论文信息

- 标题：Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum
- 时间：2022.08
- 期刊/会议：IEEE TPDS
- 架构名称：Nautils
- 作者：SJTU Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo Fellow, IEEE
- 实验环境：
- 数据集：

一、问题场景

考虑了云、边缘服务器、客户端三者参与的一个完整的微服务应用场景。每个完整的用户服务由多个阶段构成的微服务组成。微服务可以部署在边缘和云服务器上。边缘-边缘之间、边缘-云之间的通信使用不稳定的公共网络。

在进行服务部署的时候需要考虑：

- 各节点的计算能力有限；
- 各节点上的资源竞争，包括 CPU、内存、IO 等；
- 边缘-边缘、边缘-云节点之间的通信开销。

系统需要考虑同一个功能下的多个微服务之间的数据交互、资源竞争，将带有数据关联关系的微服务部署到不同的节点上，同时要最小化服务延时，用较少的资源实现更好的 QoS 。

二、研究对象

新功能/功能改进？解决（创新）思路，与现有工作的差别

2.1 传统微服务部署存在的问题

本架构研究的对象主要是单体服务中多个微服务的部署问题。传统的微服务部署架构，如 Kubernetes (K8s) ，缺少对通信开销和竞态资源的考虑，这使得在使用 99% 分位端到端时延作为 QoS 评估手段时，大部分服务质量都难以满足，尤其是在高负载下。同时，由于 K8s 将所有的微服务视为等同的，这样就导致高资源需求的微服务拉高整体资源需求，进而导致低需求的微服务被分配了过量资源，浪费了节点的可用资源。

作者同时通过实验证明，将 IO 密集型任务与微服务同时安排在一个节点上时会显著影响微服务的性能。作者因此提出：在存在外部 IO 压力时，需要将数据密集型的微服务迁移到可用的闲置节点上，降低外部 IO 对微服务的影响。但是，在进行服务迁移时，需要精确识别对整体时延有重点影响的服务，否则无法满足系统整体的时延需求。

2.2 新架构的主要挑战与设计考虑

本文所提出的架构与现有架构的主要区别在于:

1. 考虑部分微服务之间的大量数据交互，尽量将关系紧密的微服务部署在同一个节点来减少通信开销;
2. 基于运行时信息来对微服务的共享资源进行管理，尤其是对部署在相同节点的竞态资源管理;
3. 通过识别并降低部署在相同节点的 IO 敏感任务资源竞争来减少端到端时延;
4. 通过对微服务负载的动态感知对资源动态平衡分配以充分利用云边资源。

三、架构/模块/流程设计

设计理念/思路在流程/流表/系统组件/接口等的落实情况

3.1 系统设计

系统总体包括:

- 通信感知的微服务映射器
- 竞态感知的资源管理器
- IO 敏感的微服务调度程序
- 负载感知的微服务调度程序

系统工作流程是:

1. 微服务映射器将依照服务之间的通信数据量来为微服务选择部署节点，以达到最小化通信开销的目的。映射完成后将作为单次部署的初始状态;
2. 各节点部署完对应微服务后，由资源管理器负责对节点的可用计算资源进行分配，以确保服务 QoS 同时降低总体资源用量;
3. 当出现 QoS 违反时，微服务调度程序会将拥塞节点上的部分微服务迁移到其他空闲节点;
4. 在迁移完成前，资源管理器会暂停分配新资源给待迁移的微服务，直到微服务迁移完成;
5. IO 敏感的微服务调度程序会监测 IO 敏感的微服务，当微服务的 IO 访问延时增加时，迁移就会发生;
6. 负载感知的微服务调度程序会记录每个节点上微服务资源配额的总和。当节点没有空闲资源配额时，调度程序会将该节点定义为拥塞节点，并将部分微服务迁移到空闲节点。

3.2 通信感知的微服务映射器

该模块主要负责从通信开销的角度考虑将微服务映射到节点上，主要工作是运用一个多项式时间的近似算法来将微服务映射到节点上，具体来说:

微服务映射问题可以转化为最小 k 分割问题，也就是对于一个有向连通图，如何将其分为 k 个连通分量并保证分割后各分量之间的边权重最低。该模块执行的算法是 Ford-Fulkerson 。

假设现在需要将 n 个微服务映射到 k 个节点上（ $n \geq k$ ），映射器会进行 $k-1$ 次迭代，在每次迭代的过程中，映射器选择一个子图并将其分割为两个连通分量，之后算法继续对每个子图计算全局最小分割并选择最小分割 λ 。

Ford-Fulkerson 算法的近似度为 $2-2/k$ 。

参考资料

3.3 竞态感知的资源管理器

竞态感知的资源管理器主要使用基于深度强化学习的方法来对计算资源进行分配。计算资源的分配问题可以定义为:

$$\min \alpha \frac{\sum_{0<i\leq n} c_i * N_i}{C} + \beta \frac{\sum_{0<i\leq n} m_i * N_i}{M}$$

$$\min \sum_{0<i\leq k} b_i$$

$$s.t. \begin{cases} \sum_{i\in p_j} c_i N_i \leq C_j & j = 1, 2, ..., k \\ \sum_{i\in p_j} m_i N_i \leq M_j & j = 1, 2, ..., k \\ f(C, M) + \sum_{0<i\leq k} \frac{D_i}{b_i} \leq QoS \\ \min tr(C, M) \geq throughput \end{cases}$$

优化目标包括计算资源（CPU、内存）和网络资源（带宽）。

限制条件包括:

- 所有微服务共享的全局内存不能超过节点的可用内存容量;
- 所有微服务使用的计算资源不能超过节点的可用核心数;
- 微服务应用的运行时间应当符合 QoS 要求;
- 微服务应用提供的兔兔两应当对大于用户需求。

由于两个优化目标之间存在冲突，不可避免地需要在两者之间取得一个平衡，这里的优化问题实际上是一个帕累托优化的问题。

在相关工作的实现中，微服务程序各阶段的资源使用情况是预先知道的，但是本文所设计的系统并不认同这种不能在运行时获取的信息。其他工作倾向于在程序运行过程中进行采样和评估从而对应用的实际使用进行一个估计。

Nautilus 把资源分配问题转化为一个马尔可夫决策问题，之后又使用 Dueling DQN 模型来解决该问题。具体来说:

在 t_k 时刻，算法检查系统当前状态 s_k （包括：资源用量【CPU、内存、网络带宽】），同时初始化动作集合 a_k 。算法随后使用深度神经网络对不同动作下，端到端延时和峰值吞吐量进行预测。当选择能够获得最好表现的动作后，算法获得对应的奖励 R_k 。最后，算法存储状态集合 s_k 、动作集合 a_k 和奖励 R_k 到经验池中。

深度学习的奖励函数设计包括了约束中的两个最重要的部分：端到端延时和吞吐量。

$$R = \begin{cases} -\theta_1 \frac{}{QoS_r} tr_r + \theta_2 Res, & QoS_r > 1 or tr_r < 1 \\ Res, & Otherwise \end{cases}$$

其中 Res 指的是资源奖励，也就是如果一次资源分配能够用更少的资源，就说明该次资源奖励更好。

$$Res = \alpha \frac{CPU_m}{CPU_u} + \beta \frac{Mem_m}{Mem_u} + \gamma \frac{IO_m}{IO_u}$$

当 QoS 要求和吞吐量要求没有达成时，就会从奖励函数中扣除一部分作为没有完成目标的惩罚。

3.4 IO 敏感的微服务调度程序

本模块关注的核心问题有两个：

1. 哪些微服务需要被迁移；
2. 是否需要迁移微服务。

系统周期性地对所有的微服务进行抽样来测量服务时间的分布特性。同时通过计算微服务和整体应用之间的“距离”来衡量某个微服务对总体应用的影响。Nautilus 使用 Bhattacharyya 因子来衡量两个分布之间的“距离”。为了消除异常值的影响，使用对数时间来计算分布规律，进而得出了 Bhattacharyya 因子的计算公式为：

$$p(x) = ecdf(\log(t_1, t_2, ..., t_n))$$
$$BC(p_i, p_t) = \sum \sqrt{p_i(x) p_t(x)}$$

如果 BC 大于某个特定的阈值，就说明该微服务显著影响到了总体的服务时间，因而 Nautilus 需要考虑对该微服务进行迁移来降低其对总体服务延时的影响。

3.5 负载感知的微服务调度程序

相关工作表明用户的需求模式服从一种每日周期，也就是说如果不对一开始的部署方式进行改动，当需求开始增加时，会导致系统出现瓶颈。此时，将一些微服务从过载的节点上迁移到空闲的节点上就是一种自然的想法。但考虑到微服务之前的强耦合性，需要特殊挑选迁移的服务。

本模块关注的两个核心问题是：

1. 哪些微服务需要被迁移；
2. 这些微服务迁移到哪个节点。

系统构建了通信增量模型来衡量迁移对通信的影响。假设微服务系统部署在一个拥有 k 个节点的集群中，微服务 i 部署在节点 N_j 上，此时将微服务从 N_j 迁移到 N_k 上带来的通信增量为:

$$\Delta C^i_{N_j \to N_k} = \sum_{m \in N_j} e_{im} - \sum_{m \in N_k} e_{im}$$

其中 $\sum_{m \in N_j} e_{im}$ 代表了 N_j 上的微服务与 i 的通信量。这个式子代表服务迁移会导致原先部署在同一节点的服务需要通过公共互联网来传递数据，同时迁移到的新节点的服务也不需要进行公网通信。

Nautilus 倾向降低通信增量，但是在进行迁移时也需要考虑新的节点是否能够为迁移的服务提供可用的计算资源。系统为每个节点建立一个资源需求表。在节点 N_j 的每一行中，记录要迁移的微服务 m_i、目标节点 N_k、微服务的 CPU 内核消耗 cpu_i 、内存容量 mem_i 和通信增量 $\Delta C_{N_j \to N_k}$ 。系统在进行迁移时会挑选 $cpu_i \leq cpu_{N_k}$ 和 $mem_i \leq mem_{N_k}$ 的节点中通信增量更少的节点作为目的节点。

四、系统实现（如果有）

什么硬件平台、工作量评估

五、实验结果

实验设置（功能 vs. 性能），如何体现优势的，是否与创新思路形成了闭环？有什么指导性建议？有何结论（结果和结论的关系密切吗？客观吗?）

1、比其他工作的优势

2、有优势的原因

3、改进空间

五、启发与思考

三条优点，学到了什么（可以关于调研、创新思路、处理技巧、论文写作、逻辑等等）

套到自己关心的问题中，有什么值得借鉴的吗？论文的方法，适用性上有何局限？