第47卷第8期 中国科学技术大学学报 Vol.47,No.8
2017年8月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Aug. 2017

文章编号:0253-2778(2017)8-0627-08

容器微云监控系统的设计和实现

张 松,疏官胜,李 京

(中国科学技术大学计算机科学与技术学院,安徽合肥 230027)

摘要:微云是一种网络边缘的小型数据中心,能够以较低的延迟为用户的资源密集型应用提供服务.相较于主流的虚拟机技术,容器技术以资源利用率高,接近物理机的性能,启动迅速等特点,成为构建微云的首选.微云的运行离不开监控,本文结合现有集群监控系统的成熟架构,并结合容器微云的特点,设计并实现了一套基于容器技术的微云的监控系统,实现了监控数据的收集,存储,汇总,展示,异常检测及阈值和收集周期动态设置,同时,为了降低带宽的占用,提出了请求聚合算法.通过实际的部署和测试,证明该系统能够以较低的负载满足各项设计需求.

关键词:微云;容器;微云监控;Docker

中图分类号:TP391 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2017.08.001

引用格式:张松,疏官胜,李京. 容器微云监控系统的设计和实现[J]. 中国科学技术大学学报,2017,47(8):627-634.

ZHANG Song, SHU Guansheng, LI Jing. Design and implementation of a monitoring system for container-based cloudlet[J]. Journal of University of Science and Technology of China, 2017,47(8):627-634.

Design and implementation of a monitoring system for container-based cloudlet

ZHANG Song, SHU Guansheng, LI Jing

(*School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027, *China*)

Abstract: Cloudlet, which can perform resource-intensive applications for users with low delay, is a small data center located at the edge of the Internet. Compared to virtual machine technology, container technology has become the first choice to build a cloudlet due to its higher resource use rate, physical-machine-like performance and faster boot speed. The monitoring system is indispensable to the operation of a cloudlet. Combining the maturity framework of existing monitoring systems and the characteristics of container-based cloudlet, a monitoring system was designed and implemented for container-based cloudlets. And the monitoring system realizes the collection, storage, aggregating and displaying of monitoring data, anomaly detection, and dynamic settings of thresholds and collection cycles. At the same time, a request aggregating algorithm was proposed to reduce. Actual deployment and testing indicate that the system can meet the design requirements with low load.

Key words: cloudlet; container; cloudlet monitoring; Docker

收稿日期:2016-05-13;**修回日期**:2016-05-28

作者简介:张松,男,1990年生,硕士生. 研究方向:消息中间件,监控系统. E-mail: songzhan@mail.ustc.edu.cn

通讯作者:李京,博士/教授. E-mail: lj@ustc.edu.cn

0 引言

云数据中心能够使用海量的资源帮助用户执行资源密集型的应用.谷歌眼镜、VR等设备和应用同时具备资源密集和低延迟交互两个特点.云数据中心由于广域网的限制,往往会产生较大的服务延迟,不能很好地为这些应用服务[1].微云[1-2]是一种小型的数据中心,包含几台或几十台主机,可以为用户提供一定量的计算资源,并且,由于微云位于网络的边缘,用户的移动设备通过无线网络便可以与微云通信,降低了网络的延迟,提高了带宽,非常适合为这类应用服务.

微云位于用户设备-微云-云计算中心架构的中间层,如图1所示.用户的移动设备将计算任务提交给微云,微云将为用户提供计算、存储等服务.微云的上层是云计算中心,微云不能够处理的任务,将被上传到云数据中心处理.

微云分布在不同的地理位置上,为了管理方便,地理位置上相邻的几个微云可以组成一个微云组,每个微云组内设置一个头结点,所有微云组由顶部数据中心进行管理,如图2所示.

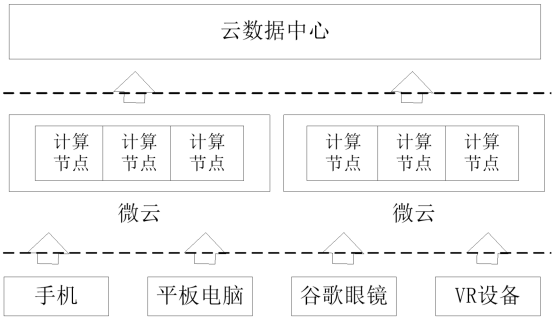

图1 用户设备-微云-云数据中心架构

Fig.1 Devices-cloudlet-cloud data center hierarchy

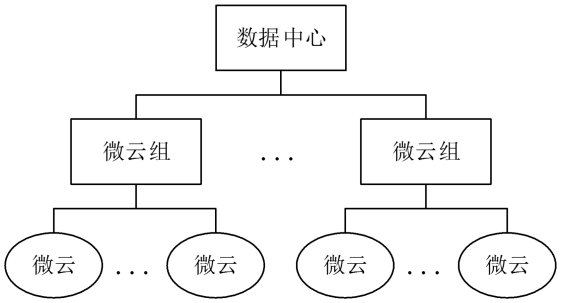

图2 微云框架

Fig.2 Cloudlet framework

云服务通常将虚拟机作为主要的服务手段,以满足资源复用和用户隔离的需求,而容器技术同样可以满足资源复用和用户隔离的需求,同时与虚拟机技术相比,有着更高的资源利用率,更快的启动速度,更接近于物理机的性能[3],Soltesz等[4]提出容器技术将取代虚拟机技术,Docker[5]的发布使容器的迁移和部署极为方便,因此,容器技术成为我们构建微云系统的首选.

监控系统对于微云是必不可少的.微云提供商为了保障微云的正常健康运行,需要监控系统进行异常的检查及告警,为了保证微云的高效运行,需要知道微云各节点及各级微云组的资源消耗,从而选择合适的节点进行服务.服务提供商为了合理分发用户的请求以及提供可靠持续的服务,也需要通过监控系统了解其服务实例的运行状况.

目前许多的监控软件如Ganglia[6],NAGIOS[7],Zabbix[8-9]等均是针对物理机集群监控设计,并没有考虑对容器的监控,因此并不能很好地监控容器微云.CAdvisor[10]是谷歌发布的一款监控Docker容器的监控软件,集收集、展示功能于一身,但是它只能监控单机上的Docker容器,不适合微云环境.DataDog[11]是一款非常成熟的监控软件,可以很好地支持多机的容器监控,但是它不是开源软件,价格昂贵.

本文的目标是设计并实现一套针对容器微云的监控系统,能够实现多种监控数据的收集,包括容器数据,计算节点数据,业务数据,各级微云组资源的运行情况数据及其存储,异常检测并报警等,同时支持检测阈值,收集周期的动态设置,且有较好的扩展性和可靠性.由于广域网的带宽比较宝贵,监控系统使用的广域网带宽受到一定的限制.

1 相关工作

1.1 监控系统的架构

监控系统一般采用树型结构,主要包括两个部分:叶子节点的监控代理和非叶子节点的监控服务器.监控代理主要完成监控数据的采集,并将监控数据发送给监控服务器,监控服务器接收监控代理采集的监控数据和下级监控服务器传输的数据,并进行处理,包括存储,异常检测等.

1.2 监控数据的收集

微云中容器数据的收集主要有3种方式:①通过Docker提供的API接口,可以获取容器的运行数据;②将容器当作一个物理机,在其中安装监控代理收集数据[12];③通过虚拟文件系统获取容器的运

行数据.

第一种方式,使用 Docker 的 API 接口时,收集的数据有限,且收集指标不支持扩展;每次调用时,都会占用大量的 CPU 资源;而且不够灵活.对于第二种方式,需要在容器内部安装代理,但用户出于数据的安全性考虑,不会允许内部安装代理,该方式只适合在私有云中实现.第三种方式可以收集到多项指标,而且 CPU 的占用率很低,因此本文采用这种方式收集容器的运行数据.

1.3　监控数据的存储

Ganglia,NAGIOS 等监控软件的工作是对监控数据进行展示和异常的检测,所以它们一般使用 RRD 数据库[13]存储数据.该数据库是一种环形的数据库,每个表的大小固定,在数据表满时,将覆盖以前的监控数据.

本文的监控系统采集到的数据,还需要供技术人员进行统计和分析,以了解微云的运行状况并对资源使用进行预测,因此 RRD 数据库并不适合.MySQL 是一款体积小、速度快的数据库,管理工作也非常成熟,很适合微云监控的场景.为了降低成本、提高数据的可靠性和满足存储的横向扩展,本文使用了分布式文件系统 GlusterFS.

1.4　监控数据的传输

监控数据一般由下级节点发送到上级节点,发送的模式包括推模式和拉模式.推模式即下级节点在获得数据后立即将数据发给上级节点,上级节点被动接收数据.拉模式是指下级节点获取数据后将其存储,一般是存在内存中,然后当上级节点请求数据时,再将这些数据发送给上级节点.

推模式的实时性较好,且能较好地保证监控代理和监控服务器之间数据的一致性,扩展性也较好,但是资源消耗较高;拉模式的资源消耗较低,但是监控代理与监控服务器之间的数据一致性较差,扩展性也稍差.

2　监控系统的设计

监控系统主要由监控代理、监控服务器两部分构成,如图 3 所示.

微云中的每台计算节点都部署了监控代理,每个微云内均部署一个监控服务器,如图 4 所示.微云组的头结点及数据中心也会部署监控服务器,而且这 3 个层次的监控服务器功能相同.

由于每个物理机上都会运行数量众多的容器,

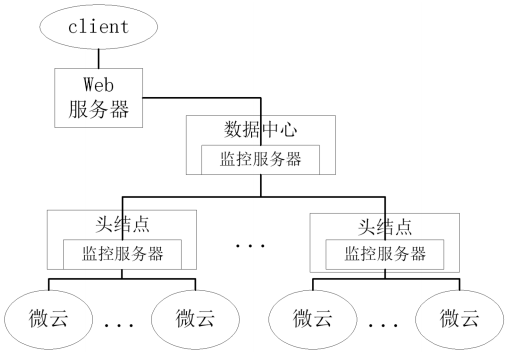

图 3　监控系统框架概要图

Fig.3　Overview of framework of monitoring system

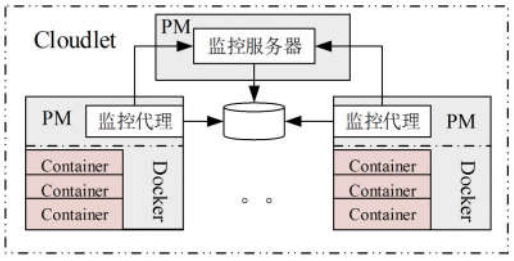

图 4　监控系统框架微云内部图

Fig.4　Framework of monitoring system within cloudlet

从而产生大量的监控数据.每个监控服务器往往会管理多个容器,为了减小监控系统中的带宽占用和监控服务器的负载,一些监控数据的处理功能被下沉到监控代理中.

2.1　监控服务器

监控服务器主要负责管理下属的各节点,接收下级节点的监控数据,并对其进行汇总、上传等,同时对用户的阈值及监控周期设置命令及数据请求命令作相应处理.

2.1.1　静态设计

如图 5 所示,监控服务器主要包括 4 个模块,传输模块负责管理监控服务器与其他节点的消息传输通道,命令传输通道和心跳等;数据处理模块,负责将接收到的监控数据进行存储和汇总,并将汇总结果上传给上层节点;命令处理模块负责接收其他节点的各种命令并进行相应的处理;拓扑管理模块,主要存储其监控服务器的下级节点的各种信息.

为了保证监控服务器的可靠性,我们采用了主备监控服务器的形式,主备之间保持数据的一致性.

2.1.2　关键机制设计

如图 6 所示,主监控服务器启动时,首先会读取配置文件,然后发送版本号为 0 的心跳.这是因为主监控服务器的启动有两种情况,一是监控系统的初始启

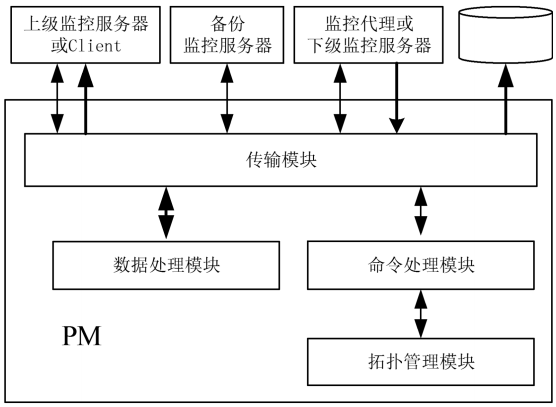

图 5 监控服务器设计

Fig.5 The design of monitoring server

动,二是主监控服务器因故障宕机然后重启,在第二种情况下,备份监控服务器可能已经切换为主监控服务器,两个主监控服务器会在监控系统内引起冲突,所以当备份监控服务器切换为主监控服务器状态时,心跳版本号加 1,因此,若存在更高版本号的心跳,说明主监控服务器已存在,该服务器自动切换为备份服务器状态.

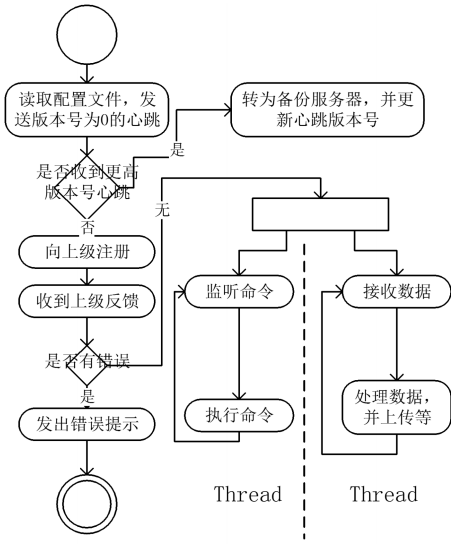

图 6 主监控服务器的启动

Fig.6 The start of primary monitoring server

服务器会向上级监控服务器注册,这里每台监控服务器都会有一个标识符,且同一个监控服务器的下属节点的标识符不能相同,因此上级的监控服务器会按标识符进行比对,若存在,则返回错误.

监控服务器启动成功,开始正常的工作状态,即一方面监听其他节点的命令,另一方面接收下级数据,并汇总上传等.

2.2 监控代理

监控代理主要负责监控数据的收集、汇总、异常检测等任务,并将汇总数据上传到监控代理节点.其内部结构如图 7 所示.

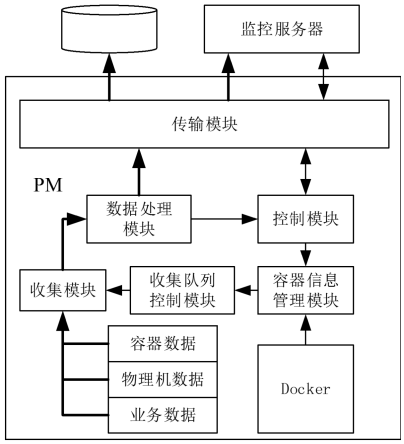

图 7 监控代理设计

Fig.7 The design of monitoring agent

2.2.1 静态设计

收集模块负责收集监控数据,包括容器数据,物理机数据及业务数据.容器的监控数据中,CPU、内存和 IO 相关信息由 Egroups[14] 产生并输出,网络相关信息在/proc 中获取,物理机数据从/proc 文件中获取.业务数据由用户获取并按规定格式发送到指定的端口上.

收集队列控制模块负责向收集模块提供待收集信息的容器列表,其中物理机被看作一个特殊的容器存于队列中.容器由于启动时间及监控周期各不相同,因此该模块将根据容器的上一次收集时间及监控周期选出此刻需要收集的容器.

容器信息管理模块主要通过 Docker 的接口维护目前运行的容器列表.包括启动时获取的容器列表,运行中监听容器的启动和死亡事件,并获取容器的镜像信息等.

数据处理模块负责对收集模块收集到的数据进行异常检测和汇总操作,并监控信息存储.

控制模块负责处理上级监控服务器下发的各种命令,包括监控周期的设置和阈值的设置等,并进行注册等操作.

2.2.2 关键机制设计

收集队列控制模块负责队列的控制,该模块维护一个按下次收集时间排序的容器队列,如算法 2.1 所示.

算法 2.1　队列控制算法

```
Input: currentTime
Output: list
function: listControl
  p ← conList
   while p ! = NULL and p. nextCollecTime < currentTime do
      list.insert(p)
      p ← conList.eraser(p)
    end while
    p ← list
    while p ! = NULL do
        p.nextCollectTime ← p.nextCollectTime + p.cycle
        conList.insert(p)
      p ← p + 1
    end while
    return list
end function
```

从队列中取出待收集的容器，然后再根据其收集时间和周期确定下次收集时间，重新插入原队列.

2.3　节点间通信

监控代理收集监控数据后，监控数据被存储进数据库，汇总数据被发送到监控代理.监控代理收到监控数据后，将汇总数据存储进数据库，并将再次汇总的数据发往上层的监控服务器.

2.4　请求聚合算法

在微云集群系统中，带宽资源非常宝贵，因此，往往会限制监控系统的带宽占用.为了降低带宽，容器的原始监控数据都存储在微云本地的存储系统中，只上传一些汇总数据等，但是仍会有各原始监控数据的请求，其中，①会存在多个连接请求同一数据的情况；②数据的传输过程中，若每个被请求数据都用一个独立的消息进行传输，大量的消息头将占据不少的带宽.为了降低带宽占用，针对这两个问题，本文设计并实现了请求聚合算法.

2.4.1　算法介绍

该算法的输入为用户的数据请求，且每个请求只包含一个数据，输出为相应的监控数据.算法分为如下几个部分：请求的注册(含合并)、请求的撤销、请求的分发、请求数据的回传以及请求数据的分发.

请求的数据种类包括容器数据，业务数据，物理机数据，微云组运行数据.我们以请求容器数据为例进行介绍.

2.4.1.1　数据结构

(Ⅰ) 请求注册表

请求的注册在 web 服务器完成，web 服务器内部维护请求注册表，如图 8 所示.表中的每个元素表示一个容器请求 req，如下所示：

```
typedef struct req{
    string          containerID;
    string          machineID;
    list            connList;
    struct req *    nextReq;
};
```

其中，containerID 表示容器的 ID，machineID 表示该容器所属物理机 ID，connList 表示请求该容器数据的连接，nextReq 指向请求注册表中下一个请求.

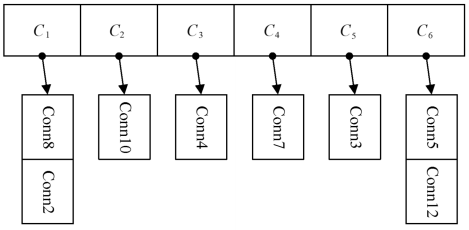

图 8　请求注册表

Fig.8　Request registry

该表是一个有序表.若请求 $c_1 < c_2$，则 c_1.machineID$<c_2$.machineID，或者 c_1.machineID$==c_2$.machineID and c_1.containerID$<c_2$.containerID.

(Ⅱ) machineID

每个监控代理和监控服务器所在物理机均有一个标识符，该标识符为长度为 2 的字符串.每个物理机均有一个 machineID，类型为字符串，该值为上级节点的 ID 值加上本机的标识符，若某个监控服务器无上级节点，则其物理机 ID 值为其标识符.

(Ⅲ) 请求表

每个监控服务器和监控代理中维持一个请求表，用以记录其该对哪些数据进行收集，结构如下：

```
typedef struct req{
    string          containerID;
    string          machineID;
    struct req *    nextReq;
};
```

其数据结构与请求注册表相似，只是没有连接表.

2.4.1.2 请求的注册

本过程发生在web服务器端,如算法2.2所示.

算法2.2 请求注册

```
Input: connReq
Output: none
function reuqestRegist
1.  q ← List
2.  while q ! = NULL do
3.      if compare(connReq,q)>0 then
4.        q ← q + 1
5.      else if compare(conn,q) == 0 then
6.        q.conn.add(connReq)
7.        break
8.      end if
9.  end while
10.  if q == NULL do
11.    q ← List.insert(q,connReq)
12.    q.conn.add(connReq)
13.  end if
end function
```

算法将连接的请求与请求注册表进行比较,若该表中已存在该请求,则将该连接插入请求的连接表中,否则,将该请求插入请求注册表中.

2.4.1.3 请求的撤销

当连接停止请求数据后,连接将会从web服务器的请求表中删除.其中,若某个请求的提出者只有该连接,该请求将从请求表中删除,否则,将该连接从请求的连接表中删除即可.如算法2.3所示.

算法2.3 请求的撤销

```
Input: req
Output: none
function requestDel
1.  q ← List
2.  while q ! = NULL do
3.    if compare(req,q) > 0 then
4.      q ← q + 1
5.      continue
6.    else if compare(req,q) == 0 then
7.      delete user from q.list
8.      if q.userList.empty() then
9.        q ← erase(q)
10.       break
11.     end if
12.   end if
13.   end while
end function
```

2.4.1.4 请求的分发

Web服务器端在完成请求的注册时,会将新的请求发往监控服务器.

监控服务器收到web服务器或上级监控服务器发来的请求后,会将请求的machineID与自己的ID进行比较.①若长度相等,且字符串相同,则表明是发送给自己的,将该请求插入请求表;②若请求的machineID字符串长度大于自己的machineID长度,且相匹配,则根据请求的machineID发给相应的子节点,并将该请求插入请求表;③否则,丢弃该请求.

2.4.1.5 请求数据的回传

请求的目的节点负责收集数据并上传给上层的监控服务器.这是一个生产者-消费者模型,其中有一个线程负责数据的收集,并将数据放进一个有序的无锁队列,是生产者,另一个线程负责周期性地从队列中取数据并发送,是消费者,如图9所示.消费者线程周期性地轮询队列,若队列不空,将所有数据放进一个消息中发送.

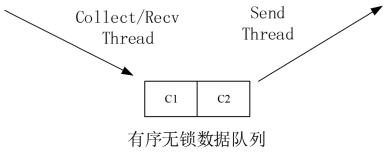

图9 监控数据收发模型

Fig.9 The model of monitoring data sending and recieving

目的节点的上层节点与目的节点的流程大致相同,只是数据是由下级节点发送,而非自己收集.其在接收到数据后,将消息拆包,并将数据放进有序无锁队列.

2.4.1.6 请求数据的分发

数据经过层层上传最终到达web服务器端,此时需要将收集到的请求数据发送给相应的请求方.对于收到的请求数据,将其与请求注册表比较.

若找到相匹配的请求,则将该数据发给请求该数据的所有连接;若某些请求在请求注册表中找不到相应的元素,则表明该请求已结束,直接丢弃该消息.

2.4.2 算法评价

该算法能有效减少监控系统的带宽占用,但是增加了web服务器和各级监控服务器的负载.由于算法的流程比较简单,增加的负载有限,而减少带宽的效果非常显著.权衡得失,我们认为该算法的应用

是可行和必要的.

3　系统部署和测试

目前我们已经完成了该系统的实现,为了测试软件设计的正确性和可靠性,我们使用数台物理机,模拟微云的运行环境,部署了该监控系统.

3.1　部署环境

我们使用了两种配置不同的物理机.其中 *A* 型物理机 5 台,*B* 型物理机 6 台,其各项配置参数如表 1 所示.

表 1　部署环境

Tab.1　Deployment environment

	配置项	参数
A 型物理机	操作系统	Ubuntu 14.04 server
	CPU	8 核,Intel(R) Xeon(R) CPU E5410 @ 2.33GHz
	内存	12GB
B 型物理机	操作系统	Ubuntu 14.04 server
	CPU	20 核,Inter (R) Xeon (R) CPU E5-2660 v3 @ 2.60GHz
	内存	120GB

3.2　监控系统的部署

我们部署了 4 个微云,每个微云包含两个计算节点,且每两个微云构成一个一级微云组;两个一级微云组构成一个二级微云组.系统已正常运行 7 天.

3.3　资源开销

监控系统的资源开销是评价监控系统非常重要的一个指标.监控系统中监控代理要监控数以百计的容器,这是资源开销最大的部分,因此我们主要针对监控代理的性能开销作了测试.

作为对比,我们部署了另一种数据采集方案,即在每个容器内安装监控代理的方式收集数据.两种方式收集的数据种类、周期均相同,设为 10 s.系统消耗如图 10 和表 2 所示.

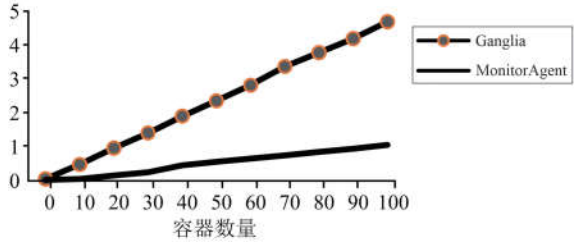

图 10　CPU 占用

Fig.10　CPU utilization

表 2　内存消耗

Tab.2　Memory occupancy

容器数量	0	30	60	90
Ganglia/kB	6 320	59 061	128 333	198 197
Monitor-Agent/kB	12 130	12 181	12 241	12 349

由图 10 和表 2 可以看到,随着容器数量的增长,CPU 占用率和内存占用都随之增长,Ganglia 的 CPU 占用和内存占用都远超监控代理.由此可知,该监控系统的系统开销较小,适合于微云的监控.

3.4　带宽占用测试

微云中的带宽非常宝贵,监控系统的带宽受限,为了减少带宽的占用,我们使用了请求聚合算法,下面通过模拟容器请求来测试该算法的效果.实验中,我们在每台计算节点上运行 200 个容器,8 台计算节点共 1 600 个容器,然后通过随机生成若干请求(20～400),并记录数据中心处监控服务器的带宽,然后将该带宽与未使用请求聚合算法比较,测试带宽节省效果,如图 11 所示.

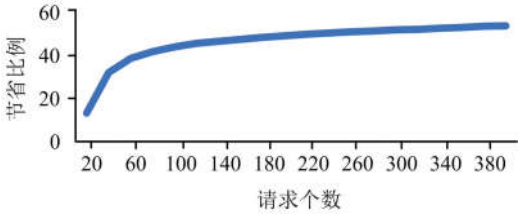

图 11　带宽节省比例

Fig.11　The ratio of bandwidth saved

请求数量在 0～60 时,节省比例快速增长.是因为此阶段消息头部个数与容器数据个数的比例急剧下降,而且由于每个容器数据的大小较小,与消息头部相当,因此随着消息头部比例的快速减少,节省比例快速下降.

当请求数量大于 60 时,消息头部个数与请求数据个数的比例已经较低,带来的变化不明显.与此同时,请求数量越多,重复请求的比例也越大,但由于请求的合并,传输的请求数据相应减少,虽然还会继续增长,但是重复的比例并没有快速增长,因此节省比例增长缓慢.

从测试结果看,算法的效果还是非常明显的.

4　结论

本文通过分析容器微云的特点,并结合现有监控系统的成熟架构,设计并实现了一套基于容器的微云的监控系统.系统实现了设计的各种需求,包括

容器数据,物理机数据和业务数据的收集,汇总,和展示等.其中,①针对容器微云的特点,即微云中每台主机都运行大量的容器,产生大量的监控数据,我们采用了“胖”监控代理-“瘦”监控服务器的设计,将监控服务器的业务负载下沉,减轻其负担;②针对微云监控系统的带宽限制,我们设计并实现了请求聚合算法,大大减少了监控系统的带宽占用;③高扩展性和可靠性.对于高扩展性,我们采用了下级节点向上级节点注册的机制,对于高可靠性,使用了主备监控服务器技术.

参考文献(References)

[1] SATYANARAYANAN M, CHEN Z, HA K, et al. Cloudlets: At the leading edge of mobile-cloud convergence[C]// 6th International Conference on Mobile Computing, Applications and Services. Austin, USA: IEEE Press, 2015:1-9.

[2] WIKIPEDIA. Cloudlet[EB/OL]. https://en.wikipedia.org/wiki/Cloudlet.

[3] FELTER W, FERREIRA A, RAJAMONY R, et al. An updated performance comparison of virtual machines and Linux containers [C]// IEEE International Symposium on Performance Analysis of Systems and Software. Philadelphia, USA: IEEE Press, 2015:171-172.

[4] SOLTESZ S, PÖTZL H, FIUCZYNSKI M E, et al. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors [J]. ACM SIGOPS Operating Systems Review, 2007, 41(3): 275-287.

[5] DOCKER. A better way to build Apps[EB/OL]. https://www.docker.com/[2016-05-02].

[6] MASSIE M L, CHUN B N, CULLER D E. The ganglia distributed monitoring system: design, implementation, and experience [J]. Parallel Computing, 2004, 30(7):817-840.

[7] NAGIOS. The industry standard in IT infrastructure monitoring[EB/OL]. https://www.nagios.org/[2016-05-02].

[8] ZABBIX. The enterprise-class monitoring solution for everyone[EO/BL]. http://www.zabbix.com/[2016-05-02].

[9] 吴兆松. Zabbix企业级分布式监控系统[M]. 北京:电子工业出版社, 2014.

[10] CADVISOR. Analyzes resource usage and performance characteristics of running containers [EB/OL]. https://github.com/google/cadvisor[2016-05-02].

[11] DATADOG. Cloud monitoring as a service[EB/OL]. https://www.datadoghq.com/[2016-05-02].

[12] INFOQ. 腾讯游戏是如何使用Docker的[EB/OL]. http://www.infoq.com/cn/articles/how-tencent-game-use-docker[2016-05-02].

[13] RRDTOOL. About RRDTool[EB/OL]. http://oss.oetiker.ch/rrdtool[2016-05-02].

[14] WIKIPEADIA. Cgroups [EB/OL]. https://en.wikipedia.org/wiki/Cgroups[2016-05-02].