Evaluation of Web Crawler's Design

Evaluate the proposed design of the web crawler system based on the fulfillment of its non-functional requirements.

We'll cover the following

Reviewing design requirements

Let’s evaluate how our design meets the non-functional requirements of the proposed system.

Scalability

Our design states that scaling our system horizontally is vital. Therefore, the proposed design incorporates the following design choices to meet the scalability requirements:

- The system is scalable to handle the ever-increasing number of URLs. It includes all the required resources, including schedulers, web crawler

workers, HTML fetchers, extractors, and blob stores, which are added/removed on demand.

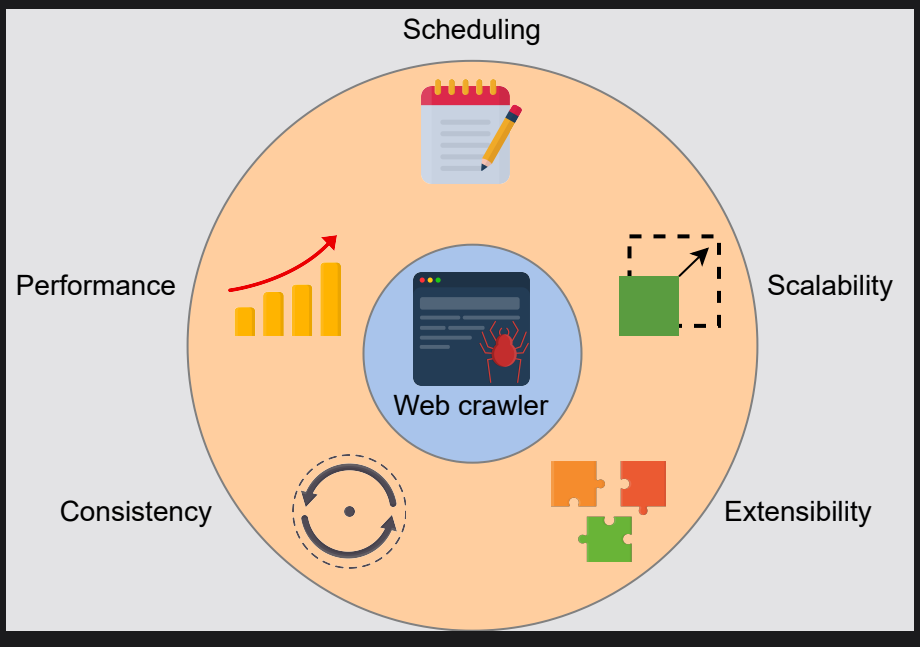
- In the case of a distributed URL frontier, the system utilizes consistent hashing to distribute the hostnames among various crawling workers, where each worker is running on a server. With this, adding or removing a crawler server isn't a problem.

Extensibility and modularity

So far, our design is only focusing on a particular type of communication protocol: HTTP. But according to our non-functional requirements, our system's design should facilitate the inclusion of other network communication protocols like FTP.

To achieve this extensibility, we only need to add additional modules for the newly required communication protocols in the HTML fetcher. The respective modules will then be responsible for making and maintaining the required communications with the host servers.

Along the same lines, we expect our design to extend its functionality for other MIME types as well. The modular approach for different MIME schemes facilitates this requirement. The worker will call the associated MIME's processing module to extract the content from the document stored in the DIS.



Consistency

Our system consists of several crawling workers. Data consistency among crawled content is crucial. So, to avoid data inconsistency and crawl duplication, our system computes the checksums of URLs and documents and compares them with the existing checksums of the URLs and documents in the *URL* and *document checksum* data stores, respectively.

Apart from deduplication, to ensure the data consistency by fault-tolerance conditions, all the servers can checkpoint their states to a backup service, such as Amazon S3 or an offline disk, regularly.

Performance

Our web crawler's performance depends on the following factors:

- **URLs crawled per second**: We can improve this factor by adding new workers to the system.
- **Utilizing blob storage for content storing**: This ensures higher throughput for the massive amount of unstructured data. It also indicates a fast retrieval of the stored content, because a single blob can support up to 500 requests per second.
- **Efficient implementation of the `robots.txt` file guideline**: We can implement this performance factor by having an application-layer logic of setting the highest precedence of `robots.txt` guidelines while crawling.
- **Self-throttling**: We can have various application-level checks to ensure that our web crawler doesn't hamper the performance of the website host servers by exhausting their resources.

Fulfilling Non-functional requirements

Requirement	Techniques
Scalability	• Addition/removal of different servers based on the inc • Consistent hashing to manage server's additic • Regular backup of the servers in Amazon S3 backup servic
Extensibility and Modularity	• Addition of a newer communication protocol module • Addition of new MIME schemes while processing the dow
Consistency	• Calculation and comparison of checksums of URLs and D data stores
Performance	• Increasing the number of workers performi • Blob stores for storing the conter • High priority to robots.txt file guidelines wh • Self-throttle at a domain while craw
Scheduling	• Pre-defined default recrawl frequenc • Separate queues and their associated frequencies fo

Scheduling

As was established previously, we may need to recrawl URLs on various

1. We can assign a default or a specific recrawling frequency to each URL. This assignment depends on the application of the URL defining the priority. A default frequency is assigned to the standard-priority URLs and a higher recrawl frequency is given to the higher-priority URLs.

   Based on each URL's associated recrawl frequency, we can decide to enqueue URLs in the priority queue from the scheduler's database. The priority defines the place of a URL in the queue.
2. The second method is to have separate queues for various priority URLs, u URLs from high-priority queues first, and subsequently move to the lower-

priority URLs.

Consider a real-time news aggregation platform, where content changes rapidly, and frequent recrawls are crucial for up-to-date information. What immediate steps would you take if the **IP** faces blocking issues during frequent crawls? Share strategies for handling IP blocks to ensure continuous access to the latest content.

Want to know the correct answer?

Share strategies for handling IP blocks

H1 H2 H3 | B I | | | | |

Provide your answer here!

Conclusion

The web crawler system entails a multi-worker design that uses a microservices architecture. Besides achieving the basic crawling functionality, our design provides insights into the potential shortcomings and challenges associated with our design and further rectifies them with appropriate design modifications. The noteworthy features of our design are as follows:

1. Identification and design modification for crawler traps
2. Extensibility of HTML fetching and content extraction modules