An Ultra-lightweight Container that Maximizes Memory Sharing and Minimizes the Runtime Environment

Liqing Zhang

School of Electronics and Information Engineering, Tongji University, Shanghai 201804, China

zhangliqing@tongji.edu.cn

Abstract

The rise of container technology has brought about profound changes in the data center, and a large number of software has been transferred to micro-service deployment and delivery. Therefore, it is of a broad practical significance to optimize the startup, operation and maintenance of large-scale containers in massive user environment. At present, the mainstream container technology represented by Docker has achieved great success, but there is still much room for improvement in image volume and resource sharing. We comb the development process of virtualization technology, and clarify that lightweight virtualization technology is the future research direction, which is very important for data-sensitive applications. By establishing a library file sharing model, we explored the impact of the degree of sharing of library files on the maximum number of containers that can be launched. We present an ultra-lightweight container design that minimizes the container runtime environment that supports application execution by refining the granularity of operational resources. At the same time, we extract the library files and the executable binary files into a single layer, which realizes the maximum sharing of the host's memory resources among containers. Then, according to the above scheme, we implement an ultra-lightweight container management engine: REG (runtime environment generation), and a REG-based workflow is defined. Finally, we carried out a series of comparative experiments on mirror volume, startup speed, memory usage, container startup storm, etc. , and verified the effectiveness of the proposed method in the large-scale container environment.

Keywords

Container, Cloudware, lightweight, runtime environment, resource sharing.

1. Introduction

In recent years, container-based infrastructure is increasingly affecting the various service models of cloud computing. At the IaaS (Infrastructure as a Service) level, container technology abstracts out the host operating system, enabling users to launch new containers with less time delay and without worrying about the underlying architecture during the expansion process. Amazon's ECS[1] (EC2 Container Service) is an example of such an application, a highly scalable, high-performance container management service that supports the industry's mainstream container implementation. At the PaaS (Platform as a Service) level, Google's App Engine[2] service also uses container technology to provide users with a platform to develop and deposit web applications. At the SaaS level, the cloudware proposed by Dong Guo[3] and et al is based on the lightweight features of container, which provides a new idea for transferring traditional software to cloud.

Containers have unparalleled nature of lightness compared with virtual machines. Containers can be created or deleted within seconds, providing more flexibility in the dynamic expansion of your applications. Modern container managers (such as LXD[4], Docker[5], Kubernetes[6], etc.) provide a more convenient way for users to use containers by automatically scheduling, expanding, and storing containers. At the same time, the combination of container technology and microservices architecture in software development further magnifies the advantages of the latter. Microservices encourage software developers to decouple the entire software into smaller functional segments that can cope with external failures. The container further extends this de-coupling ability to separate software from the underlying hardware. The result of this approach is that applications can be created faster and are easier to maintain while achieving higher quality.

With the popularity of microservices architecture in software design, more and more containers are used as an infrastructure to host applications, but the problems of existing container technologies are gradually emerging. At present, the design of the container has a strong operating system form. Although the operating system kernel is shared, it still encapsulates the functions of the entire operating system which are not used when running a specific service, resulting in redundancy of container image content. Besides, current mainstream container manager adopts the strategy of pre-building the runtime environment, so the sharing of the same runtime dependent files (mainly including executable binary files and shared library files) in multiple containers is not thoroughly supported in memory which results in memory data redundancy.

The efficient use of memory resources and storage resources is critical to cloud computing providers. A typical example is a serverless solution that has become popular in recent years (such as Amazon's AWS Lambda), which encourages application developers to implement their services as a combination of stateless functions, written in a predefined programming language. And triggered by a predefined event(such as a user request or a database change). This has led cloud computing vendors to instantiate a large number of containers that load similar runtime environments and the same programming language libraries. Another scenario is that in current online programming education, the use of containers to provide users with a specific programming environment has become mainstream, and certain specific container instances are opened or destroyed in large numbers. Therefore, it is not only beneficial but also a realistic requirement to realize the sharing of files in memory when running between multiple containers.

Main contributions of this paper are:

(1) By establishing a mathematical model of library file sharing, the effect of the sharing of library files on the number of container startups is studied.

(2) We propose a general-purpose ultra-lightweight container design that minimizes the container runtime environment by refining the granularity of operational resources

(3) We have clearly defined the life cycle of this kind of ultra-lightweight container, proposed an ultra-lightweight container state model, and designed a method for maximizing sharing of host memory resources.

(4) Based on the above method, we implemented the ultra-lightweight container management engine Reg, and carried out comparative experiments on mirror volume, startup speed, memory usage, etc., verifying the effectiveness of the proposed method in a large-scale container environment.

2. Related Work

In the past few decades, virtualization has been a major topic in academic research and has been widely used. It shares the computing and storage capabilities of the host by dividing physical resources among multiple operating systems. Virtualization is the cornerstone of cloud computing[7]. People's research on virtualization can be divided into two main types: hypervisor-based virtualization and container-based virtualization[8], the essence is to make the most of the physical resources of the computer. This part briefly reviews the two main types and introduces some new virtualization methods that have emerged in recent years in order to explore the future development of virtualization.

In 1959, Christopher S[9] proposed the concept of time-multiplexed in large computers, opening the history of virtualization research. In 1964, IBM launched a project called "Building a CP/CMS System"[10], which first proposed the concept of a virtual machine and developed a virtual machine device system370. Since then, more and more virtual machine management programs have been Developed. The hypervisor runs at the hardware level and therefore supports running a stand-alone virtual machine that is isolated from the host system. Because the hypervisor isolates the virtual machine from the underlying host system, the operating system running in both can be quite different. Popek G J [11] and others divided the hypervisor into two categories.

Type 1: Bare metal architecture (hypervisor runs directly on top of physical hard-ware)

Type 2: Hosting architecture (hypervisor runs on the host operating system)

Regardless of the architecture, a complete operating system is always encapsulated in the virtual machine, as shown in Figure 1(a). This means that the virtual machine's image will be very large, and the simulation of the virtual hardware device will also lead to more performance overhead.

Container-based virtualization provides different levels of abstraction in terms of virtualization and isolation compared to hypervisors. The hypervisor abstracts computer hardware resources, which can result in virtualization overhead for hardware and virtual device drivers, and typically runs a full operating system (such as Linux, Windows) in each virtual machine instance. In contrast, containers implement process isolation at the operating system level, which avoids this overhead. These containers share the operating system kernel running on the same underlying host, and one or more processes can run inside each container. Therefore, the container is considered a lightweight alternative to hypervisor-based virtualization. Figure 1(b) shows a container-based virtualization architecture.

Multiple containers use shared kernels and system library files. So one of the ad-vantages of a container-based solution is the ability to implement higher-density virtualized instances, while the container's disk image is smaller than a hypervisor-based solution. Container technology also has some obvious drawbacks. Because of the dependency on the host kernel, containers based on different kernel implementations are not compatible (for example, Windows containers cannot run on Linux hosts). In addition, because the host kernel needs to be exposed to the container, the container does not completely isolate the resources as the hypervisor, which has certain security problems for the multi-tenant scenario. There are several implementations for container-based virtualization, such as Linux-VServer[12], OpenVZ [23], LXD, Docker, and Rocket[13]. The Linux-based container is mainly implemented by the control group[14] and the namespace[15] in the operating system kernel mechanism. The former is mainly responsible for providing resource management functions between multiple process groups, and the latter is mainly responsible for isolating a set of private physical resources for the container. As a high-level container management platform, Docker has been widely sought after in recent years for its convenient user interface.

The work of T Harter[16] et al shows that by using the federated file system storage driver, containers based on the same-origin Docker image can share part of the library files and executable binaries, thereby making fuller use of memory resources. Ferreira [17] and others have proved that the sharing of library files between multiple containers can significantly improve memory utilization by constructing mathematical models.

Although virtual machine monitor based virtualization and container-based virtualization have become the physical infrastructure in cloud computing, both add a new layer of abstraction to the highly layered software stack. Some scholars have reservations about this: Is it really necessary to add an indirect and abstract layer every few years to put the application code on a multitude of abstraction layers? Unikernel is the solution to this standpoint. Unikernel is a dedicated single address space machine image built using the library operating system[18] [19]. Developers need to select the minimum set of operating system libraries their applications need to run, and then compile these libraries with the application code and configuration code to form a closed, purpose-specific image. The basic architecture is shown in Figure 1. c) shown.

Unikernal can run directly on hardware or hypervisors without the intervention of an intermediate operating system. With the development of cloud computing and virtualization technology, the idea of building a single-tasking operating system has also received enough attention, and many scholars are constantly looking for application scenarios in the field of cloud computing. A. Madhavapeddy[20] and others gave a specific implementation of Unikernal in 2013 and called it Mirage OS.

Subsequently, A. Bratterud[21] and others improved on Madhavapeddy's work to build Include-OS, which has a smaller mirror volume and higher resource utilization. Another implementation is OSv[22], which is an operating system designed specifically for cloud computing that also runs a single application. But unlike Mirage OS, OSv is designed to run on top of hypervisors (such as KVM, Xen, VirtualBox, and VMware), so it achieves the isolation advantage of a hypervisor-based system while avoiding virtualization. In general, OSv is primarily used to run Java-based applications as well as Linux-based C/C++ applications.

Figure 1 shows the difference in architecture among virtual machine, docker container, Unikernal, and the ultra-lightweight container engine Reg proposed in this article.

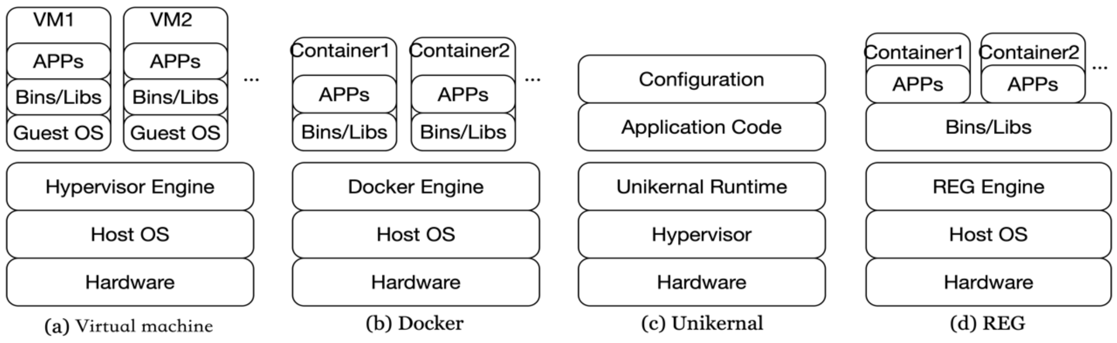


Fig. 1. Comparison of four virtualization methods

3. Design Of The Ultra-Lightweight Container

Inspired by the conclusions obtained in the previous section, we present a new approach to building an ultra-lightweight container environment.

3.1 Ultra-lightweight container runtime state model

During the execution of the ultra-lightweight container, its own state changes constantly. We define its life cycle as the conversion process of the following states, and each state needs to meet certain conditional constraints:

Generating. The process of obtaining runtime dependent files according to the declared runtime environment.

Generated. This state first needs to meet the runtime environment completeness, that is, the local repository already contains all the necessary runtime dependencies, has the ability to generate the runtime environment declared in the code, and the relevant runtime dependent files have been extracted to the container space.

Running. Runtime dependent files are loaded into memory, the runtime environment has been generated, and the application is running. To maintain the lightweight nature of the container, this state should satisfy the uniqueness of the same runtime dependent file in memory.

Inserting. The process of inserting a new component into a container, and this process is guaranteed to not corrupt the software runtime environment.

Stopped. Processes in the container are in a terminated state, and the runtime environment have been destroyed. The difference from the Generated state is that the state retains the files generated by the process running.

Removed. The container space has been destroyed and the relevant runtime dependencies have been deleted.

The transition between these states is shown in Figure 2:

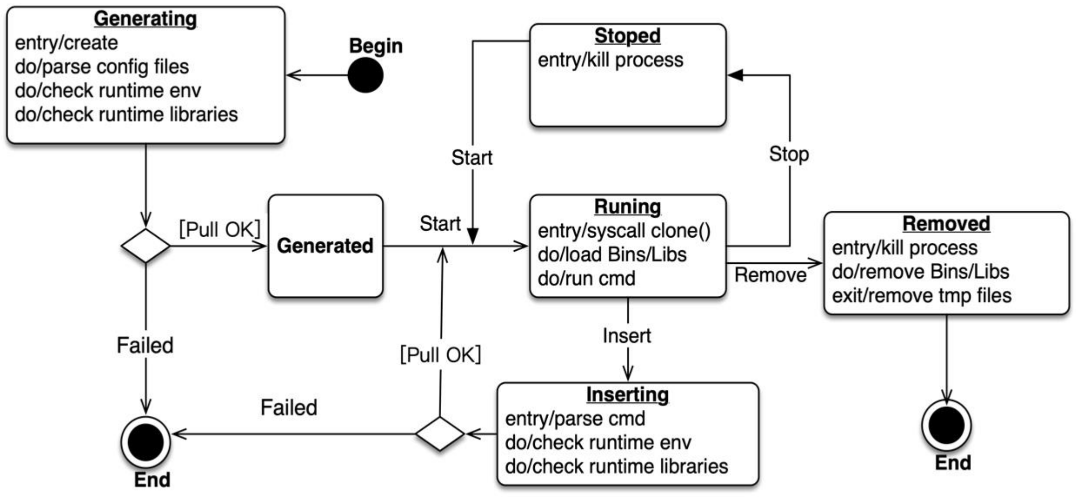


Fig. 2. Ultra-lightweight container runtime state model

3.2 Maximize memory sharing and minimize runtime environment

In traditional container design, the role of image is to provide a fixed runtime environment for the container. Because people can not determine in advance what kind of application will run in the container, the image usually contains a common set of basic programs during the packaging process. However, these basic programs are often not used in production environment, so such a runtime environment not only causes waste of memory and disk space, but also poses more security risks to the container.

At the same time, mirror-based container design is also the source of redundancy for library files in memory. Marco Cello et al. [17] mentioned that the redundancy of runtime dependent files in memory is a main factor affecting memory utilization. The industry mainstream container manager Docker only supports library file sharing for containers based on homologous images under specific storage drivers, but that cannot be implemented for non-homologous images or other storage drivers. This causes even the same runtime dependent file to be loaded into memory many times and results in a waste of memory space.

The ultra-lightweight container construction method proposed in this paper abandons the concept of image, we emphasize that the code itself as the image itself. When a user writes the application, he or she only needs to specify the runtime environment that the application depends on in the configuration file(such as Python, NodeJS, JVM, etc.). Once the container is executed, the container engine parses the configuration file and dynamically generates a user-specified runtime environment. During the process of loading container resources in memory, the granularity of resources is refined from the original fixed mirror layer to specific executable binary files and shared library files, and only the minimum runtime environment required for application execution is generated. Meanwhile, since all the dependent files are uniformly mapped from the native file system to the container space, the same files that multiple containers depend on are stored on the same index node. So we can guarantee the uniqueness of the same executable binary and shared library in memory, thereby maximizing the sharing of memory resources. What has to be done here is to ensure resource isolation between containers and local library files cannot be tampered by malicious containers. The prototype system in Chapter 4 gives a concrete implementation method. Figure 3 shows the basic architecture of the ultra-lightweight container proposed in this paper. It can be seen that each container contains only the most basic runtime environment that it needs.

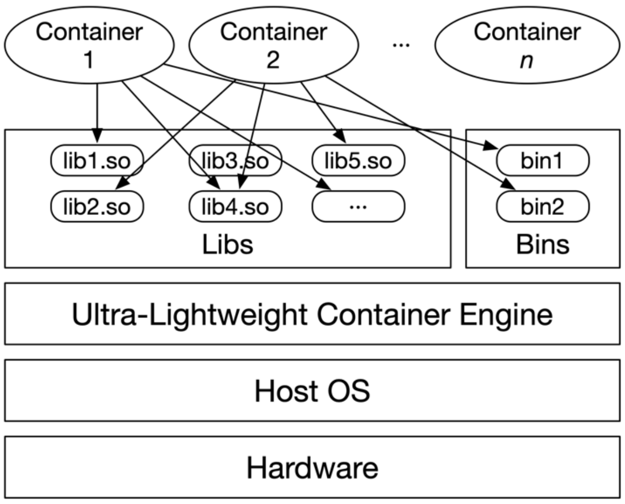


Fig. 3. Ultra-lightweight container design framework

4. Implementation And Case Study

As mentioned in the related work, the traditional container is isolated on the layer of operating system. The kernel is shared among containers, only basic operating system libraries, applications and their dependencies are encapsulated in the respective spaces. Advantages of this design pattern are eliminating the redundancy of the operating system kernel and enabling more container instances with fixed resources. Based on the ultra-lightweight container design proposed above, we build a container management engine called Reg to further deepen the idea of resource sharing. Only the respective application code is hold in container space, and Reg always ensuring that the same dependencies of multiple containers exist only copy in memory. The structural differences between the Reg container management engine and the current mainstream container management engine are shown in Figure 4.

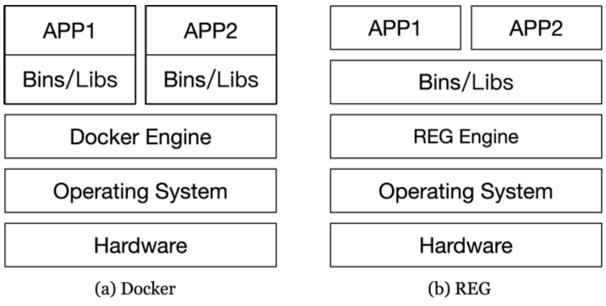


Fig. 4. Architecture differences on two type of containers

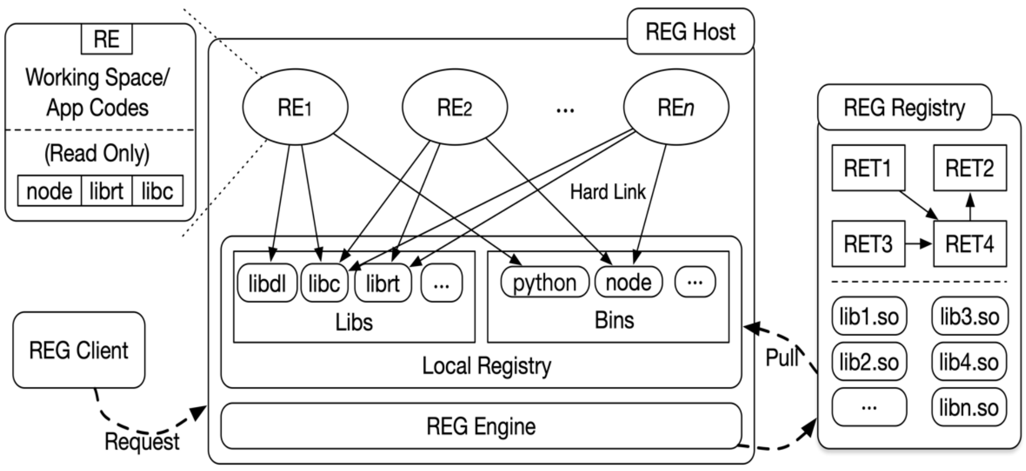


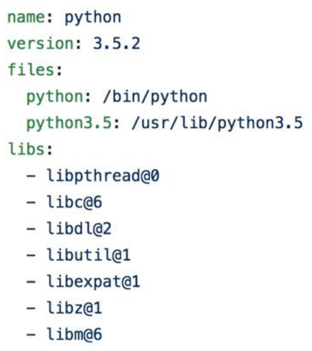
Fig. 5. Reg container management engine architecture

4.1 Architecture of Reg

As an ultra-lightweight container management engine, Reg is implemented as a simple C/S architecture. Users can establish communication with server through a client agent. The overall architecture of Reg is shown in Figure 5, which mainly includes Reg Client, Reg Host, and Remote Registry

The Reg Client is the best way for users to establish communication with the Reg Engine. Users can initiate a management request for the ultra-lightweight container through the client. When receiving request, the Reg Engine parses user's command and performs actual operations. Supported user instructions mainly include container generation, remove, start, stop, install, and uninstall.

Reg Registry is a runtime environment template (RET) repository that is deployed on the public network and is responsible for the storage and distribution of runtime templates. The role of the runtime template can be compared to the image of the Docker container. Docker generates a container based on an image, and Reg obtains the corresponding binary files and library files according to the runtime template then generates the user-specified runtime environment. As shown in Figure 6, there are two forms of files in Reg Registry: RET template file and shared library (.so) file. RET file contains the executable binary itself and the dependency description for corresponding library files. A typical RET configuration file is shown in Figure 6, which indicates the name and version of an executable and the shared library file on which the executable depends.



```
name: python
version: 3.5.2
files:
  python: /bin/python
  python3.5: /usr/lib/python3.5
libs:
  - libpthread@0
  - libc@6
  - libdl@2
  - libutil@1
  - libexpat@1
  - libz@1
  - libm@6
```

Fig. 6. RET file structure

As the main part of the architecture, Reg Host has the server-side function and is capable of receiving requests from clients. All tasks of Reg Host are performed by the engine, including parsing user requests, pulling runtime dependencies from remote repositories, container generation and lifecycle management. When a new container is created, Reg Engine copies library files from the local repository into the container space as hard links, which ensures that the same library files in each container always have the same index node number, that is, these files can be shared by multiple containers when loading into the memory.

Considering security, containers generated by Reg use OverlayFS as the file system. Executable binary and library files that make up the runtime environment are placed in the underlying directory. The read-only feature of the underlying directory implemented by OverlayFS ensures that the local repository will not be polluted. In terms of isolation, Reg uses a set of system calls of Linux Namespace to isolate the mount point, process number, and inter-process communication.

4.2 Reg instance running process

According to the implementation of Reg, we can describe the specific process of a container startup as below:

Parsing parameter. When receiving the command from the client, the Reg engine will first parsing the parameters to obtain the runtime template type and the running command. For example, 'reg run –t python@3.5.2 -c /bin/python' specifies a python with a runtime template type of 3.5.2. The command to be executed after the container is generated is '/bin/python'.

Generating directory structure. Reg Engine will generate OverlayFS file system structure in the .reg folder of the project directory, including lower, upper, work and mount folders.

Obtaining runtime template. If the local repository does not have a user-specified runtime template, Reg Engine will pull it from the public registry. Since runtime templates sometimes rely on other templates, this process is usually performed recursively.

Linking and mounting files. After confirming that the executable binary and shared library files that make up the runtime environment are present in the local repository, Reg Engine will hard link these files to the lower folder that just created. Then mounting the directory structure created in step (2) in form of OverlayFS, and mounting the user's project file to the mount folder.

Creating an isolated space. In terms of container space isolation, Reg uses the environment isolation method provided by the Linux kernel (just as Docker), mainly containing mount point, process number and the inter-process communication. Considering the need to maintain lightweight nature, Reg does not isolate the network, but uses the host network directly. At the same time, system call pivot_root is used to make the mount directory the root directory in the container, thus ensuring the isolation of the file system.

Mounting the /proc directory. /proc is a real-time api to the kernel, it can provide kernel information for Reg.

Executing command. At this time, the container already has the runtime environment file that applications depend on, Reg Engine can now execute the real command specified by user.

5. Performance Evaluation

In this section, Reg is compared with the Docker container management engine. The main evaluation aspects include the image file size, the container startup time and memory usage of the container during runtime. The hardware configuration for these experiments are listed as below:

Processor: Intel Core i7-8709G (8M cache, quad core, 8 threads);

Memory: 64G LPDDR3;

Hard disk: 500G PCIe NVMe high speed solid state drive;

Operating system: Ubuntu 16.04 (64-bit).

The software and version of the experiment are shown in Table 1.

Table 1. Software And Release Notes

Name	Version	Storage Drivers
Docker	17.06	OverlayFS
Reg	0.1	OverlayFS

5.1 Image volume

Image is a concept in traditional container technology that refers to the file structure needed to support a container to run and save container information. Although Reg does not have the concept of image, for the convenience of description, we still refer the dependent files that dynamically pulled during container generation process as its image file. We select five representative and widely used software as build objects, including Python, NodeJS, open source database MySQL, commonly used web server Nginx and Tomcat. We use Docker and Reg to build the same version of the container image for the above software respectively, and the image size comparison between the two is shown in Figure 7.

It can be seen that the five container images built by Reg are smaller than the same image built by Docker due to the elimination of redundant files outside the runtime environment. Compared to the size of the Docker image, each of the five images built by Reg reduces the disk footprint by an average of 18%. The smaller image size helps save network resources and enables image files to be quickly distributed among multiple hosts.

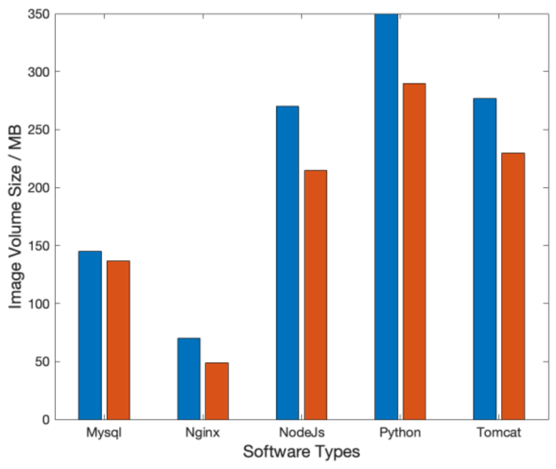


Fig. 7. Image volume comparison

5.2 Startup time

The startup time of the container can be calculated by comparing the system timestamp before and after the container is started. We independently launch 100 instances of the same type of Docker container and Reg container in the same environment, and the startup time of each container instance is recorded. The result is shown in Figure 8. The average startup time of the Docker container is 405ms, and the Reg container is 373ms, which is a 7.9% reduction compared to Docker.

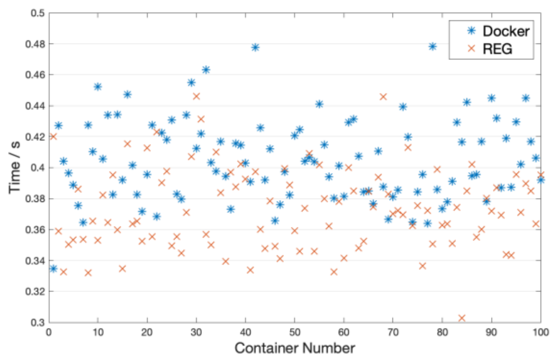


Fig. 8. Container startup time comparison

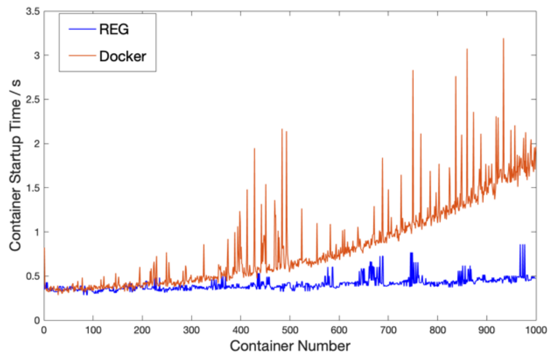


Fig. 9. Start storm performance comparison

In a real production environment, there are often scenes in which a large number of container instances are started in batches at a certain time, so-called startup storms. We used the above image to continuously launch 1000 container instances for each of the two container engines, and the startup time of each container instance is recorded as well. The result is shown in Figure 9. The initial start time of the Docker container was 357ms, but as the container instance increased, the startup time of the single instance increased significantly, extremely, the 1000th instance reached 1964ms. The startup time of Reg is also increased as we lunch more instance, but the overall curve is relatively stable, the start of the last container only took 478ms. Under these experimental conditions, it's easy to calculate that the average startup time of Docker is 827ms, and Reg is 405ms, which is a 51% reduction compared to the former.

5.3 Memory footprint

Effective use of memory resources is critical to cloud computing vendors. As can be seen from the related work in section 2, in recent years, a major development direction of virtualization technology is to save memory resources as much as possible under the premise of ensuring computing power and isolation. In order to verify the memory usage of Reg and Docker, We use the five container images built above to run same service in the same image category separately, and record the memory usage of each container. Figure 10 shows the result of this experiment.

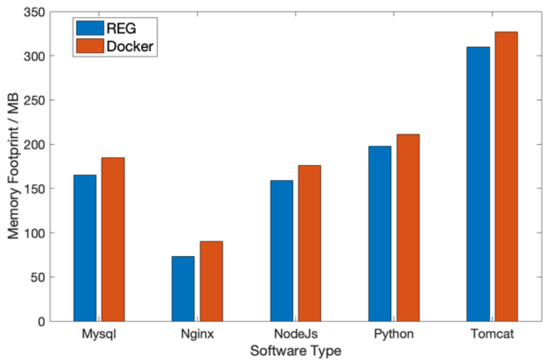


Fig. 10. Single container memory footprint comparison

In order to verify the hybrid deployment of multiple container instances, the Reg and Docker are used to start these five containers continuously in the same virtual machine environment. The specific startup sequence is shown in Figure 11, and the memory usage is recorded at five moments.

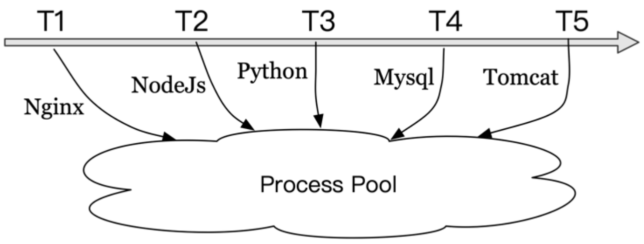


Fig. 11. Container startup sequence

For comparison, we also use a physical machine to directly run the same service as above. The memory usage at various moments in various environments is shown in Figure 12.

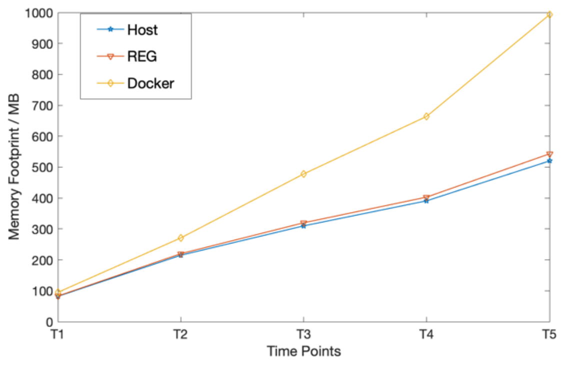


Fig.12. Changes in memory usage at different times

It can be seen that because the library file sharing of non-homologous containers haven't been implemented in Docker, the total memory occupied by the five containers during startup is approximately the same as the memory when each container is started separately, which is 47.6 % more than the direct operation on physical machine. Containers started based on Reg engine totally saved 460MB of memory compared to separate boot of the containers, and only took up 2.5% more than running directly on physical machine. This experiment verifies that Reg engine supports the sharing of library files between non-homologous images. This feature also makes the memory space utilization of Reg almost close to physical machines.

In summary, the ultra-lightweight container solution proposed and implemented in this paper has has a great improvement in image volume (average reduction of 18%), startup time (average 51% faster), and memory usage (average reduction of 47.6%), which fully demonstrates the effectiveness of the proposed method.

6. Conclusion

This paper proposes an ultra-lightweight container construction method, and designs a new container-based workflow based on the abstracted container runtime state model. Then, based on the above construction method, we implemented the Reg ultra-lightweight container management engine, which enables the maximum sharing of resources between containers, thus minimizing the runtime environment. Finally, it is verified by experiments that Reg has improved in terms of file size, startup time and memory utilization compared with Docker.

In this paper, we haven't verified the performance of Reg in terms of disk I/O, CPU utilization, etc, and future work will be supplemented. At the same time, opti-mizing the storage and distribution of runtime environment templates, and the tam-per-proofing of shared libraries and executable binaries are also the focus of future work.

References

[1] Amazon Elastic Container Service,https://aws.amazon.com/cn/ecs/,last accessed 2018/10/10.

[2] Google App Engine, https://appengine.google.com/, last accessed 2018/10/10.

[3] GUO D,WANG W,ZAHGN J X,et al.: Towards Cloudware Paradigm for Cloud. In: IEEE International Conference on Cloud Computing, IEEE, San Francisco(2017).

[4]What's LXD?, https://linuxcontainers.org/lxd/, 2018/10/10.

[5] Introduction about Docker, https://www.docker.com/what-docker, last accessed 2018/10/10.

[6] Introduction about Kubernetes, https://kubernetes.io/, last accessed 2018/10/10.

[7] Oludele A, Ogu E C, Shade K ', et al.: On the Evolution of Virtualization and Cloud Computing: A Review[J]. Journal of Computer Sciences & Applications 2(2), 40-43 (2014).

[8] Morabito R, Komu M.: Hypervisors vs. Lightweight Virtualization: A Performance Comparison, In: IEEE International Conference on Cloud Engineering, IEEE Computer Society, Tempe, AZ (2015).

[9] Strachey C.: Time sharing in large fast computers, World Computer Congress, Paris (1959).

[10] Creasy R J.: The Origin of the VM/370 Time-Sharing System[J]. IBM J Research & Development 25(5), 483-490 (1981).

[11] Popek G J, Goldberg R P.: Formal requirements for virtualizable third generation architectures. In: ACM Symposium on Operating System Principles, ACM, New York, (1973).

[12] Introduction about Linux-VServer, http://linux-vserver.org/, last accessed 2018/10/10.

[13] CoreOS is building a container runtime, rkt, http://coreos.com/blog/rocket/, last accessed 2018/10/10.

[14] Paul Menage et al. CGROUPS,https://www.kernel.org/doc/Documentation/cgroup-v1/ cgroups. txt, last accessed 2018/10/10.

[15]Namespaces - overview of Linux namespaces, http://man7.org/linux/man-pages/man7 /namespaces. 7.html,last accessed 2018/10/10.

[16] Harter T, Salmon B, Liu R, Arpaci-Dusseau AC, et al. Slacker.: fast distribution with lazy docker containers. In: Usenix Conference on File and Storage Technologies, USENIX Association, Santa Clara (2016).

[17] Ferreira J B, Cello M, Iglesias J O.: More Sharing, More Benefits? A Study of Library Sharing in Container-Based Infrastructures. Lecture Notes in Computer Science, 358–371 (2017).

[18] Engler D R, Kaashoek M F, O'Toole J. Exokernel.: an operating system architecture for application-level resource management. In: ACM Symposium on Operating System Principles, ACM, Copper Mountain Resort (1995).
[19] What are unikernels?, http://unikernel.org/, last accessed 2018/10/10.
[20] Anil Madhavapeddy , David J. Scott, Unikernels: Rise of the Virtual Library Operating System.Queue, v.11 n.11, (2013).
[21] Bratterud A, Walla A A, Haugerud H, et al.: IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In: IEEE, International Conference on Cloud Computing Technology and Science, IEEE, Luxembourg (2016).
[22] OSv – the operating system designed for the cloud, http://osv.io/, last accessed 2018/10/10.
[23] Introduction about OpenVZ, https://openvz.org/Main_Page, last accessed 2018/10/10.