分类号： 单位代码：

密　级： 学　　号：

山东大学

学士学位论文

Shandong University Bachelor’s Thesis

论文题目：基于 Raft 协议的分布式 Nosql 数据库集群--Cabinet 的实现

作 者 姓 名：郑　宇

学　　号：201200121256

专　　业：电子信息工程

指导教师姓名：王 鹏 伟

2016 年 5 月 16 日

目　录

目　录 1

摘　要 3

Abstract 4

第一章 绪论 5

1.1 选题背景 5

1.2 国内外分布式数据库集群发展 5

1.3 分布式数据库集群研究目的 5

1.4 本文的主要内容及组织结构 6

第二章 系统方案设计 6

2.1 一致性协议的比较及选择 6

2.2 数据库类型的比较及选择 6

2.3 数据分布方式的比较及选择 7

2.4 网络编程模型的比较及选择 7

第三章 Cabinet 实现原理 8

3.1 Raft 协议 8

3.1.1 节点角色及职责 9

3.1.2 选主过程 10

3.1.3 日志复制 11

3.2 一致性哈希协议 12

第四章 Cabinet 设计 12

4.1 设计目标 12

4.2 总体结构 13

4.3 消息协议 15

第五章 Cabinet 的实现和应用 16

5.1 核心类和算法的实现 16

5.1.1 网络基础框架的设计与实现 16

5.1.2 数据库 Serve 节点的设计与实现 18

5.1.3Raft 节点的设计与实现 19

5.2 Cabinet 应用实例 20

第六章 总结与展望 26

致　谢 28

参考文献 29

附　录 26

摘　要

所谓分布式数据库集群[1]，既是将数据库数据分散在不同的主机中，同时利用集群其他机器对数据进行复制。将数据分散，可以使数据库拥有更大的容量，并且可以通过分片来对数据流访问进行优化。而数据的复制，一方面为数据库提供容灾措施，提高数据库的可用性，另一方面允许读写分离技术方案的实现，有利于促进服务提供更高的并发访问量。而在这些措施和优化之上，分布式数据库集群还要提供给用户尽可能简单的访问方法，访问分布式数据库集群就像在访问一个单机数据库一样。本文基于对 Raft[2]协议的研究，在 Linux 系统上实现了分布式 Nosql[3][4]数据库集群 Cabinet。

【关键词】分布式数据库；数据分片；数据复制；容灾；高可用；高并发

[1]杨传辉. 大规模分布式存储系统[M]. 机械工业出版社, 2013.

[2] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[J]. Draft of October, 2013.

[3]姚林, 张永库. NoSQL 的分布式存储与扩展解决方法[J]. 计算机工程, 2012, 38(6):40-42.

[4]皮雄军. NoSQL 数据库技术实战[M]. 清华大学出版社, 2015.

Abstract

The so-called distributed database cluster is a database that distributes data among different servers as well as a cluster replicates all its data to all his servers. Distributing the data could larger the column for more data, besides it could optimize the reading stream by sharding. The replicated data could provide the whole database with fault-tolerence thus improves the avaliability in the one hand, in the other hand it permits the read-write seperate method to be achieved which would highly improve the concurrency of service. Among all these optimization, the distributed database cluster aims to provide the user a way to use it as easy as possible, meaning that accessing the distributed database cluster should be just like accessing a local database. This paper achieves a distributed NoSql database cluster-Cabinet on Linux system basing on the research of the Raft algorithm.

【Key words】 distributed database cluster; data sharding; data replication; fault-tolerence; high-avaliability; high-concurrency

第一章　绪论

1.1 选题背景

近些年来，互联网在国内外蓬勃发展以至白热化阶段。不同互联网公司可能提供近似的服务。在用户为王的时代，服务同质化比较严重的情况下，对服务的质量提出了更高的要求，也只有在服务的质量上了一个台阶之后，企业才能有余力进行服务的其他方面改进。数据库服务作为互联网应用的基石，为提高服务的质量，在可用性和并发量方面面临了更高的挑战。因此研究分布式数据库集群有重要的意义。

1.2 国内外分布式数据库集群发展

目前世界上较为出色的分布式数据库大多由外国公司研发，如 Google Spanner 是 Google 的全球级分布式数据库，能够扩展到数百个数据中心，数百台机器，上万亿行记录，还能通过同步复制和多版本控制来满足一致性，支持跨数据中心的事物。Amazon 的 Dynamo 也是较为出色的产品。

国内目前也有较为知名的分布式数据库，比如淘宝的 OceanBase，实现了数千亿条记录，数百万 TB 上的跨行跨表事务，支撑了淘宝庞大的业务量。

开源方面也有许多较为出色的产品，广泛适用于生产环境当中，如 MongoDB，Redis[5]，Cassandra，ZooKeeper[6]等。

1.3 分布式数据库集群研究目的

目前国内外有许多成熟的数据库集群，广泛适用于各种各样的生产环境，支撑着当今互联网的许多应用。然而其中许多产品不是开源，或者是系统结构、部署方法等较为复杂，且有较多的历史包袱，无法使用最新的理论来实现系统框架。

Raft 协议由斯坦福大学的 Diego Ongaro 和 John Ousterhout2013 年发表的论文中被提出，可用于解决分布式系统中的一致性问题。该协议在提供不低于之前风靡世界的一致性协议--Paxos 的性能和表现的情况下，将可理解性放在首位，并且在论文中给意欲实现 Raft 协议的工程师提供了完整的指导，之后甚至出了一本书来进行更进一步的

---

[5]王心妍. Memcached 和 Redis 在高速缓存方面的应用[J]. 无线互联科技, 2012(9):8-9.

[6]倪超. 从 Paxos 到 Zookeeper[M]. 电子工业出版社, 2015.

指导。

在 Redis 的 2.8 版本中提供了名为 sentinel（中文意思为哨兵）的主从切换方案，多个 sentinel 检测集群中主从机器的可用性，在检测出机器异常之后进行选举，选出一个 leader 来对机器异常作出相应的处理。其中 sentinel 选举 leader 即利用了 Raft 协议中较为重要的一部分，目前证明使用良好。除了 Redis，还有很多系统实现并利用了 Raft 协议进行一致性的处理。

然而 Raft 协议还未较完整地被用在生产环境中。本文意在实现并使用 Raft 协议，一方面实现自己的分布式数据库集群，另一方面深刻理解、探究一致性协议的使用方法及价值。

1.4 本文的主要内容及组织结构

本文的研究内容包括六章。其中，第一章为绪论部分，简单的介绍了分布式数据库集群的背景，分布式数据库集群在国内外的发展情况以及本文的研究目的；第二章重点介绍系统方案的对比和选择，包括一致性协议，数据库类型，服务器架构；第三章主要介绍 Cabinet 的实现原理，包括 Raft 协议，一致性哈希协议；第四章主要介绍了 Cabinet 的设计，包括设计目标，总体结构，实现的功能；第五章介绍了 Cabinet 的核心类，核心算法的代码实现，Linux 上 Cabinet 的使用示范；第六章总结了本文所涉及的内容，提出 Cabinet 可改进的方面，对分布式数据库集群的下一步研究进行展望。

第二章 系统方案设计

2.1 一致性协议的比较及选择

为提高数据库的可用性，数据库需要对数据进行复制、备份，作出容灾处理。

当数据处在多份的时候，就会出现一致性问题，具体比如：

(1) 集群如何管理

(2) 数据如何传输

(3) 当数据出现不一致情况如何应对

(4) 当主机异常下线怎么办

(5) 当主机重新上线之后如何传输数据

这些一致性问题既是一致性协议意在解决的。

目前广泛使用的一致性协议为 Paxos[7][8]协议，是 Leslie Lamport 于 1990 年提出的，为目前公认的此类算法当中最有效的。Google 的 Chubby 和 Yahoo！的 ZooKeeper 均使用并部分实现了 Paxos。但也正是因为 Paxos 过于复杂、含糊，所以至今没有对 Paxos 的完整实现。

2013 年斯坦福大学的 Diego Ongaro 和 John Ousterhout 发表的论文中提出了 Raft 协议，该协议在提供不低于 Paxos 的性能和表现的情况下，将可理解性放在首位，并且在论文中和之后写成的书中给想要实现 Raft 协议的工程师提供了完整的指导，作者还动手实现了一个基于 Raft 协议的分布式存储系统 LogCabin 作为参考，该系统功能类似 ZooKeeper，可以为其他分布式集群存储元数据，帮助解决集群管理问题。

综上，使用更加便于理解的 Raft 协议作为一致性协议。

2.2 数据库类型的比较及选择

此系统还需要一个数据库来存储数据，作为状态机使用。

传统的关系型数据库具有一些较为突出的优点，如：

(1) 保持数据的一致性

(2) 数据更新的开销较小

(3) 可以进行连表等复杂的操作

---

[7] Lamport L. The part-time parliament[J]. Acm Transactions on Computer Systems, 1998, 16(2):133-169.

[8] Lamport L. Paxos Made Simple[J]. Acm Sigact News, 2001, 32(4).

(4) 有较成熟的技术支持

但是也有一些场景是传统的关系型数据库不擅长处理的，比如：

(1) 大量数据的写入操作

(2) 对简单查询快速返回结果的能力

(3) 对数据表结构的表更等等

以上提到的场景往往是近些年来快速发展的互联网行业经常面临的，所以另一种数据库崛起了--NoSql，既 Not Only Sql，代表产品有 Redis 等，此类 NoSql 数据库在面对以上场景的时候有较好的表现。所以选择 NoSql 作为此系统中的状态机。

2.3 数据分布方式的比较及选择

分布式系统区别于传统的单机系统在于能把数据分布到多个主机上，而数据分布的方式主要有两种，一种为顺序分布，既按照主键整体有序将数据分布到不同节点上，另一种为哈希分布，代表性方法为一致性哈希分布[9]，较顺序分布有更加好的表现，更加适合分布式系统。

一致性哈希算法主要思想如下，给每个节点分配一个随机的哈希值，所有节点构成一个哈希环。数据操作时，先计算数据主键的哈希值，然后在哈希环上按顺时针方向找到第一个大于或等于该哈希值的节点，再在该几点上对数据进行操作。一致性哈希的有点在于节点的加入或者是删除只会影响到环中的相邻节点而不会对所有节点产生影响。

基于一致性哈希分布的优点，选用其作为数据分布方案。

2.4 网络编程模型的比较及选择

W.Richard Stevens 在他的著作[10][11]中提到过十余种当时（20 世纪 90 年代末）流行的并发网络编程模型，但是大多着墨于阻塞式网路编程，对分阻塞方面探讨较少。

进入 21 世纪，随着互联网浪潮的兴起，这一领域的研究有了较大的发展，目前常

[9] Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web[C]// Twenty-Ninth ACM Symposium on Theory of Computing. ACM, 1997:654-663.

[10] (美)史蒂文斯(Stevens, W. R.). UNIX 网络编程[M]. 人民邮电出版社, 2010.

[11] W.RICHARD STEVENS[美]. UNIX 环境高级编程[M]. 人民邮电出版社, 2014.

见的方案大概有 12 种，而较适合本系统的方案主要有两种，既单线程 IO 多路复用（Reactor）和多线程 IO 多路复用（one loop per thread）[12]。

两种方案各有利弊：在 CPU 密集型的程序中，针对每个请求的处理时间较长，适合使用多线程 IO 多路复用，更多地利用多核 CPU 的计算能力；而在 IO 密集型的程序中，更加适合使用单线程 IO 多路复用，避免多线程情况下处理锁和共享数据带来的复杂性和耗时，将力气用到刀刃上。

本系统中的 Server 节点和 Raft 节点，为 IO 密集型应用，故选择单线程 IO 多路复用为网络编程模型。

[12]陈硕. Linux 多线程服务端编程[M]. 电子工业出版社, 2013.

第三章 Cabinet 实现原理

3.1 Raft 协议

Raft 协议由斯坦福大学的 Diego Ongaro 和 John Ousterhout 在 2013 年发表的论文中提出，是一个出色的一致性算法，可用于在多个主机中进行“日志”复制，保持多个数据副本的一致性。

在数据库集群中，通过多个数据副本来保证数据的可用性甚至提高并发度，多个副本的存在就导致了对保证数据一致性的需求的显现，而 Raft 协议正式我们需要的。

以下简单介绍 Raft 协议。

3.1.1 节点角色以及职责

在 Raft 集群中，节点可能会有三种角色，每种角色有各自不同的职责，三种角色分别为：

(1) Follower（追随者）

(2) Candidate（选举者）

(3) Leader（领导者）

节点中有如下状态变量，会随着角色或者当前集群状态改变而改变：

(1) currentTerm，节点当前的纪元，是一种类似记录时间的标记

(2) voteFor，该节点投票给的节点

(3) log[]，日志数组

(4) commitIndex，经过集群确认可以提交的最后一个日志编号

(5) lastApplied，该节点应用于状态机（数据库）的最后一个日志编号

(6) nextIndex[](leader 拥有)，针对每一个 follower 节点，需要发送给它的下一个日志编号

(7) matchIndex[](leader 拥有)，针对每个节点，确认复制到其上面的最后一个日志的编号

节点职责如下：

(1) 所有节点的职责：

a)如果 commitIndex > lastApplied，增加 lastApplied，将 log[lastApplied]应用到

状态机；

b)如果 RPC 请求或者响应中的 term 大于当前节点的 currentTerm，更新 currentTerm 为 term，转变为 follower

(2) Follower 的职责：

a)响应来自 Candidate 和 Leader 的 RPC

b)如果选举超时已到却没有收到来自 Leader 的 AppendEntry RPC 或者是来自 Candidate 的请求投票的 RPC，切换为 Candidate

(3) Candidate 的职责：

a)最开始变为 Candidate 的时候，递增 currentTerm，给自己投票，重置选举超时，发送 RequestVote 的 RPC 给所有的其他节点

b)如果收到来自大多数节点的投票，变为 Leader

c)如果收到一个新的 Leader 的 AppendEntry RPC，变为 follower

d)如果选举超时达到，开始新的选举

(4) Leader 的职责：

a)刚获得选举成功的时候，发送空的 AppendEntry 给所有节点，同时在空闲时间定时发送给所有节点，避免其他节点选举超时达到

b)如果收到客户的请求，将请求日志加入本地日志，在日志应用到状态机后再将响应发回给客户

c)如果一个 follower 对应的 nextIndex 小于等于 Leader 最后一条日志的 id，将 nextIndex 对应的日志发送给该 follower。如果该 RPC 成功，更新该 follower 的 nextIndex 和 matchIndex；如果失败，减小 follower 的 nextIndex，并重试

d)如果日志中有一个日志 id 大于 commitIndex，并且大多数节点的 matchIndex 大于等于该 id，且日志的 term 等于当前 term，将 commitIndex 置为该日志 id（既提交该日志）

3.1.2 选主过程

集群启动时，节点角色为 Follower。之后节点的选举超时（各个节点有些许差异）达到之后，节点从 Follower 转变为 Candidate，并且向其他节点请求投票。如果获得大多数节点的支持，即变为 Leader，向外对客户端提供集群服务，向内管理整个集群的状态，负责将数据传播到每个节点。若当前 term 中有多个节点变为

Candidate，导致所有 Candidate 节点都无法得到大多数节点的支持，即跳过该 term，重新发起一个新的 term。

在选举过程中，每个 Candidate 都会发送 RequestVote 的 RPC 给其他节点，该 RPC 的发送参数如下：

(1) term，Candidate 的 currentTerm

(2) candidateId，Candidate 的节点唯一标示 id

(3) lastLogIndex，Candidate 的最后一个日志的 id

(4) lastLogTerm，Candidate 最后一个日志的 term

RPC 收到回复的参数有：

(1) term，回复该 RPC 的节点的 currentTerm

(2) voteGranted，是否给 Candidate 投票

收到 RequestVote RPC 的节点应该根据不同情况回复该 RPC：

(1) 如果节点的 currentTerm 大于 Candidate 的 term，回复 false

(2) 如果还为投过票，且 Candidate 的日志至少跟该节点的日志一样新，就给该 Candidate 投票

3.1.3 日志复制

在集群成功选举出 Leader 之后，Leader 向外对客户端提供服务，向内会对集群机器定时发送 AppendEntry RPC，此 RPC 一方面可以在需要日志同步的时候将日志传送给集群中的其他节点，另一方面在空闲时候充当心跳检测的机制，探测集群节点状态是否正常。

AppendEntry RPC 的发送参数如下：

(1) term，Leader 的 currentTerm

(2) leaderId, Leader 的节点 id

(3) prevLogIndex，当前 RPC 发送的日志的前一个日志的编号

(4) prevLogTerm，当前 RPC 发送的日志的前一个日志的 term

(5) entries[]，要发送的日志数组

(6) leaderCommit，Leader 的 commitIndex

收到回复的参数有：

(1) term，回复该 RPC 的节点的 currentTerm

(2) success，该 RPC 的返回结果

收到该 RPC 的节点根据当前情况的不同回复该 PRC：

(1) 如果 RPC 中的 term 小于节点的 currentTerm，回复 false

(2) 如果节点当中没有日志能对应 prevLogIndex 和 prevLogTerm，回复 false

(3) 如果存在一个日志跟 RPC 中发送的日志有冲突，删除该日志以及之后所有日志

(4) 将 RPC 中本地没有的日志保存到本地

(5) 如果 leaderCommit 大于本地节点的 commitIndex，置 commitIndex 为 leaderCommit 和最后一个日志 id 的较小者

根据日志复制，当集群中节点日志不一致时候，只要适当的 Leader 能够被选出，就能根据该 RPC 使集群节点之间的数据保持一致。

3.2 一致性哈希协议

如前所述，本系统使用一致性哈希算法作为数据分布算法。算法基本描述如下。

选择一个较大的质数 Hmax，将数据库节点均匀（或者根据负载能力不均匀）分布在[0, Hmax]区间上，组成一个环路。

当数据请求到来时，计算主键的哈希值，在哈希环上找到该哈希值所在的点，再顺时针找到第一个节点，作为处理该数据请求的数据库节点。

当哈希环中的某一节点崩溃下线之后，将该节点负责的数据转移到顺时针的下一个节点，由下一个节点负责该段环路上的数据操作。

当插入一个节点之后，即将该节点在哈希环上的数据点到逆时针上第一个点之间的数据交给新的节点负责。

通过一致性哈希算法，可以灵活处理分布式系统中数据库节点失效或者是实时扩容的问题，有效提高集群的可用性。

第四章 Cabinet 设计

4.1 设计目标

本文希望 Cabinet 能实现的目标如下：

(1) 实现 Raft 协议主要内容的 Raft 节点，实现数据库集群的一致性；

(2) 实现一个 NoSql 数据库作为系统中的状态机；

(3) 实现一个基本的命令行客户端进行集群的使用；

(4) 各个系统结构之间通过 Tcp[13]连接，实现松耦合；

(5) 实现一个健壮可用的服务器框架

(6) 实现自己的消息收发协议

本文希望通过 Cabinet 能够实现如下功能：

(1) 通过直接连接数据库节点，进行基本的数据存储；

(2) 通过完整的集群结构，实现数据一致性；

(3) 在集群节点一半以下出现故障时候，依旧保证整个集群服务可用；

(4) 将数据合理地分布到各个数据库节点；

(5) 节点故障恢复上线之后，跟集群中节点进行数据同步，保证数据一致；

(6) 提供基本数据库操作，如 get，set。

4.2 总体结构

[13] W.RICHARD STEVENS[美. TCP/IP 详解(卷 1)--协议[M]. 机械工业出版社, 2011.

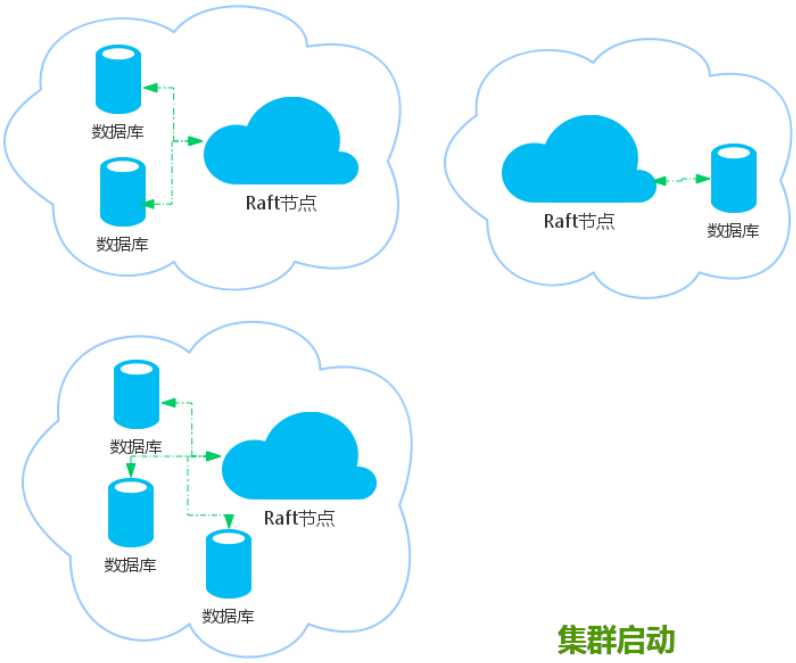

图 4.1 集群启动视图

完整的系统结构图 4.1 所示。

整个集群一共有三个 Raft 节点，Raft 节点下连接数量不一的数据库,数据库只暴露给响应的 Raft 节点，对其他 Raft 节点包括客户端为不可见状态。

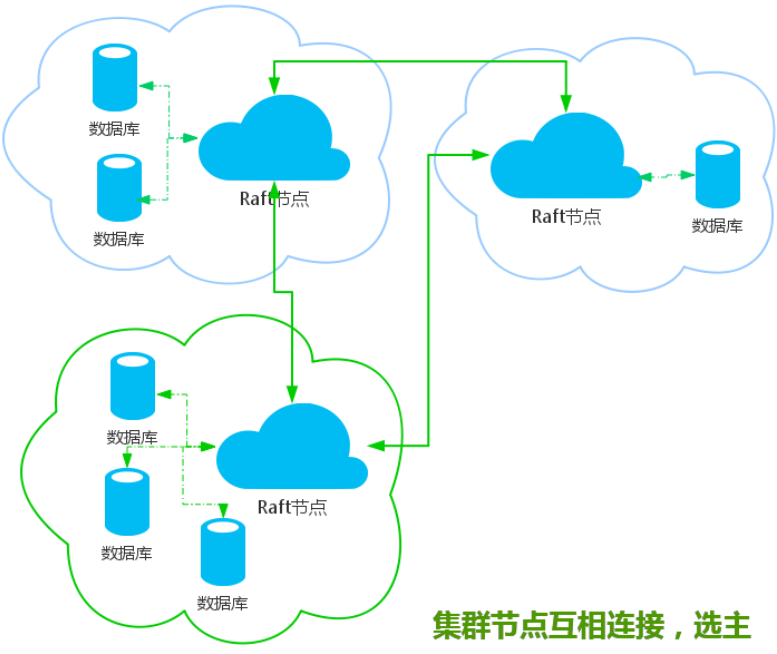

图 4.2 集群连接选主视图

如图 4.2，集群启动时，Raft 节点互相识别，相互建立连接。

启动成功之后，集群根据 Raft 协议进行选主。

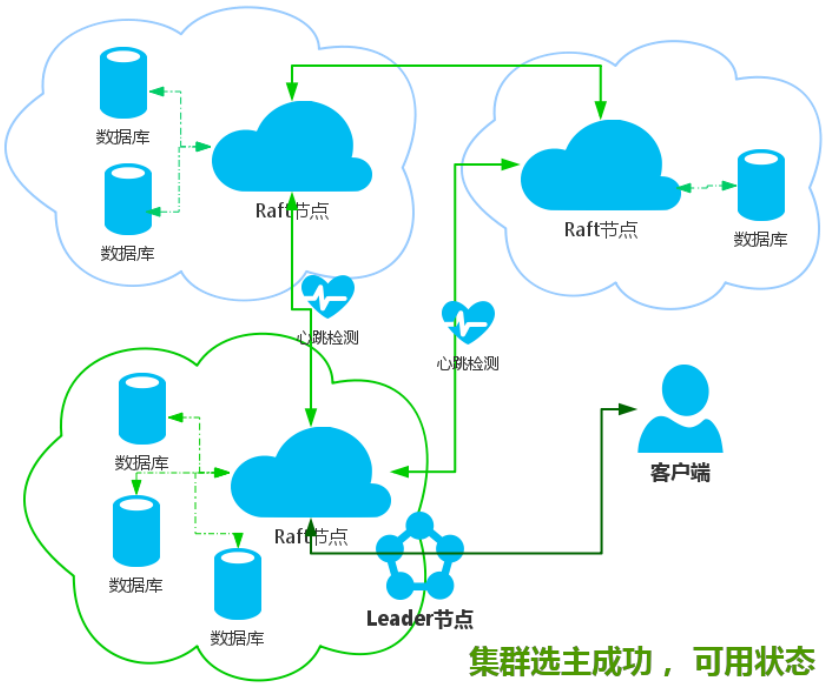

图 4.3 集群正常提供服务视图

如图 4.3 所示，选主成功之后，Leader 向客户端提供服务。当客户端连接到非 Leader 节点时，非 Leader 节点将 Leader 地址端口告知客户端，客户端重定向至 Leader 节点请求数据服务。

Leader 节点同时通过心跳检测告知 Follower 其存在并探测 Follower 节点的可用性。

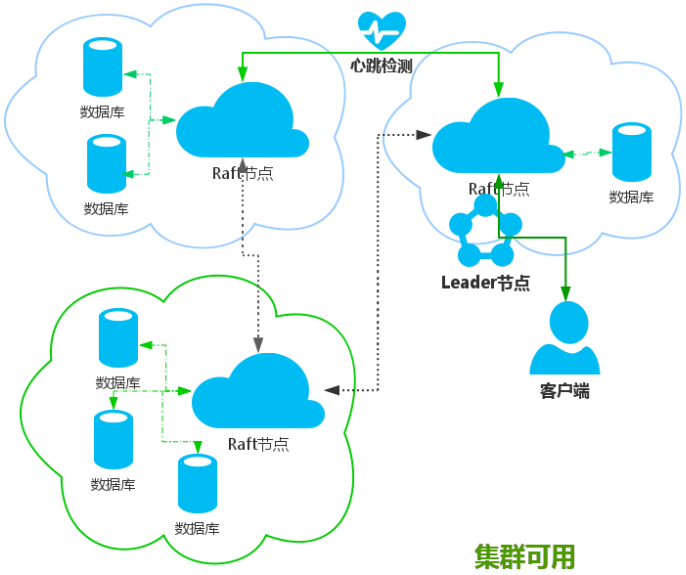

图 4.4 集群少数节点下线视图

如图 4.4，当集群中的节点下线之后，集群重新进行选主，继续为客户端提供服务。

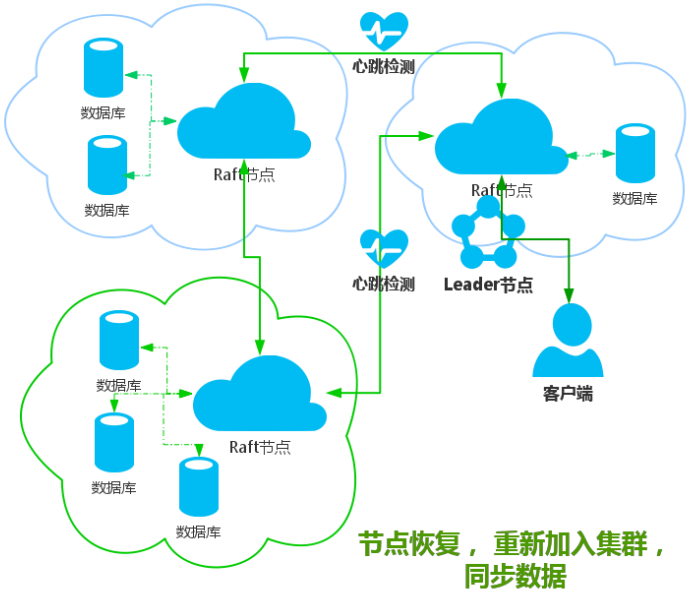

图 4.5 集群故障机器恢复视图

如图 4.5 所示，节点恢复，重新加入集群中，同步数据，继续为客户端提供服务。

4.3 消息协议

之后节点恢复，重新加入集群中，同步数据，继续为客户端提供服务。

基于 Tcp 的数据传输不能保证一次系统调用（write 套接字）时发送的数据能够一

起到达目的主机，所以还需要一个更高层的数据传送协议。

现成的数据协议较多，如 XML，JSON 等。由于此系统的消息格式较为一致，所以设计实现消息协议格式如下：

*参数个数\n

[#命令属性(读或是写)\n]

$第一个参数字节数\n

第一个参数\n

$第二个参数字节数\n

第二个参数\n

...

可以看出，一条消息以星号开始，之后紧跟参数个数，接下来是井号跟上命令的属性，之后以美元号加上参数字节数，再接上参数，以此循环直至参数全部传输完成。

消息中的每个单元以换行符\n 分割。每个参数中都先给出参数大小，因此协议是二进制安全的，特殊字符如\n 也能正常传输。

消息例子：

*2\n#r\n$3\nget\n$3\nfoo\n

可以看到此消息包含两个参数，消息为只读消息（类型为 r），第一个参数包含三个字符，第一个参数为 get，第二个参数为三个字符，第二个参数为 foo。

第五章 Cabinet 的实现和应用

5.1 核心类和算法的实现

5.1.1 网络基础框架的设计及实现[14]

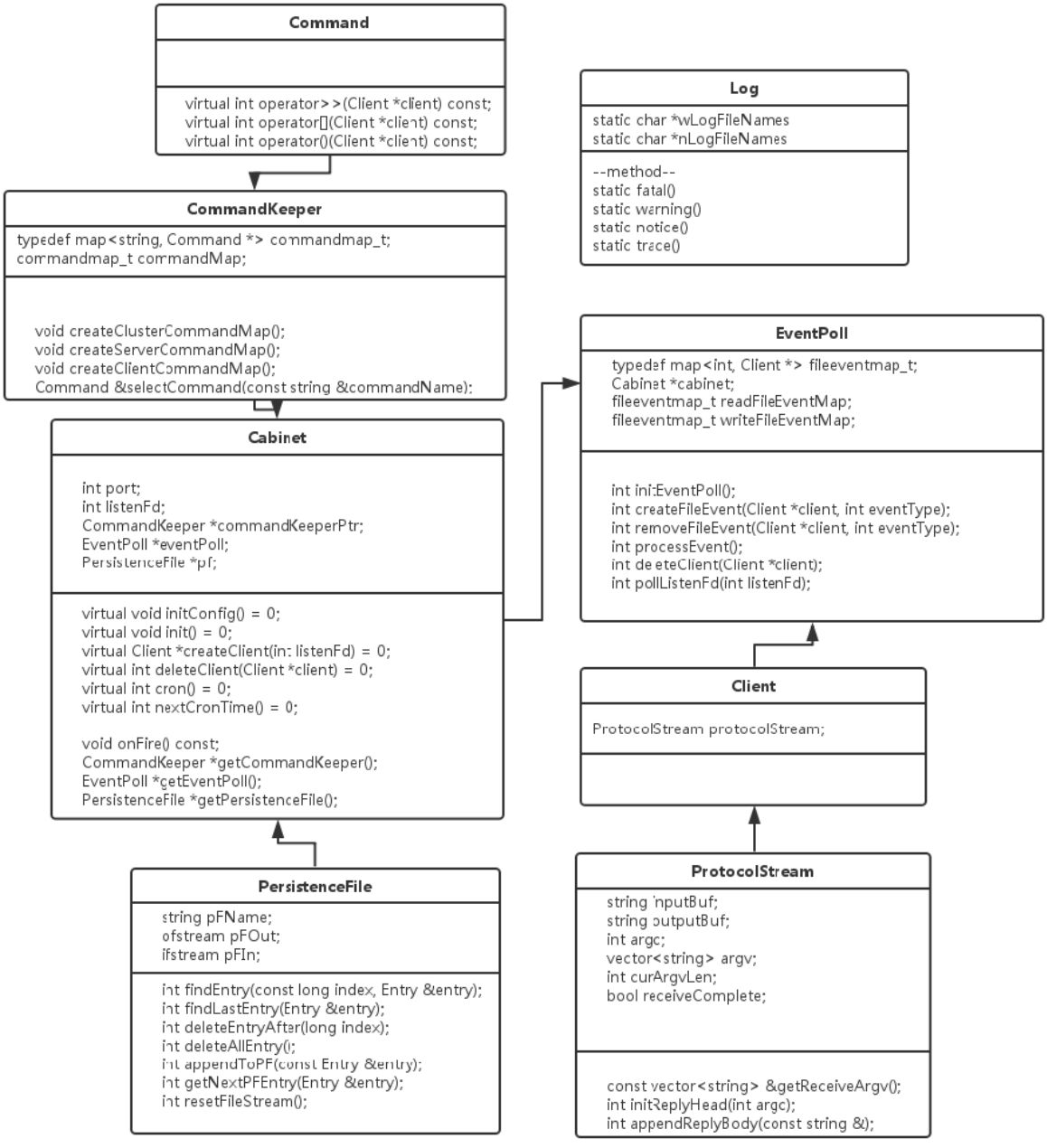

图 5.1 网络基础框架的设计及实现

如图 5.1 所示，针对系统中的服务器-客户端结构，抽取共性，实现了一个网络基

[14] (美)福勒 著;侯捷 熊节. 重构-改善既有代码的设计[M]. 中国电力出版社, 2003.

础框架。其中包括：

(1) Cabinet 类，为 Server 节点以及 Raft 节点的共同逻辑抽取出来的基类，提供接口供 EventPoll 类调用，包括创建、删除客户端等；

(2) EventPoll 类，实现了 I/O 多路复用，监听网络端口连接请求，处理 Tcp 连接中的读写事件，创建、删除客户端；

(3) Client 类，每个对服务器节点进行连接的节点创建一个 Client 类，包括节点中的基本信息，标记一个唯一的 Tcp 连接；

(4) ProtocolStream 类，处理针对客户端收发数据的缓存，负责分包并解析数据，实现了 4.3 节中描述的消息传输协议；

(5) CommandKeeper 类，针对不同的服务器主节点（Cabinet 的继承类），存储不同的命令，以命令名称作为键，以命令对象作为值，有效地提供了一个抽象层，很好地将系统中的逻辑进行归纳；

(6) Command 基类，为命令对象的基类，具体的命令逻辑继承基类，重写里面的方法逻辑，实现系统功能；

(7) PersistanceFile 类，日志持久化类，负责对本地磁盘日志进行写入，查找，删除等工作；

(8) Log 类，打印工作日志到磁盘，支持多种日志等级，包括 Fatal，Warning，Notice，Debug，方便构建系统时的调试并了解系统当前的状态；

5.1.2 数据库 Server 节点的设计及实现

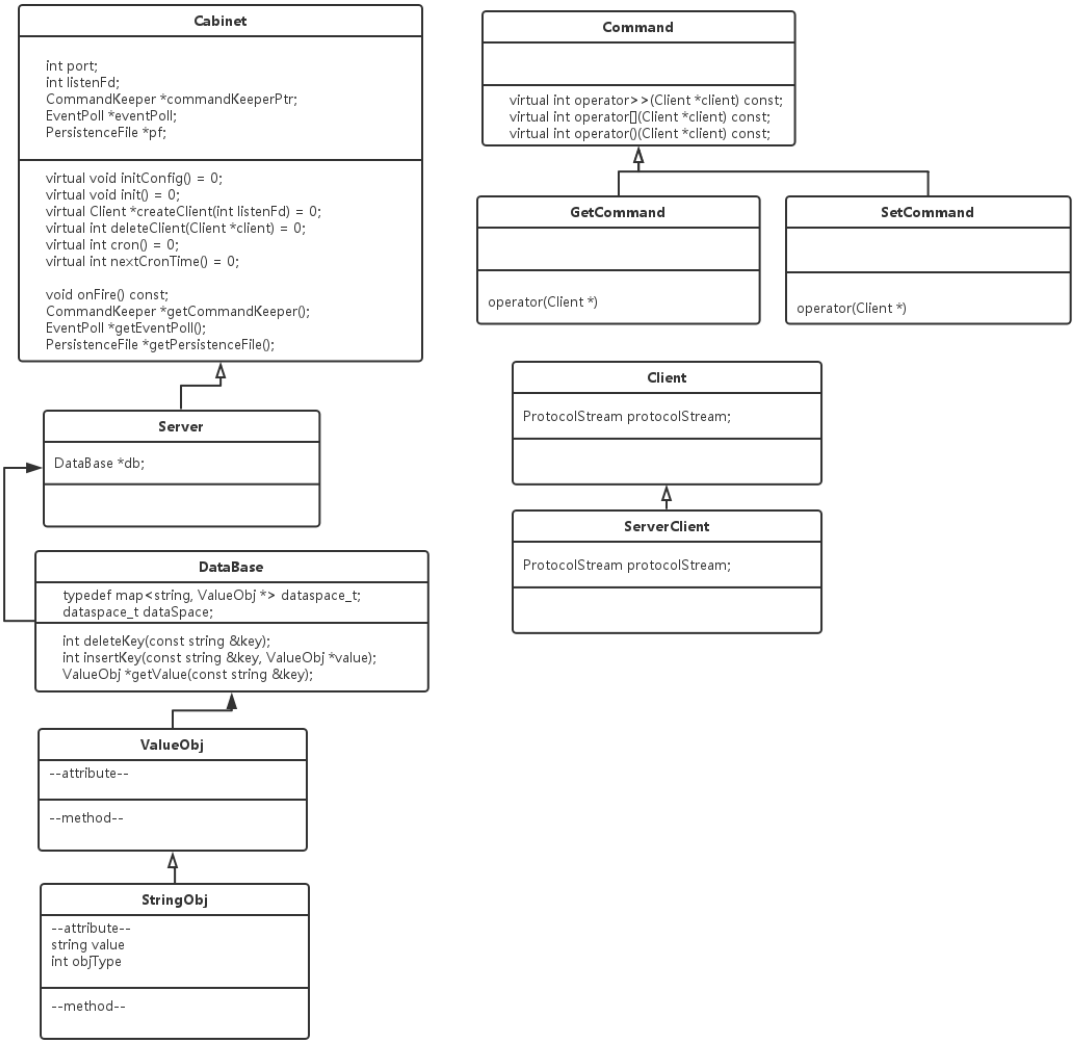

图 5.2 数据库 Server 节点的设计及实现

如图 5.2 所示为数据库 Server 节点的设计架构。

每个 Server 节点都有一个 Server 类，继承自 Cabinet 类。

Server 类中包含有 DataBase，为数据库。DataBase 中，含有以数据库键名为键，以 ValueObj 对象为键值的表。

每个数据库键类型均需要继承并实现 ValueObj 类，如图中的 StringObj，为简单的字符键。

对数据库字符键的操作有 GetCommand 以及 SetCommand，均继承自 Command 类，封装了对字符键的操作逻辑。

每个 Server 的客户端都由 ServerClient 节点表示，可能会有用户客户端连接，也有可能是 Raft 节点对 Server 节点的连接。ServerClient 继承自 Client。

5.1.3 Raft 节点的设计及实现

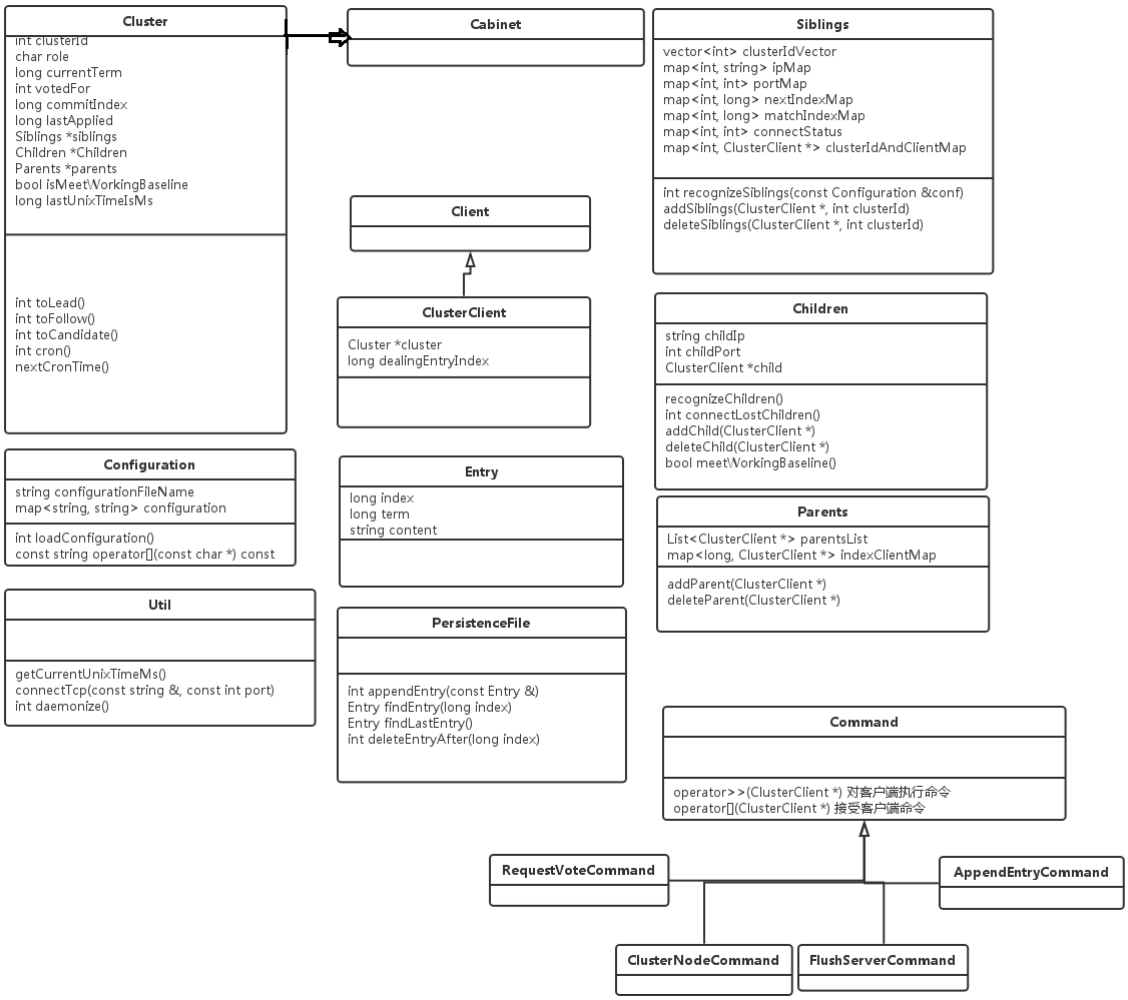

图 5.3 Raft 节点的设计及实现

如图 5.3 所示为 Raft 节点的框架设计图，其中包含有：

(1) Cluster 类，每个 Raft 节点的重要信息，包括节点 id，节点的当前 term，节点的角色，节点当前 term 投票给谁等重要状态。Cluster 类继承自 Cabinet 类；

(2) ClusterClient 类，继承自 Client 类，可能代表三种客户端，包括用户客户端对 Raft 节点的连接，Raft 节点的互相连接，以及 Raft 节点对后端数据库 Server 的连接；

(3) Siblings 类，为 Raft 节点对同级 Raft 节点的管理类，实现对同级 Raft 节点的识别，连接，增加和删除操作；

(4) Children 类，为 Raft 节点对后端数据库的管理类。后端数据库可能为一个或者多个，在 Children 中实现了数据库的分布，同时负责对后端数据库基本的识别、增

加、删除。

(5) Parents 类，负责管理连接到 Raft 节点的用户端，管理用户端的请求和答复；

(6) Entry 类，每个 Entry 对象唯一标示一个持久化日志，含有日志的 term，index 以及日志的内容；

(7) Configuration 类，负责读取并且解析本地配置，供需要在启动中获取最新配置的对象使用；

(8) Util 类，提供通用的函数，比如获取当前 Unix 时间；

(9) RequestVoteCommand 类，Raft 节点在请求投票时以及收到投票之后的逻辑封装，继承自 Command 类；

(10)AppendEntryCommand 类，为 Raft 节点在被选举为 Leader 节点之后向其余节点发送日志或者心跳以及节点收到此命令之后处理逻辑的封装，是逻辑最为重要的一个类。同样继承自 Command 类；

(11)ClusterNodeCommand 类，为 Raft 节点在连接同级 Raft 节点之后需要发送的命令，还包含了接收到连接确认的逻辑；

(12)FlushServerCommand 类，是 Raft 节点在确认本地日志可以提交到状态机（后端数据库）之后，将日志发送给后端 Server 节点的逻辑封装，也继承自 Command 类；

5.2 Cabinet 应用实例

Cabinet 一共有四个可执行文件，分别为：

(1) cabinet-cli，Cabinet 客户端，负责连接服务器节点；

(2) cabinet-server，Cabinet 数据库端，提供数据库服务；

(3) cabinet-cluster，Cabinet 的 Raft 节点，负责集群的一致性服务；

(4) control-cluster.sh 脚本，可以方便的管理集群，开启、关闭节点；

接下来演示 Cabinet 的使用实例。

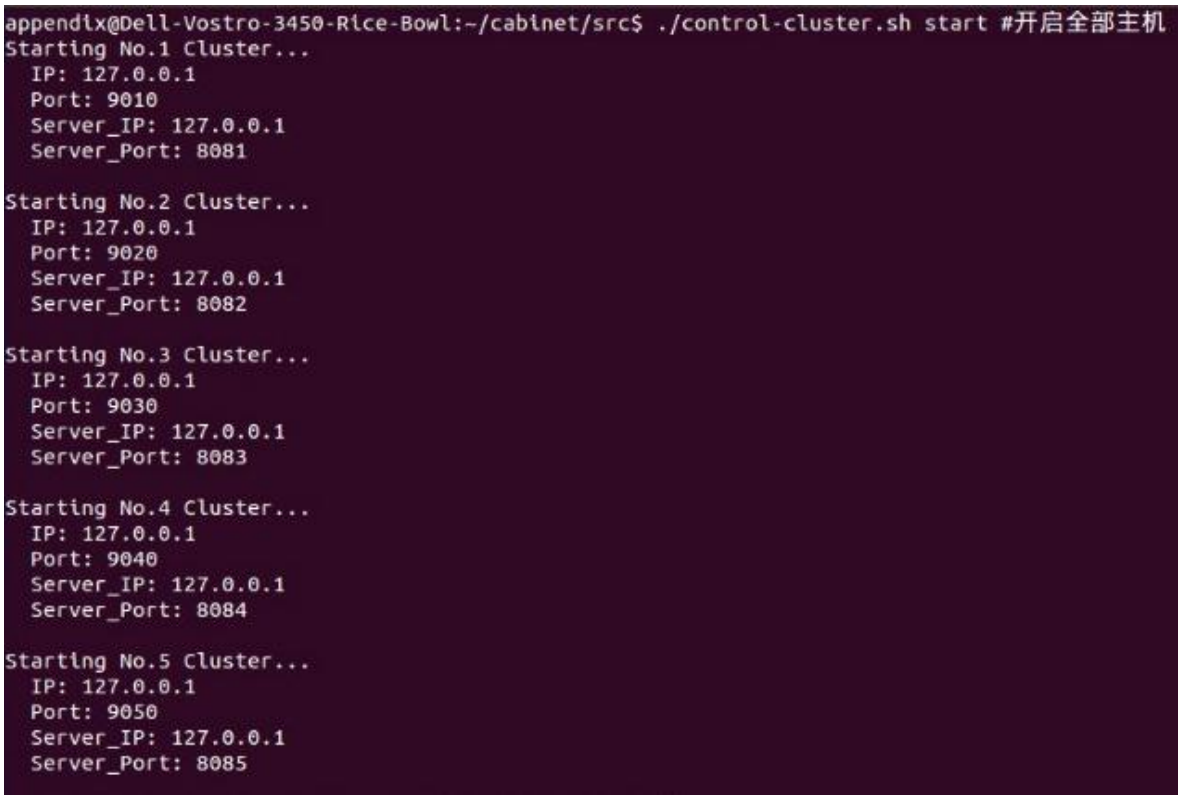

图 5.4 Cabinet 使用示例 1

如图 5.4 所示，运行 ./constol-cluster start 命令，启动集群的节点。

根据配置，一共启动了五个 Raft 节点，同时 Raft 节点之后连接一定数量的数据库。

图 5.5 Cabinet 使用示例 2

如图 5.5 所示，通过运行 ./cabinet-cli 9010 命令，连接到主机的 9010 端口，既连接到第一个 Raft 节点，同时设置 foo 键的值为 bar。

接下来关闭两个主机，既集群中只剩下 3 个主机。

在进行测试，连接到第三个集群节点，集群依旧能够提供服务，集群的可用性得到

充分的体现。

接下来将 foo 键的值设置为 no-foo-anymore，便于之后的演示。

```
appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$ ./cabinet-cli 9050 #试试重定向功能
127.0.0.1:9050>get foo
redirect
leaderip
127.0.0.1
leaderport
9030
127.0.0.1:9030>get foo
no-bar-anymore
127.0.0.1:9030>^C
```

图 5.6 Cabinet 使用实例 3

当前集群的 Leader 节点为第三个节点，如果客户端试图通过其他节点使用集群服务，应该会被重定向至 Leader 节点。

图 5.6 中客户端连接到第五节点，试图使用集群服务，成功被重定向到第三个节点，获取到了 foo 键的最新值。

```
appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$ ll *pf* #查看此时的本地持久化文件，被关闭的主机的持久化文件会比较小
-rw-rw-r-- 1 appendix appendix  82  4月  6 22:47 cabinet-cluster.pf.1
-rw-rw-r-- 1 appendix appendix  82  4月  6 22:47 cabinet-cluster.pf.2
-rw-rw-r-- 1 appendix appendix 176  4月  6 22:50 cabinet-cluster.pf.3
-rw-rw-r-- 1 appendix appendix 176  4月  6 22:50 cabinet-cluster.pf.4
-rw-rw-r-- 1 appendix appendix 176  4月  6 22:50 cabinet-cluster.pf.5
appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$
appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$ ./control-cluster.sh stop 3 #再关闭一个，集群不能正常使用
Killing No.3 Cluster...
  pid=2761
  port=9030
  server_pid=2758
  server_port=8083

appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$ ./cabinet-cli 9040
127.0.0.1:9040>get foo
cluster is not available now
```

图 5.7 Cabinet 使用实例 4

集群此时五个节点中的前两个节点已经被下线，图 5.7 中通过查看本地持久化日志，可以看出第一、二个节点的持久化日志明显小于后三个节点。

此时再关闭一个数据库节点，集群在线主机数量小于一半，无法正常提供服务。

```
appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$ ./control-cluster.sh start 1 #重新开启关掉的集群
Starting No.1 Cluster...
  IP: 127.0.0.1
  Port: 9010
  Server_IP: 127.0.0.1
  Server_Port: 8081

appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$ ./control-cluster.sh start 2 #重新开启关掉的集群
Starting No.2 Cluster...
  IP: 127.0.0.1
  Port: 9020
  Server_IP: 127.0.0.1
  Server_Port: 8082

appendix@Dell-Vostro-3450-Rice-Bowl:~/cabinet/src$ ./control-cluster.sh start 3 #重新开启关掉的集群
Starting No.3 Cluster...
  IP: 127.0.0.1
  Port: 9030
  Server_IP: 127.0.0.1
  Server_Port: 8083
```

图 5.8 Cabinet 使用实例 5

如图 5.8，重新开启集群中被关闭的节点。

图 5.9 Cabinet 使用实例 6

如图 5.9，此时重新连接集群，Leader 节点为集群的第五个节点，集群正常提供服务。

由于之前被关闭的第一节点不是 Leader 节点，我们无法通过第一节点的数据库值，所以绕过第一节点直接连接其后端数据库。

在第一节点被关闭之前，foo 键的值为 bar，在第一节点被关闭之后，foo 键的值被改为 no-bar-anymore。在节点被恢复上线之后，foo 键的最新值应该被传输过来。

通过直接查看后端数据库，foo 键的最新值已经传输到第一个节点，集群的一致性得到验证。

此时再看本地的持久化日志，五个节点的本地持久化日志大小一致。

第六章 总结与展望

这次毕设让我对大学学习的软件编程知识有了较为全面的认识，在实践中付出了汗水并收获了喜悦。

本文所论述 Cabinet 分布式数据库集群的应用所实现的功能如下：

(1)实现了基本的 NoSql 数据库；

(2)实现了简单的字符串键及其赋值和取值操作；

(3)实现了 Raft 协议的主要内容，包括集群选主，日志传输，保证了集群的一致性和可用性；

(4)实现了数据库分布，使数据库分布到多个主机但使用起来如本地数据库；

(5)实现了单线程+I/O 多路复用的网络架构；

(6)实现了定制的网络协议；

需要改进的

(1) 实现网络框架的过程比较耗时费力，可以使用已有的库，如 muduo；

(2) 实现的网络协议在设计以及实现耗费较多时间精力，之后可以尝试使用 Google 的 Protobuf[15]；

(3) 由于知识储备不够等，此项目的测试主要为集成测试，没有进行完整的单元测试[16]，对代码的质量没有良好的掌控；

[15] 李纪欣, 王康, 周立发,等. Google Protobuf 在 Linux Socket 通讯中的应用[J]. 电脑开发与应用, 2013(4):1-5.

[16] ROY OSHEROVE[以]. 单元测试的艺术[M]. 人民邮电出版社, 2014.

致　谢

在此论文即将完成之际，我要向我的指导老师王鹏伟老师表示深深的谢意。从论文的选题到完成，王老师给予了我很大帮助，并提出许多宝贵的意见。正是因为王老师“授之以渔”的教导方法，我才可以深入地学习软件编程方面的知识。在此，对王老师表示深深地谢意！同时，还要感谢我家人，同学对我的关心和帮助!

参考文献

主要参考书目:

[1] 杨传辉. 大规模分布式存储系统[M]. 机械工业出版社, 2013.

[2] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[J]. Draft of October, 2013.

[3] 皮雄军. NoSQL 数据库技术实战[M]. 清华大学出版社, 2015

[4] 姚林, 张永库. NoSQL 的分布式存储与扩展解决方法[J]. 计算机工程, 2012, 38(6):40-42.

[5] 王心妍. Memcached 和 Redis 在高速缓存方面的应用[J]. 无线互联科技, 2012(9):8-9.

[6] 倪超. 从 Paxos 到 Zookeeper[M]. 电子工业出版社, 2015.

[7] Lamport L. The part-time parliament[J]. Acm Transactions on Computer Systems, 1998, 16(2):133-169.

[8] Lamport L. Paxos Made Simple[J]. Acm Sigact News, 2001, 32(4).

[9] Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web[C]// Twenty-Ninth ACM Symposium on Theory of Computing. ACM, 1997:654-663.

[10] (美)史蒂文斯(Stevens, W. R.). UNIX 网络编程[M]. 人民邮电出版社, 2010.

[11] W.RICHARD STEVENS[美]. UNIX 环境高级编程[M]. 人民邮电出版社, 2014.

[12] 陈硕. Linux 多线程服务端编程[M]. 电子工业出版社, 2013.

[13] W.RICHARD STEVENS[美. TCP/IP 详解(卷 1)--协议[M]. 机械工业出版社, 2011.

[14] (美)福勒 著;侯捷 熊节. 重构-改善既有代码的设计[M]. 中国电力出版社, 2003.

[15] 李纪欣, 王康, 周立发,等. Google Protobuf 在 Linux Socket 通讯中的应用[J]. 电脑开发与应用, 2013(4):1-5.

[16] ROY OSHEROVE[以]. 单元测试的艺术[M]. 人民邮电出版社, 2014.

附 录

1、英文文献及翻译

In Search of an Understandable Consensus Algorithm

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [14, 15] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

Unfortunately, Paxos is quite difficult to understand, inspite of numerous attempts to make it more approachable. Furthermore, its architecture is unsuitable for building practical systems, requiring complex changes to create an efficient and complete solution. As a result, both system builders and students struggle with Paxos.

After struggling with Paxos ourselves, we set out to find a new consensus algorithm that could provide a better foundation for system building and education. Our approach was unusual in that our primary goal was under-standability: could we define a consensus algorithm and describe it in a way that is significantly easier to learn than Paxos, and that facilitates the development of intuitions that are essential for system builders? It was important not just for the algorithm to work, but for it to be obvious why it works. In addition, the algorithm needed to be complete enough to cover all the major issues required for an implementation.

The result of this work is a consensus algorithm called Raft. In designing Raft we applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety) and state space reduction (Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, students were able to answer questions about Raft 23%better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms(most notably, Oki and Liskov's View stamped Replication [28, 21]), but it has several novel features:

• Strong leader: Raft uses a stronger form of leadership than other consensus algorithms. For

example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
• Leader election: Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
• Membership changes: Raft's mechanism for changing the set of servers in the cluster uses a novel joint consensus approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. It is simpler and more understandable than other algorithms; it is described completely enough to meet the needs of a practical system; it has several open-source implementations; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

The remainder of the paper introduces the replicated state machine problem(Section 2), discusses the strengths and weaknesses of Paxos (Section 3), describes our general approach to understandability (Section 4), presents the Raft consensus algorithm (Sections 5-8), evaluates Raft (Section 9), and discusses related work (Section 10).

2 Achieving fault-tolerance with replicated state machines

Consensus algorithms typically arise in the context of replicated state machines [34]. In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines are used to solve a variety of fault-tolerance problems in distributed systems. For example, large-scale systems that have a single cluster leader, such as GFS [7], HDFS [35], and RAMCloud [29], typically use a separate replicated state machine to manage leader election and store configuration information that must survive leader crashes. Examples of replicated state machines include Chubby [2]and ZooKeeper [10].

Replicated state machines are typically implemented using a replicated log, as shown in Figure 1. Each server stores a log containing a series of commands, which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs.

Keeping the replicated log consistent is the job of the consensus algorithm. As shown in Figure 1, the consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly-reliable state machine.

Consensus algorithms for practical systems typically have the following properties:
• They ensure safety (never returning an incorrect result)under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering.
• They are fully functional (available) as long as any majority of the servers are operational and can communicate with each other and with clients. Thus, atypical cluster of five servers

can tolerate the failure of any two servers. Servers are assumed to fail by stopping; they may later recover from state on stable storage and rejoin the cluster.
• They do not depend on timing to ensure the consistency of the logs: faulty clocks and extreme message delays can, at worst, cause availability problems.
• In the common case, a command can complete as soon as any majority of the cluster has responded to a single round of remote procedure calls; a minority of slow servers need not impact overall system performance.

探究容易理解的一致性算法

绪论

Raft 是一种管理复制日志的一致性算法。它能产生等同于 Paxos 的效果，而且效率较 Paxos 更高，但它的结构跟 Paxos 有所不同。而正是这种不同使得 Raft 相比 Paxos 更加容易理解，并且为构建使用的系统提供了一个更好的基础。为了加强可理解性，Raft 协议分离了一致性协议的关键部分，比如 Leader 选举，日志复制，以及安全性，而且它强调内聚性，极大地降低了需要考虑的状态。从用户使用调查中可以得出 Raft 相比 Paxos 对于学生来说更加容易学习。Raft 同时包含了一个新的机制，用于改变集群的成员，该机制使用了多数重叠来保证安全。

1. 介绍

一致性协议允许一些机器在同一个集群中运行，而且在某些成员崩溃下线的时候依旧能够保证整体正常运行。因此，一致性协议在构建高可用的大规模软件系统中扮演了重要的角色。Paxos 在过去的几十年中统治了一致性协议的领域，具体表现为大多数一致性系统的实现都是基于 Paxos，或者是受其影响，而且 Paxos 也成为了教授学生关于一致性的重要工具。

不幸的是，尽管许多人尝试过使 Paxos 变得更加可理解，但是 Paxos 依旧是一个及其难以理解的算法。不仅如此，Paxos 的架构不适合用来构建使用的系统，因为它需要复杂的变化来构建一个高效且完整的解决方案。因此，不管是系统构建者或者是学生都被 Paxos 所困扰。

在被 Paxos 困扰了许久之后，我们开始寻求一个新的一致性算法，可以给系统构建以及教学提供一个更好的基础。我们的方案十分与众不同，因为我们把可理解性放在首位。而可理解性既是我们可否定义一个算法而且用比 Paxos 更加简单的方式把它描

述出来，同时帮助系统构建者更加容易形成对这个算法的认识。不仅让算法运行起来很重要，让人理解为什么算法能够运行也很重要。另外，算法也需要足够完整到能够涵盖在实现中可能会面临的问题。

这一系列的工作的成果就是 Raft 协议。在设计 Raft 的过程中，我们使用了一系列特殊的方法来实现可理解性，包括分解（Raft 分解了 Leader 选举，日志复制，安全性）和减少状态空间（Raft 减少了不确定性和服务之间不连续的情况）。在一个涉及两个学校 43 个学生的用户调查中，学习 Raft 的学生比学习 Paxos 的学生能够多回答 23%的问题，表明 Raft 明显比 Paxos 更加容易理解。

Raft 在一些方面跟已有的一致性算法有所相似，但其有许多突出的方面：

- 强 Leader：Raft 相对其他一致性算法使用了更加强势的主节点。比如，在 Raft 中，日志复制只能从 Leader 到其他节点。这简化了对复制日志的管理并且使得 Raft 更加容易理解。
- Leader 选举：Raft 使用了随机计时器来选举 Leader。这只在所有节点都需要的心跳检测上增加了一点机制，但却能够更快更简单地解决矛盾。
- 成员改变：Raft 的机制使用了一种联合算法来解决改变集群机器的问题，算法使用了两种不同配置中多数重叠部分。这允许了集群在改变配置的时候依旧能够正常工作。

我们相信 Raft 相比包括 Paxos 在内的其他一致性算法，不管是在教学方面还是在系统构建方面都更加出色。它更加简单而且更加容易理解；它完整描述了构建完整系统需要的内容；它有多个开源实现；它的安全性已经被正式定义并且证明；而且它的效率完全不输其他一致性算法。

论文的其余部分介绍了复制状态机的问题（第二章），讨论了 Paxos 的优缺点（第三章），描述了我们为了实现可理解性的方法（第四章），展示了 Raft 算法（第五至第八章），评价了 Raft 算法（第九章），还讨论了相关的工作（第十章）。

2. 在复制状态机中实现容错

一致性算法就是在解决复制状态机中出现的。在这种方法中，服务器的集合上的状态机计算了同一状态的唯一拷贝，并且能在一些服务器崩溃之后继续运行。复制的状态机用来解决分布式系统中一系列的容错问题。举例来说，大型系统都有一个集群的 Leader 节点，比如 GFS，HDFS 和 RAMCloud，RAMCloud 中使用了不同的复制状态

机来管理 Leader 选举并存储配置信息，从而在 Leader 崩溃之后依旧可以正常存活。复制状态机的例子还有 Chubby 和 ZooKeeper。

复制状态机一般使用复制日志来实现。每个服务器存储一个日志，其中包含了一系列的命令，而服务器中的状态机按顺序执行这些命令。每个日志包含相同的内容并且按照相同的顺序，所以状态机按照相同的顺序执行这些命令。因为状态机是确定的，所以每个状态机都计算相同的状态并且输出相同的内容。

保持复制日志的一致是是一致性算法的任务。服务器中的一致性模型接受客户的命令并且将他们加入自己的日志。服务器上的一致性模型相互通信，来保证每个日志最终包括了一样的请求并且按照一样的顺序，即使其中的某些服务器失效了。一旦日志被正确地传输，每个服务器的状态机按照日志顺序执行这些日志，并且将输出返回给客户端。因此，这些服务器组成了一个单一的高可用的状态机。

对于使用的系统来说，一致性协议应该有一下特性：

- 能够在所有非拜占庭环境下保证安全，这些环境包括网络延迟，分区，丢包，重复以及乱序。
- 只要大多数服务器能够执行而且能互相通信以及跟客户端通信，那么系统就要是能够使用的。所以，在一个五个机器的集群中需要容忍其中两个机器崩溃下线的情况。服务器可以通过停止来下线，而服务器可能之后从可靠存储中恢复状态并且再次上线。
- 服务器不依靠计时来保证日志的一致性，因为在最坏情况下，错误的时钟以及极限情况下的消息延迟可能导致可用性问题。
- 在一般情况下，一个命令只要在一轮远程服务调用中被大多数节点响应即可算完成，因此少数运行较为缓慢的节点不会影响挣个系统的运行效率。