Chapter 4

DESIGN OF A SCALABLE PARALLEL MIGRATING CRAWLER

4.1 THE ARCHITECTURE

The Parallel Migrating Crawler designed in this work consists of three subsystems as shown in Fig. 4.1.

Fig. 4.1: Three Tier Architecture of Parallel Migrating Crawler

1. Mapping subsystem: It parallely maps URLs into their IP addresses and stores them in resolved URL-IP queue.
2. Crawling subsystem: After calculating the revisit frequency, this subsystem transports the crawling code to the nearest server for the purpose of downloading the documents.
3. Search engine interface modules that executes user query on its repository.

A detailed description of each module is given in following sections:

4.1.1 THE MAPPING SUBSYSTEM

The architecture of mapping subsystem is shown in Fig. 4.2.
It may be noted that the main component of mapping subsystem is parallel URL mapper. It converts the URLs into their IP addresses. As discussed in chapter 3 modifications were made in the existing structure of URL record, the mapping subsystem initializes the various fields of modified URL record.

A brief discussion on each component is given in following section.

4.1.1.1 URL Dispatcher

URL dispatcher reads the URLs from the search engine database and fills the URL-IP queue. It sends a signal `mapUrls` to the parallel URL mapper. Its algorithm is given in Fig. 4.3:

```plaintext
URL_Dispatcher ()
Do Forever
Step 1: While not_full(URL-IP Queue)
   1.1: Read URL-IP pair from search engine repository;
   1.2: Store it into URL-IP Queue;
   2: Signal (mapUrls);
```

Fig. 4.3: Algorithm for URL Dispatcher
4.1.1.2 Parallel URL Mapper

It waits on the signal mapUrls. After receiving the signal it creates multiple mapper threads. Simultaneously it also extracts URL-IP pairs from the URL-IP queue and assembles this in set of URL-IP pairs. There after the URL-IP pair sets are distributed among the mapper threads. A URL mapper thread examines each URL-IP pair and if IP is blank then gets it resolved from DNS resolver. The resolved URL-IP pair is stored in the resolved URL-IP queue. It also invokes the field initializer to initialize the various fields of modified URL record. The algorithmic detail for parallel URL mapper is given in Fig. 4.4.

<table>
<thead>
<tr>
<th>Parallel URL Mapper ( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: Create Parallel instances of mapper threads M₁ to Mₙ;</td>
</tr>
<tr>
<td>Do forever</td>
</tr>
<tr>
<td>2: Wait (mapUrls);</td>
</tr>
<tr>
<td>3: While (URL-IP Queue not empty)</td>
</tr>
<tr>
<td>3.1: Pickup URLs from URL-IP Queue;</td>
</tr>
<tr>
<td>3.2: Assemble a set of URL-IPs;</td>
</tr>
<tr>
<td>3.3: Assign the set to an idle Map worker;</td>
</tr>
<tr>
<td>3.4: If (URL.Status!=1)</td>
</tr>
<tr>
<td>3.4.1: Call (Field Initializer);</td>
</tr>
</tbody>
</table>

Fig. 4.4: Algorithm for Parallel URL Mapper

4.1.1.3 DNS Resolver

The Internet offers a service called Domain Name System (DNS). The DNS resolver uses this service to resolve the DNS address for a URL and returns it back to the calling URL mapper thread. Generally the documents are known by the domain names of their servers. The name of the server is translated into its corresponding IP address with the help of DNS resolver. Having resolved the IP address, the crawler starts communicating with the target server.

4.1.1.4 Field Initializer

If a URL is encountered first time during the mapping phase then its record containing predefined values is stored into the modified URL record. The following example illustrates the initializing process of the URL record (See Fig. 4.5).
Set URL.LCT = null;
Set URL.ffffff = Average_Revisit_Frequency;
Set URL.λ_lower = lower threshold;
Set URL.λ_middle = middle threshold;
Set URL.λ_upper = upper threshold;
Set URL.Nearest Location = null;
Set URL.Status = 1;
Insert URL record set in database;

Fig. 4.5: Initialized Fields of URL Record

Since the URL has been encountered first time, its status field is initialized with the value ‘1’ in order to avoid future re-initializations.

The mapping subsystem also consists of some data structures as discussed below.

4.1.1.5 URL-IP Queue

It consists of a queue of unique seed URL-IP pairs. The IP part may or may not be blank. It acts as an input to the mapping manager.

4.1.1.6 Resolved URL-IP Priority Queue

It stores resolved URLs as a max heap. This heap is maintained on the basis of dynamically calculated revisit frequency i.e. the URL with highest revisit frequency shall be at the top of the heap. It acts as input to the crawling subsystem.

4.1.2 THE CRAWLING SUBSYSTEM

The crawling module is composed of six major components namely Domain Separator, Nearest Location Finder, Crawl Manager, Migrants, Security Module and Revisit Frequency Calculator. A broad level component diagram of crawling subsystem is shown in Fig. 4.6.
Domain separator separates the URLs into categories based on their domains i.e. extensions (e.g. .com, .edu etc).

Nearest location finder finds the nearest server of a mirrored document. It also determines whether a related URL is already scheduled for that server or not.

Crawl manager creates multiple downloading instances called migrants that can migrate to a site for downloading the documents.

Revisit frequency calculator computes the revisit frequency for the next visit to a site and accordingly migrants visit the web servers.

The Security module ensures a secure execution of the migrants on the web servers and safe transportation of downloaded data through Internet.

The detailed architecture of crawling subsystem is shown in Fig. 4.7.
Fig. 4.7: Architecture of Crawling Subsystem
Detailed discussion on each functional component of the crawling subsystem is given in following sections.

4.1.2.1 Domain Separator

It may be noted from chapter-2 that URLs are divided in its domains. A parallel crawler may allocate the URLs from the same domains to the different crawl worker. Due to the possible existence of cross links among the sites belonging to same domain (i.e. extension), the crawler may download the same document multiple times. For mitigating this duplicate downloading of the documents, it is proposed that before allocating to the crawl manager for downloading, the URLs may be separated in to multiple queues based on their different extensions. Thereby avoid the duplicate downloading of the documents with a strong possibility of finding the document belonging to same domain at local server itself. The domain separator module takes the URLs from resolved URL priority queue and separates them in different sets based on their domains. These sets are distributed to different queues, as shown in Fig. 4.7. A top level domain separator can also be there on the basis of geographical extensions i.e. .in, .uk etc. Thus overhead at migrants’ side is further reduced. After separating the domains it sends signal called domain separated to the revisit frequency calculator module. The algorithm of domain separator for seven generic domains is given in Fig. 4.8.

```
Domain Separator ()
Step 1: While (resolved URL priority queue ≠ empty)
  2: Pick up URL from resolved URL priority queue
  3: If (domain part of URL = .COM)
    3.1: append (LIST.COM, URL);
    Else If (domain part of URL = .EDU)
    3.2: append (LIST.EDU, URL);
    Else If (domain part of URL = .GOV)
    3.3: append (LIST.GOV, URL);
    Else If (domain part of URL = .NET)
    3.4: append (LIST.NET, URL);
    Else If (domain part of URL = .ORG)
    3.5: append (LIST.ORG, URL);
    Else If (domain part of URL = .MIL)
    3.6: append (LIST.MIL, URL);
    Else If (domain part of URL = .INT)
    3.7: append (LIST.INT, URL);
    Else
    3.8: append (LIST.OTHERS, URL);
  4: Signal (domain separated);
```

Fig. 4.8: Algorithm for Domain Separator
4.1.2.2 Nearest Location Finder

The Internet Assigned Numbers Authority (IANA) is the entity that looks after global IP address allocation, Domain Name System (DNS) and other Internet Protocol related assignments. IANA delegates local registrations of IP addresses to Regional Internet Registries (RIRs). Each RIR allocates addresses for a different area of the world. There are currently five RIRs in operation [58, 59]:

- American Registry for Internet Numbers (ARIN) for North America and parts of the Caribbean.
- RIPE Network Coordination Centre (RIPE NCC) for Europe, the Middle East and Central Asia.
- Asia-Pacific Network Information Centre (APNIC) for Asia and the Pacific region
- Latin American and Caribbean Internet Addresses Registry (LACNIC) for Latin America and parts of the Caribbean region
- African Network Information Centre (AfriNIC) for Africa.

IANA delegates the allocation of IP addresses to RIRs in large blocks. The RIRs sub-allocate smaller blocks in their regions to Internet service providers and other organizations.

The RIR of respective region is utilized by nearest location finder module to locate the best possible place of a document. An inquiry message is sent to the local RIR for getting the information for the subnets containing the IP addresses of the documents lying at more than one places. Probing messages are then sent to those subnets for knowing about the access time. Accordingly, subnet with the least probing time is used by the crawler for downloading the document.

The nearest location finder module waits on signal domain_separated. After receiving the signal, it facilitates the delegation of URLs to the migrants such that each page is crawled from the nearest possible server (i.e. from where the crawler would download the page fastest). ICMP ping tool is employed to find the information about the region of the
URL’s IP address and the URL with the smallest probing time is used to download the web pages. The algorithm for nearest location finder is given in Fig. 4.9.

<table>
<thead>
<tr>
<th>Location finder ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: Read the URL from domain specific Queue;</td>
</tr>
<tr>
<td>2: Determine the subnets containing the IP address of new URL from IP hierarchy based on RIR;</td>
</tr>
<tr>
<td>3: If IP is contained by single Subnet</td>
</tr>
<tr>
<td>Then</td>
</tr>
<tr>
<td>3.1: If (probing time to the subnet &lt; threshold probing time)</td>
</tr>
<tr>
<td>3.1.1: Update location field of URL record</td>
</tr>
<tr>
<td>3.1.2: Signal (location);</td>
</tr>
<tr>
<td>Else</td>
</tr>
<tr>
<td>3.2: Find out the subnet with lowest probing time and</td>
</tr>
<tr>
<td>3.3: Update the URL record;</td>
</tr>
<tr>
<td>4: Signal (locFound);</td>
</tr>
</tbody>
</table>

Fig. 4.9: Algorithm for Nearest Location Finder

### 4.1.2.3 Crawl Manager

The crawl manager waits on a signal called *locFound*. After receiving the signal it creates multiple domain (i.e. extension) specific migrating crawler instances called *migrants*. The information from URL from domain specific queue URLQ(1..n) is used to check whether a migrant has already been sent to that location or not. A registry of various migrants sent to the various sites is maintained with a view to keep track of the location of the migrants being sent. If some migrant is already on that site then crawl manager assigns the corresponding URL to that migrant else assigns the URL to an idle migrant. Thereafter it waits on signal called *docAvail* which is sent by a migrant after storing the downloaded documents in a temporary buffer. After receiving the signal it sends signal *update_frequency* to the revisit frequency calculator module [50]. Revisit frequency calculator module uses the fingerprinting scheme [78] (*FP Matcher* in Fig. 4.7) for detecting the changes in documents and accordingly calculates the frequency of revisit and updates URL-record. All collected documents are later stored in document and URL buffer. The algorithm for Crawl Manager is given in Fig. 4.10.
Crawl_Manager ( )
Do Forever
Step 1: wait (location);
   2: Creates multiple migrating crawler instances called
      Migrants $M_i$ to $M_n$;
   3: Pick up URLs from Domain specific URLQ(1..n)
   4: If (some migrant has already been sent to this location)
      4.1: Assign URL to that Migrant;
      Else
      4.2: Assign URL to an idle Migrant and migrate it to LOCATION;
   5: wait (data);
   6: signal (update_frequency);
   7: wait (frequency updated);

Fig. 4.10: Algorithm for Crawl Manager

4.1.2.4 Revisit Frequency Calculator

It waits on signal update_frequency. After receiving the signal it calculates the change
frequency of pages $\lambda_{\text{current}}$ using the fingerprinting scheme [78] and accordingly
computes the frequency of revisit as per the formula [50] discussed in chapter 3 and
updates the URL record. It then sends a signal frequency_updated to the crawl manager.
The algorithm for revisit frequency calculator module is given in Fig. 4.11.

Revisit Frequency Calculator ( )
Do Forever
Step 1: wait (update_frequency);
   2: Calculates the current change frequency of pages using
documents fingerprinting scheme
   3: Computes next frequency of revisit
   4: Update URL record
   5: signal (frequency_updated);

Fig 4.11: Algorithm for Revisit Frequency Calculator Module

4.1.2.5 Migrants

The migrants are managed by crawl manager that supplies each migrant a list of target
websites and thereafter monitors their movements. After migrating to target website a
migrant follow the algorithm for downloading the documents developed during
PARCAHYD [40] projects. It maintains two data structures LocalQ and MainQ. MainQ
stores the URLs supplied by the crawl manager and external links found in the downloaded documents. LocalQ stores the internal links. After downloading the documents from the server it filters the collection for relevant contents, compresses the data before sending it to search engine database and sends the signal docAvail to crawl manager. In fact, augmented hyper text documents speed up the downloading process whereas filtering and compression reduces the network load. The algorithm of migrant at remote server is shown in Fig. 4.12.

Migrant ( )
Step 1: Pickup the URLs assigned by the Migrating Crawler Manager and stores them in MainQ;
2: While (remote MainQ ≠ empty)
   2.1: Download robot.txt;
   2.2: If unable to download
      2.2.1: Set IP part as blank;
      2.2.2: Store URL in local document and URL Buffer;
      Else
      2.2.3: Read robot.txt;
      2.2.4: Download TOL;
      2.2.5: Segregate the internal and external Links;
      2.2.6: Add URL and internal Links to LocalQ;
      2.2.7: Store the External Links to Documents and URL Buffer;
3: While (Remote LocalQ ≠ empty)
   3.1: Pickup a URL from LocalQ;
   3.2: Download documents and store them in document and URL buffer;
4: Filter ( );
5: Compress ( );
6: Send the compressed and filtered documents and URLs to the Migrating Crawl Manager’s Documents and URL Buffer.
7: Signal (docAvail);

Fig. 4.12: Algorithm for Migrant

4.1.2.6 Snooping

Being domain i.e. extension specific in nature, the proposed migrating crawler reduces the possibility of multiple downloads of same version of a document. However, it is observed that some websites belonging to different domains contain cross links. For example, many of the government sites with domain .gov contain links to educational websites belonging to .edu domain as illustrated in Fig. 4.13.
Now, the migrants that visit multiple servers may download the same page due to cross links between the websites. These duplicate web pages shall not only overload the Internet but also decrease the quality of the collection. Thus it is proposed that while transmitting the documents to the search engine side for updating the collection, every intermediate node should snoop the contents of the transmitted documents (see Fig. 4.14).

As and when a duplicate page is detected, it shall be discarded on the current node itself thereby, mitigating the possible transmission of duplicate files/documents because of cross links among the sites.

It may be observed from the Fig. 4.14 that Node ‘B’ has transmitted three documents with Doc id 1, 2, and 3. Node ‘C’ also has transmitted three documents with Doc id 1, 2,
and 4. The snooping node A detects the duplicate documents with Doc id 1 and 2. It compares the two files and allows the transmission of documents with latest update time and discards the older one. In fact the snooping node significantly improves the quality by removing the duplicate documents and improves the network utilization as well.

The crawling subsystem has the following data structures:

**4.1.2.7 Domain Specific URLQ \((1..n)\)**

These are the priority queues for storing the domain specific URLs, separated by domain separator module. The priority of URLs is based on their revisit frequency \((f)\), i.e. higher the revisit frequency higher the priority. Crawl manager picks the URLs from these domain specific queues and allocates them to the migrants.

**4.1.2.8 Local Document and URL Buffer**

It is a buffer used by the migrants for storing the downloaded documents locally. The process filter and compression is applied on this buffer before sending its contents to the Crawl manager’s buffer.

**4.1.2.9 Recently Downloaded Documents Buffer**

This buffer is used to store the recently downloaded documents sent by migrants. This buffer is used by the revisit frequency calculator module for updating the revisit frequency of URL.

The performance analysis of the proposed architecture is shown in following section.

**4.2 PERFORMANCE ANALYSIS**

The migrating crawler, designed in this work, has been implemented on Java platform. The performance was compared with a conventional non migrating crawler. The conventional crawler downloads the pages and makes decision regarding overlapping and relevancy only after downloading a set of documents resulting in overloading of the
machine along with miss utilization of the network resource. On the contrary, the migrants move to the data sources perform local filtering, compress the documents before transmitting them to the search engine collection. Moreover, any redundancy is detected by snooping host which discard the duplicate documents before transmitting them to the search engine collection thereby increasing the network utilization and ensuring the quality of the collection.

4.2.1 PROCEDURE

The test was conducted on the intranet of YMCA university of Science and Technology where mother system employed three instances of migrants. On a separate setup a conventional crawler was provided with the same set of URLs as given to the migrating crawler. The results obtained thereof are discussed in the following section:

4.2.1.1 Test 1

In this test the following set of URLs was supplied to both the crawlers:

- http://www.careerlauncher.com/
- http://www.iiht.com/
- http://www.impeccabletrainers.co.in/

Results obtained thereof are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Conventional Crawler</th>
<th>Migrating Crawler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl Time (ms)</td>
<td>850376</td>
<td>657093</td>
</tr>
<tr>
<td>Total Data downloaded (KB)</td>
<td>639.9</td>
<td>639.9</td>
</tr>
<tr>
<td>Total Data after filtration (KB)</td>
<td>NA</td>
<td>552.4</td>
</tr>
<tr>
<td>Total data after compression (KB)</td>
<td>NA</td>
<td>104.1</td>
</tr>
<tr>
<td>Number of repeated URLs</td>
<td>315</td>
<td>55</td>
</tr>
</tbody>
</table>
4.2.1.2 Test 2

In this test the following set of URLs was supplied to both the crawlers:

http://www.bluestarindia.com/
http://www.videoconworld.com/
http://www.sansui-india.com/

Results obtained thereof are shown in Table 4.2.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Conventional Crawler</th>
<th>Migrating Crawler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl Time (ms)</td>
<td>1580078</td>
<td>1206953</td>
</tr>
<tr>
<td>Total Data downloaded (KB)</td>
<td>495.5</td>
<td>495.6</td>
</tr>
<tr>
<td>Total Data after filtration (KB)</td>
<td>NA</td>
<td>464.9</td>
</tr>
<tr>
<td>Total data after compression (KB)</td>
<td>NA</td>
<td>95.6</td>
</tr>
<tr>
<td>Number of repeated URLs</td>
<td>493</td>
<td>112</td>
</tr>
</tbody>
</table>

4.2.1.3 Test 3

In this test the following set of URLs was supplied to both the crawlers:

http://www.ymcaie.ac.in/
http://rkgit.edu.in/
http://www.iimtindia.net/

Results obtained thereof are shown in Table 4.3.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Conventional Crawler</th>
<th>Migrating Crawler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl Time (ms)</td>
<td>1216437</td>
<td>373203</td>
</tr>
<tr>
<td>Total Data downloaded (KB)</td>
<td>843.1</td>
<td>843.1</td>
</tr>
<tr>
<td>Total Data after filtration (KB)</td>
<td>NA</td>
<td>409.5</td>
</tr>
<tr>
<td>Total data after compression (KB)</td>
<td>NA</td>
<td>86.8</td>
</tr>
<tr>
<td>Number of repeated URLs</td>
<td>221</td>
<td>38</td>
</tr>
</tbody>
</table>
4.2.2 TIME MEASUREMENT

A comparison of crawling time obtained during Test 1 to Test 3 between conventional crawler and migrating crawler is shown in Fig. 4.15.

![Crawling Time Comparison](image)

Fig 4.15: Crawling Time Comparison

The average crawling time comparison is shown in Fig. 4.16.

![Average Crawling Time](image)

Fig 4.16: Average Crawling Time Comparison

Average crawling time of conventional crawler = 1215630
Average crawling time of migrating crawler = 745749
Benefit in crawling time by using migrating crawler = 100 - ((745749/1215630)*100) = 38.66%
4.2.3 QUALITY MEASUREMENT

A comparison of quality based on the repeated URLs traversed during Test 1 to Test 3, between conventional crawler and migrating crawler is shown in Fig. 4.17.

![Documents Quality Comparison](image)

Fig 4.17: Quality Comparison

The average numbers of repeated URLs traversed are shown in Fig. 4.18.

![Average Quality](image)

Fig 4.18: Average Quality Comparison

Average number of repeated URLs in Conventional crawler = 343
Average number of repeated URLs in migrating crawler = 68.
Percentage quality improvement by using migrating crawler = $100 - \left(\frac{68}{343}\right) \times 100$
= 80.2%
4.2.4 NETWORK RESOURCE UTILIZATION

A comparison of network load obtained during Test 1 to Test 3 between conventional crawler and migrating crawler is shown in Fig. 4.19.

![Network Load Comparison](image)

Fig 4.19: Network Load Comparison

The average network load during download is shown in Fig. 4.20.

![Average Network Load](image)

Fig 4.20: Average Network Load Comparison

Average network load by using conventional crawler = 659 KB
Average network load by using migrating crawler = 96 KB
Percentage Network load reduction by using migrating crawler = 100 - (96/659) * 100 = 85.43%
4.2.5 SUMMARY OF THE PERFORMANCE ANALYSIS

The architecture of migrating crawler designed in this work has been implemented using Java and the results obtained thereof were compared with the conventional system of crawling and following observations were made:

- The time required to crawl the same set of URLs comes out is 38.66% less than the conventional crawler.
- Average number of repeated URLs traversed is reduced by a factor of 80.2%.
- The network bandwidth exploited by the migrating crawler is found 85.43% less than the conventional crawler.

The security aspects related to migrants, host and the migrants’ downloaded data have been identified and discussed in the next chapter.