B站高可用架构的实践

Recommend
火龙果软件

目录

一、 负载均衡

二、 限流

三、 重试

四、 超时

五、 应对连锁故障

Recommend
火龙果软件

一、负载均衡

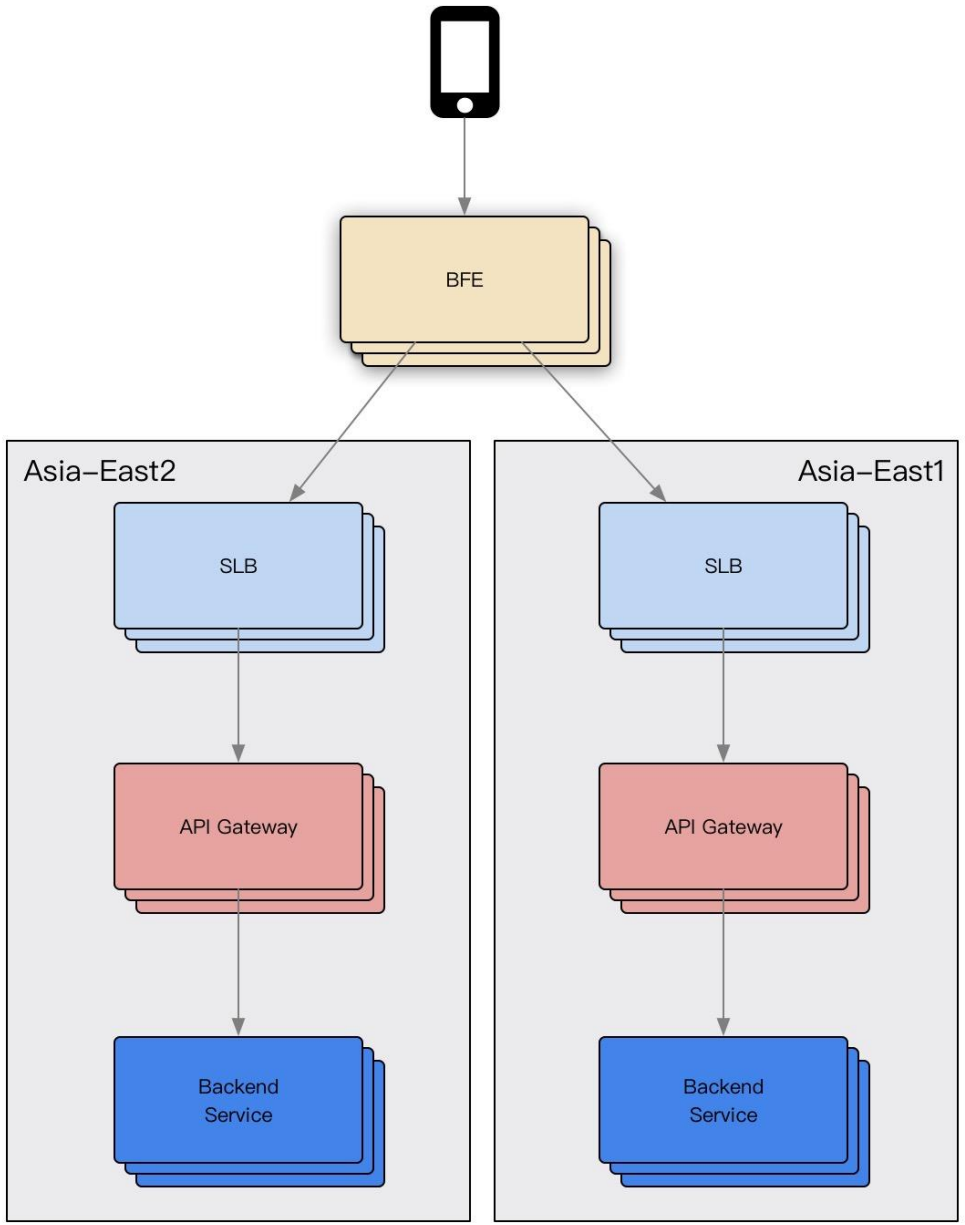
前端服务器的负载均衡

用户流量应该最优地分布于多个网络链路上、多个数据中心中、以及多台服务器上。

最优通常值得是：

- 最小化用户的请求延迟；
- 低于带宽的95线峰值；
- 基于可用服务容量平衡流量；

需要全局GSLB负载均衡进行数据中心间的流量调度。



Recommend
火龙果软件

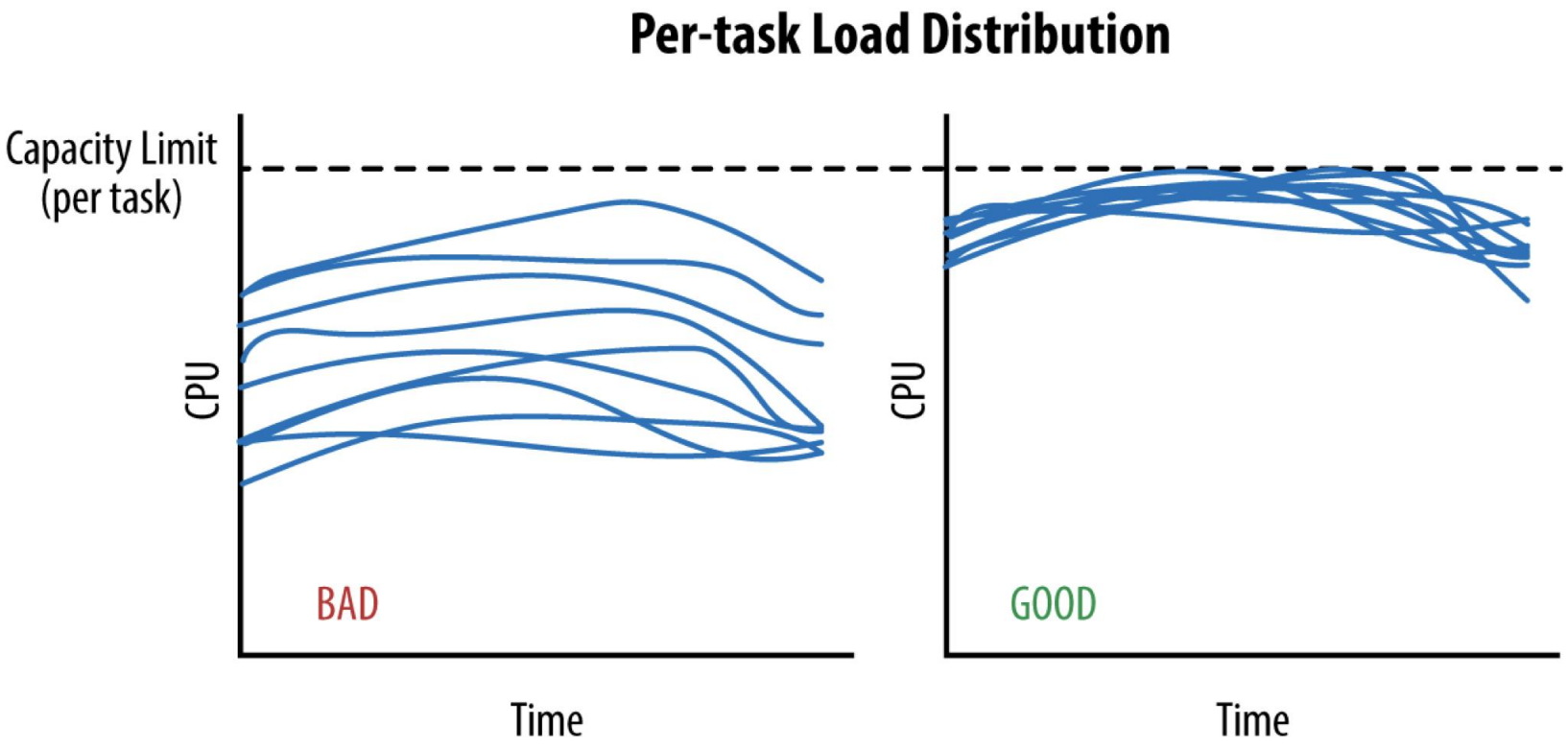
一、负载均衡

数据中心内部的负载均衡

在理想情况下，某个服务的负载会完全均匀地分发给所有的后端任务。在任何时刻，最忙和最不忙的节点永远消耗同样数量的CPU。

目标：

- 均衡的流量分发；
- 可靠的识别异常节点；
- scale-out，增加同质节点扩容；
- 减少错误，提高可用性；



Recommend
火龙果软件

一、负载均衡

子集选择和多集群

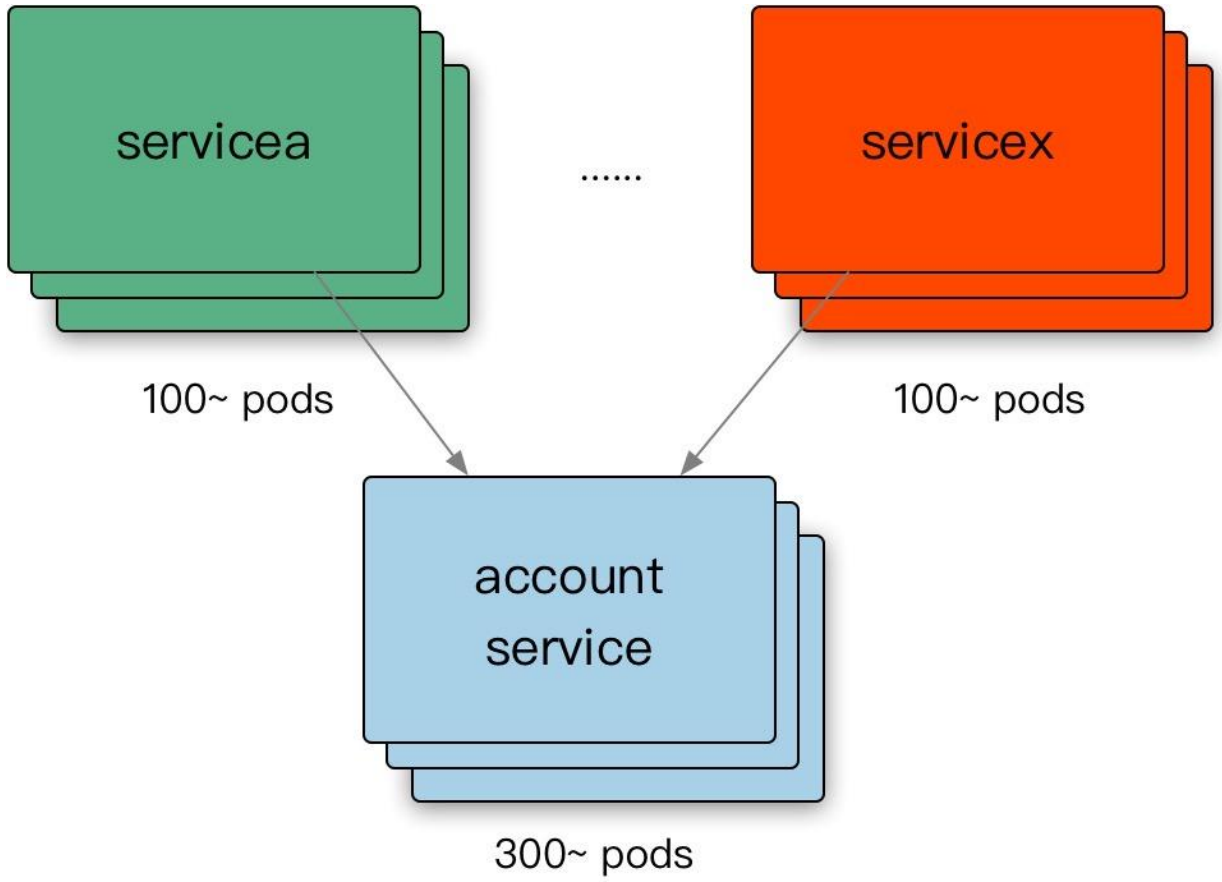
节点之间需要保持心跳（healthcheck），client需要剔除end-to-end有故障的backend节点。

scale out能力会随着client变多而带来性能退化，以及资源开销：

- 连接数：clients * backends；
- 心跳开销变的巨大，空闲时可能高达30%；

同时，处于底层的服务可能意味着有越高的重要性（L0级别）：

- 多集群（冗余的backends、缓存等等）；



一、负载均衡

子集选择和多集群

实现client到backend的子集连接：

- 将后端平均分配给客户端，同时可以处理节点的变更，持续不断的均衡连接，同时避免大幅变动；
- 客户端数量相比后端数量少很多，需要使用大子集；
- 有突发性的大并发请求，需要更大的子集分摊流量压力，通常更建议进行集群的隔离；

多集群：

- 需要考虑集群迁移的运维成本；
- 集群之间业务的数据存在较小的交集；

A Subset Selection Algorithm: Deterministic Subsetting

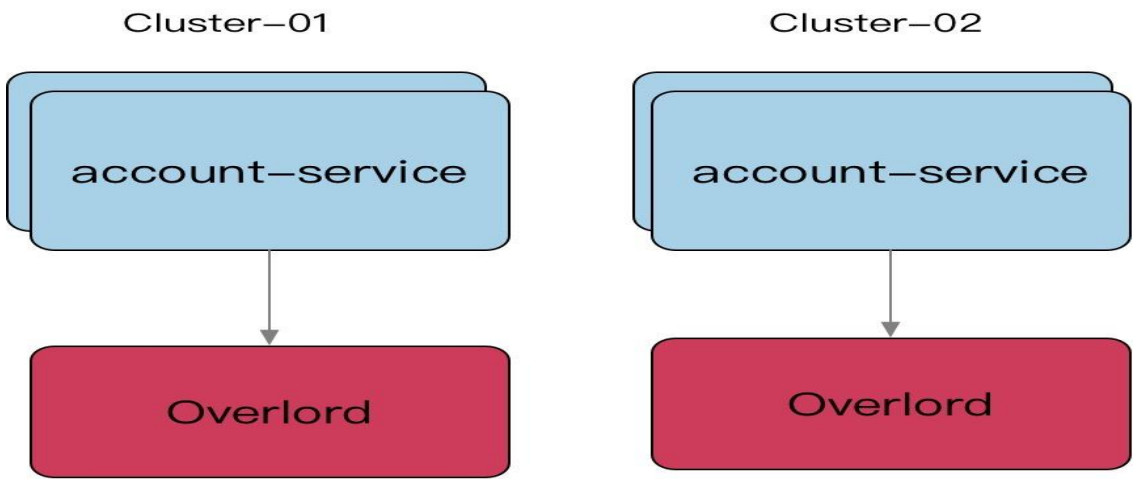
Google's solution to the limitations of random subsetting is *deterministic* subsetting. The following code implements this algorithm, described in detail next:

```
def Subset(backends, client_id, subset_size):
  subset_count = len(backends) / subset_size

  # Group clients into rounds; each round uses the same shuffled list:
  round = client_id / subset_count
  random.seed(round)
  random.shuffle(backends)

  # The subset id corresponding to the current client:
  subset_id = client_id % subset_count

  start = subset_id * subset_size
  return backends[start:start + subset_size]
```



Recommend
火龙果软件

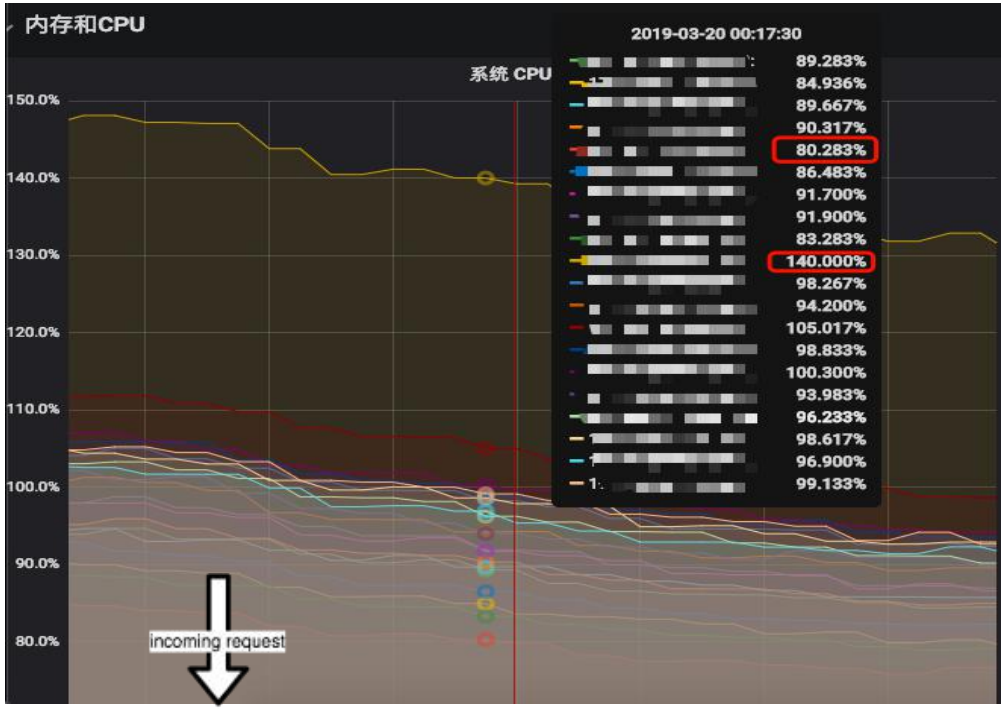
一、负载均衡

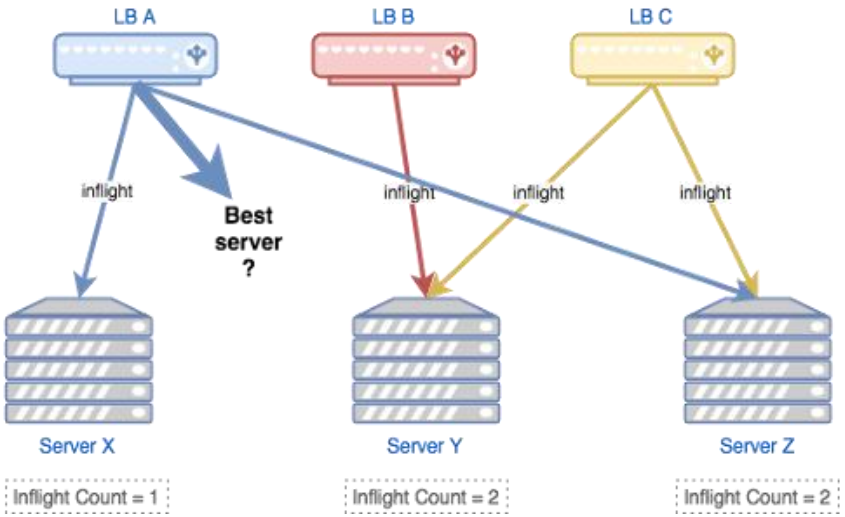
负载均衡算法

我们发现在backend之间的load差异比较大：

- 每个请求的处理成本不同；
- 物理机环境的差异
  - 服务器很难强同质性；
  - 存在共享资源争用（内存缓存、带宽、IO等）；
- 性能因素：
  - FullGC；
  - JVM JIT；

参考JSQ（最闲轮训）负载均衡算法带来的问题，缺乏的是服务端全局视图，因此我们目标需要综合考虑：负载+可用性。





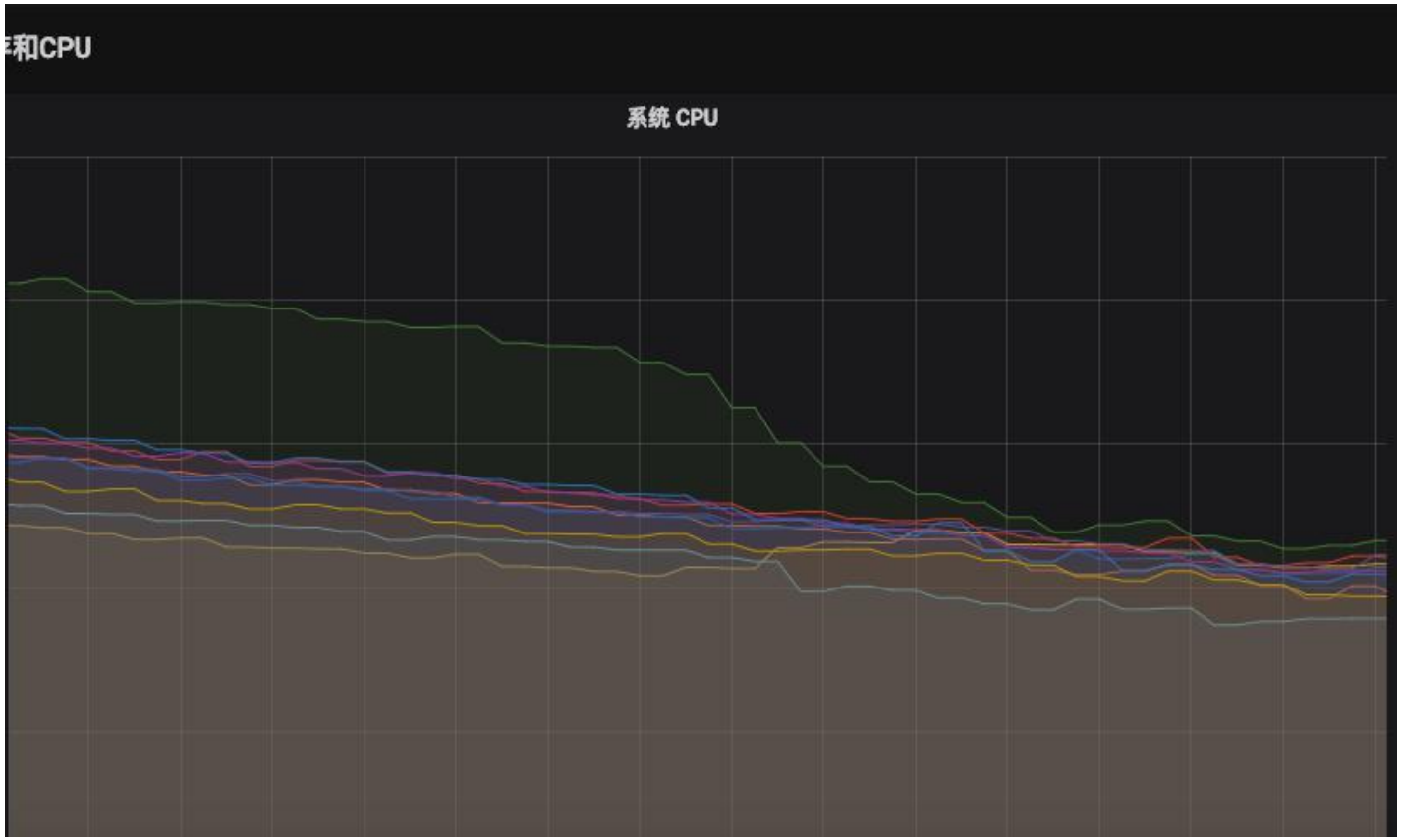
Recommend
火龙果软件

一、负载均衡

负载均衡算法

参考了《The power of two choices in randomized load balancing》的思路，我们使用the choice-of-2算法，随机选取的两个节点进行打分，选择更优的节点：

- 选择backend：CPU，client：health、inflight、latency作为指标，使用一个简单的线性方程进行打分；
- 对新启动的节点使用常量惩罚值（penalty），以及使用探针方式最小化放量，进行预热；
- 打分比较低的节点，避免进入“永久黑名单”而无法恢复，使用统计衰减的方式，让节点指标逐渐恢复到初始状态（即默认值）；



目录

一、 负载均衡

二、 限流

三、 重试

四、 超时

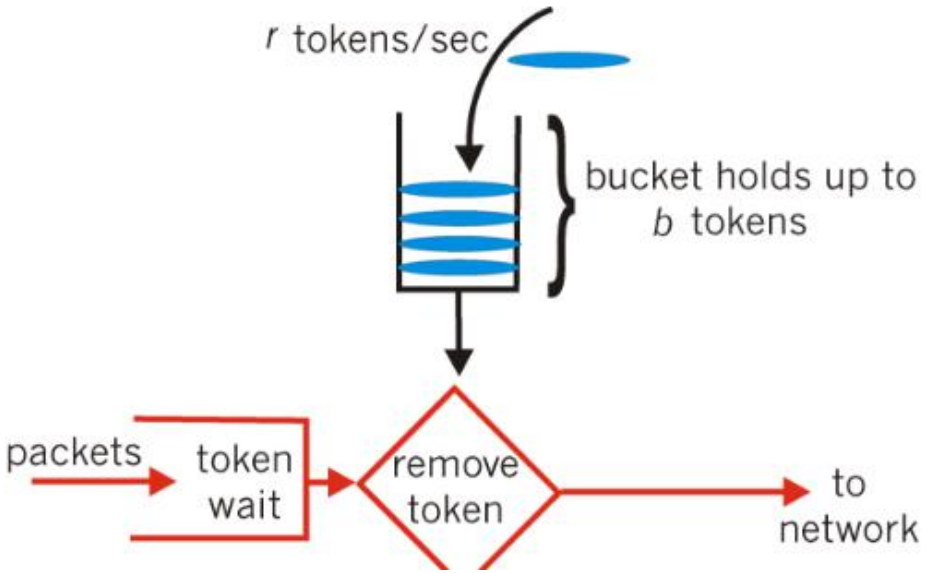
五、 应对连锁故障

Recommend
火龙果软件

限流的要点

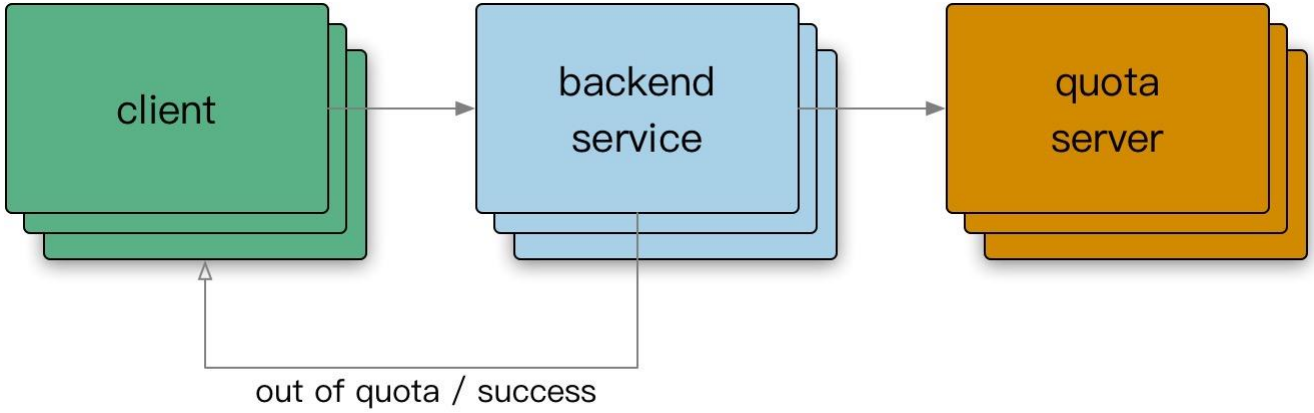
避免过载，是负载均衡的一个重要目标。随着压力增加，无论负载均衡策略如何高效，系统某个部分总会过载。我们优先考虑优雅降级，返回低质量的结果，提供有损服务。在最差的情况，妥善的限流来保证服务本身稳定。

- QPS陷阱
  - 请求成本不同；
  - 静态阈值难以配置（思考：基于CPU售卖给不同服务？）；
- 按照优先级丢弃；
- 给每个用户设置限制
  - 全局过载发生时候，针对某些“异常”进行控制非常关键；
- 拒绝请求也需要成本；
- 每个服务都配置限流带来的运维成本；



Recommend
火龙果软件

分布式限流：Quota Server



我们实现一个quota-server，用于给backend针对每个client进行控制，即backend需要请求quota-server获取quota。

- 获取一批Quota（大于1），用完再申请，可以大大减少请求Server的频次，获取完以后本地消费；
- 基于过去时间的滑动窗口内的最大值作为inflight来申请Quota；
- 使用最大最小公平算法（Max-Min Fairness），解决某个大消耗者导致的饥饿
  - 资源按照需求递增的顺序进行分配；
  - 不存在用户获得的资源超过自身的需求；
  - 对于未满足的用户，等价分享剩余资源；

	A	B	C	D	
资源需求	2	2.6	4	5	
	2.5	2.5	2.5	2.5	10/4
	2	2.666	2.666	2.666	(2.5-2)/3+2.5
	2	2.6	2.7	2.7	(2.666-2.6)/2+2.666

Recommend
火龙果软件

二、限流

客户端测截流

- 某个用户超过资源配额时，后端任务会快速拒绝请求，返回“配额不足”的错误，有可能后端忙着不停发送拒绝请求，导致过载；
- 依赖的资源出现大量错误，处于对下游的保护；

上述两种情况下，我们在client测直接进行流量抛弃，而不发送到网络层，我们使用公式：

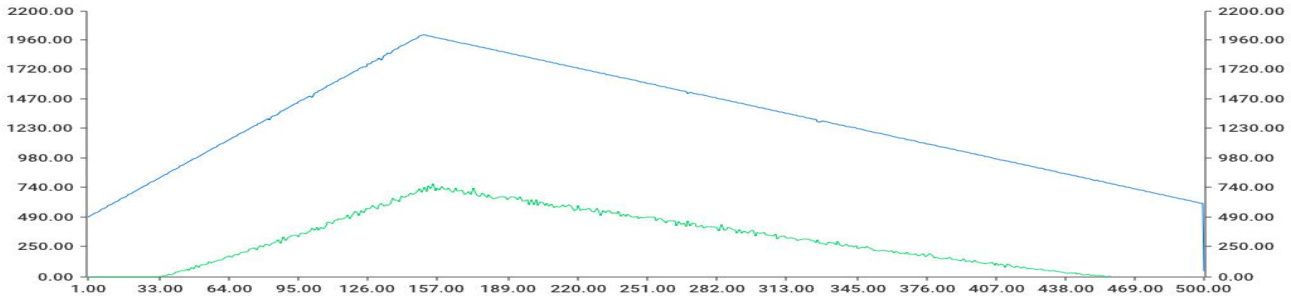
max(0, (requests- K*accepts) / (requests + 1))

client可以发送请求直到requests = K * accepts，一旦超过限制，按照概率进行截流。

```
func (b *sreBreaker) Allow() error {
    success, total := b.stat.Value()
    k := b.k * float64(success)

    if total < b.request || float64(total) < k {
        return nil
    }
    dr := math.Max(0, (float64(total)-
k)/float64(total+1))
    rr := b.r.Float64()

    if dr <= rr {
    return nil
    }
    return ecode.ServiceUnavailable
}
```



Recommend
火龙果软件

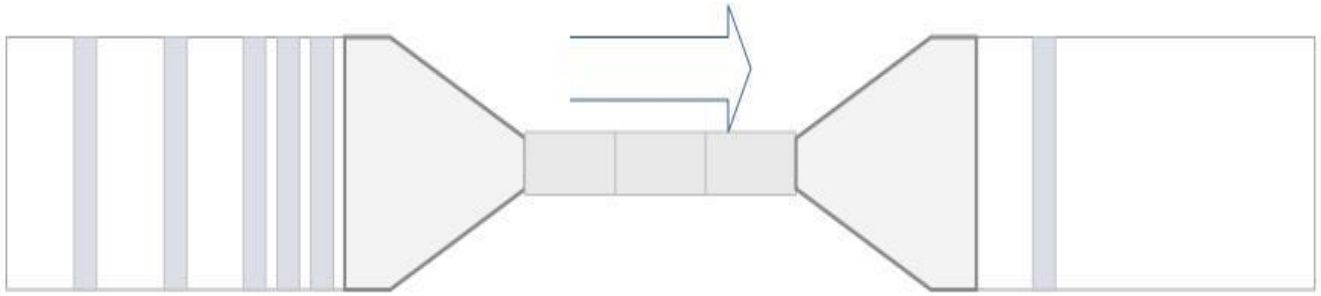
过载保护

过载保护核心目标是在服务过载时，通过丢弃一定的流量，保证系统临近过载时的峰值流量，已求自保护。

常见的做法：

- CPU、内存使用量进行节流；
- 队列管理：队列长度、LIFO；
- 可控延迟算法：CoDel；

参考TCP BBR探测的思路：计算系统临近过载时的峰值吞吐作为限流的阈值来进行流量控制，达到系统保护。



二、限流

过载保护

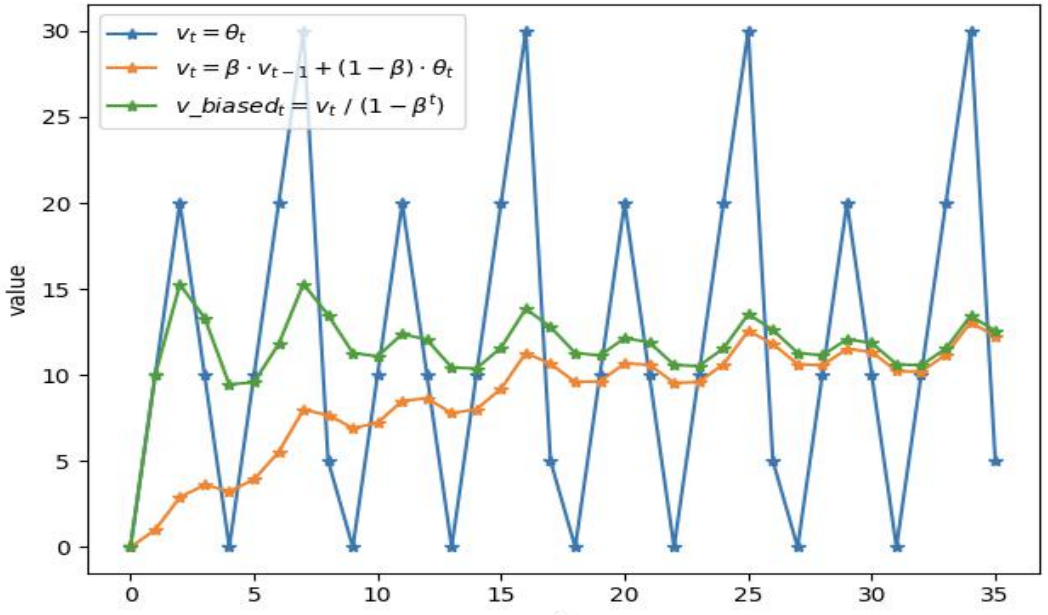
我们使用CPU的滑动均值（CPU > 800 ）作为启发阈值，一旦触发进入到过载保护阶段，算法为：

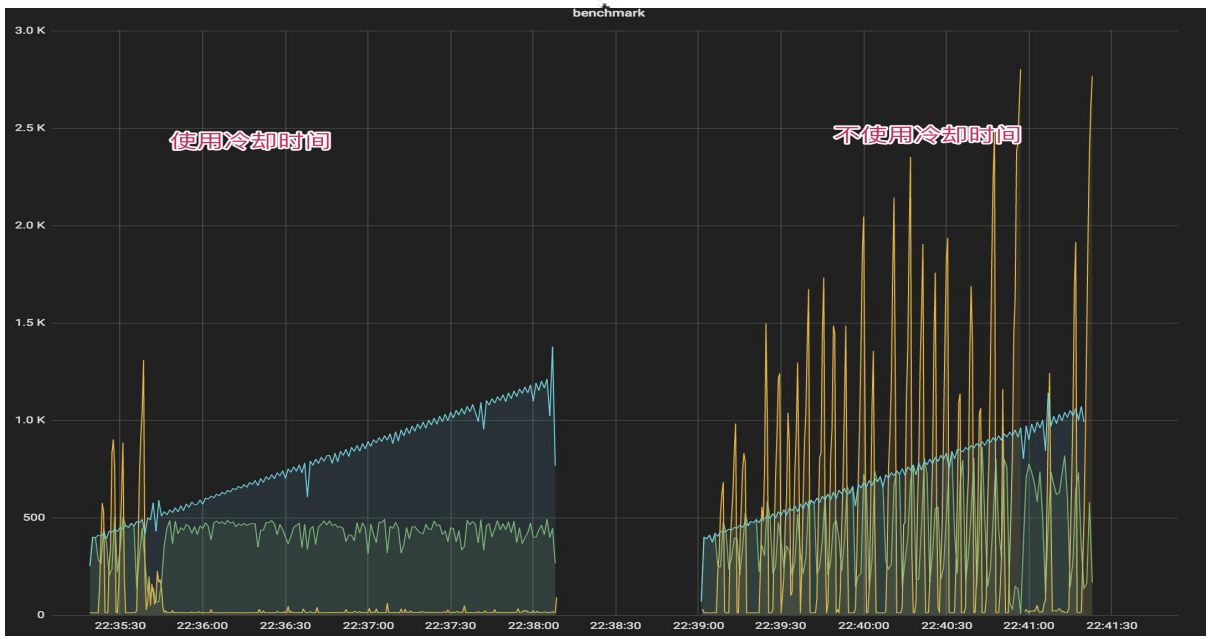
(MaxPass * AvgRT) < InFlight

其中MaxPass、AvgRT都为触发前的滑动时间窗口的统计值。

限流效果生效后，CPU会在临界值（800）附近抖动，如果不使用冷却时间，那么一个短时间的CPU下降就可能导致大量请求被放行，严重时会打满CPU。

在冷却时间后，重新判断阈值（CPU > 800 ），是否持续进入过载保护。





Recommend
火龙果软件

目录

一、 负载均衡

二、 限流

三、 重试

四、 超时

五、 应对连锁故障

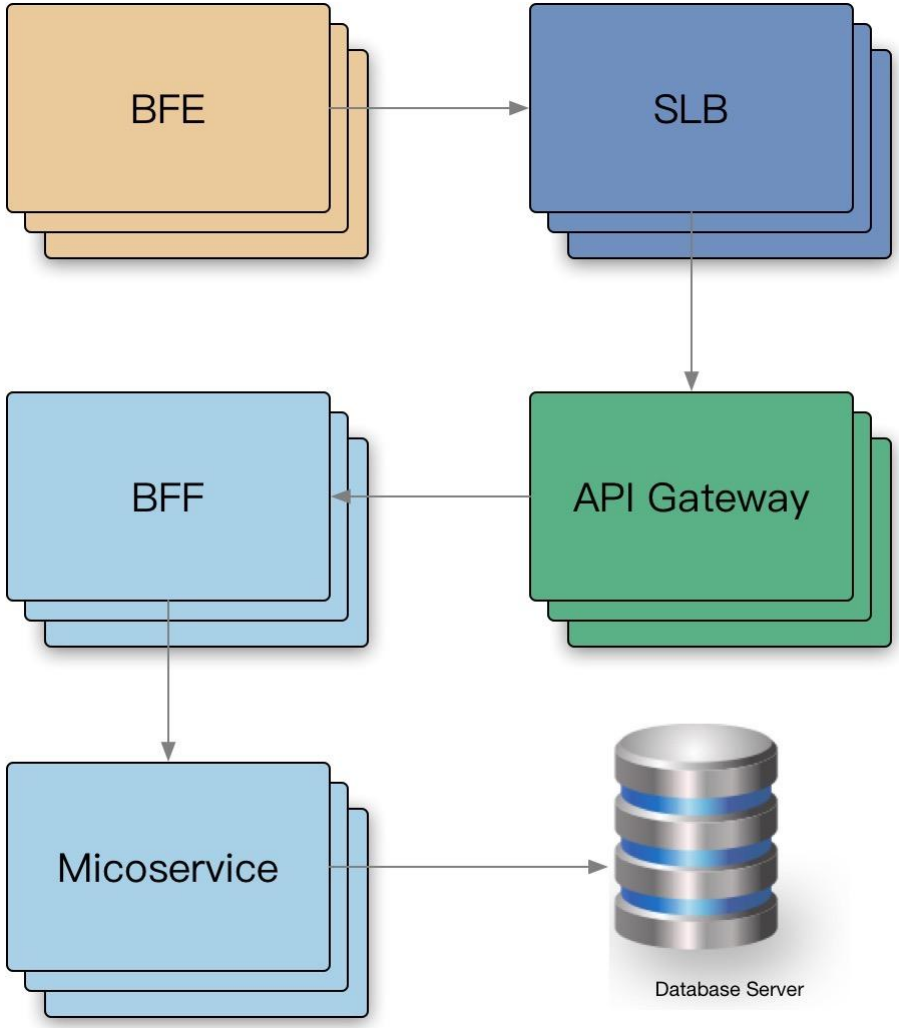
Recommend
火龙果软件

三、重试

重试策略

当请求返回错误（例：配额不足、超时、内部错误等），对于backend部分节点过载的情况下，倾向于立刻重试，但是需要留意重试带来的流量放大：

- 限制重试次数和基于重试分布的策略（重试比率：10%）；
- 只应该在失败的这层进行重试，当重试仍然失败，全局约定错误码“过载，无须重试”，避免级联重试；
- 随机化、指数型递增的重试周期：Exponential Backoff + Jitter；
- 重试速率指标，用于诊断故障；

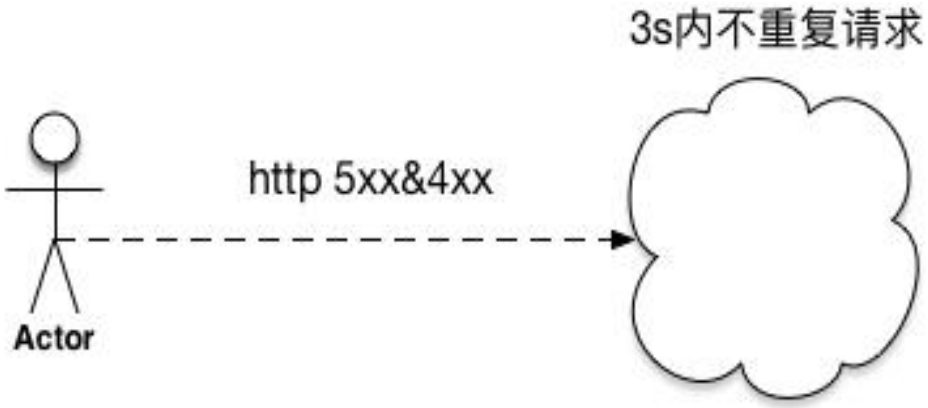


客户端限速

positive feedback：
用户总是积极重试，访问一个不可达的服务。

客户端需要限制请求频次，Backoff做一定的请求退让，可以通过接口级别的error_details，挂载到每个API返回的响应里。

```
{
  "error": {
    "code": 401,
    "message": "Request had invalid credentials.",
    "status": "UNAUTHENTICATED",
    "details": [{
      "@type": "type.googleapis.com/google.rpc.RetryInfo",
      ...
    }]
  }
}
```



Recommend
火龙果软件

目录

一、 负载均衡

二、 限流

三、 重试

四、 超时

五、 应对连锁故障

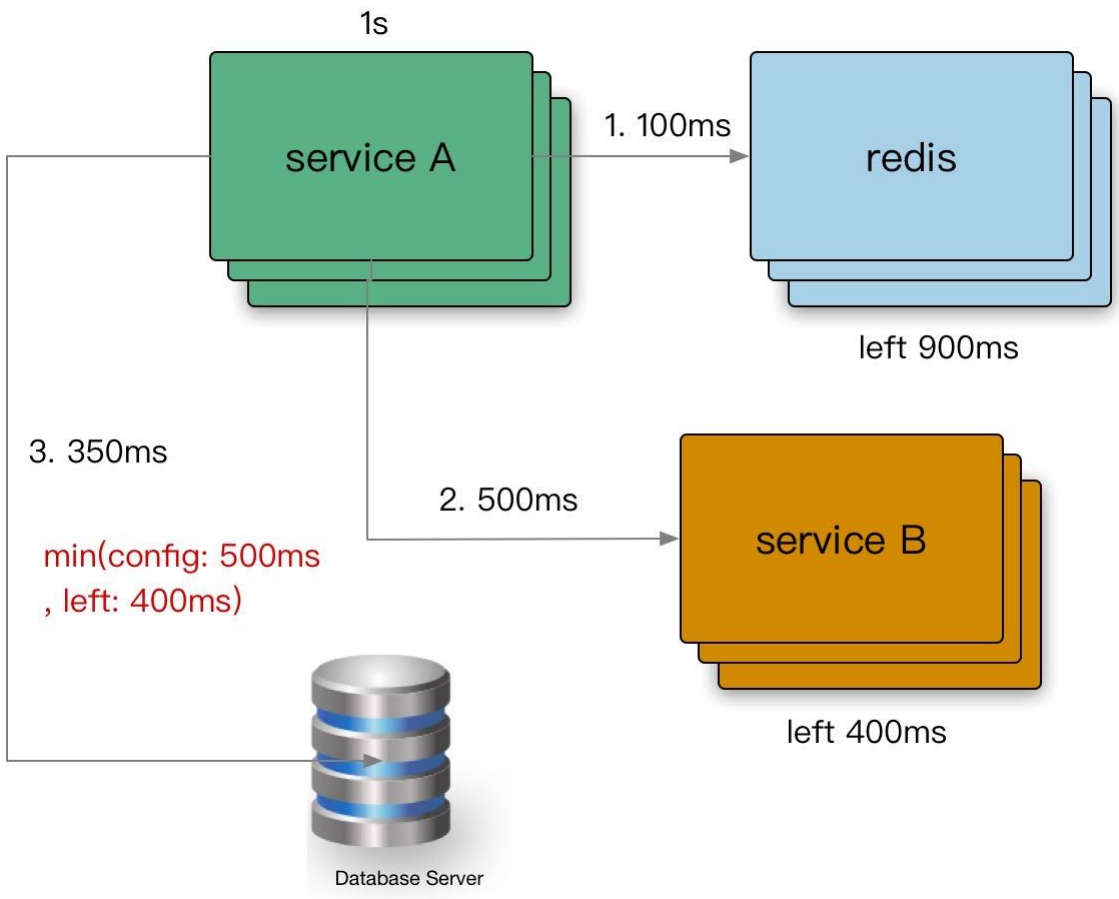
Recommend
火龙果软件

四、超时

进程内超时控制

- 高延迟服务导致Client浪费资源等待；
- 超时是一种Fail Fast策略；
- 客户端和服务端不一致的超时策略导致资源浪费；
- “默认值”策略；

一个请求在每个阶段（网络请求）开始前，就要检查是否还有足够的剩余来处理请求。

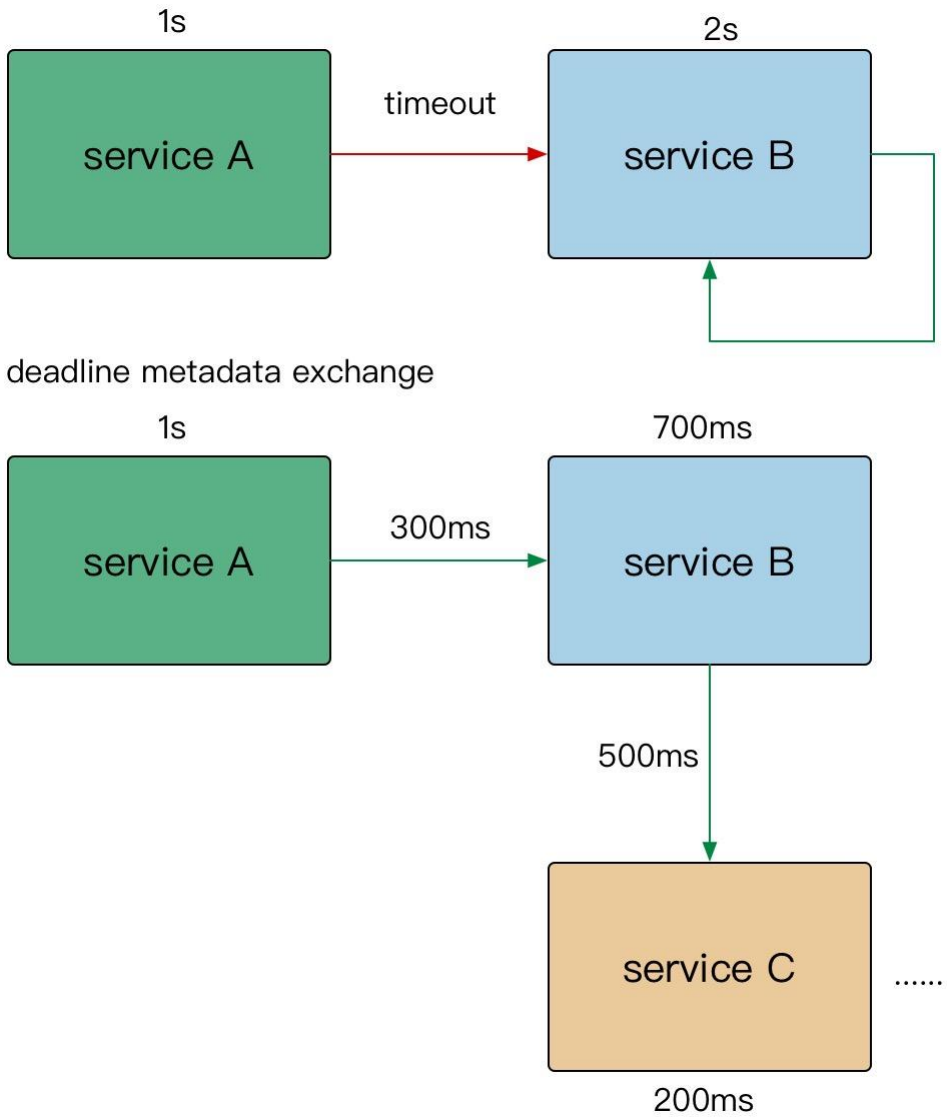


四、超时

跨进程超时控制

1. A发送RPC到B，1s超时；
2. B使用了300ms处理请求，再转发请求C；
3. C配置了600ms超时，但是实际只用了500ms；
4. 到其他的下游，发现余量不足，取消传递；

下游的RPC可以覆盖上游的超时传递和配额。



Recommend
火龙果软件

目录

一、 负载均衡

二、 限流

三、 重试

四、 超时

五、 应对连锁故障

Recommend
火龙果软件

五、应对连锁故障

- 避免过载
- 优雅降级
- 重试退避
- 超时控制
- 变更管理
- 极限压测 + 故障演练
- 扩容+重启+消除有害流量

References

- Google - Site Reliability Engineering
- The Site Reliability Workbook
- Rethinking Netflix’s Edge Load Balancing
- Performance Under Load
- Introducing QALM, Uber’s QoS Load Management Framework
- alibaba Sentinel
- Fail at Scale