NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
NWU-CS-03-27
October 24, 2003

Towards Virtual Networks for Virtual Machine Grid Computing

Ananth I. Sundararaj **Peter A. Dinda**

Abstract

Virtual machines can greatly simplify wide-area distributed computing by lowering the level of abstraction to the benefit of both resource providers and users. Networking, however, can be a challenge because remote sites are loath to provide connectivity to any machine attached to the site network by outsiders. In response, we have developed a simple and efficient layer two virtual network tool that in effect connects the virtual machine to the home network of the user, making the connectivity problem identical to that faced by the user when connecting any new machine to his own network. We describe this tool and evaluate its performance in LAN and WAN environments. Next, we describe our plans to enhance it to become an adaptive virtual network that will dynamically modify its topology and routing rules in response to the offered traffic load of the virtual machines it supports and to the load of the underlying network. We formalize the adaptation problem induced by this scheme and take initial steps to solving it. The virtual network will also be able to use underlying resource reservation mechanisms on behalf of virtual machines. Both adaptation and reservation will work with existing, unmodified applications and operating systems.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-022444 and a gift from VMWare.. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

Keywords: virtual machines, grid computing, VLANs, VPNs, overlay networks

Towards Virtual Networks for Virtual Machine Grid Computing

Ananth I. Sundararaj Peter A. Dinda
{ais,pdinda}@cs.northwestern.edu
Department of Computer Science, Northwestern University

Abstract

Virtual machines can greatly simplify wide-area distributed computing by lowering the level of abstraction to the benefit of both resource providers and users. Networking, however, can be a challenge because remote sites are loath to provide connectivity to any machine attached to the site network by outsiders. In response, we have developed a simple and efficient layer two virtual network tool that in effect connects the virtual machine to the home network of the user, making the connectivity problem identical to that faced by the user when connecting any new machine to his own network. We describe this tool and evaluate its performance in LAN and WAN environments. Next, we describe our plans to enhance it to become an adaptive virtual network that will dynamically modify its topology and routing rules in response to the offered traffic load of the virtual machines it supports and to the load of the underlying network. We formalize the adaptation problem induced by this scheme and take initial steps to solving it. The virtual network will also be able to use underlying resource reservation mechanisms on behalf of virtual machines. Both adaptation and reservation will work with existing, unmodified applications and operating systems.

1 Introduction

Recently, interest in using OS-level virtual machines as the abstraction for grid computing and for distributed computing in general has been growing [11, 21, 13, 15]. Virtual machine monitors such as VMware [37], IBM's VM [17], and Microsoft's Virtual Server [27], as well as virtual server technology such as UML [4], Ensim [9], and Virutozzo [36], have the potential to greatly simplify management from the perspective of resource owners and to provide great flexibility to resource users. Much grid middleware and application software is quite complex. Being able to package a working virtual machine image that contains the correct operating system, libraries, middleware, and application can make it much easier to deploy something new, using relatively simple middleware that knows only about virtual machines. We have made a detailed case for grid computing on virtual machines in a previous paper [11].

Unlike traditional units of work in distributed systems, such as jobs, processes, or RPC calls, a virtual machine has, and must have, a direct presence on the network at layer 3 and below. We must be able to communicate with it. VMM software recognizes this need and typically creates a virtual Ethernet card for the guest operating system to use. This virtual card is then emulated using the physical network card in the host machine in one of several ways. The most flexible of these bridges the virtual card directly to the same network as the physical card, making the virtual machine a first class citizen on the same network, indistinguishable from a physical machine.

Within a single site, this works very well, as there are existing mechanisms to provide new machines with access. Grid computing, however, is intrinsically about using multiple sites, with different network management and security philosophies, often spread over the wide area [12]. Running a virtual machine on a remote site is equivalent to visiting the site and connecting a new machine. The nature of the network presence (active Ethernet port, traffic not blocked, routable IP address, forwarding of its packets through firewalls, etc) the machine gets, or whether it gets a presence at all, depends completely on the policy of the site. The impact of this variation is further exacerbated as the number of sites is increased, and if we permit virtual machines to migrate from site to site.

To deal with this problem in our own project, we have developed VNET, a simple layer 2 virtual network tool. Using VNET, virtual machines have no network presence at all on a remote site. Instead, VNET provides a mechanism to project their virtual network cards onto another network, which also moves the network management problem from

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, EIA-0224449, and a gift from VMWare. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

one network to another. For example, all of a user's virtual machines can be made to appear to be connected to the user's own network, where the user can use his existing mechanisms to assure that they have appropriate network presence. Because the virtual network is a layer 2 one, a machine can be migrated from site to site without changing its presence—it always keeps the same IP address, routes, etc. The first part of this paper describes how VNET works and presents performance results for local-area and wide-area use. VNET is publicly available from us.

As we have developed VNET, we have come to believe that virtual networks designed specifically for virtual machine grid computing can be used for much more than simplifying the management problem. In particular, because they see all of the traffic of the virtual machines, they are in an ideal position to (1) measure the traffic load and application topology of the virtual machines, (2) monitor the underlying network, (3) adapt application as measured by (1) to the network as measured by (2) by relocating virtual machines and modifying the virtual network topology and routing rules, and (4) take advantage of resource reservation mechanisms in the underlying network. Best of all, these services can be done on behalf of existing, unmodified applications and operating systems running in the virtual machines. The second part of this paper lays out this argument, formalizes the adaptation problem, and takes initial steps to solving it.

2 Related work

Our work builds on operating-system level virtual machines, of which there are essentially two kinds. Virtual machine monitors, such as VMware [37], IBM's VM [17], and Microsoft's Virtual Server [27] present an abstraction that is identical to a physical machine. For example, VMWare, which we use, provides the abstraction of an Intel IA32-based PC (including one or more processors, memory, IDE or SCSI disk controllers, disks, network interface cards, video card, BIOS, etc.) On top of this abstraction, almost any existing PC operating system and its applications can be installed and run. The overhead of this emulation can be made to be quite low [33, 11]. Our work is also applicable to virtual server technology such as UML [4], Ensim [9], Denali [38], and Virtozzo [36]. Here, existing operating systems are extended to provide a notion of server id (or protection domain) along with process id. Each OS call is then evaluated in the context of the server id of the calling process, giving the illusion that the processes associated with a particular server id are the only processes in the OS and providing root privileges that are effective only within that protection domain. In both cases, the virtual machine has the illusion of having network adaptors that it can use as it sees fit, which is the essential requirement of our work.

The Stanford Collective is seeking to create a compute utility in which "virtual appliances" (virtual machines with task-specialized operating systems and applications that are intended to be easy to maintain) can be run in a trusted environment [30, 13]. Part of the Collective middleware is able to create "virtual appliance networks" (VANs), which essentially tie a group of virtual appliances to an Ethernet VLAN. Our work is similar in that we also, in effect, tie a group of virtual machines together as a LAN. However, we differ in that the collective middleware attempts also to solve IP address and routing, while we remain completely at layer 2 and push this administration problem back to the user's site. Another difference is that we expect to be running in a wide area environment in which remote sites are not under our administrative control. Hence, we make the administrative requirements at the remote site extremely simple and focused almost entirely on the machine that will host the virtual machine. Finally, because the nature of the applications and networking hardware in grid computing tend to be different (parallel scientific applications running on clusters with very high speed wide area networks) from virtual appliances, the nature of the adaptation problems and the exploitation of resource reservations made possible by VNET are also different. A contribution of this paper is to describe these problems. However, we do point out that one adaptation mechanism that we plan to use, migration, has been extensively studied by the Collective group [31].

Perhaps closest to our work is that of Purdue's SODA project, which aims to build a service-on-demand grid infrastructure based on virtual server technology [21] and virtual networking [22]. Similar to VANs in the Collective, the SODA virtual network, VIOLIN, allows for the dynamic setup of an arbitrary private layer 2 and layer 3 virtual network among virtual servers. In contrast, VNET works entirely at layer 2 and with the more general virtual machine monitor model. Furthermore, our model has been much more strongly motivated by the need to deal with unfriendly administrative policies at remote sites and to perform adaptation and exploit resource reservations, as we describe later. This paper also includes detailed performance results for VNET, which are not currently available, to the best of our knowledge, for VAN or VIOLIN.

VNET is a virtual private network (VPN [10, 14, 19]) that implements a virtual local area network (VLAN [18]) spread over a wide area using layer 2 tunneling [35]. We are extending VNET to act as an adaptive overlay network [1, 3, 16, 20] for virtual machines as opposed to for specific applications. The adaptation problems introduced are in some ways generalizations (because we have control over machine location as well as the overlay topology and routing) of the problems encountered in the design of and routing on overlays [32]. There is also a strong connection to parallel task graph mapping problems [2, 23].

3 Virtuoso model

We are developing middleware, Virtouso, for virtual machine grid computing that for a user very closely emulates the existing process of buying, configuring, and using an Intel-based computer, a process with which many users and certainly all system administrators are familiar with.

In our model, the user visits a web site, much like the web site of Dell or IBM or any other company that sells Intel-based computers. The site allows him to specify the hardware and software configuration of a computer and its performance requirements, and then order one or more of them. The user receives a reference to the virtual machine which he can then use to start, stop, reset, and clone the machine. The system presents the illusion that the virtual machine is right next to the user. The console display is sent back to the user's machine, the CD-ROM is proxied to the user's machine's CD-ROM, and the virtual machine appears to be plugged into the network side-by-side with the user's machine. The user can then install additional software, including operating systems. The system is permitted to move the virtual machine from site to site to optimize its performance or cost, but must preserve the illusion.

We use VMWare GSX Server [37] running on Linux as our virtual machine monitor. Although GSX provides a fast remote console, we use VNC [29] in order to remain independent of the underlying virtual machine monitor. We proxy CD-ROM devices using Linux's extended network block device, or by using CD image files. Network proxying is done using VNET, as described in the next section.

4 VNET: A simple layer 2 virtual network

VNET is the part of our system that creates and maintains the networking illusion, that the user's virtual machines are on the user's local area network. It is a simple proxying scheme that works entirely at user level. The primary dependence it has on the virtual machine monitor is that there must be a mechanism to extract the raw Ethernet packets sent by the virtual network card, and a mechanism to inject raw Ethernet packets into the virtual card. The specific mechanisms we use are packet filters, packet sockets, and VMWare's host-only networking interface. In the following, we describe VMWare's model of networking, how we build upon it, the interface of VNET, and performance results in the local and wide area.

We use the following terminology. The *User* is the owner of the virtual machines (his *VM*s) which he accesses using his *Client* machine. The user also has a *Proxy* machine for networking, although the Proxy and Client can be the same machine. Each VM runs on a *Host*, and multiple VMs may run on each Host. The *Local* environment of a VM is the LAN to which its Host it is connected, while the *Remote* environment is the LAN to which the Client and the Proxy are connected.

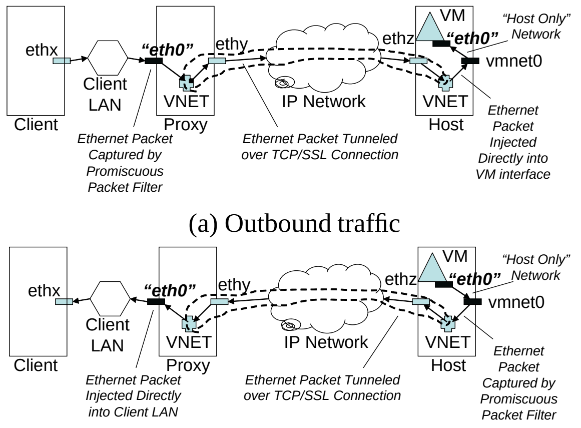
4.1 VMWare networking

VMWare, in its Workstation and GSX Server variants, can connect the virtual network interface to the network in three different ways. To the operating system running in the virtual machine (the VM), they all look the same. By themselves, these connection types are not well suited for use in a wide-area, multi-site environment, as we describe below.

The simplest connection is "bridged", meaning that VMWare uses the physical interface of the Host to directly emulate the virtual interface in the VM. This emulation is not visible to programs running on the Host. With a bridged connection, the VM shows up as another machine on the Local environment, the LAN of the Host. This creates a network management problem for the Local environment (What is this new machine that has suddenly appeared?) and for the User (Will this machine be given network connectivity? How? What's its address? Can I route to it?). Furthermore, if the VM is moved to a Host on a different network, the problems recur, and new ones rear their ugly head (Has the address to the VM changed? What about all its open connections and related state?)

The next form of connection is the host-only connection. Here, a virtual interface is created on the Host which is connected to the virtual interface in the VM. When brought up with the appropriate private IP addresses and routes, this enables programs on the host to talk to programs on the VM. Because we need to be able to talk to the VM from the Client and other machines, host-only networking is insufficient. However, it also has the minimum possible interaction with network administration in the Local environment.

The final form of connection is via network address translation (NAT), a commonly used technique in border routers and firewalls [7]. Similar to a host-only connection, a virtual interface on the Host is connected to the virtual interface on the VM, and appropriate private IP addresses and routes are assigned. In addition, a daemon running on the Host receives IP packets on the interface. For each outgoing TCP connection establishment (SYN), it rewrites the packet to appear to come from the IP address of the Host's regular interface, from some unused port. It records this mapping from the IP address and port on the VM to the address and port it assigned. Mappings can also be explicitly added for incoming TCP connections or UDP traffic. When a packet arrives on the regular interface for the IP and port, it rewrites it using the recorded mapping and passes it to the VM. To


ethx
Client
Client LAN
"eth0"
ethy
VNET
Proxy
IP Network
ethz
VM
"eth0"
"Host Only" Network
vmnet0
VNET
Host
Ethernet Packet Captured by Promiscuous Packet Filter
Ethernet Packet Tunneled over TCP/SSL Connection
Ethernet Packet Injected Directly into VM interface
Ethernet Packet Injected Directly into Client LAN
Ethernet Packet Captured by Promiscuous Packet Filter

(a) Outbound traffic

(b) Inbound traffic

Figure 1. VNET configuration for a single remote virtual machine. Multiple virtual machines on the Host are possible, as are multiple hosts. Only a single Proxy is needed, and it can be the same as the Client. (a) Outbound traffic, (b) Inbound traffic.

the outside world, it simply appears that the Host is generating ordinary packets. To the VM, it appears as if it has a direct connection to the Local environment. For our purposes, NAT networking is insufficient because it is painful to make incoming traffic work correctly as the mappings must be established manually. Furthermore, in some cases it would be necessary for the IP address of the virtual machine to change when it is migrated, making it impossible to maintain connections.

4.2 A bridge with long wires

In essence, VNET provides bridged networking, except that the VM is bridged to the Remote network, the network of the Client. VNET consists of a client and a server. The client is used simply to instruct servers to do work on its behalf. Each physical machine that can instantiate virtual machines (a Host) runs a single VNET server. At least one machine on the user's network also runs a VNET server. We refer to this machine as the Proxy. The user's machine is referred to as the Client. The Client and the Proxy can be the same machine. VNET consists of approximately 4000 lines of C++.

Figure 1 helps to illustrate the operation of VNET. VNET servers are run on the Host and the Proxy and are connected using a TCP connection that can optionally be encrypted using SSL. The VNET server running on the Host opens the Host's virtual interface in promiscuous mode and installs a packet filter that matches Ethernet packets whose source address is that of the VM's virtual interface. The VNET server on the Proxy opens the Proxy's physical interface in promiscuous mode and installs a packet filter that matches Ethernet packets whose destination address is that of the VM's virtual interface or is the Ethernet broadcast and/or (optionally) multicast addresses. To avoid loops, the packet must also not have a source address matching the VM's address. In each case, the VNET server is using the Berkeley packet filter interface [26] as implemented in libpcap, functionality available on all Unix platforms, as well as Microsoft Windows.

When the Proxy's VNET server sees a matching packet, it serializes it to the TCP connection to the Host's VNET server. On receiving the packet, the Proxy's VNET server directly injects the packet into the virtual network interface of the Host (using libnet, which is built on packet sockets, also available on both Unix and Windows) which causes it to be delivered to the VM's virtual network interface. Figure 1(a) illustrates the path of such outbound traffic. When the Host's VNET server sees a matching packet, it serializes it to the Proxy's VNET server. The Proxy's VNET server in turn directly injects it into the physical network interface card, which causes it to be sent on the LAN of the Client. Figure 1(b) illustrates the path of such inbound traffic.

The end-effect of such a VNET *Handler* is that the VM appears to be connected to the Remote Ethernet network exactly where the Proxy is connected. A Handler is identified by the following information:

- IP addresses of the Host and Proxy
- TCP ports on the Host and Proxy used by the VNET servers
- Ethernet devices used on the Host and Proxy
- Ethernet addresses which are proxied. These are typically the address of the VM and the broadcast address, but a single handler can support many addresses if needed.
- Roles assigned to the two machines (which is the Host and which is the Proxy)

A single VNET server can support an arbitrary number of handlers, and can act in either the Host or Proxy role for each. Each handler can support multiple addresses. Hence, for example, the single physical interface on a Proxy could provide connectivity for many VMs spread over many sites. Multiple Proxies or multiple interfaces in a single Proxy could be used to increase bandwidth, up to the limit of the User's site's bandwidth to the broader network.

Because VNET operates at the data link layer, it is agnostic about the network layer, meaning protocols other than IP can be used. Furthermore, because we keep the MAC address of the VM's virtual Ethernet adaptor and the LAN to which it appears to be connected fixed for the lifetime of the VM, migrating the VM does not require any participation from the VM's OS, and all connections remain open after a migration.

A VNET client wishing to establish a handler between

Command	Description
HELLO passwd version	Establish Session
DONE	Finish Session
DEVICES?	Return available network interfaces
HANDLERS?	Return currently running handlers
CLOSE handler	Tear down an existing handler
HANDLE remotepasswd	Establish a handler
local_config	(Described in text)
local_device	
remote_config	
remote_address	
remote_port	
remote_device	
macaddress+	
BEGIN local_config	Establish a handler
local_device	(Described in text)
remote_config	
remote_device	
macaddress+	

Figure 2. VNET interface.

two VNET servers can contact either one. This is convenient, because if only one of the VNET servers is behind a NAT firewall, it can initiate the handler with an outgoing connection through the firewall. If the client is on the same network as the firewall, VNET then requires only that a single port be open on the other site's firewall. If it is not, then both sites need to allow a single port through. If the desired port is not permitted through, there are two options. First, the VNET servers can be configured to use a common port. Second, if only SSH connections are possible, VNET's TCP connection can be tunneled through SSH.

4.3 Interface

VNET servers are run on the Host and the Proxy. A VNET client can contact any server to query status or to instruct it to perform an action on its behalf. The basic protocol is text-based, making it readily scriptable, and bootstraps to binary mode when a Handler is established. Optionally, it can be encrypted for security. Figure 2 illustrates the interface that a VNET Server presents.

Session establishment and teardown: The establishment of session with a VNET server is initiated by a VNET client or another server using the HELLO command. The client authenticates by presenting a password or by using an SSL certificate. Session teardown is initiated by the VNET client using the DONE command.

Handler establishment and teardown: After a VNET client has established a session with a VNET server, it can ask the server to establish a Handler with another server. This is accomplished using the HANDLE command. As shown in Figure 2, the arguments to this command are the parameters that define a Handler as described earlier. Here, `local_config` and `remote_config` refer to the Handler roles. In response to a HANDLE command, the server will establish a session with the other server in the Handler pair, authenticating as before. It will then issue a BEGIN command to inform the other VNET server of its intentions. If the other server agrees, both servers will bootstrap to a binary protocol for communicating Ethernet packets. The Handler will remain in place until one of the servers closes the TCP connection between them. This can be initiated by a client using the CLOSE command, directed at either server.

Status Enquiry: A client can discover a server's available network interfaces (DEVICES?) and what Handlers it is currently participating in (HANDLERS?).

4.4 Performance

Our goal for VNET was to make it easy to convey the network management problem induced by VMs to the home network of the user where it can be dealt with using familiar techniques. However, it is important that VNET's overhead not be prohibitively high, certainly not in the wide area. From the strongest to the weakest goal, VNET's performance should be

- in line with what the physical network is capable of,
- comparable to other networking solutions that don't address the network management problem, and
- sufficient for the applications and scenarios where it is used.

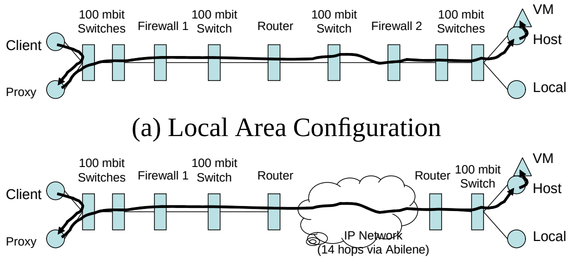
We have found that our implementation meets the later two goals, and, in many cases, meets the first, strongest goal as well.

Metrics

Latency and throughput are the most fundamental measures used to evaluate the performance of networks. The time for a small transfer is dominated by latency, while that for a large transfer is dominated by throughput. Interactivity, which is often dominated by small transfers, suffers if latencies are either high or highly variable [8]. Bulk transfers suffer if throughput is low. Our measurements were conducted on working days (Monday through Thursday) in the early morning to eliminate time-of-day effects.

Latency: To measure latency, we used the round-trip delay of an ICMP echo request/response pair (i.e., ping), taking samples over hour-long intervals. We computed the average, minimum, maximum and standard deviation of these measurements. Here, we report the average and standard deviation. Notice that this measure of latency is symmetric.

Throughput: To measure average throughput, we use the *ttcp* program. Ttcp is commonly used to test TCP and



(a) Local Area Configuration

(b) Wide Area Configuration

Figure 3. VNET test configurations for the local area (a) and the wide area (b). Local area is between two labs in the Northwestern CS Department. Wide Area is between the first of those labs and a lab at Carnegie Mellon.

UDP performance in IP networks. Ttcp times the transmission and reception of data between two systems. We use a socket buffer size of 64 KBytes and transfer a total of 1 GB of data in each test. VNET's TCP connection also uses a socket buffer size of 64 KBytes. TCP socket buffer size can limit performance if it is less than the bandwidth-delay product of the network path, hence our larger-than-default buffers. All throughput measurements were performed in both directions.

Testbeds

Although VNET is targeted primarily for wide-area distributed computing, we evaluated performance in both a LAN and a WAN. Because our LAN testbed provides much lower latency and much higher throughput than our WAN testbed, it allows us to see the overheads due to VNET more clearly. The Client, Proxy, and Host machines are 1 GHz Pentium III machines with Intel Pro/100 adaptors.

Our testbeds are illustrated in Figure 3. The LAN and WAN testbeds are identical up to and including the first router out from the Client. This portion is our firewalled lab in the Northwestern CS department. The LAN testbed then connects, via a router which is under university IT control (not ours), to another firewalled lab in our department which is a separate, private IP network. The WAN testbed instead connects via the same router to the Northwestern backbone, the Abiline network, the Pittsburgh Supercomputing Center, and two administrative levels of the campus network at Carnegie Mellon, and finally to an lab machine there. Notice that even a LAN environment can exhibit the network management problem. It is important to stress that the only requirement that VNET places on either of these complex environments is the ability to create a TCP connection between the Host and Proxy in some way.

We measured the latency and throughput of the underlying "physical" IP network, VMWare's virtual networking options, VNET, and of SSH connections:

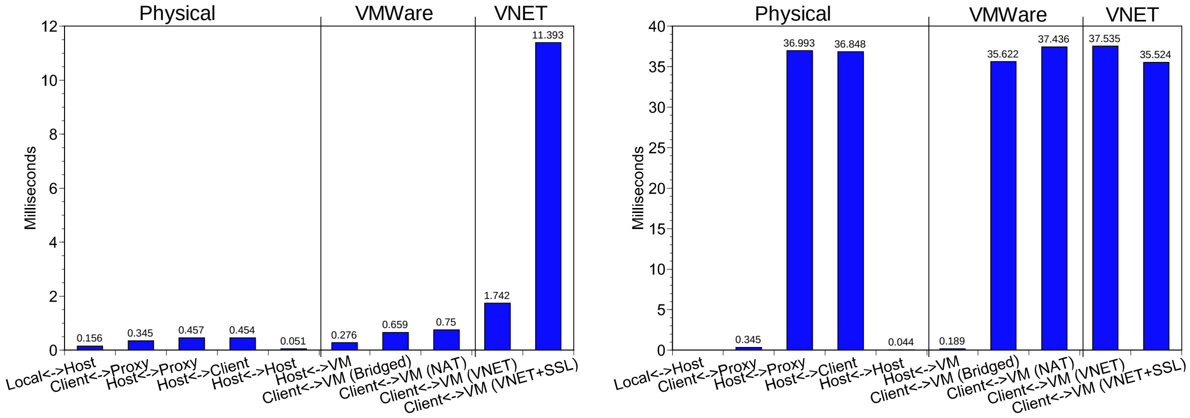
- *Physical:* VNET transfers Ethernet packets over multiple hops in the underlying network. We measure equivalent hops, and also end-to-end transfers, excepting the VM.
  - $Local \leftrightarrow Host$: Machine on the Host's LAN to/from the Host.
  - $Client \leftrightarrow Proxy$: Analogous to the first hop for an outgoing packet in VNET and the last hop for an incoming packet.
  - $Host \leftrightarrow Proxy$: Analogous to the TCP connection of a Handler, the tunnel between the two VNET servers.
  - $Host \leftrightarrow Client$: End-to-end except for the VM.
  - $Host \leftrightarrow Host$: Internal transfer on the Host.
- *VMWare:* Here we consider the performance of all three of VMWare's options, described earlier.
  - $Host \leftrightarrow VM$: Host-only networking, which VNET builds upon.
  - $Client \leftrightarrow VM$ *(Bridged)*: Bridged networking. This leaves the network administration problem at the remote site.
  - $Client \leftrightarrow VM$ *(NAT)*: NAT-based networking. This partially solves the network administration problem at the remote site at the layer 3, but creates an asymmetry between incoming and outgoing connections, and does not support VM migration. It's close to VNET in that network traffic is routed through a user-level server.
- *VNET:* Here we use VNET to project the VM onto the Client's network.
  - $Client \leftrightarrow VM$ *(VNET)*: VNET without SSL
  - $Client \leftrightarrow VM$ *(VNET+SSL)*: VNET with SSL
- *SSH:* Here we look at the throughput of an SSH connection between the Client and the Host to compare with VNET with SSL.
  - $Host \leftrightarrow Client$ *(SSH)*

Discussion

The results of our performance tests are presented in Figures 4 through 6.

Average latency: Figure 4 shows the average latency in the LAN (Figure 4(a)) and WAN (Figure 4(b)).

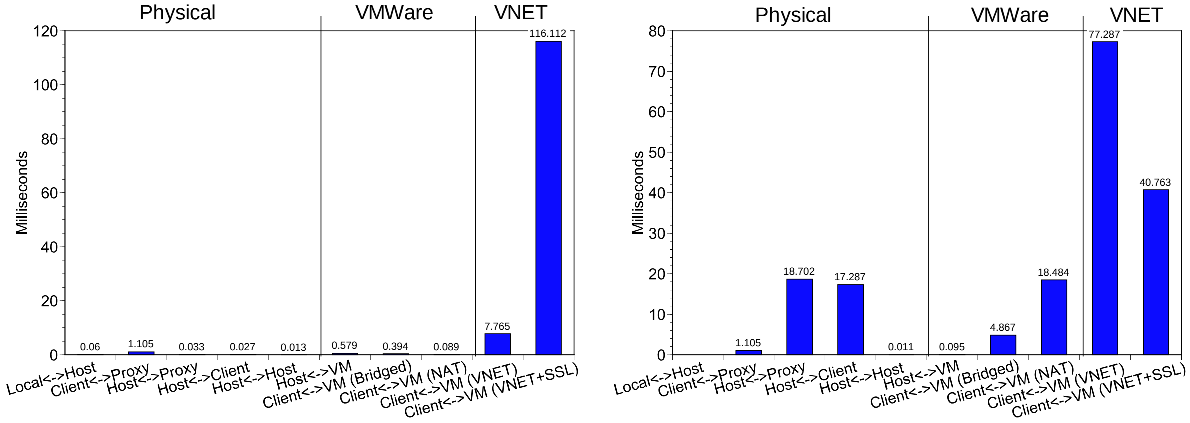
In Figure 4(a), we see that the average latency on the LAN when using VNET without SSL is 1.742 ms. It is important to understand exactly what is happening. The Client is sending an ICMP echo request to the VM. The request is first intercepted by the Proxy, then sent to the Host, and finally the Host sends it to the VM (see Figure 3(a)). The reverse path for the echo reply is similar. These three distinct pieces have average latencies of 0.345 ms, 0.457 ms, and 0.276 ms, respectively, on the physical network, which



(a) Local Area

(b) Wide Area

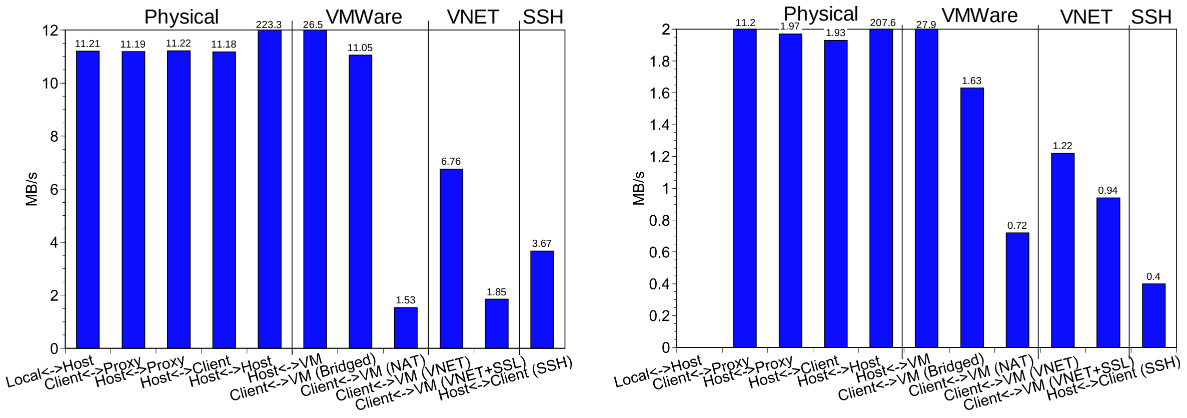
Figure 4. Average latency



(a) Local Area

(b) Wide Area

Figure 5. Standard deviation of latency



(a) Local Area

(b) Wide Area

Figure 6. Bandwidth

totals 1.078 ms. In the LAN, VNET without SSL increases latency by 0.664 ms, or about 60%. We claim that this is not prohibitive, especially in absolute terms. Hence we note that the operation of VNET over LAN does not add prohibitively to the physical latencies. The VMWare NAT option, which is the closest analog to VNET, except for moving the network management problem, has about 1/2 of the latency. When SSL encryption is turned on, VNET latency grows to 11.393 ms, 10.3 ms and a factor of 10 higher than what is possible on the (unencrypted) physical network.

In Figure 4(b), we note that the average latency on the WAN when using VNET without SSL is 37.535 ms and with SSL encryption is 35.524 ms. If we add up the constituent latencies as done above, we see that the total is 37.527 ms. In other words, VNET with or without SSL has average latency comparable to what is possible on the physical network in the WAN. The average latencies seen by VMWare's networking options are also roughly the same. In the wide area, average latency is dominated by the distance, and we get the benefits of VNET with negligible additional cost. This result is very encouraging for the deployment of VNET in the context of grid computing, our primary purpose for it.

Standard deviation of latency: Figure 5 presents the standard deviation of latency in the LAN (Figure 5(a)) and WAN (Figure 5(b)).

In Figure 5(a), we see that the standard deviation of latency using VNET without SSL in the LAN is 7.765 ms, while SSL increases that to 116.112 ms. Adding constituent parts only totals 1.717 ms, so VNET has clearly dramatically increased the variability in latency, which is unfortunate for interactive applications. We believe this large variability is because the TCP connection between VNET servers inherently trades packet loss for increased delay. For the physical network, we noticed end-to-end packet loss of approximately 1%. VNET packet losses were nil. VNET resends any TCP segment that contains an Ethernet packet that in turn contains an ICMP request/response. This means that the ICMP packet eventually gets through, but is now counted as a high delay packet instead of a lost packet, increasing the standard deviation of latency we measure. A histogram of the ping times shows that almost all delays are a multiple of the round-trip time.

In Figure 5(b), we note that the standard deviation of latency on the WAN when using VNET without SSL is 77.287 ms and with SSL is 40.783 ms. Adding the constituent latencies totals only 19.902 ms, showing that we have an unexpected overhead factor of 2 to 4. We again suspect high packet loss rates in the underlying network lead to retransmissions in VNET and hence lower packet loss rates, but a higher standard deviation of latency. We measured a 7% packet loss rate in the physical network compared to 0% with VNET. We again noticed that latencies which deviated from the average did so in multiples of the average latency, supporting our explanation.

Average Throughput: Figure 6 presents the measurements for the average throughput in the LAN (Figure 6(a)) and WAN (Figure 6(b)).

In Figure 6(a), we see that the average throughput in the LAN when using VNET without SSL is 6.76 MB/sec and with SSL drops to 1.85 MB/sec, while the average throughput for the physical network equivalent is 11.18 MB/sec. We were somewhat surprised with the VNET numbers. We expected that we would be very close to the throughput obtained in the physical network, similar to those achieved by VMWare's host-only and bridged networking options. Instead, our performance is lower than these, but considerably higher than VMWare's NAT option.

In the throughput tests, we essentially have one TCP connection (that used by the ttcps running on the VM and Client) riding on a second TCP connection (that between the two VNET servers on Host and Proxy). A packet loss in the underlying VNET TCP connection will lead to a retransmission and delay for the ttcp TCP connection, which in turn could time out and retransmit itself. On the physical network there is only ttcp's TCP. Here, packet losses might often be detected by the receipt of triple duplicate acknowledgements followed by fast retransmit. However, with VNET, more often than not a loss in the underlying TCP connection will lead to a packet loss detection in ttcp's TCP connection by the expiration of the retransmission timer. The difference is that when a packet loss is detected by timer expiration the TCP connection will enter slow start, dramatically slowing the rate. In contrast, a triple duplicate acknowledgement does not have the effect of triggering slow start.

In essence, VNET is tricking ttcp's TCP connection into thinking that the round-trip time is highly variable when what is really occurring is hidden packet losses. In general, we suspect that TCP's congestion control algorithms are responsible for slowing down the rate and reducing the average throughput. This situation is somewhat similar to that of a *split TCP connection*. A detailed analysis of the throughput in such a case can be found elsewhere [34]. The use of encryption with SSL further reduces the throughput.

In Figure 6(b), we note that the average throughput over the WAN when using VNET without SSL encryption is 1.22 MB/sec and with SSL is 0.94 MB/sec. The average throughput on the physical network is 1.93 MB/sec. Further, we note that the throughput when using VMWare's bridged networking option is only slightly higher than the case where VNET is used (1.63 MB/sec vs. 1.22 MB/sec), while VMWare NAT is considerably slower. Again, as described above, this difference in throughput is probably due to the overlaying of two TCP connections. Notice, however, that the difference is much less than that in the LAN as now there are many more packet losses that in both cases will be detected by ttcp's TCP connection by the expiration of the retransmission timer. Again, the use of encryption with SSL further reduces the throughput.

We initially thought that our highly variable latencies (and corresponding lower-than-ideal TCP throughput) in VNET were due to the priority of the VNET server processes. Conceivably, the VNET server could respond slowly if there were other higher or similar priority processes on the Host, Proxy, or both. To test this hypothesis we tried giving the VNET server processes maximum priority, but this did not change delays or throughput. Hence, this hypothesis was incorrect.

We also compared our implementation of encryption using SSL in the VNET server to SSH's implementation of SSL encryption. We used SCP to copy 1 GB of data from the Host to the Client in both the LAN and the WAN. SCP uses SSH for data transfer, and uses the same authentication and provides the same security as SSH. In the LAN case we found the SCP transfer rate to be 3.67 MB/sec compared to the 1.85 MB/sec with VNET along with SSL encryption. This is an indication that our SSL encryption implementation overhead is not unreasonable. In the WAN the SCP transfer rate was 0.4 MB/sec compared to 0.94 MB/sec with VNET with SSL. This further strengthens the claim that our implementation of encryption in the VNET server is reasonably efficient.

Comparing with VMWare NAT: The throughput obtained when using VMWare's NAT option was 1.53 MB/sec in the LAN and 0.72 MB/sec in the WAN. This is significantly lower than the throughput VNET attains both in the LAN and WAN (6.76 MB/sec and 1.22 MB/sec, respectively). As described previously in Section 4.1, VMWare's NAT is a user-level process, similar in principle to a VNET server process. That VNET's performance exceeds that of VMWare NAT, the closest analog in VMWare to VNET's functionality, is very encouraging.

Summary: The following are the main points to take away from our performance evaluation:

- Beyond the physical network and the VMWare networking options, VNET gives us the ability to shift the network management problem to the home network of the client.
- The extra average latency when using VNET deployed over the LAN is quite low while the overhead over the WAN is negligible.
- VNET has considerably higher variability in latency than the physical network. This because it automatically retransmits lost packets. If the underlying network has a high loss rate, then this will be reflected as higher latency variability in VNET. Hence, using VNET, in its current implementation, produces a trade: higher variability in latency for zero visible packet loss.
- VNET's average throughput is lower than that achievable in the underlying network, although not dramatically so. This appears to be due to an interaction between two levels of TCP connections. We are working to fix this.
- VNET's average throughput is significantly better than that of the closest analog, both in terms of functionality and implementation, in VMWare, NAT.
- Using VNET encryption increases average latency and standard deviation of latency by a factor of about 10 compared to the physical network. Encryption also decreases throughput. The VNET encryption results are comparable or faster than those using SSH.

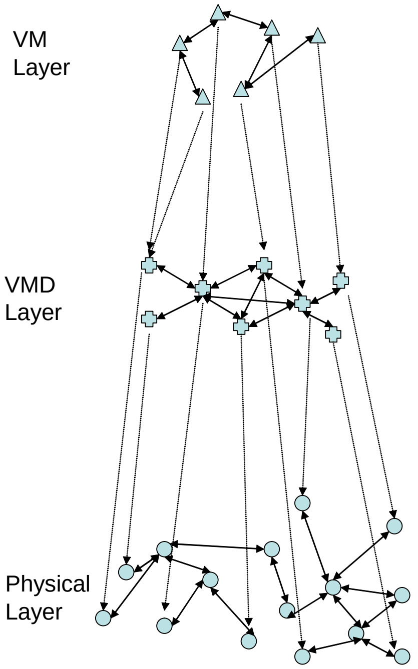
We find that the overheads of VNET, especially in the WAN, are acceptable given what it does, and we are working to make them better. Using VNET, we can transport the network management problem induced by VMs back to the home network of the user, where it can be readily solved, and we can do so with acceptable performance.

5 Towards an adaptive overlay

We designed and implemented VNET in response to the network management problems encountered when running VMs at (potentially multiple) sites where the user has no administrative connection. However, we have come to believe strongly that the overlays like it, specifically designed to support virtual machine computing, have great potential as the mechanisms for adaptation and for exploiting the special features of some networks.

An overlay network has an ideal vantage point to monitor the underlying physical network and the applications running the VMs. Using this information, it can adapt to the communication and computation behavior of the VMs, changing its topology and routing rules, and moving VMs. Requiring code modifications, extensions, or the use of particular application frameworks has resulted in limited adoption of adaptive application technologies. Here, adaptation could be retrofitted with *no modifications* to the operating system and applications running in the virtual machine, and could be completely transparent to the running VMs. Similarly, if the overlay is running on a network that can provide extended services, such as reservations or light-path setup and teardown in an optical network, it could use these features on behalf of an unmodified operating system and its applications.

We are now designing a second generation VNET implementation that will support this vision. The second generation VNET will include support for arbitrary topologies and routing, network and VM monitoring, and interfaces for adaptative control of the overlay, including VM migration, and for using underlying resource reservation mechanisms. In the following, we describe these extensions and then elaborate on the adaptation problem.



G_{vm}(VM,VE)	virtual machine graph (directed)
vm_i in VM	virtual machine (VM) (node)
$ve_{i,j}$ in VE	exists if vm_i sends data to vm_j (edge)
fixed(vm_i,vd_k)	true if vm_i may only be assigned to vd_k
comp(vm_i)	compute rate demand of vm_i
size(vm_i)	size of vm_i in bytes
bw($ve_{i,j}$)	bandwidth demand on $ve_{i,j}$
lat($ve_{i,j}$)	latency demand on $ve_{i,j}$

vmap(vm_i)	***assignment of vm_i to appropriate vd***
	many-to-one, dynamic

G_{vmd}(VD,VDE)	VM daemon graph (directed)
	VM daemons host zero or more VMs
vd_i in VD	VM Daemon (VMD) (node)
$vde_{i,j}$ in VDE	***exists if vd_i sends data directly to vd_j (edge)***
	dynamic
route(vd_i,vd_j)	***path (<vd_k>) taken by data sent from vd_i to vd_j***
route($ve_{i,j}$)	***dynamic***
mcomp(vd_i)	measured compute rate at vd_i
mbw($vde_{i,j}$)	measured available bandwidth on $vde_{i,j}$
mlat($vde_{i,j}$)	measured latency on $vde_{i,j}$

vdmap(vd_i)	***assignment of vd_i to appropriate host***
	static
route($vde_{i,j}$)	***route that $vde_{i,j}$ takes in network topology graph***

G_{net} (N,E)	Network topology graph (directed)
	VM Daemons run only on hosts
n_i in N	router or host
host(n_i)	true if node is a host
$e_{i,j}$ in E	exists if n_i sends data to n_j (edge)
route(n_i,n_j)	path (<n_k>) taken by data sent from vd_i to vd_j
mbw(n_i)	backplane bandwidth of a router or network
mbw($e_{i,j}$)	measured available bandwidth on edge
	may not exist
mlat($e_{i,j}$)	measured latency on edge
	may not exist

Figure 7. Terminology.

5.1 Terminology

Figure 7 serves as a point of reference for the terminology we use in describing our design and the problems it induces. On the left hand side, we can see the three layers: the virtual machines themselves (VM layer), the virtual machine daemons that host them (VMD layer), and the physical network resources on which the VMDs run (Physical layer). The graph of VM Daemons is the overlay itself. A VM daemon or VMD is a generalization of a VNET server that is able to manage VMs and measure their traffic and the characteristics of the underlying network. The nodes and edges at the VM layer are mapped to nodes and routes at the VMD layer. The physical layer is the underlying IP network itself. The nodes of the VMD layer are mapped to the end-systems of this network. The right hand side of Figure 7 shows the symbols we use at each layer and the mapping between layers. The boldfaced symbols represent where adaptation and the use of underlying resource mechanisms can take place.

The VM layer consists of the individual VMs ($vm_i \in VM$) and the communication edges among them ($ve_{i,j} \in VE$), represented as a graph ($G_{vm}(VM, VE)$). If vm_i sends to vm_j, then there is an edge $ve_{i,j}$ in VE. The VM layer is essentially a representation of the demands that the user's VMs are placing on the system. $bw(ve_{i,j})$ and $lat(ve_{i,j})$ are the bandwidth and latency requirements of communication between vm_i and vm_j. $comp(vm_i)$ is the computational demand of vm_i, while $size(vm_i)$ is the total size of its machine image.

The VMD layer consists of VM daemons ($vd_i \in VD$), and the communication edges among them ($vde_{i,j} \in VDE$), represented as a graph ($G_{vmd}(VD, VDE)$). If vd_i sends to vd_j, there must be a route ($route(vd_i, vd_j)$) between them. A virtual machine vm_i is assigned to a single VMD ($vmap(vm_i)$). Hence, an edge $ve_{i,j}$ in the VMD layer corresponds to a route $route(ve_{i,j}) = route(vmap(vm_i), vmap(vm_j))$ in the VMD layer. Multiple VMs may be assigned to a single VMD. At the VMD layer, we also maintain the measured bandwidth and latency of edges in the G_{vmd}, $mbw(vde_{i,j})$, $mlat(vde_{i,j})$, and the computational rate at each node ($mcomp(vd_i)$).

The physical layer consists of the underlying topology, $G_{net}(V, E)$, where the nodes $v_i \in V$ are the routers and hosts at the IP layer and $e_{i,j} \in E$ are the links. $route(v_i, v_j)$ are the routes chosen by the network, and $host(v_i)$ is true if v_i is a host. In addition, we may be able to measure the bandwidth and latency of the elements of a link ($mbw(e_{i,j})$, $mlat(e_{i,j})$), the backplane bandwidth of a router ($mbw(v_i)$), and the raw computational power of a host ($mcomp(v_i)$). Each VMD is assigned to a single host in the physical layer, and each host has at most a single VMD. This mapping is via $vdmap(vd_i)$. Each edge in the G_{vmd} turns into a route $route(vde_{i,j})$ at the physical layer.

5.2 Supporting arbitrary topologies and routing

Currently, a VMD (VNET server) tunnels Ethernet traffic for a particular address to another VMD over a TCP connection, a relationship we refer to as a handler, as described in Section 4.2. In principle, multiple handlers can be multiplexed over a single TCP connection. Hence, the edges in the VMD graph are simply TCP connections. In such a configuration, the VMDs supporting a user form an overlay network with a star topology centered on the proxy machine on the user's network. All messages are routed through the proxy machine.

It is straightforward to see why it is possible to support arbitrary topologies. Each VMD can effectively behave like a switch or router, sending an packet that arrives on a TCP connection to another connection instead of injecting it onto the local network. Indeed, for Ethernet topologies, we can simply emulate the Ethernet switch protocols, as in VIOLIN [22]. Each Ethernet packet contains a source and destination address. A modern Ethernet switch learns the location of Ethernet addresses on its ports based on the source addresses of traffic it sees. Switches also run a distributed algorithm that assures that they form a spanning tree topology.

Because VNET understands that it is supporting VMs, it can go beyond simply emulating an IP or Ethernet network, however. For example, hierarchical routing of Ethernet packets is possible because the Ethernet addresses of the user's VMs are not chosen by Ethernet card vendors, but are assigned by VNET.

5.3 Monitoring the VMs and the network

The VMD layer is ideally placed to monitor both the resource demands placed by the VMs and the resource supplies offered by the underlying physical network, with minimal active participation from either.

Each VMD sees every incoming/outgoing packet to/from each of the VMs that it is hosting. Given a matrix representation of G_{vm}, if the VMD is hosting vm_i it knows the ith row and ith column. Collectively, the VMDs know all of G_{vm}, so a reduction could be used to give each one a full copy of G_{vm}. Hence, without modifying the OS or any operating systems, the VMDs can recover the application topology.

Each VMD is also in a good position to infer the bandwidth and latency demands for each edge, $bw(ve_{i,j})$, $lat(ve_{i,j})$, the computational demand of each VM, $comp(vm_i)$, and the total size of each VM image, $size(vm_i)$ corresponding to the VMs that it is hosting. Again, a reduction would give each VMD a global picture of the the resource demands. Beyond inference, this information could also be provided directly by a developer or administrator without any modifications to the applications or operating system.

VMDs transfer packets on behalf of the VMs they host. An outgoing packet from a VM is routed through it's hosting VMD, then through zero or more transit VMDs, the host VMD of the destination VM, and finally to the destination VM. When such a message is transfered from vd_i to vd_j, the transfer time is a free measurement of the corresponding path $route(vde_{i,j})$ in the underlying network. From collections of such measurements, the two VMDs can derive $mbw(vde_{i,j})$ and $mlat(vde_{i,j})$ using known techniques [5]. A VMD can also periodically measure the available compute rate of its host ($mcomp(vd_i)$) using known techniques [6, 39].

Network monitoring tools such as Remos [5] and NWS [40] can, in some cases, determine the physical layer topology and measure its links, paths, and hosts.

5.4 VM assignment problems

Let us define some additional terminology. Each VM computes at some actual rate:

$$ComputeRate(vm_i)$$

Each pair of VMs have some actual bandwidth and latency:

$$PathBW(vm_i, vm_j) = PathBW(ve_{i,j}) = \min_{vde \in route(vmap(vm_i), vmap(vm_j))} mbw(vde)$$

$$PathLatency(vm_i, vm_j) = PathLatency(ve_{i,j}) = \sum_{vde \in route(vmap(vm_i), vmap(vm_j))} mlat(vde)$$

A VMD has an allocated compute rate, which is the sum of all the rates of VMs mapped to it:

$$AllocatedComputeRate(vd_i) = \sum_{vm \in VM : vmap(vm) = vd_i} comp(vm)$$

Similarly, an edge between two VMDs has an allocated bandwidth:

$$AllocatedBandwidth(vde_{i,j}) = \sum_{ve \in VE : vde_{i,j} \in route(ve)} bw(ve)$$

The VM assignment problem is to find a $vmap$ function that meets the following requirements:

- Complete: $\forall vm \in VM : vmap(vm)$ exists.
- Compliant: $fixed(vm_i, vd_j) \Rightarrow vmap(vm_i) = vd_j$

- Computationally feasible: $\forall vm \in VM : ComputeRate(vm) \geq comp(vm)$ and $\forall vd \in VD : AllocatedComputeRate(vd) \leq mcomp(vd)$
- Communication feasible: $\forall ve \in VE : PathLatency(ve) \leq lat(ve) \wedge PathBW(ve) \geq bw(ve)$ and $\forall vde \in VDE : AllocatedBW(vde) \leq mbw(vde)$
- Routable: $\forall ve \in VE : route(ve)$ exists.

We define four variants of the VM assignment problem. The first two are offline versions while the second two are online problems.

Simple offline VM assignment problem: Given G_{vm} and its associated functions $fixed$, $comp$, $size$, bw, and lat, and G_{vmd} and its associated functions $route$, $mcomp$, mbw, and $mlat$, choose a $vmap$ that meets the $vmap$ requirements. The VMDs are fixed, as are their overlay topology and routing rules. We envision three situations in which this problem arises. The first is if the user has a private VMD network and runs a multi-VM application on it. The problem needs to be solved at program startup, and thereafter whenever the communication patterns have changed dramatically. The second case is when multiple users can map their own virtual machines to a shared VMD infrastructure. In this situation, the VMs of other users can be treated as $fixed$. The problem also occurs if it is the VMD infrastructure that determines the mapping. In that situation, the problem can be solved with no VMs $fixed$ whenever a new set of VMs enters or leaves the system, or periodically.

Complex offline VM assignment problem: Given G_{vm} and its associated functions, determine $vmap$, VDE, and $route$ such that $vmap$ meets the $vmap$ requirements. Here, the VMDs are fixed, but their overlay topology and its routing rules can be chosen to help find a suitable $vmap$. We see this problem occurring in two contexts. The first is if the user has a private VMD network that supports topology and routing changes. The second is for a shared VMD overlay where the VMDs solve the problem collectively over all running VMs.

Simple online VM assignment problem: Given an existing G_{vm}, G_{vmd}, and their associated functions, an existing $vmap$, and a new G'_{vm}, determine $vmap' = f(vmap, G_{vm}, G'_{vm})$ such that it meets the $vmap$ requirements.

Complex online VM assignment problem: Given an existing G_{vm}, G_{vmd}, and their associated functions, an existing $vmap$, $route$, and a new G'_{vm}, determine a new $(vmap', VDE', route') = f(vmap, G_{vm}, G'_{vm})$ such that $vmap$ meets the requirements.

5.5 Connected components

The simple online VM assignment problem can be formulated in terms of connected components. The connected components of a graph are a partitioning of the graph into subgraphs. A connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $U \subseteq V$ such that for every pair of vertices u and v in U, there is a path connecting them. Hence two vertices are in the same connected component if and only if there exists a path between the vertices. If a graph contains only one connected component then it is said to be a connected graph.

The directed graph G_{vm} is a connected graph while the directed graph G_{vmd} may or may not be connected. It will be connected if and only if $\forall\ vd_i \in VD$ there $\exists$ at least one $vm_i \in VM$ such that $vmap(vm_i) = vd_i$. Note that since G_{vm} is connected, the graph of $vd \in VD$ where $\forall\ vd$ there $\exists\ vmap$ such that $vd = vmap(vm_i)$ such that $vm_i \in VM$, will also be connected.

Given an existing G_{vm}, G_{vmd}, and their associated functions, we calculate $cost(vde_{i,j})$ where

$$cost(vde_{i,j}) = f(mbw(vde_{i,j}), mlat(vde_{i,j})) \forall vde_{i,j} \in VDE$$

$cost(vde_{i,j})$ is a measure of the network *cost value* associated with each edge in the VM daemon graph G_{vmd}. The cost value would be an integrated metric, a function of the measured available bandwidth, $mbw(vde_{i,j})$, and latency, $lat(vde_{i,j})$, on the edges in the VM daemon graph G_{vmd}.

The VM assignment problem may now be described as:

Input:

- G_{vm}, G_{vmd}, and their associated functions
- an existing $vmap$
- a connected component $VDC \subseteq VD$ such that $\forall vd_i \in VDC$ there $\exists$ at least one $vm_i \in VM$ such that $vmap(vm_i) = vd_i$
- a cost value $cost(vde_{i,j})\ \forall vde_{i,j} \in VDE$

Output:

- a connected component $VDC' \subseteq VD$ such that $\forall vd_i \in VDC'$ there $\exists$ at least one $vm_i \in VM$ such that $vmap'(vm_i) = vd_i$ and $\sum_{\forall vd_i, vd_j \in VDC'} cost(vde_{i,j})$ is minimum over the set of all possible connected components

Algorithm:

- Each connected component (VDC, VDC', VDC'', etc) represents one particular assignment of $vm_i \in VM$ to $vd_i \in VD$, i.e. a particular $vmap$ meeting all the $vmap$ requirements
- So from amongst all such possible connected components (assignments) we choose that which has the least cost associated with it, i.e. $\sum_{\forall vd_i, vd_j \in VDC'} cost(vde_{i,j})$ is minimum over the set of all possible connected components
- This is equivalent to enumerating all the possible connected components of size | VM | and then choosing the one which has the least cost associated with it.

5.6 Engineering a VMD overlay

For a VMD infrastructure that supports a single user, it is sensible to think of solving the VM assignment problems by exploiting simultaneously the freedom to change the VMD topology and routes, as well as to move the VMs. However, if the VMD infrastructure is to be shared among multiple users, a two stage approach, engineering a sensible general-purpose VMD topology and then doing dynamic assignment of VMs to that topology, is likely to be more sensible.

One approach is to require that the topology of the VMDs, G_{vmd} be a mesh or a hypercube. This induces a network engineering problem, that of finding $vdmap$ such that the chosen topology behaves close to our expectations for its type in terms of the bandwidth and latency of the edges. For both meshes and hypercubes, this means that each edge should be equal in terms of bandwidth and latency. Given that the underlying overlay has such as simple, regular topology, we would assign groups of VMs that exhibit a particular topology to partitions of the VMD graph. For example, it is well known that trees and neighbor patterns can be embedded in hypercubes [24]. If a parallel application running across a set of VMs exhibits one of these patterns, its VMs can be readily (and quickly) assigned to VMDs.

5.7 Exploiting resource reservations

Some networks support reservations of bandwidth along paths. For example, various optical networks support lightpath set up, essentially allowing for an arbitrary topology to be configured. The ODIN software [25] provides an application with an API to exploit such networks. However, it is the onerous task of the programmer to determine the appropriate topology and then use the API to configure it. Here, the VMDs could do the same on behalf of the unmodified application, since they collectively know G_{vm}.

Light-path services, as well as traditionally ISP-level reservations such as DiffServ [28], could also be used to implement an engineered overlay that is shared by multiple users, as described previously.

6 Conclusions

A strong case can be made for grid computing using virtual machine monitors or virtual servers. In either case, the combination of a virtual machine's need for a network presence and the multi-site, multi-security-domain nature inherent in grid computing creates a challenging distributed network management problem. We have described and evaluated a tool, VNET, that addresses this problem by converting it into the familiar single-site network management problem. The combination of VNET and similar virtual network tools and virtual machines present an opportunity: adaptation and exploitation of resource reservations for existing, unmodified operating systems and applications. We described this opportunity in depth, pointing out specific adaptation problems and interactions with a specific resource reservation system. We are currently extending VNET to take advantage of this opportunity.

Acknowledgements

We would like to thank David O'Hallaron, Julio Lopez, and Renato Figueiredo for giving us access to the machines we used in our wide area performance evaluation, and Ashish Gupta, Sonia Fahmy, and Dongyhan Xu for helpful discussions on the overlay aspects of VNET.

References

[1] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (2001).

[2] BOLLINGER, S., AND MIDKIFF, S. Heuristic techniques for processor and link assignment in multicomputers. *IEEE Transactions on Computers 40*, 3 (March 1991).

[3] CAMPBELL, A., KOUNAVIS, M., VILLELA, D., VICENTE, J., MEER, H. D., MIKI, K., AND KALAICHELVAN, K. Spawning networks. *IEEE Network* (July/August 1999), 16–29.

[4] DIKE, J. A user-mode port of the linux kernel. In *Proceedings of the USENIX Annual Linux Showcase and Conference* (Atlanta, GA, October 2000).

[5] DINDA, P., GROSS, T., KARRER, R., LOWEKAMP, B., MILLER, N., STEENKISTE, P., AND MILLER, N. The architecture of the remos system. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 2001)* (August 2001), pp. 252–265.

[6] DINDA, P. A. Online prediction of the running time of tasks. *Cluster Computing 5*, 3 (2002). Earlier version appears in HPDC 2001. Summary in SIGMETRICS 2001.

[7] EGEVANG, K., AND FRANCIS, P. The ip network address translator (nat). Tech. Rep. RFC 1631, Internet Engineering Task Force, May 1994.

[8] EMBLEY, D. W., AND NAGY, G. Behavioral aspects of text editors. *ACM Computing Surveys 13*, 1 (January 1981), 33–70.

[9] ENSIM CORPORATION. http://www.ensim.com.

[10] FERGUSON, P., AND HUSTON, G. What is a vpn? Tech. rep., Cisco Systems, March 1998.

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

[12] Foster, I., Kesselman, C., and Tuecke, S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications 15*, 3 (2001).

[13] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (October 2003).

[14] Gleeson, B., Lin, A., Heinanen, J., Armitage, G., and Malis, A. A framework for ip-based virtual private networks. Tech. Rep. RFC 2764, Internet Engineering Taskforce, February 2000.

[15] Hand, S., Harris, T., Kotsovinos, E., and Pratt, I. Controlling the xenoserver open platform. In *Proceedings of OPENARCH 2003* (April 2003).

[16] hua Chu, Y., Rao, S., Sheshan, S., and Zhang, H. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM 2001* (2001).

[17] IBM Corporation. White paper: S/390 virtual image facility for linux, guide and reference. GC24-5930-03, Feb 2001.

[18] IEEE 802.1Q Working Group. 802.1q: Virtual lans. Tech. rep., IEEE, 2001.

[19] Italiano, G., Rastogi, R., and Yener, B. Restoration algorithms for virtual private networks in the hose model. In *Procedings of IEEE INFOCOM 2002* (June 2002).

[20] Jannotti, J., Gifford, D., Johnson, K., Kaashoek, M., and Jr., J. O. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI 2000* (October 2000).

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

[22] Jiang, X., and Xu, D. Violin: Virtual internetworking on overlay infrastructure. Tech. Rep. CSD TR 03-027, Department of Computer Sciences, Purdue University, July 2003.

[23] Kwong, K., and Ishfaq, A. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing 59*, 3 (1999), 381–422.

[24] Leighton, T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[25] Mambretti, J., Weinberger, J., Chen, J., Bacon, E., Yeh, F., Lillethun, D., Grossman, B., Gu, Y., and Mazzuco, M. The photonic terastream: Enabling next generation applications through intelligent optical networking at igrid2002. *Future Generation Computer Systems 19*, 6 (August 2003), 897–908.

[26] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *Prodeedings of USENIX 1993* (1993), pp. 259–270.

[27] Microsoft Corporation. Virtual server beta release.

[28] Nichols, K., Blake, S., Baker, F., and Black, D. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. Tech. rep., Internet Engineering Task Force, 1998.

[29] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. Virtual network computing. *IEEE Internet Computing 2*, 1 (January/February 1998).

[30] Sapuntzakis, C., Brumley, D., Chandra, R., Zeldovich, N., Chow, J., Lam, M. S., and Rosenblum, M. Virtual appliances for deploying and maintaining software. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA 2003)* (October 2003).

[31] Sapuntzakis, C. P., Chandra, R., Pfaff, B., Chow, J., Lam, M. S., and Rosenblum:, M. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)* (December 2002).

[32] Shi, S., and Turner, J. Routing in Overlay Multicast Networks. In *Proceedings of IEEE INFOCOM 2002* (June 2002).

[33] Sugerman, J., Venkitachalan, G., and Lim, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).

[34] Sundararaj, A. I., and Duchamp, D. Analytical characterization of the throughput of a split tcp connection. Tech. rep., Department of Computer Science, Stevens Institute of Technology, 2003.

[35] Townsley, W., Valencia, A., Rubens, A., Pall, G., Zorn, G., and Palter, B. Layer two tunneling protocol “l2tp”. Tech. Rep. RFC 2661, Internet Engineering Task Force, August 1999.

[36] Virtuozzo Corporation. http://www.swsoft.com.

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

[38] Whitaker, A., Shaw, M., and Gribble, S. Scale and performance in the denali isolation kernel. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)* (December 2002).

[39] Wolski, R., Spring, N., and Hayes, J. Predicting the CPU availability of time-shared unix systems. In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99* (August 1999), IEEE, pp. 105–112. Earlier version available as UCSD Technical Report Number CS98-602.

[40] Wolski, R., Spring, N. T., and Hayes, J. The network weather service: A distributed resource performance forecasting system. *Journal of Future Generation Computing Systems* (1999). To appear. A version is also available as UC-San Diego technical report number TR-CS98-599.