CIS 455/555: Internet and Web Systems

Spring 2013

Team Project Specifications (Draft!)

Project plan due April 10, 2013 | Code due April 29, 2013 | Final report due May 7, 2013

1 Overview

For the term project, you will build a peer-to-peer Web indexer/crawler and analyze its performance. This will involve several components, each of which is loosely coupled with the others:

- Crawler
- Indexer/TF-IDF Retrieval Engine
- PageRank
- Search Engine and User Interface
- Experimental Analysis and Final Report

In addition, you may need to build a command line interface or an interactive menu to launch the components. More details on each component are provided below. The project is relatively open-ended and includes many possibilities for extra credit. However, you are strongly encouraged to get the basic functionality working first.

Suggested approach: Spend some time early coordinating with your group-mates and deciding which modules from your previous homework assignments are "best of breed." Designate one person to be responsible for each task.

Make sure adequate time is spent defining interfaces between components (in this case, appropriate interfaces might be Pastry messages, Web service calls, and perhaps common index structures), and also plan to spend significant time integrating, especially with EC2. As in previous projects, please consider the use of automated tools for building your project (ant) and for version control (cvs, subversion)

Note that the report includes a non-trivial evaluation component.

You may want to make use of (1) the VM and/or the spec cluster for debugging and development, (2) subversion for sharing your work, (3) JUnit tests for validating that the code works (or remains working), (4) Amazon EC2 to evaluate your system.

2 Project specifications

2.1 Crawler

The Web crawler should build upon your past homework assignments, and it should be able to parse typical HTML documents. It should check for and respect the restrictions in `robots.txt` and be wellbehaved in terms of concurrently requesting at most one document per hostname. Requests should be distributed, Mercator-style, across multiple crawling peers built over Pastry. The crawler should track visited pages and not index a page more than once.

Extra credit: Add support for digests to detect when the same document has been retrieved more than once, under different URLs. If so, the document should only be stored once – but two "hits" should be returned.

2.2 Indexer

The indexer should take words and other information from the crawler and create a **lexicon**, **inverted index**, and any other necessary structures for returning weighted answers making use of TF/IDF, proximity, and any other ranking features that are appropriate. It should be able to store data persistently across multiple nodes, using Pastry and BerkeleyDB.

Extra credit: Include document and word metadata that might be useful in creating improved rankings (e.g., the context of the words).

Extra credit: Make the indexing system restartable with a different number of peers. The *simplest* way to do this is to have a procedure where, at startup, each node with an existing BerkeleyDB index will (1) rename it, (2) create a new index file, and (3) scan through each entry in the old index file and PUT it into the DHT.

2.3 PageRank

Given information from crawling, you should perform **link analysis** using the PageRank algorithm, as discussed in class and in the Google PageRank paper.

Your implementation should make use of MapReduce to do the analysis. There are many ways of doing a distributed PageRank. A very simple one is described here:
http://www.cs.toronto.edu/~jasper/PageRankForMapReduceSmall.pdf

Extra credit. Other papers have slightly different approaches to distributed PageRank; you might consult these for alternative ideas.

http://www.cs.columbia.edu/~simha/cal/pubs/pdfs/distributedpagerank.pdf
http://wwwcsif.cs.ucdavis.edu/~yeshao/cikm05.pdf

Extra credit will be given for a performance comparison between different approaches.

2.4 Search Engine and Web User Interface

This component is fairly self-explanatory, as the goal is to provide a search form and weighted results list. One aspect that will take some experimentation is determining how much to weight each item (PageRank, TF/IDF, other word features). how much to rate each item!

Extra credit: Integrate Yahoo search results into your keyword listings, using the REST interfaces. A challenge here is how to interleave ranked results.

Extra credit: Integrate Amazon search results into your keyword listings, using the REST or SOAP interfaces. A challenge here is how to interleave ranked results, especially given that some topics may be more or less suited to Amazon.

Extra credit: Implement a simple Google-style spell-check: for words with few hits, try simple edits to the word (e.g., adding, removing, transposing characters) and see if a much more popular word is "nearby."

Extra credit: Consider adding AJAX (Asynchronous Javascript And XML) support to your search interface, so users can provide feedback about which entries are "good" or "bad," and use these to re-rank the results.

3 Experimental Analysis and Final Report

Building a Web system is clearly a very important and challenging task, but equally important is being able to convince others (your managers, instructors, peers) that you succeeded. We would like you to actually *evaluate* the performance of your methods, for instance relative to scalability.

For evaluation, you should log into multiple Amazon EC2 nodes and run the system.

One approach is to use the system with one, two, up to *n* peers (where *n* is, say, 10 EC2 nodes), and compare overall response or completion time. This can be done for crawling and for query answering. For the latter, you can write a simple query generating tool that poses many queries at once, and compare response time. What is the maximum number of concurrent requests you can reasonably handle, when varying the number of nodes? Can you separate out the overhead of the different components (including network traffic)?

Your final report (a PDF document of six pages or less, due on May 10) should include at least:

- Introduction: project goals, high-level approach, milestones, and division of labor
- Project architecture
- Implementation: non-trivial details
- Evaluation
- Conclusions

Note that the quality of the report *will* have substantial bearing on your grade: it is not simply something to be cobbled together at the last second!

4 Requirements

Your solution must meet the following requirements (please read carefully!):

1. Your project plan, due on April 10, 2013, must be a PDF file of two pages or less. It must contain the first two sections of the final report, i.e., the introduction, a description of the project architecture, some rough milestones, and a division of labor. One team member must submit the project plan via `turnin`, using the project name `projectplan`.
2. Your code submission, due on April 29, 2013, must contain a) the entire source code for the project, as well as any supplementary files needed to build your solution, and b) a README file. The README file must contain 1) the full names and SEAS login names of all the project members, 2) a description of all features implemented, 3) any extra credit claimed, 4) a list of source files included, and 5) detailed instructions on how to install and run the project. The code must contain a reasonable amount of documentation. The code must be submitted by one team member via turnin (!), using the project name `projectcode`.

3. Each team will need to schedule a project demo between April 30 and May 3.
4. The final report, due on May 7, 2013, must be a PDF file of six pages or less. It must include all the information from the project plan (possibly revised and/or with more details), plus a description of your implementation, your evaluation results, and your conclusions. One project member must submit the report via `turnin`, using the project name `projectreport`.

You may not use any third-party code other than the standard Java libraries, code from previous CIS455/555 homework submissions by you or your team members, and any code we provide or explicitly approve.

5 Hints

Based on my experience with last year's project, many teams tend to divide up the work at the beginning and then meet again a few days before the deadline. This rarely works well; most people tend to under-estimate the amount of work that is needed to integrate the various components at the end. To avoid this, I recommend that you

- Define clear interfaces at the beginning. Ideally, write a few unit tests, so that everyone under-stands what their component is supposed to do, and how it interacts with the other components.
- Do integration testing as early as possible. Ideally, everyone builds a very simple demo version of their component first, exchanges that code with the others, and then adds features one by one.
- Meet regularly and keep everyone posted on your progress.
- **Start early!!!**

An important factor in the evaluation of your search engine will be the quality of the results it returns. To get good results, you need a) a sufficiently large corpus, and b) a good ranking function. Thus, any attempt to do this assignment at the last minute is likely to end in disaster. Here are a few tips to help you avoid this:

- Try to have preliminary versions of all components ready at least two weeks before the deadline, so you have enough time to tweak the ranking function and to get good results.
- I recommend designing your crawler in such a way that it is possible to interrupt a crawl and to continue it later. That way, you can keep crawling while you're working on the other components.

Finally, please avoid using Pastry's key-based routing primitive for transferring large amounts of data between nodes - this is likely to cause performance problems!