Задание 3. Raft

Part 3A

Начало: 6 февраля 2017 года  
Конец: 12 февраля 2017 года (23:59:59)

Part 3B

Начало: 13 февраля 2017 года  
Конец: 19 февраля 2017 года (23:59:59)

Part 3C

Начало: 20 февраля 2017 года  
Конец: 26 февраля 2017 года (23:59:59)

### Introduction

This is the first in a series of labs in which you'll build a fault-tolerant key/value storage system. In this lab you'll implement Raft, a replicated state machine protocol. In the next lab lab you'll build a key/value service on top of Raft. Then you will “shard” your service over multiple replicated state machines for higher performance.

A replicated service (e.g., key/value database) achieves fault tolerance by storing copies of its data on multiple replica servers. Replication allows the service to continue operating even if some of its servers experience failures (crashes or a broken or flaky network). The challenge is that failures may cause the replicas to hold differing copies of the data.

Raft manages a service's state replicas, and in particular it helps the service sort out what the correct state is after failures. Raft implements a replicated state machine. It organizes client requests into a sequence, called the log, and ensures that all the replicas agree on the the contents of the log. Each replica executes the client requests in the log in the order they appear in the log, applying those requests to the replica's local copy of the service's state. Since all the live replicas see the same log contents, they all execute the same requests in the same order, and thus continue to have identical service state. If a server fails but later recovers, Raft takes care of bringing its log up to date. Raft will continue to operate as long as at least a majority of the servers are alive and can talk to each other. If there is no such majority, Raft will make no progress, but will pick up where it left off as soon as a majority can communicate again.

In this lab you'll implement Raft as a C# object type with associated methods, meant to be used as a module in a larger service. A set of Raft instances talk to each other with RPC to maintain replicated logs. Your Raft interface will support an indefinite sequence of numbered commands, also called log entries. The entries are numbered with index numbers. The log entry with a given index will eventually be committed. At that point, your Raft should send the log entry to the larger service for it to execute.

Only RPC may be used for interaction between different Raft instances. For example, different instances of your Raft implementation are not allowed to share C# variables. Your implementation should not use files at all.

In this lab you'll implement most of the Raft design described in the extended paper, including saving persistent state and reading it after a node fails and then restarts. You will not implement cluster membership changes (Section 6) or log compaction / snapshotting (Section 7).

You should consult the **extended Raft paper (included in task)** and the Raft lecture notes. You may find it useful to look at this [advice](https://thesquareplanet.com/blog/students-guide-to-raft/) written for students, and this [illustrated guide](http://thesecretlivesofdata.com/raft/) to Raft. For a wider perspective, have a look at Paxos, Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and [Bolosky et al.](http://static.usenix.org/event/nsdi11/tech/full_papers/Bolosky.pdf)

* Start early. Although the amount of code to implement isn't large, getting it to work correctly will be very challenging. Both the algorithm and the code is tricky and there are many corner cases to consider. When one of the tests fails, it may take a bit of puzzling to understand in what scenario your solution isn't correct, and how to fix your solution.
* Read and understand the **extended Raft paper** and the Raft lecture notes before you start. Your implementation should follow the paper's description closely, particularly Figure 2, since that's what the tests expect.

This lab is due in three parts. You must submit each part on the corresponding due date. This lab does not involve a lot of code, but concurrency makes it potentially challenging to debug; start each part early.

### Collaboration Policy

You must write all the code you hand in for this course, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution, you are not allowed to look at code from previous years, and you are not allowed to look at other Raft implementations. You may discuss the assignments with other students, but you may not look at or copy anyone else's code, or allow anyone else to look at your code.

Please do not publish your code or make it available to current or future students. github.com repositories are public by default, so please don't put your code there unless you make the repository private. You may find it convenient to use visual studio online, but be sure to create a private repository.

### The code

Your implementation must support the following interface, which the tester and (eventually) your key/value server will use.

// create a new Raft server instance:

Make(peers, me, persister, applyCh)

// start agreement on a new log entry:

Start(command interface{}) (index, term, isleader)

// ask a Raft for its current term, and whether it thinks it is leader

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

A service calls Make(peers,me,…) to create a Raft peer. The peers argument is an array of established RPC connections, one to each Raft peer (including this one). The me argument is the index of this peer in the peers array. Start(command) asks Raft to start the processing to append the command to the replicated log. Start() should return immediately, without waiting for this process to complete. The service expects your implementation to send an ApplyMsg for each new committed log entry to the applyCh argument to Make().

Your Raft peers should exchange RPCs using the WCF that we provide to you. It is modeled after .NET [rpc library](https://golang.org/pkg/net/rpc/), but internally uses WCF channels rather than sockets. Procedure is sends an RPC (sendRequestVote()) and that handles an incoming RPC (RequestVote()).

This lab may be your first exposure to writing challenging concurrent code and your first implementation may not be clean enough that you can easily reason about its correctness. Give yourself enough time to rewrite your implementation so that you can easily reason about its correctness. Subsequent labs will build on this lab, so it is important to do a good job on your implementation.

### Part 3A

Implement leader election and heartbeats (AppendEntries RPCs with no log entries). The goal for Part 2A is for a single leader to be elected, for the leader to remain the leader if there are no failures, and for a new leader to take over if the old leader fails or if packets to/from the old leader are lost.

* Add any state you need to the Raft struct.You'll also need to define a struct to hold information about each log entry. Your code should follow Figure 2 in the paper as closely as possible.
* C# marshals only the *public* fields in structure marked [DataContract] and [DataMember] passed over RPC. Public fields are the ones whose names start with capital letters. Forgetting to make fields public by naming them with capital letters is the single most frequent source of bugs in these labs.
* Fill in the RequestVoteArgs and RequestVoteReply structs. Modify Make() to create a background routine that will kick off leader election periodically by sending out RequestVote RPCs when it hasn't heard from another peer for a while. This way a peer will learn who is the leader, if there is already leader, or become itself the leader. Implement the RequestVote() RPC handler so that servers will vote for one another.
* To implement heartbeats, define an AppendEntries RPC struct (though you may not need all the arguments yet), and have the leader send them out periodically. Write an AppendEntries RPC handler method that resets the election timeout so that other servers don't step forward as leaders when one has already been elected.
* Make sure the election timeouts in different peers don't always fire at the same time, or else all peers will vote for themselves and no one will become leader.
* The tester requires that the leader send heartbeat RPCs no more than ten times per second.
* The tester requires your Raft to elect a new leader within five seconds of the failure of the old leader (if a majority of peers can still communicate). Remember, however, that leader election may require multiple rounds in case of a split vote (which can happen if packets are lost or if candidates unluckily choose the same random backoff times). You must pick election timeouts (and thus heartbeat intervals) that are short enough that it's very likely that an election will complete in less than five seconds even if it requires multiple rounds.
* The paper's Section 5.2 mentions election timeouts in the range of 150 to 300 milliseconds. Such a range only makes sense if the leader sends heartbeats considerably more often than once per 150 milliseconds. Because the tester limits you to 10 heartbeats per second, you will have to use an election timeout larger than the paper's 150 to 300 milliseconds, but not too large, because then you may fail to elect a leader within five seconds.
* You may find C# [time](https://golang.org/pkg/time/) and [rand](https://golang.org/pkg/math/rand/) packages useful.
* If your code has trouble passing the tests, read the paper's Figure 2 again; the full logic for leader election is spread over multiple parts of the Figure.
* A good way to debug your code is to insert print statements when a peer sends or receives a message, and collect the output in a file with 2A > out. Then, by studying the trace of messages in the out file, you can identify where your implementation deviates from the desired protocol. You might use any logger like NLog or Log4Net.

Be sure you pass the 2A tests before submitting Part 2A. Note that the 2A tests test the basic operation of leader election. Parts B and C will test leader election in more challenging settings and may expose bugs in your leader election code for which the 2A tests do not test.

### Part 3B

We want Raft to keep a consistent, replicated log of operations. A call to Start() at the leader starts the process of adding a new operation to the log; the leader sends the new operation to the other servers in AppendEntries RPCs.

Implement the leader and follower code to append new log entries. This will involve implementing Start(), completing the AppendEntries RPC structs, sending them, fleshing out the AppendEntry RPC handler, and advancing the commitIndex at the leader.

* You will need to make your implementation robust against various kinds of failures. Many of the tests involve servers failing and the network losing RPCs.
* You will need to implement the election restriction (section 5.4.1 in the paper).
* While the Raft leader is the only server that initiates appends of new entries to the log, all the servers need to independently give each newly committed entry to their local service replica (via their own applyCh). You should try should try to keep the goroutines that implement the Raft protocol as separate as possible from the code that sends committed log entries on the applyCh (e.g., by using a separate goroutine for delivering committed messages). If you don't separate these activities cleanly, then it is easy to create deadlocks, either in this lab or in subsequent labs in which you implement services that use your Raft package. Without a clean separation, a common deadlock scenario is as follows: an RPC handler sends on the applyCh but it blocks because no goroutine is reading from the channel (e.g., perhaps because it called Start()). Now, the RPC handler is blocked while holding the mutex on the Raft structure. The reading goroutine is also blocked on the mutex because Start() needs to acquire it. Furthermore, no other RPC handler that needs the lock on the Raft structure can run.
* Give yourself enough time to rewrite your implementation because only after writing a first implementation will you realize how to organize your code cleanly. For example, only after writing one implementation will you understand how to write an implementation that makes it easy to argue that your implementation has no deadlocks.
* Figure out the minimum number of messages Raft should use when reaching agreement in non-failure cases and make your implementation use that minimum.

Be sure you pass the 2A and 2B tests before submitting Part 2B.

### Part 3C

If a Raft-based server reboots it should resume service where it left off. This requires that Raft keep persistent state that survives a reboot. The paper's Figure 2 mentions which state should be persistent.

A “real” implementation would do this by writing Raft's persistent state to disk each time it changes, and reading the latest saved state from disk when restarting after a reboot. Your implementation won't use the disk; instead, it will save and restore persistent state from a Persister object. Whoever calls Raft.Make() supplies a Persister that initially holds Raft's most recently persisted state (if any). Raft should initialize its state from that Persister, and should use it to save its persistent state each time the state changes. Use the Persister's ReadRaftState() and SaveRaftState() methods.

Implement persistence by first adding code that saves and restores persistent state to persist() and readPersist(). You will need to encode (or "serialize") the state as an array of bytes in order to pass it to the Persister.

You now need to determine at what points in the Raft protocol your servers are required to persist their state, and insert calls to persist() in those places. You must also load persisted state in Raft.Make(). Once you've done this, you should pass the remaining tests.

In order to avoid running out of memory, Raft must periodically discard old log entries, but you **do not** have to worry about this until the next lab.

* The encoder you'll use to encode persistent state only saves public fields (fields whose names start with upper case letters). Using small caps for field names is a common source of mysterious bugs, since C# doesn't warn you that those won't be saved.
* In order to pass some of the challenging tests towards the end, such as those marked "unreliable", you will need to implement the optimization to allow a follower to back up the leader's nextIndex by more than one entry at a time. See the description in the extended Raft paper starting at the bottom of page 7 and top of page 8 (marked by a gray line). The paper is vague about the details; you will need to fill in the gaps, perhaps with the help of the Raft lectures.

Be sure you pass all the tests before submitting Part 2C.

### Handin procedure

Before submitting, please run all the tests one final time. Some bugs may not appear on every run, so run the tests multiple times.

Submit your code via the class's submission website.

Check the submission website to make sure you submitted a working lab!

You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days. Your grade is determined by the score your solution **reliably** achieves when we run the tester on our test machines.

## Оценка

|  |  |
| --- | --- |
| Реализация | **15 баллов** |

## Оформление

Вашу программу следует размести в Git (Желательно создать учетную Visual Studio Online (VSTS)..

Весь код должен быть покрыт модульными тестами. Ссылку на доступ к репозиторию можно прислать вместо архива.

Структура:

**src** – исходники, без каких либо бинарных файлов

**lib** – дополнительные библиотеки, которые используются в вашем решении (кроме стандартных). Если пользуетесь **.nuget**, то заливать библиотеки не надо

**bin** – в этой директории должен хранится zip архив с вашими исполняемыми файлами и библиотеками. После распаковки архива, программа должна запускаться без дополнительных действий. Если есть особенности, указывайте их в readme.txt

Не забудьте положить в архив файл **readme.txt**. В файле описать интерфейс программы (алгоритм работы с программой, пункты меню, управляющие клавиши)

!Внимание: обратите внимание на правильное именование и структуру работы, а также на содержание файла readme.txt. Если ваша работа не будет соответствовать требованиям, баллы могут быть снижены. Время сдачи работы оценивается по последнему комиту. Если у вас есть модификации к работе после последнего срока – делайте branch и заливайте изменения туда.

# Результаты работы

Результаты смотрите в интернете. Совместная работа или обмен кусками кода, при условии что факт командной работы не был указан в readme.txt, считается плагиатом и оценивается в -5 баллов всем участникам.