Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

EXPERIENCE REPORT

Big Data Deployment in Containerized Infrastructures through the Interconnection of Network Namespaces

Carla Sauvanaud[1] | Ajay Dholakia[3] | Jordi Guitart[1,2] | Chulho Kim[3] | Peter Mayes[3]

[1]Computer Architecture Department, Universitat Politecnica de Catalunya (UPC), Barcelona, Spain
[2]Computer Sciences Department, Barcelona Supercomputing Center (BSC), Barcelona, Spain
[3]Lenovo Data Center Group, Lenovo, Morrisville, USA

Correspondence
Jordi Guitart, Barcelona Supercomputing Center (BSC), Jordi Girona, 31, Barcelona, Spain. Email: jordi.guitart@bsc.es

Summary

Big Data applications tackle the challenge of fast handling of large streams of data. Their performance is not only dependent on the data frameworks implementation and the underlying hardware, but also on the deployment scheme and its potential for fast scaling. Consequently, several efforts have focused on the ease of deployment of Big Data applications, notably through the use of containerization. This technology was indeed raised to bring multi-tenancy and multi-processing out of clusters, providing high deployment flexibility through lightweight container images. Recent studies have focused mostly on Docker containers. Notwithstanding, this paper is actually interested in recent Singularity containers as they provide more security and support high performance computing (HPC) environments and, in this way, they can make Big Data applications benefit from the specialized hardware of HPC. Singularity 2.x however does not isolate network resources as required by most Big Data components. Singularity 3.x allows allocating each container with isolated network resources, but their interconnection requires a non-trivial amount of configuration effort. In this context, this paper makes a functional contribution in the form of a deployment scheme based on the interconnection of network namespaces, through underlay and overlay networking approaches, to make Big Data applications easily deployable inside Singularity containers. We provide detailed account of our deployment scheme when using both interconnection approaches in the form of a 'how-to-do-it' report, and we evaluate it by comparing three Big Data applications based on Hadoop when performing on a bare-metal infrastructure and on scenarios involving Singularity and Docker instances.

KEYWORDS:
Singularity containers ; Hadoop ; Big Data ; Namespaces

1 | INTRODUCTION

Virtualization and containerization have been lately used to support Big Data frameworks because of the advantages they provide regarding the pooling of resources for different applications from different users, the elasticity of resource provision, as well as the isolation of these applications in terms of both resources and security. The ability to encapsulate the execution environment could make a difference to simplify the deployment of Big Data frameworks given the amount of dependencies they have and the specific version requirements of dependent libraries.

In more detail, virtualization abstracts hardware in separated pools of resources called virtual machines (VMs), and makes VMs able to run any operating system (OS). As for containerization, it allows resource gathering in containers while sharing kernel functions, modules, and libraries of the host OS. Both of these technologies allow getting multi-processing out of a cluster, as well as multi-tenancy, which enables to share the hardware resources between multiple internal lines-of-business teams.

The overhead of using virtualization for Big Data applications has been studied in several works [1,2,3], coming out with promising results. For example, VMware evaluated to what extent such technologies could actually benefit to the performance of Big Data applications [2] and highlighted that the overall performance of Hadoop and Spark on Hadoop can be improved using virtualization compared to a bare-metal deployment, when running several virtual machines in a physical host and sizing them in order to fit within a single Non-Uniform Memory Access (NUMA) node. More recently, it is the containerization that got more interest due to its quick deployment, short starting time, and smaller sized images [4,5,6,7]. Those works particularly focused on Docker[1] containers, especially because Docker is currently the most popular technology for services deployments.

Singularity [8] is a novel containerization technology that proposes quickly deployable and transferable containers without encapsulating an entire OS inside the containers images. Singularity also overcomes some issues of former containerization technologies, especially as for security since it does not create containers as spawned child processes of a root owned daemon (like Docker does).

Currently, Singularity containers are mostly deployed in high performance computing (HPC) environments where they are proven to introduce less overhead than Docker [9] while being more secure and providing native GPU support[2]. They are mostly used to run Message Passing Interface (MPI) jobs, which do not require isolated network resources for each container when running on Singularity. The integration between Singularity and MPI is transparent to the user who only runs the command `mpirun` while the containers deployment and the processes launching and communication is handled behind-the-scenes by the MPI process management daemon (ORTED). However, this is not the case with Big Data applications and most enterprise services in general, for which each worker (or slave) must have separate network resources in order to be identified and reachable by other workers or the master.

[1] https://www.docker.com/
[2] https://hpc.nih.gov/apps/singularity.html#gpu

Singularity 2.x releases do not implement isolated network resources. Consequently, as several containers are created on the same host, all of them share the same IP address and ports as the host. This precludes the deployment of several components of a Big Data application in several container instances in a physical host if they are configured to use the same ports. Therefore, the advantages of exploiting NUMA locality in Big Data applications that have been demonstrated with virtual machines are not possible with Singularity 2.x containers.

In late 2018, Sylabs released Singularity 3.0, which introduced the option to join a starting container to a new network namespace and provided also full integration with the Container Network Interface (CNI)[3] project developed by the Cloud Native Computing Foundation[4], which consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins. Whereas this version of Singularity allows allocating each container with isolated network resources, their interconnection by means of CNI profiles is far from trivial. Furthermore, currently supported CNI plugins lack important functionality, such as the interconnection of containers by means of overlay approaches (i.e. using tunnels). This requires using third-party tools that implement the CNI specification, such as Flannel[5].

Regardless of these issues, the Singularity advantages regarding security, together with fast and flexible deployments are largely of interest for Big Data users. They are willing to accelerate the configuration and deployment processes of their applications, while not having to preoccupy about security. Besides, major assets regarding the improvement of performance might arise from the deployment of such applications over GPU and Infiniband network fabrics from HPC. The convergence between Big Data and HPC can benefit to businesses willing to dedicate their HPC environments (or HPC-like environments, also called Enterprise Performance Computing, EPC) to several purposes such as Deep Learning, Artificial Intelligence, and Big Data. Researchers and industries already started to exploit this convergence [10,11,12]. As an example of a practical case, Kamburugamuve et al. exploited HPC high performance interconnects, namely Infiniband and Intel Omni-Path, in order to improve the performance of the Apache Heron Big Data framework [13].

In this context, a comprehensive but effortless solution to deploy Big Data applications in Singularity container instances is needed. It must provide individual and isolated network resources to each container instance so that several of them can run on the same physical host and exploit NUMA locality if desired. It must enable also the interconnection of those container instances, even if they run in separate physical hosts. Furthermore, container instances will benefit from having general access to Internet apart from the connectivity with the other instances, especially during the software development phase.

As a result of this, the main contribution of this paper is to propose a detailed deployment scheme for Big Data applications to be deployed in Singularity container instances in a distributed environment. This scheme notably provides individual network resources to containers by means of network namespaces. Those namespaces can be interconnected by using either an overlay

[3] https://github.com/containernetworking/cni
[4] https://cncf.io/
[5] https://github.com/coreos/flannel

networking approach, using either Generic Routing Encapsulation (GRE)[14] or Virtual Extensible LAN (VXLAN)[15] tunnels, or an underlay networking approach, through MACVLAN [6]. Each container instance is also enabled to access the Internet through the corresponding network interface in the host.

This paper provides detailed account of our deployment scheme when using both interconnection approaches in the form of a 'how-to-do-it' report to allow its application in related projects in the same field. As for the evaluation, we present our experience from using our scheme for the deployment of Hadoop YARN[7] of the Apache Software Foundation and the comparative study of the performance of the deployment scheme in terms of computation time of Hadoop-based applications when using the various interconnection approaches.

The rest of this paper is organized as follows. The next section presents the related technical background. Section 3 introduces our deployment scheme and provides detailed instructions to allow for its reproducibility. In Section 4, four experimental deployment scenarios are studied and the performance, the scalability, and the capabilities of the proposed deployment scheme are evaluated. Section 5 presents the related work, and finally, Section 6 concludes the paper and discusses about the future work.

2 | TECHNICAL BACKGROUND

This section presents the basic technical background required for the deployment scheme presented in this paper. Concepts from the Linux kernel are hereby introduced, namely network namespaces, virtual Ethernet devices, GRE and VXLAN tunnels, which respectively enable to create abstracted spaces with isolated network devices, stacks, ports..., connect them with each other and to the Internet, and achieve a point-to-point or multicast communication between those namespaces across several hosts. As such, Linux kernels 3.8 or above are required, with the `gre` and `vxlan` modules being loaded. Moreover, Singularity container instances are also introduced, which supposes that Singularity 2.4 or above is available.

2.1 | Network namespaces

Network namespaces are a Linux kernel feature. Like other namespaces, they provide isolation of a resource, and in that case, this isolation is network-related. In more detail, network namespaces provide new and isolated network protocol stacks, devices, ports, routing tables, etc.

At first, a network namespace only gets access to a loopback interface (which is down by default, therefore it needs to be set up). As a result, it does not get any other access (for example, no access to the Internet or to the host routing table). In order for such a namespace to communicate, Virtual Ethernet devices or *veth*[8], from the Linux kernel must be used. Their interest

[6] https://www.linux.org/docs/man8/ip-link.html
[7] https://hadoop.apache.org/
[8] https://man7.org/linux/man-pages/man4/veth.4.html

in the context of this paper lies in their ability to work as a pair of devices, acting as a 'tunnel' that connects different network namespaces. In this way, two network namespaces created on the same host can communicate when each of them respectively has one end of the pair (i.e., one of the virtual devices), both devices are set up with private IP addresses of the same subnet, and the correct default route is enabled.

Network namespaces can also be handled remotely through an `ssh` connection if the corresponding daemon is run inside the namespace.

2.2 | Hostname namespaces

By default the hostname within a network namespace is the same hostname of the host. However, some Big Data applications require the hostname of each component to be unique. In this case, it is necessary to run a hostname namespace (i.e., a namespace isolating only the hostname of a machine) inside each network namespace, and start each Big Data component inside the corresponding hostname namespace. For example, the `unshare` command can be executed to open a Bash in a newly created hostname namespace. In this hostname namespace, one can then easily define a new hostname (with the `hostname` command), configure the `/etc/hosts` file to associate the local IP address to the new hostname, and run the corresponding application component.

2.3 | Singularity container instances

Singularity is a container solution created to offer the benefits of containerization to scientific applications. Its main focus was to offer mobility of compute and reproducibility to the programmers of those applications. Given that scientific focus, it was not until version 2.4 that Singularity introduced *container instances* to run services within containers. A container instance is a persistent and isolated version of the container image that runs in the background. This is similar to the concept of running container in Docker, and fits well with the needs of Big Data services to be run in containers.

A running container instance benefits from the network resources of the network namespace in which it is run. If an instance is executed in the host network namespace (so-called *root network namespace*) it will have access to all the host network interfaces. If it is started from any other network namespace, it will only have access to the network resources of that namespace. As a result, if the components of a Big Data application are executed within a single Singularity container instance that runs in a given network namespace, they will be able to communicate through the network provided by the namespace. Contrariwise, if we aim to execute different components in separate container instances, we should either run all the instances in the same network namespace, which is not feasible when considering multi-host deployments, or run each instance in a different network namespace and interconnect them, as we propose in this paper. However, Singularity 2.x does not offer built-in support for network namespaces, which was not introduced until Singularity 3.0.

2.4 | Interconnection of container instances

The deployment scheme proposed in this paper supports the interconnection of container instances through both underlay and overlay networking approaches. In underlay network approaches, container instances are directly exposed to the host network. In this paper, we consider MACVLAN as an example of this approach. It allows configuring multiple MAC addresses on a single physical interface. This can be used to assign a different MAC address (and consequently a different IP address) to each container instance, making it appear to be directly connected to the physical network. In that way, container instances can be addressed through their IP addresses. However, MACVLAN requires those addresses to be on the same broadcast domain as the physical interface. MACVLAN is a simple and efficient approach but the underlying network could restrict its application, in particular by limiting the number of different MAC addresses on a physical port or the total number of MAC addresses supported, or by forbidding multiple MAC addresses to be assigned on a single physical interface. Furthermore, MACVLAN is not generally supported for wireless network interfaces.

In overlay network approaches, a logical network between the container instances is built by using networking tunnels to deliver communication across hosts. Those tunnels add an additional level of encapsulation to the underlying network, and because of this, they may introduce some extra overhead when compared to an underlay approach, due to the encapsulation overhead of the frame size and the processing overhead on the server. Nevertheless, overlay network approaches are very flexible as they decouple the virtual network topology from the physical network, which supports for instance the mobility of components independently of the physical network. In addition, they essentially support an unlimited number of components as they do not suffer from restrictions to the number of addresses imposed by the physical network.

As examples of overlay network approaches, this paper considers GRE and VXLAN tunnels. A GRE tunnel is a point-to-point IP-over-IP tunnel which allows encapsulating data packets and sending them to a destination device that will de-encapsulate them. The data packets to be sent become the payload of a new packet which outer IP header is defined by the GRE. A VXLAN tunnel is a similar encapsulation approach that uses UDP instead of TCP as transport protocol. It supports point-to-point communications as well as multicast communications and has been commonly used in the virtualization community [16].

3 | DEPLOYMENT SCHEME

This section presents the deployment scheme proposed in this paper. It takes advantage of the abstractions presented in the technical background to communicate the components of a Big Data application which are deployed in Singularity container instances.

We assume a pool of M hosts, where each one is hosting N components of a Big Data application, each of them running within a different container instance. Each instance is executed on a separate network namespace to have its own network resources.

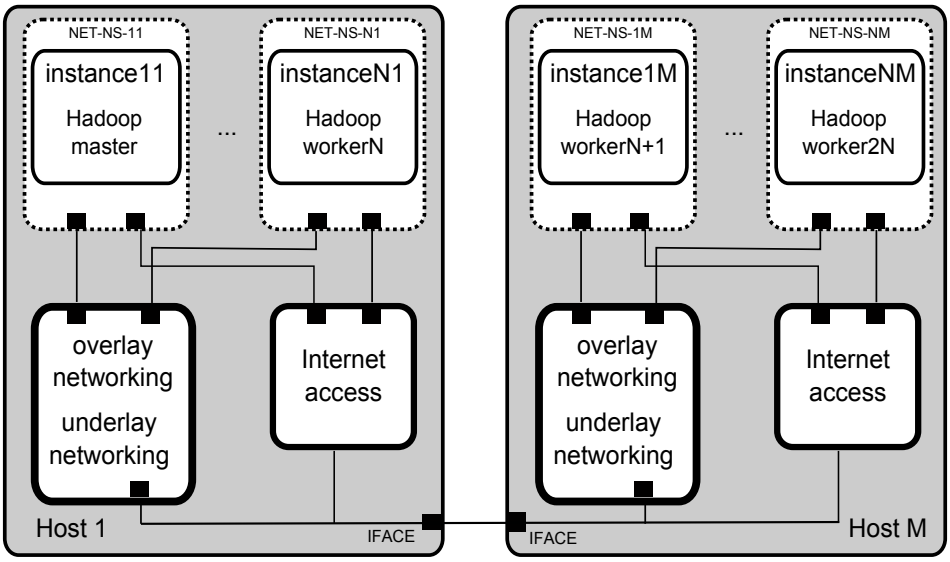

FIGURE 1 Deployment scheme for the interconnection of Singularity container instances.

Those networking resources are provided by two components deployed on each host: one of them enables the interconnection of namespaces by using both overlay and underlay network approaches, and the other enables the namespaces to access the Internet. This scheme for the deployment and interconnection of those components is shown in Figure 1. In order to carry out that deployment the next steps are required on each host $j \in [1, M]$:

1. Create the network namespaces *NET-NS-ij*, $i \in [1, N]$ where the container instances will be executed. When using Singularity 2.x, we must create them explicitly by running the command `ip netns add NET-NS-ij`. Singularity 3.x can do this for us on creating the container instances if we start them with the `-net` parameter (see step 4 below).

2. Interconnect the network namespaces *NET-NS-ij*.

   a) Using an overlay approach. See details on Section 3.1.

   b) Using an underlay approach. See details on Section 3.2.

3. Enable the access to the Internet for the network namespaces *NET-NS-ij*. See details on Section 3.3.

4. Create the Singularity container instances. When using Singularity 2.x, we must create them explicitly within the corresponding network namespaces *NET-NS-ij* that we interconnected before. When using Singularity 3.x, container interconnection is based on the definition of CNI profiles, hence we must use the `-network` parameter to provide the instances with the name of the appropriate profiles: one for the interconnection of namespaces using an overlay (*my-bridge-overlay-j*) or underlay approach (*my-bridge-macvlan-j*), and another to enable the namespace to access the Internet

(*my-bridge-net-j*). Those CNI profiles are created by means of configuration files, which are described in the corresponding subsections.

- Singularity 2.x:

```
$ ip netns exec NET-NS-ij singularity instance.start IMAGE.simg INSTANCE-ij
```

- Singularity 3.x:

```
$ singularity instance start -hostname=INSTANCE-ij -net -network
my-bridge-overlay-j,my-bridge-net-j IMAGE.sif INSTANCE-ij
```

5. Start some Big Data application component on the corresponding container instance *INSTANCE-ij*. When using Singularity 2.x, we must first use the `unshare` command to create a new host namespace within the container instance so that it has its own hostname. Singularity 3.x can do this for us if we provide the `-hostname` parameter when creating the container instance (see step 4 above).

   - Singularity 2.x:

```
$ singularity exec instance://INSTANCE-ij unshare -u bash -c 'hostname INSTANCE-ij ;
START-BIGDATA-APP.sh'
```

   - Singularity 3.x:

```
$ singularity exec instance://INSTANCE-ij bash -c 'START-BIGDATA-APP.sh'
```

3.1 | Overlay approach for interconnection

As shown in Figure 2a, to enable the interconnection of network namespaces through an overlay approach, we deploy a bridge on each host where we connect the network namespaces running on that host through virtual Ethernet devices pairs. The various bridges are then interconnected using tunneling, either GRE o VXLAN.

3.1.1 | Bridge and *veth* devices

When using Singularity 2.x, the required steps to deploy a bridge on each host *j* which connects the network namespaces on that host through *veth* pairs are as follows:

1. Create a bridge *BR-ov-j* for the connection to the overlay network and set it up. The bridge must be given an IP address *OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-j*, typically private (e.g. 192.168.x.x[9]).

[9]IP addresses in this paper are written in dot-decimal notation.

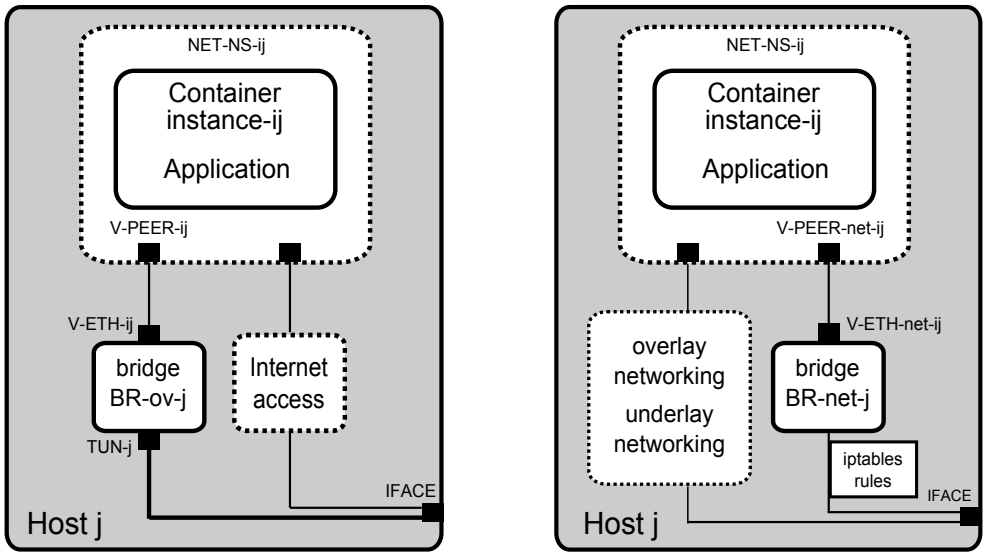

FIGURE 2 (a) Overlay networking approach; (b) Internet access for containers.

```
$ ip link add BR-ov-j type bridge
$ ip address add OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-j dev BR-ov-j
$ ip link set dev BR-ov-j up
```

2. Create *veth* pairs between each network namespace *NET-NS-ij* and the root network namespace. Each pair needs to get one end located inside the network namespace itself and the other end being located in the root network namespace.

```
$ ip link add V-ETH-ij type veth peer name V-PEER-ij
$ ip link set dev V-PEER-ij netns NET-NS-ij
```

3. Connect the *V-ETH-ij* end of the *veth* pair on each network namespace *NET-NS-ij* to the bridge *BR-ov-j* and set it up.

```
$ ip link set dev V-ETH-ij master BR-ov-j
$ ip link set dev V-ETH-ij up
```

4. As overlay approaches encapsulate a data packet as the payload of a new packet, we must reduce the Maximum Transmission Unit (MTU) of the *BR-ov-j* bridge and both ends of the *veth* pairs of each network namespace *NET-NS-ij*, subtracting the size of the new outer IP header to the default MTU. As a result, the new MTU must be 1450 for VXLAN and 1462 for GRE.

```
$ ip link set dev V-ETH-ij mtu new-MTU
```

```
$ ip link set dev BR-ov-j mtu new-MTU
$ ip netns exec NET-NS-ij ip link set dev V-PEER-ij mtu new-MTU
```

5. Configure the *veth* device *V-PEER-ij* for each network namespace *NET-NS-ij* with a unique IP address *OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-ij*. This IP address should be in the same subnet as the bridge *BR-ov-j*. Since our testbed only involves a small pool of hosts, here we chose to put all the *veth* devices from all the hosts in the same subnet. More elaborate network routing rules should be added when considering a different subnet for each host. To simplify the configuration of the Big Data application running within *NET-NS-ij* namespace, we also change the name of each device *V-PEER-ij* to a more standard form (*eth0*).

```
$ ip netns exec NET-NS-ij ip address add OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-ij dev V-PEER-ij
$ ip netns exec NET-NS-ij ip link set dev V-PEER-ij name eth0
```

6. Change the routing table within the *NET-NS-ij* namespace so that all the packets sent to nodes in the same overlay subnet *OV-IP-1.OV-IP-2.OV-IP-3.0/24* are routed through the device *V-PEER-ij* (now *eth0*) and set this device and the loopback device up.

```
$ ip netns exec NET-NS-ij ip link set dev eth0 up
$ ip netns exec NET-NS-ij ip route add OV-IP-1.OV-IP-2.OV-IP-3.0/24 dev eth0
$ ip netns exec NET-NS-ij ip link set dev lo up
```

When using Singularity 3.x, the previous steps to deploy a bridge on each host *j* that connects the network namespaces on that host through *veth* pairs are carried out through a *bridge* CNI profile, which is defined by means of a configuration file (*my-bridge-overlay-j.conflist*) as follows:

```
{
  "cniVersion": "0.3.1"
  "name": "my-bridge-overlay-j",
  "plugins": [ {
    "type": "bridge",
    "bridge": "BR-ov-j",
    "isGateway": true,
    "ipMasq": true,
    "mtu": new-MTU,
    "ipam": {
      "type": "host-local",
      "ranges": [ [ {
        "subnet": "OV-IP-1.OV-IP-2.OV-IP-3.0/24",
```

```
        "rangeStart": "OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-j-START",
        "rangeEnd": "OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-j-END",
        "gateway": "OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-j"
      } ] ],
      "routes": [ {
          "dst": "OV-IP-1.OV-IP-2.OV-IP-3.0/24",
          "gw": "OV-IP-1.OV-IP-2.OV-IP-3.OV-IP-4-j"
      } ]
    }
  } ]
}
```

3.1.2 | GRE and VXLAN tunneling

Since plugins currently implemented by CNI does not provide support for tunneling, the required steps to interconnect the various bridges using tunnels, either GRE o VXLAN, are the same for both Singularity 2.x and 3.x, and they are as follows:

1. Create a tunnel device *TUN-j* (either GRE or VXLAN) on the physical network interface *IFACE-j* of the host (let *IP-j* be the IP address of this interface):

   a) In the case of GRE, each host j establishes point-to-point tunnels with the rest of hosts $y \in [1, M]$, $y \neq j$ by running the command below. Those hosts should run the same command with the local and remote IP addresses inverted. When many hosts are to be connected together, lots of point-to-point tunnels should be configured, which makes this a tedious task (and inefficient as demonstrated in our evaluation). One could use instead Multipoint Generic Routing Encapsulation (mGRE) supporting multipoint tunnels, or deploy VXLAN in multicast mode (as supported by our scheme).

   ```
   $ ip link add TUN-j type gretap local IP-j remote IP-y dev IFACE-j
   ```

   b) In the case of the VXLAN, each host j establishes a multicast tunnel with the others on address *MCAST-ADDR* (typically 239.1.1.1).

   ```
   $ ip link add TUN-j type vxlan id 99 dev IFACE-j dstport 4789 local IP-j group MCAST-ADDR
   ```

2. Connect the tunnel interface *TUN-j* to the bridge *BR-ov-j* and set it up.

   ```
   $ ip link set dev TUN-j master BR-ov-j
   $ ip link set dev TUN-j up
   ```

3.2 | Underlay approach for interconnection

To enable the interconnection of network namespaces through an underlay approach, each of them is allowed access to the physical network of the host through a MACVLAN device, which sets a different MAC address for that namespace on the interface.

When using Singularity 2.x, the required steps to deploy a MACVLAN device for each network namespace *NET-NS-ij* on each host *j* are as follows:

1. Create devices *MACVLAN-ij* on the physical network interface *IFACE-j* of the host and assign them to the corresponding network namespaces *NET-NS-ij*.

```
$ ip link add MACVLAN-ij link IFACE-j type macvlan mode bridge
$ ip link set dev MACVLAN-ij netns NET-NS-ij
```

2. Assign each device *MACVLAN-ij* with an IP address *IP-1.IP-2.IP-3.IP-4-ij*. This address must be in the same subnet (*IP-1.IP-2.IP-3.0/24*) as the underlying associated network interface in this host (*IFACE-j*). To simplify the configuration of the Big Data application running within *NET-NS-ij* namespace, we also change the name of each device *MACVLAN-ij* to a more standard form (*eth0*).

```
$ ip netns exec NET-NS-ij ip address add IP-1.IP-2.IP-3.IP-4-ij dev MACVLAN-ij
$ ip netns exec NET-NS-ij ip link set dev MACVLAN-ij name eth0
```

3. Change the routing table within the *NET-NS-ij* namespace so that all the packets sent to nodes in the same subnet *IP-1.IP-2.IP-3.0/24* are routed through the device *MACVLAN-ij* (now *eth0*) and set this device and the loopback device up.

```
$ ip netns exec NET-NS-ij ip link set dev eth0 up
$ ip netns exec NET-NS-ij ip route add IP-1.IP-2.IP-3.0/24 dev eth0
$ ip netns exec NET-NS-ij ip link set dev lo up
```

When using Singularity 3.x, the previous steps to deploy a MACVLAN device for each network namespace *NET-NS-ij* on each host *j* are carried out through a *macvlan* CNI profile, which is created by means of a configuration file (*my-macvlan-net-j.conflist*) as follows:

```
{
```

```
  "cniVersion": "0.3.1",
  "name": "my-macvlan-net-j",
  "plugins": [ {
    "type": "macvlan",
    "master": "IFACE-j",
    "ipam": {
      "type": "host-local",
      "ranges": [ [ {
        "subnet": "IP-1.IP-2.IP-3.0/24",
        "rangeStart": "IP-1.IP-2.IP-3.IP-4-j-START",
        "rangeEnd": "IP-1.IP-2.IP-3.IP-4-j-END",
        "gateway": "IP-1.IP-2.IP-3.1"
      } ] ],
      "routes": [ {
        "dst": "IP-1.IP-2.IP-3.0/24",
        "gw": "IP-1.IP-2.IP-3.1"
      } ]
    }
  } ]
}
```

3.3 | Internet access

As shown in Figure 2b, to enable the Internet access for the network namespaces, we deploy a bridge on each host where we connect the network namespaces through *veth* pairs.

When using Singularity 2.x, the required steps to deploy a bridge on each host *j* that enables the Internet access for the network namespaces are as follows:

1. Create a bridge *BR-net-j* for the connection to the Internet and set it up. The bridge should be given an IP address *NET-IP-1.NET-IP-2.NET-IP-3.NET-IP-4-j*, typically private (e.g. 192.168.x.x).

   ```
   $ ip link add name BR-net-j type bridge
   $ ip address add NET-IP-1.NET-IP-2.NET-IP-3.NET-IP-4-j dev BR-net-j
   $ ip link set dev BR-net-j up
   ```

2. Create *veth* pairs between the root and the *NET-NS-ij* network namespaces.

   ```
   $ ip link add V-ETH-net-ij type veth peer name V-PEER-net-ij
   $ ip link set dev V-PEER-net-ij netns NET-NS-ij
   ```

3. Connect the *V-ETH-net-ij* end of the *veth* pair on each network namespace *NET-NS-ij* to the bridge *BR-net-j* and set it up.

   ```
   $ ip link set dev V-ETH-net-ij master BR-net-j
   ```

```
$ ip link set dev V-ETH-net-ij up
```

4. Configure the IP address and route for the Internet of the *veth* device *V-PEER-net-ij* for each network namespace *NET-NS-ij*. IP address should be in the same subnet as the bridge address *NET-IP-1.NET-IP-2.NET-IP-3.NET-IP-4-j*. The default routing gateway of the network namespace should be set to the the *veth* device *V-PEER-net-ij* (now eth1).

```
$ ip netns exec NET-NS-ij ip address add NET-IP-1.NET-IP-2.NET-IP-3.NET-IP-4-ij dev V-PEER-net-ij
$ ip netns exec NET-NS-ij ip link set dev V-PEER-net-ij name eth1
$ ip netns exec NET-NS-ij ip link set dev eth1 up
$ ip netns exec NET-NS-ij ip route add default dev eth1
```

5. Configure the forwarding rules to provide the access to the Internet to the bridged devices through the physical network interface *IFACE-j* of the host (which must have Internet connectivity).

```
$ ip netns exec NET-NS-ij iptables -t nat -A POSTROUTING -s NET-IP-1.NET-IP-2.NET-IP-3.0/24
-o IFACE-j -j MASQUERADE
$ ip netns exec NET-NS-ij iptables -t filter -I FORWARD -i BR-net-j -j ACCEPT
```

When using Singularity 3.x, the previous steps to deploy a bridge on each host *j* that enables the Internet access for the network namespaces are carried out through a *bridge* CNI profile, which is created by means of a configuration file (*my-bridge-net-j.conflist*) as follows:

```
{
  "cniVersion": "0.3.1",
  "name": "my-bridge-net-j",
  "plugins": [ {
    "type": "bridge",
    "bridge": "BR-net-j",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
      "type": "host-local",
      "ranges": [ [ {
         "subnet": "NET-IP-1.NET-IP-2.NET-IP-3.0/24",
         "rangeStart": "NET-IP-1.NET-IP-2.NET-IP-3.NET-IP-4-j-START",
         "rangeEnd": "NET-IP-1.NET-IP-2.NET-IP-3.NET-IP-4-j-END",
         "gateway": "NET-IP-1.NET-IP-2.NET-IP-3.NET-IP-4-j"
       } ] ],
       "routes": [ {
         "dst": "0.0.0.0/0"
       } ]
```

```
    }
  } ]
}
```

4 | EXPERIMENTS

The performance, the scalability, and the capabilities of the proposed deployment scheme for Singularity 2.x and 3.x are hereby evaluated and compared with bare-metal and Docker deployments through multiple experiments with four different deployment scenarios of the framework Hadoop YARN (called Hadoop for short in this paper). Hadoop is a map-reduce framework used to run computing jobs, schedule them, and manage their resources. Each deployment scenario involves a Hadoop *master* and some *workers*. A master is composed of the NameNode, Secondary NameNode, ResourceManager, and HistoryServer services, and a worker is composed of the DataNode and NodeManager services.

4.1 | Deployment scenarios

Each scenario is evaluated using two testbeds, namely a small-scale testbed comprising two hosts and a large-scale testbed comprising ten hosts. We use the former to assess the capabilities of the proposed deployment scheme and to compare the performance of the various deployment options, and the latter to account for their scalability. In order to fairly compare Hadoop performance among all the scenarios, all the small-scale experiments run 1 master and 2 workers, whereas all the large-scale experiments run 1 master and 10 workers.

Hadoop components are Java processes that communicate by means of IP address/port pairs. As such, given a host with a single network interface, a master and a worker can coexist on that host because they do not use the same ports. However, in a standard deployment scheme, two workers cannot coexist as they would use the same IP/port pair of the same network interface. As a consequence, to allow a fair comparison among the different deployment options, the experiments run only one worker per host. As our deployment scheme circumvents this limitation and allows to run several workers per host, we include also experiments to evaluate the potential benefit of exploiting this option.

Table 1 presents the distribution of master and worker processes for the various scenarios. Scenario *A* evaluates the performance of Hadoop on bare-metal (neither any master nor any worker runs in a container). In this scenario, the master shares its host server with one worker, and each of the other workers runs alone in a server. Note that no other worker could be deployed. This scenario corresponds to the nominal case study and Hadoop is currently deployed on most infrastructures in this way.

Scenarios *B*, *C*, *D* evaluate the performance of Hadoop when running on container instances, including Singularity 2.x (`SNG2`), Singularity 3.x (`SNG3`) and Docker (`DCK`). Scenario *B* is similar to scenario *A* except that, in this case, components are placed within container instances. This scenario only evaluates the overhead of using containers. As with scenario *A*, scenario

TABLE 1 Description of experimental scenarios.

Id	host-1		host-j	total workers		net namespaces	containers
	master	workers	workers	small-scale	large-scale		
A	1	1	1	2	10	No	No
B	1	1	1	2	10	No	Yes
C	1	1	1	2	10	Yes	Yes
D	1	0	2	2	-	Yes	Yes

B makes use of the underlying network connecting the host servers for the communication between components. Each instance accesses directly the network interface at the host where it runs and adopts the corresponding IP address. As a result, not more than a single worker can be deployed in a given host.

Scenario *C* evaluates, in addition of scenario *B*, the overhead of using an overlay network, either a GRE or a VXLAN tunnel, or the underlay network, through MACVLAN, for the communication between components. This scenario actually corresponds to our deployment scheme and it enables a strict performance comparison with scenario *B* because the distribution of Hadoop components is exactly the same in both scenarios.

Finally, scenario *D* exploits the capabilities of our deployment scheme by considering two workers on the same host. Note that this configuration could not be possible without our deployment scheme. Furthermore, our scheme would allow deploying more workers on any host in this scenario, but it is restricted to two workers to keep the same number of total workers and allow the performance comparison with the other scenarios.

4.2 | Platform configuration

Experiments are run at Lenovo premises on a cluster comprising 10 server hosts, each one with two Intel(R) Xeon(R) CPU E5-2697 v4 of 18 cores at 2.30GHz with Hyperthreading and 256 GB DDR4-2133 DIMM memory. For all the scenarios, each component (either in a container instance or not) stores its data in a *tmpfs* file system hosted in a 128 GB RAM disk. All the hosts run CentOS Linux release 7.6.1810 and are connected through 1 Gbps Ethernet. The MTU of the network devices has been reduced to 1462 and 1450 when deploying GRE and VXLAN tunnels, respectively.

We use Hadoop version 2.9.2, which is configured with the parameters shown in Table 2. Parameters that are not mentioned are set with their default value. To allow the comparison between the various scenarios, each of the Hadoop components (either master or worker) is run with 36 cores by means of the `numactl` command.

TABLE 2 Hadoop parameters.

File	Parameter	Value
core-site.xml	io.file.buffer.size	131072
hdfs-site.xml	dfs.blocksize	268435456
	dfs.replication	1
	dfs.namenode.handler.count	64
	dfs.client-write-packet-size	131072
yarn-site.xml	yarn.scheduler.minimum-allocation-mb	1024
	yarn.scheduler.maximum-allocation-mb	120832
	yarn.scheduler.minimum-allocation-vcores	1
	yarn.scheduler.maximum-allocation-vcores	36
	yarn.nodemanager.resource.memory-mb	120832
	yarn.nodemanager.resource.cpu-vcores	36
mapred-site.xml	mapreduce.framework.name	yarn
	mapreduce.task.io.sort.mb	400
	mapreduce.task.io.sort.factor	40
	mapreduce.reduce.shuffle.parallelcopies	10
	yarn.app.mapreduce.am.resource.mb	4000
	yarn.app.mapreduce.am.command-opts	-Xmx3200m
	mapreduce.map.output.compress	true
	mapreduce.map.output.compress.codec	org.apache.hadoop.io.compress.SnappyCodec

Singularity 2.x experiments run version 2.6.0. Singularity 3.x experiments run version 3.1.0. Docker experiments run version 18.09.7. Multi-host overlay networking with Docker is deployed without using Docker Swarm by configuring an external etcd[10] discovery service.

Three Map-Reduce applications included in Hadoop, namely TeraGen, TeraSort, and TeraValidate are run as benchmarks. TeraGen deploys map tasks in order to generate a dataset that will be stored to the disk, so it is mostly disk intensive. TeraSort samples the generated data and deploys map and reduce tasks to sort it. This application is CPU and memory intensive. TeraValidate ensures by means of map and reduce tasks that the output of TeraSort is sorted. It is mostly disk intensive as it performs numerous read operations from the disk. TeraGen, TeraSort, and TeraValidate applications are configured with the parameters shown in Table 3. The number of map tasks in TeraGen is set to the number of logical cores in the cluster minus one (corresponding to the Hadoop ApplicationMaster). This results in 71 map tasks in the small-scale experiments and 359 in the large-scale ones. The number of reduce tasks in TeraSort is set to half the number of logical cores in the cluster minus one. This results in 35 map tasks in the small-scale experiments and 179 in the large-scale ones. Experiments at small scale run the applications with datasets of $[10, 20, 40]$ GB. Large-scale experiments use datasets of $[100, 200, 300]$ GB. Datasets are relatively small as our goal was not to evaluate the maximum performance of the platform but to do a comparative analysis.

[10]https://etcd.io

TABLE 3 Map-Reduce parameters for TeraGen/TeraSort/TeraValidate.

Benchmark	Parameter	Value (Small-scale)	Value (Large-scale)
TeraGen	mapreduce.job.maps	71	359
	mapreduce.map.memory.mb	2048	
	mapreduce.map.java.opts	-Xmx1638m	
	mapreduce.map.cpu.vcores	1	
TeraSort	mapreduce.job.reduces	35	179
	mapreduce.map.memory.mb	2048	
	mapreduce.reduce.memory.mb	3072	
	mapreduce.map.java.opts	-Xmx1638m	
	mapreduce.reduce.java.opts	-Xmx2457m	
	mapreduce.map.cpu.vcores	1	
	mapreduce.reduce.cpu.vcores	2	
TeraValidate	mapreduce.map.memory.mb	2048	
	mapreduce.reduce.memory.mb	3072	
	mapreduce.map.java.opts	-Xmx1638m	
	mapreduce.reduce.java.opts	-Xmx2457m	
	mapreduce.map.cpu.vcores	1	
	mapreduce.reduce.cpu.vcores	2	

4.3 | Performance comparison (small-scale testbed)

This section evaluates the Hadoop performance when running on a small-scale testbed with different deployment scenarios, namely *A*, *B*, and *C*. Figures 3,4,5 present the average CPU time (in seconds) and the standard error, which are calculated from the results of 10 runs, that TeraGen, TeraSort and TeraValidate benchmarks take to handle datasets of [10, 20, 40] GB.

These results allow first to evaluate the overhead of using containers. For that purpose, we have quantified the performance difference between the mean execution time of each scenario using containers (`B-xxx`) with respect to its bare-metal counterpart (`A-xxx`) and we have conducted statistical tests to assess whether this difference was statistically significant or due to randomness. We intended to use unpaired two-samples T-tests to compare the two means. According to this, we defined null hypotheses for each scenario describing a situation where there is no difference between the mean execution time of Hadoop in that scenario regarding the time in the baseline scenario. The lowest the p-value statistic of the T-test is, the most probable is that we can safely reject the null hypothesis that there is no difference between the means, and hence the performance difference is statistically significant. Otherwise, a high p-value does not allow rejecting the null hypothesis and thus the observed difference might be due to randomness. Typically, a threshold p-value of 0.05 is used to discriminate these two situations.

Classical unpaired two-samples T-tests require that the two groups of samples are normally distributed, so we had to verify that by using Shapiro-Wilk tests [17]. We saw that some groups of samples were not normally distributed. As a consequence, when

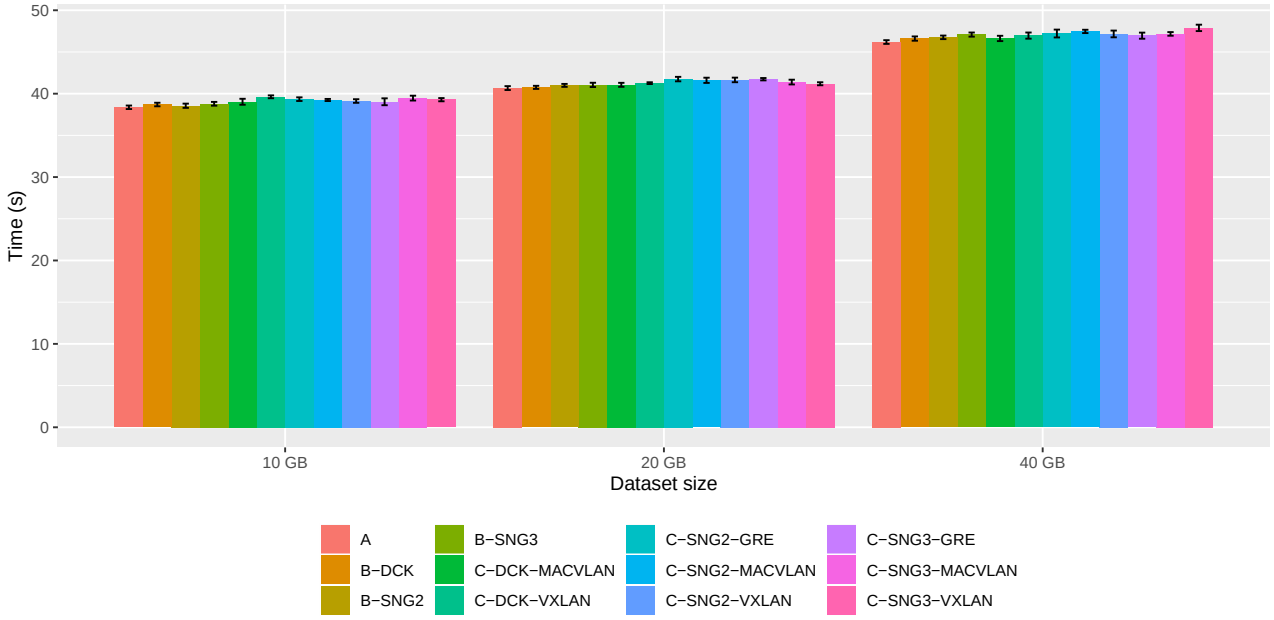

FIGURE 3 TeraGen application performance on deployment scenarios A, B, and C (small-scale testbed).

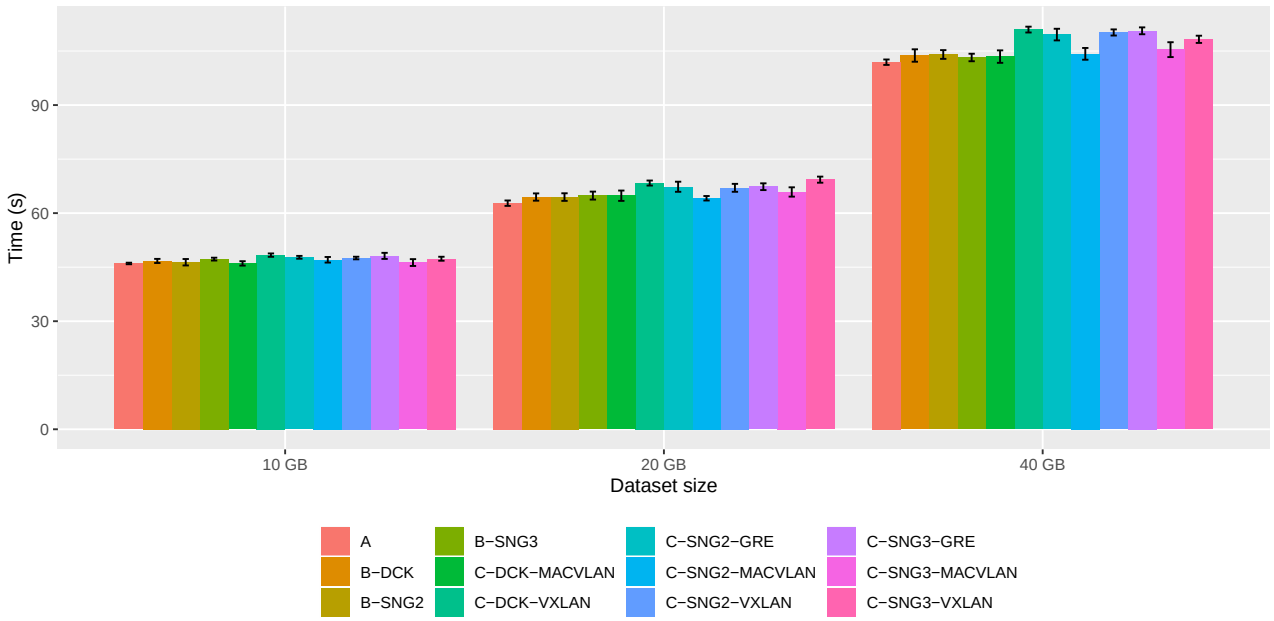

FIGURE 4 TeraSort application performance on deployment scenarios A, B, and C (small-scale testbed).

some of the groups of samples being compared were not normally distributed, we used the Wilcoxon rank sum test (a.k.a. Mann-Whitney test[18]) instead of the classical two-samples T-test. Unpaired two-samples T-tests also require that the variances of the two groups are equal. Again, we verified this by using Fisher's F-tests and saw that variances were not generally equal. As a consequence, in those cases we decided to use the Welch T-tests[19] instead of the classical T-tests.

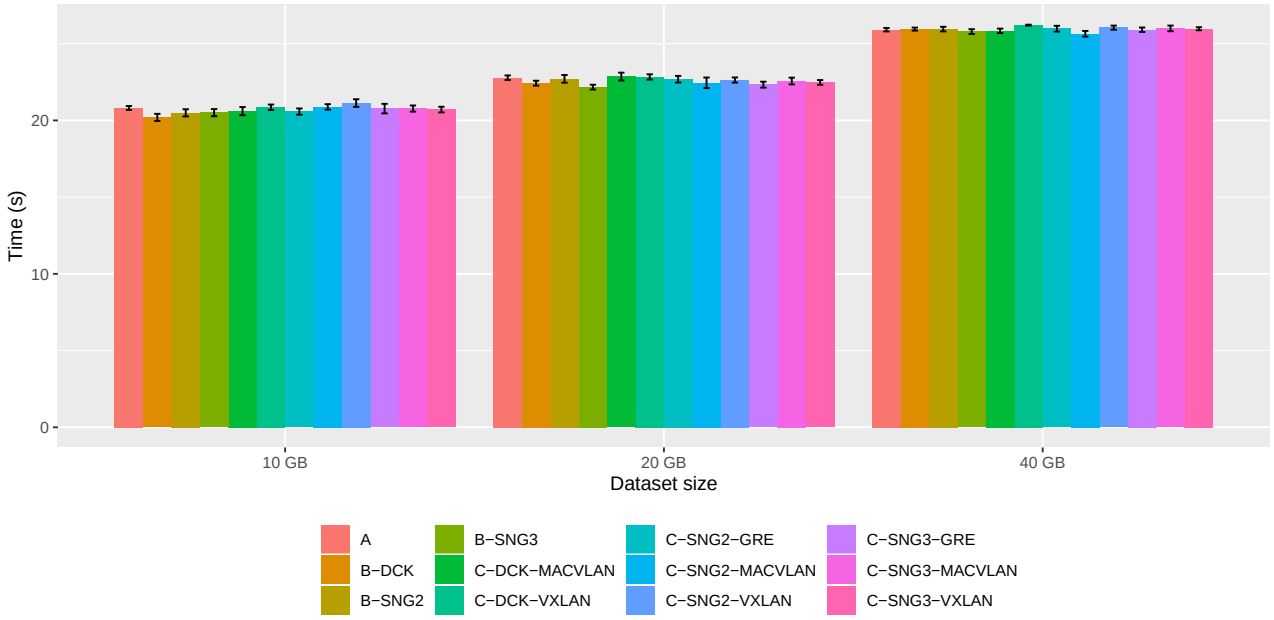

FIGURE 5 TeraValidate application performance on deployment scenarios A, B, and C (small-scale testbed).

TABLE 4 Performance using containers (`B-xxx`) with respect to its bare-metal counterpart (`A-xxx`).

Scenario	Benchmark	10 GB		20 GB		40 GB	
		% Diff	P-Value	% Diff	P-Value	% Diff	P-Value
B-DCK	TeraGen	0.86%	0.28	0.21%	0.77	0.90%	0.24
	TeraSort	1.53%	0.27	2.72%	0.19	1.82%	0.34
	TeraValidate	-2.95%	0.02	-1.57%	0.06	0.16%	0.68
B-SNG3	TeraGen	1.10%	0.19	0.94%	0.28	1.93%	0.02
	TeraSort	2.62%	0.02	3.36%	0.13	1.28%	0.32
	TeraValidate	-1.47%	0.19	-2.69%	0.009	-0.47%	0.53
B-SNG2	TeraGen	0.48%	0.58	0.80%	0.28	1.22%	0.09
	TeraSort	0.74%	0.72	2.71%	0.21	2.10%	0.15
	TeraValidate	-1.52%	0.63	-0.33%	0.68	0.16%	0.05

Table 4 shows the average performance of TeraGen, TeraSort, and TeraValidate when running on Singularity 2x, Singularity 3x, and Docker containers (`B-xxx`) with respect to its bare-metal counterpart (`A-xxx`). Although there are some performance differences in the mean values that range from -2% to 3%, in most cases those differences are not statistically significant, as the p-values of the corresponding t-tests are higher than 0.05. Therefore, we can conclude that running on container instances does not add significant overhead to the execution and in the few cases where it does, this overhead is very small.

We have also quantified the performance difference between the mean execution time of each overlay/underlay container interconnection scenario (`C-xxx`) with respect to its counterpart running containers but using the underlying network connecting the

TABLE 5 Performance of overlay/underlay container interconnection (`C-xxx`) with respect to its counterpart connecting containers through the underlying network in the server (`B-xxx`).

Scenario	Benchmark	10 GB		20 GB		40 GB	
		% Diff	P-Value	% Diff	P-Value	% Diff	P-Value
C-DCK-MACVLAN	TeraGen	0.86%	0.44	0.74%	0.33	0.03%	0.97
	TeraSort	-1.46%	0.42	0.57%	0.97	-0.29%	0.90
	TeraValidate	2.02%	0.26	1.92%	0.24	-0.45%	0.60
C-SNG3-MACVLAN	TeraGen	1.75%	0.08	0.82%	0.39	0.17%	0.81
	TeraSort	-2.02%	0.37	1.55%	0.56	2.11%	0.36
	TeraValidate	1.31%	0.28	1.79%	0.17	0.80%	0.85
C-SNG2-MACVLAN	TeraGen	1.79%	0.01	1.48%	0.03	1.52%	0.03
	TeraSort	1.47%	0.57	-0.51%	0.79	0.17%	0.93
	TeraValidate	1.88%	0.25	-1.13%	0.56	-1.17%	0.08
C-DCK-VXLAN	TeraGen	2.39%	0.003	1.24%	0.03	0.77%	0.44
	TeraSort	3.46%	0.04	6.03%	0.006	6.94%	0.002
	TeraValidate	3.28%	0.04	1.82%	0.03	0.98%	5e-04
C-SNG3-VXLAN	TeraGen	1.27%	0.11	0.33%	0.67	1.70%	0.10
	TeraSort	0.23%	0.88	6.82%	0.006	4.89%	0.03
	TeraValidate	0.97%	0.39	1.39%	0.08	0.71%	0.85
C-SNG2-VXLAN	TeraGen	1.47%	0.11	1.60%	0.12	0.87%	0.39
	TeraSort	2.51%	0.25	3.98%	0.11	5.88%	8e-04
	TeraValidate	3.12%	0.08	-0.33%	0.85	0.37%	0.82
C-SNG3-GRE	TeraGen	0.62%	0.62	1.66%	0.04	-0.28%	0.77
	TeraSort	1.91%	0.35	3.80%	0.11	7.18%	5e-05
	TeraValidate	1.29%	0.51	0.72%	0.53	0.46%	0.63
C-SNG2-GRE	TeraGen	2.05%	0.03	1.84%	0.03	0.98%	0.39
	TeraSort	2.96%	0.19	4.44%	0.12	5.31%	0.01
	TeraValidate	0.40%	0.85	-0.10%	0.94	0.10%	0.80

host servers (`B-xxx`). This evaluates the overhead of using an overlay network, either a GRE or a VXLAN tunnel, or the underlay network, through MACVLAN, for the communication between components. As we did before, we perform the corresponding statistical tests to assess whether the performance difference is statistically significant. Results are shown in Table 5.

When using MACVLAN networks, the performance differences in the mean values range from -2% to 2%, but they are not statistically significant, as the p-values of the corresponding t-tests are higher than 0.05. Only Singularity 2.x has some significant difference when running TeraGen, but it is only around 1.5%. We can then conclude that MACVLAN is not introducing significant overheads with respect to using the underlying network connecting the host servers.

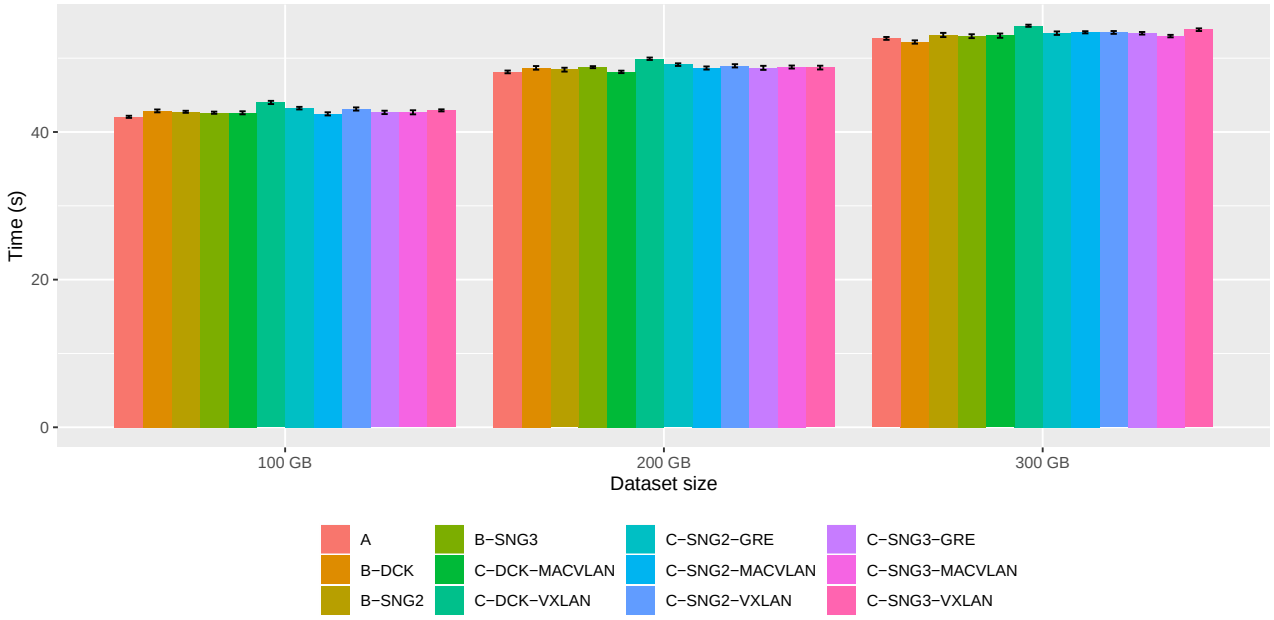

FIGURE 6 TeraGen application performance on deployment scenarios A, B, and C (large-scale testbed).

When using VXLAN overlay networks, TeraSort executions exhibit the most significant performances differences, introducing an overhead which could be up to 6-7%. The performance differences for TeraGen and TeraValidate executions are not statistically significant enough when running with Singularity 2.x and 3.x, but they could range from 1% to 3% when running with Docker. Results are similar when using GRE overlay networks, with significant overheads in TeraSort executions up to 5-7%. Some TeraGen executions also present significant overheads around 2%. We can conclude that overlay networking approaches based on packet encapsulation introduce some additional overhead (up to 7%), which is roughly the same for both GRE and VXLAN tunnels in the small-scale testbed comprising only two hosts.

4.4 | Scalability comparison (large-scale testbed)

This section evaluates the Hadoop scalability when running on a large-scale testbed with different deployment scenarios, namely *A*, *B*, and *C*. Figures 6,7,8 present the average CPU time (in seconds) and the standard error, which are calculated from the results of 10 runs, that TeraGen, TeraSort and TeraValidate benchmarks take to handle datasets of [100, 200, 300] GB.

As we did before, we evaluate first the overhead of using containers. For that purpose, we have quantified the performance difference between the mean execution time of each scenario using containers (`B-xxx`) with respect to its bare-metal counterpart (`A-xxx`) and we have performed the corresponding statistical tests to assess whether the performance difference is statistically significant. Results are shown in Table 6.

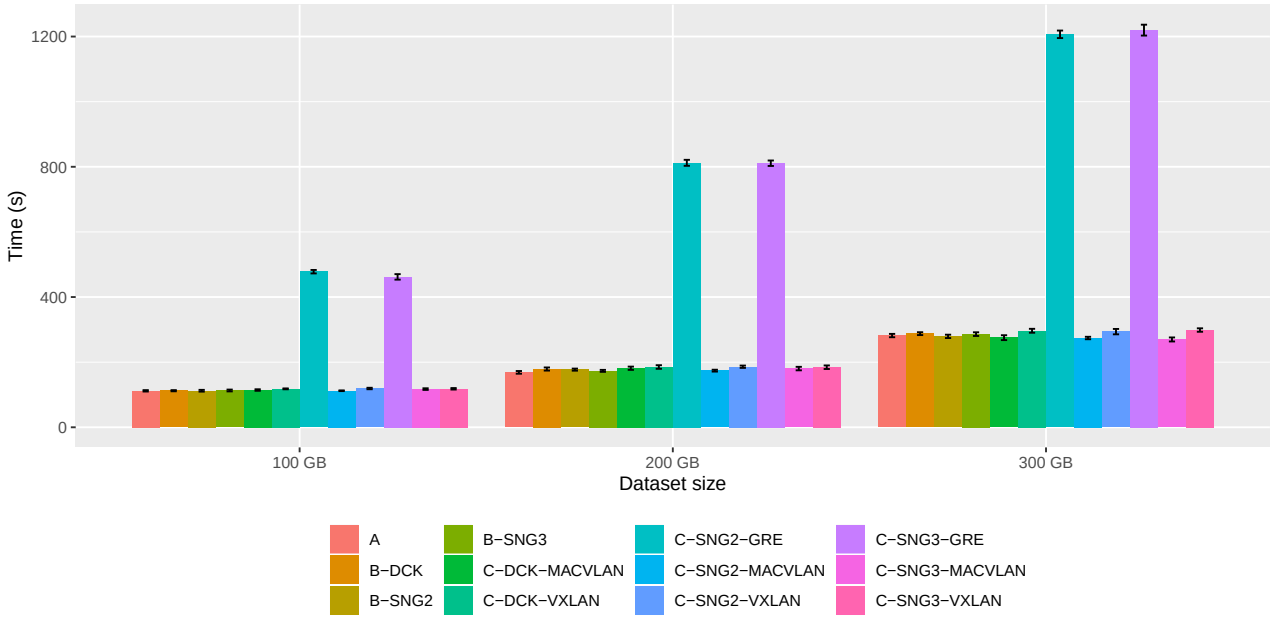

FIGURE 7 TeraSort application performance on deployment scenarios A, B, and C (large-scale testbed).

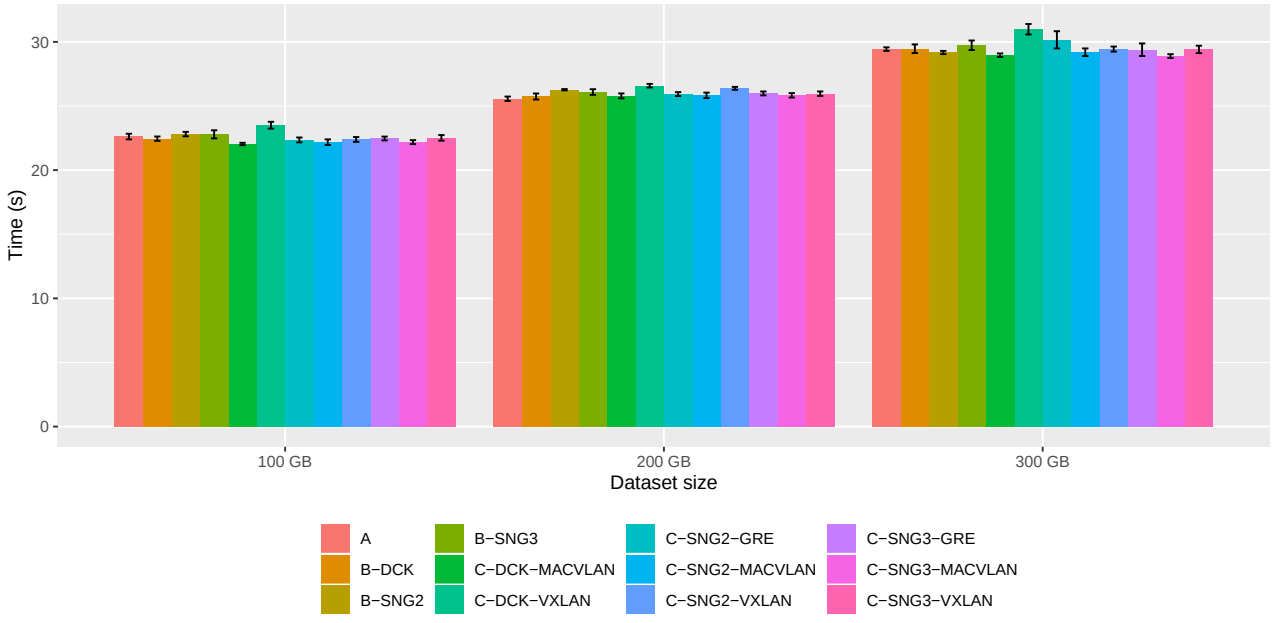

FIGURE 8 TeraValidate application performance on deployment scenarios A, B, and C (large-scale testbed).

There are some performance differences in the mean values that range from -1% to 6%. As in the small-scale experiments, many of those differences are not statistically significant. Only some TeraGen and TeraValidate executions have some significant overhead around 1-2%. Regarding TeraSort, although the p-values of the corresponding t-tests are higher than 0.05, some of them

TABLE 6 Performance using containers (`B-xxx`) with respect to its bare-metal counterpart (`A-xxx`) (large-scale testbed).

Scenario	Benchmark	100 GB		200 GB		300 GB	
		% Diff	P-Value	% Diff	P-Value	% Diff	P-Value
B-DCK	TeraGen	1.91%	0.004	1.14%	0.09	-0.93%	0.11
	TeraSort	0.33%	0.90	6.08%	0.12	2.08%	0.40
	TeraValidate	-0.73%	0.19	0.68%	1	0.11%	0.28
B-SNG3	TeraGen	1.32%	0.02	1.34%	0.01	0.57%	0.37
	TeraSort	0.93%	0.77	2.73%	0.38	1.55%	0.58
	TeraValidate	0.76%	0.53	2.05%	0.04	1.02%	0.71
B-SNG2	TeraGen	1.62%	0.003	0.63%	0.36	0.88%	0.19
	TeraSort	0.24%	0.94	4.98%	0.12	-0.88%	0.74
	TeraValidate	0.86%	0.82	2.76%	0.01	-0.89%	0.03

are quite low (around 0.1), which could indicate that the overhead when running on container instances has a higher probability to be significant in a large-scale experiments, especially with Docker.

We have also quantified the performance difference between the mean execution time of each overlay/underlay container interconnection scenario (`C-xxx`) with respect to its counterpart running containers but using the underlying network connecting the host servers (`B-xxx`). This evaluates the scalability in larger testbeds when using an overlay network, either a GRE or a VXLAN tunnel, or the underlay network, through MACVLAN, for the communication between components. As we did before, we have performed the corresponding statistical tests to assess whether the performance difference is statistically significant. Results are shown in Table 7.

As with the small-scale experiments, the performance differences when using MACVLAN networks are not statistically significant and we can conclude that MACVLAN can scale when running large-scale experiments.

When using VXLAN overlay networks, Docker exhibits significant performance differences in the executions of the three benchmarks, which range from 2% to 5%. Singularity 2.x and 3.c have better results, with TeraGen and TeraValidate showing not significant differences, or significant overhead around 1% only. TeraSort executions present performance differences from 4% to 6%. Although the p-value of those experiments is higher than 0.05, it is low enough to indicate that VXLAN might be introducing significant overhead also in large-scale experiments. In any case, this overhead is roughly the same than the overhead on the small-scale experiments, so we can conclude that VXLAN overlay networks can scale on large-scale experiments.

Results are similar when using GRE overlay networks for TeraGen and TeraValidate executions, that is, not significant differences or significant overhead around 1-1.5%. However, the performance of TeraSort executions degraded more than 300% independently of the dataset size for both Singularity 2.x and 3.x. This might be because GRE requires each host to establish

TABLE 7 Performance of overlay/underlay container interconnection (`C-xxx`) with respect to its counterpart connecting containers through the underlying network in the server (`B-xxx`) (large-scale testbed).

Scenario	Benchmark	100 GB		200 GB		300 GB	
		% Diff	P-Value	% Diff	P-Value	% Diff	P-Value
C-DCK-MACVLAN	TeraGen	-0.60%	0.38	-1.12%	0.08	1.66%	0.03
	TeraSort	1.84%	0.49	1.35%	0.73	-4.28%	0.17
	TeraValidate	-1.84%	0.32	0.18%	0.58	-1.71%	0.52
C-SNG3-MACVLAN	TeraGen	0.13%	0.86	0.03%	0.95	0.02%	0.97
	TeraSort	3.78%	0.27	4.04%	0.27	-5.72%	0.07
	TeraValidate	-2.66%	0.14	-0.98%	0.44	-2.84%	0.06
C-SNG2-MACVLAN	TeraGen	-0.65%	0.29	0.47%	0.44	0.69%	0.27
	TeraSort	-0.004%	0.99	-1.76%	0.47	-1.81%	0.44
	TeraValidate	-2.76%	0.08	-1.67%	0.04	0.07%	0.85
C-DCK-VXLAN	TeraGen	2.67%	0.001	2.54%	7e-04	4.22%	4e-07
	TeraSort	5.05%	0.02	3.47%	0.40	3.07%	0.25
	TeraValidate	4.68%	0.007	3.28%	7e-04	5.15%	7e-04
C-SNG3-VXLAN	TeraGen	0.75%	0.13	-0.12%	0.85	1.66%	0.01
	TeraSort	4.59%	0.17	6.45%	0.09	4.40%	0.12
	TeraValidate	-1.20%	0.49	-0.50%	0.85	-1.09%	0.50
C-SNG2-VXLAN	TeraGen	0.89%	0.16	1.07%	0.15	0.65%	0.33
	TeraSort	6.08%	0.07	5.04%	0.08	5.20%	0.16
	TeraValidate	-1.83%	0.35	0.41%	0.68	0.92%	0.04
C-SNG3-GRE	TeraGen	0.10%	0.87	-0.19%	0.76	0.74%	0.28
	TeraSort	308.68%	4e-13	367.96%	1e-05	326.35%	1e-14
	TeraValidate	-1.42%	0.47	-0.42%	0.91	-1.17%	0.41
C-SNG2-GRE	TeraGen	1.15%	0.04	1.41%	0.05	0.46%	0.52
	TeraSort	325.72%	3e-16	358.67%	1e-15	332.21%	2e-16
	TeraValidate	-2.00%	0.32	-1.29%	0.06	3.39%	0.08

point-to-point tunnels with the rest of hosts. While this was not a problem in the small-scale experiments with two hosts, it cannot scale when running on 10 hosts. As discussed previously, this can be avoided by using instead Multipoint Generic Routing Encapsulation (mGRE) supporting multipoint tunnels, or VXLAN in multicast mode (as we do in this paper).

A commonly suggested configuration improvement for the execution of Big Data applications is the use of Jumbo Frames, i.e. increasing the MTU of the physical network interfaces to 9000 bytes instead of the 1500 bytes sent by default on every Ethernet frame. This would reduce the number of individual frames that must be sent for a given amount of data and also the need to separate data blocks into multiple Ethernet frames[20], which could be helpful to reduce the performance impact of overlay networking approaches that include additional overhead on each Ethernet frame because of the encapsulation. The benefit of using Jumbo Frames is directly related with the size of the datasets. As the datasets used in this paper are not especially big, the

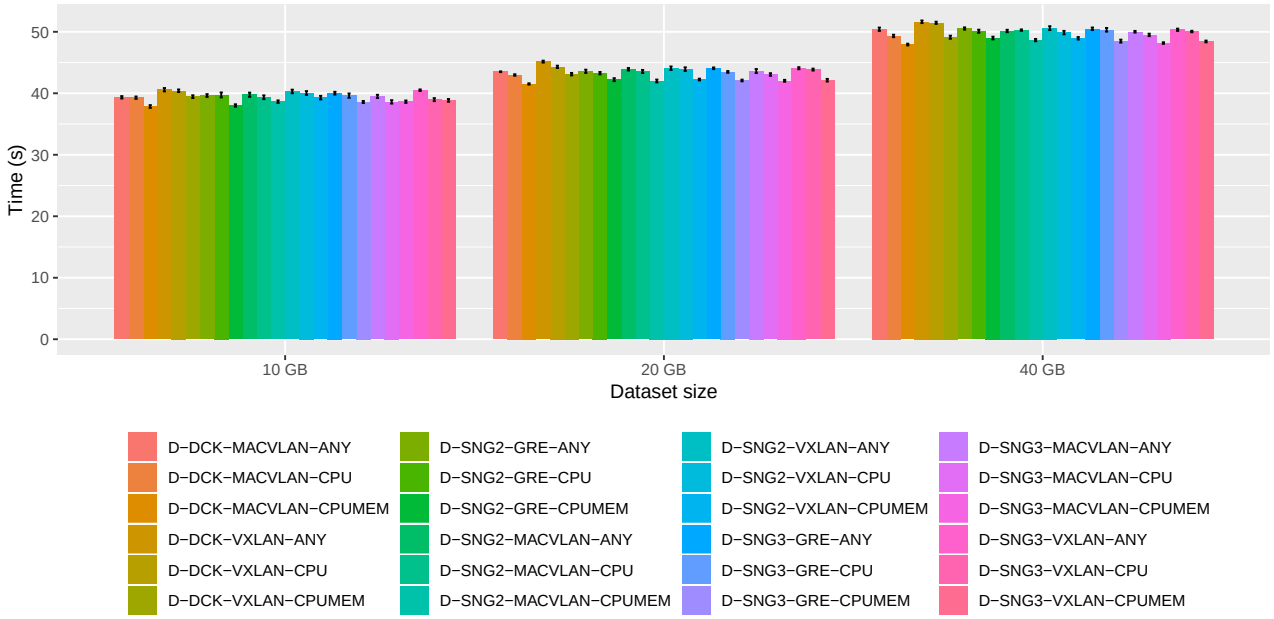

FIGURE 9 TeraGen application performance on deployment scenario D (small-scale testbed).

impact of this configuration change would not be very significant. As a reference, the performance improvement when using Jumbo frames to run TeraSort with a dataset of 100 GB has been reported to be only around 4%[21]. Because of this, we left this as future work.

4.5 | Impact of processor/memory affinity

This section assesses how the capabilities of our deployment scheme could be exploited to improve Hadoop performance. In particular, we evaluate different processor and memory affinity configurations on deployment scenario *D*, which can run two Hadoop workers on the same host server. The different affinity configurations determine what processors and memory sockets each worker can use, which could be used to exploit NUMA locality. In particular, we evaluated i) `ANY`: workers do not have any processor or memory affinity, hence the actual distribution is decided by the operating system, ii) `CPU`: we define specific processor affinity for each worker container on cores belonging to the two processor sockets available in the host, and iii) `CPUMEM`: we define specific processor and memory affinity for each worker container, each one on cores and memory modules belonging to a different processor socket. Figures 9,10,11 present the average CPU time (in seconds) and the standard error, which are calculated from the results of 10 runs, that TeraGen, TeraSort and TeraValidate benchmarks take to handle datasets of $[10, 20, 40]$ GB.

We have quantified the performance difference between the mean execution time of each scenario when running workers in the same host with several affinity configurations (`D-xxx`) with respect to its counterpart running a single worker per host (`C-xxx`).

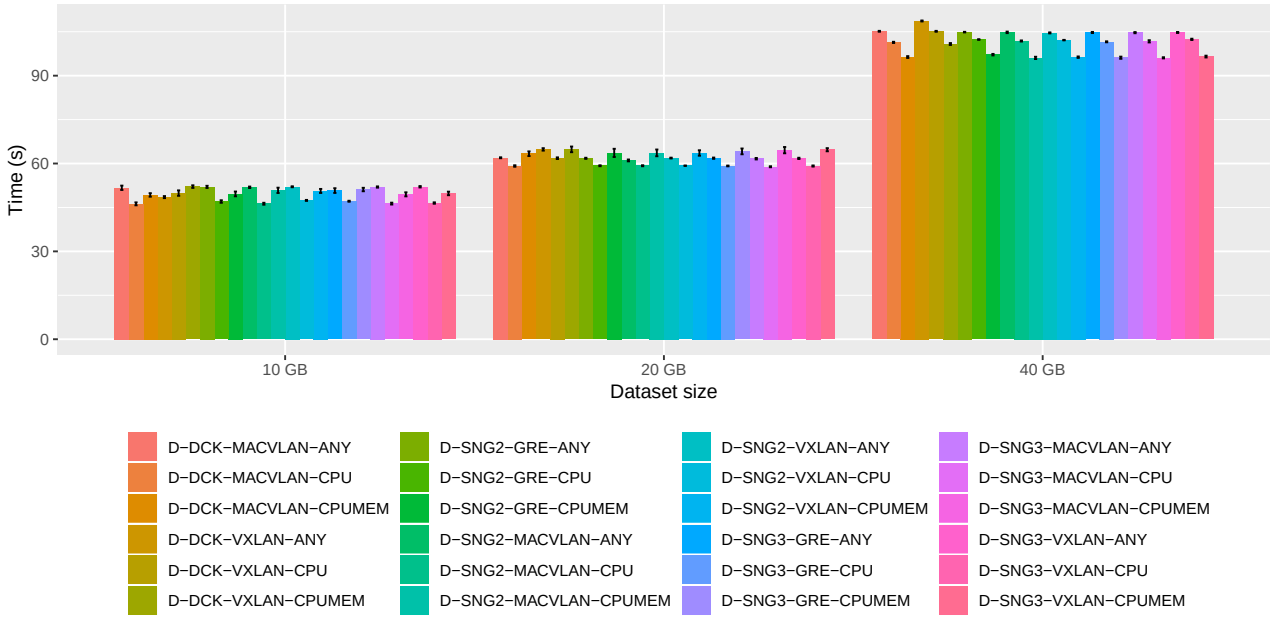

FIGURE 10 TeraSort application performance on deployment scenario D (small-scale testbed).

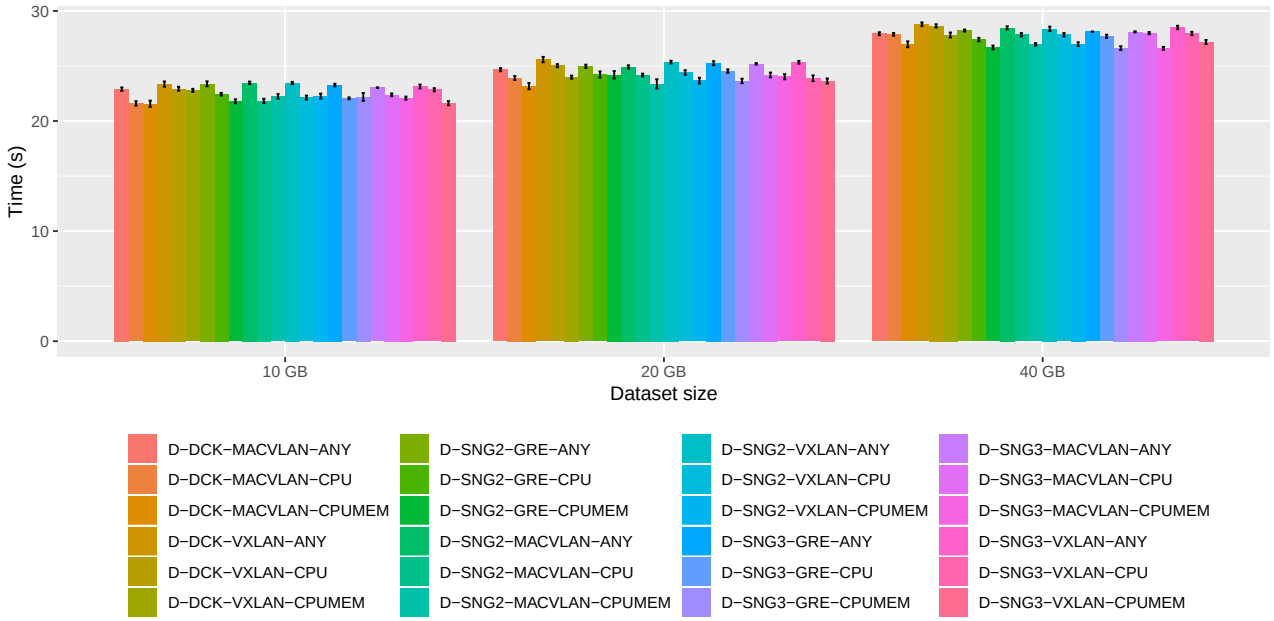

FIGURE 11 TeraValidate application performance on deployment scenario D (small-scale testbed).

This evaluates how the capabilities of our deployment scheme could be exploited to improve Hadoop performance by means of processor and memory affinity. As we did before, we have performed the corresponding statistical tests to assess whether the performance difference is statistically significant. Results when using MACVLAN, VXLAN and GRE networks are shown in Table 8, Table 9, and Table 10, respectively.

When running two workers in the same host with ANY affinity, there are significant performance differences in TeraGen and TeraValidate executions, with degradations ranging from 6% to 12% with respect to its counterpart running a single worker per host when using MACVLAN, and from 2% to 13% when using VXLAN or GRE. Results with TeraSort are mixed. Whereas there is significant performance degradation up to 12% with smaller datasets (10 GB), there are noticeable performance improvements up to 11% with 20 GB datasets, and up to 5% with 40 GB. This seems to indicate that this application can benefit from further improvements when running two workers in the same host with other affinity configurations.

When running two workers in the same host with CPU affinity, the observed performance degradation with TeraGen and TeraValidate has been reduced slightly, ranging from 4% to 8% with respect to its counterpart running a single worker per host when using MACVLAN, and from 2% to 10% when using VXLAN or GRE. TeraSort results have also reduced considerably the performance degradation with smaller datasets (10 GB) so that they do not present statistically significant difference with their single worker counterparts, while the performance improvement has increased up to 14% with 20 GB datasets, and up to 8% with 40 GB datasets. This confirms that TeraSort can benefit from processor affinity, a capability that we could enable thanks to our deployment scheme.

When running two workers in the same host with CPUMEM affinity, TeraGen performance is improved, being able to outperform its counterpart running a single worker per host up to 3% with 10 GB datasets while reducing the degradation with bigger datasets (40 GB) to not more than 4%. TeraValidate behavior still exhibits performance degradation ranging from 2% to 9%, which indicate that it cannot benefit much from memory affinity. Results with TeraSort are mixed again. The performance degradation with 10 GB datasets has increased again, ranging from 5% to 8%. Results with 20 GB datasets are not statistically different regarding their counterpart running a single worker per host when using MACVLAN, but they still show a performance improvement around 5-6% when using VXLAN and GRE. However, results with 40 GB datasets show their best values with performance improvements up to 13%. This shows that when the dataset size is big enough, TeraSort can also take profit of memory affinity, enabled thanks to our deployment scheme, to obtain significant performance speedup.

5 | RELATED WORK

5.1 | Virtualization and containerization for Big Data

VMware has worked in virtualizing the Hadoop framework and evaluating its performance when deployed on a fixed number of servers and with different number of hosted VMs on ESXi hypervisors[1]. Results exhibit some guidelines to deploy VMs while keeping as good performance as when using a bare-metal infrastructure. In addition, VMware started to deploy some efforts expanding upon their previous work while studying the overhead of virtualization in a recent cluster[2], while studying Hadoop

TABLE 8 Performance of MACVLAN underlay interconnection of containers when running workers in the same host with `ANY` (`D-xxx-MACVLAN-ANY`), `CPU` (`D-xxx-MACVLAN-CPU`), and `CPUMEM` (`D-xxx-MACVLAN-CPUMEM`) affinities with respect to its counterpart running a single worker per host (`C-xxx-MACVLAN`).

Scenario	Benchmark	10 GB		20 GB		40 GB	
		% Diff	P-Value	% Diff	P-Value	% Diff	P-Value
D-DCK-MACVLAN-ANY	TeraGen	0.81%	0.45	5.96%	2e-06	8.15%	5e-08
	TeraSort	12.30%	1e-04	-4.43%	0.53	1.64%	0.35
	TeraValidate	11.11%	2e-05	7.91%	1e-05	8.09%	1e-05
D-SNG3-MACVLAN-ANY	TeraGen	0.06%	0.95	5.37%	8e-05	6.02%	1e-08
	TeraSort	12.29%	2e-04	-6.42%	0.02	-0.66%	0.74
	TeraValidate	10.87%	1e-05	11.64%	1e-05	8.10%	5e-07
D-SNG2-MACVLAN-ANY	TeraGen	1.33%	0.03	5.55%	1e-05	5.62%	2e-08
	TeraSort	10.26%	1e-04	-4.79%	7e-04	0.54%	0.97
	TeraValidate	12.40%	1e-05	10.93%	4e-05	11.00%	1e-05
D-DCK-MACVLAN-CPU	TeraGen	0.65%	0.54	4.67%	7e-06	5.81%	3e-06
	TeraSort	0.40%	0.82	-8.77%	1e-05	-2.02%	0.48
	TeraValidate	4.84%	0.003	4.58%	0.007	7.88%	1e-05
D-SNG3-MACVLAN-CPU	TeraGen	-2.19%	0.04	4.03%	2e-04	4.94%	3e-07
	TeraSort	0.07%	0.91	-10.60%	4e-04	-3.47%	0.11
	TeraValidate	7.70%	1e-05	7.16%	3e-04	7.65%	1e-05
D-SNG2-MACVLAN-CPU	TeraGen	0.28%	0.62	4.68%	1e-04	5.91%	5e-09
	TeraSort	-1.67	0.37	-7.63%	2e-05	-2.28%	0.48
	TeraValidate	4.54%	0.004	7.70%	6e-04	8.56%	1e-05
D-DCK-MACVLAN-CPUMEM	TeraGen	-3.02%	0.01	1.12%	0.22	2.80%	0.003
	TeraSort	7.00%	0.001	-2.30%	0.85	-6.86%	0.003
	TeraValidate	4.66%	0.02	1.37%	0.39	4.39%	0.002
D-SNG3-MACVLAN-CPUMEM	TeraGen	-2.11%	0.03	1.55%	0.10	2.13%	0.001
	TeraSort	6.90%	0.01	-1.98%	0.68	-8.82%	0.001
	TeraValidate	6.28%	3e-04	6.45%	0.007	2.32%	0.007
D-SNG2-MACVLAN-CPUMEM	TeraGen	-1.54%	0.02	0.96%	0.35	2.50%	7e-04
	TeraSort	8.05%	0.009	-0.78%	0.71	-7.83%	7e-04
	TeraValidate	6.53%	7e-04	4.12%	0.11	5.12%	3e-04

and Spark. The study shows that the best performance is obtained when deploying one VM per NUMA node on individual hosts because it actually enables the VMs to perform faster memory accesses.

Intel, on the other hand, oriented its work on containerization. It notably currently proposes a software platform called BlueData EPIC which makes use of Docker containers to deploy Big Data applications. In a recent publication[4], Intel presents its benchmark study comparing the performance of Hadoop when deployed on their platform and on a bare-metal infrastructure. Results show that the performance is similar, and in some cases (especially when using 10 nodes in the cluster) it is even better using BlueData EPIC (increasing the number of nodes in the cluster tends to bring the performance of both cases closer).

TABLE 9 Performance of VXLAN overlay interconnection of containers when running workers in the same host with `ANY` (`D-xxx-VXLAN-ANY`), `CPU` (`D-xxx-VXLAN-CPU`), and `CPUMEM` (`D-xxx-VXLAN-CPUMEM`) affinities with respect to its counterpart running a single worker per host (`C-xxx-VXLAN`).

Scenario	Benchmark	10 GB		20 GB		40 GB	
		% Diff	P-Value	% Diff	P-Value	% Diff	P-Value
D-DCK-VXLAN-ANY	TeraGen	2.42%	0.009	9.42%	2e-12	9.93%	2e-08
	TeraSort	0.39%	0.75	-5.17%	6e-04	-2.06%	0.02
	TeraValidate	11.95%	1e-05	12.03%	1e-05	9.84%	1e-05
D-SNG3-VXLAN-ANY	TeraGen	3.10%	8e-05	7.03%	1e-9	5.11%	7e-05
	TeraSort	9.97%	4e-06	-10.88%	5e-06	-3.24%	0.007
	TeraValidate	11.74%	1e-05	12.72%	1e-05	9.71%	2e-04
D-SNG2-VXLAN-ANY	TeraGen	3.02%	0.005	5.85%	1e-04	7.29%	4e-06
	TeraSort	9.52%	5e-08	-7.70%	0.001	-5.04%	9e-05
	TeraValidate	11.00%	2e-06	11.99%	1e-05	8.95%	1e-05
D-DCK-VXLAN-CPU	TeraGen	1.99%	0.01	7.36%	1e-09	9.58%	1e-05
	TeraSort	3.21%	0.39	-9.57%	2e-06	-5.24%	1e-05
	TeraValidate	9.91%	1e-05	9.60%	1e-05	9.29%	1e-05
D-SNG3-VXLAN-CPU	TeraGen	-0.80%	0.33	6.47%	2e-09	4.51%	2e-04
	TeraSort	-1.77%	0.19	-14.64%	4e-07	-5.41%	2e-04
	TeraValidate	10.30%	1e-05	6.24%	7e-04	7.62%	2e-04
D-SNG2-VXLAN-CPU	TeraGen	2.36%	0.03	5.48%	0.0002	5.75%	5e-05
	TeraSort	-0.28%	0.75	-11.59%	6e-05	-7.28%	5e-06
	TeraValidate	4.71%	0.005	7.86%	1e-05	6.87%	1e-05
D-DCK-VXLAN-CPUMEM	TeraGen	-0.44%	0.545	4.48%	2e-06	4.59%	2e-04
	TeraSort	7.80%	2e-05	-5.16%	0.007	-9.15%	8e-08
	TeraValidate	9.25%	1e-05	5.04%	0.002	6.13%	2e-04
D-SNG3-VXLAN-CPUMEM	TeraGen	-1.10%	0.17	2.27%	0.004	1.13%	0.21
	TeraSort	5.17%	0.008	-6.61%	4e-04	-10.84%	3e-07
	TeraValidate	4.42%	0.03	5.14%	0.001	4.63%	0.002
D-SNG2-VXLAN-CPUMEM	TeraGen	0.45%	0.63	1.41%	0.04	3.78%	0.002
	TeraSort	6.52%	0.002	-5.12%	0.03	-12.56%	1e-08
	TeraValidate	5.35%	0.003	4.57%	0.007	3.60%	0.002

Moreover, containers are studied by Zhang et al.[7], who propose a method for the automatic configuration of Hadoop jobs hosted in Docker containers. The overhead of using containers hosting Big Data applications is studied also by Xavier et al.[22]. They present the case of a MapReduce based benchmark and compare the benchmark performance when being deployed on a bare-metal infrastructure, LXC containers, Linux-VServer containers, and OpenVZ containers. The study shows that LXC containers provide the smaller overhead. Nevertheless, the authors state that future work still needs to inquire more about performance isolation of the solution, especially at the network level.

TABLE 10 Performance of GRE overlay interconnection of containers when running workers in the same host with `ANY` (`D-xxx-GRE-ANY`), `CPU` (`D-xxx-GRE-CPU`), and `CPUMEM` (`D-xxx-GRE-CPUMEM`) affinities with respect to its counterpart running a single worker per host (`C-xxx-GRE`).

Scenario	Benchmark	10 GB		20 GB		40 GB	
		% Diff	P-Value	% Diff	P-Value	% Diff	P-Value
D-SNG3-GRE-ANY	TeraGen	2.53%	0.05	5.59%	7e-10	7.53%	7e-07
	TeraSort	5.46%	0.03	-8.19%	2e-04	-5.31%	1e-04
	TeraValidate	12.00%	1e-05	13.09%	2e-09	8.50%	2e-04
D-SNG2-GRE-ANY	TeraGen	0.76%	0.35	4.38%	1e-04	7.04%	3e-05
	TeraSort	8.97%	3e-07	-8.23%	0.003	-4.29%	0.02
	TeraValidate	13.62%	1e-05	10.09%	1e-05	8.61%	1e-05
D-SNG3-GRE-CPU	TeraGen	1.50%	0.30	4.17%	9e-08	7.19%	1e-06
	TeraSort	-2.11%	0.27	-12.13%	1e-05	-8.19%	4e-06
	TeraValidate	6.23%	0.002	9.88%	8e-08	6.89%	2e-04
D-SNG2-GRE-CPU	TeraGen	1.00%	0.41	3.66%	3e-04	6.10%	1e-04
	TeraSort	-1.50%	0.24	-12.01%	3e-04	-6.60%	0.002
	TeraValidate	9.04%	1e-05	6.79%	4e-04	5.47%	2e-05
D-SNG3-GRE-CPUMEM	TeraGen	-1.15%	0.34	0.84%	0.12	3.26%	0.003
	TeraSort	6.20%	0.009	-4.74%	0.03	-13.10%	1e-08
	TeraValidate	6.82%	0.008	5.86%	2e-04	2.76%	0.01
D-SNG2-GRE-CPUMEM	TeraGen	-3.36%	3e-04	1.12%	0.21	3.77%	0.005
	TeraSort	3.93%	0.06	-5.49%	0.08	-11.32%	3e-05
	TeraValidate	5.92%	0.001	6.68%	0.003	2.70%	0.08

5.2 | Interconnection of containers across hosts

Enabling the interconnection of containers across hosts is challenging due to the large number of networking technologies available. To address this challenge, the Cloud Native Computing Foundation proposed a driver-based network model, where the container runtime can offload the configuration of containers networking to network specific drivers. They released the Container Network Interface (CNI), which consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers. The specification defines a JSON schema that defines the inputs and outputs expected of a CNI plugin and provides a clear separation of concerns for the container runtime and the CNI plugin.

They also released CNI plugins for a variety of basic networks such as bridge, ipvlan, or macvlan. However, more sophisticated networks, such as overlay networks, require the use of third-party tools compliant with the CNI specification, such as Flannel, Calico[11], and Weave[12], or configuring the overlay network by means of our deployment scheme, which enables simple and easy deployments without making use of third-party software.

[11] https://github.com/projectcalico/calico-cni
[12] https://github.com/weaveworks/weave

Apart from Singularity, CNI is also supported by other popular container runtimes, such as CoreOS rkt [13], and container orchestrators, such as Kubernetes[14] or Apache Mesos[15]. On the other side, Docker implements its own networking specification, so-called the Container Network Model (CNM)[16], which supports multi-host networking through both underlay (based on MACVLAN) and overlay native drivers.

Docker implements overlay networks through VXLAN tunnels in a conceptually similar scheme to the one proposed in this paper, although it can also integrate with third-party tools such as Calico and Weave. Initially, overlay networking with Docker required an external key-value store such as etcd or Consul[17]. The key-value store is used to hold information about the network state which includes discovery, networks, endpoints, IP addresses[18]. Later, Docker integrated all the overlay networking functionality within Docker Swarm [19], which enables to deploy services into containers and scale them automatically. We used the former approach in this paper, as it allows the user to be still in charge of the placement of the different components of a Big Data application, similarly to our deployment scheme.

6 | CONCLUSIONS AND FUTURE WORK

This paper has presented a deployment scheme aimed at making Big Data applications deployable on Singularity container instances, so that they can benefit from Singularity assets related to ease of scalability and security. The scheme makes also possible the deployment of Big Data applications into HPC infrastructures proposing to deliver resources through Singularity container instances. It is based on network namespaces, and by this means it enables to assign isolated network resources to container instances and notably provide them with their own IP addresses and ports. In order to interconnect container instances together, the deployment scheme supports both overlay and underlay networking approaches.

Three Big Data applications included in Hadoop are have been run and their performance has been compared for different deployment scenarios. Our results showed that in most cases running on container instances does not add significant overhead to the execution, that using underlay networks based on MACVLAN does not introduce significant overheads with respect to using the underlying network connecting the hosts, that overlay networks based on multicast VXLAN tunnels can introduce an additional overhead up to 7% in both small- and large-scale experiments, and that overlay networks based on point-to-point GRE tunnels introduce roughly the same overhead than VXLAN in small-scale experiments, but they degrade the performance more than 300% in large-scale experiments.

[13] https://coreos.com/blog/rkt-cni-networking.html
[14] https://kubernetes.io/docs/admin/network-plugins/
[15] https://github.com/apache/mesos/blob/master/docs/cni.md
[16] https://github.com/docker/libnetwork/blob/master/docs/design.md
[17] https://www.consul.io/
[18] https://docs.docker.com/network/overlay-standalone.swarm/
[19] https://docs.docker.com/network/overlay/

This confirms Singularity as a good candidate to deploy Big Data applications on containers and suggests that underlay approaches based on MACVLAN are preferred from a performance perspective when the platform does not restrict its use (e.g., by limiting the number of different MAC addresses on a physical port). For the rest of cases, our work showed that overlay approaches based on multicast VXLAN could be an acceptable alternative with low overhead and good scalability.

Furthermore, our results also demonstrated that the capabilities of our deployment scheme can improve the performance of some applications by enabling them to exploit processor and memory affinity. In particular, we showed performance improvements of TeraSort up to 13% in some scenarios. However, results also showed that other application can suffer performance degradation when trying to exploit affinity, so this feature must be used with caution. In any case, our deployment scheme provides the required flexibility so that the users of Big Data applications can set the best deployment configuration for their specific use case.

Future works need to address the deployment of this scheme using bigger datasets, assessing to what extent the overhead introduced using overlay networks can be reduced, for instance, by using Jumbo Frames. It would also be interesting to study new experimental scenarios where more than two container instances per host are deployed. This would offer additional opportunities to play with the processor and memory affinity of each instance, but would introduce further competition among the instances for memory, disk, and network resources, which must be understood and managed adequately. Finally, the use of Multipoint Generic Routing Encapsulation (mGRE) to reduce the performance degradation of overlay networks based on GRE in large-scale testbeds should be also evaluated.

ACKNOWLEDGMENT

This work was partially supported by Lenovo as part of Lenovo-BSC collaboration agreement, by the Spanish Government under contract TIN2015-65316-P, and by the Generalitat de Catalunya under contract 2017-SGR-1414.

References

1. Buell J. Virtualized Hadoop Performance with VMware vSphere 6 on High-Performance Servers. tech. rep., VMware; 2015.

2. Jaffe D. Fast virtualized Hadoop and Spark on all-flash disks. tech. rep., VMware; 2017.

3. Raj A, Kaur K, Dutta U, Sandeep VV, Rao S. Enhancement of Hadoop Clusters with Virtualization Using the Capacity Scheduler. In: *2012 Third International Conference on Services in Emerging Markets*; 2012: 50-57

4. P. M. Bare-metal performance for Big Data workloads on Docker Containers. tech. rep., Intel; 2017.

5. Ye K, Ji Y. Performance Tuning and Modeling for Big Data Applications in Docker Containers. In: *2017 International Conference on Networking, Architecture, and Storage (NAS)*; 2017: 1-6

6. Bhimani J, Yang Z, Leeser M, Mi N. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*; 2017: 1-7

7. Zhang R, Li M, Hildebrand D. Finding the Big Data Sweet Spot: Towards Automatically Recommending Configurations for Hadoop Clusters on Docker Containers. In: *2015 IEEE International Conference on Cloud Engineering*; 2015: 365-368

8. Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 2017; 12(5): 1-20. doi: 10.1371/journal.pone.0177459

9. Arango C, Dernat R, Sanabria J. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *CoRR* 2017; abs/1709.10140.

10. Fox G, Qiu J, Jha S, Ekanayake S, Kamburugamuve S. Big Data, Simulations and HPC Convergence. In: Rabl T, Nambiar R, Baru C, Bhandarkar M, Poess M, Pyne S., eds. *Big Data Benchmarking*Springer International Publishing; 2016; Cham: 3–17.

11. Panda DK, Lu X. HPC Meets Cloud: Building Efficient Clouds for HPC, Big Data, and Deep Learning Middleware and Applications. In: *Proceedings of the10th International Conference on Utility and Cloud Computing*UCC '17. ACM; 2017; New York, NY, USA: 189–190

12. Lu X, Shankar D, Gugnani S, Panda DKDK. High-performance design of apache spark with RDMA and its benefits on various workloads. In: *2016 IEEE International Conference on Big Data (Big Data)*; 2016: 253-262

13. Kamburugamuve S, Ramasamy K, Swany M, Fox G. Low Latency Stream Processing: Apache Heron with Infiniband & Intel Omni-Path. In: *Proceedings of the 10th International Conference on Utility and Cloud Computing*UCC '17. ACM; 2017; New York, NY, USA: 101–110

14. Hanks S, Li T, Farinacci D, Traina P. Generic Routing Encapsulation (GRE). RFC 1701, RFC Editor; 1994.

15. Mahalingam M, Dutt D, Duda K, et al. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, RFC Editor; 2014.

16. Mehta R. VXLAN Performance Evaluation on VMware vSphere 5.1. tech. rep., VMware; 2013.

17. Shapiro SS, Wilk MB. An analysis of variance test for normality (complete samples). *Biometrika* 1965; 52(3-4): 591-611. doi: 10.1093/biomet/52.3-4.591

18. Mann HB, Whitney DR. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 1947; 18(1): 50–60. doi: 10.1214/aoms/1177730491

19. Welch BL. The Generalization of Student's Problem When Several Different Population Variances Are Involved. *Biometrika* 1947; 34(1-2): 28-35. doi: 10.1093/biomet/34.1-2.28

20. Trivedi M, Nambiar R. Lessons Learned: Performance Tuning for Hadoop Systems. In: Nambiar R, Poess M., eds. *Performance Evaluation and Benchmarking. Traditional - Big Data - Internet of Things*Springer International Publishing; 2017; Cham: 121–141.

21. Prakash P, Lee M, Hu Y, Kompella R, Wang J, Dassarma S. Jumbo Frames or Not: That is the Question!. Tech. Rep. 1770, Purdue University; 2013.

22. Xavier MG, Neves MV, Rose CAFD. A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*; 2014: 299-306

How to cite this article: Sauvanaud C, Dholakia A, Guitart J, Kim C, and Mayes P (2019), Big Data Deployment in Containerized Infrastructures through the Interconnection of Network Namespaces, *Softw Pract Exper.*, *2019;00:1–35*.