Unified Monitoring and Analytics in the Cloud

Ricardo Koller, Canturk Isci
IBM T.J. Watson Research

Sahil Suneja, Eyal de Lara
University of Toronto

Abstract

Modern cloud applications are distributed across a wide range of instances of multiple types, including virtual machines, containers, and bare metal servers. Traditional approaches to monitoring and analytics fail in these complex, distributed and diverse environments. They are too intrusive and heavy-handed for short-lived, lightweight cloud instances, and cannot keep up with rapid the pace of change in the cloud with continuous dynamic scheduling, provisioning and auto-scaling. We introduce a unified monitoring and analytics architecture designed for the cloud. Our approach leverages virtualization and containerization to decouple monitoring from instance execution and health. Moreover, it provides a uniform view of systems regardless of instance type, and operates without intervening with the end-user context. We describe an implementation of our approach in an actual deployment, and discuss our experiences and observed results.

1 Introduction

The cloud is dynamic and difficult to understand when looking at all its pieces independently. A typical cloud application is made of components which are rapidly created, relocated, and destroyed to adapt to different load, for multiple tasks, and simultaneously for multiple tenants. Moreover, with modern, cloud-native applications, the information needed to solve problems are now distributed across an increasingly large and diverse set of sources, including bare-metal servers, virtual machines (VMs), and containers. This complexity makes it harder for cloud providers and tenants to have an understanding and complete view of their systems. Tenants can rent systems or software in any of these units and should expect all of them to be monitored with the same level of detail, low overhead, and seamless effort. Traditional monitoring and analytics solutions operate within their silos and rely on long-running, stable system characteristics. In contrast, with the diverse, and highly-dynamic nature of cloud resources and the applications running on top, there is need for a new, unified monitoring approach that is designed for the cloud, which can collect information from every relevant piece of the cloud infrastructure and can keep up with the diversity, density and dynamism of cloud. Such an approach is critical for delivering management and analytics solutions like compliance scanning, health-check, license management, and root cause analysis for both cloud tenants and the providers.

Existing monitoring and analytics solutions follow one of three approaches: 1. In-system agents for data collection and interpretation; 2. Hooks that enable access into systems or application run-time context; and 3. Limited black-box monitoring that limits observations to what is available outside of system context. The first two approaches are highly intrusive and require guest cooperation. Hooks that enable system access are points of vulnerability and agents begin to become too heavyweight to be practically applicable for lightweight virtualization and containerization. Moreover, in-system solutions can easily be compromised and cannot be trusted. The latter black-box approaches are less intrusive, and less prone to compromise, but provide limited visibility. An ideal solution combines the best of both worlds, providing deep, seamless visibility into systems, without any side effects, overheads or intrusion. Our approach to cloud monitoring aims to hit this sweet spot, providing deep, seamless and secure visibility into both VMs and containers—with the same fidelity, and without intrusion or overheads on guest environments.

We present a new approach that leverages virtualization and containerization to decouple monitoring and analytics from guest execution context. This decoupling enables us to run our monitors without impacting the end user environment, or requiring any form of cooperation. By effectively interpreting VM and container state from outside the guest context, we can provide deep and seamless visibility into these systems. In our solutions, we leverage VM introspection (VMI) principles for monitoring VMs. VMI is the process of extracting logical system information from raw memory and disk data structures. For containers, we use underlying *namespace* and *cgroup* APIs to map and monitor state of different containers. Our results show the feasibility and practical application of this approach on an actual cloud deployment. We show that under realistic assumptions, the described out-of-band monitoring of VMs and containers is not only viable, but has significant advantages over traditional approaches.

This paper makes three contributions. First, we design and implement an architecture of a unified pipeline for operational analytics in the cloud. In our architecture we put particular emphasis on the data collection layer, and show that the most viable option for seamless and lightweight monitoring in the cloud is based on out-of-band VMs and container monitoring. Second, we describe our VMI and namespace mapping techniques for VM and containers respectively, and show that they can be applied in practice in actual cloud deployments running KVM/OpenStack and Docker containers. Finally, we present preliminary performance numbers, and results for a prototype application for network monitoring.

2 Background on monitoring and applications

Consider the following situation. There is a distributed web service, running on a LAMP (Linux-Apache-Mysql-PHP) setup. Let us assume that we install this setup correctly and place Mysql and the PHP frontend in two VMs, and the Apache server in a container. At some point, the web service suddenly stops servicing requests, and all that the users get is a `500 Error`. The reason for this failure could be that the Apache server is using the wrong port for connecting to Mysql, lack of disk space for any of the services, or because of some missing library in the PHP front end. The information necessary to find the root cause of any of these problems is distributed across a large set of configuration files, log files, and within the state of the two VMs and the container. To quickly understand what changed, we need an approach that can monitor across all these systems irrespective of the platform, and help us correlate without overwhelming our run-time environment.

Our approach to troubleshoot scenarios like these is to compare the global state at the moment the problem occurs, to a good state when the system was just installed, or functioning normally. Figure 1 shows the proposed architecture of a pipeline capable of implementing such kind of applications. A set of collectors based on introspection of VMs and containers emit system data to a scalable data bus. An indexer consumes data from the data bus and serves as the basis for building a search service. An analytics application (”Drift detection” in the figure) would use this search service to compute the differences in state and configuration, between the good and the bad setup, and will then sort the *diffs*, and filter out the ones that should not affect system behavior. Sorting and filtering *diffs* is research in progress at the moment.

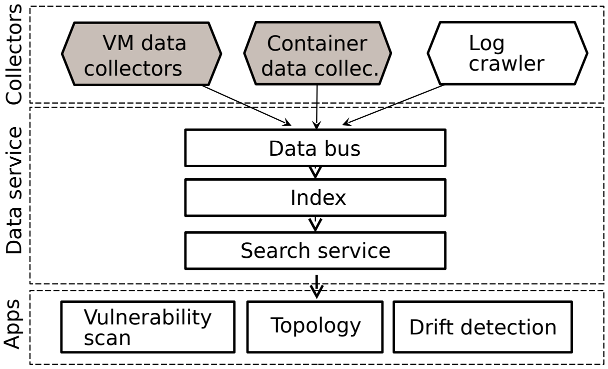

Figure 1: Architecture of the unified operational analytics pipeline. This paper focuses on the shadowed boxes. The *topology* application is used later in the evaluation as a proof-of-concept application.

Our main focus in this paper is on the data collection layer of the analytics pipeline. The target data we are interested in collecting includes: processes, network connections, process map details, modules loaded, performance metrics (memory, CPU and IO), configuration files, packages installed, and file system information (like the last access time of a file).

3 Unified Monitoring

In order to minimize the cost of ownership, cloud providers try to increase the density of virtual machines per host. With this in mind, there have been many efforts of trying to make VMs more lightweight: library OSes [10, 6], lightweight Linux distributions [2], and more recently a renewed interest in OS level virtualization with containers or jails [3]. As VMs and containers become more lightweight and transient or ephemeral, minimizing the over-heads and setup times for monitoring them become more critical.

In addition, existing agent-based monitoring approaches have three important drawbacks. **First**, they require the monitored system to be live and healthy to be able to collect and send data. **Second**, the guest OS can be compromised with a kernel rootkit, so the agent would be getting incorrect information, possibly because of the rootkit trying to hide itself. And, **third**, agents need to be installed or at least baked into the VM images, which adds unnecessary complexity that is pushed onto the tenants of the cloud, where they have to install and maintain these agents and their dependencies in their instances and base images.

We propose a new unified monitoring approach for the cloud that provides a uniform view of systems regardless of instance type by leveraging virtualization and containerization abstractions to decouple monitoring from guest execution context. For VMs, we use VMI techniques and for containers we use underlying OS APIs to inspect system state encapsulated by the container. For brevity, we refer to both techniques as VM and container introspection respectively. Both techniques share some common principles. They both operate without requiring guest cooperation, without installing any components in end-user context, and both can function without relying on guest health. We discuss the details of these in sections 3.1 and 3.2.

3.1 VM Introspection

Our VM monitoring solution is based on exposing and interpreting raw guest memory and disk state from the underlying hypervisor platform [4]. We use well-known data structure layouts to extract the logical view of the VM state. For example, to determine the hostname for a Linux guest, we identify the OS data structure that holds this information and then determine its memory location. In the case of `hostname`, this is stored as a string (`.name`) inside the `struct init_uts_ns` data structure. In order to read this string we need the address of the structure- `struct init_uts_ns`- in kernel memory, the offset from the start of the structure to the `.name` field, and access to the memory of the guest.

Our technique is based on source-code analysis principles [5], and is a type of rule hand-crafting. As our experiences show, this can be very effective and practically applicable with some basic assumptions on cloud instances. The idea is that if we had access to the kernel source code, we could setup a rule for reading the hostname of a Linux guest as: `*UTS_BASE + UTS_OFFSET`. This rule would be valid for any Linux kernel, but each one of them would have a specific value for `UTS_BASE` and `UTS_OFFSET` based on the source code and the compilation configurations.

The problem now is that this rule might change. What if a newer version of the Linux kernel stored `hostname` on a hash-table of system configurations instead? For example, `hostname` location actually changed in kernel `2.6.19` [9] with the introduction of `hostname` namespaces. However, this change still has the form: `*BASE + OFFSET`, for `hostname` at least. This is the offset rule used in our system:

```
["UTSNAME_OFFSET", [
    ["struct new_utsname", "sysname"],
    ["struct uts_namespace", "name.sysname"]]
]
```

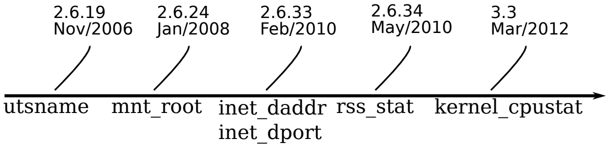

Figure 2: Timeline of relevant Linux kernel source code changes.

Although more drastic changes can occur and our system would be broken if they did, it turns out that changes to core data structures like these in modern and well established OSes like Linux are very rare. We have rules for Linux kernels spanning from `2.6.11` to `3.19`. These are versions spanning from the year 2005 to 2015. Figure 2 shows the changes affecting our rules for the key system features we track (i.e., processes, connections, modules, files, packages, configurations, loaded libraries and system metrics). We extract data from 96 data structures to collect all this information. Out of these 96 structures, only 6 have changed across these kernel versions spanning a decade.

The only issue now is that we need the debugging symbols and the symbol table. They are available as long as it is a Linux distribution and the packages are available (we limit the discussion to Linux OSes). These symbols might be unavailable if, for example, the cloud images were using a custom kernel. However, due to standardization, most cloud offerings have a limited number of VMs operating systems, so the number of hand written rules for VMI should be limited.

Another problem with VMI is inconsistencies with monitoring due to the lack of synchronization between the outside of the VM and the inside. For example, creating a process involves many updates to memory, if we get the state in the middle of any of those, we get an inconsistent state. These inconsistencies are due to not pausing the VM, and also due to the fact that the operations are not atomic [12]. This can happen for example for process reaping: which consists of first marking the process as dead (an atomic operation), followed by sequentially deleting all the members of the process structure. Some techniques to alleviate this are using transactional memory [8]. The only source of inconsistencies is process reaping [12] which can be a long process given that it requires parent reaping the child when reading the exit status. Inconsistencies can lead to missing data at worst, but not to wrong information.

3.2 Container introspection

OS level virtualization (`chroot`, Linux Containers, FreeBSD Jails, OpenVZ [11, 3]) is a type of virtualization that has gained some traction in recent years. When using this type of virtualization, some processes can be grouped into containers, and each container be given a unique view of the system. For example, there can be a host running a process with `PID 1` with access to `NIC eth0` with `IP 1.1.1.1`. Simultaneously, there can be a different process with `PID 100` running in a container with a different view of the system: it sees itself with `PID 1` and using `eth0` with `IP 2.2.2.2`. One consequence of the fact that container processes are just host processes with a different view of the system is that they are visible from the host.

Unlike VMs, where the only options to get into the context of a VM is to log in into it (i.e., using an ssh) or do introspection of memory and disk state, containers are more convenient for introspection. Given that all processes are visible from the outside, the only missing information is the mapping between resources as seen from inside the container and the outside, for example, mapping the process with `PID 100` to `PID 1` in the example above. In order to monitor the processes in this virtualized system, all we need is a list of all the processes associated with it, and the mapping between processes' PIDs as seen from inside the container and from the host.

This mapping has to be stored somewhere in the kernel as it is needed for accounting of container resources. As an example, in Linux this mapping can be easily observed with the `/proc` file system but the method we propose for monitoring containers and jails is to use attachment operations. `setns()` in Linux (or `jexec()` in freebsd) allow a process to attach to an existing *process namespace*. The monitor process then just attaches to a process namespace for each container, collects data from it, and attaches to the original namespaces.

Although this is not exactly introspection, as the data collector process is "moving" to the container context, it still provides the qualities we were looking for with pure introspection. Specifically, it avoids the three issues we identified with agent based monitoring in section 3. **First**, they can run even if the guest is unresponsive (i.e., ssh is not working or the network connectivity is broken). **Second**, they do not require installation, nor require the guests to have any special library for the data collector code. Both of these problems are solved in containers because by design, attachment operations keep the file descriptors opened before the attachment valid and pointing to the files or sockets in the host context (i.e., namespaces). The consequence of this is that if the data collector loads the necessary libraries, and opens a socket to emit the collected data, it will still have these after the attachment, even if the container file system is corrupted or empty, and the network interfaces are down. And **third**, they can run even if the guest is compromised, as they will get the view of the system from outside of the container. The exception to this is if a container process jailbreaks and gets access to the host context.

Another interesting consequence of using containers is that with VMs, memory and disk are dis-aggregated, we need to take the data from both and later aggregate them. With containers, and specially with the technique proposed, the monitor has a holistic view of the system.

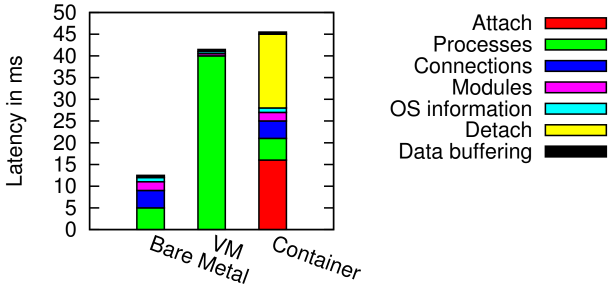

Figure 3: Data collection latency for a bare metal server, a VM and container data collectors.

4 Evaluation

We now evaluate the performance of our cloud monitoring solution, and later show a proof-of-concept application built on top of the analytics framework we described. Our experimental setup consists of a host with 8GB of memory and 8 Intel Xeon @ 2.60GHz cores, running Linux 3.13, QEMU-KVM 1.5, and Docker version 1.0.1 The monitored containers and VMs for the performance evaluation run Ubuntu 14.04. Out of band monitoring of VMs is implemented by getting the memory state of the VM using `/proc/<qemu-process-pid>/mem`, and getting the memory layout from kernel symbols (`system.map`), and a kernel image with debugging enabled from the related `vmlinux` file. We used kafka [7] as the data bus, so all collectors buffer and send data through kafka.

The first set of results, shown in Figure 3, show the latency for a single data collection operation from a fresh Ubuntu installation deployed as VM, container, and bare metal. Each collect operation lists the processes, connections, kernel modules, and basic system information. The first observation is that data collection from containers takes 30 ms more than bare metal, and all the overhead comes from attaching to and detaching from the container. Out of band VM monitoring comes second and spends most of its time getting process information. This is because a single pass of the process structures gets the information about opened connections per process, compared to containers and bare metal where there have to be individual calls (actually, these are reads to `/proc/`). Additionally, there is no equivalent of an attach and detach for monitoring VMs, as the memory is accessed directly.

The next experiment does simple scalability measurements for container monitoring. We run redis-bench and setup the cluster as one master replicating to 99 slaves. Then we measure the data collection time for all the containers for varying number of data collector worker processes (these workers get a subset of all the containers to monitor). Figure 4 shows the results of this experiment when collecting the same features as the previous exper-

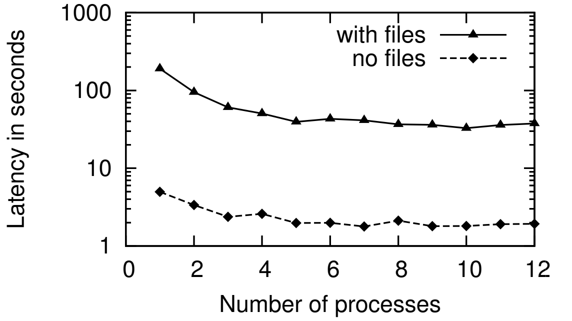

Figure 4: Single data collection time for 100 containers at varying number of data collector worker processes.

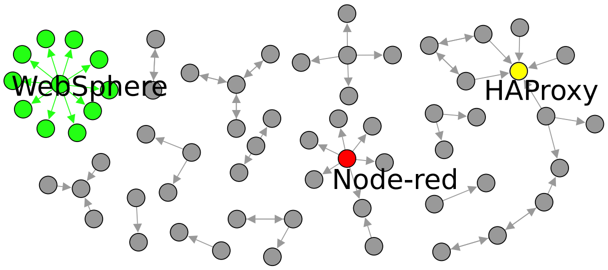

Figure 5: Topology application showing a network graph for containers.

iment (the ”no files” line), and those features plus config files content and information about the files in the tree. How to parse config files and how to identify if a file is a config file is out of the scope of this paper. The expected observation is that more processes help, up to the number of cores. The second observation is that collecting files takes hundreds of seconds. The consequence of this is that the frequency of the monitoring should be specific to each feature, and there is possibly room for several optimization's about collecting file information. Our approach is to naively walk in the tree from user space with several syscalls per file and directory.

Finally, we show results for a proof-of-concept application built using our analytics pipeline. The application is called *topology* and uses Lucene [1] as the index and search service (see Section 2). The monitored cloud was an internal experimental deployment made of 300 containers running on 10 hosts. Figure 5 shows the output of the *topology* application: a graph where nodes are systems, and edges represent established connections at the moment the data collection was made. This is useful as a validation tool, tracking the evolution and interactions of applications, and also for network and placement optimizations, understanding sprawl and the usage patterns of different offered services.

5 Conclusions

In this paper, we present a new, unified monitoring and analytics framework for the cloud. The key observation that drives our work is that the traditional solutions in this space are not a good fit for the dynamism, diversity and density exhibited by applications and instances deployed on the cloud. Existing approaches become increasingly heavyweight and sluggish relative to the highly dynamic and ephemeral nature of cloud instances with their emerging lightweight virtualization and containerization trends. To mitigate these issues, we present a new, out-of-band monitoring approach for VMs and containers, and show how this approach can be leveraged to provide a unified framework across different instance form factors. We implement this framework and deploy it on an actual cloud environment, where we show that our technique is practically feasible, following some realistic assumptions on deployment patterns. We demonstrate the quantitative overheads and trends of our unified approach, and some key potential points of improvement. We further present an end-to-end application built on top of our framework that discovers applications and their topology patterns in a live cloud environment. Overall, our results show the potential and applicability of our unified, out-of-band monitoring and analytics framework, which can be a critical asset for both the tenants and the operators of cloud services, providing deep operational visibility into their environments.

References

[1] Białecki, A., Muir, R., and Ingersoll, G. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval* (2012), pp. 17–24.

[2] CoreOS. http://coreos.com, 2015.

[3] Docker. http://www.docker.com, 2015.

[4] Garfinkel, T., Rosenblum, M., et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS* (2003), vol. 3, pp. 191–206.

[5] Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R. Sok: Introspections on trust and the semantic gap.

[6] Kivity, A., Laor, D., Costa, G., Enberg, P., HarEl, N., Marti, D., and Zolotarov, V. Osvoptimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference* (2014), pp. 61–72.

[7] Kreps, J., Corp, L., Narkhede, N., Rao, J., and Corp, L. Kafka: a distributed messaging system for log processing. netdb11.

[8] Liu, Y., Xia, Y., Guan, H., Zang, B., and Chen, H. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014), IEEE, pp. 416–427.

[9] LWN.net. http://lwn.net/articles/179345/, 2006.

[10] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 461–472.

[11] Parallels. Openvz, 2015.

[12] Suneja, S., Isci, C., de Lara, E., and Bala, V. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2015), VEE '15, ACM, pp. 133–146.