LEARNING RACTOR WITH RAFT

Micah Gates

WHO AM I? WHAT IS THIS? WHY ARE WE HERE?

- To learn about distributed consensus
- To learn about Ractor
- To experiment and have fun

WHAT IS RACTOR

Ractor is designed to provide a parallel execution feature of Ruby without thread-safety concerns.

WHAT IS RACTOR

BEFORE

- Process
  - Thread
  - Thread
- Process
  - Thread

WHAT IS RACTOR

AFTER

- Process
  - Ractor
    - Thread
    - Thread
  - Ractor
    - Thread

WHAT IS RACTOR

WHAT DOES IT GET YOU?

- One GVL/GIL (Global Interpreter/VM Lock) per Ractor
- Thread safety with much less headache.

WHAT IS RACTOR

HOW?

- Share (almost) nothing
- Pass message carefully
- It's the Actor Model

WHAT IS RACTOR

HOW?

- Share (almost) nothing
- Pass message carefully
- It's the Actor Model

WHAT IS RAFT?

Raft is a consensus algorithm for managing a replicated log.

- consensus algorithm
- managing
- replicated log

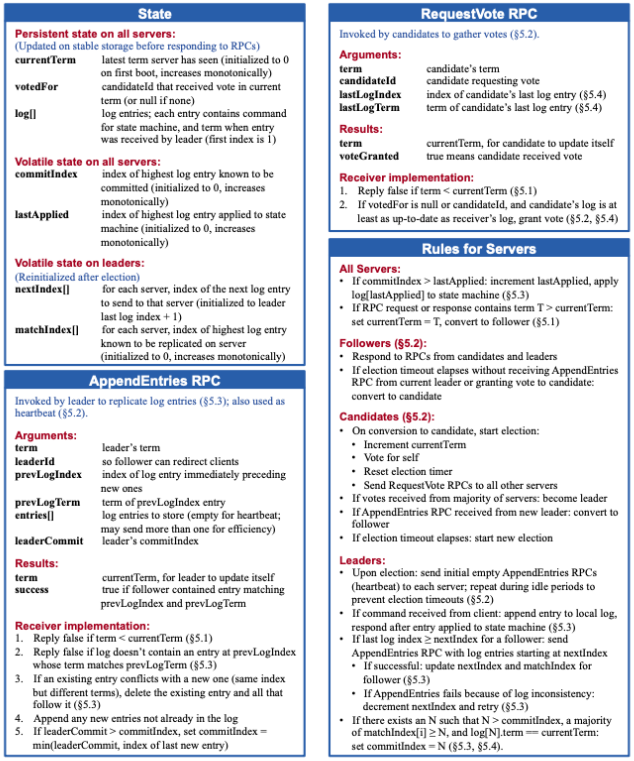
WHAT IS RAFT?

our primary goal was understandability

- Distinct states
- Clear rules
- Limited scope

WHAT IS RAFT?

https://raft.github.io/raft.pdf



State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):
- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.

WHAT IS RAFT?

- Distributed systems
- Databases
- Coordinating systems

A FEW CAVEATS

RACTOR IS EXPERIMENTAL

<internal:Ractor>:267: warning: Ractor is experimental, and the behavior may change in future versions of Ruby! Also there are many implementation issues.

A FEW CAVEATS

THIS IS NOT A REAL IMPLEMENTATION OF RAFT

WHAT ARE WE DOING

- A Set of Ractors
- Talking to each other
- Implementing (mostly) Raft

OUR FIRST RACTOR

```ruby
def new_node(id)
  Ractor.new name: id do
    while true do
      log "I'm #{name}"
      sleep rand
    end
  end
end

def log(msg)
  puts "#{Ractor.current.name}: #{msg}"
```

OUR FIRST RACTOR

```
1     def new_node(id)
2       Ractor.new name: id do
3         while true do
4           log "I'm #{name}"
5           sleep rand
6         end
7       end
8     end
9
10    def log(msg)
11      puts "#{Ractor.current.name}: #{msg}"
```

OUR FIRST RACTOR

```ruby
def new_node(id)
  Ractor.new name: id do
    while true do
      log "I'm #{name}"
      sleep rand
    end
  end
end

def log(msg)
  puts "#{Ractor.current.name}: #{msg}"
```

OUR FIRST RACTOR

```
1     def new_node(id)
2       Ractor.new name: id do
3         while true do
4           log "I'm #{name}"
5           sleep rand
6         end
7       end
8     end
9
10    def log(msg)
11      puts "#{Ractor.current.name}: #{msg}"
```

OUR FIRST RACTOR

```
🐮: I'm 🐮
🐱: I'm 🐱
🐶: I'm 🐶
🐵: I'm 🐵
🐭: I'm 🐭
🐭: I'm 🐭
🐶: I'm 🐶
🐶: I'm 🐶
🐭: I'm 🐭
🐶: I'm 🐶
🐱: I'm 🐱
```

THE CLUSTER

```
require 'timeout'

def new_node(id)
  Ractor.new name: id.to_s do
    while true do
      sender, command, body =
      begin
        Timeout.timeout(rand) do
          Ractor.receive
        end
      rescue Timeout::Error
```

THE CLUSTER

```
require 'timeout'

def new_node(id)
  Ractor.new name: id.to_s do
    while true do
      sender, command, body =
      begin
        Timeout.timeout(rand) do
          Ractor.receive
        end
      rescue Timeout::Error
```

THE CLUSTER

```
require 'timeout'

def new_node(id)
  Ractor.new name: id.to_s do
    while true do
      sender, command, body =
      begin
        Timeout.timeout(rand) do
          Ractor.receive
        end
      rescue Timeout::Error
```

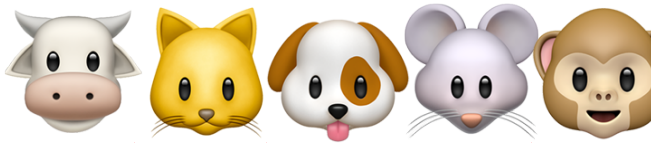
THE CLUSTER

```
require 'timeout'

def new_node(id)
  Ractor.new name: id.to_s do
    while true do
      sender, command, body =
      begin
        Timeout.timeout(rand) do
          Ractor.receive
        end
      rescue Timeout::Error
```

THE CLUSTER

```
require 'timeout'

def new_node(id)
  Ractor.new name: id.to_s do
    while true do
      sender, command, body =
      begin
        Timeout.timeout(rand) do
          Ractor.receive
        end
      rescue Timeout::Error
```

ELECTIONS



"VOTE COW FOR CLUSTER LEADER"

ELECTIONS

```
when :no_message
  log "no messages for me"
  if leader.nil? &&
          (voted_for.nil? || election_timeout < Time.now)
    election_timeout = Time.now + 2 + rand
    current_term += 1
    log "trying to get elected📦  for term #{current_term}"
    votes << self
    cluster.each do |member|
      send_message member, [self, :request_vote,
                            term: current_term]
```

ELECTIONS

```
when :no_message
  log "no messages for me"
  if leader.nil? &&
        (voted_for.nil? || election_timeout < Time.now)
    election_timeout = Time.now + 2 + rand
    current_term += 1
    log "trying to get elected  for term #{current_term}"
    votes << self
    cluster.each do |member|
      send_message member, [self, :request_vote,
                            term: current_term]
```

ELECTIONS

```
when :no_message
  log "no messages for me"
  if leader.nil? &&
          (voted_for.nil? || election_timeout < Time.now)
    election_timeout = Time.now + 2 + rand
    current_term += 1
    log "trying to get elected📦  for term #{current_term}"
    votes << self
    cluster.each do |member|
      send_message member, [self, :request_vote,
                            term: current_term]
```

ELECTIONS

```
1  when :no_message
2    log "no messages for me"
3    if leader.nil? &&
4            (voted_for.nil? || election_timeout < Time.now)
5      election_timeout = Time.now + 2 + rand
6      current_term += 1
7      log "trying to get elected  for term #{current_term}"
8      votes << self
9      cluster.each do |member|
10       send_message member, [self, :request_vote,
11                             term: current_term]
```

ELECTIONS

```
when :request_vote
  if body[:term] >= current_term && voted_for.nil?
    log "voting✋ for #{sender.name} for #{body[:term]}"
    send_message sender, [self, :vote_granted]
    election_timeout = Time.now + (3 * rand)
    voted_for = sender
    current_term = [current_term, body[:term]].max
  end
```

ELECTIONS

```
1 when :request_vote
2   if body[:term] >= current_term && voted_for.nil?
3     log "voting✋ for #{sender.name} for #{body[:term]}"
4     send_message sender, [self, :vote_granted]
5     election_timeout = Time.now + (3 * rand)
6     voted_for = sender
7     current_term = [current_term, body[:term]].max
8   end
```

ELECTIONS

```
1 when :request_vote
2   if body[:term] >= current_term && voted_for.nil?
3     log "voting✋ for #{sender.name} for #{body[:term]}"
4     send_message sender, [self, :vote_granted]
5     election_timeout = Time.now + (3 * rand)
6     voted_for = sender
7     current_term = [current_term, body[:term]].max
8   end
```

ELECTIONS

```
when :request_vote
  if body[:term] >= current_term && voted_for.nil?
    log "voting✋ for #{sender.name} for #{body[:term]}"
    send_message sender, [self, :vote_granted]
    election_timeout = Time.now + (3 * rand)
    voted_for = sender
    current_term = [current_term, body[:term]].max
  end
```

ELECTIONS

```
when :vote_granted
  votes << sender
  if votes.length > (cluster.length / 2) && leader != self
    log "I'm the leader now"
    leader = self
    votes = Set.new
    voted_for = nil
    election_timeout = nil
    cluster.each do |member|
      send_message member, [self, :append,
                                  term: current_term,
```

ELECTIONS

```
when :vote_granted
  votes << sender
  if votes.length > (cluster.length / 2) && leader != self
    log "I'm the leader now"
    leader = self
    votes = Set.new
    voted_for = nil
    election_timeout = nil
    cluster.each do |member|
      send_message member, [self, :append,
                                  term: current_term,
```

ELECTIONS

```
when :vote_granted
  votes << sender
  if votes.length > (cluster.length / 2) && leader != self
    log "I'm the leader now"
    leader = self
    votes = Set.new
    voted_for = nil
    election_timeout = nil
    cluster.each do |member|
      send_message member, [self, :append,
                                  term: current_term,
```

ELECTIONS

```
1  when :vote_granted
2    votes << sender
3    if votes.length > (cluster.length / 2) && leader != self
4      log "I'm the leader now"
5      leader = self
6      votes = Set.new
7      voted_for = nil
8      election_timeout = nil
9      cluster.each do |member|
10       send_message member, [self, :append,
11                             term: current_term,
```

ELECTIONS

```
1 if body[:term] >= current_term
2   if leader != sender
3     log "#{sender.name} is the leader now"
4     leader = sender
5     election_timeout = nil
6     voted_for = nil
7     current_term = [current_term, body[:term]].max
8   end
```

ELECTIONS

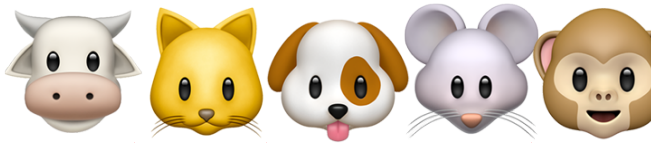
```
1 if body[:term] >= current_term
2   if leader != sender
3     log "#{sender.name} is the leader now"
4     leader = sender
5     election_timeout = nil
6     voted_for = nil
7     current_term = [current_term, body[:term]].max
8   end
```

ELECTIONS

```
1 if body[:term] >= current_term
2   if leader != sender
3     log "#{sender.name} is the leader now"
4     leader = sender
5     election_timeout = nil
6     voted_for = nil
7     current_term = [current_term, body[:term]].max
8   end
```

ELECTIONS

WRITING SOME DATA



 "STORE FOO"

 "GOT IT!"

WRITING SOME DATA

```
when :message
  message = body[:message]
  if leader && self != leader
    log "forwarding '#{message}' to the leader #{leader.name}"
    send_message leader, [sender, command, body]
  elsif self == leader
    log "adding '#{message}' to the log and replicating"
    log += message
    (cluster - [leader]).each do |member|
      send_message member, [self, :append,
                            term: current_term,
```

WRITING SOME DATA

```
when :message
  message = body[:message]
  if leader && self != leader
    log "forwarding '#{message}' to the leader #{leader.name}"
    send_message leader, [sender, command, body]
  elsif self == leader
    log "adding '#{message}' to the log and replicating"
    log += message
    (cluster - [leader]).each do |member|
      send_message member, [self, :append,
                            term: current_term,
```

WRITING SOME DATA

```
when :message
  message = body[:message]
  if leader && self != leader
    log "forwarding '#{message}' to the leader #{leader.name}"
    send_message leader, [sender, command, body]
  elsif self == leader
    log "adding '#{message}' to the log and replicating"
    log += message
    (cluster - [leader]).each do |member|
      send_message member, [self, :append,
                            term: current_term,
```

WRITING SOME DATA

```
1  when :message
2    message = body[:message]
3    if leader && self != leader
4      log "forwarding '#{message}' to the leader #{leader.name}"
5      send_message leader, [sender, command, body]
6    elsif self == leader
7      log "adding '#{message}' to the log and replicating"
8      (cluster - [leader]).each do |member|
9        send_message member, [self, :append,
10                                  term: current_term,
11                                  values: message]
```

WRITING SOME DATA

```
  when :append
    if body[:term] >= current_term
      if leader != sender
        log "#{sender.name} is the leader now"
        leader = sender
        election_timeout = nil
        voted_for = nil
        current_term = [current_term, body[:term]].max
      end
      log += body[:values]
...
```

WRITING SOME DATA

```
  when :append
    if body[:term] >= current_term
      if leader != sender
        log "#{sender.name} is the leader now"
        leader = sender
        election_timeout = nil
        voted_for = nil
        current_term = [current_term, body[:term]].max
      end
      log += body[:values]
...
```

WRITING SOME DATA

```
1  🐭: adding '[0]' to the log and replicating
2  🐵: forwarding '[1]' to the leader 🐭
3  🐭: adding '[1]' to the log and replicating
4  🐭: no messages for me
5  🐵: forwarding '[2]' to the leader 🐭
6  🐭: adding '[2]' to the log and replicating
7  🐵: no messages for me
8  🐵: forwarding '[3]' to the leader 🐭
9  🐭: adding '[3]' to the log and replicating
10 🐭: no messages for me
11 🐵: forwarding '[4]' to the leader 🐭
```

WRITING SOME DATA

```
1  🐭: adding '[0]' to the log and replicating
2  🐵: forwarding '[1]' to the leader 🐭
3  🐭: adding '[1]' to the log and replicating
4  🐭: no messages for me
5  🐵: forwarding '[2]' to the leader 🐭
6  🐭: adding '[2]' to the log and replicating
7  🐶: no messages for me
8  🐵: forwarding '[3]' to the leader 🐭
9  🐭: adding '[3]' to the log and replicating
10 🐭: no messages for me
11 🐶: forwarding '[4]' to the leader 🐭
```

WRITING SOME DATA

```
1  🐻: adding '[0]' to the log and replicating
2  🐵: forwarding '[1]' to the leader 🐱
3  🐻: adding '[1]' to the log and replicating
4  🐻: no messages for me
5  🐵: forwarding '[2]' to the leader 🐱
6  🐻: adding '[2]' to the log and replicating
7  🐶: no messages for me
8  🐵: forwarding '[3]' to the leader 🐱
9  🐻: adding '[3]' to the log and replicating
10 🐻: no messages for me
11 🐶: forwarding '[4]' to the leader 🐱
```

GETTING A NEW LEADER

```
1 if self == leader
2   cluster.each {|member| send_message member, [self, :append, term: current_term, values: []]
3 elsif leader && Time.now > heartbeat_timeout
4   leader = nil
5   log "haven't heard from the leader in a while, they're out"
6 end
```

GETTING A NEW LEADER

```
1 if self == leader
2   cluster.each {|member| send_message member, [self, :append, term: current_term, values: []]
3 elsif leader && Time.now > heartbeat_timeout
4   leader = nil
5   log "haven't heard from the leader in a while, they're out"
6 end
```

GETTING A NEW LEADER

```
if self == leader
  cluster.each {|member| send message member, [self, :append, term: current term, values: []]
elsif leader && Time.now > heartbeat_timeout
  leader = nil
  log "haven't heard from the leader in a while, they're out"
end
```

GETTING A NEW LEADER

```
if leader.nil? && (voted_for.nil? || election_timeout < Time.now)
  election_timeout = Time.now + 2 + rand
  current_term += 1
  log "trying to get elected📦  for term #{current_term}"
```

GETTING A NEW LEADER

```
1 when :append
2   if body[:term] >= current_term
3     if leader != sender
4       ...
5     end
6     heartbeat_timeout = Time.now + 3 + rand
7     log += body[:values]
```

GETTING A NEW LEADER

```
when :end_if_leader
  if self == leader
    log "bye bye"
    throw :byebye
  end
```

GETTING A NEW LEADER

```
def send_message(member, message)
  begin
    member.send message
  rescue Ractor::ClosedError
  end
end
```

GETTING A NEW LEADER

WOOHOO

We have:

- A cluster
- That can elect a leader
- Write data
- and Recover from a failed leader

WHAT WE SKIPPED

- Ensuring the log is up to date during elections
- Adding or replacing a node
- Validating replication

WHAT WE SKIPPED

- Ensuring the log is up to date during elections
- Adding or replacing a node
- Validating replication

WHAT WE SKIPPED

- Outboxes (`yield`/`select`)
- More complex `objects` (ractor.send(object, move: true/false)
- `receive_if`
- return value
- Ractor-local store `Ractor.current[:foo] = "bar"`

WHAT WE SKIPPED

- Outboxes (`yield`/`select`)
- More complex `objects` (ractor.send(object, move: true/false)
- `receive_if`
- return value
- Ractor-local store `Ractor.current[:foo] = "bar"`

THE END

Micah Gates | @mgates_com | github.com/mgates

Raft: raft.github.io

Thanks: Friends and coworkers who provided feedback, reveal.js, Diego Ongaro and John Ousterhout for Raft, and Koichi Sasada (ko1) for Ractor.

/html>