CS 105: Data Gathering

Outline

- What is Web Crawling?
- How to build a web crawler
- Scrappy – off the shelf web-crawler
- Data gathering using APIs

What is Web Crawling?

- A **Web crawler**, sometimes called a **spider**, is a bot that systematically browses the WWW (by following a link from one page to another) to perform Web indexing (learn about web-pages and the graph network) and to scrape data from websites.

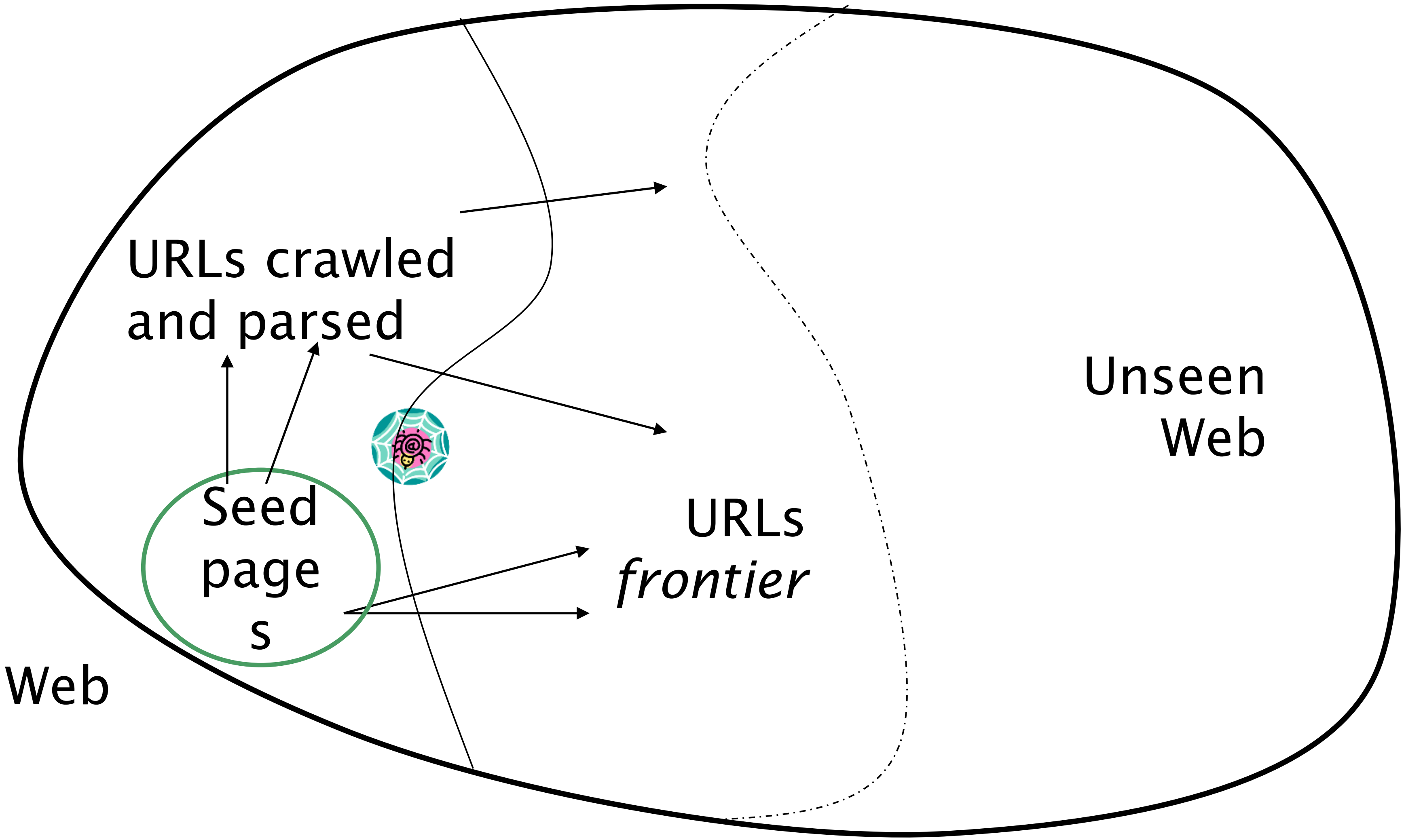
- A **Web scraper** is a specialized tool designed to accurately and quickly extract data from a web page.

- So the difference is that the web-crawler locates web-pages, and a scraper simply extracts the content of interest from a given page.

Basic crawler operation

- Begin with known “seed” URLs
- Fetch and parse them
  - Extract URLs they point to
  - Place the extracted URLs on a queue
- Fetch each URL on the queue and repeat

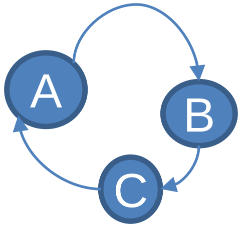
Crawling picture



Challenges faced when building a crawler

- Web crawling isn't feasible with one machine
  - Must build a robust distributed web-crawler

- Robust to malicious pages
  - Spam pages
  - Spider traps



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

- Even non-malicious pages pose challenges
  - Latency/bandwidth to remote servers vary
  - Webmasters' stipulations
    - How "deep" should you crawl a site's URL hierarchy?

What any crawler must do

- Avoid crawling duplicate pages / URLs
    - Duplicates can imply duplicate URLs that can be collected from the distributed set of machines that

- Be Polite: Respect implicit and explicit politeness considerations
  - Only crawl allowed pages
  - Respect robots.txt (more on this shortly)

What any crawler should do

- Be capable of distributed operation: designed to run on multiple distributed machines

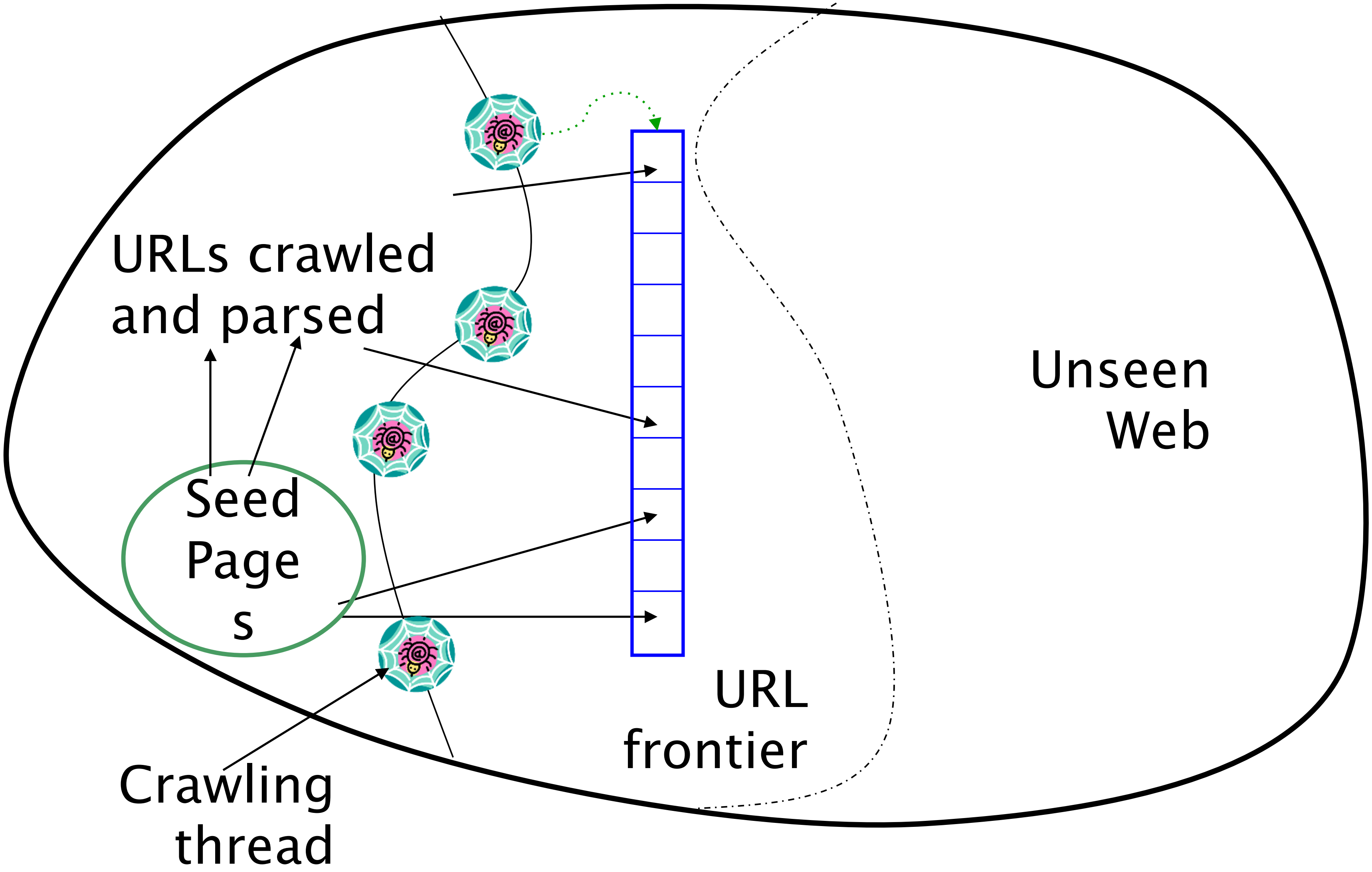
- Be scalable: designed to increase the crawl rate by adding more machines

- Performance/efficiency: permit full use of available processing and network resources

- Fetch pages of “higher quality” first

- Continuous operation: Continue fetching fresh copies of a previously fetched page

Updated crawling picture



URLs crawled and parsed

Seed Page s

Crawling thread

URL frontier

Unseen Web

Explicit and implicit politeness

- What is URL frontier?
  - Can include multiple pages from the same host
  - Must avoid trying to fetch them all at the same time

- 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

Robots.txt

- Protocol for giving spiders ("robots") limited access to a website, originally from 1994
  - www.robotstxt.org/wc/norobots.html

- Website announces its request on what can(not) be crawled
  - For a server, create a file /robots.txt
  - This file specifies access restrictions

Robots.txt example

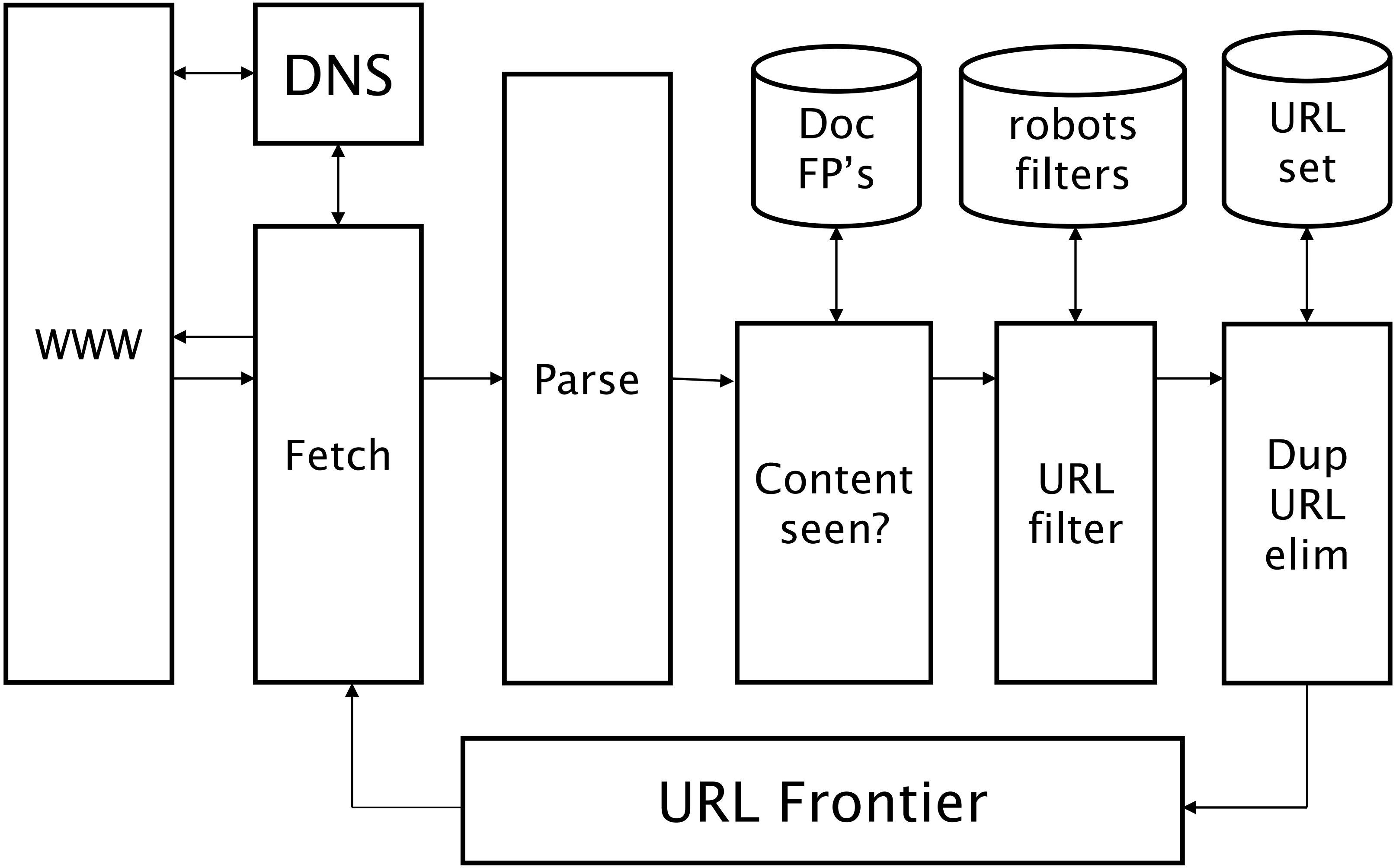
- No robot should visit any URL starting with "/yoursite/temp/", except the robot called “searchengine":

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

User-agent: searchengine
Disallow:
```

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)

Basic crawl architecture



Content seen?

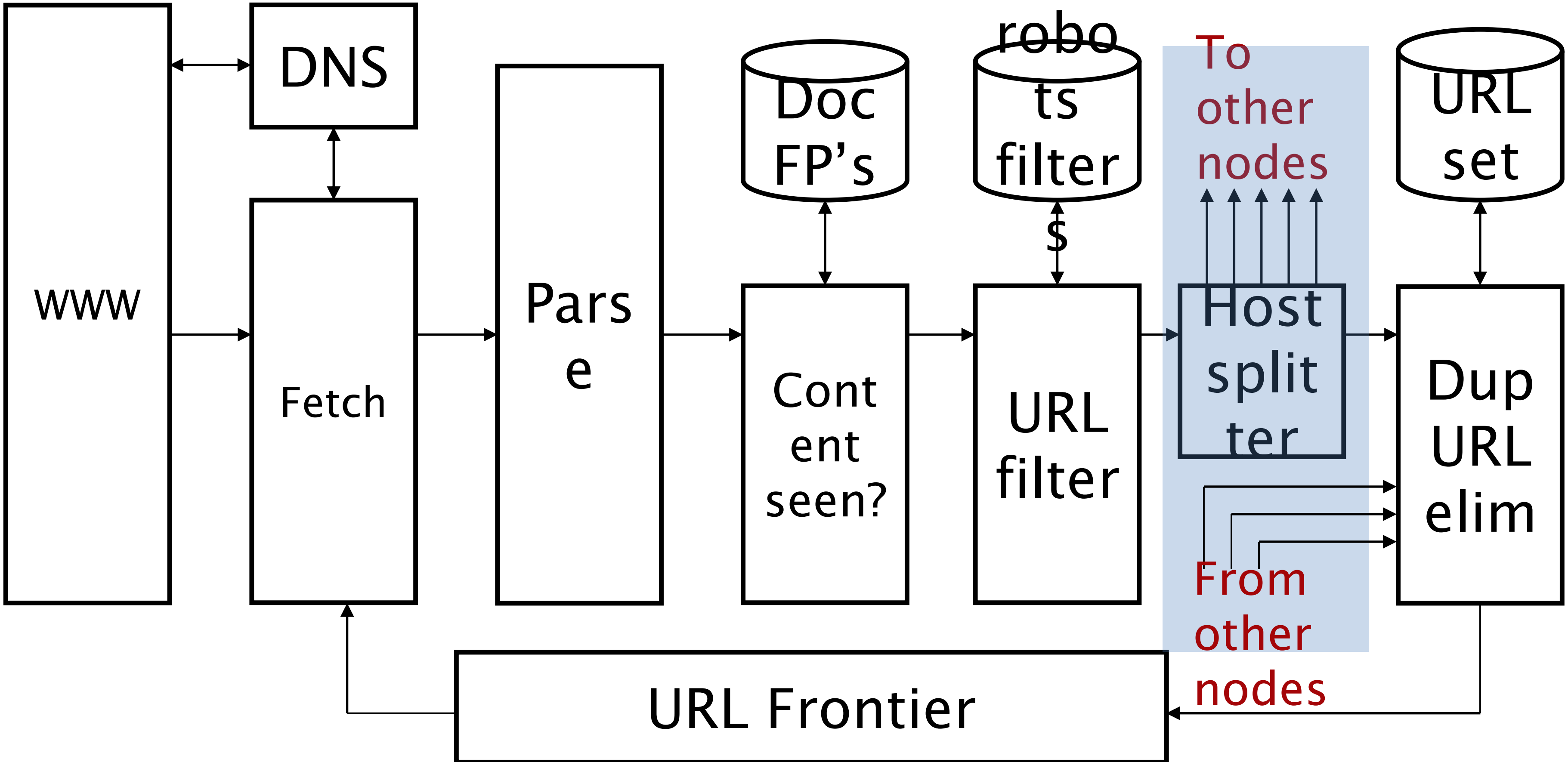
- Duplication is widespread on the web
- If the page just fetched is already in the index, do not further process it
- This is verified using document fingerprints or shingles

Distributing the crawler

- Run multiple crawl threads, under different processes – potentially at different nodes
  - Geographically distributed nodes
- Partition hosts being crawled into nodes
  - Hash used for partition
- How do these nodes communicate and share URLs?

Communication between nodes

- Output of the URL filter at each node is sent to the Dup URL Eliminator of the appropriate node



URL frontier: two main considerations

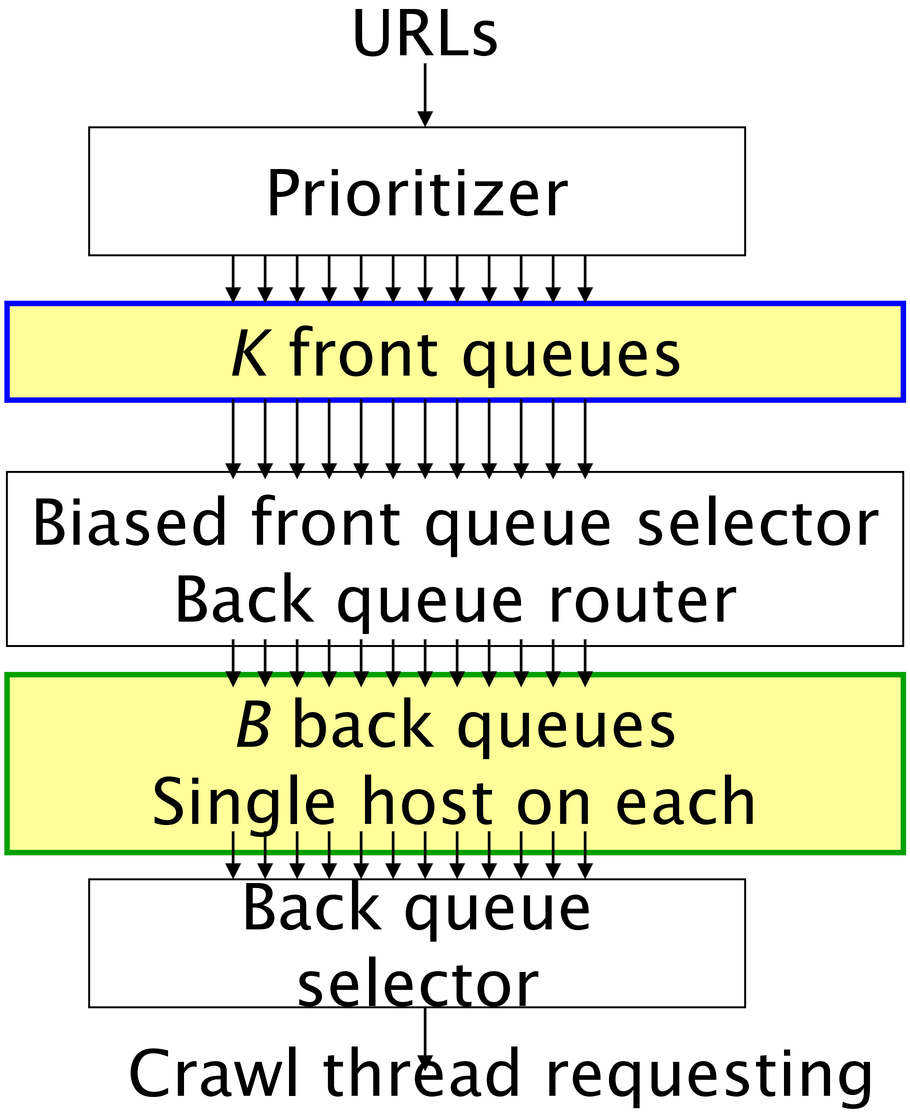
- Politeness: do not hit a web server too frequently

- Freshness: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

These goals may conflict 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.)

URL frontier: Mercator scheme



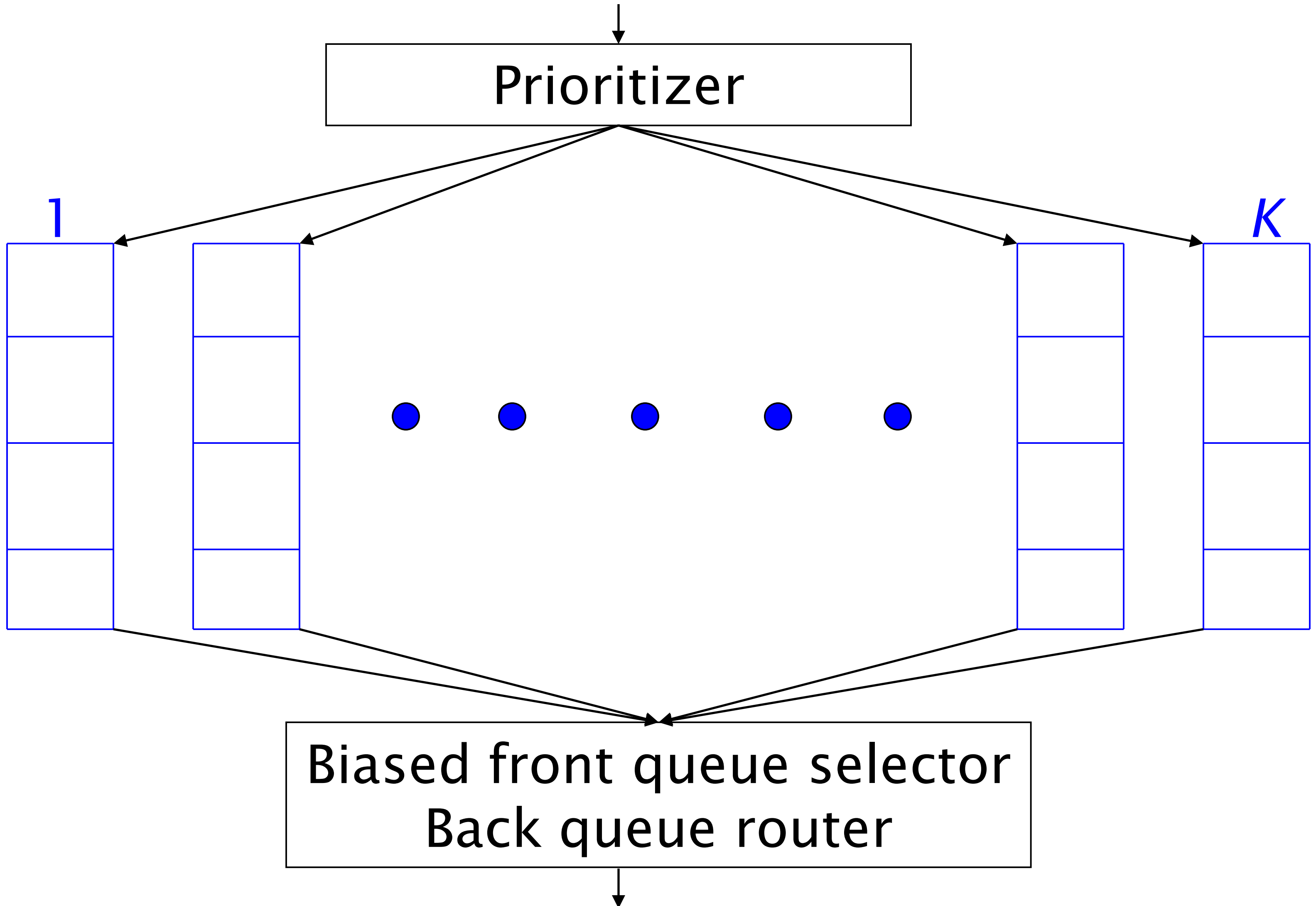
URLs flow in from the top into the frontier

Front queues manage prioritization

Back queues enforce politeness

Each queue is FIFO

Front queues



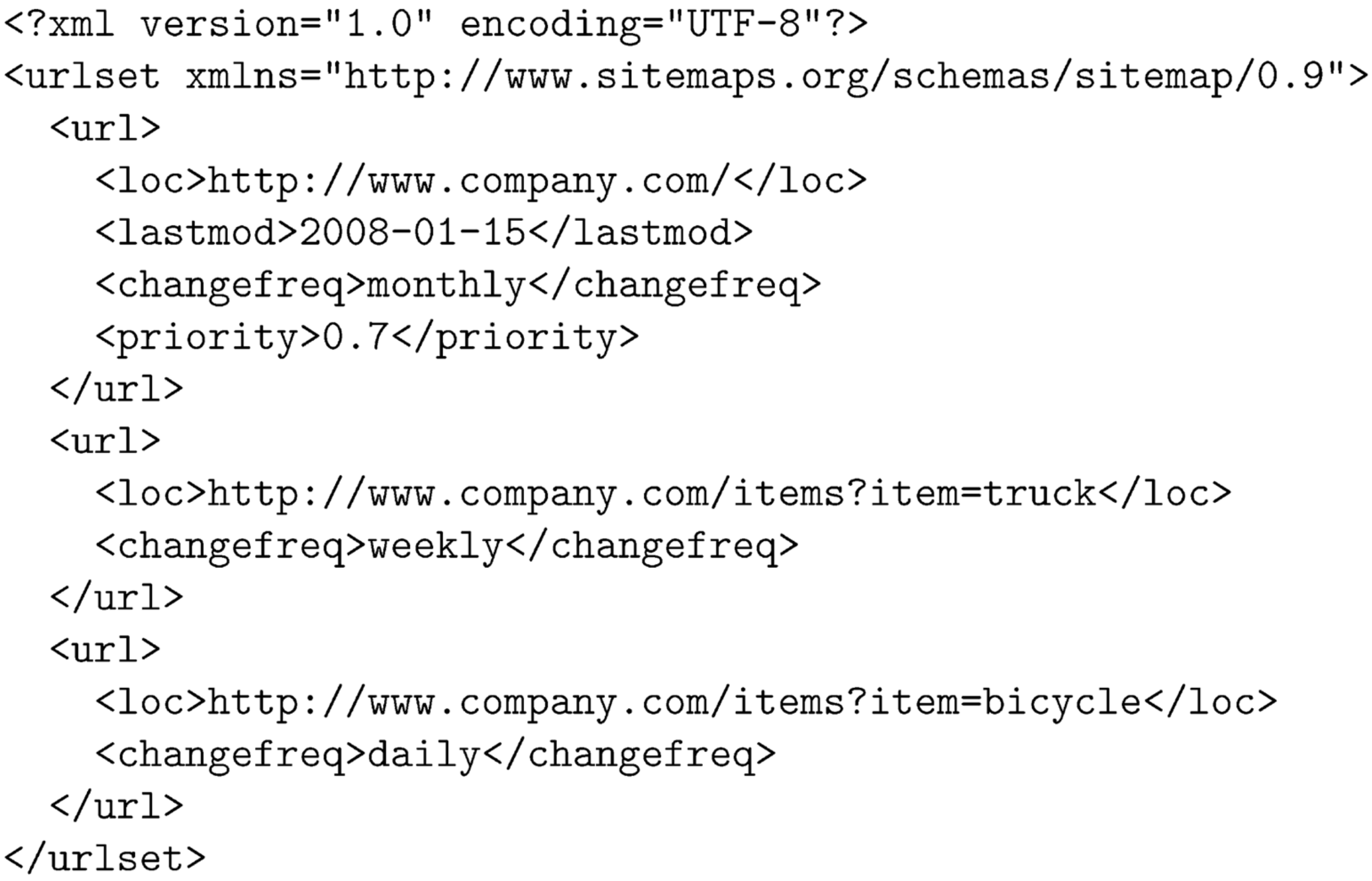
Prioritizer

Biased front queue selector
Back queue router

Sitemaps

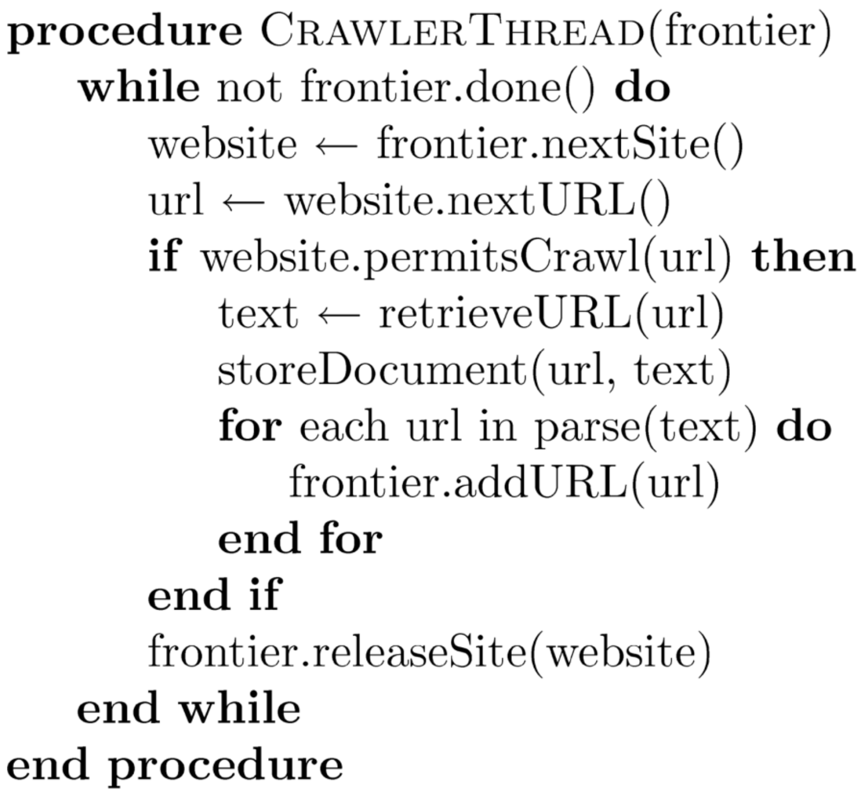
- Sitemaps contain lists of URLs and data about those URLs, such as modification time and modification frequency
- Generated by web server administrators
- Tells crawler about pages it might not otherwise find
- Gives crawler a hint about when to check a page for changes

Sitemap Example



```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.company.com/</loc>
    <lastmod>2008-01-15</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.7</priority>
  </url>
  <url>
    <loc>http://www.company.com/items?item=truck</loc>
    <changefreq>weekly</changefreq>
  </url>
  <url>
    <loc>http://www.company.com/items?item=bicycle</loc>
    <changefreq>daily</changefreq>
  </url>
</urlset>
```

Simple Crawler Thread



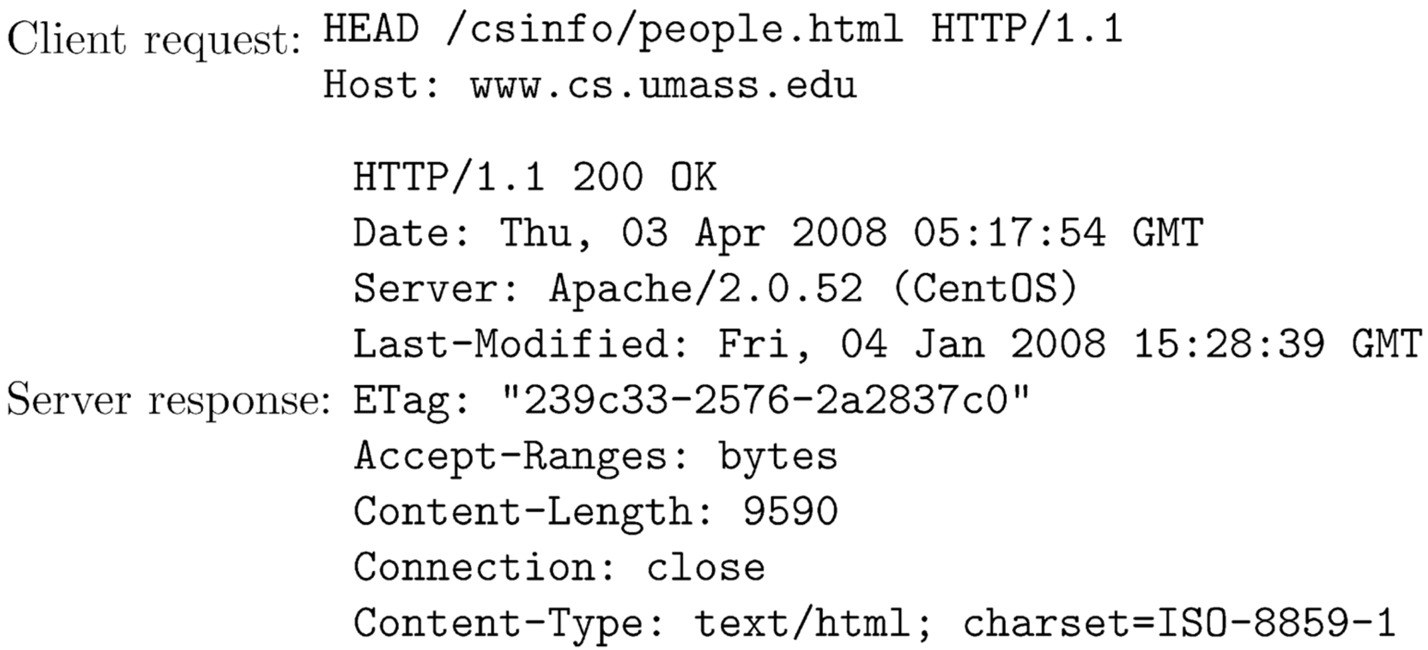
```
procedure CrawlerThread(frontier)
    while not frontier.done() do
        website ← frontier.nextSite()
        url ← website.nextURL()
        if website.permitsCrawl(url) then
            text ← retrieveURL(url)
            storeDocument(url, text)
            for each url in parse(text) do
                frontier.addURL(url)
            end for
        end if
        frontier.releaseSite(website)
    end while
end procedure
```

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
```

1) Scrapy

Open source web spider written in Python

2) PySpider

3) MechanicalSoup + BeautifulSoup

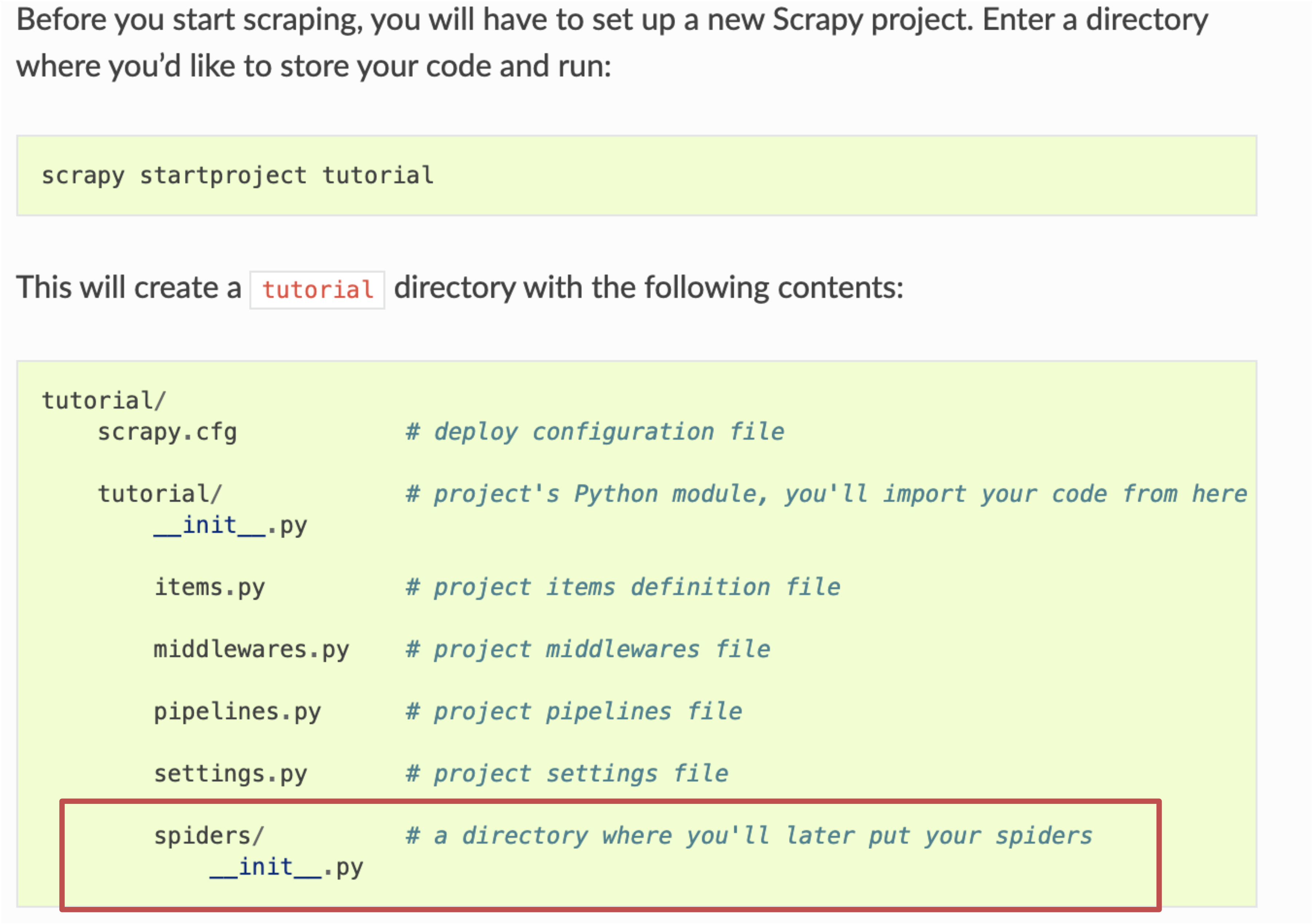
What is Scrapy

- Scrapy is a open source and collaborative framework for crawling the web

- Scrapy is an excellent choice for focused crawls
  - Scrapy is generally faster than other open source crawlers
  - Scrapy is written in Python

- Scrapy is written in pure Python and depends on a few key Python packages (among others):
  - lxml, an efficient XML and HTML parser
  - parsel, an HTML/XML data extraction library written on top of lxml,
  - w3lib, a multi-purpose helper for dealing with URLs and web page encodings
  - twisted, an asynchronous networking framework
  - cryptography and pyOpenSSL, to deal with various network-level security needs

Tutorial: https://docs.scrapy.org/en/latest/intro/install.html#intro-install



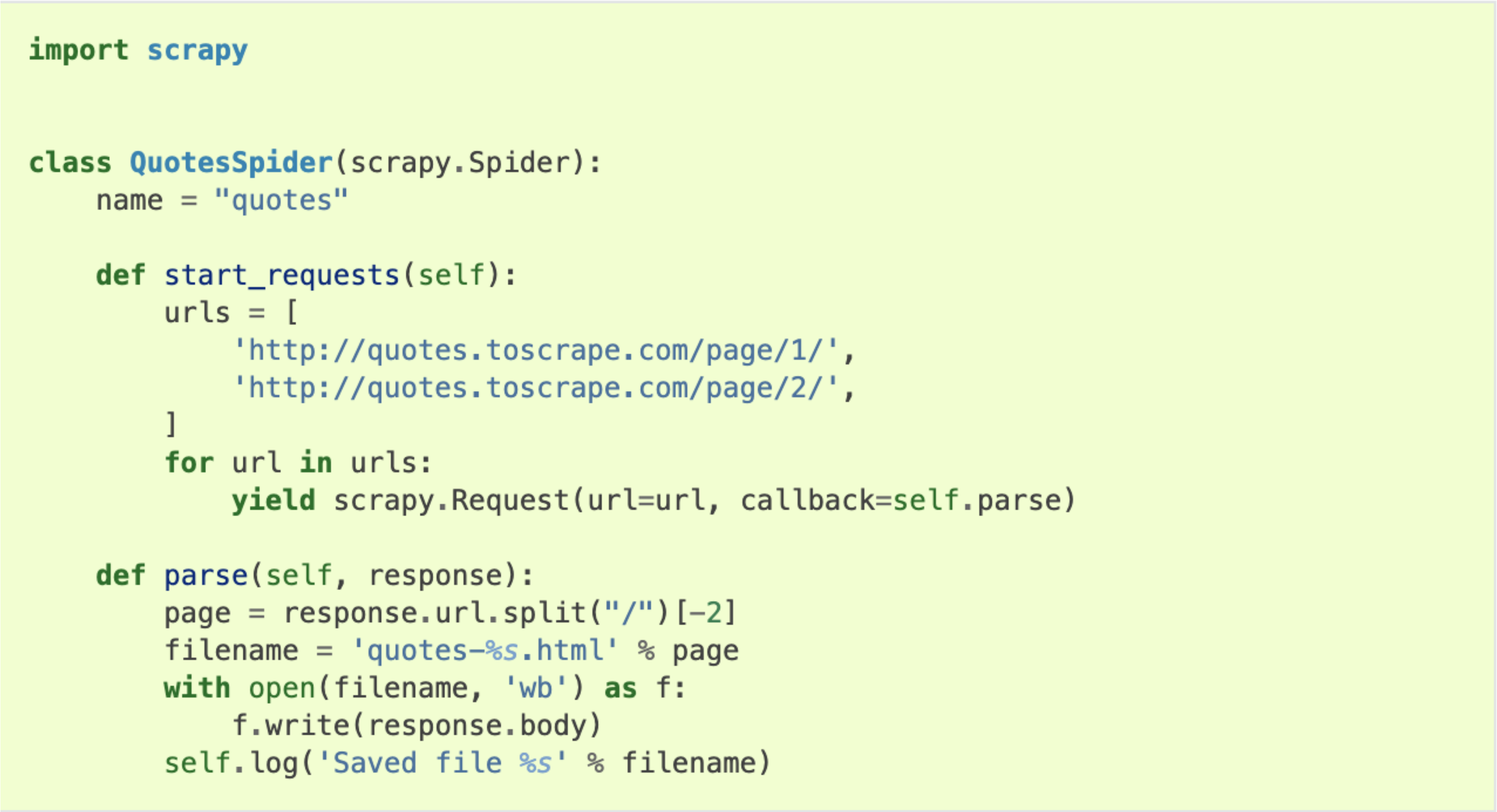
Before you start scraping, you will have to set up a new Scrapy project. Enter a directory where you'd like to store your code and run:

```
scrapy startproject tutorial
```

This will create a `tutorial` directory with the following contents:

```
tutorial/
    scrapy.cfg            # deploy configuration file

    tutorial/             # project's Python module, you'll import your code from here
        __init__.py

        items.py          # project items definition file

        middlewares.py    # project middlewares file

        pipelines.py      # project pipelines file

        settings.py       # project settings file

        spiders/          # a directory where you'll later put your spiders
            __init__.py
```

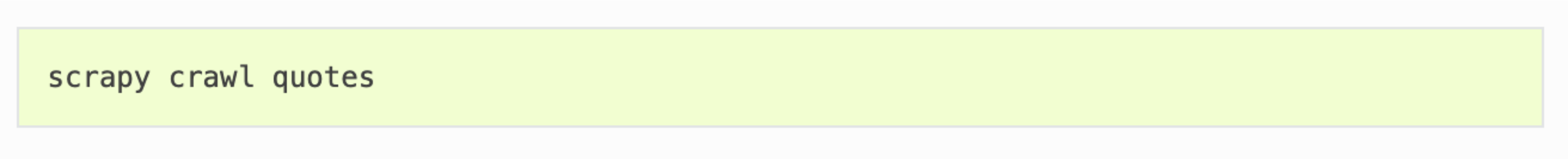
We write our spiders in this directory



```python
import scrapy


class QuotesSpider(scrapy.Spider):
    name = "quotes"

    def start_requests(self):
        urls = [
            'http://quotes.toscrape.com/page/1/',
            'http://quotes.toscrape.com/page/2/',
        ]
        for url in urls:
            yield scrapy.Request(url=url, callback=self.parse)

    def parse(self, response):
        page = response.url.split("/")[-2]
        filename = 'quotes-%s.html' % page
        with open(filename, 'wb') as f:
            f.write(response.body)
        self.log('Saved file %s' % filename)
```

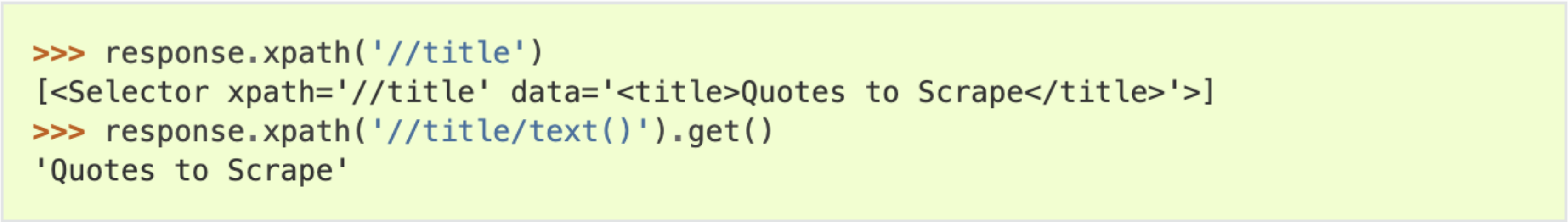
Spiders are classes that you define and that Scrapy uses to scrape information from a website (or a group of websites). They must subclass Spider and define the initial requests to make, optionally how to follow links in the pages, and how to parse the downloaded page content to extract data.

- Run the spider by going to the project top level directory and run



```
scrapy crawl quotes
```

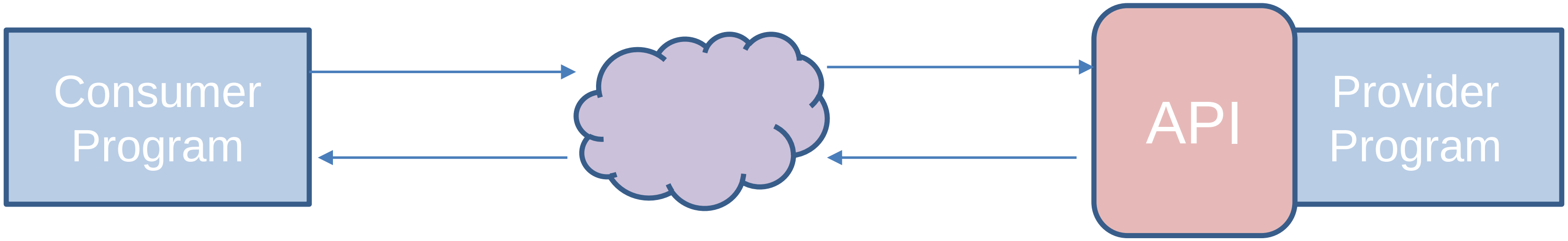
- The next step is to write code to extract select content from the page.
- This is accomplished by selecting elements using CSS, or Xpath
  - XPath is a expression that allows you to select elements based on structure as well as content.



```
>>> response.xpath('//title')
[<Selector xpath='//title' data='<title>Quotes to Scrape</title>'>]
>>> response.xpath('//title/text()').get()
'Quotes to Scrape'
```

Other ways to acquire data

- In some cases, companies make their data and functionality available via APIs.
- What is an API?
  - Formally, Application Programming Interface (API) is a specification intended to be used as an interface by software components to communicate with each other.



A bit about APIs

Private

Private APIs are used internally to facilitate the integration of different applications and systems used by a company.

Partner

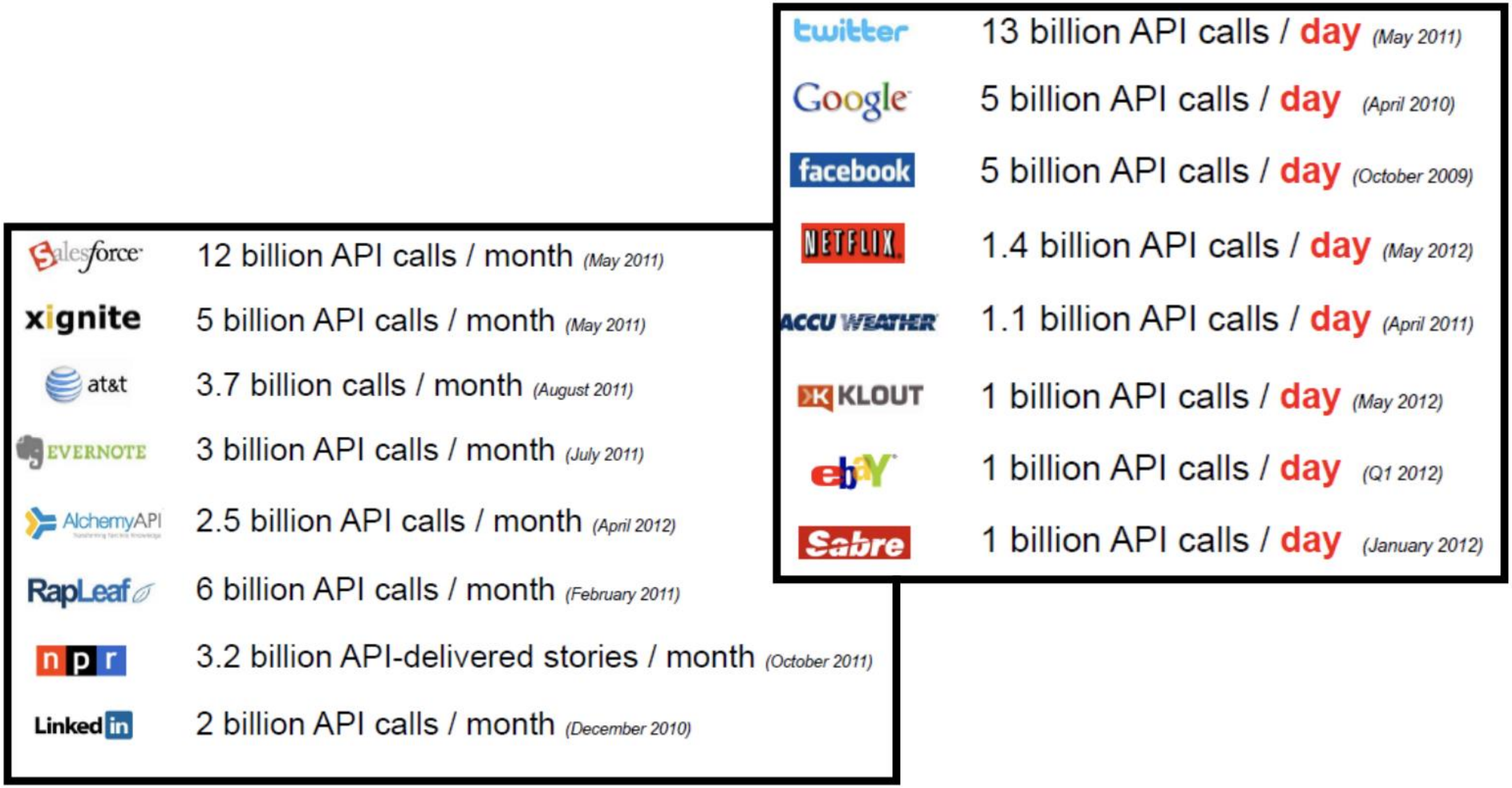
Partner APIs are used to facilitate communication and integration of software between a company and its business partners.

Public

Public APIs allow companies to publicly expose information and functionalities of one or various systems and applications to third parties that do not have a relationship.

Ex. Release data so developers can create an app

Extensive use of APIs

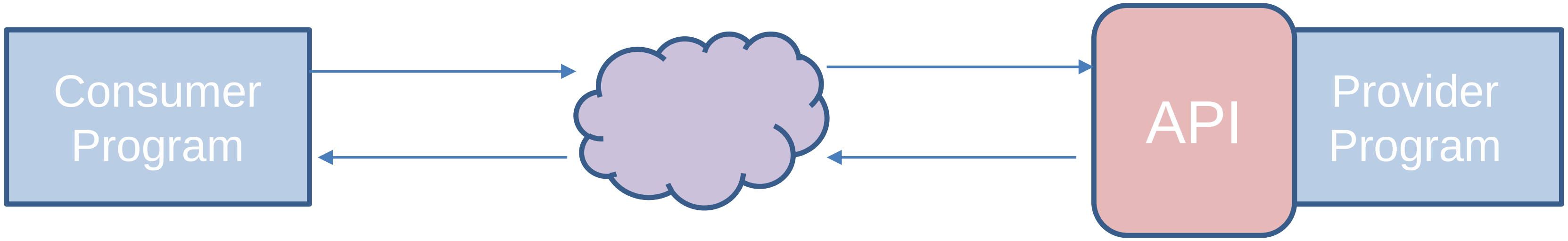


twitter	13 billion API calls / **day** *(May 2011)*
Google	5 billion API calls / **day** *(April 2010)*
facebook	5 billion API calls / **day** *(October 2009)*
NETFLIX	1.4 billion API calls / **day** *(May 2012)*
ACCU WEATHER	1.1 billion API calls / **day** *(April 2011)*
KLOUT	1 billion API calls / **day** *(May 2012)*
ebY	1 billion API calls / **day** *(Q1 2012)*
Sabre	1 billion API calls / **day** *(January 2012)*

salesforce	12 billion API calls / month *(May 2011)*
xignite	5 billion API calls / month *(May 2011)*
at&t	3.7 billion calls / month *(August 2011)*
EVERNOTE	3 billion API calls / month *(July 2011)*
AlchemyAPI	2.5 billion API calls / month *(April 2012)*
RapLeaf	6 billion API calls / month *(February 2011)*
npr	3.2 billion API-delivered stories / month *(October 2011)*
Linked in	2 billion API calls / month *(December 2010)*

How to use an API

- APIs consist of three parts:
  - User: the person who makes a request
  - Client: the computer that sends the request to the server
  - Server: the computer that responds to the request



- APIs offer endpoints to obtain, change, or delete data
- There are four types of actions an API can take:
  - **GET:** requests data from a server — can be status or specifics (like last_name)
  - **POST:** sends changes from the client to the server; think of this as adding information to the server, like making a new entry
  - **PUT:** revises or adds to existing information
  - **DELETE:** deletes existing information

How to use an API

- Well, you need to read the documentation for the particular API you are interested in.

- Most APIs require an API key. Once you find an API you want to play with, look in the documentation for access requirements.

- The easiest way to start using an API is by finding an HTTP client online, like REST-Client, Postman, or Paw. These ready-made (and often free) tools help you structure your requests to access existing APIs with the API key you received.

- Once you pull data from the API, understand the format of the data (XML or JSON) and how to extract the data of interest.