VLANs and GVRP on Linux: quickly from specification to prototype using the Click router platform

Pim Van Heuven (pim@think-wize.com), Federic Van Quickenborne, Filip De Greve, Brecht Vermeulen, Steven Van den Berghe, Filip De Turck, Piet Demeester
{frederic.vanquickenborne, filip.degreve, brecht.vermeulen, steven.vandenberge, filip.deturck, demeester}
@intec.UGent.be

1 Introduction

This paper presents how a working prototype of a VLAN (Virtual LAN) and GVRP (Generic VLAN Registration Protocol) aware Ethernet switch for Linux can be built using a "Writing code is good but integrating existing code is much better" approach. The paper is split into two parts. The first part describes the development of the signalling component of the network: the GVRP daemon, the second part describes the Click router architecture and the modifications made to support VLANs.

2 VLANs and GVRP

Before we delve into the details of GVRP it is interesting to examine why VLAN and GVRP are used. This will be explained in the next two subsections. After these two subsections we will examine how GVRP works and illustrate this with an example.

2.1 Why VLANs?

Switched Ethernet networks are networks composed out of 802.1d-compliant Ethernet switches [1]. This requires that devices support Layer 2 Bridging (including the MAC address learning process) and the Spanning Tree protocol (STP). The Layer 2 forwarding process specifically requires that no traffic loops may exist in the network topology. The STP is designed to solve this problem by removing the redundant paths. The STP reduces the topology to a tree structure while still guaranteeing complete connectivity (this is the definition of a "spanning tree"). After network failure, the STP will reconfigure the spanning tree in order to recover the connectivity.

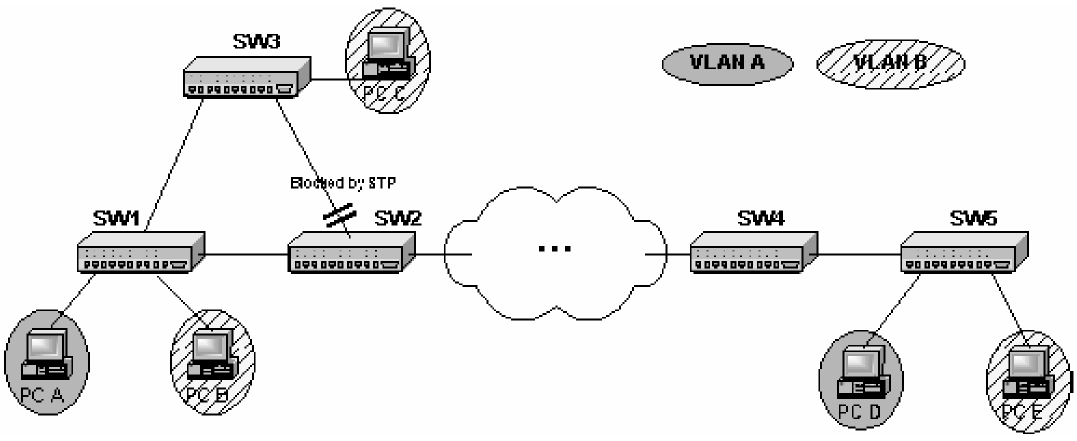

Figure 1: An example Ethernetwork.
The spanning tree protocol blocks the link SW2-SW3 to remove the cycle (SW1-SW2-SW3)

The spanning tree protocol is illustrated in Figure 1, the STP detects the cycle SW1-SW2-SW3 and will remove this redundancy by blocking, for instance, the link between SW2 and SW3.

The Ethernet network can be extended by introducing the concept of Virtual LANs (IEEE

802.1Q, [2]). VLANs provide the means to separate a single physical network into logical sub-networks. In order to support VLANs the the Layer 2 Bridging functionality needs to be extended to guarantee that Ethernet frames are restricted to their corresponding VLAN. To support this forwarding the Ethernet frames are marked with a VLAN tag.

The introduction of VLANs improves the overall network performance because broadcasting is restricted to a certain VLAN and not the whole network. VLANs also offer label-based Layer 2 security and are an easy set-up method for multicasting.

In the example network of Figure 1 PC A en PC D belong to VLAN PC A while PC B, PC C and PC E belong to VLAN B. To support correct VLAN forwarding it is not sufficient to configure solely the end stations but the switches have to be configured too. These entries in the VLAN Filtering Database can be created manually (e.g. via SNMP) but this has to be done for every port of every Ethernet switch. The required effort to configure the network will increase with the amount of switches residing between SW3 and SW4.

2.2 Why GVRP?

GVRP is an IEEE standard that performs automatic set-up of VLANs in Switched Ethernet networks. GVRP removes the burden of manually installing and maintaining VLANs from the network administrator’s hands. This automatic VLAN registration is performed in a more consistent and reliable way compared to the laborious manual VLAN configuration on every switch in the network. Instead configuring on every port of every switch the VLANs are declared on a limited amount of edge devices while the intermediate devices can learn these VLANs through signaling.

The automatic set-up of VLANs not only reduces the chance of incorrect VLAN configurations but also makes the VLANs resilient to Layer 2 network failures because it works in conjunction with the spanning tree protocol. After spanning tree protocol has converged the VLANs are automatically remapped to the new active topology induced by the new spanning tree. A manually configured VLAN remains is ignorant to the SPT network reconfiguration and may use failed links or blocked ports.

2.3 How does GVRP work?

GVRP provides a mechanism for Ethernet switches to configure their VLAN port membership in a dynamic way. Each GVRP-aware switch is running an instance of the GVRP application with the corresponding state-machines on every active Ethernet port. These components can send and receive GVRP messages (or PDUs) on the LAN segment attached to the port. Internally these components communicate with each other by means of Service Requests. The Service Requests reside inside a propagation context, which enables the VLAN information registered on GVRP-aware devices to be propagated across the Ethernet network. This propagation context will always be defined by the current active Spanning Tree topology.

The operation of GVRP is based on two primitives: declaration (or withdrawal) and registration (or de-registration) of VLANs. The prime relation between these two primitives is that VLAN registration (or de-registration) can only occur on these ports that have received a GVRP PDU containing a VLAN declaration (or withdrawal). A GVRP PDU containing a VLAN declaration, is called a Join message. A GVRP PDU containing a VLAN withdrawal, is called a Leave message. The fact that GVRP participants have declared or registered a VLAN on a port, is recorded by means of state variables associated with the GVRP application on that port.

To understand the basic operation of GVRP, consider the following example (Figure 2). We reused the network of Figure 1 but all Ethernet switches are now GVRP-aware. The two VLANs need to be configured in order to separate the physical network in two logical sub-networks. The operation of GVRP will be explained by looking at the set-up process of VLAN B.

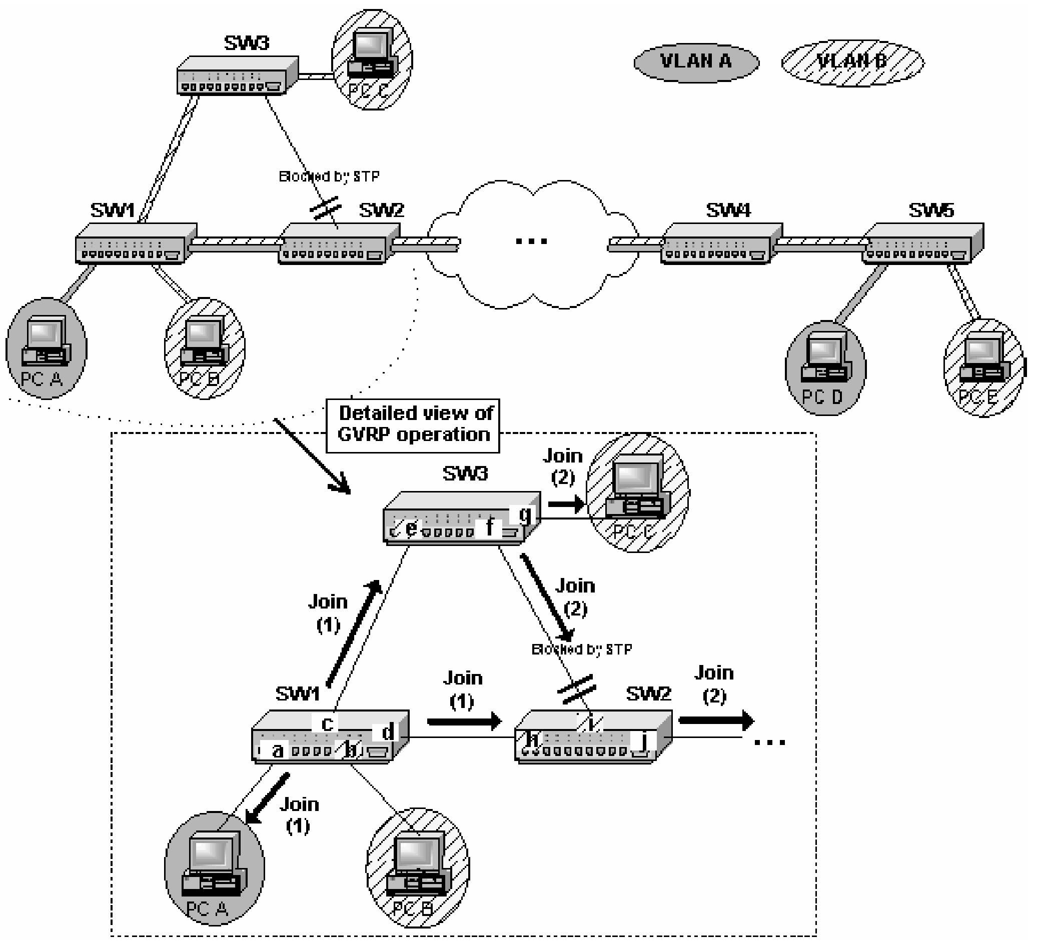

Figure 2: The working of GVRP illustrated.

PC B is member of VLAN B and would like to connect to other members of VLAN B. The network administrator registers VLAN B membership on port b of SW1. This registration is detected by GVRP and distributed by means of a Join Request towards all the other ports on the spanning tree of SW1 (a, c and d). These ports send a Join message (Join 1) on their attached LAN segments. The ports e and h on SW3 and SW2 receive this Join message, register the VLAN B and become member of the VLAN B. PC A is GVRP-unaware and will ignore the GVRP PDU. On switches SW2 and SW3 the VLAN registration is again distributed towards all the other active ports (Join 2).

However, the behavior of the blocked port i of SW2 differs from the others. VLAN B is registered after receiving a Join message from port f but no Join Requests are distributed because port f is blocked. Also incoming Join Requests on port i (from port f) are not propagated on the attached LAN segment.

This process is repeated for all the Ethernet switches of the network so that every switch knows how to reach the PC B on VLAN B.

In order to enable communication between PC B and PC C on VLAN B. PC C needs to join the VLAN too. PC C registers VLAN B membership on port g of SW3 and the entire process is repeated. The result is that ports g and c also register VLAN B.

PC A and PC C are now able to communicate on VLAN B but no Ethernet frames of VLAN B will reach the network parts attached to SW4 or SW5. For example PC E will only be able to communicate on VLAN B after explicit registration of VLAN B on SW5 and completion of the GVRP set-up process.

The advantage of GVRP is that the network administrator only has to configure the ports directly connected to the end users, GVRP will take care of the rest. Compare this to the effort needed to configure the necessary ports on all the intermediate switches of a large network.

3 VLAN and GVRP support for Linux

Now that we have given the benefits of VLANs and GVRP will investigate the current support for these two technologies in Linux.

3.1 802.1d and VLAN support

VLAN support also requires ordinary VLAN support i.e. 802.1d support which encompasses that the network supports bridging and the spanning tree protocol.

Ethernet bridging

The Linux kernel supports Ethernet bridging by enabling CONFIG_BRIDGE. When enabled the kernel will transparently forward packets from one Ethernet segment on one interface to all other segments on the other interfaces, combining the Ethernet segments into one large Ethernet.

A bridge can be set-up with the brctl tool [3], for example to make a bridge with eth0 and eth1, one has to:

```
brctl addbr br0
brctl addif br0 eth0
brctl addif br0 eth1
```

Spanning tree protocol

By enabling CONFIG_BRIDGE Linux automatically also supports the spanning tree protocol meaning that it can react to topology changes and network faults by recalculating a new spanning tree.

VLAN interfaces

Standard Linux (since around 2.4.14) allows you to create VLAN interfaces on your Ethernet interfaces by enabling CONFIG_VLAN_8021Q in your kernel. With the vconfig tool [4] one can add a VLAN to an interface.

```
vconfig add eth0 7
```

For example this creates a VLAN device eth0.7 with VLANID 7 on eth0. You can then add an IP address (for example 10.0.0.5) to this VLAN interface:

```
ifconfig eth0.7 10.0.0.5 netmask 255.255.255.0 up
```

VLAN bridging support

Linux supports VLAN bridging however there are a few drawbacks with the Linux bridging implementation which makes it not very suitable to be used in conjunction with GVRP. When one broadcasts over a bridge all the interfaces over the bridge receive the packet i.e. it is a real bridge and not a switch. Another problem is that there are potentially 4096 different VLANs, supporting them all requires 4096 bridges. We therefor decided to develop our own VLAN switching code based on the Click modular router. We will explain this in section 5 but first we look at GVRP.

3.2 GVRP support

To our current knowledge there is no open source implementation of a GVRP daemon for Linux available. In the next section we describe how such a GVRP daemon can be developed.

4 Implementing a GVRP daemon

When implementing a network protocol daemon the first place to look is always the relevant standards. Fortunately the IEEE 802.1d [1] standard contains an example Generic Attribute Registration Protocol (GARP) implementation. GARP provides a generic attribute dissemination capability that is used by participants in GARP applications to register and de-register attribute values with other GARP participants within a bridged LAN. GARP is a generic protocol, meaning of the definitions are dependent of the particular implementation of the GARP application like GVRP. Another example of a GARP protocol is the GARP Multicast Registration Protocol (GMRP). IEEE 802.1q standard [2] contains an example implementation of the GVRP protocol which combined with the GARP code in 802.1d is already a partial implementation of a GVRP daemon.

The code in the IEEE standards lacks all platform dependent code and code to support and administrative interface (f.i. a management CLI).

In rest of this section we look at how the IEEE reference implementation of GARP and GVRP are used to make a GVRP daemon for Linux and how the missing parts are filled in.

4.1 Extracting the code

Unfortunately there is no online repository where the source code contained in the IEEE specifications can be downloaded. So the first task in the development of the daemon is the extraction of the source code.

Extracting the source code is quite simple. First load the PDF specification in a PDF reader (`acroread`, `kghostview`, `gv` etc.). Then select the relevant pages of the document and print them to a postscript file. The corresponding postscript file can then be converted to plane text by using `ps2ascii`. Subsequently the headers and footers containing copyright information and page number can be removed or commented out by tools like `grep` or a savvy editor (`vim` or `emacs` are easily up to the task). At this moment you have one monolithic file which contains all the code of all the *.c and *.h files. Since every *.c or *.h file in the specification is presented in a different section it is possible to write a script that extracts and properly names every file from the monolithic file. Another possibility is to take the monolithic file and manually split it up in the separate files. This can be a good opportunity to read the source and indent and format it the way you want. Alternatively you can use `indent` tool to do that.

As already mentioned the IEEE example code lacks important functionality. The lacking functionality includes the encapsulation and decapsulation of the GVRP messages, timers, logging, memory allocation, an event queue and VLAN switching and tagging. We will now explain how the most interesting of these components were added.

4.2 Adding the timers

Linux normally only supports one timer per process. GVRP needs at least two timers per interface. One solution is to use threads to increase the number of "processes". Since there was no other reason to use threads we decided to implement the timers not directly through the OS primitives but by using a timer queue.

The timer queue stores *timer events* ordered by expiration time. A timer event is the combination of a callback function and due time. At the due time the callback function is called. The proper working of the timer requires that the timer scheduler is called regularly (synchronously). Every time the timer scheduler is called all the events that are due are executed. It is clear that the time between the individual calls of the scheduler determines the precision of the timer. GVRP specification is quite lax with respect of the timer precision so the requirements of the specification are easily met. We return to how exactly the timer scheduler is called when we discuss the main event loop.

4.3 Adding logging

There is no real formal requirement to add logging to the GVRP daemon. However debugging a distributed network protocol without some kind of visual feedback is quasi impossible. We choice to use different log levels in the code. A few of them are illustrated in Figure 3.

```
#define GVRPD_DEBUG      2
#define GVRPD_ERROR      3
#define TIMER_DEBUG      4
#define TIMER_ERROR      5
#define GVR_DEBUG        8
#define GVR_ERROR        9
#define GID_DEBUG       16
#define GID_ERROR       17
#define GIP_DEBUG       32
#define GIP_ERROR       33
#define GVF_DEBUG       64
#define GVF_ERROR       65
#define SYS_DEBUG      128
#define SYS_ERROR      129
               ...
```

Figure 3: Some of the debug channels used in the GVRP daemon.

As you can see the different log levels are split up in DEBUG and ERROR levels. The error levels are always logged while the debug levels are only logged if the GVRP is configured to do so (this is easy by doing a bitwise & 1 on the debug channel number). Each component of the GVRP daemon also has its own debug channel. This is very useful during development because you typically debug different components during different times. By setting a debug mask one can filter out the unnecessary debug channels (because the log channels number are all a power of 2).

4.4 Administrative console

In order to set-up a VLAN some kind of interaction between the end user and the GVRP daemon is necessary. We decided to develop a small stand-alone utility that sends requests to the GVRP daemon via a special CLI socket. Every GVRP daemon listens to a well defined port and when it receives a message it parses it and calls the correct function. By using standard sockets it is possible to send request from a single host to all daemons.

Note that this is actually a poor man’s version of SNMP and this approach is severely insecure.

4.5 The main event queue

The main event queue serves three different purposes: verify if messages have arrived on the CLI socket (section 4.4), schedule timer events (section 4.2) and see if GVRP packets have arrived. The select synchronous I/O multiplexer of the standard C library is a good candidate to implement the event loop. This is illustrated in Figure 4. Note that we will explain how the GVRP frames are received in section 6 “Putting it all together”.

```
/* The main scheduler starts here */
    while(1){
      FD_ZERO(&fds);
      FD_SET(cs, &fds);

      // Scheduler granularity 10ms (important for timer)
      intvp.tv_sec = 0;
      intvp.tv_usec = 10000;
      rc = select(FD_SETSIZE, &fds, NULL, NULL, &intvp);

      // We receive a msg from the CLI socket
      if (rc>0){
        if(FD_ISSET(cs, &fds))
          process_cli_msg(cs);
      }

      if (rc == 0){
        // Search for timer events
        timer_wakeup(0);

        /* Poll for incoming GVRP L2 frames
           Should be NON BLOCKING ! */
        while (receive_pdu(l2h, &len, buffer, &from_port)){
          pdu_received(len, buffer, from_port);
        }
      }
```

Figure 4: The GVRP main event loop.

5 The Click modular router

5.1 Introduction

Click architecture

Click is a software architecture for building flexible and configurable routers and it consists of a Linux kernel patch and a user-level driver [5, 6]. A router can be configured by using a declarative language readable by humans and easily manipulated by tools. We present an example configuration for an Ethernet switch that supports VLANs.

The configuration describes the *elements* used and their interconnections. An element is a basic packet processing module that implements a simple router function such as queuing or scheduling. By writing new elements it is very easy to extend a Click router. We developed a new element that supports VLAN tagging and switching. This new element combined with other standard Click elements makes the Click a fully VLAN compliant switch. This VLAN element was contributed back to the Click project [7].

The impact of the flexibility and the ease of configuration and extension of a Click router on the its forwarding performance is investigated in section 5.2. This section contains measurements of the forwarding performance of a Click router compared to a standard Linux router.

Kernel Environment

The Click kernel module uses Linux's /proc filesystem to communicate with user processes. To bring a router on line, the user creates a configuration description in the Click language and writes it to /proc/click/config. Other files in /proc/click export information about the currently installed configuration and memory statistics. When installing a router configuration, Click creates a subdirectory under /proc/click for each element. This subdirectory contains that element's handlers. The code that handles accesses to /proc runs outside the Click driver thread, in the context of the reading or writing process.

Click GUI

Because a Click router consists out of elements which have to be connected to each other, it is rather straight forward to configure such a router via a GUI (Figure 5), if you understand the working of the Click router. Details of the configuration are given in the next section.

Figure 5 shows a Click configuration for a IEEE 802.1d-compliant Ethernet switch with VLAN support as visualised by the Click GUI is developed internally by IBCN [8]. Some elements are related to the learning functionality of the Ethernet Switch, others are used to make the bridge VLAN aware. So it acts as a IEEE 802.1q-compliant bridge and participates with other bridges to determine a spanning tree for the network.

If the ExtEtherSwitch element is used alone, it acts as a simple, functional learning bridge. It differs from a normal EtherSwitch element by means of the IEEE 802.1q support. Doing so, the interfaces of the bridge only accepts VLAN-tagged packets with a VLAN ID that is subscribed in the VLAN forwarding database of the interface.

The other ExtSpanTree and Suppressor elements are necessary only to avoid cycles when multiple bridges are used in a LAN. ExtEtherSpanTree implements Spanning Tree Protocol for constructing a network spanning tree; it works by controlling the Suppressor elements. Suppressor normally forwards packets from each input to the corresponding output, but also exports a method interface for suppressing and unsuppressing individual ports. Packets arriving on a suppressed port are dropped. ExtEtherSpanTree uses this interface to prevent ExtEtherSwitch from forwarding packets along blocking links of the spanning tree.

The ToHostSniffers elements are used to pass through the GVRP messages to the GVRP daemon. Note that only GVRP messages are sent to the daemon otherwise the daemon would be flooded with layer 2 packets. The GVRP message can easily be recognised by their MAC address (01:80:C2:00:00:21).

More details can be found in section 6 “Putting it all together’, but before we look at the integration of the Click and the GVRP daemon we evaluate the performance of the Click router.

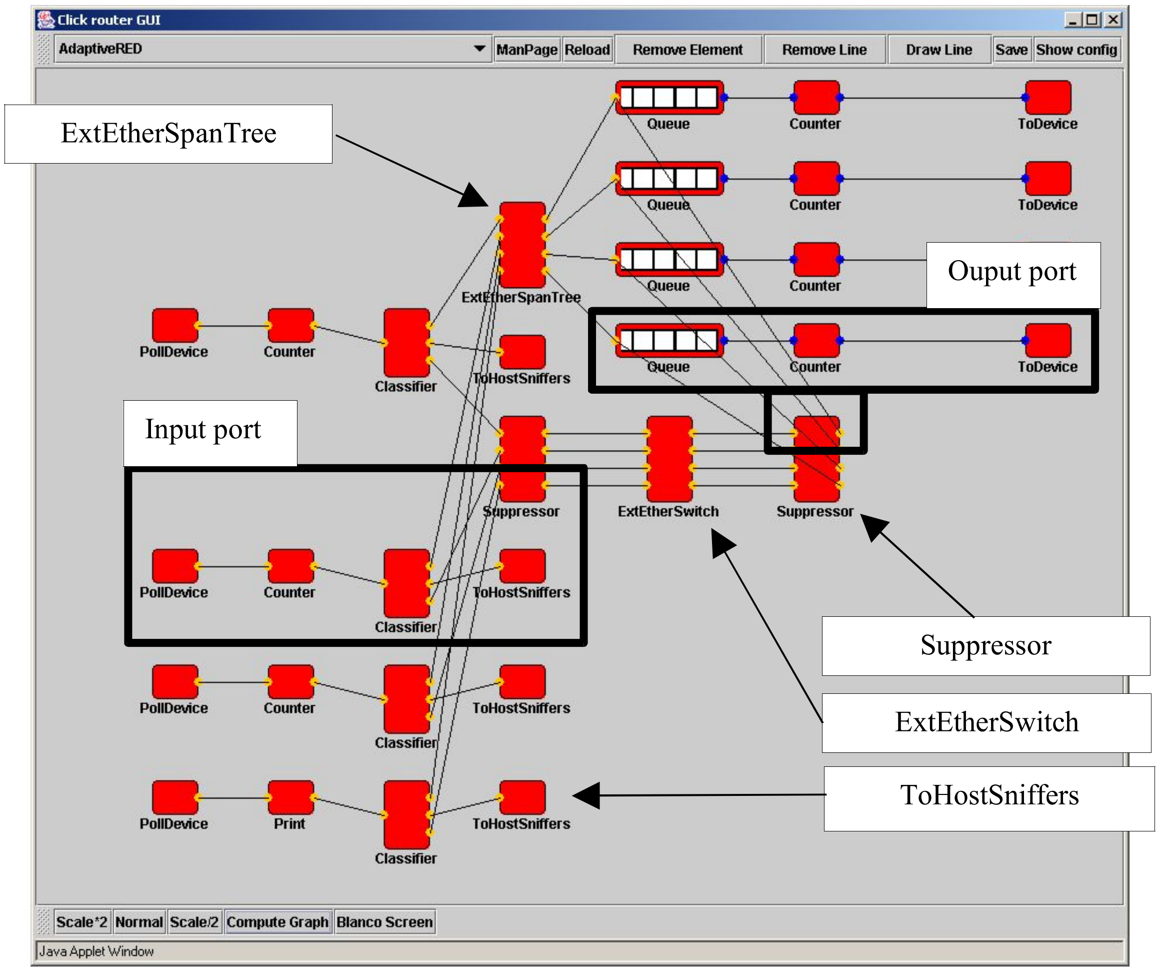

Figure 5: Click Configuration for Ethernet Switch with VLAN support

5.2 Evaluation

In this section we look at the performance of a Click router compared to a standard Linux kernel.

The setup of the experiment

The tests were done with a Smartbits professional analyser [9] with 4 100/10 Mbit/s interfaces. For all tests, UDP packets were sent with a size of 64, 128 or 256 bytes. These lengths include Ethernet (14 bytes), IP (20 bytes), UDP (8 bytes) header and Ethernet CRC (4 bytes). One has also the 64 bit Ethernet preamble and the 96 bit interframe gap. The tests were done for 10 seconds for loads varying from 10% to 100%. All links are 100 Mbit/s full duplex with crossed cables.

Click can use polling instead of interrupts to detect packet receival. The DLINK DFE-570 (4 port) is used because the tulip driver supports polling in the Click router. Also the DLINK DFE-530 cards are used to see the difference with non-polling. The purpose of this experimental setup is to measure the difference between a plain Linux router and a Click router, with or without polling in Click.

We did three measurements: the first one with a plain Linux Router without polling devices, the second one with a Click router but still without polling devices and the last one also with the Click router but now with the polling devices.

The results

Figure 6 shows the throughput of the tested routers versus the load of the machine under test. All the values are given in million bits per second. First of all the ideal graph is shown: a load of 10 Mbit/s gives a throughput of 10 Mbit/s and a load of 100 Mbit/s gives a throughput of 100Mbit/s.

The worse result is achieved by the Click router without polling devices. This setup can sustain a good performance for a load up to 20 Mbit/s. Above this load, the throughput falls to 0 Mbit/s. A slightly better performance is achieved by the plain Linux router. It can sustain a throughput below 30 Mbit/s, but above this value the performance drops slowly to 0 Mbit/s. The Click router with polling shows us a much better result. The graph follows the ideal one until a load of about 80 Mbit/s. A higher load leads to a saturation of the throughput to about 85 Mbit/s without dropping to zero.

Click without polling and Linux suffer of the *Livelock problem* under high loads [10]. When an enormous amount of packets arrives, the CPU only processes the interrupts without forwarding any frames, so throughput drops to zero. The Click routers based on polling do not suffer for this livelock so there throughput does not drop to zero on high loads.

Note that the Linux results are obtained on a kernel without NAPI [11] support. It will be very interesting to compare the behaviour of the Click with polling with a plain Linux kernel with NAPI. At the least these result show the need for more advanced interrupt handling available through NAPI.

Also note that the Click performance with polling is inferior to the Linux forwarding. The flexibility and abstraction of the Click architecture does have a penalty on the forwarding performance. Other tests (not shown here) reveal that the Click also has a higher latency than the standard Linux kernel.

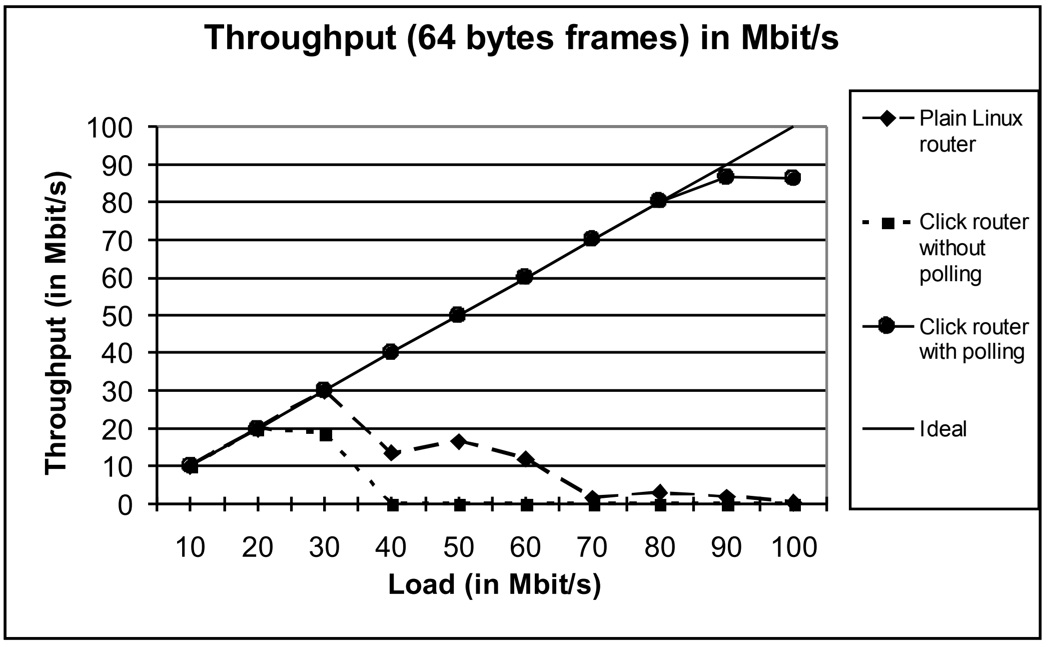

Figure 6 Throughput versus load on a plain Linux router, a Click router with polling and a Click router without polling compared to the ideal throughput.

6 Putting it all together

So far we have discussed the GVRP signalling and the Click VLAN forwarding. In this section we will briefly look at how the two components are combined.

6.1 Adding a VLAN forwarding entry

Thanks to the Click's use of a proclike filesystem adding a VLAN is very easy as illustrated by the following code fragment:

```
Boolean set_VLAN_member_click(Octet port_id,Int16 VLAN_id){

  FILE* file = fopen("/proc/click/esw/addVLANtoPORT", "w");

  if (file == 0){
       perror("Unable to open CLick config file");
       return False;
  }

  fprintf(file, "%d %d", VLAN_id, port_id);
  fclose(file);
  return True;
}
```

Figure 7: This function illustrates how a VLAN is added from the GVRP daemon.

Adding a VLAN is as easy as writing the VLAN ID and the port number to the correct file. Similar functions are available to remove VLAN forwarding entries and to query the spanning tree.

6.2 Sending and receiving packets

As mentioned in the Click section, the Click is responsible to send the GVRP L2 frames to the GVRP daemon. It uses the Click element "ToHostSniffers" to do that. In order to receive the GVRP frame in the daemon a pcap packet filter socket is used. Using pcap sockets is very easy if one uses libpcap library [9].

Sending GVRP L2 frames is done with the Libnet library [10]. Sending a GVRP frame with Libnet is as easy as filling the MAC header and pointing to the correct buffer that contains the payload.

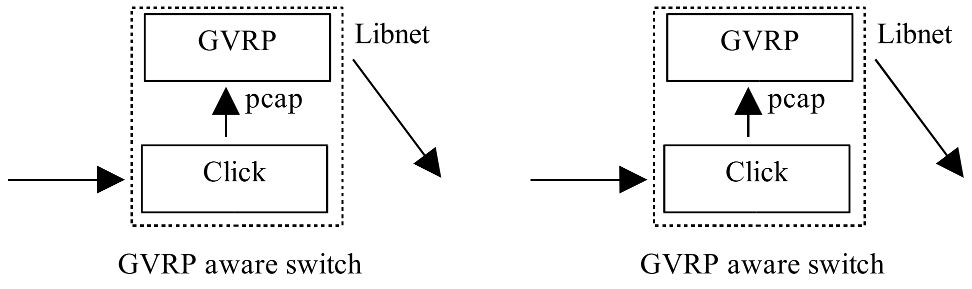

Figure 8: The sending and receiving L2 frames by the GVRP daemon

Note that due the use of the available libraries the sending and receiving of the VLAN packets was very easy. The contrasts with having to use raw sockets and obscure socket options to achieve the same result.

7 Conclusion

First we explained why VLANs and GVRP are used. Then we mentioned the current support for VLAN and GVRP under Linux.

Further on we discussed the technical decisions made to implement a GVRP daemon. We discussed how the example code was extracted from the IEEE standards. How the timers, logging and administrative console were implemented. Finally we discussed how these elements are combined in the main event loop. From the explanation it should be clear that it is very important to use the proper libraries if one wants to speed up the development.

Afterwards we introduced the Click modular router. The architecture of Click is very interesting if you need to modify it, e.g. writing new components or combining the existing components in different ways. It gives also a good insight in the working of a router. The Click router outperforms a Linux router that does not use NAPI. However without polling Click router is a little slower because of the use of the modular architecture.

The section following that explained how the Click and the GVRP daemon are able to communicate with each other.

In conclusion, this paper presents a number of techniques that can be used to reduce the time to develop a prototype of a networking protocol. This is illustrated with the GVRP protocol but these techniques can also be applied to other network protocols and are presented here because they might be useful for other developers.

8 References

[1] IEEE 802.1D, “Standards for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges”, 1998.

[2] IEEE 802.1Q, “Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks”, 1998.

[3] “Linux Ethernet bridge tools”, [Web site], http://www.math.leidenuniv.nl/~buytenh/bridge.

[4] Ben Greear, “VLAN Setup and Configuration”, [Web site], http://www.candelatech.com/~greear/vlan.html#setup

[5] PDOS and ICIR, “The click modular router project”, [Web site], http://www.pdos.lcs.mit.edu/click/

[6] E. Kohler, R. Morris, B. Chen, J. Jannotti and M. F. Kaashoek, “The Click modular router”, ACM Transactions on Computer Systems, Volume 18 number 3, p236 –297, 2000.

[7] “The Click mailing list”, [Mailinglist], https://amsterdam.lcs.mit.edu/pipermail/click/, August 2003.

[8] INTEC Broadband Communications Network, “IBCN web site”, [Web site], http://ibcn.intec.rug.ac.be

[9] Spirent Communications, “Smartbits”, [Web site], “http://www.spirentcom.com/analysis/product_line.cfm?pl=33&WS=200&wt=1

[10] J. Mogul and K. K. Ramakrishnan, “Eliminating receive livelock in an interrupt-driven kernel”, Winter USENIX Conference, January 1996.

[11] J. H. Salim, R.Olsson and A. Kuznetsov, “Beyond Softnet”, [Online], ftp://robur.slu.se/pub/Linux/net-development/NAPI/

[12] “bibpcap software entry at Freshmeat”, [Web site], http://freshmeat.net/projects/libpcap/?topic_id=809

[13] “Libnet software entry at Freshmeat”, [Web site], http://freshmeat.net/projects/libnet/?topic_id=43%2C809%2C150

9 Acknowledgements

Part of this work is supported by the Flemish Government through the IWT-project PANEL (IWT020052) and IWT scholarships.