

Applying Distributed System Programming Principles to Web Crawling: Implementation and Evaluation of a Distributed Crawler

Joel Ethan Batac

*Department of Software Technology
De La Salle University
Manila, Philippines
joel_batac@dlsu.edu.ph*

Alonzo Andrei Rimando

*Department of Software Technology
De La Salle University
Manila, Philippines
alonzo_rimando@dlsu.edu.ph*

Abstract—This paper presents a distributed web crawler that applies multiprocessing and distributed systems principles to achieve scalable and fault-tolerant web crawling. The crawler adopts a master-worker architecture in which a central coordinator manages global state while multiple worker nodes perform parallel crawling using Pyro5-based Remote Method Invocation for inter-machine communication. The coordinator maintains consistency through synchronized data structures, dynamic service discovery via a Name Server, and timeout-based fault tolerance that reassigns stalled tasks. Each worker employs multithreading to enable local concurrency, producing dual-layer parallelism across machines and threads. Experimental evaluations across multiple node-thread configurations show that horizontal scaling yields significantly higher throughput than adding threads within a single node, with multi-node setups exhibiting super-linear speedups due to reduced contention and increased network parallelism. A three-node, four-thread-per-node configuration delivered the best sustained performance, successfully scraping 9,315 HTML pages and discovering 18,523 URLs. Results demonstrate that distributed coordination and RMI-driven scheduling substantially enhance performance and reliability for web crawling applications.

Index Terms—Web Crawling, Distributed Systems, Multithreading, Synchronization

I. INTRODUCTION

This project implements a distributed web crawler developed to demonstrate fundamental principles in distributed systems, remote object communication, and parallel task coordination. In contrast to a traditional single-process crawler that sequentially downloads and parses webpages, this system distributes its workload across multiple worker processes—each running independently and potentially deployed on different machines or network environments. A central coordinator process maintains global state, assigns crawling tasks, receives results, and orchestrates overall system behavior. By decentralizing page-fetching operations and enabling multiple workers to run concurrently, the system can scale to cover larger websites more efficiently, handle node failures, and maintain continued progress even in the presence of network latency, intermittent connectivity, or worker crashes. This approach

reflects the real-world design patterns used in large-scale web indexing systems and distributed data-processing pipelines.

At its core, the crawler uses Pyro5 (Python Remote Objects) as the communication middleware that enables transparent remote method invocation. The coordinator exposes several remote endpoints that workers can call to request new URLs, report completed pages, submit extracted hyperlinks, and notify the system of encountered errors. Workers, acting as remote clients, crawl webpages, extract titles and links, and communicate updates to the coordinator in real time.

To maintain correctness and robustness, the system integrates several distributed systems mechanisms. It uses remote object invocation to support communication across networked processes and a Pyro5 Name Server for service discovery, enabling workers to dynamically locate the active coordinator without hardcoding network endpoints. A fault-detection mechanism monitors “in-flight” URLs and requeues any tasks that remain uncompleted beyond a configurable timeout, ensuring progress even if a worker becomes unresponsive. Each worker internally uses multithreading to parallelize crawling at the machine level, demonstrating intra-node concurrency. State consistency is preserved through fine-grained locking within the coordinator, preventing race conditions as multiple workers simultaneously update shared data structures such as the `frontier`, `visited` set, and `results` dictionary. Global shutdown is managed through a timed execution system, which triggers a stop signal after the user-defined duration, along with checks to ensure that remaining in-flight tasks are properly handled before final termination.

The project requirements center on constructing a fully functioning distributed web crawler that can: (1) recursively crawl webpages starting from a given seed URL; (2) distribute crawling tasks across multiple workers; (3) connect these workers to a remote coordinator through Pyro5-based RPC; (4) demonstrate scheduling, synchronization, failure recovery, and coordination consistent with distributed systems principles; (5) terminate gracefully after a user-specified time limit while

ensuring all active workers stop safely; and (6) generate output files summarizing crawling statistics and storing the collected URLs and titles in both text and CSV format.

Overall, this project serves as a practical and applied exploration of distributed systems theory. By combining remote object communication, centralized coordination, concurrency, and fault tolerance, it provides a concrete example of how distributed architectures can be designed to achieve scalability, resilience, and efficient parallel processing. Through this implementation, students gain hands-on experience with key concepts such as remote invocation, distributed coordination, multithreading, and reliable task scheduling—skills highly relevant to modern cloud systems, large-scale web crawling infrastructure, and real-world distributed applications.

II. PROGRAM IMPLEMENTATION

This section presents an in-depth discussion of the internal design, operational workflow, and architectural decisions underpinning the distributed web crawler. The primary objective of this component is not only to detail the mechanics of the system but also to elucidate the distributed systems principles demonstrated through its implementation. The crawler is composed of two primary Python modules—`distributed_crawler.py` and `main.py`—and two shell scripts—`master_run.sh` and `worker_run.sh`—that together orchestrate coordination, parallelism, remote communication, and decentralized execution. The Coordinator machine executes the master script, which initializes the Pyro5 Name Server, launches the Coordinator daemon, and spawns a local Worker. Additional Worker machines join the system via the worker script, thereby contributing distributed computational resources to the crawling task. The following subsections describe the structure, data models, communication patterns, concurrency mechanisms, and fault tolerance policies of the system, offering a comprehensive examination of how distributed systems concepts are realized in the crawler’s design.

A. Program Structure and Execution Flow

The program adheres to a modular, layered structure in which responsibilities are cleanly separated across the Coordinator, the Worker, and the supporting scripts. The Python file `main.py` serves as the command-line entry point. It parses arguments, identifies whether the current execution context is that of a Coordinator or Worker, and dispatches execution to the appropriate initialization function:

```
def main():
    args = parse_args()
    if args.role == "coordinator":
        start_coordinator(start_url=args.start_url,
                          minutes=args.minutes,
                          host=args.host,
                          ns_host=args.ns_host,
                          ns_port=args.ns_port)
    elif args.role == "worker":
```

```
start_worker(start_url=args.start_url,
             worker_id=args.worker_id,
             threads=args.threads,
             ns_host=args.ns_host,
             ns_port=args.ns_port)
```

The Coordinator machine initiates the system’s distributed environment via `master_run.sh`, which executes three sequential tasks:

- 1) start the Pyro5 Name Server to enable remote service discovery,
- 2) launch the Coordinator daemon, and
- 3) spawn a local Worker process, allowing the Coordinator machine to contribute computational resources.

```
python3 -m Pyro5.nameserver -n "$HOST_IP" --port
9090 &
python3 main.py coordinator "$START_URL" "$MINUTES"
...
python3 main.py worker --start-url "$START_URL" ...
```

This design aligns with classical master-worker architectures in distributed systems, in which the master node simultaneously manages task delegation while participating in computation. In contrast, remote Worker machines execute `worker_run.sh`, which exclusively initializes the Worker entity and integrates the machine into the distributed crawler network:

```
python3 main.py worker \
--start-url "$START_URL" \
--worker-id "$WORKER_ID" \
--threads "$THREADS" \
--ns-host "$NS_IP" --ns-port 9090 &
```

Once the Coordinator and Workers are active, execution proceeds into the distributed crawling phase. The Coordinator enters a Pyro5 request loop, responding to remote calls from Workers. Workers continuously request URLs, retrieve content, parse HTML, and submit results. This distributed loop persists until the Coordinator’s countdown timer elapses, at which point the Coordinator stops assigning new work and begins a staged, consistent shutdown. This workflow exemplifies distributed lifecycle management, wherein processes across multiple machines coordinate termination in an orderly manner to preserve consistency.

B. Core Data Structures

The system’s correctness and performance depend heavily on meticulously designed data structures within the Coordinator. These structures represent the global state of the crawling task and form the backbone of the distributed coordination logic. The Coordinator maintains a frontier using a `queue.Queue`, selected deliberately for its thread-safe properties and its suitability for breadth-first traversal of hyperlinked pages:

```
self.frontier = queue.Queue()
```

The visited set records all URLs processed:

```
self.visited = set()
```

The discovered set ensures that newly identified URLs are not redundantly enqueued:

```
self.discovered = set()
```

This distinction between visited and discovered improves crawler efficiency and prevents cycles.

Two additional structures anchor the system's fault tolerance logic: `in_flight`, which records each assigned URL and the timestamp of its assignment, and `worker_for_url`, which records which worker received that assignment:

```
self.in_flight = {}          # url -> timestamp
                           assigned
self.worker_for_url = {}     # url -> worker_id
```

These enable the Coordinator to detect stalled tasks and initiate requeueing. Also we have an additional set data structure called `active_workers`, which is responsible for containing the string IDs of currently registered and active workers.

```
self.active_workers = set()
```

Finally, `results` maps URLs to their parsed titles:

```
self.results = {}
```

All modification operations involving these structures are guarded by a `threading.Lock` to ensure atomicity and prevent race conditions. This design reflects careful consideration of concurrency hazards inherent in distributed systems, particularly those employing multi-threaded remote call handling.

C. Communication Between Machines via Remote Invocation

Inter-machine communication is facilitated by Pyro5, which provides a high-level Remote Method Invocation (RMI) abstraction. Rather than handling raw TCP sockets or marshaling data manually, Pyro5 allows the Coordinator to expose methods as remotely callable functions through the use of decorators:

```
@Pyro5.api.expose
@Pyro5.api.behavior(instance_mode="single")
class Coordinator:
    ...
```

Workers interact with these methods transparently via Pyro5 proxies:

```
proxy = Pyro5.api.Proxy(self.coordinator_uri)
url = proxy.request_url(worker_tag)
proxy.submit_page(worker_tag, url, title, links)
proxy.report_failure(worker_tag, url, reason)
```

Pyro5 abstracts network communication details, enabling the distributed crawler to treat remote interactions as if they were local method calls. Workers discover the Coordinator using the Pyro5 Name Server, which acts as a decentralized directory service. Through:

```
ns = Pyro5.api.locate_ns(host=ns_host, port=9090)
uri = ns.lookup("crawler.coordinator")
```

Workers dynamically retrieve the Coordinator's URI, eliminating the need for static network configuration. This capability significantly enhances deployment flexibility, especially across heterogeneous or dynamic networks, and mirrors real-world distributed systems such as CORBA, gRPC-based microservices, and cloud cluster service discovery mechanisms.

D. Architecture and Design Model

The system follows a master-worker distributed architecture, where a central Coordinator object handles global state while multiple Workers perform decentralized crawling.

The Coordinator Machine acts as a scheduler and global repository for URLs. It stores the frontier, maintains the visited pages, and reassigns tasks when failures occur. Worker machines are stateless except for their local thread pool and HTTP session. Their main role is to pull URLs from the Coordinator's frontier queue through a remote method invocation of its function, `request_url()`. Then, the Worker crawls through the pulled URL and submits the links it extracted to the Coordinator machine's frontier queue.

Notably, each worker machine supports internal multi-threading, allowing even a single physical machine to run multiple crawler threads concurrently. This architecture provides two layers of parallelism: distributed parallelism across machines and local parallelism within each machine.

E. Synchronization and Concurrency Control

Because the Coordinator handles simultaneous remote invocations and internal thread operations, synchronization is critical. Pyro5 automatically spawns handler threads for every incoming remote call, so the Coordinator may simultaneously receive requests like `request_url()`, `submit_page()`, and `report_failure()` from different workers.

To prevent race conditions, the Coordinator wraps all modifications to shared structures within `self.lock` blocks:

```
with self.lock:
    self.in_flight[url] = time.time()
    self.worker_for_url[url] = worker_id
```

This ensures that no two Workers retrieve the same URL, and no updates collide. It is important to take note that choosing a queue as the data structure for the frontier was intentional as Queues in Python are inherently thread-safe. With the

combination of thread-safe Queues, locks and Pyro5 request serialization guarantees that everything remains consistent.

Workers also rely on concurrency control internally. Each worker machine spawns several threads:

```
t = threading.Thread(target=self._thread_main,
    args=(idx+1,), daemon=True)
```

Each thread independently performs crawling operations but safely interacts with the remote Coordinator without causing interference.

F. Coordinator Machine Operation

The Coordinator performs Global Scheduling, fault management, and shutdown control. When it starts, it initializes its core data structures, places the start URL into the frontier, and launches two background threads:

- **Countdown Timer thread** — sets the global stop flag after the time (number of minutes) the user has specified.
- **Requeue Watchdog Thread** — monitors in-flight tasks and reassigns stuck URLs in case of crashes.

Each time a worker requests a URL through `request_url()`, the coordinator assigns one and records the timestamp:

```
url = self.frontier.get_nowait()
self.in_flight[url] = time.time()
self.worker_for_url[url] = worker_id
```

When a page is submitted by a Worker, the Coordinator removes the corresponding entry from `in_flight`, records the page title, and pushes new links from the submitted page into the frontier unless the timer has already expired.

When the timer elapses, the Coordinator stops distributing new tasks (or urls) and enters the shutdown sequence. Once all in-flight tasks are resolved, the Pyro5 daemon halts and the system writes output files.

G. Worker Machine Operation

Workers play the role of distributed tasks executors. Upon startup, each worker

- 1) Locates the nameserver
- 2) Retrieves the Coordinator's URI
- 3) Registers itself as an active worker
- 4) Spawns multiple threads for concurrent crawling

During crawling, each thread repeatedly requests a URL from the Coordinator:

```
url = proxy.request_url(worker_tag)
```

If a valid URL is received, the worker fetches it using an HTTP session:

```
resp = session.get(url, timeout=5,
    allow_redirects=True, stream=True)
```

After parsing the page with BeautifulSoup and extracting hyperlinks, it submits the extracted hyperlinks to the Coordinator, so that other worker machines and their threads can crawl them:

```
proxy.submit_page(worker_tag, url, title, links)
```

If any error occurs, such as connection timeouts, invalid responses, or parsing failures, the worker reports it with:

```
proxy.report_failure(worker_tag, url, "Error
    message")
```

Workers continuously operate until the Coordinator signals a stop, after which they unregister and shut down.

H. Distributed Systems and Concepts Applied

The crawler integrates multiple distributed systems principles. Remote Method Invocation (RMI) enables transparent cross-machine communication. The Name Server provides service discovery, allowing workers to dynamically join the system without prior configuration. Worker pull-based scheduling aligns with modern distributed frameworks that scale easily by simply adding more workers.

The system also showcases fault tolerance through timeout-based requeuing and network error handling. Concurrency principles appear through the combination of multi-threaded workers, Pyro5 request-handling threads, and Coordinator-level locking. These techniques together form a robust, scalable, and fault-tolerant distributed crawler architecture.

I. Output Generation and Data Recording

When the Coordinator completes the crawling session, it writes two output files summarizing the results:

- 1) `distributed_results.txt` — contains aggregate statistics
- 2) `distributed_sites.csv` — contains all successfully scraped URLs and titles

The relevant code uses a lock to ensure the final write operation is consistent:

```
with coordinator.lock:
    with open("./outputs/distributed_sites.csv",
        "w") as f:
        for link, title in
            coordinator.results.items():
            f.write(f"{link},{title}\n")
```

This final step provides a persistent record of all crawling results.

Fault tolerance is implemented primarily through the Coordinator’s watchdog thread. Each time a URL is assigned, its timestamp is stored:

```
self.in_flight[url] = time.time()
```

The watchdog periodically checks if any URLs have been in flight for too long:

```
stuck = [url for url, ts in self.in_flight.items()
         if now - ts > timeout]
```

Stuck URLs are requeued, allowing healthy workers to retrieve and process them:

```
self.frontier.put(url)
```

This ensures the system continues making progress even if a worker crashes, a thread becomes unresponsive, or a network issue occurs. Workers also terminate safely when they lose contact with the Coordinator, preventing rogue or orphaned behavior.

However, an inherent limitation of this fault-tolerance protocol emerges when a target webpage legitimately requires a longer retrieval time than the predefined timeout threshold. In such cases, the Coordinator’s watchdog thread may incorrectly classify the responsible Worker as unresponsive or failed, even though the Worker is merely waiting for a slow server response. Because the watchdog relies solely on elapsed time to determine liveness, long-running network operations—such as crawling large HTML files, pages with heavy embedded resources, or servers experiencing temporary delays—can trigger a false failure detection. As a result, the Coordinator prematurely requeues the URL, while the original Worker continues processing it, leading to duplicated effort, redundant crawling, and potential inconsistencies in the in-flight tracking structure.

This design decision reflects a trade-off inherent in distributed crawling systems: the watchdog must assume that tasks exceeding a certain duration are more likely to result from network failures, stalled HTTP connections, or worker crashes than from legitimately slow responses. In practice, distributed crawlers often adopt this conservative assumption because prolonged network operations correlate strongly with instability or partial failure. Nonetheless, this approach sacrifices accuracy for responsiveness, and the effectiveness of the fault tolerance mechanism depends heavily on the appropriateness of the timeout value relative to the typical latency of the target domain. Fine-tuning or dynamically adjusting the timeout could mitigate this issue, but such enhancements were beyond the scope of the current implementation.

A. Experimental Setup

To evaluate the performance of the distributed crawler, seven experimental configurations were tested. Six of these consisted of short one-minute runs across different combinations of nodes and threads per node specifically:

- 1 node with 4, 8 threads
- 2 nodes with 4, 8 threads
- 3 nodes with 4, 8 threads

These produced six short-duration performance measurements. An additional long-duration run was later executed to assess sustained efficiency under the most favorable configuration.

The selection of thread counts was informed by prior single-node experiments where four threads produced the best trade-off between throughput and overhead. Beyond four threads, local resource contention—particularly CPU scheduling, network waits, and memory pressure—began degrading performance. Thus, thread counts of four and eight were chosen to assess both the optimal baseline and the onset of diminishing returns in a distributed environment.

B. Performance Trends

Fig. 1 illustrates the relationship between the number of nodes and the number of threads per node, measured in pages scraped per minute.

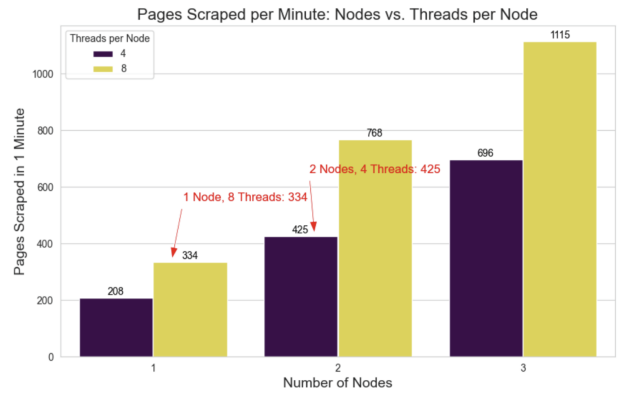


Fig. 1. Pages Scraped per Minute: Nodes vs. Threads per Node

1) Node Scaling outperforms thread scaling

A central observation is that adding more nodes consistently provided higher performance gains than increasing thread counts within a single node. For example:

- 1 node, 8 threads achieved 334 pages/min,
- while 2 nodes, 4 threads each reached 425 pages/min.

Although both configurations employ eight total threads, distributing them across nodes led to an approximately

27% increase in throughput.

2) Super-linear scaling emerges with additional nodes

The performance gains from adding nodes were not merely proportional—they were super-linear. For instance:

- Moving from 1 → 2 nodes (both with 4 threads) improved throughput from 208 → 425 pages/min, more than doubling performance.
- Increasing to 3 nodes with 4 threads yielded 696 pages/min, again surpassing linear expectations.

3) Causes of super-linear behavior

This phenomenon is likely attributable to several distributed-systems effects:

- *Reduced contention on the Coordinator machine:*
With more independent nodes, the coordinator’s CPU and network bandwidth are no longer saturated by a single node generating many simultaneous requests.
- *Improved cache and memory locality:*
Each node maintains its own HTTP session and HTML parsing environment, reducing per-node overhead and avoiding local thrashing.
- *Distributed GIL avoidance:*
Since processes on different nodes operate in separate Python interpreters, they bypass the Global Interpreter Lock, enabling true parallelism across machines.
- *Network concurrency advantages:*
Multiple physical or virtual network stacks perform simultaneous fetches, increasing total request throughput.

Overall, these results demonstrate that horizontal scaling (more nodes) is significantly more impactful than vertical scaling (more threads per node) for distributed crawling.

C. Scaling Analysis

The scaling analysis confirms that distributing work across nodes provides a more efficient and robust pathway to increasing system throughput than merely increasing threads within a single machine. Because each node runs an independent Python process, the system escapes the inherent limitations of Python’s GIL for I/O-heavy workloads. Multiple nodes concurrently fetch webpages without interfering with each other’s CPU, memory, or network resources. Furthermore, the distributed design enhances resilience: if one node encounters congestion, delay, or temporary slowdown (e.g., due to a slow URL, throttling, or intermittent network issues), the remaining nodes continue operating at full capacity. This behavior is visible in the results, where nodes with multiple threads ensure that isolated stalls do not meaningfully impede the global throughput of the crawler. Collectively, these observations em-

phasize that the crawler achieves effective horizontal scalability, reinforcing the benefits of distributed worker architectures in real-world crawling applications.

D. Long Duration Performance

Based on the short-duration experiments, the configuration of three nodes with four threads each was selected for the extended run, balancing concurrency with system stability. Prior single-node experiments indicated that using eight threads occasionally triggered out-of-memory (OOM) errors on cc-scloud due to rapid `frontier` expansion caused by non-HTML resources (e.g., videos, PDFs, images) accumulating in memory. The current implementation mitigates this issue by discarding non-HTML pages at fetch time, but potential memory pressure remains an important consideration when selecting thread counts. Under the selected configuration, the crawler completed a full-run test and achieved:

- 9,315 successfully scraped pages with valid titles, and
- 18,523 total discovered addresses, including error pages and non-HTML content.

Toward the end of execution, the `frontier` gradually emptied and nearly all Worker threads transitioned into an idle state. This behavior indicates that the crawler approached complete coverage of all reachable pages on the target domain. The system’s ability to run to exhaustion without failures also validates the effectiveness of the fault tolerance protocols, coordinator scheduling logic, and distributed workload distribution. Overall, the long-duration experiment demonstrates not only high throughput but also operational stability, confirming that the chosen configuration provides an optimal balance between scalability, reliability, and resource usage.

IV. CONCLUSION

This project demonstrated how core distributed systems techniques can be applied to the design and implementation of a scalable web crawler, integrating multiprocessing at the node level with coordinated distributed execution across multiple machines. The system leveraged Remote Object communication through Pyro5 to enable seamless interaction between Workers and the Coordinator, illustrating how Remote Method Invocation (RMI) facilitates transparent cross-machine communication without requiring explicit management of sockets or serialized message formats. The Coordinator functioned as a centralized authority for global state management—maintaining the `frontier`, tracking visited and `in-flight` URLs, and ensuring that no two Workers crawled the same link—thereby embodying the principles of strong consistency and centralized coordination. To support robustness in a distributed environment, the crawler employed timeout-based fault tolerance, where the Coordinator monitored unreturned tasks and requeued URLs that exceeded a predefined processing threshold. Together, these mechanisms

ensured orderly progress even in the presence of slow responses, intermittent delays, or worker-level faults.

At the Worker level, multiprocessing and thread-level concurrency were used to decompose tasks, allowing each node to run several crawler threads concurrently. Lock-based synchronization protected shared data within each Worker, while the Coordinator's lock guaranteed consistent access to global structures. This combination of distributed coordination and local parallelism reflects the hybrid nature of many real-world systems, where nodes benefit from both intra-machine concurrency and inter-machine cooperation.

The performance evaluation highlighted a central insight: distributed systems techniques significantly improved throughput, but not all forms of parallelism were equally effective. While multithreading within a single node produced speedups initially, its benefits diminished at higher thread counts due to increasing competition for CPU time, memory bandwidth, and interpreter resources. In contrast, distributing work across multiple nodes—each with independent processors, memory hierarchies, and network stacks—led to disproportionately higher performance gains. Because web crawling is fundamentally I/O-bound, horizontal scaling allowed multiple machines to issue parallel network requests without contending for local resources. The empirical results demonstrated super-linear scaling in several configurations, implying that distributing threads across nodes reduced contention on the Coordinator and improved overall system responsiveness. These findings underscore a key principle of distributed computing: for workloads dominated by I/O latency rather than computation, adding machines is more effective than increasing threads within one machine.

Distributed system techniques also improved reliability. The Coordinator's fault-tolerance protocol ensured that unresponsive Workers or stalled requests did not halt the crawl, and its consistent global state prevented redundant work. Workers were able to join or exit the system cleanly, reflecting the desirable property of elasticity in distributed architectures. Although the timeout-based fault detection occasionally misinterpreted slow but valid responses as failures, the design successfully balanced responsiveness with the need for continuous progress—an acceptable trade-off for large-scale crawling tasks.

In summary, this project provided a concrete demonstration of how distributed coordination, RMI-based communication, multi-node parallelism, and fault-tolerant task scheduling can be combined to scale a real-world web-crawling workload. The results highlight not only the performance advantages of distributed systems but also the architectural considerations required to achieve correct and efficient execution in a multi-node environment. Ultimately, the crawler's behavior reaffirmed a central principle in distributed systems design: the structure of parallelism matters. For network-bound applications such as web crawling, horizontal scaling across machines offers substantially greater and more sustainable performance

gains than vertical scaling through additional threads on a single device.

REFERENCES

- [1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.
- [2] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, MA, USA: Addison-Wesley, 2000.
- [3] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS Spring Joint Comput. Conf.*, 1967. [Online]. Available: <https://api.semanticscholar.org/CorpusID:195607370>
- [4] Python Software Foundation, "threading — Thread-based parallelism," *Python 3 Documentation*. [Online]. Available: <https://docs.python.org/3/library/threading.html>
- [5] Python Software Foundation, "concurrent.futures — Launching parallel tasks," *Python 3 Documentation*. [Online]. Available: <https://docs.python.org/3/library/concurrent.futures.html>
- [6] Python Software Foundation, "queue — A synchronized queue class," *Python 3 Documentation*. [Online]. Available: <https://docs.python.org/3/library/queue.html>
- [7] Requests Library, "requests — HTTP for Humans," *Requests Documentation*. [Online]. Available: <https://requests.readthedocs.io/en/latest/>
- [8] L. Richardson, "Beautiful Soup Documentation," Crummy.com. [Online]. Available: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- [9] De La Salle University, "Official Website," 2025. [Online]. Available: <https://www.dlsu.edu.ph/>