



# **BOOSTING THE CLOUD META-OPERATING SYSTEM WITH HETEROGENEOUS KERNELS. A NOVEL APPROACH BASED ON CONTAINERS AND MICROSERVICES**

D Bikshapathi, SWAPNA PADAKANTI, ABBA CHETHANA

Assistant Professor, Department of Computer Engineering, Ellenki college of Engineering and Technology, patelguda (vi), near BHEL ameenpur (m), Sangareddy Dist. Telangana 502319

## **ABSTRACT**

**high SLA is a big challenge for the Cloud providers as they have to ensure the sustainability of their customers' workloads that depend closely on the undelying OS. The kernels are the cores of the OS, and the monolithic kernels**

### Introduction

The Operating Systems provide services to run the applications. These services are invoked via system calls implemented into libraries and APIs.

With the emergence of the Cloud, the service providers undertake on SLA levels too high to run the customers' workloads. This SLA can be closely linked to the quality of the OS that must be both reliable and efficient.

The model without exceptions [1] for example has shown how the design of the OS can improve the performance of applications.

The operating systems are based on the kernels which form their cores. Different approaches of design of the kernels are proposed. Nevertheless, the monolithic kernels

[7] like Linux kernels are the most powerful. Their fragility is due to the fact that the kernels include the majority of the critical features (to avoid the switches of context and gain performance). The slightest flaw in a module of the kernel may entail its interruption and the break of the entire operating system.

Today, hardware virtualization is required as a compelling solution for the consolidation of workloads and the sharing of physical resources. This technology is even supported natively in the hardware. Technologies like Intel- VT/AMD-V and SR-IOV can be cited as

examples. The virtualization, in addition to the benefits of consolidation, offers by conception an ideal isolation of virtual machines. Each kernel of a given virtual machine runs in a secure and separate space independatly of other kernels. Consequently, thanks to the virtualization, multiple heterogeneous systems can coexist on the same host.

independently. Applying some principles like the Single Responsibility Principle, changes made to a microservice don't affect the integrity of other microservices.

Hardware virtualization can offer a solution to improve the reliability of the OS. This hypothesis is supported by VirtuOS [2] which defines an architecture of an operating system subdivided functionally in service domains. Each service domain manages a functionality, like storage, network, etc. and is executed in a virtual machine on the Xen hypervisor to offer a perfect insolation and security. As well, the system calls of the applications that run in the primary domain are routed to the adequate service domain. Thanks to this architecture a flaw that is registered at a service domain like a malfunction of a driver does not cause the break of the entire system. The restart of the failed domain does just affect partially the process waiting for replies from it. Other solutions implement the same priciple like the domain-0 disaggregation in the new generation of Xen Server called Windsor. Qubes is also an OS containing the same architecture as VirtuOS but it is more mature and even workable in practice.

on the other hand, Docker [6] is considered as promising OS virtualization technique that acts

at the OS level by abstracting the process execution. Although the containerisation is an old technology in the Unix/Linux systems, docker provides a very interesting layer of features like the containers migration from a version of the kernel to another version or the application of resources constraints on the containers, etc. via firendly tools.

The combination of the two virtualization technologies (hardware and OS) by running containers in virtual machines provides better isolation and reliability.

The purpose of our work is to extend the architecture of VirtuOS by making collaborate several heterogeneous kernels considered as domain services. We should apply the principle of the scale cube [5] by splitting the APIs according to the Y axis in small APIs executed in containers as microservices [6] or simply as processes hosting Web Services. These small APIs will be executed on lightweight

virtual machines (domain services) based on heterogeneous kernels.

This heterogeneity may engendrer a Meta-Operating System more reliable and efficient in bringing the benefits of heterogeneous kernels. The Meta-Operating System can be seen as an overlay of kernels where the kernel X offers the services of the network via its drivers as well as the implementation of the TCP stack while the kernel Y offers the service of storage for access to files for example.

Our Meta-Operating System's kernels could be virtual machines on the same host but we intend to extend this feature to discover other machines and domain services in the network. The Web Services as the implementation technology of microservices are a good candidate to respond to this need through standards like WS-Discovery.

The defined objectives in our work are the following:

- Define the architecture of a Meta-Operating System based on several domain services hosting heterogeneous kernels in the form of virtual machines co-existing on the same host or discovered in the network.
- Define the architecture of an API subdivided in small domain services APIs. These small APIs will be developed as

microservices and executed within containers or as Web Services hosted into process. The system calls routing will be abstracted by remote calls of methods of Web Services.

- Define a proactive supervision mechanism so that the Meta-Operating System isolates the faulted domain services prior to prevent any performance degradation and increase reliability. This same mechanism would be able to migrate any microservice during the execution from a suspected service domain to another domain more reliable even on the basis of kernels of different versions.

- Define an intelligent mechanism for dispatching the remote system calls towards local or remote service domains taking into account two parameters: Reliability and Performance.

The rest of the paper is organized as follows:

- Part 2 : Meta-Operating System architecture design.
- Part 3 : Experimental tests results.
- Part 4 : Quick related work presentation.
- Part 5 : Conclusion.

## 1. The Meta Operating System Based on Heterogenous Kernels

VirtuOs [2] and WSNFS [8] are two different solutions that have in common the externalization of the execution of system calls with a possibility to reuse non existing functionalities on the original environment that are offered by the remote domains.

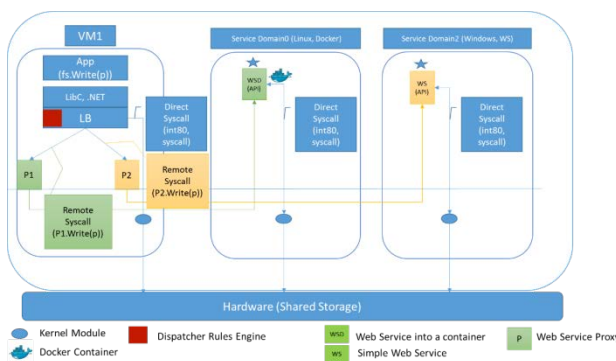
VirtuOS externalizes the system calls towards the service domains to improve the whole system reliability. Whereas, WSNFS aims to abstract heterogenous file systems by developing a single driver using Web Services. This will also allow to the solution to be more flexible and reliable when adding more WSNFS gateways and drivers.

Both solutions are the basis of the Meta-Operating System whose kernel consists of several heterogeneous kernels co-existing on a hypervisor or remote kernels discovered in the network. The Meta-OS characteristics are as follows:

- The Meta-OS is composed of varied and heterogenous service domains. Every domain

implements a feature or set of features based on a given kernel.

- The offered features are exposed via entry points that could be invoked using modified APIs.
  - Automatic discovery mechanism of new service domains or new features in existing domains.
  - Intelligent system calls routing from the domain where the users' processes are running towards remote service domains, taking into account two parameters: reliability and performance.
- inside containers. Different protocols can be used to invoke the frontend APIs like TCP, HTTP, REST,



- A proactive monitoring mechanism so that the Meta-OS isolates the faulty domain services before recording a performance degradation event.
- To reuse remote service domains' functionalities, an intelligent API is designed to dispatch system calls to the adequate domain service. This API has to abstract the dispatching system end ensure the existing applications portability.

Consequently, the Meta-OS can be seen as an overlay of multiple kernels.

The Meta-OS is based on two categories of domains:

- Service domains: they aim to provide services to applications and other domains. These domains can be instantiated, started, stopped, paused, cloned and migrated upon request by the Meta-OS subsystem.
- Applications domains: they aim to provide the execution runtime for the users' applications. They can be considered as customer domains consuming services from

service domains but they can play the role of service domains.

A. Principle of system calls' externalization  
 Monolithic applications present some drawbacks like debugging difficulties and the Single Point of Failure design. Consequently, some patterns are invented to divide monolithic applications into small manageable modules. The microservices [6] are one of the paradigms that aim to achieve these objectives. The microservices are the small modules that collaborate and communicate using standard protocols like HTTP/Rest.

Each microservice is developed and deployed independently. Applying some principles like the Single Responsibility Principle, changes made to a microservice don't affect the integrity of other microservices.

In our case, the APIs can apply the microservices architecture, where each API is composed of two parts (Fig. 1):

- Frontend API: this part encapsulates the system calls invocation routines to local and remote service domains and a load balancer engine (LB) that aims to dispatch syscalls to backend APIs. As these latteres are executed into Web Services, the Frontend API implements proxies to invoke them. The Frontend API aims also by abstracting the backend routines to ensure the portability of existing applications.
- Backend APIs: each backend API implements a subset of functionalities inside a service domain. They will be executed into Web Services hosted on processes or

Calling the Write method on the object fs (FileStream type) can generate the routine of the system call directly on the local host, an invocation of the backend API as a Web Service hosted in a container via the proxy (P1.Write) or finally an invocation of the backend API as a Web Service hosted in a process via the proxy (P2.Write).

B. The Meta-OS architecture  
 The following figure presents the architecture of the Meta- OS grouping two scenarios:

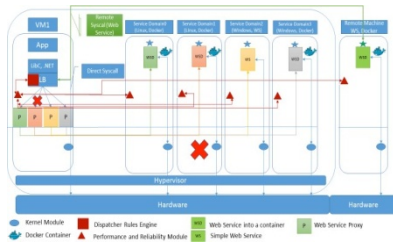


Fig. 2. The Meta-OS architecture

1) Scenario1: co-existed domains on the same host This scenario is based on the architecture of VirtuOS. Thanks to advanced hardware based virtualization technologies, the domain services can own a part of hardware resources like in SR-IOV. The hypervisor ensures critical tasks like memory management, tasks scheduling, etc. Each domain integrates a performance and reliability module to instrument the syscalls without causing any overhead. Some tools could be used to achieve this goal like Sysdig. The performance module ensures also the discovery of other domains performance modules, consolidating all the data and maintaining references to the discovered service domains.

Unlike VirtuOS, the syscalls dispatching system is not accomplished in kernel mode but is abstracted by the remote Backend APIs invocation in user mode. We are aware of the performance degradation but this can be improved using the Hypervisor bus as communication medium between the co-hosted domains.

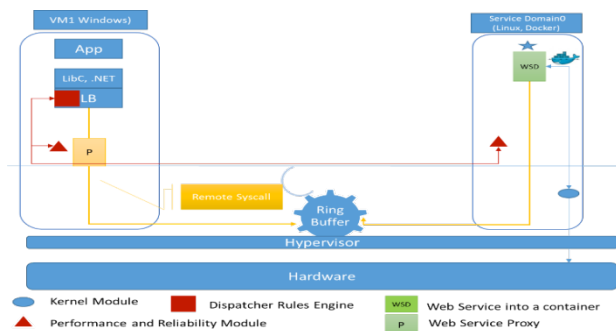


Fig. 3 Syscalls processing via the hypervisor bus channel.

2) Scenario2: discovered remote domains in the network This scenario is based on the architecture of WSNFS. The processing of the

syscalls is externalized via the remote Web Services executed into containers or hosted directly by processes.

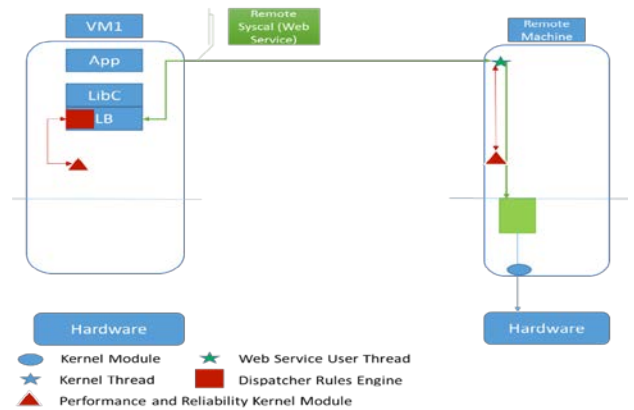


Fig. 4. Syscalls processing

C. The intelligent API

The frontend API is so crucial in the process of syscalls invocation. It implements a business rules engine that helps in selecting the best available service domain for the current thread and transforming if needed the generated syscall routine to be supported by the selected domain. To simplify the processing of future syscalls for a given thread, an affinity is maintained through a table that associates a thread to its corresponding service domains.

Three domains detinations types are defined:

- Local handler
- Remote handler on a virtual machine via the shared bus.
- Remote handler on a virtual machine via the network.

- User Time: time required to capture or restore in user mode.
- Kernel Time: time required to capture or restore in Kernel Mode.
- Real Time: the whole time required to capture or restore taking into account interruptions.

3. Conclusion

In this paper we tried to present the architecture of a Meta- Operating System based on heteregenous domain services discovered in the network or running on the same host.

The aim of this Meta-Operating System is to reap the benefits of all the discovered service

domains by developing a technique of dispatching syscalls routines towards the most adequate service domain. This technique is leveraged by decomposing the APIs into frontend APIs running in the original applications domain and backend APIs that expose syscall routines services.

To underpin our approach, we developed a prototype by extending the classic API file processing. Backend APIs as microservices run directly in storage domain services or in containers.

We measured performance data in different situations and obtained very promising results that could be optimized using advanced techniques like GPGPU [15] to accelerate messages processing, Exceptionless syscalls to eliminate the switching mode overhead, etc.

This is an Open Access article distributed under the terms of the Creative Commons Attribution Licence

#### References

1. Soares, L., & Stumm, M. (2011, June). Exception-Less System Calls for Event-Driven Servers. In USENIX Annual Technical Conference (Vol. 10).
2. Nikolaev, R., & Back, G. (2013, November). VirtuOS: an operating system with kernel virtualization. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (pp. 116-132). ACM.
3. Rutkowska, J., & Wojtczuk, R. (2010). Qubes OS architecture. Invisible Things Lab Tech Rep, 54.
4. Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2.
5. Martin L. A. & Michael T. F. (2015). *Art of Scalability, The: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, 2nd Edition. "Addison-Wesley Professional, Inc."
6. Newman, S. (2015). *Building Microservices*. "O'Reilly Media, Inc."
7. Abraham S., Peter B.G. & Greg G.(2013). *Operating System Concepts*, Ninth Edition. "John Wiley & Sons, Inc."
8. Hwang, G. H., Yu, C. H., Sy, C. C., & Chang, C. Y. (2008). WSNFS: A Web-

Services-Based Network File System. *J. Inf. Sci. Eng.*, 24(3), 933-947.

9. Borchert, C., & Spinczyk, O. (2016). Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-

program analysis. *ACM SIGOPS Operating Systems Review*, 49(2), 37-43.

10. Colmenares, J. A., Eads, G., Hofmeyr, S., Bird, S., Moretó, M., Chou, D., ... & Asanović, K. (2013, May). Tessellation: refactoring the OS around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference* (p. 76). ACM..

11. Wentzlaff, D., & Agarwal, A. (2009). Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2), 76-85.

12. Jacobsen, C., Khole, M., Spall, S., Bauer, S., & Burtsev, A. (2016). Lightweight capability domains: towards decomposing the Linux kernel. *ACM SIGOPS Operating Systems Review*, 49(2), 44-50.

13. LeVasseur, J., Uhlig, V., Stoess, J., & Götz, S. (2004, December). Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *OSDI* (Vol. 4, No. 19, pp. 17-30).

14. Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., & Isaacs, R. (2008, June). Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems* (p. 27).

15. Jordan, V. (2010). XML query processing using GPGPU (Doctoral dissertation, Master's thesis, University of Tsukuba & Université de technologie de Belfort-Montbéliard).