**Lab2: Raft**

| | | | |
|---|---|---|---|
| **笔记本：** | Distributed System | | |
| **创建时间：** | 2018/10/18 9:37 | **更新时间：** | 2018/12/13 16:26 |
| **作者：** | 956318206@qq.com | | |

# 目标：一个容错（fault-tolerant）的key/value存储系统

Lab2：实现Raft，一种复制状态机协议。
Lab3：在Raft之上构建一个key/value服务
Lab4：在多个复制状态机（replicated state machines）上分片（shard）服务以获得更高的性能

# 容错（fault tolerance）

复制服务（a replicated service）通过在多个副本服务器（replica serves）上存储其状态(即数据)的完整拷贝（complete copies）来实现容错
挑战在于故障（failures）可能导致副本（replicas）保存不同的数据拷贝（differing copies of data）

# Raft

Raft管理服务的状态副本，特别是它可以帮助服务找出失败后的正确状态。

## 复制状态机（a replicated state machine）

Raft实现了一个复制状态机。
(1) 它将客户端请求组织为一个称为日志（log）的序列，并确保所有的副本（replicas）都同意日志的内容
(2) 每个副本按日志中出现的顺序执行客户端请求，将这些请求应用到副本的服务状态的本地拷贝

## 正确性：

由于所有的活跃副本（live replicas）都看到相同的日志内容，因此它们都以相同的顺序执行相同的请求，因此继续持有相同服务状态

## 容错：

(1) 如果服务器出现故障但稍后恢复，Raft负责使其日志更新
(2) 只要至少大多数服务器处于活跃状态并且可以互相通信，Raft将继续运行
(3) 如果没有这样的多数，Raft将不会取得任何进展，但只要大多数能够再次通信，它就会从中断的地方继续运行

## 日志条目（log entries）

(1) 实现的Raft接口将支持无限长的编号命令序列（an indefinite sequence of numbered commands），也称为日志条目（log entries）
(2) 条目（entries）用索引号（index numbers）编号
(3) 最终将提交具有给定索引的日志条目
(4) 此时，实现的Raft应该将日志条目发送到更大的服务以便执行

# 任务

实现extended-raft论文中描述的大多数Raft设计，包括保存持久化状态（persitent state）并在节点故障后读取它并重新启动。
不需要实现集群成员关系（cluster menbership changes）（Section 6）和日志压缩/快照（log compaction/snapshotting）（Section 7）。
实现分为三个部分：

Part 2A：领导选举。实现领导者选举和心跳（leader election and heartbeats）（借助不包含日志条目的追加条目RPCs（AppendEntries RPCs））
Part 2B：日志复制。
Part 2C：保存持久化状态

- # 代码

**通过向raft/raft.go添加代码来实现Raft。**在该文件中你将找到一些框架代码（a bit of skeleton code），以及如何发送和接收RPCs的例子。
你的实现必须支持以下接口，测试代码和（最终）你的key/value服务器将使用这些接口。你能从raft.go中的注释里找到更多详细信息。

```
// create a new Raft server instance:
rf := Make(peers, me, persister, applyCh)

// start agreement on a new log entry:
rf.Start(command interface{}) (index, term, isleader)

// ask a Raft for its current term, and whether it thinks it is leader
rf.GetState() (term, isLeader)

// each time a new entry is committed to the log, each Raft peer
// should send an ApplyMsg to the service (or tester).
type ApplyMsg
```

一个服务调用`Make(peers, me, ...)`来创建一个Raft对等点（a Raft peer）。“`peers`”参数是Raft对等点（包括这个）的网络标识符数组，用于labrpc RPC。“`me`”参数是对等点数组中该对等点的索引。
Start(command)要求Raft启动处理以便将命令追加到复制日志。Start()应该立即返回，而不必等待日志追加完成。
该服务期望你的实现为每一个新提交的日志条目（each newly committed log entry）发送一个`ApplyMsg`到`Make()`函数的“`applyCh`”参数。
你的Raft对等点应该使用我们提供给你的Go语言包labrpc来交换RPCs。它以Go语言的rpc库为模型（is modeled after Go's rpc library），但是内部使用Go channels而不是sockets。raft.go包含一些发送RPC（`sendRequestVote()`）和处理传入的RPC（`RequestVote()`）的示例代码。你必须使用labrpc而不是Go的RPC包的原因是，测试代码告诉labrpc延迟RPCs、重新排序它们、删除它们来模拟具有挑战性的网络条件，在这些条件下你的代码应该正确工作（work correctly）。
**不要依赖对labrpc的修改，因为我们将使用分发的labrpc来测试你的代码。**
你的第一个实现可能不够清晰（clean）以使你很容易地推断出它的正确性。**给自己足够的时间重写你的实现以至于你可以很容易推断出它的正确性。**后序的实验建立在这个实验的基础上，所以在实现上做好工作非常重要。

- # Part 2A

提示：

- **往raft.go中的Raft结构里增加任何你需要的状态。你还需要定义一个结构来保存有关每个日志条目的信息。**你的代码应该尽可能遵从论文中的图2。
- **填充`RequestVoteArgs`和`RequestVoteReply`结构。修改`Make()`函数来创建一个后台goroutine，当有一段时间内没有收到来自其他对等点的消息时，通过发送`RequestVote` RPCs来周期性地启动（kick off）领导者选举。**通过这种方式（this way）一个对等点将了解谁是领导者，如果已经有领导者，或者成为一个领导者本身。实**现`RequestVote()` RPC处理程序，以便服务器可以为其他服务器投票。**
- **要实现心跳（heartbeats），请定义一个`AppendEntries` RPC结构（尽管你可能还不需要所有参数），并让leader定期发送它们。编写一个`AppendEntries` RPC的处理程序方法，该方法重置选举超时**，以便其他服务器在一个服务器已经被选中时不会作为leader向前迈进。
- **确保不同对等点的选举超时不总是同时发生**，否则所有的对等点都只会给自己投票，没有谁会成为leader。
- **测试代码要求leader每秒发送心跳RPCs不超过10次。**
- **测试代码要求你的Raft在旧的领导者失败后5秒内要选出一个新的领导者**（如果大多数的对等点仍能能够通信的话）。**然而，请记住，在瓜分投票（a split vote）的情况下，领导选举可能需要多轮**（如果信息包丢失或者候选者不幸选择了相同的随机回退时间（random backoff times），可能会发生这种情况）。**你必须选择足够短的选举时间（并且也是心跳间隔（heartbeat intervals））**，使得即使需要多轮，一次选举也非常可能在不到5秒的时间内完成。
- 论文5.2节提到选举超时在150到300ms的范围内。**只有当领导者比每150ms发送一次以上的心跳时，这个范围才有意义。**因为测试代码将你的心跳限制在每秒10次，所以**你必须使用比论文中150到300ms更长的选举超时，但**

**是不能太长**，因为这样你可能无法在5秒内选出一个领导者。
- **你可能会发现Go的`rand`很有用。**
- **你需要编写定期或者延时时间后执行操作的代码。最简单的方法是创建一个循环调用`ran`的goroutine。**困难的方法是使用Go的`time.Timer`或`time.Ticker`，这些都很难正确使用。
- 如果你对锁（locking）感到疑惑，你可能会发现这个建议很有用。
- 如果你的代码无法通过测试，请再次阅读论文的图2；**领导者选举的完整逻辑分布在（is spread over）图中的多个部分。**
- **调试代码的一个好方法是，当对等点发送或接收消息时插入打印语句，并且使用`go test -run 2A > out`收集输出到一个文件。**然后，通过研究`out`文件中的消息跟踪，你可以确定你的实现在哪里偏离了协议的期望。你可能会发现util.go中的`DPrintf`是有用的，在调试不同的问题时打开或关闭打印。
- **Go RPC只发送名称以大写字母开头（start with captial letters）的struct字段。**子结构也必须有大写的字段名（例如，数组中的日志记录字段）。labgob包会警告你这一点，不要忽略这些警告。
- 你应该使用`go test -race`检查代码，并修复它报告的任何竞争（race）。

# Part 2B

实现领导者和跟随者代码来追加新的日志条目。这将涉及到实现`Start()`，完成AppendEntires RPC结构，发送它们，充实（fleshing out）AppendEntry RPC处理程序（handler），以及推进（advancing）领导者的`commitIndex`。
你的第一个目标应该是通过`TestBasicAgree2B()`测试（在test_test.go中）。一旦你成功了，你应该通过所有的2B测试（go test run -2B）。
提示（Hint）：

- 你将需要实现选举限制（restriction）（论文5.4.1节）
- 不通过Lab 2B早期测试的一个方式是举行（hold）不需要的（un-needed）选举，也就是说，即使现任领导者还存活，而且可以与所有peers交谈，选举也要举行。这可以防止在测试者相信可能达成一致的场景下达成一致。选举计时器（election timer）管理中的bug，或者赢得选举后没有立即（immediately）发出（send out）心跳，可能会造成不必要的选举。
- 你可能需要等编写等待某些事件出现（occur）的代码。不要编写在没有暂停的情况下连续（continuously）执行的循环，因为这会使你的实现慢到测试失败。你可以使用Go的channels，或者Go的条件变量，或者（如果其他方法都失败了）在每个循环迭代中插入一条time.Sleep(10 * time.Millisecond)来有效地等待。
- 根据从组织并发代码中学到的经验（in light of lessons learned about structuring concurrent code），给自己时间重写实现。在以后的实验中，你将感谢自己拥有尽可能清晰（clear）和干净（clean）的Raft代码。如果你有什么想法，你可以重新访问（re-visit）下我们的结构，锁和指南页面。

“ok raft 38.029s”意味着Go测量2B所有的测试花费的时间是实际（挂钟（wall-clock））时间的38.029秒。“user 0m1.460s”意味着代码消耗了1.460秒的CPU时间，即实际（actually）执行指令（而不是等待或睡眠）的时间。**如果你的解决方案用于2B所有测试的实际时间远远超过（much more than）1分钟，或者CPU时间远远超过（much more than）5秒，那么你以后可能会遇到麻烦。**查找花费在睡眠或等待RPC超时的时间，运行时没有睡眠或没有等待条件或channel消息的循环，或发送的大量RPC。

# Part 2C

如果一个基于Raft（Raft-based）的服务器重新启动，它应该从停止的地方（where it left off）恢复（resume）服务。这要求Raft保持在重启后仍然存在的（that survives a reboot）持久状态（persistent state）。论文图2提到了哪些状态（which state）应该是持久的，并且raft.go包含了如何保存（save）和恢复（restore）持久状态的示例。
“真正的（real）”实现可以做到这一点，通过在每次发生变化时将Raft的持久状态写入磁盘，并在重启后重新启动时从磁盘读取最新（latest）保存的状态。你的实现不使用磁盘；相反，它将从Persister对象（见persister.go）保存和恢复持久状态。`Raft.Make()`的调用者提供了初始时持有（initially holds）Raft最近（most recently）持久状态（如何有的话（if any））的Persister。Raft应该从那个Persister初始化它的状态，并且应该在每次状态变化时使用它来保存其持久状态。使用Persister的`ReadRaftState()`和`SaveRaftState()`方法。
通过添加保存和恢复持久状态的代码，来完成raft.go中`persist()`和`readPersist()`函数。你需要将状态编码（encode）（或序列化（serialize））为字节数组（an array of bytes）以便将它传递给Persister。使用我们提供的labgob编码器来完成此操作；请参阅persist()和readPersist()中的注释。labgob来源于（is derived from）Go的gob编码器；唯一的区别是，如果你尝试用小写字段名称（lower-case field names）编码结构，labgob将打印错误消息。
现在你需要确定（determine）在Raft协议中的哪些点（what points in the Raft protocol）你的服务器被要求持久化其状态，并在这些地方插入对persist()的调用。在Raft.Make()中已经有对readPersister()的调用。完成这些之后，你应该通过剩余的测试。你可能想先尝试通过“basic persistence”测试（go test -run 'TestPersist12C'），然后处理（tackle）其余的测试（go test -run 2C）。

- 注意

  为了避免运行内存耗尽（out of memory），Raft必须定期（periodically）丢弃（discard）旧的日志条目，但是直到下个实验之前（until the next lab）你不必（do not have to）担心这个问题。
- 提示
  - 2C的许多测试涉及到服务器故障（failing）和网络丢失RPC请求（requests）或回复（replies）
  - 为了在最后（towards the end）通过一些具有挑战性的（challenging）测试，例如（such as）那些标记为“不可靠的（unreliable）” 的测试，你将需要实现优化，以允许跟随者一次通过多个条目（by more than one entry at a time）回退（back up）领导者的nextIndex。请参阅the extended Raft论文的从第7页底部和第8页顶部（用灰色线标记）开始的描述。论文在细节上含糊不清（vegue）；你将需要填补这些空白（the gaps），可能需要借助6.824的Raft课件（lectures）的帮助。
  - 整套Lab2测试（a full set of Lab 2 tests）（2A+2B+2C）合理的时间消耗（a reasonable amount of time to consume）是4分钟的真实时间和1分钟的CPU时间。