Computer Science and Application 计算机科学与应用, 2020, 10(12), 2257-2269
Published Online December 2020 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2020.1012237

Hans汉斯

一种大规模容器镜像分发加速模型及其实现方法

潘　瑭

同济大学，电子与信息工程学院，上海
Email: goodm3owing@gmail.com

收稿日期：2020年12月1日；录用日期：2020年12月15日；发布日期：2020年12月23日

摘　要

云计算逐渐成为一种主流的基础服务，容器由于其轻便型是目前云计算之中最常使用的一种虚拟化抽象。基于镜像这样的模板容器可以迅速部署应用在云服务器上。然而现有的容器引擎在大规模镜像分发时容器会产生冷启动时间过长的问题。本文分析并验证了影响Docker容器冷启动的关键因素，提出了一种大规模容器镜像分发加速模型。通过以文件为粒度的延迟加载，以及按使用顺序的文件分层，加快了镜像的传输速度，进而加速容器启动。基于该模型，实现了镜像分发系统D4C (Doing Deft Distribution of Docker Container)。并从容器冷启动时间、启动镜像大小、网络传输量三个方面进行测试，验证了D4C在这几个方面的优势。

关键词

容器，Docker，镜像分发，传输加速

A Large-Scale Container Image Distribution Acceleration Model and its Implementation Method

Tang Pan

School of Electronics and Information Engineering, Tongji University, Shanghai
Email: goodm3owing@gmail.com

Received: Dec. 1st, 2020; accepted: Dec. 15th, 2020; published: Dec. 23rd, 2020

Abstract

Cloud computing has gradually become a mainstream basic service, and containers are currently the most commonly used virtualization abstraction in cloud computing due to their lightness.

文章引用: 潘瑭. 一种大规模容器镜像分发加速模型及其实现方法[J]. 计算机科学与应用, 2020, 10(12): 2257-2269.
DOI: 10.12677/csa.2020.1012237

Based on such template containers, applications can be quickly deployed on cloud servers. However, the existing container engine will have a long cold start time when the container is expanded and distributed. The key factors affecting the cold start of Docker containers are analyzed and verified here, and an expanded container stack acceleration model is proposed. By taking the file as the model, the distributed system D4C (Deft distribution to Docker containers) is realized. Tests were conducted in terms of container cold start time, boot image size, and network transmission volume to verify the advantages of D4C in these aspects.

Keywords

Container, Docker, Image Distribution, Transportation Acceleration

1. 引言

容器(container)是虚拟化技术的一种。由于其轻便性是目前云计算之中最常使用的一种虚拟化抽象。容器与虚拟机不同的地方在于，容器与宿主机共享内核。因此通常比传统虚拟机占用内存小[1]。容器有三个关键组件：1) 强制执行进程级隔离的 OS 机制(例如 Linux cgroups 和命名空间机制)，2) 应用程序打包系统和运行时(例如 Docker，Rkt)和 3) 跨机器部署，分发和管理容器的编排管理器(例如 Docker Swarm，Kubernetes)。

目前最流行的容器是 dotCloud 公司的 Docker [2]。Docker 运行时环境被打包在一组文件中，这些文件称为 Docker 镜像(image)。Docker 镜像由一系列层(layer)组成。每层都是一个只读文件系统包含一组文件或目录。Docker 仓库中存储有关于 Docker 镜像的两种文件：Manifest 和 Blob。Manifest 描述了有关 Docker 镜像的元信息。它指示镜像所具有的层以及每个层所引用的 Blob。Blob 是图层的压缩文件。每一层都有一个对应的 Blob。

Docker 容器的新部署包括两个步骤：1) 从 Docker 仓库中拉取(pull)已发布(push)的镜像，以及 2) 基于该镜像启动一个容器。然而，新部署容器所花费的时间中有 76%将花费在拉取镜像上[3]。为了加快速度，通常会在本地集群中部署私有 Docker 仓库。但是 Docker 仓库所在主机的带宽无疑将成为大规模容器部署的瓶颈。部署两个或多个 Docker 仓库实例可以在某种程度上缓解此问题，但会引发新问题，例如镜像同步和负载平衡。因此，需要一种快速、优雅的方法来分发 Docker 镜像以进行大规模容器部署。

本文立足于解决集群内部容器镜像大规模传输部署的加速问题，探究多种容器镜像分发加速方法，构建一个新型的镜像分发加速模型。最终实现一套容器镜像分发加速系统。

本文的主要贡献有 3 个方面：

1) 通过分析和实验证明了在容器启动过程中，只有少量文件会被使用。并且通过分析典型容器的镜像，给出了镜像大小与启动时文件使用率的关系。

2) 提出了一种针对大规模容器镜像分发的加速模型，通过镜像启动时文件被加载的先后时间来优先传输必须的文件，减少了启动必须的文件传输的时间。

3) 基于以上思路实现了一种大规模容器镜像分发加速模型 D4C (Doing Deft Distribution of Docker Container)。在顺序启动和并发启动速度、网络传输数据量等方面进行对比实验，验证了所实现模型的有效性。

本文第 2 节介绍容器分发加速的相关工作，第 3 节介绍影响 Docker 容器启动速度的关键因素，第 4 节介绍大规模容器镜像分发模型的整体设计，第 5 节介绍 D4C 的原型实现，第 6 节介绍试验和结果分析，第 7 节进行总结和展望。

2. 相关工作

2.1. 点对点传输

P2P 技术是一种经典的内容分发技术，它使用的 BitTorrent 协议可以解决密集下载时单一源地址网络负载过大导致速度下降的问题。BitTorrent 协议在 Internet 上广泛用于文件共享。它将文件分成小块，对等节点(peer)通过相互交换文件块来共享文件。

在容器之前，P2P 技术已经在 VDN [4]、VMtorrent [5]、Orchestra [6]等虚拟机镜像分发系统中得到了很多利用。

DID、Quay 和 Docket 是以镜像为共享单位，最早将 P2P 方式应用在镜像分发中的开源软件。FID [7]第一个实现了企业级别的 P2P 容器镜像仓库系统。CoMICon [8]在实现 P2P 传输的同时，每个主机只保留部分的镜像层，达到减少存储空间和网络负载的效果。HDID [9]优化了 BitTorrent 协议，并对进行 P2P 传输的镜像层做了筛选，获得了比传统方法更好的效果。Dragonfly 是阿里开源的一个 P2P 镜像分发系统，是一套切实可用的解决方案。

P2P 的问题在于，当传输的文件粒度过小时，管理 P2P 下载信息的消耗可能大于性能的提升。当传输的文件粒度过大时，又会出现很大的冗余。P2P 传输应该的粒度应该是怎样的，找到这之中的平衡点是一个问题。

2.2. 延迟加载

延迟加载是指不预先拉取整个容器镜像，而是只拉取部分，其余等待有需要时再进行单独加载。容器方面延迟加载相关的研究相对虚拟机较少。DRR 提高了针对密集型容器密集型工作负载的普通写时复制性能。Tarasov 等[10]研究了存储驱动器选择在容器内运行的不同工作负载时对 Docker 容器性能的影响。Slacker 以块为粒度加载容器镜像来最小化启动延迟，减少了部署周期。YOLO [11]将容器启动时所必须的磁盘块搜集起来作为启动镜像，而其他的数据单独加载。另有研究[12]在拉取镜像时仅仅加载一个记录了镜像层存储地址的文本文件。

分析哪些部分可以进行延迟加载可能会带来额外的 CPU 开销。大多数延迟加载采用了新的机制，需要改动 Docker 镜像运行的方式。即需要修改 Docker 源码，难以实现与现行方式的兼容。例如：slacker 通过一个 bitmap 判断那些文件块是镜像间共享的，从而以块为粒度进行延迟加载。但是 Docker 镜像划分的最小粒度只是层。

2.3. 镜像缓存

缓存和预取一直是提高系统性能的有效技术。例如，现代数据中心使用分布式内存缓存服务器[13]通过缓存查询结果来提高数据库查询性能，优化数据在数据中心的存储[14]也是一个提升的方向。大量工作[15] [16]研究了组合缓存和预取的效果。

在内存中缓存直接镜像是一个比较简单的思路，这方面针对虚拟机已经有许多应用。有的尝试通过挂起整个虚拟机并在必要时恢复以加速其启动时间[17]，但是由于虚拟机的重量性，这样会给内存带来很大压力。VMThunder+ [18]利用 SSD 而不是内存来作为缓存设备，μVM [19]是一种基于延迟恢复技术的解决方案，使用最少的资源配置来还原已引导的 VM 的快照，但是这两种方案仍然没有解决占用内存过大的问题。

Docker 本身在拉取镜像时会保留所有的镜像层，若新的镜像使用到了这些层则不必重复拉取。IBM 公司[20]通过对生产环境下镜像拉取记录的分析，将 12 小时以内更新过的镜像层放入了镜像仓库的内存中缓存。Bolt [21]利用了 IBM 的日志数据，在新的仓库设计中对镜像层中小于 1MB 的部分进行了内存缓存。

2.4. 网络文件系统

网络文件系统(NFS)是文件系统之上的一个网络抽象，来允许远程客户端以与本地文件系统类似的方式，来通过网络进行访问。网络文件系统可以避免每一个服务节点上都单独保留一份镜像副本，减少网络传输的压力。

Cider [22]利用 Ceph 实现了网络存储，节省了镜像存放的磁盘空间和拉取的时间。Wharf [23]通过本地和全局两个状态的管理，通过分布式文件系统共享了容器镜像。Slacker 利用网络文件系统，所有的镜像在本地都不存在副本。

大多数使用 NFS 存储镜像的系统没有本地缓存，每一次使用启动容器时都会重新获取一次镜像。NFS 需要与每个节点都保持长久连接，在同时连接的节点规模较大时需要进行负载均衡。这会消耗额外的时间和网络。此外 NFS 的本质是集中存储，如果没有容灾策略，那么文件系统所在机器发生宕机时后果十分严重。NFS 还与 P2P 策略有一定的冲突。

2.5. 镜像精简

镜像传输的时间是与镜像的大小成正比的，精简容器镜像是加速传输最有效率的方式。

CNTR [24]利用开源软件 Docker-slim 进行了镜像精简，精简过后的镜像减少了一些非必要功能，但是大小比原来下降了一到两个数量级。YOLO 将容器启动时所必须的磁盘块搜集起来作为启动镜像，而其他的数据单独加载。Slimmer [25]分析镜像所包含的文件，通过去重保证所有文件只保留必须的一份，大大压缩了镜像空间。

但是镜像精简会额外消耗 CPU 性能。并且需要预处理，无法在收到镜像分发请求时就同步进行。随着精简方式的不同，镜像有可能会损失部分的功能。

3. 容器启动速度影响分析

3.1. 容器传输时间分析

图 1 展示了容器镜像启动过程中传输镜像的时间占总时间的比重。我们选取的镜像是在 Docker 官方仓库中被拉取次数靠前的多种类别的镜像，分别是 ubuntu (系统类)、redis (数据库类)、python (语言类)、nginx (网络服务器类)、node (网络框架类)。不同种类的镜像，由于其实现的功能不同，因此构成它们的镜像层是大相径庭的，这也就使得它们包含的文件也有较大差别，启动时所需求的文件亦然。因此使用几种不同种类的镜像进行实验，具有较好的代表性。所有镜像在本地仓库没有任何镜像层的缓存。使用 Go 语言构件的脚本统计传输时间与容器启动时间。容器启动时间从发出命令开始计算，以成功执行完毕该镜像所能提供的最基本操作作为结束标志。镜像传输时间从发出拉取命令开始计算，到文件拉取完成结束计算。可以看到，传输镜像的时间占了时间的绝大多数，大部分容器在启动过程中都需要消耗 70%以上的时间用于镜像传输。即使是最少的一个也占用了 62%的时间。这种时间占比并没有随着镜像大小的变化产生递减的趋势。也就是说，对于更大的镜像，就要消耗更多的时间在文件传输上。

一个普通 Docker 镜像分发部署的过程主要包括获取镜像的元数据、检测本地镜像层、获取镜像层压

缩包、解压和载入镜像层。由于 Docker 镜像的分层存储，当有多个镜像共享一个镜像层时，就可以减少镜像层的传输，从而加速容器启动的速度，这也是 Docker 设计的优势所在。甚至如果本地仓库含有对应容器的完整镜像，这部分时间就可以几乎忽略不计。

但是在大规模镜像分发的场景中，镜像往往都是初次被分发到部署的机器上，本地存在共享镜像层的概率也不高，在这样的情况下，容器镜像传输的时间就成为了容器启动的绝对瓶颈。

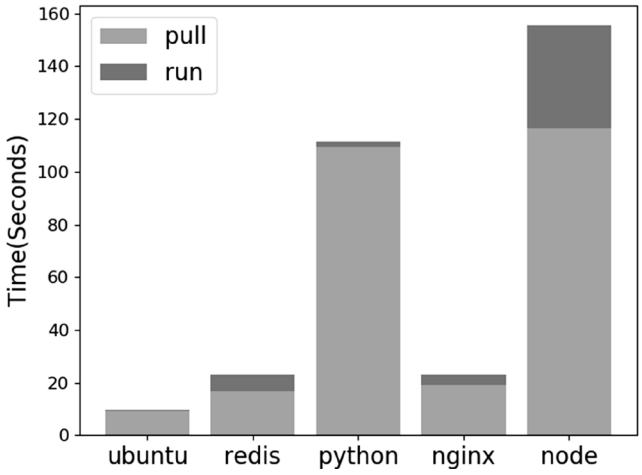

Figure 1. Cold start time of various containers
图 1. 各类容器的冷启动时间

3.2. 容器文件加载分析

图 2 展示了容器启动时被使用的文件占镜像所有文件的大小的比重。所选取的镜像与上一小节一致。在容器成功启动以后，通过 stat()系统调用查看文件的最近访问时间，并以此为依据统计启动以后被访问的文件。可以看到，大部分容器成功启动以后，被访问的文件不超过所有镜像文件的 11%。对于部分镜像，这个数字甚至低到了 6%。

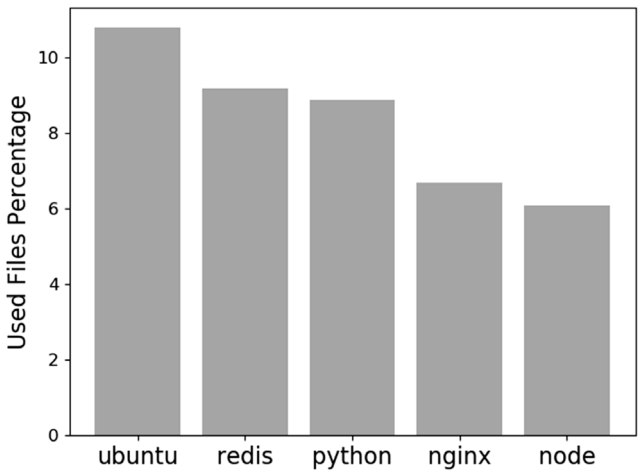

Figure 2. Percentage of file usage at startup of various containers
图 2. 各类容器启动时的文件使用百分比

这就是传统的 Docker 容器部署方式的弊端。它要求容器在启动时就必须将整个镜像都下载到本地。

这无疑大大拖慢了容器的启动速度。但实际上启动时所需的文件比率非常之小。因此，需要探究新的方法减少这部分数据的传输，快速准备容器所需的启动环境。

Docker 容器启动时进行全量传输的根源在于它的分层。它是以容器的制作顺序而非启动顺序进行分层的。Docker 将执行容器所需的一组运行时环境打包在镜像之中。然而并非镜像中的所有文件都是容器运行必须的。镜像制作时，通常会选择一个基础镜像，基础镜像之中便会包含许多不需要的文件。在基础镜像之上叠加镜像层的过程中，同样会产生很多冗余文件。这就使得镜像实际上成为了运行时环境的一个超集。当背景变为启动一个容器时，冗余的文件就更多了。

3.3. 小结

通过以上分析与事实可以获知，在容器启动的过程中，传输镜像文件所花费的时间占据了绝大多数。对于大部分的镜像，传输镜像文件的时间都超过了启动时间 70%，这无疑是容器启动的关键瓶颈。在需要迅速进行机器扩容等大规模镜像分发的场景中，容器镜像需要在待部署机器上不存在镜像缓存的情况下进行快速传输，镜像的大小将大大影响容器启动的时间。而在容器启动时，这些传输的文件又是大部分不需要使用的，可以在启动完成之后再进行延迟加载。在大部分容器成功启动以后，被访问的文件不超过所有镜像文件的 11%。并且随着镜像的大小逐渐增加，镜像启动时的文件利用率呈现逐步下降的趋势。这是符合我们的直觉的：因为体积越大的镜像，一般就包含了越多的依赖文件和镜像层，而大部分这些依赖在启动时是不会被使用的。

因此，优先传输启动必须的文件，减少启动时传输的镜像大小，可以极大地加速容器的启动。若是考虑到后续过程中文件访问的顺序，让运行一段时间之后才利用到的文件进行延迟传输和加载，可以使得访问文件时需要等待的传输时间减少更多。

4. 大规模容器镜像分发系统设计

在上一节的分析中，我们得到了两个有用的结论：镜像传输时间占据了容器启动时间的大部分；容器启动时所需求的文件只占镜像文件的小部分。依据这两点，本节提出了一种新型的大规模容器镜像分发系统，用于解决镜像分发的耗时问题。本文的设计基于 Docker 镜像，因此会主要对比与其的区别，下面从三个方面介绍。

4.1. 延迟加载镜像文件

如图 3 所示。首先，基于容器启动只利用小部分镜像文件，实现镜像的延迟加载。容器的启动不需要完整的镜像文件，也就意味着镜像的传输不需要传输完整的文件。在启动容器时，仅仅需要传输镜像中必须的小部分文件即可。当这部分文件传输完成，便可通知系统，可以进行下一步的操作，后续的文件则可以同步进行，或者等到需要时再进行额外传输。这样做相当于以流水线的方式，提前了容器启动的时间，在传输没有完全完成时就可以开始，自然加快了容器的启动速度。

4.2. 新的镜像分层

新的镜像分发系统将依据启动顺序分层而不是创建顺序分层，新的分层就是延迟加载镜像文件的依据。前文提到，Docker 镜像层是按照制作镜像时的顺序叠加的，这种方式有利于镜像层的管理，在一些条件下也有利于镜像的传输。在多个镜像共享某一镜像层时就可以减少传输的时间，只需要对该层镜像进行一次传输操作。但是根本上，这种方式仍然需要传输完毕所有的镜像层，才能进行容器启动的后续操作。

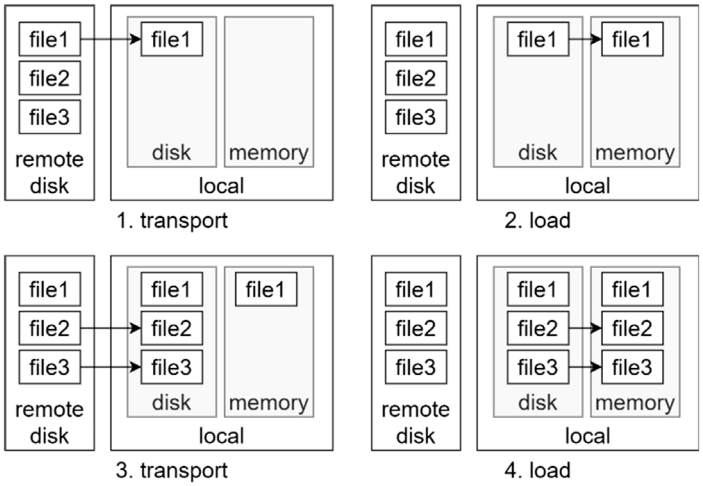

Figure 3. Delay loading of image files
图 3. 延迟加载镜像文件

延迟加载使得可以先加载小部分文件先实现容器启动。而首先加载的文件如何确定，就要依赖于新的镜像分层。新的镜像分层以访问时间为顺序为文件排序，将其分为若干层次。如图 4 所示，首先，所有必要的文件会被从 Docker 的原始镜像中抽取出来，也就是消除原本镜像层的分层，将其压缩成一层。而后，会根据访问时间，对这些文件进行分类，较早被访问与较晚被访问到的文件分处于不同的层次中，依据原始镜像的大小和特定参数进行分层数的确定。最后，根据上一步的分类重新构件镜像。此时镜像的分层已经使用新的方式，并且存储也不在是以压缩形式。

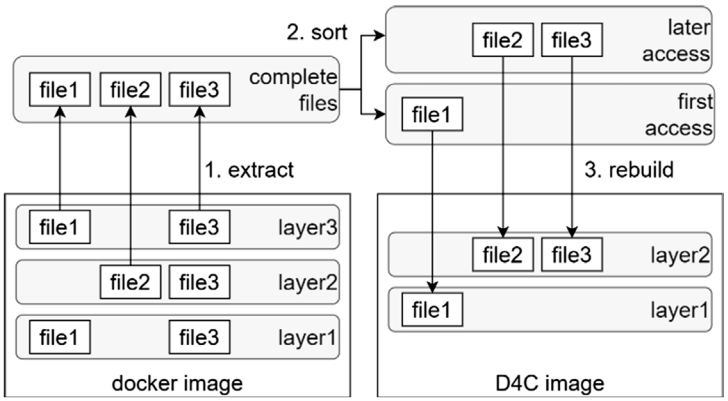

Figure 4. Image layering based on access time
图 4. 基于访问时间的镜像分层

具体流程如下：在进行容器分发之前，需要预先启动一次该容器，记录下容器对各个文件的访问时间，并按照一定的访问间隔将其划分到不同的层次。先被访问的划分到底层，后被访问的划分到高层。在进行镜像传输时，按照自顶向上的顺序进行传输。因为底层的文件是最先被访问的，最底层的文件可以定为在容器启动完成以后会访问的文件。这样只要底层加载完毕，就可以完成容器的启动，在其它层的文件可以延迟加载。

仅记录容器启动时所必须的文件可以完成镜像最底层的筛选，但是在后续请求文件的顺序仍然无法确定。因此可以在容器运行时添加一项可选的监控，获取镜像中各个文件在容器运行过程中被访问的时间。并以此为依据制定后续的文件分层。

4.3. 文件为粒度的镜像传输

如果使用镜像层作为传输的单位，那么文件的延迟加载仍然不够灵活。新的镜像分层仍然是包含了多个文件的。尽管已经预先根据访问顺序做了合理的分层，仍然可能出现访问少数几个文件却需要传输整个镜像层的行为。或者是当所请求的文件位于较高层时，需要将中间所有层都全部进行传输。

此外，由于与现有的 Docker 镜像层的定义并不一致，若以镜像层作为传输单位，便无法复用 Docker 现有的镜像管理系统。

因此，处于传输灵活性的考虑，同时为了在实现时最大可能地利用现有的 Docker 系统，并且与其兼容，新的镜像分发系统(图 5)将以文件而不是镜像层作为传输粒度。数据传输的单位是文件而不是镜像层，但是能够以镜像层为步长进行传输。具体地说，当发生文件缺失时，会首先出发单个文件的传输；当这种缺失发生的频率达到一个阈值，就会触发下一层镜像层的传输；而当传输的镜像层层数超过阈值，则又会触发全量的容器镜像传输。阈值会在首次启动原始 Docker 镜像，构建 D4C 镜像时确定。前者没有比较固定规律，后者一般超过 60%镜像层则触发。阈值的确定规则是，当剩余部分的传输时间介于原始方法耗时的 1 到 2 倍时，就以原始方法进行传输。但是 D4C 传输关键文件，镜像的启动时间是比传统方式要快的，但是传输同样数目的文件时，以文件方式传输必然会比压缩后的原始传输方式要多传输数据。因此，当传输剩余部分所需要的时间已经接近原始方法传输完整镜像时，就不再以 D4C 的方式进行传输。

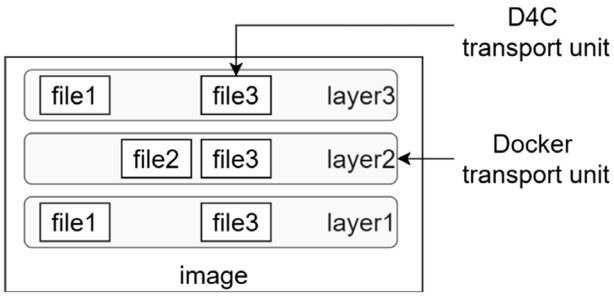

Figure 5. File-based image transport
图 5. 文件为粒度的镜像传输

5. 原型实现

本节基于上一节中提到的设计思路，实现了针对大规模容器镜像分发的专用系统 D4C。D4C 的整体框架如图 6 所示，其主要包 D4CLocalRepository 和 D4CRemoteRegistry 两个部分。

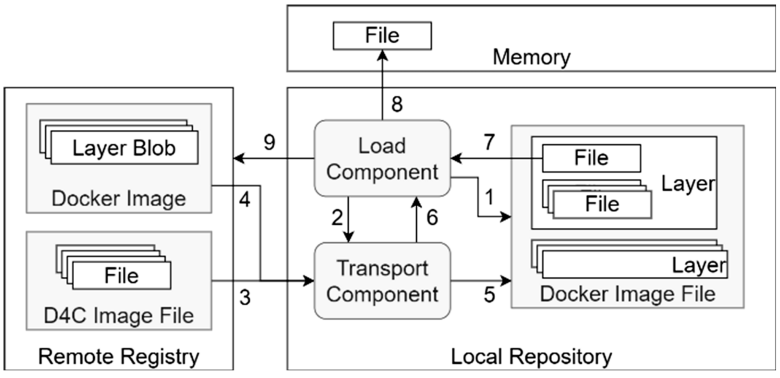

Figure 6. Architecture of D4C
图 6. D4C 架构图

D4C 工作的流程如下：1) LoadComponent 监控本地仓库的 Docker 镜像文件。2) 发生文件缺失时，通知 TransportComponent。3) 若是一般情况，则以文件为粒度进行精准传输。4) 若是缺失文件过多，则触发原始镜像拉取方式。5) 无论以何种方式传输，最终文件还是以 Docker 镜像文件的方式存储。6) 完成文件传输以后 TransportComponent 会通知 LoadComponent 已完成缺失文件的传输。7) LoadComponent 按需加载所需文件。8.最终将文件加载到内存中供容器运行使用。

D4CLocalRepository 是 D4C 系统的本地镜像仓库。充分复用 Docker 原本的镜像管理系统。具有两种拉取镜像的方式，一种是使用传统的镜像传输方式，利用 DockerRegistry 传输镜像层压缩包，再解压成为实际的镜像文件。另一种使用 D4C 特有的镜像拉取方式。以文件为粒度，从专门的 D4CRemoteRegistry 按需拉取镜像文件。后者拉取到的文件，与从 DockerRegistry 解压出来的文件没有区别。

D4CLocalRepository 又由两个部分组成，分别是 D4CLoadComponent、D4CTransportComponent。

D4CLoadComponent 负责管理文件的加载，运行时需要的文件就由该部件提供，当发生缺页，本地的仓库又没有相应文件时，该部件就会做出判断，该以何种方式加载缺失的文件。当缺失文件的频率不高时，仍然以 D4C 系统的方式进行补偿，仅进行缺失文件的传输。保证在最短时间内可以获取到运行所需的文件。但当缺失文件的频率过高时，就触发镜像的全量传输。使用传统的方式先拉取镜像各层压缩包，然后再解压成为完整的镜像文件，最后加载到内存中使用。根据我们的设计，容器启动时所必须的文件已经在启动时加载完毕，保证了容器启动的速度。而后续运行中缺失的文件再另外进行加载。

此外，该部件有一个 recordmode，在该模式下运行时，会记录下各个容器各个文件的访问顺序，并在一定时间以后发送给 D4C RemoteRegistry，作为后续优化文件加载顺序的依据。

D4CTransportComponent 负责进行文件的传输。当容器运行时发生文件缺失，就会传递消息给该部件，然后由其完成后续的文件传输操作。这部分的操作对于上层的 LoadComponent 而言是透明的。当文件传输完成以后 TransportComponent 会向 LoadComponent 发出信号，此时后者会控制容器完成新文件的加载操作。

D4CRemoteRegistry 是 D4C 系统的远程镜像仓库。与本地仓库的两种拉取镜像方式对应，也有两种存储镜像的方式。传统的 Docker 远程仓库存储的只有压缩后的镜像层，而对于 D4C 的仓库，还需要额外存储解压后完整的镜像文件。但是以额外的存储空间换取传输镜像时的高效，这样的代价是可以接受的。远程仓库额外存储完整的镜像文件，是为了在收到对特定文件的请求时，可以快速地建立连接进行传输，而不是等到解压结束以后才能进行。远程仓库会根据请求的类型，建立不同的传输方式。当收到传统的镜像传输请求时，它与一般的 Docker 镜像仓库工作方式无异。当收到 D4C 系统的请求时，远程仓库会根据请求的文件列表来单独传输所需的文件，而不用传输大量的无用文件。

另外，RemoteRegistry 会接收 LocalRepository 中以 recordmode 运行的 LoadComponent 所发回的文件访问顺序。在 RemoteRegistry 中有一项设置，用于判定容器启动多长时间以内调用的文件是启动必须的。远程仓库会以此设置与收到的文件访问顺序为依据，不断更新 bootfilelist，也就是启动容器时所必须的镜像文件列表。

在不使用 D4C 的方式拉取镜像时，远程仓库的地址可以不必是定制的 D4CRemoteRegistry，这也保证了 D4C 方式与传统镜像拉取方式的最大兼容。

6. 实验结果与分析

本节我们以 D4C 系统与传统的 Docker 仓库进行容器启动的性能对比。主要从容器冷启动时间、启动镜像大小、网络传输量三个方面进行测试，实验环境为两台机器，一台用于配置容器远程仓库，一台用于配置容器客户端，具体配置如下：

1) 处理器：8 核 Intel Xeon E5 2.20GHz
2) 内存：32GB LPDDR3
3) 硬盘：256GB PCIeNVMe 高速固态硬盘
4) 操作系统：Centos 7.6.1810 64 位
5) Docker Server：Community 19.03.13

6.1. 冷启动时间

本小节对 D4C 和 Docker 容器引擎冷启动的时间进行了测试。容器启动时间从发出命令开始计算，以成功执行完毕该镜像所能提供的最基本操作作为结束标志。通过容器启动时输出的时间戳和测试程序输出的时间错可以精确地计算出容器启动花费的时间。两者均保证在开始发出容器启动命令时，在本地仓库中不存在任何对应的镜像缓存。

之所以要保证发出容器启动命令时本地仓库没有缓存，是由于 Docker 会自动利用共享镜像层来进行镜像拉取加速。而我们所要探究的是冷启动的时间。在实际场景中，需要部署容器的机器很可能是由于应用的扩容需要刚刚扩展的，并不会有镜像层缓存。

我们在相同的系统环境下分别使用 D4C 和 Docker 进行了冷启动时间测试，结果如图 7 所示。可以看到，大部分的容器在改为使用 D4C 进行分发以后，冷启动时间都缩短了超过 50%。系统类镜像(ubuntu)缩减的相对值最多，大概在 75%。语言类(Python)、Web framework 类(node)的镜像缩减的绝对值最多。这是由于 D4C 是通过延迟加载镜像，优先传输启动必须的文件，提高容器的启动速度，而语言类 web framework 类的镜像在开始运行时也要消耗较多的时间。这部分的时间不是本文所研究的内容，因此不受影响。

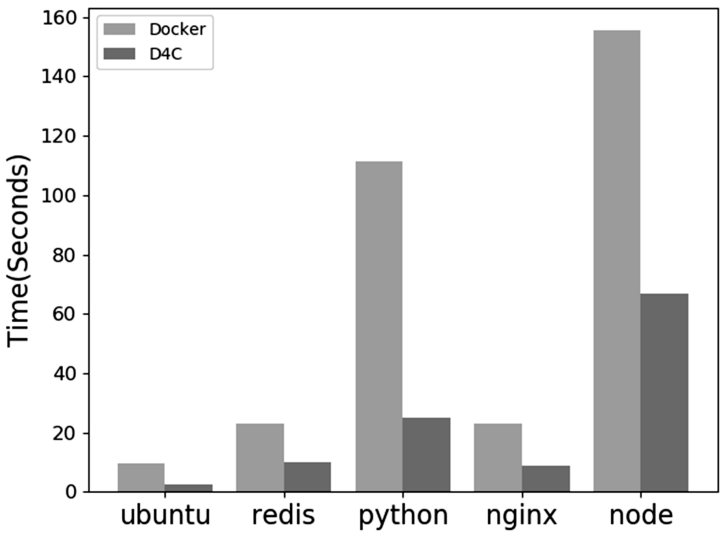

Figure 7. Comparison of cold start time
图 7. 冷启动时间对比

6.2. 启动镜像大小

镜像的大小决定了容器传输所消耗的时间，越小的镜像在存储时也占用更少的磁盘空间。D4C 采用的是镜像文件延迟加载的模式，因此本地存储的镜像最终形态会与 Docker 引擎的原始镜像保持一致。但是在启动时，D4C 仅仅传输必要的文件，因此在启动镜像上，要远远小于原始的容器镜像。

下图 8 展示了各个实验的容器镜像的原始大小与 D4C 优化后的启动镜像大小对比。所有的镜像均有不同程度的大小缩减。对于不同镜像，这个数值在 85%左右浮动。由于启动所需的文件只占镜像总数的很小一部分，因此启动镜像可以缩减到非常小的体积。

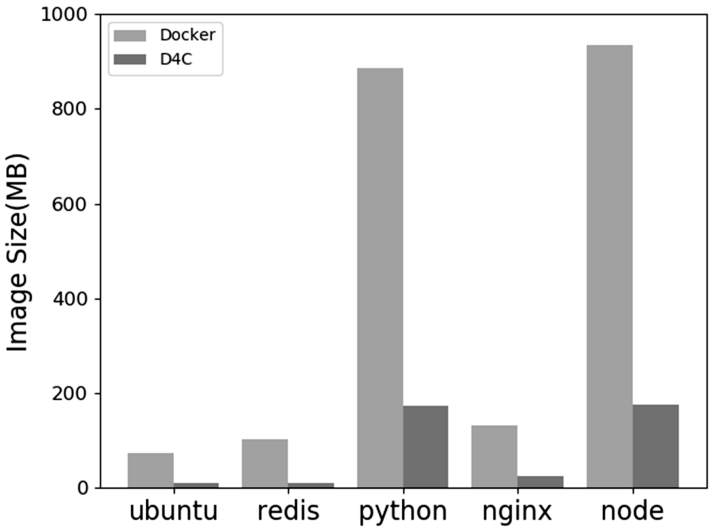

Figure 8. Comparison of boot image size
图 8. 启动镜像大小对比

进行对比以后发现，较小的镜像文件冗余度较少，启动时镜像内的文件访问率高，但启动镜像的缩减比例反而大。因此可以看到这样一个趋势，越大的镜像，文件使用率越低，但启动镜像的缩减比例会越小。这可能是由于体积较大的镜像往往是更定制化的镜像，相对于底层的镜像，会多出许多较大的文件。例如 node 这样的网页框架，会加载非常大的文件；而 ubuntu 这样的系统镜像，文件都较小。

但尽管大镜像体积削减的比例较小，但绝对值仍然是很大的。在实际应用中，很少会直接部署 Linux 基础的系统镜像，而是需要在其上进一步添加镜像层，构建出许多不同的应用。而我们的实验表明，对于这样的镜像，体积可以削减非常大，减少大量传输镜像文件的时间。

6.3. 网络传输

D4C 镜像传输系统与原始的 Docker 进行镜像传输时的网络消耗如图 9 所示。可以看到，原始 Docker 在启动的开始阶段就产生了大量的网络数据传输，在后续的过程中则几乎没有产生额外的网络请求。而 D4C 的网络传输呈阶梯状，且初始的网络消耗较小。最终的网络消耗，D4C 总是要小于 Docker。

这符合两个系统不同的镜像传输策略。原始 Docker 在启动之前需要拉取完整的镜像，而在之后则不需要进行额外传输。而 D4C 一开始只获取少量的镜像文件，随着容器的运行逐渐要求更多镜像层，并且由于阈值触发的模式，网络请求呈现出明显的阶段性变化。在容器运行足够长时间的情况下，容器镜像内的文件理应会被逐一访问，此时 D4C 的总体的网络传输应当会大于传统 Docker 方式。因为最终传输的都是一个完整的镜像，D4C 是以文件形式传输，而 Docker 是以压缩后的镜像层传输，未压缩的文件一般是压缩后镜像的 2~3 倍。但是我们观察到的是 D4C 的流量消耗总是小于 Docker。这是因为，容器镜像内的文件在启动时仅使用了一小部分，而运行时所访问的文件也是跟实际镜像的类型相关联的，有的镜像本身就包含很多冗余文件。这些文件即使在容器长时运行的情况下，被访问到的概率也不大。

这更进一步说明了 D4C 的意义。首先，起始阶段的网络消耗不大，就说明相同的带宽，D4C 可以支

持更多机器的镜像传输。相对于原始 Docker，理论上可以支持更多的并发。其次，最终的传输量也是 D4C 较少。这说明其设计的思路是正确的，很多文件在容器运行并不是必须的，减少这部分文件的传输，与延迟加载非启动必须的文件，是加速容器启动的可行方法。

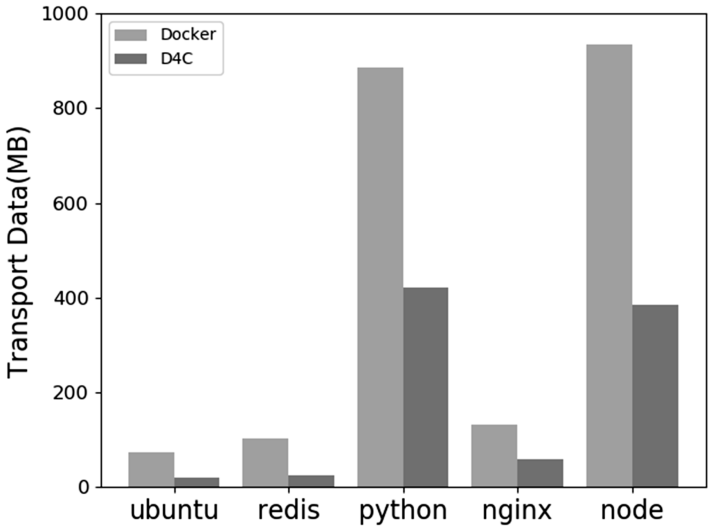

Figure 9. Comparison of network transportation
图 9. 网络传输情况对比

7. 总结与展望

本文提出了一种针对大规模容器镜像分发的加速模型并将其实现。该模型对镜像的分层重新进行了设计，以容器运行时的访问顺序而非镜像构建时的创建顺序来加载文件。通过延迟加载非启动必须的镜像文件，并且细化了镜像传输的粒度，加速了容器镜像的传输，从而加速了容器的启动。基于上述方法以及对原始 Docker 引擎的改造，开发了大规模镜像分发系统 D4C。实验表明，相对于传统的 Docker 镜像传输方式，D4C 缩短了冷启动时间，缩减了启动镜像的大小，并减少了网络传输的消耗。

本文实验为验证 D4C 系统在磁盘 I/O、CPU 和内存利用率等方面的性能表现，这部分工作将在未来给予补充。

参考文献

[1] Sharma, P., Chaufournier, L., Shenoy, P., *et al.* (2016) Containers and Virtual Machines at Scale: A Comparative Study. In: *Proceedings of the* 17*th International Middleware Conference*, ACM, Article No. 1. https://doi.org/10.1145/2988336.2988337

[2] Merkel, D. (2014) Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, **239**, 2.

[3] Harter, T., Salmon, B., Liu, R., *et al.* (2016) Slacker: Fast Distribution with Lazy Docker Containers. 14*th USENIX Conference on File and Storage Technologies* (*FAST*'16), Santa Clara, 22-25 February 2016, 181-195.

[4] Burns, B., Grant, B., Oppenheimer, D., *et al.* (2016) Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade. *Queue*, **14**, No. 1. https://doi.org/10.1145/2898442.2898444

[5] Reich, J., Laadan, O., Brosh, E., *et al.* (2012) VMTorrent: Scalable P2P Virtual Machine Streaming. *CoNEXT*, **12**, 289-300. https://doi.org/10.1145/2413176.2413210

[6] Chowdhury, M., Zaharia, M., Ma, J., *et al.* (2011) Managing Data Transfers in Computer Clusters with Orchestra. *ACM SIGCOMM Computer Communication Review*, **41**, 98-109. https://doi.org/10.1145/2043164.2018448

[7] Wang, K.J., Yang, Y., Li, Y., *et al.* (2017) FID: A Faster Image Distribution System for Docker Platform. 2017 *IEEE* 2*nd International Workshops on Foundations and Applications of Self* Systems* (*FAS* W*), Tucson, 18-22 September 2017, 191-198. https://doi.org/10.1109/FAS-W.2017.147

[8] Nathan, S., Ghosh, R., Mukherjee, T., *et al.* (2017) CoMICon: A Co-Operative Management System for Docker Container Images. 2017 *IEEE International Conference on Cloud Engineering* (*IC2E*), Vancouver, 4-7 April 2017, 116-126. https://doi.org/10.1109/IC2E.2017.24

[9] Liang, M., Shen, S., Li, D., *et al.* (2016) HDID: An Efficient Hybrid Docker Image Distribution System for Datacenters. In: Zhang, L. and Xu, C., Eds., *Software Engineering and Methodology for Emerging Domains. NASAC* 2016. *Communications in Computer and Information Science*, Vol. 675, Springer, Singapore, 179-194. https://doi.org/10.1007/978-981-10-3482-4_13

[10] Tarasov, V., Rupprecht, L., Skourtis, D., *et al.* (2017) In Search of the Ideal Storage Configuration for Docker Containers. 2017 *IEEE* 2*nd International Workshops on Foundations and Applications of Self* Systems* (*FAS* W*), Tucson, 18-22 September 2017, 199-206. https://doi.org/10.1109/FAS-W.2017.148

[11] Nguyen, T.L., Nou, R. and Lebre, A. (2019) YOLO: Speeding up VM and Docker Boot Time by Reducing I/O Operations. In: Yahyapour, R., Ed., *Euro-Par* 2019: *Parallel Processing. Euro-Par* 2019. *Lecture Notes in Computer Science*, Vol 11725, Springer, Cham, 273-287. https://doi.org/10.1007/978-3-030-29400-7_20

[12] Hardi, N., Blomer, J., Ganis, G. and Popescu, R. (2018) Making Containers Lazy with Docker and CernVM-FS. *Journal of Physics*: *Conference Series*, **1085**, 032019. https://doi.org/10.1088/1742-6596/1085/3/032019

[13] Cheng, Y., Gupta, A. and Butt, A.R. (2015) An In-Memory Object Caching Framework with Adaptive Load Balancing. In: *Proceedings of the Tenth European Conference on Computer Systems*, ACM, Article No. 4. https://doi.org/10.1145/2741948.2741967

[14] Anwar, A., Cheng, Y., Gupta, A., *et al.* (2015) Taming the Cloud Object Storage with Mos. In: *Proceedings of the* 10*th Parallel Data Storage Workshop*, ACM, 7-12. https://doi.org/10.1145/2834976.2834980

[15] Butt, A.R., Gniady, C. and Hu, Y.C. (2005) The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. *ACM SIGMETRICS Performance Evaluation Review*, **33**, 157-168. https://doi.org/10.1145/1071690.1064231

[16] Cao, P., Felten, E.W., Karlin, A.R., *et al.* (1996) Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems* (*TOCS*), **14**, 311-343. https://doi.org/10.1145/235543.235544

[17] De, P., Gupta, M., Soni, M., *et al.* (2012) Caching VM Instances for Fast VM Provisioning: A Comparative Evaluation. European Conference on Parallel Processing. In: Kaklamanis, C., Papatheodorou, T. and Spirakis, P.G., Eds., *Euro-Par* 2012 *Parallel Processing*. *Euro-Par* 2012. *Lecture Notes in Computer Science*, Vol 7484, Springer, Berlin, Heidelberg, 325-336. https://doi.org/10.1007/978-3-642-32820-6_33

[18] Zhang, Z., Li, D. and Wu, K. (2016) Large-Scale Virtual Machines Provisioning in Clouds: Challenges and Approaches. *Frontiers of Computer Science*, **10**, 2-18. https://doi.org/10.1007/s11704-015-4420-7

[19] Razavi, K., Van Der Kolk, G. and Kielmann, T. (2015) Prebaked µVMs: Scalable, Instant VM Startup for IaaS Clouds. 2015 *IEEE* 35*th International Conference on Distributed Computing Systems*, Columbus, 29 June-2 July, 245-255. https://doi.org/10.1109/ICDCS.2015.33

[20] Anwar, A., Mohamed, M., Tarasov, V., *et al.* (2018) Improving Docker Registry Design Based on Production Workload Analysis. 16*th USENIX Conference on File and Storage Technologies* (*FAST* '18), Oakland, February 2018, 265-278.

[21] Littley, M., Anwar, A., Fayyaz, H., *et al.* (2019) Bolt: Towards a Scalable Docker Registry via Hyperconvergence. 2019 *IEEE* 12*th International Conference on Cloud Computing* (*CLOUD*), Milan, 8-13 July 2019, 358-366. https://doi.org/10.1109/CLOUD.2019.00065

[22] Du, L., Wo, T., Yang, R. and Hu, C.M. (2017) Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. 2017 *IEEE* 19*th International Conference on High Performance Computing and Communications*; *IEEE* 15*th International Conference on Smart City*; *IEEE* 3*rd International Conference on Data Science and Systems* (*HPCC*/*SmartCity*/*DSS*), Bangkok, 18-20 December 2017, 332-339. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.44

[23] Zheng, C., Rupprecht, L., Tarasov, V., *et al.* (2018) Wharf: Sharing Docker Images in a Distributed File System. In: *Proceedings of the ACM Symposium on Cloud Computing*, ACM, 174-185. https://doi.org/10.1145/3267809.3267836

[24] Thalheim, J., Bhatotia, P., Fonseca, P., *et al.* (2018) Cntr: Lightweight OS Containers. 2018 *USENIX Annual Technical Conference* (*USENIXATC*'18), Boston, 11-13 July 2018, 199-212.

[25] Zhao, N., Tarasov, V., Anwar, A., *et al.* (2019) Slimmer: Weight Loss Secrets for Docker Registries. 2019 *IEEE* 12*th International Conference on Cloud Computing* (*CLOUD*), Milan, 8-13 July 2019, 517-519. https://doi.org/10.1109/CLOUD.2019.00096