ICT-257422

CHANGE

CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions

Specific Targeted Research Project

FP7 ICT Objective 1.1 The Network of the Future

D4.4 – Network architecture: inter-platform communication software

Due date of deliverable: October 1, 2012

Actual submission date: October 26, 2012

Start date of project	October 1, 2010
Duration	36 months
Lead contractor for this deliverable	University Politehnica of Bucharest
Version	1.0, October 26, 2012
Confidentiality status	Public

Abstract

Flow processing has become an integral part of the Internet today, but it is reserved only to on-path entities deploying invisible middleboxes. The CHANGE vision calls for explicitly architecting middleboxes into the Internet by creating general-purpose software platforms that any party - including end-users - can use to perform flow processing.

This document presents the ongoing implementation of the CHANGE architecture. It discusses the client API as well as an initial prototype of the CHANGE architecture that allows users to specify and instantiate basic flow processing functionality. We also describe the software architecture of the signaling components that will be used to control flow processing functionalities among CHANGE platforms deployed in different administrative domains.

Target Audience

Network experts and researchers. This document is public, but the accompanying source code is restricted to the Commission Services and groups specified by the consortium.

Impressum

Full project title	CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions
Title of the workpackage	D4.4 – Network architecture: inter-platform communication software
Editor	Costin Raiciu, University Politehnica of Bucharest
Project Co-ordinator	Adam Kapovits, Eurescom
Technical Manager	Felipe Huici, NEC

This project is co-funded by the European Union through the ICT programme under FP7.

Executive Summary

Flow processing has become an integral part of the Internet today, but it is reserved only to on-path entities deploying invisible middleboxes. The CHANGE vision calls for explicitly architecting middleboxes into the Internet by creating general-purpose software platforms that any party - including end-users - can use to perform flow processing.

This document reports on the state of the implementation of the CHANGE network architecture, focusing on the inter-platform communication software. The document begins by presenting the API offered to the CHANGE client in Chapter 2, and then discusses our prototype implementation that allows us to instantiate basic flow processing functionality in Chapter 3. We are working towards implementing an extensible CHANGE architecture. We also report our work on building software components required to support flow processing among different platforms in Section 4.

List of Authors

Authors	Costin Raiciu (PUB), Francesco Salvestrini (NXW), Alessandro Canessa (NXW)
Participants	University Politehnica of Bucharest (PUB), Nextworks s.r.l. (NXW)
Work-package	WP4 – Network Architecture Implementation
Confidentiality	PUBLIC (PU)
Nature	Report (R)
Version	1.0
Total number of pages	37

Contents

List of Figures

Acronyms

CC	Control Channel.
CLI	Command Line Interface.
DCCP	Datagram Congestion Control Protocol.
DNS	Domain Name System.
E2E	End to End.
FPR	Flow Processing Route.
FPRR	Flow Processing Record Route.
FSM	Finite State Machine.
FTP	File Transfer Protocol.
GIST	General Internet Signaling Protocol.
IP	Internet Protocol.
NAT	Network Address Translation.
NHOP	Next HOP.
NSIS	Next Step in Signaling.
NSLP	NSIS Signaling Layer Protocol.
NTLP	NSIS Transport Layer Protocol.
PDU	Protocol Data Unit.
PHOP	Previous HOP.
PM	Processing Module.
QoS	Quality of Service.
RSVP	Resource ReSerVation Protocol.
RTSP	Real Time Streaming Protocol.

SCTP	Stream Control Transmission Protocol.
SIP	Service Initiation Protocol.
SSH	Secure Shell.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
XML	Extensible Markup Language.

1 Introduction

Flow processing has become an integral part of the Internet today, but it is reserved only to on-path entities deploying invisible middleboxes. The CHANGE vision calls for explicitly architecting middleboxes into the Internet by creating general-purpose software platforms that any party - including end-users - can use to perform flow processing.

This document describes the high level details of the CHANGE network architecture implementation prototype and refers to architectural and functional concepts provided by specific documents delivered by WP2 (D2.4), WP3 (D3.1) and WP4 (D4.1, D4.2 and D4.3).

The document begins with a discussion of the client API in Chapter 2. Next, we describe an initial prototype of the CHANGE architecture that allows users to specify and instantiate basic flow processing functionality in Chapter 3. In Chapter 4 we discuss the the implementation of a framework for signaling applications that is adaptable to both centralized and distributed CHANGE platform deployments. Both the network control/management plane architecture and the signaling control part in particular can be deployed either in a centralized or distributed model. In general, both these models provide the same functionalities for signaling coordination, flow composition/routing and resource configuration; they can be applied in CHANGE with minimum effort by extending the current prototype in few key points.

The architectural design pivots around the Flow Processing Route (FPR) concept, as described in D4.2. The Flow Processing Route encapsulates a concatenation of hops which constitutes an explicitly routed path. Using the FPR, the paths taken by flows can be administratively pre-determined, automatically computed by a decision entity or demanded to a hop-by-hop decision. This concept represents the foundation for an extensible signaling framework that could be easily adapted to different deployments and routing models. As described in D4.2, the current implementation supports the source-routed approach. The architectural framework and the software design however allow to easily extend the implementation in order to support the hop-by-hop approach.

The software presented in this deliverable is modular and its architecture allows flexible integration and incremental development. These features are fundamental in CHANGE in order to allow the proof of all the concepts presented in WP2 deliverables while developing the network architecture presented in WP4. The flow processing management presented in section 2 and the signaling framework components presented in section 4 could be imagined as different software packages loosely interacting at the moment. All the functional modules and components (i.e. the signaling daemons, the service processing, the various utilities, the building framework etc.) will however converge into software packages implementing the CHANGE network architecture in the final implementation deliverable (D4.5).

The CHANGE Network Architecture will be finalized in the next months. Integration or testing activities might introduce upgrades, in that case the significant evolutions updates to this document will be included in the planned deliverable D4.5.

2 The CHANGE User Interface

In this section we describe the CHANGE client interface and its ongoing implementation. We use the term client in a broad sense to denote an entity that requests processing to be performed on its behalf on a CHANGE platform, and it could include: enterprises, Internet Service Providers, and even end-users.

In this project we assume that there is already a trust-relationship between the client and the CHANGE provider which is the entity running CHANGE platforms. One way to build such relationship would be to use the Amazon EC2 model where users register with EC2 by providing their credit card details. The users then receive a public/private keypair and a certificate from EC2 which they use to authenticate with EC2 when making requests. In CHANGE we assume that the user has completed this initial step and that the provider and the client can perform mutual authentication if needed. We also assume that billing is done outside the architecture, using the statistics provided. Once the trust relationship is established between the CHANGE provider and the client, billing is easy to support.

Clients must first install CHANGE software on their machines to be able to interact with the architecture. This software is essentially a set of tools that allow the client to instantiate and terminate processing, and request flow processing statistics. The software uses the existing authentication mechanisms (e.g. client's and provider's public/private keypair). This ensures that users cannot instantiate processing unless they have established a trust relationship with a CHANGE provider.

Although it is possible to run individual commands, clients will typically write configuration scripts to instruct the software to perform flow processing. The scripts are then interpreted by the CHANGE software and executed. The remainder of this section describes the primitives the client can use in such scripts.

To instantiate flow processing, the CHANGE client must first select one or a few CHANGE platforms it wants to run processing on. Ideally, the user should be able to discover "on-path" platforms for its traffic, but in the first phase of deployment it is highly unlikely that CHANGE platform deployments will be dense enough to cover a meaningful fraction of flows. Even worse, it is highly unlikely they will be "on-path" for any traffic - traffic will have to be steered explicitly into the platform.

That is why, the CHANGE providers will initially provide simple primitives that allow the user to locate a platform nearby a point of interest. Each primitive returns a list of platforms along with their characteristics (location, type, interfaces, etc) and unique identifiers which will be used to instantiate processing:

- FINDPLATFORM (COUNT) instructs the CHANGE provider to return N different platforms (if possible). The provider has the freedom to choose the appropriate platforms based on availability, etc.

- FINDPLATFORMCLOSETOIP(ADDRESS) instructs the CHANGE provider to return the available platform closest to the provided address. This could be implemented by running pings from all platforms, or by using geo-location.

- FINDPLATFORMCLOSETOADDRESS(COORDINATES) instructs the provider to find the geographically closest platform to the specified GPS location. This could be used to ensure that platforms are within a desired geographical region (e.g. country).

- PLATFORM ID=0 NAME="OMU.CS.PUB.RO" defines a platform explicitly by providing a name for it and giving it a locally unique identifier. This identifier will be used later to name platforms concisely when instantiating flow processing. Another way to define platforms is by using the discovery API mentioned before. Explicit definitions are useful to allow the user to bypass the discovery service if needed.

- INTERFACE ID=0 PLATFORM_ID=0 NAME="ETH0" TYPE="REAL" defines an interface on the platform identified with ID. Interfaces can be real or virtual, and are given an identifier that must be unique on the specified platform.

To attract traffic to a platform, the client has two alternatives: using routing or using DNS. The related primitives are:

- ATTRACT_DNS NAMESERVER="141.85.37.200" DOMAIN="CHANGE.RO" KEY="DNSSEC.PRIVATE.KEY" IP="10.0.0.5" allows the client to update the ip associated to the given domain to an explicit IP or the the address of an interface on a CHANGE platform specified by if="X:Y“). Note that the nameserver could run either on a CHANGE platform or at any other site. As this command effectively modifies DNS zone in order to attract traffic to a CHANGE platform, we require for security that needs that the client and nameserver have previously shared a DNSSEC key which guarantees that the client has proper rights to modify the zone.

- ATTRACT_ROUTE PLATFORM=ID PREFIX=A.B.C.D/LENGTH KEY="RPKI.PRIVATE.KEY" CERTIFICATE="RPKI.CERTIFICATE" This instructs a CHANGE platform that implements route advertisements on behalf of customers to advertise a given prefix. To ensure security we use the RPKI infrastructure.

Next we review a list of primitives that are available to the user and that allow it to instantiate flow processing. We distinguish two types of commands: commands that can create processing modules and commands that allow chaining of processing modules and platforms into end-to-end flow processing.
To create a processing module, the user has available the following command:

- PM ID=10 PLATFORM_ID=0 SPEC="$CFG" IN=2 OUT=3 defines a processing module with identifier 10 on platform 0. Packets will go into the processing module on interface 2 and leave the processing module on interface 3. The specification defines the type of processing module to instantiate. In the example above it takes the value of variable CFG, which could be defined as follows:

```
let cfg = {
        BASESYS=xen.clickos

        ft  :: FromNet(``in'')
        tt  :: ToNet(``out'')
        mirror1 :: Mirror('')

        ft -> [0]mirror1[0] -> tt
    }
```

The following primitives allow "plumbing" of different processing modules residing on possibly different platforms into one unified flow processing. The primitives refer to platforms and interfaces by using unique identifiers. To uniquely specify an interface we use the notation X:Y where X denotes the platform identifier and Y denotes the interface identifier.

- FLOW NAME="A" IF=0:1 FILTER="-S 192.168.1.1" A flow can be "created" by specifying a platform, an interface (possibly virtual), and optionally a filter. The filter above is given in *iptables* format. Once a flow is defined, the user can specify flow processing for it.

- REROUTE [FLOW="A"|FROM=0:1] TO=1:0 INVARIANT="MODIFY, ORIGINATE, READ" reroutes (existing) flow A to the interface given in the TO parameter. Instead of "flow", the user can alternatively provide a "from" parameter describing the source interface of the traffic. Rerouting means transporting the flow's packets to the specified interface and will be implemented by using some form of tunneling. The invariant specifies what type of guarantees the user wants for the tunnel. MODIFY disallows changes to the flow's packets, ORIGINATE stops parties en-route from inserting packets into the flow. Finally, READ prevents other parties from reading the packet contents.

- REDIRECT [FLOW="A"|FROM=X:Y] TO=2:1 INVARIANT="MODIFY, ORIGINATE, READ" changes the destination address of (existing) flow A to the interface specified, thereby redirecting the flow to that interface. The user can specify an invariant in this case only if the destination is another CHANGE platform or a CHANGE user.

- FORWARD [FLOW="A"|FROM=X:Y] TO=2:2 simply forwards the specified flow to an interface on the *same* platform. Invariants are not needed in this case.

When parsing the script, the client software remembers for every flow the last "position" of the flow. In the example above, the flow is first defined by the FLOW command and thus resides on interface "0:1". Next, the flow is REROUTED from platform "0" to platform 1, interface 0. The redirect takes the flow from "1:0" to "2:1", on platform 2. Finally, the flow is forwarded onto interface "2:2".

Since every processing module has incoming and outgoing interface(s), the flow commands do not need to explicitly mention processing modules. It is very easy to get traffic in and out of a processing module: all the user needs to do is to specify the identifier of the processing module's incoming and outgoing interfaces.

CHANGE starts instantiating a script either when it has parsed all of it or the user forces instantiation with the command RUN that takes no parameters.

Once a flow processing script is instantiated, the software returns a unique identifier for that script as well as for any of the actions of the script that have resulted in CHANGE platform actions, including flow definitions. Using these identifiers the client can stop flow processing by using the primitive CANCEL(ID).

Once the processing is running, the user can access basic statistics by running the STATS(ID) command where id specifies a platform id and an interface/processing module id. The return value is a string containing an ATTR=VALUE pair on every line. The possible values for ATTR might include packets and bytes processed, packets dropped, etc.

3 Implementation of the CHANGE User Interface

We have implemented a CHANGE client software that provides the interface described in the previous chapter. The code supports two distinct functionalities: parsing client scripts and instantiating them. These were implemented in a frontend and backend respectively to allow better extensibility and reusability while proofing the project concepts and implementing the final network architecture. The client high level structure is depicted in Fig. 3.

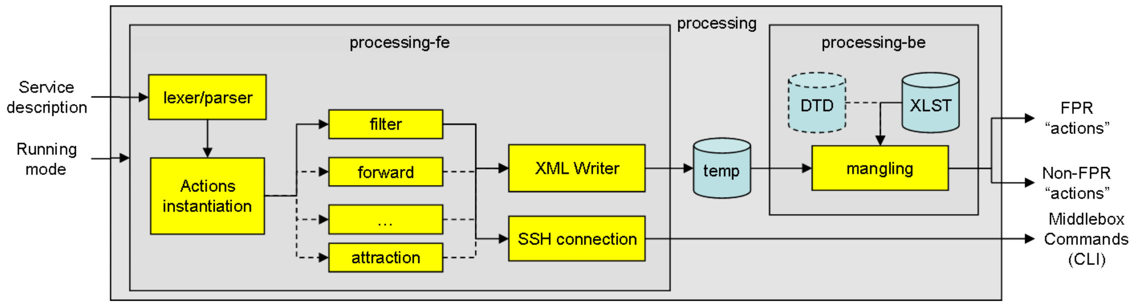

Figure 3.1: CHANGE service processing

The *frontend* parses the service description provided by the client and is coded in C++. It provides feedback on syntax or parameter errors and populates the necessary data structures needed to instantiate flow processing. The frontend relies on *flex* to tokenize the input and on *bison*(a version of *yacc*) to parse it. We defined a simple grammar that allows parsing CHANGE configuration files.

The frontend runs as follows. As the rules in the grammar are reduced, we create objects describing the requested actions by using the appropriate class. Example classes include processing module, forward, redirect, reroute, and so forth. These objects are then added to global object lists (e.g. the list of platforms, interfaces, processing modules). If a primitive requires instantiation its associated class must derive from an abstract class called INSTANTIATE and implement two methods: INSTANTIATE and GETPRIORITY. When parsed, the objects resulting will be added to a list of objects needing instantiation.

Once the entire input file has been parsed, or when the user calls RUN, the job of the frontend is to start the instantiation of the commands. The list of objects to instantiate is sorted according to each object's priority - this ensures that dependencies between instantiations are satisfied (e.g. a tunnel is created before traffic is directed into the tunnel). Next, the frontend invokes the backend to instantiate the processing.

The backend has been designed to be easily replaceable as the CHANGE platform implementation evolves. Currently we have implemented two backends that can be selected using a runtime switch.

If the XML backend is used, the frontend simply outputs the desired actions in a machine readable form (XML). These can then be read by any other backend written in any language to implement the desired actions. We have currently started implementing a backend that translates the desired actions and signals them to the CHANGE platforms. This backend is not yet functional.

Our second backend runs in the same process as the frontend and instantiates the actions as soon as the frontend starts calling instantiate on each command of the user. As an initial prototype, we use SSH with public-key authentication to connect to the processing platforms and run SSH commands locally to interact with the platform. Allowing SSH access to CHANGE platforms might create security issues in the long run; we have chosen it because it allows quick short-term prototyping.

We have used SSH as well as the primitives offered by the CHANGE platforms to create processing modules to implement support for most of the CHANGE user interface. We currently support most of the flow processing primitives, we support the instantiation of ClickOS primitives as well as defining arbitrary middleboxes and platforms. Traffic attraction is supported via DNS: the implementation checks the DNSSEC credentials of the user and updates the nameserver (which we assume for now is the popular *bind* nameserver) accordingly. The code still offers no support for defining flows, routing attraction, platform discovery, retrieving statistics, as well as stopping flow processing (we do this manually for the time being). Further, the code does not yet implement invariants, leaving the burden of specify the tunnel type to the user - here we will implement adaptive algorithms that decide the best way of implementing a given invariant between two given CHANGE platforms. Support for all these missing features will be added in the next code release.

3.1 Example configuration file and execution

The example below shows a flow processing configuration where traffic is filtered based on the source address on platform 0 and then it is sent to the destination (platform 1).

```
#Generic config specifying keypair and default username for authentication
CONFIG username=costin password=bla public_key_file="id_rsa.pub"
private_key_file="id_rsa"

PLATFORM id=0 name=141.85.37.150 sys_type=linux
INTERFACE id=0 platform_id=0 name=eth0

PLATFORM id=1 name=192.168.1.4 sys_type=macos username=localadmin
INTERFACE id=0 platform_id=1 name=en1

ATTRACT_DNS nameserver="141.85.37.200" domain="rinciog.ro" key="dnssec.priv"
ip="141.85.37.150" duration=3600

#this creates a TCP based tunnel between the two interfaces; the tcp client
#is 1:1 because reverse_direction is true
REROUTE from=0:1 to=1:1 proto=tcp reverse_direction=true
```

```
#this filters traffic and puts it into the tunnel
FORWARD from=0:0 to=0:1 filter="-s 193.63.58.134"
```

Say we have saved the script above as “simple.chg”. After the code has been compiled (see appendix B for details) we can run the script above by typing in the command line:

```
./processing-fe -c simple.chg
```

To see more options run:

```
./processing-fe -c simple.chg
```

3.2 Sources overview

The sources are organized in the following directory structure.

```
/ ...........................................................................
├── bootstrap ........ The script used to bootstrap the build framework from the repository
├── configure ............... The script that must be used to configure the source package
├── configure.ac ........................................ the “configure” script source file
├── src ................................................... The main source code directory
│   ├── be ...................................... The backend source code (processing-be)
│   │   └── processing-be.in ................................. The backend main program
│   │       └── xml ......................................... XML processing XSLT data files
│   ├── fe ..................................... The frontend source code (processing-fe)
│   ├── libs ................................... The Frontend/Backend libraries directory
│   └── scripts ............ A directory containing scripts used by the backend/frontend
├── tests ................................................ A directory containing test files
├── build-aux ..................... A directory containing build related files and scripts
└── m4 ........................................... A directory containing build related files
```

Figure 3.2: Processing package source code directory structure

4 The inter-platform signaling framework software

This section presents a high-level design and implementation overview of the main functional entities described in D4.2: the *Service Manager*, *the Signaling Manager* and the *Service Broker*. These entities share a layered software design that splits their "core" logic from the transport issues in order to minimize the implementation efforts by reusing common components as much as possible. Therefore all the elements described in the following sections share a common lower layer block, the *transportd*, and implement their logic in the upper layer.

For debugging purposes all the entities available in the prototype implementation support two different "transport" modes to communicate each other: via the aforementioned *transportd* and through a "simulated" transport realized with traditional TCP sockets. That twofold approach has been utilized in order to parallelize the development and testing processes (particularly for the *Signaling Manager* Protocol FSM debugging). The final implementation that will be released in D4.5 will obsolete the simulated transport, preferring the transport functionalities provided by the final implementation of the *transportd*.

This section also describes the status of the implementation, which is on-going. We do not discuss the parts of the CHANGE network architecture still under development. These will be completed and tested in the next months, before the planned D4.5 final implementation release. The prototype implementation already provides the development framework, most of the libraries and the reusable parts used by the components under development.

4.1 transportd

The *transportd* represents a common lower layer that will provide the transport related services to the CHANGE signaling entities described in the following sections. This lower layer is expected to realize the establishment and maintenance of the signaling adjacencies, the signaling messages exchange and provide the *Control Channels* functionalities described in D4.2.

We have decided to adopt the NSIS framework in order to ease the implementation of the lower signaling protocol mechanics. Following this approach, the *transportd* implements both the NSIS NTLP and NSLP layers for the CHANGE network entities included in this deliverable (i.e. *Signaling Manager*, *Service Manager* and *Service Broker*).

The *transportd* may be imagined as composed by two logical blocks: the *vNSLP* and the *nsis-ka*, as depicted in Fig. 4.1. The former is the CHANGE specific NSLP implementation while the latter is NTLP part of the Karlsruher Institut für Technologie implementation of the NSIS framework [2].

The vNSLP interacts with the signaling application (i.e. the *Service Broker*, the *Signaling Manager* and the *Service Manager*) and with the NTLP layer across the following interfaces:

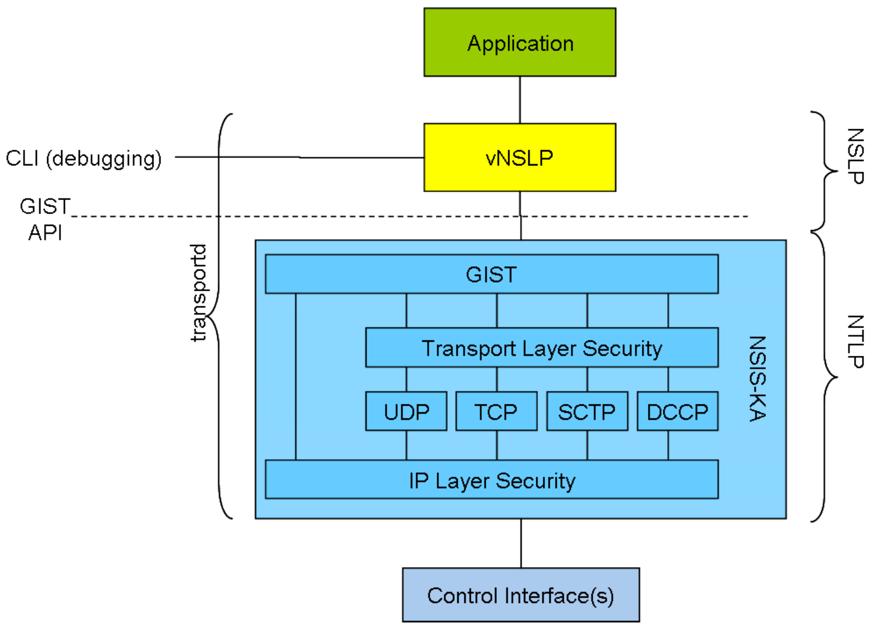

Figure 4.1: The structure of transportd

- The northbound interface (towards the signaling application) mainly provides the *Control Channel* management functionalities (e.g. creation, deletion, configuration and messages delivery with the peering entities).
- The southbound interface (towards the NTLP layer) uses the GIST API provided by the *nsis-ka* framework to interact with the NTLP, as described in [7].

The vNSLP also provides an interactive CLI (mainly used for debugging purposes at the moment). This interface might be kept in the final implementation as an enabler for partner's application integration.
Since the *transportd* and all other CHANGE signaling framework components described in following sections are implemented in different languages, each component higher layer must interface with the *transportd*. To avoid complexity and reduce the implementation efforts, the *transportd* northbound interface might be automatically generated from the *vNSLP* interface headers using *SWIG* [1], leaving to the component's upper layer the specific parts of the interface implementation.

4.2 Signaling Manager

The Signaling Manager runs on every CHANGE platform and is in charge of implementing the platform-to-platform signaling. The *Signaling Manager* architecture, as all other CHANGE signaling framework components presented in this document, is split in two halves: the upper part (the *signald*) handles the signaling logic while the lower part (the *transportd*) handles the transport related mechanisms, as shown in Fig. 4.2.

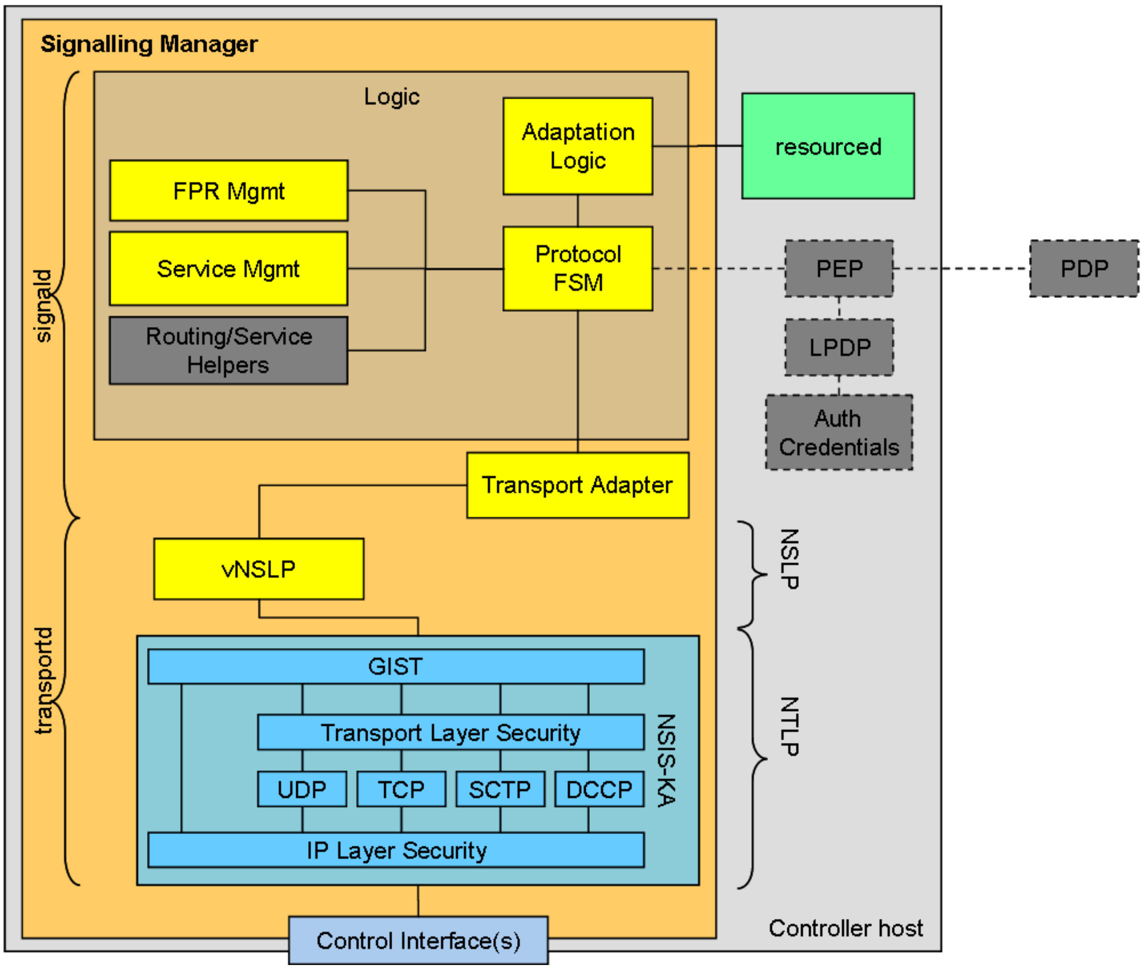

Figure 4.2: Signaling Manager

The *signald* architecture is composed by the following logical blocks:

- The manager *logic*:
  - Adaptation logic: The *Adaptation logic* block interacts with the FlowStream platform *resourced* daemon by adapting the *Protocol FSM* resource allocation requests into the *resourced* interface format and parsing the related *resourced* replies.
  - Protocol FSM: The *Protocol FSM* implements the signaling protocol FSM detailed in section 4.2.1 and interacts with all the *logic* functional components.
  - FPR Management: The *FPR Management* block mainly provides the FPR management functionalities to the *logic* components, e.g. it detects if the current platform is the first/last one in

a FPR, detects if the current platform is the first/last in the domain, mangles the FPR objects by (re)moving the current platform in order to provide a message that will be sent towards the next peering entity.

  – Service Management: The *Service Management* handles the message exchanges with the *Service Manager* and manages the *Signaling Manager* data-model mainly maintaining the information related to the various (in-progress or already instantiated) services, e.g. it performs controls on the correct service instantiation, the service information place-holders creation/removal.

  – Routing/Service helpers: The *Routing/Service Helpers* block handles the FPR completion, routing and service composition for the *logic* as described in D4.2.

- Transport Adapter: The *Transport Adapter* handles the serialization/deserialization of the messages to/from the *transportd*.

4.2.1 Protocol FSM

The Protocol FSM is described in Fig. 4.2. The following sections briefly describe its states and events.

4.2.1.1 FSM States

- *START*: This is the initial state the FSM enters once it is instantiated and awaits to become idle after performing few initialization steps.

- *IDLE*: The FSM waits for the arrival of new messages related events.

- *MSG_SSREQ*: An FSM instance enters the *MSG_SSREQ* state after receiving a *Service Setup Request* message. After processing the FPR object (e.g. to check if it is *exact/sparse* and/or *strict/loose*), the NHOP and PHOP are processed and a resource reservation request to the platform's *resourced* is performed.

- *ROUTING_SERVICE_HELPER*: The FSM waits for the reply from the *Routing & Service Helpers* about a previously issued FPR completion request (i.e. the FPR was *sparse* and/or *loose*).

- *SDRESP_ERR*: The FSM enters this state upon the detection of errors regarding the reception/processing of a Service Deletion Request message. In this state the previous operations performed for the service deletion cannot be revoked and the FSM moves towards the *SERVICE_DOWN* state

- *SSREQ_ERR*: The FSM enters this state whenever a *Service Setup Request* reception/processing error occours. The FSM releases all the previously allocated resources.

- *RES_LOCKED*: The FSM verifies if the current platform is the first or the last one involved into the service. In the former case a *Service Setup Request* is propagated towards the NHOP while in the latter a *Service Setup Allocation* is propagated towards the PHOP.

- *RES_ERR*: A resource reservation procedure (initiated towards the *resourced*) has been unsuccessful. The FSM releases all the previously allocated resources.

- *ASSOC_ERR*: The FSM enters this state when the Service-ID and the Reservation-ID received from the *resourced* cannot be bind together. The FSM releases all the previously allocated resources.

- *MSG_SSALLOC*: The FSM enters this state upon the reception of a *Service Setup Allocation*. An allocation request is sent to the *resourced* and its reply handled accordingly. If the current platform is the first platform involved into the service, the FSM generates a *Service Setup Confirmation* that will be sent towards the NHOP. Otherwise the *Service Setup Allocation* is modified and forwarded towards the PHOP.

- *SSALLOC_ERR*: The FSM enters this state upon the detection of errors regarding the reception/processing of a Service Setup Allocation message. The FSM releases all the previously allocated resources.

- *MSG_SSCONF*: The FSM enters this state upon the reception of a *Service Setup Confirmation* message. If the current platform is not the last platform involved into the service, the received message is modified and then forwarded to the NHOP. Otherwise two different behaviors may happen, depending on the signaling passes involved into the service setup: in the 3-passes case the *ev_ServiceUP* is generated while in the 4-passes case a *Service Setup ACK* message is sent towards the PHOP.

- *SSCONF_ERR*: The FSM enters this state upon the detection of errors regarding the reception/processing of a *Service Setup Confirmation* message. The FSM releases all the previously allocated resources.

- *MSG_SSACK*: The FSM enters this state upon the reception of a *Service Setup ACK* message. If the current platform is the first platform involved into the service, the *ev_ServiceUP* event is generated. Otherwise, the *Service Setup ACK* message is modified and the forwarded towards the PHOP.

- *ACK_ERR*: The FSM enters this state if the reception/processing of a *Service Setup ACK* message is unsuccessful. The FSM releases all the previously allocated resources.

- *SERVICE_UP*: The FSM enters this state if the service has been instantiated successfully.

- *MSG_SDREQ*: The FSM enters this state from the *SERVICE_UP* state upon the reception of a *Service Deletion Request*. The message is processed in order to detect its upstream/downstream directionality. If the current platform is the not the last platform involved into the service, the message is modified and then forwarded towards the NHOP otherwise a *Service Deletion Response* is generated and finally propagated towards the PHOP.

- *SDREQ_ERR*: The FSM enters this state upon the unsuccessful reception/processing of a *Service Deletion Request* message. The FSM revokes the operations required for the service deletion and moves towards the *SERVICE_UP* state

- *MSG_SDRESP*: The FSM enters this state upon the reception of a *Service Deletion Response*. If the current platform is the first platform involved into the service, the FSM generates an *ev_ServiceDown* event. Otherwise the *Service Deletion Response* message is modified and then forwarded towards the PHOP.

- *SERVICE_DOWN*: A service deletion request has been completed successfully.

- *MSG_NOTIFY*: The FSM transitions into state from the *SERVICE_UP* state upon the reception of a *Notify* message. The message is processed in order to detect its upstream/downstream direction. If the current platform is not the last platform involve into the service, the message is modified and forwarded towards the NHOP. If the platform is the last platform involved in the service two different behaviors may happen, depending on the passes required for the *Notify* processing: a) if the *Notify* does not require an acknowledgment the *ev_NotifyCompleted* event is generated and the FSM transitions into the *SERVICE_UP* state b) a *Notify ACK* message is propagated towards the PHOP/NHOP depending on the initial *Notify* message downstream/upstream directionality.

- *PLAT_INSTALL_ERR*: The FSM enters this state upon the detection of errors regarding the resource allocation. The FSM releases all the resources previously allocated.

- *MSG_NOTIFYACK*: A *Service Notify ACK* has been received. The message is processed in order to detect errors and its direction (i.e. upstream or downstream). If the current platform is the first involved in the upstream direction an *ev_NotifyCompleted* is generated. Otherwise the *Notify ACK* message is modified and then propagated towards the NHOP/PHOP, depending on the direction.

- *NOTIFY_ACK_ERR*: The FSM enters this state upon the detection of errors while receiving/processing a *Notify ACK* message. The auto-generated *ev_NotifyNotCompleted* event brings the FSM into the *SERVICE_UP* state.

- *NOTIFY_ERR*: The FSM enters this state upon the detection of errors while receiving/processing a *Notify* message. The auto-generated *ev_NotifyNotCompleted* event brings the FSM into the *SERVICE_UP*.

4.2.1.2 FSM Events

- *ev_FSMInitialized*: The FSM has been initialized successfully and is now awaiting new events.

- *ev_MSG_SSRequestReceived*: A *Service Setup Request* has been received.

- *ev_NotInFPR*: This event is generated if the current platform is not contained into the received message's FPR.

- *ev_ResourcesLocked*: This event is generated if the *resourced* replies positively to the resources reservation request.
- *ev_UnavailableResources*: This event is generated if the *resourced* replies negatively to the resources reservation request.
- *ev_SSRequestErr*: The *Service Setup Request* message contains error(s).
- *ev_CompletionReqSent*: This event is generated when a FPR completion request is issued to the *Routing & Service Helpers*.
- *ev_CompletionRespRcvd*: This event is generated when a positive reply from the *Routing & Service Helpers* is received (this reply contains the completed FPR).
- *ev_ErrinCompletion*: This event is generated if the FPR completion cannot be performed.
- *ev_CreateAssocErr*: The ServiceID (contained into the message) and the AllocationID (received from the *resourced*) cannot be bound together.
- *ev_CannotSendSSRequest*: The *Service Setup Request* cannot be propagated to the next hop.
- *ev_MSG_SSAllocationReceived*: A *Service Setup Allocation* message has been received.
- *ev_ResourcesUnlocked*: The previously reserved resources have been released successfully.
- *ev_ErrSpecReceived*: The ERROR-SPEC field contained into the message is not empty (i.e. an error has been detected by the downstream/upstream platforms).
- *ev_SSAllocationErr*: The *Service Setup Allocation* contains error(s).
- *ev_MSG_SSConfirmationReceived*: A *Service Setup Confirmation* has been received.
- *ev_MSG_SSConfirmationSent*: The *Service Setup Confirmation* has been propagated to the next hop.
- *ev_ErrResourced*: The allocation request has been refused.
- *ev_MSG_SSACKReceived*: A *Service Setup ACK* has been received.
- *ev_ServiceUp*: The service setup has been completed successfully.
- *ev_SSConfirmErr*: A *Service Setup Confirmation* message contains error(s).
- *ev_SSAckErr*: A *Service Setup ACK* message contains error(s).
- *ev_MSG_SDRequestReceived*: A *Service Deletion Request* message has been received.
- *ev_MSG_NotifyReceived*: A *Notify* message has been received.

- *ev_ForceServiceDeletion*: This event is generated to force the elimination of a particular service.
- *ev_MSG_SDResponseReceived*: A *Service Deletion Response* message has been received.
- *ev_SDRequestErr*: The *Service Deletion Request* contains error(s).
- *ev_ResourcedResponseErr*: The *resourced* replies to a request with an error.
- *ev_AllocIdNotFound*: The allocation identifier received from the *resourced* differs from the identifier contained into the data model.
- *ev_ServIdNotFound*: The message does not contain a valid service identifier or the service identifier contained into the message is missing into the system.
- *ev_DeletionErr*: The service deletion cannot be completed.
- *ev_ServiceDown*: A service deletion request has been completed successfully.
- *ev_SDResponseErr*: The *Service Deletion Response* message contains error(s).
- *ev_Timeout_expired*: A timeout has been fired.
- *ev_MSG_NotifyACKReceived*: A *Notify ACK* message has been received.
- *ev_NotifyErr*: The *Notify* message processing towards the destination platform cannot be completed successfully.
- *ev_ServiceIDErr*: The *Service-ID* field contained into the received message has error(s).
- *ev_NotifyCompleted*: The *Notify* request has been received correctly by the destination platform.
- *ev_NotifyNotCompleted*: The *Notify* request has not been received by the destination platform.
- *ev_NotifyACKErr*: The *Notify ACK* message contains error(s).

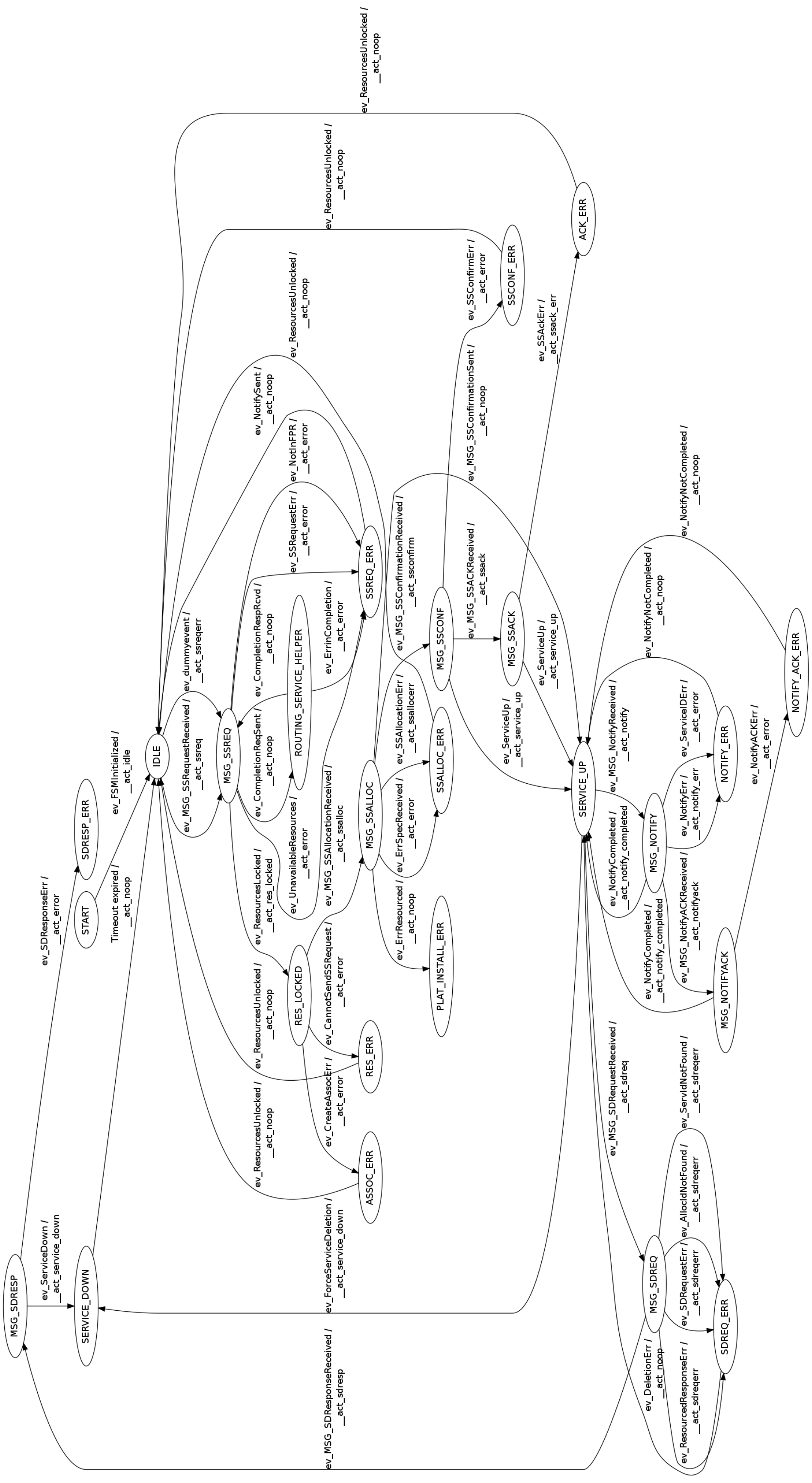
START
IDLE
SDRESP_ERR
MSG_SDRESP
SERVICE_DOWN
MSG_SSREQ
ROUTING_SERVICE_HELPER
SSREQ_ERR
RES_LOCKED
RES_ERR
ASSOC_ERR
MSG_SSALLOC
SSALLOC_ERR
PLAT_INSTALL_ERR
MSG_SSCONF
SSCONF_ERR
MSG_SSACK
ACK_ERR
SERVICE_UP
MSG_NOTIFY
NOTIFY_ERR
MSG_NOTIFYACK
NOTIFY_ACK_ERR
MSG_SDREQ
SDREQ_ERR
ev_FSMInitialized / __act_idle
Timeout expired / __act_noop
ev_SDResponseErr / __act_error
ev_ServiceDown / __act_service_down
ev_MSG_SSRequestReceived / __act_ssreq
ev_dummyevent / __act_ssreqerr
ev_NotifySent / __act_noop
ev_ResourcesUnlocked / __act_noop
ev_ResourcesUnlocked / __act_noop
ev_ResourcesUnlocked / __act_noop
ev_ResourcesUnlocked / __act_noop
ev_ResourcesUnlocked / __act_noop
ev_NotInFPR / __act_error
ev_SSRequestErr / __act_error
ev_CompletionRespRcvd / __act_noop
ev_CompletionReqSent / __act_noop
ev_ErrInCompletion / __act_error
ev_UnavailableResources / __act_error
ev_ResourcesLocked / __act_res_locked
ev_MSG_SSAllocationReceived / __act_ssalloc
ev_CannotSendSSRequest / __act_error
ev_CreateAssocErr / __act_error
ev_ForceServiceDeletion / __act_service_down
ev_MSG_SDResponseReceived / __act_sdresp
ev_MSG_SSConfirmationReceived / __act_ssconfirm
ev_SSAllocationErr / __act_ssallocerr
ev_ErrSpecReceived / __act_error
ev_ErrResourced / __act_noop
ev_SSConfirmErr / __act_error
ev_MSG_SSConfirmationSent / __act_noop
ev_MSG_SSACKReceived / __act_ssack
ev_SSAckErr / __act_ssack_err
ev_ServiceUp / __act_service_up
ev_ServiceUp / __act_service_up
ev_NotifyNotCompleted / __act_noop
ev_NotifyNotCompleted / __act_noop
ev_MSG_NotifyReceived / __act_notify
ev_NotifyCompleted / __act_notify_completed
ev_NotifyCompleted / __act_notify_completed
ev_ServiceIDErr / __act_error
ev_NotifyErr / __act_notify_err
ev_MSG_NotifyACKReceived / __act_notifyack
ev_NotifyACKErr / __act_error
ev_MSG_SDRequestReceived / __act_sdreq
ev_ServIdNotFound / __act_sdreqerr
ev_AllocIdNotFound / __act_sdreqerr
ev_SDRequestErr / __act_sdreqerr
ev_ResourcedResponseErr / __act_sdreqerr
ev_DeletionErr / __act_noop

Figure 4.3: Protocol FSM

4.3 Service Manager

The Service Manager runs at the CHANGE provider and transforms the client requests into signaling that CHANGE platforms can understand. The architecture of the *Service Manager* is shown in Fig. 4.3. The higher part handles the component logic while the lower part (the *transportd*) handles the transport related mechanisms.

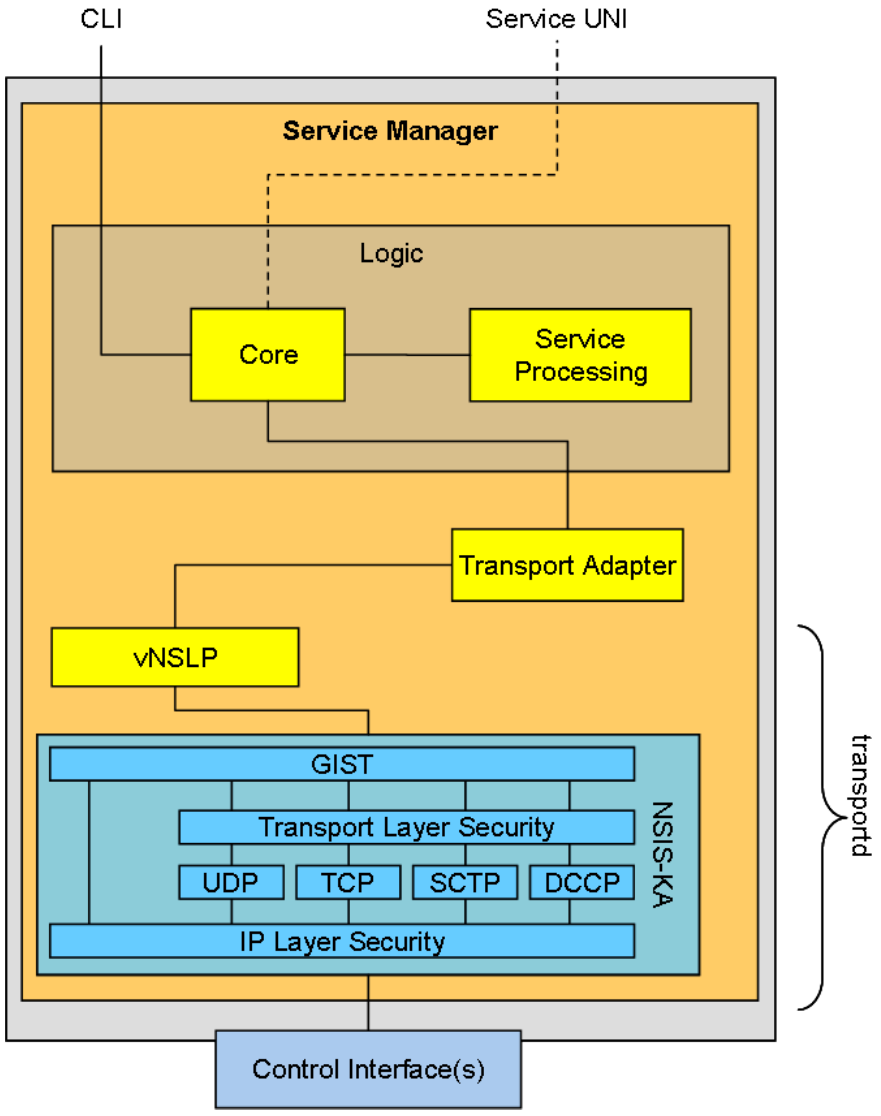

Figure 4.4: Service Manager

The upper part of the *Service Manager* is composed by the following logical blocks:

- The manager *logic*:
  - Core: This logical block mainly implements the *Service Manager* logic. It mainly:
    - ∗ Handles the Service-UNI requests/replies.
    - ∗ Performs the ingress service request processing (sec. 3) by traslating the requests into lower-level actions that will be instantiated into the platforms via the signaling protocol as described in D4.2.

    * Performs the service composition requests when the service description is incomplete (ref. D4.2).
    * Instantiate the signaling among the CHANGE platforms, depending on the deployment model.
  - Service Processing: The *Service Processing* block handles the user request processing and "forms" a correct, loop-free FPR containing the per-platform actions to be instantiated along the E2E path. The processing is performed using the tool described in sec. 3, with the XML backend.

- Transport Adapter: The *Transport Adapter* handles the serialization/deserialization of the messages to/from the *transportd*.

The *Service Manager* supports an interactive Command Line Interface (CLI) for debugging purposes. This interface will be obsoleted by the Service-UNI interface in the final release due for D4.5. For the *transportd* details refer to sec. 4.1

4.3.1 Command Line Interface

The *Service Manager* CLI provides the following commands (refer to D4.2 for the various parameters):

- Configuration:
  - *fpr-create*: Create an empty *Flow Processing Route*
    * input parameter: FPR-ID.
  - *fpr-build*: Fill a previously created *Flow Processing Route*
    * input parameters: FPR-ID, PLATFORM-ID, PLATFORM-ID, ..., PLATFORM-ID
  - *fpr-delete*: Delete a previously created *Flow Processing Route*
    * input parameter: FPR-ID.
  - *fpr-show*: Show all *Flow Processing Routes* information.
  - *platform-add*: Add a platform module
    * input parameters: PLATFORM-ID, PLATFORM-ADDRESS and PLATFORM-PORT.
  - *platform-config*: Configure a platform with one or more processing modules
    * input parameters: PLATFORM-ID, PM-ID, PM-ID, ..., PM-ID
  - *platform-del*: Delete a platform
    * input parameter: PLATFORM-ID
  - *platform-show*: Show the platforms configuration
  - *pm-add*: Add a processing module

    * input parameters: PM-ID, PM-TAG, PM-TYPE, PM-VISIBILITY, PM-CONSTRAINTS, PORT-ID, PORT-ID ... PORT-ID
  - *pm-config*: Configure a processing module
    * input parameters: PM-ID and a string (containing the processing module configuration).
  - *pm-del*: Delete a processing module
    * input parameter: PM-ID.
  - *pm-show*: Show the processing modules configuration
  - *port-pm-add*: Add a processing module port
    * input parameters: PM-ID and PORT-TYPE.
  - *port-pm-del*: Delete a processing module port
    * input parameter: PORT-ID.
  - *port-pm-show*: Show the processing module ports configuration
- Messages:
  - *sdreq-create*: Create an empty *Service Deletion Request* message
    * input parameter: SDREQ-ID
  - *sdreq-build*: Fill a previously created *Service Deletion Request* message
    * input parameters: SDREQ-ID, SERVICE-ID, SESSION-ID, FID and HOP.
  - *sdreq-send*: Send a previously created *Service Deletion Request* message
    * input parameters: SDREQ-ID, PLATFORM-ID, PLATFORM-PORT and a TIMEOUT
  - *sdreq-delete*: Delete a previously created *Service Deletion Request* message
    * input parameter: SDREQ-ID
  - *ssreq-create*: Create an empty *Service Setup Request* message
    * input parameter: SSREQ-ID
  - *ssreq-build*: Fill a previously created *Service Setup Request* message
    * input parameters: SSREQ-ID, SERVICE-ID, SESSION-ID, HOP, FPR-ID
  - *ssreq-send*: Send a previously created *Service Setup Request* message (it must contain a *Flow a Processing Route*)
    * input parameters: SSREQ-ID, PLATFORM-ADDRESS, PLATFORM-PORT and a TIMEOUT
  - *ssreq-delete*: Delete a previously created *Service Setup Request* message

- input parameter: SSREQ-ID

- *notify-create*: Create an empty *Notify* message
  - input parameter: NOTIFY-ID
- *notify-build*: Fill a previously created *Notify* message
  - input parameters: NOTIFY-ID, SERVICE-ID, SESSION-ID, FID, HOP, PLATFORM-ID, MESSAGE-ID, ACK-FLAG and OPAQUE-DATA
- *notify-send*: Send a previously created *Notify* message
  - input parameters: NOTIFY-ID, PLATFORM-ADDRESS, PLATFORM-PORT and TIMEOUT.
- *notify-delete*: Delete a previously created *Notify* message
  - input parameter: NOTIFY-ID

- Miscellaneous:
  - *variable-set*: Set a variable value
    - input parameter: A string representing the variable value. The value is automatically transformed for boolean variables (e.g. *warning-as-errors*, *exit-on-error* and *show-prompt*).
  - *variables-show*: Show all the variables values.
  - *?*, *help*: Print a brief help.
  - *exit*: Exit from the CLI.

The program behavior is controlled by variables accessible from the CLI. Their value can be changed using the *variable-set* command. The following variables are exported to the CLI:

- *show-prompt*: Enables/disables the command prompt.
- *warnings-as-errors*: If set to "true" the program handles warnings as errors
- *exit-on-error*: If set to "true" the program exits at the first error
- *log-format*: Changes the logs format.
- *log-level*): Sets the logs level (valid levels are: *debug*, *info*, *warning*, *error* and *critical*).

The *show-prompt*, *warnings-as-errors* and *exit-on-error* commands are used by the regression tests performed during 'make check'. Refer to sec. A for further details.

4.3.2 Usage example

The following example represents the set of commands that have to be issued to the *Service Manager* in order to create and then destroy a service across three different platforms.

```
#
# Let's add few definitions that will be referenced later on:
#

# The PMs ports:
port-pm-add port1 input
port-pm-add port2 output
port-pm-add port3 input
port-pm-add port4 output
port-pm-add port5 input
port-pm-add port6 output

# The PMs:
pm-add name1 image1 pm-type-1 True constraints port1 port2
pm-add name2 image2 pm-type-2 True constraints port3 port4
pm-add name3 image3 pm-type-3 True constraints port5 port6

# The PMs configurations:
pm-config pm-type1 http port 8080
pm-config pm-type2 udp port 10000
pm-config pm-type3 tcp port 20000

# The platforms control interfaces:
platform-add platform0 10.0.2.185 50002
platform-add platform1 10.0.2.237 50002
platform-add platform2 10.0.2.234 50002

# Finally, the platforms configuration:
platform-config platform0 pm-type1
platform-config platform1 pm-type2
platform-config platform2 pm-type3
```

```
...

# We must create an empty FPR:
fpr-create test

# And then stitch the platforms there (the sequence reflects the
# platform ordering in the E2E path):
fpr-build test platform0 platform1 platform2

# The Service Manager now holds, in its internal data-structures, the
# FPR. This FPR will be (indirectly) used for more than one type of
# signaling messages (at least for service creation and deletion).

...

#
# Service creation:
#

# We now build an empty Service Setup Request:
ssreq-create test-creation

# Then we "build" the message (the second parameter represents the
# SERVICE-ID that will be referred also in the deletion of this
# service)
ssreq-build test-creation 1 99 fid1 10.0.2.185 test

# And finally send the message (containing the previously built FPR):
ssreq-send test-creation 10.0.2.185 50001 60

...

#
# Service deletion:
```

```
#


# To delete the previously instantiated  service we have to create a
# new message (a Service Deletion Request this time):
sdreq-create test-deletion

# As in the Service Setup Case we must fill the message with its
# parameters. The Service-ID (the second parameter) will be used by
# the Service Manager to retrieve the (previously built) FPR from its
# internal data-structures:
sdreq-build test-deletion 1 99 fid1 10.0.2.185

# And finally send the message:
sdreq-send test-deletion 10.0.2.185 50001 500
```

4.4 Service Broker

The *Service Broker* runs at CHANGE providers and its goal is to filter the signaling from/to a particular domain. It allows changing the FPR when crossing into a new domain; this may be needed if, for instance, the routing models of the neighboring domains differ.

The *Service Broker* architecture is shown in Fig. 4.4.

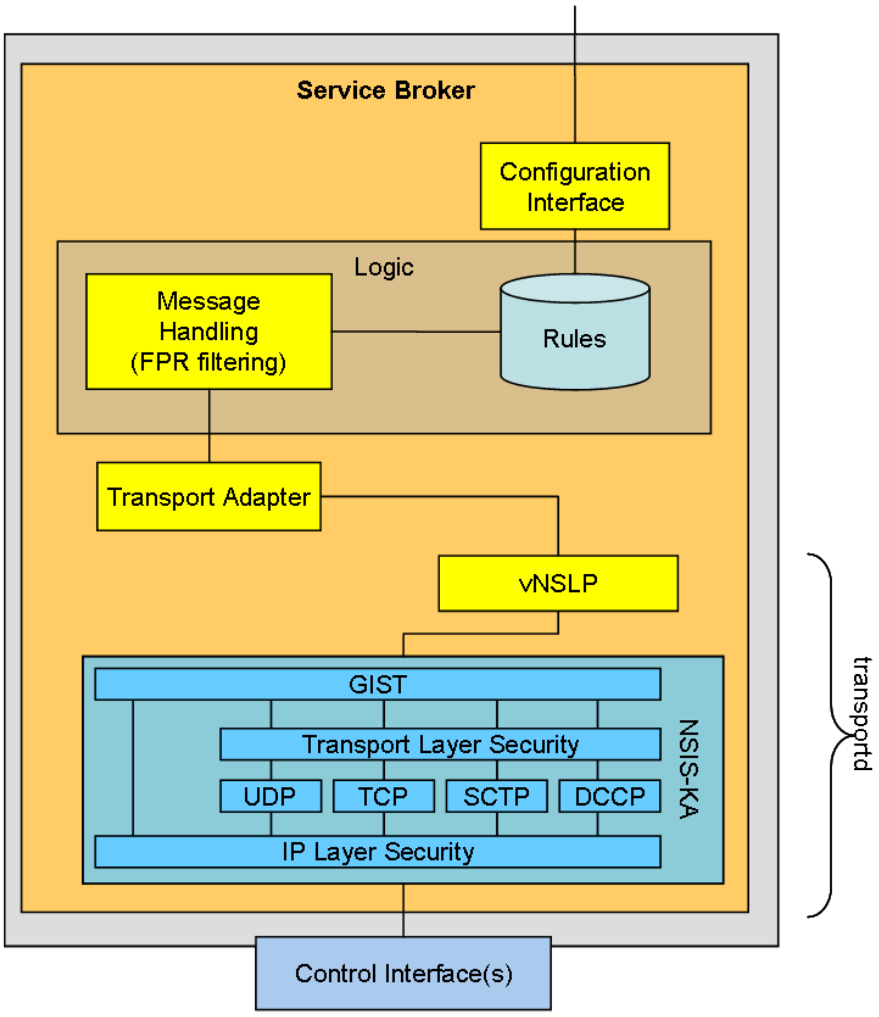

Figure 4.5: Service Broker

The higher part of the *Service Broker* (Fig. 4.4) is composed by the following logical blocks:

- Configuration interface handler: Handles the commands issued from the interactive Command Line Interface (CLI) and allows to change the filtering rules.
- The broker *logic*:
  - Message handler: Handles the ingress/egress messages by applying pre-configured rules (e.g.: blocking and/or masquerading the requests traveling towards particular domains and/or platforms).
  - Rules DB: Maintains the configuration rules.

  - Transport Adapter: The *Transport Adapter* handles the serialization/deserialization of the messages to/from the *transportd*.

For the *transportd* details refer to sec. 4.1

4.4.1 Command Line Interface

The *Service Broker* CLI provides the following commands (refer to D4.2 for the various parameters). This interface will be enhanced in the next months to support more functionalities.

- Configuration:
  - *masquerade-fprr-set*: Set masquerade-fprr parameter
    - ∗ input parameters: A boolean value. At the moment all the FPR is masqueraded.
  - *masquerade-fprr-show*: Show the masquerade-fprr parameter value
  - *refuse-frompid-add*: Add a refusing rule for the requests from a platform
    - ∗ input parameters: PLATFORM-ID.
  - *refuse-frompid-del*: Delete a refusing rule for the requests from a platform
    - ∗ input parameters: PLATFORM-ID.
  - *refuse-frompid-show*: Show the refusing rules for the requests from a platform
  - *refuse-topid-add*: Add a refusing rule for the requests towards a platform
    - ∗ input parameters: PLATFORM-ID.
  - *refuse-topid-del*: Delete a refusing rule for the requests towards a platform
  - *refuse-topid-show*: Show refusing rules for the requests towards a platform
  - *rules-show*: Show all the rules
- Miscellaneous:
  - *?*, *help*: Print a brief help
  - *exit*: Exit from the CLI

4.5 Sources overview

The package source-code directory layout is shown in Fig. 4.5:

```
/ ..........................................................................
├─ bootstrap ............... The script used to bootstrap the build framework from the repository
├─ configure ....................... The script that must be used to configure the source package
├─ configure.ac ........................................ the “configure” script source file
├─ src .................................................. The main source code directory
│  ├─ libs ......................................................... Common libraries
│  │  └─ siglib ............................ The Python libraries common to all components
│  ├─ transportd ........................................ The transportd sources directory
│  │  └─ nsis-ka .......................................... The NSIS-ka embedded package
│  ├─ sigmgr ............................................. The Signaling Manager sources
│  │  └─ sigmgr.in ................................. The Signaling Manager main program
│  │     └─ signald ......................................... The signald sources directory
│  │        ├─ signald.in ..................................... The signald main program
│  │        └─ sigmgr ............................... The signald specific (Python) libraries
│  ├─ svcbrk .................................................. The Service Broker sources
│  │  ├─ svcbrkd.in .................................... The Service Broker main program
│  │  └─ svcbrk ............................... The Service Broker specific (Python) libraries
│  └─ svcmgr ................................................ The Service Manager sources
│     ├─ svcmgrd.in .................................. The Service Manager main program
│     └─ svcmgr ............................. The Service Manager specific (Python) libraries
├─ test ...................................... A directory containing self-consistency checks
├─ tools ........................................ A directory containing testing related tools
├─ build-aux ........................... A directory containing build related files and scripts
└─ m4 ............................................ A directory containing build related files
```

Figure 4.6: Signaling package source code directory structure

The *src/sigmgr/sigmgr.in*, *src/sigmgr/signald.in*, *src/svcbrk/svcbrkd.in* and *src/svcmgrd.in* must be considered as input files utilized by the build framework to produce their related main programs (i.e. the *signald*, *svcbrkd* and *svdmgrd* respectively) during compilation.

5 Conclusions

This document has provided an overview of the implementation of the CHANGE inter-platform communication software. The software that accompanies this deliverable is a first prototype of the CHANGE architecture, allowing clients to instantiate flow processing seamlessly.

This prototype is still under development and many features are missing. A complete version will be released with Deliverable 4.5.

A The build framework

The build framework is common to all D4.4 packages (i.e. *signaling* and *processing*). It relies on *autoconf* (ref. [3]), *automake* (ref. [4]) and *libtool* (ref. [5]) development tools. This set of tools is also referred with the term *autotools*.

A.0.1 Building and installing a package

The shell commands './configure; make; make install' configure, build, and install a package from a bootstrapped build environment or a tarball. Refer to A.0.2 and A.0.3 for further details on bootstrapping a build environment and/or building a package's tarball.

The following more-detailed instructions are generic; see the 'README' file for instructions specific to each package. A package may provide an 'INSTALL' file with more detailed install instructions and may not implement all the features documented below. The lack of an optional feature in a given package is not necessarily a bug.

The 'configure' shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a 'Makefile' in each directory of the package. It may also create one or more header files (e.g. usually header files) containing system-dependent definitions. Finally, it creates the shell script 'config.status' that can be used to recreate the current configuration, and a 'config.log' file containing detailed output (useful mainly for debugging the 'configure' script).

The simplest way to compile a package is:

1. 'cd' to the directory containing the package's source code
2. Type './configure' to configure the package for your system. Running 'configure' might take a while. While running, it prints some messages telling which features it is checking for. The 'configure' script accepts options that enable/disable particular checks and/or compilation options. Please refer to its help for additional information. To access the 'configure' help type './configure –help'.
3. Type 'make' to compile the package.
4. Optionally, type 'make check' to run any self-tests that come with the package, generally using the just-built uninstalled binaries.
5. Type 'make install' to install the programs and any data files and documentation. When installing into a prefix owned by root, it is recommended that the package be configured and built as a regular user, and only the 'make install' phase executed with root privileges.

The program binaries and object files removal from the source code directory can be performed by typing 'make clean'. To also remove the files that 'configure' created (in order to re-compile the package for a different system or with a different set of options) type 'make distclean'. To remove the installed package from the system type 'make uninstall'.

A.0.2 Bootstrapping a build environment

To bootstrap a package's build environment use the following procedure:

1. 'cd' to the directory containing the package's source code.
2. Type './bootstrap' to (re)build the 'configure' script and all the intermediate files (e.g. the Makefile.in files).

A.0.3 Building a tarball from a repository snapshot

To build a package tarball from a source repository snapshot (i.e. *svn checkout*) use the following procedure:

1. 'cd' to the directory containing the package's source code.
2. Type './bootstrap' to bootstrap the build framework.
3. Type './configure' to configure the package for your system.
4. Type 'make dist' to finally build the package's tarball.

A.0.4 Notes

As a consequence of adopting the *autotools*, the following notes apply to all the D4.4 packages:

- The *configure.ac* file is the input file used by the framework to build the *configure* script.
- All the *Makefile.am* and *Makefile.in* files are the input files used by the framework to produce the correspondent *Makefile* files during "configuration" time.
- The files contained in *build-aux* are used by the build framework during both the *bootstrap* phase and the *configuration* phase.
- The files contained into the *m4* directory are M4/autom4te macros (ref. [6]) used during the *bootstrap* phase to build the *configure* script.

Bibliography

[1] Dave Beazley et al. SWIG.

[2] Karlsruher Institut für Technologie. NSIS-ka a free C++ implementation of NSIS protocols.

[3] GNU Project - Free Software Foundation. Autoconf.

[4] GNU Project - Free Software Foundation. Automake.

[5] GNU Project - Free Software Foundation. GNU Libtool - The GNU Portable Library Tool.

[6] GNU Project - Free Software Foundation. GNU M4.

[7] H. Schulzrinne and R. Hancock. GIST: General Internet Signalling Transport. RFC 5971, October 2010.