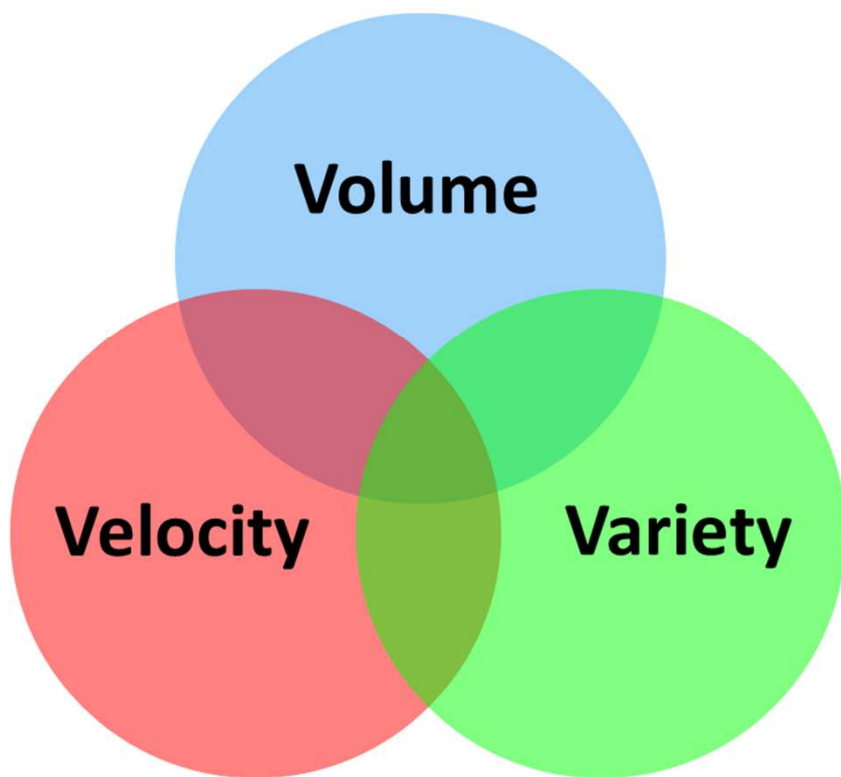


大数据系统与大规模数据分析

# 大数据存储系统 (2)



陈世敏

中科院计算所  
计算机体系结构  
国家重点实验室

©2015-2017 陈世敏

# Outline

- Key-Value Store
  - ❑ Dynamo
  - ❑ Bigtable / Hbase
  - ❑ Cassandra
- Distributed Coordination: ZooKeeper

# 为什么叫No-SQL?

- 这些系统大部分是由互联网公司研发的
  - 研发的目标是支持本公司的某类重要的应用
- 放弃使用关系型系统，转而开发专门的系统以支持目标应用
  - 由于应用规模巨大或数据种类繁多，当时的关系型产品不能完全支持
  - 并行数据库系统高配也通常只有几十台服务器
  - 而这些系统则使用成千上万台机器，和存储PB级的数据
- 针对目标应用进行开发，简化了许多关系型系统的功能，以提高系统性能和降低研发成本
  - 不支持（完全的）SQL
  - 不支持（完全的）ACID
  - 等等
- 所以，这类系统被称作No-SQL

# 为什么叫No-SQL?

- 那么关系型与No-SQL究竟孰优孰劣?

- 这个不能一概而论

- 关系型有其生命力，已经存在了40多年，还在被广泛的使用

- 优美的数学模型支持
    - SQL与ACID等都在实践中被证明了是非常有用的
    - 但是关系型系统的实现确实没有考虑到上述超大规模、多种数据类型

- No-SQL系统确实很好地支持了它们的目标应用

- 但是为了支持更加丰富的应用，人们发现已有的No-SQL系统的不足

- 所以，这两者将以某种方式融合

- 这种趋势已经出现

# Key-Value Store

- Key-Value store是一种分布式数据存储系统
  - 简而言之，数据形式为<key, value>，支持Get/Put操作
  - 实际上，多种不同的系统的数据模型和操作各有差异
- 我们将主要介绍三个系统
  - Dynamo: 由Amazon公司研发
  - Bigtable / HBase: Bigtable起源于Google公司, Hbase是开源实现
  - Cassandra: 由Facebook研发，后成为Apache开源项目

# Key-Value Store: Dynamo

- “*Dynamo: Amazon's Highly Available Key-Value Store.*”  
Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.  
(Amazon.com). **SOSP 2007.**
- 支持亚马逊公司电子商务平台上运行的大量服务
  - 例如, best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog
  - 存储这些服务的状态信息

# Dynamo数据模型和操作

- 最简单的<key, value>

- key = primary key: 唯一地确定这个记录
- value: 大小通常小于1MB

- 操作

- Put(key, version, value)
- Get(key) → (value, version)

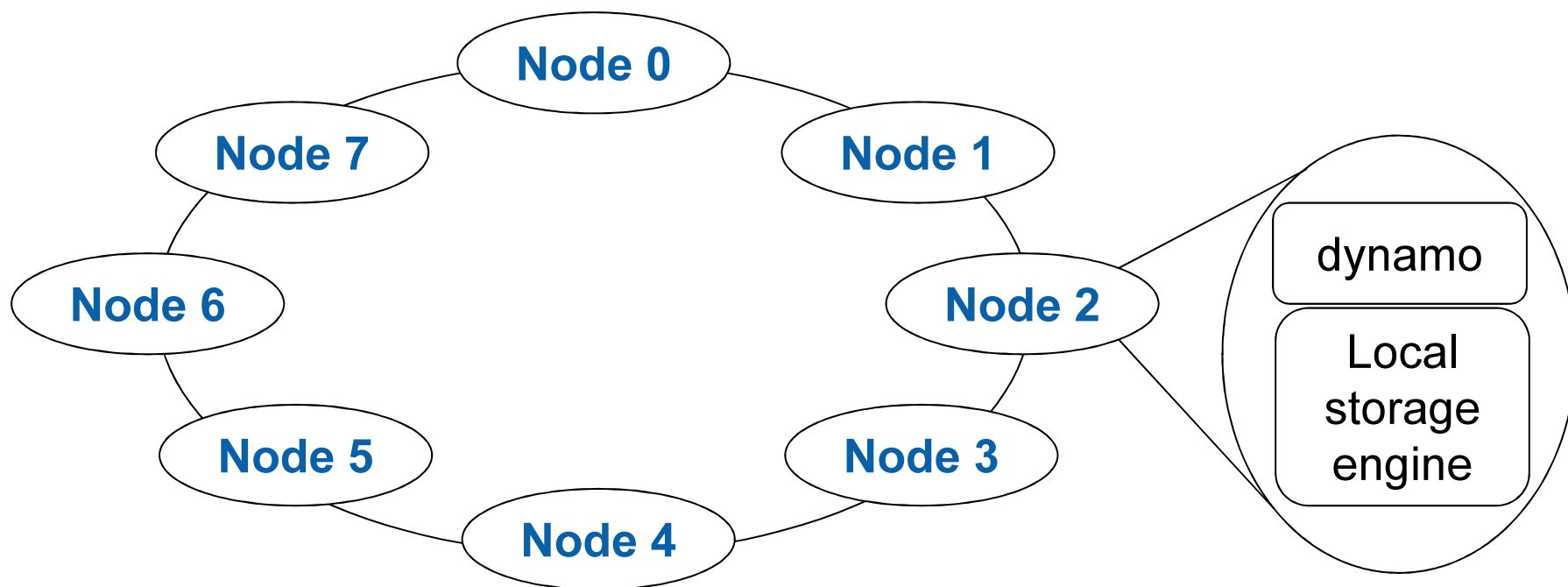
- ACID?

- 没有Transaction概念
- 仅支持单个<key,value>操作的一致性

修改多个<key, value>可能出现什么问题?

各种不一致情况, 要求上层应用设计时考虑这点

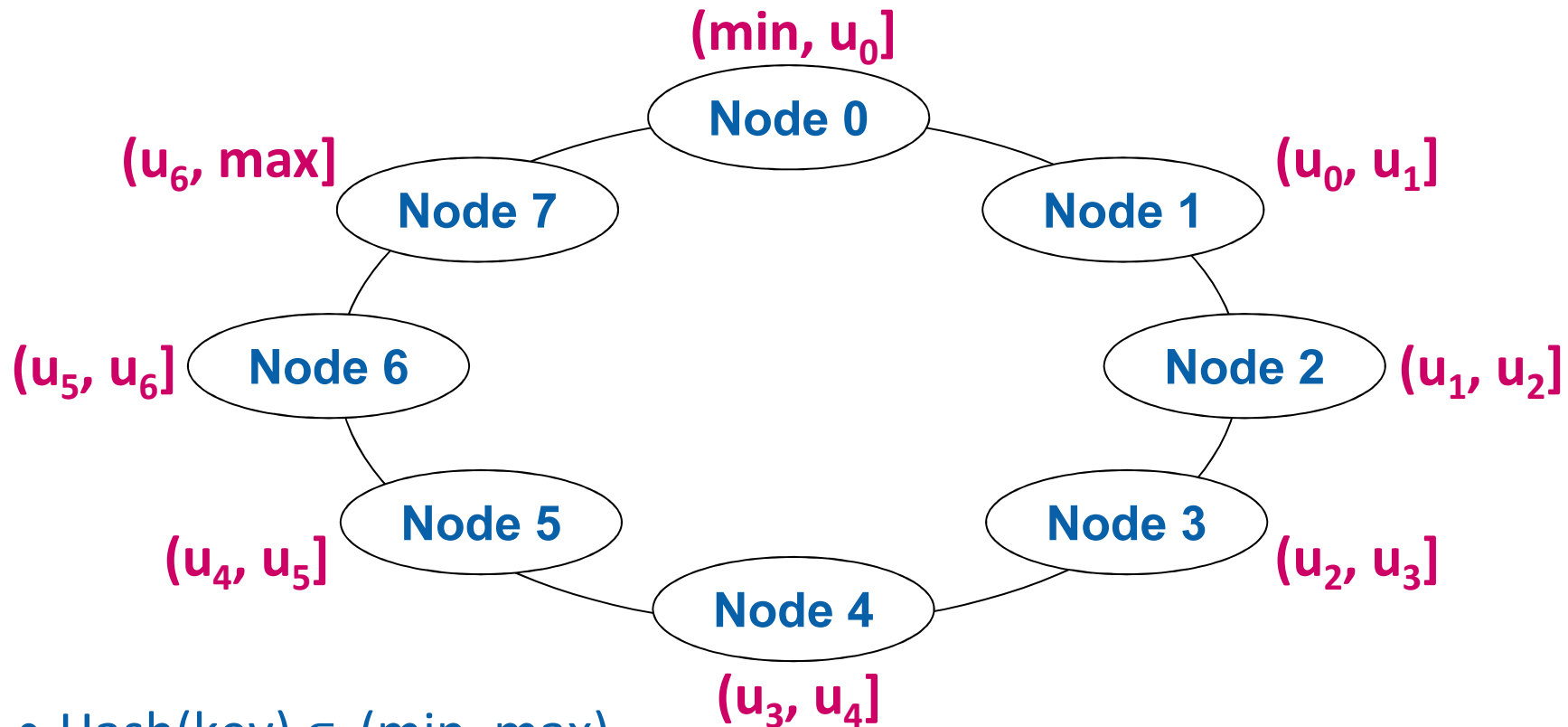
# Dynamo 系统结构



- 多个nodes互连形成分布式系统
- 每个node上由local storage engine + dynamo软件层组成
  - Local storage engine: Berkeley DB, 或MySQL, etc.
  - 用于存储<key, value>

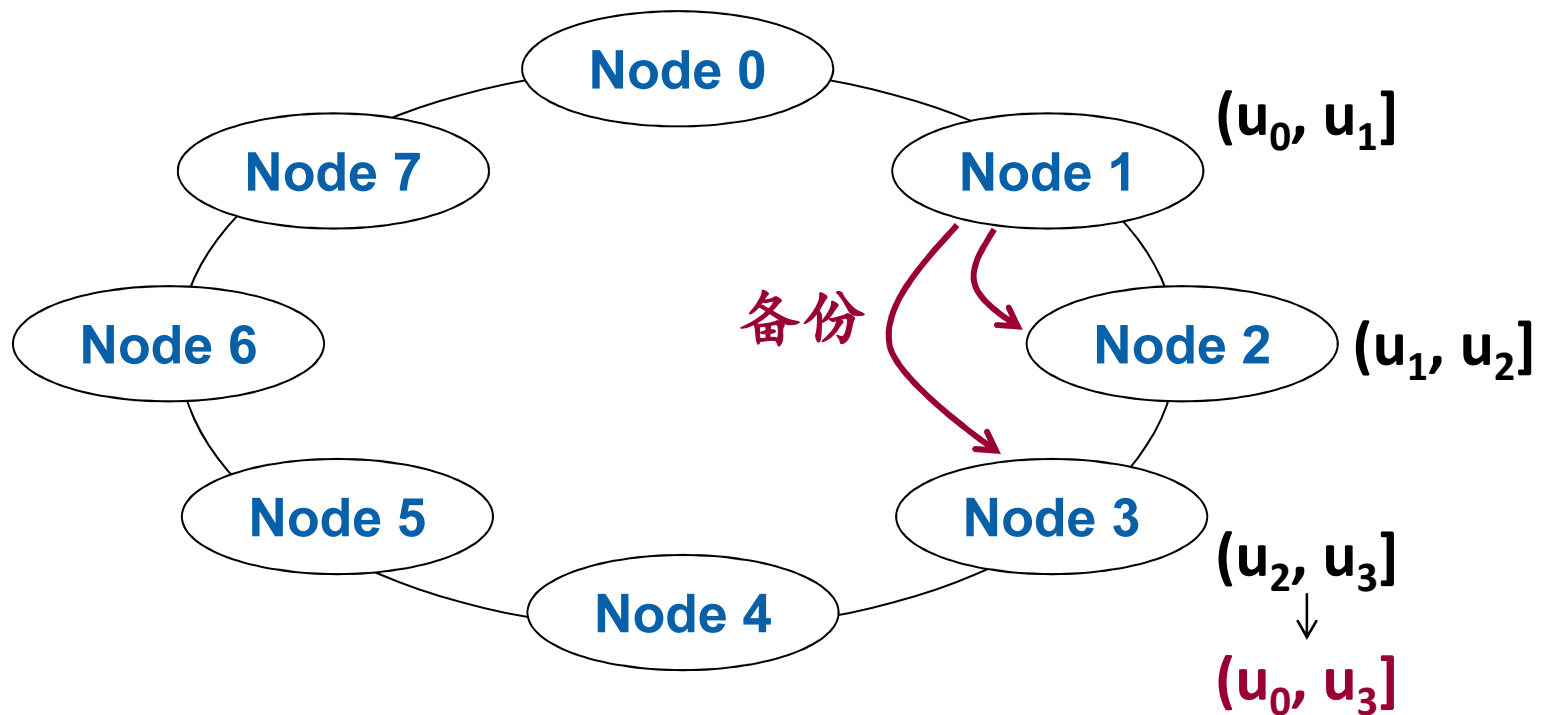


# Consistent Hashing (p2p的关键技术)



- $\text{Hash}(\text{key}) \in (\min, \max)$ 
  - 每个key对应为hash value区间中的一个点
- 为每个node设置一个值:  $\min < u_0 < u_1 < u_2 \dots < u_7 = \max$ 
  - 每个node对应一个区间的所有key

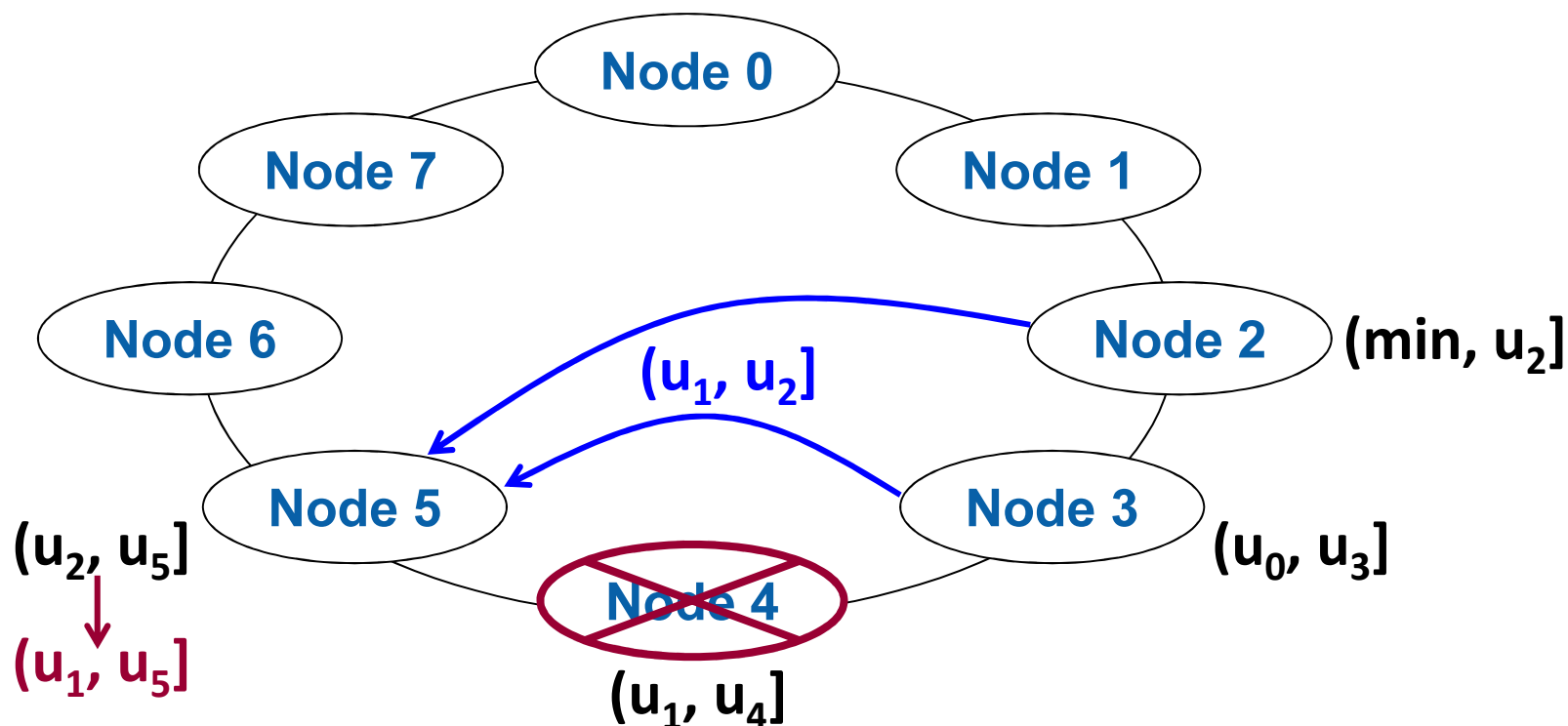
# Consistent Hashing: 备份



- 举例：3副本备份

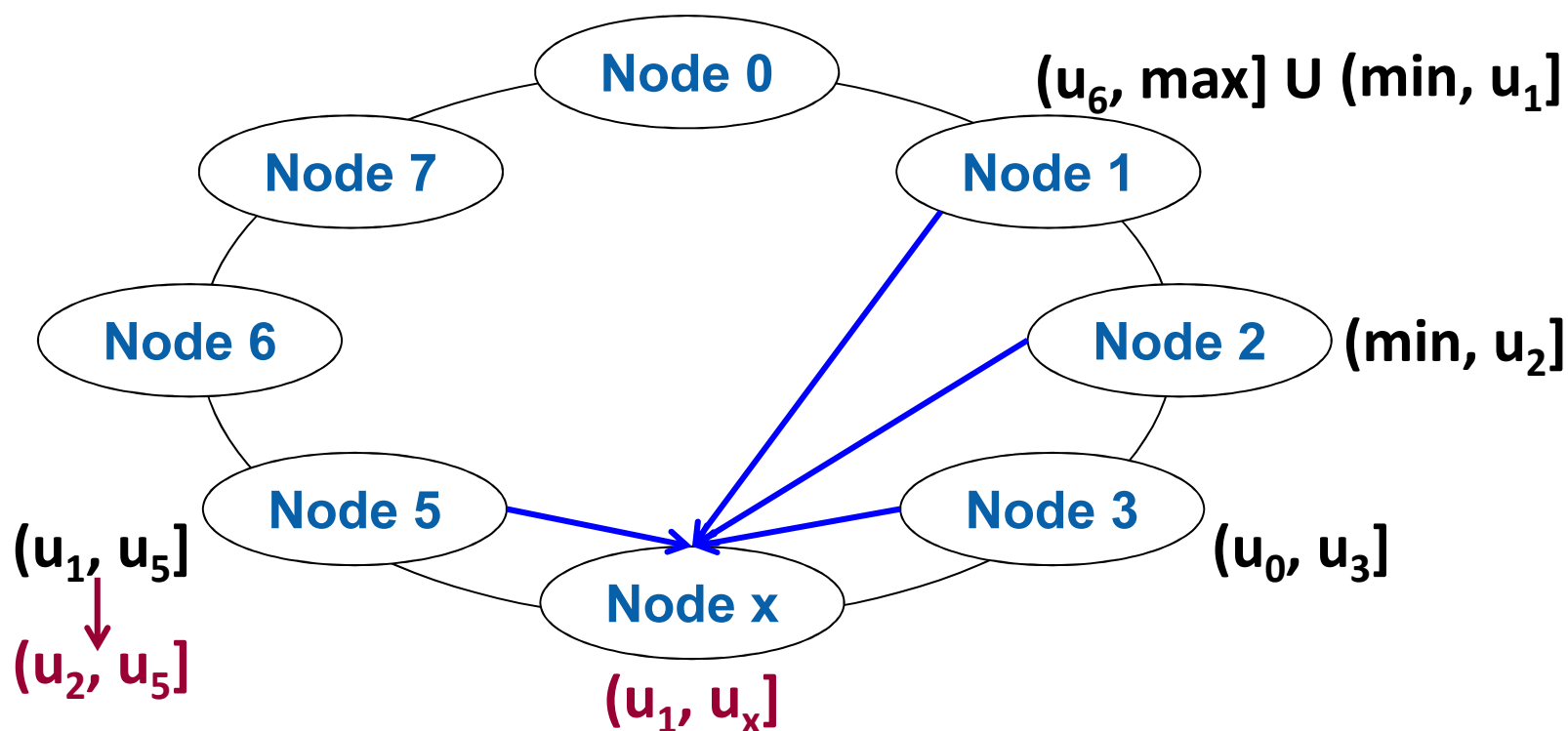
- Put到Node  $j$ 上的数据，要备份到Node  $j+1$ 和Node  $j+2$ 上
- 所以一个Node  $j$ 上实际存储的数据是  $(u_{j-3}, u_j]$

# Consistent Hashing: 减少一个node



- 改变区间，拷贝数据
- 对Node 6与node 7有类似修改

# Consistent Hashing: 增加一个node



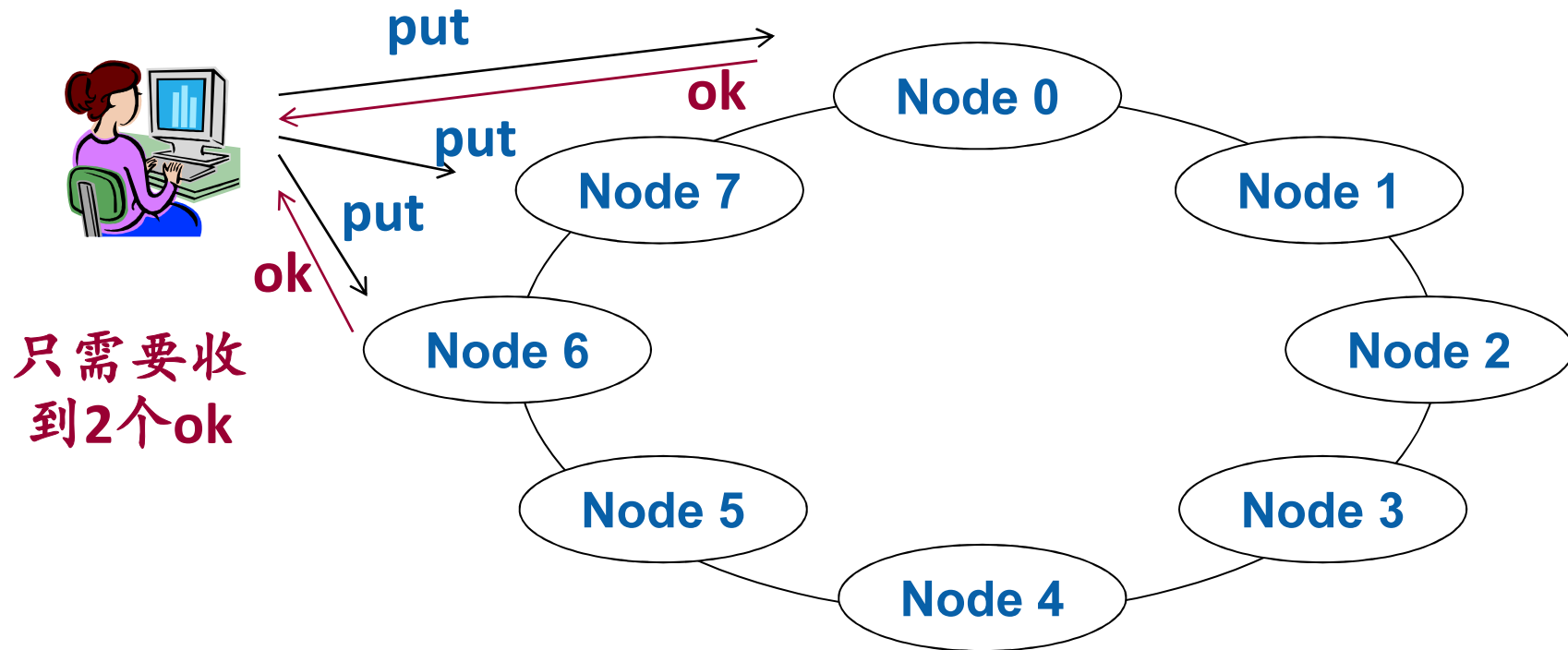
- 给新node赋值（假设： $u_x$ 在 $u_3$ 和 $u_5$ 之间）
- 改变区间，拷贝数据
- 对Node 6与node 7有类似修改

# Quorum机制

- Quorum: 翻译为法定人数
- 如果有N个副本，要求写的时候保证至少写了W个副本，要求读的时候至少从R个副本读了数据，满足 $R+W>N$ ，那么一定读到了最新的数据。
  - N: 副本个数
  - W: 至少写了W个副本
  - R: 至少从R个副本读了数据
- 用于实现读写的一致性
- (N, W, R): 例如(3, 2, 2)

# Put操作

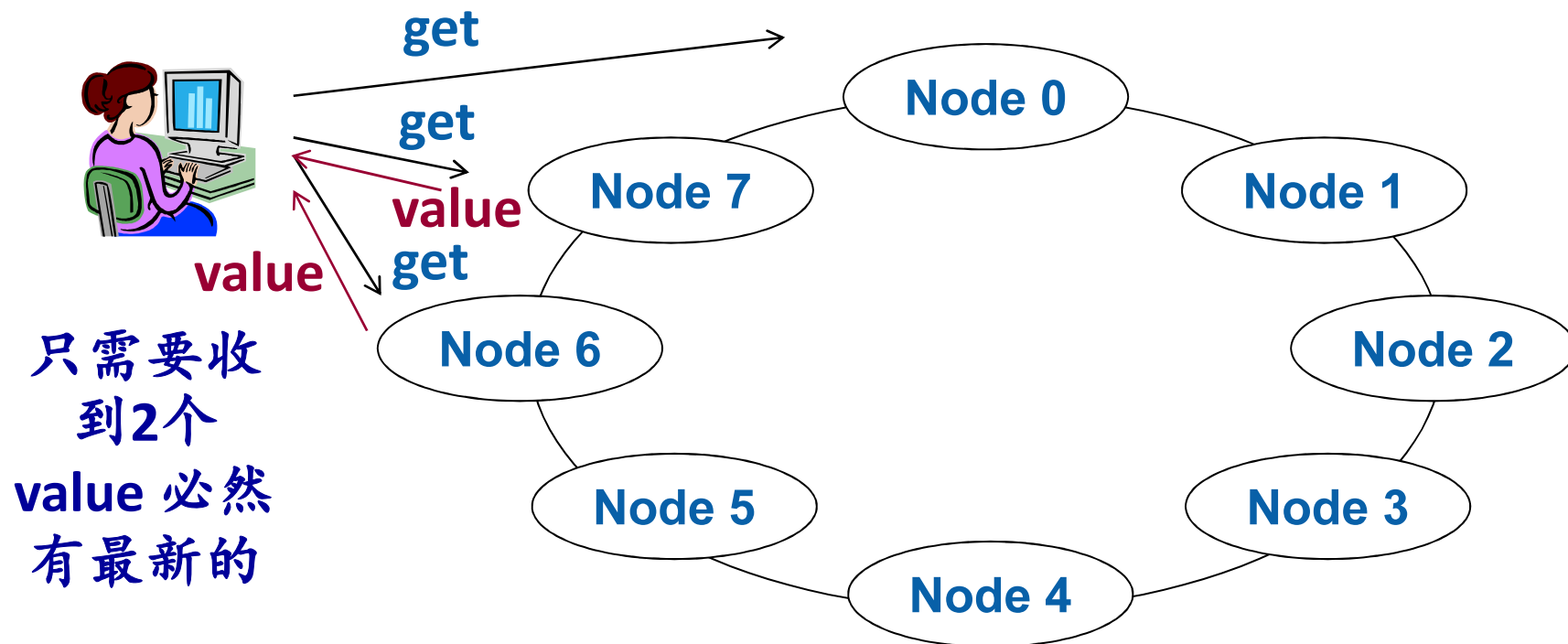
(N, W, R): 例如(3, 2, 2)



- Client根据hash(key)得到所有N个副本所在的节点
- Client向所有N个副本所在的节点发出put
- 等到至少W个节点完成的响应，就认为写成功

# Get操作

(N, W, R): 例如(3, 2, 2)



- Client根据hash(key)得到所有N个副本所在的节点
- Client向所有N个副本所在的节点发出get
- 等到至少R个节点的value, 就必然包含最新一次写的值

# Quorum设计

- $N=5$
- 哪些是可能的Quorum? ( $N$ ,  $R$ ,  $W$ )
  - $(5, 1, 5)$
  - $(5, 2, 4)$
  - $(5, 3, 3)$
  - $(5, 4, 2)$
  - $(5, 5, 1)$
  - .....
- $R$ 小, 那么读的效率就高
- $W$ 小, 那么写的效率就高



# Eventual Consistency

- Put操作并没有等待所有N个节点写完成
  - 可以提高写效率
  - 可以避免访问出错/下线的节点，提高系统可用性
- 系统总会最终保证每个<key,value>的N个副本都写成功，都变得一致
  - 但并不保证能够在短时间内达到一致
  - 最终可能需要很长时间才能达到
- 这种“最终”达到的一致性就是eventual consistency

# Durability vs. Availability

- Durability: 持久性

- 数据不会因为crash/power loss等消失

- Availability: 可用性

- 更进一步，即时出现crash等情况，数据仍然可以被访问

- 在互联网应用中，不仅要durable，而且要available

- 后者直接关系到用户体验

# Dynamo小结

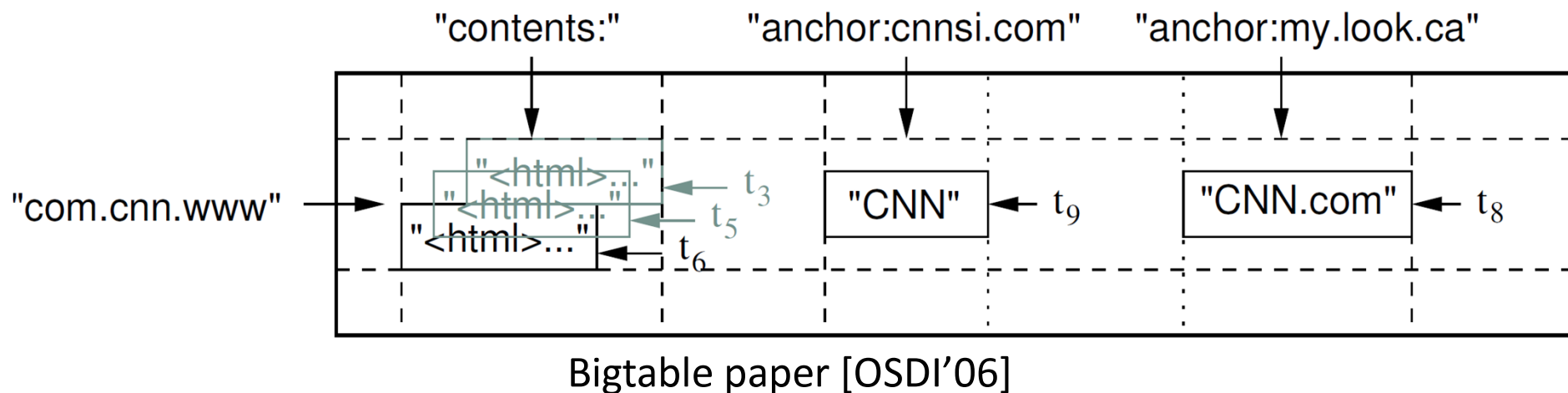
- 最简单的<key,value>模型, get/put操作
- 单节点上存储由外部存储系统实现
- 多节点间的数据分布
  - Consistent hashing
  - Quorum (N, W, R)
  - Eventual consistency

# Key-Value Store: Bigtable / HBase



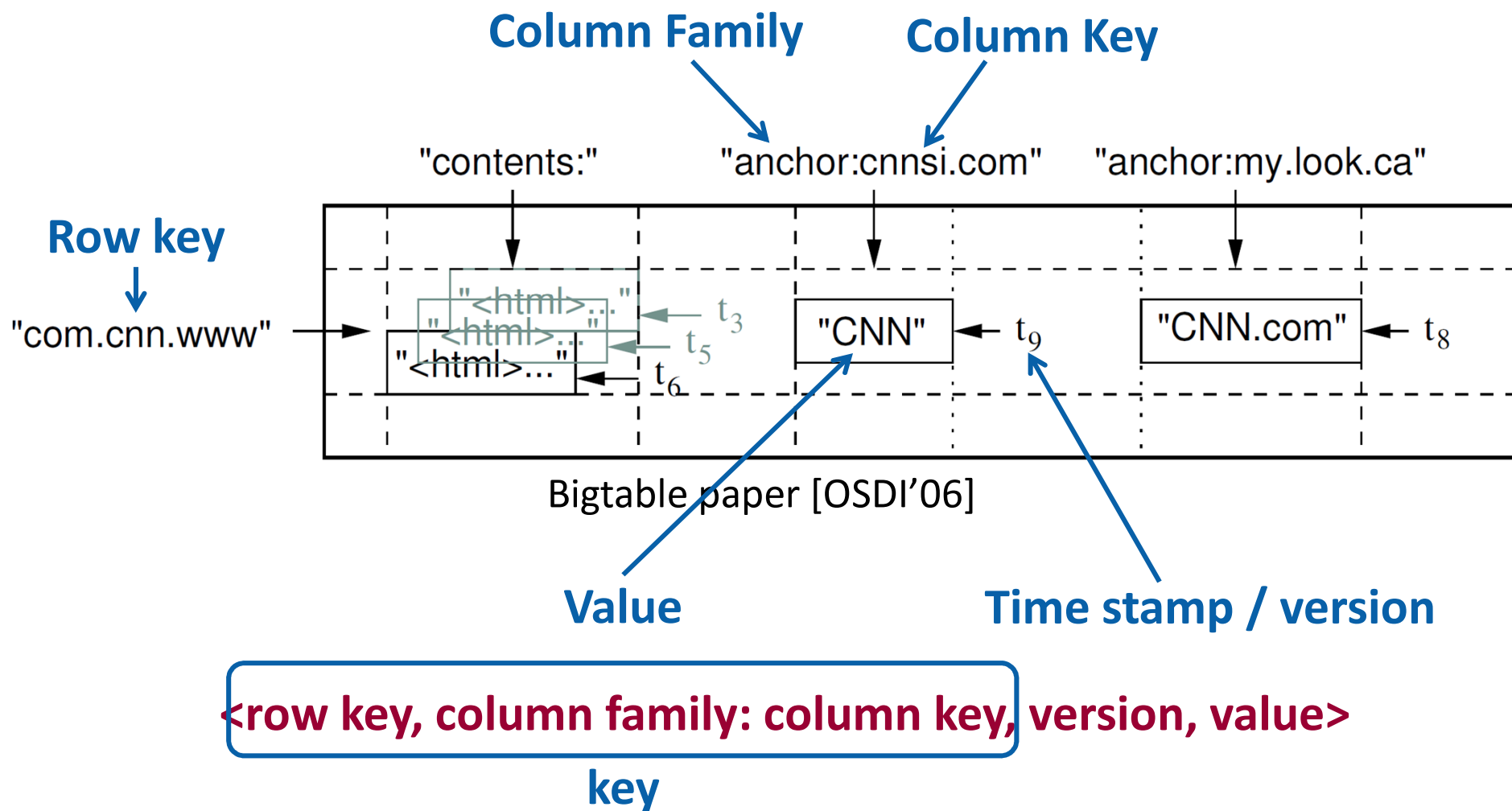
- “Bigtable: A Distributed Storage System for Structured Data.”  
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. (Google).  
**OSDI 2006.**
- 支持Google多种服务
  - “Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth.”
- HBase是Bigtable的Java开源实现，是一个Apache开源项目

# 数据模型：举例Bigtable存储Web page



- Key是domain name的倒置（排序后同一域名会在一起）
- 每个web page记录包含多种类型的信息
  - contents: web page内容
  - anchor: 是指向这个web page的源地址和标签信息
- 每个数据都包括产生时间的信息

# 数据模型：举例Bigtable存储Web page



# 数据模型

  
<row key, column family: column key, version, value>

- Bigtable

- Key包括row key与column两个部分
- 所有row key是按顺序存储的
- 其中column又有column family前缀
  - Column family是需要事先声明的，种类有限（例如~10或~100）
  - 而column key可以有很多
- 具体存储时，每个column family将分开存储（类似列式数据库）

# Key-Value 与 Relational Schema 忽略version部分

- 简单<key, value>可以对应为一个两列的Table

Key	Value
...	...
...	...

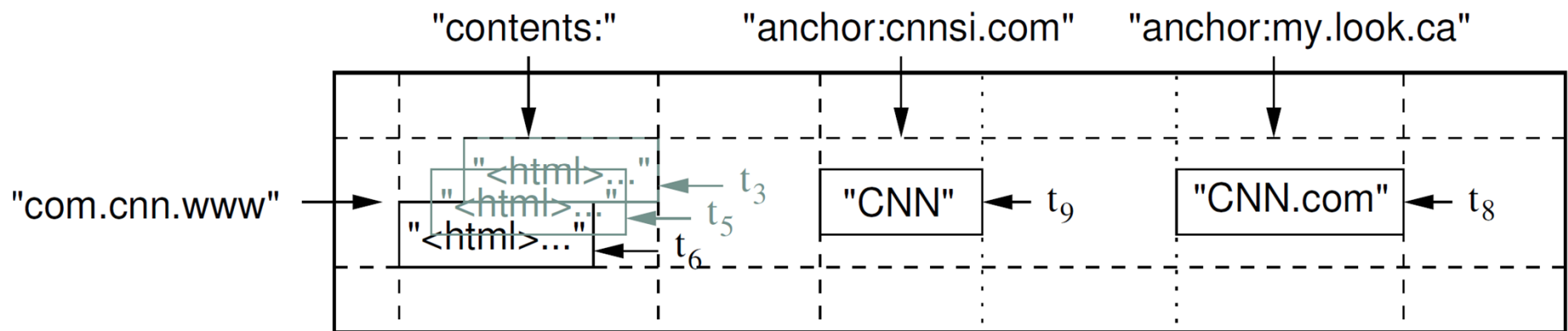
- <row key, column family: column key, value>  
每个column family可以对应为一个3列的Table

Row Key	Column family 1' s column key	Value
...	...	...
...	...	...

Row Key	Column family 2' s column key	Value
...	...	...
...	...	...



# Key-Value 与 Relational Schema 忽略version部分



Row Key	contents	Value
com. cnn. www	“	<html>...
...		...

Row Key	anchor	Value
com. cnn. www	cnnsi. com	CNN
com. cnn. www	my. look. ca	CNN. com

# Bigtable / Hbase 操作

- Get

- ☐ 给定row key, column family, column key
- ☐ 读取value

- Put

- ☐ 给定row key, column family, column key
- ☐ 创建或更新value

- Scan

- ☐ 给定一个范围，读取这个范围内所有row key的value
- ☐ Row key是排序存储的

- Delete

- ☐ 删除一个指定的value

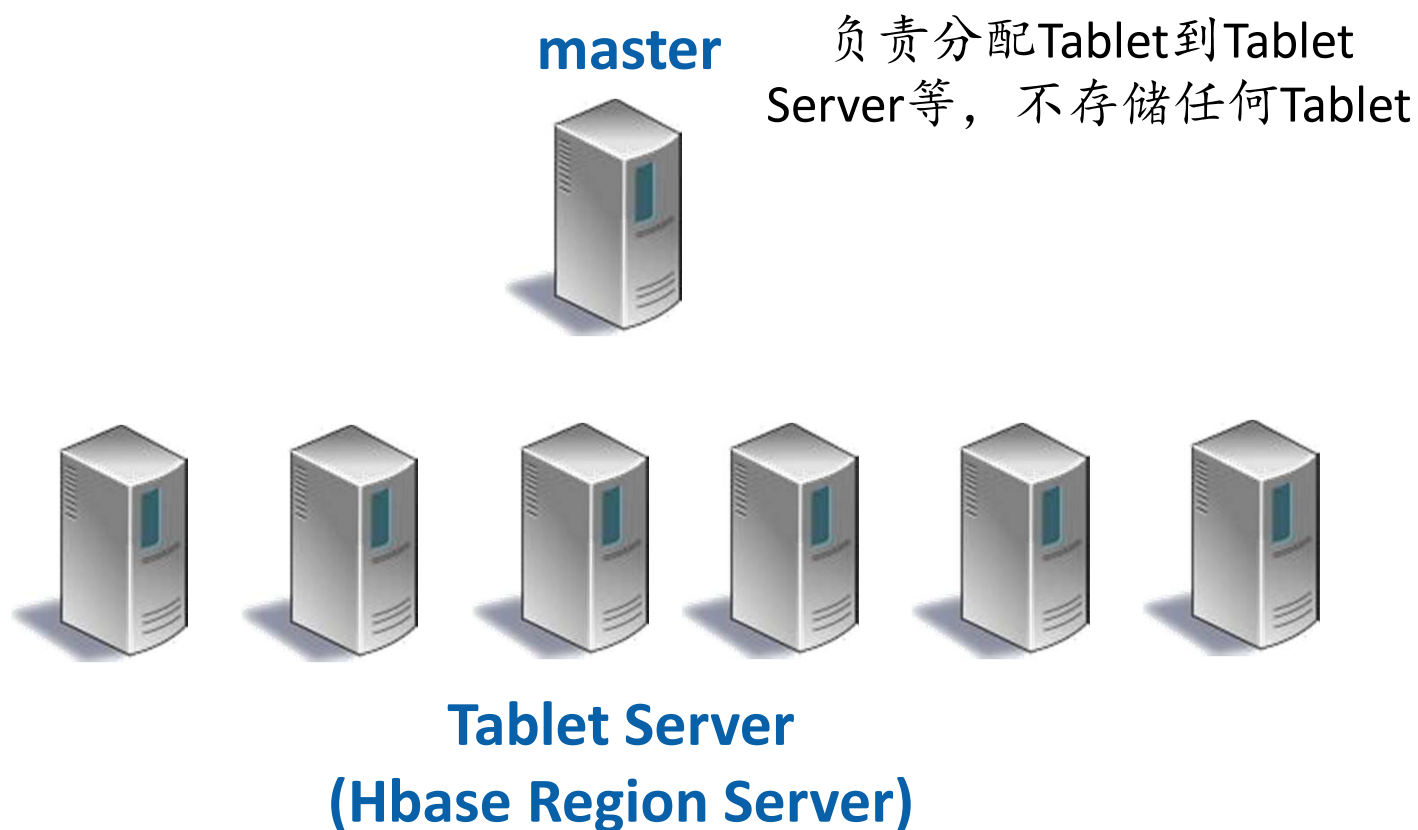
# HBase Scan举例：扫描一组网页

```
➡ HTable htable = ...           // instantiate HTable

➡ Scan scan = new Scan();
➡ scan.addColumn("contents".getBytes(), "".getBytes());
➡ scan.setStartRow(Bytes.toBytes("com.cnn.edition"));
➡ scan.setStopRow(Bytes.toBytes("com.cnn.www"));

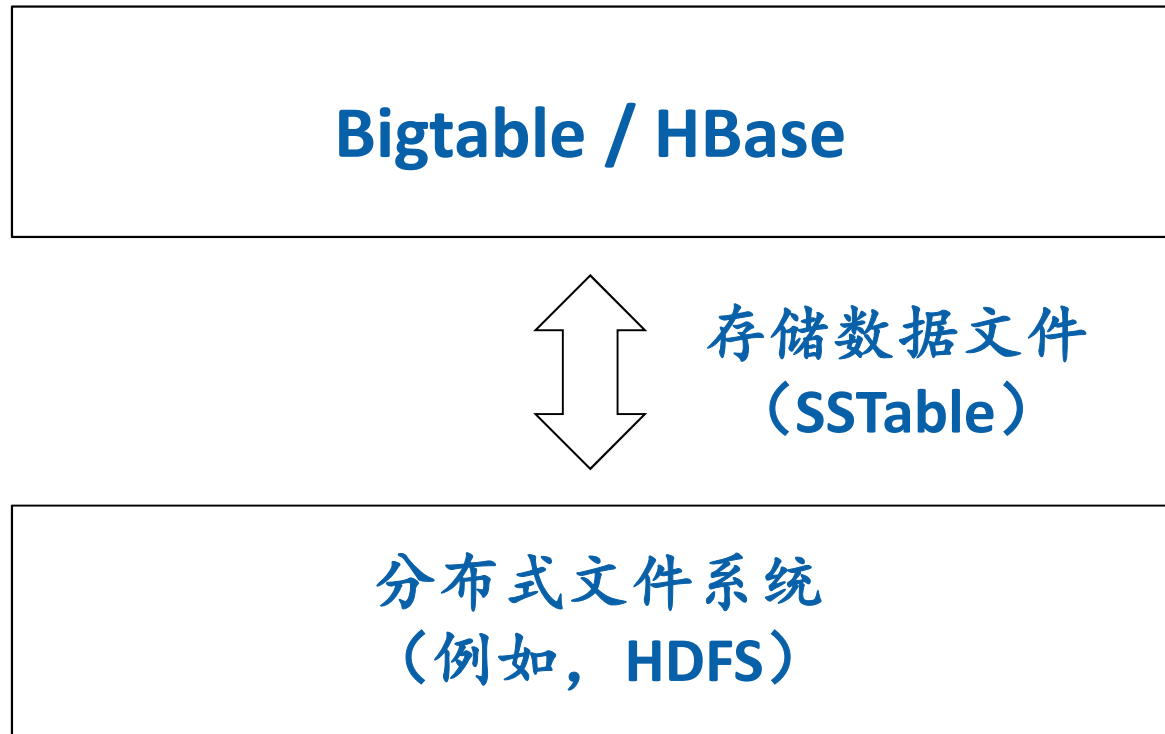
➡ ResultScanner rs = htable.getScanner(scan);
  try {
➡   for (Result r = rs.next(); r != null; r = rs.next()) {
        // process result...
    }
  } finally {
➡   rs.close(); // always close the ResultScanner
  }
```

# Bigtable / HBase 系统结构



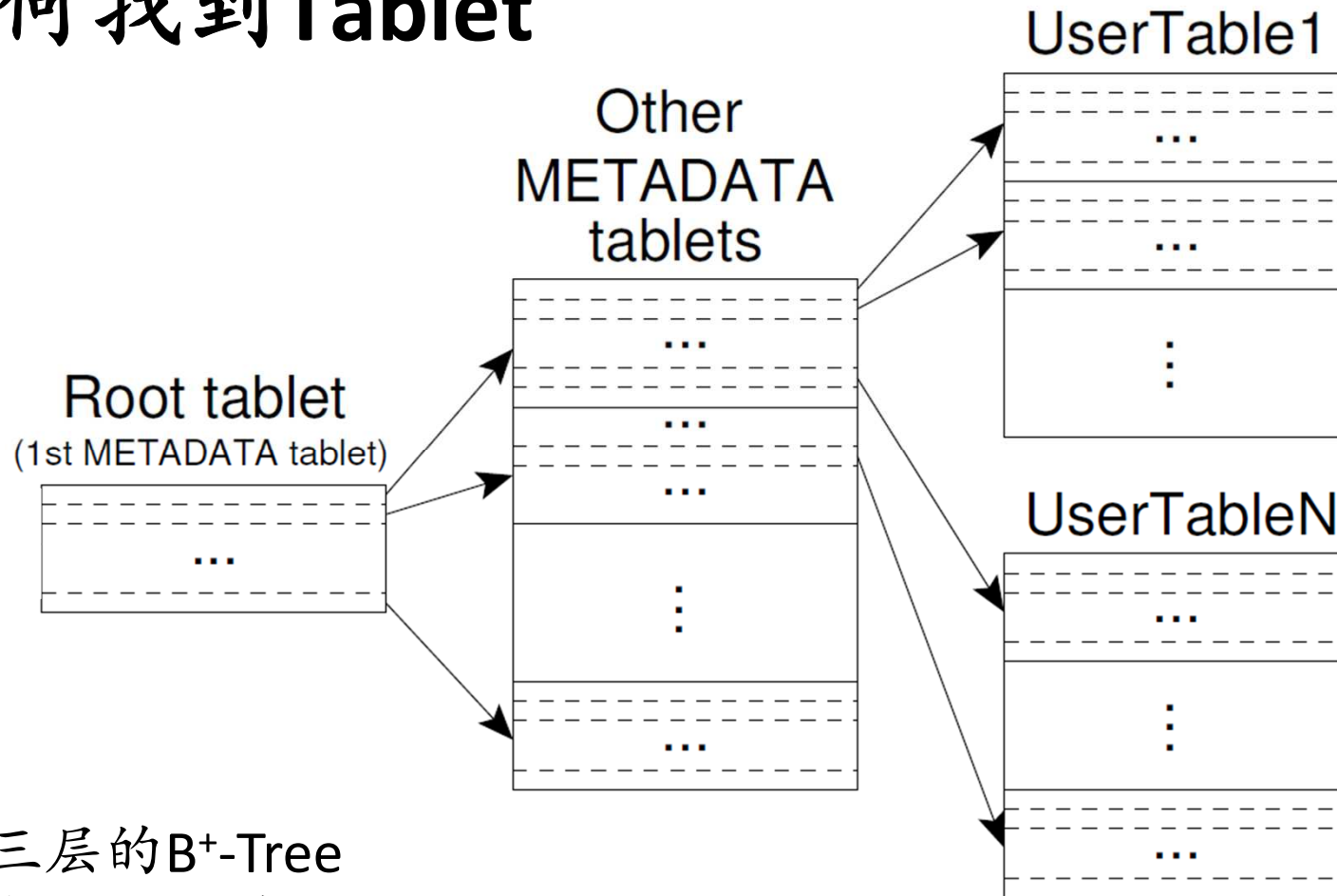
- Tablet是一个分布式Bigtable表的一部分
  - HBase中Tablet被称作Region
  - 我们下面使用Google Bigtable的术语

# Bigtable / HBase 系统结构



- 注意：数据冗余由下层的分布式文件系统提供，所以在 Bigtable 中每个 Tablet 仅存一份

# 如何找到Tablet



- 三层的B<sup>+</sup>-Tree
- 每个叶子节点是一个Tablet
- 内部节点是特殊的MetaData Tablet
- MetaData Tablet 包含Tablet位置信息

Bigtable paper [OSDI'06]

# Tablet Server

多个ColumnFamily

内存

MemTable

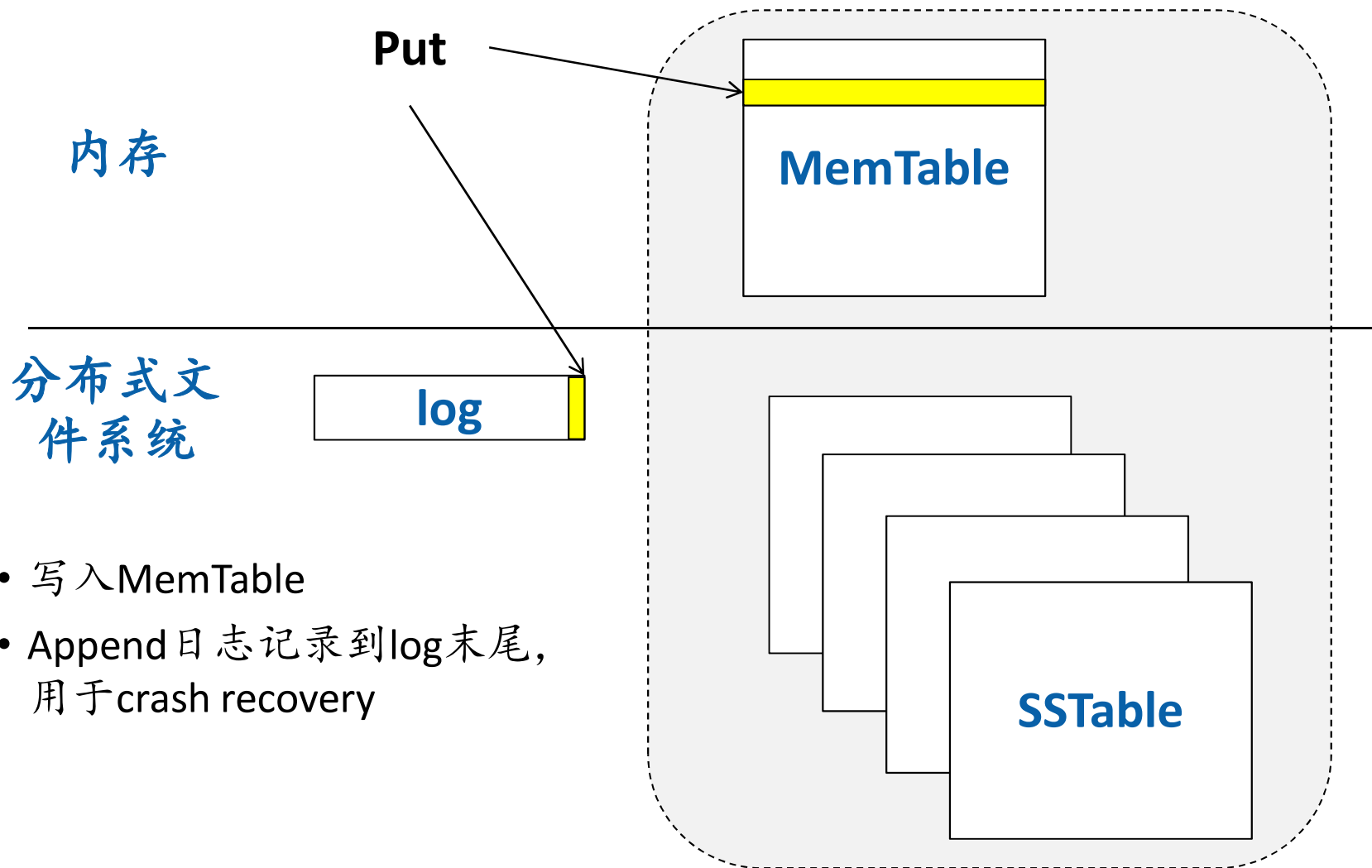
分布式文件  
系统

log

SSTable

- 每个ColumnFamily分别存储
- 内存有一个MemTable
- 文件系统中存储多个SSTable

# Tablet Server: Put操作



- 写入MemTable
- Append日志记录到log末尾, 用于crash recovery



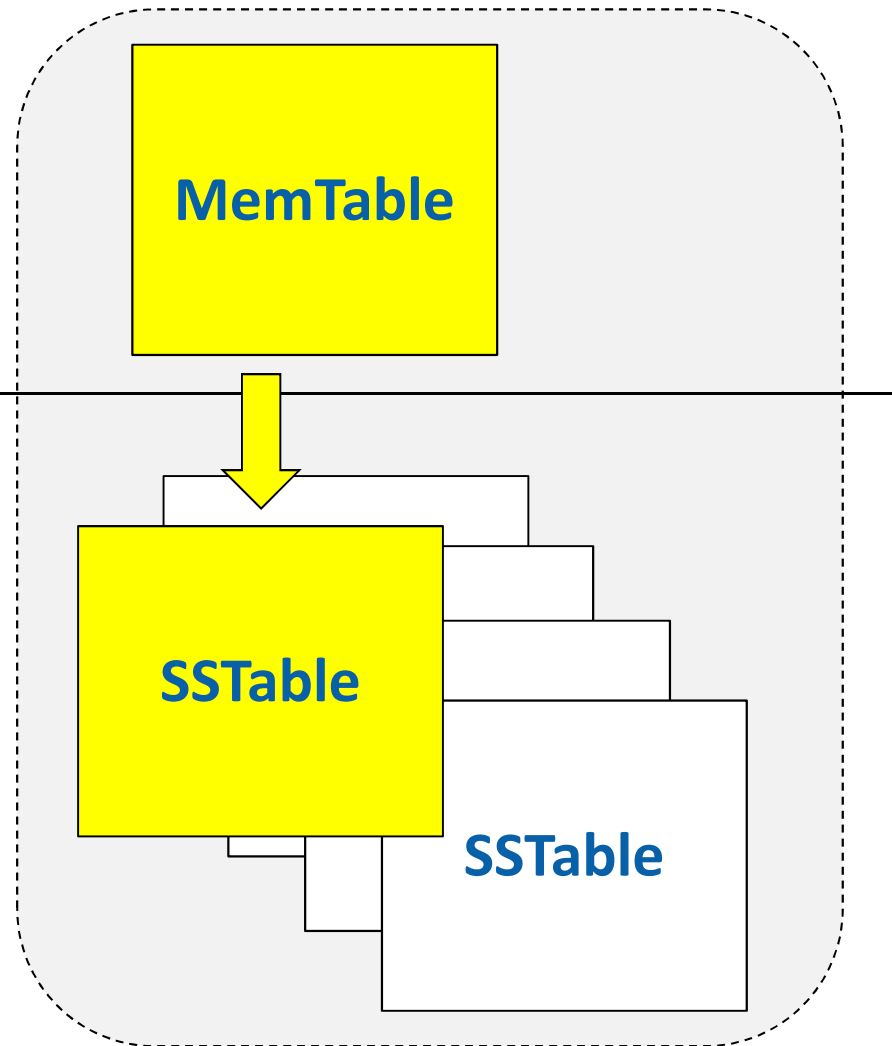
# Tablet Server: 当MemTable满了

内存

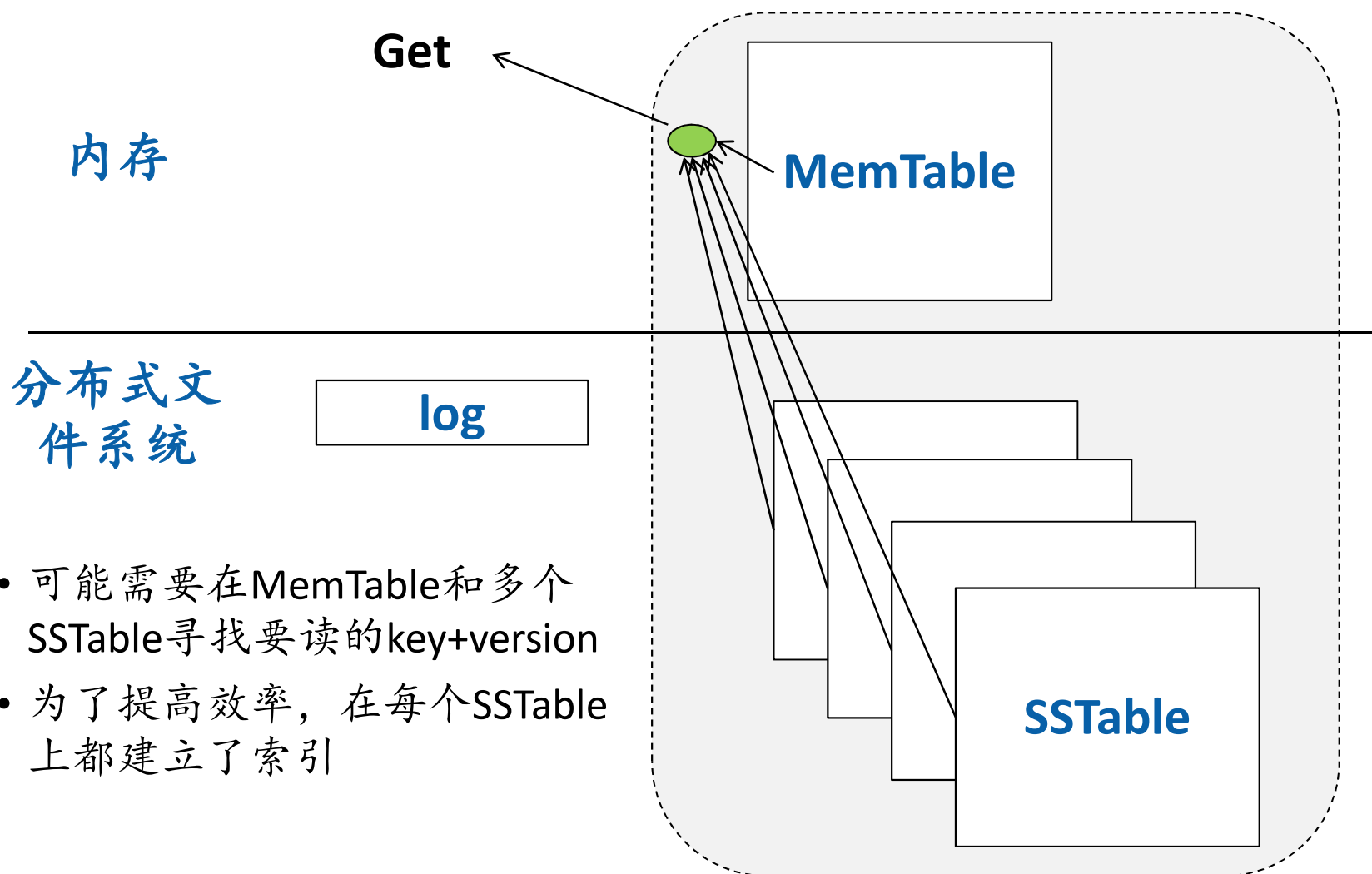
分布式文件系统

log

- 对MemTable排序
- 将MemTable写入文件系统，生成一个新的SSTable
- SSTable是只创建写一次，不会修改，最后删除
  - 适合GFS / HDFS



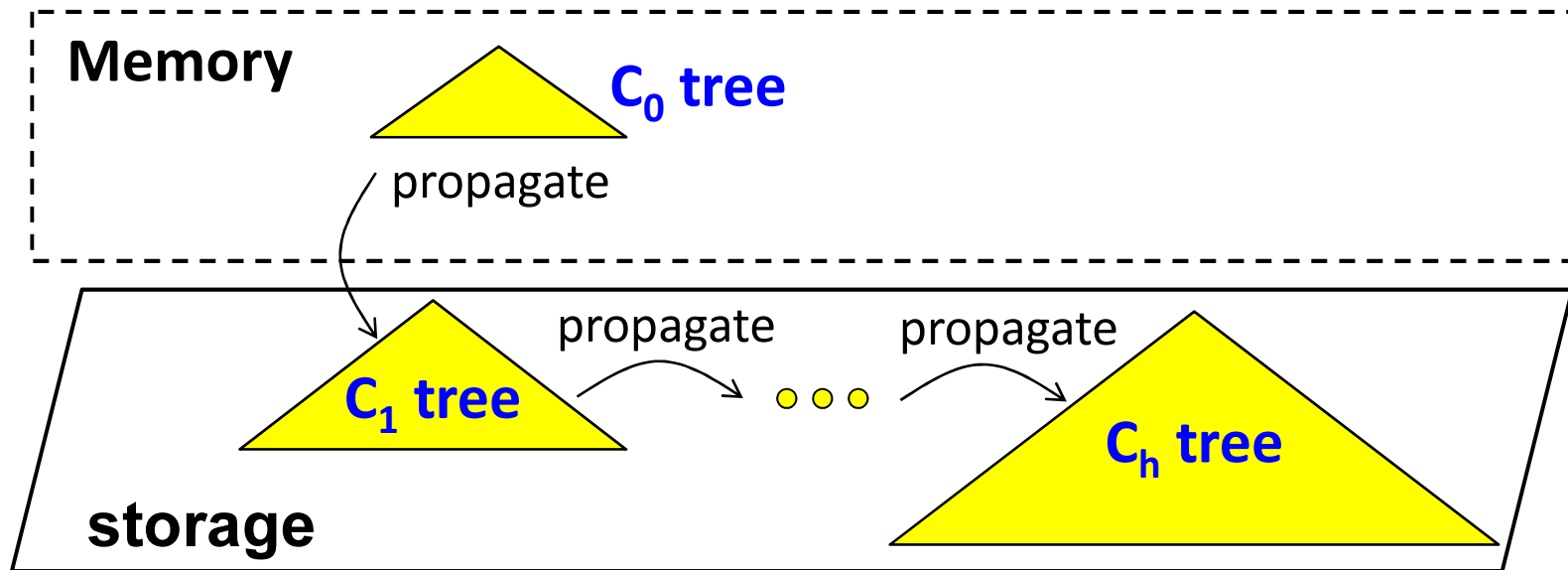
# Tablet Server: Get操作



- 可能需要在MemTable和多个SSTable寻找要读的key+version
- 为了提高效率，在每个SSTable上都建立了索引

# Log Structured Merge Tree

[O'Neil et al. '96]



- LSM-tree vs. B<sup>+</sup>-tree
  - 写优化 vs. 读/写优化
- MemTable/SSTable是LSM-tree的一种变型

# Bigtable / Hbase小结

- Key包含了row key, column key的结构
- 除了Get/Put, 还提供Scan(范围扫描操作)
  - 按照row key有序存储
- 底层存储采用了分布式文件系统
- Master与Tablet Server
- Tablet Server的内部结构: MemTable, SSTable, 和log

# Key-Value Store: Cassandra



- Facebook为了Index Search功能研发了Cassandra
  - Cassandra的研发人员中有Dynamo文章的一位作者
- 之后，Facebook把Cassandra开源，2008年在google code上公布了源代码，2010年Cassandra成为了Apache 开源项目
- Cassandra是基于Java实现的

# Cassandra与Dynamo和Bigtable

- Cassandra可以看作是Dynamo和Bigtable的结合体

	Dynamo	Bigtable	Cassandra
数据模型中的key	key	row key, column key	row key, column key, <b>super column key</b>
数据存储	Berkeley DB, MySQL	内存: MemTable 分布式文件系统: SSTable, Log	内存: MemTable <b>本地文件:</b> SSTable, Log
备份冗余	Consistent hashing	分布式文件系统	Consistent hashing

# Outline

- Key-Value Store
- Distributed Coordination: ZooKeeper
  - 概念
  - 数据模型和API
  - 基本原理
  - 应用举例

# Distributed Coordination

- 分布式系统中，多个节点协调

- ❑ Leadership election: 选举一个代表负责节点
- ❑ Group membership: 哪些节点还活着？发现崩溃等故障
- ❑ Consensus: 对一个决策达成一致
- ❑ ...

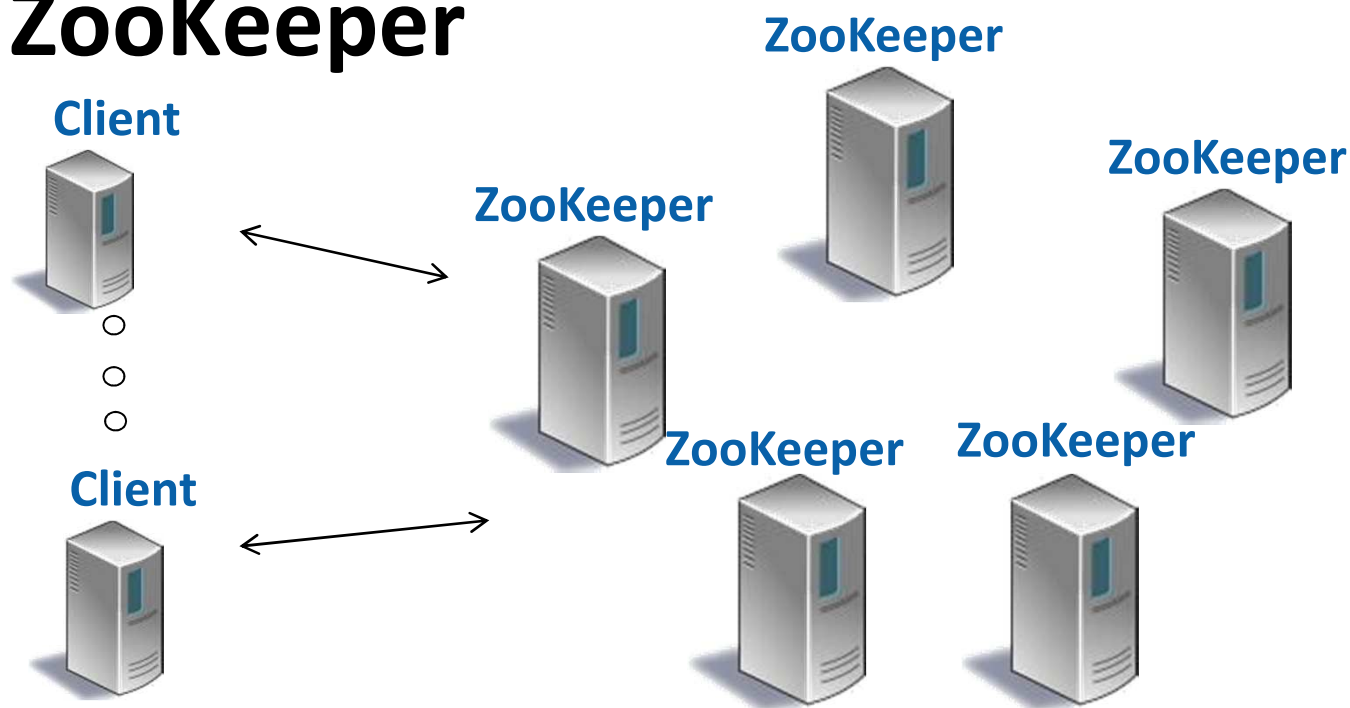
- ZooKeeper

- ❑ Yahoo! 研发的开源分布式协调系统
- ❑ Hadoop/HBase环境的一部分
- ❑ 目前广泛应用于分布式系统对于master节点的容错
  - 使用多台机器运行master节点，一台为主，其余为备份
  - 当主master出现故障，某台备份可以成为主master
- ❑ 例如：HDFS, HBase, Hadoop...

“ZooKeeper: Wait-free Coordination for Internet-scale Systems”. USENIX Annual Technical Conference 2010



# ZooKeeper



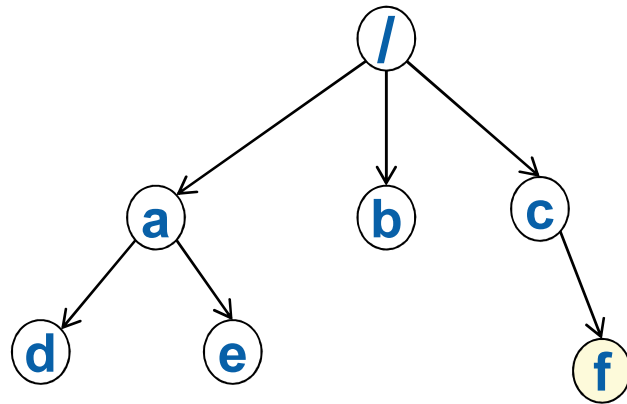
实际上，Client与ZooKeeper可以运行在同一台机器上

- 多个ZooKeeper维护一组共同的数据状态
  - 支持分布式的读和写操作
- $2f+1$ 个ZooKeeper节点可以容忍 $f$ 个节点故障，仍然正确
  - $f=1$ : 3个ZooKeeper节点可以容忍1个节点故障
  - $f=2$ : 5个ZooKeeper节点可以容忍2个节点故障
  - $f=3$ : 7个ZooKeeper节点可以容忍3个节点故障

# ZooKeeper

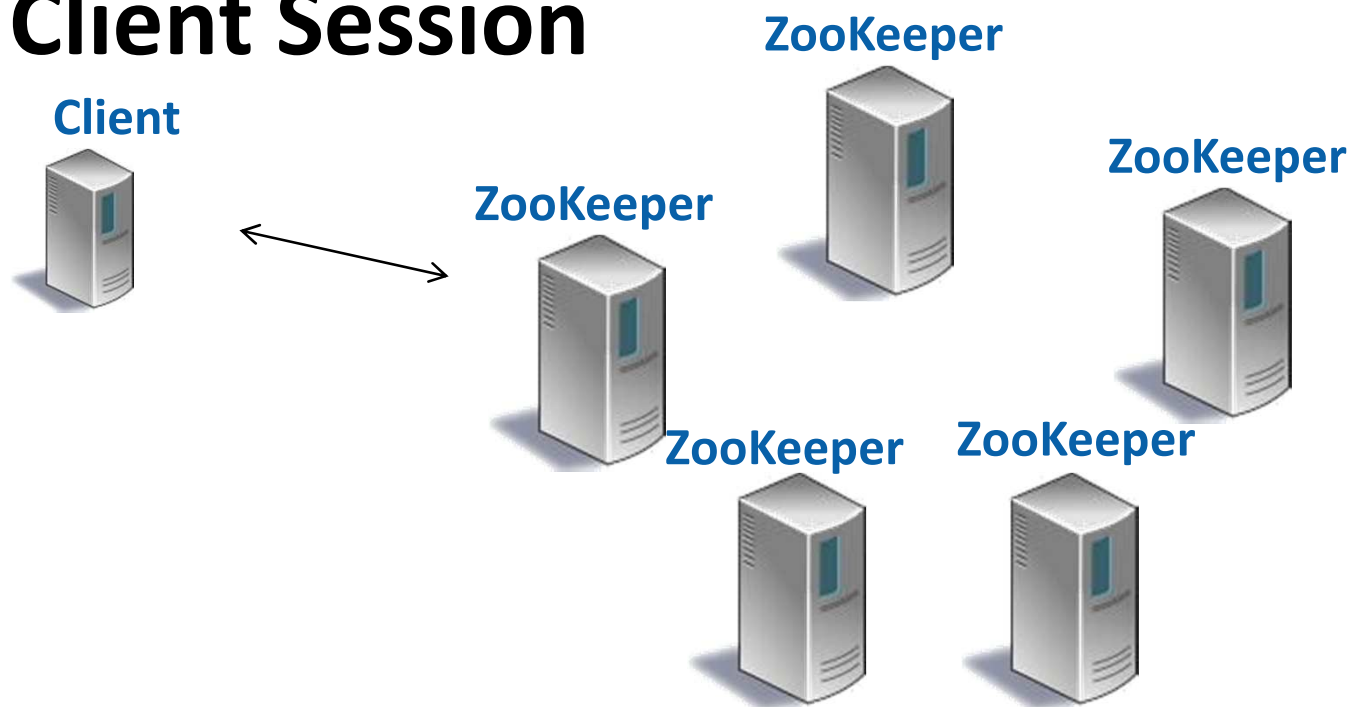
- 数据模型是什么？
- 支持哪些分布式操作？
- 内部是如何实现的？
- 可以用来支持哪些功能？

# ZooKeeper的数据模型：Data Tree



- ZooKeeper维护一组共同的数据状态
  - 表达为一棵树，实际上是一个简化的文件系统
- 树的每个顶点称为Znode，有下列属性
  - Name: 一个Znode可以用一条从根开始的路径唯一确定
    - 类比文件路径，例如：/c/f
  - Data: 可以存储任意数据，但长度不超过1MB
  - Version: 版本号
  - Regular/Ephemeral: 正常的/临时的
    - 对于Ephemeral的Znode，系统将在Client session结束后自动删除

# Client Session



- 一个Client连接到ZooKeeper，就开始一个Session(对话)
- Session结束
  - Client主动关闭
  - 经过一个Timeout时间，ZooKeeper没有收到Client的任何通信

# Client API

- 创建Znode/删除Znode/判断Znode存在
  - create(path, data, flags)
  - delete(path, version)
  - exists(path, watch)
- 读Znode数据/修改Znode数据
  - getData(path, watch)
  - setData(path, data, version)
- 找孩子Znode
  - getChildren(path, watch)
- 等待前面操作完成
  - sync()

# Watch

- Watch机制

- Client的读操作都可以注册watch
- 当ZooKeeper对应的数据发生改变时，就通知Client一次
- ZooKeeper的watch只被调用一次，如果需要继续关注，那么需要再次读+watch

# Client API

- 创建Znode/删除Znode/判断Znode存在

- create(path, data, flags)

- Path
    - Data
    - Flags: regular/ephemeral? 需要Version吗? (不需要为-1)
    - ZooKeeper根据path创建新Znode
    - 如果需要Version, ZooKeeper自动给定一个Version, 保证同一个parent的所有children的Version递增
    - 返回Znode的name

- delete(path, version)

- 只有当version与现在Znode一致, 才执行删除操作

- exists(path, watch)

- 返回True/False
    - 可以设置一个watch, 当Znode被删除/新建时, 收到通知

# Client API

- 读Znode数据/修改Znode数据

- `getData(path, watch)`

- 返回数据和Version
    - 可以设置watch, 当修改时被通知

- `setData(path, data, version)`

- 只有当Version与现在Znode一致, 才执行修改操作

- 找孩子Znode

- `getChildren(path, watch)`

- 返回所有的孩子的name, 可以设置watch

- 等待前面操作完成

- `sync()`: 等待直至ZooKeeper之前所有看到的写操作(create/delete/setData)都完成



# 支持同步和异步方式

- Synchronous: 同步

- Client发一个请求，阻塞等待响应；  
再发下个请求，再阻塞等下个响应
- 当请求个数很多时，同步操作就很慢

- Asynchronous: 异步

- 允许Client发送多个请求，不需要阻塞等待请求完成
- 提供Callback函数，当请求完成时，Callback被调用

- 前面的API都提供同步和异步两种实现

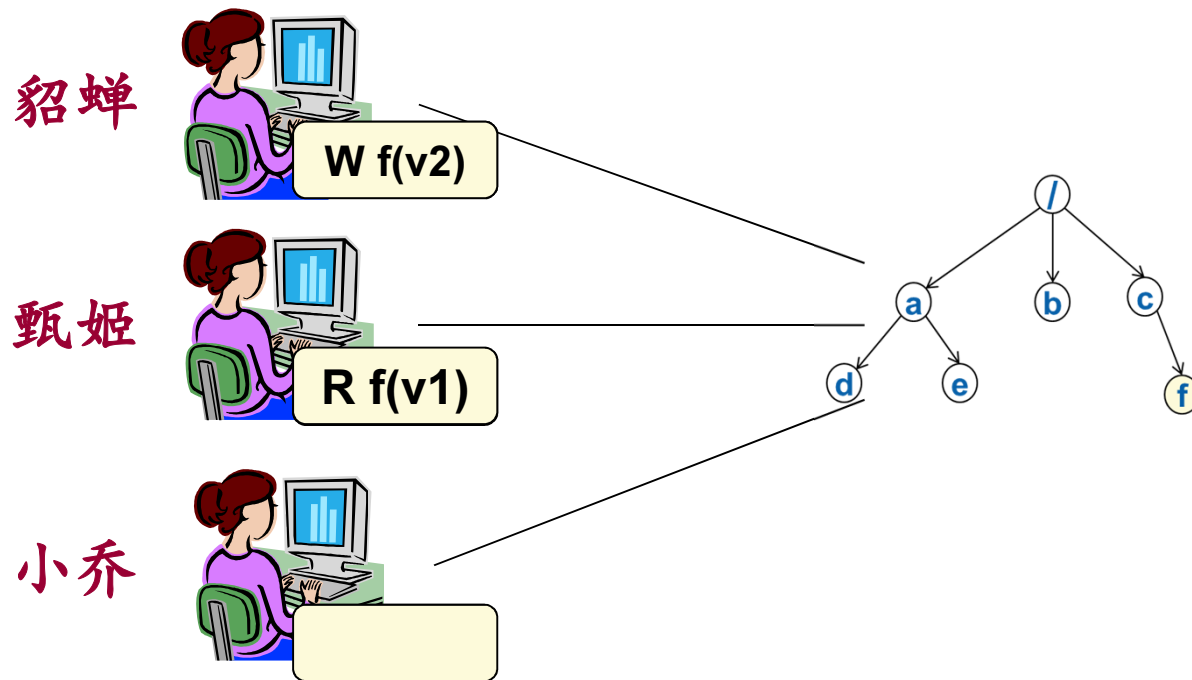
# ZooKeeper保证

- Linearizable writes: 所有的写操作都可以串行化
- FIFO client order: 每个Client的读写操作是按照FIFO的顺序发生的
  - 其它Client都看到这一顺序
  - 例如：一个client先进行了写操作x，然后进行了写操作y，那么所有其它的client如果看到了y，那么一定也看到了x
- 不同Client之间的读写顺序？没有任何保证
  - 一个Client读可能是另一个Client的写前或写后的数据
  - 如果一定要读最新数据，那么调用sync

# ZooKeeper保证

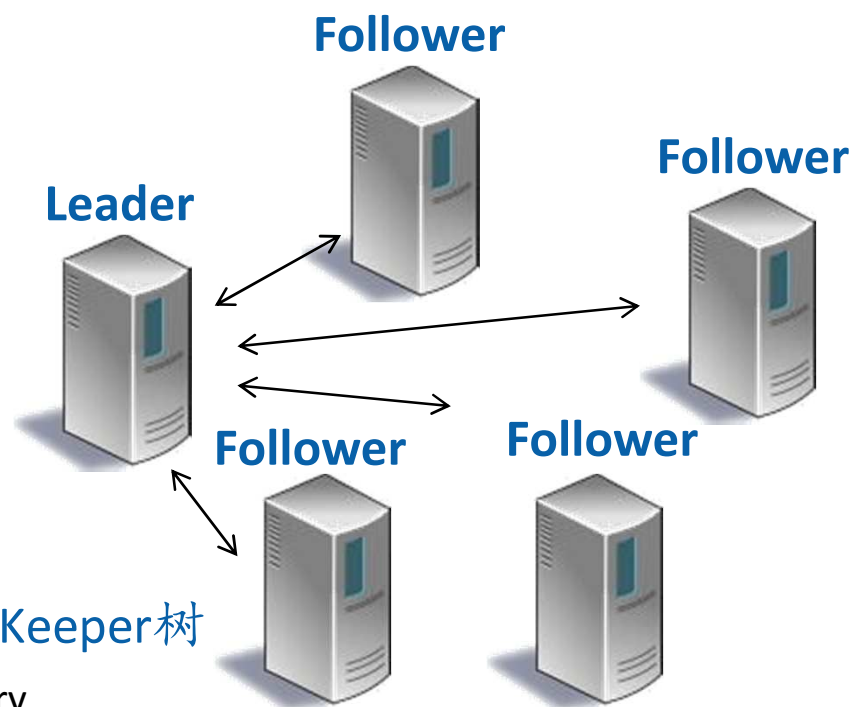
## • 注意

- 这里读操作可能读到旧数据
- sync()然后读，就总会是最新数据



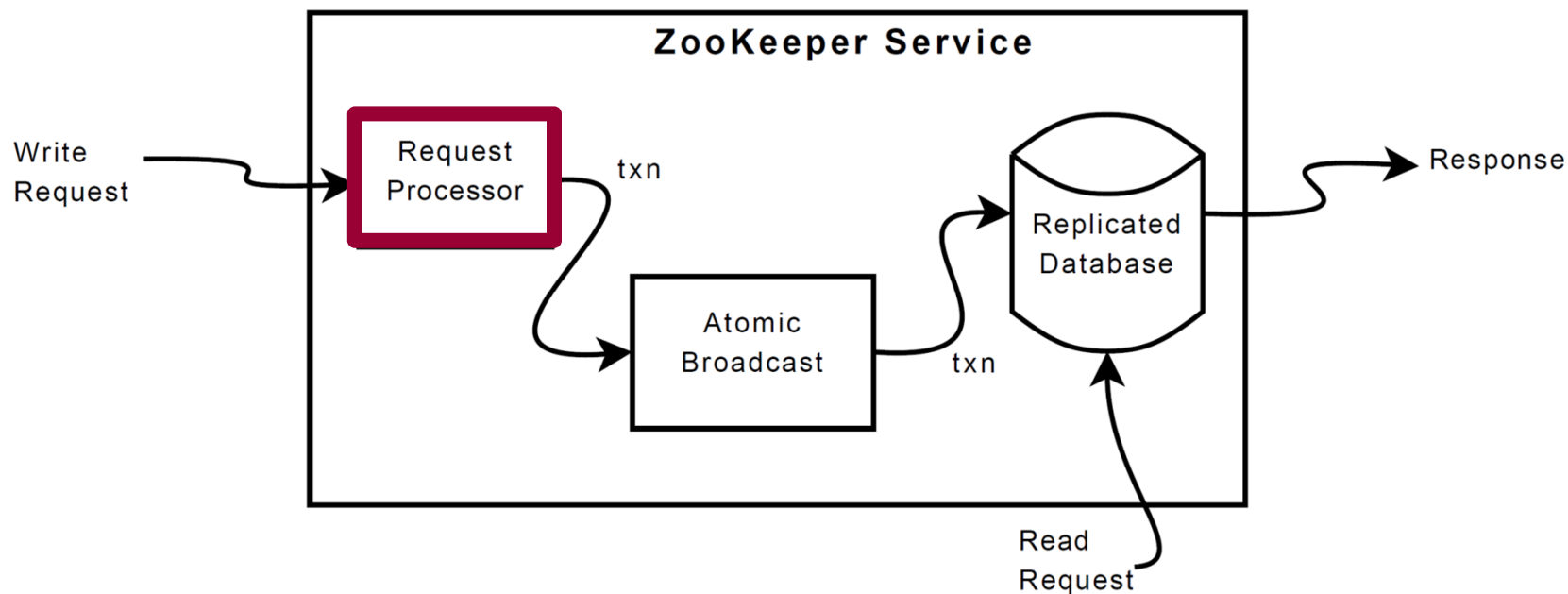
- Client貂蝉写了Znode f(v2)
- Client甄姬读Znode f, 可能看到的是f(v1)
- 只有当甄姬sync后, 再读, 才能保证读到最新的

# ZooKeeper 系统结构



- 所有节点都在内存中维持相同的ZooKeeper树
  - 外存有snapshot+log, 来提供crash recovery
- 每个Client只连接到一台ZooKeeper服务器
  - 所有的读操作都由这台服务器用其本地的状态来回复
- 对于写操作：一个Leader, 其它ZooKeeper节点为Follower
  - Follower把Client的写请求都发给Leader
  - Leader协调所有的Follower一起完成写操作

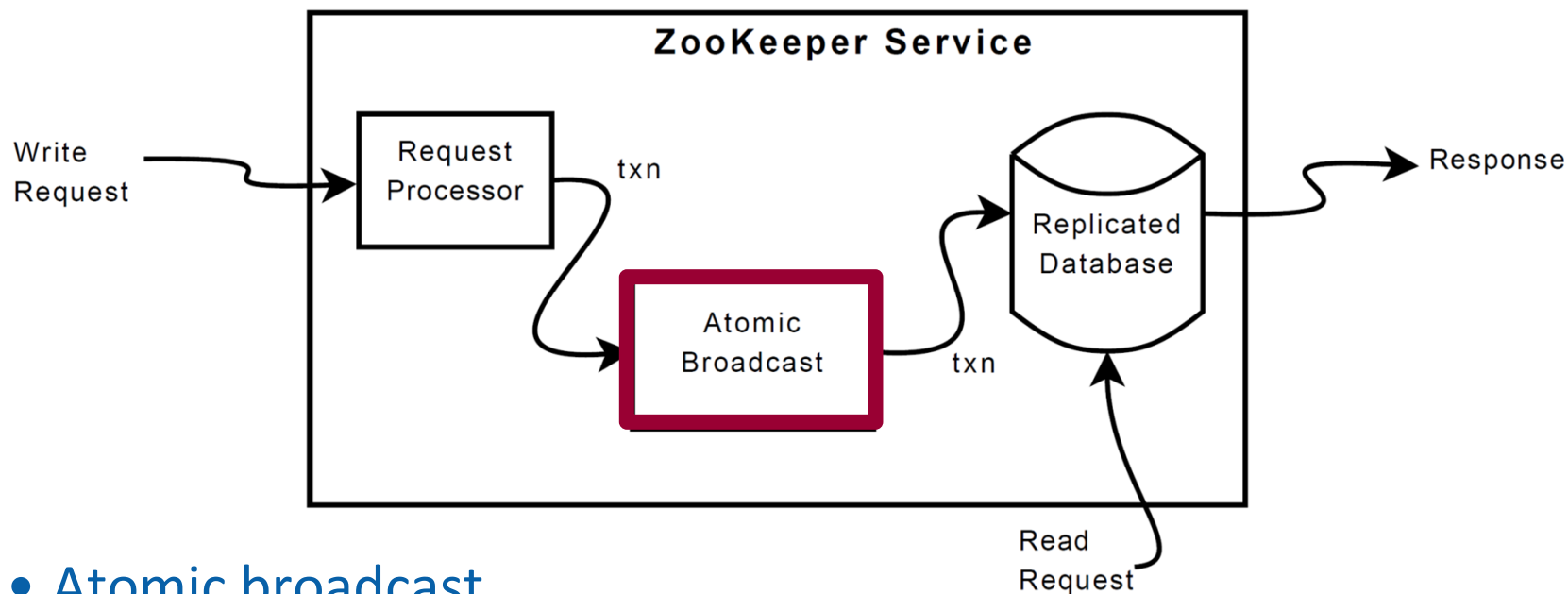
# ZooKeeper节点内部结构



- Request processor

- ❑ 注意：读请求将直接由本地的replicated database回答
- ❑ 对于写请求，将发给Leader统一处理
- ❑ Leader把每个Client写操作，包装成为一个Idempotent Transaction（包括分配新的Version等），这样每个Txn可以执行多次来恢复（概念与NFS相似）
- ❑ Txn有递增的唯一的ID

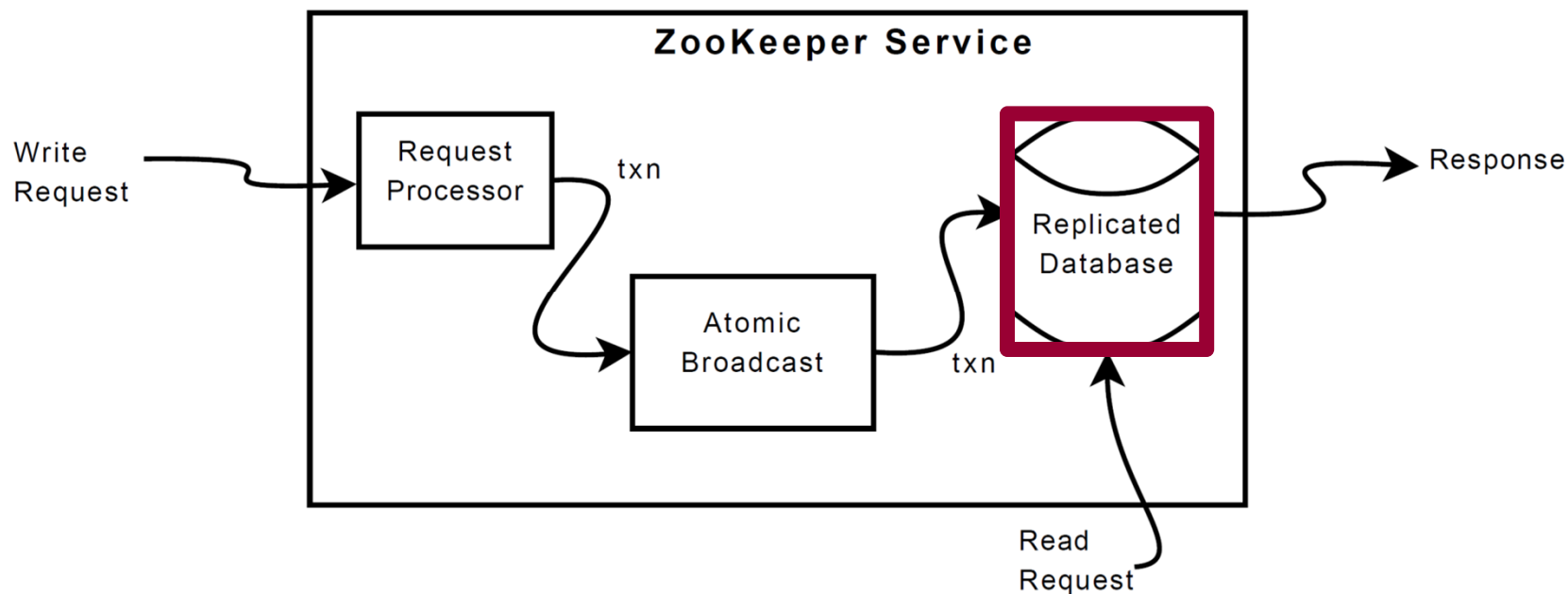
# ZooKeeper节点内部结构



- Atomic broadcast

- Leader带领Follower，保证写操作在全局是串行化的
- 使用的协议是2PC的变形，称为ZAB

# ZooKeeper节点内部结构



- Replicated database: ZooKeeper内存的树

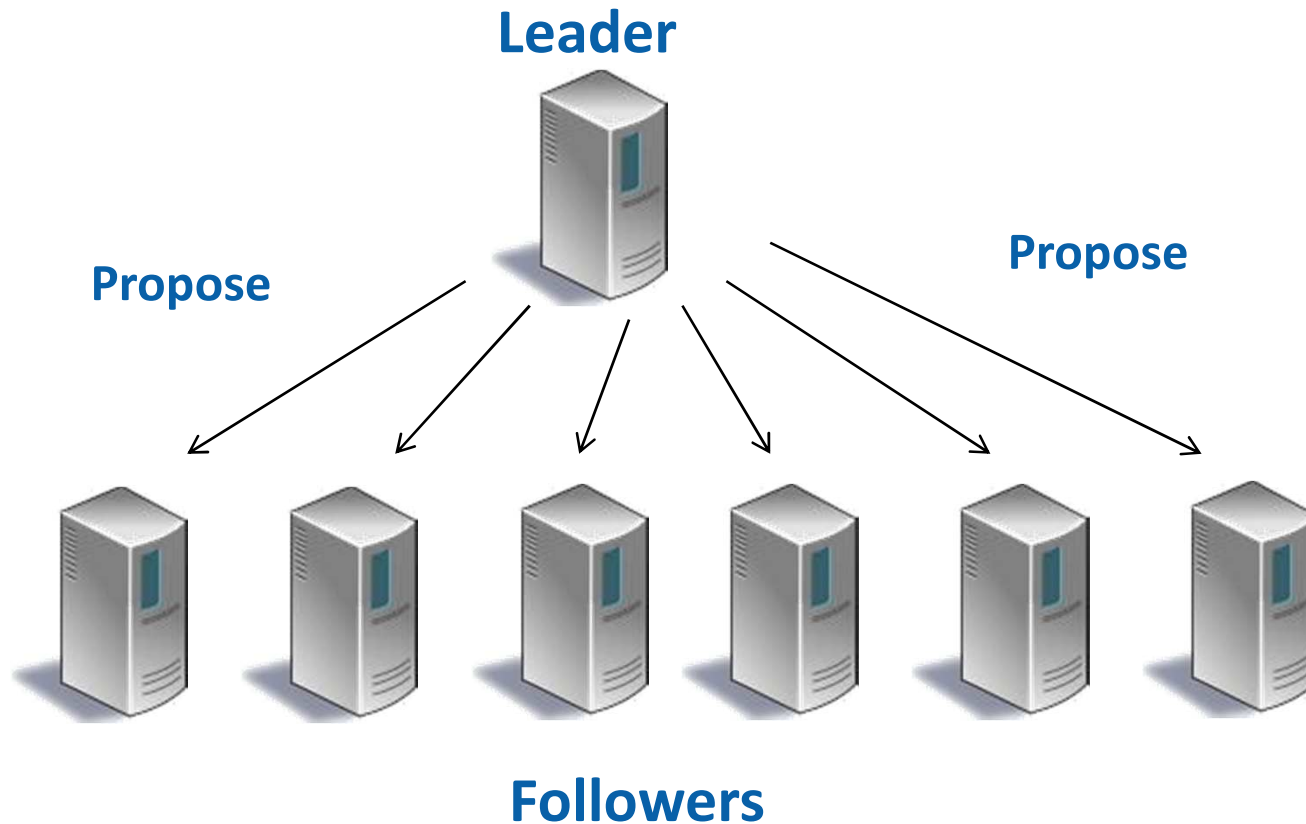
- ❑ 每个Leader/follower: 读操作可以绕开前端, 直接读
- ❑ 在Atomic Broadcast后, 写操作修改本地的Replicated database

# ZAB

- 两个主要工作模式
  - 正常 Broadcast
    - Leader向Follower广播新的写操作
  - 异常 Recovery
    - 竞争新的Leader
    - 新的Leader进行恢复

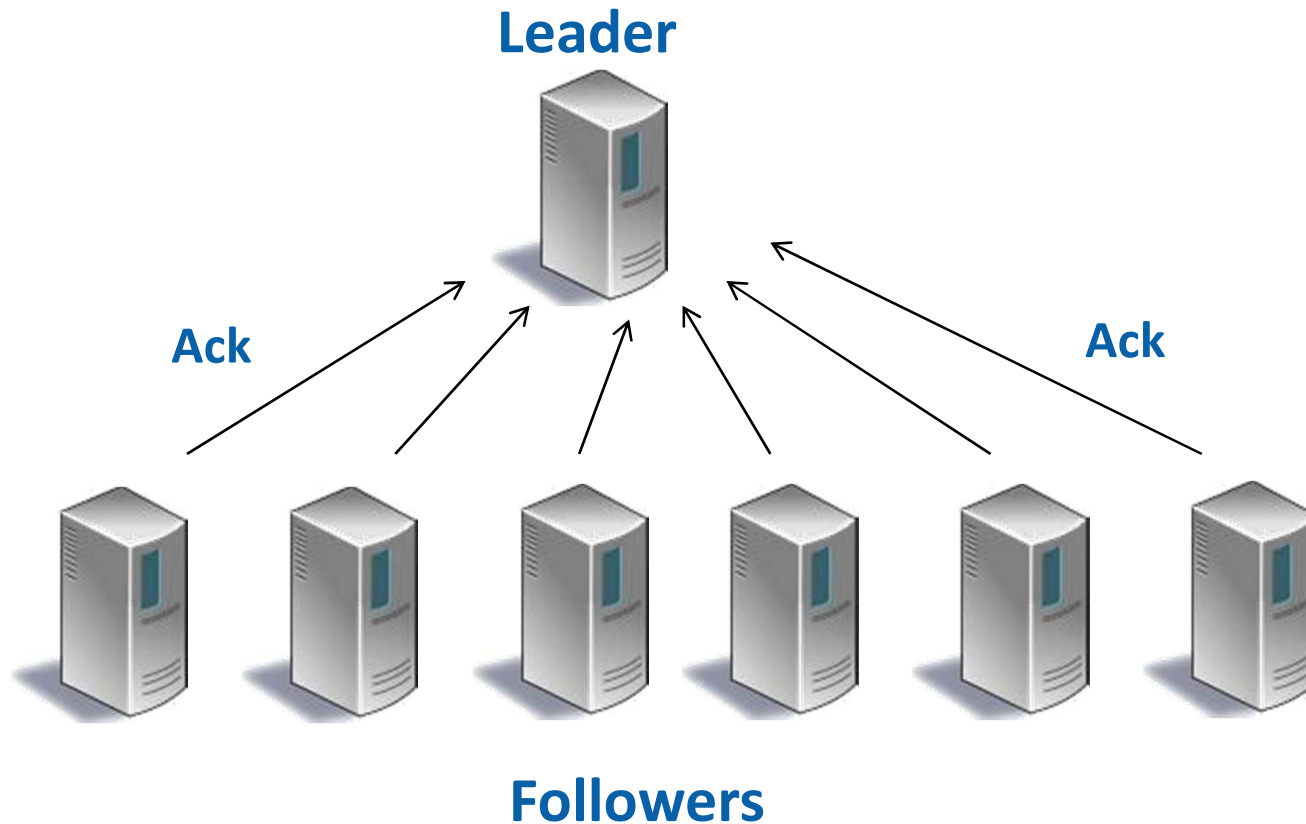


# ZAB Broadcast: phase 1 (propose)



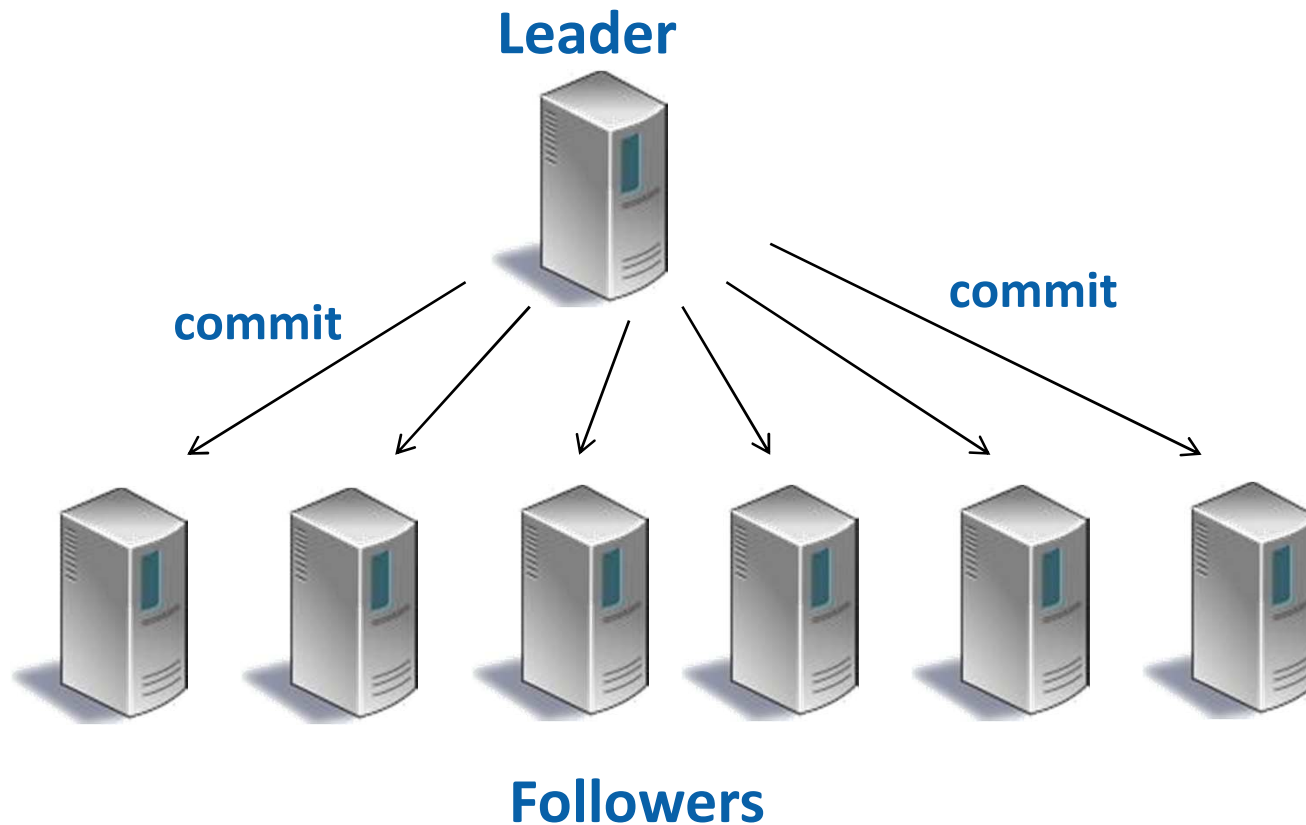
- Leader把一个新的txn写入本地log, 广播Propose这个txn

# ZAB Broadcast : phase 1 (propose)



- 每个Follower收到Propose后，写入本地log，向Leader发回Ack
- 一定可以commit，所以不需要yes/no，只要Ack

# ZAB Broadcast : phase 2 (commit)



- Leader收到  $f$  个Ack后(在所有  $2f+1$  个节点中, 共有  $f$  个followers和自己= $f+1$  节点记住了这个txn), 写Commit到log, 广播Commit, 修改ZooKeeper树
- Follower收到Commit消息, 写Commit到log, 然后修改ZooKeeper树

# ZAB Broadcast

## • 2PC的简化

- 原因：通知新的Transaction发生，所有节点的写操作是一样的
- Propose阶段
  - Leader把一个新的txn写入本地log，广播Propose这个txn
  - 每个Follower收到Propose后，写入本地log，向Leader发回Ack
- Commit阶段
  - Leader收到 f 个Ack后，写Commit到log, 广播Commit，然后修改自己的ZooKeeper树
  - Follower收到Commit消息，写Commit到log，然后修改ZooKeeper树

## • 注意

- 可以异步发送多个Propose，从而可以批量写入log
- Commit阶段不需要Ack
- 如果Leader未收到f个Ack(timeout了)或Follower长时间未收到Leader的消息，那么就发现了故障，需要进入Recovery

# ZAB Recovery

- 竞选Leader

- 每个节点察看自己看到的最大Txn ID
- 选择Leader为看到max(TxnID)为最大的节点
- 可以最大限度地保护Client写操作

- TxnID共64位：高32位代表epoch，低32位为in-epoch id

- 每次选Leader, epoch ++
- 在一个Leader内部，新的txn增计低32位
- 于是，每次Recovery后，一定使用了更高的txn id

- 新的Leader

- 把所有正确执行的Txn都确保正确执行（idempotent，再广播一次）
- 其它已经提交但是还没有执行的Client操作，都丢弃
- Client会重试

# 应用举例（1）

- Configuration Management

- 一个分布式系统可以事先确定一个ZooKeeper路径path
  - 例如：/app1/conf
- 把配置信息存储在这个给定的Znode
- 分布式系统的每台机器，都去getData(path, watch)
  - 获得当前配置信息
  - 当配置信息更新时，会收到watch的通知，于是再getData(path, watch)

# 应用举例（2）

- Group Membership

- 用一个Znode代表这组节点
  - 例如: /app1/group
- 每个成员都在group下面创建一个ephemeral 的孩子
  - 例如: /app1/group/machine1, /app1/group/machine2, ...
- 当某个成员崩溃了, 那么它对应的孩子就被删除
  - 在一定Timeout时间后, 相应的Client session被终止, 这个ephemeral节点被删除
- 从而可以读group的孩子来确定组的成员
  - Group下的节点与仍然工作的成员一一对应

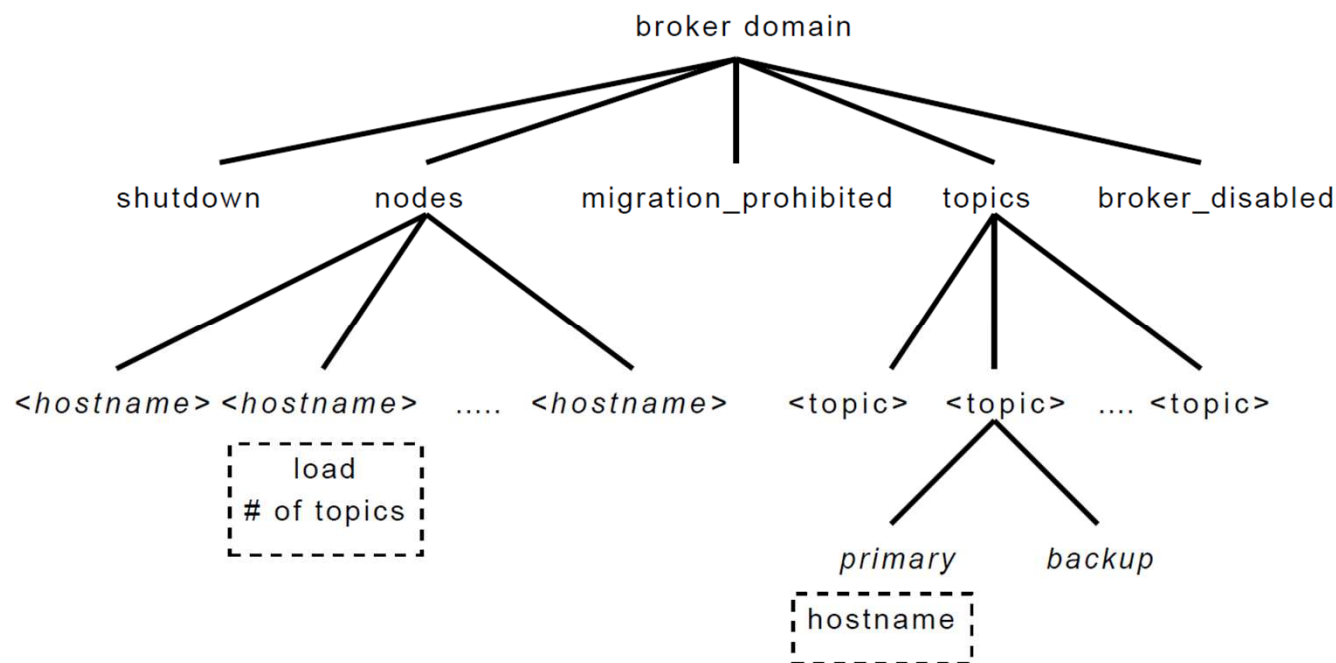
# 应用举例（3）

- Simple Lock

- 可以把lock对应为一个Znode
- 加锁=创建Znode，解锁=删除Znode
- 加锁不成功，可以用watch，当Znode被删除时，可以得到通知



# 应用举例：Yahoo Message Broker



- Publish/subscribe 系统
- 看一下
  - ❑ Group membership: /broker domain/nodes
  - ❑ 配置: topic在哪台机器上: /broker domain/topics
  - ❑ 重要事件: /broker domain/shutdown, /broker domain/migration\_prohibited等

# 小结

- Key-Value Store

- ☐ Dynamo
- ☐ Bigtable / Hbase
- ☐ Cassandra

- Distributed Coordination: ZooKeeper

- ☐ 概念
- ☐ 数据模型和API
- ☐ 基本原理
- ☐ 应用举例