GopherChina2018

Go在Grab地理服务中的实践

张志印

大纲

- What’s Grab
- 一个典型的派单流程
- 一个核心地理服务系统演进历程
- Why go
- 压测与调优
- QA

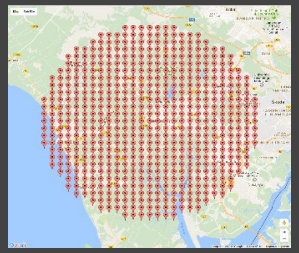
What's Grab

东南亚最大的出行平台

连接供(Supply,司机)需(Demand,乘客)

从派单流程说起

派单流程



找附近司机	过滤及分配	锁定司机	通知司机
调用Nearby Service获取附近司机	多层过滤 选出一个司机： 1) ETA/ETD 2) 各种派单策略	1) 加锁保证任意时刻同一司机只能被分配 给一个行程 2) 若加锁失败 会重新进行 Nomination(Recycle)	1) 最终获得一个司机 资源 通过长连接推送任务到他的APP 2) 如果司机忽略 该行程 重新进入 Nomination(Recycle)

如何构建一个高可用Nearby服务?

要解决的问题

- 写(Update Location)
  - 司机坐标秒级实时上传 高并发写
  - 司机坐标动态变化
- 读(Nearby Search)
  - 查找附近的司机
  - 司机坐标具有时效性
- 可用性要求高
  - 不能宕机
- 性能要好
  - 读写都要低延时

Nearby服务的演进历程

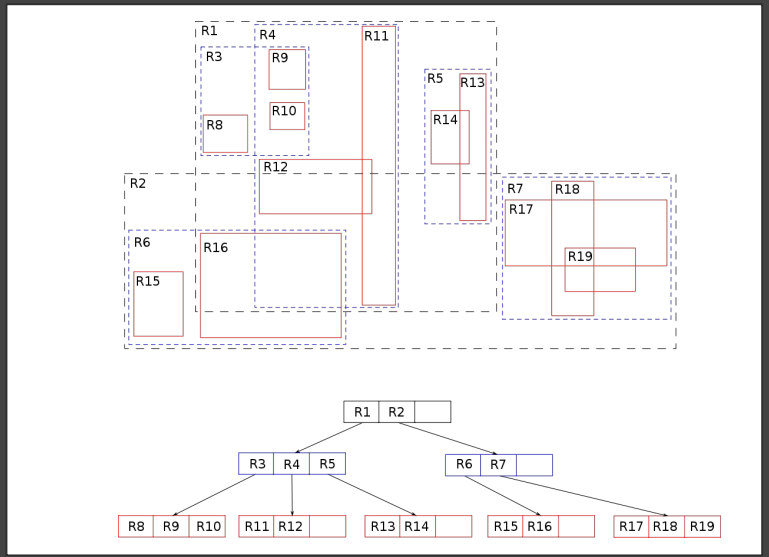
我们的目标

我们从这里起步

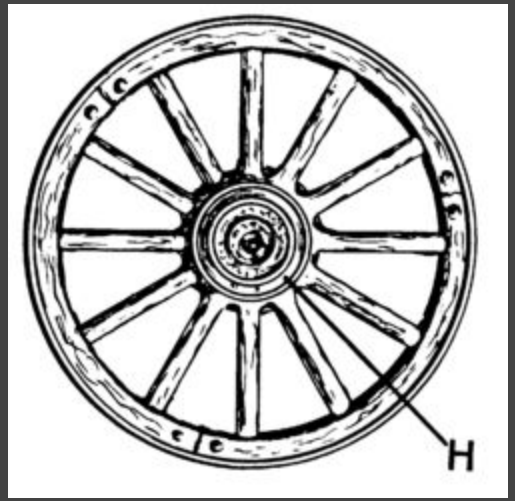
NearbyV1 - 最初版本

- Node.js + PostGIS
- 缺点
  - 写性能低
    - 约几百毫秒
    - 基于R-Tree
      - 空间数据存储 一种平衡树
      - 高频写 触发频繁再平衡
  - 可用性低
    - 高峰时服务经常崩掉

可用性完全不可接受

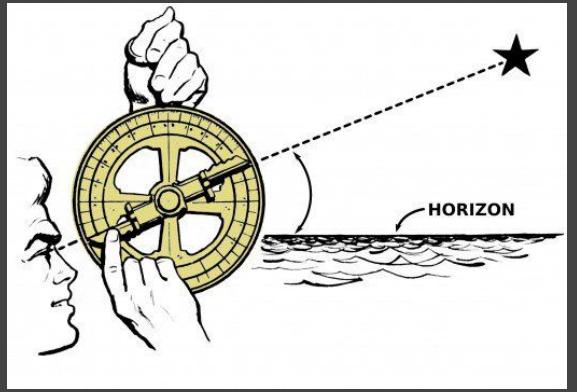


有节操的程序猿都是在造轮子



NearbyV2 - Astrolabe

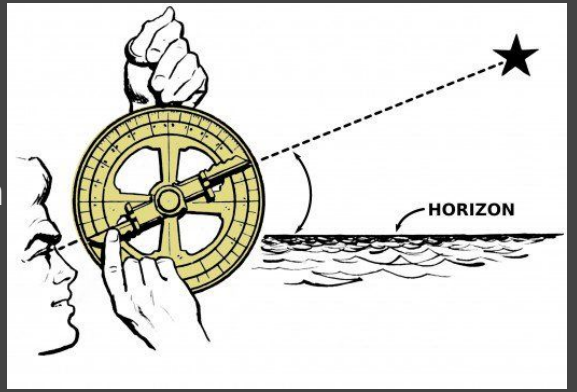
- 脱胎于我们内部的一次Hackathon
  - R-Tree => Geohash
  - Node.js => Golang
  - PostGIS => Redis
- Geohash特点
  - http://geohash.org/wtw6j89guz9y
    - 坐标: 31.292494, 121.533371
  - 使用一维字符串表示二维坐标数据
  - 具有相同前缀的Geohash字符串地理位置相近
    - 邻近搜索(Nearby)
      - 基于SortedSet作Range查询



How to make one Astrolabe

NearbyV2 - Astrolabe

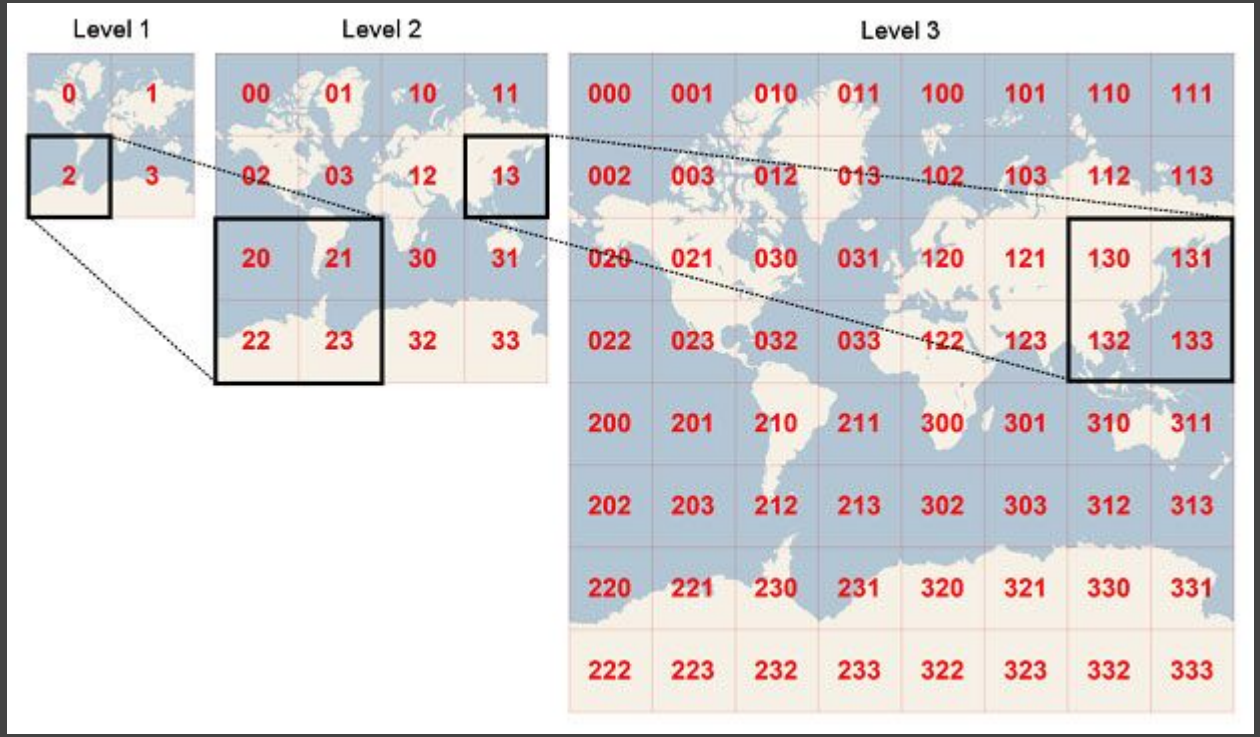
- 写
  - 计算Geohash
  - 按城市将Geohash存储到Redis的zset中
- 读
  - 计算MBR(最小外接矩形) 获取左下/右上两个位置的Geohash
  - 基于Redis zset range粗略获取一批司机
  - 计算距离找到附近的司机
- 解决了哪些问题
  - 写性能提高了
    - ~10ms
  - 可用性更高了 很少宕机



How to make one Astrolabe

Astrolabe - 主要问题

- Nearby准确度低
  - Geohash在赤道附近出现跳变



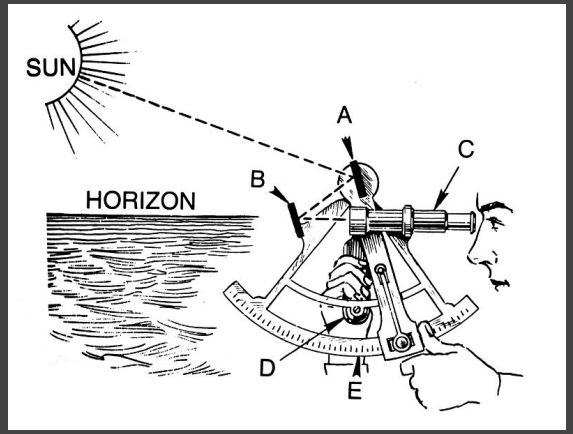
Astrolabe - 主要问题

- 准确度低
  - Geohash在赤道附近出现跳变
- 很难横向扩展
  - 基于Geohash有序性的Range查询决定了每个城市属于一个独立 SortedSet
  - 但司机人数在不快速增长 瓶颈日益显现
- CPU/IO非常高 内存非常低

不能横向扩展 成为业务发展的主要瓶颈

造一个更好的轮子

NearbyV3 - Sextant



回顾 - 要解决的问题

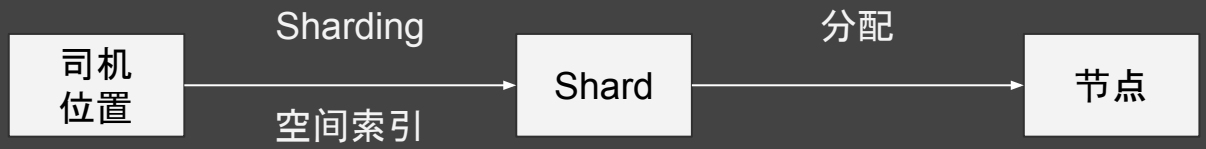
- 业务高速增长
  - 快速横向扩容
- 写(Update Location)
  - 上百万司机坐标秒级实时上传
  - 司机坐标动态变化
- 读(Nearby Search)
  - 查找附近司机
- 可用性要求高
  - 不能宕机
- 性能要好
  - 读写都要低延时 即使在流量高峰

从扩容说起

GopherChina2018

要扩容 先Sharding

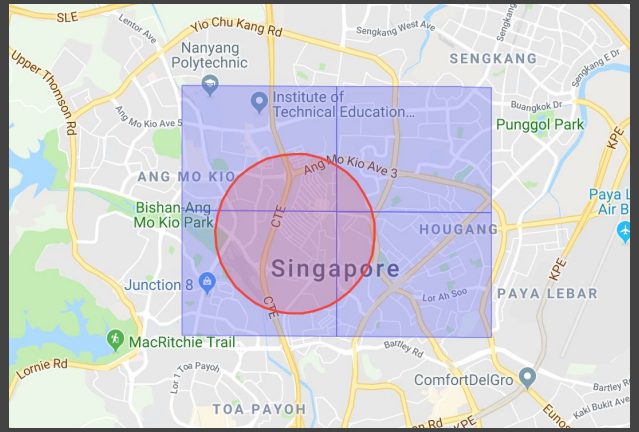
- Sharding就是基于空间索引将不同的司机分布到不同的服务器节点上

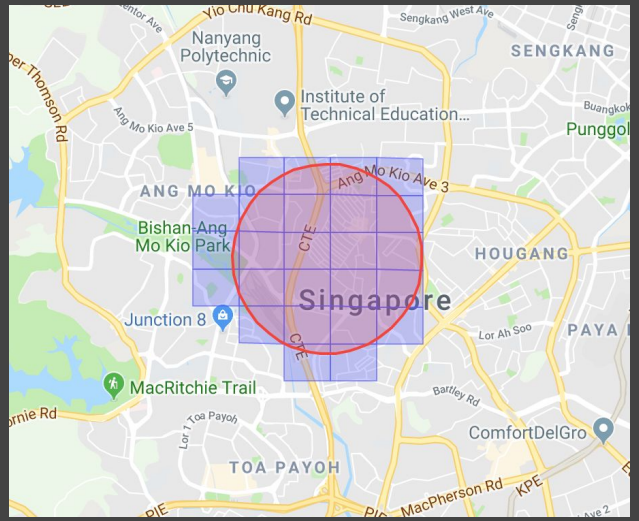


- 司机位置通过空间索引映射到某一个Shard里 最终被分配到服务器节点上
- Shard完全在内存中
- 每个Shard运行在至少2个节点上

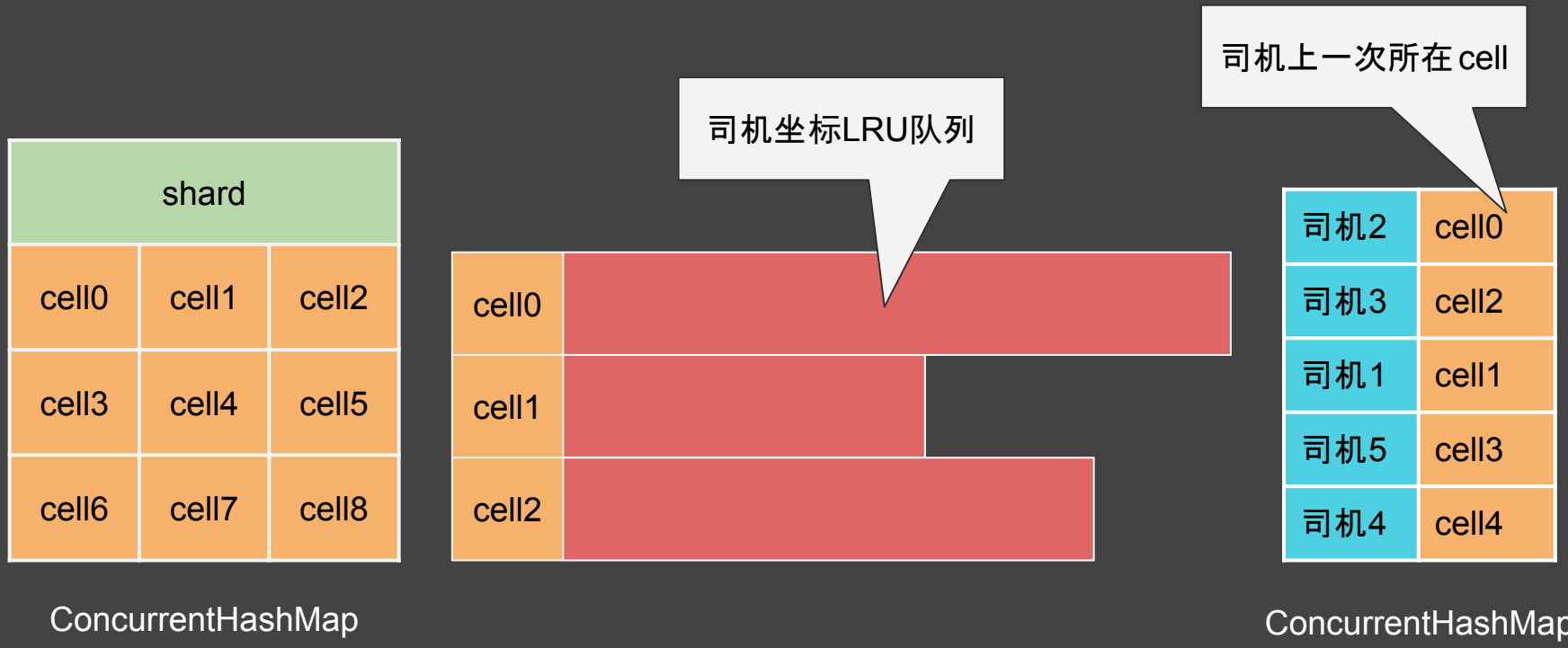
Sharding策略 - 空间索引

- 地球平面被分成四边形的Grid
  - 每Grid ID由经纬度和Resolution生成
- 分层Grid
  - 地理空间被Shard铺满
  - 一个Shard内有若干Cell
- 特点
  - 任意一个经纬度都可以算出所在的Shard和Cell



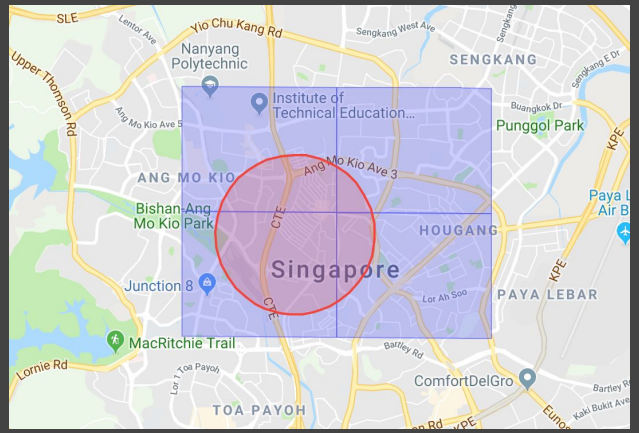


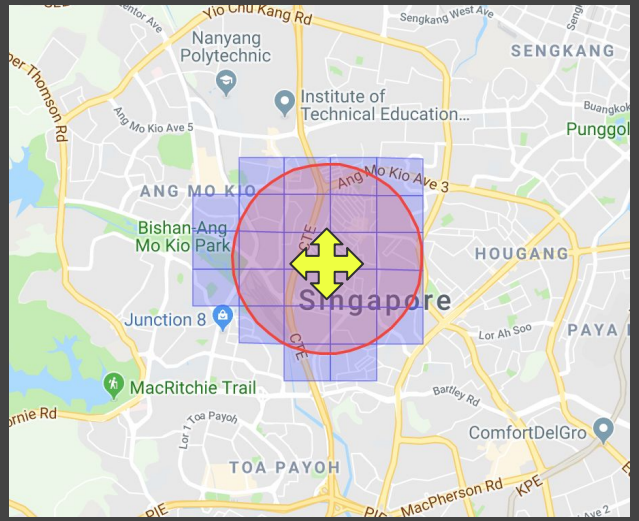
Sharding策略 - Shard内存数据结构



Sharding策略 - 读/写

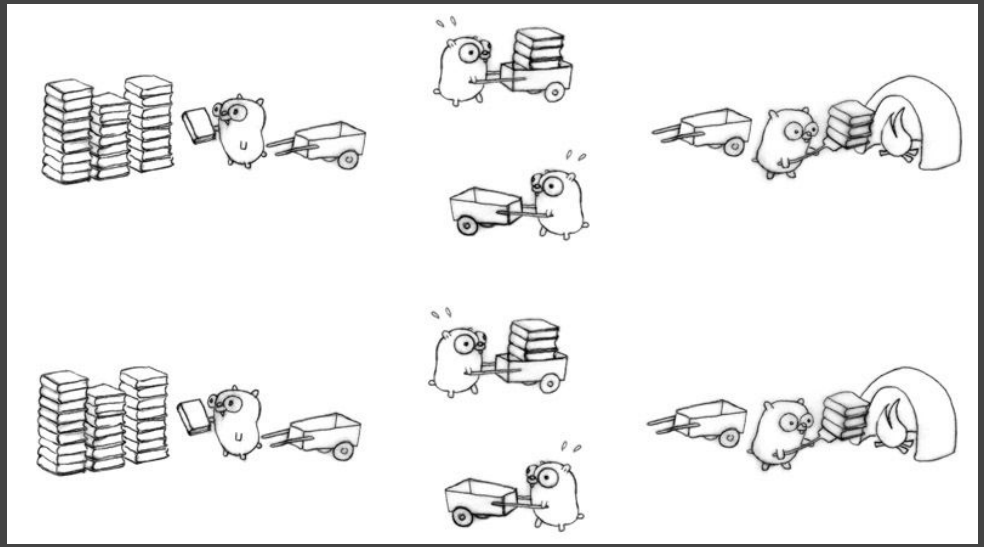
- 写(Update Location)
  - 根据经纬度计算出Shard ID及Cell ID
  - 若上一次在同一个Cell
    - 直接更新当前Cell里的LRU队列
  - 若上一次在不同的Cell
    - 先从上一次Cell里删除
    - 再加入当前Cell的LRU队列
- 读(Nearby Search)
  - 计算相交Shard
  - 对每个Shard做并行Nearby计算
    - 按广度优先查找相应的Cell
    - 过滤出Cell中符合条件的司机
  - 聚合，取Top N





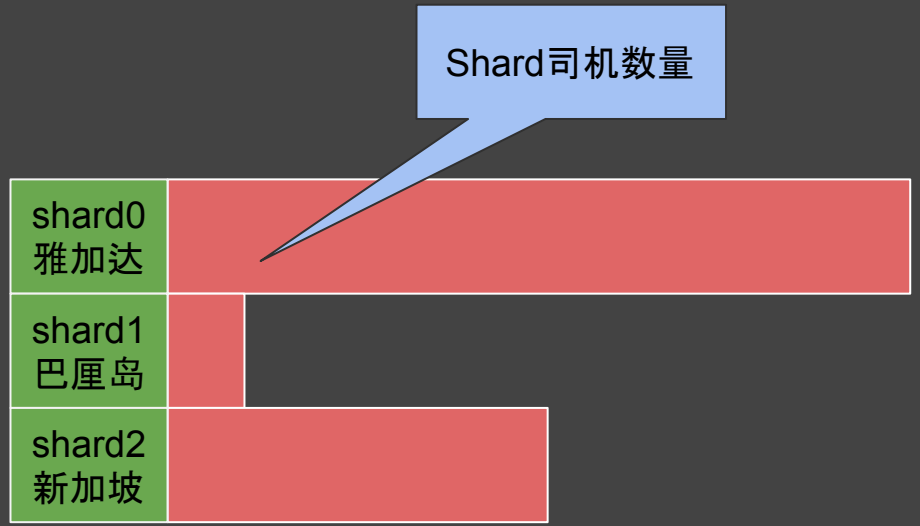
Sharding策略 - 清理

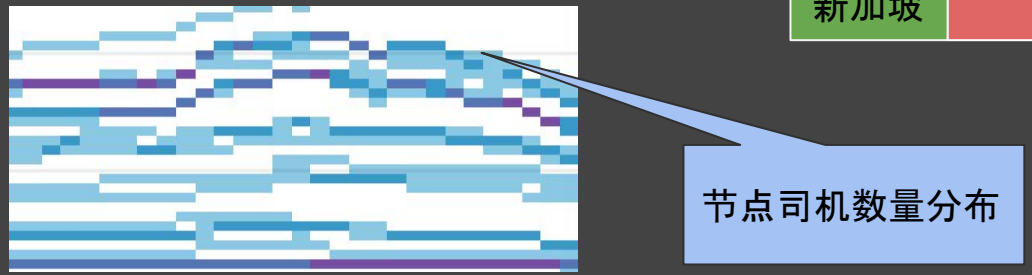
- 清理过期数据
  - 各种原因如网络问题 司机可能会遗留一个过期坐标在某个Cell里
  - 专有的Goroutine去清理Cell里过期数据



Sharding策略 - Shard分配策略

- 一致性哈希
  - Ketema algorithm
  - 支持Replica
- 写的问题
  - 不同的Shard司机数量不均衡
  - 简单按Shard哈希分配造成司机在 节点上分布极度不均衡

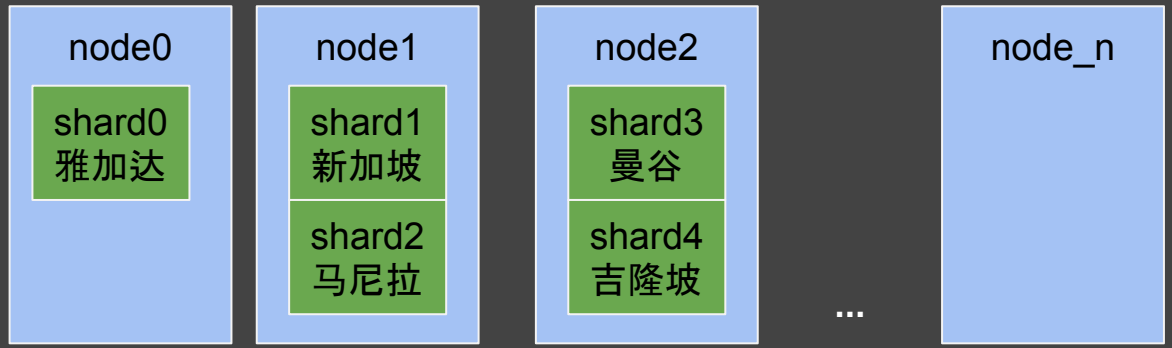




Sharding策略 - Shard分配策略

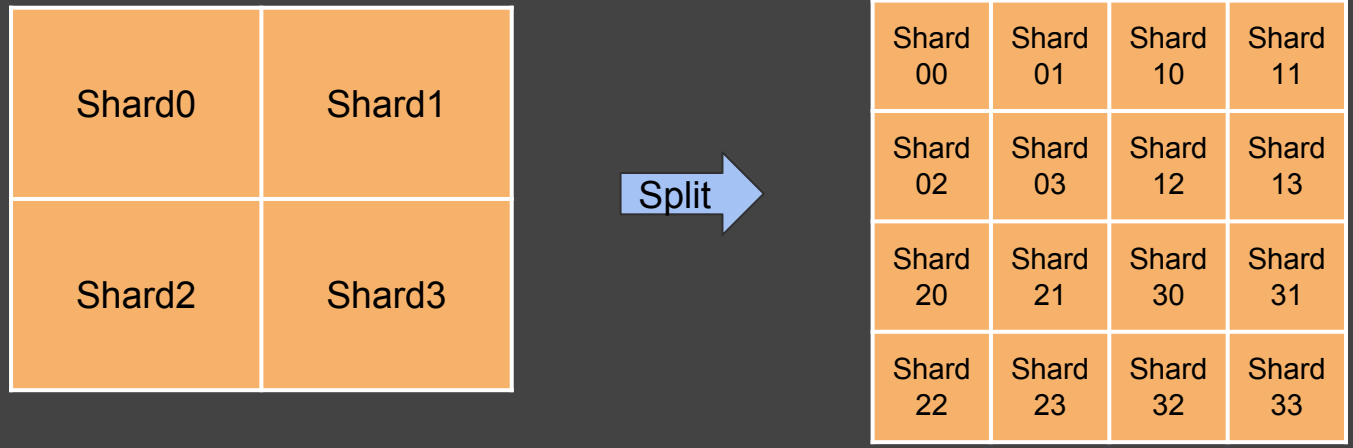
- ShardTable
  - 针对大Shard进行定制化 按权重 更均衡
  - 一致性哈希仍然是有益的 补充
    - 司机位置不断变化 可能产生新Shard

节点司机数量分布



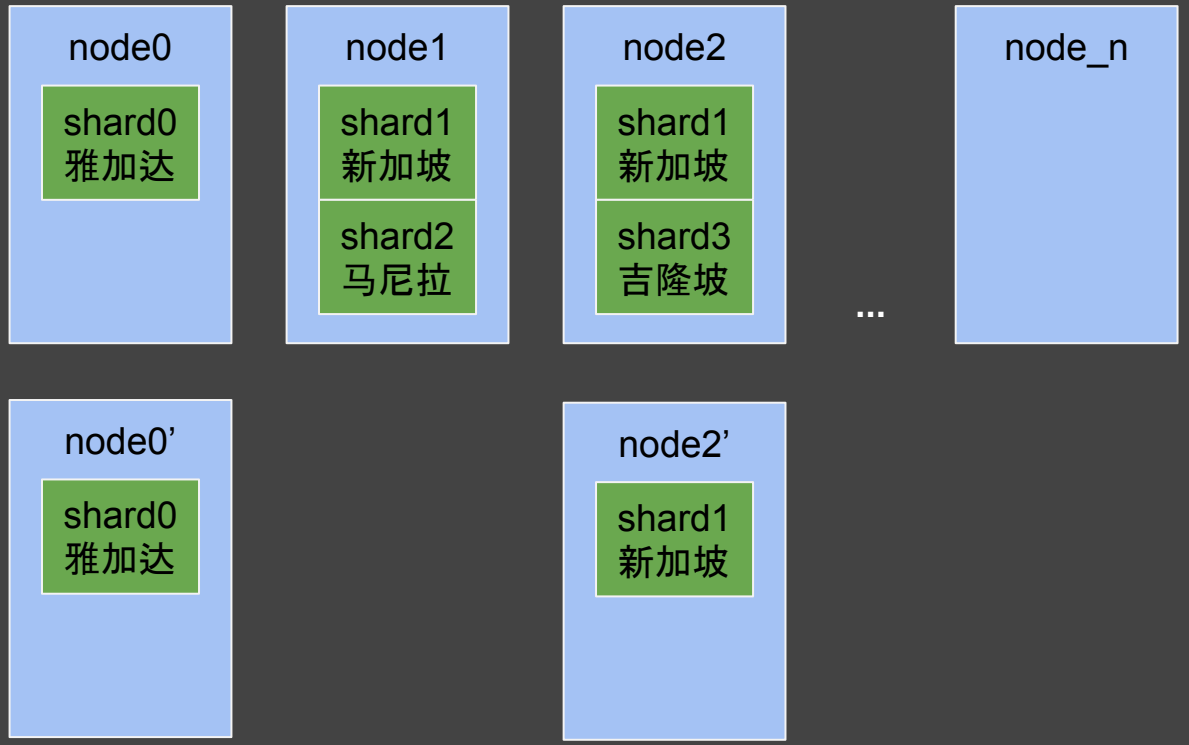
Sharding策略 - Shard分裂

- 独立节点写吞吐已到上限
  - 调整Resolution 分裂大Shard为小Shard



Sharding策略 - Shard分配策略

- 读不平衡
  - 加Replica



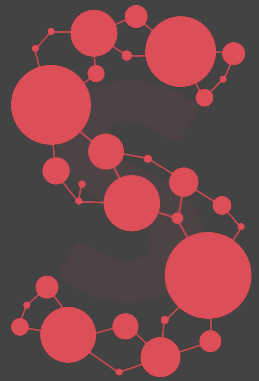
分布式怎么做?

分布式

- 一致性要求
  - 最终一致性
- 关键路径
  - 减少对第三方服务的依赖 包括服务发现如ETCD/ZK等
  - 基于Gossip的P2P对等网络 去中心

分布式

- 一致性
  - 最终一致性
- Gossip
  - Serf
  - Write in Golang



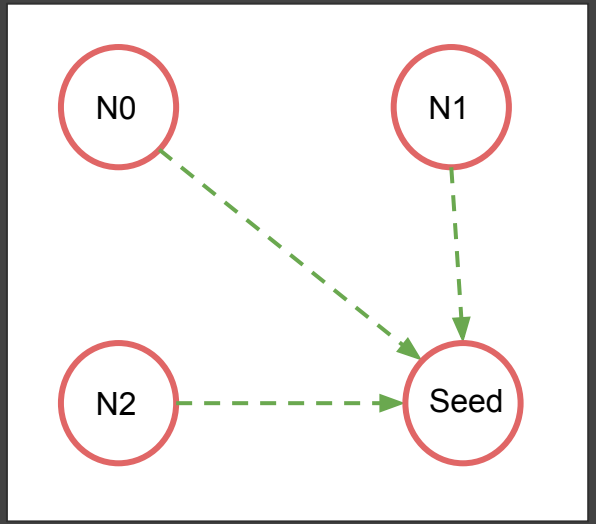
分布式

- 最终一致性
- Serf
  - 基于Gossip的Membership
    - Gossip: 八卦, 反熵, C*

Gossip Might Be Good

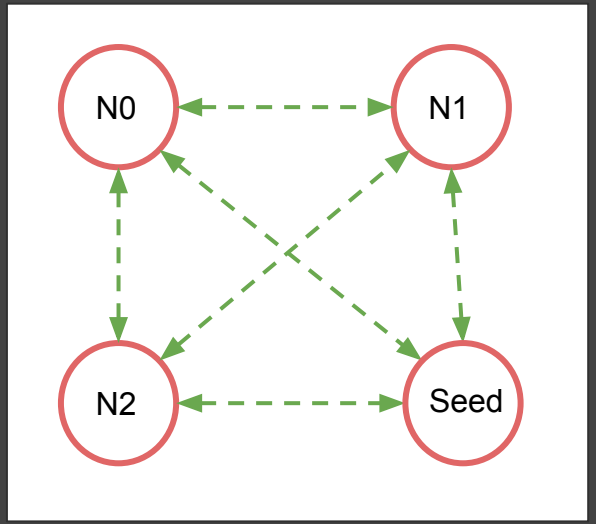
分布式

- 最终一致性
- Serf
  - 基于Gossip的Membership
    - Gossip: 八卦, 反熵, C*
    - Membership: 找到其它网络节点



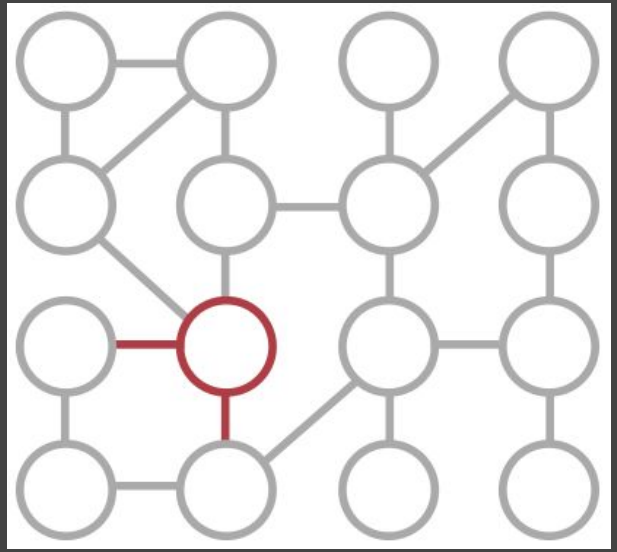
分布式

- 最终一致性
- Serf
  - 基于Gossip的Membership
    - Gossip: 八卦, 反熵, C*
    - Membership: 找到其它网络节点
    - 最终每个节点都可以获取到整个网络拓扑

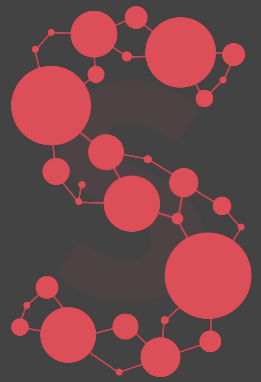


分布式

- 最终一致性
- Serf
  - 基于Gossip的Membership
    - Gossip: 八卦, 反熵, C*
    - Membership: 找到其它网络节点
    - 最终每个节点都可以获取到整个网络拓扑
  - 故障检测
    - 大集群时故障是常态
    - 更小的误报率
      - 减少false-positive(非故障被识别)
      - 减少false-negative(已故障未识别)
    - 快速摘除
  - 不只是服务发现
    - 提供可靠性保证



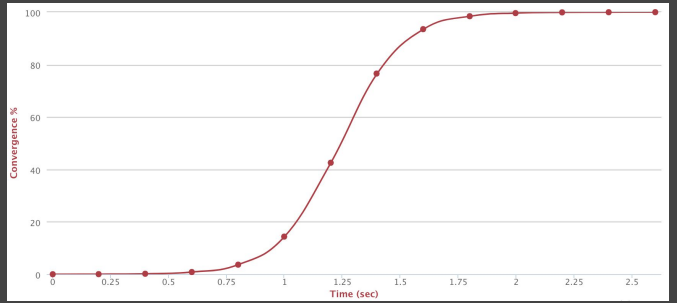
分布式



- 最终一致性
- Serf
  - 基于Gossip的Membership
  - 故障检测
- SWIM
  - Scalable Weakly-consistent Infection-style process group Membership protocol
    - 可扩展弱一致性感染型 进程组成员协议

SWIM

- Scalable
  - 随着结点的增多 故障检测耗时不会大幅增加
    - 如heart-beating机制 消息通信次数呈指数 级
- Weakly
  - 在某一时刻 不同的结点看到不同的系 统状态
  - 最终会收敛到相同的状 态
- Infection-style
  - Gossip
  - 每个结点持有部分结点信息 每个结点与其子结点交换信息
  - 最终所有结点通过`八卦`别人的信息获得全局信息
- Membership
  - 找到其它网 络结点



Nodes: 10K

Gossip Protocol in serf

- SWIM
  - 故障检测
  - 传播

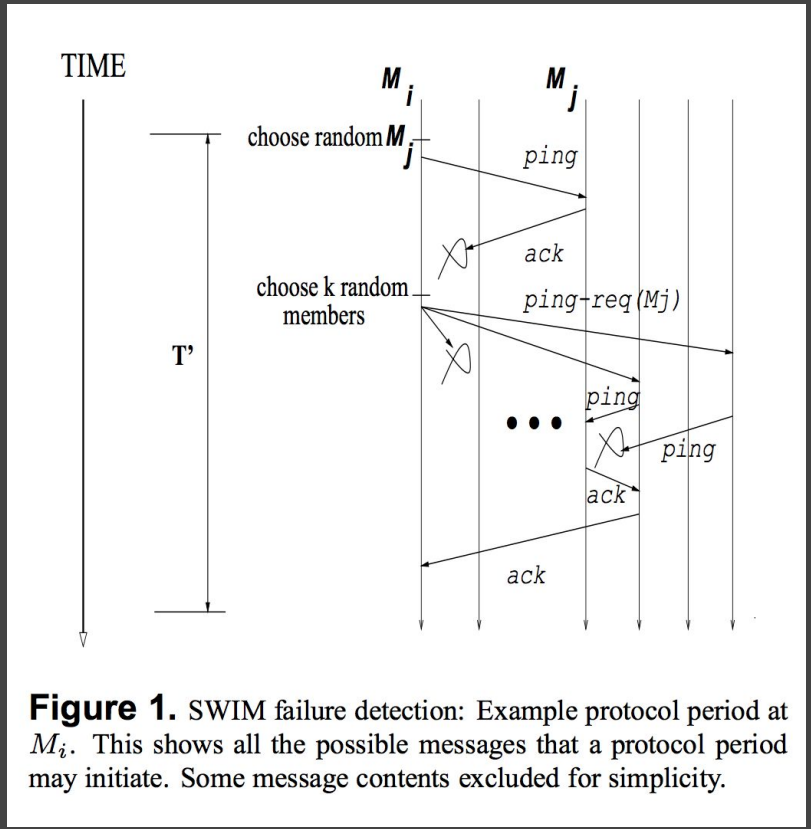
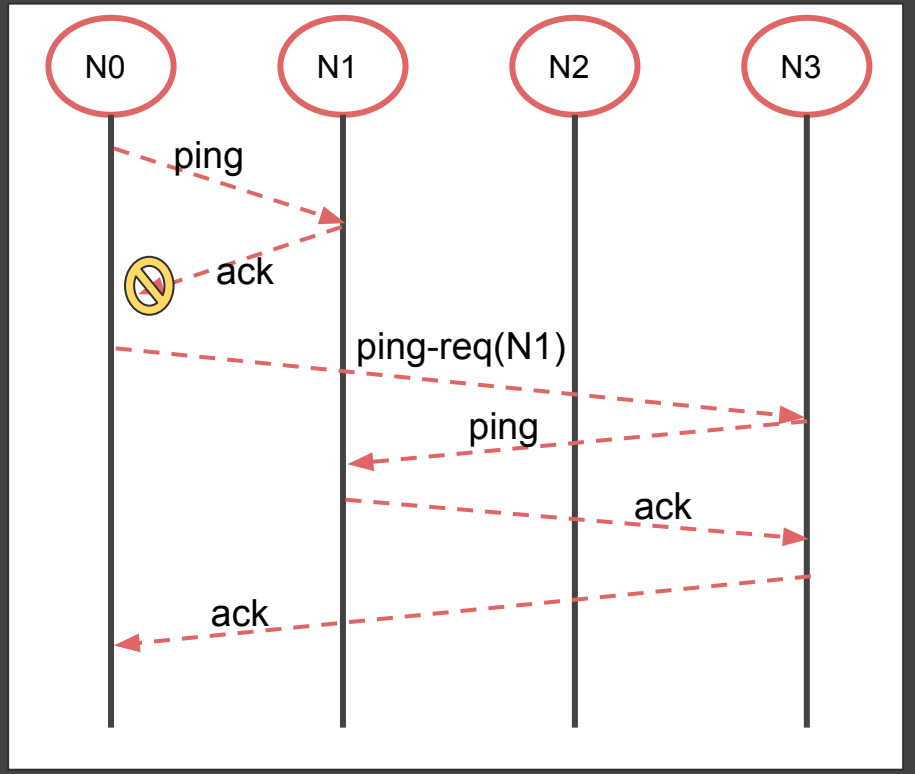


Figure 1. SWIM failure detection: Example protocol period at M_i. This shows all the possible messages that a protocol period may initiate. Some message contents excluded for simplicity.

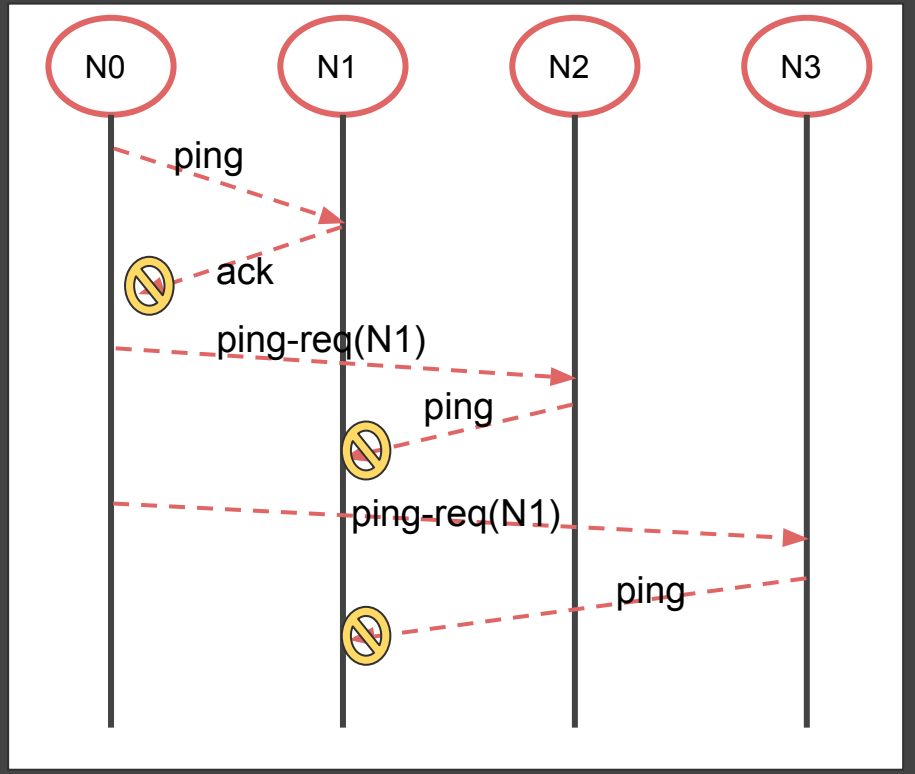
Gossip Protocol in serf

- SWIM
  - 故障检测
    - 间接探测 - 至少一个成功



Gossip Protocol in serf

- SWIM
  - 故障检测
    - 间接探测 - 全部失败
    - 标记为dead



Gossip Protocol in serf

- SWIM
  - 故障检测
  - 传播

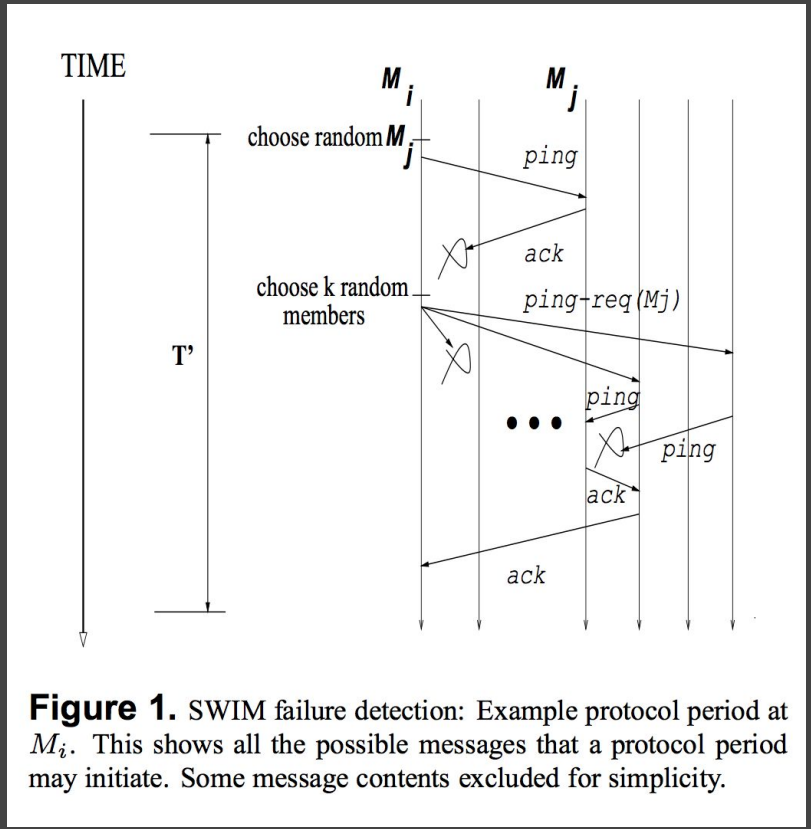


Figure 1. SWIM failure detection: Example protocol period at M_i. This shows all the possible messages that a protocol period may initiate. Some message contents excluded for simplicity.

Gossip Protocol in serf

SWIM: *S*calable *W*eakly-consistent *I*nfection-style Process Group *M*embership Protocol

Abhinandan Das, Indranil Gupta, Ashish Motivala*
Dept. of Computer Science, Cornell University
Ithaca NY 14853 USA
{asdas,gupta,ashish}@cs.cornell.edu

Abstract

Several distributed peer-to-peer applications require weakly-consistent knowledge of process group membership information at all participating processes. SWIM is a generic software module that offers this service for large-scale process groups. The SWIM effort is motivated by the unscalability of traditional heart-beating protocols, which either impose network loads that grow quadratically with group size, or compromise response times or false positive frequency w.r.t. detecting process crashes. This paper reports on the design, implementation and performance of the SWIM sub-system on a large cluster of commodity PCs.

Unlike traditional heartbeating protocols, SWIM separates the failure detection and membership update dissemination functionalities of the membership protocol. Processes are monitored through an efficient peer-to-peer periodic randomized probing protocol. Both the expected time to first detection of each process failure, and the expected message load per member, do not vary with group size.

1. Introduction

As you swim lazily through the milieu,
The secrets of the world will infect you.

Several large-scale peer-to-peer distributed process groups running over the Internet rely on a distributed membership maintenance sub-system. Examples of existing middleware systems that utilize a membership protocol include reliable multicast [3, 11], and epidemic-style information dissemination [4, 8, 13]. These protocols in turn find use in applications such as distributed databases that need to reconcile recent disconnected updates [14], publish-subscribe systems, and large-scale peer-to-peer systems[15]. The performance of other emerging applications such as large-scale cooperative gaming, and other collaborative distributed applications, depends critically on the reliability and scalability of the membership maintenance protocol used within.

Briefly, a membership protocol provides each process ("member") of the group with a locally-maintained list of other non-faulty processes in the group. The protocol en-

Lifeguard: Local Health Awareness for More Accurate Failure Detection

Armon Dadgar James Phillips Jon Currey
{armon,james,jc}@hashicorp.com

***Abstract*—SWIM is a peer-to-peer group membership protocol with attractive scaling and robustness properties. However, slow message processing can cause SWIM to mark healthy members as failed (so called false positive failure detection), despite inclusion of a mechanism to avoid this.**

We identify the properties of SWIM that lead to the problem, and propose Lifeguard, a set of extensions to SWIM which consider that the local failure detector module may be at fault, via the concept of *local health*. We evaluate this approach in a precisely controlled environment and validate it in a real-world scenario, showing that it drastically reduces the rate of false positives. The false positive rate and detection time for true failures can be reduced simultaneously, compared to the baseline levels of SWIM.

I. Introduction

Three key issues that any distributed system must address are discovery, fault detection, and load balancing among its components. Group membership is an intuitive abstraction that can be used to address all three issues simultaneously. Members of a group and its clients are offered a dynamically updating view of the current group membership, and use this view to perform actions such as request routing and state migration.

SWIM [1] is a group membership protocol with a number of attractive properties. It's peer-to-peer design and use of randomized communication make it highly scalable, robust to both node and network failures, and easy to deploy and manage. Its simplicity make it easy to implement and debug, compared to many distributed systems protocols.

proposes a Suspicion subprotocol, that trades increased failure detection latency for fewer false positives.

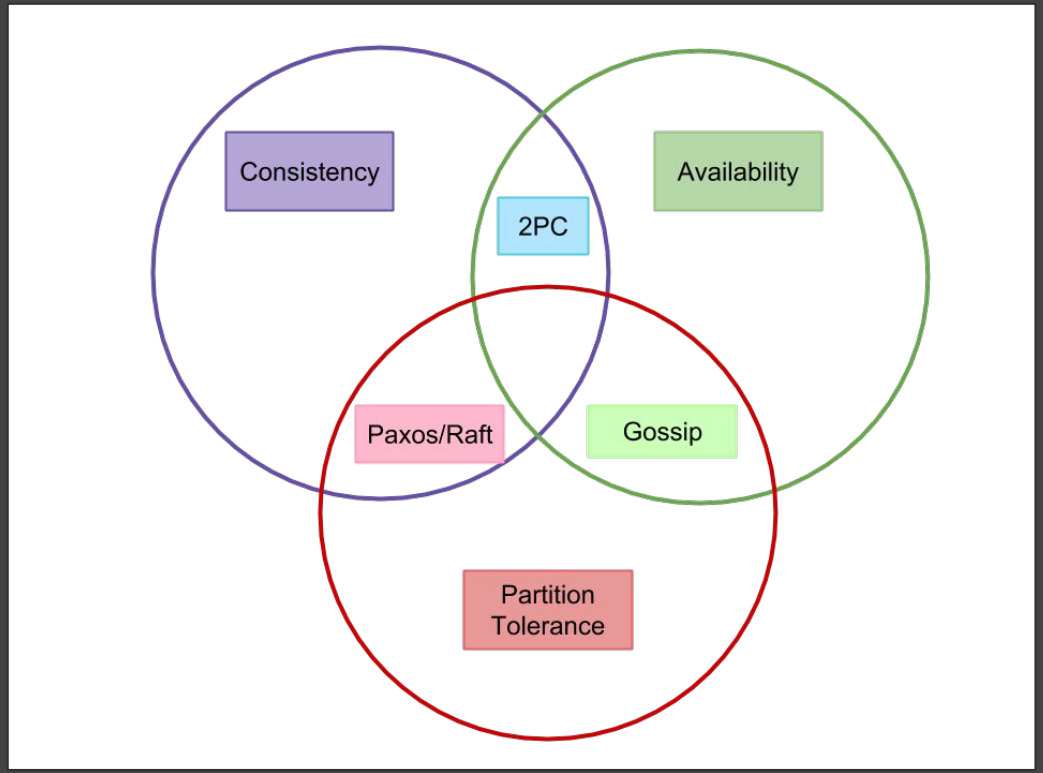
However, our experience supporting Consul and Nomad shows that, even with the Suspicion subprotocol, slow message processing can still lead healthy members being marked as failed in certain circumstances. When the slow processing occurs intermittently, a healthy member can oscillate repeatedly between being marked as failed and healthy. This 'flapping' can be very costly if it induces repeated failover operations, such as provisioning members or re-balancing data.

Debugging these scenarios led us to insights regarding both a deficiency in SWIM's handling of slow message processing, and a way to address that deficiency. The approach used is to make each instance of SWIM's failure detector consider its own health, which we refer to as *local health*. We implement this via a set of extensions to SWIM, which we call Lifeguard. Lifeguard is able to significatly reduce the false positive rate, in both controlled and real-world scenarios.

The rest of the paper is structured as follows: Section II motivates the advantages of SWIM that lead us to use it, and the kinds of scenarios where we have encountered this problem. Section III describes SWIM and memberlist, the implementation of SWIM that we use to evaluate Lifeguard. Section IV describes the Lifeguard extensions to SWIM. Section V describes the experimental evaluation of the components of Lifeguard, individually and in combination. Section VI describes Lifeguard's relationship to prior work. In Section VII we discuss the conclusions that can be drawn, and potential future work.

分布式

- CAP中的AP
- 去中心



如何保证可用性

可用性

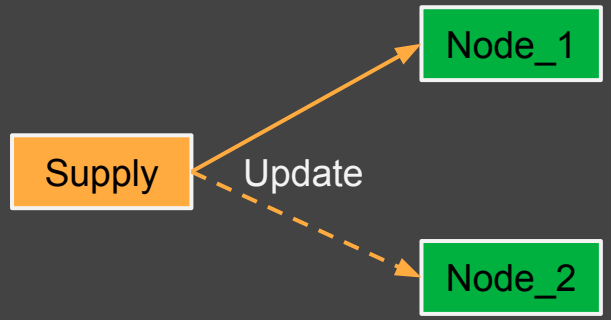
- 关键路径
  - No Down Time
  - 快速恢复
- 部分可用
  - 区域性不可用 不能影响全局
  - 容错
    - 如:不能因部署导致不可用
- 过载保护 快速失败

可用性

- No downtime
  - 不是No Down, 是要及时发现并处理
  - Serf提供的可靠性保证
    - 快速摘除坏掉的节点
  - Redundancy(冗余)
    - Replica set(副本集)

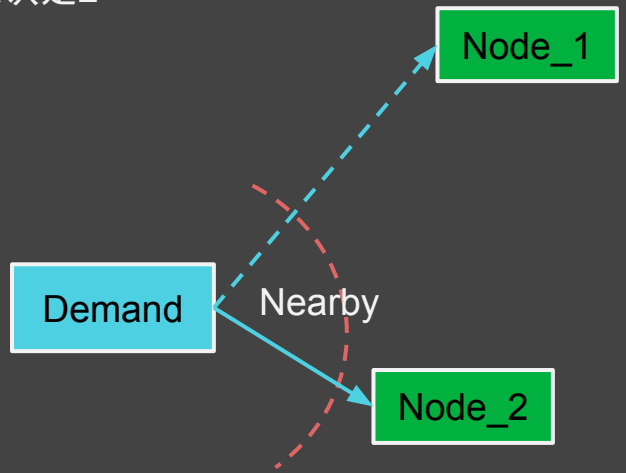
可用性

- Replica Set
  - 每个Shard以副本集的形式存在于多个 节点
  - 副本数越多 可用性越高 但需要与资源消耗做个trade off 默认是2
- 写(Update Location)
    - 多副本 并发写 可设置写一致性level



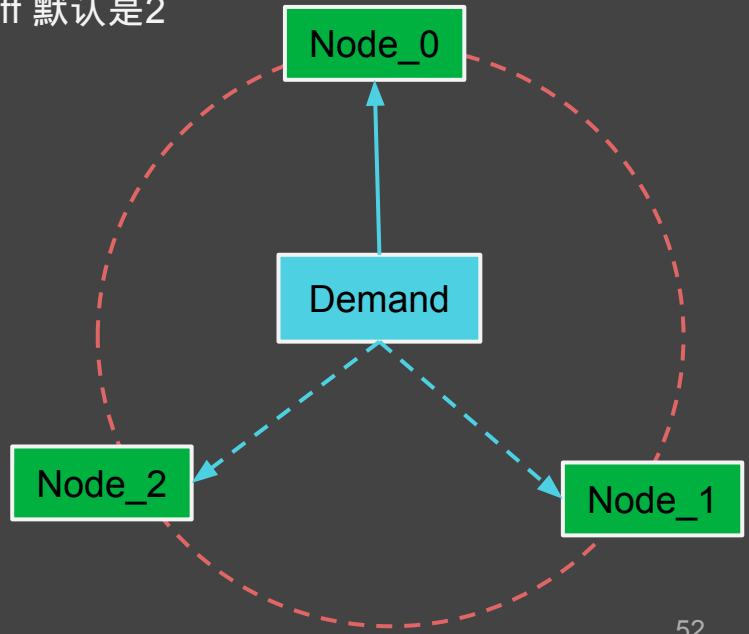
可用性

- Replica Set
  - 每个Shard以副本集的形式存在于多个 节点
  - 副本数越多 可用性越高 但需要与资源消耗做个trade off 默认是2
- 写(Update Location)
  - 多副本 并发写 可设置写一致性level
- 读(Nearby)
  - Fanout
    - 并发读多个节点, 优先使用Latency最低的



可用性

- Replica Set
  - 每个Shard以副本集的形式存在于多个 节点
  - 副本数越多 可用性越高 但需要与资源消耗做个trade off 默认是2
- 写(Update Location)
    - 多副本 并发写 可设置写一致性level
- 读(Nearby)
    - Fanout
      - 并发读多个节点, 优先使用Latency最低的
    - Round-robin
      - 按照当前Shard所在节点轮询调度



可用性

- 快速恢复
  - 定时存储快照
  - 启动时加载快照 重建空间索引

Down time happens

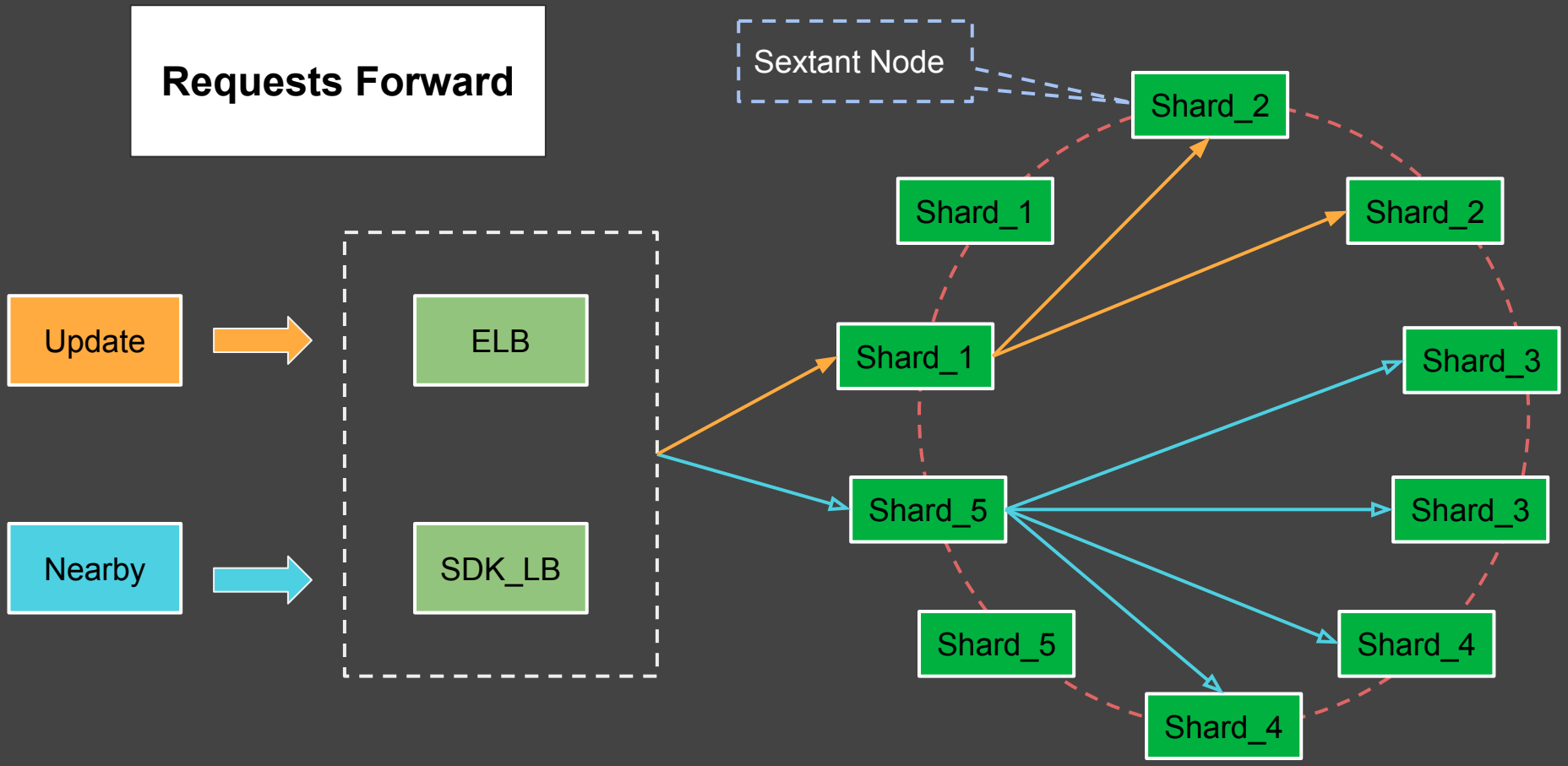
可用性

- 区域性不可用
  - 某节点坏掉
    - 及时摘除 告警
    - 基于AWS的自动扩容机制会触发增加新的节点
  - 某Shard高并发读写导致的节点负载高
    - 扩容
- 日常部署
  - Fanout模式

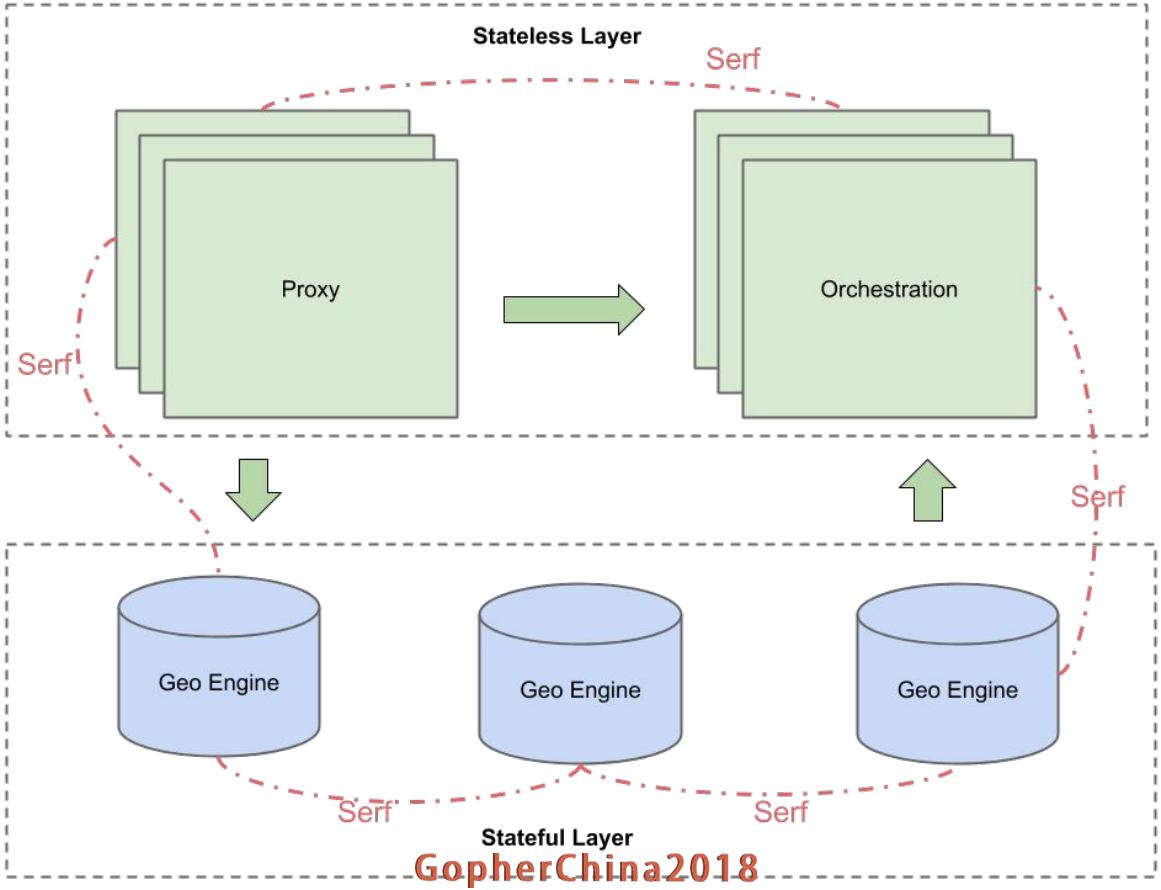
可用性

- 过载保护
  - Circuit Breaker(熔断)
  - Rate Limiter(限流)
- 快速失败
  - 基于Serf故障检测机制

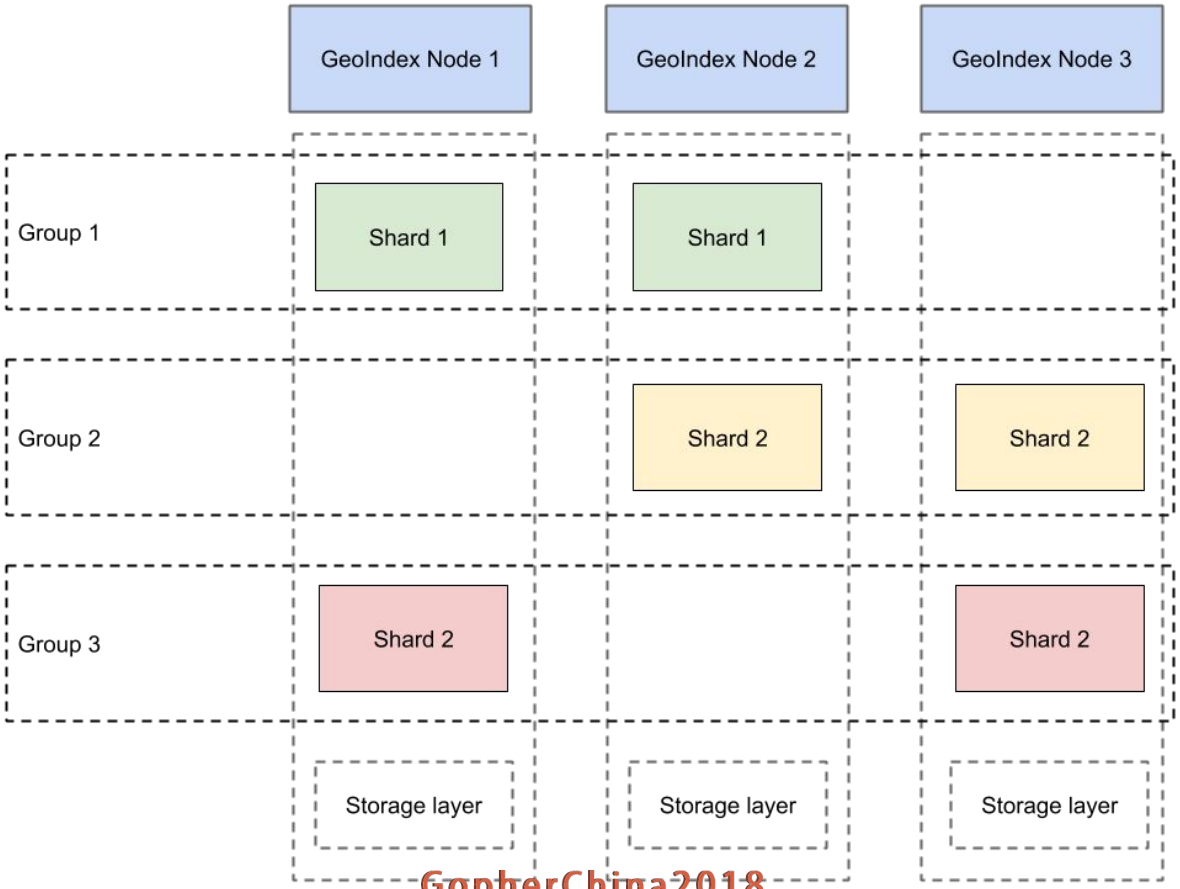
Requests Forward



Sextant Architecture



Sextant - Geo Engine



我们做到了什么？

- 去中心 无依赖
- 支持扩容及再平衡 再也不用担心数据热点了
  - 运营小伙伴可以放心大胆搞促 销
- No Downtime
  - 上线1年多以来没有 发生一次服务宕机事故
- 更好的性能
  - 写: ~2毫秒
  - 读: ~10毫秒
- 支撑业务高速增长

我们还可以做得更好

- Shard分裂
  - 目前还需要重启 不能平滑分裂
- 引擎
  - 当前空间索引不是最优的 比如最坏情况下 要遍历整个Shard空间
  - 如何支持多引擎 比如S2

永远在路上

大纲

- What's Grab
- 一个典型的派单流程
- Nearby这个核心地理服务系统演进历程
- Why go
- 压测与调优
- QA

Why Go?

业务场景

- 大量的几何运算
- 大流量网络吞吐
- CPU-Bound & I/O-Bound

业务场景

- 大量的几何运算
- 大流量网络吞吐
- CPU-Bound & I/O-Bound

Go

- 基于goroutine/channel 并行处理出色
- 基于epoll 对高并发支持良好
- 工程化能力及日益丰富的周边生态
- 简洁易上手

Go in Grab

- Golang Camp
- 统一Repo 大量最佳实践 基础库及框架
- Review before land
- 单元测试覆盖度
- 比较完善的CI/CD
- Automate everything

大纲

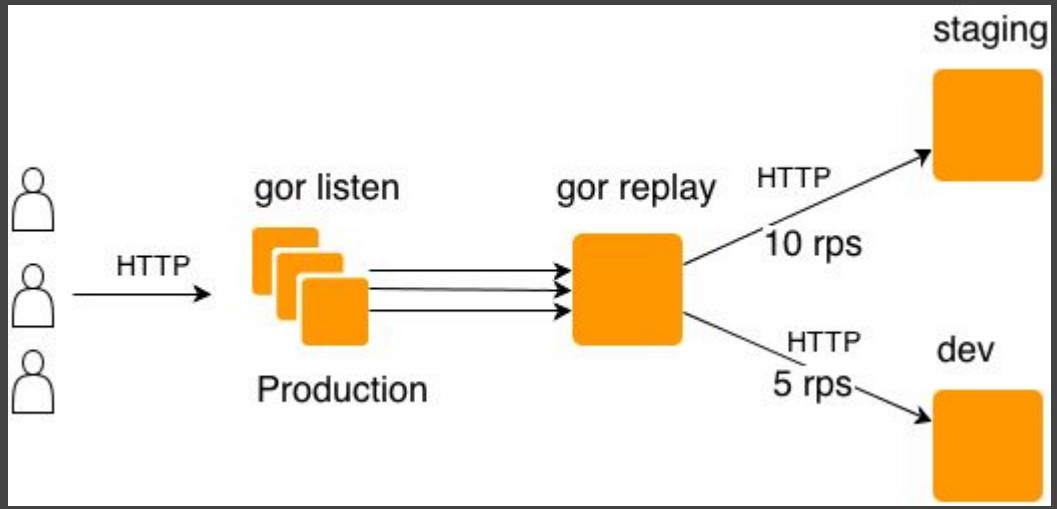
- What's Grab
- 一个典型的派单流程
- Nearby这个核心地理服务系统演进历程
- Why go
- 压测与调优
- QA

基于生产数据的测试

GoReplay

- 流量重放
- 格式易读
- 无侵入
- Golang

GO REPLAY



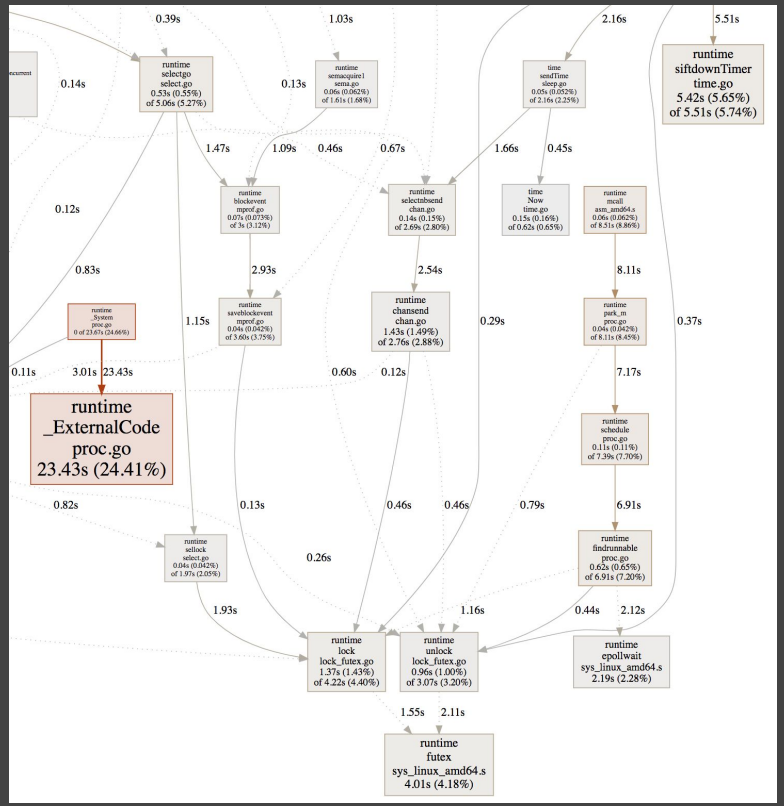
分布式并行压测

Vegeta + Fabric

- Vegeta
  - Golang
  - 不只是命令行 也可以作为Library
- Fabric
  - Pythonic remote execution and deployment

调优 - go tool pprof

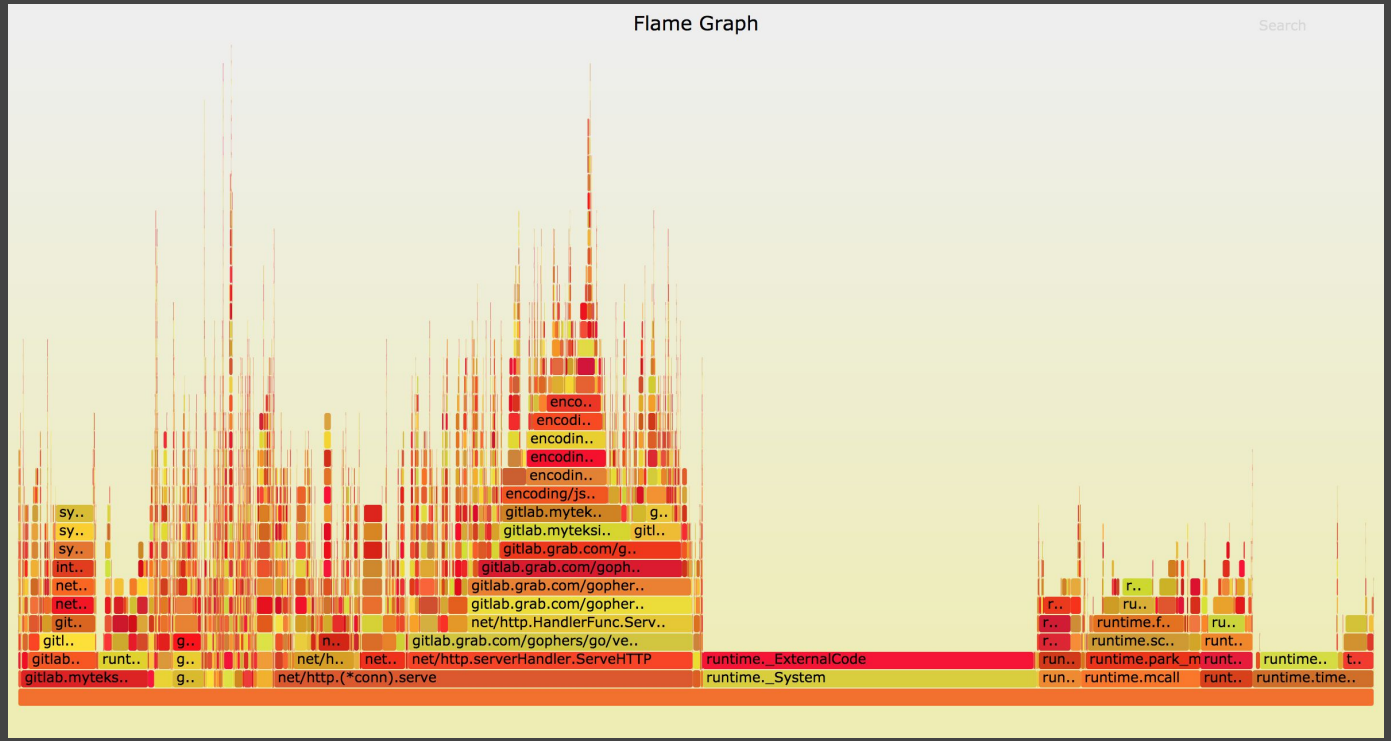
CPU Profiling



调优 - go-torch

颜色不是重点

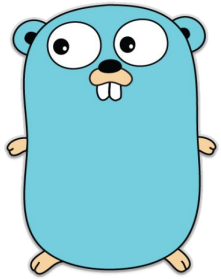
宽度才是



刚刚，我们收购了Uber东南亚业务

KEEP
CALM
AND
NO
PANIC

Tech Stack



GopherChina2018 Amazon EC2

We’re hiring

zhiyin.zhang@grab.com

Thanks