Raftian: Pygame Meets Raft

MARCO BELLÒ and TULLIO VARDANEGA, University of Padua, Italy

The problem of shared consensus in a distributed system is both older than a millennium and more relevant than ever: while in the Aegean island of Paxos the challenge was to keep track of the many laws passed in a parliament where legislators had better to do than attend sessions full-time, nowadays the ubiquity of web-based architectures and applications (from a simple cloud storage to the whole banking system) gives daily headaches to developers and system administrators alike. To reduce ibuprofen consumption in the IT sector, Ongaro and Ousterhout devised an easily understandable and implementable alternative to the Paxos algorithm, namely the Raft consensus algorithm, which we employ in this work to implement communication via XML-RPC between multiple Pygame applications, each running a game instance that aims to simulate a simplified version of a real-time strategy game.

1 INTRODUCTION

From sharing spreadsheets between a handful of laptops in a small basement office, through large-scale rendering on a supercomputer, to the entire global finance system, distributed computing has become an essential component of the modern world that we almost take for granted: nowadays, what most people need a computer for can be done in the browser thanks to services like email clients, cloud calendars, media streaming platforms and web-based office suites (like Google Docs) that expose word editors, spreadsheets managers, presentations programs and more, all while being constantly synchronized to the cloud, which not only ensures data persistence and availability, but also enables sharing and collaboration between users.

It does not end here: other examples of distributed applications include cloud storage services like Dropbox, Google Drive or OneDrive, streaming services like Netflix, YouTube or Spotify, distributed computing like blockchain technologies or AWS, online banking services (the banking system itself is distributed since way before), social networks, and even maritime and aircraft traffic control systems. Moreover, the rise of the gaming industry played a significant role in pushing forward distribution: in 2024 the gaming market revenue was estimated to be 187.7 billion U.S. dollars [2], making it a hefty slice of the pie that is the entertainment industry [13], with 111 billions generated by free-to-play games [3] (70 billions from social and casual games alone [8]), which interests us since their business model often relies on cosmetics, game passes and advertisements, forcing them to be constantly on-line.

Let's now define what distribution *is*: a distributed system is a computer system whose inter-communicating components are located on different networked computers [1, 17], which coordinate their actions via message-passing to achieve a common goal. There are three significant challenges to overcome: maintaining components' concurrency [1], eliminating global-lock reliance and managing the independent failure of components, all while ensuring scalability (often the purpose is scaling itself) and transparency to the user, meaning interactions with any exposed interface must be done while being unaware of the complexity behind them.

Most importantly, shared consensus must be guaranteed: it does not require much thought to see that all servers in a cluster should agree on one or more shared values, lest becoming a collection of un-related components that have little to do with collaboration (thus distribution). In the most traditional single-value consensus protocols, such as Paxos [10], cooperating nodes agree on a single value (e.g., an integer), while multi-value alternatives like Raft [11] aim to agree on a series of values (i.e., a log) growing over time forming a sort-of cluster's history. It is worth noting that both goals are hindered by the intrinsically asynchronous nature of real-world communication, which make it impossible to achieve consensus via deterministic algorithms, as stated by Fischer, Lynch and Paterson in their FLP impossibility theorem [5]. Thankfully this can be circumvented by injecting some degree of randomness.

1.1 Project's Contributions

The concepts and examples we mentioned so far allow us to finally present the goal of this project: we will create a simplified clone of Travian [2], an old real-time [3] player-versus-player [4] strategy game [5], where players build their own city and wage war on one another (less wrinkly readers may be more familiar with the modern counterpart Clash of Clans [6]), built with Pygame [7], a Python library that creates and manages all necessary components to run a game such as game-engine, graphical user interface, sounds, plyer inputs and the like, where each player resides in a separate server (or node) that communicate with the others via an algorithm modelled after Raft's specifications.

This choice follows the authors' interest in exploring Raft capabilities and ease of implementation in a fun and novel way, using a language that while extremely popular is seldom used in such a fashion and, as far as we can tell, never with this mixture of libraries.

Both game and algorithm implementations have been reduced to a reasonably complex proof of concept to keep the project scope manageable: it is possible to instantiate games up to five players, each of which is restricted to the only action of attacking the others, while Raft's functionalities are limited to log replication and overwriting.

Experiments were conducted to evaluate both game responsiveness and the communication algorithm correctness.

All source code is visible at the following link: https://github.com/mhetacc/RuntimesConcurrencyDistribution/blob/main/raftian/raftian.py.

---

[1]Concurrency refers to the ability of a system to execute multiple tasks through simultaneous execution or time-sharing (context switching), sharing resources and managing interactions. It improves responsiveness, throughput, and scalability [6, 12, 14, 16, 18].

[2]Travian: Legends is a persistent, browser-based, massively multiplayer, online real-time strategy game developed by the German software company Travian Games. It was originally written and released in June 2004 as "Travian" by Gerhard Müller. Set in classical antiquity, Travian: Legends is a predominantly militaristic real-time strategy game. Source: https://www.travian.com/international

[3]Real-time games progresses in a continuous time frame, allowing all players (human or computer-controlled) to play at the same time. By contrast, in turn-based games players wait for their turn to play.

[4]Player-versus-player (PvP) is a type of game where real human players compete against each other, opposed to player-versus-environment (PvE) games, where players face computer-controlled opponents.

[5]Strategy video game is a major video game genre that focuses on analyzing and strategizing over direct quick reaction in order to secure success. Although many types of video games can contain strategic elements, the strategy genre is most commonly defined by a primary focus on high-level strategy, logistics and resource management. [15]

[6]Clash of Clans: https://supercell.com/en/games/clashofclans/

[7]Pygame: https://www.pygame.org/docs/

2 PYTHON

We will now discuss the technologies employed in the development of the project (section 2) as well as presenting our implementation of the algorithm Raft.

Python is a high-level, dynamically typed and interpreted programming language that is often used for scripting, data analysis and developing small applications, 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 1) [4, 7] which translates to abundant documentation and resources, and secondly it has a huge ecosystem of libraries that implement all the functionalities we need for this project, namely: *xmlrpc* for the remote procedure calls (RPCs), *threading* to handle local concurrency and *Pygame* to manage everything game-related.
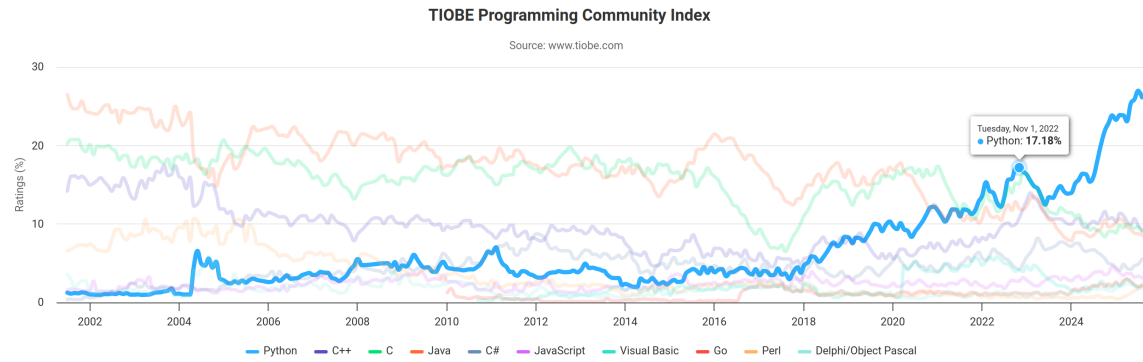

Fig. 1. TIOBE Programming Community Index, focus on Python statistics, 2025. (https://www.tiobe.com/tiobe-index/).

2.1 Remote Procedure Calls

In Raft's specifications it is stated that nodes communicate with each other via remote procedure calls [11], 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 [8] and Cockroach Labs [9], available for many languages (Python included), but we opted for the standard library *xmlrpc* [10] 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 the following code, `client` is an instance of `ServerProxy`, which acts as the client-side interface for XML-RPC, allowing you 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.

---

[8]Netflix Ribbon is an Inter Process Communication library built in software load balancers: https://github.com/Netflix/ribbon
[9]Cockroach Labs is the company behind CockroachDB, a highly resilient distributed database: https://www.cockroachlabs.com/
[10]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

```
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.

```
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
```

For this project, we extended *SimpleXMLRPCServer* to create a class that implements the Raft protocol (more details in section 3)

2.2 Concurrency

In this project the need for concurrent programming arises from two challenges: every server have 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 can be simplified as two *"while true"* loops.

Most Raft implementations achieve concurrency with asynchronous programming, using libraries such as *asyncio* [11], which while powerful and efficient, makes writing code a bit awkward and cumbersome. We thus opted for a more traditional approach using multithreaded programming: in computer science, a thread of execution is the smallest sequence of programmed instruction that can be managed independently by the scheduler [9], and multiple threads may be executed concurrently sharing resources such as memory, which is directly counterpointed to multiprocessing where each process has its own storage space (moreover, processes are typically made of threads).

In Python there are modules in the standard library for both of them, respectively *threading* [12] and *multiprocessing* [13]. 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."* [14].

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

---

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

[12]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

[13]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

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

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 2), that move around the screen offsetting their position each frame by a random amount between minus five and plus five pixels (pseudocode 2.2.1).

Then we ran both of them in various 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) [15] 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.

```
# 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)
```
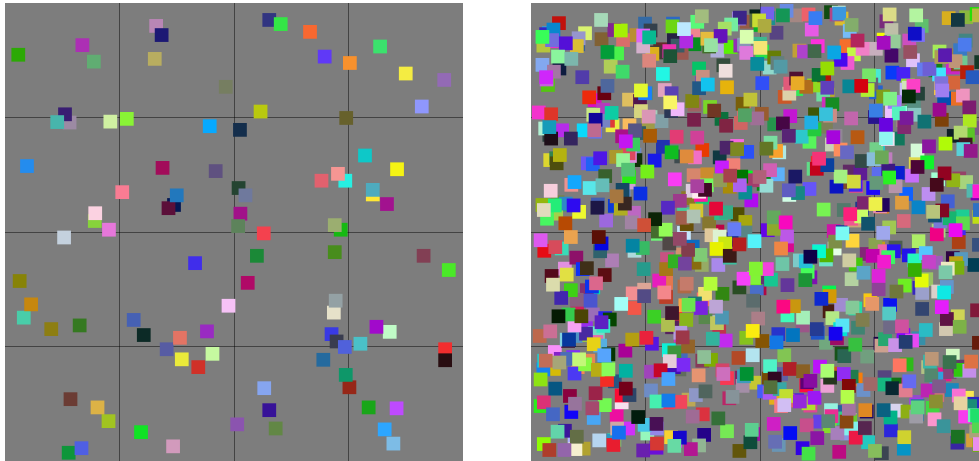

Fig. 2. Two game instances made with Pygame, with respectively 100 and 1000 dots that randomly move around.

Results, shown in the graph at figure 3 shows us that:

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

[15]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, si able to generate frames. Source: https://en.wikipedia.org/wiki/Frame_rate

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.
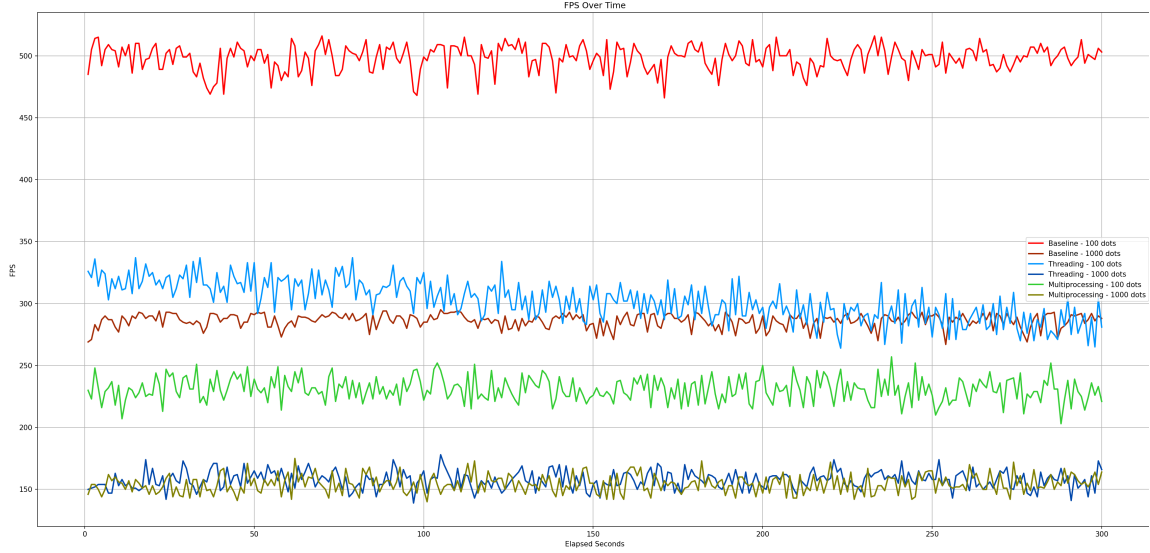

Fig. 3. Performance evaluation graph: red hues for baselines, blue hues for threading and green hues for multiprocessing. Darker shades for 1000 dots and lighter shades for 100 dots game instances.

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.

2.3 Game Engines

There are many ways to implement a graphical user interface: from clever shell tricks like htop [16], to full fledged game engines like Unity [17] or Godot [18] 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.

---

[16]htop is a cross-platform text-mode interactive process viewer: https://htop.dev/

[17]Unity is a cross-platform game engine developed by Unity Technologies: https://unity.com/

[18]Godot is a cross-platform open-source game engine: https://godotengine.org/

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

In this section we are not going to present the Raft algorithm, since Ongaro and Ousterhout already did so in their excellent work "In Search of an Understandable Consensus Algorithm" [11]. Instead, we will discuss the interpretation and implementation we did to adapt it to our own use case.

The algorithm uses three types of nodes, namely leader, follower and candidate, and revolves around three core functionalities: leader election, log replication and overwriting, and cluster membership change. Log compaction is also mentioned, while a byzantine fault tolerant variant is never explored by the original authors.

One last element, fundamental to the functioning of the algorithm, is the *term*: everything happens in a certain term, which divides time logically and increments every election.

Our Raft class directly extends *simpleXMLRPCServer* from XML-RPC module, as seen at code 3.

```python
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)
```

3.1 Node Types

There are three types of node, namely leader, follower and candidate (code 3.1). 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* 3.1

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


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

```python
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 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, it sends periodically (every 150-300ms) an empty remote procedure call called *heartbeat*

3.1.2 Follower Node. All nodes, except for the leader, are classified as followers. They are not allowed to replicate their own log, and when they receive any request they have to forward them to the leader.

   To make sure that the cluster never remains without a leader, every follower have an election timeout (between 150ms and 300ms) which resets every time an RPC from the leader gets received. If it times out, the follower change its state to *candidate*, increment 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 either leader's term is smaller, or leader's index is smaller while having the same term, the append is rejected and a new election is started.

3.1.3 Candidate Node. When a follower's election timeout times out, it becomes a candidate and starts an election. It votes for itself and then wait for either of two outcomes: it wins, thus becoming a new leader, or looses (ether another leader gets elected or the old one manifest 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 4). We defined entires as Python *Data Classes* [19] (decorators that simulate C's structures) such as shown in code snippet 3.2.

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


Fig. 4. Raf's log is fundamentally and array made of entries.

3.3 Log Replication and Overwriting

When the leader have some new entries

3.3.1 Log Compaction.

3.4 Leader Election

3.5 Cluster Membership Change

3.6 Byzantine Raft

ACKNOWLEDGMENTS

REFERENCES

[1] 2009. *Distributed Programs*. Springer London, London, 373–406. https://doi.org/10.1007/978-1-84882-745-5_11

[2] Michiel Buijsman, Devan Brennan, Tianyi Gu, Lester Isaac Simon, Tomofumi Kuzuhara, Spyros Georgiou, Michael Wagner, Ngoc Linh Nguyen, Brett Hunt, Alejandro Marin Vidal, and Tiago Reis. 2024. Newzoo's Global Games Market Report 2024. https://newzoo.com/resources/trend-reports/newzoos-global-games-market-report-2024-free-version

[3] Jessica Clement. 2025. Free-to-play (F2P) games market revenue worldwide from 2018 to 2024. https://www.statista.com/statistics/324129/arpu-f2p-mmo/

[4] Stephen Crass. 2024. The Top Programming Languages 2024. https://spectrum.ieee.org/deep-brain-stimulation-depression

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[6] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2012. *Distributed Systems: Concepts and Design* (5 ed.). Addison-Wesley.

[7] Paul Jansen. 2025. TIOBE Index for August 2025. https://www.tiobe.com/tiobe-index/

[8] Rida Khan and Zack Aboulazm. 2023. $300 Billion of Video Gaming Revenue, by Segment (2017-2026). https://www.visualcapitalist.com/sp/video-games-industry-revenue-growth-visual-capitalist/

[9] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[10] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[11] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[12] David A. Patterson and John L. Hennessy. 2008. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)* (4th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] PWC. 2025. Global entertainment and media industry revenues to hit US$3.5 trillion by 2029, driven by advertising, live events, and video games: PwC Global Entertainment & Media Outlook. https://www.pwc.com/gx/en/news-room/press-releases/2025/pwc-global-entertainment-media-outlook.html

[14] Michael J. Quinn. 1994. *Parallel computing (2nd ed.): theory and practice*. McGraw-Hill, Inc., USA.

[15] A. Rollings and E. Adams. 2003. *Andrew Rollings and Ernest Adams on Game Design*. New Riders. https://books.google.it/books?id=Qc19ChiOUI4C

[16] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2008. *Operating System Concepts* (8th ed.). Wiley Publishing.

[17] A.S. Tanenbaum and M. van Steen. 2017. *Distributed Systems*. CreateSpace Independent Publishing Platform. https://books.google.it/books?id=c77GAQAACAAJ

[18] Albert Y. H. Zomaya. 1996. *Parallel and distributed computing handbook*. McGraw-Hill, Inc., USA.

A   APPENDIX SECTION

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.

A.1   Appendix Subsection

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.