/

四川大學

本科生毕业设计（技术报告）

题　　目 基于golang的docker管理系统

学　　院 软件学院

专　　业 软件工程

学生姓名 孙林桐

学　　号 2014141463177 年级 2014级

指导教师 李晓华

教务处制表

二〇一八年五月二十日

基于 golang 的 docker 管理系统

软件工程

学生 孙林桐 指导老师 李晓华

[摘要] 随着 IT 行业的发展，越来越多、各种各样的应用被开发了出来，在产品的维护过程中，许多企业都需要面对这两个问题：

（1）如何更高效、简单地管理应用环境，以让团队更加专注于产品的研发？

（2）如何建立一个内部研发人员共享的平台，以便研发者能对产品的每一个版本、每一个依赖插件一目了然？

2013 年 docker 的诞生，让应用的开发者们能更专注于自己产品的研发，而不再在各种繁琐的环境上耗费太多精力，算是解决了第一个问题，本项目的目标则是在使用 docker 的情况下，解决第二个问题。让所有研发人员能清晰明了地了解到企业服务器上的 docker 镜像、容器状况，一键式操作直白地看到产品及其依赖的运行情况，简单、高效地查看产品日志，分析错误原因。

本项目主要内容：

（1）在服务器上（Linux 系统）建立一个多用户的 docker 管理平台，提供网页服务。

（2）用户分为普通用户和管理员用户，每个用户都有自己的工作空间，工作空间之间隔离。每个用户在平台上的所有操作都被记录，记录只有自己和管理员可见，生成的镜像、容器也只有自己和管理员可见。因此，平台管理员可查看服务器所有资源与用户操作记录，便于企业管理人员查看每个员工的工作记录。

（3）平台具体功能主要包括：用户信息管理；用户操作日志管理；镜像的拉取、创建、删除、打标签、生成容器；容器的创建、删除、启动、终止、提交等；容器日志的一键输出查看（日志筛选）。

项目开发语言：后端：golang，前端：html5 , javascript , css

项目开发框架：后端：beego，前端：Amaze UI

主要技术：go web 编程、并发编程、session 会话控制、路由过滤器、数据库操作及事务处理、ajax 前后端交互、前端动态页面、git 版本控制、docker go API 库使用、mysql docker 镜像使用

经过测试，本系统很好地实现了项目的所有功能，且具有较高稳定性。但功能上稍有不足，实现的大多是基本操作，诸多复杂的 docker 操作还没实现。不过，作为 1.0 版本，本系统还是有较大应用价值的。因为随着云计算 2.0 时代的到来，docker 作为主流技术，已吸引了越来越多的企业将自己的产品往 docker 中迁移，对这些企业，一个简洁明了的、用于企业内部使用的、建立在内部服务器上的 docker 管理（也包括用户及日志管理）的平台是有较大价值的。

[关键词] docker, golang，工作空间，镜像，容器，日志

Docker Management System Based on Golang

Software Engineering

Student： SUN Lin-tong Adviser: LI Xiao-hua

[Abstract] With the development of the IT industry, more and more applications have been developed. In the process of product maintenance, many enterprises face these two problems：

（1）How to manage the application environment more efficiently and simply so that the team can focus more on product development ?

（2）How to build a shared platform for internal developers so that they can clearly see every version and every plug-in of products?

The birth of docker in 2013 allows the developers to focus more on their own product development, instead of consuming too much energy in a variety of tedious environments. The first problem is solved. The goal of this project is to solve second problems with the use of docker. Let all R & D personnel clearly understand the docker image and container status on the enterprise server clearly and clearly. One key operation is straightforward to see the operation of the product and its dependencies, simply and efficiently view the product log and analyze the cause of the error.

The main contents of this project:

(1) build a multi-user service docker management platform on the server (Linux system), providing webpage services.

(2) users are divided into ordinary users and administrators users. Each user has his own working space and the isolation between workspaces. All operations on the platform are recorded by each user. The records are visible only by themselves and the administrators, and the generated images and containers are also visible only by themselves and the administrators. Therefore, the platform administrator can see all resources and user operation records of the server, so that business managers can see every employee's work records.

(3) the specific functions of the platform include: user information management, user operation log management, mirror image pulling, creating, deleting, tagging, generating containers; container creation, deletion, startup, termination, submission, etc.; the one key output view of the container log (log screen selection)

Project development language: back-end: golang front-end: HTML5, JavaScript, CSS.

Project development framework: back-end: beego front-end: Amaze UI.

Main technologies: go web programming, concurrent programming, session session control, routing filter, database operation and transaction processing, AJAX front and back end connection, front end dynamic page, GIT version control, docker go API library use, MySQL docker mirror image use

After testing, this system has basically achieved all the functions of the project, but there are still some shortcomings in the design, such as multi user server concurrency, which still needs to be solved. However, as the 1 version I developed, it still has great application value. With the advent of the 2 era of cloud computing, docker, as a mainstream technology, has attracted more and more enterprises to migrate their products to docker. For these enterprises, a concise, internal use of the enterprise, docker Management (also including user and log management) on the internal server. The platform is of great value.

[Key Words] docker, golang , work space, image, container, log

目　录

1 绪 论

1.1 引言

随着 IT 行业的发展，越来越多、各种各样的应用被开发了出来，在产品的维护过程中，许多企业都需要面对这两个问题：

（1）如何更高效、简单地管理应用环境，以让团队更加专注于产品的研发？

（2）如何建立一个内部研发人员共享的平台，以便研发者能对产品的每一个版本、每一个依赖插件一目了然？

2013 年 docker 的诞生，算是解决了第一个问题。Docker 在 Linux 操作系统和应用之间建立了额外的软件抽象层，它可以打包应用程序及其相关依赖，也可以打包其他容器，可以在任何 Linux 服务器上运行，有相当高的灵活性和便携性。让应用程序能在任何地方运行，无论是单机还是公有云、私有云，让开发团队不用投注大量精力在运行环境的搭建上，也不用担心应用的迁移[1]。

为解决第二个问题，需要建立一个 docker 的管理平台。这个平台能让所有研发人员清晰明了地了解到企业服务器上的 docker 镜像、容器状况，一键式操作它们，直白地看到产品及其依赖的运行情况，简单、高效地查看产品日志，分析错误原因。

由于这是一个面向企业的系统，用于运行在企业的服务器上，管理服务器的镜像、容器、日志。需要设计两类用户，一类是管理员（技术总监、项目经理使用），拥有查看所有用户操作记录、资源使用情况的权限，以便于了解产品运行状态与开发人员的进度。一类是普通用户（开发人员使用），只能看到 public 工作空间和自己的生成的镜像、容器、日志，只能看到自己的操作记录，各个用户创建的资源通过数据库分隔开。

随着云计算 2.0 时代的到来，docker 作为新兴的主流技术之一，已吸引了越来越多的企业将自己的产品往 docker 中迁移，对这些企业，一个简洁明了的、用于企业内部使用的、建立在内部服务器上的 docker 管理（也包括用户及日志管理）的平台是有较大价值的。这个平台没有实现容器编排、调度相关的策略，可以理解为它只是单纯的作为一个 docker 相关资源的管理平台。

1.2 技术现状

1.2.1 Docker 使用情况

Docker 作为一项新兴技术，在国内外影响很大。Datadog 在 2017 年做了一份调研报告[2]，该报告汇编了来自 10,000 家公司和 1.85 亿个容器的样本的使用数据，几乎是有史以来发布的最大和最准确的 Docker 采用审查，报告主要内容如下：

（1） 2016 年 3 月初，Datadog 的客户中有 13.6%采用了 Docker。一年后这一数字增长到 18.8%。这 12 个月内几乎达到了 40%的市场份额增长

（2） 两年前，Docker 拥有约 3%的市场份额，现在它运行在 Datadog 所监控主机中的 15%

（3） 随着这些公司使用 docker，docker 日益成为他们生产环境不可分割的一部分，因此他们正在寻找帮助他们有效管理和编排容器的工具。截止 2017 年 3 月，大约 40%的运行 Docker 的 Datadog 的客户还运行 Kubernetes, Mesos, Amacon ECS, Google Container Engine 或其他管理工具

（4） Docker 采用者在第一个月到第十个月的使用期间，其生产的平均容器数量增长了接近 5 倍。且这种内部使用增长率是非常线性的，在第十个月之后没有显示出减少的迹象。

（5） 采用 Docker 的中位数公司在每台主机上同时运行七个容器，高于九个月前的五个容器。这一发现似乎表明，Docker 事实上通常被用作共享计算资源的轻量级计算方式。它不仅仅是为了提供可知的，版本化的运行时环境而被重视。

从该报告可以看出，近几年来，越来越多的企业开始使用 Docker, 随着使用时间的增加，他们与 docker 日益不可分割，使用 docker 的许多企业还在寻找容器编排与管理工具。

1.2.2 Docker 平台产品调研

近几年，随着着 docker 的诞生，许多 docker 管理、容器编排的云平台应运而生，比较知名的有：

表 1.1 国内外 Docker 服务提供商/产品

国外	开源	kubernetes, Swarm, Mesos,Rancher
	闭源	tutum.co, cloud66 , stackengine
国内		ghostcloud, cSphere, daocloud, aluadacloud

这里就其中比较具有代表性两款产品进行简要介绍，一个是开源的 kubernetes，一个国内 ghostcloud 的产品 EKOS

1）Kubernetes

Kubernetes 是一个大型的分布式操作系统，其操作的对象以容器为核心，围绕容器编排又抽象出了一层概念如 pod,deployment,service 等等。通过 kubernetes, 开业人员可以轻松做到快速部署应用；快速扩展应用；无缝对接新的应用功能；节省资源，优化硬件资源的使用等等 [3]。

其具体的功能颇多且复杂，它是一个面向计算机集群的容器管理系统，它能调度应用容器使之在各个主机节点之间自动迁移，避免主机发生意外（如断电、资源丢失、中毒等等）而导致的应用服务中止，也会为应用服务器创建负载均衡，把服务分摊到多个操作单元上进行执行[4]，兼容主流日志系统 elasticsearch，主流监控系统 prometheus 。

Kubernetes 有着庞大的社区，其功能也是日益完善，但它并没有做完 docker 相关的所有工作，因为它用自己的 api 封装了 docker 的 api，隐藏了 docker 的许多细节，不算是一个直接将 docker 暴露给用户的管理平台，而是一个在 docker 之上、让开发人员将之当做一个分布式操作系统来管理他们的应用、自动化容器编排的平台。

2）EKOS

Ghostcloud 的产品 EKOS 的底层也是 kubernetes（国内绝大多数相关产品都是如此），在 kubernetes 之上，添加了许多插件实现了：镜像仓库管理集群镜像、负载均衡器、集群网络管理、集群日志、资源监控、一键部署微服务等功能。功能比较完备，面向的是传统 IT 企业，帮助他们将产品往云平台上迁移。而 docker 的操作还是得用户自己来做，包括镜像、容器的管理，容器的日志操作。

综合国内外的行业现状，搭建一个专注于管理 docker 镜像、容器的平台是有价值的。这个平台需要能有效的操作镜像、容器，简单直白地展示系统资源，一键式的查看日志。在这些基础上，我开始了 docker 管理系统的开发。

1.3 主要工作

1.3.1 论文工作

（1）开发环境配置说明及开发工具的介绍
（2）需求分析，确立工作内容
（3）系统设计，根据需求详细地设计系统具体功能
（4）用户界面设计，配合系统设计以在前端设计实现所有功能点
（5）系统具体实现，分模块介绍系统流程
（6）界面展示，展示开发完成后的系统所有关键界面及其相应情况
（7）展示测试方法及测试结果

1.3.2 项目工作

（1）在本地建立网页服务，通过用户的注册登录进入应用容器管理平台，用户分权限
（2）用户访问控制与会话控制
（3）系统外镜像、容器资源监测与同步，资源隔离
（4）管理各种操作所产生的镜像、容器，在网页上实现操作镜像、容器的功能
（5）建立平台日志系统，分权限展示
（6）容器日志管理

1.4 论文组织与结构

本论文主体部分由六部分组成，分别为：
第一部分：绪论，介绍本系统的开发背景。
第二部分：相关知识和技术介绍，介绍本系统用到的关键知识技术。
第三部分：系统需求分析，分析并确定本系统需要实现的功能。
第四部分：系统设计，设计本系统所有功能点。

第五部分：系统实现。

第六部分：系统测试。

第七部分：结束语。

2 相关知识和技术介绍

2.1 系统预备知识简介

本系统是对较前沿技术 docker 的一次简单应用，采用的编程语言 Go 也才出生不到十年，这里就简略介绍一下这两项技术的特点。

2.1.1 Docker

简单来说，Docker 就是一个能够封装整个应用运行环境的容器[5]。Docker 容器模拟了一个 Linux 系统。我们可以把它当做一个跑在 Linux 系统之上的又一层系统，不过这样使用毫无意义。Docker 的设计初衷就是一个应用对应一个容器，将不同应用的运行环境分离开来，将应用高度解耦，减少相互之间的依赖[6]。这样，当我们为一款应用制作好了一个 docker 镜像后，我们再也不用担心该应用的运行环境、迁移等问题[7]。因为，在任何 Linux 操作系统上，只要上面装有 docker，我们就能通过一行命令将之运行起来。不仅如此，Docker 有相当强大的命令行工具，足以对容器内的应用实现几乎所有操作[8]。本系统的所有 docker 相关的后端函数都是模仿 docker 命令行工具、对 docker 客户端 API 进行一次封装而实现 [9]。

2.1.2 Go

Go 是目前云计算领域比较热门的编程语言，包括最为知名的 docker 和 kubernetes 项目都是由它编写。它有许多优点，最著名的是：

（1）性能上：Go 编译后的程序，其运行速度可以媲美 C/C++

（2）并发上：Go 通过 Goroutine 可以轻松实现高并发，一个 Go 程序可以运行超过数万个 Goroutine

但这些不是我选用 Go 的主要原因，它真正吸引我的是简洁的编程模式和独特的设计理念：

（1）Go 的库不多，没有像 java 那样庞大的工具库，但也能较轻松的实现应该实现的功能

（2）简洁的面向对象，没有诸多面向对象的关键字，也没有继承和重载，却能通过结构体等方式实现这些功能，非常简洁

（3）接口不用显示实现，一个对象实现了接口所有方法便隐式实现了该接口 [10]

随着学习的深入，我逐渐喜欢上了 Go 的编程模式。

2.1.3 Beego

Beego 是国人开发的一个可以快速开发 Go 应用的 HTTP 框架，它能够用来快速开发 API、Web 及后端服务等各种应用，是一个 RESTful 框架 [11]。

Beego 是一个典型的 MVC 架构，执行逻辑如下图：

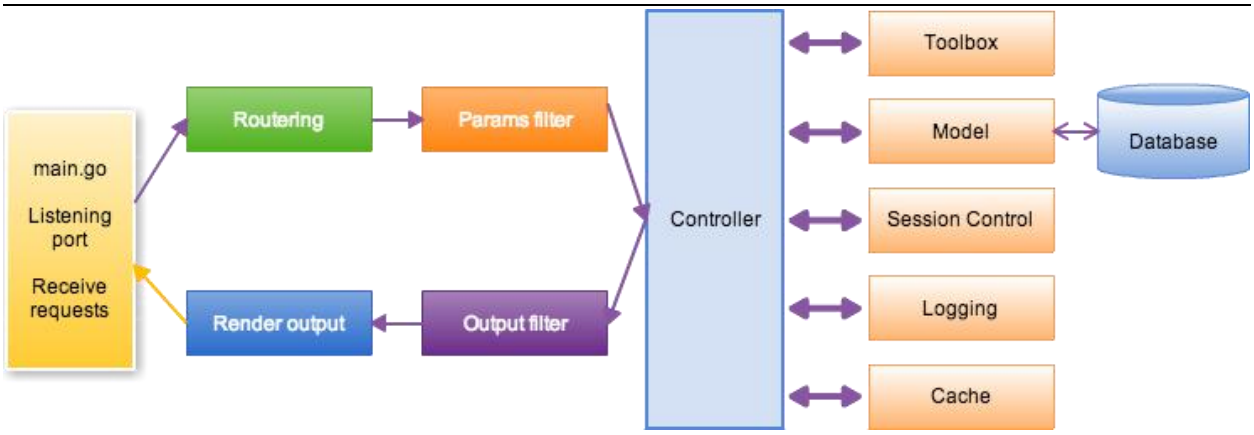

图 2.1 beego 执行逻辑

2.2 开发环境及工具说明

2.2.1 开发环境

简要说一下本项目的开发环境，由于本项目只能运行在 Linux 上，在编写部分后端代码及编译测试时，基本都是在 Cent OS（一个 Linux 发行版）上进行的，需要配置的东西较多，而 windows 上主要写前端代码和部分后端代码。

1）Windows 开发环境（Window7）

1. 安装 Microsoft VS Code 代码编辑工具
2. 安装 Go 工具包
3. 为 Microsoft VS Code 安装 Go 插件
4. 安装 Windows Git 工具，作为版本控制系统
5. 用 go get 命令获取项目所依赖的外部包，如 beego
6. 安装 Chrome 作为浏览器

2）Linux 开发环境（Cent OS7）

1. yum install vim 安装 vim 代码编辑工具
2. 为 Vim 安装 Go 插件
3. yum install git 安装 git
4. 安装 Go 工具包
5. 用 go get 命令获取项目所依赖的外部包，如 beego

2.2.2 数据库和 docker

本项目的数据库环境通过官方的 mysql 镜像来部署，这里我通过 mysql 的 docker 镜像的使用来介绍一下 docker 一些使用方法，同时也介绍怎样配置本项目数据库环境（Cent OS 7 系统）。

（1）service docker start; systemctl enable docker 启动 docker 服务，并设为开机自启

（2）docker pull docker.io/mysql:latest

1. docker pull 拉取镜像

2. docker.io 是镜像仓库名

3. mysql 是镜像名

4. latest 是镜像标签，表示拉取最新的 mysql 镜像

（3）docker run - -name slt-mysql -d -i -t -p 3306:3306 -e MYSQL_ROOT_PASSWORD =hbuckwpzg mysql:latest bash

1. docker run 运行一个容器

2. - -name 定义容器名为 slt-mysql

3. -d 后台运行，-i 以交互模式运行容器，-t 为容器分配一个伪输入终端

4. -p 添加容器与主机的端口映射（mysql 的端口是 3306）

5. -e 为容器添加环境变量（MYSQL_ROOT_PASSWORD 是官方镜像要求的 mysql root 用户密码环境变量的 key）

6. mysql:latest 是容器的源镜像

7. bash 是在容器启动后在其中执行的命令

这样，一个 mysql 容器就在后台运行起来了，我们可以通过本地的 3306 端口访问它。

（4）docker exec -i -t slt-mysql bash

1. docker exec 进入一个运行中的容器

2. -i 以交互模式运行容器，-t 为容器分配一个伪输入终端

3. slt-mysql 是容器名（也可以用容器 id）

4. bash 是执行的容器命令

（5）mysql -u root -p hbuckwpzg 在容器中登录 mysql

（6）CREATE DATABASE myapp 创建数据库 myapp,至此数据库环境已配好

2.2.3 配置文件

本系统配置文件为 myapp/conf/app.conf,文件中比较重要的是 mysql 相关配置，用于连接数据库：

表 2.1 数据库相关配置

参数	说明
mysqlhost = "localhost"	mysql 服务 ip
mysqlport = "3306"	mysql 服务端口号
mysqluser = "root"	mysql 用户名
mysqlpassword = "hbuckwpzg"	mysql 密码
mysqldatabase = "myapp"	所用数据库名

2.3 本章小结

本章简要说明了项目的主要技术、开发环境搭建及开发工具的安装过程。技术方面虽然用的是比较新的语言 Go，但有 java 基础的话上手还是比较快的，用一段时间就会被它的种种特性所吸引。核心工具 Docker 上手简单，但实际用起来比较复杂，因为它不仅有许多概念需要深入理解，还需要对 Linux 操作系统十分熟练。Docker 有着很强大、复杂的命令行工具，需要时间来掌握，也有让人头疼的 Dockerfile 的编写（本项目中没有涉及），因此简化 docker 操作也是我开发此项目的意图之一。

本章相对比较复杂的是部署 mysql。但它其实已经非常简单，这个过程中，我们可以很明显地感受到 docker 的便捷：整个过程完全不用关心宿主机的 Linux 系统类型，仅仅几行 docker 命令便部署成功，相比于传统方式（下载、编译、安装）快捷、方便了许多，使用起来也和直接在宿主机上安装的 mysql 相差无几。

3 系统需求分析

3.1 系统需求概述

首先，本系统针对一个企业所有开发人员与其管理人员，故需要设计为多用户、分权限的管理系统。普通用户之间分资源内容实现资源的隔离与共享，便于开发人员之间的独立与协作。管理员有权查看、操作所有资源。

其次，既是多用户、分权限的系统，需要对所有用户进行操作记录，并分权限进行展示。

然后，作为一个 docker 管理系统，当然需要实现 docker 镜像、容器的基本操作，如创建、删除、启动、停止、便签等功能。

最后，docker 容器内运行着的是应用，只是操作 docker 是很难满足用户需求的，所以简单明了得展示容器内应用的日志。

综上分析，本系统主要需要具有以下几点功能：

1. 用户注册登录，用户分权限
2. 用户访问控制与会话控制
3. 系统外镜像、容器资源监测与同步，资源隔离
4. 镜像、容器的多种操作
5. 建立平台日志系统，分权限展示
6. 容器日志管理

3.2 系统功能需求分析

3.2.1 用户登录注册功能

1）注册账号

1. 用户填写注册信息

   （1）注册用户名

   （2）用户密码

   （3）是否注册为管理员（可选）

2. 系统操作部分

   （1）判断注册账号是否合法（是否可以注册为管理员，是否已存在该用户，输入数据是否合法）

   （2）注册成功时生成当前时间戳作为用户信息的一部分，将用户的输入密码用加密函数加密后插入数据库 user 表

   （3）生成用户注册的每一种操作日志并插入数据库 log 表

2）用户登录

1. 用户登录输入信息
   （1）用户名
   （2）密码
2. 系统操作部分
   （1）判断是否能登录成功（是否有该用户、密码是否正确、输入信息是否合法）
   （2）登录成功时设置用户 session 为该用户（session 在访问和会话控制中再作介绍）
   （3）生成用户登录的操作日志，插入数据库 log 表
   （4）登录成功后跳转至系统首页（/index 页面）

3）用户信息展示

管理员展示所有用户信息，非管理员展示当前用户信息：

（1）用户名

（2）注册时间

（3）用户类型

（4）按钮（查看该用户日志）

3.2.2 用户访问控制与会话控制

1）访问控制（访问过滤器）

1. 登录前
   （1）用户请求/register 和/login 页面正常跳转
   （2）用户请求它们之外的任何页面时，跳转至/login 页面
2. 登录后，用户可以正常请求任何界面
3. 退出系统后，同登录前

2) 会话控制

1. 用户登录前服务器无会话
2. 用户登录后，服务器记录该用户的用户名和用户类型至 session，在所有页面上显示当前用户名和用户类型
3. 用户所有操作被服务器以日志方式记录，用户名为 session 中的用户名
4. 退出后终止会话，删除用户 session

3.2.3 平台日志系

1）获取平台日志

1. 按用户名筛选
2. 按请求筛选

（1）GET 请求：若请求源为管理员，展示若有用户日志；若请求源为普通用户，展示当前用户日志

（2）POST 请求：这种情况发生在用户在主页点击查看某用户日志，传入请求用户名，直接展示该用户日志，不分是否为管理员（因为非管理员界面只有查看自己日志的按钮）

2）生成平台日志

1. 日志记录信息

（1）用户名

（2）日志生成时间

（3）日志内容

2. 系统生成日志

（1）获取当前时间

（2）获取当前用户名

（3）获取日志内容（错误信息、提示信息）

（4）插入数据库

3.2.4 镜像、容器资源管理

1）资源信息

1. 镜像

（1）镜像 ID

（2）镜像分组（分组若为 public，则所有用户可见、可操作；其他分组均为私有，仅该用户可见、可操作）

2. 容器

（1）容器 ID

（2）容器分组（分组若为 public，则所有用户可见、可操作；其他分组均为私有，仅该用户可见、可操作）

2）初始化系统资源

1. 镜像

（1）系统建立时，生成在数据库中 image 表，清空其中信息

（2）清空后获取本地所有镜像信息，将 ID 和分组名（Public）插入 image 表

2. 容器

（1）系统建立时，生成在数据库中 container 表，清空其中信息

（2）清空后获取本地所有容器信息，将 ID 和分组名（Public）插入 container 表

3）同步本地和系统资源

1. 每 5 秒进行一次镜像资源同步：

（1）本地镜像是否有被删除，若有，删除数据库对应元组

（2）本地是否新增镜像，若有，创建新元组插入数据库，分组设为 OutOfMyapp

（3）生成系统日志，用户名设为 unknown

2. 每 5 秒进行一次容器资源同步：

（4）本地容器是否有被删除，若有，删除数据库对应元组

（5）本地是否新增容器，若有，创建新元组插入数据库，分组设为 OutOfMyapp

（6）生成系统日志，用户名设为 unknown

4）用户查看系统资源

1. 管理员可查看并操作所有镜像、容器
2. 普通用户仅能查看并操作分组为自己的或是 public 的镜像、容器

5）用户生成新镜像、容器

1. 提交容器操作的新镜像

（1）获取该镜像 ID，分组为当前用户名

（2）生成元组插入 image 表

（3）生成日志插入 log 表

2. 创建容器产生的新容器

（1）获取该容器 ID，分组为当前用户名

（2）生成元组插入 container 表

（3）生成日志插入 log 表

3. 拉取镜像产生的镜像

该操作产生的镜像资源分组设为 public，为所有用户共享，因此不作处理。镜像的同步线程会检测到新镜像的产生，自动生成资源记录（分组会设为 Public）插入数据库

3.2.5 镜像、容器操作

1)镜像删除

1. 请求信息

（1）镜像 ID

（2）镜像操作类型

表 3.1 镜像基本操作

镜像操作类型	对应请求
删除	remove
强制删除	force
删除所有镜像层	removeall
强制删除所有镜像层	forceall

3. 系统处理

（1）获取容器镜像，判断操作类型

（2）调用 model 层对应函数进行处理

（3）生成操作日志，返回操作结果

2）镜像拉取

1. 请求信息

（1）镜像名及标签

2. 系统处理

（2）执行该操作

（3）生成操作日志，返回操作结果

3）镜像标签

1. 请求信息

（1）源镜像名

（2）目标镜像名

（3）事件类型（添加或删除）

2. 系统处理

（1）判断事件类型

（2）获取源镜像名和目标镜像名

（3）执行该操作,返回处理结果

4）容器基础操作

1. 请求信息

（1）容器 ID

（2）事件类型

表 3.2 容器基本操作

容器操作类型	对应请求
启动	start
停止	stop
杀死	kill
删除	remove

2. 系统处理

（1）判断事件类型

（2）获取容器 ID

（3）执行该操作

（4）生成操作日志，返回处理结果

5）提交容器（生成镜像）

1. 请求信息

（1）容器 ID

（2）目标镜像名

2. 系统处理

（1）获取请求信息

（2）处理该请求，获取生成的镜像 ID

（3）将该镜像 ID 插入数据库，分组设为当前用户名

（4）返回处理结果，生成处理日志

6）创建容器

1. 请求信息

（1）源镜像名

（2）创建后在容器中执行的命令

2. 系统处理

（1）获取请求信息

（2）命令格式转换

（3）根据这两项信息创建新容器，获取新容器 ID

（4）将该容器 ID 插入数据库，分组设为当前用户名

（5）返回处理结果，生成处理日志

3.2.6 容器日志

1. 请求信息

（1）容器 ID

（2）是否输出标准输出日志

（3）是否输出标准错误输出日志

（4）是否为日志添加时间戳

（5）是否输出日志详情

（6）输出的日志最末行数

2. 系统处理

（1）解析请求信息

（2）为请求后 5 项内容构造容器日志请求参数结构体对象

（3）将容器 ID 和请求参数结构体对象传入 model 层对应函数处理

（4）返回容器日志、处理结果

（5）生成处理日志

3.3 系统性能需求分析

1. 本系统后台多线程，定时更新数据库信息，保证资源及时更新
2. 本系统数据库关键操作进行事务处理，保证没有数据库错误
3. 本系统所有数据库操作都进行错误处理，保证每一种可能的错误都记录在日志中
4. 本系统所有页面加载时间由网络延迟决定，系统不予保证
5. 本系统的容器、镜像资源操作延迟由服务器运算速度和网络延迟决定，系统不予保证

3.4 系统业务流程分析

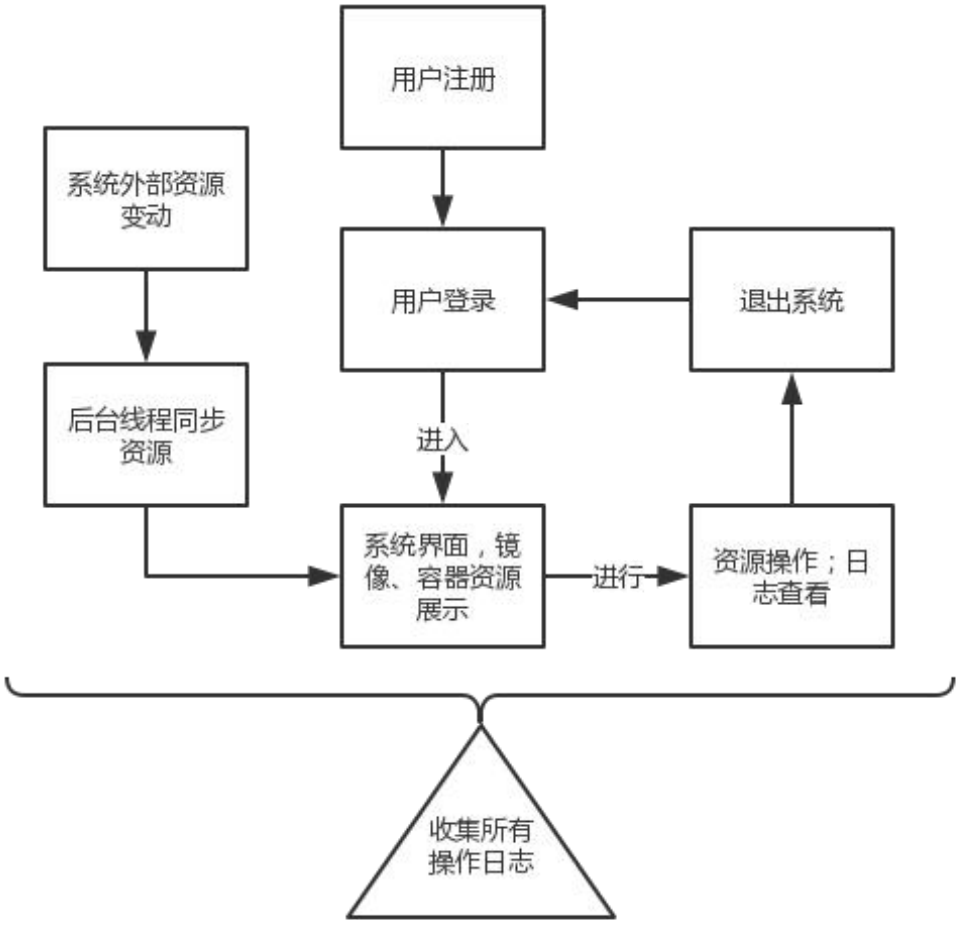

图 3.1 系统业务流程分析

3.5 本章小结

需求分析是我们整个软件工程中最关键的一个步骤，虽然本系统功能不难，但功能点比较多，逻辑处理也比较繁琐复杂，基本上每一个小功能点都涉及到前端与后端的交互，各个功能之间的联系也比较多，因此我必须做好详尽的需求分析。需求分析完成后，便可以针对其中的每一个功能点制定详尽的实现策略，各个功能之间的交互也了然于胸，指定详细的系统实现时也会有全局感，充分考虑到每一个功能之间的联系。这样，之后再一个功能一个功能写，边写边测，一切也就水到渠成。这一步若是没做好，开发过程中的需求变动很可能会影响到很多功能的变动，导致我们付出许多不必要的精力去修改。

4 系统设计

4.1 系统概要设计

表 4.1 系统概要设计

功能模块名	模块描述	实现方式
登录注册	控制用户的登录注册操作，用户分管理员与非管理员	数据库操作
访问和会话控制	控制用户的访问方式，登录后才能访问系统其它页面。控制用户平台会话，获取当前用户信息。	session＋ 路由过滤器
资源管理	平台资源（镜像、容器）分权限查看，平台内创建的资源为用户私有，平台外资源为用户公有，系统资源与本地保持同步	数据库操作 ＋ Goroutine 并发编程
资源操作	镜像的创建、删除、拉取; 容器的启动、停止、终止、删除、提交、创建	封装 docker API
平台日志	平台日志分权限查看，记录时间、用户名、操作记录	数据库操作
容器日志	容器日志查看	Docker API

4.1.1 登录注册模块

1）注册

1. 用户填入注册信息
2. 判断是否可以注册该用户，注册信息是否合法

2）登录

1. 用户填入登录信息
2. 判断是否登录成功，若是，将之存入 session，跳转至系统首页

4.1.2 访问和会话控制

1. 用户未登录时只能访问登录和注册页面
2. 用户未登录时请求所有其他页面都跳转至登录页面
3. 登录后系统能知晓当前用户展示其信息到所有页面
4. 退出平台时删除当前用户 session

4.1.3 系统资源管理

1. 管理员展示所有用户资源，普通用户展示 public 和自己产生资源
2. 系统建立时的已有资源全作为 public 资源
3. 用户生成的资源，分组名为当前用户名
4. 定时检测本地资源变动，更新系统资源
5. 系统外操作产生的新资源分组为“OutOfMyapp”

4.1.4 系统资源操作

1. 在镜像列表页面显示当前用户能看到的所有镜像，并提供相关操作

(1) 删除镜像，由于一个镜像的组成十分复杂，一个镜像是由许多镜像层组成，一个镜像 ID 也对应许多镜像标签，所以得提供多个操作：

1）删除：删除当前 ID 的镜像层

2）彻底删除：删除当前镜像 ID 下的所有镜像层

3）强制删除：当一个镜像 ID 有着多个标签时，这样会删除该层的所有镜像标签

4）强制彻底删除：删除所有标签及所有镜像层

（2）拉取镜像：用户输入镜像仓库名/镜像名:镜像标签（如：docker.io/mysql:latest）,点击确认进行镜像拉取

（3）创建容器：输入基础镜像名以及要在容器内执行的命令，来创建一个容器，该容器为当前用户私有

2. 在容器列表页面显示当前用户能看到的所有容器，并提供相关操作

（1）启动：启动已停止运行的容器

（2）停止：停止正在运行的容器

（3）强制终止：强制终止正在运行的容器

（4）删除：删除已停止运行的容器

（5）提交：提交一个容器，生成镜像，并有用户输入指定新的镜像名，该镜像为当前用户私有

4.1.5 系统日志

1. 在首页可通过点击某用于已专门查看该用户日志
2. 在系统日志页面可看到当前用户的所有操作日志（管理员看到所有用户操作日志）

4.1.6 容器日志

1. 在容器日志页面显示所有正在运行的容器列表
2. 通过点击某个容器以查看该容器的日志
3. 在该页面可设置 5 个参数来控制日志输出：

（1）是否显示标准输出

（2）是否显示标准错误输出

（3）是否添加时间戳

（4）是否显示日志详细内容

（5）设置显示日志最末多少行

4.2 系统详细设计

4.2.1 登录注册

控制用户的登录注册操作，用户分管理员与非管理员

表 4.2 登录注册

控制器	HTTP 方法	控制器方法	作用
LoginController	GET	Get()	渲染登录页面
	POST	Post()	处理登录请求
RegisterController	GET	Get()	渲染注册页面
	POST	Post()	处理注册请求

4.2.2 访问和会话控制

控制用户的访问方式，登录后才能访问系统其它页面。控制用户平台会话，获取当前用户信息。

表 4.2 访问和会话控制

核心函数	函数说明
FilterUser	1. 用户未登录时只能访问登录和注册页面 2. 用户未登录时请求所有其他页面都跳转至登录页面 3. 该函数作为参数传给 beego.InsertFilter 函数
*BaseController.Prepare()	1. 获取当前 session 2. 将 session 值赋给 User 对象，便于之后获取当前用户

4.2.3 系统资源管理

平台资源（镜像、容器）分权限查看，平台内创建的资源为用户私有，平台外资源为用户公有，系统资源与本地保持同步

表 4.3 系统资源管理

核心函数	函数作用时	函数功能
Init() 来自文件 image.go	初始化 myapp/models/db 包时	初始化数据库 image 表，所有本地 image 作为 public 资源插入
Init() 来自文件 container.go	初始化 myapp/models/db 包时	初始化数据库 container 表，所有本地 container 作为 public 资源插入
GetContainerIdsByUser(user types.User) ([]string, error)	用户请求容器列表时	由用户获取待展示的容器 ID
GetImageIdsByUser(user types.User) ([]string, error)	用户请求镜像列表时	由用户获取待展示的镜像 ID
SyncContainers()	系统建立后，每 5 秒执行一次	定时检测本地容器变动，同步数据容器资源与本地一致
SyncImages()	系统建立后，每 5 秒执行一次	定时检测本地镜像变动，同步数据镜像资源与本地一致

4.2.4 系统资源操作

镜像的创建、删除、拉取；容器的启动、停止、终止、删除、提交、创建。

表 4.4 系统资源操作

控制器	HTTP 请求	控制器方法	作用
AdminImageController	GET	Get()	渲染镜像列表页面
ImageController	GET	GetAllImages()	镜像列表 API，为前端提供镜像列表数据
	POST	OperateImage()	镜像操作 API，处理前端镜像基础操作请求：remove,removeall,force,forceall
ImagePullController	POST	PullImage()	镜像拉取 API，获取镜像名，处理镜像拉取请求
ImageTagController	POST	OperateTag()	镜像标签 API，获取源镜像名、目标镜像名、标签事件类型，处理奖项标签操作请求
AdminContainerController	GET	Get()	渲染容器列表页面
ContainerController	GET	GetAllContainers()	容器列表 API，为前端提供容器列表数据

表 4.4（续）系统资源操作

	POST	OperateContainer()	容器操作 API，处理前端容器基础操作请求：start,stop,kill,remove
ContainerCreateController	POST	CreateContainer()	容器创建 API，获取源镜像名和容器命令，创建新容器，
ContainerCommitController	POST	Commit()	容器提交 API，获取容器名和目标镜像名，基于该容器生成新镜像

4.2.5 系统日志

系统日志分权限查看，记录时间、用户名、操作记录。

表 4.5 系统日志

控制器	HTTP 请求	控制器方法	作用
UserLogController	GET	GetUserLog()	系统日志 API，从数据库中获取当前用户能看到的所有系统日志
	POST	SetUserSession()	首页用户想查看某个用户日志时，设置 user_name session，以便页面跳转时 GetUerLog()能正确加载

4.2.6 容器日志

查看正在运行的容器的日志

表 4.6 容器日志

控制器	HTTP 请求	控制器方法	作用
ContainerRunning Controller	GET	GetRunningContainers()	运行容器列表 API，加载正在运行的容器列表
	POST	GetContainerLog()	容器日志 API，获取容器 ID 及容器日志控制的 5 个参数:ShowStdout,ShowStderr,Timestamps, Details,Tail。收集容器日志输出流，存入字符串，传给前端

4.3 系统数据（库）设计

4.3.1 数据库 ER 图

本系统数据库设计很简单，都是围绕用户的一对多关系，下图为数据库设计的 ER 图：

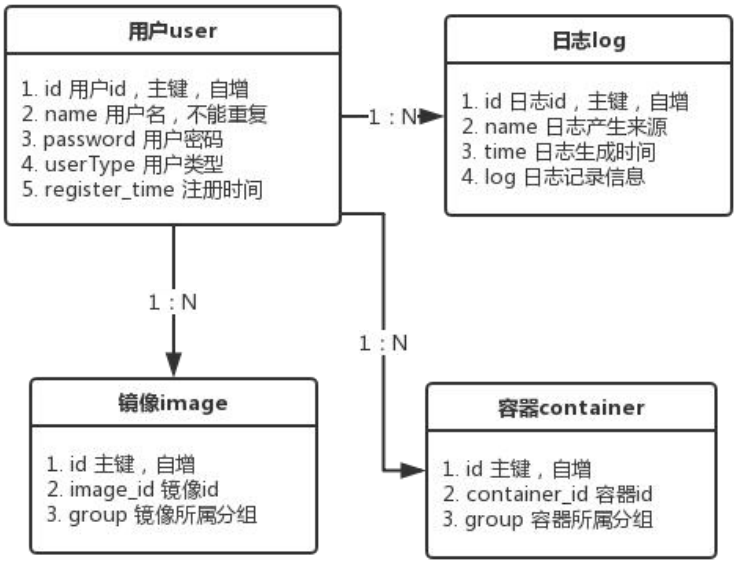

图 4.1 数据库 ER 图

4.3.2 反转数据库

下图是系统建立后通过 Mysql Workbench 对 myapp 数据库进行反转得到：

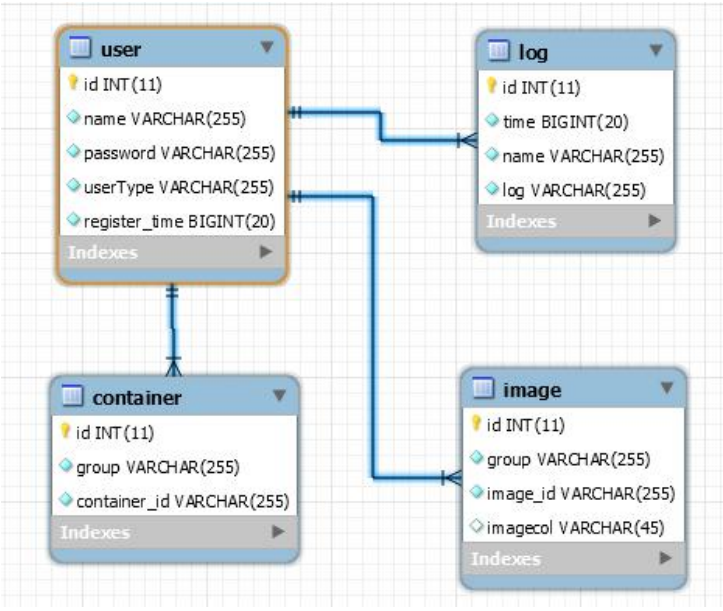

图 4.2 Mysql Workbench 反转数据库

4.4 系统用户界面设计

4.4.1 注册界面

1. 注册信息填选
   （1）用户名 输入框
   （2）密码 输入框
   （3）是否注册为管理员 勾选框
2. 操作按钮
   （1）登录 跳转至登录
   （2）注册 提交信息，尝试注册

4.4.2 登录界面

1. 登录信息填写
   （1）用户名
   （2）密码
2. 操作按钮
   （1）登录 提交信息，尝试登录
   （2）注册 跳转至注册

4.4.3 系统首页

1. 展示用户信息（管理岗展示所有用户，非管理员展示自己）
   （1）用户名
   （2）注册时间
   （3）用户类型
   （4）右上角展示当前用户名及用户类型
2. 操作按钮
   （1）首页 跳转至首页
   （1）镜像列表 跳转至镜像页面
   （2）容器列表 跳转至容器页面
   （3）容器日志 跳转至容器日志页面
   （4）系统日志 跳转至系统日志页面
   （5）退出平台 退出平台
   （6）查看该用户操作记录 查看选中用户的系统日志

4.4.4 镜像页面

1. 展示信息
   （1）镜像名

（2）镜像 ID

（3）镜像大小

（4）镜像创建时间

2. 操作按钮及输入框

（1）删除

（2）彻底删除

（3）强制删除

（4）强制彻底删除

（5）生成容器 弹出输入框要求输入执行的命令

（6）镜像名输入框及确认按钮 拉取镜像

4.4.5 容器页面

1. 展示信息

（1）容器 ID

（2）容器名

（3）源镜像

（4）执行的命令

（5）端口映射

（6）容器状态

2. 操作按钮

（1）启动

（2）停止

（3）杀死

（4）删除

（5）提交 弹出输入框以输入目标容器名

4.4.6 系统日志页面

展示信息：请求的日志列表

（1）日志生成时间

（2）日志生辰用户

（3）日志内容

4.4.7 容器日志页面

1. 展示信息：

（1）容器名

（2）容器 ID

（3）容器运行状态

（4）容器源镜像

2. 操作框及按钮

（1）是否输出标准输出日志 勾选框

（2）是否输出标准错误输出日志 勾选框

（3）是否为日志添加时间戳 勾选框

（4）是否输出日志详情 勾选框

（5）输出的日志最末行数 输入框

（6）查看日志按钮 提交上述所有信息，获取后端响应，弹出容器日志

4.5 本章小结

本章按照需求分析将系统的 6 个模块进行详细设计，确立了每一个模块应该实现的功能以及实现的方法，确立了后端要写的 API 接口。系统设计的目的是明确本系统的目标。因为在开发过程中，每一个功能的实现可能会涉及到好几个模块。比如实现容器的创建功能时，如果不明确整个系统的目标，也许只会考虑怎么实现该功能，之后又根据新的需求（日志、数据库等）来增删代码。而做好详细的系统设计后，实现该功能时，就会想到需要对该操作进行日志记录，还会发生数据库资源变动等等情况。这样，我们就会带着全局性的思维来设计、实现每个细节。

数据库的设计也很重要，它体现了系统的结构以及内容，做好详细的数据库设计，绘好 ER 图对一个系统至关重要。本系统的数据库较为简单，以 user 为核心，与 log，image 和 container 都是一对多关系。

本系统的界面为本章设计的 7 个模块，并不复杂，但功能点较多。做好界面设计后，前端的界面编写也轻松了许多。

5 系统实现

5.1 实现环境与工具

开发环境：Windows 7 + Cent OS 7

开发工具：

（1）Microsoft VS Code: Windows 下部分 Go 代码、所有 HTML 和 Javascript 代码编辑工具

（2）Vim：Cent OS 下的 Go 代码编辑工具

（3）Git：版本控制系统

（4）Mysql：数据库

（5）Chrome：界面和前端代码调试的浏览器

（6）Xshell：虚拟机 ssh 连接工具

5.2 系统框架及描述

5.2.1 登录注册

1）相关 API：

（1）GET /login 获取登录界面

（2）POST /login 登录信息处理：验证是否有该用户、加密后密码是否正确、登录成功后将该用户存入 session、生成日志、返回处理结果

（3）GET /register 获取注册界面

（4）POST /register 注册信息处理：判断是否能注册为管理员、用户名是否重复、创建用户、生成日志、返回处理结果

2）处理流程：

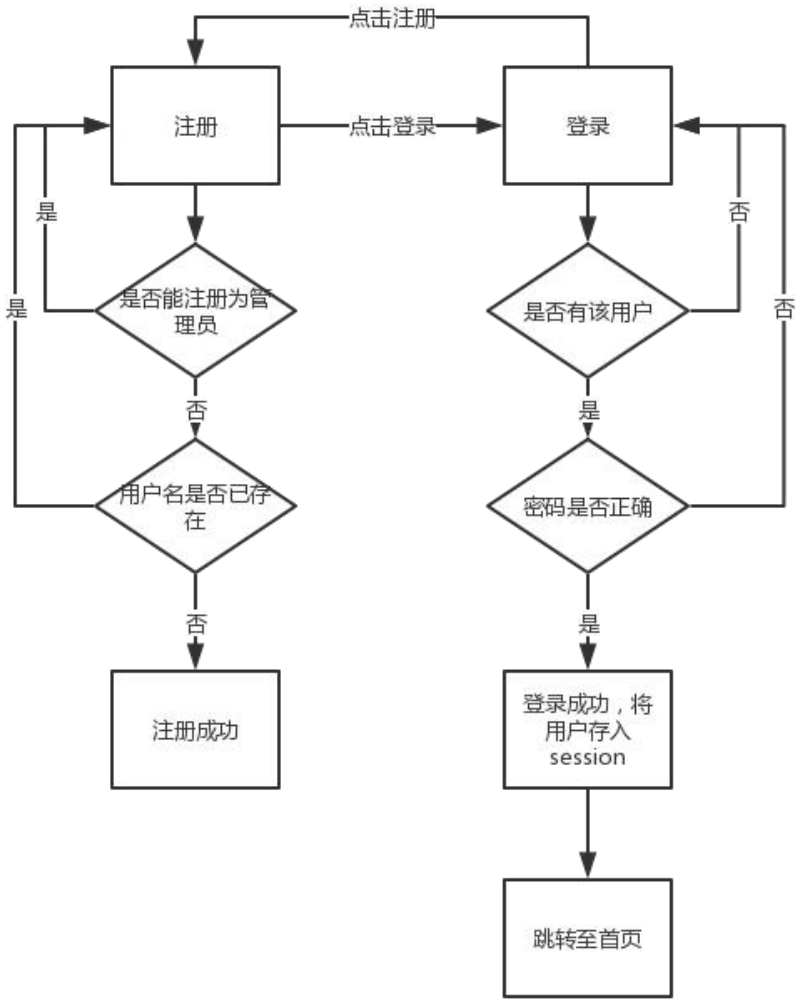

图 5.1 登录注册处理流程图

5.2.2 访问与会话控制

1）相关 API:

（1）POST /api/user 删除当前用户 session

（2）GET /api/user/name 返回展示在界面上的当前用户名及类型字符串给前端

2）过滤器和控制器函数

Router 中添加请求过滤器“FilterUser”，处理所有 HTTP 请求,若没有登录成功时设置的 session，所有除“/login”和“/register”之外请求都跳转至“/login”界面，做到了访问控制。

Controller 中重写 Prepare 函数，将 session 存入 User 结构体。由于每一次 HTTP 请求都会交由 Controller 处理，做到了会话控制。

3）每一次 HTTP 请求的处理流程：

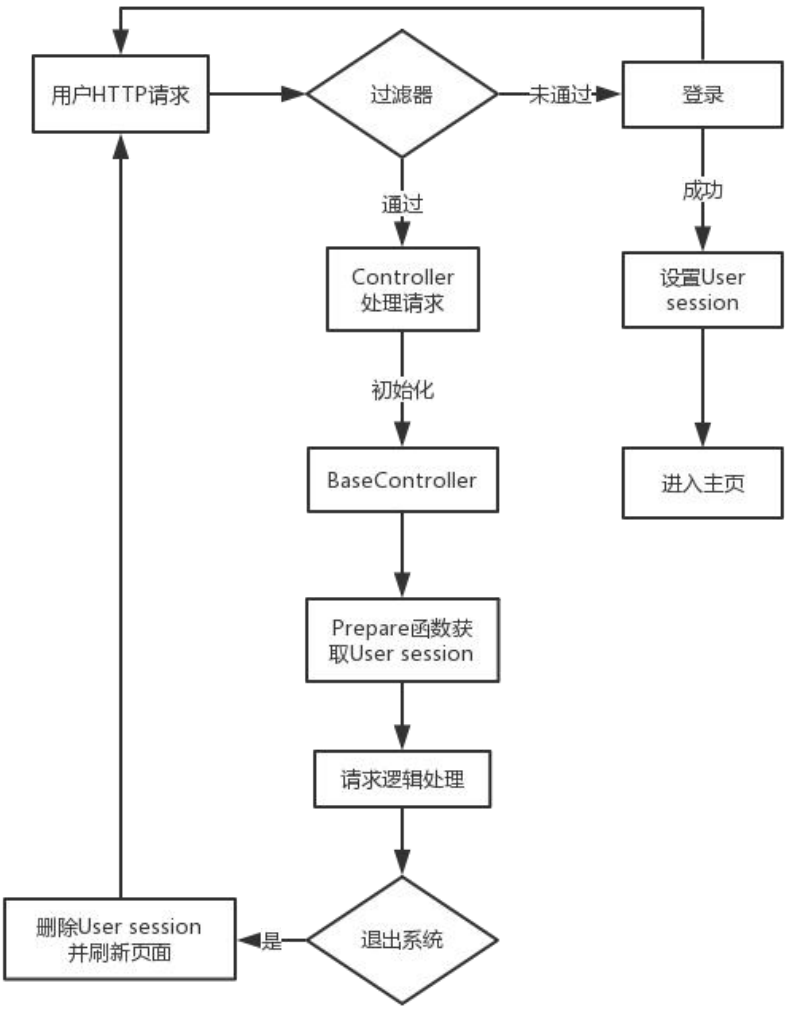

图 5.2 访问控制和会话控制处理流程图

5.2.3 平台日志系统

1）相关 API：

（1）GET /log　平台日志界面

（2）GET /api/log　由 user_name session 和当前用户名及用户类型共同判断加载哪些用户的日志

（3）POST /api/log 设置由/index 页面添加的 user_name session

2）处理流程：

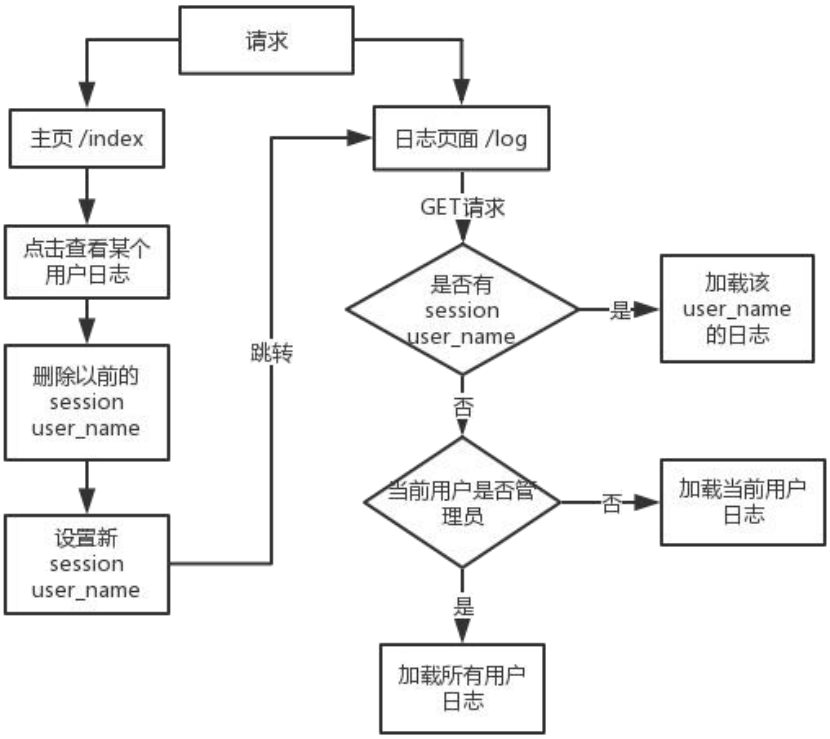

图 5.3 平台日志系统流程图

5.2.4 镜像、容器资源管理

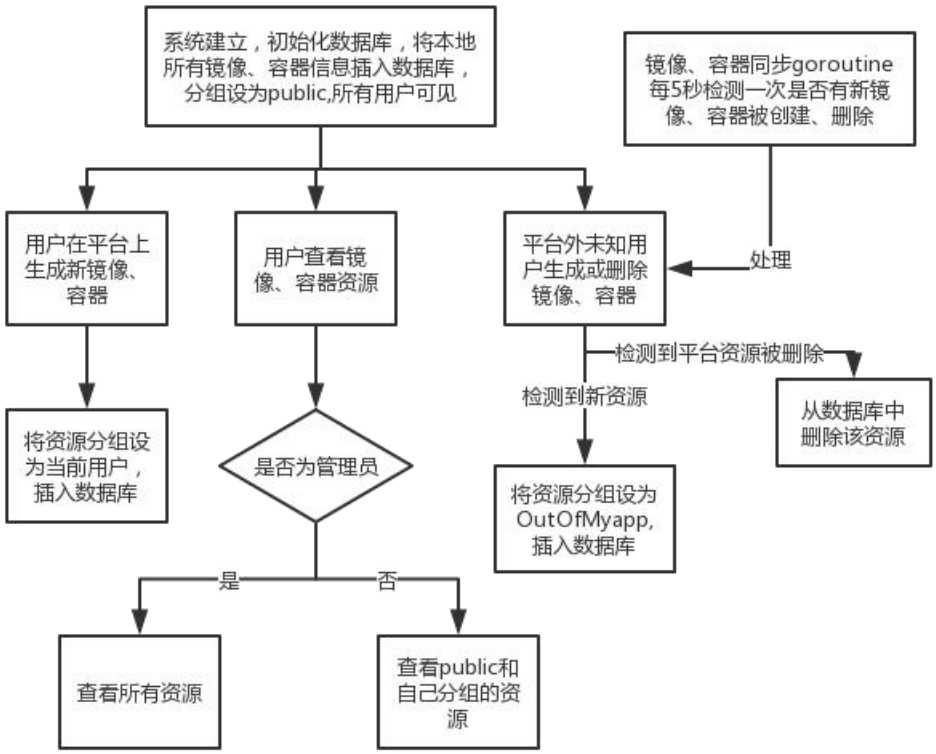

图 5.4 镜像、容器资源管理流程图

5.2.5 镜像、容器操作

1）相关 API：

（1）GET /container　容器管理界面

（2）GET /image　镜像管理界面

（3）GET /api/container　加载容器列表

（4）GET /api/container/running　加载正在运行的容器列表

（5）GET /api/image　加载镜像列表

（6）POST /api/container/operate　处理容器各种基础操作请求：停止、启动、强制终止、删除、运行一个新的容器

（7）POST /api/image/operate　处理镜像各种基础操作请求：删除、强制删除、彻底删除、强制彻底删除（删除所有 tag、所有镜像层）

（8）POST /api/container/commit　处理容器提交（生成镜像请求），生成后获取当前用户名，生成日志，将日志及新的镜像信息插入数据库

（9）POST /api/container/create　处理容器创建请求，成功后获取当前用户名生成日志，将日志及新的容器信息插入数据库

（10）POST /api/image/pull　处理镜像拉取请求，拉取用户在界面上传入的镜像名的镜像，生成日志和新的镜像资源信息插入数据库

（11）POST /api/image/tag　处理镜像标签操作请求，为镜像添加便签。

2）镜像流程图

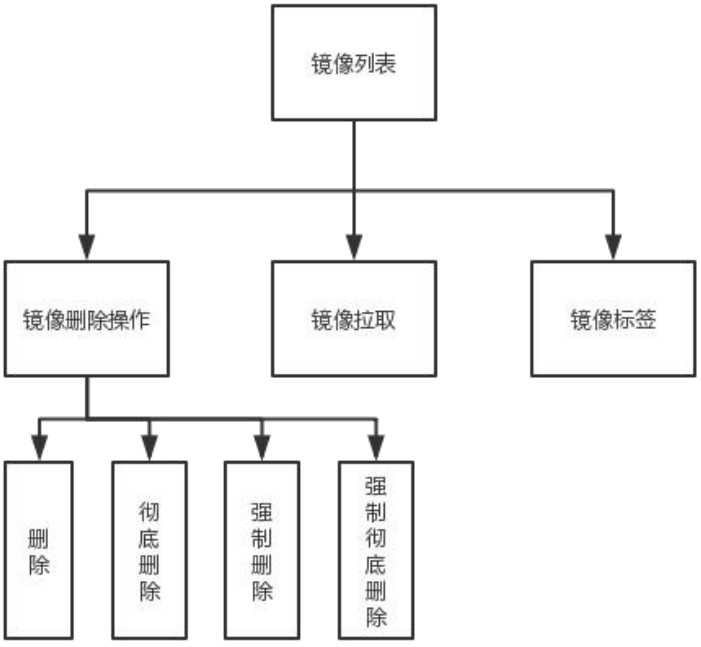

图 5.5 镜像操作流程图

3）容器流程图

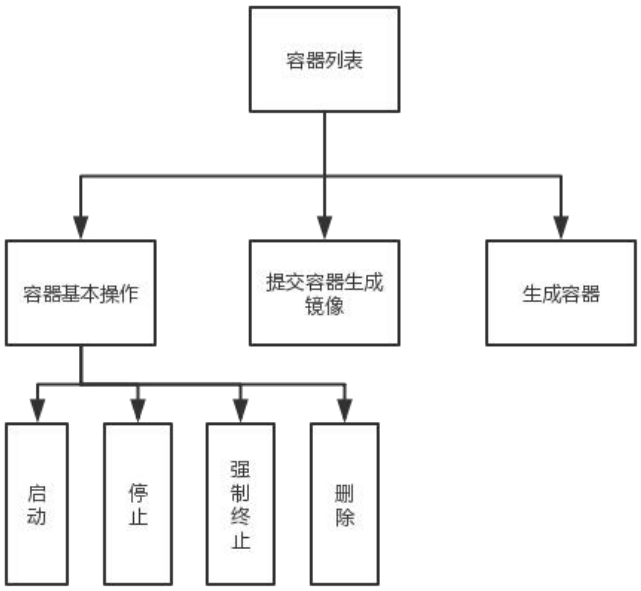

图 5.6 容器操作流程图

5.2.6 容器日志

1）相关 API：

POST /api/container/log 处理前端传入的 6 个参数：

表 5.1 容器日志请求项

请求参数	参数意义
Container_ID	请求的容器 ID
ShowStdout	是否返回标准输出日志
ShoeStderr	是否返回标准错误输出日志
Timestamps	是否返回时间戳
Details	是否返回日志详情
Tail	需返回的日志最末行数

2）流程图

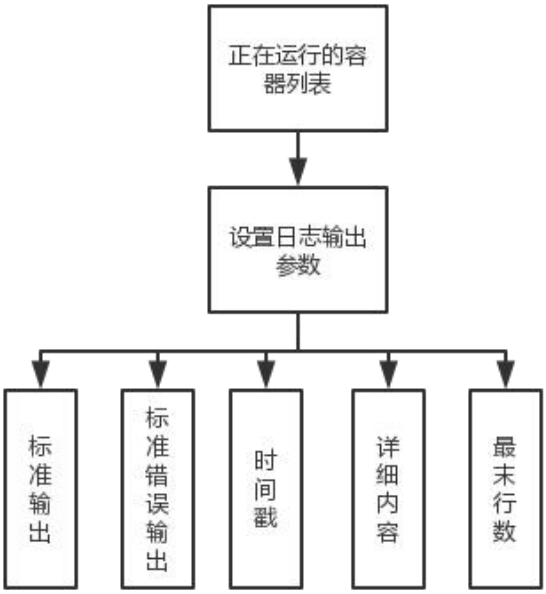

图 5.7 容器日志流程图

5.3 系统运行界面

5.3.1 注册界面

图 5.8 注册界面

5.3.2 登录界面

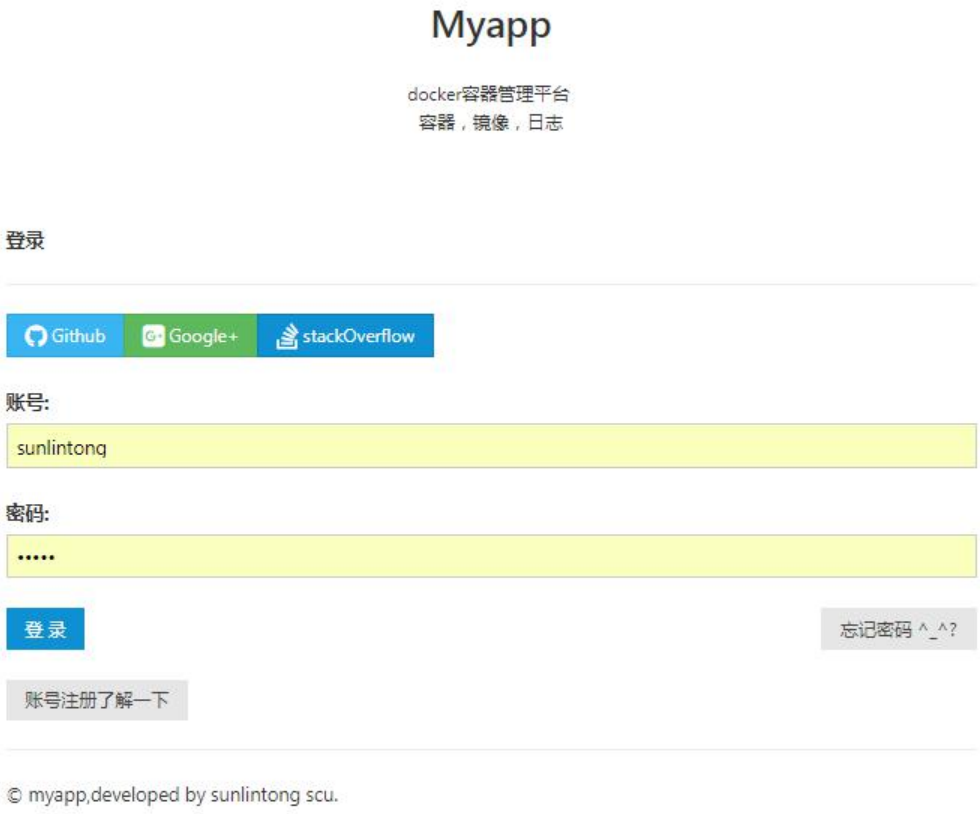

图 5.9 登录界面

5.3.3 首页

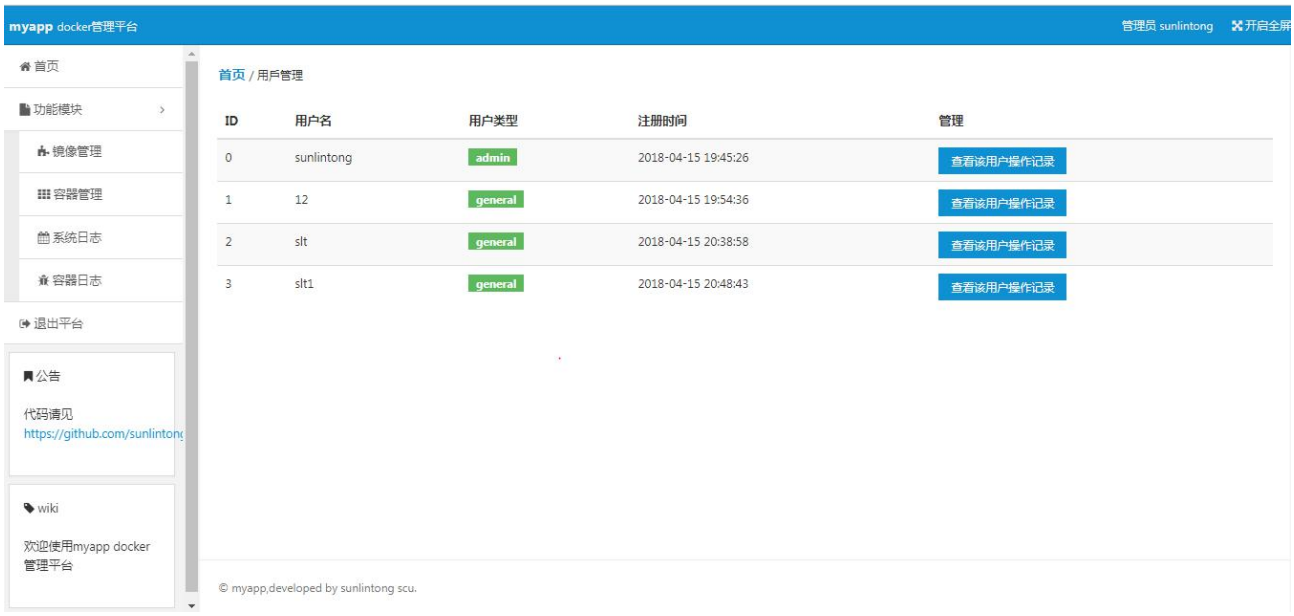

图 5.10 系统首页（管理员界面）

5.3.4 镜像界面

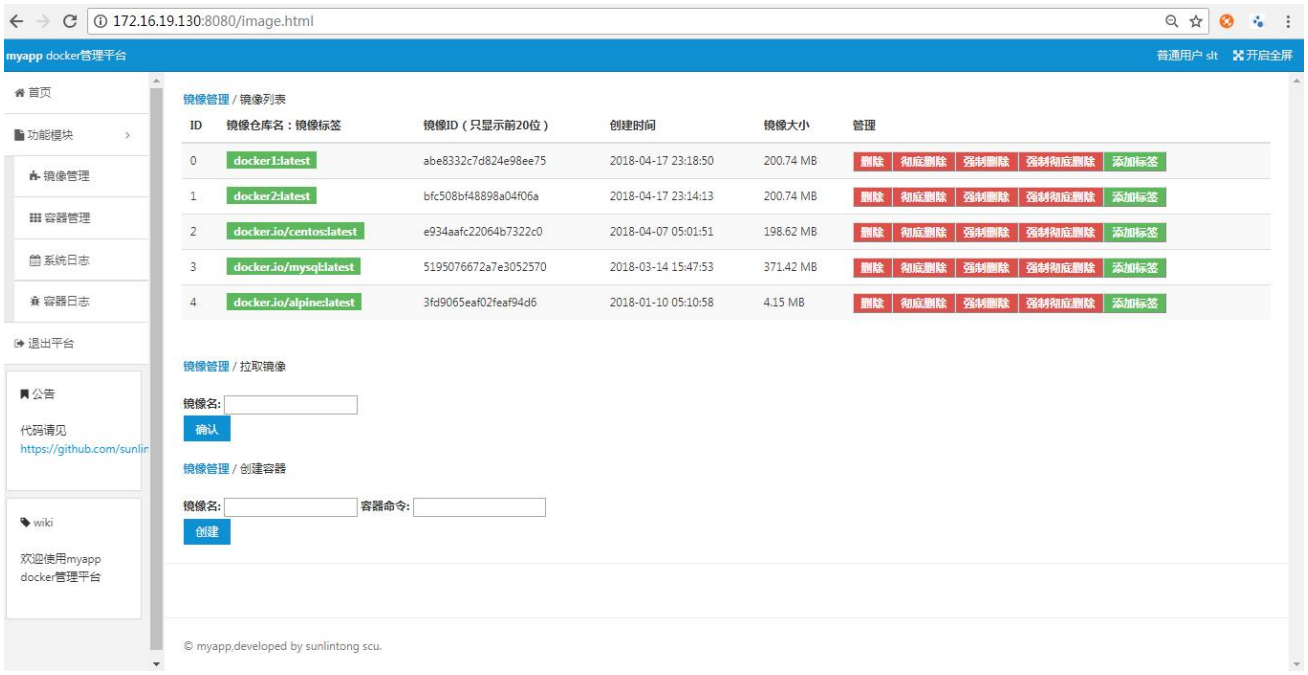

图 5.11 镜像界面

5.3.5 容器界面

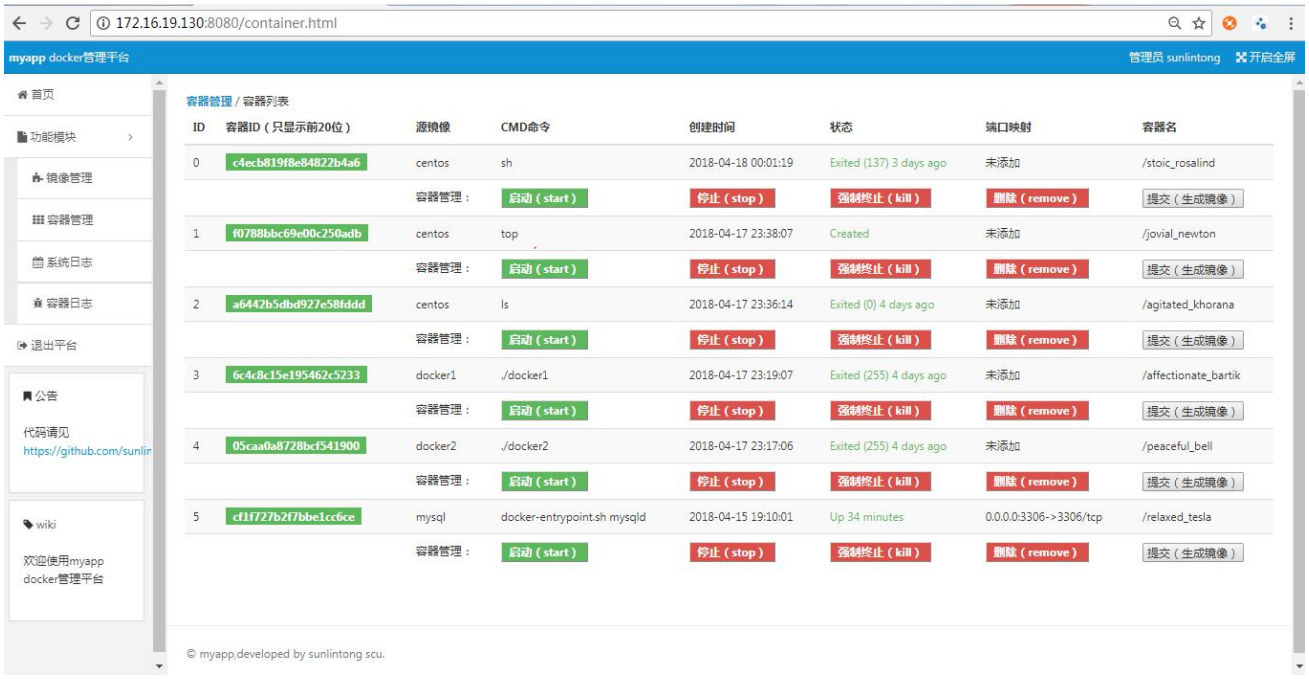

图 5.12 容器界面

5.3.6 系统日志界面

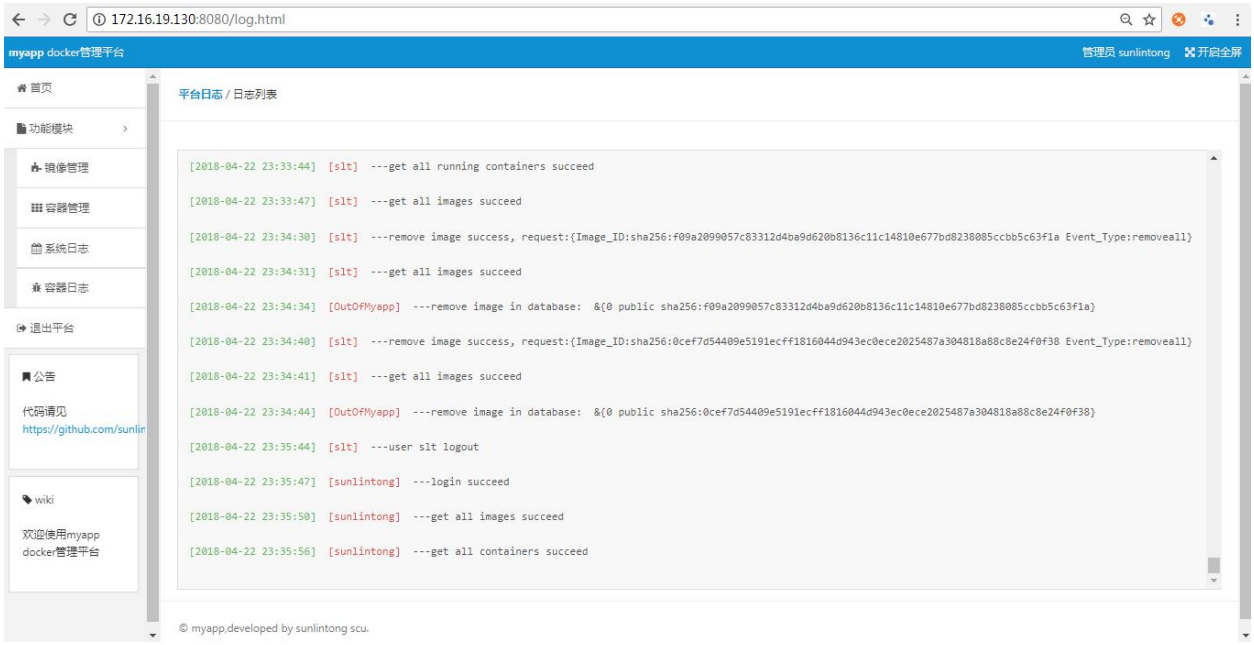

图 5.13 系统日志界面

5.3.7 容器日志界面

1）显示正在运行的容器列表

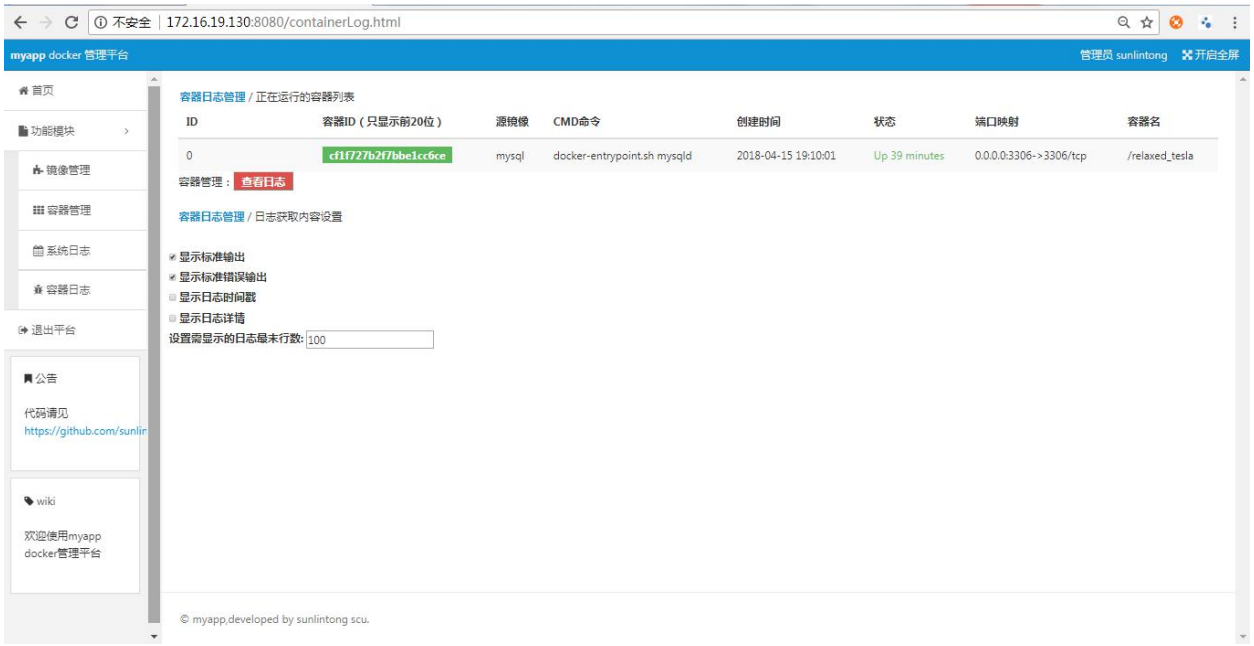

图 5.14 容器日志操作界面

1） 显示容器日志

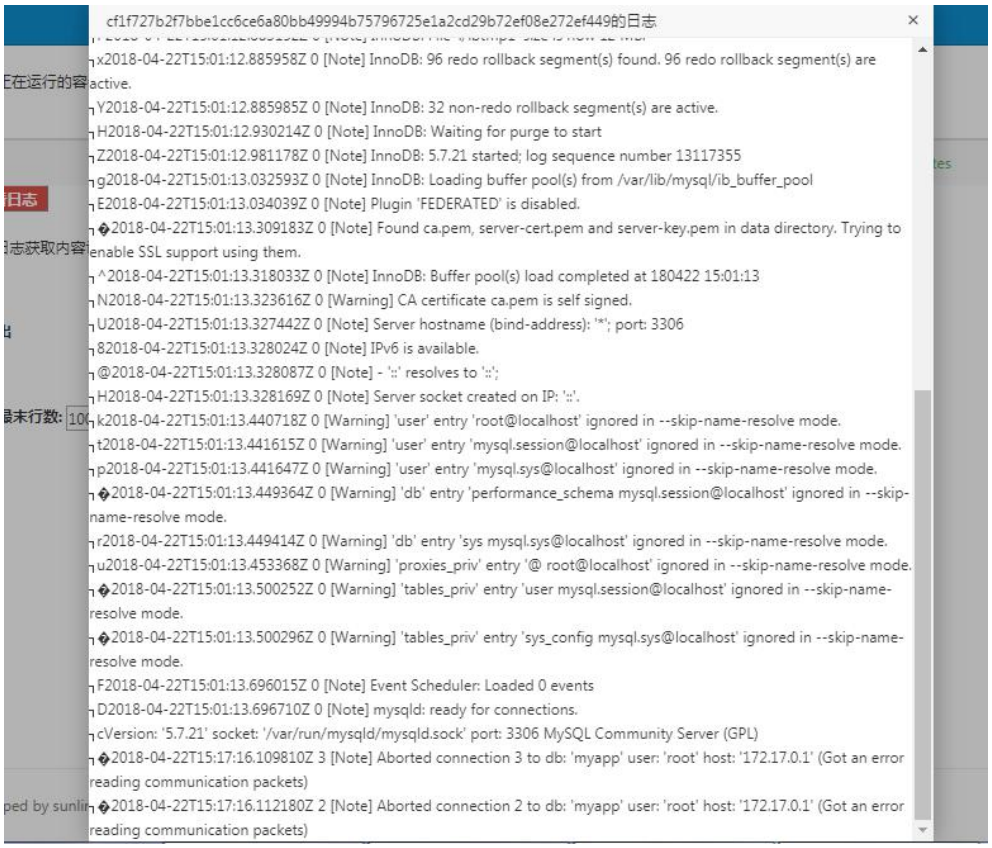

图 5.15 容器日志显示

5.4 本章小结

本章详细介绍了本系统 6 个模块中，每一个模块的业务处理流程。用流程图的方式进行展示，简明扼要、清晰明了地展示了每个模块所需要考虑的所有情况，理清了所有的业务逻辑。

最后展示了平台的所有主界面，但只展示运行界面并没有充分体现出本系统的所有功能。本系统在许多方面考虑得比较全面，比如用户隔离、权限管理、访问和会话控制，还有后台在运行的两个数据库同步线程，这些功能在详细的视频演示中可以充分展示出来。

6 系统测试

6.1 测试环境

1）主机硬件

1. CPU: 2.5 GHz
2. RAM: 8 GB
3. 硬盘：1000GB

2）虚拟机硬件

1. 处理器：1 个
2. 内存：1GB
3. 硬盘（SCSI）：30GB

3）主机软件

1. Windows7 64 位操作系统
2. Chrome 浏览器
3. Windows7 64 位操作系统
4. VMware 虚拟机平台
5. Xshell ssh 连接工具

4）虚拟机软件

1. Cent OS7 64 位操作系统

6.2 测试程序和测试数据设计

1）数据库测试

1. 在虚拟机上运行程序，查看数据库内容是否初始化成功
2. 在本地直接用 docker 命令添加或删除镜像容器，查看是否输出系统更新数据库日志，查看数据库中是否已更新相关信息
3. 在系统中进行资源操作，查看数据库是否进行相应更改

2）注册测试

1. 注册为管理员，成功后再注册一个新管理员，查看是否成功，查看前端响应
2. 注册时用户名选择已注册的用户名，查看前端响应

3）登录测试

1. 输入系统中没有的用户名进行登录，查看前端响应
2. 输入错误的密码，查看前端响应
3. 登录成功是否跳转至主页

4）访问控制测试

1. 不登录系统，在浏览器中输入登录注册以外的URL，查看页面响应、后端日志
2. 登录系统后，在浏览器中输入登录注册以外的URL，查看页面响应、后端日志
3. 点击页面的退出平台，查看页面响应，再在浏览器中输入登录注册以外的URL

5）会话控制测试

1. 查看登陆后所有页面右上角是否正确显示当前用户类型及用户名
2. 进行一些操作，查看系统日志是否正确记录操作用户名

6）资源管理测试

1. 登录普通用户，在镜像列表页面创建一个容器
2. 在容器列表页面查看是否显示该容器
3. 退出当前用户，登录另一个普通用户
4. 在容器列表页面查看是否显示该容器
5. 退出当前用户，登录管理员账户
6. 在容器列表页面查看是否显示该容器

7）镜像删除测试

1. 进入镜像列表页面，点击“删除”、“彻底删除”、“强制删除”、“强制彻底删除”
2. 查看前端响应及页面信息变化

8）镜像拉取测试

1. 在镜像管理/拉取镜像模块输入 镜像名，点击确认
2. 查看拉取中页面响应
3. 查看拉取完成后前端响应及浏览器Console 输出

9）容器创建测试

1. 在镜像管理/创建容器模块中输入镜像名及容器命令，点击创建
2. 查看前端响应、Console 输出
3. 在容器列表页面查看是否新增该容器及其信息

10）容器基本操作测试

1. 在容器管理/容器列表页面点击启动、停止、强制终止、删除
2. 查看前端响应及控制台输出

11）容器提交测试

1. 在容器管理/容器列表界面点击提交，并根据提示进行输入，点击确认
2. 查看前端响应及 Console 输出
3. 进入镜像列表页面查看是否新增该容器及其信息

12）系统日志测试

1. 登录管理员账户，进入系统首页，点击某用户“查看该用户操作记录”

2. 查看页面跳转后日志显示内容是否只有该用户
3. 刷新当前页面，查看是否显示所有用户日志
4. 退出管理员账户，登录普通用户账户
5. 在首页点击用户（只有自己）的“查看该用户操作记录”
6. 查看页面跳转后日志显示内容是否只有该用户
7. 刷新当前页面，查看是否只显示当前用户日志
8. 在系统内进行一些操作，在查看系统日志是否更新

13）容器日志测试

1. 在容器日志界面根据默认日志输出设置点击某容器的“查看日志”
2. 都是查看弹出窗是否有显示容器日志
3. 勾选“显示日志时间戳”，查看每行日志前是否新增时间
4. 设置需显示的最末行数，查看输出是否正确

6.3 测试运行和测试记录

1）数据库测试

id	group	image_id
1	public	sha256:84865f45bcbf2076a7dde6b7b841a8ec...
2	public	sha256:b1baf466a95af2f4261228408e8037b4...
3	public	sha256:254c55f4f1e8d10fd76cffa91d1fb962c3...
4	public	sha256:abe8332c7d824e98ee75b427e1cdd354...
5	public	sha256:bfc508bf48898a04f06a614bfb7d54132...
6	public	sha256:e934aafc22064b7322c0250f1e32e5ce9...
7	public	sha256:5195076672a7e30525705a18f7d352c9...
8	public	sha256:3fd9065eaf02feaf94d68376da5254192...
9	public	sha256:99dd8ed83009447203a3d0a12abfa442...
NULL	NULL	NULL

图 6.1 初始 image 表

id	group	container_id
1	public	c4ecb819f8e84822b4a6f59667c4e6352441613...
2	public	f0788bbc69e00c250adb5943c633b328c5369db...
3	public	a6442b5dbd927e58fddd6e5d603ac945451ec5d...
4	public	6c4c8c15e195462c52331b21b234606a543df31...
5	public	05caa0a8728bcf5419001 4833920be52f2c922c...
6	public	cf1f727b2f7bbe1cc6ce6a80bb49994b75796725...
NULL	NULL	NULL

图 6.2 初始 container 表

删除一个镜像：docker rmi alpine,日志响应：

```
2018/04/26 18:47:22 remove image in database:  &{0 public sha256:3fd9065eaf02feaf
94d68376da52541925650b81698c53c6824d92ff63f98353}
```

图 6.3 删除镜像日志响应

2）注册测试

1. 系统已有管理员 sunlintong，尝试注册为管理员

图 6.4 不能注册为管理员

2. 注册系统已有用户

图 6.5 不能注册已有用户

3）登录测试

此功能不再截图，和注册功能一样都是 alert()提示

4）访问控制测试

1. 未登录请求/index 页面，跳转过程无法截图，日志截图：

```
2018/04/26 19:04:37 url: /index, session: {Name: IsAdmin:false}, pass?: false
2018/04/26 19:04:37.979 [D] [server.go:2694] 172.16.19.1 - - [26/Apr/2018 07:0
ws NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/66.0.3359.117

2018/04/26 19:04:37 url: /login, session: {Name: IsAdmin:false}, pass?: false
2018/04/26 19:04:37 controller prepare get session failed!
```

图 6.6 请求跳转至/login

2. 登录成功跳转跳转：

```
2018/04/26 19:07:39 url: /index, session: {Name:sunlintong IsAdmin:true}, pass?: true
2018/04/26 19:07:39 controller prepare: {Name:sunlintong IsAdmin:true}
2018/04/26 19:07:39.250 [D] [server.go:2694] 172.16.19.1 - - [26/Apr/2018 07:07:39] "GET
0:8080/login Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) C

2018/04/26 19:07:39 url: /api/user, session: {Name:sunlintong IsAdmin:true}, pass?: true
2018/04/26 19:07:39 controller prepare: {Name:sunlintong IsAdmin:true}
```

图 6.7 登录成功跳转

5）会话控制测试

1. 页面显示当前用户

图 6.8 管理员

图 6.9 普通用户

6）资源管理测试

1. 用户 sunlintong 创建容器

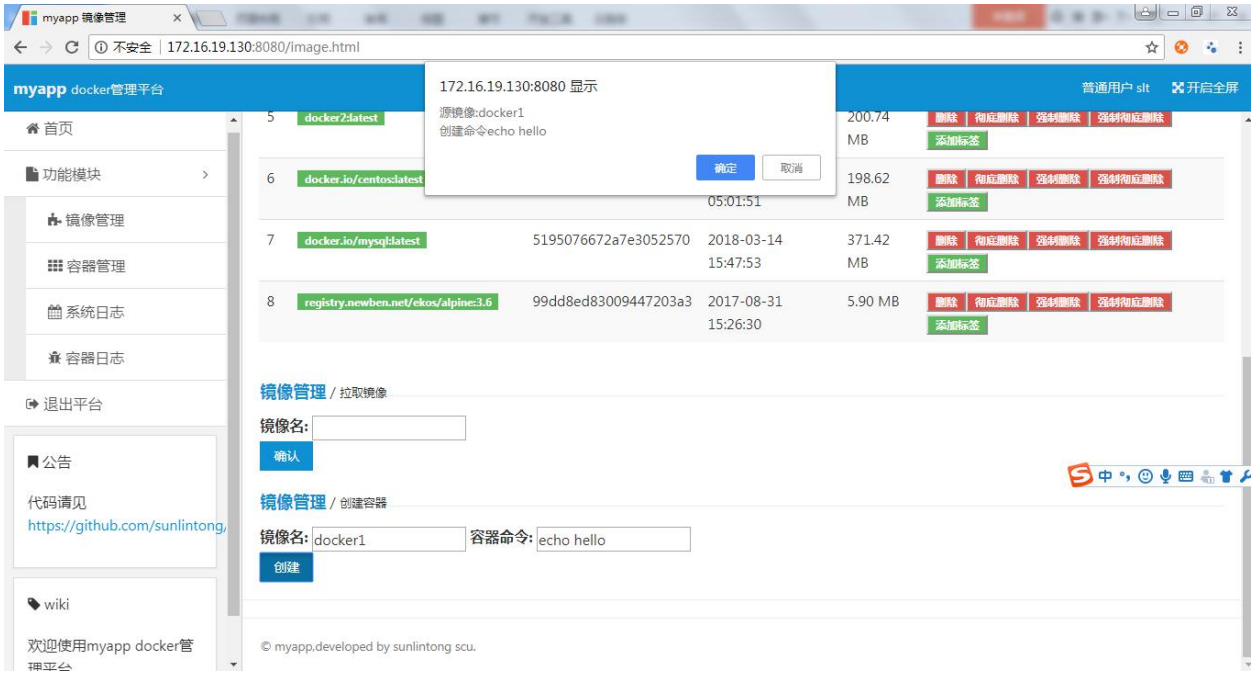

图 6.9 用户 sunlintong 创建容器

2. 用户 sunlintong 新增容器

图 6.10 用户 sunlintong 新增容器

7）镜像删除测试

下图是强彻底删除 myapp:1.0 的截图

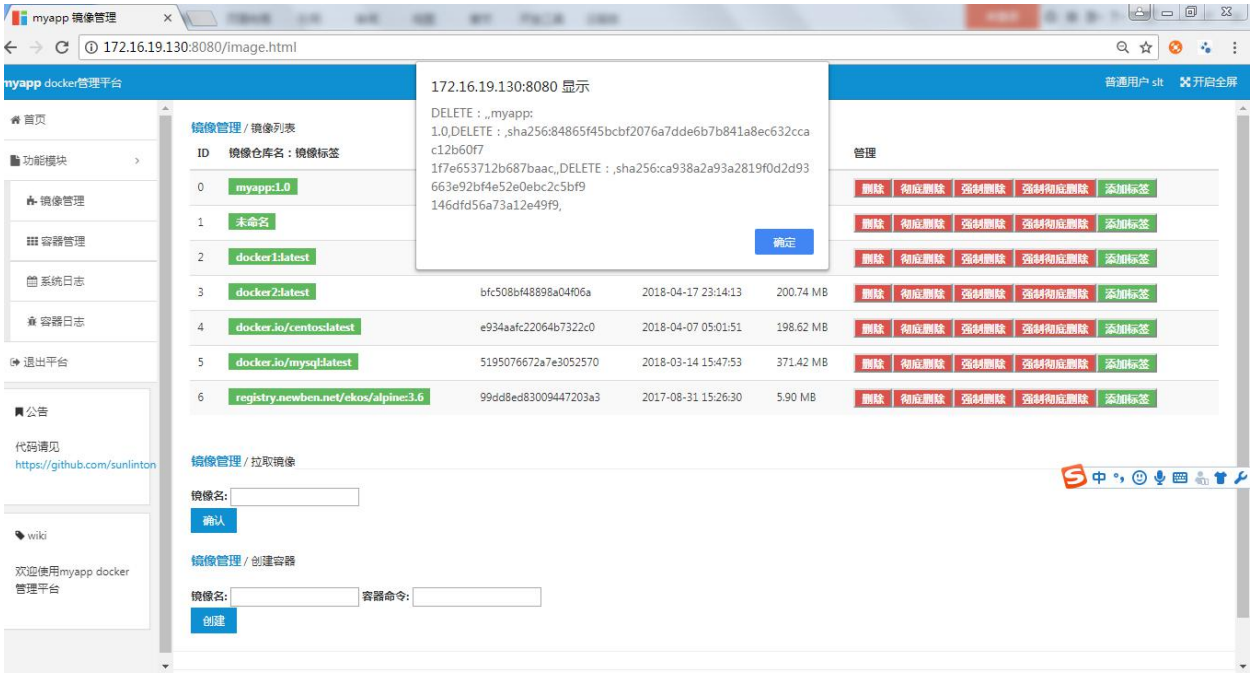

图 6.11 删除镜像

8）镜像拉取测试

F12 打开浏览器 Console，在拉取镜像模块输入 docker.io/alpine:latest，点击确认，观察页面翻迎，查看 Console 输出；

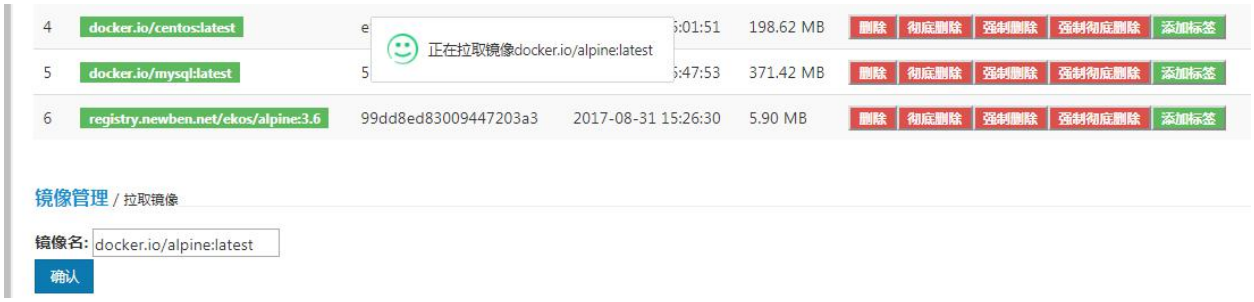

图 6.12 正在拉取镜像

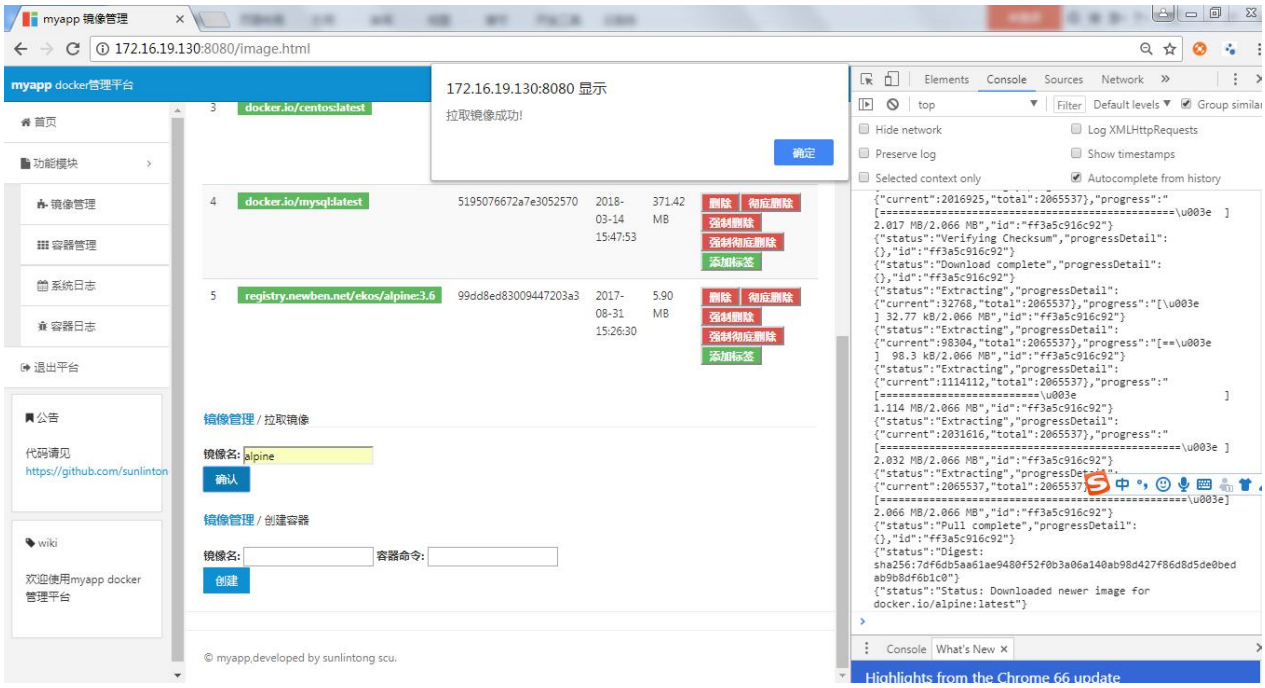

图 6.13 拉取镜像成功

9）容器创建测试

此测试用例在资源管理测试中已做

10）容器基本操作测试

1. 启动已停止的容器 docker2

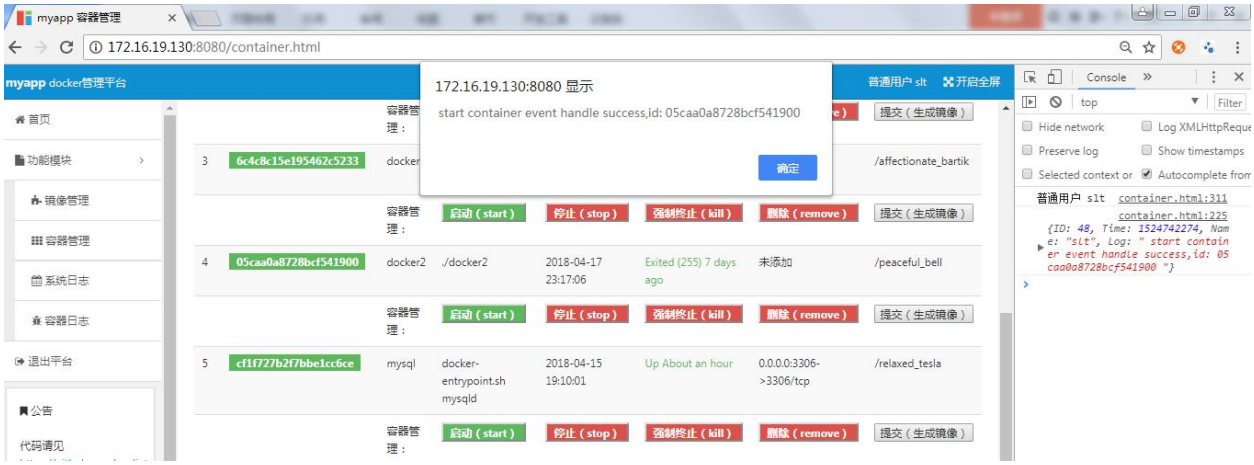

图 6.14 启动容器

2. 其他三个基本操作运行结果类似，此处不再截图

11）容器提交测试

1. 提交容器 docker1

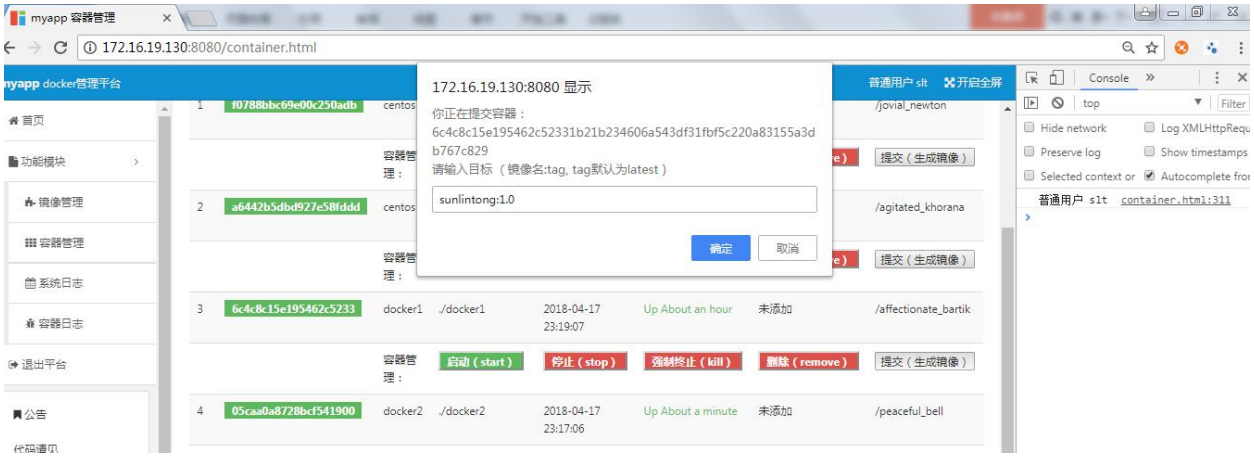

图 6.15 提交 docker1

图 6.16 提交成功

12）系统日志测试

管理员查看日志：

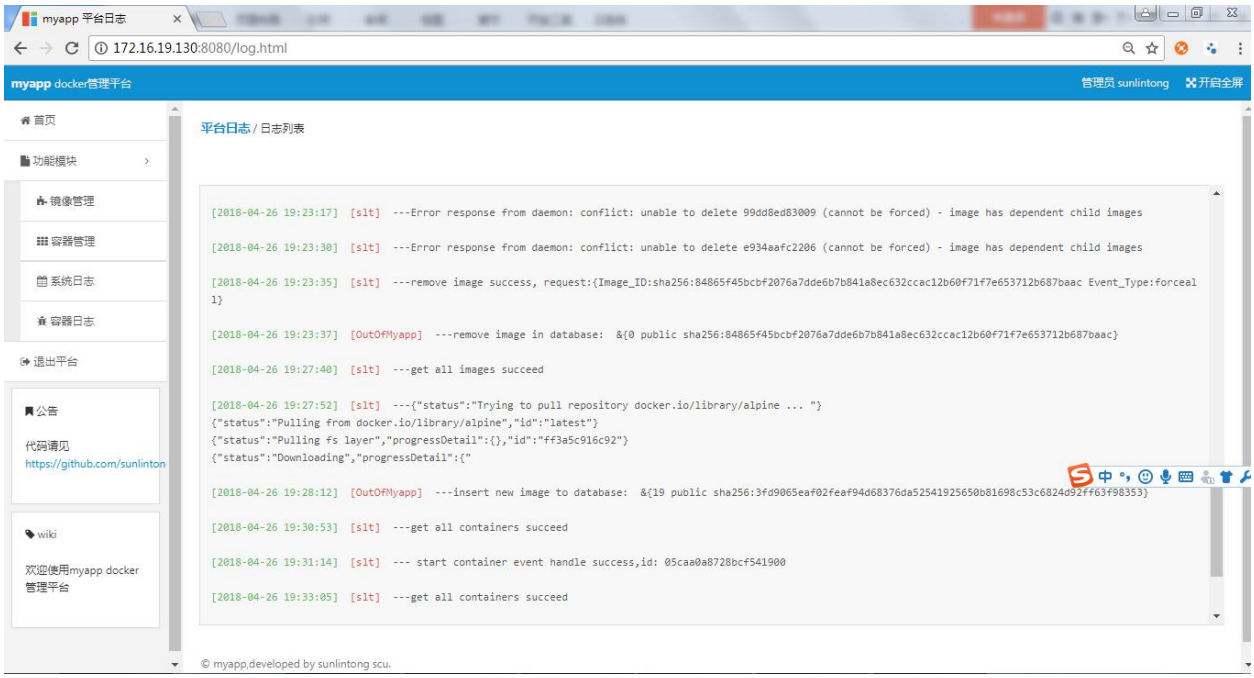

图 6.17 管理员查看日志

13）容器日志测试

1. 查看容器 docker2 日志

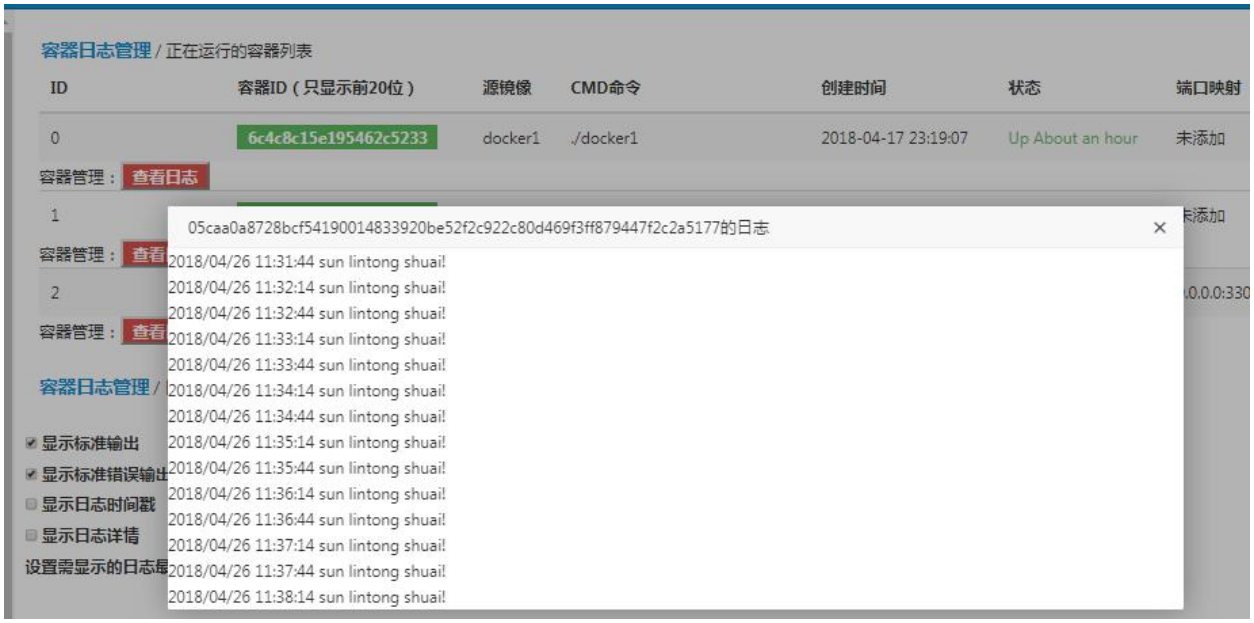

图 6.18 docker2 日志

2. 设置显示最末 5 行：

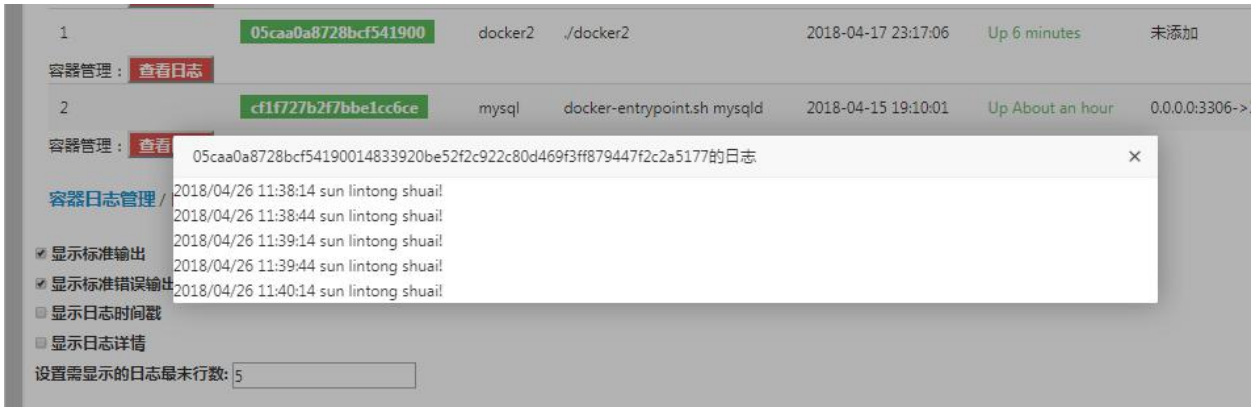

图 6.19 设置显示最末 5 行

6.4 测试结果分析

1）结果分析

本测试工作对系统的所有功能及需要达到的性能进行全面的测试，根据测试结果，系统完全到达了设计要求，而且在用户体验上做得比较好，体现在以下三点上：

1. 每个操作都有前端响应，优化增加了用户操作体验

2. 每种设计中考虑到的用户不当操作都进行了处理，并在前端进行响应提示，如登录注册的操作错误、镜像拉取时镜像名错误、创建容器时镜像名输入错误等等。

3. 所有操作错误也在控制台及前端界面进行了反馈，同时记录在了系统日志，能较好地引导用户进行正确地操作

2）待改进的点

通过经过全面的测试后，我从一个客户而不是开发者的角度完整地体验了本系统，发现了一些待改进的点：

1. 登录界面。登录成功时提示的 alert 框每次都要客户手动去点“确认”，会造成体验上的不适。改为短时间、能自动消失的提示框并同时进行页面跳转更好一点。

2. 系统日志界面。系统的运行时间增加会产生大量日志，在日志界面增加时间选择控件，以按时间筛选显示的日志[12]。

3. 提示框。本系统的前端提示框基本都是用的最简单的 javascript alert()方法，不太美观，与系统界面也不协调。这方面可自己写一个提示框函数来优化。

4. 镜像容器操作界面按钮不美观，界面设计有待改进

6.5 本章小结

本章的系统测试至关重要，开发人员往往在开发完系统所有功能之后不愿意做一次系统上的测试，或者测试工作没有做好，导致客户在产品使用过程中体验不好。

客户的许多操作行为是在开发人员预料之外的，我们在做测试工作前，应当梳理一遍整个系统的功能点，然后站在客户的角度，分析他们在每个功能上可能做出的操作，然后对这些操进行相关测试，便能不断完善我们的系统。

7 结束语

7.1 工作总结

1）进度安排

毕业设计工作从 2017 年 11 月 10 日至今持续了接近半年,刚开始时整个过程的进度安排如下：

1. 2017-11-10 至 2017-11-20：资料收集，项目思路规划。
2. 2017-11-20 至 2017-11-30：开始尝试代码编写，简单的 web 页面搭建
3. 2017-12-1 至 2017-12-31：阅读 docker 官方给出的 API 文档，自己做些封装
4. 2018-1-1 至 2018-2-10：用封装后的 API 写出系统基本后端功能
5. 2018-2-10 至 2018-2-28：多用户实现
6. 2018-3-1 至 2018-3-10：前端实现，界面优化
7. 2018-3-11 至 2018-3-31：视察、测试系统，修复其中的不足与 bug
8. 2018-4-1 至答辩：文档编写、整理

2）出现问题

实际的项目实施过程，基本和按照进度安排一直，但其中第 6、7 步出了一些问题：

1. 本以为前端写起来会很快，但我对前端知识一窍不通，因此花了很多时间学习前端

2. 测试时，由于设计时是多用户系统，但测试时发现没有对用户进行隔离，权限管理页眉做好。这样的多用户系统是没有意义的，因此我又加了新的用户隔离的需求，这个工作做起来不提容器，花了较长时间，导致后面的文档编写时间被压缩了很多。

3）成果

这次的毕业设计工作，成果可谓颇多。开发出了这么一个 docker 管理系统的确是一个较大的成果，但对我自己来说，更为重要的是在开发的过程中，我学到了许多技术，学会了自己去发掘新知识、学习新技术。从前端到后端的开发过程中，让我体会到了全栈工程的魅力，坚定了在软件工程道路上不断学习的决心。

4）下一步工作

1. 实现 docker 远程管理，让本系统不止能管理本地的 docker 资源，还能访问远程的 docker 守护进程，进行向本地一样的操作

2. 封装更多的 docker API，实现更多更为复杂的操作

3. 在系统界面上直接进入容器，以便在浏览器中操作容器内部

4. 系统日志按时间过滤

7.2 心得体会

毕业设计期间，收获颇多，不仅是技术上的知识，还有思想上的转变、学习动力的增加。

技术上，作为一个网页上的管理系统，我不仅需要写出后端的 API，还需要自己去写前端界面，用 JS 函数调后端 API。这项工作是非常庞大的，在不断的开发过程中衍生了许多问题，解决这些问题的过程自然增长我的知识和技术。比如用 Git 来进行版本控制、用 Vim 来编写代码、用 JQuery 来进行前后端交互、用 docker 来一键部署 Mysql 数据库等等。这些都是宝贵的技术知识，增加了我的开发经验。

思想及学习上，我们作为软件开发人员，以后从业的过程中必然会有许多新知识需要学习。这次毕设所学到的东西不一定能在以后能用到，但我通过这次毕设所提升的学习能力以及对原意学习新知识的思想态度，对我来说无疑是一笔宝贵的财富，这也是毕设中我最大的收获。

还有一方面就是这次毕设中，我阅读了许多 docker 的源代码和官方的英文文档，提升了自己的编程能力以及英文阅读能力，也通过他们规范的代码学到了怎样做一个严谨、规范的软件开发人员。

毕设工作的结束也意味着大学生涯即将结束。我从小开始学习，每个阶段都能学到新知识、能够成长许多。而大学，可以说是我真正学到了东西、学习东西最多的地方，在这里我也确立了自己的三观、人生的方向。但通过这次毕设，我认识到了我的学习之路并不能止于大学，还有很多的知识、技术需要我去学习。总之，路漫漫其修远兮，吾将上下而求索。

参考文献

[1] 马越、黄刚 . 基于 Docker 的应用软件虚拟化研究 .软件 2015

[2] DATADOG 官网. 8 surprising facts about real docker adoption . https://www.datadoghq.com/docker-adoption/ . 2017.4 .

[3] 龚正. Kubernetes 权威指南：从 Docker 到 Kubernetes 实验全接触. . 北京：电子工业出版社，2016.10 .

[4] Kubernetes 官网 . What is Kubernetes . https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ . 2018

[5] Docker 官网 . Docker overview . https://docs.docker.com/engine/docker-overview/ . 2018.

[6] 刘思尧，李强，李斌 . 基于 Docker 技术的 容器隔离性研究[J]. 软件, 2015, 36(4): 110-113 .

[7] C Boettiger .An introduction to Docker for reproducible research .Acm Sigops Operating Systems Review 2015 .

[8] 汪凯，张功萱，周秀敏 . 基于容器虚拟化技术研究[J]. 计算机技术与发展，2015,08:138-141.

[9] 浙江大学 SEL 实验室. Docker-容器与容器云[M]. 人民邮电，2015.

[10] 秦云霞 . 试谈 Go 语言的面向对象技术. 电脑编程技巧与维护. 2014 (24) :13-14 .

[11] beego 官网 . beego 简介 . https://beego.me/docs/intro/ . 2018 .

[12] Anton A.Chuvakin . 日志管理与分析权威指南 . 北京：机械工业出版社，2014 .

声　明

本人声明所呈交本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得四川大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

本学位论文成果是本人在四川大学读书期间在导师指导下取得的，论文成果归四川大学所有，特此声明。

学位论文作者（签名）_____

论文指导教师（签名）_____

年　月　日

致　谢

感谢毕设过程中四川大学李晓华老师与成都精灵云科技有限公司张行才老师的指点，感谢我们软件学院同学们的帮助。

感谢大学四年学习过程中的所有任课老师的指导，感谢黎红友辅导员在我大学学习中迷茫时的指点，同时感谢所有在学习上、生活中帮助过我同学、室友以及朋友。

感谢大家，没有大家的帮助就没有今天的我。

附录 外文翻译

原文：

A performance comparison of container-based technologies for the Cloud

Zhanibek Kozhirbayev , Richard O. Sinnott

ABSTRACT

Cloud computing allows to utilize servers in efficient and scalable ways through exploitation of virtualization technology. In the Infrastructure-as-a-Server (IaaS) Cloud model, many virtualized servers (instances) can be created on a single physical machine. There are many such Cloud providers that are now in widespread use offering such capabilities. However, Cloud computing has overheads and can constrain the scalability and flexibility, especially when diverse users with different needs wish to use the Cloud resources. To accommodate such communities, an alternative to Cloud computing and virtualization of whole servers that is gaining widespread adoption is micro-hosting services and container-based solutions. Container-based technologies such as Docker allow hosting of micro-services on Cloud infrastructures. These enable bundling of applications and data in a manner that allows their easy deployment and subsequent utilization. Docker is just one of the many such solutions that have been put forward. The purpose of this paper is to compare and contrast a range of existing container-based technologies for the Cloud and evaluate their pros and cons and overall performances. The OpenStack-based Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR) Research Cloud (www.nectar.org.au) was used for this purpose. We describe the design of the experiments and benchmarks that were chosen and relate these to literature review findings.

1. Introduction

Nowadays Cloud platforms and associated virtualization tech- nologies are in great demand. Many software companies such as VMware (VMware), Citrix (Xen), Microsoft (Hyper-V) domi- nate the virtualization market with their solutions and companies which are oriented to hardware such as Intel and AMD now offer advanced processors to support virtualization. Collectively these technologies are utilized for server consolidation—typically in data centers that offer large collections of servers for external communities in a flexible manner through for example elastic scaling. There has been much research related to virtualization performance. Some of these works concentrate on HPC facilities [1,2] whilst others focus on Cloud environments. Previous research identified that technologies which utilize hypervisor-based virtualization, face high performance overheads. In addition they suffer from I/O limitations and hence are

normally avoided in HPC environments. Recently container-based virtualization and support for microhosting services has gained significant acceptance since it provides a lightweight solution that allows bundling applications and data in a simpler and more performance-oriented manner that can run on different Cloud infrastructures. This way of dealing with virtualization offers horizontally scalable, deployable systems without the difficulty of high-performance challenges of traditional hypervisors and the overheads of managing large scale Cloud infrastructures [3]. In this work, we undertake a review of microhosting services and perform a number of experiments to provide a comprehensive performance evaluation of container-based virtualization technologies for the Cloud. We focus in particular on representative systems: Docker [4 – 6] and Flockport (LXC) [7,8] as leading offerings.

The purpose of this work is to compare the performance of container-based virtualization technologies on the Cloud. This work focused specifically on CPU, memory as well as I/O devices capacities. In order to meet these criteria four objectives were defined:

- Critically review performance experiments of related works to measure the performance of different existing virtualization technologies on different environments;
- Identify performance-oriented case studies in order to evaluate the performances of virtualization technologies on the Cloud;
- Implement several case studies to evaluate the performances of the microhosting technologies; and
- Compare the results obtained from the performed experiments and identify the pros and cons of microhosting technologies under these experimental conditions.

This paper is organized as follows: Section 2 presents an overview of various technologies for virtualization and related work on benchmarking applications. More precisely, it describe container-based virtualization and hypervisor-based virtualization as well as representative examples of these solutions including Docker, LXC (Flockport) and CoreOs Rocket. Section 3 compare the key features of the container-based technologies that may influence to their performance. The design of the experiment executed to evaluate virtualization performance is described in Section 4 including the Cloud environment, the system architecture and benchmarking tools that were used to perform the benchmarking case studies. The implementation details of the experiments and the results of the performance comparison are presented in Section 4. The summary of the performance comparison and areas of further research are given in Section 5.

2. Background and related work

Virtualization of resources typically includes utilizing an additional software layer above the host operating system with an eye to handle multiple resources. Such virtual machines (VMs) can be considered as a separate execution environment. Several approaches are used for virtualization purposes [9]. One

popular technique is hypervisor-based virtualization. Well-known solutions based on hypervisor-based virtualization are: KVM and VMware. In order to use this kind of technology there should be a virtual machine monitor above the underlying physical system. Each virtual machine also has support for (isolated) guest operating systems. It is quite possible that one host operating system may support many guest operating systems within this virtualization approach [9].

Container-based virtualization technology represents another approach. In this model, the hardware resources are divided by implementing many instances with (secure) isolation properties [9]. The difference between the two technologies can be seen in Fig. 1. Here, guest processes obtain abstractions immediately with container-based technologies as they operate through the virtualization layer directly at the operating system (OS) level. In hypervisor-based approaches however there is typically one virtual machine per guest OS [9]. One OS kernel is typically shared among virtual instances in container-based solutions. Therefore, there is an assumption that the security of this kind of approach is weaker than with hypervisors. From a users perspective, containers operate as autonomous OSs, which appear able to run independently of hardware and software [10].

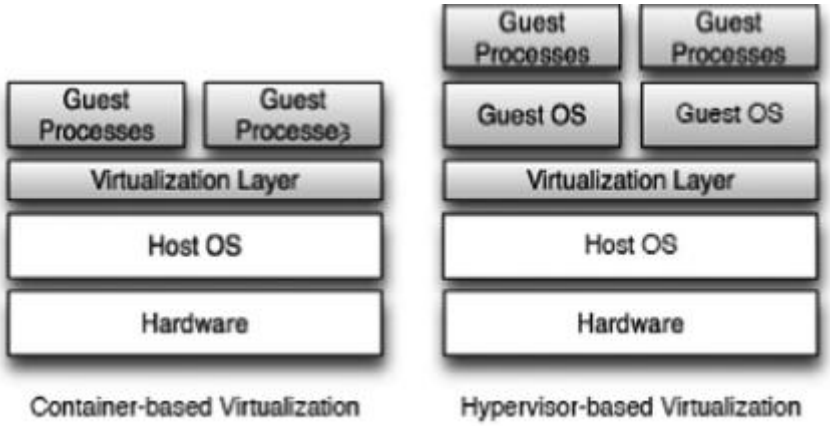

Fig. 1. Comparison of container-based and hypervisor-based approaches

Biederman [11] argues that kernel namespaces are responsible for handling the isolation property of containers. This is considered as a Linux kernel characteristic attribute, which allows processes to obtain the necessary levels of abstraction. Despite the fact that there is no interaction between containers and outside of a namespace layer, there is isolation between the Host OS and Guest Processes and each container has its own operating system. According to Biederman [11], file systems, process identifiers, networks as well as inter-process communication are considered to be isolated over namespaces. However, there is a restriction of the resource utilization in accordance with process groups in container-based virtualization technologies. This procedure is managed by cgroups [12]. To be precise, cgroups are responsible for determining the priority for CPU, memory as well as I/O utilization in container-based virtualization. However certain technologies, which use containers implement their management of the resources in accordance with the consistency of cgroups.

Using such container-based solutions allows for the dynamic deployment and use of micro-services in bundled hosting environments. Micro-service patterns are not a new idea in software architecture. Nowadays they are widely recognized as an efficient solution to develop applications. Prior to the micro-services architecture, the general approach for service development was to create largely monolithic applications. From a functional point of view, this required a single environment that handled all the things. In Cloud environments, many of these issues can be overcome through scripting approaches supporting Infrastructureas-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-asa-Service (SaaS). However, such solutions are challenged when there is many ad hoc projects and communities involved with their own diverse software and data demands, e.g. where communityspecific virtual machine images are not available for the Cloud. In such environments, light-weight Cloud-enabled solutions are beneficial. Micro-services are one such model.

The main concept of the micro-services architecture is ‘‘divide and conquer’’. Fundamentally, micro-services replace a single larger code base with multiple small-scale code foundations controlled by small, agile groups. These code foundations have a single API relationship with one another. The benefits behind this concept are that every group may work in isolation and be protected/disconnected from one another—so called exemption cycles. However, these exemption cycles may be connected in certain cases, e.g. when there is dependency between services in different groups [13].

A range of container-based micro-hosting services now exists. The most established of these are Docker, CoreOS and LXC. Docker provides a less complicated way to wrap an application into a container including the accessories it requires for execution. This action is conducted through a targeted set of tools and integrated Application Programming Interface Guiding technologies with kernel-level structure, e.g. Linux containers, control groups as well as a copy-on-write file system. Acting as a file system for containers Docker is dependent on Advanced Multi-Layered Unification File system (AuFS). AuFS is able to explicitly superimpose single or multiple available file systems. It enables Docker to utilize images required for the container’s foundation. For instance, a person may use a Ubuntu image that in turn might provide the foundation for multiple other containers. With the help of the Advanced Multi-Layered Unification File system Docker utilizes a single copy of Ubuntu. This dramatically saves storage and reduces the use of memory in line with prompt launching of containers. AuFS has one more advantage which enables it to create image versions. Every latest version is marked as diff, which is a file collation benefit that identifies the difference between two files. This allows for example for image files to be reduced in volume. Another efficiency of AuFS is that every modification made on image versions can be tracked— much like software development code versioning systems [4] .

CoreOS is a comparatively recent distribution of Linux, which has been designed to provide characteristics required to operate stacks of software systems. This technology provides a reduced Linux kernel to

decrease overheads to the maximum. Moreover, CoreOS supports a cluster with tools in order to ensure redundancy as well as methods of protecting systems from failure

Recently, CoreOS introduced a new product—rocket container runtime (rkt) [14], which runs the application container specification. The main purpose of this technology is to build a specified model for a container. This approach also supports Amazon Machine Images. The rocket container runtime is an option to Docker, which includes advanced security as well as other demands necessary for production activities on servers. The rocket container runtime is aligned with the Application Container specification, which offers a novel set of formats for containers that allows them to be easily carried or moved. In Docker, every process runs via a daemon and from security point of view it does not provide as much assurance as Rocket. In order to correct this phenomenon it has been suggested that Docker should be rewritten completely.

LXC is maintained in the standard Linux kernel and the project enables instruments to manage container and OS images. Containers can be considered as lightweight OS facilities which execute together with the host OS. Containers do not imitate the hardware layer and therefore they can execute at almost native speed in the absence of other performance overheads. In their standard utilization, applications as well as web stacks are established and put in a particular form in bare-metal servers for testing purposes. For example, PHP, MySQL, Nginx and Drupal can be installed and configured to run in parallel. However, currently applications are dedicated to the machine where they have been established and cannot simply be migrated. A virtual machine can be installed and migrated and it may give some sense of being easily moved but this compromises performance.

The LXC container can give almost bare-metal throughput and the capability to simply migrate throughout systems by establishing similar stack into containers. An LXC container has improved performance and flexibility, leading to the illusion that there is a separate server. The containers can be cloned, backed up and snapshotted. LXC makes it easy to manage containers and introduces new degrees of flexibility in executing and launching apps. Flockport ensures web stacks as well as software in LXC containers can be launched on any Linux-based machines. LXC containers directly support flexibility and throughput, whereas Flockport is a tool to distribute LXC containers and simplify their utilization.

Several researchers have concentrated on reducing the difference between the virtualization technologies and non-virtualized approaches in terms of performance and optimization. The methodology and tools used are typically different in each of these researches. Moreover, the examined and compared compositions of techniques vary also. For instance, the gap between native systems and visualized versions of them and the performance differences between container-based and hypervisor-based virtualization technologies are examined and compared in recent research papers. We note that this is a fast moving field and hence some of the earlier papers are based on out of date software. Moreover, they do not perform analysis on recent visualization technologies.

Four different hypervisor-based virtualization technologies were compared by Hwang et al. in [15]. They did not discover a hypervisor with any noticeably higher performance. Accordingly, the suggestion offered by them was to use various software as well as hardware platforms in Cloud facilities in order to satisfy customer requirements.

Abdellatief et al. [16] performed a performance comparison of technologies such as VMware, Microsoft Hyper-V, and Citrix Xen in different scenarios. The methodology of evaluation they used was to apply customized SQL instances. With the help of this method, they simulated millions of products, customers and orders. The same kind of analysis was performed by Varrette et al. [17], but with different technologies and associated testbed environment. They utilized kernel-based virtual machines in place of Microsoft Hyper-V, with experiments related to high performance computing. Their experiments were focused on the consumption of power, energy efficiency as well as scalability.

Despite the fact that there was an inconsistent demonstration of virtualization overheads, [17] identified that the virtualization layer for almost every hypervisor provided significant influence on the performance of the virtualized environments, especially for high performance computing domains.

Recent publications consider the similarity or dissimilarity between hypervisors with container approaches. According to Dua et al. [18], containers are becoming popular in supporting PaaS facilities. The representatives of both types of virtualization namely KVM, Xen, and LXC were benchmarked by Estrada et al. [19]. The main methodology of their research was to measure the similarities as well as the differences of the runtime performance of each mentioned technology. The basis of their experiments and benchmarking was supporting sequence-based applications.

Felter et al. [5] also compared technologies for hypervisor and container-based system namely KVM and Docker respectively. They conducted a comprehensive analysis in terms of CPU, memory, storage as well as networking bandwidth and latencies. According to their benchmark results, the performance of the Docker container was almost the same as the “bare metal” system. Their benchmarks were based on memory transfers, floating point handling, network resources, block I/O as well as database capacities. However, the authors work did not examine methodically the influence of containers on traditional hypervisors.

Utilizing containers to deploy applications in an efficient and repeatable manner was introduced in the Heroku PaaS provider [20]. Heroku offers a container as a process with additional isolation properties instead of considering it as a virtual server. As a consequence application deployment containers provide a lightweight technology with insignificant overheads and almost the same isolation as virtual machines. It also has resource sharing properties as standard processes. Such containers were heavily utilized by Google in their infrastructure. Moreover, a standard format for images as well as management tools for application containers was offered by Docker.

The main distinction compared to previous publications and this work is that the analysis made in this work is specifically related to benchmarking the performance of open source container-based technologies and hence identifying their associated advantages and disadvantages.

3. Comparison of micro-hosting environments

Each container-based technology has its own features. This section presents some key characteristics of the current leading container-based technologies that may have impact on the performance. We note that even though CoreOs Rocket was identified in the previous section, the performance evaluation for Rocket was not conducted since CoreOS have not yet released an official version of their product.

3.1. *Docker*

Docker utilizes several Linux kernel features in order to run containers in an isolated way [9] .

• **namespaces**: Docker employs namespaces to deploy containers. There are several types of namespaces utilized by Docker to perform the tasks of creating isolated containers [4]:

– Docker uses *pid* as a base for containers which ensures that all processes in the container are not allowed to affect processes in other containers;

– It uses *net* in order to manage network interfaces or more precisely, it provides isolation of the system resources regarding networking;

– *ipc* is used to provide isolation to specific inter-process communication (IPC) resources, namely System V IPC objects and POSIX message queues. This means that each IPC namespace has its own inter-process communication resources.

– In order to allow processes to have their own view of a filesystem and of their mount points Docker uses *mnt* namespace.

– Isolation of kernel and version identifiers is performed through *uts*.

• **control groups**: Docker executes *cgroups* so that existing containers can share available hardware resources and it is possible that there might be a limitation in use of these resources at a given time.

• **union file system** is a file system that functions by establishing layers which are utilized to provide the building blocks for containers.

• **container format** is considered as a wrapper that integrates all of the fore-mentioned mechanisms.

3.2. *LXC*

Linux Container is a container-based virtualization technology that enables the building of lightweight Linux containers without difficulty by use of a common and flexible API and associated implementations [21]. On the other hand, Docker is an applicationcentric technology based on containers. They share some

common features but have numerous differences. Firstly, LXC is an operating-system-level virtualization technique for executing several isolated Linux containers on a single LXC host. It does not make use of a virtual machine, but rather allows utilizing a virtual environment which has its own CPU, memory, blocking I/O, network as well as the resource control mechanism. This is offered through the *namespaces* and *cgroups* features in the Linux kernel on the LXC host. It is similar to a *chroot*, but provides much more isolation. Various virtual network types and devices are supported by LXC. Secondly Docker utilizes fixed layers to enable reuse of well-structured designs, but this can be at the cost of complexity as well as throughput. The restriction of one application per container reduces the utilization possibilities. With LXC, single and/or multiple applications may be created. Furthermore LXC allows multiple system containers to be created, which may be clones of just one sub-volume that can be run by utilizing a *btrfs* file. This feature of LXC can tackle sophisticated issues of file system levels. Thirdly, LXC offers an extensive list of facilities and privileges to design as well as execute containers. Finally LXC enables the creation of unprivileged containers that ensure nonroot users can build containers. Docker does not yet support this feature.

3.3. *CoreOS rocket*

Rocket [14] is a container-based technology introduced by CoreOS, which provides an alternative to Docker. Both Rocket and Docker introduce automation of the application deployment in the form of virtual containers that can execute independently based on the server's characteristics. However, whilst Docker has developed into a sophisticated environment, which supports a diversity of requirements as well as operations, Rocket is structured to perform simple functions but in a secure manner targeted to application deployment. Rocket functions as a commandline instrument for executing application containers that are descriptions of image designs. Rocket is focused on the application container specification introduced by CoreOS as a composition of descriptions that allows for a container to be easily migrated.

As recognized by Polvi [14], Rocket may be more difficult to use compared to Docker since Docker simplifies the whole process of constructing a container through its descriptive interface. [14] argues that Rocket should stay as a command-line based environment and be less likely to change.

4. Evaluation methodology and benchmarking

There are many perspectives that can be used to compare technologies, especially from a performance perspective. In order to evaluate container-based technologies from the perspective of their overheads, it was necessary to understand (measure) the overheads incurred based upon non-virtualized environments. The analysis undertaken here focused upon a range of performance criteria: the performance of CPU, memory, network bandwidth and latency and storage overheads. In all of the benchmarking multiple

experiments were repeated 15 times to assess the accuracy and consistency of the various results. Average timing and standard deviation was recorded.

The Cloud environment that was used for these activities was the Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR) Research Cloud (www.nectar.org.au). NeCTAR provides a Cloud environment for all researchers across Australia. It offers access to 30,000 servers across eight availability zones— typically located in the State capitals (Melbourne, Canberra, Hobart etc.). The NeCTAR project is led by the University of Melbourne and funded by the Department of Education. NeCTAR utilizes the OpenStack middleware to realize the Cloud infrastructure.

The performance studies are all executed on NeCTAR Research Cloud instances. The following instance configurations were used for performing the experiments: Model: Processor: AMD Opteron 62xx class @ 2.60 GHz; Processor ID: AuthenticAMD Family 21 Model 1 Stepping 2; Memory: 3955 MB; OS: Ubuntu 12.04 (64-bit)

The virtualization technologies and their versions are given in Table 1. For conformity, all Docker and Flockport (LXC) containers utilized a Ubuntu 64 bit system image. Moreover, both of them were running on the host OS which itself was based on Ubuntu 12.04 64-bit.The outcomes of the performed experiments are demonstrated in this section. As mentioned previously, the chosen benchmark tools evaluate CPU, memory, network bandwidth and latency, storage overhead performances.

Table 1
The virtualization technologies and their versions.

Virtualization technologies	Version
Docker	1.4.0
Flockport (LXC)	1.1.2

Table 2
pbzip2 compressor results.

Platforms	Wall clock (s)
Native	13.7
Docker	14.8
Flockport (LXC)	14.9

Table 3
Multi-core efficiency results from Y-cruncher.

Platform	Multi-core efficiency
Native	99.2%
Docker	99.3%
Flockport	99.4%

4.1. *CPU performance*

The first scenario to evaluate the CPU performance was based on use of a compressor. Compression is a regularly utilized module of Cloud environments processing. PBZIP2 [22] is a parallel realization of the bzip2 block-sorting data compression utility. It utilizes *p* number of threads and can reach an almost linear acceleration. pbzip2-1.1.12 version was used in order to compress a file. An input file (enwik8) [23], which is 100 MB dump data and frequently applied for compression testing purposes. In order to concentrate on compression 900 kB BWT Block Size and 900 kB File Block Size was used.

The performance of the pbzip2 compressor is presented in Table 2. From the perspective of CPU evaluation, Docker performs slightly better than LXC. In case of Flockport, average elapsed time is 14.9 s and standard deviation is ±0.03 s, whilst average time by Docker is 14.8 s and standard deviation is ±0.01 s. However, the size of input file should be taken into consideration. If the size increases, the difference of results can be significant.

The second CPU benchmarking tool is Y-cruncher [24], which is used to compute Pi. It is typically

executed as a stresstesting tool for CPUs and frequently used as a test for multithreaded tools running in multi-core systems. Besides calculating the value of Pi, Y-cruncher can also compute a range of other constants. Y-cruncher performs various outcomes such as multicore efficiency, computation time, and total execution time. The total time is applied to confirm the outputs, and it includes the total computation time added to the time requested to exploit and perform the outcome.

The performance results of the Y-cruncher benchmarking tool can be seen in Fig. 2. In terms of computation time, Docker performs similarly to the native (non-virtualized) system, whereas Flockport takes on average about 2 s longer. The multi-core efficiency results of these systems are presented in Table 3. This assessment describes how the CPU is effectively utilized in calculation of Pi. This pattern also shows that Docker shows marginally better performance than Flockport.

The next benchmarking tool explored was the standard HPC benchmarking tool: Linpack. There are two options for this tool. The first one is an optimized version by Intel [25], whereas the second one [26] allows to operate on all machines and not only Intel machines. Linpack finds the solution for a system of linear correspondences utilizing an approach that performs 'lower upper' decomposition of numerical analysis with partial pivoting. A great number of computational operations consist of multiplying a process of a scalar with a vector in double-precision floating-point format as well as processes for adding the outcomes to different vectors. The benchmarking tool is built on the basis of linear algebra functions which are modified for the chosen computer architecture. The main Linpack function utilizes a random matrix M of size N and a vector V which is determined as $M * X = V$. The Linpack benchmark tool performs two steps as follows: 'Lower Upper' decomposition of M, and subsequently the 'Lower Upper' decomposition applied to solve the linear problem $M * X = V$. The outcomes of Linpack are typically provided in MegaFLOPS—floating-point operations per second. Executing this tool and increasing the value of N, it can be seen from experiment that the behavior of the CPU usage changes and various phases can be identified: rising zone, where no challenge occurs in local memory or processor; flat zone, where challenges occur in processor functionality, and the decaying zone, where challenges occur in the local cache memory.

The results of Linpack run in the micro-hosting environments can be seen in Fig. 3. These outcomes are obtained by applying a specific scenario with $N = 1000$. As seen, even though the gap between them is relatively small, the performance of Flockport in executing the Linpack benchmark is slightly better than Docker.

The final tool used to evaluate the performance of the CPU was Geekbench [27]. This tool is useful to test performance of the Floating Point Unit as well as the throughput of memory systems. An upper bound for the above-mentioned throughput features was assessed by this application. In comparison with Ycruncher, Geekbench supports the evaluation of single as well as multi-core architectures. It can execute

various workloads generating multiple indexes such as Integer Performance, Floating Point Performance, and Memory Performance. Moreover, the index of the complete system can be generated

As can be seen from Figs. 4 and 5, there is no significant difference in either single-core or multi-core testing of results. However, regarding memory performance, Flockport produces approximately 100 points and 200 points more than Docker performance in single-core and multi-core testing respectively.

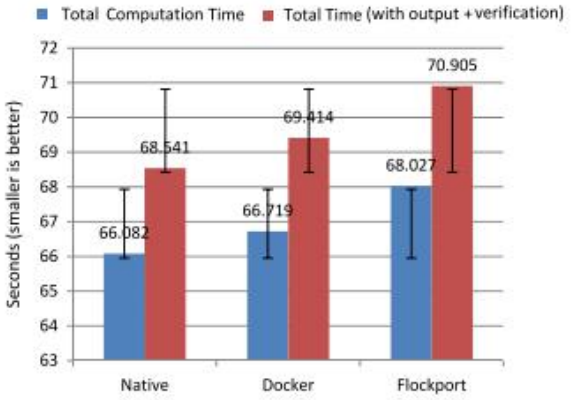

Fig. 2. Performance results of the Y-cruncher benchmark.

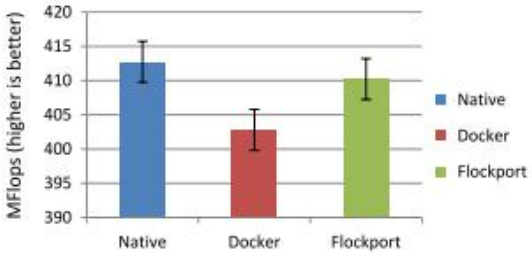

Fig. 3. Linpack results (where $N = 1000$).

4.2. *Disk I/O performance*

A key aspect of performance is the evaluation of disk I/O performance, specifically volumes given as a non-ephemeral storage, were attached to instances. These volumes were created in the same availability zone as the associated instances. The first applied benchmarking tool used to evaluate the micro-hosting environments was Bonnie++ [28]. Bonnie++ is an open-source application that describes disk throughput. Bonnie++ has to be configured for different cases so a common test file was used. A 4 Gb dataset was used for this purpose. Fig. 6 presents the Bonnie++ outcomes for Block Output as well as Block Input based upon sequential write and read respectively. All three systems produce almost same results in terms of sequential write. However Flockport shows a slightly better performance compared to Docker regarding sequential reading of files. Through the Bonnie++ benchmarking application, the speed can be assessed when reading, writing and flushing processes take place in a file. Table 4 shows the Random Write Speed as well as Random Seeks for each system .There is relative alignment between random write speed outcomes at which a file is read and then written and flushed to the disk as depicted in Fig. 7. However, the arrangement of the systems is different in their random seek evaluation which shows the number of blocks which Bonnie++ can seek to per second. In this case, Flockport has 100% better results than Docker and is almost 6% better than the native platform.

To further evaluate the disk I/O throughput, the Sysbench benchmark tool [29] was utilized. This tool involves many modules as the basis of design. It also can be used as a cross-platform as well as multi-threaded measurement software for measuring operating system characteristics. The main concept of Sysbench is to gain a representation at a fast speed about system throughput without deploying database

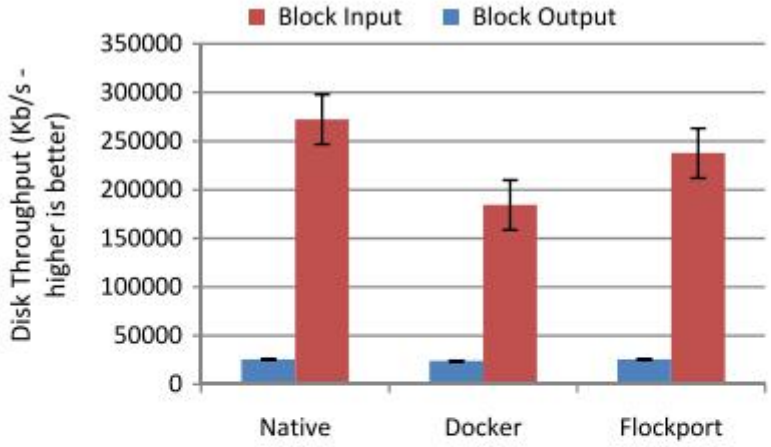

Fig. 6. Bonnie++ results for sequential write and sequential read.

Table 4
Random write speed and random seeks.

Platform	Random write speed (kb/s)		Random seeks	
Native	24829	%	3741	%
Docker	22494	−9.4%	389.2	−89.6%
Flockport	24524	−1.2%	3961	+5.9

systems. Actual versions of the suite allow evaluating the system characteristics such as file I/O throughput, memory allocation and transferring speed scheduler throughput. In performing this scenario, the tool was used to assess the platforms performance to read as well as write from specified files. In order to evaluate file IO performance, a test file should be created that should be bigger in size than the available RAM. The size of the test file used here was 35 GB—specifically 128 files of 270 Mb each

The results of this measurement are given in Table 5. Total time taken by event execution on Docker and Flockport is 27.488 s with standard deviation of ±0.0003 s and 27.491 s with standard deviation of ±0.0001 s respectively. There is no significant difference.

The results achieved by Sysbench reflect those obtained through Bonnie++. Native as well as Flockport showed approximately the same outcome without any discernible difference. However, Docker performed better than both platforms in some runs.

For the purpose of thoroughness, it should be noted that a default file format was utilized for disk images for Native as well as Flockport, whereas Docker applies the advanced multi layered unification file system that ensures layering as well as image versioning. As such, other tests should be conducted in this experiment.

4.3. *Memory performance*

The evaluation of Memory I/O performance is presented in this subsection. The benchmark tool used to test the microhosting environments was the STREAM software [30]. STREAM assesses memory

throughput utilizing straightforward vector kernel procedures. The outcomes of four procedures namely Copy, Scale, Add as well as Triad are generated by this tool. These procedures and how they are calculated are shown in Table 6. According to the STREAM software, there is a hard relation between the evaluated throughput and the size of the CPU cache.Moreover, there is rule that every stream array has to be at least four times the accessible cache memory size. Therefore, the size of the stream array in the software must be established correctly. As seen in Fig. 7, the difference between the results is not significantly high, but Docker produces slightly better results than Flockport and is nearly the same as the native platform.

Table 5
Sysbench results.

Platform	Read (Gb)	Written (Mb)	Total transferred (Gb)	Throughput (Mb/s)	Elapsed time (s)
Native	1.22	834.17	2.04	20.85	27.568
Docker	1.21	829.38	2.02	20.73	27.488
Flockport	1.17	796.88	1.95	19.92	27.491

4.4. *Network I/O performance*

This subsection describes the performance of container-based environments with regards to their Network I/O. Any containers running on the same host, more precisely on the same host bridge, can contact each other through IP. Network address translation (NAT) networking principles was used for all container traffic. This makes possible for all containers to contact outside world including other Docker containers on different hosts, but it does not allow the outside network to communicate with the containers. This regulation might be avoided by mapping container ports to ports on the host network interface. Linux containers have the same networking. The test cases were conducted on two Docker containers located on two different hosts. The same condition was applied to Flockport containers also. In both cases, one acted as a server and the other as a client. For native conditions, two identical NeCTAR Research Cloud instances were used

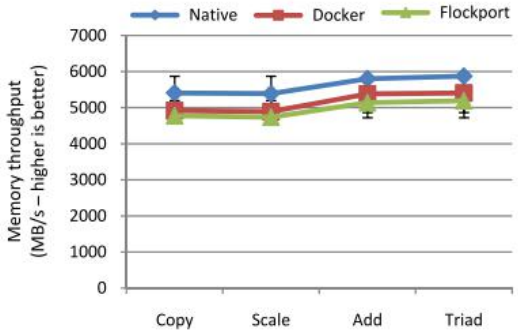

Fig. 7. STREAM results.

Table 6
Stream procedures.

Procedure	Kernel
Copy	$x[i] = y[i]$
Scale	$x[i] = q * y[i]$
Add	$x[i] = y[i] + z[i]$
Triad	$x[i] = y[i] + q * z[i]$

The Netperf benchmark tool [31] was used to measure Network I/O. This tool has many predetermined subtests to evaluate network throughput between a server and clients. It enables execution of data transfers operating in a single direction either with TCP or UDP protocol. Time spent to establish connections between the netperf client and the netserver is not comprised in these evaluations. The assessment outcomes present the throughput of arriving packets is shown in Table 7

Table 7 shows the outputs for both Docker and Flockport and for both TCP_STREAM and UDP_STREAM. For the TCP_STREAM test case, Docker performed almost 200 Mbps better than Flockport, whereas for the UDP_STREAM test case, Docker again offers approximately 150 Mbps better performance. Moreover, as mentioned, the Netperf benchmark tool has assessment cases including request and response, which measure the amount of TCP as well as UDP transactions. The following establishment connection occurs during these cases: the netperf client dispatches requests to the netserver and the netperf server dispatches a response to the netperf client.

As depicted in Table 8, Flockport again produces the worst results for both TCP_RR as well as UDP_RR test cases.

In order to get a thorough understanding of Network I/O performance, a second tool was applied to test Network throughput: the Iperf suite [32]. Iperf offers a complete suite for evaluating connection performances using either TCP or UDP protocols. Table 9 shows the results of Iperf tests for both TCP and UDP traffic.

It can be seen from Table 9 that Docker is slower when compared to the previous test case on I/O performance. The possible reason is the TCP window size and the UDP buffer size. That is, the quantity of data that can be buffered in the time window of a given connection without verification differs. The sizes of data can be between the range of 2 and 65,535 bytes, but they were small in *Iperf* by default (60.0 kB) and were not tuned.

5. Conclusions and future work

Container-based technologies are challenging hypervisor-based approaches as the basis for Clouds. Modern container-based approaches are considered to be lightweight. In this paper, a thorough performance assessment of leading micro-hosting virtualization approaches was presented with specific focus on Docker and Flockport and their comparison with native platforms. As noted, CoreOS was not considered due to the unavailability of the systems at the time of writing.

Regarding the results of performed experiments, many common patterns can be seen. As shown there were roughly no overheads on memory utilization or CPU by either Docker or Flockport, whilst I/O and operating system interactions incurred some overheads. The overheads in these cases appear by way of additional cycles for every input–output operation. Therefore, applications with increased input–output operations have more disadvantages compared to applications with lower input–output demands. As a consequence, input–output latency is exacerbated by these overheads. The CPU cycles required for utility tasks can also cause performance degradation.

Docker includes several other capabilities such as Network Address Translation, which helps to reduce some of the difficulties of Docker container utilization. However, these capabilities directly impact input on

throughput quality. As a result, Docker containers utilizing no extra features can be no quicker compared to Flockport in some cases. Software that is either file system or disk intensive must use the advanced multi layered unification file system by applying volumes. The impact of Network Address Translation may be removed by utilizing nethost. However, this negates the advantages of network namespaces. Eventually, the design of one IP address for each Docker container as suggested by the Kubernetes [33] may support quality assurance as well as throughput

Table 8
Netperf TCP_RR and UDP_RR results.

Platform	TCP_RR (Transfer rate per second)	UDP_RR (Transfer rate per second)
Docker	44363.03	45093.28
Flockport	39321.02	40625.07
Native	48451.11	49221.17

Table 9
Iperf results.

Platform		TCP	UDP
Docker	Interval	0.0–10.0 s	0.0–10.0 s
	Transfer	966 MB	11.9 MB
	Bandwidth	810 Mb/s	10.0 Mb/s
Flockport	Interval	0.0–10.0 s	0.0–10.0 s
	Transfer	1.34 GB	11.9 MB
	Bandwidth	1.15 Gb/s	10.0 Mb/s
Native	Interval	0.0–10.0 s	0.0–10.0 s
	Transfer	1.64 GB	1.19 MB
	Bandwidth	1.41 Gb/s	1.00 Mb/s

The generated results here may provide some direction of how the architecture of Cloud environments should be designed. Thus current considerations are that IaaS is more to utilization of virtual machines and PaaS developed to utilize containers. If IaaS is designed to use containers instead, they can provide better throughput as well as simpler deployment opportunities. Moreover, containers can reduce the difference between IaaS and "bare metal" systems because they support the management and give almost the same performance as native systems. Another question here is launching containers within virtual machines, thus these can conflict with the throughput overheads of virtual machines while providing no advantages compared to launching containers immediately on native hosts. Such pragmatic considerations should be factored in to the choices of the technologies used to build and manage IaaS and PaaS systems for performance demanding communities. Multi-tenancy is a further very important problem in Clouds, and containers or micro-service applications typically consist of multiple services running in separate containers sharing the same resources. The implications and performance impact on shared resources will be conducted in future work

Acknowledgment

The authors would like to thank the NeCTAR Research Cloud (www.nectar.org.au) for the resources used to perform these investigations.

References

[1] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. Shin, Performance Evaluation of Virtualization Technologies for Server Consolidation, Enterprise Systems and Software Laboratory HP Laboratories Palo Alto HPL-2007-59, 2007

[2] N. Regola, J. Ducom, Recommendations for virtualization technologies in high performance computing, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CloudCom, 30 2010-dec. 3 2010, pp. 409–416.

[3] N. Slater, Using Containers to Build a Microservices Architecture, viewed 1 April 2015, URL https://medium.com/aws-activate-startup-blog/usingcontainers-to-build-a-microservices-architecture.

[4] Docker – Build, Ship, and Run Any App, Anywhere, viewed 1 April 2015, URL http://www.docker.com.

[5] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An Updated Performance Comparison of Virtual Machines and Linux Containers, viewed 1 April 2015, URL http://www.research.ibm.com/.

[6] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment, Linux J. 2014 (239) (2014).

[7] Flockport, viewed 1 April 2015, URL http://www.flockport.com.

[8] P. Rubens, Docker Not the Only Container Option in 2015, IT Business Edge, 2015.

[9] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, C. De Rose, Performance evaluation of container-based virtualization for high performance computing environments, in: 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, (PDP), IEEE, 2013.

[10] S. Soltesz, H. Potzl, M. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, SIGOPS Oper. Syst. Rev. 41 (3) (2007) 275–287.

[11] E.W. Biederman, Multiple instances of the global linux namespaces, in: Proceedings of the Linux, 2006.

[12] P. Menage, Control groups definition, implementation details, examples and api, viewed 1 April 2015, URL http://www.kernel.org/doc/Documentation/ cgroups/cgroups.txt.

[13] L. Marsden, The Microservice Revolution: Containerized Applications, Data and All, viewed 1 April 2015, URL http://www.infoq.com/articles/ microservices-revolution.

[14] A. Polvi, CoreOS is building a container runtime, Rocket, viewed 1 April 2015, URL http://coreos.com/blog/rocket.

[15] J. Hwang, S. Zeng, T. Wood, A component-based performance comparison of four hypervisors, in: 2013 IFIP/IEEE International Symposium on Integrated Network Management, (IM 2013), IEEE, 2013.

[16] E. Abdellatief, N. Abdelbaki, Performance evaluation and comparison of the top market virtualization

hypervisors, in: 2013 8th International Conference on Computer Engineering & Systems, (ICCES), IEEE, 2013.

[17] S. Varrette, M. Guzek, V. Plugaru, V. Besseron, P. Bouvry, HPC performance and energy-efficiency of Xen, KVM and VMware hypervisors, in: 25th International Symposium on Computer Architecture and High Performance Computing, (SBAC-PAD), IEEE, 2013.

[18] R. Dua, A.R. Raja, D. Kakadia, Virtualization vs containerization to support PaaS, in: 2014 IEEE International Conference on Cloud Engineering, (IC2E), IEEE, 2014.

[19] Z. Estrada, Z. Stephens, C. Pham, Z. Kalbarczyk, R. Iyer, A performance evaluation of sequence alignment software in virtualized environments, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (CCGrid), IEEE, 2014.

[20] Heroku, viewed 1 April 2015, URL https://heroku.com.

[21] Linux Containers, viewed 1 April 2015, URL http://linuxcontainers.org.

[22] PBZIP2, viewed 1 April 2015, URL http://www.compression.ca/pbzip2/.

[23] enwik8, viewed 1 April 2015, URL http://mattmahoney.net/dc/textdata.

[24] Y-cruncher – A Multi-Threaded Pi-Program, viewed 1 April 2015, URL http://www.numberworld.org/Y-cruncher.

[25] Intel Math Kernel Library – LINPACK Download, viewed 1 April 2015, URL https://software.intel.com/en-us/articles/intel-math-kernel-librarylinpackdownload.

[26] LINPACK_BENCH – The LINPACK Benchmark, viewed 1 April 2015, URL http://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html.

[27] Geekbench 3 – Cross-Platform Processor Benchmark, viewed 1 April 2015, URL http://www.primatelabs.com/geekbench.

[28] Bonnie++, viewed 1 April 2015, URL http://www.coker.com.au/bonnie++.

[29] Sysbench in Launchpad - SysBench: a system performance benchmark, viewed 1 April 2015, URL https://launchpad.net/sysbench.

[30] J. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, a continually updated technical report (1991-2007), 2015, viewed 1 April URL http://www.cs.virginia.edu/stream.

[31] The Netperf Homepage, viewed 1 April 2015, URL http://www.netperf.org.

[32] Iperf, viewed 1 April 2015, URL http://www.iperf.fr.

[33] Kubernetes, viewed 1 April 2015, URL http://kubernetes.io.

译文：

基于容器的云计算技术性能比较

Zhanibek Kozhirbayev , Richard O. Sinnott

摘要

云计算允许通过利用虚拟化技术以高效和可扩展的方式利用服务器。在基础结构作为一个服务器（IaaS）云模型中，可以在单个物理机上创建许多虚拟化服务器（实例）。现在有许多这样的云提供商正在广泛地提供这样的能力。然而，云计算具有开销，并且可以限制可扩展性和灵活性，特别是当不同需求的不同用户希望使用云资源时。为了适应这样的社区，云计算和整个服务器虚拟化的一个替代方案正在被广泛采用的是微托管服务和基于容器的解决方案。基于容器的技术，如 DOCKER 允许在云基础设施上托管微服务。这些使应用程序和数据捆绑在一起，使得它们易于部署和后续使用。DOCKER 只是许多已经提出的这样的解决方案之一。本文的目的是比较和对比现有的基于云的容器技术的范围，并评估它们的利弊和整体性能。基于 OpenStack 的澳大利亚国家研究开发工具和资源（NeCTAR）研究云（www. nutal.org.au）被用于此目的。我们描述了实验和基准的设计，并选择这些与文献综述结果。

1. 简介

当今云平台和相关的虚拟化技术需求量很大。许多软件公司如 VMware（VMware），Citrix（Xen），Microsoft（Hyper-V）以其解决方案占据着虚拟化市场，面向硬件的公司如英特尔和 AMD 现在提供先进的处理器来支持虚拟化。这些技术共同用于服务器整合 - 通常是在通过弹性缩放等灵活方式为外部社区提供大量服务器的数据中心中。有很多关于虚拟化性能的研究。其中一些作品专注于 HPC 设施[1,2]，而另一些则专注于云环境。以前的研究发现，利用基于虚拟机管理程序的虚拟化技术面临高性能开销。另外，它们受 I / O 限制的影响，因此通常在 HPC 环境中避免。最近，基于容器的虚拟化和对微托管服务的支持得到了重要的认可，因为它提供了一个轻量级的解决方案，允许以更简单，更具性能的方式捆绑应用程序和数据，并且可以在不同的云基础架构上运行。这种处理虚拟化的方式提供了可横向扩展的可部署系统，没有传统管理程序高性能难题的困难，也没有管理大型云基础架构的开销[3]。在这项工作中，我们对微托管服务进行了审查，并进行了大量实验以提供针对云的基于容器的虚拟化技术的全面性能评估。我们特别关注代表性系统：Docker [4-6]和 Flockport（LXC）[7,8]作为领先产品。

这项工作的目的是比较基于容器的虚拟化技术在云上的性能。这项工作专注于 CPU，内存以及 I / O 设备的容量。为了达到这些标准，确定了四个目标：

- 审慎评估相关工作的性能实验，以衡量不同现有虚拟化技术在不同环境下的性能;
- 确定以绩效为导向的案例研究，以评估虚拟化技术在云上的表现;
- 实施几个案例研究来评估微托管技术的性能;和

- 比较从实验获得的结果，并确定在这些实验条件下微宿主技术的优缺点。

本论文的安排如下：第 2 部分概述了虚拟化的各种技术以及基准测试应用程序的相关工作。更准确地说，它描述了基于容器的虚拟化和基于管理程序的虚拟化，以及这些解决方案的代表性示例，包括 Docker，LXC（Flockport）和 CoreOs Rocket。第 3 部分比较了可能影响其性能的基于容器的技术的关键特征。第 4 部分介绍了执行该实验以评估虚拟化性能的设计，其中包括用于执行基准测试案例研究的云环境，系统架构和基准测试工具。第 4 节介绍了实验的实施细节和性能比较的结果。第 5 节给出了性能比较的总结和进一步研究的领域。

2. 背景和相关工作

资源的虚拟化通常包括在主机操作系统上利用额外的软件层来处理多个资源。这些虚拟机（VM）可以被视为一个独立的执行环境。有几种方法用于虚拟化目的[9]。一种流行的技术是基于管理程序的虚拟化。基于基于虚拟机管理程序的虚拟化的众所周知的解决方案是：KVM 和 VMware。为了使用这种技术，应该在底层物理系统之上安装虚拟机监视器。每个虚拟机还支持（独立）客户操作系统。在这种虚拟化方法中，一个主机操作系统很可能支持许多客户操作系统[9]。

基于容器的虚拟化技术代表了另一种方法。在这个模型中，硬件资源通过实施具有（安全）隔离属性的许多实例来划分[9]。这两种技术之间的区别如图 1 所示。这里，访客进程通过基于容器的技术立即获取抽象，因为它们直接在操作系统（OS）级别通过虚拟化层进行操作。然而，在基于管理程序的方法中，每个客户操作系统通常都有一个虚拟机[9]。一个 OS 内核通常在基于容器的解决方案中的虚拟实例之间共享。因此，有一种假设认为这种方法的安全性比虚拟机管理程序弱。从用户角度来看，容器作为独立的操作系统运行，似乎可以独立于硬件和软件运行[10]。

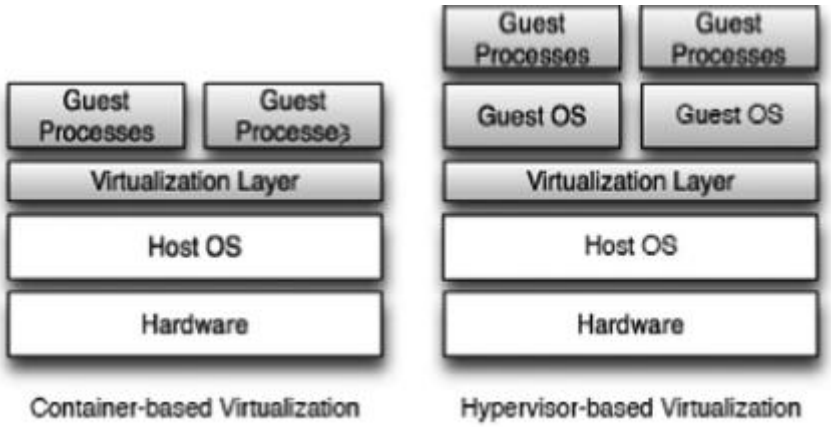

Fig. 1. 比较基于容器和基于管理程序的方法

Biederman [11]认为内核命名空间负责处理容器的隔离属性。这被认为是 Linux 内核特征属性，它允许进程获得必要的抽象级别。尽管容器之间和名称空间层之外没有交互，但主机操作系统和访客进程之间存在隔离，每个容器都有自己的操作系统。根据 Biederman [11]，文件系统，进程标识符，网络以及进程间通信被认为是在命名空间上被隔离的。但是，根据基于容器的虚拟化技术中的进程组限制资源利用率。这个过程由 cgroups 管理[12]。确切地说，cgroup 负责确定 CPU，内

存以及基于容器的虚拟化中的I / O 利用率的优先级。然而，某些使用容器的技术会根据 cgroup 的一致性来执行对资源的管理。

使用这种基于容器的解决方案可以在捆绑托管环境中动态部署和使用微服务。微服务模式在软件架构中并不是一个新概念。如今它们被广泛认为是开发应用程序的高效解决方案。在微服务体系结构之前，服务开发的一般方法是创建基本上单一的应用程序。从功能角度来看，这需要一个处理所有事情的单一环境。在云环境中，可以通过支持 Infrastructureas-a-Service（IaaS），平台即服务（PaaS）和 Software-asa-Service（SaaS）的脚本方法来克服许多这些问题。然而，当有许多特定项目和社区涉及他 们自己的各种软件和数据需求时，这样的解决方案受到挑战，例如，云社区特定的虚拟机映像不可用。在这种环境中，轻量级云计算解决方案是有益的。微服务就是这样一种模式。

微服务架构的主要概念是“分而治之”。从根本上说，微型服务取代了一个较大的代码库，其中包含多个由小型敏捷组控制的小规模代码基础。这些代码基础彼此具有单一的 API 关系。这个概念背后的好处是，每个小组都可以独立工作并相互保护/断开 - 即所谓的豁免周期。然而，这些免除周期可能在某些情况下被连接，例如，当不同群体的服务之间存在依赖关系时[13]。

现在存在一系列基于容器的微型托管服务。其中最成熟的是 Docker，CoreOS 和 LXC。 Docker 提供了一种更简单的方式来将应用程序包装到包含它需要执行的附件的容器中。这一行动是通过一组有针对性的工具和集成的应用程序编程接口引导技术来实现的，这些技术具有内核级结构，Linux 容器，控制组以及写时复制文件系统。充当容器文件系统 Docker 依赖于高级多层统一文件系统（AuFS）。 AuFS 能够明确地叠加单个或多个可用的文件系统。它使 Docker 能够利用容器基础所需的图像。例如，一个人可能会使用一个 Ubuntu 映像，这个映像反过来可能会为多个其他容器提供基础。在高级多层统一文件系统的帮助下，Docker 使用 Ubuntu 的单个副本。这极大地节省了存储空间，并且随着集装箱的快速启动而减少了内存的使用。 AuFS 还有一个优势，它可以创建图像版本。每个最新版本都被标记为差异，这是识别两个文件之间差异的文件归类收益。这允许例如图像文件的体积减小。 AuFS 的另一个效率是，可以跟踪图像版本上的每个修改，就像软件开发代码版本控制系统[4]一样。

CoreOS 是 Linux 的一个相对较新的发行版，它的设计目的是提供操作栈的软件系统所需的特性。该技术提供了一个简化的 Linux 内核，以最大限度地降低开销。此外，CoreOS 使用工具支持集群，以确保冗余以及保护系统免遭故障的方法

最近，CoreOS 引入了一个新的产品 - 火箭容器运行时（rkt）[14]，它运行应用程序容器规范。这项技术的主要目的是为容器建立一个指定的模型。这种方法也支持亚马逊机器镜像。火箭容器运行时是 Docker 的一个选项，其中包括高级安全性以及其他必要的需求

3. 和微主机环境的比较

每种基于容器的技术都有其自己的特点。 本节介绍当前领先的基于容器的技术的一些关键特

性，这些特性可能会影响性能。 我们注意到，尽管 CoreOs Rocket 在前一节中已经确定，但由于 CoreOS 尚未发布其产品的正式版本，因此未对 Rocket 进行性能评估。

3.1 Docker

Docker 利用多个 Linux 内核功能来以独立的方式运行容器[9]。

• 命名空间：Docker 使用名称空间来部署容器。 Docker 使用几种类型的命名空间来执行创建隔离容器的任务[4]：

- Docker 使用 pid 作为容器的基础，确保容器中的所有进程不被允许影响其他容器中的进程;

- 它使用网络来管理网络接口，或者更准确地说，它提供了有关网络的系统资源的隔离;

- ipc 用于为特定的进程间通信（IPC）资源提供隔离，即 System V IPC 对象和 POSIX 消息队列。这意味着每个 IPC 命名空间都有自己的进程间通信资源。

- 为了让进程拥有自己的文件系统及其挂载点的视图，Docker 使用 mnt 命名空间。

- 通过 uts 执行内核和版本标识符的隔离。

• 控制组：Docker 执行 cgroups，以便现有的容器可以共享可用的硬件资源，并且在给定的时间可能会限制使用这些资源。

• 联合文件系统是一个文件系统，它通过建立用于为容器提供构建块的层来运行。

• 容器格式被认为是一个整合所有前面提到的机制的包装。

3.2 LXC

Linux Container 是一种基于容器的虚拟化技术，通过使用通用且灵活的 API 和相关实现，可轻松构建轻量级 Linux 容器[21]。另一方面，Docker 是基于容器的以应用为中心的技术。他们有一些共同的特点，但有很多不同之处。首先，LXC 是一种操作系统级虚拟化技术，用于在单个 LXC 主机上执行多个独立的 Linux 容器。它不使用虚拟机，而是允许利用具有自己的 CPU，内存，阻塞 I / O，网络以及资源控制机制的虚拟环境。这是通过 LXC 主机上 Linux 内核中的命名空间和 cgroups 功能提供的。它类似于 chroot，但提供更多的隔离。 LXC 支持各种虚拟网络类型和设备。其次，Docker 利用固定层来重新使用结构良好的设计，但这可能会以复杂性和吞吐量为代价。每个容器限制一个应用程序会降低使用可能性。使用 LXC，可以创建单个和/或多个应用程序。此外，LXC 允许创建多个系统容器，这可能只是一个可以通过使用 btrfs 文件运行的子卷的克隆。 LXC 的这一特性可以解决文件系统级别的复杂问题。第三，LXC 提供了广泛的设施和权限设计和执行容器。最后，LXC 允许创建无特权的容器，以确保非 root 用户可以构建容器。 Docker 还不支持这个功能。

3.3 CoreOS rocket

Rocket [14]是 CoreOS 推出的基于容器的技术，它提供了 Docker 的替代方案。Rocket 和 Docker

都以虚拟容器的形式引入了应用程序部署的自动化，虚拟容器可以根据服务器的特性独立执行。然而，尽管 Docker 已经发展成为一个复杂的环境，它支持各种各样的需求以及操作，但 Rocket 被构造为执行简单的功能，但是以安全的方式针对应用程序部署。火箭作为命令行工具执行应用程序容器，这些应用程序容器是图像设计的描述。 Rocket 专注于由 CoreOS 引入的应用程序容器规范，作为允许轻松迁移容器的描述组合。

正如 Polvi 认可的那样，与 Docker 相比，Rocket 可能更难以使用，因为 Docker 简化了通过描述性接口构建容器的整个过程。[14]认为 Rocket 应该保持基于命令行的环境，并且不太可能改变。

4. 评估方法和基准

有许多观点可以用来比较技术，特别是从性能的角度来看。为了从其开销角度评估基于容器的技术，有必要了解（测量）基于非虚拟化环境的开销。此处进行的分析集中于一系列性能标准：CPU，内存，网络带宽和延迟以及存储开销的性能。在所有基准测试中，多次重复实验重复 15 次，以评估各种结果的准确性和一致性。记录平均时间和标准偏差。

用于这些活动的云环境是澳大利亚全国 eResearch 协作工具和资源（NeCTAR）研究云（www.nectar.org.au）。 NeCTAR 为澳大利亚所有研究人员提供了一个云环境。它提供 8 个可用区域的 30,000 台服务器 - 通常位于州首府（墨尔本，堪培拉，霍巴特等）。 NeCTAR 项目由墨尔本大学领导，并由教育部资助。 NeCTAR 利用 OpenStack 中间件来实现云基础架构。

性能研究全部在 NeCTAR Research Cloud 实例上执行。以下实例配置用于执行实验：型号：处理器：AMD Opteron 62xx class @ 2.60 GHz;处理器 ID：AuthenticAMD 系列 21 型号 1 步进 2;内存：3955 MB;操作系统：Ubuntu 12.04（64 位）

表 1 给出了虚拟化技术及其版本。为了符合要求，所有 Docker 和 Flockport（LXC）容器都使用了 Ubuntu 64 位系统映像。而且，它们都在基于 Ubuntu 12.04 64 位的主机操作系统上运行。本节将演示所执行实验的结果。如前所述，所选择的基准测试工具评估 CPU，内存，网络带宽和延迟以及存储开销表现。

Table 1
The virtualization technologies and their versions.

Virtualization technologies	Version
Docker	1.4.0
Flockport (LXC)	1.1.2

Table 2
pbzip2 compressor results.

Platforms	Wall clock (s)
Native	13.7
Docker	14.8
Flockport (LXC)	14.9

Table 3
Multi-core efficiency results from Y-cruncher.

Platform	Multi-core efficiency
Native	99.2%
Docker	99.3%
Flockport	99.4%

4.1 CPU 性能

评估 CPU 性能的第一种情况是基于使用压缩机。压缩是经常使用的云环境处理模块。PBZIP2 [22]是 bzip2 块分类数据压缩工具的并行实现。它利用 p 个螺纹并可以达到几乎线性的加速度。使用 pbzip2-1.1.12 版本来压缩文件。一个输入文件（enwik8）[23]，它是 100 MB 的转储数据，并经常用于压缩测试目的。为了专心于压缩，使用了 900 kB 的 BWT 块大小和 900 kB 的文件块大小。

表 2 列出了 pbzip2 压缩器的性能。从 CPU 评估的角度来看，Docker 的性能略好于 LXC。在

Flockport 的情况下，平均经过时间为 14.9 s，标准偏差为±0.03 s，而 Docker 的平均时间为 14.8 s，标准偏差为±0.01 s。但是，应该考虑输入文件的大小。如果尺寸增加，结果的差异可能很大。

第二个 CPU 基准测试工具是 Y-cruncher [24]，它用于计算 Pi。它通常作为 CPU 的压力测试工具来执行，并且经常用作多核系统中运行的多线程工具的测试。除了计算 Pi 的值外，Y-cruncher 还可以计算一系列其他常数。 Y-cruncher 执行各种结果，如多核效率，计算时间和总执行时间。总时间用于确认输出，它包括总计算时间加上要求利用和执行结果的时间。

Y-cruncher 基准测试工具的性能结果如图 2 所示。就计算时间而言，Docker 的性能与本机（非虚拟化）系统类似，而 Flockport 的平均耗时约为 2 秒。表 3 列出了这些系统的多核效率结果。该评估描述了如何在计算 Pi 时有效利用 CPU。这种模式还表明，Docker 的性能比 Flockport 稍好。

探索的下一个基准测试工具是标准 HPC 基准测试工具：Linpack。这个工具有两个选项。第一个是英特尔的优化版本[25]，而第二个[26]允许在所有机器上运行，而不是在英特尔机器上运行。Linpack 找到了线性对应系统的解决方案，该系统采用了一种方法，该方法通过部分旋转执行数值分析的'较低'分解。大量的计算操作包括将标量处理与双精度浮点格式的向量相乘，以及将结果添加到不同向量的处理。基准测试工具建立在线性代数函数的基础上，并根据所选计算机体系结构进行修改。主 Linpack 函数利用大小为 N 的随机矩阵 M 和确定为 M * X = V 的向量 V.Linpack 基准工具执行两个步骤：M 的“Lower Upper”分解，然后是“Lower Upper” '分解应用于解决线性问题 M * X = V. Linpack 的结果通常以每秒 MegaFLOPS-浮点运算提供。执行此工具并增加 N 的值，从实验中可以看出，CPU 使用情况的变化和各个阶段的行为可以被识别：上升区，本地存储器或处理器中未发生任何挑战;在处理器功能方面出现问题的平坦区域和本地高速缓存存储器中发生挑战的衰退区域。

Linpack 在微托管环境中运行的结果可以在图 3 中看到。这些结果是通过应用 N = 1000 的特定场景获得的。如所见，即使它们之间的差距相对较小，Flockport 的性能在执行 Linpack 基准测试中比 Docker 稍好。

用于评估 CPU 性能的最终工具是 Geekbench [27]。此工具对于测试浮点单元的性能以及内存系统的吞吐量非常有用。本申请评估了上述吞吐量特征的上限。与 Ycruncher 相比，Geekbench 支持评估单核以及多核架构。它可以执行各种工作负载，生成多个索引，如整型性能，浮点性能和内存性能。此外，可以生成完整系统的索引

从图中可以看出，如图 4 和图 5 所示，结果的单核或多核测试没有显着差异。然而，关于内存性能，Flockport 分别在单核和多核测试中产生的性能分别比 Docker 性能高出约 100 分和 200 分。

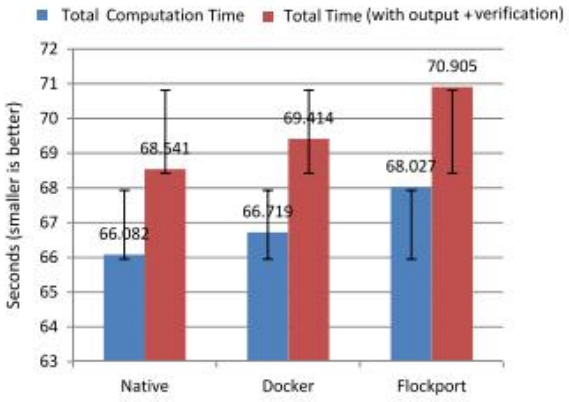

Fig. 2. Performance results of the Y-cruncher benchmark.

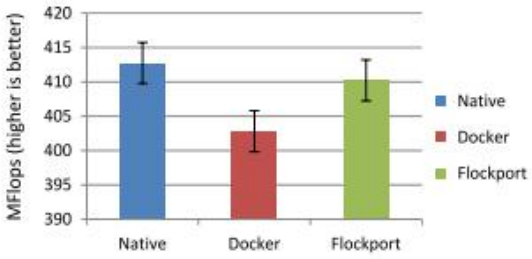

Fig. 3. Linpack results (where $N = 1000$).

4.2 磁盘 I/O 性能

性能的一个关键方面是对磁盘 I / O 性能的评估，特别是作为非短暂存储的卷的评估被附加到实例。这些卷是在与关联实例相同的可用区中创建的。用于评估微托管环境的第一个应用基准测试工具是 Bonnie ++ [28]。 Bonnie ++是一款描述磁盘吞吐量的开源应用程序。 Bonnie ++必须针对不同的情况进行配置，以便使用通用的测试文件。一个 4 Gb 数据集用于此目的。图 6 分别给出了基于顺序写入和读取的块输出以及块输入的 Bonnie ++结果。所有三个系统在顺序写入方面产生的结果几乎相同。然而，与 Docker 有关顺序读取文件相比，Flockport 显示出稍好的性能。通过 Bonnie ++基准测试应用程序，可以在读取，写入和冲洗过程在文件中进行评估时评估速度。表 4 显示了每个系统的随机写入速度以及随机寻道。

如图 7 所示，在随机写入速度结果之间有一个相对的对齐，在这个结果中，一个文件被读取，然后被写入并刷新到磁盘。然而，系统的布置在它们的随机查找评估中是不同的，其显示了块的数量 Bonnie ++可以每秒寻找。在这种情况下，Flockport 比 Docker 的效果好 100％，比原生平台好将近 6％。

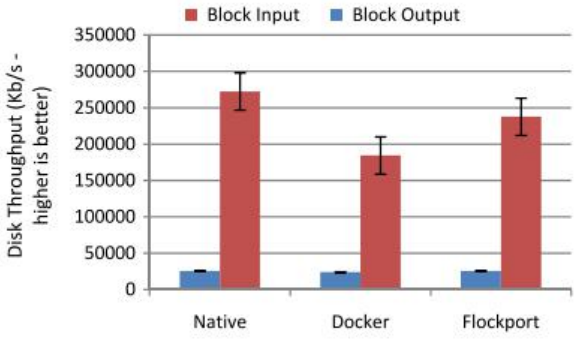

Fig. 6. Bonnie++ results for sequential write and sequential read.

Table 4
Random write speed and random seeks.

Platform	Random write speed (kb/s)		Random seeks	
Native	24829	%	3741	%
Docker	22494	−9.4%	389.2	−89.6%
Flockport	24524	−1.2%	3961	+5.9

为了进一步评估磁盘 I / O 吞吐量，使用了 Sysbench 基准测试工具[29]。该工具包含许多模块作为设计的基础。它也可以用作测量操作系统特性的跨平台和多线程测量软件。 Sysbench 的主要概念是在不部署数据库系统的情况下快速获得关于系统吞吐量的表示。该套件的实际版本允许评估系统特性，如文件 I / O 吞吐量，内存分配和传输速度调度程序吞吐量。在执行此方案时，该工具用于评估要读取的平台性能以及从指定文件写入的性能。为了评估文件 IO 性能，应该创建一个测试文件，该文件应该大于可用 RAM 的大小。这里使用的测试文件的大小是 35 GB，特别是每个 270 Mb 的 128 个文件。

表 5 中给出了这种测量的结果。Docker 和 Flockport 事件执行的总时间为 27.488 秒，标准偏差分别为±0.0003 秒和 27.491 秒，标准差分别为±0.0001 秒。没有显着差异。

Sysbench 实现的结果反映了通过 Bonnie ++获得的结果。本地以及 Flockport 显示大致相同的结果，没有任何明显的差异。然而，在某些运行中，Docker 比两个平台都表现更好。

为了彻底，应该注意的是默认文件格式用于 Native 和 Flockport 的磁盘映像，而 Docker 应用先进的多层统一文件系统，以确保分层以及映像版本化。因此，在这个实验中应该进行其他测试。

4.3 内存性能

内存 I / O 性能评估在本小节中介绍。 用于测试微托管环境的基准工具是 STREAM 软件[30]。STREAM 使用简单的向量内核过程来评估内存吞吐量。 这个工具生成了四个程序的结果，即 Copy，Scale，Add 和 Triad。 表 6 显示了这些过程及其计算方法。根据 STREAM 软件，评估的吞吐量与 CPU 高速缓存的大小之间存在着很大的关系。此外，还有规定每个流数组必须至少为 是可访问的缓存内存大小的四倍。 因此，软件中流数组的大小必须正确设置。 如图 7 所示，结果之间的差异并不显着，但 Docker 产生的结果略好于 Flockport，几乎与原生平台相同。

Table 5
Sysbench results.

Platform	Read (Gb)	Written (Mb)	Total transferred (Gb)	Throughput (Mb/s)	Elapsed time (s)
Native	1.22	834.17	2.04	20.85	27.568
Docker	1.21	829.38	2.02	20.73	27.488
Flockport	1.17	796.88	1.95	19.92	27.491

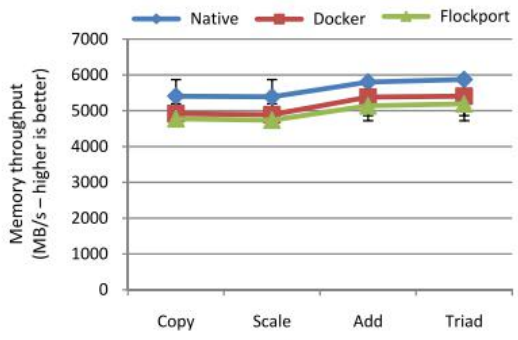

Fig. 7. STREAM results.

Table 6
Stream procedures.

Procedure	Kernel
Copy	$x[i] = y[i]$
Scale	$x[i] = q * y[i]$
Add	$x[i] = y[i] + z[i]$
Triad	$x[i] = y[i] + q * z[i]$

4.4 网络 I/O 性能

本小节描述了基于容器的环境在网络 I / O 方面的性能。任何在同一主机上运行的容器，更确切地说在同一个主桥上，都可以通过 IP 相互联系。网络地址转换（NAT）网络原则用于所有集装箱运输。这使得所有容器都可以与外界联系，包括不同主机上的其他 Docker 容器，但它不允许外部网络与容器进行通信。通过将容器端口映射到主机网络接口上的端口，可以避免此规定。 Linux 容器具有相同的网络。测试用例在位于两个不同主机上的两个 Docker 容器上进行。同样的条件也适用于 Flockport 容器。在这两种情况下，一个充当服务器，另一个充当客户端。对于本地条件，使用两个相同的 NeCTAR Research Cloud 实例

Netperf 基准测试工具[31]被用来测量网络 I / O。该工具有许多预定的子测试来评估服务器和客户端之间的网络吞吐量。它支持使用 TCP 或 UDP 协议在单一方向上执行数据传输。在 netperf 客户端和 netserver 之间建立连接所花费的时间不包含在这些评估中。评估结果显示了到达数据包的吞吐量如表 7 所示

表 7 显示了 Docker 和 Flockport 以及 TCP_STREAM 和 UDP_STREAM 的输出。对于 TCP_STREAM 测试用例，Docker 比 Flockport 执行速度近 200 Mbps，而对于 UDP_STREAM 测试用例，Docker 再次提供大约 150 Mbps 的更好性能。此外，如上所述，Netperf 基准测试工具包含评估案例，包括请求和响应，这些评估案例测量 TCP 和 UDP 事务的数量。在这些情况下会发生以下建立连接：netperf 客户端将请求分派给 netserver，并且 netperf 服务器将响应分派给 netperf 客户端。

如表 8 所示，Flockport 再次产生 TCP_RR 和 UDP_RR 测试用例的最差结果。

为了全面了解网络 I / O 性能，我们使用了另一个工具来测试网络吞吐量：Iperf 套件[32]。 Iperf 提供了一个完整的套件，用于评估使用 TCP 或 UDP 协议的连接性能。表 9 显示了 TCP 和 UDP 流量的 Iperf 测试结果。

从表 9 可以看出，与以前的 I / O 性能测试案例相比，Docker 速度较慢。可能的原因是 TCP 窗口大小和 UDP 缓冲区大小。也就是说，在没有验证的情况下，在给定连接的时间窗口中可以缓冲的数据量是不同的。数据的大小可以在 2 到 65,535 字节的范围内，但默认情况下它们在 Iperf 中很小（60.0 kB），并且未被调整。

5. 结论和未来的工作

基于容器的技术正在挑战基于虚拟机管理程序的方法作为云的基础。现代的基于容器的方法被认为是轻量级的。在本文中，对领先的微托管虚拟化方法进行了全面的性能评估，特别关注 Docker 和 Flockport 及其与本地平台的比较。如前所述，由于系统在撰写本文时无法使用，因此未考虑 CoreOS。

关于所进行的实验的结果，可以看到许多常见的模式。如图所示，Docker 或 Flockport 在内存利用率或 CPU 上几乎没有任何开销，而 I / O 和操作系统交互则会产生一些开销。这些情况下的开

销会通过每个输入输出操作的附加周期出现。因此，与输入输出要求较低的应用相比，输入输出操作增加的应用具有更多缺点。结果，这些开销加剧了输入输出延迟。实用程序任务所需的 CPU 周期也会导致性能下降。

Docker 还包括其他一些功能，如网络地址转换，这有助于减少 Docker 容器利用率的一些困难。但是，这些功能直接影响对吞吐量质量的投入。因此，在某些情况下，与 Flockport 相比，不使用额外功能的 Docker 容器不会更快。文件系统或磁盘密集型软件必须通过应用卷来使用高级多层统一文件系统。网络地址转换的影响可以通过使用 nethost 来消除。但是，这否定了网络名称空间的优点。最终，Kubernetes [33]建议的每个 Docker 容器的一个 IP 地址的设计可以支持质量保证和吞吐量。

Table 8
Netperf TCP_RR and UDP_RR results.

Platform	TCP_RR (Transfer rate per second)	UDP_RR (Transfer rate per second)
Docker	44363.03	45093.28
Flockport	39321.02	40625.07
Native	48451.11	49221.17

Table 9
Iperf results.

Platform		TCP	UDP
Docker	Interval	0.0–10.0 s	0.0–10.0 s
	Transfer	966 MB	11.9 MB
	Bandwidth	810 Mb/s	10.0 Mb/s
Flockport	Interval	0.0–10.0 s	0.0–10.0 s
	Transfer	1.34 GB	11.9 MB
	Bandwidth	1.15 Gb/s	10.0 Mb/s
Native	Interval	0.0–10.0 s	0.0–10.0 s
	Transfer	1.64 GB	1.19 MB
	Bandwidth	1.41 Gb/s	1.00 Mb/s

这里产生的结果可能会提供一些有关如何设计云环境架构的方向。因此，目前的考虑是 IaaS 更多地是利用虚拟机和 PaaS 来开发容器。如果 IaaS 被设计为使用容器，他们可以提供更好的吞吐量以及更简单的部署机会。此外，容器可以减少 IaaS 和“裸机”系统之间的差异，因为它们支持管理并提供与本机系统几乎相同的性能。这里的另一个问题是在虚拟机中启动容器，因此它们可能会与虚拟机的吞吐量开销相冲突，而与在本地主机上立即启动容器相比没有优势。这种务实的考虑应该考虑到用于构建和管理 IaaS 和 PaaS 系统的性能要求较高的社区的技术选择。多租户是云中更为重要的问题，容器或微服务应用程序通常由多个服务组成，这些服务在共享相同资源的不同容器中运行。对共享资源的影响和对性能的影响将在未来的工作中进行

致谢

作者想感谢 NeCTAR 研究云（www.nectar.org.au）提供用于执行这些调查的资源。

参考文献

[1] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. Shin, Performance Evaluation of Virtualization Technologies for Server Consolidation, Enterprise Systems and Software Laboratory HP Laboratories Palo Alto HPL-2007-59, 2007

[2] N. Regola, J. Ducom, Recommendations for virtualization technologies in high performance computing, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CloudCom, 30 2010-dec. 3 2010, pp. 409–416.

[3] N. Slater, Using Containers to Build a Microservices Architecture, viewed 1 April 2015, URL https://medium.com/aws-activate-startup-blog/usingcontainers-to-build-a-microservices-architecture.

[4] Docker – Build, Ship, and Run Any App, Anywhere, viewed 1 April 2015, URL http://www.docker.com.

[5] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An Updated Performance Comparison of Virtual Machines and Linux Containers, viewed 1 April 2015, URL http://www.research.ibm.com/.

[6] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment, Linux J. 2014 (239) (2014).

[7] Flockport, viewed 1 April 2015, URL http://www.flockport.com.

[8] P. Rubens, Docker Not the Only Container Option in 2015, IT Business Edge, 2015.

[9] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, C. De Rose, Performance evaluation of container-based virtualization for high performance computing environments, in: 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, (PDP), IEEE, 2013.

[10] S. Soltesz, H. Potzl, M. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, SIGOPS Oper. Syst. Rev. 41 (3) (2007) 275–287.

[11] E.W. Biederman, Multiple instances of the global linux namespaces, in: Proceedings of the Linux, 2006.

[12] P. Menage, Control groups definition, implementation details, examples and api, viewed 1 April 2015, URL http://www.kernel.org/doc/Documentation/ cgroups/cgroups.txt.

[13] L. Marsden, The Microservice Revolution: Containerized Applications, Data and All, viewed 1 April 2015, URL http://www.infoq.com/articles/ microservices-revolution.

[14] A. Polvi, CoreOS is building a container runtime, Rocket, viewed 1 April 2015, URL http://coreos.com/blog/rocket.

[15] J. Hwang, S. Zeng, T. Wood, A component-based performance comparison of four hypervisors, in: 2013 IFIP/IEEE International Symposium on Integrated Network Management, (IM 2013), IEEE, 2013.

[16] E. Abdellatief, N. Abdelbaki, Performance evaluation and comparison of the top market virtualization

hypervisors, in: 2013 8th International Conference on Computer Engineering & Systems, (ICCES), IEEE, 2013.

[17] S. Varrette, M. Guzek, V. Plugaru, V. Besseron, P. Bouvry, HPC performance and energy-efficiency of Xen, KVM and VMware hypervisors, in: 25th International Symposium on Computer Architecture and High Performance Computing, (SBAC-PAD), IEEE, 2013.

[18] R. Dua, A.R. Raja, D. Kakadia, Virtualization vs containerization to support PaaS, in: 2014 IEEE International Conference on Cloud Engineering, (IC2E), IEEE, 2014.

[19] Z. Estrada, Z. Stephens, C. Pham, Z. Kalbarczyk, R. Iyer, A performance evaluation of sequence alignment software in virtualized environments, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (CCGrid), IEEE, 2014.

[20] Heroku, viewed 1 April 2015, URL https://heroku.com.

[21] Linux Containers, viewed 1 April 2015, URL http://linuxcontainers.org.

[22] PBZIP2, viewed 1 April 2015, URL http://www.compression.ca/pbzip2/.

[23] enwik8, viewed 1 April 2015, URL http://mattmahoney.net/dc/textdata.

[24] Y-cruncher – A Multi-Threaded Pi-Program, viewed 1 April 2015, URL http://www.numberworld.org/Y-cruncher.

[25] Intel Math Kernel Library – LINPACK Download, viewed 1 April 2015, URL https://software.intel.com/en-us/articles/intel-math-kernel-librarylinpackdownload.

[26] LINPACK_BENCH – The LINPACK Benchmark, viewed 1 April 2015, URL http://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html.

[27] Geekbench 3 – Cross-Platform Processor Benchmark, viewed 1 April 2015, URL http://www.primatelabs.com/geekbench.

[28] Bonnie++, viewed 1 April 2015, URL http://www.coker.com.au/bonnie++.

[29] Sysbench in Launchpad - SysBench: a system performance benchmark, viewed 1 April 2015, URL https://launchpad.net/sysbench.

[30] J. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, a continually updated technical report (1991-2007), 2015, viewed 1 April URL http://www.cs.virginia.edu/stream.

[31] The Netperf Homepage, viewed 1 April 2015, URL http://www.netperf.org.

[32] Iperf, viewed 1 April 2015, URL http://www.iperf.fr.

[33] Kubernetes, viewed 1 April 2015, URL http://kubernetes.io.