5. NODOS BGP

La clase *BGPnode* emula el comportamiento de un nodo BGP. Es, por tanto, la base de este simulador y todas las demás clases están orientadas al funcionamiento de esta clase según rige la **RFC 1771**.

La clase *BGPnode* deriva de la clase *Module* del simulador (librería *simul*), ya que esta última clase representa a aquellos elementos de la simulación que son capaces de intercambiar información entre sí (esta información está contenida en objetos de la clase *Packet*) habiendo "conectado" previamente. No se ha escogido la clase *Node* como clase padre porque el elemento diferencial respecto a *Module* es la inclusión de colas, y no hacemos uso de éstas en *BGPnode*.

Como se ha visto en secciones anteriores de esta memoria, un nodo BGP (un objeto de la clase *BGPnode*) puede mantener distintas conexiones con otros nodos BGP (lo que serían sus pares BGP), estas conexiones están gestionadas por gestores de conexión (objetos *gestConn*) que tendrán un identificador de conexión asociado idéntico en ambos extremos de la conexión, es decir, en cada uno de los nodos BGP implicados en dicha conexión. No obstante, algunos de los procesos que se desarrollan en cada una de las conexiones afectarán al comportamiento de las demás (por ejemplo, la elección de una nueva ruta, conocida a través de una determinada conexión, como factible, hará que un nodo BGP comunique a sus pares la presencia de dicha ruta a través de las demás conexiones). Este tipo de procesos que aunque son particulares de una conexión, afectan a las demás, se han incluido dentro de la clase *BGPnode*. Los procesos que por el contrario, sólo afectan a la conexión donde tienen lugar se localizan en la clase *gestConn* (por ejemplo la elección del valor del *Hold Timer*).

En la clase *BGPnode* se sitúan las Bases de Información de Enrutamiento Local (*Loc_RIB*) y de Entrada (*Adj_RIB_In*). En la primera de ellas, *Loc_RIB*, se almacenarán las rutas que se han tomado como válidas para el enrutamiento del tráfico hacia los posibles destinos. En *Adj_RIB_In* se almacenarán las rutas que se conocen a través de nodos BGP vecinos, sean o no las escogidas para enrutar el tráfico; sobre este conjunto de rutas se aplicará el proceso de decisión que dará como resultado una nueva entrada en *Loc_RIB*. Ambos conjuntos de rutas se han situado en la clase *BGPnode* y no en *gestConn* debido al carácter global de ambas, es decir, en *Adj_RIB_In* se almacenan todas las rutas entrantes y de entre ellas saldrán las rutas que se usen para el encaminamiento, en *Loc_RIB* estarán esas rutas. En el caso de las *Adj_RIB_Out's*, se ha implementado una en cada gestor de conexión porque están más ligadas a cada una de las conexiones al tratarse de las rutas que se anuncian a cada uno de los pares BGP vecinos.

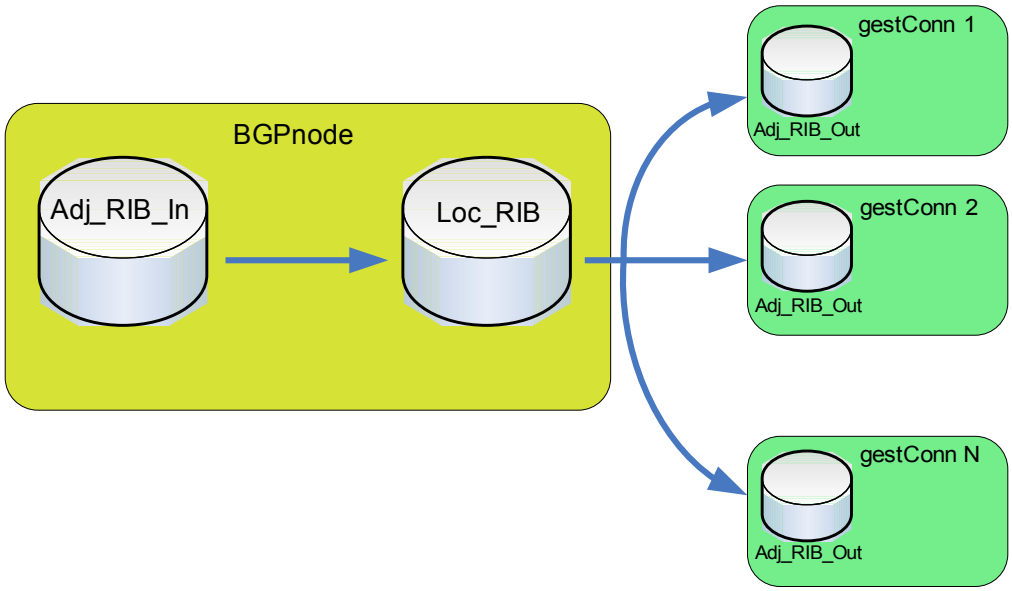

Figura 10. Implementación de RIBs.

La clase *BGPnode* implementa el envío y recepción de los distintos tipos de mensajes descritos por la **RFC 1771** (OPEN, KEEPALIVE, UPDATE y NOTIFICATION), así como de su procesado para que posteriormente el gestor de conexión apropiado actúe en consecuencia. Vemos, pues, que la clase *BGPnode* desarrolla funciones de transporte al servicio de los gestores de conexión, aunque obviamente no se limita a esto. Entre otras cosas verifica la integridad de la trama (con el chequeo del campo *marker* de la cabecera), elimina aquellas rutas que han sido declaradas como no válidas en mensajes UPDATE y aplica el proceso de decisión sobre aquellas rutas que son anunciadas como válidas por los nodos BGP vecinos. También se verifica que para cada una de las conexiones no expire el *Hold Timer*.

Por otra parte, se ha implementado en *BGPnode* funciones propias de la simulación, como la simulación de la caída de un nodo o de un enlace, simulación de errores en transmisión, activación de nodos en instantes concretos de la simulación, etc… Además de funciones de apoyo o de utilidad para otros procesos, como puede ser la comparación de estructuras de tipo *rib* o de tipo *nlri*, o la búsqueda de un gestor de conexión concreto en función de su identificador.

5.1. DESCRIPCIÓN DE LA CLASE BGPnode

A continuación vamos estudiar con detalle cada uno de los componentes de la clase BGPnode.

5.1.1.Variables miembro

unsigned short int as

Esta variable almacena el identificador del sistema autónomo al que pertenece un objeto BGPnode concreto. Como ya vimos en la sección dedicada al fichero *libr.h*, se trata de una variable *unsigned* porque no tiene sentido que tengamos identificadores de sistema autónomo negativos y *short int* para que ocupe 2 bytes (tomando valores

entre 0 y 65535). Cada nodo BGP de la simulación va a tener un identificador de sistema autónomo único que lo diferencia de los demás. Además, en esta simulación no se considera (por simplicidad) que dos nodos BGP puedan pertenecer al mismo sistema autónomo, ya que ello implicaría la implementación de comunicación intradominio.

unsigned short int hold_time

Esta variable también se ha tratado en *libr.h*, en la sección que trata dicho fichero se explica el por qué de su tipo, por lo que en esta sección se tratará la semántica.

La variable *hold_time* almacena el valor que por defecto se asigna al *Hold Timer* de cada una de las conexiones que se establezcan en un nodo BGP, es decir, que inicialmente tiene cada conexión y que se compara con el que tiene el otro extremo de la conexión, de modo que finalmente y para esa conexión concreta se adoptará el menor de los valores propuestos por cada extremo de la conexión.

ip ide

Se trata de un *struct* de tipo *ip* definido en *libr.h* donde se almacena la dirección IP de un nodo BGP concreto.

vector <rib> loc_rib

Se trata de un vector donde se almacenan las *rib* de las rutas que se consideran válidas para el encaminamiento y que son el resultado del proceso de decisión. Es, por tanto, el equivalente a la *Loc_RIB* descrita por la **RFC 1771**. Se irá actualizando conforme los nodos BGP pares al considerado vayan anunciando nuevas rutas o bien conforme haya rutas que dejen de ser válidas (*withdrawn*), ya sea porque un nodo vecino nos lo comunique o porque lo detecte el propio nodo.

Para almacenar las *rib* válidas se ha escogido un vector debido a su naturaleza dinámica, ya que se puede ir añadiendo o quitando elementos sin estar limitados por un tamaño prefijado, y porque además podemos acceder a los distintos miembros sin necesidad de que este acceso sea ordenado (caso de las listas).

vector <rib> adj_rib_in

Se trata del vector donde se almacenan todas las rutas que un nodo BGP aprende a través de sus nodos BGP vecinos. Es, por tanto, el equivalente a la *Adj_RIB_In* descrita por la **RFC 1771**. Se actualizará conforme los nodos BGP vecinos vayan anunciando nuevas rutas mediante mensajes UPDATE. También se eliminarán de éste aquellas rutas que se hayan anunciado como *withdrawn* o bien aquellas que el propio nodo considere como no factibles. De este conjunto de rutas saldrán aquellas que conforman la Loc_RIB tras la aplicación del proceso de decisión de rutas.

vector <links> connections

Se trata de un vector donde se tiene registrados a todos los gestores de conexión asociados al Nodo BGP considerado y el identificador de la conexión que gestiona cada uno de ellos. Para ello se ha definido el *struct links*, éste tiene dos miembros, el primero de ellos es un entero que contiene el identificador de la conexión

gestionada y el segundo miembro es un puntero al gestor de la conexión identificada. La utilidad de este vector es el poder acceder a un gestor de conexión concreto a partir del identificador de la conexión que gestiona. Son los propios gestores de conexión quienes se registran directamente en este vector.

int version_rib

Esta variable miembro contiene la versión de la Loc_RIB actual. Inicialmente su valor es cero y se va incrementando conforme Loc_RIB va actualizándose. No se tendrá una primera versión de Loc_RIB (versión 0) hasta que el nodo BGP establezca una conexión con un nodo BGP vecino.

int id_to_crash

Esta variable almacena el identificador de una conexión del nodo BGP que se desea que caiga en un instante determinado de la simulación. Es decir, se emplea para la simulación de la caída de un enlace (dicho enlace está identificado por *id_to_crash*) a través del método *BGPnode::crashLink*.

bool stopped_

Se trata de una variable booleana que indica si el nodo está o no activo. Si esta variable está a *true* el nodo BGP estará inactivo aunque reciba mensajes de otros nodos, es por esto por lo que se usa para simular la caída de un nodo. Dada su naturaleza crítica para el correcto funcionamiento de un nodo BGP se ha definido como variable privada y su modificación se realiza mediante los métodos *BGPnode::startNode* (para poner a *true*) y *BGPnode::stopNode* (para poner a *false*). Cuando se genera un objeto *BGPnode* su estado es inactivo y hay que hacer una llamada explícita al método *BGPnode::start* para pasar el nodo a estado activo en un instante concreto de la simulación; este método encola en el manejador de eventos una llamada a *startNode* en dicho instante.

double * p_terrorTx

Este puntero a *double* se utiliza para acceder al instante de tiempo en que se producirá un error en transmisión, si es que se desea simular dicho error. Dicho instante de tiempo es, lógicamente, de tipo *double* como todos los valores de tiempo que se manejan en el simulador. El uso de esta variable está íntimamente ligado al uso de la variable *p_errorTx* y se explicará cuando se trate el error en transmisión.

bool * p_errorTx

Este puntero a *bool* se usa para saber si en la simulación se producirá o no error en transmisión, es decir, se emplea para saber si habrá que simular un error en transmisión. Como ya se ha indicado, su uso está muy ligado a la variable *p_terrorTx*, y se comentará más adelante.

5.1.2.Métodos

BGPnode::BGPnode(int n_as, int hold_t, ip dir, bool * pb, double * pd, double rbin)

El constructor de la clase *BGPnode* recibe seis parámetros. El primero de ellos es el número de Sistema Autónomo al que pertenece el nuevo nodo, el segundo es el valor del *Hold Timer* que se usará por defecto, el tercer parámetro es un *struct* de tipo *ip* que contiene la dirección IP del nodo. El cuarto parámetro es un puntero a una variable booleana, es un paso de parámetro por referencia en vez de por valor, que indica al nuevo nodo si está habilitado el error en transmisión. El quinto parámetro es un puntero a double, de nuevo un paso de parámetro por referencia, que indica cuándo se producirá el error en transmisión en el caso de estar habilitado, es decir, en el caso de que la variable booleana apuntada por el cuarto parámetro sea *TRUE*. El sexto y último parámetro es el valor del régimen binario (en bits por segundo) al que trabaja el nodo y que por defecto valdrá 64000 bps.

El constructor lleva acabo la inicialización de los parámetros del nuevo objeto, esta inicialización consiste en:

- Fijar la variable miembro *stopped_* a *TRUE*, con esto tenemos que inicialmente el nodo estará inactivo.
- Fijar el valor del *Hold Timer* por defecto.
- Fijar la dirección IP del nodo y el AS al que pertenece.
- Apuntar las variables miembro *p_terrorTx* y *p_errorTx* a las direcciones de las variables de tipo *double* y *bool* que contienen información sobre el error en transmisión.

Por otra parte se llama al constructor de la clase padre (*Module*) pasándole como parámetro el régimen binario.

Finalmente se genera un *struct* de tipo *rib*, *to_store*, donde el campo *nlri* es el del propio nodo (la dirección IP del nodo y prefijo de 24 bits) y los *path attributes* siguientes: el campo *origin* a 0, indicando que la ruta es de origen local, el campo *as_path* sólo con el propio (la ruta comienza en este AS), el campo *next_hop* con la dirección IP del nodo, y el campo *local_pref* puesto a su valor máximo (0xffffffff = 65535) ya que su preferencia por esta ruta a sí mismo es máxima. Esta *rib* (*to_store*) se almacena en la Loc_RIB del nodo. Esto se hace porque el propio nodo está inmerso en un sistema autónomo, y como tal debe almacenarlo en su Loc_RIB para tener conciencia de su propio AS, de este modo la ruta a sí mismo (al propio AS) propuesta por él siempre estará en Loc_RIB, es decir, ninguna ruta que nos anuncien nodos vecinos hacia el propio Sistema Autónomo llegará a almacenarse en Loc_RIB. Además, al establecer una conexión con un nodo par se le envía todas y cada una de las entradas en Loc_RIB, al recibir esta ruta el nodo par (vecino) tomará esta ruta hacia nosotros como la mejor ya que sólo hay un salto. También y como medida de precaución se limpia Adj_RIB_In llamando al método *adj_rib_in.clear()*. Como podemos observar, la ruta generada en el constructor y que hace referencia al AS propio no se almacena Adj_RIB_In, ya que al ser una ruta conocida localmente (no proviene de ningún nodo vecino) no pasa por ningún proceso de decisión.

void BGPnode::sendOpen(double now, unsigned short int h_t, int conn, MessageOpen * pp)

Este metodo se activa cuando hay que enviar un mensaje de tipo OPEN y es llamado por el gestor de conexión que desea enviar dicho mensaje. Los parámetros que se le pasan son: en primer lugar el instante (*now*) en que comienza el envío del mensaje (este comienzo es virtual, ya que el envío efectivo se realiza en el instante indicado en la llamada al método *send(…)* y que se verá más adelante), en segundo lugar el valor del *Hold Timer*, que irá dentro del mensaje OPEN, en tercer lugar el

identificador de la conexión a la que está asociado el mensaje y finalmente un puntero al objeto *MessageOpen* que tiene el gestor de conexión para no perder la información contenida en el mensaje.

Lo primero que se hace es generar la cabecera del mensaje, con el campo *marker* (*marker1*, *marker2*, *marker3* y *marker4*) al valor 0xffffffff (todo a '1' según la **RFC 1771**), si se da la condición de error en transmisión se introduce un error en el campo *marker* de la cabecera (0xffaaffff) y se resetea la condición de error. El campo *type* a 1 (indicando que se trata de un mensaje OPEN) y el campo de longitud igual al tamaño (en bytes) de la cabecera (*sizeof(header)*) más el tamaño de los datos (*sizeof(data_open)*). Posteriormente se genera, con la cabecera anterior y con los datos de nuestro nodo, un mensaje OPEN (llamada al constructor de la clase *MessageOpen*). El nuevo objeto *MessageOpen* (*msg*) se almacena en el gestor de conexión mediante la instrucción *"*pp = msg;*" para que permanezca la información cuando se recupere el mensaje en el receptor, y se hace que un puntero a la clase *Message* (*pms*) apunte también al objeto del gestor de conexión, este puntero se usará para hacer efectivo el envío a través del método *send(…)* heredado de *Module*. Pero antes de llamar a *send(…)* se fijan los valores necesarios para que el envío se haga correctamente usando la librería *simul*, es decir, se fija el origen con "pms->setOrigin(this)" (de este modo el "módulo" receptor sabe quién le envía el mensaje y lo acepta en el caso de que haya una conexión entre ellos), la conexión a la que pertenece el mensaje con "*pms->setConnId(conn)*" y el sentido de envío con "*pms->toDown()*". Estos tres métodos son métodos heredados por la clase *Message* de la clase *Packet* (librería *simul*). Una vez preparado el mensaje se envía con el método *send(…)*. Este método recibe como parámetros el instante de envío, que será el instante actual más el tiempo que se tarda en transmitir el mensaje (*timeToTx(pms->size())*), que a su vez depende del régimen binario al que opera el nodo, y el puntero a Message, que se convierte automáticamente en puntero a *Packet*, que es con lo que opera *send(…)*.

void BGPnode::sendNotif(double now, char err, char err_subc, void * pd, int conn, MessageNotif * pn)

Este método se activa cuando un gestor de conexión desea enviar un mensaje de tipo NOTIFICATION. Este método recibe seis parámetros; el primero de ellos es el instante en que comenzará el envío del mensaje, los dos siguientes parámetros son el código y subcódigo de error; el cuarto parámetro nos permite enviar cualquier tipo de información adicional sobre el error que se está comunicando en el mensaje, aunque en este simulador no se hace uso de este campo adicional, pero se contempla para futuras ampliaciones. El quinto parámetro es el identificador de la conexión a la que pertenece el mensaje. El sexto y último parámetro es un puntero a un objeto de la clase *MessageNotif* (paso de parámetro por referencia) que se aloja en el gestor de conexión que hace la llamada para evitar la pérdida de información en la recepción del mensaje.

Lo primero que se hace en este método es generar la cabecera del mensaje, para ello se le da al campo *marker* (*marker1*, *marker2*, *marker3* y *marker4*) el valor 0xaaaaaaaa. La **RFC 1771** no especifica el valor de este campo para mensajes NOTIFICATION, sólo indica que se debe usar para autenticación o bien para detectar posibles pérdidas de sincronismo; en este caso se ha tomado el valor 0xaaaaaaaa más por sincronismo que por autenticación, pues la codificación binaria de 0xaa es 10101010. Además, como se verá más adelante, el error en transmisión consiste en una pérdida de sincronismo. Al campo *type* de la cabecera se le asigna el valor 2,

indicando que se trata de un mensaje de tipo NOTIFICATION. Finalmente el campo *length* toma el valor del tamaño de la cabecera más el tamaño de los datos.

En segundo lugar se genera un objeto de la clase *MessageNotif* (*msg*), con esto ya tenemos el mensaje NOTIFICATION que se va a enviar. Tras generar el mensaje a enviar se procede del mismo modo que en *BGPnode::sendOpen(…)*, es decir, se almacena el mensaje en el gestor de conexión (utilizando el puntero *pn*), se prepara para ser enviado (usando el puntero a *Message*, *pms*, y los métodos *setOrigin*, *setConnId* y *toDown*) y se envía haciendo la correspondiente llamada al método *send(…)* de igual modo que en el método anterior.

void BGPnode::sendUpd(double now, data_update * upd, int conn, MessageUpdate * pu)

Este método se activa cuando un gestor de conexión desea enviar un mensaje UPDATE. Recibe cuatro parámetros, el primero consiste en el instante de tiempo (de la simulación) en el que se desea que comience el envío del mensaje. El segundo parámetro es un puntero a un *struct* de tipo *data_update*, ya que por simplicidad el campo de datos del mensaje estará generado previamente a la llamada a *sendUpd(…)*. El tercer parámetro es el identificador de la conexión a la que pertenece el mensaje UPDATE que se va a enviar. El cuarto parámetro es un puntero al objeto de la clase *MessageUpdate* que tiene el gestor de conexión para evitar la pérdida de la información del mensaje en el receptor.

Como en los casos anteriores, lo primero será generar la cabecera. Si se da la condición de error en transmisión se introduce un error de sincronismo en el campo *marker* de la cabecera (0xaaffaaaa) y se resetea la condición de error, si no se cumple dicha condición el campo marker será el mismo que en los mensajes NOTIFICATION (0xaaaaaaaa). El campo *type* toma el valor 4, indicando que se trata de un mensaje UPDATE. Al campo *length* se le asigna la longitud de la cabecera más la longitud del campo de datos. Lo siguiente será generar el mensaje UPDATE (*msg*) a partir de la cabecera y el campo de datos (que nos ha sido pasado por referencia). Una vez tenemos el mensaje, lo almacenamos en el objeto *MessageUpdate* del gestor de conexión llamante para evitar pérdida de información en el receptor, se prepara para el envío y se envía con la llamada a *send(…)*.

void BGPnode::sendKeep(double now, int conn, Message * pm)

Este metodo se activa para enviar mensajes KEEPALIVE en una determinada conexión. El primero de los tres parámetros que recibe es el instante de tiempo en que comenzará el envío del mensaje KEEPALIVE; el segundo es el identificador de la conexión a la que pertenece el mensaje KEEPALIVE y el tercero es un puntero al objetos *Message* del gestor de conexión que hace la petición de envío para evitar la pérdida de información en el receptor.

El primer paso en este método es la generación de la cabecera, que se desarrolla de igual modo que en *BGPnode::sendUpd(…)*. Lo siguiente será generar el mensaje KEEPALIVE, que como ya se ha visto sólo contiene la cabecera, con lo que nos basta con generar un objeto de la clase *Message*. Finalmente y como hemos visto en los anteriores métodos de envío de mensajes, se almacena el mensaje en el objeto *Message* del gestor de conexión que ha invocado a este método, se prepara para su transmisión y se envía usando el método *send(…)*.

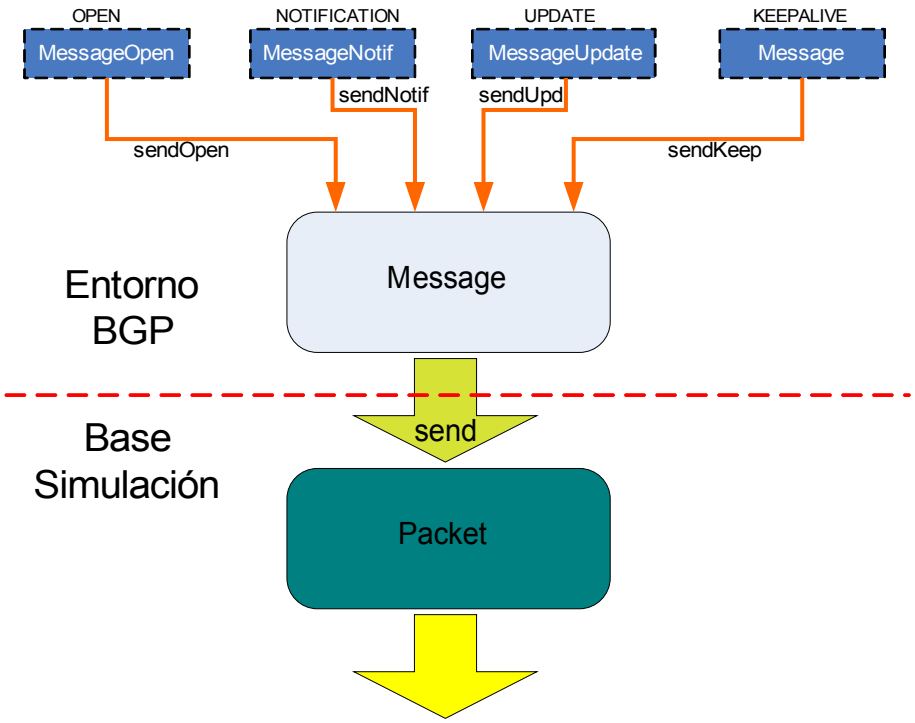

Figura 11. Envío de mensajes. Encapsulación.

Como vemos en la figura anterior, en los métodos de envío de los distintos tipos de mensajes BGP (OPEN, NOTIFICATION, UPDATE y KEEPALIVE) los objetos de las distintas clases que representan estos mensajes, se "encapsulan" en la clase *Message* (se hace que un puntero a la clase padre los apunte) para después volver a "encapsular" en la clase *Packet* y poder enviar los mensajes utilizando la Base de Simulación, que no consiste sino en la librería *simul*. Con esto conseguimos el envío de mensajes en el entorno BGP de modo transparente a como se haga en la Base de Simulación sin más que preparar levemente el mensaje y llamar al método *send(…)*.

void BGPnode::recvUp(double now, Packet * msg)

Este método se activa cuando un nodo BGP recibe un mensaje (un paquete de datos). Es un método heredado de la clase *Module* que ha sido sobrescrito para la clase *BGPnode*. El método recibe dos parámetros, el primero de ellos es el instante en que se recibe el mensaje y el segundo es un puntero a *Packet* de acuerdo a su definición, que es donde se encuentra el mensaje.

Lo primero que se hace en este método es comprobar que el nodo no esté inactivo, para ello se comprueba la variable miembro *stopped_*, de modo que si está inactivo (*stopped_* = *TRUE*) se ignora el mensaje entrante, con esto simulamos un nodo inactivo y por extensión un nodo que se ha caído, provocando que en el extremo transmisor venza el *Hold Timer* y se aplique todo el proceso de deshabilitación de rutas. Una vez comprobado que el nodo está activo, se hace que un puntero a *Message* (*pmsg*) apunte a la misma dirección que *msg* (puntero a *Packet*), con lo que podemos acceder a la cabecera del mensaje recibido para discriminar el tipo de mensaje de que se trata (comprobando el valor de *pmsg->h_.type*). En función del valor del campo *type* de la cabecera del mensaje (1, 2, 3 ó 4) tendremos un tipo distinto de mensaje (OPEN, NOTIFICATION, KEEPALIVE o UPDATE respectivamente), y según esto llamaremos al método apropiado para realizar un

análisis del mensaje recibido (*analyseOpen*, *analyseNotif*, *analyseKeep* o *analyseUpd*). Si el valor de *type* no se corresponde con ninguno de los mensajes definidos por la **RFC 1771** el mensaje se descarta, es como si se hubiese perdido el mensaje.

void BGPnode::analyseOpen(double now, Packet * pmsg)

Este metodo es llamado por *BGPnode::recvUp(…)* una vez éste ha determinado que el mensaje entrante es de tipo OPEN. Recibe dos parámetros que se describen a continuación: el primero de ellos, *now*, indica el instante en que se recibe el mensaje, el segundo, *pmsg*, es un puntero a *Packet* y apunta al contenido del paquete de datos recibido (que no es sino un mensaje de tipo OPEN).

Lo primero que se hace en este método es hacer que un puntero a *MessageOpen*, *pms*, apunte al paquete de datos, con esto conseguimos tener acceso al contenido del mensaje OPEN. En segundo lugar se obtiene el identificador de conexión del mensaje (llamada al método *MessageOpen::connId()* heredado de la clase *Packet* y que nos devuelve un entero con el identificador que almacenamos en *conn*) y con éste buscamos al gestor de conexión que interpretará el contenido del mensaje recibido, mediante el uso del método *BGPnode::findgestconn(conn)* que nos devuelve una referencia a dicho gestor de conexión. Seguidamente accedemos a la cabecera del mensaje para comprobar la integridad de ésta, es decir, para verificar que no se ha producido ningún error en transmisión. Una vez comprobada la integridad de la cabecera (*auten* = *true*) se imprimen por pantalla los parámetros del mensaje y se extrae el contenido del campo de datos mediante el uso de un *struct data_open* auxiliar, *aux*, que junto con *now* se pasan como parámetros del método *gestConn::initResp(…)*, que como hemos visto se encarga de analizar los datos y actuar en consecuencia. Si al comprobar la integridad de la cabecera se detecta un error, se deberá enviar un mensaje NOTIFICATION al nodo transmisor y cerrar la conexión con dicho nodo, además de activar el proceso de deshabilitación de rutas, para ello se llama al método *gestConn::notification(…)* del objeto *gestConn* que gestiona la conexión considerada y cuya referencia hemos obtenido antes.

void BGPnode::analyseNotif(double now, Packet * pmsg)

Este metodo es llamado por *BGPnode::recvUp(…)* una vez éste ha determinado que el mensaje entrante es de tipo NOTIFICATION. Recibe dos parámetros que se describen a continuación: el primero de ellos, *now*, indica el instante en que se recibe el mensaje, el segundo, *pmsg*, es un puntero a *Packet* y apunta al contenido del paquete de datos recibido (que no es sino un mensaje de tipo NOTIFICATION).

Lo primero que se hace en este método es hacer que un puntero a *MessageNotif*, *pms*, apunte al paquete de datos, con esto conseguimos tener acceso al contenido del mensaje NOTIFICATION. Seguidamente se obtiene el identificador de conexión del mensaje (llamada a *MessageNotif::connId()*) para así tener una referencia al gestor de conexión que interpretará el contenido del mensaje recibido, mediante el uso del método *BGPnode::findgestconn(conn)*, aunque antes de esto se imprime por pantalla la notificación de haber recibido un mensaje NOTIFICATION, su código de error y el instante en que se ha recibido. Una vez se tiene la referencia al gestor de la conexión considerada se llama al método *gestConn::notifResp(…)* para que aplique el proceso ya explicado que debe seguirse en el caso de recibir un mensaje de tipo NOTIFICATION.

En este método no se ha creído conveniente hacer la comprobación de la integridad de cabecera, aunque igualmente se podría, por simplicidad, ya que eventualmente conduce a la misma situación, es decir, si el mensaje tuviese errores en la cabecera se enviaría un mensaje NOTIFICATION y se cerraría la conexión por este extremo y en el extremo remoto se ignoraría dicho mensaje NOTIFICATION, pues la conexión fue cerrada tras enviar el mensaje NOTIFICATION con errores de transmisión.

void BGPnode::analyseKeep(double now, Packet * pmsg)

Este metodo es llamado por *BGPnode::recvUp(…)* una vez éste ha determinado que el mensaje recibido es de tipo KEEPALIVE. Recibe como parámetros los dos siguientes: *double now*, que indica el instante en que se recibe el mensaje, y *Packet * pmsg*, que se trata de un puntero a *Packet* y que apunta al contenido del paquete de datos recibido (que es de tipo KEEPALIVE).

El primer paso es hacer que un puntero *Message*, *pms*, apunte al paquete de datos recibido (apuntado previamente por *pmsg*), de este modo tendremos acceso a la cabecera, que, como ya se ha visto, en el caso de mensajes KEEPALIVE es el único contenido. Seguidamente se obtiene la referencia al gestor de la conexión a la que pertenece el mensaje de igual modo que en los casos anteriores. Tras esto se comprueba la integridad de cabecera y si todo es correcto se notifica por pantalla la llegada de un mensaje KEEPALIVE y se llama al método *gestConn::KeepResp(…)* para que el gestor de conexión opere en consecuencia. Si se detectase algún error en la cabecera se operaría de igual modo a como se ha descrito en *BGPnode::analyseOpen(…)*.

void BGPnode::analyseUpd(double now, Packet * pmsg)

Este método es llamado por *BGPnode::recvUp(…)* una vez este último ha determinado que el mensaje recibido es de tipo UPDATE. Se recibe dos parámetros: el primero de ellos, *double now*, indica el instante en que se recibe el mensaje, y *Packet * pmsg*, un puntero a *Packet* que apunta al contenido del paquete de datos recibido (que, en este caso, es de tipo UPDATE).

En primer lugar se hace que un puntero a *MessageUpdate*, *pms*, apunte al paquete de datos, con lo que tendremos acceso a los campos del mensaje UPDATE. Una vez tenemos garantizado el acceso a los datos del mensaje obtenemos, mediante una llamada a *BGPnode::findgestconn(…)*, la referencia al gestor de conexión al que pertenece el mensaje. Tras prepararnos para procesar el mensaje lo siguiente será comprobar la integridad de la cabecera, de modo que si se verifica la integridad de ésta pasaremos al procesamiento de los datos.

El procesado de los datos se desarrollará como se describe a continuación: lo primero será extraer los datos efectivos del mensaje, para ello se utiliza un *struct data_update* (*data*) auxiliar, tras esto se comprueba si en el mensaje se están anunciando rutas que han dejado de ser válidas (*withdrawn*), para ello se analiza el campo *data.unf_route_l* (longitud de rutas no válidas) de modo que si es distinto de cero se aplicará el proceso de deshabilitación de rutas con la llamada al método *BGPnode::withdrProcess(…)*, este método está en *BGPnode* y no en *gestConn* porque su aplicación afecta igualmente a todas las conexiones del nodo, y no sólo a la que recibe el mensaje UPDATE. Una vez terminado el procesado de rutas no válidas pasamos al procesado de la ruta anunciada (en el caso de que el mensaje UPDATE

contenga una nueva ruta); para saber si hay nueva ruta se comprueba el campo *data.tot_PA_l* (Total Path Attributes Length), de modo que si es no nulo indica que efectivamente hay una nueva ruta, con lo cual se almacenan los campos de esta ruta en *to_input* (*rib*), aunque previamente se actualizan los siguientes campos: *nuevo* se pone a *true*, de este modo el proceso de decisión sabe que procesa una ruta nueva, y no una que ha dejado de ser válida, el campo *local_pref* (preferencia local por la ruta recibida) se decrementa, ya que el valor recibido es la preferencia local del nodo vecino y al seguir un paso en su segregación debe decrementar su valor en uno antes de su análisis, por último el campo *origin* se pone a '2', con lo que sabremos que se trata de una ruta aprendida externamente a nuestro sistema autónomo, el resto de campos de la ruta se almacenan sin alterarse. Tras actualizar y almacenar los datos de la ruta se comprueba que ésta no tenga bucles con un análisis de *to_input.pta.as_path* (*AS_Path*), de esta comprobación se encarga el método *BGPnode::check_loop(…)*, que devuelve un valor booleano que almacenamos en la variable *no_loop* y que será *true* si se corrobora que no hay bucles en la ruta, si los hubiese, se descartaría la ruta anunciada. Comprobada ya la inexistencia de bucles en la nueva ruta, el siguiente paso es comprobar que la ruta no esté ya almacenada en *Adj_RIB_In*, ya que si estuviese, su procesado resulta innecesario y redundante, pues al estar ya almacenada indica que ya ha sido procesada. Si finalmente se comprueba que la ruta es nueva, se aplicará el proceso de decisión sobre la ruta con la llamada a *BGPnode::decisionProcess(…)* e independientemente y tras proceso de decisión se almacenará la ruta en *Adj_RIB_In*, pues aunque no se pase la ruta a *Loc_RIB*, en un futuro, al eliminar una ruta, puede ser una alternativa válida, el almacenamiento en *Adj_RIB_In* es posterior al proceso de decisión porque éste compara la ruta con las que ya hay almacenadas en *Adj_RIB_In*, con lo que se compararían dos rutas idénticas y no llegaría a *Loc_RIB*, pues el proceso de decisión asume que si hay una ruta igual en *Adj_RIB_In* esta última ya habría sido almacenada en *Loc_RIB*, y no la almacena.

Recapitulando, vemos que previo al proceso de decisión se comprueba que la ruta no tenga bucles y que no se tenga conocimiento previo de ella, y que además se guardará en *Adj_RIB_In* como una alternativa de futuro a una ruta que deje de ser válida.

Tras todo este tedioso proceso se hace la llamada al gestor de conexión para que prosiga con la comunicación con el nodo vecino (*gestConn::keepResp(…)*).

Por supuesto y como hemos visto en los anteriores métodos de análisis, si la cabecera contiene errores (error en transmisión) se llamará al método *gestConn::notification(…)* para cerrar la conexión.

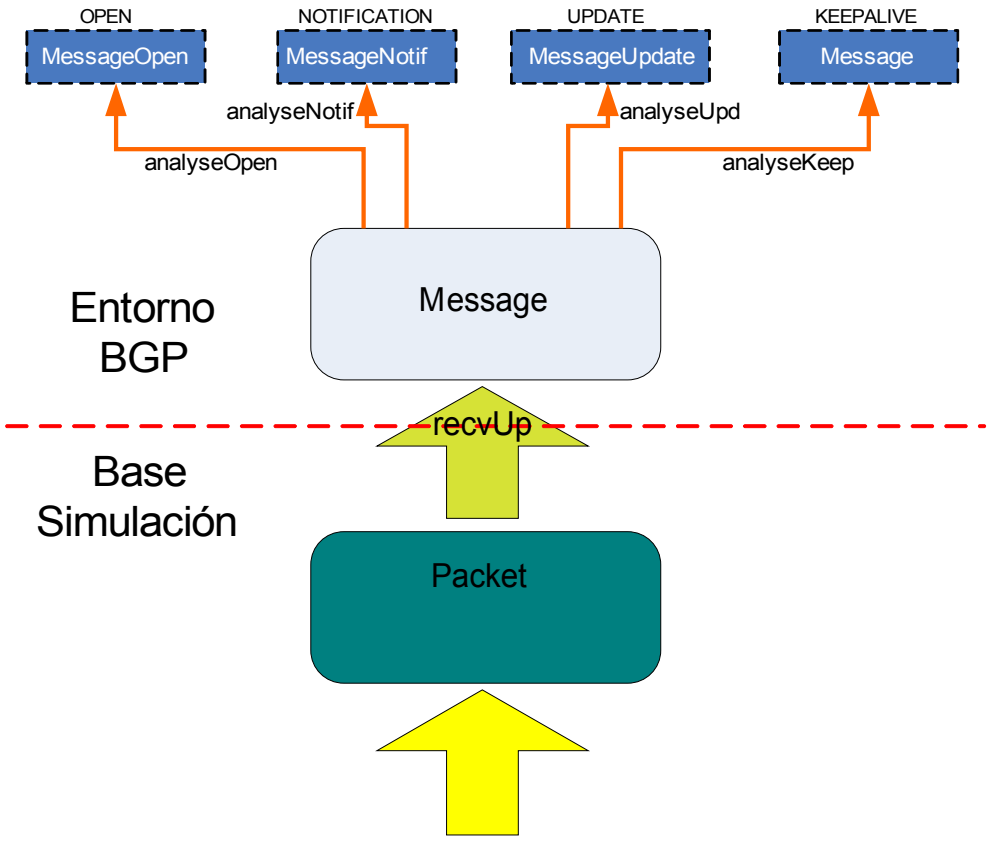

Figura 12. Análisis de mensajes. Desencapsulación.

Como podemos observar en la figura anterior, todos los mensajes BGP se reciben encapsulados en la clase *Packet*. Esto es debido a que el simulador BGP se ha desarrollado de modo que se pueda emplear lo ya desarrollado en la Base de Simulación, que trabaja con *Packet*. Para saber de qué tipo de mensaje se trata se usa un puntero a *Message*, pues con éste podemos acceder directamente a la cabecera, de donde determinamos el tipo. Una vez determinado el tipo de mensaje, ya podemos analizar los datos sin más que hacer que un puntero al tipo de mensaje concreto apunte al mensaje.

void BGPnode::start(double tinit)

Este método se emplea para activar un nodo BGP. Recibe como parámetro un *double* que indica el instante en que debe quedar activo dicho nodo. Lo que hace este método es generar el evento para que el nodo se active en *tinit*, para lo cual se hace una llamada a *wakeUp(…)* (heredado de *Module*) pasándole como parámetros *tinit* y la referencia al método *BGPnode::startNode(…)*. Este método se llama desde el *main* de la aplicación.

void BGPnode::startNode(double now, Packet *)

Este metodo es el que, efectivamente, activa el nodo. Para ello simplemente pone a *false* la variable miembro *stopped_*, permitiendo el envío y recepción de mensajes. Vemos que este método recibe dos parámetros y que en ningún caso se usan, esto se hace así por transparencia con el método *Module::wakeUp*, que como está definido sólo permite el almacenamiento de llamadas a métodos (eventos) con estos parámetros.

void BGPnode::stop(double tstop)

Este método se emplea para desactivar un nodo BGP. Recibe como parámetro un *double* que indica el instante en que debe quedar inactivo dicho nodo. Lo que hace este método es generar el evento para que el nodo se desactive en *tstop*, para lo cual se hace una llamada a *wakeUp(…)* (heredado de *Module*) pasándole como parámetros *tstop* y la referencia al método *BGPnode::stopNode(…)*. Este método se llama desde el *main* de la aplicación.

void BGPnode::stopNode(double now, Packet *)

Este metodo es el que, efectivamente, desactiva el nodo. Para ello se da el valor *true* a la variable miembro *stopped_*, deshabilitando el envío y recepción de mensajes. Vemos que este método recibe dos parámetros y que en ningún caso se usan, la justificación es similar al caso de *BGPnode::startNode(…)*. Además, se obtiene una "fotografía" de la visión que tiene del sistema el nodo BGP considerado mediante una llamada al método *BGPnode::printLRIB(now)*. De este modo tenemos la visión final del sistema para el nodo BGP considerado. Este método se emplea también para simular la caída de un nodo BGP, ya que desactiva la transmisión y recepción de mensajes a partir del instante indicado. Para la transmisión, realmente desactiva la programación del envío de mensajes, es decir, la programación de futuros eventos de envío de mensajes; si antes del instante indicado se ha "programado" enviar un mensaje en un instante posterior al indicado por el parámetro *now*, este mensaje se enviará porque ya está almacenado en la cola de eventos del manejador de eventos.

void BGPnode::stop_link(double tslink, int id)

Este método permite la desactivación de una conexión del nodo BGP para emular la caída de un enlace. Recibe como parámetros el instante en que se desea que ocurra (*tslink*) y el identificador de la conexión que quedará inactiva (*id*). Lo primero que se hará es fijar la variable miembro *id_to_crash* al valor de *id*, con lo que ya sabemos cuál será la conexión que "caerá". Después se generará un evento para que en el instante tslink se active el método *BGPnode::crashLink(…)*, haciendo efectiva la inhabilitación de la conexión. Como vemos, el evento se genera también a través de *wakeUp(…)*, pero esto hace que necesitemos la variable miembro *id_to_crash* para saber cuál es la conexión que debe caer (pues se trata de un *int*) y además nos restringe a que sólo caerá una conexión por nodo BGP. Este método se llama desde el *main* de la aplicación.

void BGPnode::crashLink(double now, Packet *)

Este método es el que, efectivamente, hace que una conexión de un nodo BGP quede desactivada, con lo que se consigue simular la caída de una conexión. El funcionamiento de este método está ligado a la variable miembro *id_to_crash*, de modo que lo que se hace es consultar su valor para saber cuál es el identificador de la conexión que quedará inhabilitada, obtener una referencia al gestor de conexión identificado y una vez obtenida dicha referencia se fuerza el valor del estado (variable *gestConn.status_*) del gestor de conexión a '0' (estado IDLE). Este método recibe dos parámetros que, como en los casos previos, no se usan; la justificación es la misma que en dichos casos.

void BGPnode::link(int id_con)

Este método se utiliza para introducir una conexión (mediante su identificador) en el conjunto de conexiones válidas de un nodo BGP concreto. Este método es vital para la correcta comunicación entre nodos, ya que si un nodo trata de enviar un paquete de datos cuyo identificador no se encuentra entre las conexiones válidas, dicho paquete es ignorado (no se envía), pues el método *send(…)* comprueba el conjunto de conexiones válidas. Lo que hace es introducir el identificador, *id_con*, en el conjunto (*set*) *conns_* heredado de la clase *Module*, a través del método *insert(id_con)*, también heredado de *Module*.

void BGPnode::check_ht(double now, Packet * p)

Este método sirve para comprobar que en cada una de las conexiones activas del nodo BGP no haya expirado el *Hold Timer*. Este método recibe dos parámetros; el primero de estos parámetros es un *double*, *now*, que indica el instante en que se hará la comprobación. El segundo de estos parámetros, *Packet * p*, no se usa más que por compatibilidad con las librerías del simulador (el valor que le dará quién lo llame será *null*), ya que este método se llama como un evento a través de *wakeUp(…)*.

Lo primero que se hace es obtener una referencia al inicio del vector de conexiones del nodo BGP (*vector <links>::iterator index = connections.begin()*), una vez tenemos la referencia se recorre dicho vector desde el principio hasta el final, de modo que para aquellas conexiones que estén en estado '3' (ESTABLISHED) y siempre que el nodo esté activo (!stopped_) se comprueba que la diferencia entre el instante último en que se recibió un mensaje (*gestConn.now_old*) y el instante de comprobación (*now*) no supere el valor del *Hold Timer* para la conexión considerada (*gestConn.holdT*). Si en alguna de las conexiones comprobadas se supera el valor de *Hold Time* se llamará al método *notification(…)* del gestor de esa conexión para que cierre la conexión previo envío de mensaje NOTIFICATION con código de error '4' (expiración del *Hold Timer*) y subcódigo '0' (unspecific).

void BGPnode::withdrProcess(list <nlri> wd, int ident, double now)

Este método se utiliza para eliminar las rutas "tachadas" de *withdrawn*, es decir, aquellas rutas que, bien por indicación de un nodo vecino, bien por proceso interno al nodo BGP, han dejado de ser válidas para encaminar tráfico entre sistemas autónomos.

Este método recibe tres parámetros, el primero de ellos, *wd*, es una lista con la información de alcanzabilidad del nivel de red (NLRI) de las rutas que han dejado de ser válidas, en resumen es la lista de aquellas rutas que se van a eliminar. El segundo parámetro, *ident*, es el identificador de la conexión por la que se ha conocido que el conjunto de rutas formado por la lista de *wd* ha dejado de ser válido. El tercer y último parámetro, *now*, indica el instante actual en la simulación, es decir, el instante en que se conoce y elimina los elementos de *wd*.

Las rutas a eliminar (*wd*) se eliminarán de Adj_RIB_In y de Loc_RIB. En primer lugar se eliminarán de Adj_RIB_In, puesto que la eliminación de rutas de Loc_RIB conlleva aplicar un proceso de decisión sobre las rutas almacenadas en Adj_RIB_In y si no han sido eliminadas previamente de Adj_RIB_In se llegaría a resultados absurdos, es decir, se tomaría como mejor ruta a un destino concreto para Loc_RIB una ruta que ha sido declarada como *withdrawn*.

La eliminación de rutas de Adj_RIB_In se desarrolla como se indica a

continuación: se va a recorrer la lista *wd* de principio a fin de modo que cada uno de sus elementos, referenciado por la variable local *i_wd* (*list <nlri>::iterator*), se va a comparar con cada uno de los elementos del vector *adj_rib_in*, referenciados a su vez por *i_ribin* (*vector <rib>::iterator*) mediante la llamada *BGPnode::compare_nlri(*i_wd,i_ribin->stored)* (se compara dos *structs* de tipo *nlri*) de modo que si coinciden y además coincide el identificador de la conexión por la que fueron conocidas (*i_ribin->idcn == ident*), ya que podemos tener el mismo NLRI en dos rutas conocidas por distintas conexiones, entonces la ruta se eliminará del vector *adj_rib_in* y se guardará en una lista auxiliar, *aux* (*list <nlri>*). Vemos que se comprueba que la ruta esté en nuestro vector *adj_rib_in*, esto se hace porque el nodo vecino que nos anuncia sus rutas no válidas nos anuncia todas las rutas que ha eliminado, tanto de Adj_RIB_In como de Loc_RIB, y nuestro nodo sólo conoce aquellas que estaban en su Loc_RIB, así se evita que se intente eliminar una ruta de la que no tenemos conocimiento.

La eliminación de rutas de Loc_RIB es muy parecida a la ya explicada, salvo que si se elimina una ruta, seguidamente se aplica el proceso de decisión para sustituirla. Así pues, de nuevo se recorre la lista *wd* de principio a fin de modo que cada uno de sus elementos se compara con cada uno de los elementos del vector *loc_rib*, referenciados a su vez por *i_lrib* (*vector <rib>::iterator*) mediante la llamada *BGPnode::compare_nlri(*i_wd,i_lrib->stored)* de modo que si coinciden y además coincide el identificador de la conexión por la que fueron conocidas (*i_lrib->idcn == ident*), la ruta se eliminará del vector *loc_rib*. Una vez eliminada la ruta, se llama al proceso de decisión, indicándole que se trata de decidir por una ruta que acaba de ser eliminada (segundo parámetro de la llamada a *false*). Como se puede observar en el código, el recorrido del vector *loc_rib* es distinto al de *adj_rib_in*, en este caso el incrementar el iterador al vector para comprobar el siguiente elemento sin saber si se ha introducido un nuevo elemento *loc_rib* en sustitución del eliminado nos puede llevar a un bucle infinito, por lo que se comprueba antes de incrementar el iterador, de modo que si no se ha introducido un elemento en sustitución, no se incrementa, pues si se incrementase y el elemento eliminado fuese el último del vector nos pasaríamos el fin del vector sin darnos cuenta y nunca llegaríamos al final de éste. Además, si incrementamos el iterador sin haber encontrado una ruta de sustitución (sin ser la última ruta la eliminada), no se comprobarían todos los elementos.

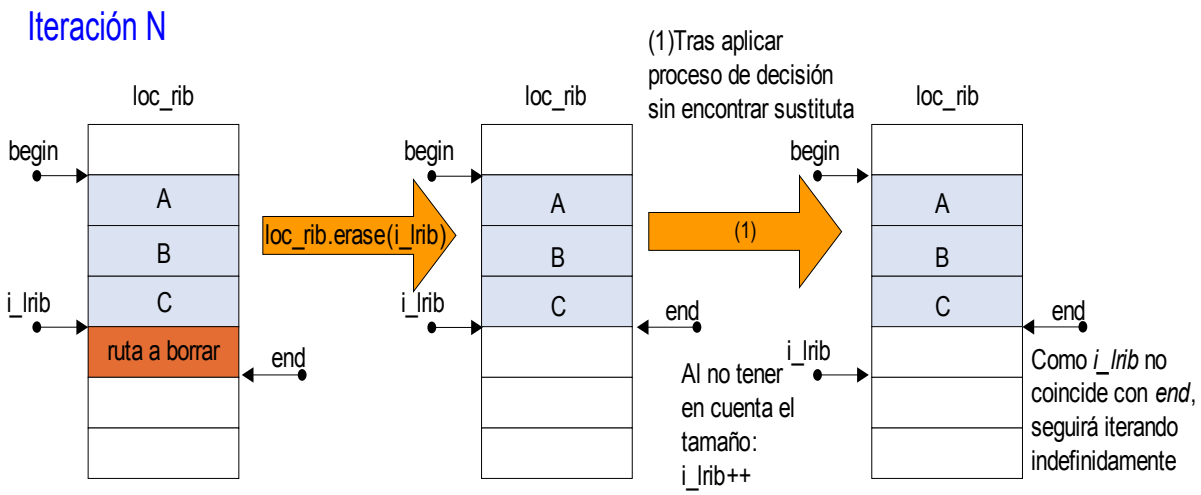

Figura 13. Causa de bucle infinito al eliminar rutas de loc_rib.

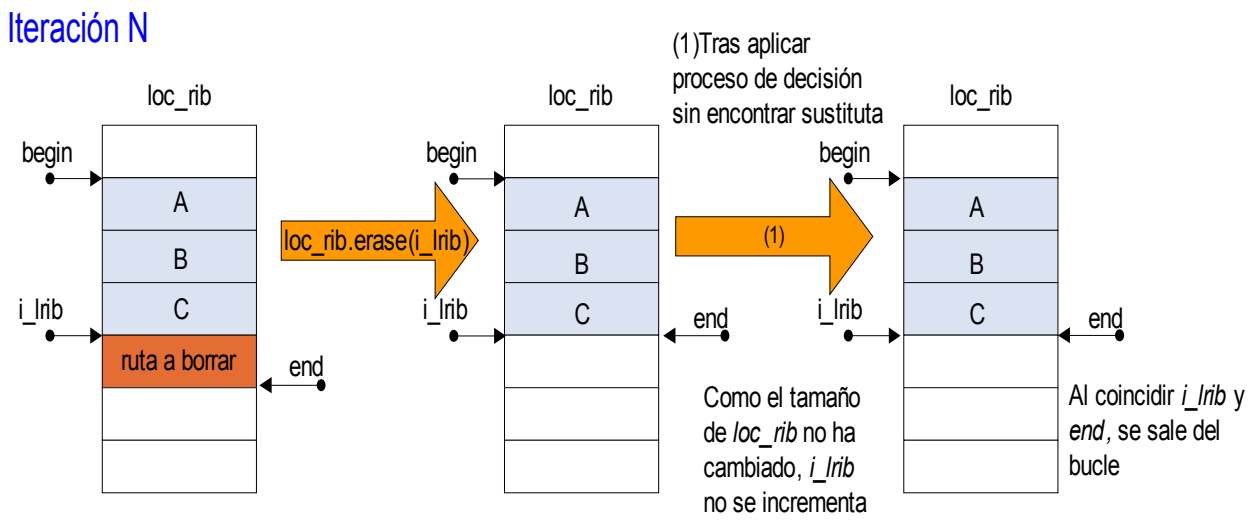

Figura 14. Solución a bucle infinito al eliminar rutas de loc_rib.

Tras la eliminación de rutas de Loc_RIB se hará una "foto" de las RIBs del nodo (se guardan en un fichero, ribN*k*_v*m*.txt) en el caso de que sí se hayan eliminado rutas de Loc_RIB. Con lo que podremos seguir la evolución del sistema considerado. Para saber si ha habido modificaciones se usa la variable booleana local *flag*.

El último paso del proceso de eliminación de rutas consiste en comunicar a todos los gestores de conexión del nodo BGP (salvo el que nos indicó las rutas no válidas, que ya tiene conocimiento de ellas) las rutas que han sido eliminadas, ya que éste es un proceso global al nodo y afecta por igual a todas las conexiones. Para ello se recorre el vector *connections* (*vector <links>*), donde el nodo tiene registrado cada uno de los gestores de conexión de sus conexiones con nodos vecinos, y se introducen las rutas guardadas durante el proceso en *aux* en la lista de conexiones no válidas de cada gestor, *withdrawn*.

void BGPnode::decisionProcess(rib p_rib, bool nueva, double now, int con_origin)

Este método implementa el proceso de decisión de la mejor ruta a un destino concreto. Puede ser invocado por dos motivos; el primero consiste en que se haya conocido a través de una determinada conexión del nodo BGP, es decir, a través de un nodo vecino, una nueva ruta a un determinado destino y, por tanto, haya que ver si existe alguna otra ruta a dicho destino y si es así ver cuál es la mejor. El segundo motivo para la invocación de este método es que se haya decidido eliminar una ruta de Loc_RIB, que habrá sido "tachada" de ruta *withdrawn*, y, por tanto, habrá que buscar de entre las rutas que conocemos y que están almacenadas en Adj_RIB_In una ruta en sustitución de la anterior, si es que conocemos otra ruta al mismo destino que la eliminada. Es por esto que el método está dividido en dos bloques. Bien es cierto que se podría haber implementado de modo independiente dos métodos, uno para cada uno de los casos anteriormente expuestos, pero se ha decidido implementar todo bajo un mismo método porque en ambos casos responde a la búsqueda de la mejor ruta hacia un destino concreto.

El método recibe cuatro parámetros que se describirán a continuación: el primero de ellos *p_rib*, contiene la ruta que vamos a considerar, ya sea por nueva para comparar con las demás, ya sea por tratarse de la ruta eliminada que intentamos sustituir. El segundo de los parámetros recibidos es una variable booleana, *nueva*, que

hace la diferenciación entre ruta nueva (a *true*) y ruta a sustituir (a *false*). El tercer parámetro, *now*, indica el instante actual en la simulación. El cuarto y último parámetro, con_origin, indica la conexión por la que se ha recibido la ruta a la que se va a aplicar el proceso de decisión, de modo que en el caso de ser una nueva ruta válida, no se anunciará al otro extremo de dicha conexión la presencia de la ruta, pues ya la conoce.

El primer bloque se aplica en el caso de que la ruta sea nueva, en este caso habrá que comprobar si ya existe en Adj_RIB_In alguna ruta al mismo destino que la que estamos considerando (todos los destinos conocidos están en Adj_RIB_In), es decir, si ya se tiene conocimiento del destino, de modo que si no hay ninguna otra aparte de la ruta nueva, ésta entrará automáticamente en Loc_RIB. Se tomará, en principio, la nueva ruta como la mejor, almacenándola en la variable *rib* local *elected*, se recorre el vector miembro adj_rib_in desde el principio hasta el final y se va comparando cada uno de sus miembros mediante el método *BGPnode::compare_nlri(…)*, si no se encuentra ninguna coincidencia, la variable booleana local *unica* tendrá el valor *true* al culminar el recorrido de *adj_rib_in*. Si por el contrario la ruta nueva coincide, en destino, con una ruta ya conocida habrá que compararlas para quedarnos con la mejor (además se pone a *false* la variable *unica*). Esta comparación se lleva a cabo del siguiente modo: en primer lugar se compara la preferencia local por cada una de las rutas consideradas, de modo que nos quedaremos con la que tenga mayor preferencia. Si la preferencia local coincide en ambas rutas entonces el siguiente paso será comparar el campo *as_path* de las rutas, más concretamente el último elemento de este campo, es decir, el identificador del último Sistema Autónomo por el que pasó cada una de las rutas antes de llegar al nodo BGP que estamos considerando, de modo que nos quedaremos con aquella ruta en la que dicho valor sea menor. La ruta elegida se almacenará en la variable local *rib elected*. Esta comparación se hará con todos los elementos del vector *adj_rib_in*, de modo que nos iremos quedando, en cada comparación, con la mejor según el criterio descrito y almacenándola en *elected*, con lo que al final del recorrido tendremos la mejor de todas las rutas al destino en cuestión almacenada en *elected*. Una vez decidida la mejor ruta, la ruta inicial (*p_rib*) se almacena en *adj_rib_in*, puesto que ya hemos terminado el proceso de decisión para una ruta nueva. El siguiente paso es guardar la ruta elegida en Loc_RIB, si la ruta era la única al destino (*unica=true*) simplemente hay que introducirla en Loc_RIB (*loc_rib.push_back(elected)*) y luego actualizar los campos *next_hop* (que ahora será nuestro nodo BGP) y as_path (añadiendo el identificador de AS de nuestro nodo BGP) para anunciar la elección de esta nueva ruta a los nodos vecinos mediante mensajes UPDATE. Lo último en este proceso será introducir la nueva ruta elegida en los vectores *adj_rib_out_* de cada gestor de conexión del nodo, de modo que la ruta elegida y actualizada se propague a los nodos BGP vecinos en los anteriormente citados mensajes UPDATE y hacer la "foto" de nuestras RIBs mediante la llamada a *BGPnode::printLRIB(...)*.

Si la ruta nueva no era la única al destino considerado entonces el proceso de almacenamiento cambia, sólo debemos almacenar la ruta en Loc_RIB si la ruta elegida coincide con la que ha iniciado el proceso de decisión, ya que si no coincide con la ruta inicial es porque ya teníamos conocimiento de una ruta mejor, lo cual implica que la ruta elegida ya estaba almacenada en Loc_RIB. Además, el hecho de no ser la única ruta al destino considerado implica que ya existía una ruta a ese destino en Loc_RIB, y puesto que esa ruta "antigua" ya no es la mejor, hay que eliminarla de Loc_RIB, por tanto la buscamos en el vector *loc_rib* y una vez la encontremos la eliminamos e introducimos la nueva. La ruta "antigua" no se debe "tachar" de *withdrawn*, puesto que sigue siendo una ruta factible que sólo ha dejado de ser la mejor. Finalmente se

actualiza la nueva ruta, se guarda en las Adj_RIBs_Out para que se comunique a los nodos vecinos (salvo la conexión origen de la ruta, que ya la conoce) y se hace la "foto" de las RIBs.

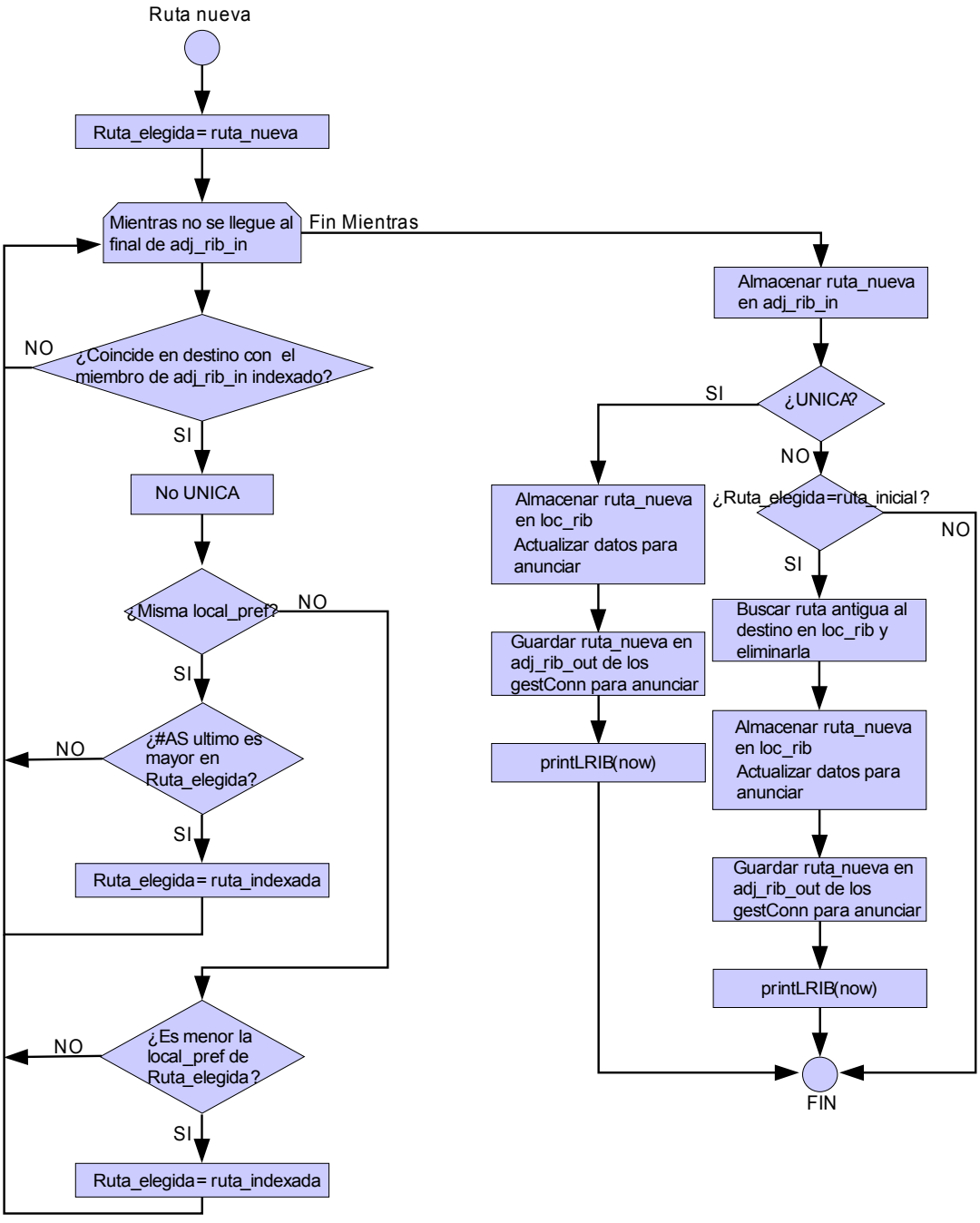

Figura 15. Diagrama de Bloques del Proceso de decisión de ruta nueva.

El segundo bloque de este método se aplica en el caso de que la ruta no sea nueva (*nueva=false*), es decir, se trata de una ruta *withdrawn*, y debemos comprobar si hay otras rutas al mismo destino que ella y, en caso de haberlas, elegir la mejor. Por tanto, lo que se hará es recorrer el vector *adj_rib_in* para ver si hay alguna ruta que coincida en destino con la ruta almacenada en *p_rib*, de modo que si encontramos una primera ruta que coincida pondremos la variable *unica* a *false* (inicialmente estaba a *true*) y almacenaremos directamente esta ruta en *elected*, después seguiremos con

nuestro recorrido de *adj_rib_in* en busca de nuevas coincidencias, de modo que si se encuentran nuevas coincidencias se compararán las rutas de igual modo que en el bloque anterior, de modo que al final tendremos la mejor ruta que habrá de sustituir a la ruta "tachada" de *withdrawn*, o ninguna ruta que pueda sustituirla, si se da el último caso, no habrá que hacer nada, ya que lo ha hecho todo el método *BGPnode::withdrProcess(…)* que es quien ha invocado a este método. Si se tenemos una ruta en sustitución de la *withdrwn* lo que haremos será almacenarla en el vector *loc_rib* y actualizarla como en el bloque anterior antes de almacenarla en las Adj_RIBs_Out de los gestores de conexión para que sea anunciada a los nodos vecinos mediante mensajes UPDATE.

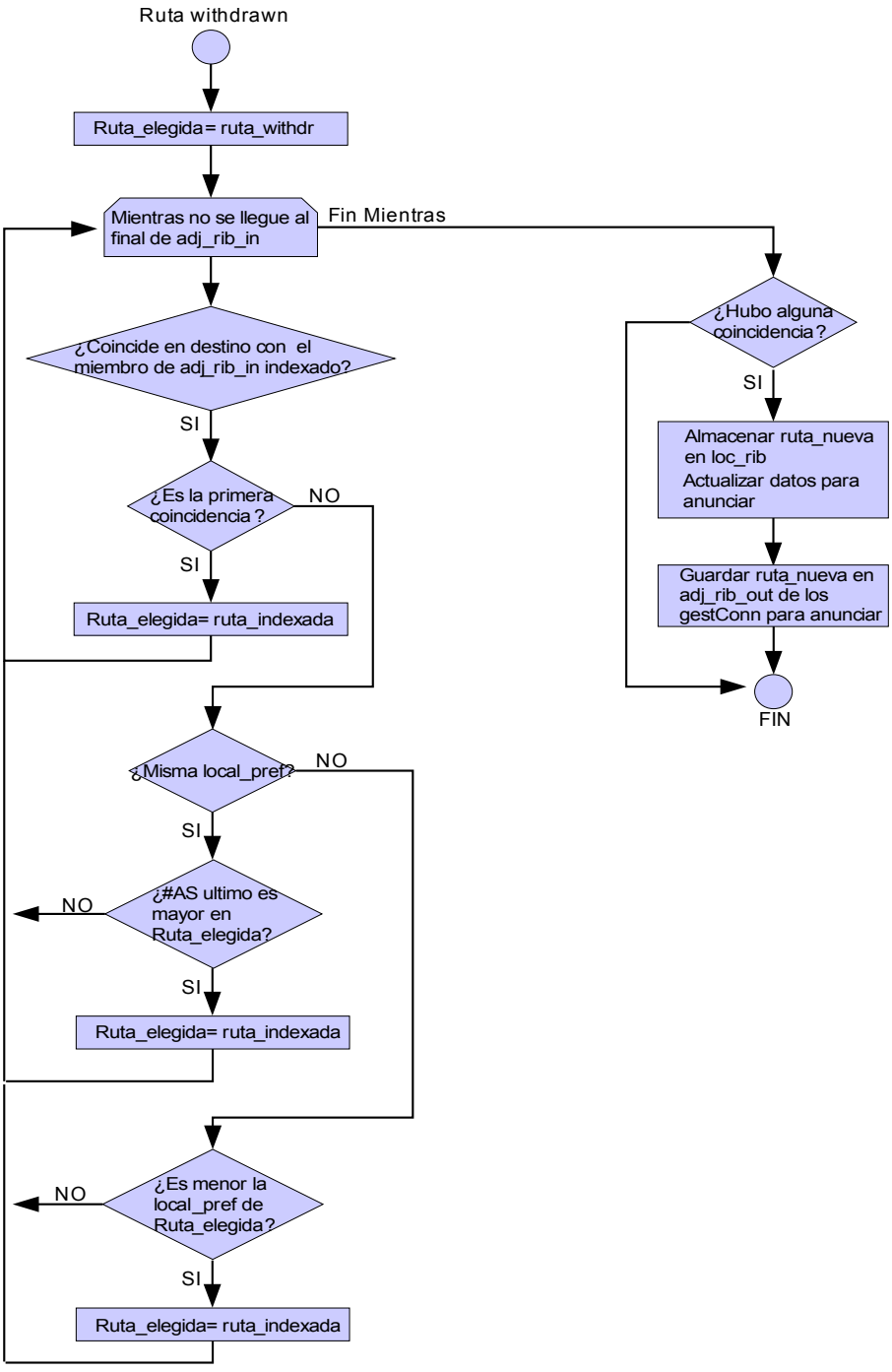

Figura 16. Diagrama de Bloques del Proceso de decisión para rutas withdrawn.

bool BGPnode::check_loop(list <unsigned short int> check)

Este método es llamado para comprobar que una ruta no contiene bucles. Recibe como parámetro una lista de enteros sin signo que se corresponde con el campo AS_PATH de la ruta que estamos comprobando, donde cada uno de los enteros sin signo que forman parte de *check* se corresponde con un identificador de sistema autónomo. Además, este método devuelve un valor booleano que será *true* si la lista check no contiene bucles y *false* si los contiene. Una ruta tendrá un bucle si el número de Sistema Autónomo de un nodo que recibe dicha ruta ya está incluido de en la lista de Sistemas Autónomos por los que ha pasado la ruta.

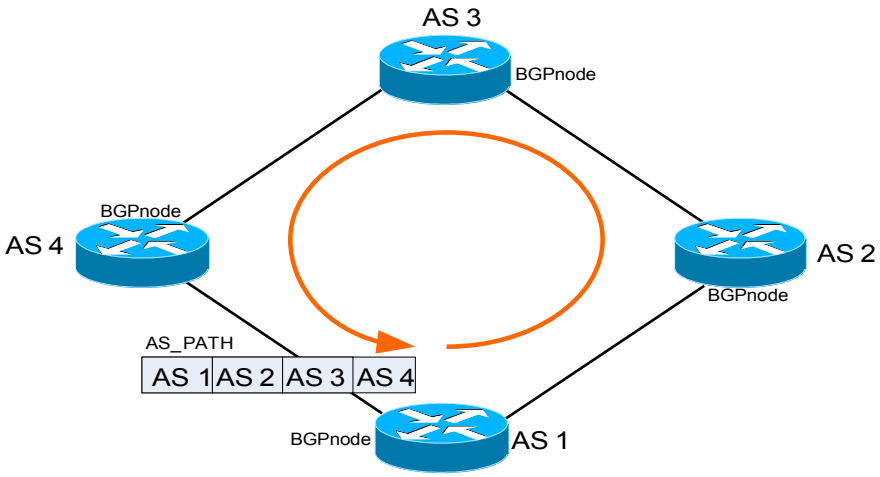

Figura 17. Ruta con bucle.

Este método procede del siguiente modo: en primer lugar se pone la variable local booleana *result* a *true*, tras esto se recorre la lista *check* desde el primer elemento hasta el último, de modo que si alguno de sus elementos coincide con el número de Sistema Autónomo del nodo BGP considerado (variable miembro *as*) se pone a *false* la variable *result*. Finalmente se devuelve el valor de *result*.

gestConn * BGPnode::findgestconn(int to_find)

Este método busca un gestor de conexión en el vector de gestores de conexión de un nodo BGP (*connections*). Recibe como parámetro el identificador del gestor de conexión que se está buscando, este identificador es el mismo que el de la conexión gestionada. Devuelve la referencia a dicho gestor de conexión; recordemos que el vector miembro *connections* está formado por pares *identificador-referencia* (*idc-pgest*) *de gestor* para cada conexión que soporta el nodo BGP.

En este método se recorre cada uno de los elementos de *connections* comparando el campo *idc* con *to_find*, de modo que si coinciden se devolverá el puntero del gestor de conexión que se corresponde con *to_find*.

Básicamente este método se invoca al recibir un mensaje, del cual se extrae el identificador de conexión para buscar el gestor de conexión que deberá interpretar dicho mensaje y actuar en consecuencia.

bool BGPnode::compare_nlri(nlri m, nlri n)

Este método se utiliza para comparar dos *structs nlri*. Recordemos que NLRI es la información de alcanzabillidad de la capa de red (Network Layer Reachability Information). Lógicamente, los dos parámetros que recibe son los dos *structs nlri* a comparar (los compara campo a campo) y devuelve un valor booleano que indica si coinciden (*true*) o no (*false*). Este método se emplea para saber si dos rutas tienen el mismo destino.

bool BGPnode::compare_rib(rib m, rib n)

Este método es invocado para comparar dos *structs rib*. Cada uno de estos *structs* se corresponde plenamente con una ruta, por lo que este método se emplea para comprobar si dos rutas son iguales. Recibe como parámetros los dos *structs rib* a comparar (la comparación es campo a campo) y devuelve un valor booleano que será *true* si coinciden o *false* si no coinciden. Puesto el NLRI forma parte de la ruta (*rib*) internamente a este método se hace una llamada a *BGPnode::compare_nlri(…)*.

void BGPnode::printLRIB(double now)

Este método se emplea para hacer una "fotografía" de las Bases de Información de Encaminamiento (RIBs) Local y de Entrada de un nodo BGP en un instante determinado de la simulación. De este modo podremos observar la evolución de dichas RIBs a lo largo de la simulación para comprobar el funcionamiento de BGP. La antes mencionada "fotografía" consiste en un fichero *.txt* cuyo nombre contiene el identificador de AS del nodo BGP considerado y la versión de la Loc_RIB de dicho nodo, que está almacenada en la variable miembro *version*, de modo que cada vez que cambie ésta tendremos una nueva versión y, por tanto, una nueva "fotografía". Además, se hace una llamada a este método cuando se desactiva un nodo.

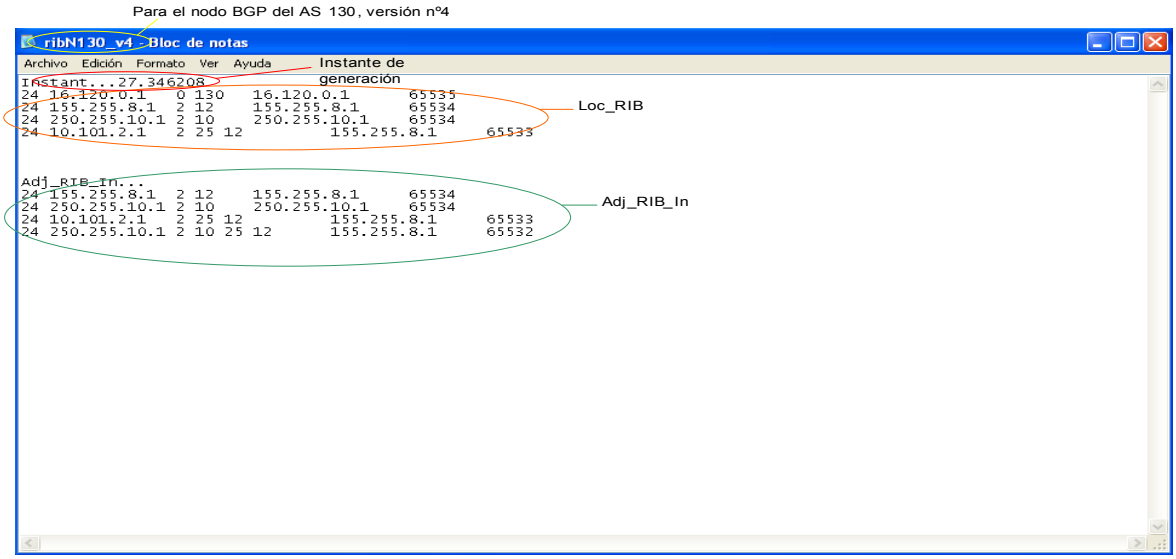

Figura 18. Ejemplo de "fotografía" de RIBs.

En la figura tenemos un ejemplo de "foto" de RIBs, en ella podemos observar en primer lugar el nombre del archivo, *ribN130_v4*, como ya se ha comentado incluye el identificador de AS al que pertenece el nodo (130) y la versión de Loc_RIB (4). Después podemos ver que lo primero que encontramos en el fichero es el instante de generación del fichero, que es el instante en que tenemos la nueva versión de

Loc_RIB. Seguidamente tenemos las rutas que forman parte de Loc_RIB, con una línea por cada entrada de Loc_RIB. Cada ruta o entrada de Loc_RIB consta de (por orden de aparición): la NLRI de la ruta (primero la longitud de bits a '1' que representa a la máscara y luego la dirección de destino), luego el tipo de origen de la ruta (0=local, 2=nodo vecino), tras esto viene el AS_PATH de la ruta, luego el campo next_hop, y finalmente la preferencia local por la ruta. Tras la Loc_RIB viene Adj_RIB_In con todas las rutas que la conforman representadas de igual modo que en Loc_RIB.