Towards Massive Consolidation in Data Centers with SEaMLESS

Andrea Segalini†, Dino Lopez Pacheco†, Quentin Jacquemart†,
Myriana Rifai, Guillaume Urvoy-Keller†, Marcos Dione
† Université Côte d'Azur, CNRS, I3S, France
dino.lopez@unice.fr

***Abstract*—In Data Centers (DCs), an abundance of virtual machines (VMs) remain *idle* due to network services awaiting for incoming connections, or due to established-and-idling sessions. These VMs lead to *wastage of RAM* – the scarcest resource in DCs – as they lock their allocated memory.**

In this paper, we introduce SEaMLESS, a solution designed to (i) transform fully-fledged idle VMs into lightweight and resourceless virtual network functions (VNFs), then (ii) reduces the allocated memory to those idle VMs. By replacing idle VMs with VNFs, SEaMLESS provides fast VM restoration upon user activity detection, thereby introducing limited impact on the Quality of Experience (QoE). Our results show that SEaMLESS can consolidate hundreds of VMs as VNFs onto one single machine. SEaMLESS is thus able to release the majority of the memory allocated to idle VMs. This freed memory can then be reassigned to new VMs, or lead to massive consolidation, to enable a better utilization of DC resources.

I. Introduction

Data Centers rely on virtual machine (VM) placement algorithms to choose the best server to deploy a given VM, so as to globally optimize the use of available physical resources, e.g. maximize the number of running VMs on a given physical infrastructure. At the same time, some studies show that idle VMs are a common problem in Data Centers. For instance, [1] reports that 30% of VMs deployed within a data center remain in a comatose state, with little to no sign of activity. In public clouds, idle VMs are frequently encountered when tenants instantiate their own DNS or mail servers [2]. In private clouds of software outsourcing companies, VMs are used by software developers to design and test new applications; these VMs exhibit frequent idle periods and are rarely powered off, even outside of office hours or during holiday periods.

Even though they are not actively used, these VMs still lock the physical resources they have been allocated, especially RAM, which is currently the scarcest resource in data centers [3], [4]. Placement algorithms are unfit to manage this kind of behavior and may end up unable to deploy new VMs due to unavailable resources.

Consequently, the abundance of idle VMs introduces a *waste of memory problem*, which derives from the standard data center administration practice to avoid memory overcommitment. Unfortunately, these idle VMs cannot be powered off as they usually host network-based services that are essential to end-users. Reducing the amount of memory allocated to idle VMs is thus necessary. Existing solutions either rely on an application-level proxy server to turn off the VMs and waking them up on demand [2], [5]; or put stringent constraints on the application's design [4]. As a result, they heavily impact the user experience due to extensive delays and fail to propose a generic or easily implementable methodology.

We previously briefly introduced SEaMLESS and its core concept of replacing idle VMs with lightweight and resourceless Virtual Network Functions (VNFs) by leveraging Linux namespaces and process migration [6]. Now, we extend SEaMLESS by providing means of decreasing the memory allocated to idle VMs and stop their execution. We also provide a large evaluation of the SEaMLESS performance and capabilities.

Hereafter, we present the details of SEaMLESS and the procedures to *migrate* a so-called Gateway Process from within an idle VM to an external, lightweight and resourceless Sink Container (Section II). These VNFs provide a feeling of service availability to end-users. We will show that we are able to replace *hundreds* of idle VMs hosted over multiples physical servers by their corresponding VNF on a *single* physical server or VM (Section IV-D). The Sink Container monitors the service for signs of user activity (Section II-C). Upon positive detection, the original VM environment is resumed and the service will continue its execution transparently with minimum impact on the quality of experience (Section IV-B). Because the VNFs are considerably lighter than the full VM environments, we will show how to employ SEaMLESS to reduce the memory footprint of idle VMs at the scale of a data center (Section IV-F), while preserving the quality of experience. This freed memory can then be assigned to new VMs, leading to better utilization of the physical resources in the data center. Or, coupling this methodology with ubiquitous consolidation solutions, such as BtrPlace [7], enabling *massive consolidation in data centers*, and the possibility to group idle VMs on few servers while powering down the empty ones, leading to energy saving and economies on operating costs.

II. Solving the Idle-VM Problem

Virtual Machines (VMs) in data centers are accessible through one or several processes waiting for incoming connections or requests by listening to network ports. We refer to each of these processes as *Gateway Processes*. In this Section, we present how SEaMLESS migrates every Gateway Process from a VM that has been idle for a long-enough period of time to a *Sink Server*.

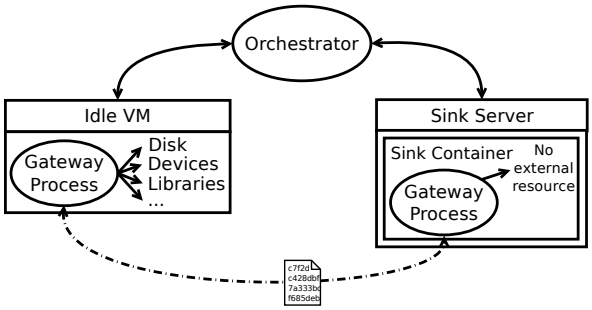

Fig. 1: Components and architecture of SEaMLESS

In more details, using Figure 1 as illustration, a virtual machine hosts a large set of processes, including a Gateway Process. Once the VM has been detected as idle, the SEaMLESS *Orchestrator* – an agent responsible for the synchronization of the migration – transfers the Gateway Process to the Sink Server with all its states and including any open socket, effectively turning it into a Virtual Network Function (VNF). Typically, processes access and reference elements such as files, devices or libraries. These external resources are not copied to the Sink Server to ensure a lightweight environment. When the VNF exhibits signs of user activity, the Gateway Process will be migrated back from the Sink Server and restored in its original VM environment so as to fulfill the user requests.

SEaMLESS is completely transparent from the end user point of view. Indeed, migrating back and forth the Gateway Process between its VM and the Sink Server is faster than migrating the entire VM, and by keeping the Gateway Process running while the VM is paused or turned off, SEaMLESS can maintain any persistent idle connections, either at the transport or at the application layer.

The remainder of this Section is organized as follows. Section II-A focuses on the creation and the structure of the VNF for the Gateway Process. Afterwards, Section II-B describes the migration procedures that are followed by the Orchestrator to guarantee a successful migration. Finally, Section II-C presents the way we detect user activity in the Sink Container, in order to know when to resume the VM and migrate the Gateway Process back.

A. The Gateway Process VNF

SEaMLESS relies on the creation of an ad-hoc Virtual Network Function for the Gateway Process. This VNF must run in an environment that provides isolation, and also be lightweight in terms of memory and CPU consumption. For these reasons, we chose to run Gateway Process VNFs inside Linux namespaces, the underlying element of Linux containers, such as used by Docker [8]. In the remainder of this article, we interchangeably use namespace and container.

The VNF itself is a lightweight version of the Gateway Process. Namely, we *migrate* every Gateway Process from its original environment, inside the virtual machine, to a container located on the Sink Server. As illustrated in Figure 2, when a Gateway Process runs inside its VM, it is part of a *process ecosystem*, that is formed, among other things, by its process ID (PID), file system beacons, file descriptors, libraries, etc. Our goal in SEaMLESS is to *only* migrate the Gateway Process itself, without these side elements. However, this ecosystem is essential for the Gateway Process to continue executing faultless. We now provide some insights into how to achieve this goal.

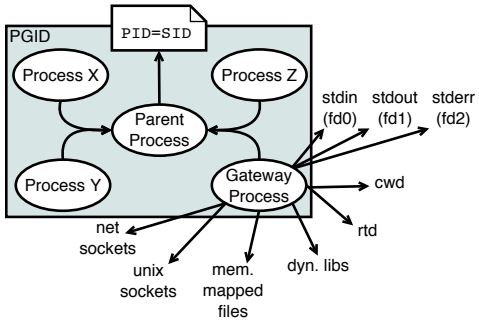

Fig. 2: A possible Gateway Process ecosystem and its resources

First, a process is identified by its unique PID. Its value must be preserved to guarantee successful migrations, which we achieve by relying on both the mount and the PID namespace in our Sink Container to isolate the `proc` and `sys` filesystems. Consequently, when multiple processes from multiple VMs are migrated to the Sink Server, it is possible for these gateway processes to use the same PID.

Second, processes rely on external, dynamically linked shared libraries, which are loaded, unloaded, and linked to a process during runtime. If these libraries are unavailable, the program might experience segmentation faults. Since VMs deployed in IaaS environments are created from a template image, SEaMLESS can use this very same image to instantiate the Sink Server, thereby guaranteeing the availability of these libraries. In particular, it prevents the transfer of the libraries between the VM and the Sink Server.

Third, a process possesses many file descriptors to standard outputs and standard input devices (e.g. `stdout`, `stderr` and `stdin`), but also to standard files, directories, memory-mapped files, and various devices. To handle external resources eventually needed by the Gateway Process, we decided to point them out to empty, temporary files in the Sink Container created on the fly. We leverage the `mount namespace` to ensure consistency between the Sink Container and the original VM by recreating the required folder trees. Since the Gateway Process is expected to access the external resources only in case of user activity, which will trigger the migration of the Gateway Process to the VM (discussed in Section II-C), the empty files at the Sink Container will not introduce any problem during the Gateway Process's runtime. In the case where the Gateway Process needs to access external resources in absence of user activity (e.g. for generating log entries),

SEaMLESS can be configured to properly export the data back to these VMs or to ignore them altogether.

Fourth, a process usually depends on Unix sockets or network-based sockets for external communications. SEaMLESS shall be able to maintain existing connections (e.g. established SSH sessions) even after the VM has been stopped. Users shall remain connected to the Gateway Process VNF after the process migration, without connection teardown or data loss. To achieve this, and also avoid customized kernels with obscure patches and exclusive hardware devices, we rely on CRIU (Checkpoint/Restore In Userspace) [9], a modern actively developed software to enable the migration of a processes. CRIU's hooks are now part of the official Linux kernel code. CRIU is able to *checkpoint* or *dump* a given process to later restore it while preserving all sockets and pipes. As a result, with CRIU, SEaMLESS is able to write to disk a Gateway Process, and later restore this Gateway Process along with all its established network connections (e.g. keeping alive already established SSH session), either at the Sink Container or the VM.

In summary, we *migrate* a Gateway Process inside a VM to a Sink Container on a Sink Server. The Sink Container is composed out of Linux namespaces: a `PID namespace`, a `mount namespace` and a `network namespace` populated with two virtual ethernet devices (veths), supported by default in current Linux distributions. One veth is connected to the public LAN, where the data packets from the users are expected to pass and must be configured with the IP address of the VM where the Gateway Process is coming from. The second veth is connected to the management LAN, used to exchange the Gateway Processes images between the Sink Containers and VMs.

B. Migration Procedures

In this Section, we present the procedures to orchestrate the migrations of a Gateway Process between the Sink Container and its VM, and vice versa. Note that the Gateway Process migration must be perfectly coordinated with network rerouting strategies to avoid data packet losses.

1) Migrating from the VM to the Sink Server: When a virtual machine is known and expected to be idle for a long period of time, the Orchestrator will trigger the Gateway Process migration to the Sink Server.

The complete procedure for a process migration from the VM to the Sink Server must follow a set of steps enumerated below. Note that when a single step includes several actions, these actions should be parallelized.

Step #1: The Orchestrator asks the working VM to dump every Gateway Process to a file. It keeps the processes running, which allows the VM to detect any user activity. Should any user activity be detected, the VM sends an abort message to the Orchestrator to cancel the migration, without any message loss. **Step #2:** Once the process image has been generated, the working VM sends the process image file to the Sink Server. **Step #3:** The Sink Server deploys the Gateway Process VNF. **Step #4:** The networking devices are modified to forward packets to the Sink Container. At the same time, the Orchestrator asks the working VM to kill the Gateway Process. **Step #5:** The Orchestrator reduces the memory footprint of the VM (explained in Section III).

Note that the detection of any user activity during the Sink Container deployment will result in a roll-back mechanism that is completely transparent from the user perspective.

2) Migrating from the Sink Server to the VM: The procedure to move a Gateway Process back from the Sink Server to the VM is close to the reverse procedure described earlier, except that when a process is moved from the Sink Server to the VM, the user has already issued a request and user's packets can flow at any time.

Step #1: Upon detection of user activity in the VNF, the Sink Server notifies the Orchestrator that the corresponding VM must be woken up. **Step #2**: Using the Netfilter Queue (NFQUEUE) available in current Linux distributions, a buffer is deployed in the physical server hosting the VM in order to store packets sent with the VM as destination. **Step #3**: The Orchestrator resumes the VM and modifies the routing table to forward future packets to the VM instead of the Sink Container. While these operations take place, the Gateway Process at the Sink Server is dumped. **Step #4**: Once the VM is on, the process image is sent from the Sink Server to the VM. **Step #5**: The process is deployed at the VM, **Step #6**: Eventual packets are unbuffered from the NFQUEUE buffer, the filter is destroyed, and the communication is now handled by the VM.

3) Addressing Routing Issues: To properly reroute packets to the Gateway Process VNF or to the VM, SEaMLESS entirely relies on Software Defined Network (SDN) forwarding devices and/or Virtual eXtensible Local Area Network (VxLAN) [10]. Hence, the Orchestrator asks the SDN controller to inject the ad-hoc rules into the SDN switches during the migration procedures or the Orchestrator properly updates the required VxLAN tunnels.

Note that SDN (through OpenvSwitches devices – OvS – [11]) and VxLAN technologies are already deployed in current Data Centers when enabling VM migrations and Networking as a Service (NaaS) to provide Infrastructure as a Service (IaaS). This is the case for instance of OpenStack configurations with Neutron [12].

C. Detecting User Activity

Once the Sink Container is deployed, it is the Sink Container that will first handle any communication with the remote user. Two questions arise at this point: (i) how to detect user activity; and (ii) how to prevent the Gateway Process VNF from taking over the entire communication with the end user.

First, we highlight that not all incoming network packets are a prelude to user activities, such as ICMP, ARP, or application-level keep-alive messages. They consequently must not trigger the process of VM awaking. Typically, messages at the transport layer or below are directly replied to by the protocol stack available at the Sink Container. However, application-layer keep-alive messages require the execution of code at

the Gateway Process, but do not need user data. SEaMLESS implements an accurate solution to successfully handle keep-alive applications messages without the need for restoring the VM.

To understand the way SEaMLESS detects real user activity, we need to understand first the behaviors that a Gateway Process may have when it receives a new user's message. If the Gateway Process employs TCP, a user message might request a new connexion setup with the server, verify the existence of an application-level channel (e.g. SSH keep-alive messages) or ask for external data (not available at the Sink Container). If the Gateway Process employs UDP, a user message might carry a membership verification/update at the application-level or ask for external data (again, not available at the Sink Container).

After analyzing several Gateway Processes applications, we have observed that a request for a new TCP connection raises an `accept()` system call (usually referred to as a *syscall* simply), which is applied through the TCP socket's file descriptor.

Keep-alive application messages and membership verification, or membership update, raise a `read()` syscall followed by a `write()` syscall, both on the file descriptor of the network (TCP or UDP) socket. This means that the message of the user has been processed by the Gateway Process code without needing external data.

User's messages requesting external data raise a `read()` syscall on the file descriptor of the networking socket, followed by a `write()` or a `read()` on a different file descriptor, which in our case will point to a dummy zero-byte file.

Consequently, to accurately detect user activity needing data not available at the Sink Container, SEaMLESS relies on *syscalls tracking* over the Gateway Process inside the Sink Container. In particular, we track the `read()`, `write()` and equivalent system calls made through file descriptors pointing to dummy files or newly created file descriptors, as well as `accept()` syscalls wherever it occurs.

Syscall tracing can be easily done in Linux using the `ptrace` library, whose required hooks are available by default in most Linux kernel versions. Technically, with the `ptrace` library, for each (specified) syscall, the kernel will trap the Gateway Process (via a `SIGTRAP` signal), and notify SEaMLESS of the syscall. Namely, the Gateway Process is stopped at every trap. After we have collected the file descriptor numbers, we need to restart the execution of the Gateway Process by means of the restart (`SIGCONT`) signal, if the SEaMLESS heuristic decides that the Sink Container can process the message by itself. Otherwise, SEaMLESS stops the Gateway Process execution completely (with the `SIGSTOP` signal) and starts the migration procedure of the Gateway Process from the Sink Container to the VM. Hence, it is the VM and not the Sink Container that will reply to the user's request.

Figure 3 graphically illustrates the case where a message is received through the TCP socket with file descriptor ID 4, which triggers a write through a different file descriptor identifier, hence making SEaMLESS stopping the execution of the Gateway Process, to move back later such a process to the VM.

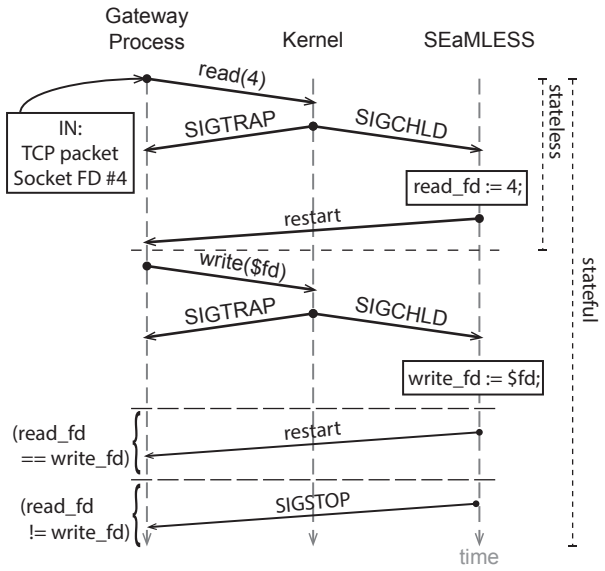

Fig. 3: Detecting user activity exhibited by the Gateway Process VNF

III. Solving the Waste of Memory Problem

SEaMLESS substitutes a lightweight resourceless VNF inside a Sink Container in place of an idle VM. Consequently, while the Gateway Process VNF awaits for incoming user requests, the idle VM resources can be freed. However, stopped VMs should keep all user states, in order to provide an unchanged environment when the Gateway Process needs to be restored inside the original VM.

Most hypervisors support two modes of stopping VMs which keep the VM states unchanged: *suspend to RAM* and *suspend to Disk*. When suspending to RAM, a VM and its processes are paused and remain in the memory of the physical server. Hence, no memory is actually released; but restoring the VM only takes a few milliseconds. When suspending to disk, the VM and its processes are written to disk. Consequently, the delay when restoring a suspended to disk is longer than the equivalent delay for a suspend to RAM. The measured time required to reload a VM suspended to disk depends on the virtualization technology. For instance, while in QEMU/KVM the entire VM memory needs to be reloaded into the RAM memory before the VM is able to execute any instruction, VMWare ESXi adopts an approach called *lazy restore*, where the memory is only partially loaded into the RAM before the VM starts executing any instruction [13]. If the instructions executed by the VM are not yet loaded into memory, the read fault is caught by the hypervisor and the missing region loaded as soon as possible. The instruction is then finally executed.

Inspired by this lazy restore approach of VMWare, we devised a solution we call *suspend to swap*, able to deallocate memory from an idle VM *while* providing fast restoration *and* fast availability of services. We currently tested suspend to swap with the QEMU/KVM hypervisor. Suspend to swap meets the objectives of SEaMLESS: avoid any modification in the hypervisor, avoid any kernel-level patching, and leveraging only currently available Linux features.

Our suspend to swap relies on *ballooning* [14], available in all major virtualization solutions, and on Linux `cgroups` (control groups). It works as follows. (i) To deallocate memory from an idle VM, we first inflate the *balloon* to recover the free memory of the guest VM on the physical host. Note that inflating the balloon beyond the available free memory can lead to a high memory pressure on the guest VM, possibly triggering out-of-memory errors and the associated procedure which kills some processes to mitigate the memory pressure, potentially introducing severe modifications in the user environment. (ii) Using cgroups on the host server, we limit the maximum memory usage of the guest VM to 10 MB. This triggers a *hypervisor swapping* procedure, where the hypervisor writes to (the physical) swap the memory pages of the VM to meet the host memory constraint. After this step, the memory footprint of the idle VM is equal to 10 MB, effectively releasing an important proportion of the memory used by the idle VM. (iii) We remove the cgroup memory constraint on the VM and we perform a *dummy Gateway Process restoration*. The goal of this dummy restore is to only load in the VM's RAM most of the shared libraries needed by the Gateway Process, as well as the memory regions needed by the system for a correct execution, which have been swapped-in due to the cgroup constraint. This step enables a fast service restoration of the services when the idle VM needs to be active again. (iv) At the end of the previous step, a proportion of the memory employed to execute the dummy restoration will be unused again. Therefore, we once more inflate the balloon to recover this freed memory for the host. (v) Finally, the virtual machine is now paused, relying on suspend to RAM. The execution of the above steps will provide a memory footprint of the idle VM smaller than 600 MB, as shown in Section IV-F.

Note that pure hypervisor swapping with cgroups is not a good option. Indeed, with pure hypervisor swapping both the dirty and free memory of an idle VM might land to swap. Hence, when SEaMLESS will restore the Gateway Process, the VM can employ the free memory available in the swap, therefore leading to a very slow Gateway Process restoration.

Later, when a VM needs to be restored by SEaMLESS, it is resumed and the balloon is completely deflated. These operations are very fast operations. Hence, our *suspend to swap* strategy provides fast virtual machine restoration, including the activation of the Gateway Process, as shown in the evaluation of our wake up delay in Section IV.

IV. Performance Evaluation

In this Section, we evaluate the performance of SEaMLESS wrt. the perceived end-user Quality of Experience and the resulting memory savings. In Section IV-B we evaluate the delay due to the main SEaMLESS components when a Gateway Process must be moved from the Sink Container to the VM. We assess the impact of our suspend to swap strategy in Section IV-C. The scalability of SEaMLESS is analyzed in Section IV-D, its reactiveness in Section IV-E, and finally, we provide insights about the amount of released memory with SEaMLESS in Section IV-F.

A. Network Testbed

We tested our SEaMLESS prototype on one of the clusters of the Grid5000 network [15], a large-scale testbed for research experiments on distributed systems. We used Dell PowerEdge R430 servers equipped with 2 CPU Intel Xeon E5-2620, 32 GB of memory, 10 Gbps Ethernet NICs.

The testbed consists of three physical hosts on the same cluster. The first node hosts the SEaMLESS Orchestrator, the second acts as the Sink Server, and the third node hosts the user VM. The SEaMLESS management network (used to transfer Gateway Processes images, and to orchestrate the SEaMLESS events) and the public network (where users' packets flow), were setup using VXLAN tunnels.

Rerouting is executed by the Orchestrator by means of OpenFlow (SDN) rules installed in OvS switches.

B. Impact on the Quality of Experience

A full restoration process involves the execution of the following phases, each contributing to the unavailability period of the service: (i) Gateway Process dumping; (ii) image compression; (iii) image transfer; (iv) image decompression; (v) processes restoration; and (vi) synchronization time between the Sink Container, the Orchestrator and the VM to dump, transfer and restore a Gateway Process.

To evaluate the time needed by SEaMLESS to restore a Gateway Process at the VM, we carried out several tests with different ubiquitous Gateway Process applications on their default configuration and evaluated the delay of every task using the system clock. The results are available in Table I, where we report the delays due to dumping, compressing, transferring, decompressing, and restoring a Gateway Process. The column labeled "Total" corresponds to the sum of all previous delays. The response time delay (labeled "Resp. Delay") corresponds to the observed delay between the first packet sent by the client to the VM, and the first packet sent back from the restored machine. Hence, the response delay comprises the components from the "Total" column, plus the time needed to synchronize the SEaMLESS events (e.g. the signaling of user activity from the Sink Container to the Orchestrator). The columns "Image Size" and "VNF size" correspond to the compressed image size of a Gateway Process (in a `tar.lzo` file) and the size of the Sink Container hosting such a Gateway Process respectively. Images are securely transferred with the `scp` command, as it would be done in a real data center.

From our results in Table I, we see that the application with the largest image file is Tomcat (1.172 MB), followed by Apache 2 with PHP enabled (0.428 MB). The lightest image corresponds to the vsftpd application (an FTP/SFTP server) with only 0.107 MB.

We would like to point out that the PHP application used with Apache 2 does not impact the Sink Container size, nor the Gateway Process image size. Indeed, the Gateway Process image only includes the main PHP engine libraries, and not the PHP applications themselves, which are loaded as external resources when needed.

The number of SSH worker processes for both Dropbear and OpenSSH (and, therefore, the size of the process image) depends on the number of established SSH sessions. In our tests, we maintained one single SSH connection for both Dropbear and OpenSSH, leading to two processes to be dumped and restored (i.e. the main daemon process, plus the worker process).

From our tests in Table I, we see that the response delay of SEaMLESS is generally smaller than 1 second (except for the case of Tomcat). This is lower than the response time typically expected by the end users, as reported in [16]. Therefore, the response delay introduced by SEaMLESS only has little impact on the end-user quality of experience.

C. Impact of Suspend to Swap

We discussed several VM suspend strategies in Section III. Namely, suspend to RAM, suspend to disk, and suspend to swap, which we introduced. In Figure 4, we quantify the delay that the user will experiment, with the Apache 2-PHP application, and the OpenSSH application, under those different strategies.

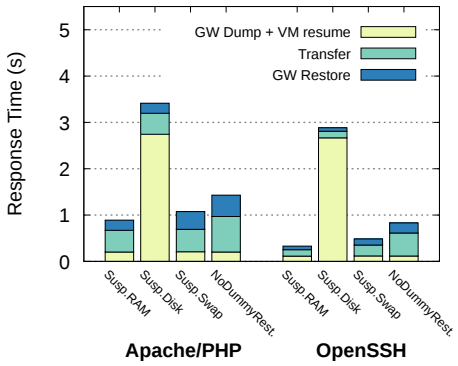

Fig. 4: Response time when using different disabling techniques on a 3 GB VM.

From Figure 4, we can observe that the suspend to RAM strategy is the fastest solution to restore a Gateway Process. Indeed, the response time is around 1 second for Apache 2-PHP and lower than 0.5 seconds for OpenSSH. However, as we stated in Section III, suspend to RAM is unable to release memory resources. Suspend to disk is the worst solution: the time linearly increases with the memory size of the VM and, for the 3GB instance used in our experiment, it leads to a response time of around 3 seconds with both Apache 2-PHP and OpenSSH. Our suspend to swap strategy features a faster restoration than the suspend to disk case method. Suspend to swap, without pre-swapping out the Gateway Process library with the dummy restoration, leads to a response time slightly lower than 1.5 second for Apache 2-PHP, and 0.9 seconds for OpenSSH, while effectively releasing memory allocated to an idle VM (details will be discussed in Section IV-F). Moreover, the complete suspend to swap solution (i.e. adding the dummy Gateway Process restoration) significantly improves the performance of SEaMLESS, leading to a response delay slightly higher than 1 and 0.5 seconds for Apache 2-PHP and OpenSSH respectively.

D. Scalability of the Sink Server

In this Section, we focus on the the scalability of the Sink Server as a function of the number of Sink Containers. For doing so, we used as a Sink Server a VM with 1 vCPU and 1 GB of RAM. Figures 5 and 6 illustrate the memory and CPU consumption of the Sink Server VM when deploying solely Apache 2-PHP VNFs or OpenSSH VNFs.

As expected, both the memory and CPU usage increase linearly with the number of deployed Sink Container. Also, we notice that the number of VNFs that can be installed is limited by the RAM capacity: deploying more than 10 Apache 2-PHP VNFs introduces memory swapping as the memory usage is close to 100%. The swapping phenomenon corresponds to the plateau in the memory curve. However, the CPU utilization remains lower than 1%. If we only deploy OpenSSH VNFs, which has a smaller memory footprint than the Apache 2-PHP VNF, we can reach up to 43 OpenSSH VNFs before saturating the 1GB memory, while the CPU consumption remains lower than 6%.

These results show that a typical data center server configured with 32GB of RAM can host around 320 Apache 2-PHP Sink Containers or 1376 OpenSSH Sink Containers, before experiencing memory swapping. These figures are much higher than the number of idle VMs that could possibly be running simultaneously on the same server with only 32GB of RAM.

E. Reactiveness

To assess the reactiveness of SEaMLESS, we have performed stress tests where we either have several OpenSSH Sink Containers or several Apache 2-PHP Sink Containers in a single VM with 5 GB of memory and 1 vCPU. We then sent one request to each deployed Sink Container, at once, to emulate a burst of user activity. Each test case was executed 20 times.

Figure 7 shows the response time (which does not include the wake up delay of the VM), for the cases of 1, 10, and up to 50 Sink Containers with 10 Sink Containers step increases. We can observe that the response time increases almost linearly. For one Apache 2-PHP Sink Container it is around 0.9 second;

Application	Main Tasks Time (s)					Total (s)	**Resp. Time** (s)	Image Size (MB)	VNF Size (MB)
	Dump	Compr.	Transfer	Decompr.	Restore				
Dropbear	0.026	0.018	0.115	0.02	0.02	0.199	0.406	0.115	11.18
Vsftpd	0.108	0.016	0.107	0.02	0.032	0.283	0.412	0.107	7.81
OpenSSH	0.102	0.024	0.133	0.028	0.038	0.325	0.456	0.133	15.93
Lighttpd/PHP	0.082	0.079	0.287	0.093	0.063	0.605	0.706	0.287	46.43
Apache2/PHP	0.112	0.118	0.428	0.151	0.089	0.898	0.948	0.428	67.52
Tomcat	0.185	0.275	1.172	0.282	0.1	2.015	2.158	1.172	206.96

TABLE I: Response Time of real-world Gateway Process applications.

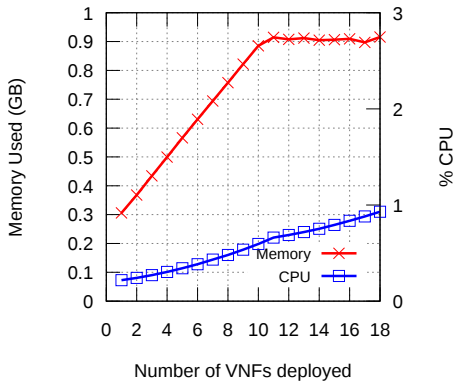

Fig. 5: RAM and CPU used as a function of the number of deployed Apache 2 with PHP module VNF.

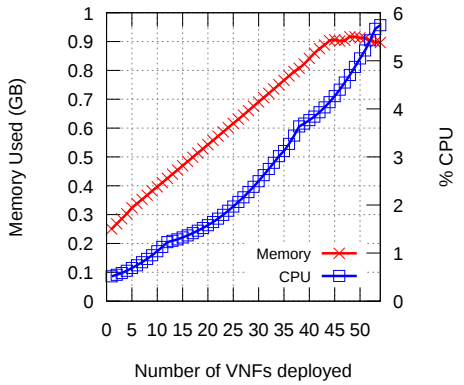

Fig. 6: RAM and CPU used as a function of the number of deployed OpenSSH with 1 connection VNF.

8 seconds for 10 Sink Containers; and 13 seconds for 20 Sink Containers. For OpenSSH Sink Containers, one Sink Container needs a little less than 0.5 second; around 3 seconds for 10 Sink Containers; and around 6 seconds for 20 Sink Containers. Note that the observed response delays exhibit little variation, with very narrow interquartile ranges.

We have found that the main contributor to the response time is the access to the disk. Since a single disk is used to dump and transfer the Gateway Process image at the Sink Server VM, and at the physical server hosting the VMs a single disk is used to read the Gateway Process image to be restored, all those disk operations are sequentially executed.

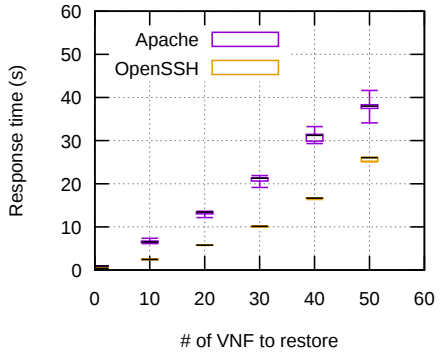

Fig. 7: Response time as a function of the number of parallel OpenSSH VNFs with user activity.

F. Memory Savings

The main objective of SEaMLESS is to release the physical memory locked by idle VMs. To determine the amount of RAM that can be freed, we need to estimate the expected memory consumption of a memory-reduced VM and the expected memory consumption of its respective Sink Container. Indeed, the difference of memory consumption (M_{diff}) provides insights about the amount of memory to be released. This value is expressed as follows: $M_{\text{diff}} = M_{\text{VM}} - M_{\text{VNF}} - M_{\text{sVM}}$, where M_{VM} is the size of the VM memory, M_{VNF} the size of Sink Container, and M_{sVM} the size of the reduced VM.

The amount of RAM needed by a VM depends on its RAM size. For instance, using KVM/QEMU, a VM configured with 1 GB of RAM using 80% of its virtual memory will need around 3.25 GB at the physical host, and a 7 GB VM using 80% of its virtual memory will need around 13 GB of the physical RAM. The amount of extra memory needed by the physical host to deploy a VM depends on both the hypervisor and the virtual RAM size.

However, with our suspend to swap strategy, the size of a reduced VM shows only small variations. Hence, a 1.7 GB VM will be reduced to around 500 MB while a 35 GB VM will be reduced to around 620 MB, according to our tests using the Apache 2-PHP Gateway Process. Regarding the Sink Container, its size depends on the Gateway Process application to be deployed. For instance, the Apache 2-PHP Sink Container needs around 68 MB, while the OpenSSH Sink Container only needs around 16 MB.

To exemplify the memory gains that we can obtain by deploying SEaMLESS in a data center, we consider a scenario

derived from observations made for Amazon Web Services (AWS) in terms of distribution of VM size and number of VMs created every day, which amounts to around 50K [17], [18]. Assuming 30% of idle VMs [1], and that all VMs deploy an Apache 2-PHP instance, we estimate, in Table II, the amount of memory that can be released for this scenario. Note that the reported memory savings have been estimated assuming that the amount of physical RAM needed by a VM is equal to the virtual RAM size of the instance, and therefore, ignoring the memory overhead at the hypervisor to keep the VM states. With these hypotheses, for a total amount of 282 TB RAM allocated to host the VMs, SEaMLESS can save an impressive 76 TB of RAM every day.

V. Discussion

There are scenarios in which SEaMLESS is unable to operate due to technical limitations in the underlying technologies (CRIU and `ptrace`). Currently, CRIU does not support the reconnection of Unix stream sockets. Indeed, checkpointing and killing a process cause the Unix stream socket peer to close the connection. Hence, it is not possible to restore a process with a Unix stream socket, as this latter will not retrieve the expected system environment.

In II-A we stated that instantiating the sink server from the same template of the idle VM avoids transferring libraries loaded by the Gateway Process. We are aware that any alteration from the original template may result in failures in the restoration. However, different versions are easy to detect and prevent the migration in this case. Additionally, SEaMLESS is meant to be used in private clouds where VM's templates are finely prepared for specific applications thus modifications are not contemplated.

Since SEaMLESS provides two complementary solutions – one to create a lightweight Sink Container with the Gateway Process, and another one to deallocate memory from idle VM – one might ask why the idle VM is not just shrunk, but leaving the Gateway Process at the VM, without pausing it. Detaching the Gateway Process (that must always run), from the VM (which can be paused), brings several benefits. First, we have observed that the basic memory set of an idle VM hosting a Gateway Process slowly grows over time (also observed in [19]), and could even show sudden memory increases due to some services such as applications updates. Hence if cgroups are employed to avoid drastic memory fluctuations of that VM, future memory needs at the VM will lead to swap in and out events, inducing system instability at the physical host. Second, multiple server reconsolidations in the data center can be easily executed when a VM is suspended, which is achieved in SEaMLESS thanks to the creation of the Sink Container. Third, idle VMs need more processing capacity at the physical server than Sink Containers with paused VMs. Indeed, the hypervisor needs to monitor the VM events even if the VM is idle, which is not the case when the VM is paused.

Finally, an important point is the detection of idle VMs. While defining metrics that allow the identification of idle VMs is out of the scope of this article, one solution is to rely on user-based metrics, such as system load, transferred data on the network, and so on. These metrics are already available to cloud providers and are frequently used for billing. A number of possible smart solutions are also proposed in [20].

VI. Related Work

Server resources optimization in data center networks can be achieved using server consolidation [21], [22], [23]. Multiple strategies can be employed to achieve server consolidation or reconsolidation, such as dynamic relocation of VMs [24], live migration [25], [26], and scheduling and migration algorithms for cloud data centers [27]. SEaMLESS goes beyond server (re-)consolidation by reducing the memory footprint of idle VMs and stopping them. Hence, active VMs can be reconsolidated in the smallest set of physical servers. The released resources can then be reused or the unused servers can enter low power modes. In the past, some propositions have already been made to tackle the problem of idle VM or desktops [28], [29], [5], [2], [30]. We briefly discuss those proposals below and highlight the differences w.r.t. SEaMLESS.

In [28], the authors propose to save energy from idle desktop machines without any service disruption by live migrating the user's desktop environment from the personal computer to a server VM in a remote data center. This strategy indeed saves energy at the personal computer. However, on the data center side, energy is wasted to keep idle VMs on, which is the problem tackled by SEaMLESS.

The closest approaches to SEaMLESS are [29], [5], [19], [2], [30]. In [29] the authors employed a sleep-proxy per subnet, to wake up a client machine whenever a TCP connection is requested. It then forwards the TCP connection demand to the client and also replies to ARP requests on behalf of the sleeping client. Since this solution is only able to recognize TCP connexion requests (by looking at the SYN flag of the TCP header), already established TCP connections or UDP-based applications cannot be successfully handled by the sleep-proxies. SEaMLESS, relying on the Gateway Process of the idle VM and a simple heuristic monitoring some system calls (see Section II-C), is able to work with TCP and UDP, and even encrypted channels, such as the one created by SSH. Last but not least, in [29], obtaining a response from the idle client requires at least 8.5s, which means that the TCP SYN packet will be transmitted several times by the machine initiating the connection, as TCP will experience timeouts. In SEaMLESS, TCP SYN and data packets are immediately acknowledged by the Sink Container, and buffered by CRIU, avoiding TCP timeouts and data losses.

In [30] a solution to tackle the problem of idle web applications is proposed. The authors introduce a system, called DreamServer, where a suspension-aware proxy intercepts the HTTP requests and awakes the corresponding virtual appliance hosting the server. Fast virtual machine resuming is achieved using lazy restoration, a concept comparable to the suspend to swap employed by SEaMLESS. However, DreamServer only works for HTTP and requires to write an application-specific proxy for each other protocol. SEaMLESS, by design,

AWS Instances	Size (GB)	%	# VMs	# Idle VMs	Total Mem. (GB)	Total Idle Mem. (GB)	Reduced VM (GB)	Memory Saving (GB)
t1.micro	0.61	11.8	5882	1765	3605.67	1081.95	0.501	81.52
c1.medium	1.70	9.4	4706	1412	8000.20	2400.40	0.474	1638.51
m1.small	1.70	22.4	11176	3353	18999.20	5700.10	0.474	3890.89
c3.large	3.75	3.5	1765	530	6618.75	1987.50	0.496	1690.15
m1.medium	3.75	10.6	5294	1588	19852.50	5955.00	0.496	5064.07
m3.medium	3.75	2.4	1176	353	4410.00	1323.75	0.496	1125.70
c1.xlarge	7.00	9.4	4706	1412	32942.00	9884.00	0.515	9063.74
m1.large	7.50	17.6	8824	2647	66180.00	19852.50	0.511	18326.90
m1.xlarge	15.00	8.2	4118	1235	61770.00	18525.00	0.527	17793.61
m2.xlarge	17.10	2.4	1176	353	20109.60	6036.30	0.560	5815.69
m2.2xlarge	34.20	2.4	1176	353	40219.20	12072.60	0.598	11838.33
Total			50000	15001	282707.12	84819.10		76329.12

TABLE II: Memory saving in a 50 000 VMs data center, each deploying a 67MB VNF.

supports every kind of application including protocols where long-running connections are established.

In [2], the authors propose Picocenter to tackle the problem of idle VMs. Picocenter introduces a proxy-based architecture, where a *hub* orchestrates the entire cloud, deploying applications on the Data Center and updating the DNS databases. Afterwards, users can interact directly with the virtual instances, without passing through the hub. In case the virtual instance is idle, Picocenter uses a modified CRIU version to partially dump the deployed application to a disk (local or remote) and leaving untouched the memory pages used by the idle application. If a memory page is needed, e.g. in case of user activity or because Picocenter failed to detect all the memory pages used by an idle application, the pages must be retrieved from disk and loaded to RAM.

Although Picocenter is close to SEaMLESS, there are a number of key differences. First, Picocenter does not work with Virtual Machines, but with containers only. In contrast, one of the requirement in SEaMLESS was to maintain the presence of legacy VMs, which is often the virtual machine unit in Data Centers, mostly for security reasons. Second, the Picocenter hub represents a single point of failure. Hence, any bug or problem at the Picocenter hub would introduce a network outage. SEaMLESS avoids the introduction of a complex single point failure, as the Sink Container can be placed everywhere. SEaMLESS does not need to modify the cloud manager and the failure of any SEaMLESS agent, in the worst case, will not introduce more problems than a crashing VM in the cloud would introduce.

Regarding our suspend to swap strategy to deallocate memory from an idle VM, some similar strategies can be found in the literature. For instance, in [19] the authors propose Oasis, which uses VM partial migration to transfer the active memory only of an idle VM to a consolidation server. The remaining memory of that idle VM is left on the RAM of the original physical host. Hence, unavailable memory pages are recovered from the original host on an on-demand basis. Note that the pages retrieval introduced by Oasis will penalize the QoE while the set of pages to provide a service is not fully recovered. This is not the case of SEaMLESS where the entire memory to process a user request is already available at the Sink Container and at the VM once the latter is restored.

In [31], the basic memory to properly run a VM is estimated to provide fast VM migration. To do so, the VM memory size is constrained with cgroups, then swapped-in pages are marked as *hot pages* and will build the minimum working set memory of a VM. In this solution, the cgroups constraints are conservatively tested, with probings periods of up to 30 seconds. SEaMLESS, however, provides a fast estimation of the *hot pages* by enabling a *dummy restore Gateway Process* procedure. [31] will also suffer from slow service restoration if the required service hits several cold pages, which is not the case of SEaMLESS where the idle VM and its original memory environment is restored in case of user activity.

Migrating or relocating VMs requires the migration of all the connections and processes that are using this VM. Several solutions have been proposed to migrate connections. For example, Snoeren et al. [32], introduce some TCP options in order to enable migration capabilities. This solution requires support on both, the client and the server sides. Another solution is SockMi [33], which requires copying all the TCP/IP stack states and data from one socket to another by using the SockMid daemon implying, therefore, modifications on user gateway processes. In contrast, SEaMLESS requires no modification of the guest application.

SEaMLESS is orthogonal to solutions aiming to minimize the virtualization's overhead, for instance by integrating the application (e.g., a web server) inside Unikernels [34] or directly into containers. In this cases, no action is taken against the idle process ecosystem surrounding the application while in SEaMLESS only the communicating part, namely, the Gateway Process is spared while the rest is removed from the memory.

VII. Conclusion

One of the most popular solutions to optimize resource usage in data centers is *server consolidation*, which aims at minimizing the number of powered on physical servers to run the required number of virtual servers. This approach is rendered useless when a large portion of these virtual instances are idle, leading to a situation where RAM is wasted to maintain virtual kernels afloat, without doing any useful work.

For this reason, we proposed SEaMLESS, a service able to (i) convert a *Gateway Process* inside an idle VM into a Virtual Network Function (VNF) running on a *Sink Server*,

(ii) deallocate memory from idle VMs, and (iii) quickly restore the VM when needed. Because the VNFs are lightweight and resourceless, one single Sink Server can host hundreds of Gateway Process VNFs, as opposed to only a few tens of fully-fledged VMs on the same equipment.

At the same time, the Gateway Process VNFs provide a sense of always-on availability. When SEaMLESS detects user activity on one of the VNFs, the corresponding virtual machine is resumed, and takes over the communication with the user, leading to a *seamless* experience from the end user perspective.

Our experiments demonstrate that the SEaMLESS impact on the Quality of Experience is limited. Indeed, SEaMLESS features a response delay of around 1 second for Apache 2 with PHP, and around 0.5 seconds for OpenSSH; both values including the VM resuming delay and any restoring service procedure. More importantly, SEaMLESS offers important memory savings, enabling massive server reconsolidation.

Finally, because of our architectural choices, SEaMLESS can be integrated into standard Data Center managers, such as OpenStack. This integration is the next step on our agenda, along with the study of its interplay with the BtrPlace scheduler.

VIII. Acknowledgments

The work done by Quentin Jacquemart has been funded by the French National Research Agency (ANR), through the program "Investments for the Future", reference #ANR-11-LABX-0031-01.

The research leading to these results has received funding from the European Commission's Horizon 2020 Framework Programme for Research and Innovation (H2020), under grant agreement #732339: PrEstoCloud.

References

[1] J. Koomey and J. Taylor, "Zombie/comatose servers redux," 2017, visited on 2017-11-07. [Online]. Available: http://anthesisgroup.com/wp-content/uploads/2017/03/Comatsoe-Servers-Redux-2017.pdf

[2] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove, "Picocenter: Supporting long-lived, mostly-idle applications in cloud environments," in *EuroSys 2016*. ACM.

[3] G. Moltó, M. Caballer, and C. de Alfonso, "Automatic memory-based vertical elasticity and oversubscription on cloud platforms," *Future Gener. Comput. Syst.*, vol. 56, no. C, Mar. 2016.

[4] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *EuroSys 2013*. ACM.

[5] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan, "Jettison: Efficient idle desktop consolidation with partial vm migration," in *EuroSys 2012*. ACM, 2012, pp. 211–224.

[6] D. L. Pacheco, Q. Jacquemart, A. Segalini, M. Rifai, M. Dione, and G. Urvoy-Keller, "Seamless: A service migration cloud architecture for energy saving and memory releasing capabilities," in *ACM SoCC 2017*. ACM.

[7] F. Hermenier, J. L. Lawall, and G. Muller, "Btrplace: A flexible consolidation manager for highly available applications," *IEEE Trans. Dependable Sec. Comput.*, vol. 10, no. 5, pp. 273–286, 2013.

[8] Docker, "Build, ship, and run any app, anywhere," https://www.docker.com/.

[9] CRIU, "Checkpoint/Restore In Userspace," https://criu.org/Main_Page.

[10] M. Mahalingam, T. Sridhar, M. Bursell, L. Kreeger, C. Wright, K. Duda, P. Agarwal, and D. Dutt, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," RFC 7348, Aug. 2014.

[11] O. vSwitch, "Open vSwitch," http://openvswitch.org/.

[12] Neutron, "Neutron," https://wiki.openstack.org/wiki/Neutron.

[13] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, "Optimizing vm checkpointing for restore performance in vmware esxi."

[14] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[15] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Communications in Computer and Information Science – Cloud Computing and Services Science*. Springer, 2013, vol. 367.

[16] Y. Chen, T. Farley, and N. Ye, "Qos requirements of network applications on the internet," *Inf. Knowl. Syst. Manag.*, vol. 4, no. 1, pp. 55–76, Jan. 2004.

[17] "Will aws t2 replace 30 percent of instances? not so fast," https://www.rightscale.com/blog/cloud-cost-analysis/will-aws-t2-replace-30-percent-instances-not-so-fast.

[18] "Inside amazon's cloud: Just how many customer projects?" https://www.cio.com/article/2424378/virtualization/inside-amazon-s-cloud--just-how-many-customer-projects-.html.

[19] J. Zhi, N. Bila, and E. de Lara, "Oasis: Energy proportionality with hybrid server consolidation," in *EuroSys 2016*. ACM.

[20] Giovanni Franzini, "Idle virtual machine detection in fermicloud," Fermi National Accelerator Laboratory Scientific Computing Division Grid and Cloud Computing, Tech. Rep., sep 2012.

[21] R. A. C. da Silva and N. L. S. da Fonseca, "Topology-aware virtual machine placement in data centers," *Journal of Grid Computing*, vol. 14, no. 1, pp. 75–90, 2016.

[22] J. A. Pascual, T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Towards a greener cloud infrastructure management using optimized placement policies," *Journal of Grid Computing*, vol. 13, no. 3, 2015.

[23] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755 – 768, 2012.

[24] A. Beloglazov and R. Buyya, "Energy efficient resource management in virtualized cloud data centers," in *IEEE/ACM CCGRID 2010*.

[25] H. Liu, H. Jin, C.-Z. Xu, and X. Liao, "Performance and energy modeling for live migration of virtual machines," *Cluster computing*, vol. 16, no. 2, pp. 249–264, 2013.

[26] J. Sekhar, G. Jeba, and S. Durga, "A survey on energy efficient server consolidation through vm live migration," *International Journal of Advances in Engineering & Technology*, vol. 5, no. 1, pp. 515–525, 2012.

[27] C. Ghribi, M. Hadji, and D. Zeghlache, "Energy efficient vm scheduling for cloud data centers: exact allocation and migration algorithms," in *CCGrid 2013*. IEEE/ACM.

[28] T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. G. Shin, "Litegreen: Saving energy in networked desktops using virtualization," in *ATC*. USENIX, June 2010.

[29] J. Reich, M. Goraczko, A. Kansal, and J. Padhye, "Sleepless in seattle no longer," in *ATC 2010*. USENIX, 2010.

[30] T. Knauth and C. Fetzer, "Dreamserver: Truly on-demand cloud services," in *Proceedings of International Conference on Systems and Storage*. ACM, 2014, pp. 1–11.

[31] U. Deshpande, D. Chan, T.-Y. Guh, J. Edouard, K. Gopalan, and N. Bila, "Agile live migration of virtual machines," in *Parallel and Distributed Processing Symposium*. IEEE, 2016.

[32] A. C. Snoeren and H. Balakrishnan, "TCP connection migration," Working Draft, IETF Secretariat, Internet-Draft, Nov 2000. [Online]. Available: http://www.rfc-editor.org/internet-drafts/draft-snoeren-tcp-migrate-00.txt

[33] M. Bernaschi, F. Casadei, and P. Tassotti, "Sockmi: a solution for migrating tcp/ip connections," in *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, Feb 2007, pp. 221–228.

[34] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 218–233.