文章编号:0253-2778(2019)10-0820-08

# 微云环境下请求分发和容器部署成本优化

郑晓杰,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**:随着物联网(IoT)的发展,微云(cloudlet)正在为更多低延迟、高带宽要求的应用进行服务.物联网应用请求量在时间和空间分布极其不均,如果微云仅处理周围请求,会导致一部分微云过载时,另一部分微云负载不足.此外,物联网应用的重要程度各不相同,请求量大的服务可能会抢占微云资源,使关键服务得不到执行.微云负载不均和关键服务无法执行会使云基础设施提供商成本激增.于是基于容器的微云资源分配模型,提出请求分发和容器部署成本优化问题,最优化云基础设施提供商的成本.在此基础上,提出了成本优化贪心算法(CO-Greedy),该算法能够寻找合理的请求分发和容器部署方案.实验结果表明,该算法在不同场景下表现都优于已有算法.

**关键词**:微云,容器,资源分配

**中图分类号**:TP391 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2019.10.007

**引用格式**:郑晓杰,李京. 微云环境下请求分发和容器部署成本优化[J]. 中国科学技术大学学报,2019,49(10):820-827.

ZHENG Xiaojie, LI Jing. Cost optimization of request dispatching and container deployment in cloudlets[J]. Journal of University of Science and Technology of China, 2019,49(10):820-827.

# Cost optimization of request dispatching and container deployment in cloudlets

ZHENG Xiaojie[1], LI Jing[1]

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

**Abstract**: With the development of the Internet of Things (IoT), cloudlet is serving more low-latency, high-bandwidth applications. The application request is dynamic in time and space. If a cloudlet only processes the surrounding requests, some cloudlets will be overloaded while others are underloaded. In addition, IoT applications vary in importance. Some unimportant services with large requests may preempt the cloudlet resources, resulting in critical services unable to be executed. Cloudlet load imbalance and critical service starvation will increase the cost of the cloud infrastructure provider. Cost optimization of request dispatching and container deployment in container-based cloudlets are investigated and a cost optimization greedy algorithm named CO-Greedy is proposed which optimizes cost by dispatching the request to the surrounding cloudlet. The experimental results show that the algorithm has better performance in all scenarios.

**Key words**: cloudlet; container; resource allocation

**收稿日期**:2019-03-28;**修回日期**:2019-05-28

**作者简介**:郑晓杰,男,1994年生,硕士生,研究方向:移动云计算,边缘计算. E-mail:stanleye@mail.ustc.edu.cn

**通讯作者**:李京,博士/教授. E-mail:lj@ustc.edu.cn

## 0　引言

随着生活水平提高，人们对应用服务质量(QoS)的要求日益增加，低延迟应用(如增强现实，自动驾驶等)数量快速增长. 传统移动云计算(MCC)模型因为数据中心远离用户，所以延迟高、带宽有限，往往不能满足低延迟应用的需求，因此 Satyanarayanan 等提出微云(cloudlet)[1]的概念，通过在基站或无线访问接入点(AP)附近部署小型数据中心，给用户带来低延迟和高带宽的体验. 一个城市中可以部署多个微云，相互连通协作[2]，为周围用户服务.

物联网应用包含若干服务进程，应用服务提供商通过在微云物理机或者虚拟机上部署这些服务进程，为移动用户提供服务. 随着容器技术的发展，通过容器部署服务也成为一种选择[3-5]. 物联网应用使用场景不同，因此对服务质量要求也不同. 应用服务提供商通过和云基础设施提供商签订服务水平协议(SLA)，规定应用对服务质量要求.

对云基础设施提供商而言，保证应用服务质量并节约成本是其追求的目标，而实现这个目标必须同时考虑请求分发和容器部署. 容器部署个数决定云基础设施提供商的资源消耗成本，容器部署位置以及请求分发则决定请求的响应时间，进而影响应用服务质量. 此外，在微云环境下，每个区域产生的请求数量各不相同，各个微云负载也不同，通过合理请求分发可以增加微云整体的利用率.

为了最优化云基础设施提供商的成本，研究在微云场景下提出关注成本的负载分发和服务放置联合优化问题，并通过启发式贪心算法解决该问题[6]. 研究则在满足应用延迟要求前提下，提出虚拟机放置和负载分发的联合优化问题，最小化硬件的消耗[7]. 以上研究并不关注应用的重要程度，所有应用的重要程度一致，而在实际情况中，不同应用的重要程度有很大差距. 单位 SLA 违反惩罚是指在应用无法达到服务质量要求时，服务提供商向云基础设施提供商收取的相应罚款，可以评估应用的重要程度. 如果不考虑应用的重要程度，在微云过载时，低 SLA 违反惩罚的服务可能会抢占微云资源，致使关键服务无法执行，产生较高的 SLA 违反惩罚.

与已有工作不同，本文在考虑 SLA 违反惩罚的情况下，提出了微云环境下的请求分发和容器部署成本优化问题，并提出优化成本的贪心算法(CO-Greedy). 在应用延迟要求、单位 SLA 违反惩罚互异的情况下，CO-Greedy 算法会优先部署重要程度更高的服务容器，并在微云资源使用量超过预设阈值时，能够将获取增益最小的容器迁移，降低云基础实施提供商的整体成本. 最后通过模拟实验，将 CO-Greedy 与已有算法对比，证明算法的有效性.

## 1　相关工作

在公有云领域，已有优化成本的资源分配研究[8-9]. 公有云和微云有很大的区别，微云延迟低、资源有限，必须要考虑过载的情况，因此这些工作并不能迁移到微云之中.

Zhang 等[10]提出在 SLA 约束下，通过树形回填机制，同时解决负载分发和资源分配问题；但是该研究仅仅考虑单微云场景下的资源分配问题，没有考虑多微云之间的互联.

在多微云场景下，Yousefpour 等[11]提出 QoS 约束下的动态服务提供(QDFSP)问题，通过 minVoil，minCost 两种启发式贪心算法，分别最优化 SLA 违反率和云基础设施提供商成本. 该工作仅仅在负载分发已知的情况下进行研究.

Yang 等[6]第一个在微云场景下提出了负载分发和服务放置联合优化问题，将问题形式化为基础服务放置问题(BSPP)和成本感知的服务放置问题(CSPP)，分别优化延迟以及成本. 该研究中所有应用是等价的，没有考虑不同的应用会有不同 SLA.

Wang 等[7]综合负载分发和虚拟机放置问题，最小化资源消耗成本，并将问题转化为混合整数线性规划(MILP)问题. 该研究没有考虑应用的重要程度. 此外，该研究仅仅形式化问题，没有给出多项式算法. 因为容器资源占有少，每个微云能部署容器的数量远超虚拟机，搜索算法并不适用于基于容器的资源分配模型.

## 2　系统结构与模型

在请求量变化大，资源分配变化频繁的多微云场景下，相比较于虚拟机，轻量级且有更好水平弹性的容器更加适合微云系统，因此本文使用容器作为微云的底层技术.

如图 1 所示，在该模型中存在 3 个不同角色：云基础设施提供商(CIP)是微云和数据中心的资源拥有者，提供应用执行的硬件资源；服务提供商(SP)是应用的开发者，购买云基础设施提供商的硬件资

源部署应用服务，并为用户提供服务. 服务提供商在购买资源之后，需要向微云管理节点上传应用的服务容器镜像，微云管理节点收到镜像后，向各个微云节点分发镜像，从而为用户提供高质量的服务；移动用户(MU)则使用物联网设备不断向微云节点发送请求，获取相应的计算服务. 用户请求首先会发送到所在区域的微云节点，也能通过网络，在相互连接的微云和数据中心之间转发. 本章之后部分将从微云拓扑、服务、用户 3 个方面形式化微云资源分配模型.

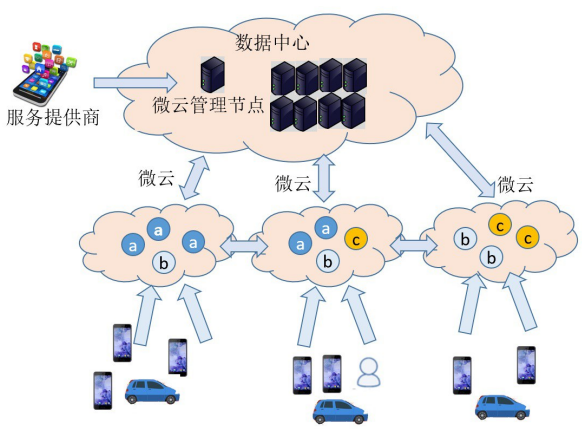


**图 1 基于容器技术的微云系统**

**Fig. 1 Container-based cloudlet system**

**微云拓扑模型** 我们使用 $G$ 表示微云拓扑. 假设微云总共有 $N$ 个节点和一个数据中心，我们使用 $j\in[1,N]$ 表示 $N$ 个微云节点，$j=N+1$ 则表示数据中心. 微云是一个资源的集合，我们使用 $R_j^m$，$R_j^c$，$R_j^s$ 代表微云 $j$ 拥有内存、CPU、存储资源量，其中 $R_{N+1}^m$，$R_{N+1}^c$，$R_{N+1}^s$ 表示数据中心的资源量，其资源规模远大于微云. 我们使用 $L_j^m$，$L_j^c$，$L_j^s$ 分别表示每个微云的内存、CPU、存储的单位成本，数据中心单位资源成本比微云单位资源成本更加低廉. 微云相互连通，即任意两个微云之间存在一条边，每条边具有两个属性，$d_{ij}$ 表示微云 $i$ 到微云 $j$ 的延迟，$r_{ij}$ 表示微云 $i$ 到微云 $j$ 的带宽. 微云和数据中心通过部署容器提供计算资源，$\alpha_{jk}$ 表示在微云 $j$ 上部署服务 $k$ 的容器个数.

**服务模型** 我们假设在微云上总共注册 $M$ 个应用，每个应用对应一个服务，$k\in[1,M]$ 表示各个服务. 每个服务消耗的资源各不相同，内存消耗、CPU 消耗、存储消耗分别使用 $U_k^m$，$U_k^c$，$U_k^s$ 表示. 每个服务容器能够处理的请求有限，使用 $\pi_k$ 表示. 服务提供商向基础设施提供商注册应用时，会提供一组服务质量要求 $(q_k,\mathrm{th}_k,p_k)$，保证服务良好运行，其中 $q_k$ 表示服务 $k$ 的可用性，$\mathrm{th}_k$ 则表示服务 $k$ 的时间阈值，$p_k$ 代表单位 SLA 违反惩罚. 例如对于给定的服务 $k$，服务提供商可以设定 $q_k=95\%$，$\mathrm{th}_k=10$ ms，$p_k=100$，表示服务 $k$ 响应时间小于 10 ms 的请求数要在 95%之上，如果没有达到该要求，每百分之一需要支付罚款 100.

**用户和请求模型** 将用户按照距离最近的微云，划分为若干个区域，$N$ 个微云对应 $N$ 个区域，即 $i\in[1,N]$. 假设区域内物联网设备到微云的延迟是一个固定值 $d_i'$，带宽是 $r_i'$. 用户通过发送请求获取云端资源，每个请求可以用三元组 $(x_k,y_k,w_k)$ 表示，$x_i$ 表示上传数据量，$y_i$ 代表下载数据量，$w_i$ 表示需要的执行时间. $\lambda_i^k$ 表示在 $i$ 区域内产生服务 $k$ 请求的个数，请求个数通过网络设备获取. 在请求发送到最近微云之后，有一定概率向各个微云转发，$\beta_{ijk}$ 为在区域 $i$ 发出服务 $k$ 的请求，最终在微云 $j$ 上处理的概率.

## 3 问题建模与算法

### 3.1 请求分发和容器部署成本优化问题定义

给定微云的拓扑，服务参数，以及每个区域的请求分布，求微云容器部署方案 $\alpha$ 以及请求分发概率 $\beta$，最后能够最小化云基础实施提供商的成本. 微云环境下请求分发和容器部署成本优化问题可以被描述为 $P1$.

优化目标：

$$P1:\ \text{minimize}\ \mathrm{Cost}_{\mathrm{res}}+\mathrm{Cost}_{\mathrm{sla}} \tag{1}$$

约束条件：

$$\sum_{k=1}^{M}\alpha_{jk}*U_k^c\leqslant R_j^c,\ \forall j\in[1,N+1] \tag{2}$$

$$\sum_{k=1}^{M}\alpha_{jk}*U_k^m\leqslant R_j^m,\ \forall j\in[1,N+1] \tag{3}$$

$$\sum_{k=1}^{M}U_k^s\leqslant R_j^s,\ \forall j\in[1,N+1] \tag{4}$$

$$\sum_{j=1}^{N+1}\beta_{ijk}=1,\ \forall i\in[1,N],\forall k\in[1,M] \tag{5}$$

$$\sum_{i=1}^{M}\beta_{ijk}*\lambda_{ik}\leqslant\alpha_{jk}*\pi_k,\ \forall j,\forall k \tag{6}$$

$$\alpha_{jk}\in\mathbb{N},\ \forall j,\forall k \tag{7}$$

$$0\leqslant\beta_{ijk}\leqslant 1,\ \forall i,\forall j,\ \forall k \tag{8}$$

公式(1)描述问题的优化目标，其中 $\mathrm{Cost}_{\mathrm{res}}$ 表示资源成本，$\mathrm{Cost}_{\mathrm{sla}}$ 则表示 SLA 违反惩罚. 成本的详细计算公式在下文描述. 公式(2)～(8)描述问题

的约束条件.公式(2)～(4)表示容器的资源约束,其中公式(2)表示 CPU 资源的限制,公式(3)表示内存资源的限制,公式(4)表示对存储资源的限制.公式(5)～(8)则描述对容器个数和分发目标的约束,其中公式(5)表示从任意区域发出的任意服务请求,向各个微云转发的概率之和为 1.公式(6)则描述在向微云转发服务的请求数小于它所能处理的请求总和,公式(7)～(8)则描述了 $\alpha,\beta$ 的取值范围.

### 3.2　成本的计算

在请求分发与容器部署方案已知的情况下,资源消耗成本和 SLA 违反惩罚相应的计算公式如下.

**资源消耗成本**　资源消耗成本等于 CPU、内存、存储成本之和.

$$\mathrm{Cost}_{\mathrm{res}}=\sum_{j=1}^{N+1}\sum_{k=1}^{M}(U_k^c * L_j^c * \alpha_{jk}+U_k^m * L_j^m * \alpha_{jk}+U_k^s * L_j^s) \tag{9}$$

**SLA 违反惩罚**　计算 SLA 违反惩罚首先要计算响应时间.响应时间由 3 个部分组成:传输时间、执行时间以及延迟,如公式(10)所示. $T_{ijk}^{\mathrm{trans}}$ 代表服务 $k$ 的请求从区域 $i$ 到微云 $j$ 传输数据时间,它等于应用上传以及下载数据总和与请求到微云之间最小带宽的比值. $T_k^{\mathrm{exec}}$ 代表请求的执行时间,即请求到达容器到离开容器的时间间隔 $w_k$. $T_{ij}^{\mathrm{delay}}$ 表示请求从区域 $i$ 到微云 $j$ 的延迟,它等于区域 $i$ 的请求到最近微云延迟与微云 $i$ 到微云 $j$ 延迟的总和.

$$T_{ijk}^{\mathrm{response}}=T_{ijk}^{\mathrm{trans}}+T_k^{\mathrm{exec}}+T_{ij}^{\mathrm{delay}}=\frac{x_k+y_k}{\min\{r'_i,r_{ij}\}}+w_k+d'_i+d_{ij} \tag{10}$$

服务 $k$ 的违反率 $P_k^{\mathrm{viol}}$ 则等于能满足服务延迟要求的请求数与请求总数的比值.

$$P_k^{\mathrm{viol}}=\frac{\sum_{i=1}^{N}\sum_{j=1}^{N+1}\theta_{ijk} * \lambda_{ik} * \beta_{ijk}}{\sum_{i=1}^{N}\lambda_{ik}} \tag{11}$$

式中,$\theta_{ijk}$ 表示在服务 $k$ 的请求在区域 $i$ 发出并在微云 $j$ 接受处理是否满足应用的延迟要求,即

$$\theta_{ijk}=\begin{cases}0, & T_{ijk}^{\mathrm{response}}<\mathrm{th}_k\\ 1, & \text{其他情况}\end{cases} \tag{12}$$

总 SLA 违反成本通过累加每个服务的 SLA 违反惩罚得到.

$$\mathrm{Cost}_{\mathrm{sla}}=\sum_{k=1}^{M}\max\{(0,q_k-P_k^{\mathrm{voil}}) * p_k \tag{13}$$

### 3.3　CO-Greedy 算法

在本节中,我们提出成本优化的贪心算法 CO-Greedy,解决微云环境下请求分发和容器部署成本优化问题. CO-Greedy 算法主要思想有两点,第一是通过循环部署成本降低最多的容器,第二则是在微云部署容器时如果超过资源阈值,则迁移微云中收益最小的容器,到符合其服务质量要求并且有足够资源的微云中.该资源阈值由云基础设施提供商根据微云系统整体运行状态预先设定.

通过比较部署容器之后的总成本,高 SLA 违反惩罚的服务优先级相对较高,从而保证关键服务得到执行.而通过将容器从资源使用量高的微云迁移到资源使用量较低的微云,能够使更多的服务能够满足其服务质量的要求,降低 SLA 违反惩罚.具体算法如算法 3.1 所示.

**算法 3.1　CO-Greedy**

输入:微云拓扑 G,资源相关参数,服务相关参数,请求分布 $\lambda_{ik}$.

输出:微云的容器部署情况 $\alpha_{jk}$ 和微云的负载分发情况 $\beta_{ijk}$

```
1 L[j]表示微云 j 部署的容器集合;
2 While 请求分配未完成且微云资源过剩 do
3   j,k ← find_min_cost_deployment();
4   deployment(j,j,k);
5   While 微云 j 上资源到达阈值 do
6     c,dst←find_min_cost_move(j);
7     If c=null then
8       break;
9     else
10        undeployment(j,c);
11        deployment(c.region,dst,c.service);
12    end
13  end
14 end
15 统计每个微云处理的请求个数并计算 β;
16 return α,β
```

该算法首先通过 find_min_cost_deployment()寻找部署之后成本下降最多的服务与微云组合(3),其过程在算法 3.2 中描述.在找到相应的服务以及微云之后,需要在该微云上为服务分配相应的资源(4),部署操作如算法 3.3 所示.部署完成之后,检查该微云资源是否到达预设阈值(5),如果未到达阈值,则不断重复部署过程.而当微云资源到达阈值时,需要将部署之后成本降低最少的容器迁移.首先通过 find_min_cost_move(j)寻找最优迁移方案(6),过程在算法 3.4 中描述.如果不存在迁移方案,则退出循环(7～9).在找到迁移的容器以及微云之

后,迁移该容器到目标微云之中,即卸载原微云中的容器(10)并部署到目标微云(11).卸载操作和部署操作相反,也需要更新相应的参数.在请求分配完成或者微云资源不足以启动新的容器时,循环终止.最后通过统计每个容器服务的请求个数,计算最终的请求分发参数 $\beta$,并且返回 $\alpha$ 和 $\beta$(15~16).

寻找部署之后成本下降最多的服务与微云组合如算法 3.2 所示.该算法使用二重循环不断枚举服务和微云(2),并判断微云的剩余资源以及延迟是否能够部署该服务(3).如果剩余资源与延迟能够满足,则计算部署之后的总体成本(4),成本计算公式在 3.2 节中描述.通过不断更新最优成本 minCost(5~7),寻找最优服务和微云组合并且返回(10).

**算法 3.2 find_min_cost_deployment**

输入:剩余资源,未分配请求.

输出:部署之后成本降低最多服务与微云.

```
1 minCost←∞;
2 For j in cloudlets,k in services do
3   if can Deployment(j,k) then
4     t←服务 k 部署在微云 j 之后总成本;
5     if min Cost≥t then
6       minCost←t; minj←j; mink←k;
7     end
8   end
9 end
10 return (minj,mink);
```

deployment 函数如算法 3.3 所示.该函数的输入为区域 $i$,微云 $j$ 以及服务类型 $k$ 三个参数,即将在微云 $j$ 上部署服务 $k$ 的容器,并为区域 $i$ 中的请求提供服务.deployment 函数有以下操作,首先将更新参数 $\alpha$,并在 $L[j]$ 集合中新增该容器的实例,容器实例需要记录该容器的服务类型、处理请求的个数、请求来源地等信息.在此之后,区域 $i$ 中的请求个数,微云 $j$ 剩余资源需要相应的减少(3~4).

**算法 3.3 deployment**

输入:区域 i,微云 j,服务类型 k

```
1 α_jk++;
2 L[k]中增加该容器实例;
3 更新请求分布;
4 更新剩余资源;
```

寻找最优迁移方案如算法 3.4 所示,算法的输入是需要迁出容器的微云.首先枚举此微云上运行的容器(2),针对每一个容器计算释放该容器之后的总成本(3),成本的计算与之前相同.再遍历所有微云(5),计算枚举微云 $j$ 的资源使用率(6),此处的资源使用率使用内存、CPU 以及存储资源中使用率最高的一项表示.在此之后判断微云 $j$ 剩余资源以及延迟是否能够满足容器 $c$ 的要求(7),如果满足则通过比较成本与资源使用率,寻找微云 $j$ 中成本降低最少的容器以及能够运行该容器并且资源使用量最少的微云(8~18).

**算法 3.4 find_min_cost_move**

输入:微云 src

输出:容器 c,目标微云 dst

```
1 minCost←∞; container←null;
2 for c in L[src] do
3   t←释放容器 c 之后总体成本;
4   minUseRate←∞;
5   for j in cloudlets do
6     useRate←微云 j 的资源使用率;
7     if can Deployment(j,c) then
8       if minCost>t then
9         minCost←t;
10          minUseRate←useRate;
11          container←c;dst←j;
12        end
13        if minCost=t&use Rate<minUseRate then
14          minUseRate←useRate;
15          dst←j;
16        end
17      end
18  end
19 end
20 return container,dst;
```

### 3.4 复杂度分析

CO-Greedy 算法的复杂度分析如下:假设微云需要部署 $Q$ 个容器,则平均每个微云需要部署 $Q/N$ 个容器.算法 3.2 二重循环的复杂度为 $O(NM)$,计算成本的复杂度可以通过记录当前已使用的资源量降低为 $O(1)$,因此算法 3.2 的复杂度为 $O(NM)$.部署操作仅仅更新资源因此算法 3.3 的复杂度为 $O(1)$.算法 3.4 中枚举容器的复杂度为 $O(Q/N)$,枚举微云的复杂度为 $O(N)$因此算法 3.4 的复杂为 $O(Q)$.在算法 3.1 中,循环(5)~(14)仅在负载超过阈值时出现,一般仅会运行常数次,循环(2)~(16)则会运行 $Q$ 次,所以算法 CO-Greedy 的复杂度为 $O(Q^2+QNM)$.

## 4 实验分析

本节通过模拟实验评估算法 CO-Greedy 的有

效性. 我们将比较 CO-Greedy 算法与 cloud, first-fit, minCost 以及 OPT 算法.

cloud:将所有的请求分发到云中.

first-fit:请求向能够满足延迟要求的微云分发,微云为先来到的请求启动容器,并且为之服务.

minCost:由 Yousefpour 等[11]提出,在请求分发已经确定时,在尽量满足应用服务质量要求的前提下,最小化云基础设施提供商的成本.

OPT 算法:通过搜索寻找最优解,首先对各个微云中不同种类的服务容器个数进行枚举,并由此计算资源消耗成本. 再通过最大流算法计算能够满足延迟要求的请求总数,并且计算 SLA 违反惩罚. 因为复杂度过于高,该算法仅能在小数据规模下求解.

### 4.1　实验参数配置

我们使用 java 程序模拟生成一个微云的网络,此网络的拓扑包括一个数据中心,8 个微云,微云间延迟随机生成;并且根据微云的特点生成了一个 15 个服务请求分布,如下图 2 所示. 横坐标代表微云,纵坐标代表请求的个数,不同的颜色(深度)代表不同的服务. 每个服务可能仅仅在部分微云中出现,并且微云的负载各不相同. 其他更多的参数如表 1 所示.

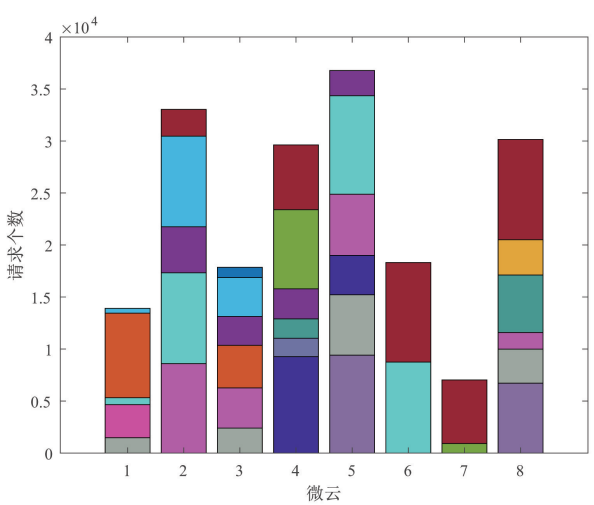


**图 2　请求分布**

**Fig. 2　Requests distribution**

实验时假设每个微云随机拥有 200 到 400 个 CPU,400 GB 到 800 GB 的内存以及 5 000 GB 到 10 000 GB 的磁盘容量,并且设定数据中心资源远超微云,分别有 10 000 个 CPU,20 000 GB 的内存,以及 100 000 GB 的存储. 设定微云之间的带宽 20 Mbps,相连微云之间的延迟为 10 ms 到 30 ms. 设定每个服务需要消耗 200 MB 到 2 GB 的内存,需要 1 到 4 个 CPU,以及 200m 到 2GB 的存储资源. 服务可用率在 90%到 99%之间,服务时间阈值设定为 10 ms 到 90 ms,单位 SLA 违反惩罚为 20 到 200. 请求上传下载的数据为 0.1 k～200 k,处理时间则在 5 ms～20 ms 之间. 每个区域和本地微云的延迟设定为 5 ms～10 ms,每个区域会产生 0 到 10 000 的不同请求. 微云的 CPU、内存、存储成本设定为 0.1 每 CPU,0.2 每 GB 以及 0.1 每 GB,而数据中心资源成本设定为微云一半左右,分别为 0.06 每 CPU,0.1 每 GB 以及 0.1 每 GB.

**表 1　实验参数**

**Tab. 1　Experimental parameters**

| $N$ | 8 | $M$ | 15 |
|---|---|---|---|
| $R_j^c$ | U(200～400)个 | $L_j^c$ | 0.1 每 CPU |
| $R_j^m$ | U(400～800)GB | $L_j^m$ | 0.2 每 GB |
| $R_j^s$ | U(5000,10000)GB | $L_j^s$ | 0.1 每 GB |
| $R_{N+1}^c$ | 10000 个 | $L_{N+1}^c$ | 0.06 每 CPU |
| $R_{N+1}^m$ | 20000GB | $L_{N+1}^m$ | 0.1 每 GB |
| $R_{N+1}^s$ | 100000GB | $L_{N+1}^s$ | 0.1 每 GB |
| $U_k^m$ | U(0.2,2)GB | $q_k$ | U(90～99)% |
| $U_k^c$ | U(1,4)个 | $th_k$ | U(10ms～90ms) |
| $U_k^s$ | U(0.2,2)GB | $p_k$ | U(20～200)每 1% |
| $r_{ij}$ | 20Mbps | $d_{ij}$ | U(10ms～30ms) |
| $x_k$ | U(0.1KB～200KB) | $y_k$ | U(0.1KB～200KB) |
| $w_k$ | U(5ms～20ms) | $\pi_k$ | U(100～200)个 |
| $d_i'$ | U(5ms～10ms) | $\lambda_i^k$ | U(0,10000)个 |

### 4.2　实验结果

**不同时间阈值下算法表现**　我们测试了时间阈值对 CO-Greedy、cloud、first-fit 以及 minCost 算法的影响,如图 3 所示. 其中 $x$ 轴是时间阈值,测试时将所有服务的时间阈值设定为 $x$,$y$ 轴是成本.

当时间阈值比较小时,大量服务无法找到满足其服务质量要求的微云,会产生大量 SLA 违反惩罚. 此时仅有在本地微云处理请求才能满足服务的延迟要求,所以 minCost 和 CO-Greedy 表现一致,而 first-fit 可能会部署 SLA 违反惩罚较小的服务,放弃关键服务,所以其成本较另外两种算法高. 当时间阈增加到达 30 ms 左右,越来越多的微云能够满足延迟要求,所以 first-fit, minCost 和 CO-Greedy 将更多服务部署到微云中. 因为 CO-Greedy 算法能优先部署关键应用的容器,所以其产生的 SLA 违反

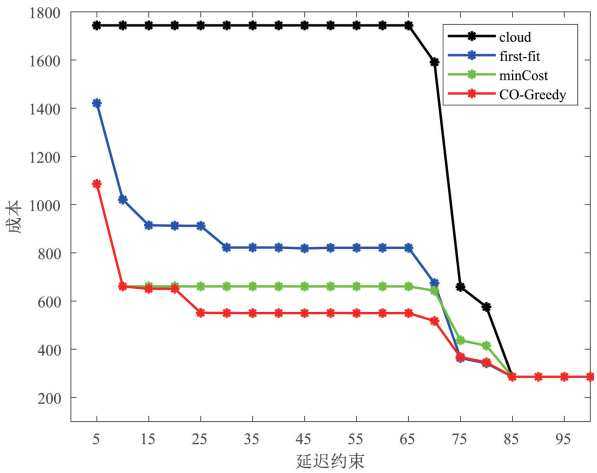


**图 3 时间阈值对成本的影响**

**Fig. 3 Impact of delay threshold on cost**

惩罚最小，因此成本相对较低. 随着阈值增加到达 80 ms 左右，数据中心也能够满足服务延迟要求，所以服务会大量迁移到数据中心，4 种算法趋向一致.

**不同资源规模下算法表现** 我们测试了资源规模对 CO-Greedy、cloud、first-fit 以及 minCost 算法的影响，如图 4 所示. 其中 $x$ 轴为资源规模，例如 $x=0.8$ 表示资源规模是原先的 0.8 倍，$y$ 轴则是成本.

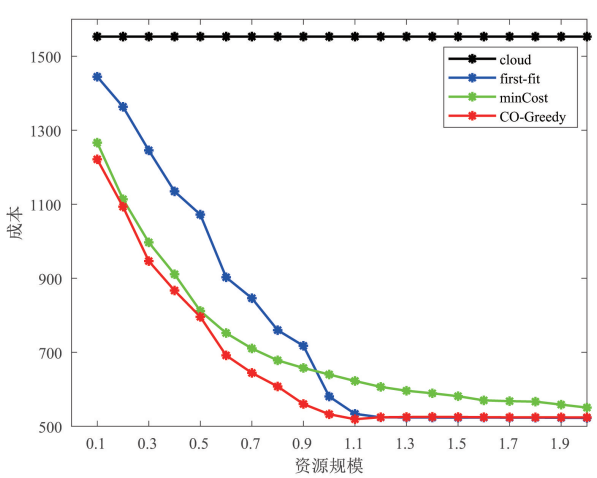


**图 4 资源规模对成本的影响**

**Fig. 4 Impact of resource scale on cost**

图 4 说明，因为延迟无法满足要求，所以 cloud 算法会产生大量 SLA 违反惩罚. 相比较于图 3 中的 cloud 算法，图 4 中 cloud 算法成本相对较低，这是因为在随机生成时间阈值情况下，部分服务的实时性要求较低，因此部署在数据中心也不会产生 SLA 违反成本. 在资源严重不足的情况下，由于没有额外的资源可以调度，minCost 算法和 CO-Greedy 算法相差无几，而 first-fit 则因为无法部署关键应用，成本会高于 minCost 和 CO-Greedy. 随着资源的增加 CO-Greedy 算法能成本下降更快，表现好于 first-fit 和 minCost. 当资源一直增加到 1.1 倍时，此时因为 CO-Greedy 算法和 first-fit 算法都能很好地利用微云服务所有请求，所以表现一致，而 minCost 因为容器无法迁移，整体资源利用率不高导致成本大于其余两种算法. 当资源足够多时，3 种算法都会部署足量容器，降低 SLA 违反惩罚，此时 3 种算法表现趋向一致.

**不同请求规模下算法表现** 我们测试了不同请求规模下 CO-Greedy、cloud、first-fit 以及 minCost 算法的表现，如图 5 所示. 其中 $x$ 轴表示请求的规模，$x=0.8$ 时表示请求数是原先的 0.8 倍，$y$ 轴则是成本.

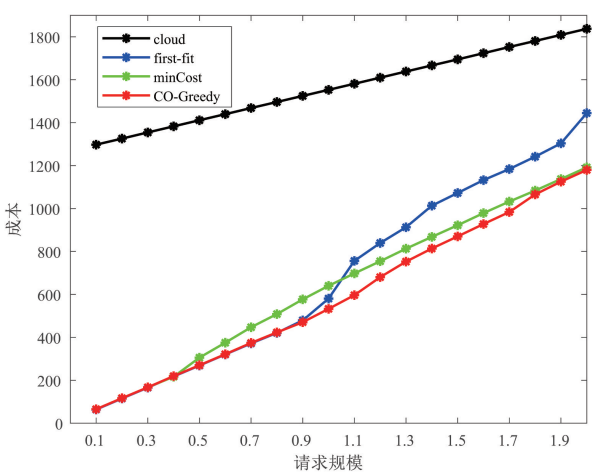


**图 5 请求规模对成本的影响**

**Fig. 5 Impact of request scale on cost**

图 5 说明，因为延迟无法满足，cloud 算法会产生大量的 SLA 违反惩罚. 随着请求规模的增加，所需要的资源成本也会增加，所以总成本会上升. 其余 3 种算法，在请求规模较小时，first-fit 算法、minCost 算法以及 CO-Greedy 算法都能够部署足量服务，满足所有请求的延迟要求，所以表现一致. 随着请求规模的扩大，CO-Greedy 算法合理部署容器的优势得到体现，成本一直小于 minCost 算法和 first-fit 算法. 请求规模到达足够大的程度时，微云处理能力有限，仅仅能够处理本地产生的请求负载，所以此时 minCost 和 CO-Greedy 表现一致，而 first-fit 虽然能够提高资源利用率，但是关键服务可能得不到部署，其成本值较其余两种算法更高.

**最优算法 OPT 与 CO-Greedy 算法的比较** 我们在小数据规模下测试最优搜索算法 OPT 与 CO-

Greedy 算法的成本偏差. 因为 OPT 算法的复杂度非常高,我们仅在 $N=2, M=2$ 的情况下,随机生成几组数据,进行测试. 实验结果如表 2 所示,表格中的数据为随机生成 5 组数据两种算法的成本值以及成本值偏差.

**表 2　CO-Greedy 算法与 OPT 算法成本比较**

**Tab. 2　Cost comparison of CO-Greedy and OPT**

| | | | | | |
|---|---|---|---|---|---|
| CO-Greedy | 48.28 | 32.02 | 45.14 | 380.48 | 217.44 |
| OPT | 45.71 | 30.37 | 43.88 | 362.05 | 209.29 |
| 成本偏差 | 5.32% | 5.13% | 2.79% | 4.84% | 3.75% |

比较 CO-Greedy 算法与 OPT 算法,我们发现两种算法之间成本偏差在 2%～6%之间. 存在差距的主要原因在于 CO-Greedy 算法可能会存在若干容器仅仅服务少量请求,而最优算法不会存在冗余容器. 此外 CO-Greedy 算法贪心地将容器迁移到资源使用率最小的微云之上,可能会陷入局部最优. 随着微云个数与服务个数的增加,OPT 算法的计算时间指数增加,无法得到有效解,CO-Greedy 则能够在多项式时间内得到方案.

### 4.3　实验总结

我们分别测试了不同时间阈值、不同资源规模以及不同请求规模下 CO-Greedy、cloud、first-fit 以及 minCost 算法的表现. 实验证明,CO-Greedy 算法在各种场景下表现最优. 此外,我们在小数据规模下比较了最优搜索算法 OPT 与 CO-Greedy,两种算法成本偏差在 2%～6%之间.

## 5　结论

本文主要研究微云环境下请求分发和容器部署成本优化问题,在综合考虑资源使用成本和 SLA 违反惩罚的前提下,提出成本优化的贪心算法 CO-Greedy. 该算法能够提供合理的请求分发与容器部署方案,降低云基础设施提供商的总体成本. 最后我们通过模拟实验验证了 CO-Greedy 算法比已有算法(cloud,first-fit,minCost)更加优秀,与最优搜索算法相比,CO-Greedy 算法的成本偏差在 2%～6%之间,但执行时间远远小于最优搜索算法.

### 参考文献(References)


[1] SATYANARAYANAN M, BAHL P, CACERES R, et al. The case for VM-based cloudlets in mobile computing[J]. IEEE Pervasive Computing, 2009, 8(4):14-23.

[2] TRAN T X, HAJISAMI A, PANDEY P, et al. Collaborative mobile edge computing in 5G networks: New paradigms, scenarios, and challenges[J]. IEEE Communications Magazine, 2017, 55(4): 54-61.

[3] ISMAIL B I, GOORTANI E M, AB KARIM M B, et al. Evaluation of docker as edge computing platform [C]// IEEE Conference on Open Systems. Bandar Melaka, Malaysia: IEEE, 2015: 130-135.

[4] BELLAVISTA P, ZANNI A. Feasibility of fog computing deployment based on docker containerization over RaspberryPi [C]// Proceedings of the 18th International Conference on Distributed Computing and Networking. Hyderabad, India: ACM, 2017: 1-10.

[5] FARRIS I, TALEB T, FLINCK H, et al. Providing ultra-short latency to user-centric 5G applications at the mobile network edge[J]. Transactions on Emerging Telecommunications Technologies, 2018, 29(4): e3169(1-13).

[6] YANG L, CAO J N, LIANG G Q, et al. Cost aware service placement and load dispatching in mobile cloud systems[J]. IEEE Transactions on Computers, 2016, 65(5):1440-1452.

[7] WANG W, ZHAO Y L, TORNATORE M, et al. Virtual machine placement and workload assignment for mobile edge computing [C]// IEEE 6th International Conference on Cloud Networking. Prague, Czech Republic: IEEE, 2017: 1-6.

[8] BORGETTO D, MAURER M, DA-COSTA G, et al. Energy-efficient and SLA-aware management of IaaS clouds[C]// Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet. Madrid, Spanish: ACM, 2012, No. 25:1-10.

[9] WU L L, GARG S K, VERSTEEG S, et al. SLA-based resource provisioning for hosted software-as-a-service applications in cloud computing environments [J]. IEEE Transactions on services computing, 2014, 7(3):465-485.

[10] ZHANG F F, GE J D, LI ZJ, et al. A load-aware resource allocation and task scheduling for the emerging cloudlet system [J]. Future Generation Computer Systems, 2018, 87:438-456.

[11] YOUSEFPOUR A, PATIL A, ISHIGAKI G, et al. Fogplan: A lightweight QoS-aware dynamic fog service provisioning framework [J/E]. Internet of Things Journal /https://personal. utdallas. edu/～ ashkan/papers/QDFSP. pdf, 2019.