分布式存储系统设计

—— 莫惊池里寻常满，一井清泉是上源。

Pprun
nextry@qq.com
https://github.com/pizirun

修订历史

版本	时间	描述
0.01	2011 年 02 月 02 日	开始基于初步的实现编写本文档
0.02	2011 年 02 月 05 日	加入 数据模型 部分
0.03	2011 年 02 月 06 日	加入 外部接口 部分
0.04	2011 年 02 月 07 日	添加 动态平衡 部分
0.05	2011 年 03 月 05 日	添加 通讯通道 部分
0.06	2011 年 03 月 19 日	加入 一致性 部分
0.07	2011 年 04 月 05 日	更新大量术语，在翻译 Amazon's Dynamo 中文 的同时
0.08	2011 年 04 月 19 日	加入 系统元数据，Membership, Vector Clock 部分
0.09	2011 年 04 月 20 日	加入 安全 部分
0.10	2011 年 04 月 21 日	添加 MapReduce 部分和 参考 部分
0.11	2011 年 04 月 23 日	添加 负载均衡 部分
0.12	2011 年 04 月 24 日	Release to Public
0.13	2011 年 04 月 27 日	更新 MapReduce 处理流程
0.14	2018 年 08 月 01 日	Redirect to Github.com

采用类似于 Amazon Dynamo 的去中心化 (Decentralized) 设计：

- 类似于 BigTable 的基于 ColumnFamily 数据模型
- 持久化层：MemTable, SSTable, CommitLog
- 一跳式分布式哈希表 (1 Hop-DHT)
- 数据中心，机架拓扑敏感的动态副本策略(Replica Strategy)
- W + R (>, =, <) N 可配置的一致性策略
- 客户端 Timestamps 并发冲突解决方案
- 写操作 Hinted Hand-Off, MerkleTree, 反熵 (Anti-Entropy)
- 读操作修复 (Read Repair)
- Gossip
- φ Accrual Failure Detector
- REST Client API
- MapReduce / Compute Server(Sun Grid Compute Server)
- 智能客户端负载均衡

- OAuth 云安全

内容目录

一. 数据模型

- 实体
- 数据存储
- 系统元数据

1. 实体

- Column
- ColumnFamily
- KeySpace
- 排序(Sorting)
- 限制
- 其他

主流的 NoSQL 存储系统实现分为两大派：
- Key-Value Based Storage (源自于 Amazon's Dynamo)
- ColumnFamily Based Storage (源自于 Google's BigTable)

由于 Key-Value Based 的系统简单到可以描述为：

```
v = storage.get(k);
storage.put(k, v);
storage.delete(k);
```

因此，根本没有必要长篇讨论数据模型，因为 Key-Value 本身是作为通用数据存储系统而诞生的，如: Amazone SimpleDB.

所以，这里讨论的是 ColumnFamily Based 系统，支持 Column/ColumnFamily 是为了应对复杂的业务需求：复杂的查询与特定的排序。

1.1 Column

列 (Column) 为 ColumnFamily Based 的系统中最小的原子单元 (atomic unit), 由 name, value and timestamp 组成

Column	
Field	**Type**
Name	byte[]
Value	byte[]

Version	long(timestamp)

实例

一个用来存储 User 的 username 的列

username		
Name	**Value**	**Timestamp**
“username”	“pprun”	*1356048000000*

** 1356048000000 为 时间自 epoch 至 ”2012-12-21 00:00:00 UTC”毫秒数*

1.2 ColumnFamily

大体上，列族是为了将相关的列群组在一起，以便一次性读出相关数据，而不是单个分散的数据。

Key 决定从哪里找，怎么找到对应的列族（见 排序）

ColumnFamily							
key	**Column List** (1 .. * Columns)						
	Name	**Value**	**Timestamp**	**Name**	**Value**	**Timestamp**	**...**

实例

一个列族用于存放用户的信息：username, age, gender, ... (因为空间有限，只列出前两列)

User							
key	**username**			**age**			**...**
	Name	**Value**	**Timestamp**	**Name**	**Value**	**Timestamp**	
1	“username”	“alpha”	1322105600386	“age”	18	1356105600381	
2	“username”	“beta”	1332105600386	“age”	22	1356105600383	
3	“username”	“gamma”	1352105600386	“age”	25	1356105600370	
4	“username”	“pprun”	1356134400000	“age”	20	1356105600387	

** 列的最终组织方式是根据 ColumnFamily 指定的排序类型确定（见排序）*

1.3 KeySpace

KeySpace 是顶级的数据单元，可以想象成 RDBMS 的 Schema，一个分布式存储中可以有一个或同时有多个 KeySpace。因此，同一个存储系统就可以被多个不同的应用共享。

KeySpace 包含 1..* 个 ColumnFamily.

通常一个应用定义一个 KeySpace。其中包含多个 ColumnFamily，虽然，也许这些 ColumnFamily 之间并无任何关系，但是系统这种组织是为了将全局的配置／策略施之于 KeySpace。

你可能会想象以下的概念图形化表示：

KeySpace								
ColumnFamily 1					**ColumnFamily 2**			
key	**Column List**				**Column List**			
	Name	**Value**	**Timestamp**	…	**Name**	**Value**	**Timestamp**	…

但是，事实上，实现往往是取 KeySpace 的一小部分区域作为物理单元（也就是存在磁盘空间上）来分布式存放的：

- Tablet (Google BigTable)
- Regions (Apache HBase)
- ColumnFamily (Apache Cassandra)

另外，由于同一个 Key 可以绑定多个 ColumnFamily ，而每个 ColumnFamily 的业务模型并没有强关联性，所以数据最终呈现出”稀疏”（sparse）的特性

KeySpace								
ColumnFamily 1					**ColumnFamily 2**			
key	**Column List**				**Column List**			
	Name	**Value**	**Timestamp**	…	**Name**	**Value**	**Timestamp**	…
1	V	V	V					
2					V	V	V	
3					V	V	V	
4								

* V 表示填充

1.4 有序(Sorting)

1.4.1 主序或行序（Major Ordering/Row Ordering)

ColumnFamily 中某行数据存放在分布式的集群的什么位置（节点：哪台机器）是由 Key 的分配策略/ Partitioner 决定的。甚至在基于动态哈希（DHT）的分配策略中，根本就不存在什么顺序，那个 key 只是用来定位该行最终存储在什么地方。(本节暂不讨论这一主题)。

1.4.2 列序

列按照 ColumnFamily 指定的排序类型（comparator)自动有序地存储是 ColoumFamily base 的存储系统的最大特点。在定义 ColumnFamily 时，指定 排序类型，然后，列族中的所有列以"列名"(column name) 按照这一类型进行排序后。想象在实现时通过 SortedMap 来存储列。

例如，如果 列族 User 指定 key 的类型为 字典类型排序的话，那么，上述表格组织方式是不对的，因为 **a**ge 应该排在 **u**sername 之前，即 a 在字典中，排在 u 之前（如果按照从左到右为顺序的话），所以最终结果应该是这样：

User							
key	**age**			**username**			**...**
	Name	**Value**	**Timestamp**	**Name**	**Value**	**Timestamp**	
1	"age"	18	1356105600381	"username"	"alpha"	1322105600386	
2	"age"	22	1356105600323	"username"	"beta"	1332105600134	
3	"age"	25	1356105600370	"username"	"gamma"	1352105613336	
4	"age"	20	1356105301700	"username"	"pprun"	1356048000000	
...							

1.5 限制

只要是资源有限，任何数据模型都需要对数据长度加以限制，以满足空间和时间的平衡。

- 列名称长度
- 列值 byte[] 的最大长度，这个可能会跟最终的文件系统息息相关

1.6 其他

1.6.1 Row (行) 在哪?

在 Column-Based 的存储系统里存储本身是以 ColumnFamily 为单元组织的，因此在 KeySpace 中，columnFamily 由（key + columnFamily (1..*)) 组成一行:

- Keyspace
- Columnfamily

- row (indexed, key)
- column (sorted name, value, timestamp)

之所以在文中提到过 Row 的概念，是由于在访问数据时，只能过 [keySpace, key, columnFamily, column] 层次结构访问的：

```
insert(keyspace, key, columnFamily, rowMutation)

get(keyspace, key, columnFamily, column)

delete(keyspace, key, columnFamily, column)
```

1.6.2 要不要嵌套 ColumnFamily?

象 Cassandra 的 SuperColumn/SuperColumnFamily 那样?

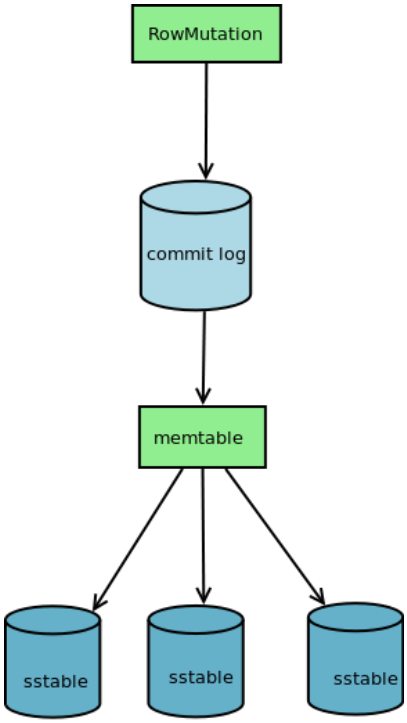
分布式的存储是为了利用数据的“分布”能力而不出现节点访问热点或瓶颈（另外由于 Replication 的缘故，达到数据多点备份来解决单点失效问题），而将业务相关的数据定义在嵌套的 ColumnFamily 中，势必影响数据的分布能力。况且不谈这一概念会给数据模型设计和实现带来多少的开销。。。

2. 数据存储 (Data Persistence)

- 存储体系
- BloomFilter
- CommitLog
- MemTable
- SSTable

2.1 存储体系

沿用 BigTable 的术语，内存中的表示用 memtable 表示，数据文件为 SSTable (Sorted String Table)。Write Path 需要通过以下几个层次：



- `CommitLog` 提交的“重做”日志，对数据的修改操作首先要写入这一日志中以确保数据的完整性

- MemTable　　内存中数据的表示形式，周期地存储到磁盘上。
- SSTable　　磁盘上数据的最终表示形式

2.2 BloomFilter

(1 - e ^ (-kHash * nElements / mBitSetSize)) ^ kHash

BloomFilter 由 Burton Howard Bloom 发明，*网上有很多开源的实现，我这里只是消化与吸收一下概念:*

1. 已知 **n** 个待查找的值，并且在标记集合中，每个值将分配 **c** 个标记位，则构造出 bitset，其大小为 **m** = n * c
2. 将每个已知值分别用 MD5 消息摘要函数求 Hash 1 ~ Hash **k**，
3. 对于每个 Hash 的结果（理论上该结果的取值范围是无限或超出目前机器能力）映射到有限的但冲突的概率很小的域 m 中（即在集合中作个标记, 这样，如果上面使用 8 个 Hash，则对于每个已知值会有 8 个标记 ）
4. 对于每个待查找的值，同样进行 step 2
5. 对于每个 Hash 使用，同样进行 step 3, 如果集合中没有做标记，则它肯定不存在，否则，

继续比较下一个 Hash k, 直到所有 k 个 hash 都比较完，如果，所有的 hash 都在集合中有标记，则

取决于 n, k, m 设置，这个值有可能存在, 存在的概率可以通过上面的公式求得。

结果出现 False Positive 的概率的原因是，我们将 HASH 函数理论上无限的值“映射”到有限长的 bitset 中，当冲突产生时，就会出现这种假存在。然而，结果决不会出现“假不存在”的可能，所以，大多应用是利用后者来做断言。

2.2.1. Java BloomFilter 实现

```
/*
 *  Public Domain
 *
 *  As explained at http://creativecommons.org/licenses/publicdomain
 *
 *  Written by pprun (quest.run@gmail.com)
 */
package filter;

import java.nio.charset.Charset;
import java.security.MessageDigest;
```

```
import java.security.NoSuchAlgorithmException;
import java.util.BitSet;

/**
 * <a href="http://en.wikipedia.org/wiki/Bloom_filter">BloomFilter</a> is a
non-deterministic algorithm for testing whether an element is a member of a
set.
 * It is non-deterministic because it is possible to get a <b>false-
positive</b> result, but not a false-negative.
 * Bloom Filters work by mapping the values in a data set into a bit array and
condensing a larger dataset into a digest string.
 * The digest, by definition, uses a much smaller amount of memory than the
original data would.
 *
 * @author <a href="mailto:quest.run@gmail.com">pprun</a>
 * @see <a href="http://www.google.com.hk/ggblog/googlechinablog/2007/07/bloom-
filter_7469.html">数学之美系列二十一 － 布隆过滤器（Bloom Filter）</a>
 */
public class BloomFilter<E> {

    private int mBitSetSize;
    private int nElements;
    private int kHash;
    private BitSet bitset;

    public BloomFilter(double bitsPerElement, int nElements, int kHash) {
        this.mBitSetSize = (int) Math.ceil(bitsPerElement * nElements);

        this.nElements = nElements;
        this.kHash = kHash;
        this.bitset = new BitSet(mBitSetSize);
    }

    public static long createHash(String val, Charset charset) {
        return hash(val.getBytes(charset));
    }

    public static long hash(String val) {
        return createHash(val, Charset.forName("UTF-8"));
    }

    public static long hash(byte[] data) {
        long h = 0;
        byte[] res = null;
        try {
            res = MessageDigest.getInstance("MD5").digest(data);
        } catch (NoSuchAlgorithmException ex) {
            System.out.println(ex);
        }


        for (int i = 0; i < 4; i++) {
            h <<= 8;
            h |= ((int) res[i]) & 0xFF;
        }
        return h;
    }
```

```
    public void add(E element) {
        long hash;
        String valString = element.toString();
        for (int x = 0; x < kHash; x++) {
            hash = hash(valString + Integer.toString(x));
            hash = hash % (long) mBitSetSize;
            bitset.set(Math.abs((int) hash), true);
        }
    }

    public boolean has(E element) {
        long hash;
        String valString = element.toString();
        for (int x = 0; x < kHash; x++) {
            hash = hash(valString + Integer.toString(x));
            hash = hash % (long) mBitSetSize;
            if (!bitset.get(Math.abs((int) hash))) {
                return false;
            }
        }
        return true;
    }

    /**
     * Calculate the false positive probability given the specified number of
inserted elements.
     *
     * <p>
     * (1 - e^(-kHash * nElements / mBitSetSize)) ^ kHash
     * </p>
     */
    public double falsePP(double nElements) {
        return Math.pow((1 - Math.exp(-kHash * (double) nElements
                / (double) mBitSetSize)), kHash);
    }

    public double p() {
        return falsePP(nElements);
    }

    public static void main(String[] args) {
        BloomFilter<String> geneFilter = new BloomFilter<String>(25, 8, 8);

        // Add dna to the bloom filter
        String tcaa = "tcaa"; // for assert
        String nonGene = "ccaaaaaaaa"; // for assert non gene

        geneFilter.add("aatt");
        geneFilter.add("atgca");
        geneFilter.add("tttt");
        geneFilter.add("gcaa");
        geneFilter.add("cctt");
        geneFilter.add(tcaa);
        geneFilter.add("gcgc");
        geneFilter.add("agag");
```

```
        //System.out.println(dna.bitset.toString());
        //System.out.println("---------------------------------------");

        if (geneFilter.has(tcaa)) {
            System.out.format("%s is a code gene with probability %f\n", tcaa,
(1 - geneFilter.p()));
        } else {
            System.out.format("%s is definitely not a gene\n", tcaa);
        }

        // ccaaaaaaaa
        if (geneFilter.has(nonGene)) {
            System.out.format("%s is a code gene with probability %f\n",
nonGene, (1 - geneFilter.p()));
        } else {
            System.out.format("%s is definitely not a gene\n", nonGene);
        }
    }
}
```

以上程序输出：

```
tcaa is a code gene with probability 0.999968
ccaaaaaaaa is definitely not a gene
BUILD SUCCESSFUL (total time: 0 seconds)
```

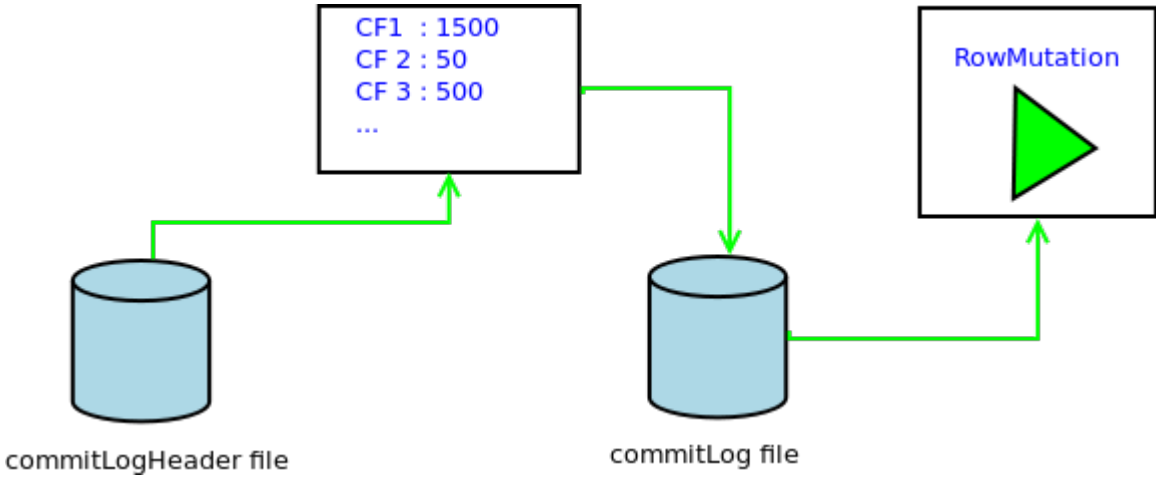
2.2.2 应用于分布式系统

对于一个索引主键 key 的查找过程可以利用 BloomFilter 快速判断 key 是否存在，而不需要 I/O 访问：

```
BloomFilter bf = ...;

   File ifile = ... ;

   public long getKeyPosition(Key key)

    {
        // step 1: check the bloom filter
        if (!bf.isPresent(key)) {
                return -1;
        }

        // step 2: check the key cache
        Long cp = getCachedPosition(key);
        if (cp != null)
            return cp;

        // step 3: scan the on-disk index
```

```
        ifile.open();
        ifile.seek();
        // ...
```

2.3 CommitLog



在存储系统中，CommitLog 是一种特殊的文件，代表所有的写操作（insert/delete/update)。

为了不丢失数据，任何写操作，首先被记录在 CommitLog 中，然后才是内存中的表示，最后，当内存中的表示到达预设定的极限值时，所有的写操作被存储到磁盘的数据文件。

2.3.1. 数据表示

2.3.1.1. RowMutation

继承于 BigTable, 对数据的操作的原子性保证是基于 Row，因此 insert/delete/update 操作被封装成 RowMutation:

```
RowMutation {
   keyspace
   key
   Map<ColumnFamily .id, ColumnFamily>
}
```

RowMutation Serializer

```
RowMutation
```

Keyspace.name						
key						
Cf 1.id						
Cf 1						
Name	Value	TimeStamp	Name	Value	TimeStamp	...
...						
Cf n.id						
Cf n						
Name	Value	TimeStamp	Name	Value	TimeStamp	...

2.3.1.2 CommitLogHeader

每个 CommitLog Header 文件是一个映射， 包含所有 ColumnFamily.id 到其在 commitLog 文件中的重做位置 (replay Position)。这样，在恢复的时候，先读取这个 replayPosition, 然后直接 seek(replayPosition) commitLog 进行恢复:

```
CommitLogHeader{

    Map<cfId, replayPostion> cfReplayPosMap

}
```

Map 表示

在恢复时，需要取出最小的 replay position, 然后从那个位置开始 replay:

ColumnFamily id	Replay position in commitlog file
1	50
2	150
3	5
...	...
n	...

CommitLogHeader Serializer

CommitLogBody
CfReplayPosMap.size
CRC(CfReplayPosMap.size)

Cf 1.id
Cf 1 replay position in commitlog file
...
...
Cf n.id
cf n replay position in commitlog file
CRC(cf1.id + cf 1 replay position + … + cf n.id + cf n replay position

* CRC 校验和

2.3.1.3 CommitLog

每个 CommitLog 文件包含一个 CommitLogHeader 队列，以及 executor，

```
CommitLog {
    Queue<CommitLogHeader>

    Executor CommitLogExecutor
}
```

CommitLog Serializer

CommitLog
sizeof(RowMutation Serializer)
CRC(sizeof(RowMutation Serializer))
RowMutation Serializer
CRC(RowMutation Serializer)

* CRC 校验和

2.3.2 同步策略

工业标准的两种策略：

- Write through
- Write back

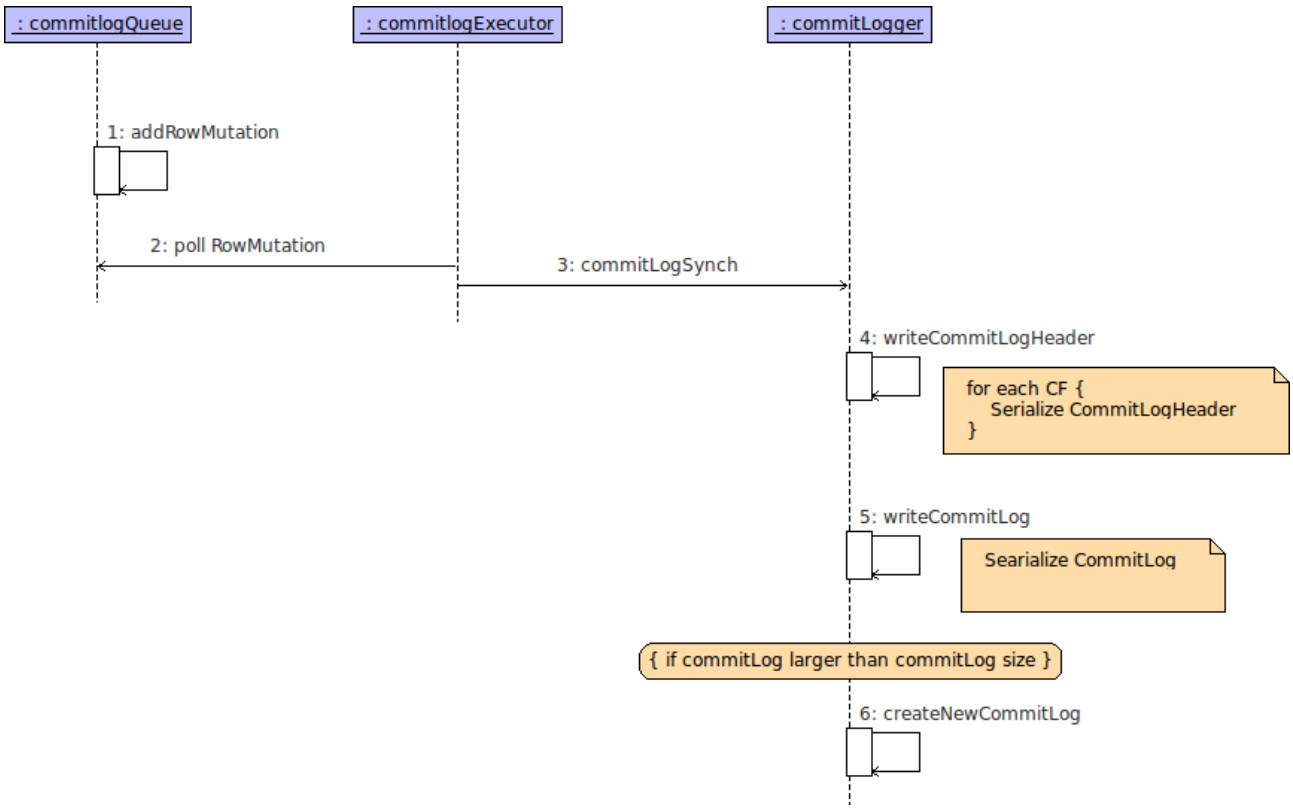
采用 write through 策略在分布式的环境里势必影响 写 的效率，所以通常的做法是采用改进的 batch 的方法，即，在同步发生之前，允许一定的 buffer，可以指定为 1s，此间内的所有的操作不是直接 fsynch, 而是将操作加入队列。（在意外情况下，这个 buffer 内的写操作有可能会丢失）

Write back, 本质上要简单且写效率也会大大提高。实现原理是将所有的写操作加入到队列，一个单一线程周斯地取出队列中的记录并进行同步这个写操作。缺点是，在意外情况下，一个周期内的写操作有可能会丢失（因为队列中的写操作还没有来得及同步到磁盘上）

2.3.3 处理流程

2.3.3.1 add RowMutation

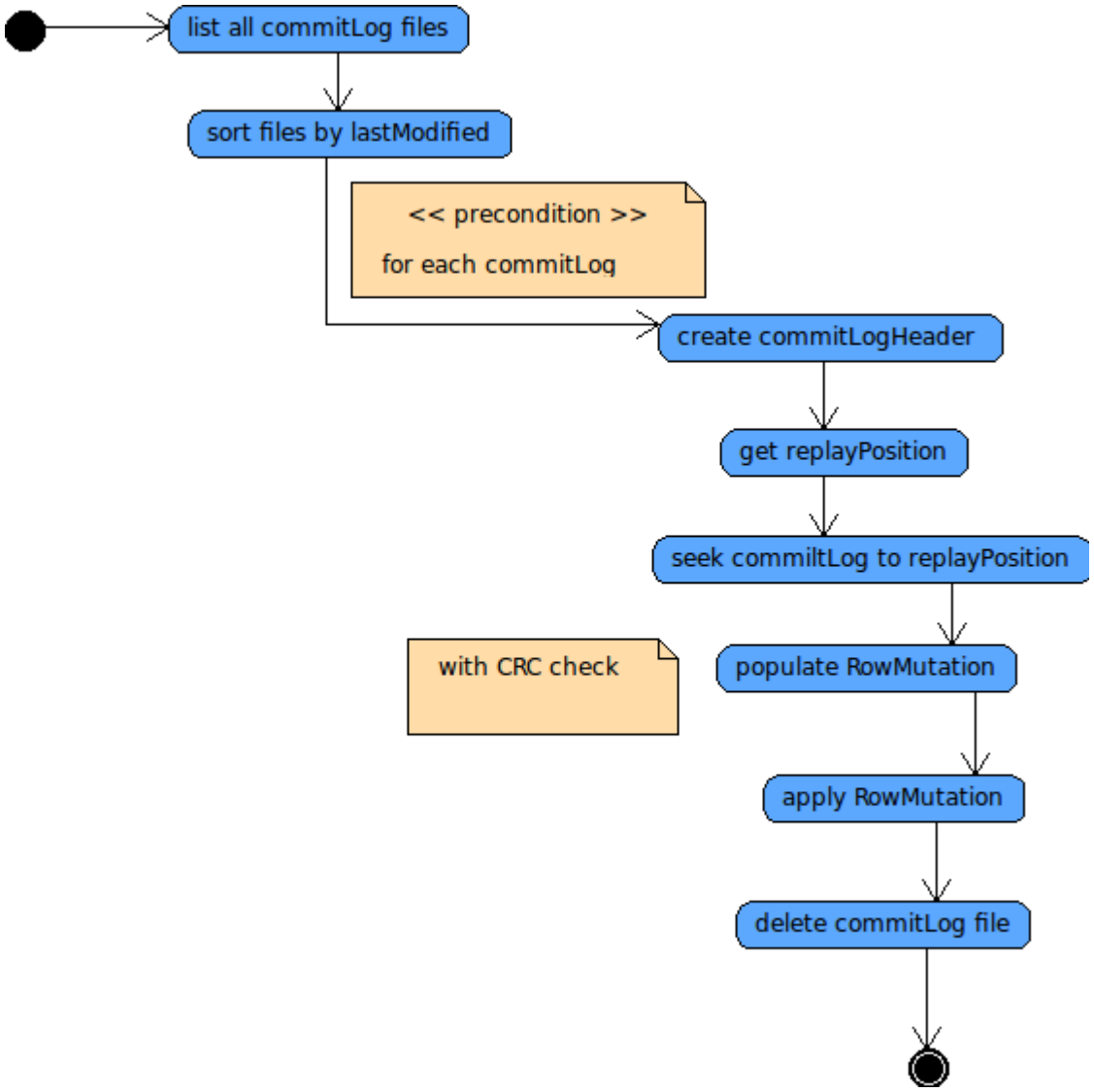
1. 对 columnFamily 的任何修改将导致一个 RowMutation 生成，并插入到 commitLogQueue
2. commitLog Executor 在执行周到来时从队列中取出 rowMutation
3. 将修改同步到 commitLog
4. 为每个 columnfamily 写 commitLogHeader 标记 (cf.id ： replayPosition)
5. 写 commitLog 的 serializer 表示
6. 如果 commitLog 文件的大小超过了预设定的值，重新生成一个新的 commitLog 文件



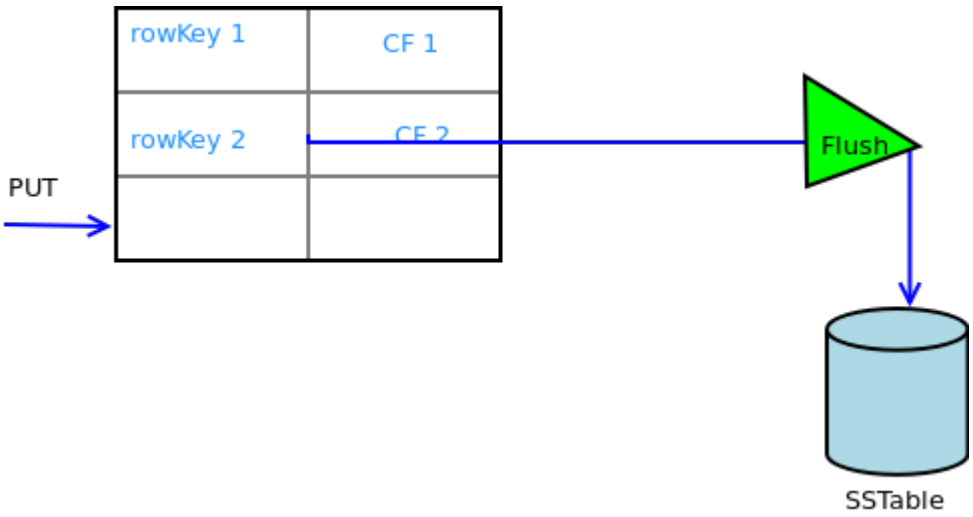
2.3.3.2 replay RowMutation

以下描述基于 commitLog 的恢复过程：

1. 从文件系统得到所有的 commitLog 文件
2. 对文件按最后修改时间进行排序
3. 对于每个 commitLog 文件
    1. 得到对应的 CommitLogHeader
    2. 得到 replay 位置
    3. seek 到 commitLog 相应的位置
    4. 基于 commitLog 数据生成(De-serialization) RowMutation
    5. 应用这个 RowMutation 到 columnFamily
4. 删除 replay 过的 commitLog 文件



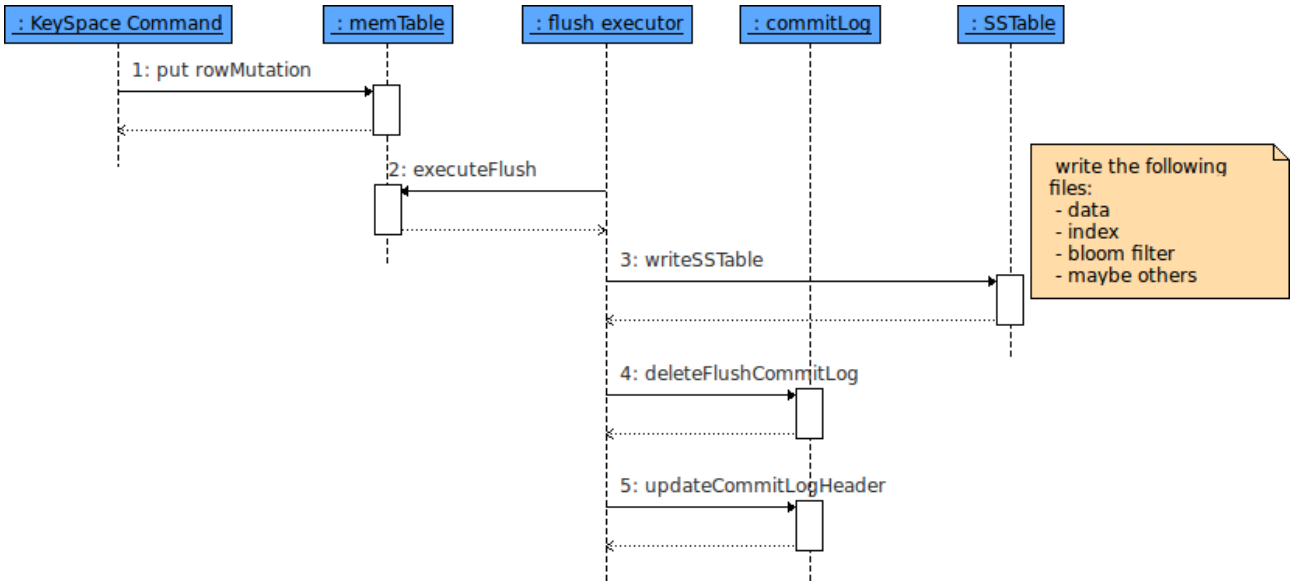
2.4 MemTable



从前面可知，CommitLog 是 Write path 上的第一道关，之后，就是第二道关 MemTable 了。MemTable 本质上是对每 Row 的所有 ColumnFamily 数据的一个 write-back 的缓存，当缓存的大小达到预设定的值时（即 Full 时), 所有 MemTable 的数据最终会写入 Write Path 的第三道关 SSTable 。

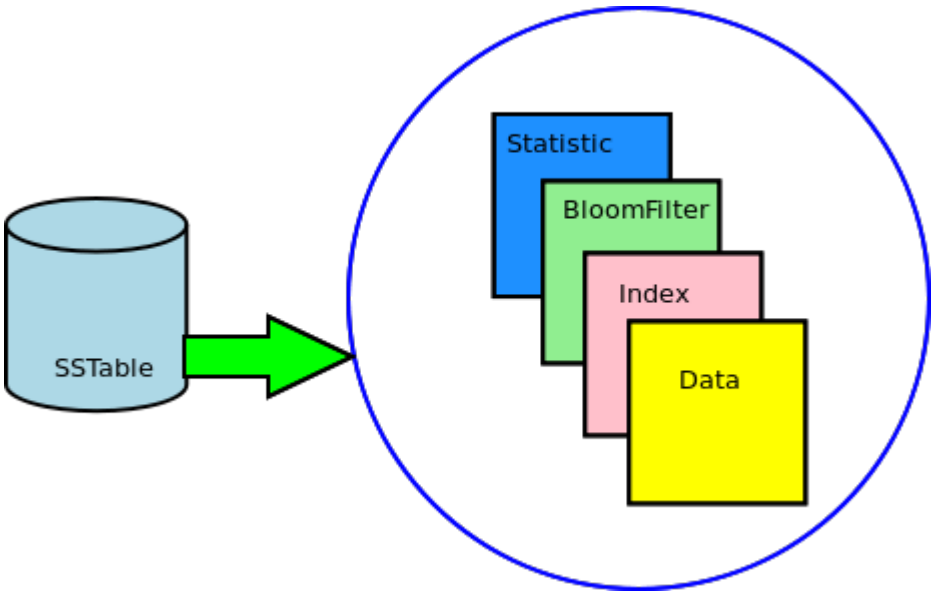
2.4.1 MemTable 处理流程

1. 对 columnFamily 的任何修改将导致一个 RowMutation 生成，在执行这一修改时将导致一条记录插入到 MemTable
2. 在达到 memTable 设定的极限值后 将执行 flush.
   Flush 一般情况需要在一个后台线程中进行，并要和对 ColumnFamily 的修改操作进行同步控制。
3. flush 将生成 (SSTable)
   1. data (columnfamily 的 serialiser 表示)
   2. index (Row 索引)
   3. bloom filter 的 serialiser 表示
   4. 其它文件系统，如果需要（比如　统计信息）
4. 因为数据文件已经生成，则 commitLog 不再需要
5. 同样，将 commitLogHeader 对应的标识删除



2.5 SSTable

按照 BigTable 的定义，SSTable 叫做 **S**orted **S**tring **Table**. 其本质上是一种文件格式用于存储数据到磁盘上。



2.5.1 SSTable 的种类

2.5.1.1 Data

Data 文件用于存储所有 ColumnFamily 的信息，即其包含的 Column 和 Column Index(注意：列索引没有单独的文件，如果采用 B+-Tree 则需要有单独的文件).

存储格式

SSTable Data File
Key.length
key
DataFile.size
ColumnIndex
BloomFilter (column)
sizeOf(below fields)
BloomFilter.hashCount
Bits.length
Bit[0] ~ bit[bits.length -1]
Column Index List
Index 0
sizeOf(below fields)
FirstColumnName.length
firstColumnName
LastColumnName.length
lastColumnName
StartPosition (offset in this File)
EndPosition – StartPosition (length of this Index element)
… Index N -1
ColumnFamily (Sorted Columns)
Column count
Column 0
ColumnName.length
columnName
ColumnValue.length
ColumnValue
TimeStamp
.. Column N -1

- columnIndex List 的元素由预设置的 size 来组织，例如，当 column_size = 64K, 当累加的 column 的大小超过这一值时，创建一个新的 index, 并加入到 columnIndex List
- column BloomFilter 用于对 Column 的访问

2.5.1.2 Row Index

包含 Row 的基于 rowKey 的索引

存储格式

Index File
Key List
Key 0
Key.length
key
StartPositoin in datafile for the key
Key N-1

内存中的数据表示 for Reader

Key0	StartPosition0 in data file
…	…
KeyN-1	StartPositionN-1 in data file

2.5.1.3 Row Key BloomFilter

Row Key BloomFilter 用于对基于 Row key 索引的访问。

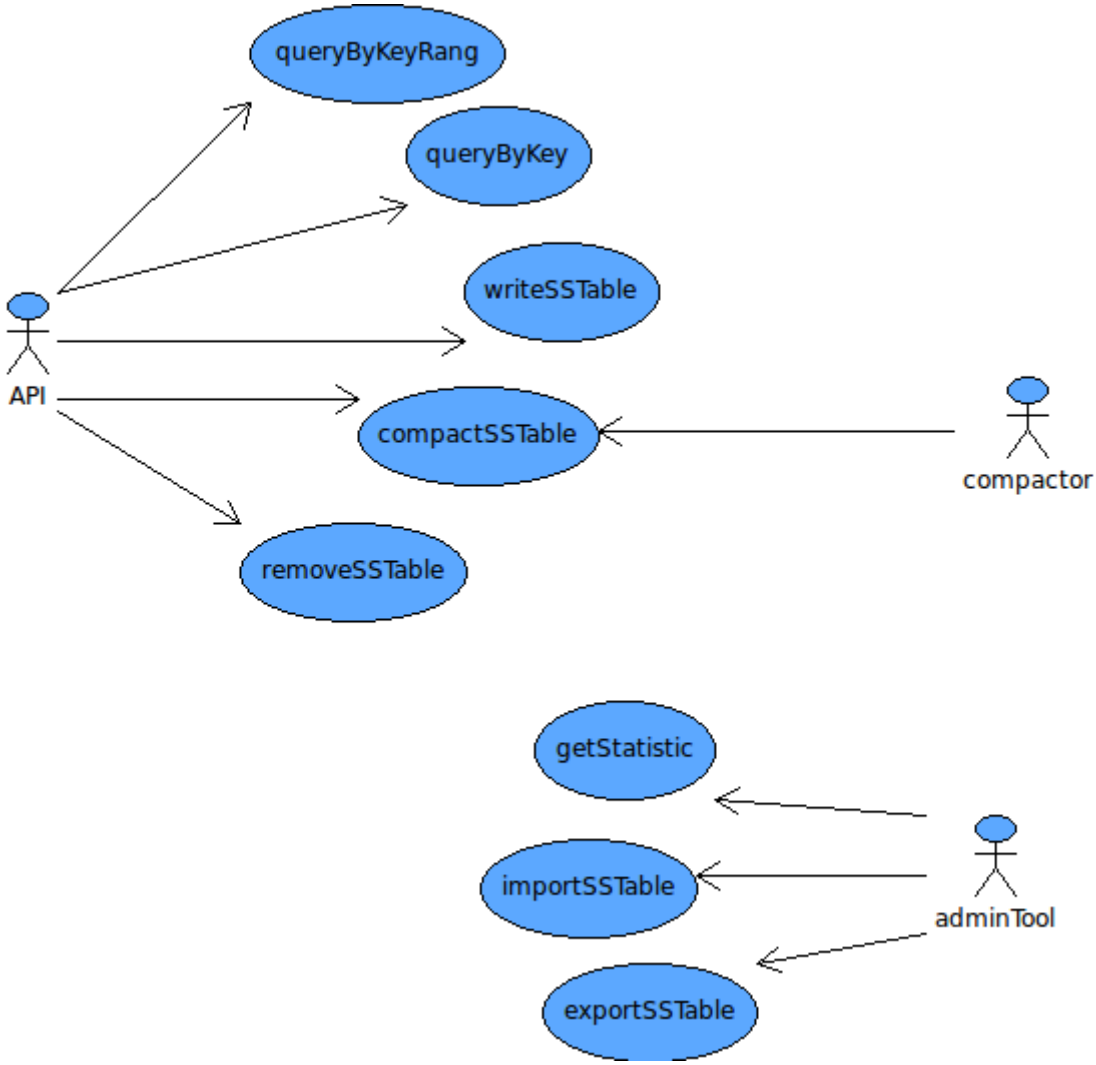
Row Key BloomFilter File
sizeOf(below fields)
BloomFilter.hashCount
Bits.length
Bit[0] ~ bit[bits.length -1]

2.5.1.4 Statistics

统计文件用于统计一个 SSTable 包含的 rowCount，columnCount。一般情况下会以柱状图的形式出现（Histogram）

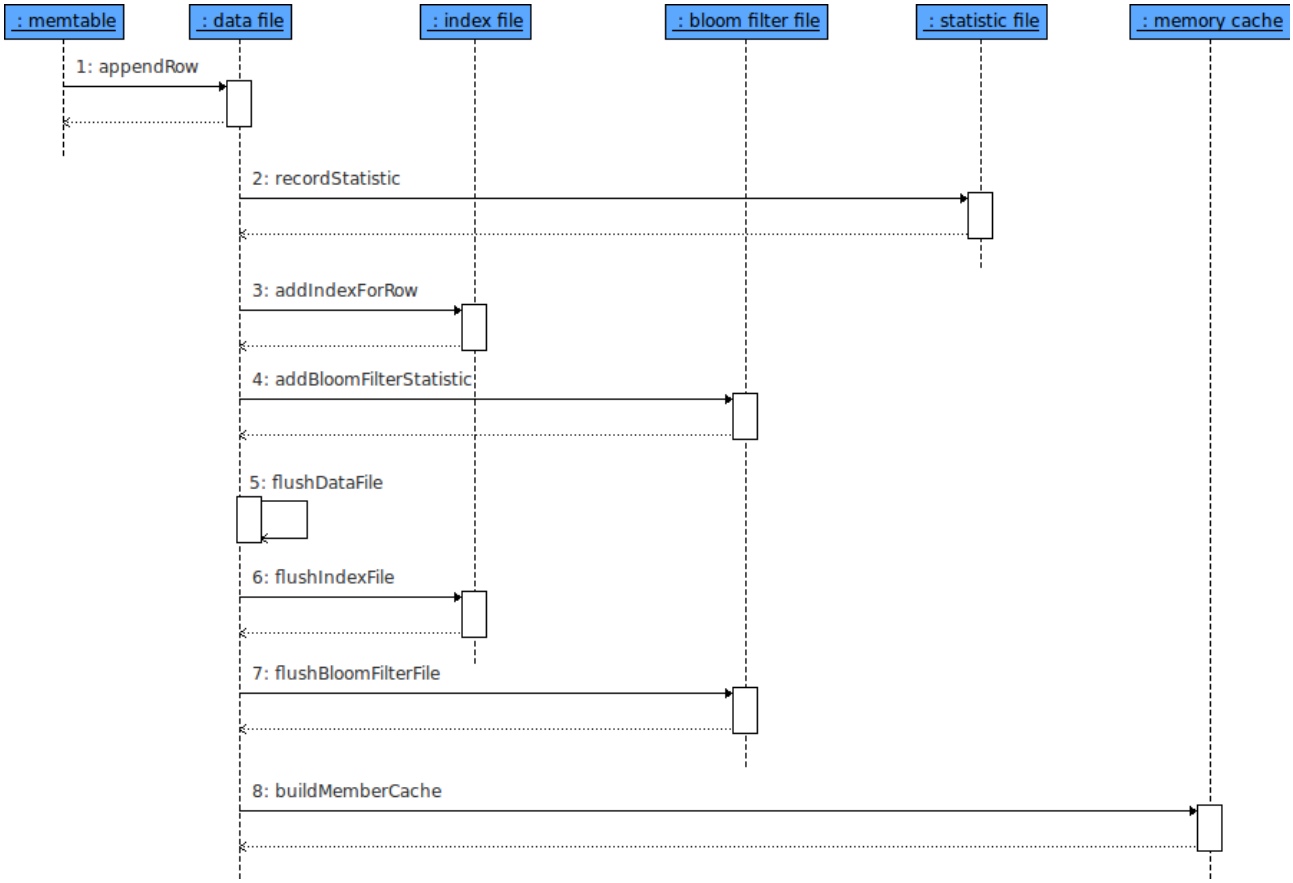
Statistics File
Row Count
Series [0] ~ Series[n-1]
Statistic[0] ~statistic[n-1]
Column Count
Series [0] ~ Series[n-1]
Statistic[0] ~statistic[n-1]

2.5.2 操作



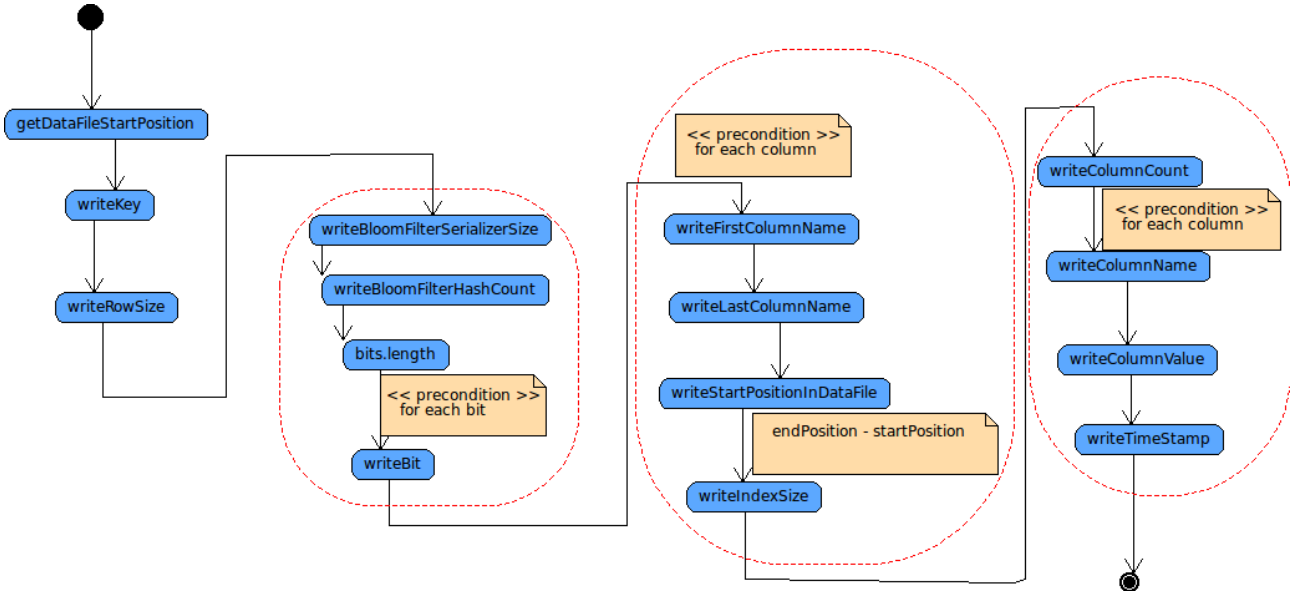
2.5.2.1 Write

当 MemTable 达到预设置的极限值时，其内容将依次被 flush 到磁盘上。SSTable 的 Data, index, filter, statistic 文件将被生成。



write data file

MemTable flush 时，写入 data 文件：



1. 因为 key 是按照升序排列的，所以，首先根据 key，获得其在文件中的位置
2. 将 key 写入
3. 整个 row 的大小写入
4. 写入 column index 的 bloom filter 的信息
5. 写入 column index 的信息 (start position, end position, index size)
6. 写入 column 的信息 (name, value, timestamp)

write Row index file

每次向 Data 文件增加一行记录时，都将向 Row index 文件加入以下数据：

- 写入 Key
- 写入 key 在数据文件中对应的 startPosition

write Row index BloomFilter

每次向 Data 文件增加一行记录时，都向 row Index bloomfilter 增加统计信息，当数据文件最终写到磁盘后，对应的 BloomFilter 文件将生成，原则上是将 BloomFilter 的 Serializer 形式写入磁盘即可：

- 写入 hashCount
- 写入 bits 数据的长度
- 依次写 bits [0] 到 bits[n-1] 的内容

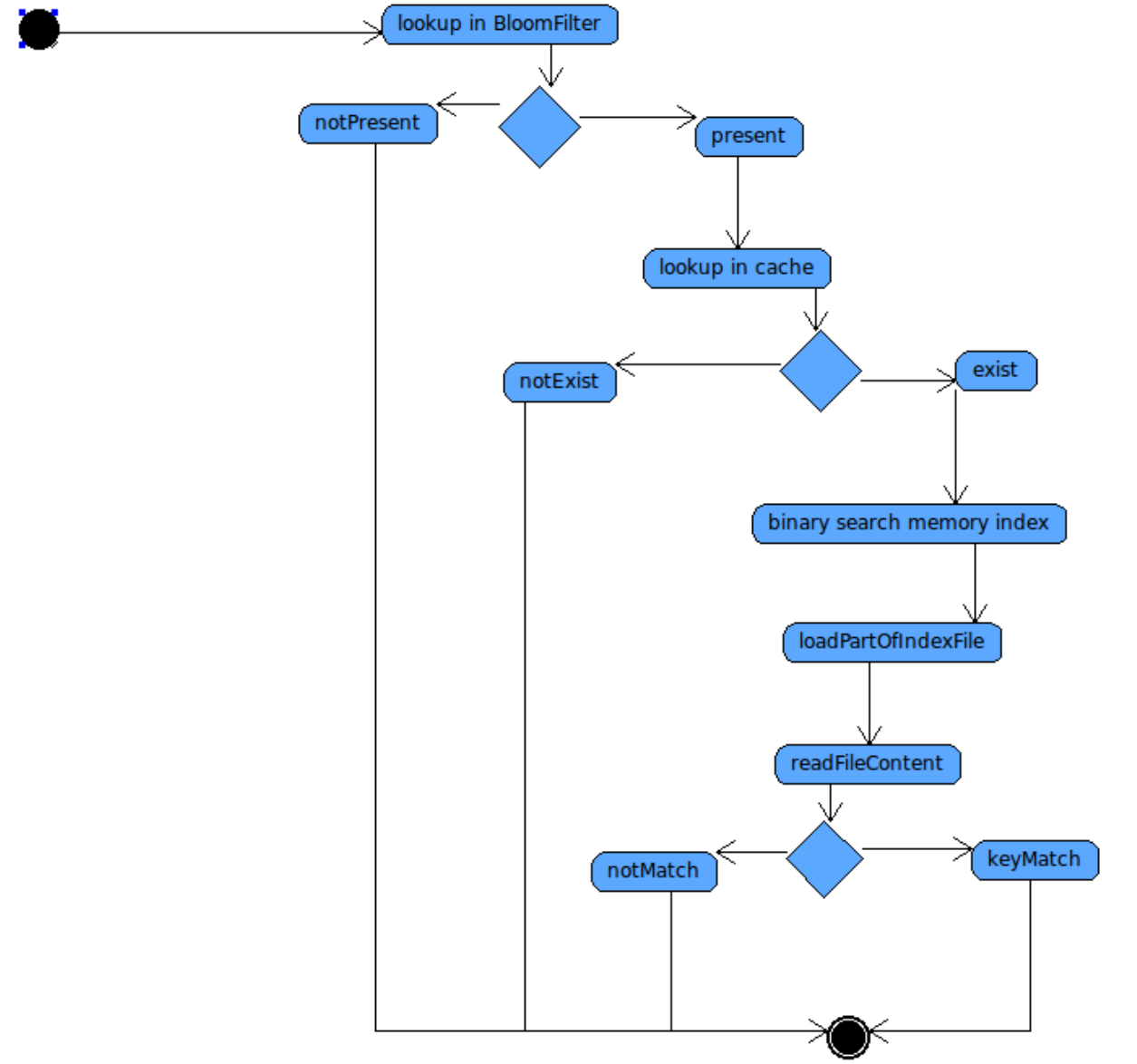
2.5.2.2 Read

当系统初始化时，所有的 SSTable 需要加载到系统.

QueryByRowKey

1. 首先通过 BloomFilter 看看 key 是否存在，如果不存在直接返回（注意：bloomFilter 永远不会返回假的不存在）
2. 如果存在，则需要进一步证实 BloomFilter 不是 false `Positive`。
3. 通过查找内存的 cache 看看 key 是否可以命中 cache, 如果命中，直接返回命中的记录；
4. 如果没有命中通过二分查找法在内存中的 index 信息中找到 key 在 data 文件中的位置并返回（startPosition)

以上过程中会涉及到数据统计信息（例如 bloomFilter false `Positive`）以及 `cache` 的更新（在通过磁盘加载到 `key` 的记录后）



QueryByRowKeyrange

基于 range 的查询返回的是一个 List，本质上是一个 getPostion(left key) 和 getPostion(right key)

2.5.2.3 Compact

归并(Compact，**这个词真没有好的对应的中文**) 可以分为两种类型：

minor

为系统自动执行，系统设置以下参数

- 最小 SSTable 个数 m1

- 最大 SSTable 个数 m2
- SSTable 大小平均值 A: 比如 50M

统计存在的 SSTable ，如果最终，大小处在 [½ A, 2/3 A] 的 SSTable 的个数超过了 m1, 则进行一次 minor 归并。

Major

Major 归并与 Minor 归并不同之处在于，将进行一次 ColumnFamily 级的剪切（ purge) – 将做了删除标识的 column 清除。

Major 归并可以通过手工触发，比如在系统资源不够时；
另外，当在自动进行 minor 归并时，如果代处理的 SSTable 列表包含了该 ColumnFamily 的所有的 SSTable(也即，完整地处理了一个 Column Family), 那么也将进行一次 major 归并。

对所有的 ColumnFamily, 为了效率起见，可以选择地忽略太大的文件

归并处理流程

1. 根据所有 SSTable 的大小，估算出待生成的归并 SSTable 的大小， 并试图得到归并后的文件路径
2. 如果磁盘目前大小满足不了这一请求，则依次去掉最大的文件，直接最后剩下两个文件，如果还没法满足，则报告错误
3. 如果 SSTable 列表即是此 ColumnFamily 的所有 SSTable, 则可以进行一次 major 归并
4. 根据所有 SSTable 的大小，估算出估计的行数，此参数用于生成 row index cache 的大小
5. 将所有 SSTable 按 key 的排序累积
6. 根据上述文件位置和估计的行数 生成一个输出文件 newSSTable
7. 依次迭代累积的 SSTable
   - 如果是一次 major 归并，或者当前 row 全部清除，则清除 columnFamily 中所有作了清除标记和需要清除的 column 以及 column Family，
   - 输出到 newSSTable
8. 生成新的 newSSTable 对应的 row index, filter, statistic 文件
9. 将新生成的 newSSTable 整理到系统中其它 SSTabe
10. 更新 cache

2.5.2.4 Delete

当多个 SSTable 被归并(Compact) 到一个更大的 SSTable 文件时，旧的 SSTable 文件需要删除以释放磁盘空间。
但由于在分布式的环境中，出于性能考虑，在 Compact 过程中，删除操作不会立即执行，而是将对应的文件加入到一个后台删除线程。

3. 系统元数据

- 系统 Column Family
- Index (Row and Column)

3.1 系统 Column Family

系统元数据维护节点如下信息，这些信息都存储在 ColumnFamily 中:

- Token 的映射信息
- 引导(bootstrap)信息
- 对等点的 Token 信息
- 建立了列索引的 Column Family 的信息

System Metadata		
Node Column Family		
Row key	**Column List**	
	Name	**Value**
“token”	Token	节点的 Token
	Startup times	启动次数
	Partitioner	Token partitioner 的名字
	ClusterName	集群的名字
“node id”	NodeId	节点的 id
“node ip”	“hint”	Hinted handoff 记录
“peer token”	Token	对等节点的 Token
“bootstrap”	Bootstrap	该节点是否被引导过
Index Column Family		
Row key	**Column List**	
	Name	**Value**
Keyspace name	Column family name	Empty ByteBuffer

3.2 Index (Row and Column)

因为系统的数据模型决定了系统需要应对非常大的维度 (行 x 列)。因此，以一个 ColumnFamily 为

存储单元，在行维度上数据量巨大时，为快速定位 Row，需要用到 Row 索引。
但通常的应用需要按数据列进行查询，因此需要列索引。

3.2.1 Index on Row Key

TBD

3.2.2 Index on Column

TBD

二. 动态平衡

- 分布式哈希表 (Distributed Hash Table - DHT)
- 备份/复制 (Replication)
- 成员关系 (Membership)

4 分布式哈希表 (Distributed Hash Table – DHT)

- 举例
- 保持平衡
- 1- hop DHT (Key-based routing)

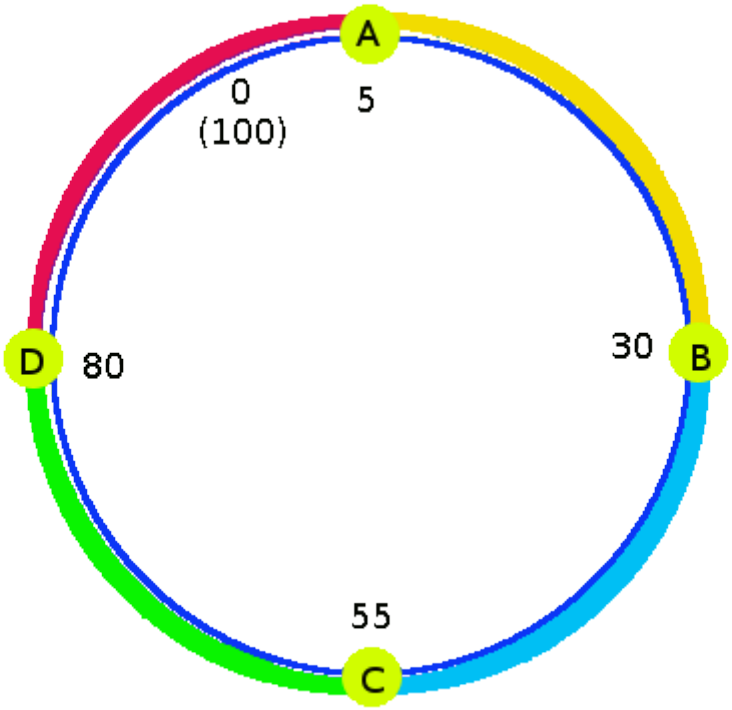
为了达到去中心化的设计目的，DHT 通过基于 key-based 路由策略来找到对应的存储节点：

- 整个分布式系统组成一个环，环根据节点的数目被化分为相应的区域
- 每个区域用一个 token 表示范围值 (n-1, n]
- 每个节点负责一个区域

通常采用 MD5 作为 hash 算法来分配 key 值给环上的区域：

- 设节点数为 N
- MD5 算法将输出一个 128 bit 的整数，为方便表示，去掉负数，因此 $0 \le token \le 2^{127}$
- 因此对于任一的节点，为其分配范围 (token n-1, token n]， **即 (前一节点的最大值，本节点的值]，**它将负责 MD5(k)值落在其中的所有 Key
- 必定存在一个 首尾相接的区域 (wrapping range), 其为一个特殊的区域：一定是包含最小 token 值的区域，且一般为 left > right, 其分配的值分成两部分：
  - MD5(k) = (0, token $_0$] 即最小节点值到范围的最小值（由于去掉负数，所以为 0)
  - MD5(k) = (token $_{n-1}$, 2^{127}],

4.1 举例



下面以一个 4 节点，token 范围为 [0, 100] 为例说明：

- A (80, 5] 此为 wrapping range（因为其包含了最小值 0，且左值大于右值）。 因为最小范围值规定为 0，最大范围值只能到 100，其实际可以拆分为 (80, 100] , (0, 5]
- B (5, 30]
- C (30, 55]
- D (55, 80]

非 wrapping range 比较好理解，(n-1, n] 表示从 n-1 开始（不包括 n-1）到 n;
对于 wrapping 其有以下特点：

1. 左值 > 右值
2. 包含最小值的 token 值
3. 其包含所有比当前所有分配的节点最大值的范围（本例中为 80) 但是，隐含条件必须小于最大的取值范围（本例中为 100), 所以得到 (80, 100]
4. 其包含所有比当前所有分配的节点最小值的范围（本例中为 5) 但是，隐含条件必须大于最小的取值范围（本例中为 0)，所以得到 (0, 5]
5. 0 与 100 重叠， 这即是 wrapping range 的来历

4.2 保持平衡

为了保持均衡的动态分配（即不引起热点），对于采用 MD5 算法的 DHT, 在分配节点 token 值时，需要按照以下公式计算对应的值：

$token_n$ = i * (2^127 / N) for i = 0 .. N-1 (N 为节点个数)

```
    // i * (2^127 / N) for i = 0 .. N-1
    static BigInteger bigIntegerToken(int i, int nNodes)
    {
        return BigInteger.valueOf(i).multiply(new
BigInteger("2").pow(127).divide(BigInteger.valueOf(nNodes)));
    }
```

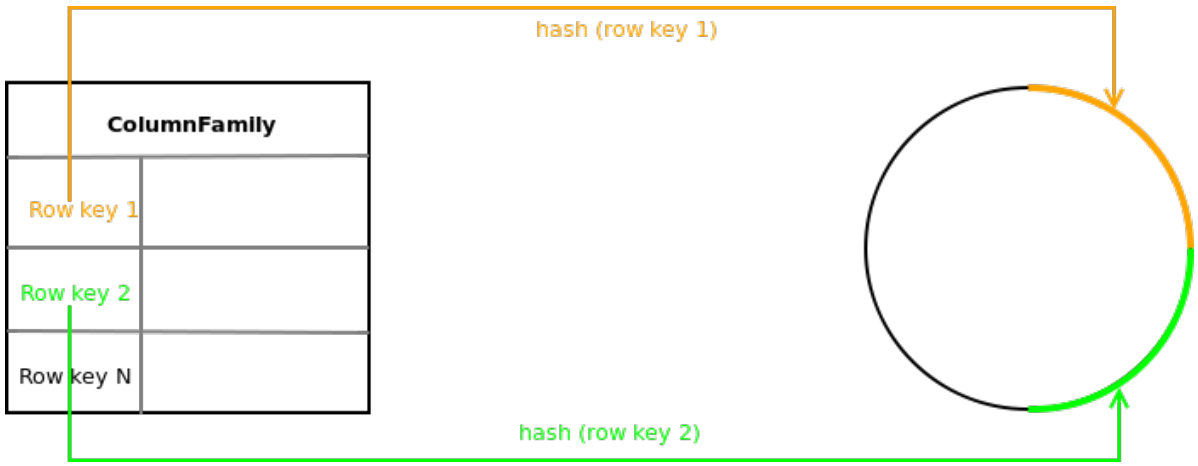
4.3 1- hop DHT (Key-based routing)

无论 read path 还是 write path, 一般对数据的访问都是通过协调节点计算: `keyspace → key → columnfamily` 的形式。

给定 key,

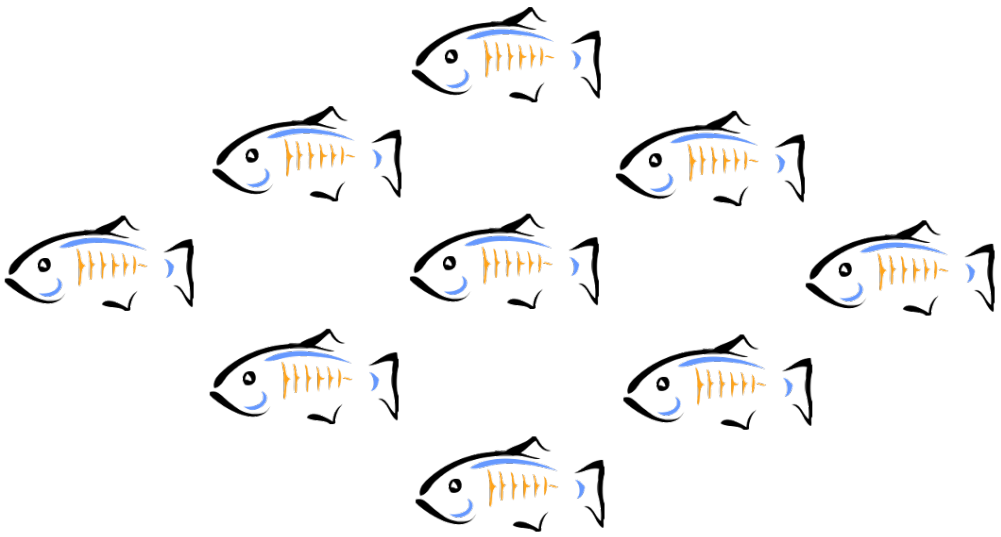
Key → hash (key) → Token_k → binarySearch (node's tokenList)

即可定位到节点。因此，此方案也被称为 1- hop DHT



5 备份/复制 (Replication)

- 备份策略
- 配置网络拓扑
- 动态平衡策略



（我第一眼看到 GlassFish 的 logo 时就特别喜欢，所以这里借用一下组成集群）

DHT 提供基于 Token 的逻辑拓扑环(logical topology - token ring ordering) 的方式来查找 Key 的主要责任（primarily responsible ）节点, 但 DHT 本身并不能提供容错（failure tolerant）能力。
为了在主要负责的节点失效的情况下，系统仍然工作，分布式的系统往往采用 replication 策略来保持 N 份拷贝在物理布局 (physical layout) 中位于不同的数据中心(Data Center)，或不同的机架的主机上 (Host IP addresses）。这个N被称为重复因子 (replication factor)。这样，在主要责任节点外失效的情况下，系统将通过联系其它**就近**的备份来完成请求。除了主要的责任节点外，那 N-1 个备份通过怎样的方式安置在其它备份节点上，即是下面要讨论的备份策略(Replication Strategy)。

5.1 备份策略

5.1.1 简单网络拓扑备份策略

- 第一个备份节点将被安置在与主要责任节点(primary responsible)不同的数据中心（如果有多个不同的数据中心）
- 第二个备份节点将被安置在与主要责任节点(primary responsible)在同一数据中心，但位

于不同的机架
- 如果还未达到 备份的个数， 随机地选择逻辑拓扑环(token ring ordering) 上还未使用的节点

5.1.2 用户定制的备份策略

- 用户提供一个配置，指定不同的数据中心和机架的权重（比如：根据系统的物理配置），各数据中心的权重和等于 N (比如 N = 6, datacenter 1 = 4, datacenter 2 = 2)
- 系统迭代数据中心列表，将备份节点安置不同的机架上
- 如果迭代完数据中心列表后，还未达到备份的个数，则试着将备份节点放在相同的机架但还未使用的节点上

5.2 配置网络拓扑

以上提到数据中心(Data Center)和机架(Rack) 概念，那么怎样配置这些信息给运行的系统呢?

5.2.1 Ip地址约定的模式

Ip 地址通用格式为： octet1:octet2:octet3:octet4
- octet2 为数据中心标识地址，不同的数据中心分配不同的 octet2
- octet3 为同一数据中心不同机架标识地址，不同的机架分配不同的 octet3

5.2.2 配置文件模式

通过配置文件指定数据中心和机架信息，例如:

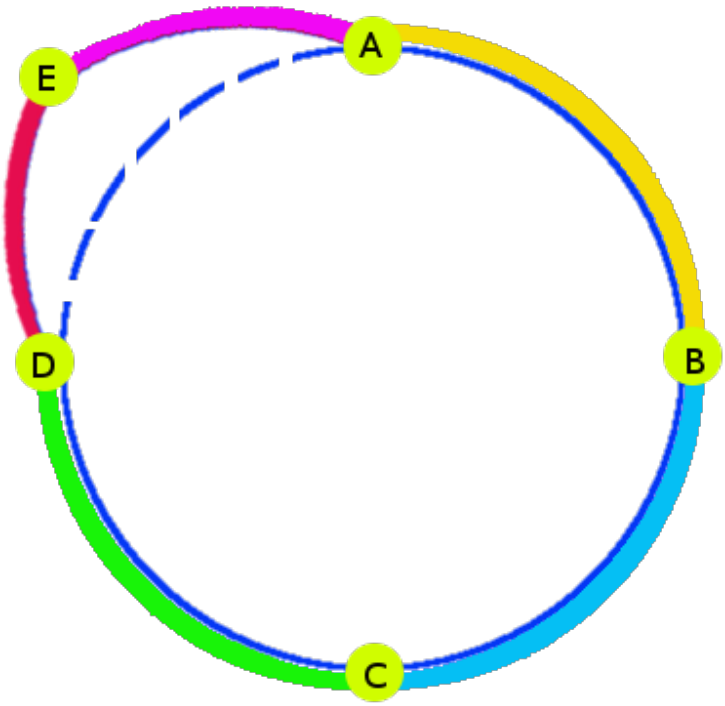
```
# ip 地址=数据中心名:机架名
10.0.0.10=甲:子
10.0.0.11=甲:子
10.0.0.12=甲:丑

10.20.0.10=乙:子
10.20.0.11=乙:丑
```

5.3 动态平衡策略

上面提到过，当主要负责的节点失效的情况下，系统将就近地选择备份节点来完成请求。但是，可以想象，地理位置近的节点，也许在特定的情况下并不是最好的选择，比如负载；比如相同 range 的 key 的访问，在主节点失效的情况下，如果所有的请求都被重定向到相同的就近的节点。这种情况下，最好的方法是记录整个 N replica 的 metric 信息，这些信息可以分为：负载，响应延时，健康状况，等等。
根据不同的 metric 监测方法，可以实现不同的动态平衡策略，在后面要讲到的一种 **φ 累积失效检测器**模型，通过这种模型，可以为各节点的 latency 打分，latency 值越小则越有机率被选择为侯选的节点。

6 成员关系 (Membership)

- Join
- Leave
- 响应成员变化
- 成员关系图谱



对用户来讲，集群中的所有节点都同等的。实际上，从上面的 DHT 原理看出， 每个节点只负责整个数据集的不同部分。但是，每个节点又必须具备服务任何特定的 key 的请求。为了达到这种对用户透明度，每个节点必须维护一些原数据：

- 所有其他节点的列表
- 所有节点当前状态
- 对于一个给的 key, 其主要负责节点和 n-1 副本节点 （n 为 replication factor）信息。

6.1 Join

一个节点加入到 Token 环的处理流程:

1. 启动 gossip 服务 (需要和其他节点通信以获得对等信息)
2. 启动消息服务
3. 发起各节点广播各自负载信息
4. 如果是一个新节点
    1. 如果配置为该新节点指定了 token，则使用它，否则
    2. 基于存在的节点的负载信息得到，得到当前负载最大的节点，并从它那里得到

token，该 token 将大约分担它的一半负载
3. 本地更新并保存这个 token
4. Gossip 状态信息到所有其他节点，接下来将做进行引导了（ bootstraping)
5. (开始引导过程)
    1) 基于 keyspace 的配置信息，得到可能的负责这部分数据的节点列表
    2） 向他们发起 pipelineIn 请求，传递这部分数据

5. 如果是一个曾经启动过的节点（即 down，可能是人工和错误原因导致), 从配置表读取保存的 Token
6. 对本地系统元信息 应用这个更新： new Token
7. Gossip 状态信息，通知其他节点这个节点的新的 token

6.2 Leave

一个节点离开 Token 环 的处理流程:

1. Gossip 节点即将 leaving 的状态
2. 重新调整 Range 信息
    1. 得到所有受影响的 range
    2. 得到新的负责这些 range 的节点
3. 计算要传送给其他节点的数据
    1） 得到即将离去的节点负责所有区域 (**＊进一步描述，见成员关系图谱**)
    2） 得到当前负责这些区域的所有副本的地址
    3） 合计出当该节点离开后， Token 的新分配
    4） 对各个在第 1）步中的 Range, 重新计算其副本的地址
    5） 计算 Range 之差：最终剩下的 Range 即是需要的 range
4. 向所有将代替当前节点的节点 pipeline out 数据
5. 将当前节点从 token 元数据中移除
6. Gossip 消息到其他节点，当前节点已经离开了
7. 停止该节点的 Gossip 服务
8. 停止该节点的消息服务

6.3 响应成员变化

存在的节点在收到节点状态改动时的动作在不同的状态变迁时虽然有些不同，但大体上执行以下操作：

1. 状态检验，以确保不会造成冲突
2. 记录对应节点的该状态，以用于校验
3. 更新（增加／修改／删除）系统元数据
4. 更新 Range 信息
    1） 得到所有受影响的 range
    2） 得到新的负责这些 range 的节点

6.4 成员关系图谱

6.4.1 Node IP : Range 映射

1. 将所有 Token 排序
2. 对于每个 Token, 得到它们各自负责的区域，即 (前继, 当前]
3. 对于每个 Token，基于复制因子 N, 得到当前首选列表中前 N 个节点
4. 以上形成 ip → rang 的一个映射

6.4.2 Node Token : Range 权重 (Token 环百分比)

- 只有一个节点： $Token_{Node\ 1}$: 100%
- N 个节点: Token : ((Token(i) - Token(i-1) + 2^127) % 2^127) / 2^127

例如 Token 环被 4 个节点均分的情况:

```
Token1: 0.25
Token2: 0.25
Token3: 0.25
Token4: 0.25
```

三. 一致性

- 可配置的 一致性 (Tunable Consistency)
- Vector Clock vs. Client Timestamps
- MerkleTree
- Hinted handoff
- 反熵 (Anti-Entropy)
- Read Repair

7 可配置的 一致性 (Tunable Consistency)

- 可观察到的模型
- 可配置的一致性

人们在权衡系统的一致性，可用性和容错性的过程中，总结出一套理论：”放松一致性将使可用性得到提升，这也隐含了容错性的提升（在极端的情况，允许网络分割）；反过来，提高一致性将导致系统在某些情况不可用“，这即是 CAP 理论：三者（data consistency, system availability, and tolerance to network partition） 只能取其二 。

7.1 可观察到的模型

(详细的讨论，见 Eventually Consistent)

1. 强一致性(strong consistency) 只要更新操作完成，后续对数据的访问都看到更新的内容
2. 条件强一致性　　如果没有出现错误的情况下，能保证强一致性，但，如果出现错误，则不能保证任何一致性
3. 弱一致性　　存在一个短暂的时间段（inconsistency window)，后续的操作不能保证看见最近的更新
4. 最终一致性　如果没有新的更新，最终后续的操作都将看见最近的更新；如果没有失效发生，这个不一致的窗口是由以下因素决定的：通讯延时，系统负载和副本数

该模型又可以分为以下几种（为简单起见略去不常见的）

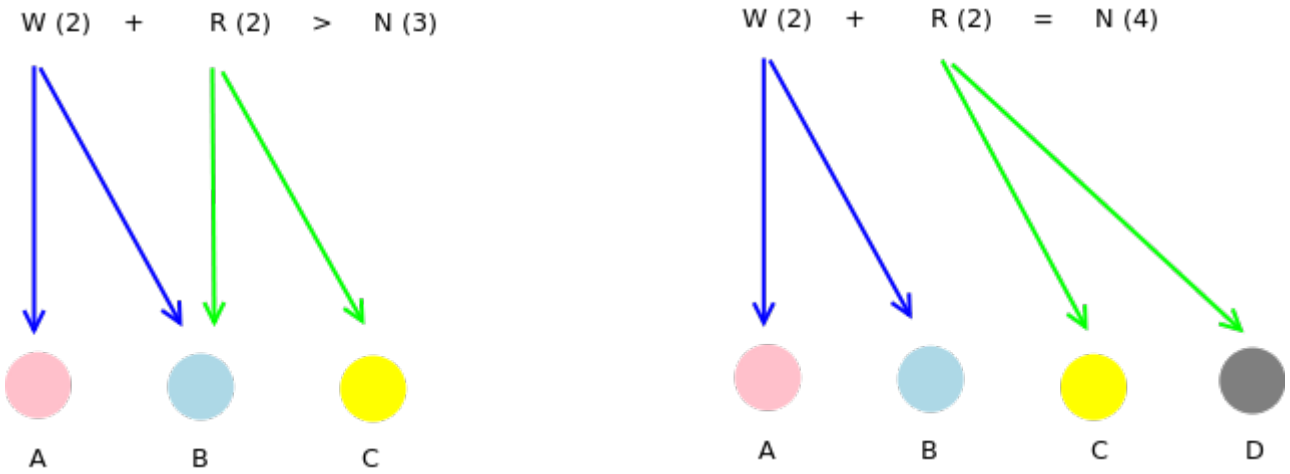
4.1　因果关系(causal consistency)一致性 两次更新存在“因果”关系，那么后面的更新将包括前面所有的更新

4.2　读你所写(read your writes)一致性　　当对数据作了更新，后续的操作将一定能看见这个更新，而不会看到老的版本

4.3　单调读一致性(Monotonic read consistency)　　当看到一个特定的值后，后续的访问不会看到任何比这个特定的值老的值

7.2 可配置的一致性

W + R (>, =, <) N



- **N** - 存储数据所有副本的节点数 （简称，副本数；即，包含该数据的节点的数，包括主节点 (primary node)）
- **W** - 在一写操作成功前，需要参与写的副本数
- **R** – 在一个读操作成功前，需要联络的副本数

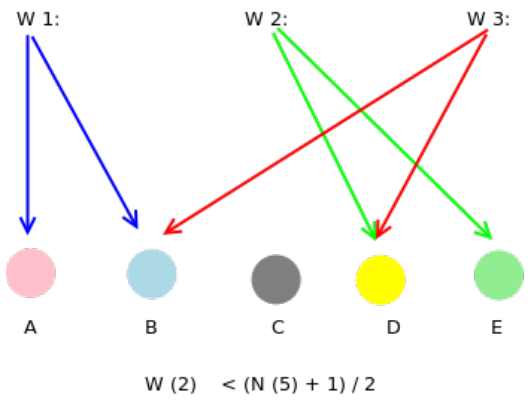
1. **W + R > N　强一致性**
   如图所示，W(2) + R(2) > N (3), 至少会有一个节点重叠，在写操作完成后，读操作一定会读到至少一个最新的写，然后，读操作会将版本最新的返回客户端，并启动 read-repair 用最新的数据来更新陈旧的节点

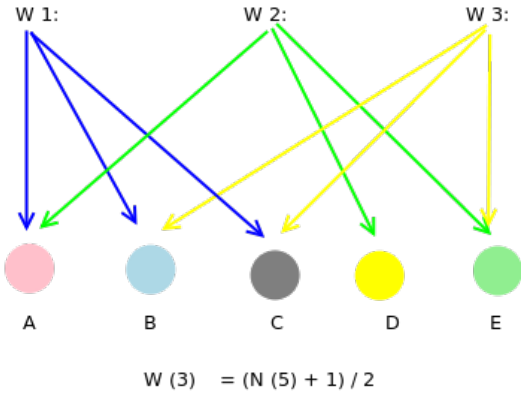
2. **W + R <= N 弱/最终一致性**
   如图所示 W(2) + R(2) = N (4) (W(2) + R(2) < N (5) 同理)，这种情况碰巧，读和写没有重叠，所以，读将会读到陈旧的数据。

3. **W = 1, R = N**　专门为“写“的优化配置的**条件强一致性**
   只要一个节点写成功，写就可以认为成功。写是飞快的！但是， 如果正好那台写成功的机器在完成写后就失效了且写没有传播出去，在它恢复前，是非一致性的。（当然，实现要求，在该节点恢复后，那个写要被传播出去，如果还是有效的话）

4. **R = 1, N = W**　专门为“读”的优化配置的**强一致性** 只要一个节点成功响应读操作，读就可以成功返回。

5. **W < (N+1)/2**　存在写冲突





如图,

- **w(2) < (N(5) + 1) / 2** 写操作 W3 要更新的两个节， 之前存在两个不存在因果关系的更新操作 W1(应用于 B) 和 W2(应用于 D)，因此发生冲突
- **w(3) = (N(5) + 1) / 2** 因为在任何情况下，都能保证更新的节点，存在因果关系，因此不会发生冲突

6. **W >= (N+1)/2; R >= (N+1)/2** 在一些特定的实现中，比如 Cassandra 中叫做 Quorum
   通常的应用使用配置 W(2) + R(2) > N(3) ，这个配置满足强一致性配置条件：W+R>N
7. **ALL（W = N; R = N）** **强一致性**，但是性能和可用性不好

8. **只对 “写“的场景**
   因为读操作必须实际地读到数据才能返回。所以，以下设置对 Read 无意义。
   如果 W > 0, 那么，意味着，在写操作成功返回前，必须至少一个副本节点是活着的。但为了提高写操作的可用性，放松一致性要求，特定的配置如下：

   - **W = 0**， 不保证一致性，写操作异步地执行且立即返回成功，
   - **W = any**， 不保证一致性，如果所有首选列表节点都失效，只要有任何一个非首选节点存活，写操作成功，在首选节点修复后，将更新写回到本该发生的首选节点。这一特征叫做 Hinted handoff

在非强一致性的配置下，如果要达到”读你所写(read your writes)”一致性，一般需要用到乐观锁（版本号或时间戳），将进来的更新请求所携带的锁与当前系统维护的版本相比，如果过时了，更新失败（太乐观）；对于读操作，结果将被丢弃。

而对存在写冲突的配置，如 **W < (N+1)/2**，需要有冲突解决方案，典型的是：矢量时钟和客户端时间戳。下面进行讨论。

8. Vector Clock vs. Client Timestamps

- 矢量时钟
- Voldemort 's VectorClock
- Client TimeStamp

8.1 矢量时钟

Vector Clock(参考 版本数据) 是服务端用于存在因果关系的版本冲突的解决方案，目的是减轻版本冲突对 Client 的侵扰，从而降低应用的复杂度；
但是，在特定的弱/最终一致性的配置下，会出现非因果关系的版本冲突，这种情况下，服务端就无能为力了，只有依赖于 Client 的协助。

8.2 Voldemort 's VectorClock

如对 vectorclock 感兴趣，Voldemort 实现了完整的 VecotClock 版本冲突解决方案，其核心代码如下：

8.2.1 比较

```
    /**
     * Is this Reflexive, AntiSymetic, and Transitive? Compare two
VectorClocks,
     * the outcomes will be one of the following: -- Clock 1 is BEFORE clock 2
     * if there exists an i such that c1(i) <= c(2) and there does not exist a
j
     * such that c1(j) > c2(j). -- Clock 1 is CONCURRENT to clock 2 if there
     * exists an i, j such that c1(i) < c2(i) and c1(j) > c2(j) -- Clock 1 is
     * AFTER clock 2 otherwise
     *
     * @param v1 The first VectorClock
     * @param v2 The second VectorClock
     */
    public static Occured compare(VectorClock v1, VectorClock v2) {
        if(v1 == null || v2 == null)
            throw new IllegalArgumentException("Can't compare null vector
clocks!");
        // We do two checks: v1 <= v2 and v2 <= v1 if both are true then
        boolean v1Bigger = false;
        boolean v2Bigger = false;
        int p1 = 0;
        int p2 = 0;

        while(p1 < v1.versions.size() && p2 < v2.versions.size()) {
            ClockEntry ver1 = v1.versions.get(p1);
            ClockEntry ver2 = v2.versions.get(p2);
            if(ver1.getNodeId() == ver2.getNodeId()) {
                if(ver1.getVersion() > ver2.getVersion())
                    v1Bigger = true;
```

```
                else if(ver2.getVersion() > ver1.getVersion())
                    v2Bigger = true;
                p1++;
                p2++;
            } else if(ver1.getNodeId() > ver2.getNodeId()) {
                // since ver1 is bigger that means it is missing a version that
                // ver2 has
                v2Bigger = true;
                p2++;
            } else {
                // this means ver2 is bigger which means it is missing a
version
                // ver1 has
                v1Bigger = true;
                p1++;
            }
        }

        /* Okay, now check for left overs */
        if(p1 < v1.versions.size())
            v1Bigger = true;
        else if(p2 < v2.versions.size())
            v2Bigger = true;

        /* This is the case where they are equal, return BEFORE arbitrarily */
        if(!v1Bigger && !v2Bigger)
            return Occured.BEFORE;
        /* This is the case where v1 is a successor clock to v2 */
        else if(v1Bigger && !v2Bigger)
            return Occured.AFTER;
        /* This is the case where v2 is a successor clock to v1 */
        else if(!v1Bigger && v2Bigger)
            return Occured.BEFORE;
        /* This is the case where both clocks are parallel to one another */
        else
            return Occured.CONCURRENTLY;
    }
```

8.2.2 冲突解决

```
    public List<Versioned<T>> resolveConflicts(List<Versioned<T>> items) {
        int size = items.size();
        if(size <= 1)
            return items;

        List<Versioned<T>> newItems = Lists.newArrayList();
        for(Versioned<T> v1: items) {
            boolean found = false;
            for(ListIterator<Versioned<T>> it2 = newItems.listIterator();
it2.hasNext();) {
                Versioned<T> v2 = it2.next();
                Occured compare = v1.getVersion().compare(v2.getVersion());
                if(compare == Occured.AFTER) {
                    if(found)
                        it2.remove();
                    else
```

```
                        it2.set(v1);
                }
                if(compare != Occured.CONCURRENTLY)
                    found = true;
            }
            if(!found)
                newItems.add(v1);
        }
        return newItems;
    }
```

8.2 Client TimeStamp

相比于矢量时钟，客户端时间戳 (Client TimeStamp) 的实现要简单的多，也不存在 Vector Clock 的限制：

- 只能解决存在因果关系的冲突
- 必须限制每个对象 矢量时钟 的数量

但是 Client Timestamp 同样存在缺点：

- 客户端接口需要提供附加的参数 Timestamp，这或多或少增加了些复杂度
- 所有客户端的系统时钟必须同步，这在某些情况下可能会比较麻烦

8.2.1 比较

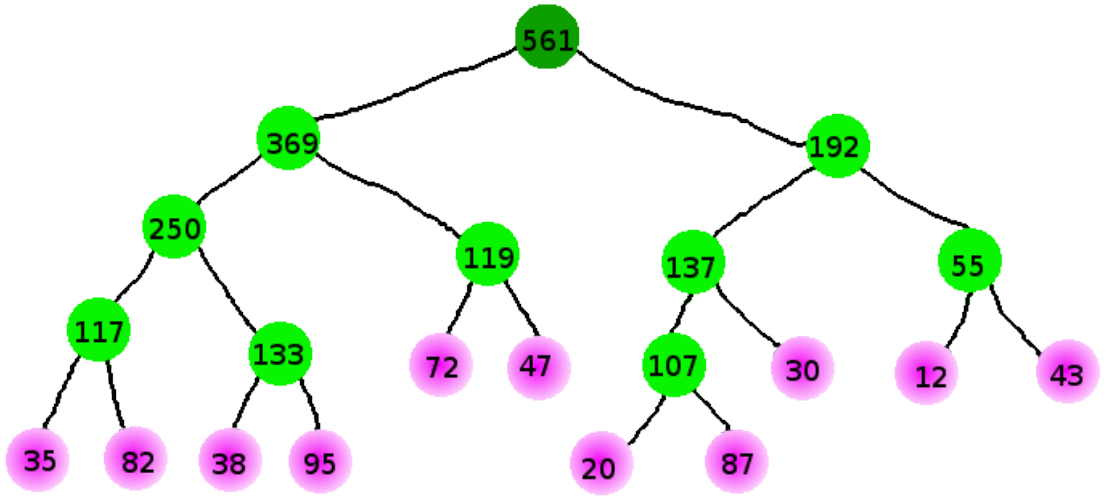
```
    public Column compare(Column column) {
        if (this.timestamp() < column.timestamp()) {
            return column;
        }
        return null;
    }
```

8.2.2 冲突解决

```
    public Column resolveConflicts(IColumn column) {
        if (timestamp() == column.timestamp())
            return value().compareTo(column.value()) < 0 ? column : this;

        return timestamp() < column.timestamp() ? column : this;
    }
```

9. MerkleTree

- MerkleTree 的 Java 实现



MerkleTree

树无处不在!
Binary-tree, 234-tree, Red-black tree, B-tree, B+-tree, ...

这里要说的是 MerkleTree 也称为 HashTree, 其叶子节点，存储的是代表示的数据，而非叶子节点存储的是其子节点的 Hash (如果，有两个子点子，将两个节点的数据合成在一起再计算 hash, 或者将各个的 hash 异或)。其被广泛用于 P2P 网络的数据交换过程。

9.1 MerkleTree 的 Java 实现

```
/*
 *  Public Domain
 *
 *  As explained at http://creativecommons.org/licenses/publicdomain
 *
 *  Written by pprun (quest.run@gmail.com)
 */
package tree;

import java.nio.charset.Charset;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;

public class MerkleTree {

    private Node root;
    private int depth;
    private Map<Long, Node> leafToNode;
```

```
    static class Node implements Comparable<Node> {

        long hash;
        Node parent;
        Node leftChild;
        Node rightChild;
        int leftSubtreeSize;
        int rightSubtreeSize;

        void updateHash() {
            long leftHash = leftChild == null ? 0 : leftChild.hash;
            long rightHash = rightChild == null ? 0 : rightChild.hash;
            // for demo
            hash = hash(String.valueOf(leftHash)) +
hash(String.valueOf(rightHash));
            //hash = hash(String.valueOf(leftHash)) ^
hash(String.valueOf(rightHash));
        }

        @Override
        public String toString() {
            return ((leftChild == null || rightChild == null) ? (hash + "*") :
Long.toString(hash));
        }

        public static long hash(String val) {
            // for demo
            return Long.parseLong(val);
            //return createHash(val, Charset.forName("UTF-8"));
        }

        public static long createHash(String val, Charset charset) {
            return hash(val.getBytes(charset));
        }

        public static long hash(byte[] data) {
            long h = 0;
            byte[] res = null;
            try {
                res = MessageDigest.getInstance("MD5").digest(data);
            } catch (NoSuchAlgorithmException ex) {
                System.out.println(ex);
            }


            for (int i = 0; i < 4; i++) {
                h <<= 8;
                h |= ((int) res[i]) & 0xFF;
            }
            return h;
        }

        @Override
        public boolean equals(Object obj) {
            if (obj instanceof Node == false) {
                return false;
```

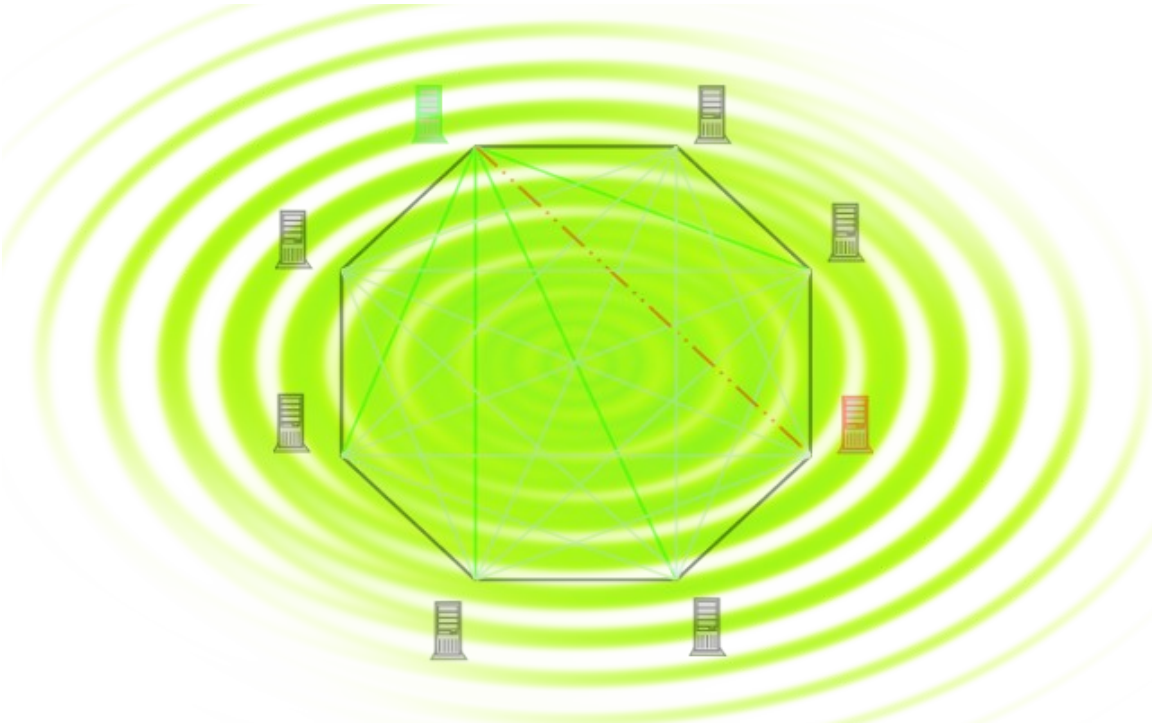
```
            }
            Node that = (Node) obj;
            return hash == that.hash;
        }

        @Override
        public int hashCode() {
            int result = 3;
            result = 83 * result + (int) (this.hash ^ (this.hash >>> 32));
            return result;
        }

        public int compareTo(Node o) {
            return hash - o.hash == 0 ? 0 : hash - o.hash > 0 ? 1 : -1;
        }
    }

    public MerkleTree() {
        root = new Node();
        depth = 0;
        leafToNode = new HashMap<Long, Node>();
    }

    public boolean insertLeaf(long leafvalue) {
        if (leafToNode.containsKey(leafvalue)) {
            return false;
        }

        Node curNode = root;

        Node leafNode = new Node();
        leafNode.hash = leafvalue;
        leafToNode.put(leafvalue, leafNode);

        if (root.leftChild == null) {
            assert root.rightChild == null;
            root.leftChild = leafNode;
            leafNode.parent = root;
            root.leftSubtreeSize = 1;
            root.updateHash();
            depth = 1;
        } else if (root.rightChild == null) {
            root.rightChild = leafNode;
            leafNode.parent = root;
            root.rightSubtreeSize = 1;
            root.updateHash();
        } else {
            // Repeatedly until reach a leaf.

            int curdepth = 0;
            while (curNode.leftSubtreeSize > 0) {
                if (curNode.leftSubtreeSize > curNode.rightSubtreeSize) {
                    curNode = curNode.rightChild;
                } else {
                    curNode = curNode.leftChild;
                }
                ++curdepth;
```

```
            }

            assert curdepth <= depth;

            // reached a leaf.
            assert curNode.rightSubtreeSize == 0;

            // Create a new internal node, and insert it in place of curNode.
            // curNode will become the new leftChild node, and the new leaf
will become
            // its new rightChild node.
            Node newInternal = new Node();
            newInternal.leftSubtreeSize = newInternal.rightSubtreeSize = 1;
            newInternal.parent = curNode.parent;
            if (curNode.parent.leftChild == curNode) {
                curNode.parent.leftChild = newInternal;
            } else {
                assert curNode.parent.rightChild == curNode;
                curNode.parent.rightChild = newInternal;
            }
            newInternal.leftChild = curNode;
            newInternal.rightChild = leafNode;
            leafNode.parent = curNode.parent = newInternal;
            newInternal.updateHash();


            if (curdepth == depth) {
                ++depth;
            }

            // Update the subtree counts and hashes on the path to the root.
            for (curNode = newInternal.parent; curNode != null; curNode =
curNode.parent) {
                curNode.leftSubtreeSize = curNode.leftChild.leftSubtreeSize +
curNode.leftChild.rightSubtreeSize;
                curNode.rightSubtreeSize = curNode.rightChild.leftSubtreeSize +
curNode.rightChild.rightSubtreeSize;
                curNode.updateHash();
            }
        }

        return true;
    }

    public void display() {
        Stack<Node> treeStack = new Stack<Node>();
        treeStack.push(root);
        int nBlanks = 32;
        boolean isRowEmpty = false;
        System.out.println(
                "......................................................");
        while (isRowEmpty == false) {
            Stack<Node> localStack = new Stack<Node>();
            isRowEmpty = true;

            for (int j = 0; j < nBlanks; j++) {
                System.out.print(' ');
```

```
            }

            while (treeStack.isEmpty() == false) {
                Node temp = (Node) treeStack.pop();
                if (temp != null) {
                    System.out.print(temp.toString());
                    localStack.push(temp.leftChild);
                    localStack.push(temp.rightChild);

                    if (temp.leftChild != null || temp.rightChild != null) {
                        isRowEmpty = false;
                    }
                } else {
                    System.out.print("--");
                    localStack.push(null);
                    localStack.push(null);
                }
                for (int j = 0; j < nBlanks * 2 - 2; j++) {
                    System.out.print(' ');
                }
            }
            System.out.println();
            nBlanks /= 2;
            while (localStack.isEmpty() == false) {
                treeStack.push(localStack.pop());
            }
        }
        System.out.println(
                "..................................................");
    }

    public static void main(String[] args) {
        MerkleTree mt = new MerkleTree();

        mt.insertLeaf(35);
        mt.insertLeaf(20);
        mt.insertLeaf(72);
        mt.insertLeaf(12);
        mt.insertLeaf(38);
        mt.insertLeaf(47);
        mt.insertLeaf(30);
        mt.insertLeaf(43);
        mt.insertLeaf(82);
        mt.insertLeaf(95);
        mt.insertLeaf(87);

        // display
        mt.display();
    }
}
```

10. Hinted handoff

- 处理流程



如果严格遵守 W + R > N 且 (W>=1)的策略，那么意味着系统至少要一个首选列表中的主机活着，写操作才能成功。为了提高可用性，在有些特定的情况下，如，碰巧对于某个 key, 其所有主要责任的所有节点都短暂失效，但其他的节点都正常，如 Data Center 间的网络分割。放松这一要求，能提高系统的可用性。
Hinted handoff 是用来处理系统短暂的失效的方法，当所有主要负责的 N 个节点均失效的情况下，它试图将信息存放在非主要责任节点的一个**特殊的位置**，并记下一个 hint, 其包含这次写操作的真正目标节点信息。当消息服务收到一个 Gossip 信号得知有新的节点从失败中恢复过来时，它查看该节点该节点有没有需要移交 (handoff)的数据。通过检查它是否是那个 hint 提到的节点，如果是，包含 hint 的那个节点将向它移交 (handoff) replica.

Hinted Handoff 实现为一个后台线程，并用一个队列来记录所有要移交的 hint 信息。

10.1 处理流程

1. 从队列中取出一个代处理的 hinted-handoff 信息
2. 进行移交工作的节点，通过上面提到的**特殊的位置**得 hint 相关信息
3. 根据上述 hint 中的目标接收节点信息， 通过消息中心发给目标接收的节点（即恢复的节点）
4. 接收节点收到消息后，应用这个 hint 到本地

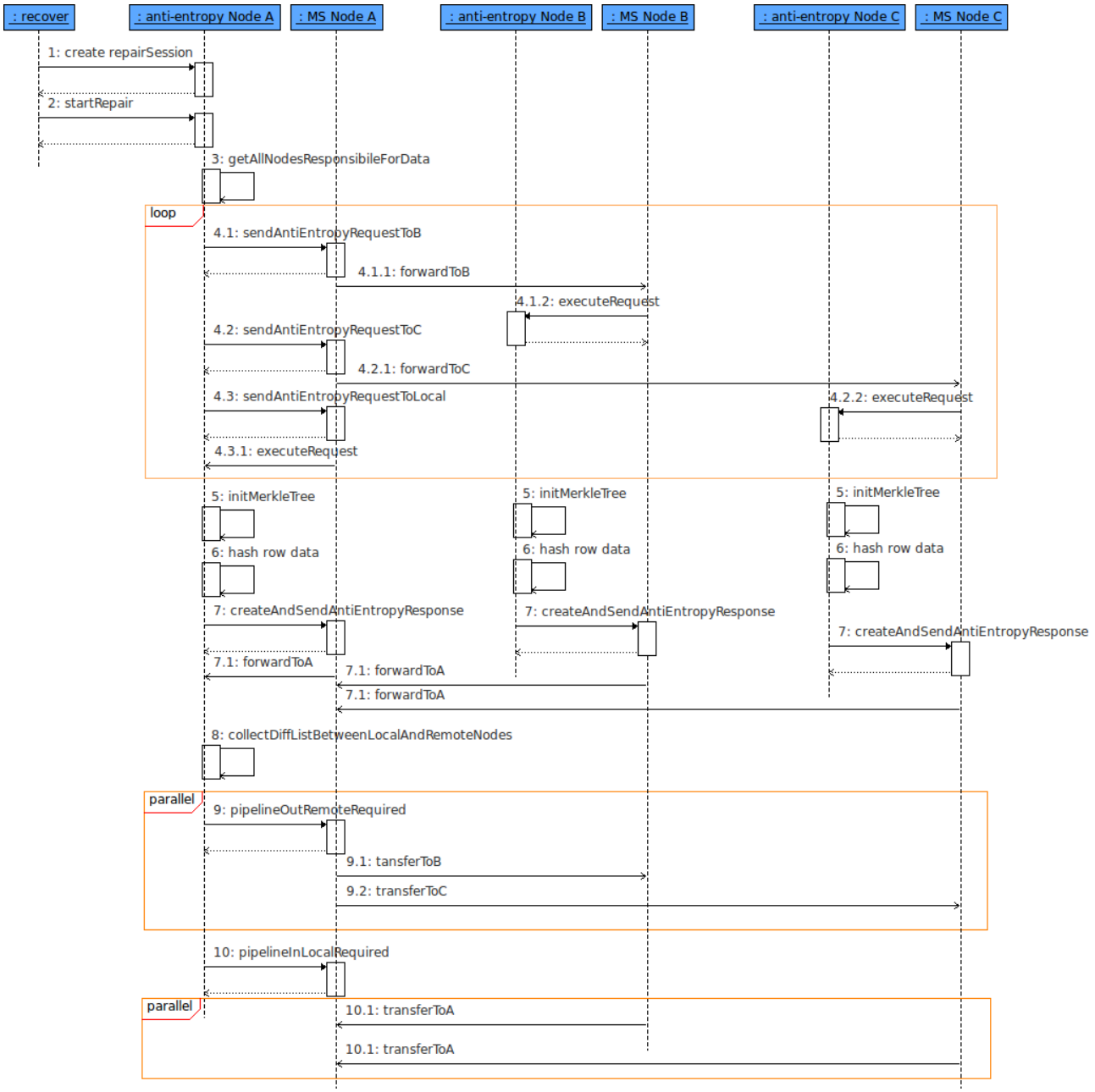
11. 反熵 (Anti-Entropy)

- 处理流程

上面提到的 Hinted handoff 应用在节点的故障率比较低且短暂失效的情况，但是，在一些情况下，如果失效的主机没有及时得到修复，即 Hinted handoff 信息没有被及时从 hinted handoff 主机送回它本应该的存在主机，系统是不可用的。
这时要用到 Anti-Entropy ， 该机制被用来保证在不同节点上的备份**(replica)** 都持有最新版本。由于涉及的处理很大，一般情况下，这种机制只用于永久性的错误恢复，而不用于普通的 read repair。如同 amazon dynamo 一样。
另外，为了将节点间的数据传输降到最低，在实际数据传输前，各节点交换的是自己那份数据的 message digest。实现中将采用 MerkleTree (其叶子节点存储的是数据文件，而非叶子节点存储的是其子节点的 Message Digest)。

11.1 处理流程

1. recover 过程从 Node A 创建一个 repair session
2. recover 过程从 Node A 开始 repair
3. Node A 从请求信息中得到数据 replica 所在的节点（例子中，除了 Node A 外，还有 Node B 和 Node C)
4. 迭代这个 replica 节点列表
   4.1. 通过本地的消息服务发送 anti-entropy 请求到 Node B
   4.1.1 Node B 接收并执行请求
   4.2. 通过本地的消息服务发送 anti-entropy 请求到 Node C
   4.2.1 Node C 接收并执行请求
   4.3. 通过本地的消息服务发送 anti-entropy 请求到 本地 Node A（自己）
   4.3.1 本地 Node A 接收并执行请求
5. 初始化 MerkleTree
6. 对每行数据计算其 hash 值并加入到 merkleTree
7. 各节点将自己的 MerkleTree 作为 anti-entropy response 送给本地的消息服务
   7.1 各节点的本地的消息服务将 anti-entropy response 返回给 Node A 的消息服务
8. Node A 收集各个 anti-entropy response 中的 MerkleTree 与本地的 MerkleTree (4.3.1 收到为本地的) 的不同 (diff)
9. 如果需要, 通过 pipelineOut 向 远程节点发送更加新的数据
10. 如要需要，通过 pipelineIn 向远程节点请求理加新的数据


: recover
: anti-entropy Node A
: MS Node A
: anti-entropy Node B
: MS Node B
: anti-entropy Node C
: MS Node C
1: create repairSession
2: startRepair
3: getAllNodesResponsibileForData
loop
4.1: sendAntiEntropyRequestToB
4.1.1: forwardToB
4.1.2: executeRequest
4.2: sendAntiEntropyRequestToC
4.2.1: forwardToC
4.2.2: executeRequest
4.3: sendAntiEntropyRequestToLocal
4.3.1: executeRequest
5: initMerkleTree
6: hash row data
7: createAndSendAntiEntropyResponse
7.1: forwardToA
8: collectDiffListBetweenLocalAndRemoteNodes
parallel
9: pipelineOutRemoteRequired
9.1: tansferToB
9.2: transferToC
10: pipelineInLocalRequired
parallel
10.1: transferToA

12. Read Repair

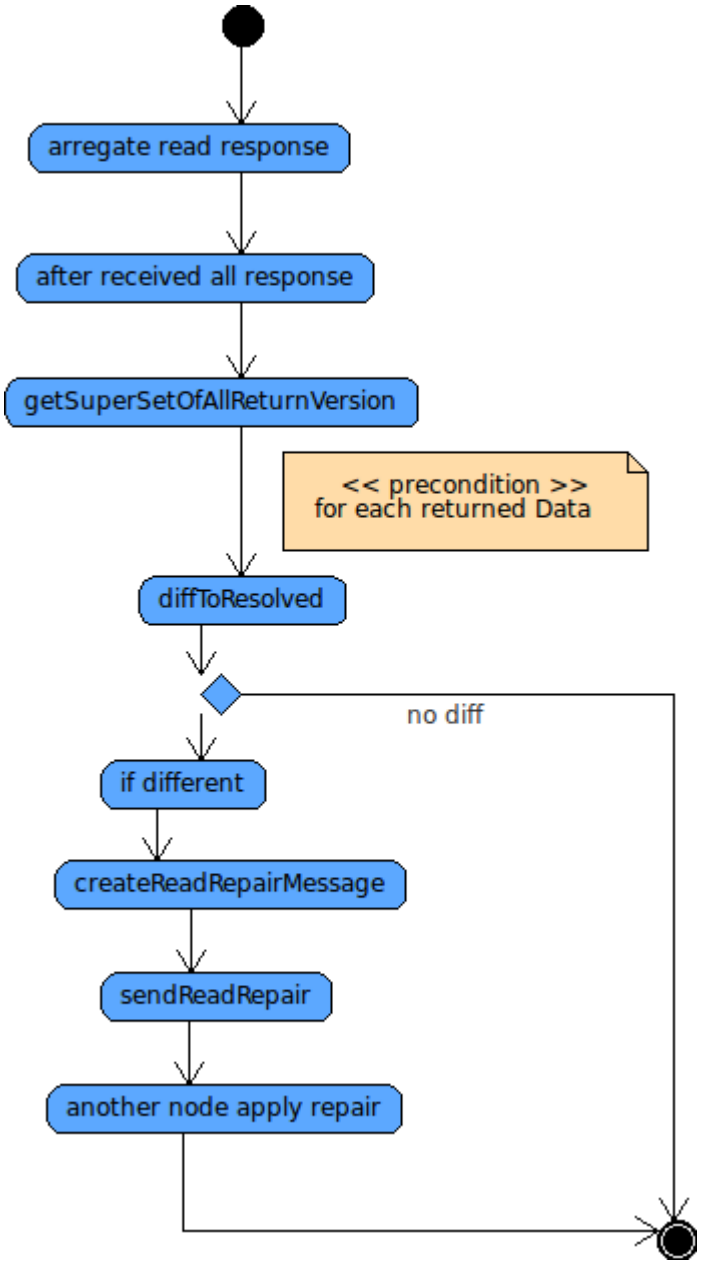
- 处理流程
- Read 比 Write 慢?

另一个保证一致性的机制是 read repair。如果读操作的 R > 1 时，在向主要的责任节点请求数据的同时，需要向“就近”的节点请求相同数据的“摘要”(digest)，在数据返回时，计算返回的数据的摘要并与其他节点返回的摘要对比，如果发现不匹配，便发起 read repair 过程。

12.1 处理流程

1. 将所有返回的数据累积在一个列表
2. 在所有 R 个 response 都返回时（通过 lock/Condition 控制）
3. 计算所有数据的超集，也就是计算出的版本将包含所有的更新
4. 迭代所有的返回数据
    4.1 计算其与超集的差异，如果存在差异，则
    4.2 构造 read repair 消息，并发送

5. 接收节点收到 read repair 命令，更新本地存储



12.2 Read 比 Write 慢?

由于这一特性，在 **R(2) + W(2) > N(3)** 的一致性配置情况下，通常，读操作比写操作慢。

另外，由于不象传统的关系型数据库使用 B-tree 数据结构；而是采用 BigTable 的设计：Sstable, memtable 数据结构。读操作往往需要 touch 多个 SSTable片段，如果没有命中 cache 的话。而 write 操作往往只是写到内存内的数据结构 memtable – 其周期地存储到磁盘上，因此没有随机磁盘访问。
注意： 对于 R(1) 的配置，read repair 往往在后台线程进行。

四. 通信通道

- Gossip
- The φ Accrual Failure Detector(累积失效检测器)
- 消息协议

13 Gossip

- 协议
- 处理流程

Gossip 是一种被用在分布式的非强一致性系统中用来同步各节点状态的方法。

因为在去中心化的集群环境里，各节点“实时”地洞察其他节点的重要信息是非常重要的。这消息包括：

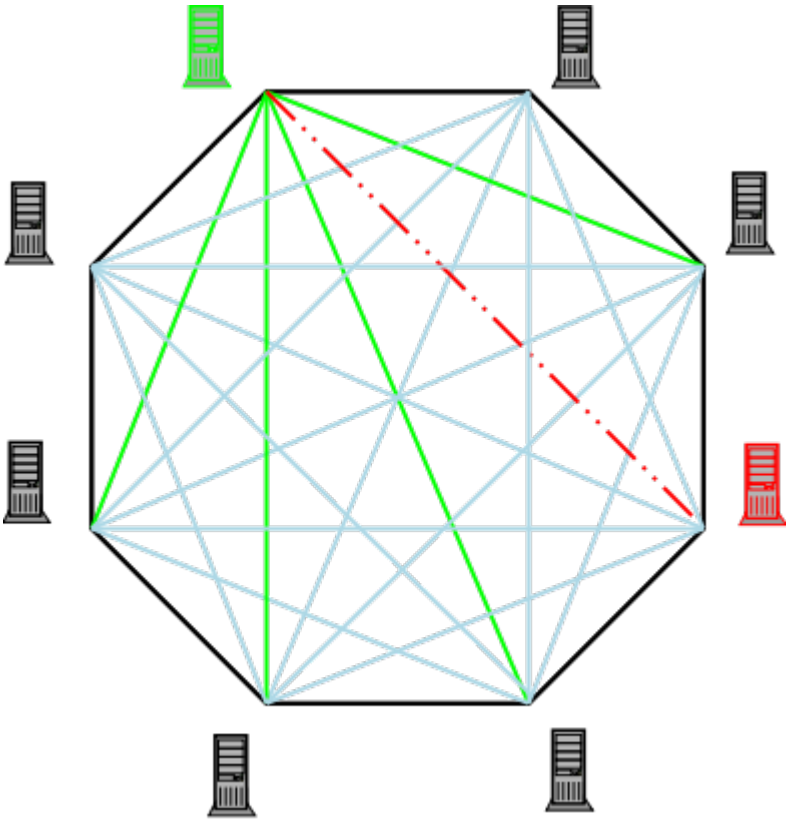
- 节点的心跳
- 节点的状态（失效检查／live/dead）
- 节点当前负载

Gossip 被设计成低 CPU 开销和低网络带宽占用。因此非常适合大型的 P2P 网络。

Gossip 周期地随机地选择一个节点并发起一轮 Gossip 会话，一个 Gossip 包括 3 个消息。比如 Node A 向 Node B 发起一个 Gossip：

- A → B GossipRequestMessage
- B → A GossipAckMessage
- A → B GossipResponseMessage

并且，基于以上来来回回的消息传送来探测失效与否。



13.1 协议

13.1.1 GossiperSummary

节点的概要信息

Ip	Node ip address
generation	代: 如果更新，代表一个大的状态改动，如 data model 更动
version	版本: 一个小的状态版本更新,比如节点 restart

13.1.2 GossipAppState

应用状态消息

HeartBeat	心跳: generation version

event	Status (状态事件): bootstrap/join/leaving/moving/removing/.. load （负载事件） datamodel (数据模型事件)

13.1.3 GossiperRequestMessage

由 gossiper 生成，发送给 gossipee
List<GossiperSummary> 包含发起节点的当前状态

13.1.4 GossipeeAckMessage

在收到 gossiper 的请求后，由 gossipee 生成，并发回给 gossiper

List<GossiperSummary> in
包含与进来的 GossiperRequestMessage 的差值，即 本地节点（gossipee）的信息已经过时，发起者(gossiper)有更新的信息，需要同步进来。

Map<Ip, GossipAppState> out
本地节点（gossipee）有更新的 GossipAppState 信息，需要同步回给 发起者(gossiper)

13.1.5 GossiperResponseMessage

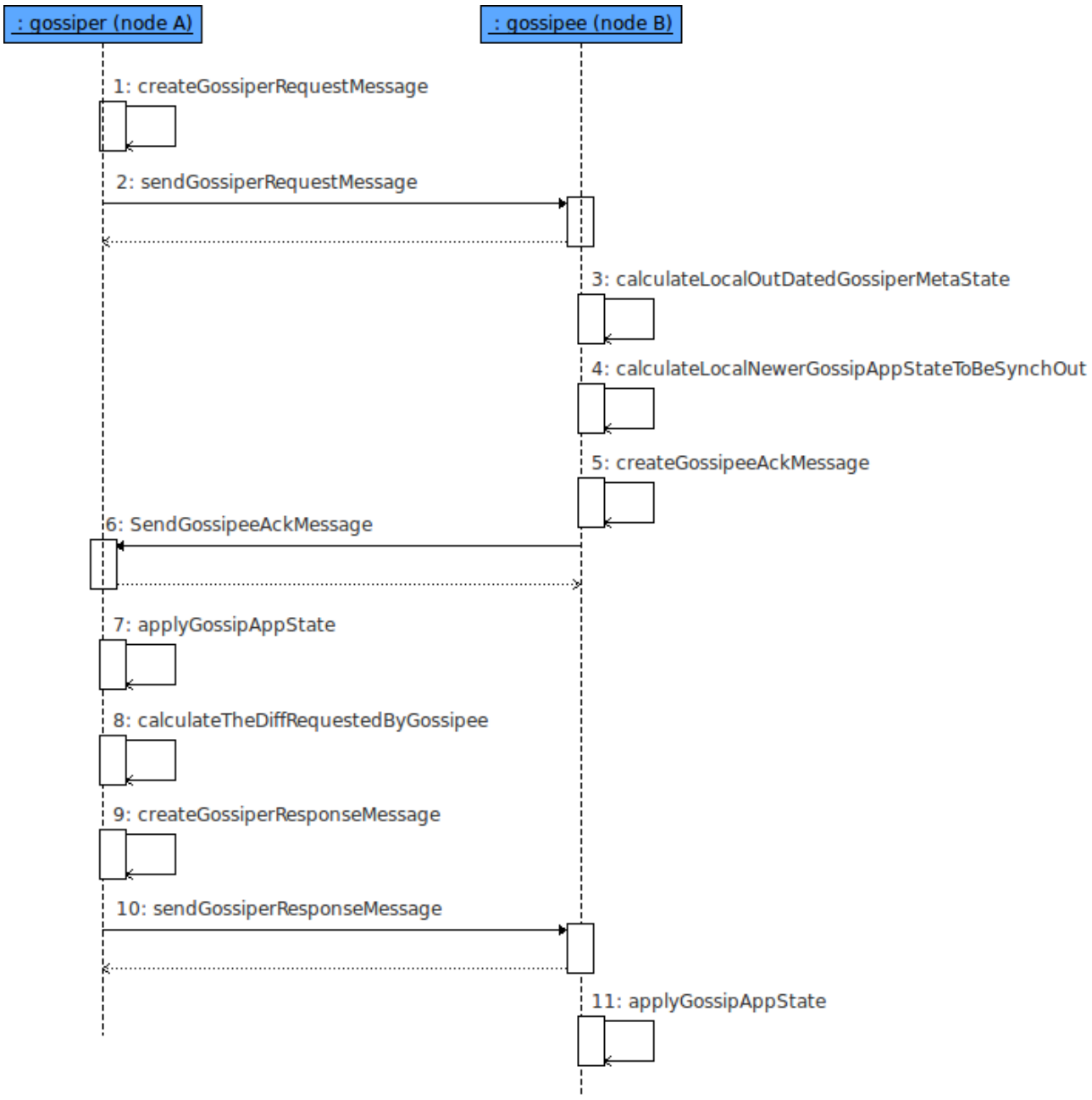
在收到 gossipee 的响应后，由 gossiper 生成

Map<Ip, GossipAppState> out
回传给 gossipee 请求的差异部分（即 gossipee 发现 gossiper 有更新的 GossipAppState 信息，要求 gossiper 回传）

13.2 处理流程

1. gossiper 基于自己本地缓存的系统拓扑信息，为其缓存的每个节点分别创建 GossiperSummary，再基于这个列表，创建出 GossiperRequestMessage
2. 向各个 live 节点发送 GossiperRequestMessage
3. 接收方 gossipee 解析请求消息，合计出自己过时了的状态信息 GossiperSummary
4. 接收方 gossipee 同时计算出自己比 gossiper (发送方）的更新的状态信息 GossipAppState
5. gossipee 基于上述计算，创建出 GossipeeAckMessage

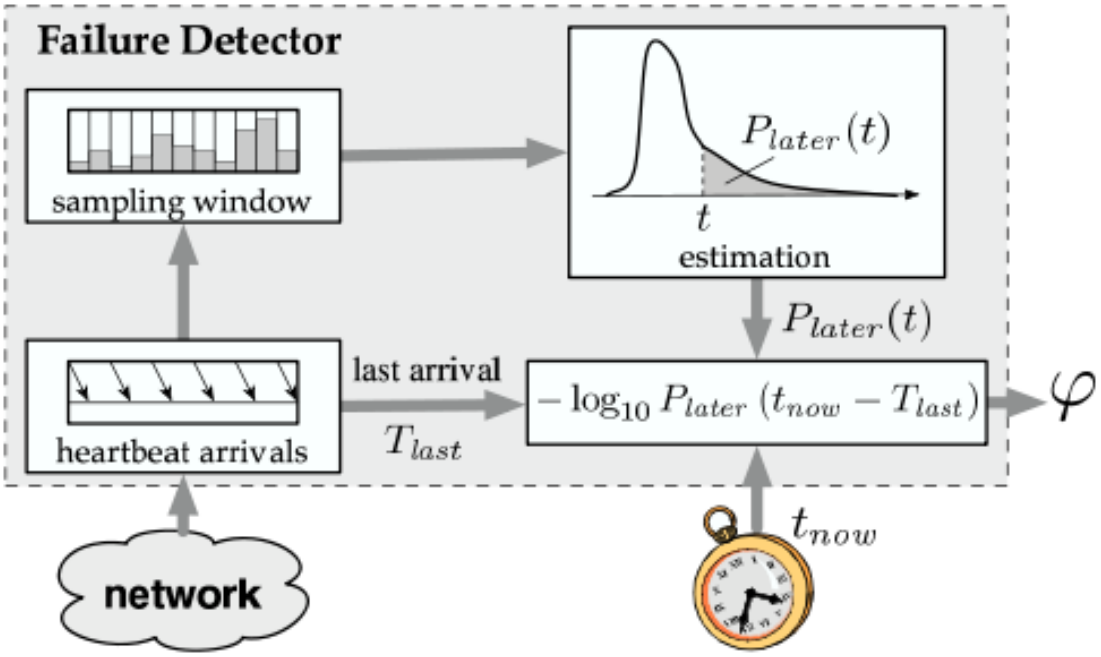
6. gossipee 向 发送方发送 GossipeeAckMessage 消息
7. 发送方基于 gossipee 送过来的 GossipAppState 更新自己状态
8. 发送方根据 gossipee 的请求，计算出 GossipAppState （此信息即是 gossipee 过时了的状态）
9. 发送方 基于上述计算，创建 GossiperResponseMessage 信息
10. 发送方向 gossipee 发送 GossiperResponseMessage 信息
11. gossipee 根据发送方回送过来的 GossipAppState 更新自己状态



14 The φ Accrual Failure Detector(累积失效检测器)

- φ 的含义
- Java 实现

14.1 The φ Accrual Failure Detector (累积失效检测器)



分布式环境中，对主机的心跳统计，根据以往心跳间隔的经验值，判断主机是否宕机。

1. 给定一个阀值 Φ
2. 在一定时间内，记录各个心跳间隔时间
3. 对心跳的间隔值求指数分布（Exponential distribution）概率：

   P = E ^ (-1 * (now - lastTimeStamp) / mean)

   其表示，自上次统计以来，心跳到达时间将超过 now - lastTimeStamp 的概率
4. 计算 **φ** = - $\log_{10}$ P

14.2 φ 的含义

当 φ >= Φ 时，我们断定 主机 宕机，出错的可能为：

Φ = 1, 1%

Φ = 2, 0.1%

Φ = 3, 0.01%

...

14.3 Java 实现

```
/*
 *  Public Domain
 *
 *  As explained at http://creativecommons.org/licenses/publicdomain
 *
 *  Written by pprun (quest.run@gmail.com)
 */
package phi;

import java.util.ArrayDeque;
import java.util.Iterator;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Java Demo of modified implementation of paper <a
href="http://citeseerx.ist.psu.edu/viewdoc/download?
doi=10.1.1.80.7427&rep=rep1&type=pdf">The φ Accrual Failure Detector</a>.
 *
 * <p>
 * The original paper suggests that the distribution is approximated by the
Gaussian distribution
 * This implementation uses the Exponential Distribution to be a better
approximation in our Network environment.
 * </P>
 *
 * @author <a href="mailto:quest.run@gmail.com">pprun</a>
 */
public class PhiAccrualFailureDetector {

    private static final int sampleWindowSize = 1000;
    private static int phiSuspectThreshold = 8;
    private SamplingWindow simpleingWindow = new
SamplingWindow(sampleWindowSize);

    public PhiAccrualFailureDetector() {
    }

    public void addSample() {
        simpleingWindow.add(System.currentTimeMillis());
    }

    public void addSample(double sample) {
        simpleingWindow.add(sample);
    }

    public void interpret() {
        double phi = simpleingWindow.phi(System.currentTimeMillis());
        System.out.println("PHI = " + phi);

        if (phi > phiSuspectThreshold) {
            System.out.println("We are assuming the moniored machine is
down!");
        } else {
            System.out.println("We are assuming the moniored machine is still
```

```
running!");
        }
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        PhiAccrualFailureDetector pafd = new PhiAccrualFailureDetector();

        // first try with phi < phiSuspectThreshold
        for (int i = 0; i < 10; i++) {
            pafd.addSample();

            try {
                Thread.sleep(10L);
            } catch (InterruptedException ex) {
                // no op
            }
        }
        try {
            Thread.sleep(500L);
        } catch (InterruptedException ex) {
            // no op
        }

        System.out.println(pafd.simpleingWindow.toString());
        pafd.interpret();

        // second try result phi > phiSuspectThreshold
        for (int i = 0; i < 10; i++) {
            pafd.addSample();

            try {
                Thread.sleep(10L);
            } catch (InterruptedException ex) {
                // no op
            }
        }
        try {
            Thread.sleep(1500L);
        } catch (InterruptedException ex) {
            // no op
        }

        System.out.println(pafd.simpleingWindow.toString());
        pafd.interpret();
    }

    /**
     * Sampling heartbeat arrivals: The monitored process (p in our model) adds
a sequence number to
     * each heartbeat message. The monitoring process (q in our model) stores
heartbeat arrival times into a
     * sampling window of fixed size WS . Whenever a new heartbeat arrives, its
arrival time is stored into the
     * window, and the data regarding the oldest heartbeat is deleted from the
```

```
window
     */
    static class SamplingWindow {

        private final Lock lock = new ReentrantLock();
        private double lastTimeStamp = 0L;
        private StatisticDeque arrivalIntervals;

        SamplingWindow(int size) {
            arrivalIntervals = new StatisticDeque(size);
        }

        void add(double value) {
            lock.lock();
            try {
                double interval;
                if (lastTimeStamp > 0L) {
                    interval = (value - lastTimeStamp);
                } else {
                    interval = 1000 / 2;
                }
                lastTimeStamp = value;
                arrivalIntervals.add(interval);
            } finally {
                lock.unlock();
            }
        }

        double sum() {
            lock.lock();
            try {
                return arrivalIntervals.sum();
            } finally {
                lock.unlock();
            }
        }

        double sumOfDeviations() {
            lock.lock();
            try {
                return arrivalIntervals.sumOfDeviations();
            } finally {
                lock.unlock();
            }
        }

        double mean() {
            lock.lock();
            try {
                return arrivalIntervals.mean();
            } finally {
                lock.unlock();
            }
        }

        double variance() {
            lock.lock();
```

```
            try {
                return arrivalIntervals.variance();
            } finally {
                lock.unlock();
            }
        }

        double stdev() {
            lock.lock();
            try {
                return arrivalIntervals.stdev();
            } finally {
                lock.unlock();
            }
        }

        void clear() {
            lock.lock();
            try {
                arrivalIntervals.clear();
            } finally {
                lock.unlock();
            }
        }

        /**
         *
         * p = E ^ (-1 * (tnow - lastTimeStamp) / mean)
         */
        double p(double t) {
            double mean = mean();
            double exponent = (-1) * (t) / mean;
            return Math.pow(Math.E, exponent);
        }

        double phi(long tnow) {
            int size = arrivalIntervals.size();
            double log = 0d;
            if (size > 0) {
                double t = tnow - lastTimeStamp;
                double probability = p(t);
                log = (-1) * Math.log10(probability);
            }
            return log;
        }

        @Override
        public String toString() {
            StringBuilder s = new StringBuilder();
            for (Iterator<Double> it = arrivalIntervals.iterator();
it.hasNext();) {
                s.append(it.next()).append(" ");
            }

            return s.toString();
        }
    }
```

```
/**
 * Queue with statistic method based on the data in the queue.
 */
static class StatisticDeque implements Iterable<Double> {

    private final int size;
    protected final ArrayDeque<Double> queue;

    public StatisticDeque(int size) {
        this.size = size;
        queue = new ArrayDeque<Double>(size);
    }

    public Iterator<Double> iterator() {
        return queue.iterator();
    }

    public int size() {
        return queue.size();
    }

    public void clear() {
        queue.clear();
    }

    public void add(double o) {
        if (size == queue.size()) {
            queue.remove();
        }
        queue.add(o);
    }

    public double sum() {
        double sum = 0D;
        for (Double interval : this) {
            sum += interval;
        }
        return sum;
    }

    public double sumOfDeviations() {
        double sumOfDeviations = 0D;
        double mean = mean();

        for (Double interval : this) {
            double d = interval - mean;
            sumOfDeviations += d * d;
        }

        return sumOfDeviations;
    }

    public double mean() {
        return sum() / size();
    }
```

```
        public double variance() {
            return sumOfDeviations() / size();
        }

        public double stdev() {
            return Math.sqrt(variance());
        }
    }
}
```

Run Result

```
500.0 11.0 10.0 10.0 10.0 10.0 10.0 10.0 11.0 10.0
PHI = 3.7487243285905687
We are assuming the moniored machine is still running!
500.0 11.0 10.0 10.0 10.0 10.0 10.0 10.0 11.0 10.0 512.0 10.0
11.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0
PHI = 10.98274413649897
We are assuming the moniored machine is down!
BUILD SUCCESSFUL (total time: 2 seconds)
```

15 异步消息机制

- 消息协议
- 处理流程
- PipeLine

从效率和可伸缩性的角度考虑，系统各节点相间的相互通信采用异步通信模式。
虽然，也许引入现有的消息组件实现，如 大部分JMS 的开源实现，也可以完成这一功能。但是为了不依赖第三方实现，毕竟，各个实现会有不同的限制，自包含的实现可以提高系统的可靠性，以及简化后续的性能调整。

15.1 消息协议

15.1.1 MessageHeader

ipAddress	消息的发起者的ＩＰ地址
消息命令（消息类型）	指定消息的类型

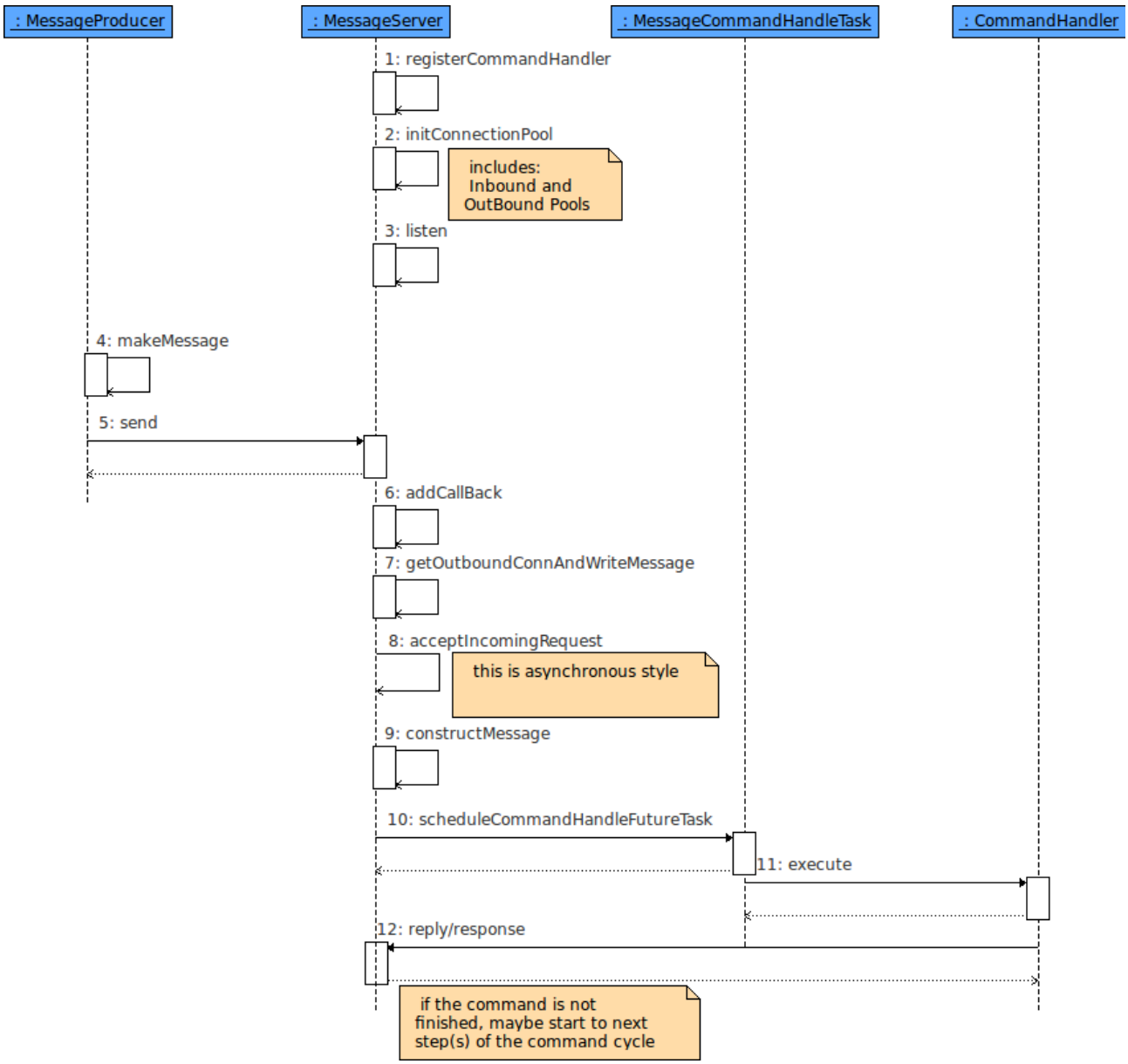
15.1.2 MessageBody

Header	MessageHeader
Version	消息的版本(用于不同的 node 运行不同的版本，只允许兼容性的版本的节点相互发送消息)
Body	消息体

15.1.3 消息命令

命令	描述
KeySpace Mutation	KeySpace 定义改动
Read	读数据
Write	写数据
Read Repair	读修复
Gossip Request	Gossiper 请求
Gossip Ack	Gossipee 确认
Gossip Response	Gossiper 响应
Anti-Entropy Request	反熵请求
Anti-Entropy Request	反熵响应
Pipleline Request	Pipeline 请求
Pipeline response	Pipeline 响应

15.2 处理流程



15.3 PipeLine

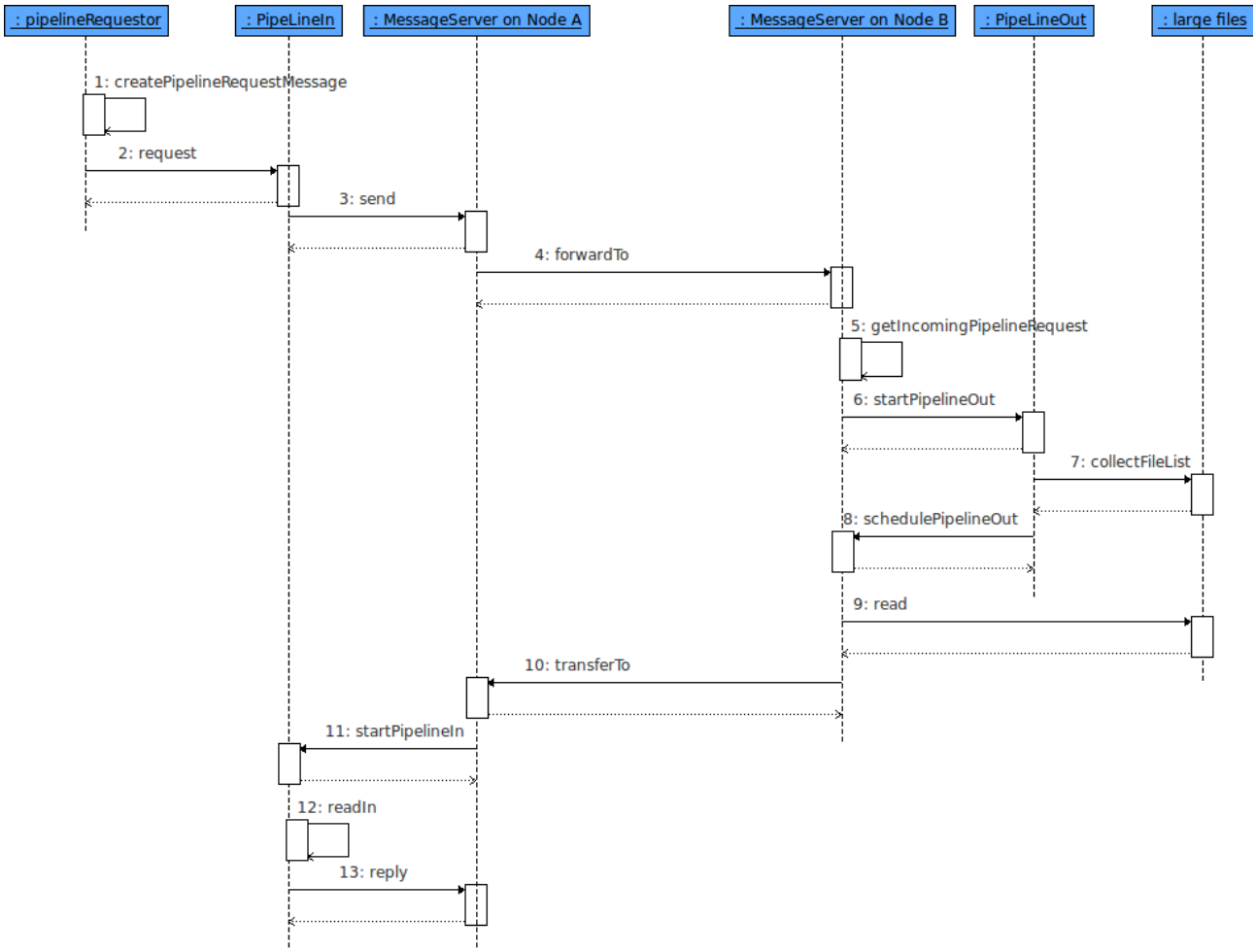
以上基本流程没有提到对数据流的处理。通常有基于 Basic IO 和基于管道 (PipeLine) 两种模式。管道在传输大文件时，避免 文件到缓冲拷贝或 Memory Map 的同时(在实现的底层，可以节省从 IO, 内核, 用户层间的数据拷贝)

15.3.1 举例

Node A 要向 Node B 发起文件传输请求

- pipelineIn 对 NodeA 代表进来的请求 (upStream)
- MessageServer 消息中心
- pipelineOut 对 Node B 代表出去的响应或反馈 (downStream)

1. Node A 生成 pipeline 请求消息
2. Node A 向 pipeline(In 从 A 看来) 发送请求
3. pipeline 各 MessageServer 递送请求
4. Message Server 查直目的地址，并将请求转发给 Node B
5. Node B 进来一个 pipeline 消息请求
6. Node B 将请求交给 pipeline (out 从 B 看来)
7. pipeline 收集文件列表
8. pipeline 调试一个 pipelineout task 给 message server
9. message server 读取文件
10. message server 将文件直接写出到目的地 Node A message server
11. Node A 发现进来一个 pipeline 写请求，调用 pipleLine 开始 pipeline
12. pipeline 读取内容
13. 直到完成，返回 message server 响应（也许是完全成功，在出错的情况下，也许是 retry)


: pipelineRequestor
: PipeLineIn
: MessageServer on Node A
: MessageServer on Node B
: PipeLineOut
: large files
1: createPipelineRequestMessage
2: request
3: send
4: forwardTo
5: getIncomingPipelineRequest
6: startPipelineOut
7: collectFileList
8: schedulePipelineOut
9: read
10: transferTo
11: startPipelineIn
12: readIn
13: reply

五. Interface

- API
- MapReduce
- 计算服务
- 负载均衡

接口是外部应用与存储系统的通信的通道，分为客户端接口和管理接口:

- Client API　　提供给使用存储系统的应用的 API，对存储的数据进行 CRUD 操作
- Admin API　　提供给存储系统管理工具的 API，用于进行系统管理

对于公共访问的存储系统，通常将接口封装成语言中立的接口，如 REST。就象 Amazon S3 一样。但是，象 Voldemort, Hbase, Cassandra 的实现都采用支持多种语言的 Service/Serialization 框架: Apache Thrift / Avro 。

16. API

- Client API
- Admin API

Table. REST API Cheat Sheet

URI	HTTP Method	Description
/yijing/keyspace/{keyspace}/columnfamily/{columnfamily}/column/{column}/consistency/{consistency}	GET	查询存放在指定的路径 keyspace/columnfamily 下的列，使用 consistency 级别
/yijing/keyspace/{keyspace}/columnfamily/{columnfamily}/consistency/{consistency}	POST	将一个新的 column 存放在指定的 keyspace/columnfamily 路径下 column 的信息在请求 body 中
/yijing/keyspace/{keyspace}/columnfamily/{columnfamily}/column/{column}/consistency/{consistency}	UPDATE	更新指定的 keyspace/columnfamily 路径下的 column. column 的信息在请求 body 中 keys
/yijing/keyspace/{keyspace}/columnfamily/{columnfamily}/column/{column}/consistency/{consistency}	DELETE	删除指定的 keyspace/columnfamily 路径下的 column
/yijing/keyspace/{keyspace}/columnfamily/{columnfamily}/column/{column}/predicate/{predicate}/consistency/{consistency}	GET	使用“谓词” predicate 查询存放在指定的路径 keyspace/columnfamily 下的列，使用 consistency 级别. 注意，实现需要对 ' predicate' 进行检查，确保“恶意”的操作。

16.1 Client API

API	Description
Column getColumn(keyspace, rowkey, columnfamily, column, consistency)	• Keyspace column 所在的 columnFamily 存放的 keyspace • Rowkey column 所在的 columnFamily 对应的 Row • columnfamily column 所在的 columnFamily • column 要查询的列 • consistency 指定客户期待的一致性极别
Void insertColumn(keyspace, rowkey, columnfamily, column, consistency)	
Void updateColumn(keyspace, rowkey, columnfamily, column, consistency)	
Void removeColumn(keyspace, rowkey, columnfmaily, column, timestamp, consistency)	
Map<rowKey, ColumnList> getColumnList(keyspace, rowkey, columnfamily, predicate, consistency)	1. Predicate 包含查询的语义： • columList-待查询的 column 列表 • start – 开始的 column 名 • finish – 结束的 column 名 • count – 总共要查询的列数 • reversed – 结果是否为 order by DESC 2. consistency 指定客户期待的一致性极别

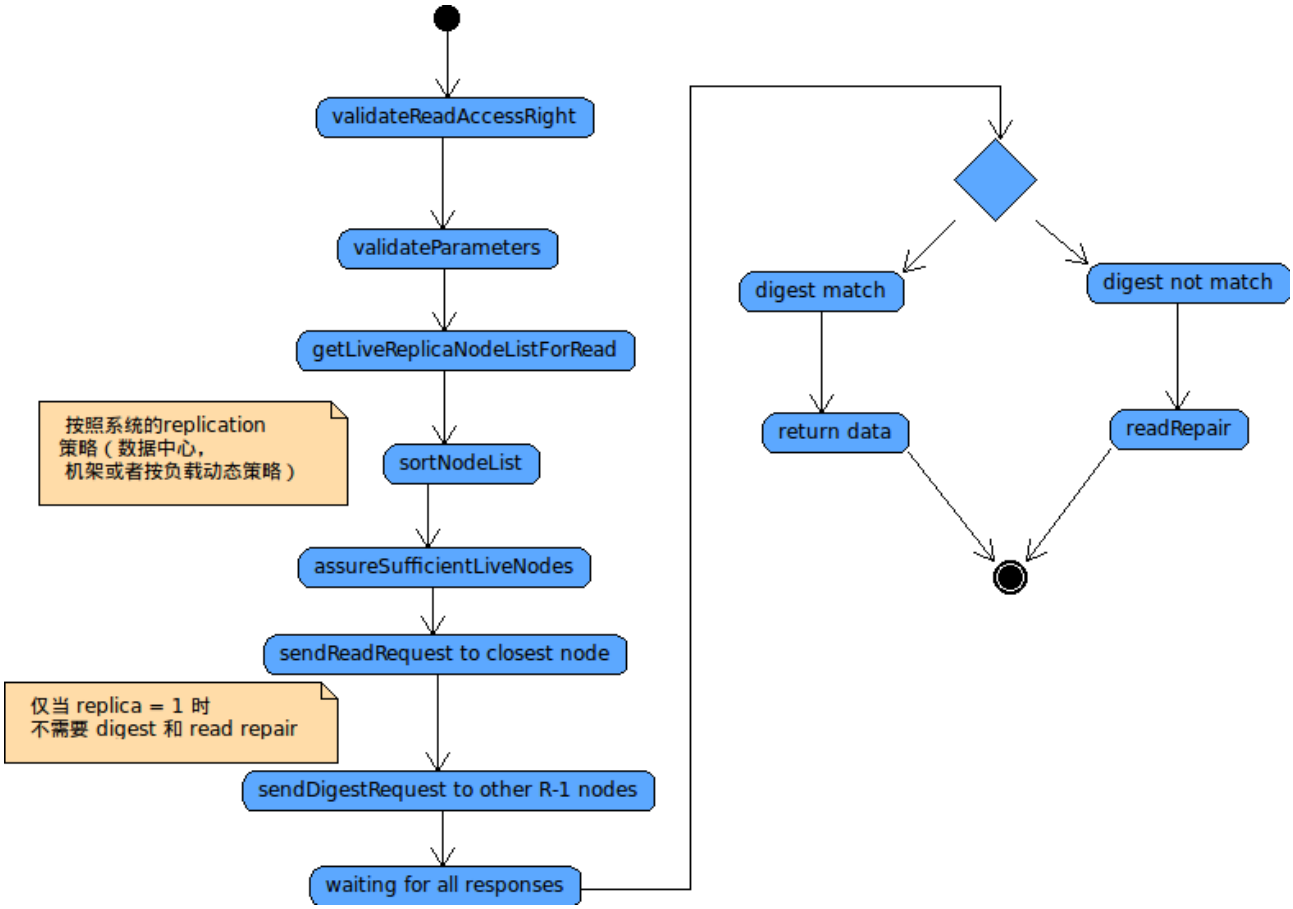
16.1.1 getColumn

查询 Column 涉及到两部分的操作：

- 发送 GET column command 请求
- 处理 GET column command 请求

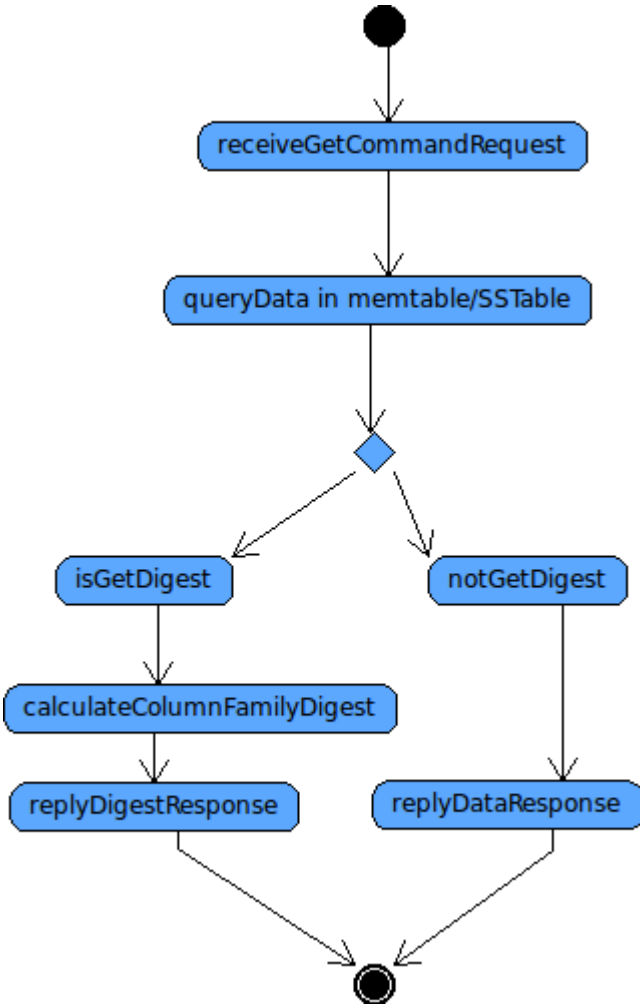
发送 GET column command 命令

1. 校验当前用户是否有对该 columnfamily 有 read 访问权限
2. 校验输入参数
3. 得到所有对应该读操作现在可用的首先 Node 列表
4. 将 node 列表按“就近”配置原则排序
5. 确保当前活动节点数目满足 consistency 要求
6. 发送一个 读请求到最近的 node
7. 如果 R > 1，则 发送 Digest 请求到排序后的较低其他 node (按照从近到远，直到满足 R 为止)
8. 等待所有的 response



处理 Get Column Command **请求**

1. 收到 Get Column Command 命令请求
2. 查询该 column 对应的数据（先从 memtable，如果没有找到，则从 SSTable 中找）
3. 如果为非 Digest 请求，返回找到的数据
4. 否则，计算数据的 digest
5. 返回该 digest



16.1.2 insertColumn

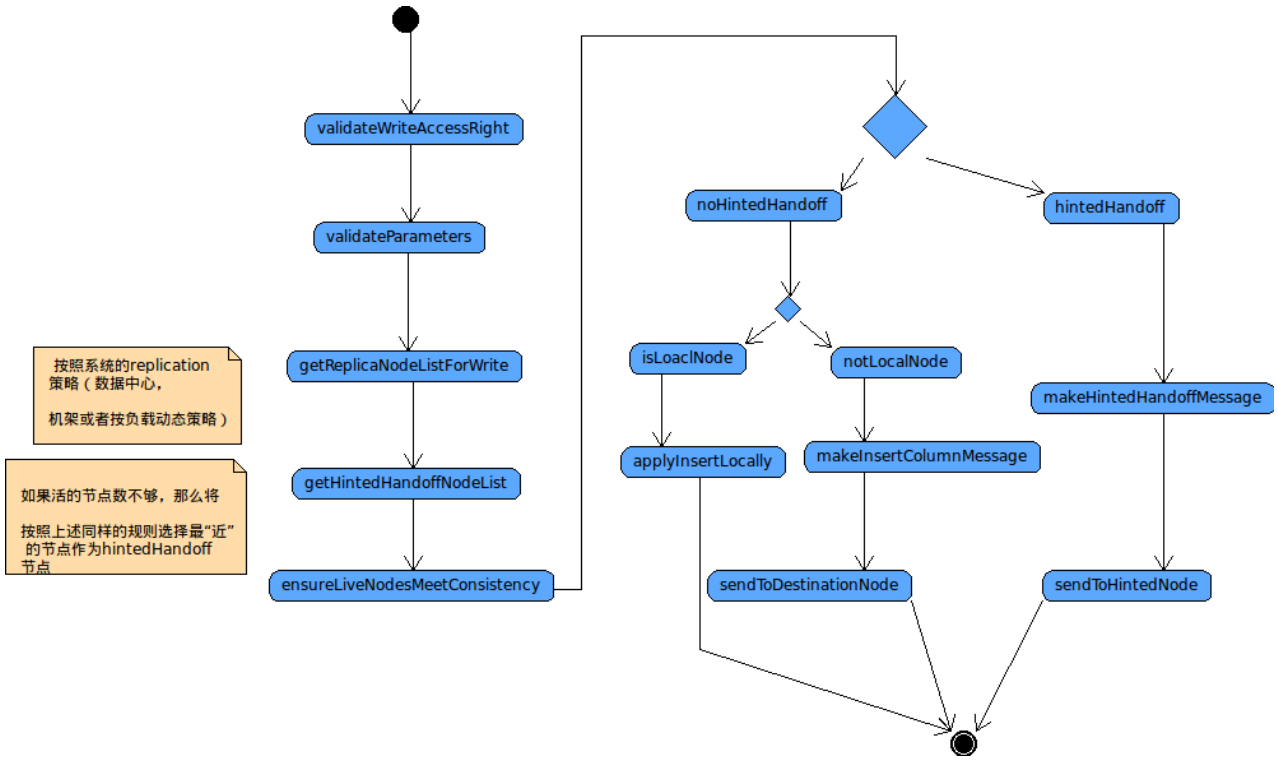
insert Column 涉及到两部分的操作:

- 发送 INSERT column command 请求
- 处理 INSERT column command 请求

发送 INSERT column command 请求

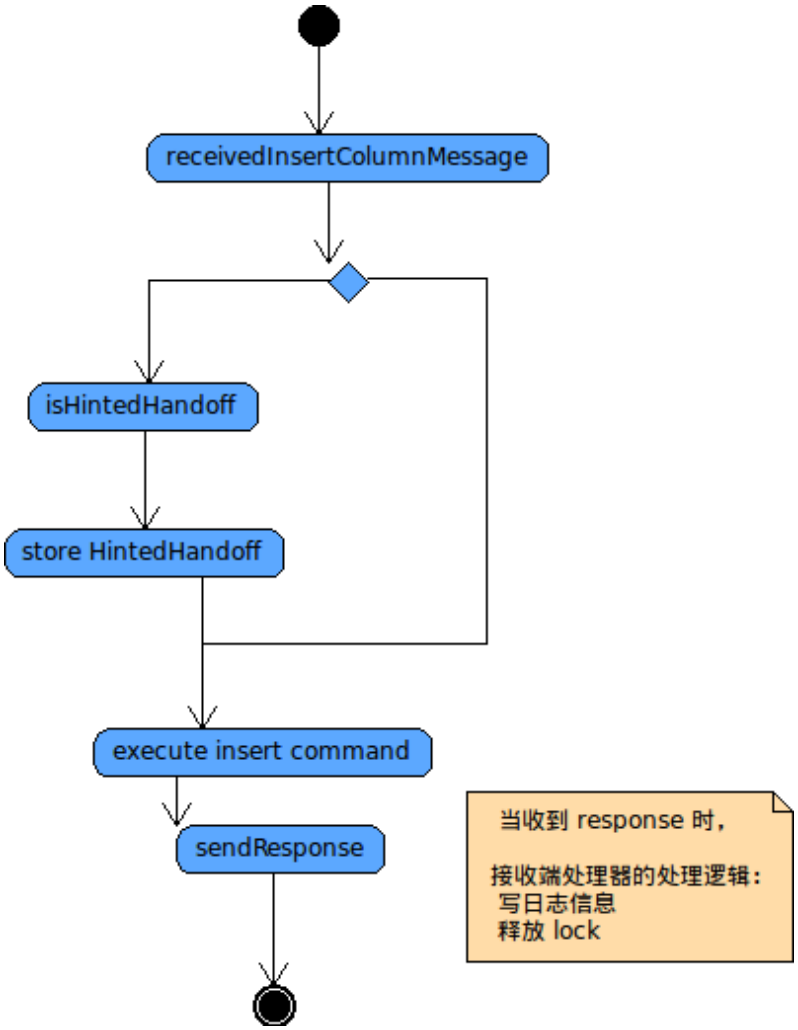
1. 校验当前用户是否对该 columnfamily 有 write 权限
2. 校验输入参数
3. 得到所有对应该写操作现在可用的首先 Node 列表
4. 计算 hintedHandoff 节点列表（如果当前有节点不可用，则会生成该列表）
5. 确保当前活动节点数满足 consistency 要求
6. 迭代 节点列表
   1. 如果需要处理 hinted handoff
      - 生成 hintedHandoff 消息
      - 发送 hintedHandoff *消息*
   2. 否则,判断 write 目标节点是否为本地节点

2.1 如果是，在本地执行写命令
2.2 否则
- 生成 insert column command 消息
- 发送 insert column command 消息



处理 INSERT column command 请求

1. 接收到 insert column command 请求
2. 是否为一个 hintedHandoff 请求
3. 如果，是，则在存储 hinted handoff 相关的数据
4. 执行 insert column 命令
5. 生成 response 并返回



16.1.3 updateColumn

update 的处理流程与 insert 类似，只是命令为 update column ，因此执行的逻辑不一样

16.1.4 removeColumn

update 的处理流程与 insert 类似，只是命令为 remove column ，因此执行的逻辑不一样

16.1.5 getColumnList

此为接口可以一次返回多个 column

16.2 Admin API

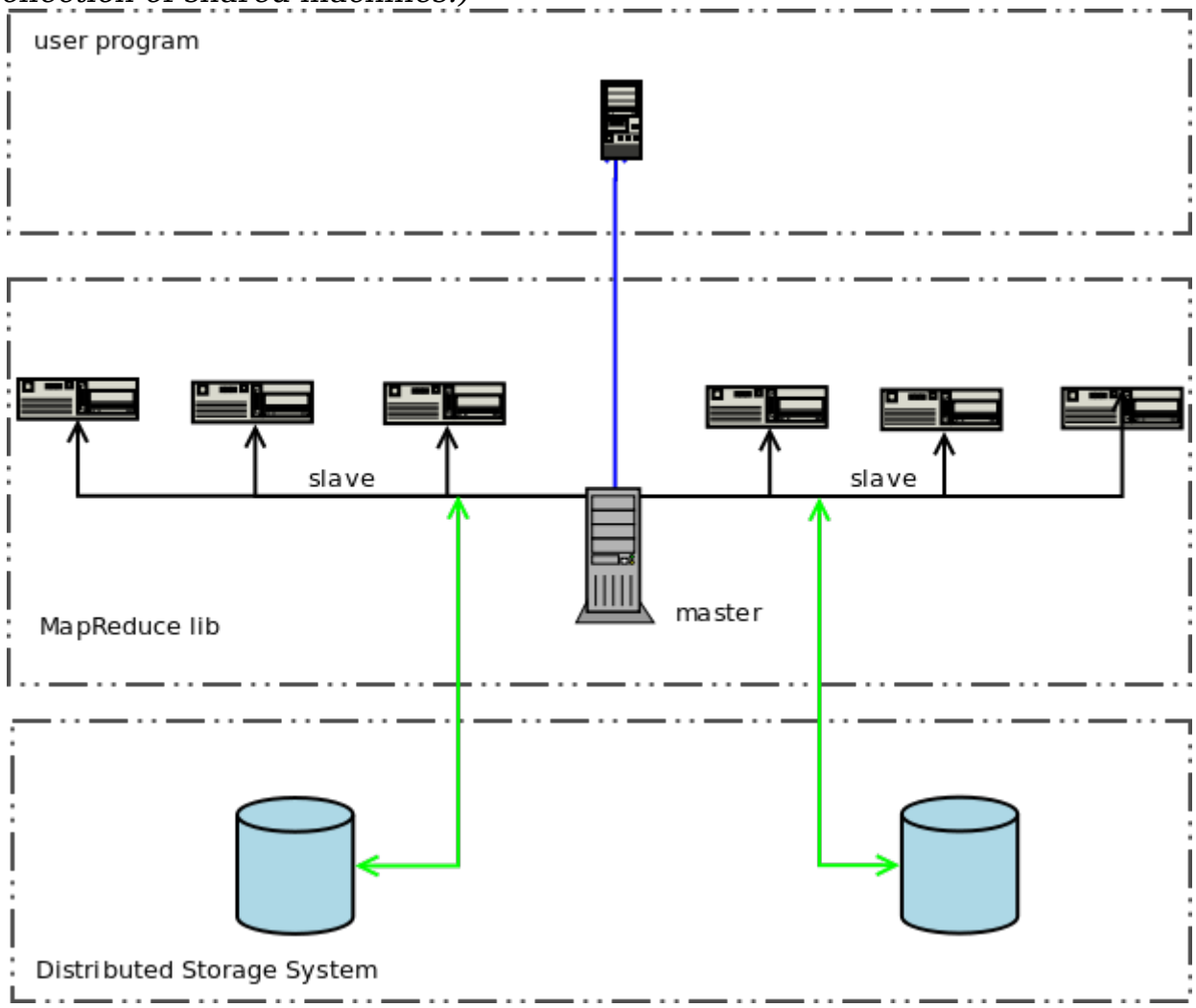
API	Description
Add/delete/update/get keyspace	对 keyspace 进行增删改查
Add/delete/update/get columnfamily	对 columnfamily 进行增删改查
GetTokenMap	得到所有 live node 的 Token:INetAddress 的映射
GetNaturalNodeList(rowKey)	得到负责给定的 key 的首选的 N (N 为 副本因子 replica-factor) 个节点列表
Showkeyspace	显示 keyspace 的所有系统元数据: cluster name version token ring partitioner
Login/logout	登录（认证／授权）接口

17. MapReduce

- Apache Hadoop
- MapReduce 处理流程
- MapReduce 数据流状态
- 基于 Hadoop 的 MapReduce 支持

从众多技术站点和博客对流行的 NoSQL 数据存储系统的评估可以发现，对 MapReduce 的原生支持是其中考察的环节之一。因为 MapReduce 的提出就是用来处理巨大的数据集的，而 NoSQL 非常适合 MapReduce 的数据源头和数据目的地。

之所以把 MapReduce 规为 分布式存储的外部接口，是基于 MapReduce 的分层体系设计架构：
MapReduce 封装了并行处理、容错处理、数据本地化优化、负载均衡等等技术难点的细节。
从 google 原始论文可得知，MapReduce 依赖于一个 Cluster 来管理分发，运行和协调任务.
(The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines.)



17.1 Apache Hadoop

Apache Hadoop 是 MapReduce 的开源实现 Java，事实上的参考实现。 而且似乎严格的照搬 Google 的架构: HDFS(GFS), ZooKeeper(Chubby)，Master/Slave, ...

17.1.1 MapReduce

(Overview)
MapReduce 框架由一个单一的 master JobTracker 和集群中每节点一个 TaskTracker 组成。Master 负责调度工作组件的任务到 Slave 上去，对它们进行监控，以及在失败时重新执行。Slave 按照 Master 的指示执行任务。
(The Map/Reduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.)

17.1.2 HDFS

(NameNode and DataNodes)
HDFS 基于 Google GFS 设计理念，一个 master/slave 的体系架构。一个 HDFS 集群是由一个单一的 NameNode, 一个 master server 管理文件系统的命令空间以及控制客户对文件的访问，和集群中每节点一个 DataNode 组成。DataNode 管理该节点的存储器。
(*HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on*)

17.1.3 Hbase

(Hbase Replication)
(The Apache HBase Book)
Hbase 是基于 Google BigTable 原型。其充当上图中的 Distributed Storage System. 其按照 BigTable 的设计理念，架构在一个分布式文件系统(HDFS)之上。
Hbase 需要一个 HDFS 文件系统实现并且依赖于一个运行着的 ZooKeeper (Google Chubby 的 java 实现)集群。HBase 默认管理着一个 ZooKeeper 集群。
(HBase requires an instance of the **Hadoop Distributed File System** (HDFS) and depends on a running ZooKeeper cluster, HBase by default manages a ZooKeeper "cluster" for you。)

17.1.4 基于 Hbase 的 MapReduce

从以上引用文字不难看出，Hadoop 的基础架构都是 Single Master/Multiple Slaves 的架构，该体系有与生俱来的单点失效（Single point of failure）的风险。因此在架设基于 Hadoop 的集群环境时，需要考虑到这一点。可以参考 Hadoop 单点失效（Single Point of Failure）。

虽然 Hadoop 可以抛开 Hbase 直接运行在 HDFS 之上，但在生产环境，业务数据与文件之间并不存太强的关系，例如 java 对象和数据库用来存储关系表的文件一样，因此，一个分布式存储可以给应用提供业务模型的支持。虽然 K/V 存储器业务模型比较单一。

基于 Hbase 的 MapReduce 架构是一个复杂的分层集群，自顶向下依次为：

- MapReduce 集群
- Hbase 集群
- HDFS 集群 （往往与 Hbase 共用）
- ZooKeeper 集群（虽然，HBase 默认携带了一个内嵌的）

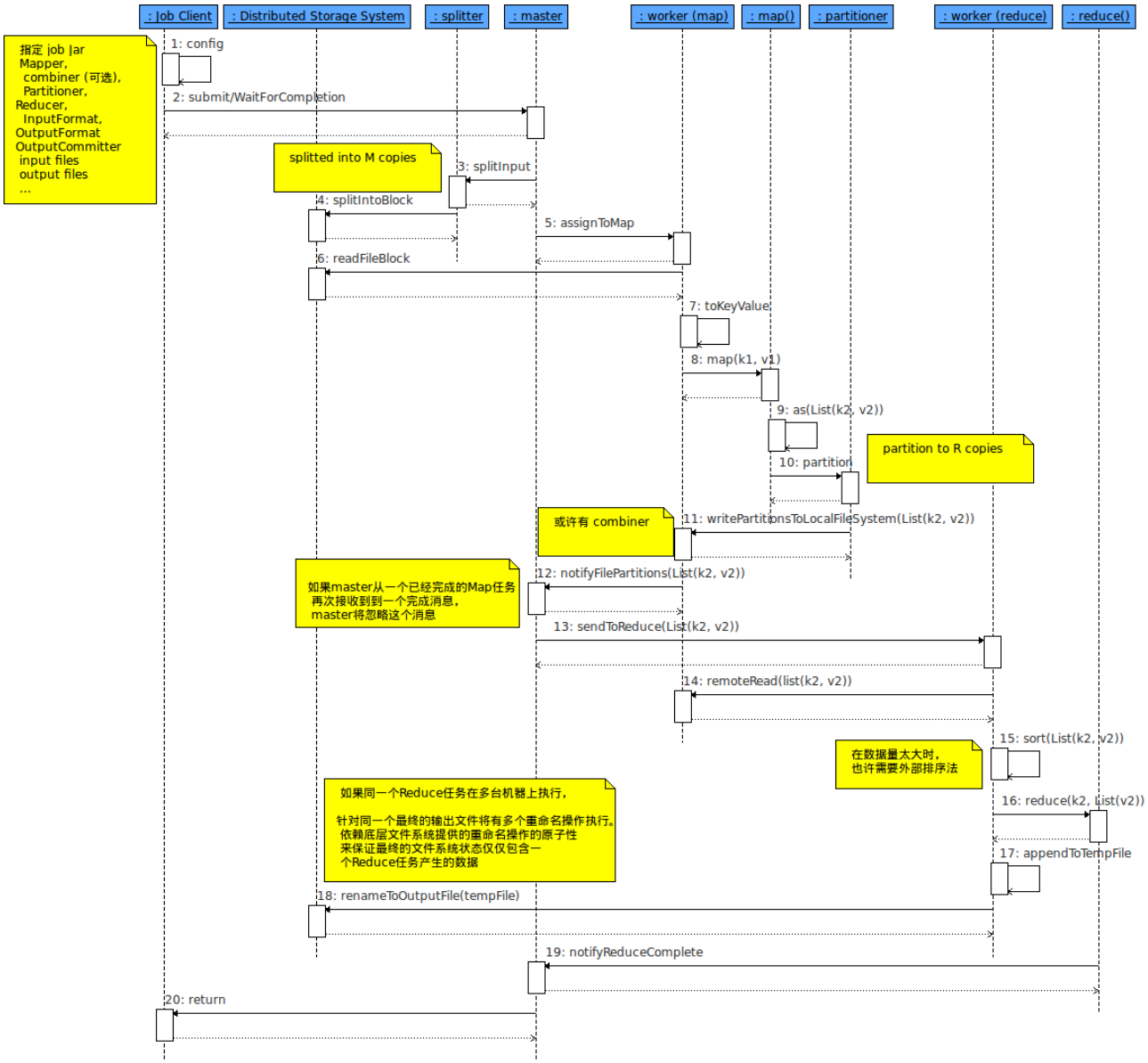
采用这种架构的目的是仅仅是为了让 HBase 成为 MapReduce job 的数据源和存储目的地。你是不是被这一堆”家伙”吓倒了？
反正我是吓到了！我们是不是把问题搞复杂了？为了运行一个 MapReduce Job 需要 4 个集群?

17.2 MapReduce 处理流程

看一下 Google MapReduce 论文中描述的执行流程先:

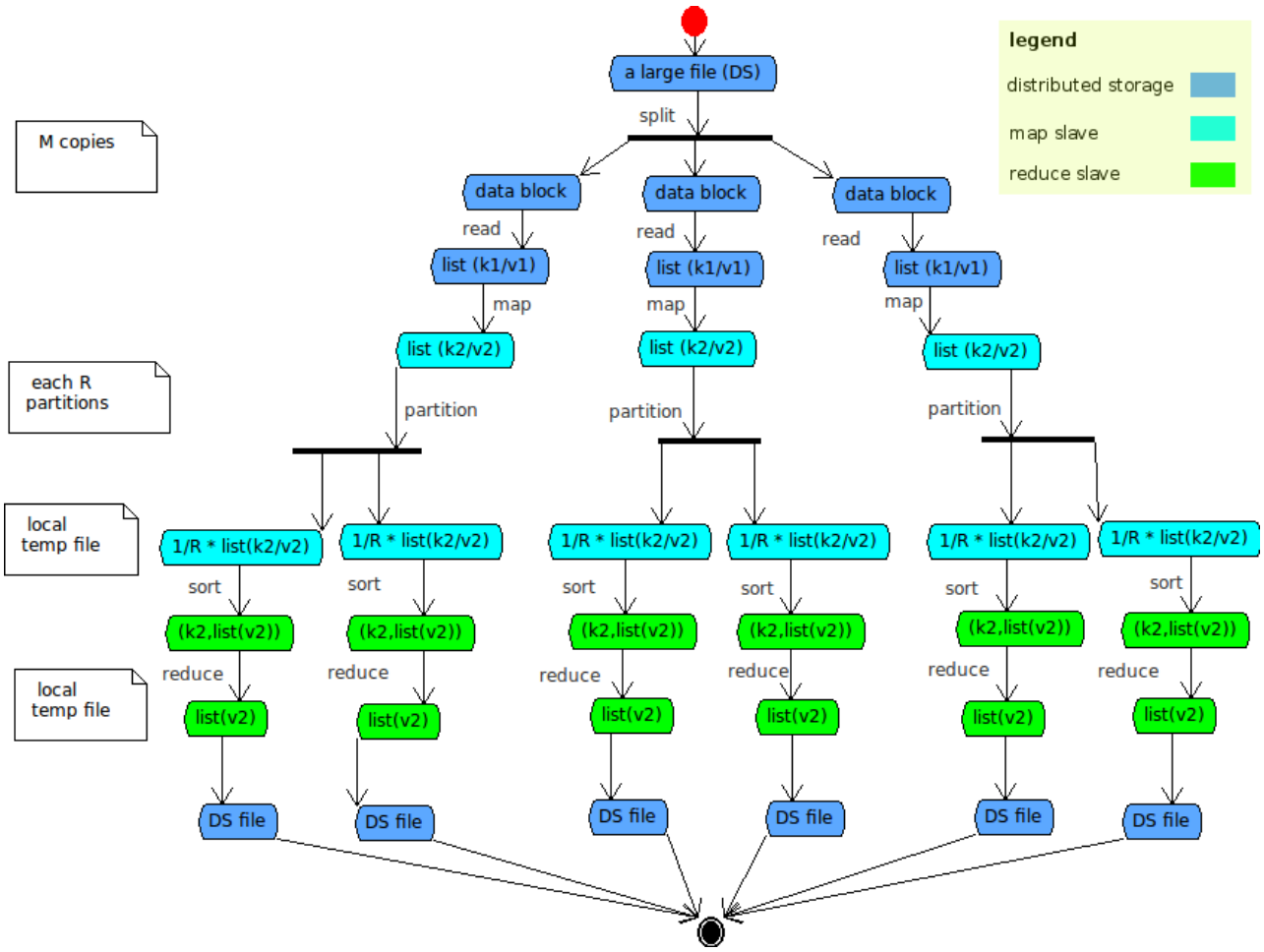
1. Job Client 收集配置参数：job Jar, Mapper, combiner (可选), Partitioner, Reducer, InputFormat, OutputFormat, OutputCommitter, input files, output files
2. Job Client 将一个 MapReduce Job 交给 MapReduce 的 Master，或等待 MapRedduce job 完成（如图所示）. Master 将 MapReduce Job (jar 文件和配置) 分发给 slaves，调度，监控运行状态以及重新调试失效的任务，反馈状态和诊断信息给 job client.
3. MapReduce 调用默认（或扩展实现）的数据分割器处理输入（这通常是分布式存储或文件系统中的数据）
4. **分割器**(Splitter)调用 split 功能，将输入数据分成 M 份
5. 对 **M** 份数据的每一份，**Master** 挑选一个空闲（或者基于负载平衡分析，目前不算忙的）Map Slave 并分配其一个 map 任务
6. **Map Slave** 从分布式存储远程读取属于它的那一块数据
7. 读取后，数据被解析成 list(k1/v1)
8. Map Slave 将每个 k1/v1 对 送给 用户定义的 map 方法
9. **map** 方法将 每个 k1/v1 转换成一人中间状态的 k2/v2 对，因此，对于每块数据，最终变成了比较大的 list (k2/v2)
10. **Partitioner** 利用 partition 方法将比较大的 list(k2/v2) 划分为 R 份 (此即是一个文件最终变成了 **M * R** 份了的原因)
11. Map Slave 将划分后的数据周期地存储在本地文件系统中
（此处，如果 map 产生的 key/value 对中 value 重复率高，为减少网络传输的消耗，允许定义一个 **combiner** 方法，该方法与下面将要讨论的 reduce 是一样，这样，combiner 在 map 方法执行的机器上执行，合并一次数据以减少数据的重复）

12. Map Slave 将每份划分后的(对 R 来讲)数据的位置通知给 Master
13. Master 将每份数据的位置发送给 一个空闲的 **Reduce Slave**
14. **Reduce Slave** 使用远程调用从 Map Slave 上获得划分过后的数据
15. 因为划分后的数据是一个 list(k2/v2)， Reduce Slave 将其按 key 排序以使相同 key 的数据集中在一起，即 list(k2, list(v2))
16. Reduce Slave 迭代 list(k2, list(v2))， 并将每个 k2 对应的 list(v2) 送给 用户定义的 reduce 方法
17. Reduce Slave 将 处理中的 list(v2) 存储在本要的临时文件中
18. Reduce Slave 通过远程调用将本地临时文件 rename/move 到最终的分布式存储
19. Reduce Salve 通知 Master 该 reduce 工作完成
20. Master 通知用户， MapReduce Job 结束



17.3 MapReduce 数据流状态

```
1 File/DataSet on Distributed Storage System →
Splitter reads and splits M Blocks →
InputRecordReader parses a block into list(k1,v1) →
Mapper map(k1,v1) → list(k2,v2) →
Partitioner partition into R copies and store at local Storage →
Reducer sort list(k2,v2) → list(k2, list(v2)) →
Reducer reduce reduce(k2,list(v2)) → list(v2) →
Reducer appends output data to local temp file →
Reducer stores the temp file onto Distribute  Storage System →
output M*R
```



从以上数据流可以看出，一个文件，会分成 M * R 份，最终每个 Reducer 输出一个输出到存储系统。因此一个输入，最终有 M*R 个输出。

17.4 基于 Hadoop 的 MapReduce 支持

如果不想"重新发明整个轮子"，那么基于 Hadoop 框架，要让分布式存储作为 MapReduce 的数据源和存储目的地，需要扩展 Hadoop MapReduce。

17.4.1 扩展实现

1. 扩展 InputSplit 将输入数据分割成存储系统 Token Range 列表
2. 扩展 RecordReader 将输入数据转换成存储系统的数据模型，并作为 kay/value 对，以传递给 map 方法
3. 扩展 InputFormat，封装以上 InputSplit 和 RecordReader 实现，以作为 MapReduce 的 job 输入规范：
   - validate 输入
   - split 输入
   - 读取分割后的数据并转换成 key/value 对
4. 扩展 RecordWriter 将 reduce 方法输出的 key/value 转换成 分布式存储系统的数据模型，并应用相关的更新。
   - 扩展 OutputFormat 封装 RecordWriter 作为 MapReduce 的输出规范：
   - validate reduce 的输出
5. 将 reduce 的输出 key/value 转换成存储系统的数据模型并存入系统
6. 扩展 OutputCommitter 对 MapReduce job 任务进行配置及管理

17.4.2 部署

可采用两种部署架构:

1. 源用 Hadoop 分层的集群策略，只不过，将 HBase 替换为分布式存储 YiJing，自顶向下依次为：
   - MapReduce 集群
   - YiJing 集群
   - HDFS 集群 （MapReduce 无论如何需要用到 HDFS）
   - ZooKeeper 集群（虽然，HBase 默认携带了一个内嵌的）

2. 将 MapReduce 的 DataNode/TaskTracker 与 YiJing 的节点重合。也就是，除了单独为 MapReduce 的 Master(NameNode/JobTracker) 准备一台机器外，其他的 Slave 都放在 YiJing 的节点上，这样做的好处是，体系层次简化，更高的数据本地命中率，并减少网络的开销。缺点是，对于每个节点，其实是担当双重角色，系统的硬件或许需要更高的配置。

18. 计算服务 (Compute Server)

- MapReduce != ForkJoin
- ForkJoin 的世界

18.1 MapReduce != ForkJoin

InfoQ 对 Doug Lea 的一次采访所知，ForkJoin 在 1998 年就被提出了，只是

18.2 ForkJoin 的世界

当我们需要海量数据的 Live (在线） 处理结果，而不仅仅是批处理(batch Processing) 时，我们需要进入 ForkJoin (没有 MapReduce) 的世界。

19. 负载均衡 (Load Balance)

- 参考实现
- 3-hop/2-hop/1-hop 路由策略

我们在分布式哈希表一节描述了，采用一致性的哈希将 key 空间均匀地分布在所有的节点上以确保负载均衡。但下面要讨论的是如何将客户端的请求均衡地分配到集群的节点上去。因此，将这一节放在了系统 Interface 部分。
因为，纵使分布存储器实现了上述负载均衡，而且每个节点是平等地，都可以作为请求的派发者。但是，客户使用 Client API，比如 REST， 访问系统时总会涉及到它要进行的通讯的主机或地址。虽然在去中心化的设计中，所有节点都可以服务请求，但是，客户端怎么知道：

1. 从整个集群的角度，哪个节点当前比较空闲
2. 如果没有一种 round-robin 策略，总是使用同一个节点，势必使该节点成为瓶颈，同时，根本没有发挥出去中心化的优势
3. 哪个节点当前不可用
4. 在请求的节点失效时，使用何种策略进行重试

大家可能会认为，这个问题难道还需要讨论吗？按照现存的工业标准方案，如配置传统的应用服务或数据库的那样：

1. 为数据存储集群架设一个硬件 load balancer 前端
2. 使用 round-robin DNS 指向数据存储集群所有的节点
3. 使用 软件实现的 load balancer 前端

这里不具体讨论上述方案，这样的体系架构自有它的用途。但外部的 load balancer 不可能会比存储节点更了解自身的 Partition 和负载状态，况且对于分布式存储系统来讲，额外的一层增加了额外的网络一跳 (1 network hop):

1. 而对于去中心化的设计的分布式存储，如果其“卖点”为 O(1) hop，增加一跳有着很大的影响。
2. 而本来 o(1) hop 的设计中，每一个节点都可以做为请求的分派者，何必需要一个傻傻的外部 load balancer 呢?
3. Amazon Dynamo (中文版：客户端驱动或服务器驱动协调) 中做的比较结果：***客户端驱动的协调方法，99.9 百分位减少至少 30 毫秒的延时，以及降低了 3 到 4 毫秒的平均延时。延时的改善是因为客户端驱动的方法消除了负载平衡器额外的开销以及网络一跳，这在请求被分配到一个随机节点时将导致的开销。***

但使用这种“智能“或”笨重“(thick) 客户端的做法有是有缺陷的：

1. 在比较简单的实现方案中，对集群成员状态的了解会有延时，因此它所做出的负载均衡策略的有效性在极端情况下，可能会大大降低。
2. 强迫客户端需要使用这一实现的 library

19.1 参考实现

事实上，去中化的存储系统的客户端路由策略是一个关注度非常高的主题，以下介绍相关实现。

19.1.1 Amazon Dynamo 客户端实现

在这个方案中，客户端应用程序使用一个该实现 library 在本地执行请求协调。客户端定期随机选取

一个节点，并下载其当前的成员状态视图。显然，这个方案的有效性是信赖于客户端的成员信息的新鲜度的。目前客户每 10 秒随机地轮循一遍节点来更新成员信息。然而，在最坏的情况下，客户端可能持有长达 10 秒的陈旧的成员信息。

19.1.2 笨重的 (thick) 客户端

因为一个真正智能的客户端是需要实时地了解当前集群的数据副本位置，Token 范围，负载状况，节点状态等信息。如果一个客户实现成这样，这难道不是试图将客户端作为集群的一个成员对待吗？在这各情况下，客户端除了做自己的事情外，还需要象其他集群节点一样，处理来自消息服务的各种控制信息，因为它此时它不再是集群的“被动”使用者，而是一个“主动”的成员。这也是它的名字(thick) 的来历。

19.1.3 智能客户端

定期使用管理接口获得 Token Map，而不是维护一个静态的集群节点列表
使用 RoutingStrategy 策略

19.1.3.1 路由策略接口

```
public interface RoutingStrategy {

    /**
     * Get the node preference list for the given key. The preference list is a
     * list of nodes to perform an operation on.
     *
     * @param key The key the operation is operating on
     * @return The preference list for the given key per replica strategy
     */
    public List<InetAddress> routeRequest(ByteBuffer key);
}
```

19.1.3.2 路由策略实现

RoutingStrategy 的实现周期地(10 seconds) 发送 Admin APi (getTokenMap, getNaturalNodeList(rowKey)) 请求，并更新本地的 live node 列表。

注意：这里有个”鸡生蛋，蛋生鸡“的问题：因为去中心化的设计使每个节点都是同等的，但是，发送 admin api 请求时，试图联系的节点不可用时怎么处理？ 因此，一个初始的列表是必须的，这可以通过配置文件列举出当前所有节点。因为 getTokenMap/getNaturalNodeList(rowKey) 的功能关系到整个 RoutingStrategy 的实现的有效性，所以当请求失败时，它会尝试列表中的下一个结点，直到成功为止。

19.1.3.2.1 路由策略(RoutingStrategy)

路由策略将客户端的操作映射到集群当前的节点。

Try_One_More_Consistent_Node

在放弃并返回失败前，按照顺序从首选节点列表选择一个活动的节点 (首选节点列表是通过 getNaturalNodeList(rowKey) 获得的，结果是按 replication 的优先级排好序的)

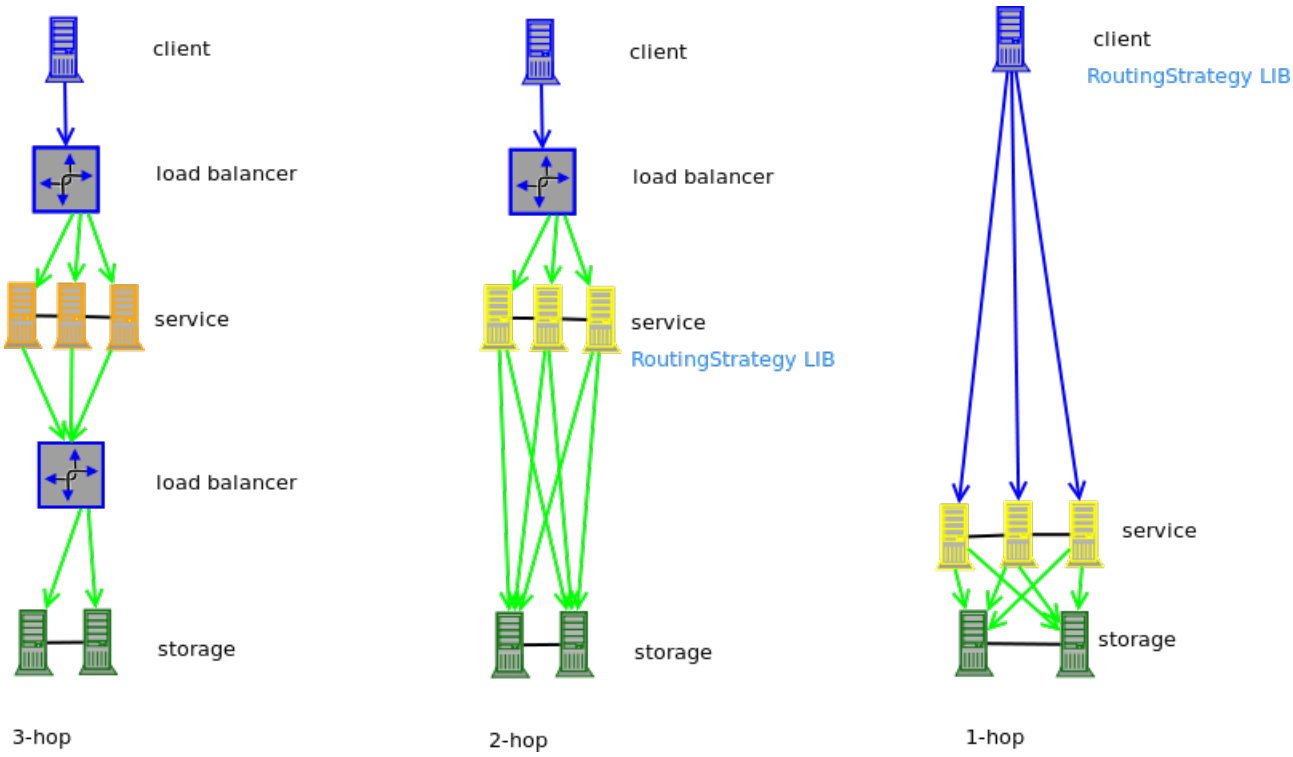
Try_All_Live_Node
在放弃并返回失败前，挨个试所有当前活的节点。

19.2 3-hop/2-hop/1-hop 路由策略

分布式存储系统的性能法则:

1. 通过分割（partition 的尺度来减少磁盘“寻道时间” (seek time）且将热点分布到多点上）和缓存来减少随机性磁盘访问 (cache and memtable 都是基于这一设计的)
2. 尽可能地减少网络跳数 (network hop), 因为减少跳数意味着减少网络延迟（对**响应时间**有利）和由于网络层次中硬件不均衡带来的潜在的瓶颈（对**吞吐量**有利）

下面介绍一下路由策略，以及为什么上述实现被归为 “2-hop” 路由策略。



19.2.1 3-hop

如果将一外部的 load balancer 置于 distributed storage 的前端，这一架构属于 3-hop 路由策略（图中最左中，3 次绿色的箭头）
客户端可见的是 service 层的前端 load balancer
service 层的前端 load balancer 根据负载均衡策略选择”最佳”的 service 主机 (1 hop)
service 主机访问数据层通过数据层的 load balancer (2 hop)
数据层的 load balancer 选择”最佳”的 存储节点 (3 hop)

上述描述中之所以将 ”最佳”打上引号，是因为，不同的 load balance 的实现会有不同的表现。对于数据层的前端，还难想象它可以实现一个 DHT 的策略（它岂不要实现所有本文提到的大多数设计决定？）
所以，除非是客户端(service) 不能使用 RoutingStrategy Library 的情况下，才使用这一策略。

19.2.2 2-hop

相比于 3-hop 策略，省去的一跳由实现的 RoutingStrategy Lib 来完成。正因为这一点，Service 作为存储系统的客户端，在对数据层的访问时，真正的 target Endpoint 是由这一 library 来指派的。
对于企业服务甚至“云”服务，service 层还是企业控制的范畴，例如 Amazon S3/SimpleDB/EC2。对于实现了 client 层的分布式存储器，只需要将这实现的路由策略置于 REST API 的实现中即可。所有外部的访问都通过 REST 接口，因此，对于最终的用户（REST 上层的 service 或最终的 Client）来讲是透明的。
唯一的限制来自于，第三方实现 存储的客户端，但又不想使用这一 RoutingStrategy library.

19.2.3 1-hop

从上面最右端的图可以看出，service 层和数据层合并在一块了。并且由于客户端使用 RoutingStrategy Library 了。这是所谓的 thick 客户端，因为 service 作为存储系统的客户端，真正成为了存储系统中的一员，唯一的一跳是在存储系统内部（service 到真正的存储节点）。
这一实现模糊的体系层次，紧耦合的设计最终会使系统失去必要的弹性(scalability)，而且存储系统需要区别对待 service (thick client) 和真正的节点.
因此，除非是一个真正私有的存储系统(因为 `RoutingStrategy Library` 直接被 `Client` 使用)，此策略应尽量避免。

六. 安全

- 安全基本概念
- 云安全

20. 安全基本概念

- 用户和群组
- 资源
- 许可
- 认证和授权

20.1 用户和群组

只有在系统中登录的用户才有可能授予相应的许可以对系统资源进行操作。
用户可以按其职能进行分组，如：系统操作员，普通用户。

20.2 资源

存储系统中，文件即代表资源，表示成文件在操作系统上的文件路径。

20.3 许可

许可即代表允许对资源进行的操作，包括：
admin
read
write

20.4 认证和授权

20.4.1 认证

Authentication(认证) 指的是通过检查声明的用户（一般是在 login 过程中传入的）与系统登录的用户进行校验，成功则返回一系统登录的用户对象，否则，认证失败。
简单的实现可以基于 Java properties，对于安全要求高的应用，需要将 用户名密码进行 MD5/SHA 加密

20.4.2 授权

Authorization(授权）是按照系统规则将认证的用户分配(映射)相应的许可的过程。规则可以是一简单的 java properties 文件的 key-value 列表，或者正则表达式表示

21. 云安全

- 基于数字签名(signature)的安全方案
- 密钥初始化 并关联到用户数据
- 客户端构造请求
- 服务端验证

对 public 的存储系统，需要更强壮加密技术，同时为了简化客户端的安全操作，OAuth 被国际巨头所采纳：Twitter, Yahoo, Google, Amazon
OAuth 2 正在被 IETF OAuth WG 采纳，期待早日标准化，以上实现是基于 OAuth 思想，简化其 3-legs 成 2-legs, 去掉 Token, 且不支持 revoke 功能

21.1 基于数字签名(signature)的安全方案

用户只需要一个登录用户管理系统的用户名／密码对，这个密码不会用作 服务身份证，如： REST API.用户只需登录并到 Profile 中取出 API Key 和 Secret Key.

21.2 密钥初始化 并关联到用户数据

```
${access key} = MD5(RSA KeyPair.public)
${secret key} = MD5(RSA KeyPair.private)
```

MD5** 的目的是使 **Key** 的长度缩短，以方便用于构造 **URI

21.3 客户端构造请求

HTTP HEADER:

```
${date} =
Date: EEE yyyy-MM-dd HH:mm:ss zzz

${api key} = URLEncoder.encode(access key, CommonUtil.UTF8);

${secretKey}=通过登录从用户信息中获得

${signature} =
URLEncoder.encode (
rfc2104HmacSha512(
httpMethod\n
date\n
resourcePath\n),
secretKey,
CommonUtil.UTF8);

${Request} =
http://host:port/resourcePath?apiKey=${api key}&signature=$
```

```
{signature}
```

因为 http 1.1 并没有规范化 HTTP HEADER 的字符集，从兼容的目的，大多数 SERVER 采用 us-ascii。所以格式化日期时，使用 TIME_ZONE_UTC, Locale.US

例如:

```
pprun@pprun-laptop:~$ curl -H 'Content-Type: application/xml' -H 'Accept:
application/xml' -H 'Date: Fri 2011-02-11 19:10:46 UTC'
'http://localhost:8080/hjpetstore/rest/products/dog?
apiKey=e4fae4f09fd3b2e6201b7b213d4deae7&signature=zVe5WbWJuJuPQpFLJVpJ4XMYTThdROf5iaU76
zdWLweeKvGSLBBJTAce4BayNH07x3poa8gHsIxLkIpFXsd5OQ%3D%3D&page=1&max=100
```

21.4 服务端验证

```
${access key} = URLDecoder.decode(api key, CommonUtil.UTF8);
${secret key}  =get from user data by ${access key}
${date} = Get from HTTP HEADER
${httpMethod} = HTTP METHOD
${resourcePath} = Request URI
${signature} = URLDecoder.decode(signature passed in URI,
CommonUtil.UTF8);

${calculated signature} =
URLEncoder.encode (
rfc2104HmacSha512(
httpMethod\n
date\n
resourcePath\n),
secretKey,
CommonUtil.UTF8);


if (calculatedSignature.equals(signature) == true)
   authenticate successful
else
   failed
```

例如:

```
private void validateSignature(HttpServletRequest httpServletRequest) {

String apiKey = httpServletRequest.getParameter("apiKey");

if (apiKey == null || apiKey.trim().isEmpty()) {

throw new
```

```
CodedValidationException(ErrorCodeConstant.ERROR_CODE_INVALID_API
_KEY, "apiKey can not be null");

}

if (log.isDebugEnabled()) {

log.debug("apiKey before decode: " + apiKey);

}

// decode back

try {

apiKey = URLDecoder.decode(apiKey, CommonUtil.UTF8);

} catch (UnsupportedEncodingException ex) {

throw new SignatureException("UnsupportedEncodingException when
URLEncoder.decode: " + CommonUtil.UTF8, ex);

}

if (log.isDebugEnabled()) {

log.debug("apiKey after decode: " + apiKey);

}



String secretKey = userService.getUserSecretKeyForApiKey(apiKey);

if (secretKey == null) {

throw new
CodedValidationException(ErrorCodeConstant.ERROR_CODE_INVALID_API
_KEY, "no secretKey can be found for the apiKey: " + apiKey);

}

String signature = httpServletRequest.getParameter("signature");

if (signature == null || signature.isEmpty()) {

throw new
```

```
CodedValidationException(ErrorCodeConstant.ERROR_CODE_INVALID_SIG
NATURE, "signature can not be null");

}

if (log.isDebugEnabled()) {

log.debug("signature before decode: " + signature);

}

// decode back

try {

signature = URLDecoder.decode(signature, CommonUtil.UTF8);

} catch (UnsupportedEncodingException ex) {

throw new SignatureException("UnsupportedEncodingException when
URLEncoder.decode: " + CommonUtil.UTF8, ex);

}

if (log.isDebugEnabled()) {

log.debug("signature after decode: " + signature);

}



String date = httpServletRequest.getHeader("Date");

if (date == null || date.isEmpty()) {

throw new
CodedValidationException(ErrorCodeConstant.ERROR_CODE_NO_FOUND_
DATE_HEADER, "Date header MUST be set in the request");

}

if (log.isDebugEnabled()) {

log.debug("header Date: " + date);
```

```
}

// String has been Signed = HTTP-Method + "\n" + Date + "\n" +
resourcePath + "\n";

String httpMethod = httpServletRequest.getMethod();

if (log.isDebugEnabled()) {

log.debug("httpMethod: " + httpMethod);

}

String resourcePath = httpServletRequest.getRequestURI();

if (log.isDebugEnabled()) {

log.debug("resourcePath: " + resourcePath);

}

String calculatedSignature =
MessageDigestUtil.calculateSignature(httpMethod, date, resourcePath,
secretKey);

if (log.isDebugEnabled()) {

log.debug("calculateSignature = " + calculatedSignature);

}

if (calculatedSignature.equals(signature) == false) {

throw new SignatureException("Invalid signature: the calculated signature
(" + calculatedSignature +

") does not match with the signature passed in (" + signature + ").");

}

}
```

七. 操作需求

- 管理工具
- 水平扩展 Scale Horizontally

22. 管理工具

Admin Tool

rolling restart

23. 水平扩展 (Scale Horizontally)

- 垂直扩展 Scale Vertically (Scale Up)
- 水平扩展 Scale Horizontally (Scale Out)
- 弹性存储 (Elastic Storage)

本节关注存储系统的扩展性（也叫伸缩性 Scalability）

23.1 垂直扩展 Scale Vertically (Scale Up)

a

23.2 水平扩展 Scale Horizontally (Scale Out)

a

23.3 弹性存储 (Elastic Storage)

a

参考资料

[1] Amazon Dynamo

[2] Eventual Consistency Revisited
[3] CAP
[4] Gossip
[5] Phi Accrual Failure Detector
[6] BloomFilter
[7] MerkleTree
[8] Distributed Hash Table
[9] O(1)lookup performance for power-law query distributions in peer-to-peer overlays
[10] Google GFS
[11] Google BigTable
[12] Google MapReduce
[13] Voldemort Project
[14] Apache Hadoop
[15] Apache Hbase
[16] Apache Cassandra