4. GESTORES DE CONEXIÓN

La clase gestConn (gestor de conexión) surge como respuesta a la necesidad de gestionar cada una de las conexiones de un nodo BGP de modo independiente a las demás (negociaciones, temporizaciones y envío de los mensajes apropiados). La idea es que un nodo BGP pueda tener tantas conexiones con otros pares BGP como se desee y sea (el nodo) la plataforma común a todas ellas para el envío, procesado y recepción de los mensajes correspondientes. Lógicamente, esta idea recargaría en exceso la clase BGPnode si ésta tuviese que gestionar todas y cada una de las conexiones, por lo que se añadió una clase que fuese capaz de gestionar una conexión y a la vez interactuar con su nodo BGP "base".

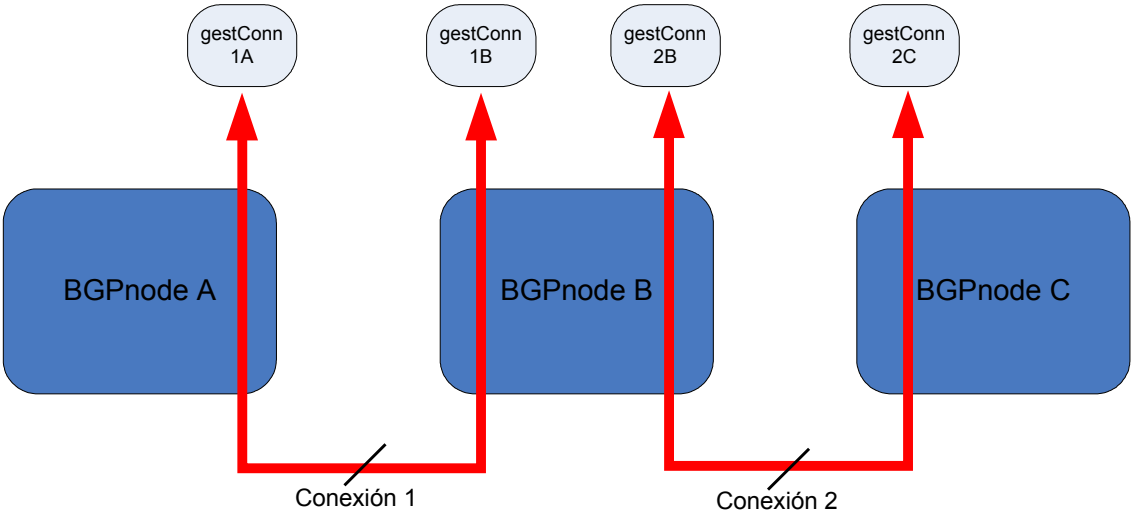

Figura 4. Esquema general de comunicaciones.

Los gestores de conexión de un determinado nodo BGP se diferencian entre sí mediante un identificador de conexión asociada, de este modo al recibir un nodo BGP un determinado mensaje, sólo tiene que leer el identificador de la conexión a la que pertenece el mensaje para saber a qué gestor de conexión remitirse cuando termine el procesado de dicho mensaje. Un nodo BGP tiene un listado con el identificador y una referencia de cada uno de sus gestores de conexión, este listado será la ligadura entre estas dos clases tan relacionadas, ya que cada gestor de conexión se incluirá en ella al generarse (en su constructor) y cada vez que el nodo BGP reciba un mensaje, buscará en este vector el gestor de conexión apropiado (a través del método *BGPnode::findgestconn*) que indique al nodo lo que debe hacerse. En algún momento hay que incluir dicha conexión en la lista de conexiones permitidas para el nodo (llamada desde el *main* al método *BGPnode::link*), puesto que si no, no podrá enviar ni recibir mensajes por dicha conexión (ya que no la reconocería como propia).

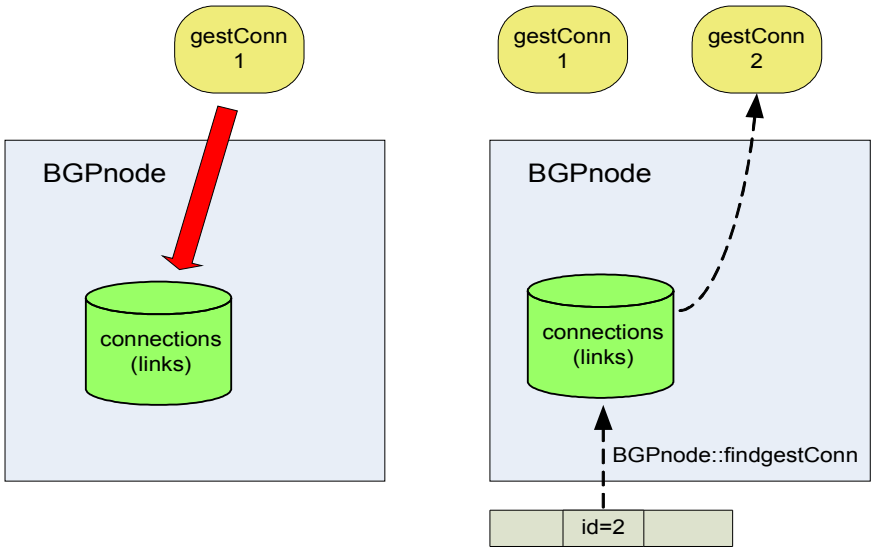

Figura 5. Inscripción de un gestor de conexión e identificación de gestor

El gestor de conexión implementa el modelo de máquina de estados descrita por la **RFC 1771**, aunque en esta máquina de estados se han obviado aquellos estados que se consideran innecesarios para el desarrollo de la simulación. En la siguiente figura se observa el modelo de máquina de estados de la Norma y el considerado para esta simulación.

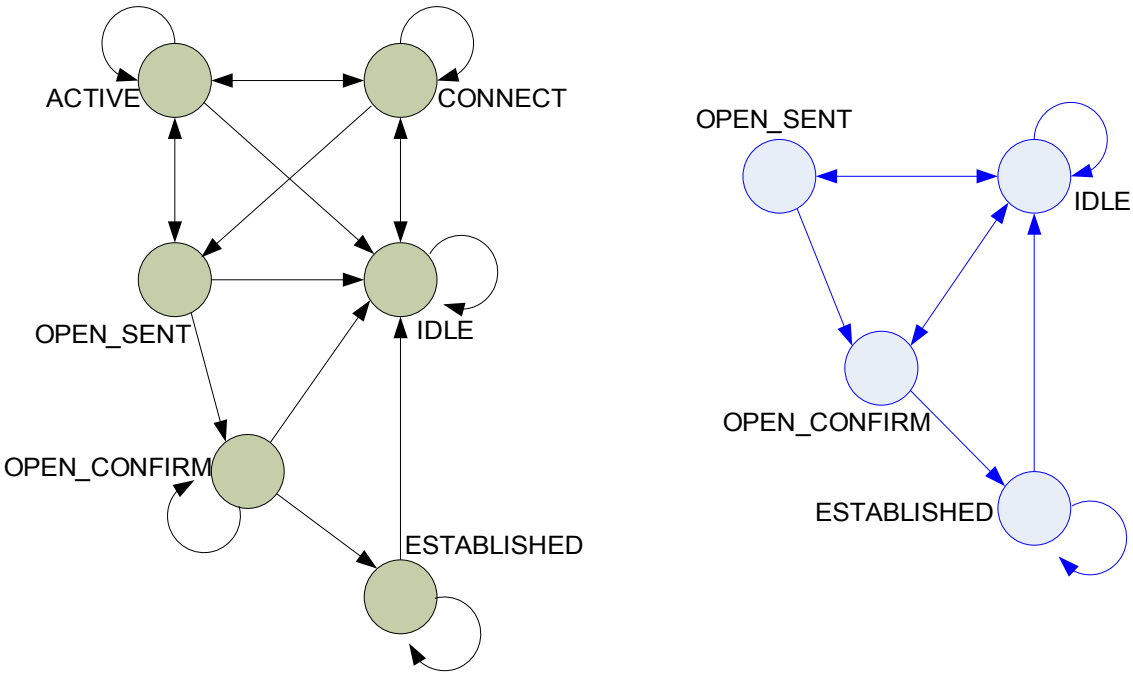

Figura 6. Máquinas de Estados de la Norma (izda.) y de la Simulación (dcha.).

Como se puede apreciar, en la máquina de estados de la simulación se han suprimido dos estados, CONNECT y ACTIVE, esto se ha hecho así por simplicidad, ya que el estado CONNECT es para iniciar o para escuchar intentos de conexión TCP (ya que BGP se apoya sobre TCP) y el estado ACTIVE es para cuando sólo está a la escucha de conexiones TCP, y puesto que se trata de estados dependientes del establecimiento de conexiones TCP, no aportan a nuestra simulación, ya que aunque en realidad hay todo un proceso subyacente a la conexión BGP, para la simulación

resulta intrascendente. Centrándonos en la máquina de estados de la simulación, el inicio de cualquier conexión simulada será el estado IDLE, en este punto, puede ser nuestro nodo BGP el que inicie la conexión (enviamos un mensaje OPEN) o, por el contrario, puede ser un nodo BGP remoto quien inicie la conexión (recibimos un mensaje OPEN). Si es nuestro nodo el iniciador, al enviar un mensaje OPEN pasaremos a OPEN_SENT y esperaremos el mensaje OPEN del nodo BGP remoto; cuando se reciba dicho mensaje se pasará al estado OPEN_CONFIRM. Si nuestro nodo BGP no es el nodo iniciador, puesto que hemos recibido ya un mensaje OPEN, pasaremos a directamente al estado OPEN_CONFIRM y enviaremos un mensaje OPEN. Al entrar en el estado OPEN_CONFIRM se enviará un mensaje KEEPALIVE para confirmar al otro extremo que estamos listos para comenzar la comunicación normal, a la espera de su confirmación. Una vez recibida la confirmación del otro extremo pasaremos a ESTABLISHED y la comunicación será normal. En cualquiera de los estados antes mencionados se puede volver a IDLE si se produce algún fallo en la cabecera, o si se produce el vencimiento del *Hold Timer*.

4.1. DESCRIPCIÓN DE LA CLASE gestConn

Ya se ha descrito la funcionalidad de esta clase, ahora pasaremos a estudiar cada uno de sus componentes.

4.1.1.Variables miembro

double now_old

Se trata de una variable que almacena el instante de tiempo en el que se recibió por última vez un mensaje. Se utiliza para comprobar que no ha expirado el *Hold Timer*.

list <nlri> entries

Se trata de una lista donde se almacenan los *nlri* de las rutas recibidas en mensajes UPDATE a través de la conexión gestionada por este gestConn. Esta lista nos va a resultar muy útil cuando se recibe o se transmite un mensaje NOTIFICATION por esta conexión, ya que conocemos todas las rutas que entraron por esta conexión y, por tanto, a partir del mensaje NOTIFICATION dejan de ser válidas, para ello se enviará la lista al método *BGPnode::withdrProcess* que se encargará de eliminarlas (tanto de Adj_RIB_In como de Loc_RIB) y anunciarlas como *withdrawn* al los pares BGP con los que tenga conexiones válidas.

MessageOpen mopen

Se trata de una instancia de la clase *MessageOpen* donde se almacena el último mensaje de tipo OPEN enviado. Esto es necesario porque en el método *BGPnode::sendOpen* se guarda un evento para que con posterioridad se acceda al mensaje en el receptor, si esto se hace directamente en el método anterior, al intentar acceder a los datos del mensaje nos daremos cuenta que se han perdido, puesto que se generó un mensaje en un método que ya ha concluido y por tanto las variables de este método se han liberado (Figura 7). Una posible solución a esta situación es declarar las variables oportunas en el método *BGPnode::sendOpen* como *static*, pero esto hace que se solapen mensajes de distintas conexiones, con lo que en el receptor

del mensaje se acceden a datos que no tienen que ver con su conexión (Figura 8). Así, la solución es crear un mensaje (OPEN en este caso) en el método *BGPnode::sendOpen* y copiarlo en la variable de la clase *gestConn* (mopen) (Figura 9). De este modo conseguimos la estaticidad necesaria para los mensajes y que *BGPnode* (que es la base de cada conexión) sea independiente de las conexiones que se apoyan en él.

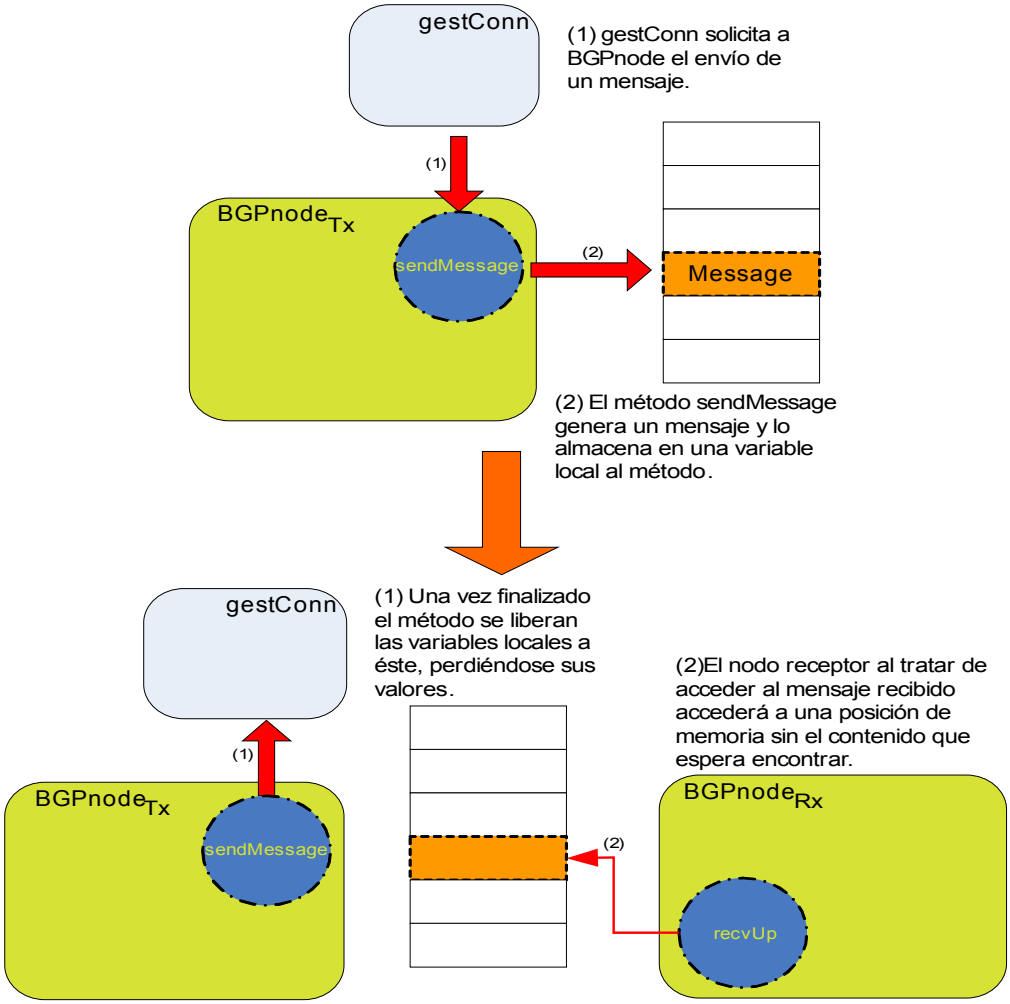

Figura 7. Pérdida de mensajes por falta de estaticidad.

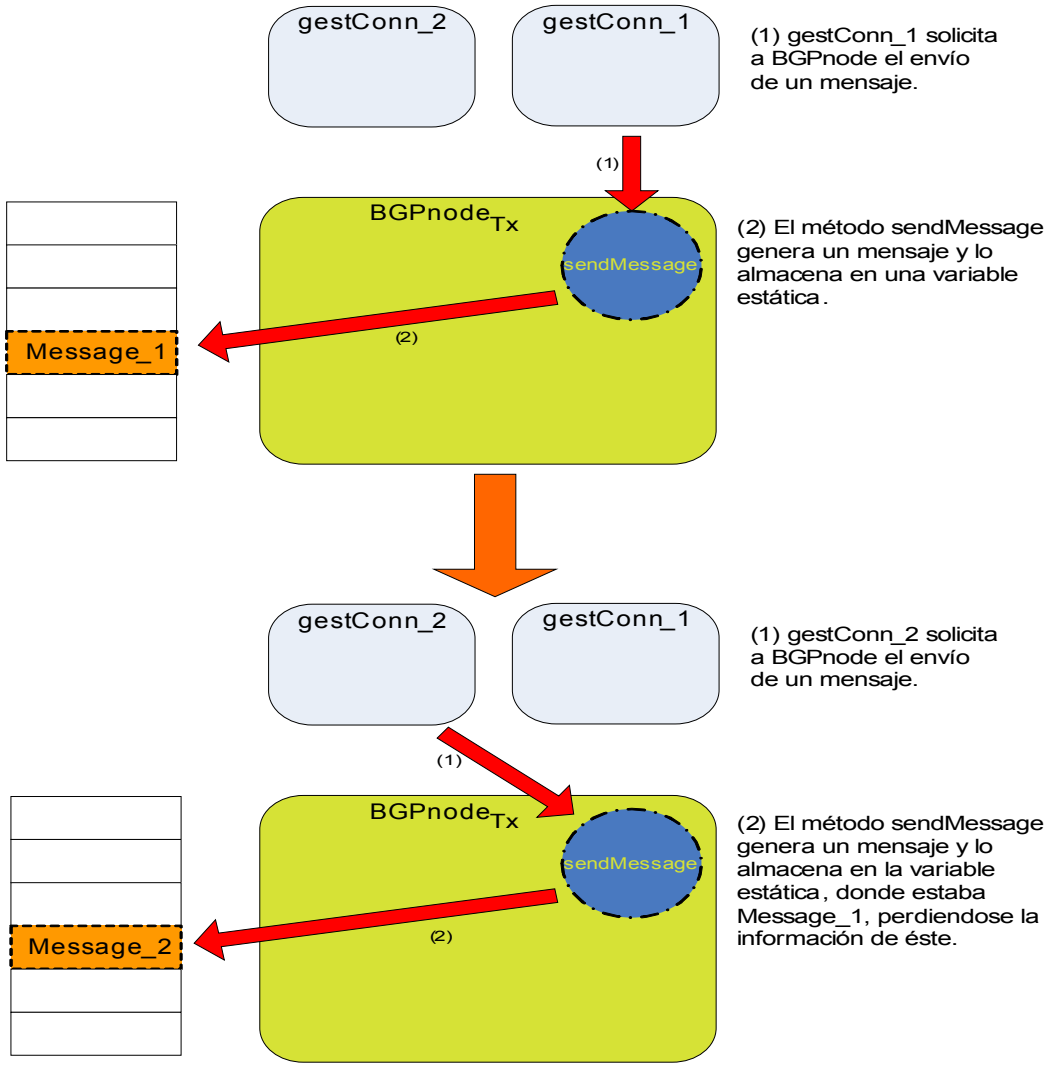

Figura 8. Pérdida de mensajes por solapamiento.

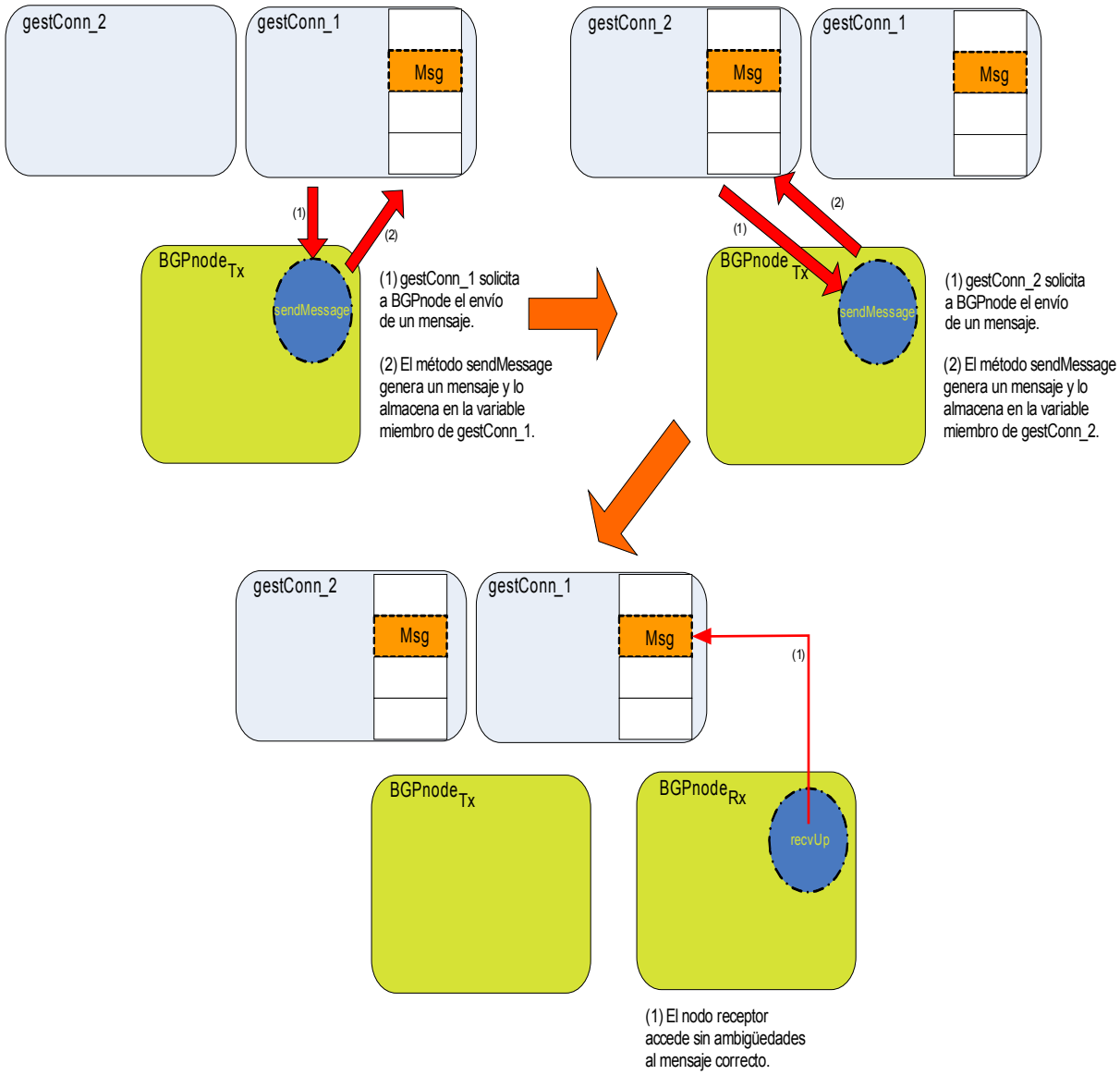

Figura 9. Solución a la pérdida de mensajes.

MessageNotif mnotif

Se trata de una instancia de la clase MessageNotif donde se almacena el último mensaje de tipo NOTIFICATION enviado. Su justificación es la misma que para *mopen*.

Message mm

Se trata de una instancia de la clase Message donde se almacena el último mensaje KEEPALIVE enviado. Su justificación es la misma que para *mopen*.

MessageUpdate mupd

Se trata de una instancia de la clase MessageUpdate donde se almacena el último mensaje UPDATE enviado. Su justificación es la misma que para *mopen*.

BGPnode * nodo_

Se trata de un puntero a *BGPnode*. Con este puntero podremos acceder a los diferentes miembros del *BGPnode* concreto sobre el que se apoya el gestor de conexión.

int idcon_

Se trata de una variable de tipo entero donde se almacena el identificador de conexión al que está asociado el gestor y que como ya se ha explicado, sirve como ligadura entre el *BGPnode* y el *gestConn*.

vector <rib> adj_rib_out_

Se trata de un vector donde se almacenan los *rib* de las rutas que deben ser anunciadas como nuevas rutas factibles a los nodos BGP vecinos. Además se usa (junto con *list <nlri> withdrawn*) para saber si hay que enviar un mensaje KEEPALIVE o un UPDATE. Almacena *rib* porque contiene información más completa que el *nlri* (además de tener el *nlri* tiene los *path attributes* y el identificador de conexión) y así cada uno de sus elementos se introducirá directamente en una estructura *data_update* para ser enviado en un mensaje UPDATE. Una vez se anuncian sus elementos, se eliminan.

unsigned short int holdT

Se trata del valor del *Hold Timer* que se usará en esta conexión, en principio es el valor del *Hold Time* del nodo BGP, pero durante la negociación de la conexión tomará el menor de los valores propuestos por los gestores de conexión implicados. Este valor marca el tiempo (en segundos) que como máximo se mantendrá una conexión activa sin que se registre actividad en ella. Transcurrido dicho tiempo la conexión se cerrará.

bool firstKeep

Se trata de un valor booleano que se usa para saber si el nodo ha enviado o no el mensaje KEEPALIVE con el que se confirma la conexión, lo cual nos es útil en el establecimiento de la misma para que se lleven a cabo los cambios de estado oportunos y el consecuente envío de mensajes correcto.

list <nlri> withdrawn

Se trata de una lista de *nlri* en la cual se almacenan las rutas que han dejado de ser válidas. Se mapea directamente sobre el campo *withdr_routes* de la estructura *data_update* para ser enviada en un mensaje UPDATE. Como ya se ha indicado, también se emplea para saber si se debe enviar un mensaje UPDATE o un KEEPALIVE. Una vez se anuncia, se borra todo su contenido.

enum status_

Se trata de un enum en el que se tiene el estado en que se encuentra la conexión (la máquina de estados) y que, por tanto, condiciona el funcionamiento de la conexión. Sus posibles valores son: IDLE=0, OPEN_SENT=1, OPEN_CONFIRM=2 y ESTABLISHED=3. Su valor va variando conforme va evolucionando la conexión de

acuerdo a la figura anterior donde se presenta la máquina de estados.

4.1.2. Métodos

gestConn::gestConn(BGPnode * nodo, int conn)

El constructor de la clase *gestConn* recibe dos parámetros, el primero de ellos es un puntero a *BGPnode*, que se utiliza para obtener la referencia al nodo BGP al que se asocia el gestor de conexión, y el segundo es un entero que indica al gestor el identificador de la conexión que gestionará.

El constructor lleva acabo la inicialización de los parámetros de la clase, esta inicialización consiste en:

- Fijar la referencia al nodo BGP asociado.
- Fijar el estado inicial de la conexión a "IDLE".
- Almacenar en Adj_RIB_Out, para anunciar, las rutas que el nodo tiene almacenadas en Loc_RIB.
- Fijar el Hold Time de la conexión al Hold Timer del nodo BGP (para negociarlo después).
- Fijar el identificador de la conexión asociada.
- Limpiar (por precaución) las listas *entries* y *withdrawn*.
- Poner la variable *firstKeep* a *true* (para la evolución de la máquina de estados).

Finalmente se introduce en el vector de conexiones del nodo (para que éste tenga acceso al gestor).

void gestConn::init(double now)

Este método marca el inicio de una conexión, es llamado desde el *main* de la aplicación y hace que el gestor de conexión de un nodo inicie la conexión con el gestor asociado a otro nodo mediante el envío de un mensaje OPEN. En nuestro lado de la conexión pasamos del estado "IDLE" al estado "OPEN_SENT", ya que hemos pasado de estar ociosos a haber enviado un mensaje OPEN para establecer una conexión BGP. El parámetro *now* indica el instante en el que se enviará dicho mensaje OPEN.

void gestConn::initResp(double now, data_open data)

Este método se activa cuando el nodo BGP recibe un mensaje de tipo OPEN con el identificador de la conexión a la que está asociado nuestro gestor. Lo primero que se hace es comprobar si el valor *Hold Time* que viene en el mensaje OPEN es menor que el nuestro, en cuyo caso adoptaremos ese *Hold Time* remoto para el *Hold Timer* de nuestra conexión, si no es así, nos quedaremos con el nuestro asumiendo que en el otro extremo se ha tomado o se tomará también éste (esto será la negociación de la conexión BGP).

Existen dos posibles situaciones en las que recibiremos un mensaje OPEN, la primera que se ha considerado es que ya estemos en el estado "OPEN_SENT", es decir somos el lado que ha iniciado la conexión, hemos enviado con anterioridad un mensaje OPEN y estamos a la espera de un mensaje OPEN por parte del otro extremo para fijar los parámetros de la conexión (*Hold Time*). En ese caso pasaremos al estado "OPEN_CONFIRM" y enviaremos el primer mensaje KEEPALIVE (en el envío de este mensaje ya se activa la función de chequeo del *Hold Timer*, esto se hace introduciendo

un evento en el instante *now+holdT* que llama a la función de chequeo *BGPnode::check_ht*), indicando que la conexión ya está establecida a falta de la indicación del otro extremo (mensaje KEEPALIVE del otro extremo). Además se pondrá a *false* la variable *firstKeep*. La segunda posible situación es que nuestra conexión esté en estado "IDLE" y reciba una petición de conexión por parte de un nodo BGP remoto (el mensaje OPEN), en ese caso pasaremos directamente del estado "IDLE" a "OPEN_CONFIRM" y enviaremos un mensaje OPEN con nuestros parámetros.

El parámetro *now* indica el instante de recepción (actualizará la variable *now_old* para chequeo del *Hold Timer*). El parámetro *data* es donde están almacenados los datos del mensaje OPEN (de donde obtenemos el valor de *Hold Time*).

void gestConn::keepResp(double now)

Este método gestionará el envío de mensajes KEEPALIVE y UPDATE mientras que la conexión esté activa. Lo primero que se hace es comprobar que el estado de la conexión en nuestro extremo no sea "IDLE" (caso de simulación de la caída del enlace) ni que el nodo esté parado (caso de simulación de la caída de un nodo), ya que en estos casos no se deben enviar mensajes KEEPALIVE ni UPDATE. Si no se da ninguna de estas situaciones, actualizaremos *now_old* (para el chequeo).

Podemos entrar en este método en distintas situaciones. En primer lugar podemos estar en el estado "OPEN_CONFIRM" con *firstKeep* a *true* (**Situación 1**), esto ocurre cuando no somos el extremo iniciador de la conexión y ya hemos enviado el mensaje OPEN. En segundo lugar podemos estar en el estado "OPEN_CONFIRM" con *firstKeep* a *false* (**Situación 2**), lo cual implica que somos el extremo iniciador de la conexión y ya hemos enviado un primer mensaje KEEPALIVE indicando que por nuestra parte estamos listos para una comunicación normal, pero a la espera de esta indicación por parte del otro extremo de la conexión. La tercera situación posible es que ya la comunicación sea normal (que estemos en el estado "ESTABLISHED") (**Situación 3**).

En función de la situación en la que estemos, el comportamiento será distinto. Analicemos los distintos comportamientos:

- **Situación 1:** en esta situación pasaremos al estado "ESTABLISHED" y enviaremos un mensaje KEEPALIVE indicando que a partir de ese momento la comunicación será normal, generando además el evento de chequeo.
- **Situación 2:** en esta situación sólo nos queda pasar al estado "ESTABLISHED" y puesto que ya hemos enviado con anterioridad el mensaje KEEPALIVE de confirmación (en *gestConn::initResp*) estamos en situación de comunicación normal y pasamos, por tanto, a la **Situación 3**.
- **Situación 3:** esta es la situación de comunicación normal y se enviarán mensajes KEEPALIVE o UPDATE en función de que haya o no datos sobre rutas para transmitir. Este funcionamiento se explicará en los siguientes párrafos.

Se va a detallar ahora el comportamiento del gestor de conexión en la **Situación 3**. Lo primero será saber si se debe enviar un mensaje KEEPALIVE o bien un mensaje UPDATE; para ello se comprueba si el vector *adj_rib_out_* del gestor está

o no vacío, si lo está indica que no hay rutas nuevas para comunicar al nodo BGP extremo de la conexión y debemos comprobar si la lista *withdrawn* está o no vacía, si está vacía indica que no hay rutas que hayan dejado de ser válidas, por tanto, ya que no hay nada nuevo que comunicar al extremo de la conexión, se envía un mensaje KEEPALIVE (llamada a *nodo_->sendKeep(…)*)para que la conexión permanezca activa. Posteriormente se generará el evento de chequeo. Si no tenemos rutas nuevas para anunciar pero sí hay rutas que hayan dejado de ser válidas (*withdrawn* no vacía), tendremos que enviar un mensaje UPDATE. Este mensaje tendrá la peculiaridad de que no anuncia rutas nuevas, con lo que el campo *tot_PA_l* (total path attributes length) de la estructura *data_update* que se usará para generar el mensaje UPDATE valdrá 0, esto hará que el receptor sepa que no hay rutas nuevas en este mensaje. La lista *withdrawn* se volcará en dicha estructura y después se borrará su contenido (ya que ha sido anunciado). Finalmente se llamará al método *sendUpd* del nodo al que está asociado el gestor de conexión para que éste genere y envíe el mensaje UPDATE y se generará el evento de chequeo. La última posibilidad es que *adj_rib_out_* no esté vacía, es decir, que haya rutas nuevas para ser anunciadas, en este caso se toma el primer elemento del vector (*adj_rib_out_.begin()*) y se mete en la estructura *data_update*, una vez hecho esto se elimina dicha ruta de *adj_rib_out_* (puesto que va a ser anunciada) y se comprueba si hay rutas que hayan dejado de ser válidas, si no hubiese, se pondría el campo *unf_route_l* de la estructura *data_update* a 0, indicando al receptor esta situación, si hubiese, se introducirían en dicha estructura y finalmente se procede a llamar al método *sendUpd* del nodo para que genere el mensaje y lo envíe. También se generará el evento de chequeo.

Como se ha podido observar, en este método se genera una estructura *data_update* que se pasará al nodo asociado al gestor para que éste genere el mensaje UPDATE. Además, estos mensajes sólo anuncian una nueva ruta cada vez, con lo que si hubiese más de una ruta nueva, se enviarán varios mensajes UPDATE, aunque no se enviarán todos a la vez.

Otro punto destacable es que cuando se llama a los métodos *sendKeep* o *sendUpd* del nodo asociado a la conexión, el instante de la transmisión del mensaje correspondiente se fija a un tercio de *Hold Time* sobre el instante actual. Esto se hace así por recomendación de la **RFC 1771**.

void gestConn::notification(double now, char err, char err_subc)

Este método se activa cuando se produce una situación de error. Esta situación de error puede ser o bien un fallo detectado en un mensaje KEEPALIVE o en un mensaje UPDATE, o bien el vencimiento del *Hold Timer* por la caída del enlace o por la caída de un nodo.

En el caso de activarse este método se eliminará aquellas rutas aprendidas por la conexión asociada a este gestor mediante la llamada al método *BGPnode::withdrProcess(…)* pasándole como rutas no válidas todas las rutas almacenadas en la lista *entries*. Después de esto se forzará la conexión al estado IDLE y se enviará un mensaje NOTIFICATION al par BGP remoto indicando que se ha producido un error. Vemos que la finalización de la conexión en caso de error es una desconexión abrupta, ya que no espera confirmación de la desconexión por parte del extremo de la conexión.

Los parámetros pasados a este método son el instante en que se produce el error (*now*) y el código (*err*) y subcódigo (*err_subc*) de error, cuyos valores serán los

fijados por la **RFC 1771**.

void gestConn::notifResp(double now)

Este método se activa cuando se recibe un mensaje NOTIFICATION. En este caso se eliminan las rutas aprendidas por la conexión asociada de igual modo que en *gestConn::notification(…)* y se fuerza la conexión al estado IDLE. El único parámetro que se le pasa a este método es el instante de reopción del mensaje (*now*).