A Didactic Platform for Testing and Developing Routing Protocols

Adam Kaliszan
Chair of Communication and Computer Networks
Poznan University of Technology
ul. Polanka 3, 60-965 Poznań, Poland
Email: adam.kaliszan@gmail.com

Mariusz Głąbowski
Chair of Communication and Computer Networks
Poznan University of Technology
ul. Polanka 3, 60-965 Poznań, Poland
Email: mariusz.glabowski@put.poznan.pl

Sławomir Hanczewski
Chair of Communication and Computer Networks
Poznan University of Technology
ul. Polanka 3, 60-965 Poznań, Poland
Email: slawomir@hanczewski.pl

***Abstract*—This paper presents a platform for testing and the development of new routing protocols. The platform is an alternative to already existing solutions of the type based on physical devices or on virtualization. In the proposed solution, the testbed nodes are simple routers, i.e., the devices of System on Chip type, with embedded Linux system. These routers perform only packet switching functions, i.e., the function of the Data Plane. The functions related to supporting routing protocols, i.e., the functions of the Control Plane, for all nodes have been moved to a dedicated computer. The Control Plane functions are provided by the Quagga software router, modified for the purposes of the platform. With low cost and small size of single nodes, the platform can also be used in teaching.**

***Keywords*-Routing protocols; Software Router.**

I. Introduction

The computer networks classes, conducted in Chair of Communication and Computer Network at Poznan University of Technology, among others, are usually carried out using proprietary devices such as Cisco, Juniper or Allied Telesis. The advantage of lab classes with the use of professional routers and switches is to enable students to get familiarized with a configuration of devices that they might meet in practice – in corporate and providers' networks. The disadvantage of this solution that was reported by students, is the operating systems proprietary code for these devices, and hence no chance of modifications and testing of networks protocols, i.a. routing protocols.

The following two solutions that allow for testing of routing protocols have been used in the hitherto known platforms:

- Hardware, i.e., the one where each testing network's node is an independent, totally functional router (either a router or a PC computer with an appropriate software),
- Virtualization, which enables to build a test network of a particular typology on a single physical server.

The solutions based on physical nodes are characterized by high stability — each node is independent and its load does not decrease performance of other nodes. Such a network is, however, difficult to maintain, especially as far as teaching laboratories are concerned where various subjects are being held/lectured (the need to manage connections of different network topologies). Depending on the type of equipment used, the costs of building such a network can be large and, as previously indicated, the possibility of modifying the protocols in case of proprietary solutions, can be significantly reduced or even impossible. Therefore, the testbed of this type is generally built on the basis of PCs running under Linux. For the implementation of new protocols, it is necessary to update the software on each node. An update procedure itself is sometimes very time-consuming.

In the other existing solution used for testing routing protocols, testbeds are using topology virtualization techniques, i.e., they are made of virtual machines (that are a network's nodes), embedded on a single physical server (working under the Linux system). The virtual testbed allows for setting up any connection topology for an indefinite period of time, a quick network's reconfiguration and easy changes in the number of nodes (their number depends on the server performance).

The main cost of building a testbed in the mentioned solution is a purchase of the server. Despite the undoubted advantages of this solution, the obtained test results, such as routing protocols output, might not be reliable, because an overload of a single node may have a negative impact on performance of other nodes, which results from task division of the server's processors.

In order to find the best way of how to eliminate the drawbacks of hardware and virtualization solutions, a new concept of the routers' architecture has been developed in the Chair of Communications and Computer Networks at Poznan University of Technology. It allows students to conduct advanced tests (along with possible modifications) for existing routing protocols, as well as to start and test newly developed routing protocols within class hours (including thesis). The proposed solution combines both the advantages of platforms that use physical devices only with those offered by virtualization. The nodes in a proposed testbed are built from very simple devices, responsible only for switching of the packets (Data Plane). This is the System of the Chip devices, running under Linux. As a result of

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-199-1

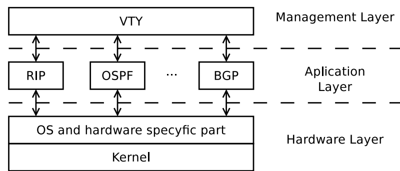

Figure 1. A general architecture of the software router

such an approach, each individual network node is physically independent, and the cost per unit does not exceed 20 euros. The functions responsible for routing (Control Plane) have been transferred in the suggested architecture to a dedicated computer that is running the Quagga software router. The handling of the Control Plane functions of all nodes is implemented by Quagga, after its appropriate modification. The independence of routing function for given nodes is obtained by activating a separate routing protocol process for each of them. The architecture obtained in this way is characterized by high simplicity and low building cost. With full access to the existing routing protocols and the ability to run new protocols, the proposed solution perfectly fits in the teaching of computer networks.

The further part of the article is structured in the following way. Section II presents the idea of software routing and gives an overview of popular solutions of this kind. Section III describes the concept of teaching networks, proposed in the article. Section IV includes a description of implementation procedures. Section V shows the usage scenarios of the elaborated platform. Section VI is a summary of the article.

II. Software router

At present, the most popular router projects with an open source are: Quagga [1] [2] and Xorp [3]. Figure 1 presents the general architecture of the software router. It consists of three layers: the hardware, application and the management layer.

The hardware layer uses the API operating system, or it refers directly to the hardware resources. This leads to its dependence on the operating system and on the hardware platform. The hardware layer is responsible for the preparation of the routing table based on the information received from the application layer. The selection of the route for a packet and its switching to the certain output port is possible owing to the entries in the routing table. The packet switching process takes place in the Data Plane (DP). Packet switching can be implemented via hardware or software. The hardware layer collects information on the status of the interfaces and on their Data Link layer addresses and Network layer addresses. The information on the status of the interfaces and their addresses is then passed on to the application layer, i.e., to the process supporting a given routing protocol.

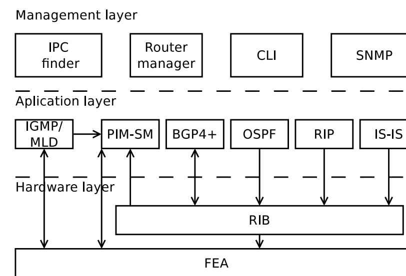

Figure 2. The architecture of XORP router

The application layer communicates with the hardware layer (kernel sublayer) via a special interface. This interface provides a hardware abstraction, which makes the application layer independent of the hardware platform and the operating system. Information on routing paths is sent to the hardware layer using the defined interface (between the application layer and the hardware layer). The applied solution allows simultaneous functioning of multiple processes in the application layer that are associated with various routing protocols. At the same time, the support of many routing processes does not reduce the system stability, since each process runs independently of the others. This solution also makes an easy addition of a new process with new routing protocol possible.

The management layer simplifies the configuration of the routing protocols. It provides access to a configuration of all protocols by CLI (Command Line Interpreter) or other protocols, e.g., WWW (World Wide Web), SNMP (Simple Network Management Protocol), TL1 (Transaction Language 1), etc.

The applied layered router architecture makes it easier to transfer the software of a router onto another operating system/platform, since the required modifications are mainly related to the hardware layer. This vastly simplifies the routing protocols migration between the devices and enables an easy and fast starting of software routers on many hardware platforms running on different operating systems.

The XORP project opts for the convenience of implementation, obtained thanks to C++ language. The whole code has been very well documented [4]. Figure 2 presents the architecture and the functional division of XORP router into particular modules. A possibility of multicast support has been additionally provided in the project. An implementation of the XORP project in the C++ programming language is less efficient; hence producers of network devices, such as software routers, choose the Quagga project.

The Quagga project is a fork of the Zebra software router project. A particular emphasis in Quagga project was put on the productivity and, therefore, the whole Quagga code was written in the C programming language. Rather than using the standard libraries, new ones have been written specially for the project's needs in order to achieve the

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-199-1

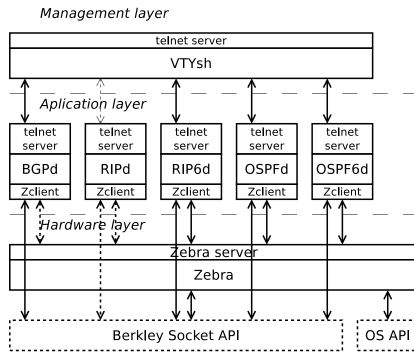

Figure 3. The architecture of Quagga router

Bits 0-15	Bits 16-23	Bits 24-31	Bits 32-47	Bits 48 - ...
Message size	Marker	Version	Command	Message data

Figure 4. The message format of the Zebra protocol

highest performance. Figure 3 presents the architecture of a Quagga router. The Zebra module is responsible for the API operating system support. It reads status of the interfaces and their addresses. It also modifies the system routing table. The Zebra module supports the following operating systems: Linux, Solaris, FreeBSD.

The modules of the application layer (ospf6d, ospfd, rip6d, pipd, bgpd) connect with the Zebra module via a local or TCP connection. The managing of modules in the application layer is possible by using CLI. Each module has a telnet server and supports multiple CLI sessions simultaneously. The VTYsh management layer module connects to all modules of the application layer. At the same time, it provides a telnet protocol server to which users can connect as well. Owing to the VTYsh module, the user has access to all modules of the application layer from a single console that supports the connection with VTYsh. It should be noted that the management layer in the Quagga project is not indispensable. Its addition is to make the command interpreter similar to the one used on Cisco routers.

The following section presents the idea of the modification of the Quagga project. It will consist of adding some functionality to the interface between the hardware layer and the application layer. The interface between the hardware layer and the application layer is described further in this section. The Zebra module operates as a server to which clients are connected – the application layer modules. The communication between the modules is provided by the Zebra protocol. This protocol does not have any documentation and was changing with the evolution of the project. Figure 4 shows the message format of the Zebra protocol. The first field **message size** specifies in bytes the whole message size (along with the header). It is a 16-bit field and the bytes are written in network order. Next 8-bit field **marker** is introduced to keep the compatibility with an older version

Table I
ZEBRA PROTOCOL MESSAGES

code	command	dir
1	ZEBRA_INTERFACE_ADD	C $\leftrightarrow$ S
2	ZEBRA_INTERFACE_DELETE	C $\leftrightarrow$ S
3	ZEBRA_INTERFACE_ADDRESS_ADD	C $\leftarrow$ S
4	ZEBRA_INTERFACE_ADDRESS_DELETE	C $\leftarrow$ S
5	ZEBRA_INTERFACE_UP	C $\leftrightarrow$ S
6	ZEBRA_INTERFACE_DOWN	C $\leftrightarrow$ S
7	ZEBRA_IPV4_ROUTE_ADD	C $\leftrightarrow$ S
8	ZEBRA_IPV4_ROUTE_DELETE	C $\leftrightarrow$ S
9	ZEBRA_IPV6_ROUTE_ADD	C $\leftrightarrow$ S
10	ZEBRA_IPV6_ROUTE_DELETE	C $\leftrightarrow$ S
11	ZEBRA_REDISTRIBUTE_ADD	C $\rightarrow$ S
12	ZEBRA_REDISTRIBUTE_DELETE	C $\rightarrow$ S
13	ZEBRA_REDISTRIBUTE_DEFAULT_ADD	C $\rightarrow$ S
14	ZEBRA_REDISTRIBUTE_DEFAULT_DELETE	C $\rightarrow$ S
15	ZEBRA_IPV4_NEXTHOP_LOOKUP	C $\rightarrow$ S
16	ZEBRA_IPV6_NEXTHOP_LOOKUP	C $\rightarrow$ S
17	ZEBRA_IPV4_IMPORT_LOOKUP	C $\rightarrow$ S
18	ZEBRA_IPV6_IMPORT_LOOKUP	N/A
19	ZEBRA_INTERFACE_RENAME	N/A
20	ZEBRA_ROUTER_ID_ADD	C $\rightarrow$ S
21	ZEBRA_ROUTER_ID_DELETE	C $\rightarrow$ S
22	ZEBRA_ROUTER_ID_UPDATE	C $\leftarrow$ S

of the protocol. The value of this field should equal 255. In the older version of the protocol this field was interpreted as a command. The following 8-bit field **version** specifies the version of the Zebra protocol. The present version of the protocol is 1. The last field of the header field is a 16-bit field **command** that specifies the command. Command code is written in network order. Content of the field **message data** depends on the command. In the Zebra protocol there are 22 messages provided, listed in Table I. The first column specifies the value of the message code that is placed in the field **command**. The second column contains the name of the message, and the third one the direction in which it is sent. The letter C stands for a process running in the application layer, and the letter S means a module running in the layer within the Zebra equipment. These messages can be sent to the Zebra module (direction C $\rightarrow$ S), to a module with routing process (direction C $\leftarrow$ S), or in both directions (C $\leftrightarrow$ S). Messages sent in both directions often have asymmetric forms. Therefore, it is necessary to use the right tool for an analysis of sent messages. In this case, it may be the Wireshark program. Unfortunately, the implemented module in the above mentioned program is for the analysis of the older, now outdated, version of the protocol. Therefore, a Wireshark modification, that includes the new version of the Zebra protocol, needs to be prepared for the construction of the test platform.

The process supporting the routing connects to the Zebra module. Zebra server, running in the Zebra module, may support many connections simultaneously. It should be noted that not all messages included in the Zebra protocol have been implemented. Those not implemented are marked in the column specifying the direction as N/A.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-199-1

The module supporting the routing protocol is running on an abstract hardware and, in this way, it is partially independent from the operating system. The module is not fully independent because in order to send or receive a signal, such as Ospf Hello message for OSPFv3 protocol, direct use of the socket API is required. By sending a signal message via the API operating system, the routing protocol specifies an interface through which the message is to be sent. Similarly, it also uses the API operating system to read messages. The system returns the received message, the source and destination address, as well as the information on the interface that received the message. This requires an application of a particular function from the API operating system. Some differences in functioning of the socket API may occur, depending on the operating system. This forces the adjustment of a program, running at the application layer, to a specific operating system.

The architecture of the Quagga project, optimized with regards to its performance, has some disadvantages: no process can be moved on another machine and the implementation of routing protocols is dependent on the API operating system. In order to make the mentioned platform applicable for the analysis and testing of existing routing protocols and to design new routing protocols (also for teaching purposes), a modification that enables a physical division of the Data Plane (DP) and the Control Plane (CP) functions is proposed in the next section.

III. A CONCEPT OF A PLATFORM FOR TESTING AND DEVELOPING ROUTING ALGORITHMS

The main idea of the proposed platform is to move the application layer to a dedicated computer. In the proposed solution, the application layer meets the CP functionality, and the hardware layer implements the functionality of the DP. A similar approach was applied to the GMPLS system [5], introducing a distinction between CP and DP. In comparison to the GMLS system, the difference is that CP in the proposed solution does not have its own signaling network. The routing protocol messages (supported by CP) are sent via network, supported by DP. Owing to this division, the application layer does not use the hardware resources directly. In this way, one machine (virtual or physical) can support many application layers concurrently, each for a separate node.

Figure 5 presents a network consisting of N nodes. Their CP is moved to a separate machine, shared by all nodes. Each router shown in Figure 5 has a dedicated router implementing the DP functions, while the machine supporting CP has many CP instatnces running. Each such instance is shown as a rectangle drawn with a dotted line. The proposed network may have one central computer that is running numerous CP instances for all nodes (as shown in Figure 5), or CP can be distributed across different machines.

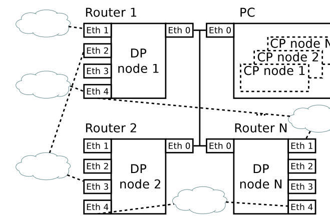

Figure 5. The proposed platform for testing and developing routing algorithms

In the extreme case, each node has a separate computer implementing the CP functions[1].

In order to connect CP with DP, a separate network was dedicated for this purpose. In Figure 5 all devices implementing DP functions are connected to this network via eth0 interface. This interface is unavailable for the DP network and invisible for the routing protocols. The routing protocols specify the path for the DP networks that consist of the nodes, using the eth1–eth4 interfaces. The computer with CP instances is connected to the network that supports an interface between CP and DP (in Figure 5 via eth0 interface).

Each CP instance must be properly configured. The configuration specifies the eth0 interface address of the DP device that is supported by a corresponding instance of CP. A single CP instance is composed of multiple processes, each supporting a different routing protocol. The routing protocol can be configured using the CLI. Access to the console is controlled using the telnet protocol. Any process that supports routing process listens for TCP connections on the specified port. This port must be configured so as to be unique within a given machine. This requires additional settings.

The Quagga software router processes running save their IDs in file /var/run/quagga. The file name depends on the routing protocol, supported by a given process. It allows for a simple stops/starts of the processes. Supporting of multiple processes for the same routing protocol requires a unique name of the file stored in the folder /var/run/quagga for each of the processes. A unique file name should depend on the routing protocol and the node on which the protocol operates. The last parameter that must be set is the name of the file where the configuration of routing protocol, designed for a given node, is stored.

In summary, the computer supporting multiple CP instances have multiple processes with the same name running, which are serving the same routing protocols. For each process, the following items have to be set:

- IP address of DP device,
- Unique port number at which a telnet server is listening,
- Unique filename of file, where process ID is stored,
- Unique filename of file that stores configuration of

[1] An idea of a central CP was proposed in an OpenFlow project [6]

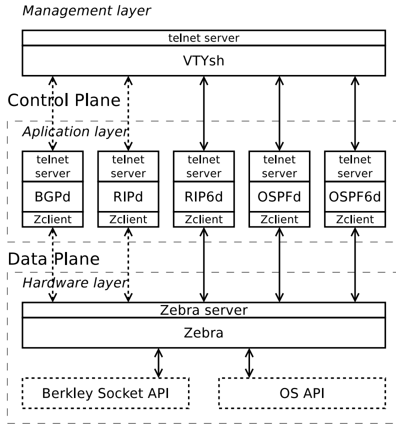

Figure 6. The node architecture in the proposed platform

routing protocol.

The presented approach has many advantages, despite the need to provide additional parameters for each of the processes. The entire network configuration is stored on one machine. All consoles to configure the routing protocols are available from the same machine, so there is no need for the management layer. There is also no need to output the console cables for all nodes. The modification of the process supporting the routing protocol is easier -– it only needs to be compiled once and then restart a modified process for all nodes. All of these steps can be implemented with access to a single computer.

The processes take the mentioned parameters from the arguments with which the process was started. This allows to write a simple script that starts, restarts, or turns off the processes for a single or all nodes. The following is a sample script that runs routing protocol OSPFv3 for node 1 with an IP address eth0 10.0.1.1.

```
./ospf6d --demonize \
 --dpaddr 10.0.1.1 --cliport 2701 \
 --pid /var/run/quagga/n1_ospf6d.pid \
 --conf /etc/quagga/n1_ospf6d.conf
```

To stop the OSPFv3 process for node 1, the following script needs to be started.

```
kill 'cat /var/run/n1_ospf6d.pid'
```

Figure 6 presents the node architecture (software router) in the proposed platform for testing and developing the routing protocols. The top rectangle drawn with a dotted line includes the processes responsible for the CP functions, and the bottom one includes the processes responsible for the DP. Each CP process communicates via the Zebra protocol with a Zebra module. The Zebra module supports API that controls the work of DP. The Zebra server is responsible for communication with a higher layer. In the proposed node, the modification the Zebra module additionally supports sending and receiving of signaling messages of routing

Table II
THE NEW MESSAGES OF ZEBRA PROTOCOL

code	command	dir
23	ZEBRA_CONFIGURE_RECEIVER	C –> S
24	ZEBRA_TRANSMIT	C <–> S

+	Bits 0-7	Bits 8-15	Bits 16-23	Bits 24-31
0	Message size		255	1
32	24		Interface Idx	
64	Interface Idx			
96	IPv4 / IPv6 packet with header			
...				

Figure 7. The format of ZEBRA_TRANSMIT message

protocol. The implementation of these steps required a Zebra protocol to be modified. For this purpose, two new messages were added. They are presented in Table II. Sending and receiving of a Zebra protocol message follows via ZEBRA_TRANSMIT command. Figure 7 presents a format of such message. A message length depends on the length of a packet (IPv4 / IPv6) with the signaling message that we want to send. The first parameter **interface idx** in the data field is a 32-bit value with an index of the interface through which a message is to be sent. The index of the interface is the same as the one in the message, adding a new interface. Interface index was saved as 32-bit value intentionally, since all the interface indexes in API of the Linux system [7] are written in the form of 32-bit numbers. The final element of the message is a IPv4 or IPv6 packet (including the header). In the header of the transmitted packet a destination IP address is stored. In the case of IPv6, the packet being sent should include the calculated checksum before sending it, although it is possible that DP counts the sum. Receiving a signaling messages from the network by CP requires DP mediation. DP needs to know the destination address of the packet that CP wants to receive. To do so, the right filter must be set, where unicast or multicast address is specified. The command ZEBRA_CONFIGURE_RECEIVER helps to configure DP. After an appropriate configuration, the DP transmits the received messages to CP via ZEBRA_TRANSMIT command. The format of this message is symmetric and remains the same regardless of the direction.

IV. IMPLEMENTATION

The implementation is in progress. The aim of the DP implementation is to build a firmware image which is then flashed into the router. Linux is frequently chosen as an embedded system. The firmware image of embedded Linux consists of a kernel and a file system. Open Embedded [8] is a toolset and sources for embedding Linux. The set offers many possibilities, ranging from a kernel with a basic toolkit up to a system with a graphic interface. With regards to the network devices, Open WRT [9] distributions have been developed. It includes a toolset for an oblique compilation,

the addresses of repositories with a system and programs kernel sources, and a set of patches that allow kernel or programs adjustment to the given hardware platform. The software set has been narrowed. An Open WRT is simpler in implementation, as compared to Open Embedded, and a software developer with a little experience can easily build a firmware image, using a creator. The configurator attached to the project can be run, using the `menu config` command. It enables a choice of hardware platform, a device and a set of programs. Moreover, it makes an addition of one's own programs possible. In order to build a firmware image for DP, the code with a Zebra module from Quagga project needs to be added and then modified. The modifications consist in adding the commands and functions to send and receive the messages or routing protocols described in the previous section.

The implementation of CP software consists in downloading of a Quagga project source code and its further modifications. The modification is based on adding the new commands to the Zebra protocol. It is necessary to modify the way of sending and receiving of messages for each routing algorithm. These operations are executed via the Zebra module. Thus, in order to send the packet via a given interface, its number and packet need to be placed in a message data field compatible with Zebra protocol. Similarly, packet reception is activated after Zebra module message is received. This message includes in its data field an interface number and the packet that has been received by this interface. The functions for sending and receiving packets are stored in file x_network.c, where x stands for a process name, e.g., ospf6d for the OSPFv3 protocol.

It needs to be noted that the system will run in the same way as the Quagga software router, if both softwares, DP (Zebra module) and CP (the modules with routing algorithms) are installed on the same machine. That means that the proposed code modification, after being implemented, may be added to the Quagga project. As a result, it will be possible to disperse one node onto more machines, to place all the processes supporting the routing protocol on one machine and to port very easily a project onto the new operating systems (all that needs to be done is to modify the Zebra module).

V. Usage scenarious

The proposed platform enables fulfilling the labs with routing protocols like RIP or OSPF. Regardless of the routing algorithm, the labs include the following tasks:

- Preparing physical connections between DP devices;
- Connecting the CP device to the network, and checking the communication between CP and DP devices;
- Launching the routing protocol, e.g. RIP;
- Watching the entries in forward (routing) table;
- Checking if the network is working correctly (ping command);
- The analysis of exchanged routing protocols' messages;
- The analysis of messages exchanged between CP and DP;
- Checking if the network is able to establish a new path afer physical breaking the link (disconnecting the cable);
- Making the changes in routing protocols configuration, e.g. changing the timers in the RIP protocol.

VI. Conclusion

This article presents a new concept of a network platform that enables both an analysis of existing network protocols and the implementation, plus testing, of new protocols. The proposed platform, based on the Quagga software router concept, combines the advantages of solutions relying on the network node vitualization with the testing networks implemented with the help of hardware routers. The platform enables a simple addition and modification of routing protocols in a testing network without a need for cross-compilation and uploading the firmware to each of the routers. Simultaneously, the complexity of a routing protocol does not affect the functioning of a hardware node. This allows to implement the platform using much simpler and cheaper hardware nodes. In further works, related to the proposed platform for testing the routing protocols, an analysis module for the Wireshark program will be developed. This should allow to pick up an information exchange with hardware nodes on the CP interface.

References

[1] "Quagga homepage." [Online]. Available: www.quagga.net/ <retrieved: May, 2012>

[2] A. Bianco, R. Birke, J. Finochietto, L. Giraudo, F. Marenco, M. Mellia, A. Khan, D. Manjunath, "Control and management plane in a multi-stage software router architecture," in *High Performance Switching and Routing*, May 2008, pp. 235–240.

[3] "Xorp homepage." [Online]. Available: www.xorp.org/ <retrieved: May, 2012>

[4] "Xorp architecture." [Online]. Available: http://xorp.run.montefiore.ulg.ac.be/latex2wiki/design_overview <retrieved: May, 2012>

[5] E. Mannie, "Generalized Multi-Protocol Label Switching (GMPLS) Architecture," RFC 3945 (Proposed Standard), Internet Engineering Task Force, Oct. 2004.

[6] "Openflow switch specyfication," Feb. 2011. [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf <retrieved: May, 2012>

[7] *The Linux Kernel*. [Online]. Available: http://kernelbook.sourceforge.net/ <retrieved: May, 2012>

[8] "OpenEmbedded." [Online]. Available: http://www.openembedded.org <retrieved: May, 2012>

[9] "OpenWRT." [Online]. Available: https://openwrt.org/ <retrieved: May, 2012>

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-199-1