**Computer Science and Application** 计算机科学与应用, 2021, 11(1), 101-112
Published Online January 2021 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2021.111012

Hans汉斯

# 大规模在线实训环境下的快速容器并发启动研究

**毕枫林[1*]，田春岐[1]，王　伟[2#]，唐　骏[1]**

[1]同济大学，电子与信息工程学院，上海
[2]华东师范大学，数据科学与工程学院，上海
Email: 515186469@qq.com, #wwang@dase.ecnu.edu.cn

收稿日期：2020年12月25日；录用日期：2021年1月19日；发布日期：2021年1月26日

## 摘　要

**随着在线教育业务的快速增长，需要越来越多的技术来支撑在线教育的不同场景。本文进行了在大规模在线实训环境下的虚拟化技术的可行性探索，得出了容器技术符合在线实训环境的需求的结果，并且得出以下结论：1) 确定数学模型来选择与Docker通信的最佳方式；2) 对大规模在线实训环境下的容器并发启动场景进行了研究，总结出了CPU是影响容器并发启动时延的主要因素；3) 容器镜像大小对容器并发启动影响较小；4) 容器并发启动性能消耗主要阶段在容器启动阶段。**

## 关键词

**Docker，虚拟化技术，并发，容器，通信方式**

# Research on Container Concurrent Initiation under Large-Scale Online Training Environment

**Fenglin Bi[1*], Chunqi Tian[1], Wei Wang[2#], Jun Tang[1]**

[1]School of Electronics and Information Engineering, Tongji University, Shanghai
[2]School of Data Science & Engineering, East China Normal University, Shanghai
Email: 515186469@qq.com, #wwang@dase.ecnu.edu.cn

Received: Dec. 25th, 2020; accepted: Jan. 19th, 2021; published: Jan. 26th, 2021

*第一作者。
#通讯作者。

文章引用: 毕枫林, 田春岐, 王伟, 唐骏. 大规模在线实训环境下的快速容器并发启动研究[J]. 计算机科学与应用, 2021, 11(1): 101-112. DOI: 10.12677/csa.2021.111012

## Abstract

**With the rapid growth of online education, more and more technologies are needed to support different scenarios of online education. This paper explores the feasibility of virtualization technology in large-scale online training environment, and concludes that container technology meets the requirements of online training environment. The following conclusions are drawn: 1) the mathematical model is determined to choose the best way of communication with Docker; 2) the container concurrent startup scenario under large-scale online training environment is studied, and it is concluded that CPU is the main factor affecting the container concurrent startup delay; 3) the container image size has little influence on the container concurrent startup; 4) the container concurrent startup performance consumption is mainly in the container startup stage.**

## Keywords

**Docker, Virtualization Technology, Concurrent, Container, Communication Mode**



## 1. 引言

随着容器技术越来越流行，Docker 的应用也越来越广泛，不仅仅只有用来更快地开发或者部署应用[1]，还有了其他应用场景，比如用来实现 Serverless [2]，利用 Docker 来管理交换机的存储、计算和网络资源[3]，完成高性能多任务计算[4]等等。Docker 容器由 Docker 引擎执行和控制，这与虚拟机(Virtual machine 简称 VM)的管理程序不同。由于它不包含完整的客户操作系统(Guest OS)，所以 Docker 容器比 VM 更小，启动速度更快。Docker 容器装载一个单独的根文件系统，其中包含类 Unix 操作系统的目录结构，以及运行用户应用程序所需的所有配置文件、二进制文件和库[5]。可以模拟用户进行学习的实验场景。Docker 也被应用于构建大规模在线实训平台，利用 Docker 轻量级虚拟化技术，并且拥有快速弹性伸缩的特性，能够给予用户一个完整的实训体验。

在此类场景下特点有：1) 容器生灭快，容器生存周期短。2) 容器操作并发量大。3) 容器镜像稳定，实训环境固定，容器镜像变化不频繁。4) 大量线程跟 Docker 宿主机进行(Http or Socket)通信，要求网络稳定，网络抖动小。5) 充分利用云平台特性，动态地给予用户云平台资源。6) 不进行高性能计算(HPC)。

本文主要针对以下问题进行研究：1) 确定高并发容器启动下的容器通信方式，确定一个可靠的数学模型可以进行通信方式选择。2) 确定影响容器并发启动的各个因素，并且确定容器并发启动的性能瓶颈。

## 2. 研究现状

Docker 是一个新的工具，它可以在 Linux 容器中自动部署应用程序。它提供了操作系统级虚拟化的抽象层和自动化。容器在 Linux 上本地运行，并与其他容器共享主机的内核。它运行一个独立的进程，不占用比其他任何可执行程序更多的内存，使其轻量级。Docker 本身并不是一项新技术，但它是一个高级工具，最初构建在 LXC API 之上，并提供附加功能。Docker 容器由 Docker 引擎执行和控制，这与 VM 的管理程序不同。因为它不包括一个完整的客户访问的操作系统(Guest OS) Docker 容器比 VM 更小，启动速度更快[4]。

Docker 与 VM 的不同之处：1) Docker 容器由 Docker 引擎执行和控制，这与 VM 的管理程序不同。2) 因

为它不包括一个完整的客户操作系统(Guest OS) Docker 容器比 VM 更小，启动速度更快[6]。见图 1。

针对大规模在线实训平台的场景，我们先比较了各个虚拟化方案：1) 内核虚拟机(Kernel-based Virtual Machine，简称 KVM) [7]：需要消耗的资源太大，过于重量级，不适用于快速生灭的特点。2) 容器[8]：是轻量级虚拟化的一种形式，通过对进程制造一定的限制来完成虚拟化工作，容器化的系统都运行在主机的内核上。3) unikernels [9]：硬件级虚拟化的一种，将简约的 LibOS 和目标应用程序专门化为在虚拟机监控器(Virtual Machine Monitor，简称 VMM)上运行的单一用途虚拟机，不适用于开放式场景。

通过各个方案比较确定使用容器作为该场景下的虚拟化方案。并且在 Http 代理场景下，Docker 性能要优于 KVM [10]，符合线上运行环境要求。

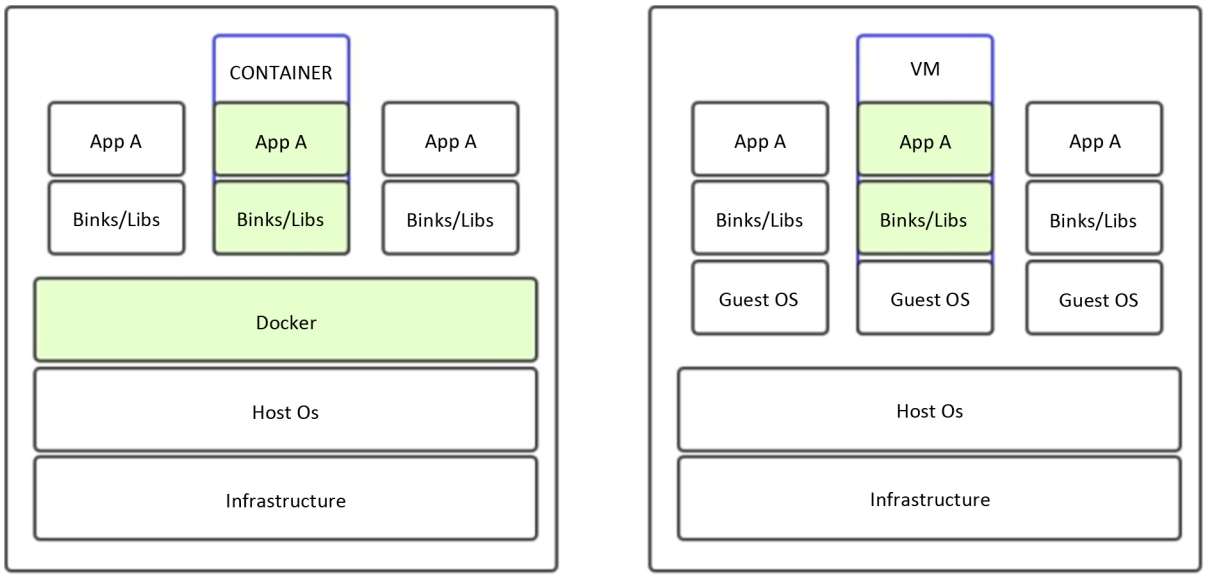


**Figure 1.** The Container architecture versus the VM architecture
**图 1.** Container 与 VM 架构对比

## 2.1. 国外研究现状

有不少人针对虚拟化技术进行过测评，文献[11]在基于 Linux 容器的 Docker 和虚拟机管理程序(Hypervisor)上构建了云环境，分析了它们的大小、启动速度和 CPU 性能。文献[12]在性能和可伸缩性方面比较了 Linux 容器和虚拟机。文献[10]在 Http 代理场景中提供了两个开源虚拟化解决方案 KVM 和 Docker 的性能分析。文献[13]提供了在 OpenStack 云平台中并发地请求多个运行实例时，对 KVM (虚拟机)、Docker (容器)和 OSv (Unikernel)的评估。

还有很多人做了容器相关的测评工作 SOCK [14]分别对容器存储，逻辑隔离，性能隔离做了测评。SLACKER [15]对容器镜像大小，执行需要多少数据，推、拉和运行图像需要多长时间等方面进行了测评。还有一些专注于 MYSQL [16]或者一些更复杂的网络应用程序，比如安装了 Mysql 的负载平衡的 Wordpress [17]，文献[18]对 Docker 容器的网络模式进行了评测。文献[5]对不同容器在 HPC 平台下的容器性能的评测。

## 2.2. 国内研究现状

国内对容器的研究也有很多，文献[19] [20]对 Docker 本身的网络架构做出了深入研究，利用多种技术和方法来对网络隔离和网络控制做出了改进。文献[21]将依赖库文件与可执行二进制文件单独抽取成层，实现了容器对主机内存资源的最大化共享，以较小的时间延时启动新的容器。国内大多数工作与对容器

资源调度有关，提供一个有效的算法模型来提高容器的资源调度，文献[22]将 Kubernetes 结合已有 Openstack 云平台，提出一种基于容器的弹性调度策略，建立了一个提高集群资源利用率的优化模型，通过对云平台各个服务器节点四种类型资源的监控和应用队列预设模板匹配，选择调度资源利用率最高的服务器。还有文献[23]提出了一种基于三次指数平滑法和时间卷积网络的云资源预测模型，根据预测值为应用及时、准确、动态地调度和分配资源。

据我们所知极少人人对容器的并发启动进行过相关的工作，而容器并发启动在很多场景下非常有必要。许多场景下都会需要容器大规模的并发启动。

## 3. 通讯方式的选择

Docker deamon 是 Docker 最核心的后台进程，他负责响应来自 Docker client 的请求，然后将这些请求翻译成系统调用完成容器管理操作。该进程会在后台启动一个 API server，负责由 Docker client 发送的请求；接收到的请求将通过 Docker deamon 分发调度，再由具体的函数来执行请求[24]。

要与 Docker daemon 通过 rest API 通信，有三种通信方式：1) Unix domain socket (简称 Unix Socket)，2) Systemd socket activation，3) TCP。其中 Systemd socket activation 的工作方式是让 systemd 守护进程代表应用程序打开监听 Socket，并且仅在连接进入时才启动它。然后将套接字交给新启动的应用程序，由它负责[25]。本质上也是 Unix Socket 通信。

为了研究高并发情境下哪种方式与 Docker 守护进程通信最佳，我们进行了多组实验。

### 3.1. 实验环境

本文测试环境使用服务器 1 台，主要是与容器通讯方式的选择和容器并发启动影响的相关因素进行研究，容器版本为 Docker18.06.2-ce，服务器配置如表 1。为了排除网络带宽限制等影响，我们采用直接请求本机的 Docker 守护进程，这样不会产生额外的消耗，使 Tcp 和 Unix Socket 结果更加真实。

**Table 1.** Experimental environment
**表 1.** 实验环境

| 组件 | 规格 |
|---|---|
| CPU | 8 CPUS x Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20 GHz |
| 内存 | 16G |
| 操作系统 | Ubuntu 16.04.6 LTS |
| Docker 版本 | 18.06.2-ce |

### 3.2. 顺序执行

我们对两种通信方式进行实验，分别顺序执行 200 次 List Images [26]方法请求 Docker deamon，时延为以向 Docker demon 发送请求开始到 Docker deamon 返回数据为止。得到的实验结果中 Unix Socket 通信响应时间的平均值为 2499.73 μs，Tcp 通信的平均响应时间为 2909.11 μs，如图 2。实验结果中 Unix Socket 与 Tcp 的传输时延区别不是非常明显，Unix domain socket 平均通信时间比 Tcp 提高了大约 10%，不过我们为了测量网络抖动，引用了变异系数来确定网络抖动情况，公式为：

$$C = \frac{\sqrt{\frac{n\sum x^2 - \left(\sum x\right)^2}{n(n-1)}}}{\frac{\sum x}{n}} \tag{1}$$

公式(1)中 $x$ 为一次请求的延时，$n$ 为请求次数。最后得出结果 Unix Socket 的变异系数为 0.002724，Tcp 的变异系数 0.004679，可见 Http 通信方式的网络抖动比 Unix Socket 通信方式高了大约一倍。

因此我们可以得到结论，在顺序启动容器时与 Docker demon 进行通信时，Unix Socket 的通信延时和网络稳定性都优于 Tcp 方式。

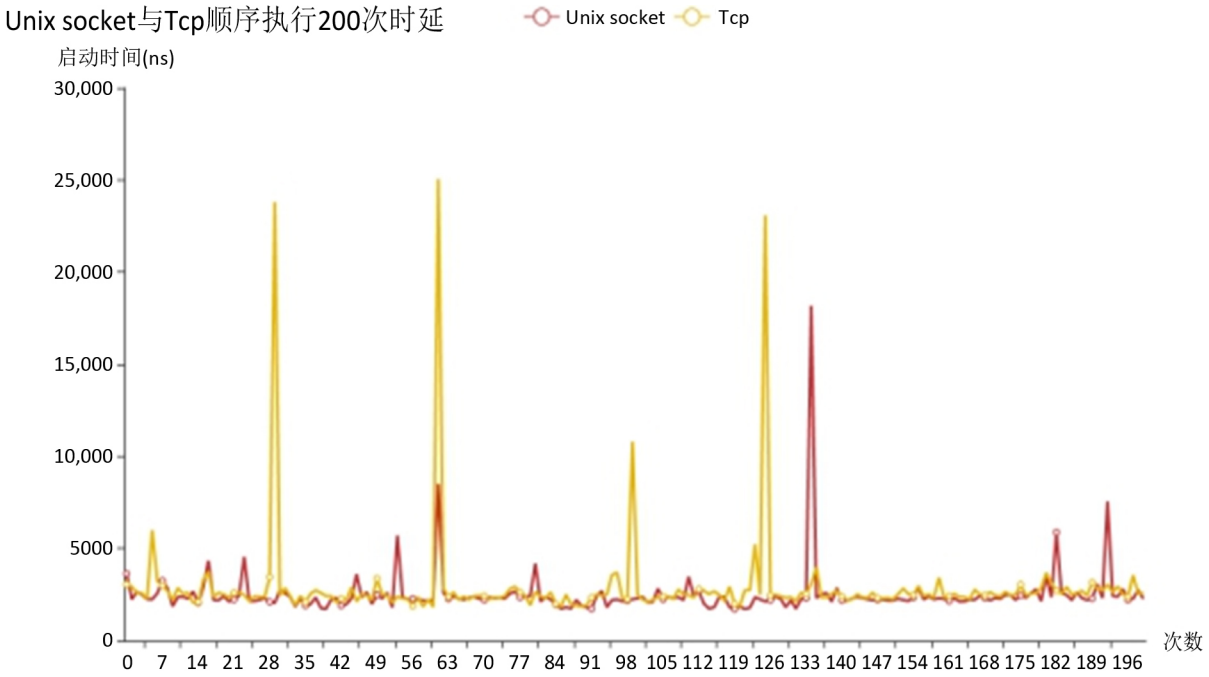


**Figure 2.** Unix Socket and Tcp 200 times sequential request delays
**图 2.** Unix Socket 与 Tcp 顺序执行 200 次请求时延

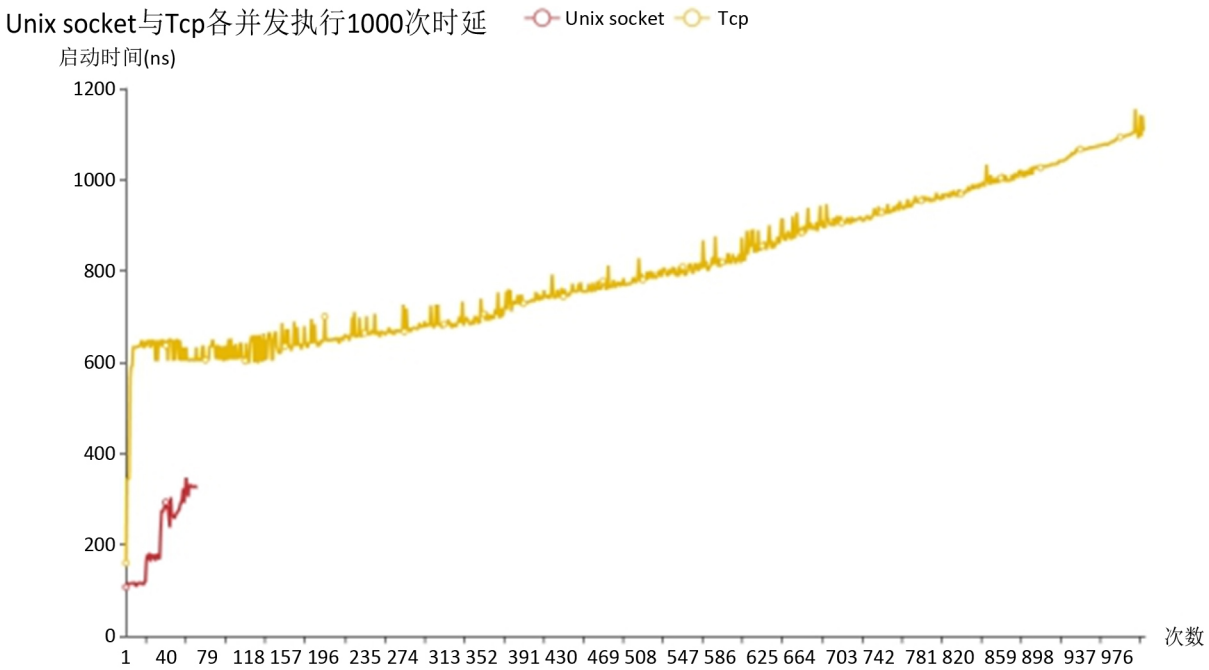


**Figure 3.** Unix Socket and Tcp 1~1000 times concurrent request delays
**图 3.** Unix Socket 与 Tcp 1~1000 次并发请求时延

### 3.3. 并发执行

我们对两种通信方式进行实验，分别并发执行 1~1000 次 List Images [24]方法请求 Docker deamon，时延为以向 docker demon 发送请求开始到 Docker deamon 返回数据为止。我们发现，当达到一定并发量时，使用 Unix Socket 通信方式请求 Docker deamon，Docker deamon 无法正常响应。图 3 为 Unix Socket 与 Tcp 各并发执行 1000 次的时延，当 Unix sock 时延没有数据说明 Docker deamon 出现无法正常响应的情况，容器也没有启动成功，根据图中数据可以得出当前实验环境下 Unix Socket 能接受的最大并发请求数为 80 次左右，而 Tcp 方式不论并发请求次数多少都可以正常响应。

表 2 是不同并发次数的各个数据，为了降低偶然性，我们不同并发次数执行各了 10 次，然后将 10 次数据汇总起来一起统计。整体而言在并发情况下，可以看出 Unix Socket 通信方式响应速度更快，可以承载的并发数量低，网络抖动小。Tcp 通信方式响应速度慢，可以承载的并发数量高，网络抖动大。因此我们建立了一个数学模型来确定 Unix Socket 和 Tcp 两种方式的优劣

$$R = \frac{t_x - t_{\min}}{t_{\max} - t_{\min}} + \frac{c_x - c_{\min}}{c_{\max} - c_{\min}} + err \tag{2}$$

$$S = \min\left(R_u, R_t\right) \tag{3}$$

公式(2)为计算优劣系数，其中 $t_x$ 为平均通信时延，$t_{\min}$ 为通信时延最小值，$t_{\max}$ 为通信时延最大值，$c_x$ 为变异系数值，$c_{\min}$ 为变异系数最小值，$c_{\max}$ 为变异系数最大值。$err$ 为请求失败次数。

公式(3)为取 Unix Socket 和 Tcp 两种通信方式计算得出的优劣系数最小的值为最佳通信方式。根据我们的实验结果得出结论并发启动在一定数量之前选择 Unix Socket 通信方式最佳。在并发启动数量在一定数量之后选择 Tcp 通信方式最佳。

**Table 2.** Unix Socket and Tcp two kinds of communication statistics
**表 2.** Unix Socket 与 Tcp 两种通信统计

| 通信方式 | 并发次数 | 最大响应时延(ns) | 最小响应时延(ns) | 平均通信时延(ns) | 变异系数 | 异常次数 |
|---|---|---|---|---|---|---|
| Unix Socket | 100 | 69 | 3 | 29.304 | 0.000478028 | 0 |
| tcp | 100 | 88 | 3 | 32.916 | 0.000535878 | 0 |
| Unix Socket | 200 | 110 | 6 | 59.776 | 0.000201821 | 2 |
| tcp | 200 | 120 | 6 | 63.309 | 0.000194561 | 0 |
| Unix Socket | 300 | 180 | 5 | 77.171 | 0.000150895 | 89 |
| tcp | 300 | 197 | 9 | 92.733 | 0.000127485 | 0 |
| Unix Socket | 500 | 275 | 12 | 131.661 | 0.000096416879 | 349 |
| tcp | 500 | 303 | 12 | 155.422 | 0.000081914477 | 0 |
| Unix Socket | 1000 | 515 | 3 | 201.462 | 0.000067467604 | 2083 |
| tcp | 1000 | 624 | 28 | 331.870 | 0.000040800157 | 0 |

## 4. 容器并发影响因素

### 4.1. 容器启动过程

我们测试容器并发启动，因此容器镜像下载的时延不在我们考虑之内，所以我们提前下载好要测试的镜像，确保容器镜像传输不影响容器并发启动实验数据。

单个容器启动过程如图 4，1) Client 发送一个创建容器请求。2) Docker deamon 接收请求进行处理后调用 Linux namespace 等命令进行创建容器操作。3) 容器创建成功发送创建成功的响应给 Client。4) Client 发送容器启动请求给 Docker deamon。5) Docker deamon 进行启动容器操作。6) Docker demon 返回给 Client 端响应，同时容器进行加载库文件和启动相关服务等操作，为了避免 Docker demon 没有正常响应请求问题，我们统一采用 Tcp 方式请求 Docker deamon。

测试容器启动成果条件根据容器镜像类型不同而不同，我们对容器类型进行了分类，如表 3。我们利用在容器中运行最简单的任务或等待容器报告就绪来度量启动时间。作为语言类容器，任务就是的编译或者解释一个简单的“hello world”语言程序。Linux 发行版的镜像就是运行一个非常简单的 shell 命令，通常是“echo hello”。对于长时间运行的服务器(尤其是数据库和 web 服务器)，我们测量到容器输出“ready”的信息的时间。对于一些特别的服务器，将轮询公开端口，直到有响应为止。

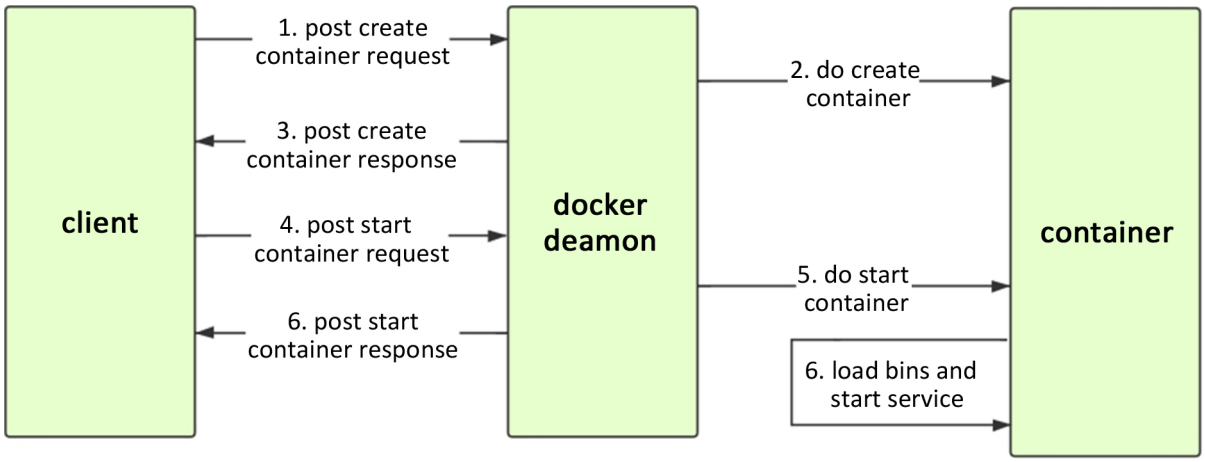


**Figure 4.** Container startup process
**图 4.** 容器启动流程

**Table 3.** Container image classification
**表 3.** 容器镜像分类

| 类别 | 镜像 |
|---|---|
| Linux 发行版 | alpine, busybox, centos, cirros, crux, Debian, fedora, mageia, opensure, oracleLinux, ubuntu, ubuntu-debootstrap, ubuntu-upstart |
| 数据库类 | cassandra, crate, elasticsearch, mariadb, mongo, mysql, percona, postgres, redis, rethinkdb |
| 语言类 | Clojure, gcc, golang, haskell, hylang, java, jruby, julia, mono, perl, php, pypy, python, r-base, rakudo-star, ruby, thrift |
| Web 服务器类 | Glassfish, httpd, jetty, nginx, php-zendserver, tomcat |
| Web 框架类 | Django, iojs, node, rails |
| 其他 | Drupal, ghost, hello-world, jenkins, rabbitmq, registry, sonarqube |

## 4.2. 容器并发启动硬件因素分析

首先我们对容器并发启动时延进行了测试，启动的镜像为 ubuntu。结果如图 5，可知容器的平均启动时延与并发启动的次数成正比，平均启动时延随着并发启动次数的增加而增大。

我们对启动过程影响因素进行了汇总，主要通过容器并发启动时 CPU 使用率，内存使用情况和磁盘 IO 值来确定。磁盘 IO 值使用磁盘工作时间占用总时间的百分比。公式为：

$$util = \frac{\Delta_{io}}{\Delta_t} \tag{4}$$

公式(4)中为 $\Delta_{io}$ 写操作消耗的时间，$\Delta_t$ 为采样周期。为了研究并发启动时何种因素对容器启动速度影响最大。我们采集了并发启动数量从 1 次到 10 次的数据，每次测试 10 次取中值。

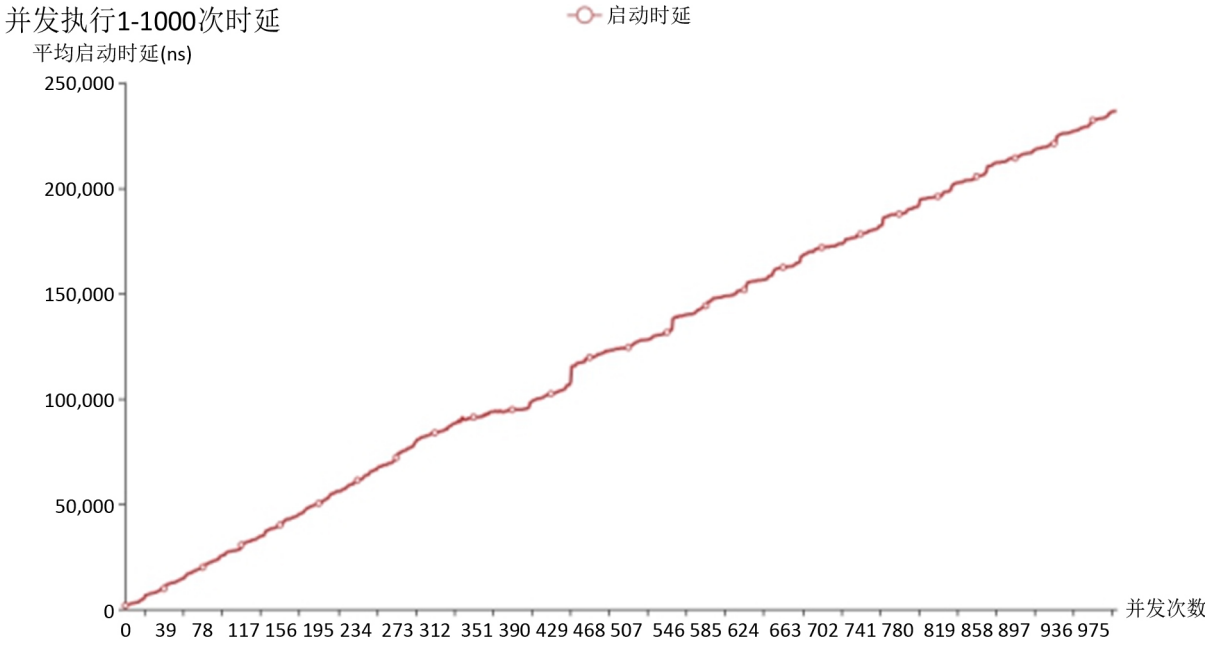


**Figure 5.** Containers 1~1000 concurrent startup delays
**图 5.** 容器 1~1000 次并发启动时延

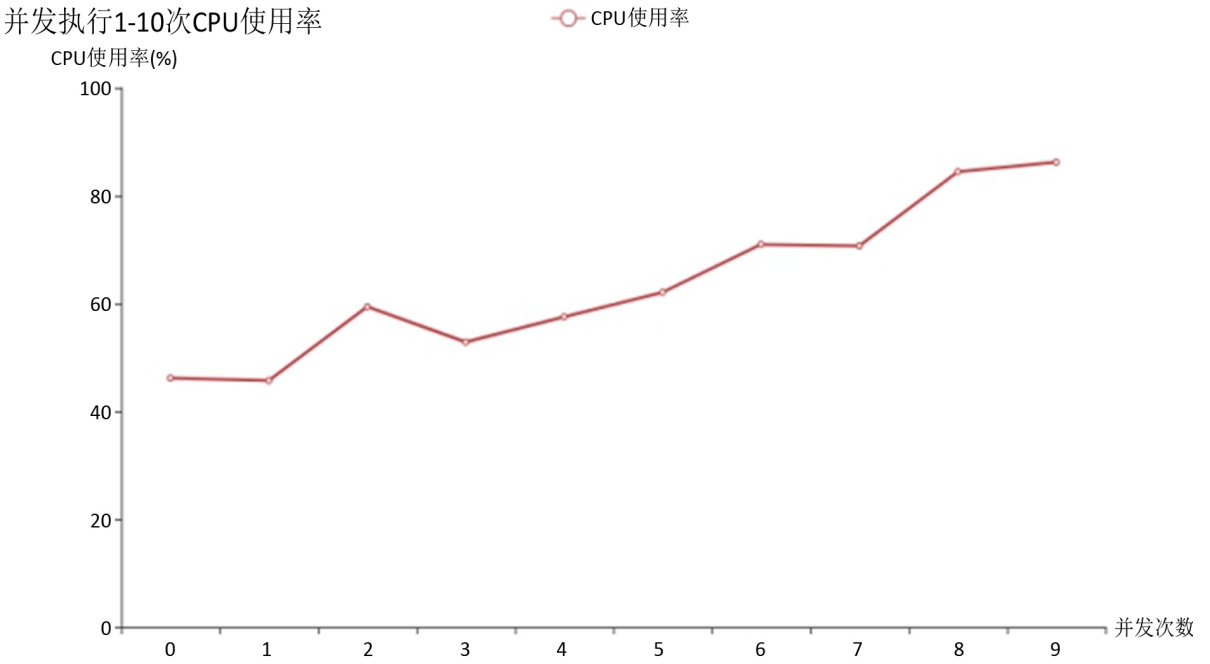


**Figure 6.** The container 1~10 times concurrently startup CPU usage
**图 6.** 容器并发启动 1~10 次 CPU 使用率

图 6 为容器并发启动 1~10 次 CPU 使用率，可以看到 CPU 成线性增长，到最后基本达到饱和状态。图 7 为容器并发启动的详细时延，可以得到启动时延根据并发启动的数量增加而增加。图 8 为容器并发启动的内存使用情况，可以看到内存几乎稳定在 3000 M 到 4000 M 之间，变化不明显。图 9 为容器并发启动磁盘 IO 值，可以看到磁盘 IO 值为 0.10%~0.15%之间，可以认为基本不使用磁盘 IO。

综上可得，影响容器并发启动时延的主要因素为 CPU 使用率，如果要提高并发性能，可以考虑优化 CPU 使用率。

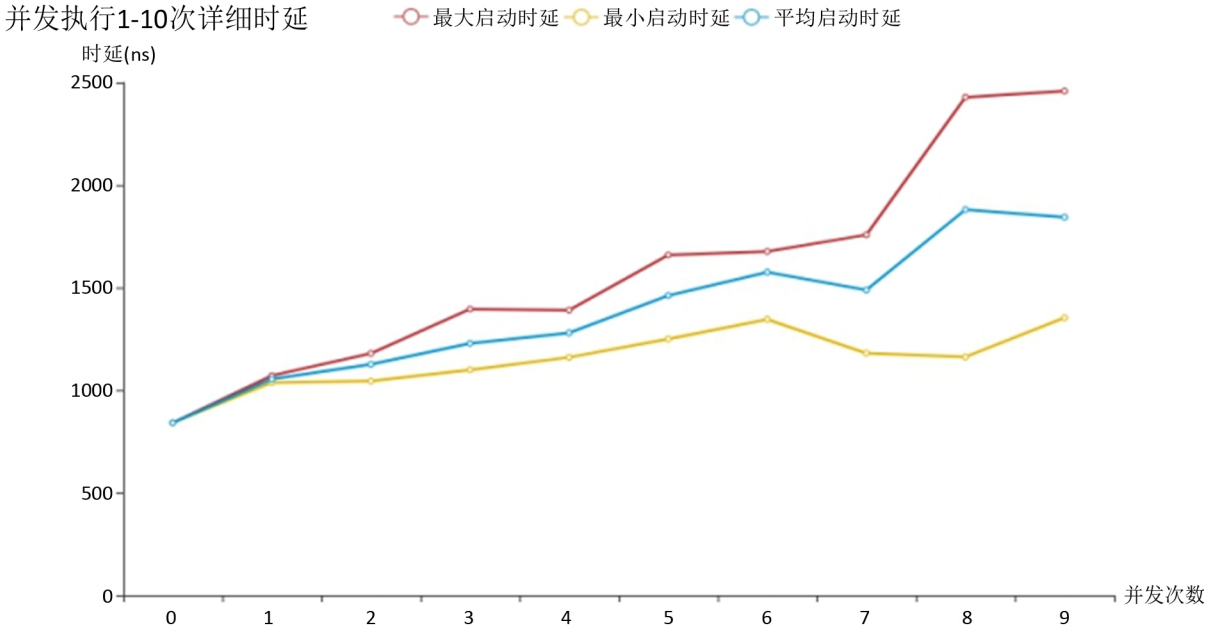


**Figure 7.** The container 1~10 times concurrently startup details
**图 7.** 容器并发启动 1~10 次详细时延

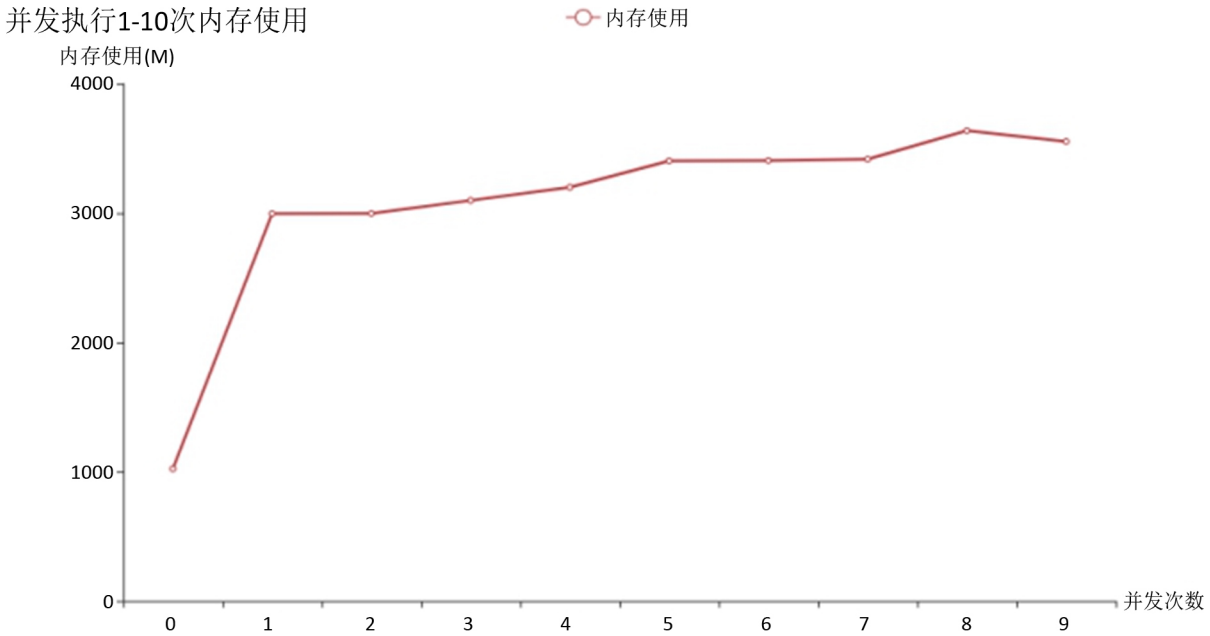


**Figure 8.** The container 1~10 times concurrently startup memory usage
**图 8.** 容器并发启动 1~10 次内存使用

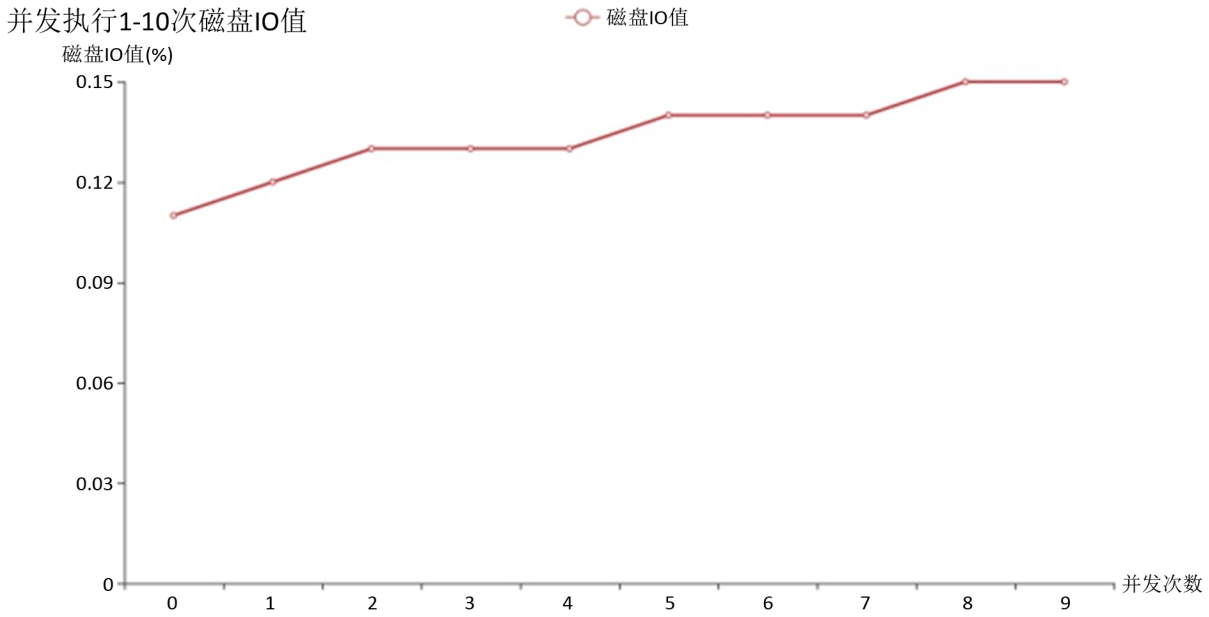


**Figure 9.** The container 1~10 times concurrently startup disk IO
**图 9.** 容器并发启动 1~10 次磁盘 IO

### 4.3. 容器并发启动镜像因素

为了确定容器并发启动速度是否和镜像大小相关，我们分别并发启动了 1000 个镜像大小为 78.43 M 的 ubuntu 镜像和 1000 个镜像大小为 908 MB 的 Node 镜像。镜像为 Docker hub [27]官方镜像，并且对它们的启动时延进行对比。

结果如图 10，发现 Ubutu 镜像和 Node 镜像启动时延相差不明显，得出结论镜像大小不是影响并发启动的主要因素。

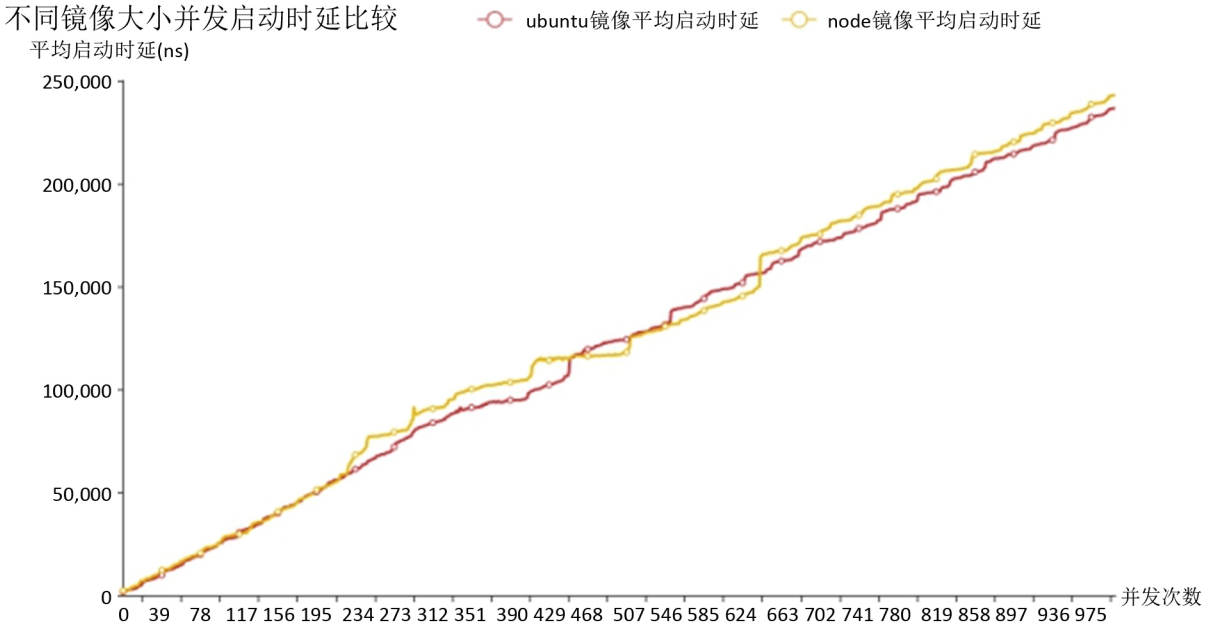


**Figure 10.** Comparison of concurrent startup delays between different image sizes
**图 10.** 不同镜像大小并发启动时延比较

### 4.4. 预先创建容器

根据 docker 容器启动步骤，分别为容器创建过程和容器启动过程[1]。因此我们确定容器预先创建好之后是否能提高容器并发启动效果。结果如图 11。发现预先启动容器镜像再并发启动时延比正常容器并发启动时延提升了 10%，效果不是很明显。所以得出结论预先创建容器对容器并发启动效果提升不明显，容器并发启动最主要的性能损耗在容器启动阶段，不是在容器创建阶段。

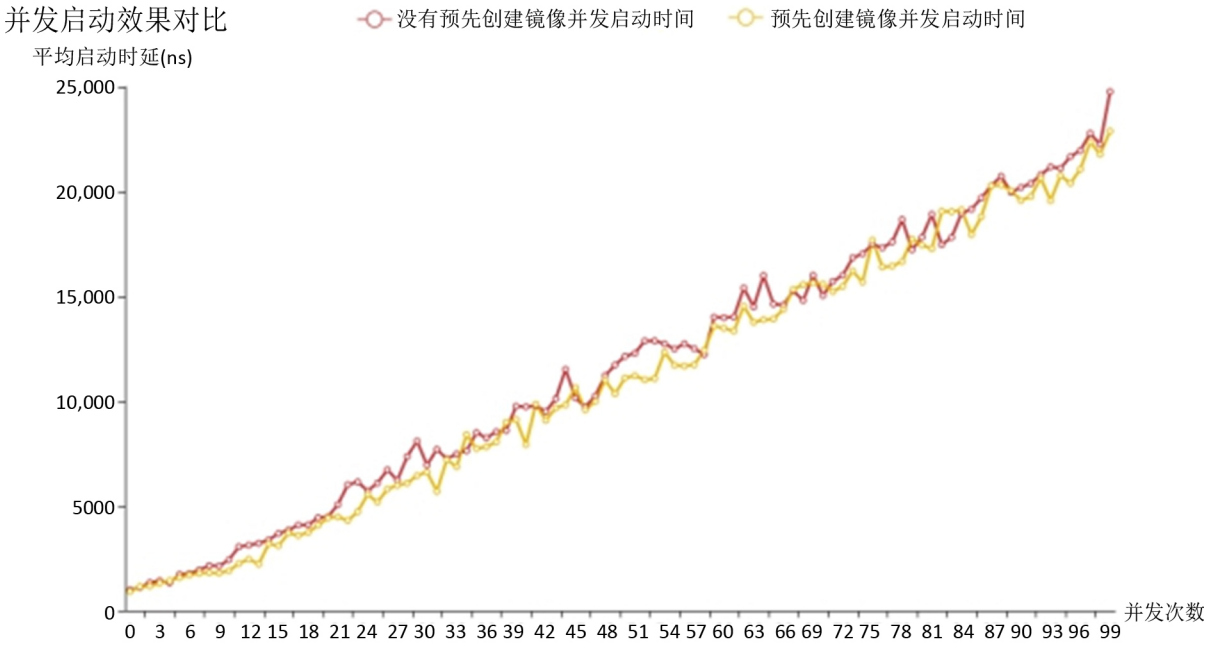


**Figure 11.** Concurrent startup of the container image after pre-creation
**图 11.** 预先创建容器镜像后并发启动效果对比

## 5. 未来展望

针对容器并发启动研究，我们可以根据得出的结论来针对这一特定场景进行专门的优化，来提高容器并发启动性能。可以采用 CPU 分时计算的思想来提高 CPU 利用率，根据时间序列预测的方法提前创建容器等操作，也可以进一步细化容器并发启动研究，针对容器启动过程的各个阶段来进行实验得出结果，并且容器并发启动应用环境不只有实训平台下的应用场景，有可能在高性能计算等领域也会出现相关场景，可以根据特定场景的特性来进行针对性研究。

## 基金项目

国家自然科学基金(61672384, 61772372)。

## 参考文献

[1] Docker, M.D. (2014) Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, **2014**, Article No. 2.

[2] Akkus, I.E., Chen, R., Rimac, I., *et al*. (2018) SAND: Towards High-Performance Serverless Computing. 2018 *USENIX Annual Technical Conference*, Boston, 11-13 July 2018, 923-935.

[3] Xu, Y., Mahendran, V. and Radhakrishnan, S. (2016) SDN Docker: Enabling Application Auto-Docking/Undocking in Edge Switch. 2016 *IEEE Conference on Computer Communications Workshops* (*INFOCOM WKSHPS*), San Francisco, 10-14 April 2016, 864-869. https://doi.org/10.1109/INFCOMW.2016.7562199

[4] Azab, A. (2017) Enabling Docker Containers for High-Performance and Many-Task Computing. 2017 *IEEE International Conference on Cloud Engineering*, Vancouver, 4-7 April 2017, 279-285. https://doi.org/10.1109/IC2E.2017.52

[5] Kovács, Á. (2017) Comparison of Different Linux Containers. 2017 40*th International Conference on Telecommunications and Signal Processing* (*TSP*), Barcelona, 5-7 July 2017, 47-51. https://doi.org/10.1109/TSP.2017.8075934

[6] Bernstein, D. (2014) Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, **1**, 81-84. https://doi.org/10.1109/MCC.2014.51

[7] Paolino, M., Hanmayun, M.M. and Raho, D. (2014) A Performance Analysis of ARM Virtual Machines Secured Using Selinux. *Cyber Security and Privacy Forum*, Springer, Cham, 28-36.

[8] Kratzke, N. (2014) A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms. *Open Journal of Mobile Computing and Cloud Computing*, **1**, 17-30. https://doi.org/10.4236/jcc.2014.212001

[9] Madhavapeddy, A., Mortier, R., Rotsos, C., *et al*. (2013) Unikernels: Library Operating Systems for the Cloud. *ACM SIGARCH Computer Architecture News*, **41**, 461-472. https://doi.org/10.1145/2490301.2451167

[10] Eiras, R.S.V., Couto, R.S. and Rubinstein, M.G. (2016) Performance Evaluation of a Virtualized HTTP Proxy in KVM and Docker. 2016 7*th International Conference on the Network of the Future* (*NOF*), Buzios, 16-18 November 2016, 1-5. https://doi.org/10.1109/NOF.2016.7810144

[11] Joy, A.M. (2015) Performance Comparison between Linux Containers and Virtual Machines. *International Conference on Advances in Computer Engineering and Applications*, Ghaziabad, 19-20 March 2015, 342-346. https://doi.org/10.1109/ICACEA.2015.7164727

[12] Seo, K.-T., Hwang, H.-S., Moon, I.-Y., Kwon, O.-Y. and Kim, B.-J. (2014) Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud. *Advanced Science and Technology Letters*, **66**, 105-111. https://doi.org/10.14257/astl.2014.66.25

[13] Xavier, B. and Jersak, L. (2016) Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform. 2016 16*th IEEE/ACM International Symposium on Cluster*, *Cloud and Grid Computing* (*CCGrid*), Cartagena, 16-19 May 2016, 277-280. https://doi.org/10.1109/CCGrid.2016.86

[14] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. and Arpaci-Dusseau, R. (2018) SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. *USENIX Annual Technical Conference*, Boston, 11-13 July 2018, 57-70. https://www.usenix.org/conference/atc18/presentation/oakes

[15] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. (2016) Slacker: Fast Distribution with Lazy Docker Containers. *Proceedings of the* 14*th Usenix Conference on File and Storage Technologies*, Santa Clara, 22-25 February 2016. 181-195. https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[16] Rizki, R., Rakhmatsyah, A. and Nugroho, M.A. (2016) Performance Analysis of Container-Based Hadoop Cluster: OpenVZ and LXC. 2016 4*th International Conference on Information and Communication Technology*, Bandung, 25-27 May 2016, 1-4. https://doi.org/10.1109/ICoICT.2016.7571957

[17] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J. (2015) An Updated Performance Comparison of Virtual Machines and Linux Containers. 2015 *IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), Philadelphia, 29-31 March 2015, 171-172. https://doi.org/10.1109/ISPASS.2015.7095802

[18] Xie, B., Sun, G. and Ma, G. (2017) Docker Based Overlay Network Performance Evaluation in Large Scale Streaming System. *Proceedings of* 2016 *IEEE Advanced Information Management*, *Communicates*, *Electronic and Automation Control Conference*, Xi'an, 3-5 October 2016, 366-369. https://doi.org/10.1109/IMCEC.2016.7867235

[19] 李巍, 赵永彬, 王鸥, 等. 基于 Macvlan 的 Docker 容器网络架构研究[J]. 机械设计与制造, 2017(5): 270-272.

[20] 王志伟, 杨超. 基于流量控制的 Docker 容器网络带宽控制机制[J]. 计算机应用, 2019(12): 3628-3632.

[21] 张礼庆, 郭栋, 吴绍岭, 等. 一种最大化内存共享与最小化运行时环境的超轻量级容器[J]. 计算机研究与发展, 2019, 56(7): 1545-1555.

[22] 张可颖, 彭丽苹, 吕晓丹, 等. 开源云上的 Kubernetes 弹性调度[J]. 计算机技术与发展, 2019, 29(2): 109-114.

[23] 谢晓兰, 张征征, 王建伟, 等. 基于三次指数平滑法和时间卷积网络的云资源预测模型[J]. 通信学报, 2019, 40(8): 143-150.

[24] Paraiso, F., Challita, S., Al-Dhuraibi, Y., *et al*. (2016) Model-Driven Management of Docker Containers. 2016 *IEEE* 9*th International Conference on cloud Computing* (*CLOUD*), San Francisco, 27 June-2 July 2016, 718-725. https://doi.org/10.1109/CLOUD.2016.0100

[25] Smith, S. (2015) On-Demand Activation of Docker Containers with System. https://blog.developer.atlassian.com/docker-systemd-socket-activation

[26] https://docs.docker.com/engine/api/v1.38/#operation/ImageList

[27] https://hub.docker.com