034115

PHOSPHORUS

Lambda User Controlled Infrastructure for European Research

Integrated Project

Strategic objective:
Research Networking Testbeds

Deliverable reference number: D.4.3.2

ForCES Token Based Switch

Due date of deliverable: 30-09-2008
Actual submission date: 30-09-2008
Document code: <Phosphorus-WP4-D.4.3.2>

Start date of project:
October 1, 2006

Duration:
30 Months

Organisation name of lead contractor for this deliverable:
University of Amsterdam

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract

This document describes the software architecture design and implementation of the ForCES Token Based Switch, which is a gateway router in a Token Based Networking. The Token Based Networking allows binding of applications' traffic from end-nodes (e.g., IP packets in a campus network) to end-to-end lightpaths (e.g., GMPLS circuits). In other words, the Token Based Networking architecture establishes authorised paths over multiple network domains on behalf of the end-nodes applications. The Token Based Switch is a programmable hardware and software system that processes IP packets at high speeds (multi-gigabits/sec). We designed and built our gateway router that operates at IP layer (TBS-IP) by using specialised hardware (network processors) programmable by higher-level software through a standard protocol: ForCES. ForCES is a new protocol designed by IETF, nearing the point of being RFC, which separates the Forwarding and the Control planes in a traffic processing system. ForCES provides a scalable way to configure and manage the Forwarding Plane using a standard protocol.

The current document provides complete information on the TBS-IP software and hardware modules, their expected functionalities and core structure needed to be used in the Phosphorus test-bed when enhancing a traditional GMPLS end-node with capabilities of sharing the lightpaths to multiple end-hosts and their applications in a secure manner.

The document introduces a new approach in regard to network security, the *Token* concept in the Token Based Networking architecture, and describes how the ForCES protocol was used to implement a flexible, scalable and configurable Token Based Switch (TBS-IP) for secure networking over multiple domains. For instance, the document discusses the modelling of the required hardware functionalities into ForCES Logical Function Blocks (LFBs) according to the ForCES protocol.

Table of Contents

List of Figures

List of Tables

0 Executive Summary

The document introduces the Token Based Networking architecture, and describes how the ForCES protocol was used to implement a flexible, scalable and configurable Token Based Switch (TBS-IP) for secure networking over multiple domains.

Section 1 introduces a new practical approach in regard to secure network access control: the Token Based Networking concept.

Section 2 briefly introduces the ForCES protocol as well as it describes the architecture of ForCEG, a ForCES Gateway component enhanced with WebService capabilities in a TBS-IP system.

Sections 3 describes the architecture of the ForCES Token Based Switch that implements the Token Based Networking concepts and uses the ForCES protocol to provide a flexible, scalable, and configurable Switch for establishing secure data paths between applications on end-hosts in a IP campus network over a multi-domain circuits based network.

Finally, section 4 discusses the implementation of the ForCES Token Based Switch (TBS-IP) in both hardware, and software and the modelling of the hardware into ForCES Logical Function Blocks (LFBs).

A set of appendixes provide further details about XML Token and Authorization Ticket formats used in the authorisation process and the XML schema of the ForCES LFBs that were implemented to program the low-level hardware processing units (network processors).

1 Token Based Networking

Token Based Networking (TBN) is a new concept for a network that handles packets according to a specific tag, built-in the packet, called token. TBN uses tokens built from small encrypted pieces of data (e.g., a few bytes). By its mean, a token allows for applying different access/enforcement/filtering rules/criteria when processing packets that carry the tokens.

Tokens are a simpler way to authorise resource usage than certificates due to their concept: a policy enforcement point authorise the resource usage (e.g., a packet takes a certain path) only if the built-in token matches a cryptographic result applied to the packet using a set of keys provided beforehand by an authority at the moment of the path reservation. Tokens can bind to different semantics (in particular, it can be associated with the group of users and share its use when accessing designed resources), and they decouple the time of authorisation decision from the time of use (e.g., enable the use of resource for 10~hours from now or from tomorrow at 5 o'clock). Moreover, Tokens are a simple way of controlling access to authorised/reserved network resources that separates reservation/policy decision and access stages.

On the one hand, in the context of access control for high speed networks, one may see the tokens as a “glue” between upper layer authorisation infrastructure, which uses policies to describe the resource usage, and the lower layer enforcement points of the data plane.

On the other hand, in the context of applications interconnected over the networks, we see the tokens as a glue between the application’s traffic (sockets) and the network resources (lightpaths). In other words, a token binds applications’ IP packets and circuits, as illustrated in Figure 1.1.

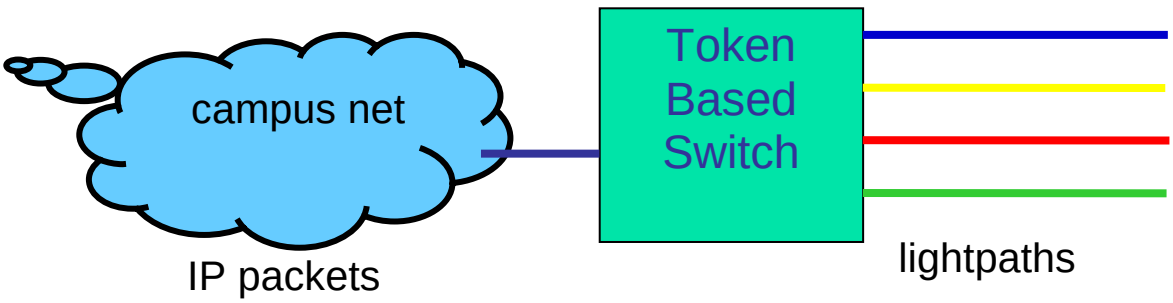

Figure 1.1 Connecting applications through the optical paths.

1.1 TBN architecture

Figure 1.2 shows the architecture used for setting up a lightpath using token based networking. In a nutshell, the process is as follows:

1. On behalf of a subset of its users, ISP_A generates a Link Access Request (LAR) message to use a particular link (from the available links of its peering point) for a certain period of time. The details of LAR generation based on user requests are covered in the deliverable D4.1-Chapter3;
2. The LAR is sent to the AAA server of peering point A ;
3. The AAA server fetches a policy from the policy repository that validates the request;
4. Next, a key (TokenKey) is generated and sent to ISP_A and peering point A. This token key will be next used for both for generating tokens at the ISP or user side and verifying tokens at each network link gateway. ISP_A may use the key to create a token for each packet that should use the requested lightpath. The token will be inserted in the packets which are then forwarded to the peering point. The entity responsible for token injection is known as the token builder (TB). On the other side, peering point A uses the same TokenKey as ISP_A to check the incoming token-labelled packets;
5. For each received packet, peering point A computes a local token and checks it against the embedded packet token to check the packet authenticity/validity/authorisation for the requested lightpath. The packet is authenticated when both tokens match. An authenticated packet is then forwarded to its corresponding fibre-optic link (5);
6. All other packets are transmitted on a 'default' link that is connected to the transit network (6).

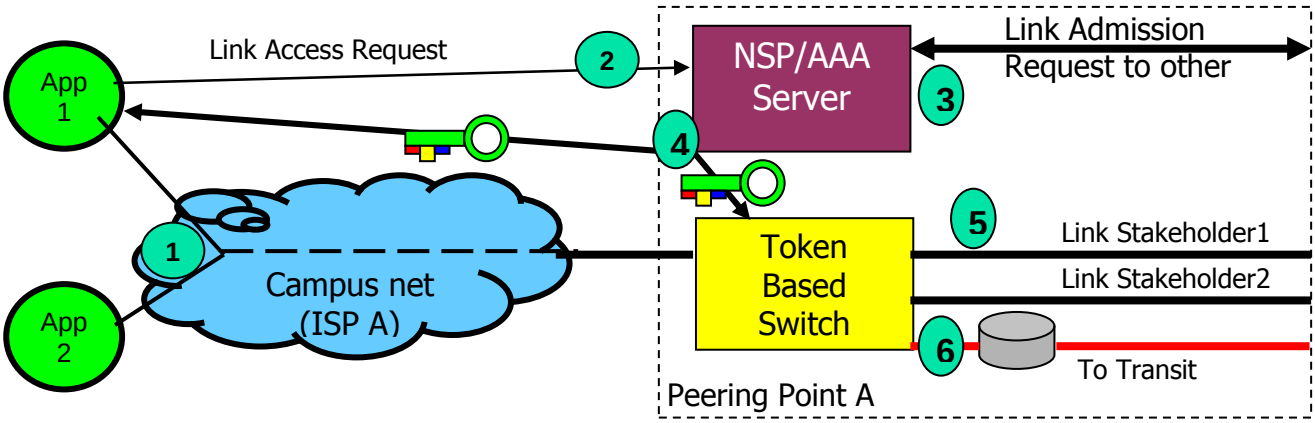

Figure 1.2 TBN architecture.

The entity responsible for token checking and packet switching is known as the token based switch. When a packet arrives at the token switch, we must find the appropriate key to use for computing the token locally. Some fields in the packet tell which key to use. For instance, we may associate a key with an IP

source and destination pair, so all traffic between two machines is handled with the same key. However, other forms of aggregation are possible. For instance, we may handle all traffic between two networks with the same key, or all machines participating in a Grid experiment, etc. In general, we allow the key to be selected by means of an aggregation identifier, embedded in the packet. The aggregation identifier is inserted in the packet together with the token by the ISP to signal the key to use.

1.2 TBN in multi-domains

The TBN architecture offers to the user applications an authenticated access control mechanism to the reserved high-speed links (lightpaths) across multi-domain hybrid networks. For example, in Figure 1.3, we show a scenario with three domains and a setup for several possible lightpaths. The procedure to establish a lightpath consists of two phases that are decoupled in time: (1) a high-level set-up phase (obtaining tokens from an AAA web-service), and (2) a reserved path access including authenticity and authorisation checks (per-packet token checks at network edges within a multi-domain end-to-end connection).

The first phase allows individual users, or group of users (e.g., a research institution), or even user applications, to request privileged end-to-end connection across multi-domain networks by contacting, at service plane, an inter-domain or local ISP network provisioning service. Next, the network provisioning service would check whether the requested end-to-end path is available and return a positive result to the requestor containing the “unique” global reservation identifier (GRI) and autorisation key (Key).

The second phase determines how TBN authenticates network traffic (TCP connections, UDP transmissions, or other protocols) and how it checks the traffic for authorisation on behalf of their applications. The second phase is also responsible for preventing non-authorised use of lightpaths in a multi-domain network. Two network components are involved in the data plane: the token builder (TB) and the token based switch (TBS-IP). Although there are many components involved in the high-level processing (service and control planes) such as NSP/AAA, we designed and implemented the Token Validation Service (TVS) component that validates the TokenKey/GRI requests from higher level (NSP/AAA) and pushes the provisioning requests into the low-level hardware (PEP).

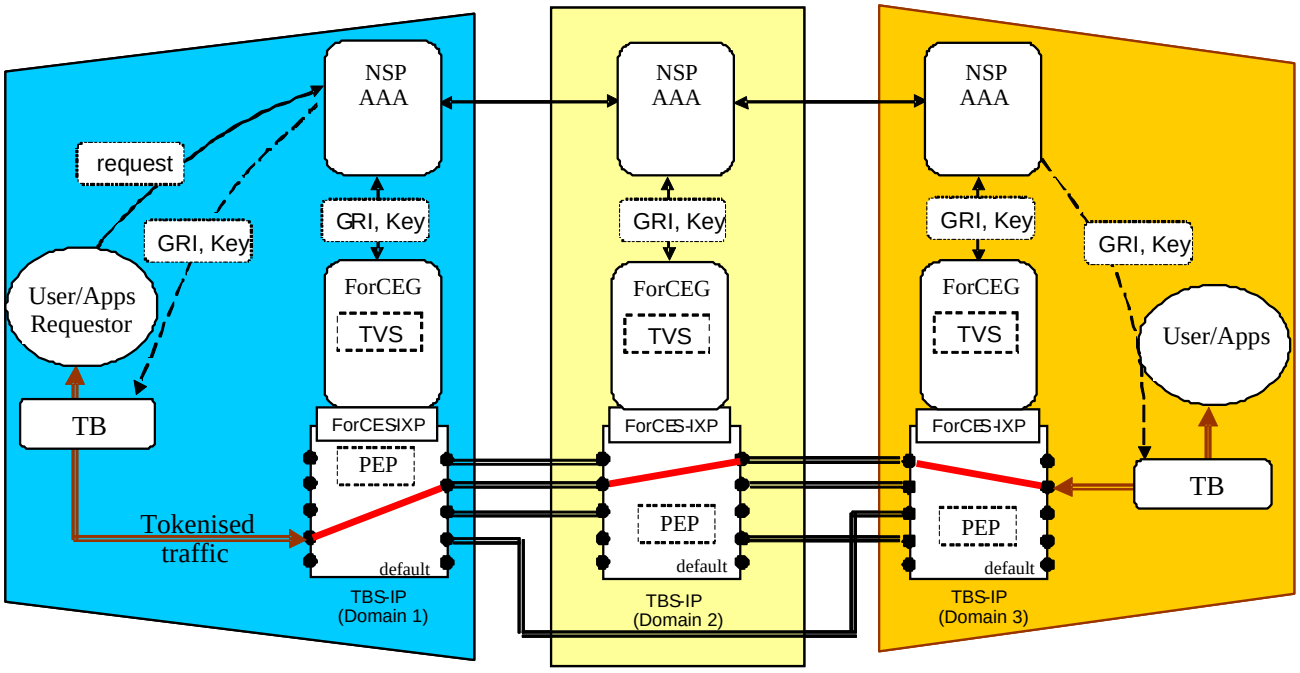

Figure 1.3 TBN in multi-domain networks.

1.3 Token usage in IPv4

Compared to other access control mechanisms (such as certificates), a token is a general type of trusted and cryptographically protected proof of authorisation with flexible usage policies. A token created for an IP packet is essentially the result of applying an HMAC algorithm over a number of packet fields as illustrated in Figure 1.4 and explained below. An HMAC algorithm is a key-dependentway to create a one-way hash (see RFC2401). Currently, in our implementation we opted for a strong proof of authorisation by means of HMAC-SHA1.

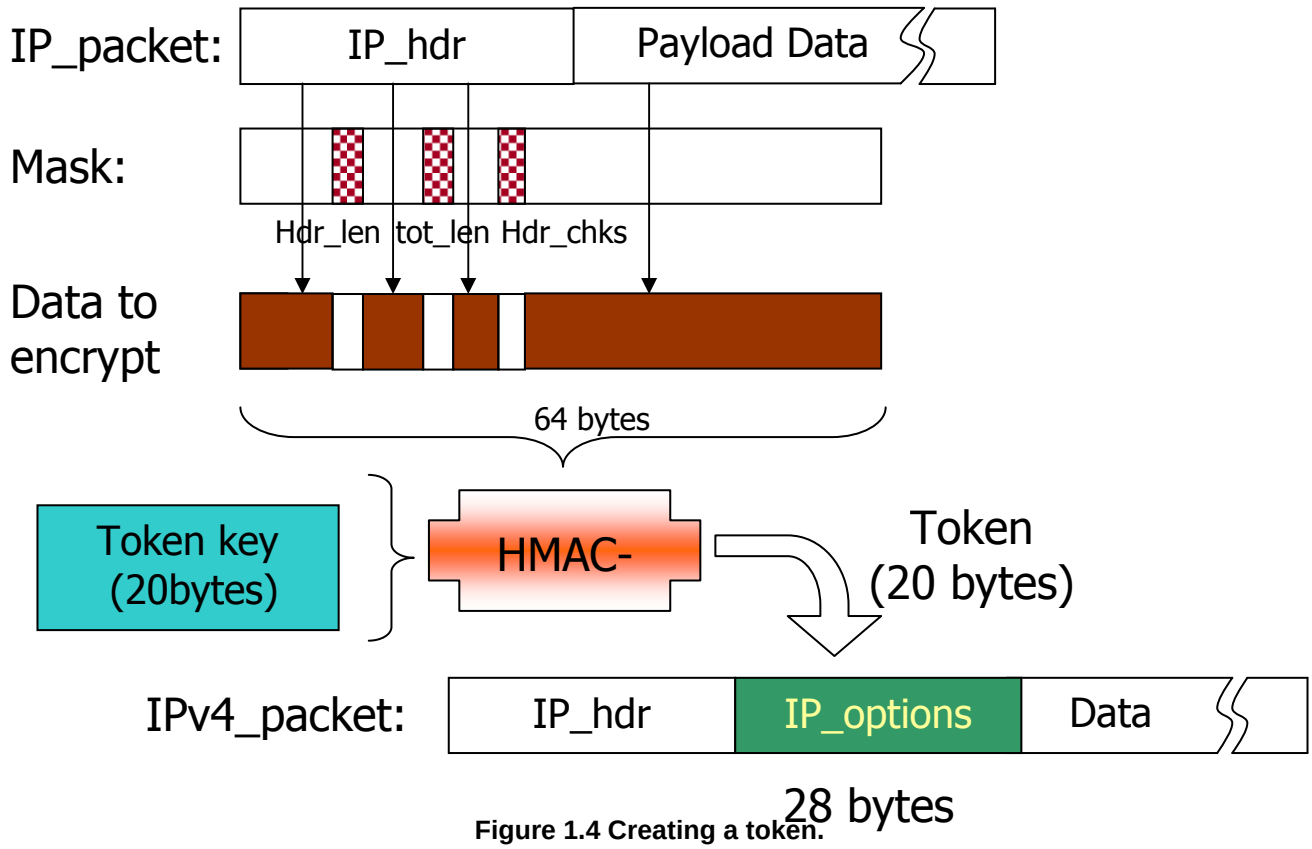

Figure 1.4 Creating a token.

To evaluate the TBS principles, we developed a prototype that stores the token in each packet's IP option field (as defined in RFC 791). An IP option can be of variable length and its field will be ignored by normal routers. Although some routers have a slower processing path for the IP option packets than simple IP packets (because higher level headers will be at different positions in the packet), we noticed that our TBS system works well in high speed, important and pricey sites (e.g., ISPs, grid nodes interconnection points) where all systems and also routers are updated to the state-of-the-art hardware.

Figure 1.4 illustrates the process of token creation and insertion in the IP option field. In our prototype, the HMAC-SHA1 algorithm generates the unique token (20 bytes) that will be injected into the packet's IP option field. As an IP option field has a header of two bytes, and network hardware systems commonly work most efficiently on chunks of four or eight bytes, we reserve 24 bytes for the IP option field. In order to ensure the token uniqueness for packets, we need to include fields that are different in every packet. Therefore, a part of the packet data will be included together with the packet header in the HMAC calculation. We mention that some of the first 64 bytes of an Ethernet frame are masked in order to make the token independent of the IP header fields which change when a token is inserted (e.g., header length, total length, IP checksum) or when the packet crosses intermediate routers (e.g., TTL). The mask also provides flexibility for packet authentication, so that one could use the (sub)network instead of end node addresses, or limit the token coverage to the destination IP address only (e.g., by selectively masking the IP source address).

2 ForCES Framework

The emergence of off-the-shelf network processor devices that implement the fast path or forwarding plane in network devices such as routers, along with the appearance of a new generation of third party signalling, routing, and other router control plane software, has created the need for standard mechanisms to allow these components to be combined into functional wholes. ForCES [1] aims to define a framework and associated mechanisms for standardizing the exchange of information between the logically separate functionality of the control plane, including entities such as routing protocols, admission control, and signalling, and the forwarding plane, where per-packet activities such as packet forwarding, queuing, and header editing occur. By defining a set of standard mechanisms for control and forwarding separation, ForCES will enable rapid innovation in both the control and forwarding planes.

A standard separation mechanism allows the control and forwarding planes to innovate in parallel while maintaining interoperability. The IETF ForCES protocol design team has been working on a merger of three protocol proposals. A first version of the common protocol draft was released in June 2004.

ForCES aims to define a framework and associated protocol(s) to standardize information exchange between the control and forwarding plane. Having standard mechanisms allows Control plane Elements (CEs) and Forwarding plane Elements (FEs) to become physically separated standard components. This physical separation accrues several benefits to the ForCES architecture most of all interoparablity between individual vendors. This interoperability translates into increased design choices and flexibility for the system vendors. Overall, ForCES will enable rapid innovation in both the control and forwarding planes while maintaining interoperability. Scalability is also easily provided by this architecture in that additional forwarding or control capacity can be easily added to existing network elements.

Currently, the work of the ForCES WG have published three RFCs, Linux Netlink as an IP Services Protocol (RFC 3549), Requirements for Separation of IP Control and Forwarding (RFC 3654), Forwarding and Control Element Separation (ForCES) Framework (RFC 3746), and also have three current drafts ForCES Forwarding Element Model, ForCES Protocol Specification, and ForCES MIB undergoing review.

2.1 ForCES Model

The ForCES model is an abstraction of the actual Forwarding Plane, using distinct logical functional blocks (LFBs), which are interconnected in a directed graph, and receive, process, modify, and transmit packets along with metadata.

Forwarding Elements (FEs) are logical entities in the forwarding plane that implement the ForCES protocol. Fes, expose their capabilities and state to their assigned Control Element (CE) using a standardized model. A key design choice of the modelling is to avoid an extensive and hence complex modelling of the FE and instead use a coarse model combined with runtime error reporting.

CEs are logical entities that implement the ForCES protocol in the control plane and use it to instruct one or more FEs how to process packets.

A ForCES Network Element is comprised of at least one CE and at least one FE.

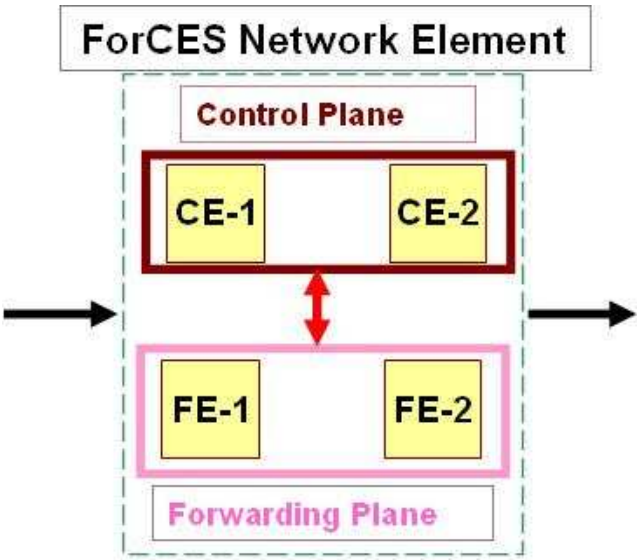

Figure 2.1: ForCeS Network Element

The modeling is based on Logical Function Blocks (LFBs), which are blocks encapsulating fine-grained operations of the forwarding plane. Each LFB is responsible for completing only a single, specific task, such as reducing the TTL field of an IPv4 packet [[4]]. FEs may have a static LFB topology for which only the state of some LFBs can be changed using the ForCES protocol, or a dynamic topology that allows the linkage between LFBs to be logically reorganized dynamically, hence allowing new functions to be created. Note that the LFB-based model is independent of the actual implementation of the FE; it only provides a view of its capabilities and state that can be acted upon using the ForCES protocol. The model can be described with XML, but the protocol uses a more compact representation on the wire between the CE and FE.

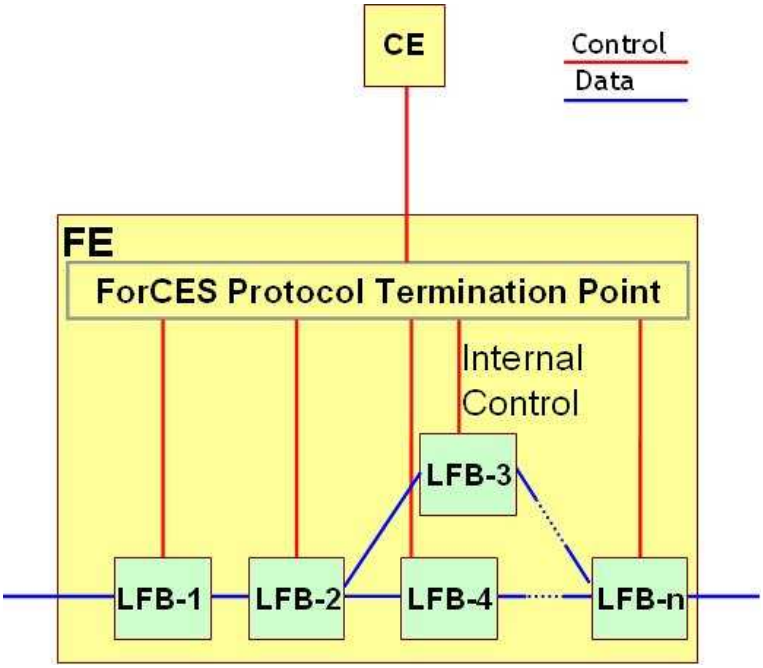

Figure 2.2: FE Internally.

The modeling language used by the WG is XML. XML was chosen as the specification language because it has the advantage of being both human and machine readable with widely available tools support.

The ForCES FE model includes both a capability and a state model.

The FE/LFB capability model describes the capabilities and capacities of an FE/LFB by specifying the variation in functions supported and any limitations. Capacity describes the limits of specific components (example would be a table size limit).

The state model describes the current state of the FE/LFB, that is, the instantaneous values or operational behavior of the FE/LFB.

Only modelling of the forwarding plane is within the scope of the ForCES working group. Also, FEs can only be controlled by a single active CE, hence different services that require state changes in LFBs belonging to the same FE have to go over the same CE.

2.2 ForCES Protocol

Forwarding and Control Element Separation (ForCES) defines an architectural framework and associated protocols to standardize information exchange between the control plane and the forwarding plane in a ForCES Network Element (ForCES NE). ForCES protocol is to standardize the information exchange between Control Elements (CEs) and Forwarding Elements (FEs) while the ForCES FE model presents a formal way to define

FE (LFBs) using XML. The LFBs within the FE are accordingly controlled in a standardized way by the ForCES protocol.

The protocol is split into a higher layer transport-independent protocol layer (PL), and a lower layer called Transport Mapping Layer (TML) that performs the mapping to specific transport layers, such as TCP, UDP, Ethernet, etc. At the current stage, the design team has defined the PL layer with message types and necessary TLVs (Type Length Value) to perform and maintain the association between Control Entities (CEs) and Forwarding Entities (FEs), to send configuration messages to the FE, to query FE information, and to subscribe to FE events. In parallel, the ForCES model team has come up with a comprehensive state and capability model for FEs, including a first definition of common logical functional blocks (LFBs) that can be manipulated over the ForCES protocol.

Having standard mechanisms allows CEs and FEs to become logically, and potentially physically, separated standard components. ForCES focuses on the communication and model necessary to separate control-plane functionality such as routing protocols, signalling protocols, and admission control, from data-forwarding-plane per-packet activities, such as packet forwarding, queuing, and header editing [[1]], [[4]], [[5]].

Simultaneously, another IETF standard regarding Network Configuration is currently been researched, named Netconf. Netconf and ForCES have similar aims but use a different model. As stated in [[1]], the reasons for separating the control from the forwarding plane is to provide increased scalability and to allow the two planes to evolve independently, promoting innovation, and extending the scope of applications.

2.2.1 ForCES Protocol Details

2.2.1.1 *ForCES Protocol Phases*

The ForCES interface between CE and FE is configured during a pre-association phase. The actual ForCES protocol takes place in the post-association phase.

In this phase, the FE and CE components communicate with each other using the ForCES protocol (PL over TML) as defined in this document.

There are three sub-phases:

1. Association Setup Stage.

   The FE attempts to join the NE. The FE may be rejected or accepted. Once granted access into the NE, capabilities exchange happens with the CE querying the FE. Once the CE has the FE capability information, the CE can offer an initial configuration (possibly to restore state) and can query certain components within either an LFB or the FE itself.

2. Established Stage.

In this stage, the FE is continuously updated or queried. The FE may also send asynchronous event notifications to the CE or synchronous heartbeat notifications if programmed to do so. This stage continues until a termination occurs, either due to loss of connectivity or due to a termination initiated by either the CE or the FE.

3. Association Lost Stage

In this state, both or either the CE or FE declare the other side is no longer associated. The disconnection could be initiated by either party for administrative purposes but may also be driven by operational reasons such as loss of connectivity.

2.2.1.2 *ForCES Protocol Mechanisms*

ForCES provides several mechanisms:

- Transaction capabilities and Atomicity of transactions.
- Two phase commits.
- Batching/parallelization.
- High availability and failover as well as command pipelines.
- Scalability.
- Heartbeat messaging.

2.3 ForCEG Specification

While the current scope of ForCES is the communication between one CE and one FE in a local area network, and the configuration of the FE by the CE, ForCEG [10] extends the availability of ForCES by providing a Web Service Interface, which facilitates the coexistence of Control Elements using a central Control Logic, and manages the state of the forwarding plane elements by staying aware of the current state of all the FEs.

The protocol stack that the ForCEG is based on is shown in Figure 2.3. ForCES protocol messages are exchanged over TCP. Web Services use XML, which is encapsulated in SOAP messages. The Control Element sends a Web Service request to the ForCEG, which attempts to translate the request into a ForCES message. When an FE wants to send a message to a CE, the ForCEG attempts to translate the ForCES message into a Web Service response that the CE will recognize.

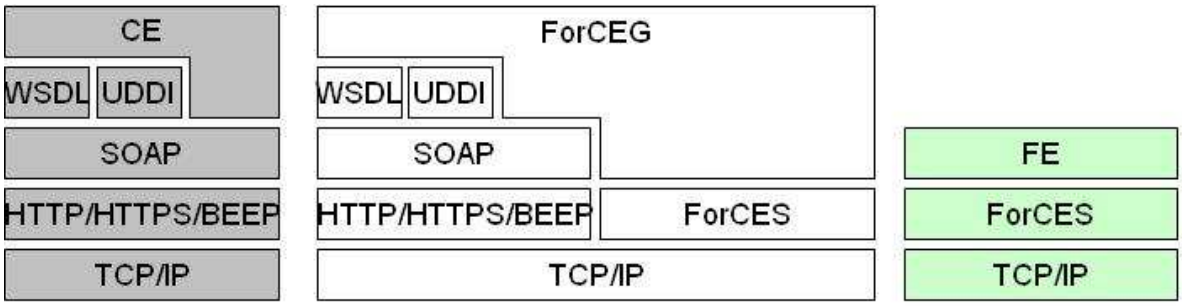

Figure 2.3 ForCEG Protocol Stack

In addition to translation from XML messages to ForCES messages and back, the ForCEG has a consistency-checking functionality: As it is placed between the Control Element and the hardware, it is the checking point of all exchanged messages. Messages from some CEs may disrupt the normal functionality of other CEs. The ForCEG monitors such messages and either informs the CEs as to which functionality is about to be changed, or disallows the message.

The ForCEG can recognize which LFBs the CE needs to modify. A CE "targets" a higher-layer function, provides the necessary attributes according to the "target" field, and then inserts an operation (such as SET or GET). The ForCEG is then able to construct the necessary ForCES messages to be sent to the appropriate LFBs.

Figure 2.4 depicts the "Target" concept. The ForCES FE Start/Termination Point is a necessary component for the ForCES protocol as it is the source and the destination of ForCES messages. As the QoS software module needs to setup classifier rules, it "targets" a QoS-related function, provides the necessary classification rules, and issues a SET command. A translator module inside the ForCEG will translate the attributes of the message, and a ForCES CE Start/Termination Point will transmit the ForCES message to the.

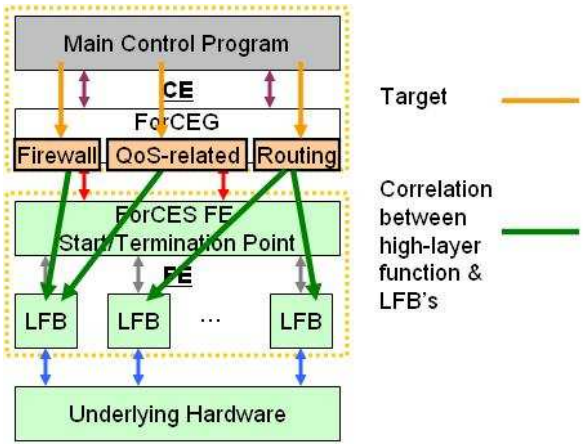

Figure 2.4. "Target" Concept

2.4 ForCEG Architecture

The ForCEG architecture and its major architectural components is depicted in Figure 2.5.

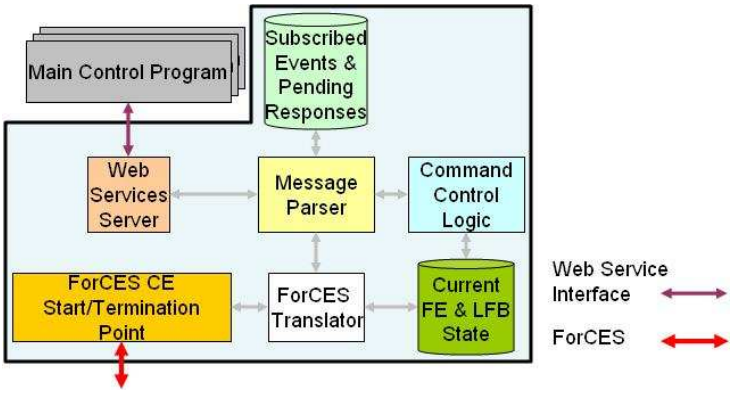

Figure 2.5 ForCEG Architecture

The Web Services Server hosts the interfaces between the CE and the remainder of the ForCEG. It is responsible for sending and receiving messages from different CEs. All interfaces between CEs and the Web Services Server are through Web Services.

The Message Parser is a central module that has two functions, depending on the flow of a message. If the message arrived from the CE then the Message Parser receives the XML message from the Web Service Server, parses it, and instructs the Command Control Logic (CCL) to check the validity of the message. When it receives a positive response (i.e., a message will not create any conflict with LFB configuration), it forwards the message to the ForCES Translator. If the message arrived from the FE then the Message Parser receives the XML message from the ForCES Translator and instructs the Subscribed Events & Pending Responses (SEPR) module to identify the CE(s) that should receive the response message(s). Once the CE(s) has been identified, the Message Parser sends the message(s) to the appropriate CE(s) through the Web-Service Server.

The CCL module is responsible for performing coordination functions to prevent contradictory commands from different processes. It is responsible for examining each command issued by each CE and making sure that a new message does not affect the current functionality of existing CEs in any way. It takes the data it requires from the “Current FE & LFBs’ state” module.

The ForCES Translator module has two functions, depending on the flow of a message. If the message arrived from the CE then the ForCES Translator receives an XML message from the Message Parser and translates the message into a ForCES protocol message that is based on the higher-layer function the CE wants to configure. It obtains the correlation data from the “Current FE & LFB’s state” module. If the message arrived from the FE then the ForCES Translator receives a ForCES message from the FCSTP and translates the ForCES message into XML. First the ForCES translator reads the ForCES message and determines the ForCES message type (i.e. packet redirection from an FE to an CE, notification message), in order to add an XML tag which will be identified by the Parser, and then the ForCES Translator inserts the packet into the XML message. Depending on the message type, the treatment of the packet varies. If the message is a packet

redirection, the packet is encapsulated into an array of bytes in the XML message, whereas in the case of a notification message, the notification is processed and transmitted as string values.

The FCSTP module has two functions depending on the flow of a message. If the message has arrived from the CE then the FCSTP receives a ForCES protocol message from the Translator. The FSCTP is responsible for sending the message to the appropriate FEs by checking the specific field in the ForCES protocol message. If the message has arrived from the FEs then the FCSTP sends the ForCES message directly to the ForCES Translator. In addition to the message-based functions, the FSCTP has functions regarding the ForCES protocol: it is responsible to create, maintain and destroy associations with FEs.

The SEPR module contains information as to which CE any response arriving from the FE should be re-directed to. CEs may also subscribe themselves to specific events. The event notification messages that arrive at the FCSTP are redirected to the required CEs based on information it holds about which CE has subscribed to which event(s).

The CFLS module holds information about all of the associated FEs and their LFBs. It also needs to have all configurations of the LFBs. This data is used by both the ForCES Translator and by the Command Control Logic.

3 ForCES Token Based Switch Architecture

The section provides an overview of the ForCES TBS-IP architecture and design considerations and then describes the token-based policy enforcement (TBS) mechanism used in the TBN implementation. TBS on TBN is being developed by UvA as a solution of adding in-band policy enforcement function to on-demand network resources provisioning [6]– [7].

3.1 TBS-IP architectural/design requirements

The TBS architecture offers to user applications a flexible access control approach/solution to the reserved high-speed links (lightpaths) across multi-domain hybrid networks. The procedure consists of two phases that are decoupled in time: (1) a resource reservation and activation/set-up phase (resulted in obtaining reservation ID and security context for the reservation from the domain controller of domain AAA service), and (2) the reserved datapath access achieved with low-level authorisation checks (per-packet token checks at network edges within a multi-domain end-to-end connection). In other words, the first phase allows individual users or group of users (e.g., a research institution), or even user applications, to request privileged end-to-end connection across multi-domain networks by contacting relevant authority. e.g. their own ISP or local NRPS. The second phase determines how TBS authenticates network traffic (TCP connections, UDP transmissions, or other protocols on top of IP protocol) and how it checks the traffic for authorisation on behalf of their applications. The second phase is also responsible for preventing malicious use of lightpaths in a multi-domain network. From the technical point of view, one component is involved in the control plane, the Token Validation Service (TVS), and two network components are involved in the dataplane: the token builder (TB) and the token switch (TS).

The following goals had been identified:

- Implementing the token over IP principles inside the data path software modules, token builder (TB) and token switch (TS), developed on a network processor hardware architecture. The software modules are remote configurable through ForCES standard interface that stands for FORwarding and Control Element Separation;

- Ensuring design modularity in order to have a single TBS-IP system that works on multiple application scenarios and hence, different low-level requirements, chosen at loading time through the ForCES control path. For example, TBS-IP/TB is a Token Builder configuration, TBS-IP/TS is a Token Switch configuration, or TBS-IP/TS-GMPLS is a Token Switch with gateway to GMPLS domains;

- Allowing interconnection to 3rd party systems such as Dragon-VLSR used in GMPLS and therefore, it provides token features over the GMPLS paths (TBS-IP/TS-GMPLS);

- Ensuring fully system configurability using secure Webservice as a high-level interface to an authority (Domain Controller or AAA servers).

Summarising, the request is to create a software system that is modular, configurable for different application scenarios, and using standard communications and special hardware for packet processing at high speeds (network processors).

In the next section, Figure 3.3 shows schematically one Token Based Switch system at IP layer (TBS-IP) that uses a network processor hardware platform (IXP2850 dual-NPU). An IXP2850 network processor has one general purpose CPU (XScale) for control of the entire architecture composed of parallel specialised cores for traffic processing called microengines (µEngines).

3.2 TBS-IP operation in TBN

The generic TBN architecture uses the token-based signalling at the reservation or path building stage and token based access control when accessing the reserved path. Both stages use different types of tokens that share common GRI as identification of the reserved path/resource. In the token based switch over IP (TBS-IP) we opt for in-band signalling for reasons of flexibility resulting from the per-packet granularity. Specifically, we insert tokens into each IP packet as a proof of authorisation. Tokens are a simple way to authorise resource usage which may convey different semantics, or security context. For instance, we may specify that only packets with the appropriate token are allowed to use a pre-established network connection in a specific time-frame and embed these tokens in the packets of an application distributed over many IP addresses [6]. An advantage of such approach is that time of reservation is decoupled from the time of access or use.

Token Based Switch (TBS) is a low-level system for traffic routing at high speeds (multi gigabits/sec) based on packet authentication. TBS can be used with high-performance computing and grid applications that require high bandwidth links between grid nodes to bypass the regular Internet for authorised packets by establishing shortcuts on network links with policy constraints.

TBS is fast and safe and uses the latest network processor generation (Intel IXP2850). TBS is feasible at multigigabit link rates. In addition, it has the following goals: (1) path selection with in-band admission control (specific tokens gives access to shortcut links), (2) external control for negotiating access conditions (e.g., to determine which tokens give access to which links), and (3) secured access control.

Figure 3.1 and Figure 3.2 illustrate a few use cases for a TBS-IP router usage in multi-domain networks.

A TBS-IP system works as follows: the client (e.g., user/application requestor in Figure 3.1) contacts an AAA server separate from the datapath (e.g., by using using separate or dedicated channel) to obtain authorisation for the use of network resources (for instance, the optical shortcut B). Then, the AAA server checks the credit of the client and also the availability of the requested network resources. When both conditions are positive, the AAA server generates an authorisation ticket (AuthZ ticket) that contains as proof of authorisation the following items: a unique Global Reservation Identifier (GRI), a key (TokenKey), and a description of the required lightpath across the network. The AAA server sends the AuthZ ticket to every policy enforcement point (PEP) through their TVS along the required path and also back to the client. Next, when the user is ready to use the resources, the user client pushes the proof of such authorisation to the service equipment such as the Token Validation Service in a token builder module (TVS/TB) on the network device. TVS/TB module checks the validity of the AuthZ ticket and generates a proper token for each data packet that goes out of the client's system. This token travels together with data across the networks. Every PEP across a multi-domain network (TBS-IP domain1, TBS-IP domain2, etc.) has specific hardware that checks the built-in token from each received data packet against a local AuthZ ticket. When the PEP's check is positive, then data packet takes an authorised path. Otherwise, it takes a default path such as slow routed networks. In other words, the proof of authorisation (token) drives the data over the networks in order to reach the destination within the required specifications (bandwidth, delay, etc.) by making use of specific network resources such as lightpath shortcuts (e.g., route (B) in Figure 3.1).

Figure 3.2 shows another context where TBS-IP systems work together with TBS-GMPLS. In this scheme we suppose to have some GMPLS lightpaths within the end-to-end connection that uses normally IP packets.

A Token Based Switch (TBS) is hardware and software system that receives authorisation requests for certain IP packet flows from a high-level authority (NSP/AAA servers) and allows for intensive packet processing (encryption operations) at high speeds (multigigabits/sec). This system is going to be plugged into an AAA framework for high speeds lightpaths selections especially when the traffic crosses multi-domain networks. In order to achieve the above mentioned requirements, we need to design and develop a modular system that separate the control path by data path in a manner that also provides standard and secured communications to different levels of components. In this respect, we use SOAP/XML for high-level and ForCES for the low-level communications.

Summarising, each TBS-IP system plugged in multi-domain networks (as shown in Figure 3.1) provides packet routing based on a global reservation identifier (GRI). This GRI is a value unique per end-to-end lightpath and eventually issued per application (not a host, as an application might use multiple physical hosts). There is no need for direct signalling between TBS-IP and GMPLS systems because the signals (open/close path) are handled by the high-level authority (NSP/AAA) which knows specific interfaces to both TBS-IP and GMPLS systems.

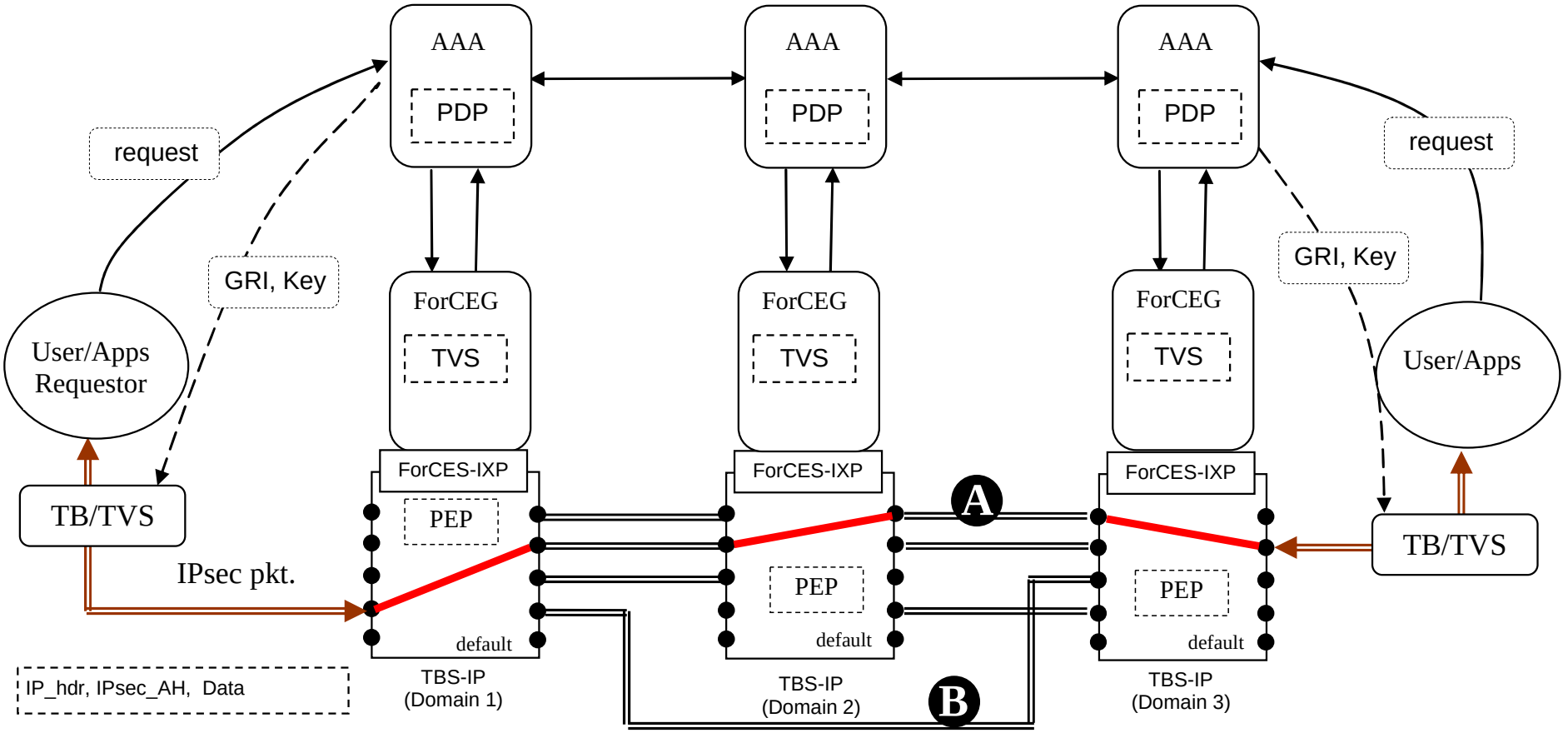

Figure 3.1. Providing end-to-end lightpaths over multidomain networks using TBS-IP systems interconnected into an AAA framework.

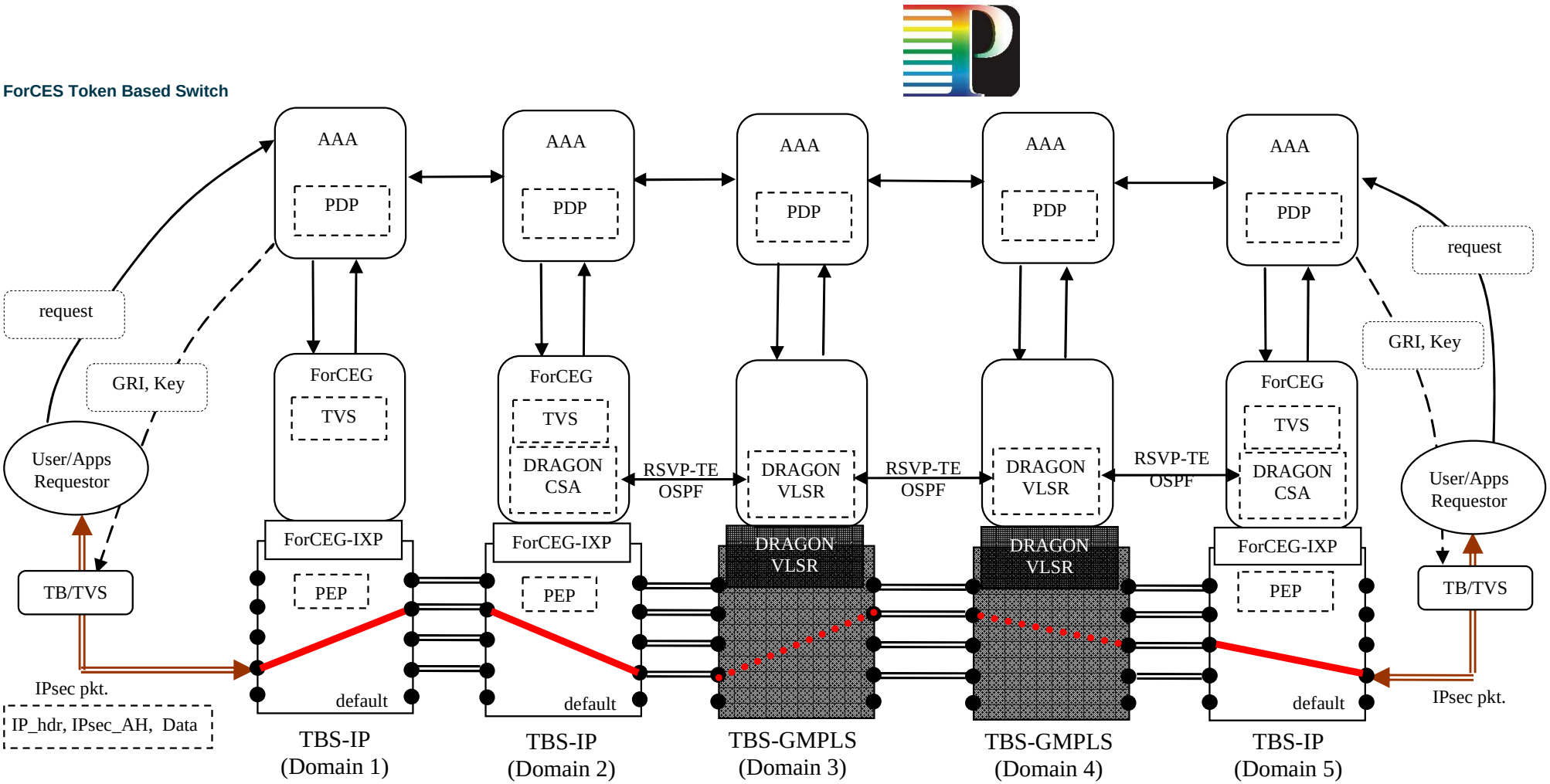

Figure 3.2. Providing end-to-end lightpaths over multidomain networks using a mix of TBS-IP with GMPLS systems.

3.3 ForCES TBS-IP system architecture

As shown in Figure 3.3, a host PC runs a webservice for the outside world interface (AAA server). The host PC connects to the specialised hardware for packet processing, IXDP2850, through a standard interface: ForCES [5]. The IXDP2850's control core (XScale) runs embedded linux that supports the control path of the ForCES interface, and each µEngine runs a custom packet processing task that implements specific forwarding elements like packet receiver, token builder, token switch, packet transmitter.

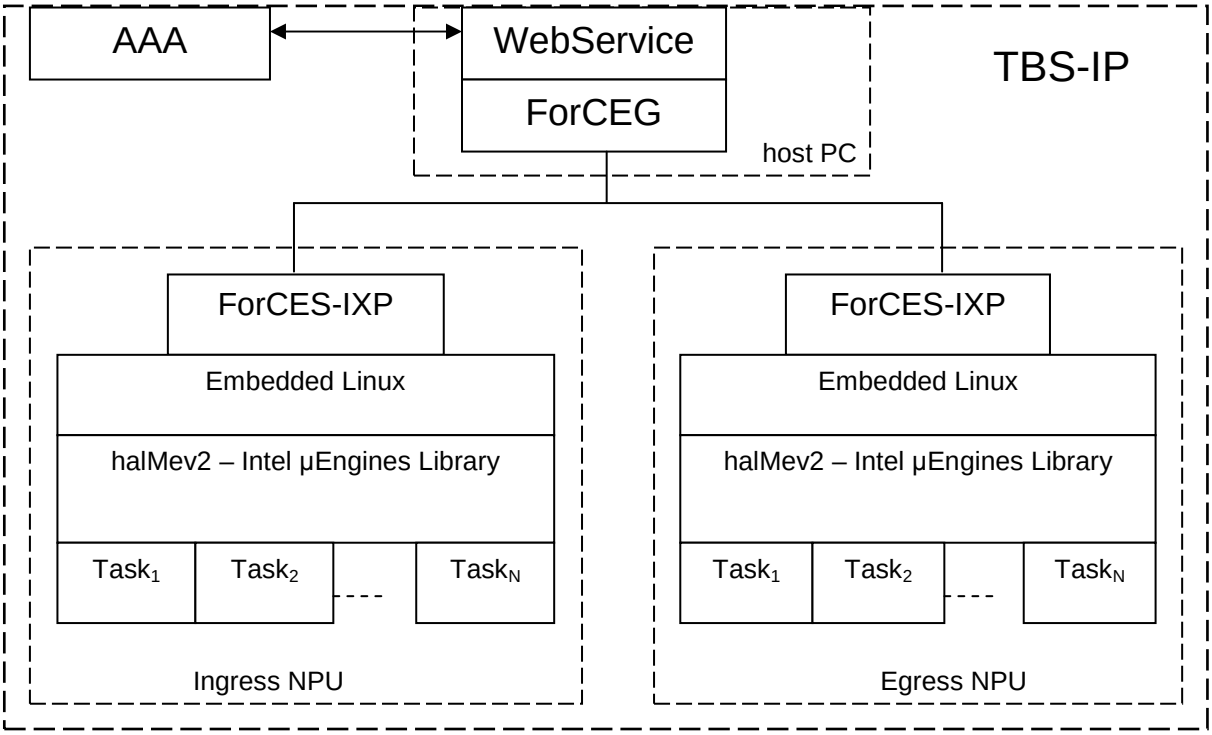

Figure 3.3. TBS-IP system architecture.

4 ForCES Token Based Switch Implementation

4.1 Requirements

Designing a Token Based Switch architecture following the ForCES standard guidelines we identified the following requirements:

- Modularity – provides flexibility in adding new features;
- Safety – prevents a rogue user to use privileged resources such as ligtpaths;
- Configurable – allows for various test-bed scenarios;
- Hiding low-level implementation details;
- Uses hardware specifically designed for network processing at high speeds (Network Processors) and also benefits of hardware supported encryption.

The TBS system has the followings states:

- Self-Initialisation: hardware components (NPUs, memory, registers, etc.) and software components (O.S. loading, drivers loading, description tables, buffers);
- A host connects to the TBS system remotely;
- The host browses for the capabilities (features) within the current TBS system;
- The host configures the connected TBS in a required state (e.g., Token Builder, Token Switch);
- The host starts the system;
- The host updates in run-time the required parameters in the TBS system: adds new authorised TokenKeys, removes ‘expired’ TokenKeys from the AuthorisationTable;
- The host fetches periodically some of the debugging information (e.g., authorised packets, unauthorised packets);

4.2 Software item architectural design

This section describes what software modules are needed and how they are used.

As illustrated in Figure 4.1, the system is composed of the following software modules, described in a bottom-up approach (low-level, data-path, to high-level, control path):

- FIX2850, having the following sub-modules:
    - Rx;
    - Tx;
    - TB (TokenBuilder);
    - TS (TokenSwitch);
- ForCES-IXP, having the following sub-modules:
    - Init_HW, Init_SW, ueManager;
    - FE-IXP server (over TCP);
- GMPLS gateway;
- ForCEG-WebService interface to the outside-world.

4.3 Interface design

Figure 4.1 also highlights the interfaces used in the TBS-IP system:

1) outside world via WebServices;
2) CE-CE using a simple communication over TCP;
3) CE-FE using ForCES standard over TCP;
4) FE-LFBs (control – data path specific implementation for IXP network processors).

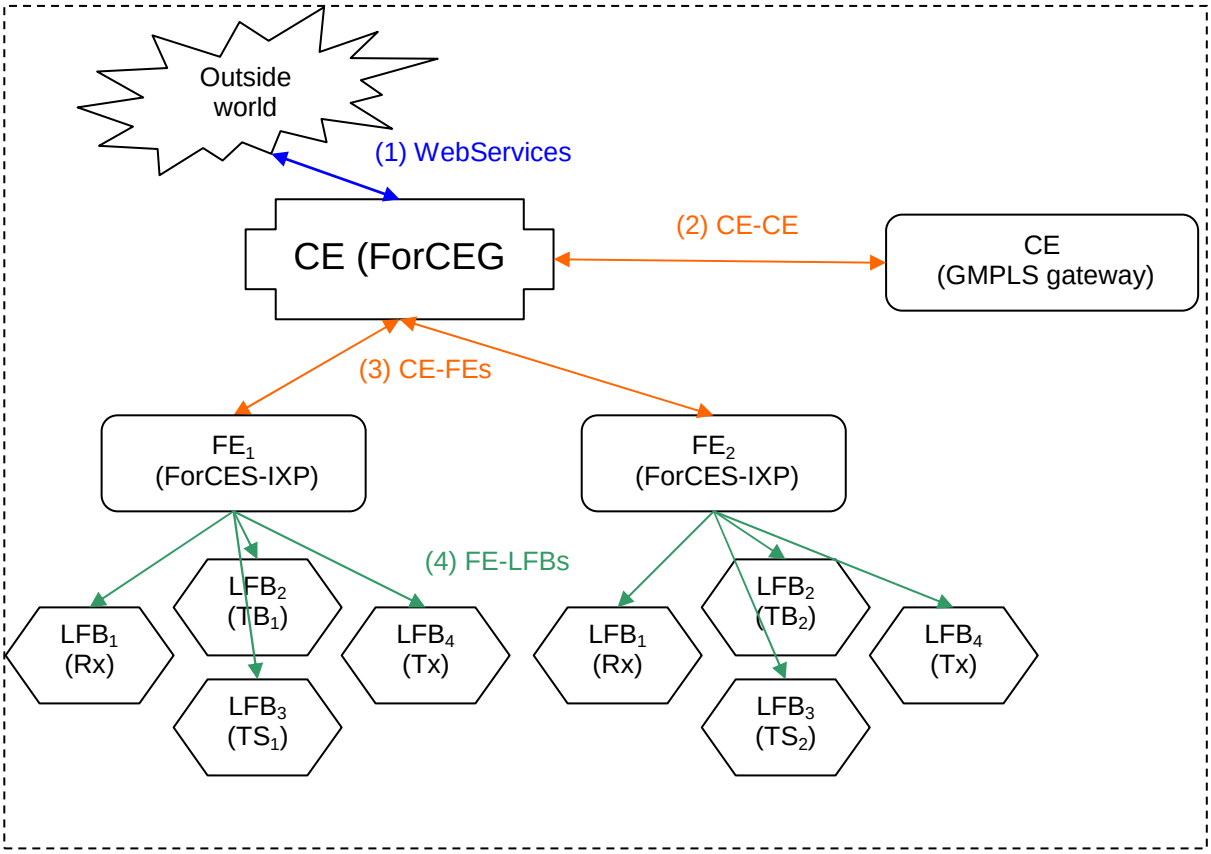

Figure 4.1. Interfaces used in the TBS-IP system

Although Figure 4.1 shows all the interfaces the TBS-IP system might use, we identify practically only the following four configuration schemes for different usage of TBS-IP:

1) TBS-IP/TB (Token Builder);
2) TBS-IP/TS (Token Switch);
3) TBS-IP/TS-TB (Token Switch and Builder);
4) TBS-IP/TS-GMPLS (Token Switch Gateway to GMPLS);

Each configuration is described simultaneously in the following sections on both levels: control and data paths.

4.4 Interface specification: TBS-IP/TB

TBS-IP/TB illustrates the Token Builder application. It is usually located at both ends of an end-to-end light-path. In other words, each user needs to have such a system that annotates the packets of authorised user applications with “tokens”.

In our dual-NPU implementation, we share the effective packet processing task for building and inserting the token (TB) between those two NPUs in order to benefit of all hardware encryption units available (2 units per NPU). The system works as follows: the received packets are stored into a shared packet buffer and made available for the next processing tasks (TB_1 and Tx) so as to load balance between these two tasks as shown

in Figure 4.2, (1). One half of the received packets is further processed by the TB_1 task (2) and the other half is simple forwarded out to the second NPU (egress) by the Tx module (3). The Egress receiver does the same job as the one in Ingress, except that the load-balance is done by checking whether the packet has been already processed by the TB_1 in the first NPU or not. The processed packets are forwarded out of the system via Tx module, and the other packets are en-queued to the TB_2 for processing.

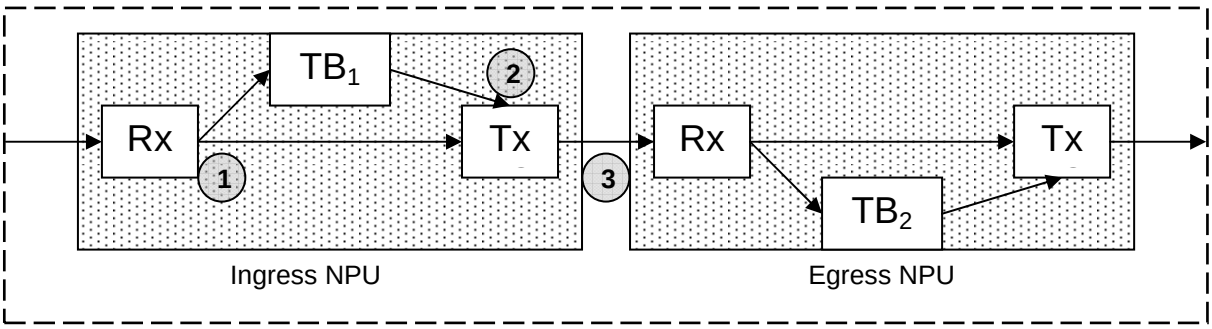

Figure 4.2. TBS-IP/TB Interface

The control path of this TBS-IP/TB application is shown in Figure 4.3. It illustrates the ForCES connections from CE (ForCEG running on a hostPC) to those two FEs (FE-IXP on Ingress and Egress NPUs).

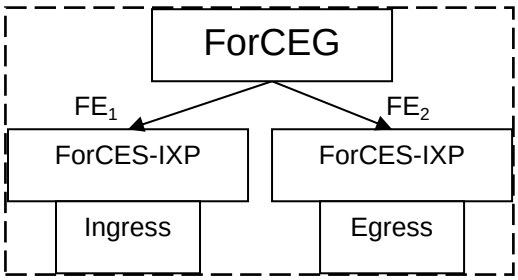

Figure 4.3. Two FEs (ForCES-IXP) slaves connected to the CE master (ForCEG).

4.5 Interface specification: TBS-IP/TS

TBS-IP/TS describes the Token Switch application and is illustrated in Figure 4.4. TBS-IP/TS is located at every domain border across a multi-domain light-path. This application decides which packets should take a certain 'short-cut' (part of the established light-path) or should be forwarded out to the routed network (Internet).

Same as in TBS-IP/TB application, we need to make use of the dual-NPU features and hence, we share the effective packet processing task for checking the token (TS) between those two NPUs.

The control path of TBS-IP/TS application is same as the one used in TBS-IP/TB as shown in Figure 4.3.

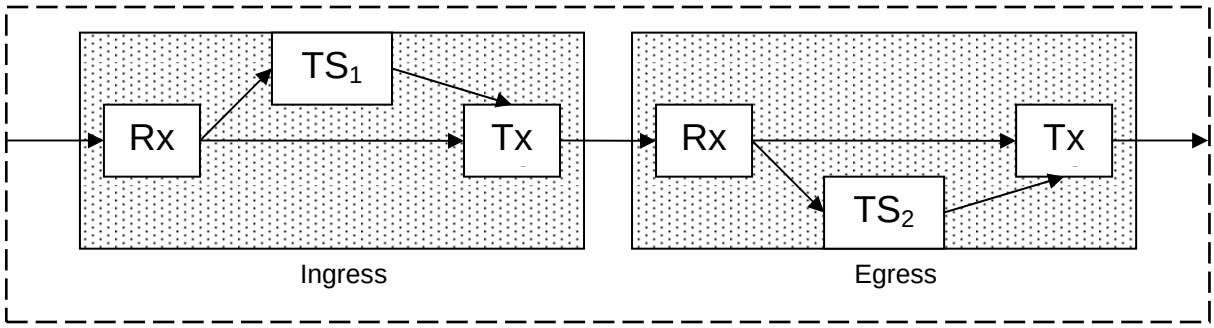

Figure 4.4. TBS-IP/TS Interface

4.6 Interface specification: TBS-IP/TS-TB

TBS-IP/TS-TB describes a combination of the above mentioned two applications: Token Builder and Token Switch. It can be located in every domain border across a multi-domain light-path by replacing the simple Token Switch application when the next domain requires a different key/identifier than the current domain. This application, like the simple Token Switch, decides which packets should take a certain 'short-cut' (part of the established light-path) or should be forwarded out to the routed network (Internet). In addition to the TokenSwitch, the TBS-IP/TS-TB application also rebuilds the token of every outgoing packet to the authorised ports.

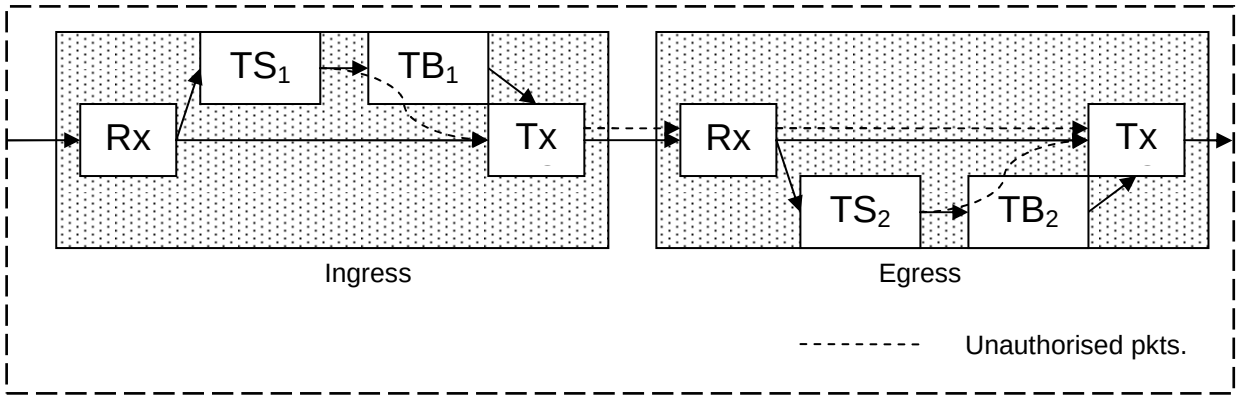

Figure 4.5. TBS-IP/TS-TB Interface

Same as in TBS-IP/TB application, we make use of the dual-NPU features and hence, we share the effective packet processing task for checking the token (TS) and for building a new token between those two NPUs. We chosen to map the TS, TB modules onto those two NPUs as shown in Figure 4.5 because of the following facts gained from a demo system built previously: the most computation required by the TS module consists of encryption, while TB also uses lots of memory operations for token insertion in the IP packet. Therefore, the chosen mapping (TS-TB pairs on each NPU) fits better than an eventually TS-TS / TB-TB mapping.

The control path of TBS-IP/TS-TB application is same as the one used in the previous two applications as shown in Figure 4.3.

4.7 Interface specification: TBS-IP/TS-GMPLS

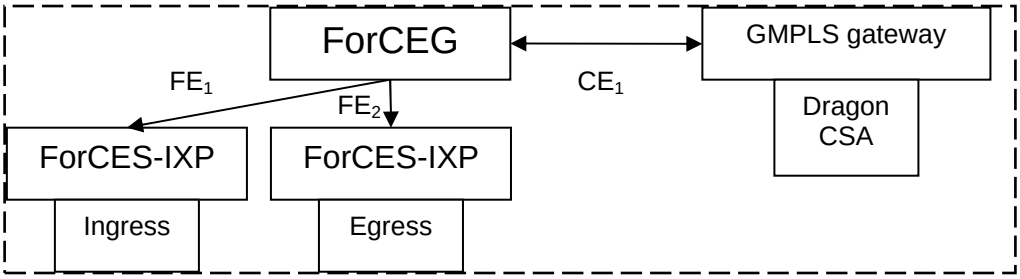

Figure 4.6 TBS-IP/TS-GMPLS Interface.

TBS-IP/TS-GMPLS is another application that has special requirements for routed packets across two domains that use different technologies (one uses tokens over IP, while the other one might use tokens over GMPLS). In this case, we use for the data-path the same mapping as in TBS-IP/TS application, but for control path we need to connect to the control path of the GMPLS system that is called Dragon-CSA framework. Therefore, as shown in Figure 4.6, the ForCES architecture uses one CE interface that consists of the connection to the CSA system.

4.8 Interface specification: TBS-IP to outside world (AAA server)

Figure 4.7 shows the interface between the outside world (AAA server) and the ForCEG.

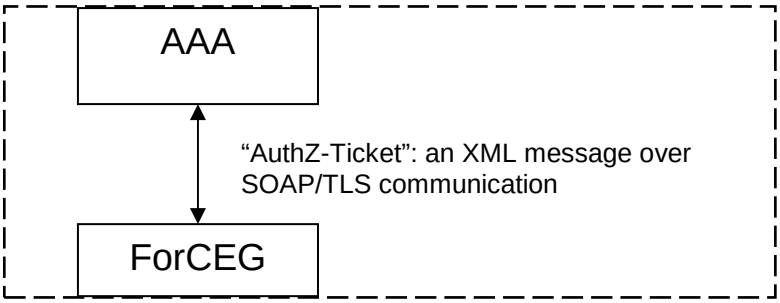

Figure 4.7 Interface TBS-IP to outside world (AAA server)

4.9 ForCES protocol in a TBS-IP router

4.9.1 Functionalities from ForCES required for CE-FE interface in TBS-IP

We model the interface between Control Element (CE) and multiple Forwarding Elements (FEs) using ForCES standard, as follows:

- **Association phase:**
  - CE interconnects to FE(s);
  - CE asks for capabilities to the FE(s);
  - FE(s) answers: (e.g., TS, TB);
  - CE requests to one FE to become TB, etc.;
  - FE then instructs HW to map the TB application;
- **Configuration phase:**
  - Add/Remove tuple, Changing in LFB attributes (e.g., Connector changes);
- **Events:** diagnostic messages from low-level reported to higher level;
- **Query:** reading of specific attributes of LFBs such as packet counters: pkts_count, processed_pkts_count

4.9.2 Modeling of applications in ForCES

We model the required function blocks for TBS-IP applications using ForCES standard, as follows:

- **FEs:** ingress, egress;
- **LFBs:** Rx, Tx, TB, TS, Tx_CSIX, Rx_CSIX;
- **Attributes:**
  - TUPLE (GRI, TokenKey, GRI_NextDomain, TokenKey_NextDomain, Port1, Port2, IPPacketMask, IPSource, IPDestination, PortSource, PortDestination, Status);
  - LOCATION (e.g., LFB.Rx located on hw_core 0);
  - Connector(val,val);
  - Pkt_count that is available in LFB(Rx, Tx, Tx_CSIX, Rx_CSIX);
  - Processed_pkt_count available in LFBs(TS, TB);

4.9.3 XML Model of TBS based on ForCES Model

Based on the ForCES Model draft which provides an XML schema for creating LFBs, there are only for distinct LFBs (Rx, Tx, TB and TS) because the Rx and Rx_CSIX have the same model, as Tx and Tx_CSIX. The actual XML can be found in **Appendix C**.

Each LFB has a Packet Processed counter and TS has an additional counter Bad Token Counter for each Tuple. The ForCEG will be able to register for an event based on the Bad Token Counter. If the Bad Token Counter exceeds a specific threshold, which will be set by the ForCEG, then the TS will send a message containing the GRI and the number of Bad Tokens.

4.10 Software item detailed design

The following software modules are described in this section:

- ForCEG high-level interface to the outside-world;
- ForCEG – plugins:
  - Message-parser;
  - TVS (Token Validation Service);
  - Gmp (GMPLS gateway);
  - APM (Advanced Power Management);
- FIX2850, having the following sub-modules:
  - Rx_Gig;
  - Tx_CSIX;
  - Rx_CSIX;
  - Tx_Gig;
  - TB (TokenBuilder);
  - TS (TokenSwitch);
- FE-IXP, having the following sub-modules:
  - FE (ForCES FE module implemented on IXP);
  - Init_HW, Init_SW, ueManager;
  - FE-IXP server (over TCP).

4.11 ForCEG high-level interface to the outside world

Figure 4.12 illustrates an example of Authorisation ticket AuthZ ticket as expressed in XML format.

Note that some of the AuthzTicket fields are not mandatory and hence, we provide specific mechanisms to derive local meanings from a global identifier. For instance, if AuthZ ticket has no specific field as “TokenKey”,

we chose to derive the TokenKey from a mandatory field SessionID also known as a global reservation identifier (GRI). Thus, TokenKey = Encryption(GRI, secret), where the secret is a shared TVS specific phrase. However, GRI could be much bigger than that of the encryption algorithm required. For instance, using HMAC-SHA1 encryption algorithm, we use for deriving of the TokenKey only the first 20Bytes of the GRI.

```
<AuthzTicket xmlns="http://www.aaauthreach.org/ns/#AAA"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.aaauthreach.org/ns/#AAA
E:\Projects\Phosphorus\Workspace\AuthZ\azticket-ehalep.xsd" Issuer="urn:ph:trusted:tickauth:pdp"
TicketID="cba06d1a9df148cf4200ef8f3e4fd2b3" SessionID="00112233445566778899AABBCCDDEEFF00">
  <Decisions><Decision ResourceID=" http://resources.test-bed.phosphorus.eu/ForCES1"
result="Permit"/>
  </Decisions>
  <Resources>
      <Resource ResourceID="http://resources.test-bed.phosphorus.eu/ForCES1">
         <LRI purpose="MyDomain">00112233445566778899AABBCCDDEEFF00</LRI>
         <Key>AAAABBBBCCCCDDDDEEEE</Key>
         <Ports>
             <Port1>6</Port1>
             <Port2>8</Port2>
         </Ports>
         <ApplicationFlow>
               <IPpacketMask>A1A2A3A4A5A6A7A8A9B1B2B3B4B5B6B7B8B9C1C2</IPpacketMask>
               <IPsource>192.168.1.10</IPsource>
               <IPdestination>"192.168.1.100"</IPdestination>
               <PortSource>123456</PortSource>
               <PortDestination>654321</PortDestination>
         </ApplicationFlow>
      </Resource>
      <Resource ResourceID="http://resources.test-bed.phosphorus.eu/ForCES2">
         <LRI purpose="NextDomain">AA112233445566778899AABBCCDDEEFF00</LRI>
         <Key>AAAABBBBCCCCDDDDEEEE</Key>
         <Ports>
             <Port1>6</Port1>
             <Port2>8</Port2>
         </Ports>
         <ApplicationFlow>
             <IPpacketMask>A1A2A3A4A5A6A7A8A9B1B2B3B4B5B6B7B8B9C1C2</IPpacketMask>
             <IPsource>192.168.1.10</IPsource>
             <IPdestination>"192.168.1.100"</IPdestination>
             <PortSource>123456</PortSource>
             <PortDestination>654321</PortDestination>
         </ApplicationFlow>
      </Resource>
   </Resources>
   <Conditions NotBefore="2006-06-08T12:59:29.912Z" NotOnOrAfter="2006-06-09T12:59:29.912Z">
</Conditions>
</AuthzTicket>
```

Figure 4.8 Example of XML based AuthZ ticket format.

As also illustrated in Figure 4.9.(1), the Authorisation ticket is an XML message received by the ForCEG-Webservice module through a local WebService container (e.g., Apache Tomcat or Sun Application Server JWSDP). Next, this message is passed to the ForCEG software package. ForCEG will send the XML message to the “Message-parser” module for processing, as shown in Figure 4.9.(2).

When Message-parser identifies a received XML stream as AuthZ ticket, it parses and calls the SetEntryTVStable() or DeleteEntryTVStable() with parsed parameters to the TVS (Token Validation Service) plugin.

4.12 ForCEG – plugins

ForCEG is a software package written in Java that implements several functionalities such as WebService interface to outside world, ForCES interface down to hardware (network processors), and other specific functionalities (e.g., validation services for the XML stream received from outside world, connection to 3rd party system that is not ForCES compliant). As new functionality was required, outside of the scope of ForCEG, these specific functionalities were designed and developed as ***plugins***. As illustrated in Figure 4.9, we have the following plugins:

- Message-parser (See Figure 4.9.(2));
- TVS (Token Validation Service, as in Figure 4.9.(3));
- Gmp (GMPLS gateway, as in Figure 4.9.(5));
- APM (Advanced Power Management, as in Figure 4.9.(6));

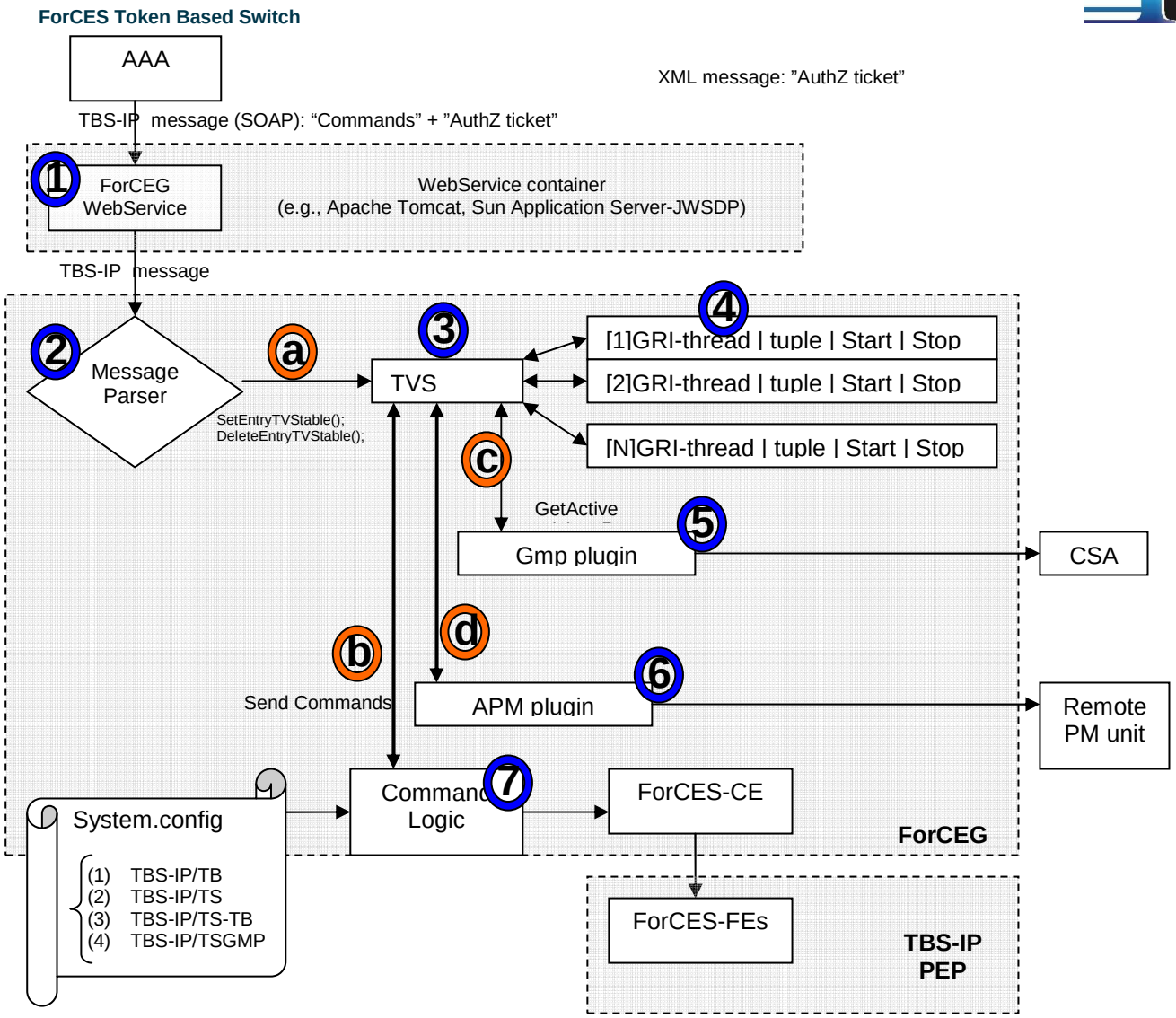

Figure 4.9 Work flow in TBS-IP switch.

When the system starts up, the ***Command Logic*** (Figure 4.9.(7)) loads a system configuration file that specifies in which mode the TBS-IP router is going to run: (1) TB-IP/TB, (2) TBS-IP/TS, (3) TBS-IP/TS-TB, or (4) TBS-IP/TSGMP. By default, the TBS router requires the following plugins: ***Message Parser***, TVS, and ***APM***. Moreover, in case of the last option, the Command Logic also starts the ***gmp*** plugin such as to establish a connection between TBS-IP router and GMPLS systems (by connecting the ForCEG to CSA).

Each of plugins is part of the same Java package: ForCEG. The plugins are described in the following sections.

4.12.1 Message-parser

Message-parser (see Figure 4.9.(2)) is a gateway module that allows future extensions to the ForCEG with other plugins. Message-parser receives SOAP/XML messages; it parses the header of the message in order to classify it. For instance, if the message has a command “Add” and the body has an AuthZ ticket, then it extracts

the useful fields from the ticket (e.g., GRI, StartTime, StopTime), and it calls “SetEntryTVSTable” to a TVS plugin, as shown in Figure 4.9.(a).

4.12.2 Token Validation Service (TVS) plugin

API:

Function	Called by module
Boolean SetEntryTVSTable(struct TUPLE, date StartTime, date StopTime);	Message-parser
Boolean SetEntryTVSTable(struct TUPLE);	Message-parser
Boolean RemoveEntryTVSTable(struct TUPLE);	Message-parser
Boolean AddPath(struct TUPLE);	GRI-thread
Boolean RemovePath(struct TUPLE);	GRI-thread
List(GRI) GetActiveGRI();	Gmp

The ***TVS*** plugin (the module shown Figure 4.9.(3)) is integrated within the ForCEG package and implements the following functionalities, as also illustrated in Figure 4.10:

- Wait for new commands (1):
    - if a ticket came with a “*Remove*” command, then it searches through the GRI-thread list, issues the *RemovePath* command if it was an active GRI (such as the hardware also removes the active path), and then it removes itself (2).
    - If a ticket came with an “*Add*” command, checks for NotValidBefore and NotValidAfter fields (3):
        - If it has no validation fields, then it is directly executes *AddPath* command without creating any thread (4);
        - When it has a validation field (specifies a future period of working time), TVS creates a new “GRI-thread” and it initialises the GRI-thread (5) with the following info: NotValidBefore, NotValidAfter, the TUPLE (LRI, Keys, AuthPort), and a reference to the parent module (TVS plugin). The reference is used for callbacks to parent;

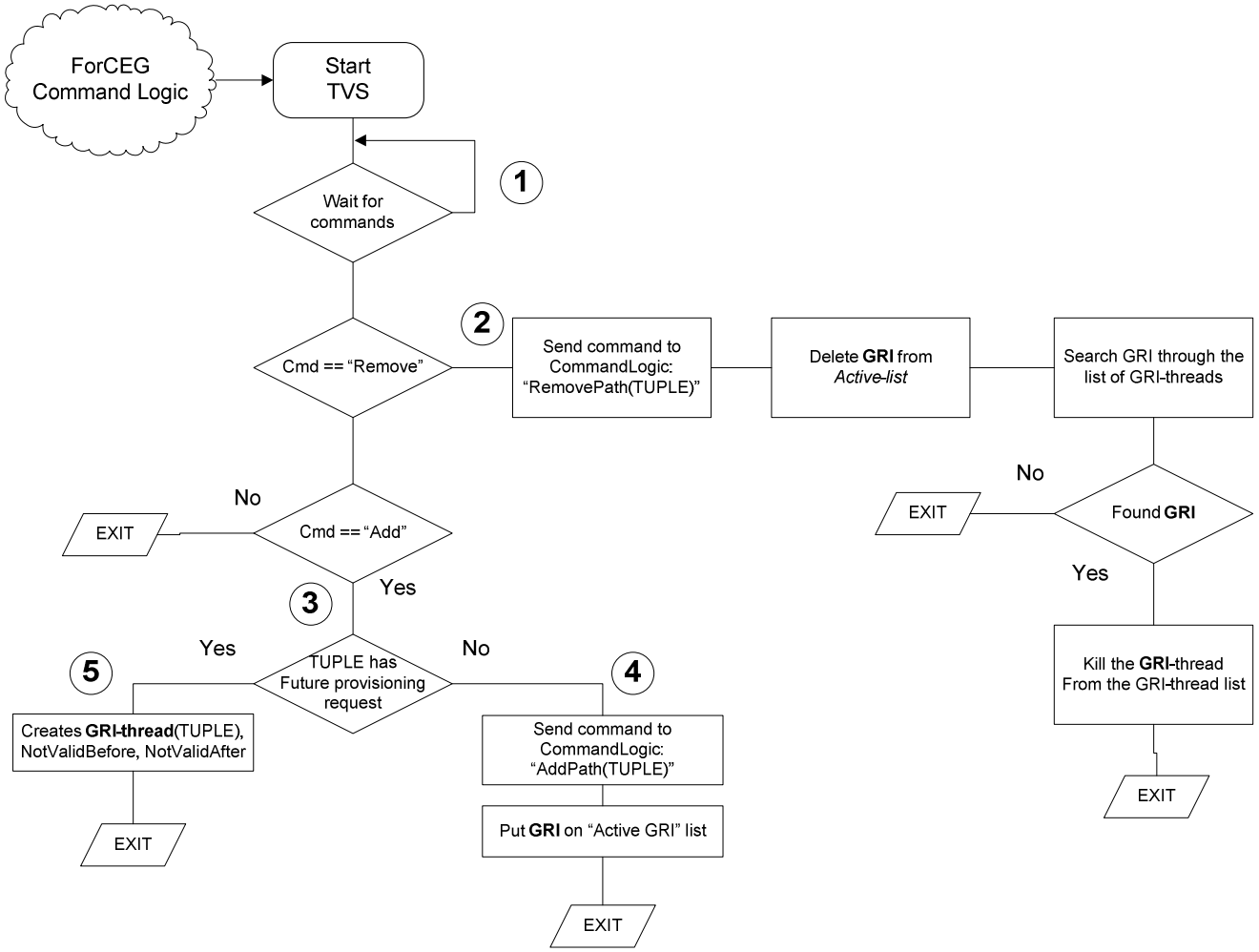

Figure 4.10 Workflow in TVS module.

- Sends commands to ForCEG on behalf of the GRI-thread (see Figure 4.9.[b]):
  - BOOL ReturnVal = AddPath(struct TUPLE);
  - This function also puts the GRI-thread on an "*Active GRI*" list.
  - BOOL ReturnVal = RemovePath(struct TUPLE);
  - This function removes the "GRI-thread" from the "*Active GRI*" list. It also executes the following checks for the power management module:
    - Computes the "*FirstWakeUp*" time (is the earliest time in future when a passive GRI-thread will send an AddPath command to hardware);
    - If (*FirstWakeUp > PM_Threshold)* then TVS sends "*STOP_HW*" command to the APM module, where *PM_Threshold* is the minimum time (e.g., 1 hour) while the hardware should be turned off for power saving reasons. As shown in Figure 4.9.[d], TVS sends commands to the APM module and receives the hardware status (ON/OFF) through the same interface (d).
  - Object AttributeVal = ReadLFBAttribute(string LFBname, string AttributeName);
  - The return object contains all values of the queried attribute. Used for checking of packet counters from Rx, Tx modules for statistic or debugging purposes.

- Process diagnostic messages received from lower levels (events from an LFB running on hardware):
  - SubscribeEvent(string LFBname, string AttributeName, int iCondition, int iThreshold);

- callback ProcessDiagnostic(string LFBname, string AttributeName, int iValue);

- Implements a power management function:
  - When TVS receives a notification signal from a passive GRI-thread, it checks if HW_STATUS==OFF then sends "*START_HW*" command to the APM module;

4.12.3 Global Reservation Id threading (GRI-thread)

A GRI-thread (Figure 4.9.(4)) is a class that runs in a separate thread container and it is instantiated by the TVS plugin. A GRI-thread has the following functionalities, as also illustrated in Figure 4.11:

1. Wait until (ValidNotBefore – *PM_StartUp)*, where *PM_StartUp* is a time needed by the power management unit to prepare the hardware (e.g., *PM_StartUp=5min)*;
2. Send a notification signal to the TVS module that this GRI needs the hardware ready (within *PM_StartUp* time),
3. Sleep for (*PM_StartUp)*, where PM_StartUp is the effective time needed to wake up the hardware;
4. Send AddPath(tuple) command to its parent (TVS plugin);
5. Wait until ValidNotAfter;
6. Send RemovePath(tuple) command to its parent;
7. Wait till proper acknowledge received when RemovePath() returns and then exit the process (kills itself);

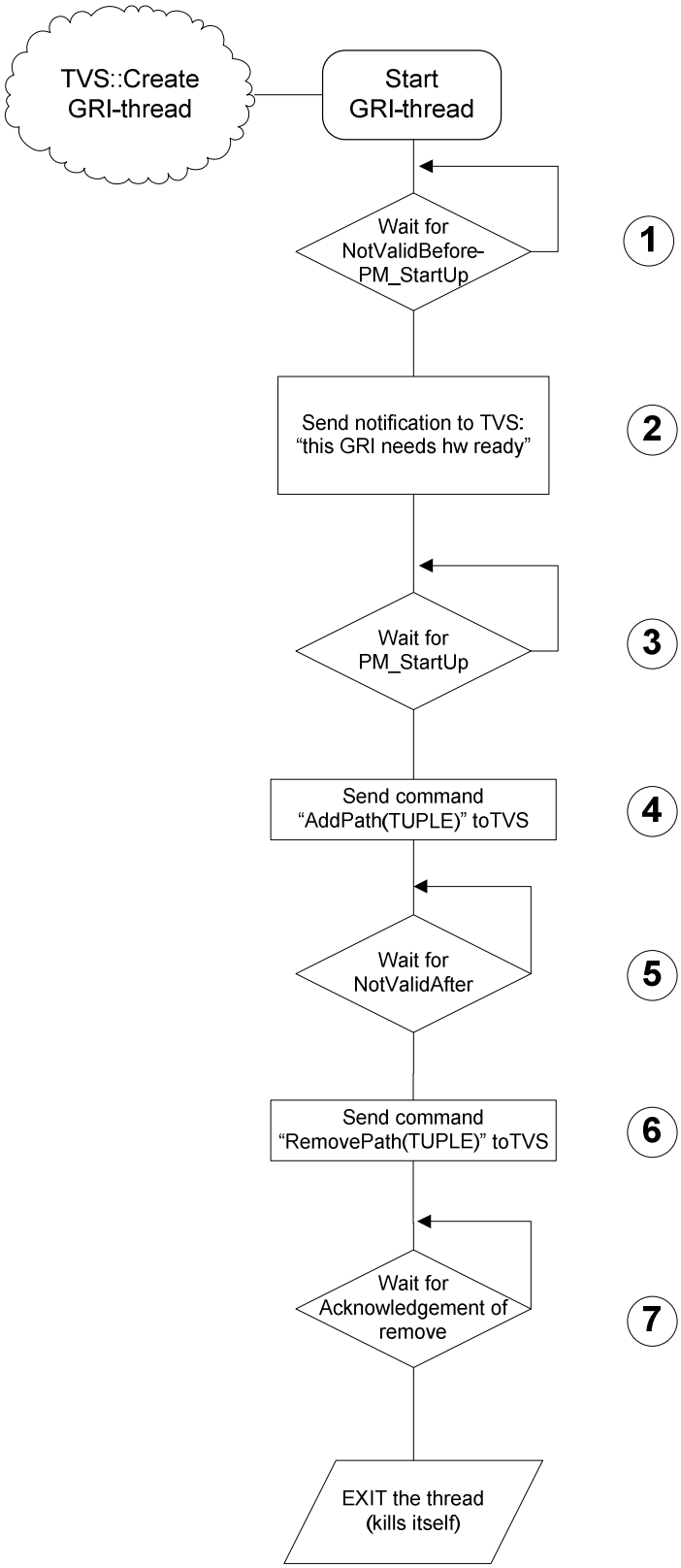

Figure 4.11 Workflow in GRI-thread module.

4.12.4 GMPLS (Gmp) plugin

API:

Function	Called by module
Boolean StartKeepAlive();	TVS
Boolean StopKeepAlive();	TVS

Gmp plugin (Figure 4.9.(5)) connects ForCEG to a 3rd party system: a GMPLS router. The GMPLS router runs a VLSR-client software called CSA. We design and implement an interface between ForCEG and CSA as a CE-CE interface (master-to-master).

The Gmp plugin has the following goals:

1. connects ForCEG to CSA at the start-up of ForCEG in case of loading of TBS-IP/TSGMP configuration;
2. disconnects ForCEG from CSA when ForCEG closes;
3. sends periodic messages to CSA: one message per path update (an entry in the AuthTable), or one message contains a batch of all active paths;
    a. When Timer ticks 10sec then search for active GRIs,
    b. take their iDs and build a message for CSA,
    c. and send the update message to CSA.

4.12.5 Advance Power Management (APM) plugin

API:

Function	Called by module
Boolean StartHW();	TVS
Boolean StopHW();	TVS
Boolean GetHWStatus():	TVS

APM plugin (Figure 4.9.(6)) is a software module that interconnects ForCEG to a hardware unit (e.g., APC MasterSwitch) that controls remotely the power socket relays in use by the network processors, routers, etc. The module interconnects to the hardware unit over SNMP protocol.

4.13 FFPF software on network processors: FIX2850

FIX2850 [43] is a software project written in MicroC (ANSI-C language adapted specifically for parallel µEngine programming). It consists of multiple modules, each mapped on one µEngine. We need to build a flexible architecture that (re-)configures the installed LFBs onto the µEngines.

4.13.1 FE-LFB in NPU: mapping of LFBs on hardware cores

At loading time, the control code on XScale performs the LFB mapping onto available hardware cores (µEngines). Some LFBs are mapped, by design, on chosen µEngines, as follows:

- RX_Gig – running on µEngine0 of the IngressNPU;
- Tx_CSIX – running on µEngine7 of the IngressNPU;
- Rx_CSIX – running on µEngine0 of the EgressNPU;
- Tx_Gig – running on µEngine7 of the EgressNPU;

The custom application modules (TB, TS) can be mapped on any other available µEngine (e.g., µEngine1, µEngine2).

When FE is requested to map a certain application (e.g., TS, or TB), it locally instructs the NPU hardware how to map the required application. In other words, it takes the ForCES model of FEObject Location(LFB), as follows:

LFB("Rx")->Attribute("Location")->Value(0);

LFB("TS1")->Attribute("Location")->Value(1);

The LFB mapping onto µEngines is then described in the hardware by means of a description table (DT), as in the following example:

Table 4.1 – Example of LFB Mapping

Ingress NPU		Egress NPU	
LFB	**Location (µEngine)**	**LFB**	**Location (µEngine)**
RX	0	RX_csix	0
TX_csix	7	TX	7
TS_1	1	TS_2	1
TB_1	2	TB_2	2

4.13.2 FE-LFB in NPU: mapping of processing hierarchy

The LFB graph connection (processing hierarchy) is programmable by means of using specific "routing" code at the end of each packet processing dataflow within an LFB. This routing code decides where the current processed packet should go for further processing.

Figure 4.12 shows an example of graph interconnection when using one TB module per NPU. Therefore, the incoming traffic is first split in two (load balancing between those 2 NPUs): one half goes to the ingress TB1 and the other half goes to the egress TB2 via the local TX_csix module.

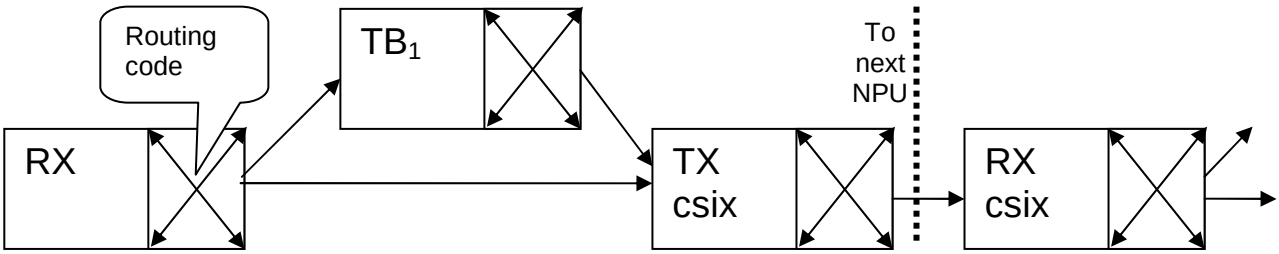

Figure 4.12 Workflow in fast data-path.

The routing code for each LFB in the given example in Figure 6.15 is illustrated in Figure 6.16 and then described as follows:

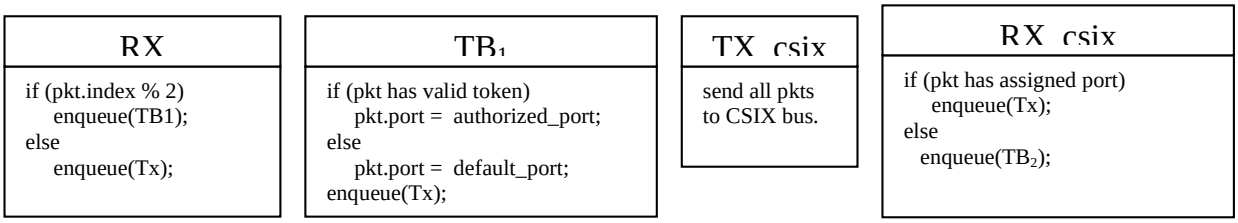

Figure 4.13 Code examples in decisional connectors.

The forwarding packet feature is expressed by the following line code: enqueue(LFBiD). The flexibility of re-mapping of LFBs is given by the routing code itself. However, when using Intel IXP network processors, changing the routing code requires re-compilation of the entire LFB code. In order to make a dynamic re-mapping of LFBs (assuming that all the required LFBs are available and loaded) we use the LFBiD parameters. In other words, we model one LFBs connection in a dependency form:

"LFBiD-src -> connector -> LFBiD-dest". Where the connector is one of the condition branch that when it is true activates the specific connector.

When FE is requested to map a certain application (e.g., TS, or TB), then it locally instructs the NPU hardware how to map the required processing hierarchy. In other words, it sets the ForCES attribute "Connector" on each LFB, as in the following example:

LFB("Rx")->Attribute("Connector")->[Value(0)], Value(1)];

LFB("Rx")->Attribute("Connector")->[Value(1), Value(7)];

LFB("TB1")->Attribute("Connector")->[Value(1), Value(7)];

A connection graph would be described by a table as follows:

Table 4.2 Connection graph example

	LFB_src	connector	LFB_dest
Ingress_TB	0 (RX)	0	1 (TB_1)
	0	1	7 (Tx)
	1 (TB_1)	0	7
	1	1	7
Egress_TB	0	0	7
	0	1	1
	…		
	…		
	…		

This connection graph table will be loaded at start-up by a boot loader code from a shared Description Table (DT) into local registry on each µEngine. DT is filled in by the control code (FE-IXP) running on the XScale control processor.

4.14 LFB attributes

Each LFB has some attributes such as packet counters, diagnostic states, AuthTable. Each item uses certain data types such as DWORD for counters, BYTE for states, array of struct TUPLE for AuthTable. The LFB attributes should be addressable in a logic form such as:

FE(Ingress)->LFB(Rx)->Attribute(Pkts_count);

FE(Ingress)->LFB(TS1)->Attribute(TUPLE);

Example of attributes:

Table 4.3 LFB Attributes

LFB	Attribute	Property	Type
Rx	Packet_Count Connector	Read-Write Read-Write	Integer Array (u8, u8)
Tx	Packet_Count Connector	Read-Write Read-Write	Integer Array (u8, u8)
Rx_CSIX	Packet_Count Connector	Read-Write Read-Write	Integer Array (u8, u8)
Tx_CSIX	Packet_Count Connector	Read-Write Read-Write	Integer Array (u8, u8)
TB	Proc_Packet_Count Connector Tuple	Read-Write Read-Write Read-Write	Integer Array (u8, u8) Array of TUPLE
TS	Proc_Packet_Count Connector	Read-Write Read-Write	Integer Array (u8, u8)

	Tuple Bad_Token_Count	Read-Write Read-Write	Array of TUPLE Integer

An example of complex LFB attributes is an AuthTable used by any TB or TS LFB. Each entry of the AuthTable is a TUPLE structure, defined as follows:

```
struct TUPLE

{
        char[20]        LRI;
        char[20]        LRI_NextDomain;
        char[20]        TokenKey;
        char[20]        TokenKey_NextDomain;
        u8    Port1;
        u8    Port2;
        char[20]        IPpacketMask;
        char[6]    IPsource;
        char[6]    IPdestination;
        char[4]    PortSource;
        char[4]    PortDestination;
        char    Status;
};
```

Summarising, an AuthTable looks as follows.

Table 4.4 AuthTable Example

Entry	0	1	2	3
LRI				
LRI_NextDomain				
TokenKey				
TokenKey_NextDomain				
Port1				
Port2				
IPpacketMask				
IPsource				
IPdestination				
PortSource				
PortDestination				
Status				

4.15 FE-IXP

FE-IXP is a cross-platform module that implements the first two layers of the ForCES (ForCES communication on top of TCP and ForCES modelling). FE-IXP runs as a deamon in userspace of the embedded linux OS.on an NPU hardware. It implements the ForCES standard in a FE that controls all LFBs within the current NPU and connects to the master CE located on a remote host. It mainly consists of the following layers:

- TCP server communication module with external host;
- ForCES model in FE<-> LFBs;
- Low-level connection (hardware abstraction layer – HAL) that is implemented in an external module: μeManager.

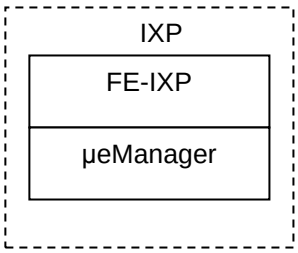

Figure 4.14 IXP

4.15.1 FE

FE is a module that implements ForCES protocol for FE (Forwarding Element), and is implemented in C++ and cross-compiled for ARM target (XScale control processor from IXP). Each IXP runs one FE that connects to its CE (Control Element) parent: ForCEG.

4.15.2 μeManager

μeManager governs the μEngines: it partitions the available DRAM memory, it assigns the memory to LFBs and it maps the LFBs onto μEngines by filling up the descriptor table (DT). μeManager also has to perform some hardware initialization such as MSF Unit. We choose for implementation as a userspace process running on the XScale processor of each IXP2850 NPU. This gives us more flexibility in what we can really do. We are not limited in memory allocation, execution of other process (like initialization scripts, compilers), network access etc. However, all control registers we need to interact with μEngines are available through /dev/mem device which gives direct access to physical memory space (with help of some external libraries for IXPs such as uengine and IXA_SDK). Since most of developers use Intel IXA_SDK for writing μEngine code, we can use this IXA library for uploading of the object code.

API:

Function	Called by module
Bool LFB_MapOnHw(string LFBname, int HW_CoreID);	FE
Bool LFB_MapHierarchy(int HW_CoreID_src, int HW_CoreID_dest,int condition);	FE
Bool LFB_AddPath(struct TUPLE);	FE
Bool LFB_RemovePath(struct TUPLE);	FE
Bool LFB_GetValue(string LFBName, string AttributeName);	FE

One important issue is the interaction with micro-code. Once the application is uploaded onto both NPUs, it needs to synchronize its start-up each other. This is accomplished by a “Start” signal generated by the host’s CE and received (almost) simultaneously on both NPUs from the unique host.

4.15.3 Init_chip

It configures the IXP2800 MSF's SPI-4.2, CSIX and flow control paths, and on the egress NPU, additionally configures the IXF1110 MAC.

ue_ixp_init.[c,h] – implements the HW initialization of each sub-component:

npu_reset(); npu_init_pre_clocks(); npu_init_clocks(); npu_enable_rx_tx(); npu_train_links();

ixf1110_init();

4.15.4 Init_software

It mainly performs the following operations:

- allocates the buffers (PBuf, IBufs for each µEngine, etc.);
- initialises the inter-communication paths (XScale <-> µEngines);
- loads the datapath code onto µEngines;
- starts the hardware units (µEngines);

4.15.5 µEngines => XScale interface

µEngine generates an interrupt for XScale.

Updating data from µEngine to XScale:

µEngines:
Packet comes in & token is valid:
If (Status is 0) then {Status++; sends-to-XScale “Status”;}
Else {Status++; if (Status reaches MAX) then {Status=1; sends-to-XScale “Status”;} }
If update signal received from XScale: Perform the requested operation such as update a certain status.
OBS. MAX gives the dead-time between µEngines –to- XScale messages.

XScale:
When $Timer_1$=10secs:
Check every Status[] from the list and for each Status not 0 => sends up the GMPLS update message;
When $Timer_2$=3secs:
Check every Status ? Status_old:
If Status = = Status_old: sends a time-out message to µEngines to clear the Status;
If Status <> Status_old: Status_old=Status.

4.15.6 XScale => µEngines interface

The interface uses shared memory to send messages on demand with the help of signals. It uses Interthread signal: XScale sends the Interthread_ME signal to a certain thread in a certain µEngine.

For instance, when XScale receives a request to add/remove a path, it first updates the shared memory (SDRAM, or SRAM) and then sends a signal to the µEngine that runs the specific LFB (e.g., TS, or TB).

4.15.7 Mapping of high-level identifiers into low-level hardware

As described before, the authorisation table (AuthTable) contains some large fields such as LRI, IPpacketMask, etc. However, given the hardware constrains in memory, we need to provide to the µEngines only the strict fields needed in runtime. Therefore, the TUPLE is going to be mapped into MINI_TUPLE, as follows:

```
typedef struct _TUPLE
{
  u_int8_t   LRI[20];
  u_int8_t   LRI_NextDomain[20];
  u_int8_t   TokenKey[20];
  u_int8_t   TokenKey_NextDomain[20];
  u_int8_t   Port1;
  u_int8_t   Port2;
  u_int8_t   IPpacketMask[20];
  u_int8_t   IPsource[4];
  u_int8_t   IPdestination[4];
  u_int8_t   PortSource[2];
  u_int8_t   PortDestination[2];
  u_int8_t   Status;
}TUPLE;
```

```
typedef struct _MINI_TUPLE
{
  u_int32_t   iD;
  u_int32_t   TokenKey[5];
  u_int32_t   TokenKeyNextDomain[5];
  u_int32_t   Port1;
  u_int32_t   Port2;
  u_int32_t   Status;
}MINI_TUPLE;
```

The Authorization table is composed of MINI_TUPLE entries and it resides in the slowest memory (and the smallest): LocalMem, or SRAM.

Table 4.5 Authorization Table

Entry	iD [4Bytes]	TokenKey [20Bytes]	TokenKey_NextDomain [20Bytes]	Port1 [4Bytes]	Port2 [4Bytes]	Status [4Bytes]
0	…					
1	…					
2	…					

The mapping of one TUPLE into MINI_TUPLE is done at each AddPath() call. It consists of computing a small identifier iD as follows:

iD = Hash64(IPpacketMask over the IPsource, IPdestination, PortSource, PortDestination);

The other fields remain same: TokenKey, TokenKey_NextDomain, Port1, Port2.
We have chosen to use Hash64 because it is hardware supported and a fast function to provide (cvasi)unique identifier over a large set of data. Note that the same hash function will be performed in datapath for each incoming packet in order to find the unique iD for a searching in the AuthTable in order to retrieve the proper entry.

5 Conclusions and summary

The report described the development of a ForCES Token Based Switch (TBS-IP). The ForCES Token Based Switch implements the token concept of the Token Based Networking architecture and uses the ForCES communication standard to program specialised hardware for network traffic processing.

The report initialy introduced the Token Based Networking (TBN) architecture. The TBN architecture is used to establish lightpaths over multiple network domains. Lightpaths are setup on behalf of authorised applications that need to bypass transit networks. On the one hand, TBN uses a secure signature of pieces of an IP packet as a token that is placed inside the packet to recognise and authenticate traffic. The applications traffic is first tokenised by the TokenBuilder (TB) of a local domain (e.g., a campus network), after which it is enforced by the TokenSwitch (TS) at each inter-domain controller along the end-to-end path. On the other hand, TBN makes use of a separate service and control plane. The control plane consists of a AAA server in the push sequence as explained by the Authorisation Authentication Accounting (AAA) framework (RFC 2904). The AAA server acts as an authority that is responsible for the reservation and provisioning of the end-to-end paths, possibly spanning multiple network domains.

The report also showed the usage of the ForCES protocol to implement our TBS-IP system. The ForCES protocol separates the Forwarding and Control Path by providing an open protocol for communication and control.

In order to make the Token Based Switch more configurable, a ForCES Gateway (ForCEG) architecture was used as an alternative interface for configuring the TBS-IP using specific XML commands instead of using the ForCES protocol directly over Web Services. These XML commands are appropriate to the 3rd party AAA authorisation systems (e.g., GAAA-TK) and they are internally translated into ForCES protocol messages. Web Services provide an open interface over the web, where multiple applications can use them (e.g., other network provisioning systems from different domains).

The report described the implementation of a Token Based Switch over IP, the API's used for inter-communication, and the modelling of the architecture into ForCES LFBs model. Also it provided an overview about the development on the Intel IXP network processors and their usage to build a performant TBS-IP system that works at multi-gigabit speeds.

The next step for the ForCES Token Based Switch is to be compatible with the GAAA Toolkit created and specified in [11], allowing the ForCES Token Based Switch to be accessible by all other GAAA Toolkit enabled

implementations. For example, using GAAA-TK authorisation interface, it will also be possible to seamless interconnect a network domain with TBS-IP technology to a G^2MPLS or ARGON domain.

6 References

[1]. Horzmud Khosravi, and Todd A. Anderson, "Requirements for Separation of IP Control and Forwarding", IETF RFC 3654, November 2003.

[2]. Avri Doria, Fiffi Hellstrand, Kenneth Sundell, and Tom Worster, “General Switch Management Protocol (GSMP) V3”, IETF RFC 3292, June 2002.

[3]. David Durham, Jim Boyle, Ron Cohen, Shai Herzog, Raju Rajan, and Arun Sastry, “The COPS (Common Open Policy Service) Protocol”, IETF RFC 2748, January 2000.

[4]. Avri Doria, Robert Haas, Jamal Hadi Salim, Hormuzd Khosravi, Weiming Wang, “ForCES Protocol Specification”, IETF draft, work in progress, March 2008, <draft-ietf-forces-protocol-14.txt>.

[5]. Joel Halpern, Elen Deleganes, Jamal Hadi Salim, "ForCES Forwarding Element Model", IETF draft, work in progress, July 2008, <draft-ietf-forces-model-12.txt>.

[6]. "The Token Based Switch: Per-Packet Access Authorisation to Optical Shortcuts", by Mihai-Lucian Cristea, Leon Gommans, Li Xu, and Herbert Bos, in Proceedings of IFIP Networking, Atlanta, GA, USA, May 2007.

[7]. "Applications drive secure lightpath creation across heterogeneous domains", by Leon Gommans, Freek Dijkstra, Cees de Laat, Arie Taal, Alfred Wan, Inder Monga, Franco Travostino, in IEEE Communications Magazine 44(3), March 2006.

[8]. “New to SOA and Web Services”, <http://www-106.ibm.com/developerworks/webservices/newto/websvc.html>.

[9]. David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard, “Web Services Architecture”, W3C Working Group, < http://www.w3.org/TR/ws-arch/>, February 2004.

[10]. Evangelos Haleplidis, Robert Haas, Spyros Denazis, Odysseas Koufopavlou, IWAN2005, France, A Web Service-and ForCES-based Programmable Router Architecture, November 2005.

[11]. IST-034115 Phosphorus Deliverable D.4.3.1 “GAAA toolkit pluggable components and XACML policy profile for ONRP”.

Appendix A Acronyms

AAA	**Authentication, Authorisation, Accounting**
AuthZ	**Authorization**
CE	**ForCES Control Element**
CCL	**Command Control Logic**
CFLS	**Current FE & LFB State**
COPS	**Common Open Policy Service Protocol Overview**
DRAGON	**Dynamic Resource Allocation over GMPLS Optical Networks**
EGEE	**Enabling Grids for E-sciencE (European Grid Project)**
FCSTP	**ForCES CE Start/Termination Point**
FE	**ForCES Forwarding Element**
ForCES	**Forwarding and Control Element Separation**
FT	**ForCES Translator**
GAAAPI	**Generic Authentication/Authorisation Application Programming Interface**
GMPLS	**Generalized MPLS (MultiProtocol Label Switching)**
GRI	**Global Reservation Identifier**
LFB	**Logical Function Block**
MAC	**Mandatory Access Control**
MP	**Message Parser**
PEP	**Policy Enforcement Point**
QoS	**Quality of Service**
SAML	**Security Assertion Markup Language**
SNMP	**Simple Network Management Protocol**
SEPR	**Suscribed Events and Pending Responses**
STS	**WS-Trust Secure Token Service**
TBN	**Token Based Networking**
TBS	**Token Based Switch**
TBS-IP	**TBS over IP traffic**
TB	**Token Builder**
TS	**Token Switch**
TVS	**Token Validation Service**
VLSR	**Virtual Label Switching Routing**
XML	**eXtensible Markup Language**

Appendix B XML Ticket and Token Examples

B.1 AuthzTicket example

The listing below provide an example with the Resource element as required for TBS programming generated with the GAAAPI package. The listing also contains comments that explain a suggested mapping to SAML2.0 Authorisation assertion elements, which demonstrates that even for basic AuthZ session data, few extension elements are required for extended security context expression.

```
<AAA:AuthzTicket xmlns:AAA="http://www.aaauthreach.org/ns/#AAA" Issuer="urn:cnl:trust:tickauth:pep"
TicketID="cba06d1a9df148cf4200ef8f3e4fd2b3" SessionID=" f8f3e4fd2b3 df148cf4200">
  <AAA:Decisions><AAA:Decision ResourceID="http://resources.collaboratory.nl/Philips_XPS1"
result="Permit"/>
  </AAA:Decisions>
  <!-- SAML mapping: <AuthorizationDecisionStatement Decision="*" PolicyRef="*" Resource="*"> -->
<AAA:Resources><AAA:Resource ResourceID="http://www.aaaarch.org/TBS/ForceG">
  <AAA:LRI purpose="String">text</AAA:LRI>
  <AAA:TokenKey>String</AAA:TokenKey>
  <AAA:Ports>
    <AAA:port1>String</AAA:port1>
    <AAA:port2>String</AAA:port2>
  </AAA:Ports>
  <AAA:ApplicationFlow>
    <AAA:IPpacketMask>String</AAA:IPpacketMask>
    <AAA:IPsource>String</AAA:IPsource>
    <AAA:IPdestination>String</AAA:IPdestination>
    <AAA:PortSource>String</AAA:PortSource>
    <AAA:PortDestination>String</AAA:PortDestination>
  </AAA:ApplicationFlow>
</AAA:Resource></AAA:Resources>
  <AAA:Actions>
    <AAA:Action>cnl:actions:CtrlInstr</AAA:Action>     <!-- SAML mapping: <Action> -->
    <AAA:Action>cnl:actions:CtrlExper</AAA:Action>
  </AAA:Actions>
  <AAA:Subject Id="subject">
    <AAA:SubjectID>WHO740@users.collaboratory.nl</AAA:SubjectID>
    <!-- SAML mapping: <Subject>/<NameIdentifier> -->
    <AAA:SubjectConfirmationData>
         IGhA11vwa8YQomTgB9Ege9JRNnld84AggaDkOb5WW4U=</AAA:SubjectConfirmationData>
    <!-- SAML mapping:  EXTENDED <SubjectConfirmationData/> -->
    <AAA:Role>analyst</AAA:Role>
    <!-- SAML mapping:
            <Evidence>/<Assertion>/<AttributeStatement>/<Assertion>/<Attribute>/<AttributeValue> -->
    <AAA:SubjectContext>CNL2-XPS1-2005-02-02</AAA:SubjectContext>
    <!-- SAML mapping:
         <Evidence>/<Assertion>/<AttributeStatement>/<Assertion>/<Attribute>/<AttributeValue> -->
```

```
  </AAA:Subject>
  <AAA:Delegation MaxDelegationDepth="3" restriction="subjects">
  <!-- SAML mapping:  LIMITED <AudienceRestrictionCondition> (SAML1.1),
                                  or <ProxyRestriction>/<Audience> (SAML2.0)  -->
    <AAA:DelegationSubjects>
      <AAA:SubjectID>team-member-2</AAA:SubjectID>
      <AAA:SubjectID>team-member-1</AAA:SubjectID>
    </AAA:DelegationSubjects>
  </AAA:Delegation>
  <AAA:Conditions NotBefore="2006-06-08T12:59:29.912Z"
                        NotOnOrAfter="2006-06-09T12:59:29.912Z" renewal="no">
  <!-- SAML mapping: <Conditions NotBefore="*" NotOnOrAfter="*"> -->
    <AAA:ConditionAuthzSession PolicyRef="PolicyRef-GAAA-RBAC-test001" SessionID="JobXPS1-2006-001">
    <!-- SAML mapping: EXTENDED <SAMLConditionAuthzSession PolicyRef="*" SessionID="*"> -->
      <AAA:SessionData>put-session-data-Ctx-here</AAA:SessionData>
      <!-- SAML mapping:  EXTENDED <SessionData/> -->
    </AAA:ConditionAuthzSession>
  </AAA:Conditions>
  <AAA:Obligations>
    <AAA:Obligation>put-policy-obligation(2)-here</AAA:Obligation>
    <!-- SAML mapping:  EXTENDED <Advice>/<PolicyObligation> -->
    <AAA:Obligation>put-policy-obligation(1)-here</AAA:Obligation>
  </AAA:Obligations>
</AAA:AuthzTicket>
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo> ... </ds:SignedInfo>
  <ds:SignatureValue>e4E27kNwEXoVdnXIBpGVjpaBGVY71Nypos...</ds:SignatureValue>
</ds:Signature>
```

Figure B.1. Example of XML based AuthZ ticket format used for TVS programming. (Note. Comments refer to the suggested SAML2.0 mapping)

B.2 XML Token example

XML token is used to provide a reference to already reserved/authorised resource at the access/usage stage of the reserved service or resource. Typically token use is supported by the detailed reservation information stored/cached at the resource.

XML token format uses a special “TVS/TBN” profile of the more general AuthzToken format. Example of the full TVS XML token is shown below:

```
<AAA:AuthzToken xmlns:AAA="http://www.aaauthreach.org/ns/#AAA"
        Issuer="urn:aaa:gaaapi:token:TVS"
        SessionId="a9bcf23e70dc0a0cd992bd24e37404c9e1709afb"
        TokenId="d1384ab54bd464d95549ee65cb172eb7">
    <AAA:TokenValue>ebd93120d4337bc3b959b2053e25ca5271a1c17e</AAA:TokenValue>
    <AAA:Conditions NotBefore="2007-08-12T16:00:29.593Z" NotOnOrAfter="2007-08-13T16:00:29.593Z"/>
</AAA:AuthzToken>
```

where the element <TokenValue> and attributes SessionId and TokenId are mandatory, and the element <Conditions> and attributes Issuer, NotBefore, NotOnOrAfter are optional.

Minimum token format is illustrated in the following example:

```
<AAA:AuthzToken xmlns:AAA="http://www.aaauthreach.org/ns/#AAA"
        SessionId="a9bcf23e70dc0a0cd992bd24e37404c9e1709afb"
        TokenId="d1384ab54bd464d95549ee65cb172eb7">
     <AAA:TokenValue>ebd93120d4337bc3b959b2053e25ca5271a1c17e</AAA:TokenValue>
```

```
</AAA:AuthzToken>
```

Attributes “Issuer” allow for distinguishing different AuthzToken profiles. The TVS profile is identified by the URN "urn:aaa:gaaapi:token:TVS".

Appendix C ForCES model in TBS-IP

The XML for the LFB's of the ForCES based TBS can be found in the ForCES model draft [[5]].

C.1 Schema of RX LFB

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://ietf.org/forces/1.0/lfbmodel:\Projects\Phosphorus\FORCES~1\LFBschemas.xsd"
provides="RxLFB">
    dataTypeDefs>
    <dataTypeDef>
        name>Ports</name>
        synopsis>Values that can be applied to Incoming and Outoging Port for Condition
Forwarding</synopsis>
        atomic>
        <baseType>u8</baseType>
        <rangeRestriction>
            allowedRange max="0" min="9"></allowedRange>
        </rangeRestriction>
        /atomic>
    </dataTypeDef>
    <dataTypeDef>
        name>Connector</name>
        synopsis>A Conditional connector between an incoming port and outgoing port</synopsis>
        struct>
        <element elementID="1">
            name>IncomingPort</name>
            synopsis>The Incoming Port/MicroEngine</synopsis>
            typeRef>Ports</typeRef>
        </element>
        <element elementID="2">
            name>Condition</name>
            synopsis>The condition upon which from 1 will go to 2</synopsis>
            typeRef>u8</typeRef>
        </element>
        <element elementID="3">
            name>OutgoingPort</name>
            synopsis>The Outgoing Port/MicroEngine</synopsis>
            typeRef>Ports</typeRef>
        </element>
        /struct>
    </dataTypeDef>
    /dataTypeDefs>
    LFBClassDefs>
    <LFBClassDef LFBClassID="3">
```

```
        name>RxLFB</name>
        synopsis>The port Rx LFB for the FE - TVS</synopsis>
        version>1.0</version>
        attributes>
        <attribute elementID="1" access="read-write">
            name>Condition_Fwd</name>
            synopsis>An array with condition forwarding</synopsis>
            typeRef>Connector</typeRef>
        </attribute>
        <attribute elementID="2" access="read-reset">
            name>Packet_Count</name>
            synopsis>The number of packets that are coming into the IXP</synopsis>
            typeRef>uint32</typeRef>
        </attribute>
        /attributes>
    </LFBClassDef>
    /LFBClassDefs>
</LFBLibrary>
```

C.2 Schema of Tx LFB

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://ietf.org/forces/1.0/lfbmodel:\Projects\Phosphorus\FORCES~1\LFBschemas.xsd"
provides="TxLFB">
<LFBClassDefs>
    LFBClassDef LFBClassID="4">
    <name>TxLFB</name>
    <synopsis>The port Tx LFB for the FE - TVS</synopsis>
    <version>1.0</version>
    <attributes>
        attribute elementID="1" access="read-reset">
        <name>Packet_Count</name>
        <synopsis>The number of packets that are going out of the IXP</synopsis>
        <typeRef>uint32</typeRef>
        /attribute>
    </attributes>
    /LFBClassDef>
</LFBClassDefs>
</LFBLibrary>
```

C.3 Schema of TB LFB

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://ietf.org/forces/1.0/lfbmodel:\Projects\Phosphorus\FORCES~1\LFBschemas.xsd"
provides="TBLFB">
<dataTypeDefs>
    dataTypeDef>
    <name>Tuple</name>
    <synopsis>A Tuple from the TVS Ticket</synopsis>
<!-- LRI, TokenKey, NewLRI, NewTokenKey, Port1, Port2, IPPacketMask, IPSource, IPDestination,
PortSource, PortDestination, Status -->
    <struct>
        element elementID="1">
        <name>LRI</name>
        <synopsis>Local Reference Identifier</synopsis>
        <typeRef>byte[20]</typeRef>
        /element>
```

```
        element elementID="2">
        <name>TokenKey</name>
        <synopsis>A key used for building the encrypted token</synopsis>
        <typeRef>byte[20]</typeRef>
        /element>
        element elementID="3">
        <name>NewLRI</name>
        <synopsis>A Local Reference Identifier for the next domain</synopsis>
        <typeRef>byte[20]</typeRef>
        /element>
        element elementID="4">
        <name>NewTokenKey</name>
        <synopsis> A key used by the next domain</synopsis>
        <typeRef>byte[20]</typeRef>
        /element>
        element elementID="5">
        <name>Port1</name>
        <synopsis>Port 1 Number</synopsis>
        <typeRef>u8</typeRef>
        /element>
        element elementID="6">
        <name>Port2</name>
        <synopsis>Port 2 Number</synopsis>
        <typeRef>u8</typeRef>
        /element>
        element elementID="7">
        <name>IPPacketMask</name>
        <synopsis>IPPacketMask</synopsis>
        <typeRef>byte[20]</typeRef>
        /element>
        element elementID="8">
        <name>IPSource</name>
        <synopsis>IPSource</synopsis>
        <typeRef>byte[4]</typeRef>
        /element>
        element elementID="9">
        <name>IPDestination</name>
        <synopsis>IPDestination</synopsis>
        <typeRef>byte[4]</typeRef>
        /element>
        element elementID="10">
        <name>PortSource</name>
        <synopsis>Port Source</synopsis>
        <typeRef>byte[2]</typeRef>
        /element>
        element elementID="11">
        <name>PortDestination</name>
        <synopsis>Port Destination</synopsis>
        <typeRef>byte[2]</typeRef>
        /element>
        element elementID="12">
        <name>Status</name>
        <synopsis>A Status</synopsis>
        <typeRef>string</typeRef>
        /element>
    </struct>
    /dataTypeDef>
    dataTypeDef>
    <name>ActiveTuples</name>
    <synopsis>An Array with the Active Tuples</synopsis>
    <array type="variable-size">
        typeRef>ActiveTupleCount</typeRef>
    </array>
    /dataTypeDef>
    <dataTypeDef>
        name>Ports</name>
        synopsis>Values that can be applied to Incoming and Outoging Port for a Connector</synopsis>
        atomic>
        <baseType>u8</baseType>
```

```
        <rangeRestriction>
            allowedRange max="0" min="255"></allowedRange>
        </rangeRestriction>
        /atomic>
    </dataTypeDef>
    <dataTypeDef>
        name>Connector</name>
        synopsis>A Conditional Connector between an incoming port and outgoing port</synopsis>
        struct>
        <element elementID="1">
            name>IncomingPort</name>
            synopsis>The Incoming Port/MicroEngine</synopsis>
            typeRef>Ports</typeRef>
        </element>
        <element elementID="2">
            name>Condition</name>
            synopsis>The condition upon which from 1 will go to 2</synopsis>
            typeRef>u8</typeRef>
        </element>
        <element elementID="3">
            name>OutgoingPort</name>
            synopsis>The Outgoing Port/MicroEngine</synopsis>
            typeRef>Ports</typeRef>
        </element>
        /struct>
    </dataTypeDef>
    <dataTypeDef>
        name>ActiveTupleCount</name>
        synopsis>An Active Tuples with it's counters</synopsis>
        struct>
        <element elementID="1">
            name>ActiveTuplesEntry</name>
            synopsis>An Active Tuples Entry</synopsis>
            typeRef>Tuple</typeRef>
        </element>
        <element elementID="2">
            name>Token_Count</name>
            synopsis>The number of tokens that are coming into the TB</synopsis>
            typeRef>uint32</typeRef>
        </element>
        <element elementID="3">
            name>Bad_Token_Count</name>
            synopsis>The number of bad tokens that have been checked</synopsis>
            typeRef>uint32</typeRef>
        </element>
        /struct>
    </dataTypeDef>
</dataTypeDefs>
<LFBClassDefs>
    LFBClassDef LFBClassID="5">
    <name>TBLfb</name>
    <synopsis>Token Builder LFB</synopsis>
    <version>1.0</version>
    <attributes>
        attribute elementID="1" access="read-write">
        <name>Tuples</name>
        <synopsis>The tuples inside the TB</synopsis>
        <typeRef>ActiveTuples</typeRef>
        /attribute>
        <attribute elementID="2" access="read-write">
            name>Condition_Fwd</name>
            synopsis>An array with condition forwarding</synopsis>
            typeRef>Connector</typeRef>
        </attribute>
        <attribute elementID="3" access="read-reset">
            name>Packet_Count</name>
            synopsis>The number of packets that are coming into the TB module</synopsis>
            typeRef>uint32</typeRef>
        </attribute>
```

```
    </attributes>
    /LFBClassDef>
</LFBClassDefs>
</LFBLibrary>
```

C.4 Schema of TS LFB

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://ietf.org/forces/1.0/lfbmodel:\Projects\Phosphorus\FORCES~1\LFBschemas.xsd"
provides="TSLFB">
    dataTypeDefs>
    <dataTypeDef>
        name>Tuple</name>
        synopsis>A Tuple from the TVS Ticket</synopsis>
        !-- LRI, TokenKey, NewLRI, NewTokenKey, Port1, Port2, IPPacketMask, IPSource, IPDestination,
PortSource, PortDestination, Status -->
        struct>
        <element elementID="1">
            name>LRI</name>
            synopsis>Local Reference Identifier</synopsis>
            typeRef>byte[20]</typeRef>
        </element>
        <element elementID="2">
            name>TokenKey</name>
            synopsis>A key used for building the encrypted token</synopsis>
            typeRef>byte[20]</typeRef>
        </element>
        <element elementID="3">
            name>NewLRI</name>
            synopsis>A Local Reference Identifier for the next domain</synopsis>
            typeRef>byte[20]</typeRef>
        </element>
        <element elementID="4">
            name>NewTokenKey</name>
            synopsis>A key used by the next domain</synopsis>
            typeRef>byte[20]</typeRef>
        </element>
        <element elementID="5">
            name>Port1</name>
            synopsis>Port 1 Number</synopsis>
            typeRef>u8</typeRef>
        </element>
        <element elementID="6">
            name>Port2</name>
            synopsis>Port 2 Number</synopsis>
            typeRef>u8</typeRef>
        </element>
        <element elementID="7">
            name>IPPacketMask</name>
            synopsis>IPPacketMask</synopsis>
            typeRef>byte[20]</typeRef>
        </element>
        <element elementID="8">
            name>IPSource</name>
            synopsis>IPSource</synopsis>
            typeRef>byte[4]</typeRef>
        </element>
        <element elementID="9">
            name>IPDestination</name>
            synopsis>IPDestination</synopsis>
            typeRef>byte[4]</typeRef>
        </element>
        <element elementID="10">
```

```
            name>PortSource</name>
            synopsis>Port Source</synopsis>
            typeRef>byte[2]</typeRef>
        </element>
        <element elementID="11">
            name>PortDestination</name>
            synopsis>Port Destination</synopsis>
            typeRef>byte[2]</typeRef>
        </element>
        <element elementID="12">
            name>Status</name>
            synopsis>A Status</synopsis>
            typeRef>string</typeRef>
        </element>
        /struct>
    </dataTypeDef>
    <dataTypeDef>
        name>ActiveTuples</name>
        synopsis>An Array with the Active Tuples and their counters</synopsis>
        array type="variable-size">
        <typeRef>ActiveTupleCount</typeRef>
        /array>
    </dataTypeDef>
    <dataTypeDef>
        name>ActiveTupleCount</name>
        synopsis>An Active Tuples with it's counters</synopsis>
        struct>
        <element elementID="1">
            name>ActiveTuplesEntry</name>
            synopsis>An Active Tuples Entry</synopsis>
            typeRef>Tuple</typeRef>
        </element>
        <element elementID="2">
            name>Token_Count</name>
            synopsis>The number of tokens that are coming into the TS</synopsis>
            typeRef>uint32</typeRef>
        </element>
        <element elementID="3">
            name>Bad_Token_Count</name>
            synopsis>The number of bad tokens that have been checked</synopsis>
            typeRef>uint32</typeRef>
        </element>
        /struct>
    </dataTypeDef>
    <dataTypeDef>
        name>Ports</name>
        synopsis>Values that can be applied to Incoming and Outoging Port for a Connector</synopsis>
        atomic>
        <baseType>u8</baseType>
        <rangeRestriction>
            allowedRange max="0" min="255"></allowedRange>
        </rangeRestriction>
        /atomic>
    </dataTypeDef>
    <dataTypeDef>
        name>Connector</name>
        synopsis>A Conditional Connector between an incoming port and outgoing port</synopsis>
        struct>
        <element elementID="1">
            name>IncomingPort</name>
            synopsis>The Incoming Port/MicroEngine</synopsis>
            typeRef>Ports</typeRef>
        </element>
        <element elementID="2">
            name>Condition</name>
            synopsis>The condition upon which from 1 will go to 2</synopsis>
            typeRef>u8</typeRef>
        </element>
        <element elementID="3">
```

```
            name>OutgoingPort</name>
            synopsis>The Outgoing Port/MicroEngine</synopsis>
            typeRef>Ports</typeRef>
        </element>
        /struct>
    </dataTypeDef>
    /dataTypeDefs>
    LFBClassDefs>
    <LFBClassDef LFBClassID="5">
        name>TSLfb</name>
        synopsis>Token Switch LFB</synopsis>
        version>1.0</version>
        attributes>
        <attribute elementID="1" access="read-write">
            name>Tuples</name>
            synopsis>The tuples inside the TB</synopsis>
            typeRef>ActiveTuples</typeRef>
        </attribute>
        <attribute elementID="2" access="read-write">
            name>Condition_Fwd</name>
            synopsis>An array with condition forwarding</synopsis>
            typeRef>Connector</typeRef>
        </attribute>
        /attributes>
        events baseID="1">
        <event eventID="1">
            name>BadCountSurpassed</name>
            synopsis>If the Bad Count is surpassed the LFB will send a message to the CE</synopsis>
            eventTarget>
            <eventField>Tuples</eventField>
            <eventSubscript>_TupleEntry_</eventSubscript>
            <eventField>BadTokenCount</eventField>
            /eventTarget>
            eventCreated>
            <eventGreaterThan>255</eventGreaterThan>
            /eventCreated>
            eventReports>
            <eventReport>
                eventField>Tuples</eventField>
                eventSubscript>_TupleEntry_</eventSubscript>
                eventField>ActiveTuplesArray</eventField>
                eventField>LRI</eventField>
            </eventReport>
            /eventReports>
        </event>
        /events>
    </LFBClassDef>
    /LFBClassDefs>
</LFBLibrary>
```