Name : Ahmed Khaled Abdelmaksod Ebrahim

1- What architectural style did you choose for the distributed web crawler (e.g., layered, event-driven, master-slave)? Why did you select this style, and how does it support the scalability and fault tolerance needed for the system?

I recommend **Microservices Architecture** as the architectural style. This approach divides the system into small, independent services that can be developed, deployed, and scaled independently. Each service in the system would handle a specific functionality of the web crawler, such as URL management, fetching, content processing, and deduplication. It supports scalability and fault tolerance as follows:

- **Scalability**:
  - Each microservice can scale independently based on the workload.
  - Services like **URL Fetcher** and **Content Processing** can handle high concurrency without bottlenecks, ensuring the system can crawl billions of URLs efficiently.
  - Load balancers distribute the workload across different instances of services for optimized performance.
- **Fault Tolerance**:
  - If a node or service fails, it does not crash the entire system. Faults can be isolated, and failed requests can be retried by a separate failure handling service.
  - Services are loosely coupled, ensuring that the failure of one service (like the **Headless Browser Service**) does not affect others like **URL Deduplication**.

2- Which design patterns would you use to:

- Manage communication and task distribution between different nodes in the system?
  - **Pattern**: **Observer Pattern**
  - **Why It's Suitable**:
    - The **Observer Pattern** allows for an event-driven architecture where different nodes (services) can act as observers to the task manager Whenever a new URL is available or a task needs distribution, the task manager (subject) notifies the relevant nodes (observers) to pick up tasks.
    - Each service can subscribe to updates and asynchronously receive tasks, allowing the system to distribute workloads across different nodes without tight coupling.

- Ensure that no URL is crawled more than once?
  - **Pattern**: **Singleton Pattern**
  - **Why It's Suitable**:
    - The **Singleton Pattern** ensures that only one instance of a deduplication service (or URL Frontier) is active across the entire system, making sure that all nodes have consistent access to the crawled URL data.

- Handle the retry and failure management process across multiple nodes?

    - **Pattern**: **Observer Pattern** or **Retry Pattern (via Decorator)**
    - **Why It's Suitable**:
        - **Observer Pattern** can also be used for failure handling, where nodes subscribe to events indicating failed tasks and retry the failed processes.
        - Alternatively, the **Decorator Pattern** could be applied to enhance the functionality of task handling methods, wrapping them with retry logic. If a task fails, the decorator would automatically trigger retries and log failures without modifying the underlying task-fetching methods

- Dynamically filter URLs and allow for flexible criteria?
    - **Pattern**: **Strategy Pattern**
    - **Why It's Suitable**:
        - The **Strategy Pattern** allows the system to dynamically select and apply different filtering rules at runtime. Each filtering criterion (e.g., domain filtering, keyword-based filtering, file type filtering) can be encapsulated in its own strategy class.
        - The system can decide which filtering strategy to apply based on the crawling context, enabling flexible and dynamic filtering of URLs without modifying the core logic of the crawler.

Key system component :

1. **URL Seed Service**:
    - **Responsibility**:
        - This service provides an initial list of URLs (seeds) to be crawled by the system. The seeds can come from an external system via a REST API or a predefined list of starting points for the crawler.
        - It passes the URLs to the **Load Balancer** for distribution.
    - **Tool**:
        - A REST API interface can be used to dynamically inject seeds into the system from various sources.
2. **Load Balancer**:
    - **Responsibility**:
        - Distributes tasks (URLs to crawl) across different nodes and services in the system to balance the load, ensuring that no single node is overwhelmed.
        - Ensures efficient utilization of the available resources by routing requests to less loaded services.
    - **Tool**:
        - Tools like **HAProxy** or **NGINX** can be used as load balancers.
3. **URL Frontier**:
    - **Responsibility**:
        - Acts as a task scheduler that manages the pool of URLs that need to be crawled. It prioritizes URLs based on certain criteria (e.g., domain, URL freshness, depth) and selects URLs to send to the **URL Fetcher** service.

        - Ensures the orderly flow of URLs and prevents overload on other services.
4. **URL Fetcher**:
    - **Responsibility**:
        - Responsible for fetching web pages and retrieving content based on the URLs it receives from the URL Frontier.
        - It includes **DNS resolution** and rate-limiting to prevent overloading target web servers.
    - Tool :
        - **Rate-limiting service** ensure the system doesn't overload web servers or get blocked.
5. **Content Processing Service**:
    - **Responsibility**:
        - Processes the fetched content, extracts metadata, and filters it based on predefined criteria (e.g., keywords, file types).
6. **Deduplication Service**:
    - **Responsibility**:
        - Ensures that no URL is crawled more than once by checking whether a URL has already been processed before fetching it.
        - It interacts with a cache to store and look up previously crawled URLs.
    - **Tool**:
        - **Redis** is ideal for maintaining a fast in-memory store of crawled URLs to quickly check for duplicates.
7. **URL Extractor**:
    - **Responsibility**:
        - After the content is processed, the URL Extractor scans the page for additional URLs to add to the URL Frontier for future crawling.

        - These URLs are passed back to the **URL Frontier** to continue the crawling process.
    - Tool :
        - Parser to extract the URL for recursive crawling.
8. **Failure Handling Service**:
    - **Responsibility**:
        - Monitors failed tasks and retries them. If a URL fetch fails (due to timeouts or server errors), this service will attempt to reschedule the task and redistribute it to other nodes.
    - Tool:
        - **Kafka** also handles failure recovery by republishing failed tasks to the queue for retries.
9. **Data Storage**:
    - **Responsibility**:
        - Stores the raw HTML content, metadata, and extracted information from the crawled pages.
        - Provides a searchable index for future data retrieval, allowing for efficient querying of the crawled data.
    - **Tools**:
        - **MongoDB** is chosen for its flexibility in storing web content.
        - **Elasticsearch** can be used for indexing and quickly searching the crawled web content based on keywords and metadata.

10. **Cache**:
    - **Responsibility**:
        - Caches frequently used data or recently fetched pages to reduce unnecessary fetch operations and speed up lookups.
        - Also stores the deduplication data (i.e., URLs that have already been crawled).
    - **Tool**:

- **Redis** is used as a fast, in-memory cache to store frequently accessed data and previously crawled URLs for deduplication

---

How will you ensure the system can scale efficiently as the number of URLs increases. Explain the strategies used for load balancing and resource distribution.

- The system can be scaled horizontally as the load increases.
- Microservices allow each component to scale independently based on demand.
- A load balancer distributes incoming requests to multiple nodes in a round-robin or based on their current load. This ensures that no single node is overwhelmed with too many requests at once.
- Elastic storage solutions (MongoDB, Elasticsearch) scale as data increases.
