面向在线支付业务的容器资源弹性管理框架①

任 明, 陈 晨, 施跃跃, 鲁逸丁

(中国银联股份有限公司, 上海 201201)

摘 要: 随着在线支付业务的大规模应用, 系统运维人员需要更便捷的服务器资源管理机制来满足系统扩展需求. 本文提出一种基于容器技术的轻量化、弹性资源管理框架, 并给出应用服务的性能诊断方法以及集群规模的动态调整方法. 利用在线支付业务的典型数据处理场景进行实验, 实验结果验证了方法的有效性.

关键词: 容器集群; 弹性资源管理; 性能诊断

Elastic Container Resource Management Framework for Online Payment

REN Ming, CHEN Chen, SHI Yue-Yue, LU Yi-Ding

(China UnionPay, Shanghai 201201, China)

Abstract: With the large-scale application of online payment business, system operations staff need more convenient server resources management mechanism to meet the demand for system scaling. In this paper propose a lightweight and elastic resource management framework based on the technology of container. It also proposes the application service performance diagnosis method and the cluster dynamic adjustment method. Experimental results show that this approach is effective under typical data processing applications for online payment.

Key words: container cluster; elastic resource management; performance diagnoses

随着 IT 技术的不断发展, 日常生活中人们的在线支付频率快速提高, 特别是在金融服务领域, 在线支付业务已经大规模普及[1]. 以中国银联为例, 截至 2015 年 6 月, 发卡数量已经超过 55 亿张, 日均交易量超过 1 亿笔, 接口商户超过 2000 万家; 同时还积极发展互联网相关业务, 推出了包括银联钱包、银联在线支付、大众持卡人在内的互联网化产品. 业务的不断发展导致了 IT 系统的服务器规模激增, 运营维护的压力直线上升, 运维人员需要更便捷的服务器资源管理机制来满足系统运行环境的弹性伸缩需求, 保障应用服务的响应性和扩展性[2].

针对系统资源的弹性管理需求, 业界普遍采用云计算技术将应用所需的计算任务分配在资源池中, 允许不同的应用根据要求获取计算节点、存储空间、网络带宽等系统资源, 进而综合考虑系统运行时的资源监测结果以及应用类型, 实现资源利用率的优化. 然而, 传统基于虚拟机的云计算支撑技术存在系统运行开销大、生命周期管理开销大等问题[3][4], 难以满足在线支付业务的服务器资源快速弹性供给的轻量化管理需求.

容器是新型的操作系统级轻量级虚拟化与资源隔离技术, 与传统的虚拟机技术相比, 其资源管理开销更小、启动速度更快, 并更有利于实现发布过程的自动化[3,4]. 本文针对在线支付业务的服务器资源管理需求, 结合容器技术的轻量化优点, 提出一种服务器资源弹性管理框架, 并利用银联支付系统的差错处理平台实现该框架的测试验证, 测试结果显示, 应用该框架可以实现服务器资源的秒级管理和按需供给.

1 相关研究

虚拟化技术是普遍采用一种云计算支撑技术, 通过运行在物理服务器和操作系统之间的中间软件层, 利用虚拟机监视器(VMM)实现同一台物理机上的不同虚拟机相互隔离, 其优点在于资源管理难度低、隔离

① 收稿时间:2015-11-20;收到修改稿时间:2015-12-21 [doi:10.15888/j.cnki.csa.005245]

性好, 但缺点是系统开销较大、启动和停止时间长. 例如, VMM 的性能开销超过 30%, 进而也影响应用性能和扩展性[5,6]. 根据 VMWare[1]的配置建议, 1 个物理 CPU 最多支持 3 个虚拟 CPU.

容器是 Linux 操作系统隔离和管理系统资源的方式. 系统中可以同时运行多个容器, 系统内核为每一个容器创建独立的命名空间, 并可以限制其使用的系统资源的上限, 因此不同容器间的运行环境、性能相互隔离. 容器是一种新的虚拟化技术, 通过操作系统内核隔离不同的进程, 可以实现与传统虚拟机近似的隔离能力, 并且具有更小的资源开销、更快的启动速度. LXC[2]是容器技术的典型代表, 已被集成到 Linux 操作系统内核.

通过表 1 中虚拟机和容器在客户机系统、通信、性能、隔离、启动时间、存储六个方面的对比可以看出, 容器比虚拟机资源消耗少、启动时间短. 使用容器可以实现应用的快速启动、运行, 更适用于系统资源快速弹性供给的场景.

表 1　容器与虚拟机对比

	虚拟机	容器
客户机系统	每个客户机有自己的操作系统	客户机共享主机的操作系统
通信	通过网络通信	使用 IPC 机制通信
性能	虚拟机需要把客户机系统的机器指令翻译为主机操作系统的机器指令, 开销较大.	和主机的性能相近
隔离	虚拟机之间不能共享文件	容器之间可以共享文件
启动时间	分钟级	秒级
存储	每个虚拟机需要存储操作系统和应用程序	每个容器只需存储应用程序

2　基于容器的弹性集群管理框架

基于容器的集群管理架构如图 1 所示. 在逻辑拓扑层面, 系统中包括多个容器集群, 每个集群对应的一个应用服务, 单个容器集群包括一个或者多个容器实例(多容器实例情况下, 需要一个容器作为负载均衡器). 当集群规模调整后, 通过负载均衡器将负载进行重新分配. 由于集群的状态信息都存储在特定的键值存储主机中, 所以集群服务的停止或失效不会丢失应用状态, 保证了服务的可用性.

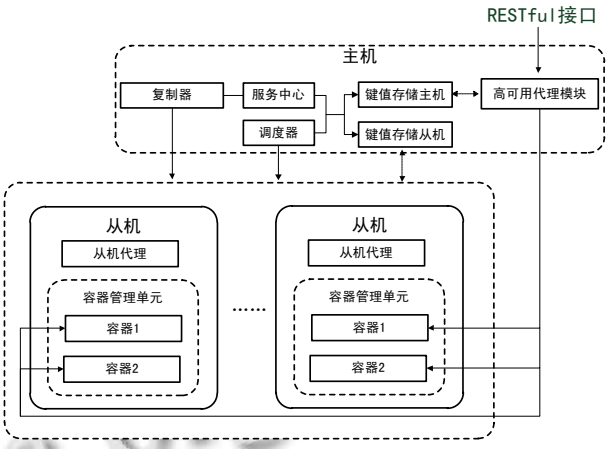


图 1　基于容器的集群管理架构图

在物理拓扑层面, 系统包括多台物理服务器, 每个集群中的节点都以容器实例的方式运行在服务器中; 同一个服务器可以运行一个或多个来自于多个集群的节点, 所以它们可以共用操作系统和服务器提供的服务和资源. 应用服务在部署之后具有唯一的访问地址, 将该地址存入在服务中心的路由表中, 终端用户访问应用服务之前, 从路由表中查找真实的访问地址后在进行访问.

主服务器(主机)用于管理和监控系统中服务器和运行的多个集群, 它不运行集群节点. 主要包括服务器管理和容器集群管理两部分, 前者包括从机池管理、资源监控等, 容器集群管理包括拓扑管理、部署管理、资源监控等. 每台从服务器(从机)中都运行了一个容器管理器, 负责监控服务器状态, 收集各种运行时信息, 控制运行的容器实例, 并与主机进行交互, 通过状态报告定期返回服务器和运行的容器实例的各种性能数据, 同时接收和执行主机的控制命令. 集群资源调整操作由主机计算得出, 并将控制命令通过调度器发往从机的容器管理器执行, 最后通过负载均衡达到调整的效果.

下面对本文中用到的符号进行定义:

定义 1.　S_1、S_2、…、S_n 表示 n 台服务器, $ssize(S_i)$ 表示服务器 S_i 上运行的应用服务数量, 即容器数量, S_{ij} 表示在第 i 个服务器上运行的第 j 个应用服务.

定义 2.　C_1、C_2、…、C_m 表示 m 个集群, $ssize(C_i)$ 表示集群 C_i 的应用服务数量, 即容器数量, C_{ij} 表示第 i 个集群的第 j 个应用服务, $server(C_{ij})$ 表示 C_{ij} 所在的服务器.

[1] http://www.vmware.com

[2] https://linuxcontainers.org

定义 3. KS_i 表示第 i 个服务器的当前性能状态, KS_{ij} 表示第 j 个应用服务的当前性能状态. S_CPU_i、S_MEM_i、S_DR_i、S_DW_i、S_NR_i、S_NW_i 分别表示第 i 个服务器的 CPU、内存、磁盘读取、磁盘写入、网络读取、网络输出的资源使用率, S_CPU_{ij}、S_MEM_{ij}、S_DR_{ij}、S_DW_{ij}、S_NR_{ij}、S_NW_{ij} 分别表示第 i 个服务器的第 j 个应用服务的相关资源使用率.

定义 4. KC_{ij} 表示集群 C_i 的第 j 个应用服务的当前性能状态. C_CPU_{ij}、C_MEM_{ij}、C_DR_{ij}、C_DW_{ij}、C_NR_{ij}、C_NW_{ij} 分别表示第 i 个集群的第 j 个应用服务的相关资源使用率.

2 集群资源弹性调整方法

系统资源的调整按照“监测-诊断-分析-执行”的环路实现资源的弹性供给. 如图 2 所示, 节点管理器通过监视服务器的资源使用率, 并通过系统诊断方法将计算得到系统状态发送给主机. 主机找出空闲态以及处于性能瓶颈态的服务器和集群, 并综合考虑全局的资源和负载分布状况制定调整计划, 然后相应的调整动作分配给资源回收管理器和资源供给管理器执行, 最后依赖于各个集群的负载均衡器, 将负载进行重新分布.

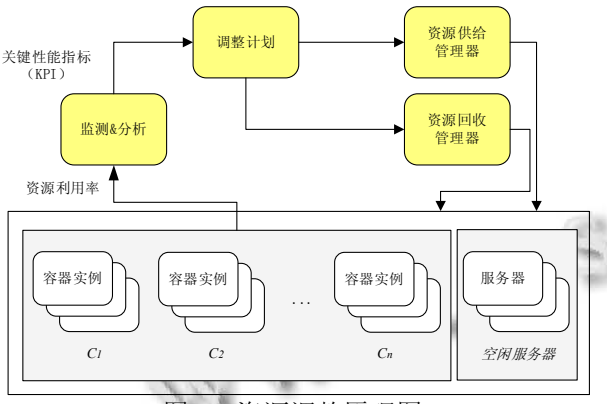


图 2 资源调整原理图

2.1 性能诊断

为了衡量服务器、应用服务的运行状况, 通过权重的方式对各种系统性能指标的重要性进行区分, 计算得到关键性能指标(KPI). 前述 KS_i 即是表示相应服务器的 KPI 值, 而 KS_{ij}、KC_{ij} 用于表示相应的应用服务 KPI 值. 定义 KPI 的取值范围是[0, 1000], 如图 3 所示, 当 KPI < 100 时, KPI 值随着各种资源使用率的上升接近线性方式增长, 此时系统性能(吞吐率)也基本呈线性增长; 当 KPI = 100 时, 系统性能达到最佳状态; 当 KPI > 100 时, 随着资源使用率的上升, KPI 值增长速度加快, 此时吞吐率开始下降, 在超过 150 时, 系统达到瓶颈状态, 负载增加将使系统性能下降. 基于上述指标, 将系统状态划分为如下区间: 空闲态(KPI ∈ [0, 30])、亚空闲态(KPI ∈ (30, 50])、常态(KPI ∈ (50, 120])、最佳态(KPI = 100)、亚瓶颈态(KPI ∈ (120, 150])、瓶颈态(KPI ∈ (150, 1000]).

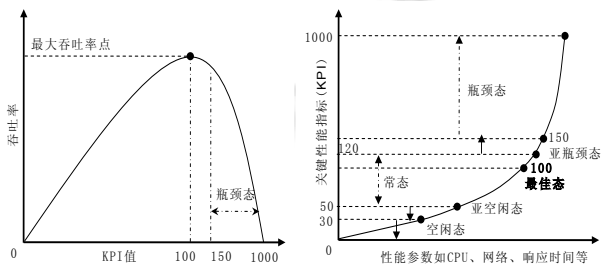


图 3 KPI 与吞吐率、性能参数之间的关系

记整体 KPI 指为 K, CPU、内存、磁盘读取、磁盘写入、网络读取、网络输出的单项 KPI 分别为: K_{cpu}、K_{mem}、K_{dr}、K_{dw}、K_{nr}、K_{nw}, 每项权重分别为 W_{cpu}、W_{mem}、W_{dr}、W_{dw}、W_{nr}、W_{nw}. 给出 K 值的计算方法如下:

$$K=\begin{cases}\sum_{type} W_{type}\times K_{type}, & when\ \forall(K_{type})\le 150\\ \max(W_{type}\times K_{type}), when\ \exists(K_{type})>150\end{cases}$$

其中 type = {cpu, mem, dr, dw, nr, nw}, K_{type} 的计算涉及具体资源的使用率(*usage_rate*)和使用阈值(*threshold*), 这里的阈值指特定系统资源的最佳使用状态, 例如, 根据排队论, 单核 CPU 的使用率在 75%时达到最佳. 由此可得:

$$K_{type}=f(usage\_rate, threshold)$$

函数 f 主要根据 KPI 值的增长曲线特征进行选取, 使得 K_{type} 取值在 *usage_rate* <= *threshold* 时基本呈线性增长, 在 *usage_rate* 达到 *threshold* 时, K_{type} 达到最佳状态值 100, 当 *usage_rate* ∈ (*threshold*, 100]时, K_{type} 随着 *usage_rate* 的增加增长率也越来越快. 简单的, 可以将 *usage_rate* 通过线性映射到相应的函数区间内, 映射公式为:

$$X=\begin{cases}usage\_rate*10/threshold, \\ \quad when\ usage\_rate\le threshold\\ (usage\_rate-threshold)*20/(100-threshold), \\ \quad when\ usage\_rate>threshold\end{cases}$$

基于上述关键性能指标, 可以对服务器、应用服务、容器集群的状态进行诊断, 为弹性资源调整提供依据. 这里使用符号 *check_bottleneck* 和 *check_idle* 描述诊断结果, 诊断方法如下:

服务器 S_i 诊断: 如果 KS_i > 150, 那么 *check_bottleneck*(S_i) = true, 即出现性能瓶颈; 如果 KS_i < 30, 则 *check_idle*(S_i) = true, 即服务器空闲.

服务 S_{ij} 的诊断: 如果 *ssize*(S_i) = 1, 即 S_{ij} 独享服务器 S_i, 此时, 为了使 S_{ij} 充分利用独占的服务器资源, 设定只有当 *check_bottleneck*(S_i) = true 时, 才有 *check_bottleneck*(S_{ij}) = true; 如果 *ssize*(S_{ij}) > 1, 即 S_{ij} 与其它服务共用一个服务器, 那么当 KS_{ij} > 50, 有 *check_bottleneck*(S_{ij}) = true, 即服务出现瓶颈, 这样可以避免 S_{ij} 对资源的过分占用影响同服务器中其它服务的运行. 如果 KS_{ij} < 10, 那么 *check_idle*(S_{ij}) = true, 即该服务现在处于空闲状态.

集群 C_i 的诊断: 集群诊断依赖于对每个服务的诊断, 当集群多数服务都处于空闲状态(比如超过 50%), 则 *check_botttle*(C_i) = true, 此时需要扩展集群的容器规模. 当集群多数服务处于空闲状态, 将之标记为空闲态, 即 *check_idle*(C_i), 可以减少集群中的容器数量.

2.2 弹性调整

集群资源调整包括集群扩展和集群收缩, 通过这两个操作, 实现应用服务所需资源的按需供给.

1)集群扩展算法

当集群各服务负载升高, 集群服务所在服务器不能处理转发过来的用户请求, 并对该服务器中的其它集群服务性能造成影响, 此时触发集群的扩展操作, 增加一个服务实例, 并部署到满足部署约束的服务器, 同时更新负载均衡转发列表.

算法 1　集群扩展算法

Algorithm expand_cluster(C_i): check if the cluster should be expand and add a service to it if needed.

Input: the cluster to expand

```
1.  /*check if the cluster is in bottleneck state*/
2.  if check_bottleneck(Ci) = true
3.      /*construct a service using the application, config and
    resource vector need */
4.      construct a service SR: application + service config +
    resource vector
5.      /*calculate KS'i and fill the state sections*/
6.      for each Si in Slaves do
7.          calculate KS'i using SR resource vector and Si
    resource usage vector
8.          if  ssize(Si) == 0
9.              add Si to candidate_set1
10.         else if  50≤KS'i≤90
11.             add Si to candidate_set2
12.         else if  30≤KS'i<50
13.             add Si to candidate_set3
14.         else if  KS'i<30
15.             add Si to candidate_set4
16.         end if
17.     end for
18.     /*find the server from normal set*/
19.     if candidate_set2 not empty
20.         S = ssize(S) is the smallest in candidate_set2
21.     end if
22.     /*find the sub-idle set */
23.     if candidate_set3 not empty
24.         S = ssize(S) is the largest candidate_set3
25.     end if
26.     /*find the server from idle set*/
27.     if candidate_set4 not empty
28.         S = ssize(S) is the largest candidate_set4
29.     end if
30.     /*return the server has not service yet*/
31.     if candidate_set1 not empty
32.         S = any S ∈ candidate_set4
33.     end if
34.     if S≠null
35.         /*do the deploy action*/
36.         deploy service SR to server S
37.         /*reset the load balance redirect list and let the
    new service to take load*/
38.         reset the load balance redirect list
39.     end if
40. end if
```

算法伪代码如算法 1 所示, 当判断集群处于瓶颈状态后, 在算法第 6 至 33 行选择部署新服务的服务器. 首先计算每个服务器在放置服务新增服务实例 SR 后的预期状态 KS'_i, 并据此将服务器划分为四类, 分别为未使用服务器集(*candidate_set*$_1$)、正常服务器集(*candidate_set*$_2$)、亚空闲服务器集(*candidate_set*$_3$)、空闲服务器集(*candidate_set*$_4$). 如果存在处于正常状态的服务器, 那么从中选择部署服务数量最少的服务器, 否则, 选择亚空闲态服务器, 即 $KS'_i \in [30, 50]$. 如果不存在上述两种状态的服务器, 则选择处于空闲态的服务器, 并将服务部署到服务数量较多的服务器中. 如果上述方法无法找到服务器, 则选择一个未部署服务的服务器.

由于算法的核心部分为服务放置选择过程, 即对

所有服务器按照状态进行分类, 并在可能存在的查找集合中进行最大(小)值操作, 所以算法复杂度为线性时间复杂度 O(N), 其中 N 为系统中服务器的数量.

2)集群收缩算法

集群收缩意在降低集群规模, 减少资源消耗. 对于集群 C_i, 检查 *check_idle*(C_i) = true 和 *ssize*(C_i) > 1, 如果满足条件, 则选择停止集群服务 C_{ij}, 并从负载均衡转发列表中删除, 从而完成集群的收缩操作.

该选择算法的伪代码如算法 2 所示. 首先判断是否存在服务所在服务器处于亚瓶颈状态或瓶颈状态, 并选择其中 KPI 最高的服务器, 并将其上的服务关闭, 减少其服务器的压力, 使其尽快进入正常的工作状态, 保持服务器的稳定. 如果存在服务所在服务器处于空闲状态, 那么选择其中运行服务数量最少者, 并将其上的服务关闭, 使其可以更快的进入未使用状态. 特别的, 当该服务器的服务数量为 1 时, 关闭服务后使其进入未使用的状态, 从而达到减少服务器数量的目标. 如果存在服务所在服务器处于亚空闲态, 那么选择服务数量最少者, 从而促使其进入到空闲态, 在进行服务器合并操作时处理, 减少服务器的使用数量. 若依然没有找到服务, 服务所在的服务器必然都处于正常工作态. 此时, 如果存在服务器 KPI 值大于 100, 那么选择 KPI 值最高者, 关闭其上运行的服务, 否则, 选择服务数量最多的服务器.

算法 2 集群收缩服务选择算法

Algorithm select_cluster_service(C_i): select a service from the idle cluster to shutdown

Input: the cluster name to expand

Output: the service to shutdown

```
1.  /*divide the sections by server state*/
2.  for each C_ij in Cluster C_i do
3.      m = server(C_ij)
4.      if KS_m ≥ 120
5.          add S_m to candidate_set_1
6.      else if KS_m ≤30
7.          add S_m to candidate_set_2
8.      else if KS_m ≤50
9.          add S_m to candidate_set_3
10.     else
11.         add S_m to candidate_set_4
12.     end if
13. end for
14. /*find from bottleneck set*/
15. if candidate_set_1 not empty
16.     return the service SR: server(SR)=S_m , S_m ∈
        candidate_set_1, and KS_m is the largest
17. end if
18. /*find from the idle set*/
19. if candidate_set_2 not empty
20.     return the service SR: server(SR)=S_m , S_m ∈
        candidate_set_2, and ssize(S_m) is the smallest
21. end if
22. /*find the server sub-idle set*/
23. if candidate_set_3 not empty
24.     return the service SR: server(SR)=S_m , S_m ∈
        candidate_set_3, and ssize(S_m) is the smallest
25. end if
26. /*find the service from norm-set*/
27. if candidate_set_4 not empty
28.     find S_t : KS_m is the largest
29.     if KS_t>100
30.         return the service SR: server(SR)=S_t , S_t ∈
            candidate_set_4
31.     else
32.         return the service SR: server(SR)=S_m , S_m ∈
            candidate_set_4, and ssize(S_m) is the largest
33.     end if
34. end if
```

上述算法主要对集群服务所在服务器的分类以及这些服务器范围内的查找操作, 所以其时间复杂度为 O(N), 其中 N 为集群的大小 *ssize*(C_i).

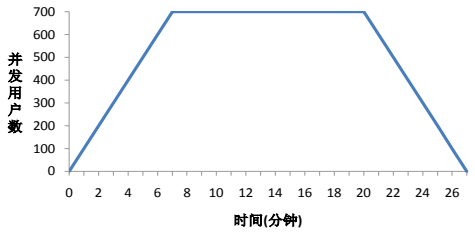
3 系统实验

实验采用银联支付系统的差错处理平台作为测试系统, 测试服务器包括 1 台主机以及 10 台从机, 每台服务运行一个节点管理器, 另有两台负载均衡器生成请求负载, 服务器的网络连接结构如图 5 图所示.

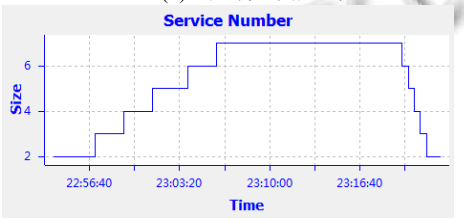
图 5 负载均衡测试环境配置

初始部署使用一个负载均衡器和一个 Web 应用服务器实例. 使用差错仿真软件模拟外部用户的真实的差错处理业务给系统增加负载, 负载变化如图 6(a)所示, 图 6(b)至(d)分别为集群规模、CPU、内存的变化

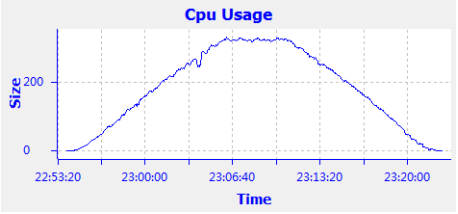
情况. 其中, 图 6(b)反映集群服务数目(即 Service Number)的变化(包括一个负载均衡器服务); 图 6(c)反映系统使用的 CPU 总量的变化(图中 Y 轴表示各个集群服务 CPU 使用率转化为单核使用率后的总和, 单位为%); 图6(d)反映系统使用的总内存的变化(图中 Y 轴表示各集群服务使用的内存之和, 单位为 MB). 可以看出, 图 6(c)的 CPU 总量曲线与负载曲线基本一致, 随着负载的增加, CPU 总量逐渐升高, 当判断集群处于瓶颈状态后, 图 6(b)显示集群服务数目逐渐增加, 并且每个新增服务的启动时间均小于 10s, 在负载稳定时最多有 6 个 Web 应用服务器实例, CPU 总量最高达到 300%以上; 随着负载的下降, CPU 总量也随之减少, 这表明多数集群服务进入空闲态, 进而触发集群规模的收缩, 并逐渐收缩到初始的单个 Web 应用服务器实例的状态. 图 6(d)中的内存总量变化曲线与集群服务数量变化曲线相似, 并未随着负载的减小而立刻降低, 这是由于 Web 应用服务器的内存使用变化速度不如 CPU 快捷, 并且自身需要占用较大内存, 只有当其被回收关闭后才会释放内存资源. 通过实验及所得的数据分析可见, 利用设定集群负载关键性能指标及最佳状态值, 通过监控-诊断-分析-执行的环路资源弹性方法及集群收缩算法, 可以有效稳定的根据差错业务的负载增减, 实现 CPU、内存、磁盘等资源指标的按需自动调整.



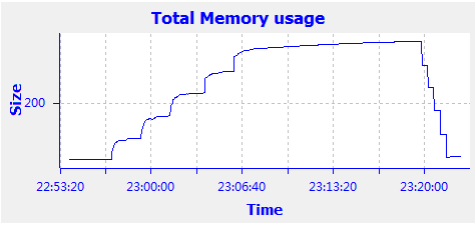
(a) 集群负载曲线



(b) 集群服务数目变化



(c) 集群使用的总 CPU 变化



(d) 集群使用的总内存变化

图 6 集群调整效果及资源使用情况

4 结语

本文对服务器资源的弹性管理问题进行了研究, 提出了一种基于容器技术的轻量化、弹性资源管理框架, 并给出应用服务的性能诊断方法以及集群规模的动态调整方法. 测试结果显示, 应用该框架可以实现服务器资源的秒级管理和按需供给.

参考文献

1 喻启红.关于支付机构跨行资金清算模式的研究.金融经济:理论版,2014,(8):152−154.

2 Xiao Z, Song W, Chen Q. Dynamic resource allocation using virtual machines for cloud computing environment. IEEE Trans. on Parallel & Distributed Systems, 2013, 24(6): 1107−1117.

3 Dua R, Raja AR, Kakadia D. Virtualization vs containerization to support PaaS. 2014 IEEE International Conference on Cloud Engineering (IC2E). IEEE. 2014. 610−614.

4 Xavier MG, Neves MV, Rossi FD, et al. Performance evaluation of container-based virtualization for high performance computing environments. 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE. 2013. 233−240.

5 Cherkasova L, Gardner R. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. USENIX Annual Technical Conference. Apr 2005.

6 End-To-End Performance Study of Cloud Services. http://highscalability.com/blog/2010/5/26/end-to-end-performance-study-of-cloud-services.html.