Reykjavik University

T-419-CADP Project Report

Distributed Consensus

Kristófer Fannar Björnsson
kristoferb21@ru.is

Logi Sigurðarson
logis21@ru.is

Group 5

Instructor: Marcel Kyas

April 12, 2024

1 Introduction

This report explains the implementation details of group five's implementation of Raft as well as answers some questions about the implementation's structure and behavior. However, for more details on running and understanding the code, we refer to the code-base's README document and comments.

2 Server nodes

2.1 Following protocol description

Trace each function of the protocol to the protocol description of Ongaro and Ousterhout (2014) and explain why you implemented the protocol correctly.

The Raft protocol functions, and the corresponding top-level functions in the code-base can be seen below in Table 1. Each mapping displays *where* servers both receive and send the corresponding protocol in the code-base. The details of *how* these top-level functions follow the protocol is described in detail in the enumeration further below.

#	Protocol	Implementation	
		Receive	Send
1	RequestVoteRequest	HandleVoteRequest	WaitForTimeout
2	RequestVoteResponse	HandleVoteResponse	HandleVoteRequest
3	AppendEntriesRequest	HandleAppendEntriesRequest	AnnounceLeadership
4	AppendEntriesResponse	HandleAppendEntriesResponse	HandleAppendEntriesRequest

Table 1: Protocol functions and corresponding code methods

1. **RequestVoteRequest**

   Sending Each server sends a RequestVoteRequest message in the WaitForTimeout[1] function. This function is invoked as a new goroutine in the main process's Start[2] method, and runs throughout the server's entire lifecycle.

   WaitForTimeout continuously listens for messages on a timeout channel, which is populated when the server timer times out. Whenever

[1]WaitForTimeout - server/core/server.go:158
[2]Start - server/core/server.go:66

a timeout message is received from the channel, the function updates its term, becomes a candidate, and votes for itself as the term's leader. Subsequently, it creates and sends a RequestVoteRequest message to all other servers.

Receiving Each server receieves and handles RequestVoteRequest messages in the HandleVoteRequest[3] function. The function first checks whether the requesting candidate is valid, following the rules set in Raft, and depending on their eligibility either responds with a granted vote or a denial. The function ensures to update its term and become a follower if the requesting candidate's is higher, no matter whether they are valid or not.

2. **RequestVoteResponse**

   Sending RequestVoteResponses are sent by the same function that receives RequestVoteRequests, HandleVoteRequest, which is described immediately above this paragraph.

   Receiving RequestVoteResponses are received and handled in HandleVoteResponse[4]. The receiving server must be a candidate, and handles the response differently based on whether the voter denied or granted their vote request. If the vote was accepted, the candidate counts it and checks whether they have reached a majority. If they have reached majority, they become a leader, invoking AnnounceLeadership[5], which is explained in the paragraph below. If the vote was not granted, the candidate can not be entirely sure why in most cases. However, the exception to this is if the sender's term is higher than the candidate's, in which case the candidate updates the term to match the sender and becomes a follower.

3. **AppendEntriesRequest**

   Sending A server sends AppendEntriesRequest messages if they become a leader, which is executed by the AnnounceLeadership function, which was references in the paragraph above. The function changes the

[3]HandleVoteRequest - server/core/handler.go:35
[4]HandleVoteResponse - server/core/handler.go:70
[5]AnnounceLeadership - server/core/server.go:189

server's state to a leader, initializes all servers' NextIndex & MatchIndex values to be the Log array length & -1, respectively, and then invokes a new function, SendHeartbeats[6], in a specific goroutine which runs while the server maintains leadership.

SendHeartbeats periodically creates and sends AppendEntriesRequest messages to each of the other servers in the network, sleeping a randomized, although controlled, amount inbetween calls, imitating the heartbeat protocol.

Receiving Receiving and handling the AppendEntriesRequest is done by HandleAppendEntriesRequest[7]. This function begins by resetting the timeout and ensuring that it continues to follow this server as a leader by setting its global leaderId variable to the sender's Id. The function also checks whether the sender is a valid leader, checking for term values and the log length of the leader. Otherwise, it views the sender as the true leader and begins synchronizing its logs and matchIndex with the leader, following Raft's algorithm. Additionally, it checks whether the leader has signed any new logs for approval and writes all committed logs into its log file. Finally, the function responds with an appropriate AppendEntriesResponse to the leader.

4. **AppendEntriesResponse**

Sending Sending the AppendEntriesResponse is handled by HandleAppendEntriesRequest, the function which is explained in the paragraph immediately above this one.

Receiving Recieving AppendEntriesResponse messages is handled by the appropriately named HandleAppendEntriesResponse[8], which is only ran by a server in a leader state. The function simply checks whether the sender had success handling its last AppendEntiresRequest message, marking if the sender failed to handle its last message. If the sender was successful, the leader increments their count of the sender's nextIndex if it is not up-to-date, and updates its count of the sender's matchIndex aswell. Following that check, the leader checks whether it can commit a

[6]SendHeartbeats - server/core/server.go:207
[7]HandleAppendEntriesRequest - server/core/handler.go:97
[8]HandleAppendEntriesResponse - server/core/handler.go:153

new logs, and increments its commitIndex if the check succeeds. If the sender failed handling the leader's prior AppendEntriesRequest message, the leader decrements its count of the sender's nextIndex, which means that when it sends the next AppendEntriesRequest message, it will be based on a lower nextIndex, allowing the sender to catch up to the log.

2.2 Tracking responses

Explain how your implementation tracks requests and responses to ensure that a response eventually answers all requests.

Each server has a dedicated MessageProcessing goroutine that wait for messages from other servers and depending on the type of the request or response passes it to a dedicated handler.

The handler then performs the logic of the request/response and if applicable sends a response in relation to the original request. To go in more detail the main thread continuously checks if it has received any message, if it has it passes the message with the sender address into a channel of messages, which the message processing goroutine is listening for.

2.3 Ensuring heartbeat deliveries

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.

Lets start by explaining the mechanisms the group uses to ensure timeouts are reached when a follower does not get an AppendEntriesRPC. Each server has two synchrounous channels called TimeoutReset and TimeoutDone. Both use empty structs to communicate message, because they take up zero bytes.

In the start function two goroutines are ran which are called StartTimeout and WaitForTimeout. The StartTimeout function infinitely loops and waits for a message to get passed to the TimeoutReset channel. When this happens the global timer of the server is reset, preventing him from timing out and transition into the candidate state. When a AppendEntriesRPC message is received an empty struct is passed into the TimeoutReset channel, thus restarting the timer. If no message is received on the channel before the timer runs out, then the StartTimeout passes a empty struct to the TimeoutDone channel. Then the WaitForTimeout function kicks in and transitions the follower into a candidate state, increments the term and sends voting request to all servers. Then he waits for VoteResponses and if the majority are positive he transitions to a leader state.

Looking at this from the perspective of the leader, then he has to send AppendEntriesRPC faster than the timeouts of the other servers to hold control. This is what the group implements, the leader sends AppendEntriesRPC calls ten times faster than the timeouts of the other servers. The group set the servers timeout to be in the range 150-450ms, so to combat this as a leader, it sends AppendEntriesRPC request every 15-45ms. With this in place the leader ensures he stays in control as long as he does not fail. This is implemented by calling another goroutine SendHeartbeats which continuously loops as long as the state is leader, and then sleeps for the aforementioned time 15-45ms.

2.4 Deadlock freedom

Explain why your implementation ensures progress and is deadlocks-free.

The group's solution does not contain any locks, thus negating one of the four necessary condidtions for a deadlock to exist: the Hold and Wait condition. The server nodes are carefully implemented in a way so that while multiple goroutines execute concurrently for each server, only one goroutine, the *server.MessageProcessing* method call in *server.Start*, manipulates global variables. This ensures that sections of the code do not have to be locked, and no mutual exclusion is necessary.

3 Client node

3.1 Deadlock freedom

The client functionality is quite trivial, and only requires a single goroutine iterating through a loop. Out of the four necessary conditions required for a deadlock to take place, at least three are not met in the client code:

1. **Mutual Exclusion**
   As the client code only requires a single goroutine, the main process, there is no exclusivity or sharing of resources to be had.

2. **Hold and Wait**
   Again, as the client code only utilizes a single goroutine, the main thread, there is no use in mutex locks, which is precisely why none are present in the solution. Therefore, the goroutine has nothing to hold, and nothing to wait for.

3. **Circular Wait**
   Again, the single goroutine present in the client code has nothing to wait for, and definitely can not create a circle of waiting processes, as it is the only operating process.

In reality, we are slightly unsure of why the question of deadlock freedom in the client node is raised, as we simply do not see why (or even how) a student implementation might introduce deadlocks.

4 Conclusion

In this report the group went over its implementation of the Raft consensus algorithm as described by Ongaro and Ousterhout (2014). Deadlock avoidance and protocol adherence were discusses as well as the design and structure of the code.

5 Thoughts

For our Raft implementation to be usable & debug-able, a *debugScale* variable was added to each random timeout generation, which was used to generate random durations both for timeouts and heartbeat sleeps. The most user-friendly value turned out to be 200, which meant that the initial random heartbeat duration spanned from 3 - 9 seconds rather than 150 - 450 milliseconds. As the timeout values are set to be a tenth of the heartbeat generated value, to match the order of magnitude ratio suggested in the Raft paper, that meant that the timeout span ranged from 30 - 90 seconds.

To combat these slow timeouts in a fast testing environment, a new *timeout* CLI command was added to the servers, which would manually time them out. For the submission, *debugScale* is however set to 1.

Thank you for reading this report and going over our assignments this spring. Raft was definitely challenging but also one of the most rewarding assignments this term. We hope you enjoyed our solution and have a great spring!