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

Adam Vail
University of Wisconsin-Madison

Robert Jellinek
University of Wisconsin-Madison

Abstract

This paper investigates server-side methods for determining whether an NFS client is 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 substantially reduced bandwidth due to processing overhead in their network stack. Clients can then be distinguished according to bandwidth profiles. Finally, we investigate the sources of the reduced user-mode networking bandwidth and identify the memory management overhead inherent in network address translation as the primary contributor to slowdown.

1 Introduction

This paper investigates how an NFS server can differentiate native, non-virtualized NFS clients from clients running on 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 [3] to find operations that could potentially differ between the virtualized and native context, or otherwise uniquely mark the underlying platform of the client machine. Second, we performed a number of client and server-side network traces on NFS 4.0 `mount` operations and file transfers with the goal of isolating differences in the action of the NFS protocol in the virtual and native settings. Finally, we timed NFS reads of varying sizes in the virtual and native settings in order to determine if timing differences could be used to distinguish native 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 native setting at the application layer, either in our reading of the NFS 4.0 RFC or in our network 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. Sec-

ond, our NFS read timings showed that virtualized clients using user-mode (NAT) networking experience a substantial drop in bandwidth compared to a native-client 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 traced and identified the key points in the code path of KVM+QEMU [1] user-mode networking that cause this slowdown.

The rest of the paper is organized as follows. Section 2 describes our network trace collection and timing tests, and the corresponding heuristics for differentiating native and virtualized NFS clients. Subsection 2.2 introduces and compares user-mode and bridged networking, which form the basis of our heuristics for differentiating clients by MAC address (Subsection 2.3) and bandwidth profiles (Subsection 2.4). Finally, in Section 3 we examine the source of slowdown in user-mode networking in KVM+QEMU.

2 Differentiating Native and Virtualized NFS Clients

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 network 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 from the host, 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 [2] to perform network address translation. SLiRP was originally used to emulate TCP/IP and provide transport-layer functionality for users with dial-up connections. The emulation code has been incorporated into the main branch of KVM+QEMU, and has been adapted 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. Creating the TAP device requires root access, and so is not an option for all users. Guest VMs attach to the TAP device through their virtual interfaces and participate as fully 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

Since under virtual bridged networking the guest is assigned its own IP and MAC address and appears as a separate host on the network, it must get its IP and MAC address from some source. Typ-

ically, it is assigned its IP via DHCP or static assignment in a similar manner to the physical host. But its MAC address must first be assigned by some other means. Here, we can take advantage of the behavior of most virtualization platforms, which assign a default MAC address to the guest's virtual NIC unless the user or administrator actively sets the MAC address. The first three octets of each default hypervisor-assigned 48-bit MAC address are fixed according to the virtualization platform, which allows us to use the MAC address to identify the client's underlying platform. This MAC address is 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 under 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 user or administrator does not specify an alternative address. So there is certainly a chance for false negatives, where virtual machines using bridged networking 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, for example, but it is not possible to determine that the guest VM resides on a given physical machine based on this information alone. Finally, this approach works only for guests using bridged networking, since NATed guests use the IP and MAC address of their underlying physical host.

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

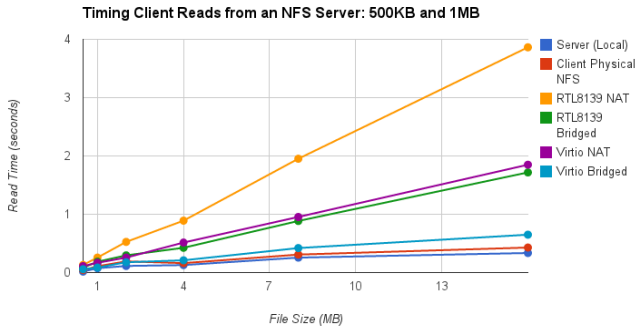

Figure 1: When reading small files, bandwidth profiles are not yet clearly ordered.

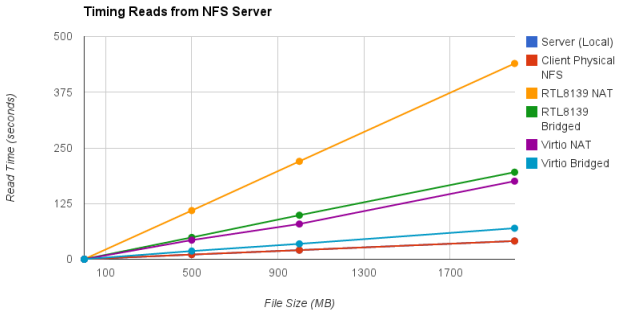

Figure 2: When reading larger files, the bandwidth profiles are clearly differentiated.

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, an NFS server can take advantage of this dramatically decreased bandwidth relative to a native-client baseline to conclude that the client is virtualized.

Figures 1 and 2 show the results of timing tests in which the client VMs under various networking configurations read 500KB, 1MB, 2MB, 4MB, 8MB, 16MB, 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

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 *(Source:http://www.techrepublic.com/blog/networking/mac-address-scorecard-for-common-virtual-machine-platforms/538)*

the various client-VM networking configurations have not yet achieved a stable relative ordering, though it is clear that both NATed clients as well as the RealTek bridged client are differentiated for reads as small as 4MB. When reading larger files, as shown in Figure 2, a clear relative bandwidth 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. 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 10.7x slowdown relative to our baseline.

Table 2 shows the bandwidths observed across the different network configurations of our client KVM+QEMU VM when performing the NFS read-timing tests. Importantly, the bandwidths are highly differentiated. In a controlled environment, this would allow an NFS server to take a "bandwidth fingerprint" of an NFS client by performing a sustained read and recording the bandwidth, and infer from its bandwidth whether the client was running on a native or virtualized platform.

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

2.5 NFS Client Identification Conclusions

Given a lack of signals at the application layer that might differentiate native and virtual NFS clients, the two heuristics described above provide an NFS server with a rough means of separating the two classes of clients. When combined, the heuristics should provide a good indication of a client's underlying platform. If the client is using bridged networking under a default configuration, their bandwidth may be close to the native-client baseline but their MAC address will identify them as virtualized with a high degree of certainty. On the other hand, if the client is using user-mode networking, they will not be identifiable based on their MAC address, but their substantially decreased bandwidth can suggest their use of user-mode networking.

In the next section, we shift gears to try to isolate the origin of the decreased bandwidth of user-mode networking from the client side.

3 Investigating User-mode Networking

The results in Figure 2 led us to investigate why user-mode networking incurs such a large delay over bridged networking. We traced the control path for packets going from a guest VM out to the network as well as from the network to a guest VM under both networking modes. We focus on packets going from the guest to the network since this path incurs the largest overhead in the system. The path from the network to the guest is detailed in the Appendix. We chose the RealTek 8139 driver because it is fully emulated by the system (unlike the paravirtualized virtio driver) and thus incurs the largest amount of delay. The RTL8139 driver is also one of the most widely supported drivers and is the default for user-mode networking when the user does not specify an explicit configuration. While Figure 2 clearly shows a difference in performance between the drivers, the distinction between the different modes of networking was through the use of SLiRP and not specific to the driver.

3.1 Transmission from a Guest to the Network

The control flow of a packet is split into two parts. The packet first travels through the networking stack of the guest VM, and then moves through code in QEMU that is shared by both user-mode and bridged networking. Once QEMU determines which mode of networking is being used, the code paths diverge and the sources of the slowdown become evident.

3.1.1 Shared Control Path

Figure 3 presents the code path originating at a guest VM's application layer. The packet moves down the guest's networking stack to what it believes to be the physical network interface card. At this point the packet is passed to whatever driver has been selected by the user. In KVM+QEMU the options are `rtl8139`, `virtio`, or `e1000`. Once the packet is processed by the driver, it encounters a sizeable amount of shared code (detailed in the Appendix) which is designed to determine which mode of networking is being used and passes the packet off to the correct function. If the guest is bridged then the packet is passed to `tap_receive`, otherwise `net_slirp_receive` is invoked.

3.1.2 Bridged Packet Flow

Figure 4a depicts the code path for bridged networking. The code is relatively straight forward and concise. `Tap_receive` simply verifies the TAP state and then calls `tap_receive_raw`. `Tap_receive_raw` translates the packet into iovectors. `Tap_write_packet` is just a wrapper around the `writev` system call, checking the return value and error code. Finally, `writev` writes the iovectors to the TAP.

3.1.3 User-mode Packet Flow

Figure 4b shows a flowchart of the user-mode networking path. QEMU's SLiRP implementation, as you can see, requires considerably more code to process a packet compared to bridged networking. The control flow starts when the RealTek driver passes the packet to the `net_slirp_receive` function. `Net_slirp_receive` simply passes the packets along to `slirp_input` along with the SLiRP connection state. `Slirp_input` calls two functions before the control flow continues: first to `m_get`, which in turn calls `slirp_insque`. In `m_get`, memory is allocated and then passed on to `slirp_insque` which manages a free list of memory. The chunk of memory is then returned to `slirp_input` where the packet is then copied into the buffer.

Once the packet has been copied, it is passed "up the stack" to `ip_input`. `Ip_input` uses the IP headers to sanity check different aspects of the packet such as the checksum and length. If for some reason the packet has been fragmented, it also attempts to reassemble it. If the packet is found to be malformed then it is dropped and its memory is freed.

If the packet passes the tests in `ip_input` it moves 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 the guest's IP and

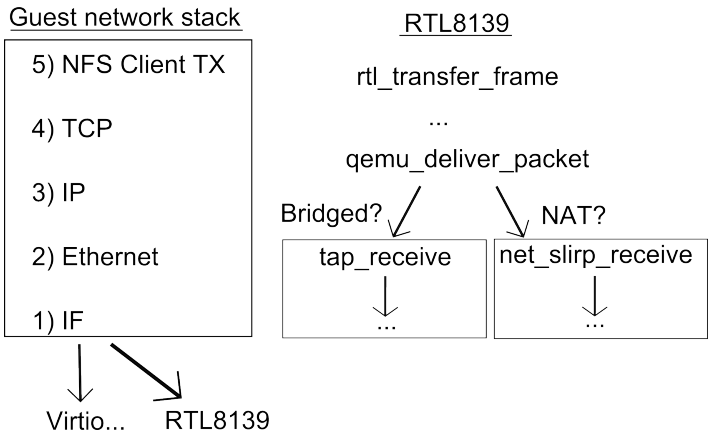

Figure 3: Virtualized client TX code path under bridged and user-mode (NAT) networking using the RTL8139 fully emulated driver. Both start at the top of the guest network stack, but diverge after `qemu_deliver_packet`.

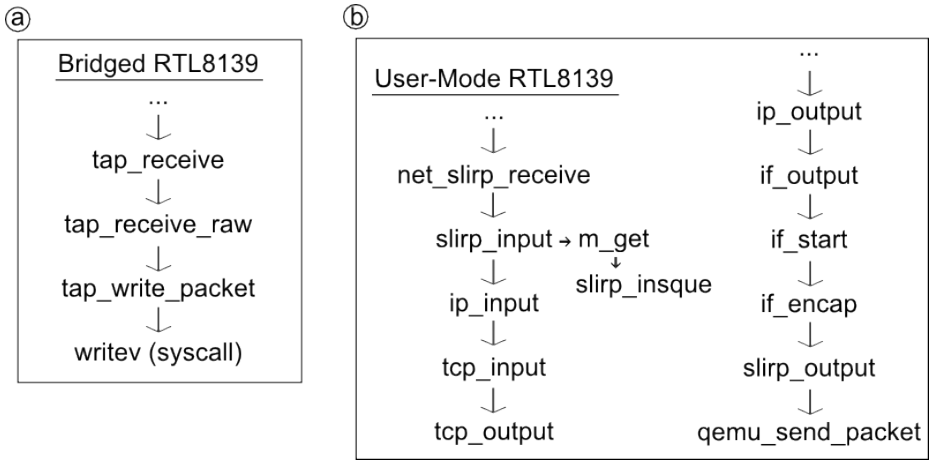

Figure 4: Important functions in the divergent branches of bridged and user-mode networking.

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 to determine when a packet should actually be sent. Once a packet is cleared to send, the TTL and TOS fields are filled in and the packet is passed along. With the transport layer complete the packet moves to `ip_output` where its IP header fields are filled in with the host's information. At this point the packet is fully formed.

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, latency insensitive packets are placed on the `batchq`. Each output queue is a doubly linked list of doubly linked lists of mbufs, each belonging to a separate socket. This is intended to ensure that packets are fairly chosen for transmission across all the sockets of a system. If a socket is found that does not have a linked list yet, then another is created. To create the linked list, `if_output` calls `m_get` which, again, 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. `If_start` also cleans up the memory from each packet once the functions under it return.

Once a packet is chosen by `if_start` 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, `slirp_output` receives the packet which simply gives it to QEMU to be put on the wire through `qemu_send_packet`.

3.2 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 `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.

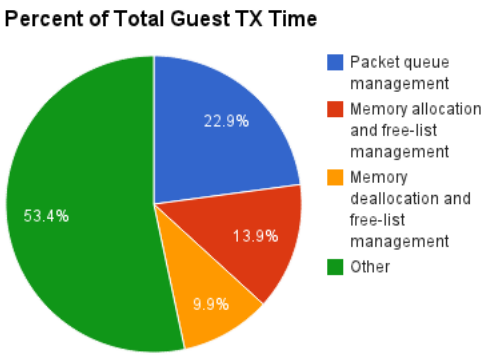

Figure 5: Almost half of the time spent in user-mode networking is spent in memory management.

We then instrumented the SLiRP functions and found that almost 50% of the overhead in user-mode networking is due to the memory management that takes place in `slirp_input`, `m_get`, `slirp_insque`, `if_output`, `if_start`, `m_free`, and `slirp_remque`. The most expensive of these functions was the management of the doubly linked lists of doubly linked lists of mbufs done in `if_output` and `if_start` which accounts for 22.9% of the transmit time. Therefore, the management of memory was seen to be a major contributor to the slow down in user-mode networking. Bridged networking does not have this problem since there is no need to manipulate the packet, which allows it to use the TAP to place the packet directly on the wire.

We also note that user-mode networking is essentially duplicating much of the work already done by the guest's network stack in order to rewrite the correct header fields to make the packet look as though it originated from the host machine. A packet must travel down the guest's networking stack, back up to the transport layer, and then back down before finally getting sent.

4 Future Work

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 show 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 code path between the guest and host network stacks.

5 Conclusions

We began by searching for methods by which an NFS server could differentiate native from virtualized clients, and developed two heuristics for this purpose. 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 10.7x 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.

References

[1] Kvm. http://www.linux-kvm.org. Accessed: 2013-05-10.

[2] KELLY PRICE, D. G. Slirp, the ppp/slip-on-terminal emulator. http://slirp.sourceforge.net/. Accessed: 2013-05-10.

[3] SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M., AND NOVECK, D. Network file system (nfs) version 4 protocol, 2003.

Appendix

Shared Code Path: Transmit Side

Here we provide a brief outline of the code path shared by both bridged and user-mode networking on guest transmissions as a packet travels from the bottom of the guest network stack and moves toward either tap_receive or net_slirp_receive.

```
rtl8139_transfer_frame (end guest
 |                      network stack)
 |
 -> qemu_send_packet
 |
 -> qemu_send_packet_async
 |
 -> qemu_send_packet_async_with_flags
 |
 -> qemu_net_queue_send
 |
 -> qemu_net_queue_deliver
 |
 -> qemu_deliver_packet
 |
 -> net_hub_port_receive
 |
 -> net_hub_receive
 |
 -> qemu_send_packet' (now it's running
 |                     through different
 |                     interfaces: figured
 |                     out that it needs
 |                     to be moving toward
 |                     the TAP device.)
 |
 -> qemu_send_packet_async'
 |
 -> qemu_send_packet_async_with_flags'
 |
 -> qemu_net_queue_deliver'
 |
 -> qemu_deliver_packet'
 |
 -> +-> tap_receive (bridged)
    |
    |
    -> net_slirp_receive (NAT)
       |
       -> slirp_input
```

Note that the second run of each function (indicated as function') performs different tasks than the first run, as they are performing operations using different function pointers. At a high level, this shared code processes packets originating from the guest network stack and forwards them toward the correct intermediate code depending on whether the guest VM is using bridged or NATed networking.

Code Path: 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—in our case, this is to the RTL8139 emulated network driver. The functions encountered at the start of the shared receive-side code below, qemu_send_packet_*, forward the incoming packets to guest network stack based on their incoming

queue.

Bridged

The functions in the bridged receive-side code do not touch the packet headers at all—they just pass the packets straight up to the guest from the physical interface based on their incoming queue. Thus, the packets have come directly from the physical NIC and don't need to have their headers retranslated.

```
main_loop_wait
 |
 -> qemu_io_handler_poll
 |
 -> tap_send
 |
 -> tap_read_packet
 |
 -> qemu_send_packet
 |
 -> qemu_send_packet_async
 |
 -> ... continue to shared code.
```

NAT

In the NAT receive-side code, all the fields in the arriving packets' headers are in host byte order because they have passed through the full host network stack and are 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 once again decapsulates the packets for the guest application layer. So the flow is: decapsulate the packet in host network stack; re-encapsulate it here and pass it to bottom of the guest network stack; the guest network stack again decapsulates the packet (essentially repeating the work of the host network stack, but with the headers that were modified by NAT).

```
main_loop_wait
 |
 -> slirp_select_poll
 |
 -> tcp_output -- strips the headers
 |
-> ip_output -- gets new headers based
|
-> if_output
|
-> if_start
|
-> if_encap
|
-> slirp_output
|
-> qemu_send_packet
|
-> qemu_send_packet_async
 |
-> ... continue to shared code.
```