计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(2):55−61 [doi: 10.15888/j.cnki.csa.006790]

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

基于服务网格的微服务架构服务治理研究①

郑俊褒, 沈林强

(浙江理工大学 信息学院, 杭州 310018)
通讯作者: 沈林强, E-mail: 1036272643@qq.com

摘　要: 由于微服务细粒度的服务拆分方式和去中心化的架构设计相比于传统 SOA 架构更适合当前互联网敏捷开发、快速迭代的需求, 但是传统微服务的服务治理技术难以实现不同技术框架和通讯协议建设的服务之间互联互通, 并且存在服务治理与服务高耦合的问题. 本文基于服务网格思想实现一个具有服务注册发现、负载均衡、协议转换的网络代理作为微服务架构的服务治理独立组件, 并通过 Netty 框架、protobuf 序列化方式、Etcd 注册中心和加权轮询的负载均衡算法最大化网络代理性能, 实验结果表明本文的设计克服了传统微服务存在的问题, 并且网络代理具备高可用、高并发、高吞吐量的性能.

关键词: 微服务; 服务网格; 服务治理; Netty; 加权轮询负载均衡

引用格式: 郑俊褒,沈林强.基于服务网格的微服务架构服务治理研究.计算机系统应用,2019,28(2):55–61. http://www.c-s-a.org.cn/1003-3254/6790.html

Research on Service Governance of Microservice Architecture Based on Service Mesh

ZHENG Jun-Bao, SHEN Lin-Qiang

(School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou 310018, China)

Abstract: Because microservice fine-grained service splitting and decentralized architecture design is more suitable for current Internet agile development and rapid iteration than traditional SOA architecture, but traditional microservice service governance technology is difficult to realize the interconnection between services of different technical frameworks, and communication protocols, and there is a problem of high coupling between service governance and services. Based on the service mesh idea, this study implements a network proxy with service registration discovery, load balancing and protocol conversion as a service governance independent component of the microservice architecture, and maximize network proxy performance through Netty's framework, protobuf serialization, ETCD registry, and weighted polling load balancing algorithms. The experimental results show that the design of this study overcomes the problems of traditional microservice, and the network proxy has high availability, high concurrency, and high throughput performance.

Key words: microservice; service mesh; service governance; Netty; weighted polling load balancing

日益丰富的业务场景和不断提高的系统性能要求使得软件系统逐渐走上了分布式的道路. 而电商平台、物联网平台这种需要快速响应市场变化、系统更新迭代频繁的软件系统对系统扩展性、敏捷开发能力提出了更高的要求, 传统面向服务 (Service-Oriented Architecture, SOA) 架构[1,2]建设的系统由于其粗粒度的服务划分和企业服务总线的设计无法满足这类系统的性能需求, 随着 Docker 等容器技术[3]和云计算技术的发展, 微服务架构逐渐成为更优的选择.

微服务架构[4–6]是一种架构风格, 将大型复杂系统细粒度的拆分成多个能够独立运行、职能单一的服务, 服务之间通过通用的协议进行通讯. 微服务架构的系

① 收稿时间: 2018-08-29; 修改时间: 2018-09-27; 采用时间: 2018-09-30; csa 在线出版时间: 2019-01-28

统在系统扩展性、敏捷性等方面相比于 SOA 架构的系统都具有明显的优势, 但是这种细粒度拆分服务的方式势必会使服务治理变得异常复杂. 传统服务治理框架, 如 Dubbo、Spring Cloud 为微服务服务治理提供了切实有效的解决方案, 但都存在难以实现不同技术框架和通讯协议建设的服务之间互联互通和服务治理与服务高耦合的问题. 本文基于服务网格的思想对微服务服务治理进行改造, 从服务中抽取出服务治理相关功能并集成在网络代理上, 网络代理作为服务治理的独立组件解决了服务治理和服务高耦合的问题, 并且网络代理会对所有进出服务的流量进行拦截, 能够实现不同技术框架和通讯协议建设的服务之间互联互通.

1 相关工作

1.1 服务治理

在微服务架构中服务治理是整个系统正常运行的关键技术, 在大型复杂系统环境下, 服务间的调用会变得非常复杂, 如果没有一套完善的、经过大规模生产环境验证的服务治理方案的话, 系统将会处于非常危险的境地. 为解决复杂环境下服务治理的问题, 出现了很多服务治理框架, 其中应用最广泛有 Dubbo 和 Spring Cloud.

Dubbo 是由阿里巴巴中间件团队开源的基于 Java 的高性能远程过程调用[7](Remote Procedure Call, RPC) 通讯框架, 在其提供的服务整合能力支持下, 使用 RPC 可以像使用本地调用一样轻松便捷, 并且 Dubbo 是阿里巴巴内部 SOA 服务治理方案的核心框架, 每天为 2000 多个服务提供数亿次访问量支持[8], Dubbo 已不只是单纯的服务通讯框架, 更是一套完备的服务治理框架, Dubbo 也因为其优秀的服务治理能力和高效的 RPC 通讯能力成为了微服务架构的一种优秀解决方案.

Spring Cloud 是一系列 Spring 框架的集合, 以 SpringBoot 作为开发的基础, 通过 SpringBoot 可以简单高效地集成服务发现注册、负载均衡、断路器[9]等其他 Spring 家族的分布式框架[10], 相比于 Dubbo 框架, Spring Cloud 最大的特色在于一站式的分布式系统架构.

但无论是以 Dubbo 还是 Spring Cloud 作为服务治理核心框架的微服务系统都需要对所有接入的服务引入组件, 并在业务服务中并暴露或消费相关的服务, 这会大大增加服务治理和业务服务之间的耦合度, 增加服务开发的成本; 此外这种微服务架构系统虽然从一定程度上为不同技术框架和通讯协议建设的服务之间互联互通提供了条件, 但是其实现的成本是相当高昂的, 例如在以 Dubbo 微服务架构的系统中引入另一个 Spring Cloud 为技术框架的服务, 那么可能需要对原先系统的每个服务进行升级才能实现, 这在实际场景下很难做到.

1.2 服务网格

2017 年 4 月, Buoyant 的首席执行官 William Morgan 在其公司平台首次定义了服务网格[11], 与传统微服务架构不同, 服务网格另辟蹊径, 其实现服务治理的过程不需要改变服务本身, 通过代理或边车形式部署网络代理, 所有进出服务的流量都会被网络代理拦截并加以处理. 这样一来微服务场景下的各种服务治理能力都委托给一个高可用的网络代理, 降低了服务治理和服务的耦合; 而且当具有协议转换功能的网络代理作为服务之间的传输媒介时, 可以很方便的实现基于不同技术框架和通讯协议建设的服务之间互联互通.

当一个系统采用服务网格化的微服务架构时, 系统网络拓扑图如图 1 所示. 网络代理作为服务治理的独立组件会和每个业务服务部署在同一容器中, 当业务服务需要调用其他服务时, 业务服务只向同一容器下的网络代理发送请求, 实际的服务通讯和治理是在两个网络代理之间完成的, 整个系统以网络代理作为服务间通讯的专用基础设施.

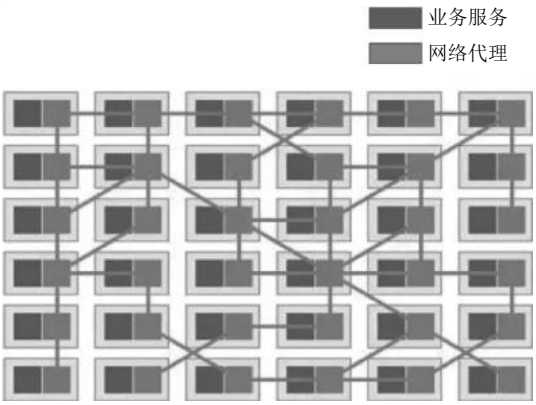
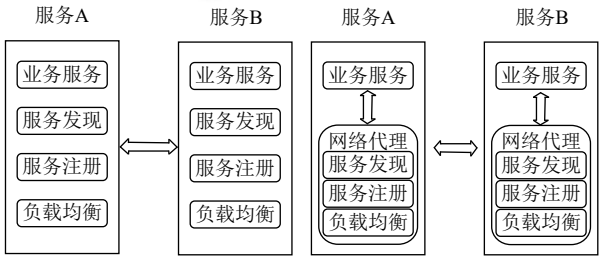


图 1 服务网格化的微服务系统网络拓扑图

图 2 展示了微服务和服务网格实现的系统服务调用的区别, 其中图 2(a) 是传统微服务服务调用方式示

意图, 图 2(b) 是服务网格服务调用方式示意图. 传统微服务没有实现业务服务和服务治理的分离, 服务治理通过框架配置等方式和业务服务部署在同一服务中, 耦合性高, 不利于服务的升级迭代, 开发维护人员除了要完成业务服务之外还要兼顾服务发现注册等服务治理工作, 无疑也增加了开发维护的成本; 此外当系统需要接入不同技术框架和通讯协议建设的服务时, 需要对每个相连的服务增加协议转换功能, 这对于一个大型系统是不能接受的. 服务网格化的服务抽取了服务发现注册等服务治理功能并集成于独立的网络代理, 降低了服务和服务治理的耦合性, 服务只包含具体业务服务, 开发维护人员只需要专注实现业务服务, 提高了开发维护效率; 服务调用时会请求部署在同一容器的网络代理, 实际的服务通讯在两个不同容器的网络代理间完成, 当系统需要接入不同技术框架和通讯协议建设的服务时, 只需要在通用的网络代理中支持协议转换等功能, 不会对服务造成侵入.



(a) 传统微服务服务调用方式 (b) 服务网格服务调用方式

图 2 服务网格化的微服务和传统微服务服务调用方式

综上所述克服传统微服务的两大问题的关键是实现一个独立的、高可用的、具备服务治理、协议转换等功能的网络代理; 另一方面, 由于网络代理的存在, 每次服务调用会增加两次从服务到网络代理的网络开销, 因此网络代理还必须具备高并发、高吞吐量的性能要求. 网络代理功能、性能需求如下:

(1) 良好的服务治理能力. 网络代理作为服务调用的基础设施, 服务治理是其最基础的能力.

(2) 支持协议转换. 目前主流的微服务架构技术框架有以 HTTP 作为传输协议的 Spring Cloud 框架、以 DUBBO 等作为传输协议的 Dubbo 框架, 技术框架选择很多, 可选的传输协议也很多, 微服务的架构设计为不同技术框架和不同传输协议建设的服务之间互联互通提供了前提条件, 在本文中将协议转换的功能委托给网络代理, 在不同传输协议的服务之间通讯时, 只需要通用的网络代理实现了协议转换功能, 而不需要对每个服务都进行升级.

(3) 支持高吞吐量的网络通讯能力. 系统吞吐量是系统每秒钟处理完的事务数, 是衡量一个系统性能的关键指标. 本文通过代理的方式讲服务治理从服务中独立, 同时也增加了额外的网络开销, 如果网络代理的性能不佳, 会大幅提高系统的响应时间, 因此高吞吐量的网络传输能力也是网络代理必不可少的.

2 网络代理实现

针对上文网络代理功能和性能的需求, 本文以图 3 网络代理的实现机制对网络代理进行构建. 其中 consumer-agent 和 provider-agent 是网络代理对应于服务调用的消费者网络代理和生产者网络代理, 图中以 Spring Cloud 为技术框架的 consumer 去调用以 Dubbo 为技术框架的 provider 为例, 服务调用的流程如下:

(1) 启动 provider-agent 后, 会通过 Etcd 客户端向 Etcd 注册中心注册服务信息.

(2) consumer 通过 HTTP 协议请求 consumer-agent, consumer-agent 内部的 HTTP 服务端接收 HTTP 请求并交给 HTTP 编解码器解码, 得到请求信息.

(3) HTTP 服务端将请求信息转交给 TCP 客户端, TCP 客户端会根据请求信息获取需要调用的服务, 并通过 Etcd 客户端向 Etcd 注册中心拉取提供该服务的 provider 信息, 如果有多个提供该服务的 provider, 会根据加权轮询负载均衡算法选择路由. 确定 provider 之后, 将请求信息通过 protobuf 编解码器进行编码, 并通过 TCP 协议传输给 provider-agent 的 TCP 服务端.

(4) TCP 服务端通过 protobuf 编解码器将信息解码, 并交给 DUBBO 客户端, 再通过 DUBBO 编解码器将信息以 DUBBO 协议传输给 provider.

(5) Provider 将请求解析并进行具体的业务处理, 而后逆反上续步骤将处理结果返回给 consumer, 完成一次服务调用.

网络代理的关键技术点有三: 服务治理、网络传输和协议转换, 下文对这三个关键技术点的具体实现和技术选型进行了详细的阐述. 在这之前本文确定选择 Java 语言实现网络代理, 选择 Java 语言的原因有两点: 其一 Java 是分布式系统领域使用最为广泛的语言, 切合网络代理的使用场景; 其二 Java 语言从诞生开始就主打的几乎完美的跨平台能力, 这对于分布式场景特别是异构系统场景是相当重要的能力.

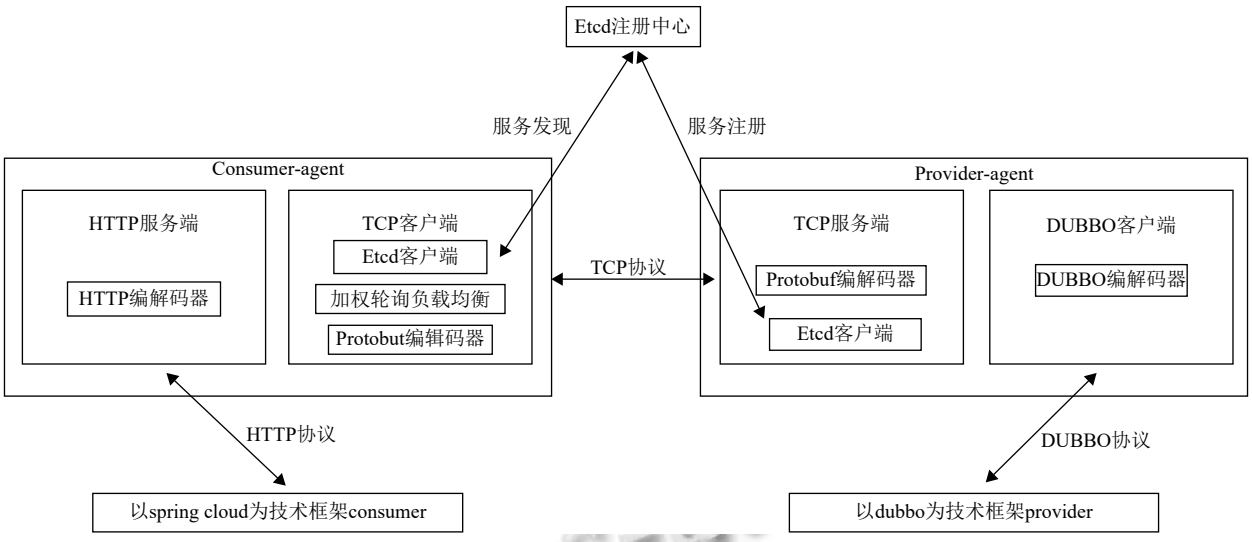


图 3 网络代理实现机制

2.1 服务治理

服务治理包括服务注册发现、负载均衡等, 其中服务的注册发现是最重要最基础的服务治理能力, 而负载均衡能力是一个稳定、高性能微服务系统不可缺少的能力, 本文的网络代理设计服务治理包括服务发现注册和负载均衡.

服务注册和发现依赖服务注册中心实现, 在本设计中, 网络代理的轻量化会是更优雅的选择, 因此选用轻量、简单、可靠的注册中心及其客户端会是更优的选择. Etcd 是一个分布式的 key-value 存储系统, 其特点就是简单安全, 方便可靠, 特别由于 Etcd 提供 HTTP+JSON、gRPC 接口, 能很好地实现跨语言跨平台的要求, 并且 Etcd 已经在 Google 著名的容器管理工具 Kuberbetes 有广泛应用, 是一款完善的、经过大规模生产环境验证的注册中心产品, 与传统的注册中心 Zookeeper 相比, Etcd 在一致性协议、运维管理、社区活跃度等方面都完胜于 Zookeeper[12]. 本文采用 Etcd 作为服务的注册中心, 在网络代理中通过 Etcd Java 客户端实现服务的发现和注册.

负载均衡算法主要有轮询法、加权轮询法、随机法、原地址哈希法等, 对于配置性能相同的服务来说采用轮询或随机的方法简单有效, 但是多数分布式场景中服务的性能是不相同的, 热点服务或需要大量计算服务的配置往往更好, 这是分布式系统的一大优势. 本文采用加权轮询的方法作为负载均衡的算法, 这种方式能实现对不同配置的服务按需分配不同比例的负载, 提高系统抗压能力, 而且结合 Etcd 注册中心, 能很好地实现负载调整. 上文说到 Etcd 是一个 key-value 存储系统, 在服务注册的时候, 把 key 作为服务提供者的 IP 地址, value 作为负载权重, 就可以实现对不同服务的加权处理. 在服务消费者网络代理选择路由时, 消费者网络代理将服务发现得到的对应服务生产者网络代理信息保存记录, 并在内部维护一个线程安全的计数器, 每次需要选择负载的时候根据计数器和权重选择路由就可以实现.

2.2 网络传输

网络代理需要支持高吞吐量的远程调用能力, 异步非阻塞的网络 I/O 模型会是更优的选择, 异步非阻塞的网络 I/O 模型是应用程序接收到 I/O 操作请求后将 I/O 处理交给操作系统, 应用程序并不直接参与, 通过事件通知或轮询的方式得到 I/O 操作的结果, 这种方式的 I/O 处理模型在高并发或 I/O 处理耗时长等场景都具有不阻塞网络, 系统资源利用率高的优点. 网络代理采用 Netty 作为网络 I/O 模型技术框架, Netty 是基于 Java 的 NIO 框架, 是 Java 生态圈首选的网络通讯框架. Netty 提供异步的、非阻塞的、事件驱动的网络应用程序框架和工具, 用以快速开发高性能、高可靠性的网络服务器和客户端程序, 由于其良好的线程模型设计, 以 Netty 实现的服务端可以花费少量系统开销就能实现上万的并发连接[13]. 图 3 中包括 HTTP 服务端、TCP 客户端、TCP 服务端和 DUBBO 客户端都是基于 Netty 实现的.

除了网络 I/O 模型外, 网络传输协议也是网络传输性能的关键因素. TCP 面向连接的传输协议相比于

UDP 无连接传输协议最大的特点在于可靠安全, 这在本文的微服务场景下是非常重要的性能. 在生产环境下, 集群的网络环境常常是不稳定的, 网络传输的可靠性是整个系统的一项重要指标. 网络代理之间的传输协议采用 TCP 协议, 为了保证传输效率, 不多次反复创建 TCP 连接, 采用 TCP 长连接的方式.

网络代理具备高吞吐量的远程调用能力的另一个关键是网络传输时的序列化方式, 不同的序列化方式产生的字节流是不同的, 一般来说, 码流越小的网络传输传输效果更快. 本文采用 protobuf 作为网络代理之间连接的序列化编解码器, 相比于原生 JDK 序列化、JackSon、Hessian 等序列化方式无论是在处理时间、空间占用方面都有较大优势[14].

2.3 协议转换

HTTP 和 DUBBO 协议是目前微服务最常用的通讯协议, 本文的网络代理需要实现这两种协议的互相转换. 上文提到网络传输的设计是基于 Netty 实现的, Netty 提供了一整套完善的 NIO 客户端、服务端处理框架, 能够很好地处理网络代理之间的连接心跳、协议转换等问题, 网络代理的协议转换功能就是基于 Netty 的编解码器实现.

3 实验设计

3.1 实验场景

实验设计一个以 Spring Cloud 实现的 consumer 来消费三个性能不同的以 Dubbo 实现的 provider 的场景, 并通过 wrk 压测工具向 consumer 施压, 通过 lua 脚本获取压力测试结果, 实验场景并不复杂 (如图 4), 评测系统只涉及单接口评测, 将压力测试得到的 QPS 作为吞吐量的评价分数, 以此来评价网络代理的性能.

得益于 Docker 容器技术可快速方便地部署本实验的场景. 实验环境部署在两台物理机上, 一台为施压机, 配置为 MACOS, 4 核 8 G, 通过 wrk 压测工具向另一台物理机施压; 另一台为被压机, 配置为 CentOS7, 8 核 16 G, 通过 Docker 容器技术部署 5 个 Docker 实例:

(1) 以 Spring Cloud 为技术框架实现的 consumer 及其网络代理 agent, JVM 参数为 1536 M 的堆内存分配.

(2) 以 Dubbo 为技术框架实现的 small-provider 及其网络代理 agent, JVM 参数为 512 M 的堆内存分配.

(3) 以 Dubbo 为技术框架实现的 medium-provider 及其网络代理 agent, JVM 参数为 1536 M 的堆内存分配.

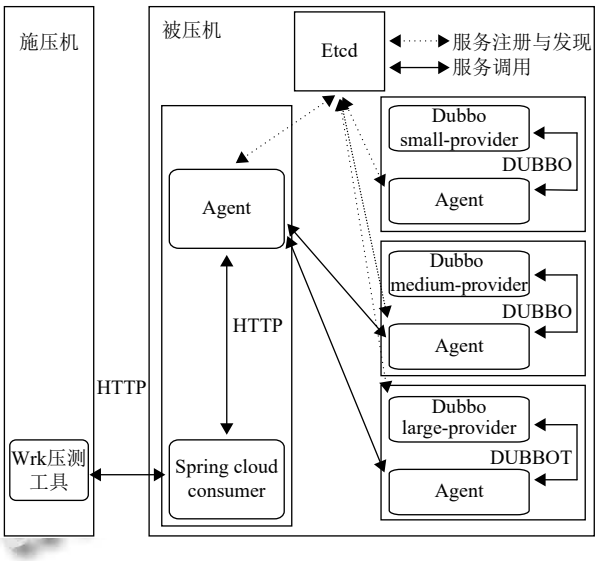


图 4 实验场景

(4) 以 Dubbo 为技术框架实现的 large-provider 及其网络代理 agent, JVM 参数为 2560 M 的堆内存分配.

(5) 注册中心 Etcd.

实验整体流程如下: consumer 在接收到客户端请求以后, 会生成一个随机字符串, 该字符串经过 consumer-agent 和 provider-agent 后到达 provider, 为了模拟现实情况下查询数据库等耗时的操作, provider 人为增加 50 ms 延时, 并计算哈希值后返回, client 会校验该哈希值与其生成的数据是否相同, 如果相同则返回正常 (200), 否则返回异常 (500). 每个通信环节的通讯协议如表 1 所示.

表 1 通讯环节与远程通讯协议

通讯环节	远程通讯协议
client → consumer	HTTP
consumer → consumer-agent	HTTP
consumer-agent → provider- agent	TCP
provider-agent → provider	DUBBO
provider → provider- agent	DUBBO
provider- agent → consumer- agent	TCP
consumer- agent → consumer	HTTP
consumer → client	HTTP

3.2 实验结果

实验通过网络代理实现以 Dubbo 为技术框架、DUBBO 为传输协议建设的服务和以 Spring Cloud 为技术框架、HTTP 为传输协议建设的服务之间的通讯, 并且网络代理和业务服务互相独立, 实现了网络代理的功能要求. 另一方面, 实验通过 wrk 压测工具对实验

场景进行施压, 并将 QPS 作为网络代理性能的评价分数. 为了模拟不同的业务环境, 并发数分别为 128, 256 和 512, 单次实验结果如图 5 所示, 为减少实验环境的不稳定, 进行 10 次不同压测记录, 结果如图 6 所示.

128并发

```
2 threads and 128 connections
Thread Stats   Avg      Stdev     Max   +/- Stdev
  Latency    53.11ms    2.37ms 127.38ms   93.49%
  Req/Sec     1.21k    51.90     1.29k    85.17%
Latency Distribution
   50%   52.54ms
   75%   53.35ms
   90%   54.71ms
   99%   61.28ms
 144607 requests in 1.00m, 19.45MB read
Requests/sec:   2408.55
Transfer/sec:    331.65KB
----------
Durations:       60.04s
Requests:        144607
Avg RT:          53.11ms
Max RT:          127.378ms
Min RT:          51.339ms
Error requests:  0
Valid requests:  144607
QPS:             2408.55
----------
```

256并发

```
2 threads and 256 connections
Thread Stats   Avg      Stdev     Max   +/- Stdev
  Latency    59.81ms    8.76ms 288.63ms   90.95%
  Req/Sec     2.15k   159.59     2.53k    81.08%
Latency Distribution
   50%   57.57ms
   75%   61.77ms
   90%   67.86ms
   99%   91.10ms
 257076 requests in 1.00m, 34.57MB read
Requests/sec:   4282.89
Transfer/sec:    589.73KB
----------
Durations:       60.02s
Requests:        257076
Avg RT:          59.81ms
Max RT:          288.633ms
Min RT:          51.358ms
Error requests:  0
Valid requests:  257076
QPS:             4282.89
----------
```

512并发

```
2 threads and 512 connections
Thread Stats   Avg      Stdev     Max   +/- Stdev
  Latency    87.54ms   25.44ms 469.10ms   82.44%
  Req/Sec     2.95k   296.72     3.65k    74.08%
Latency Distribution
   50%   83.34ms
   75%   96.81ms
   90%  115.50ms
   99%  176.59ms
 352091 requests in 1.00m, 47.48MB read
Requests/sec:   5865.06
Transfer/sec:    809.83KB
----------
Durations:       60.03s
Requests:        352091
Avg RT:          87.54ms
Max RT:          469.103ms
Min RT:          51.358ms
Error requests:  0
Valid requests:  345831
QPS:             5760.79
----------
```

图 5 单次压测 128、256、512 并发实验结果

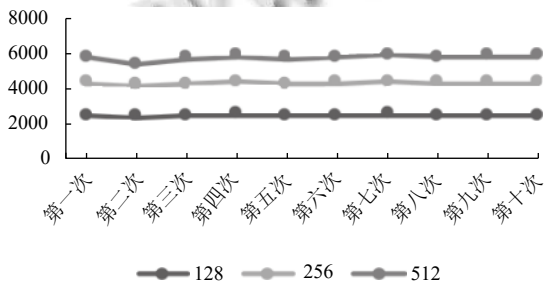


图 6 压测 10 次实验结果

由图 5 和图 6 可知, 系统平均响应时间去除 provider 中 50 ms 的人为延时, 128 并发下能做到整个流程的传输时间在 4 ms 以内, 256 并发下整个流程的传输时间在 12 ms 以内, 并且 10 次压测结果波动很小, 网络代理在并发量不大的环境下响应速度快, 性能稳定; 在 512 并发下, 整个流程的传输时间在 40 ms 以内, 10 次压测结果虽然有一定的波动, 但波动浮动不大, 并且将近 400 万的请求中没有出现异常错误的情况, 网络代理能满足高并发、高吞吐量的性能要求.

实验也对随机法、轮询法和加权轮询法的负载均衡算法进行对比, 通过记录每个 provider 处理的请求数和压测结果 QPS 分析不同负载均衡算法的性能, 实验结果如图 7 所示. 实验表明, 轮询和随机的负载均衡算法三个 provider 处理的请求数基本相同, 轮询法的 QPS 相对更加稳定, 并且这两种负载均衡算法在 512 并发情况下均出现了请求异常的情况, 这是因为没有合理分配负载, 过量的请求将配置低的 small-provider 线程池撑满溢出, 造成请求失败的情况. 而加权轮询的负载均衡算法三个 provider 处理的请求数基本按照 small:medium:large=1:2:3 分配, 负载合理, QPS 比另外两种负载均衡算法都高, 并且全程没有出现请求失败的情况. 实验结果表明, 加权轮询的负载均衡算法更适合网络代理的设计.

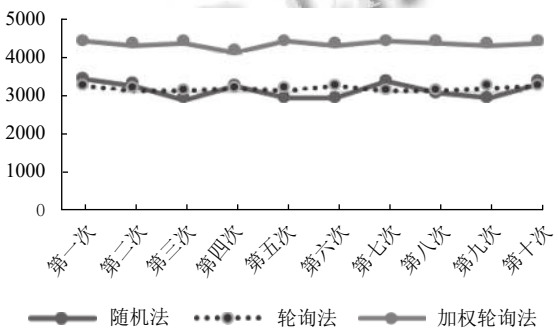


图 7 不同负载均衡算法实验结果

为了比较不同序列化方式对网络传输性能的影响, 实验对 Java 原生序列化、JackSon、Hessian 和 protobuf 进行了对比, 通过 10 次 256 并发下的压测结果分析不同序列化方式网络传输的性能, 实验结果如图 7 所示. 实验结果表明, 基于 protobuf 序列化方式的系统 QPS 总是优于其他几种序列化方式, 更适合网络代理的设计.

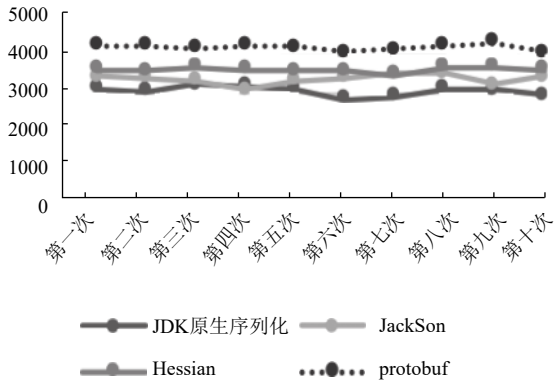


图 8 不同序列化方式的实验结果

4 结论

本文基于服务网格思想实现了一个具有服务注册发现、负载均衡、协议转换的网络代理作为微服务架构的服务治理独立组件, 解决了传统微服务架构中服务治理与服务之间高耦合的问题; 并且在每个服务容器中部署网络代理实现了对所有进出服务的流量拦截控制, 通过网络代理能优雅的实现不同技术框架和通讯协议建设的服务之间互联互通. 实验结果表明, 本文的设计实现了以 Dubbo 技术框架、DUBBO 通讯协议建设的服务和以 Spring Cloud 技术框架、HTTP 通讯协议建设的服务之间的互联互通和服务治理从服务中的独立; 另一方面, 实验对系统进行不同并发的压力测试, 实验结果表明本文的设计具备高并发、高吞吐量、高可用的性能要求.

参考文献

1 郭正敏. 基于 SOA 架构的分布式服务化治理方案的研究[硕士学位论文]. 南京: 南京邮电大学, 2016.
2 Choi J, Nazareth DL, Jain HK. Implementing service-oriented architecture in organizations. Journal of Management Information Systems, 2010, 26(4): 253–286. [doi: 10.2753/MIS0742-1222260409]
3 张忠琳, 黄炳良. 基于 OpenStack 云平台的 Docker 应用. 软件, 2014, 35(11): 73–76. [doi: 10.3969/j.issn.1003-6970.2014.11.015]
4 Thönes J. Microservices. IEEE Software, 2015, 32(1): 116. [doi: 10.1109/MS.2015.11]
5 郭栋, 王伟, 曾国荪. 一种基于微服务架构的新型云件 PaaS 平台. 信息网络安全, 2015, 15(11): 15–20. [doi: 10.3969/j.issn.1671-1122.2015.11.003]
6 王磊. 微服务架构与实践. 北京: 电子工业出版社, 2015.
7 Birrell AD, Nelson BJ. Implementing remote procedure calls. ACM Transactions on Computer Systems, 1984, 2(1): 39–59. [doi: 10.1145/2080.357392]
8 Alibaba open sesame. http://dubbo.apache.org/en-us/, 2016.
9 Nygard MT. Release it! America: Pragmatic Bookshelf, 2007.
10 Spring Cloud. http://projects.spring.io/spring-cloud/, 2017.
11 Service Mesh. https://blog.buoyant.io/2017/04/25/, [2017-04-25].
12 周佳威. Kubernetes 跨集群管理的设计与实现[硕士学位论文]. 杭州: 浙江大学, 2017.
13 Maurer N, Wolfthal MA. Netty in Action. Shelter Island, NY, USA: Manning Publications, 2015.
14 聂晓旭, 于凤芹, 钦道理. 基于 Protobuf 的数据传输协议. 计算机系统应用, 2015, 24(8): 112–116. [doi: 10.3969/j.issn.1003-3254.2015.08.019]