Server Side NFS Identification and Client-Side Packet Tracing in a Virtualized Environment

Rob Jellinek and Adam Vail

May 13, 2013

Abstract

Need to do the abstract

1 intro

We start with NFS: asking the question, how can an NFS server differentiate requests coming from physical machines vs. virtual machines–and in particular, from multiple virtual machines on the same physical machine. Knowing this could allow the server to coalesce packets it's sending if those packets are all going to the same physical machine.

From what we were able to determine, the NFS protocol, being an application-layer protocol, is indistinguishable in the virtual and native settings. Without explicit signals built into the NFS protocol, nothing from the application layer NFS protocol signals the origin and platform of the client.

We learned that the physical server can determine whether requests are coming from bridged VMs by detecting a set of standard MAC addresses assigned by the various virtual machine platforms. The caveat is that the user can always set/spoof their own MAC address. In this case, if you're bridged and have spoofed a legitimate MAC address, the VM's traffic is indistinguishable from traffic coming from a physical device, both in terms of the MAC address, and in terms of timing.

If the client is NATed, then all packet header information in the first three layers of all client-VM outgoing packets get remapped to the physical machine's headers. In this case, there is a dramatic slowdown due to the time it takes the NAT code to map packets from the guest network stack to the host network stack. The remainder of our work focused on determining why and how user-mode networking suffers such a dramatic slowdown compared to bridged networking, and in particular, what code path in the NATed environment causes this slowdown.

2 design

Talk about our system setup here.

3 NFS

We began our quest by searching for a way to differentiate NFS requests from physical and virtual clients, from the standpoint of an NFS server. If a server could distinguish the two clients–and moreover, if it could identify the single physical machine from which multiple VM-accesses originated–then there is the potential to coalesce NFS server responses to multiple VMs on the same physical machine.

We investigated this on three fronts. First, we searched through the NFS 4.0 RFC specification to try to find operations that could potentially differ in the VM or physical context, or otherwise somehow uniquely mark the origin of the client machine. Second, we performed a number of tcpdump traces on NFS 4.0 `mount` operations as well as file transfers using `cat`, with the goal of isolating differences in the action of the NFS protocol in the virtual and physical settings. The tcpdump traces were executed with the flags `tcpdump -vvnnXSs 1536` in order to

examine the full contents of each NFS packet exchanged between the client and server. Finally, we timed NFS reads of varying sizes in the virtual and physical settings in order to determine if timing differences could differentiate physical from virtual clients.

We learned several things from these investigations. First, we were unable to differentiate any aspect of the NFS protocol in the virtual setting from its behavior in the physical setting at the application layer, either in our reading of the NFS 4.0 RFC or in our tcpdump traces. This is not surprising, as there is nothing that should inherently differ about the behavior of the application layer in a virtual machine. Looking lower in the network stack, however, we realized that the MAC addresses assigned to the virtual NICs of bridged connections took on default values under all the virtualization platforms we investigated, which would allow a server to distinguish the virtualization platform of VM-based client applications in these cases. We discuss our findings and the implications in more detail in Section 4 below. Finally, we found that NFS reads from a KVM+QEMU VM using NAT networking and virtio PCI paravirtualized network driver took an average of 72% longer than reads from a native-hardware client, whiel NFS reads from a KVM+QEMU VM using bridged networking with the virtio driver took an average of 26% longer than the client running on the same underlying native-hardware setup. Our experiment and tests are discussed below in Section 5.

4 Identifying Bridged Hosts by MAC Address

While we did not discover characteristics of NFS application-layer requests or responses that indicate whether an NFS client is running on a VM or native hardware, we did find a distinguishing characteristic at layer 2 of virtualized clients' bridged networking stack. In virtual bridged networking, the guest network stack gets its own IP address and MAC address and appears as a separate host on the network, though it uses its host's physical NIC by passing packets through a tap device on the host (see SubSection 5.1 below for details on bridging). The key thing to note is that the main virtualization platforms assign a default MAC address to the virtual NIC, which is then visible to the server in the MAC address fields of the Ethernet headers exchanged with the client. See Table 1 for a list of default MAC addresses assigned to virtual NICs in common virtualization platforms.

This helps us distinguish virtual from native clients to some extent, but still leaves a lot to be desired if we wish to identify and group VM-based NFS clients. First, in the bridged context, it is possible for a client VM to specify their MAC address to be anything they desire. These defaults are merely that: default settings allocated by the virtualization manager or hypervisor if the VM administrator does not specify an alternative address. So there is certainly a chance for false negatives, where virtual machines using bridging on a certain platform cannot be identified by their MAC address because the default MAC address is not used. Second, the identification is overly broad: the NFS server can detect that it is talking with a KVM+QEMU or VMWare guest VM, but it is not possible to identify those guest VMs as belonging to any given physical machine based on this information alone. Finally, the MAC address distinction does not help us in the case of VM-based clients that are using NAT networking instead of bridging, as those guests are invisible to the rest of the network and send their traffic through their host NIC; all VM-guest traffic is rewritten inside the host to contain the MAC and IP addresses of the host NIC

5 Bridged vs. NAT Timing

5.1 Virtual Networking: Bridging vs. NATing

[TODO: Introduce the differences between bridged and NAT networking.] graph with iperf results: show it's a big difference.

RTL8139 (KVM's default)

Virtualization Platform	MAC
KVM+QEMU	52:54:00
VMware ESX 3, Server, Workstation, Player	00-50-56, 00-0C-29, 00-05-69
Microsoft Hyper-V, Virtual Server, Virtual PC	00-03-FF
Parallells Desktop, Workstation, Server, Virtuozzo	00-1C-42
Virtual Iron 4	00-0F-4B
Red Hat Xen	00-16-3E
Oracle VM	00-16-3E
XenSource	00-16-3E
Novell Xen	00-16-3E
Sun xVM VirtualBox	08-00-27

Table 1: First three octets of default MAC addresses by virtualization platform[1]

Possible TODO: Answer: why is Virtio faster than the fully emulated RTL8139

6 Following the Code Path

In order to determine where the bottlenecks are occurring using the RTL8139 driver which performs full emulation, we traced the code path of a packet from the guest's application layer through the guest and host network stacks. We provide an explanation of the traces and relevant functions for the code path for the transmit side below, and an outline of the path for the receive side in the Appendix, along with the code path followed by packets on the transmit side that are shared by both the NAT and bridged code.

6.1 RealTek 8139 NAT: Guest Transmit

User-mode networking under QEMU uses network address translation (NAT) in order to allow a guest VM to establish connections with outside networks. QEMU performs address translation using a protocol from the 1990s known as SLIRP, which was originally used to provide TCP/IP access to dial-up shell account users by emulating the TCP/IP stack. Though that usage is largely obsolete, the same address translation techniques are applied in QEMU user-mode networking

6.2 RealTek 8139 Bridged: Guest Transmit

We first come out the end of the shared code (see Appendix) and get to $\text{tap}_receive.The flow continues as follows$:

$\text{tap}_receive tap_receive_r awtap_write_p acketwritev$

7 Appendix

[TODO: provide a graph of the code that happens whether you're shared or bridged on a guest transmit] The code is implementing a deisgn pattern through the following functions:

GUEST NETWORK STACK finishes here — start QEMU user space emulating physical NIC here — $\text{rtl8139}_transfer_frameqemu_send_packetqemu_send_packet_asyncqe$
$figuredoutthatitneedstobemovingtowardtheTAPdevice.)qemu$
$slirp_inputdependingonwhetheryou'reusingbridgedorNAT.$

This is shared by everyone. It's connecting the driver at the top to the actual networking at the bottom.

——

RECEIVE SIDE: These functions bridge the gap between the top of the host network stack and pass the incoming packets to the bottom of the guest network stack–the RTL8139 emulated network driver.

Bridged Here $\text{tap}_send, etc., don'tmesswiththepacketheadersa$
$-theyjustpassthepacketsstraightuptotheguestfromthephysica$
$-sothepacketshavecomedirectlyfromthephysicalNICanddon't$
$\text{main}_loop_waitqemu_io_handler_polltap_sendtap_read_packetifthesi$

$0 : qemu_send_packet_async ... continue to shared code. The interested reader can check out the KVM source, wazzup.$

NAT Everything is in host byte order, because it's passed through the full host network stack and is coming off the application layer of the host network stack. So at this stage everything gets ripped off and translated back to network byte order and passed to the RTL8139 network driver of the guest stack, which the once again decapsulates the packets for the guest application layer. So the flow is: decapsulate in host network stack; re-encapsulate here and pass to bottom of guest network stack; guest network stack re-decapsulates packet (essentially repeating the work of the host network stack, but with the headers that were modified by NATing).

$\text{main}_loop_wait - slirp_select_poll - tcp_output - slirp_output - ip_output - if_start - if_encap - slirp_output$

This could be why we're getting weird iperf data–because

References