University POLITEHNICA of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

Multiengine Software Switch Implementation Based on LiSA

Technical Advisers:

Ing. Ioan Petru Nicu

Ing. Radu Rendec

Scientific Adviser:

Şl. dr. ing. Răzvan Deaconescu

Author:

Mihaela Alexandra Mărtinaş

Bucharest, 2013

I would like to thank Ioan Nicu and Radu Rendec
for offering me the opportunity to continue developing
LiSA. They challenged me to think outside the box for finding
the solutions that are the most suitable for the project.

An other person which played an important role is
Răzvan Deaconescu, whose enthusiasm about the
project motivated the team.

Also, I would like to thank to my colleagues who
contributed to this project: Andreea Hodea,
Claudiu Ghioc and Victor Duţă.

Abstract

LiSA stands for Linux Switching Appliance and it is an open-source project that was developed by two former students of this faculty: Ioan Nicu and Radu Rendec. The application is a software switch which offers support for Data Link Layer and Network Layer packet switching.

Improvements were brought to the project since it was originally created, but it still did not meet the conditions to be integrated in a Linux kernel. Having this purpose in mind, an API was created. Adopting this approach, LiSA is able to support multiple back-end implementations.

Beside contributing to the implementation of the API, the goal of my project was to implement an engine that unifies the different back-end implementations. Given this feature, the user is able to manage multiple pieces of back-end in a transparent manner.

Contents

List of Figures

Notations and Abbreviations

API – Application Programming Interface
BPDU – Bridge Protocol Data Unit
CAM – Content Addressable Memory
CDP – Cisco Discovery Protocol
CLI – Command Line Interface
FDB – Forwarding Database
IGMP – Internet Group Management Protocol
IP – Internet Protocol
IPC – Inter Process Communication
IPC – Inter Process Communication
JSON – JavaScript Object Notation
LiSA – Linux Switching Appliance
POSIX – Portable Operating System Interface
RSTP – Rapid Spanning Tree Protocol
SNAP – Subnetwork Access Protocol
STP – Spanning Tree Protocol
VDB – Vlan Database
VLAN – Virtual Local Area Network
VM – Virtual Machine

Chapter 1

Introduction

Taking a quick glance over the history one can observe that there is a constant element which persisted over the time: the need of communication, of exchanging information. Along the time various methods to satisfy this necessity were developed, but the Internet was a technological breakthrough. Over the time it developed and became oriented on the needs of every user, each can choose to use it in the way that suits him the best. But the Internet it is not just the interface displayed in a browser or in a program, the Internet has many hidden faces.

In the last decade communication over Internet took complex shapes: from the classic email it evolved into video conferences, multimedia presentations. Real time interaction became an every day necessity. Due to these demands, new solutions had to be found in order to ensure user access to the resources that he needs. The new architecture materialized in a distributed system which had to meet some basic conditions: scalability, adaptability, availability and security. This new design led to large physical networks which, in order to run at optimal parameters, should be backed up by proper hardware solutions. Networking devices are continuously improved to meet the need of high-speed data transmission. "Ethernet, which at the beginning ran at 3 to 10 Mbps, now runs at 100 (fast Ethernet), 1000 (gigabit Ethernet) or 10000 Mbps (10 gigabit Ethernet)"[1].

In order to exploit all the resources that a hardware device propounds, it shouldn't be seen as a equipment that can provide a single functionality. Virtualization can be used to decrease hardware usage because it enables multiple operating systems to run on the same system, sharing hardware resources.

The key of a network is communication, hence a switching module is needed to ensure this functionality. But the question that rises is what should one choose: a hardware switch or a software switch? In the following section (Section 1.1) a few arguments that back up the idea of software switch are presented.

1.1 Software Switch

Both hardware and software switches serve the same purpose: enable connection between computers, servers, printers or other devices from an area. The difference is that

a software switch is an application that links network devices or network segments. A software implementation makes it hardware independent and the application is portable on any system that is compatible with the software.

To take advantage of the benefits offered by virtualization, the VMs(Virtual Machines) should be connected to be able to exchange data. A software switch would be a complement for virtualization because it will have the same functionality as a hardware switch without a dedicated equipment to help it. This infrastructure formed from VMs and a software switch can be used for testing new features before deploying the application in the environment for which it was meant. Also, deployment of an application in general would be a big plus for this architecture, because numerous install activities would be eliminated.

As mentioned before, the hardware necessary to support nowadays applications has now to meet the highest standards. From this reason many computers became outdated because they were not able to cope with application requirements. Having in mind the fact that a software switch does not need many hardware resources, the equipment can be used for switching.

An other advantage of a software switch is that it can offer the possibility to manage multiple software switches in a transparent way. Multiple software switches can be aggregated, each of them providing access to different resources. The access to the component switches is made through an interface that is multiplexing the received commands by redirecting them to the members. This new unified switch has a high degree of usability because the user does not have to configure one switch at a time, the interface will send the commands to the proper switch.

The didactic purpose of a software switch should also be mentioned. It is ideal for learning about switching and it does not imply buying a dedicated equipment. It can be installed on a personal computer and with VMs one can design small networks to simulate a certain behaviour of a network.

There are software developers which showed interest in developing software switch implementations. Between these developers can be mentioned the team that initially designed the software switch named LiSA.

1.2 LiSA as a Software Switch

LiSA was started by Ioan Nicu and Radu Rendec as a diploma project in 2005. It became an open source project to which contributed not only the initial authors, but also other students interested in software switching.

The project comes to offer a convenient switching solution for small sized networks. It was designed for systems which use Linux distribution and it uses the facilities offered by the system: switching, packet filtering and traffic shaping.

LiSA implements Data Link Layer and Network Layer switching and offers support for VLANs (*8*02.1q), CDP (Cisco Discovery Protocol), RSTP (Rapid Spanning Tree Protocol) and IGMP (Internet Group Management Protocol) snooping. The packet

switching is entirely done in software. Of course, a hardware implementation would seem a more efficient solution, but the results are quite satisfactory. This hardware independent implementation makes it portable and ready to use on any system that runs a Linux distribution.

When the design was made, the users were an important variable that was took into account. They are offered the possibility to configure, control and monitor the switching process through a switch like *Command Line Interface* or shorted *CLI*. This comes as an advantage for those who had already used a similar interface, but also for beginners because they will learn generic commands which can be used on other switches.

The elements that can be configured using LiSA are VLAN routing and interfaces (ethernet interface, netdevice interfaces or virtual interfaces). Through the CLI, MAC addresses and IP addresses can be managed for the interfaces which are part of the switch. Also, information about the current configuration can be displayed.

Other important features that LiSA offers are the two daemons which implement the above mentioned protocols: RSTP and CDP. The daemons are standalone programs that should be run on their own in order to provide the desired functionality.

The initial design of LiSA has been changed since the initial project was released, due to the desire to obtain a generic software switch that can support different back-end implementations. In order to obtain this new architecture a new level of abstraction had to be introduced to ensure backward compatibility with the old implementation, but also to be able to use new back-end implementations.

One of the problems faced in the process of migrating to a generic infrastructure was the functionality of the daemons. They communicated with the kernel through special sockets created for LiSA. This would require modifying kernel sources for each new back-end that intends to use LiSA.

One of the goals of my project was to adapt LiSA to use raw sockets for other implementations beside the initial one. Raw sockets[1] are supplied by every Linux distribution. Adopting this approach, developing new back-end implementations would require only deploying functions which provide LiSA the software switch behaviour.

Due to the fact that LiSA is now able to back different back-end implementations, managing multiple pieces of back-end at a time seemed a necessary feature. This lead to the idea of multiengine: an entity to which LiSA switches with differenet back-end implementations are connected. This multiengine works as a mediator between the user and the component switches. One does not have to configure each switch at a time, the new entity will provide the functions to be able to access the switches implemented using LiSA in a transparent manner. Designing the multiengine and implementing its functionality became also part of my project.

[1] http://linux.die.net/man/7/raw

Chapter 2

LiSA Overview

As mentioned in Section 1.2 from Chapter 1, the initial architecture of LiSA suffered some changes along the time. In order to observe the modifications and to acknowledge their necessity, a review of each architecture will be made in the next sections.

2.1 Initial LiSA Architecture

The purpose of LiSA was to offer a switching solution for systems which run Linux as an operating system. A generic presentation of LiSA architecture can be observed in the following figure (Figure 2.1):

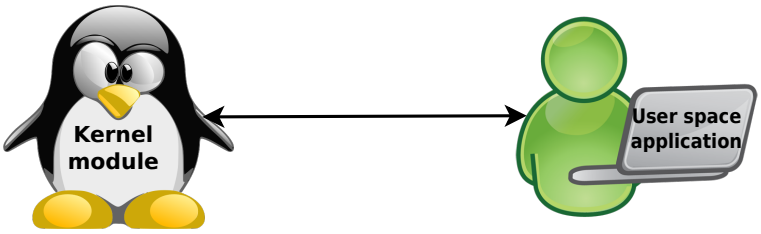

Figure 2.1: Generic LiSA architecture

The architecture covers both kernel space and user space. The functionality of each component is well delimited: the mechanism for packet switching is implemented in kernel space and user space is responsible for configuring, monitoring and controlling the switching process.

When the user space was designed, it was taken into consideration that one might open multiple configuration sessions at a time. For this type of behaviour race conditions are most likely to appear, so a locking mechanism had to be used in order to maintain the consistency of configurations.

All the decisions regarding the infrastructure of LiSA were based on the principle that the functionality of a component is not replicated on other parts of the implementation.

More architectural details can be observed in Figure 2.2 where the elements which compose each entity are specified and also the links that exist between them.

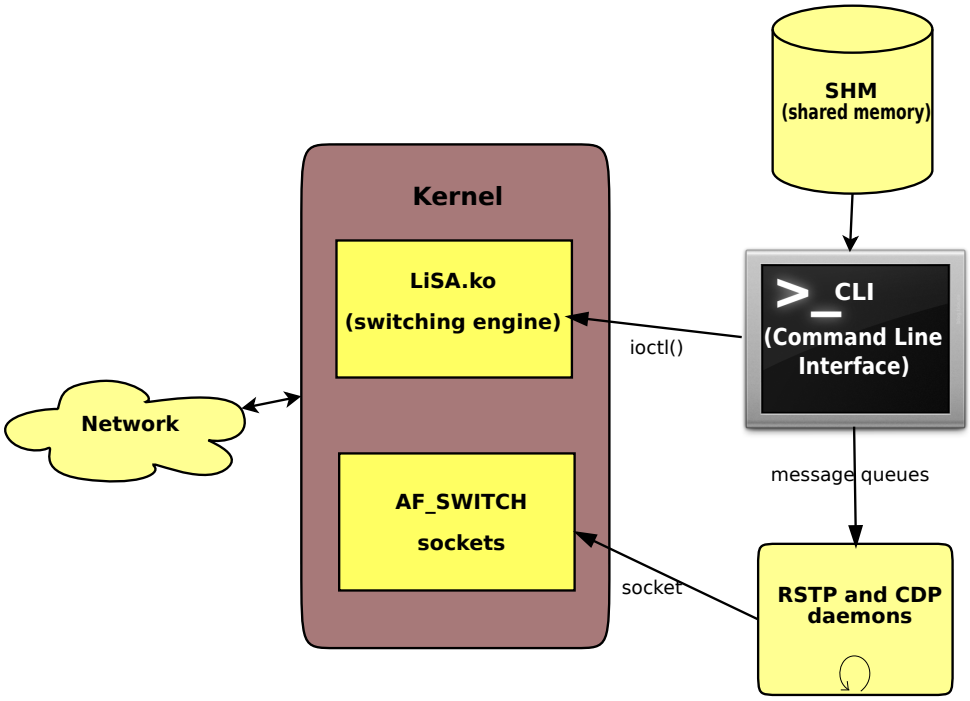

Figure 2.2: Initial LiSA Architecture

The kernel space contains the core of LiSA, the kernel module, which is responsible for the packet switching mechanism. To obtain this functionality the module can be divided into subcomponents:

- *Switching Engine*: it is the foundation of the module due to the fact that it receives the packets, makes switching decisions and applies algorithms used for efficiently communicating with the ports of the switch.
- *Forwarding Database (FDB)*: every switch should contain such a database or table for creating the CAM table [1].
- *VLAN Database (VDB)*: contains all the necessary information for VLAN routing.
- *VLAN interfaces (VIFs)*: necessary for implementing VLAN routing using the functionalities that Linux already offers.

The link between the user space application and the LiSA kernel module is made through *ioctl()* function calls. The kernel module implements the functionality for the functions requested through *ioctl()*.

To benefit from the facilities that the switch can offer, the user has to interact with the user space component. One of the most important pieces from user space is the command interpretor. The functionalities are packed into a shared library[2] which exposes

[1] CAM table is table held by a network switch to map MAC addresses to ports.
[2] http://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

the functions necessary to manage a switch. The elements which were used to obtain the shared library are the following:

- *The shell*: is used as an input for the application. The user configures the switch through the commands written in the shell. The Readline[1] library is used because offers word completion features for commands and it also maintains a history of the commands.
- *Parser*: the commands introduced are identified and validated in order to be executed by the *Command Runner*.
- *Command Runner*: is the entity which transforms the input received from the user into an actual switch action. For each CLI command there is a *ioctl()* function call. As mentioned before this is the way in which the information from user space is passed to kernel space. The kernel module component which handles *ioctl* calls will be the one to make the changes in the switch configuration.

As mentioned in the general description of LiSA, the design should allow multiple management sessions to run simultaneously. Hence, configuration information should be shared between all the instances. The information which is specific to the kernel space component (switching information) is shared by default between all instances, the problem appears in user space. To solve this inconvenience, an IPC solution was used: shared memory. All the information that should be shared: enabled users, passwords, VLAN description and interface description can be found in a shared memory segment which is protected using a lock, to ensure exclusive access to data.

Beside the *Command Line Interface*, the daemons which implement the protocols CDP and RSTP, are also part from user space. In order to benefit from the functionalities exposed of these protocols, they have to be launched before starting a switch management session. As seen on Figure 2.2, the CLI will communicate using message queues with the daemons. Because the daemons need to communicate with the network layer for sending and receiving packets, sockets were necessary. Moreover, packet filtering was necessary and from this reason `AF_SWITCH` sockets were designed for LiSA. This implied modifying kernel sources, not only adding a new module. More details about the daemons and the sockets used for communication will be exposed in Chapter 3.

Having the kernel module, *lisa.ko*, and the shared object, *liblisa.so* is not enough because the user cannot invoke directly their functions. An executable, named *swcli* was created, which makes some initial configurations before passing the access to the command interpretor. It can be started from any Linux standard shell and the user is given the highest privilege level. To run the executable, the user must have root privileges.

Although LiSA provided the behaviour required for a software switch, when those who initiated the project considered integrating it in a Linux distribution, they received a negative answer. The motivation was that there already exist two Linux modules(*bridge* and *8021q*) which combined can offer the same functionality. This led to the idea of making LiSA adaptable for multiple back-end implementations, this meaning that instead of the kernel module *lisa.ko* other implementations can be used. This new solution meant creating a Generic Software Switch based on LiSA.

[1] http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html

2.2 LiSA as a Generic Software Switch

In Figure 2.2 one could observe that the communication between the CLI and the kernel module is done without any mediator, using *ioctl* calls over `AF_SWITCH` sockets. Using these sockets which were designed for LiSA, the CLI was unable to support other back-end implementations.

There also existed an other problem regarding the implementation of the command line interface. The CLI makes one think about a commands executor, the instructions are received as in input, their syntax is analysed and some "magic" happens behind, this means that the CLI should not be aware of how things are implemented. But in the initial LiSA implementation it was also responsbile for implementing the functionality of the command. There was not set a bound between command processing at a syntactic level and the effective implementation of the functionality.

Due to the features implemented by the switching engine from kernel space, discarding the module *lisa.ko* was not considered as an option. The CLI also brought LiSA a big advantage because it made the interaction with the switch user friendly. In order to keep the old functionality, but also to be able to use other switching engines beside *lisa.ko*, an intermediate layer was introduced, named Middleware.

The details of the new architecture can be seen in the following figure:

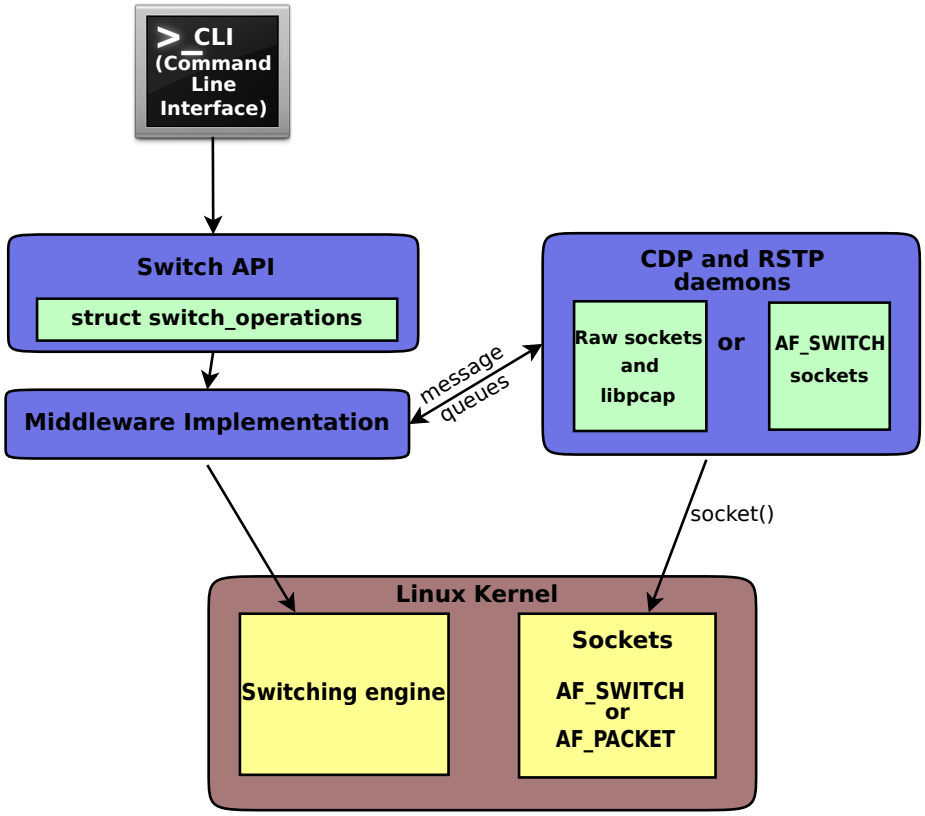

Figure 2.3: Current LiSA architecture

Putting side by side the former LiSA architecture and the current one, it can be observed that the CLI does not communicate directly with the kernel. The middleware will be a mediator between these two entities. Beside this aspect, an important improvement is that ioctl calls are not the default way to communicate with the kernel. Each middleware implementation which is made for a certain back-end, uses its specific means to transmit the commands in kernel space.

One of the drawbacks of the initial design was the usage of `AF_SWITCH` sockets for ioctl calls, but also for assuring the functionality of the daemons used for implementing the network protocols(CDP, RSTP). The solution found was the usage of *raw sockets* combined with the functionality of *libpcap*[1] library. Raw sockets can be used on any Linux distribution this meaning they would be available for any banck-end implementation. The protocols need to communicate with the kernel to send and receive notification packets.

Because the shared memory was not dependent on the nature of the implementation, no changes had to be made. The same data has to be shared between multiple switch sessions, no matter the nature of the implementation. The difference is that shared memory modifications will not be made from CLI, but from the implementation of the middleware.

Theoretically, for all the inconveniences a solution was found, but an effective implementation had to be made. From this reason LiSA API was created to fulfill the middleware functionality.

From an application which had two basic components (the command interpretor and the kernel module from kernel space), LiSA became a multilayer switch containing elements which accomplish a well established purpose:

- *Command Line Interface*: used only for receiving commands from the switch. It calls middleware functions for obtaining the desired functionality, without having any knowledge about the back-end.
- *Middleware*: contains the API called by the CLI to obtain the desired switch configuration. Also, its functions have to be implemented by each back-end to offer support for LiSA
- *Middleware implementation*: each back-end version will implement the API, putting up the software switch behaviour using its own features.

After introducing the middleware layer, LiSA became a generic software switch which is able to sustain different back-end implementations by virtue of the new layer.

The *switch* API, also named LiSA API, was shaped as a structure which contains pointers to functions that should be implemented by each middleware version. The members of the structure correspond to a command line instruction and there also exists a function which initializes the elements specific for each back-end. The definition of the structure can be found in the Appendix A.1. Each middleware implementation will contain the earlier mentioned structure and define the content of its functions.

[1] http://www.tcpdump.org/pcap.html

The *Command Line Interface* will change the behaviour of its functions. The direct link with the kernel through *ioctl* [1] call will be eliminated, being replaced by generic calls. The difference can be observed in the following figure which contains a snippet from the CLI function which handles the adding of Ethernet interfaces:

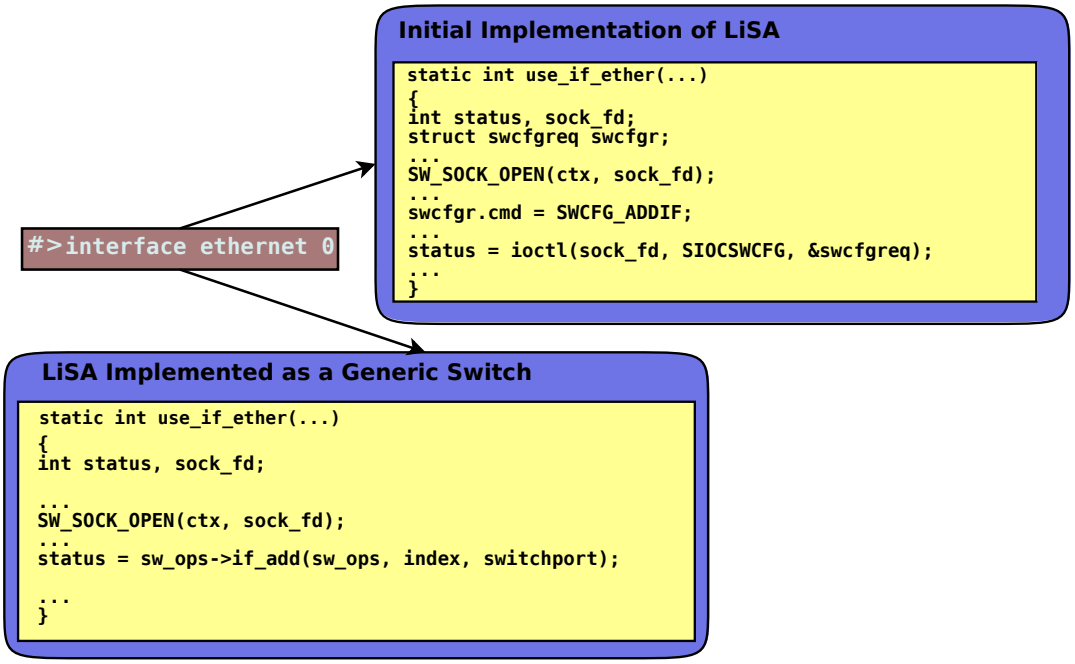

Figure 2.4: Example of Switch API call from CLI

In the former LiSA implementation, the CLI implemented the functionality by calling an ioctl function. It was not flexible for using other pieces of back-end, being strictly connected to the *lisa.ko* module which contained the handlers for the *ioctl* functions.

From Figure 2.4 one can observe that the CLI does not call a back-end specific function in LiSA implementation as a generic switch. An API function is called and back-end details are not made available to the CLI. Only the middleware implementation is aware of the back-end specific details. To be able to use the generic functions, the CLI keeps a reference to a *switch_ operations* structure, whose content points to the functions implemented using details specific to the back-end.

Using this approach, when a new back-end is implemented, it will not have to modify the CLI because the function call will be the same. This makes the CLI independent from the implementation layer.

With the generic switch implementation ready, the possibility of developing various pieces of back-end was explored. In the next section the approaches that were considered until now will be presented.

[1] http://man7.org/linux/man-pages/man2/ioctl.2.html

2.3 LiSA Back-end Implementations

The back-end implementation which seemed the most suitable for testing the new architecture was the one based on *lisa.ko* module, also named **LiSA back-end**. Each function from the structure *switch_ operations* was implemented and pointers to these implementations were set to ensure that the CLI uses the proper function definitions. All the API functions were implemented using *ioctl()* calls which were handled by the *ioctl handlers* implemented in the kernel module. The new design did not affect the former LiSA kernel module. Just a few changes were brought to the initial implementation of the module, but without a significant meaning. This back-end version still uses `AF_SWITCH` sockets for the *ioctl* calls and for implementing the protocols (CDP and RSTP).

The reason why LiSA was modified to be a generic switch in the first place, became a source of inspiration for the second back-end implementation. The problem was that Linux already has two modules *bridge* and *8021q* which combined can offer the same functionality as *lisa.ko* module. This back-end implementation named **bridge + 8021q** is based on the modules with the same names offered by Linux. Like all the back-end versions, this also has to implement the functions contained by the *switch_ operations* structure. The difference is that it uses *ioctl()* calls with sockets and requests specific to the two kernel modules provided by Linux. Other particularity of this implementation is the usage of *netlink sockets*[1] for some functionalities. Also, the `AF_SWITCH` sockets are not available for this implementation, hence *raw sockets* were used because they are available on any Linux distribution.

Beside the earlier mentioned back-end possibilities, other considered option was integrating LiSA with OpenWrt[2]. This back-end implementation was considered so that LiSA can offer support for hardware devices which have memory constraints and can run a small size Linux distribution, but also for SoC (Systems on a Chip[3]). Using LiSA, one can control the Ethernet switch integrated in these types of equipments. A drawback of this back-end is that it does not offer support for the protocols.

From integrating LiSA on an embedded device the ideas went to a higher level: integrating LiSA with a virtualization API, to be able to control multiple virtual machines. For this purpose *libvirt*[4] was chosen. The back-end implementation using *libvirt* makes LiSA to be an option for controlling networks composed from virtual machines.

Having all these back-end possibilities that can be independently controlled, one might think why not control all these back-end implementations at the same time, in a transparent way for each of them?

To handling this aspect, a new back-end was implemented, the so called **multiengine**. It can be considered a controller for the switches which it aggregates. A new level was introduced in order to obtain this functionality, a new API which contains a series of functions. The old *switch API* was kept for all the other pieces of back-end. As one can observe on Figure 2.5, the configurations can be changed using the CLI attached

[1] http://qos.ittc.ku.edu/netlink/html/
[2] OpenWrt is a Linux distribution for embedded devices.
[3] http://whatis.techtarget.com/definition/system-on-a-chip-SoC
[4] http://wiki.libvirt.org/page/Main_Page

to the multiengine, and the switches that compose the multiengine can not be accessed through the CLI.

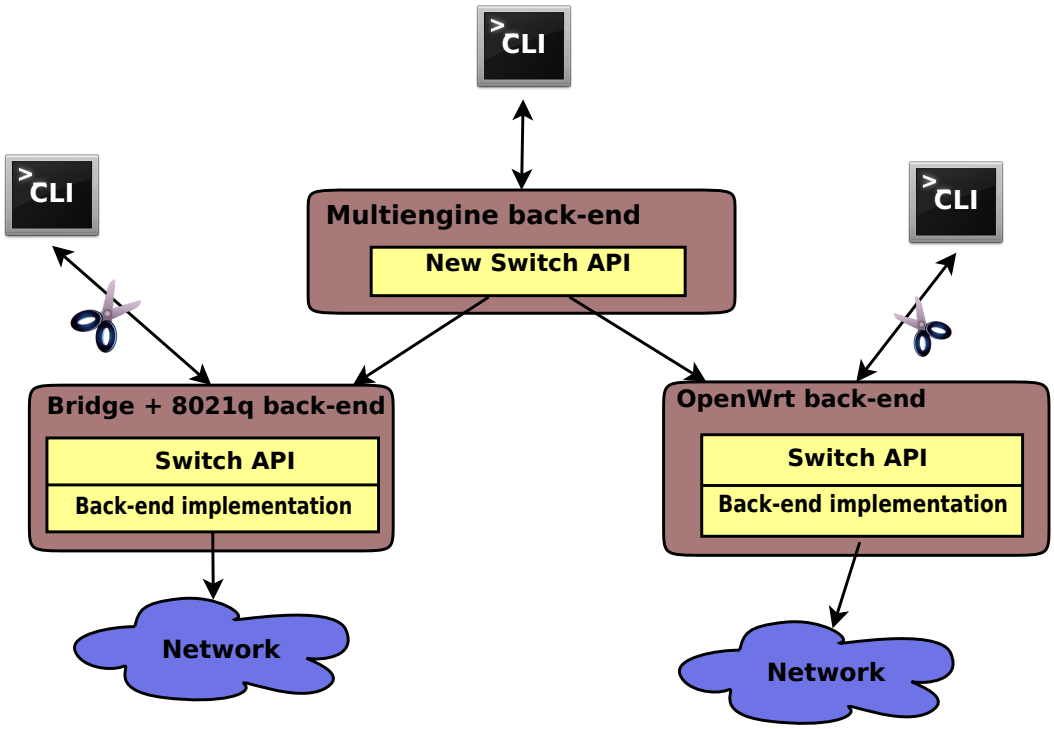

Figure 2.5: Multiengine General Structure

More details about the multiengine architecture and implementation can be found in the chapter dedicated to this subject.

Through the years LiSA implementation improved. From a switch with a back-end dependent implementation it became a generic switch that can be adapted to use multiple back-end implementations (LiSA back-end, Bridge + 8021q back-end, intergration with OpenWrt, usage of LiSA with libvirt for controlling virtual mahcines and the multiengine back-end). Each of these implementations inherited from the original implementation the CLI, but also the protocols CDP and RSTP which had to be adapted in order to function on all the back-end implementations. How the protocols were initially implemented and what needed to be changed is an aspect discussed in the next chapter.

Chapter 3

CDP and RSTP Protocols in LiSA Implementation

The protocols are used in networking to enable a consistent communication between endpoints. A protocol is a set of rules known by all the entities which participate in the process of exchanging messages. Its implementation should not affect the behaviour, the output should be the same.

"The nice thing about standards is that you have so many to choose from."[1]. From those many choices that Andrew Tannenbaum mentioned, for LiSA two of them were chosen to be implemented: CDP and RSTP. For better understanding the implementation details of these two protocols, a brief presentation of their functioning mechanism will be made.

3.1 CDP Overview

CDP is the acronym for Cisco Discovery Protocol. It is a propriety protocol, developed by Cisco as a Data Link Layer protocol[1]. Being a layer two protocol, "two connected systems which run protocols at different network layers can learn about each other" [2]. It runs between network entities directly connected (switches, routers, stations, remote access devices, IP telephones etc.). The purpose of the protocol is to obtain the protocol address and the operating system version of the neighbouring devices. CDP is not routable and can be used only between devices which are directly connected. It is enabled by default on all Cisco equipments.

CDP uses specific messages, named advertisements or announcements, which are sent periodically to a multicast[2] destination address (01:00:0C:CC:CC:CC). The receivers of these packets are any Cisco devices directly connected to the sender or any device that runs CDP on their interface connected to the sender. By default the spacing between

[1] Data Link Layer is the second layer in the seven layer OSI model. In TCP/IP model corresponds to the *link layer*.

[2] An multicast address is used for sending the same message at once to a group of devices from one source.

the advertisements is 60s. The information received in an announcement is kept in a table, in order to learn about the neighbours and determine when the interfaces they expose go up or down. Each entry has a holdtime attached, signifying its lifetime. By default the holdtime is 180s. If no announcements are received in this period from the neighbour, the entry is discarded. On the other hand, when an advertisement is received its holdtime is updated. The information contained by an announcement can vary according to the version of the running operating system or the type of the device.

3.2 RSTP Overview

RSTP is a protocol introduced by IEEE as the standard *802.1w*. Its purpose is to ensure a loop-free topology in bridged networks. The protocol is based upon STP which eventually became obsolete since IEEE standard 802.1D-2004 embedded RSTP. The evolved version of STP, RSTP, ensures backward compatibility with the initial version. The difference between the two of them is that the latter converges faster after topology changes.

The functionality of this protocol is based on discovering the topology of the network using special frames named BPDU(Bridge Protocol Data Unit). These are highly used for initializing the switches and later for verifying if there were any updates in the network. Also, there are multiple states which can be assigned to a switch port:

- *Discarding* - the port is not used for sending data over the network and does not learn MAC addresses.
- *Learning* - at this state MAC addresses learning is enabled, but frame forwarding is disabled. MAC tables are created by analysing the MAC addresses contained by the incoming frames.
- *Forwarding* - the port is fully operational, it receives and forwards data frames and BPDUs, also, it updates the MAC table with MAC addresses.

Each port can be assigned roles by RSTP:

- *Root port* - a forwarding port that is the closest from a root bridge to a non-root one.
- *Designated port* - "A port is designated if it can send the best BPDU on the segment to which it is connected."[3]
- *Alternate and Backup ports* - are used as backup ports for root and designated ports.

To build the spanning tree there are three necessary steps: the root of the tree must be chosen (*root bridge*), followed by the choosing of the root ports and of the active ports.[4]

3.3 CDP and RSTP Implementation for the Initial Architecture of LiSA

To continuously improve the capabilities of LiSA a new feature was considered: usage of network protocols to exchange data with the other devices. Until now two protocols were implemented: CDP and RSTP.

RSTP was implemented as a diploma project by Andrei Faur in 2009 and CDP was implemented by the ones who started the project. Beside the details concerning the implementation of the protocols functionality (which will not be discussed in this chapter), they also had to be linked with the implementation of LiSA.

In order to benefit from the usage of the implemented protocols, the user has to be given access to enable, configure and retrieve information about their running parameters. The CLI is the one which intermediates the communication between the two entities.

The aspects to focus on are: the communication mechanism used between the CLI and the protocols and how the protocols retrieve data from the network. How this is implemented can be observed in the following figure.

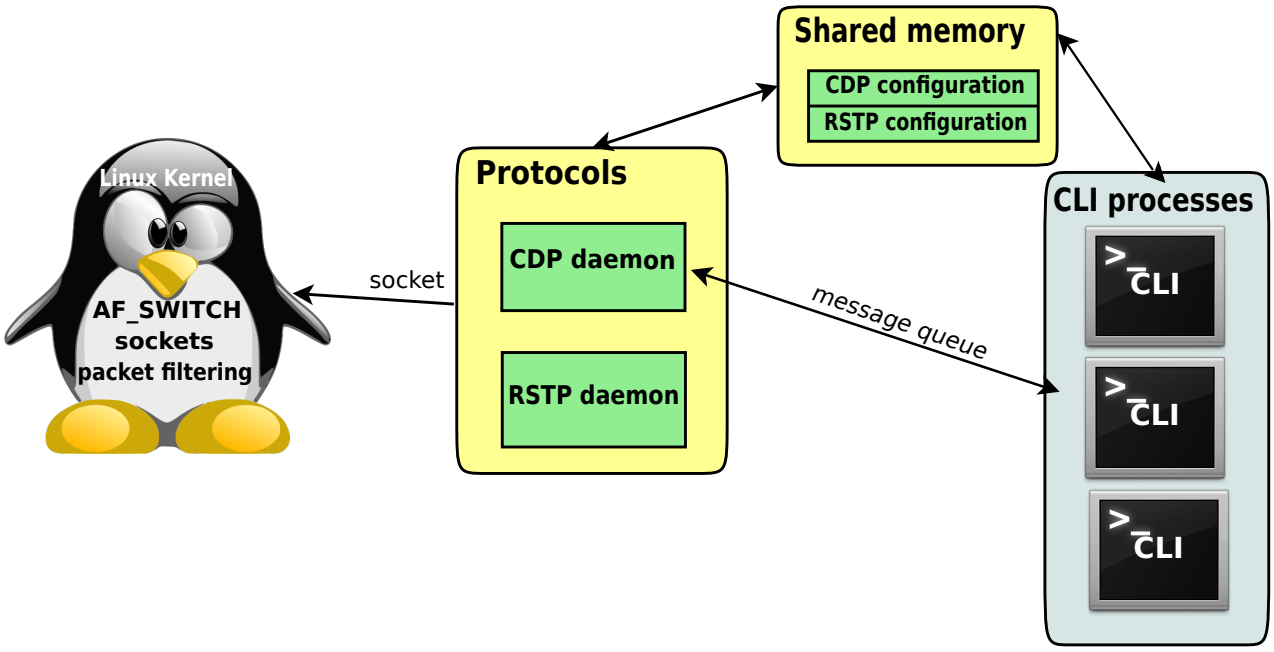

Figure 3.1: Integration of CDP and RSTP with the initial LiSA implementation

The protocols are standalone programs that can be started from the Makefile which builds the application or by running them one by one. It was chosen for the protocols to be daemons[1] because they do not need an attached terminal to enable the access of a user. The two daemons run multiple threads. Each thread is specialized on receiving commands from CLI, sending specific network packets or receiving packets from the network.

[1] A *daemon* is a process that runs in background, does not have a terminal attached and a user cannot directly interact with it.

As mentioned before, multiple LiSA configuration sessions can be opened at the same time on the same station. This type of behaviour has as a side effect the need of sharing certain information between all the active instances. Hence, an IPC method had to be used. As one can observe from Figure 6.1, shared memory was the choice which seemed the most suitable.

```
1  struct shared_memory {
2    /* Enable secrets (crypted) */
3    struct {
4      char secret[SW_SECRET_LEN + 1];
5    } enable[SW_MAX_ENABLE+1];
6    /* Line vty passwords (clear text) */
7    struct {
8      char passwd[SW_PASS_LEN + 1];
9    } vty[SW_MAX_VTY + 1];
10   /* CDP configuration */
11   struct cdp_configuration cdp;
12   /* RSTP configuration */
13   struct rstp_configuration rstp;
14   /* List of interface tags */
15   struct mm_list_head if_tags;
16   /* List of vlan descriptions */
17   struct mm_list_head vlan_descs;
18   /* List of interface descriptions */
19   struct mm_list_head if_descs;
20
21   /* List of VLAN specific data */
22   struct mm_list_head  vlan_data;
23   /* List of interface  specific data */
24   struct mm_list_head  if_data;
25   ........
26 };
```

Listing 3.1: Shared memory information

Beside information about the interfaces, the VLANs or about the passwords, the information about the configuration of a protocol must be the same on all instances. An example of configuration information would be the state of the protocol: enabled or disabled, or other elements specific to the protocol (for example for CDP: version, holdtime). Because any entity can modify these information, exclusive access has to be provided. For LiSA was also implemented a locking mechanism to use shared memory.

But the information from shared memory is not enough for CDP, because one is not interested only in viewing the configuration information. Other interesting pieces of information would be the neighbours or the status of an interface. To ensure the communication with the daemon of the protocol, *message queues*[1] are used. The information is requested by the CLI and the CDP thread dedicated to hande CLI requests will render it and send an appropriate response.

Until now the connection between the CLI and the protocol daemons has been explained. The other important aspect, which created an issue when the architecture was modified to make LiSA a generic switch, is the communication with the kernel space, used for retrieving packets from the network.

[1] *Message queues* are used for interprocess communication. They offer an asynchronous protocol of communication between two endpoints. The messages are placed onto the queue and are stored until the recipient retrieves the message.

The word protocol suggests that a certain behaviour is expected and a certain format of the packets. This principle also applies to CDP and RSTP implementations for LiSA. In order to receive only the packets intended for the protocol the `AF_SWITCH` sockets are used.

```
1  static int setup_switch_socket(int fd, char *ifname) {
2    struct sockaddr_sw addr;
3
4    memset(&addr, 0, sizeof(addr));
5    addr.ssw_family = AF_SWITCH;
6    strncpy(addr.ssw_if_name, ifname, sizeof(addr.ssw_if_name)-1);
7    addr.ssw_proto = ETH_P_CDP;
8
9    if (bind(fd, (struct sockaddr *)&addr,
10   sizeof(addr))) {
11     perror("bind");
12     close(fd);
13     return -1;
14   }
15   fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NONBLOCK);
16   return 0;
17 }
```

Listing 3.2: Setting up a socket for filtering CDP packets

When the socket is attached to a port, it is specified the type of packets to be received using that connection. In this way the packet filtering responsibility rests with the kernel, more precise with the *lisa.ko* module. The filtering is done the same as above for the RSTP. Moreover, RSTP uses *ioctl()* calls over `AF_SWITCH` to obtain information from kernel space. For this kernel dependence it had to be found a solution in order to obtain a generic switch implementation.

3.4 Integrating CDP and RSTP Daemons with LiSA Generic Software Switch

Due to the fact that LiSA became a back-end independent switch, `AF_SWICTH` sockets can be used only when the back-end is *LiSA*. As an alternative for the other implementations *raw sockets* and *library pcap* were used.

3.4.1 Raw Sockets in Linux

Raw sockets are a way of bypassing the sending and receiving of Internet Protocols packets by not specifying any information about transport layer. On standard sockets, the information transmitted is encapsulated according to the specified protocol (e.g. TCP, UDP). On raw sockets no TCP/IP processing is done, the packets are transmitted in a raw form.

The receiver of the packet is responsible for analysing the headers and the content, a job that usually the kernel handles. Raw sockets are supported by POSIX sockets which are available on all Linux distributions. This ensures flexibility for new back-end implementations.

Handling packets through raw sockets implies a great responsibility, because when a packet is sent to the kernel, it does not put any headers. This has to be done explicitly by the sender. A packet with the wrong header is a packet that will, most likely, be dropped or will reach to the wrong destination.

The structure of the packets to be sent is known for the implemented protocols and the information to be encapsulated into the packet is also available at sending time. Having all the necessary elements, raw sockets can be used for sending packets instead of `AF_SWITCH` sockets.

3.4.2 Packet Filtering Using *libpcap*

Because communication in a protocol means sending packets, but also receiving them from the network, a method for handling incoming packets was searched. *Pcap library* was considered a proper solution.

Libpcap is a library which provides a high level interface for capturing system packets. For better understanding how this library can be used for network sniffing[1], the flow of an application will be presented:

1. First, one should establish the name of the interface on which the application will sniff on. The provided name should be a string or if the name of the interface is not known, the library can provide it.
2. Initialize *pcap* by furnishing it the device to sniff on. The device was obtained at the previous step. Multiple devices can be sniffed on at the same time. This is possible because for each session a different handle is obtained.
3. For sniffing only a certain type of traffic a filter[2] should be built. A filter is a string built using terms that the library can recognize. The string must be converted in a format that *pcap* can read, because it is compiled. There is no need of an extra application to compile the filter, because *pcap* provides a function to handle this. Then, the pcap session is informed to use the new created filter.
4. The effective capture is done: a loop can be used to receive a preset number of packets or only one packet can be grabbed.
5. Close the session.

The usage of this library offers an user space alternative for filtering network packets. Hence, it can be integrated with the generic implementation of LiSA. Still, there is a drawback: the flexibility of the filters. The set of elements from which a filter can be composed might not cover all the possibilities. This would lead to receiving all the packets, with no filter attached to *pcap*, and dissect them to analyse if their content matches the expected information.

[1] A *network sniffer* is a computer program that can intercept and/or log traffic passing through a network.

[2] For more details about *pcap* filters the following link can be consulted http://docs.nimsoft.com/prodhelp/en_US/Probes/Catalog/net_traffic/1.3/index.htm?toc.htm?1925170.html

3.4.3 Network Protocols in the Generic Software Switch LiSA

Network protocols implemented for LiSA do not have a separate implementation for each back-end version. The way in which the communication with kernel space is managed has been adapted according to the back-end that uses it.

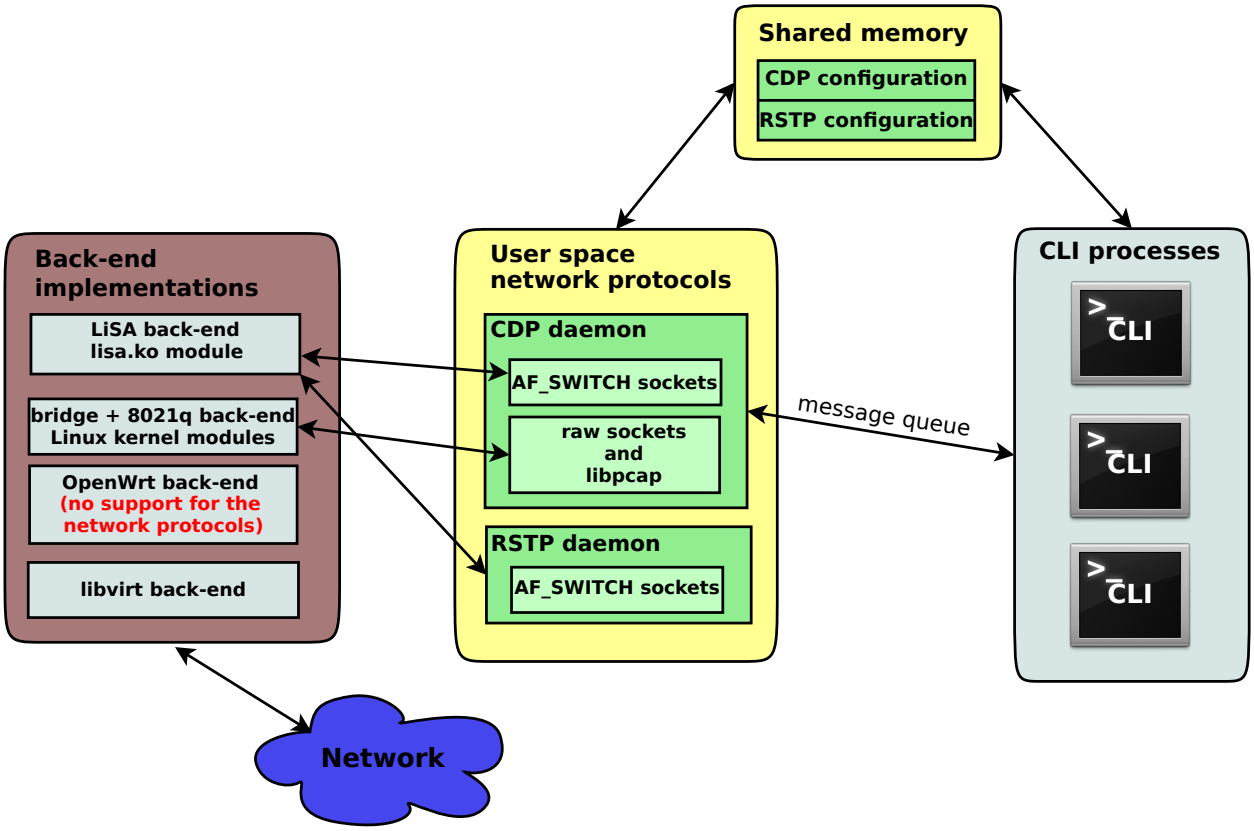

Figure 3.2: CDP and RSTP communication with the generic implementation of LiSA

From Figure 3.2 one can observe that shared memory is preserved without any modification, because the information stored there is not affected by the back-end implementations. If multiple pieces of back-end run at the same time, the memory access is protected using locks, so there will not be race conditions[1].

CDP daemon keeps the message queues to exchange information with the CLI. The modifications brought were at the level of communication between user space CDP implementation and the kernel space. Instead of opening `AF_SWITCH` sockets, raw sockets are used from `AF_PACKET` family. Raw packets can be sent and received to the Data Link Layer (OSI Layer 2). These sockets are used only for sending packets.

A filter is needed in order to receive only packets which are destined to the protocoli. *Pcap library* offers support for filtering networks packets. Due to this feature, it was used to receive and filter CDP packets. The filter was written, using the terminology specific to *pcap* library. The conditions that a packet has to meet to be a CDP packet

[1] A race condition arises when multiple threads or processes attempt to operate over the same shared resource. This behaviour can lead to unexpected results, corrupting the data.

are: the destination Ethernet address must be multicast, the Ethernet source must be different from the station which sends the packet and the protocol ID must correspond to SNAP. The ID is needed because any protocol that supports SNAP[1] within its frame can run CDP.

```
1 #define CDP_FILTER    "ether multicast and ether[20:2] = 0x2000 and ether src not %02
      hx:%02hx:%02hx:%02hx:%02hx:%02hx"
```

Listing 3.3: Pcap filter for CDP

The RSTP protocol is a more complex protocol, because of the states in which the interfaces should be put and the Linux kernel does not offer support for these states. At the moment the protocol can be used with *LiSA back-end*. It can also be implemented for *bridge and 8021q back-end* because the kernel modules with the same name offer support for this protocol. Modifying the already existing implementation for the back-end based on the two above mentioned Linux modules would bring LiSA back to the issue which led to the generic switch implementations in the first place: writing a new implementation for an already existing functionality.

Libvirt back-end is meant to connect LiSA with the virtual machines, to access their interfaces. No switching mechanism is needed to assure this connection. For this reason, the two protocols are not necessary for this particular back-end.

The procedure of adapting the protocol implementations in order to be used with the generic switch architecture of LiSA was different from the one of translating the initial LiSA architecture. Decisions about implementation details had to be made, not about how to redesign the implementations of CDP and RSTP. As a solution, *raw sockets* and *pcap library* were found, but these can not be considered to be universal. Each back-end may need to do further changes in order to obtain the desired behaviour or other pieces of back-end will not require the two protocols.

[1] http://standards.ieee.org/getieee802/download/802-2001.pdf

Chapter 4

Multiengine Back-end Implementation with LiSA

In contrast with the other back-end implementations available for LiSA, which have to define the functions that are contained by `struct switch_operations`, the *multiengine* attaches a new layer to the implementation. Its role is to aggregate multiple pieces of back-end and configure them in a transparent manner. Due to this new layer, the architecture of LiSA has changed, but it maintained the generic aspect.

4.1 Multiengine Architecture

A new layer, the *aggregator* layer, has been added between the CLI and the *switch* API previously created. It will become the new mediator between the two entities. The initial API will be preserved to ensure an uniform access to the pieces of back-end. The *multiengine* was designed taking into consideration the possibility to also control switches implemented with LiSA, located on remote hosts.

The *mulitengine* offers to the user a transparent way for managing multiple switches implemented based on LiSA . There were some elements that had to be considered:

- how will the *multiengine* be aware of what back-end implementations will have to manage
- what happens if multiple switches have the same name for the interfaces, how will be identified
- the management of virtual interface, on which switch will be an interface added

As a solution for identifying the switches that have to be unified, a configuration file has to be filled. There was the option to write a regular file and to impose formatting rules, in order to parse it or to use a known file format whose structure allows a name - value association. The latter proposal was chosen, because the users are familiar with the format and there were already implemented libraries that one can use to generate as well as to parse the configuration file. This solution comes to help the ones who will be using the application, due to the possibility to automate the process of generating the

file, without having to write it from scratch when something changes. More information about the content of the file will be given in Section 4.2 where implementation details are discussed.

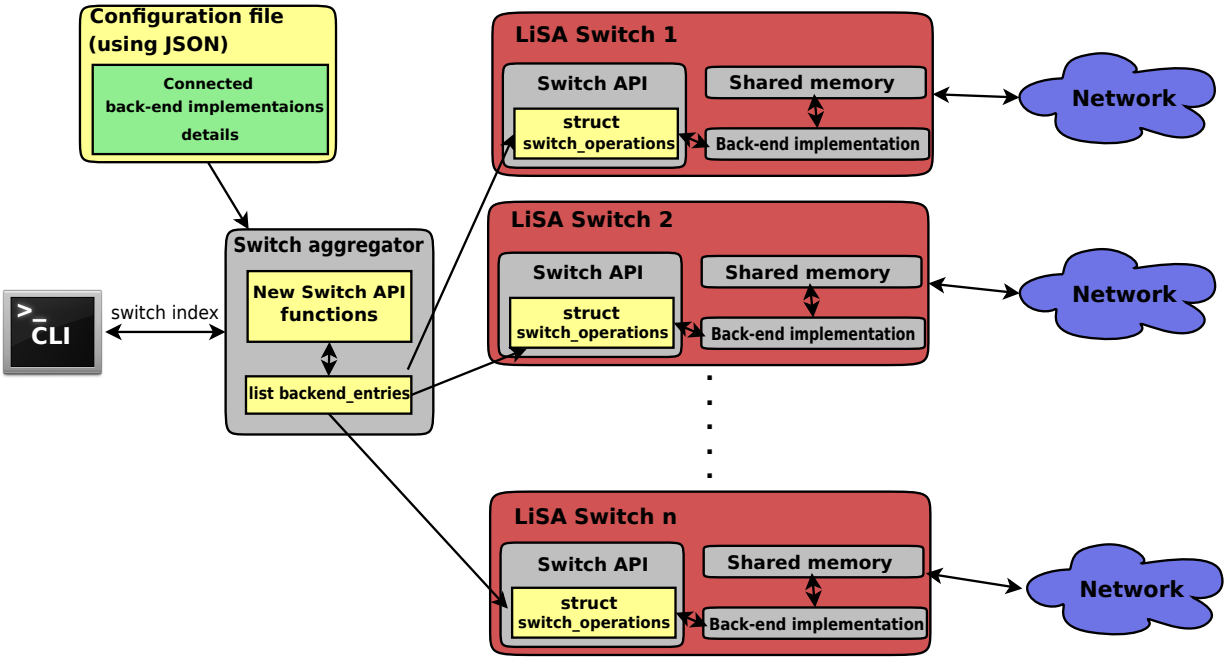

Figure 4.1: Architecture of multiengine back-end for LiSA

Now that the new API has information about the entities to be managed, the list can be passed to the CLI, to display indexes of the switches, in order to inform the user what entities he can configure. The indexing solves the problem regarding multiple interfaces with the same name on different switches. Attaching the number associated to the switch to the name of the interface there will be no confusion because there can not be two interfaces with the same name on the same device.

But beside receiving commands from the CLI, the main purpose of the *multiengine* is the communication with the other back-end implementations. If the switches are located on the same device as the *multiengine*, there is no need to use `sockets` or other means of communication. The shared object obtained for the implementation is available on the same device and can be accessed directly(as can be observed from Figure 4.1). For the remote host, connection parameters to the host/device on which the back-end implementation resides can be found in the configuration file.

For this back-end in particular, no configuration information is kept in the shared memory. It only commands the other back-end implementations and they will manage information in their own specific way, without any intervention from the outside.

The *aggregator* API will substitute the *switch* API. Hence, through it, all the back-end implementations will be managed. Even if there exists only one back-end implementation, it will be registered to the *multiengine* and it will be the only instance controlled.

4.2 Multiengine Implementation

One of the reasons for not preserving the initial API is that even if it was generic, it was not flexible for handling multiple back-end implementations at the same time. The *switch* API functions, which are members of `struct switch_operations`, receive as a parameter a pointer to the structure with the same name. The structure received as a parameter contains pointers to the functions implemented using the features that the back-end provides. Maintaining this behaviour, the CLI would have been not only a command executer, it would have become also a manager of the unified back-end implementations. It would have to determine the destination switch for a command and this means more than being a command processor. To maintain a clear delimitation between the layers of the generic switch implemented using LiSA, the *aggregator* layer was introduced.

4.2.1 CLI Changes

The responsibilities of the *switch API* are taken by the new layer, which materialized into a set of functions that are homonym to the ones defined for the previous API. The difference stands in their behaviour. The ones which are part from the new mediator between CLI and the back-end implementations, can be considered as a connection multiplexor: they redirect the received commands to the intended addressee. The differences between the two versions of API can be observed from the CLI commands, but also from the definition of the functions which parse them.

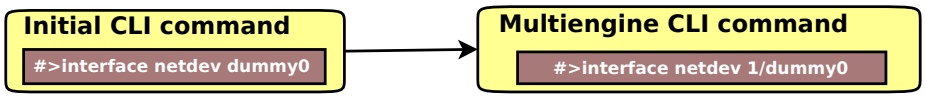

Figure 4.2: Differences between the CLI commands

As shown in Figure 4.2, the format of the CLI commands also had to be adapted to support the *multiengine back-end*. For the ones that can be directed only to a certain entity, the index associated to a certain switch implementation has to be mentioned. If it is omitted, it will be considered by default the switch with index '0'. A default behaviour was chosen for virtual interface adding. This type of interfaces are added only on the first back-end implementation which uses *bridge and 8021q* kernel modules.

4.2.2 Multiengine Back-end Initialization

Beside the API, other specific element of the discussed implementation is the capability to manage multiple back-end implementations. As mentioned in Section 4.1, a configuration file is used, to be more specific, a JSON type file. For parsing it was used *cJSON*[1], an API which provides the basic functions to extract he nodes and their content, but also to write a file having this type. The elements are extracted from the

[1] http://cjson.sourceforge.net/

nodes according to their type. By extracting their values this way, it was facilitated the handling of the elements sent to the output by the cJSON API functions.

The parsing of the file is made only once, at the moment when the application is launched. There is a function that initializes the structures of the *multiengine* and also triggers the parsing process. It has the attribute *constructor* and this forces the function to be executed before the function `main()`.

The information associated with a certain back-end implementation is stored in a structure, which has as members the fields that can be found in the configuration file. An example of configuration file can be found in the appendix. A linked list of all the structures that correspond to different pieces of back-end it is kept. Each implementation has certain interfaces which correspond to it. Because the number of interfaces can vary, a linked list was chosen to memorize this association. The content of the above mentioned structure is captioned in the following listing.

```
1  struct switch_interface {
2    char if_name[MAX_NAME_SIZE];
3    struct list_head lh;
4  };
5
6  struct backend_entries {
7    int sw_index;
8    char port[MAX_NAME_SIZE];
9    char ip[MAX_NAME_SIZE];
10   char type[MAX_NAME_SIZE];
11   char locality[MAX_NAME_SIZE];
12   struct switch_operations *sw_ops;
13   struct list_head if_names_lh;
14   struct list_head lh;
15 };
```

Listing 4.1: Multiengine structures

As mentioned before, the switch index will play an important part because it will be the only way to distinguish between two interfaces with the same name from two different pieces of back-end. The `port` and `ip` members of the structure `backend_entries` are initialized only when the back-end is on a remote host. The `locality` field is used for specifying whether the shared object associated with an implementation of back-end is available on the same device as the *aggregator* (the value of locality is set to 'local') or on a remote device (the value of locality is set to 'remote').

For obtaining the value of `sw_ops` field the parsing of the configuration file was not the only prerequisite. The `switch_operations` structure is specific to the *switch* API, that was preserved for the back-end implementations. By exposing this API, the multiengine can access the functions implemented for a specific back-end.

The definitions of the *switch* API functions are concentrated in a shared object, that in the former generic switch implementations using LiSA was linked to the CLI module. Because this connection was superseded, a new manner of accessing them had to be found. For the case when the field `locality` from the structure `backend_entries` has the value 'local', `dlopen`[1] function is used. In this way the shared object is made available through a handle to the calling program.

[1] http://pubs.opengroup.org/onlinepubs/009695399/functions/dlopen.html

After obtaining access to the definitions contained by the library, the pointer to the API functions has to be obtained. This is possible due to a side effect of opening the library, the initialization function of the shared object is called, which has associated the attribute *constructor*.

Its responsibility is to extract the pointer to the `switch_operations` structure that stores the context of a certain back-end implementation. The context contains details specific to the back-end, but all contexts have in common the pointer to the API functions. Beside being saved in the context structure, it is declared as an extern global variable, named `sw_ops`. Its attributes extern and global make possible to be obtained by calling `dlsym()` function at the moment when the *multiengine* back-end is initialized. The earlier mentioned function looks-up variables or functions from a shared object and returns the address where symbol is loaded. The pointer obtained by calling `dlsym()` functions is stored in the structure associated to the back-end implementation.

Going further with analysing the members of the `backend_entries` structure, the `if_names_lh` element can be observed. It is the head of the list which contains the names of the interfaces associated with the back-end implementations. The names are extracted from the configuration file. The structure used for storing the information about an interface is named `switch_interface` and it only contains the name of the interface and the link to the next member of the list.

The last member of the structure `backend_entries`, `lh`, is used for ensuring the linkage between the pieces of information about each back-end implementation described in the configuration file. The type of the element is `struc list_head`, the same as the one used for kernel lists. This is not a coincidence, the kernel lists API was also implemented in user space for LiSA, facilitating list operations.

Saving the information it is done by parsing only once the configuration file. The data is saved as the file is parsed node by node, including the pointer to the implementation of the *switch API*.

After this initialization step, all the information is available to the *aggregator* API functions and can be used for implementing the desired functionalities.

4.2.3 Aggregator API Implementation

The *aggregator* API implementation is not back-end dependant. Each API function has the purpose to redirect the received command to the corresponding back-end. A reference to the list containing the back-end details is held. When a function is called, according to the received parameters, a look-up is made to determine the switch to which the command is addressed. The headers of the API functions can be found in Appendix C.

The functions can be grouped into categories, according to the targeted entities:

- Functions for configuring interfaces
- Functions for VLAN management

- Functions for configuring the protocols
- Functions for general switch configurations

A function that configures interfaces has to direct the command to a certain switch. The addressee is determined using the index of the switch received as a parameter. The index can be set to a default value, this indicating that the targeted LiSA switch is the first one from the list kept by the *aggregator* layer. Otherwise, it is looked-up the switch that has the index received as parameter. To obtain the desired configuration, it is called the corresponding *switch* API function from `struct swicth_operations`.

```
1  int if_add(int sw_index, char *if_name, int mode)
2  {
3    int if_index, sock_fd;
4    struct backend_entry *sw_entry;
5
6    /* extract switch entry from the linked list */
7    sw_entry = get_switch_entry(index);
8
9    if_index = if_get_index(if_name, sock_fd);
10
11   return sw_entry->sw_ops->if_add(sw_entry->sw_ops, if_index, mode);
12 }
```

Listing 4.2: Implementation of the function for interface adding from *aggregator* API

In Listing 4.2, the interface adding functionality is described. The function `get_switch_entry` is responsible for searching the switch associated with a certain index. Because the interface is identified by name and the switch API functions receive the index of the interface, the function `if_get_index` is used to make this translation. After obtaining all the necessay elements, the corresponding *switch* API function is called.

There is a particular type of interfaces that are added only on LiSA switches implemented using the kernel modules *8021q.ko* and *bridge.ko*: the virtual interfaces. This approach was chosen because these modules use the interfaces for inter-vlan routing.

The commands for VLAN management are not dependant of a certain switch and can be considered broadcast because are sent to all the switches that are part of the *multiengine*. For adding a VLAN, the list of switches is iterated through and for each is called the function `vlan_add`. If an error ocurrs while adding a VLAN on a certain switch, the ones that were successfully added are deleted from the corresponding switches. A snippet of code illustrating the mentioned functionality can be found in Listing 4.3:

```
1  int vlan_add(int vlan)
2  {
3    int status, sw_idx;
4    struct backend_entry *entry;
5
6    status = 0;
7
8    list_for_each_entry(entry, &head_sw_ops, lh) {
9      status = entry->sw_ops->vlan_add(entry->sw_ops, vlan);
10     if (status != 0) {
11       sw_idx = entry->sw_idx;
12       goto del_vlan;
13     }
14   }
15
```

```
16 del_vlan:
17   list_for_each_entry(entry, &head_sw_ops, lh) {
18     if (entry->sw_index >= sw_idx)
19       break;
20
21    entry->sw_ops->vlan_del(entry->sw_ops, vlan);
22   }
23
24   return status;
25 }
```

Listing 4.3: Implementation of the function for vlan adding from *aggregator* API

The instruction `list_for_each_entry` from Listing 4.3 is a macro definition that is part of the API for handling the linked lists, which is similar to the one used in the Linux kernel.

The functions used for showing the configuration of the switch, will in fact gather information from all the switches that are part of the *multiengine*. The pieces of information are unified and displayed to the user. For implementing this functionality, the list of switches is iterated through and a specific function is called, according to the information that is necessary.

To configure the networking protocols (CDP and RSTP), there is no need to identify a certain switch. All the components of the *multiengine* are set up with the same parameters. The implementation is similar to the one used for managing VLANs.

The *multiengine* back-end brought significant changes to the architecture of LiSA by introducing the *aggregator* layer. It can be used when a single back-end implementation is available, but also with multiple back-end implementations. Due to the multiple back-end support, changes had to be done in order to adapt the CLI implementation to the new format of the commands. The implementation encapsulated by the API functions does not need to be changed when other pieces of back-end are added. This module will be a start point for developing further the project.

Chapter 5

Testing and Results

To be able to test the functionality of the *multiengine* back-end, the component switches must provide the desired behaviour. Beside being able to run as a standalone switch, each LiSA switch, no matter the nature of the back-end, should be able to expose the desired functionality even when it is linked to another switch implemented using LiSA.

Before verifying the compatibility between two different types of back-end, unit testing was performed for each back-end. The unit tests were written along with the implementation of the functions for the LiSA back-end. It was convenient manner of testing the implemented functionalities without integrating it yet with the CLI. The unit tests can be considered a simplified Command Line Interface, the input commands being an one to one association with the switch API functions. For accessing the tests, *swctl* executable should be run (it is available with the project). In Appendix D a snippet of code containing the implementation of the test which verifies interface adding can be found.

After verifying that the implementation of the *switch* API functions it works, a compatibility test it was made. The purpose of the test was to verify that multiple LiSA switches with different back-end implementations can work together.

The back-end implementations used for this test were *bridge + 8021q* back-end (implemented using two Linux kernel modules) and *LiSA* back-end (based on *lisa.ko* kernel module). The first of them was installed on a workstation on which Debian OS was running, with the kernel version 3.2. The latter was installed on a CentOS, with the kernel version 2.6. To be able to run the back-end implemented using *lisa.ko* module, additional packages had to be installed.[1]

The stations used for hosting the switches were equipped with two network cards: one of them having the capacity of 100Mbit/s and the other of 1Gbit/s.

The topology of the network used for testing contains three switches: two of them running a LiSA switch that uses *bridge + 8021q* back-end and the other runs a LiSA switch with *LiSA* back-end. There are also four stations linked as can be seen in the next figure:

[1] The packages and the install steps can be found at the following address:https://github.com/lisa-project/lisa-user/wiki/Install-and-run-LISA-using-kernel-rpm-packages

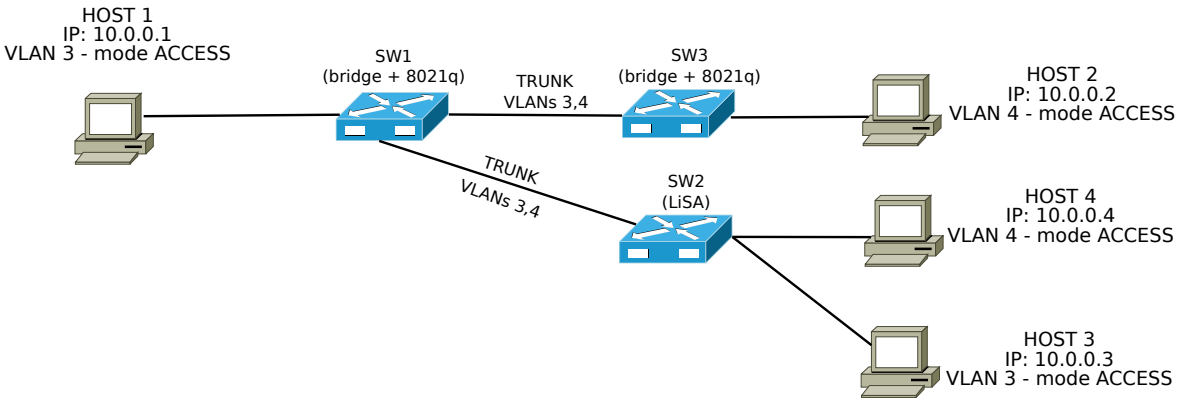

Figure 5.1: Topology of the network used for testing

To test all the features that LiSA can offer, VLANs where configured on hosts and on the interfaces associated with the switches. The inter-switch links were configured in `trunk` mode. The VLANs allowed were VLAN 3 and 4. The interface of the host connected to a certain switch was added into a VLAN an set into `access` mode. The hosts were also given IP addresses from the same network to enable the communication between them, using only a Data Link Layer device.

After running the *swcli* executable on the stations on which LiSA switches were installed, the configurations were made: the interfaces were added to the switch, the VLANs were also added and the next step was to put the interfaces in trunk mode as well as to establish what VLANs are allowed on the trunk links.

With the switches configured, `ping` utility was used to test the connectivity between the hosts. This enabled the process of learning the MAC addresses on each switch. In Table 5.1 the output of the command `show mac-address-table` is displayed.

Table 5.1: Mac address table on switch number two (SW2)

Destination Address	Address Type	Vlan	Destination Port
00e0.208c.01ed	Dynamic	4	eth2
00e0.2082.44d9	Dynamic	4	eth2
00e0.2082.44d9	Dynamic	3	eth2
7071.bc18.1c1c	Dynamic	4	eth2
7071.bc18.1bff	Dynamic	3	eth2
7071.bc18.1bff	Dynamic	4	eth2
7071.bc08.2588	Dynamic	4	eth4
7071.bc08.257e	Dynamic	3	eth3

The switch learnt the destination address and the port that should be used for sending a packet to a certain address. As one can observe from the table, one MAC address is listed twice because the trunk port can be used for sending packets in VLAN 3, but also in VLAN 4.

Moving a step further, `iperf3`[1] utility was used to determine the bandwidth perfor-

[1] https://code.google.com/p/iperf/

mance. For testing two hosts were necessary, hence `HOST 1` and `HOST 3` were chosen, because it was desired that the packets pass through two switches that run two different pieces of back-end. In order to transmit a packet from `HOST 1` to `HOST 3`, SW1 and SW2 are the ones that intermediate this connection. One host acted as a server (HOST 1) and the other as a client (HOST 3). The server received the connection and the result of the analysis was outputted on the client side.

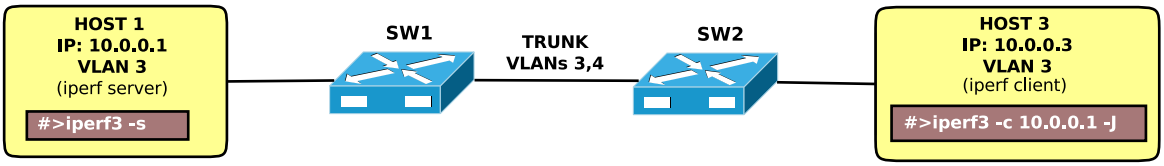

Figure 5.2: *iperf3* usage on hosts

In Figure 5.2, one could observe the commands that were ran on each station. On `HOST 3`, the `'-c'` parameter indicates that the host is a client and it must be followed by the IP of the server. It was preserved the default time to transmit: 10s. The `'-J'` parameter requests the output in the JSON format.

The result was that 120586240 Bytes were sent in 10.1821 seconds and the number of bits transmitted per second 94.7436Mbit/s. The following snippet is part of the output in JSON format. Also it can be observed the CPU usage on the localhost, but also on the remote host.

```
"streams":  [{
    "sent": {
      "socket": 4,
      "start":  0,
      "end":  10.1821,
      "seconds":  10.1821,
      "bytes":120586240,
      "bits_per_second": 9.47436e+07,
      "retransmits": 0
    },
    "received":
    {
      "socket": 4,
      "start":  0,
      "end": 10.1821,
      "seconds": 10.1821,
      "bytes": 120324096,
      "bits_per_second": 9.45377e+07
    }
  }],
"cpu_utilization_percent":
{
  "host" : 7.69756,
  "remote" : 3.7996
}
```

Listing 5.1: Bandwith performance analysis using *iperf3*

There was also another testing topology that involved two systems with LiSA switches using *lisa.ko* module and to hosts. Between the switches there was a trunk connection. The transfer speed obtained by using *iperf3* was similar to the one obtained with the topology described using Figure 5.1: 95,0919Mbit/s.

These tests have proved that multiple pieces of LiSA back-end can be used together, independent of the nature of the back-end: all the switches can use the same back-end implementation or different ones. Due to the compatibility demonstrated using these tests, LiSA can be considered as an option when it comes to choosing a switching device for small networks. Moreover, it comes as a confirmation that the *multiengine* back-end has applicability and it is adaptable, being able to integrate new back-end implementations that are compatible with the ones which exist.

Chapter 6

Conclusion and Further Work

Tom Landry said that: "Setting a goal is not the main thing. It is deciding how you will go about achieving it and staying with that plan." One of the goals of this project derived from the need to redesign the architecture of the initial LiSA implementation. This redesign meant introducing a new layer to mediate the communication between the CLI and the back-end implementation. The mediator is in fact an API, shaped as a data structure that contains pointers to functions, which will be implemented by each back-end.

But the CLI and the kernel module were not the only modules that composed the initial LiSA architecture. There were also the CDP and RSTP daemons, used for assuring the functionality of the network protocols implemented for LiSA. The goal was to be able to use the protocols with other back-end implementations, beside the one based on *lisa.ko* kernel module.

CDP implementation was adapted to use raw sockets and *pcap library* when it is linked with other pieces of back-end beside LiSA. For RSTP, the kernel modules *8021q.ko* and *bridge.ko* already expose functions through which its functionality can be implemented. Adapting this protocol for *bridge + 8021q* back-end can be further developed. Changing the current implementation of the protocol would mean implementing a functionality that can already be provided using other means.

Now that the switch API layer was introduced, one can choose what back-end implementation to use. The other possibilities of back-end were taken into consideration, when the process of adapting the protocols was started: the one based OpenWRT and the one using *libvirt* API.

The OpenWrt back-end is not compatible with the two protocols and the one that uses *libvirt* does not do switching related activities, it only enables LiSA to connect to a virtual machine, to access its interfaces. The back-end implementations mentioned above did not receive yet the final touch. They are still a work in progress.

Having all these back-end possibilities to choose from, a step further would be accessing all of them through the same CLI, without having to configure each of them independent. This led to the second goal of the project: to design and implement a back-end named *multiengine*. A new layer was introduced between the CLI and the *switch* API, named

aggregator layer. This layer is implemented using a set of functions that will dispatch the received commands to the intended receivers. The introduced layer does not change the behaviour of LiSA as a generic switch.

Beside the API, a configuration file having the type JSON is used. It contains the description of each back-end that will be part of the back-end. The API functions use the data extracted from the configuration file to command the component switches.

Changes had to be brought to the implementation of the CLI because the format of the commands (especially those referring to interfaces) was modified. At the moment, not all the CLI functions are supported by the *multiengine* back-end. Another aspect to be mentioned is that it only offers support for the back-end implementations which are located on the same workstation as the *multiengine*.

The next step is to offer support for all the CLI functions using the *multiengine* back-end and to be able to handle remote back-end implementations. An architecture proposal for this case can be seen in the following figure:

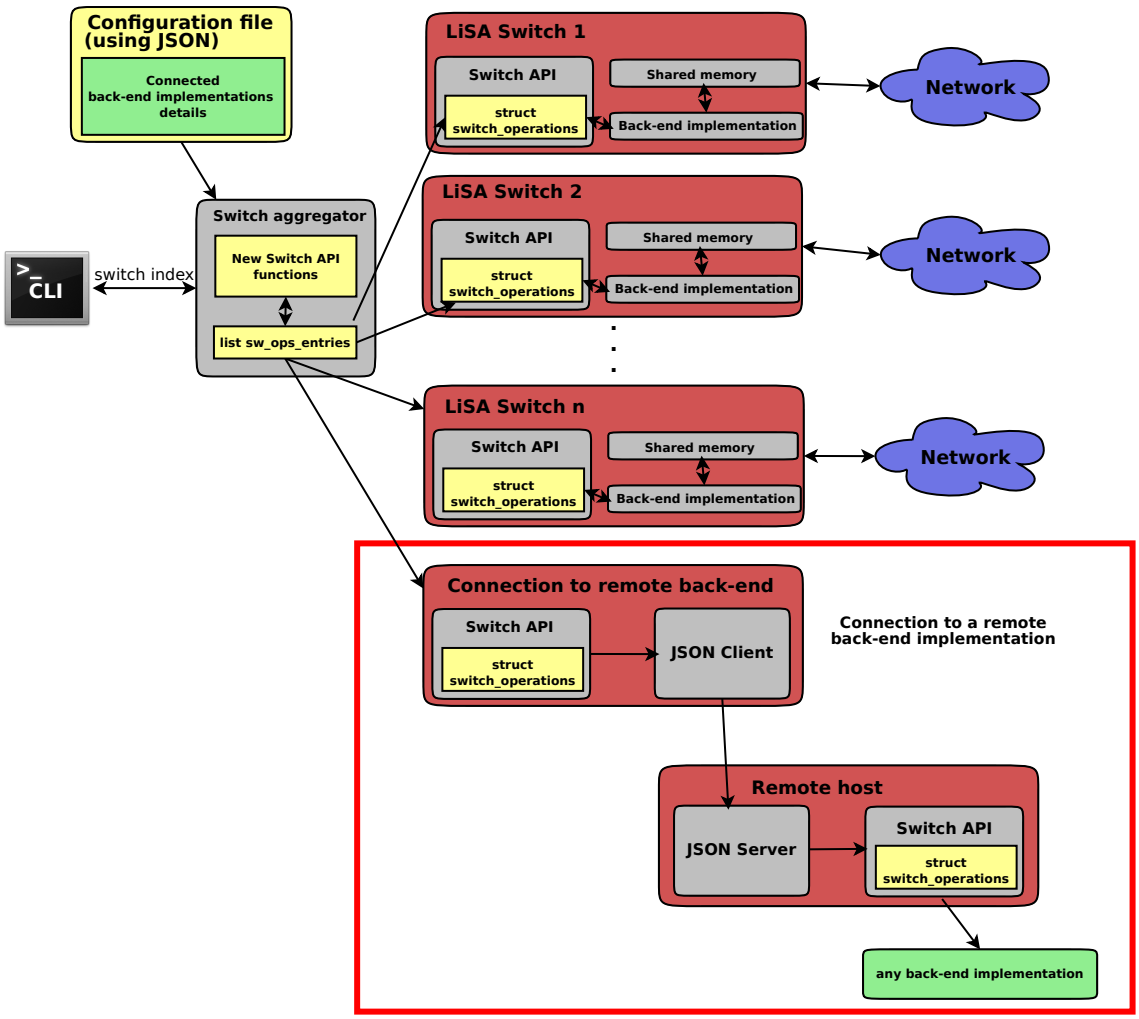

Figure 6.1: Multiengine architecture for handling remote switches

As one can observe, a new back-end can be introduced, used only for connecting to remote hosts, which has to implement the switch API functions. It has an associated JSON client used for connecting to a JSON server on the remote host. The JSON server will transfer the commands to the API located remote.

The *multiengine* will be the base component used for all the back-end implementations that are going to be developed based on LiSA. The generic architecture of LiSA makes it adaptable to new back-end implementations, according to the needs of the users.

Appendix A

The API of the Generic Software Switch Based on LiSA

```
1  struct switch_operations {
2          int (*backend_init) (struct switch_operations *sw_ops);
3          int (*if_add) (struct switch_operations *sw_ops, int ifindex, int mode);
4          int (*if_remove) (struct switch_operations *sw_ops, int ifindex);
5          int (*if_set_mode) (struct switch_operations *sw_ops, int ifindex, int mode,
               int flag);
6          int (*if_set_port_vlan) (struct switch_operations *sw_ops, int ifindex, int
               vlan);
7          int (*if_get_cfg) (struct switch_operations *sw_ops, int ifindex, int *flags,
                int       *access_vlan, unsigned char *vlans);
8          int (*if_get_type) (struct switch_operations *sw_ops, int ifindex, int *type,
                int *vlan);
9          int (*if_enable) (struct switch_operations *sw_ops, int ifindex);
10         int (*if_disable) (struct switch_operations *sw_ops, int ifindex);
11         int (*if_clear_mac) (struct switch_operations *sw_ops, int ifindex);
12         int (*if_add_trunk_vlans) (struct switch_operations *sw_ops, int ifindex,
               unsigned char *vlans);
13         int (*if_set_trunk_vlans) (struct switch_operations *sw_ops, int ifindex,
               unsigned char *vlans);
14         int (*if_del_trunk_vlans) (struct switch_operations *sw_ops, int ifindex,
               unsigned char *vlans);
15         int (*get_if_list) (struct switch_operations *sw_ops, int type, struct
               list_head *net_devs);
16         int (*vlan_add) (struct switch_operations *sw_ops, int vlan);
17         int (*vlan_del) (struct switch_operations *sw_ops, int vlan);
18         int (*vlan_set_mac_static) (struct switch_operations *sw_ops, int ifindex,
               int
19                         vlan, unsigned char *mac);
20         int (*vlan_del_mac_static) (struct switch_operations *sw_ops, int ifindex,
               int vlan, unsigned char *mac);
21         int (*vlan_del_mac_dynamic) (struct switch_operations *sw_ops, int ifindex,
               int vlan);
22         int (*get_vlan_interfaces) (struct switch_operations *sw_ops, int vlan, int
               **ifindexes, int *no_ifs);
23         int (*get_vdb) (struct switch_operations *sw_ops, unsigned char *vlans);
24         int (*vif_add) (struct switch_operations *sw_ops, int vlan, int *ifindex);
25         int (*vif_del) (struct switch_operations *sw_ops, int vlan);
26         int (*igmp_set) (struct switch_operations *sw_ops, int vlan, int snooping);
27         int (*igmp_get) (struct switch_operations *sw_ops, char *buff, int *snooping)
               ;
28         int     (*mrouter_set) (struct switch_operations *sw_ops, int vlan, int
               ifindex, int setting);
29         int (*mrouters_get) (struct switch_operations *sw_ops, int vlan, struct
               list_head *mrouters);
```

```
30          int (*get_mac) (struct  switch_operations  *sw_ops,      int  ifindex, int
                vlan, int mac_type, unsigned char *optional_mac, struct list_head *macs);
31          int     (*get_age_time) (struct  switch_operations *sw_ops, int *age_time);
32          int     (*set_age_time) (struct switch_operations *sw_ops, int age_time);
33  }
```

Listing A.1: Switch API

Appendix B

Example of Configuration File Used for Multiengine Back-end Implementation

In order to be able to identify the switches that are going to be part of the multiengine, a configuration file is provided. The format JSON it was chosen because of its standard structure. In Listing B.1 it is an example of such a file, with two switches located on the same system as the multiengine (one switch uses LiSA back-end and the other uses *bridge + 8021q* back-end, also named Linux) and one located remote.

```
{
        "comment"       : "The␣first␣backend␣should␣_always_␣be␣_linux_
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣to␣be␣able␣to␣add␣virtual␣interfaces␣on␣it.␣**Ignore**␣this␣object␣
    when
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣parsing␣the␣json␣file.",
        "backend_objects" : [
                                {
                                        "type"          : "lisa",
                                        "shared_object" : "liblisa.so",
                                        "locality"      : "local",
                                        "interfaces"            : [
                                        { "if_name"     : "dummy3"},
                                        { "if_name"     : "dummy0"},
                                        { "if_name"     : "dummy1"}
                                        ]
                                },
                                {
                                        "type"                  : "linux",
                                        "shared_object" :       "libswitch.so",
                                        "locality"              : "local",
                                        "interfaces"    :[
                                        { "if_name"             : "dummy0"},
                                        { "if_name"             : "dummy1"},
                                        { "if_name"             : "dummy2"}
                                        ]
                                },
                                {
                                        "type"                  : "lisa",
                                        "shared_object" : "libswitch.so",
                                        "locality"              : "remote",
                                        "ip"                    : "192.168.101.23",
                                        "port"                  : "5598",
                                        "interfaces"    : [
```

```
33                                          { "if_name"             : "dummy2"},
34                                          { "if_name"             : "dummy3"},
35                                          { "if_name"             : "dummy4"}
36                                  ]
37                                  }]
38  }
```

Listing B.1: Configuration File for Multiengine Back-end

Appendix C

Aggregator API Functions

For the *multiengine* back-end, a new layer was introduced named the *aggregator*. It materialized in a set of functions implemented by this back-end. The headers of the functions are presented in the following listing:

```
1
2  int if_add(int sw_index, char *if_name , int mode);
3
4  int if_remove(int sw_index, char *if_name);
5
6  /**
7   * @param mode   Switchport mode: access or trunk.
8   * @param flag Specifies if the mode is set to on(flag = 1) or off (flag = 0)
9   */
10 int if_set_mode(int sw_index, char *if_name, int mode, int flag);
11
12 int if_set_port_vlan(int sw_index, char  *if_name, int vlan);
13
14 /**
15  * @param vlans  Vlans are returned using bitmap positive logic.
16  */
17 int if_get_cfg(int sw_index, char *if_name , int *flags, int *access_vlan, unsigned
       char *vlans);
18
19 /**
20  * @param mode This parameter will return as side effect the type of the
21 interface:
22  * ethernet or a virtual vlan interface. In case of vlan interface, also
23  * return the vlan as side effect.
24  */
25 int if_get_type(int sw_index, char *if_name, int *type, int *vlan);
26
27 int if_enable(int sw_index, char *if_name);
28 int if_disable(int sw_index, char *if_name);
29
30 int if_clear_mac(int sw_index, char *if_name);
31
32 int if_add_trunk_vlans(int sw_index, char *if_name, unsigned char *vlans);
33 int if_set_trunk_vlans(int sw_index, char *if_name, unsigned char *vlans);
34 int if_del_trunk_vlans(int sw_index, char *if_name, unsigned char *vlans);
35
36 int get_if_list(int type, struct list_head *net_devs);
37
38 /**
39  * Add vlan to vlan database.
40  *
41  * Important: Vlan description is held within the shared memory
```

```
42   segment. If vlan_add returns successfully, call switch_set_vlan_desc
43   (see  $LiSA_USER_HOME/tools/swctl.c for an example). */
44  int vlan_add(int vlan);
45
46  /**
47   * Remove a vlan from vlan database.
48   *
49   * Important: If vlan_del returns successfully, don't forget
50   to also
51   * remove the description using switch_del_vlan_desc() (see
52   vlan_add for further information).
53   */
54  int vlan_del(int vlan);
55  int vlan_set_mac_static(int sw_index, char *if_name, int vlan, unsigned char *mac);
56
57  /**
58   * @param mac   MAC in binary format.
59   */
60  int vlan_del_mac_static(int sw_index, char *if_name, int vlan, unsigned char *mac);
61  int vlan_del_mac_dynamic(int sw_index, char *if_name, int vlan);
62
63  int get_vlan_interfaces(int sw_index, char *if_name, int **ifindexes, int *no_ifs);
64
65  int igmp_set(int sw_index, char *if_name, int vlan, int snooping);
66  int igmp_get(int sw_index, char *if_name, char *buff, int *snooping);
67
68  /**
69   * Return a VLAN bitmap.
70   * VLAN descriptions can be obtained using  swicth_get_vlan_desc().
71   * @param vlans  VLAN bitmap.
72   */
73  int get_vdb(int sw_index, char *if_name, unsigned char *vlans);
74
75  int mrouter_set(int sw_index, char *if_name,  int vlan, int ifindex, int setting);
76
77  /* Return a list of net_switch_mrouter_e. */
78  int mrouters_get(int sw_index, char *if_name, int vlan, struct list_head *mrouters);
79
80  /* Return a list of net_switch_mac_e. */
81  int get_mac(int sw_index, char *if_name, int ifindex, int vlan, int mac_type,
        unsigned char *optional_mac, struct list_head *macs);
82
83  int get_age_time(int sw_index, char *if_name, int *age_time);
84  int set_age_time(int sw_index, char *if_name, int age_time);
85
86  int vif_add(int vlan, int *ifindex);
87  int vif_del(int vlan);
```

Listing C.1: Aggregator API functions

Appendix D

Unit Testing Tool

For unit testing *swctl* executable is used. It receives as parameters when it is executed the name of the command and the command parameters.

As one can be observe from Listing D.1, the received parameters are parsed to identify the command. Because the switch API function receives the index of the interface, not the name, `if_get_index` is called to obtain its index. After having all the necessary parameters the API function is called. By default, a interface is added in the switch in mode trunk, hence the presence of the parameter `IF_MODE_ACCESS` for `if_add` function. The function returns a status that is used to determine if the execution ended successfully or not.

```
1  ...
2
3  if (!strcmp(argv[1], "add")) {
4    int if_index;
5    if (argc < 3) {
6      usage();
7      return 0;
8    }
9
10   if_index = if_get_index(argv[2], sock);
11   status = sw_ops->if_add(sw_ops, if_index, IF_MODE_ACCESS);
12
13   if(status)
14       perror("add_failed");
15
16   return 0;
17 }
18
19 ....
```

Listing D.1: Test for interface adding

Bibliography

[1] David J. Wetherall and Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 5th edition, October 2010.

[2] Cisco. *Cisco IOS Configuration Fundamentals Configuration Guide*. http://www.cisco.com/en/US/docs/ios/12_2/configfun/configuration/guide/fcfbook.pdf, April 2010.

[3] Cisco. *Understanding Rapid Spanning Tree Protocol (802.1w)*. http://www.cisco.com/image/gif/paws/24062/146.pdf, October 2006.

[4] Răzvan Rughiniș, Răzvan Deaconescu, Andrei Ciorba, and Bogdan Doinea. *Rețele locale*. Printech, 2009.