Programming Assignment 3

Marcel Kyas

Version 0.9.1, 9 March 2022

This assignment accounts for 25 % of your final grade. We have estimated that typical students in a team of two should each allocate 26 h to work together to complete the assignment.
You have a related *individual* that accounts for 15 % of your final grade.

Objectives and Overview

The objective of this assignment is to get you familiar with concepts in replication, consistency, and fault tolerance.

Some Context

An important issue in distributed systems is the replication of data. Processes operate on different versions of the replicas. Thus, we need to establish a *consistent* view on the data.
van Steen and Tanenbaum (2017) discusses consistency and replication in Chapter 7 of the textbook. Consistency is relatively easy to achieve in the absence of faults.
In the presence of faults, achieving the consistency of replicas becomes a challenge. To many, the *essence* of distributed programming is achieving consensus among all processes in the presence of faults.
A popular algorithm is Paxos by Lamport (1998) (See also van Steen and Tanenbaum (2017, pp. 438–449)). Legend has it that there are at most five people on the planet that understand all of Paxos. Instead, we will focus on Raft (Ongaro and Ousterhout, 2014). Raft is, like PAXOS, a distributed consensus algorithm. Ongaro and Osterhout designed Raft specifically to be *understandable*.
Ongaro and Osterhout define the core of Raft in Section 5 and Figure 2 of their paper (also see the lecture). We will implement the protocol without configuration changes and log compaction.

We may modify this assignment to clarify any questions (and the version number incremented), but the crux of the assignment and the distribution of points (see

Section 6) will not change.

1 Components

There are two components that you are building for this assignment: server processes and client processes. There are many server processes and at least one client process connected to each server.

1.1 Server

You will implement server processes. The server processes communicate using *UDP*. The servers implement the Raft protocol as specified in Section 5 of Ongaro and Ousterhout (2014). They can be suspended and restarted using the command interface specified in Section 3.

The servers will operate in four states: Candidate, Follower, Leader, and Failed. Unless the server is in the Failed state, the server shall process calls from the client (cf. Figure 4). The server shall receive all messages in the Failed state but not respond. Neither shall it send heartbeat messages or initiate elections.

For sake of your implementations, servers in the Follower and Failed state may forward command names from clients to the Leader instead of redirecting the client to the current leader. Since servers in the Candidate state do not a leader yet, they may buffer such commands and process them after the election concludes, or they may just drop these messages.

The replicated state machine of each server will only log all the transactions that the system agrees on. It shall write the command name to a log file called `server-host-port.log`[1] Each line shall follow the format

```
term,index,command
term,index,command
```

You can introduce more state variables to your server. Raft does not assume that messages are delivered in order. Thus, you need to keep track of the response messages that your server is expecting. Especially during the leader election, new calls may announce a leader before your server received all the RequestVote response messages.

You shall not implement the persistence of the variables. We use the suspend and resume commands on the user interface to simulate crashing and recovering processes. This way, you can retain the server's state without storing and loading these from the disk.

1.2 Client

The clients generate command names to the server. For our purpose, these command names are uninterpreted strings. The servers will store the names in their

[1] Make sure to replace all colons ':' with dashes '-' to avoid errors on Windows.

log file. You shall not implement a state machine that interprets and executes those command names.

To get started, you can use a few command names, for example `insert`, `select`, and `update`. During later phases, it helps in debugging to use unique strings; you can encode client identities and sequence numbers, and track what command names are processed.

Important

Like every replicated state machine, Raft guarantees that all state machines commit the commands in the same order. Raft does not promise that the state machines will eventually commit all commands submitted by clients. It is okay if the servers do not commit all of the clients' commands as long as many are committed.

Commands may be any non-empty string that consists of lower case letters, upper case letters and digits. Commands may include dashes and underscores but may not incorporate punctuation or whitespace characters.

Clients should not wait for acknowledgements of their commands committed by the state machines.

2 Interactions between the components

All communications in this assignment shall use *UDP* as transport method. The messages use Google protobuf format again. We provide a format definition from which you can generate marshal/unmarshal code. You *must not* change the message format.[2]

The message format follows the specification of Ongaro and Ousterhout (2014, Figure 2) and is displayed in Listing 1. The network interactions are summarized in Figure 2. You need not implement the algorithms in Section 6 and Section 7! Your project will use a *fixed* cluster configuration.

Each server will listen to a port designated in its first command-line argument. Use the full server-host:server-port to identify this server to the other servers.

[2] Protobuf and its interface to Go are described at:

- `https://developers.google.com/protocol-buffers/docs/proto3`,
- `https://developers.google.com/protocol-buffers/docs/gotutorial`
- `https://developers.google.com/protocol-buffers/docs/reference/go-generated`,
- `https://pkg.go.dev/google.golang.org/protobuf/proto`, and
- `https://developers.google.com/protocol-buffers/docs/reference/go/faq`.

```
message LogEntry {
        uint64 Index=1;
        uint64 Term=2;
        string CommandName=3; // Empty string is noop
}

// AppendEntries RPC
message AppendEntriesRequest {
        uint64 Term=1;
        uint64 PrevLogIndex=2;
        uint64 PrevLogTerm=3;
        uint64 LeaderCommit=4;
        string LeaderId=5;
        repeated LogEntry Entries=6;
}

message AppendEntriesResponse {
        uint64 Term=1;
        bool   Success=4;
}

// RequestVote RPC
message RequestVoteRequest {
        uint64 Term=1;
        uint64 LastLogIndex=2;
        uint64 LastLogTerm=3;
        string CandidateName=4;
}

message RequestVoteResponse {
        uint64 Term=1;
        bool VoteGranted=2;
}

// Wrapper
message Raft {
        oneof Message {
              AppendEntriesRequest AppendEntriesRequest=1;
              AppendEntriesResponse AppendEntriesResponse=2;
              RequestVoteRequest RequestVoteRequest=3;
              RequestVoteResponse RequestVoteResponse=4;
              string CommandName=5;
        }
}
```

Listing 1: Raft message format

Make sure to send all UDP packets from this address to the other servers. The function `net.ReadFromUDP` returns this address to the receiver. Otherwise, the receiver may not be able to send the response back.

3 Interacting with processes

Both the servers and the clients shall run as foreground processes. Support for the following commands helps in debugging the programs. The commands to be supported are specific to the two components.

3.1 Server process

Start each server process with two arguments: its own identity and a file of all server identities.

The format lists all servers on an individual line. For example:

```
localhost:2000
127.0.0.1:2020
server2:2000
```

The server shall assert that its identity occurs in the file. It will listen to the port designated in its identity.

The following commands will help you in debugging your implementation.

log Print the log entires.

print Print the current term, leader voted for, state, commitIndex, and lastApplied index, nextIndex, and matchIndex.

resume Switch a server into the normal execution states.

suspend Switch a server to a suspended state. The server will receive messages from the client and other servers in this state. It shall not respond to any message while in this state. It shall not send any heartbeat messages.

3.2 Client process

Every client will read a line from its standard input stream. If the string does not match the special commands below; it shall send it as a `CommandName` of a `message Raft` to the server.

It must not send any command that contains white spaces or punctuation characters. Only letters and digits are permitted.

exit This allows a client node to exit the system.

4 Command line arguments for the two components

You should organize your code in a directory called "group-n", where n is the number of your submission group. The command-line arguments for the messaging nodes and registry are listed below:

```
go run raftserver.go server-host:server-port filename

go run raftclient.go server-host:server-port
```

We want to be able to run the system on the same host (localhost) and on different hosts.

5 Verification

We will start a network of three to ten servers. We have clients connect to random servers and issue arbitrary commands. You may not assume that the commands encode any information about the issuing clients, nor that all commands are unique.

The servers shall write the committed commands to a log file (see Section 1.1). For verification purposes, we will compare the contents of these log files.

Correct implementation will have the same sequence of commands in every log file.

6 Grading

This assignment counts for 25 % towards your final grade. The programming component accounts for 40 % with the explanation accounting 60 %. We will award 18 points in total, 15 points for the server (see Table 1) and 3 points for the client (see Table 2).

6.1 Deduction

There will be a 100 % penalty if you violate any of the following restrictions

1. If you specify your message or wire format. You must use the predefined one.
2. If you use RPC or any other middleware to implement this assignment.
3. If you use a network protocol different from *UDP*.
4. If you build any graphical user interface.
5. If you import packages that are not permitted
6. If you exchange code with your peers *without acknowledgement*

7. If you use code from websites without citation and attribution. Exception: code examples from the official Go documentation, including their tutorials, this text, and the lectures, are fine to use.

7 Milestones

You have three weeks to complete this assignment. Focus on Figure 2 by Ongaro and Ousterhout (2014), which summarises the necessary implementation steps.

Milestone 1 Start servers, listen to their designated port, and connect to the servers in the configuration file. (Estimated work time: 4 h for the whole group)

Milestone 2 Implement the client that sends Commands to one server, as read from standard input. (Estimated work time: 4 h for the whole group)

Table 1: Breakdown of Raft server nodes

Points	Feature
1	The servers listen on UDP and exchange protobuf messages.
1	The server receives commands from clients and issues corresponding AppendEntriesrequests to the other servers.
2	The leader sends heartbeats punctually, and servers elect leaders correctly
2	All nodes agree on the same order of commands
3	Trace each function of the protocol to the protocol description of Ongaro and Ousterhout (2014) and explain why you implemented the protocol correctly.
2	Explain how your implementation tracks requests and responses to ensure that a response eventually answers all requests.
2	The heartbeat mechanism relies on periodic broadcasts of heartbeat messages. Explain how you ensure that a working leader sends heartbeats in time to avoid elections. Explain how your mechanism calls for new elections if heartbeats are not received on time.
2	Explain why your implementation is ensuring progress and is free of deadlocks

Table 2: Breakdown of Raft client nodes

Points	Feature
1	The client reads messages from the terminal and sends it to a server.
2	There is a short description of why the client does not deadlock

Milestone 3 Implement the leader election algorithm in Section 5.2 by Ongaro and Ousterhout (2014). (Estimated work time: 6 h for the whole group)

Milestone 4 Implement the replication algorithm in Section 5.3 by Ongaro and Ousterhout (2014). (Estimated work time: 6 h for the whole group)

Milestone 5 Iron out any wrinkles that may preclude you from getting the correct. (Estimated work time: 6 h for the whole group)

8 Submission

Your submission should be a single tar file that contains:

- All go files related to the assignment
- A README file containing a description of each file and any information you feel the TA needs to grade your assignment.

Your reasoning can be part of the README file. It can be in a pdf file inside the tar file. It can also be in source code comments. It can be any combination of these. Please clearly specify where we find your documentation in the README file.

9 Take care of yourself

Do your best to maintain a healthy lifestyle this semester by eating well, exercising, avoiding drugs and alcohol, getting enough sleep and taking some time to relax. This will help you achieve your goals and cope with stress.

All of us benefit from support during times of struggle. You are not alone. Many helpful resources are available on campus. An essential part of the college experience is learning to ask for help. Asking for support sooner rather than later is often helpful.

A Minimal UDP client and server

The code in Listing 2 and Listing 3 demonstrates a minimal UDP server and client implementation.

References

Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 12(2):133–169, May 1998. doi:10.1145/279227.279229.

```go
package main

import (
        "log"
        "net"
)

func main() {
        addr, err := net.ResolveUDPAddr("udp", ":2000")
        if err != nil {
                log.Fatal(err)
        }

        listener, err := net.ListenUDP("udp", addr)
        if err != nil {
                log.Fatal(err)
        }

        for {
                data := make([]byte, 65536)
                length, addr, err := listener.ReadFromUDP(data)
                if err != nil {
                        log.Fatal(err)
                }
                log.Printf("From %s: %v\n", addr.String(), data[:length])
        }
}
```

Listing 2: Minimal UDP server in Go

```go
package main

import (
        "log"
        "net"
)

func main() {
        conn, err := net.ListenPacket("udp", ":0")
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()

        dst, err := net.ResolveUDPAddr("udp", "localhost:2000")
        if err != nil {
                log.Fatal(err)
        }
        _, err = conn.WriteTo([]byte("data"), dst)
        if err != nil {
                log.Fatal(err)
        }
}
```

Listing 3: Minimal UDP client in Go

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

Marten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. CreateSpace Independent Publishing Platform, 3.01 edition, February 2017. URL `https://www.distributed-systems.net/`.