Steven Bauer & Bret Gourdie

Cloud Computing

TinyGoogle Part I Write Up

# Overview

Our implementation of TinyGoogle is written in Java using the client/server model. The project is split into the following separate applications. We have included a makefile to make compiling the project easy but the makefile doesn’t run the programs. This is because it doesn’t make sense to run them all on the same machine and it doesn’t seem possible to create a makefile that would run them across the elements cluster. For proper execution the nameserver must be run first, then TinyGoogle and the workers need to be spun up, and lastly the client. While the connection code should handle timeouts, it doesn’t make sense to run them in a different order.

# Name Server

The name server continually loops taking requests to resolve addresses and register services. The name server holds this information in a two dimensional array to provide fast access to the relevant information. The name server creates “nameserver.txt” in its current directory which contains its IP address and Port so that the other machines can connect to it.

# TinyGoogle

TinyGoogle is the most complicated piece of the project because it contains many parts and has several jobs. Upon start up TinyGoogle acquires the network information of the name server and registers itself. TinyGoogle then sets up its sockets and awaits a request from the client. When a request is received, TinyGoogle creates a thread to execute the client’s request so that the main program can stay listening to handle additional requests.

The spawned thread parses the user’s request and calls the search or index method. Before it is possible for the work to be assigned, the thread has to acquire the network connection information of all the available worker machines. Ports to send and receive information from each of the workers are created and added to a two dimensional ArrayList structure.

We handle sending jobs to the workers via a Producer and Consumer situation. Using a blocking queue, the consumer attempts to consume a worker and send one term to be searched. Each time it consumes a worker, the consumer spawns a Producer thread. The Producer thread passes either a term to be searched or a filepath to be indexed.

If the client requested a file to be indexed, the thread divides the file to be indexed into 500 word chunks. These chunks will be sent to the worker. Indexing is explained further in the worker section. As indexing finishes, the TinyGoogle thread gathers the filenames sent back by the workers and begins to read in the results of indexing into a hashtable. This hashtable is used to update the master index record, inserting the file’s words and frequency alongside other documents or adding a new entry altogether. TinyGoogle then sends a success or failure notice to the client.

On the other hand, if the client requests a phrase to be searched, the work is divided by the number of terms in the query. Again, the searching is explained in more detail in the worker section. After all of the searching has been completed by the workers, the TinyGoogle thread aggregates the data. It maps the documents to a hashtable where the file name is linked to an array of the ranks of each word found in the master index record. For example, if a document has the most instances of the word “apple,” it would receive a rank of 1; if it has the second most instances, the rank would be 2, and so on. These ranks are added together. Any document not containing all of the searched terms is out of the ranking. Each document that has a rank for each word is then compared against one another: the lowest score wins so we can retrieve the most relevant document. The lowest ten scores are then sent to the client as the top ten results.

# Worker(s)

As with the other pieces of software, each worker’s first job is to get the name server’s connection information and register itself. Once the worker has set up its ports, it waits for TinyGoogle to send it a job. If the job is a search, the worker will access the relevant inverted index file (if the search was “Apple,” it would open a.txt) and then finds the requested word and returns a string containing every document that word occurs in and its frequency. These strings are collected by TinyGoogle, which will calculate their relevancy once all terms have been searched. If the job is an index request, the worker receives a path to the chunk of the file requested to be indexed. This file is read word by word and the worker creates a hashtable where the key is the word and the value of a key is its occurrence. Once the entire chunk has been read and entered into the hashtable, the hashtable is written to a file due to socket limitations. The file path is passed to the TinyGoogle thread, which will collect all file names and add them to the master index record.

# Client

The client is the simplest piece of the project. The client’s first job is to read the file containing the connection information of the name server. Once it has contacted the nameserver, the client requests the connection information of the TinyGoogle server. The client prompts the user to enter their command and relevant arguments and then sends them to the TinyGoogle server waiting for the results. The client does almost no work; all parsing, tokenizing, and formatting is handled server-side by TinyGoogle and its threads.