大数据管理系统与大规模数据分析 - June 10'rd, 2018

Storage Concepts

https://github.com/rh01

分布式系统类型

- Client / Server
  - 客户端发送请求，服务器完成操作，发回响应
  - 例如：3-tier web architecture
    - Presentation: web server
    - Business Logic: application server
    - Data: database server
- P2P (Peer-to-peer
  - 分布式系统中每个节点都执行相似的功能
  - 整个系统功能完全是分布式完成的
  - 没有中心控制节点
- Master / workers
  - 有一个/一组节点为主，进行中心控制协调
  - 其它多个节点为 workers，完成具体工作

故障模型 (Failure Model)

- Fail stop
  - 当出现故障时，进程停止/崩溃
- Fail slow
  - 当出现故障时，运行速度变得很慢
- Byzantine failure
  - 包含恶意攻击

CAP 定理

Consistency 多份数据一致性；**A**vailability 可用性；**P**artition tolerance 容忍网络断开；**三者不可兼得**.

Idempotent（幂等性：重复多次结果不变）

- READ 操作是 Idempotent
  - 在没有其它操作前提下，重复多次结果是一样的
  - 因为不改变数据
- WRITE 操作是 Idempotent

  - 在没有其它操作前提下，重复多次结果是一样的，
  - 因为在相同位置写相同的数据

NFS

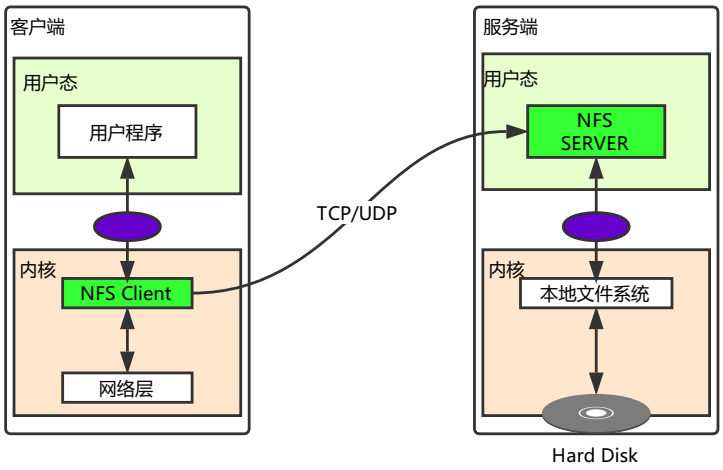

图 1: NFS 的系统架构.

Server Crash Recovery

- NFS Server
  - 只用重启，什么额外操作都不用
  - 因为 Stateless
- NFS Client
  - 如果一个请求没有响应，那么就不断重试
  - 因为 Idempotent

NFSv2.0

- NFSv2 设计目标 2
  - 远程文件操作性能高
- 解决思路：Client cache (在内存中)，缓存读写的数据
  - 尽量利用 cache 数据
  - 避免远程操作
- 存在的问题
  - Cache Consistency
    * Flush-on-close (又称作 close-to-open) consistency

    * 在文件关闭时，必须把缓存的已修改的文件数据，写回 NFS Server
    * 每次在使用缓存的数据前，必须检查是否过时
        · 用 GETATTR 请求去 poll（轮询），获得最新的文件属性
        · 比较文件修改时间
    * 性能问题
        · 大量的 GETATTR（即使文件只被一个 client 缓存）
        · 关闭文件的写回性能

AFS (Andrew File System)

- 设计目标：Scalability
  - 一个服务器支持尽可能多的客户端
  - 解决 NFS polling 状态的问题
    * Invalidation
      · Client 获得一个文件时，在 server 上登记
      · 当 server 发现文件修改时，向已登记的 client 发一个 callback
      · Client 收到 callback，则删除缓存的文件

NFSv2 vs AFS

- AFS 缓存整个文件
-
  - 而 NFS 是以数据页为单位的
  - AFS open: 将把整个文件从 Server 读到 Client
  - 多次操作：就像本地文件一样
  - 单次对一个大文件进行随机读/写：比较慢
- AFS 缓存在本地硬盘中
  - 而 NFS 的缓存是在内存中的
  - 所以 AFS 可以缓存大文件
- AFS
  - 有统一的名字空间，而 NFS 可以 mount 到任何地方
  - 有详细权限管理等

GFS/HDFS

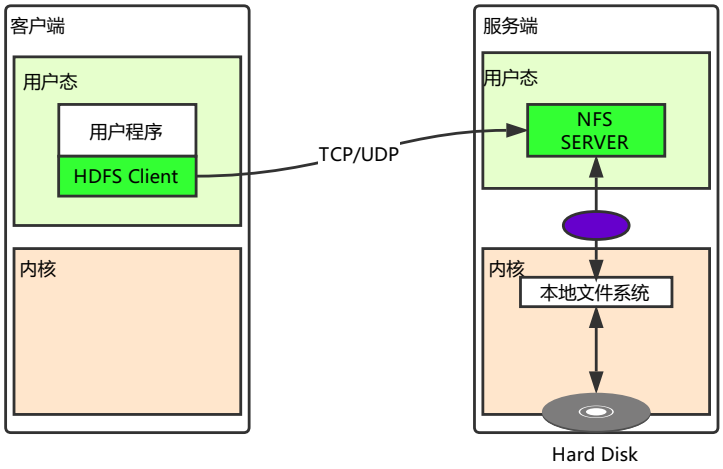

图 2: HDFS 客户端与服务端架构.

GFS 设计目标

- 优化
  - 大块数据的顺序读
  - 并行追加 (append)
- 不支持
  - 文件修改 (overwrite) 操作
  - 所以，consistency 的实现可以大大简化！

HDFS/GFS 系统架构

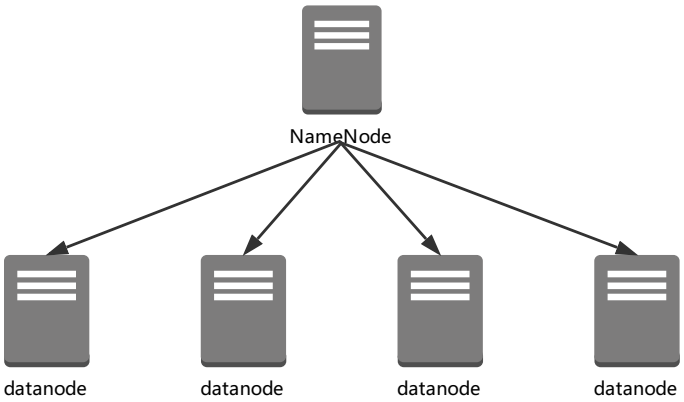

图 3: HDFS/GFS 系统架构.

- Name Node：存储文件的 metadata(元数据)
  - 文件名，长度，分成多少数据块，每个数据块分布在哪些 Data Node 上
  -
- Data Node: 存储数据块
  - 文件切分成定长的数据块（默认为 64MB 大小的数据块）
  - 每个数据块独立地分布存储在 Data Node 上
  - 默认每个数据块存储 3 份，在 3 个不同的 data node 上
- Rack-aware

HDFS/GFS 文件操作

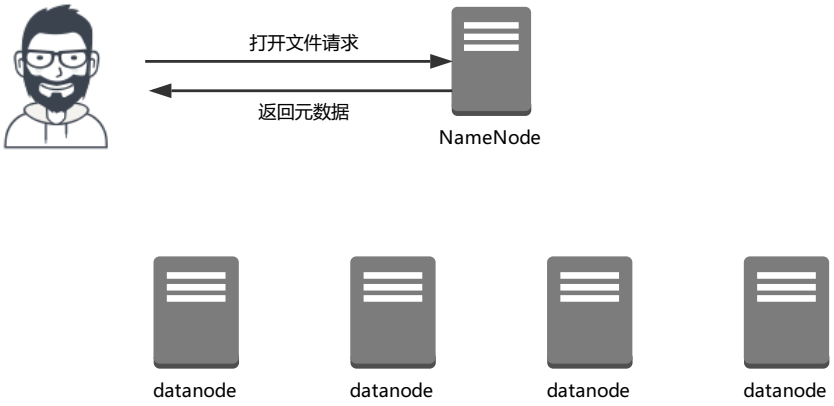

图 4: Open.

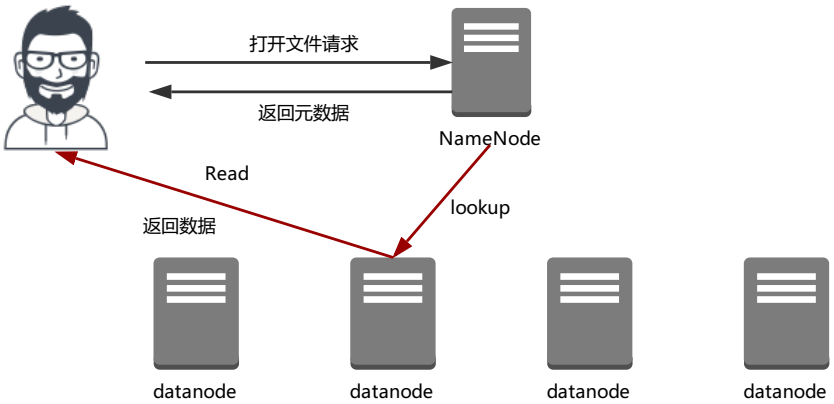

图 5: Read.

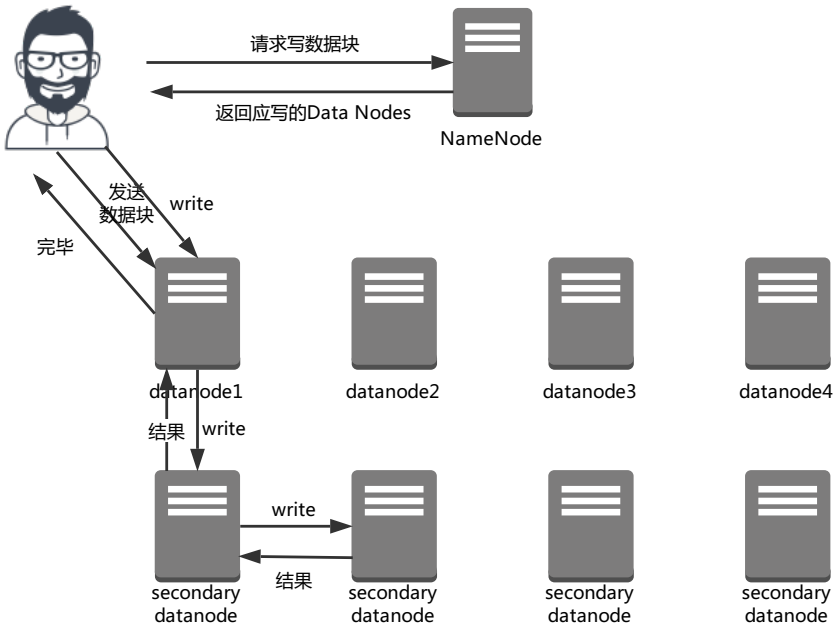

图 6: Write.

并发写的问题

如果一个写操作跨了两个数据块，那么就被分解成为两个独立的写数据块操作，完全没有任何关于两个独立数据块写顺序的规定，实际上是 distributed transaction，有问题！

并发 **Append**

- 文件最后一个数据块在同一个 primary data node
  - 可以在单机上完成 concurrency control
  - **保证并行 append 成功，但是不保证 append 的顺序**

HDFS/GFS 小结

- 分布式文件系统
- 很好的顺序读性能
  - 为大块数据的顺序读优化
- 不支持并行的写操作：不需要 distributed transaction
- 支持并行的 append

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

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) $\Longrightarrow$ (value, version)
- ACID?
  - 没有 Transaction 概念
  - 仅支持单个 <key,value> 操作的一致性

Dynamo 系统结构

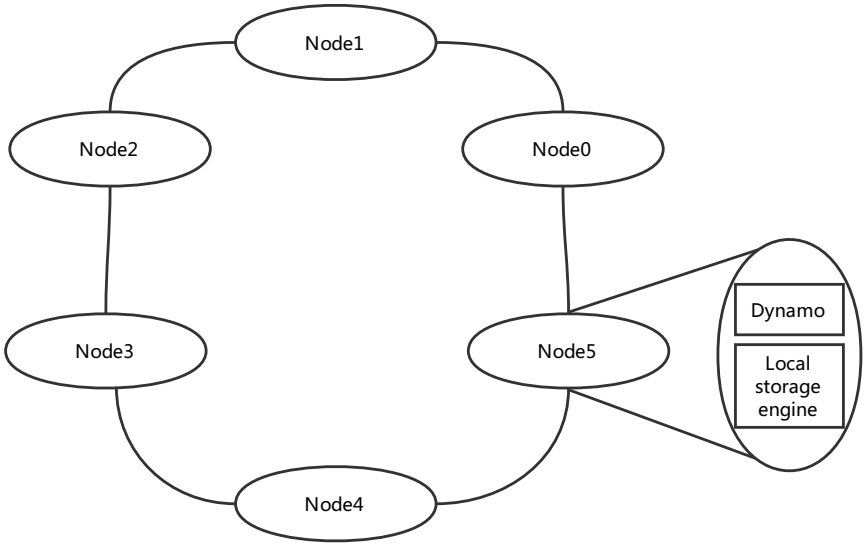

图 8: Dynamo 系统结构.

Consistent Hashing 一致哈希算法 (p2p 的关键技术)

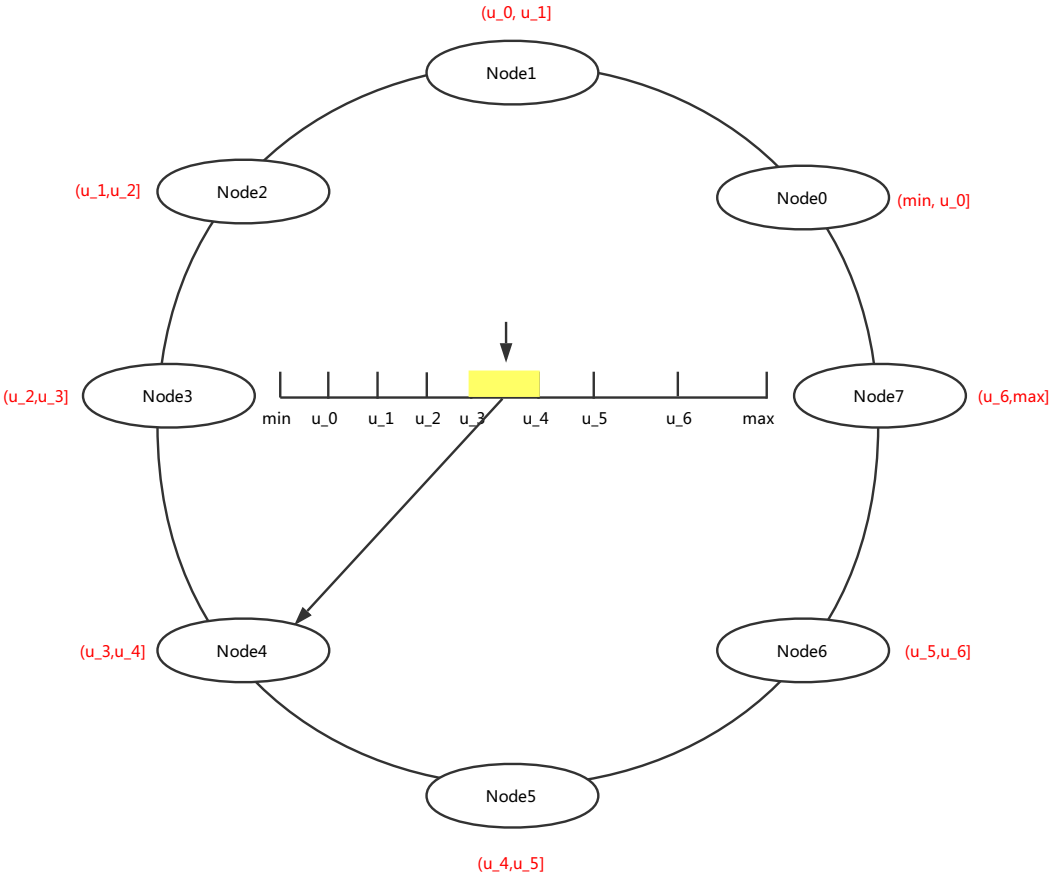

图 9: Consistent Hashing.

Consistent Hashing：备份

Consistent Hashing：增加一个节点

Consistent Hashing：减少一个节点

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)

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 最终一致原则

- 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

数据模型

Ex. <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

简单 <key, value> 可以对应为一个两列的 Table

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

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

Bigtable / HBase 系统结构

Ex. master $\Longrightarrow$ Tablet Server (Hbase Region Server)

- Tablet 是一个分布式 Bigtable 表的一部分

如何找到 Tablet

- 三层的 B + -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
- Tablet Server 的内部结构：MemTable, SSTable, 和 log

Key-Value Store: Cassandra

Cassandra 与 Dynamo 和 Bigtable 比较

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

表 1: Cassandra 与 Dynamo 和 Bigtable 比较

Distributed Coordination: ZooKeeper

- 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, Hadoo.....

ZooKeeper

- 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 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**.

Zookeeper: Client Session

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

Zookeeper: Client API

Watch 机制

- 注册：Client(读操作) 可注册 watch
- ZooKeeper 通知
  - 当对应的数据发生改变时，通知 Client
- 通知之后，Watch 就被删除了
  - 如果需要继续关注，那么需要再次注册 watch

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

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 都看到写的顺序
  - 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 系统结构

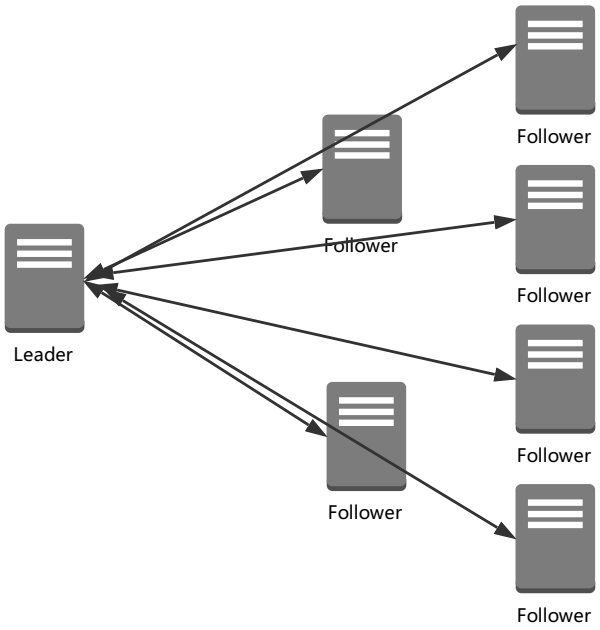

图 10：ZooKeeper 系统结构.

- 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(故障恢复)
- 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
- 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

写请求的处理

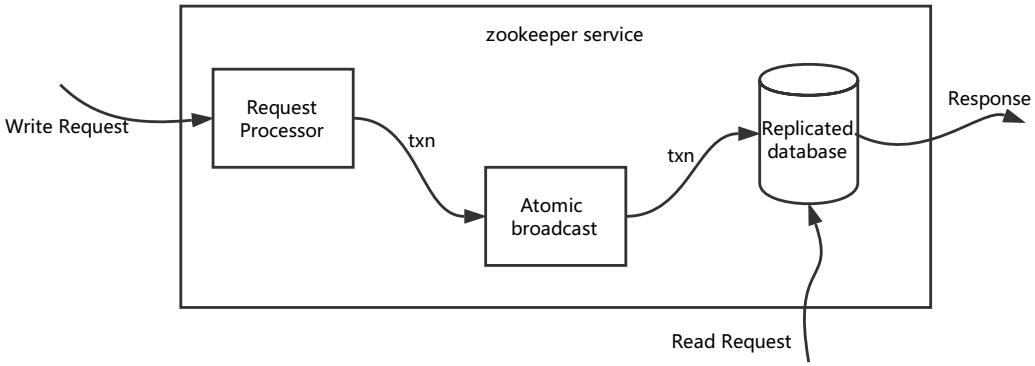

图 11：ZooKeeper Service.

- Follower 上的 Request processor
  - 对于写请求，将发给 Leader 统一处理
- Leader 上的 Request processor
  - 把写请求包装成为一个 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
- Atomic broadcast：广播写请求
  - 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**.
- 所有节点的 Replicated database: ZooKeeper 内存的树
  - 在 Atomic Broadcast 后，写操作修改本地的 Replicated database
- 每个节点都可以处理读请求
- 读请求将直接由节点本地的 replicated database 回答
  - 写的全局顺序有 Leader 决定
  - 但读是分布的，如果写还没有广播完成，读可能看到旧数据

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

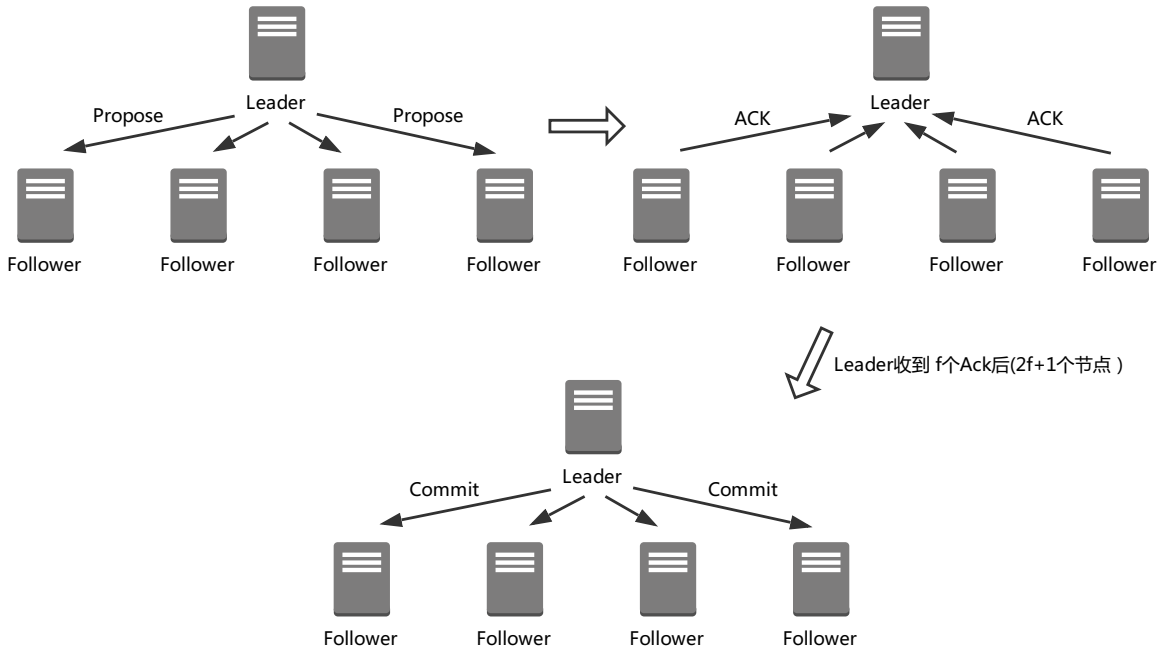

图 12：ZAB.

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