分布式 Key Value Store 漫谈

V1.0

广州技术沙龙(09/08/08)

Tim Yang

http://timyang.net/

Agenda

- Key value store 漫谈
  - MySQL / Sharding / K/V store
  - K/V store性能比较
- Dynamo 原理及借鉴思想
  - Consistent hashing
  - Quorum (NRW)
  - Vector clock
  - Virtual node
- 其他话题

说明

- 不复述众所周知的资料
  - 不是Key value store知识大全
- 详解值得讲解或有实践体会的观点

场景

- 假定场景为一IM系统，数据存储包括
  - 1. 用户表(user)
    - {id, nickname, avatar, mood}
  - 2. 用户消息资料(vcard)
    - {id, nickname, gender, age, location…}
  - 好友表(roster)
    - {[id, subtype, nickname, avatar, remark],[id2,…],…}

单库单表时代

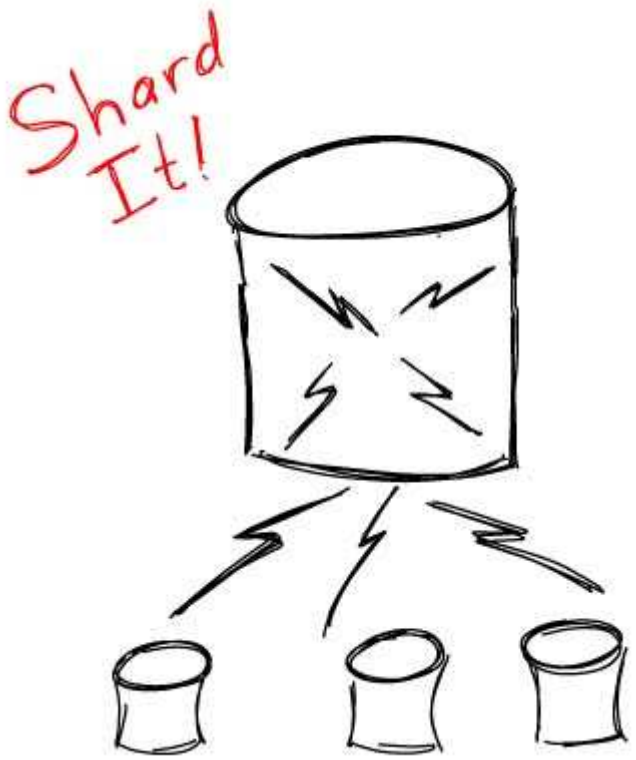
- 最初解决方案
  - 单库单表, MySQL
- 随着用户增长，将会出现的问题
  - 查询压力过大
- 通常的解决方案
  - MySQL replication及主从分离

单库单表时代

- 用户数会继续增大，超出单表写的负载
- Web 2.0, UGC, UCD的趋势，写请求增大，接近读请求
  - 比如读feed, 会有 “like” 等交互需要写
- 单表数据库出现瓶颈，读写效率过低

分库分表时代

- 将用户按ID(或其他字段)分片到不同数据库
- 通常按取模算法 hash() mod n
- 解决了读写压力问题

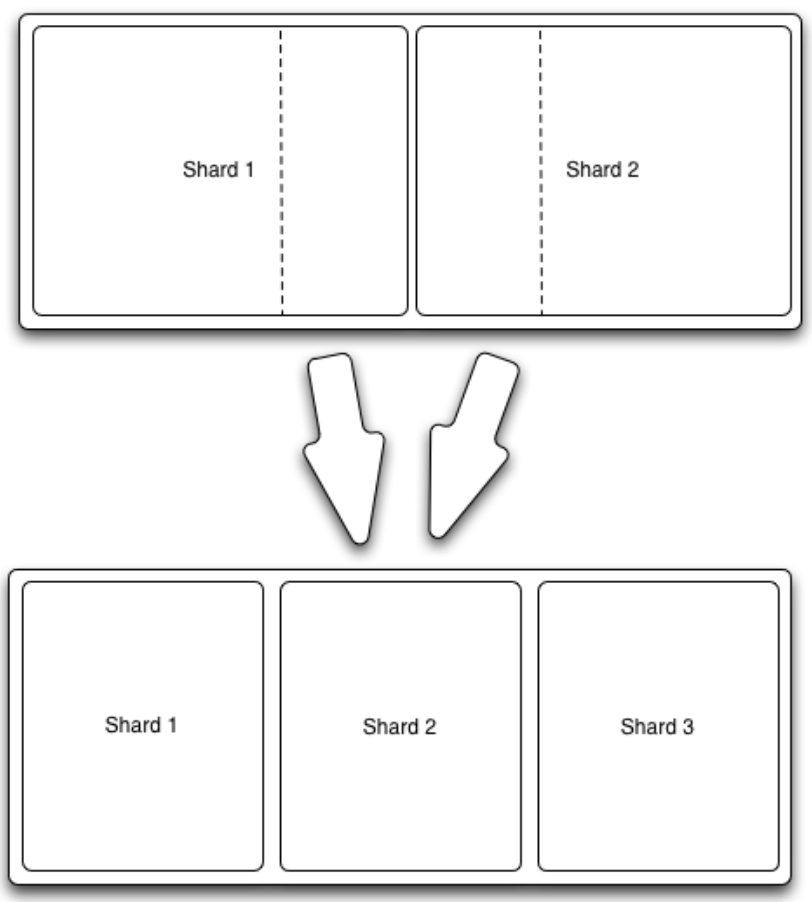


但是，Shard ≠ 架构设计

- 架构及程序人员的精力消耗在切分上
- 每一个新的项目都是重复劳动

不眠夜-resharding

通知：我们需要21:00-7:00停机维护



- 有办法避免吗？

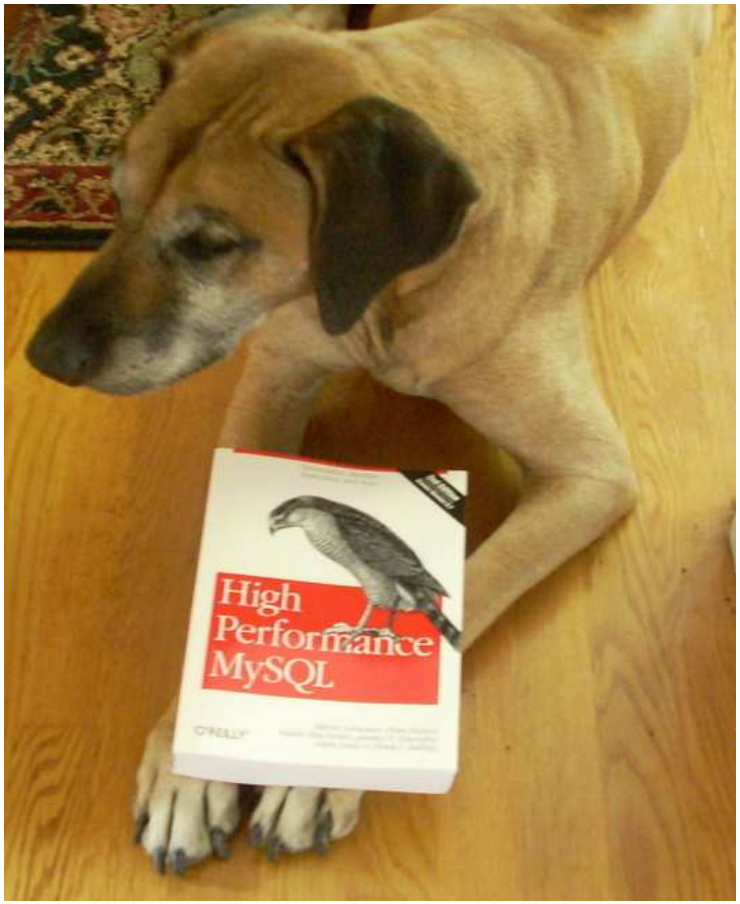
Sharding framework

- 框架，如hivedb
  - –隔离分库的逻辑
  - –期望对程序透明，和单库方式编程
- 没有非常成功的框架
  - –数据存储已经类似key/value
  - –期望用SQL方式来用，架构矛盾

- 框架之路也失败了，为什么？

分库分表过时了

- 无需继续深入了解那些切分的奇巧淫技
- nosql!



Key value时代

- 我们需要的是一个分布式，自扩展的storage
- Web应用数据都非常适合key/value形式
  - User, vcard, roster 数据
    - {user_id: user_data}

百家争鸣

- **Berkeley DB**(C), MemcacheDB(C)
- Bigtable, Hbase(Java), Hypertable(C++, baidu)
- Cassandra(Java)
- CouchDB(Erlang)
- Dynamo(Java), Kai/Dynomite/E2dynamo(Erlang)
- MongDB
- PNUTS
- Redis(C)
- **Tokyo Cabinet**(C)/Tokyo Tyrant/LightCloud
- Voldemort(Java)

问题

- Range select：
  - 比如需删除1年未登录的用户
- 遍历
  - 比如需要重建索引
- Search
  - 广州，18-20

- 没有通用解决方法，依赖外部
- Search
  - –Lucene
  - –Sphinx

非分布式key/value store

- 通过client hash来实现切分
- 通过replication来实现backup, load balance
- 原理上和MySQL切分并无区别，为什么要用？
  - 读写性能
  - 简洁性 (schema free)

Key store vs. MySQL

- 性能
  - Key store读写速度几乎相同 O(1)
    - O(1) vs. O(logN)
  - 读写性能都比MySQL快一个数量级以上
- 使用方式区别
  - MySQL: Relational ⇔ Object
  - Key store: Serialize ⇔ De-serialize

非分布式k/v缺少

- 自扩展能力，需要关心数据维护
  - 比如大规模MCDB部署需专人维护
- Availability, “always on”
- Response time, latency
  - No SLA(Service Level Agreement)
- Decentralize
  - Master/slave mode

产品

- Berkeley db及其上层产品, 如memcachedb
- Tokyo cabinet/Tyrant及上层产品，如LightCloud
- Redis及上层产品
- MySQL, 也有用MySQL来做，如friendfeed

分布式K/V store

- Dynamo (Amazon)
- **Cassandra (facebook)**
- **Voldemort (LinkedIn)**
- PNUTS (Yahoo)
- Bigtable (Google)
- **HyperTable(Baidu)**

(* 粗体为开源产品)

Benchmark

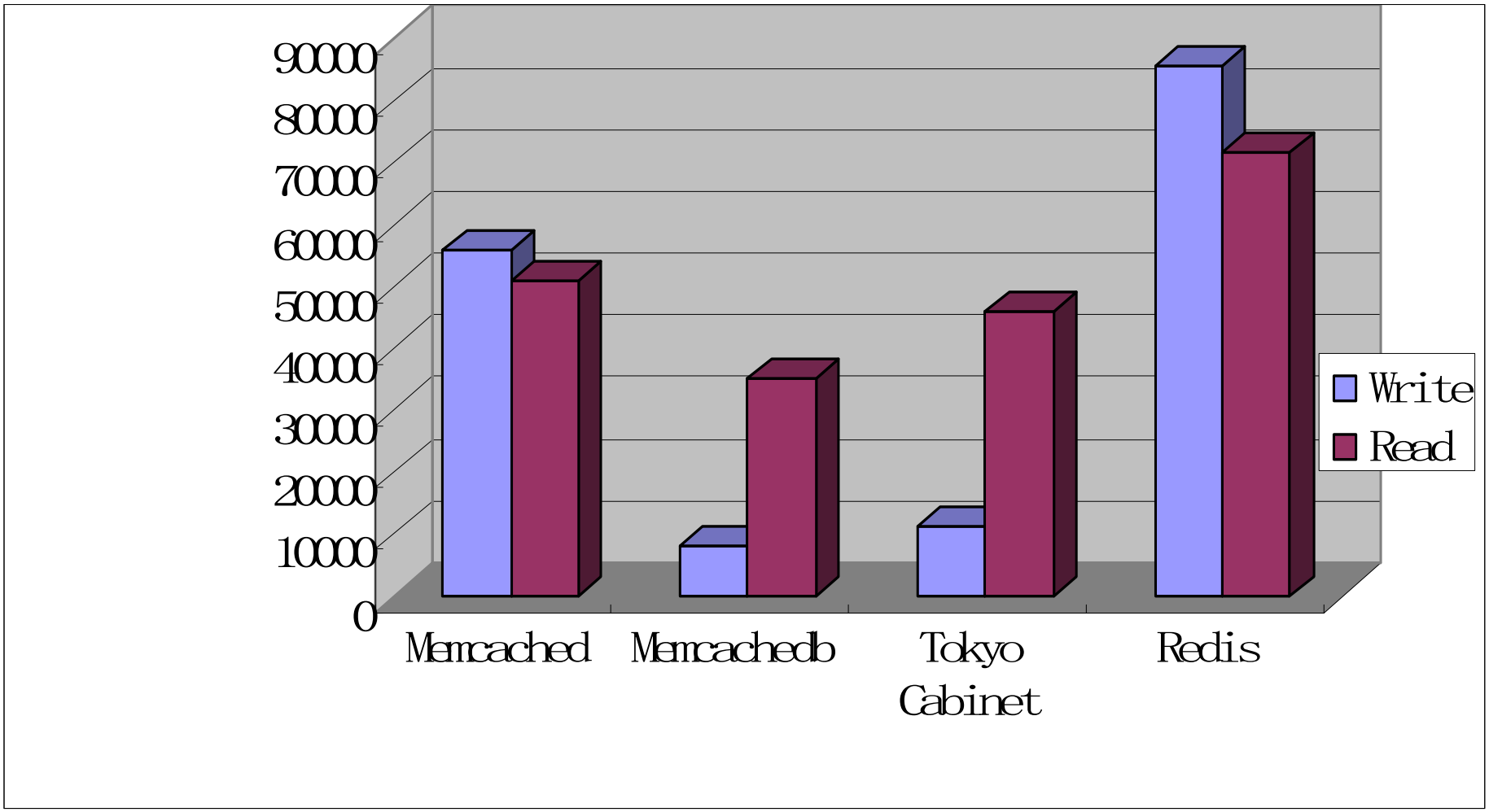
- Key value store
  - Berkeley db - memcachedb
  - Tokyo cabinet - Tyrant
  - Redis
- 测试环境
  - XEON 2*4Core/8G RAM/SCSI
  - 1 Gigabit Ethernet

Benchmark

- Server使用同一服务器、同一硬盘
- Client为局域网另外一台,16线程
- 都是用默认配置，未做优化
  - Memcached 1.2.8
  - bdb-4.7.25.tar.gz
  - memcachedb-1.2.1(参数 -m 3072 –N –b xxx)
  - tokyocabinet-1.4.9, tokyotyrant-1.1.9.tar.gz
  - redis-0.900_2.tar.gz (client: jredis)

- 三大高手
  - –Tokyo cabinet
  - –Redis
  - –Berkeley DB
- 究竟鹿死谁手？

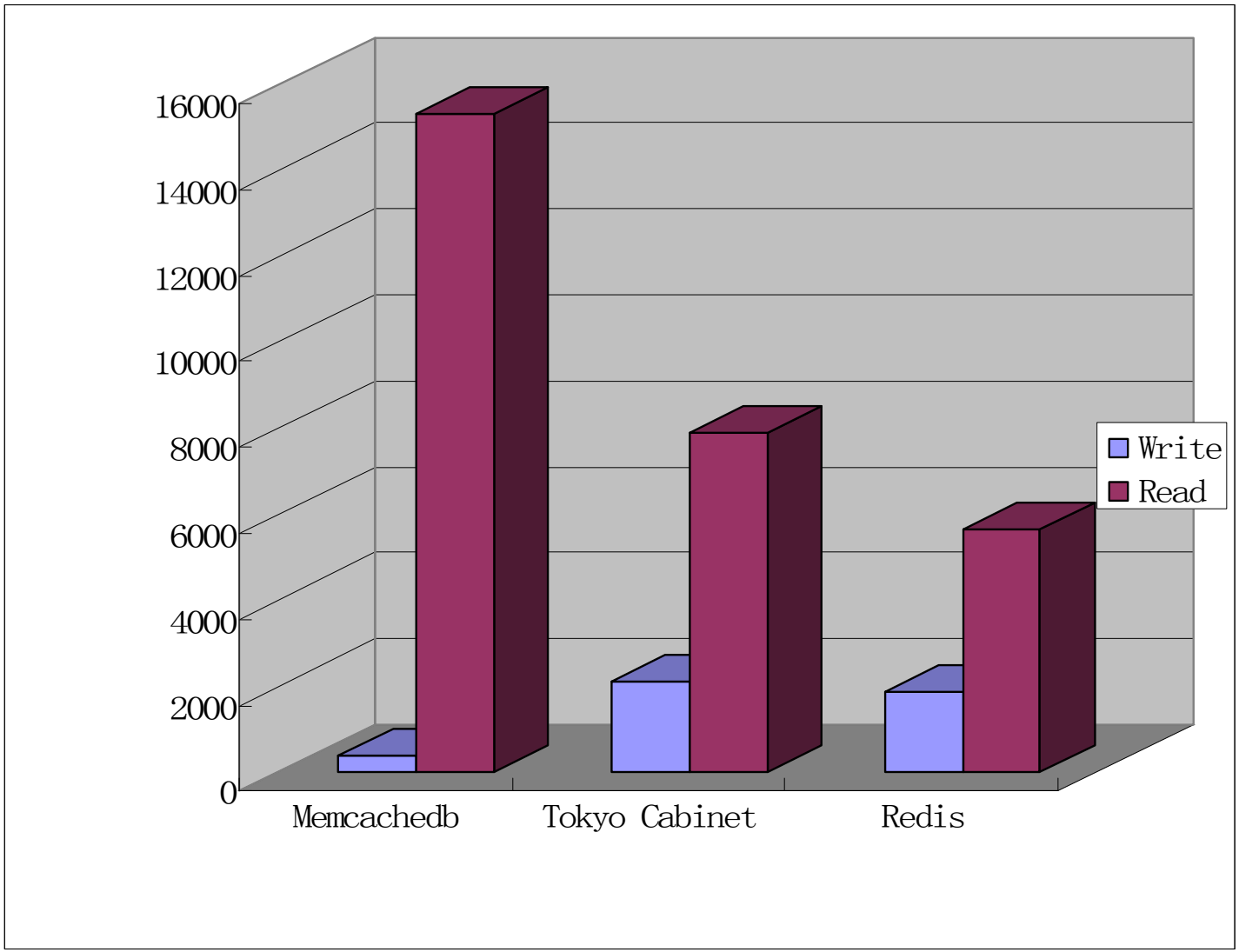
100 bytes, 5M rows request per second



Requests per second, 16 threads

Store	Write	Read
Memcached	**55,989**	**50,974**
Memcachedb (bdb)	**8,264**	**35,260**
Tokyo Cabinet/Tyrant	**11,480**	**46,238**
Redis	**85,765**	**71,708**

20k data, 500k rows request per second



Requests per second, 16 threads

Store	Write	Read
Memcachedb	**357**	**15,318**
Tokyo Cabinet	**2,080**	**7,900**
Redis	**1,874**	**5,641**

- 到此，我们已经了解了
  - 繁琐的切分
  - Key value store的优势
  - Key value store的性能区别
- 可以成为一个合格的架构师了吗
- 还缺什么

新需求

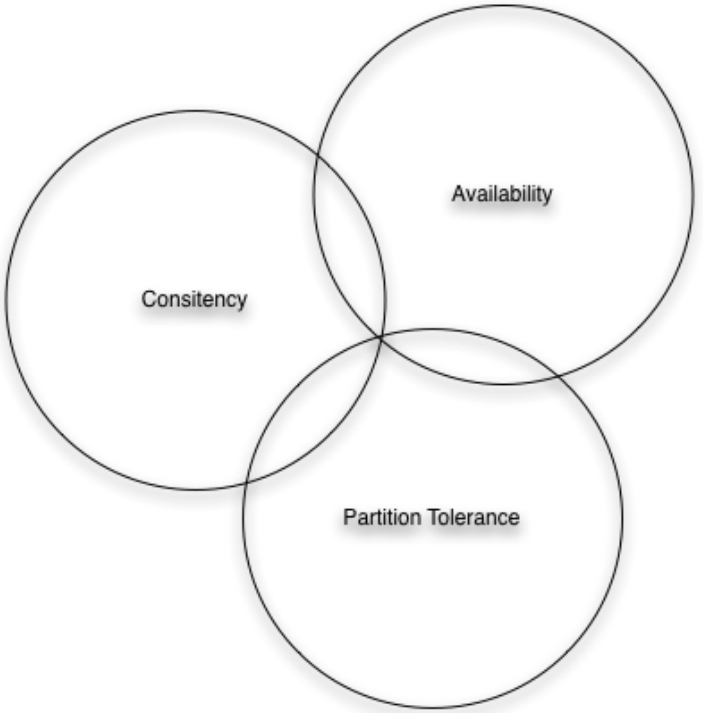
- 如何设计一个分布式的有状态的服务系统？
  - –如IM Server, game server
  - –用户分布在不同的服务器上，但彼此交互
- 前面学的“架构”毫无帮助

分布式设计思想红宝书

- **Dynamo: Amazon's Highly Available Key-value Store**
- Bigtable: A Distributed Storage System for Structured Data

CAP理论

- 分布式领域CAP理论，Consistency(一致性), Availability(可用性), Partition tolerance(分布) 三部分在系统实现只可同时满足二点，没法三者兼顾。
- 架构设计师不要精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍



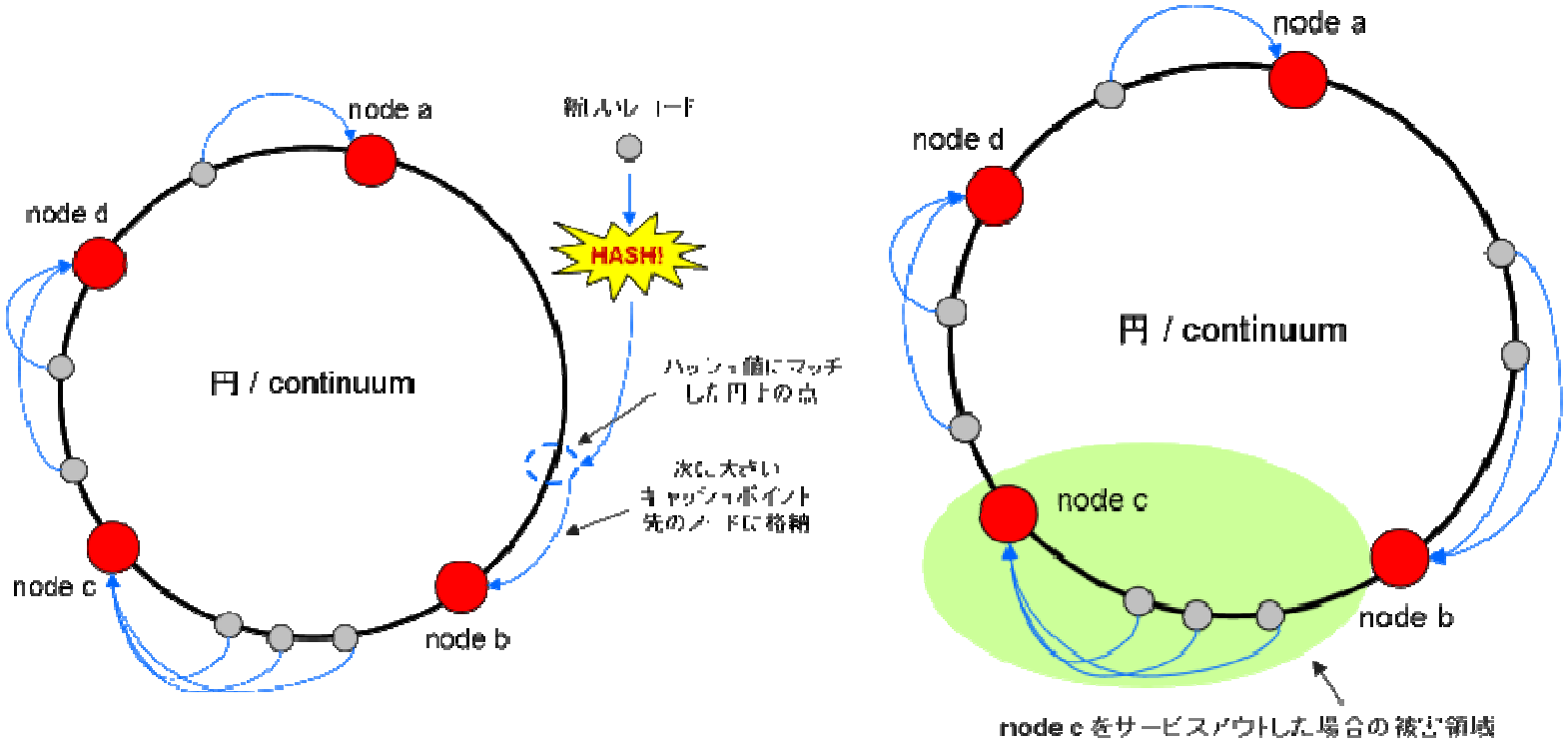
Dynamo 思想

- The A.P. in CAP
  - 牺牲部分consistency
  - “Strong consistency reduce availability”
  - Availability: 规定时间响应(e.g. < 30ms)
- Always writable
  - E.g. shopping cart
- Decentralize

1. Consistent Hashing

- 传统的应用
  - 如memcached, hash() mod n
  - Linear hashing
- 问题
  - 增删节点，节点失败引起所有数据重新分配
- Consistent hash如何解决这个问题？

1. Consistent Hashing



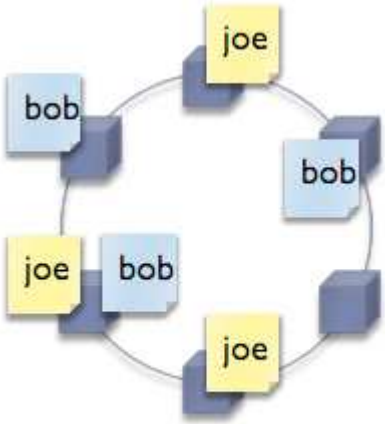
* Image source: http://alpha.mixi.co.jp/blog/?p=158

如何确保always writable

- 传统思路
  - 双写？
  - 如何处理版本冲突，不一致？

2. Quorum NRW

- NRW
  - N: # of replicas
  - R: min # of successful reads
  - W: min # of successful write
- 只需 W + R > N



Dynamo of N = 3

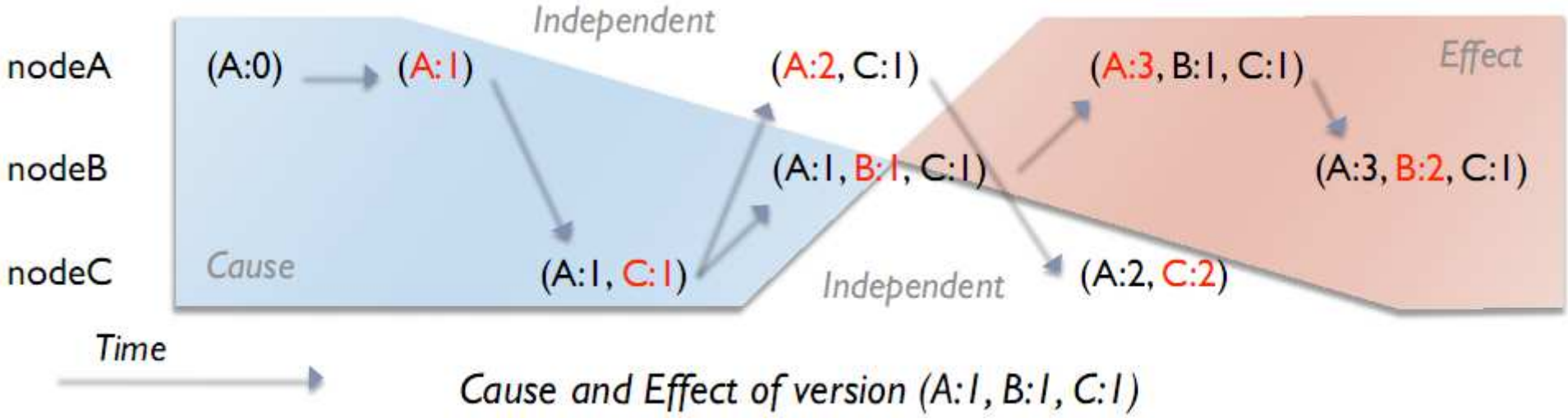
场景

- 典型实现：N=3, R=2, W = 2
- 几种特殊情况
- W = 1, R = N, 场景？
- R = 1, W = N , 场景？
- W = Q, R = Q where Q = N / 2 + 1

- 如果N中的1台发生故障，Dynamo立即写入到preference list中下一台，确保永远可写入
- 问题：版本不一致
- 通常解决方案：timestamp，缺点？
  - Shopping cart

3. Vector clock

- Vector clock
  - 一个记录(node, counter)的列表
  - 用来记录版本历史



* Image source: http://www.slideshare.net/takemaru/kai-an-open-source-implementation-of-amazons-dynamo-472179

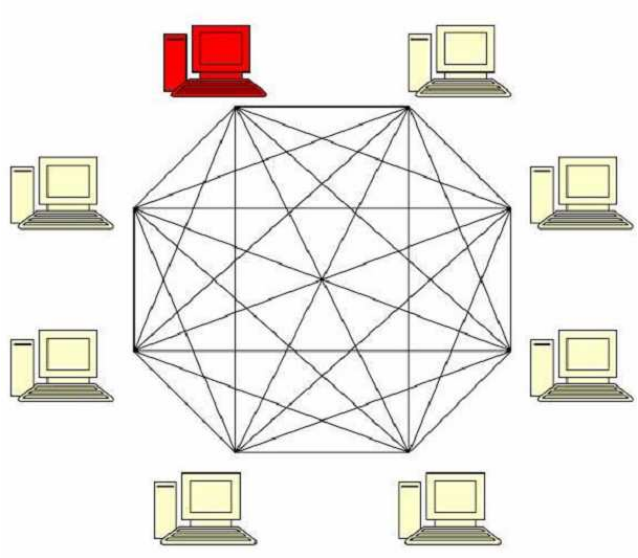
Reconciliation, Merge version

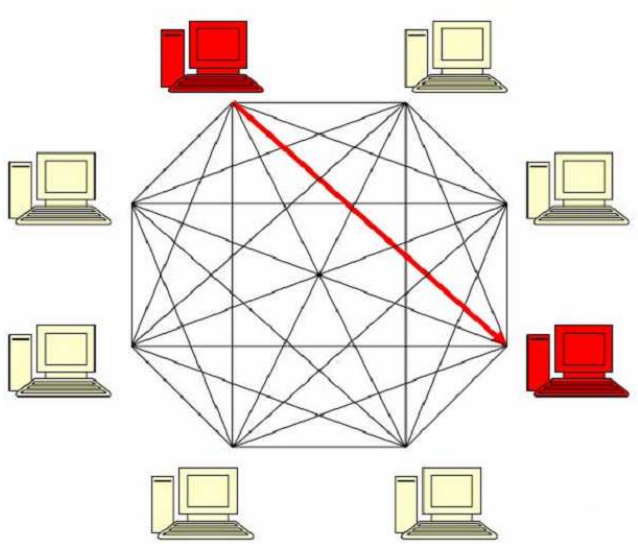
- 如何处理(A:3,B:2,C1)？
  - Business logic specific reconciliation
  - Timestamp
  - High performance read engine
    - R = 1, W = N
- 越变越长？ Threshold=10

如果节点临时故障

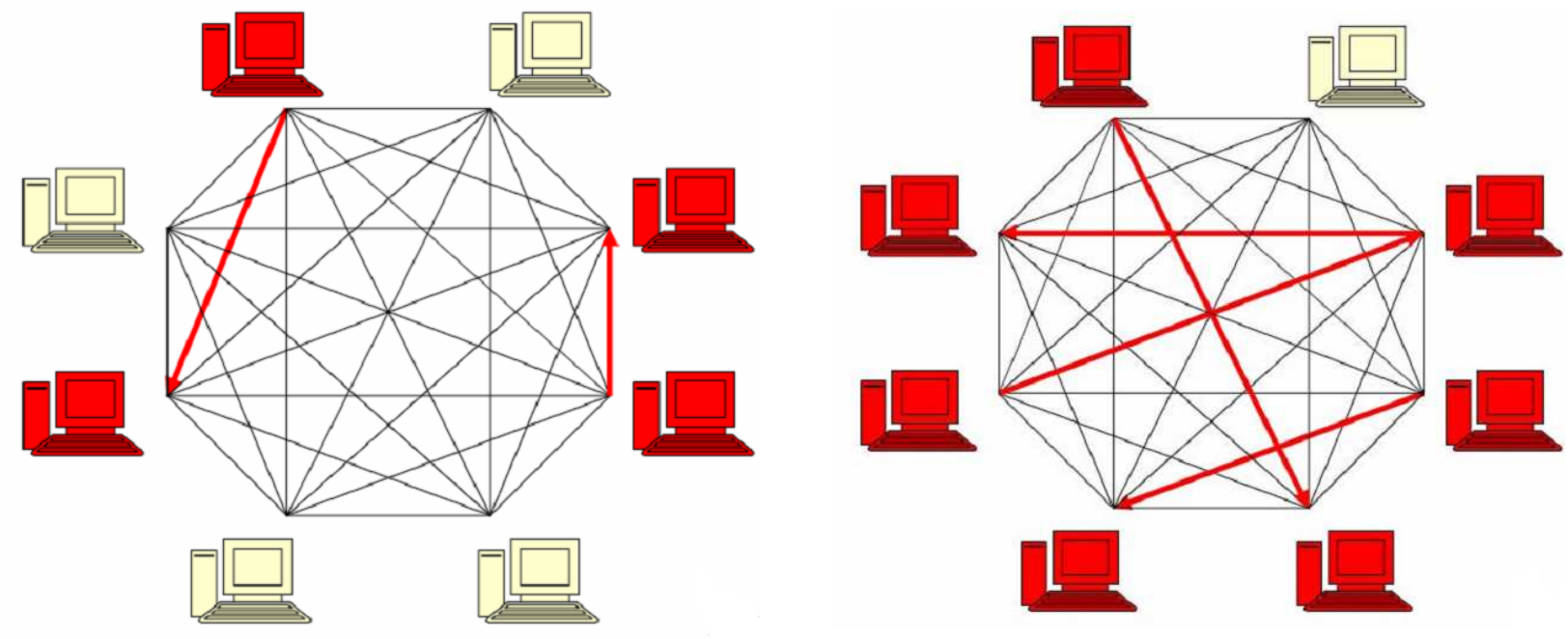
- 典型方案：Heart-beat, ping
- Ping如何处理无中心，多节点？
- 如何保证SLA?
  - –300ms没响应则认为对方失败
- 更好的方案？

gossip





gossip



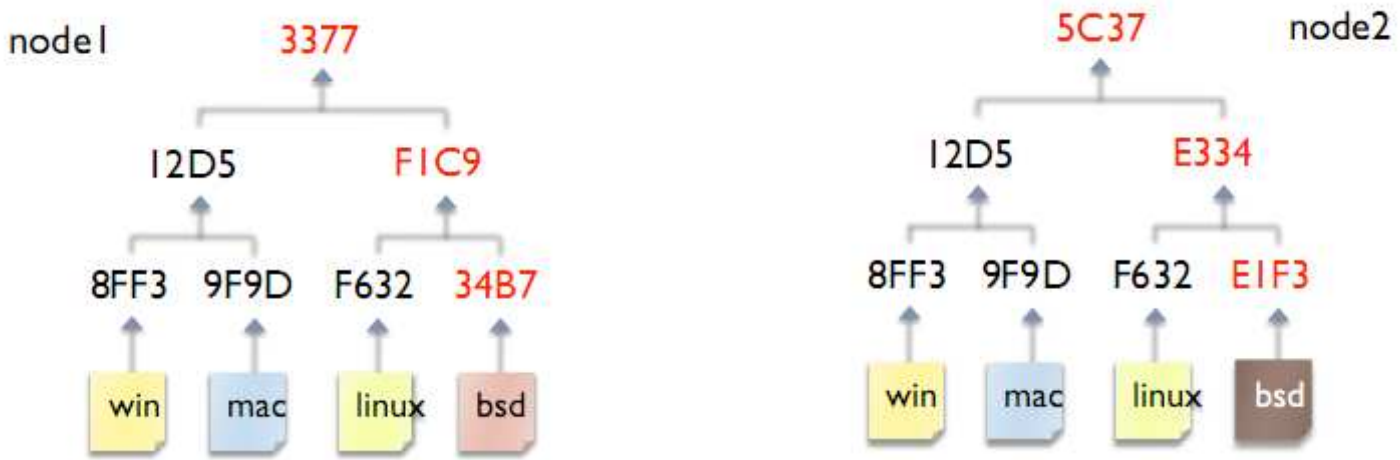
* Image source: http://www.slideshare.net/Eweaver/cassandra-presentation-at-nosql

临时故障时写入的数据怎么办

- Hinted handoff

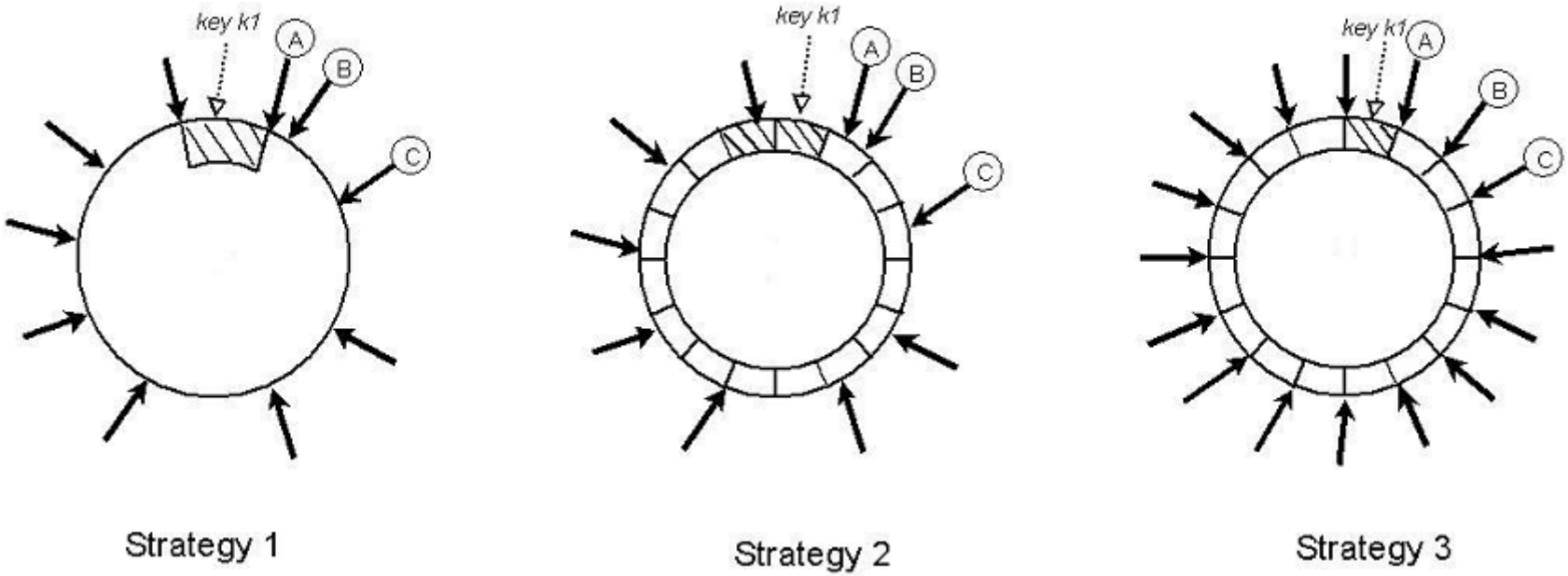
由于存在故障因素
如何检查数据的一致性

- Merkle tree



4. Virtual Node

- Consistent hashing, 每个节点选一个落点，缺点？



增删节点怎么办

- 传统分片的解决方法：手工迁移数据
- Dynamo: replication on vnode

IDC failure

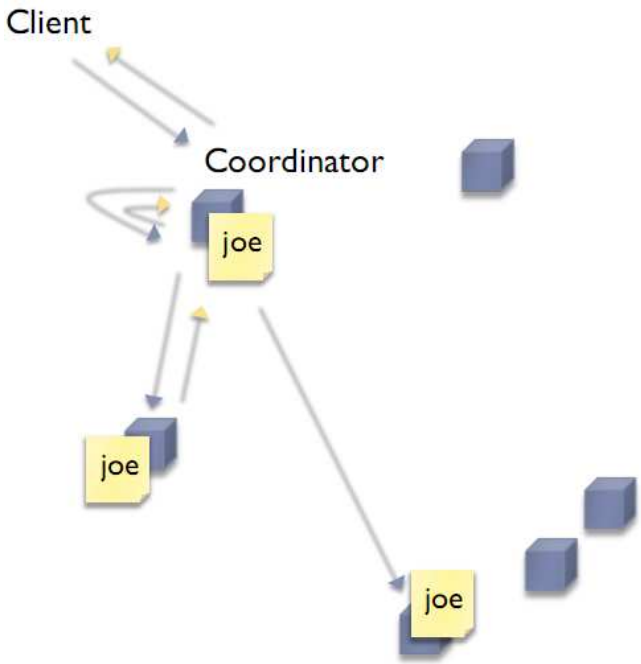
- Preference list
- N nodes must in different data center

程序如何组织上述一切

- Node has 3 main component
  - Request coordination
  - Membership
  - Failure detection

Client driven vs. server driven

- Coordinator
  - Choose N nodes by using consistent hashing
  - Forwards a request to N nodes
  - Waits responses for R or W nodes, or timeout
  - Check replica version if get
  - Send a response to client



get/put operations for N,R,W = 3,2,2

Dynamo impl.

- Kai, in Erlang
- E2-dynamo, in Erlang
- Sina DynamoD, in C
  - N = 3
  - Merkle tree based on commit log
  - Virtual node mapping saved in db
  - Storage based on memcachedb

- 已经学习了4种分布式设计思想
  - –Consistent hashing
  - –Quorum
  - –Vector clock
  - –Virtual node

- 只是分布式理论冰山一角
- 但已经够我们使用在很多场合
- 实战一下？

分布式Socket Server
可借鉴的思想

- 节点资源分布
- 假定5个节点采用取模分布, hash() mod n
- 某台发生故障，传统解决解决方法
  - 马上找一个替换机启动
  - 将配置文件节点数改成4，重启
  - 都需影响服务30分钟以上

应用更多分布式设计

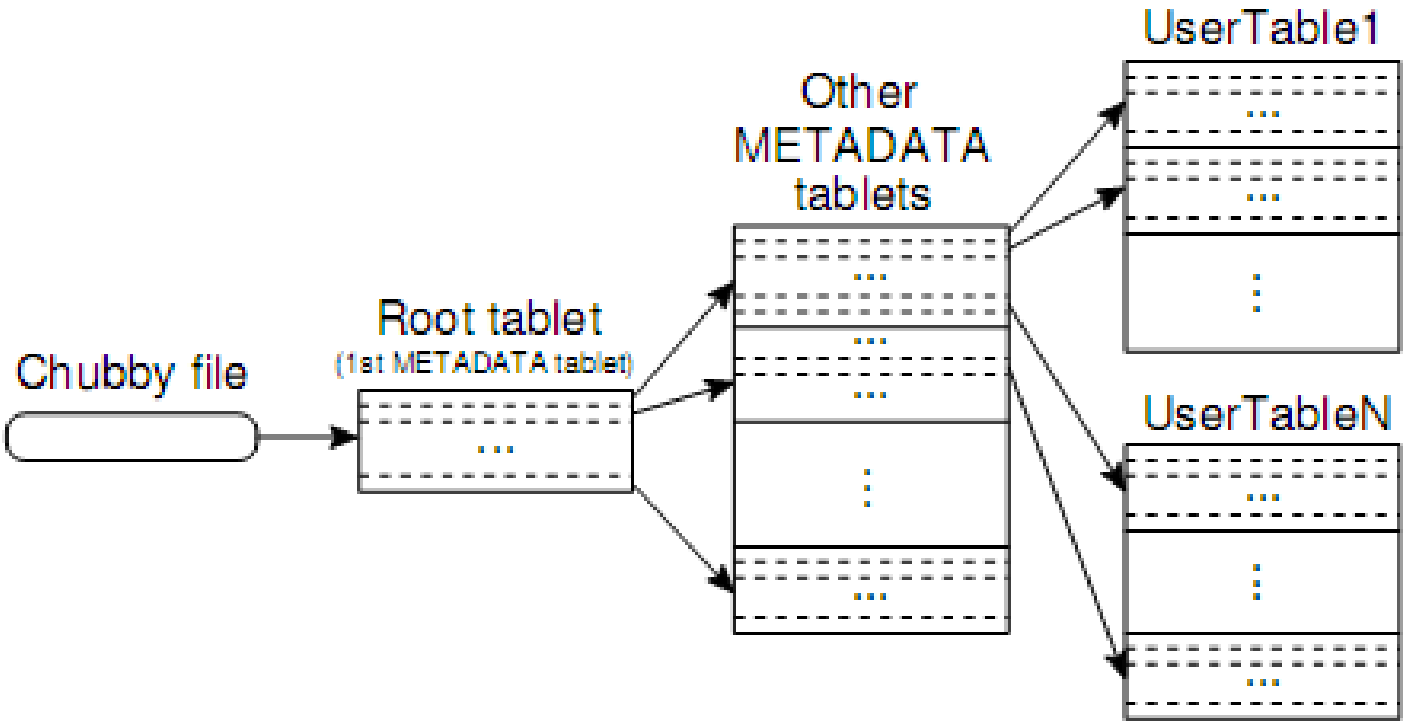
- 1. consistent hashing
- 2. 临时故障？Gossip
- 3. 临时故障时候负载不够均匀？vnode

Hinted handoff

- 失败的节点又回来了，怎么办？
- Node 2 back, but user still on Node1
- Node 1 transfer handoff user session to Node2

其他话题？

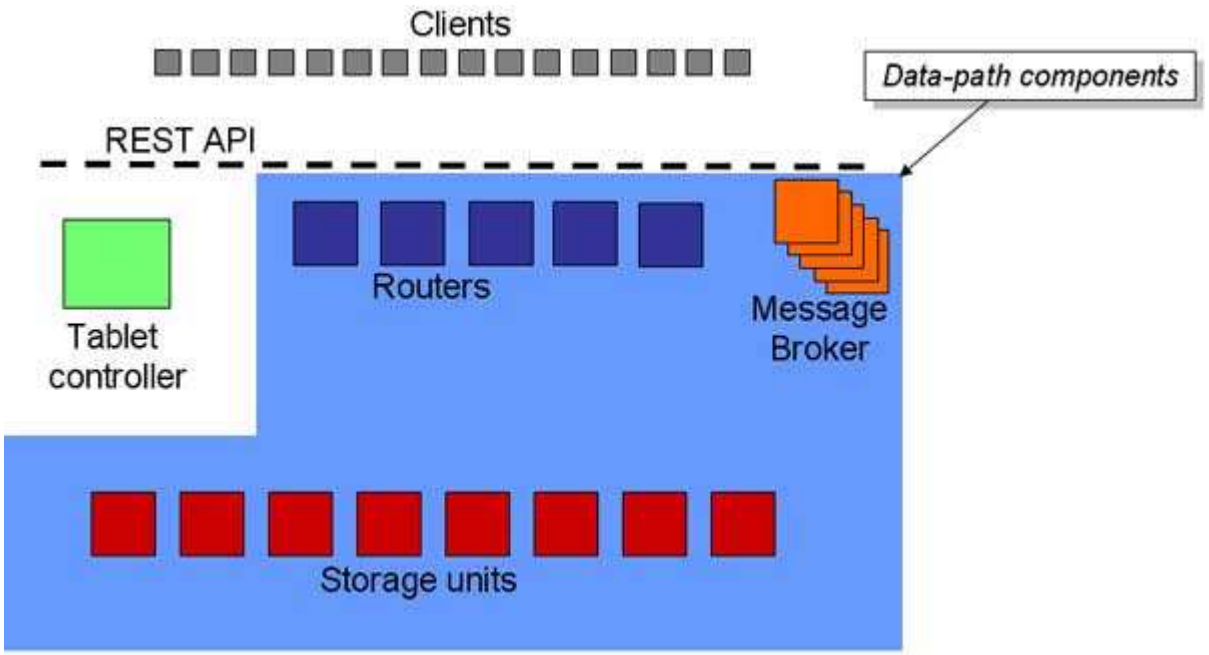
Bigtable


UserTable1
Other
METADATA
tablets
Root tablet
(1st METADATA tablet)
Chubby file
UserTableN

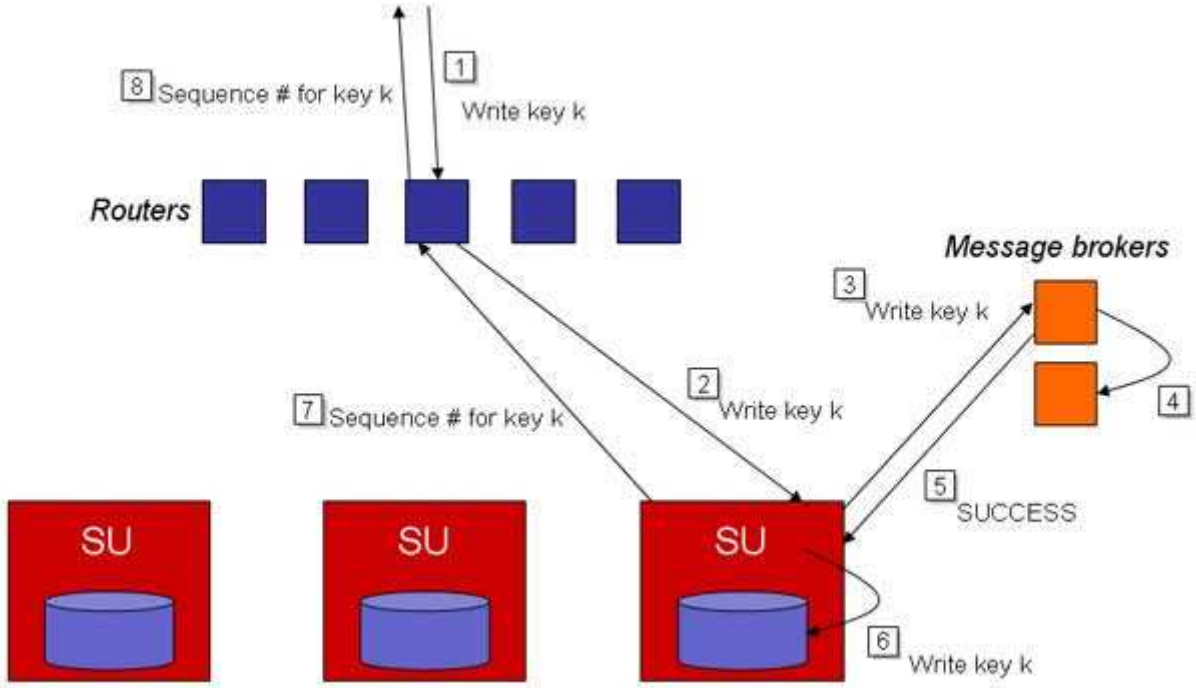
Bigtable

- 相比Dynamo, Bigtable是贵族式的
  - –GFS vs. bdb/mysql
  - –Chubby vs. consistent hashing
- Dynamo更具有普遍借鉴价值

PNUTS


Clients
Data-path components
REST API
Routers
Message Broker
Tablet controller
Storage units

PNUTS


8 Sequence # for key k
1 Write key k
Routers
Message brokers
3 Write key k
2 Write key k
7 Sequence # for key k
4
5 SUCCESS
SU
SU
SU
6 Write key k

进阶指南-了解源码

- 如果关注上层分布式策略，可看
  - Cassandra (= Dynamo + Bigtable)
- 关注底层key/value storage，可看
  - Berkeley db
  - Tokyo cabinet

Language

- 实现分布式系统的合适语言？
- Java, Erlang, C/C++
  - Java, 实现复杂上层模型的最佳语言
  - Erlang, 实现高并发模型的最佳语言
  - C++, 实现关注底层高性能key value存储。

Q&A

Thanks you!

- Tim Yang: http://timyang.net/
- Twitter: xmpp