Crawling

October 6, 2021
Midterm 1

- In class on Wed. Oct 13
  - 75 minutes
  - If you’re remote, email me to set up alternative arrangements

- I will release a previous exam on Piazza
  - Note that the format and style will be different!

- Format:
  - Multiple choice and short answer
  - Includes everything from lecture (slides, things I said, etc.)
  - Also includes readings and project
Midterm 1

You are allowed a 1 page (front/back) cheat sheet

- Handwritten
- OR 11-point Times New Roman, 1 in margins, single spaced, default kerning
- Write your name on it and turn it in with your test
- No other assistance! No tech, no wandering eyes, no talking to your friends
Plan for today

- Document Object Model (DOM)
- XPath
- Basic crawling
- Mercator
- Publish/subscribe
- Streaming XML
- XFilter
The Basic Crawler is Simple

Basic process:

1. Initialize Q with a set of seed URLs
2. Pick the first URL from Q and download the corresponding page
3. Extract all URLs from the page (<base href> tag, anchor links, CSS, DTDs, scripts, optionally image links)
4. Append to Q any URLs that a) meet our criteria, and b) are not already in P
5. If Q is not empty, repeat from step 2

But we need to be a good citizen and understand there are malicious actors
Crawling complications

- What order to traverse in?
  - Polite to do BFS - why?

- Be robust to malicious pages
  - Spam pages / SEO
  - Spider traps (incl. dynamically generated ones)

- Be robust to general messiness
  - Cycles, site mirrors, duplicate pages, aliases
  - Varying latency and bandwidth to remote servers
  - Broken HTML, broken servers, ...
  - Web masters' stipulations

- How deep to crawl? How often to crawl?
  - Continuous crawling; freshness
Robustness

- Must be able to:
  - Handle badly formed HTML (modern parsers, e.g. JSoup, can do this)
  - Handle corrupt request/response
  - Handle dynamically generated “spider traps”

- In fact we can think of robustness of the search engine itself, to quirks in the data for Search Engine Optimization (SEO)
SEO: "White-hat" version

- There are several ways you can make your web page easier to crawl and index:
  - Choose a good title
  - Use `<meta>` tags, e.g., description
  - Use meaningful URLs with actual words
    - BAD: http://xyz.com/BQF/17823/bbq
  - Use mostly text for navigation (not flash, JavaScript, ...)
  - Descriptive file names, anchor texts, ALT tags
  - Have a fast / secure / mobile-friendly website!

- More information from search engines, e.g.:
  - https://www.microsoft.com/web/seo
SEO: "Black-hat" version

- Tries to trick the search engine into ranking pages higher than it normally would:
  - Doorway pages
  - Keyword stuffing
  - Hidden or invisible text
  - Link farms
  - Scraper sites
  - Page hijacking
  - Blog/comment/wiki spam
  - Cloaking
  - Shadow domains
  - ...

© 2021 A. Haeberlen, Z. Ives, V. Liu

University of Pennsylvania
Normalization and eliminating duplicates

- Some of the extracted URLs are relative URLs
  - Example: /~cis455/
  - Normalize it: http://www.cis.upenn.edu:80/~cis455/

- Duplication is widespread on the web
  - e.g., many copies of GPL or Apache license, default pages, etc.
  - If the fetched page is already in the index, do not process it
  - Can verify using document fingerprint (hash) or shingles
Crawler Etiquette – Or, how to not get IP-blocked by the world

- **Explicit politeness**
  - Look for meta tags; for example, ignore pages that have `<META NAME="ROBOTS" CONTENT="NOINDEX">`
  - Implement the robot exclusion protocol; for example, look for, and respect, robots.txt

- **Implicit politeness**
  - Even if no explicit specifications are present, do not hit the same web site too often
Robots.txt

What should be in robots.txt?

- See http://www.robotstxt.org/robotstxt.html

- To exclude all robots from a server:
  
  User-agent: *
  Disallow: /

- To exclude one robot from two directories:
  
  User-agent: BobsCrawler
  Disallow: /news/
  Disallow: /tmp/

- Can also have Allow, which overrides Disallow

http://www.cis.upenn.edu/robots.txt
Recap: Crawling

- How does the basic process work?

- What are some of the main challenges?
  - Duplicate elimination
  - Politeness
  - Malicious pages / spider traps
  - Normalization
  - Scalability
Plan for today

- Basic crawling
- Mercator
- Publish/subscribe
- Streaming XML
- XFilter
Mercator: A scalable web crawler

- Written entirely in Java

- Expands a “URL frontier”
  - Avoids re-crawling same URLs

- Also considers whether a document has been seen before
  - Same content, different URL [when might this occur?]
  - Every document has signature/checksum info computed as it’s crawled
1. Dequeue frontier URL
2. Fetch document
3. Record into RewindInputStream (RIS)
4. Check against fingerprints to verify it’s new
5. Extract hyperlinks
6. Filter unwanted links
7. Check if URL repeated (compare its hash)
8. Enqueue URL
Mercator’s polite frontier queues

- Tries to go beyond breadth-first approach
  - Goal is to have only one crawler thread per server
  - What does this mean for the load caused by Mercator?

- Distributed URL frontier queue:
  - One subqueue per worker thread
  - The worker thread is determined by hashing the hostname of the URL
    - Thus, only one outstanding request per web server
  - This is our first instance of **sharding** of data – the same value always maps to the same node!
Mercator’s HTTP fetcher

- First, needs to ensure robots.txt is followed
  - Caches the contents of robots.txt for various web sites as it crawls them
- Designed to be extensible to other protocols

- Had to write own HTTP requestor in Java – their Java version didn’t have timeouts
  - Today, can use setSoTimeout()
- Could use Java non-blocking I/O:
  - [http://www.owlmountain.com/tutorials/NonBlockingIo.htm](http://www.owlmountain.com/tutorials/NonBlockingIo.htm)
  - But they use multiple threads and synchronous I/O
- Multi-threaded DNS resolver
Other caveats

- Infinitely long URL names (good way to get a buffer overflow!)
- Aliased host names
- Alternative paths to the same host
  - Can catch most of these with signatures of document data (e.g., MD5)

- Crawler traps (e.g., CGI scripts that link to themselves using a different name)
  - May need to have a way for human to override certain URL paths – see Section 5 of paper
  - Checkpointing!!
Further considerations

- May want to prioritize certain pages as being most worth crawling
  - Focused crawling tries to prioritize based on relevance

- May need to refresh certain pages more often

- General approach: Data partitioning
  - By domain, to coordinate activity with many machines
  - The same trick works on each server: separate subqueue for each domain
  - Then, can use priority queues or round-robin to visit pages while respecting politeness
Mercator document statistics

Histogram of document sizes

(60M pages)
Where to go from here

- You will need to build a crawler for HW2MS1 and for the final project

- Please learn from others' experiences!
  - Several crawling-related papers are linked from the reading list - e.g., the Google paper, the Mercator paper, ...
  - Reading these papers carefully before you begin will save you a lot of time
    - Get a sense of what to expect
    - Avoid common problems and bottlenecks
    - Identify designs that won't work well
Plan for today

- Basic crawling
- Mercator
- Publish/subscribe
- Streaming XML
- XFilter
Consider Some Common Tasks

- We want to monitor Twitter for #mentions of our product that are retweeted at least twice
- We have a set of memes we are tracking in Facebook
- We want to collect articles on our favorite band posted on popular news sites

- Can we develop a general solution to these tasks?
The publish/subscribe model

- Each **publisher** produces **events**
  - Example: Web page update, stock quote, announcement, ...
- Each **subscriber** wants a subset of the events
  - But usually not all
- **How do we implement this efficiently?**
Pub/Sub: great for event-driven architectures

- Fits well with the event driven paradigm
- Used for:
  - News publication, announcements, ...
  - Microservices / serverless compute
  - Stream Processing
More generally: Stream processing

- Stream engines take one item at a time, i.e., details about an event

- They feed to a set of queries that process each item as it comes
  - The queries may maintain state from item to item
  - They may output results, possibly as streams
An Example Stream

<table>
<thead>
<tr>
<th>User</th>
<th>Reply-To</th>
<th>Message-ID</th>
<th>Tweet</th>
</tr>
</thead>
<tbody>
<tr>
<td>@engineers_feed</td>
<td></td>
<td>1368411082264547333</td>
<td>On Mar 7, 1876 – <strong>Alexander Graham Bell</strong> is granted a patent for an invention he calls the &quot;telephone&quot;.</td>
</tr>
<tr>
<td>@HarriLuuppalaa</td>
<td>13684110822645 47333</td>
<td>1368444542148575232</td>
<td>The very same morning Elisha Gray filed his patent. Bell’s lawyer had to wait, that invention was first patented in Britain. Bell got patent in three weeks! Me an my fellow worker applied a patent some years ago an it took some years. Perhaps colors takes more time :-)</td>
</tr>
<tr>
<td>@Pilsner_Maxwell</td>
<td>13684110822645 47333</td>
<td>1368420923607486466</td>
<td>Good patent... but does it beat &quot;Method of exercising a cat&quot;? <a href="https://patents.google.com/patent/US5443036A/en">https://patents.google.com/patent/US5443036A/en</a></td>
</tr>
</tbody>
</table>
One Stream Processor: Apache Storm

Three main abstractions:

- A set of operations implemented with classes -- spouts and bolts
- Multiple workers running in different threads/machines that instantiate the operations
- A topology that defines how the operations logically connect
  - Includes information about which data goes to which workers

Twitter Spout → Noun Extractor Bolt → Noun Counter Bolt → Report Bolt
On Mar 7, 1876 – Alexander Graham Bell is granted a patent for an invention he calls the "telephone".
Plan for today

- Publish/subscribe
- Streaming XML
- XFilter
- Google File System
Variations on Matching Streams of Tweets

- Extracting parts of speech from Twitter is hard
  - Typos, nonstandard abbreviations, etc.
  - Requires a special parser

- But we’ve seen other versions where parsing of content is easy – say, XML!
Example: RSS

- Web server publishes XML file with events
- Clients periodically request the file to see if there are new events
- Is this a good solution?
What If I Want XML Data (or XML with particular Data)?

May want to retrieve documents with particular patterns

- All articles
- Articles about sports
- etc.

```
Crawler
  "spout"
```

```
Filter on documents
  "bolt"
```
Suppose we want to crawl XML documents based on user interests

We need several parts:

- A list of “interests” – expressed in an executable form, perhaps XPath queries
- A crawler – goes out and fetches XML content
- A filter / routing engine – matches XML content against users’ interests, sends them the content if it matches