Revolutionizing Concurrent Crawling: A Novel Approach to Enhance PHP-Python Integration using AMQP, Selenium, Celery, and RabbitMQ

Yosua Alvin Adi Soetrisno
Department of Electrical Engineering
Diponegoro University
Semarang, Indonesia
yosua@live.undip.ac.id

M. Arfan
Department of Electrical Engineering
Diponegoro University
Semarang, Indonesia
arfan@elektro.undip.ac.id

Eko Handoyo
Department of Electrical Engineering
Diponegoro University
Semarang, Indonesia
eko_handoyo@elektro.undip.ac.id

Aghus Sofwan
Department of Electrical Engineering
Diponegoro University
Semarang, Indonesia
asofwan@elektro.undip.ac.id

Enda Wista Sinuraya
Department of Electrical Engineering
Diponegoro University
Semarang, Indonesia
enda_sinuraya@elektro.undip.ac.id

Maman Somantri
Department of Electrical Engineering
Diponegoro University
Semarang, Indonesia
mmsomantri@live.undip.ac.id

***Abstract*— This research proposes a practical solution for seamlessly integrating PHP with Python in web development, focusing on achieving efficient web crawling. The problem is that many PHP applications need to call the Python application for machine learning or crawling work. With default functions from PHP, such as PHP-exec, the Python program could be executed but cannot be maintained smoothly if the program is a lengthy task in the background. By leveraging the AMQP (Advanced Message Queuing Protocol) library and the Selenium Crawler, Celery, and RabbitMQ, we establish interoperability between PHP and Python. In our approach, PHP acts as the front end, initiating web crawling tasks by doing some action in the web component. These requests are queued with a message broker application such as RabbitMQ, the message broker. RabbitMQ connected with Celery for the seamless scheduling and execution of tasks. This research enables effective web crawling and concurrent data scraping by seamlessly integrating the Selenium Crawler with Celery. Results from extracted data from the crawler are saved to the database to give a certain status of whether the data collection process is done or pending. Through experimentation, we validate the effectiveness of our seamlessly integrated approach by making a variation of worker and concurrent connection. The testing scenario shows that increasing workers only increase a small amount of memory. This result indicates that workers could help maintain the response time if there is some user, but need some consideration based on the number of users and availability of memory and CPU.**

Keywords—AMQP, Celery, RabbitMQ, Selenium, interoperability

I. INTRODUCTION

When building things on the internet, sometimes we need interoperability between computer languages to use the true nature power of each language to get everything to work together. This task can be tricky, especially when we want a program that searches the data from a web application or the internet. This research built an alternative way to connect PHP and Python using special queue tools like the AMQP library, Celery, and RabbitMQ. The system is also connected to the Selenium crawler to extract data from the web application and get tuned with Celery to do a multi-crawling at once.

PHP with the Code Igniter framework helps manage user requests and start web crawling tasks. These requests can be for specific information, like searching for things from a certain period. RabbitMQ supports ensuring these tasks get handled in the queue. Celery ensures that the crawling tasks can be done in concurrent workers queued, extending the capability to use our computer's resources wisely. Selenium crawler was added for web crawling and data scraping jobs [1]. Selenium crawler could search website components for automatic content exploration and get information from them. The integrated approach has been proven effective through thorough experimentation, particularly regarding integrating PHP for connecting the GUI interface with the database and Python for web crawling [2].

This research proposes an effective method of combining Python and PHP to enable efficient web crawling. With this strategy, developers can leverage Python's web crawling capabilities to improve PHP application capabilities by optimizing the management of task queues and message exchanges [3]. This process could make a Python script called by some users with different sessions.

The proposed solution aims to advance the integration of PHP and Python, offering a practical solution for effective web crawling. This mechanism enables developers to extract crucial information from diverse web sources by incorporating Python's web crawling features into PHP applications, facilitating the creation of data collection and information-driven web applications known as microservices. A testing mechanism has also been implemented to evaluate the crawler's performance when multiple users access it concurrently compared to the available workers.

II. RELATED WORK

In this section, there is a thorough review of research related to the integration of PHP with Python, effective web crawling using Selenium, and the utilization of RabbitMQ for queue management. One notable research by Thirupathi [4] focuses on IoT communication protocols and offers the Web of Things as a potential solution. Web of Things emphasizes the importance of HTTP and MQTT for maintaining IoT data stream queues and technologies such as HTTP and RDF for facilitating communication and data description between devices and web services. The Web of Things enables IoT devices to be remotely monitored, controlled, and accessed in real time, enabling intelligent decision-making.

Another approach [5] uses a queue handling engine of event logs in IT infrastructure. It connects different system components with a dedicated Syslog server, a NoSQL Cassandra database, and the Advanced Message Queuing Protocol (AMQP). AMQP provides features like message orientation, queuing, routing, reliability, and security [6]. Apache QPID is the queue-handling engine that transfers messages from the Syslog server formed in the queue to be parsed, saving the NoSQL Cassandra database messages.

Baier's research [7] explores a multi-agent platform that enables the production of interactive agents that can perform various tasks, which in our research applied as using multiple crawlers. Baier integrates sensors and interpretation components in multimodal data known as episodic graphs. It provides a versatile structure for developing agents that can effectively operate in complex real-world scenarios.

RabbitMQ and Celery are tools combined for communication and task management within road user services [8]. RabbitMQ acts as a message broker, enabling the transfer of client messages between different systems and applications. Celery could execute tasks that require significant computational resources [9] as the task management engine and asynchronously across multiple worker servers. Celery utilizes the capabilities of both RabbitMQ and Redis technologies to ensure efficient task distribution and management [10].

The Fine-Grained and Coarse-Grained models [11], another Celery model, manage asynchronous tasks within each node. RabbitMQ is vital for exchanging sub-tasks and their results among worker nodes. This strategy ensures efficient task execution, faster computation, and improved scalability. Additionally, RabbitMQ facilitates communication between nodes in the Coarse-Grained model, promoting better node interaction management and organization.

There is some testing mechanism and parameter to be tested and used in other research. The parameter is cost and time or efficiency [12]. There is also a parameter for the variation of task amount, page category, and thread per second [13]. For crawl-queue development over time, there is runtime and several crawlers in the queue [14].

This review provides essential and valuable information about the present condition of the related research and methods for integrating PHP with Python, effective web crawling, and the application of RabbitMQ for queue management. It pinpoints specific research areas and highlights these technology's significance in various fields, such as IoT, event log management, and communication services for road users. This research focuses on the technology that could help the integration of PHP and Python together and testing if there is a more efficient way to do crawling after incorporating task management.

III. Methodology

This section presents a comprehensive guide on implementing PHP-Python integration through the AMQP Library, Selenium Crawling with Celery, and RabbitMQ for queue management. This research set up the following architecture and environment to achieve this integration.

A. System Design Architecture

The HTTP request/response cycle, like social media or chat messages, is generally fast and synchronized, with communication between clients and servers taking milliseconds. Although the data transaction is fast, there is another process, such as crawling is a time-consuming task. The command to do a crawling could come quickly because many users could instruct the same tasks, but the crawling process takes longer than ordering the crawling. This situation must be handled asynchronously by implementing a task queue. A task queue is a solution to save fast requests to do slow processes later in a queued manner. Fast requests could come with a risk of data loss if not queued. This condition is also the problem that also addressed with this research.

For a web application to operate efficiently, it is essential to distinguish between shorter and longer tasks that may require scheduling or external interactions. Longer tasks should be handled separately in different processes to ensure optimal performance and responsiveness for a better user experience. The asynchronous processing model [15] plays a critical role in this situation, with the message queue serving as a mediator between services that generate and consume processing tasks.

In the context of a web application, the producer refers to the PHP client application that generates messages based on user actions. For example, when a user performs an action that necessitates storing data in a database, the PHP application produces a corresponding message. In contrast, the consumer is embodied by a daemon process, like Celery, that consumes these messages and carries out the required database tasks with the help of Selenium. This integration facilitates seamless communication between PHP and Python, transmitting messages through RabbitMQ and using Celery with Selenium for web crawling and data storage in the database [8].

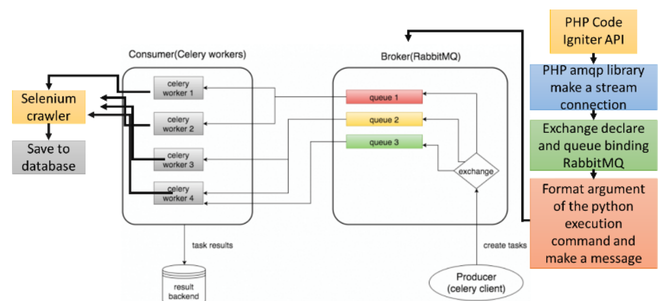

Fig. 1. PHP-Python Celery, RabbitMQ, and Selenium Architecture

Fig. 1 explains the complete system architecture. System information was made with Code Igniter (CI) as a PHP framework. CI application then defines a function to create a stream connection to RabbitMQ with the AMQP library. The queue needs to be bound in the channel declared with the exchange channel of the RabbitMQ. Some JSON format argument needs to be sent and could be executed as the channel message. JSON format contains information about the specific argument that needed to be performed in Python. After the message is sent to RabbitMQ, RabbitMQ can detect whether or not Celery is running. If the Celery is running, the worker executes the Python command. In this case, The Python command is to start crawling using Selenium. After the data from Selenium is get, then is formatted at the data frame, the data is saved in the database.

B. Algorithm and Environment Setup

Firstly, during the initialization phase, the installations of PHP, Python, RabbitMQ, and Celery must be performed correctly. A specific format of message and mechanism to run the Celery and the main program is also needed. There is a dependency on a PHP library such as AMQP to send the message to the channel. There is another dependency on Python libraries, such as Selenium and Celery. Verifying that RabbitMQ is properly configured and running is crucial to ensure communication between the different components is run on the system.

Next, in creating the PHP interface, there is a function from the controller of the Code Igniter that could handle the AMQP message and send it to the Celery. The view interface of a web application is just like a button with a specific parameter like date selection or a particular ID that must be crawled. This PHP web page with a button element becomes a gateway to receive user requests to initiate the crawling process. Because there are some users in that system, it is necessary to design the queue mechanism to ensure the crawler is queued and can be executed when they get to the queue.

This crawling mechanism could be done concurrently if some users click that button together. There is a parameter like a worker in Celery to handle this simultaneous action. This parameter is tested in this research to see the effect of increasing the number of workers when there is some request. This mechanism is shown in Figure 2 in the PHP box and continues to the RabbitMQ box, interconnected with the Celery box. This system has two main Python functions: a long process for executing the crawler based on the range of date criteria and one process for executing the crawler with a specific ID. There is also a creation of a RabbitMQ consumer using the AMQP library in Python to receive the messages PHP sends.

Celery is used to dispatch crawling tasks to the pre-configured Celery workers. After completing the crawling task, the Celery workers should return the results to RabbitMQ using the AMQP protocol. This mechanism to get the result back does not always work. The result could not be returned without specific conditions, like how long the crawler must take to get the parsed data. The solution is that the status of the crawling process is saved to the database.

The Selenium box in Fig. 2 shows the crawling process. The mechanism to access and click the element in the browser could be seen in the development phase because Selenium runs under Windows or some operating system with GUI. Selenium must be headless in the production phase to run the process in the server's background. Some parameters are carried from the argument, like the year of the data and the specific ID needed to be extracted. After the year is selected and the particular ID is inputted into the search button, the parsing element function extracts and saves the data to the database.

There is a mathematical formulation that exists in the problem of web crawling. There is a page that needs to be crawled, given by P_i, over a connection of bandwidth B. Trivial solution, we download all the pages simultaneously at a speed proportional to the size of each page [16].

$$B_i = \frac{P_i}{T^*} \tag{1}$$

T^* is the optimal time to use all the available bandwidth:

$$T^* = \frac{\Sigma P_i}{B} \tag{2}$$

$$b_i = \frac{N}{T} \tag{3}$$

b_i is the efficiency rate or the jobs the distributed crawler could do. N is the amount of the distributed crawler. The higher the amount of the crawler, the higher the efficiency. T is the total time of the distributed crawler to finish the task.

We could extend the formula from the basic equation according to the number of pages to be downloaded and the request per connection. P(s,i) is the number of pages downloaded over server s with i-th connection, and T(s,i) is the time to download them [17].

$$R(s,i) = \frac{P(s,i)}{T(s,i)} \tag{4}$$

P(s,i) and *R(s,i)* could only be known after downloading pages over the connection. Concurrent connection also uses a scheduling concept to estimate value a priori. The performance of server and network conditions could be changed constantly, although we keep the condition of the testing scenario. We could derive the formula to:

$$P(s,i) = \min\{\text{RequestPerConnection}(s,i), \text{TaskURLs}(s,i)\} \tag{5}$$

$$T(s,i) = 2 * \text{ConnectionTime}(s,i) + \frac{P(s,i) * ResponseTime(s,i)}{p} \tag{6}$$

P(s,i) is a set of two components: RequestPerConnection and TaskURLs. RequestPerConnection represents the number of URL requests that server *s* handles per connection, while TaskURLs indicate the size of the URL queue on server *s*.

The function *T(s,i)* comprises two parts: the time required to establish and terminate the connection (2 * *ConnectionTime(s,i)*) and the cumulative transfer time needed to retrieve the content of the web pages *P(s,i)*. Variable *p* represents a saving factor applied to this component and data on the web page. Connection time is adaptively estimated in the following manner:

$$ConnectionTime(s,i) = ConnectionTime(s,i-1) * \alpha + MeasureConnectionTime(s,i-1) * (1 - \alpha) \tag{7}$$

$$ResponseTime(s,i) = ResponseTime(s,i-1) * \alpha + MeasureResponseTime(s,i-1) * (1 - \alpha) \tag{8}$$

ConnectionTime(*s, i-1*) and *ResponseTime*(*s, i-1*) are the original estimations, and *MeasureConnectionTime* and *MeasureResponseTime* are the currently applied concurrent observations. Using the new estimation based on the testing mechanism, we could set the parameter α at 0.95 obtained from the comparison between two workers and four concurrency and four workers and eight concurrency to smooth the estimation. Performance-based scheduling: the server queue is ordered by each server's *R(s,i)*, while URLs are ordered as FIFO. This research contributes to a lower response time, so the performance *R(s,i)* could increase.

Based on the crawl-ability-based scheduling, there a performance and quality factors that could be considered as follows:

$$C(s,i) = \frac{AQ(s,i)}{T(s,i)} \tag{9}$$

$$AQ(s,i) = \sum_{p=1}^{P(s,i)} Quality(Entry_p_in_URL_queue) \tag{10}$$

C(s,i) shows the time-average page quality of the crawler needed to extract the content from server s. High crawl ability indicates the server could do the crawling job fast with the variation of components of the HTML. This research maintains that the crawler could do a job concurrent with another crawling task, increasing the quality factor based on the page's entry in the queue.

Finally, appropriate error handling management throughout the crawling procedure was considered. This error handling encompasses the careful handling of timeouts, connection errors, and any potential errors originating from the crawled websites. Furthermore, establishing and integrating suitable monitoring and error reporting mechanisms could effectively address potential issues and continuous monitoring. This research tries to make a testing mechanism to ensure that the crawler can be concurrently executed with the load variation.

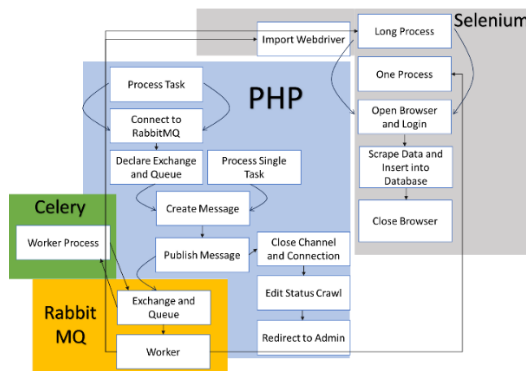

Fig. 2. PHP-Python Celery, RabbitMQ, and Selenium Process Flow

Fig. 2 shows the complete algorithm process that is done inside the system. The worker process in Celery could be run with "gevent" library to check the event when the crawler is running. For the testing scenario, a tricky step is that the program must be run in the Celery mode and run the function through the standard Python execution. Fig. 3 shows the primary crawling function in interconnection with RabbitMQ, Selenium, and Celery, explained in Pseudocode.

This methodology provides a practical framework for enhancing the integration between PHP and Python in concurrent crawling using AMQP, Selenium, Celery, and RabbitMQ. The implementation can be customized based on specific needs and developer preferences.

IV. Experiment and Result

This experiment aims to show that the integration process works between PHP and Python for concurrent crawling tasks using AMQP, Selenium, Celery, and RabbitMQ. The script required main libraries, such as Celery and Selenium. The data formatting and parsing library use "json", "re", "pandas", "csv", and "numpy". There is also a support library like "threading", "psutil", "os", and "logging", which is used to test memory and CPU usage parallel with the crawling process. The libraries facilitate the crawling process's concurrent task and extract data from a targeted website. This experiment also evaluates the system's performance by adjusting the number of workers and concurrency levels.

A MySQL database was also set up with a dedicated database to store the crawled data. A RabbitMQ server was configured to run locally on the designated host machine to establish communication and task distribution. This setup ensured that all the necessary components were in place to execute the experiment successfully.

```
FUNCTION example_task():
  BEGIN
        Enter_Rabbit_MQ_Queue(selenium_task_to_be_cra
wled)
        SET start_time TO time.time()
        FOR i=0 TO selenium_task_queued DO
        Begin_Celery_Session
        SET chromeOptions TO Options()
  SET browser TO webdriver.Chrome(chromeOptions)

Start_Crawling('https://simakbmu.undip.ac.id/auth/user/login')
        Find_Login_Element_and_Fill_Login_Information('us
ername','password')
        Find_DataTable_Element()
        IF User_Selection IS Year THEN
        SET maximal_pagination TO
Get_Maximal_Pagination_Page()
        FOR i=0 TO maximal_pagination DO
        FOR i=0 TO supply_line DO
        Find_Element_Needed_to_be_Inserted_in_Database
        Insert_to_MySQL
        FOR i=0 TO supply_detail_line DO
        Find_Element_Needed_to_be_Inserted_in_Database
        Insert_to_MySQL
        END FOR
        END FOR
        END FOR
  SET end_time TO time.time()
  SET runtime TO end_time - start_time
        Response_Time_Calculation()
  RETURN {"result": "Task_Finished", "runtime": runtime}
        END FOR
```

Fig. 3. Pseudocode of main crawling process

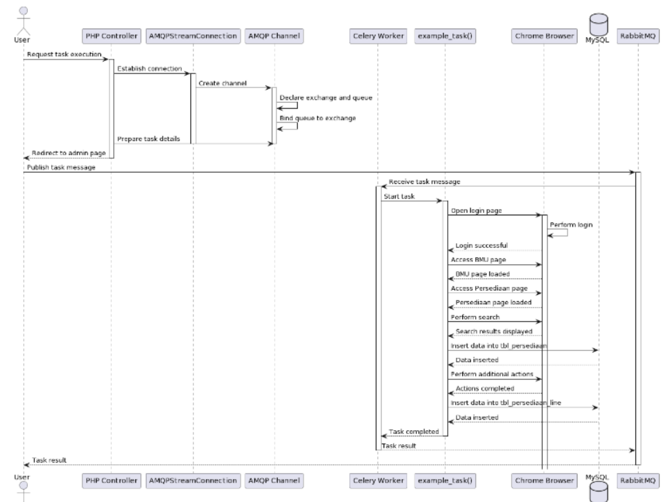

Fig. 4. Sequence diagram between PHP, AMQP, RabbitMQ, Celery, Chrome Browser, and user

The sequence diagram in Fig. 4 illustrates the user, PHP controller, RabbitMQ, Celery worker, browser, and database interaction in the task execution process. When the user requests task execution, the PHP controller establishes a connection with RabbitMQ and creates a channel. It prepares the task details and publishes the task message to RabbitMQ. The Celery worker receives the message, which initiates the task. The worker interacts with the browser to perform various actions, such as logging in, accessing specific pages, and completing tasks. It also interacts with the database to insert

data into the appropriate tables. Once the job is completed, the result is sent back through RabbitMQ to the user. Overall, this sequence diagram provides a concise overview of the communication flow and collaboration between the components executing the task using PHP, RabbitMQ, and Celery.

The experiment was carried out by running crawling tasks with different configurations. Three scenarios were tested, involving varying numbers of workers and concurrency levels. Each arrangement involved a predetermined number of workers. The system's performance was observed through testing as the number of workers and concurrency levels changed. The experiment recorded the memory and CPU usage during the execution of the crawling tasks for each configuration.

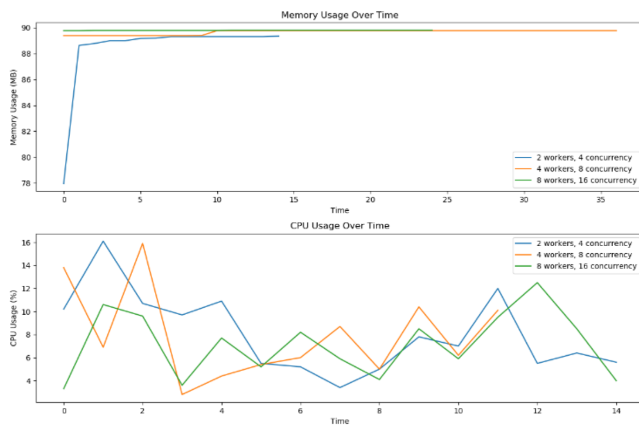

Fig. 5. Memory and CPU Usage Comparison between the Variation of Workers and Concurrency

Fig. 5 compares memory and CPU usage across different variations of workers and concurrency. In the initial testing scenario with two workers and four concurrency conditions, the memory usage increased from 78 MB to 88 MB, indicating a difference of 10 MB. Afterwards, when using two workers, the memory usage was approximately 89.31 MB, representing an increase of 1.31 MB compared to the baseline. Scaling up to 4 workers resulted in memory usage of 89.77 MB, reflecting a difference of 1.46 MB from the 2-worker configuration. Finally, with eight workers, the memory usage slightly increased to 89.81 MB, indicating a difference of 0.04 MB compared to the 4-worker setup.

The CPU usage showed peak utilization during the initiation stage. However, when navigating to the page where the data exists is performed, the CPU usage decreases. Once the process of extracting data begins, there is an increase in CPU activity. It is important to note that the monitoring of CPU activity is not evenly distributed across time but rather follows a discernible pattern. Specifically, when using eight workers, the CPU utilization experiences an increase in certain states due to the corresponding increase in concurrency. Table 1 shows the comparison of response time and throughput in each variation.

Firstly, two tasks were completed with a throughput of 0.08 tasks/second in the testing with two workers and four concurrencies. The average response time was 23.34 seconds, indicating efficient task completion within a reasonable time frame.

Secondly, in the testing with four workers and eight concurrencies, two tasks were also completed with the same throughput of 0.08 tasks/second. The average response time was 23.47 seconds, slightly increased compared to the two workers and concurrency four scenarios.

TABLE I. COMPARISON OF THE RESPONSE TIME AND THROUGHPUT FOR WORKER AND CONCURRENCY VARIATION

Variation	Response Time	Throughput	Tasks
Two workers, four concurrencies	23.34	0.08	2
Four workers, eight concurrencies	23.47	0.08	2
Eight workers, 16 concurrencies	24.26	0.08	2

However, despite the increase in the number of workers and concurrency, the difference in average response time between the two workers with four concurrency and four workers with eight concurrency scenarios is relatively small. This event shows that the concurrency must balance with the number of workers. The increase of workers in a rationable manner is not increasing memory exponentially. This result shows that other factors, such as network conditions or resource availability, could influence response time. Further analysis and testing may be required to fully understand the relationship between the number of workers, concurrency, and their impact on the average response time in the system.

V. CONCLUSION

The research on enhancing PHP-Python integration for concurrent crawling using AMQP, Selenium, Celery, and RabbitMQ has provided valuable insights into the system's performance. Implementing these technologies has improved efficiency and performance compared to traditional sequential crawling mechanisms.

One of the main performance metrics analyzed in the research was the average response time. By leveraging the concurrency and parallelism capabilities offered by Celery and RabbitMQ, the system achieved significant reductions in response time compared to sequential crawling. Utilizing multiple workers and concurrent processing led to faster task completion and enhanced system responsiveness.

Furthermore, the research also looks at resource utilization, including CPU and memory usage. Celery and RabbitMQ facilitated the effective allocation of CPU resources by distributing tasks across multiple workers. This distribution not only reduced overall processing time but also prevented CPU bottlenecks. Moreover, the research showcased efficient memory management, ensuring optimal usage throughout the crawling process.

The significant reduction in average response time indicated faster task completion and an improved user experience. Additionally, utilizing resources such as CPU and memory demonstrated improved efficiency, resulting in better overall performance and scalability.

The research findings demonstrate that leveraging AMQP, Selenium, Celery, and RabbitMQ in PHP-Python integration for concurrent crawling enhances system performance. The reduced response time, efficient resource utilization, and improved scalability contribute to the effectiveness and efficiency of the crawling system, opening up possibilities for various web crawling applications.

ACKNOWLEDGEMENT

This work is supported and fully funded by Electrical Engineering, Diponegoro University.

REFERENCES

[1] D. S. Sand, "A Framework for Scalable Web Data Collection,", Course Final Project, Control and Automation Engineering of the Federal University of Santa Catarina, 2022.
[2] A. Yudidharma, N. Nathaniel, T. N. Gimli, S. Achmad, and A. Kurniawan, "A systematic literature review: Messaging protocols and electronic platforms used in the internet of things for the purpose of building smart homes," *Procedia Comput. Sci.*, vol. 216, pp. 194–203, 2023, doi: 10.1016/j.procs.2022.12.127.
[3] L. Suutari and J. Holappa, "Future Proofing Lovelace System Development Environment", Bachelor's Thesis, Degree Programme in Computer Science and Engineering, University of Oulu, 2022.
[4] V. Thirupathi and K. Sagar, "Web of Things an intelligent approach to solve interoperability issues of Internet of Things communication protocols," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 981, no. 3, p. 032094, Dec. 2020, doi: 10.1088/1757-899X/981/3/032094.
[5] "Open-source log management software, Syslog, AMQP, NoSQL, Syslog server, Distributed database Syslog server, Cassandra database Syslog application, NXLog, Apache, PHP", American Journal of Computer Architecture 2022, vol. 9, no. 1, p.1-7, Apr. 2022.
[6] M. Kaasila and M. Pennanen, "Decoupling Between Lovelace's Checker Server and Main Server", Bachelor's Thesis, Degree Programme in Computer Science and Engineering, University of Oulu, 2022.
[7] T. Baier, S. B. Santamaria, and P. Vossen, "A modular architecture for creating multimodal agents." arXiv, Jun. 01, 2022. Accessed: Jun. 11, 2023. [Online]. Available: http://arxiv.org/abs/2206.00636
[8] N. Smirnov, S. Tschernuth, W. Morales-Alvarez, and C. Olaverri-Monreal, "Interaction of Autonomous and Manually-Controlled Vehicles:Implementation of a Road User Communication Service." arXiv, Apr. 28, 2022. Accessed: Jun. 11, 2023. [Online]. Available: http://arxiv.org/abs/2204.13643
[9] J. Munke *et al.*, "Data System and Data Management in a Federation of HPC/Cloud Centers," in *HPC, Big Data, and AI Convergence Towards Exascale*, 1st ed.New York: CRC Press, 2022, pp. 59–80. doi: 10.1201/9781003176664-4.
[10] J. Maxant, R. Braun, M. Caspard, and S. Clandillon, "ExtractEO, a Pipeline for Disaster Extent Mapping in the Context of Emergency Management," *Remote Sens.*, vol. 14, no. 20, p. 5253, Oct. 2022, doi: 10.3390/rs14205253.
[11] V. Skorpil and V. Oujezsky, "Parallel Genetic Algorithms' Implementation Using a Scalable Concurrent Operation in Python," *Sensors*, vol. 22, no. 6, p. 2389, Mar. 2022, doi: 10.3390/s22062389.
[12] G. Sun, H. Xiang, and S. Li, "On Multi-Thread Crawler Optimization for Scalable Text Searching," *J. Big Data*, vol. 1, no. 2, pp. 89–106, 2019, doi: 10.32604/jbd.2019.07235.
[13] Z. Wang, "Web Crawler Scheduler Based on Coroutine," in *2019 International Conference on Intelligent Computing, Automation and Systems (ICICAS)*, Chongqing, China: IEEE, Dec. 2019, pp. 540–543. doi: 10.1109/ICICAS48597.2019.00118.
[14] S. Lenselink, "Concurrent Multi-browser Crawling of Ajax-based Web Applications", Master's Thesis, Degree of Master of Science in Computer Science, TU Deflt, 2010.
[15] C. Auer, M. Dolfi, A. Carvalho, C. B. Ramis, and P. W. J. Staar, "Delivering Document Conversion as a Cloud Service with High Throughput and Responsiveness," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, Jul. 2022, pp. 363–373. doi: 10.1109/CLOUD55607.2022.00060.
[16] C. Castillo, M. Marin, and A. Rodriguez, "Scheduling algorithms for web crawling," in *WebMedia and LA-Web, 2004. Proceedings*, Ribeirao Preto-SP, Brazil: IEEE, 2004, pp. 10–17. doi: 10.1109/WEBMED.2004.1348139.
[17] F. Cao, D. Jiang, and J. P. Singh, "Scheduling Web Crawl for Better Performance and Quality", Princenton University, USA, Technical Report TR-682-03, Sep. 2023.