Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

ANÁLISIS DE LOADNG COMO PROTOCOLO DE RUTADO DINÁMICO PARA LAS COMUNICACIONES PLC EN MT

DOCUMENTO Nº4: MANUAL DEL PROGRAMADOR

Alumno *Rodríguez Fernández, Jorge*
Director *de la Vega Moreno, David*
Departamento Ing. de Comunicaciones
Curso académico *2017/2018*

Bilbao, a 6 de Febrero de 2018

Índice

Índice de ilustraciones

1 Anexo II: Manual del programador

1.1 Introducción

En este documento de hace un resumen de referencias para que el programador que en un futuro realice modificaciones o mejoras del software sepa donde está ubicada cada funcionalidad del programa.

1.2 Estructura global

Módulo Node.ned simularía el funcionamiento de un router con el protocolo LOADng. Está formado por un submódulo App.ned, un submódulo Routing.ned y tantos submódulos L2Queue.ned como conexiones tenga el router.

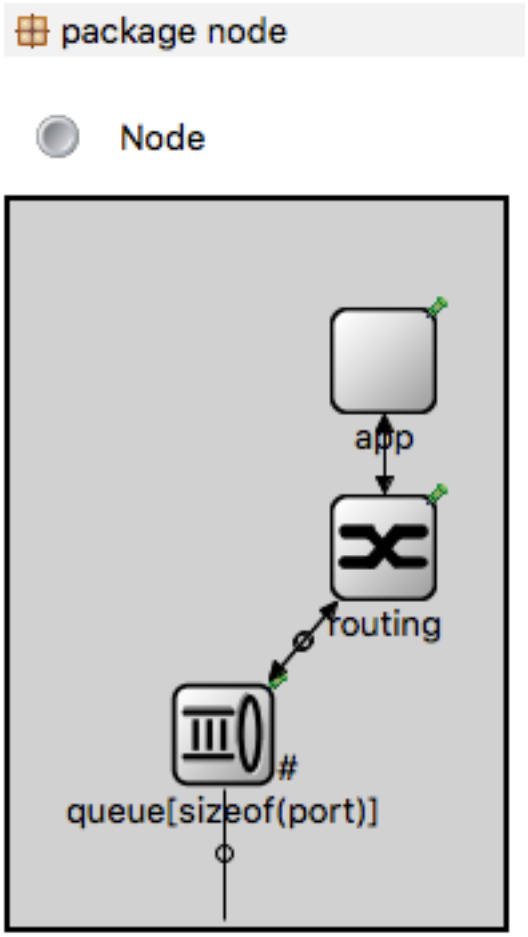

Ilustración 37 Node

Una conexión bidireccional entre app y routing y también una conexión bidireccional entre routing y queue.

App se encarga de generar los paquetes según las configuraciones de tamaño, instante y destino recogidas del documento .ini. Simula una aplicación que necesita mandar información a un destino.

Routing cada mensaje que recibe lo trata siguiendo la lógica del protocolo, según la necesidad puede crear, duplicar o destruir mensajes. Es aquí donde se decide qué hacer con los mensajes. También es aquí donde se almacenan

las tablas de enrutamiento y donde se producen las temporizaciones necesarias. Obtiene los parámetros de configuración del archivo .ini .

Queue resulta ser una cola FIFO bidireccional que lo único que simula don las colas de entrada y salida de los mensajes del router. Será este submódulo también el que se encargue de detectar las posibles desconexiones de enlaces en la red y será así mismo el encargado crear y mandar los mensajes de error que aparecerán al suceder estos eventos.

Por otra parte, el módulo eventos.ned actúa en función de documento .xml donde se han configurado los eventos que sucederán durante la simulación.

1.3 App.cc

En App básicamente se crean los mensajes que se quieren mandar y se destruyen los que han llegado a su destino. Se crean según la configuración del .ini .

- `Void App::Initilize()`

Aquí se obtiene la información de la configuración el .ini y se genera un mensaje `generatePacket` cuando toque crear el paquete.

- `Void App::handleMessage(cMessage *msg)`

Trata lo mensajes que llegan a App, si se trata de un `generatePacket` se crea un mensaje con la información necesaria y de tipo PACK. Si se trata de otro mensaje, únicamente se puede tratar de un mensaje que alcanza su

destino, se recaba la información necesaria de él para las estadísticas y tras esto, se borra.

1.4 Eventos.cc

Esto es el módulo que obtiene los eventos que van a suceder del .xml . Cada función aquí representada es cada uno de los comandos que se pueden utilizar, es decir, cada evento que puede suceder en la simulación. Cada posible evento se encuentra explicado en `eventos.xml`. Modulo sacado de INET.

- `void processSetParamCommand(cXMLElement *node);`

Esta función permite cambiar un parámetro de un nodo en concreto.

- `void processSetChannelAttrCommand(cXMLElement *node);`

Esta función permite cambiar un parámetro de un canal entre nodos.

- `void processCreateModuleCommand(cXMLElement *node);`

Esta función permite la creación de un módulo nuevo.

- `void processDeleteModuleCommand(cXMLElement *node);`

Esta función permite la eliminación de un módulo nuevo.

- `void processConnectCommand(cXMLElement *node);`

Esta función permite la creación de una conexión entre nodos.

- `void processDisconnectCommand(cXMLElement *node);`

Esta función permite la eliminación de una conexión entre nodos.

1.5 L2Queue.cc

Aquí se simulan las colas de los router:

- `void initialize() override;`

Se inicializan las colas como no ocupadas.

- `void handleMessage(cMessage *msg) override;`

Cada mensaje que llega a alguna cola se comprueba si la cola esta libre y se manda o si la cola está ocupada se pone a la espera de terminar la transmisión

- `void refreshDisplay() const override;`

Muestra durante la simulación si la cola se encuentra ocupada o no cambiando el módulo de color.

- `void startTransmitting(cMessage *msg);`

Cuando un mensaje puede ser mandado, se realiza, pero antes se comprueba si la cola está conectada a modulo por el que seguir el envío. En caso de no ser así se trataría de una desconexión y se crea aquí entonces el mensaje RRER que sería mandado por el camino de vuelta actualizando las tablas de enrutamiento.

1.6 Routing.cc

Aquí es donde se encuentra la lógica del protocolo, es donde están las tablas de enrutamiento de cada nodo y donde se decide qué hacer con cada mensaje que llega a cada router. Hay 2 tablas principales, temporal, que son las tablas que crean por el RREQ y, direcciones, que son las tablas creadas por los RREP, las rutas finales.

- `void initialize() override;`

Se obtienen todos los datos necesarios del .ini y se inicializan las tablas y temporizadores con entrada en blanco en cada una ellas.

- `void handleMessage(cMessage *msg) override;`

Cuando llega un mensaje se identifica que tipo de mensaje es y se manda a cada una de las siguientes funciones según el mensaje que sea. También se muestran por consola las tablas cada vez que llega un mensaje para ir viendo cómo se van actualizando.

- `void handlepack(cMessage *msg);`

Cuando el mensaje que llega se trata de un PACK, lo primero que se hace es comprobar que su destino es distinto al propio nodo. En caso de ser distinto, se comprueba si ya existe ruta para él. Si hay ruta, se manda por ella. Si no hay ruta, comienza el proceso de búsqueda de ruta y se crea el mensaje RREQ y se inician los temporizadores de espera de respuesta.

- `void forwardMessage(cMessage *msg);`

Esta función se utiliza cuando se crea un RREQ y es para mandar ese mensaje por todos los puertos de los que disponga el router.

- `void handleRREQ(cMessage *msg);`

Cuando un RREQ llega a un router según el criterio de selección de ruta se va consultando la ruta temporal y en función de las entradas de la tabla se actualiza la tabla, y los mensajes se van borrando o continuando en todos los puertos posibles menos por el que ha venido. Si alcanza el destino, se crea el mensaje RREP y se direcciona por el camino de vuelta al origen del RREQ por la ruta más óptima.

- `void handleRREP(cMessage *msg);`

Cuando llega un RREP, se van consultando las tablas temporales y la tabla direcciones se va actualizando para ir haciendo las rutas, según el criterio configurado en el .ini .

- `void handleRRER(cMessage *msg);`

Cuando llega un RRER quiere decir que hay una ruta que ya no es válida, se borra la entrada en la tabla de direcciones que ya no es válida y se reenvía.

- `void handleWaitForRREP(WaitForRREP *rrepTimer);`

Espera por la respuesta al encontrar un destino, actualiza los mapas de espera

- `void handleWaitForRREQ(WaitForRREQ *rreqTimer);`

Función pensada para posible futura mejora del protocolo, actualmente no hace nada.

- `LOADngRREQ *createRREQ(int destAddr);`

Aquí se crea el mensaje RREQ con la información del PACK y el router.

- `LOADngRREP *createRREP(LOADngRREQ *rreq);`

Aquí se crea el mensaje RREP con la información del RREQ y el router.

- `virtual void refreshDisplay() const override;`

Esto se encarga de mostrar durante la simulación en tiempo real las tablas de direcciones sobre el submódulo routing.ned.

1.7 Packet.msg

Aquí están definidos los tipos de mensajes que se utilizan:

- `packet Packet`

Este es el paquete general que se crea en App, la información que tiene es el origen del paquete, su destino, el número de saltos dados y el tipo de paquete PACK.

- `enum LOADngControlPacketType`

Esto es únicamente es para que cada tipo de paquete tenga un tipo int asignado para facilitar su uso.

- `class LOADngRREQ extends Packet`

Aquí se define la clase del mensaje RREQ, la información que tiene es el tipo de paquete RREQ, el número de saltos dados, el número de enlaces débiles por los que ha pasado, un ID único, su destino, su origen, y nodo previo por el que pasado.

- `class LOADngRREP extends Packet`

Aquí se define la clase del mensaje RREP, la información que tiene es el tipo de paquete RREP, el número de saltos dados, su destino, su originador, el nodo previo por donde ha pasado, el número de enlaces débiles por los que ha pasado y su ID.

- `class LOADngRRER extends Packet`

Aquí se define la clase del mensaje RREP, la información que tiene es el tipo de paquete RRER, el destino del mensaje y el nodo que lo origina.

- `message WaitForRREP`

Mensaje que representa el tiempo de espera por un RREP, únicamente utiliza el destino.

- `message WaitForRREQ`

Mensaje que representa el tiempo de espera por un RREQ, utiliza el origen y el destino.