Exploring Containers for Scientific Computing

J. Gomes[1], J. Pina[1], G. Borges[1], J. Martins[1], N. Dias[1], H. Gomes[1], C. Manuel[1]

Laboratório de Instrumentação e Física Experimental de Partículas Lisboa, Portugal
jorge@lip.pt

Abstract. Researchers have access to a diversified ecosystem of computing resources ranging from notebooks to computing clusters and distributed infrastructures. Running applications across these different environments requires considerable effort. The Linux Containers technology allows multiple operating system environments to be executed on top of a shared Linux kernel space. It takes advantage of standard Linux kernel features to provide lightweight isolated Linux environments similar to virtual machines, but without the overheads of conventional virtualization. In this paper we highlight the benefits of this approach, and explore paths towards the use of containers to simplify the packaging and execution of scientific applications.

1 Introduction

Most scientific applications are developed for Unix-like environments. Still porting applications to each specific computing environment requires considerable effort and expertise. Furthermore users are today faced with a wide range of computing resources ranging from notebooks to computing clusters and distributed infrastructures. These resources have their own peculiarities and interfaces that often constitute obstacles to a transparent usage.

In order to exploit the computing resources at their disposal, the researchers need to port applications to multiple computing environments, and adapt them each time the environment changes. This is one of the reasons behind the interest on more flexible paradigms such as cloud computing, where users can have their own customized environments backed by virtual machines. This flexibility comes at a price, the scientists are then confronted with the need of managing virtual networks, virtual storage, virtual machines and their operating systems. In addition, at each site the management of these virtual resources is often performed differently.

Researchers thus face a complex ecosystem of heterogeneous computing resources. In this context lightweight forms of operating system virtualization may provide consistent portable run-time environments for the scientific applications, potentially facilitating their execution across computing resources.

2 Background

2.1 Lightweight Virtualization

Operating systems enable the coordinated exploitation of computing resources by multiple processes. Traditionally all processes share one common operating system configuration and compete individually for the system resources. Conventional virtual machines are frequently used to provide better separation between sets of activities running in the same physical machines, and to provide differentiated system configurations tailored to each activity. However, this separation can also be achieved in some operating systems through *operating system-level virtualization* [1] with the following benefits:

- No emulation or hypervisor overheads. All virtual execution environments share a single host kernel through process confinement.
- Multiple file system environments can be supported within a single host by confining the processes to specific file system trees.
- Processes within virtual execution environments are transparently scheduled and managed by the host kernel, allowing more efficient use of resources such as memory.
- No extra resource consumption related to the simultaneous execution of multiple operating system kernels and operating system services.
- Faster provisioning and environment start-up.

The main disadvantages are: inability of running software that requires an operating system kernel different from the one provided by the host, and exposure of all virtual environments to security vulnerabilities in the host kernel. Both are consequences of sharing the kernel.

2.2 State of the art

Examples of operating system-level virtualization are: Solaris Zones [2] available on Solaris and OpenSolaris, FreeBSD Jails [3] available on FreeBSD systems, and OpenVZ [4], Linux-VServer [5], Linux Containers (LXC) [6], and libvirt-LXC [7] on Linux. Since most scientific computing software and infrastructures are based on Linux, we therefore focus on Linux implementations.

Both OpenVZ and Linux-VServer are mature implementations but require modified Linux kernels which constitutes a major obstacle to their wider adoption. Conversely LXC and libvirt_LXC rely on vanilla Linux kernel features and therefore do not require modified kernels. LXC is available out of the box in many Linux distributions, and is more mature than libvirt_LXC. Both allow the manipulation of kernel features to create comprehensive contained environments, enabling the execution of multiple Linux distributions over one single kernel. The most relevant Linux kernel features for operating system-level virtualization are:

- Kernel namespaces [8]: are used to isolate a particular global system resource making it appear to the processes within the namespace that they are using the full resource.

  - Mount namespaces: isolate the mount points accessible to a group of processes.
  - UTS namespaces: provide hostname and domain name isolation.
  - IPC namespaces: isolate interprocess communication resources such as queues, semaphores and shared memory.
  - PID namespaces: isolate process identifiers, so that processes within a group cannot see processes in other groups, furthermore the process identifiers are remapped allowing processes within different groups to have the same process number.
  - Network namespaces: isolate network resources such as network devices, IP addresses, and routing tables.
  - User namespaces: isolate user identifiers and group identifiers. These identifiers are remapped and can be different inside and outside of the namespace.
- AppArmor [9] and SELinux [10]: are kernel security modules that implement mechanisms to support mandatory access control security policies. They can be used to protect the host system against accidental abuses from privileged users from inside the contained environment such as changes to cgroups, or writing into devices.
- Seccomp [11]: provides system call filtering.
- chroot [12]: provides isolated directory trees.
- cgroups [13]: provide hierarchical task grouping and per-cgroup resource accounting and limits. They are used to limit block and character device access and to suspend sets of processes. They can be further used to limit memory and block i/o, guarantee minimum CPU shares, and to bind processes to specific CPUs.
- POSIX capabilities [15]: split the privileges traditionally associated with the root account into distinct units, which can be independently enabled and disabled.

The availability of these features is fostering the interest around operating system-level virtualization in Linux. Nevertheless their correct and safe use even through tools such as LXC requires considerable knowledge and configuration.

2.3 Application Containers

Docker [16] is an open-source engine that automates the creation and deployment of applications in lightweight portable contained environments (containers). Initially Docker relied on LXC to manipulated the necessary kernel features, but currently it can support multiple execution drivers. It has its own native driver named *libcontainer*, and its modular nature may in the future allow it to run with other implementations (e.g. OpenVZ, FreeBSD jails, Solaris zones). Docker inherits many of the LXC features. However with Docker the creation of containers is greatly automated and simplified with a higher level of abstraction.

The Docker containers encapsulate the applications payload allowing them to run in almost any Linux host with a recent distribution. Docker is inspired in

the intermodal shipping container metaphor, where a multiplicity of goods can be packaged and transported via a multiplicity of means on standard containers. The availability of a common agreed format to ship goods has simplified their transport, storage and management. Similarly Docker offers a uniform way to package, distribute, deploy and execute applications.

When creating a container for an application only the software components strictly required for execution are needed. Since the operating system kernel is shared, there is no need of having one in the containers. Similarly usual processes that are started at boot time are also not needed. Many functions such as time synchronization, device management, power management, network configuration and others are simply performed by the host. Still an environment that resembles a common Linux system can also be made available inside the containers, including:

- A network interface with an IP address.
- A dedicated file system for each container.
- A set of system devices.
- Interactive shell support via pseudo-ttys.

LXC and Docker have a modular storage architecture that supports Union Filesystems namely AuFS [14], Snap-shotting Filesystems and copy-on-write block devices. Docker goes further by implementing a container image format with a layered structure. This approach allows each container file system to be mounted as a stack of superimposed read-only layers, on top of which a read-write layer is added. Any changes are thus performed and reflected in the read-write layer. When execution finishes this layer can be either discarded or added to the container image as another layer. Images are tar files that contain layers, each layer is also a tar file.

Docker can act as an image builder automating the steps required to create a new image that otherwise would have to be performed manually. This process takes a *Dockerfile* that specifies as input an existing image to which a sequence of actions will be applied. These are the actions that would need to be performed manually to compile or install the required software. Moreover new files can also be added. The image resulting from this process is a new storage layer that contains the changes performed during the build process. Existing images can thus be used as building blocks to create more complete images.

To facilitate image distribution and deployment Docker can push and pull images from a remote registry. Since images are made of layers, the registry implements a transparent incremental download and upload of the images transferring only the required layers.

2.4 Orchestration

Containers offer interesting features to support the execution of applications across computing resources. However most implementations such as LXC, libvirt_LXC and Docker operate at the host level as virtual machine managers. They do not offer any orchestration or clustering features. Hence, Docker is being incorporated in many software products such as OpenStack [17] or Apache mesos [19] allowing

the execution of containers in clouds and dedicated clusters. Other related projects are: CoreOS [18] a Linux distribution designed to support the execution of services in clusters and cloud resources. CoreOS uses Docker to encapsulate and execute services. Flynn [20] and DEIS [21] use Docker to provide platform as a service features and scale to cloud providers. Both Flynn and DEIS use a git push deployment model and are targeted at facilitating the flow from development to deployment. Several other software projects around Docker are emerging. These systems are mostly intended at creating local clusters for the execution of permanent services, and to scale out to public clouds.

3 Computing with Containers

For scientific computing, the described tools do not currently offer a comprehensive solution capable of simplifying the execution of applications across the e-infrastructures ecosystem. In addition researchers want to focus on science instead of computing. They require easier tools to simplify their work, enabling them to address their scientific goals. Although cloud computing is gaining expression, most intensive simulations and data processing continues to be performed on batch farms. Both approaches are complementary and they will likely be used for a long time. Therefore scientists need a simple way to execute applications across these and other computing resources without changing them. This implies running transparently on grid computing, cloud computing and other resources. Based on these assumptions the authors developed a proof of concept aimed at testing the usability of state-of-the-art containers in the IBERGRID infrastructure. The table 1 shows the software versions used in this study.

Distribution	Virtual machine manager	Kernel
Centos 6.5	Docker 0.11.1 + execution driver lxc-0.9.0	2.6.32
	Docker 1.0.0 + execution driver native-0.2	
Centos 6.5	LXC 0.9.0	
	LXC 1.0.0	
Fedora 20	Docker 1.0.0 + execution driver native-0.2	3.14.6
Fedora 20	LXC 0.9.0	
Ubuntu 14.04	Docker 0.9.1 + execution driver native-0.1	3.13.0-29
Ubuntu 14.04	LXC 1.0.3	

Table 1. Software versions used in this study

3.1 Use case

The proof of concept is based on the following use case. The researcher uses its own workstation to develop applications which are compute and/or data intensive. He wants to execute those applications on local computers accessible via SSH, and also in IBERGRID computing resources in Portugal. The IBERGRID resources are

computing farms accessible via CREAM Computing Elements (CREAM-CE) [22]. The operating systems in the researcher workstation and in the target execution hosts are different. He wants to execute his applications in the computing resources at his disposal with a minimum effort.

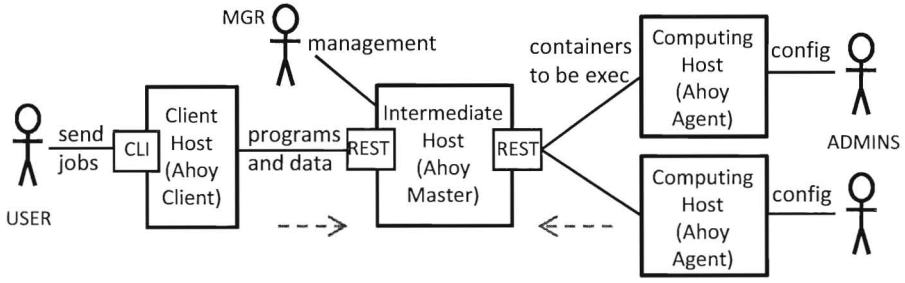

Fig. 1. Context diagram

The figure 1 shows the context diagram of the proof of concept. It shows three main components: the *Ahoy client* installed in the desktop computer, the *Ahoy master* installed in an intermediate host, and the actual computing hosts that execute the user applications which are launched through the *Ahoy agent.*

The *master* receives from the *clients* requests to perform actions on the user behalf, including building images and submitting them for execution in compute hosts. Those requests are validated and translated into container management actions. The *agents* contact the *master* periodically to obtain jobs for execution. The jobs are in fact containers to be launched. The *master* acts as an intermediate host between the user and the resources, hiding the details of accessing computing hosts and managing containers. It requires inbound and outbound Internet connectivity. A basic flow is as follows:

- Preparation of an application image:
  - The user invokes the *Ahoy client* to prepare an application image build request . The request is sent to the *Ahoy master* for processing. Therefore the user does not need to install in his computer any software besides the client.
  - As defined in the request, the *Ahoy master* builds a container using a pre-existing operating system image to which the user provided files and software packages are added.
- Execution of the application:
  - The user sends a request for application execution (job) to the *Ahoy master.* The request contains: metadata describing the job namely: the name of the application image built in the previous step, and the input data files. The files can be sent with the request itself or can be downloaded later at execution time.
  - The request is validated, prepared and queued by the *Ahoy master.*

- If needed the *Ahoy master* starts instances of the *Ahoy agent* in the computing hosts via the SSH or CREAM interfaces.
- The *Ahoy agent* contacts the *Ahoy master* to get jobs for execution.
- The *Ahoy agent* pulls the container image and starts it with the appropriate environment and restrictions.
- The user can check the job status by querying the *Ahoy master* which periodically gets information from the agents.
- Upon completion the output can be uploaded to the *Ahoy master* or other external destination accessible via supported protocols.
- Finally a cleanup of the computing host is performed by the agent.

Besides the user itself, in this scenario there are the following roles. The manager (labeled MGR) is the person responsible for the coordinated use of the distributed resources. In his role he may need to have a tighter control of how resources are shared and the need to implement policies and job resource requirements, therefore he manages the *Ahoy master*. The site administrators (labeled *ADMINS*) are responsible for managing the actual computing hosts, namely they install the software, perform its configuration, control the usage and ensure that usage policies are respected. They may work in different organizations and they have a direct concern on how and by whom their resources are used. The deployment and use of any form of virtualization requires their acceptance and cooperation.

3.2 Architecture

The figure 2 shows an architecture designed to explore the described use case using Linux containers. A demonstrator was developed in Python [25], the current prototype is focused at exploring the Linux containers features. The architecture was conceived to be generic and modular, enabling to experiment with other virtual machine managers. The *Ahoy master* implements two RESTful APIs the first is used to receive requests from the *Ahoy client* and the second to receive status information, send commands and jobs to the *Ahoy agents*. Both clients and agents need to communicate with the master. Since they are frequently behind firewalls the communication is always started by them. There are no permanently open communication channels. Connections are established by the agents periodically to send status information and pool for new jobs to process. From the point of view of the user most details are hidden, and he can use a set o simple actions to prepare and execute applications.

Creating base images With the *Ahoy client* tool the user can produce base operating system images on top which applications can be added. The aim is to empower the user to create base images of his operating systems, thus ensuring that applications will run on the same environment. The images are minimal by default but the user is free to add packages. These images are mainly produced with tools such as *yum* [23] or *apt* [24] depending on the flavor of the Linux distribution. Unfortunately creating such images requires privileges that may not be available to the user. In that case pre-built images available from the *Ahoy master* server may be used instead. The base images are stored in the *Ahoy master* repository, and become available as building blocks to create application images.

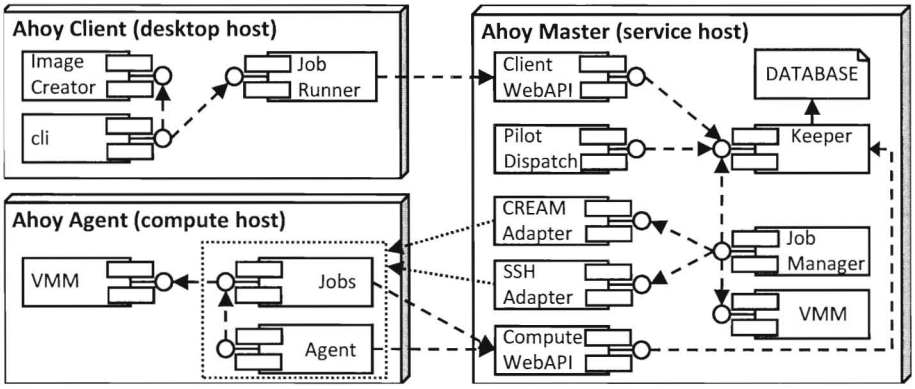

Fig. 2. Architecture diagram

Creating application images Application images are created by combining a base operating system image with the user specified software. The *Ahoy client* takes as input programs or directories to be merged with an existing operating system image. The operating system image is identified by its unique name and must already exist in the *Ahoy server*. The user can also specify additional software packages to be added to the final combined image, these are installed via *yum* or *apt*. The user does not need to specify the software dependencies since the *Ahoy client* is capable of analyzing the input files and packages and find their dependencies. This ensures that any required software packages and libraries are added. Finally the user may choose the program to be executed when the container is started, otherwise a wrapper will be the default executable. The *Ahoy wrapper* can hide much of the details of downloading input files and uploading results, in addition can also be used to download the executable thus making the application image more flexible. The actual operation of building an application image is performed on the *Ahoy master*. The resulting image is then made available by pushing it to a Docker repository to take advantage of the Docker layered images. The image can also be made available via a web server or via the *Ahoy master* itself.

Executing an application A job execution request contains: the name of the file to be executed, a list of input files and a list of output files. Additional information regarding the job requirements can also be added such as the amount of memory and number of processors. There is no need to pass information about the operating system software environment because those requirements are already encapsulated in the application image. In addition the mounting of volumes from the computing host can also be requested. The volume names must be coherent across the infrastructure. Each name is mapped by the *ADMINS* at each site into a local directory to be mounted in the container. The mapping is defined in the *Ahoy agent* configuration file. This feature allows access to data already available

at the site. The input files can be uploaded from the *Ahoy client* to the *Ahoy master* together with the job execution request. Alternatively the list of input files may contain references to files available from an external ftp or web server. The executable can be: a file previously added to the image, a file uploaded by *Ahoy client* jointly with the other input files, or a file to be downloaded from an external server via the *Ahoy wrapper*.

Upon receiving a request, the *Ahoy master* performs a validation and checks if there are free resources capable of executing the job. This is verified by checking the status and capabilities of the *Ahoy agents* that are active and ready to pull jobs. If enough *Ahoy agents* matching the request are available and free then the *Ahoy master* just waits to be contacted by one of them, at that moment the job is given to that agent. If no resources with the required characteristics are free, then the *Ahoy master* searches the list of potential dynamic resources that are alive and submits the required *Ahoy agents*. The *Ahoy master* can be configured to over provision agents thus allowing request to be quickly served.

The *Ahoy agent* has two modes of operation. A static mode where it runs as a persistent system process (daemon), and a dynamic mode aimed to be started on-demand by the *Ahoy master*. If a dedicated pool of compute hosts is needed then these nodes can be setup in static mode. The user workstation can also be setup as a static compute host. The agent is a lightweight process that can be launched manually whenever needed. The dynamic mode is more suitable for transient resources such as cloud and grid computing resources that may be available for a short period of time and have to be activated when needed. Currently the *Ahoy master* has adapters to support the activation of *Ahoy agents* in dynamic mode via SSH and via the grid cream-ce. The adapters encapsulate the details of starting the *Ahoy agents* in each type of resource. They are modular and easily implementable. The *Ahoy master* itself is also a lightweight process that can be installed and operated by an individual user to exploit resources at his disposal. However it requires inbound network connectivity.

Currently, starting the application images as containers requires privileges on the compute host. Therefore one of the functions of the *Ahoy agents* is to allow an unprivileged user to start the containers in a secure way. Both the *Ahoy master* and the *Ahoy agent* supports abstraction interfaces to perform container actions using virtual machine management systems. Abstraction interfaces for Docker and LXC are implemented. Deploying an agent implies the installation of LXC or Docker, as well as the agent itself. The installation requires privileges and therefore the cooperation of the *ADMINS*.

The abstraction interface for Docker needs to communicate with the Docker service via its protected UNIX socket. For security reasons only trusted users should have direct access to this socket. Therefore the *Ahoy agent* must be installed under a dedicated UNIX user with access to the socket. This can be easily accomplished by creating a dedicated unprivileged user and adding it to the docker UNIX group. The agent can then be started by end-users through the UNIX *sudo* command, which needs be configured to start the agent under the control of the dedicated user. This is the default approach followed by the *Ahoy master* to start the agents on computing hosts.

The agent can be configured by the *ADMINS* to impose limits such as memory consumption. The specification of volume names and their mapping into local directories is also accomplished in the same manner.

The *Ahoy agent* periodically contacts the master to send its status and check for applications to be executed. When it gets a request it downloads the application image either from a web server or from a Docker repository. In simple scenarios the *Ahoy master* can also serve the images directly from its internal repository. The image is then started by the agent with the appropriate parameters such as: local volumes to be mounted, resource usage restrictions, and also with job metadata that is translated into environment variables. Once started, the *Ahoy wrapper* inside the container uses the environment variables to perform actions such as downloading the input files and starting the application. Upon completion the wrapper transfers the output files back to the *Ahoy master* or to an ftp or web server. The *Ahoy agent* can then transfer the container log files, notify the master that the execution has finished, delete the image, and eventually exit. The *Ahoy agent* can log information about the execution of the application for accounting and auditing purposes which becomes locally available to the site administrators.

The *Ahoy master* periodically receives information about each job from the *Ahoy agent*. This information can be polled by the *Ahoy client*. When the job finished the user can download the output files from the master or other external location.

During execution the user can cancel the execution of its jobs, the request is sent to the *Ahoy master* and relayed to the agent upon one of its contacts. In such case only the execution logs are made available.

3.3 Findings

Docker actions are performed by a daemon process that requires root privileges to create network bridges, manipulate firewall rules, mount file systems, or extract image layers with files owned by others. It uses a Unix socket for communication with the clients using a RESTful API. However this protocol does not have authentication. Any user with access to the socket is thus able to have full control over the Docker operation.

By default the processes running in Docker containers appear both to the host and to the container as root processes. This may lead to traceability and auditing issues since it is not obvious who started and is actually executing the contained processes. However, it is also possible to execute Docker containers under unprivileged UNIX user and group identifiers. In this case both from the host and container perspectives the user running the processes will be same unprivileged user.

Docker can mount local directories inside containers. Therefore a user with access to the Docker can start a container with both root privileges and a host file system mounted in. In this situation the user can from inside the container change the host file system. Docker can also be told to remove several important security restrictions potentially allowing breaches from the container. Due to these and other features the Docker cannot be made directly available to the end-users.

In this respect Docker it is not different from most virtual machine managers, but unfortunately this constitutes a limitation to its direct use by end-users.

The ability to run as root inside the isolated containers is an attractive feature that akin to conventional virtualization largely contributes to the versatility of the containers. There are however considerations that should be carefully evaluated. Processes inside containers run directly on top of the host kernel, therefore security vulnerabilities in the kernel can be exploited by processes inside containers to gain access to the host system and to other containers. There are several measures that LXC and Docker perform to mitigate this and other risks, such as dropping some of the POSIX capabilities to reduce the root privileges, or the use of Seccomp to limit the access to system calls. It is highly advisable to enable SELinux or AppArmor in the host system to further constraint the attack surface in case of container breach. It is important to understand that the Linux containers stack is still fairly recent and not yet complete, therefore processes inside containers should preferably run without privileges.

Due to the extensive security restrictions imposed on containers there are operations that by default even the root inside a container cannot perform. Examples are: mounting file system, use raw sockets, use protocols such as GRE, create device nodes, change certain file attributes, load kernel modules. These limitations may in some circumstances create problems. For instance using GPUs requires direct access to devices. Fortunately many of these issues can be circumvented by passing appropriate directives upon container start.

The use of Docker with batch schedulers is challenging mainly because all processes within containers are started by the Docker daemon which is not controlled by the schedulers. This poses additional accounting and policy enforcement problems. In this regard LXC has the advantage of not requiring a daemon thus allowing the control of the processes by the scheduler. Furthermore LXC already supports the direct execution of containers by unprivileged users while still enabling root privileges inside the container via user namespace. This feature can simplify the use of containers with benefits in terms of security, traceability and accounting. However it requires both very recent Linux kernels and security components not yet widely available in the Linux distributions.

4 Conclusions and Future Work

LXC and Docker are powerful tools that ease the creation and deployment of containers. However they still have limitations in terms of security. Several issues have been identified, but LXC and Docker are still being heavily developed and many of the current limitations will be addressed as they become more mature.

Docker is currently more suitable to the development and execution of services, thus many of its design characteristics at intended to be used by privileged users. The evolution of LXC towards direct usability by unprivileged users seems to be more promising as a method to encapsulate applications. However the Docker layered images are more flexible, and they fit very well the incremental building and transfer of applications in distributed computing environments.

The *Ahoy* demonstrator is currently able to execute across the intended computing resources with the described limitations. It was possible to executed several common operating system environments and applications on these resources. We aim at continue the development of the presented architecture and framework to explore the coordinated deployment of scientific applications across heterogeneous computing resources. The next steps include: addressing the authentication issues, exploit cloud resources, enhance the current abstraction interfaces and improve the orchestration.

Acknowledgements

This article was only possible due to the financial supported provided by the FEDER funds trough the COMPETE program and by Portuguese national funding trough the Portuguese Foundation of Science and Technology (FCT).

References

1. operating system-level virtualization http://wikipedia.org/wiki/Operating_system-level_virtualization
2. Solaris Zones https://java.net/projects/zones
3. P.-H. Kamp and R. N. M. Watson: Jails: Confining the Omnipotent Root. In Proceedings of the 2nd International SANE Conference, Maastricht, The Netherlands, May 2000.
4. OpenVZ http://openvz.org
5. Linux-Vserver http://linux-vserver.org/
6. Linux Containers https://linuxcontainers.org
7. libvirt LXC container driver http://libvirt.org/drvlxc.html
8. Namespaces in operation https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt
9. Application Armor http://en.wikipedia.org/wiki/AppArmor
10. Security Enhanced Linux http://selinuxproject.org
11. SECure COMPuting with filters https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
12. chroot http://wikipedia.org/wiki/Chroot
13. Control Groups https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt
14. AnotherUnionFS http://aufs.sourceforge.net/aufs.html
15. Linux Capabilities https://www.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt
16. Docker http://www.docker.com
17. OpenStack https://www.openstack.org
18. CoreOS http://coreos.com
19. Apache Mesos http://mesos.apache.or
20. Flynn https://flynn.io
21. DEIS http://deis.io
22. Cream Computing Element http://grid.pd.infn.it/cream
23. Yum Package Manager http://yum.baseurl.org
24. Advanced Packaging Tool https://wiki.debian.org/Apt
25. G. van Rossum, Python tutorial, Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.