A Virtual Test Environment for MPI Development: Quick Answers to Many Small Questions

Wolfgang Schnerring, Christian Kauhaus, Dietmar Fey

Lehrstuhl für Rechnerarchitektur und -kommunikation, Institut für Informatik,
Friedrich-Schiller-Universität, 07737 Jena, Germany.
{wosc,kauhaus,fey}@cs.uni-jena.de

Abstract. MPI implementations are faced with growingly complex network configurations containing multiple network interfaces per node, NAT, or dual stacks. To implement handling logic correctly, thorough testing is necessary. However, the cost of providing such diverse setups in real hardware is prohibitively high, resulting in a lack of testing. In this article, we present a *Virtual Test Environment (VTE)* that considerably lowers this barrier by providing complex network environments on a single computer and thus enables testing in situations where it otherwise would not be feasible.

1 Introduction

As today's cluster computers become more and more sophisticated, complex network setups are increasingly common. Of course, application writers do not want to be bothered with network topology and expect their MPI implementation to cover the details. In consequence, nearly any modern MPI library contains nontrivial amounts of logic to perform the initial wire-up: starting daemons, querying addresses, selecting network interfaces, etc.

To implement wire-up logic code properly, frequent and thorough testing is necessary. During the development of our IPv6 extension to Open MPI [1], we ran into the problem of verifying correct behaviour on configurations like clusters with multiple networks, multi-domain clusters using both private and public IPv4 addressing, mixed IPv4/IPv6 environments, and others. The cost of providing such setups was prohibitively high, resulting in a lack of testing and, in consequence, undiscovered bugs.

To remedy this problem, we present a *Virtual Test Environment*[1] that lowers the testing effort significantly, so that the execution of some classes of functional tests becomes practicable at all. Additionally, it is lightweight enough to facilitate a rapid feedback cycle during development. VTE builds virtual clusters on the developer's workstation from high-level descriptions by employing kernel virtualisation. The *Unit Under Test (UUT),* e. g., a MPI implementation, is exercised in a variety of network setups. Although VTE was created to facilitate

[1] The software can be obtained from http://www2.informatik.uni-jena.de/cmc/vte/.

the development of our IPv6 extension to Open MPI, it is not constrained to this task. In fact, VTE is useful for the development of any distributed application which interacts with a TCP/IP network environment.

This paper is organised as follows. Section 2 explains the design and implementation of VTE. Section 3 illustrates the capabilities of VTE using real-world examples. Section 4 concludes and outlines VTE's further development.

2 Design and Implementation

The aim of VTE is to lower the cost associated with the testing process. As VTE has been specifically designed to reflect this goal, we have made several architectural decisions. We first give an overview over those decisions, and then present them in more detail.

First, VTE needs a concise description language for test networks. If it is too difficult to specify the test setup (e. g., spending a day in the machine room installing cables), the developer is very likely to skip testing. Second, tests should run quickly. If tests take longer than a few minutes, it is too tedious to test frequently. Third, the test environment should be transparent to the UUT, requiring no test-specific modifications. But the test environment should *not* be transparent to the developer, providing him with effective control and inspection devices. Fourth, the test environment should be able to model complex network configurations to be representative of most cluster environments seen today.

Related to our work are MLN [2], an administration tool for virtual machines that allows a declarative description, and Xen-OSCAR [3], a virtualisation-based tool to test cluster installation processes. Unfortunately, both are not fit for MPI implementation testing, since they are not optimised for short running time and provide no means to reconfigure the network during runtime.

2.1 Concise Description of Network Configurations

To provide the developer with concise means of specifying tests, we designed a domain-specific language to describe network configurations. The virtual network can be built out of components like *host* and *switch*, which model physical devices, but do more than their real-world counterparts. For example, when connecting VMs with a switch, they are assigned IP addresses automatically.

To enable for such concise expression, VTE is implemented in Ruby, a language that allows to express facts on a very high, abstract level. Configurations are executable Ruby programs that specify all necessary information to create virtual machines and networks. While executing a network description, VTE both builds internal data structures describing the configuration and interacts with the host system through shell commands to realise those structures.

Thus, specifying

```
cluster = Cluster.new(16)
switch = Switch.new("192.168.5.0/24").connect(cluster.vm)
```

configures a virtual cluster with 16 nodes that are connected through a switch and have IP addresses from the subnet 192.168.5.0/24.

This example shows some of the most important language elements. The Cluster class governs a set of virtual machines. With Cluster.new, virtual machines are booted and the Cluster.vm accessor provides references to all running VMs. After startup, VMs do not have a network connection. A virtual network is created using Switch.new, which also causes VTE to create the supporting operating system structures (see 2.4). The Switch.connect method creates network interfaces, allocates IP addresses, and creates the connections. More configuration language elements are documented in the API reference.

We designed the configuration language to use sensible defaults, while allowing exceptions to be specified if desired. If the network above was not given explicitly but rather just as the type (using Switch.new(:ipv4)), a suitable IPv4 address range would be allocated automatically.

2.2 Fast Execution

To run fast, VTE requires a lightweight virtualisation technology, since the main overhead stems from running virtual machines. The "smallest" form of virtualisation is kernel-based virtualisation [4], which merely creates a compartment inside the running kernel to isolate processes of one VM from those of other VMs. We have chosen OpenVZ [5] since it is the most mature of the freely available kernel-based virtualisers for Linux and offers thorough network support.

Kernel-based virtualisation allows for rapid creation of virtual machines, since creating a VM mostly consists of spawning another init process and a few required daemons, for example sshd, reusing the already running kernel. This also means that VMs are not able to use a different operating system than the one which is already present, but since VTE focuses on complex network configurations, heterogeneous operating systems are not a primary concern.

We accelerate the startup process even further by reusing one file system image for all virtual machines: Instead of setting up separate file systems for newly created VMs, only one master file system is mounted repeatedly using copy-on-write semantics. Write operations are diverted to a separate location, while the master file system stays read-only. With this technique, VTE is able to create VMs very quickly: Figure 1 shows the total time required for starting VMs and shutting them down again on an Intel Pentium 4 at 3 GHz with 2 GB RAM running kernel version 2.6.18-ovz028test019. Even 50 VMs take only about a minute, but most tests will not require much more than 10 VMs. Since their startup and shutdown takes only 16 seconds, this presents practically no barrier to frequent testing. OpenVZ is also quite efficient: VTE is capable of simulating a cluster with 300 nodes on the same machine. An MPI ring test then takes about 3 minutes, plus about 8 minutes VM startup and shutdown time.

To test the UUT with several network configurations, it should not be necessary to stop the VMs and start them again in a different setup. Therefore, we implemented all virtual network components in a way they can be completely removed and new ones added while the VMs keep running.

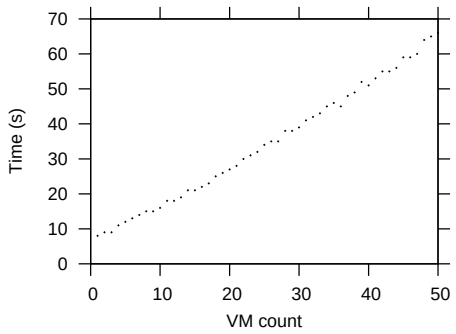

Fig. 1. Time required for startup and shutdown.

To bring the UUT into the test environment, it is not necessary to copy it onto all virtual machines. OpenVZ allows to simply mount directories of the host into a VM, which saves a potentially large amount of data transfers. For example, to inject a MPI implementation into the test environment, the developer's working directory can be mounted, even under the same path as on the host, so the MPI implementation does not notice any difference.

2.3 Semi-Transparency

VTE is able to run real software, and it appears as a real cluster to the software running inside it. But the developer has full control over this environment, so VTE could be imagined as a semi-transparent mirror.

The test environment can be controlled by executing shell commands inside the virtual machines and examining their output. We implemented running shell commands using OpenVZ's `vzctl exec` mechanism. An alternative would be to inject commands into VMs using `ssh` calls via a virtual service network, which would of course influence the network setup. This way, however, VTE can be configured exactly as required for the tests—even completely without a network. Since the virtual machines export their network interfaces to the host, network sniffers or any other tools can be run on the host to examine traffic on the virtual network.

VTE provides a self-contained environment. Its only external dependencies are OpenVZ to provide the virtualisation, and root privileges for operations concerning the virtual environment, which are encapsulated in `sudo` calls.

2.4 Complex Network Configuration

To be able to test the behaviour of MPI implementations even for complicated cases, VTE should model fairly complex network environments. We limited the network support to TCP/IP though, favouring simplicity and conciseness of the configuration language over heterogeneous network families.

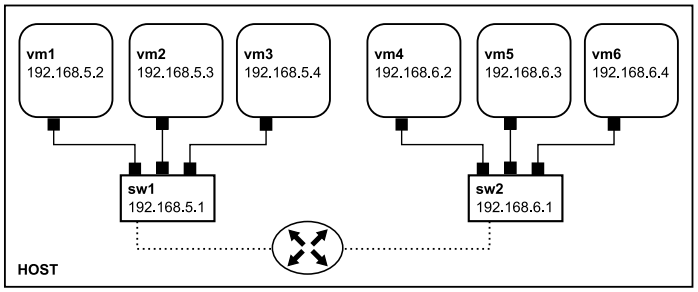

Fig. 2. Example network configuration.

Figure 2 shows an example setup where two clusters are connected via a router. VTE constructs this virtual network on the host, since the network interfaces of the virtual machines are easily accessible from there. Switches are implemented by using standard Linux bridge−utils to create virtual bridges that connect the network interfaces of the VMs. Since the bridges can serve a dual purpose and function as network interfaces of the host, the host itself can also be connected to a virtual switch if desired. We use this to implement a router using the host's routing table. Network Address Translation (NAT) is also performed on the host via iptables. Thus, virtual network components can be combined to model multi-cluster setups, multi-link connections (e. g. channel bonding), dual stack (IPv4/IPv6) networks, and other configurations.

3 Examples

To show the ease of use that VTE offers for specifying and running tests, we examine the behaviour of MPI implementations that occurred to us during our day-to-day development work to show how VTE can help preventing this kind of problems by facilitating regular testing.

3.1 Striping

If there are multiple network paths between any two nodes, Open MPI optionally uses striping to maximise throughput. With striping, Open MPI fragments big messages and sends them in parallel through several interfaces. VTE allows for each VM to have multiple network interfaces, for example by specifying an interface number when connecting VMs with a switch:

```
cluster = Cluster.new(2)
Switch.new(:ipv4).connect(cluster.vm, 0)
Switch.new(:ipv4).connect(cluster.vm, 1)
```

In this example, each VM will be configured with two network interfaces, eth0 and eth1. Running the ping-pong benchmark from the Intel MPI Benchmark

Suite (IMB) 3.0 shows that once the payload is large enough, Open MPI will use striping to distribute the traffic over all available interfaces. The tcpdump output shows connections on both switches.

```
# tcpdump -i sw1
IP 1.0.0.2.56003 > 1.0.0.3.33364
IP 1.0.0.3.33364 > 1.0.0.2.56003
# tcpdump -i sw2
IP 2.0.0.2.54115 > 2.0.0.3.59036
IP 2.0.0.3.59036 > 2.0.0.2.54115
```

3.2 NAT

Many clusters use private IPv4 addresses for their nodes, which connect to the outside world using a NAT gateway. Combining two clusters of this kind to a multi-cluster is not possible, since the private addresses of one cluster are not reachable from the other. MPI implementations should detect this erroneous condition. We examine the behaviour of LAM/MPI and Open MPI using VTE.

Two clusters with 3 nodes each that use private IPv4 addresses and are connected with NAT (see Figure 2) can be modelled like this:

```
cluster1 = Cluster.new(3)
cluster2 = Cluster.new(3)
switch1 = Switch.new("192.168.5.0/24").connect(cluster1.vm)
switch2 = Switch.new("192.168.6.0/24").connect(cluster2.vm)
router = Router.new.connect(switch1, nat=true).connect(switch2, nat=true)
```

LAM/MPI [6] is started with the lamboot command. The required host file is automatically constructed by concatenating the host names of all VMs.

```
hostnames = (cluster1.vm + cluster2.vm).map { |vm| vm.hostname }.join("\n")
cluster1.vm[0].cmd("echo '#{hostnames}' > lamhosts")
cluster1.vm[0].cmd("lamboot lamhosts")
```

lamboot tries to start LAM's management daemon lamd on all hosts, but since some hosts use private addresses and are unreachable, it terminates with an error message:

```
ERROR: LAM/MPI unexpectedly received the following on stderr:
ssh: connect to host vm3 port 22: Network is unreachable
```

Open MPI [7] starts its daemon orted implicitly when mpirun is called.

```
hostnames = (cluster1.vm + cluster2.vm).map { |vm| vm.hostname }.join(",")
cluster1.vm[0].cmd("mpirun -np #{cluster.vm.size} -host #{hostnames} ./ringtest")
```

Open MPI also fails to contact some hosts, but instead of terminating it hangs until VTE's timeout kills it:

```
ssh: connect to host vm3 port 22: Network is unreachable
[vm0:21702] ERROR: A daemon on node vm3 failed to start as expected.
[vm0:21702] ERROR: There may be more information available from
[vm0:21702] ERROR: the remote shell (see above).
Timeout: aborting command "vzctl" with signal 9
```

The problem is that after Open MPI notices the error, it terminates by telling orted on all hosts to shut down—including those hosts that it was unable to connect in the first place. Since these hosts never return a "shutdown successful" notification, the process keeps waiting for them forever.

3.3 Installation Prefix

If Open MPI is installed below a different path than the prefix specified at compile time, it is unable to find its binaries and libraries. One method of explicitly specifying installation locations is to use application context files. The manual page for mpirun states that "[...] --prefix can be set on a per-context basis, allowing for different values for different nodes." With VTE, two nodes with different Open MPI installation locations can be described as follows:

```
cluster = Cluster.new(2)
switch = Switch.new(:ipv4).connect(cluster.vm)
cluster.vm[0].mount("/real/path/to/openmpi", "/usr/local/openmpi")
cluster.vm[1].mount("/real/path/to/openmpi", "/opt/openmpi")
cluster.vm[0].cmd("echo '#{<<__EOT__}' > appfile")
  -np 1 -host vm0 --prefix /usr/local/openmpi hostname
  -np 1 -host vm1 --prefix /opt/openmpi hostname
__EOT__
cluster.vm[0].cmd("mpirun --appfile appfile date")
```

Unfortunately, the --prefix settings does not take effect and mpirun fails with /usr/local/openmpi/bin/orted: No such file or directory on vm_1. Perhaps the writing of the manual page proceeded a little bit quicker than the writing of the code.

3.4 Dual-stack

When migrating to IPv6 [8], it is quite common to have a dual-stack setup, that is both address families on a single interface, connected by the same switch. In VTE an IP subnet is usually represented by a switch, but in this case there is only one switch but two networks. Therefore we have to configure the second set of IP addresses explicitly, using the provided helper functions:

```
cluster = Cluster.new(3)
Switch.new(:ipv6).connect(cluster.vm)
net = Switch.generate_network(:ipv4)
cluster.vm.each_with_index do |v, i|
  v.configure_ipv4(net.nth(i+1)) unless v.hostname_short == "vm1"
end
Cluster.update_hostfile
```

VTE's configuration language is plain Ruby, so all normal language constructs are available to the developer. This makes it easy to describe exceptions like "every host has IPv4 except vm_1."

Running a ring test and observing the traffic on the switch with tcpdump shows that Open MPI with our IPv6 extension now opens connections via both

IPv4 and IPv6, depending on the connectivity available. vm_0 connects to vm_2 via IPv4 (1.0.0.1–1.0.0.3), but to vm_1 via IPv6 (aa01::1–aa01::2).

```
# tcpdump -i sw1
IP 1.0.0.1.33126 > 1.0.0.3.52353
IP 1.0.0.3.52353 > 1.0.0.1.33126
[...]
2001:638:906:aa01::1.38252 > 2001:638:906:aa01::2.43756
2001:638:906:aa01::2.43756 > 2001:638:906:aa01::1.38252
```

These examples show some of the VTE's possibilities to exercise MPI implementations in a variety of environments. The configuration language is expressive enough to formulate even bug-provoking border cases in a compact form.

4 Conclusion

We have presented a Virtual Test Environment that considerably lowers the barrier to functional testing of distributed applications, thereby enabling testing in cases where it were not feasible otherwise, since the costs of physical test setups are too high. VTE has an expressive configuration language for describing complex network environments and short execution times. Being able to run networking tests in just a few minutes allows the developer to establish a close feedback loop, which results in better software quality.

References

1. Kauhaus, C., Knoth, A., Peiselt, T., Fey, D.: Efficient message passing on multi-clusters: An IPv6 extension to Open MPI. In: Proceedings of KiCC'07, Chemnitzer Informatik Berichte CSR-07-02. (February 2007)
2. Begnum, K.M.: Managing large networks of virtual machines. In: Proc. LISA'06: 20th Large Installation System Administration Conference, Washington, D.C., USENIX Association (December 2006) 205–214
3. Vallée, G., Scott, S.L.: OSCAR testing with Xen. In: Proc. 20th Int. Symp. on High-Performance Computing in an Advanced Collaborative Environment (HPCS'06), Washington, DC, IEEE Computer Society (2006) 43–48
4. Soltesz, S., Pötzl, H., Fiuczynski, M., Bavier, A., Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: Proceedings of EuroSys 2007, Lisbon, Portugal (March 2007)
5. SWSoft: OpenVZ – server virtualization open source project. http://openvz.org/ Accessed on June 27, 2007.
6. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Proc. 10th European PVM/MPI Users' Group Meeting. Number 2840 in LNCS, Springer (2003) 379–387
7. Gabriel, E., Fagg, G.E., Bosilca, G.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (September 2004) 97–104
8. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard) (December 1998)