Distributed Peer-to-Peer Metasearch Engine

CIS 555 Project, Spring 2007

Debajit Adhikary (debajit@seas.upenn.edu)
Khader Naziruddin (khadera@seas.upenn.edu)
Shalin J. Shah (shalinjs@seas.upenn.edu)

May 4, 2007

1 Introduction

Distributed Peer-to-Peer Metasearch Engine

Project Goals

The project builds a peer-to-peer implementation of a Google-style search engine, with distributed, scalable crawling, indexing with ranking, and integration of search results with a commercial search engine.

The broad goals of the project are as follows:

- To design a system which allows a user to search (a subset of) the Web by specifying one or more keywords.
- The search results should be of meaning to the user, in a manner suitable to most users.
- Irrelevant pages containing keywords should be accorded a lower significance. The most relevant pages should appear at the top of the search results page, for maximum benefit to the user.
- It should be easy to scale the system, or any component of it.
- The system should be reasonably self-sustaining with minimal manual intervention.

Other non-technical goals are:

- To design a distributed system that is fault-tolerant, robust and scalable, at the same time efficient enough to be used in a real-life scenario.

- To apply the concepts learnt in this *Internet & Web Systems* course and apply them firsthand in a project of practical significance.

2 Project Architecture

SYSTEM ARCHITECTURE & DESIGN

The project involves the following components, each of which is loosely coupled with the others:

- **CRAWLER.** The crawler is multithreaded, and has the following features:
  - It checks for and respects *robots.txt*
  - It is well-behaved in terms of requesting two or more documents from a server only after a configurable time interval (we used 100ms).
  - Requests are distributed, Mercator-style, across multiple crawling peers built over FreePastry.
  - The crawler maintains a list of visited pages, and does not crawl a page more than once.

- **FORWARD INDEXER.** The forward indexer module is responsble for the following:
  - It **downloads** a page (the page link is sent by the Crawler node), and extracts all links from it, adding them to a queue.
  - It performs **text-processing** on the page to extract words, and stores this information in the Forward Index Store.
  - It **analyzes word metadata** and performs **hit processing** in a manner analogous to what early Google used to do. Word metadata analyzed include the following:

WORD METADATA ANALYZED

- HTML tag information (For instance, text inside tags H1 though H6 are assigned higher weights)
- Text in the Meta tags of HTML documents (which directly specify keywords etc.)
- Text in anchors.
- Text in the title of a web page.
- Text in the URL itself of a link. (Someone searching for, say, a firm like "microsoft" would generally expect to get authoritative information from its corporate official site *microsoft.com*, although other pages may themselves be more relevant)
- Capitalization information is also stored.

  - The Forward Indexer broadcasts the word hits information to the Inverted Indexer.
  - The Forward Indexer incrementally **generates a lexicon**. It broadcasts to other nodes (like the Inverted Indexer) information about the partial lexicon as it is built.
  - It calculates **term frequencies** for words, on a per-document basis.

- **INVERTED INDEXER.** The Inverted Indexer stores word information against a list of documents.
  - It calculates the IDF (Inverse Document Frequency) of keywords
  - An Inverted Index Generator thread is responsible for going through the Forward Indexer's data and generating the inverted index store. This is analogous to the Sorter thread in the Google architecture.
  - It has two dedicated threads which wait on two queues and listen to incoming requests for a list of docID's matching a wordID, document count etc.

- **DISTRIBUTED PAGERANKER.** Given information from crawling, the Page Ranker performs link analysis using the PageRank algorithm. The current projects a **distributed Page Rank** algorithm, distributed across various Pastry nodes.

- **SEARCH ENGINE & USER INTERFACE.** A web front-end interface is provided for the end user for entering search terms, as well as to display the search results. The front end also integrates search results from Yahoo.

PEER-TO-PEER COMMUNICATION

The Pastry peer-to-peer substrate is used to provide a decentralized, self-organizing and fault-tolerant overlay network over which the application is deployed.

This is the main underlying messaging network over which all the other components of the system communicate with each other. One Pastry ring is deployed, which is transparent to all other system components. These components use the services of a pastry node object to send anycast or pseudo-multicast messages to other nodes. The relevant listeners at these nodes know which messages to dequeue. These messages are then added to the appropriate queues, from which the worker thread(s) at the nodes can service them.

- To ensure a more evenly distributed hashing, the hostname of a node is concatenated with itself to generate the minimum bit-length required by the *buildNodeId()* function.
- The system implements a large number of its own queues to avoid dropped messages in the Pastry ring. (Pastry does limited buffering of outgoing messages).

In addition to this, a dedicated framework consisting of

- One listener thread
- One (or more) worker threads
- Request queue
- Response queue

is provided to most modules for efficient bufferred inter-module communication.

3 Implementation

CRAWLER

The crawler has been implemented in a fully distributed manner, over a pastry ring. One (or a limited number of) Crawler thread is run at every node.

If additional crawling performance is required, additional nodes can be deployed, and the system would scale gracefully.

The web crawler consists of the following nodes:

- **LINK EXTRACTOR.** It extracts links from a downloaded web page.
- **ROBOTS.TXT HANDLER.** This specifies which directories and URLs in the web site should or should not be crawled. This module supports the Allow and Disallow directives as specified in the robots.txt specification, and supports the '*' clause.
- **URL FRONTIER.** This is used to maintain crawler politeness, and is used to distribute the crawling tasks across the various nodes.
- **DOWNLOADER.** This downloads the HTML code for a web page. The page is split into hits and meta-information, which is stored.
- **URL QUEUE.** The crawler adds URLs to this queue. The crawler is responsible for dequeing elements from this queue.
- **URL OBJECT QUEUE.** The Forward Indexer fetches a URL object from this queue, downloads the document, and performs extracts various information from the same.

PAGE RANKER

The page ranker is implemented in a **distributed** fashion.

- Whenever a new page is crawled it would have outgoing links. The outgoing links are cleansed for multiple outgoing links and self links if present are also removed from the ArrayList of outgoing links.
- Once this is done, the crawler sends two types of messages outgoing links and incoming links.
- Whenever a pageranker receives a incoming link message *(UpdateIncoming)* it checks in its class level variable *outgoingLinks* wether there are any outgoing links for that. If there are any outgoing links it sends a message to the link pointing to it saying it is healthy *(HealthyLink)*. Whenever a node receives a *HealthyLink* message it sends a messsage of type *UpdateRank* to all its healthy outgoing links.
- Whenever a node receives a *UpdateRank* it recomputes its rank and if its greater than the threshold difference it sends it updated value to only the healthy outgoing links.

- Once a page stabilizes its rank it sends its new rank only to its dangling links.

PageRank is calculated using the formula:

$$PR(A) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{C(T_n)}\right)$$

$$d = 0.85$$

where $PR(A)$ is the PageRank of page A, and pages T_1 through T_n have incoming links to page A. C(X) is the number of outgoing links from a page X.

This is an iterative and incremental process when a new page is crawled the ranks would be recomputed for all the links to which this document point. This helps us to start with our search engine as soon as our crawler starts and moreover it stabilizes incrementally whenever a new document joins.

FORWARD INDEXER

A lexicon is replicated at every node of the distributed indexer. IDF associated with every word. Also, is is possible to dynamically specify at runtime which words need not be indexed.

The indexing system consists of the following components:

- **DOCUMENT PARSER.** The document parser is responsible for text parsing of documents, and word hit analysis.
- **FORWARD INDEXER.** The Forward Indexer populates the DocWordProperties data structure, which represents the forward index from a docID to a list of wordIDs.
- **DOCWORDPROPERTIES.** This is the primary forward index. It stores information about a docId, a list of wordIds for it, and metadata such as hit information for those words.
- **LEXICONMANAGER.** The Forward Index generates a **Tiny Lexicon** at the end of indexing a web document. It is the responsibility of the LexiconManager consolidate this information into one large lexicon.

INVERTED INDEXER

The inverted indexer system is used to generate an inverted index from a wordID to a list of docIDs. Additional data is also stored for efficiency, and optimized query results. It comprises the following components:

- **DOCCOUNTREQUESTHANDLER.** This provides a dedicated service to other modules which need to find the number of documents which are relevant for a given keyword. It is typically used by the metasearch engine for result item relevancy ranking. It consists of one Listener thread, a **DocCountRequest** Queue, and one worker thread to service requests

- **DOCLISTREQUESTHANDLER.** This returns a list of docIDs and other relevant meta-information for a specified wordID. It is called by the Metasearch node.
- **INVERTEDINDEXGENERATOR.** This listens to DocWordProperies pastry messages broadcast by the Forward Indexer, and converts the information into an inverted index format.
- **INVERTEDINDEXSTORE.** This data structure is used to store the inverted index for a node. Each node in the distributed system has its own version of the InvertedIndexStore. It is internally stored as a Berkeley DB database.

METASEARCH ENGINE

This is a distributed system which listens for user search requests and generates the results after querying the InvertedIndexer and ForwardIndexer.

4 Technical Implementation Details

Language:	Java (JDK 1.5, Servlets 2.4)
Database Backend:	Berkeley DB Java Edition 3.2.13
Application/Web Server:	Apache Tomcat 5.5 (For client user interface)
Peer-to-Peer Substrate:	FreePastry 1.4.4
Result Rendering:	XSLT 1.0 generated XHTML

5 Evaluation

PERFORMANCE METRICS		
TASK	NODES	TIME TAKEN (SECONDS)
Cached query	1	0.3
New query (not in cache)	1	2.3

6 Division of Labor

A broad division of labor among the team members is outlined below. All members were involved in overall design, development, coding and testing.

DEBAJIT

- Inverted Index and related frameworks
- Berkeley DB components
- Implementation of Worker thread/queue model
- Pastry skeleton
- Report

KHADER

- Crawler
- Indexer
- Overall integration
- Search interface

SHALIN

- Distributed PageRank
- Pastry code for overall peer-to-peer node communication
- Queueing data structures
- Robots.txt handler

7 Conclusions

Our current implementation of the distributed peer-to-peer search engine was observed to perform upto expectations, and had met all the requisite goals outlined at the outset of its design.

- Each and every component (including PageRank, Crawler, Indexer, and Metasearch) is fully distributed, resulting in a system which is fault-tolerant and scalable with minimal effort.
- Replication of data structures have been used in some modules (like the Forward Indexer and the Page Ranker) for faster query reponse time (at the expense of a marginally higher background indexing time)

We have endeavored to create a system that is robust, scalable and one that can be deployed in a practical scenario, if not (yet) a commercial one.

One of the bottlenecks in our implementation has been the high proliferation of network messages. This is one area which we hope we can optimize even further in any future work based on our codebase.

On the whole, the entire design and development effort has been a great learning experience in building a large scale distributed system, and the results have fully met our expectations.