Nodo 63:

- Tabla *finger*:

```
eq FT63 = 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          4 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
         16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
         32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
         64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
```

```
128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" .
```

- Tabla de claves:

```
eq KT63 = 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48" ,
          0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" .
```

- Lista de sucesores:

```
eq SL63 = N1 & N15 & N30 & N48 .
```

- Lista de predecesores:

```
eq PL63 = N48 || N30 || N15 || N1 .
```

5.2 Resultado de los procesos

Una vez definido el estado inicial, ejecutamos cada uno de los procesos sobre este explicando mediante pantallas y aclaraciones el significado de cada uno de los resultados obtenidos.

Cada uno de los procesos se ejecutan debido a que ocurre algún evento en la red, tales como: un nodo se incorpora a la red, inicio del proceso de estabilización, un nodo decide abandonar la red o que un nodo inicie la búsqueda de la clave de un fichero en la red. Dado que la especificación realizada no incluye un concepto de tiempo, simulamos el comienzo de un proceso usando un mensaje en el término que represente el estado sobre el que queremos comprobar el resultado del proceso.

En primer lugar, iniciamos Maude con el siguiente comando:

./maude/maude.intelDarwin

cuyo resultado se muestra en la figura 5.2.

Figura 5.2 Comando para ejecutar Maude

```
Maude2 — maude.intelDarwi — 80×12
Last login: Sun Aug 12 21:31:53 on ttys000
MacBook-Pro-de-Sara:~ saramanchado$ cd Desktop
MacBook-Pro-de-Sara:Desktop saramanchado$ cd Maude2
MacBook-Pro-de-Sara:Maude2 saramanchado$ ./maude/maude.intelDarwin
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.6 built: Dec 10 2010 11:12:39
            Copyright 1997-2010 SRI International
                   Sun Aug 12 21:32:37 2012
Maude>
```

A continuación, cargamos el archivo donde se encuentra almacenada la especificación de la Tabla Hash Distribuida Chord con el siguiente comando:

load test.maude .

Figura 5.3 Comando que carga la especificación de la Tabla Hash Distribuida Chord

```
Maude2 — maude.intelDarwi — 80×18
Maude> load test.maude .
Advisory: redefining module MAP+.
Advisory: redefining module LIST+.
Advisory: redefining module BITSTRING.
Advisory: redefining module NODE-ID.
Advisory: redefining module NODE-IP.
Advisory: redefining module INFO-NODE.
Advisory: redefining view vBitString.
Advisory: redefining view vInfo-Node.
Advisory: redefining module KEYSTABLE.
Advisory: redefining module FINGERTABLE.
Advisory: redefining module SUCCESSORLIST.
Advisory: redefining module PREDECESSORLIST.
Advisory: redefining module RPCS.
Advisory: redefining module CONFIGURATION.
Advisory: redefining module PEER.
Advisory: redefining module INIT.
Maude>
```

Una vez tenemos cargado el archivo de la especificación en Maude, reescribimos la expresión del estado inicial (*init-state*) utilizando el comando *rewrite* (*rew* de forma abreviada).

Debido a la posibilidad de no terminación, este comando admite un argumento adicional que limita el número de aplicaciones de reglas. Así pues, la reescritura quedaría de la siguiente forma: *rew[1] init-state .*

5.2.1 Incorporar un nodo a la red

Para que un nodo pueda conectarse a la red debe conocer el identificador de un nodo activo en la red, llamado nodo contacto.

Para simular el hecho de que un nodo está conectándose a la red insertamos el mensaje ficticio JOIN-NODE(*InfoNode,nat*), donde el parámetro *InfoNode* indica el nodo contacto de la red y el parámetro *nat* el valor en decimal del identificador del nuevo nodo. Por ejemplo, supongamos que el nodo con identificador 20 (N20) se incorpora a la red definida en el apartado 5.1 contactando con el nodo con identificador 1 (N1). El término que define el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
JOIN-NODE(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" , 20) .
```

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[46] init-state .
```

A continuación se explica cómo se va reescribiendo el estado:

1) El nodo contacto (N1) comprueba si él es el nodo sucesor del nodo que se va a incorporar (N20) o si su nodo sucesor (N15) es el sucesor del nodo que se va a incorporar. En este caso, no se cumple ninguna de las condiciones por lo que el nodo contacto (N1) envía un

mensaje FIND-SUCCESSOR(0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") al nodo más cercano (N15) con el nodo que se va incorporar (N20) en su tabla *finger*.

2) A continuación, se comprueba si el nodo 15 es el sucesor del nodo 20 o si el sucesor del nodo 15 (N30) es el sucesor del nodo 20. Como el nodo 30 es el sucesor del nodo 20, el nodo 15 le envía un mensaje GET-FT-SUCC(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") al nodo 30 para que envíe al nodo 20 su tabla *finger*.

3) El nodo 30 le envía al nodo 20 su tabla *finger* y su tabla de claves mediante el mensaje BUILD-FINGERS(0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"), (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15")).

4) El nodo 20 construye su tabla *finger* y su tabla de claves con la información que le ha enviado el nodo 30 y envía un mensaje UPDATE-PRED(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") al nodo 30 para que actualice su lista de predecesores. Al mismo tiempo envía un mensaje KEYS-DELETE(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15")) al nodo 30 para que elimine la claves que ya no le corresponden e inicia el proceso de estabilización de su predecesor (N15) para actualizar sus tablas, STABILIZE(0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") .

5) Cuando el nodo 15 finaliza el proceso de estabilización, envía un mensaje a su sucesor para estabilizar el resto de la red. Se observa que en total se realizan 46 pasos para dar por finalizado el proceso.

La figura 5.4 muestra el resultado de insertar el nodo con identificador 20 a la red. Como se puede observar, el nodo con identificador 20 se ha insertado correctamente y el resto de nodos han actualizado sus tablas *finger*, sus listas de predecesores y sus listas de sucesores incorporando al nodo con identificador 20. Las líneas verdes señalan la incorporación del nodo con identificador 20 y las líneas rojas indican las actualizaciones en las tablas *finger*, las listas de predecesores y las listas de sucesores de los otros nodos.

5.2.2 Eliminar un nodo de la red

Para eliminar un nodo de la red insertamos el mensaje ficticio LEAVE-GROUP(*InfoNode*) que simula el hecho de que un nodo abandona la red. El parámetro *InfoNode* indica el nodo que abandona la red. Por ejemplo, supongamos que el nodo con identificador 15 desea abandonar la red. N15 es la constante definida para el nodo con identificador 15. El término que define el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
LEAVE-GROUP(N15) .
```

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[25] init-state .
```

Figura 5.4 Resultado de insertar el nodo con identificador 20 a la red Chord

A continuación se explica cómo se va reescribiendo el estado:

El nodo que abandona la red (N15) envía un mensaje NOTIFY-LEAVE-PRED(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") a su nodo predecesor (N1) para que actualice su lista de sucesores. Al mismo tiempo envía un mensaje NOTIFY-LEAVE-SUCC(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") a su nodo sucesor (N30) para que actualice su lista de predecesores; un mensaje KEYS-ADD(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48")) para que el nodo sucesor (N30) incorpore a su tabla de claves las claves del nodo 15; y envía un mensaje FIX-FINGERS(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") a su nodo predecesor para que actualice su tabla *finger*. Por último, envía un mensaje FIX-FINGERS-NEXT(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") a su nodo sucesor (N30) para que actualice su tabla *finger* y se propague el proceso por toda la red. Se observa que en total se realizan 25 pasos para dar por finalizado el proceso.

La figura 5.5 muestra cómo queda el estado final del resto de nodos de la red tras el abandono del nodo con identificador 15 de la red. Como podemos observar, el nodo con identificador 15 ya no se encuentra entre los nodos que forman parte de la red. Además, el resto de nodos han eliminado las referencias que tenían del nodo con identificador 15 en sus tablas *finger*, sus listas de sucesores y sus lista de predecesores. La línea roja indica la posición donde estaba el nodo con identificador 15. Las líneas verdes indican el nuevo sucesor inmediato del nodo con identificador 1 y el nuevo predecesor inmediato del nodo con identificador 30. Las líneas azules indican las tablas *finger*, listas de sucesores y listas de predecesores de los otros nodos actualizadas.

Figura 5.5 Resultado de eliminar el nodo con identificador 15 de la red Chord

```
Maude2 — maude.intelDarwi — 180x34
rewrites: 260841 in 168ms cpu (169ms real) (1543638 rewrites/second)
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @
    "127.0.0.30", 4 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ;
    1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0
    ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1
    @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") >
Maude>
```

5.2.3 Estabilización de la red

El proceso de estabilización se ejecuta cada cierto periodo de tiempo por los nodos que forman la red para mantener actualizada la información de sus listas de sucesores y predecesores y de sus tablas *finger*. Además, es necesario iniciar este proceso después de que un nodo se haya incorporado o haya abandonado la red.

Para simular el hecho de que un nodo inicia el proceso de estabilización insertamos el mensaje ficticio STABILIZE(*InfoNode*), donde el parámetro *InfoNode* indica el nodo que desea iniciar el proceso.

La figura 5.6 muestra el estado de la red. Como se puede observar marcado en verde, el nodo con identificador 15 tiene como sucesor inmediato el nodo 30. Sin embargo, marcado en rojo, podemos observar que el nodo con identificador 20 está activo y este es su sucesor inmediato válido según está la configuración actual de la red. Seguramente este nodo se habrá incorporado recientemente al sistema.

Supongamos que ha pasado un tiempo *x* y el nodo con identificador 15 inicia el proceso de estabilización.

El término que define el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT  : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.20" : Peer | FT: FT20 , SL: SL20 , PL: PL20 , KT: KT20 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
STABILIZE(N15) .
```

Las variables del término representan ahora la configuración de la figura 5.6.

Figura 5.6 Estado de la red antes de ejecutar el proceso de estabilización por el nodo con identificador 15

```
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 0 ; 1
    ; 0 ; 0 @ "127.0.0.20") || 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0
    ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20"),
  PL : ((0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0
    ; 0 @ "127.0.0.20") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") || (0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") >
Maude>
```

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[35] init-state .
```

A continuación se explica como se va reescribiendo el estado:

1) El nodo 15 inicia el proceso de estabilización enviando un mensaje STABILIZE-GET-PRED(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") a su nodo sucesor (N30) para que le indique su nodo predecesor.
2) El nodo 30 contesta al nodo 15 que su predecesor es el nodo 20 con el mensaje STABILIZE-RES(0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20"). Como no coincide el sucesor del nodo 15 (N30) con el nodo 20, entonces el nodo 15 manda un mensaje NOTIFY(0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") al nodo 20 para indicarle que es su predecesor. Al mismo tiempo con el mensaje FIX-FINGERS(0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") modifica su tabla *finger* y manda un mensaje FIX-FINGERS-NEXT(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") a su predecesor para propagar el proceso de estabilización por toda la red. Por último, manda el mensaje UPDATE-LIST-SUCC-JOIN(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") a su nodo predecesor para que actualice sus listas de sucesores y predecesores y lo propague por el resto de la red. Se observa que en total se realizan 35 pasos para dar por finalizado el proceso.

La figura 5.7 muestra como queda el estado de los nodos que forman la red tras finalizar el proceso de estabilización. Como se puede observar marcado en rojo, el nodo con identificador 15 ha actualizado su lista de atributos incorporando al nodo con identificador 20. La lista de atributos del resto de nodos que forman la red no modifica.

Figura 5.7 Resultado tras ejecutar el proceso de estabilización por el nodo con identificador 15

```
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 0 ; 1
    ; 0 ; 0 @ "127.0.0.20") || 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 2 |-> 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 4 |-> 0 ; 0 ;
    0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") >
< 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0
    ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20"),
  PL : ((0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0
    ; 0 @ "127.0.0.20") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.20") || (0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") >
Maude>
```

5.2.4 Publicación de un fichero

En este proceso se simula que un nodo sube un fichero a la red para ponerlo a disposición de los otros usuarios. Para ello insertamos un mensaje ficticio INIT-ADD-KEY (*InfoNode,BitString*), donde *InfoNode* indica el nodo que desea publicar un fichero en la red y *BitString* el identificador de la clave del fichero a publicar. Por ejemplo, supongamos que el nodo con identificador 15 desea publicar el fichero con identificador 10111110. Nótese que el nodo con identificador 15 es el nodo que publica el fichero, sin embargo, no tiene por qué coincidir que sea el nodo encargado de almacenarlo en la red. De hecho, en este caso no coincide, el nodo 1 es el nodo que deberá almacenar la clave del fichero. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
INIT-ADD-KEY(N15, 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0) .
```

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[5] init-state .
```

A continuación se explica como se va reescribiendo el estado:

El nodo 15 recibe un mensaje ADD-KEY-NET(0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0) para que busque el nodo más cercano a la clave del fichero 10111110. El nodo 15 manda un mensaje ADD-KEY-NET(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0) al nodo 1, el cual es el nodo responsable de almacenar la clave. Se observa que en total se realizan 5 pasos para dar por finalizado el proceso.

Figura 5.8 Resultado tras publicar en la red el fichero 10111110 por el nodo con identificador 15

```
Maude2 — maude.intelDarwi — 182×49
rewrites: 26652 in 28ms cpu (29ms real) (935125 rewrites/second)
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32 |->
    0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ; 0
    ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1
    ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0
    @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ; 1
    ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1
    @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
    ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ; 1
    ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1
    @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0
    ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0
    |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0 ;
    0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ;
    1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ;
    0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    0 ; 1 @ "127.0.0.1") >
```

Como podemos observar en la figura 5.8 marcado en verde, el nodo con identificador 1 ha insertado una nueva entrada en su tabla de claves para la clave del fichero 10111110 figurando el nodo son identificador 15 como el nodo que ha publicado el fichero.

5.2.5 Eliminar la clave de un fichero

Para simular el hecho de que un nodo desea eliminar un fichero de la red que puso a disposición a los otros usuarios insertamos el mensaje ficticio INIT-DELETE-KEY(*InfoNode,BitString*), donde *InfoNode* indica el nodo que desea eliminar la clave de la red y *BitString* el identificador de la clave del fichero a eliminar. Por ejemplo, supongamos que el nodo con identificador 15 desea eliminar el fichero con identificador 10000010. Nótese que el nodo con identificador 15 es el nodo que publicó la clave del fichero en la red y ahora desea eliminarla de la red. Sin embargo, el nodo con identificador 1 es el responsable de almacenarla en la red. Por ello, en el resultado final el nodo con identificador 1 deberá haber eliminado de su tabla de claves la entrada que contiene la clave con identificador 10000010. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
INIT-DELETE-KEY(N15,1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0) .
```

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[2] init-state .
```

A continuación se explica como se va reescribiendo el estado:

El nodo 15 busca el nodo responsable (N1) de la clave del fichero 10000010 a eliminar en su tabla *finger* y le envía un mensaje DELETE-KEY-NET(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0) para que elimine la clave de su tabla de claves. Se observa que en total se realizan 2 pasos para dar por finalizado el proceso.

En la figura 5.9 podemos observar que en la tabla de claves del nodo 1, marcada en rojo, la clave del fichero 10000010 ya no está almacenada.

5.2.6 Búsqueda de una clave

El primer paso en el proceso de búsqueda es buscar el nodo responsable de almacenar la clave a buscar en la red. Una vez encontrado el nodo, se busca en su tabla de claves la clave. Si la clave está contenida en la tabla de claves, el nodo responsable de la clave envía un mensaje FILE-FOUND al nodo que inicio la búsqueda. En caso contrario, el nodo responsable de la clave envía un mensaje FILE-NOT-FOUND al nodo emisor.

Para simular la búsqueda de una clave de un fichero en la red insertamos el mensaje ficticio INIT-LOOKFOR(*InfoNode*, *BitString*), donde *InfoNode* indica el nodo que desea buscar la clave, y *BitString* es el identificador de la clave del fichero a buscar. Por ejemplo, supongamos que el nodo con identificador 1 desea buscar el fichero con identificador 00011110. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
INIT-LOOKFOR(N1, 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0) .
```

Figura 5.9 Resultado tras eliminar el fichero 10000010 de la red por el nodo con identificador 15

```
Maude2 — maude.intelDarwi — 180×48
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @
    "127.0.0.30", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ;
    1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ;
    0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1
    ; 1 ; 1 ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |->
    0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1
    ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0
    ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 0 ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ;
    1 ; 1 ; 1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |->
    0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1
    ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0
    ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1
    ; 1 ; 1 ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ;
    0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |->
    0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ;
    0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ;
    0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ;
    0 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ;
    1 ; 1 ; 1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0
    ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ;
    0 ; 0 ; 0 ; 1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0
    ; 1 ; 1 ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0
    ; 0 ; 0 ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") >
```

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[3] init-state .
```

A continuación se explica cómo se va reescribiendo el estado:

1) El nodo 1 busca el nodo responsable (N30) de la clave 00011110 y le envía el mensaje LOOKFOR-KEY(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0) para que busque la clave del fichero 00011110 en su tabla de claves.
2) La clave 00011110 está dentro de la tabla de claves del nodo 30 y envía un mensaje FILE-FOUND(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0) al nodo 1, que es el nodo que inició la búsqueda. Se observa que en total se realizan 3 pasos para dar por finalizado el proceso.

La figura 5.10 muestra marcado en verde el resultado de la búsqueda. Como se puede observar, la búsqueda ha sido satisfactoria ya que el nodo con identificador 1 recibe un mensaje FILE-FOUND del nodo con identificador 30 el cual es el responsable de almacenar la clave 00011110.

Sin embargo, podría darse el caso de que la clave a buscar no se encontrase en la red, en cuyo caso se recibe un mensaje NOT-FILE-FOUND. Veamos un ejemplo:

Supongamos que el nodo con identificador 1 desea buscar el fichero con clave 00000101. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
INIT-LOOKFOR(N1, 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1) .
```

Figura 5.10 Resultado de buscar la clave 00011110 por el nodo con identificador 1

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[3] init-state .
```

A continuación se explica como se va reescribiendo el estado:

1) El nodo 15 recibe el mensaje LOOKFOR-KEY(0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1) del nodo 1 debido a que es el nodo responsable de almacenar la clave del fichero 00000101 y se comprueba si la clave está en su tabla de claves.
2) Como la clave 00000101 no está en su tabla de claves, el nodo 15 envía el mensaje FILE-NOT-FOUND(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1) al nodo 1 para indicar que no se encuentra el fichero compartido en la red. Se observa que en total se realizan 3 pasos para dar por finalizado el proceso.

La figura 5.11 muestra marcado en verde el resultado de la búsqueda de la clave 00000101. La búsqueda ha sido fallida ya que el nodo con identificador 1 recibe un mensaje FILE-NOT-FOUND del nodo con identificador 15 el cual es el nodo responsable de almacenar la clave. Esto significa que el nodo con identificador 15 no tiene la clave 00000101 en su tabla de claves, por lo que la clave del fichero no está compartida en la red y se da por finalizada la búsqueda.

Con ello se garantiza que el algoritmo de búsqueda tiene un coste logarítmico.

Figura 5.11 Resultado de buscar la clave 00000101 por el nodo con identificador 1

5.2.7 Nodo verifica su predecesor

Cada cierto periodo de tiempo un nodo comprueba si su nodo predecesor inmediato es válido, es decir, si sigue activo. Para simular el hecho de que un nodo inicia este proceso insertamos un mensaje ficticio CHECK-PRED (*InfoNode*), donde *InfoNode* indica el nodo que inicia el proceso.

Por ejemplo, supongamos que pasado un tiempo *x* el nodo con identificador 30 inicia el proceso. Inicialmente, el nodo con identificador 30 tiene la siguiente lista de predecesores:

```
eq PL30 = N1 || N63 || N48
```

donde el nodo con identificador 1 es su predecesor inmediato. Sin embargo, como podemos observar en la figura 5.12 marcado en verde su predecesor inmediato debería ser el nodo con identificador 15 y no el nodo con identificador 1, marcado en rojo en la figura.

El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
CHECK-PRED(N30) .
```

El comando de reescritura usado para obtener el resultado del comportamiento del estado inicial sería:

```
Maude> rew[3] init-state .
```

Figura 5.12 Estado de la red antes de verificar el predecesor del nodo con identificador 30

```
Maude2 — maude.intelDarwi — 181×48
rewrites: 6615 in 7ms cpu (9ms real) (885068 rewrites/second)
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
    ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") >
Maude>
```

A continuación se explica como se va reescribiendo el estado:

1) El nodo 30 solicita a su nodo predecesor (N1) su nodo sucesor a través del mensaje GET-PRED(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30").
2) El nodo 30 recibe el nodo sucesor (N15) del nodo 1 con el mensaje CHECK-PRED-SUCC(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"). Como el nodo predecesor (N1) del nodo 30 no coincide con el nodo sucesor (N15) del nodo predecesor (N1), significa que el nodo 30 tiene un predecesor no válido y actualiza su lista de predecesores incorporando al nodo 15 como su predecesor inmediato. Se observa que en total se realizan 3 pasos para dar por finalizado el proceso.

En la figura 5.13 podemos observar marcado en verde que el nodo 30 ha actualizado su predecesor inmediato. Ahora su lista de predecesores está formada por los nodos 15, 1, 63 y 48, la cual es coherente con el estado actual de la red.

Si la información de la lista es correcta, cuando se realiza el proceso no se modificará. La figura 5.14 muestra el resultado de iniciar de nuevo el proceso cuando la lista de predecesores del nodo con identificador 30 es correcta. Como se puede observar en la figura marcado en verde, su lista de predecesores no ha sido modificada.

Figura 5.13 Resultado tras verificar el predecesor del nodo con identificador 30

```
Maude2 — maude.intelDarwi — 182×49
rewrites: 6865 in 12ms cpu (53ms real) (556501 rewrites/second)
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32 |->
    0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ; 0
    ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0
    @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ; 1
    ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1
    @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
    ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ; 1
    ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1
    @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0
    ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0
    |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0 ;
    0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ;
    1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ;
    0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    0 ; 1 @ "127.0.0.1") >
Maude>
```

Figura 5.14 Resultado tras chequear de nuevo el predecesor del nodo con identificador 30

5.2.8 Paralelismo entre procesos

El objetivo de esta sección es presentar un conjunto de pruebas en las cuales se ejecutan en paralelo dos o más procesos sobre nodos iguales o diferentes con la finalidad de mostrar el comportamiento de la especificación al paralelismo en estos casos.

La aplicación de las reglas en Maude es secuencial. Al ejecutar varios procesos en paralelo el resultado puede ser no determinista pues puede depender del orden en que se ejecuten las reglas. Un resultado es determinista cuando siempre se obtiene el mismo resultado independientemente del orden en el que se ejecuten los procesos. Un resultado es no determinista cuando se obtienen resultados distintos según el orden en que se ejecuten los procesos. Cuando los nodos sobre los que actúan los procesos son nodos diferentes el resultado de nuestra especificación será determinista. Sin embargo, cuando se ejecutan varios procesos actuando sobre un mismo nodo el resultado puede ser no determinista. Una situación muy clara, es por ejemplo, cuando se elimina un nodo y al mismo tiempo se desea ejecutar un proceso que actúa sobre ese nodo, tal como insertarle nuevas claves a su tabla de claves. En este caso, podemos estar trabajando sobre un nodo que ya no forma parte de la red. Si ambas acciones ocurren en paralelo dependiendo del orden en que se apliquen las reglas de la especificación el resultado será diferente.

A continuación presentamos varios casos de prueba que simulan varios procesos que actúan sobre los mismos nodos o nodos distintos para analizar como son los resultados de la especificación.

Empezaremos presentando el caso de prueba en el que el nodo con identificador 63 desea abandonar la red y dos nodos, con identificadores 255 y 37, desean incorporarse a la red contactando con el nodo con identificador 1.

El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
LEAVE-GROUP(N63)
JOIN-NODE(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" , 255)
JOIN-NODE(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" , 37) .
```

La figura 5.15 muestra la configuración resultante de la ejecución de los tres procesos. Como se puede observar en la figura marcado en rojo el nodo con identificador 63 ya no forma parte de la red, y marcado en verde están los nuevos nodos incorporados con identificadores 255 y 37. Además, el resto de los nodos cuyos identificadores son : 1, 15, 30 y 48, han actualizado sus listas de atributos, eliminando al nodo con identificador 63 e incorporando los nodos nuevos.

En este caso el resultado es determinista ya que los tres procesos actúan sobre nodos distintos y no afecta un proceso sobre otro. El hecho de que el nodo con identificador 63 abandone la red, no afecta a que los nodos con identificadores 255 y 37 se quieran incorporar a ella.

Probemos a ejecutar en paralelo el proceso de estabilización y otro proceso que modifica el estado de la configuración de la red, tal como, eliminar un nodo de la red. Así pues, mientras que el nodo con identificador 63 está ejecutando el proceso de estabilización, el nodo con identificador 15 decide abandonar la red.

El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
```

Figura 5.15 Resultado tras ejecutar tres acciones paralelas (abandono de un nodo e incorporación de dos nodos a la red)

```
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32
    |-> 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
   0 ; 0 @ "127.0.0.48"),
  PL : ((1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37") || (0 ; 0 ; 0 ; 1 ; 1
   ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0
   ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0
   ; 0 ; 0 @ "127.0.0.48", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ;
   0 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 1 ; 1 ; 1 ; 1 ; 1 ; 1 ;
   1 ; 1 @ "127.0.0.255"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 1 ; 0 ; 0 ;
   1 ; 0 ; 1 @ "127.0.0.37") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37", 2 |-> 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37", 4 |-> 0 ; 0
   ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1
   ; 1 ; 1 @ "127.0.0.255", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ;
   1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") & 0 ; 0 ; 0 ; 0 ; 0 ; 0
   ; 0 ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") || (0 ; 0 ; 1 ; 1 ; 0 ;
   0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37") >
< 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0
   ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 32 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1
   ; 1 ; 1 @ "127.0.0.255", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1
   ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ;
   1 ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 1 ;
   1 ; 1 ; 1 @ "127.0.0.255") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 2 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 4 |-> 1 ;
   1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 8 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 16 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 32 |-> 1 ; 1 ; 1 ; 1 ; 1
   ; 1 ; 1 ; 1 @ "127.0.0.255", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
   1 ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ;
   0 ; 0 ; 1 @ "127.0.0.1") || 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") >
< 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ;
   0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 32 |-> 0 ; 0 ; 1 ; 0 ; 0 ; 1 ;
   0 ; 1 @ "127.0.0.37", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ;
   1 ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1
   ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1 |-> 0 ; 0
   ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 0 ; 0 ; 1 ;
   0 ; 1 @ "127.0.0.37"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.37") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ;
   1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") >
Maude>
```

```
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
STABILIZE(N63)
LEAVE-GROUP(N15) .
```

La figura 5.16 muestra el resultado de la configuración tras la ejecución de los dos procesos. Como se puede observar marcado en rojo el nodo con identificador 15 ya no forma parte de la red, y el nodo con identificador 63 tiene actualizado correctamente su lista de atributos la cual está marcada en verde en la figura. Los otros nodos que conforman la red (1, 30 y 48) también han actualizado sus listas de atributos.

En este caso, el resultado es determinista ya que cada uno de los dos procesos (estabilización y abandono de la red) actúan sobre un nodo diferente (63 y 15).

Otro caso interesante es cuando un nodo desea incorporarse a la red y al mismo tiempo el nodo contacto abandona la red. Por ejemplo, el nodo con identificador 255 desea incorporarse a la red mediante el nodo con identificador 1 y el nodo con identificador 1 desea abandonar la red. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
JOIN-NODE(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" , 255)
LEAVE-GROUP(N1) .
```

La figura 5.17 muestra el resultado de la configuración tras la ejecución de los dos procesos. Como se puede observar en la figura, el nodo con identificador 1 ya no forma parte de la red, y el nodo con identificador 255 se ha incorporado correctamente a la red. Además, el resto de los nodos (15, 30, 48 y 63) han actualizado sus listas de atributos, eliminando al nodo con identificador 1 e incorporando al nodo con identificador 255.

Figura 5.16 Resultado tras ejecutar dos acciones paralelas (proceso de estabilización y abandono de un nodo de la red)

En este caso el resultado es no determinista. El resultado de la figura 5.17 muestra el resultado cuando el nodo 255 se incorpora antes de que el nodo 1 abandone la red. Sin embargo, el resultado podría verse modificado dependiendo del orden en que se ejecutan los procesos. Si, inicialmente, el nodo con identificador 1 abandona la red, el nodo con identificador 255 no podrá ser incluido ya que su nodo contacto, nodo 1, ya no forma parte de la red.

Probemos a ejecutar en paralelo el proceso de búsqueda de una clave con la publicación de un fichero en la red. Por ejemplo, el nodo con identificador 1 decide publicar el fichero con clave 11111111 en la red y al mismo tiempo el nodo con identificador 1 inicia el proceso de búsqueda de la clave con identificador 00111111. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT: FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
INIT-ADD-KEY(N1, 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1)
INIT-LOOKFOR(N1, 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1) .
```

La figura 5.18 muestra el resultado de la configuración tras la ejecución de los dos procesos. Como se puede observar marcado en rojo, el nodo con identificador 1 ha añadido en su tabla de claves la clave de fichero 11111111 de la cual él era el responsable. Por otro lado, marcado en verde, el nodo con identificador 1 recibe un mensaje FIND-FOUND del nodo 63 que indica que la clave de fichero 00111111 está compartida en la red y el nodo 63 es el responsable de almacenarla.

El resultado obtenido es determinista. En este caso los procesos actúan sobre el mismo nodo, el nodo con identificador 1, pero esto no afecta al resultado debido a que los procesos no realizan ninguna modificación sobre la lista de atributos del nodo 1.

Figura 5.17 Resultado tras ejecutar dos acciones paralelas (incorporación y abandono de un nodo de la red)

```
Maude2 — maude.intelDarwi — 180x48

result Configuration: < 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @
    "127.0.0.30", 4 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 1 ; 1 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.255"),
  PL : ((1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ;
    1 ; 1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ;
    1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ;
    0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 32 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.255", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") || 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 2 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 4 |-> 1 ; 1
    ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 8 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 16 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 32 |-> 1 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 ; 1 @ "127.0.0.255", 64 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 1 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.255") >
< 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 4 |-> 0 ; 0
    ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 128 |-> 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255"),
  KT : (0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ;
    0 ; 0 ; 1 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") >
Maude>
```

Figura 5.18 Resultado tras ejecutar dos acciones paralelas (búsqueda y publicación de una clave)

```
Maude2 — maude.intelDarwi — 181×48
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
    ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") >
FILE-FOUND(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1)
```

Otro posible ejemplo, es ejecutar en paralelo la incorporación de un nodo y la publicación de una clave cuyo nodo responsable de almacenarla es el nuevo nodo incorporado. Supongamos que el nodo con identificador 255 desea incorporarse a la red y, seguidamente, el nodo con identificador 1 publica la clave 11111111. El nodo con identificador 255 es el responsable de almacenar la clave 11111111. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT:FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
JOIN-NODE(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" , 255)
INIT-ADD-KEY(N1, 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1) .
```

La figura 5.19 muestra el resultado de la configuración tras la ejecución de los dos procesos. Como se puede observar marcado en verde el nodo con identificador 255 ahora forma parte de la red, y marcado en rojo la clave 11111111 está insertada en la tabla de claves del nodo con identificador 255 ya que es el nodo responsable.

Observar que en este caso el resultado es determinista ya que si se inserta el nodo antes que la clave, esta se insertará en el nuevo nodo, y si se inserta la clave del fichero en la red antes que el nodo, inicialmente, la clave se insertará en el nodo 1, pero en el momento de que se inserte el nodo 255 en la red la clave se transfiere al nodo 255.

Figura 5.19 Resultado tras ejecutar dos acciones paralelas (incorporación de un nodo y publicación de un fichero)

Por último, ejecutaremos dos búsquedas al mismo tiempo. Supongamos que el nodo con identificador 15 inicia la búsqueda de la clave 00011111 y el nodo con identificador 30 inicia la búsqueda de la clave 00011110. El término que representa el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT:FT15 , SL: SL15 , PL: PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT: FT30 , SL: SL30 , PL: PL30 , KT: KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT: FT48 , SL: SL48 , PL: PL48 , KT: KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT: FT63 , SL: SL63, PL: PL63 , KT: KT63 >
INIT-LOOKFOR(N15, 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1)
INIT-LOOKFOR(N30, 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0) .
```

El resultado es determinista. Varias búsquedas simultáneamente no producen ninguna modificación en las listas de atributos de los nodos, por lo que el resultado siempre va a ser determinista.

La figura 5.20 muestra marcado en verde el resultado de ejecutar de las dos búsquedas.

Figura 5.20 Resultado tras ejecutar dos búsquedas simultáneamente.

```
result Configuration: < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
    ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") >
FILE-FOUND(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0)
FILE-FOUND(0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1)
```

5.2.9 Comando search

El comando *search* obtiene las posibles soluciones al realizar un proceso mediante una búsqueda en anchura en el árbol de activaciones.

A continuación presentamos algunos ejemplos utilizando este comando.

Supongamos que queremos comprobar si a partir de un estado en el que aparece un mensaje de inserción del nodo con identificador 255 se obtiene un estado en el cual dicho nodo se ha insertado correctamente.

El término que representa el estado inicial quedaría como sigue:

```
eq init-state = < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : FT15 , SL : SL15 , PL : PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : FT30 , SL : SL30 , PL : PL30 , KT : KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : FT48 , SL : SL48 , PL : PL48 , KT : KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : FT63 , SL : SL63, PL : PL63 , KT : KT63 >
JOIN-NODE(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" , 255) .
```

Escribimos en Maude el siguiente comando:

```
Maude> search [3, 5] init-state =>+ < 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : empty , SL :
S:SuccessorList, PL : nilPredecessor , KT : empty > C:Configuration .
```

donde el valor 3 indica el número de soluciones que queremos buscar, el valor 5 la profundidad de búsqueda y en la parte de la derecha definimos la condición que deben cumplir las soluciones. En este caso queremos que en la configuración final el nodo con identificador 255 forme parte de la red y que sus tablas estén vacías salvo la lista de sucesores.

Las figuras 5.21, 5.22 y 5.23 muestran marcado en rojo las primeras tres soluciones pedidas en las que se obtiene un estado que cumple la condición.

Figura 5.21 Solución 1 del comando *Search* con condición < 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : empty , SL : S:SuccessorList, PL : nilPredecessor , KT : empty>

.

```
Solution 1 (state 1)
states: 2  rewrites: 6762 in 0ms cpu (0ms real) (21466666 rewrites/second)
C:Configuration --> FIND-SUCCESSOR(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255")
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 4 |-> 0 ; 0 ; 0
    ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ;
    0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
    ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") >
S:SuccessorList --> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"
```

Figura 5.22 Solución 2 del comando *Search* con condición < 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : empty , SL : S:SuccessorList, PL : nilPredecessor , KT : empty > .

```
Solution 2 (state 2)
states: 3  rewrites: 33526 in 21ms cpu (21ms real) (1585153 rewrites/second)
C:Configuration --> FIND-SUCCESSOR(0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255")
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 4 |-> 0 ; 0 ; 0
    ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ;
    0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
    0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
    1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
    ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
    1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
    ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
    1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
    0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
    0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
    ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
    "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
    ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
    ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") >
S:SuccessorList --> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"
```

Figura 5.23 Solución 3 del comando *Search* con condición < 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255" : Peer | FT : empty , SL : S:SuccessorList, PL : nilPredecessor , KT : empty > .

```
Solution 3 (state 3)
states: 4  rewrites: 68998 in 44ms cpu (44ms real) (1559594 rewrites/second)
C:Configuration --> FIND-SUCCESSOR(0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.255")
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 4 |-> 0 ; 0 ; 0
   ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ;
   0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0 ;
   0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ;
   1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
   ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ; 1
   ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0 ;
   0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
   ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ; 0
   ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
   ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
   1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0 ;
   1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1
   ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1
   ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1
   ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0 ;
   1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
   1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ;
   0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 ;
   0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
   ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
   1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
   ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
   "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1
   ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0
   ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0
   ; 0 ; 1 @ "127.0.0.1") >
S:SuccessorList --> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"
Maude>
```

A continuación presentamos un ejemplo en el que se comprueba si a partir de un estado en el que aparece un mensaje de búsqueda de una clave que se encuentra compartida en la red se obtiene un estado en el cual aparece un mensaje FILE-FOUND.

El término que representa el estado inicial quedaría como sigue:

```
eq init-state = < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : FT15 , SL : SL15 , PL : PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : FT30 , SL : SL30 , PL : PL30 , KT : KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : FT48 , SL : SL48 , PL : PL48 , KT : KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : FT63 , SL : SL63, PL : PL63 , KT : KT63 >
INIT-LOOKFOR(N1, 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0) .
```

Escribimos en Maude el siguiente comando:

```
Maude> search [10, 10] init-state =>+ FILE-FOUND(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ;
0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0) C:Configuration .
```

En la figura 5.24 podemos observar marcado en verde que solo hay una solución. La solución encontrada, marcada en rojo, es un estado en el cual aparece un mensaje FILE-FOUND que envía el nodo responsable de almacenar la clave 01000010 al nodo que inició la búsqueda (N1).

Completamos el ejemplo ejecutando el comando:

```
Maude> search [10, 10] init-state =>+ FILE-NOT-FOUND(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",
0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0) C:Configuration .
```

donde se modifica la condición. En este caso se comprueba si a partir del estado en el que aparece el mensaje de búsqueda de la clave 01000010 se obtiene un estado en el cual aparece un mensaje FILE-NOT-FOUND.

Figura 5.24 Resultado del comando *Search* con condición mensaje FILE-FOUND.

```
Solution 1 (state 1)
states: 2  rewrites: 6777 in 1ms cpu (1ms real) (6644117 rewrites/second)
C:Configuration --> < 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.15", 4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 32
    |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 0
    ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 0 ; 0 ; 0 ; 0 ; 1 ;
    0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  SL : ((0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & 0 ; 0 ; 1 ; 1 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.63"),
  PL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || 0 ; 0 ; 0 ; 0 ; 1 ;
    1 ; 1 ; 1 @ "127.0.0.15") >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 4 |-> 0 ; 0
    ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0
    ; 0 ; 0 @ "127.0.0.48", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  KT : (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 0 ;
    0 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    0 ; 1 @ "127.0.0.1"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || 0 ; 0 ; 0 ; 1 ; 1 ; 1
    ; 1 ; 0 @ "127.0.0.30") >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 4 |-> 0 ; 0
    ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.63", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30"),
  KT : (0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ;
    1 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.15", 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15"),
  SL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & 0 ; 0 ; 0 ; 0 ; 1 ; 1 ;
    1 ; 1 @ "127.0.0.15"),
  PL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : (1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 4 |-> 0 ; 0
    ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    0 ; 1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  KT : (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1
    ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 0 ; 1 ; 1 ;
    1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48"),
  SL : ((0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & 0 ; 0 ; 0 ; 1 ; 1 ; 1 ;
    1 ; 0 @ "127.0.0.30"),
  PL : ((0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 0 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.63") >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ;
    0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ;
    1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 128 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  KT : (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 1 ;
    1 ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.48", 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
  SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & 0 ; 0 ; 1 ; 1 ; 0 ; 0 ;
    0 ; 0 @ "127.0.0.48"),
  PL : ((0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ;
    0 ; 0 ; 1 @ "127.0.0.1") >

No more solutions.
states: 3  rewrites: 6778 in 3ms cpu (3ms real) (2146974 rewrites/second)
```

La figura 5.25 muestra en rojo que no se encuentra solución. Esto es debido a que la clave del fichero si se encuentra almacenada y por ello no se puede dar un estado que cumpla la condición definida.

Figura 5.25 Resultado del comando *Search* con condición mensaje FILE-NOT-FOUND.

```
Maude> search [10, 10] init-state =>+ FILE-NOT-FOUND(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0) C:Con
figuration .
search [10, 10] in INIT : init-state =>+ C:Configuration
FILE-NOT-FOUND(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0) .

No solution.
states: 3  rewrites: 6778 in 2ms cpu (3ms real) (2269926 rewrites/second)
Maude>
```

Con este comando hemos probado que si un fichero está publicado en la red siempre se encuentra.

Capítulo 6 - Conclusiones

En este documento hemos presentado una especificación en Maude de un sistema *peer-to-peer* implementado usando la Tabla Hash Distribuida Chord. La especificación ha sido diseñada para la compartición de archivos, de forma que los nodos puedan conectarse de la red, desconectarse de la red, publicar un fichero o buscar un fichero. Este ha sido en todo momento un objetivo ambicioso y motivador.

Además de la especificación, en este trabajo se presentan distintas Tablas Hash Distribuidas con el fin de dar una visión al lector de las posibles formas de implementar un sistema *peer-to-peer*.

La especificación expresa en un lenguaje formal la descripción del protocolo, lo que permite clarificar los procesos presentados en la especificación informal y puede ser una fuente útil de consulta a los usuarios que quieran desarrollar el sistema.

Esta especificación es un punto de partida para futuros estudios sobre el comportamiento del protocolo con la finalidad de compararlo con otras tablas hash o introducir mejoras de sus funcionalidades.

Uno de los trabajos futuros pendientes es simular en tiempo real la especificación usando la herramienta "Real-Time" de Maude con el fin de hacer lo más realista posible la especificación.

Otra de las posibles mejoras es ampliar la especificación para que sea capaz de detectar posibles ataques en la red, tales como el ataque Sybil o los ataques a las tablas de encaminamiento y almacenamiento.

6.1 Reflexión

Habiendo finalizado el trabajo me siento satisfecha por el trabajo realizado y por alcanzar los objetivos planteados desde su comienzo.

Respecto al desarrollo del proyecto, mencionar que hemos tenido que afrontar distintos desafíos sin que estos hayan afectado al alcance de los objetivos planteados inicialmente. La realización de la especificación en el lenguaje Maude ha supuesto todo un reto, ya que la autora no tenía ningún conocimiento previo de dicho lenguaje. Por otro lado, el trabajar a distancia con mi coordinadora durante la realización del trabajo ha constituido otro desafío para el desarrollo del proyecto.

A pesar de todo, se ha desarrollado una solución completa y funcional que cumple con los objetivos del trabajo.

Además, ha sido una experiencia muy valiosa para mi desarrollo profesional ya que he adquirido conocimientos técnicos pertenecientes a distintas ramas de la informática y he aprendido a planificar, gestionar y desarrollar un proyecto *software* por completo enmarcado en el ámbito de la especificación formal.

Por último me gustaría destacar el interés despertado en la especificación formal de los sistemas distribuidos centralizados como consecuencia del trabajo e investigaciones realizadas.

Bibliografía

[1] aMule homepage http://www.amule.org

[2] Bakhshi, R. y Gurov, D. Verification of peer-to-peer algorithms: A case study. Electronic Notes Theoretical Computer Sciences. Paginas 35–47, 2007.

[3] BitTorrent homepage http: //www.bittorrent.com

[4] Bolosky, W. J., Douceur, J. R., Ely, D. y Theimer, M. Feasibility of a severless distributed file system deployed on an existing set of desktop PCs. *Proceedings of the ACM international conference on Measurement and modeling of computer systems. SIGMETRICS (2000)*, páginas 34-43, Santa Clara, CA. Junio 2000.

[5] Castro, M., Costa, M. y Rowstron, A. Performance and Dependability of Structured Peer-to-Peer Overlays. *In Proc. International Conference on Dependable Systems and Networks*. IEEE Computer Society Press, Los Alamitos, CA., páginas 9–18. 2004.

[6] Castro, M., Druschel, P., Kermarrec, A-M., Nandi, A., Rowstron, A. y Singht, A. SplitStream: high-bandwidth multicast in cooperative environments. *In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP 2003)*, páginas. 298-313. ACM Press, Nueva York, 2003.

[7] Castro, M., Druschel, P., Kermarrec, A-M. y Rowstron, A. Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. IEEE Journal on Selected Areas in Communication 20, paginas 100–110.Octubre 2002.

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. y Talcott, C. All About Maude: A High-Performance Logical Framework, volume 4350 of Lecture Notes in Computer Science. Springer, 2007.

[9] Crosby, S. y Wallach, D. An Analysis of BitTorrent's Two Kademlia-Based DHTs Technical Report TR-07-04, Department of Computer Science, Rice University, Houston, TX, USA., 2007.

[10] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R. y Stoica, I. CFS: Wide-area cooperative storage with CFS. *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01),* Octubre 2001.

[11] Douceur, J. R. The Sybil Attack. *In Proc. 1st International Workshop on Peer-to-Peer Systems*. Lecture Notes on Computer Science, vol. 2429. Springer-Verlag, páginas 251–260, Londres, UK, 2002.

[12] Druschel, P. y Rowstron, A. PAST: A large-scale, persistent peer-to-peer storage utility. *In Proceedings of the Hot Topics in Operating Systems(HotOS-VIII).* Schloss Elmau, Alemania, Mayo 2001.

[13] eMule homepage http://www.emule-project.net

[14] Gnutella homepage http://gnutella.wego.com/

[15] Gupta, I., Kenneth, P., Prakash, L., Demers, A. y Renesse, V. Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead. Páginas 160-169. 2003.

[16] Iyer, S., Rowstron, A. y Druschel, P. Squirrel: A descentralized peer-to-peer web cache. *In Proceedings of Principles of Distributed Computing'02*, 2002.

[17] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C. y Zhao B. OceanStore: An Arquitecture for Global-Scale Persistent Storage. *In Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS 2000),* Noviembre 2000.

[18] Leong, B., Liskov, B. y Demaine, E. D. EpiChord: Parallelizing the Chord lookup algorithm with reactive routing state management. MIT Technical Report MIT-LCS-TR-963, (Cambridge, MA), Agosto 2004.

[19] Lu, T., Merz, S. y Weidenbach, C. Model checking the Pastry routing protocol. In J. Bendisposto, M. Leuschel, and M. Roggenbach, editors, 10th International Workshop Automatic Verification of Critical Systems, AVOCS 2010, páginas 19–21. Universität Düsseldorf, 2010.

[20] Malkhi, D. y Ratajczak, D. Viceroy : A Scalable and Dynamic Emulation of the Butterfly. *Proceedings of the twentyfirst annual symposium on Principles of distributed computing,* 2002.

[21] Maymounkov, P. y Mazieres, D. Kademlia: A peer-to-peer Information System Based on the XOR Metric. *In Proceedings of the 1st International Workshop on Peer-to Peer Systems (IPTPS02)*, 2002.

[22] Mysicka, D. Reverse Engineering of eMule. An analysis of the implementation of Kademlia in eMule. Semester thesis, Department of Computer Science, Distributed Computing group, ETH Zurich, 2006.

[23] Meseguer, J. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96(1):73–155, 1992.

[24] Napster homepage http://www.napster.com

[25] Pita, I. A formal specification of the Kademlia distributed hash table. In V. M. Gulías, J. Silva, and A. Villanueva, editors, *Proceedings of the 10 Spanish Workshop on Programming Languages*, PROLE 2010, páginas 223–234. Ibergarceta Publicaciones, 2010. http://www.maude.sip.ucm.es/kademlia. Informal publication–Work in progress.

[26] Pita, I. y Riesco, A. Specifying and analyzing the kademlia protocol in Maude. En 9th International Workshop on Rewriting Logic and its Applications, WRLA 2012.

[27] Ratnasamy, S., Francis, P., Handley, M., Karp, R. y Shenker, S. CAN: A Scalable Content-Addressable Network. *Proceedings of ACM SIGCOMM,* 2001.

[28] Rowstron, A. y Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001.

Proceedings In Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (2001), paginas 329-350. 2001.

[29] Rowstron, A., Kermarrec, A-M., Castro, M. y Druschel, P. Scribe: The design of a large-scale event notification infrastructure. *In Proceedings of NGC 2001*, Noviembre 2001.

[30] Saroiu, S, Gummadi, P. y Gribble, S. A Measurement Study of Peer-to-Peer File Sharing Systems. Technical Report UW-CSE-01-06-02, Department of Computer Science and Engineering, University of Washington, Julio 2001.

[31] Sit, E., Dabek, F. y Robertson, J. UsenetDHT: A low overhead usenet server. *Proceedings of the 3nd IPTPS*, 2004.

[32] Sit, E. y Morris, R. Security Considerations for Peer-to-Peer Distributed Hash Tables. *In Proc. 1st International Workshop on Peer-to-Peer Systems (Cambridge, MA).* Lecture Notes on Computer Science, vol. 2429. Springer-Verlag, Berlin, páginas 261–269. 2002.

[33] Steinmetz, R. y Wehrle, K. Peer-to-Peer Systems and Applications. Lecture Notes in Computer Science 3485 Springer 2005, ISBN 3-540-29192-X.

[34] Stoica, I., Morris, R., Karger, D., Kaashoek, M. y Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. IEEE/ACM Trans. Netw., vol. 11, número 1, páginas 17-32. 2003.

[35] Stutzbach, D. y Rejaie, R. Improving lookup performance over a widely-deployed DHT. *In Proc. Infocom 06*, Abril 2006.

[36] Wang, P., Tyra, J., Chan-Tin, E., Malchow, T., Foo Kune, D., Hopper, N. Y Kim, Y. Attacking the Kad network. *In proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*. 2008.

[37] Waldman, M., Rubin, A. D. y Cranor, L. F. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. *In Proceedings of the 9th USENIX Security Symposium,* páginas 59-71, Agosto 2000.

Apéndice A - Glosario de términos y acrónimos

Peer-to-peer (*P2P*)**.** Red informática formada por un conjunto de nodos que se comportan al mismo tiempo como clientes y como servidores. Por ello, todos los nodos se comportan igual y pueden realizar las mismas operaciones. Algunos ejemplos de redes P2P son: *Napster*, *Kazaa* o *eMule*.

Sistemas distribuidos. Sistemas cuyos componentes *hardware* y *software*, que están en ordenadores conectados en red, se comunican y coordinan sus acciones mediante el paso de mensajes, con el fin de alcanzar un objetivo. Se establece la comunicación mediante un protocolo prefijado por un esquema cliente-servidor .

Maude. Lenguaje de especificación formal desarrollado por un grupo de investigadores dirigidos por J. Meseguer en el *Computer Science Laboratory* de *SRI International* (EE.UU.). El lenguaje Maude está basado en la lógica de reescritura.

Protocolo. Conjunto conocido de reglas y formatos que se utilizan para la comunicación entre procesos que realizan una determinada tarea.

Malware (Malicious Software). Programas cuya función es dañar un sistema o causar un mal funcionamiento.

eD2K. Enlace que permite localizar archivos de forma unívoca dentro de la red *peer-to-peer* eDonkey.

Kad. Red *peer-to-peer* cuya implementación está basada en la Tabla Hash Distribuida Kademlia. Fue desarrollada por los creadores de eMule.

Software. Conjunto de programas, instrucciones y reglas informáticas que permiten ejecutar distintas tareas en una computadora.

Hardware. Conjunto de componentes que conforman la parte física de una computadora. El hardware define los componentes físicos internos (disco duro, microprocesador, circuitos, cables,…) y a los periféricos (escáner, impresoras,…).

Sistema UNIX. Sistema operativo multitarea y multiusuario desarrollado en los laboratorios *Bell* por Kernighan y Thompson.

THD (Tabla Hash Distribuida). Estructura de datos que permite localizar de forma eficiente la información compartida en la red. Las claves de los ficheros se almacenan en los nodos que forman parte de la red.

XOR. Es una operación lógica que consiste en sumar los productos de dos variables, cuyo producto se caracteriza por tener variables de forma negada y no negada.

CAN (Content Addressable Network). Es un ejemplo de una Tabla Hash Distribuida desarrollada en la Universidad de California por Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp y Scott Shenker.

Función SHA-1. Una de las funciones más usada del grupo de funciones hash criptográficas SHA. Fue diseñada por la Agencia de Seguridad Nacional de los Estados Unidos. La función SHA-1 genera valores de 160 bits.

CFS (Cluster File System). Sistema de archivos que se puede aplicar simultáneamente en varios nodos.

GUID (Globally Unique Identifier). Es un identificador alfanumérico que identifica de forma única un nodo o clave de un fichero.

TLA (Temporal Logic of Action). Lógica para especificar sistemas concurrentes.

Apéndice B - Red con 15 nodos. Estado inicial y análisis

B.1 Estado inicial

Este ejemplo es un anillo Chord de 8 bits cuyo estado inicial está formado por 15 nodos cuyos identificadores son: 1, 15, 25, 30, 48, 63, 79, 86, 96, 102, 128, 170, 212, 223 y 245. Este ejemplo se utilizan 8 bits para representar los identificadores de los nodos y los identificadores de las claves.

Figura B.1 Representación del estado inicial del anillo

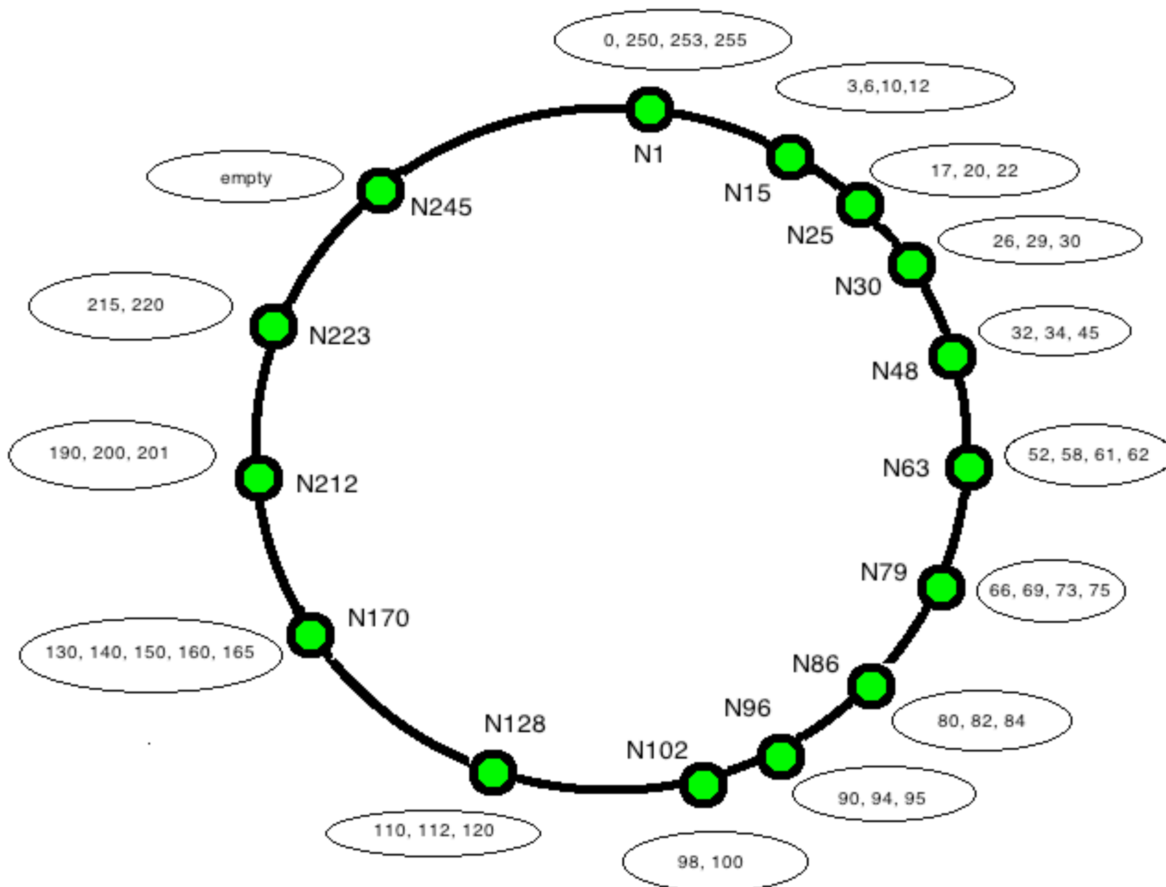


Tabla B.1 Ficheros que comparte cada uno de los 15 nodos de la red

Identificador del nodo	Identificador del fichero
1	26, 62, 90, 100
15	0, 6, 32, 82, 98, 190, 250
25	17
30	12, 255
48	10, 84, 120, 201, 253
63	3, 75, 112, 220
79	20, 66, 110
86	34, 80
96	45, 73, 95
102	52, 130
128	29, 61, 94, 140
170	---
212	58, 150
223	30, 69, 215
245	22, 165

```
ops N1 N15 N25 N30 N48 N63 N79 N86 N96 N102 N128 N170 N212 N223 N245 :
    -> Info-Node .
```

```
eq N1 = 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" .
eq N15 = 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" .
eq N25 = 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" .
eq N30 = 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" .
eq N48 = 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" .
eq N63 = 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" .
eq N79 = 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" .
eq N86 = 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" .
eq N96 = 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" .
eq N102 = 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" .
eq N128 = 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" .
eq N170 = 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" .
eq N212 = 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" .
eq N223 = 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" .
eq N245 = 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" .

ops FT1 FT15 FT25 FT30 FT48 FT63 FT79 FT86 FT96 FT102 FT128 FT170 FT212
FT223 FT245 : -> FingerTable .

ops SL1 SL15 SL25 SL30 SL48 SL63 SL79 SL86 SL96 SL102 SL128 SL170 SL212
SL223 SL245 : -> SuccessorList.

ops KT1 KT15 KT25 KT30 KT48 KT63 KT79 KT86 KT96 KT102 KT128 KT170
KT212 KT223 KT245 : -> KeysTable.

ops PL1 PL15 PL25 PL30 PL48 PL63 PL79 PL86 PL96 PL102 PL128 PL170 PL212
PL223 PL245 : -> PredecessorList.

eq FT1 = 1 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         2 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         4 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
```

```
                  8 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
                 16 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
                 32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
                 64 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
                128 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" .

eq FT15 = 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
          2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
          4 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
          8 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
         16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         32 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         64 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
        128 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" .

eq FT25 = 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
          2 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
          4 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
          8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
         64 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
        128 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" .

eq FT30 = 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
          2 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
          4 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
          8 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         16 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         32 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
         64 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
```

```
            128 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" .

eq FT48 = 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
              2 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
              4 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
              8 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
             16 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
             32 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" ,
             64 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
            128 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" .

eq FT63 = 1 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
              2 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
              4 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
              8 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
             16 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
             32 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
             64 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
            128 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" .

eq FT79 = 1 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" ,
              2 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" ,
              4 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" ,
              8 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
             16 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
             32 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
             64 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
            128 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" .

eq FT86 = 1 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
              2 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
```

```
              4 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
              8 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
             16 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" ,
             32 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
             64 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
            128 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" .

eq FT96 = 1 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" ,
          2 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" ,
          4 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" ,
          8 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
         16 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
         32 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
         64 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
        128 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" .

eq FT102 = 1 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
           2 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
           4 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
           8 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
          16 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
          32 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
          64 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
         128 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" .

eq FT128 = 1 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
           2 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
           4 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
           8 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
          16 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
          32 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" ,
```

```
                64 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.212" ,
               128 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" .

eq FT170 = 1 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.212" ,
           2 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.212" ,
           4 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.212" ,
           8 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.212" ,
          16 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.212" ,
          32 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.212" ,
          64 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
         128 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" .

eq FT212 = 1 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" ,
           2 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" ,
           4 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" ,
           8 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" ,
          16 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
          32 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
          64 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
         128 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" .

eq FT223 = 1 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
           2 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
           4 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
           8 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
          16 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" ,
          32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",
          64 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         128 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" .

eq FT245 = 1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",
```

```
                2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",
                4 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",
                8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1",
               16 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
               32 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
               64 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
              128 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" .

eq KT1 = 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         1 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
         1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
         1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" .

eq KT15 = 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"  ,
          0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
          0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" ,
          0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" .

eq KT25 = 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" ,
          0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
          0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" .

eq KT30 = 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 0 ; 1 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
          0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" .

eq KT48 = 0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
          0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" ,
          0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 ; 1 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 @ "127.0.0.96" .

eq KT63 = 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" ,
```

```
               0 ; 0 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" ,
               0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
               0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" .

eq KT79 = 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
          0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" ,
          0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" ,
          0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" .

eq KT86 = 0 ; 1 ; 0 ; 1 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" ,
          0 ; 1 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
          0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" .

eq KT96 = 0 ; 1 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
          0 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
          0 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" .

eq KT102 = 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
           0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" .

eq KT128 = 0 ; 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" ,
           0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" ,
           0 ; 1 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" .

eq KT170 = 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" ,
           1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" ,
           1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" ,
           1 ; 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" ,
           1 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" .

eq KT212 = 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" ,
```

```
                    1 ; 1 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" ,
                    1 ; 1 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" .

eq KT223 = 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" ,
                    1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" .

eq KT245 = empty .

--- Las listas de sucesores y predecesores están formadas por solo los 5 primeros nodos
--- sucesores y los 5 primeros nodos predecesores.

eq SL1 = N15 & N25 & N30 & N48 & N63 .
eq SL15 = N25 & N30 & N48 & N63 & N79 .
eq SL25 =  N30 & N48 & N63 & N79 & N86 .
eq SL30 = N48 & N63 & N79 & N86 & N96 .
eq SL48 = N63 & N79 & N86 & N96 & N102 .
eq SL63 = N79 & N86 & N96 & N102 & N128 .
eq SL79 = N86 & N96 & N102 & N128 & N170 .
eq SL86 = N96 & N102 & N128 & N170 & N212 .
eq SL96 = N102 & N128 & N170 & N212 & N223 .
eq SL102 = N128 & N170 & N212 & N223 & N245 .
eq SL128 = N170 & N212 & N223 & N245 & N1 .
eq SL170 = N212 & N223 & N245 & N1 & N15 .
eq SL212 = N223 & N245 & N1 & N15 & N25 .
eq SL223 = N245 & N1 & N15 & N25 & N30 .
eq SL245 = N1 & N15 & N25 & N30 & N48 .

eq PL1 = N245 || N223 || N212 || N170 || N128 .
eq PL15 = N1 || N245 || N223 || N212 || N170 .
eq PL25 = N15 || N1 || N245 || N223 || N212 .
eq PL30 = N25 || N15 || N1 || N245 || N223 .
```

eq PL48 = N30 || N25 || N15 || N1 || N245 .

eq PL63 = N48 || N30 || N25 || N15 || N1 .

eq PL79 = N63 || N48 || N30 || N25 || N15 .

eq PL86 = N79 || N63 || N48 || N30 || N25 .

eq PL96 = N86 || N79 || N63 || N48 || N30 .

eq PL102 = N96 || N86 || N79 || N63 || N48 .

eq PL128 = N102 || N96 || N86 || N79 || N63 .

eq PL170 = N128 || N102 || N96 || N86 || N79 .

eq PL212 = N170 || N128 || N102 || N96 || N86 .

eq PL223 = N212 || N170 || N128 || N102 || N96 .

eq PL245 = N223 || N212 || N170 || N128 || N102 .

B.2 Análisis

B.2.1 Búsqueda de la clave 01011010 por el nodo con identificador 223

El término que define el estado inicial quedaría como sigue:

eq init-state =

< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >

< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : FT15 , SL : SL15 , PL : PL15 , KT : KT15 >

< 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" : Peer | FT : FT25 , SL : SL25 , PL : PL25 , KT : KT25 >

< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : FT30 , SL : SL30 , PL : PL30 , KT : KT30 >

< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : FT48 , SL : SL48 , PL : PL48 , KT : KT48 >

< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : FT63 , SL : SL63 , PL : PL63 , KT : KT63 >

< 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" : Peer | FT : FT79 , SL : SL79 , PL : PL79 , KT : KT79 >

< 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" : Peer | FT : FT86 , SL : SL86 , PL : PL86 , KT : KT86 >

< 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" : Peer | FT : FT96 , SL : SL96 , PL : PL96 , KT : KT96 >

< 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" : Peer | FT : FT102 , SL : SL102 , PL : PL102 , KT : KT102 >

< 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" : Peer | FT : FT128 , SL : SL128 , PL : PL128 , KT : KT128 >

< 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" : Peer | FT : FT170 , SL : SL170 , PL : PL170 , KT : KT170 >

< 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" : Peer | FT : FT212 , SL : SL212 , PL : PL212 , KT : KT212 >

< 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" : Peer | FT : FT223 , SL : SL223 , PL : PL223 , KT : KT223 >

< 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" : Peer | FT : FT245 , SL : SL245 , PL : PL245 , KT : KT245 >

INIT-LOOKFOR(N223, 0 ; 1 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0) .

La figura B.2 muestra en verde el mensaje obtenido como resultado de la búsqueda. La búsqueda ha sido satisfactoria ya que el nodo con identificador 223 recibe un mensaje FILE-FOUND del nodo con identificador 96 el cual es el responsable de almacenar la clave 01011010.

Figura B.2 Resultado de buscar la clave 01011010 por el nodo con identificador 223

Maude2 — maude.intelDarwi — 181×48

```
  0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") & (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @
  "127.0.0.86") & (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") & (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") & 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128"),
 PL : ((1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") || (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") || (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") || (0 ; 1 ; 0 ; 1 ; 0 ;
  1 ; 1 ; 0 @ "127.0.0.86") || (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48")
  || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0
  ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") || (1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223") || 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") >
< 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" : Peer | FT : (1 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 2 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 4 |-> 1 ; 1
  ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 8 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 16 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245", 32 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1
  ; 0 ; 1 @ "127.0.0.245", 64 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25", 128 |-> 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86"),
 KT : (1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 1 ; 1 ; 0 ;
  0 ; 1 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1"),
 SL : ((1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223") & (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ;
  1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ;
  0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") & (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") & (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @
  "127.0.0.96") & (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") & (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") & 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170"),
 PL : ((1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") || (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") || (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") || (0 ; 1 ; 1 ; 0 ; 0
  ; 0 ; 0 ; 0 @ "127.0.0.96") || (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") || (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @
  "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") || (0 ; 0 ;
  0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") || 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @
  "127.0.0.223") >
< 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" : Peer | FT : (1 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245", 2 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245", 4 |-> 1 ; 1
  ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245", 8 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245", 16 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245", 32 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0
  ; 0 ; 1 @ "127.0.0.1", 64 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48", 128 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96"),
 KT : (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 1 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63"),
 SL : ((1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 0 ;
  0 ; 1 @ "127.0.0.25") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ;
  1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") & (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") & (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") & (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @
  "127.0.0.102") & (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") & (1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") & 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212"),
 PL : ((1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") || (1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") || (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") || (0 ; 1 ; 1 ; 0 ; 0
  ; 1 ; 1 ; 0 @ "127.0.0.102") || (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") || (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") || (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
  "127.0.0.79") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ;
  0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @
  "127.0.0.245") >
< 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" : Peer | FT : (1 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 2 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 4 |-> 0 ; 0 ; 0
  ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 8 |-> 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1", 16 |-> 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15", 32 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1
  @ "127.0.0.25", 64 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 128 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128"),
 KT : empty,
 SL : ((0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1
  ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") & (0 ; 1
  ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") & (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") & (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") & (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @
  "127.0.0.128") & (1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") & (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") & 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223"),
 PL : ((1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223") || (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") || (1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") || (1 ; 0 ; 0 ; 0 ; 0
  ; 0 ; 0 ; 0 @ "127.0.0.128") || (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") || (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") || (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @
  "127.0.0.86") || (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ;
  0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @
  "127.0.0.1") >
FILE-FOUND(0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96", 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 0 ; 1 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0)
Maude>
```

B.2.2 Nodo con identificador 150 se incorpora a la red a través de su nodo contacto, el nodo con identificador 86

El término que define el estado inicial quedaría como sigue:

```
eq init-state =
< 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1" : Peer | FT : FT1 , SL : SL1 , PL : PL1 , KT : KT1 >
< 0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15" : Peer | FT : FT15 , SL : SL15 , PL : PL15 , KT : KT15 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25" : Peer | FT : FT25 , SL : SL25 , PL : PL25 , KT : KT25 >
< 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30" : Peer | FT : FT30 , SL : SL30 , PL : PL30 , KT : KT30 >
< 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48" : Peer | FT : FT48 , SL : SL48 , PL : PL48 , KT : KT48 >
< 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63" : Peer | FT : FT63 , SL : SL63 , PL : PL63 , KT : KT63 >
< 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79" : Peer | FT : FT79 , SL : SL79 , PL : PL79 , KT : KT79 >
< 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86" : Peer | FT : FT86 , SL : SL86 , PL : PL86 , KT : KT86 >
< 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96" : Peer | FT : FT96 , SL : SL96 , PL : PL96 , KT : KT96 >
< 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102" : Peer | FT : FT102 , SL : SL102 , PL : PL102 , KT : KT102 >
< 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" : Peer | FT : FT128 , SL : SL128 , PL : PL128 , KT : KT128 >
< 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" : Peer | FT : FT170 , SL : SL170 , PL : PL170 , KT : KT170 >
< 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" : Peer | FT : FT212 , SL : SL212 , PL : PL212 , KT : KT212 >
```

< 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223" : Peer | FT : FT223 , SL : SL223 , PL : PL223 , KT : KT223 >
< 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245" : Peer | FT : FT245 , SL : SL245 , PL : PL245 , KT : KT245 >
JOIN-NODE(0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86", 150) .

La figura B.3 indica en verde el nodo 150 incorporado a la red. Las líneas rojas indican las actualizaciones en la lista de atributos de su nodo sucesor (N170) y de su nodo predecesor (N128). Las líneas azules muestran algunas de las actualizaciones realizadas por los otros nodos que forman la red. Nótese que no se muestra todo el resultado final de la inserción debido a su longitud.

Figura B.3 Resultado tras la incorporación del nodo con identificador 150 a la red

```
Maude2 — maude.intelDarwi — 181×49
    (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") || (1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1
    ; 1 @ "127.0.0.223") || (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") || (1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") || (1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150") ||
    1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") >
< 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128" : Peer | FT : (1 |-> 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150", 2 |-> 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150", 4 |-> 1 ; 0
    ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150", 8 |-> 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150", 16 |-> 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150", 32 |-> 1 ; 0 ; 0 ; 1 ; 0 ; 1
    ; 1 ; 0 @ "127.0.0.150", 64 |-> 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150", 128 |-> 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150"),
  KT : (0 ; 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 0 |-> 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79", 0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63", 0 ; 1 ; 1 ;
    1 ; 1 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48"),
  SL : ((1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150") & (1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") & (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") & (1 ; 1 ; 0 ; 1 ; 1 ; 1
    ; 1 ; 1 @ "127.0.0.223") & (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0
    ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.63") & (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") & (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") & (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") & 0 ; 1 ; 1 ; 0
    ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102"),
  PL : ((0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") || (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") || (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") || (0 ; 1 ; 0 ; 0 ; 1 ;
    1 ; 1 ; 1 @ "127.0.0.79") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30")
    || (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 0 ; 1 ;
    0 ; 1 @ "127.0.0.245") || (1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223") || (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") || (1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170")
    || 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150") >
< 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150" : Peer | FT : (1 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170", 2 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170", 4 |-> 1 ; 0
    ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170", 8 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170", 16 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170", 32 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0
    ; 1 ; 0 @ "127.0.0.170", 64 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170", 128 |-> 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170"),
  KT : (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 |-> 0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102", 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 |-> 1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128", 1 ; 0 ; 0
    ; 1 ; 0 ; 1 ; 1 ; 0 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212"),
  SL : ((1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") & (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") & (1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223") & (1 ; 1 ; 1 ; 1 ; 0 ; 1
    ; 0 ; 1 @ "127.0.0.245") & (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") & (0
    ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.79") & (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") & (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") & (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") & 1 ; 0 ; 0 ;
    0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128"),
  PL : ((1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") || (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") || (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") || (0 ; 1 ; 0 ; 1 ; 0 ;
    1 ; 1 ; 0 @ "127.0.0.86") || (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48")
    || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") || (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") || (1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223") || (1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212")
    || 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170") >
< 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 @ "127.0.0.170" : Peer | FT : (1 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212", 2 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212", 4 |-> 1 ; 1
    ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212", 8 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212", 16 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212", 32 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1
    ; 0 ; 0 @ "127.0.0.212", 64 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212", 128 |-> 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212"),
  KT : (1 ; 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 |-> 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30", 1 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 |-> 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245"),
  SL : ((1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") & (1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223") & (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") & (0 ; 0 ; 0 ; 0 ; 0 ; 0
    ; 0 ; 1 @ "127.0.0.1") & (0 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") & (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") & (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") & (0 ;
    0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") & (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.63") & (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") & (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @
    "127.0.0.86") & (0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.96") & (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") & (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") & 1 ; 0 ; 0 ;
    1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150"),
  PL : ((1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.150") || (1 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.128") || (0 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.102") || (0 ; 1 ; 1 ; 0 ; 0
    ; 0 ; 0 ; 0 @ "127.0.0.96") || (0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 @ "127.0.0.86") || (0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.79") || (0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.63") || (0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 @ "127.0.0.48") || (0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 0 @ "127.0.0.30") || (0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1 @ "127.0.0.25") || (0 ; 0 ;
    0 ; 0 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.15") || (0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1 @ "127.0.0.1") || (1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 @ "127.0.0.245") || (1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @
    "127.0.0.223") || 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212") >
< 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 @ "127.0.0.212" : Peer | FT : (1 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 2 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 4 |-> 1 ; 1
    ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 8 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 16 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1 @ "127.0.0.223", 32 |-> 1 ; 1 ; 0 ; 1 ; 1 ; 1
```