

## Internet Information 2006/2007 Crawling

Valentin Jijkoun  
Maarten de Rijke

ISLA, University of Amsterdam

<http://ilps.science.uva.nl/Teaching/II0607>

February 12, 2007

1

## Information access

- Get the data
- Analyse the data
- Search the data
  - ✓ Last week: Boolean, VS, LM
    - This was the focus of much of the past research in IR
- Act with data
- Today:
  - Getting the data: crawling, document processing
  - Analyzing and searching: constructing and using indexes

Internet Information/MIKI6, February 12, 2007

Valentin Jijkoun, Maarten de Rijke 2

## Crawling

- Task of the crawler: getting web documents to be analyzed at local machines
  - Indexing, text mining, link analysis, clustering, ...
- History
  - First crawler: Wanderer, 1993 (Matthew Gray)
  - Other terms used, distinction now blurred:
    - crawler or spider: go and bring the data
    - robot or bot: do something useful (Wikipedia maintenance), entertaining (chatterbots), commercial/malicious (online voting, auctions, spam delivery)...
- What is important when designing a high-performance (scalable) web crawler?

Internet Information/MIKI6, February 12, 2007

Valentin Jijkoun, Maarten de Rijke 3

## Features of a web crawler

- Robust: network delays/failures, invalid markup, spider traps
- Polite: self-limiting
  - minimal load on servers, META tags, robots.txt
- Scalability: easy to add machines and network connections
  - distributed architecture
- Performance: effective use of resources
  - CPU, memory, network bandwidth, communication overhead, storage
- Quality: get "useful" pages first
  - sometimes: focused (on-topic) crawling
- Freshness: estimate change rates and update crawls

Internet Information/MIKI6, February 12, 2007

Valentin Jijkoun, Maarten de Rijke 4

## Crawling the web

- Web pages
  - average page: 10K of HTML
  - Contain links to URLs of other resources (HTML or not)
  - Served through the Internet using the Hypertext Transfer Protocol (HTTP)
- Crawler
  - Uses HTTP's GET/HEAD/POST requests to fetch the pages to the computer
  - Once retrieved, data is passed to storage/analysis/indexing/...

Internet Information/MIKI6, February 12, 2007

Valentin Jijkoun, Maarten de Rijke 5

## HTTP: Hypertext Transfer Protocol

- Next layer over the Transmission Control Protocol (TCP)
  - request/response mode
- Steps (from client end)
  - Resolve the server host name to an Internet address (IP)
    - via Domain Name Service (DNS) (name-to-IP mappings)
    - [ilps.science.uva.nl](http://ilps.science.uva.nl) ⇒ 146.50.22.166
- Contact the server using TCP
  - Connect to the specified TCP port (HTTP: default 80)
  - Enter the HTTP request headers (e.g.: GET index.html)
  - Receive the response header
    - Includes MIME (Multipurpose Internet Mail Extensions) type
    - Meta-data standard for email and Web content transfer
  - Receive the (hopefully) HTML page or other requested resource

Internet Information/MIKI6, February 12, 2007

Valentin Jijkoun, Maarten de Rijke 6

## Sample HTTP session

### Request:

```
GET /Teaching/II0607 HTTP/1.1
User-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.0.7) Gecko/200
Host: ilps.science.uva.nl
Accept: text/html
```

### Response:

```
HTTP/1.1 200 OK
Date: Sun, 11 Feb 2007 20:17:07 GMT
Server: Apache/2.0.53 (Fedora)
Last-Modified: Sat, 27 Jan 2007 16:19:53 GMT
ETag: "24c74a-70-3a77c40"
Accept-Ranges: bytes
Content-Length: 112
Connection: close
Content-Type: text/html; charset=UTF-8
```

Internet Information/MIKI6, February 12, 2007

Valentin Jijkoun, Maarten de Rijke 7

```
<html>
<head>
<meta http-equiv='refresh' content='0; url=twiki/bin/view/Main/WebHome'>
</head>
<body></body>
</html>
```

Internet Information/MIKI6, February 12, 2007

Valentin Jijkoun, Maarten de Rijke 8

## Crawl "all" web pages?

- Problem: no catalog of all accessible URLs on the Web
- Solution:
  - Start from a given set of "seed" URLs
  - Progressively fetch and scan them for new outlinking URLs
  - Fetch these pages in turn
  - Save pages, extract outlinks
  - and so on

## Crawling procedure

- Conceptually simple
  - Great deal of engineering goes into industry-strength crawlers
  - Industry crawlers crawl a substantial fraction of the Web
  - E.g.: AltaVista, Google, Inktomi, Teoma
- No guarantee that all accessible Web pages will be located in this fashion
  - disconnected segments of the web (no in-links)
  - temporary network failures . . .
- When does a crawler stop?
  - after having collected "enough" data
  - or never

## Crawling overheads

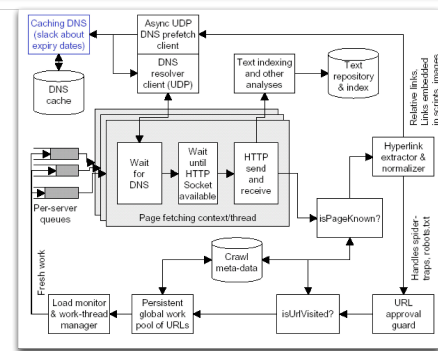
- Delays involved in
  - Resolving the host name in the URL to an IP address using DNS
  - Connecting a socket to the server and sending the request
  - Receiving the requested page in response
- Up to several seconds per page
  - A simple crawler would be idle most of the time
- Solution
  - Overlap the above delays by fetching many pages at the same time

## Anatomy of a crawler

- Parallel page fetching threads
  - Starts with DNS resolution
  - Finishes when the entire page has been fetched
- Each page
  - Stored in compressed form
  - Scanned for outlinks
- Work pool of outlinks
  - maintain network utilization without overloading it
- Continue until the crawler has collected a sufficient number of pages

## Anatomy of a large-scale crawler

■



## Large-scale crawlers: performance and reliability

- Need to fetch many pages at same time
  - Efficiently utilize the network bandwidth
  - Single page fetch may involve several seconds of network latency
- Highly concurrent and parallelized DNS lookups
- Use of asynchronous sockets (vs. multi-threading)
  - Explicit encoding of the state of a fetch context in a data structure: event-base processing
  - Polling socket to check for completion of network transfers
  - Multi-processing or multi-threading: Impractical
- Care in URL extraction
  - Eliminating duplicates to reduce redundant fetches
  - Avoiding "spider traps"

## DNS caching, pre-fetching, and resolution

- A customized DNS component with . . .
  - Custom client for address resolution
  - Caching server
  - Prefetching client
- Custom client for address resolution
  - Tailored for concurrent handling of multiple outstanding requests
  - Allows issuing of many resolution requests together
  - Perform polling at a later time for completion of individual requests
  - Facilitates load distribution among many DNS servers
- Caching DNS server
  - With a large cache, persistent across DNS restarts
  - Ignore expiration times
  - Residing largely in memory if possible



## Prefetching DNS client

- "I will need IPs of these servers soon!"
- Steps
  - Parse a page that has just been fetched
  - Extract host names from HREF targets
  - Make DNS resolution requests to the caching server
    - (before putting target URL in the queue!)
- Usually implemented using UDP
  - User Datagram Protocol
  - Connectionless, packet-based communication protocol
  - Does not guarantee packet delivery
- Does not wait for resolution to be completed



## Multiple concurrent fetches

- Managing multiple concurrent connections
  - A single download may take several seconds
  - Open many socket connections to different HTTP servers simultaneously
- Multi-CPU machines not useful
  - crawling performance limited by network and disk
- Two approaches
  - using multi-threading
  - using non-blocking sockets with event handlers



## Multi-threading

- Logical threads
  - physical thread provided by the OS (E.g.: pthreads) OR
  - concurrent processes
- Fixed number of threads allocated in advance
- programming paradigm
  - create a client socket
  - connect the socket to the HTTP service on a server
  - send the HTTP request header
  - read the socket (recv) until no more characters are available
  - close the socket
- Use blocking system calls



## Multi-threading: Problems

- Performance penalty
  - mutual exclusion
  - concurrent access to data structures
- Slow disk seeks
- Disk great deal of interleaved, random i/o on disk
- Due to concurrent modification of document repository by multiple threads



## Non-blocking sockets

- Use non-blocking sockets
  - connect, send or recv call returns immediately without waiting for the network operation to complete
  - poll the status of the network operation separately
- Use "select" system call
  - monitor polls several sockets at the same time
  - event-based processing
- More efficient memory management
  - code that completes processing not interrupted by other completions
  - Single thread, no need for locks and semaphores on the pool
  - Only append complete pages to the log



## Link extraction & normalization

- Goal: Obtaining a canonical form of URL
- URL processing and filtering
  - Avoid multiple fetches of pages known by different URLs (which are actually the same resource)
    - Can't just use IP address of host
      - Many IP addresses for a single host
      - Many hostnames for a single IP address
      - Use canonical hostname from DNS
    - Relative URLs need to be interpreted w.r.t. to a base URL (../pubs/index.html)



## Canonical URL

- Formed by
  - Using a standard string for the protocol
  - Canonicalizing the host name (single case)
  - Adding an explicit port number
  - Normalizing and cleaning up the path
- Example
  - <http://www.Science.UvA.nl/%7mdr/...>
  - <http://www.science.uva.nl/~mdr/...>



## Robot exclusion

- Check
  - Whether the server prohibits crawling the URL
  - In /robots.txt file (in the HTTP root directory)
  - Specifies a list of path prefixes which crawlers should not attempt to fetch
- Robots META Tag within HTML doc
  - `<META NAME="ROBOTS" CONTENT="NOINDEX,NOFOLLOW">`
- Resources:
  - <http://www.robotstxt.org/wc/exclusion.html>
  - <http://www.kollar.com/robots.html>

## More concerns: refreshing crawled pages

- Search engine's index should be fresh
- Web-scale crawler never 'completes' its job
- High variance of rate of page changes
- "If-modified-since" request header with HTTP protocol
  - Impractical for a crawler: still need DNS lookups and HTTP requests
- Solution
  - At commencement of new crawling round, estimate which pages have changed
  - Crawling policy: pages that change more frequently are revisited more often

## Determining page changes

- "Expires" HTTP response header
  - For page that come with an expiry date
- Otherwise need to guess if revisiting that page will yield a modified version
  - Score reflecting probability of page being modified
  - Crawler fetches URLs in decreasing order of score
  - Assumption: **recent past predicts the future**

## Estimating page change rates

- Brewington & Cybenko/Cho & Garcia-Molina
  - Algorithms for maintaining a crawl in which most pages are fresher than a specified epoch
- Prerequisite
  - Average interval at which crawler checks for changes is smaller than the inter-modification times of a page
- Use small scale intermediate crawler runs
  - To monitor fast changing sites
  - E.g.: current news, weather, etc.
  - Patch intermediate indices into master index

## Other Concerns

- Frames
  - Might want to follow subframe links (iframes?)
  - Consider associating text on subframes with outerframe URL
- Redirection:
  - Many methods
    - Within HTTP (e.g. <http://www.science.uva.nl/>)
    - Within HTML META tags
    - With JavaScript
- Index text? Associate text with which URL?

## Eliminating already-visited URLs

- Checking if a URL has already been fetched
  - Before adding a new URL to the work pool
  - Needs to be quick!
- Achieved by computing hash function (e.g., MD5) on URL
- 10 billion URLs  $\times$  64 bits = 80GB for hash?
- Exploiting spatio-temporal locality for disk access
  - Two-level hash function.
    - most significant bits (say, 24) derived by hashing the host name plus port
    - lower order bits (say, 40) derived by hashing the path
    - concatenated bits used as a key in a B-tree
- Qualifying URLs added to frontier of the crawl
- Hash values added to B-tree

## Spider traps

- Protecting from crashing on
  - Ill-formed HTML
  - E.g.: page with 68 kB of null characters
- Misleading sites
  - indefinite number of pages dynamically generated by CGI scripts
    - dynamically generated by CGI scripts
    - dynamically generated by CGI scripts
  - paths of arbitrary depth created using soft directory links and path remapping features in HTTP server

## Spider Traps: Solutions

- No automatic technique can be foolproof
- Check for URL length
- Guards
  - Preparing regular crawl statistics
  - Excluding dominating sites by the guard module
  - Disable crawling active content such as CGI form queries
  - Eliminate (and don't parse!) URLs that return non-textual data types

## Load monitor

- Keeps track of various system statistics
  - Recent performance of the wide area network (WAN) connection
  - E.g.: latency and bandwidth estimates
- Operator-provided/estimated upper bound on open sockets for a crawler
- Current number of active sockets

## Per-server work queues

- Avoid creating a Denial of Service (DoS) attack
  - Limit the number of active requests to a given server IP address at any time
  - Maintain a queue of requests for each server
- Use the HTTP/1.1 persistent socket capability.
  - Distribute attention relatively evenly between a large number of sites
  - Place delays between subsequent requests (10 sec)
  - Consider remote resources (per country, per org)
- Access locality vs. politeness dilemma
  - most links are inter-site links, but we should not follow them all at once

## Text repository

- Crawler's last task
- Dumping fetched pages into a repository
- Decoupling crawler from other functions for efficiency and reliability preferred
- Page-related information stored in two parts
  - meta-data
  - page content

## Storage of page-related info

- Meta-data
  - relational in nature
  - usually managed by custom software to avoid relational database system overheads
  - text index involves bulk updates
  - includes fields like content-type, last-modified date, content-length, HTTP status code, etc.

## Page contents storage

- Typical HTML Web page compresses to 2–4 kB (using zlib)
- File systems have a 4–8 kB file block size: too large – inefficient!
- Page storage managed by custom storage manager
  - Simple access methods for
    - Crawler to add pages
    - Subsequent programs (Indexer etc) to retrieve documents

## Page Storage

- Small-scale system
  - Repository fitting within the disks of a single machine
  - Use of storage manager (E.g.: Berkeley DB)
  - Manage disk-based databases within a single file
    - configuration as a hash-table/B-tree for URL access key
    - To handle ordered access of pages
    - configuration as a sequential log of page records.
    - Since Indexer can handle pages in any order
- Large-scale system (Google File System)
  - Repository distributed over a number of servers
  - Connected to the crawler through a fast local network
  - *hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients*

## Available code

- Reference implementations of the HTTP client protocol from the World-wide Web Consortium <http://www.w3c.org/>
- w3c-libwww package
- simple, single-threaded
- Elsewhere, java, perl libraries available
- No freely available implementations of web scale crawlers

## Wrapping up crawling

- Crawling is about fetching your data
- Essential step for web retrieval/mining
- Easy to do small-scale
- Very serious engineering effort on web scale

## Coming up

- ✓ Crawling
- Document processing
- Indexing