Implementation of ARP-Path Low Latency Bridges in Linux and OpenFlow/NetFPGA

Guillermo Ibáñez[1], Bart De Schuymer, Jad Naous[2], Diego Rivera[1], Elisa Rojas[1], Juan A. Carral[1]

1 University of Alcala,, Madrid Spain 2 Stanford University, Stanford, CA USA

***Abstract*—This paper describes the implementation of ARP-Path (a.k.a. FastPath) bridges, a recently proposed concept for low latency bridges, in Linux/Soekris and OpenFlow/NetFPGA platforms. These ARP-based Ethernet Switches rely on the race between the replicas of a standard ARP Request packet flooded over all links, to discover the minimum latency path to the destination host, complemented in the opposite direction by the ARP Reply packet directed to the source host. Implementations show that the protocol is loop free, does not block links, is fully transparent to hosts and neither needs a spanning tree protocol to prevent loops nor a link state protocol to obtain low latency paths. Implementations in Linux and OpenFlow on NetFPGA show inherent robustness and fast reconfiguration. Previous simulations showed a superior performance (throughput and delay) than the Spanning Tree Protocol and similar to shortest path routing, with lower complexity.**

***Index Terms*—Ethernet, Routing bridges, Shortest Path Bridges, Spanning Tree**

I. Introduction

Ethernet switched networks offer important advantages in terms of price/performance ratio, compatibility and simple configuration without the need of IP addresses administration. But the spanning tree protocol (STP) [1] limits the performance and size of Ethernet networks. Current standards proposals, such as Shortest Path Bridges (SPB) [2] and Routing Bridges [3] rely on a link-state routing protocol, which operates at layer two, to obtain shortest path routes and build trees rooted at bridges. They have significant complexity both in terms of computation and control message exchange and need additional loop control mechanisms. A simple, zero configuration protocol may be effective in data center and campus networks to replace spanning tree when the complexities of the above mentioned protocols are not justified and high performance at low cost is the priority.

In this paper, we describe the implementation of ARP-Path Ethernet Switches in Linux/Soekris and Openflow/NetFPGA platforms, recently proposed in [4] as Fast-Path. ARP-Path is a simple, zero-configuration, minimum latency protocol suitable for metro, campus, enterprise, and data center networks that enables the use of all available links without routing computations or a spanning tree. Experimental results show robustness, fast reconfiguration and low latency.

II. ARP-Path Protocol

A. ARP-Path Set up

The ARP-Path protocol relies on the race between a flooded ARP request to establish the fastest path.

1) ARP-Path Discovery (ARP Request).Fig.1

When host S wants to send an IP packet over Ethernet to host D by IP address, it needs D's MAC address. If this mapping of IP address to MAC address is not in its ARP cache, S broadcasts an ARP Request for D's MAC address (shown as B in fig.1a). Ingress bridge 2 receives the frame from S and temporarily associates (*locks*) the global MAC address of S to the ingress port. Further broadcast frames from S arriving to other input ports of bridge 2 will be discarded as late frames from that source. S's address is now in a locked state and bridge 2 broadcasts B on all other ports (fig.1 a). Bridges 1 and 3 behave similarly, locking S's address to B's ingress port and broadcasting B over all other ports, thus sending duplicate copies to each other. Because these frames arrive at a different port from the one already locked to S, they are discarded (fig.1 b). In turn, bridges 4 and 5 process B the same way finally delivering B to the destination host D. There is now a chain of bridges, each with a port locked to S's MAC address forming a temporary path between S and D (fig.1 c).

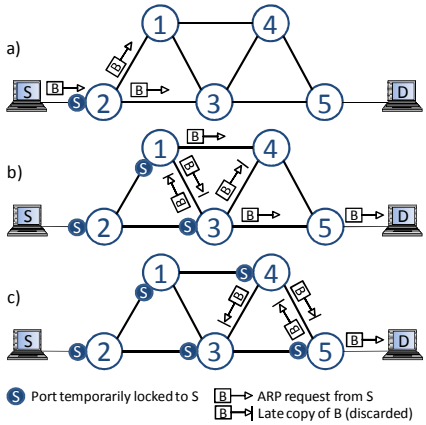

Figure 1. ARP-Path discovery from host S to host D. The small bubbles on the links show which switch port locked S's address.

2) ARP-Path Confirmation (ARP Reply)

The path is confirmed in the reverse direction (i.e. from D to S) when Host D sends the ARP Reply to host S in a unicast frame U that traverses the same path and confirms the learned

addresses. The confirmation mechanism ensures that the ARP-Path is symmetric. Path symmetry is required to prevent path oscillations. Specific priority mechanisms are used to ensure the uniqueness of the confirmed path in special situations like simultaneous ARPs exchanged by two hosts in opposite directions.

3) Path Repair

The method for path repair when an unicast frame arrives at a bridge where the destination MAC address has expired is described in detail in [4]. It uses the Path_Fail, Path_Request and Path_Confirm packets between the affected bridges to restore the complete path between source and destination bridges by emulating the ARP exchange. Simpler methods, relaying on direct broadcast of ARP Request from the "failed" switch in all directions, have also been tested and they also work in some topologies.

Notice that a MAC address may not be in the learning table for several reasons: no ARP was emitted to create the path, the entry was not refreshed and it expired, the link associated to the learnt port failed or even because the whole switch went down (and therefore, its table). In every of these cases, the repair method applied is the same.

B. Advantages

The protocol has several important advantages over other protocols that that explicitly build routes, namely: minimum latency (the selected path is the minimum latency path as found by the ARP Request message), zero configuration (there is no need to configure anything on hosts and bridges) and simplicity (functions performed at switches are MAC address learning and, optionally, ARP proxying to reduce broadcasts).

III. ARP-PATH IMPLEMENTATIONS

ARP-Path bridges have been successfully and successively implemented in several platforms. First, a Linux implementation with 100 Mbit/s links was performed to verify protocol correctness, robustness and compatibility in operation with existing networks. Afterwards, an implementation providing high performance and wide applicability in OpenFlow/NetFPGA was started.

The objective of the implementations was to verify the robustness of the ARP-Path transparent bridge concept at 100 Mbit/s and 1 Gbit/s wired networks, using all infrastructure links, without the spanning tree protocol or any ancillary routing protocol operating at layer two or three. Test networks were, respectively, a small 100 Mbit/s switched network of 3 ARP-Path Linux boxes (Soekris boards) and a high performance 1 Gbit/s network with four switches made with NetFPGA boards running OpenFlow, with an external controller.

Not only compatibility with existing campus networks have been shown, but also fast reconfiguration after link failure and absence of broadcast storms or other infinite loops. There were no broadcast loops, even when two ports of the same bridge were connected to each other. The ARP-Path bridges network mesh is directly connected to hosts on one side and to the campus network via one port on the other to avoid broadcast loops created by the standard bridges reinjecting frames to the ARP Path network.

Note that mesh topologies were selected in order to show how efficient this protocol is at avoiding loops, although the protocol could be used in bigger topologies. In fact, the protocol has previously been tested by using software simulations over Data Center topologies with two levels and up to 250 hosts connected. Also, these mesh topologies were the more complex ones with the hardware resources that were available (in this case 3 Soekris boards and 4 NetFPGA cards), to show the robustness of the protocol.

A. 100 Mbit/s. ARP Path Switches Linux Implementation

The first proof-of-concept of the ARP-Path bridge protocol has been implemented on a Linux 2.6 kernel and operates in kernel and user space by using *ebtables* [10]. We decided to use existing Linux kernel functionality so that no changes to the Linux kernel had to be made. This makes debugging the application easier and allows the application to run on any fairly recent Linux distribution. The disadvantage is, of course, loss of throughput due to extra context switches and packet buffering. The requirements on the functionality of the Linux platform are only minor:

- *brctl*: tool to setup the bridge

- *ifconfig*: for bringing up the bridge device

- *ebtables*: tool for Ethernet bridge filtering

- a recent Linux kernel with bridging and *ebtables* support (our code was tested on kernel version 2.6.31-14)

The implementation uses the Linux bridge functionality in hub mode, which is accomplished by setting the *ageing time* and *forwarding delay* of the bridge to zero. The decision whether to forward a packet from one bridge port to another bridge port is done by using *ebtables* filtering rules. The *ebtables* program is used to configure the Ethernet bridge packet filter that runs in the kernel. Setting up the *ebtables* filtering rules is done by a user space program that inspects all relevant traffic. To copy the packets from the kernel to the user space program, we use *ebtables*' ulog target, which uses the *netlink* infrastructure as communication channel. Excessive traffic will easily overload *netlink* communication, leading to the message "recvmsg: No buffer space available". Tuning the value in */proc/sys/net/core/rmem_max* did not help. To reduce the impact of this bottleneck as much as possible, the *ebtables* rules were adapted in such a way that the user space program only receives messages that cannot yet be processed by the active *ebtables* rules. The following optimisations were done to reduce the number of packets sent to user space in order to prevent overloading the *netlink* buffer:

- Traffic of a confirmed path is handled completely by *ebtables* rules, without further inspection by the user space program. To be able to do this, the *ebtables* FORWARD chain was used for all rules. This is needed because a confirmed

path is partly identified by an input and output bridge port. The packet counters of the *ebtables* rules are periodically parsed in order to be able to update the time out values of active confirmed paths and remove paths that have timed out.

- In the ARP-Path Repair situation, where the application itself has sent an ARP request and is waiting for the ARP reply, subsequent packets that would trigger the same ARP request are rate limited. This is done using the *ebtables* limit watcher. This has as side effect that some packets may be dropped if the traffic is dense. In practise, we experienced this scenario with unicast video traffic for a confirmed path that had timed out, while the streaming server's ARP cache entry hadn't expired yet. For most scenarios, multiple *ebtables* rules have to be added at once. *Ebtables* provides two mechanisms to enable atomic insertion of multiple rules:

- Create a new chain and add all the necessary rules one-by-one to the chain. Finally, add a rule to an appropriate other chain which jumps to the previously created chain.

- Add the rules one-by-one to a user space version of the *ebtables* filter table, using the *atomic file* functionality. Then use the *atomic commit* functionality to atomically commit the complete table to the kernel. A disadvantage of this mechanism is that the packet counters of existing rules are reset after the atomic commit.

Link failure detection is implemented in a separate thread that listens on a NETLINK_ROUTE socket of the NETLINK address family. A change in an interface's link state is evented using *netlink* messages of type RTM_NEWLINK and RTM_DELLINK. Both bringing a bridge port down using *ifconfig* and unplugging a network cable are evented by this mechanism. When link failure is detected for a bridge port, all associations of remote MAC addresses to the specific bridge port are removed and the *ebtables* filter rules are updated. Functional testing during development of the ARP Path protocol was done using virtualization. This enables creating virtual networks of various setups without requiring any additional hardware infrastructure. We chose to use User-Mode Linux, which allows running a complete custom-built Linux kernel as a program on the host. Using some basic scripts to automatically create a virtual test network, provides a fast and easy way to recreate a network test environment. We did notice, however, that the UML clients had a tendency to crash when put under relatively heavy network load. Performance testing was done on real networks.

Implementing the protocol in user space allowed for fast prototyping, especially since much existing kernel functionality could be reused. However, using *netlink* as a communication channel for passing network packets between kernel and user space proved to be a bottleneck for performance in a real-world scenario. However, we are confident that when the application would be rewritten as a Linux kernel module, the obtained throughput performance in normal scenarios would be comparable to the standard Linux bridge implementation.

The test network scenario is shown in figs. 3 and 4. Three Soekris Linux boxes operate as three ARP-Path switches fully connected in a triangle. Two of these switches are connected to standard hosts (*H, S*), and the third switch is connected to the Internet through a wired Fast Ethernet connection.

The path establishment between H and S is continuously monitored with repetitive pings. The first ping takes more time because path has to be set up at switches and Linux bridges execute it partially in user space. The partial user space implementation has been chosen because it was simpler to code in order to prove the concept, despite the possible impact on performance. Disconnecting a link from the path in use activates path reconfiguration. The new selected path is, obviously, the alternative and longer two-hop path via the third bridge, with a bit longer delay as a result of the additional hop. After path repair and link reconnection, the original path is only reselected if there is a fail in the selected path. To show video connectivity performance, the two hosts are configured respectively as video server and client by using Videolan (VLC), where S is transmitting via HTTP to H. When a cable is unplugged, the path reconfiguration is visually verified through video reception which is very shortly interrupted and, later, recovered.

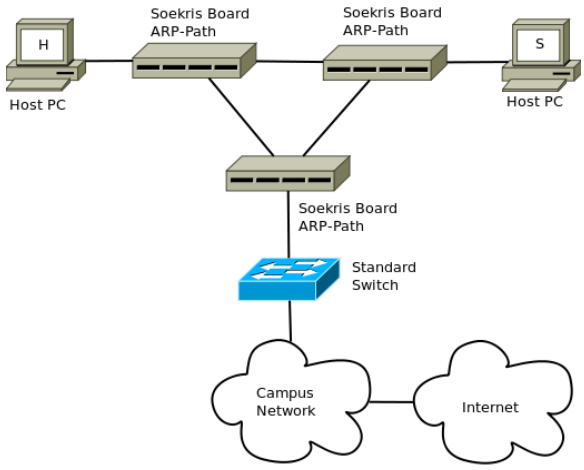

Figure 3. ARP Path Switches (Linux impl.) 100 Mbps test network

Figure 4. ARP-Path Switches network with Soekris (Linux) boards

Most of the reconfiguration time is consumed by the link failure detection and by the processing of ARP Request to establish the new path, performed in user space, instead of in the Linux kernel. Reconfiguration time is around 1070 ms. with Soekris boards at 500 Mhz speed, half time with more powerful processors. When the path is already established, ping round trip time is 900 microseconds.

B. *1 Gbit/s OpenFlow/NetFPGA implementation*

After implementing ARP-Path in Linux, a higher capacity implementation at 1 Gbit/s link speed was the objective. A pure NetFPGA implementation [8] would be advisable since,

opposite to many other implementations using OpenFlow, there is no intrinsic central function in the ARP-Path protocol: all switches are fully independent. But, taking into account the ease of protocol modification and the diversity of available platforms, an implementation based in OpenFlow [9] controlling NetFPGA switches was chosen. OpenFlow makes possible the validation across many platforms, ranging from fully virtualized networks to commercial switches OpenFlow compliant. Since OpenFlow specification does not specify a controller, we chose NOX [10]. It is important to remark that, in an OpenFlow setting, the implementation of the protocol resides in the controller, i.e., NOX. The OpenFlow standard defines the communication language between the switches and the controller, and the NetFPGA cards operate as OpenFlow switches, but they are completely independent from the protocol implementation. Fig. 5 shows a generical OpenFlow network.

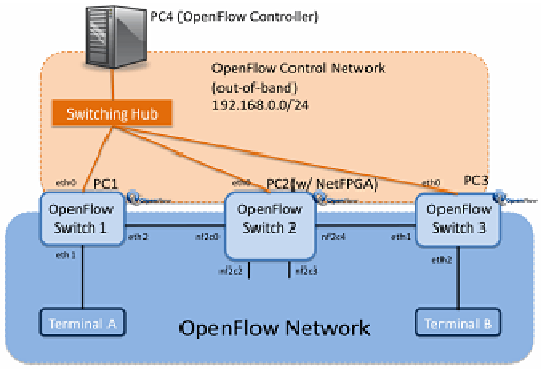

Figure 5. Generic OpenFlow network: controller and Openflow switches [9]

The controller part is identical in both the virtualized and the real implementation (with NetFPGA switches) and it is what really defines the behaviour of the ARP-Path protocol. NOX lets the user develop items in C++ and Python; ARP-Path protocol has been written in Python. The ARP-Path module consists of several files, but just one defines the protocol logic, named "`modswitch.py`". This file describes, as the rest of NOX modules, two important functions: "`__init__`", which initializes the component, variables and logging (for later result analysis), and "`install`", which mainly registers switch events in the controller (such as receiving a frame, listening to port status changes, etc). The most important event registered is for packets coming in and it calls the function "`handle_packet_in`", which owns almost the whole logic of the ARP-Path protocol. This function first classifies broadcast, multicast and unicast frames in three different sections. The multicast section is just used for ARP-Path multicast packets (specifically the ones needed for repairing paths), while the broadcast and unicast sections will look for ARP frames (in order to lock and confirm paths), which will create the associated learning tables, and other frames, which will be forwarded or discarded (if a broadcast needs to be blocked to avoid loops), or will even start the repairing method if the destination MAC address is not located in any table.

After developing the ARP-Path code at the controller, the tests start. This module should be running in the controller that will tell the network switches to behave like ARP-Path switches. The first phase of the implementation does not require any NetFPGA hardware. The complete test environment can be virtualized by running OpenFlow and NOX on a virtual machine. One method would be using a virtual machine for each PC (one for the controller and some more for each switch and host in the topology), nevertheless this is quite a tedious work. The alternative that was used in this virtual implementation was Mininet, an idea from the McKeown Group Wiki [11]. Mininet can create large OpenFlow networks with virtual hosts, switches on a single PC, and is available at the wiki of the OpenFlow Web page [9]. Although still an alpha release, it already is a powerful tool to easily test big networks. Only two virtual machines are needed with Mininet, one for the NOX controller in which the ARP-Path protocol logic is executed, and another one for the Mininet software, which simulates the different topologies to be tested. Once this ARP-Path module is running (by starting the NOX controller with a sentence like this "`./nox_core -i ptcp: modswitch`"), it is possible to connect to it the different switches in the topology (e.g. by running "`sudo mn --controller=remote --ip=CONTROLLER_IP --port=CONTROLLER_PORT --custom TOPOLOGY_PATH --topo mytopo`" in Mininet or alternatively starting the OpenFlow protocol in each of the switches of a real implementation).

In Fig. 6 the OpenFlow dialogue is shown. On the left, it is the Mininet virtual machine, whose network topology is defined by a simple Python file that describes hosts, switches and their links. On the right, the controller NOX owns the whole decision making process regarding packet processing rules, it is where the ARP-Path protocol logic is defined. An OpenFlow dialogue will be created between both machines. Initially, when Mininet is started and switches power up. Later, the controller will listen to some previously registered events, such as "*a new host has joined the network*" or "*switch 2 has received the following frame*", and will give different instructions to some parts or the whole network, such as "that frame should be forwarded via eth0 from now on"

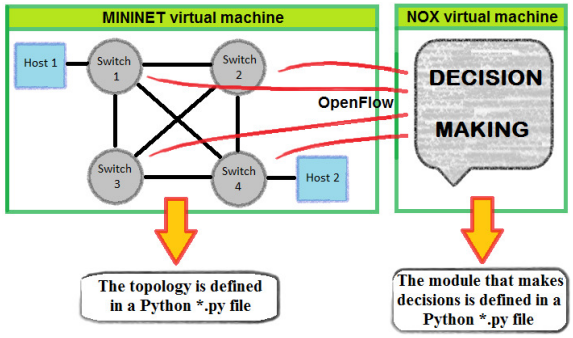

Figure 6: OpenFlow communication among NOX and the Mininet topology

When the protocol has been tested, the real deployment is designed. The 1 Gbit/s test network is shown at figs. 7 and 8.

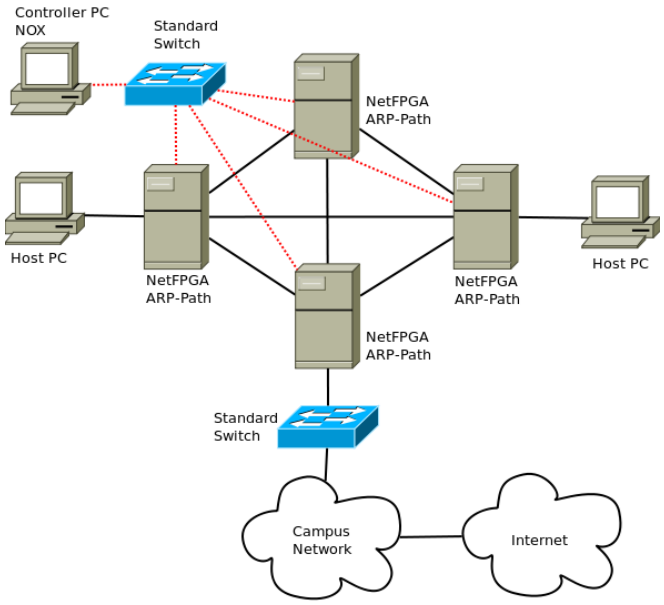

Figure 7. Test network for OpenFlow/NetFPGA 1 Gbps

The network consists of a fully connected mesh of 4 NetFPGAs (one per PC) acting as modified OpenFlow switches, with one central OpenFlow controller that implements the ARP-Path logic for every switch. Connectivity test and path reconfiguration after link failure were performed as for Linux demo with repetitive pings. Hosts were configured as video servers and also connected as clients with Videolan and to video webs via http. All network links were connected and active at the same time and shortest (direct in this case) paths were selected. Paths were monitored by displaying the log of the NOX controller. The measured path set up time with repetitive pings between source and destination was 40-50 ms (if the MAC address is not already in the table at the ARP-Path switches). Path reconfiguration after unplugging a link cable takes 40-50 ms after link failure detection (opposite to Linux implementation, link failure detection is not yet available in the NetFPGA version used, so a timeout was used to detect link failure). Once the path is established, it takes only 0.5 msec for a ping round-trip time. ARP stress tests show handling of ARP Request rates of up to 1000 ARPs per second, by using the arp-sk tool.

Figure 8. OpenFlow/NetFPGAs network set up

C. *100 Mbit/s OpenFlow implementation on Soekris boards.*

Another tested scenario was a mixture of the two described above. We installed OpenFlow in three Soekris boards, and then used the same code we had implemented for the OpenFlow/NetFPGA setting in them as shown in fig. 9. This is possible thanks to the OpenFlow architecture, which allows testing a protocol over any compatible device. Soekris boards are internally specialized PC boards with Linux installed, so it is possible to install OpenFlow on them and then use another PC, connected via Ethernet, as the controller. This set-up gave us similar advantages of the OpenFlow/NetFPGA implementation (such as the easy reconfiguration of the protocol, which just means changing the controller code instead of installing every single board), but in a more portable and easier to configure scenario. Once Openflow was installed in each Soekris board, we used again the NOX controller installed in a laptop. The connections between the Soekris boards remain as they were described in Linux implementation section, creating a triangle topology, with hosts attached to the switches. With this set-up we were able to test ping connection, and also HTTP video streaming between hosts, and Internet connection through an Ethernet link connected to another Soekris board.

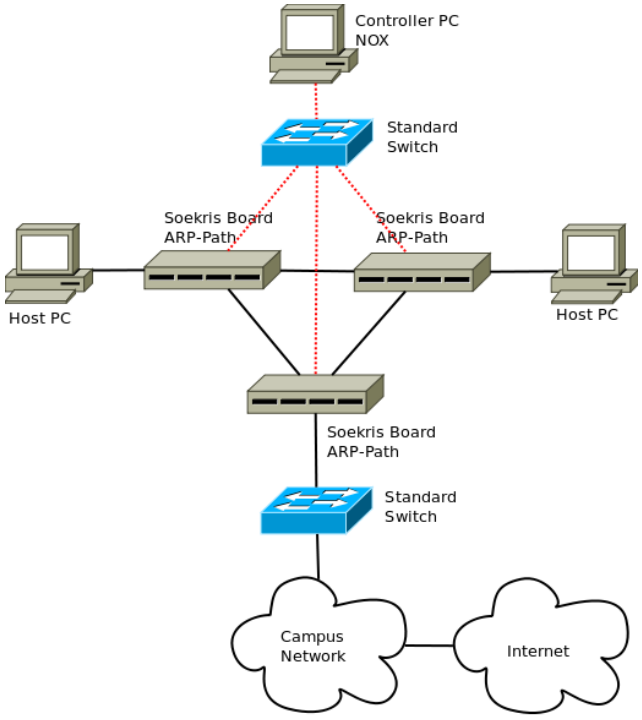

Figure 9. Test network for OpenFlow/Soekris 100 Mbps

D. *Comparison*

Although every proposed test worked well in each scenario (there were no problems with ICMP packets, video streaming or Internet connection, including auto configuration via DHCP), there were some differences in performance measures for each one. These differences are visible in ping RTT (Round Trip Time) results. In the Linux implementation, there was an average RTT of 798 microseconds (without taking the starting time into account, when the path is being created). In the OpenFlow/NetFPGA implementation, this time decreased

to 627 microseconds. The difference between both values is understandable knowing that the first one is running part of the protocol in user space, while the NetFPGAs implement OpenFlow as a hardware project. In OpenFlow/Soekris, OpenFlow is installed as an application in user space, instead of using a hardware programmed device (like NetFPGAs), and this clearly has a negative influence in performance. The average RTT is 2,82 miliseconds, a higher value than the other two, as expected. The more realistic value of these options is the 627 microseconds obtained from the Openflow/NetFPGA implementation. Comparing this value with the average RTT by using Spanning Tree Protocol with OpenFlow and NetFPGAs, we get similar values (612 microseconds for STP), which is reasonable considering the size of the topology (the implementations were prepared to test the protocol itself and were not so focused on the performance).

IV. Acknowledgment

This work was supported in part by grants from Comunidad de Madrid and Comunidad de Castilla la Mancha through Projects MEDIANET-CM (S-2009/TIC-1468), EMARECE (PII1I09-0204-4319) and T2C2(TIN2008-06739-C04-04).

V. Conclusion

ARP Path is a simple and efficient protocol low latency transparent bridges, currently under active development and optimization, suitable for enterprise networks and data center networks. The success and robustness proved with the different implementations are a strong proof of concept, which show the protocol behavior under real applications like video streaming or Internet access. Each implementation has some advantages and disadvantages, and can be used for different purposes. For example, the Linux implementation is the simplest one, in terms of configuration, but it is not the most realistic in measurement. Also, every change made in the code, needs to be recompiled in each device individually. The OpenFlow implementations do not have that problem, as everything is managed by a centralized controller. In fact, using NetFPGAs with OpenFlow gives us a very realistic set-up. The main problem with the OpenFlow/NetFPGA approach is that is not an easily transportable model, so is not very suitable for demonstration purposes. That is where the OpenFlow/Soekris configuration is better, because that scenario is easy to move, and it gives a good demonstration model. The problem with it, comes with the worse performance derived from the lack of OpenFlow hardware integration in the Soekris boards. A pure NetFPGA implementation is ongoing.

VI. References

[1] IEEE 802.1D-2004 IEEE standard for local and metropolitan area networks-Media access control (MAC) Bridges. Available online: standards. ieee.org/getieee802 /802.1.html.

[2] M. Seaman. Shortest Path Bridging. Available online: ieee802.org/ 1/files/public/docs2005/new-seaman-shortestpath-par-0405-02.htm.

[3] Transparent interconnection of lots of links (TRILL) WG. Available on line at: ietf.org/html. charters/trill-charter.html

[4] Ibanez G. et al. ARP-Path Ethernet Switching: On-demand Efficient Transparent Bridges for Data Center and Campus Networks. LANMAN May 2010. Available on line at: hdl.handle.net/10017/6298

[5] Elmeleegy, Khaled and Cox, Alan L. EtherProxy: Scaling The Ethernet By Suppressing Broadcast Traffic. Proceedings of IEEE INFOCOM 2009, Rio de Janeiro, Brazil.

[6] Omnet Simulatior. Available on line: omnetpp.org

[7] NRS Reference Networks. Available on line at: ibcn.intec.ugent.be/ INTERNAL/NRS/index.html

[8] NetFPGA: netfpga.org

[9] Openflow: openflowswitch.org

[10] Ebtables: http://ebtables.sourceforge.net

[11] McKeown Group Wiki: http://www.openflowswitch.org/foswiki/bin/view/Main/WebHome