Project 3 - High Throughput Web Crawler (Due 11:59PM, December 11, 2018)

Overview

Content is king on the Internet today, and business thrive or fail based on the value of content they collect, hoard, and serve to their users. Protecting this data is difficult, because the large majority of that content is available to users, which means it is available to automated web crawlers that mimic real users.

For this project, you will be implementing a parallel web crawler that maximizes throughput when retrieving content from a target website. This will be an adversarial interaction, between the crawler will be trying to pull down data from a server that is wary and prepared against aggressive crawlers. The web server will have implemented multiple layers of rate limits for incoming network connections, which the crawler must overcome if it wants to maximize its efficiency.

For this project, you will need learn how to interact with HTTP servers (like how you interact with FTP servers in Project 2). There are a set of interactions (again like project 2) which your crawler (HTTP client) will need to recognize/parse and respond).
You can start from https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol which provides the list of the commands.

There are three versions of the crawler that should be implemented as part of this assignment.

1. [40%] **Basic crawler.** Your crawler will connect to the target website, and fetch the main page. Once it retrieves the main page (index.html), it will parse the page and extract all links to objects and other pages, put them on the crawler queue, and crawl them in turn, repeating the process until all files and pages reachable from index.html in a transitive closure. To identify itself to the server, the crawler uses a single cookie with a user ID.

2. [40%] **Parallel flows vs. per-flow rate limits**. The server has per-flow rate limits that throttle any single network connection to a max rate (R), and that puts a ceiling on how fast the basic crawler can work. Version 2 of the crawler overcomes this limit by implementing parallel connections with coordination. In this mode, the crawler spawns multiple threads (like you guys did in assignment 2), where each thread cooperates with the others to speed up the download. Key difference is that all threads share a single download queue, and any parsed links to files or other pages gets put on the same shared queue. Implemented correctly, *N* multiple threads will ideally speed up the aggregate download rate by roughly a factor of *N,* since each thread can potentially max out throughput at R. Note that here, all threads identify themselves to the server using the same user cookie file.

3. [20%] **Shared cookies vs. per-user rate limits**. The server is smarter than you thought, and also implements a per-user rate limit, that limits the total download rate by any number of connections by the same user. This means that instead of getting *N*R* total throughput with your N threads, you're instead getting some total throughput *U*, where *U<N*R*. In order to overcome this defense, you need to retrieve multiple distinct cookie files from the server, to emulate multiple users simultaneously downloading from the server. You'll need to manage distinct cookies per thread, but still have the threads share a single download queue so they can cooperate. Implemented correctly, this allows your *N* threads to actually achieve the ideal aggregate throughput of *N*R*.

Implementation

The crawler you implement will take in a target server address (server name), and start fetching starting from the main default page (index.html). It will also take in as a parameter the maximum number of threads. It (and any threads it spawns) will write downloaded files to the local directory.

mcrawl [-n max-flows] [-h hostname] [-p port] [-f local-directory]

[-n] : number of maximum threads your crawler can

spawn to perform a cooperative crawl
[-h] : hostname of the server to crawl data from
[-p] : port number on the server where the web server is running
[-f]: local directory to host the download files

Cookie will be sent by the server (we set up and you can download using get). After the crawler user downloads it, stores locally, and sends to the server on demand to verify itself. The following link : https://en.wikipedia.org/wiki/HTTP_cookie provides more information about the use of cookies in HTTP.

Testing

We will put up a test server in the cs.uchicago domain (after Thanksgiving weekend). By default, you should assume that the server always runs with both per-flow and per-user rate limits turned on. We might run multiple servers on different ports where some servers run without per-user rate limits.

The amount of data on the server will be finite, and you can test the total performance of your crawler by the total time it takes to download all files from the server.

We are working with the dept staff to set up the server, and will provide more information about the server itself after thanksgiving. In the meanwhile, you can test your basic crawler design using any public server (e.g.

www.cnn.com), but I suggest to start from my lab server: http://sandlab.cs.uchicago.edu/ which does not require any cookie.

Submission

Again using a dropbox upload link (different if you do normal or extended submission):

normal submission

https://www.dropbox.com/request/Z0UHcwQj2RnbxlfNsDMD

with 2 day extension

https://www.dropbox.com/request/2IpQnRpSPt7BzsBn5pH5