

Industrial Training
on
Implementation and Improvement of Raft Consensus Algorithms and
its application in Federated Learning

SUBMITTED

BY

SAGNIK CHATTERJEE

180905478

Roll No. 61

Section: B

sagnik.chatterjee1@learner.manipal.edu

+91 9380063827

Under the Guidance of:

R Vijaya Arjunan

Associate Professor

Dept. of Computer Science and Engineering

MIT-Manipal



MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

August 2021

Industrial Training Certificate



MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL

(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Date: July 31, 2021

To whomsoever it may concern

This is to certify that Mr. Sagnik Chatterjee, a student of Manipal Institute of Technology, bearing registration No. 180905478 has successfully undergone Industrial Training from Apr 2021 to Aug 2021.

The title of the project is Implementation and Improvement of Raft Consensus Algorithms and its application in Federated Learning. We found his performance satisfactory. We wish him good success in future endeavours.

Name of the Project Guide: R Vijaya Arjunan

Designation: Associate Professor

Department: Department of Computer Science and Engineering

Institute : MIT, Manipal

ACKNOWLEDGEMENT

I would like to thank **Manipal Institute of Technology** and **Dept of CSE,MIT-Manipal** for providing me the opportunity to undertake the Research Internship position.

Asserting the importance of practical experience for an engineer motivated me to use this internship for learning about the various intricate details on Distributed Systems and how they work in the actual lower level. This not only helped me in understanding the technology from scratch but also helped me in developing the app from scratch. This also put me in a position to do and read more research papers and new technologies that are emerging and have helped us in developing my interest in both the software development and also for understanding a major problem of large datasets coming up in Machine Learning and Deep Learning.

I want to express my deepest gratitude to **R Vijaya Arjunan** sir, for giving me an opportunity to conduct research under his mentorship and guiding me along the course of this internship imparting theoretical and practical knowledge with his vast experience and being an expert in this domain. Not only did he help me understand core principles of the core development stack he also helped me inculcate a sense of developing the software and understanding things in depth and applying the knowledge in the project.

I also want to thank **Dr Ashalatha Nayak**, and the entire CSE Dept for their continuous support and help and helping me inculcate a sense of research and also helping me by giving me the opportunity to do this research internship so that I can develop my skills as a researcher and software developer to understand and implement large software systems.

ABSTRACT

This project's aim is to explain and using Raft distributed consensus algorithm for using in federated learning and how to manage those states and large distributed processing and compute using consensus algorithms. Being a vast domain, I narrowed down my domain to working and implementing a Raft consensus algorithms and implemented parts of Federated learning mechanism on large systems.

Due to the large number of IoT and Edge devices present globally and the vast majority of it using Google services , Google can use those services to train large deep learning models on large data sets , on a scale that is practically unachievable using traditional datasets which traditional methods in machine learning and deep learning use of either using the whole data from a single source or running the computer intensive tasks on cloud.

Federated Learning enables mobile phones to collaboratively learn a shared prediction model while keeping all the training data on device, decoupling the ability to do machine learning from the need to store the data in the cloud.[1] This may not necessarily be used for only query suggestions but can also be extended on for ex: distributed neural networks .

The amount of time to develop a better predictive model on less resource intensive devices grows exponentially better for large systems as we go on using Federated Learning and this fundamentally breaks down into a large infrastructure and distributed systems problem.

In this process Raft will play an important role on the consensus algorithm for connecting for large distributed systems and will lay out the protocol for how to connect for different devices and how to connect with them. Not only the connections of connecting to the master node , for the edge devices but also we can use them to fundamentally lay out how the data transfer should lay out , and how to handle the input sources from a large number of systems in real time and how to check for possible duplicates for training some models.

Another fundamental thing that is important in Federated Learning is privacy and how the information will be used. Although it is not not discussed here extensively and I did not research into the methodology on how to share data partially from large number of devices and doing calculation on encrypted data , recent advanced in Homomorphic Encryption shows promise, eg:SEAL[2] library from Microsoft Open source on using the data for doing large scale machine learning.

TABLE OF CONTENTS

Content	Page Number
1. Details of the Organization	6
2. Motivation	7
3. Definition of Consensus	8
4. Raft ,Leader Election,Log Replication	9
5. Membership Changes,rafty	10
6. Federated Learning	11
7. Implementation Specific Details	12-13
8. Protocols and Implementation	14-18
9. Serialization and Architecture	19-25
10. Log Management	26
11. Client	27
12. Federated Learning +Raft	28- 31
13. Conclusion and Results	32- 34
14. References	35

DETAILS OF THE ORGANIZATION

Manipal Institute of Technology was ranked 21 among engineering colleges in India by *India Today* in 2020, 20 among engineering colleges by *Outlook India* in 2019 and 45 among engineering colleges by the National Institutional Ranking Framework (NIRF) in 2020 and 43 in 2019.

Enhancing research skills among the students and scholars is one of the primary objectives of the institute. The institute makes adequate budgetary provisions for maintenance of research equipment through capital allocations.

Research has always been an area of paramount importance to Manipal Academy of Higher Education. The importance increased tremendously after obtaining the deemed university status. There has been a dramatic rise in the publications, doctoral degree awards and grants awarded by funding agencies since then.[0]

MIT-Manipal's Objectives for Research are:[0]

- **To identify research priorities and initiate programmes.**
- **To seek research grants and promote focused research of national importance.**
- **To foster new collaborations and strengthen existing ones.**
- **To disseminate research findings in high impact publications.**
- **To provide logistics for researchers to implement ideas.**
- **To support IP related activities and Technology Transfer.**

MOTIVATION

My primary motivation for doing this project came from a blog article I read at the December 2020 on the impact Google is having in the development on Federated Learning and that data need not be present in a physical single location to work and that it can work seamlessly on cryptographic data without any problem using Homomorphic Encryption.

From lobster.rs I also came to know about PySyft and how it is working to bring Federated Learning for small powered devices using PyTorch and the SEAL homomorphic encryption library.

During the same time I was also taking courses on Distributed Systems and came to know about how existing solution in distributed systems like Raft consensus algorithms help in making consensus of large distributed systems and how that can leverage flow of data from multi devices using set of predefined rules and make the whole experience a seamless integration although the data sources are coming from different parts.

It was majorly these two things that motivated me to learn more about the distributed systems and how can I build Raft consensus algorithms with Federated Learning for multi-device and multi platform integration for learning algorithms.

Also I was motivated by the works of MoshPit SGD and Google AI blog on Federated Learning and how it used this for search query and how well it predicts search query and how it provides a better performance for large systems that are virtually different and how it can train data on so many devices and how it can scale for large number of users.

Definition of Consensus

Consensus algorithms address a common problem in distributed systems: getting a cluster of machines to share a mutable data structure that remains coherent in spite of failing nodes and unreliable networks connecting them. In the case of Raft consensus algorithm, the objective is to replicate a state machine within a cluster of servers and coordinate the interaction of said cluster with clients.

A state machine is a collection of the state variables and commands which can either transform the state variables or produce some output. Also, since the commands provided by the state machine is deterministic for the network and their execution being processes at atomic level, this will allow the state to represent the state variables as an ordered collection of commands.

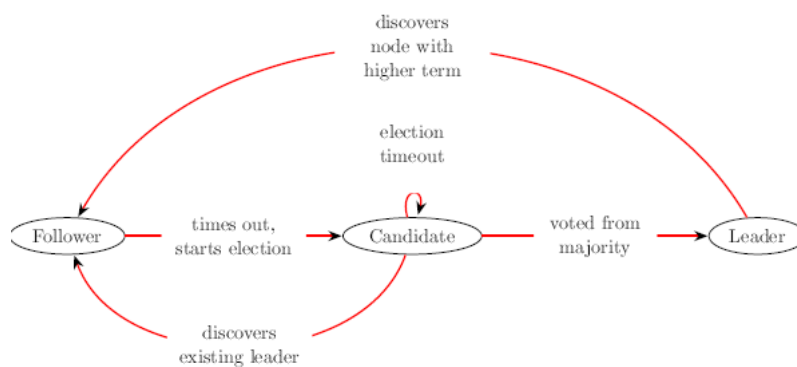
Leslie Lamport in 1989 solved this problem of state machine replication using Paxos algorithm [3], that elegantly solved the state machine replication problem, but the main shortcomings of Paxos is the fact that working implementations need significant changes from the specification(multi-Paxos).

Raft algorithm was developed by Diego Ongaro and John Ousterhout in Stanford in 2013 to address the above issues, resulting in an easier algorithm with a better understandability, proven by a user study [4].

Raft

Raft algorithm works by distributing to each node of the cluster an identical log, where each entry contains an update command for the state machine. Once a log entry is replicated in a majority of the cluster, it is applied to the state machine. This result is achieved by integrating 2 separate mechanisms:

1. Leader Election.
2. Log Replication



Leader Election

Leader election[5] subsystem is in charge of managing the role of nodes in the cluster, and works as follows: each server in the cluster can be either a Leader, a Candidate or a Follower. The cluster starts with all the nodes as Followers; after a short-randomized time-out (in range of 100 ms) where no heartbeat was received by a Leader, each Follower will convert itself to Candidate. In doing so, it will vote for itself, and will send a vote request to the rest of the cluster. All the nodes that had not yet voted, assign their vote to the first request received. This election process is repeated until a Leader emerges (no tie condition).

Log Replication[6]

Once a leader is elected, that node has the responsibility for the log replication subsystem. The Leader in fact is in charge of getting new append requests from clients, distributing them to the rest of the cluster and waiting for a majority to acknowledge them and finally sending a successful reply to the client.

Membership Changes

Fundamental feature of the Raft algorithm is the membership changes subsystem. This handles seamless transitioning of nodes in and out of the cluster due to a technique called joint consensus that makes the majority of 2 different configurations overlap during the transition period.

rafty

Rafty presents itself to the user as a simple dictionary. However, a Raft dictionary can be automatically replicated across multiple machines and is highly available.

In addition, an automated benchmark environment was developed with two objectives in mind: first, the development process benefited greatly from quick local benchmarks, leading to the discovery and elimination of many performance bottlenecks. Second, large-scale distributed benchmarks are useful to the end users, allowing them to evaluate the library with respect to speed in its final deployment context.

Federated Learning

Federated Learning is a machine learning setting where the goal is to train a high-quality centralized model with training data distributed over a large number of clients each with unreliable and relatively slow network connections. Here learning algorithms are considered in this setting where on each round, each client independently computes an update to the current model based on its local data, and communicates this update to the central server, where the client-side updates are aggregated to compute a new global model.

Here the datasets are assumed to be typically heterogeneous and their size may span several orders of magnitude. Also, the clients involved in federated learning can be unreliable as they are subject to more failures or drop out. This makes it an identical case as that for a consensus algorithm for a cluster of nodes.

Most current federated learning techniques that are currently being developed are done so on synchronized model updates. This may hold true for laboratory conditions but in the real world, from CAP theorem, availability, consistency and persistence all of them cannot be guaranteed at the same time, so although if a system is being available does not mean the state machine is also being consistent.

Instead of this, an asynchronized model leverages the properties of neural networks to exchange model updates as soon as the computations of a certain layer are available.

Deep Learning requires large amounts of data, that in several instances are proprietary and confidential to many businesses.

In order to respect individual organization's privacy in collaborative machine learning, federated learning could play a crucial role. Such implementations of privacy preserving federated learning find applicability in various ecosystems like finance, health care, legal, research and other fields that require preservation of privacy.

Most of such implementations are driven by a centralized architecture in the network, where the aggregator node becomes the single point of failure, and is also expected with lots of computing resources at its disposal.

The idea is there is no one permanent aggregator, but instead a transient, time-based elected leader, which will aggregate the models from all the peers in the network. The leader (aggregator) publishes the aggregated model on the network, for everyone to consume.

Implementation Specific Details

rafty

rafty is the Raft implementation and is written in Python 3.8.9 .

The server module uses asyncio library and the async/await primitive and thus the package is supported only on Python 3.6 and above.

Concurrency Model, Asyncio and Python GIL

Due to the global interpreter lock [7] (GIL for short), present in CPython implementation, an instance of the CPython interpreter will only be able to execute a single instruction at any given point in time, therefore classic concurrency strategy like multithreading and not very effective in CPython.

For ease of development and general better performance, asyncio library in Python was chosen as it provides support for single threaded asynchronous programming using pluggable event loop, coroutines and socket I/O access multiplexing.[8]

Networking

The intra-cluster networks use UDP as a transport protocol, while TCP is being used for client server messages. An earlier version only used TCP, UDP was later chosen for intra-cluster exchanges since many of the functionalities provided by TCP overlap with Raft's own mechanism.

Serialization

Messages for both the intra-cluster and client-server communication are serialized using the MessagePack data interchange format.

Original intention was to use JSON, but MessagePack was advantageous in the following respect[9]:

1. Binary representation that allows for more compact messages.
2. Faster de/serializer implemented in C++.
3. Support for greater range of datatypes and in particular allows for non-string keys in the map datatype.
4. MessagePack allows for non-string keys in the map datatype.

Python 3

I used Python for the implementation as I was both familiar with programming in Python and it helped in quick iteration of the programs. Although there were some hiccups in the way for eg: the limiting factor of using GIL and multiprocessing in Python are an issue and this is limiting for large distributed consensus algorithms.

To counter this I have used asynchronous primitives in Python.

Detailed Implementation

rafty features a client-server architecture, where the server can be a cluster ranging from one to tens of machines. Its purpose is to replicate and persist to disk the log entries that generate the state machine.

The client is a library that has to be integrated into the user's software and is used to interact with the cluster.

The rafty client interface is a Python dictionary, therefore it exposes to the user the typical API of a key-value datastore, comprising just a set and a delete operation. The communication protocol between client and server however features some additional RPCs, such as a diagnostic call, to retrieve the status of a particular server and a configuration call, to perform cluster membership changes.

Protocol

Two types of communication channels are needed in rafty: the client-server channel, and intra-cluster channels that connect servers with each other:

1. Client - Server Protocol
2. Intra - Cluster Protocol

Client-Server Protocol

- This protocol is layered on top of TCP, benefiting greatly from its reliability.
 - The client is the only entity that initiates requests and its ports are therefore ephemeral; the server port, on the other hand, is stable and has to be configured at start-up.
 - All of the message exchanges are part of a request-response cycle consisting of just one request, followed by its response, after which the connection is terminated.
-
- Requests and responses consist of nested map msgpack objects with varying _elds, in particular, the type _eld determines the purpose of the message.
 - The requests types are as follows.
 - **GET**: This request is used to get the complete state of the state machine, that is, the result of the application of all the log entries to the state machine.
 - **RESPONSE**: This request cannot fail, the state machine is returned as a map without additional metadata.
 - **APPEND**: Request is used to enter new data into the key-value datastore or to likewise update the value of an existing key.

- This request has to necessarily target the leader, therefore if a Follower receives such a request , it sends back a redirect request along with the address of the current leader. All the subsequent requests will directly target the leader.

Key	Value
"type"	"get"

Append Response

Key	Value
"type"	"append"
"data"	

Append request

- **Diagnostic** Retrieve information from any of the servers about their status.
 - The stats entries represents, for each category (read, write, append) a queue of length 10 indicating the quantity of operations of that type executed on the log in the last 10 seconds, along with their UNIX timestamp.

Diagnostic Request

Key	Value
"type"	"diagnostic"

Diagnostic Response

Key	Key								
"status"	"Leader", "Follower", "Candidate"								
"persist"	<table> <tr> <th>Key</th><th>Value</th></tr> <tr> <td>"votedFor"</td><td>address:port</td></tr> <tr> <td>"currentTerm"</td><td>integer</td></tr> </table>	Key	Value	"votedFor"	address:port	"currentTerm"	integer		
Key	Value								
"votedFor"	address:port								
"currentTerm"	integer								
"volatile"	<table> <tr> <th>Key</th><th>Value</th></tr> <tr> <td>"leaderId"</td><td>address:port</td></tr> <tr> <td>"cluster"</td><td>[address1:port1, address2:port2,]</td></tr> <tr> <td>"address"</td><td>address:port</td></tr> </table>	Key	Value	"leaderId"	address:port	"cluster"	[address1:port1, address2:port2,]	"address"	address:port
Key	Value								
"leaderId"	address:port								
"cluster"	[address1:port1, address2:port2,]								
"address"	address:port								
"log"	<table> <tr> <th>Key</th><th>Value</th></tr> <tr> <td>"commitIndex"</td><td>integer</td></tr> </table>	Key	Value	"commitIndex"	integer				
Key	Value								
"commitIndex"	integer								
"stats"	<table> <tr> <th>Key</th><th>Value</th></tr> <tr> <td>"reads"</td><td>[integer,]</td></tr> <tr> <td>"writes"</td><td>[integer,]</td></tr> <tr> <td>"append"</td><td>[integer,]</td></tr> </table>	Key	Value	"reads"	[integer,]	"writes"	[integer,]	"append"	[integer,]
Key	Value								
"reads"	[integer,]								
"writes"	[integer,]								
"append"	[integer,]								

- **DELETE:** To delete an entry from the state machine.

Configuration: This request is used to perform cluster membership changes, that is , adding or removing members to the cluster. In order to add members, the action field has to be set to add the address, port fields to the new node's IP and port.

Key	Value						
"type"	"append"						
"data"	<table> <tr> <th>Key</th><th>Value</th></tr> <tr> <td>"action"</td><td>"delete"</td></tr> <tr> <td>"key"</td><td>key</td></tr> </table>	Key	Value	"action"	"delete"	"key"	key
Key	Value						
"action"	"delete"						
"key"	key						

Request for delete

Response for delete

Key	Value
"type"	"result"
"success"	boolean

- **Configuration:** This request is used to perform cluster membership changes, that is, adding or removing members to the cluster. In order to add members, the action_eld has to be set to add the address, port fields to the new node's IP and port.

Intra-Cluster Protocol

- Intra-cluster networking was also initially built on top of TCP.
- However, this transport protocol was quickly deemed suboptimal for the task since it clashes in some respects with the functionalities provided by Raft such as : segment retransmission and sequencing. As a result, the UDO protocol was finally chosen for intra cluster networking.

Key	Value
"type"	"config"
"address"	address
"port"	port
"action"	add, delete

Reconfiguration Request

Key	Value
"type"	"result"
"success"	boolean

Reconfiguration Response

- When a heartbeat is lost in Raft, the system quickly recovers when the next heartbeat is successfully delivered.
- When this mechanism is layered on TCP, the transport protocol's own retransmission policy results in a late delivery of two out-of-time heartbeats that only generate an unpredictable behavior of the system.
- Further, Raft also contains an acknowledgment system, represented by the heartbeat responses, thus a subsystem performing the same task is of no help, since such duties are better performed as high as possible on the networking stack .

- As with the client-server protocol, the requests and responses consist of nested map msgpack objects with varying fields.

Vote Request:

- Leader sends this request periodically every 0.5-20 ms to each Follower containing the log entries following the highest log entry known to be replicated on that server.
- It support batching, that is, sending up to 100 items with each message within the limits imposed on the size of a single message by the transport protocol.
- Append entries request also works as an heartbeat, that is, when the Leader has no new entries to send, it sends an empty request in order to maintain its leadership position over the cluster.
- When a Follower signals the Leader that its last commit index is lower than the last compacted element's index, the Leader adds to the next Append entries request the complete compacted log, along with its last index and term of the last item.

Key	Value
"type"	"request_vote"
"term"	integer
"candidateId"	address:port
"lastLogIndex"	integer
"lastLogTerm"	integer

Vote Request

Key	Value
"type"	"response_vote"
"voteGranted"	boolean
"term"	integer

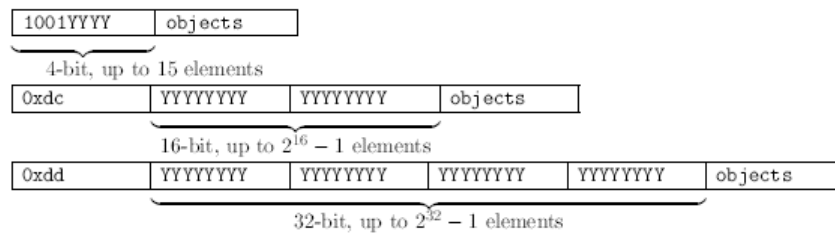
Vote Response

SERIALIZATION

- Initially, the json format was used for serialization of objects in both network messages and disk persistence for several reasons:
 - a text format is easier to understand when intercepting the messages at the IP level for debugging purposes, a process often used in the early development stages.
 - JSON library is integrated into the Python standard library, making it easier for development.
- In the testing stage, de/serialization was occupying much of the CPU time on the Leader node. The standard library's json implementation was substituted with the more efficient json library written in C, but the gains were marginal, going from 23% to 18% of CPU time spent in de/serialization.
- After a comparison and testing phase of other serialization format at the end msgpack was used.

	JSON	MessagePack	CapNProto	ProtoBuffer	FlatBuffer
Encoding	text	binary	binary	binary	binary
Append to array capability	N	Y	Y	N	N
Schemaless	Y	Y	N	N	N
Stream de/serialization	N	Y	Y	N	Y
Usable as mutable state	Y	Y	N	N	N
Presence in PyPi	Y	Y	Y		
Zero copy	N	N	Y	N	Y

- Benefits of using msgpack:**
 - Good performance (3 to 4%) of the CPU time
 - A binary encoding that allows for space-efficient serialization
 - Similarity with JSON in being a schema less protocol
 - capability of appending to an array in-place is useful for cheaply persisting the log to disk.
- MessagePack serializes arrays as a variable amount of bytes (from 1 to 4) containing the array length, followed by the actual array elements.

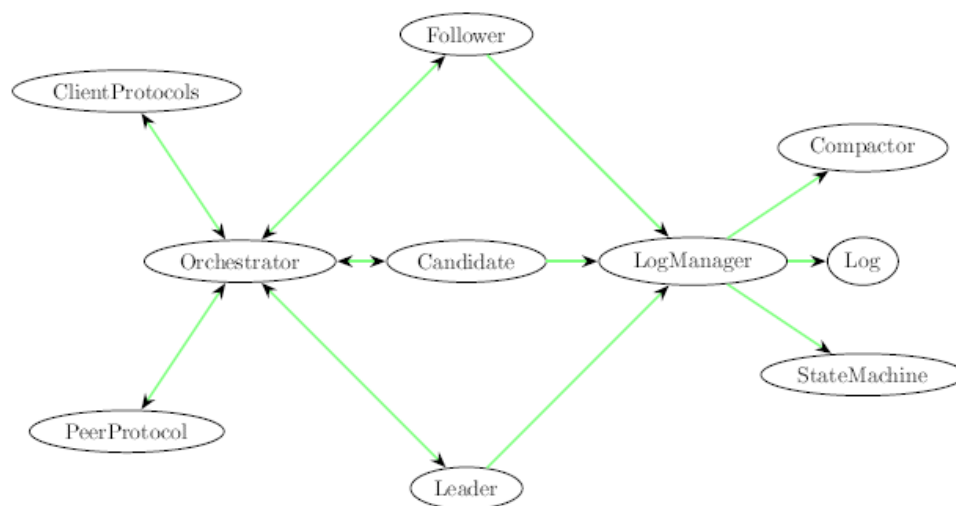


Flexible Serialization using Msgpack

SERVER

- Rafty server is a single threaded application with a non-pre-emptive concurrency model based on coroutines , multiplexed nonblocking I/O and a event loop.

ARCHITECTURE



- **Orchestrator** The Orchestrator is the first class to be initialized after the event loop, its task is to switch between Raft States whenever a state yields control to another, and to handle the message brokerage between the communication subsystems and the current state.
- **Protocols** The **TCP and UDP** transport channels are handled by the asyncio library with a construct called Protocol.
 - A ClientProtocol is created for each client connection. Messages received by a ClientProtocol are forwarded to the Orchestrator that will in turn forward them to the current State. Likewise, when a message is to be delivered to a client, the orchestrator passes it to the appropriate ClientProtocol for dispatch.
 - Since UDP is a connectionless protocol, a single PeerProtocol is sufficient to handle the full duplex communication between a node and all of its peers.

- c. Upon Orchestrator initialization, both protocols begin to listen for connections on the same port, specified in the node configuration; this implies that both clients and peers should contact a node by
- d. using the same port (although with two different protocols).

States In Rafty:

- The Raft states are represented by three classes. Only one instance of a single class can exist at any given time; that instance manages all of the incoming requests. Upon initialization, the Orchestrator instantiates a Follower class.
- Each class contains the following types of methods:
 - a. **__init__**: initialize the State
 - b. **__teardown__**: frees timers and resources before leaving the State
 - c. **on_client_VERB**: respond to client request
 - d. **on_peer_VERB**: respond to peer request
 - e. **on_peer_response_VERB**: handle peer response
 - f. Periodic actions executed with a timeout
- **State** State is an abstract class that holds all the variables and methods common to every Raft state such as the client get method.
- State is subclassed by the Follower and the Leader classes; the Candidate class subclasses the Follower class instead.

Pseudo Code for describing the State class

```
1 on client_append(client):
2     reply redirect
3 on client_config(client):
4     reply redirecy
5 on client_get(client):
6     reply state_machine
```

- **Follower:** This state inherits from the State class,
 1. it is able to respond to request vote,
 2. append entries peer requests and it owns the restart election timer, which, upon expiration,
 3. triggers a state change to Candidate. The timer is set to a random value in the 0:1 to 0:4s range and
 4. is reset every time the node receives a message from its Leader.

Pseudo Code for describing the Follower class

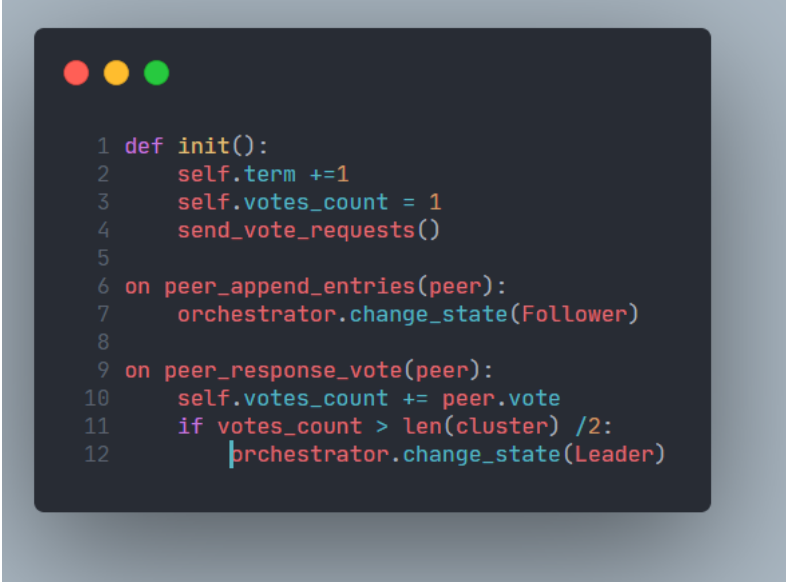
```

1 def init():
2     election_timeout.start()
3 def teardown():
4     election_timeout.stop()
5
6 on peer_request_vote(peer):
7     term_is_current = peer.term ≥ self.term
8     can_vote = self.votedFor is None
9     index_is_current = peer.lastLogTerm ≥ self.log[-1].term
10    reply voteGranted: term_is_current and can_vote and index_is_current
11
12 on peer_append_entries(peer):
13     election_timeout.reset()
14     self.leader = peer
15     if peer.term ≥ self.term and self.log[-1].term = peer.prevLogTerm:
16         if peer.compacted_log:
17             self.log = peer.compacted_log
18         else:
19             self.log.append(peer.entries)
20             self.log.commit(peer.leaderCommit)
21     reply: success: true
22 else:
23     reply: success: false
24 on election_timeout():
25     orchestrator.change_state(Leader)

```

- **Candidate:** This state inherits from the Follower class.
 1. Upon initialization, it broadcasts a vote request throughout the cluster.
 2. The on peer append entries method is overwritten to force a
 3. transition back to the Follower state in the event that an append entries message is received with
 4. some significant lag from the Leader.

Pseudo Code for Candidate Class



```
1 def init():
2     self.term += 1
3     self.votes_count = 1
4     send_vote_requests()
5
6 on peer_append_entries(peer):
7     orchestrator.change_state(Follower)
8
9 on peer_response_vote(peer):
10    self.votes_count += peer.vote
11    if votes_count > len(cluster) / 2:
12        orchestrator.change_state(Leader)
```

- **Leader** The leader is the most complex of the three states.
 1. Its internal state consists of the
 2. matchIndex variable, which is a dictionary where the keys are other nodes addresses and the values
 3. are all initialized to 0, and nextIndex with the same keys as matchIndex, but initialized to the last
 4. committed log entry's index +1.
 5. The waiting client's variable holds a dictionary which associates to every uncommitted log entry
 6. the client connections that are waiting for that entry to be committed for returning a response.
 7. Finally, append timer holds a timer randomly selected in the 0:01 to 0:04s interval which triggers.
 8. the append entries broadcast.

Pseudo Code for Leader Class

```

1 def init():
2     self.leader = self
3     matchIndex = {peer: 0 for peer in cluster}
4     nextIndex = {peer: self.log.index + 1 for peer in cluster}
5     waiting_clients = {}
6     start_append_timer()
7
8 def teardown():
9     stop_append_timer()
10    stop_config_timer()
11    for clients in waiting_clients:
12        for client in clients:
13            client.reply_success: False
14
15 def send_append_entries():
16     for peer in cluster:
17         msg = {'term': log.currentTerm,
18              'leaderCommit': log.commitIndex,
19              'leaderId': self.address,
20              'prevLogIndex': nextIndex[peer] - 1,
21              'prevLogTerm': log.term[nextIndex[peer] - 1],
22              'entries': log[nextIndex[peer]: nextIndex[peer] + 100],
23              }
24         if nextIndex[peer] ≤ log.compacted.index:
25             msg['compacted'] = log
26         orchestrator.send(peer, msg)
27
28 on peer_response_append(peer):
29     if peer.success:
30         matchIndex[peer] = peer.matchIndex
31         nextIndex[peer] = peer.matchIndex + 1
32
33         commit_index = statistics.median_low(matchIndex.values)
34         log.commit(commit_index)
35         send_client_append_response()
36     else:

```



```

37     nextIndex[peer] = max(0, nextIndex[peer] - 1)
38
39 on client_append(client):
40     log.append_entries(client.entry)
41     waiting_clients[log.index].append(client)
42
43 def send_client_append_response(client):
44     for index, clients in waiting_clients:
45         if index ≤ log.commitIndex:
46             for client in clients:
47                 reply success: True
48             del waiting_clients[index]
49
50 on client_config(client):
51     if other configs are pending:
52         retry in 0.4s
53     if client.action == 'add' and client.peer not in cluster:
54         cluster.add(client.peer)
55         nextIndex[client.peer] = 0
56         matchIndex[client.peer] = 0
57         success = True
58     elif client.action == 'delete' and client.peer in cluster:
59         cluster.remove(client.peer)
60         del nextIndex[peer]
61         del matchIndex[peer]
62         success = True
63     else:
64         success = False
65
66     if success:
67         log.append_entries(key='cluster', 'value': cluster, action='change')
68     reply success: success
69

```

LOG MANAGEMENT

- In order to achieve the maximum separation of concerns, the LogManager is implemented in a standalone module, which could potentially be replaced in order to implement new models of the state machine other than the current dictionary-based one.
- The **LogManager module** is in charge of managing the actual state machine Log, the state machine itself, the log compaction mechanism and finally abstracting all of the previous-mentioned components into a simple interface inspired by the list and dictionary Python data structures.
- The module is divided into the following classes:
 1. **Log** The Log class subclasses the Python list and stores the actual state machine command entries.
 - The Python list is expanded with methods that ensure the persistence to disk after every change to this structure.
 2. **Compactor:** The Compactor takes a snapshot of the state machine and persists it to disk, with additional information about the last compacted item, namely: its index and its term.
 3. **DictStateMachine:** The Dict State Machine is a subclass of the Python dictionary with the addition of a method that translates state machine commands into dictionary operations.
 4. **LogManager** The LogManager exposes a simplified interface to the above mentioned components for the rest of the application: the log entries are accessed using the Python method `__getitem__`, the current index and term are accessible through respective properties, and ,
 5. Finally, a commit method is exposed in order to advance the commit index.
- An internal compact method manages the compaction process as follows: every 0:01s the amount of committed and uncompact entries is measured, if greater than a given parameter (60 by default), the compaction is started.
- Although these constants have been chosen from experiments that have shown for them to provide the best results in terms of CPU time and peer messages length, a dynamic calculation based on the current load is planned.

CLIENT

- Raft client is structured into a fundamental class **AbstractClient** that implements the communication primitives used to interact with the server, and a **DistributedDict** class that implements the actual dictionary based client.
- **AbstractClient:** The primitives exposed by the **AbstractClient** are:
 1. **_request** which, given a dictionary representation of a requests, which performs the actual msgpack encoding/decoding and TCP request, given a dictionary based request representation.
 - a. This method also takes care of bouncing to the correct server upon receiving a redirect message.
 2. **_get_state** retries the state machine, selecting a random node of the cluster as a target.
 3. **_append_log** appends to the remote log.
 4. **_diagnostic** retries the diagnostic information for a given node.
 5. **_config_cluster** requests a cluster reconfiguration (that is, adding or deleting a node).
- **DistributedDict** The **DistributedDict** is a mixing of the Python dictionary class and the **AbstractClient** class. It therefore exposes a typical dictionary interface by binding it to the **AbstractClient** methods.
- Refresh Policies: - By default, the **DistributedDict** refreshes its state machine by fetching it from the cluster before every read. Although this approach is guaranteed to never return outdated reads, it can easily lead to significant slowdowns, especially in case of high client-server latencies or high frequency reads.
 - In order to solve this problem, rafty offers an easy mechanism called the RefreshPolicies for caching the state machine for a few read operations, based on different conditions.
 - i. **RefreshPolicyAlways** This Policy forces the client to update itself before every read.
 - ii. **RefreshPolicyLock** This Policy provides a toggle that can be flipped to arbitrarily choose when to enable or disable pre-read updates.
 - iii. **RefreshPolicyCount** This Policy updates the client after n reads, where n can be configured on initialization or at runtime.
 - iv. **RefreshPolicyTime** This Policy forces an update only if a specific amount of time has passed since the last update. The delay is expressed with a Python timedelta object from datetime module.

Federated Learning + RAFT

Vanilla Federated Networks

1. The main idea is to have an “aggregator node” and a set of “participant nodes” in the federated network.
2. The participant nodes send their local model gradients to the aggregator node, and the aggregator node collectively composes all the received models together into a singular global model, which is again sent back to all participant nodes.
3. In this setup, the participants benefit from common learning, without knowing about all the underlying data. While federated learning does take care of privacy, security and anonymity, aggregator nodes become the single point of failure in the federated network.
4. Most current implementations like PySyft rely upon centralized servers to do the final aggregation on the local gradient outputs.

SOLUTION

1. To solve the single point of failure issue, the idea is to use a fully peer-to-peer network for implementation with equivalent capabilities on the network and, all of them should be able to communicate with each other.
2. The network, at any given point of time, has only one node, that would be acting as a “leader” and rest of the nodes will act as “followers”.
3. In case the leader crashes, the network is intelligent enough to conduct a new leader election, and select the leader among the remaining peers. The said mechanism follows a raft consensus to conduct elections and assign leaders.
4. To distribute the workload among peers, and give a fair chance to every peer within the network to assume the role of a “leader”, a new leader is elected after a predefined time interval is elapsed.
5. If the current leader becomes unavailable or unresponsive for any reason, the network is capable of recovering, by electing a new leader among the peers, by conducting fair elections. This ensures each node gets a fair chance of performing aggregation.
6. The peer-to-peer network uses an RPC-mechanism, to provide reliable, highly available and robust mechanisms. This eliminates the computational overhead of maintaining the data consistency over the network.

IMPLEMENTATION SPECIFIC DETAILS

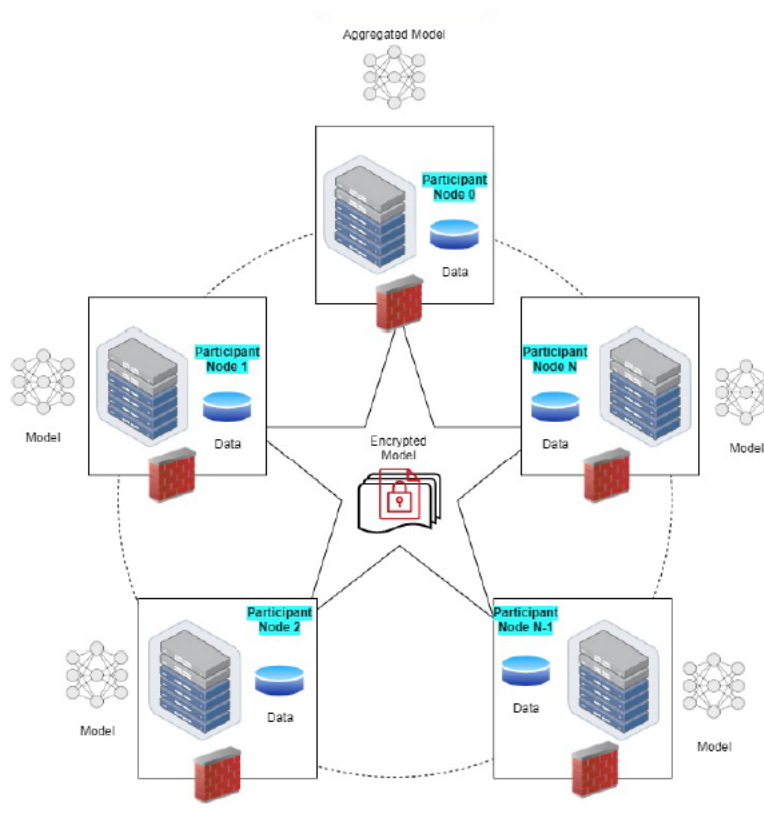
- Any healthy peer in the network could assume any of the roles:
 - a. **Leader (Aggregator) node** - for aggregation role
 - b. **Follower (Participant) node** - participants in the network
 - c. **Candidate node** - potential peers, who contest in the election
- The peer-to-peer communication between different participants leverages the existing remote procedure calls, using the TCP layer of networking. Since it's a peer-to-peer communication, all the messages sent by a specific node are sent as broadcast to the network.
- All the messages from a given node are broadcast to the network, just as in any other p2p network.
- Based on the message type and if encrypted, only the intended recipient can read the message. If the messages are not encrypted, all the nodes can read and perform necessary action. (Note: the security methodology for the messages and how to ensure it shall not be discussed at length.)

Federated Learning + Raft ARCHITECTURE[10]

- The major components of federated learning in peer-to-peer setup include:
 - 1) Leader election and resignation
 - 2) Handling fault and re-election
 - 3) Model aggregation by the leader
 - 4) Model information broadcast by the leader
 - 5) Model fetch, initiated by follower
- A leader, when elected by raft consensus algorithm, assumes the role of the model aggregator, which is transient by design.
- In normal network operation, the elected node stays as leader for T period, as governed by RAFT design.
- A leader is elected only for a specific duration, and at the end of which another
- leader is elected from the available nodes. In order to ensure the leader is available for that duration, all the participant nodes expect a heartbeat every predefined frequency from the leader.
- A missed heartbeat is an indication of unavailability of the leader. In such a case, some peers from the network may promote themselves as candidate nodes for the next leader.
- After a fair election, one of the candidate nodes gets elected as leader and the rest of the nodes continue to remain followers till the next election.

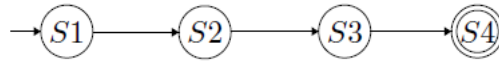
- After the network assigns leadership to the elected node, the node behaves as an aggregator node, starting with sending broadcast messages with its newly generated RSA key pair's public key.
- The message payload can contain information related to model exchange, updates, etc to the aggregator node.
- By design, all the messages will be sent as broadcast on the network. The unencrypted messages are available for all of the participants and each participant can read and act as necessary.
- However, the messages that are intended for certain participants would be encrypted and only nodes with valid private keys can decrypt the message.
- Leader node sends the model's information, at a given frequency to all the participant nodes.

Architecture of Federated Learning with Raft



Description of the states

- S1: Participant node encrypts the message with the aggregator node's public key.
- S2: Aggregator node decrypts the message using private key of itself
- S3: Aggregator node encrypts the message using participant node's public key
- S4: Participant node decrypts the message using the private key generated in the initial state.



Dynamic Message Encryption Decryption

CONCLUSION

TESTING METHODS:

- **Local Testing**

- Local tests are run with a single command line script that accepts as parameters:
- The number of server instances, the number of client instances, the quantity of total entries to create
- and the dimension of each entry.
- Such a task can benefit greatly of all the available cores of a machine, therefore it is implemented
- using the multiprocessing. Pool module, which cores multiple separate interpreters to achieve multiprocessing.
- When finished, the script reports information about:
 - Average duration across all the clients [seconds]
 - Standard deviation of the duration across all the clients
 - Cumulative requests per second
 - Average time per request
 - Total weight in MB of the log items

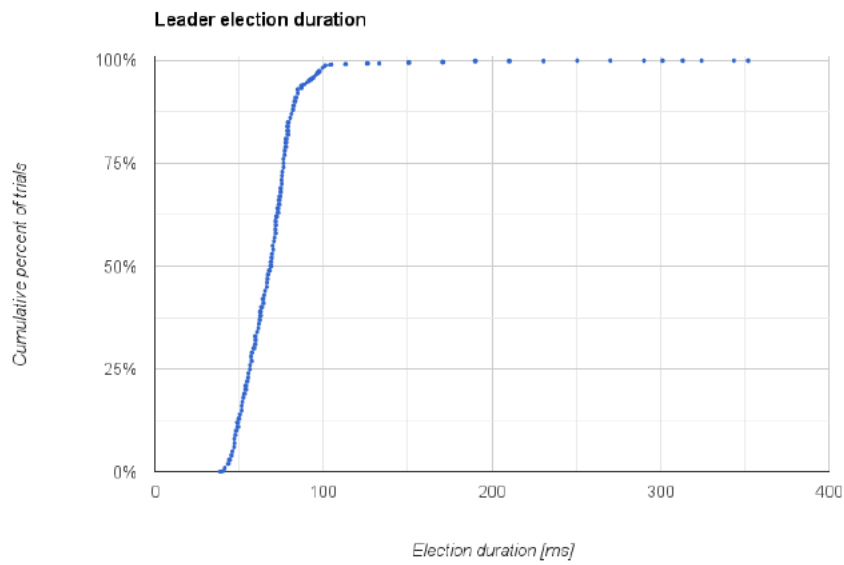
The local test machine is i7-11800H processor with 16GB RAM.

- **Distributed Testing**

- The distributed load testing script can be launched in two modes: server or client. The former launches a single instance of the server, while the latter can manage more than one client to take full advantage of multicore machines.
- Some steps have been taken in order to ease as much as possible the configuration process: only one "head" node has to be manually configured specifying the amount of log entries, workers per physical client and dimension of the log entry.
- This node then starts as a server and adds to its state machine the test configuration data. The rest of the nodes involved in the test only need the address of the head node; once connected to it, they will automatically fetch the configuration data and synchronize the timings for the test among themselves.
- After the test execution, each node pushes its performance results to the distributed state machine, where it is downloaded and persisted to disk by the head node.
- The test setup was built on the Amazon Web Services platform. In this case, two C4, large instances (Intel Xeon E5-2666 v3, 8GB ram) machines constituted the server cluster, and two identical machines were used for the clients.

RESULTS AND CONCLUSIONS

1. **Local Test Results:** For what concerns Local testing, repeated trials showed that:
 - i. A single client can dispatch 100 requests/s at most
 - ii. The maximum write throughput of 2000 requests/s is obtained with a nominal cluster of 3 servers and 64 clients
 - iii. The same setup slowly decreases in throughput down to 1500 requests/s with 256 concurrent clients
2. Those outlined above are the limits of the developer's machine, after which the servers get pre-empted too often and the Leader doesn't manage to remain in control of the cluster.
3. **Distributed:** The distributed environment was configured for iterating over several tests consisting of all the permutations of the following parameters:
 - a. 10; 100; 1000; 5000; 10000 log entries
 - b. 10; 100; 256 clients per physical server
 - c. 100; 1000 bytes of payload per entry
4. **Write tests:** Peak throughput was reached at 500requests/s, and this result was independent of the entry size.
5. **Election speed** For this benchmark, a cluster of 5 was setup on the AWS platform, where the latency between each pair of machines was lower than 4ms. For what concerns election speed, the results are similar to the reference implementation (Log-Cabin) as can be seen in the graph below.



In the Benchmark with 1000 trials, 87% of the elections terminated in under 80ms, 98% terminated in under 100ms and very few trials were in the hundreds of ms region.

References

- [0] <https://manipal.edu/mu/directorate-of-research.html>
- [1] <https://arxiv.org/abs/1812.02903>
- [2] <https://github.com/microsoft/SEAL>
- [3] <https://lamport.azurewebsites.net/pubs/paxos-simple.pdf>
- [4] <https://raft.github.io/raft.pdf>
- [5] <https://www.educative.io/edpresso/how-does-the-leader-election-in-raft-work>
- [6] <https://eli.thegreenplace.net/2020/implementing-raft-part-2-commands-and-log-replication/>
- [7] <https://wiki.python.org/moin/GlobalInterpreterLock>
- [8] <https://eng.paxos.com/python-3s-killer-feature-asyncio>
- [9] <https://msgpack.org/index.html>
- [10] https://www.techrxiv.org/articles/preprint/Federated_Learning_using_Peer-to-peer_Network_for_Decentralized_Orchestration_of_Model_Weights/14267468