基于容器技术的云计算资源自适应管理方法

树 岸 彭 鑫 赵文耘
(复旦大学软件学院 上海 201203) (上海市数据科学重点实验室(复旦大学) 上海 201203)

摘 要 云计算的发展使得越来越多的软件应用选择云平台作为部署平台。为了应对动态变化的工作负载、应用场景和服务质量目标,应用提供商希望能以一种可伸缩的方式对云计算资源进行动态调整。基于虚拟机的资源管理较为重载,难以实现细粒度的资源动态调整与混合云中跨平台的服务快速迁移。容器技术在一定程度上弥补了虚拟机的不足,然而传统的资源管理方法在诸多方面并不十分适用于容器技术。针对这一问题,提出了基于容器技术的云计算资源自适应管理方法,设计了更适用于容器的资源架构方案与资源之间的调度方式。与传统的线性建模方法不同,所提方法使用非线性函数对云计算资源进行更加精确的建模,同时用遗传算法进行参数调优,使得自适应调整响应更快、总体性能更好。所提方法还针对不同容器多维度的异构性,合理分配容器部署位置,提高物理资源利用率。此外,所提方法结合了容器技术多方面的底层特性,在分配负载等方面进行适应性调整。最后通过实验分析初步确认了所提方法的有效性。

关键词 云计算,Linux容器,自适应
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.07.023

Self-adaptive Approach for Container-based Cloud Resources Management

SHU An PENG Xin ZHAO Wen-yun
(School of Software, Fudan University, Shanghai 201203, China)
(Shanghai Key Laboratory of Data Science(Fudan University), Shanghai 201203, China)

Abstract Under the support of cloud computing, more and more applications choose cloud platform as their deployment platform. In order to respond to changing workloads, application scenarios, and user target, service providers need to dynamically adjust the existing resources in a scalable way. However, cloud platform resource management approach via virtual machine has many drawbacks. Virtual machines are heavyweight, it is not suitable for fine-grained and flexible allocation of resources. Meanwhile in the hybrid cloud background, the virtual machines can not migrate quickly. Container technology, to some extent, can make up for the lack of a virtual machine. However, traditional resource management method based on the virtual machine is not very suitable for container technology. To solve the above problems, this approach designed resources infrastructure solution and scheduling way which are more suitable to container. This approach models cloud resources precisely via nonlinear function, and could tune parameters using genetic algorithms, in order to optimize the performance of adjustment. This approach also computed rational allocation of container deployment location to improve physical resource utilization. In addition to that, this approach combines many bottom characteristics of container to adjust the procedure of load-balancing. We also conducted experiments to verify the effectiveness of the method.

Keywords Cloud computing, Linux container, Self-adaptive

1 引言

随着Web服务和互联网应用的不断发展和壮大,软件应用提供商选择云平台作为部署平台,并以按需使用的方式进行付费[1]。在竞争日趋激烈的市场环境下,提高用户的总体满意度,获取最大的商业价值,是软件应用提供商面临的关键问题。

软件应用提供商与用户之间往往会签订一个服务等级协议(Service-Level Agreement, SLA),该协议对服务类型、服务质量和客户付款等进行了定义[2-3]。对于软件应用提供商而言,服务等级协议反映了使用方对于服务质量(Quality of Service, QoS)的需求[4]。

为了达到最大化收益的目的,软件应用提供商需要对已有资源进行动态调整。常见的方法是通过控制论等手段,制

到稿日期:2016-02-24 返修日期:2016-05-22 本文受国家高技术研究发展计划(863)(2015AA01A203)资助。
树 岸(1991—),男,硕士生,主要研究方向为软件工程,E-mail:shuanyouxia@126.com;彭 鑫(1979—),男,教授,博士生导师,主要研究方向为软件维护、自适应软件、软件产品线;赵文耘(1964—),男,教授,博士生导师,主要研究方向为软件工程、电子商务。

定一些自适应策略,进行动态的自动调整。这种解决方案是基于将服务部署在云平台的虚拟机(Virtual Machine,VM)这一模式的,通过对系统当前负载进行监控,动态地对整体的虚拟资源进行伸缩性调整,以达到合理利用资源、节约成本、提高收益的目的。

然而传统的以虚拟机为载体的云平台虚拟资源管理方式存在着许多弊端。首先虚拟机对物理资源的需求较高,单台物理机(Physical Machine,PM)可部署的虚拟机资源也极为有限,使用这种方法较为浪费;其次,虚拟机较为重载,不适合细粒度灵活地分配资源;最后,某些场景下 QoS 的降低是由于用户所处网络环境变化所引起的,针对某些特定应用,为了保证用户服务不中断,合理的措施应为服务资源的网络迁移,虚拟机由于启动速度、镜像传输速度较慢,无法满足该需求[5]。

较为先进的容器技术由于具有占据物理资源少、启动速度快、并发性高、可在多种环境下运行等优势,在一定程度上弥补了虚拟机的不足,越来越多地被云计算平台所使用。然而由于容器和虚拟机的底层实现方式与调度方式具有多方面的区别,传统的基于虚拟机的资源管理方法对容器并不十分适用。

针对以上问题,本文提出了一种基于 Linux 容器技术的云计算资源自适应管理方法。所提方法对资源的单位进行了重新定义,并且设计了更适用于容器的资源架构方案与资源之间的调度方式。此外,所提方法还设计了遵循自适应计算的监控、分析、规划和执行控制环路的自适应管理框架,对云平台的资源伸缩性、部署位置、负载分配等进行管理。所提方法还使用非线性函数对云计算资源进行更加精确的建模,同时采用遗传算法进行参数调优,使得自适应调整响应更快、总体性能更好。

基于上述方法,本文设计并实现了一套云计算资源自适应管理工具,并部署了一套基于 Linux 容器的云计算平台,进行相关模拟实验以验证本文方法的有效性。

2　背景知识

Linux 容器(Linux Container,LXC)是一种内核虚拟化技术,可提供轻量级的虚拟化,使得资源和进程得到隔离[6-7]。

Docker 是 Linux 容器的一个开源项目,已加入 Linux 基金会,遵循 Apache2.0 协议,已经形成围绕 Docker 的生态体系[8]。Docker 的主要目标是"Build,Ship and Run Any App,Anywhere",即通过对应用组件的封装(Packaging)、分发(Distribution)、部署(Deployment)、运行(Runtime)等生命周期的管理,达到应用级别的"一次封装,到处运行"。这里的应用组件既可以是一个 Web 应用,也可以是一套数据库服务,甚至是一个操作系统或编译器。

从底层技术方面,Docker 区别于传统的虚拟化技术,Docker 基于 LXC,采用 container 方式进行隔离,主要通过内核的 namespace 将进程、网络、文件系统等隔离开[9]。Docker 作为一种轻量级的虚拟化方式,在运行上与传统的虚拟机方式相比具有显著的优势。首先 Docker 容器启动和停止比传统虚拟机方式更快,从分钟级缩短到秒级;其次,Docker 占据系统资源较少,单台物理机可以支持上千个容器,而虚拟机仅支持几十个,从镜像文件角度,Docker 一个镜像通常占据 MB 级别的磁盘空间,虚拟机通常为 GB 级别;最后,Docker可以通过类似 Git 的操作来方便用户获取、分发和更新应用镜像,并且可以通过 Dockerfile 配置文件来支持灵活的自动化创建和部署机制。

此外,容器技术也使得软件应用部署与调度方式产生了变化。软件应用的性能的瓶颈可能存在于某个模块的 CPU、内存、硬盘或者 I/O 中的某一方面,并且不同模块的安全敏感性也不同。所以针对一般软件应用,较为高效的部署方式是将软件应用分模块部署在不同的计算资源中。然而运行虚拟机本身就耗费大量额外资源,物理资源大部分被用于虚拟化,按模块部署软件应用很可能得不偿失。与虚拟机不同,容器占用的额外资源微乎其微,因此在容器技术的支持下,软件应用就可以采用分模块部署的方式,根据不同模块的负载进行伸缩性调整,并且实现内部的负载均衡调节,提高服务整体性能。

3　相关工作

在利用容器技术构建云平台方面,文献[5]提出了一种轻量级的资源配置方法,并且构建了一个云平台。文献[14]则基于性能以及资源利用率,对容器的部署方案进行了更加精确的设计。本文的主要区别在于结合了效用函数等方法来考虑商业价值,对云计算平台的收益权衡进行自适应控制,使得软件应用提供商的利益最大化,并且将应用分为模块进行部署。

文献[6-7,17]对虚拟机和容器的性能进行了实验测试,并分析了两者在不同条件下的优势与劣势。

在对受控对象进行控制方面,文献[18]阐述了软件与控制工程两者的相互影响,建议将控制工程的方法融入到自适应软件系统的设计中;文献[19]采用精确的建模方法通过控制器对系统进行优化;文献[12]通过遗传算法对传统的 PID 控制器微分、比例、积分参数进行优化。本文优化了对受控对象的建模方法,使用非线性的较为复杂的模型进行建模,使得模型更加准确,同时使用遗传算法对模型参数进行调节。

文献[20]对云计算中数据访问控制的方法进行研究,在实现细粒度与可伸缩控制的同时,还考虑了数据的安全敏感性,所提方法对云计算中的资源实现了细粒度与可伸缩的管理,但并未着重考虑安全问题。

文献[2]介绍了面向 SLA 的云计算资源分配方法,阐述了资源分配的挑战、架构方案以及解决方案。文献[3]使用了基于 SLA 的效用函数来寻找云服务中最优的资源配置方案,使得服务提供商能够在复杂的服务环境下获取最大的商业利益。

4　云计算资源自适应管理框架

4.1　平台资源架构

4.1.1　资源单位定义

基于容器的资源架构设计中,资源的最小单元是一个应用模块容器(Application Module Container,AMC),该容器可

以运行在 PM 或者云平台的 VM 中，其中部署了一个应用的一个模块。

这种最小单元的划分模式与以单个虚拟机为载体的划分模式主要存在两方面的不同：1)虚拟机为载体，对底层环境依赖程度较高，若进行迁移，则只能运行在物理机上，而容器可以迁移至物理机或者云平台中，前提只是需要运行相关引擎；2)由于其重量级的方式，虚拟机内部署的应用往往是一个完整的应用。以一个 Web 服务为例，单个虚拟机资源通常部署了 Web 服务器(例如 Tomcat，Weblogic 等)、数据库(例如 Mysql，Oracle 等)以及相关的后台逻辑计算模块，但是如果把应用模块容器 AMC 作为最小的资源单元，在一个 AMC 中可以单独部署一个软件应用模块。

4.1.2 平台资源架构

平台架设在包含公有云和私有云的混合云中，云内部部署着 PM 和 VM 的集群，外部的用户访问前端页面、中心控制器(Central Controller，CC)，中心控制节点根据用户所访问的具体服务，将请求转发给相应的 Web 容器进行处理。对于云中的 PM 节点，同样需要部署物理机控制器(Physical Machine Controller，PMC)，以监控 PM 运行的状态，并对 CC 的操作指令进行响应。平台的资源架构如图 1 所示。

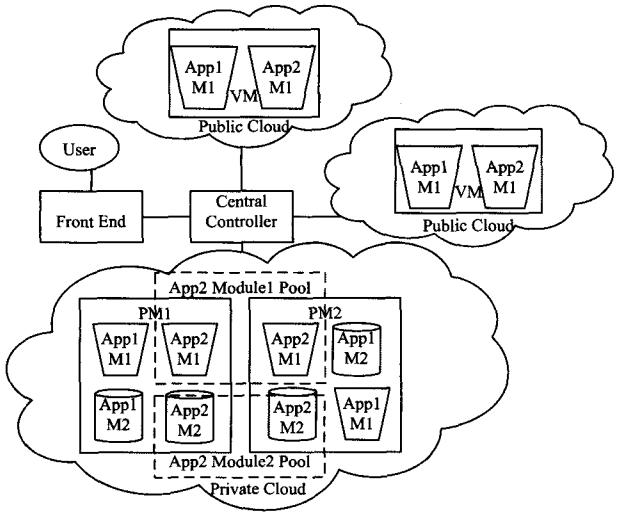

图 1 平台的资源架构

从图 1 中可以看出，公有云和私有云集群中都部署有若干 PM 或 VM，每台 PM 或 VM 中运行着若干 AMC，而 AMC 中部署着一个软件应用的其中一个模块，多个 AMC 中的模块可以构成一个完整的可为用户服务的应用。基于容器技术带来的优势，所提方法采取了将软件应用分模块部署在不同计算资源的方式，使得资源自适应管理更加高效。此外，在考虑安全性问题的情况下，可通过设立优先级的方式分配数据库模块的部署位置，对安全问题进行权衡[10]。

根据实际的负载情况，不同模块的调用关系不再是以往简单的一对一的调用关系(见图 1)，不同应用的相同模块，即部署了相关模块的多个 AMC，逻辑上组成了一个应用模块池(Application Module Pool，AMP)。

4.2 平台自适应控制架构

4.2.1 控制器模块构成

基于容器技术的云计算平台的自适应管理是基于多个层次的控制器，对相关控制对象进行监控与管理，并对各种情况进行响应。平台的中心控制器是最重要的控制器，其由以下几个模块构成。

外层负载均衡模块(Outer Load-balancing Module)，转发用户的请求，并且进行负载均衡的工作。

内层负载均衡模块(Inner Load-balancing Module)，对于同一应用的不同模块，CC 首先需要确定 AMC 之间具体的调用关系，然后根据不同模块的负载，确定它们之间的比例关系。

伸缩调控模块(Scale Module)，根据实际并发访问的请求数量以及系统当前的各项状态指标，对应用的 AMC 数量进行伸缩性调整。在资源有限的情况下，需要根据不同应用的收益情况，对资源进行合理分配，以实现利益最大化。

资源放置模块(Resource Placement Module)，对于某些特定应用，CC 需要监控用户网络状态，将相关 AMC 迁移至与用户所在网段更接近的云中。通过镜像新启动 AMC 时，CC 需要根据安全性约束，以及不同网段的负载情况，安排 AMC 是新建在何种云中。

以上简略地介绍了中心控制器的主要工作，下文会阐述具体控制方法以及相关算法。

在 AMC 与 PM 上也部署有控制器，其主要工作是监控 AMC 和 PM 的系统状态指标，将数据发送给 CC 进行处理，以及对 CC 发出的改变操作进行响应。

4.2.2 自适应框架

平台的自适应管理控制部分遵循自治计算的监控、分析、规划和执行(Monitor，Analyze，Plan and Execute，简称 MAPE)控制环路[11]。一个完整的 MAPE 流程为管理控制的一个自适应周期。自适应框架如图 2 所示。

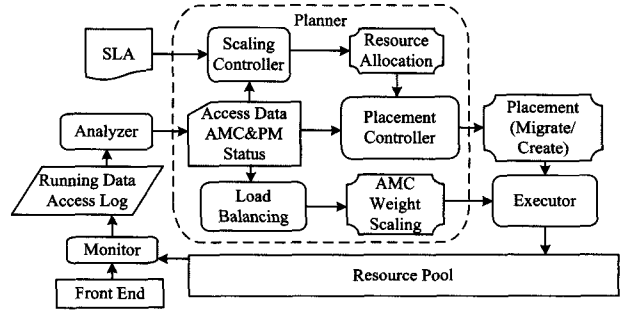

图 2 平台自适应框架设计

从自适应流程框架图中可以看出监视器(Monitor)负责收集 Web 服务的访问日志(Access Log)和 AMC 与 PM 的运行时数据(AMC & PM Running Data)。

分析器(Analyzer)处理监视器采集到的数据，以获得软件应用访问状态(App Access Status)和 AMC 与 PM 运行状态(AMC & PM Status)。

规划(Planner)部分由多个控制模块完成，伸缩调控模块利用分析器得出的数据，结合 SLA，参照不同的自适应策略计算出当前的调整方案。

执行(Executor)部分将执行伸缩调整方案、放置调整方案，具体操作即调用容器引擎相关接口，通过镜像库中的镜像新建容器实例或停止运行容器实例，执行负载均衡调整方案

即修改转发和调用的配置文件修改权值。

5　平台控制器设计

5.1　伸缩调控模块设计

上文简略介绍了伸缩调控模块的主要工作，结合系统当前状态，根据优化目标，对平台不同软件应用的各个功能模块的 AMC 数量进行伸缩性调整。伸缩性调控模块的设计主要包括以下几个具体的方面：计算资源的建模与参数调节、基于效用的资源配置方案调整。

5.1.1　计算资源的建模与参数调节

根据上文平台资源逻辑架构中的设计，平台上部署的一个软件应用由多个功能模块组成，由于大量并发访问的需要，每个功能模块都会有许多 AMC 实例组成一个应用模块池 AMP。

一个 AMP 的平均响应时间主要由当前对该模块的并发访问人数和 AMP 中的同类 AMC 数量决定（不考虑容器资源的异构性）。所提方法进行了大量模拟实验，每个 AMC 中部署相同的应用，在并发访问人数线性增加的情况下，计算 AMP 的平均响应时间。实验数据图如图 3 所示。

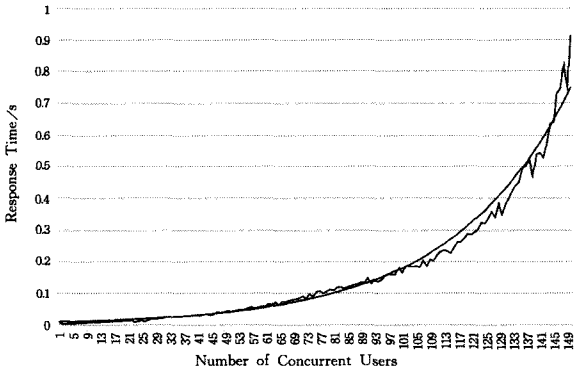

图 3　响应时间随并发用户变化的关系

从图 3 中可以看出，在 AMC 数量相同的情况下，响应时间（Response Time，RT）随并发访问人数提高的速度由慢变快。在数据量足够大时，其整体呈非线性关系，近似指数增长趋势。图 3 中的数据经过拟合，与函数 $y=1.029386^{x-160}$ 具有较高契合度。而在相同并发人数的情况下，随 AMC 数量的线性增加，响应时间降低速度由快变慢，也近似指数下降趋势。通过对大量的数据进行拟合分析，可得 RT 与并发访问人数、AMC 个数的近似数学式(1)，即受控对象的模型。

$$RT(t)=\alpha^{\beta U(t)-\gamma\cdot N(t)}+\xi(t) \tag{1}$$

其中，$RT(t)$ 为 t 时刻 AMP 的平均响应时间，$U(t)$ 为 t 时刻的用户数量，$N(t)$ 为 t 时刻 AMP 中 AMC 的数量，$\xi(t)$ 为 t 时刻系统的扰动。α 是指数函数的基数，β 是 $U(t)$ 的系数，γ 是 $N(t)$ 的系数。所提方法采用遗传算法对模型公式进行系数整定，主要在于遗传算法克服了单点出发搜索的缺点，在解空间进行高效启发式搜索，操作方便，速度较快[12]。

使用遗传算法对式(1)的参数进行优化整定，主要包括参数编码、种群初始化、适应度函数选定、选择操作、交叉操作、变异操作。

需要寻优的 3 个参数为 α,β,γ，前期先进行大量数据拟合，估算出 α,β,γ 的大致范围，选择合适的编码位数。染色体个体编码形式为：

$$a_{11}a_{12}\cdots a_{1i}a_{21}a_{22}\cdots a_{2j}a_{31}a_{32}\cdots a_{3k} \tag{2}$$

其中，i,j,k 分别代表 α,β,γ 3 个参数选择的二进制编码位数。本方法的每个参数选择 8 位无符号二进制编码来表示（$i=8,j=8,k=8$），每个基因长度为 24，$a_{xy}\in\{0,1\}$。

在种群初始化的过程中，本方法先使用拟合工具利用历史数据拟合出受控对象的近似模型公式，再根据公式中的各个参数大致估算出参数的取值范围，在此范围内均匀生成初始种群。

本方法主要根据受控对象模型计算出的 RT 与真实值之间的差距来评判染色体个体的优劣。本方法采用的适应度函数为：

$$F=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(RT_i-rt_i)^2}^{-1} \tag{3}$$

其中，n 为一个采样周期内的样本个数；$rt_1,rt_2,\cdots,rt_n$ 为样本的实际响应时间；$RT_1,RT_2,\cdots,RT_n$ 为当 3 个参数为当前染色体所代表的值时，通过式(1)计算出的理论响应时间，目标函数值越大，则当前染色体的适应度越高，该解越优。

本方法的选择操作采用比例选择算子，每个染色体个体被选择的概率与该染色体适应度成正比。染色体个体 i 被选择的概率为：

$$P_i=F_i/\sum_{i=1}^{M}F_i \tag{4}$$

其中，M 为种群的大小，F_i 为该个体的适应度函数。

所提方法的交叉操作采用线性组合的方法将两条染色体个体组合出多个新的个体。由于染色体分为 3 段，每段需要分别决定是否进行交叉及交叉的位置，因此使用随机函数生成 3 个[0,1]范围内的浮点数 a,b,c 来决定 3 段的交叉情况。变异操作也使用类似的随机数方法来决定染色体个体的某一位是否进行变异，即由 0 变为 1，或由 1 变为 0。

经过若干次迭代，得到 α,β,γ 的最优解，即完成了受控对象模型的参数调节工作，得到的模型公式能够较为准确地反映出当前系统的状态。

5.1.2　基于效用的资源配置方案调整

效用策略表明系统服务层属性的需要程度，可以通过效用函数来完成[13]。单个请求的收益由软件应用提供商和用户签订的 SLA 决定，服务的质量等级决定了收益情况。以 RT 作为服务质量的衡量标准，则常见的 SLA 存在着以下关系：

$$p=\begin{cases}p_{\max}, & 0<RT_{real}\leqslant RT_{good}\\ p_{\max}\cdot\dfrac{RT_{bad}-RT_{real}}{RT_{bad}-RT_{good}}, & RT_{good}<RT_{real}\leqslant RT_{bad}\\ 0, & RT_{real}>RT_{bad}\end{cases} \tag{5}$$

其中，$p_{\max}$ 是收益的最大值，RT_{real} 是实际响应时间，RT_{good} 是 SLA 中约定的满意响应时间，RT_{bad} 是 SLA 中约定的不满意响应时间，实际值在这两值之间时收益线性降低。

单个受控对象 AMP 的单位时间成本主要在于其占用物理资源的多少，本文将其所占资源简化为 AMC 的数量，原因在于，虽然不同 AMC 的异构性决定了它们占用不同物理资源的比例不一（如计算模块容器占据 CPU 时间片更多，而占

用硬盘很少),但是下文将介绍的资源放置模块能够合理分配 AMC 所在的 PM,使得不同类型的 AMC 能够交叉放置,物理空间最大化利用,这样不同类型的 AMC 占用的总体物理资源比例近似相等,因此可以将 AMP 所占资源简化为 AMC 的数量。

对于单个受控对象而言,在当前负载下增减 AMC 的数量均会对该受控对象的平均响应时间产生影响,而影响的具体值可通过模型式(1)估算得出。结合式(1)和式(5),当改变 AMP 数量时,收益的变化量为:

$$\Delta UX = U \cdot (p(RT(N+\Delta N)) - p(RT(N))) - C \cdot \Delta N \tag{6}$$

其中,U 为当前系统的访问人数,ΔN 是 AMC 的变化量,C 是单个 AMC 的成本,其余是根据式(1)和式(5)的计算值。

本文的平台设计中,一个完整的软件应用由多个模块组成,因此一个用户的完整服务请求的响应时间为:

$$RT_{total} = \sum_{i=1}^{N} RT_{Mi} + T_{trans} \tag{7}$$

其中,N 为软件应用的模块类型数,RT_{Mi} 为第 i 类模块的响应时间,如果本次请求没有调用 i 类模块,则其值为 0,T_{trans} 为网络传输的时间。

针对资源总量不受限制的情况,不同软件应用之间的资源占用不产生冲突,只考虑成本和收益,为使整体效用最大化,本文使用算法 1 对单个软件应用的各模块 AMC 数量进行伸缩性调整。算法 1 中 n 表示该软件应用内的功能模块类型数。

算法 1

1. $\Omega=\{N_1,N_2,\cdots,N_n\}$;
2. $\Phi=\{RT_1,RT_2,\cdots,RT_n\}$;
3. $\Theta=\{\Delta UX_1,\Delta UX_2,\cdots,\Delta UX_n\}$;
4. while(not both increase & decrease)
5. if($RT_{total}>RT_{good}$) then
6. calculate $\Theta(\Delta N=1)$;
7. $\max(\Theta)=\Delta UX_i$;
8. If($\max(\Theta)>0$)then
9. increase N_i;
10. update $RT_{total}^{-}=RT_i(N_i+\Delta N)-RT_i(N_i)$;
11. end if
12. else
13. Calculate $\Theta(\Delta N=-1)$;
14. $\max(\Theta)=\Delta UX_j$;
15. If($\max(\Theta)>0$)then
16. decrease N_j;
17. update $RT_{total}^{-}=RT_j(N_j+\Delta N)-RT_j(N_j)$;
18. end if
19. end if
20. end while
21. return Ω;

算法 1 中循环的中止条件是进行了个数增加和减少两种变化,该中止条件可避免死循环。通过算法 1 可以得到新的资源配置方案,使得单个软件应用的总体效用最高。但算法 1 仅适用于资源总数不受限的情况。

当资源总数受限时,不能保证云计算平台上所有的软件应用都在效用最大化状态,但是要使得资源往获利更高的方向发展,需使得整体效用最大化。整体效用为平台上所有软件应用效用之和:

$$UX_{total} = \sum_{i=1}^{m} UX_i \tag{8}$$

其中,m 为平台上软件应用的总数。若通过减少软件应用 1 的 AMC 数量,增加软件应用 2 的 AMC 数量,使得 $\Delta UX_1+\Delta UX_1>u$($u$ 为设定的阈值,$u>0$,因为资源流动也耗费一定成本),那么这次资源重新整合则为有效的。本文在资源受限的情况下使用算法 2 进行资源流动整合,m 为软件应用种类数量,ϕ 为每个软件应用的 $\Delta N=1$ 时的效用变化预测值,θ 为 $\Delta N=-1$ 时的值。循环的中止条件是理论上最高效的资源流动方式也不能满足 $\Delta UX_1+\Delta UX_1>u$。

算法 2

1. $\omega=\{\Omega_1,\Omega_2,\cdots,\Omega_m\}$;
2. $\phi=\{\Delta UXI_1,\Delta UXI_2,\cdots,\Delta UXI_m\}$;
3. $\theta=\{\Delta UXD_1,\Delta UXID_2,\cdots,\Delta UXID_m\}$
4. while(!no new suitable resource flow)
5. calculate ϕ,θ;
6. sort(decrease) ϕ,θ;
7. If($\max(\phi)+\max(\theta)>u$) then
8. resource flow success;
9. update w;
10. else
11. no new suitable resource flow;
12. end if
13. end while
14. return w;

5.2 资源放置模块设计

资源放置位置主要包含两方面:1)部署在私有云或不同的公有云上;2)部署在云中具体哪一台 PM 上。针对混合云的部署,决定因素主要是安全和效率。安全要求性较高的,比如软件应用的某些重要数据库模块,更适合部署在私有云上,但是具体约束规则由软件应用提供商决定。效率问题主要在于不同的访问用户来自不同的网段,例如局域网或外网,或者是不同运营商的网段,如果某一网段用户访问量较大,但资源却主要部署在另一网段,那么也会造成响应时间较高和 QoS 较低。

对于一个应用 App_i,它部署在网段 Net_j 的优先度为:

$$P_{ij} = \sum_{k=1}^{K} w_k \cdot \rho_k \tag{9}$$

其中,K 为影响因子的种类,ρ_k 为第 k 种影响因子的影响度,w_k 为它的权值,本文选择网段 Net_j 在 App_i 所有访问的比例、网段 Net_j 的平均传输时间作为影响因子,故优先度为:

$$P_{i,j} = w_{time} \cdot \rho_{time} + w_{ratio} \cdot \rho_{ratio} \tag{10}$$

部署的比例与优先度成正比,故应用 App_i 部署在网段 Net_j 的比例为(共 j 种不同网段):

$$R_{ij} = P_{ij} / \sum_{k=1}^{J} P_{ik} \tag{11}$$

其中所计算出的部署比例用于需要大量迁移 AMC 时的重新部署(实际比例与优化比例差别较大时),如果当前部署状态较好,且只需要决定新增的 AMC 的部署网段,则一般选择

R_{ij} 最大值的网段。

对于部署在云中的具体 PM 的位置，主要关注点在于物理资源的利用率。PM 的物理资源包括 CPU、内存、硬盘、带宽等方面，用一个元组进行描述：

$$\gamma=\langle r_1, r_2, \cdots, r_n \rangle \tag{12}$$

其中，元组字母分别代表上述 n 个资源因子，其值有两种表示方法：1）一台 PM 的资源总数 γ_{value}，如 CPU 核数、内存总数、硬盘容量、带宽；2）这些指标的使用百分比 γ_{radio}。对于一个 AMC，同样需要对这些指标进行描述，值的表示方法采用第一种，具体而言，CPU 和内存在新建容器时可以进行初始限定，CPU 使用权值系数（Docker 中，在系统空闲时可以超过该权值，但系统繁忙时不会分配低于权值的 CPU 时间），内存限制具体的最大值，硬盘和带宽可以使用 Linux 的相关工具，如 quota，cgroup 的 net_cls。

为了使得物理资源 PM 的利用率最高，需要遵循两个策略[14]：对于一台已知 γ_{value} 的 PM，优先将 γ_{value} 各个因子的比例与它相近的 AMC 部署在这台 PM 中，这样有利于让 PM 尽可能地避免出现某类资源使用耗尽而另一些资源过剩的情况，达到资源各个维度的均衡；对于一台已知 γ_{radio} 的 PM，不同的因子使用率不尽相同，对于使用率较低的物理资源，应该尽可能地将该类资源需求量大的 AMC 部署在这台 PM 中，反之亦然，可以避免出现资源瓶颈。

计算各项资源比例时首先应建立标准单位体系，如 CUP 单位 core，内存单位 GB，磁盘单位 TB，带宽单位 MB/s。当新增一台需要部署的容器时，且在整体 PM 各项资源使用率较平衡的情况下，优先使用第一种策略，有资源瓶颈趋势时，优先使用第二种策略。

5.3　负载均衡模块设计

平台需要在内外两层进行负载均衡，外层使用 Nginx 服务器作为用户访问的入口，将负载分发给平台的 Web 容器，内层对内部模块之间的调用进行平衡，即对容器模块池进行负载均衡，具体而言，模块之间调用时，不直接指定目标模块的 IP 地址与端口号，而是向负载均衡器请求一个地址，负载均衡器每次收到请求时，根据相关 AMP 中各个 AMC 的权值选择一个 AMC 的 IP 地址和端口号将之回复给请求调用的模块。

所提方法采用基于权重的轮询算法，将负载分配给相应的计算资源。由于资源的异构性[15]，不能简单地将任务平均分配，而是基于资源的性能动态调整负载。

本文采用了文献[16]中的方法，基于各个资源的平均响应时间与预设的期望之差，决定需要提高/降低负载的比例，使其性能与预设期望相仿，达到负载均衡的目的。

为了实现基于反馈控制的负载均衡控制器，首先需要确定一个合适的反馈量。由于系统的总体负载与系统的响应时间之间存在着近似的线性关系[36,40]，因此响应时间常常被用作反馈量来进行反馈控制。考虑到 Docker 容器的一些特性，如当内存使用率较高，且超过预先设定的最大分配值时，容器会出现崩溃的情况。由此可见，内存的使用率对容器正常运行尤其重要。故所提方法使用响应时间与内存使用率两个量做反馈调节，一方面能够保证服务的质量，从而保证收益，另一方面能避免出现虚拟资源崩溃的情况。

6　实验

为了验证设计方法的有效性，本文实现了一套基于 Docker 的小型云计算平台，以及基于上述方法的自适应资源管理系统，并且进行了仿真实验。

6.1　实验设置

实验在两台 DELL PowerEdge R210 服务器（4 * Intel Xeon X3430@2.40GHz CPU，4GB RAM，1TB Disk，10M/s Net Bandwidth）上安装了操作系统 Ubuntu 14.04，Docker 1.5.0，作为云计算平台的基础设施，一台模拟私有云，一台模拟公有云，分别在不同的网段。

编写了两个小型 Web 应用作为测试案例，应用分为前端页面部分和后端数据库部分，前端部署在 Tomcat 7.0.55上，后端部署在 MySQL 5.5.31 中。制作了 4 个不同的镜像，分别是部署完成的两个软件应用的 4 个模块，用于在平台上新建容器。Tomcat 容器的初始参数设置为 0.25 core CPU，256MB RAM，MySQL 容器的初始参数设置为 0.2 core CPU，256MB RAM。此外，在一台服务器上部署了 Nginx 1.1.19作为反向代理服务器，负责将用户访问按照一定比例分发给相应的虚拟资源。

实验过程中，使用压力测试工具 JMeter 2.11 来模拟并发的用户访问，时间持续 60 分钟，其中 App1 的并发用户数在 0－50 分钟从 0 线性增加到 200，50－60 分钟保持不变；调用数据库服务的前 55 分钟用户不超过 20 人，55－60 分钟迅速上升至 50 人；用户来自两个网段（本实验中用 IP 进行区分网段）的比例保持恒定。

App2 的并发访问数始终保持在 150；调用数据库服务的用户不超过 30 人；用户来源网段中，来自公有云网段的比例在 10－50 分钟从 90%降低到 50%。

SLA 中对服务收费情况进行以下约定：App1 费用最大值为 1 元/（人 * 分钟），App2 费用最大值为 0.8 元/（人 * 分钟），满意响应时间为 100ms，不满意响应时间为 300ms，单个容器成本为 6 元/分钟。

6.2　实验结果与分析

实验过程中 App1 的总体响应时间和总体利润随时间变化的图像如图 4 所示。

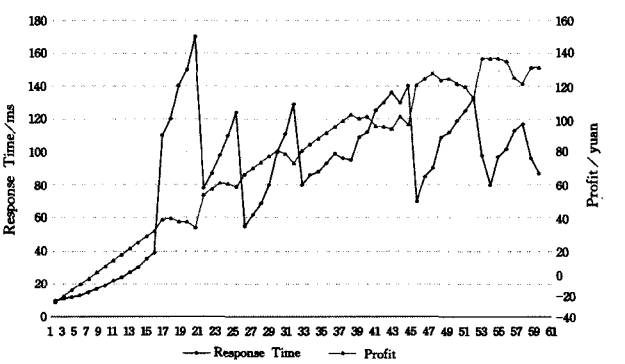

图 4　App1 响应时间、收益变化图

从图 4 中可以看出，总体响应时间在整个实验过程中一共发生了 7 次明显变化，分别为 $t=14,19,24,31,45,53,58$

时刻，结合系统日志，发现第一次变化是 App1 的资源总数减少了 1 个，后 6 次每次增加 1～2 个。对照利润变化不难看出，当系统性能下降、收入降低时，系统通过增加服务资源，提高性能，使整体利润提高，并且除第六次外，反应速度较快。在资源种类上，前 6 次增加的是 Tomcat 资源，第七次增加的为 MySQL 资源，结合实验设置，数据库服务访问在 55 分钟后开始变多，导致整体响应时间提高，此时增加 MySQL 资源最为有效，实验结果与控制需求相符。

在 57 分钟时，App2 的资源总数减少了 1，App1 的 MySQL 资源数增加了 1，结合日志可以发现，当时内存使用率超过了 70%，已超过阈值，系统认定为资源受限，无法再增加容器资源，通过资源流动的方式，使整体利润增加。

在 43 分钟时，系统触发了一次资源迁移，3 个 App2 的容器从公有云服务器迁移至私有云服务器，结合实验设置，是因为实际部署比例与优化比例差距过大，触发了资源迁移。

在 60 分钟后系统达到较为稳定状态时，私有云服务器的 CPU 使用率为 70.8%，内存使用率为 78.6%，公有云服务器的 CPU 使用率为 66.3%，内存使用率为 75.5%，物理资源利用较为平均，本文的资源放置控制模块效果良好。

通过 Docker Stats 工具查看容器运行状态与各个不同容器的实际响应时间，各个容器负载较为平均，负载均衡模块效果良好。

根据以上分析可以看出，所提方法在伸缩性、资源放置、负载均衡几个方面表现良好。同时在实验过程中也发现了一些问题：首先由于 Docker 的底层实现原因，容器的 CPU 的时间片分配不是十分精确，如 3.3 节提到的，Docker 中在系统空闲时可以超过该权值，但系统繁忙时不会分配低于权值的 CPU 时间，因此对于 CPU 需求较大而内存需求不大的应用，使用伸缩性控制的方法效果可能低于预期，本文的应用案例对内存需求较高；实验中后期出现调整不及时的情况，经过相关日志分析，可能是在系统负载较高时，受控对象的模型产生较大的变化，使得公式准确性下降，自适应滞后。

7 总结

本文针对传统的以虚拟机为载体的云平台虚拟资源管理方式存在着许多弊端，提出了一种基于 Linux 容器技术的云计算平台资源自适应管理方法。该方法采用按模块部署软件应用的方式，对不同类型的模块分配多种定制化的容器作为资源的载体，同时根据系统状态对资源进行合理分配、动态迁移，使服务质量提高的同时，软件应用提供商的收益也得到了增加。

在受控对象的建模与控制方面，本文提出了区别于传统的线性建模的方法，使用非线性函数进行更加精确的建模，同时用遗传算法进行参数调优，使得模型公式更加准确。本文还进行了案例研究以验证方法的有效性，效果符合预期。

本文的研究工作还存在一些不足，由于 Docker 的底层实现原因，容器的 CPU 的时间片分配不是十分精确，因此对于 CPU 需求较大的应用，使用伸缩性控制方法的效果可能低于预期；实验中后期出现调整不及时的情况，经过相关日志分析，可能是在系统负载较高时，受控对象的模型产生较大的变化，使得公式准确性下降，自适应滞后。

结束语 本文利用容器技术对云计算资源进行自适应管理，提高系统的整体效用。本文的后续工作会针对 Docker 的一些特性以及复杂模型的自适应控制，对方法进行优化，同时考虑非全局集中的控制方法，利用容器控制自身生命周期，将两种方法进行分析比较。

参 考 文 献

[1] DIKAIAKOS M D, KATSAROS D, MEHRA P, et al. Cloud computing: Distributed internet computing for IT and scientific research[J]. Internet Computing, IEEE, 2009, 13(5): 10-13.

[2] BUYYA R, GARG S K, CALHEIROS R N. SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions[C]// Proceedings of the 2011 International Conference on Cloud and Service Computing (CSC). IEEE, 2011: 1-10.

[3] RANALDO N, ZIMEO E. Capacity-Aware Utility Function for SLA Negotiation of Cloud Services[C]// Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC). IEEE, 2013: 292-296.

[4] LI J, WANG Y, ZHANG J. Research of web qos control model based on dynamic resource reallocation scheme[C]// Proceedings of the 2008 International Symposium on Information Science and Engineering (ISISE'08). IEEE, 2008: 103-106.

[5] HE S, GUO L, GUO Y, et al. Elastic application container: A lightweight approach for cloud resource provisioning[C]// Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications (aina). IEEE, 2012: 15-22.

[6] JOY A M. Performance comparison between Linux containers and virtual machines[C]// Proceedings of the 2015 International Conference on Advances in Computer Engineering and Applications (ICACEA). IEEE, 2015: 342-346.

[7] FELTER W, FERREIRA A, RAJAMONY R, et al. An updated performance comparison of virtual machines and linux containers [C]// Proceedings of the 2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS). IEEE, 2015: 171-172.

[8] BERNSTEIN D. Containers and cloud: From lxc to docker to kubernetes[J]. IEEE Cloud Computing, 2014(3): 81-84.

[9] 杨保华，戴王剑. Docker 技术入门与实战[M]. 机械工业出版社，2015.

[10] YU S, WANG C, REN K, et al. Achieving secure, scalable, and fine-grained data access control in cloud computing[C]// Proceedings of the 2010 IEEE Infocom. IEEE, 2010: 1-9.

[11] KEPHART J O, CHESS D M. The vision of autonomic computing[J]. Computer, 2003, 36(1): 41-50.

[12] CHEN Y, WU Q. Design and implementation of PID controller based on FPGA and genetic algorithm[C]// Proceedings of the 2011 International Conference on Electronics and Optoelectro-

nics (ICEOE). IEEE,2011:V4-308-V4-311.

[13] YEO C S,BUYYA R. Integrated risk analysis for a commercial computing service[C]// Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007). IEEE,2007:1-10.

[14] XU X,YU H,PEI X. A Novel Resource Scheduling Approach in Container Based Clouds[C]// Proceedings of the 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE). IEEE,2014:257-264.

[15] DEJUN J,PIERRE G,CHI C H. EC2 performance analysis for resource provisioning of service-oriented applications[C]// Proceedings of the 2010 International Conference on Service-oriented Computing. Springer-Verlag,2010:197-207.

[16] YU L,XIE Y,CHEN B H,et al. Towards Runtime Dynamic Provision of Virtual Resources using Feedforward and Feedback Control[J]. Journal of Computer Research and Development, 2015(4):889-897.

[17] ADUFU T,CHOI J,KIM Y. Is container-based technology a winner for high performance scientific applications? [C]// Proceedings of the 2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS). IEEE,2015:507-510.

[18] CAI K Y,WANG X Y. Towards a control-theoretical approach to software fault-tolerance[C]// Proceedings of the 2004 Fourth International Conference on Quality Software (QSIC 2004). IEEE,2004:198-205.

[19] LIN R,CHEN B,XIE Y,et al. Learning-Based Multi-controller Coordination for Self-Optimization[C]// Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (COMPSACW). IEEE,2012:164-169.

[20] YU S,WANG C,REN K,et al. Achieving secure,scalable,and fine-grained data access control in cloud computing[C]// Proceedings of the 2010 IEEE Infocom. IEEE,2010:1-9

(上接第 106 页)

(3)假设仲裁者 Trent 是不诚实的,那么他企图伪造签名也是不可能成功的。首先,U_i 所拥有的签名信息 k_{u_i} 对 Trent 来说是未知的;其次,U_i 传递的信息都是使用密钥 $K_{U_iU_{i+1}}$ 进行加密的,因此 Trent 是不能获得签名信息的,那么他仅凭随机猜测伪造的签名是不能够通过最后的验证的。即使 Trent 与第三方的攻击者 Eve 进行合谋,也不能得到 k_{u_i} 的信息,进而不能够伪造签名。

结束语　本文提出了一个量子多重代理盲签名协议,首先介绍了 Bell 态和 Bell 测量之间的关联特性,然后利用关联特性构造密钥之间的联系,使得原始签名人的密钥与多个代理签名人密钥异或的结果构成关联,进而满足了多个人代理一个人签名的要求,而且代理签名的人数可以根据实际的需求而定,增加了协议的灵活性和应用性。最后的安全性分析表明了本协议是一个安全可实现的量子多重代理盲签名协议。

参 考 文 献

[1] STALLINGS W. Cryptography and Network Security Principles and Practices[J]. Printice Hill Publishing Pp,2005,11(7):655-660.

[2] CAO Z J,LIU M L. Classification of signature-only signature models[J]. Science in China Series F: Information Sciences,2008,51(8):1083-1095.

[3] SHOR P W,PRESKILL J. Simple proof of security of the BB84 quantum key distribution protocol[J]. Physical Review Letters,2000,85(2):441.

[4] HUGHES R J,MORGAN G L,PETERSON C G. Quantum key distribution over a 48km optical fibre network[J]. Journal of Modern Optics,2000,47(2/3):533-547.

[5] BOSTROM K,FELBINGER T. Deterministic secure direct communication using entanglement[J]. Physical Review Letters,2002,89(18):187902.

[6] ZENG G,ZHANG W. Identity verification in quantum key distribution [J]. Physical Review A,2000,61(2):22301-22303.

[7] HILLERY M,BUZEK V,BERTHIAUME A. Quantum secret sharing[J]. Physical Review A,1999,59(3):1829-1834.

[8] ZENG G H,MA W P,WANG X M,et al. Signature scheme based on quantum cryptography[J]. Acta Electronica Sinica,2001,29(8):1098-1100. (in Chinese)
曾贵华,马文平,王新梅,等. 基于量子密码的签名方案[J]. 电子学报,2001,29(8):1098-1100.

[9] WEN X J,LIU Y,ZHANG Z J. Signature Scheme Based on Quantum Entanglement Swapping[J]. Journal of Electronics & Information Technology,2005,27(5):811-813. (in Chinese)
温晓军,刘云,张振江. 基于纠缠交换的量子信息签名方案[J]. 电子与信息学报,2005,27(5):811-813.

[10] LI Q,CHAN W H,LONG D Y. Arbitrated quantum signature scheme using Bell states[J]. Physical Review A,2009,79(5):1744-1747.

[11] WANG Y W. Cluster state quantum entangled signature scheme [J]. Computer Engineering and Applications,2012,48(5):93-95. (in Chinese)
王郁武. Cluster 态的量子签名方案[J]. 计算机工程与应用,2012,48(5):93-95.

[12] Cheng Xiao-feng. Arbitrated quantum signature with single particle[J]. Chinese Journal of Quantum Electronics,2015,32(1):77-82. (in Chinese)
陈晓峰. 基于单粒子的仲裁量子签名方案[J]. 量子电子学报,2015,32(1):77-82.

[13] SHI R H,HUANG L S,YANG W,et al. Multiparty quantum secret sharing with Bell states and Bell measurements[J]. Optics Communications,2010,283(11):2476-2480.

[14] BUHRMAN H,CLEVE R,WATROUS J,et al. Quantum fingerprinting[J]. Physical Review Letters,2001,87(16):167902-167904.

[15] TIAN J H,ZHANG J Z,LI Y P. A Quantum Multi-proxy Blind Signature Scheme Based on Genuine Four-Qubit Entangled State [J]. International Journal of Theoretical Physics,2016,55(2):809-816.