DISEÑO E IMPLEMENTACIÓN DEL PROTOCOLO ForCES

Autor:
PEDRO LUIS GONZALEZ RAMIREZ

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
MAESTRÍA EN INGENIERÍA ELECTRÓNICA
BOGOTÁ - COLOMBIA.
ENERO DE 2012

DISEÑO E IMPLEMENTACIÓN DEL PROTOCOLO ForCES

Autor:
PEDRO LUIS GONZALEZ RAMIREZ
Ingeniero Electrónico

Proyecto de Grado presentado como requisito para optar el titulo de:
MAGISTER EN INGENIERIA ELECTRONICA

Director
LUIS CARLOS TRUJILLO
Ingeniero Electrónico

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
MAESTRÍA EN INGENIERÍA ELECTRÓNICA
BOGOTÁ - COLOMBIA.
ENERO DE 2012

PONTIFICIA UNIVERSIDAD JAVERIANA

FACULTAD DE INGENIERÍA

MAESTRÍA EN INGENIERÍA ELECTRÓNICA

RECTOR MAGNÍFICO: R.P. JOAQUIN EMILIO SANCHEZ GARCÍA S.J

DECANO ACADÉMICO: Ing. FRANCISCO JAVIER REBOLLEDO MUÑOZ

DECANO DEL MEDIO UNIVERSITARIO: R.P SERGIO BERNAL RESTREPO S.J

DIRECTOR DE MAESTRÍA: Ing. CARLOS ALBERTO PARRA R, Ph.D

NOTA DE ADVERTENCIA

"La Universidad no se hace responsable de los conceptos emitidos por algunos de sus alumnos en los proyectos de grado. Solo velará porque no se publique nada contrario al dogma y la moral católica y porque no contengan ataques o polémicas puramente personales. Antes bien, que se vea en ello el anhelo de buscar la verdad y la justicia."

Artículo 23 de la Resolución No. 13, del 6 de julio de 1946, por la cual se reglamenta lo concerniente a Tesis y Exámenes de Grado en la Pontificia Universidad Javeriana.

Nota de aceptación

4.2 (Cuatro Dos)

_____

Luis Carlos Trujillo

_____

Presidente del Jurado

_____

Jurado

_____

Jurado

Bogotá - Miercoles, 01 de Febrero de 2012

DEDICATORIA

Dedico este proyecto de grado con todo mi corazón:

A Dios y a mi maravillosa familia

“No son héroes los desesperados, sino los que en plena serenidad y juicio prosiguen un camino trazado y avanzan, sin que se precipite su pulso ni se enardezca su sangre.” E.W. Stevens.

AGRADECIMIENTOS

El autor de este trabajo expresa sus agradecimientos a:

Luis Carlos Trujillo Arboleda director del poyecto de grado, a la Universidad Central, directivos y compañeros, también a todos mis amigos que me acompañaron durante este tiempo.

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE ANEXOS

GLOSARIO

Arquitectura ForCES: Los componentes principales de la arquitectura de un NE (*Network Element*, Elemento de Red), son los CE (*Control Element*, Elemento de Control), el FE (*Reenvío Element*, Elemento de Reenvío) y el protocolo de interconexión.

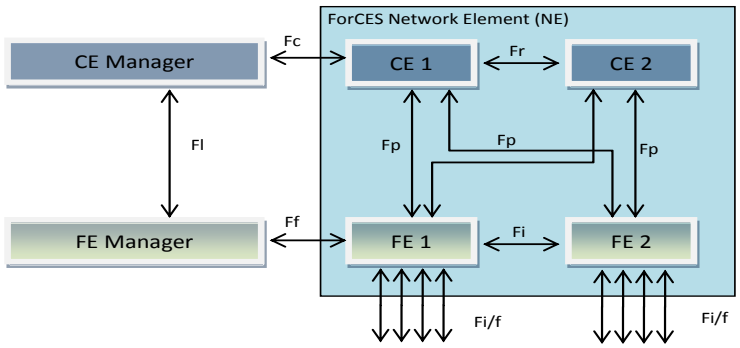
Control Element (CE): Elemento de Control (CE), Es una entidad lógica que implementa el protocolo ForCES y la utiliza para transportar paquetes de información con mensajes de manejo sobre las funciones de los FEs.

CE Manager (CEM): Administrador de CE, Es una entidad lógica que opera en la fase de pre-asociación y es responsable de determinar a cada FE con que CE se debe comunicar. Este proceso se llama descubrimiento de FEs, e involucra al administrador del CE, aprendiendo de las capacidades disponibles en los FEs.

FE Manager (FEM): Administrador de FE, Es una entidad lógica que opera en la fase de pre-asociación y es responsable de determinar a cada CE con que FE se debe comunicar. Este proceso se llama descubrimiento de CEs, e involucra al administrador de los FE, poniendo a disposición las capacidades disponibles en los FEs a los CEs.

FE Model: Modelo FE, Un modelo que describe las funciones de procesamiento lógico de un FE.

ForCES Network Element (NE): En la figura 1, se muestra un ejemplo de un Elemento de Red (NE), compuesto de un CE y FEs. Los FEs y el CE, requieren una configuración mínima como parte del proceso de pre-configuración y este puede ser hecho por el Administrador de FE (FE Manager) y el Administrador del CE (CE Manager). Estos componentes están por fuera del alcance de la arquitectura y los requerimientos del protocolo ForCES, el cual únicamente involucra los CEs y los FEs.



Fp: CE-FE interface
Fi: FE-FE interface
Fr: CE-CE interface
Fi/f: interface externa FE

Fc: Interface entre un administrador CE y un CE
Ff: Interface entre un administrador FE y un FE
Fl: Interface entre un administrador CE y un administrador FE

Figura 1. ForCES Network Element (NE)

ForCES Protocol: Protocolo ForCES, Si bien puede haber múltiples protocolos utilizados en la arquitectura general ForCES, el término "protocolo ForCES" se refiere sólo al protocolo utilizado en el punto **FP** (CE-FE interface, comunicación entre un CE y un FE) definido en el RFC3746.

Este protocolo no aplica para comunicaciones entre CE a CE, FE a FE, y para comunicaciones entre administradores CE y FE.

ForCES Protocol Layer (PL): Capa de Protocolo ForCES (ForCES PL), La capa de protocolo ForCES, en la arquitectura lo definen los mensajes, es donde se construyen y encapsulan los mensajes del protocolo ForCES (incluyendo los requerimientos del TML).

ForCES Protocol Transport Mapping Layer (TML): Protocolo de la capa de asignación de transporte (ForCES TML), Es una capa de la arquitectura que específicamente aborda al protocolo, en lo que tiene que ver con el transporte de mensajes. Por ejemplo, los mensajes de protocolo se asignan a los diferentes medios de transporte (TCP, UDP, SCTP) y es el que implementa la seguridad de los datos.

El dominio del protocolo ForCES se encuentra en el punto de referencia **FP** visto en la figura.

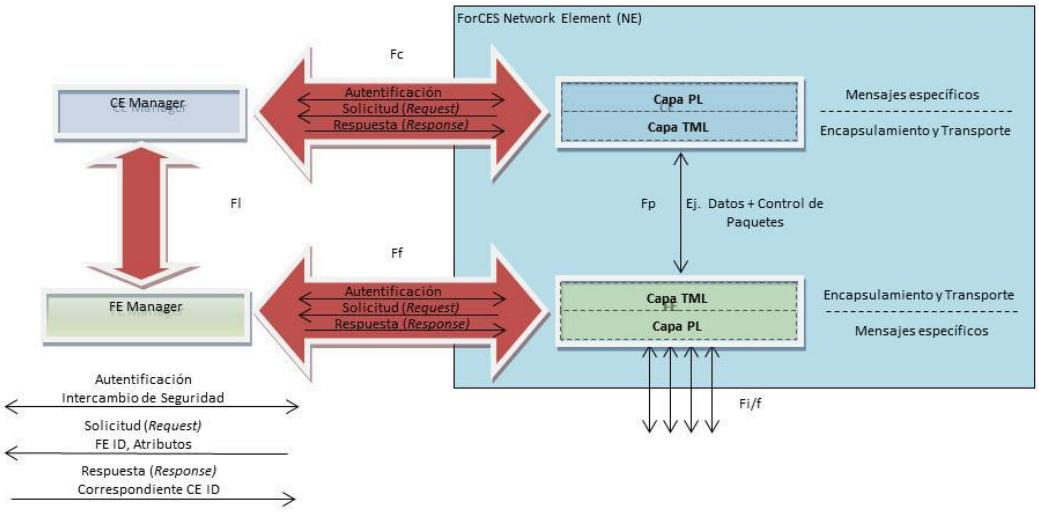


Figura 2. Punto de Referencia (FP) donde opera el protocolo ForCES

Forwarding Element (FE): Elemento de Reenvió (FE), Es una entidad lógica que implementa el protocolo ForCES. Los FEs usan el correspondiente hardware que maneja los paquetes a nivel de la tarjeta de red donde residen los LFBs, y es controlado por el CE a través del protocolo ForCES.

LFBs: Bloques lógicos funcionales, son pequeñas entidades definidas a través de una estructura XML, que tienen operaciones especificas definidas mediante atributos, las cuales al estar conectadas a través de una topología forman una función especifica en el FE.

Marco de referencia o Especificaciones del protocolo ForCES (Framework): Este protocolo está diseñado para ser utilizado entre un Elemento de Control (CE) y un Elemento de Reenvío (FE) dentro de un Elemento de Red (NE).

Network Element (NE): Elemento de Red (NE), Es una entidad compuesta de uno o más CEs y FEs. Para las entidades que están fuera del NE, el NE representa un único punto de gestión. Del mismo modo, un NE suele ocultar su organización interna de las entidades externas.

Plano de Control: En este plano el CE es responsable de operaciones como la señalización, el procesamiento del protocolo de control y la implementación de los protocolos de administración. Basado en la información adquirida a través del procesamiento de control, los CEs dictan el comportamiento de los paquetes procesados por los FEs, utilizando el protocolo de interconexión. Por ejemplo, el CE puede controlar a un FE por medio de la manipulación de las tablas de enrutamiento, el estado de sus interfaces o la adición y remoción de un NAT (Network Address Translation, Traducción de Direcciones de Red).

Plano de Datos: En este plano el FE es responsable por el procesamiento y el manejo de los paquetes. Con el objeto de lograr independencias entre los planos de control y de datos, los diferentes tipos de FEs pueden ser desarrollados, algunos de propósito general y otros más especializados. Precisamente, algunas de estas funciones que pueden ser desarrolladas en los FEs incluyen el reenvió de paquetes de capa 3, medición, formato, cortafuegos, NAT, encapsulación, des encapsulación, encriptación, contabilidad, etc. Todas las combinaciones de estas funciones pueden estar presentes en la implementación práctica de los FEs.

Post-association Phase: Fase de Post-asociación, Es el período de tiempo durante el cual un FE sabe cuál CE es su controlador y viceversa, incluido el tiempo durante el cual el CE y el FE establecen una comunicación el uno con el otro[1].

Pre-association Phase: Fase de Pre-asociación, Es el período de tiempo durante el cual un administrador de FE y un administrador de CE, están determinando que FE y CE debe ser parte del elemento de la misma red.

Punto Lógico FP : Interfaz dedicada a la comunicación entre el CE y el FE, donde se implementa el protocolo ForCES.

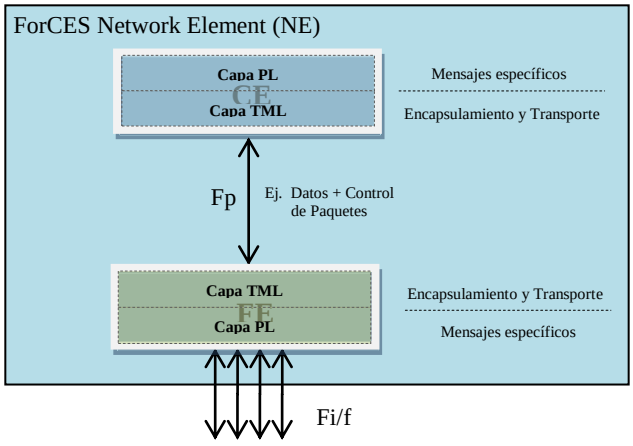


Figura 3. Interacción entre el la Arquitectura y el Framework del protocolo ForCES

XML: Lenguaje de marcas extensible, es la manera como se escribe y describen las operaciones a través atributos que contiene un LFB, es similar a una base de datos que se consulta para saber cómo opera un LFB especifico.

RESUMEN

Este proyecto de grado propone la implementación del protocolo ForCES, el cual está centrado en la conexión del punto lógico **FP** de la arquitectura ForCES, y es estrictamente una comunicación entre el CE y el FE.

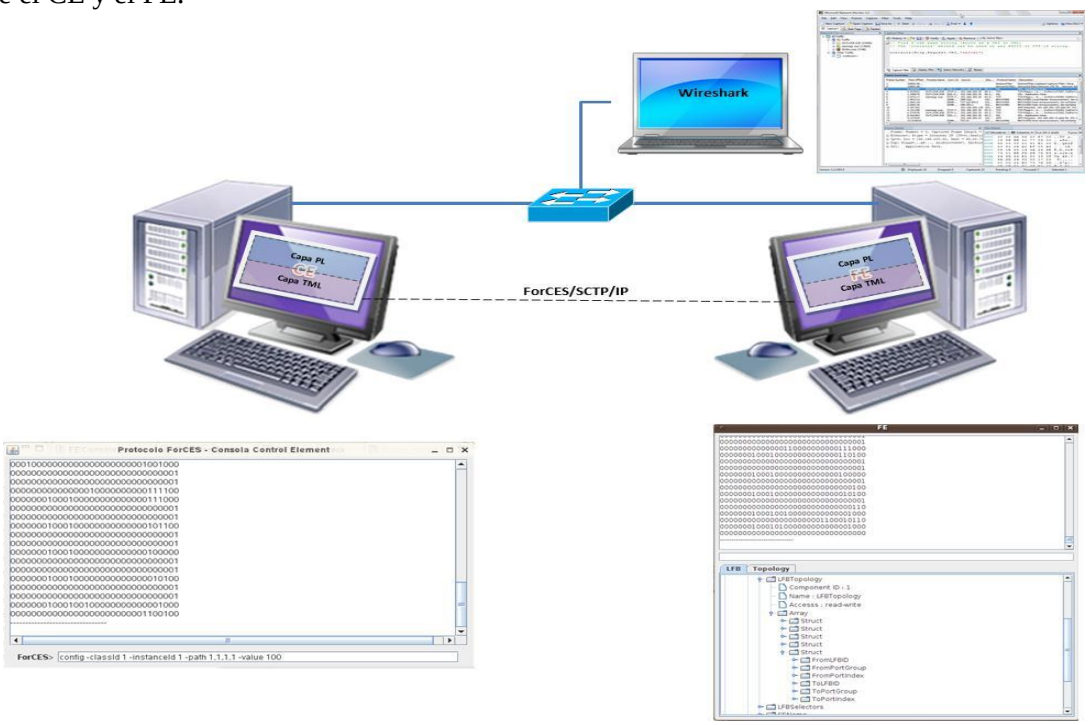


Figura 4. Implementación del protocolo ForCES

Dicha estructura del protocolo está establecida dentro de las especificaciones emitidas y definidas por el grupo de trabajo ForCES, y consiste en formar un protocolo que transporte la información del plano de datos disponible en los FE y llevarlos al plano de control en el CE aprovechando la arquitectura flexible de ForCES.

Este proyecto plantea la implementación del protocolo ForCES, sobre la arquitectura ForCES, a través de una simulación en Java que le permite al investigador tener la posibilidad de comparar funcionalidad típica de un enrutador convencional con el enrutador basado en la arquitectura ForCES que se plantea, y de esta manera aprovechar la ventaja de esta arquitectura para reprogramar dichas funcionalidades.

Este proyecto busca que mediante un ejemplo básico de LFBs descubiertos por el CE e Informada por el FE, se pueda probar el funcionamiento del protocolo ForCES que los intercomunica, permitiendo comprobar que los datos que son enviados de un extremo a otro, bajo las condiciones expuestas en las especificaciones, son correctas y se han programado sin errores.

En este programa de prueba el CE usa los diferentes tipos de topologías LFBs y construye diferentes funcionalidades con los atributos disponibles en los FEs, es importante aclarar que estas dos emulaciones están fuera del alcance de este proyecto, sin embargo las interfaces correspondientes están disponibles para continuar con su desarrollo.

1. INTRODUCCION

Lograr una comunicación extremo a extremo es uno de los retos de las redes IP, y por ende es el éxito de la gran expansión del internet en el mundo. Según este principio, la mayoría de las funcionalidades de un nodo de red son esencialmente dirigidas al enrutamiento de paquetes, u otras tareas en las que la mejora del rendimiento así lo justifique. A pesar de que esta idea está ampliamente aceptada, actualmente existe una clara tendencia a incrementar el procesamiento que realizan los nodos de red, dando mayor importancia a los siguientes servicios: Firewalls (Muro de Fuego o Cortafuego, Herramienta de seguridad que controla el tráfico de entrada/salida de una red), servidores NAT (*Network Address Translation*, Traducción de Direcciones de Red), (*Web Proxies*, Servidor de interceptación) y balanceadores de carga entre otros.

Sin embargo, este requerimiento de desarrollar nuevos servicios de red, tales como la calidad de servicio, y otros, no son fáciles de implementar con la arquitectura que poseen los nodos de las redes actuales. Estos sistemas están cerrados e integrados de manera vertical, esto significa que el fabricante distribuye tanto el hardware que se encarga de las comunicaciones y de los datos, como el software que implementa el plano de control, sin posibilidad de modificación. Esta dependencia con el fabricante limita la interoperabilidad y dificulta el desarrollo de nuevos servicios, puesto que para que puedan ser difundidos deben pasar por un largo y arduo proceso de estandarización y pruebas antes de ser aprobado por el fabricante. Esta es una de las causas por las que mecanismos relativamente recientes, como IPv6, todavía no han podido ser implantados a gran escala.

Ante esta situación, se inició la investigación en: redes programables y sistemas con arquitecturas flexibles, las cuales surgieron inicialmente como dos líneas de investigación paralelas, pero que ahora están interrelacionadas mutuamente entre sí, formando una sola.

Las redes programables, son principalmente redes activas con interfaces abiertas, que permiten resolver necesidades de procesamiento en los nodos intermedios de red y su objetivo es reducir al mínimo el tiempo de implantación de nuevos protocolos y servicios, a partir de la definición de una arquitectura común y flexible de red en la que los sistemas de enrutamiento pudiesen ser reprogramados, y de esta forma adaptar el comportamiento de la red a las condiciones de crecimiento y cambio de tecnología, sin afectar el procesamiento.

Respecto a lo anterior, los organismos de estandarización han comenzado a definir interfaces estándar y abiertas entre los diferentes elementos y capas que forman un nodo de red. De esta forma se pretende fomentar la interoperabilidad entre fabricantes y simplificar el desarrollo de nuevos protocolos, de modo que podrían adaptarse rápidamente a los sistemas de diferentes fabricantes. En síntesis, lo que se defiende, es la idea del diseño de nodos de red programables con capacidades extendidas y escalables.

Bajo este enfoque y este planteamiento, el siguiente proyecto de grado busca utilizar esta nueva propuesta orientada a arquitecturas flexibles y reprogramables, usando la arquitectura ForCES e implementando el protocolo ForCES, definida y creada por el IETF a través del grupo de trabajo *Forwarding and Control Element Separation* (ForCES).

2. OBJETIVOS

2.1. OBJETIVO GENERAL

Implementar en la plataforma Java "el Protocolo de comunicación ForCES" definido y creado por la IETF a través del grupo de trabajo, *Reenvío and Control Element Separation* (ForCES), bajo la "Arquitectura ForCES" usando la especificación que ellos suministran llamada draft-doria-forces-protocol-01.

2.2. OBJETIVOS ESPECIFICOS

1. Implementación del protocolo de comunicación ForCES según lo describe el RFC 5811 en la plataforma Java, indispensable para comunicar los dos elementos del NE que se simularán, y que para este trabajo de grado se compondrá solo de un CE y un FE, de acuerdo a las especificación draft-doria-forces-protocol-01 referente a la arquitectura ForCES.

2. implementar las funciones básicas de comunicación, que le permitan al protocolo ForCES, soportar los requerimientos de modificación de atributos y encadenamiento de los LFBs que controlan las funcionalidades del FE desde los CE, y de esta manera brindar una flexibilidad tal, para que este protocolo a futuro pueda transportar los cambios de funciones mas robustas que se programen en el CE y desde el cual se cambia las funciones del FE, cumpliendo con lo sugerido por el RFC 5810.

3. Simular las funciones básicas de un FE y un CE, mediante dos interfaces de usuario llamadas consolas, desarrolladas en la plataforma Java e instaladas en dos PCs respectivamente, que permitan al protocolo ForCES interactuar con los FE y CE emulando una conexión completa, y de este modo verificar la operación del protocolo ForCES cuando el CE cambia las funcionalidades del FE.

4. Definir y aplicar un esquema de evaluación de la funcionalidad del protocolo desarrollado, que, fase a fase, permita verificar que efectivamente lo implementado corresponde a lo establecido en las especificaciones RFC 5810 y el RFC 5811. Esta verificación fase a fase se realizará por parte del director del proyecto y de un asesor externo.

3. MARCO TEORICO

3.1. ARQUITECTURA DEL ENRUTADOR CONVENCIONAL

Un enrutador (*router*), es un dispositivo electrónico, con capacidad para distribuir cada paquete de información que recibe y decidir la manera más conveniente de enviarlo a su destino. Un enrutador está encargado de desempeñar dos tareas fundamentales: Enrutamiento y Reenvió de paquetes. [10]

3.1.1. ENRUTAMIENTO

El proceso de enrutamiento construye una vista de la topología de la red y calcula las mejores rutas, para que un paquete llegue a su destino, esto lo hace mediante la información que se intercambia entre enrutadores vecinos, usando diferentes protocolos de enrutamiento. Las mejores rutas son almacenadas en una estructura llamada TABLA DE REENVÍO.

3.1.2. REENVÍO DE PAQUETES (*Forwarding*)

El proceso de Reenvío "*Forwarding*" de paquetes, mueve un paquete desde una interfaz de entrada de un enrutador, a una interfaz de salida apropiada, basada en la información contenida en la tabla de Reenvío.

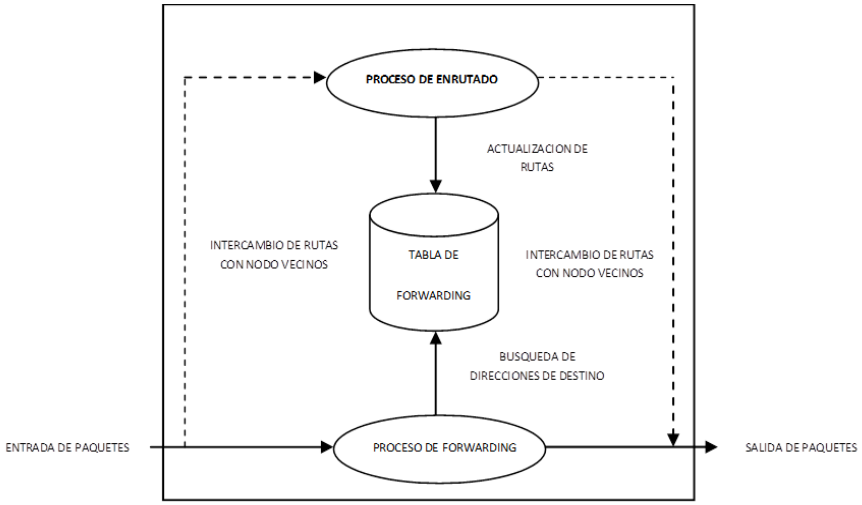


Figura 1. Proceso de forwarding

3.1.2.1. Reenvío

El desempeño del proceso de Reenvío, es el que determina el desempeño total del enrutador. Este proceso se divide en dos grupos principalmente.

3.1.2.2. Reenvío Básico

Define el mínimo conjunto de funciones que un enrutador que debe implementar en orden, para transmitir paquetes entre interfaces.

3.1.2.3. Reenvío Complejo

Son funciones que representan el proceso adicional, requerido por los enrutadores, dependiendo del ambiente donde se use.

3.1.2.4. Funciones del Reenvío básico

Para enviar un paquete IP desde una interfaz de entrada a una de salida, un enrutador implementa las siguientes funciones básicas:

- **Validación del encabezado IP**

El enrutador debe comprobar que cada paquete IP que llega necesita ser validado, se verifica que el paquete este correctamente formado, para enviarlo al siguiente nivel, mientras que los otros son descartados. Se verifica la versión del protocolo, la longitud del encabezado, el cálculo del *checksum* y se determinar la existencia de algunas otras funciones.

- **Control del tiempo de vida del paquete**

Los enrutadores deben decrementar el tiempo de vida del paquete, esto se realiza en el campo TTL del encabezado IP del paquete, esto permite evitar que los paquetes queden atrapados en los lasos (*loops*) de enrutamiento. Si el valor del TTL es 0 o negativo, el paquete es descartado y un mensaje ICMP es generado y enviado al remitente original.

- **Recalculo del checksum**

Si el valor del TTL es modificado, el checksum del encabezado necesita ser actualizado.

- **Búsqueda de ruta**

La dirección de destino del paquete, es usada para buscar en la tabla de Reenvío el puerto de salida. El resultado de esta búsqueda, indicará si el paquete es enviado al enrutador, a un puerto de salida (*unicast*) ó a múltiples puertos de salida (*multicast*).

- **Fragmentación**

Es posible que la máxima unidad de transmisión (MTU) del enlace de salida, sea más pequeño que el tamaño del paquete que necesita ser transmitido. Esto significa que el paquete necesita ser dividido en múltiples fragmentos antes de ser transmitido.

- **Operaciones Ip**

Indica que hay un procesamiento especial que requiere el paquete para ser transmitido.

3.1.2.5. Funciones de Reenvío complejas

Las siguientes son las funciones complejas de Reenvío que deberán estar implementadas en un enrutador:

- **Clasificación de paquetes**

Para distinguir los paquetes, un enrutador debe necesitar examinar no solamente la dirección IP de destino, sino otros campos como dirección fuente, puerto de destino y puerto de origen. El proceso de diferenciación de paquetes y aplicación de acciones necesarias, acordadas a ciertas reglas es conocido como clasificación de paquetes.

- **Traducción de paquetes**

Como las direcciones IPV4 públicas se han ido agotando, existe la necesidad de mapear varios host a una dirección IP pública. Entonces, un enrutador que actúa como una puerta de enlace (*gateway*) para una red, necesita soportar traducción de direcciones NAT, el cual mapea direcciones IP públicas a direcciones IP privadas y viceversa. Un enrutador requiere mantener una lista de host conectados y sus direcciones locales y traducir los paquetes de entrada y salida.

- **Priorización de tráfico**

Un enrutador, debe garantizar cierta calidad de servicio (QoS) según el cierto acuerdo de nivel de servicio (SLA). Esto involucra la aplicación de diferentes prioridades de flujo de datos, a diferentes clientes que proveen un nivel de desempeño, acorde con el acuerdo de servicio predeterminado.

Esto implica por ejemplo, el número de paquetes debe ser entregado a una tasa constante, el cual es necesario para el video multimedia en tiempo real (IPTV) y VoIP.

3.1.3. TIPOS DE ENRUTADOR

Los enrutadores pueden tener un grado de complejidad diferente, basados en el sitio de la red donde se encuentre ubicado y de cuanto tráfico ellos necesitan mantener. Por tal motivo se clasifican en tres tipos principales:

3.1.3.1. Enrutadores de nucleo

Son usados por los proveedores de servicio, para interconectar unas pocas miles de redes, de tal manera que el costo del movimiento de tráfico es compartido entre grandes clientes.
Desde que el tráfico llega al enrutador, este es altamente agregado y debería ser capaz de sostener una gran cantidad de tráfico, por lo tanto, un enrutador de nucleo debe tener los requerimientos principales de alta velocidad y fiabilidad.

La velocidad a la cual un enrutador puede reenviar paquetes, está en la mayoría de los casos limitado por el tiempo que se gasta en buscar una ruta en la tabla de Reenvío. Las rutas de nucleo (*core*), forman los nodos críticos de la red, es esencial, que estos enrutadores no fallen bajo ninguna condición. La confiabilidad de un enrutador depende de la confiabilidad de los elementos físicos, tales como, tarjetas, estructuras del conmutador (*switch*) y tarjetas de control de procesamiento de ruteo. La confiabilidad de estos elementos físicos, es alcanzada por una completa redundancia, fuentes de poder duales, conmutación en espera (*stand byk*), tarjetas duplicadas y tarjetas de control de procesamiento de rutas.

3.1.3.2. Enrutador de borde.

Conocidos también como enrutador de acceso, son desplegados en el borde de la red del proveedor de servicio, para conectar a los clientes desde el hogar y negocios pequeños. La necesidad cada día de tener un mayor ancho de banda, ha permitido que se introduzca una gran variedad de tecnologías de acceso, moduladores (*modems*) de alta velocidad, xDSL y los cables de los moduladores. Estos enrutador necesitan implementar lo más recientes protocolos, tales como, protocolo de túnel punto a punto (PPTP), protocolo punto a punto sobre Ethernet (PPPoE), IPSec que soportan las VPNs. Como característica fundamental, este tipo de enrutadores deben ser capaces de soportar una gran cantidad de tráfico.

3.1.3.3. Enrutadores empresariales

Interconectan sistemas finales localizados en compañías, universidades, etc. Ellos proveen continuidad a muy bajo costo a un gran un número de sistemas finales. En conclusión, un requerimiento deseable, es el de permitir diferenciación de servicios para proveer QoS, garantizado para diferentes departamentos de una empresa.
Una red típica de una empresa, está construida usando muchos segmentos Ethernet interconectados por concentrador (*hubs*), puentes y conmutadores (*switches*), pero una red que se construye con esos elementos económicos, tienden a degradar el desempeño, en cuanto incrementa el tamaño de la red. Adicionalmente, estos enrutadores requieren un soporte eficiente para tráfico multidifusión (*multicast*) y difusión (*broadcast*), como aplicaciones de video, que es el más predominante en las empresas. De igual manera, necesitan implementar muchas tecnologías viejas que se emplean aún en las empresas. Como estos enrutadores deben de conectar muchas redes LAN, se requiere que estos puedan soportar un gran número de puertos.

3.1.4. ELEMENTOS DE UN ENRUTADOR.

Un enrutador puede ser visto desde dos perspectivas diferentes:

- **Funcional**

Puede ser lógicamente visto, como una colección de módulos, donde cada modulo implementa un conjunto de funciones relacionadas, para lograr el objetivo total del reenvió de paquetes.

- **Arquitectural**

Un enrutador puede ser considerado como una interconexión de diferentes tipos de tarjetas, que corren con un software especializado.

3.1.4.1. Perspectiva funcional

Un enrutador puede ser dividido en principalmente en seis módulos funcionales que implementa varios requerimientos. Estos son:

- **Interfaces de red**

Contienen muchos puertos, que proveen la conectividad para un enlace físico de red. Un puerto es un enlace físico en el enrutador y sirve como entrada y salida para los paquetes entrantes y salientes respectivamente (por ejemplo: puerto Ethernet o SONET).

La interface de red tiene las siguientes funciones:

- Entiende varios protocolos de estado de enlace, así que cuando el paquete llega, se puede des encapsular, quitando el encabezado de nivel 2 (L2).
- Extrae el encabezado IP (nivel 3) y lo envía al "motor Reenvío" para buscar la ruta, mientras el paquete entero es almacenado en la memoria.
- Provee la funcionalidad de encapsulación L2 antes que el paquete sea enviado a la salida.

- **Núcleo de Reenvío (Motor de Reenvío)**

Es responsable de decidir a cual interface de red el paquete que entra debería ser reenviado. Cuando un puerto recibe un nuevo paquete, des encapsula el encabezado L2 y envía el paquete entero IP o simplemente el encabezado del paquete al motor de Reenvío.
El motor de Reenvío consulta una tabla, por ejemplo, participa en una función de búsqueda de ruta y determina a cual interface de red el paquete debería ser enviado, esa tabla se conoce como Información básica de Reenvío o simplemente tabla de Reenvío, dependiendo de la arquitectura, la búsqueda puede ocurrir en el hardware del cliente o en una caché de rutas locales en la tarjeta de línea. Además de proveer garantías de QoS, el motor Reenvío puede necesitar clasificar paquetes dentro de una clase de servicio predefinido.

- **Gestor de colas**

Este componente tiene buffers para almacenamiento temporal de los paquetes cuando un enlace saliente desde un enrutadores está saturado. Cuando los buffers se desbordan debido a la congestión de la red, el gestor de colas selectivamente extrae los paquetes. La responsabilidad de este componente, es gestionar la ocupación de las colas e implementar políticas acerca de la extracción de los paquetes cuando hay demasiado encolamiento.

- **Gestor de trafico**

Este componente, es responsable de priorizar y regular el tráfico de salida, dependiendo del nivel de servicio deseado, esto es necesario en los enrutadores que transportan tráfico desde diferentes suscriptores y es importante, para asegurar que ellos consigan el nivel de servicio por el cual ellos pagaron. Cuando un enrutador recibe tráfico desde un suscriptor, el gestor de tráfico asegura que no se acepten más de lo que se especifico en el contrato.

- **Tarjeta de conexión de módulos (Backplane)**

Provee la conectividad para las interfaces de red, así que los paquetes desde una interface de red entrante pueden ser transferidos, a una tarjeta de interface de red saliente. La tarjeta de conexión puede ser, ya sea compartido, donde solamente dos interfaces pueden comunicarse en algún instante o conmutar, donde múltiples interfaces pueden comunicarse simultáneamente. El ancho de banda total de todas las interfaces adjuntas define el ancho de banda requerido por la tarjeta de conexión.

- **Procesado de control de rutas**

Es el responsable de implementar y ejecutar los protocolos de enrutamiento. Mantiene una tabla de enrutamiento, que es actualizada siempre que ocurre un cambio en el enrutador. Con base en el contenido de la tabla de enrutamiento, la tabla de Reenvío es calculada y actualizada. En adición, también corre el software para configurar y gestionar el enrutador.

3.1.4.2. Perspectiva arquitectural

En cuanto a la perspectiva arquitectural que puede tener un enrutador, se mencionan a continuación las principales:

- **Tarjeta de puertos**

Implementan las interfaces de red. Cada tarjeta de puerto, es capaz de sostener solamente un medio especifico, por ejemplo, una tarjeta de puertos soportaría solamente Ethernet, mientras que otra podría soportar solamente SONET. Estas tarjetas contienen procesamiento lógico L2, que entiende el formato de paquetes L2 para este medio específico.

- **Tarjetas de línea**

Implementa una gran mayoría de los componentes funcionales, Reenvío motor, gestor de colas y gestor de tráfico. Analiza la carga útil IP y usa los contenidos del encabezado para tomar decisiones acerca del Reenvío, colas y descarte de paquetes durante periodos de congestión del enlace. También contiene buffers de memoria, para el almacenamiento de paquetes durante el procesamiento y encolamiento.

- **Tarjetas de conmutación**

Mientras una línea de tarjeta implementa las funciones de procesamiento de paquetes, una tarjeta de conmutación sirve como tarjeta de conexión para transferir paquetes, desde una tarjeta de entrada a una de salida.

- **Tarjetas de procesamiento de ruteo**

Los protocolos de enrutamiento y el software de gestión corren sobre estas tarjetas. En enrutadores de gama alta, estas tarjetas usan procesadores de propósito general con gran cantidad de memoria.

3.1.4.3. Funcionalidades de un enrutador IP

En resumen los enrutadores consisten en los siguientes componentes básicos:

Varias interfaces de red para acoplar redes, módulos de procesamiento, módulos de almacenamiento y una unidad de interconexión interna (o una estructura de conmutación). Típicamente, los paquetes son recibidos en una interface de entrada, procesados por el modulo de procesamiento y posiblemente almacenadas en el modulo de almacenamiento, luego ellos son enviados a través de una unidad de interconexión interna a una interface de salida que los transmite al próximo salto y de allí a su destino final. Como se muestra en la figura, el enrutador opera en dos planos diferentes:

- **Plano de Control**

Es el plano donde el enrutador informa cual es la interface de salida apropiada para la transmisión de los paquetes a determinados destinos.

En el Plano de Control de procesamiento se construye la tabla de enrutamiento o base de información de enrutamiento (RIB). El RIB podrá ser utilizado por el plano de Reenvío para buscar la interface externa para un determinado paquete, o, en función de la implementación del enrutador.

El Plano de Control construye la tabla de enrutamiento con el conocimiento de sus interfaces locales, de los códigos de rutas estáticas, y del intercambio de información de los protocolos de enrutamientos con otros enrutadores. La tabla de enrutamiento almacena las mejores rutas a determinados destinos de la red, las métricas de enrutamiento asociados con esas rutas y el camino al próximo enrutador.

Los enrutadores mantienen el estado de las rutas en la tabla de enrutamiento, pero esto es muy diferente a mantener el estado de los paquetes individuales que se han transmitido.

- **Plano de Reenvío**

El plano de Reenvío se conoce también como Plano de datos. Por la función de Reenvío del protocolo de internet (IP), el diseño de los enrutadores procura reducir a un mínimo el estado de la información almacenada sobre los paquetes individuales. Una vez que se envía un paquete, el enrutador no debe mantener más que la información estadística del envío. Es en el punto final del envío en el que se mantiene la información sobre errores.

Entre las decisiones más importantes del Reenvío está decidir qué hacer cuando se produce congestión. En este plano también se construye la tabla de Reenvío, que es consultada por el enrutador para determinar la interface de salida donde necesita reenviarse un paquete entrante, entonces cada entrada en la tabla de Reenvío mapea un prefijo IP a una interface de salida. Dependiendo de la implementación, las entradas deben contener información adicional tal como direcciones MAC para el próximo salto y estadísticas acerca del número de paquetes reenviados a través de una interface.

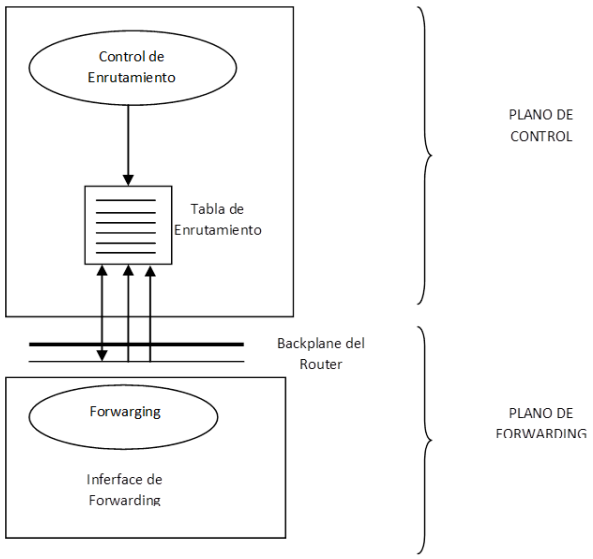


Figura 5. Planos de datos y control

4. ESTADO DEL ARTE

En el 2001 el IETF[1] creó el grupo de trabajo llamado ForCES para definir un protocolo estándar para la comunicación entre el plano de datos y el plano de control de los enrutadores IP basados en arquitecturas distribuidas, flexibles y reprogramables, que usa el mismo nombre ForCES (*Forwarding and Control Element Separation*), y que ha ido evolucionando en la medida que este grupo se reúne para discutir y generar estándares. En este proyecto las principales fuentes de información fueron los RFCs[2] 5810, 5811, 5812[3] y los draft-ietf-forces-interoperability-04 y draft-ietf-forces-lfb-lib-05.

Inicialmente, y a lo largo del proyecto se utilizaron los draft[4] de la arquitectura ForCES donde se publicaban los avances y las mejoras de lo que hoy conocemos como protocolo ForCES, y los cuales al finalizar el trabajo se convirtieron en RFCs. Estos draft permitieron el planteamiento de la idea de esta implementación, leyendo y siguiendo al pie de la letra cada una de sus indicaciones para su elaboración.

En la revisión bibliográfica, se encontró muy poco material que hablará de manera clara y específica sobre el protocolo ForCES, así que se procedió a usarcé como únicos documentos base los RFCs y *draft* emitidos por el IETF, en su sitio web. Algunos otros documentos encontrados solo daban algunos indicios de cómo elaborar la implementación del protocolo mostrando una parte de la interfaz grafica desarrollada.

[1] IETF: Regula las propuestas y los estándares de Internet, conocidos como RFCs, y se encarga de que la arquitectura de Internet y los protocolos que la conforman funcionen correctamente.

[2] RFC: Peticiones de comentarios, son los documentos que describen y especifican implementaciones, estándares y discusiones, sobre las normas referentes a las tecnologías y los protocolos relacionados con Internet y las redes.

[3] RFC5810: Protocol Specification, Forwarding and Control Element Separation (ForCES)
RFC5811: SCTP-Based Transport Mapping Layer (TML) for the Forwarding and Control Element Separation (ForCES) Protocol.
RFC5812: Forwarding Element Model, Forwarding and Control Element Separation (ForCES)
draft-ietf-forces-interoperability-04: ForCES Interoperability Draft
draft-ietf-forces-lfb-lib-05: ForCES Logical Function Block (LFB) Library

[4] draft: Documento preliminary al RFC

5. DISEÑO DEL SOFTWARE DEL PROTOCOLO

5.1.1. DIAGRAMA DE CLASES

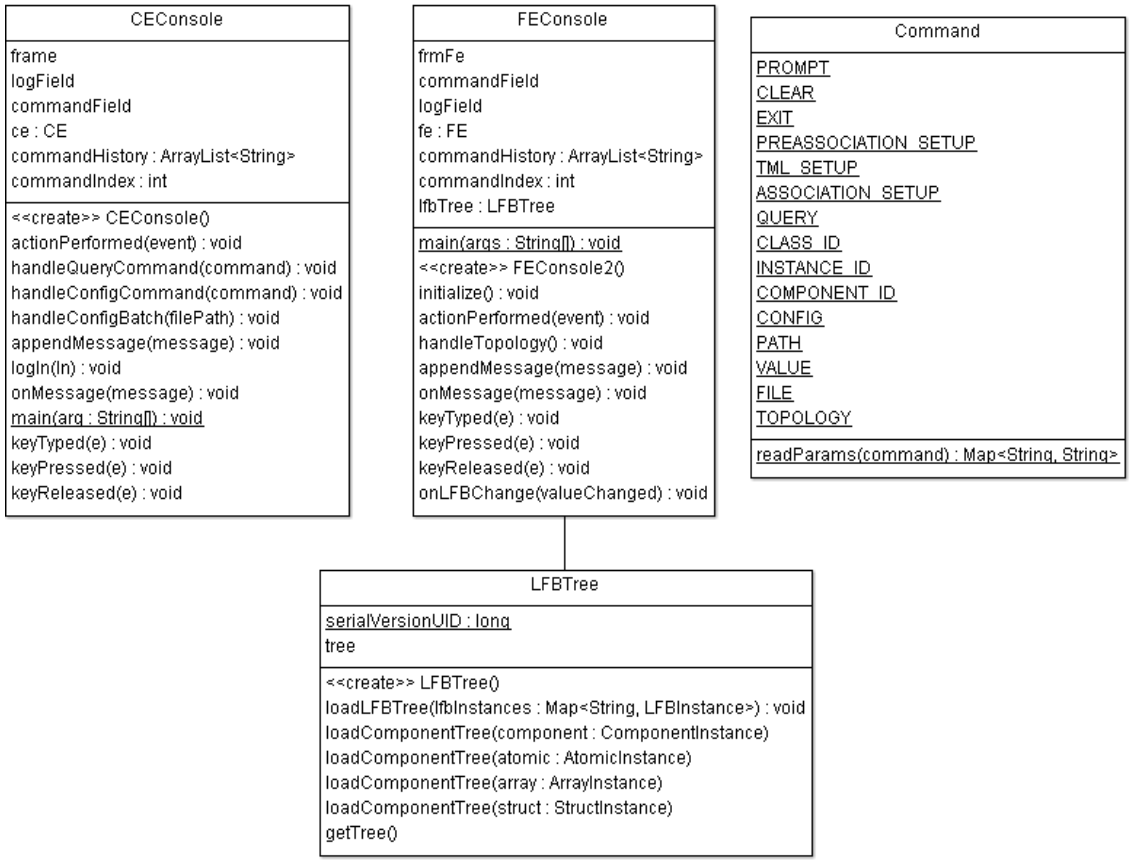


Figura 6. Diagrama de Clases del Paquete Consola

El paquete *Console*, contiene las clases de la interfaz gráfica de la herramienta que permite interactuar con el protocolo ForCES que se diseña, haciendo uso de sus respectivos métodos:

CEConsole: Consola del Elemento de Control, *Control Element (CE)*
FEConsole: Consola del Elemento de Reenvío, *Forwarding Element (FE)*
Command: Clase manejadora de los comandos usados en las consolas
LFBTree: Clase auxiliar para el manejo de los árboles que representan la estructura de los LFBs

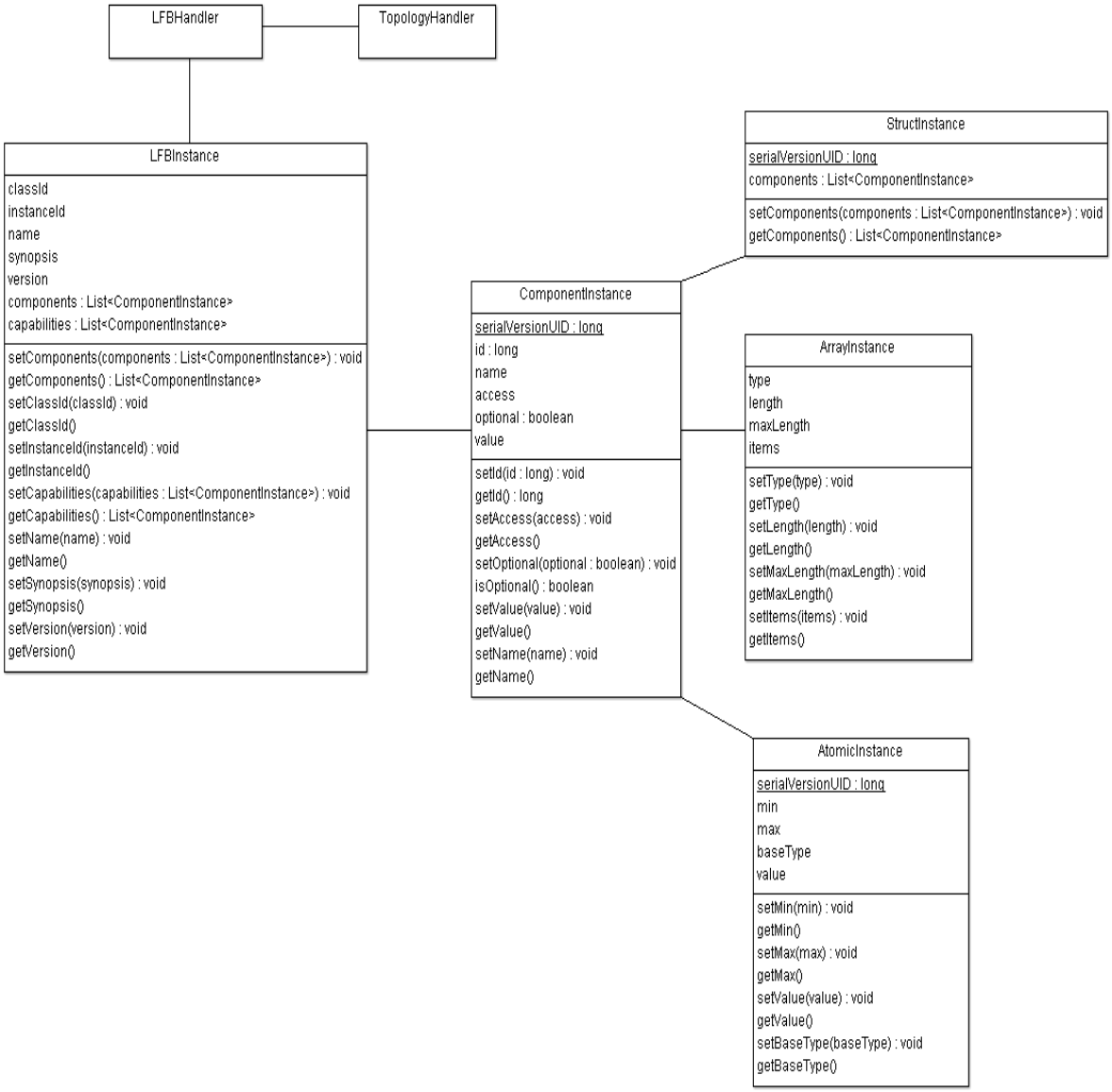


Figura 7. Diagrama de Clase del Paquete LFB

Paquete que administra el modelo LFB de la arquitectura ForCES, está compuesto por clases manejadoras del modelo y clases que representan instancias de los diferentes tipos de datos: *component*, *struct*, *array* y *atomic type*.

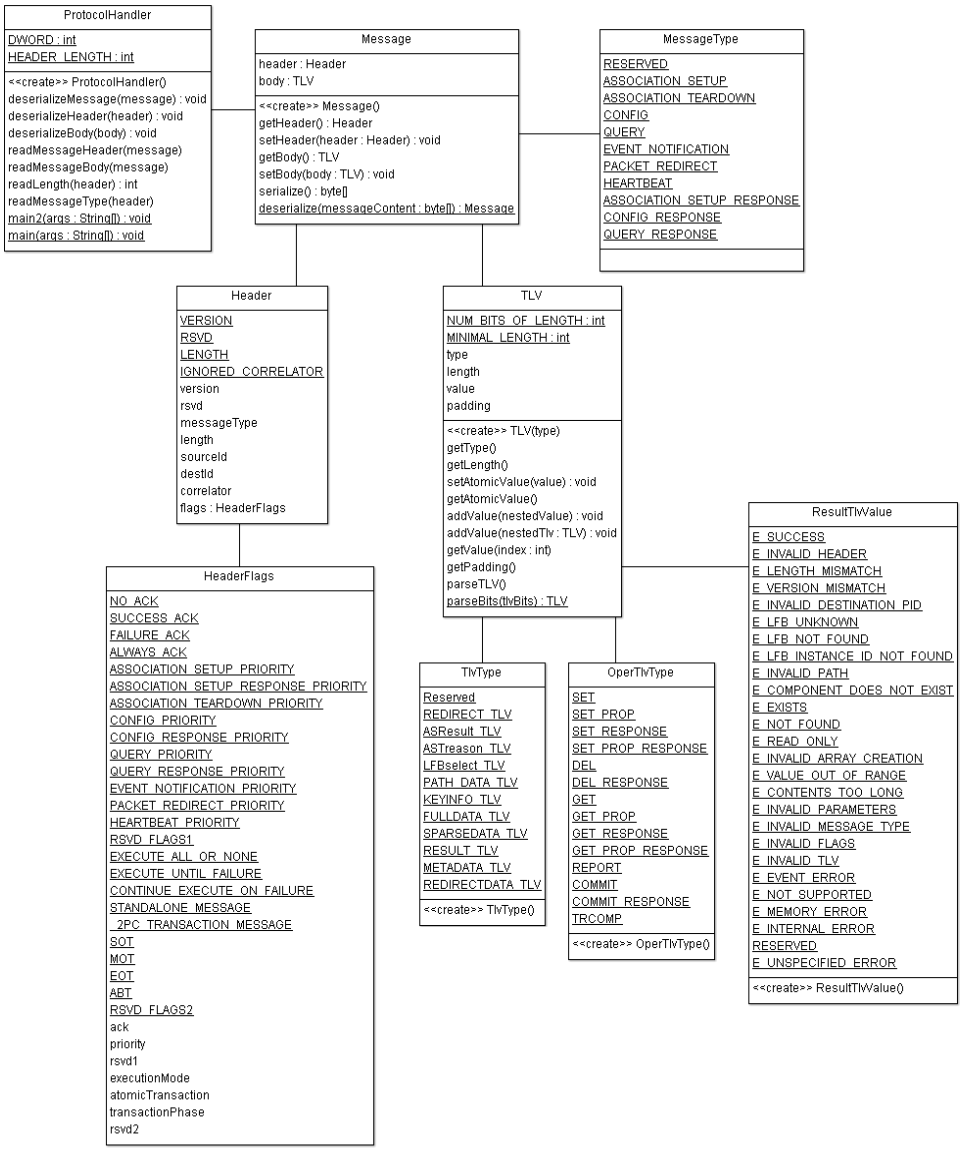


Figura 8. Diagrama de Clase del Paquete PL

Paquete controlador de toda la capa de protocolo ForCES (PL, *Protocol Layer*), tiene una clase manejadora del protocolo y otras que representan todos los elementos de la mensajería ForCES: *Message*, *Header*, *TLVs*, *flags*, etc.

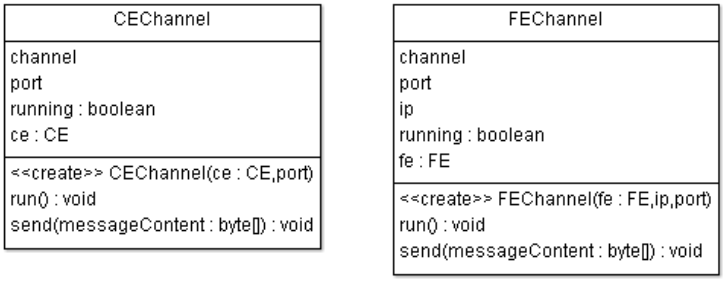


Figura 9. Diagrama de Clase del Paquete TML

Paquete que encapsula la funcionalidad de la capa TML (*Transport Mapping Layer*), está compuesto por dos clases que representan los canales SCTP de los elementos de control y Reenvío.

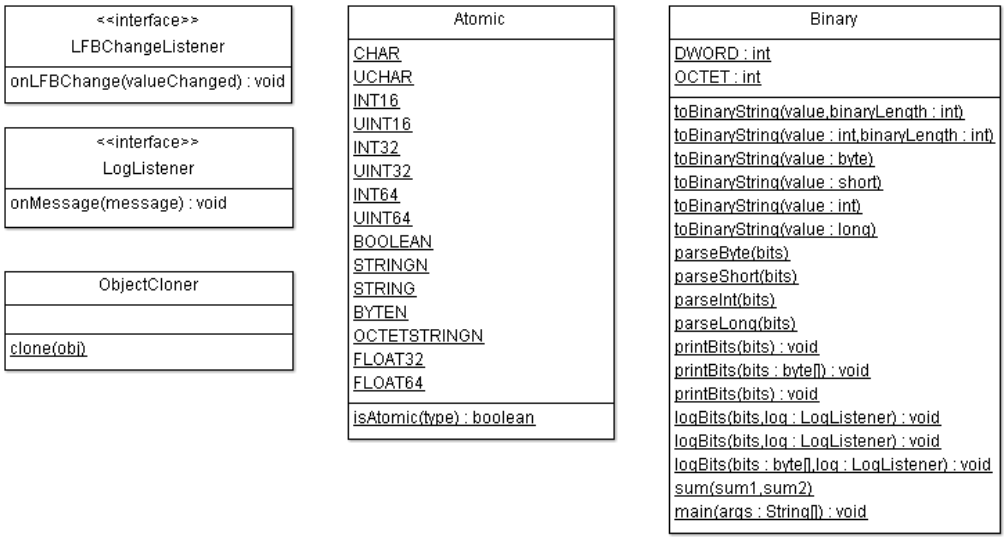


Figura 10. Diagrama de Clase del Paquete UTIL

Paquete con clases utilitarias de la herramienta:

LFBChangeListener: Interfaz que escucha los cambios que se realicen sobre los LFBs.
LogListener: Interfaz que notifica a la consola los mensajes.
ObjectCloner: Clase que permite clonar instancias de clases Java.
Atomic: Clase con constantes usadas por los tipos de dato atómicos.
Binary: Clase con funciones para la manipulación de datos binarios: sumas, conversiones, cálculo de longitudes, etc.

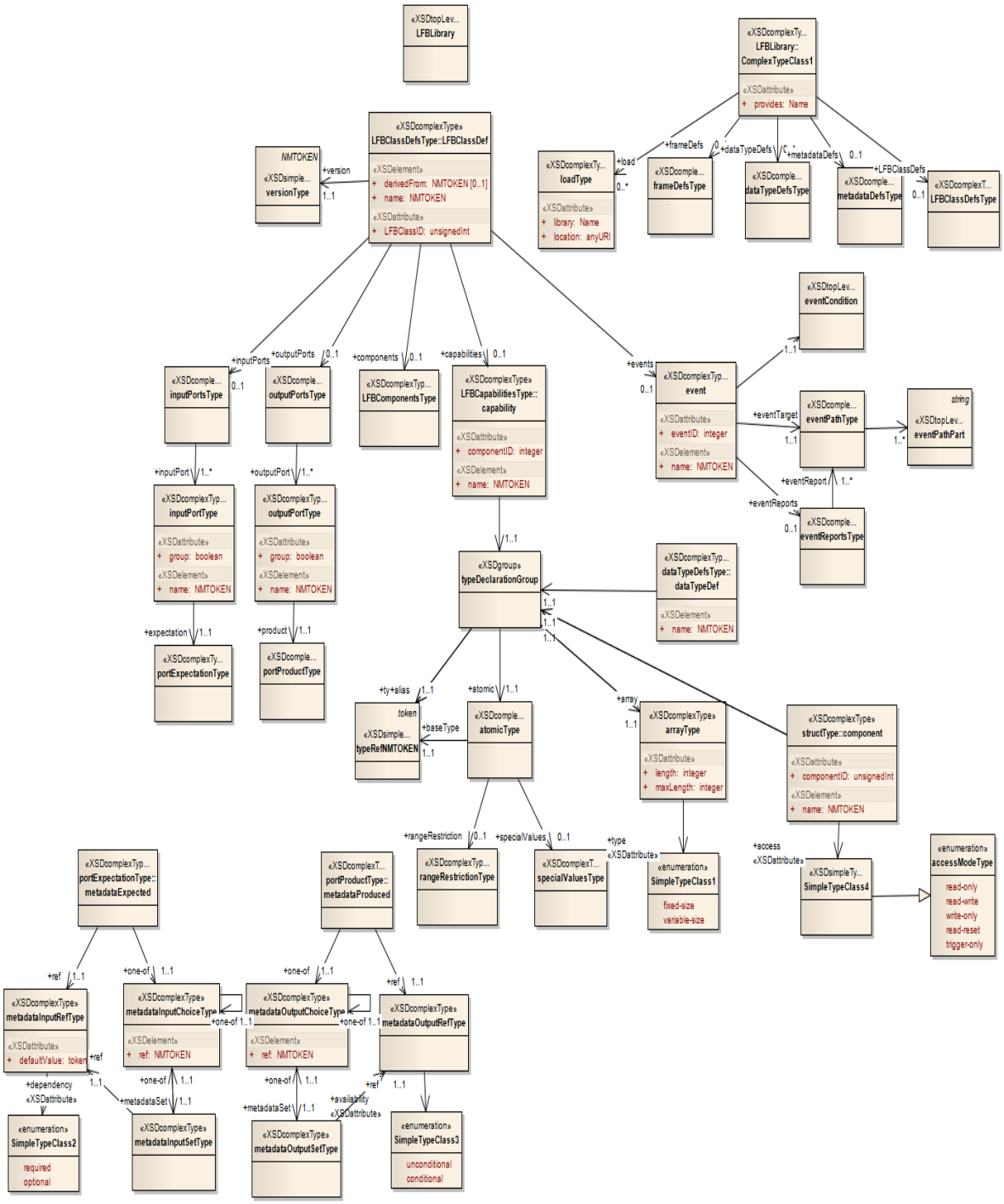


Figura 11. Diagrama de Clase de la librería "LFB *Schema*"

Diagrama del esquema XML que representa el modelo LFB del protocolo ForCES. Para mayor información ver la sección: 4.9. XML *Schema for LFB Class Library Documents* del RFC 5812.

5.1.2. DIAGRAMA DE DESPLIEGUE

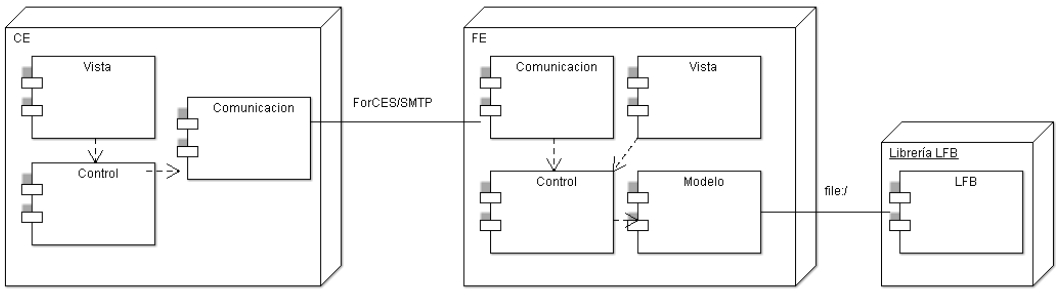


Figura 12. Diagrama de Despliegue

Vista de despliegue de la solución, está compuesta por tres componentes:
CE: aplicación Java SE que representa el elemento de control de la arquitectura ForCES, se comunica con el FE a través del protocolo ForCES sobre SCTP.
FE: aplicación Java SE que representa el elemento de Reenvío de la arquitectura ForCES, se comunica con el CE a través del protocolo ForCES sobre SCTP, y con las librerías LFB a través del protocolo de acceso a archivos.
Librería LFB: Conjunto de archivos y esquemas XML que componen la librería LFB.

5.1.3. DIAGRAMA DE PAQUETES

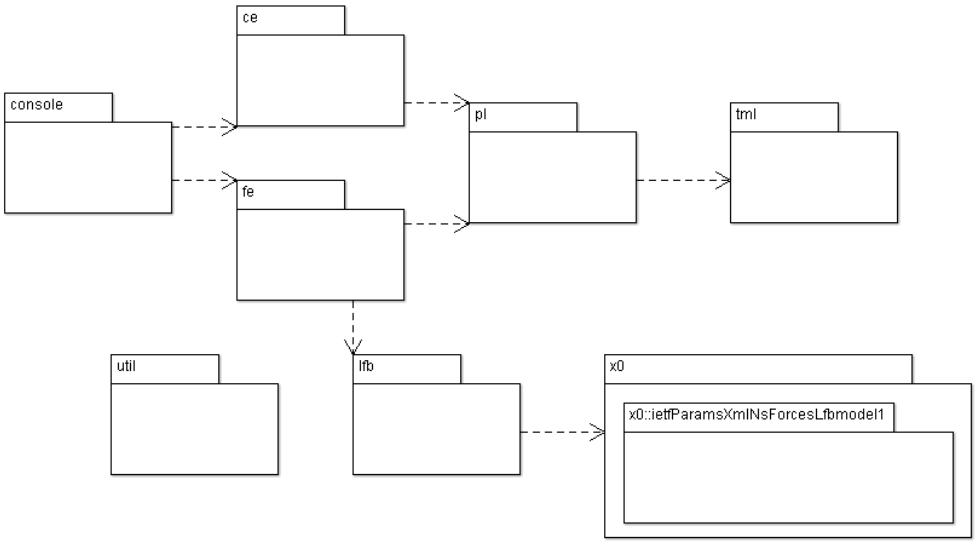


Figura 13. Diagrama de paquetes

Vista de paquetes de la arquitectura de la solución, el paquete *console* es común para los elementos de control y Reenvío, al igual que el paquete PL (*Protocol Layer*) y el paquete TML (*Transport Mapping Layer*), mientras que el paquete LFB si es de uso exclusivo del FE.

5.1.4. DIAGRAMA DE SECUENCIA

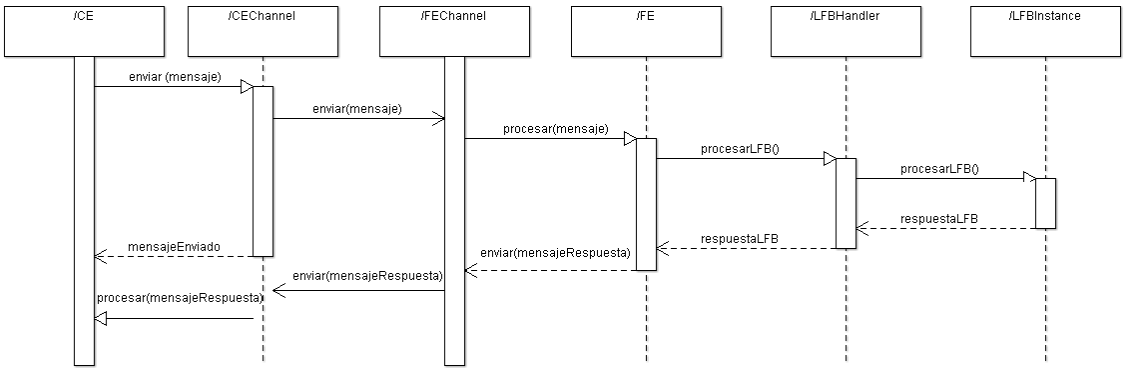


Figura 14. Diagrama de Secuencia

Diagrama de secuencia donde se evidencia la colaboración de objetos de las diferentes capas de la solución. El diagrama representa un escenario general donde se envía un mensaje desde el CE hacia el FE y el FE retorna un mensaje de repuesta al CE.

5.1.5. DIAGRAMA DE CASOS DE USO

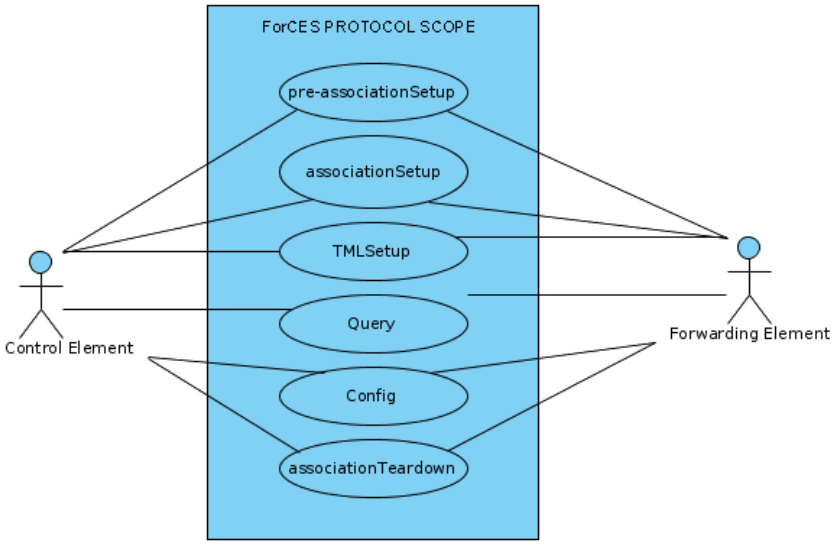


Figura 15. Casos de Uso

Este diagrama solo se usa para definir los requerimientos y solamente se describe la interacción entre los actores y el sistema, no se escribe lo que sucede dentro del sistema, el sistema es como una caja negra, por esto solo existe la interacción con el actor.

Nombre del caso de uso: *preassociationSetup*
Actores: Elemento de reenvío (FE) y Elemento de Control (CE)
Descripción del caso de uso: En este caso de uso se configuran los parámetros iniciales de los elementos de control y Reenvío.

Nombre del caso de uso: *associationSetup*
Actores: Elemento de reenvío (FE) y Elemento de Control (CE)
Descripción del caso de uso: El FE envía un mensaje de asociación al CE

Nombre del caso de uso: *TMLSetup*
Actores: Elemento de reenvío (FE) y Elemento de Control (CE)
Descripción del caso de uso: Los actores solicitan el establecimiento de los canales de comunicación.

Nombre del caso de uso: *Query*
Actores: Elemento de reenvío (FE) y Elemento de Control (CE)
Descripción del caso de uso: El CE ejecuta una consulta sobre el modelo del FE

Nombre del caso de uso: *Config*
Actores: Elemento de reenvío (FE) y Elemento de Control (CE)
Descripción del caso de uso: El CE ejecuta una configuración sobre el modelo del FE

Nombre del caso de uso: *associationTeardown*
Actores: Elemento de reenvío (FE) y Elemento de Control (CE)
Descripción del caso de uso: El CE o el FE finaliza la asociación.

6. DESARROLLO DEL SOFTWARE DEL PROTOCOLO

6.1. ARQUITECTURA FORCES

La arquitectura *Forwarding and Control Element Separation*, es una nueva propuesta para la elaboración de enrutadores de red, sus siglas en ingles *ForCES* que significa "Separación de los Elementos de Control y Reenvío", es decir, se separan los mecanismos de funcionamiento del plano de control y del plano de datos y usa el **protocolo ForCES** para comunicarse, con el propósito de que cualquier mejora que se realice en los elementos CE y FE se puedan hacer de manera separada, a diferencia de la arquitectura de los enrutadores convencionales, ésta operación entre el plano de control y el plano de datos solo la conoce el fabricante de los enrutadores, y solo el fabricante puede hacer modificaciones o mejoras.

La arquitectura ForCES es un modelo flexible escrito en un RFC público al que todos tienen acceso, y del que se puede leer el documento y realizar sus propias implementaciones, permite reprogramar y mejorar los elementos del enrutador sin intervención o permiso de los fabricantes por tratarse de una arquitectura abierta en el que un grupo de trabajo coopera entre sí para que cualquier otro investigador o usuario pueda realizar aportes a esta implementación. Este proyecto muestra la implementación del protocolo ForCES que permite la conexión y transporte de información entre los elementos CE y FE.

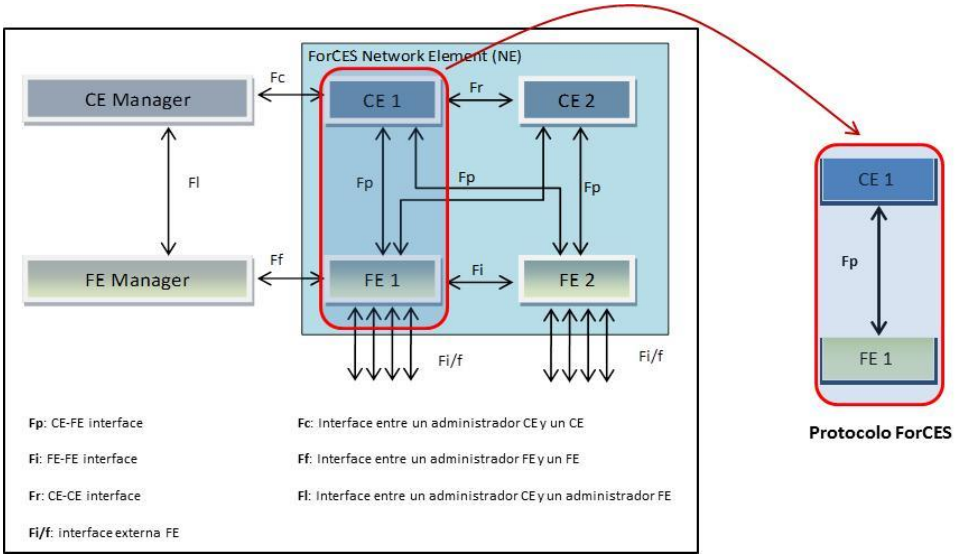


Figura 16. Arquitectura ForCES. Tomado RFC 5810, pág. 11

Dentro de la arquitectura ForCES existen varios puntos de conexión como lo muestra la figura. El punto de conexión **FP** es donde se concentrará el desarrollo de este proyecto, pues es aquí donde se implementa el protocolo ForCES. Para la implementación de este protocolo siguiendo la arquitectura, se plantea que el elemento CE, FE y el protocolo ForCES que las conecta sea

programado haciendo uso del lenguaje Java. Cualquier otro punto de conexión que se muestre en la arquitectura según la especificación **RFC5810** en la página 11, y la interacción entre el FEM y el CEM el cual es parte de la fase de pre asociación, se encuentran fuera del alcance de este proyecto.

La especificación define en su parte introductoria de la página 5 del **RFC5810** que el protocolo ForCES trabaja en modo maestro-esclavo, en donde los elementos CEs son los maestros y los elementos FEs los esclavos, y que el protocolo debe incluir comandos de transporte para configurar la información de los LFBs en el FE y todos los mensajes de asociación, configuración, consulta de estado y notificación de eventos. La arquitectura también presenta el uso del modelo FE de ForCES (ForCES FE *model*, **RFC5812**) donde se definen los bloques lógicos funcionales LFBs, usando XML. Todo el proceso de elaboración del protocolo se basa en la pila del protocolo ForCES que se ha diseñado para su implementación y que se puede apreciar en la figura.

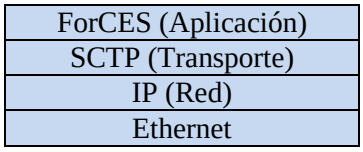


Figura 17. Pila del protocolo ForCES (*ForCES Protocol Stack*)

Para el desarrollo de este programa se tuvieron en cuenta las siguientes variables: El sistema operativo, el lenguaje de programación y los escenarios de prueba de la implementación. Los lenguajes de programación más utilizados para la elaboración del protocolo son: C++ y Java, Se escogió el lenguaje Java por facilidades de programación del autor del proyecto y por la gran variedad de APIs que ofrece este lenguaje, entre ellas el API SCTP, necesario para el establecimiento del canal de transporte que utiliza el protocolo ForCES, y por este mismo motivo fue obligatorio realizar el desarrollo del programa en el sistema operativo LINUX porque en la versión JDK7 de Java sólo se encuentra disponible esta API, para sistemas operativos LINUX o SOLARIS. Seguido a esto y según la arquitectura ForCES y el documento **draft-ietf-forces-interoperability-04** que propone la realizacion de las pruebas del protocolo por escenarios, se planteo entonces un diseño de los elementos de red CE y FE basados en consolas, que permitirá realizar el establecimiento del protocolo paso a paso a traves de comandos.

Según lo anterior y apegado a la especificacion **RFC5810** que muestra la arquitectura de un enrutador flexible reprogramable, se planteo la pila del protocolo ForCES como se observa en la anterior figura, en donde en su base se encuentra la tecnología de interconexión Ethernet, y sobre él se encuentra el sistema operativo LINUX en la capa de transporte, que permite el funcionamiento de la API SCTP el cual estableéce un punto de comunicación llamado *socket* el cual realiza el proceso de emitir o recibir información, en la siguiente capa se encuentra el protocolo ForCES.

6.2. EL CE Manager (CEM) Y EL FE Manager (FEM)

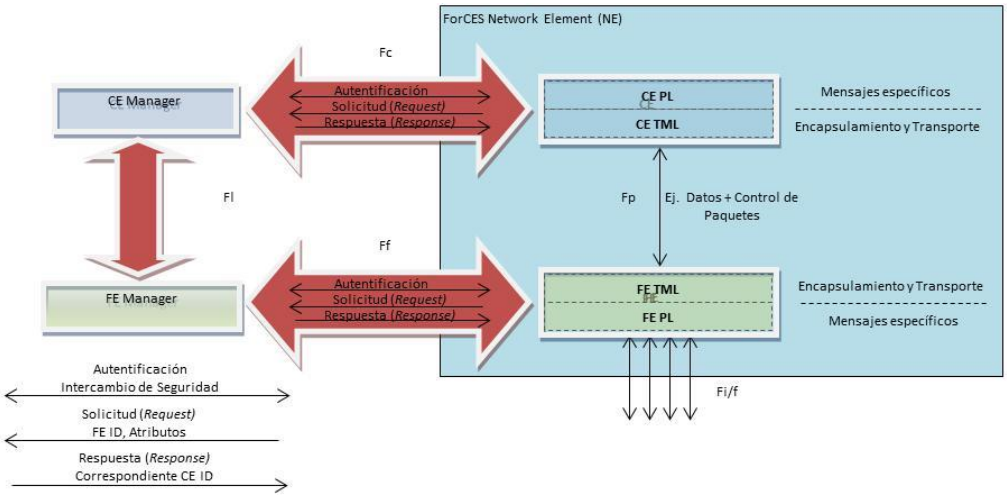


Figura 18. Interacción entre el CEM, el FEM y el NE

Además de los Elementos de Control (CE) y de Reenvío (FE), un NE tiene otras dos entidades: el Administrador de CE y el Administrador de FE. Sin embargo estos elementos tan sólo se emplean en la fase de Pre-Asociación, cuando los CEs descubren a los FEs disponibles, en este proceso se crea una interfaz que contiene la configuración y los parámetros que serán usados por las capas PL y TML dentro del CE y el FE respectivamente, antes de quedar activo y que el protocolo ForCES empiece su ejecución (o incluso puede ocurrir en tiempo de ejecución según se desarrolle la aplicación). Estos elementos de administración se encuentran fuera del alcance del protocolo ForCES (ver **RFC5810** pág. 16). Mientras que actualmente el grupo de trabajo ForCES está centrado en la fase de Post-Asociación, cuando los CEs y FEs ya se conocen entre sí. (**RFC5810** pág. 17).

6.3. FASE DE PRE-ASOCIACIÓN Y POST-ASOCIACIÓN

Estas fases se refieren a los periodos de tiempo durante el cual los CEs y FEs interactúan entre sí. En la fase de Pre-Asociación estos elementos interactúan a través del *CE Manager* y *FE Manager*, y son estas dos entidades administradoras quienes realizan las tareas de descubrimiento antes de que la fase de post-asociación donde se ejecuta como tal el protocolo ForCES empiece a funcionar, en otras palabras, cuando se habla de la fase de pre-asociación se está hablando de lo que sucede antes de que se conozcan los CEs y los FEs, y la fase de post-asociación son todos los procesos que se ejecutan después.

6.3.1. FASE DE PRE-ASOCIACIÓN

En esta fase se realiza la configuración de la interfaz CE y FE, esta configuración es estática y simplemente lee un archivo de configuración, que contiene el identificador del CE y el FE, la dirección IP y los puertos de cada uno de los canales según la prioridad. La lectura de estos archivos de configuración se lleva a cabo a través del comando Preassociation.

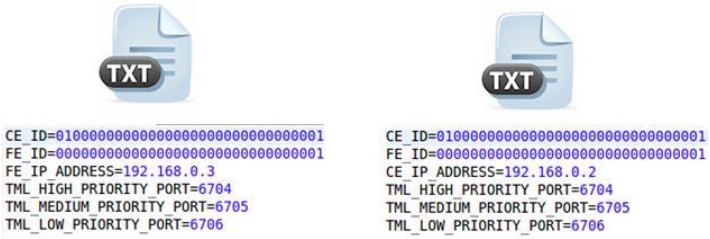


Figura 19. Archivos CE.properties y FE.properties

Según el RFC se deja a la libertad y consideración de desarrolladores o fabricantes, la forma en la que se diseñe el funcionamiento de los elementos administradores del NE que consiste en el proceso de descubrimiento e intercambio de parámetros de los CEs y los FEs.

6.3.2. FASE DE POST-ASOCIACIÓN

En esta fase los elementos CE y FE se comunican uno con el otro usando el Protocolo ForCES (PL sobre TML) y usando la configuración de la interfaz creada en Pre-Asociación.

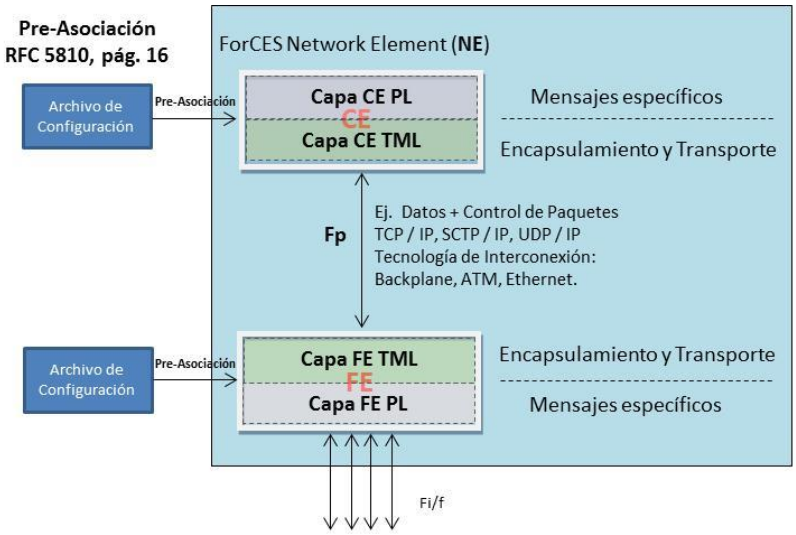


Figura 20. Proceso de Pre-Asociación y Post-Asociación

Es aquí donde ahora se realiza la asociación, este proceso se lleva a cabo a través del comando asociationsetup el cual enviará el mensaje de asociación entre los elementos CE y FE, que serán conocidos después de que la fase de Pre-Asociación se haya completado. De aquí en adelante todos los mecanismos referentes al protocolo como lo es, el establecimiento del canal de transporte del protocolo ForCES, el envío de los mensajes haciendo uso de los comandos, la construcción de las instrucciones para la configuración de las distintas topologías de los LFBs, la visualización en árbol por carpetas de la estructura XML, se encuentran dentro de la fase de post–asociación.

6.4. EL CE Y EL FE

El CE es el elemento del **plano de control**, y el FE el elemento del **plano de datos**, tanto el FE como el CE contienen internamente dos capas, la capa PL (mensajes específicos del protocolo ForCES) y la capa TML (encapsulamiento y transporte de los mensajes del protocolo ForCES) que

a su vez se intercomunican entre sí. El plano de control se encarga de tomar las decisiones, y administrar de manera lógica lo que ocurre frente a algunas tareas que requiere el plano de datos donde su función principal se basa en el reenvío de paquetes, pero para algunas operaciones lógicas es el CE en el plano de control quien determina a través de los LFBs como debe funcionar.

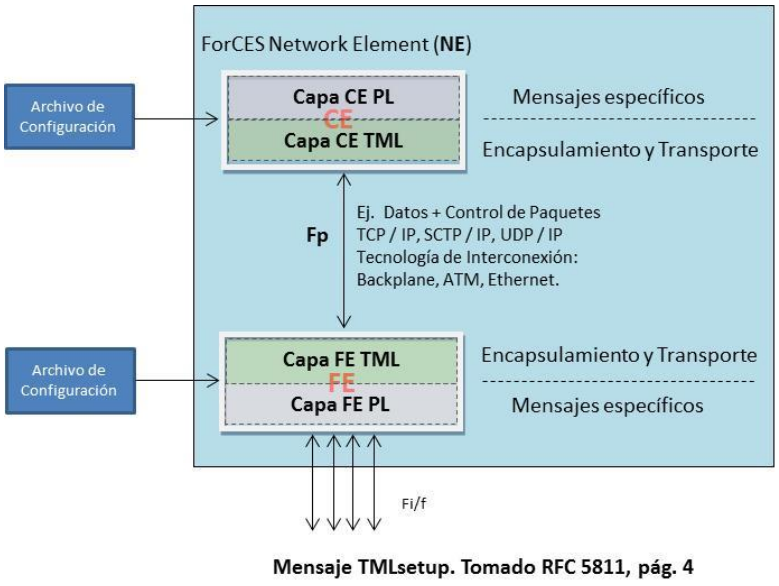


Figura 21. Interacción del Protocolo ForCES entre el CE y FE

En el paquete consola del programa en el que se implementa el Protocolo ForCES, se encuentran las clases CEConsole, Command, FEConsole, JGraphTest, LFBTree. Estas clases tienen programada la interfaz gráfica de creación de la consola, los comandos que utilizan la consola y el manejador grafico con que se representa por medio de un árbol de carpetas en el FE todas las estructuras de las librerías XML que utiliza el FE.

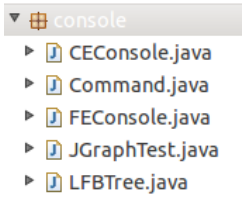


Figura 22. Paquete Consola

En las consolas CE y FE se mantiene el mismo entorno gráfico a excepción del FE que incluye dos pestañas, una para mostrar el árbol de carpetas de los LFBs y la otra para mostrar la topología que se dibuja con los datos enviados desde el CE.

6.4.1. EL CE

El CE (elemento de control) es el elemento de control basado en *software*, esta es la parte lógica del sistema y dentro de este elemento se desarrollan los algoritmos que permiten tomar las decisiones de operación del enrutador, específicamente en este proyecto estas operaciones están limitadas solamente a el establecimiento del protocolo ForCES y a la organización topológica de los LFBs en el FE que este tenga a disposición según como lo indica el **draft-ietf-forces-lfb-lib-05,** el

cual contiene dos ejemplos de organización para una funcionalidad *IPv4 Reenvío* y *ARP*. También realiza las operaciones de control de los LFBs básicos del FE como el *LFB Protocol Object (FEPO)* y el *FE Object (FEO)*.

El CE usa una interfaz de usuario en forma de ventana o también llamada consola, la cual tiene una caja de texto para visualizar el intercambio de mensajes en formato binario y los eventos que ocurran durante la operación del protocolo, además posee otra caja de texto donde se escriben las líneas de comando que se desean ejecutar.

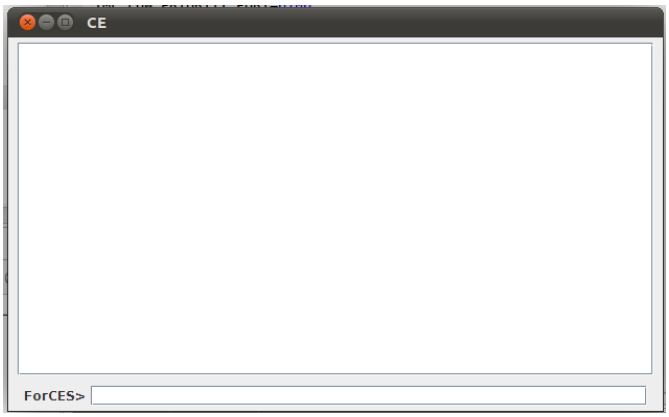


Figura 23. Consola del CE

```
public CEConsole() {
    frame = new JFrame("CE");
    JPanel center = new JPanel();
    JPanel north = new JPanel();
    logField = new JTextArea("", 21, 55);
    logField.setEditable(false);
    north.add(new JScrollPane(logField));
    commandField = new JTextField("", 48);
    commandField.addActionListener(this);
    commandField.addKeyListener(this);
    center.add(new JLabel(Command.PROMPT));
    center.add(commandField);
    frame.getContentPane().add(center, BorderLayout.CENTER);
    frame.getContentPane().add(north, BorderLayout.NORTH);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setMinimumSize(frame.getPreferredSize());
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);

    commandHistory = new ArrayList<String>();
    commandHistory.add(Command.PREASSOCIATION_SETUP);
    commandHistory.add(Command.TML_SETUP);
    commandHistory.add(Command.ASSOCIATION_SETUP);
    commandHistory.add(Command.QUERY);
    commandHistory.add(Command.CONFIG);
    commandHistory.add(Command.TEARDOWN);

    commandIndex = commandHistory.size();
}
```

Figura 24. Código de la consola CE

6.4.2. EL FE

Este elemento de red dedicado al reenvío de paquetes, se conforma de *software* y de *hardware*, en el *hardware* están las tarjetas de red y a nivel de software la comunicación mediante el protocolo con el CE. En el FE se encuentra adicionalmente las capas PL y TML, las estructuras LFBs basadas en XML las cuales son cargadas mediante librerías para darle las funcionalidades que debe tener el FE. En este caso y para este proyecto se encontrarán en el FE tres librerías XML, dos que contienen los LFBs básicos llamados *LFB Protocol Object (FEPO)* y *FE Object (FEO)* y una librería de ejemplo llamada librería LFB que contiene toda la estructura de una funcionalidad *IPv4 Reenvío.* Dentro del diseño de esta consola el manejo de los LFBs a través del árbol de carpetas se pone al alcance del CE después de ejecutar la fase de pre asociación.

La ventana o consola del FE está diseñada para que se visualicen tres elementos: Los mensajes ForCES PL en formato de líneas de 32 bits, las librerías en árbol de carpetas de los LFBs y la topología de los LFBs. En la línea de instrucción del FE opera el comando *Topology* el cual una vez ejecutado toma los datos que han sido almacenados en la estructura XML y los dibuja.

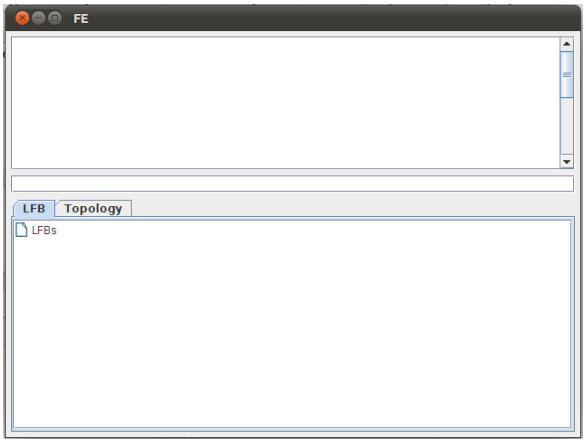


Figura 25. Consola del FE

La construcción de la consola FE se basa en el mismo código utilizado en la elaboración de la consola CE, con la diferencia de que en la del FE este llama los métodos necesarios para dibujar el árbol de carpetas y la topología.

```
public void appendMessage(String message) {
    logln("--------------------------------");
    for (int i = 0; i < message.length() / 32; i++) {
        if (message.length() >= i * 32 + 32) {
            String messageLine = message.substring(i * 32, i * 32 + 32);
            logln(messageLine);
        } else {
            String messageLine = message.substring(i * 32, message.length());
            logln(messageLine);
        }
    }
    logln("--------------------------------");
    logField.setCaretPosition(logField.getDocument().getLength());
}
```

Figura 26. Formato del mensaje en las consolas

```
private void initialize() {
    fe = new FE();
    fe.setLogListener(this);
    fe.setLFBChangeListener(this);
    frmFe = new JFrame();
    frmFe.setTitle("FE");
    frmFe.setBounds(100, 100, 640, 480);
    frmFe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frmFe.getContentPane().setLayout(
            new MigLayout("", "[grow]", "[grow][][grow]"));

    JScrollPane scrollPane = new JScrollPane();
    frmFe.getContentPane().add(scrollPane, "cell 0 0,grow");
    logField = new JTextArea(21, 34);
    logField.setEditable(false);
    scrollPane.setViewportView(logField);
    commandField = new JTextField();
    commandField.addActionListener(this);
    commandField.addKeyListener(this);
    frmFe.getContentPane().add(commandField, "cell 0 1,growx");
    commandField.setColumns(10);
    JTabbedPane tabbedPane = new JTabbedPane(JTabbedPane.TOP);
    frmFe.getContentPane().add(tabbedPane, "cell 0 2,grow");

    lfbTree = new LFBTree();

    JScrollPane scrollPane_1 = new JScrollPane();
    tabbedPane.addTab("LFB", null, scrollPane_1, null);
    scrollPane_1.setViewportView(lfbTree.getTree());

    topologyPanel = new JScrollPane();

    mxGraph graph = new mxGraph();
    Object parent = graph.getDefaultParent();

    graph.getModel().beginUpdate();
    graph.getStylesheet().getDefaultVertexStyle().put(mxConstants.STYLE_ROUNDED, true);
    graph.getStylesheet().getDefaultVertexStyle().put(mxConstants.STYLE_WHITE_SPACE, "wrap");
    try {
        Object v1 = graph.insertVertex(parent, null, "fehmngslsd:20:78", 20, 20, 70, 34);
        Object v2 = graph.insertVertex(parent, null, "node2 lfbmodelforces", 240, 150, 70, 34);
        graph.insertEdge(parent, null, "Edge", v1, v2);
    } finally {
        graph.getModel().endUpdate();
    }

    mxGraphComponent graphComponent = new mxGraphComponent(graph);
    topologyPanel.setViewportView(graphComponent);

    // ----------------------------------------

    tabbedPane.addTab("Topology", null, topologyPanel, null);

    commandHistory = new ArrayList<String>();
        commandHistory.add(Command.PREASSOCIATION_SETUP);
        commandHistory.add(Command.TML_SETUP);
        commandHistory.add(Command.ASSOCIATION_SETUP);
        commandHistory.add(Command.QUERY);
        commandHistory.add(Command.CONFIG);
        commandHistory.add(Command.TEARDOWN);
    commandIndex = commandHistory.size();
}
```

Figura 27. Figura Código De Consola FE

6.4.2.1. Los LFBs

En ForCES no es posible definir todas las operaciones del plano de datos del enrutador. En su lugar, ForCES define un lenguaje **XML** que permite construir un modelo genérico del Elemento de

Reenvío, en el que se divide el proceso de enrutamiento en varias etapas. Según este modelo, un FE está formado por *Logical Functional Blocks* (LFB) interconectados entre sí. Cada LFB modela una parte del proceso de reenvío de un datagrama IP. Por tanto el denominado **protocolo ForCES** será un protocolo de control que permitirá a los CE modificar los atributos de cada LFB de un FE.

También será posible cambiar la **topología de los LFBs**, esto es; como se interconectan entre sí, por tanto el protocolo ForCES puede emplearse tanto para acceder al estado de los LFBs, así como para configurar su comportamiento, de forma que cada LFB tendrá atributos de estado y atributos de configuración. El modelo LFB hace parte del FE y por tanto del Protocolo ForCES, y se consulta y modifica a través de los mensajes *Query* y *Config*, respectivamente. La función de estos comandos y la construcción de su instrucción se verán más adelante.

Los FEs son controlados por los CEs a través de los LFBs, que a su vez son administrados por una estructura definida por la especificación **RFC5812**, basada en lenguaje de marcas o etiquetas denominada XML (*Extensible Markup Language*), el cual es un formato basado en texto y específicamente diseñado para almacenar y transmitir datos.

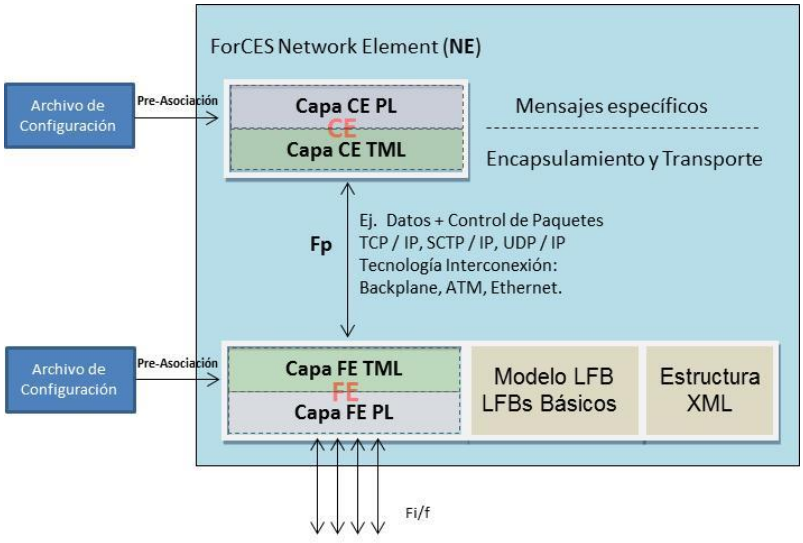


Figura 28. Modelo LFB y Estructura XML en el FE

Los datos a consultar o trasmitir en el protocolo son los atributos que forman las funcionalidades de cada uno de los LFBs y que se organizan estructuralmente haciendo uso del lenguaje XML.

Un LFB se crea a partir de un *ClassID*, un *InstanceID* y una variedad de componentes, y dentro de estos pueden haber más componentes, y dentro de cada componente una colección de atributos, también tiene un puerto de entrada y uno de salida, en algunos casos se usa un grupo de puertos cuando un LFB se conecta con varios LFBs y se le conoce como bifurcación de entrada o salida.

Para hacer una conexión entre dos LFBs, la entrada y salida debe tener el mismo puerto que se identifica con un número de 32 bits, para direccionar un componente se utiliza el comando Path que fue diseñado siguiendo algunas indicaciones que entrega la especificación **RFC5812**, pagina 32 donde se muestra como se direcciona un componente dentro de otro, la grafica que se muestra a continuación es una recopilación de varias gráficas que se encuentran en el **RFC5812**, paginas 30,

31 y 32, que ilustra cómo es un LFB y las partes que lo conforman. Esta representación de un LFB es una manera de explicar gráficamente un LFB, pero en realidad su descripción está hecha esencialmente de parámetros escritos en lenguaje XML y que son algo complejo de comprender.

Los LFBs pueden tener varias entradas y salidas, que además pueden agruparse para formar grupos de entrada/salida. En general las salidas de un LFB se agrupan cuando todas ellas transmiten el mismo tipo de paquetes, pero deben ser procesados por LFBs diferentes (Por ejemplo: un datagrama IPv4 puede ser enviado a otro FE o reenviado por otra interfaz del FE que lo recibió), mientras que las excepciones en el procesamiento de los paquetes tienen salidas propias. Por tanto, los LFBs con varias salidas deben elegir una de ellas basándose en el contenido del paquete o en su meta-información asociada.

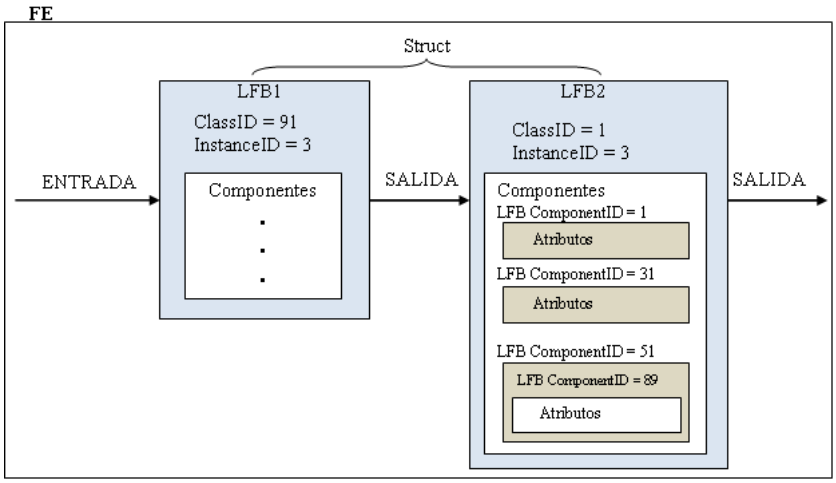


Figura 29. Arquitectura LFB

Las entradas/salidas de un LFB siempre están unidas a otros LFBs, de forma que un LFB recibe un paquete por alguna de sus entradas, lo procesa y lo reenvía por alguna de sus salidas. Las únicas excepciones a este comportamiento de flujo son: los *LFB Dropper* que no tienen salidas sino que sirven para descartar paquetes, y los LFB de Port que representan los puertos físicos del FE, y por tanto cuando reciben un paquete, lo envían por la línea, mientras que sus paquetes de salida, acaban de ser recibidos por el FE. De hecho, incluso la comunicación dentro del enrutador (CE-FE o FE-FE) se modela con puertos LFBs, lo que permite independizar ForCES y la comunicación inter FE de la tecnología de interconexión empleada.

A partir de la representación gráfica se puede entender y diseñar como conectar diferentes tipos de LFBs con el objetivo de agruparlos para que realicen una función específica.

Los LFBs que se encuentran a disposición mantienen una estructura XML y cada vez que se desee una función adicional en el FE se debe adicionar una librería XML con toda la descripción de estas funcionalidades, para el caso del FE de este proyecto tiene dos librerías LFB base y una de ejemplo. Estos LFBs los desconoce el CE, cualquier cambio o adición de librerías LFBs sólo las conoce el FE, y para que el CE las conozca usa el mensaje Query del protocolo ForCES mediante el comando Query para conocer el estado de las variables de las librerías LFBs.

Los valores que se encuentran dentro de cada uno de los atributos de los componentes LFBs se encuentran por defecto en cero (0), es decir el contenido de estos atributos depende del diseño funcional que se quiera implementar.

6.4.2.1.1. Struct

Para llegar al *struct* la ruta de direccionamiento apunta dentro de la librería *FEObject* donde está el componente *LFBTopology* en el que se encuentra el *Array* que contiene una colección de elementos llamada *struct,* que describen los atributos que le permite al LFB conectarse con otro LFB. La especificación **RFC5812** solo presenta una colección *struct* que hace referencia a un solo LFB base, y a partir de este se deben crear los LFBs que sean necesarios según el diseño de conexión que se busca para una funcionalidad especifica. Como se vio antes, un LFB es un bloque lógico y cada uno es diferente de otro según su función, pero su composición externa es la misma, por ello se usa la misma colección de atributos del *struct* para conectar los diferentes LFBs.

En java este componente *struct* se copia como un objeto creando uno nuevo cada vez según el número de LFBs que se necesiten, como un *struct* es una conexión entre dos LFBs los atributos de esta colección tiene un puerto de conexión común y los otros atributos describen de donde viene y hacia dónde va la conexión.

6.4.2.1.2. Clonación del *struct*

Es importante en este punto hablar sobre la referencia de los objetos java, cuando se crea un objeto se crean las dos cosas, la referencia y el objeto, cuando a un objeto se le asigna otro objeto, el lo que hace es crear una segunda referencia apuntando al mismo objeto, es decir cuando se dice, al objeto1 asígnele el objeto2 no se está creando el objeto2, no se está creando un nuevo objeto, sino que realmente lo que se está haciendo es pasándole la referencia. Entonces el objeto2 queda apuntándole al mismo objeto. Cuando se hace un cambio en el objeto1, este cambio se refleja inmediatamente en el objeto2, por que los dos están referenciando el mismo objeto. Por esto, este proceso de creación de objetos no sirve para la creación de objetos de las colecciones dentro de los *arrays* de los LFBs del proyecto, ya que al crearlos y modificar uno de ellos se modificarían todos. Lo que se necesita es un objeto totalmente nuevo a partir de un **objeto base** que en este caso es una colección de elementos LFBs llamada *struct* dentro de los *arrays.* A partir del objeto base *struct* lo que se hace es hacer un **clon**, al clonar este procedimiento permite hacer una copia idéntica del objeto base lo que significa que al crear este objeto se crea una nueva referencia de ese objeto de manera que cada *struct* opera independientemente.

Para el proceso anterior el código del programa usa un método llamado *ObjectClone* al cual se le pasa un objeto y este devuelve el clon de ese objeto, lo que hace es que el método lo escribe en un *Stream* llamado *ObjectOutputStream,* como si este se fuese a enviar, a ser escrito en un archivo o en algún recurso externo, en lugar de esto lo escribe sobre otro Stream de entrada llamado *ObjectInputStream,* Es decir simula como si enviara el objeto y lo devolviera y luego se lee, dicho objeto leído es ya una copia totalmente diferente al objeto que se paso para clonar.

6.4.2.2. Estructura XML

Hay una librería de Apache que se llama *XMLBeans* (http://xmlbeans.apache.org/) y brinda la funcionalidad de tomar un documento XML, hacerle el cambio (*parse*) y mapearlo en objetos JAVA, es muy fácil de usar, disminuye el tiempo de desarrollo, es una librería que ya ha sido probada muchas veces, y es poco susceptible a errores. Lo que se tiene que hacer es cargar el archivo XML como muestra el ejemplo:

Primero hay que crear un archivo XSD a partir de un documento XML, para ello se puede usar el ambiente de desarrollo de Java Eclipse, que permite crear un nuevo documento XML de dos tipos: XML *File* y XML Schema.

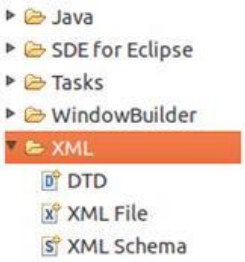


Figura 30. Creación de un archivo XSD

El archivo *XMLFile*, crea un XML y el archivo *XMLSchema*, crea un XSD a partir del **RFC5812**, Capitulo 4 pág., 41. Donde se hace la descripción de todo el esquema, y el código del esquema también se encuentra en este mismo **RFC5812** en la pág., 88-98, llamado *XML Schema for LFB Class Library Documents*.

Se selecciona *XMLSchema*, una vez seleccionada la ruta de creación del archivo se le asocia el nombre LFBClassLibrarySchema.xsd, y luego de este procedimiento se observa un documento en blanco que contiene apenas algunas líneas de código que indican el encabezado de la estructura XML que se va a construir, la especificación del protocolo ForCES entrega la descripción de muchos de sus LFBs en formato XML y XSD. El esquema XSD y la estructura XML fueron copiados a este documento en blanco desde el **RFC5812** y fue guardado en un archivo con extensión XSD, el cual es un metadato que define que estructura puede tener un XML.

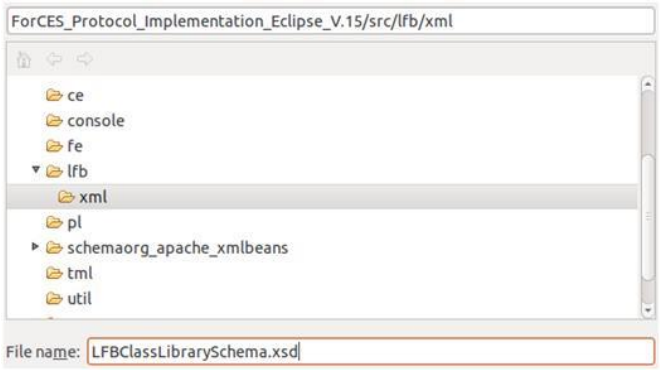


Figura 31. Nombre de documento XML

Después de obtener y guardar el archivo XSD, se necesita generar un .Jar a partir del esquema XSD, para poder manejar desde java la colección de atributos de XML como objetos java. Para crear esta clase manejadora hay dos opciones, una es crear un .Jar sin tener acceso al código fuente y la otra creando las clases con el código fuente.

Ahora se hace uso del XMLBeans quien se encarga de tomar el archivo XSD y generar las clases de java indispensables para manejar la estructura XML. Lo siguiente que se debe hacer es tener instalado el XMLBeans como se muestra a continuación:

Para instalar se debe dar *click* en los siguientes enlaces:

1. Instalar *XMLBeans*.
2. *Download the XMLBeans release*.
3. *Apache mirrors*
4. *http://apache.opensourceresources.org//xmlbeans*
5. *xmlbeans-current.tgz*

Se inicia la descarga de una carpeta comprimida con el programa *XMLbeans*. Terminada la descarga, la instalación del *XMLbeans* es solo descomprimir una carpeta, la cual tiene varias herramientas, una de ellas es el *scomp* que se ejecuta a través de unas opciones, y con estas opciones se generan las clases de Java que se necesitan.

```
scomp - out elemploschema.jar ejemploschema.xsd
```

La instrucción del ejemplo anterior genera un .Jar con los *class* de toda la estructura XML, es decir, no genera un código fuente solo los binarios, pero se le puede dar la opción de que genere un código fuente como se hizo para la realización de este proyecto y es por esto que en el explorador de proyectos del IDE Eclipse, se pueden ver todos los archivos *Class* y todos los archivos auxiliares que en el programa usan la extensión XSB.

Al descomprimir la carpeta XMLBeans se encuentra otra carpeta llamada BIN que contiene el comando *scomp* y desde la consola de comandos de Linux se busca la carpeta donde se ubica este comando, y seguimos el ejemplo de la instrucción anterior para generar el archivo .Jar:

```
pedro@pedro-Inspiron-1525:~/ForCES_Tools/xmlbeans-2.5.0/bin$ sh scomp -out xmlSchema.jar
/home/pedro/workspace/ForCES_Tools/ForCES_Protocol_Implementation_Eclipse_V.15/src/lfb/x
ml/ LFBClassLibrarySchema.xsd
```

Después de ejecutado este comando desde la consola de Linux se ingresa a la carpeta donde está ubicado el XMLBeans y a la carpeta BIN donde se observa que se ha creado un nuevo archivo llamado xmlSchema.jar, que es una librería con todas las clases generadas a partir del esquema XML. Para este proyecto no se utilizo esta opción del .Jar, sino que se genero el código fuente para poderlo manejar, entonces volvemos a la consola de Linux y en vez de utilizar la instrucción *out* se utiliza la instrucción *src* y se le indica la ruta donde se va a generar el archivo con el código fuente haciendo uso de la opción *–d*:

```
pedro@pedro-Inspiron-1525:~/ForCES_Tools/xmlbeans-2.5.0/bin$ sh scomp -src xmlSchema.jar -
d/home/pedro/workspace/ForCES_Tools/ForCES_Protocol_Implementation_Eclipse_V.15/src/lfb/x
ml/ LFBClassLibrarySchema.xsd
```

Después de ejecutado este comando se generan dos carpetas llamadas x0 donde está el código fuente generado en archivos .java y .class y también la carpeta *schemaorg_apache_xmlbeans*, donde están los archivos auxiliares XSB, los nombres de las carpetas que se generaron fueron tomados del espacio de nombre (*namespace*) del esquema XML.

Entonces se copian estas dos carpetas en la carpeta *src* del proyecto y dentro del código del proyecto se programa una clase *LFBHandler* que permite usar las clases contenidas en la carpeta x0. El cual carga los documentos XML con los LFBs y hace la interacción con las clases de Apache que se generaron, entonces se utiliza el método inicializar LFB (*InitLFBModel*), el cual empieza a iterar sobre las colecciones FEObjectLFB.xml, FEProtocolLFB.xml,

BaseLFBLybrary_Complete.xml tomadas del **RFC5812**; que está dentro de la carpeta lfb.xml del proyecto.

```
File xmlFile = new File("c:\employees.xml");

// Bind the instance to the generated XMLBeans types.
EmployeesDocument empDoc =
        EmployeesDocument.Factory.parse(xmlFile);

// Get and print pieces of the XML instance.
Employees emps = empDoc.getEmployees();
Employee[] empArray = emps.getEmployeeArray();
for (int i = 0; i < empArray.length; i++)
{
        System.out.println(empArray[i]);
}
```

Figura 32. Ejemplo XMLBeans, tomado de http://xmlbeans.apache.org/

La clase *LFBHandler* tiene un método llamado *initLFBModel* con un *for* que itera creando un objeto de tipo archivo como sugiere la referencia del XMLBeans ya antes visto y que sugiere utilizar la instrucción (File xmlFile = new File ("c:\employees.xml")) y todo el ejemplo de la figura anterior haciendo uso del *parse (xmlFile)*.

```
public void initLFBModel() throws Exception {
    lfbClasses = new HashMap<String, LFBLibraryDocument>();
    lfbInstances = new HashMap<String, LFBInstance>();
    File currentDirectory = new File (".");

    for (int i = 0; i < LFB_NAMES.length; i++) {
        StringBuilder lfbFilePath = new StringBuilder();
        lfbFilePath.append(currentDirectory.getCanonicalPath());
        lfbFilePath.append(PATH).append(LFB_NAMES[i]);
        File lfbFile = new File(lfbFilePath.toString());
        LFBLibraryDocument lfbClass = LFBLibraryDocument.Factory.parse(lfbFile);
        LFBClassDef[] lfbClassDefs = lfbClass.getLFBLibrary().getLFBClassDefs().getLFBClassDefArray();
        for (int j = 0; j < lfbClassDefs.length; j++) {
            long lfbClassId = lfbClassDefs[j].getLFBClassID();
            lfbClasses.put(lfbClassId+","+INSTANCE_ID, lfbClass);
            LFBInstance lfbInstance = initInstance(lfbClassDefs[j]);
            lfbInstances.put(lfbClassId+","+INSTANCE_ID, lfbInstance);
        }
    }
    topology = new TopologyHandler(this);

}
```

Figura 33. Método initLFBModel de la clase LFBHandler

Por ejemplo el tipo complejo *ArrayType* que se encuentra dentro de toda la estructura XML como un segmento que define una estructura de datos con una secuencia de atributos y con el XMLBeans genera los objetos java en un archivo llamado *ArrayType*.java. Otro ejemplo es el atributo *length* lo convierte en un objeto java así: java.*math.BigInteger.getLength* (); el XMLBeans lo que hace es tomar la secuencia de atributos del esquema y la mapea a clases java (*Class*).

El BaseLFBLybrary_Complete.xml es un draf candidato a RFC llamado **draft-ietf-forces-lfb-lib-05** que contiene la estructura XML para las funcionalidades básicas de un enrutador (IPv4-Reenvío) en el capitulo *XML for Base Type Library* ubicado en las páginas 17-37. Este documento XML está regido por el esquema XSD.

6.4.2.3. Árbol LFB

El navegar a través del documento XML que se está utilizando en este proyecto es complejo para el entendimiento humano así que se pensó por medio de toda la estructura XML una representación grafica en forma de carpetas de toda la colección de elementos y componentes de las librerías XML de los LFBs. Para ello se utilizó un manejador grafico de java que hace un barrido a través de todas las estructuras escalonadas del XML y las va dibujando en forma de árbol de carpetas, así es más fácil visualizar los cambios en los valores de los elementos y componentes de los LFBs.

Por ejemplo cuando se envían una cadena de datos desde el plano de control en el CE, estos llegan al FE y se almacenan en la estructura XML de los LFBs modificando sus valores para poder visualizarlos si estos llegaron correctamente no es necesario entrar en el código del programa sino que se despliega el árbol de carpetas y se visualiza directamente en tiempo de ejecución del programa.

En este proceso es indispensable tener todo el XML convertido en objetos para que así el manejador del árbol LFB pueda recorrer y dibujar más rápido todos los componentes. En el paquete *Console* del programa hay una clase llamada *LFBTree* que se encarga de dibujar en la consola LFB lo que el método *LoadLFBTree* trae del esquema XML.

```
public LFBTree() {
    super();
    DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode("LFBs");
    tree = new JTree(rootNode);
    tree.getSelectionModel().setSelectionMode (TreeSelectionModel.SINGLE_TREE_SELECTION);
}

public void loadLFBTree(Map<String, LFBInstance> lfbInstances) {
    DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode("LFBs");
    DefaultMutableTreeNode lfbBasicNode = new DefaultMutableTreeNode("Object y Protocol LFB");
    DefaultMutableTreeNode lfbLibraryNode = new DefaultMutableTreeNode("Librería LFB");
    rootNode.add(lfbBasicNode);
    rootNode.add(lfbLibraryNode);
    for (LFBInstance lfbInstance : lfbInstances.values()) {
        DefaultMutableTreeNode lfbNode = new DefaultMutableTreeNode(lfbInstance.getName());
        lfbNode.add(new DefaultMutableTreeNode("Class ID : "+lfbInstance.getClassId()));
        lfbNode.add(new DefaultMutableTreeNode("Instance ID : "+lfbInstance.getInstanceId()));
        lfbNode.add(new DefaultMutableTreeNode("Name : "+lfbInstance.getName()));
        lfbNode.add(new DefaultMutableTreeNode("Synopsis : "+lfbInstance.getSynopsis()));
        lfbNode.add(new DefaultMutableTreeNode("Version : "+lfbInstance.getVersion()));
        if (lfbInstance.getComponents() != null) {
            DefaultMutableTreeNode componentsNode = new DefaultMutableTreeNode("Components");
            for (ComponentInstance component : lfbInstance.getComponents()) {
                componentsNode.add(loadComponentTree(component));
            }
            lfbNode.add(componentsNode);
        }
        if (lfbInstance.getCapabilities() != null) {
            DefaultMutableTreeNode capabilitiesNode = new DefaultMutableTreeNode("Capabilities");
            for (ComponentInstance capability : lfbInstance.getCapabilities()) {
                capabilitiesNode.add(loadComponentTree(capability));
            }
            lfbNode.add(capabilitiesNode);
        }
        if (lfbInstance.getClassId().equals("1") || lfbInstance.getClassId().equals("2")) {
            lfbBasicNode.add(lfbNode);
        } else {
            lfbLibraryNode.add(lfbNode);
        }
    }
    DefaultTreeModel treeModel =(DefaultTreeModel)tree.getModel();
    treeModel.setRoot(rootNode);
    treeModel.reload();
}
```

Figura 34. Método LoadLFBTree

6.4.2.3.1. Instrucciones basadas en el árbol LFB

Los comandos y las instrucciones utilizadas desde la consola del CE se crearon y fueron inspiradas a partir de los casos de uso del apéndice D del **RFC5810**, pág. 107, que da una idea de un escenario

de ejecución y de cómo direccionar los LFBs del protocolo que se quieren leer y modificar, el cual menciona cómo se direccionaría a partir de los identificadores y de los componentes de los LFBs para asignar y leer los valores de estos componentes. Hay un ejemplo de un comando de configuración en la página 109, donde se envían varios valores de un arreglo a través de varias instrucciones, esto origino que en el proyecto se modificara el comando *Config* para que lea un script.txt que contiene varias líneas de comando para no estar enviando una cada vez, sino todo el paquete, esta es una idea que se acerca a algo más legible para el ser humano y que lo da el **RFC5810**, y direccionando todo con los IDs en diferentes niveles, hay un ejemplo de direccionamiento de niveles similar al que se plantea en este proyecto y que se encuentra en este RFC en la pagina 112 que diferencia los niveles a través de puntos (.). En este proyecto los diferentes niveles se direccionan por medio de una ruta separada por comas (,). A partir de esto fue que se diseño la línea de comandos para el *Config* y el *Query* y de esta manera construir una instrucción a un nivel más detallado de direccionamiento, en donde colocará el valor que se quiere enviar en el mensaje. Entonces cualquier elemento o componente de los LFBs se direccionan a través del ID de la clase y el ID de la instancia y una ruta dentro de los componentes.

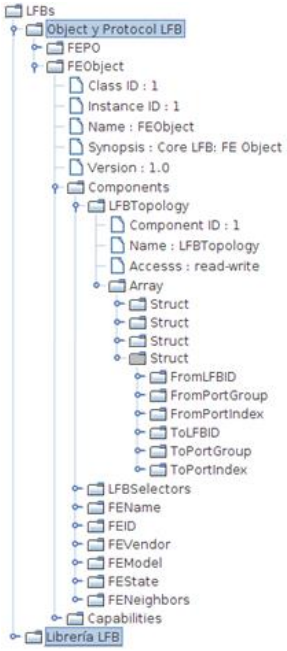


Figura 35. Árbol de carpetas LFB

6.4.2.3.2. Topología

El objetivo de hacer la topología es tener acceso a la estructura XML, modificar sus valores y recombinar sus conexiones para crear nuevas funciones. La topología es una aplicación a nivel lógico más concreta de lo que se puede diseñar con un LFB es decir dentro del modelo ForCES se define un esquema XML el cual es un modelo de datos con el que se supone se puede modelar cualquier cosa. En la página 32 de la especificación **RFC5812** se encuentra la manera como deben ser usados y conectados los LFBs a través del componente *LFBTopology* y de lo cual define que debe ser por medio de él, que se debe hacer toda la representación gráfica de un diseño funcional LFB. Esta representación gráfica se hace a partir de las rutas de dirección dentro de un FE, con los nodos que representan las instancias LFB.

Entonces la topología de ejemplo encontrada en el **draft-ietf-forces-lfb-lib-05** muestra como el modelo es adaptable a cualquier cosa, a dibujar un diagrama a definir las características de un puerto, a características de una tabla de enrutamiento es decir como es un modelo tan versátil entonces el modelo da para definir casi cualquier funcionalidad a nivel lógico, así como quienes definieron este *draft* se basaron en los LFBs y la definición del modelo LFB del **RFC5812** para definir toda una arquitectura de un enrutador básico también se podría tomar este modelo LFB ya para aplicarlo y definir la estructura de cualquier otro componente de red.

Para configurar la topología se toma la instrucción y se agrupan varias de ellas dentro de un *script* para luego ser enviados a través de la línea de comandos del CE y configurar todos los valores del árbol XML. Se modifico el comando *Config* para que pueda cargar todas las sentencias de comandos que se han construido según el diseño de alguna topología en especial para el caso de este proyecto se diseño un *script* especial con todas las líneas de comando necesarias para configurar la topología de una funcionalidad IPv4 *Reenvío* que se encuentra en el **draft-ietf-forces-lfb-lib-05** pagina 12. Y entonces se coloca en la línea de comandos la instrucción Config –file "ruta del script.txt" y se envía desde el CE al FE. Al verificar el árbol de carpetas LFB en el *LFBObject* en el componente *topology* se encuentran todas las carpetas *struct* creadas con la información enviada desde el CE y luego de verificar que los valores se encuentran en las rutas indicadas se dibuja el diseño en la ventana *topology* a través del comando Topology.

```
public void handleTopology() {
    mxGraph mxGraph = new mxGraph();
    mxGraph.getStylesheet().getDefaultVertexStyle().put(mxConstants.STYLE_ROUNDED, true);
    mxGraph.getStylesheet().getDefaultVertexStyle().put(mxConstants.STYLE_WHITE_SPACE, "wrap");
    fe.updateTopology(mxGraph);
    mxOrganicLayout organicLayout = new mxOrganicLayout(mxGraph);
    organicLayout.execute(mxGraph.getDefaultParent());
    mxGraphComponent graphComponent = new mxGraphComponent(mxGraph);
    System.out.println("--- handleTopology ---");
    topologyPanel.setViewportView(graphComponent);

}
```

Figura 36. Método handleTopology

Los nodos que se dibujan sobre la plantilla de la pestaña topology en el FE es una representación del LFB que se tiene conectado, esta representación gráfica y la distribución de todos los LFBs en la plantilla obedecen a un algoritmo denominado *mxOrganicLayout* [http://www.jgraph.com/] que se encarga de realizar las conexiones y las distribuciones topográficas de los LFBs dependiendo de la cantidad de nodos que haya basado en métricas de distancia y proximidad entre los nodos y también teniendo en cuenta los datos correspondientes a el puerto, y el nodo LFB de donde viene la conexión y hacia dónde sale una nueva conexión. Sobre estos valores el algoritmo calcula la distancia que debe haber entre cada nodo, usando la librería jgraphx.jar. La clase manejadora que construye la topología LFB se llama *TopologyHandler* y fue creada basada en la clase *mxGraph* contenida en la librería jgraphx.jar, esta clase grafícadora es usada en el método *UpdateTopology*.

```
public mxGraph updateTopology(LFBInstance lfbObjectLFBInstance, mxGraph mxGraph) {
    ComponentInstance lfbTopology = LFBHandler.getComponentByID(1, lfbObjectLFBInstance.getComponents());
    ArrayInstance arrayLFBLinks = (ArrayInstance)lfbTopology.getValue();
    List<Object> lfbLinks = arrayLFBLinks.getItems().subList(1, arrayLFBLinks.getItems().size());
    for (Object lfbLinkObject : lfbLinks) {
        StructInstance lfbLink = (StructInstance)lfbLinkObject;

        ComponentInstance fromLFBID = LFBHandler.getComponentByID(1, lfbLink.getComponents());
        StructInstance fromLFBSelector = (StructInstance)fromLFBID.getValue();
        String fromLFBClassID = Binary.parseLong(getLFBClassID(fromLFBSelector))+"";
        String fromLFBInstanceID = Binary.parseLong(getLFBInstanceID(fromLFBSelector))+"";
        AtomicInstance fromPortIndexAtomic = (AtomicInstance)LFBHandler.getComponentByID(3, lfbLink.getComponents()).getValue();
        String fromPortIndex = Binary.parseLong((String)fromPortIndexAtomic.getValue())+"";

        ComponentInstance toLFBID = LFBHandler.getComponentByID(4, lfbLink.getComponents());
        StructInstance toLFBSelector = (StructInstance)toLFBID.getValue();
        String toLFBClassID = Binary.parseLong(getLFBClassID(toLFBSelector))+"";
        String toLFBInstanceID = Binary.parseLong(getLFBInstanceID(toLFBSelector))+"";
        AtomicInstance toPortIndexAtomic = (AtomicInstance)LFBHandler.getComponentByID(6, lfbLink.getComponents()).getValue();
        String toPortIndex = Binary.parseLong((String)toPortIndexAtomic.getValue())+"";

        if (fromLFBClassID == null || fromLFBInstanceID == null || fromPortIndex == null) {
            System.out.println("Inconsistencia en origen");
            continue;
        }
        if (toLFBClassID == null || toLFBInstanceID == null || toPortIndex == null) {
            System.out.println("Inconsistencia en detino");
            continue;
        }
        if (!fromPortIndex.equals(toPortIndex)) {
            System.out.println("Inconsistencia en Puertos");
            continue;
        }
        String fromLFBName = getLFBName(fromLFBClassID+","+fromLFBInstanceID);
        String toLFBName = getLFBName(toLFBClassID+","+toLFBInstanceID);
        addTopologyLink(fromLFBName+":"+fromLFBClassID+","+fromLFBInstanceID, fromPortIndex, toLFBName+":"+toLFBClassID+","+toLFBInstanceID
    }
    return mxGraph;
}
```

Figura 37. Método UpdateTopology

6.5. CAPAS PL Y TML

El protocolo ForCES debe ser independiente de la tecnología de interconexión subyacente, ya que los elementos de Control (CE) y de Reenvío (FE) pueden estar interconectados por diferentes tecnologías como por ejemplo una tarjeta de conexión de módulos (*backplane*), una matriz de conmutación ATM, o incluso *Ethernet* si residen en chasis diferentes. A raíz de este requisito, el protocolo ForCES se subdivide en dos capas: *Protocol Layer (PL)* y *Transport Mapping Layer (TML)*.

La capa PL define los mensajes, la máquina de estados, y en general todos los aspectos del protocolo ForCES que sean independientes de la tecnología de comunicación subyacente.

Por el contrario, **la capa TML** se ocupa del transporte de los mensajes de la capa superior a través de la red de interconexión interna, por tanto para cada tecnología se definirá una capa TML adecuada. Por ejemplo, TCP/IP, SCTP/IP, UDP/IP los cuales se proponen como protocolos de TML para transportar los **mensajes ForCES PL** sobre **redes IP**.

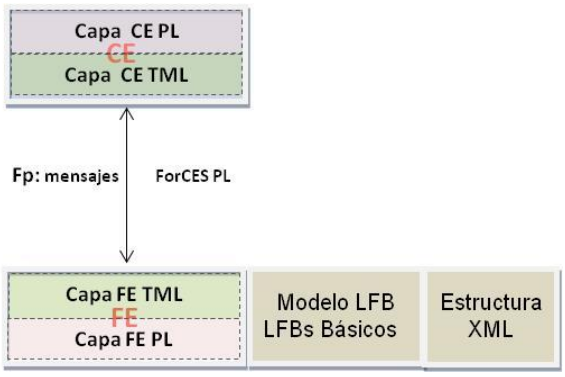


Figura 38. Capas PL y TML

En la figura se puede apreciar los elementos CE y FE subdivididos cada uno por las capa PL y TML respectivamente, el elemento FE tiene incorporado los LFBs basados en estructura XML.

6.5.1. CAPA PL

La Encapsulación de los mensajes ForCES se lleva a cabo en la capa PL, luego son pasados a la capa TML para que sean transportados.

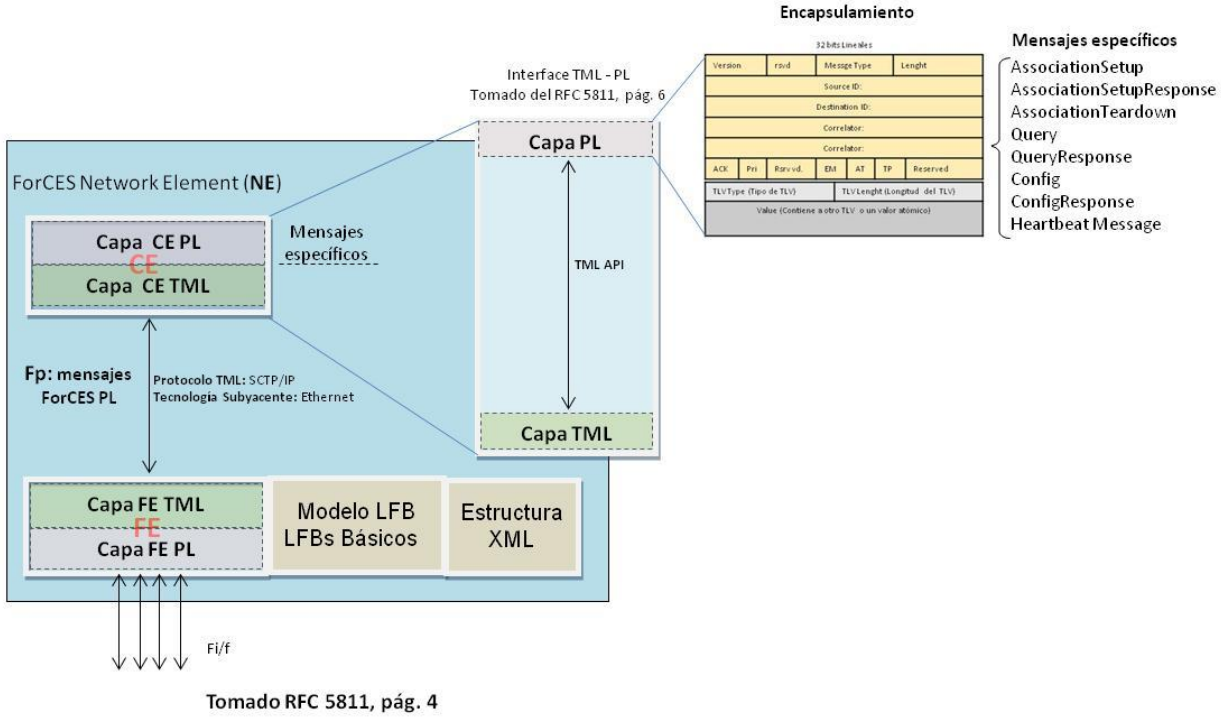


Figura 39. Capa PL

En esta capa se lleva el proceso de encapsulación de los mensajes ForCES que se basan en un encabezado (*header*) y en un cuerpo del mensaje (*body*).

6.5.1.1. Construcción de los Mensajes ForCES PL

Los **mensajes ForCES PL** están compuestos por un Header (Cabecera) y un Body (Cuerpo del Mensaje).

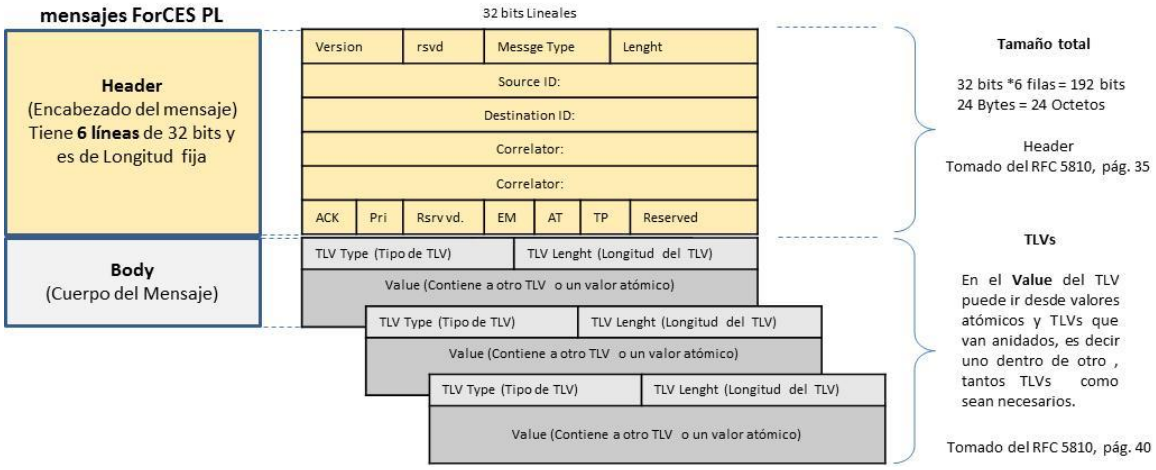


Figura 40. Encapsulamiento de un mensaje ForCES

6.5.1.1.1. Header (Cabecera)

Es el encabezado del mensaje (RFC 5810, pág. 35), hace parte de la composición de la PDU del protocolo (unidades de datos de protocolo, *protocol data unit*), la cual se utiliza para el intercambio entre unidades parejas de los mensajes del protocolo. Este sirve para controlar el comportamiento completo del protocolo en sus funciones de establecimiento y ruptura de la conexión, control de flujo, control de errores, etc.

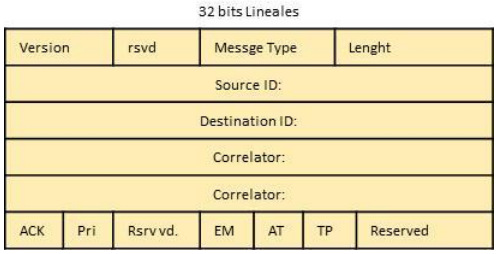


Figura 41. Encabezado del mensaje (*Header*)

El encabezado del mensaje contiene seis líneas de 32 bits, el tamaño de este paquete es de longitud fija a diferencia del *body* (cuerpo del mensaje) que se debe recalcular su longitud dependiendo de la cantidad de datos que se desean enviar. Cada uno de los bits de esta cabecera se modifica según la especificación **RFC5810**, pág. 35.La longitud (*Length*) del encabezado esta dado en *D-Words* es decir 4 bytes, como cada línea de 32 bits es igual a 4 *D-Words*, esto significa que la longitud indica el número de líneas de la cabecera del mensaje.

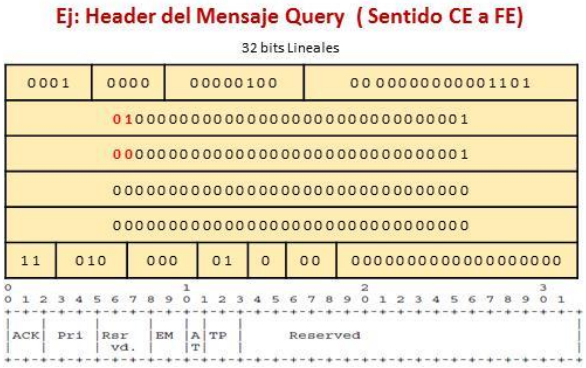


Figura 42. Ejemplo encabezado del mensaje Query

Un ejemplo de cómo se formatean los bits se puede ver en las figuras, para este caso se tomo el *header* del mensaje *Query*.

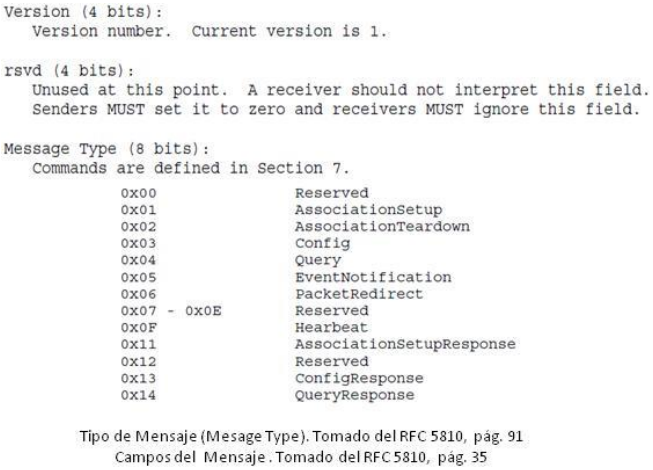


Figura 43. Versión, rsvd y tipo de mensaje

6.5.1.1.1.1. Mecanismos del protocolo

Se exponen varias semánticas a través de la cabecera de los mensajes PL del protocolo, en la que se incluye las capacidades de transacción, atomicidad de las transacciones y dos fases que implica el envío por particiones o por paralelización con alta disponibilidad y tolerancia a fallos, así como túneles para los comandos. Las banderas EM (modo de ejecución), AT (transacción atómica), y TP (fase de transacción) están relacionadas con estos mecanismos.

Figura 44. Banderas del encabezado

Estas tres banderas están directamente relacionadas con el campo *correlator* ubicado en la cuarta y quinta línea del *header*.

6.5.1.1.1.2. Correlator

Es un campo que se usa para correlacionar las secuencias de mensajes, para el caso de transacciones se puede enviar múltiples mensajes y cuando el mensaje es muy grande se puede partir. Los mensajes de este proyecto no hacen parte de una transacción, es un mensaje atómico.

Una transacción se usa al igual que una base de datos, se generan comandos como en el caso de los mensajes *Query* y *Config*, hasta que no se envíe el último mensaje de la transacción él no aplica los cambios en el LFB, cuando todos los mensajes son atómicos, apenas lleguen a su destino él aplica los cambios en el LFB, por ello se setea o se configura este campo en cero (0) según el **RFC5810**, pág. 37.

En las implementaciones que se hicieron en las universidades Zhejiang Gongshang y Patras, esta fue una de las precondiciones que se pusieron para realizar las pruebas. "**Para cada comando se usa un único mensaje**".

6.5.1.1.1.3. Configuración de las banderas

ACK (1 1): Es utilizada por el CE, al enviar un mensaje de configuración (*Config Message*) o un mensaje de *Heartbeat*, para indicar que el receptor del mensaje requiere o no una respuesta del remitente. Es importante resaltar que para otros mensajes que no sean el mensaje de configuración o el mensaje de *Heartbeat* este indicador deberá ser ignorado. Los valores de los indicadores se definen como sigue:

NOACK (0b00): Para indicar que el receptor del mensaje no deberá devolver un mensaje de respuesta al remitente del mensaje.

SuccessACK (0b01): Para indicar que el receptor del mensaje debe devolver un mensaje de respuesta sólo cuando el mensaje ha sido procesado con éxito por el receptor.

FailureACK (0b10): Para indicar que el receptor del mensaje debe devolver un mensaje de respuesta sólo cuando hay incumplimiento por parte del receptor en el procesamiento (de ejecución) del mensaje. En otras palabras, si el mensaje se pudo procesar con éxito, el emisor no espera ninguna respuesta por parte del receptor.

AlwaysACK (0b11): Para indicar que el receptor del mensaje debe enviar siempre un mensaje de respuesta.

Pri (0 1 0): Prioridad son 3 bits, y establecen la prioridad del canal según el tipo de mensaje. Ver RFC5811, pág. 10

Reservado (0 0 0): 3 bits.

EM (0 1): Modo de ejecución son 2 bits.

0 0: Reservado.
0 1: Ejecute todo o nada.

1 0: Ejecute hasta que falle.
1 1: Continúe a pesar de que haya una falla.
Ver **RFC5810**, pág. 10, 20

AT (0): *Atomic Transaction* 1 bit.
0: El mensaje es Atómico.
1: El mensaje hace parte de una transacción.
Ver **RFC5810**, pág. 10, 19, 21-25

TP (0 0): Fase de la Transacción 2 bits.
Depende de lo anterior, si se ha dicho antes que no es una transacción, entonces se procede al inicio del mensaje. **RFC5810**, pág. 10, 19, 21-25

Reservado: Todo en ceros.

6.5.1.1.1.4. Identificador de Destino y Origen

En la segunda y tercera línea del encabezado van los identificadores de origen y destino ocupan cada uno la línea de 32 bits, en el **RFC5810**, pág. 36. Se encuentra el formato en el que se deben expresar los bits, el campo TS tiene dos bits a la izquierda que se refiere al tipo de elemento donde corresponde el TS (00) para definir un elemento FE y el TS (01) para definir un elemento CE. Los siguientes treinta bits a la derecha define el número del CE o del FE respectivamente.

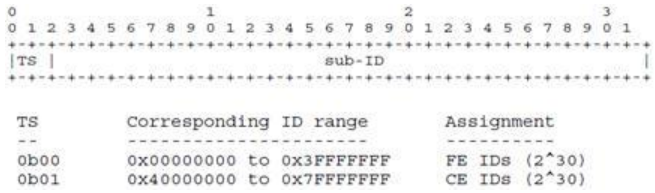


Figura 45. Identificador de Destino y Origen del encabezado

La segunda línea de la cabecera configura a un elemento CE o FE como origen y la tercera línea lo configura como destino. Es importante conocer el sentido en el que se va a enviar el mensaje, cual es el origen y cuál es el destino.

6.5.1.1.2. Body (Cuerpo del Mensaje)

Cada cuerpo del mensaje está formado por uno o más TLVs de alto nivel. Un TLV de alto nivel puede contener uno o más sub-TLV, estos sub-TLV se describen en este proyecto como un OPER-TLV, porque describen una operación a realizar. Para el caso de un **comando *config*** el OPER-TLV= SET, y para el **comando *Query*** el OPER-TLV= GET, como se podrá apreciar más adelante cuando se construyen los mensajes *Config* y *Query*.

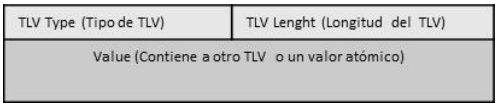


Figura 46. Cuerpo del mensaje (Body)

6.5.1.1.2.1. Estructura de un TLV

TLVs (Valores de Longitud y Tipo), son bloques de información que transporta valores incluyendo el tipo de información que contiene y la longitud de la información, la cual para este proyecto es de longitud variable. Algunas de sus ventajas son: la codificación de una secuencia de datos de fácil interpretación, también se pueden añadir *tags* a los mensajes sin hacer que el mensaje sea incompatible, permite anidar estructuras TLVs una dentro de otra de forma que el dato de un TLV puede ser a su vez otro TLV. Además es fácil saber si el mensaje ha llegado completamente sin necesidad de carga extra.

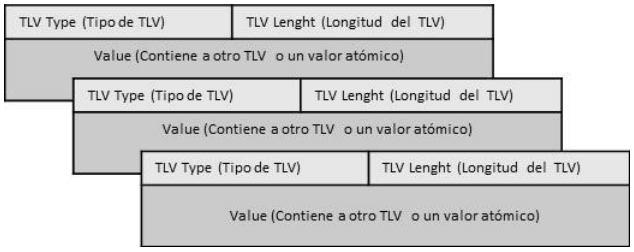


Figura 47. TLVs anidados

Para la construcción de los mensajes PL ForCES se anidan entre sí distintos tipos de TLVs, entre ellos el OPER-TLV, FULLDATA-TLV, PATH-DATA-TLV.

Hay diferentes tipos de TLVs, este es el LFBselect que es un LFB Básico para la funcionalidad del protocolo ForCES y tiene a su vez un LFBCLASSID y un LFBInstance para direccionar un LFB en particular, y dentro de este tiene a otro TLV del tipo OPERATION, que dice si es un GET o un SET, además todos los IDs de asociación y todos los tipos de operación de los mensajes ForCES que de manera general están compuestos de esta forma. (**RFC5810**, pág. 44).

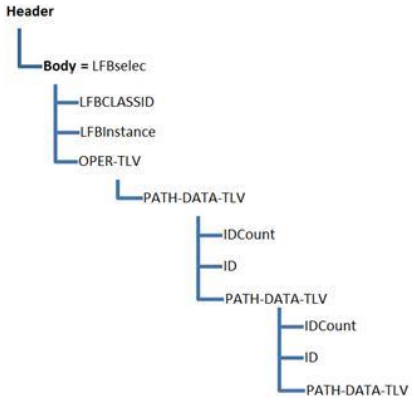


Figura 48. Direccionamiento de entidades LFBs

Hay una tabla que indica cuales son cada uno de los LFBs que usa cada mensaje (**RFC5810**, pág. 45) y también se encuentra la definición de cómo se configura un OPER-TLV (**RFC5810**, pág. 46), en el cual la primera línea del OPER-TLV es el PATH, que es donde él está apuntando. Este PATH está definido por unos FLAGS, IDCount y un ID, y luego viene el DATA o la información del OPER-TLV.

```
OPER-TLV := 1*PATH-DATA-TLV
PATH-DATA-TLV := PATH  [DATA]
PATH := flags IDcount IDs [SELECTOR]
SELECTOR :=  KEYINFO-TLV
DATA := FULLDATA-TLV / SPARSEDATA-TLV / RESULT-TLV /
        1*PATH-DATA-TLV
```

Figura 49. Ejemplo de instrucciones de direccionamiento

Para armar los mensajes ForCES referentes al *body* a nivel del código del programa y que está construido a partir de TLVs, estos se implementan con un formato en líneas de 32 bits para ser visualizados en la interfaz del usuario, pero al interior del programa estos deben ser enviados a través del canal utilizando dos clases hechas en java llamadas serialización (*serialize*) y deserialización (*deserialize).* Estas dos operaciones toman los datos del mismo (*this*), haciendo uso de la instancia *message* recorriendo todos los TLVs y convirtiéndolos en bits para armar el mensaje que se va a enviar, devolviendo un arreglo de bytes. El *deserialize* hace todo lo contrario, este recibe un contenido de un arreglo de bytes y lo convierte en bits devolviendo una instancia del mensaje (*message*). Cuando está llegando el mensaje entonces el toma los bytes que están llegando por el canal, y empieza a procesarlos generando una instancia del tipo *message* con todos los atributos que él tenga. Cuando se va a enviar el mensaje se crea el mensaje en el CE a través de una instancia *message* dependiendo del tipo de mensaje que se ha puesto dentro de la instrucción.

```
public byte[] serialize() {
    StringBuilder messageBits = new StringBuilder();
    messageBits.append(header.parseHeader());
    if (body != null) {
        messageBits.append(body.parseTLV());
    }
    Binary.printBits(messageBits.toString());
    BigInteger bi = new BigInteger(messageBits.toString(), 2);
    byte[] messageBytes = bi.toByteArray();
    int bitLength = bi.bitLength();
    if (bitLength%Binary.OCTET == 0) {
        byte[] messageBytes2 = new byte[messageBytes.length - 1];
        for (int i = 0; i < messageBytes2.length; i++) {
            messageBytes2[i] = messageBytes[i + 1];
        }
        return messageBytes2;
    } else {
        return messageBytes;
    }
}
```

Figura 50. Código del método serialize

Justo antes de enviar el contenido por el canal convertido en bits se hace uso de un método utilitario llamado *logBits* el coge y empieza a imprimir en las consolas CE y FE los bits formateados y organizados en líneas de 32 bits, porque lo que en realidad viaja a través del canal es una cadena de bits, entonces el toma esta cadena la parte y la visualiza para que sea entendible a nivel humano.

La clase *Bynari* también es una clase utilitaria que permite convertir líneas de bits en bytes y formatear los datos para que la clase *logBits* haga su trabajo. Su principal importancia es rellenar los espacios vacios de 32 bits de ceros a la izquierda cuando se hace una conversión.

```
public static Message deserialize(byte[] messageContent) {
    Message message = new Message();
    BigInteger bi = new BigInteger(messageContent);
    String messageBits = "000"+bi.toString(2);
    String headerBits = messageBits.substring(0, 6*Binary.DWORD);
    Header header = Header.parseBits(headerBits);
    message.setHeader(header);
    if (messageBits.length() > 6*Binary.DWORD) {
        String tlvBits = messageBits.substring(6*Binary.DWORD, messageBits.length());
        TLV tlv = TLV.parseBits(tlvBits);
        message.setBody(tlv);
    }
    return message;
}
```

Figura 51. Código del método deserialize

6.5.1.2. FE Object y FE Protocol LFB

Todos los mensajes PL operan en la construcción de los LFB, ya que estos proporcionan una mayor flexibilidad para futuras mejoras. Esto significa que el mantenimiento y la configuración de los FEs, el NE, y el protocolo ForCES deben ser expresados en términos de esta arquitectura LFB. Por esta razón los LFBs básicos que se están mencionando, se crean para satisfacer esta necesidad.

Para lograr esto, los LFBs básicos son los siguientes:

- **FE Protocol Object LFB (FEPO)**: Se utiliza para controlar el protocolo ForCES.
- **FE Object LFB (FEO)**: Se utiliza para controlar los componentes relativos del propio FE.

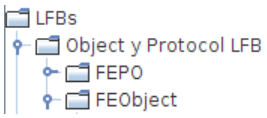


Figura 52. FE Object y FE Protocol LFB

6.5.1.2.1. FE Protocol LFB

Es una entidad lógica que se encuentra en cada uno de los FEs y es utilizado para controlar el protocolo ForCES. Al *FE Protocol LFB Class ID* se le asigna el valor de (0x2). Al *FE Protocolo LFB Instance ID* se le asigna el valor de 0x1. Debe haber una y sólo una instancia del *LFB FE Protocol* del FE que se esté modificando. Los valores de los componentes de los *LFB FE Protocol* se han predefinido por defecto en el **RFC5810**.

A menos que se realicen cambios de manera explícita a estos valores por medio del mensaje de configuración desde el CE (comando *Config*), estos valores por defecto deben ser utilizados para el correcto funcionamiento del protocolo.

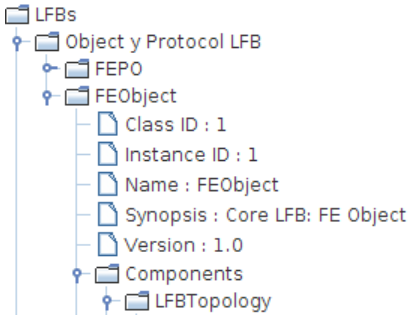


Figura 53. LFB Class ID e Instance ID

6.5.1.2.2. FE Object LFB

Aunque estos LFBs tienen la misma forma e interfaz como los otros LFBs. Se deben conocer los elementos LFB, *Class ID* e *Instance ID*. Estas son definidas estáticamente (sin permitir una instanciación dinámica), y su estado no puede ser modificado por el protocolo: Por ejemplo, si se va a desactivar un LFB, debería generar un error.

Además, estos LFBs deben estar creados (Fase Pre-Asociación) antes de que se envíe y se reciba el primer mensaje ForCES. Cada uno de los componentes de los LFBs deben tener valores predefinidos por defecto, los cuales son utilizados para encontrar y modificar el componente *LFBTopolog*, que contiene los campos que hacen referencia a entrada y salida de los LFBs que se desean conectar.

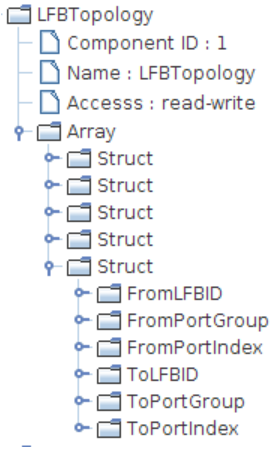


Figura 54. LFB Object y LFBTopology

6.5.1.3. Encapsulamiento y estructura general de los Mensajes ForCES PL

Los mensajes que se construyen en la capa PL tales como: Asociation Setup, Asociation Setup Response, Config Setup, Config Setup Response, Query Setup, Query Setup Response tienen un sentido de dirección que es importante tener en cuenta antes de ser encapsulados, los mensajes Teardown y *Heartbeat* pueden ser inicializados desde cualquier origen, y algunos de ellos no son mensajes tales como el comando de pre asociación que no tiene orden de inicio y el TML Setup que si debe guardar como origen al CE y como destino FE.

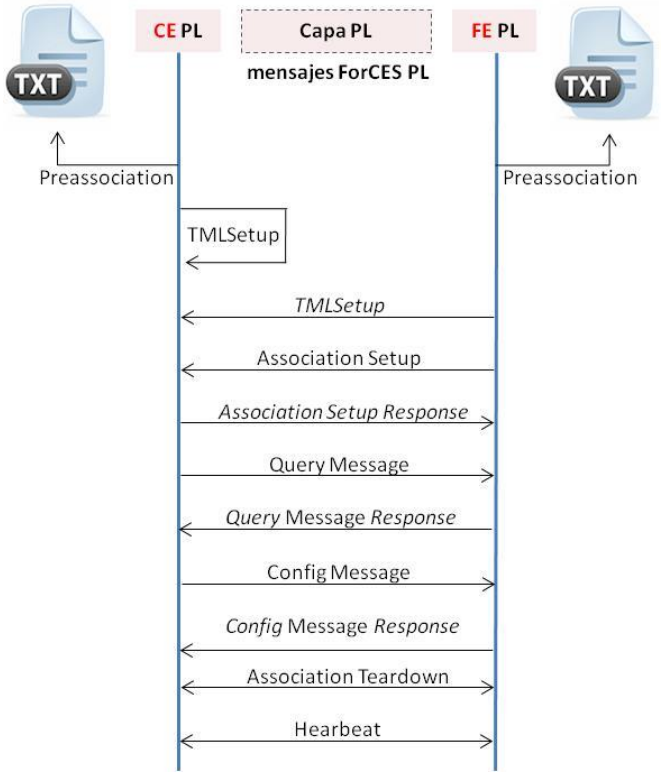


Figura 55. Establecimiento de Mensajes ForCES

6.5.1.3.1. Pre-Asociación

Antes de la transición a la fase de asociación, se simula que se ha establecido contacto con un componente de la CEM, para el caso de los CEs, y luego con los FEM para los FEs, este establecimiento no exige un orden de inicio.

La inicialización y la autenticación de la interfaz de ForCES se ha completado, leyendo un archivo txt que contiene la identificación, dirección IP y los puertos del canal SCTP. Tanto el FE como el CE tendrían ahora la información necesaria para conectarse el uno al otro mediante los mensajes del protocolo ForCES que inician con el mensaje de asociación.

En resumen, al finalizar esta etapa, ambas partes tienen ahora todos los parámetros necesarios para el establecimiento del protocolo. El punto de referencia Fl como lo indica la arquitectura puede seguir funcionando durante la fase de asociación y se puede utilizar para obligar a una disociación de un FE o un CE. Las interacciones específicas de la CEM y la FEM de la que forman parte de la fase de pre-asociación están fuera de alcance de este proyecto y del protocolo ForCES, por esta razón, estos detalles no se discuten más allá de la especificación **RFC5810** que se está implementando.

Debido a que la forma este se deja a consideración del fabricante en este proyecto por razones de comodidad para la programación se manejó un archivo de propiedades (*properties*) creado en java y nombrado CE.*properties* y FE.*properties*.

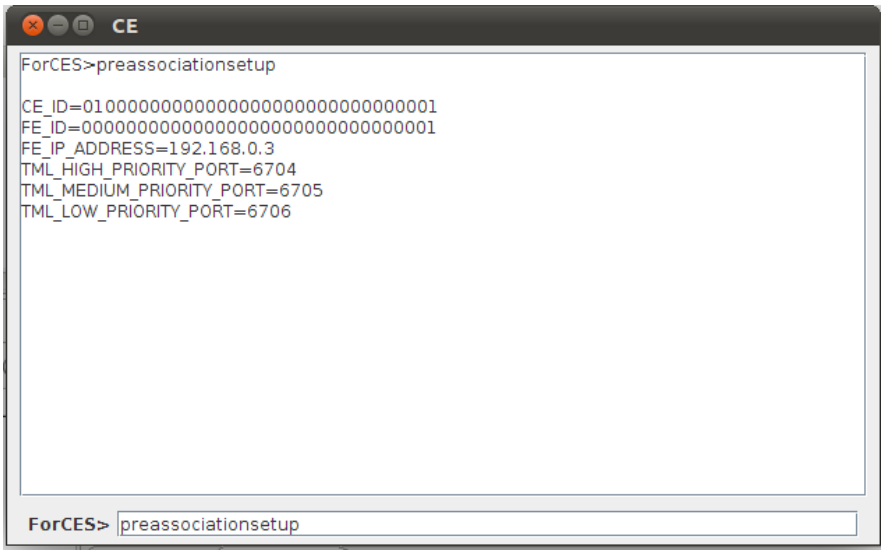


Figura 56. CE Pre asociación Setup

Al ejecutar el comando *preassociationsetup* el programa lee el archivo *properties*, y lo visualiza en cada una de las consolas CE y FE para indicar que la información de estos dos elementos ha sido cargada correctamente.

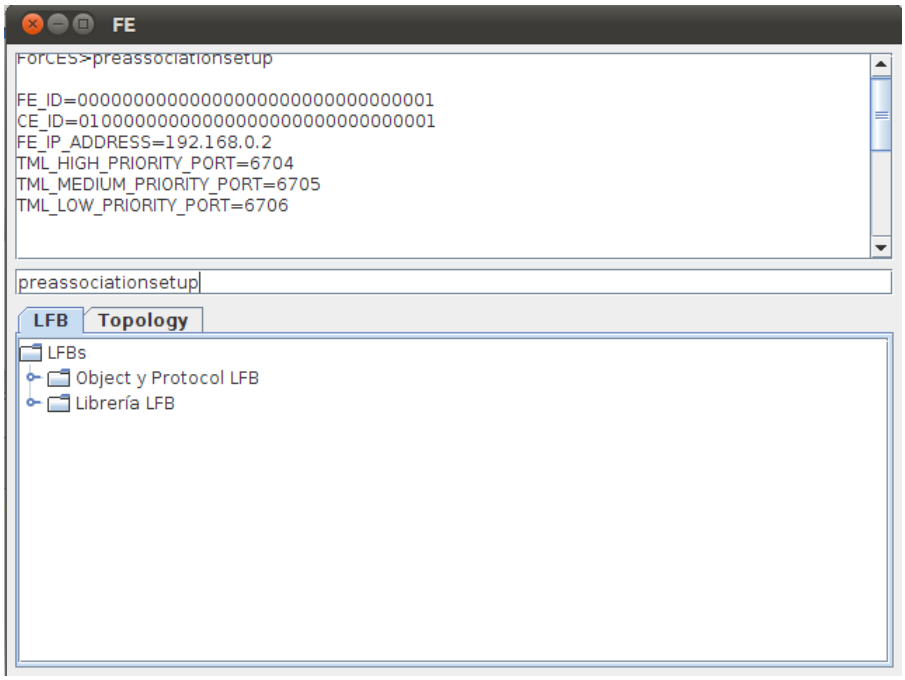


Figura 57. FE Pre asociación Setup

En el código referente a la fase de pre asociación lee los archivos estáticos y prepara la información perteneciente a cada elemento y listos para iniciar la conexión y la fase de asociación.

```
public void setupPreAssociation() {
    InputStream inputStream =
        getClass().getResourceAsStream("FE.properties");
    Properties properties = new Properties();
    try {
        properties.load(inputStream);
        feID = properties.getProperty("FE_ID");
        ceID = properties.getProperty("CE_ID");
        ceIP = properties.getProperty("CE_IP_ADDRESS");
        highPriorityPort = properties.getProperty("TML_HIGH_PRIORITY_PORT");
        mediumPriorityPort = properties.getProperty("TML_MEDIUM_PRIORITY_PORT");
        lowPriorityPort = properties.getProperty("TML_LOW_PRIORITY_PORT");

        // Pre-association Setup logging messages
        logMessage("");
        logMessage("FE_ID=" + feID);
        logMessage("CE_ID=" + ceID);
        logMessage("FE_IP_ADDRESS=" + ceIP);
        logMessage("TML_HIGH_PRIORITY_PORT=" + highPriorityPort);
        logMessage("TML_MEDIUM_PRIORITY_PORT=" + mediumPriorityPort);
        logMessage("TML_LOW_PRIORITY_PORT=" + lowPriorityPort);
        logMessage("");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figura 58. Código de fase de pre asociación

6.5.1.3.2. TMLSetup

El TMLSetup es un comando que permite el establecimiento del canal de conexión sobre la tecnología *Ethernet* con el protocolo de transporte SCTP sobre el que viajaran los mensajes PL ForCES. Por tratarse de una conexión del tipo maestro – esclavo se debe mantener el orden CE a FE para enviar el comando, debido a que el CE actúa como maestro y el FE como esclavo.

Una vez se escribe el comando en la línea de instrucciones del CE este abre los tres canales basados en los puertos que exige el RFC5811 y se queda escuchando hasta que se realiza la conexión debido a la petición del otro comando ejecutado desde el FE.

```
public void setupTML() {
    // TML priority channels connection
    logMessage("");
    logMessage("--- TML priority channels connection ---");
    highPriorityChannel = new FEChannel(this, ceIP, highPriorityPort);
    logMessage("High Priority Channel");
    mediumPriorityChannel = new FEChannel(this, ceIP, mediumPriorityPort);
    logMessage("Medium Priority Channel");
    lowPriorityChannel = new FEChannel(this, ceIP, lowPriorityPort);
    logMessage("Low Priority Channel");
    logMessage("");
}
```

Figura 59. Código del TMLSetup

6.5.1.3.3. Mensaje de Asociación

Este mensaje es enviado en sentido FE (origen) a CE (destino) para establecer una asociación ForCES entre ellos, y solo consta del encabezado del mensaje (*Header*), no contiene cuerpo del mensaje (*body*). La Asociación entre el CE y el FE condiciona el funcionamiento y envío de los mensajes Config, Query, *Heartbeat* y el teardown según el **RFC5810**, pág. 84.

En el *header* de este mensaje el campo *MessageType* se configura con *AssociationSetup*, por tratarse de un mensaje de asociación, el **RFC5810** hace referencia a que la bandera ACK del mensaje de asociación debe ser ignorada, ya que un mensaje de asociación siempre espera obtener una respuesta del receptor del mensaje, en este caso el CE. Sin embargo este mensaje se configura en (1 1) estos dos (2) bits indican que el receptor del mensaje tiene que enviar un mensaje de respuesta.

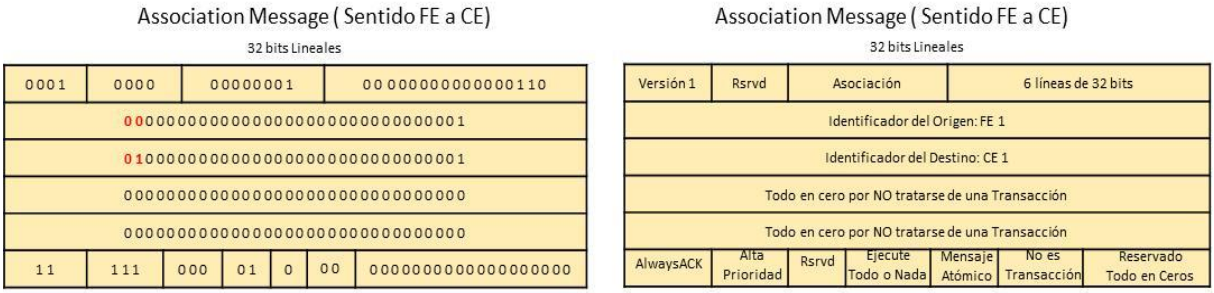


Figura 60. Mensaje de asociación

El cuerpo de mensaje es opcional, puede constar de ningún TLV es decir sin *body*, uno o dos TLVs LFBselect, tal como se ve en el siguiente diagrama.

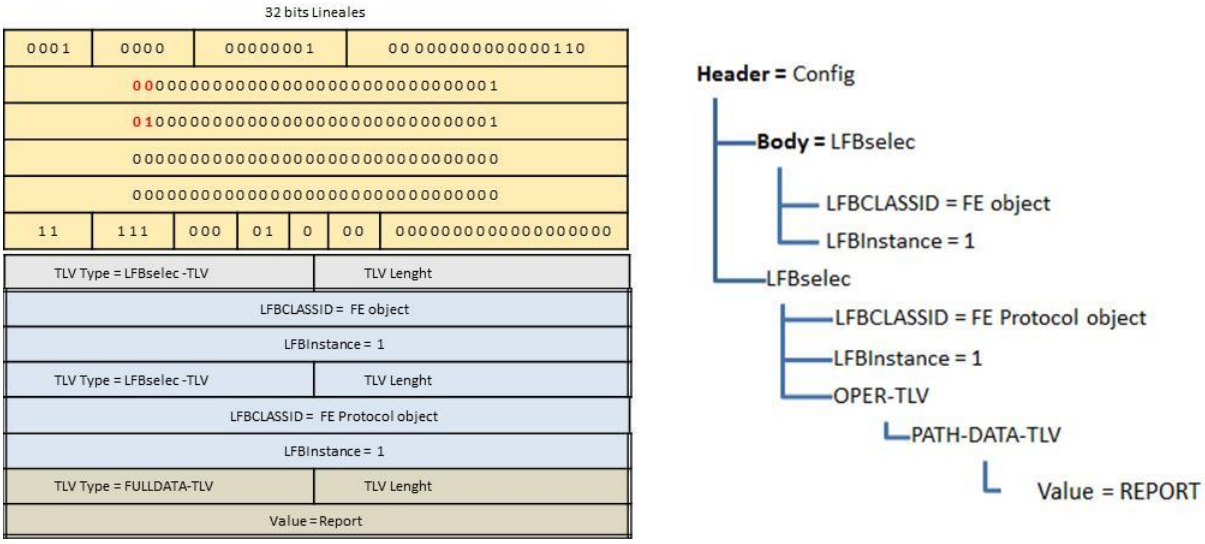


Figura 61. Mensaje de asociación opcional

El mensaje de configuración para la Asociación sólo opera en el *FE object* y *FE Protocolo LFB*, por lo tanto, el ID de clase LFB en el *TLV LFBselect* sólo apunta a estos dos tipos de LFBs.

El mensaje de respuesta *Association Setup Response Message* es semejante al mensaje de asociación, en lo único que cambia es en el campo *message type* de la cabecera *message type* = 00010001, en la cabecera del mensaje, que indica que el tipo de mensaje que se recibe es de respuesta.

```
public void associationSetup() {
    Message associationSetupMessage = new Message();
    Header header = new Header();
    header.setVersion(Header.VERSION);
    header.setRsvd(Header.RSVD);
    header.setMessageType(MessageType.ASSOCIATION_SETUP);
    header.setLength(Header.LENGTH);
    header.setSourceId(feID);
    header.setDestId(ceID);
    header.setCorrelator(Header.IGNORED_CORRELATOR);
    HeaderFlags flags = new HeaderFlags();
    flags.setAck(HeaderFlags.NO_ACK);
    flags.setPriority(HeaderFlags.ASSOCIATION_SETUP_PRIORITY);
    flags.setRsvd1(HeaderFlags.RSVD_FLAGS1);
    flags.setExecutionMode(HeaderFlags.EXECUTE_ALL_OR_NONE);
    flags.setAtomicTransaction(HeaderFlags.STANDALONE_MESSAGE);
    flags.setTransactionPhase(HeaderFlags.SOT);
    flags.setRsvd2(HeaderFlags.RSVD_FLAGS2);
    header.setFlags(flags);
    associationSetupMessage.setHeader(header);
    byte[] messageContent = associationSetupMessage.serialize();
    logMessage("--- Send: Association Setup Message---");
    Binary.logBits(messageContent, log);
    highPriorityChannel.send(messageContent);
}
```

Figura 62. Código mensaje de asociación

6.5.1.3.4. Mensaje Config

El mensaje config se envía con el comando config, en la siguiente figura se observa el formato general de un mensaje config y el mensaje config de respuesta, estos dos mensajes se componen de un encabezado (*header*) y un cuerpo (*body*).

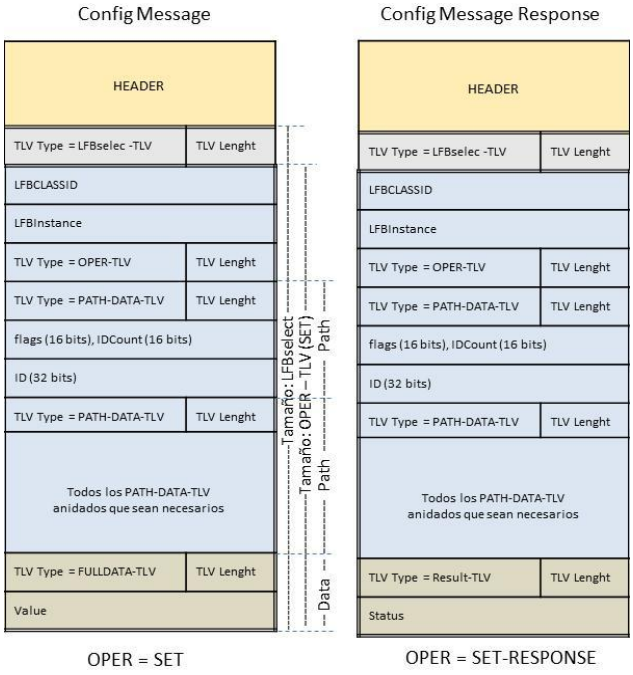


Figura 63. **Mensaje *Config Massage* y *Config Message Response***

El formateo de los datos que se ve en la siguiente figura se encapsula luego de enviar desde la ventana de comandos del CE la siguiente instrucción **config –classid 1 –instanceid 1 path 1,5,6 –component 1 –value 2001**

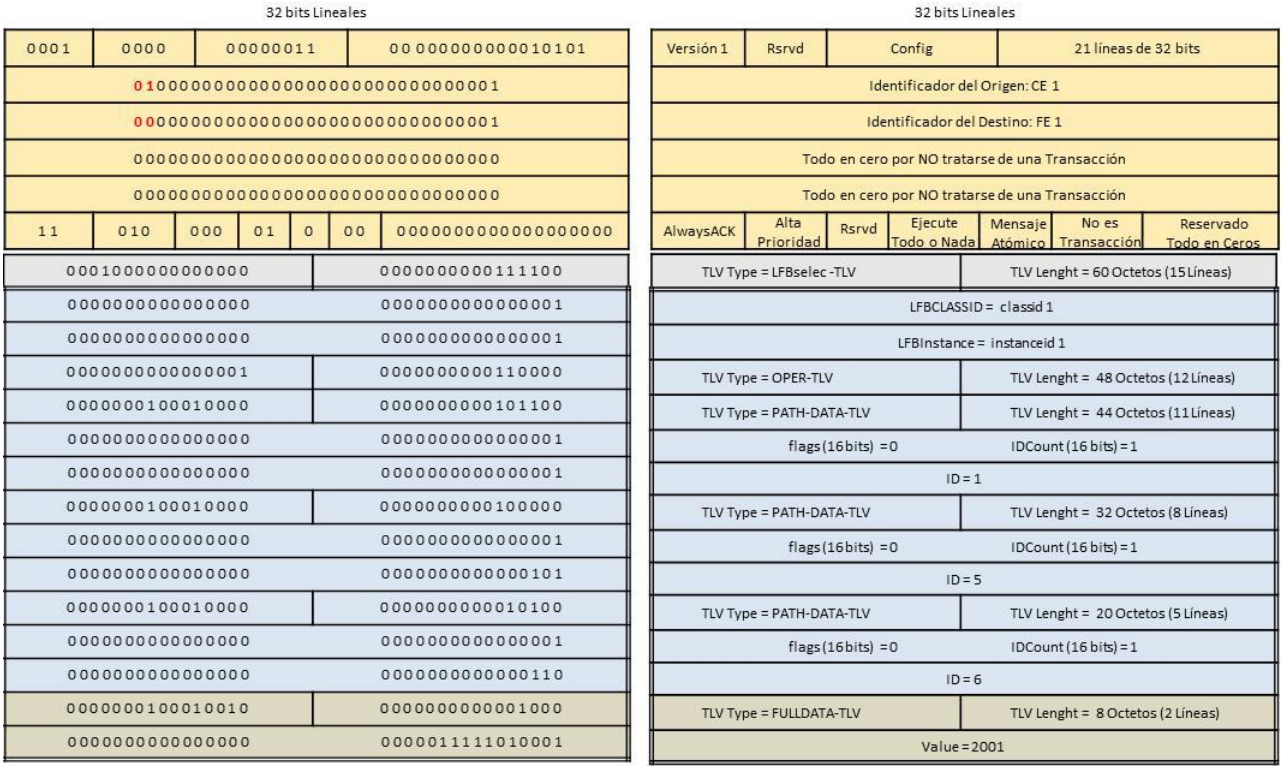


Figura 64. Formateo de datos del mensaje *Config Message*

En la figura anterior se puede apreciar el correspondiente formateo de datos según como la especificación lo indica, mostrando la correspondencia entre cada línea de bits y el campo al que esta asociado según el protocolo tanto en el encabezado como el cuerpo del mensaje.

En el cuerpo del mensaje, el cual es un TLV, se observan varios tipos de TLVs anidados uno dentro de otro, llevando la información de los componentes del LFB que se desea modificar y el valor.

```
public void handleConfig(Message configMessage) {
    TLV tlvResult = new TLV(TlvType.RESULT_TLV);

    TLV lfbSelectTlv = configMessage.getBody();
    long lfbClassId = Binary.parseLong((String)lfbSelectTlv.getValue(0));
    long lfbInstanceId = Binary.parseLong((String)lfbSelectTlv.getValue(1));
    TLV operTlv = (TLV)lfbSelectTlv.getValue(2);
    TLV pathDataTlv = (TLV)operTlv.getValue(0);
    String path = getPath(pathDataTlv);
    String value = getFullData(pathDataTlv);
    try {
        lfbHandler.config(lfbClassId, lfbInstanceId, path, value);
        this.lfbChangeListener.onLFBChange(lfbClassId+","+lfbInstanceId);
        tlvResult.addValue(ResultTlvValue.E_SUCCESS + ResultTlvValue.E_PADDING);
    } catch (Exception e) {
        e.printStackTrace();
        tlvResult.addValue(ResultTlvValue.E_UNSPECIFIED_ERROR + ResultTlvValue.E_PADDING);
    }

    Message configResponseMessage = new Message();
    Header header = new Header();
    header.setVersion(Header.VERSION);
    header.setRsvd(Header.RSVD);
    header.setMessageType(MessageType.CONFIG_RESPONSE);
    header.setLength(Header.LENGTH);
    header.setSourceId(feID);
    header.setDestId(ceID);
    header.setCorrelator(Header.IGNORED_CORRELATOR);
    HeaderFlags flags = new HeaderFlags();
    flags.setAck(HeaderFlags.NO_ACK);
    flags.setPriority(HeaderFlags.CONFIG_RESPONSE_PRIORITY);
    flags.setRsvd1(HeaderFlags.RSVD_FLAGS1);
    flags.setExecutionMode(HeaderFlags.EXECUTE_ALL_OR_NONE);
    flags.setAtomicTransaction(HeaderFlags.STANDALONE_MESSAGE);
    flags.setTransactionPhase(HeaderFlags.SOT);
    flags.setRsvd2(HeaderFlags.RSVD_FLAGS2);
    header.setFlags(flags);
    configResponseMessage.setHeader(header);

    TLV lfbSelectTlvResponse = new TLV(TlvType.LFBselect_TLV);
    lfbSelectTlvResponse.addValue(Binary.toBinaryString((int)lfbClassId, Binary.DWORD));
    lfbSelectTlvResponse.addValue(Binary.toBinaryString((int)lfbInstanceId, Binary.DWORD));
    TLV operTlvResponse = new TLV(OperTlvType.SET_RESPONSE);
    TLV pathDataTlvResponse = pathDataTlv;


    pathDataTlvResponse.addValue(tlvResult);
    operTlvResponse.addValue(pathDataTlvResponse);
    lfbSelectTlvResponse.addValue(operTlvResponse);
    configResponseMessage.setBody(lfbSelectTlvResponse);

    byte[] messageContent = configResponseMessage.serialize();
    logMessage("--- Send: Config Response Message---");
    Binary.logBits(messageContent, log);
    highPriorityChannel.send(messageContent);
}
```

Figura 65. Código del mensaje Config.

6.5.1.3.5. Mensaje Query

El mensaje Query utiliza en su instrucción con el comando query una variación respecto al comando config, en lugar de utilizar el comando path se utiliza el comando componentid. Para el

siguiente ejemplo la línea de comando es la siguiente: **query –classid 2 –instanceid 1 componentid 6**

Figura 66. Mensaje *Query Massage* y *Query Message Response*

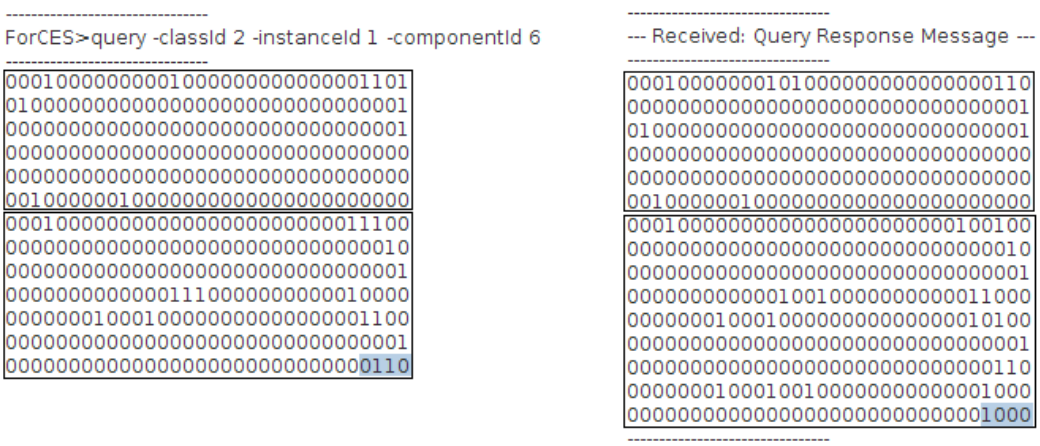


Figura 67. Formateo de datos del mensaje Query y Query Response

```
public void handleQuery(Message queryMessage) {
    TLV lfbSelectTlv = queryMessage.getBody();
    long lfbClassId = Binary.parseLong((String)lfbSelectTlv.getValue(0));
    long lfbInstanceId = Binary.parseLong((String)lfbSelectTlv.getValue(1));
    TLV operTlv = (TLV)lfbSelectTlv.getValue(2);
    TLV pathDataTlv = (TLV)operTlv.getValue(0);
    String path = getPath(pathDataTlv);
    Object value = null;
    try {
        value = lfbHandler.query(lfbClassId, lfbInstanceId, path);
    } catch (Exception e) {
        e.printStackTrace();
    }
    TLV fullDataTlv = parseFullDataTlv(value);

    Message queryResponseMessage = new Message();
    Header header = new Header();
    header.setVersion(Header.VERSION);
    header.setRsvd(Header.RSVD);
    header.setMessageType(MessageType.QUERY_RESPONSE);
    header.setLength(Header.LENGTH);
    header.setSourceId(feID);
    header.setDestId(ceID);
    header.setCorrelator(Header.IGNORED_CORRELATOR);
    HeaderFlags flags = new HeaderFlags();
    flags.setAck(HeaderFlags.NO_ACK);
    flags.setPriority(HeaderFlags.QUERY_RESPONSE_PRIORITY);
    flags.setRsvd1(HeaderFlags.RSVD_FLAGS1);
    flags.setExecutionMode(HeaderFlags.EXECUTE_ALL_OR_NONE);
    flags.setAtomicTransaction(HeaderFlags.STANDALONE_MESSAGE);
    flags.setTransactionPhase(HeaderFlags.SOT);
    flags.setRsvd2(HeaderFlags.RSVD_FLAGS2);
    header.setFlags(flags);
    queryResponseMessage.setHeader(header);

    TLV lfbSelectTlvResponse = new TLV(TlvType.LFBselect_TLV);
    lfbSelectTlvResponse.addValue(Binary.toBinaryString((int)lfbClassId, Binary.DWORD));
    lfbSelectTlvResponse.addValue(Binary.toBinaryString((int)lfbInstanceId, Binary.DWORD));
    TLV operTlvResponse = new TLV(OperTlvType.GET_RESPONSE);
    TLV pathDataTlvResponse = pathDataTlv;
    pathDataTlvResponse.addValue(fullDataTlv);
    operTlvResponse.addValue(pathDataTlvResponse);
    lfbSelectTlvResponse.addValue(operTlvResponse);
    queryResponseMessage.setBody(lfbSelectTlvResponse);

    byte[] messageContent = queryResponseMessage.serialize();
    logMessage("--- Send: Query Response Message---");
    Binary.logBits(messageContent, log);
    highPriorityChannel.send(messageContent);
}
```

Figura 68. Código del mensaje Query

6.5.1.3.6. Mensaje Heartbeat

Este mensaje es usado para realizar notificaciones asíncronas entre elementos ForCES el cual se encarga de enviar señales en un periodo de tiempo especifico con el objetivo de saber si la conexión entre el CE y el FE aún existe, haciéndolos sensibles al tráfico, o también para saber si el otro extremo está ahí o funciona realmente.

El mensaje *Heartbeat* tiene una pequeña diferencia con los otros mensajes, ya que este solo se compone de una cabecera común (*header*) y no tiene cuerpo del mensaje (*body*) este se encuentra vacio. **RFC5810**, pág. 82.

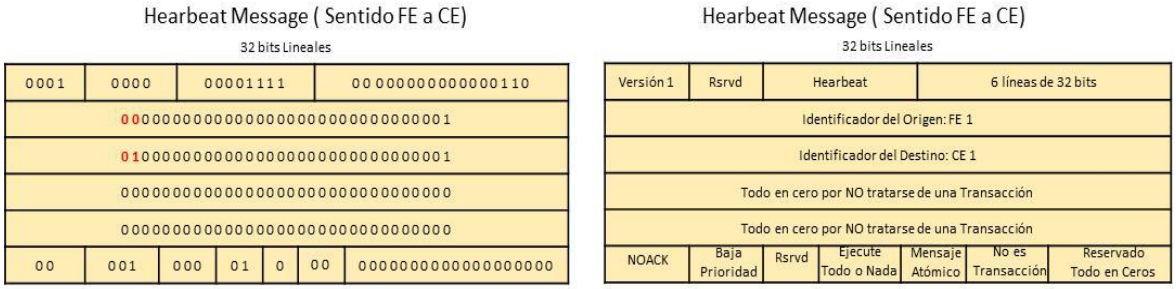


Figura 69. Mensaje *Heartbeat*

Un *Heartbeat* (HB), se envía entre el CE y el FE dentro del intervalo de tiempo que se programe, sin afectar que haya en el canal otro mensaje PL, el mensaje *Heartbeat* se inicia por parte del FE hacia el CE configurado previamente con el comando *Config*. En el FE residen los componentes LFBs de configuración del *Heartbeat*, por ello dicho comando se envía desde el CE para que posteriormente el FE comience a enviar los mensajes *Heartbeat*.

Se debe configurar tres componentes del *LFB Protocol Object* (FEPO) los cuales son: la política del *Heartbeat* del CE (CEHBPolicy, *the CE Heartbeat Policy*), la política del *Heartbeat* del FE (FEHBPolicy, *the FE Heartbeat Policy*), el intervalo de *Heartbeat* del FE que es en milisegundos (FEHI, *the FE Heartbeat Interval in the millisecs*). Estas políticas por defecto se encuentran en cero cuando se inician el CE y el FE lo cual significa que no se envía ningún mensaje *Heartbeat* por ello debe ser configurado a través del mensaje *Config*.

```
public void heartBeatMessage() {
    Message heartbeatMessage = new Message();
    Header header = new Header();
    header.setVersion(Header.VERSION);
    header.setRsvd(Header.RSVD);
    header.setMessageType(MessageType.HEARTBEAT);
    header.setLength(Header.LENGTH);
    header.setSourceId(feID);
    header.setDestId(ceID);
    header.setCorrelator(Header.IGNORED_CORRELATOR);
    HeaderFlags flags = new HeaderFlags();
    flags.setAck(HeaderFlags.NO_ACK);
    flags.setPriority(HeaderFlags.HEARTBEAT_PRIORITY);
    flags.setRsvd1(HeaderFlags.RSVD_FLAGS1);
    flags.setExecutionMode(HeaderFlags.EXECUTE_ALL_OR_NONE);
    flags.setAtomicTransaction(HeaderFlags.STANDALONE_MESSAGE);
    flags.setTransactionPhase(HeaderFlags.SOT);
    flags.setRsvd2(HeaderFlags.RSVD_FLAGS2);
    header.setFlags(flags);
    heartbeatMessage.setHeader(header);

    byte[] messageContent = heartbeatMessage.serialize();
    logMessage("--- Send: Heartbeat Message ---");
    Binary.logBits(messageContent, log);
    lowPriorityChannel.send(messageContent);
    associated = true;
}
```

Figura 70. Código del mensaje *Heartbeat*

6.5.1.3.6.1. Relación con el LFB Protocol Object (FEPO)

En la estructura XML del *LFB Protocol Object* se encuentran los componentes que definen las políticas de funcionamiento del mensaje *Heartbeat*, a través del mensaje *config* que puede cambiar estos parámetros para definir cómo va a operar el mensaje *Heartbeat*.

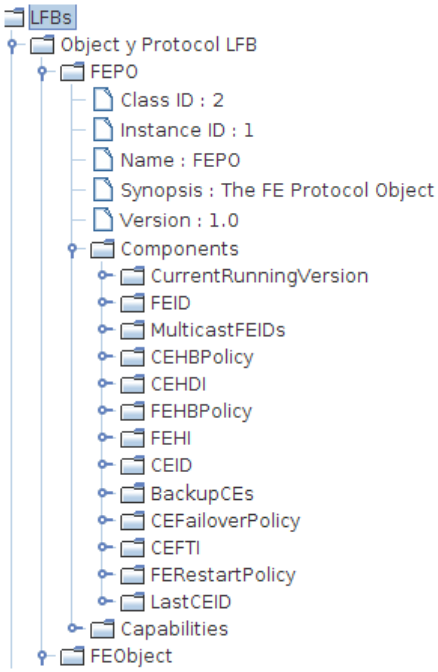


Figura 71. Políticas de operación LFB del mensaje *Heartbeat*

FEPO define varios temporizadores que se pueden utilizar en combinación con un mecanismo sensible parecido a los latidos del corazón, esto se realiza programando un hilo en java activado por el comando *config* y que se encarga exclusivamente de enviar el mensaje *Heartbeat* con el propósito de detectar la pérdida de conectividad entre TMLs, y la pérdida de la asociación entre los respectivos mensajes de la capa PL.

FEPO define una serie de políticas que también son programadas para definir el comportamiento o una pérdida detectada de asociación, el valor que se asigna a cada uno de los componentes LFBs para configurar el mensaje *Heartbeat* es el cero(0) para deshabilitar el mensaje y el uno(1) para habilitarlo, el tercer componente se fija en 500 milisegundos, que es el tiempo de separación entre mensajes *Heartbeat*. Ver RFC581, pág. 61- 62.

Estos componentes LFBs como se puede apreciar en la figura son: CEHBPolicy (componente 4), FEHBPolicy (componente 6), FEHI (componente 7), modificados a través del comando *config* y el componente CEHDI (componente 5), configurado por defecto dentro del hilo controlador.

Se crea un *script* que es un archivo de comandos *config* que van a configurar los tres componentes del *LFB Protocol*, el cual tiene la estructura XML en *LFBClassID*=2 e *InstanceID* =1, por que este tiene solamente una única instancia, el *Path* es la ruta del componente que se va a configurar esta es de una estructura simple , el *value* es el valor en que se carga en ese componente, la instrucción para modificar el valor del componente 4 llamado *CEHBPolicy* quedaría de la siguiente manera:
config –classid 1 –instanceid 1 –path 4 -value 1

El Hilo controlador de los mensajes *Heartbeat*(*Class HeartbeatController*) tiene valores por defecto con los que se inicializa el FE y son tomados de la especificación donde el intervalo de fin de la asociación por no recibir un mensaje *Heartbeat* dentro de un tiempo determinado es de 30 segundos, es decir, 30000 milisegundos. Apenas se inicializa el hilo, este empieza a correr y las políticas se encuentran en cero, evalúa la condición de que exista asociación entre el CE y el FE para que el mensaje pueda operar. Como en el caso inicial la política está en cero y la condición de asociación no existe entonces sigue haciendo el ciclo preguntando si las dos condiciones se cumplen para enviar el mensaje *Heartbeat*. Estas condiciones las evalúa cuando actualiza el estado del método de configuración del *Heartbeat* (*updateHeartbeatConfiguration*) va al FE y lee desde el LFB los valores que contenga los componentes.

```
/**
 * Constructor del hilo que recibe como parámetro una instancia del FE
 * @param fe Instancia del FE
 */
public HeartbeatController(FE fe) {
    this.fe = fe;
    heartBeatPolicy = FEHBPolicy0;
    heartBeatInterval = 500;
    heartBeatDeadInterval = 30000;
    thread = new Thread(this);
    running = true;
    thread.start();
}

@Override
public void run() {
    long threadIntervalTime = System.currentTimeMillis();
    long heartbeatIntervalTime = 0;
    while (running) {
        updateHeartbeatConfiguration();
        if (heartBeatPolicy == FEHBPolicy1 && fe.isAssociated()) {
            if (heartbeatIntervalTime >= heartBeatInterval) {
                fe.heartBeatMessage();
                heartbeatIntervalTime = 0;
                threadIntervalTime = System.currentTimeMillis();
            } else {
                heartbeatIntervalTime += (System.currentTimeMillis() - threadIntervalTime);
                threadIntervalTime = System.currentTimeMillis();
            }
        }
        try {
            Thread.sleep(100l);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public void updateHeartbeatConfiguration() {
    Map<String, Object> heartbeatMap = fe.getHeartbeatConfiguration();
    if (heartbeatMap.get("feHBPolicy") != null) {
        this.setHeartBeatPolicy((Integer)heartbeatMap.get("feHBPolicy"));
    }
    if (heartbeatMap.get("feHI") != null) {
        this.setHeartBeatInterval((Integer)heartbeatMap.get("feHI"));
    }
}
```

Figura 72. Hilo controlador del *Heartbeat*

La ruta de los componentes no es compleja como la de la configuración de las topologías debido a que cuando el FE llama al manejador de los LFBs este obtiene de manera directa el estado de la configuración ya que de manera fija se ha establecido dentro del código una ruta previa en donde saca la información del *LFBClassID*=2 e *InstanceID* =1. Esta es la única ruta pues no hay otra igual dentro de todo el esquema XML, esta información está contenida en los componentes 4,6 y 7.

```
public Map<String, Object> getHeartbeatConfiguration() {
    LFBInstance lfbInstance = lfbInstances.get("2,1");
    ComponentInstance ceHBPolicy = LFBHandler.getComponentByID(4, lfbInstance.getComponents());
    ComponentInstance ceHDI = LFBHandler.getComponentByID(5, lfbInstance.getComponents());
    ComponentInstance feHBPolicy = LFBHandler.getComponentByID(6, lfbInstance.getComponents());
    ComponentInstance feHI = LFBHandler.getComponentByID(7, lfbInstance.getComponents());
    AtomicInstance ceHBPolicyValue = (AtomicInstance)ceHBPolicy.getValue();
    AtomicInstance ceHDIValue = (AtomicInstance)ceHDI.getValue();
    AtomicInstance feHBPolicyValue = (AtomicInstance)feHBPolicy.getValue();
    AtomicInstance feHIValue = (AtomicInstance)feHI.getValue();
    Map<String, Object> heartbeatMap = new HashMap<String, Object>();
    if (ceHBPolicyValue.getValue() != null) {
        heartbeatMap.put("ceHBPolicy", Binary.parseInt((String)ceHBPolicyValue.getValue()));
    }
    if (ceHDIValue.getValue() != null) {
        heartbeatMap.put("ceHDI", Binary.parseInt((String)ceHDIValue.getValue()));
    }
    if (feHBPolicyValue.getValue() != null) {
        heartbeatMap.put("feHBPolicy", Binary.parseInt((String)feHBPolicyValue.getValue()));
    }
    if (feHIValue.getValue() != null) {
        heartbeatMap.put("feHI", Binary.parseInt((String)feHIValue.getValue()));
    }
    return heartbeatMap;
}
```

Figura 73. get*Heartbeat*Configuration

6.5.1.3.7. Mensaje Teardown

Este mensaje se utiliza para terminar la conexión y decir la razón por la cual se termina la conexión. La desconexión podría ser iniciada por cualquiera de las partes para fines administrativos, pero también puede ser impulsado por razones operativas, tales como la pérdida de conectividad.

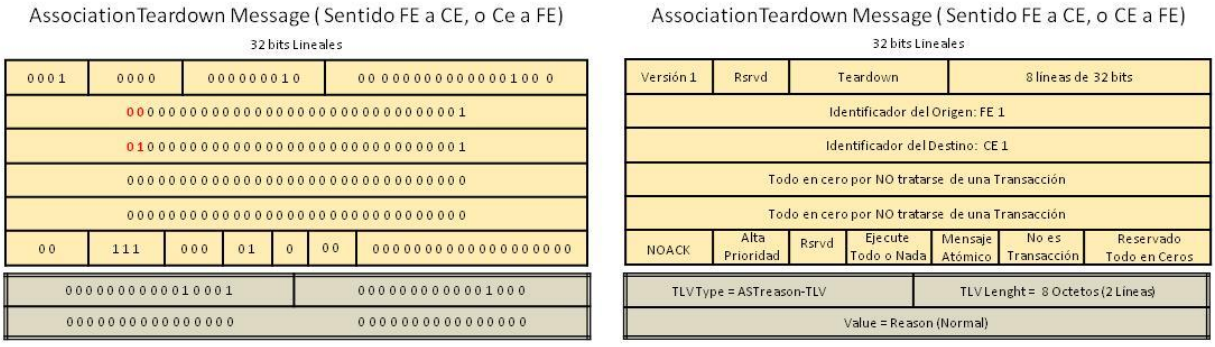


Figura 74. Mensaje Teardown

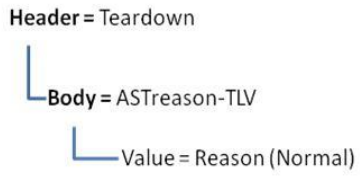


Figura 75. Ramificación del mensaje AsociationTeardown

```
public void associationTeardown() {
    Message associationTeardownMessage = new Message();
    Header header = new Header();
    header.setVersion(Header.VERSION);
    header.setRsvd(Header.RSVD);
    header.setMessageType(MessageType.ASSOCIATION_TEARDOWN);
    header.setLength(Header.LENGTH);
    header.setSourceId(feID);
    header.setDestId(ceID);
    header.setCorrelator(Header.IGNORED_CORRELATOR);
    HeaderFlags flags = new HeaderFlags();
    flags.setAck(HeaderFlags.NO_ACK);
    flags.setPriority(HeaderFlags.ASSOCIATION_TEARDOWN_PRIORITY);
    flags.setRsvd1(HeaderFlags.RSVD_FLAGS1);
    flags.setExecutionMode(HeaderFlags.EXECUTE_ALL_OR_NONE);
    flags.setAtomicTransaction(HeaderFlags.STANDALONE_MESSAGE);
    flags.setTransactionPhase(HeaderFlags.SOT);
    flags.setRsvd2(HeaderFlags.RSVD_FLAGS2);
    header.setFlags(flags);
    associationTeardownMessage.setHeader(header);

    TLV asTReasonTlv = new TLV(TlvType.ASTreason_TLV);
    asTReasonTlv.addValue(ASTreasonTLV.NORMAL);
    associationTeardownMessage.setBody(asTReasonTlv);
    header.setLength(calculateMessageLength(associationTeardownMessage));
    byte[] messageContent = associationTeardownMessage.serialize();
    logMessage("--- Send: Association Teardown Message ---");
    Binary.logBits(messageContent, log);
    highPriorityChannel.send(messageContent);
    associated = false;
}
```

Figura 76. Código mensaje Teardown

6.5.2. CAPA TML

La capa TML se encarga de transportar los mensajes que son encapsulados en la capa PL y que corresponden al protocolo ForCES, una vez se ha construido el mensaje que se desea enviar, la capa PL le pasa este mensaje a la capa TML quien se encarga de usar un protocolo de transporte (TCP, UDP, SCTP, DCCP) sobre cualquiera de las tecnologías de interconexión subyacentes (*Ethernet*, *Backplane*, *ATM*), de los cuales el que se ha seleccionado por parte del organismo internacional ForCES y como consta en el **RFC5810** y RFC5811cómo protocolo de transporte para los mensajes es el protocolo SCTP, y como tecnología de interconexión el protocolo *Ethernet* debido a que este proyecto basa su implementación en tarjetas de red ubicadas en chasis diferentes.

El API del protocolo SCTP que es utilizado en este proyecto como lo exige el RFC se implementa a través de los sistemas operativos: Linux o Solaris, debido a que hasta el momento la API SCTP sólo está implementada en estos dos. Este proyecto se desarrollo en Ubuntu versión 9.10, usando el Java

JDK 7, pero como todavía no había sido lanzada una versión completa que incluyera esta API, fue necesario usar un Snapshot Release, llamado *build b83* del 12 de Febrero de 2010.

El JDK 7 trae una implementación del Stream Control Transport Protocol (SCTP) el cuál es el protocolo obligado por la especificación para la implementación de la Transport Mapping Layer (TML).

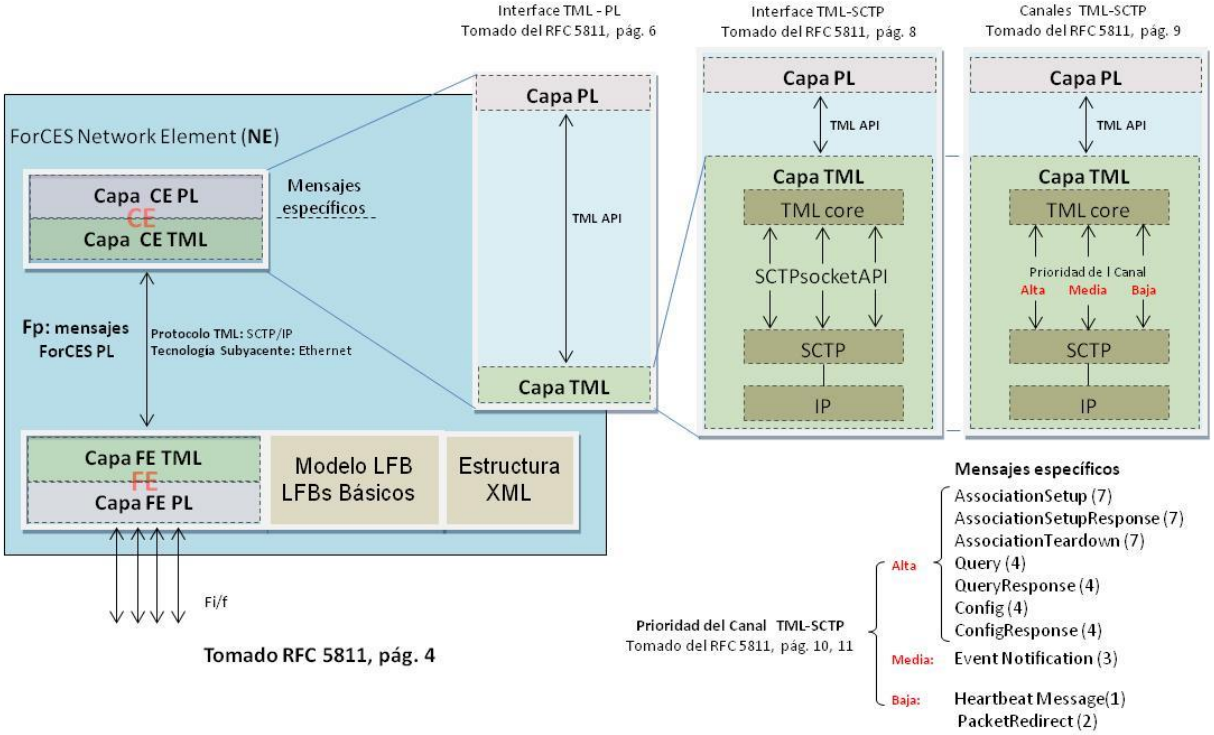


Figura 77. Capa TML

Como se puede observar en la figura ambos elementos CE y FE se componen de dos capas, la capa PL y TML. La capa TML se comunica con la capa PL a través de una interfaz que transfiere los mensajes ya encapsulados para que TML los transporte. TML contiene internamente un API SCTP que se encarga de realizar la conexión de datos entre el CE y el FE. En el CE la aplicación programada se queda escuchando dentro tres hilos, que corresponden cada uno a los tres puertos que exige la especificación y por los cuales se envían los mensajes PL dependiendo de la prioridad.

En el programa que se desarrollo se utilizaron métodos para serializar estos mensajes y enviarlos como datagramas una vez establecida la conexión. Dichos datagramas contienen en su cabecera un campo en la línea seis, la cual está dedicada a las banderas del mensaje que indica el tipo de prioridad que permite decidir por cual canal viajara la información. El establecimiento de cada uno de estos canales se adjudican al número de puerto según su prioridad: el puerto 6704 asignado a mensajes de alta prioridad, el 6705 mediana prioridad y 6706 baja prioridad.

El orden de establecimiento según el protocolo ForCES se encuentra en segundo lugar después de realizarse la pre asociación la cual tiene la información de identificación y direccionamiento IP de

los CEs y FEs que se desean conectar. El programa reconoce esta información luego de ejecutar el comando TMLSetup que realiza el proceso anteriormente mencionado.

6.5.2.1. Protocolo SCTP

SCTP es un protocolo de transporte de extremo a extremo que es equivalente a TCP, UDP, o DCCP en muchos aspectos. SCTP tiene un poco de cada uno de estos protocolos, está orientado a la conexión y al intercambio de datagramas. Por tanto al estar orientado al modelo cliente-servidor, en donde el servidor es el elemento CE y el Cliente el elemento FE, se utilizara el Diseño de una aplicación Socket basada en el modelo cliente-servidor.

El establecimiento del canal se logro haciendo uso del API SCTP del OS Linux, razón por la cual esta implementación se desarrollo en esta plataforma.

```
public void run() {
    ce.logMessage("Canal SCTP esperando por conexión en el puerto " + port);
    try {
        //SctpServerChannel serverChannel = SctpServerChannel.open();
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        InetSocketAddress inetSocketAddress = new InetSocketAddress(Integer.parseInt(port));
        //serverChannel.bind(inetSocketAddress);
        serverChannel.socket().bind(inetSocketAddress);
        // Método bloqueante
        channel = serverChannel.accept();
        ce.logMessage("CE Channel(" + port + ") Conectado");
        while (running) {
            ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
            // Método bloqueante
            //MessageInfo messageInfo = channel.receive(byteBuffer, null, null);
            int numBytes = channel.read(byteBuffer);
            //byte[] messageContent = new byte[messageInfo.bytes()];
            byte[] messageContent = new byte[numBytes];
            byte[] receivedStream = byteBuffer.array();
            for (int i = 0; i < messageContent.length; i++) {
                messageContent[i] = receivedStream[i];
            }
            ce.handleMessage(messageContent);
        }
    } catch (Exception e) {
        e.printStackTrace();
        ce.logMessage("Error: " + e.getMessage());
    }
}
```

Figura 78. Código protocolo SCTP en el CE

```
public void run() {
    fe.logMessage("Conectando...");
    try {
        InetSocketAddress serverAddress = new InetSocketAddress(ip, Integer.parseInt(port));
        //channel = SctpChannel.open(serverAddress, 0, 0);
        channel = SocketChannel.open(serverAddress);
        fe.logMessage("FE Channel(" + port + ") Conectado");
        while (running) {
          ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
          // Método bloqueante
          //MessageInfo messageInfo = channel.receive(byteBuffer, null, null);
          int numBytes = channel.read(byteBuffer);
          //byte[] messageContent = new byte[messageInfo.bytes()];
          byte[] messageContent = new byte[numBytes];
          byte[] receivedStream = byteBuffer.array();
          for (int i = 0; i < messageContent.length; i++) {
              messageContent[i] = receivedStream[i];
          }
          fe.handleMessage(messageContent);
        }
    } catch (Exception e) {
        e.printStackTrace();
        fe.logMessage("Error: " + e.toString());
    }
}
```

Figura 79. Código protocolo SCTP en el FE

6.5.3. CONSTRUCCIÓN DE LA INSTRUCCIÓN (*config y query*)

La Topología de los LFBs, están definidas dentro del *FE Object*, el cual es el LFB con *Class ID = 1* e *Instance ID = 1*, por eso es que en todos los comandos de configuración (*config*) y consulta (*query*), van a afectar únicamente a este LFB, el cual se encarga de conectar todos los demás LFBs, es decir, el XML de la *librería LFB* propuesto en el **draft-ietf-forces-lfb-lib-05** contiene los LFBs necesarios para armar una funcionalidad *IPv4 Reenvío* y una *ARP processing*, pero estos LFBs no saben cómo están conectados; quién los organiza y conoce como están conectados es el LFB *FE Object* a través del componente *LFBTopology*, de manera que la primera parte de la instrucción que modifica a este LFB se hace con el comando *config* y quedaría así: **config –classid 1 –instanceid 1**

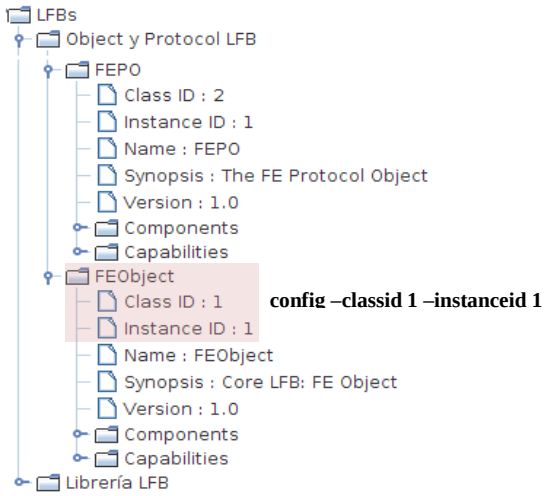


Figura 80. Class ID y Instance ID de los LFBs Básicos (FEPO y FE Object)

Todos los comandos *config* modifican los componentes de un LFB a través del comando *path* (ver componentes Anexo A), para modificar el componente uno (1) llamado *LFBTopology* se usa la primera opción del comando *path* en 1, que indica el primer componente del LFB que se está direccionando, para este caso es el *FE Object*, y cada una de las posiciones que sean necesarias dependiendo del número de elementos que estén anidados uno dentro del otro en el árbol XML del LFB, este árbol se dibuja para mayor claridad en forma de carpetas, cada carpeta o elemento dentro del árbol estará separado por una coma (,) en el comando *path* y los elementos que se cuentan para construir la instrucción son las carpetas que se encuentran una dentro de la otra comenzando desde el 1 hasta las n carpetas que se contengan, quedando de esta manera concadenado el comando *path* a la instrucción anterior del comando *config* así: **config –classid 1 –instanceid 1 –path 1,**

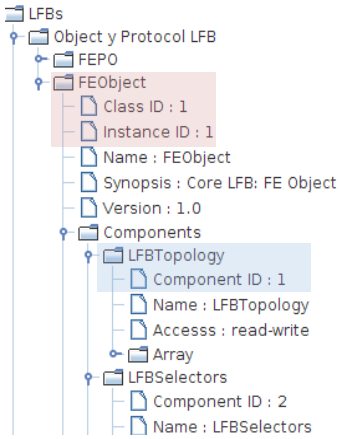


Figura 81. Componentes del LFBObject

La segunda posición del comando *path* hace referencia al índice del elemento dentro del arreglo (*Array*), el cual se crea cuando se le indica en el comando *path* el numero de índice con el que quedara creada, usando la clase *clon* de la que se hablo en la sección 6.4.2.1.2 las estructuras del arreglo se enumeran por carpetas con la excepción de que aquí se comienza desde cero (0) hasta las (n) carpetas *struct* que sean necesarias. Cada nueva estructura es una copia exacta de la estructura cero (0) como se explica en la figura 82 y contiene internamente los siguientes valores: el origen, el puerto de origen, el destino y el puerto de destino.

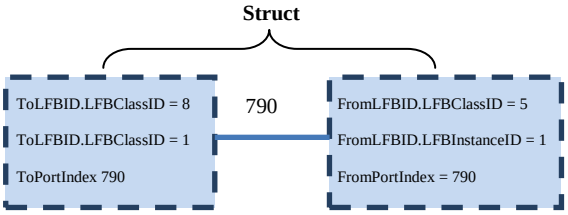


Figura 82. Una estructura LFB (Struct)

Como se ve en el ejemplo de la siguiente figura, el índice 4 crea la siguiente estructura del arreglo, completando la instrucción así: **config –classid 1 –instanceid 1 –path 1,4**

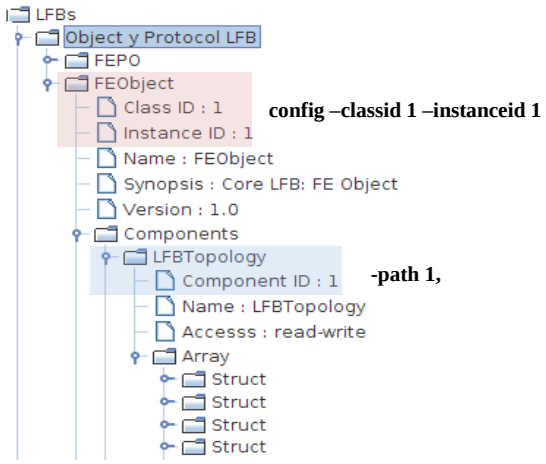


Figura 83. Creación de una nueva estructura (Struct)

Dentro de este elemento (4) *struct* o estructura, se encuentran las entradas y salidas del nodo LFB como se vio en la sección 6.4.2.1.2, para configurarlas se necesita una instrucción por cada elemento de entrada y salida, que llevará los valores (*value*) desde el CE al LFB que se quiere configurar en el FE, por esta razón una estructura (*struct*) requiere de 6 instrucciones de configuración para lograr una conexión completa entre dos nodos.

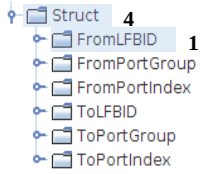


Figura 84. Entradas y Salidas del nodo LFB

La tercera posición del comando *path* configura el elemento de entrada *FromLFBID* de la estructura y la instrucción que se está construyendo quedaría así: **config –classid 1 –instanceid 1 –path 1,4,1**

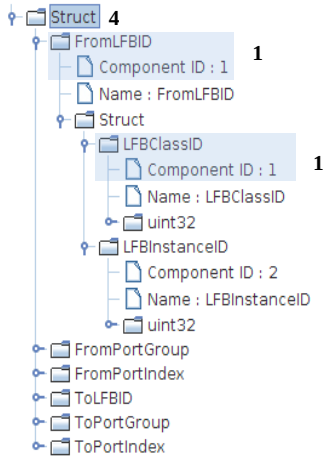


Figura 85. Componente LFBClassID

Dentro de la carpeta *FromLFBID* se encuentra el identificador del elemento *LFBClassID*, el cual es el último elemento que se direcciona en el *Path* en la cuarta posición con el numero 1, de esta manera queda la instrucción así: **config –classid 1 –instanceid 1 –path 1,4,1,1**

La instrucción value pone el valor al final de todo el direccionamiento como se ve en la figura y la instrucción final que se debe enviar desde el CE al FE es la siguiente: **config –classid 1 –instanceid 1 –path 1,4,1,1 –value 8.**

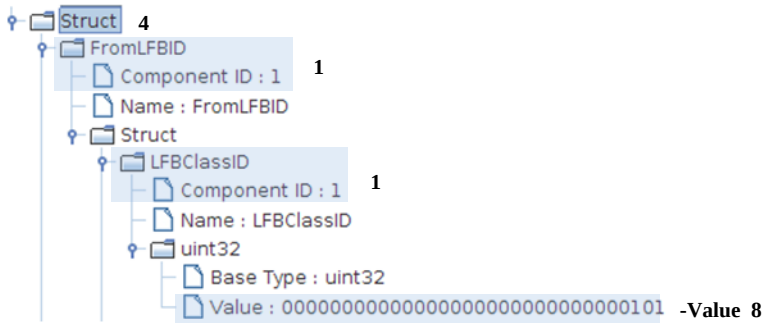


Figura 86. Componente valor (value)

Lo anterior se refiere a la construcción paso a paso de una instrucción Config enviada del CE al FE donde se muestra toda la ruta de direccionamiento del LFB que se desea configurar. Cada vez que se hace una configuración de un LFB en el FE se deben enviar una a una las instrucciones, para formar un struct entre dos LFBs se debe enviar las siguientes líneas de comando:

# LFB 5,1 a LFB 8,1 puerto 790					
# Origen		1	4	1	1 value
config -classId 1 -instanceId 1 -path 1,4,1,1 -value 8	#FEObject.LFBTopology.Struct.FromLFBID.LFBClassID =				8
config -classId 1 -instanceId 1 -path 1,4,1,2, -value 1	#FEObject.LFBTopology.Struct.FromLFBID.LFBInstanceID =				1
config -classId 1 -instanceId 1 -path 1,4,3, -value 790	#FEObject.LFBTopology.Struct.FromPortIndex =				790
# Destino					
config -classId 1 -instanceId 1 -path 1,4,4,1, -value 8	#FEObject.LFBTopology.Struct.ToLFBID.LFBClassID =				8
config -classId 1 -instanceId 1 -path 1,4,4,2, -value 1	#FEObject.LFBTopology.Struct.ToLFBID.LFBClassID =				1
config -classId 1 -instanceId 1 -path 1,4,6, -value 790	#FEObject.LFBTopology.Struct.ToPortIndex=				790

Tabla 1. Líneas de comando

7. PRUEBAS DE INTEROPERABILIDAD DEL PROTOCOLO

7.1. ESCENARIOS DE PRUEBA

El **draft-ietf-forces-interoperability-04** contiene los escenarios de prueba necesarios para evaluar el funcionamiento del protocolo, dichos escenarios consisten en el establecimiento de la conexión y los mensajes ForCES de la capa PL que se intercambian entre el elemento CE y el elemento FE. Este draft permite comprobar la interoperabilidad entre estos dos elementos haciendo las pruebas una a una secuencialmente incluyendo un escenario de ejemplo.

7.1.1. ESCENARIO 1 - PRE-ASSOCIATION SETUP

En este escenario se escribe el comando preassociationsetup en ambas consolas tanto en el CE como en el FE, sin un orden preestablecido, se puede hacer desde cualquier consola.

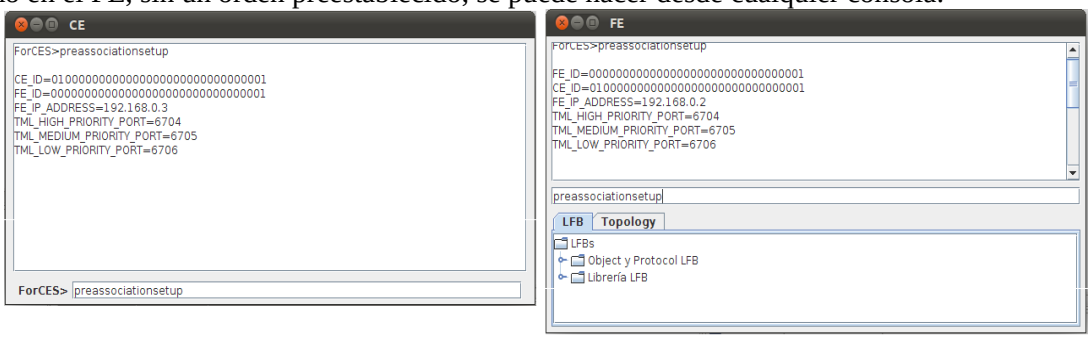


Figura 87. Escenario de Preasociación

Como se observa en la figura en la consola del CE se lee el archivo CE.properties y se visualiza su contenido en la ventana, de igual manera sucede en la consola del FE con la diferencia de que en esta consola se habilita el árbol de carpetas de las librerías LFBs.

7.1.2. ESCENARIO 2 - TML PRIORITY CHANNEL CONNECTION

Una vez cargados los datos de la fase de preasociación anteriormente vistos se escribe el comando tmlsetup, teniendo en cuenta que se debe ejecutar primero en la consola del CE, pues este es el maestro de la conexión y luego se debe ejecutar en el FE.

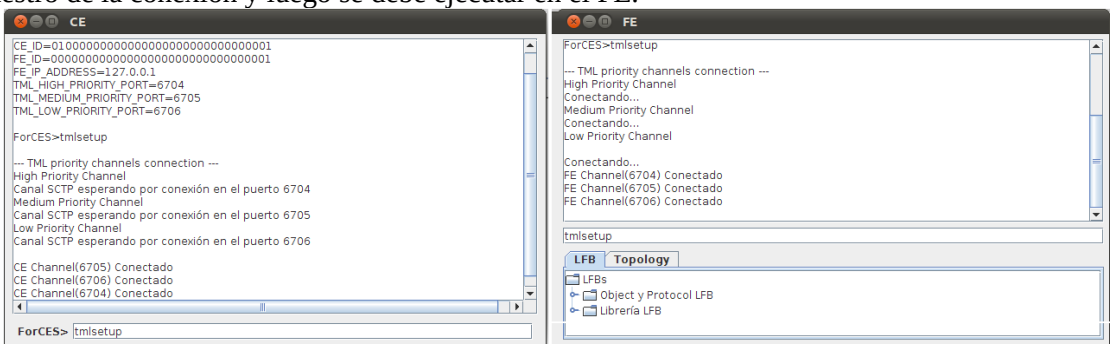


Figura 88. Escenario TMLSetup

Al ejecutar el comando en el CE este se queda escuchando a través de sus puertos hasta que el FE realice la conexión, una vez se establece la conexión se observa el mensaje conectado en cada uno de los tres puertos.

7.1.3. ESCENARIO 3 - ASSOCIATION SETUP - ASSOCIATION COMPLETE

La asociación se realiza mediante el mensaje Association Setup que es enviado desde el FE hacia el CE únicamente utilizando el comando associationsetup y el CE envía en respuesta un mensaje Association Setup Response, a partir de este momento ambos elementos CE y FE se encuentran asociados.

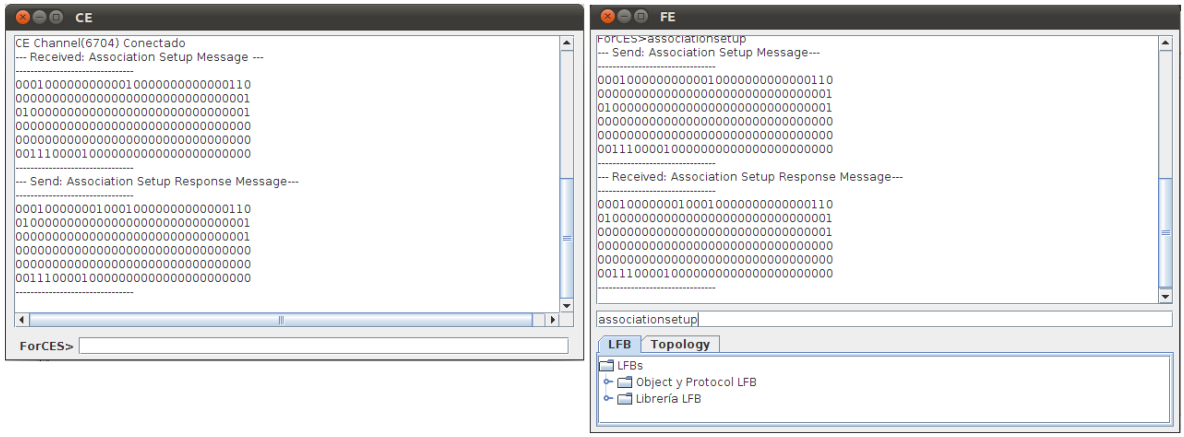


Figura 89. Escenario de asociación

7.1.4. SCENARIO 4 - CE QUERY

Mediante el mensaje *Query* el elemento CE conoce cuales son las librerías LFBs y su estado, disponibles en el elemento FE. Desde la consola CE se envía el comando Query con las instrucciones que permiten traer la información del FE al CE. Para extraer dicha información se direcciona dentro del árbol LFB al igual que la instrucción Config.

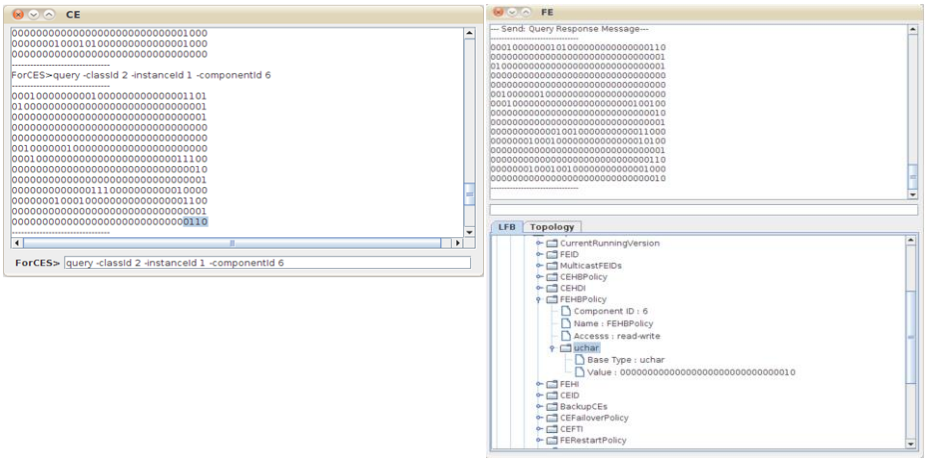


Figura 90. Escenario CE QUERY

7.1.5. SCENARIO 5 - HEARTBEAT MONITORING

Uno de los requisitos para que se habilite el monitoreo *Heartbeat*, es que exista una asociación entre los elementos CE y FE, una vez la asociación existe se debe hacer uso del mensaje *Config* para habilitar en el FE el envío del mensaje de monitoreo *Heartbeat.*

```
# LFB 2,1 The FE Protocol Object

# CEHBPolicy: 0-CEHBPolicy0, 1-CEHBPolicy1
config -classId 2 -instanceId 1 -path 4 -value 1

#CEHDI
config -classId 2 -instanceId 1 -path 5 -value 60000

# FEHBPolicy: 0-FEHBPolicy0, 1-FEHBPolicy1
config -classId 2 -instanceId 1 -path 6 -value 1

# FEHI
config -classId 2 -instanceId 1 -path 7 -value 5000
```

```
# LFB 2,1 The FE Protocol Object

# CEHBPolicy: 0-CEHBPolicy0, 1-CEHBPolicy1
config -classId 2 -instanceId 1 -path 4 -value 0

#CEHDI
config -classId 2 -instanceId 1 -path 5 -value 0

# FEHBPolicy: 0-FEHBPolicy0, 1-FEHBPolicy1
config -classId 2 -instanceId 1 -path 6 -value 0

# FEHI
config -classId 2 -instanceId 1 -path 7 -value 0
```

Figura 91. Scripts de configuración del mensaje Heartbeat

En la figura anterior se muestra la línea de comando de configuración del *Heartbeat* necesarias para habilitar o deshabilitar este monitoreo, por tratarse de tres líneas de comando se puede hacer uso de un *script* para realizar la configuración.

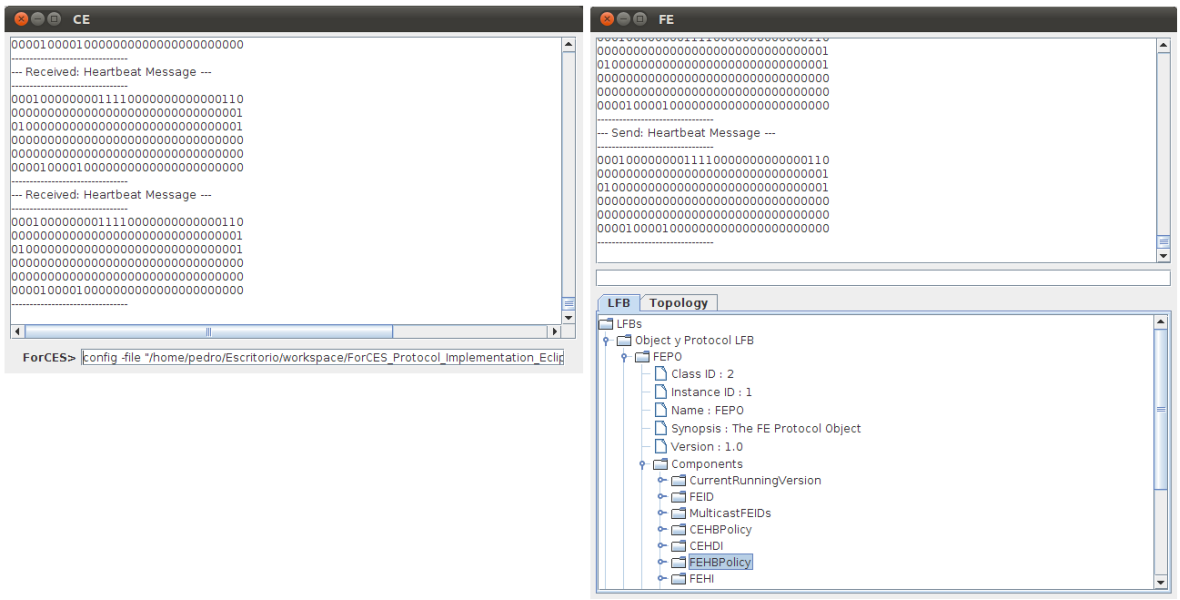


Figura 92. Configuración del mensaje Heartbeat

Una vez ejecutado el comando se observa en la ventana de las consolas que el mensaje de monitoreo *Heartbeat* es enviado por el FE cada 500 milisegundos. Cuando se envían las instrucciones para habilitar o deshabilitar el *Heartbeat* los valores del componente *Policy* del *FEProtocolObject* LFB se modifican.

7.1.6. SCENARIO 6 - SIMPLE CONFIG COMMAND

El comando *Config* se compone de cuatro elementos: un *ClassID*, un *InstanceID*, un *Path* y un *Value*, para enviar datos desde el CE al FE. Como se muestra en la figura se envía una instrucción *Config* simple que configura dos LFBs para formar un *struct.*

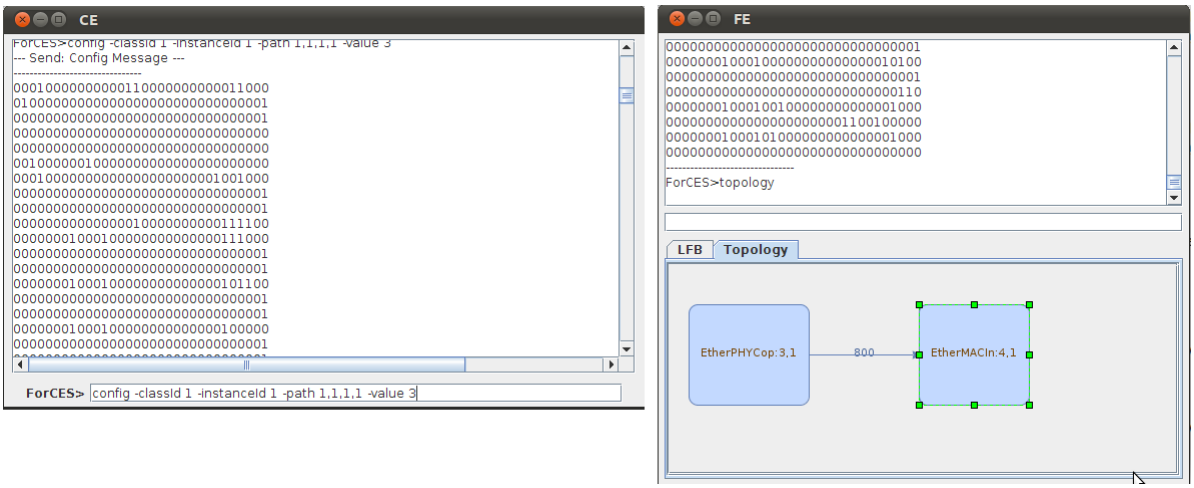


Figura 93. Configuración de un comando simple

Una vez los valores del árbol de carpetas LFBs son asignados, el siguiente paso es verificar que el diseño de conexión de los LFBs se ha hecho correctamente, para ello se escribe el comando Topology en la pestaña *topology* de la consola FE, y de esta manera observan cómo se dibujan los LFBs configurados.

7.1.7. SCENARIO 7 - ASSOCIATION TEARDOWN

El mensaje teardown se puede enviar desde cualquiera de las dos consolas el único requisito es que estén asociados los dos elementos de red CE y FE.

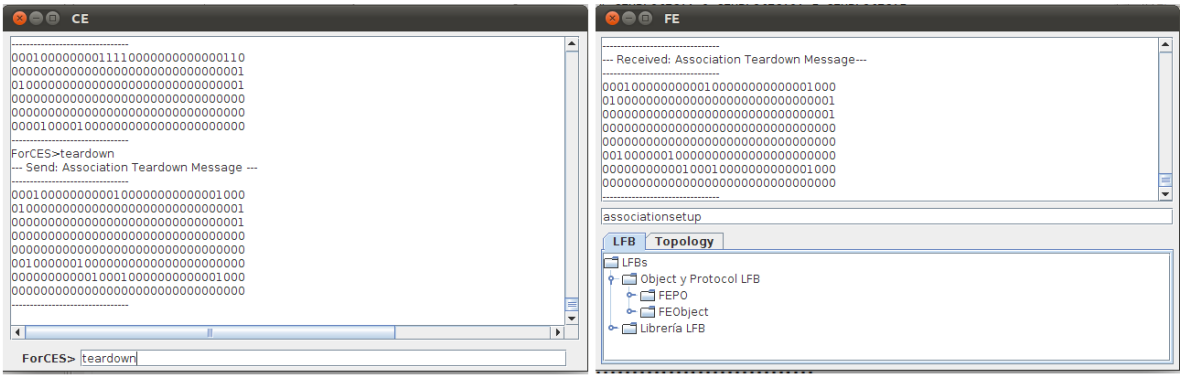


Figura 94. Escenario Teardown

Una vez se envía este mensaje ocurre la desconexión lógica del canal, dentro del mensaje viaja la información del evento que genero la desconexión, para el caso de este proyecto la razón de desconexión es *normal*.

7.2. PRUEBA CON EL ANALIZADOR DE PROTOCOLO (Wireshark)

En la pagina 10 y 12 del **draft-ietf-forces-interoperability-04,** se conecta en un escenario de prueba consistente en un solo elemento CE asociado a varios elementos FEs a través de una red LAN compuesto por un *switch* y un analizador de protocolos. En este proyecto que consta de solo

un CE y un FE asociados mediante una conexión Ethernet, se utiliza en uno de los PCs que se esta usando como CE o como FE el software analizador de protocolo *Wireshark*, que se pone a escuchar en el mismo puerto Ethernet por donde se esta conectando con el otro PC.

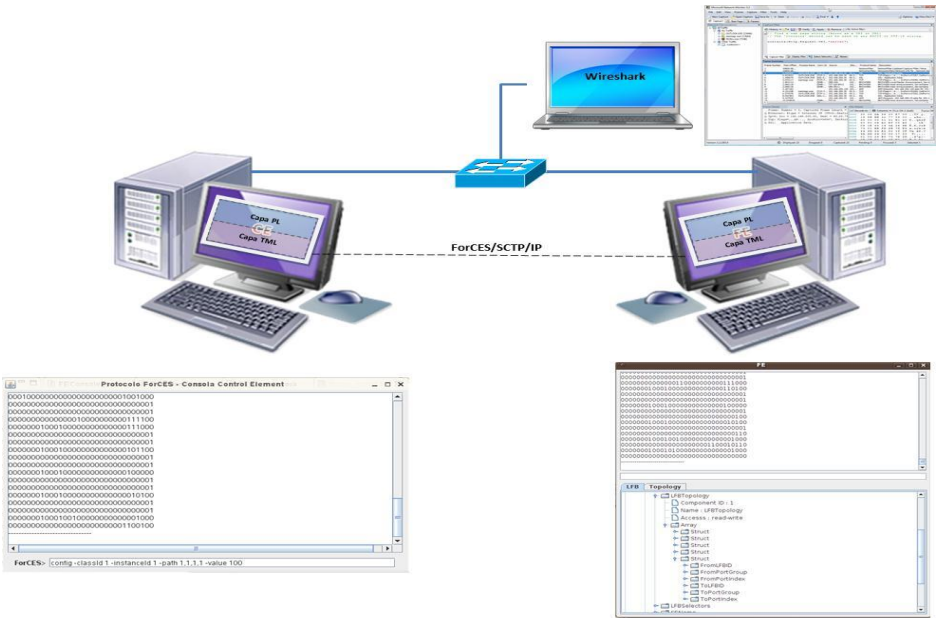


Figura 95. Analizador de protocolos Wireshark

Para efectos de pruebas se realizo la captura con el analizador de protocolos con el localhost y prubas sobre el mismo equipo a la dirección 127.0.0.1, para la activación del wireshark en Linux utilizamos el comando sudo wireshark –i lo –k, lo:localhost. Y se realizaron las siguientes pruebas:

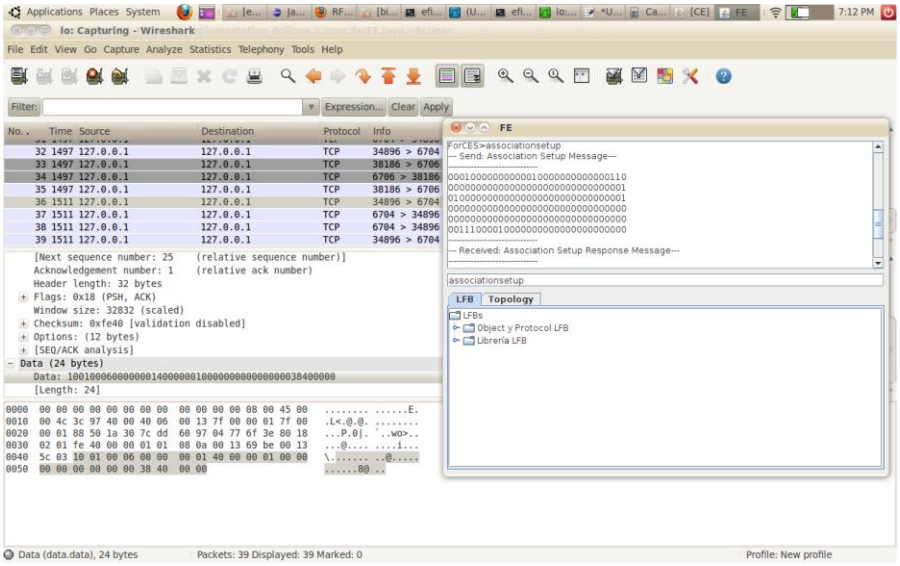


Figura 96. Capturas con wireshark del mensaje associationsetup

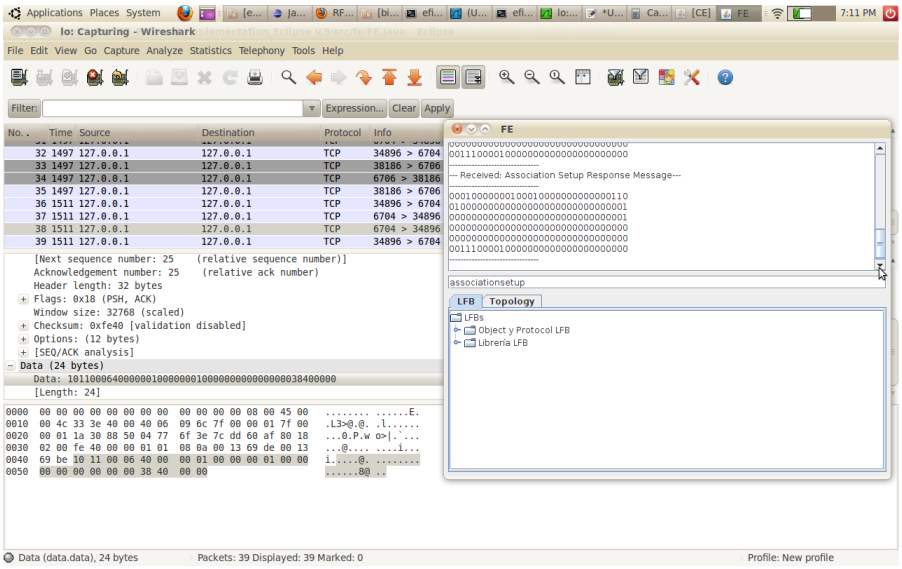


Figura 97. Capturas con wireshark del mensaje associationsetup response

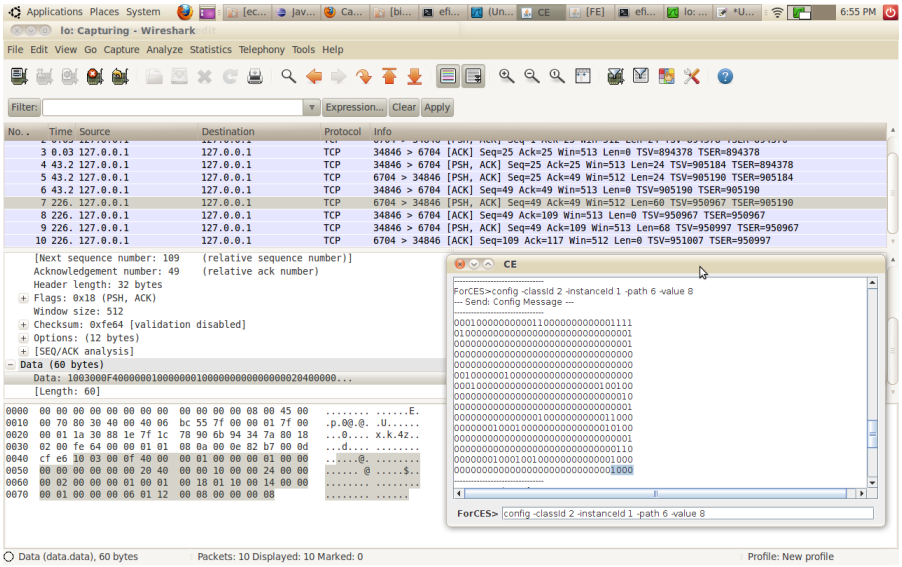


Figura 98. Capturas con wireshark del config

En las anteriores gráficas se puede apreciar las capturas del mensaje config, en donde se compara lo que se captura con el analizador de protocolos y la ventana CE desde donde se envía el comando config, la captura se hace bajo la encapsulación del protocolo SCTCP. Los datos que se comparan son los que se envían a través del protocolo ForCES.

En las siguientes gráficas se puede ver y comparar el tamaño del paquete, el cual corresponde efectivamente a los datos que se capturan con los que se ven en la ventana del CE.

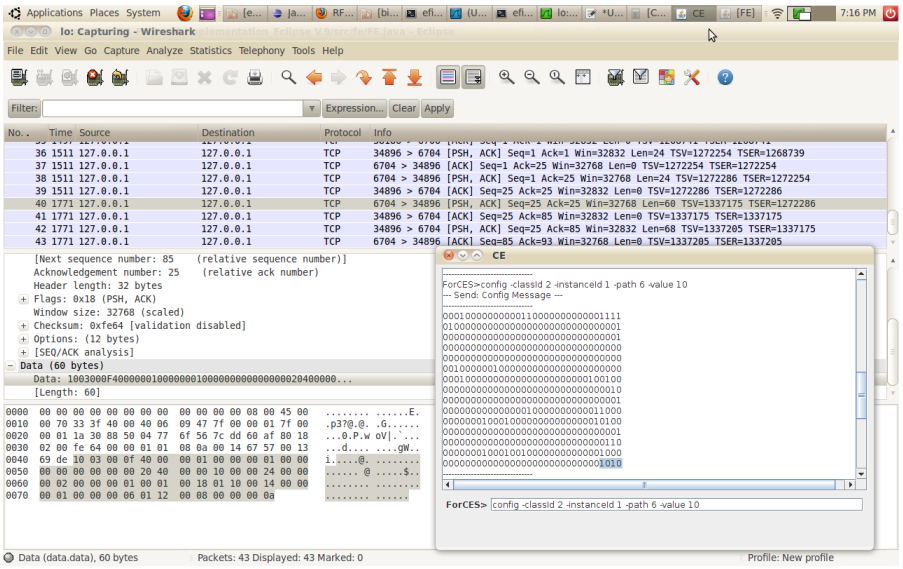


Figura 99. Capturas con wireshark comparación del tamaño del paquete ForCES

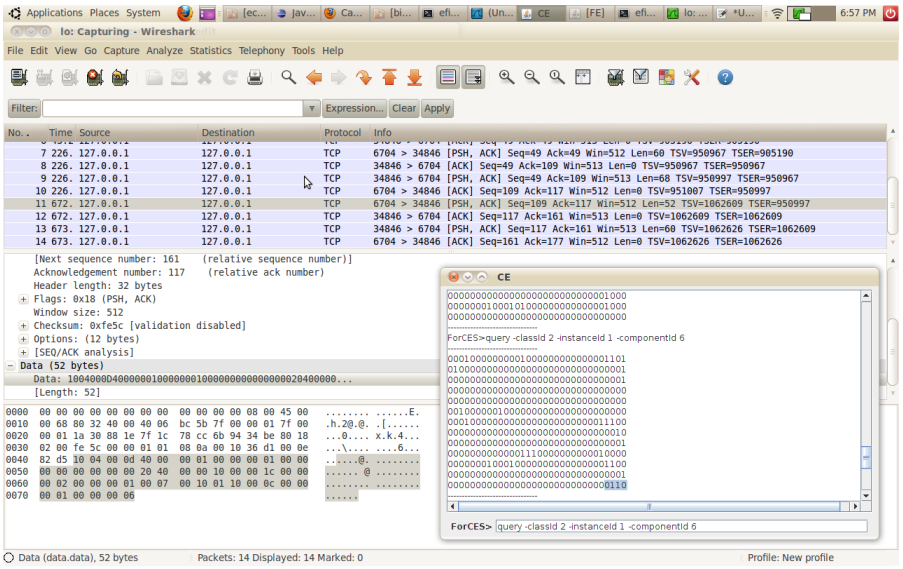


Figura 100. Capturas con wireshark del mensaje query

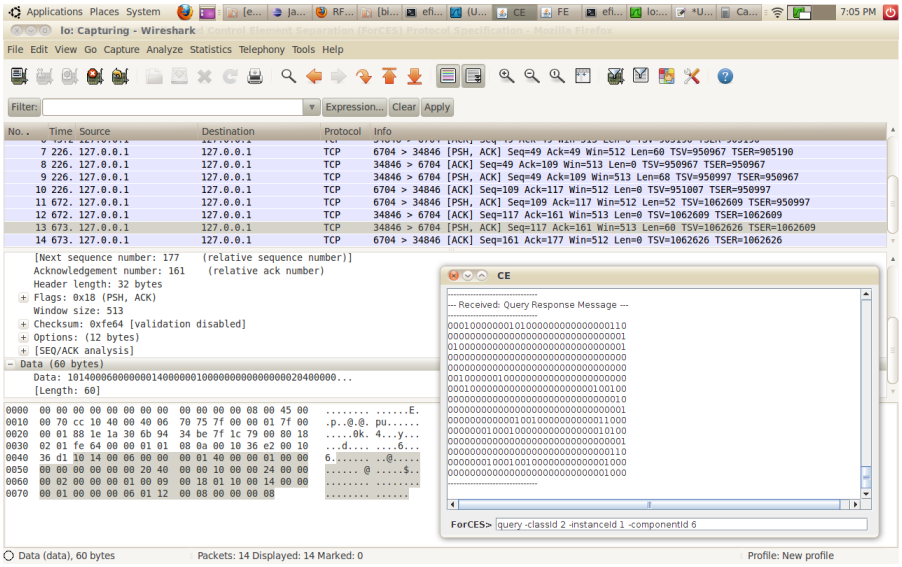


Figura 101. Capturas con wireshark del mensaje query response

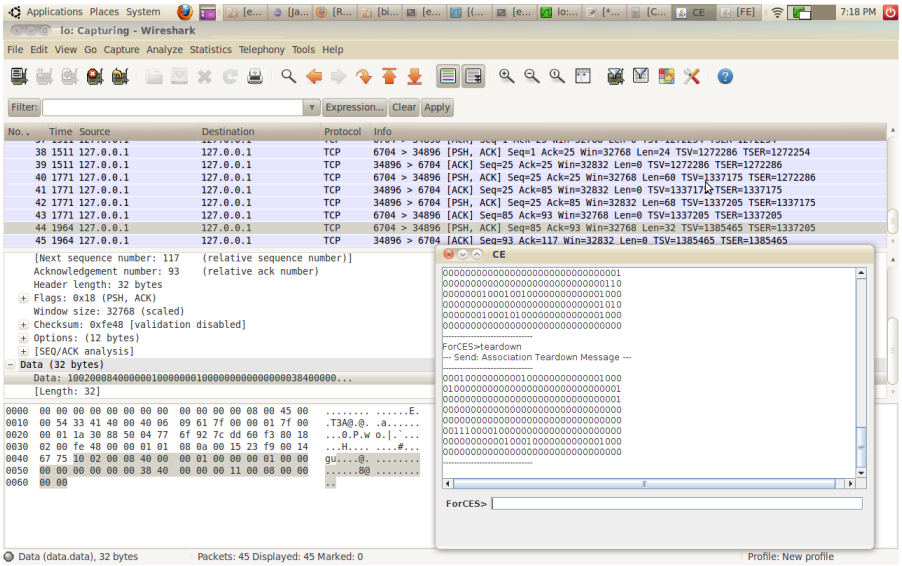


Figura 102. Capturas con wireshark del mensaje Association Teardown

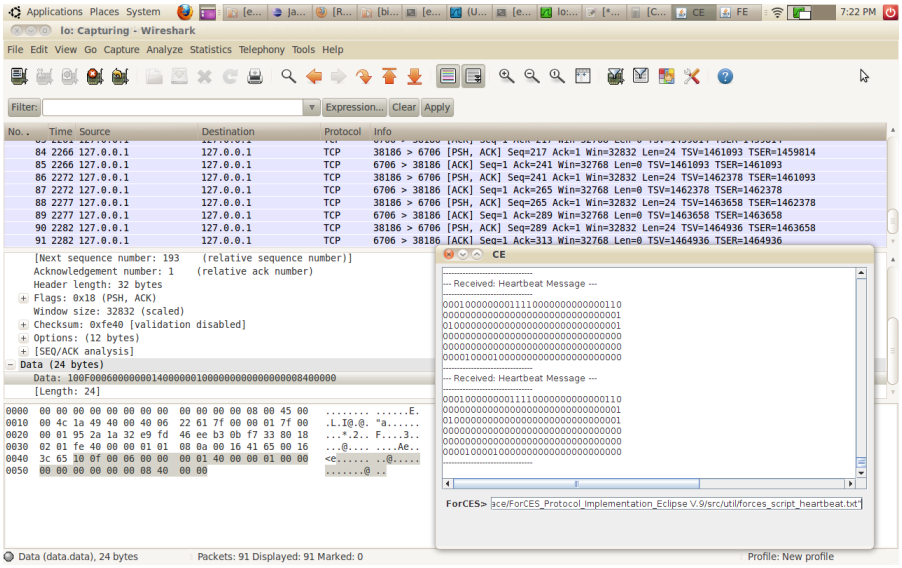


Figura 103. Capturas con wireshark del mensaje *Heartbeat*

7.3. EJEMPLO DE PRUEBA FUNCIONALIDAD (IP v4 REENVÍO)

Una vez terminado el protocolo ForCES que conecta al elemento CE y FE y se realizan las pruebas de interoperabilidad a través de escenarios como lo propone el **draft-ietf-forces-interoperability-04** se procede a usar el **draft-ietf-forces-lfb-lib-05** que contiene las librerías LFBs IP v4 *Reenvío* y ARP. Para el caso de este ejemplo se utilizo la librería LFB IP v4 *Reenvío* siguiendo en la especificación la figura de la pagina 12 donde se muestra de manera grafica la interconexión de todos los bloques lógicos funcionales del IP v4 *Reenvío* para un puerto Ethernet.

Desde la consola del CE se envían las líneas de comando que llevan la información a cada uno de los *struct* creados dentro del componente *LFBTopology* (ver anexo C), los cuales son los valores correspondientes para seleccionar los LFBs que se van a usar, esta selección se realiza a través de los índices, InstanceID y ClassID y los valores de los puestos de entrada y salida que permitirán la conexión de los LFBs. De toda la gama de LFBs que contiene esta librería que son en total 15 componentes LFBs, solo se seleccionan para este ejemplo un grupo de 9 LFBs, los cuales son los necesarios para crear la funcionalidad IP v4 *Reenvío*.

El numero de líneas de comando necesarias para hacer toda la conexión y armar la topología funcional de este grupo de LFBs es de 48 líneas que se dividen así: tres instrucciones para el LFB de origen y tres instrucciones para el LFB de destino, generando nueve grupos de seis instrucciones debido a que son muchas instrucciones se crea un *script* para enviar con el comando Config.

```
# LFB 3,1 a LFB 4,1 puerto 800 (Ether PHY co #1 a Ether MACIn)
# Origen
config -classId 1 -instanceId 1 -path 1,1,1,1 -value 3
config -classId 1 -instanceId 1 -path 1,1,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,1,3, -value 800
# Destino
config -classId 1 -instanceId 1 -path 1,1,4,1, -value 4
config -classId 1 -instanceId 1 -path 1,1,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,1,6, -value 800

# LFB 4,1 a LFB 5,1 puerto 801
# Origen
config -classId 1 -instanceId 1 -path 1,2,1,1 -value 4
config -classId 1 -instanceId 1 -path 1,2,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,2,3, -value 801
# Destino
config -classId 1 -instanceId 1 -path 1,2,4,1, -value 5
config -classId 1 -instanceId 1 -path 1,2,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,2,6, -value 801

# LFB 5,1 a LFB 8,1 puerto 802
# Origen
config -classId 1 -instanceId 1 -path 1,3,1,1 -value 5
config -classId 1 -instanceId 1 -path 1,3,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,3,3, -value 802
# Destino
config -classId 1 -instanceId 1 -path 1,3,4,1, -value 8
config -classId 1 -instanceId 1 -path 1,3,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,3,6, -value 802

# LFB 8,1 a LFB 10,1 puerto 803
# Origen
config -classId 1 -instanceId 1 -path 1,4,1,1 -value 8
config -classId 1 -instanceId 1 -path 1,4,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,4,3, -value 803
# Destino
config -classId 1 -instanceId 1 -path 1,4,4,1, -value 10
config -classId 1 -instanceId 1 -path 1,4,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,4,6, -value 803

# LFB 10,1 a LFB 12,1 puerto 804
# Origen
config -classId 1 -instanceId 1 -path 1,5,1,1 -value 10
config -classId 1 -instanceId 1 -path 1,5,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,5,3, -value 804
# Destino
config -classId 1 -instanceId 1 -path 1,5,4,1, -value 12
config -classId 1 -instanceId 1 -path 1,5,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,5,6, -value 804

# LFB 12,1 a LFB 6,1 puerto 805
# Origen
config -classId 1 -instanceId 1 -path 1,6,1,1 -value 12
config -classId 1 -instanceId 1 -path 1,6,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,6,3, -value 805
# Destino
config -classId 1 -instanceId 1 -path 1,6,4,1, -value 6
config -classId 1 -instanceId 1 -path 1,6,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,6,6, -value 805

# LFB 6,1 a LFB 16,1 puerto 806
# Origen
config -classId 1 -instanceId 1 -path 1,7,1,1 -value 6
config -classId 1 -instanceId 1 -path 1,7,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,7,3, -value 806
# Destino
config -classId 1 -instanceId 1 -path 1,7,4,1, -value 16
config -classId 1 -instanceId 1 -path 1,7,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,7,6, -value 806

# LFB 16,1 a LFB 7,1 puerto 807
# Origen
config -classId 1 -instanceId 1 -path 1,8,1,1 -value 16
config -classId 1 -instanceId 1 -path 1,8,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,8,3, -value 807
# Destino
config -classId 1 -instanceId 1 -path 1,8,4,1, -value 7
config -classId 1 -instanceId 1 -path 1,8,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,8,6, -value 807

# LFB 7,1 a LFB 3,1 puerto 808
# Origen
config -classId 1 -instanceId 1 -path 1,9,1,1 -value 7
config -classId 1 -instanceId 1 -path 1,9,1,2, -value 1
config -classId 1 -instanceId 1 -path 1,9,3, -value 808
# Destino
config -classId 1 -instanceId 1 -path 1,9,4,1, -value 3
config -classId 1 -instanceId 1 -path 1,9,4,2, -value 1
config -classId 1 -instanceId 1 -path 1,9,6, -value 808
```

Figura 104. Script de configuración de la topología IP v4 Reenvío

En la consola CE se utiliza la instrucción config –file "forces_script_IPv4Reenvío.txt" para enviar todas las instrucciones necesarias con la información que crean la topología y llenar el árbol LFB de la consola CE

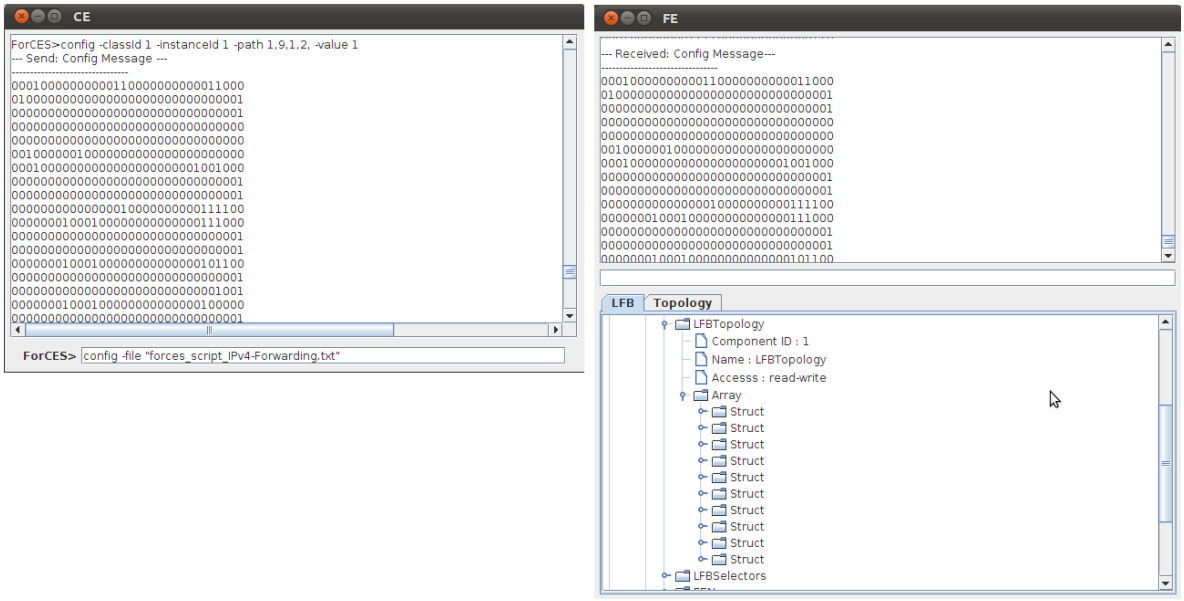


Figura 105. Configuración de la funcionalidad IPv4-Reenvío

Luego de que se verifica la información en el árbol de carpetas LFBs dentro de las 9 estructuras necesarias para esta función, se procede en la pestaña *topology* a dibujar la conexión de los LFBs involucrados.

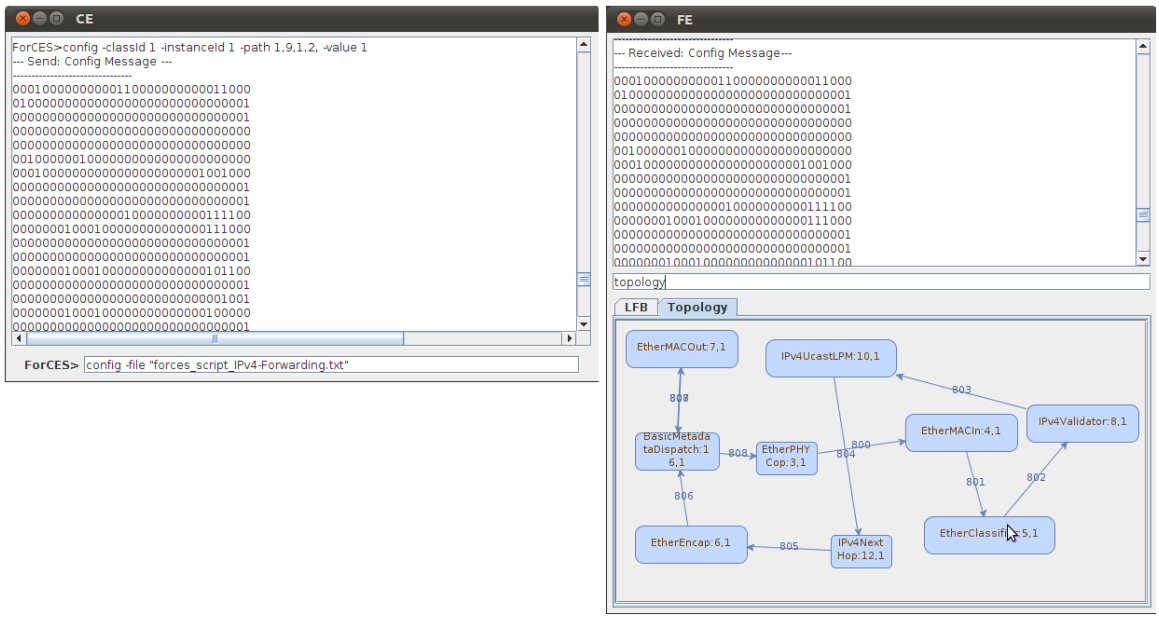


Figura 106. Visualización de la topología IPv4 Reenvío

La siguiente figura muestra el dibujo de la función IP v4 Reenvío tomada del **draft-ietf-forces-lfb-lib-05** donde se puede apreciar como se crearon las distintas estructuras junto con su colección de atributos tales como el InstanceID, ClassID los cuales son los números separados por comas (,) que se ven dentro de cada recuadro LFB y el valor numérico de los puertos que se puede ver en la conexión que se realiza entre dos recuadros LFBs.

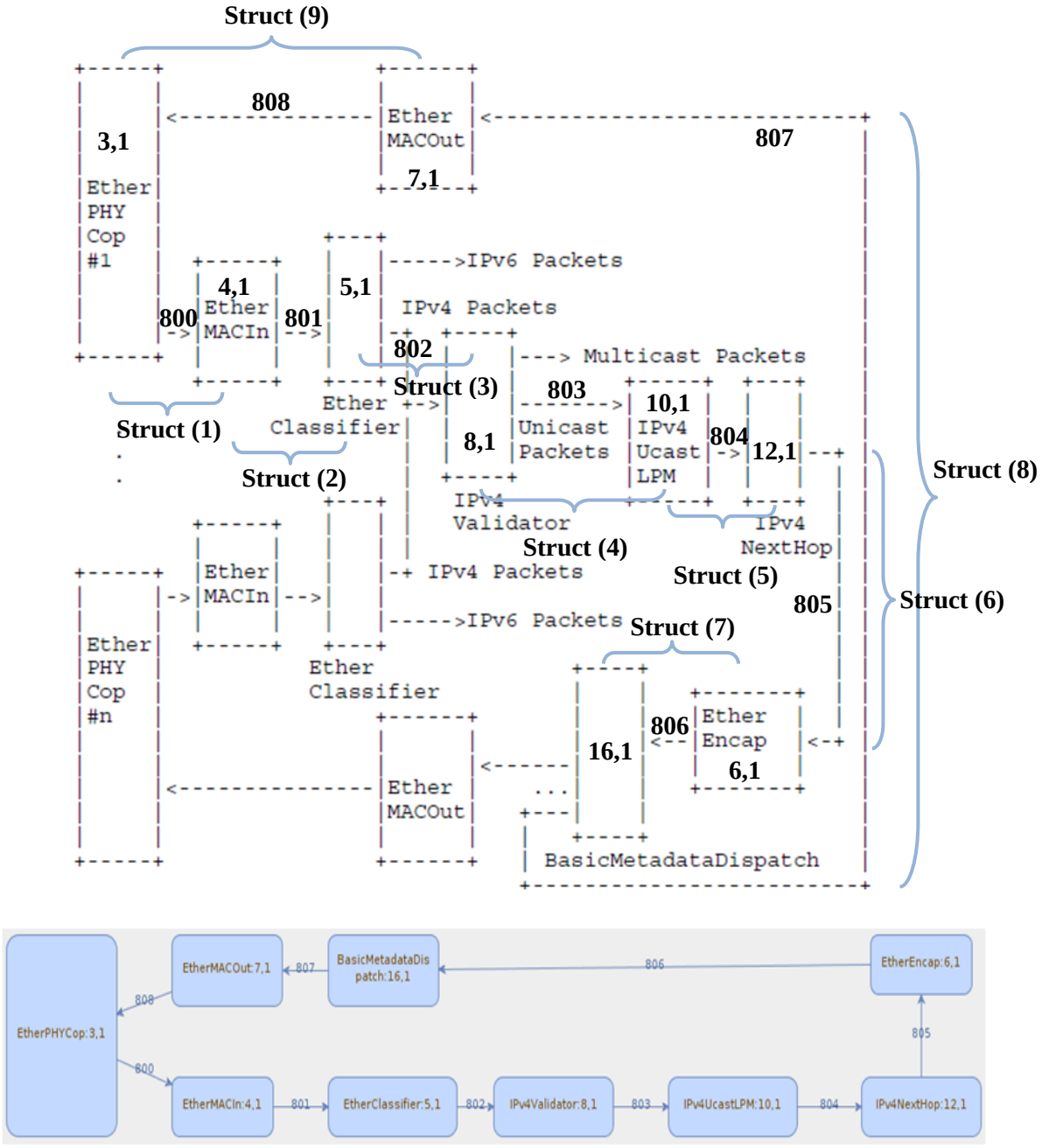


Figura 107. Diseño y Resultado de la topología IPv4 Reenvío

8. CONCLUSIONES

- Con el desarrollo de este proyecto, se podrá utilizar el protocolo ForCES bajo las bases teóricas de la arquitectura ForCEs, aprovechando las ventajas de una arquitectura abierta, flexible y reprogramable para que se realicen pruebas de simulación sobre la interfaz de usuario Java y mejoras al código abierto del programa Java que se entrega.

- Las personas que directamente se verán beneficiadas son aquellas que pertenecen al grupo de desarrollo de ForCEs y a los estudiantes e investigadores en el ámbito académico mundial, que quieran seguir trabajando y mejorando las funcionalidades de la interfaz y el protocolo, ya que les brinda un buen punto de partida para sus futuros desarrollos en este tema.

- Con los resultados obtenidos, se puede brindar apoyo a futuras investigaciones, también brindará a los distintos grupos de investigación una herramienta muy valiosa que les permita trabajar observando comportamientos de diferentes tipos de topologías y a través de estas construir diferentes funcionalidades de los elementos de red contenidos en un enrutador con arquitectura reprogramable ForCES.

- Esta implementación permite proponer nuevas Topologías de prueba de los LFBs en los CEs que estructuren funciones de un enrutador típico, aunque aún se encuentra en proceso de investigación y no aseguran un manejo claro sobre los atributos disponibles en los FEs.

- Se lograron las pruebas siguiendo el esquema de evaluación propuesto por el draft-ietf-forces-interoperability-04, a través de comandos por escenarios, y luego la captura de la trama de datos con el analizador de protocolos wireshark.

- Aunque internamente los equipos de red tienen planos de datos y control diferenciados, esta separación lógica no permite que cada plano evolucione de forma totalmente independientemente, puesto que el fabricante es el único que puede innovar en cualquiera de los dos campos. La implementación de este protocolo es una muestra del avance que se tiene hasta el momento para lograr esta separación.

- La arquitectura ForCES y su protocolo, se encuentran en la actualidad en un proceso de desarrollo, de evolución, por lo que se presentan de manera continua actualizaciones en sus especificaciones. Esto se evidenció claramente al comienzo de este proyecto pues se experimentaron muchos de estos cambios en las distintas versiones de la documentación emitida por el IETF, y aunque ya hay un RFC donde se dan algunas pautas para la implementación del protocolo ForCES, todavía no está plenamente definida y es suceptible a cambios. Teniendo en cuenta esta circunstancia el software se diseño de manera tal que su estructura permita de manera muy clara y sencilla la incorporación de las diferentes mejoras y/o actualizaciones del protocolo.

- El definir y utilizar una metodología de desarrollo por fases, fue fundamental en el cumplimiento de los objetivos del proyecto, pues permitió obtener una implementación que

correspondiera con la especificación definida en el RFC, y su respectiva verificación mediante la fase de pruebas.

- Realizar un protocolo que se ajuste a la política de flexibilidad y reprogramación que exige ForCES dentro de su especificación implica un alto grado de complejidad, que involucra un esfuerzo considerable en la parte del desarrollo del software, ya que se necesita interpretar mucha información de los RFCs y Draft que no son muy claros al momento de programar y mantener el estándar de ForCES.

- Al utilizar el protocolo ForCES, este solo se encarga de llevar la información de un extremo a otro bajo las condiciones expuestas en la especificación. De manera que la implementación de una estructura de hardware del Modelo FE podría ser bastante compleja cuando se desee hacer una implementación sobre dispositivos FPGAs.

9. RECOMENDACIONES

- Los desarrollos posteriores y las mejoras a este proyecto permitirán realizar pruebas fiables hasta llegar a emulaciones de los elementos de red CE y FE. Por ahora el protocolo ForCES que se entrega, asegura la conexión entre estos dos elementos, dejando la inquietud en estudiantes de pregrado y maestría de la Universidad para que continúen con la construcción de un simulador y emulador robusto, o que incluso, se pueda integrar a futuros desarrollos de hardware de los elementos FE en sistemas FPGAs, conectados a través del bus PCI de una computadora de alto desempeño de procesamiento, y así realizar pruebas de manejo de datos entre maquinas PCs actuando como elementos CEs.

- También se puede crear un nodo de red en la universidad conformado por equipos PCs actuando de elementos CE, con sus respectivos puertos PCI asociados a tarjetas FPGAs que actuarían como elementos FEs. Y realizar pruebas de datos sobre tecnologías de comunicaciones nuevas sin tener que estar cambiando los equipos del nodo, sino simplemente actualizar su software y firmware para que el nodo quede operativo para esta tecnología.

- Si se realizan las mejoras necesarias para lograr simular, emular o incluso implementar en hardware y software el nodo de red en su totalidad, y lograr que los resultados de las pruebas resulten ser bastante eficientes tanto en la flexibilidad como en el rendimiento, permitirá que muchos fabricantes giren su mirada a desarrollar sistemas como estos dispositivos de red.

- Los enrutadores se encuentran separados entre un plano de datos y uno de control, lo cual es un principio de diseño conocido y plenamente aceptado, en la práctica este principio aún no ha sido aplicado hasta sus últimas consecuencias. Típicamente los equipos de red son sistemas integrados verticalmente, esto es, el fabricante proporciona tanto el hardware de comunicaciones como el software de control.

10. BIBLIOGRAFÍA

[1] IETF ForCES Working Group. Reenvío and Control Element Separation (ForCES). [Online]. https://datatracker.ietf.org/wg/ForCES/

[2] A, Doria; J, Hadi Salim; R, Haas; H, Khosravi; W, Wang; L, Dong; R, Gopal; J, Halpern, "Reenvío and Control Element Separation (ForCES) Protocol Specification," RFC 5810, Marzo, 2010.

[3] J, Hadi Salim; K, Ogawa, "SCTP-Based Transport Mapping Layer (TML) for the Reenvío and Control Element Separation (ForCES) Protocol," RFC 5811, Marzo, 2010.

[4] J, Hadi Salim; J, Halpern;, "Reenvío and Control Element Separation (ForCES) Reenvío Element Model," RFC 5812, Marzo, 2010.

[5] W, Wang; E, Haleplidis; K, Ogawa; C, Li; J, Halpern, "ForCES Logical Function Block (LFB) Library," **draft-ietf-forces-lfb-lib-05**, Julio 10, 2011.

[6] E, Haleplidis; K, Ogawa; X, Wang; C, Li, "ForCES Interoperability Draft," **draft-ietf-forces-interoperability-04**, Septiembre 7, 2009.

[7]Chrysoulas, C; Haleplidis, E; Kostopulos, S; Denazis, S; Koufopavlou, O, **"A Distributed Enrutador's Modelling and Implementation"**, to be presented in COMMUNICATION SYSTEMS, NETWORKS AND DIGITAL SIGNAL PROCESSING, Fifth International Symposium, 19-21 July, 2006, Patras Greece

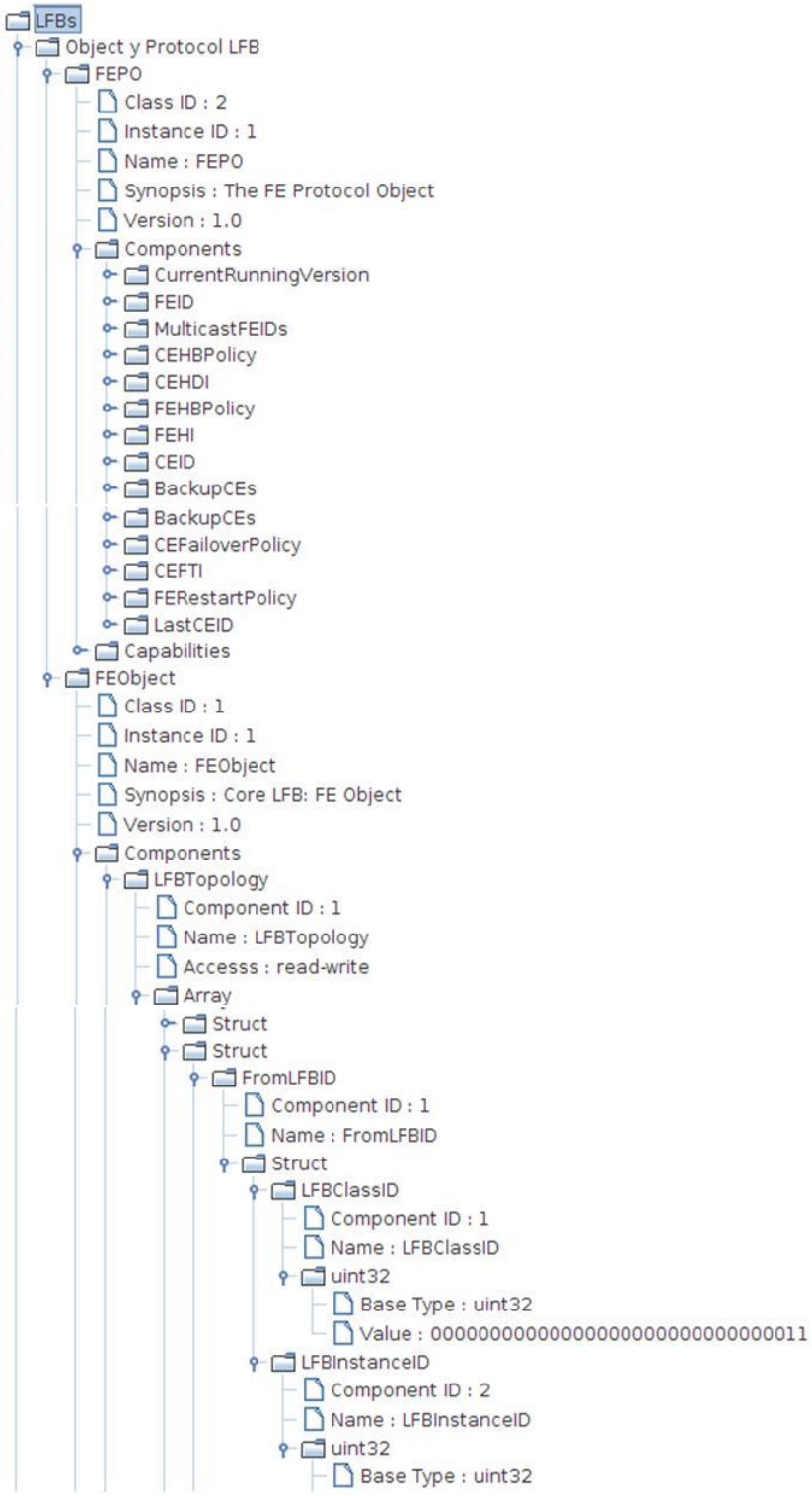
[8] Weiming, W; Dong, L; Zhuge, B; **"ForTer - An Open Programmable Enrutador Based on Reenvío and Control Element Separation",** Journal of Computer Science and Technology, Vol. 23, No. 5. (2008), pp. 769-779.

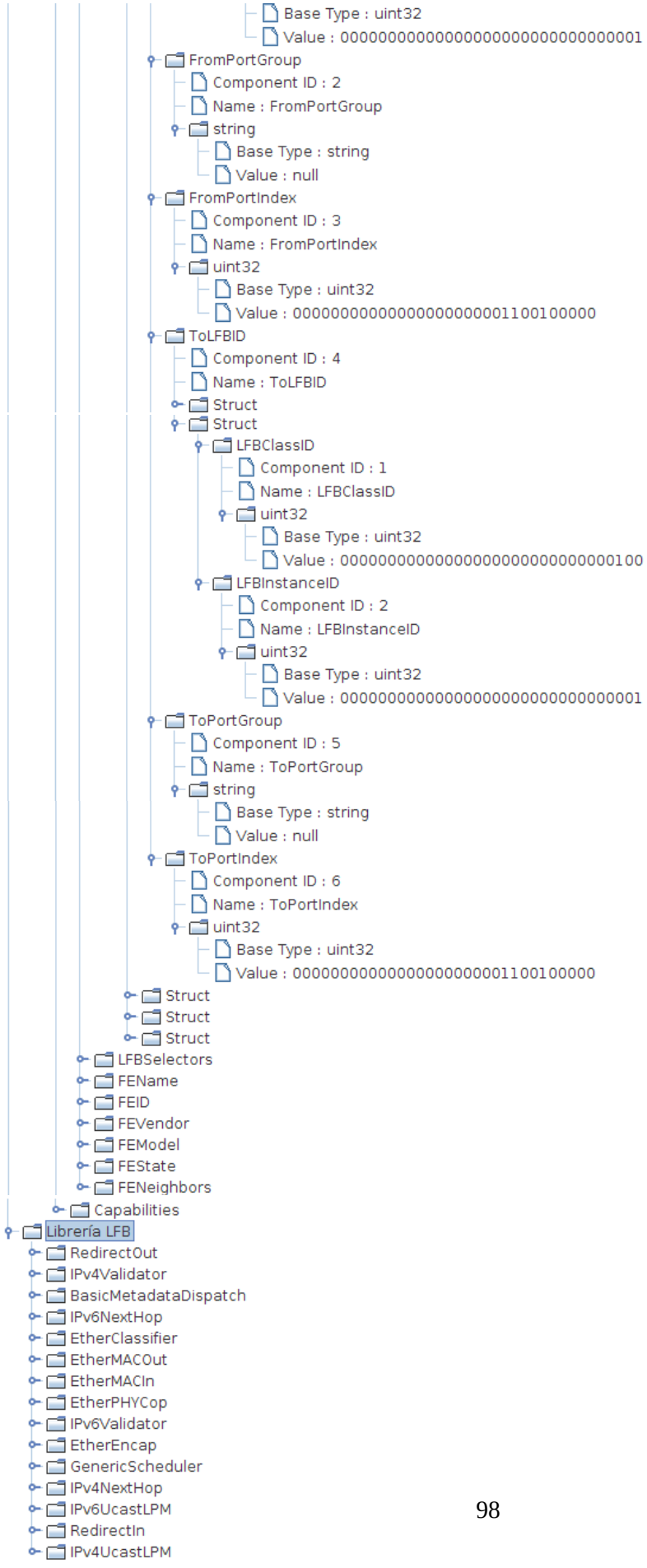
[9] Weiming, W; Gao, M; Dong, L and Xi, J, **"An Improved Algorithm for ForCES LFBs Topology in J2EE",** International Journal of Web Services Practices, Vol. 3, No.1-2 (2008), pp. 89-93.

[10] Medhi, D; Ramasamy, K, **"Network Routing: Algorithms, Protocols, and Architectures"**, editorial Elsevier, San Francisco, 2007.

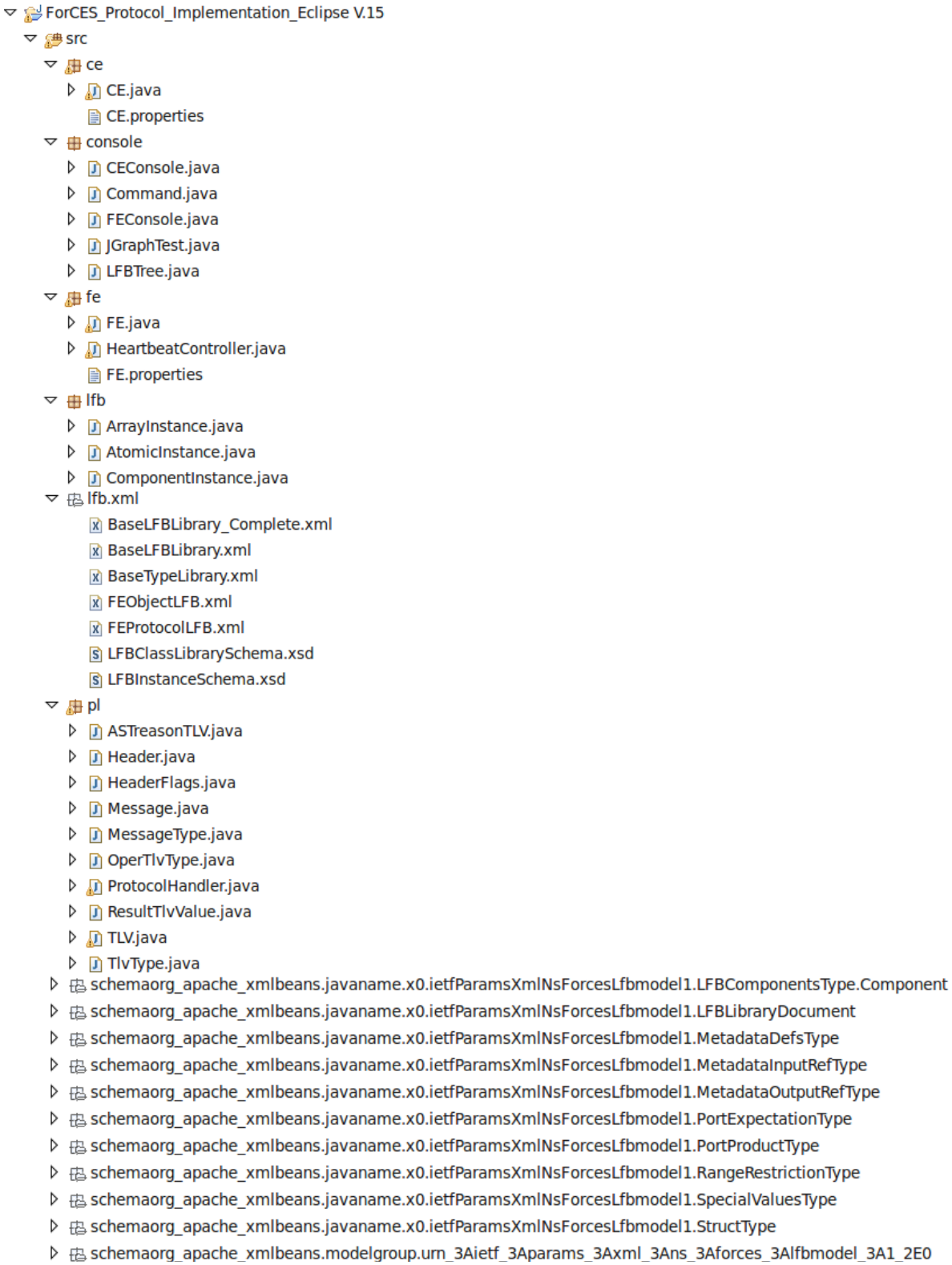
11. INDICE

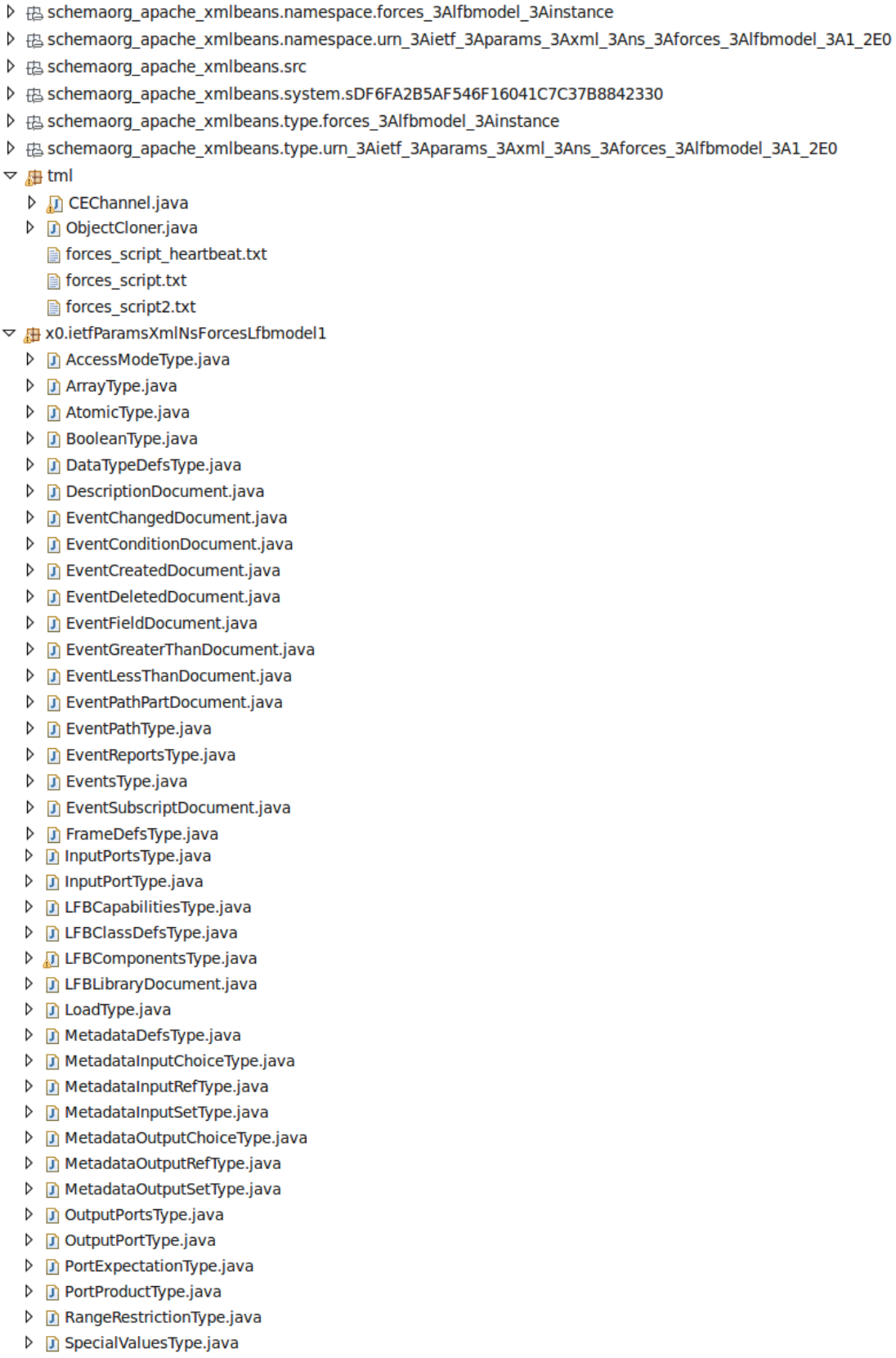
ANEXO A.

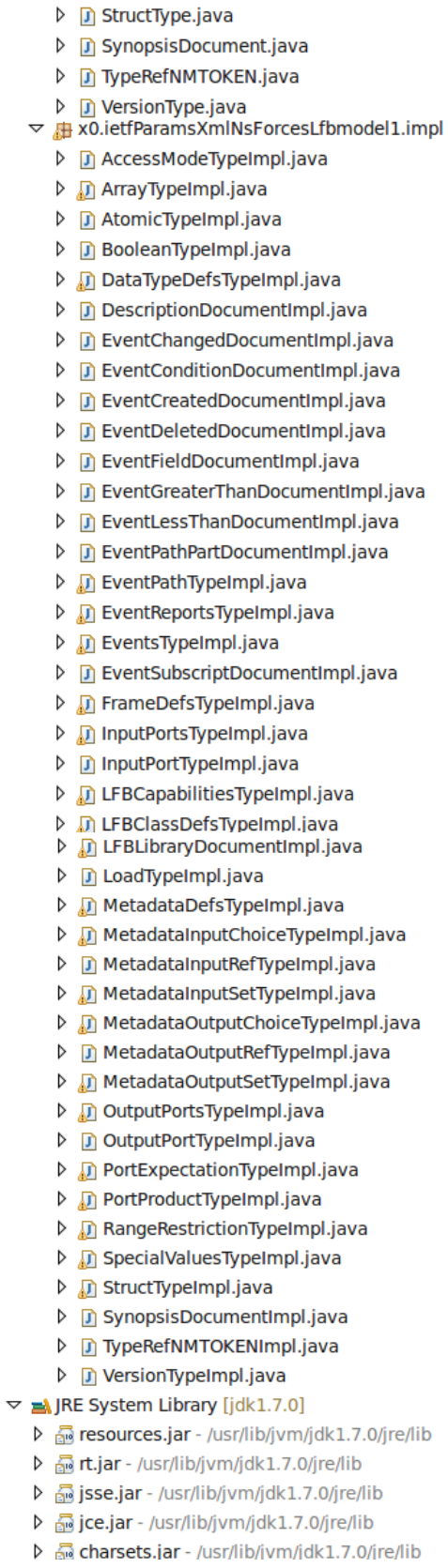

LFBs
Object y Protocol LFB
FEPO
Class ID : 2
Instance ID : 1
Name : FEPO
Synopsis : The FE Protocol Object
Version : 1.0
Components
CurrentRunningVersion
FEID
MulticastFEIDs
CEHBPolicy
CEHDI
FEHBPolicy
FEHI
CEID
BackupCEs
BackupCEs
CEFailoverPolicy
CEFTI
FERestartPolicy
LastCEID
Capabilities
FEObject
Class ID : 1
Instance ID : 1
Name : FEObject
Synopsis : Core LFB: FE Object
Version : 1.0
Components
LFBTopology
Component ID : 1
Name : LFBTopology
Accesss : read-write
Array
Struct
Struct
FromLFBID
Component ID : 1
Name : FromLFBID
Struct
LFBClassID
Component ID : 1
Name : LFBClassID
uint32
Base Type : uint32
Value : 00000000000000000000000000000011
LFBInstanceID
Component ID : 2
Name : LFBInstanceID
uint32
Base Type : uint32


Base Type : uint32
Value : 00000000000000000000000000000001
FromPortGroup
Component ID : 2
Name : FromPortGroup
string
Base Type : string
Value : null
FromPortIndex
Component ID : 3
Name : FromPortIndex
uint32
Base Type : uint32
Value : 00000000000000000000001100100000
ToLFBID
Component ID : 4
Name : ToLFBID
Struct
Struct
LFBClassID
Component ID : 1
Name : LFBClassID
uint32
Base Type : uint32
Value : 00000000000000000000000000000100
LFBInstanceID
Component ID : 2
Name : LFBInstanceID
uint32
Base Type : uint32
Value : 00000000000000000000000000000001
ToPortGroup
Component ID : 5
Name : ToPortGroup
string
Base Type : string
Value : null
ToPortIndex
Component ID : 6
Name : ToPortIndex
uint32
Base Type : uint32
Value : 00000000000000000000001100100000
Struct
Struct
Struct
LFBSelectors
FEName
FEID
FEVendor
FEModel
FEState
FENeighbors
Capabilities
Librería LFB
RedirectOut
IPv4Validator
BasicMetadataDispatch
IPv6NextHop
EtherClassifier
EtherMACOut
EtherMACIn
EtherPHYCop
IPv6Validator
EtherEncap
GenericScheduler
IPv4NextHop
IPv6UcastLPM
RedirectIn
IPv4UcastLPM

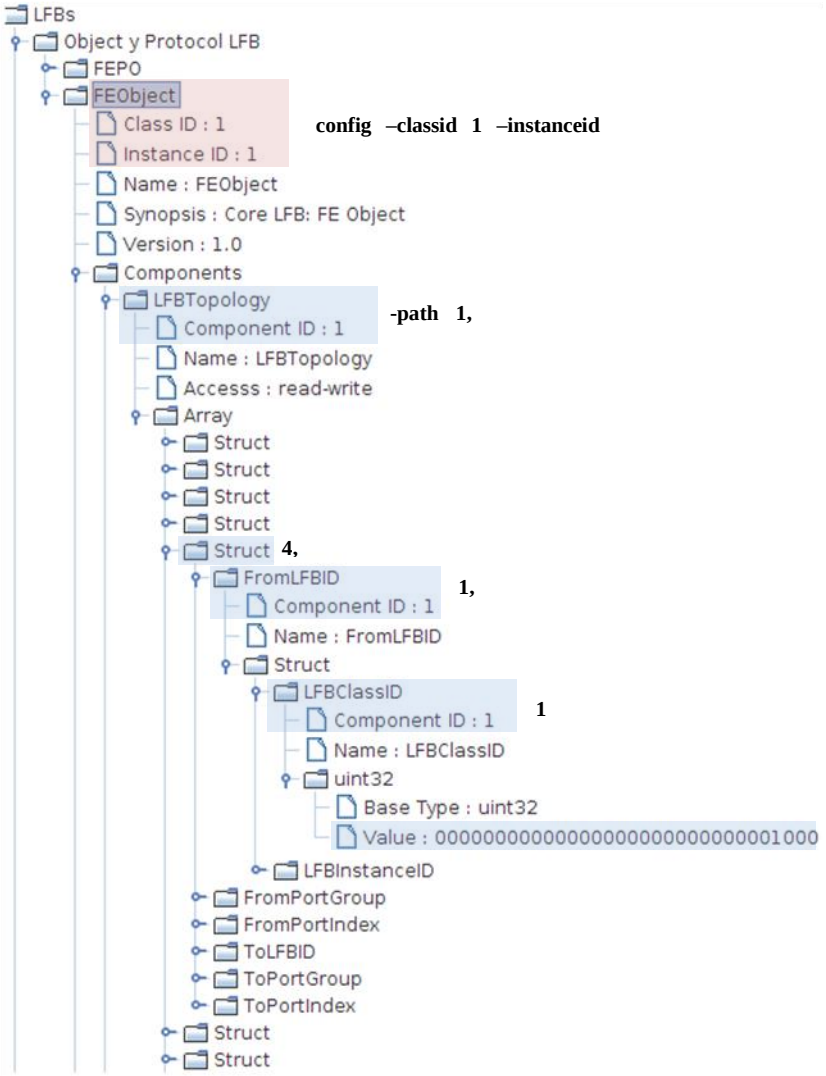
ANEXO B.


ForCES_Protocol_Implementation_Eclipse V.15
src
ce
CE.java
CE.properties
console
CEConsole.java
Command.java
FEConsole.java
JGraphTest.java
LFBTree.java
fe
FE.java
HeartbeatController.java
FE.properties
lfb
ArrayInstance.java
AtomicInstance.java
ComponentInstance.java
lfb.xml
BaseLFBLibrary_Complete.xml
BaseLFBLibrary.xml
BaseTypeLibrary.xml
FEObjectLFB.xml
FEProtocolLFB.xml
LFBClassLibrarySchema.xsd
LFBInstanceSchema.xsd
pl
ASTreasonTLV.java
Header.java
HeaderFlags.java
Message.java
MessageType.java
OperTlvType.java
ProtocolHandler.java
ResultTlvValue.java
TLV.java
TlvType.java
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.LFBComponentsType.Component
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.LFBLibraryDocument
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.MetadataDefsType
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.MetadataInputRefType
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.MetadataOutputRefType
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.PortExpectationType
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.PortProductType
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.RangeRestrictionType
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.SpecialValuesType
schemaorg_apache_xmlbeans.javaname.x0.ietfParamsXmlNsForcesLfbmodel1.StructType
schemaorg_apache_xmlbeans.modelgroup.urn_3Aietf_3Aparams_3Axml_3Ans_3Aforces_3Alfbmodel_3A1_2E0


schemaorg_apache_xmlbeans.namespace.forces_3Alfbmodel_3Ainstance
schemaorg_apache_xmlbeans.namespace.urn_3Aietf_3Aparams_3Axml_3Ans_3Aforces_3Alfbmodel_3A1_2E0
schemaorg_apache_xmlbeans.src
schemaorg_apache_xmlbeans.system.sDF6FA2B5AF546F16041C7C37B8842330
schemaorg_apache_xmlbeans.type.forces_3Alfbmodel_3Ainstance
schemaorg_apache_xmlbeans.type.urn_3Aietf_3Aparams_3Axml_3Ans_3Aforces_3Alfbmodel_3A1_2E0
tml
CEChannel.java
ObjectCloner.java
forces_script_heartbeat.txt
forces_script.txt
forces_script2.txt
x0.ietfParamsXmlNsForcesLfbmodel1
AccessModeType.java
ArrayType.java
AtomicType.java
BooleanType.java
DataTypeDefsType.java
DescriptionDocument.java
EventChangedDocument.java
EventConditionDocument.java
EventCreatedDocument.java
EventDeletedDocument.java
EventFieldDocument.java
EventGreaterThanDocument.java
EventLessThanDocument.java
EventPathPartDocument.java
EventPathType.java
EventReportsType.java
EventsType.java
EventSubscriptDocument.java
FrameDefsType.java
InputPortsType.java
InputPortType.java
LFBCapabilitiesType.java
LFBClassDefsType.java
LFBComponentsType.java
LFBLibraryDocument.java
LoadType.java
MetadataDefsType.java
MetadataInputChoiceType.java
MetadataInputRefType.java
MetadataInputSetType.java
MetadataOutputChoiceType.java
MetadataOutputRefType.java
MetadataOutputSetType.java
OutputPortsType.java
OutputPortType.java
PortExpectationType.java
PortProductType.java
RangeRestrictionType.java
SpecialValuesType.java


StructType.java
SynopsisDocument.java
TypeRefNMTOKEN.java
VersionType.java
x0.ietfParamsXmlNsForcesLfbmodel1.impl
AccessModeTypeImpl.java
ArrayTypeImpl.java
AtomicTypeImpl.java
BooleanTypeImpl.java
DataTypeDefsTypeImpl.java
DescriptionDocumentImpl.java
EventChangedDocumentImpl.java
EventConditionDocumentImpl.java
EventCreatedDocumentImpl.java
EventDeletedDocumentImpl.java
EventFieldDocumentImpl.java
EventGreaterThanDocumentImpl.java
EventLessThanDocumentImpl.java
EventPathPartDocumentImpl.java
EventPathTypeImpl.java
EventReportsTypeImpl.java
EventsTypeImpl.java
EventSubscriptDocumentImpl.java
FrameDefsTypeImpl.java
InputPortsTypeImpl.java
InputPortTypeImpl.java
LFBCapabilitiesTypeImpl.java
LFBClassDefsTypeImpl.java
LFBLibraryDocumentImpl.java
LoadTypeImpl.java
MetadataDefsTypeImpl.java
MetadataInputChoiceTypeImpl.java
MetadataInputRefTypeImpl.java
MetadataInputSetTypeImpl.java
MetadataOutputChoiceTypeImpl.java
MetadataOutputRefTypeImpl.java
MetadataOutputSetTypeImpl.java
OutputPortsTypeImpl.java
OutputPortTypeImpl.java
PortExpectationTypeImpl.java
PortProductTypeImpl.java
RangeRestrictionTypeImpl.java
SpecialValuesTypeImpl.java
StructTypeImpl.java
SynopsisDocumentImpl.java
TypeRefNMTOKENImpl.java
VersionTypeImpl.java
JRE System Library [jdk1.7.0]
resources.jar - /usr/lib/jvm/jdk1.7.0/jre/lib
rt.jar - /usr/lib/jvm/jdk1.7.0/jre/lib
jsse.jar - /usr/lib/jvm/jdk1.7.0/jre/lib
jce.jar - /usr/lib/jvm/jdk1.7.0/jre/lib
charsets.jar - /usr/lib/jvm/jdk1.7.0/jre/lib

ANEXO C.



config –classid 1 –instanceid 1 –path 1,4,1,1 –value