Data Mining: BeeViva Challenges

MARCO BELLÒ, University of Padua, Italy

All code, images, LaTeXsources, as well as a less polished version of this report in markdown format, split into two files for the two challenges, can be found at the following repository: https://github.com/mhetacc/DataMiningChallenges/.

To be precise:

- Rice Varieties markdown report: https://github.com/mhetacc/DataMiningChallenges/blob/main/RiceChallenge/Rice_report.md
- Rice Varieties R code: https://github.com/mhetacc/DataMiningChallenges/blob/main/RiceChallenge/rice_challenge.r
- Phone Users markdown report: https://github.com/mhetacc/DataMiningChallenges/blob/main/PhoneUserChallenge/Phone_report.md
- Phone Users R code: https://github.com/mhetacc/DataMiningChallenges/blob/main/PhoneUserChallenge/phone_challenge.r

ACM Reference Format:
Marco Bellò. 2025. Data Mining: BeeViva Challenges. 1, 1 (December 2025), ?? pages. https://doi.org/XXXXXXX.XXXXXXX

1 RICE VARIETIES

1.1 Preliminary Observations

1.1.1 Dataset. All following considerations are made using the datasets provided on the challenge page (*rice_test.csv* and *rice_train.csv*), but it is worth noting that they both stem from an original one that can be found at the dataset source page. I used it to compute an alternative version of the *test* dataset that contains the true attribute for *Class*.

Listing 1. Merge datasets

```
df1 = pd.read_csv("rice_test.csv").round(10)
df2 = pd.read_csv("Rice_Cammeo_Osmancik.csv").round(10)

keys = [col for col in df1.columns]
merged = df1.merge(df2, on=keys, how="inner")
```

1.1.2 Scatter Plot. From the *scatter plot* we can infer that there are four features (*Area*, *Perimeter*, *Convex_Area* and *Major_Axis_Length*) that seem to be extremely correlated

1.1.3 Correlation Matrix. We can use a correlation matrix to see exactly how related to each other the features are. The results are as follow.

As suspected, all four features mentioned above have a high degree of correlation (over ninety percent). Once way to handle this is to either use models robust against collinearity, or to either combine or remove some of the correlated features.

Author's address: Marco Bellò, marco.bello.3@studenti.unipd.it, University of Padua, Padua, Italy.

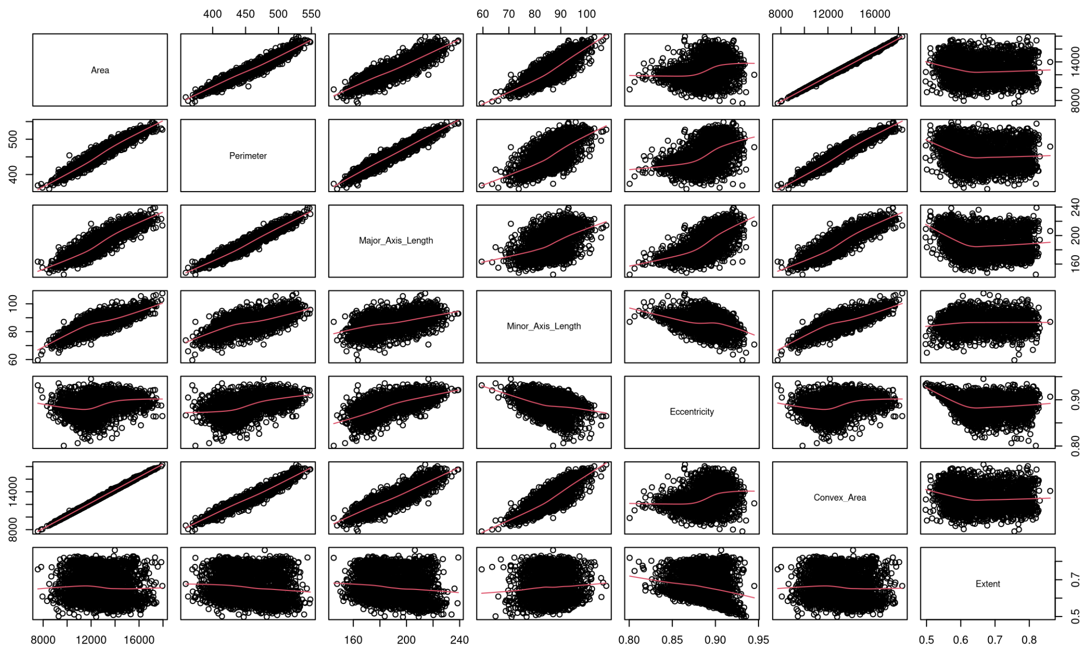


Fig. 1. Scatter Plot for the *rice_train* dataset with LOESS smoothing lines for each class.

Table 1. Correlation matrix of features

Area	Perimeter	Major_Axis_Length	Minor_Axis_Length	Eccentricity	Convex_Area	Extent	
Area	1.00000000	0.96704011	0.90356325	0.79022701	0.34653177	0.99895272	-0.06002223
Perimeter	0.96704011	1.00000000	0.97163180	0.63486482	0.53784650	0.97046788	-0.12718015
Major_Axis_Length	0.90356325	0.97163180	1.00000000	0.45675159	0.70625660	0.90390912	-0.13562592
Minor_Axis_Length	0.79022701	0.63486482	0.45675159	1.00000000	-0.29393931	0.78975480	0.06192558
Eccentricity	0.34653177	0.53784650	0.70625660	-0.29393931	1.00000000	0.34707340	-0.19301157
Convex_Area	0.99895272	0.97046788	0.90390912	0.78975480	0.34707340	1.00000000	-0.06397213
Extent	-0.06002223	-0.12718015	-0.13562592	0.06192558	-0.19301157	-0.06397213	1.00000000

1.1.4 Features' Importance. To get a better idea on the importance of each feature, I trained a simple linear regression model, and looked at the result with `summary(fit)`. The results follows.

Table 2. Regression coefficients and coefficients' importance

Parameter	Estimate	Std. Error	t value	Pr(>|t|)	Importance
(Intercept)	-2.152e+00	1.990e+00	-1.081	0.279633	
Area	5.601e-04	1.023e-04	5.474	4.79e-08	***
Perimeter	8.440e-03	2.221e-03	3.800	0.000148	***
Major_Axis_Length	-2.198e-02	5.709e-03	-3.851	0.000120	***
Minor_Axis_Length	4.669e-02	1.213e-02	3.850	0.000121	***
Eccentricity	4.534e+00	1.828e+00	2.481	0.013170	*
Convex_Area	-8.609e-04	9.418e-05	-9.141	< 2e-16	***
Extent	7.146e-02	7.144e-02	1.000	0.317237	

Table 3. Standardized regression coefficients

Parameter	Standardized Coefficient
(Intercept)	NA
Area	1.95970669
Perimeter	0.60605955
Major_Axis_Length	-0.77272038
Minor_Axis_Length	0.54256082
Eccentricity	0.18931314
Convex_Area	-3.09099064
Extent	0.01114328

Table 4. Summary of principal components

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
Standard deviation	2.2924	1.2436	0.9562	0.51393	0.10621	0.07758	0.04519	0.02052
Proportion of Variance	0.6569	0.1933	0.1143	0.03302	0.00141	0.00075	0.00026	0.00005
Cumulative Proportion	0.6569	0.8502	0.9645	0.99753	0.99894	0.99969	0.99995	1.00000

The significance stars tells us visually that, with the exception of *Minor_Axis_Length*, the features that are highly correlated to each other contributes strongly to the model, while the *Eccentricity* and *Extent* contribute very little.

We can also measure the total variance explained by all predictors combined using $R^2 = 0.6953849$, and we can check the standardized coefficients to better compare predictors' importance. The results follow.

Once again, we can see that Extent and Eccentricity contribute very little to the overall model.

1.2 Principal Components

A good way to aggregate and simplify data is by decomposing it into its principal components (centered in zero and scaled to have unit variance). Results follow.

We can see that more than ninety nine percent of overall variance can be explained with just three components, which means that not only we could make good prediction with lower degree data, but also that a 2D visualization of data that uses only two components should give us a good idea of the whole dataset.

2 PYTHON

We are now going to discuss the technologies employed in the development of the project, as well as presenting our implementation of the Raft's algorithm.

Python is a high-level, dynamically typed and interpreted programming language that is often used for scripting, data analysis and small application development, making it a non-obvious choice for this project, which does not fall into any of these categories.

As a language, it has two main advantages compared to others: first of all it is undoubtedly the most popular and widely used in the world (figure **??**) [**? ?**], meaning abundant documentation and resources. Secondly it has a huge ecosystem of libraries that implement all the functionalities we need for this project, namely: *XML-RPC* for the remote procedure calls (RPCs), *threading* to handle local concurrency and *Pygame* to manage everything game-related.

2.1 Remote Procedure Calls

In Raft's specifications it is stated that nodes communicate with each other via remote procedure calls [**?**], which in distributed computing is when a program causes a procedure (or subroutine) to execute in another address space (commonly on another computer on a shared network) calling it as if it were local (that is, the programmer writes the same code whether the subroutine is local or remote).

There are many libraries that implement this functionality, like gRPC (https://grpc.io/), which is a high performance open source RPC framework used by many big players, such as Netflix [1] and Cockroach Labs [2], available for many languages (Python included), but we opted for the standard library *XML-RPC* [3] thanks to its promised simplicity and ease of use.

The library provides both server and client implementations, encapsulating the former in its own loop, while the latter can be fired as needed allowing a bit more flexibility in its usage.

In code **??**, *client* is an instance of *ServerProxy*, which acts as the client-side interface for XML-RPC, allowing it to call the remote procedure *test_foo* as if it were a local function, even though it executes on a server in a different networked location.

Listing 2. Client as server proxy

```
with xmlrpc.client.ServerProxy('http://localhost:8000', allow_none=True) as client:
    print(client.test_foo(42)) # print returned value
```

The server must be instantiated and kept running by calling its event loop (e.g., using *serve_forever*), and all remote procedure calls must be registered using the *register_function* method of *SimpleXMLRPCServer* (code **??**).

Listing 3. Server

```
with SimpleXMLRPCServer (('localhost', 8000)) as server:
    def test_foo(number):
        return f'The number is {number}'

    server.register_function(test_foo)
    server.serve_forever() # keep server alive
```

[1] Netflix Ribbon is an Inter Process Communication library built in software load balancers: https://github.com/Netflix/ribbon

[2] Cockroach Labs is the company behind CockroachDB, a highly resilient distributed database: https://www.cockroachlabs.com/

[3] XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport: https://docs.python.org/3/library/xmlrpc.html

images/TIOBEindex.png

Fig. 2. TIOBE Programming Community Index, focus on Python statistics, 2025. (https://www.tiobe.com/tiobe-index/)

For this project, we extended *SimpleXMLRPCServer* to create a class that implements the Raft protocol (more details in section **??**).

2.2 Concurrency

In this project the need for concurrent programming arises from two challenges: every server has an internal timer that fires at certain intervals, and every node has to run a game engine and the server itself at the same time, both of which are, by design of their own respective libraries, independent, blocking and perpetually executing loops.

Some Raft implementations we examined achieve concurrency through asynchronous programming[4], using libraries such as *asyncio* [5], thereby avoiding the need to manage common multithreading challenges like ensuring thread-safety by preventing race conditions or data corruption and other hazards such as lock-free reordering or incorrect granularity (see footnote for more[6]). That being said, while powerful and efficient, writing asynchronous code can be awkward and cumbersome, so we opted for a more traditional approach using multithreaded programming: in computer science, a thread of execution is the smallest sequence of programmed instructions that can be scheduled independently [**?**], and multiple threads may be executed concurrently sharing resources such as memory. This is directly counterpointed to multiprocessing, where each process has its own storage space, and moreover processes are typically made of threads. Processes and threads are profoundly different and do not serve the same purpose, but it is useful to cite both of them to provide the context needed to fully understand section **??**.

In Python there are modules in the standard library for both of them, respectively *threading* [7] and *multiprocessing* [8]. It is fundamental to note that the former does not provide real multi-threading since, due to the Global Interpreter Lock of CPython (the, for want of a better word, official Python implementation), only one thread can execute bytecode at once. To cite directly from the documentation: *"[GIL is] The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines."* [9].

Thankfully, this does not apply with the *multiprocessing* module, which creates separate processes instead, offering both local and remote concurrency effectively side-stepping the Global Interpreter Lock, allowing programmers to fully leverage multiple cores. As previously stated, processes are much heavier than threads and thus more expensive to create, but do not incur the risks of shared memory.

2.2.1 Comparison. To evaluate which of the two modules is more suited for our purposes, we devised a simple experiment: we created two game instances with one hundred and one thousands coloured dots respectively (figure **??**), that move around by offsetting their position each frame of a random amount between minus five and plus five pixels (pseudocode **??**).

Then we ran both of them in three scenarios: with the game instance alone (baselines), with a server alive in a thread and with a server alive in a process, and we measured the *frames per second* (FPS) [10] in each case, since it is the most common metric to evaluate game performance. Higher FPS-count translates to a smoother and more responsive, i.e., better, gaming experience.

Listing 4. Pygame graphical dot offset

[4]Two examples of Raft's implementations that leverage asynchronous programming are Raftos (https://github.com/zhebrak/raftos/tree/master) and Zatt (https://github.com/simonacca/zatt/tree/master)

[5]Asyncio is a library to write concurrent code using the async/await syntax: https://docs.python.org/3/library/asyncio.html

[6]Multithreading Hazards, Microsoft: https://learn.microsoft.com/en-us/archive/msdn-magazine/2008/october/concurrency-hazards-solving-problems-in-your-multithreaded-code

[7]The threading module provides a way to run multiple threads (smaller units of a process) concurrently within a single process: https://docs.python.org/3/library/threading.html

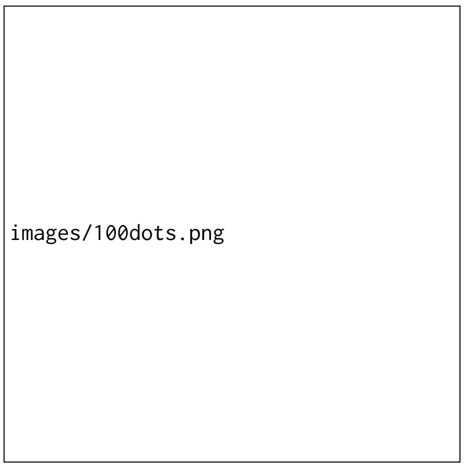
[8]The multiprocessing module is a package that supports spawning processes using an API similar to the threading module: https://docs.python.org/3/library/multiprocessing.html

[9]Global Interpreter Lock: https://docs.python.org/3/glossary.html#term-global-interpreter-lock

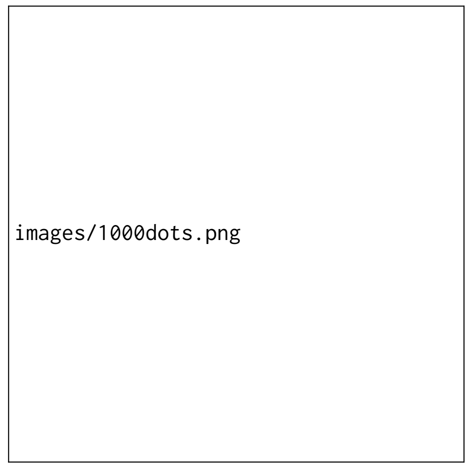
[10]Frame rate, most commonly expressed in frames per second or FPS, is typically the frequency (rate) at which consecutive images (frames) are captured or displayed. This definition applies to film and video cameras, computer animation, and motion capture systems, while in the context of computer graphics is the rate at which a system, particularly the graphic card, is able to generate frames. Source: https://en.wikipedia.org/wiki/Frame_rate

```python
# create random offsets for both x and y coordinates
xmov = random.randint(-5,5)
ymov = random.randint(-5,5)

# move the dot by a certain offset
dot.move_by(xmov, ymov)
```



(a) 100 dots game instance



(b) 1000 dots game instance

Fig. 3. Two game instances made with Pygame, with respectively 100 and 1000 dots that randomly move around

Results, shown in the graph at figure **??**, tell us us that:

- Increasing the number of dots from 100 to 1000 halves the FPS count;
- Adding a server in a separate thread halves performance;
- Using *multiprocessing* yields worse performance than *threading* in the 100-dots scenario (about -30%), while performing similarly in the 1000-dots one.

This leads us to conclude that, for our specific purposes, the *threading* module is the best choice, especially since the final game will be way less computationally expensive from a graphical standpoint, hence using a lighter weight alternative should be even more beneficial than tested.

All tests have been performed with the following machine:

- OS: Ubuntu 24.04.1 LTS x86_64;
- Kernel: 6.8.0-52-generic;
- Shell: bash 5.2.21;
- CPU: 13th Gen Intel i7-13620H;
- GPU: NVIDIA GeForce RTX 4050 Laptop GPU;
- Memory: 15610MiB;
- Python version: 3.12.3;
- Power Mode: Balanced;
- Power Supply: 100W via type C.

images/performance_eval_fps_graph.png

Fig. 4. Performance evaluation graph: red hues for baselines, blue hues for threading and green hues for multiprocessing. Darker shades for 1000 dots and lighter shades for 100 dots game instances

2.3 Game Engines

There are many ways to implement a graphical user interface: from clever shell tricks like htop [11], to full-fledged game engines like Unity [12] or Godot [13] that often come with their own editor and a *top-down* approach, meaning build the UI first and then go down to code as needed for scripting and refining.

Unfortunately, our needs are quite opposite: what we want is a code-only, mono-language framework that while slowing down game development should simplify merging Raft with it. The choice thus boiled down to two alternatives: tkinter and Pygame.

2.3.1 Comparison. Let's list strengths and weaknesses of the two.

- **tkinter:**
  - 👍 Module of the standard library;
  - 👍 Few lines of code to make simple UIs;
  - 👎 Low flexibility;
  - 👎 No game loop;
  - 👎 Not a game engine;
- **Pygame:**
  - 👍 Extreme flexibility;
  - 👍 Direct access to game loop;
  - 👍 APIs to access many kinds of user inputs;
  - 👎 Verbose to obtain simple UIs;
  - 👎 Non-standard community-made framework.

We ultimately decided to opt for Pygame for four reasons: it is extremely flexible, exposes many useful functions (for example to catch different user inputs), gives direct access to the game loop and it is a novel and fun framework that has never, to the best of our knowledge, been used in such a fashion.

3 RAFT

It is not our intention to plagiarize Ongaro and Ousterhout's excellent work *"In Search of an Understandable Consensus Algorithm"* [?] by presenting the Raft algorithm's specifications. Instead, we are going to discuss how we molded it to our own use case.

The algorithm divides its nodes into three roles, namely *leader*, *follower* and *candidate*, and revolves around three core functionalities: leader election, log replication and cluster membership change. Log compaction is also mentioned, while a byzantine fault tolerant variant is never explored by the original authors. To grant consistency, Raft's design choice is to centralize all decisions on one node, the above mentioned leader, that synchronizes all cluster's nodes.

One last component, instrumental to the functioning of the algorithm, is the *term*: everything happens in a certain term, which divides time logically and increments every election. This is necessary to recognize out-of-date leaders: if some follower has a term greater than the leader's, said leader is outdated.

Our Raft class directly extends *simpleXMLRPCServer* from XML-RPC module, as shown at code **??**.

[11] Htop is a cross-platform text-mode interactive process viewer: https://htop.dev/
[12] Unity is a cross-platform game engine developed by Unity Technologies: https://unity.com/
[13] Godot is a cross-platform open-source game engine: https://godotengine.org/

Lastly, to fire off non-blocking concurrent RPCs on the cluster, we leverage the *concurrent.futures* module using *ThreadPoolExecutor*. To avoid creating and destroying pools every time a server needs to communicate with the cluster, we embedded a finite amount of workers as class attributes (code **??**).

Listing 5. Class Raft definition

```
class Raft(SimpleXMLRPCServer):
    def __init__(self,
                 addr: tuple[str, int],
                 allow_none: bool = True,
                 # ...
                 last_index_on_server: list[tuple[int, int]] | None = None
                 ):
        SimpleXMLRPCServer.__init__(self, addr=addr, allow_none=allow_none)
```

Listing 6. ThreadPoolExecutor created with as many workers as there are servers in the cluster

```
class Raft(SimpleXMLRPCServer):
    def __init__(self,
                 # ...
                 )
        # start executors pool
        self.executor = concurrent.futures.ThreadPoolExecutor(max_workers=len(self.cluster))
```

3.1 Node Types

As previously stated, there are three node types: leader, follower and candidate (code **??**). In this section we are going to show their characteristics and similarities. Note that all nodes have a timer: it is randomized for each of them and has been implemented by extending *threading.Timer*, thus making it thread-safe (code **??**)

Listing 7. Node modes

```
class Raft(SimpleXMLRPCServer):
    class Mode(Enum):
        LEADER = 1
        CANDIDATE = 2
        FOLLOWER = 3

    def __init__(self,
                 # ...
                 mode: Mode = Mode.FOLLOWER,
                 )
```

Listing 8. Threadsafe looping timer

```
class LoopTimer(Timer):
    def __init__(self, interval, function, args=None, kwawrgs=None):
        Timer.__init__(self, interval, function, args, kwawrgs)
        self.was_reset : bool = False
    # ...

class Raft(SimpleXMLRPCServer):
    def __init__(self,
                 # ...
```

```
            )
        # start timer
        self.timer = LoopTimer(timeout, self.on_timeout)
        self.timer.start()
```

3.1.1 *Leader Node.*

The algorithm revolves around, and requires the existence of, one and only one leader node, whose job is to synchronize all servers' logs to ensure data consistency. It does so by replicating its own log on all followers (the non-leader nodes) by sending new or, if needed, old entries via remote procedure calls.

To make sure all nodes believe the leader's alive, periodically sends an empty remote procedure call called *heartbeat* (every 150-300ms).

3.1.2 *Follower Node.*

All nodes, except for the leader, perform as followers. They are not allowed to replicate their own log, and they have to forward any request to the leader.

To make sure the cluster never remains without a leader, every follower has an election timeout (between 150ms and 300ms) which resets every time an RPC from the leader is received. If it times out, the follower changes its state to *candidate*, increments its current term and starts a leader election.

Followers become candidates in another scenario: whenever they receive an entry from the leader, they compare it with their own last log entry. If the leader's term is smaller, it is out of date and a new election is started.

3.1.3 *Candidate Node.*

When a follower's election timeout times out, it becomes a candidate, increments its own term and starts an election. Votes for itself and then waits for one of two outcomes: wins, thus becoming a new leader, or loses (either another leader gets elected or the old one manifests itself) thus reverting back to being a follower.

3.2 Log

As stated, the leader's job is to accept requests (in our specific case they are player inputs) and then forward them to the followers. Let's talk about the structure of the log.

The log is basically a *list* (or an *array*) of entries, where *entry* is an element that encapsulates data (like an integer or a string), has an index (unique for each entry) and the term of its creation (figure **??**). We defined entries as *Data Classes* [14] (decorators that simulate C's structures) as seen in code **??**.

Listing 9. Dataclass Entry definition

```python
@dataclass
class Entry:
    term: int
    index: int
    command: str
```

3.3 Log Replication and Overwriting

All log propagation revolves around one remote procedure call named *append_entries_rpc*, which the leader calls on a list of server proxies that connect it to the followers. On their end, each follower calls the RPC on a proxy of the leader. It must be registered in the server to be callable, as seen in listing **??**.

[14] Data Classes module provides a decorator and functions for automatically adding generated special methods to user-defined classes: https://docs.python.org/3/library/dataclasses.html

images/logStructure.png

Fig. 5. Raft's log is fundamentally an array made of entries

Listing 10. Register, thus making it callable, the remote procedure call *append_entries_rpc*

```
def handle_server():                                    # enclose server in a callable function
    with Raft(...) as server:                           # creates SimpleXMLRPCSever
        def append_entries_rpc(entries, term, commit_index, all_log):
            # ...
        server.register_function(append_entries_rpc)    # makes function callable on the other side
        server.serve_forever()                          # keeps server alive
```

3.3.1 Leader Propagates Entries. The leader (each node as a matter of fact) periodically checks whether there are new commands to propagate (always stored in queue *pygame_commands*, more details in section **??**), by overriding *SimpleXMLRPCServer*'s method *service_actions* (listing **??**).

Then, it translates them into entries by giving each of them the current term and a log index that starts from $lastLogEntry(index) + 1$ and increases by one for each entry. To clarify: if $lastLogEntry(index) = 7$ and we have three new commands, their indices will respectively be eight (8), nine (9) and ten (10). The translation can be seen at listing **??**.

At this point, it propagates *new_entries* to the whole cluster, updating the commit index (necessary for applying log to state) as soon as propagations are successful on at least half of the cluster, like so: $commitIndex = lastNewEntry(index)$.

What happens if the *append entries* gets rejected? The leader adds to *new_entries* its own last log entry: $new\_entries = lastLogEntry + new\_entries$ (figure **??**). Then, it repeats the propagation procedure, for each reject a new *last log entry* gets added, progressively traversing the log backwards. If, at a certain point, $new\_entries == allLog + new\_entries$ (i.e., all leader's log gets propagated) the flag *all_log* is set to *True*.

Since every server may reject or accept different sets of entries, depending on their own local log, every propagation must be "local" for each follower.

The flow of execution for the log propagation is: *Raft: service_actions* ➔ *Raft: propagate_entries* ➔ *propagate_entries :encapsulates_proxy: append_entries_rpc*. The last one gets called as many times as needed on every single follower.

Of course, all propagation happens concurrently using a *ThreadPoolExecutor*, and the code for entries propagation (leader's side) can be seen at listing **??**.

images/newEntries.png

Fig. 6. New entries with the last log's entry pushed to the top of the list

Listing 11. Periodically checks whether there are new commands

```
def service_actions(self):                          # RUN method of the server, override
    if time.time() - self.countdown >= .005:        # do actions every .005 seconds
        global pygame_commands

        if not pygame_commands.empty():             # propagate entries to cluster
            self.propagate_entries()
```

Listing 12. Translates commands into new entries

```
def propagate_entries():
# ...
    while not pygame_commands.empty():
        command = pygame_commands.get()
        log_index += 1                              # lastLogEntry(index) + 1
        self.new_entries.append(Raft.Entry(         # append to support list 'new_entries'
            term=self.term,
            index=log_index,
            command=command
        ))
```

Listing 13. Leader propagation procedure, for complete code refer to project's repository

```
def propagate_entries(self):
    # ...
    if self.log:                                # travel backwards through self.log for each reject
        entries: list[Raft.Entry] = []
        entries.append(self.log[-1])
        entries.extend(self.new_entries)
        log_iterator: int = -2                  # log_iterator soft resets for each follower
    else:
        entries: list[Raft.Entry] = self.new_entries
        log_iterator: int = -1

    # ...
    # inner function necessary for concurrent execution
    def encapsulate_proxy(self, follower, entries, log_iterator):
        # ...
        with xmlrpc.client.ServerProxy(complete_url, allow_none=True) as proxy:
            while not propagation_successful:
                # send new entries (local for each follower)
                # ...
                result = proxy.append_entries_rpc(entries, self.term, self.commit_index, all_log)
                if result[0] == False:
                    # add another entry from self.log to new entries
                    entries = [self.log[log_iterator]] + entries
                    log_iterator -= 1
                elif result[0] == True:
                    propagation_successful = True

        return propagation_successful # to propagate_entries, make propagation counter increase

    results = []

    # fires RPCs concurrently using ThreadPoolExecutor
```

```python
    future_result = {            # clever python syntax trick
        self.executor.submit(
            encapsulate_proxy,  # function
            self,               # function's parameter
            follower,           # function's parameter
            entries,            # function's parameter
            log_iterator        # function's parameter
            ): follower for follower in self.cluster}
    for future in concurrent.futures.as_completed(future_result):
        # results of RPCs
        data = future.result()
        results.append(data)

    # finally counts if propagation was successful enough
    if results.count(True) >= len(self.cluster) / 2:
        self.log.extend(self.new_entries)               # add new entries to log
        self.new_entries.clear()                        # clear new entries list
        self.commit_index = self.log[-1].index          # ensure log gets eventually applied
    else:
    #   new entries are not cleared, so they will be propagated again
```

3.3.2 Follower Receives Entries. When a follower receives an *append entries* request from the leader, first checks whether leader is up to date. If it's not, i.e., $leaderTerm < followerTerm$, rejects by answering with the tuple $(False, followerTerm)$. In this context, *answering* is done via the remote procedure call's return value.

On the other hand, if the leader's term is equal or greater than its own (i.e., $leaderTerm \geq followerTerm$), the follower updates its commit index and, if $leaderEntries \neq \emptyset$, checks the *all_log* flag. If it's *True*, it clears all its own log to overwrite it with the leader's (fundamental to log forcing, listing **??**). Otherwise ($all\_log \neq True$), the leader did not send all its log, so the follower searches through its own log for an entry equal to the leader's previous one (i.e., the entry preceding the new ones). Let's make an example:

- Leader's log = $[1, 2, 3, 4, 5]$;
- Leader's new entries = $[6, 7]$;
- Thus leader's prev = $[5]$.

If it finds an entry equal to leader's previous (i.e., $followerLog(someEntry) == leaderPrev$), deletes all log entries that follow it and appends the new ones, otherwise ($\nexists(followerLog(someEntry) == leaderPrev)$) rejects the request. Since the leader, when faced with a reject, adds a new *prev* and keeps repeating the send until it comprises all its log, at a certain point the follower will be forced to overwrite all its log, thus making it equal to the leader's. This overwriting is called *log forcing* and ensures that all logs are equal to the leader's.

The code can be seen at listing **??** (for the complete one refer to the repository).

Listing 14. Follower clears its own log to overwrite it with the leader's

```python
if all_log == True:
    server.log.clear() # if leader sent all its log, clear and rewrite log (leader's log forcing)
```

Listing 15. Follower search in its own log for an entry equal to leader's prev

```python
if commit_index is not None:
        server.commit_index = commit_index  # update commit index

```

```python
if entries is not None:      # not an heartbeat
    if all_log == True:      # complete overwrite
        server.log.clear()

    if server.log:  # if follower's log not empty search for an entry equal to leader's prev
        entry_log_index: int | None = None              # save its log index (!= entry index)
        for i, my_entry in enumerate(server.log):
            if (my_entry.index == entries[0].index
                and my_entry.term == entries[0].term):
                entry_log_index = i
                break # no need to search further
        # here entry_log_index == (position of entry equal to leader.prev) | None

        if entry_log_index is None:         # entry equal to leader's prev not found
            return(False, server.term)      # rejects

        del server.log[(entry_log_index ):] # delete all log following leader prev

    server.log.extend(entries) # append new entries
```

3.3.3 Follower Sends Entries. Since every server is a Raftian node with a game instance and thus player inputs, followers have their own Pygame commands to propagate. Just like the leader, in their *service_actions* function they periodically check whether there are new commands to propagate and call *propagate_entries* accordingly. Then, they translate all Pygame commands into entries (same code as listing **??**) and propagate them to the leader via *append_entries_rpc*. Nothing else.

As previously stated, followers are *passive*, meaning they do not apply their own player inputs when they register them, but only after the leader propagates them back to the whole cluster.

3.3.4 Leader Receives Entries. The leader does very little when receives entries from the followers: it just puts them into its own *pygame_commands* queue. They will get processed and propagated eventually, as stated in section **??**.

3.4 Apply Log to State

Let's first explain two key attributes: *commit index* and *last applied*. Both of these represent an index, but the former is the highest-index entry successfully propagated in the cluster, while the latter is the highest-index entry already applied to state.

Every node, whether leader or follower, applies entries to state in the same way: inside their function *service_actions* they periodically check if there is a discrepancy between *commit index* and *last applied* attributes (i.e., $commit\_index > last\_applied$). Then, starting from the last applied entry, they apply to state all successive entries up to and including the one with the same index as *commit_index*, updating *last_applied* as they go. To clarify: servers apply all entries between $log(entry.index == last\_applied)$ and $log(entry.index == commit\_index)$ as shown in figure **??**.

To apply entries in our context means that they get appended to the queue *raft_orders*. The code can be seen at listing **??** (for the complete source refer to the repository)

Listing 16. All nodes apply entries to state based on *commit_index*

```python
def service_actions(self):
        # ...
        if self.commit_index is not None and self.commit_index > self.last_applied:
```

images/applyToState.png

Fig. 7. All entries between *last_applied* and *commit_index* (included) get applied to state

```
    global raft_orders  # applying means appending entries to this queue

    #...

    last_applied_log_position: int = -1
    for i, my_entry in enumerate(self.log):
        if (my_entry.index == self.last_applied):
            last_applied_log_position = i
            break # found log position of last applied entry

    log_iterator = last_applied_log_position + 1    # improves code clarity

    while self.last_applied != self.commit_index:
        raft_orders.put(self.log[log_iterator])
        self.last_applied = self.log[log_iterator].index
        log_iterator = log_iterator + 1
    # here self.last_applied == self.commit_index
```

3.5 Log Compaction

This functionality, as well as the following ones, have not been implemented due to time constraints. We decided to include them anyway since they were still the product of careful consideration and could prove useful in future implementations.

Log compaction, also called *snapshot*, is a way to clear servers' logs and save them in persistent memory, necessary in long-running or message-heavy applications. Every node autonomously decides when to do it.

The idea is as follows: inside their function *service_actions*, servers check whether their log is larger than a certain size (to be determined) and then call a *snapshot* method accordingly, which saves all entries in a JSON file progressively deleting them from the log, from the first one up to (but excluding) the *last applied*.

An alternative interpretation, closer to Ongaro and Ousterhout's idea, is saving the state of the cluster. When applied to our context, we can imagine a JSON file that describes all servers' state, by saving for each of them *id*, *url*, *port*, and *hp* values. *Index* and *term* of the last snapshotted entry should also be saved, as well as the current configuration (which will be mentioned in section **??**). An example of the JSON structure can be seen at listing **??**.

This method requires more pre-processing to generate the JSON, and more post-processing to later check log's correctness, but is also more compact, which helps when the leader calls *install snapshot*, a remote procedure call that can sometimes be needed to bring up to speed new or outdated servers by sending them the leader's snapshot.

It goes without saying, but whichever method is ultimately chosen, every new snapshot must comprise all information contained in the old ones.

Listing 17. The JSON for a snapshot that saves cluster's state would look something like this

```
{
    servers:[
        {
            id : 1,
            url : "localhost",
            port : 8000,
            hp : 70
        },
        {}, {}
    ],
    lastIndex : 11,
    lastTerm : 3,
    lastConfig : 8
}
```

3.6 Leader Election

As aforementioned, the leader is fundamental to grant consistency. To make the protocol fault tolerant, it can dynamically change over time via a distributed election: whenever a follower finds out that the leader is either outdated or missing (i.e., internal follower's timer times out before receiving any call from it), said follower starts an election. It changes its internal state to *candidate*, increases its own term by one, votes for itself, and then propagates to the whole cluster a specialized remote procedure call named *request_vote_rpc* (code, removed in the final version, at listing **??**). Votes are given on a first-come-first-served basis, and to prevent split votes each server's election timeout is randomized between 150ms and 300ms at the start of every election. This ensures that in most cases only one server will be candidate at a time.

At this point there are two possible outcomes: more than half of the cluster votes for the candidate (which we will call *"A"*), that therefore becomes leader and propagates a heartbeat to the whole cluster, or another candidate (which we will call *"B"*) is more up-to-date (i.e., *B's term* is greater than *A*'s or equal but with a greater *lastIndex*). In this last case, candidate *A* reverts back to follower and votes for *B*. The pseudocode for all the above can be seen at listing **??**.

One last eventuality is that the old leader manifests itself. In this case, if the old one is equally or more up-to-date than the new one (both term and last index count), the latter reverts back to follower and the preceding monarch gets reinstantiated.

Listing 18. Pseudocode for *request_vote_rpc*

```
def request_vote_rpc(...):
    # if candidate less up to date -> reject
    if self.term > candidate_term:
        return (self.term, False)

    # if a candidate already exists
    if self.voted_for is not None and not candidate_id:
        return (self.term, False)

    # vote for candidate
    self.voted_for = candidate_id
    return (self.term, True)
#...
server.register_function(request_vote_rpc)
```

Listing 19. Pseudocode for *to_candidate*, gets fired on *election timer* timeout

```
def to_candidate(self):
    self.mode = Raft.Mode.CANDIDATE
    self.term += 1
    self.voted_for = self.id

    self.timer.reset() # reset election timer

    for server in self.cluster:
        server.request_vote_rpc(...)
        count votes

    if some_return.more_up_to_date:
        self.mode = Raft.Mode.FOLLOWER
        self.voted_for = some_return.id

    if votes > len(self.cluster) / 2:
        self.to_leader()    # handles mode change and heartbeat
```

3.7 What is Missing

There is one last functionality discussed by Ongaro and Ousterhout, which is gracefully managing changes in the cluster's members by leveraging a configuration attribute and keeping multiple configurations alive simultaneously for a certain period of time (figure **??**). We never intended to include it due to time constraints, therefore there is nothing we can add beyond the original work.

Another concern, which was not considered in the Raft paper, is faults caused by bad actors that purposely send malicious information. This is a real problem for our use case, since players want to win and are therefore incentivized to act maliciously by cheating (a practice so widespread that has created its own multimillion-dollar market [**?**]).

The implementation of Byzantine fault tolerance was beyond the scope of this work. Therefore, we refer the reader to some example works for further details: *"A Raft Algorithm with Byzantine Fault-Tolerant Performance"* by Xir and Liu [**?**], and *"VSSB-Raft: A Secure and Efficient Zero Trust Consensus Algorithm for Blockchain"* by Tian et al. [**?**].

`images/raftMembershipChange.png`

Fig. 8. Cluster goes through a hybrid configuration to pass from the old to the new one. Source: Raft paper [**?**]

4 RAFTIAN NODE ARCHITECTURE

In the previous section (**??**), we explained in detail how nodes communicate with each other and handle their log. Now we will explain what actually happens inside a node, i.e., the architecture of a single node of the application, comprising of both a server and a game instance. Before showcasing it, we will briefly explain how Pygame works, thus taking the opportunity to present the user interface.

4.1 Pygame

Pygame's approach is very straightforward: first comes the declaration and set up of all the graphical components, such as game window, fonts, colors, variables and constants. Every element gets positioned on the main window by coordinates *(x,y)*, where *(0,0)* is the top left corner. Most items are made of two fundamental Pygame classes: *Rect*, which creates non-graphical objects that expose many useful methods, for example to position, move, and resize themselves, or to detect collisions and mouse clicks, and *Surface*, which is the most basic graphical component that has dimensions and can be drawn upon. Often we want to bind, thus constrain, surfaces with rects so that we use the latter for spatial operations. One last fundamental is the *blit* function, a method that draws one image onto another or, to be precise, that draws a source Surface onto the object Surface that calls it. We can give it an optional argument to specify a drawing destination, either with coordinates or a rect. To clarify: $baseSurface.blit(sourceSurface, destination)$ draws *sourceSurface* onto *baseSurface* at the coordinates specified by *destination*. An example of all the above can be seen at listing **??**.

Pygame, being low-level in nature, is very flexible and allows us to do pretty much whatever we want. For example, we defined our players as dataclasses that encapsulate both the players' data (like *id* or *health points*) and their Rect and Surface objects, as in listing **??**.

Finally, Pygame gives us direct access to the game loop, which is implemented as nothing more than a *while loop*. In it, we can process player inputs and refresh the screen, dynamically changing what is displayed. In short, we manage everything that happens while the game is running. In listing **??** we can see two types of player input, one for quitting the game and a left-mouse click, the latter of which causes a refresh of the header. Specifically, if *Player 2* gets clicked, the header will display *"Player 2 pressed"*, reverting back to its original state after a couple of seconds. The last command, *clock.tick(fps)*, allows us to limit the framerate, effectively slowing down or speeding up the game engine itself by constraining the amount of times per second the game loop repeats itself.

Listing 20. Pygame base components

```
pygame.init()                                   # starts pygame
GREY = (125, 125, 125)                          # define a color
DISPLAY = pygame.display.set_mode((1000, 1200)) # creates game window 1000x1200 pixels in resolution
clock = pygame.time.Clock()                     # necessary to mange fps
font = pygame.font.Font(None, 60)               # creates default font
toptext = font.render("Top Text", False, BLACK) # header text
rect_header = pygame.Rect(0, 0, 1000, 100)      # creates rect for header
header = pygame.Surface((1000, 100))            # creates surface for header
header.fill(WHITE)                              # draw on surface

DISPLAY.blit(header, rect_header)               # draw on DISPLAY the header surface
                                                # position is given by rect_header
#...
# draw text on coordinates
DISPLAY.blit(toptext, (rect_header.centerx - xoffset, rect_header.centery - yoffset))
```

Listing 21. Players defined as dataclasses that encapsulate Pygame elements

```
@dataclass
class Player:
    id: int
    hp: int
    rc: pygame.Rect     # represents player position and size
    ui: pygame.Surface  # exposes UI of the player e.g., colour

player1 = Player(
    id=1,
    hp=100,
    rc=pygame.Rect(585, 685, 80, 80),  # x0, y0, width, height
    ui=pygame.Surface((80,80))
)
```

```
player1.ui.fill(RED)                    # colour player red
DISPLAY.blit(player1.ui, player1.rc)    # draw on display via rect
```

Listing 22. All interactions and frame-by-frame rendering happen in the game loop

```
while True: # game loop
    for event in pygame.event.get():    # process player inputs
        if event.type == pygame.QUIT:
            pygame.quit()

        if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:  # left mouse button click

            pos = pygame.mouse.get_pos() # gets mouse position

            if player.rc.collidepoint(pos): # rect allows us to detect collisions
                toptext = font.render(f"Player {player.id} pressed", False, BLACK)

                DISPLAY.blit(header, rect_header)   # erase previous text
                DISPLAY.blit(toptext, (rect_header.centerx - xoffset, rect_header.centery - yoffset))

    pygame.display.flip()  # refresh on-screen display
    clock.tick(60)         # limits framerate
```

4.2 Raftian User Interface

Figure **??** shows different phases of a normal Raftian's game session. Specifically, they demonstrate how the interface changes when the player repeatedly clicks on (thus *attack*) Player 3, the blue one on the top left (players are represented as four coloured squares on the board). Players' colours become progressively desaturated as their health points decrease, by modifying their alpha channels as shown in listing **??**. When a player is dead, its colour changes to a darker shade.

In the header an attack message gets written, reverting back after half a second. Said message changes depending whether the player is still alive; if not, damage is ignored.

Listing 23. Whenever a player gets damaged, its colour gets desaturated

```
for player in players:
    if player.id == order.command and player.hp > 0:
        player.hp -= 30  # apply damage to player:

        if player.hp < 90 and player.hp >= 60:
            player.ui.set_alpha(190)
            DISPLAY.blit(player_UI_cleaner, player.rc)  # clean player UI
            DISPLAY.blit(player.ui, player.rc)          # redraw player UI
```

4.3 Raftian Node Architecture

The architecture of a Raftian node can be seen at figure **??**. Let's explain it: first of all, the game loop and the server are encapsulated in two functions to be handed over to two different threads, enabling concurrent execution (listing **??**). Whenever a player clicks on (i.e., attacks) one of the four players, the game engine does not apply damage immediately. Instead, it generates a *command* which represent, if we want, the *intention* of attacking said player. This *command* is images/raftian1.png

(a) Game starts

images/raftian2.png

(b) Player 3 is attacked

images/raftian3.png

(c) Header reverts to default

images/raftian4.png

(d) Player 3 is left with forty HP

images/raftian5.png

(e) Player 3 is left with ten HP

images/raftian7.png

(f) Player 3 cannot be attacked again

Fig. 9. Different phases of a normal Raftian's game session. Player 3 keeps getting damaged until it dies

thus appended to a queue called *pygame_commands*, one of the two synchronized FIFO queues[15] necessary to allow communication between server and Pygame's threads (listing **??**). Both are instances of Python's standard library queue module[16], which implements thread-safe, multi-producer, multi-consumer queues.

At this point, Pygame does not concern itself anymore with said user input. The server, by itself, periodically checks the *pygame_commands* queue (as in listing **??**) and, when not empty, removes elements from it (as in listing **??**) and propagates them as entries to the leader (or to the whole cluster if said server *is* the leader, as in listing **??**).

Then, the leader propagates the received commands to the whole cluster, which we will now call *orders*. Each server adds received orders to its own log, as explained in section **??**, so that they can later be appended to the *raft_orders* queue when entries get applied to state (as in section **??**). This way, the original user input gets propagated back to the server that generated it in the first place.

Finally, Pygame checks (periodically) the *raft_orders* queue for orders. When it finds them, it removes them from the queue and updates the user interface accordingly (an example can be seen at listing **??**).

[15] FIFO, or First-In-First-Out, is a method for organizing the manipulation of a data structure, often data buffers, where the oldest data inserted is the first that gets processed, making it work in a sense like a pipeline

[16] Python's queue, a synchronized queue class: https://docs.python.org/3/library/queue.html

The whole idea is to keep server and game engine as separated as possible: the former reads commands, propagates them and writes received orders, the latter reads orders, updates the UI, and writes commands, following a unidirectional cyclic communication pattern.

Listing 24. Start both Pygame and server's threads

```
def handle_pygame():
    pygame.init()
    #...
    While True:
         #...
def handle_server():
    with Raft(...) as server:
        #...
        server.serve_forever()

server_thread = threading.Thread(target=handle_server)
server_thread.start()
pygame_thread = threading.Thread(target=handle_pygame)
pygame_thread.start()
```

Listing 25. Queues for commands and orders, they allow inter-thread communication

```
# user inputs through Pygame which writes them here
# Raft reads them and propagates them to the cluster
pygame_commands = Queue()

# commands that have been applied to state are written here by Raft
# Pygame reads them and updates UI accordingly
raft_orders = Queue()
```

Listing 26. Pygame periodically checks whether there are new orders and updates the UI accordingly

```
while True: # Pygame's main loop
    #...
    while not raft_orders.empty():
        order: Raft.Entry = raft_orders.get()

        for player in players:
            if player.id == order.command and player.hp > 0:
                player.hp -= 30  # apply damage to player:
                #...
```

4.4 Evaluation and Results

While we did not have enough time to implement evaluation procedures for measuring things like response time or network latency, the game has been thoroughly tested. Responsiveness is good, with no noticeable input latency and a solid framerate way above sixty frames per second (which is still the preferred limit), and both log replication and overwriting functionalities have been confirmed working as intended. The proof of this fact can be observed in the logs at https://github.com/mhetacc/RuntimesConcurrencyDistribution/tree/main/logs, specifically by comparing logs among folders *bob1*, *bob2*, *bob3*, *bob4*, and *raftian*: all logs (meaning *Raft logs*) contain the same entries, in the same order, between all players.

images/nodeArchitecture.png

Fig. 10. Raftian node architecture

5 REFLECTION

Let's now discuss problems, potential future expansions and learning outcomes of this project.

5.1 Self-Assessment

Using *XML-RPC* and *threading* libraries proved to be sub-optimal: the former has a very contrived syntax and makes writing procedure calls a bit unintuitive, since forces the programmer to think in the "opposite direction". When writing a remote procedure call (i.e., those functions that get registered by *register_function()*) is important to keep in mind that they are going to be used by the caller and not by sender (in whom code block they are written in).

Code could be less coupled: both server and game loop reside in the same file, and a lot of components are either internal classes or nested functions. Moreover, both command and order queues are global variables, which is generally a practice to be avoided.

On the other hand, code is well documented and as understandable as possible, even though following the flow of, for example, an input propagation requires jumping through it many times.

5.2 Future Works

Apart from the two features already discussed (leader election and log compaction), future expansions could implement cluster's membership change and a Byzantine fault-tolerant version of Raft. Adding new game functionalities, thus command types, should be easy since they can be propagated by the existing infrastructure, and the same is true for adding new players: provided that a new, bigger, user interface gets created, changing cluster's size should, in our testing, work without any issues.

5.3 Learning Outcomes

We started this project by having very limited Python competencies, having never written concurrent programming, never touched a game engine and never worked with network transmission protocols. All in all, we learned all of the above, in some cases going so far as trying different alternative solutions (we implemented Raft nodes mockups with threading, multiprocessing and asyncio libraries), making this project an invaluable learning experience.