# MULTITHREADED WEB CRAWLER

PROJECT REPORT

By

**GAURI GUPTA (RA2211026010359)**

**NEELANSH BHARGAVA (RA2211026010360)**

**MRINALINI VAISH (RA2211026010365)**

Under the guidance of   
**DR. JAYAKANTH J.J.***In partial fulfilment for the Course*

of

**21CSC202J – OPERATING SYSTEMS**

Department of Networking and Communications

**FACULTY OF ENGINEERING AND TECHNOLOGY**

**SCHOOL OF COMPUTING**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR**

**NOVEMBER 2023**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Under Section 3 of UGC Act, 1956)**

BONAFIDE CERTIFICATE

Certified that this project report for the course 21CSC202P OPERATING SYSTEMS entitled "MULTITHREADED WEB CRAWLER" is the bonafide work of Gauri Gupta (RA2211026010359), Neelansh Bhargava (RA2211026010360), and Mrinalini Vaish (RA2211026010365) who carried out the work under my supervision.

# PROJECT GUIDE

Dr. Jayakanth J.J.

# Assistant Professor

**Department of Networking and**

**Communication**

SRM Institute of Science and Technology

Kattankulathur

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| S.NO. | CONTENTS | PAGE |
| 1. | Objective | 1 |
| 2. | Problem Statement | 2 – 3 |
| 3. | Flowchart | 4 |
| 4. | Hardware Requirements | 5 – 6 |
| 5. | Software Requirements | 7 – 8 |
| 6. | Code | 9 – 12 |
| 7. | Output | 13 – 15 |
| 8. | Conclusion | 16 |

**OBJECTIVE**

The objective of a multithreaded web crawler is to browse through websites efficiently and concurrently, retrieving data and links from multiple web pages simultaneously. By employing multiple threads, a multithreaded web crawler can improve its performance significantly compared to a single-threaded counterpart. This approach enables the crawler to make better use of system resources, reduce the time taken to crawl large websites, and enhance overall efficiency in collecting data from the web.

**PROBLEM STATEMENT**

Design and implement a multithreaded web crawler that is capable of efficiently and concurrently exploring websites, retrieving relevant data, and extracting hyperlinks.

The web crawler should be able to:

1. Concurrency: Implement a multithreaded architecture to allow concurrent fetching and parsing of web pages. Utilize threads efficiently to maximize the utilization of system resources.
2. Robustness: Handle various types of web content, including text, images, videos, and other multimedia elements. Implement error handling mechanisms to deal with connection timeouts, server errors, and other exceptions gracefully.
3. Scalability: Design the system to be scalable, enabling it to crawl large websites with millions of pages without significantly affecting performance. Consider distributed computing techniques if necessary.
4. Data Extraction: Implement algorithms to extract relevant data from web pages. This may include text content, metadata, hyperlinks, and other structured or unstructured data as specified.
5. URL Filtering: Implement mechanisms to filter URLs to avoid crawling duplicate or irrelevant pages.

Handle URL normalization and canonicalization to ensure consistency in the crawled data.

1. Storage: Provide options for storing the crawled data, such as saving it in a database, indexing it for search, or exporting it to various file formats.
2. Monitoring and Reporting: Implement monitoring tools to track the progress of the crawling process, including the number of pages crawled, time taken, and any encountered errors. Generate reports summarizing the crawling activity.
3. Customizability: Allow users to configure the crawler parameters, such as the maximum depth of crawling, maximum number of threads, user-agent string, and other relevant settings.

**FLOWCHART**

A diagram of a web crawler

Description automatically generated

**HARDWARE REQUIREMENTS**

1. Processor (CPU): A multicore processor is essential for multithreaded applications. A quad-core or higher processor is recommended for efficient multithreaded web crawling.
2. Memory (RAM): Sufficient RAM is crucial for storing data structures, caching web pages, and managing concurrent threads. The exact amount of RAM needed depends on the size of the web pages you are crawling and the number of concurrent threads. At least 8 GB of RAM is recommended.
3. Storage: Web crawling involves storing a significant amount of data, including crawled web pages, extracted content, and other metadata. A high-capacity and high-speed storage drive is essential to handle the storage requirements. Consider having ample storage space, especially if you plan to crawl large websites or store historical data.
4. Network Interface: A fast and stable internet connection is important for fetching web pages efficiently. A Gigabit Ethernet connection or faster is recommended to ensure quick data retrieval from websites.
5. Operating System: Choose a reliable and scalable operating system that supports multithreading

efficiently. Popular choices include Linux distributions (such as Ubuntu, CentOS) or Windows Server editions..

1. Backup and Redundancy: Implement backup solutions and redundancy measures to prevent data loss in case of hardware failures. Regularly backup crawled data to prevent loss due to unforeseen circumstances.

**SOFTWARE REQUIREMENTS**

1. Programming Language: Popular languages for web crawling include Python, Java, C++, and Go.
2. Multithreading Library or Framework: Utilize a multithreading library or framework provided by the chosen programming language to implement concurrent processing. For example, Python has threading and concurrent. futures modules, Java offers the java. util. concurrent package.
3. Web Crawling Framework: Provide features like URL management, request handling, and data extraction, allowing you to focus on customizing the crawler's behavior. Examples include Scrapy (Python), and Apache Nutch (Java).
4. HTTP Client: Ensure the library supports features like connection pooling and asynchronous requests for efficient crawling.
5. HTML Parser: Employ an HTML parsing library to extract data from web pages. Example, BeautifulSoup (Python), and Jsoup (Java).
6. Database Management System (DBMS): If your web crawler needs to store crawled data persistently, choose a suitable database system. Options include relational databases like MySQL, PostgreSQL, or

SQLite.

1. Robots.txt Parser: Implement a parser for robots.txt files to respect the rules specified by websites.
2. Caching Mechanism: Implement a caching mechanism to store visited web pages temporarily. Caching helps reduce redundant HTTP requests and speeds up the crawling process.
3. Logging and Monitoring Tools: Integrate logging mechanisms to record the crawler's activities, errors, and performance metrics.
4. Version Control System: Use a version control system like Git to track changes in your codebase, collaborate with team members, and manage the development process efficiently.

**CODE**

from urllib.request import Request, urlopen, URLError, urljoin

from urllib.parse import urlparse

import time

import threading

import queue

from bs4 import BeautifulSoup

import ssl

class Crawler(threading.Thread):

    def \_\_init\_\_(self,base\_url, links\_to\_crawl,have\_visited, error\_links,url\_lock):

        threading.Thread.\_\_init\_\_(self)

        print(f"Web Crawler worker {threading.current\_thread()} has Started")

        self.base\_url = base\_url

        self.links\_to\_crawl = links\_to\_crawl

        self.have\_visited = have\_visited

        self.error\_links = error\_links

        self.url\_lock = url\_lock

    def run(self):

        # we create a ssl context so that our script can crawl

        # the https sties with ssl\_handshake.

        #Create a SSLContext object with default settings.

        my\_ssl = ssl.create\_default\_context()

        # by default when creating a default ssl context and making an handshake

        # we verify the hostname with the certificate but our objective is to crawl

        # the webpage so we will not be checking the validity of the cerfificate.

        my\_ssl.check\_hostname = False

        # in this case we are not verifying the certificate and any

        # certificate is accepted in this mode.

        my\_ssl.verify\_mode = ssl.CERT\_NONE

        # we are defining an infinite while loop so that all the links in our

        # queue are processed.

        while True:

            # In this part of the code we create a global lock on our queue of

            # links so that no two threads can access the queue at same time

            self.url\_lock.acquire()

            print(f"Queue Size: {self.links\_to\_crawl.qsize()}")

            link = self.links\_to\_crawl.get()

            self.url\_lock.release()

            # if the link is None the queue is exhausted or the threads are yet

            # process the links.

            if link is None:

                break

            # if The link is already visited we break the execution.

            if link in self.have\_visited:

                print(f"The link {link} is already visited")

                break

            try:

                # This method constructs a full "absolute" URL by combining the

                # base url with other url. this uses components of the base URL,

                # in particular the addressing scheme, the network

                # location and  the path, to provide missing components

                # in the relative URL.

                # in short we repair our relative url if it is broken.

                link = urljoin(self.base\_url,link)

                # we use the header parameter to "spoof" the "User-Agent" header

                # value which is used by the browser to identify itself. This is

                # because some servers will only allow the connection if  it comes

                # from a verified browser. In this case we are using FireFox header.

                req = Request(link, headers= {'User-Agent': 'Mozilla/5.0'})

                # we are opening the url using a ssl handshake.

                response = urlopen(req, context=my\_ssl)

                print(f"The URL {response.geturl()} crawled with \

                      status {response.getcode()}")

                # this returns the html representation of the webpage

                soup = BeautifulSoup(response.read(),"html.parser")

                # in this case we are finding all the links in the page.

                for a\_tag in soup.find\_all('a'):

                    # we are checking of the link is already visited and (network location part) is our

                    # base url itself.

                    if (a\_tag.get("href") not in self.have\_visited) and (urlparse(link).netloc == "www.python.org"):

                        self.links\_to\_crawl.put(a\_tag.get("href"))

                    else:

                        print(f"The link {a\_tag.get('href')} is already visited or is not part \

                        of the website")

                print(f"Adding {link} to the crawled list")

                self.have\_visited.add(link)

            except URLError as e:

                print(f"URL {link} threw this error {e.reason} while trying to parse")

                self.error\_links.append(link)

            finally:

                self.links\_to\_crawl.task\_done()

print("The Crawler is started")

base\_url = input("Please Enter Website to Crawl > ")

number\_of\_threads = input("Please Enter number of Threads > ")

links\_to\_crawl = queue.Queue()

url\_lock = threading.Lock()

links\_to\_crawl.put(base\_url)

have\_visited = set()

crawler\_threads = []

error\_links = []

#base\_url, links\_to\_crawl,have\_visited, error\_links,url\_lock

for i in range(int(number\_of\_threads)):

    crawler = Crawler(base\_url = base\_url,

                      links\_to\_crawl= links\_to\_crawl,

                      have\_visited= have\_visited,

                      error\_links= error\_links,

                      url\_lock=url\_lock)

    crawler.start()

    crawler\_threads.append(crawler)

for crawler in crawler\_threads:

    crawler.join()

print(f"Total Number of pages visited are {len(have\_visited)}")

print(f"Total Number of Errornous links: {len(error\_links)}")

**OUTPUT**

****

**A screenshot of a computer code

Description automatically generated**

**A white paper with black text

Description automatically generated**

**A close-up of a text

Description automatically generated**

**A close up of a text

Description automatically generated**

**A close up of a paper

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A close up of a text

Description automatically generated**

**CONCLUSION**

A multithreaded web crawler operating system offers significant advantages in terms of performance and efficiency. By utilizing multiple threads, it can parallelize the crawling process, allowing for faster data retrieval and processing. This design can enhance the scalability of web crawling applications, making them well-suited for large-scale data harvesting tasks.