Transparent Network Services via a Virtual Traffic Layer for Virtual Machines

John R. Lange Peter A. Dinda
Department of Electrical Engineering and Computer Science, Northwestern University
Evanston, IL, USA
jarusl@cs.northwestern.edu, pdinda@northwestern.edu

ABSTRACT

We claim that network services can be transparently added to existing unmodified applications running inside virtual machine environments. Examples of these network services include protocol transformations (e.g. TCP to UDT), network connection persistence during long duration unavailability (e.g. wide area VM migration), and network flow modification (e.g. local acknowledgments and Split-TCP). To demonstrate the utility of this concept, and to enable the practical implementations of these examples and others, we have developed VTL. VTL is a framework for packet modification and creation whose purpose is to modify network traffic to and from a VM, doing so transparently to the VM and its applications. We explain how to use VTL to implement the examples mentioned above and others, such as providing anonymized connectivity for a virtual machine through the Tor anonymizing network, and creating cooperative selective wormholing services for network intrusion detection systems.

Categories and Subject Descriptors: C.2 (Computer-Communication Networks), D.4 (Operating Systems), C.4 (Performance of Systems)

General Terms: Transparent Network Services, Protocols, Virtual Networking

Keywords: Network Services, Virtual Machines, Overlays

1. INTRODUCTION

There has been a fast growing interest in virtualization technologies and their practical uses [8], including in the high performance distributed computing community [7, 18, 33, 17, 9, 38]. Increasingly fast virtual machine monitors [3, 41, 40] and lower overhead device virtualization [24, 27] are making virtual execution environments ever more applicable to high-end computing [15]. Researchers have also been offering prognostic glimpses of new capabilities and uses offered by virtualization environments [11, 10]. Perhaps the most obvious example of this is migration of virtual environments [35, 31, 5, 30, 20], which has finally made process migration [29] widely available.

An important attraction of virtual execution environments is that they permit the transparent addition of new services to applications without requiring application changes. For example, our group has demonstrated the transparent addition of virtual networks [37], application topology inference [13], network monitoring [14], scheduling [26], adaptive migration [38], and optical network reservations [22] to existing, unmodified applications. These services, and all of the others we are aware of, are concerned with observing and controlling the execution of VMs, or of observing and routing their network traffic.

We propose the notion of *transparent network services* for virtual execution environments. A transparent network service can not only monitor traffic, and control its routing, but it can also *manipulate the data and signaling of a flow or connection*. It can statefully manipulate the packet stream entering or exiting a VM at the data link, network, transport, and (to a limited extent) application layers. However, despite this dramatic freedom, a transparent network service must work with existing, unmodified applications. Furthermore, it must not require any changes to the guest OS, its configuration, or other aspects of the VM.

Our motivation in introducing the transparent network service notion comes from two sources. First, in the broader networking community there exists a wide range of networking techniques that could be more broadly used if they could be seamlessly integrated with unmodified applications. If the application runs in a VM, transparent network services can provide this bridge. Second, there are networking challenges that uniquely emerge in the context of VM-based computing, particularly in distributed computing using VMs. Transparent network services can be created that solve these problems. In both cases, new transparent network services could enhance functionality or performance.

We claim that to facilitate the creation of transparent network services in a non-ad hoc way, a common framework is needed. In this paper we introduce the Virtual Traffic Layer (VTL) toolset, which is designed to enable the rapid development of transparent networking services. Using VTL we explore the possibilities offered by virtual networking, implementing a range of transparent network services.

VTL can be used to build standalone services or services that are integrated into VNET, our virtual overlay net-

This work is in part supported by the NSF (awards ANI-0093221, ANI-0301108, and EIA-0224449), the DOE via ORNL, and by gifts from VMware, Dell, and Symantec.

work [37, 38]. Both the standalone and VNET implementations enable a wide range of possibilities for creating new network services and evaluating experimental techniques for harnessing the capabilities of virtual networks. VTL consists of four components, each exporting an API:

- Packet acquisition and serialization
- Packet inspection and modification
- Maintenance of connection state
- Utility functions for common compositions of the previous three

These APIs are used to construct modules that implement services. The modules are then integrated into a run-time core. VTL runs on both Unix (Linux) and Windows. It has been evaluated with both the Xen [3] and VMWare [39] virtual machine monitors, and should work with other virtual machine monitors. It is currently implemented as a userspace tool.

Using VTL we have designed the following transparent network services:

- Tor-VTL: This service bridges the applications running in a VM to the Tor overlay network, resulting in networking being anonymous.
- Subnet Tunneling: This service alters the default routing behavior between two VMs that are on the same physical network, requiring network and data link layer packet manipulation, resulting in enhanced performance.
- Local Acknowledgments: This service generates TCP acknowledgments locally to improve TCP performance on high reliability networks.
- Split-TCP: This service improves TCP performance by splitting a connection into multiple connections.
- Protocol Transformation: This service transforms TCP connections into high-performance protocol connections, such as UDT [12] connections.
- Stateful Firewall: This service is a firewall that is unmodifiable by code in the VM because it exists outside of the VM.
- TCP Keep-Alives: This service maintains TCP connections in a stable, open state despite a long duration network disconnection.
- Vortex: This service provides the wormholing of traffic on the unused ports of volunteer machines back to an intrusion detection system.

These transparent network services and others can augment current services provided to virtual machines. For example, virtual machine migration can now be augmented with network migration, such that the virtual machine is unaware that a network change occurred and retains its open connections. Local acknowledgments can be combined with an optical network reservation to help an application transfer data faster.

Our purpose in describing the above services is to illustrate the flexibility of the VTL toolkit, and the value of the concept of transparent network services.

Experimental Environment. Experiments and measurements reported in this paper were done using an IBM e1350 cluster consisting of dual 2.0 GHz Xeon IBM x335 computers with 1.5 GB of RAM and gigabit NICs connected to an HP 2810-48G gigabit switch. These machines ran Red Hat Enterprise Linux ES 4 update 1 with Xen 3.0 and VMware Server 1.0.1, with the same OS in the VM guests.

2. VTL TOOLSET

VTL is intended to provide developers a simplified framework for enabling the creation of new transparent network services, or the transparent integration of existing network services such that they can be used with a wide range of existing, unmodified applications running in virtual machines.

The core design of VTL consists of a library of packet functions and a suite of modules implementing various mechanisms made possible with the framework. The goal of VTL is to provide an extensible infrastructure to allow for the rapid implementation of new services. This infrastructure includes methods of inspecting packet headers and content, cloning packets, creating new packets, and maintaining per-connection state. Put together these methods provide the components to do packet introspection and manipulation, and to create virtual connection endpoints.

VTL is designed to provide packet access and modification capabilities from the transport layer down to the data link layer. As such VTL does not provide direct functionality to handle application-specific protocols or packet formats. However VTL is extensible and can be augmented with application-specific functionality. For example, we will later discuss a DNS extension.

VTL can be used alone, or as an extension to VNET. In the latter case, VNET is in a position to optimize, on the fly, live network traffic. We are currently studying how to integrate the VTL mechanisms into our adaptation and inference mechanisms to provide automatic instantiation of traffic optimization.

2.1 Requirements

VTL requires that the virtual machines be configured with "host-only" network interfaces to ensure that VTL is the sole recipient of all network traffic. In this way VTL can inspect and modify all traffic seen by the virtual machine without any modifications to the guest OS or applications. VTL must be able to efficiently attach to the virtual network interfaces provided by the VMM, to be able to send and receive packets.

VTL is built on top of the libpcap [28] libraries and libnet [25] (if VTL is running under a Unix variant). Libpcap provides packet capture functionality and is available for all standard Unix environments and as Winpcap [42] for Windows systems. VTL utilizes libnet for packet injection in Unix environments, and uses the winpcap injection mechanisms under Windows. Currently VTL is able to interface with any virtual machine monitor that provides a virtual networking interface accessible to pcap and libnet. VTL can also connect to other applications as long as the network traffic can be routed through TUN/TAP devices. This includes both Xen and Linux.

2.2 VTL and VNET

We now expand our description of VTL's relationship with VNET.

Standalone VTL. VTL is capable of standalone operation, and includes all the necessary mechanisms for integrating itself with a virtual environment. The main functionality of VTL is included as a library that is used by standalone utilities to connect to virtual networks and manipulate the packet streams to/from VMs.

VNET. VNET is a layer 2 overlay network for virtual machines. VNET manages the illusion that a distributed collection of virtual machines each maintain a network presence on a given local area network, while simultaneously striving to provide high performance connectivity among them. Each physical machine hosting a VM runs a VNET process that intercepts VM traffic and tunnels it to the appropriate destination. The destination is either another VM that can be contacted directly through VNET or an address external to the overlay. Traffic destined for an external address is routed through the overlay to a VNET proxy node, which is responsible for injecting the packets onto the appropriate network. The overlay thus consists of a set of TCP connections or UDP peers (VNET links) and a set of rules (VNET routes) to control routing on the overlay.

VTL with VNET. While VTL is designed to operate independently, it can also be used as an extension to VNET. In the VNET environment, VTL is capable of either monitoring/transforming traffic as it passes through VNET, controlling its own set of the overlay links that VNET routes onto, or handing routing decisions for a subset of VNET traffic. This enables far greater flexibility in managing VNET traffic, and provides VNET an extensible framework for incorporating new functionality.

VTL runs as as a companion process connected to VNET with a fast full duplex channel. This allows packets to be handed over to VTL with the same mechanism used for transmitting packets between VNET nodes. In this way VTL is exposed as an alternative type of VNET link that can be instantiated and routed to using the same control methods as ordinary VNET links. The VTL modules are reconfigured to read and write packets via the channel instead of a network interface. This allows all VTL modules (services) implemented as standalone tools to be seamlessly integrated into the VNET overlay.

2.3 Network Interface API

VTL provides a common API to capture and inject packets to and from a network device, virtual or real. While this API is currently built on top of libpcap and libnet, it exists to simplify packet capture and injection and to provide an interface to other packet capture mechanisms. The API includes functions to connect and disconnect from a device, and functions to read and write packets to the device. There is also a notification access function that returns a environment-specific handle that will indicate when a packet is available to be captured.

As an example, VTL allows us to easily bridge a virtual host-only network to a physical network device. We illustrate this in Figure 1, which implements a VTL module that simply writes packets received on one device out to another.

VTL Overhead. The current VTL implementation is completely in user space, which places limits on the possible performance available to a VTL module. In Figure 2 we show the maximum bandwidth possible using ttcp when bridging a virtual network interface to a physical interface. We present results for three different types of bridges: the standard Xen bridge configuration (all in kernel) and two different VTL modules (both in user space). The first VTL module runs as a single process for half duplex operation, while the second consists of two processes, together forming a full duplex connection.

```
RawEthernetPacket pkt;

iface_t * src_if = if_connect("src_device");
iface_t * dst_if = if_connect("dst_device");

while (if_read(src_if, &pkt)) {
    if_write(dst_if, &pkt);
}
```

Figure 1: Simple one-way VTL bridge.

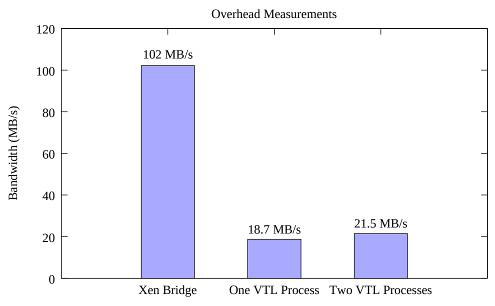

Figure 2: Performance overhead of VTL.

While the results show a significant degree of performance degradation, it is important to make several points. First, VTL is intended as a proof of concept tool. We have not yet explored how to optimize VTL, but there is no good reason to believe that performance would be degraded if it were integrated into the Dom0 kernel of Xen, or equivalent in VMware. Second, the available performance is still quite reasonable for WAN environments.

2.4 Packet Access API

Most of VTL's functionality is encapsulated in an API allowing inspection and modification of packets. VTL provides primitives that allow direct access to standard packet fields which can be used directly, or accessed through various functions that have been built on top of them. Examples for accessing the destination address in the IP header are illustrated in Figure 3.

Besides offering a unified packet access API, VTL builds on these primitives to provide more advanced functionality. Such functions include packet queries, field swapping functions, header calculations, and derivative packet creation (e.g., ACK generation).

2.5 State Models

The VTL framework also provides methods for maintaining connection protocol state, in the form of state models. VTL modules can manipulate state corresponding to either end point of a connection, since the state models contain state variables for both end points. This allows a VTL mod-

```
RawEthernetPacket pkt;
unsigned long dst, new_dst;

dst = *(uint32 *)IP_DST(pkt.data);
*(uint32 *)IP_DST(pkt.data) = new_dst;

dst = GET_IP_DST(&pkt);
SET_IP_DST(&pkt, new_dst);
```

Figure 3: Example of basic packet access.

```
int create_data_pkt(vtl_model_t * model,
                    char * data,
                    int data_len) {

  RawEthernetPacket data_pkt;

  create_empty_pkt(&model,
                   &data_pkt,
                   OUTBOUND_PKT);

  memcpy(TCP_DATA(data_pkt.data),
         data, data_len);

  ip_len = GET_IP_TOTAL_LEN(data_pkt.data);
  ip_len += data_len;
  SET_IP_TOTAL_LEN(data_pkt.data, ip_len);

  compute_ip_checksum(&data_pkt);
  compute_tcp_checksum(&data_pkt);

  sync_model(&model, &data_pkt);

  pkt_len = data_pkt.get_size() + data_len;
  data_pkt.set_size(pkt_len);

  queue_pkt(&data_pkt);
  return 0;
}
```

Figure 4: Creating a data packet.

ule, for example, to create packets that appear to originate from either the local or remote network end point. Included with the framework are basic models for standard protocols that can be stacked to provide support for additional higher level protocols. For instance, a TCP state model consists of the various state parameters needed for a TCP connection (ACK/SEQ numbers, timestamps, etc) as well as an IP state model that contains the necessary information needed by the IP protocol (IP_ID). This allows VTL to be extended to track state at multiple layers of the networking stack with the same interface.

The state inside a state model is maintained by the VTL state API. This API allows initialization of state models, synchronization with connection traffic, and connection interaction. State models can either be initialized manually in the case where the developer wishes to create a connection inside VTL, or by supplying an example packet from the connection. Once initialization is complete the model can be modified directly by the user, or by updating the model with a connection packet. The model can also recognize packets that belong to the tracked connection, as well as transform packets so that they belong to the tracked connection. Furthermore, VTL can create new packets belonging to the connection from the VTL models. Later we discuss how we used this feature to tunnel TCP connections through a SOCKS proxy server. A simple example of packet creation is shown in Figure 4.

3. EXAMPLE SERVICES

We now describe how a range of transparent network services are implemented using VTL.

3.1 Anonymous Networking For Any Application

The Tor network is a cooperative overlay that allows TCP connections to be routed such that the source of the connection remains unknown to the destination or anyone observing the traffic. The Tor network is based on the Onion Router [6], which provides randomized and encrypted routing through an application-layer overlay.

Applications route connections through Tor by being configured to send their traffic through a standard SOCKS [19] interface. Currently Tor is only available for applications that have implemented the SOCKS interface for proxying connections or are running in an environment configured to use a myriad set of tools that provide SOCKS proxy support for applications not implementing it themselves. An extreme example of this is the Anonym.OS [1] live CD, which provides a pre-configured environment that only runs applications that can be interfaced with Tor.

Using VTL, we have developed a transparent network service that interfaces any application running in the VM with the Tor network. No changes to the application or its VM environment are needed. This service illustrates the utility of VTL.[1] Beyond basic Tor connectivity, the combination of the VM and Tor-VTL results in us being able to completely prevent any information leakage since all traffic not routable by Tor is simply dropped by the host-only network. Furthermore, Tor-VTL will work with any OS or application running in the VM as long as it uses TCP. A typical Tor-VTL configuration is shown in Figure 5.

VTL Implementation. The Tor-VTL module impersonates the destination of any and all network packets sent from the VM. This is done by using the VTL state models to maintain per connection state for each connection opened by the guest environment. The data segments for each connection are extracted and proxied through a SOCKS connection to the Tor network. The SOCKS proxy connection is maintained in states analogous to the states of a TCP connection. The management of the connections' state is done in four stages:

- Open – Establish SOCKS connection, initialize state model, handle SYN sequence
- Established – Maintain state models, handle data transfer
- Close – Close SOCKS connection, handle FIN sequence, delete state model
- Error – SOCKS connection closed, VM signaled via RST packet

[1]Tor-VTL runs under Linux and Windows and is available for download at `http://www.artifex.org/~jarusl/TorVTL`

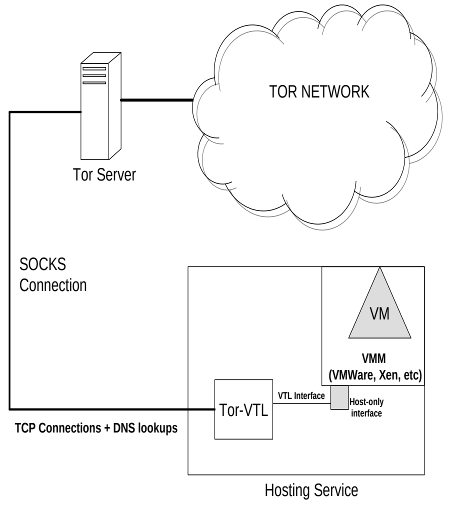

Figure 5: Network configuration of Tor-VTL.

The implementation of the Tor-VTL module is accomplished in ∼1000 lines of C code. It is the most complex module (service) that we illustrate in this paper. We now discuss three of these stages along with the special cases for DNS lookups and ARP requests.

Open. The open mechanism is triggered by the reception of a SYN packet from the virtual machine. When this occurs Tor-VTL extracts the destination IP address and port number from the packet and sends a SOCKS connection request to the Tor proxy server. Tor-VTL then initializes a state model for the connection and adds it to a list of current connections, along with a reference to the socket connected to the proxy. The connection request latency of Tor network can be significant, so at this point Tor-VTL returns to process other pending events. The connection establishment latency is frequently longer than the timeout value for TCP SYN packets, which results in multiple SYN packet retransmissions, which Tor-VTL simply drops. Once the Tor server has established the connection it sends a reply message back to Tor-VTL signifying success or failure. The module then handles the reply, generates a SYN-ACK packet, updates the state model, and queues the packet for delivery to the VM. At this point the module is ready to transfer data.

Established. Once the connection has been established, an open socket to the Tor proxy exists along with a state model describing the TCP connection from the virtual machine. At this point Tor-VTL can handle data transmissions from either Tor or the VM. Because of the stateful nature of TCP connections Tor-VTL must track various values included in the TCP packets. These include the current sequence and acknowledgment numbers as well as the window size and optional timestamp values. This is all handled by the VTL state models, and only requires that the module keep the model synchronized via calls to a synchronization function.

When the VM sends data, Tor-VTL captures the packets and searches the state models for the connection. Once the connection is located Tor-VTL extracts the data and writes it onto the open Tor proxy socket. Tor-VTL then updates the state models with the values included in the intercepted data packet. Finally the module generates an empty ACK packet, again updates the state model, and queues the ACK packet for delivery to the VM.

When data is received from the destination host through the SOCKS connection, Tor-VTL must also ensure that it is delivered to the virtual machine. An incoming data transfer is read from the Tor proxy socket in increments the size of the current MSS of the TCP connection, available via the state model. These segments are then used to generate TCP data packets that are transmitted to the VM. As the packets are sent the module continuously updates the state model to maintain connection consistency.

Close. Similar to data transfers, a connection close request can come from either the VM or the SOCKS proxy. Depending on the source Tor-VTL makes sure that the proper sequence of actions occur to close the connection gracefully for both the VM and the Tor proxy.

When the VM closes a connection it transmits a FIN packet that is captured by Tor-VTL. The module then closes the correct connection to the Tor proxy, and generates a FIN-ACK packet that is sent to the VM. The connection state is then marked as CLOSED and the connection deleted. The final ACK from the VM is dropped by the module.

The Tor server handles a remote close by closing the corresponding proxy socket. This is detected by Tor-VTL which also does a graceful close on the socket. The module then generates a FIN packet and queues it for delivery. The connection state is marked as FIN_WAIT1, and processing continues. When the subsequent FIN-ACK packet is received from the VM, the module generates a final ACK packet, queues it for delivery, and deletes the connection state.

DNS. Standard DNS lookups to well known DNS servers can possibly lead to a leakage of information, even if the subsequent TCP connection is routed through Tor. To prevent this Tor has implemented the SOCKS4a protocol extension that provides a method of doing DNS name lookups through the Tor network. Tor-VTL also includes a special case handler for DNS packets to allow a virtual machine to do standard DNS lookups.

DNS lookups are handled by a VTL DNS extension that provides DNS packet processing functions to the VTL framework. The Tor-VTL module includes these extensions along with a mechanism for tracking pending requests. DNS packets are a special case of UDP packets that are detectable by the destination port of the UDP packet. When Tor-VTL receives a DNS packet it uses the VTL DNS extensions to extract the lookup requests into a list of request data structures. Because the SOCKS4a extension is only capable of handling one hostname lookup per request, DNS packets containing multiple requests are broken into multiple requests. For each request Tor-VTL opens a connection to the Tor proxy and transmits a lookup request in the SOCKS4a format. The request is then stored in a request list and the module returns to process pending events. Once the lookup has completed, the result is received on one of the lookup sockets and matched to a pending request. The module then uses the DNS extensions to generate a DNS response packet that is delivered to the VM. Currently, Tor only supports forward lookups, and so Tor-VTL simply drops DNS packets containing reverse lookups.

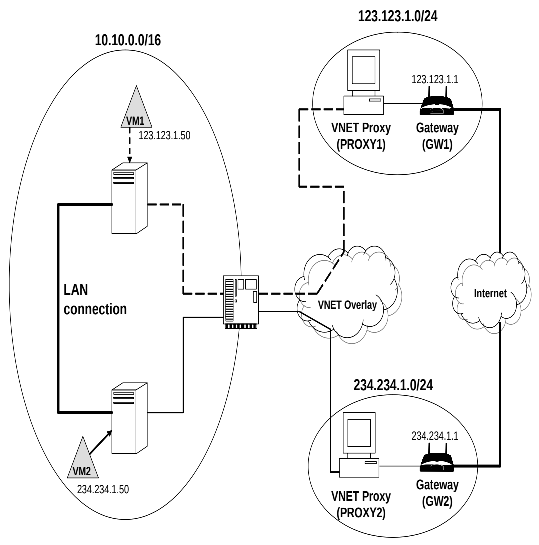

Figure 6: Subnet tunneling example.

ARP. Tor-VTL is required to handle ARP packets in order to virtualize the endpoint to the virtual machine. When the VM initiates a connection, it first transmits an ARP request for either the destination or gateway IP addresses depending on its routing table. When Tor-VTL receives an ARP request packet it automatically generates an ARP reply with a fake but valid MAC address, and sends that reply back to the VM. This allows the virtual machine to begin transmitting IP packets.

3.2 Subnet Tunneling

In our experience with working with our VNET overlay we encountered situations similar to that shown in Figure 6. Here two VMs are hosted on a common local area network, but are configured to use two separate VNET proxies located on two different remote LANs. VNET is architected to present the illusion to a VM that its network presence is actually physically located on the same LAN as the proxy server it is configured to use, and as such it is assigned an IP address from the same subnet as the proxy.[2] Thus whenever a VM seeks to communicate with an address on a different subnet than that of its proxy, data flows through the proxy. The result can be a circuitous routing of traffic between machines that are physically located very near each other.

In the example of Figure 6, traffic from VM1 to VM2 is first routed through VNET to PROXY1 where it is injected onto the physical network. The destination MAC address of these packets is the MAC address of the gateway GW1. The packet is then routed through the Internet, based on its destination IP address, so that it eventually arrives at the gateway GW2. The VNET proxy located on that LAN (PROXY2) then captures that packet off the physical network and routes it through VNET to VM2. In this case the packet is routed through the Internet at large three times (twice through the VNET overlay, once between the physical gateways), just to arrive at a VM that is physically connected to the same gigabit switch as the sender.

[2]In the example, the proxies might correspond to users who desire that their VM's network presence is on their own LAN, but who also want fast communication between them.

VNET prevents this situation when the VMs are configured to use the same proxy.[3] The destination MAC address is set to the destination VM instead of the gateway, and VNET establishes an overlay link directly between the VMs. It may appear that we could do precisely the same thing in the situation of Figure 6, but the routing tables inside of the source VM will get in the way. The packet will be set to the MAC address of the gateway. VNET could send the packet directly to the destination VM, but that VM would drop it because its MAC address wouldn't match.

The subnet tunneling service eliminates this problem by rewriting the packets sent over the direct overlay link.

VTL implementation. VTL is used to allow VNET to measure traffic between IP addresses in addition to MAC layer addresses. This allows an adaptation mechanism to recognize when two VMs configured to use two different subnets are communicating. This case can be distinguished from the normal cases because the MAC addresses will not match the destination specified by the IP address. Once this mismatch is detected, VNET can use VTL to rewrite the Ethernet headers for packets between the two hosts.

In the new scenario, traffic from VM1 to VM2 is handled inside the VMs exactly as it was before. VM1 transmits a packet to VM2 with the MAC address of the gateway GW1. However, VNET has detected the communication between VM1 and VM2 previously and has already created a direct overlay link that only traverses the local gigabit switch. The packet is first handed to VTL, which rewrites the destination MAC address with the MAC address of VM2, and is then sent on the direct link to VM2. In this way we can transparently allow two virtual machines configured to use two different subnets to communicate directly as if they were on the same LAN, without requiring modifications to the routing tables inside the guest environment.

3.3 Enhancing Network Performance

The idea of improving the performance of TCP by inserting a stateful proxy between the source and destination is well known. RFC 3135 [4] contains a discussion of many of the methods that have been employed in the past. We now show the utility of using VTL to implement several of these methods. We concentrate on improving TCP performance in networks with large bandwidth-delay products, a common problem in high performance distributed computing.

The TCP flow control and congestion control algorithms are well known to perform poorly in high bandwidth-delay product (high BDP) networks [21]. The problems arise predominantly from the slow start mechanism as well as the maximum limit of the TCP window size. Application mistakes, such as small socket buffers, are also common.

Local Acknowledgments. One technique that is used to improve performance of TCP in high BDP paths that are reliable is the creation of local ACK packets at the source or at a location near the source. A local ACK is an empty TCP packet that simply ACKs the last data packet sent by the source endpoint. This avoids the problems imposed by the high BDP by fooling the source host into thinking that the destination is located nearby and latency is minimal.

[3]For example, when they all belong to one user or VO or can put on a common IP subnet.

This addresses the problems with TCP window constraints and socket buffer sizes in that the TCP stack in the virtual machine never has to wait for an acknowledgment before sending the next packet. Essentially it takes the TCP window out of the picture, and transforms TCP into a protocol that provides a relatively steady stream of packets, clocked out at the rate of the local acknowledgment. For network architectures that have their own reliable delivery guarantees this method solves the performance problems of applications developed for TCP.

Local acknowledgments are trivially implemented in VTL. We limit our discussion of them here as they are also a part of protocol transformation, discussed below.

Split-TCP. On networks that do not provide delivery guarantees, we can still use VTL to provide the advantages of local acknowledgments. One of the more well known methods of achieving this is Split-TCP [2], which transforms TCP connections into multi-hop instead of end-to-end connections. This method is most advantageous in networks with differing link characteristics. VTL can be used to straightforwardly integrate a VM into a Split-TCP environment, or to implement Split-TCP techniques themselves.

Combining VTL with VNET allows dynamic instantiation of Split-TCP connections. By using Wren [43, 14] or other network monitoring software, VNET can determine the physical network characteristics underlying the overlay. Using this information VNET can then use VTL to instantiate a Split-TCP node where it would be advantageous.

Protocol Transformation. Related to Split-TCP is the VTL functionality that provides protocol transformations. Here we seek to improve TCP performance by transforming it into another transport protocol. Existing applications and OSes can thus be retrofitted with the newest transport protocols without programming.

The methodology for choosing protocol transformation is similar to that of Split-TCP. When VNET detects network characteristics that indicate performance would be improved by using a different protocol, a VTL module is started at that location. In this case the VTL module would intercept the packets belonging to the target protocol (typically TCP) and transform them into a new protocol. At the end of the path another VTL module transforms the packet back into the target protocol for delivery to the end host. To evaluate this technique we again looked at high BDP networks and several experimental protocols that have been developed for them. We discuss UDT, one such protocol next.

VTL-based TCP⇔UDT⇔TCP Transformation. UDT (UDP-based Data Transfer) [12] is a high-performance protocol developed for networks operating with a high BDP. UDT is a reliable delivery transport protocol that provides the same assurances as TCP but using congestion and flow control algorithms that are optimized for network paths with high latency. We use UDT to demonstrate the capability of VTL to improve network performance for TCP applications running inside a virtual machine.

We created a VTL module to transform TCP flows into UDT flows. The design of this module is essentially the same as the TOR/SOCKS module we described earlier (Section 3.1) with a few notable exceptions. The code to manage the TCP state and interact with the virtual machine were mostly untouched. The SOCKS interface functionality was replaced with UDT-specific functions. Also instead of forwarding traffic to a central proxy server, UDT connections are created based on the destination address and port extracted from TCP-SYN packets.

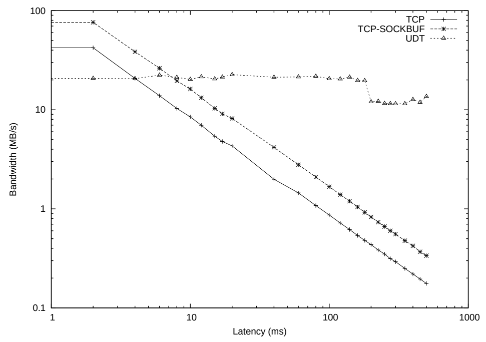

Figure 7: Bandwidth of TCP, TCP with large socket buffers, and TCP transformed to UDT. All as a function of path latency.

To evaluate the performance of the UDT transformation module we ran a ttcp benchmark between a Xen VM and a second node, while varying the path latency. The latency was emulated using the netem extension of iproute2 included with the Linux kernel. We used VTL to intercept all traffic from the source VM, send it through our transformation, and deliver it to a server running on the destination host. For comparison we also ran ttcp benchmarks without the Xen kernel, both with the default parameters, and with the socket buffer configured to 1.5 times the BDP of our path.[4]

Figure 7 shows the results. As the path latency increases the performance of TCP declines. The increased socket buffer size does improve performance, but doesn't stop the decline. In contrast, TCP transformed to UDT offers consistent and stable performance on the high latency path. The performance dip of UDT was due to the fact that the module scales the ACK rate based on current conditions. The dip corresponds to the point at which one ACK is generated for every two packets instead of for every three packets.

In the future we intend to integrate this UDT transformation module into our VNET overlay architecture. VNET already includes application and network bandwidth measurement mechanisms, thus VNET could dynamically create UDT connections between virtual machines when the environment is appropriate and the application demands warrant it.

3.4 Transparent Security Layer

Almost all recent operating systems include embedded firewalls that for the most part share the same functionality. Packets are compared to user supplied firewall rules, usually based on address and port signatures, to determine the appropriate action to take (e.g. allowing or dropping). They are also all implemented inside the host's operating system. While these firewalls do help to curb some of the more simplistic security threats to systems, they all have

[4]We did try these benchmarks within Xen as well, but a fatal interaction of Xen and netem led to very low performance. The present point of comparison shows TCP and TCP with large socket buffers in the most positive light.

the same drawbacks. Since the firewall is implemented in the actual system nothing prevents the user or a system level process from altering the rules or even the firewall implementation itself. If an externally available system level service is compromised the firewall itself is compromised, since the attacker will be able to access and modify the firewall rules directly.

With VTL we can implement a firewall that has the same functionality, but that can be implemented and used without modifying the OS or applications, and that cannot be compromised by the OS or any other system level process. Like the other examples the firewall is a VTL module that vets every packet into and out of the VM. Since the operating system and any rogue process are contained inside the VM there is no way for them to access or modify the firewall.

3.5 Enabling Connection Persistence During Long Duration Migration and Hibernation

Lately migration has been espoused as on of the most promising advantages of virtual machine environments. Both the VMWare and Xen developers have spent considerable effort in improving migration capabilities for their products, leading to live migration capabilities on LANs [5, 30]. Migration over wider areas, where bandwidths are lower and more state (e.g., disk) needs to be transfered, results in longer migration times during which the VM does not run [35, 31, 5, 20]. While strategies for migrating the execution environment have been developed they have yet to deal with the issue of migrating open network connections over the wide area. Examination of migrating network connections reveals two issues that must be addressed:

1. Routing: Ensuring that the packets are routed to the new physical location correctly.
2. Timeouts: Preventing connections from closing due to timeout events during long periods of disconnection due to migration or hibernation.

The routing issue can be addressed with the use of VNET as discussed in earlier work [38]. Preventing timeouts is a problem that we have addressed in VTL framework.

Network timeouts can arise from multiple sources. From the system level, TCP connections maintain a timeout timer that closes the connection after a length of time in which no packets have arrived from the remote host. This can arise in one of two situations. Either the connection is inactive and not sending data for prolonged periods of time or the amount of data is large enough to overflow the host's receive buffer, in which case additional data packets are dropped by the end host. To prevent these situations from causing a timeout on the sender, TCP will transmit keep-alive packets. These packets signal the sender that the remote host is up, but not ready to accept data. The second type of timeout that can occur is an application timeout. These timeouts are more difficult if not impossible to address and require an understanding of the applications behavior. By using VTL we are able to address the issue of TCP timeouts, but not application timeouts.

VTL Implementation. We have developed a VTL module that allows us to indefinitely suspend a VM while maintaining its open connections for the length of time that the VM is not running. The module tracks the connection state of all the open connections of the VM. Whenever the VM is suspended (for example, when migration begins), a signal is sent to the module to start listening for incoming packets. The VTL module then answers every received packet with an acknowledgment showing a window size of zero. The remote hosts stop sending data and periodically send probe packets. These probes are received by the VTL module and acked to ensure that the connections remain active. When the VM is resumed at its new location a signal is sent to the VTL module to stop processing packets for that VM.

During migration, this maintenance of connection state works in conjunction with VNET. VNET guarantees that addresses don't change regardless of the location of the VM, and can also route packets anywhere on the overlay. This means that when a VM migrates VNET reconfigures itself to route that VM's packets to the VTL module instead. When the VM is resumed VNET is configured to now deliver packets to the VM before disabling the VTL module. This ensures that only the VM or VTL are responsible for handling traffic for the VM at any given time, as well as ensuring that there is always perceived connectivity to that virtual machine.

3.6 Cooperative Selective Wormholing

Collecting and analyzing network traffic to detect new methods of attack has long been recognized as a necessity by the security community, and numerous systems have been developed to provide such a service. Nearly all of them operate by aggregating network traffic from some source. We have developed a new method of traffic aggregation called *Cooperative Selective Wormholing* (CSW) that relies on volunteers contributing their hosts' unused network ports and a portion of their bandwidth [23]. CSW wormholes capture traffic destined for those ports and tunnel it to generic backend systems stood up by researchers and others. The attacker is unaware that he is interacting with a backend instead of with the volunteer's machine. The volunteer sees limited (and controlled) performance impact, and a CSW wormhole will immediately get out of the way if the port is otherwise used.

VTL Implementation. Using VTL we created Vortex, a prototype implementation of CSW. Because VTL is cross platform, the Vortex volunteer software (the client) can run on both Unix/Linux and Windows. The Vortex implementation uses many VTL features to maintain invisibility to the attacker, restrict access to the volunteer PC and the LAN it is on, and control the use of resources. As shown in Figure 8, the Vortex client runs side by side with the network stack on the volunteer machine, and delivers Ethernet packets (rewritten for anonymity) to a VNET overlay, which coveys them to almost any kind of backend system. The Vortex client consists of only 800 lines of code.

Vortex helps to demonstrate the generality of VTL, providing an example of a VTL-based service that operates directly on a commodity PC without VMs.

4. RELATED WORK

There are many toolsets that provide specific examples of packet or network flow modification to achieve improved performance or additional functionality. However, we believe that our work is novel in that we provide a general framework that targets virtual environments. The previous

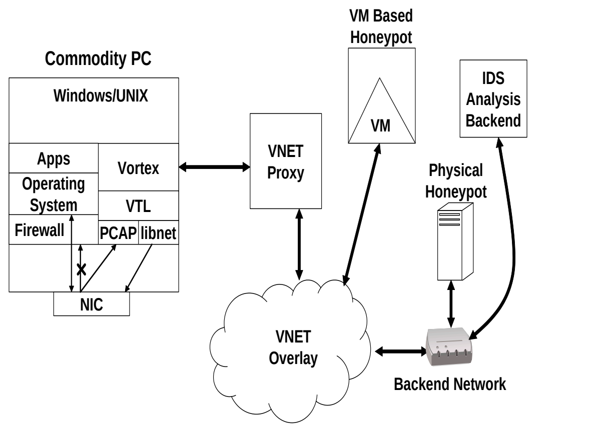

Figure 8: Vortex architecture. Vortex uses VTL to capture Ethernet packets before they are dropped by the host firewall. The packets are conveyed via VNET to just about any kind of IDS backend.

work typically targets a specific case, and often requires reconfiguration of the host environment or modification of an application. VTL allows simplified implementations of many approaches while allowing them to operate in a manner that is transparent to the application.

The TESLA architecture [34] provides capabilities very similar to VTL, but does so at the session layer. By using a library interposition technique TESLA is able to intercept network-related system calls and add transparent functionality to an application. TESLA and VTL share many motivations and capabilities, however in many situations there is a significant advantage in operating at the network level. For example, our techniques can work regardless of the OS in the VM.

There are also tools that use VMs to add additional functionality to the network. Of note here is JanusVM [16], which sends all of the traffic from a VM through the Tor Network. While this appears to accomplish the same function as our Tor-VTL tool, there are differences. First, JanusVM operates inside the guest by establishing a VPN that uses a proxying tool to handle all traffic. This requires the use of a specially configured virtual machine, and further configuration of the host environment. Tor-VTL is a self-contained executable that requires no special system configuration.

5. CONCLUSION AND FUTURE WORK

We have defined the notion of a transparent network service, made the case for such services, and presented the design, implementation, and use of VTL, a framework to help build such services. Using VTL, we illustrated a range of services, including bandwidth optimizers for high bandwidth-delay product networks, network anonymization, tools for network migration with persistent TCP connections, and tools for wormholing for IDSes. All of our examples and the VTL framework itself can operate without having to modify any applications or operating systems.

In the future we plan to explore more transparent network services, using the VTL framework to do so. For example, we plan to investigate adding security policy traversal capabilities, such as those provided by CODO [36] and STUN [32], to existing, unmodified applications.

6. REFERENCES

[1] AnonymOS LiveCD. http://sourceforge.net/projects/anonym-os/.

[2] Bakre, A., and Badrinath, B. R. I-tcp: Indirect tcp for mobile hosts. In *Proc. International Conference on Distributed Computing Systems* (April 1995), pp. 136–143.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. of the 19th ACM symposium on Operating systems principles (SOSP)* (October 2003), pp. 164–177.

[4] Border, J., Kojo, M., Griner, J., Montenegro, G., and Shelby, Z. Performance enhancing proxies intended to mitigate link-related degradations. Tech. Rep. RFC 3135, Network Working Group, June 2001.

[5] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proc. of the Symposium on Networked Systems Design and Implementation (NSDI)* (July 2005), pp. 273–286.

[6] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *Proc. of the 13th USENIX Security Symposium* (August 2004).

[7] Figueiredo, R., Dinda, P. A., and Fortes, J. A case for grid computing on virtual machines. In *Proc. of the 23rd International Conference on Distributed Computing Systems (ICDCS)* (May 2003), pp. 550–559.

[8] Figueiredo, R., Dinda, P. A., and Fortes, J. Special issue on virtualization. *IEEE Computer* (May 2005).

[9] Ganguly, A., Agrawal, A., Boykin, P., and Figueiredo, R. Wow: Self-organizing wide area overlay networks of virtual workstations. In *Proc. of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (June 2006), pp. 30–42.

[10] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th ACM symposium on Operating systems principles SOSP* (October 2003), pp. 193–206.

[11] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).

[12] Gu, Y., and Grossman, R. L. Udt: An application level transport protocol for grid computing. In *2nd International Workshop on Protocols for Long-Distance Networks (PFLDNet)* (February 2004), pp. 13–14.

[13] Gupta, A., and Dinda, P. A. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proc. of the 10th*

Workshop on Job Scheduling Strategies for Parallel Processing (JSPPS) (June 2004), pp. 125–143.

[14] Gupta, A., Zangrilli, M., Sundararaj, A., Huang, A., Dinda, P., and Lowekamp, B. Free network measurement for virtual machine distributed computing. In *Proc. of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (April 2006).

[15] Huang, W., Liu, J., Abali, B., and Panda, D. K. A case for high performance computing with virtual machines. In *Proc. of the 20th annual International Conference on Supercomputing (ICS)* (June 2006), pp. 125–134.

[16] JanusVM: An Internet Privacy Appliance. http://janusvm.peertech.org.

[17] Jiang, X., and Xu, D. Soda: A service-on-demand architecture for application service hosting utility platforms. In *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (June 2003), pp. 174–183.

[18] Keahey, K., Foster, I., Freeman, T., and Zhang, X. Virtual workspaces: Achieving quality of service and quality of life in the grid. *Scientific Programming* 3, 14 (2005).

[19] Koblas, D., and Koblas, M. R. Socks. In *UNIX Security III Symposium* (September 1992), Usenix, pp. 77–88.

[20] Kozuch, M., Satyanarayanan, M., Bressoud, T., and Ke, Y. Efficient state transfer for Internet suspend/resume. Tech. Rep. IRP-TR-02-03, Intel Research Laboratory at Pittsburgh, May 2002.

[21] Lakshman, T., and Madhow, U. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking 5*, 3 (July 1997), 336–350.

[22] Lange, J., Sundararaj, A., and Dinda, P. Automatic dynamic run-time optical network reservations. In *Proc. of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2005), pp. 255-264.

[23] Lange, J., Dinda, P., and Bustamante, F. Vortex: Enabling Cooperative Selective Wormholing for Network Security Systems, In Submission.

[24] LeVasseur, J., Uhlig, V., Stoess, J., and Goetz, S. Unmodified device driver reuse and improved system dependability. In *Proc. of the Symposium on Operating Systems Design and Implemetation (OSDI)* (December 2004), pp. 17–30.

[25] Libnet. http://libnet.sourceforge.net/.

[26] Lin, B., and Dinda, P. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proc. of ACM/IEEE SC (Supercomputing)* (November 2005), pp. 8.

[27] Liu, J., Huang, W., Abali, B., and Panda, D. K. High performance VMM-bypass I/O in virtual machines. In *Proc. of the USENIX Annual Technical Conference* (May 2006).

[28] McCane, S., and Jacobson, V. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter* (January 1993), pp. 259–270.

[29] Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. Process migration. *ACM Computing Surveys 32*, 3 (September 2000), 241–299.

[30] Nelson, M., Lim, B.-H., and Hutchins, G. Fast transparent migration for virtual machines. In *Proc. of the USENIX Annual Technical Conference* (April 2005), pp. 391–394.

[31] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The design and implementation of Zap: A system for migrating computing environments. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2002), pp. 361–376.

[32] Rosenberg, J., Weinberger, J., Huitema, C., and Mahy, R. Stun: Simple traversal of user datagram protocol (udp) through network address translators (nats). Tech. Rep. RFC 3489, Internet Engineering Task Force, March 2003.

[33] Ruth, P., Jiang, X., Xu, D., and Goasguen, S. Virtual distributed environments in a shared infrastructure. *IEEE Computer* (May 2005), pp. 63–69.

[34] Salz, J., Snoeren, A., and Balakrishnan, H. TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-End Network Services. In *4th Usenix Symposium on Internet Technologies and Systems* (Seattle, WA, March 2003).

[35] Sapuntzakis, C., Chandra, R., Praff, B., Chow, J., Lam, M., and Rosenblum, M. Optimizing the migration of virtual computers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2002), pp. 377–390.

[36] Son, S., Allcock, B., and Livny, M. Codo: Firewall traversal by cooperative on-demand opening. In *Proc. of the 14th IEEE International Symposium on High-Performance Distributed Computing (HPDC)* (July 2005), pp. 233–242.

[37] Sundararaj, A., and Dinda, P. Towards virtual networks for virtual machine grid computing. In *Proc. of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM)* (May 2004).

[38] Sundararaj, A., Gupta, A., , and Dinda, P. Increasing application performance in virtual environments through run-time inference and adaptation. In *Proc. of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2005), pp. 47–58.

[39] VMWare Corporation. http://www.vmware.com/.

[40] Waldsburger, C. Memory resource management in vmware esx server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2002), pp. 188–194.

[41] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 195–209.

[42] WinPcap. http://www.winpcap.org/.

[43] Zangrilli, M., and Lowekamp, B. B. Using passive traces of application traffic in a network monitoring system. In *Proc. of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (June 2004), pp. 77–86.