UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in Informatica

Elaborato finale

Zatt

A Python implementation of the Raft algorithm for distributed consensus

Supervisore
Alberto Montresor

Laureando
Simone Accascina

Anno accademico 2016/2017

Contents

Chapter 1

Summary

This technical report describes Zatt: an implementation of the Raft consensus algorithm in the Python programming language.

1.1 Motivation

When the work on Zatt was started, four Python implementations of Raft were listed on the official website [6], but none of them was feature-complete or in active development. Since Python is one of the most widely used programming languages of our time [10] [9], the Zatt project was started, with the goal of adding a usable, feature-complete library for state machine replication to the Python ecosystem.

1.2 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, 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 *state variables* and *commands* which can either transform the state variables or produce some output. Furthermore, commands are deterministic and their execution is atomic, this allows to represent the state variables as an ordered collection of commands.

The problem of state machine replication is an old one and was first described in [11]. In 1989 Leslie Lamport provided the Paxos algorithm [12] that elegantly solved the state machine replication problem, with a proof of its safety; since then, Paxos has been widely used in major systems such as Google distributed lock manager Chubby, the Neo4j graph database, the Apache Mesos cluster manager and others.

Despite its popularity, a widely perceived shortcoming of Paxos is the difficulty in understanding and implementing it, enhanced by the fact that working implementations need significant changes from the specification (see multi-Paxos). Diego Ongaro and John Ousterhout developed the Raft algorithm at Stanford in 2013 to address the aforementioned issues, resulting in an easier algorithm with a better understandability, proven by a user study conducted with university students.

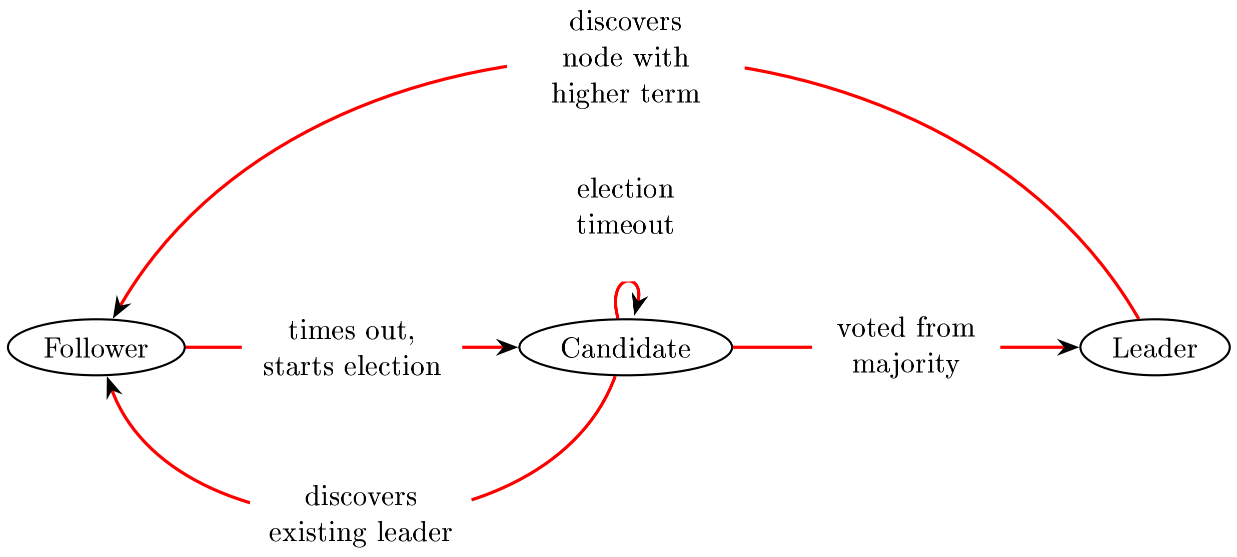
1.3 Raft

The 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 two separate mechanisms: **leader election** and **log replication**.

1.3.1 Leader election

The leader election subsystem is in charge of managing the role of nodes in the cluster, and works as follows: each server in a 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 the order of a few hundreds of

Figure 1.1: Raft states and allowed transitions between them


discovers node with higher term
election timeout
Follower
times out, starts election
Candidate
voted from majority
Leader
discovers existing leader

milliseconds) where no heartbeat was received by a Leader, each Follower convert itself to Candidate. In doing so, it votes for himself and sends 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 ties).

1.3.2 Log replication

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

1.3.3 Membership changes

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

1.4 Zatt

In the spirit of "Pythonic" code, Zatt presents itself to the user as a simple dictionary. However, a Zatt dictionary can be automatically replicated across multiple machines and is highly available.

The Zatt project is now complete and published on the Pypi repository; a test suite has been developed to ensure strict adherence to the API exposed to clients and for future regression testing. 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.

1.4.1 Python compatibility

Concerning the specifics of the implementation, although version 3 of the Python language is now mature and preferable for new software projects [8], version 2 is extremely popular and will still be supported until 2020. In fact, some major companies maintain large codebases in Python2 that are not being migrated at the present moment [7], [4] [3]. This is the reason why the Zatt client was designed to be compatible with both major versions of the language.

The Zatt server on the other hand is only compatible with Python 3.5 or newer, because of the use of the `asyncio` library and the async/await concurrency primitives; this limited compatibility should not pose issues with adoption since the server is a standalone application that can be installed alongside preexisting Python2 codebases.

1.4.2 Concurrency model

The implementation of the server posed interesting challenges regarding the choice of a concurrency model: because of the the global interpreted lock [2] (GIL), an instance of the CPython interpreter is only able to execute a single instruction at any given point in time, therefore classic concurrency strategies like multithreading are not as effective in CPython (the most widely used implementation of the Python language) as they are in other environments. Other alternatives consisted of multiprocessing and the new `asyncio` core library [1]: the latter, a library for single-threaded asynchronous programming featuring a pluggable event loop, coroutines and socket I/O access multiplexing, was chosen for ease of development and better performance.

1.4.3 Networking

For what concerns the networking, intra-cluster communication uses UDP as a transport protocol, while TCP is used for client-server messages. While 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 mechanisms.

1.4.4 Serialization

Messages for both intra-cluster and client-server communication are serialized using the MessagePack [5] data interchange format; originally JSON was chosen for this task, however MessagePack presents many compelling features that favored the switch, namely: a binary representation that allows for more compact messages, a faster de/serializer implemented in C++, support for a greater range of datatypes, in particular MessagePack allows for non-string keys in the map datatype.

1.4.5 Deployment

A Docker image based on Alpine Linux containing the Zatt server was created and made available on the Docker Hub. The image was initially created to facilitate the internal benchmarking process but will be maintained nonetheless to ease the adoption process for new developers.

Chapter 2

Zatt implementation

Zatt 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 Zatt 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.

2.1 Protocol

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

2.1.1 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 fields, in particular, the `type` field determines the purpose of the message as outlined below.

Key	**Value**
`"type"`	"get"

Table 2.1: Get request

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.

Every machine in the cluster is able to respond to this request; for this reason, clients randomize the choice of which server to target for every request in order to evenly distribute the load.

Response Since this request cannot fail, the state machine is returned as a map without additional metadata.

Append

(a) Append request

Key	Value	
"type"	”append”	
"data"	**Key**	**Value**
	"action"	”change”
	"key"	key
	"value"	value

(b) Append response

Key	Key
"type"	”result”
"success"	boolean

This 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` response along with the address of the current leader. This iterative approach to leader discovery is preferable to a recursive one in which the Follower relays the request to the Leader for efficiency reasons. In fact, all the subsequent requests will directly target the leader. In the event of a failure of the current leader, a client will send its requests to another random node of the cluster, hoping to be redirected to the new leader.

Diagnostic This request is used to retrieve information from any of the servers about their status.

Table 2.3: Diagnostic request

Key	Value
"type"	”diagnostic”

Table 2.4: Diagnostic response

Key	Key	
"status"	"Leader", "Follower", "Candidate"	
"persist"	**Key**	**Value**
	"votedFor"	address:port
	"currentTerm"	integer
"volatile"	**Key**	**Value**
	"leaderId"	address:port
	"cluster"	[address1:port1, address2:port2,]
	"address"	address:port
"log"	**Key**	**Value**
	"commitIndex"	integer
"stats"	**Key**	**Value**
	"reads"	[integer,]
	"writes"	[integer,]
	"append"	[integer,]

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.

Delete This request is used to delete an entry from the state machine.

(a) Delete request

Key	Value	
"type"	"append"	
"data"	**Key**	**Value**
	"action"	"delete"
	"key"	key

(b) Delete response

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` field has to be set to `add` and the `address,port` fields to the new node's IP and port. A similar syntax is in place for node removal.

2.1.2 Intra-Cluster Protocol

Intra-cluster networking was also initially build on top of TCP, but this transport protocol was quickly deemed suboptimal for the task since it clashes in some respects with the functionalities provided by

(a) Reconfiguration request

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

(b) Reconfiguration response

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

Raft such as: segment retransmission and sequencing. As a result, the UDP protocol was finally chosen for intra-cluster networking.

An example of the above mentioned functionality overlap is the retransmission logic: 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. Furthermore, 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 summarized below, consist of nested `map` msgpack objects with varying fields.

Vote request This request is broadcasted [1] to the whole cluster by a node upon transitioning to the Candidate state,.

Each node will respond with a boolean value representing its vote.

(a) Request vote

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

(b) Vote request response

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

Append entries The 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.

The 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.

2.2 Serialization

In the infancy of this project, the **json** format was used for serialization of objects in both network messages and disk persistence for several reasons: first, 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. Furthermore, a JSON library is integrated into the Python standard library, a

[1] In the current implementation, the server performs broadcast by sending a packet to every individual server

(a) Append entries request

Key	Value
"type"	"append_entries"
"term"	integer
"leaderCommit"	integer
"leaderId"	address:port
"prevLogIndex"	integer
"prevLogTerm"	integer
"entries"	log entries

(b) Append entries response

Key	Value
"type"	"response_append"
"term"	integer
"success"	boolean
"matchIndex"	integer

good step towards the goal of minimizing the external dependencies. At a later stage, when the implementation of the core algorithms was complete, testing showed how the de/serialization was occupying much of the CPU time on the Leader node. At first, the standard library's json implementation was substituted with the more efficient `ujson` library written in C, but the gains were marginal, going from 23% to 18% of cpu time spent in de/serialization. This prompted a survey of other major serialization formats, in order to find a better alternative, whose results are summarized in Table 2.9.

In the end, **MessagePack** was determined to be the best choice for the Zatt project because of the combination of four features: Good performance (3 – 4%) of the CPU time), a binary encoding that allows for space-efficient serialization, the similarity with JSON in being a schemaless protocol, thus not requiring changes in the application logic; finally, the capability of appending to an array in-place is useful for cheaply persisting the log to disk. A patch to the MessagePack algorithm had to be developed in order to support the last mentioned capability, which is necessary for cheaply persisting log entries to disk.

In particular, MessagePack serializes arrays as a variable amount of bytes (from 1 to 4) containing the array length, followed by the actual array elements.

While this approach is good for minimizing the memory footprint, updating an already serialized object in place by appending additional items only works up to the maximum length representable by a single byte (the first). Subsequently, the programmer would have to shift all the elements by one byte n order to be able to increment the counter past the first byte. Furthermore, this shift operation becomes increasingly burdensome for each additional byte.

With the a priori knowledge that Zatt persists fairly long arrays to disk, we can alleviate the load on the persistence subsystem with a simple modification to MessagePack: we directly allocate the full four bytes at the beginning of the file, thus making an append operation always consist of just a write to the beginning of the file, a seek to the end of it, and a final write.

Figure 2.1: MsgPack flexible serialization header

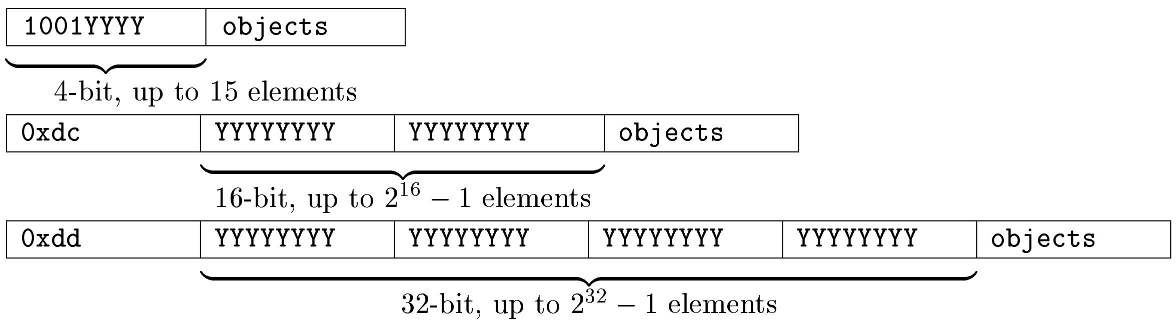


Table 2.9: Feature comparison of the candidate serialization formats

	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

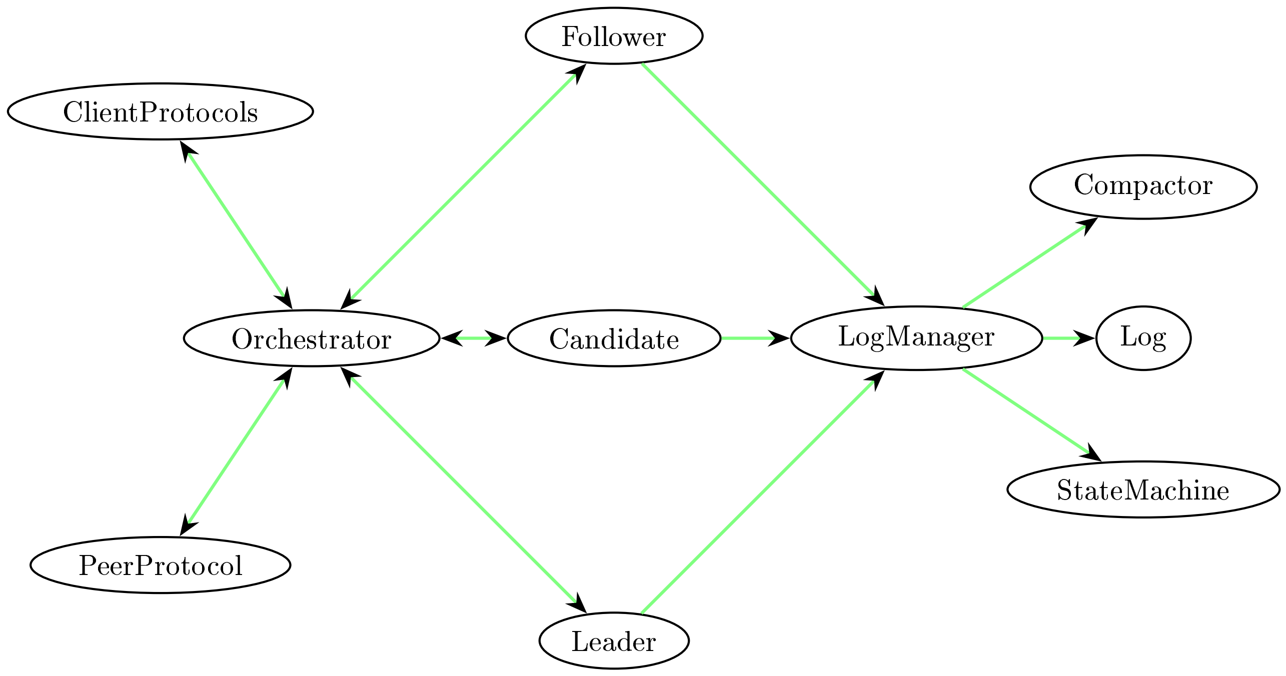
2.3 Server

The Zatt server is a single threaded application with a non-preemptive concurrency model based on coroutines, multiplexed nonblocking I/O and an event loop.

2.3.1 Architecture

The Zatt server comprises the following components: several `States`, several `Protocols` and an `Orchestrator` to manage the application.

Figure 2.2: Zatt subsystems and their relationships



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.

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 using the same port (although with two different protocols).

States In Zatt, 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:

- `__init__`: initialize the State
- `teardown`: frees timers and resources before leaving the State
- `on_client_VERB`: repond to client request
- `on_peer_VERB`: respond to peer request
- `on_peer_response_VERB`: handle peer response
- 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.

Listing 2.1: Pseudocode describing the `State` class

```
on client_append(client):
  reply redirect
on client_config(client):
  reply redirecy
on client_get(client):
  reply state_machine
```

Follower This state inherits from the `State` class, it is able to respond to `request_vote`, `append_entries` peer requests and it owns the `restart_election` timer, which, upon expiration, triggers a state change to `Candidate`. The timer is set to a random value in the $0.1 - 0.4s$ range and is reset every time the node receives a message from its `Leader`.

Listing 2.2: Pseudocode describing the `Follower` class

```
def init():
  election_timeout.start()
def teardown():
  election_timeout.stop()

on peer_request_vote(peer):
  term_is_current = peer.term >= self.term
  can_vote = self.votedFor is None
  index_is_current = peer.lastLogTerm >= self.log[-1].term
  reply voteGranted: term_is_current and can_vote and index_is_current

on peer_append_entries(peer):
  election_timeout.reset()
  self.leader = peer
  if peer.term >= self.term and self.log[-1].term == peer.prevLogTerm:
    if peer.compacted_log:
      self.log = peer.compacted_log
    else:
      self.log.append(peer.entries)
      self.log.commit(peer.leaderCommit)
    reply: success: true
  else:
    reply: success: false
on election_timeout():
  orchestrator.change_state(Leader)
```

Candidate This state inherits from the `Follower` class. Upon initialization, it broadcasts a vote request throughout the cluster. The `on_peer_append_entries` method is overwritten to force a transition back to the Follower state in the event that an `append_entries` message is received with some significant lag from the `Leader`.

Listing 2.3: Pseudocode describing the `Candidate` class

```

def init():
  self.term +=1
  self.votes_count = 1
  send_vote_requests()

on peer_append_entries(peer):
  orchestrator.change_state(Follower)

on peer_response_vote(peer):
  self.votes_count += peer.vote # in Python, int(True) == 1, int(False) == 0
  if votes_count > len(cluster) /2:
    orchestrator.change_state(Leader)
```

Leader The leader is the most complex of the three states: its internal state consists of the `matchIndex` variable, which is a dictionary where the keys are other nodes addresses and the values are all initialized to 0, and `nextIndex` with the same keys as `matchIndex`, but initialized to the last committed log entry's index +1.

The `waiting_clients` variable holds a dictionary wich associates to every uncommitted log entry the client connections that are waiting for that entry to be committed for returning a response.

Finally, `append_timer` holds a timer randomly selected in the $0.01 - 0.04s$ interval which triggers the `append_entries` broadcast.

Listing 2.4: Pseudocode describing the `Leader` class

```
def init():
  self.leader = self
  matchIndex = {peer: 0 for peer in cluster}
  nextIndex = {peer: self.log.index + 1 for peer in cluster}
  waiting_clients = {}
  start_append_timer()

def teardown():
  stop_append_timer()
  stop_config_timer()
  for clients in waiting_clients:
    for client in clients:
      client reply success: False

def send_append_entries():
  for peer in cluster:
    msg = {'term': log.currentTerm,
           'leaderCommit': log.commitIndex,
           'leaderId': self.address,
           'prevLogIndex': nextIndex[peer] - 1,
           'prevLogTerm': log.term[nextIndex[peer] - 1],
           'entries': log[nextIndex[peer]: nextIndex[peer] + 100],
        }
    if nextIndex[peer] <= log.compacted.index:
      msg['compacted'] = log
    orchestrator.send(peer, msg)

on peer_response_append(peer):
  if peer.success:
    matchIndex[peer] = peer.matchIndex
    nextIndex[peer] = peer.matchIndex + 1

    commit_index = statistics.median_low(matchIndex.values)
    log.commit(commit_index)
    send_client_append_response()
  else:
    nextIndex[peer] = max(0, nextIndex[peer] - 1)

on client_append(client):
   log.append_entries(client.entry)
   waiting_clients[log.index].append(client)

def send_client_append_response(client):
  for index, clients in waiting_clients:
    if index <= log.commitIndex:
      for client in clients:
        reply success: True
      del waiting_clients[index]

on client_config(client):
  if other configs are pending:
    retry in 0.4s
  if client.action == 'add' and client.peer not in cluster:
    cluster.add(client.peer)
    nextIndex[client.peer] = 0
    matchIndex[client.peer] = 0 # the new peer needs the complete log
    success = True
  elif client.action == 'delete' and client.peer in cluster:
    cluster.remove(client.peer)
    del nextIndex[peer]
    del matchIndex[peer]
    success = True
  else:
    success = False

  if success:
    log.append_entries(key='cluster', 'value': cluster, action='change')
  reply success: success
```

2.3.2 Log managment

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 above mentioned components into a simple interface inspired by the `list` and `dictionary` Python data structures.

The module is divided into the following classes:

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.

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.

DictStateMachine The Dict State Machine is a subclass of the Python `dictionary` with the addition of a method that translate state machine commands into dictionary operations, namely:

- {`'action:'change','key': key, 'value': value`} → `state_machine[key] = value`
- {`'action:'delete','key': key'`} → `del state_machine[key]`

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 so called Python "special" method `__getitem__`, the current index and term are accessible trough respective properties, and 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 uncompacted 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.

2.4 Client

The Zatt 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. This design promotes the creation of new clients by maximizing the code reuse.

AbstractClient The primitives exposed by the `AbstractClient` are:

`_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. This method also takes care of bouncing to the correct server upon receiving a `redirect` message.

`_get_state` retries the state machine, selecting a random node of the cluster as a target.

`_append_log` appends to the remote log.

`_diagnostic` retries the diagnostic information for a given node.

`_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.

This issue cannot be solved a priori in Zatt since the tolerance for outdated reads is highly dependent on the specific use case where this library is used.

For this reason, Zatt offers an easy mechanism called `RefreshPolicies` for caching the state machine for a few read operations, based on different conditions.

RefreshPolicyAlways This Policy forces the client to update itself before every read.

RefreshPolicyLock This Policy provides a toggle that can be flipped to arbitrarily choose when to enable or disable pre-read updates.

RefreshPolicyCount This Policy updates the client after n reads, where n can be configured on initialization or at runtime.

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 `datetime.timedelta` object.

Chapter 3

Performance Evaluation

Distributed databases such as Zatt have obvious performance requirements due to the concurrent usage from a possibly vast quantity of clients. Moreover, Zatt is written in Python and is therefore single-threaded, which makes this optimization task even more critical.

In order to address the performance concern, two types of load testing were devised: firstly, **local** tests run on the developer's machine and provide a quick feedback on the performance gain/loss for every VCS commit for regression testing; these tests were mainly used during development in tandem with profilers to optimize the server by identifying and eliminating bottlenecks.

The other type of tests are **distributed** tests, where the user can setup an arbitrary number of clients and servers, and the system takes care of generating random data, running the test and finally reporting the timing results to the user in a unified form. Distributed tests are useful for the end users of the library, to evaluate the performance in the particular deployment scenario.

Both test frameworks are capable of measuring three parameters:

- Read speed and throughput
- Write speed and throughput
- Election duration

All the test scripts are located in the Zatt repository under the `tests` directory.

3.1 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 fires 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

Local testing was carried out on a laptop with an Intel i7-4600U @ 2.10GHz (two physical cores, 4 logical ones), 8GB of DDR3 RAM @ 1600Mhz.

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 head can also schedule multiple increasingly challenging tests at once to be automatically executed and saved. Finally, the tests can be run in a supervised fashion, where the user is regularly prompted with the test status on the head console, and can take action by for example repeating unsatisfactory tests.

Distributed testing was carried out on two separate setups:

One cluster was setup with the Scaleway provider, consisting of up to 6 servers with 32GB or DDR3 RAM, an Intel Avoton C2750 @ 2.40GHz CPU and a dual NIC with a total 5Gbps of network throughput. The clients were hosted on 64 4-cores ARMv7 CPU with 2GB of RAM and a 1 Gbit/s network card.

The second 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.

3.2 Results and Discussion

[1]

Local For what concerns **Local** testing, repeated trials showed that:

- A single client can dispatch 100 requests/s at most
- The maximum write throughput of 2000 req/s is obtained with a nominal cluster of 3 servers and 64 clients
- The same setup slowly decrease in throughput down to 1500 requests/s with 256 concurrent clients

Those outlined above are the limits of the developer's machine, after which the servers get preempted too often and the Leader doesn't manage to remain in control of the cluster.

Distributed The distributed environment was configured for iterating over several tests consisting of all the permutations of the following parameters:

- $10, 100, 1000, 5000, 10000$ log entries
- $10, 100, 256$ clients per physical server
- $100, 1000$ bytes of payload per entry

[1]The complete results are available in Appendix A

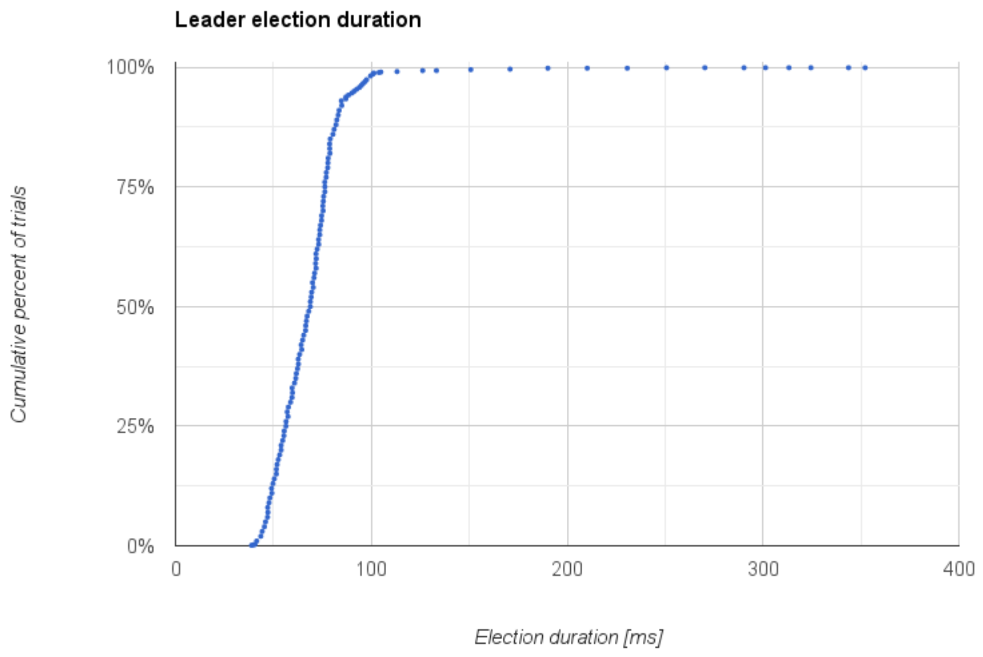
Write tests The results showed that the peak throughput was reached at $500 requests/s$, and this result was independent of the entry size.

It is worth noting that tests with few entries might not have accurate results due to the non-trascurable influence of the Python interpreter setup, which becomes trascurable for longer trials.

Some particular edge cases still present inexplicable slowness that is being investigated at the time of this writing.

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 (LogCabin) as can be seen in the graph below.



In a 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.

Performance data The following table describes results from the read and write benchmarks.

Table 3.1: Distributed test

servers	clients	workers	entries	size [bytes]	time [s]	s/entry	req/s	leader troughput [KB/s]	connections	platform	stability issues
2	1	10	100	100	1.23	0.0123	81	8.13	10	local-i74600U-8GB	
2	1	10	100	1000	1.03	0.0103	97	96.98	10	local-i74600U-8GB	
2	1	10	1000	100	3.05	0.0031	328	32.77	10	local-i74600U-8GB	
2	1	10	1000	1000	2.45	0.0025	408	407.61	10	local-i74600U-8GB	
2	1	100	1000	100	1.34	0.0013	746	74.60	100	local-i74600U-8GB	
2	1	100	1000	1000	2.01	0.0020	498	498.04	100	local-i74600U-8GB	
2	1	256	1000	100	7.39	0.0074	135	13.52	256	local-i74600U-8GB	
2	1	256	1000	1000	8.45	0.0085	118	118.34	256	local-i74600U-8GB	
2	1	10	5000	100	10.71	0.0021	467	46.67	10	local-i74600U-8GB	
2	1	10	5000	1000	12.67	0.0025	394	394.50	10	local-i74600U-8GB	
2	1	100	5000	100	3.41	0.0007	1466	146.60	100	local-i74600U-8GB	
2	1	100	5000	1000	10.36	0.0021	483	482.70	100	local-i74600U-8GB	
2	1	256	5000	100	26.47	0.0053	189	18.89	256	local-i74600U-8GB	
2	1	256	5000	100	10.21	0.0020	490	48.99	256	local-i74600U-8GB	
2	1	256	5000	1000	33.54	0.0067	149	149.06	256	local-i74600U-8GB	Y
2	1	256	5000	1000	33.87	0.0068	148	147.64	256	local-i74600U-8GB	Y
2	1	10	10000	100	20.86	0.0021	479	47.93	10	local-i74600U-8GB	
2	1	10	10000	1000	32.47	0.0032	308	308.02	10	local-i74600U-8GB	
2	1	100	10000	100	6.93	0.0007	1442	144.25	100	local-i74600U-8GB	
2	1	100	10000	1000	22.19	0.0022	451	450.58	100	local-i74600U-8GB	
2	1	256	10000	100	10.06	0.0010	995	99.45	256	local-i74600U-8GB	
2	1	256	10000	1000	14.14	0.0014	707	707.24	256	local-i74600U-8GB	
2	4	10	100	100	0.65	0.0065	153	15.31	40	2-aws-E5-2666v3-32GB	
2	4	10	100	1000	1.04	0.0104	96	96.24	40	2-aws-E5-2666v3-32GB	
2	4	10	1000	100	2.05	0.0021	487	48.68	40	2-aws-E5-2666v3-32GB	
2	4	10	1000	1000	2.07	0.0021	483	482.67	40	2-aws-E5-2666v3-32GB	
2	4	100	1000	100	1.49	0.0015	672	67.25	400	2-aws-E5-2666v3-32GB	
2	4	256	1000	100	16.16	0.0162	62	6.19	1024	2-aws-E5-2666v3-32GB	

Table 3.2: Distributed test

servers	clients	workers	entries	size [bytes]	time [s]	s/entry	req/s	leader troughput [KB/s]	connections	platform	stability issues
2	4	256	1000	1000	10.94	0.0109	91	91.40	1024	2-aws-E5-2666v3-32GB	
2	4	10	5000	100	6.32	0.0013	792	79.17	40	2-aws-E5-2666v3-32GB	Y
2	4	10	5000	1000	7.02	0.0014	712	711.85	40	2-aws-E5-2666v3-32GB	
2	4	100	5000	100	4.17	0.0008	1200	120.04	400	2-aws-E5-2666v3-32GB	
2	4	100	5000	1000	11.85	0.0024	422	421.78	400	2-aws-E5-2666v3-32GB	
2	4	256	5000	100	7.15	0.0014	699	69.92	1024	2-aws-E5-2666v3-32GB	
2	4	256	5000	1000					1024	2-aws-E5-2666v3-32GB	Y
2	4	10	10000	100	12.22	0.0012	818	81.82	40	2-aws-E5-2666v3-32GB	
2	4	10	10000	1000	15.22	0.0015	657	656.90	40	2-aws-E5-2666v3-32GB	
2	4	100	10000	100					400	2-aws-E5-2666v3-32GB	Y
2	4	256	10000	100	9.08	0.0009	1101	110.08	1024	2-aws-E5-2666v3-32GB	
2	4	256	10000	1000	32.01	0.0032	312	312.35	1024	2-aws-E5-2666v3-32GB	
2	2	10	100	100	0.84	0.0084	119	11.94	20	1-aws-E5-2666v3-32GB	
2	2	10	100	1000	1.24	0.0124	81	80.82	20	1-aws-E5-2666v3-32GB	
2	2	10	1000	100	3.05	0.0031	327	32.74	20	1-aws-E5-2666v3-32GB	
2	2	10	1000	1000	2.86	0.0029	350	349.98	20	1-aws-E5-2666v3-32GB	
2	2	100	1000	100	0.91	0.0009	1097	109.66	200	1-aws-E5-2666v3-32GB	
2	2	100	1000	1000					200	1-aws-E5-2666v3-32GB	Y
2	2	256	1000	100	9.11	0.0091	110	10.97	512	1-aws-E5-2666v3-32GB	
2	2	256	1000	1000	10.71	0.0107	93	93.36	512	1-aws-E5-2666v3-32GB	
2	2	10	5000	100	10.91	0.0022	458	45.84	20	1-aws-E5-2666v3-32GB	
2	2	10	5000	1000	11.16	0.0022	448	447.89	20	1-aws-E5-2666v3-32GB	
2	2	100	5000	100	3.13	0.0006	1597	159.75	200	1-aws-E5-2666v3-32GB	
2	2	256	5000	100					512	1-aws-E5-2666v3-32GB	Y
2	2	10	10000	100	21.39	0.0021	468	46.75	20	1-aws-E5-2666v3-32GB	
2	2	10	10000	1000	23.53	0.0024	425	424.95	20	1-aws-E5-2666v3-32GB	
2	2	100	10000	100					200	1-aws-E5-2666v3-32GB	Y
						average	average				
						0.0041	498				

Chapter 4

Future work

Looking forward, some additional features and improvements are possible:

Linarizable semantics Linearizable semantics, as described in Ongaro's dissertation is to be implemented in order to guarantee that each command to the state machine is executed exactly once.

Intra-cluster protocol The current implementation limits the size of a log entry to the maximum size of the UDP datagram. Although this should not be an issue with the typical workload of Zatt, which mainly involves small metadata, a solution could be devised in the form of either sequencing of the UDP packets or by returning to TCP with a workaround pertaining the problems mentioned in section 2.1.2.

Bibliography

[1] The asyncio library. https://docs.python.org/3/library/asyncio.html. last access: 07/2016.

[2] The global interpreter lock. https://wiki.python.org/moin/GlobalInterpreterLock. last access: 07/2016.

[3] Google app engine. https://cloud.google.com/appengine/docs/python/. last access: 07/2016.

[4] Instragram engineering. http://instagram-engineering.tumblr.com/. last access: 08/2016.

[5] Msgpack format. https://msgpack.org/. last access: 08/2016.

[6] Official raft website. https://raft.github.io. last access: 07/2016.

[7] Pyston project. https://github.com/dropbox/pyston. last access: 08/2016.

[8] Python 2 vs 3 decision guide. https://wiki.python.org/moin/Python2orPython3. last access: 07/2016.

[9] Stackoverflow developer survey. https://stackoverflow.com/research/developer-survey-2016#technology. last access: 07/2016.

[10] Tiobe programming language popularity index. http://www.tiobe.com/tiobe-index/. last access: 07/2016.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[12] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.