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

Rob Jellinek and Adam Vail

May 15, 2013

Abstract

This paper investigates server-side methods for determining whether NFS client are virtualized or native. We perform client-side and server-side network traces that identify two key differences that can be used as heuristics in identifying virtualized clients. First, virtualized clients using bridged networking can often be detected by the first three octets of their MAC address at the link layer, which are set to unique defaults by most common virtualization platforms. Second, virtualized clients using user-mode networking experience substantial latency in their network stack. Clients can then be distinguished according to bandwidth profiles. Finally, we investigate the sources of user-mode networking latency, and identify memory management in network address translation as the primary contributor to slowdown.

1 Introduction

This paper investigates approaches by which an NFS server can differentiate requests originating from native, non-virtualized machines from those originating from virtual machines. Ultimately, it is desirable for an NFS server to be able to identify and group all virtual machines resident on the same physical machine. Such identification and grouping would allow the server to coalesce its outgoing network traffic to the virtual machines, in essence performing VM-based redundancy elimination. This would lead to better NFS performance, and lower network utilization.

Our approach consisted of three parts. First, we searched through the NFS 4.0 RFC specification [1] to find operations that could potentially differ in the VM or physical context, or otherwise uniquely mark the origin of the client machine. Second, we performed a number of client and server-side tcpdump traces on NFS 4.0 `mount` operations as well as file transfers, with the goal of isolating differences in the action of the NFS protocol in the virtual and physical settings. Finally, we timed NFS reads of varying sizes in the virtual and physical settings in order to determine if timing differences could be used to distinguish 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. Thus, unless explicit signals are incorporated into the NFS protocol for the purpose of identifying a client as virtualized, nothing from the vanilla NFS protocol implementation seems to identify the platform of the client.

However, we were able to establish two heuristics for identifying virtualized NFS clients. First, our network traces showed that although NFS application-layer semantics were indistinguishable, VM-based clients could be identified by a unique MAC address assigned by default to virtual NICs when the VM is using bridged networking. Second, our NFS read timings showed that virtualized clients using user-mode

(NAT) networking experience a substantial drop in bandwidth compared to a native baseline. By profiling client accesses and establishing a baseline bandwidth, it is possible to use bandwidth profiles to identify a virtualized guest in a controlled environment.

The remainder of our work focused on determining why and how user-mode networking suffers such a dramatic slowdown compared to bridged networking. In particular, we trace and identify the key points in the code path of QEMU+KVM user-mode networking that cause this slowdown.

The rest of the paper is organized as follows. We introduce and compare user-mode and bridged networking (Subsection 2.2), which form the basis of our heuristics for differentiating clients by MAC address (Subsection 2.3) and bandwidth profiles (Subsection 2.4). Finally, we end by examining the source of slowdown in user-mode networking in KVM+QEMU (Section 3).

2 NFS

2.1 Experimental Setup

The traces and timing experiments discussed below were run on a simple two-node topology connected by a Cisco E-2000 gigabit router. The client was a Thinkpad T60 with a dual-core 1.83GHz Intel T2400 CPU, 2GB RAM, and an Intel 82573L gigabit NIC. The server was an AMD Opteron 148 operating at 1GHz, with 1GB of RAM and a gigabit on-motherboard NIC. The client and server both used ext4 as their primary filesystem, and the client KVM+QEMU guest VM used ext4 on an 8GB QCOW2 virtual disk. The client was running Ubuntu 12.04LTS natively and in its guest VMs, and the server was running CentOS 5.2.

The tcpdump traces we collected 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.

2.2 User-mode/Bridged Networking

Both of the heuristics we discuss below rely on key differences in network behavior and timing caused by the two main virtual networking configurations used by guest VMs. Guest VMs can be configured to run either user-mode, or bridged networking.

User-mode networking performs network address translation (NAT) behind the host, which isolates guest VMs from the outside network. That is, no entities outside the physical host's network can see the guest VMs, and so without incorporating port-forwarding, those guests cannot be contacted from the outside. Importantly, this also has the effect that all guest VMs share a common IP and MAC address with the host; when the guest has an established TCP session open and the host receives packets destined for that guest, the host demultiplexes those packets and forwards them to the correct guest.

KVM+QEMU uses the SLiRP protocol to perform network address translation. SLiRP was originally used to emulate TCP/IP and provide transport-layer functionality for users with dialup connections. The emulation code has been incorporated into the main branch of KVM+QEMU, and uses this emulation support to provide support for user-mode networking.

Bridged networking, on the other hand, uses a TAP device that is configured on the host. Both the physical NIC and the TAP device must be connected to a virtual network bridge on the host, which requires root access. Guest VMs then attach to the TAP device through their virtual interfaces and participate as full fledged members of the network. This means that their virtual NICs are assigned unique IP addresses that are visible to the outside network, as well as MAC addresses that must be unique within the network.

2.3 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 **??** 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 user 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.

2.4 Identifying NATed Host by Bandwidth Profiles

The MAC address distinction does not help us in the case of VM-based clients that are using user-mode (NAT-based) networking instead of bridging, as NATed guests are not visible to the rest of the network and do not have an externally identifiable IP and MAC address. As discussed above, all guest VM traffic is rewritten inside the host to contain the MAC and IP addresses of the host NIC. However, user-mode networking suffers from a substantial slowdown compared to physical or bridged hosts. In a controlled environment, this dramtically decreased bandwidth relative to a native-client baseline can enable a server to conclude that the client is virtualized.

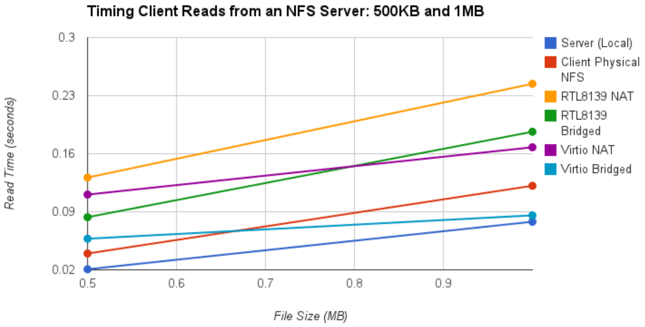

Figure 1: When reading small files, bandwidth profiles are not yet clearly ordered.

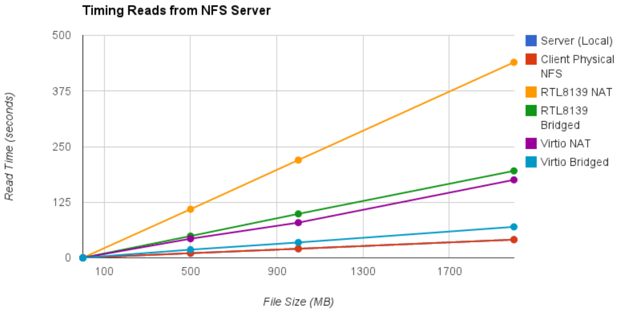

Figure 2: When reading larger files, the bandwidth profiles are clearly differentiated.

Figures 1 and 2 show the results of timing tests in which the client VMs under various networking configurations read 500KB, 1MB, 500MB, 1GB, and 2GB files from an NFS server according to the configuration describe in Subsection 2.1. Files were read in 4KB blocks, and file buffer caches were cleared between tests by running `echo 3 > /proc/sys/vm/drop_caches`. Figure 1 shows that when reading smaller files, the bandwidths of the various client-VM networking configurations have not yet achieved a stable relative ordering. However, when reading larger files, as shown in Figure 2, a clear relative band-

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]

Client Network Stack	**Bandwidth** (MB/sec)
Native	48.3
Virtio Bridged	28.3
Virtio NAT	11.9
RealTek Bridged	10.2
RealTek NAT	4.6

Table 2: NFS Bandwidth Profiles

width ordering emerges.

Not surprisingly, reads on the server itself (directly from the local ext4 file system) were fastest and were mirrored almost exactly by reads from the NFS client running on native hardware. Note that the reads performed locally on the server and from the native client differed by only 43 milliseconds across all tests, so their lines are coincident in Figure 2. Of the virtualized clients, using the paravirtualized virtio driver under bridged networking imposed only a slight overhead relative to the native-client baseline. However, switching to either the fully emulated RealTek RTL8139 virtual network driver or using NATed instead of bridged networking imposed a substantial bandwidth penalty. Doing both (using the RTL8139 under user-mode networking) results in 10.5x slowdown relative to our baseline.

3 Tracing Traffic from the Guest to the Network

The timing tests we ran led us to question the source of the 10.7x slowdown experienced by the RTL8139 fully emulated driver under user-mode networking. We were particularly interested because this is the default network configuration in KVM+QEMU if a user does not specify a configuration explicitly. It is likely the default because the RTL8139 is widely supported, and because users will not be able to use bridged networking if they cannot set up a TAP device on their machine, a process that requires root access. Because of that, it is most likely to be accessible, but as we can see, is also the least performant.

In order to determine where the bottlenecks are occurring using the RTL8139, we traced the code path of a packet from the guest's application layer through the guest network stack and QEMU network emulation code using `callgrind` and `gdb`. We discuss the traces and relevant functions in the code path for the guest transmissions below. We provide an outline of the traces involved in the guest's receive path in the Appendix, along with the code path followed by packets on the transmit side that are shared by both the user-mode and bridged code.

3.1 Usermode Networking

Figure 4b shows a flowchart of functions that a packet passes through when a guest VM sends a packet to the Internet while using user-mode networking. The control flow starts with the RealTek driver passing the packet to the `slirp_input` function. `Slirp_input` calls two functions before the control flow continues, `m_get` which in turn calls `slirp_insque`. In `m_get`, memory is allocated and then passed on to `slirp_insque` to be added a doubly linked list which acts as a packet buffer. The chunk of memory is then returned to `slirp_output` where the packet is then copied into it. It is this allocating, managing, and copying of memory done for every packet in these three functions that causes the majority of the slowdown in usermode networking.

Once the packet has been copied into the buffer space, it is passed "up the stack" to `ip_input`. The function uses the packet's IP headers to sanity check different aspects such as the checksum and length. Then, IP options are processed and if the packet is fragmented, it attempts to reassemble it. If the packet is malformed then it is dropped and the memory used to store it is freed.

If the packet passes the tests in `ip_input` it is passed along to `tcp_input`. `Tcp_input` is a direct implementation of the TCP specification from September, 1981. It does all the standard operations associated with TCP such as calculating the receive window and updating the TCP control block that QEMU maintains. It then strips both the IP and TCP headers from the packet, leaving just the TCP payload, which is passed to `tcp_output`. It is at this point that the packet starts to move back "down the stack" to be put out onto the wire. `Tcp_output` handles TCP timers which determine when a packet should actually be sent. Once a packet is cleared to be sent, the TTL and TOS fields are filled in and the packet is passed along.

With the transport layer complete the packet moves to the `ip_output` where the packet's IP header fields filled in with the host's information. At this point the packet is fully formed and ready to put on the wire. It is handed off to `if_output` which manages two different packet queues, `fastq` and `batchq`. If a packet has the `IPTOS_LOWDELAY` flag set, then it is placed on the `fastq`, which is intended for packets from an interactive application. Otherwise, the packet is placed on the `batchq`. "Each output queue is a doubly linked list of doubly linked lists of mbufs, each list belonging to a separate socket." If a socket is found that does not have a linked list yet, then another is created and the packet is added to this new linked list. To create the linked list, `if_output` calls `m_get` which in turn calls `slirp_insque`.

The packet then moves on to `if_start` which services both the `fastq` and the `batchq`. As the names would indicate, the `fastq` is serviced first. This function passes the packets from the queues on and then cleans up memory upon

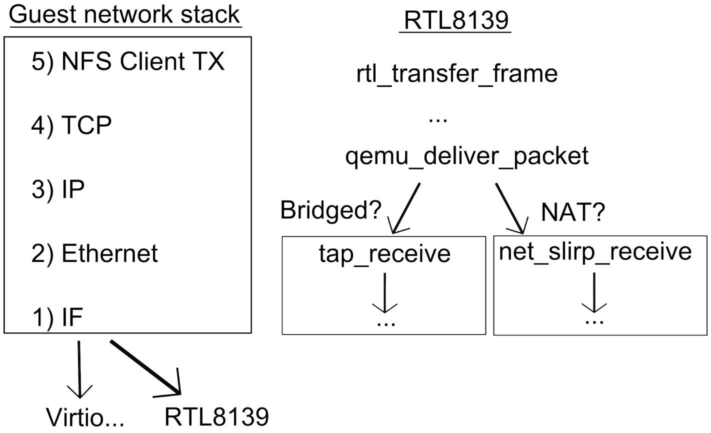

Figure 3: Virtualized client TX code path under bridged and user-mode (NAT) networking using the RTL8139 fully emulated driver. Both start at the top of the guest network stack, but diverge after `qemu_deliver_packet`.

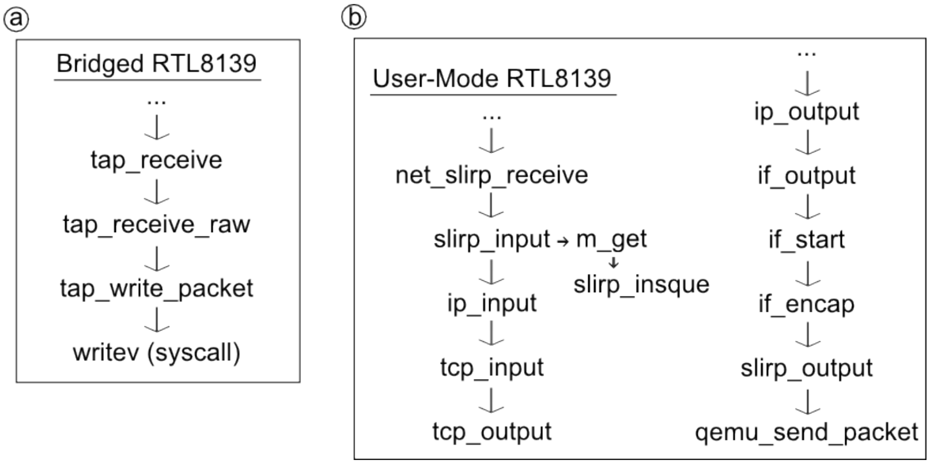

Figure 4: Important functions in the divergent branches of bridged and user-mode networking.

return through the `m_free` and `slirp_remque` functions.

When a packet is serviced it is passed to `if_encap`. Here the packet is prepared for going out on the wire by copying the packet header information into the appropriate header structs. Finally the packet is passed along to `slirp_output`. Essentially `slirp_output` just puts the packet on whatever queue is passed into it along with additional slirp state information. This is done by simply calling `qemu_send_packet`. At this point qemu takes over and the rest of the control path is no longer dictated by the fact that usermode networking is in use.

3.2 RealTek 8139 Bridged: Guest Transmit

Perhaps not surprisingly, the main code path of the RealTek 8139 under bridged networking is substantially more straightforward.

The packet first comes out the end of the shared code (see Appendix) and is sent by `qemu_deliver_packet` to `tap_receive`. `Tap_receive` then performs some header checks and sends the packet to `tap_receive_raw`, which breaks the packet into IO vectors and sends it to `tap_write_packet`. That function is simply a wrapper around the `writev` system call, which writes the packet out to the TAP device.

As we can see, the bridged networking code is much simpler and performs only minimal memory accesses, making it a great deal faster than user-mode networking.

3.3 Code Path Timing

We timed the diverging sections of user-mode and bridged networking transmission paths. We began timing both paths at the function `rtl8139_transfer_frame`, ending the user-mode timing at `slirp_output` and the bridged timing at function `tap_write_packet`. The user-mode path took an average of 1.1 milliseconds, while the bridged path took an average of 44 microseconds, demonstrating a factor of 25 slowdown for user-mode networking relative to bridged.

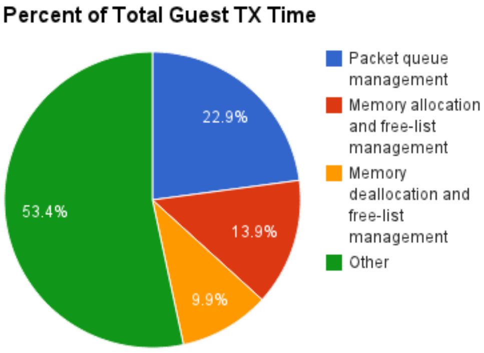

Figure 5: Almost half of the time spent in user-mode networking is spent in memory management.

We found that almost 50% of the overhead in user-mode networking is due to the memory management that takes place in `slirp_input`, `if_output`, and `if_input`. The use of NAT in SLiRP means that a packet has to travel down the guest network stack, and then back up and down the network stack in SLiRP as it gets deencapsulated and reencapsulated with the host's network information, before finally getting sent to the host network stack. The memory management involved in this process—managing doubly linked lists of doubly linked lists of mbufs containing the packet information—necessarily incurs substantial overhead.

4 Conclusions

We began searching for methods by which an NFS server could differentiate native from virtualized clients, and arrived at two conclusions. First, bridged clients can be detected by matching the first three octets of their MAC address against a list of default MAC address prefixes used by virtualization platforms for their bridged guest VMs. If a match is found, this indicates the client is virtualized and is using the corresponding platform. However, false negatives are possible if the client or administrator proactively sets a different MAC address.

Our second method assumes a controlled or predictable network environment and uses bandwidth profiles to distinguish virtual clients against a baseline set by accesses from a native NFS client. It takes advantage of our observation that virtualized clients, especially when using user-mode networking and the fully emulated RTL8139 driver, experience as much as 10.5 drop in bandwidth relative to the native-client baseline.

This led us to investigate the source of the poor performance under user-mode networking, which we determined was primarily the result of the extensive memory management and packet manipulation inherent in the NAT process as the packet travels between guest and host network stacks.

Our initial cross-comparison of virtual network drivers and networking configurations could be expanded in future work to encompass the full range of drivers available under KVM+QEMU. This would contribute to a fuller understanding of the networking landscape under KVM+QEMU, and which options and combinations are the most efficient under varying workloads.

While it is clear that KVM+QEMU users searching for optimal network performance should use virtio under bridged networking, that option may not be available to all users. Future work might also look at providing a better out-of-the-box default option for user-mode networking, perhaps by performing code-path and memory-management optimizations in the NAT process between the guest and host network stacks.

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:

References

[1] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. Network file system (nfs) version 4 protocol, 2003.