A Multiserver User-space Unikernel for a Distributed Virtualization System

Pablo Pessolani
Departamento de Ingeniería en Sistemas de Información
Facultad Regional Santa Fe, UTN
Santa Fe, Argentina
ppessolani@frsf.utn.edu.ar

Abstract— **Nowadays, most Cloud applications are developed using Service Oriented Architecture (SOA) or MicroService Architecture (MSA). The scalability and performance of them is achieved by executing multiple instances of its components in different nodes of a virtualization cluster. Initially, they were deployed in Virtual Machines (VMs) but, they required enough computational, memory, network and storage resources to hold an Operating System (OS), a set of utilities, libraries, and the application component. By deploying hundreds of these application components, the resource requirements increase a lot. To minimize them, usually small OSs with small memory footprint are used. Another way to reduce the resource requirements is integrating the application components in a Unikernel. This article proposes a Unikernel called MUK, based on a multiserver OS, to be used as a tool to integrate Cloud application components. MUK was built to run in user-space of a Distributed Virtualization System. Both technologies facilitate the scattering of application components in a virtualization cluster keeping the isolation properties and minimal attack surface of a Unikernel.**

Keywords: virtualization, multiserver, middleware

I. INTRODUCTION

Nowadays, there is a need to build and to deploy distributed applications to satisfy the performance needs of Cloud information systems. They must be able to manipulate big data volumes (sounds, videos, photographs, documents, etc.) in huge and complex databases. Most of these Cloud applications are developed using the SOA [1] or MSA [2] methodologies. Both software architectures, consist of factoring a monolithic application into multiple components that communicate with each other through some standardized protocol (such as SOAP [3]), with a weak coupling between them.

The application scalability and performance is achieved by deploying multiple instances of its components in different nodes of a virtualization cluster (usually called "*swarms*"). Initially, Virtual Machines (VMs) were used, but they required enough computational, memory, network and storage resources to hold a complete Operating System (OS), its utilities, libraries, and the application component. When deploying a swarm of VMs, the resource needs increase proportionally. To reduce that demand, OSs with small memory footprint are often used.

Later, as management tools such as Docker [4], Kubernetes [5], Mesos [6] (among others) were improved, application components began to be deployed in Containers. Containers (and similar OS abstractions such as Jails [7] and Zones [8]) are isolated execution environments or domains in user-space to execute groups of processes. Although Containers share the same OS, they provide enough security, performance and failure isolation. As Containers demand fewer resources than VMs [9], they are a good choice to deploy swarms.

Another option to reduce resource requirements is to use the application component embedded in a Unikernel [10, 11]. A Unikernel is defined as "*specialized, single-address-space machine image constructed by using library operating system*" [12]. A Unikernel is a technology which integrates monolithically network, storage, and file systems services with the application component resulting in a single executable code. This way of building an environment to run the application optimizes resource usage because only those services that the application requires are integrated. Therefore, the size of the resulting executable code, memory footprint and the application attack surface are reduced to a minimum. Those facts improve application security, and make application components easily distributable, replicable and deployable.

It is unlikely that the application component embedded into a Unikernel will be completely autonomous. Probably, it needs to communicate with other components running within other Unikernels, and with external servers such as storage servers, file servers, logging-servers, database servers, key-value servers, etc. A provider-class Cloud application must coordinate the actions among its components and with the external servers to provide fault tolerance and replicated services. Peer to peer and group communications, leader election mechanisms, fault detection, distributed locking, etc., are requirements to coordinate actions among distributed components. All of these are complex services which must be considered in the application design and development stages and in the application-Unikernel build.

This article presents a Unikernel called MUK, based on a multi-server user-space OS instead of a Library OS. As its components (servers) are autonomous by definition, the integration with the application is facilitated. The chosen OS used as a base to build MUK was *Minix Over Linux* (MoL) [13], a user-space Minix branch which run as a set of Linux processes. Minix is a POSIX-compliant OS, and because MoL is a port of it to user-space, it is compliant too. As MUK is the Unikernel version of MoL, it keeps the POSIX-

compliance which allows it to execute legacy applications under its environment. This feature represents an important advantage over some other Unikernels which provide their own non-compliance APIs forcing developers to port their legacy applications.

MUK runs inside a Distributed Container of a Distributed Virtualization System (DVS) [14]. A Distributed Container covers several related Linux Containers scattered on nodes of a virtualization cluster. The Unikernel and DVS technologies facilitate the distribution of application components as swarms across the nodes of a DVS cluster. A DVS provides MUK and its integrated application components, several services required by distributed applications such as location-transparent IPC, a Group Communication System (GCS), fault detection, replication, leader election mechanisms, easy deployment and management, etc. All these features are the platform requirements to develop and to deploy elastic, high availability, high performance, distributed Cloud application.

The rest of this article is organized as follows. Section II refers to related works and background technologies. Section III describes the architecture of MoL as a Unikernel (MUK) and the design and implementation of its prototype. Partial performance evaluation results of the prototype are reported in section IV and finally, Section V presents conclusions, in-progress works, and future research projects.

II. Related Works and Background Technologies

The most commonly adopted virtualization technologies are VMs and Containers as isolated execution environments. Both are partitions of a physical computer, so that their computing resources are limited by the latter.

The architecture model of a DVS [14] is based on several Virtualization and DOS [15, 16] technologies getting the benefits of both worlds. Due to its features, it is a good choice to offer high performance provider-class Cloud services. In this sense, it presents attractive features for Cloud service providers such as high availability, replication, elasticity, load balancing and process migration. In a DVS, the administrator can build execution domains, called Distributed Containers (DC), which can be extended beyond the limits of a physical machine or cluster node (Fig. 1).

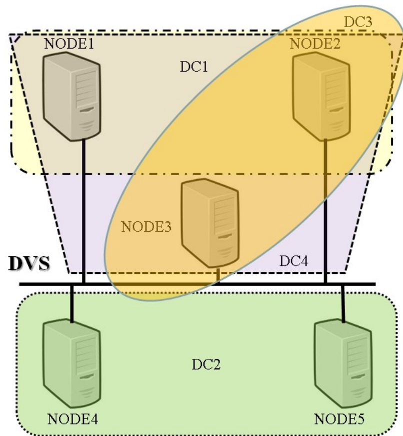

Figure 1. Distributed Virtualization System Topology.

Instead of running the application on the hosts-OS, a VOS (Virtual Operating Systems) [15] is needed to improve security and fault isolation and provide a greater variety of options for applications. The DVS enables the execution of multiple instances of different VOS each one executing in its own DC. A subset of nodes is allocated for each DC, but they can share the nodes between several DCs (Fig. 1). This higher degree of granularity of resource allocation allows a better use of the infrastructure and provides greater elasticity and efficiency.

A. DVS Architecture

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

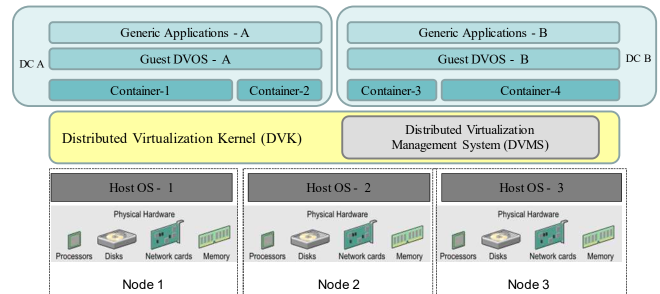

Figure 2. Distributed Virtualization System architecture model.

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, etc.

2) Distributed Virtualization Management System (DVMS): It is the software layer that allows the DVS administrator to both manage and monitor the resources of the cluster.

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

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. Only processes belonging to the same DC can communicate transparently between them using DVK provided mechanisms.

5) Virtual Operating System (VOS): Any kind of VOS can be developed or ported to meet DVS architecture requirements. 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. MUK is a proof of this assertion.

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

The DVS architecture opens several issues related to itself and to its components, such as the VOSs. This article presents the research and development works of a Unikernel VOS based on a multi-server OS.

B. *Minix over Linux (MoL)*

MoL [13] is a VOS formed by a set of Linux processes which exchange messages through the IPC mechanisms provided by a pseudo-kernel (***molkernel***). The Process Manager (PM) is emulated by a daemon named ***molpm***, the Filesystem Server (FS) is emulated by another daemon named ***molfs***, and some Minix tasks have their MoL counterpart daemons.

Several instances of MoL (different instances of ***molkernel***, ***molfs***, ***molpm***, and MoL tasks) could be running on the same Linux host, each of them as a VOS in user-space. On the other hand, it must be clear that MoL processes do not need to run in the same Linux Host, they could be scattered among different servers, acting as a primitive SSI-DOS.

MoL maps each Minix-emulated process to a Linux process covered with an isolation shield around it. The shield handles System Calls and traps Linux signals. When the Minix-emulated process invokes a System Call, the shield intercepts and converts it into a custom RPC using the format of Minix messages. The shield also traps Linux signals addressed to the Minix-emulated process and, depending on the kind of signal and the mask that the process has set, it calls the process handler, ignores it, or notifies the pseudo-kernel about the received signal.

Process scheduling, memory management, exception handling, timer and interrupt management and alarm signaling are handled transparently by the Linux-Host through the process shield.

C. *Interprocess Communications (M3-IPC)*

A critical component of every distributed system is the software communication infrastructure. To simplify the development of a VOS, the M3-IPC [17] infrastructure was developed. It allows building VOS components, such as clients, servers and tasks with a uniform semantics without considering process location. Provider-class requirements were considered in the design stage, such as process replication, process migration, communications confinement, and high 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. It can be used as a component of the DVS or alone as a high performance IPC mechanism for generic (centralized or distributed) applications.

D. *Exokernels and Unikernels*

Generally, in classical virtualization, a Guest-OS is associated with each VM, who offers file, IPC and network services for applications. Several years ago, another form of application processing was proposed using an exokernel [18]. On an exokernel, the applications, the services of a Library OS and device drivers are compiled into a single binary program that is executed using exokernel services. One of the biggest drawbacks of this technology is the incompatibilities of the Library-OS with the diversity of hardware devices. Anyway, this aspect was partially solved because a large part of the libraries used to build the exokernel are the same as to build an ordinary OS or its applications.

A Unikernel [10] is like an Exokernel, but it does not run directly on the hardware, it uses the services provided by a hypervisor (in paravirtualized mode) thus achieving greater portability. This mode of running applications within a VM is very efficient in terms of resource utilization, security and scalability. Usually, a Unikernel does not provide System Calls, it does not differentiate between user-mode and kernel-mode, and it runs all its components with the same CPU privileges.

As it was shown, a Unikernel is a minimalist monolithic kernel, which is custom built for the application that will use it. The application is integrated into the Unikernel itself along with the rest of its required components. It is said to be minimalist because only the components required to execute both the service components (device drivers, filesystems, network protocol stacks, etc.) and the embedded application are included in its code. While this strategy reduces the attack surface against security threats, it should also be considered that the Unikernels often run with kernel privileges (which include the application) so, an error or vulnerability in an application can affect the Unikernel completely. Even worse, if Unikernels communicate among them (a common pattern in Cloud applications) a faulty one can induce errors, defects and inconsistencies in the other members (contamination).

In recent years, possibly due to the resource consumption of VMs, several Unikernel projects emerged.

The Rumpkernel [19] enables developers to build their application components around a software stack. A Rumpkernel could run on several platforms such as userspace POSIX, bare metal, Xen DomU and Genode.

IncludeOS [20] is a library OS that allows building a single task Unikernel. As it is not fully-compatible with Linux some application could need to be modified.

MirageOS [21] is another library OS tool to build Unikernels on platforms such as Xen or KVM hypervisors. It is fully event-driven, but it does not support for preemptive threading.

There exists more Unikernels which can run on a variety of platforms, providing POSIX-compatible APIs or its own APIs, written in different languages such as C, C++, GO, python, Clang, etc.

III. MoL as a Unikernel (MUK)

Generally, a Unikernel is a monolithic piece of software often based on the source code of a monolithic OS. In this way, the application instead of using System Calls to request services, directly calls kernel functions that, in their OS version, implement the System Calls.

On the other hand, a microkernel OS, is formed by a set of autonomous modules with established liabilities and scope. This property allows building more modular Unikernels, using IPC mechanisms as the interface between

its components. Furthermore, it enables the communication of Unikernel internal modules, with other components or applications outside the Unikernel. If the Unikernel runs in a DVS, its components can communicate transparently using IPC with other internal modules or with other Unikernels components of the swarm located in other nodes of a virtualization cluster. If the set of Unikernels and external modules are executed within the context of a DC, they can be executed on any node of the cluster that is covered by the DC. This feature provides simplicity, flexibility and elasticity for the deployment and management of application swarms.

If all the application components are executed within the context of a DC, the deployment, operation and maintenance of the full application are facilitated. There is no need to configure IP addresses and ports to be used in communications between Unikernels and DB servers, nor to configure firewall rules to protect the traffic between them. It is only necessary to configure the IP addresses of the virtual interfaces of the Unikernels to publish web services. With the help of the DVS features, programmers can focus on writing application code instead of managing networks, protocols, containers, storage, VMs and hosts resources.

MUK differs from other Unikernels because it must run in a DVS cluster, which provides it of essential services to simplify the programming of distributed and fault-tolerant Cloud applications.

A. MUK Prototype Design

MUK prototype is made up by a single Linux process with multiple threads. The use of at least one thread for the execution of each server or task was established as a design principle.

A well-known property of threads belonging to the same process is that they share global variables, functions and all host-OS allocated resources (such as file descriptors). Having considered this property at the design stage, the possibility of building an internal IPC mechanism among MUK threads was evaluated. Threads have the advantage of sharing memory, so those IPC copy operations would be avoided. But, as a counterpart, mutual exclusion and synchronization mechanisms will be required to build the internal IPC, which meant transferring the control to the host-OS kernel. Therefore, this approach was discarded. It was established that the IPC mechanism to be used in MUK would be M3-IPC with some copy operations as exceptions to this rule.

The following are the components of the MUK prototype:

- *System task (SYSTASK):* It manages process descriptors and their endpoints. It also allows the copy of data blocks between different processes by controlling the privileges established for the requester process.
- *Clock Task (CLOCK):* It provides *SYSTASK* with timer functions and other time related services.
- *Process Manager (PM):* It allocates PIDs to each process in MUK namespace. It also notifies events through signals to the destination processes.
- *File* System *Server (FS):* It manages directories and files stored in block devices [22].
- *Disk Task (RDISK*): It manages a virtual disk mapped on an OS-host regular file. *RDISK* can run replicated on several nodes, but this option was disabled [23].
- *Information Server (IS) and Web Information Server (WIS)*: It allows to get configuration, status and statistical information of each MUK server or task. It can be visualized in plain text mode (IS) or in HTML (*WIS*) format through a web browser.
- W*eb Server (NWEB)*: It is a web server of static web pages stored by the FS. In the current version, *NWEB* network access is made using the host-OS TCP/IP protocol stack.
- *File transfer server M3-IPC (M3FTP):* It allows file transfers to/from an external process belonging to the same DC from/to a file stored in MUK filesystem. The communication protocol used for the transfers is M3-IPC (not TCP/IP).

The main MUK process with all of its threads, are executed inside a Container of the OS-host which is a member of a DC. In this way this property enables that any MUK thread can communicate with other external processes (local or remote) of the same DC.

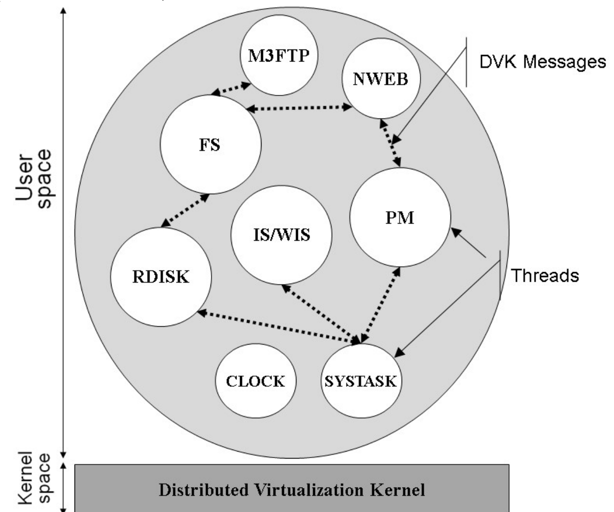

Figure 3. MUK Prototype Architecture.

B. MUK Prototype Implementation

One benefit of any system made up by a set of autonomous modules is that they can communicate through well-established interfaces. This positive feature presents as a negative one when trying to merge all of these modules into a single one, breaking the isolation between them. One of these issues refers to the names of global variables because they are shared among the threads of the same process. For example: The *RDISK* driver uses a global variable called *buffer*, but as the *FS* server also uses its own global variable called *buffer* a name collision happens. Therefore, the global variables with the scope of a thread had been renamed to avoid name collision with other. In this

way, the *buffer* variable of *RDISK* was renamed *rd_buffer* and that of *FS* is now *fs_buffer*.

A similar situation happens with the name of some functions. For example, the function that handles the *fork()* system call in *PM* is called *do_fork()*, same as in *FS*. These functions were renamed as *pm_do_fork()* and *fs_do_fork()* respectively.

The *SYSTASK* has a shared memory area with the DVK through the */sys/kernel/debug* virtual filesystem. In this memory area, the DVK stores the process descriptors of each DC. Due to MUK tasks and servers are implemented using threads, they share a memory space among them. Therefore, process descriptors can be directly accessed from any other MUK thread, or they can use *memcpy()* to get a copy of them.

When MUK starts, its main thread starts the set of component threads (tasks or servers) sequentially. Each thread initializes its state, and registers itself with an endpoint to the DVK (*dvk_bind()*) in order to use M3-IPC. Once it is ready to serve its clients, it notifies the main thread using Linux provided condition variables and mutexes.

MUK can be build integrating several tasks and servers, but a configuration file is used to specify which threads will be started and registered to the DVK. Each MUK instance, must belong to a DC (*dcid* entry), and each of its threads must have its own endpoint number (*xxx_ep* entries) (Fig. 4). As each server needs other configuration parameters, they can be configured using their own configuration files.

```
muk SERVER1 {
  dcid            0;
  pm_ep           0;
  fs_ep           1;
  is_ep           8;
  rd_ep           3;
  web_ep          22;
  ftp_ep          14;
  fs_cfg          "molfs.cfg";
  rd_cfg          "rdisk.cfg";
  web_cfg         "m3nweb.cfg";
  ftp_cfg          "m3ftp.cfg";
};
```

Figure 4. Example of MUK configuration file.

MUK also have its own tools to get thread information. Sending a SIGUSR1 signal to the IS thread, it prints to an output file the information of DC configuration parameters, *SYSTASK's* process descriptors, *PM's* process descriptors, *FS's* process descriptors, *FS's* superblocks of mounted devices, etc.

The *IS* thread starts another thread for the *WIS* server. It enables getting MUK information in HTML mode from a Web Browser. Using the same approach of the provided servers, any application included into the MUK code could be monitored using the text mode o html mode interfaces.

Two main server applications were included into the MUK prototype: a web server and a *M3FTP* server. The web server only uses static web pages stored in files into the *FS*.

The current version of MUK prototype was implemented on Debian 9.4 (called "*stretch*") with a Linux kernel version 4.9.88. When MUK is running, its (virtual) memory usage is about 89 Mbytes, with all the servers and task presented in this paper as its components. The size of the MUK executable file is only 852 Kbytes.

IV. Prototype Evaluation

To evaluate the performance and the operation of the prototype, two DVS nodes were used with the following characteristics:

- *NODE0*: AMD A6-3670, 2.7 GHz, 8GB RAM.
- *NODE1*: Intel(R) Celeron(R) CPU G1820, 2.7 GHz, 4 GB RAM.

The nodes of the DVS cluster were connected through a dedicated 1 Gbps LAN Switch. A notebook (Intel Core I5-7200U CPU 2.71 GHz, 4 GB RAM) with Windows 10 was used as the client for HTTP file transfers also connected to the same LAN switch.

The data and message transfers between *NODE0* and *NODE1* were performed using TIPC-based [24] communication proxies. Although TCP or UDP proxies could be used, the TIPC proxies showed superior performance in previous evaluations [17]. The current TIPC proxies have the following additional features such as message batching and data compression.

To verify the versatility of MUK, it was tested in three scenarios with different configurations:

- *Config-A*: All components are integrated on MUK, including the *RDISK* task.
- *Config-B*: All components with the exception of *RDISK* are integrated on MUK. *RDISK* runs as an external *local* storage server.
- *Config-C*: All components with the exception of *RDISK* are integrated on MUK. *RDISK* runs as a *remote* storage server.

Several types of benchmarks were performed for each configuration but, by space limitations, the following is reported here: HTTP file transfer from a remote web Client (*wget*), where remote means "*on another computer*" in the same network (LAN). The same benchmarks were done on an Apache web server as a reference for performance comparison.

The file transfer benchmarks were done several times with files sizes of 10 Mbytes and 50 Mbytes. The displayed values are averages of the measurements. As all benchmarks were done with dedicated computers and LAN switch, the standard deviation of the results was negligible and therefore, it is omitted here. The *RDISK* task was configured to use an "*in memory*" disk image file and the Apache web server used a document directory in a RAM filesystem to eliminate the stochastic values of hard disk latencies.

The throughput of a remote Client file download is presented in Fig. 5. In this case, the gaps between of Config-A, Config-B vs. Apache are less noticeable because in all cases the HTTP communications over the network between the remote Client and the web Server has greater impact. Config-C is doubly affected by data transfers over the network with RDISK running on other node.

Another important metric for Unikernels is the boot time: MUK process, with all of its threads, is started inside a DC (which includes creating a Linux Container) in 22 [ms].

V. Conclusions and Future Works

Minix's architecture based on a microkernel and device drivers in user-mode has influenced in such a way that INTEL has integrated it as part of several of its CPU chips [25]. It uses message transfers to communicate processes, servers and tasks. Those features facilitated its port to user-space such as MoL. On the other hand, a DVS allows the configuration of isolated execution environments (DCs) (which can cover several related Linux Containers each), scattered on nodes of a virtualization cluster.

The primary contribution of this work is to present MUK, a Unikernel based on user-space multi-server VOS (MoL) built to run on a DVS. MUK improves its security features running within a DC which can cover one or more Linux Containers. MUK components run integrated in one process, but can use external services running on the same host or distributed in several nodes of the DVS cluster without changing the applications. The communication of MUK components with other external (local or remote) servers is transparent about server locations. From the operational viewpoint, the MUK prototype works according to its design specifications: simple, elastic, easy to deploy and manage.

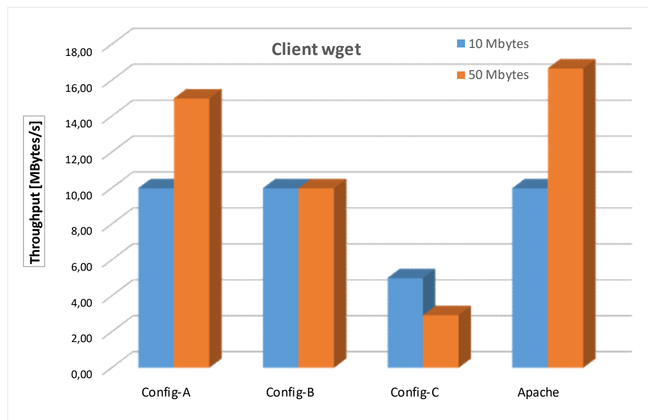

Figure 5. Client wget Throughput.

As MUK is POSIX-compatible it allows fast application port and, because it is multi-server, more than one component could run within the same Unikernel. As running MUK on a DVS facilitates the deployment of Cloud applications, programmers can focus on writing application code instead of managing networks, protocols and hosts.

To fully test DVS features in several scenarios, other VOS projects were started and remain in progress. The MUK version presented in this article runs its servers as threads, but at the time of this writing they can run as co-routines [26]. A project just finished allows running *UML* [27] on the DVS and an ongoing project is about running a *rumpkernel* [19]. Running *UML* instances and *rumpkernels* concurrently on a DVS will allow executing a wide range of existing applications and it will definitely convince the community about the advantages of using a DVS to deploy applications.

References

[1] N. Bieberstein et al., "*Service-Oriented Architecture Compass*", Pearson, ISBN 0-13-187002-5, 2006.

[2] 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.

[3] SOAP, https://www.w3.org/TR/?title=soap, last access at January 2019.

[4] J. Turnbull, "*The Docker Book*", 2014, Available online at: https://www.dockerbook.com/, last access at January 2019.

[5] N. Poulton, "*The Kubernetes Book*", ISBN-13: 978-1521823637, ISBN-10: 1521823634 ,2017.

[6] B. Hindman, et al., "*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), Berkeley, CA, USA, 2011.

[7] P. Kamp, R. N. M. Watson, "*Jails: Confining the omnipotent root*", in Proc. 2nd Intl. SANE Conference, 2000.

[8] D. Price, A. Tucker, "*Solaris Zones: Operating System Support for Consolidating Commercial Workloads*", in 18th Large Installation System Administration Conference,2004.

[9] W. Felter, et al., "*An Updated Performance Comparison of Virtual Machines and Linux Containers*", IBM Research Report, 2014.

[10] A. Madhavapeddy, et al., "*Unikernels: library operating systems for the cloud*", Proc. of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13), 2013.

[11] Anil Madhavapeddy and David J. Scott. 2013. "*Unikernels: Rise of the Virtual Library Operating System*", Queue 11, (Dec.2013).

[12] Unikernel.org; http://Unikernel.org/ last access at January 2019.

[13] P. Pessolani, O. Jara, "*Minix over Linux: A User-Space Multiserver Operating System*", in Proc. Brazilian Symposium on Computing System Engineering, Florianopolis, 2011.

[14] P. Pessolani, F. G. Tinetti, T. Cortés, and S. Gonnet, "*An Architecture Model for a Distributed Virtualization System*", Proceedings of the Ninth International Conference on Cloud Computing, GRIDs, and Virtualization (Cloud Computing 2018), págs. 1-11, 2018.

[15] Oikawa, M. Sugaya, M. Iwasaki and T. Nakajima, "*Using virtualized operating systems as a ubiquitous computing infrastructure*", Second IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, Vienna, 2004.

[16] OpenSSI (Single System Image) Clusters for Linux, http://www.openssi.org/cgi-bin/view?page=openssi.html, last access at January 2019.

[17] P. Pessolani, T. Cortes, F. G. Tinetti, and S. Gonnet, "*An IPC Software Layer for Building a Distributed Virtualization System*", in CACIC 2017, La Plata, Argentina, October 9-13, 2017.

[18] D. R. Engler, M. F. Kaashoek, J. O'Toole Jr., "*Exokernel: an operating system architecture for application-level resource management*", in Proc. 15th ACM SOSP, Copper Mountain, 1995.

[19] Rumpkernel, http://rumpkernel.org/, last access at January 2019.

[20] IncludeOS, https://www.includeos.org/, last access at January 2019.

[21] Mirage, https://mirage.io/, last access at January 2019.

[22] D. Padula, M. Alemandi, P. Pessolani, S. Gonnet, T. Cortes, F. Tinetti, "*A User-space Virtualization-aware Filesystem*", in CoNaIISI 2015, Buenos Aires, 2015.

[23] M. Alemandi, O. Jara, "*Un driver de disco tolerante a fallos*", (in Spanish) Jornada de Jóvenes Investigadores Tecnológicos (JIT 2015), Rosario, 2015.

[24] J. P. Maloy, "*TIPC: Providing Communication for Linux Clusters*", Proceedings of the Linux Symposium, 2004

[25] A. Tanenbaum, "*An Open Letter to Intel*", https://www.cs.vu.nl/~ast/intel/, last access at January 2019.

[26] Libtask: a Coroutine Library for C and Unix. https://swtch.com/libtask/, last access at January 2019.

[27] J. Dike, "*A user-mode port of the Linux kernel*", USENIX Association. Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta Oct 10 -14, 2000.