Pierre Ynard

December 2006

Virtual Ring Routing : a Port to GNU/Linux

Project Report
CS 218 – Fall 2006

Contents

Introduction

VRR, or Virtual Ring Routing, is a network routing protocol with a fresh design. It is inspired by overlay routing algorithms using Distributed Hash Tables, but instead of being built at the application layer, it is a routing protocol implemented directly on top of the link layer. VRR provides both traditional end-to-end routing, and DHT key-access functionalities.

The main characteristics of VRR are that it does not flood the network to acquire routing information, but it uses topology-independent information, sorted in a DHT-like ring: nodes store some routes toward endpoints, and when forwarding a packet, forward it to next hop along the path to the known endpoint the closest to the destination *on the virtual ring*. It is scalable and does not need external setup servers, and is intrinsically robust by the use of DHTs. Thus, it is particularly useful for routing in mesh wireless dynamic ad hoc network, disaster relief networks… Indeed, VRR performs very well across a wide range of network environments and workloads.

The Network Research Lab in UCLA is currently experimenting in VANETs. They consider using VRR on virtualized environments onboard cars. The current implementation of VRR has been made by Microsoft for the Windows XP kernel, and we have access to its source code: I was given the task of porting this code to the GNU/Linux environment.

I. Microsoft's implementation of VRR

1. Overview

VRR has been designed and developed in collaboration with the Microsoft Research team, and its initial implementation has been released in September 2006, under the version number *Release 1.0*. This is an experimental kernel-mode driver for Windows XP. The release contains the source code and binaries of the driver itself, and also of C# user-level libraries and applications. It is distributed under a Microsoft Research Shared Source License Agreement (MSR-SSLA). The key points of this agreement are:

- A license for non-commercial use
- Distribution of derivative works is authorized only under the same terms.
- Upon distribution of derivative works, Microsoft is granted back a free non-exclusive full license on them.

The driver code itself amounts to approximately 30000 lines of code, written in C. It is quite documented, although there is no real general driver overview documentation, such as a developer `README` file.

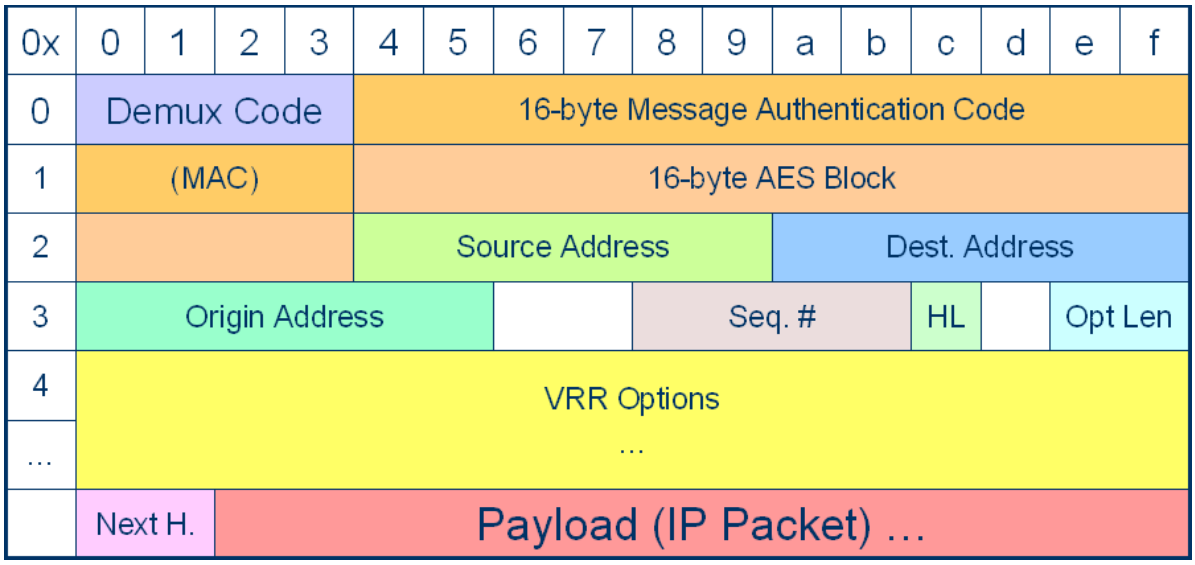
As the documentation explains, the VRR stack takes place on top of the link layer, and allows running Windows IPv4 and IPv6 stacks transparently on top of it: it is nominally a 2.5-layer protocol. It exports an Ethernet-like virtual interface to the IP stacks, and fits in Ethernet frames as a network protocol, using the Microsoft-assigned Ethertype `0x886f`. Likely for

obvious compatibility reasons, VRR uses 48-bits virtual addresses, of the same format as classic MAC addresses.

However, an important limitation to VRR is that it does not support broadcasting and multicasting. Subsequently, it is impossible to directly operate address resolution on the virtual ring: ARP and NDisc have to be managed separately. Furthermore, neither is it possible to use autoconfiguration protocols like DHCP.

2. The VRR header

In practice, when a frame is received from IP stacks by the VRR virtual Ethernet device, a VRR header is created and prepended to the IP packet, and the new packets is re-encapsulated in a frame and sent on the physical media. The structure of the VRR header is as follows:



A VRR header

The Ethertype currently used for VRR packets is `0x886f`, an Ethertype assigned to Microsoft. It is also used by several other Microsoft protocols, for example LQSR or NLB. So it is necessary to demultiplex those protocols, so that the VRR driver will only take into account VRR packet: that is the purpose of the initial 4-byte demux code.

When the VRR driver is loaded, a MAC address is assigned to the virtual Ethernet device that it exports: this address will be the virtual address of the node on the ring. When a frame is first transmitted by this device, the source and destination Ethernet addresses are considered as respectively the origin and destination virtual addresses on the ring, and replaced into the corresponding fields of the VRR header. Additionally, each time a node forwards a VRR packet, it fills in the “Source Address” field with its own virtual address.

The VRR header also features a frame sequence number for identifying frames, a hop count field, and a variable number of VRR options. The original IP packet, along with its

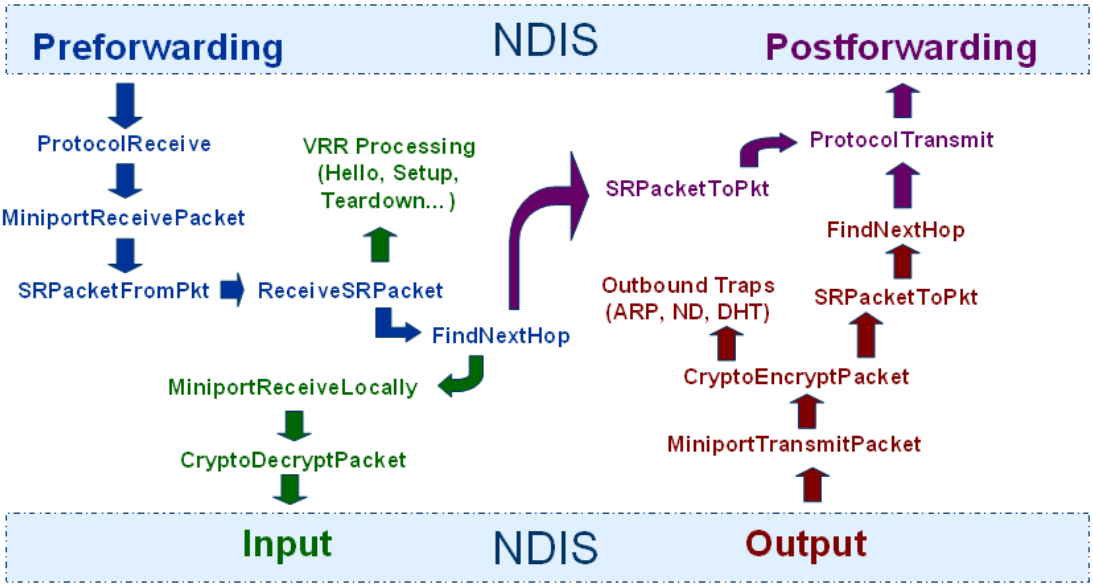
original Ethertype (stored in the “next header” field), then follows. VRR options are control messages for the VRR network, like Hello advertisement, Setup Requests to another node, or Teardown of an existing path. In a “pure” VRR control packet, there is no next header field and payload.

VRR supports encryption of packet payloads. When encryption is enabled, the original Ethertype and payload are encrypted and decrypted by endpoints, and at each hop the Message Authentication Code covering the whole packet, including the VRR header, is recomputed and then verified.

As we can see, the VRR header is poorly aligned. Actually, the VRRHeader struct is even defined using architecture-dependant data types, with padding automatically added for alignment. Assuming that the all nodes are supposed to use the same packet format, it seems that unfortunately, Microsoft developers were not really concerned with interoperability at the moment (indeed i386 architecture is free of alignment constraints). Hopefully, in a more stable version were a unique Ethertype would be used for VRR, the header could be redesigned.

3. Packet flow

The VRR driver takes place into the Windows kernel as a Miniport Virtual Adapter, bound to several Protocol Adapters on physical devices. It interacts with a key component of the Windows networking stack, which is NDIS, or Network Driver Interface Specification. Both incoming and outbound packets are handled off to the VRR driver by NDIS kernel threads, then processed by the VRR stack, and handled back to NDIS. The packet flow follows the scheme below:



Packet flow through the VRR driver

In short, received packets are parsed and stored into and internal SRPacket struct to be processed. If it is a control packet, it is processed by the VRR stack. Otherwise, a lookup is performed in the internal VRR routing table and node table, to determine the next hop: then packet is then either received locally, or forwarded to another node on the ring.

4. Outbound packet trapping

An interesting feature is that some packets will be directly trapped and intercepted by the VRR driver. Those packets are: ARP requests, ND packets, and DHT applications' packets.

a. Address resolution packets

As we said before, since VRR does not support broadcast, it is impossible to perform ARP resolution. Thus, output ARP requests are trapped, and looped back to userland in a UDP packet: then, they can be received and processed by a DHT application that will manage the ARP cache.

The same problem occurs for IPv6 Neighbour Discovery. But with IPv6, there is a similarity between hardware addresses and autoconfiguration link-local addresses, that is used by the driver. Normal ND requests are sent to a link-layer multicast address in 33:33:ff:0:0:0/24. When the VRR sees a ND request, it guesses the destination hardware address from the requested IPv6 address, and uses it as the destination virtual address of the packet in replacement of the multicast address above, and forwards the packet to the ring. This allows to still operate successfully limited NDisc.

b. DHT applications' packets

The third category of trapped outbound packets is DHT application packets. It provides an easy way to interact with the network to application using the DHT. As an example, we can take the VROOM project of the NRL: patrolling police cars maintain a VRR network to communicate and collect information. Let us say that a police car wants to contact another police car, but only knows its key on the DHT, that is, its virtual address. It cannot use an IPv4, or IPv6, address to contact it. Instead, it transmits a packet to a "magic" IPv6 address.

```
// Magic addresses used to support DHT functionality on VRR.
// These addresses are real addresses taken from host msrc-gregos05.
// Therefore nobody else should ever legitimately conflict with them.
//
IPv6Addr DHTtxAddr = {0xfe, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                      0x02, 0x0d, 0x56, 0xff, 0xfe, 0x6d, 0xf0, 0x2c};
IPv6Addr DHTrxAddr = {0xfe, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                      0x02, 0x04, 0x23, 0xff, 0xfe, 0xa3, 0x21, 0x13};
PhysicalAddress DHTMagicL2 = {0x00, 0x0d, 0x56, 0x6d, 0xf0, 0x2c};
```

Sample code

It is an address that does not really exists on the network. First, it sends an ND requests for this DHTtxAddr: the VRR driver traps this ND, and replies with a fake NA binding the IPv6 address to the Ethernet address DHTMagicL2. Then, it sends the DHT packet toward DHTtxAddr/DHTMagicL2. The real destination virtual address is enclosed by the DHT application in the packet. The DHTMagicL2 address is again trapped by the driver, and the driver looks up in the packet in the DHT header, for the real destination, and then replaces DHTMagicL2 by the right address and forwards the packet, toward this time the DHTrxAddr IPv6. Every node in the VRR has the same DHT application binding this DHTrxAddr, which receives and processes the packet.

Instead of contacting another node, DHT packets can also be used to query the local driver: when it detects that the DHT packet is a local query, the driver fills the response in another DHT packet, which is looped back to userland. This query mechanism is an alternative to the also available classic IOCTL calls.

II. Design for a Linux port

1. Kernel-space vs. user-space

When porting the VRR driver to Linux, the conditions and constraints in the NRL experimentations and in the coding environment poses the question of whether it should be ported into the Linux kernel, or as an user-space application. Indeed, although a stack operating in the kernel causes less overhead, developing a kernel driver is a tedious work, since it requires lots of recompiling, and hard debugging.

Moreover, from the legal point of view, I am not working on clean-room re-implementation at all, but merely porting code, that is licensed by Microsoft under the MSR-SSLA. Clearly, this license is not free and incompatible with the GPL (General Public License) of the Linux kernel. This is clear that this driver will never be part of the official kernel, and at best, it may be only possible to write this driver simply as an external non-free kernel module. But that would mean, for the user too, building and inserting an external kernel module, whose compatibility is not maintained by the official kernel developers, which is much less convenient than running a user-space application.

In the Linux user-space, it is possible to build non-free applications, using the user-space kernel APIs, and linking to LGPL (Lesser GPL) libraries, including the GNU C Library. For those reasons, I took the decision to look for a user-space design for the VRR Linux port.

2. Universal TUN/TAP device driver

The main feature that allowed me to do so is the Linux kernel universal TUN/TAP device driver. It allows packet reception and transmission for user-space programs. By opening the `/dev/net/tun` device file, user-space applications can register virtual point-to-point (tun) or Ethernet (tap) devices. Packets or frames transmitted to those interfaces are then written to the user-space program which can read them through a file descriptor, and vice versa.

Examples of applications currently using this TUN/TAP driver are NSTX, which sets up IP over DNS tunnel, or simply OpenVPN. This kind of behaviour is exactly what we are looking for, so we can set up a tap as our VRR virtual Ethernet device.

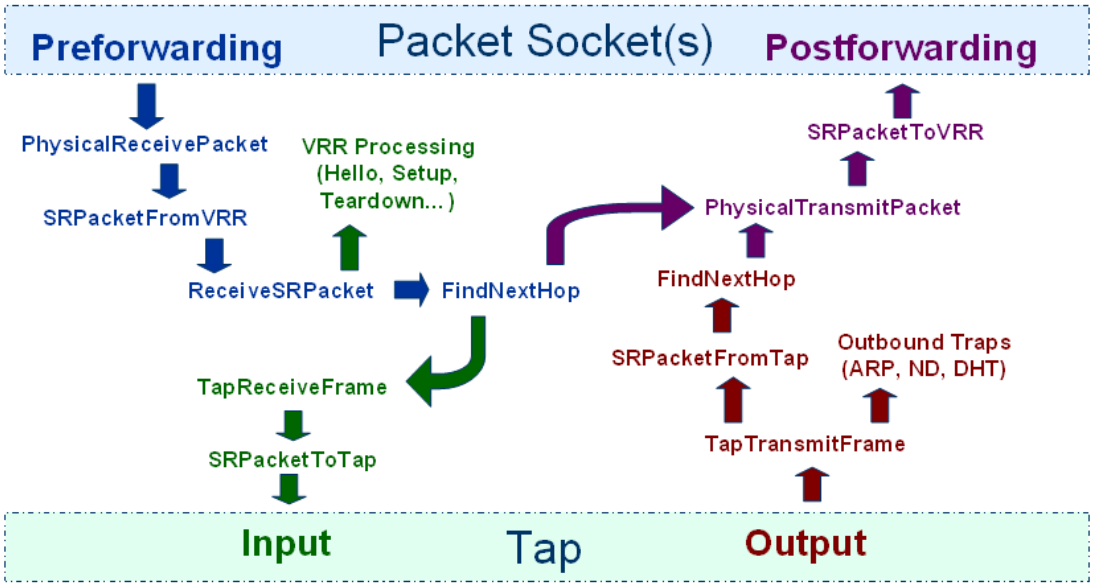
3. Packet socket

We still need to send and receive VRR packets through physical devices. For this purpose, we can use sockets of the `AF_PACKET` family. This family of sockets is used by applications that communicate with the network devices without an intermediate networking stack in the kernel, for example network monitors like `tcpdump`. It allows to select a given network protocol designed by its Ethertype.

In our case, we would use: `socket(AF_PACKET, SOCK_DGRAM, htons(ETYPE_MSFT));` and send and receive raw VRR packets on top of the link layer. This socket can either operate through all interfaces without being bound, or be bound to a specific interface if we want control over devices involved in the VRR or not.

4. Packet flow

Rewriting the network I/O functions to use these tools, the packet flow becomes this:



New packet flow of the Linux port

5. Design consequences

Those radical changes in the design of the VRR driver have some consequences. First, execution thread management is completely different. Instead of being handled packets by kernel threads, and handling them back in an asynchronous way, we can process the packets synchronously from our own receiving threads. This really simplifies this portion of the code, since asynchronous handling in Windows requires registering handles and passing them to NDIS to free memory when it is done with the packets.

The second main simplification is about network devices: with the TUN/TAP driver, our Tap is managed by the kernel, and can be brought up and configured using usual tools. We do not need to manage its configuration ourselves.

But there are also new functionalities that we will have to re-implement in our user-space application. Along with thread spawning and management, we cannot rely on kernel code to manage packet fragmentation, ICMP errors, and such: we have to do it by hand. Moreover, we need a new command interface in replacement of IOCTL calls. A very usual CLI operating through a local socket would be a good, quite normal design for this user-space application.

6. Current state of the port

At the moment, after the hard work of reading and understanding the code through, I am mostly done porting the main network I/O mechanisms. I have had a small sample program running, which registers a Tap, reads Ethernet frames from it, encapsulates them in VRR packets and sends them through a physical device. From what I understand of the code, this is the most platform-dependant part to port. The other main parts left which need future effort are porting the routing and node tables, and designing the new I/O interface.

Conclusion

After going through the code, it seems that Microsoft's VRR implementation is of quite good quality. Though it is a quite different approach, the Linux port to user-space should provide a convenient and portable way to experiment with VRR. The VRR implementation features quite handy DHT functionalities, which will be useful to the NRL in his future works, such as the VROOM project.

References

M. Caesar, M. Castro, E. Nightingale, G. O'Shea and A. Rowstron, *Virtual Ring Routing: Network routing inspired by DHTs*, Sigcomm 2006, Pisa, Italy, September 2006.

M. Castro, G. O'Shea and A. Rowstron, *Zero Servers With Zero Broadcasts*, MobiShare 2006, LA, USA, September 2006.

M. Gerla, *VROOM: Virtual Ring Routing and Repository - Open Overlay Mesh*, November 2006