Creating Private Network Overlays for High Performance Scientific Computing

Edward Walker[1]

[1] Texas Advanced Computing Center, The University of Texas at Austin, Austin, Texas 78758, USA
ewalker@tacc.utexas.edu

Abstract. In this paper we describe an approach in creating private network overlays in user-space to support the dynamic creation of personal clusters on-demand. These personal clusters are created by submitting job proxies to High Performance Computing (HPC) clusters. Job proxies contribute CPU resources back to the personal cluster when they eventually run, allowing application jobs to execute on them in a system call virtualized run-time environment. The virtualized run-time environment enables additional personal cluster-wide services to be interposed, including a private network overlay instantiated for each personal cluster created. The interposed private network overlay allows the personal clusters to tunnel IP traffic thorough gateway nodes at each contributing HPC cluster site in order to provision resources across private networks, survive transient network outages, support critical services like distributed filesystems, and in some cases, improve network transfer throughput across the wide-area network. This paper describes our design and implementation strategy, and concludes with some general guiding principles to aid other projects of a similar nature.

Keywords: High performance computing, resource management, cluster computing, overlay networks.

1 Introduction

We describe our experience in implementing a user-space private network overlay across High Performance Computing (HPC) clusters to support a TeraGrid production software system called MyCluster [1][2].

TeraGrid is a multi-year, multi-million dollar, NSF funded project to build the world's largest HPC cyberinfrastructure for open scientific research [3]. The project currently links nine resource provider sites across the continental United States, providing in aggregate over 200 teraflops of compute resource and four petabytes of online disk storage. Resources on the TeraGrid include HPC clusters, visualization clusters, and online data collections, accessible through a 30 Gbps wide-area network (WAN) backbone.

MyCluster is a system for provisioning resources from distributed HPC sites into personal clusters created on-demand. These personal clusters can be created on a per-

user, per-experiment, basis, allowing them to be used as job containers for experiments conducted within a personalized computing laboratory. In 2006, the system was used to acquire over 800,000 CPUs for researchers on the TeraGrid to support computational experiments across a broad range of scientific disciplines [5][6].

The MyCluster system provisions resources for building personal clusters by deploying semi-autonomous agents at each HPC site. These semi-autonomous agents, reacting to local and global load conditions, submit and manage job proxies through the local scheduler at each HPC site. Job proxies then contribute CPUs back to the personal clusters when they are run by the local scheduler. Job proxies also allow application jobs, submitted into the personal cluster, to execute in a system call virtualized run-time environment where additional cluster-wide services are interposed in user-space.

MyCluster enables users to select a commodity job management system to aggregate the provisioned CPU resources into their personal clusters. Job management systems that are supported, in various stages of prototype to production form, include Condor [7][8], Sun Grid Engine (SGE) [10] and OpenPBS [9]. Users therefore benefit from using a single, well-known interface to interact with their jobs across the heterogeneous clusters on the TeraGrid. Users also benefit from the ability to reuse the plethora of tools that have been developed for these systems over the years.

Finally, MyCluster is a completely user-space system, with no requirement for administrator privilege for deployment. This choice allows the system deployment model to easily scale, allowing it to aggregate any HPC cluster accessible across the internet. A user can simply invoke a self-installer at a site, boot-strap a personal version of the system, and enable the provisioning of resources from that site for computational experiments.

Within the context of the MyCluster project, we have been developing a system for building private network overlays across HPC clusters to enable the seamless creation of personal clusters. Like MyCluster, the system is deployable in user-space, without administrator privilege. It allows the creation of a class B network, enabling compute nodes in internal networks to acquire a virtual IP address, advertise it, and exchange messages between each other using the standard BSD network socket interface. Programs do not have to be recompiled to use the network overlay. Our system transparently tunnels IP (TCP and UDP) traffic through a series of packet relays in the overlay to deliver message packets between addresses in the private network.

The network overlay instills some critical properties to the MyCluster system. First, the network overlay enables MyCluster to provision resources from HPC clusters with compute nodes in internal networks, allowing its deployment on most HPC cluster configurations. Second, the network overlay allows MyCluster to survive transient WAN outages, providing increased quality of service guarantees for long running experiments executing in the personal clusters. Third, the network overlay allows the WAN distributed filesystem XUFS [4] to be deployed within the personal cluster. This allows jobs running in the personal cluster to transparently access files from the submission directory across remote sites, emulating the shared filesystem in a real cluster. Fourth, in some cases, the network overlay improves

transfer throughput across the WAN, enabling more efficient bulk data transfers across remote sites.

The rest of this paper will be as follows. Section 2 compares our work against other similar projects described in the literature. Section 3 describes the design and implementation of our network overlay in detail. In particular, it describes the mechanisms used to transparently interpose our overlay behavior into unmodified applications, the algorithm used to provide fault-tolerance to applications using our overlay, an *optional* kernel patch to improve the potential performance of one version of our system, and a brief overview of how a distributed filesystem is enabled by our overlay. Section 4 describes the experimental evaluation of our network overlay on the local area network (LAN) and on the TeraGrid WAN. It also describes some apparent anomalies in our experimental results, and determines their cause. The section also examines how much overhead is introduced to HPC applications running in our virtualized run-time environment. Finally, section 5 derives some important guiding principles from our experience, and concludes this paper.

2 RELATED WORK

MyCluster creates personal clusters using resources across the WAN. It is therefore related to the projects like Cluster-On-Demand (COD) [12][13], VioClusters [14], In-VIRGO [15], WOW [16] and Virtual Workspaces [17]. In particular, VioClusters, InVirgo and WOW also create network overlays using the technologies VIOLIN [26], Virtuoso [24][25], and IPOP [27] respectively.

At the risk of over simplifying, these systems adopt the same basic approach of instantiating a system virtual machine with its network device bridged to a TAP device, configured in promiscuous mode, on the host computer. Marshalling processes on the host computer then forward Ethernet frames from the TAP device, originating from the system virtual machine, to the external network as UDP packets or, more interestingly, as packets to a P2P overlay like Brunet [29]. Conversely, un-marshalling processes on the host computer forward Ethernet frames it receives from the external network back to the TAP device into the system virtual machine.

The use of system virtual machines provides many useful properties to these projects. In particular, system virtual machines ensure resource, fault and security isolation between applications running on the same server. These isolation properties are important in IT hosting environments and infrastructures like PlanetLab [28] where multiple applications may be consolidated on the same server hardware. System virtual machines also offer the opportunity for users to ensure a correct operating system environment for their running jobs. This guarantees some level of quality of service, in terms of expected operating system environment, when jobs run. Our system however does not use system virtual machines for a number of reasons.

The first reason why we do not use system virtual machines is because the isolation properties offered by them are not as compelling a reason for their use in HPC clusters. This is because running jobs are already naturally resource isolated by the job schedulers on HPC clusters. These job schedulers fairly allocate time and space

on the cluster nodes to jobs requesting them. Thus, for the period of time when the job runs, no other users are consuming resources on the same nodes. Also, HPC jobs are user-level processes that do not require kernel modifications, i.e. the installation of kernel modules. Unlike IT hosting environments where applications with special kernel module requirements may cause crashes and affect other running applications on the same server, HPC jobs run on dedicated compute nodes in isolation from other jobs in the cluster. Thus, a fault in a HPC job does not affect other running jobs in a cluster. Finally, the compute nodes in many HPC clusters are within private networks, completely isolated from the corporate and external network. Also, these nodes are re-imaged on a weekly or bi-weekly basis. Thus compute nodes already offer good security isolation.

The second reason why we do not use system virtual machines is because of the finely tuned execution environments in HPC clusters. HPC clusters have many software packages that are compiled and configured by HPC specialist to run well at the site. Also, these HPC clusters have operating system environments that are configured to reduce phenomena like operating system jitter [11], and to function well with internal components like the high-speed interconnect and parallel filesystem. For example, the IBM GPFS [30] and Lustre [31] filesystems on many HPC clusters work with only a small subset of Linux kernel versions which need to be further patched with vendor-specific modifications. These site-specific requirements severely restrict the choice of operating systems that can be instantiated by the user anyway.

The third reason why we do not use system virtual machines is because they often require the pre-installation of administrator-level components like virtual machine monitors, and/or hypervisors. Even with type II virtual machine monitors [18] (e.g. VMWare Workstation and User-Mode-Linux) setting up external networking for these virtual machines require bridge devices like TUN/TAP to be set up and configured in promiscuous mode. These actions require administrator permission, limiting the broad applicability of the approach only to sites that have agreed to deploy the appropriate configuration.

The fourth reason why we do not use system virtual machines is because each virtual machine instance requires a root filesystem image to boot. Root filesystem images are typically at least 500 megabytes in size, and this needs to be distributed across the WAN and replicated for every virtual machine instance created. For large HPC computational runs, there could be many thousands of virtual machines in use, requiring the replication and management of multiple gigabytes of image files.

Our network overlay project is also related to other tools that support IP traffic tunneling between private networks and the WAN. These tools include SOCKS [19], GCB [22], OpenSSH, OpenVPN [20] and PPTP [21]. All these tools have at least one deficiency which prevented their use in our system. SOCKS does not allow connections from an external network to a node in a private address space, while GCB assumes that the node in the private address space has at least outbound external network connectivity in order to operate correctly. In many cluster configurations, compute nodes have no external network connectivity in any direction. Also, OpenSSH, OpenVPN and PPTP represent point-to-point solutions, i.e. from a client to a gateway node. They do not easily cater for the scenario our network overlay supports, i.e. multiple nodes behind multiple gateways joining a private network

infrastructure, without requiring extensive scripting and additional coding effort. Also, all the above technologies do not assign virtual IP addresses to the compute nodes in the private network. This is needed to prevent addressing conflicts for nodes from multiple internal networks.

Realm-Specific IP (RSIP) [23] is an experimental IETF proposal that is very similar to our network overlay solution. RSIP allows nodes in a private address space to register and temporarily lease a public IP address from a RSIP gateway. These nodes can then advertise their addresses and have external connections to them relayed through the RSIP gateway. There are however a number of problems with RSIP. First RSIP it is not widely implemented. It is intended as a replacement for NAT, but this has not occurred. Second, RSIP leases public IP addresses to the nodes in the private address space. This approach limits the number of available addresses that can be used.

3 DESIGN AND IMPLEMENTATION

3.1 MyCluster Overview

MyCluster builds Condor, SGE or OpenPBS clusters when a user creates a *virtual login session*. Within this virtual login session, users can submit, monitor and manage jobs through a single job management interface, emulating the experience of a traditional cluster login session. Fig 1 shows an example of a SGE virtual login session.

```
ewalker@blanco ~]$ vo-login -S
Enter GRID pass phrase:
Spawning on tg-login.tacc.teragrid.org
Spawning on tg-login3.ncsa.teragrid.org
Setting up VO participants ......Done

Welcome to your MyCluster/Sun Grid Engine environment
To shutdown environment, type "exit"
To detach from environment, type "detach"

blanco(grid)% agent_jobs
GATEWAY: lonestar.tacc.utexas.edu
    753018 (RUNNING)
GATEWAY: tg-login3.ncsa.teragrid.org
    526521.tg-master.ncsa.teragrid.org (RUNNING)
blanco(grid)% qhost
HOSTNAME                ARCH         NPROC  LOAD   MEMTOT   MEMUSE   SWAPTO   SWAPUS
-------------------------------------------------------------------------------
global                  -               -     -        -        -        -        -
mc1-14                  glinux          2  0.03     2.0G    79.1M     2.0G      0.0
mc1-9                   glinux          2  0.00     2.0G    92.6M     2.0G      0.0
tg-c491                 ia64linux       2  0.65     3.9G   383.3M     5.9G    23.5M
tg-c861                 ia64linux       2  0.00     3.9G   404.7M     6.0G    16.5M

blanco(grid)% qsub -t 1000 sub.cmd
your job-array 1.1000-1000:1 ("sub.cmd") has been submitted
blanco(grid)% qdel 1
ewalker has registered the job-array task 1.1000 for deletion
```

Fig 1. Formatted snapshot of a SGE virtual login session.

A high level overview of the MyCluster processes relevant to the discussion in our paper is shown in Fig 2. When a user first starts a virtual login session, the system remotely spawns a *proxy manager* at the head node of each of the clusters contributing resources to the session. These proxy managers submit and manager *job*

proxies to the local scheduler at the site. When the local scheduler runs the job proxy, it starts the job starter daemon for the job management system selected for the session, i.e. Condor, SGE or OpenPBS. The job starter daemon then registers back to the master processes at the job submission host across the WAN. Jobs submitted to the personal cluster can then be dispatched to the newly registered job starter, with the user seeing an expanding and shrinking cluster as these job starter daemons register and terminate over time.

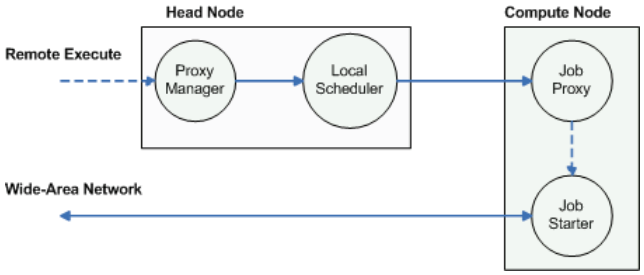


Fig 2. MyCluster process architecture overview

3.2 Private Network Overlay Architecture

The MyCluster system requires the provisioned compute nodes in a virtual login session to have full bi-directional access to the external network. This is to allow the job starter to register and accept jobs from the job management master processes in the personal cluster. Many of the TeraGrid clusters have compute nodes with full network connectivity to the TeraGrid WAN, and thus MyCluster is easily supported on these systems.

However, to enable MyCluster to be deployed on HPC clusters with the more traditional configuration where compute node are within a private network, a network overlay has been implemented to enable external network connectivity for these job starters.

The network overlay we have implemented allows a private class B network to be deployed, instantiated, and destroyed by user–level processes, requiring no administrator privilege, pre-installed virtual machine or outbound WAN connectivity for the compute nodes. Using interposition methods, our solution overrides the socket system calls in the application process to allow connections between privately assigned class B IP addresses to be tunneled through a series of subnet routers. These subnet routers run as user-level processes and are strategically located at the head (or some gateway) node of the clusters with access to the external network.

Subnet routers initialize their internal route tables by reading a route table file `.uvpn_route_<id>` created for each virtual network instance. This file specifies, for a particular session, the subnet to which the host cluster belongs and the contact addresses for routers of other subnets in the network overlay.

Fig 3 shows the augmented MyCluster process architecture with the network overlay support for routing IP traffic between compute nodes provisioned by a virtual

login session. When a user starts a virtual login session, each host cluster is allocated a subnet in a virtual class B network and a subnet router is spawned at the head node. Each node provisioned by a job proxy is then assigned a virtual IP address in the subnet, with the home router (at the cluster head node) keeping a database of real to virtual IP address mapping.

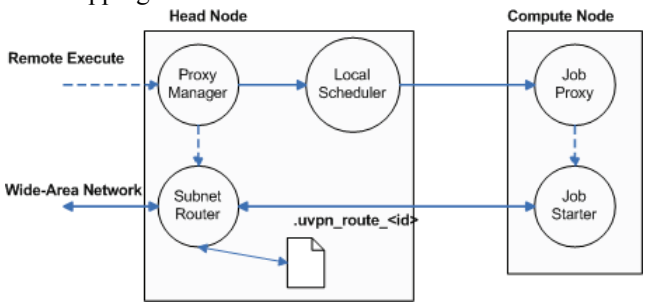


Fig 3. MyCluster process architecture with a network overlay

When the application process in a job proxy, i.e. the job starter, invokes the `connect()` system call to an address within the private network, a series of connections are made to establish an IP packet relay path between the source and destination addresses. First, the interposed `connect()` makes a connection to the home router and sends a connection header containing the source and destination address end-points associated with the desired virtual connection. The format of the connection header is shown in Fig 4.

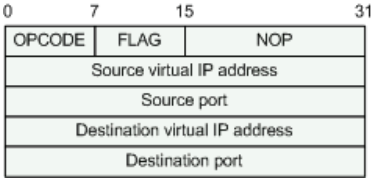


Fig 4. Network overlay connection header

The home router then searches its route table for the location of the peer router responsible for the subnet associated with the virtual destination address. A connection is then made to the peer router and the connection header forwarded. The peer router, on receiving the connection header, looks up the real IP address associated with the virtual destination address, and makes a connection to this real address. The connection header is then forwarded to the destination process, which then stores the information contained in the connection header for future reference.

When the relay path is finally created from the home router, through the peer router, to the real IP address, the `connect()` call returns the socket descriptor for this relay. Equivalently, the `accept()` call, which is also interposed at the destination, returns the socket descriptor for the other end of this relay. This relay connection simulates a dedicated leased-line, allowing the processes at both ends to issue `send` (or `write`) and `recv` (or `read`) calls to exchange bytes between them. If a process decides to query information associated with the socket connection, we

interpose the `getsockname()` and `getpeername()` calls to return the information associated with the virtual connection, instead of information associated with the real connection to the home routers.

3.3 Interposition Mechanisms

The system supports two mechanisms for interposing our network overlay functionality into the BSD socket call interface. The first uses the UNIX shared object preloading mechanism available in most UNIX variants. For example, on Linux this involves defining the location of a shared object in the LD_PRELOAD environment variable. This shared object will then be used by the linker to override the system shared library implementation of the socket interface, allowing our overlay behavior to be interposed.

Table 1. Interposed socket system calls

PRELOAD	PTRACE
Connection-less based protocols	
`sendto, recvfrom`	`sendto, recvfrom`
Connection based protocols	
`connect, accept`	`connect, accept`
Shadow socket management	
	`socket, listen, bind, setsockopt`
Connection information	
`getsockname, getpeername`	`getsockname, getpeername`
Connection termination	
`close`	`close`
Shadow socket duplicate tracing	
	`dup, dup2, fctnl, fork`

The shared library preloading mechanism however only works on dynamic linked executables. Furthermore, some UNIX variants, in particular AIX, do not support the preloading mechanism. To remedy this, we provide an alternative mechanism to interpose our overlay functionality. This alternative mechanism uses the UNIX ptrace debugging interface. The ptrace interface allows a parent process to monitor the execution of its children processes, allowing system calls in the children processes to be traced and modified by manipulating the CPU architecture registers prior to and after their invocation.

The socket calls interposed by the two mechanisms are shown in Table 1. The preloading mechanism interposes the `connect()` and `accept()` calls to support

the creation of the virtual connection relay for connection based protocols. For connection-less protocols, the `sendto()` and `recvfrom()` calls are also interposed. The `getsockname()` and `getpeername()` calls are interposed to return the correct information about the virtual connection as explained before. Finally, the `close()` system call is interposed to allow the managed termination of virtual connections.

The preloading mechanism allows our system to directly query and manipulate socket descriptors in the interposed application. Interposing the overlay functionality using the ptrace mechanism however requires additional effort. The ptrace monitoring process runs in a separate process image, making querying and manipulating socket descriptors in the monitored application process difficult.

To overcome this difficulty, our strategy for the ptrace mechanism is to implement the overlay functionality in the parent monitoring process itself. Overlay connections are established between shadow sockets in the ptrace monitoring processes that exactly mirror the sockets created in the application processes. Fig 5 illustrates the basic idea.

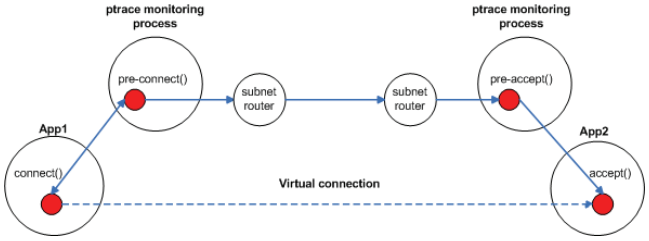


Fig 5. Shadow sockets negotiate relay connection between ptrace parent processes

For every successful `socket()` invocation in the interposed application process, the ptrace monitoring process also creates an equivalent shadow socket. Subsequent calls to `bind()`, `listen()`, and `setsockopt()` on a socket in the application process causes the same action to be performed on its associated shadow socket in the ptrace monitoring process.

To avoid clashes in the local network port namespace, the application requested port number for the `bind()` call on the original socket is replaced with a free port in the range [51000, 52000]. Only the shadow socket is allowed to bind to the application requested port number instead. Note in particular that the shadow socket at the receiving peer is now set to listen to the application requested port instead of the original socket. To avoid confusing the user application, the `getsockname()` call is interposed to return the application requested port for the original socket when it is invoked.

When a TCP connection is initiated in a network overlay using the ptrace mechanism, the connecting application process is redirected to connect to a dynamically created relay thread in the monitoring parent. The shadow socket that is associated with the original socket then creates the virtual connection relay to the monitoring parent at the receiving peer as described before. The monitoring parent at the receiver peer then connects to the socket in the application process which is listening on the alternative port we had previously assigned it to.

Messages to/from the virtual connection are then routed through the additional relay thread in the monitoring parent to/from the subnet routers. The information contained in the connection header for the virtual connection is maintained in the monitoring parent at either end-point, and the `getpeername()` and `getsockname()` calls in the application are interposed to return this information as necessary.

3.3.1 Tracing duplicate socket descriptors

In order to properly terminate virtual connections, the ptrace mechanism also needs to closely trace socket descriptors that are duplicated within and across processes. To do this, our network overlay system also traces the `dup()`, `dup2()`, `fcntl()` and `fork()` system call. For the `fork()` call in particular, our system will check if any shadow sockets has the FD_CLOEXEC flag set. If it is not set, the socket descriptor is expected to be duplicated in the new process.

When a socket descriptor is duplicated, we increment a reference count to a structure record we maintain for each shadow socket. Subsequent `close()` calls on the duplicated socket descriptor decrements this shared reference count. When the reference count reaches 0, the shadow socket is then closed, and any associated virtual connection terminated.

3.4 Tolerating WAN Outages

An important benefit of creating our network overlay is the WAN fault tolerant properties it instills to the end-points in the virtual connection. The IP packet relay created by our network overlay effectively isolates the connected application processes at either end-points from the less reliable WAN. When a network outage causes the connection between peer subnet routers to be temporarily disconnected, this disconnection is not propagated to the end-points.

To prevent the lose of in-flight messages during a WAN outage, Fig 6 details the WAN recovery algorithm used in our network overlay. Messages that are sent from the application to the subnet router are immediately forwarded to the destination socket connected to a peer subnet router in the connection relay. If the forwarding is successful, a count `s_bytes` is incremented with the number of bytes sent, while at the destination peer subnet router, the count `r_bytes` is incremented with the number of bytes received. The message itself is also appended to a circular buffer of sent messages. The size of this circular buffer is equal to the size of the socket's internal send buffer size, i.e. `SO_SNDBUF`.

If the subnet router fails to forward an application message, the message is appended to an unsent message file and the subnet router periodically (every 5 minutes) attempts to reconnect with the peer subnet router in the connection relay.

When the connection with the peer subnet router is re-established, a hand-shake is performed to ensure recovery of any lost in-flight data. First, the peer router sends its `r_bytes` count to the reconnecting router. Second, the reconnecting router compares this against its `s_bytes` count. Third, if the `s_bytes` value is larger then the received `r_bytes` value, the reconnecting router sends `s_bytes-`

`r_bytes` bytes from the circular buffer of sent messages. Fourth, the reconnecting router then forwards the content of the unsent message file to the peer router, after which the connection relay resets back to its original fault-free state.

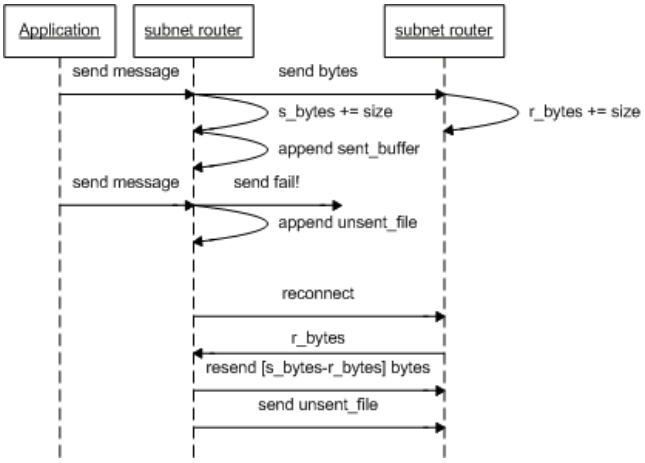


Fig 6. WAN outage recovery algorithm

3.5 Personal Global File Namespaces

The network overlay allows compute nodes provisioned in the virtual login session to communicate with the submission workstation from which the personal cluster is created. An important collateral benefit is that this provides the ability for compute nodes, provisioned from an internal private network, to mount the XUFS distributed filesystem enabling jobs running on them to access files from the submission directory. XUFS allows the submission directory to be mounted in user-space and, like our network overlay, can be deployed, instantiated and destroyed without administrator privilege. Also, similar to our network overlay solution, XUFS uses interposition mechanism to allow this transparent remote access to files and directories. More details about XUFS can be found in our prior publications [3].

3.6 "ptrace is slow"

In adopting the ptrace debugging mechanism as one method of imposing our network overlay functional into the socket interface, we have often encountered the comment "ptrace is slow". This is usually accompanied by anecdotal stories supporting the claim. Later in section 4.4, we examine the overhead of running a collection of HPC benchmarks representing different workload types in our ptrace interposed environment. We will see later that for many HPC workloads this assertion is not necessarily true.

In this section we look at the degenerate case where a program's execution time is dominated by many repeated system calls. An example of such a degenerate case is a program whose only task is reading and writing a very large file using very small read/write message buffers. Later we show in section 4.1 an example of such a degenerate case in one of our scenarios in the experimental evaluation of the TCP throughput of our network overlay solution

In these degenerate cases, the `ptrace` mechanism is expected to introduce large overheads. This is because the ptrace mechanism causes the operating system to stop the application process every time a system call is invoked. This allows the monitoring parent process to examine and modify the execution of the application as necessary. Furthermore, this stop-start behavior occurs twice for every system calls invoked in the application process; once prior to a system call invocation and once after it has been completed by the operating system.

We have implemented an optimization to the ptrace mechanism in Linux to allow the monitoring process to selectively decide what system calls are of interest to it. For example, our network overlay mechanism is only interested in a subset of the socket system calls. Often repeated system calls like `send()`, `write()`, `recv()`, and `read()` do not need to be interposed by our system. This design choice is deliberate to ensure we introduce as little overhead to the original application as possible.

The optimization we have implemented in Linux introduces a new PTRACE_SYSCALL_MASK option to the ptrace system call. We allow the monitoring process to use a bit-mask data structure to selectively set the bits associated with system calls of interest to it. The monitoring process then uses this bit-mask as the input parameter to the `ptrace()` system call when the PTRACE_SYSCALL_MASK option is used. The ptrace mechanism then only stops the application processes when a system call defined in this bit-mask is invoked. A code fragment illustrating how this option is used by a monitoring process who is only interested in the `open()` system call is shown below:

```
scall_set syscall_mask;

SC_ZERO(&syscall_mask);
SC_SET(__NR_open, &syscall_mask);
ptrace(PTRACE_SYSCALL_MASK,pid,&syscall_mask,__NR_open+1);
```

We have implemented this ptrace enhancement in the Linux kernel version 2.6.16 [32]. We show in our experimental evaluation section later that this improves the performance in all our scenarios when comparing the TCP connection throughput performance in our network overlay against the native socket connection throughput. We are encouraged by this, and also by the fact that this option is already under discussion by the mainline Linux development community, albeit in a different implementation version [33].

4 EXPERIMENTAL EVALUATION

4.1 Local area network TCP throughput evaluation

In this section, we describe results from experiments comparing the TCP transfer throughput of a native connection versus a connection through the proposed network overlay on a local area network (LAN). The experiments were conducted between two Linux 2.6.16 X86_64 hosts connected through a 100 Mbs switch. Each host was designated a subnet, with a subnet router running on each. For all experiments, the TCP throughput was measured using NETPERF [34]. Fig 7 illustrates the LAN experiment setup for the network overlay.

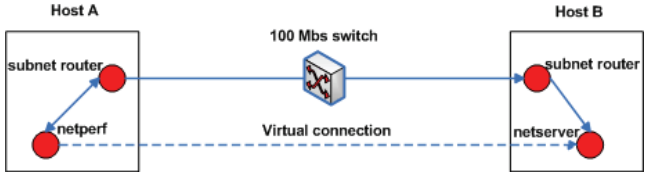


Fig 7. LAN experiment setup for the network overlay.

Table 2 shows the TCP throughput on the LAN using a native TCP socket connection versus a connection made through the network overlay using the preloading mechanism. The network overlay connection shows no degradation in throughput performance compared to the native connection.

Table 2. TCP throughput (Mbs) of native connection versus connection through the overlay using the preload mechanism

Send size (bytes)	100	200	300	400	500	600	700	800	900	1000
Native	93.95	94.03	94.03	94.03	94.03	94.03	93.99	94.02	94.03	94.02
preload	94.2	94.25	94.3	94.27	94.31	94.28	94.28	94.3	94.29	94.3

Fig 8 shows the TCP throughput on the LAN using a native TCP socket connection versus a connection made through the network overlay using the ptrace mechanism. The experiment show the throughput performance degrading considerably for small send sizes. This experiment demonstrates the degenerate case expounded on in section 3.6. For small send sizes, many more `send()` system calls are invoked, causing the application process to be stopped much more frequently then when large send sizes are used.

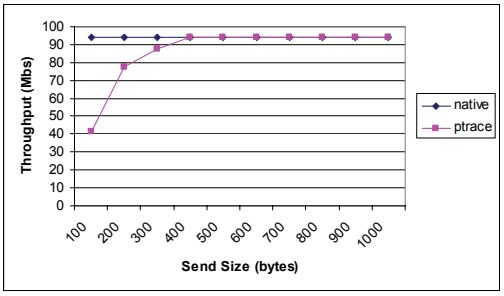


Fig 8. TCP throughput (Mbs) of native connection versus network overlay connection using the ptrace mechanism

Table 3 shows the TCP throughput on the LAN using a native socket connection versus a connection made through the network overlay using the ptrace mechanism with the PTRACE_SYSCALL_MASK enhancement enabled. We see that the network overlay connection shows no degradation in throughput.

Table 3. TCP throughput (Mbs) of native connection versus connection through the overlay using the ptrace mechanism (PTRACE_SYSCALL_MASK enhancement enabled)

Send size (bytes)	100	200	300	400	500	600	700	800	900	1000
Native	93.95	94.03	94.03	94.03	94.03	94.03	03.99	94.02	94.03	94.02
Ptrace+enh	94.29	94.28	94.3	94.28	94.28	94.31	94.3	94.3	94.29	04.3

4.2 Wide area network TCP throughput evaluation

In this section, we describe results from experiments comparing the TeraGrid WAN TCP transfer throughput of a native connection versus that of a connection through our network overlay. The experiments were conducted between an SDSC cluster compute-node and the NCSA cluster head node, with the network overlay subnet routers deployed at the cluster head nodes. All IP traffic was tunneled through the firewall at each site using one of the free ports in the range [50000,51000] open for traffic between the two sites. For all experiments, the TCP throughput was measured using NETPERF. Fig 9 illustrates the experimental setup. The PTRACE_SYSCALL_MASK ptrace option was not tested in the experiments in this section because we did not have to the opportunity to patch the running kernels at these sites.

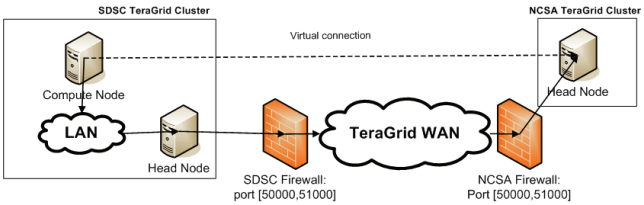


Fig 9. TeraGrid WAN experimental setup for the network overlay.

Fig 10 shows the TCP throughput on the TeraGrid WAN using a native connection versus connections through the network overlay using the different interposition mechanisms. The TCP throughput through the network overlay show no perceptible overhead, except for the degenerate case for small send sizes when the ptrace interposition mechanism is used. The throughput in the network overlay using the ptrace mechanism shows a 21% degradation in performance for the 100 byte send size scenario. However, we note that because of the lower bandwidth of the WAN, this degradation is not as pronounced as that observed in the LAN experiments.

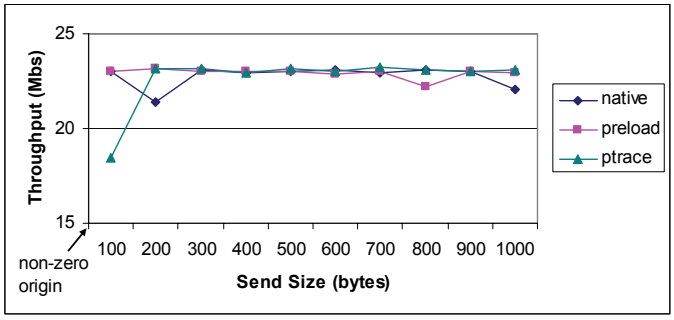


Fig 10. TCP throughput of native connection versus network overlay connections between NCSA and SDSC.

4.3 TCP throughput anomaly

An experiment was conducted on the TeraGrid WAN between TACC and NCSA with the setup similar to Fig 9, except with the SDSC cluster replaced with the TACC cluster. Fig 11 shows the TCP throughput of a native connection and a network overlay connection between a TACC cluster compute-node and the NCSA cluster head node. The network overlay configuration tunnels the TCP connection through a relay between subnet routers located at the TACC and NCSA cluster head nodes. Surprisingly, the results show a 500% improvement in the TCP throughput using the network overlay compared to the native TCP socket connection.

After some investigation, this apparent anomaly was explained by the different network device configurations at the compute and head nodes on the TACC cluster. The TACC cluster compute nodes had their network device MTU (Maximum Transmission Unit) set to the default 1500. This MTU value is optimized for the LAN rather than the WAN, because the network device was also used for mounting the internal NFS (network file system) home directories on the compute node.

However, the TACC cluster head node had its network device MTU set to 9000, optimized for sending jumbo packets across the WAN. Therefore, rerouting IP traffic through the head node improved the TCP throughput performance across the WAN significantly.

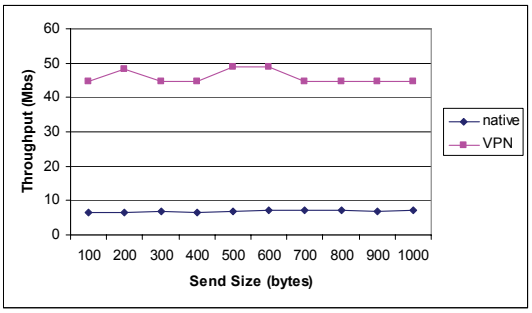


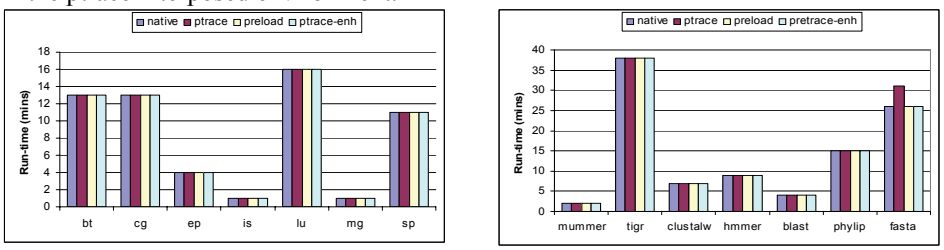
Fig 11. TCP throughput of native connection versus network overlay connection between TACC and NCSA.

4.4 Execution overhead

In this section, we investigate the overhead introduced when HPC applications are executed in the system call virtualized environment where the network overlay and distributed filesystem (XUFS) services are interposed.

Fig 12 (a) and (b) show the run-time of the applications in the NAS [35] and BioBench [36] benchmarks respectively. We ran the benchmarks in our virtualized environment and compared the run-times when executed natively. All benchmarks were run on a Linux 2.6.16 kernel X86_64 host with one gigabyte of memory.

Most of the benchmark applications show no perceptible overhead when executed in our system call virtualized run-time environment. Only the FASTA application in the BioBench benchmark exhibited a 19% degradation in performance when executed in the ptrace interposed environment.



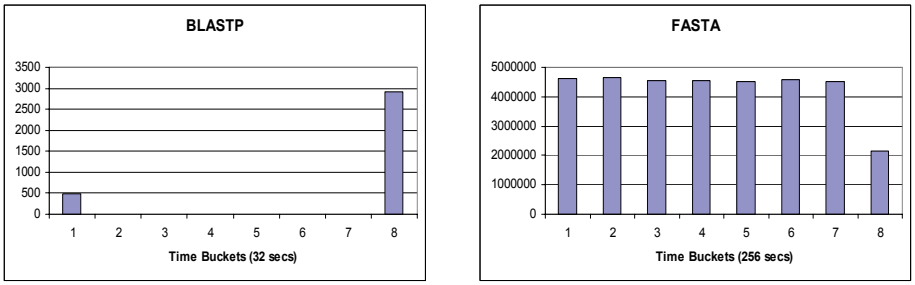
(a) (b)

Fig 12. Run-times of NAS (a) and BioBench (b) benchmarks in the system call virtualized run time environments

All the benchmark applications, except FASTA, have system call profiles similar to the BLASTP system call invocation histogram shown in Fig 13 (a). The BLASTP profile illustrates the read, compute, and write phases common to most HPC

applications. The profile also shows that the majority of the run time is dominated by the compute cycle.

In contrast, for FASTA we see a more evenly spread system call invocation profile across the entire run-time of the application, as shown in Fig 13 (b). But because of the compute bound nature of FASTA, this mitigates the overhead introduced by the ptrace mechanism to only 19% in our case.



(a) (b)

Fig 13. Histogram of system call frequency for BLASTP (a) and FASTA (b)

5 CONCLUSIONS

We have described our mechanism for providing a network overlay to support the creation of personal clusters in the MyCluster system. The system is unique in providing a completely user-space solution, requiring no pre-installation of virtual machine monitors or hypervisors. Furthermore, our solution provides additional fault-tolerance to application processes communicating over the WAN and throughput benefits in certain deployment scenarios.

Some general principles can be derived from our experience to help guide other projects building similar widely distributed system call virtualized run-time environments.

First, user-space interposition mechanisms do not provide the properties of resource, fault, and security isolation, but they are highly appropriate for augmenting the properties of the native system to enable more productivity for the user. Consider if the required level of isolation guarantees is already satisfied by the system, and if needed, consider the range of alternative mechanisms for providing this, such as QoS schedulers [38], kernel-level interposition techniques [39][40] or full system virtual machines. For example, MyCluster currently provides isolation properties through personal cluster containers with HPC cluster QoS schedulers assigning dedicated resources for each instance.

Second, shared object preloading is a very efficient user-space interposition mechanism and should be used where possible to implement overlay behavior. Where appropriate this can be used in conjunction with other techniques for providing isolation properties to the system.

Third, the ptrace debugging interface is an acceptable mechanism to interpose overlay behavior for compute-bound applications. For non compute-bound applications which frequently invoke system calls, high execution overheads can be expected. However, new ptrace system call options like PTRACE_SYSCALL_MASK can be implemented to mitigate this.

Fourth, WAN fault-tolerant properties can be transparently added into network applications by isolating the connection end-points from the WAN in an overlay. This can be used to ensure legacy applications, originally developed for the LAN, are able to survive transient network outages in the less reliable WAN.

Finally, network bulk transfer throughput across a WAN can be improved by routing network connections through WAN optimized intermediaries in an overlay. The cost of implementing additional connection hops can sometimes be more then compensated by the gain in transfer throughput.

References

[1] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Personal Adaptive Clusters as Containers for Scientific Jobs", *Cluster Computing*, vol. 10(3), September 2007.

[2] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Creating Adaptive Clusters in User-Space for Managing Scientific Jobs in a Widely Distributed Environment", in *Proc. of IEEE Workshop on Challenges of Large Applications in Distributed Environments (CLADE'2006)*, Paris, July 2006.

[3] NSF TeraGrid, http://www.teragrid.org

[4] E. Walker, "A Distributed File System for a Wide-Area High Performance Computing Infrastructure", in *Proc. of the 3rd USENIX Workshop on Real, Large Distributed Systems (WORLDS'06)*, Seattle, Nov 2006.

[5] E. Walker and C. Guiang, "Challenges in Executing Large Parameter Sweep Studies Across Widely Distributed Computing Environments", in *Proc. of IEEE Workshop on Challenges of Large Applications in Distributed Environments (CLADE'2007)*, Monterrey, CA, June 2007

[6] E. Walker, "How to Run A Million Jobs in Six Months on the NSF TeraGrid", in *Proc. of TeraGrid'07*, Madison, WI, June 2007.

[7] Condor, High Throughput Computing Environment, http://www.cs.wisc.edu/Condor/

[8] M. Litzkow, M. Livny, and M. Matka. Condor – A Hunter of Idle Workstations, In *Proc. of the International Conference of Distributed Computing Systems*, pp. 104—111, June 1988.

[9] Portable Batch System, http://www.openpbs.org

[10] Sun Grid Engine, http://gridengine.sunsource.net/

[11] D. J. Kerbyson, S. Pakin, and F. Petrini, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q", in *Proc. of ACM/IEEE Conference on High Performance Networking and Computing (SC03)*, Phoenix, Arizona, Nov. 15-21, 2003.

[12] J. Chase, L. Grit, D. Irwin, J. Moore and S. Sprenkle, "Dynamic Virtual Clusters in a Grid Site Manager", in *Proc. of the 12th Intl Symp on High Performance Distributed Computing (HPDC-12)*, 2003.

[13] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi and J. Chase, "Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control", in *Proc. Of the ACM/IEEE Conference on High Performance Networking and Computing (SC06)*, Tampa, FL, Nov. 2006.

[14] P. Ruth, P. McGachey, X. Jiang, and D. Xu, "VioCluster: Virtualization for Dynamic Computational Domains", in *Proc. of the IEEE International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, Sept. 2005.

[15] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu, "From Virtualized Resources to Virtual Computing Grids: The In-VIGO System", Future Generation Computer Systems, 2004.

[16] A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo, "WOW: Self-Organizing Wide Area Overlay Networks of Virtual Workstations", in *Proc. of the 15th IEEE Intl. Symp. On High Performance Distributed Computing (HPDC-15)*, Paris, 2006.

[17] K. Keahey, K. Doering, and I. Foster, "From Sandbox to Playground: Dynamic Virtual Environments in the Grid", in *Proc. of the 5th International Workshop in Grid Computing*, 2004.

[18] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973.

[19] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "SOCKS Protocol Version 5", IETF RFC 1928, March 1996.

[20] OpenVPN, http://www.openvpn.net

[21] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn, "Point-to-point Tunneling Protocol (PPTP)", IETF RFC 2637, July 1999.

[22] S. Son and M. Livny, "Recovering Internet Symmetry in Distributed Computing", in *Proc. of the 3rd Intl. Symp. On Cluster Computing and the Grid (CCGrid)*, Tokyo, Japan, May 2003.

[23] M. Borella, J. Lo, D. Grabelsky, G. Montenegro, "Realm Specific IP: Framework", IETF RFC 2102, July 2000.

[24] P. Dinda, A. Sundaraj, A. Gupta, "Dynamic Topology Adaption of Virtual Networks of Virtual Machines", in *Proc. of the 7th Workshop on Languages, Compilers and Run-time Support for Scalable Systems*, Oct. 2004.

[25] A. Sundararaj and P. Dinda, "Towards Virtual Networks for Virtual Machine Grid Computing", in *Proc. of the 3rd USENIX Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[26] X. Jiang and D. Xu, "Violin: Virtual Internetworking on Overlay Infrastructure", in *Proc. of the 2nd International Symposium Of Parallel and Distributed Processing and Applications*, Dec. 2004.

[27] A. Ganguly, A. Agrawal, P. Oscar Boykin, and R. Figueiredo, "IP Over P2P: Enabling Self-Configuring Virtual IP Networks for Grid Computing", in *Proc. of the 20th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 2006.

[28] PlanetLab, http://www.planet-lab.org/

[29] P. O. Boykin, J. Bridgewater, J. Kong, K. Lozev, B. Rezaei, and V. P. Roychowdhury. Brunet software library, http://brunet.ee.ucla.edu/brunet/

[30] F. Schmuck, and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", in *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST)*, Monterey, CA, 2002

[31] P. Schwan, "Lustre: Building a File System for 1,000-node Clusters", in *Proc. of Ottawa Linux Symposium*, 2003.

[32] PTRACE_SYSCALL_MASK Linux 2.6.16 patch, http://www.tacc.utexas.edu/~ewalker/syscall_mask.patch

[33] "Virtual Time" posting, LWN.net, http://lwn.net/Articles/179829/

[34] NETPERF, http://www.netperf.org/netperf/

[35] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks", RNR Technical Report, RNR-94-007, March 1994.

[36] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C. W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications", in *Proc. of the 2005 IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS 2005)*, Austin TX, March 2005.

[37] Globus Security Infrastructure, http://www.globus.org/toolkit/docs/4.0/security/

[38] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A Survey of QoS Architectures", *Multimedia Systems*, vol. 6, no. 3, May 1998.

[39] R. Davoli, M. Goldweber, and L. Gardenghi, "UMView: View-OS implemented as a System Call Virtual Machine", in *Poster at USENIX Operating System Design and Implementation (OSDI06)*, 2006, http://www.usenix.org/events/osdi06/posters/davoli.pdf

[40] S. Soltesz, H. Poltzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors", in *Proc. of 2nd ACM EuroSys Conference*, Lisbon, Portugal, March 2007.