Web Crawler

- Finds and downloads web pages automatically
  - provides the collection for searching
- Web is huge and constantly growing
- Web is not under the control of search engine providers
- Web pages are constantly changing
- Crawlers also used for other types of data
Retrieving Web Pages

- Web crawler client program connects to a domain name system (DNS) server
- DNS server translates the hostname into an internet protocol (IP) address
- Crawler then attempts to connect to server host using specific port
- After connection, crawler sends an HTTP request to the web server to request a page
  – usually a GET request

Crawling the Web
Web Crawler

- Starts with a set of *seeds*, which are a set of URLs given to it as parameters
- Seeds are added to a URL request queue
- Crawler starts fetching pages from the request queue
- Downloaded pages are parsed to find link tags that might contain other useful URLs to fetch
- New URLs added to the crawler’s request queue, or *frontier*

Processing steps in crawling

- Pick a URL from the frontier
- Fetch the document at the URL
- Parse the URL
  - Extract links from it to other docs (URLs)
- Check if URL has content already seen
  - If not, add to indexes
- For each extracted URL
  - Ensure it passes certain URL filter tests
  - Check if it is already in the frontier (duplicate URL elimination)

E.g., only crawl .edu, obey robots.txt, etc.
Basic crawl architecture

What any crawler must do

- **Be Robust**: Be immune to spider traps and other malicious behavior from web servers

- **Be Polite**: Respect implicit and explicit politeness considerations
What any crawler *should* do

- Be capable of **distributed** operation: designed to run on multiple distributed machines
- Be **scalable**: designed to increase the crawl rate by adding more machines
- **Performance/efficiency**: permit full use of available processing and network resources

Web Crawling

- Web crawlers spend a lot of time waiting for responses to requests
- To reduce this inefficiency, web crawlers use threads and fetch hundreds of pages at once
- Crawlers could potentially flood sites with requests for pages
- To avoid this problem, web crawlers use *politeness policies*
  - e.g., delay between requests to same web server
URL frontier: two main considerations

- **Politeness**: do not hit a web server too frequently
- **Freshness**: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

These goals may conflict with each other.
(E.g., simple priority queue fails – many links out of a page go to its own site, creating a burst of accesses to that site.)

Explicit and implicit politeness

- **Explicit politeness**: specifications from webmasters on what portions of site can be crawled
  - robots.txt
- **Implicit politeness**: even with no specification, avoid hitting any site too often
Controlling Crawling

- Even crawling a site slowly will anger some web server administrators, who object to any copying of their data.
- Robots.txt file can be used to control crawlers.

Robots.txt example

User-agent: *
Disallow: /private/
Disallow: /confidential/
Disallow: /other/
Allow: /other/public/

User-agent: FavoredCrawler
Disallow:

Sitemap: http://mysite.com/sitemap.xml.gz

Robots.txt example

- No robot should visit any URL starting with "/yoursite/temp/", except the robot called "searchengine":

User-agent: *
Disallow: /yoursite/temp/

User-agent: searchengine
Disallow:
Politeness – challenges

- Even if we restrict only one thread to fetch from a host, can hit it repeatedly
- Common heuristic: insert time gap between successive requests to a host that is >> time for most recent fetch from that host

URL frontier: Mercator scheme

URLs

Prioritizer

$K$ front queues

Biased front queue selector

$B$ back queues

Single host on each

Back queue selector

Crawl thread requesting URL
Mercator URL frontier

- URLs flow in from the top into the frontier
- **Front queues** manage prioritization
- **Back queues** enforce politeness
- Each queue is FIFO

![Diagram of Mercator URL frontier]

---

Front queues

- Prioritizer
- **Biased front queue selector**
- Back queue router

![Diagram of Front queues]
Front queues

- Prioritizer assigns to URL an integer priority between 1 and $K$
  - Appends URL to corresponding queue
- Heuristics for assigning priority
  - Refresh rate sampled from previous crawls
  - Application-specific (e.g., “crawl news sites more often”)

Biased front queue selector

- When a back queue requests a URL (in a sequence to be described): picks a front queue from which to pull a URL
- This choice can be round robin biased to queues of higher priority, or some more sophisticated variant
  - Can be randomized
Back queues

Biased front queue selector
Back queue router

Back queue selector

Heap

Back queue processing

- A crawler thread seeking a URL to crawl:
  - Extracts the root of the heap
  - Fetches URL at head of corresponding back queue $q$ (look up from table)
  - Checks if queue $q$ is now empty – if so, pulls a URL $v$ from front queues
    - If there’s already a back queue for $v$’s host, append $v$ to it and pull another URL from front queues, repeat
    - Else add $v$ to $q$
  - When $q$ is non-empty, create heap entry for it
Freshness

• Web pages are constantly being added, deleted, and modified
• Web crawler must continually revisit pages it has already crawled to see if they have changed in order to maintain the *freshness* of the document collection
  – *stale* copies no longer reflect the real contents of the web pages

Freshness

• HTTP protocol has a special request type called HEAD that makes it easy to check for page changes
  – returns information about page, not page itself

Client request: HEAD /csinfo/people.html HTTP/1.1
Host: www.cs.umass.edu

HTTP/1.1 200 OK
Date: Thu, 03 Apr 2008 05:17:54 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Fri, 04 Jan 2008 15:28:39 GMT

Server response: ETag: "239c33-2576-2a2837c0"
Accept-Ranges: bytes
Content-Length: 9590
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Freshness

- Not possible to constantly check all pages
  - must check important pages and pages that change frequently
- Freshness is the proportion of pages that are fresh
- Optimizing for this metric can lead to bad decisions, such as not crawling popular sites
- Age is a better metric

Freshness vs. Age

- Diagram showing the relationship between freshness and age with updates and crawls.
Age

• Expected age of a page $t$ days after it was last crawled:

$$\text{Age}(\lambda, t) = \int_{0}^{t} P(\text{page changed at time } x)(t - x)dx$$

• Web page updates follow the Poisson distribution on average
  – time until the next update is governed by an exponential distribution

$$\text{Age}(\lambda, t) = \int_{0}^{t} \lambda e^{-\lambda x}(t - x)dx$$

Age

• The older a page gets, the more it costs not to crawl it
  – e.g., expected age with mean change frequency $\lambda = 1/7$ (one change per week)