Lomonosov Moscow State University

Faculty of Computational Mathematics and Cybernetics

Department of Automation Systems of Computer Systems

Vasilenko Anatoly

521 group

Network applications requirements to lightweight virtualization for network modelling usage

Moscow, 2016

1 Contents

2 Introduction

Modern IT world continues to evolve, creating new concepts. In particular, despite the already great age of this industry, it is happening in the world of network technologies. The most striking examples of contemporary trends in the world is the concept of *SDN* and *NFV*, which are associated with the other trend of general IT: virtualization.

For networking industry virtualization technology based on containers can be used in two ways: running abstract application within the container, which generates traffic to be routed, and running virtualized network service within the container, which purpose is to handle the traffic passing through it.

These trends are in active development and technology continue to improve, gaining new functionality.

The goal of this paper is

1. Exploring the possibilities offered by the containers and matching them with the requirements of applications. If possible identifying drawbacks of lightweight virtualization applied to network modelling.

The tasks that have to be solved to achieve the goal:

1. Explore the possibilities and shortcomings of modern lightweight virtualization technologies and compare them with the possibilities of a fully-fledged virtualization.
2. Identify the possible usecases of lightweight virtualization in today's world.
3. Examine the role of lightweight virtualization for modeling networks, and mark its functional requirements.

The work structure is based on solving the established tasks to achieve the goal of the essay after the preliminary acquaintance with the subject area. With every step, if possible, will be revealed disadvantages of modern virtualization technologies that are essential to run applications of various kinds, particularly in network simulation. In conclusion, the necessary requirements for the simulation related to lightweight virtualization will be stated.

3 Introduction to the subject area. Definitions

Virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, operating systems, storage devices, and computer network resources.

Virtual machine is an emulation of a particular computer system. It is usually software-based and uses different types of hardware acceleration technologies: Intel-VT, AMD-V.

As the concept of virtualization technology grew, people work in the direction of overhead minimization. Hardware support has been created and various hypervisors were developed (KVM, XenServer, VMWare ESXi ...), systems dealing with the management of virtual machines for deployment and support of cloud computing (OpenStack, VMware vSphere ...) were developed.

An idea of virtualization at OS level appeared: containers (LXC (Linux Containers), OpenVZ, FreeBSD jails, Solaris zones). On its basis community created management systems like Docker (based on LXC, libcontainer) or Virtuozza (based on OpenVZ).

Linux Containers (**LXC**) is an operating-system-level virtualization environment for running multiple isolated Linux systems (containers) on a single Linux control host.

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux. It allows you to "pack" the application with all its surroundings and dependencies in a container that can be moved to any Linux system supporting Docker.

NFV (*Network Function Virtualization*) is a network architecture concept that uses the technologies of IT virtualization to virtualize entire classes of network node functions into building blocks that may connect or chain together to create communication services.

In this paper, I will address issues of simulation and emulation of network nodes, so I will define the necessary terms. **Simulation** is an imitation of any physical process using an artificial system. **Emulation** is a set of hardware or software designed to copy the functions of a computer system *(the guest)* on the other *(the host)*, different from the first system, to emulate the behavior as close as possible to the behavior of the original system. Thus, the purpose of the simulation is to produce an imitation of any object for an external observer, and the purpose of emulation is the most accurate reproduction of the internal behavior. The essential difference leads to a difference in their purpose: the main objective of the simulation is an **API** (*Application Programming Interface*) simulation allowing development of external systems using its API. Whereas emulation is usually used

to launch an existing object, but under different environment[1]. Emulation also solves the problem of the simulation however tend to be slower.

Unlike the simulator, emulator has its own internal clock, counting the steps, according to which the system is operating. This may allow modifying the performance of the emulated system by a factor k, wherein if the performance of the external component does not change, in terms of an emulated system, and if everything is compatible, it would appear for system that environment performance respectively lower or higher by a factor of 1/k.

The network model consists of the following main parts: the description of the topology and resources, the description of the data streams and channel load, description of the traffic processing ([3]).

4 Virtualization

The main advantage of virtualization is the ability to run multiple independent virtual systems on single computer.

After years of development, virtualization technologies have the following advantages:

1. Implementation on a single computer system, several different virtual systems (it allows separation of resources between virtual systems and allows applications to run in their environment).
2. Isolated execution of applications without ability to influence other systems. (security)
3. Emulation of different architectures for computer systems.
4. Ease of creating, deploying, and even online migration of systems. (flexibility)
5. Automate management of multiple virtual systems[2]. (flexibility)
6. Using the appropriate drivers and hardware acceleration technologies, the hypervisor overhead in general-purpose systems dropped down to about tens of percent[3]. (performance)

The following features should guarantee this:

[1] Examples of emulators are: hypervisor QEMU, FPGA (Field-Programmable Gateway)
[2] Exists many of such projects. The most successful are OpenStack, Docker, Amazon Web Services EC2, Virtuozzo.
[3] Speaking of reasonable performance, I am referencing hypervisors that do not emulate different platform from host's platform (as an examples can serve hypervisors VMware ESXI, KVM, but not QEMU)

1. Modeling necessary logical resources and possibility to hand over direct control under certain physical resources to a particular virtual machine.
2. Management of physical resources (CPU, network adapters, memory, storage), control and restriction of the resources use.

Considering virtualization within the containers, there is a restriction by design: as the containers are working on the operating system level, the guest cannot use different operating system.

Note that hypervisors have been created primarily for the purpose of isolating the load (control of resources) and the creation of a virtual environment, but strictly speaking, it is the operating system responsibility, and therefore virtualization on its level is more appropriate.

In practice, virtualization is often used for the following purposes: organization of data centers (in particular for provisioning IaaS, PaaS, and SaaS services), organization of testbeds, developers use and use for educational purposes.

5 Simulation and emulation

The simulation or emulation is one of the clearest examples of lightweight containers use cases in the networking industry. On this basis, different network models created, including a variety of systems and services. Its purposes can be categorized as follows (in different experiments, goals can be mixed):

1. To provide a network model to a certain software or hardware devices to be tested.
2. For testing algorithms, directly interacting with the model and using it as an object of influence (for example, the study of the protocol convergence or topology properties).
3. For testing algorithms, which use network only for communication, and do not influence it directly. (e.g., the study of the worm propagation through the network)

Simulation system according to the authors of [3] should have the following components:

1. Run-time environment necessary for the model functioning.
2. The model description, creation and management subsystem.

3. Analytical tools (e.g. monitoring tools) and experimental data simulation.

Also will be useful following components:

4. Subsystem of emulation time management.
5. Experimental data visualization.

Proper time management is important for several reasons ([3]):

1. Synchronization between different nodes (for example, for the correct tracking of log files, or for the correct modeling of the experimental load)
2. With the ability to change the time, you can compensate some of the delays. You can also slow down time (lowering performance) to achieve the effect on a particular node of seeing the outer systems more efficient (however, it is primarily a feature of emulation, not simulation).

In my opinion it should be clarified which uses simulation and emulation and where. Usually one node topology is emulated, i.e. it creates a virtual system (e.g. container), which performs some operations (for example, generates network traffic, handles requests to the database ...). However, considering the network as a whole, it is correct to speak of the simulation, because another type of problems are been challenged (e.g. alignment of delays between all connections (real and emulated that connect two virtual systems in a single host)). Thus, there is a mixture of emulation and simulation, which should always be borne in mind. As a result, for example, in [6], which tells about MaxiNet, based on Mininet (software that allows you to emulate the system based on the containers within a single host), authors prefer to call their system emulator. On the other hand, for example, in [5], which tells about PRIME *(scalable simulation system for productive servers)* authors talk about their product, as a simulator. And this is correct, but sometimes in different papers authors are mixing this concepts.

Simulation and emulation always have limitations, it is necessary to monitor the correctness of the model, because it must correspond to the experiment to give satisfactory conclusions. The model should not be too general and neglect important experimental parameters. Thus, for example, it can be quite difficult to carry out stress testing for some real equipment or simulation does not allow finding some types of the bugs in the applications and their configurations.

Network simulation is a compromise between the detailed network description, the complexity of the network, accurate behavior predictions and configuration complexity of the model ([3]).

6 Linux Containers

6.1 General information about Linux containers

It is expected that Linux containers are being profitably than virtual machines, because there is no extra operating system for each virtual machine.

As part of OpenVZ the possibility of transferring the container to other equipment on the fly are developed (the project named *"CRIO").* However, it is better to create applications, so that they can be at any time deleted and recreated that is creating "stateless" applications[4].

One of the most convenient features of the container is its mobility in terms of rapid deployment. It also enables repeatability of the experiment designed in the form of a container. It is very suitable for research ([4]).

In paper [7] authors tested the possibility of monitoring systems and the number of containers that can be run simultaneously. They used Dell PowerEdge 1950 server with two Quad-Core Intel Xeon E5430 (2.66GHz) processors, 8GB RAM and Gigabit Ethernet NIC. For testing, they used OpenVZ containers, with launched apache, and as a client, they used wget. This choice was made because they intended to load not processor and memory, but block I/O and network interfaces. As a result, the equipment allowed running on the server 600 containers simultaneously.

6.2 Internals of Linux Containers

Main types of resources available in system is CPU, RAM, network and block I/O. However, for executable process there is extra “resource” - environment: process signals transmission, user id, file system ... Container tasks combine the main tasks of the hypervisor and operating system: management of physical and logical resources, support for environment variables and implementation of inter-process communications.

[4] Application integrity and fault tolerance against a sudden errors must be implemented by the application itself in the first place, but not by the environment (container)

6.2.1 Namespaces

Namespaces represent a group of processes having access to some particular resource or environment element. When implementing the container, i.e. isolated environment for processes execution, it is necessary to imitate next scopes for each virtual entity:

1. File system namespace. It can change its structure and its access rights to individual parts (can be emulated in different ways[5], allows realize innovative solutions, such as the distributed file system UnionFS)

   When creating a new scope, all parameters will be copied from the current scope, in which process works. Mounting of different partitions is not seen across the scopes (unless the process of the scope enables it to be seen), but mount made in host machine will be seen to everyone.

2. Network namespace. Enables the isolation of virtual devices, ip addresses, rules and routing tables, filters, open sockets ...

   Network interface and socket may simultaneously belong to only one network scope, but been transferred to another. In practice, usually, virtual interface is created in each container (for each network scope), via which the Linux-bridge mechanism is connected with other network interfaces and host.

3. UTS namespace (*unix time-sharing*). This gives the possibility to change the name of the host *(hostname*) and NIS domain name for processes within the appropriate scope in a data structure returned by command "uname".

4. IPC namespace. Within this scope, processes can use means of interprocess communications (IPC) between themselves (shared memory, semaphores, messages queues) isolated from other scopes.

5. PID namespace changes the number of process within a specific scope. Usually process has two identifiers, one within the namespace, and another across the system, but if the process is in several scopes, then greater number of identifiers can be assigned.

   Depending on processes scope, the folder "/proc" will be simulated for it accordingly.

6. User namespaces (UID/GID) allows processes to work simultaneously on behalf of different users, depending on the scope. (For example, the

[5] It is considered as an extension of the possibilities offered by "chroot"

processes user id can equal to zero within a container, but it will not have privileged rights outside the namespace)

There were additional namespaces discussed in the world of IT:

1. "Security namespace" - the scope to isolate the security policies, and "Security keys namespace" (the aim is to provide a place to store security keys isolated from each other). However, they are still not developed, and the only security mechanism is based on "user namespace", additional security must be implemented by other means (e.g. SELinux).
2. "Device namespace" ([15]). Existing capabilities allow isolating network devices and file systems in their scope, but lacks the possibility of isolation and separation for other devices on Linux.

   For example, this could come in handy in Android devices, to organize two phones in one, so that human work and personal phones are not mixed and have been isolated from each other, which would increase the level of security. It also will enable companies to introduce more strict policy on the corporation phone part.

   Note that in this case it is necessary to solve the problem of interactive scenarios: “which container should have pass access at the moment?”, for example when the screen is touched by the user. (Giving simultaneous access to both containers will not be safe solution because the container could interfere into operation of other container and specific device).

   However, this concept also touches another subject in the context of the scope – saving the state. It is needed, because division between corporate and personal entities in phone is constant. System that will cope with this task must be created. Considering Docker as one of the possible alternatives is not a good idea, because despite the fact that it can somehow cope with this task (freezing containers, creating images and deploying them when the system is rebooted), it is still a system for bigger purposes and is not optimal for discussed problem.

   In 2013, discussions began about creation of "device namespace", were created patches for the Linux[6], but as the full commissioning of this technology involves rewriting a lot of drivers for various types of devices, the work has not yet been continued.

[6] https://github.com/Cellrox/devns-patches/wiki/DeviceNamespace%3APatches

3. "Time namespace". This type of scope implies processes see their own time in each namespace. This idea is not widespread, perhaps because the developers did not really need it; it is much easier to make the necessary changes with time directly in application.

Scopes for processes are recorded as files in a virtual file system on the path "/proc/<pid>/ns".

Related processes communication (pipes) are not altered, i.e. if the related processes are in different scopes, they can still use these methods to communicate.

6.2.2 Cgroup

Cgroup (Control group) are used to control resources. It sets scopes associated with a set of parameters and constraints. Process groups may be arranged hierarchically in a tree structure inheriting the parameters of each other. Cgroup controls network, memory, CPU usage and block I/O, that is, it is a tool for group resource management and offers advantages:

1. Limiting the resources. For example, it can be configured to control the memory limit or control access to block devices.
2. Prioritization. Some groups may get more CPU time and disk I/O.
3. Resources accounting: measuring the amount of resources that are used by certain processes.
4. Control. Freezing groups of processes, saving intermediate states and restart.

To implement these features control groups have various subsystems:

1. *blkio* - subsystem limiting input\output of block devices (disk, usb, ...).

   You can specify the priority in each group for each device (can be specified for both reading and writing (op/s or b/s))

2. *cpu* - subsystem, which uses the task scheduler to distribute access to the CPU.

   It allows specifying the priority, but cannot set hard limits on the distribution of CPU time. Thus, we can define the percentage of CPU time that will be given for the group; however, these relations can be violated in case the machine is overloaded.

3. *cpuacct*. This subsystem generates automatic reports of cpu usage by control groups.

4. *cpuset* - subsystem designed to match the selected control group and specific processor cores or memory slots[7].
5. *devices* - subsystem controlling read/write access permissions for device.
6. *freezer* is a subsystem for stopping and continuing processes in the control group.
7. *Memory*. This subsystem sets limits on the amount of memory used by the control group, and generates automatic reports on memory usage.

   Limitations for the control group can be set simultaneously in two types: hard and soft limits. Soft limits are not coercive, but hard limits will lead to immediate action, such as freezing processes in the control group, or removal of processes, changing limits, notice userspace …
8. *net_cls*. This subsystem marks traffic with labels called "classid", allowing traffic controller on Linux to determine the source of the package. It is necessary for correct functionality of tc/iptables.
9. *net_prio* allows to dynamically prioritize network traffic for different network interfaces, that will be considered in queuing. Processes only outbound traffic.
10. *hugetlb* allows limiting support for large memory pages in the Linux kernel (This subsystem is not widespread and is hardly ever used).
11. *perf_event* allows to track resource usage by control group in general, as a single process with the utility "perf".

Management of control groups is made through file system, i.e. delete/create directory, mount directory into "/cgroup" or into "/sys/fs/cgroup" for systemd. The file subtree is virtual, and all changes will be lost after reboot. Folder names do not matter, but matter flags, which is used to mount directories. Hierarchy of controls can be created in file system. When you create a control group, files that specify necessary settings are created, for example, PID or TGID of cgroup processes. "Systemd" based systems have some differences, but the functionality is identical.

6.2.3 Namespaces and Cgroups technologies together

Namespaces technology allows creation groups of processes that have different views on the available resources within a single operating system, and cgroups technology, allows controlling groups of processes, non-connected with

[7] Memory fixation between processes may be very important for multiprocessor systems (e.g. NUMA)

each other, and can be used both separately and altogether. Moreover, scopes of namespaces and cgroups can be quite different.[8]

Thus, altogether, namespaces and cgroups may be used as required in particular project, and there must not be used all features they present, they are very flexible. At the same time, they provide the main goal of virtualization: the control of resources and isolation.

However, the above technologies have the following drawbacks[9]:

1. Sometimes it is difficult to change the namespace (or control group) from one to another. This is mainly due to the non-obviousness of the API or some dependencies and access rights.
2. Drawbacks of hard resource division. This applies to those resources which limits are specified by priorities and not in absolute values. It is concerned the use of CPU and block I/O operations.

   In general prioritizing works properly for resource division until there is enough resources, but if system is overloaded, the division of resources gain an unpredictable manner.
3. The security problem of control areas and namespaces. RedHat disabled namespaces by default in Linux 7.1, because they believes that the technology is not yet sufficiently developed and community need better understanding of consequences of this technology for system security and must be reduced the security risks to eliminate any attack vectors associated with this technology.[10]

   The most common problem is a problem of inconsistency between namespaces and cgroups scopes. This, for example, can lead to the following consequences:

   1. Some system calls, and some parts of virtual file system may be accessed to retrieve information, which must not be available for container.
   2. Container processes, despite the fact how much resources it is permitted to use by means of cgroup boundaries (for example,

[8] Previously each cgroup scope corresponded to some namespace scope, but this concept were refused.

[9] Because of the fact that technology is still evolving, some features have not yet been implemented, and some drawbacks are still not known.

[10] In September 2015 namespaces are included in the RedHat distribution as a "Technology Preview". Until then, one of the key communities that have been developed these features was Fedora community, which enabled namespaces in their distributions. (http://rhelblog.redhat.com/2015/07/07/whats-next-for-containers-user-namespaces/).

cpu and memory), can see the list of all resources in the system: both accessible and forbidden.

For example, a file system privately mounted in some scope will be visible to all utilities using mount, because it takes the information from the "/etc/mtab" file, while the actual available mounting points are stored in "/proc/mounts". Thus, despite the fact that access to a mount point in a file system is not available, we know about its existence ([11]).

The **LSM** systems (*Linux Security Modules*), for example, SELinux, AppArmor, GRSEC, - are not properly thought about in terms of the namespaces and cgroups. Combined work of these technologies are still under discussion, and in fact did not reach the issue of implementation.

Thus, the only privilege separation way is "userspace namespace", allowing to change the UID/GID, isolation of super user (super user within the container is not the one in the host system), and the mechanism CAP_SYS_ADMIN, which allows division of super user privileges among different users.

6.3 Linux Containers (LXC) as an approach to virtualization

Linux Containers (LXC) can be considered as a technology and as a concrete implementation. Linux Container (LXC) is a technology that combines cgroups, namespaces, chroot, Linux Security Modules (LSM) and Mandatory Access Control (MAC) and provides a more abstract interface for Linux user, for organization of virtualization at the OS level. In practice, this translates for the user as a set of features in userspace, mainly for cgroups and namespaces, at a more abstract level (Figure 1 shows the level at which operates LXC).

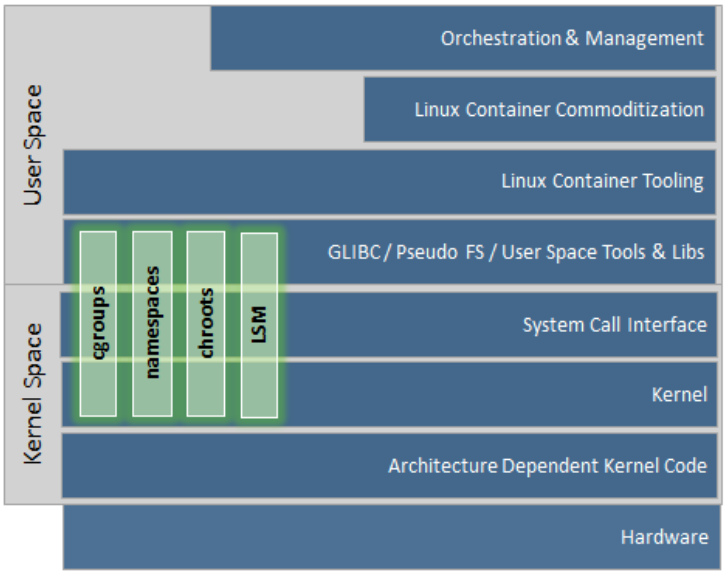

Figure 1. Linux Container Technology Stack

LXC implementation allows you to work with containers controlling necessary parameters on the basis of which namespaces and cgroups are created: hostname, the first process and its UID/GID, network settings, file logging, mount points in the file system and rights, setting any cgroup parameters, SELinux context, interceptors for certain events that happen within the container, you can also pass additional environment variables.

LXC support compatibility with "Copy on Write"[11] functionality based on specific file systems, which allow forming a cascade-united mount (for example, AUFS). This allows you to run many identical containers based on a single instance in file system. The changes made by each container recorded separately and can be used to create new file system image after user request, taking into account all the changes. This approach allows you to run in a very short time a great number of containers saving a lot of time and resources.

Comparing OpenVZ and LXC features ([21]), the only LXC weakness compared to its opponent is the lack of ability to control the quality of service provided by block I/O. In addition, unformally it is considered that the containers within OpenVZ are safer (better isolated and more secure) in comparison with LXC.

Of the possible LXC drawbacks, there can be enumerated absence of automatic containers backup, possibility of recovery in case of crashes and the lack of centralized tools for network management. However, the first two disadvantages are the area of applications responsibility by themselves in first place but not containers. A final drawback is the responsibility of the utilities involved in the

[11] Combined work of containers and shared file systems actively used in Docker project.

orchestration. History has shown that main drawback lies in the API provided by LXC, it is not optimal to be used by management projects (e.g. Docker), which is why they moved to its own implementation of the concept LXC called "libcontainer". Another significant disadvantage is unproven security of LXC containers[12], that is why it is recommended to use some system of rights differentiation, when using LXC containers (for example, GrSecurity or SELinux). At last, the lack of container migration without closing and restarting must be pointed out as the disadvantage (the project named "CRIO" based on OpenVZ has some noticeable results).

6.4 Docker - lightweight virtualization management system

Docker provides opportunities for the deployment of applications formalized as containers; provide resource management and setting limits. In fact, Docker approached the concept of treating the infrastructure as an application, which leads to the fact, that Docker images do not have dependencies, they are self-sufficient.

Docker has a client-server architecture: daemon controls images and containers, and client sends commands and can be positioned on any other machine. Docker works with three main entities: *Docker image* - a pattern that is available only for reading, it is the basic unit of distribution (distributed in the format "Dockerfile", containing the image related information and instructions for the correct preparation of the necessary resources to run the container), *Docker registries* - storage of images[13] and *Docker containers* - executable entity that can be started, stopped, destroyed …

Docker, using the concept of Linux Container[14], provides the following features:

1. Deployment of containers on multiple machines; the Docker image can be started on any machine that supports Docker technology.
2. Automatic assembly: the creation of containers with regard to configuration management systems (e.g. Chef, Puppet, Maven, Ansible, Salt, ...)

[12] This problem occurs primarily due to the security problem with namespaces and cgroups technologies.
[13] Publicly available Docker Hub is an example of Docker images repository. Repositories can be both public and private.
[14] Initially, the Docker project was based on technology LXC, but moved to its own implementation of Linux Containers called "libcontainer" (it is believed that this happened, because LXC provided too high-level interface, which did not satisfy the requirements of Docker).

3. Versioning system for Docker containers working similar to version control system named "git", memorizing the changes within the container.

   Exists the ability to support public and private repositories, such as "Docker Hub".

4. Reusing components: each container can be used as a basis for the creation of another container.

5. Instrumentation: CLI (command line interface) and REST API are available to manage Docker.

Docker must be treated as a mean for containers distribution and automation of its launching. However, despite its potential, he cannot act as a full virtualization platform because some valuable features are missed or do not have proper degree of development, e.g. components for logging, authentication, statistics collection, orchestration. Thus, Docker can be used to create more complete systems, which allows managing large virtual computing systems (e.g. data centers). However, this does not mean that it cannot be used alone for some types of systems that does not require the traditional structures for cloud computing.

In opensource, it is often said that Docker is very easy to use, which leads to some misunderstandings in general public. It should be understood that Docker is another management tool designed to simplify the deployment of Linux containers. If the project does not use containers, then Docker usage will only add complexity to the system, which will be located in the area of administrator's responsibility. From the viewpoint of the administration, it will be necessary to control ([20]):

1. Orchestration container, which has to be deployed instantly, without any delay.

2. Orchestration container should be able to roll back other containers and settings.

3. It is necessary to maintain network between containers on multiple hosts.

4. Support system for collecting telemetry from various containers.

5. Centralized collection of log messages from all containers.

6. Processing containers data (databases, object storages, file systems ...). It is also necessary to set rules, according to which data will be saved or deleted.

These capabilities must be maintained for a scalable system, and you will have to either use a system like OpenStack, changing it to use Docker as a hypervisor[15], which allows you to run multiple virtual machines, or in some way to adapt the "Configuration Management Tools" (for example, Ansible, Chef, Puppet). Also, these systems will be responsible for the deployment of machines with running Docker-servers ([20]). However, all of these approaches are ways to adapt the existing control systems with different purposes. Also, their adaptation will be complicated because Docker API and its internal structure are undergoing great changes from version to version, which makes it difficult to use some of the features and hard to tune the Docker usage.

Docker does not simplify application development by itself, and do not attempt such a task. Developers still have to implement security, principles of least privilege, think about making checkpoints, monitor the behavior of the critical services, create documentation ... ([20])

Comparing with virtual machines Docker security level is lower because containers interacts with the operating system kernel more.

Docker allows you to configure all main namespaces when starting container (PID, UTS, IPC, network, user security configuration) and all major cgroups (Memory, CPU, Block IO, Network adapters). In the parameters of memory, you can specify soft and hard limits onto the swap file, total amount of memory and available memory blocks in the case of NUMA systems. For the processor, you can specify the number of each core, which may be used by container and the priority of the container, according to which, task scheduler will divide CPU time between them. However, there is no possibility to set absolute limit for CPU usage for container. A similar flaw is present for block I/O. It is possible to specify priorities, but you cannot set hard limit for read/write operations.

This drawback is very important for actual systems (for example, for hosting services), and for the simulation systems, for example, in case we need to simulate the behavior when some nodes are overloaded.

If we consider a competitive system called virtuozza[16] based on OpenVZ technology, they support additional opportunities to limit resources. It is possible to set hard limits for container on CPU usage; you can set a quota or indicate priorities for the block I/O for each container, or for each individual user.

[15] This decision is wrong because we should not build one management system on top of another management system. It is much more productive to use LXC directly from OpenStack (how it is done already).
[16] Project is commercial and works only for specific Linux distribution.

6.5 Comparison of Linux containers and virtual machines

Virtual Machines are safer compared to containers because have less interaction with the kernel of host system, and consequently, they potentially have less vulnerabilities. In addition, hypervisors can run different operating systems. On the other hand, the container has a number of advantages:

1. As in case of using containers there is no additional operating system on each guest, it speeds up comparing with the hypervisors. All of the hypervisors use the hardware support (e.g., VT-x, VT-d, AMD-V) and unlikely to improve productivity in the future.

   The article [22] investigated the question of efficiency comparing LXC and Docker containers, KVM hypervisor and non-virtual system performance. The performance characteristics of CPU, memory, block I/O, and network studied separately. In [8] authors studied the same question, but network performance in different modes studied in more details. As a result, the following conclusions can be pointed out:

   1. Performance calculations for the processor and memory access (both random access and serial) hardly depends on the type of virtualization and slows down no more than 2-3%.
   2. Block I/O performance for containers is not so different from the performance without virtualization. However, for KVM the serial access is almost the same, while the random access is two times slower, comparing to native system, because a single reading operation request more CPU cycles.
   3. For the network performance analysis, the results vary greatly. For example, when using NAT in Docker and KVM results in doubling RTT (*round trip time*), comparing with native performance for both types of traffic: TCP and UDP (it is worth saying that KVM was slightly faster than Docker).

      For hypervisors (KVM) exist acceleration technologies[17], which excludes overhead on sending stage, but when receiving the traffic overhead is twice bigger comparing with containers. Docker overheads was about 20% comparing to the non-virtual system, both when sending and receiving TCP traffic.

   Measuring each characteristic separately is not a valid solution for performance comparison, because in this case, the overhead based on

[17] Technology is based on bypassing hypervisor and using special drivers.

the use of the operating system kernel and its context changes is not considered. This is why, in article [8] authors investigated performance of databases MySQL and Redis (data structure server). In case of MySQL, the overhead for containers equals 2%, while for the KVM - 40%. In the case of Redis, performance problems appeared primarily for network communication: for Docker with NAT took place additional load on CPU, and KVM increased the delay. As a result, the number of requests processed on the KVM were 10% less, and for Docker 20% less, compared with non-virtual system.

2. Using OS-level virtualization it is easy to implement sharing of resources, such as memory and block I/O (for example, "shared memory" or "AUFS"), while the hypervisor is almost incapable of it. ("shared memory" technology in OpenVZ containers even enables to run single instance of binary executable for many containers running the same program)

   It also allows running multiple containers in a short period of time, as they can use a single copy of the file system.

3. Containers are more flexible in the redistribution of resources on the fly.

6.6 Containers in modern world

Containers in a networking world can be viewed from the perspective of two different concepts: the concept of NFV (Network Function Virtualization) (i.e., how containers are suitable for the implementation of network function virtualization) and the concept of network modeling (for testing and modeling). Containers also can be considered in terms of the opportunities that they provide for ordinary applications using network interfaces.

Usage examples for containers for NFV concept can be:

1. Transfer of services implementation provided by operators for the companies and their subsidiaries from routers into operator’s data centers, and creating channel between router and data center (examples of services can be firewall, NAT, DPI ...).
2. Transfer services implementation provided for individual people into operator’s data center (example of service can be filtering "adult content").

Containers in the simulation can be used to research the network as a whole (for example, the speed of information expansion through given topology (either a worm or some routine information on the network), or check the protocol convergence) or simulate the network for test object (for example, SDN controller or CDN ([5])).

Large network model usually runs many containers of the same type, and it is very important to use such file systems as AUFS, because it saves a lot of space on file system, and significantly increase the speed of model deployment. It is also very important to be able to control the load on the server when creating a model and detect the presence of overloads, distorting the experiment.

When using a container for NFV implementation it is very important to be able to attach the container to a processor core, to minimize overhead of context switching. In addition, it should be possible to give access to the network card driver, because it is important to use different hardware accelerations, like DPDK.

In article [3] authors says that the simulation system NPS can be used for studying problems such as spread of malicious software on the network, the routing protocols convergence, testing network performance (maximum throughput of a certain node, overall level of delays and losses) and development of web applications.

For common applications, it is required the ability to run under containers without any changes to its functionality.

7 Container-based modelling

7.1 Examples of existing modelling systems

There are many network emulators: Mininet, Mininet-Hifi, vEmulab, PRIME, NetKit, Trellis, CORE, Estinet (commercial example), OMNeT++. There are also some add-ons that complement some systems, allowing creation of distributed network models: MaxiNet, NPS. There are specialized modelling systems, for example, RINSE ([1]), which purpose is to provide a model for the network security training.

Mininet is an opensource software, faster than most of its counterparts, such as vEmulab ([4]) and has the necessary capabilities, what makes it quite popular.

For every emulated host Mininet creates a separate container in which defines network namespace in such a manner that container has network interface connected to some software router (for example, Open vSwitch (there may be several on a single host)). Mininet makes no guarantees about the performance,

unlike vEmulab, which can limit the bandwidth speed. Thus, Mininet uses a single container opportunity: network namespace.

PRIME focuses on modeling within multiprocessor machines with shared or distributed memory ([5]). This system was successfully used to simulate the network with 45 000 nodes for CoralCDN testing.

Mininet-Hifi (high level of confidence) is based on Mininet and additionally provide performance limitation, resources preparation and performance monitoring, enabling the control of throughput performance, queue performance and delays ([4]).

Maxinet and NPS *(Network Simulator Prototyping*) represent a system which using Mininet create a distributed network between multiple independent hosts. This allows increasing the number of simulated nodes up to thousands. Maxinet does not use OpenVSwitch as a router, because it poorly works on hosts with multiple interfaces ([6]).

NPS system allows connect and configure each instance of Mininet running on the cluster (Cluster Node) using Supervisor Console. For traffic management within and between the clusters used OpenFlow controllers ([3]).

There have been successful attempts to construct a distributed system based on vEmulab using as containers FreeBSD jails. However, not all network emulators are suitable for the construction of distributed models: for example, Estinet design is difficult to be scaled ([6]).

7.2 Basic requirements of modelling systems to lightweight virtualization[18]

Such modeling systems as vEmulab, Mininet, NPS allows virtualize host, router and network, maintaining the transparency for the application, performance accuracy, possibility of interactive work, reproducibility of the experiment and the efficient use of resources ([3], [9]).

Simulation requires following features:

1. The model performance have to match the performance of today's networks. In the first place it is needed for network applications which take into account network latency.
2. Flexibility of simulation system (creation of a wide variety of topologies, and use of various applications and simulated subsystems)

[18] Paragraph uses information based on the articles [3], [4], [6], [9]

3. The ability to scale modeling system under several computer systems. The increase of the modeled network should linearly depend on the number of computer systems. Must be solved the problem of irregular delays between nodes on single host system and on different systems (difference must be not significant or it is necessary to take steps to straighten time between the virtual nodes ([6]))
4. Low model cost, what means high performance of the model, what means overhead minimization.
5. Simulation reproducibility (this can be ideally solved by using containers feature of “packing the environment”)
6. Application transparency from both: external and internal points of view, i.e. on one hand, the model should be provided for external application as set of hosts, on the other hand, every application running within the host should see the model as the network infrastructure.

   There should be no problem if you need to connect model to the Internet.

   It should be appreciated that applications such as tcpdump, may be particularly sensitive to the method of emulation, since they work primarily not with ip addresses but with network interfaces.
7. Load measure methods are needed for model, to be able to confirm correctness of the experiment. In addition, resources limitation methods also needed, with the aim of studying the behavior of the overloaded model node by fixing load on required nodes and connections.

   Usually load control utilities created separately from the model, but the model must be customized to be able to control the distribution of virtual nodes and communication channels on top of the computer system and must have API to communicate with load control utilities.
8. The ability to run a large number identical model parts (important limitations, such as the maximum number of virtual interfaces, the maximum number of processes ... in Linux systems must be big enough not to be reached).
9. The possibility of interactive interference in the model.

Requirements unsupported by container’s implementation because of their absence or imperfections:

1. There is no way to limit the bandwidth of the virtual interface, provided for container. Also in case where a single host runs many containers,

without possibility of correlation processor cores and containers, we have to set CPU priorities, therefore there is no means to fix performance of each container.

2. In real world systems is already approaching the limits of Linux systems concerning number of possible resources, such as virtual network interfaces or number of processes.
3. There are problems with the statistics collection regarding memory, CPU and Block I/O usage. And if the overload is created by one single container, it is hard to identify it. It leads to the problem of DoS (*Denial of Service*) prevention, in case one of containers uses too many resources.

7.3 Lightweight virtualization capabilities of controlling resource allocation

There are two principal ways of controlling resources in terms of modeling: the first way is to limit the resources of each container using its built-in mechanisms (this is done via cgroups, however, this approach is not self-sufficient), another way is to monitor the load degree on the computer system and run within one host as many containers as it can handle without overload. For realization of the second approach, must be created an additional tool for monitoring and automatic change of model, depending on ongoing situation. Just remember about "Heisenberg uncertainty principle", which in this case means that detailed monitoring can influence the system and decrease its productivity[19]. Depending on the model, overload while running the experiment can result in a failed experiment or the experiment can be considered uncorrupted if the overloaded computer node will be migrated immediately.

In paper [4], it has been empirically determined that CPU idle index can be used as a good criteria to look after computing nodes utilization. According to the authors of paper [3], the LXC container-based network simulation can give high precision if there is no unplanned overloaded model elements.

8 Conclusion

Containers in the past few years have been actively used in the area of lightweight virtualization (e.g. NFV), and simulation. Their abilities (flexibility, productivity, self-sufficiency, low overhead) proved to be very useful for network

[19] In paper [7] were given specific numbers: system measurements every 0.001 seconds significantly reduces its performance, however measurements every 0.01 seconds were almost insignificant.

simulation. However, simulation requires not only virtualization (i.e. control) of network in operating system, but also many other resources: CPU, memory, block I/O, networking. Not all desired features are implemented in containers, for example, they lacks the ability to limit control of CPU resource, there are problems with security and not enough capabilities for device separation between namespaces.

However, containers have its benefits. They allow sharing resources between multiple container instances in a manner that is unavailable for hypervisor-based virtualization (for example, sharing of file system or virtual memory). It allows reducing costs per container by creating a large number of similar containers that is often happening during network simulation and in real world. As a result, it leads for some considerable accelerations comparing to traditional hypervisors.

Management systems are created for lightweight virtualization (e.g. Docker), however they still have some drawbacks (for example, orchestration limitations), which probably will be fixed in the future.

Thus, lightweight virtualization capabilities are very well suited for network modeling, where it is needed to create many similar nodes.

The goal of the essay has been achieved, after discussing of different benefits and drawbacks of lightweight virtualization, which compared to the functional demands of modelling systems.

9 References

1. Liljenstam M. et al. Rinse: The real-time immersive network simulation environment for network security exercises //Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on. – IEEE, 2005. – C. 119-128.
2. Im E. G. et al. Hybrid modeling for large-scale worm propagation simulations //Intelligence and Security Informatics. – Springer Berlin Heidelberg, 2006. – C. 572-577.
3. Antonenko V., Smelyanskiy R., Nikolaev A. Large scale network simulation based on hi-fi approach //Proceedings of the 2014 Summer Simulation Multiconference. – Society for Computer Simulation International, 2014. – C. 4.
4. Handigol N. et al. Reproducible network experiments using container-based emulation //Proceedings of the 8th international conference on Emerging networking experiments and technologies. – ACM, 2012. – C. 253-264.

5. Liu J., Li Y., He Y. A large-scale real-time network simulation study using prime //Winter Simulation Conference. – Winter Simulation Conference, 2009. – С. 797-806.
6. Wette P. et al. Maxinet: Distributed emulation of software-defined networks //Networking Conference, 2014 IFIP. – IEEE, 2014. – С. 1-9.
7. Jia Q., Wang Z., Stavrou A. The Heisenberg Measuring Uncertainty in Lightweight Virtualization Testbeds //CSET. – 2009.
8. Felter W. et al. An updated performance comparison of virtual machines and linux containers //technology. – 2014. – Т. 28. – С. 32.
9. Hibler M. et al. Large-scale Virtualization in the Emulab Network Testbed //USENIX Annual Technical Conference. – 2008. – С. 113-128.
10. “Cgroups” [Electronic resource]. / Paul Menage. – Access Mode: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt
11. Rosen R. Resource management: Linux kernel Namespaces and cgroups //Haifux, May. – 2013.
12. “Resource Management Guide” [Electronic resource] / Red Hat Inc. – Access Mode: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/
13. Jerome Petazzoni. Container's Anatomy. Namespaces, Cgroups and some filesystem magic // 19 Aug. – 2015
14. Boden Russell. Realizing Linux Containers (LXC). Building blocks, Underpinnings & Motivations // 11 Mar. – 2014
15. “Namespaces in operation, part 1: namespaces overview” [Electronic resource] / Michael Kerrisk. – Access Mode: https://lwn.net/Articles/531114/
16. “Device namespaces” [Electronic resource] / Jake Edge. – Access Mode: https://lwn.net/Articles/564854/
17. Reshetova E. et al. Security of OS-level virtualization technologies //Secure IT Systems. – Springer International Publishing, 2014. – С. 77-93.
18. Kumar A. S. Virtualizing Intelligent River R: A Comparative Study of Alternative Virtualization Technologies : дис. – Clemson University, 2013.
19. “Docker Docs” [Electronic resource] / Docker Inc. – Access Mode: https://docs.docker.com/
20. “Docker misconceptions” [Electronic resource] / Matt Jaynes. – Access Mode https://valdhaus.co/writings/docker-misconceptions/
21. “Feature comparison of different virtualization solutions” [Electronic resource] / OpenVZ Inc. – Access Mode: https://openvz.org/Comparison
22. Morabito R., Kjällman J., Komu M. Hypervisors vs. Lightweight Virtualization: a Performance Comparison.
23. Xavier M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments //Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. – IEEE, 2013. – С. 233-240.