计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2017,26(11):67−75 [doi: 10.15888/j.cnki.csa.006034]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 CoreOS 面向负载整合的集群调度研究①

张宝婷[1,2], 芮建武[1], 周　鹏[1], 武延军[1]

[1](中国科学院 软件研究所 基础软件国家工程研究中心, 北京 100190)
[2](中国科学院大学, 北京 100049)

摘　要: CoreOS 是基于 Docker 的新型容器化集群服务器操作系统, 发展迅速, 已经得到 OpenStack、Kubernetes、Salesforce、Ebay 等主流云服务商的支持, 云环境中负载是动态的, 相应的其资源需求是动态变化的, 这给集群资源高效利用带来了挑战, 静态预分配峰值资源的策略带来云端资源的巨大浪费, 同时空转的计算浪费大量能耗. 本文提出的面向负载整合的集群调度系统 (简称 LICSS) 实时监控集群负载分布情况, 调度时使用紧凑式调度策略分配计算节点, 运行时利用任务迁移技术对负载进行动态整合, 实现及时收集释放空转资源降低资源能耗浪费的目的. LICSS 系统设计实现了节点负载度量、任务度量、负载整合算法, 并测算出节点自适应负载阈值. 实验表明, LICSS 系统能够根据不同时段集群负载动态变化情况对负载进行有效整合, 提高了 12.2% 的平均资源利用率, 并且基于任务整合在低负载时段触发富余节点休眠降低集群能耗.

关键词: 集群操作系统; 负载整合; 集群任务调度; 负载阈值

引用格式: 张宝婷,芮建武,周鹏,武延军.基于 CoreOS 面向负载整合的集群调度研究.计算机系统应用,2017,26(11):67–75. http://www.c-s-a.org.cn/1003-3254/6034.html

Research on Cluster Scheduling Based on CoreOS Oriented Load Integration

ZHANG Bao-Ting[1,2], RUI Jian-Wu[1], ZHOU Peng[1], WU Yan-Jun[1]

[1](National Engineering Research Center of Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: CoreOS is a new containerized cluster server operating system based on Docker. It is developing rapidly and has been supported by mainstream cloud service providers such as OpenStack, kubernetes, Salesforce and Ebay. The load in the cloud environment is dynamic, therefore the resource requirement is dynamic, which poses a challenge to the efficient utilization of cluster resources. The strategy of static pre-allocation of peak resources brings a huge waste of cloud resources, and in the meantime idle calculation wastes a lot of energy consumption. In this paper, load-integrated cluster scheduling system (LICSS) is proposed to monitor the load distribution of the cluster in real time. In order to release idle resources in time to reduce energy consumption in the scheduling process, the nodes are allocated using the compact scheduling strategy and the load is dynamically integrated by the task migration technique. The LICSS system implements the node load metric, task metric, load integration algorithm, and calculates the node adaptive load threshold. Experiments show that the LICSS system can effectively integrate the load according to the dynamic changes of the cluster load in different time periods, and can improve the average resource utilization rate by 12.2%, and reduce the cluster energy consumption by triggering the dormancy of the redundant nodes during the low load period.

Key words: cluster operating system; load integration; cluster task scheduling; load threshold

① 基金项目: 中国科学院战略性科技先导专项课题 (XDA06010600)
收稿时间: 2017-02-13; 修改时间: 2017-03-02; 采用时间: 2017-03-06

1 引言

虚拟化技术是云计算的关键技术. Docker[1]作为一种轻量级、高性能的虚拟化技术在云计算领域得到越来越广泛的应用. CoreOS[2]是基于 Docker 的面向云计算的新型容器化集群服务器操作系统, 发展迅速, 已经得到 OpenStack、kuberbetes、Salesfoece、Ebay 等主流云服务商的支持, 其极佳的可扩展性和广泛地实用性是本文选择 CoreOS 做实验平台的主要原因, 但本文的方法也适用于其他云计算集群调度平台.

资源按需分配[3]是云计算的一个主要特征, 但是为了保证服务始终可用, 实际是按照最高 (峰值) 负载需求部署服务器, 并保持所有服务器 24 小时运行, 这在数据中心很常见. 这是种静态资源预分配方案, 该方案导致在大量时段, 节点上的负载分布是稀疏的而不够紧凑, 由此带来巨大的资源浪费, 研究表明服务器低负荷运行跟高负荷运行能耗相当, 因此空转的计算同时伴随着大量能耗浪费.

据调查, 数据中心服务器的利用率很低, 一般在 30% 左右. 空转的机器带来很大的能耗浪费, 数据中心总能耗成本每年以 20% 速度增长[4,5].

针对上述问题, 本文为云计算环境设计实现了一个紧凑型任务调度系统, 在用户响应得到有效满足的前提下对任务负载进行整合, 使得任务在节点上分布紧凑, 减少运行中的空转计算节点 (比如, 闲时触发空闲节点低能耗休眠, 峰值时段按需唤醒); 任务整合发生在两个阶段: 新任务进入时调度分配一个满足任务负载需求的最忙的计算节点; 运行时选择低负载节点进行任务迁出. 该研究的难点包括: 节点负载度量、任务度量、节点自适应负载阈值选取、负载整合算法、任务迁出节点评估.

通过节点度量去动态估算节点负载, 任务度量去动态估算任务资源需求量, 来减少实时监控数据的波动误差, 为负载整合算法提供准确平滑的基础数据. 根据用户对服务请求响应时间, 自适应调整节点负载上下限阈值, 提高节点资源利用率. 为了确保服务可用, 高峰时段过载节点上的部分任务迁出到有空闲的节点, 空闲时段负载过低的节点上所有任务迁出成为清空节点, 可以触发清空节点低能耗休眠实现资源回收同时节约能耗.

2 相关工作

基于阈值虚拟机集群的动态调度[6]在研究领域的相关研究主要基于静态阈值和动态阈值的动态调度, 下面介绍相关的研究现状.

静态二维阈值触发机制[7,8]包含 CPU, 内存的二维资源负载向量, 当二维资源负载向量中的每一维资源超过其相应的上下限阈值一段时间时, 会触发动态迁移. 连续监控节点的资源消耗, 当 CPU 和内存消耗分别超过 85% 或 95% 时, 节点就判断为过载, 当节点的平均 CPU 利用率和内存消耗分别低于 50% 和 80% 时, 则判断为负载过低, 应该将上面的工作负载进行迁移, 迁移到其他合适的机器上, 然后将该机器进行关闭休眠, 节约能耗.

过载阈值过低, 迁移工作负载就会非常活跃, 频繁的迁移会提高很多的迁移成本, 降低应用性能. CPU 对阈值影响也比较大, 因为 CPU 需求比内存需求更快[9], 工作负载对于内存的需求变化非常缓慢, 所以 CPU 的阈值设置需要更谨慎一点. 20% 或 30% 的低 CPU 阈值导致 CPU 空闲时间过长, 将 CPU 阈值提高到 50% 以上, 不会降低 CPU 使用率, 因为变为内存限制, 所以把 50% 设置为 CPU 下限阈值.

当应用的负载动态变化时, 应用需要满足服务等级协议 (SLA), 例如最小的吞吐率, web 服务应用的平均和最长请求响应时间, 平均和最长服务暂停时间等. 但是上述论文利用静态的上下限阈值很显然处理不了这种情况, 不能根据 SLA 动态的调整阈值, 这样就不能在保证不影响应用服务质量的情况下, 提高服务器资源.

动态阈值的触发机制[10]通过给各个维度各自设置权重得到所在节点的综合利用率, 资源维度是 CPU、内存、网络带宽三维向量, 通过与一个经验系数相乘来调整阈值的上下限, 正相关于负载的动态变化, 该经验系数是根据用户应用的动态行为去动态设置的, 节点当前负载的状态与阈值的上下限进行比较, 小于下限阈值的或者大于上限阈值都会触发动态调度, 减小了数据中心的能耗, 但是, 该方法的不足是通过经验系数调整上下限阈值, 不能够随着应用负载的动态变化进行智能的灵活调整, 不够精准.

文献[11]根据当前节点的 CPU 利用率、内存利用率和网络带宽利用率动态的计算阈值, 这种方法能够根据不可预测的工作负载自适应的计算负载上下限阈值, 避免了频繁迁移带来的不必要的功耗和性能损失, 然而这种方法只根据单个节点的负载进行计算阈值,

却没有站在集群全局宏观的角度去计算阈值, 当集群整体的负载量很大时, 就会没有合适的节点可以接收待迁移的虚拟机.

文献[12]提出一种基于对虚拟机生命周期中收集的历史数据和节点的历史数据的统计分析结果去自适应调整资源利用率上下限阈值, 分析过程是利用样本平均值和标准差来计算收集的历史数据在最近一段时间内的分布特征, 这样做的原因是应用程序过去一段时间的行为与当前的行为是有关联的. 但是这种方法的缺陷是只考虑了 CPU 资源利用率, 没有考虑其他的资源 (内存利用率, 网络带宽利用率等).

工作负载的使用要满足用户与服务提供商协商好的服务等级协议 SLA, 例如为了使交互式 web 应用程序满足多少秒的平均响应时间, 运行应用程序的 VM 的平均 CPU 利用率必须保持在 60% 以内等. 文献[13]提出一个反馈控制系统, 对满足 SLA 的应用服务质量和更低的集群成本之间进行了一个折衷, 用 web 服务器的响应时间来控制 CPU 利用率, 用于管理应用程序的服务质量和分配使用的 CPU 资源. 该控制系统在运行 Apache web 服务器的实验测试平台上实现, 并使用第 90 个百分点的响应时间作为 SLA 度量数据.

集群状态可能由于持续触发迁移或者连续的开启或关闭服务器导致震荡的不稳定状态, 迁移的决定不仅要根据当前的状态, 还要根据之后的状态作处理. 在文献[14]中为了确保集群的稳定性, 需要用一个检测窗口分析状态的稳定性, 检测窗口过小, 会让检测速度很快, 但是可能不够稳定, 如果检测窗口很大, 会过滤掉更多的无效的迁移, 但是检测速度会很大, 为确保集群处于稳定状态, 在一定大小的检测窗口中计算平均值和标准方差, 利用最小二乘法分析状态趋势, 避免了迁移后资源利用率回落到正常范围的情况, 导致不必要的迁移, 提升迁移成本, 降低集群整体性能.

以上是关于虚拟机动态资源调度相关的研究工作, 下面介绍基于 docker 容器的集群针对动态变化的负载相关的研究现状.

Docker Swarm[15,16]是目前 Docker 社区原生支持的集群工具, 使用和 Docker engine 一样的 API, 简单, 易用, 灵活, Docker Swarm 会根据集群中服务器的 cpu, 内存等资源使用情况进行决策, 去选择部署的服务器.

Swarm 的优点是易于对集群进行管理. 但是当负载随时间动态变化时, Swarm 不支持自动缩放机器节点, 需要使用其他的解决方案, 例如: docker-machines, 在远程创建安装 docker 的主机, 并用 Swarm 命令将其添加到现有的 Swarm 集群中.

Swarm 的不足的是需要手动编写脚本, 监控集群的 CPU, 内存, 网络的使用情况, 一旦超出设定的上下限阈值, 就触发收缩集群事件, 需要有代码经验的人才能进行这个操作, 然而对于外行就没办法使用, 不方便, 还有这里的上下限阈值是根据人工经验设定的静态值, 没有根据不同任务的负载情况进行动态的自适应调整.

Kubernetes[17,18]也是一个基于容器技术的分布式架构, 其具备完备的集群管理能力, 包括服务发现, 资源调度, 负载均衡, 在线规模伸缩, 故障发现, 自我修复等. 其资源调度模块也是考虑了工作节点和容器的多种资源使用情况. 对于集群的节点数量可以根据动态变化的负载动态调整以满足最终的用户的性能要求, 这是自动化进行的, 无需手动操作, 但是也是人工设定静态的上下限阈值. 人工根据经验设置的静态阈值是有缺陷的, 不同任务对于资源的需求情况是不同的, 为了给不同的服务保证相对应的服务质量, 需要根据应用和动态的负载情况自适应的调整上下限阈值, 还有在采集资源数据时, 没有考虑数据由于网络不稳定等问题存在的瞬时波动误差, 会造成频繁的集群收缩, 造成不必要的迁移, 增加了集群运行的成本.

针对以上, 提出了一种面向 CoreOS 集群中动态的负载变化进行资源整合的算法, 通过滑动平均滤波算法, 减少采集数据的误差波动, 来提高迁移时机的准确性, 根据用户服务请求的响应时间来自适应的调整动态调度触发的阈值, 将高峰时段高负载的应用拆分迁移到空闲的服务器上, 以确保每个容器实例的资源需求, 当多个物理服务器的资源利用率过低时, 需要进行服务器合并, 将这些容器实例迁移到尽量少的物理服务器上, 使其他闲置的服务器休眠, 提高资源利用率, 达到节约能耗的目的.

3 系统设计

3.1 CoreOS 系统介绍

CoreOS 的基础组件有 Etcd, Fleet, Systemd 等, Etcd 提供分布式配置存储数据库, 包含集群全局信息, 集群中任意一个节点都能够快速获取集群中所有节点和服务的实时状态. Fleet 跨节点的操作集群中相应节点在本地的 Systemd API 来操作服务.

Etcd 是 CoreOS 集群中唯一的 key-value 分布式配置共享存储系统, 采用 raft 算法保证分布式系统强一致性, 所有数据都被存储在 Etcd 中, 例如: Unit 文件(容器服务的配置文件), 集群包括其中各个节点的元信息, Unit 状态等等. Etcd 通过其提供的 RESTful API 成为用于联系 CoreOS 集群中各个节点的桥梁. Unit 文件存储在 Etcd 的/_coreos.com/fleet/unit/路径下面, 集群中各个节点的信息存储在/_coreos.com/fleet/machines/路径下面, 容器运行的状态信息存储在/_coreos.com/fleet/states/路径下面, 可以通过 Etcd 在相关路径下面去存取相关数据.

Fleet 在机器和服务之间构建一个虚拟化层, 把服务按照用户要求调度到不同的机器上. 通过 Etcd, Fleet 可以完成对服务健康状态的管理和监控并且根据这些状态来调度服务. 采用了 Systemd 作为机器层面的服务管理工具, 借助 Systemd 的 Unit 配置文件, Fleet 可以完成集群层面的服务脚本编写. Fleet 提供了容灾机制, 当集群中有节点宕机, Fleet 会把该节点上的容器调度到其他可以被调度的节点上.

CoreOS 整体架构图如图 1 所示, 在 CoreOS 集群里, 每一个节点上都运行着一个 fleetd 服务, 提供 agent 和 engine 两个功能, engine 扮演调度决策, agent 负责执行任务. unit 文件中会提供关于部署节点的需求信息, 例如: host 元数据, 相对于其他 Unit 的位置等等, 任务的执行命令等, 任务用 unit 文件进行配置, 用容器运行在节点上. engine 负责调度决策, 在跟 agent 协调过程中, 周期性的根据 etcd 发出的事件触发相应的调度决策, 进行集群的管理. engine 使用租赁模式保证任何时候只有一个 engine. engine 适应最简单的调度算法: 即首先根据用户的设置调度到期望的服务器上, 在此基础上优先调度到运行任务数目最小的节点上面. agent 服务执行任务, agent 通过 d-bus 跟本地的 systemd 进行通信, 执行任务, agent 周期性在 etcd 上同步节点和任务状态, 完成任务的执行, 和节点状态的更新, 供 engine 进行集群的管理.

3.2 系统架构设计

本节内容主要介绍了负载整合集群调度系统 LICSS 的设计, 该系统包含节点负载度量, 任务度量, 负载整合算法, 自适应负载阈值估算方法.

LICSS 系统总体设计中:

1) 通过节点负载估量和任务估量方法, 减少资源实时监控的波动误差, 来提高迁移时机的准确性, 降低迁移的成本.

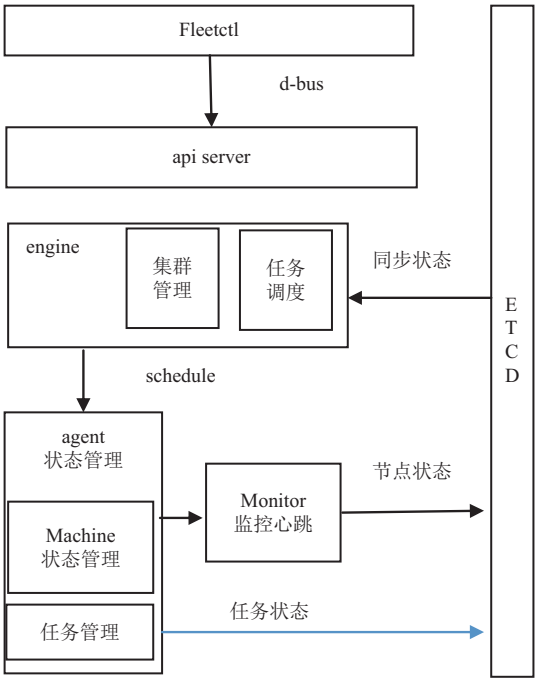

图 1 CoreOS 任务管理整体架构图

2) 通过用户服务请求的响应时间, 自适应的调整阈值, 提高资源利用率.

3) 能根据任务的动态负载进行整合, 在用户响应得到有效满足的前提下, 提高资源利用率, 节省服务器资源.

LICSS 系统的架构图如图 2 所示, 其中加黑字体表示为 LICSS 系统的模块.

3.3 节点负载度量方法

由于集群节点要支撑其上很多的任务去运行, 需要提供相应的资源, 需要度量节点上的资源使用情况, 根据这个去进行 LICSS 系统的负载整合算法, 由于容器的迁移会占用大量的网络带宽, 本文暂时不考虑这个资源维度, 只考虑对于容器运行时比较稀缺的 cpu 和内存作为监控资源的指标, 用来衡量节点负载状态, 二维向量表示为: <cpu, mem>, LICSS 算法会同时考虑两种资源维度, 避免只优化某种资源而导致忽略其他资源, 导致其他资源超负荷, 节点的资源可用量表示为 U_i=< U_{i_cpu}, U_{i_mem}>. 先通过对节点负载按照上述二维维度进行监控采集实时负载数据, 然后利用滑动平均滤波算法[19]减小采集数据的波动误差.

设负载量$\{X_t|t=1, 2, 3, \ldots, n\}$负载量数据包含两种数据成分, 一种是稳定有效数据, 一种是随机误差数据, 比如用户远程访问服务, 如果网络带宽不稳定, 就会造成用户请求的响应时间不稳定.

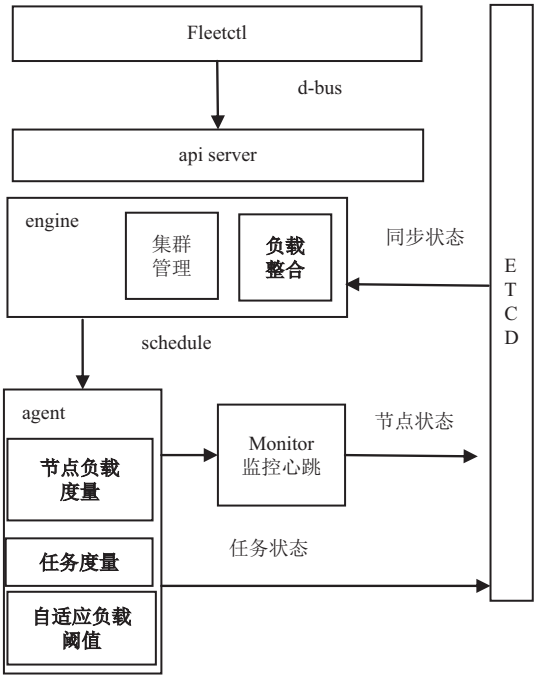

图 2 LICSS 系统架构图

负载量数据 y_i 可以表示为:

$$y_i = f_i + e_i \quad i=1, 2, 3, \ldots, n$$

y_i 为了降低误差数据对资源整合算法的影响, 所以要通过滑动平均滤波算法对 e_i 做平滑滤波处理, 得到可靠地负载量数据, 为资源整合算法奠定基础.

对 n 个采样数据没 m 个相邻数据的小区间进行不断的局部平均, 以此来过滤掉随机误差数据. 取 m 个相邻的采样数据, 对其取平均值, 如下面公式所示:

$$y_k = \frac{1}{m}\sum_{i=k-m+1}^{k} f_i \tag{1}$$

当 m=6 时,

$$\begin{aligned} y_6 &= (f_1 + f_2 + f_3 + f_4 + f_5 + f_6)/6 \\ y_7 &= (f_2 + f_3 + f_4 + f_5 + f_6 + f_7)/6 \end{aligned} \tag{2}$$

由上面两个公式看出, 每次求负载量数据都要进行 m 次累加运算, 这个操作将会很耗费 CPU 资源, 降低溪头能够运行速度, 改为如下公式:

$$y_n = y_n - 1 + (f_n - f_n - m)/m \tag{3}$$

根据上述公式计算负载量数据只有当第一次求滑动平均滤波时将区间内的 m 个数据累加取平均值, 之后的负载量数据计算仅需根据上述公式把新采集的负载量数据减去最老的数据后除以 m 即可, 很节省 CPU 资源的使用, 对于在线运算速度得到了很大的提升.

如果采样的负载数据还没有达到 m 个, 代表采集数据不够, 需要继续采集数据达到 m 个后, 就可以进行滤波计算, 对于给定的阈值 T, 如果滤波后的负载量数据连续 n 次超过阈值 T, 则发出警告进行资源整合, 否则忽略继续正常执行.

说明: 这里的每个采样数据是周期数据, 每经过一个周期采样一次数据, 进行实验得到, 这里周期定为一分钟比较合理, 不然数据量很大, 浪费 CPU 计算资源.

用户远程访问容器服务, 由于不稳定的网络带宽, 服务器处理请求时就会时而忙, 时而闲, 运行中的容器需要的资源就会不稳定, 如果在瞬间波动时, 就进行资源整合, 假如瞬时波动过后, 任务负载又恢复到了正常的负载状态, 这就是在做没必要的工作, 还花费了没必要的迁移成本. 为了降低瞬时波动对于资源整合的影响, 利用上述公式 (2) 进行滤波过滤掉误差数据, 使资源整合得到很好的效果.

3.4 任务度量方法

任务利用容器运行在在集群节点上, 根据任务的运行情况, 所在节点分配给相应的资源, 如果所在的节点不能满足资源需求, 节点负载就会过载, 影响任务正常运行的性能. 任务资源需求量表示为L_k=<L_{k_cpu},L_{k_mem}>.

先通过对任务资源需求量按照上述二维维度进行监控采集实时资源需求数据, 然后利用估量方法减小采集数据的波动误差. 估量任务的方法同上一节节点负载度量的方式. 为之后的负载整合算法奠定基础, 对于任务估量的历史记录进行存储, 为以后运行新任务时做准备, 待调度的新任务的资源需求量参考历史记录中类似任务的资源需求量.

3.5 自适应负载阈值估算

节点的可用 CPU 资源的上限阈值为$A_{n_cpu_up}$, 下限阈值为$A_{n_cpu_down}$, 节点的可用内存的上限阈值为$A_{n_mem_up}$, 下限阈值为$A_{n_mem_down}$, 首先用户手动设定静态的下限阈值, 对于上限阈值通过用户可以忍受的请求响应时间进行动态的自适应调整, 用户与 web 网站交互可接受的平均响应时间一般约为 4 秒[20]. 若请

求响应时间在 4 *s* 以内, 该节点都为合理的负载情况, 若请求响应时间在 4 *s* 以外, 判定该节点超负载, 达到了负载的上限阈值, 触发负载整合的事件.

3.6 负载整合算法

根据节点负载度量模块和任务度量模块分别得到在本周期的节点负载和任务的资源需求量, 为负载整合算法奠定基础.

1) 首先设计了目标节点选择算法, 可以作为之后负载整合共同调用的函数, 为之后的算法奠定基础.

算法1. 节点筛选算法伪代码

输入: 任务k资源需求量L_{k_cpu}, L_{k_mem}

```
for节点i in除了当前节点以外的所有节点;
    if U_{i_cpu} < L_{k_cpu}:
      过滤掉节点i;
    endif
    if U_{i_mem} < U_{k_mem}:
      过滤节点i;
    endif
endfor
```

2) 新任务进入时调度分配一个满足任务资源需求的最忙的计算节点, 通过该新任务的名字从任务估量模块的历史记录去搜索相似的任务, 得到该新任务的资源需求量L_{k_cpu}和L_{k_mem}, 带入下面的算法, 如果没有就根据用户的设置计算.

算法2. 新任务初次调度算法伪代码

```
调用 节点筛选算法;
if所有节点都被过滤掉:
    启动一台休眠的节点, 任务k调度到该节点上;
  else
    if
    过滤剩下的节点先按照CPU从小到大排序, 再按照内存从大到小进行排序;
    任务k调度到排序后的第一个节点上;
  endelse
endif
```

3) 运行时, 当节点的 CPU 负载或者内存负载超过相应上限阈值时, 触发动态负载整合, 选择低负载节点进行任务迁出, 当当节点的 CPU 和内存的资源可用率全部都低于下限阈值时, 也同样触发负载整合事件, 整合到尽可能少的节点上, 空闲出来的节点进行休眠, 节约能源.

首先根据负载超过的是哪一个上限阈值, 得到其为主要的负载资源, 根据主要负载资源进行负载整合, 如果是 CPU 超出上限阈值, 则把其上运行的任务按照 CPU 负载量从高到低排序, 从最高的开始迁出, 遍历其他正在运行的节点, 首先通过过滤掉不满足任务资源需求的节点, 如果全部的节点都不符合任务需求, 则从休眠的节点中启动一台新的节点, 把任务调度到上面. 否则在剩下的节点中, 按照 CPU 可用量进行从小到大排序, 选择最小的一个调度到上面, 这样做的目的是让任务尽量集中的调度在尽可能少的节点上, 在不影响任务性能的情况下减少节点的使用, 节约能源.

算法3. 运行时动态负载整合算法伪代码

输入: 节点负载$U_{cpu}(t_i)$, $U_{mem}(t_i)$

```
if 1-U_{cpu}(t_i) > A_{k_cpu}:
  根据CPU对该机上运行的任务从大到小排序;
  得到第一个为该迁出的任务k;
  调用 节点筛选算法;
  if所有节点都被过滤掉:
    启动一台休眠的节点, 任务k调度到该节点上;
  else
    过滤剩下的节点按照CPU从小到大排序;
    任务k调度到第一个节点上;
  endelse
endif
```

当$1-U_{mem}(t_i) > A_{n_mem_up}$时, 伪代码同上;

```
if U_{cpu}(t_i) < A_{n_cpu_down} AND U_{mem}(t_i) < A_{n_mem_down}:
  for k in该节点上正在运行的任务;
  调用节点筛选算法;
  if所有节点都被过滤掉:
    不进行迁出;
  else
    剩下的节点按照CPU, 内存可用量
    从小到大进行排序;
    任务k被调度到排序后的第一个节点上;
  endfor
  若当前节点上没有运行中的任务, 则休眠;
endif
```

内存的情况同以上 CPU 做法, 内存伪代码同上 CPU 伪代码.

循环遍历该机器上所有的正在运行的任务, 对于每一个任务, 遍历其他正在运行的节点, 首先通过过滤掉不满足任务资源需求的节点, 如果全部的节点都不符合任务资源需求, 则不进行迁出; 否则在剩下的节点中, 首先按照 CPU 可用量进行从小到大排序, 然后按照内存可用量进行从小到大排序, 选择资源可用量最

小的即排序后的第一个节点调度到上面, 那原来的这个节点上, 如果所有的任务都被迁移出去了, 则进行休眠, 节约能源.

4 系统实现

通过监控和采集集群中节点的负载情况和任务资源需求量给本文中的 LICSS 系统的负载整合算法奠定了基础. 监控数据主要根据二维向量<cpu, mem>作为指标进行采集. 本文主要利用 Prometheus 资源监控工具进行采集 Metrics 数据.

Prometheus 是一款开源的业务监控和时序数据库, 它将所有信息都存储为时间序列数据, 存储在自带的数据库中; Prometheus 能够按照给定的时间间隔收集所配置目标服务的指标、执行规则表达式、展现结果.

Prometheus 系统利用几款采集工具去采集 Metrics 数据, 例如: apache 服务数据采集工具 apache_exporter, 节点资源监控工具 node_exporter, 容器资源监控工具 cAdvisor, 通过采集到这些 Metrics 数据, 就可以全方位了解到集群中各个节点当前的负载情况和运行中的任务的资源需求量, 给负载整合算法提供数据基础.

5 实验部署与验证

5.1 实验环境

为了验证 LICSS 系统的效果, 本文实验同时在两个拥有相同配置的集群中进行相应的操作并对比. 每个集群都拥有十个节点. 每个节点的内存为 2 G、双核处理器, 在每个节点上都部署有 CoreOS stable 1185.5.0.

5.2 实验设计与结果分析

实验采用 apache 服务器运行 php 应用去模拟动态负载情况, 对外暴露 ip 和端口, 用户可以访问, 通过用 Webbench 工具给每个 php 应用模拟用户并发访问, 通过设置不同的并发量和运行测试时间, 模拟高低峰时段的动态负载情况.

求 Fibonacci 数列的计算复杂度非常高, 递归计算的复杂度是 $O(1.618^n)$, 当 n=100 时, 递归函数的计算复杂度是 2.8×10^{10} 左右, 所以用这个作为消耗 cpu 资源的任务.

使用 Webbench 进行并发压力测试, 压力测试命令是: webbench -c 10 -t 3600 http://ip:port/fibnacci.php?fib=50, 并发请求开始 10 个, 每次请求增加 10 个进入下轮的请求, 每一轮循环持续一小时, 到达最大 80 个并发时结束, fib 参数从 10 个开始, 每次请求增加 10 个进入下轮的请求, 每一轮循环持续一小时, 到达最大 80 个并发时结束. 从一天的 7:00 开始, 这样就可以持续做 8 小时的请求增加实验, 之后的每个小时减少请求量进行实验.

对于节点下限阈值的设置: 设置节点 CPU 负载下限阈值$A_{n_cpu_down}$=50%, 根据用户请求的响应时间不超过 4 秒对上限阈值进行自适应调整. 内存负载下限阈值为$A_{n_mem_down}$=80%.

设计两个实验, 第一个是平均数算法和滑动平均滤波算法在相同负载压力的变化下, 进行对比试验.

由图 3, 观察时间坐标轴在 7:00 之后, 增加负载量, 需要启动多个休眠的机器, 由于网络带宽的不稳定, 平均数算法由于数据波动性误差, 迁移次数出现明显的不稳定, 相对来说, 滑动平均滤波算法的迁移次数较少, 当 15:00 负载最大时, 因为整个集群的负载比较重, 所以整个集群都很不稳定, 两个算法迁移都很频繁, 整体看滑动平均滤波算法相对于平均数算法减少了 20% 不必要的迁移, 因此减小了资源整合的花销.

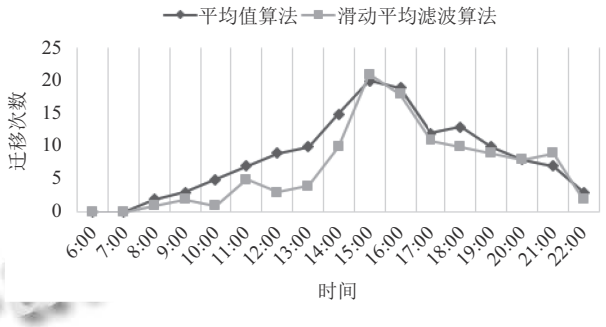

图 3 平均数算法与滑动平均滤波算法迁移次数的比较

随机分配集群调度算法, 与本文的资源整合集群调度算法, 进行对比试验.

由图 4 看出, 从 7:00 开始随着负载量的增加, 随机分配算法的节点资源利用率跟负载量成正比, 逐渐上升, 这是由于随机分配算法中, 集群中的节点全部开启, 当负载量较低时, 其中有些节点并没有分配到任务, 却依然在运行, 导致资源利用率较低. 而负载整合算法从 7:00 开始资源利用率就很高, 随着负载量的增加, 曲线基本保持平稳, 当 15:00 时由于容器动态迁移占用了一定的资源, 使性能下降, 节点资源利用率低于随机分配算法的. 之后随着并发量的减少, 随机分配算法的资源利用率下降, 而资源整合算法资源利用率比较平稳. 通

过该实验看出资源整合算法明显的提高了集群的资源利用率.

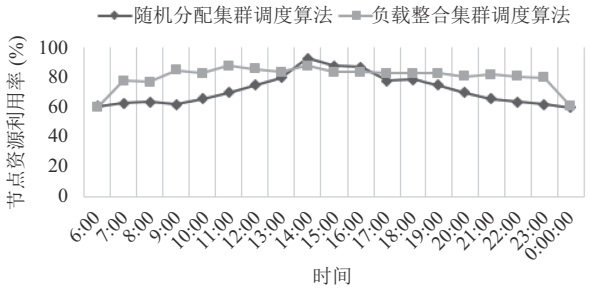

图 4 随机分配与资源整合调度算法关于节点资源利用率的对比

在 10:00 时, 当资源利用率差距最大时, 负载整合算法比随机分配算法的资源利用率提高了 37%, 对于平均资源利用率, 负载整合算法比随机分配算法提高了 12.2%, 可以看出在整体上资源利用率有一定的提高.

在实验中也绘制了随着负载的动态变化, 不同的任务在 10 个节点之间进行动态负载整合的情况.

通过图 5 看出, 刚开始时 10 个 apache web 应用都分别部署在 10 个节点上, 由于 7:00 之后一点点加负载压力, 所以 3 个应用被资源整合到一个节点上, 剩余九个节点被休眠, 之后, 随着负载压力的不断增大, 逐渐被整合到剩余被休眠的节点上, 直到 15:00 达到负载峰值, 之后随着负载减小, 运行的服务器数量也在减少. 通过该实验看出通过动态的资源整合节约了服务器资源.

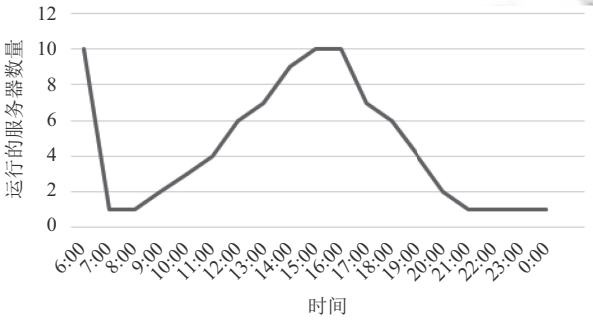

图 5 负载整合算法效果图

从实验结果看出, LICSS 系统能够根据不同时段集群负载动态变化有效整合负载, 提高了 12.2% 的平均资源利用率, 并且基于任务整合在低负载时段触发富余节点休眠降低集群能耗.

6 结语

本文针对集群负载动态变化使资源得不到充分利用的问题, 提出并设计实现了基于 CoreOS 面向负载整合的集群调度系统 LICSS, 通过节点负载度量和任务度量方法, 实现了减少实时采集数据的波动误差, 提高迁移时机的准确性, 降低负载不稳定带来的频繁迁移; 通过根据不同时段集群负载动态变化情况对负载进行有效整合, 提高了 12.2% 的平均资源利用率; 并且基于任务整合在低负载时段触发富余节点休眠降低集群能耗. 下一步将在考虑关键的二维资源指标的基础上, 把网络带宽和 IO 加入到资源指标中, 更精确的动态整合集群负载, 有效提高资源利用率, 降低能耗.

参考文献

1 What is docker. http://www.docker.com/what-docker. [2016-09-03].

2 Documents of coreOS. https://github.com/coreos. [2016-09-03].

3 Maguluri ST, Srikant R, Ying L. Heavy traffic optimal resource allocation algorithms for cloud computing clusters. Proc. of the 24th International Teletraffic Congress. Krakow, Poland. 2012. 25.

4 王克勇, 王丽, 徐靖文, 等. 绿色数据中心空调节能技术研究. 能源研究与利用, 2012, (2): 29–31.

5 Ruan SL, Lu CW, Guan YJ. Energy-saving optimization in data center based on computing resource scheduling. Information Technology and Applications: Proc. of the 2014 International Conference on Information Technology and Applications (ITA 2014). Xi'an, China. 2015. 349–353.

6 曲晓雅, 刘真. 基于阈值滑动窗口机制的虚拟机迁移判决算法. 计算机科学, 2016, 43(4): 64–69.

7 Zhu XY, Young D, Watson BJ, *et al*. 1000 Islands: Integrated capacity and workload management for the next generation data center. Proc. Fifth International Conference on Autonomic Computing. Chicago, IL, USA. 2008. 172–181.

8 Xu J, Fortes J. A multi-objective approach to virtual machine management in datacenters. Proc. of the 8th ACM International Conference on Autonomic Computing. Karlsruhe, Germany. 2011. 225–234.

9 Gmach D, Rolia J, Cherkasova L, *et al*. An integrated approach to resource pool management: Policies, efficiency and quality metrics. Proc. of IEEE International Conference on Dependable Systems and Networks With FTCS and DCC.

Anchorage, AK, USA. 2008. 326–335.
10 Sahu Y, Pateriya RK, Gupta RK. Cloud server optimization with load balancing and green computing techniques using dynamic compare and balance algorithm. Proc. of the 2013 5th International Conference on Computational Intelligence and Communication Networks. Mathura, India. 2013. 527–531.
11 Sinha R, Purohit N, Diwanji H. Energy efficient dynamic integration of thresholds for migration at cloud data centers. Special Issue of International Journal of Computer Applications on Communication and Networks. 2011. 44–49.
12 Beloglazov A, Buyya R. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. Proc. of the 8th International Workshop on Middleware for Grids, Clouds and e-Science. Bangalore, India. 2010, 4.
13 Zhu XY, Wang ZK, Singhal S. Utility-Driven workload management using nested control design. Proc. of American Control Conference. Minneapolis, MN, USA. 2006, 6.
14 Xu J, Fortes J. A multi-objective approach to virtual machine management in datacenters. Proc. of the 8th ACM International Conference on Autonomic Computing. Karlsruhe, Germany. 2011. 225–234.
15 Docker Inc. Deploy and manage any cluster manager with docker swarm. 2015. https://blog.docker.com/2015/11/deploy-manage-cluster-docker-swarm/. [2016-09-05].
16 Docker Inc. Docker swarm strategies. 2015. https://docs.docker.com/swarm/scheduler/strategy/. [2016-09-05].
17 Mouat A. Swarm v. fleet v. kubernetes v. mesos. 2015. http://radar.oreilly.com/2015/10/swarm-v-fleet-v-kubernetes-v-mesos.html. [2016-09-09].
18 Google. Kubernetes scheduler. 2015. http://kubernetes.io. [2016-09-12].
19 裴益轩, 郭民. 滑动平均法的基本原理及应用. 火炮发射与控制学报, 2001, (1): 21–23.
20 Lin JCC, Lu H. Towards an understanding of the behavioural intention to use a web site. International Journal of Information Management, 2000, 20(3): 197–208. [doi: 10.1016/S0268-4012(00)00005-0]