Pierre Ynard

July 2007

VRR: a study of design and implementation

M.S. Comprehensive
UCLA 2006-2007

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. It performs well across a wide range of network environments and workloads, but is especially intended for routing in mesh wireless dynamic ad hoc networks, or disaster relief networks and such: indeed, it is scalable, robust, and does not need external setup servers.

VRR has been designed and developed in collaboration with the Microsoft Research team, and its reference implementation has been released in September 2006, for the Windows XP operating system. It is distributed through a shared source agreement. The Network Research Lab in UCLA has been experimenting in VANETs, and they consider using VRR on virtualized environments onboard cars. As a term project in the Fall Quarter, I was initially given the task of working on a port of this code to the GNU/Linux environment [3].

Since then, I have gone on further working on VRR, on design and implementation issues. In particular, as a term project in the Winter Quarter, I designed multicast extensions to the VRR protocol [4]. This document summarizes and presents a review of all my work, as a M.S. Comprehensive report.

I. Protocol design

1. Main features

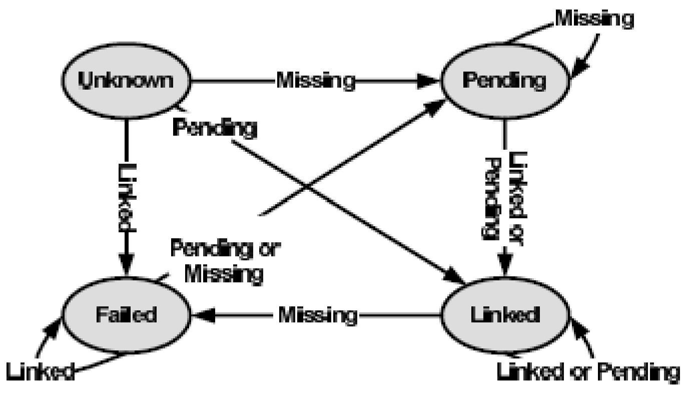
After having worked for some time with this protocol, it appears to me that the most striking features of VRR are:

- the distribution of routing information over the network, in the face of the requirements of routing
- symmetric failure detection, and more generally,
- the hard-state of the protocol, and the fact that it does not flood the network to acquire routing information

In VRR, each node is identified by a so-called *virtual address*, and these addresses are sorted on a virtual ring: identifiers are called virtual in distinction from the physical topology, of which they are independent. Each node holds some partial routing information, which comprises routes toward several endpoints. When forwarding a packet, the node uses the next hop of the route whose endpoint's address is numerically the closest to the destination address. Depending on the routing information that the forwarding node holds, the next hop may not be the most accurate with respect to the physical topology.

However, in order to achieve routing, each node is responsible for maintaining routing information to a set of a few nodes which are numerically closest on the virtual ring: this set of virtual neighbors is called the *virtual set*. Through one-time exchanges of *Setup* messages with their virtual set, virtual neighbors set up routing information all along the paths between them. This information is torn down when a failure occurs, and set up again by the virtual neighbors. Thus, by routing according to the virtual ring topology, VRR can rely on routing information within virtual sets, truly distributed over the ring: this is a key to the scalability of VRR.

One may imagine that, in order to detect failures, virtual neighbors exchange some kind of end-to-end heartbeat messages: this is not the case, as the protocol is essentially hard-state, which means that routing information is not periodically refreshed, and then passively removed when its lifetime expires, but instead, once acquired, is kept valid until an active contrary update is received. In such a distributed environment, the main point where consistency can be achieved is the local environment of a node. VRR nodes broadcast *Hello* beacons, to detect and track the state of their neighbors in the physical topology. These *Hello* messages are coupled to state machines: the state transitions ensures that if a node detects that a link with one of its neighbors is failing, both sides will detect and report the failure, and consistency will be preserved. Routing information establishing paths between two virtual neighbors that pass by the failing link can then be removed, by forwarding *Tear Down* messages along the paths, toward both endpoints, to report the failure in a reliable way.



State transitions when a Hello message is received

Distributed information and hard-state are indeed a quite beneficial combination. A third element, depending on the context of the use of VRR, can be added.

VRR is not really intended for global routing, like the Internet Protocol, but rather for use within an organization, with a pre-arranged configuration. Moreover, the principles of VRR are to use neither flooding of information, nor external reference servers. In this situation, to retrieve some piece of information on a VRR network, user-level applications can be arranged to take advantage of Dynamic Hash Table functionalities, inherently supported by the virtual ring structure of VRR. With the use of pre-deployed DHT applications on the nodes composing the network, VRR can be extended to further distribute user-level information on the ring.

As addressed in [2], this scheme can be used to operate autoconfiguration protocols like DHCP, or resolution protocols like DNS or ARP for IPv4: as said before, it allows VRR networks to run in the absence of a support infrastructure, for example in a disaster relief scenario, and without the poor performance implied by broadcasting requests through the network.

Indeed, the reference implementation of the VRR stack provides applications with extra mechanisms to easily access DHTs on the VRR network, and is shipped with such applications implementing ARP, DNS and DHCP. Though VRR really is a network protocol, standing directly on top of the link layer, the VRR network is exported as an overlay link layer, accessible through an Ethernet-like device, on which IPv4 and IPv6 stacks can be run transparently. As the idea behind VRR is never to broadcast a request through the ring, in order to mimic the behavior of a link layer, the VRR driver will in particular intercept broadcast ARP requests, and redirect them to user-space for DHT applications to handle them, as mentioned above. This means that VRR does not really need to support real broadcast or multicast to operate in normal conditions, and in fact, it does not support them at all.

However, this is still a limitation. In cases where broadcast is used to query a unique, yet unknown, node on the network, it can be replaced by DHT applications. But when the purpose is to transmit the same data to several or all nodes at the same time in an efficient way, broadcast and multicast are still missing. This problem has been the motivation of my work on the design of broadcast and multicast extensions to VRR.

2. Multicast extensions

When working on these multicast extensions, I essentially tried to conform to the design goals and decisions of the existing VRR, and take advantage of its particular features, to produce a scheme that was consistent with the original protocol: something scalable, hard-state, and that did not require flooding of information.

Among the classic ways of doing multicast, one approach was particularly suited to this case: like in PIM [5], to use a *rendezvous* point (RVP), and then rely on existing unicast routing. Indeed, for streaming sources and group members to contact each other, they need either to flood the network toward each other, or to meet at a RVP: on VRR, the DHT functionalities can easily be taken advantage of to implement RVPs and store group membership information in a distributed way. Then, as for multicast forwarding, each node holds only partial routing information, which is not enough to coordinate efficient multicasting by itself; however, existing unicast VRR routes can be used to forward messages, for example from a member toward a source, and set up a forwarding tree along the path.

This is a two-step process, where first the RVP is contacted to notify it and fetch existing group information, and then messages are sent to build a distribution tree. For scalability reasons, it is better if both endpoint types, that is, both sources and members, can initiate this process. Thus, I designed two additional VRR option types with several subtypes corresponding to those messages:

1. *Multicast Membership* messages
   Register, *Members* and *Withdraw* messages between a source and a RVP
   Join, *Sources* and *Leave* messages between a member and a RVP

2. *Multicast Tree* messages

    Seed and *Fell* messages from a source toward members
    Branch and *Prune* message from members toward the source

Subsequently, for each multicast group and source pair, each node holds an entry in a multicast forwarding table, which consists only of a list of next hops to which packets must be forwarded: since the information kept at each node is relatively small, the scheme is scalable. This information is kept in hard-state: existing symmetric failure detection is used to detect failures, and repair the tree.

I found that this scheme is not exempt of synchronization issues. Essentially, consistency can be achieved only at two points: at the RVP, and in the local environment of each node. The RVP can easily be used as a reference point, where membership information is kept in a coherent state. In fact, the only serious problem with possible race conditions is when two neighbors along a tree exchange opposite messages at the same time: a branch of the tree could be orphaned by its parent without knowing it, and fail to receive packets. This problem can be solved through the use of sequence numbers for *Multicast Tree* messages, to acknowledge or retransmit, and order those messages, and then apply a few simple synchronization rules to eliminate harmful race conditions.

To complete the review of this design, I treated the broadcast case as a special case of multicast, where all nodes are implicitly members of the group. In the end, this scheme is interesting and seems feasible, though at the moment it is only partially implemented, and neither really tested nor evaluated.

II. The Microsoft Research reference VRR implementation

During my work on VRR, my basis was essentially the code source itself of the reference implementation from Microsoft Research. Thus, I have come to some observations about it that I will expose here.

1. Overview

This implementation was 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 shipped with installation and start-up guides, but no real general driver overview documentation, such as a developer `README` file. The two VRR papers [1] and [2] are the only extra documents helping to understand the driver.

Fortunately, the code itself is quite well commented. At first, I was quite pleased by the external appearance of the code, which looked well-written, clearly commented and using good coding styles. But after working more in depth with the internals of the stack, I became quite disappointed by its quality. There are a number of points on which great improvements could be made: though it is experimental, the code does not meet the quality standards that I would have expected for a release, and, in my opinion, would rather qualify as pre-release stage code.

2. Lack of clean-ups

Indeed, when reading through the code, it is not rare to come across comments like "TODO: clean this up" or "TODO: do this in a better way." Also, obviously, the stack makes intensive re-use of code from the Microsoft Research Mesh Connectivity Layer, and especially the Link-Quality Source Routing (LQSR) stack: in itself, this code re-use is not a bad thing, but it contributes to explain the state of the code. The most pervasive example is the struct used to internally represent a VRR packet: it is called `SRPacket`, for Source Routed Packet, whereas VRR has really nothing to do with source routing. Beyond simple naming questions, other examples of things that could have been cleaned up before the release are:

- Flags describing VRR node states, that are defined in the headers but never used in the code
- Variables declared in functions, but never used, that are easily caught by turning on compiler warnings
- Old LQSR definitions, still present in the code, and marked as "to be cleared"
- Not only debug output, which is fine, but debug logic still present in the code
- Duplicate code, sometimes nearly whole functions; at other times, on the opposite, helper functions split far away into separate files, whereas they are only called once in the entire stack. This is not only cosmetic: when loaded, duplicate code consumes extra memory, which is a precious resource in kernel-space.
- Header declarations of all modules, that are gathered in a single huge file, instead of being clearly organized
- Lack of refinement in handling: for example, when parsing packet headers from the network, unknown options cause the entire packet to be rejected instead of being silently ignored.
- An identified, documented, and yet uncorrected bug that may crash the system, causing what is commonly known as a "Blue Screen of Death"

Some of these points may be details, but in overall, they constitute a disappointment in what should be professional-grade software.

3. Code complexity

Another problem that I encountered is that the internal workings of the stack seem terribly complicated. As an example, only for routing information needed to operate the protocol, the stack maintains no less than six tables:

- a Neighbor Cache: similar in function to an ARP cache, tracks the state of physical neighbors and operate mappings between virtual and physical addresses of next hops
- a Node Table: represents individual nodes on the ring, actual purpose still vague to me
- the Route Table: holds routing information, toward endpoints in the Node Table, through next hops in the Neighbor Cache
- a Tear Down Cache: used when sending Tear Down messages to remove stale routing information from the network
- a Probe List: used for metric probes
- a Zero List: used for repairing the ring after a partition

Entries in those tables hold references, with reference counts, to each other, have their own spin locks, their own timers and “housekeeping” periodic functions, and state changes of entries in one table triggers updates of related entries in other tables. There is no overview documentation describing the role and interactions of all these tables, and it is not easy for a newcomer to understand them by reading the code.

To be honest, part of the complexity also comes from the kernel context of the driver. The stack is not running by itself as a process, but called through kernel threads, hence the need for reference counts and numerous call-back functions, for example to free memory after other layers of the kernel be done with it. Unfortunately, part of the complexity is also due to the way that the implementation was written: some functions are filled with more than 500 lines of code of raw complicated business logic. This is plain bad coding practice (and of course hardly understandable).

4. Interoperability concerns

The reference implementation is a Windows XP driver, meant to run only on x86 hardware architectures. Yet, one may expect that the network protocol that it defines be interoperable with other stacks, on other operating systems and architectures. And indeed, as working on a GNU/Linux port of this stack, which could potentially be run on a lot of different hardware architectures, portability was an important concern for me. But in fact, unfortunately, the original implementation does not really respect the constraints which would make this possible. The two main points are endianness and alignment.

Endianness is the way to represent values that span over several bytes of memory, by ordering those bytes one way or the other. The first memory byte can hold the value of either the most significant byte, or the least significant byte. For example, a 4-byte integer value of `0xff000000`(approximately 4 billions) will be represented in memory as `0xff`, `0x00`, `0x00`, `0x00` on big-endian architectures, and as `0x00`, `0x00`, `0x00`, `0xff` on little-endian architectures. The x86 architecture is little-endian; PowerPC or SPARC are big-endian. For different hosts with different architectures to interoperate the same network protocols, values that spans over several bytes, typically 2 or 4, in the fields of protocol headers are by convention always converted the big-endian byte-order, also known as the network byte-order, before being sent on the wire.

0x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	Demux Code				16-byte Message Authentication Code											
1	(MAC)				16-byte AES block											
2					Source Address						Destination Address					
3	Origin Address						Frame Seq. #				HL	Opt len				
4	VRR Options															
…	…															
	Next H.		Payload (IP Packet) …													

A VRR header

Sadly, the reference implementation fails to do this. Some VRR header fields, like the 4-byte frame sequence number, are indeed byte-swapped to the network byte-order. Some others, like the 2-byte header length, are not, and are sent on the network in the little-endian byte-order. All VRR options in the headers are copied from the memory to the network buffer in a raw manner, and not processed for byte-swapping at all. So to speak, maintaining interoperability with the original stack with portable software would require at least ugly hacks.

The other problem is alignment. Again, when accessing values that span over several bytes, the memory address of this value is expected to be a multiple of its length: for example, 2-byte fields should begin on pair addresses, 4-byte fields should begin on addresses that are multiple of 4… Generally, access to misaligned values causes at best performance losses, and may cause real issues and crashes on some architectures, or even not be supported at all. For this reason, header fields of popular networking protocols are aligned in a right way.

On x86 architectures, alignment is not a requirement; indeed fields in the VRR header are not aligned, which causes performance losses and portability problems.

Moreover, when directly porting the misaligned VRR structs in the source code, compiling them with the popular GCC compiler produces a result that is different from the behavior of the original binaries. The reason is that by default GCC adds padding between unaligned fields to restore alignment. However, there is no indication of this potential difference in the original code, and indeed, I had believed for some time that the behavior of the original stack was to send padded VRR headers on the network. A close examination of some header length checks eventually hinted me at the contrary. At that time, I decided to more deeply consider the structure of the VRR headers.

5. Header analysis

The VRR protocol uses an Ethertype assigned to Microsoft, who have been using it for several other different protocols. A striking fact about the VRR header is that it begins with a 4-byte demultiplexing code, to differentiate VRR from those others protocols using the same Ethertype. This can only let the current header be considered as somehow tentative, and not ready for production. Still, a number of points could already be improved.

The VRR header contains a 4-byte frame sequence number. Typically, sequence numbers are used for reliability, to acknowledge or retransmit packets, or for other ordering and fragmentation purposes. VRR provides none of this: in fact, VRR does provide some reliability for options in its headers, but uses a separate acknowledgement mechanism for this. According to my understanding of the code and the `grep` tool, this field is actually used nowhere in the stack, and could be removed from the headers, along with the pieces of code generating these unique sequence numbers.

Furthermore, a VRR header contains three addresses: the destination address, the source address of the node that generated the packet, called origin address, and a per-hop source address field, which contains the address of the last node that forwarded the packet. VRR does not piggyback control options on data packets, but transmits several possibly unrelated options at the same time on dedicated option packets. Option packets are not forwarded; instead individual options are processed and resent if needed on a hop-by-hop basis. Thus, data packets do not need the per-hop source address, and option packets do not need an extra origin address since they are always one-hop only. Actually, options usually include their own redundant source address information. For these reasons, two source addresses fields are not needed, and one of them could be removed.

A problem already identified in the original code is the length of the length field in VRR options. Every option begins with a 1-byte type field, followed by a 2-byte length field, the problem being that 1 byte can only represent lengths up to 255 bytes, which is not enough for some options that may want to include numerous addresses, and would waste space with respect to the usual 1500-byte MTU. The same goes for the 2-byte header length field in the static header. A common solution to this, used in IP and IPv6 headers, is to count the length in units of 4 or 8 bytes. This makes all the more sense when options are anyway arranged to respect alignment constraints. A 1-byte length field counting lengths in units of 8 bytes can represent lengths up to 2040 bytes, which is greater than 1500 bytes, and is coherent with 64-bit option alignment.

The VRR header and original stack also provide both end-to-end encryption of the VRR payload and hop-by-hop integrity verification, through two dedicated fields. However, when encryption is disabled, this space is wasted. It could be a good idea to make these fields optional, and to support choice between several encryption algorithms and different key sizes.

Moreover, since option packets and data packets are separate cases, the header can be adapted to each case to remove useless fields. Data packets need no header length field, since they carry no options. Option packets have no payload, and do not use encryption; they also are one-hop only, and do not need the hop count field.

Here are a few ideas for new components that could be added to the VRR header:

- a version field
- a flags field, including an "option/data packet" flag, and possibly flags to toggle integrity verification and encryption in the case of fixed algorithms and key lengths
- alternatively, a pair of algorithm-key length fields for integrity and encryption: such a pair can fit in a single byte, with a choice of 16 algorithms and key lengths from 8 to 128 bytes
- for data packets, a 2-byte payload length field, which is somehow missing in the original VRR header
- a next header field. The reference implementation exports the VRR network transparently as an Ethernet-like device, and uses 6-byte virtual addresses for VRR nodes, compatible with classic Ethernet addresses; then it mangle Ethernet frames and encapsulates them in VRR packets, the Ethertype field being considered as part of the payload and encrypted with it. Instead of that, when encryption is disabled, the Ethertype could be moved to the VRR header, to respect the original alignment of the payload. Or, this next header field could be used in different implementations to directly run transport protocols on top of VRR.

This analysis shows that instead of the original 61-byte VRR header, a minimal VRR header without integrity or encryption could easily fit in 16 bytes for option packets, and 16 or 20 bytes for data packets, while respecting alignment constraints: indeed, there is truly room for improvement.

After all these considerations, and along with the fact that implementing multicast extensions would break compatibility with the original stack, I came to the conclusion that in order to provide a decent GNU/Linux VRR stack, in the future it would be better to drop this compatibility, and instead work on a selected basis of good parts of the original implementation.

III. A port to GNU/Linux

1. Main design orientations

When porting this Windows XP kernel module to Linux, one of the first questions was whether it should be ported as a part of the Linux kernel, or as a user-space application written for Linux. The two main arguments are:

- the ease of development in user-space, compared to kernel-space
- from a licensing point of view, the clear incompatibility between the non-free MSR-SSLA of the VRR driver and the GPL of the Linux
    - a port could never be integrated and maintained as part of the official kernel
    - a user-space application is much more convenient than an external kernel module for administrators and users

Thus, I decided to port the VRR stack as a user-space application.

The two main tools that I used to port such a networking stack to user-space are the universal TUN/TAP device driver, and the packet socket.

The TUN/TAP driver is a kernel module whose purpose is to connect a virtual networking device to a user-space process: in the VRR case, it can register an Ethernet-like device, and relay transmitted and received Ethernet frames to and from the VRR process, through reads and writes on a file descriptor. It is totally suited to the needs of this implementation of VRR, which are to transparently export an overlay link layer to the normal networking stacks.

On the other side, the `AF_PACKET` socket family allows a process to send and receive raw Ethernet frames. It can be used to build and send VRR packets directly on top of the physical link layer, and can be bound to the specific Ethertype used by VRR, so that it will only receive frames containing VRR packets. Once again, it is exactly what is needed to operate a networking protocol in user-space.

2. Port steps

Using these two tools, my first task was to port the network interfacing part of the VRR stack. The original Windows XP driver is embedded between two layers of the NDIS framework, or Network Driver Interface Specification, which is a key component of the Windows networking stack. I replaced all this code by reads and writes on a TUN/TAP file handle and socket API system calls, converting the code to the use of Ethernet and IP header definitions provided by the GNU C Library.

The second main structural difference is that the stack ported as a user-space process, instead of being run from kernel threads, is to start and run by itself, by spawning and managing its own threads. To do so, I used the popular POSIX thread library, also known as `pthread`library in short.

Then, I began porting internal business logic of the VRR stack. This is a tedious work, especially in front of the complexity and the interaction of the different parts of the code. Porting business logic includes, most of the time:

- converting routine calls like `RtlCopyMemory` to `memcpy`
- replacing kernel spin locks by `pthread` mutexes
- converting Windows kernel time representation to system calls like `gettimeofday` and GNU C Library timer functions

In the mean time, I tried to add some basic UNIX-friendly build system. I was starting from scratch, as the source code was likely only meant to be built using a Windows Development Kit, and at first would not compile at all. I wrote an approximate Makefile, that successfully provides the basic functionality to build the VRR binary by running the `make` command. Given the fact that all the headers where originally gathered in a single, huge file, I also partially reorganized them, but this area needs a serious clean-up.

One last important change from the original implementation is the administration interface. As a kernel driver, the original VRR stack can be accessed using classic IOCTLs, and also through an original API based on intercepting packets send to magic addresses. I have not worked at all on this area. To me, the most natural way would be to write some kind of Command Line Interface, communicating with the VRR process through a local socket.

3. Current state of the port

Until now, I have successfully ported parts of the VRR stack, and produced a working sample application. So far, the functionalities that are ported are:

- network interaction, packet parsing and IPv4 and IPv6 helpers
- partial route table, neighbor cache and node table support
- route table lookup and packet forwarding
- broadcasting of periodic *Hello* messages
- initialization of the stack, and spawning of the different threads

I also implemented parts of the multicast extensions that I designed:

- partial multicast forwarding table support
- sample processing of *Multicast Tree* messages
- multicast forwarding

These functionalities are working properly, and I have successfully achieved one-hop transmission over a VRR network of:

- sample UDP datagrams over IPv4 (by manually populating the ARP cache)
- exchanged IPv6 Neighbor Solicitations and Advertisements, and ICMPv6 Echo Requests and Echo Replies (commonly known as “ping”)
- a 160 kbps MPEG multicast stream, smoothly transmitted in UDP over IP multicast, using VRR multicast

IV. Personal experience

From a personal point of view, working on VRR during all this time has been quite interesting, and rewarding.

First, this has been an occasion to face a concrete case of network protocol design, and apply the ideas that I have been studying in class. Moreover, the study of an atypical network protocol is enriching, as it brought me a view different from classic approaches on networking.

Then, porting code was a very good opportunity for me to learn the C programming language. Indeed, when I arrived in UCLA, I had barely any experience with C. It also allowed me to broaden my knowledge of Linux networking and programming.

On a more practical note, it is a good experience to have been able to assess by myself the quality of source code produced by a software leader like Microsoft, whose products are usually closed-source, and preceded by their reputation.

Conclusion

I consider my work on VRR during this year in UCLA as a very good experience. It allowed me to tackle at the same time both design and implementation issues, in my main field of interest: networking. At the very least, my work resulted in interesting design ideas and directions. The VRR protocol has some interesting, strong features, but sometimes also goes into too much complication, and could use simpler, more straightforward, more robust schemes. It appears to me that the reference implementation was released more as a proof of concept than production software, and could now be greatly improved with more work and efforts. Though my eventual concrete production is less than what I would have expected, I value much the understanding and assessment results that I have reached in the process.

References

[1] M. Caesar, M. Castro, E. Nightingale, G. O'Shea, A. Rowstron
Virtual Ring Routing: Network Routing Inspired by DHTs
SIGCOMM'06, September 2006

[2] M. Castro, G. O'Shea, A. Rowstron
Zero Servers With Zero Broadcasts
MobiShare'06, September 2006

[3] P. Ynard
Virtual Ring Routing, a Port to GNU/Linux
December 2006

[4] P. Ynard
Broadcasting and Multicasting on VRR
April 2007

[5] B. Fenner, M. Handley, H. Holbrook, I. Kouvelas
Protocol Independent Multicast – Sparse Mode (PIM-SM): Protocol Specification (Revised)
RFC 4601, August 2006