Harnessing HPC resources for CMS jobs using a Virtual Private Network

Benjamin Tovar[1,*], *Brian* Bockelman[4,**], *Michael* Hildreth[2,***], *Kevin* Lannon[2,****], and *Douglas* Thain[1,†]

[1]Department of Computer Science and Engineering, University of Notre Dame, Indiana
[2]Department of Physics, University of Notre Dame, Indiana
[3]Mordridge Institute for Research, Madison, Wisconsin

Abstract. The processing needs for the High Luminosity (HL) upgrade for the LHC require the CMS collaboration to harness the computational power available on non-CMS resources, such as High-Performance Computing centers (HPCs). These sites often limit the external network connectivity of their computational nodes. In this paper we describe a strategy in which all network connections of CMS jobs inside a facility are routed to a single point of external network connectivity using a Virtual Private Network (VPN) server by creating virtual network interfaces in the computational nodes. We show that when the computational nodes and the host running the VPN server have the namespaces capability enabled, the setup can run entirely on user space with no other root permissions required. The VPN server host may be a privileged node inside the facility configured for outside network access, or an external service that the nodes are allowed to contact. When namespaces are not enabled at the client side, then the setup falls back to using a SOCKS server instead of virtual network interfaces. We demonstrate the strategy by executing CMS Monte Carlo production requests on opportunistic non-CMS resources at the University of Notre Dame. For these jobs, cvmfs support is tested via fusermount (cvmfsexec), and the native fuse module.

1 Introduction

The increase in data processing needs for CMS after the High Luminosity (HL) upgrade [1] are expected to be around 20 times the currently available computational power [2]. To meet this new requirements and projected changes in funding, the CMS collaboration will need to harness resources from High Performance Computing centers (HPCs) and national laboratories [2]. These resources are usually behind firewalls that restrict compute nodes to connect to the outside world. At a bare minimum, a compute node would need access to the job description to run, the needed software via cvmfs [3], and stage-in and out data via xrootd to successfully execute an unmodified CMS job.

*e-mail: btovar@nd.edu
**e-mail: bbockelman@morgridge
***e-mail: mhildret@nd.edu
****e-mail: klannon@nd.edu
†e-mail: dthain@nd.edu

To solve this connectivity problem, we work under three main constraints: 1) jobs must run on compute nodes without administrator privileges. As far as the HPC is concerned, the job is just another user job; 2) the CMS job description should remain unchanged; 3) network usage limits guarantees should be given and enforced.

In this paper we explore a solution that takes advantage of so-called *demilitarized zones* (DMZs) of these computing centers which provide an isolated interface to the outside network. In the DMZ, connection from the outside may be performed by vetted services outside the internal network, trusted third-parties, or by privileged hosts inside the internal network. In our solution, a Virtual Private Network (VPN) server is run in a host in the DMZ, and compute nodes route all of their network traffic through the VPN server.

At the compute node side, we use network namespaces [4] to create a virtual network interface to which all the traffic of the job is routed. A network namespace provides a user with a logical copy of the network stack, with its own routes, firewall rules, and network devices. Unprivileged namespaces are available since linux kernel version 4.18 (e.g. RHEL >= 7.8). If network namespaces are not available at the compute nodes, then the VPN client falls back to the solution proposed by [5], where the network calls are captured and interposed using using a preloaded library and a SOCKS server.

We evaluate the feasibility of our solution by execution Monte Carlo CMS production tasks, to which cvmfs is provided via the native fuse module, or as a fusermount from cvmfsexec [6]. We tested up-to 200 concurrent jobs, and for these jobs, which are not IO-bounded, our results show that there is not a noticeable hit in performance. We also directly measured the bandwidth obtained in average, and found that link saturation for more than 2 concurrent transfers occurs at similar values compared to not using our VPN approach. For single transfers, a slowdown of about %60 is observed, which is caused by using TCP over TCP in our VPN test configuration.

The recipes and instructions to construct the singularity containers that implement our solution are publicly accessible as github repository [7].

2 Architecture

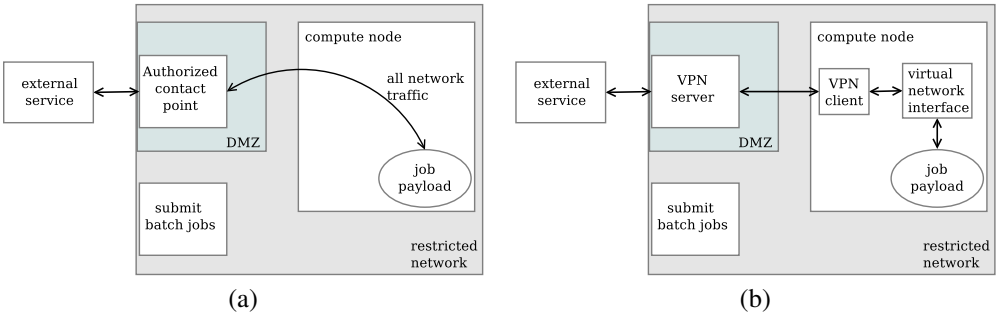

Figure 1. Redirecting network output to a trusted external service. (a) The general idea is to route all of the network traffic of a job's payload through a contact point of the DMZ. (b) A concrete implementation using a VPN server-client architecture with virtual network interfaces at the client side.

We show a general view of the architecture in Figure 1.(a). All the network traffic from the payload in a job running in a compute node is redirected through an authorized contact point in the DMZ. The network requests can then be fulfilled by trusted external services outside

the HPC restricted network. This is concretely implemented using a Virtual Private Network (VPN) server-architecture, as shown in Figure 1.(b). At the client side, a virtual network interface is created to which all traffic of the payload is redirected. This virtual network interface is configured by the VPN server with an IP address, default network routes, and name resolution. As far as the payload is concerned, it is running on a host with the IP given by the VPN server, not by original IP of the compute node.

2.1 Uses for CMS

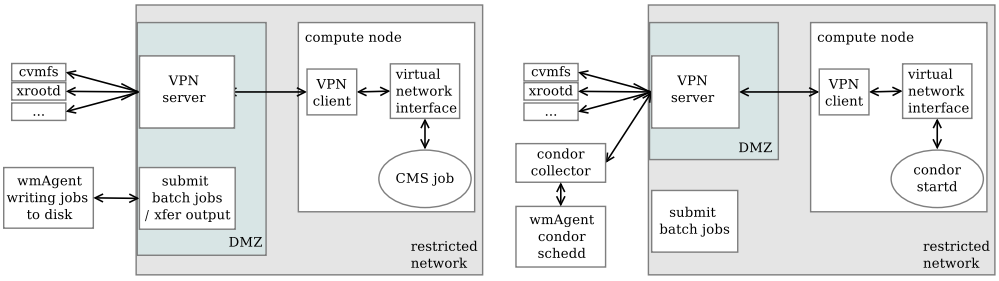

Figure 2. Possible uses for CMS. In (a), jobs from the WMAgent are written to disk, from which a process reads them and submits them to the compute nodes, reporting outputs to the WMAgent as needed. The payload of a batch job is a CMS job. In (b), HTCondor `startd`'s are submitted as payloads. These `startd`'s function as pilot jobs that contact a collector from which the WMAgent can find them and use them to execute CMS jobs.

This model can be harnessed in CMS in several ways. In Figure 2.(a) we show the prototype we follow for our tests for this paper. In the current CMS setup, jobs are created by WMAgent [8], and then submitted to the HTCondor [9] batch system for execution. We modified WMAgent so that jobs are written to disk together with their dependencies, and wrote a service that submitted these jobs inside a restricted network, reporting completions back to the WMAgent [10].

Another possible use is shown in Figure 2.(b), in which WMAgent is left unmodified, and instead the payload of the jobs running at the network restricted compute nodes are HTCondor `startd`'s daemons. A `startd` daemon, once running, declares the compute node to an HTCondor collector, from which pairings of compute node and jobs can be made. In our case, all communication from/to the `startd`'s is done through the VPN server.

In all use cases, the payload has access to services such as cvmfs and xrootd through the VPN server.

Note that the setup presented in this paper aims only to provide the necessary network connectivity to jobs running inside the HPC, but does not consider any particular pilot job solution, such as GlideinWMS[11], nor performs any resource management inside the HPC, such as in ARC Control Tower[12].

3 Implementation

To implement the VPN server-client architecture, we need to create virtual network interfaces at the client side without administration privileges. By creating a new network namespace [4]

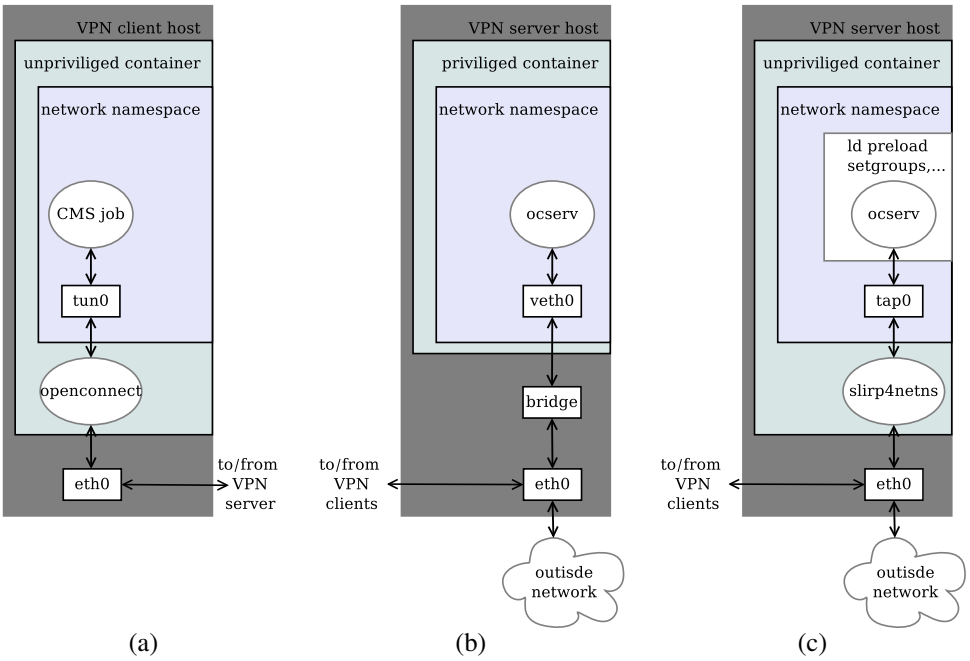

Figure 3. Implementing the VPN server-client architecture using namespaces. (a) The VPN client is run without admin privileges. All its network traffic is routed through a `tun` interface, created by `openconnect` and configured (IP, DNS, and gateway) by the VPN server. (b) VPN server runs with admin privileges inside a container and it can create directly a virtual network interface (`veth0`) and bridge, forwarding messages as needed. (b) VPN server runs without admin privileges. A `tap` device is create inside a namespace, to which forwarding and masquerading rules can be applied. The network stack for the `tap` interface is run in userspace by `slirp4netns`.

we can define this virtual network interfaces and and manipulate their addresses, routes and firewalls without administrator privileges. Any network modifications are only visible to processes that run inside the namespace.

On the client side, we use `openconnect`[13], an open source VPN client. In its default mode of operation, `openconnect` requires administrator privileges to create the network interface to route its traffic. However, instead of creating this interface itself, it gives the option to read/write to a user provided process. This allows to insert a program that creates a new namespace in which the virtual interface is created. The client then simply reads and writes to this process instead of a network interface. We use the tool `vpnns` from the open source project `ocproxy` [14] to create the namespace in which the virtual network interface is defined.

The filesystem `cvmfs` is by default provided with a fusermount via `cvmfsexec`[6], but can alternatively be provided by `parrot`[15], or mounted directly if available. We encapsulate all this dependencies inside a singularity container resulting in the client architecture is shown in Figure 3.a.

For the server side we have two options, which depend on whether we can run privileged containers in the VPN server host. The VPN server needs to manipulate the iptables related to its network interface in order to properly route packets from the VPN clients. When the VPN server runs with administrator privileges it can directly manipulate the traffic to the hardware

interface, and no other component is needed. Regardless, the host needs to be configured to allow IP forwarding. When the VPN server needs to run without administrator privileges, we again create a virtual network interface inside a namespace. However, since the VPN server manipulates Ethernet frames, the network interface created is a `tap` interface, which simulates a link layer device. This is different from the interfaces created for the clients, which are `tun` interfaces that simulate network layer devices and which carry IP packets. This subtle distinction means that the TCP/IP stack for the `tap` interface needs to be run on userspace, as regular users cannot manipulate Ethernet frames of hardware devices. We use `slirp4netns` to create the `tap` interface together with its TCP/IP stack in userspace. As with the clients, we have encapsulated all these dependencies in a container.

We use `ocserv` as the VPN server, the open source companion to the `openconnect` client. The design of `ocserv` assumes that it will be run with administrator privileges. In particular, for each client connection `ocserv` creates a worker process that immediately drops administrator privileges. This provides security guarantees by segregating worker processes, however, to drop privileges, `ocserv` makes some system calls that are not allowed to regular users even inside the namespaces, such as `setuid`, `setgid`, and `setgroups`. As we run `ocserv` as a regular user, this drop in privileges is in any case not needed, and we solve this issue by capturing the relevant system calls and always returning success with a custom ld preloaded library.

As we mentioned, our setups runs both server and clients inside containers. When running with admin privileges for the server, both docker and singularity have options to create virtual network interfaces inside a namespace and configure bridge devices to connect to the hardware network interfaces. However, this is not strictly necessary, as all the setup can be done outside the container. This is more evident for the client and a server running without admin privileges, as for example singularity cannot create any new virtual devices unless running with escalated privileges.

3.1 Configuration

For hosts running the VPN server and clients, namespaces need to be activated. This is done with the `user.max_user_namespaces=10000` sysctl kernel option. Additionally, IP forwarding should be enabled in the machine running the VPN server with `net.ipv4.ip_forward=1`. Further, this machine should allow TCP and UDP connections to the port the clients will use to contact the server (by default 8443). These are the only steps that need administration privileges to be configured.

The VPN server configures the IP, DNS, and default gateway of the virtual interfaces of the clients. When not using the `tap` interface, by default we use the servers listed in the `/etc/resolv.conf` at the host running the VPN server. This host also becomes the default gateway. With the `tap` interface, we first route traffic through the `slirp4netns` network stack. Once running, `slirp4netns` provides a default gateway and DNS to the VPN clients, which network traffic is then routed through `slirp4netns` and the `tap` interface to the VPN server.

In addition to the server address, the client `openconnect` needs the server private key fingerprint to verify the identity of the server. In our setup, new certificates are created the fist time the server runs and the server key fingerprint is written to a file which then can be included when submitting the batch jobs that will spawn the VPN clients.

3.2 Alternatives

When namespaces are not enabled at the client side, network traffic can still be manipulated by `openconnect` via a SOCKS server. Instead of creating a virtual network interface, net-

work calls are captured with the `tsocks` [16] ld preloaded library. The captured calls are sent to a SOCKS server managed by `openconnect`, such as `ocproxy` [14], or `tunsocks` [17]. In [5] such use of capturing network traffic via `tsocks`, with the SOCKS server is provided by `ssh`.

We use a VPN client + SOCKS as a fallback when namespaces are not enabled. We prefer to create virtual network interfaces when available because it is easier to guarantee that all the traffic of the job will be correctly routed (e.g., some calls may altogether be missed, a process may reset ld preloaded libraries, or may be statically linked), and because tools such as `tc` can be used to directly limit traffic through the interface. However, the SOCKS server approach has the great advantages that it does not require hosts to be configured by an administrator, and that it is simpler to setup.

4 Evaluation

We first tested out setup by transferring data with `iperf`[18] to the VPN clients through the VPN server using different configurations. The server and clients ran on machines with 1 Gbps network links, with the `iperf` server running in the machine along side the VPN server. (This to ensure that any bandwidth limit hit was between server and clients, and not caused by the data server.) Our results are summarized in Table 1, where we show the average bandwidth for 1, 2, 10, and 20 simultaneous transfers. For the server stack column, *kernel* means a direct connection to the hardware interface, while *user* uses the `tap` interface.

As it can be seen in the table for multiple transfers, both namespace and SOCKS setups offer similar performance to not using a VPN at all. As perhaps expected, the network stack in userspace for the VPN server does incur an overhead, with a bandwidth around 1/5 compared to not using a VPN. This slowdown is in accordance to `slirp4netns`'s own published benchmarks [19]. To note, the author of `slirp4netns` has an experimental tool [20] as of October 2020 that accelerates `slirp4netns` to speeds close to the native host in kernels 5.9 and above, thus improvements in bandwidth usage are likely in the short-term.

For non-concurrent transfers, our results show that using the VPN is far from saturating the 1 Gbps link. This is not observed for multiple concurrent transfers. The most likely explanation is that we are using TCP over TCP for the communication between server and clients, which is known to have performance penalties. For `openconnect`, the recommended method is TCP over UDP using the DTLS protocol, however so far we have been unable to use DTLS with the user provided process to which `openconnect` redirects traffic.

We also tested our setup by running CMS Monte Carlo production tasks. We setup these tasks to run for about one hour, using the setup suggested in Figure 2.a. The jobs description generated by a WMAgent are written to disk together with all their dependencies, and transferred to be executed on a remote site [10] that is not owned nor configured by CMS (opportunistic resources at the University of Notre Dame). In Table 2 we show running time histograms across different configurations. Since these are Monte Carlo tasks, network traffic mainly comes from cvmfs software dependencies, and staging out results to xrootd. For these test, we compare cvmfs being provided by the fuse module, and by cvmfsexec across the different VPN configurations. These results show the feasibility of running CMS tasks through the proposed VPN setup.

5 Future Directions

Two future directions already available to us which we have not explored extensively are the shaping of traffic between VPN server and client, and having HTCondor `startd`'s as

xfers	server stack	VPN client	avg per transfer	avg concurrent
1	kernel	no	871.98	871.98
		namespace	333.59	333.59
		socks	196.61	196.61
	user	namespace	196.06	196.06
2	kernel	no	470.99	926.84
		namespace	298.68	597.36
		socks	190.54	381.09
	user	namespace	103.24	206.48
10	kernel	no	94.87	890.56
		namespace	88.26	882.64
		socks	86.88	868.87
	user	namespace	19.96	199.67
20	kernel	no	47.63	846.01
		namespace	44.24	884.90
		socks	43.56	871.35
	user	namespace	7.24	144.83

Table 1. Bandwidth (Mbps) across different configurations. `xfers` indicates the number of simultaneous transfers by `iperf` for 5 minutes. File server, VPN server and clients run on 1 Gbps links. `server stack` indicates whether the VPN server connects directly to the hardware interface (kernel) or using a tap device with network stack in user space via slirp4netns (user). The VPN client may be disabled (`no`), run on top of a tun device in a network namespace, or capture network calls with a ld preload library and a SOCKS server.

concurrent jobs	VPN client	cvmfs	median runtime (minutes)
50	namespace	cvmfsexec	45.13
	namespace	fuse	41.67
	no	fuse	41.82
100	namespace	cvmfsexec	47.75
	namespace	fuse	50.71
	no	fuse	48.90
200	namespace	cvmfsexec	59.91
	namespace	fuse	60.78
	no	fuse	60.20

Table 2. Median runtime (minutes) of CMS Monte Carlo test jobs. Jobs were constructed to run for about one hour. We compare the performance of different number of concurrent jobs providing cvmfs with a fusermount in a namespace (cvmfsexec), or directly with the cvmfs fuse module.

.

the payload of the batch jobs. For example, when using virtual interfaces we can directly use standard tools such as `tc` to limit bursts and bandwidth of egress traffic. Additionally, `ocserv` provides separate ingress and egress bandwidth limits, both global and per user. Another immediate future direction is perform load-balancing using multiple VPN servers to avoid exhausting single network links.

Software

Recipes and instructions to implement our solution are publicly available here:
https://github.com/cooperative-computing-lab/userlevel-vpn-tun-tap

Acknowledgments

The authors thank Irena Johnson and Paul Brenner for their invaluable help using network resources at University of Notre Dame; Todor Ivanov for its input running CMS production jobs; and Mirko Mariotti, Daniele Spiga, and Tommaso Boccali for making their ideas and modifications to `tsocks` available to us in the development of this paper.

References

[1] CMS-Collaboration, *Projected performance of an upgraded CMS detector at the LHC and HL-LHC: Contribution to the snowmass process* (2013), 1307.7135

[2] T. Boccali, *CMS Software and Offline preparation for future runs*, in *19th International Workshop on Advanced Computing and Analysis Techniques in Physics Research* (2019)

[3] C. Aguado-Sanchez, J. Bloomer, P. Buncic, L. Franco, S. Klemer, P. Mato, *CVMFS a file system for the CernVM virtual appliance*, in *Proceedings of XII Advanced Computing and Analysis Techniques in Physics Research* (2008)

[4] *ip-netns(8) Linux User's Manual* (2021)

[5] M. Mariotti, D. Spiga, T. Boccali, *A possible solution for HEP processing on network secluded Computing Nodes*, in *ISGC 2021* (2021), pp. 22–26

[6] *cvmfsexec*, https://github.com/cvmfs/cvmfsexec (2021), Accessed: 2021-05-03

[7] *userlevel-vpn-tun-tap*, https://github.com/cooperative-computing-lab/userlevel-vpn-tun-tap (2021), Accessed: 2021-05-03

[8] E. Fajardo, O. Gutsche, S. Foulkes, J. Linacre, V. Spinoso, A. Lahiff, G. Gomez-Ceballos, M. Klute, A. Mohapatra, *A new era for central processing and production in CMS*, in *Journal of Physics: Conference Series* (IOP Publishing, 2012), Vol. 396-4

[9] D. Thain, T. Tannenbaum, M. Livny, Concurrency - Practice and Experience **17**, 323 (2005)

[10] B. Tovar, D. Thain, *Executing CMS production workflows on third-party software*, Manuscript in preparation. (2021)

[11] I. Sfiligoi, D.C. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, F. Wurthwein, *The Pilot Way to Grid Resources Using GlideinWMS*, in *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering* (2009), Vol. 2

[12] D. Cameron, A. Filipčič, W. Guan, V. Tsulaia, R. Walker, T. Wenaus, Journal of Physics: Conference Series **898** (2017)

[13] *openconnect*, http://www.infradead.org/openconnect (2021), Accessed: 2021-05-03

[14] *ocproxy*, https://github.com/cernekee/ocproxy (2021), Accessed: 2021-05-03

[15] D. Thain, M. Livny, *Parrot: An Application Environment for Data-Intensive Computing*, in *Journal of Parallel and Distributed Computing Practices* (2004)

[16] *tsocks*, https://sourceforge.net/projects/tsocks/ (2002), Accessed: 2021-05-03

[17] *tunsocks*, https://github.com/russdill/tunsocks (2019), Accessed: 2021-05-03

[18] *Iperf - the TCP/UDP band-width measurement tool*, http://dast.nlanr.net/Projects/Iperf (2020), Accessed: 2021-05-03

[19] *Bandwidth benchmarks of slirp4netns*, https://github.com/rootless-containers/rootlesskit/pull/12 (2018), Accessed: 2021-05-03

[20] *bypass4netns*, https://github.com/rootless-containers/bypass4netns (2020), Accessed: 2021-05-03