An Architecture Model for a Distributed Virtualization System

Pablo Pessolani
Facultad Regional Santa Fe
Universidad Tecnológica Nacional
Santa Fe – Argentina
e-mail: ppessolani@frsf.utn.edu.ar

Fernando G. Tinetti
III-LIDI Facultad de Informática- UNLP
Comisión de Inv. Científicas, Prov. Bs. As
La Plata, Argentina
e-mail: fernando@info.unlp.edu.ar

Toni Cortes
Barcelona Supercomputing Center & UPC
Barcelona – España
e-mail: toni.cortes@bsc.es

Silvio Gonnet
INGAR - Facultad Regional Santa Fe
CONICET - Universidad Tecnológica Nacional
Santa Fe - Argentina
e-mail: sgonnet@santafe-conicet.gov.ar

Abstract — **This article presents an architecture model for a Distributed Virtualization System, which could expand a virtual execution environment from a single physical machine to several nodes of a cluster. With current virtualization technologies, computing power and resource usage of Virtual Machines (or Containers) are limited to the physical machine where they run. To deliver high levels of performance and scalability, cloud applications are usually partitioned in several Virtual Machines (or Containers) located on different nodes of a virtualization cluster. Developers often use that processing model because the same instance of the operating system is not available on each node where their components run. The proposed architecture model is suitable for new trends in software development because it is inherently distributed. It combines and integrates Virtualization and Distributed Operating Systems technologies with the benefits of both worlds, providing the same isolated instance of a Virtual Operating System on each cluster node. Although it requires the introduction of changes in existing operating systems, thousands of legacy applications would not require modifications to obtain their benefits. A Distributed Virtualization System is suitable to deliver high-performance cloud services with provider-class features, such as high-availability, replication, migration, and load balancing. Furthermore, it is able to concurrently run several isolated instances of different guest Virtual Operating Systems, allocating a subset of nodes for each instance and sharing nodes between them. Currently, a prototype is running on a cluster of commodity hardware provided with two kinds of Virtual Operating Systems tailored for internet services (web server) as a proof of concept.**

Keywords: Virtualization, Virtual Machines, Containers, Distributed Operating Systems.

I. INTRODUCTION

Current virtualization technologies are massively adopted to cover those requirements in which Operating Systems (OS) have shown weakness, such as performance, fault, and security isolation. They also add features like resource partitioning, server consolidation, legacy application support, management tools, among others, which are attractive to Cloud service providers.

Nowadays, there are several virtualization technologies used to provide Infrastructure as a Service (IaaS) mounted in a cluster of servers linked by high-speed networks. Storage Area Networks (SAN), security appliances (network and application firewall, Intrusion Detection/Prevention Systems, etc.), and a set of management systems complement the required provider-class infrastructure.

Hardware virtualization, paravirtualization, and OS-level virtualization are the most widely used technologies to carry out these tasks, although each of them presents different levels of server consolidation, performance, scalability, high-availability, and isolation.

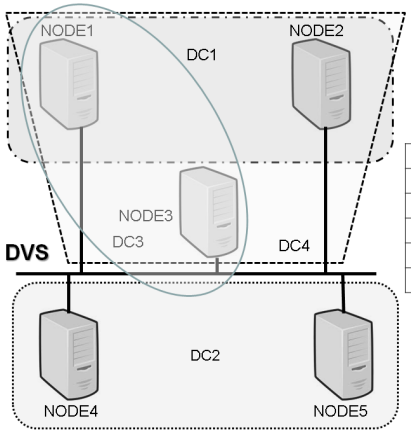
The term "*Virtual Machine*" (VM) is used in issues related to hardware virtualization and paravirtualization technologies to describe an isolated execution environment for an OS and its applications. *Containers*, *Jails*, *Zones* are the names used in OS-level virtualization to describe the environments for applications confinement. Regardless of the definition of the virtualization abstraction, its computing power and resource usage are limited to the physical machine where it runs.

Current IaaS providers use SANs in their Data Centers for storage virtualization, supplying disk drives for VMs. In some way, the resources (disks) of a VM expand outside the host and this can be seen as an exception to the above statement. If this processing mode is extended to several types of services and resources, it becomes a new model of distributed processing in virtualization technologies. The proposed architecture model takes this approach, distributing processes, services, and resources to provide virtual environments based on OS factoring and OS containers. The outcome is a Distributed Virtualization System (DVS), which combines and integrates OS-virtualization and Distributed Operating Systems (DOS) technologies, providing the same isolated instance of a Virtual Operating System (VOS) [1] on each node of a virtualization cluster.

Nowadays, to deliver high performance and scalability levels, Cloud applications are usually partitioned in several

VMs/Containers, running on the nodes of a virtualization cluster (as Docker-enabled applications) [2]. Developers often use those kinds of processing models as Platform as a Service (PaaS) because the same instance of the OS is not available on all nodes and, hence, they must use some kind of middleware, which provides the cluster with Application Programming Interfaces (APIs) and services. A DVS is suitable as infrastructure for this new trend in software development, like applications based on microservices architecture (MSA) [3], because it is inherently distributed. Furthermore, thousands of legacy applications would benefit because they would not require modifications to take advantage of DVS features. Migration of legacy applications from on-premises servers to a Cloud execution environment requires changes in their design and coding. If a standard interface, such as POSIX is available in the Cloud, the migration task is simplified by reducing costs and time.

A DVS fits the requirements for delivering high-performance cloud services with provider-class features as high-availability, replication, elasticity, load balancing, resource management, and process migration. Furthermore, a DVS is able to run several instances of different guest VOS concurrently, allocating a subset of nodes for each instance (resource aggregation), and to share nodes between them (resource partitioning). Each VOS runs isolated within a Distributed Container (DC), which could span multiple nodes of the DVS cluster as it is presented in the topology example in Fig.1. The proposed model keeps the appreciated features of current virtualization technologies, such as confinement, consolidation and security, and the benefits of DOS, such as transparency, greater performance, high-availability, elasticity, and scalability.



	DC1	DC2	DC3	DC4
NODE1	X		X	X
NODE2	X			X
NODE3			X	X
NODE4		X		
NODE5		X		

Figure 1. DVS topology example.

A DVS allows running multiple Distributed VOSs as guests that can extend beyond the limits of a physical machine. Each DVOS could have more computing power and could provide greater scalability and elasticity in its configuration as a consequence of resource and computing power aggregation. The set of resources (both physical and abstract) and the set of processes that constitute a DVOS can be scattered (and eventually replicated) in the nodes of a cluster.

This work is intended to contribute proposing a new model of virtualization that allows building several isolated execution environments that take advantage of the aggregation of computational, storage, and network resources of the nodes of a cluster.

The use of a DVS is based on the same arguments/grounds for the use of DOSs. Several related processes (in the same DC) could be executed in different nodes using the same abstract resources as those offered by the VOS. This feature simplifies application (or library) programming since standard APIs, such as operations on semaphores, message queues, mutexes, etc. can be used. On the other hand, the process location transparency is helpful for application administrators since it avoids dealing with IP addresses, ports, URLs, etc., simplifying applications deployment and management, and reducing costs and implementation times.

Let us suppose a configuration of a database server (DBMS) running on a host (or VM), and the need to perform an online backup which ensures consistency of a restored database. The backup process should run on a host (or VM), other than the DBMS for performance reasons, but connected to the same network, and both processes connected to the same SAN. As each process runs on its own OS, the DBMS process and the backup process must communicate using an ad-hoc protocol through the network in order to synchronize the access to the database. This requires setting up IP addresses, ports, names, etc. to describe the topology.

In a DVS configuration, both processes (DBMS and backup) could run on the same VOS, but on different nodes. Therefore, they can synchronize the access to the database using semaphores, mutexes, signals or any other facilities offered by the VOS.

The remainder of this paper is organized as follows. Section II explains background and related works. Section III describes the proposed architecture model, its design, and details of a prototype implementation. Section IV presents performance results of several components of the prototype. Finally, the conclusions of this contribution and future work are summarized in Section V.

II. BACKGROUND AND RELATED WORK

The term *virtualization* is usually associated with such technologies, which allow the partition of hardware resources to conform isolated execution environments called *Virtual Machines*. But there is a technology which has the opposite goals: *Reverse Virtualization*. As suggested by its name, it integrates hardware resources from multiple computers (nodes of a cluster) to provide the image of a virtual Symmetric Multiprocessing System or vSMP. Works related to Distributed Virtualization present a cluster as a virtual shared memory multiprocessor. v-NUMA [4], and the University of Tokyo's Virtual Multiprocessor [5] allow multiple physical computers to host a single OS instance. ScaleMP [6] virtualizes an SMP and defines its virtualization paradigm as an *aggregation* of computational resources as opposed to *partitioning*. Somehow, Reverse Virtualization

and a DVS share the same goals, but the latter allows not only the aggregation of resources but also their partitioning.

Since there are plenty of articles which can be used as surveys of virtualization [7][8], this section will include details only about those technologies on which the DVS model is based and other related works.

A. Background

Classical definitions about Operating System mention that it is a layer of software located between applications and hardware. In OS-virtualization technology, the guest OS does not manage real hardware, but operates on virtual devices provided by a lower layer of software, such as a host-OS. Therefore, it seems appropriate to refer to the guest-OS as a VOS. There is a noticeable similarity with the paravirtualization [9] approach, the difference lying in the fact that there is a host-OS instead of a hypervisor in the lower layer. Instead of requesting services by means of hypervisor-calls, the guest-OS uses system calls. Hence, OS-virtualization and paravirtualization share the same benefits and drawbacks.

A well-known project that allows running multiple instances of Linux over other native Linux (as host) is User-Mode Linux (UML) [10]. CoLinux [11] is another project that allows running Linux as a guest-OS, but on a Windows host. Minix over Linux (MoL) [12] allows running multiple instances of a multi-server OS, such as Minix [13] over a Linux host.

MoL emulates Minix Interprocess Communications (IPC) mechanisms using TCP sockets, so that processes of the same instance of MoL can be executed in different hosts. This was the germinal version of the architecture model proposed in this article, but the use of Linux provided IPC and a pseudo-microkernel process running in user-mode turns performance its main weaknesses. To improve it, a microkernel with its own IPC mechanisms was developed to be embedded in the Linux kernel named M3-IPC (as a lightweight co-kernel) [14][15]. Later, it was extended to exchange messages and data among processes of the same MoL instance running on several nodes, allowing a multi-server VOS to be turned into a DVOS.

Other technologies that were sources of inspiration for the proposed model are those used by DOSs [16]. They fully developed and investigated in the 1990s as a consequence of the limited performance of a single host and the growing demand for computing power and scalability. Unlike Reverse Virtualization, which is a technology that virtualizes an SMP computer, a DOS performs distributed processing by expanding OS abstract resources to all its nodes. These resources are analogous to those provided by a centralized OS, such as users, processes, files, pipes, sockets, message queues, shared memory, semaphores, and mutexes.

Software factoring is a well-known approach in the field of OSs used by microkernel technologies. Servers and processes communicate with one another by passing messages through an IPC facility furnished by the OS kernel [17]. Unlike monolithic OS, a microkernel-based OS factors the kernel functions and services into multiple layers. Each layer is made up of several isolated processes running in user-mode, and the lowest layer runs the microkernel in supervisor mode [18]. Since upper layer servers and tasks do not have the right privileges to handle the hardware by their own, the microkernel provides services which allow them to operate on the hardware indirectly. A similarity between a microkernel OS and a paravirtualization system becomes evident [19]. In some ways, the microkernel acts as a para-virtualization hypervisor for a single VM, consisting of a set of user-space processes that constitute the guest-OS. Factoring an OS into multiple user-space tasks and servers provides the isolation required by a virtualization system and allows the distribution of processes in multiple nodes of a cluster.

The proposed model takes advantage of another technology: OS-based virtualization. It is a system call level virtualization, partitioning OS resources into isolated instances of execution environments. The host-OS isolates sets of user-space applications in Containers, Jails or Zones. Linux implements Containers [20] with two main kernel features; 1) *cgroups* [21]: It allows to isolate, prioritize, limit, and account for resource usage of a set of processes named *Control Groups*; 2) *namespaces* [22]: Usually, an OS provides a global namespace for OS abstract resources like UIDs, PIDs, file names, sockets, etc. All Containers provide applications with their own execution environment, but they all share the same OS. Therefore, the isolation property seems to be weaker against hardware virtualization and paravirtualization, but the performance gain is significant [23].

Any virtualization system whose aim is to provide IaaS with provider-class quality must consider high-availability as a requirement in its design. As a distributed system, a DVS must support the dynamic behavior of clusters where nodes are permanently added and removed. In a data center, several kinds of failures occur: in computers, in processes, in the network, and in operations; hence, they should be all considered in system design. Generally, component replication is the mechanism adopted to tolerate faults. Although there is extensive research about fault handling in distributed systems and because it is a complex issue [24], it is better to use tested tools, such as Distributed Consensus [25] and Group Communications Systems (GCS) [26] for achieving fault-tolerance through replication. Birman [26] states that: "*The use of a GCS should be considered for standardization, complexity, and performance reasons*". As Birman suggests, the prototype built as a proof of concept of a DVS is based on the use of an underlying GCS, which helped in the development of fault-tolerant components. Moreover, the use of a GCS allows the decoupling of a distributed application from the group communication mechanisms and from its failure detectors.

If a critical application runs on a DVS and its distributed components are strongly coupled, a fault on one member could result in a complete application failure. A GCS could be used by critical services, such as file servers, storage servers, web servers, etc. to solve the replication issue, providing more reliable services.

B. Related Works

Clustered Virtual Machines [27] is a technology used to run applications in a distributed way across a group of containers spread on several nodes of a cluster. On *Clustered Virtualization*, each application component runs within a container using the services of the host-OS in which each container is located.

Mesosphere's Data Center Operating System (DC/OS) [28] allows developers and administrators to consider a data center as a single computer that runs applications in software containers, but it is not really an OS; it is rather a container cluster manager with frameworks that provides PaaS. *JESSICA2* [29] is a distributed Java Virtual Machine implemented as a middleware, which supports parallel execution in a networked cluster environment, but it is limited to Java applications. Another software architecture model used for application development proposes to partitioning the application in autonomous components named Microservices [3]. With a set of microservices running on a cluster of servers the application's computing and resource needs are distributed, thus increasing application performance and scalability.

Unlike *Clustered Virtualization*, running a distributed application on a DVS can share the same instance of a DVOS allowing references to the same resource namespaces and system objects (such as users, pipes, queues, files, PIDs, sockets, etc.) as if they were running on the same host. This key feature can be sometimes used by developers when they need legacy applications to migrate to the Cloud, as well as for applications specifically developed to run in the Cloud.

III. Design and Implementation

Thinking of a distributed virtualization technology seems to make sense to achieve higher performance and increase service availability. OS-based virtualization and DOS technologies lead the authors to think about their convergence to achieve these goals, extending the boundaries of the virtual execution environment to multiple hosts and thereby multiplexing a cluster among multiple isolated instances of a DVOS.

An OS-based distributed virtualization approach will explore aggregation with partitioning. In such systems, a set of server processes constitutes a DVOS running within an execution environment made up of Distributed Containers (DC). Processes belonging to a DC may be spread on several nodes of a cluster (aggregation); and processes of different DCs could share the same host (partitioning).

Several hardware virtualization products offer high-availability and fault-tolerance by replicating a VM with its inner OS and all its processes. In such systems, load distribution is made using a VM migration facility that moves a complete VM from one server to another as a whole. A DVS could allow replication and migration of either all processes of DVOS or only some of them, such as the critical ones.

DOSs implement their policies and mechanisms, such as load balancing, process migration, leader election, consensus, fault detection, etc., within the system itself. As a result of this monolithic design, software modules are strongly coupled to one another and to the kernel, so that they cannot be reused by other applications. It is also difficult to change one of these components without changing the whole system. The DVS model relaxes the coupling among components, breaking them up as independent services with specific liabilities.

A. DVS Architecture

The main components of the DVS architecture are (see Fig. 2):

1) Distributed Virtualization Kernel (DVK): It is the core software layer that integrates the resources of the cluster, manages and limits the resources assigned to each DC. It provides interfaces for low-level protocols and services, which can be used to build a VOS, such as IPC, GCS, synchronization, replication, locking, leader election, fault detection, mutual exclusion, performance parameter sensing, processes migration mechanism, and key-value services. The DVK provides interfaces to manage all DVS resources, such as nodes, DCs and processes. Process management allows the DVS administrator to assign processes to a DC and to allocate nodes for it. The node in which the process runs can be changed, as in case of a migration, or when the process was replaced by another one, such as a backup process. For communication purposes, location changes made by the replacement or migration of a process are hidden from the other processes within the DC.

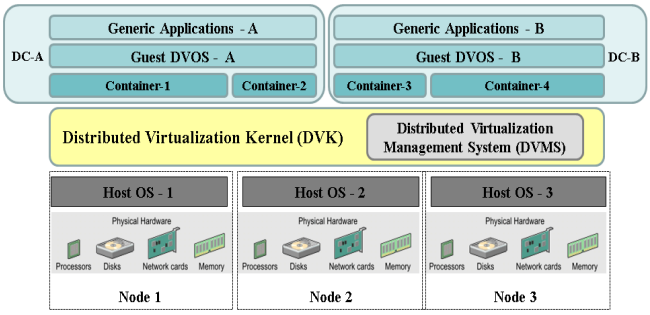


Figure 2. Distributed Virtualization System architecture model

2) Distributed Virtualization Management System (DVMS): It is the software layer that allows the DVS administrator to both manage the resources of the cluster, providing a DC for each VOS, and perform DVS monitoring.

3) Container: It is a host-OS abstraction which provides an isolated environment to run the components of a VOS. A set of Containers which belongs to the same VOS makes up a Distributed Container.

4) Distributed Container (DC): It is a set of single Containers, each one being set up by the DVMS in the host-OS of each node. There is one DC per VOS, and a DC can span from one to all nodes.

5) Virtual Operating System (VOS): Although any kind of VOS can be developed or modified to meet DVS architecture requirements, a DVOS can obtain greater benefits because it is able to distribute its processes in several nodes. Each VOS (single or distributed) runs within a DC. The task of modifying an existing OS to turn it into a VOS is simplified because it does not need to deal with real hardware resources but with virtual ones. Moreover, a VOS needs to manage neither virtual memory nor CPU scheduling because it is done by the host-OS.

6) VOS applications: They are applications (single or distributed) running within the same DC, using VOS-provided services.

The resource allocation unit for a DOS is the node as a whole; but for a DVOS (running within a DC) it is each single virtual resource provided by the host-OS on each node. This higher degree of granularity of the infrastructure unit allows a better use of resources and provides greater elasticity and efficiency.

B. DVS Prototype

Since the project startup (2013), a DVS prototype was implemented which runs on a cluster of x86 computers and Linux as OS-host. The DVS prototype considers the following abstractions, and the relations between them are presented in Fig. 3.

1) DVS: It is the top level layer that assembles all cluster nodes and it embraces all DCs.

2) Node: It is a computer that belongs to the DVS where processes of several DCs are able to be run. All nodes are connected by a network infrastructure.

3) DC: It is the group or set of related processes that might be scattered on several nodes. M3-IPC only allows communications among processes that belong to the same DC. The boundary of each DC can be based on administrative boundaries. A DC hides its internals from the outside world and hides network communication issues from its processes.

4) Proxies: They are special processes used to transfer messages and data blocks between nodes. M3-IPC does not impose a network/transport protocol to be used for inter-node communications. This feature allows programmers to choose the protocol that best fit their needs. Nodes communicate among them through proxies.

5) Process: Every process registered in a DC has an *endpoint* which identifies it. Process endpoints are unique and global within a DC, but could be repeated within other DCs.

With the exception of the process endpoints, the other DVS abstractions are hidden from the VOS and its processes. They are managed by the DVS administrator, such as adding or removing hosts as nodes of the DVS, allocating nodes to DCs, or setting proxies to communicate nodes.

Two simple VOS were developed to be executed as guests on the prototype as proof of concept. One of them is a multiserver VOS named MoL, and the other is a unikernel [30] VOS named ukVOS; both are able to provide Internet services (web server). MoL is made up of loosely coupled servers and tasks integrated as VOS components. Alternatively, they can be run alone serving Linux ordinary client processes by using some kind of kernel-user interface as FUSE or BUSE.

C. Distributed Virtualization Kernel

A DVK was implemented in the DVS prototype as a Linux kernel module and a patch, complemented by a set of libraries, commands and tools. The DVK module of each node (which includes M3-IPC) is implemented as a Linux co-kernel.

DVK APIs allow configuring and managing all DVS abstractions (DVS, DCs, nodes, and proxies), and mapping processes to DCs, DCs to nodes, and proxies to nodes.

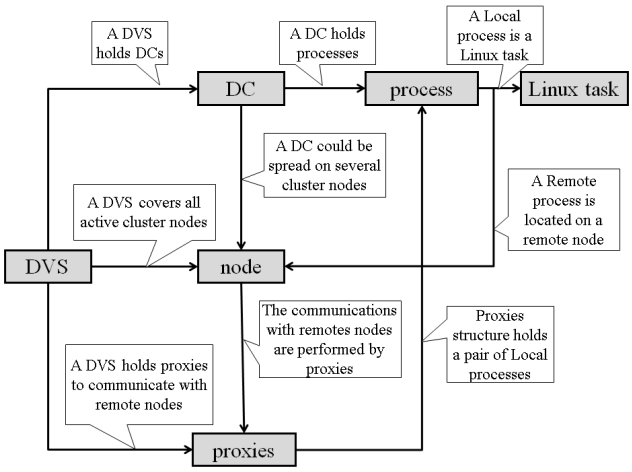


Figure 3. DVS abstractions and their relationships.

Through DVK APIs, a program can set the new node where an endpoint is located as a result of a process migration. DVK APIs also allow changing the endpoint type from Backup to Primary after the primary process has finished by a fault in a replicated service. These APIs were tested in a Virtual Disk Driver, which was developed for the prototype.

D. M3-IPC

A critical component of every distributed system is the software communication infrastructure. To simplify the development of VOS for the DVS prototype, an IPC infrastructure was developed and is named M3-IPC [14][15]. It allows building VOS components, such as clients, servers and tasks with a uniform semantics without considering process location. Provider-class features were considered in the design stage, such as process replication, process migration, communications confinement, and performance for both intra-node and inter-node communications. M3-IPC is a pluggable module embedded

in the Linux kernel, supplying the IPC primitives of a microkernel OS.

Messages and data exchange among nodes is carried out by proxies. A proxy transports messages and data between two nodes without considering source/destination processes or the DC they belong to. Proxies can run either in user-mode to provide versatility or in kernel-mode for efficiency reasons, according to the DVS administrator's choice. Running proxies in user-mode may result in an efficiency loss, but it has the benefit of granting flexibility to freely choose protocols, and to easily add facilities, such as compression and encryption. At present, several kinds of proxies were developed for the DVS prototype using different protocols (TCP, TIPC [31], UDP, UDT [32], custom Raw Ethernet), some of them in user-mode and others in kernel-mode.

E. MoL-FS

One of the main processes of MoL is Filesystem Server (FS). It handles requests from user-level applications using POSIX system calls related to filesystems, files, directories, etc.

MoL-FS [33] is a modified version of Minix FS which uses M3-IPC as a message transfer mechanism. On Minix, clients, FS server and Disk task are independent user-space processes which reside on the same host. Since MoL-FS uses M3-IPC, which it does not limit communications within the same host, clients, MoL-FS, and a storage server named MoL-VDD [34] could be on different nodes of a cluster like a distributed OS.

As MoL-FS was designed to be used as a component of a VOS, only those applications developed using M3-IPC and MoL-FS protocol could use its services. A FUSE gateway was developed to extend its use to ordinary Linux applications, taking advantage of the ability to adapt granted by FUSE. Another advantage of having the FUSE gateway is that it allows performance evaluation by using standard Linux tools.

Currently, a replicated MoL-FS server is at development stage using Spread Toolkit [35] as Group Communications System (GCS) for multicast message services, failure detection, and group membership management.

F. MoL-VDD

MoL-FS supports several storage devices: ram disks, image files, raw Linux devices, and a Virtual Disk Driver (MoL-VDD).

MoL-VDD runs as a server process within a DC, and it provides its clients with the same storage devices as MoL-FS. A fault-tolerance support through data and processing replication techniques was added to it to test the behavior of the DVS infrastructure in failure scenarios. Fault-tolerance is achieved transparently for the application through the use of the facilities offered by the DVS, M3-IPC and Spread Toolkit.

MoL-VDD supports this kind of distributed environment in a dynamic and transparent way in which user processes, servers, and drivers can migrate due to availability or performance issues. These characteristics are highly appreciated by IaaS providers because they increase the elasticity, high availability, and robustness of their offered services, and because they optimize the use of their computational and storage resources.

A BUSE [36] driver was developed so as to allow Linux to mount a MoL-VDD device. Currently, an NBD [37] gateway is at development stage, which allows MoL-VDD to mount an NBD volume.

G. Other MoL Servers and Drivers

MoL is made up of several servers and tasks, which are communicated using M3-IPC. In addition to MoL-FS and MoL-VDD, other servers and tasks were developed and implemented:

- *System Task (Systask):* It handles low level requests from other servers and tasks, and it makes its own requests to its host-OS. It is also a replicated process which must run in every node of the DC.
- *Process Manager (PM):* It handles process and memory related system calls.
- *Information Server (IS):* It allows gathering information about the state of every server and task in the DC. A Web Information Server is also available to present VOS status information to a web browser.
- *Reincarnation Server (RS):* It allows for starting processes in any node of the DC and handles process migration.
- *Data Server (DS):* It is a key-value server.
- *Ethernet Task (ETH):* It is the interface to virtual (TUN/TAP) or real host's Ethernet devices.
- *Internet Server (INET):* It is a user-space implementation of the TCP/IP protocol.

As aforementioned, all MoL components must run within a DC, and all of them could run spread on several nodes of the DVS.

H. Unikernel-like VOS

Unikernel is a recent technology, which takes up library OS concepts, but instead of working on the bare-metal it runs over some hypervisor. It has many benefits that make it attractive to provide Cloud services.

A unikernel-like VOS was developed to test the DVS prototype named ukVOS. It is a single executable image, which uses low-level services provided by the host-OS and by the DVS infrastructure instead of being provided by a hypervisor.

ukVOS is based on *LwIP* [38] code because it includes a user-space implementation of the TCP/IP protocol and a simple web server with a fixed in-memory web page. A FAT Filesystem code and M3-IPC support were added to allow several unikernel VOS images and the web server was modified to support files.

Several images of ukVOS with different configurations were developed to test the DVS. One of them is a ukVOS image with FAT filesystem support, which uses the disk image file from a Linux regular file. Another image with M3-IPC allows using an external MoL-VDD. In the other

ukVOS image, the web server gets the files from an external MoL-FS.

I. *DVS Management*

Management is a fundamental requirement of a virtualization system designed to offer Cloud services. At present, the DVS prototype is managed by a Command Line Interface (CLI). A *Webmin* [39] module is at development stage, which will allow for a web-based management interface (Fig. 4).

IV. EVALUATION

A DVS is a complex system in which several metrics may be considered as CPU, network and memory usage of cluster nodes, the time to restore a replicated service after a failure, IPC recovery time after the destination process of a message has been migrated to another node, to name a few.

The following metrics were established to be presented in this section:

1) IPC performance (Fig. 5): It is a critical issue because communications among different components of a VOS or DVOS are performed using RPC based on IPC. User-space applications use IPC to (transparently) make local or remote system calls.

2) Virtual Disk Driver performance: The throughput of storage services is critical, with or without replication.

3) Filesystem Server performance: Other components of an infrastructure are filesystem services. They provide high-level applications with files and directory services. Their throughput to store and retrieve data directly impact on applications.

4) Web server performance: File transfer throughput was considered an important metric (response time is another one) to provide web services.

For performance evaluation in distributed systems, such as a DVS, benchmarks and micro-benchmarks should be made between co-located processes and processes running on different nodes.

Benchmarks and micro-benchmarks were performed on the prototype deployed in a cluster made up of 20 PCs (Intel(R) Core(TM) i7-4790 3.60GHz), a 1-Gbps network, and Linux as the host-OS.

A. *IPC performance evaluation*

I. *Tests between Co-located Processes*

Tests between co-located processes allow the comparison of M3-IPC performance versus other IPC mechanisms available on Linux.

One of the design goals states that the expected performance should be as good as the fastest IPC mechanisms available on Linux. The following IPC mechanisms were tested using custom and [40, 41] provided micro-benchmarks: Message Queues, RPC, TIPC, FIFOs, pipes, Unix Sockets, TCP Sockets, SRR [42], SIMPL [43].

The presented results (Fig. 5-A) summarize message transfer throughput achieved by the IPC mechanisms running a single pair of client/server processes. Linux IPC mechanisms with the highest performance were pipes and named pipes (or FIFOs) followed by M3-IPC (925,314 [msg/s]).

Another micro-benchmark of message transfers between multiple pairs of client/server processes was run to evaluate performance in concurrency. The highest average throughput was 1,753,206 [msg/s], which was reached with 4 pairs of client/server processes (4 cores).

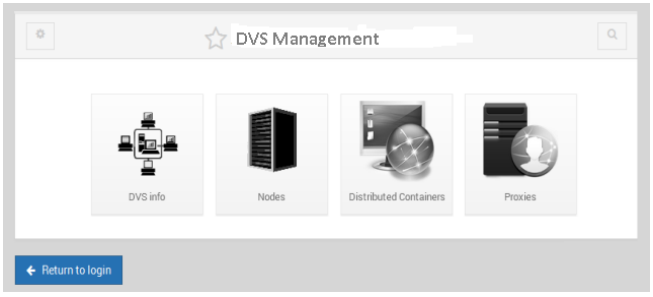


Figure 4. Webmin module menu for DVS management.

Fig. 5-B presents data transfer throughput, M3-IPC performance surpasses other IPC mechanisms on Linux. The reasons for this behavior are: 1) M3-IPC performs a single copy of data between address spaces while the others perform at least two copies (Source to Kernel, Kernel to Destination); 2) it requires a lower number of context switches; 3) it uses the Linux kernel provided *page_copy()* function, which uses MMX instructions.

II. *Tests between Processes Located on Different Nodes*

This section presents performance results of M3-IPC against RPC and TIPC.

M3-IPC does not consider flow control, error control, or congestion control. Those issues are delegated to proxies and the protocol they use. Reference implementations of M3-IPC proxies use TCP and TIPC as transport protocols.

As it can be seen in Fig. 5-C, a proxy using RAW Ethernet sockets has the highest message transfer throughput followed by TIPC. M3-IPC, using TCP on proxies, has a throughput similar to that of RPC. A custom RAW Ethernet protocol was designed to be used in M3-IPC proxies. In this protocol, not all frames are acknowledged because the upper layer protocol between proxies acknowledges messages and data transfers.

The remarkable performance of TIPC suggested that it could be well used by M3-IPC proxies as transport protocol. M3-IPC versatility and flexibility in proxy programming allowed authors to modify the source code of proxies in a few hours so as to use TIPC instead of TCP. These changes result in an improvement of performance, emphasizing the impact of the transport protocol on its throughput.

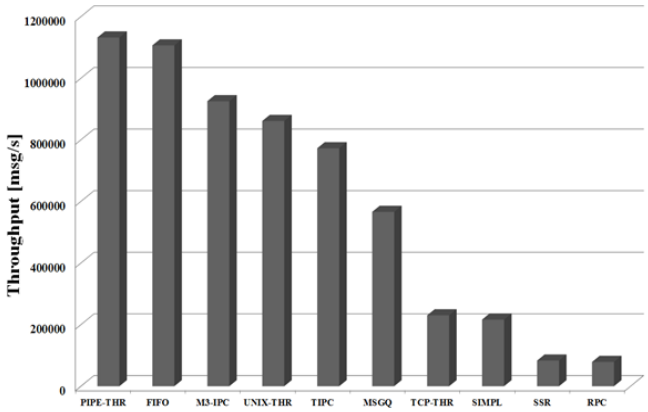
As shown in Fig.5-D, TIPC presents the highest throughput for transferring blocks of data lower than 16 [Kbytes]. The proxy using the custom RAW ethernet protocol has the highest throughput for transferring blocks of data greater than 16 [Kbytes].

Fig.5-D also shows that there is no noticeable difference in performance when using TIPC instead of TCP as transport protocol on M3-IPC proxies to copy data blocks.

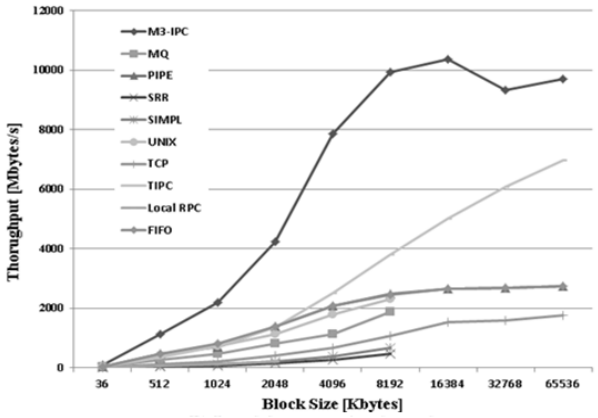
B. Mol-VDD Performance Evaluation

Two types of micro-benchmarks were developed to assess the performance of MoL-VDD and its BUSE driver:

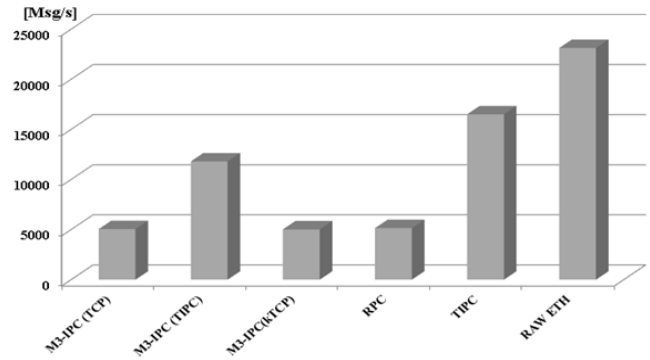
1) Local Tests: Client process and *MoL-VDD* run on the same node (Fig. 6).



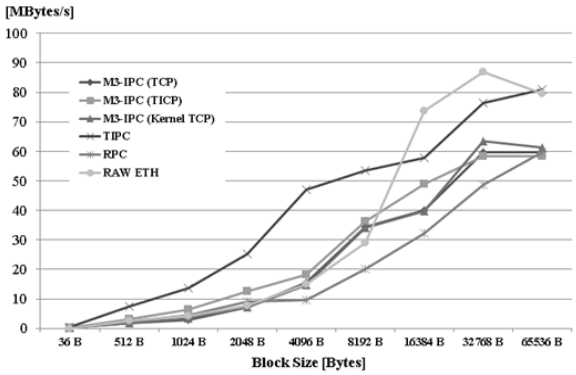
(A) Local message transfer throughput



(B) Local data transfer throughput



(C) Remote message transfer throughput



(D) Remote data transfer throughput

Figure 5. Results of IPC performance tests.

When MoL-VDD is mentioned in Fig. 6 and Fig. 7, it means that the transfer was performed between a client process and the disk driver by using M3-IPC.

When BUSE is mentioned, it means that the transfer was performed by the client process through the BUSE driver.

Since the BUSE driver was built using Linux threads and user-space mutexes, they negatively impact on its performance.

If the word *single* is mentioned, it means that the server is not replicated. If the word *replicated* is mentioned, it means that there is one backup MoL-VDD driver running on another node.

A TCP user-space proxy was used to exchange data and messages between nodes, while Spread Toolkit was used as GCS between replicas.

2) Remote or Cluster Tests: Client process and *MoL-VDD* run on different nodes (Fig. 7).

The performance evaluation was carried out in conjunction with NBD [37] for comparison purposes. Micro-benchmarks used ***time*** and ***dd*** commands to perform data transfers and take measurements.

We used image files located on a Linux RAM disk to avoid the latency of hard disks.

C. Mol-FS Performance Evaluation

As MoL-FS is implemented in user-space, its performance was compared against other user-space filesystems. An NFS Server (UNFS [44]) and an NFS Client (HSFS [45]), both implemented in user-space, were chosen for that purpose.

A FUSE driver was developed for MoL-FS, which allows mounting it and using every Linux command on its files and directories. For micro-benchmarks, a simple ***cp*** Linux command was used to evaluate the performance. The source and destination files were on a RAM disk image to avoid the delay of a hard disk. By space limitations, only tests where processes run on different nodes are presented in Table I.

Several operational scenarios were tested, but before starting micro-benchmarks, network performance was measured with other tools:

- SSH ***scp*** reports 43 Mbytes/s.
- TIPC custom application reports 81 Mbytes/s.
- M3-IPC custom application reports 58 Mbytes/s.

In *Config-A,* client process (Node0) gets/puts files from/to MoL-FS (Node1) which gets/puts raw data from an image file.

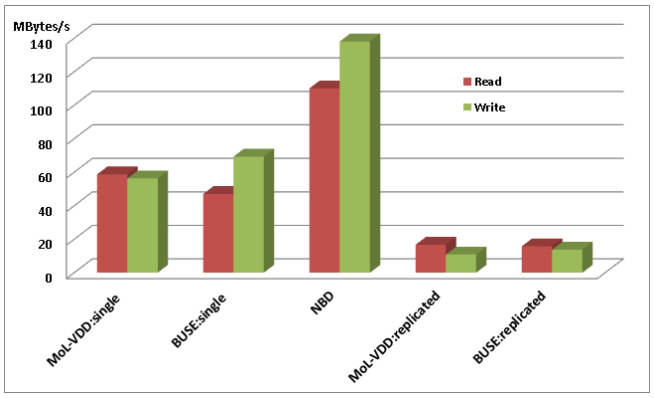


Figure 6. Mole-VDD throughput (client process in another node).

In *Config-B,* client process (Node0) gets/puts files from/to MoL-FS (Node1) which gets/puts raw data from MoL-VDD (Node1).

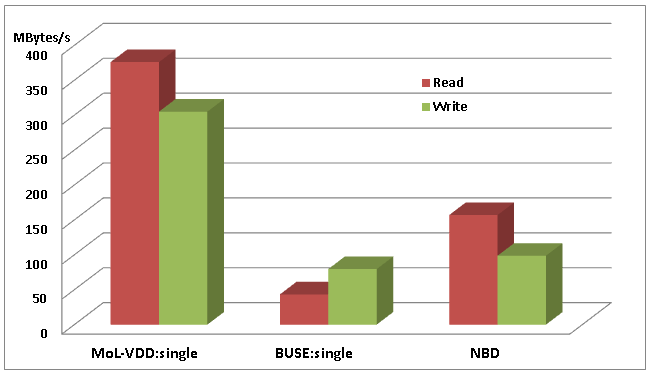


Figure 7. Mol-VDD throughput (client process in the same node).

Although the results are not attractive to adopt Mol-FS as a remote filesystem, it must be considered that they were obtained using the FUSE Gateway.

TABLE I. MOL-FS FILE TRANSFER THROUGHPUT (TWO NODES)

	Read	Mol-FS Read		Mol-FS Write	
File size [Mbytes]	UNFS/HSFS [Mbytes/s]	Config. A [Mbytes/s]	Config. B [Mbytes/s]	Config. A [Mbytes/s]	Config. B [Mbytes/s]
1	4.17	8.00	7.04	1.98	2.34
10	56.82	9.52	10.54	2.30	2.40
100	59.84	10.15	11.83	2.04	2.13

Other raw tests performed on MoL-FS without using the FUSE gateway showed a single transfer throughput of 47 Mbytes/s, but further performance optimizations must be made.

D. Web Server Performance Evaluation

Table II shows the performance results of benchmarks performed on a web server (*nweb*) running on ukVOS, and in MoL considering different file sizes and configurations. The web client program (*wget*) was located in the same DC and in the same node. The disk image files used in benchmarks were located on RAM disks to avoid hard disk latencies.

In *Config-A,* the web server gets its files from the host-OS filesystem on a RAM disk, and it represents a baseline measurement. In *Config-B,* the web server gets the files from MoL-FS which uses a disk image file. In *Config-C,* the web server gets the files from MoL-FS, which uses a MoL-VDD as a storage server to get raw data from an image file. The three processes run in the same DC and in the same node, using M3-IPC between them.

TABLE II. WEB SERVER FILE TRANSFER THROUGHPUT (SAME NODE)

	Unikernel (ukVOS)			*Multi-server (MoL)*		
File size [Mbytes]	Config-A [Mbytes/s]	Config-B [Mbytes/s]	Config-C [Mbytes/s]	Config-A [Mbytes/s]	Config-B [Mbytes/s]	Config-C [Mbytes/s]
10	7.44	7.03	7.04	74.62	70.00	67.54
50	6.86	6.76	6.85	81.43	79.10	79.52
100	7.96	6.82	6.72	97.11	92.13	92.31

There is an evident performance difference between running the web server in *ukVOS* versus running it in *MoL*. This suggests that this user-mode TCP/IP protocol stack implementation with Linux threads is not efficient.

MoL performance for those configurations in which process components are located on the same node (Config-B and Config-C) is somewhat lower than if running the web server directly in Linux (Config-A). This confirms the results presented in [23].

Table III shows the performance results of benchmarks performed on a web server and other related processes on the same DC but on different nodes.

In *Config-D,* the web server (Node0) gets the files from MoL-FS (Node0), which gets raw data from MoL-VDD (Node1), which uses a disk image file.

In *Config-E,* the web server (Node0) gets the files from MoL-FS (Node1) which gets raw data from a disk image file.

In *Config-F,* the web server (Node0) gets the files from MoL-FS (Node1) which gets raw data from MoL-VDD (Node1).

TABLE III. WEB SERVER FILE TRANSFER THROUGHPUT (TWO NODES)

	Unikernel (ukVOS)			*Multi-server (MoL)*		
File size [Mbytes]	Config-D [Mbytes/s]	Config-E [Mbytes/s]	Config-F [Mbytes/s]	Config-D [Mbytes/s]	Config-E [Mbytes/s]	Config-F [Mbytes/s]
10	2.62	2,90	2.75	4.08	3.,81	3.51
50	2.72	2.89	2.88	4.32	3.75	3.57
100	2.70	2.94	2.74	4.32	3.79	3.55

Since TCP user-space proxies were used in Node0 and Node1 for the benchmarks, better results are expected by using TIPC or RAW ethernet proxies according to the M3-IPC performance evaluation.

V. Conclusions and Future Works

The proposed DVS model combines and integrates Virtualization and DOS technologies to provide the benefits of both worlds, making it suitable to deliver provider-class Cloud services. With a DVS, the limits for an isolated execution environment for running applications are expanded to all cluster nodes (aggregation). Moreover, its utilization is improved by enabling the same cluster to be shared (partitioning) among several DCs. A DVS prototype was developed to check the design and implementation correctness; and after several testing environments, the feasibility of the proposed model was proved.

Migrating legacy applications from on-premises servers to a DVS is facilitated since POSIX APIs and RPC (supplied by a VOS) could be used, allowing code reuse. In this way, implementation costs and time are improved by reducing the encoding effort. On the other hand, new applications based on the MSA can execute their set of microservices in several nodes of the cluster by using RPC for communications.

Future research and development stages will focus on making improvements to provide scalable, elastic, high-performance and high-availability virtualization services; and integrating DVS management to Openstack [46].

Acknowledgments

Toni Cortes' involvement in this work was financially supported by the Spanish National Government (financial grant SEV2015-0493 of Severo Ochoa program), the Spanish Ministry of Science and Innovation (contract TIN2015-65316), and the Government of Catalonia (contract 2014-SGR-1051).

Fernando G. Tinetti's involvement in this work was financially supported by Universidad de La Plata (Faculty of Informatics) and the Scientific Research Board (CIC, for its Spanish initials) of Buenos Aires, Argentina.

References

[1] D. Hall, D. Scherrer, and J. Sventek, "A Virtual Operating System", Journal Communication of the ACM, 1980.

[2] J. Turnbull, "The Docker Book", 2014, Available online at: https://www.dockerbook.com/, accessed on 30 October 2017.

[3] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. "Microservices in Practice, Part 1: Reality Check and Service Design", IEEE Softw. 34, pp 91-98, Jan. 2017, 2017.

[4] M. Chapman and G.t Heiser, "vNUMA: a virtual shared-memory multiprocessor", Proc. of the 2009 conference on USENIX Annual technical conference (USENIX'09), USENIX Association, Berkeley, CA, USA, pp. 2-2.

[5] K. Kaneda, Y. Oyama, and A. Yonezawa, "A virtual machine monitor for utilizing non-dedicated clusters", Proc. of the twentieth ACM symposium on Operating systems principles (SOSP '05), ACM, New York, NY, USA, pp. 1-11, doi: https://doi.org/10.1145/1095810.1118618.

[6] ScaleSMP, "vSMP Foundation Architecture", WhitePaper, Available online at http://www.scalemp.com/media-hub/resources/white-papers, 2013, accessed on 30 October 2017.

[7] N. Goel, A. Gupta, and S. N. Singh, "A study report on virtualization technique," 2016 International Conference on Computing, Communication and Automation (ICCCA), Noida, pp. 1250-1255, doi: 10.1109/CCAA.2016.7813908.

[8] S. Liu and W. Jia, "A Survey: Main Virtualization Methods and Key Virtualization Technologies of CPU and Memory", The Open Cybernetics & Systemics Journal, vol. 9, 2015, pp. 350-358, doi: 10.2174/1874110X01509010350.

[9] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications", Proc. of the USENIX Annual Technical Conference, 2002.

[10] J. Dike, "A user-mode port of the linux kernel", Proc. of the 4th annual Linux Showcase & Conference, vol. 4, pp. 7-7, USENIX Association, Berkeley, CA, USA , 2000.

[11] D. Aloni, "Cooperative Linux", Proc. of the Linux Symposium, 2004.

[12] P. Pessolani and O. Jara, "Minix over Linux: A User-Space Multiserver Operating System," Proc. Brazilian Symposium on Computing System Engineering, Florianopolis, 2011, pp. 158-163, doi: 10.1109/SBESC.2011.17.

[13] A. S. Tanenbaum, R. Appuswamy, H. Bos, L. Cavallaro, C. Giuffrida, T. Hrubỳ, J. Herder, and E. van der Kouwe, "Minix 3: Status Report and Current Research", login: The USENIX Magazine, 2010.

[14] P. Pessolani, T. Cortes, F. G. Tinetti, and S. Gonnet, "An IPC microkernel embedded in the Linux kernel" (in Spanish), XV Workshop on Computer Science Researchers, Argentina, 2012.

[15] P. Pessolani, T. Cortes, F. G. Tinetti, and S. Gonnet, "An IPC Software Layer for Building a Distributed Virtualization System", Congreso Argentino de Ciencias de la Computación (CACIC 2017) La Plata, Argentina, October 9-13, 2017.

[16] R. Buyya, T. Cortes, and H. Jin, "Single System Image", Int. J. High Perform. Comput. Appl. Vol. 15, May 2001, pp 124-135, doi: http://dx.doi.org/10.1177/109434200101500205.

[17] G. Heiser and K. Elphinstone, "L4 Microkernels: The Lessons from 20 Years of Research and Deployment", ACM Trans. Comput. Syst., vol. 34, Article 1, April 2016, doi: http://dx.doi.org/10.1145/2893177.

[18] M.Gien and L. Grob, "Micro-kernel Architecture Key to Modern Operating Systems Design", 1990.

[19] F. Armand and M. Gien, "A Practical Look at Micro-Kernels and Virtual Machine Monitors", 6th IEEE Consumer Communications and Networking Conference, Las Vegas, NV, 2009, pp. 1-7, doi: 10.1109/CCNC.2009.4784874.

[20] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors", Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07), ACM, New York, NY, USA, pp. 275-287, doi: http://dx.doi.org/10.1145/1272996.1273025.

[21] P. B. Menage, "Adding Generic Process Containers to the Linux Kernel", Proc. of the Ottawa Linux Symposium, 2007.

[22] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbotto, "The use of name spaces in plan 9", Proc. of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring (EW 5). ACM, New York, NY, USA, pp. 1-5, doi: http://dx.doi.org/10.1145/506378.506413.

[23] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 2015, pp. 171-172, doi: 10.1109/ISPASS.2015.7095802.

[24] C. Cachin, R. Guerraoui, and L. Rodrigues, “Introduction to Reliable and Secure Distributed Programming (2nd ed.)”, Springer Publishing Company Incorporated, 2011.

[25] T. D. Chandra, R. Griesemer, and J. Redstone ,”Paxos made live: an engineering perspective”, Proc. of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC '07), ACM, New York, NY, USA, pp. 398-407, doi: http://dx.doi.org/10.1145/1281100.1281103.

[26] K. P. Birman, ”The process group approach to reliable distributed computing”, Commun. ACM 36, Dec. 1993, pp. 37-53, doi:http://dx.doi.org/10.1145/163298.163303.

[27] V. Chavan and P. R. Kaveri, “Clustered virtual machines for higher availability of resources with improved scalability in cloud computing”, First International Conference on Networks & Soft Computing (ICNSC2014), pp. 221--225, Guntur, 2014.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and Ion Stoica, “Mesos: a platform for fine-grained resource sharing in the data center”, Proc. of the 8th USENIX conference on Networked systems design and implementation (NSDI'11), USENIX Association, Berkeley, CA, USA, pp. 295-308.

[29] C. Wang, F. C. Lau, and W. Zhu, "JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support," 2013 IEEE International Conference on Cluster Computing (CLUSTER), Chicago, Illinois, 2002, pp. 381, doi: 10.1109/CLUSTR.2002.1137770.

[30] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and Jon Crowcroft, “ Unikernels: library operating systems for the cloud”, Proc. of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13), ACM, New York, NY, USA, pp. 461-472, doi: http://dx.doi.org/10.1145/2451116.2451167.

[31] J. P. Maloy, “TIPC: Providing Communication for Linux Clusters”, Proc. of the Linux Symposium, vol. 2, pp. 347-356 2004.

[32] Y. Gu and R. L. Grossman, ”UDT: UDP-based data transfer for high-speed wide area networks”, Comput. Netw., vol. 51, May 2007, pp. 1777-1799, doi: http://dx.doi.org/10.1016/j.comnet.2006.11.009.

[33] D. Padula, M. Alemandi, P. Pessolani, T. Cortes, S. Gonnet, and F. Tinetti, “A User-space Virtualization-aware Filesystem”, CONAIISI 2015, Buenos Aires, Argentina, 2015.

[34] M. Alemandi and O. Jara, “A fault-tolerant virtual disk driver”, (in spanish) JIT 2015, Venado Tuerto, Argentina, 2015.

[35] The Spread Toolkit: http://www.spread.org, accessed on 30 October 2017.

[36] BUSE: https://github.com/acozzette/BUSE, accessed on 30 October 2017.

[37] NBD: http://nbd.sourceforge.net/, accessed on 30 October 2017.

[38] LwIP: http://savannah.nongnu.org/projects/lwip/, accessed on 30 October 2017.

[39] Webmin: http://www.webmin.com/, accessed on 30 October 2017.

[40] M. Kerrisk, “The Linux Programming Interface”, No Starch Press, ISBN 978-1-59327-220-3, 2010.

[41] ipc-bench: http://www.cl.cam.ac.uk/research/srg/netos/ipc-bench/ , accessed on 30 October 2017.

[42] SRR, “QNX API compatible message passing for Linux”, http://www.opcdatahub.com/Docs/booksr.html, accessed on 30 October 2017.

[43] J. Collins and R. Findlay, “Programming the SIMPL Way”, ISBN 0557012708, 2008.

[44] “A User-space NFS Server”, http://unfs3.sourceforge.net/, accessed on 30 October 2017.

[45] HSFS: https://github.com/openunix/hsfs, accessed on 30 October 2017.

[46] Openstack: https://www.openstack.org/, accessed on 30 October 2017.