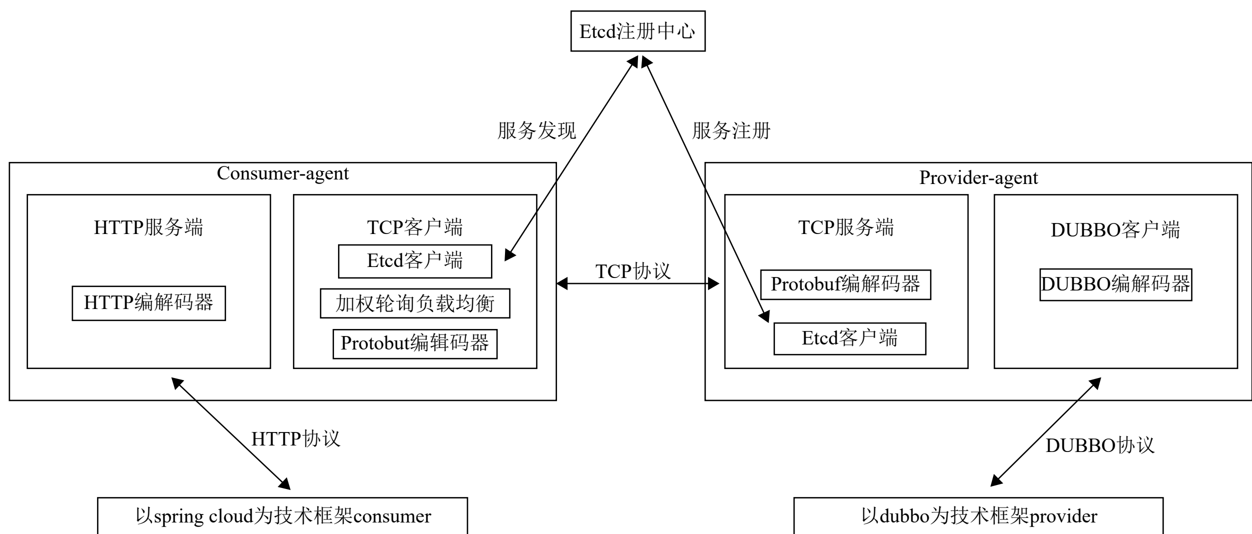

图 3　网络代理实现机制

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.

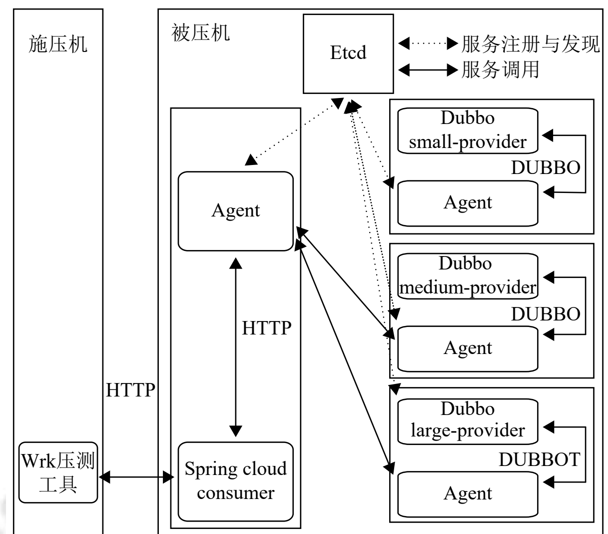

图 4 实验场景

(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

© 中国科学院软件研究所 http://www.c-s-a.org.cn

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

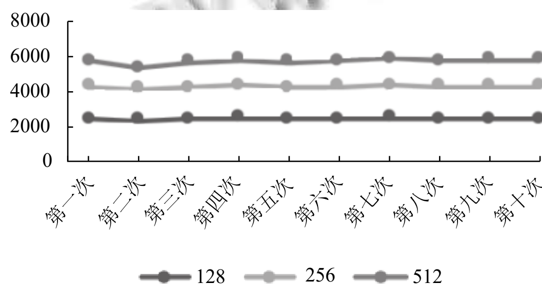

图 6　压测 10 次实验结果

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.

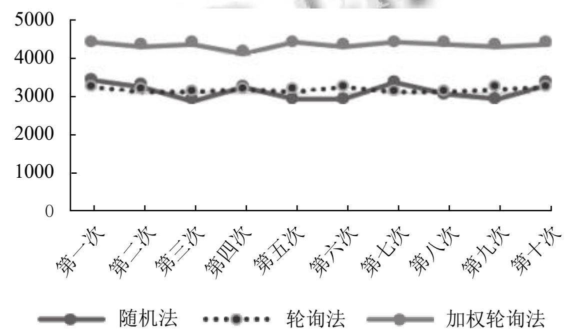

图 7　不同负载均衡算法实验结果

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.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

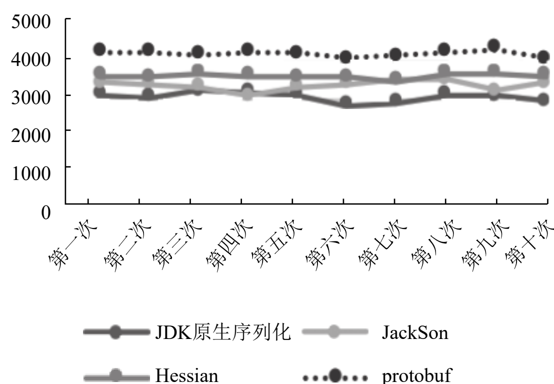

图 8　不同序列化方式的实验结果

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]

© 中国科学院软件研究所 http://www.c-s-a.org.cn