Presented to the College of Computer Studies
De La Salle University - Manila

Term 1, A.Y. 2024-2025

In partial fulfillment of the course

In NSDYST

Machine Project Documentation

Group No. 2

Submitted by:

Aperin, Johanna Christine

Elloso, Jilliane Margaux

Pecson, Richard

Ypaguirre, Gebromel

Submitted to:

Gregory Cu

December 3, 2024

Distributed Programming Project – Email Address Web Scraper

Johanna Aperin
College of Computer Studies
De La Salle University
Manila Philippines
johanna_aperin@dlsu.edu.ph

Jilliane Elloso
College of Computer Studies
De La Salle University
Manila Philippines
jilliane_elloso@dlsu.edu.ph

Richard Pecson
College of Computer Studies
De La Salle University
Manila Philippines
richard_pecson@dlsu.edu.ph

Gebromel Ypaguirre
College of Computer Studies
De La Salle University
Manila Philippines
gebromel_ypaguirre@dlsu.edu.ph

***Abstract*—This document presents a distributed email scraping system designed to efficiently extract email addresses from web pages. Leveraging Python and the Pyro4 framework, the system employs a server-client architecture where multiple nodes collaborate to crawl and parse web pages concurrently. The process is containerized using Docker to ensure scalability and portability. The system allows users to define parameters such as target URLs, scraping duration, and the number of nodes.**

Keywords—remote procedure calling, remote method invocation, Email scraping, distributed system, web crawling, Pyro4, Python, Docker, server-client architecture, data extraction, concurrency, scalability, fault tolerance, web scraping automation.

I. Introduction

Websites play a critical role in enabling organizations to disseminate information, engage with potential customers, and establish connections with partners. These websites often include email addresses as a primary point of contact, making them a valuable resource for communication. However, the process of manually extracting email addresses from each webpage is tedious, time-consuming, and prone to errors, especially for websites with extensive or frequently updated content [1].

To address these challenges, web scraping has emerged as an effective solution. Web scraping involves the use of automated tools to extract data from websites systematically. When specifically programmed, web scrapers can identify and collect email addresses across multiple pages of a website with precision. The integration of parallel programming techniques further enhances the efficiency of this process, enabling simultaneous data extraction from multiple web pages [2]. This parallelized approach not only minimizes the time required for scraping but also improves the scalability and reliability of the operation.

The proposed system, implemented using Python and Docker, employs a server-client architecture facilitated by the Pyro4 framework. The server is responsible for managing multiple scraping nodes that concurrently collect data from web pages, while the client initiates the crawling process by providing target URLs and configuring parameters such as the scraping duration and the number of nodes to deploy. Each node is tasked with parsing HTML content to identify and collect email addresses, which are subsequently saved into CSV files for further analysis.

This approach addresses key challenges in web scraping, such as handling high volumes of web pages and ensuring fault tolerance during the crawling process. Additionally, the use of Docker containers ensures portability and scalability, allowing the system to be easily deployed in various environments with minimal setup.

By automating email extraction, web scrapers reduce manual effort, optimize resource utilization, and provide organizations with a streamlined method to gather essential contact information. This paper explores the development and implementation of a web scraper that leverages parallel programming to efficiently extract email addresses from websites, addressing the challenges associated with manual data collection.

II. Program Implementation

A. Use of Distributed System APIs

```
import Pyro4
import threading

@Pyro4.expose  # Expose the class for remote access
class EmailScraperServer:
    def __init__(self):
        self.lock = threading.Lock()  # Ensure thread-safe
operations
        self.client_counter = 0  # Track connected clients

    def email_web_scraper(self, target_url,
max_time_minutes, max_nodes):
        # Implementation of the scraping logic...
        pass
```

```
def start_server():
    Pyro4.Daemon.serveSimple(
        {
            EmailScraperServer: "email_scraper.server"  #
Register the object with a name
        },
        ns=True,  # Use the Pyro4 name server for service
discovery
        host="192.168.100.23",  # Server host IP address
        verbose=True  # Enable detailed logs for debugging
    )

if __name__ == "__main__":
    print("Starting Email Scraper Server...")
    start_server()
```

Figure 1. server.py use of Pyro4

The server code exposes the 'EmailScraperServer' class as a Pyro4 remote object using the '@Pyro4.expose' decorator, enabling remote clients to call its methods. The '`start_server()`' function initializes a Pyro4 daemon via '`Pyro4.Daemon.serveSimple`', registering the '`EmailScraperServer`' object with the name "`email_scraper.server`" to make it discoverable by clients. The '`ns=True`' parameter activates the Pyro4 name server, allowing clients to resolve the server's name to a network address (e.g., 192.168.100.23). A threading lock ('self.lock') ensures thread-safe operations on shared resources, such as email storage and the client counter, enabling the handling of concurrent client requests without race conditions. The server also processes web scraping tasks and writes the results, including emails and statistics, to CSV files, which are then returned to the client for access.

```
import Pyro4

def connect_to_server():
    # Connect to the Pyro4 server using the name
registered in the Pyro name server
    server =
Pyro4.Proxy("PYRONAME:email_scraper.server@192.1
68.100.23")
    return server

def main():
    server = connect_to_server()  # Establish connection to
the server

    target_url = input("Enter Target URL to Scan: ").strip()
    max_time_minutes = int(input("Enter Scraping Time
(in minutes): "))
    max_nodes = int(input("Enter Number of Nodes
(pages) to Scrape: "))

    # Invoke the server's remote method
    result = server.email_web_scraper(target_url,
max_time_minutes, max_nodes)

    # Display the results received from the server
    print(f"Scraping completed. Results saved in
{result['emails_file_path']} and
{result['stats_file_path']}.")
    print(f"Emails found: {result['emails_found']}, Pages
scraped: {result['pages_scraped']}")

if __name__ == "__main__":
    main()
```

Figure 2. client.py use of Pyro4

The client connects to the Pyro4 server by using '`Pyro4.Proxy`' with the name "email_scraper.server", which the Pyro4 name server resolves to the server's actual network address. Through this connection, the client remotely invokes the '`email_web_scraper`' method on the server object, passing parameters such as '`target_url`', '`max_time_minutes`', and '`max_nodes`'. The server processes the request and returns the results synchronously, including file paths for emails and statistics, along with summary data.

B. Sharing of Data Between Processes

The server and client code share data and interact via the Pyro4 framework, which facilitates Remote Procedure Calls (RPC) between processes running on separate systems or on the same machine.

```
import Pyro4
from collections import deque
import requests
from bs4 import BeautifulSoup
import re
import urllib.parse
import time
import threading
import csv
import os

@Pyro4.expose
class EmailScraperServer:
    def __init__(self):
        self.lock = threading.Lock()
        self.client_counter = 0
        self.nodes = []

    def create_nodes(self, max_nodes):
        """Automatically create and register nodes based on
max_nodes."""
        with self.lock:
            self.nodes = [f"node_{i+1}" for i in
range(max_nodes)]
            print(f"[+] Created and registered {max_nodes}
nodes: {self.nodes}")

    def email_web_scraper(self, target_url,
max_time_minutes, max_nodes):
        try:
            # Automatically create nodes based on
max_nodes
            self.create_nodes(max_nodes)

            with self.lock:
```

```
            self.client_counter += 1
            client_id = self.client_counter
        print(f"[+] Client {client_id} connected and
started a scraping task.")

        # Crawl the target URL and get all URLs to
scrape
        urls = self.crawl_target_url(target_url)
        total_urls = len(urls)
        if total_urls == 0:
            raise ValueError(f"No URLs found to scrape
from {target_url}")

        print(f"[+] Total URLs to scrape: {total_urls}")

        # Distribute URLs across the nodes
        urls_per_node = total_urls // max_nodes
        nodes_workload = {node_id: [] for node_id in
self.nodes}  # Initialize empty workloads for each node

        for i, url in enumerate(urls):
            node_id = self.nodes[i % max_nodes]  #
Distribute URLs to nodes
            nodes_workload[node_id].append(url)

        # Scrape emails on each node (simulated using
threads)
        emails = {}
        stats = {"url": target_url, "pages_scraped": 0,
"emails_found": 0}

        # Keep track of the start time to enforce the max
time limit
        start_time = time.time()

        def scrape_emails(node_id, urls_to_scrape):
            """Function to scrape emails for a specific
node."""
            nonlocal emails, stats
            print(f"[+] Node {node_id} starting email
scraping...")
            node_emails = set()

            for url in urls_to_scrape:
                # Check if we've exceeded the max time
                elapsed_time = (time.time() - start_time) / 60
# in minutes
                if elapsed_time > max_time_minutes:
                    print(f"[!] Max time exceeded, stopping
scraping on Node {node_id}.")
                    break

                page_emails = self.scrape_page_emails(url)
                node_emails.update(page_emails)
                stats["pages_scraped"] += 1

            with self.lock:
                for email in node_emails:
                    if email not in emails:
                        emails[email] = {"name": "N/A",
"office": "N/A", "department": "N/A"}

            print(f"[+] Node {node_id} finished scraping.
Found {len(node_emails)} emails.")

        threads = []
        for node_id, urls_to_scrape in
nodes_workload.items():
            thread =
threading.Thread(target=scrape_emails, args=(node_id,
urls_to_scrape))
            threads.append(thread)
            thread.start()

        # Wait for all threads (nodes) to finish
        for thread in threads:
            thread.join()

        stats["emails_found"] = len(emails)

        # Save results to CSV
        emails_file = f"emails_scrape_{client_id}.csv"
        stats_file = f"scraping_stats_{client_id}.csv"

        emails_file_path = os.path.abspath(emails_file)
        with open(emails_file, "w", newline='',
encoding='utf-8') as email_file:
            csv_writer = csv.writer(email_file)
            csv_writer.writerow(["Email", "Name",
"Office", "Department"])
            for email, details in emails.items():
                csv_writer.writerow([email, details["name"],
details["office"], details["department"]])

        stats_file_path = os.path.abspath(stats_file)
        with open(stats_file, "w", newline='',
encoding='utf-8') as stats_file:
            csv_writer = csv.writer(stats_file)
            csv_writer.writerow(["Website URL", "Pages
Scraped", "Emails Found"])
            csv_writer.writerow([stats["url"],
stats["pages_scraped"], stats["emails_found"]])

        print(f"[+] Client {client_id}: Scraping
completed. Files saved as {emails_file_path} and
{stats_file_path}.")
        return {
            "emails_file_path": emails_file_path,
            "stats_file_path": stats_file_path,
            "emails_found": stats["emails_found"],
            "pages_scraped": stats["pages_scraped"]
        }

    except Exception as e:
        print(f"[ERROR] Exception in
email_web_scraper: {e}")
        raise



def crawl_target_url(self, target_url):
    """Crawl the target URL to get all the internal
```

```
links."""
        urls = set()
        base_url = urllib.parse.urlsplit(target_url).scheme +
'://' + urllib.parse.urlsplit(target_url).hostname
        try:
            response = requests.get(target_url, timeout=10)
            response.raise_for_status()
            soup = BeautifulSoup(response.text, "lxml")
            for anchor in soup.find_all("a", href=True):
                link = anchor['href']
                if link.startswith('/'):
                    link = base_url + link
                elif not link.startswith('http'):
                    link = urllib.parse.urljoin(base_url, link)
                urls.add(link)
        except requests.exceptions.RequestException as e:
            print(f"[!] Error crawling {target_url}: {e}")
        return list(urls)

    def scrape_page_emails(self, url):
        """Scrape email addresses from the page."""
        page_emails = []
        try:
            response = requests.get(url, timeout=10)
            page_emails =
re.findall(r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9-]+\.[a-zA-Z
]{2,}", response.text)
        except requests.exceptions.RequestException as e:
            print(f"[!] Error scraping emails from {url}: {e}")
        return page_emails

def start_server():
    Pyro4.Daemon.serveSimple(
        {
            EmailScraperServer: "email_scraper.server"
        },
        ns=True,
        host="192.168.100.23",
        verbose=True
    )

if __name__ == "__main__":
    print("Starting Email Scraper Server...")
    start_server()
```

Figure 3. server.py Interaction and Data Sharing

The server contains the 'EmailScraperServer' class, which implements the '`email_web_scraper`' method to perform email scraping tasks.

It uses threading locks ('`self.lock`') to ensure thread-safe operations when multiple clients invoke the server's methods concurrently. For instance, the 'client_counter' is incremented safely within the lock.

Once a client calls the '`email_web_scraper`' method, the server processes the task (scrapes web pages, finds emails, and writes results to files) and returns the paths to the result files and scraping statistics.

```
import Pyro4

def connect_to_server():
    # Connect to the server via Pyro4 proxy
    return
Pyro4.Proxy("PYRONAME:email_scraper.server@192.1
68.100.23")

def main():
    server = connect_to_server()

    # Gather input for scraping task
    target_url = input("Enter Target URL to Scan: ").strip()
    max_time_minutes = int(input("Enter Scraping Time
(in minutes): "))
    max_nodes = int(input("Enter Number of Nodes
(pages) to Scrape: "))

    # Invoke the remote method on the server
    result = server.email_web_scraper(target_url,
max_time_minutes, max_nodes)

    # Display results to the user
    print(f"Scraping completed. Results saved in
{result['emails_file_path']} and
{result['stats_file_path']}.")
    print(f"Emails found: {result['emails_found']}, Pages
scraped: {result['pages_scraped']}")

if __name__ == "__main__":
    main()
```

Figure 4. client.py Interaction and Data Sharing

The client connects to the server using Pyro4's proxy mechanism (`'Pyro4.Proxy("PYRONAME:email_scraper.server@192.168.100.23")'`). It takes user inputs (target URL, scraping duration, and max nodes/pages to scrape) and invokes the server's '`email_web_scraper`' method. The server processes the request, and the client receives the returned results (file paths and stats), which it then displays to the user.

The server ensures thread safety by using '`threading.Lock`' to synchronize access to shared resources such as 'client_counter' and the 'emails' dictionary. Scraping results are stored in CSV files ('emails_file' and 'stats_file'), making them readily accessible to clients or available for future analysis. Additionally, Pyro4 simplifies network communication by abstracting the complexities of inter-process interaction, enabling efficient data sharing between the client and server over a network.

C. Distributed Systems Techniques

The system utilizes message passing to enable seamless communication between clients and the server. Clients invoke methods on the server through Pyro4's Remote Procedure Call (RPC) mechanism, with Pyro4 handling the serialization of data, such as file paths and results, for transmission between the server and clients. Additionally, coordination techniques are employed by the server to manage concurrent client requests effectively. A threading lock ensures thread safety by preventing simultaneous access to shared resources like the client counter, thus avoiding race conditions. Furthermore, URLs to scrape are organized in a thread-safe queue using ‘collections.deque’, ensuring orderly and efficient task processing.

In the client code, ‘Pyro4.Proxy’ is used to establish a connection to the server. This acts as a message-passing mechanism, allowing the client to invoke the ‘email_web_scraper’ method remotely on the server.

```
server =
Pyro4.Proxy("PYRONAME:email_scraper.server@192.1
68.100.23")
```

Figure 5. client.py Pyro4 establishing connection

The client sends the target URL, maximum scraping time, and the number of pages to scrape to the server. This is a direct example of sending serialized data from the client to the server.

```
result = server.email_web_scraper(target_url,
max_time_minutes, max_nodes)
```

Figure 6. client.py Sending Serialized Data

The server returns the paths to the generated files and scraping statistics to the client.

```
return {
    "emails_file_path": emails_file_path,
    "stats_file_path": stats_file_path,
    "emails_found": stats["emails_found"],
    "pages_scraped": stats["pages_scraped"]
}
```

Figure 7. client.py Scraping Statistics

The ‘threading.Lock’ ensures thread-safe access to shared resources, like the ‘client_counter’ and the ‘emails’ dictionary. This prevents race conditions when multiple clients connect and modify the ‘client_counter.’

```
with self.lock:
    self.client_counter += 1
```

Figure 8. client.py Client Counter

Similarly, the lock is used when adding new emails to the ‘emails’ dictionary.

```
with self.lock:
    if email not in emails:
        emails[email] = {"name": "N/A", "office": "N/A",
"department": "N/A"}
```

Figure 9. client.py Client Counter

A deque is used to maintain the queue of URLs to scrape.

```
urls = deque([target_url])
```

Figure 10. client.py Dequeueing

URLs are safely added to the queue when new links are discovered. This ensured orderly processing and prevents duplicates, even when multiple threads or requests operate simultaneously.

```
if link not in urls and link not in scraped_urls:
    urls.append(link)
```

Figure 11. client.py Preventing Duplicates

D. Workload Distribution

The workload distribution among the nodes in this project works by ensuring efficient task allocation and balanced resource usage.

```
nodes_workload = {node_id: [] for node_id in self.nodes}
# Initialize workloads
for i, url in enumerate(urls):
    node_id = self.nodes[i % max_nodes]  # Distribute
URLs
    nodes_workload[node_id].append(url)
```

Figure 12. Distribution of URLs among nodes

In the figure above, the workload is initialized by going through the nodes first. It then iterates through the URLs and assigns each URL to a node based on the modulo operator to ensure equal distribution of the URLs among the nodes.

```
def scrape_emails(node_id, urls_to_scrape):
    node_emails = set()
    for url in urls_to_scrape:
        if (time.time() - start_time) / 60 >
max_time_minutes:
            break
        page_emails = self.scrape_page_emails(url)
        node_emails.update(page_emails)
    with self.lock:
        for email in node_emails:
            emails[email] = {"name": "N/A", "office": "N/A",
"department": "N/A"}
```

Figure 13. Email scraping with thread locking

Each node operates in parallel using threads, where each thread executes the *scrape_emails()* function to process the URLs assigned to its specific node and extracting the email addresses accordingly. It is then stored in a shared *emails* dictionary that uses *self.lock* to ensure thread safety during the concurrent execution.

III. Result

The Web Scraper is a distributed application designed to efficiently crawl websites, retrieve internal URLs, and extract email addresses. The workload is dynamically assigned to nodes, enabling balanced and concurrent processing via multithreading. Additionally, the scraper enforces a user-defined time limit for operations, ensuring resource efficiency and preventing excessive runtime. The output of the scraper includes two CSV files: one containing the scraped emails with placeholders for metadata such as names and departments, and another summarizing statistics like the number of pages scraped and emails found.

Email	Name	Office	Department
shorecenter@dlsu.edu.ph	N/A	N/A	N/A
marinestation@dlsu.edu.ph	N/A	N/A	N/A
erio@dlsu.edu.ph	N/A	N/A	N/A

Figure 14. CSV of Scraped Emails

Website URL	Pages Scraped	Emails Found
https://www.dlsu.edu.ph/	214	3

Figure 15. CSV of Statistics

Performance-wise, the scraper excels in accuracy but effectively identifies email addresses and eliminates duplicates. It demonstrates efficiency by reducing execution time through parallel processing and adhering to time constraints. Its scalability is evident in the dynamic creation of nodes, which adapt to varying workloads. The application also ensures reliability with robust error handling for unreachable pages or invalid URLs and uses thread locks to protect shared resources.

```
[+] Node node_69 finished scraping. Found 0 emails.
[!] Error scraping emails from https://www.dlsu.edu.ph/wp-content/uploads/pdf/viewbook/graduate/ccs
ead timed out.
[!] Error scraping emails from https://www.dlsu.edu.ph/wp-content/uploads/pdf/viewbook/graduate/gcc
Read timed out.
[+] Node node_70 finished scraping. Found 0 emails.
[+] Node node_77 finished scraping. Found 0 emails.
[!] Error scraping emails from https://www.dlsu.edu.ph/wp-content/uploads/pdf/stratcom/2401/2023/24
out.
```

Figure 16. Error-handling: Unreachable Pages

```
[!] Error scraping emails from https://www.dlsu.edu.ph/research/research-centers/bnscwc/: HTTPSConnectionPool(host='www.dlsu.edu.ph', port=443): Read timed out. (read timeout=10)
[!] Error scraping emails from https://www.dlsu.edu.ph/soll/: HTTPSConnectionPool(host='www.dlsu.edu.ph', port=443): Read timed out. (read timeout=10)
[!] Error scraping emails from https://www.dlsu.edu.ph/shoot-that-ball-dlsu-nba-2k-team-aims-for-victory-in-season-87/: HTTPSConnectionPool(host='www.dlsu.edu.ph', port=443): Read
 timed out. (read timeout=10)
[!] Error scraping emails from https://www.dlsu.edu.ph/colleges/cla/academic-departments/theology-religious-education/: HTTPSConnectionPool(host='www.dlsu.edu.ph', port=443): Read
 timed out. (read timeout=10)
[!] Error scraping emails from https://www.dlsu.edu.ph/administration/: HTTPSConnectionPool(host='www.dlsu.edu.ph', port=443): Read timed out. (read timeout=10)
[!] Error scraping emails from https://www.dlsu.edu.ph/colleges: HTTPSConnectionPool(host='www.dlsu.edu.ph', port=443): Read timed out. (read timeout=10)
[!] Error scraping emails from https://animo.sys.dlsu.edu.ph: HTTPSConnectionPool(host='animo.sys.dlsu.edu.ph', port=443): Read timed out. (read timeout=10)
[!] Error scraping emails from https://www.linkedin.com/school/de-la-salle-university/: HTTPSConnectionPool(host='www.linkedin.com', port=443): Read timed out. (read timeout=10)
```

Figure 17. Error-handling: Invalid URLs

The use of regular expressions provides a highly efficient way to identify email addresses that conform to standard formats, such as user@domain.com. Regex is computationally lightweight and excels at precisely matching well-formed email patterns. However, its effectiveness is limited when dealing with obfuscated emails, such as those written as user[at]domain[dot]com, or in cases where email patterns are embedded within complex or malformed data. Additionally, regex does not account for the context of the extracted email, potentially leading to the inclusion of irrelevant or placeholder addresses. This results in a high precision but moderate recall, as obfuscated or non-standard email formats remain undetected.

The crawling methodology complements regex by navigating through web pages and following hyperlinks to explore a broader range of content. This approach enhances recall by increasing the chances of discovering emails spread across multiple pages within a domain. The crawling process is particularly effective for static websites or those with consistent page structures. However, it faces challenges when dealing with dynamic or protected content, such as CAPTCHA-protected pages or websites with restrictive robots.txt files. Furthermore, irrelevant or redundant pages may introduce noise, impacting the precision of the overall system. Controlling the depth of crawling is crucial to strike a balance between capturing relevant emails and maintaining efficiency.

IV. Conclusion

This project demonstrates the effective use of distributed systems principles to create a scalable and efficient email scraping solution. By leveraging Pyro4 for remote method invocation and implementing message-passing mechanisms for communication between client and server, the system efficiently distributes tasks while maintaining coordination and synchronization.

The thread-safe design ensures that multiple clients can interact with the server concurrently without data corruption or conflicts. Additionally, the use of the naming server simplifies the discovery and communication between distributed components, making the architecture more robust and easier to manage.

This project showcases the power of distributed systems in solving real-world problems, such as web scraping, by distributing workloads across multiple clients and handling concurrency effectively. With these features, the project serves as a foundation for more advanced distributed applications, combining scalability, coordination, and efficiency in a networked environment.

References

[1] “Web Scraping Techniques and Applications: A Literature Review,” ResearchGate, 2021, doi: https://doi.org/10.52458//978-93-91842-08-6-38.

[2] M. A. Rahman, K. Kowsher, and M. R. Islam, "Email extraction from websites using automated web scraping," in Proc. Int. Conf. Comput. Appl. Ind. Eng., 2021, pp. 45–50.