Consensus on Replicated Logs

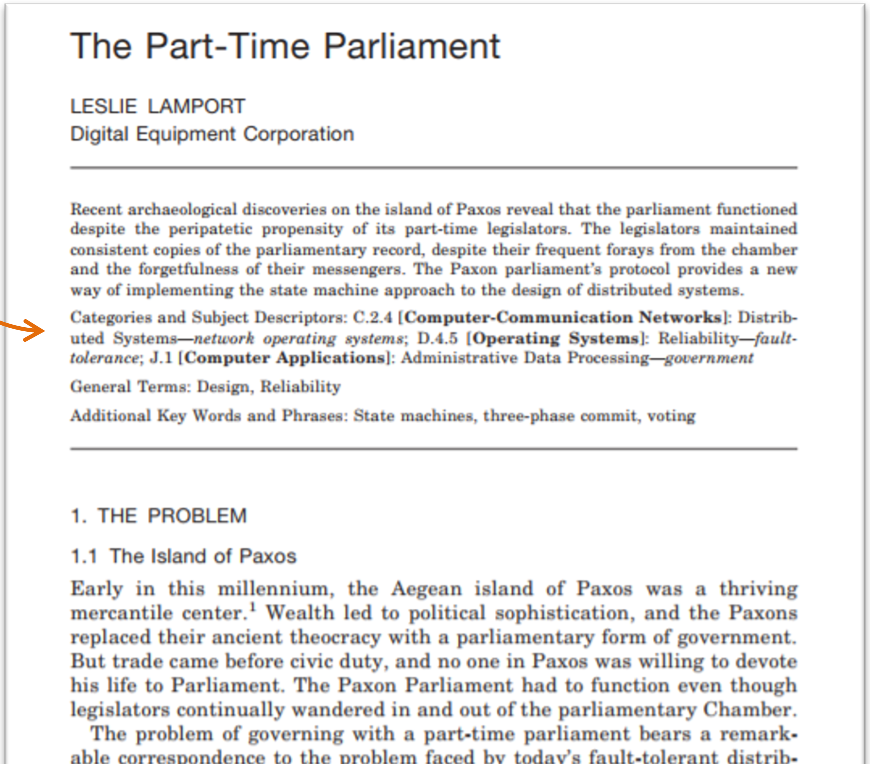
Assignments of Distributed System Course

Based on the Raft paper and Michael Freedman's slides

The Raft Paper

- Diego Ongaro, John K. Ousterhout. In search of an understandable consensus algorithm. ***USENIX ATC 2014***. [Best Paper Award]

Compared with Paxos



The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; J.1 [**Computer Applications**]: Administrative Data Processing—*government*

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

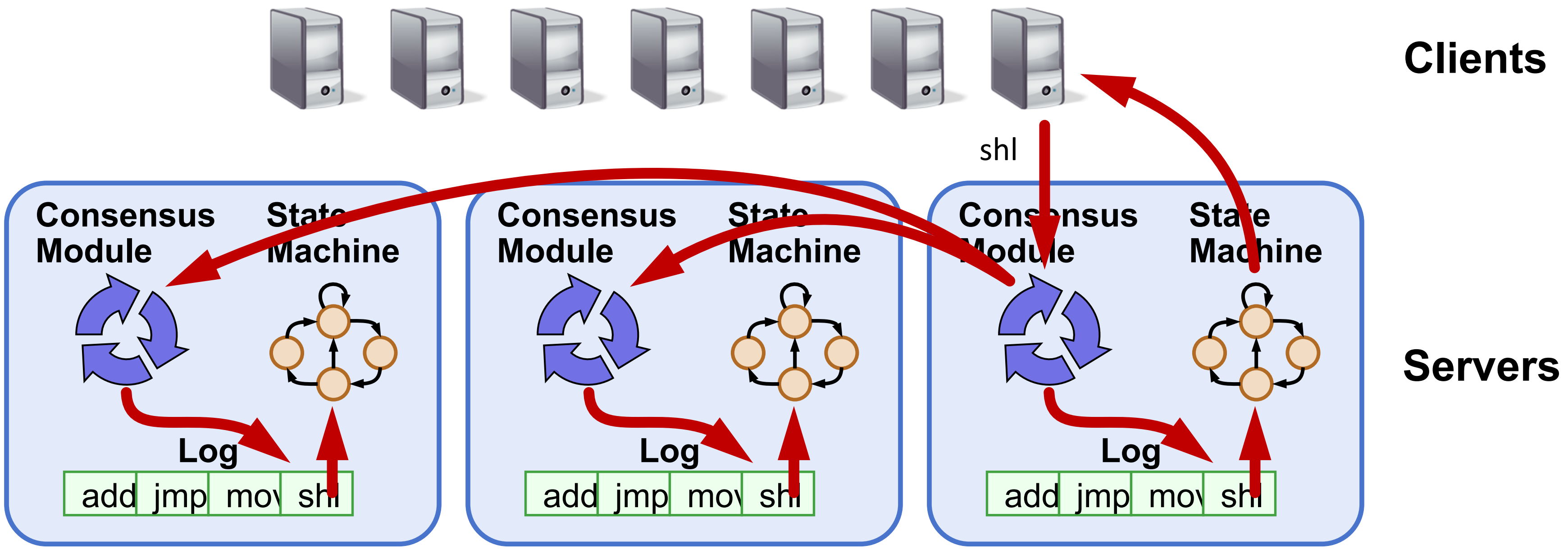
1. THE PROBLEM

1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.[1] Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distrib-

Goal: Replicated Log



- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication

Raft Overview

1. **Leader election**
2. **Normal operation (basic log replication)**
3. **Safety and consistency after leader changes**
4. **Neutralizing old leaders**
5. **Client interactions**

Server States

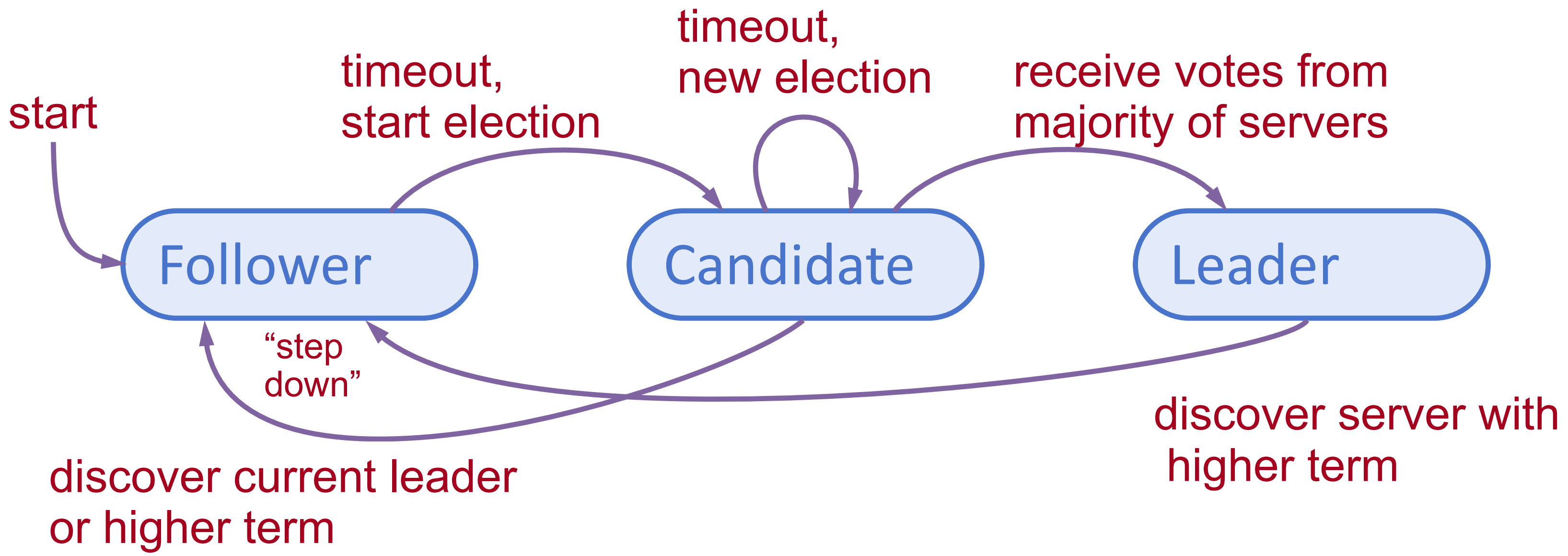
- At any given time, each server is either:
  - **Leader**: handles all client interactions, log replication
  - **Follower**: completely passive
  - **Candidate**: used to elect a new leader

- Normal operation: 1 leader, N-1 followers

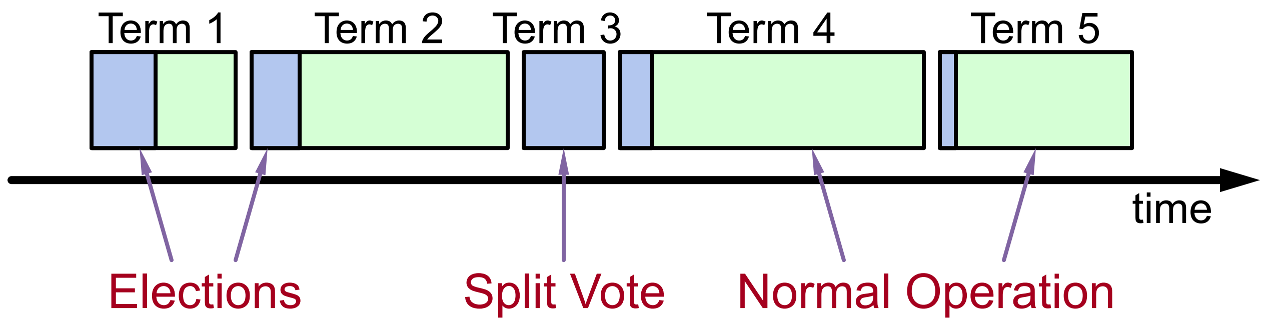
Follower
Candidate
Leader

Liveness Validation

- Servers start as followers
- Leaders send **heartbeats** (empty *AppendEntries* RPCs) to maintain authority
- If **electionTimeout** elapses with no RPCs (100-500ms), follower assumes leader has crashed and starts new election



Terms (aka epochs)



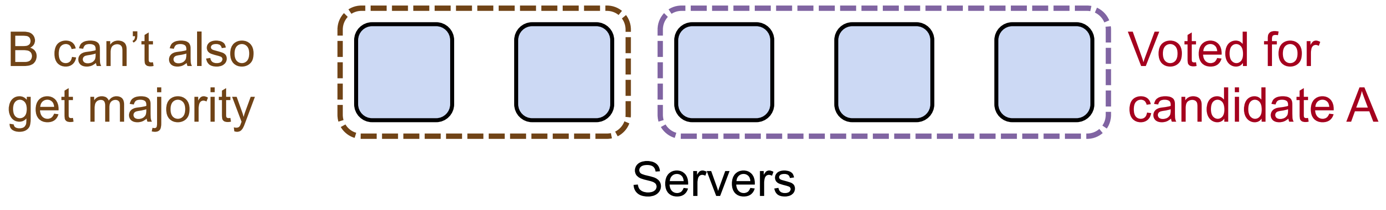
- Time divided into terms
  - Election (either failed or resulted in 1 leader)
  - Normal operation under a single leader
- Each server maintains **current term** value
- **Key role of terms: identify obsolete information**

Elections

- **Start election:**
  - Increment current term, change to candidate state, vote for self
- **Send *RequestVote* to all other servers, retry until either:**
  1. Receive votes from majority of servers:
     - Become leader
     - Send *AppendEntries* heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election

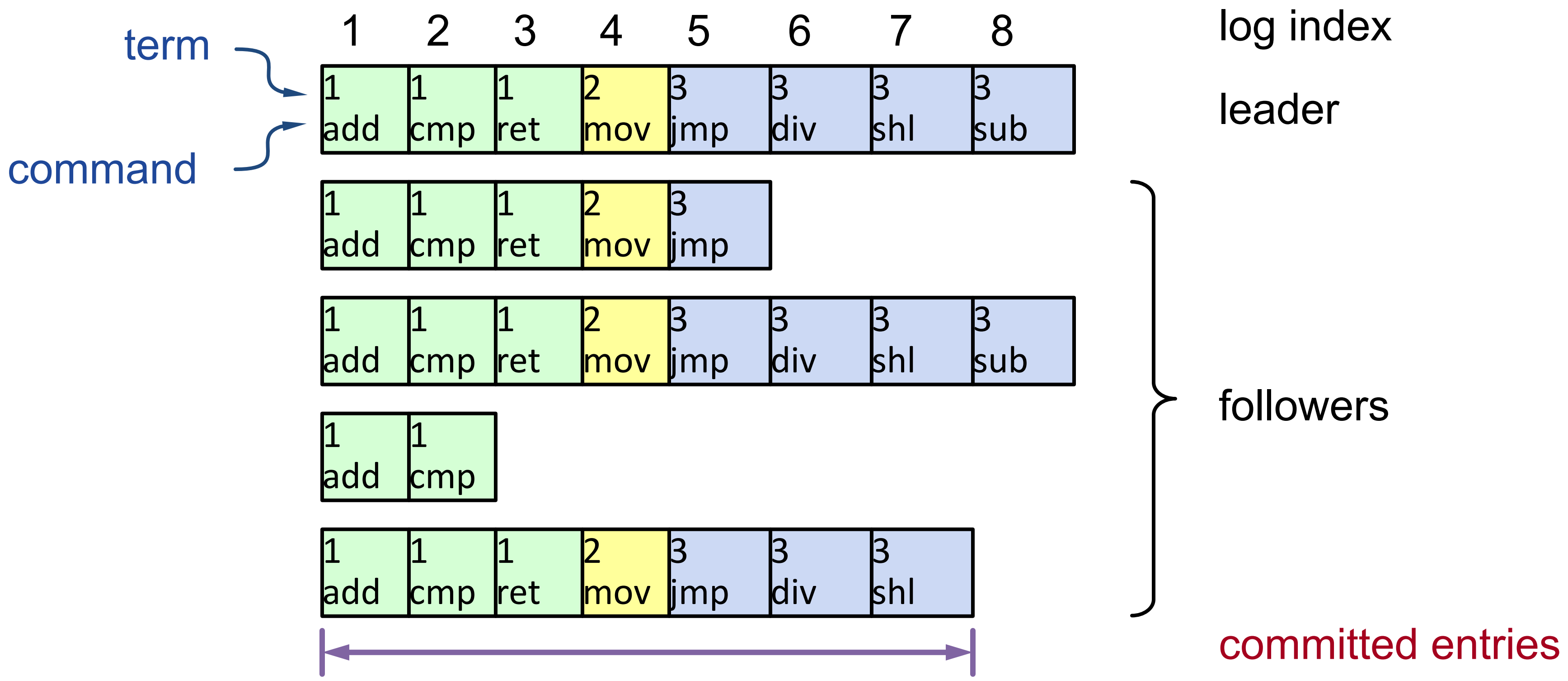
Elections

- **Safety**: allow at most one winner per term
  - Each server votes only once per term (persists on disk)
  - Two different candidates can't get majorities in same term



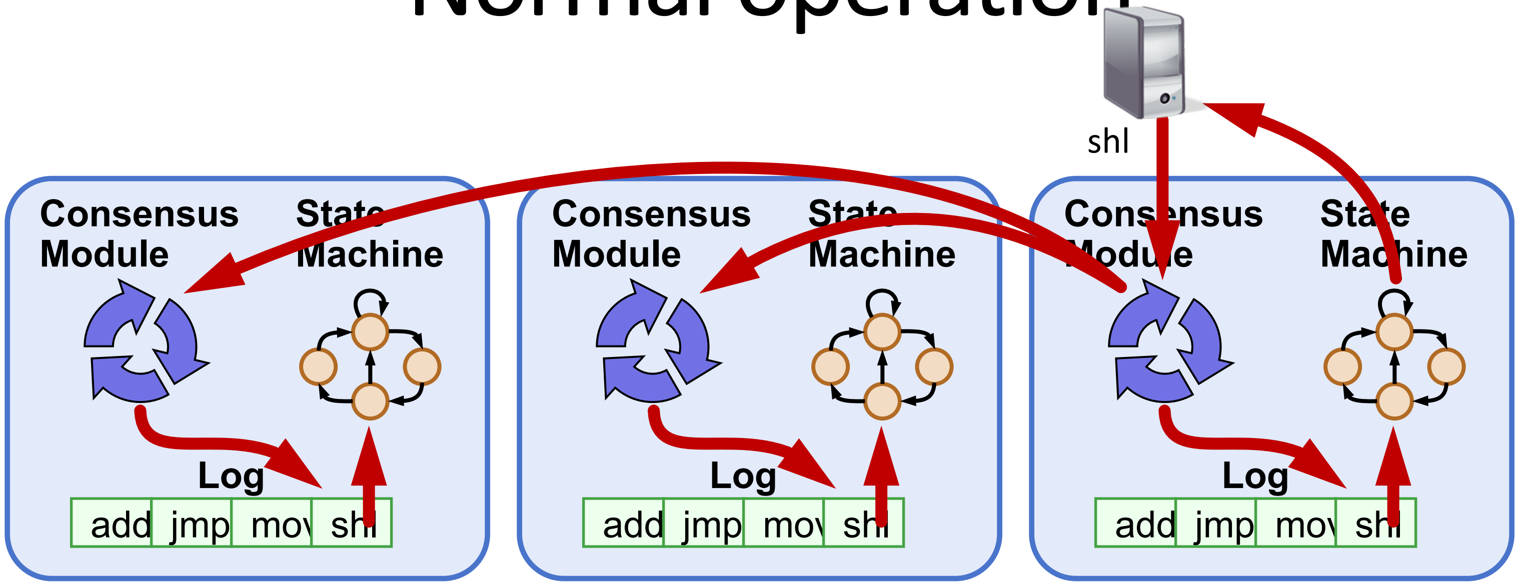
- **Liveness**: some candidate must eventually win
  - Each choose election timeouts ***randomly*** in [T, 2T]
  - One usually initiates and wins election before others start
  - Works well if T >> network RTT

Log Structure



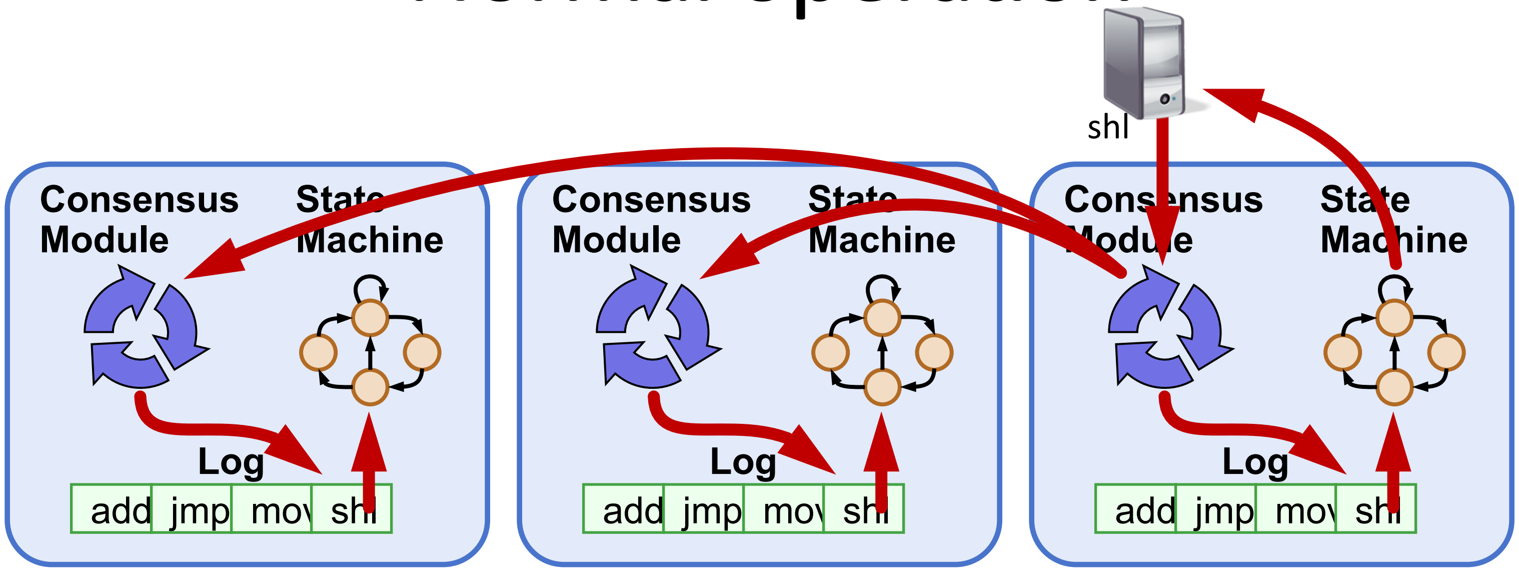
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry **committed** if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

Normal operation



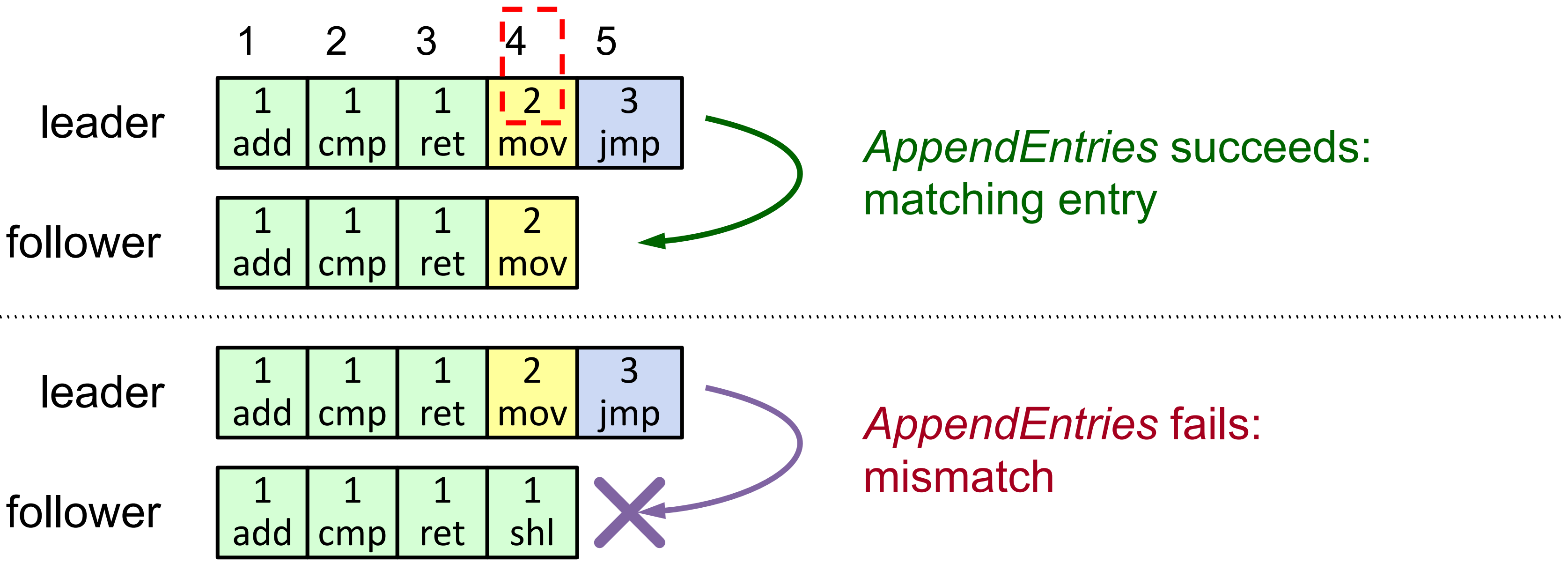
- Client sends command to leader
- Leader appends command to its log
- Leader sends *AppendEntries* RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, sends result to client
  - Leader piggybacks commitment to followers in later *AppendEntries*
  - Followers pass committed commands to their state machines

Normal operation



- Crashed / slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

Log Operation: Consistency Check



- *AppendEntries* has <index, term> of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects
- Implements an **induction step**, ensures coherency

Safety Requirement

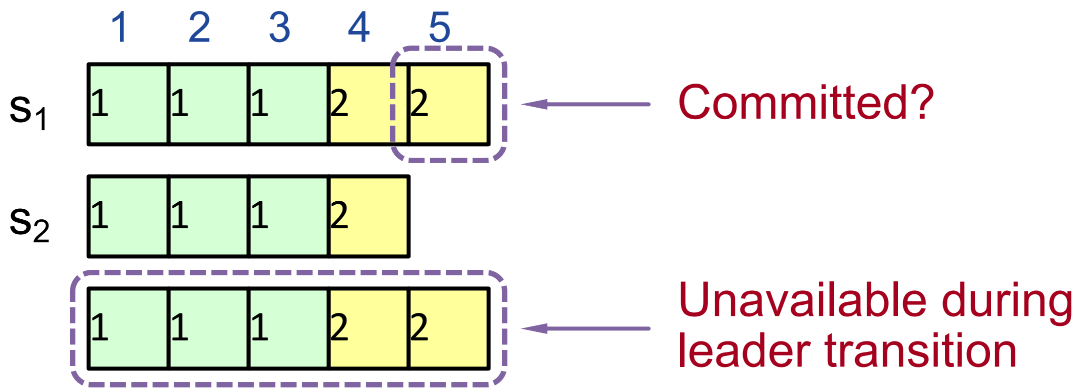
Once log entry applied to a state machine, no other state machine must apply a different value for that log entry

- **Raft safety property:** If leader has decided log entry is committed, entry will be present in logs of all future leaders
- Why does this guarantee higher-level goal?
  1. Leaders never overwrite entries in their logs
  2. Only entries in leader's log can be committed
  3. Entries must be committed before applying to state machine

$$\begin{tikzcd} & \text{Committed} \arrow[dl, bend left] & \longrightarrow & \text{Present in future leaders' logs} \arrow[dr, bend right] & \\ \text{Restrictions on commitment} & & & & \text{Restrictions on leader election} \end{tikzcd}$$

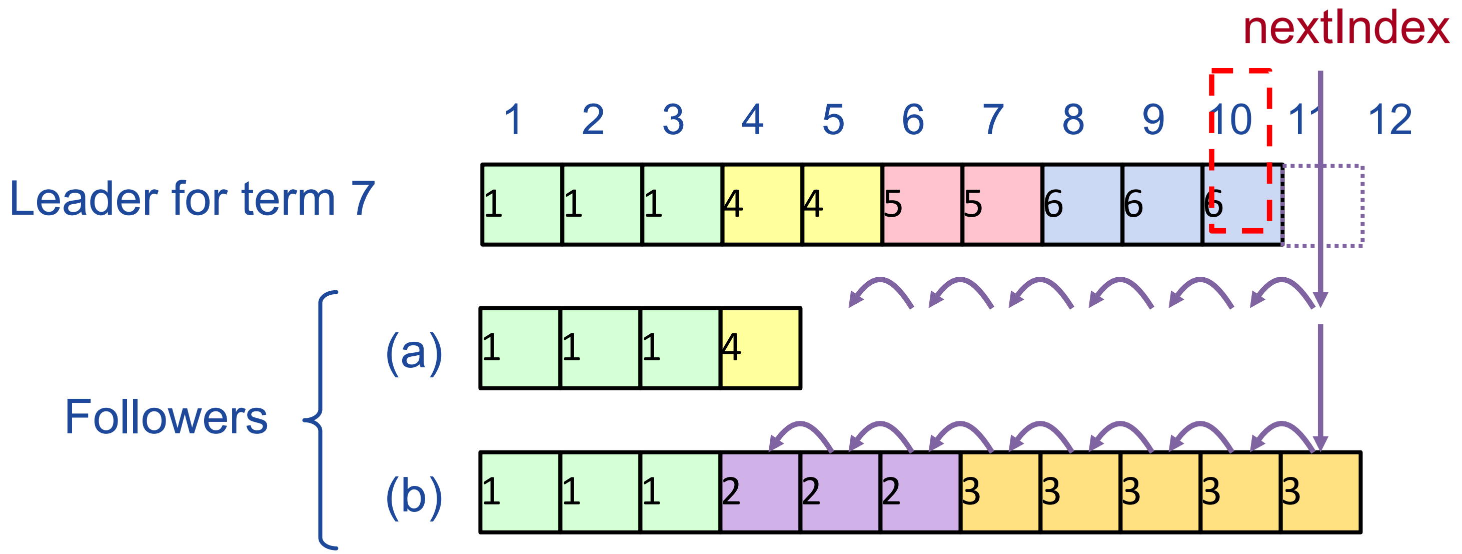
Picking the Best Leader

Can't tell which entries committed!



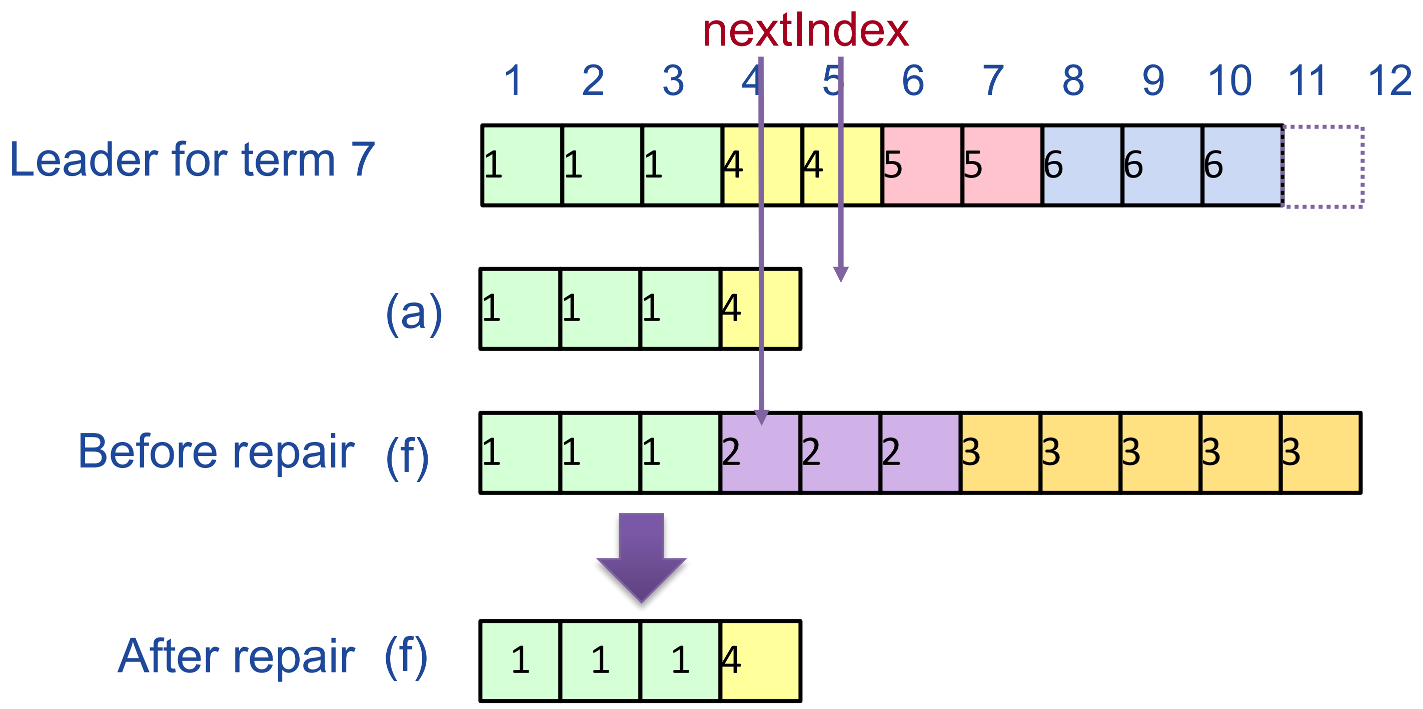
- Elect candidate most likely to contain all committed entries
  - In *RequestVote*, candidates incl. index + term of last log entry
  - Voter V denies vote if its log is "more complete": (newer term) or (entry in higher index of same term)
  - Leader will have "most complete" log among electing majority

Repairing Follower Logs



- **New leader must make follower logs consistent with its own**
  - Delete extraneous entries
  - Fill in missing entries
- **Leader keeps nextIndex for each follower:**
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader’s last index)
- If *AppendEntries* consistency check fails, decrement nextIndex, try again

Repairing Follower Logs



Neutralizing Old Leaders

Leader temporarily disconnected

→ other servers elect new leader

→ old leader reconnected

→ old leader attempts to commit log entries

- Terms used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - Sender's term < receiver:
    - Receiver: Rejects RPC (via ACK which sender processes…)
  - Receiver's term < sender:
    - Receiver reverts to follower, updates term, processes RPC
- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries

Client Protocol

- **Send commands to leader**
  - If leader unknown, contact any server, which redirects client to leader
- **Leader only responds after command logged, committed, and executed by leader**
- **If request times out (e.g., leader crashes):**
  - Client reissues command to new leader (after possible redirect)
- **Ensure exactly-once semantics even with leader failures**
  - E.g., Leader can execute command then crash before responding
  - Client should embed unique ID in each command
  - This client ID included in log entry
  - Before accepting request, leader checks log for entry with same id

Assignment Setup

- Programming in Go
  - We will do all of our assignments in Go, a language developed at Google, but now part of an open source project.

- We believe that Go is an especially suitable language for writing distributed systems for the following reasons:
  - The language is type-safe and garbage collected, avoiding many of the pitfalls of lower-level languages, such as C and C++.
  - Many useful data structures are built into the language, such as resizable arrays and dictionaries.
  - There is a large collection of packages providing access to useful system resources.
  - Go supports a model of concurrency that is cleaner and more abstract than traditional mechanisms, such as Pthreads and Java threads.

Assignment Setup

- Go installation
  - You need the Go environment for the assignments. The version should be at least Go 1.5.
  - There are instructions to install: golang.org
- Tools
  - Three useful tools in the Go ecosystem: Go fmt (https://golang.org/cmd/gofmt/) , Go vet (https://golang.org/cmd/vet), and Golint (https://github.com/golang/lint).
- Go tutorial
  - https://tour.golang.org , go through it!

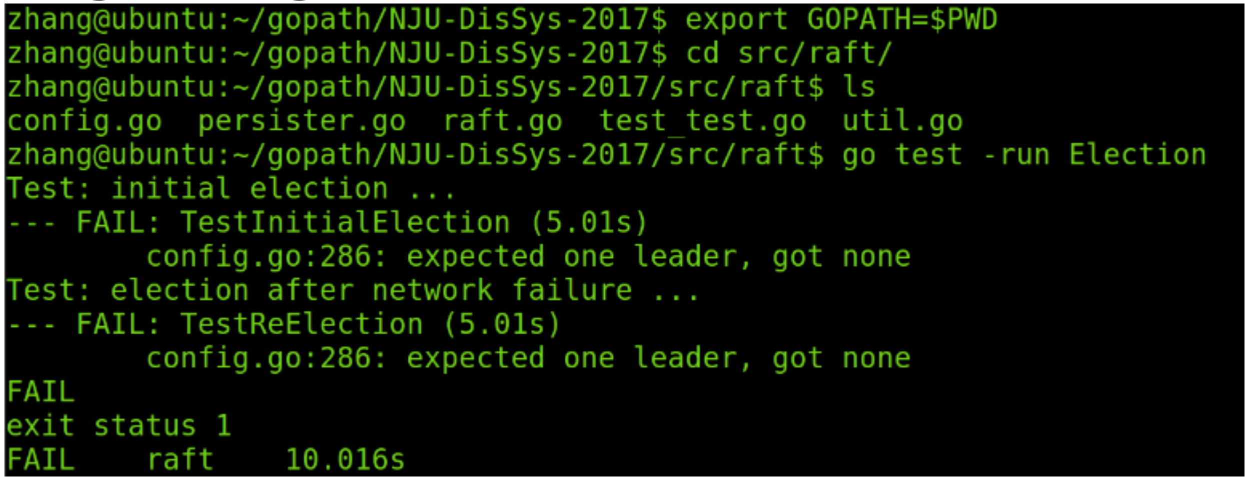
Assignment Setup

- Resources
  - Raft paper
    - Conference version: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf
    - Full version: https://raft.github.io/raft.pdf
  - illustrated Raft guide: http://thesecretlivesofdata.com/raft/

- Software
  - https://github.com/Zhang-Xiaoda/NJU-DisSys-2017
  - Focus primary on the code and tests for the Raft implementation in src/raft and simple RPC-like system in src/labrpc, (read the code in these packages first)

Assignment ***Part I***

- At the beginning:



```
zhang@ubuntu:~/gopath/NJU-DisSys-2017$ export GOPATH=$PWD
zhang@ubuntu:~/gopath/NJU-DisSys-2017$ cd src/raft/
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ ls
config.go  persister.go  raft.go  test_test.go  util.go
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run Election
Test: initial election ...
--- FAIL: TestInitialElection (5.01s)
        config.go:286: expected one leader, got none
Test: election after network failure ...
--- FAIL: TestReElection (5.01s)
        config.go:286: expected one leader, got none
FAIL
exit status 1
FAIL    raft    10.016s
```

- Implement Raft by adding code to raft/raft.go (***only***)
  - find some example code of how to send and receive RPC

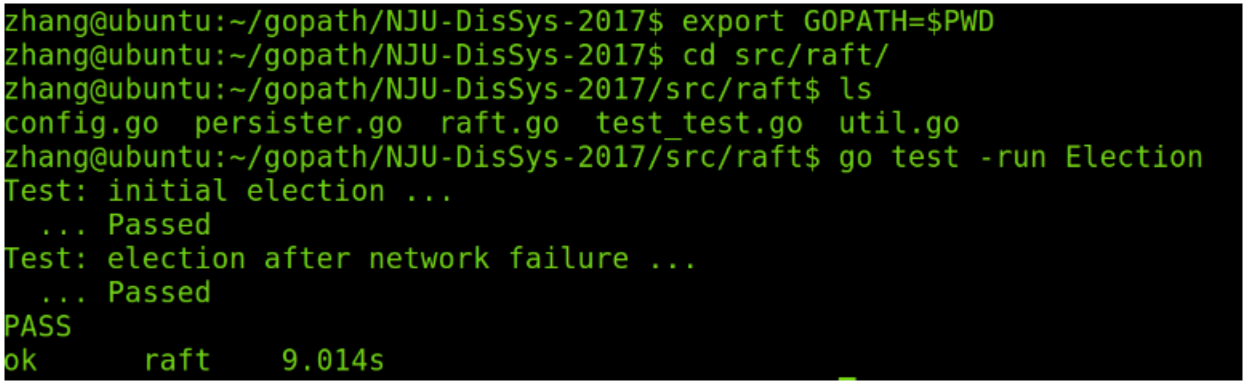
- Your task: Leader election:
  - First task is to fill the *RequestVoteArgs* and *RequestVoteReply* structs

Assignment ***Part I***

- Modify *Make()* to create a background goroutine that starts an election by sending out *RequestVote* RPC when it hasn’t heard from another peer for a while
    - You need to implement *RequestVote* RPC handler so that servers will vote for one another
- To implement heartbeats, you will need to define *AppendEntries* struct (though you will not need any real payload yet), and have the leader send them out periodically
    - Also need to implement *AppendEntries* RPC handler
- make sure the election timeouts **don't** always fire at the same time

Assignment ***Part I***

- At the end:



```
zhang@ubuntu:~/gopath/NJU-DisSys-2017$ export GOPATH=$PWD
zhang@ubuntu:~/gopath/NJU-DisSys-2017$ cd src/raft/
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ ls
config.go  persister.go  raft.go  test_test.go  util.go
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run Election
Test: initial election ...
  ... Passed
Test: election after network failure ...
  ... Passed
PASS
ok      raft    9.014s
```

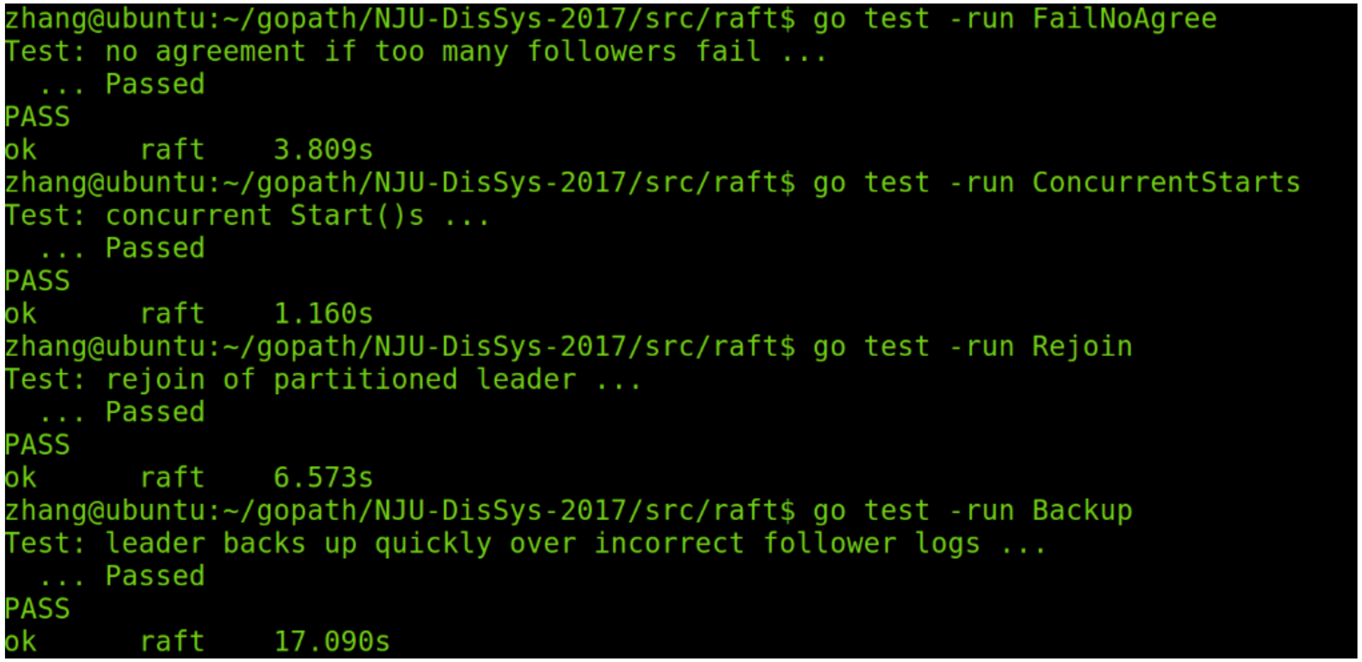
- Advice:
  - Remember field names of any structures you will be sending over RPC must start with ***capital letters***
  - Read and understand the paper before you start. Figure 2 in the paper may provide a good guideline.
  - ***Start early!***

Assignment ***Part II***

- Software as the above
- Your task in this part:
  - Implement the leader and follower code to append new log entries
    - implementing *Start(),* completing the *AppendEntries* RPC structs, sending them, and completing the *AppendEntry* RPC handler
    - pass the *TestBasicAgree*() test, try to pass all test before “Persist”

Assignment *Part II*

- The results should be like follows:



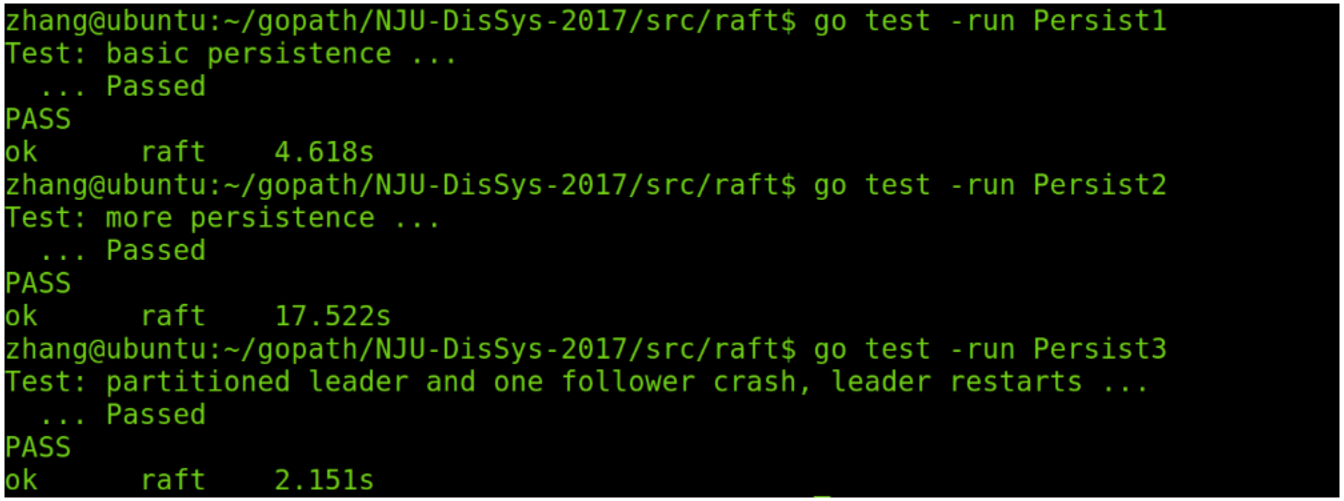
```
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run FailNoAgree
Test: no agreement if too many followers fail ...
  ... Passed
PASS
ok      raft    3.809s
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run ConcurrentStarts
Test: concurrent Start()s ...
  ... Passed
PASS
ok      raft    1.160s
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run Rejoin
Test: rejoin of partitioned leader ...
  ... Passed
PASS
ok      raft    6.573s
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run Backup
Test: leader backs up quickly over incorrect follower logs ...
  ... Passed
PASS
ok      raft    17.090s
```

Assignment ***Part III*** (Optional)

- This part is optional. There are bonus points for completing this part, and no point will be deducted if it is not completed.
  - Handle the fault tolerant aspects of the Raft protocol
    - require that Raft keep persistent state that survives a reboot (see Figure 2 for which states should be persistent)
  - won't use the disk; instead, it will save and restore persistent state from a *Persister* (see *persister.go*)
    - initialize its state from that *Persister*, and should use it to save its persistent state each time the state changes.
  - You should determine at what points in the Raft protocol your servers are required to persist their state, and insert calls to *persist*() in those places

Assignment ***Part III*** (Optional)

- Pass the Persist test:



```
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run Persist1
Test: basic persistence ...
  ... Passed
PASS
ok      raft    4.618s
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run Persist2
Test: more persistence ...
  ... Passed
PASS
ok      raft    17.522s
zhang@ubuntu:~/gopath/NJU-DisSys-2017/src/raft$ go test -run Persist3
Test: partitioned leader and one follower crash, leader restarts ...
  ... Passed
PASS
ok      raft    2.151s
```

- Try to pass the further challenging tests towards the end
  - need to implement the optimization to allow a follower to back up the leader's *nextIndex* by more than one entry at a time

作业说明

1）邮件标题格式

- DZ/MG/MF123WXYZ+姓名+DisEX

2）附件格式

- 报告为pdf，代码rar（报告与代码不必打包）
- 文件名： DZ/MG/MF123WXYZ +姓名+DisEX

3）报告内容与格式

- 详细说明如何达到目标：简述分析与设计、实现演示、总结；

4）杜绝任何形式的代码拷贝与报告抄袭，遵循学术规范。

5）作业占总成绩30分，其中第一部分20分，第二部分10分。第三部分选做，最多10分的加分计入总评。

6）格式要求

- 未按照1、2、3中格式要求的，每项扣2分；
- 未遵守4的，取消本课程成绩。

作业说明

7）提交邮箱：nju_dissys@126.com

8）截止日期：2020.12.30 23:59:59