

聊天系统和访问限制系统 Message System & Rate Limiter

课程版本: v5.0 本节主讲人: 东邪



扫描二维码关注微信/微博
获取最新面试题及权威解答

微信: [ninechapter](#)

微博: <http://www.weibo.com/ninechapter>

知乎: <http://zhuoanlan.zhihu.com/jiuzhang>

官网: <http://www.jiuzhang.com>

- Design WhatsApp
 - Work Solution
 - Real-time Service
 - Online Status: Pull vs Push
- Design Rate Limiter
- Design Datadog
- 这节课之后您可以学会
 - 设计聊天系统的核心: Realtime Service
 - Pull 与 Push 的进一步比较分析
- 相关设计题
 - Design Facebook Messenger
 - Design WeChat

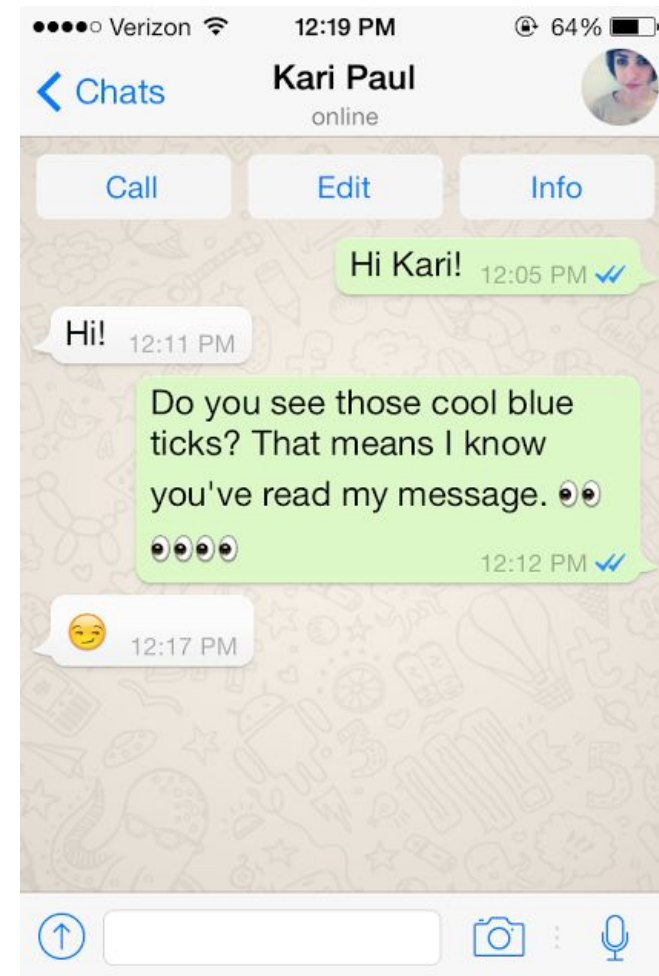


Interviewer: Design WhatsApp

设计“在干嘛”聊天APP



- 基本功能：
 - * 用户登录注册
 - * 通讯录
 - 两个用户互相发消息
 - 群聊
 - 用户在线状态
- 其他功能：
 - * 历史消息
 - * 多机登陆 Multi Devices

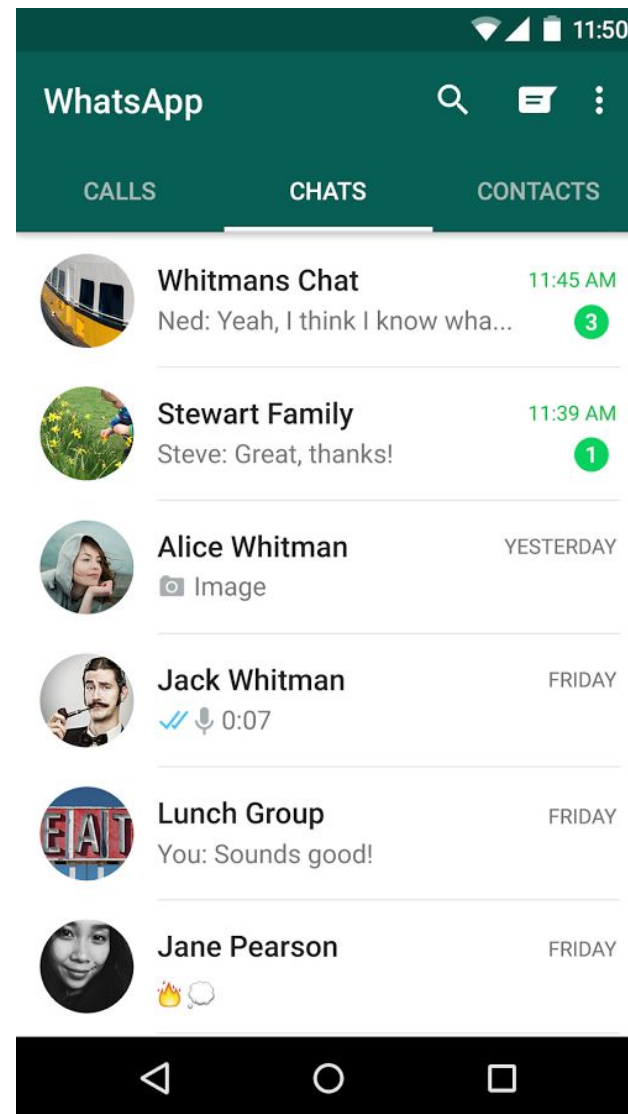
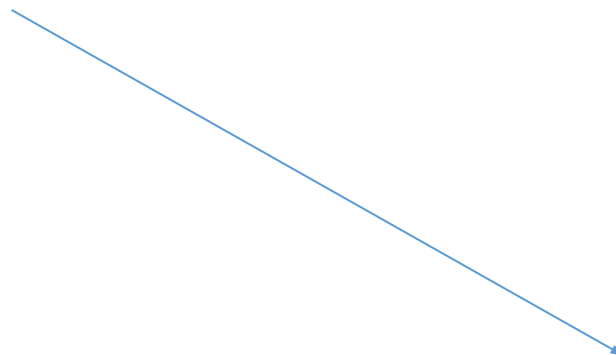


Scenario - 设计多牛的系统？

- WhatsApp
 - 1B 月活跃用户
 - 75% 日活跃 / 月活跃
 - 约750M日活跃用户
 - ——数据来自 Facebook 官方, 截止2016年3月
- 为了计算方便起见, 我们来设计一个100M日活跃的WhatsApp
- QPS:
 - 假设平均一个用户每天发20条信息
 - $\text{Average QPS} = 100\text{M} * 20 / 86400 \sim 20\text{k}$
 - $\text{Peak QPS} = 20\text{k} * 5 = 100\text{k}$
- 存储:
 - 假设平均一个用户每天发10条信息
 - 一天需要发 1B, 每条记录约30bytes的话, 大概需要30G的存储

- Message Service
 - 负责信息管理
- * Real-time Service
 - 负责实时推送信息给接受者

- Message Table (NoSQL)
 - 数据量很大, 不需要修改, 一条聊天信息就像一条log一样
- Thread Table (SQL) —— 对话表
 - 需要同时 index by
 - Owner User Id
 - Thread Id
 - Participants hash
 - Updated time
 - NoSQL 对multi indexes 的支持并不是很好



| Message Table | | |
|---------------|-----------|-------------------|
| message_id | int64 | user_id+timestamp |
| thread_id | int64 | |
| user_id | int64 | |
| content | text | |
| created_at | timestamp | |

| Thread Table | | |
|------------------|-----------|--------------------------|
| user_id | int64 | |
| thread_id | int64 | create_user_id+timestamp |
| participant_ids | text | json |
| participant_hash | string | avoid duplicate threads |
| created_at | timestamp | |
| updated_at | timestamp | index=true |

- 用户如何发送消息？
 - Client 把消息和接受者信息发送给 server
 - Server为每个接受者(包括发送者自己)创建一条 Thread (如果没有的话)
 - 创建一条message(with thread_id)
- 用户如何接受消息？
 - 可以每隔10秒钟问服务器要一下最新的 inbox
 - 虽然听起来很笨, 但是也是我们先得到这样一个可行解再说
 - 如果有新消息就提示用户

- 如何设计 Message System 和构建一个可行的 WhatsApp

Scale 拓展

性能上的拓展(支持更多的用户, 有更快的响应速度)
功能上的拓展(如支持群聊, 在线状态)

Interviewer: How to Scale?

Message 是 NoSQL, 自带 Scale 属性

Thread 按照 user_id 进行 sharding

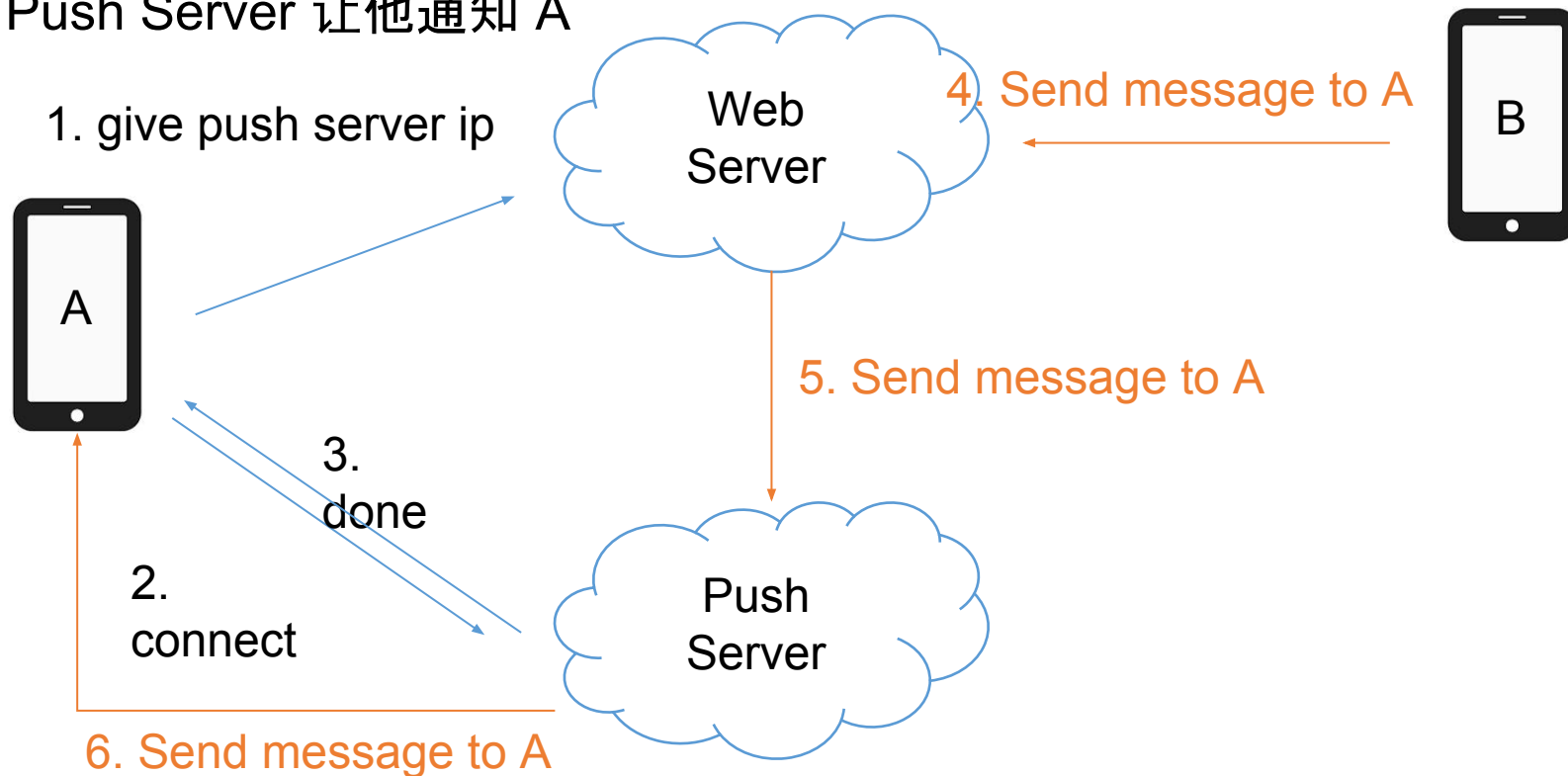
Interviewer: How to speed up?

每隔10秒钟收一次消息太慢了，聊天体验很差，不实时

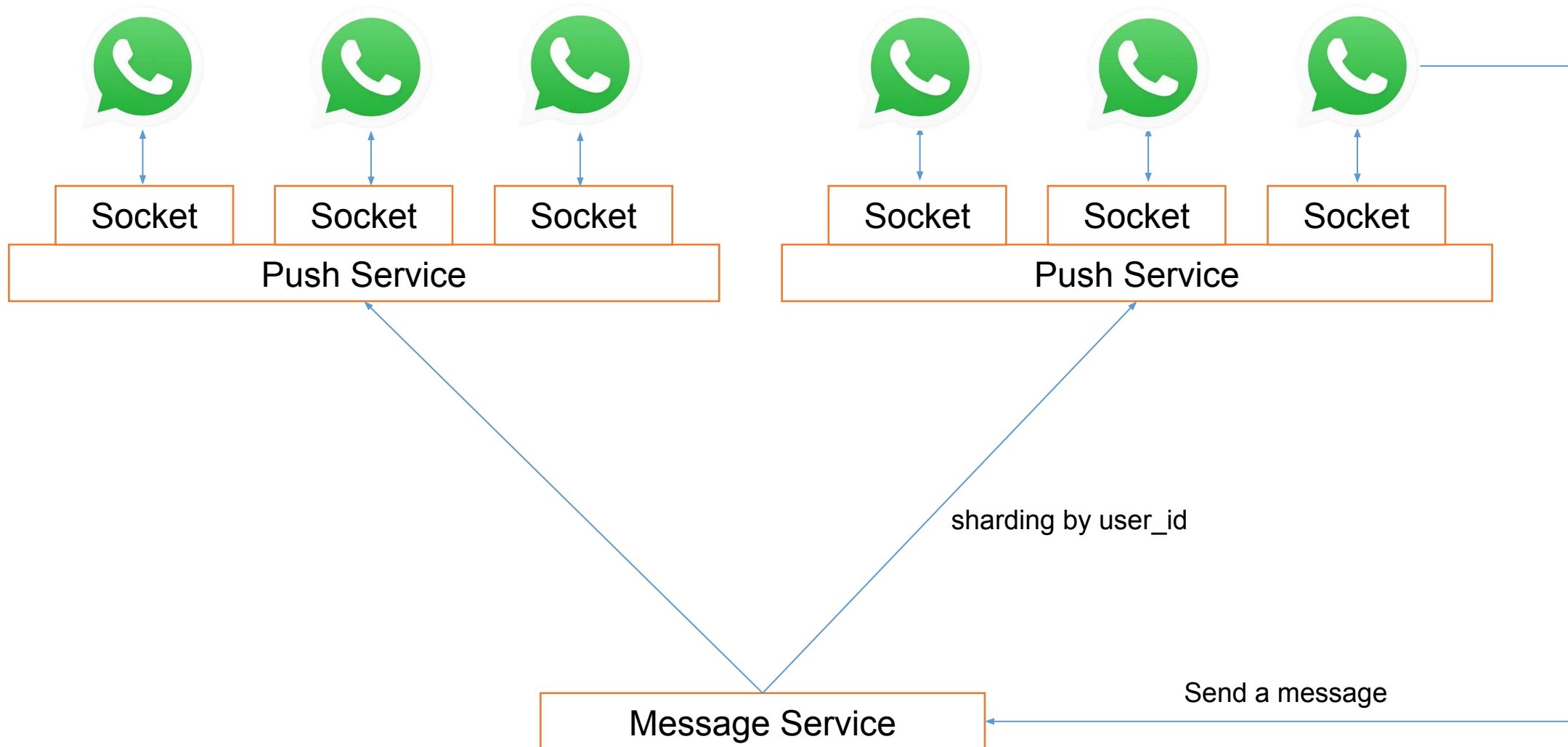
Scale 拓展 —— Socket

- 需要引入一个新的概念——Socket
- 再引入一个新的Service —— Push Service
- Push Service 提供 Socket 连接服务, 可以与Client保持TCP的长连接
- 当用户打开APP之后, 就连接上Push Service 中一个属于自己的socket。
- 有人发消息的时候, Message Service 收到消息, 通过Push Service把消息发出去
- 如果一个 用户长期不活跃(比如10分钟), 可以断开链接, 释放掉网络端口
- 断开连接之后, 如何收到新消息?
 - 打开APP时主动Pull + Android GCM / IOS APNS
- Socket 链接 与 HTTP 链接的最主要区别是
 - HTTP链接下, 只能客户端问服务器要数据
 - Socket链接下, 服务器可以主动推送数据给客户端

- 用户A打开App后, 问 Web Server 要一个 Push Service 的连接地址
- A通过 socket 与push server保持连接
- 用户B发消息给A, 消息先被发送到服务器
- 服务器把消息存储之后, 告诉 Push Server 让他通知 A
- A 收到及时的消息提醒



Scale 拓展 —— Real-time Push Service



- 如何设计 Message System 和构建一个可行的 WhatsApp
- 引入 Push Service 解决实时性问题

Interviewer: How to support large group chat?

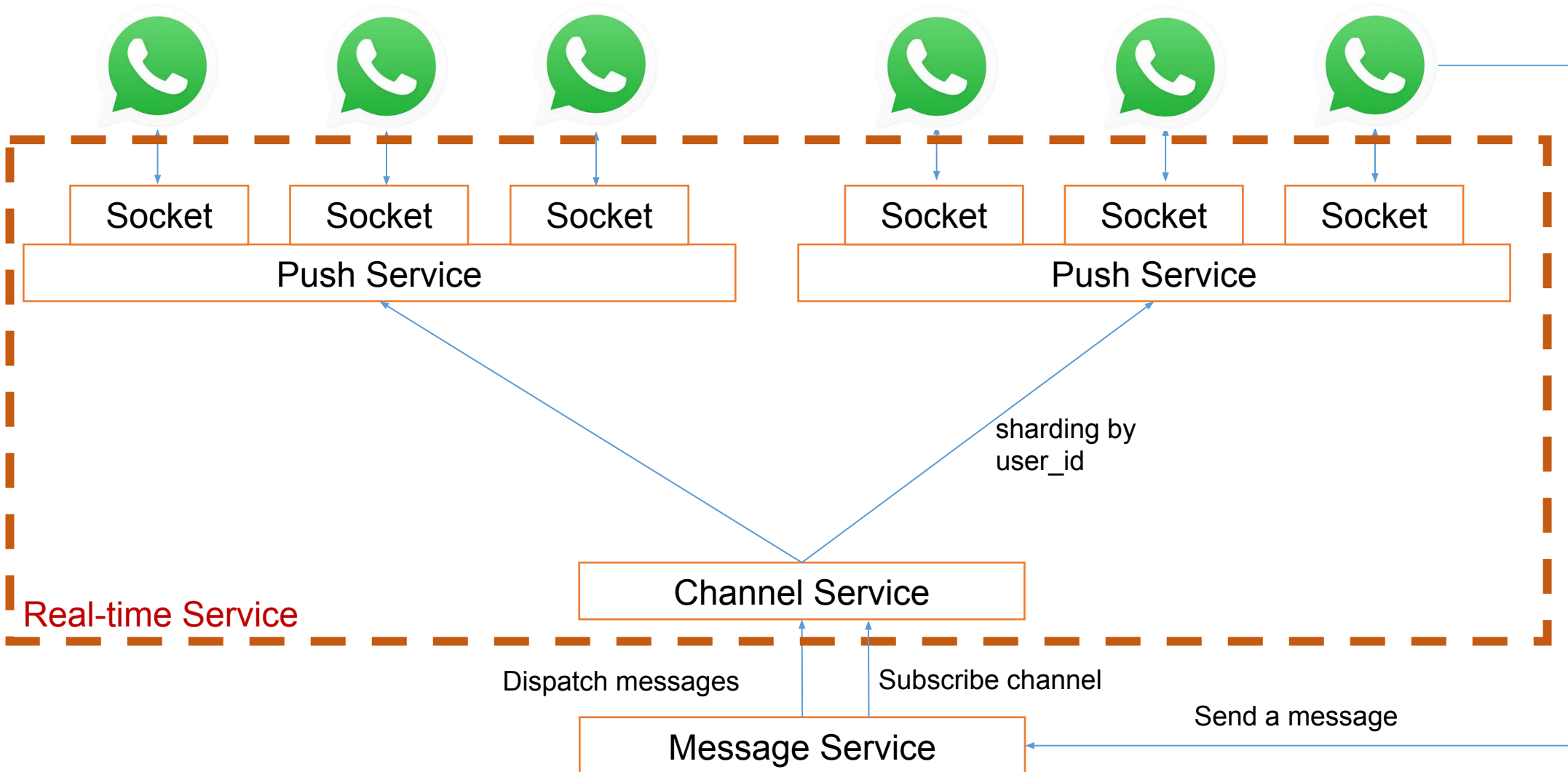
支持群聊



- 问题

- 假如一个群有500人(1m用户也同样道理)
- 如果不做任何优化, 需要给这 500 人一个个发消息
- 但实际上 500 人里只有很少的一些人在线(比如10人)
- 但Message Service仍然会尝试给他们发消息
 - Message Service (web server) 无法知道用户和Push Server的socket连接是否已经断开
 - 至于 Push Server 自己才知道
- 消息到了Push Server 才发现490个人根本没连上
- Message Service 与 Push Server 之间白浪费490次消息传递

- 解决
 - 增加一个Channel Service(频道服务)
 - 为每个聊天的Thread增加一个Channel信息
 - 对于较大群, 在线用户先需要订阅到对应的 Channel 上
 - 用户上线时, Web Server (message service) 找到用户所属的频道(群), 并通知 Channel Service 完成订阅
 - Channel就知道哪些频道里有哪些用户还活着
 - 用户如果断线了, Push Service 会知道用户掉线了, 通知 Channel Service 从所属的频道里移除
 - Message Service 收到用户发的信息之后
 - 找到对应的channel
 - 把发消息的请求发送给 Channel Service
 - 原来发500条消息变成发1条消息
 - Channel Service 找到当前在线的用户
 - 然后发给 Push Service 把消息 Push 出去



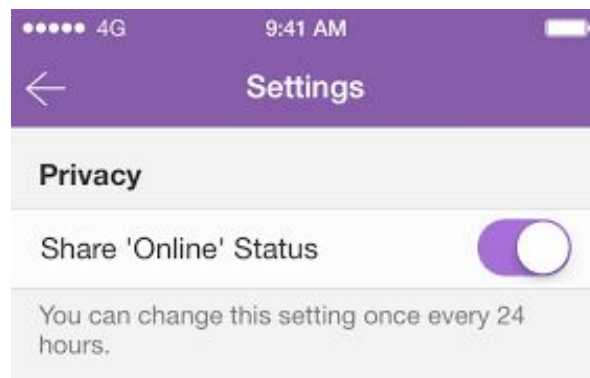
Channel Service 用什么存储数据？

内存就好了，数据重要程度不高，挂了就重启

因为还可以通过 IOS / Android 的 Push Notification 补救

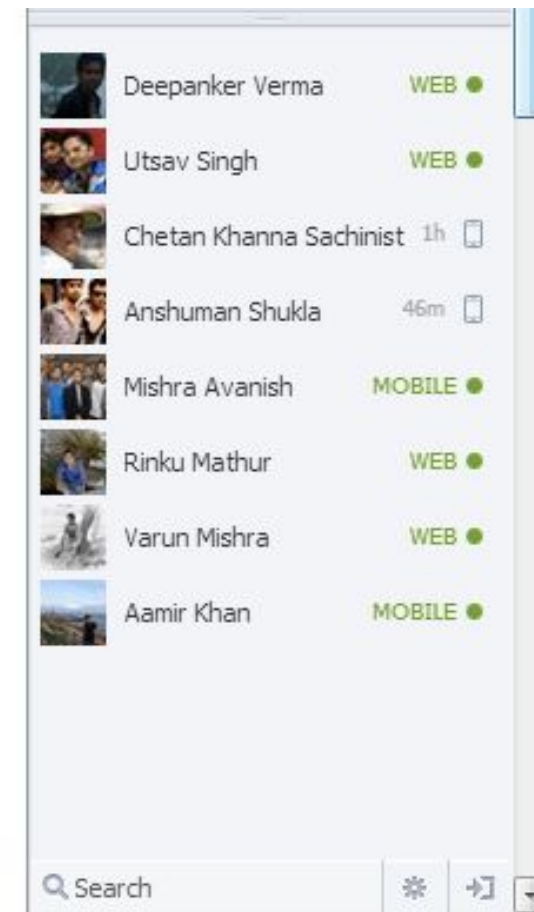
- 如何设计 Message System 和构建一个可行的 WhatsApp
- 引入 Push Service 解决实时性问题
- 引入 Channel service 解决群聊问题

Interviewer: How to check / update online status?

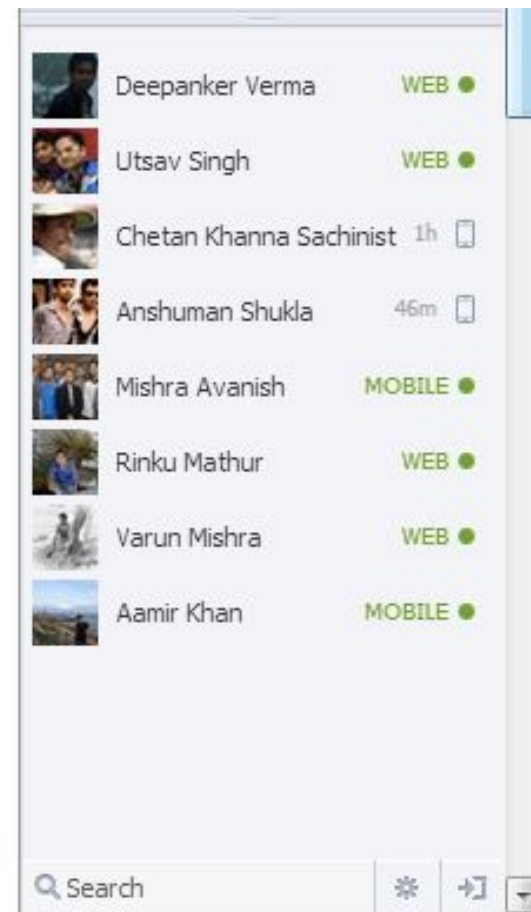


怎样实现在线状态？




- Update online status 包含两个部分
 - 服务器需要知道谁在线谁不在线(push or pull?)
 - 用户需要知道我的哪些好友在线(push or pull?)

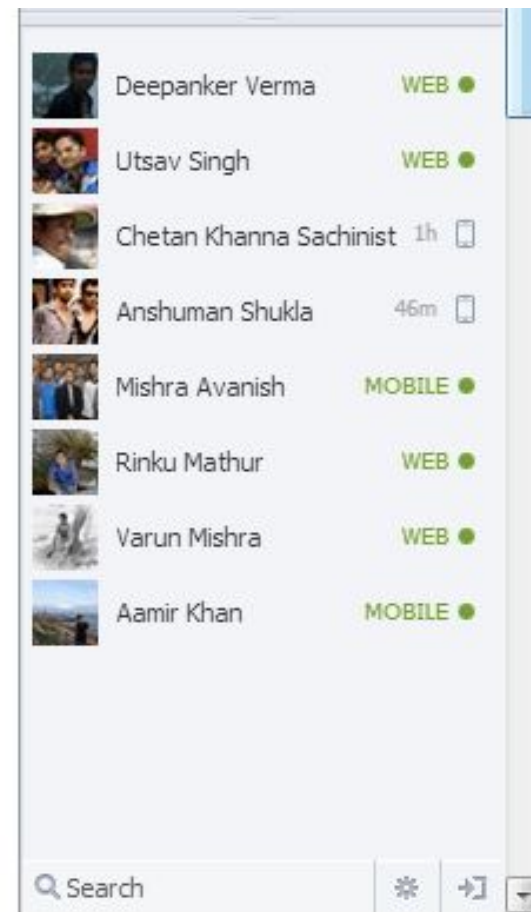


- 告诉服务器我来了 / 我走了
 - 用户上线之后, 与 Push Service 保持 socket 连接
 - 用户下线的时候, 告诉服务器断开连接
 - 问题: 服务器怎么知道你什么时候下线的? 万一网络断了呢?
- 服务器告诉好友我来了 / 我走了
 - 用户上线/下线之后, 服务器得到通知
 - 服务器找到我的好友, 告诉他们我来了 / 我走了
 - 问题1: 同上, 你怎么知道谁下线了
 - 问题2: 一旦某一片区的网络出现具体故障
 - 恢复的时候, 一群人集体上线, 比如有N个人
 - 那么要通知这N个人的 $N * 100$ 个好友, 造成网络堵塞
 - 问题3: 大部分好友不在线



- 告诉服务器我来了 / 我走了
 - 用户上线之后, 每隔3-5秒向服务器heart beat一次
- 服务器告诉好友我来了 / 我走了
 - 在线的好友, 每隔3-5秒钟问服务器要一次大家的在线状态
- 综合上述
 - 每隔10秒告诉服务器我还在, 并要一下自己好友的在线状态
 - 服务器超过1分钟没有收到信息, 就认为已经下线
- 可以打开你的 Facebook Messenger 验证一下
 - 打开 console 点击 network
 - 大概每隔3-5秒会pull一次服务器

| | | |
|--|--|-----|
|  | pull?channel=p_1312800249&seq... 3-edge-chat.facebook.com | 200 |
|  | pull?channel=p_1312800249&seq... 3-edge-chat.facebook.com | 200 |
|  | pull?channel=p_1312800249&seq... 3-edge-chat.facebook.com | 200 |



- 分析出有哪些服务和哪些数据表
- 为每个数据表选择合适的数据存储
- 细化表单结构
- 得到一个 Work Solution
- 知道按照什么 sharding
- 引入 Socket, Push Service, 加速聊天体验
- 引入Channel Service 解决large group chat问题
- 解决online status update 问题

WEAK

HIRED

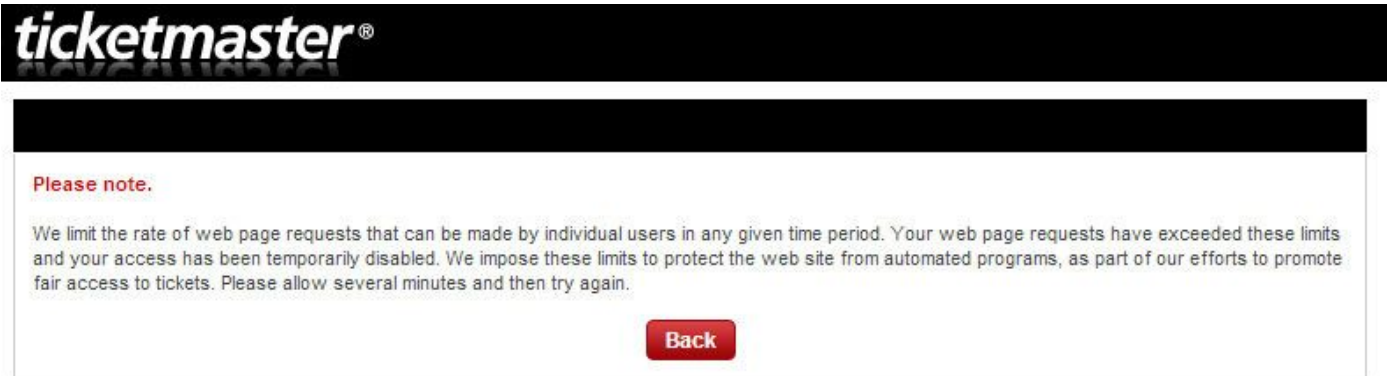


HIRED

Interviewer: How to limit requests?

如何限制访问次数？

比如 1 小时之类不能重置 >5 次密码

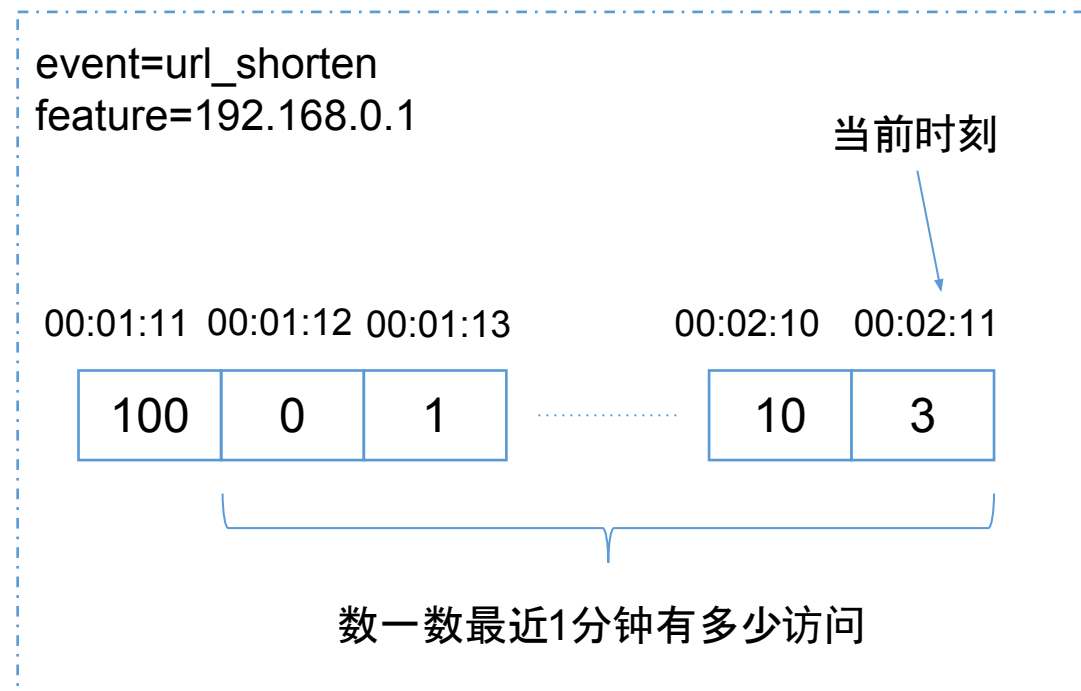


RateLimiter 访问限制器

- Rate limiter
 - 网站必用工具
 - 比如一分钟来自同一个邮箱的密码输入错误不能超过5次, 一天不超过10次
- 一些Open source的资源
 - Ruby: <https://github.com/ejfinneran/ratelimit>
 - Django: <https://github.com/jsocol/django-ratelimit>
 - 建议: 有空读一读源码
- Rate Limiter 已经是一个小型的系统设计问题
- 我们同样可以用 4S 分析法进行分析！

- Scenario 场景
 - 根据网络请求的特征进行限制(feature的选取)
 - IP (未登录时的行为), User(登录后的行为), Email(注册, 登录, 激活)
 - 系统需要做到怎样的程度
 - 如果在某个时间段内超过了一定数目, 就拒绝该请求, 返回 4xx 错误
 - 2/s, 5/m, 10/h, 100/d
 - 无需做到最近30秒, 最近21分钟这样的限制。粒度太细意义不大
- Service服务
 - 本身就是一个最小的 Application 了, 无法再细分
- Storage 数据存取
 - 需要记录(log)某个特征(feature)在哪个时刻(time)做了什么事情(event)
 - 该数据信息最多保留一天(对于 rate=5/m 的限制, 那么一次log在一分钟以后已经没有存在的意义了)
 - 必须是可以高效存取的结构(本来就是为了限制对数据库的读写太多, 所以自己的效率必须高与数据库)
 - 所以使用 Memcached 作为存储结构(数据无需持久化)

- 算法描述
- 用 event+feature+timestamp 作为 memcached 的key
- 记录一次访问:
 - 代码: `memcached.increament(key, ttl=60s)`
 - 解释: 将对应bucket的访问次数+1, 设置60秒后失效
- 查询是否超过限制
 - 代码:
 - for t in 0~59 do
 - `key = event+feature+(current_timestamp - t)`
 - `sum += memcached.get(key, default=0)`
 - Check sum is in limitation
 - 解释: 把最近1分钟的访问记录加和



- 问:对于一天来说,有86400秒,检查一次就要 86k 的 cache 访问,如何优化?
- 答:分级存储。
 - 之前限制以1分钟为单位的时候,每个bucket的大小是1秒,一次查询最多60次读
 - 现在限制以1天为单位的时候,每个bucket以小时为单位存储,一次查询最多24次读
 - 同理如果以小时为单位,那么每个bucket设置为1分钟,一次查询最多60次读
- 问:上述的方法中存在误差,如何解决误差?
 - 首先这个误差其实不用解决,访问限制器不需要做到绝对精确。
 - 其次如果真要解决的话,可以将每次log的信息分别存入3级的bucket(秒,分钟,小时)
 - 在获得最近1天的访问次数时,比如当前时刻是23:30:33, 加总如下的几项:
 - 在秒的bucket里加和 23:30:00 ~ 23:30:33(计34次查询)
 - 在分的bucket里加和 23:00 ~ 23:29(计30次查询)
 - 在时的bucket里加和 00 ~ 22(计23次查询)
 - 在秒的bucket里加和昨天 23:30:34 ~ 23:30:59 (计26次查询)
 - 在分的bucket里加和昨天 23:31 ~ 23:59(计29次查询)
 - 总计耗费 $34 + 30 + 23 + 26 + 29 = 142$ 次cache查询, 可以接受



你觉得是否需要解决误差?

Interviewer: Design a Datadog

Google url shortener

<http://goo.gl/rN7W>

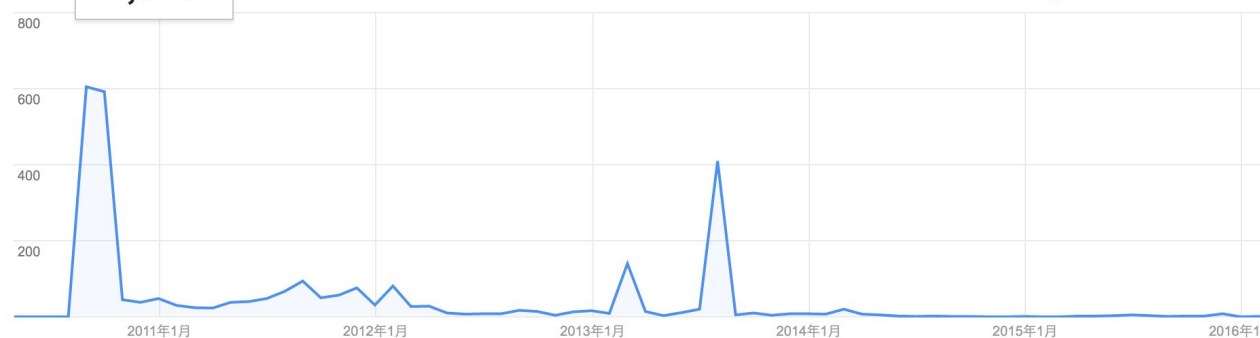
<http://www.sina.com.cn/>

Created: 2009 Dec 16



Total Clicks
2,859

Clicks for the past: [two hours](#) | [day](#) | [week](#) | [month](#) | [all time](#)



怎样统计访问数据？

- <https://www.datadoghq.com/>
- 同样进行 4S 分析！
- Scenario 设计些啥
 - 对于用户对于某个链接的每次访问，记录为一次访问
 - 可以查询某个链接的被访问次数
 - 知道总共多少次访问
 - 知道最近的x小时/x天/x月/x年的访问曲线图
 - 假设 Tiny URL 的读请求约 2k 的QPS
- Service
 - 自身为一个独立的Application，无法更细分

- Storage

- 基本全是写操作, 读操作很少
- 需要持久化存储(没memcached什么事儿了)
- SQL or NoSQL or File System?
 - 其实都可以, 业界的一些系统比如 Graphite 用的是文件系统存储
 - 这里我们假设用NoSQL存储吧
- 用NoSQL的话, key 就是 tiny url 的 short_key, value是这个key的所有访问记录的统计数据
 - 你可能会奇怪为什么value可以存下一个key的所有访问数据(比如1整年)
- 我们来看看value的结构
- 核心点是:
 - 今天的数据, 我们以分钟为单位存储
 - 昨天的数据, 可能就以5分钟为单位存储
 - 上个月的数据, 可能就以1小时为单位存储
 - 去年的数据, 就以周为单位存储
 - ...
 - **用户的查询操作通常是查询某个时刻到当前时刻的曲线图**
 - **也就意味着, 对于去年的数据, 你没有必要一分钟一分钟的进行保存**
- 多级Bucket的思路, 是不是和Rate Limiter如出一辙!

```
...
2016/02/26 23 1h 200
...
2016/03/27 23:50 5m 40
2016/03/27 23:55 5m 30
2016/03/28 00:00 1m 10
2016/03/28 00:01 1m 21
...
2016/03/28 16:00 m 2
```

怎样统计访问数据？

- 问: 2k的QPS这么大, 往NoSQL的写入操作也这么多么?
- 答: 不是。
 - 可以先将最近15秒钟的访问次数 Aggregate 到一起, 写在内存里
 - 每隔15秒将记录写给NoSQL一次, 这样写QPS就降到了100多
- 问: 如何将昨天的数据按照5分钟的bucket进行整理?
- 答: 对老数据进行瘦身
 - 当读发现一个key的value比较多的时候, 就触发一次“瘦身”操作
 - 瘦身操作把所有老的记录进行 Aggregate
 - 这些旧数据的记录的专业名词叫做: Retention

附录: 扩展阅读

- **Dynamo DB** —— 理解分布式数据库(NoSQL)的原理
 - <http://bit.ly/1mDs0Yh> [Hard] [Paper]
- **Scaling Memcache at Facebook** —— 妈妈再也不担心我的 Memcache
 - <http://bit.ly/1UlpbGE> [Hard] [Paper]
- Consistent Hashing
 - <http://bit.ly/1KhqPEr> [Medium] [Blog]
 - <http://bit.ly/1XU9uZH> [Medium] [Blog]
- Rate Limiter
 - 源码阅读: <https://github.com/jsocol/django-ratelimit>

- <http://www.lintcode.com/problem/rate-limiter/>
- <http://www.lintcode.com/problem/web-logger/>

附录: NoSQL, 也就是所谓的分布式数据库

- 分布式数据库解决的问题
 - Scalability
- 分布式数据库还没解决很好的问题
 - Query language
 - Secondary index
 - ACID transactions
 - Trust and confidence

| | IOPS | Latency | Throughput | Capacity |
|------------|--------|---------|------------|----------|
| Memory | (10M) | 100ns | 10GB/s | 100GB |
| Flash | (100K) | (10us) | 1GB/s | 1TB |
| Hard Drive | 100 | 10ms | 100MB/s | 1TB |

| | Rack | Datacenter | 远距离 |
|-------------|-------|------------|----------|
| P99 Latency | <1ms | 1ms | 100ms + |
| Bandwidth | 1GB/s | 100MB/s | 10MB/s - |