Compare Go Parallel Programming with Python

by Lukas Kiederle

Table of Contents

Motivation

The problem of having spread data over multiple networks is currently omnipresent. For example databases like Elastic Search or Docker Swarm need to deal with this efficiently in order to work reliable.

This paper is going to reflect the technical differences between go and python when building a simulated cluster, which contains its status within all of its nodes. This is accomplished with the Raft algorithm, that was introduced by Diego Ongaro and John Ousterhou in 2014.

The Raft algorithm

Raft is an understandable distributed consensus algorithm. It was created based on the consensus algorithm paxos. The key goal was to develop an alternative for paxos, which is much easier to understand and programmable.

Raft has been proven to be as efficient as paxos, but is structured differently. Diego Ongaro also claimed in his paper, that Raft is more understandable for students, which was verified by a study.

The following chapters describe the raft algorithm in detail. There will also be a comparison of an implementation in **go** and **python**.

Which problem does raft solve?

First of all there is a node. A node can store a single value. A client now sends a request to the server (this one node). Coming to an agreement which is the new state is easy. The node just updates its value. But what about having multiple nodes, that should contain the same state? How are these nodes finding a consensus?

What are the Raft Basics?

In order to have a working cluster, multiple nodes are needed, which know each other. A node can be in one of these three states: FOLLOWER, CANDIDATE or LEADER. FOLLOWER is the starting state for every node. If FOLLOWER-nodes don't hear in a specific time from a LEADER, they can become CANDIDATE.

The candidate requests votes from each other node in the cluster. Every node will answer with their vote. The candidate becomes the leader, if he gets the majority of votes. This process is called leader election. It is described in the next chapter in detail.

The cluster is now in a correct state. This includes having exactly one leader and the majority of nodes being reachable for the leader-node.

All incoming changes to the system now go through the leader-node. Its job is, to inform all of his available follower-nodes with the new state. This is accomplished with a 2-phase commit.

The 2-Phase commit

A node can represent one value. It also has an invisible log.

When a leader node sends a status update to all of his followers, the state change is added as an entry in the node's invisible log. The node replies that it stored the value. The leader waits until the majority of nodes have stored the change. If he gets enough responses, in a specific time, he sets his state to the new value. Afterwards all other nodes get the info to change to the new state, which was saved in the log previously.

The cluster now has come to consensus or agreement about the system state. This is called **log replication**.

What is a leader election in depth?

In raft there are two timeout settings which control elections. One of them is the election timeout. The election timeout is the amount of time for a follower-node waits, until becoming a candidate.

After the election timeout, a follower-node becomes a candidate and starts an election term. It then votes for itself and requests votes from all other nodes. If the receiving node hasn't voted yet, in this term, then it votes for the candidate and the node resets its election timeout. As soon as a candidate-node has more than half of the clusters votes, it becomes the leader.

The leader-node begins to send out Append Entries messages to all of its followers. These messages are sent in time periods specified by the heartbeat timeout. Every follower-node responds to every Append Entries message it receives. The leaders tenure will continue until a follower stops receiving heartbeats and becomes a candidate.

Stopping or loosing a leader in an already working cluster ends in a reelection. Only one node can be elected to a leader per term. In order to prevent having multiple candidates at the same time every node has randomized and therefore different election timer times. Otherwise an election could result in a draw. This would just trigger a reelection afterwards but cost additional time. One more important thing is that the election timer of every node always takes longer than the heartbeat timer. Otherwise a node would start an election a healthy cluster which makes no sense.

Basic parallel processing

In order to implement the raft algorithm properly, parallelism is needed as a base. This chapter explains the differences of golang and python, when it comes to parallel processing.

Goroutines:

A "thread" in golang is called goroutine. This is not the same as a normal parallel processing. A goroutine can be processed concurrently by every statement or about every line of code. In other languages like java the program can only execute whole functions parallel. This results in a finer way of parallelism and better performance. Overall a pc can execute much more goroutines than normal threads, because of this slight cutting of the code to be executed parallel.

Making a "thread" by can be easily done by writing the **go** keyword in front of a function, like displayed in the following example.

```go
package main

import (
  "fmt"
)

func main() {
  a, b := 1, 2

  go func() {
    b = a * b
  }()

  a = b * b

  fmt.Printf("a = %d, b = %d\n", a, b)
}
```

Depending on which task is being executed first, `a = b * b` or the `go func()` the result will differ. Possible results are: **a = 4, b = 8 or a = 4, b = 2**. This is called asynchronous execution.

To execute goroutines synchronous golang uses channels. In the following example almost the same code is being shown, but returning a consistent result.

```go
package main

import (
  "fmt"
)

func main() {
  a, b := 1, 2

  operationDone := make(chan bool)
  go func() {
    b = a * b

    operationDone <- true //or false
  }()

  <-operationDone

  a = b * b

  fmt.Printf("a = %d, b = %d\n", a, b)
}
```

With the channel `operationDone` and the `go func()` writing a bool into `operationDone`, the program is able to wait at the call `<-operationDone` for an element, which is saved in operationDone (Like a queue). As soon as `<-operationDone` has a value, the program goes ahead.

Python parallel programming:

In python parallel programming is done with threads. For this python offers a **Thread**-Class. A class which can be used as thread looks like the following code block.

```
from threading import Thread

class ServerThread(Thread):

    def __init__(self, counter):
        Thread.__init__(self)
        self.counter = counter

    def run(self):
        try:
            for i in range(0, self.counter):
                print(i)
        finally:
            print("Thread finished")
```

```
from src.server.serverThread import ServerThread

if __name__ == '__main__':

    for x in range(8):
        server = ServerThread(x)
        server.start()
        # wait for the thread to terminate
        server.join()
```

This thread is a class, which has a run function, that counts down one by one from a given value and prints every new decrement in the console. Once it has reached 0, the thread finishes and stops itself.

To be able to call the threads synchronous like golang does with channels, python uses the `.join()`-method.

Locking

Another tool for synchronizing is locking. The next two code blocks display a sample in python (1.) and go(2.). `Defer` and `with` are used to defer the executing of the statement until the surrounding function returns. In both examples a function an attribute is locked first and afterwards released again. This is needed a lot in the raft algorithm.

```
self.mutex = Lock()
with self.mutex:
```

```
n.mutex.Lock()
defer n.mutex.Unlock()
```

The raft algorithm implemented

In the following tables are the essential components of a raft implementation described shortly. This shows a quick overview about what to implement, before digging into the real implementation.

Name of class/structure	Description
State machine	Used for the 3 states of a node.
Replicated Log	Used for saving values in a node.
Node	Represents a node.
Timer	Used for election and heartbeat timeouts in a concurrent environment.
Cluster	Clusters multiple nodes together.

Node functions that need to be implemented:

Name of functionality	Description
request votes	Used for requesting votes from every other node in the cluster in an election.
append entries	Used for updating the nodes value.
election	Functionality which is used for the whole election process.
heartbeat	Functionality which is used for heartbeat messages.

Comparing the implementation in general

First of all the python and the go implementation have both about the same code length. While this is the case, the code complexity differs. In go there is some more trickiness, when working with goroutines, channels and node-synchronization. Other than that, the functionality, which is needed for the raft algorithm, is almost build in right from the start. Only a timer that is concurrent and can be restarted, is missing but so it is in python too. This will receive some attention later on in the chapter.

The python implementation is, as already mentioned, a little bit less complex. Understanding the code afterwards is much easier in python because there are not as many channels - just normal functions (which are of course also threads). Also, there can occur some confusion, if the reader is not that familiar with call-by-value and call-by-reference, when reading the go-code. Python does this normally for you. Even though python is easier for realizing the raft algorithm, go has a way better performance because goroutines are so much faster than python threads. Therefore, golang has the potential to start more nodes concurrently. The difference in performance and scalability was already examined by a lot of people. One good paper about this is GoLang vs Python: deep dive into the concurrency. The author compares the two languages with a sorting algorithm and presents his result in a benchmark diagram.

Another difference between go and python is, that python does not support interfaces. This leads to a dissimilar implementation of the cluster and the statemachine, but is not worth mentioning in this short paper.

Comparing the execute election function

In order to understand in depth what distinguishes python and go, a code sample is needed.

The function works in both languages about the same. First of all there is a console log message that says, that the election process starts. Afterwards, the node which starts this, is voting for itself. Next is a synchronization tool - the waitgroup. It's used for waiting on a specific amount of threads, processes or goroutines.

The function asks over the cluster-attribute for all other nodes with `nodes := n.cluster.GetRemoteFollowers(n.id)` (go) and `nodes = self.cluster.get_remote_followers(self.id)` (python). This is used for counting how many nodes to call. This number is used for the waitgroup. The next statements in between `wg.add()` and `wg.wait()` are used for calling all other nodes for votes. Finally the function checks if the election was valid. This means that the majority of nodes have chosen this node. The result is return for calling function.

Execute election in go

```go
func (n *Node) executeElection() bool {
    n.log("-> Election")
    n.votedFor = &n.id // vote for ourself

    var wg sync.WaitGroup
    nodes := n.cluster.GetRemoteFollowers(n.id)
    votes := make([]bool, len(nodes))
    wg.Add(len(nodes))
    for i, rpcIf := range nodes {
        go func(w *sync.WaitGroup, i int, rpcIf NodeRPC) {
            term, ok := rpcIf.RequestVote(n.currentTerm, n.id, 0, 0)
            if term > n.currentTerm {
                // todo
            }
            votes[i] = ok
            w.Done()
        }(&wg, i, rpcIf)
    }
    wg.Wait() // wait until all nodes have voted

    // Count votes
    nbrOfVotes := 1 // master votes for itself!
    for _, vote := range votes {
        if vote {
            nbrOfVotes++
        }
    }
    // If more than 50% respond with true - The election was won!
    electionWon := nbrOfVotes >= len(n.cluster.allNodes)/2+1
    n.log(fmt.Sprintf("<- Election: %v", electionWon))
    return electionWon
}
```

Execute election in python

```python
def execute_election(self):
        print("-> Election")
        self.votedFor = self.id  # vote for yourself

        # used for thread sync
        wg = WaitGroup()

        nodes = self.cluster.get_remote_followers(self.id)
        votes = []

        # amount of threads to wait for
        wg.add(len(nodes))

        # this function calls the request_vote from all other known nodes in the
        # cluster
        def request_votes():
            term, ok = node.request_vote(self.current_term, self.id, 0, 0)
            if term > self.current_term:
                # not not needed
                pass
            votes.append(ok)
            wg.done()

        for i, node in enumerate(nodes):
            Thread(target=request_votes).start()

        wg.wait()

        number_of_votes = 1  # Master votes for himself
        for vote in votes:
            if vote:  # if the person voted for me:
                number_of_votes += 1

        # check if over half of the cluster is alive and about to have a new
        # master
        election_won = number_of_votes > len(self.cluster.allNodes) / 2

        print('<- Election:' + str(election_won))

        return election_won
```

Even though these functions might look identical, there is still some difference other than syntactical. The `requestVote()` is called in golang with a go-prefixed function. In python `Thread(target=request_votes).start()` is needed to accomplish about the same result. Again, this is a goroutine versus a thread. The waitgroup mechanism is the same in both languages. A custom implementation is needed in python, though. The code is displayed in the following chapter.

Additional custom implementations

Custom implementation of timer:

As already mentioned previously, a custom implementation is needed in both languages in order to have a restartable timer. This is used for the two multithreadable timers (election timer and heartbeat timer) a node has.

The coding sample is python core code of the threadable timer class. The one line `if not self.finished.is_set():` blocks restarting the timer again.

```
    def run(self):
        self.finished.wait(self.interval)
        if not self.finished.is_set():
            self.function(*self.args, **self.kwargs)
        self.finished.set()
```

The restart functionality can be achieved for example with this code.

```
    def start(self):
        self.timer = Timer(self.interval, self.callback)
        self.timer.start()
```

In this case the start function is part of a class which has timer as an attribute. Every time the timer is being started, a new threadable timer is instantiated.

WaitGroup in Python

The following is a simplified custom version for a waitgroup functionality from github. Basically the waitgroup is a class which has a counter and a condition. With the add function, a number of tasks to wait for, can be added. The wait function just waits until the counter hits zero. This can only happen, when the a task that was finished calls the done function.

```
import threading


class WaitGroup(object):

    def __init__(self):
        self.count = 0
        self.cv = threading.Condition()

    # add amount of processes
    def add(self, n):
        self.cv.acquire()
        self.count += n
        self.cv.release()

    # call out that a process stopped
    def done(self):
        self.cv.acquire()
        self.count -= 1
        if self.count == 0:
            self.cv.notify_all()
        self.cv.release()

    # wait for all processes to be done
    def wait(self):
        self.cv.acquire()
        while self.count > 0:
            self.cv.wait()
        self.cv.release()
```

Conclusion

In conclusion, implementing the raft algorithm in go and in python is possible. The code needed for that, is about equal in length. On the one hand, python allows forgiving syntax while on the other hand golang is more strict with that. Other than that multithreading/concurrency is also different. Python carries out parallelism with classic threads. Go on the other hand uses goroutines, which are much more scalable and less resource consuming. Therefore, the same implementation in go can simulate much bigger clusters with the raft algorithm, than with python. The downside of go concurrent programming is, that it's a little bit harder to understand. It brings in more opportunities but also leads to more complexity.

Sources

- https://pragmacoders.com/blog/multithreading-in-go-a-tutorial
- https://www.geeksforgeeks.org/multithreading-in-python-set-2-synchronization/
- https://github.com/jweigend/concepts-of-programming-languages
- https://www.elastic.co/de/
- https://Raft.github.io/Raft.pdf
- http://thesecretlivesofdata.com/Raft/
- https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf
- https://madeddu.xyz/posts/go-py-benchmark/
- https://gist.github.com/pteichman/84b92ae7cef0ab98f5a8