Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 05, May-2020

Horizontally Scalable Web Crawler using Containerization and a Graphical user Interface

Ansuman Prusty, Oscar Mejia, Aniket Shah, Pavan Kancherlapalli, Aditya Suresh, Roland Schiebel
Harvard University

Abstract:- Web crawlers (spiders), that index pages on the web and documents on the internet, have been around almost since the beginning of the internet. As the internet has been growing very rapidly over the last 25 years, it is impossible for "single" crawlers to efficiently crawl and index the web. There are a lot more users on the internet now, specifically a large number of non-technical users, who would like to crawl the web but cannot do so programmatically. Our application addresses these concerns as it is a horizontally scalable web crawler that uses containerization (Docker and Kubernetes) for scalability and performance, and can be controlled conveniently using a web-based user interface. Our solution focuses on having separate containerized Crawler-Manager instances for each incoming user request to the application, instead of having one centralized "manager" instance. It supports machine learning to decrease the amount of storage needed by only storing content that is classified in accordance to user defined class labels. We also analyzed existing parallel crawling approaches and how they compare to our solution.

General Terms:- Web Crawling

Keywords:- Container, Docker, Kubernetes, Scalability, Web Crawler

1 INTRODUCTION

Web Crawlers (aka spiders or bots) are applications that can index web pages and documents on the internet or any network. They maintain a collection of documents (usually in a database) for the purpose of searching them for information. This paper focuses on the web crawling rather than on the parsing, searching, or display of these documents.

Web crawlers have been around since the early 1994, when the first web crawler was developed by Brian Pinkerton [1]. Since then many different types of crawlers have been developed. Desai et al. (2017) characterize them as 6 different types: Breadth-first, Incremental, Focused, Hidden, Parallel and Distributed [2]. Although they mostly do the same thing -- crawl, index and store documents -- their focus and approach is quite different. Since this paper focuses on scalability and performance, we will look at some parallel and distributed crawlers. For example, a focused crawler, which accesses only a narrow section of the web (e.g. only medical documents or documents related to pharmaceuticals) can also be implemented as a parallel or distributed crawler. In the early days of the internet, when Pinkerton (1994) developed his crawler, the web was so small that scalability and parallel crawling was not a major concern. As the web grew, parallel crawlers were soon developed [3]. Nowadays it is almost impossible to scan the internet in a reasonable amount of time without using some type of parallel crawling.

In the Literature Review, previous work on parallel crawling is discussed, followed by our approach to parallelization using Docker containers and Kubernetes. Our web-based UI is also explained in greater detail. Although several good crawling libraries already exist (e.g. Python's scrapy library [4]), the more performant ones usually require a user to have some type of programming knowledge and don't provide a graphical user interface (GUI). Our product is a custom crawler with focus on horizontal scalability and performance along with a convenient user-interface for non-technical users.

2 LITERATURE REVIEW

The research by Cho and Garcia-Molina (2002) is one of the first comparisons of the parallel crawlers in the industry. They describe the main challenges of parallel crawling, as "Overlap" (different processes may download the same page twice), "Quality" (how to decide which page to prioritize in the URL frontier) and "Communication bandwidth" (communication overhead increases with the number of crawler processes). We are addressing some of these issues in our design. The communication overhead is reduced by having independent crawler-manager docker containers managing their own crawler instances and don't need to communicate with each other directly. The page quality is increased by using a priority queue to rank the pages to download.

Even Brin and Page (1998), who invented Google, were using some type of parallelism, because their indexer and sorter ran in parallel [5]. There has been a lot of research on scalable and distributed web crawling since then [6, 7, 8, 9, 11, 10, 11, 12].

"Mercator" by Haydon and Najork (1999) is one of the early scalable crawlers [6]. It is developed entirely in JAVA, making it platform independent, and it can be extended by adding modules to complete different crawling tasks. However, its scalability is vertical, meaning it scales up or down with available memory and processing power.

Edwards et al. (2001) developed a scalable, incremental crawler for IBM, that focuses on web page changes and tries to optimize the crawling using an adaptive approach based on the change rates of the data [7].

"UbiCrawler" by Boldi and Codenotti (2003) is another JAVA application. It uses multiple autonomous agents, each of which runs multiple threads, that can crawl different hosts using a breadth-first strategy [8].

Bal and Geetha (2016) came up with a focused and distributed web crawler using a breadth-first and best-first approach [9]. They use a dynamic URL assignment strategy to evenly balance the loads between crawlers. The fact that they're using a central data repository and each crawler has

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 05, May-2020

its own local memory makes it somewhat similar to our approach. However, they are also only using one crawl master to manage many crawlers, which creates a single point of failure.

DCrawler is another scalable, fully distributed and platform-independent web crawler written in JAVA [10]. Kumar and Neelima (2011) focus on fault tolerance and autonomy of their crawler agents, which coordinate their crawling behavior with each other, making it a practical and usable approach.

Patidar and Ambasth (2016) propose a fairly complex crawler architecture with 3 management services (cluster manager, bot manager and child manager) to handle the added complexity and the challenges to I/O and network efficiency, robustness, system design and manageability, that all come with a high-performance incremental crawling system [11]. Their design uses individual MongoDB instances per manager service, which is similar to the way we utilize Redis in the crawler-manager as in-memory cache. Although MongoDB probably has a shallower learning curve and simpler API, Redis is much faster.

Jaganathan and Karthikeyan (2015), on the other hand, focused on efficiency and content quality, when they developed their scalable, focused, incremental crawler [12]. They use numerous hardware factors (e.g. CPU, memory or disk capacity) to make decisions on efficient URL distribution between crawler instances. For prioritizing links, they use a ranking model, forward link and backlink counts and the data source quality of the URL and other factors. This improves the content quality and freshness.

Chaulagain et al. (2017) developed a fully cloud-based web scraper for Big Data Applications [13] by using mostly Amazon Web Services like Elastic Compute Cloud and DynamoDB. This is an interesting approach, as it provides elastic resources, high performance and cost-efficiency.

Aside from the approaches above, there are also some web scraping services, that allow scraping of web pages without requiring any programming knowledge. Two of these are import.io [14] and webscraper.io [15]. They both offer the possibility to experience web scraping as a service (Software as a Service or SaaS) with the scraped data stored in the cloud. They provide different download formats for the data as well. However, although webscraper.io also comes as a free Google extension, their cloud-based services aren't free.

Of all the applications analyzed above, we can see that many of them provide scalability, some are fully distributed and most achieve platform-independence by being developed in JAVA. The SaaS applications mentioned above even provide the convenience of a web interface. We're not aware of any implementations, however, that provide all the features above and also provides the flexibility that comes with containerization, as well as being written in one simple and easy-to-learn language like Python.

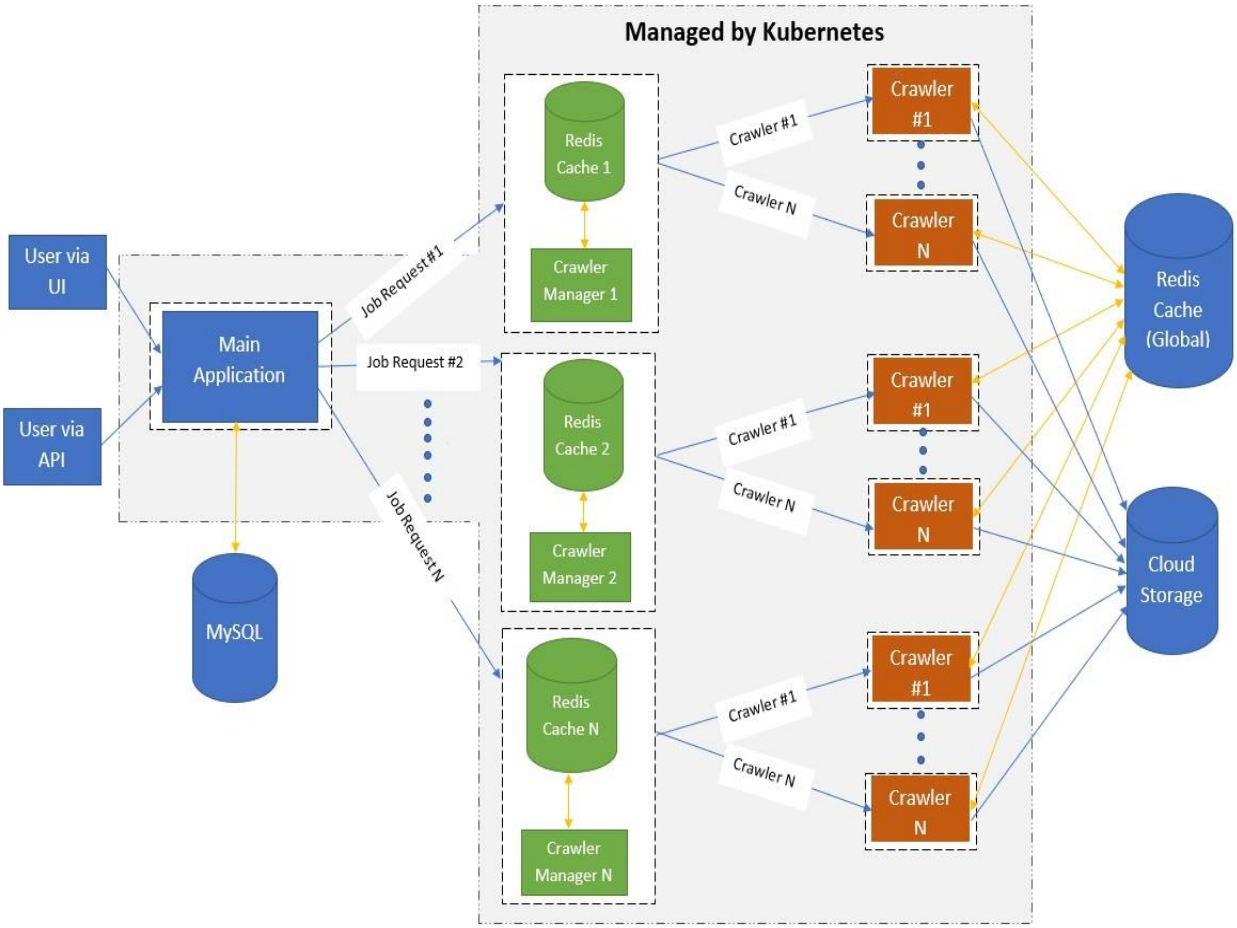

Figure 1: Our Architecture with 5 services - main app, crawl managers, crawlers, storage and cache

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 05, May-2020

3 METHODOLOGY

3.1 Architecture

Figure 1 above shows the architecture of our web crawler system. It consists of the main application, crawler manager and crawler instances. All instances run in separate docker containers, forming a Kubernetes cluster. For data storage, there is a MySQL database controlled by the main application, a central in-memory Redis cache, and cloud storage for downloaded documents. Our system utilizes Google Cloud Storage, but the application configuration can be changed to use storage systems from other cloud vendors, such as AWS S3.

Users can access the system both through a GUI and an API. A non-technical user can use the system via the graphical user interface (GUI), start crawl jobs and receive the results. More technical users can use the API to carry out the same functionalities but with the added flexibility of being able to script crawl jobs or trigger crawl requests.

The main application is written in Python, using the Django framework. It is connected to a MySQL database where user related information (e.g. login sessions) as well as information related to the individual crawl requests is stored. The MySQL database also stores some meta-data needed to access the crawl results (e.g. the exact cloud storage URL) but does not store the crawl results themselves. Those are stored in the cloud storage.

Our design provides horizontal scalability by having containerized and independent crawler-manager instances governing each crawl request separately. For every new crawl request coming from a user, a new crawler manager container is created. Each crawler manager will run in a Docker container and will use multiple crawler instances to fetch the pages. The number of crawler instances is provided by the user when the initial crawl request is created. Those crawler instances themselves will also run in separate containers. So, the user controls the initial scale (number of crawlers) at which the system will operate. Every crawler manager maintains its own queue to keep track of pending URLs and adding new child URLs.

Each crawler utilizes a number of parallel threads, which are not controlled by the user but are configurable in code. The crawler manager will keep sending URLs from its queue to its crawlers and the crawlers accept as many URLs as they have threads available. Once a crawler thread is done, it reports back to the crawler manager the results and is free to accept a new URL.

We use Kubernetes to handle and manage the entire cluster consisting of crawler-manager and crawler containers effectively. The crawlers themselves have access to the cloud storage to store the crawl results, which are the files themselves. The system only parses and scrapes content from HTML files but supports storing HTML, PDF and DOCX files in the cloud.

Aside from the central cloud storage, there is also a central Redis cache, that is utilized for in-memory caching and runs as a separate container. It is used by the crawler instances to check whether its assigned URL has been crawled within the last 24 hours. If not, the crawler will crawl the page by requesting it from the web server. If it was already crawled in the last 24h, it will retrieve the cloud storage link of the content currently stored in the cloud and not contact the web server, where it's originally hosted. This will reduce latency in our system and decrease network load for the web server hosting the domain.

The reason for this design with multiple containerized crawler managers as opposed to one central manager (which most previous implementations have used) is the reduced complexity compared to scaling with a single manager container. One of the challenges when using a single manager, is to efficiently scale the crawling service in relation to incoming traffic without affecting latency Another challenge is to also scale the database without affecting latency. This is avoided by having a separate crawler manager container per user crawl job request.

Our design with multiple crawler managers and independent crawlers prevents one job from affecting others and this allows for independent scaling requirements between different users without complex problems of rate limiting and timing control to provide fairness between users.

3.2 Technology Stack

Our goals for this project were portability and horizontal scalability, without tying the implementation to any specific cloud provider. Furthermore, we wanted to use only freely available and open-source technology for the implementation. Given these design choices, the following technology stack was finalized:

- **Python**, as a universally available development language was the preferred language for our customer.
- **Django** framework for developing the user interface and the main application. Django is fast, stable and well equipped for running web applications on high loads.
- **MySQL**, managed by Django, as the central database for storing metadata.
- **Redis**, as an in-memory cache within each crawler manager, and also as a separate container for storing links that have been crawled over the past 24 hours.
- **Google Cloud Storage** as a file storage to store all the crawled files.
- **Docker and Kubernetes** to provide scalability by containerizing crawler-managers crawler instances.

3.3 Main Application and GUI

The main application is written in Django, which is a Python framework for web applications. It controls and manages the data in the MySQL database, as well as all the forms and HTML templates that make up the GUI and also manages the API endpoints. The MySQL database stores all the information about a crawl request and the user session information.

Django already supports a sophisticated admin module, which allows GUI admin access for adding new application users and their permission to the system. After a new user was added to the system, they can log in via the web crawler GUI to create new crawl requests. Each user can only access crawl request data related to their account and won't be able to interfere with other user's data.

The web crawler GUI consists of several screens, which let users look at their existing crawl requests as well as create new requests.

Figures 2 shows the screen for creating a new crawl job. A user provides a list of URLs (seed URLs) or the entire

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 05, May-2020

domain to crawl. Users can also set different crawl request parameters, which includes the document types to store, the machine learning model, the crawling library and the number of crawlers to use. Creating the crawl job will immediately start the crawling process.

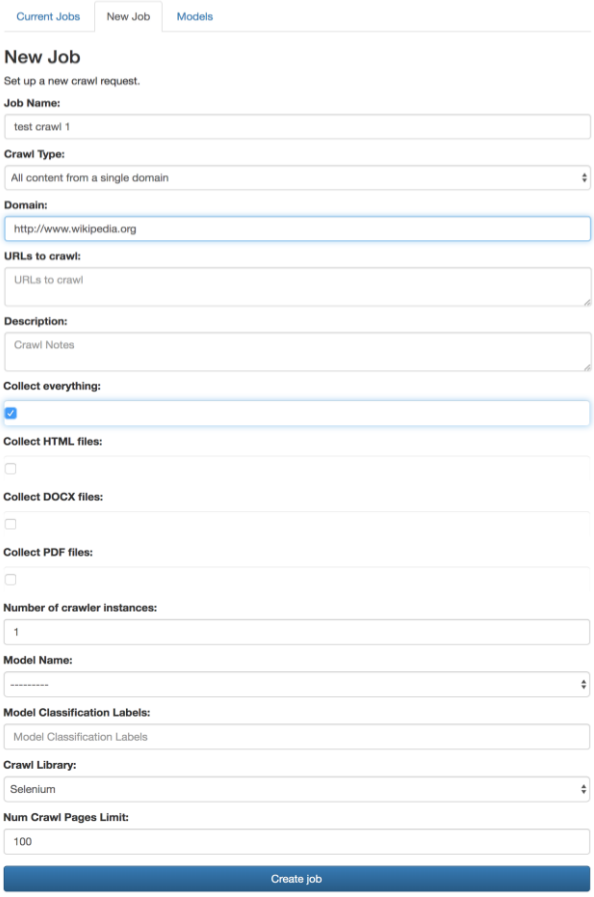

Figure 2: Create crawl request and set parameters

Figure 3 shows the screen for displaying current crawl jobs. A user can access the details for each of these requests (by clicking on them) as well as the crawl results and the manifest file after a crawl is finished. The manifest file holds the URL to the cloud storage location of the crawl results. The crawl status shows a user when a job is finished or if it has failed.

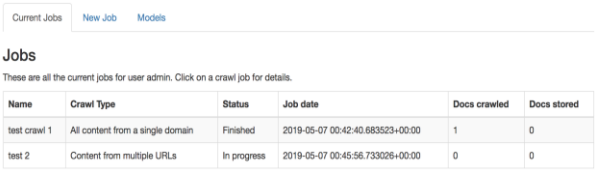

Figure 3: Current user specific crawl requests

Figure 4 shows the model upload screen. A user can upload a trained machine learning model (MLM) that allows the crawler to classify the crawled resource and decide whether that resource should be stored or dropped. 3.6 describes how the MLM works in more detail.

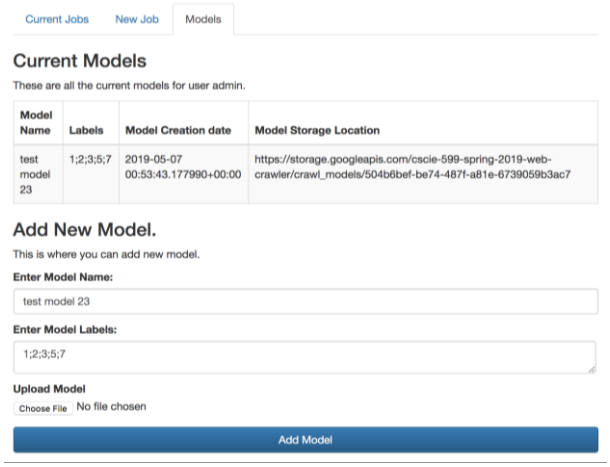

Figure 4: Upload machine learning model

Another way of interacting with the web crawler is via the Application Programming Interface (API). This can be used by a power user, which is a regular user with API permissions. The web crawler supports two different types of users, a regular user and a power user. Both of them can carry out the same functionalities but the API is for users who would like to control the crawler from a command line or from within other programs. The API communication uses JSON. Authentication for API users is achieved using JSON Web Token (JWT).

3.4 Crawler Managers and Crawlers

As we saw in Figure 1, the crawler managers and crawlers run in separate containers. A crawler manager container has two processes running: the manager application and the local Redis server. Crawlers, however, communicate with the centralized Redis server which runs in a separate container.

The following is the sequence followed by the system when initiating a crawl request from the perspective of the crawler managers and crawlers.

1. As each crawler manager container starts up, it registers with the main application and provides its IP address and port. In a similar way, when a crawler is created by a crawler manager, it also registers with its crawler manager.
2. Each crawler manager receives from the main application a domain and a list of URLs to crawl and the number of crawler instances a user would like the crawler manager to use. The manager then adds those URLs to a queue along with a set of crawler endpoints of the crawlers it created based on the number received from the user. A crawler endpoint identifies the specific URL, that is needed to communicate with that crawler.
3. If a machine learning model was selected by the user, the selected model is downloaded from cloud storage and cached within each crawler during its startup. More details on machine learning models can be found in 3.6
4. The crawler manager pops URLs from the queue and passes it to the crawlers in a round-robin manner via POST requests.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 05, May-2020

5. If a crawler is free and accepts the URL, it sends back an acceptance message. If it is busy and has no free threads to handle the request, it replies with a rejection message. The crawler manager adds those URLs for which it received a rejection message, back into the queue.
6. The crawler checks in the central Redis cache if the URL has already been processed. If not, the crawler downloads the web page related to the URL. If the crawler was provided with a machine learning model, it uses that to decide whether to store or drop the page, based on its classification and the class labels requested by the user. The content of the page is stored into a preconfigured cloud storage. The cloud storage in return provides a pre-signed URI.
7. The crawler stores the URL crawled and cloud URI in the centralized Redis cache as a key value pair.
8. The crawler extracts all the child URLs (HREF elements in the web page) and passes them to the crawler manager along with the cloud URI.
9. The crawler manager checks its local Redis cache and compares which ones of the child URLs received from the crawlers are not in the cache. Those that are not in the cache and not restricted by the domain, are added to the queue. This is done to eliminate URL duplicate processing. The above process continues until the queue is empty.
10. The crawler manager's local Redis cache is organized as a collection of key-value pairs, where each crawled URL is a key and the corresponding cloud location URL is the value. This collection of key-value pairs is specific to each crawl request and is referred to as the manifest file. The crawler manager then stores that manifest file in the cloud storage.
11. The crawler manager sends a crawl complete message back to the main application along with the cloud storage location of the manifest file.

3.5 URL prioritization

Our solution uses the "heapq" Python library to build a priority queue for prioritizing URLs to crawl. Each crawler manager maintains a separate priority queue, since all crawl jobs are independent. The queue contains all the URLs encountered by the crawlers managed by this crawler manager, including a score (i.e. counter) for each URL. The counter gets updated every time the same URL is found in one of the lists returned by the crawlers. The URL with the highest score gets taken from the queue when assigning the next URL to crawl. This approach increases the probability that the most significant URLs are downloaded and crawled first.

3.6 Machine learning model

Our solution also supports using trained Machine Learning Models (MLM) to decide on storing pages only when they satisfy certain conditions. A machine learning model is a serialized Python object file. The training of the model happens outside of the application, in advance. There are some comprehensive datasets that can be used for training. The models used in our approach were trained with the 20 Newsgroups dataset using a Random Forest algorithm but other models using Logistic Regression or Decision Tree algorithms work as well.

The classification model's outputs are integers (also referred as labels), ranging from 0-N where each integer corresponds to a category. For example, the user can train the Random Forest model on a newsgroup dataset and the output of the model can be one of the 4 integers/labels (0,1,2,3). Label 0 corresponds to category 'Politics', 1 to 'History', 2 to 'Geography' and 3 to the rest. The user has to have some knowledge of this mapping between the model's output labels/integers and the corresponding categories.

During the MLM upload, the user provides those numerical identifiers (e.g. 1, 2, 3, 5, 7) of all the labels that the MLM algorithm can classify. The MLM is stored in the cloud storage and a reference link to that storage is stored in the MySQL database in the main application.

When a new crawl request is created, a user can select one of the available, uploaded models including a subset of the label identifiers provided during the upload (e.g. 1, 3, 5). The crawler then gets the trained model file from the cloud storage and works with that model for this crawl request. The web crawler will apply the MLM to the crawled pages and only store the content of those pages, whose classification output matches that subset of labels.

Before applying the MLM to the crawled pages, the crawler converts the textual page content to a "Term Frequency Inverse Document Frequency" or TF-IDF matrix, which the model can work with.

3.7 Crawling dynamic content

For a static web crawler, it is sufficient to use the Python "requests" library along with BeautifulSoup [16] to crawl web pages. However, since so many web pages have dynamically created content these days, it is no longer sufficient to only look at static pages. Our solution is able to render the Javascript on a page and also obtain the URLs that are dynamically created.

We analyzed 3 different options for rendering Javascript before deciding on one.

3.7.1 PyQT5

PyQT is a Python wrapper around the QT framework for creating graphical user interfaces. PyQt Webkit acts as a browser, that can be used to render dynamic content. This is an approach that could be used but very few references are available on how to use it for scraping. Some information on PyQT4 was found but much has changed from PyQT4 to PyQT5. The lack of sufficient documentation in addition to the complexity of the QT framework would have resulted in too steep of a learning curve, given the time constraints.

3.7.2 Splash

This is a very light-weight service and considered to be very fast to render Javascript pages. In fact, the well-known scraper, Scrapy, uses this service as well. In most use cases, the service is run as a docker container, which is the recommended approach as mentioned in the Splash documentation [17].

Splash produced slightly better performance results than Selenium in isolated tests but also had several disadvantages. It would either require adding another central containerized service to our architecture or having Splash as

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 05, May-2020

a container inside a crawler container. The former would increase the complexity of our system and also add a single point of failure to the system, which is not a good design approach. Including a container inside the crawler containers would also add to the complexity and increase the crawler footprint, which is something we tried to avoid. In addition to that, Splash as a service, would require more network calls and slow down our system as a whole. Therefore, Splash was not chosen for our implementation.

3.7.3 Selenium

This is a well-known approach and widely used for rendering Javascript pages. There are many use cases for scraping using Selenium. Its versatility, customizability and extensive community support were a big plus. Another key factor is that Selenium is a library and doesn't run as a separate service, which won't increase the communication overhead in our architecture. Given the above benefits, our implementation uses the Selenium library along with the Chrome driver. The chrome driver is installed directly into the crawler container, making deployment simple. For the Chrome driver, we use the headless option to get the page content.

3.8 Discussion

3.8.1 Performance

Several tests were run on the Kubernetes cluster, while measuring the performance of the crawlers. The Python "requests" library was chosen for scraping and 4 concurrent threads per crawler. All content stored, no classification or file type filter set and no checking of the shared Redis cache.

Table 1. Processing time for different number of resources

Crawlers	Resources	Time in s
1	100	89
1	500	393
1	1000	537
5	1000	217

This shows the entire processing time (crawling and downloading the HTML page). The time also includes the start time of the crawler containers in the cluster but not the crawler manager start time. This makes sense, because there is always only one crawler manager per user's crawl request. So, the crawler manager spin-up time is static and won't affect performance with scale.

For one crawler instance, the download time increases almost linearly as the number of resources increases between 100 to 500 and does even better than linear when comparing 500 to 1000 resources.

When looking at multiple crawlers, we see the effect of scalability. The effect isn't entirely linear, due to time lost starting up each crawler container and other coordination and communication tasks, like registering crawlers with crawler manager, that are not directly related to downloading content.

3.8.2 Design challenges

Including so many different technologies in one project with their interaction complexities, presented one of the biggest challenges. Some effects were the interaction and communication needed between docker containers and pods within Kubernetes and how to resolve these in an efficient way, without much overhead. Using independent containers for crawler-managers as well as crawlers, however, was one of the key elements to providing a multi-user environment with horizontal per user scalability.

Testing also presented some challenges due to the variety of technologies and environments, since local testing would not always be the same as testing on Kubernetes in Google cloud.

Other challenges were related to the implementation of certain performance relevant features like URL prioritization and for example the machine learning functionalities. One thing that would be done differently, would be to include Machine Learning as a separate service in the design rather than adding it into the crawling process. This would provide greater flexibility in selecting various ML models, and would make the application more decoupled.

4 CONCLUSIONS

The web crawler application has been designed and developed to be horizontally scalable and provides a convenient user interface. Docker and Kubernetes are used to containerize our crawler and crawler manager services. In contrast to using only one central manager instance, which is the usual approach many other crawler applications follow, our approach has made it possible to avoid some of the typical complexity problems that come with using one single crawler manager (e.g. latency issues, rate limitations and timing control between users). The GUI also allows non-technical users to easily use the web crawler productively, whereas many other performant crawlers require some programming knowledge.

The implementation also addresses some of the typical challenges that crawlers face, like communication bandwidth and quality issues by using independent crawler managers to reduce communication bandwidth and using URL prioritization and machine learning to increase the quality of the crawled resources.

This was accomplished by using only open-source technology, like Django, Docker, Kubernetes and Python, which will allow anyone using this project to easily understand the inner workings and add on new functionalities if needed.

5 FUTURE WORK

There are a few improvements that can be made going forward:

- Provide indexing capabilities and search functionalities on the crawled data (e.g. by using cloud services like for example AWS Elastic Search).
- Improve support for machine learning functionalities (e.g. support different classification models for different tasks) to help with URL prioritization and page relevance to further increase the overall quality of stored content.
- Support parsing of additional files like PDF and XML. PDF parsing in particular can be challenging.
- Add more features to the UI that help increase the usability (e.g. dynamic crawl status refresh).

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 05, May-2020

6 REFERENCES

[1] Pinkerton, B. (1994). Finding what people want: Experiences with the WebCrawler. In Proceedings of the First World Wide Web Conference, Geneva, Switzerland. Retrieved 4/8/2019, from https://web.archive.org/web/20010904075500/http://archive.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/pinkerton/WebCrawler.html

[2] Desai, K., Devulapalli, V., Agrawal, S., Kathiria, P., & Patel, A. (2017). Web Crawler : Review of Different Types of Web Crawler, Its Issues, Applications and Research Opportunities. International Journal of Advanced Research in Computer Science, 8(3), International Journal of Advanced Research in Computer Science, Mar 2017, Vol.8(3). Retrieved 4/8/2019, from https://search-proquest-com.ezp-prod1.hul.harvard.edu/docview/1901457429?accountid=11311&rfr_id=info%3Axri%2Fsid%3Aprimo

[3] Cho, J., & Garcia-Molina, H. (2002). "Parallel crawlers". Retrieved 2/20/2019, from http://oak.cs.ucla.edu/~cho/papers/cho-parallel.pdf

[4] Scrapy, https://scrapy.org/

[5] Brin, & Page. (1998). The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems, 30(1), 107-117. Retrieved 4/8/2019, from https://www-sciencedirect-com.ezp-prod1.hul.harvard.edu/science/article/pii/S016975529800110X

[6] Haydon, A., Najork, M. (1999). "Mercator: A scalable, extensible Web crawler". Retrieved 2/20/2019, from https://link-springer-com.ezp-prod1.hul.harvard.edu/content/pdf/10.1023%2FA%3A1019213109274.pdf

[7] Edwards, J., McCurley, K. S., and Tomlin, J. A. (2001). "An adaptive model for optimizing performance of an incremental web crawler". In Proceedings of the Tenth Conference on World Wide Web. pp. 106-113. Retrieved 2/20/2019, from http://www10.org/cdrom/papers/210/index.html

[8] Boldi, P., Codenotti, B., Santini, M., & Vigna, S. (2003). UbiCrawler: A scalable fully distributed Web crawler. Retrieved 4/8/19, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.4239&rep=rep1&type=pdf

[9] Bal, S., & Geetha, G. (2016). Smart distributed web crawler. 2016 International Conference on Information Communication and Embedded Systems (ICICES), 1-5. Retrieved 4/8/2019, from https://ieeexplore-ieee-org.ezp-prod1.hul.harvard.edu/document/7518893

[10] Kumar, M., & Neelima, P. (2011). Design and Implementation of Scalable, Fully Distributed Web Crawler for a Web Search Engine. International Journal of Computer Applications, 15(7), 8-13. Retrieved 4/8/2019, from https://www.ijcaonline.org/volume15/number7/pxc3872629.pdf

[11] Patidar, T., & Ambasth, A. (2016). Improvised Architecture for Distributed Web Crawling. International Journal of Computer Applications, 151(9), 14-20. Retrieved 4/8/2019, from https://pdfs.semanticscholar.org/715d/b1703884cd962759cc84c53b0d30a5cb437b.pdf

[12] Jaganathan, P., & Karthikeyan, T. (2015). Highly Efficient Architecture for Scalable Focused Crawling Using Incremental Parallel Web Crawler. Journal of Computer Science, 11(1), 120. Retrieved 4/8/19, from https://thescipub.com/pdf/10.3844/jcssp.2015.120.126

[13] Chaulagain, R., Pandey, S., Basnet, S., & Shakya, S. (2017). Cloud Based Web Scraping for Big Data Applications. 2017 IEEE International Conference on Smart Cloud (SmartCloud), 138-143. Retrieved 4/8/2019, from https://ieeexplore-ieee-org.ezp-prod1.hul.harvard.edu/document/8118431

[14] Import.io, https://www.import.io/

[15] WebScraper.io, https://www.webscraper.io/

[16] BeautifulSoup, https://beautiful-soup-4.readthedocs.io/en/latest/

[17] Splash, https://splash.readthedocs.io/en/stable/install.html