CIS 455/555: Internet and Web Systems

Spring 2015

Assignment 3: MapReduce

Due April 8, 2015, at 10:00pm EDT

1 Introduction

In this assignment, you will build a simple MapReduce framework that consists of two servlets and your servlet container from Homework 1. You will test your framework by implementing a simple WordCount.

As usual, you should start by checking out the framework / skeleton code (project HW3) from `svn`.

2 Overview

Your MapReduce framework will consist of two types of nodes: a number of *workers* and a single *master*. The workers are in charge of running the `map` and `reduce` functions, and of storing the data that your MapReduce framework is working on; the master coordinates the workers and provides a user interface.

Each worker will have a *storage directory* in which it keeps its local share of the data. For instance, a worker might have a storage directory called `~/store`, with a subdirectory `~/store/webpages` in which it has collected some web pages from crawling, and a subdirectory `~/store/index` in which it creates an inverted index based on the text from these web pages. Keep in mind that there will usually be many workers, and that the data will be distributed among them; for instance, you might have a system with 10 workers and 100,000 web pages total, but each worker might only store 10,000 of them.

The master will have a *status page* on which it displays the list of workers that are currently online, as well as some information about each (e.g., the worker's IP address, and what the worker is doing). To keep this list up to date, the workers will periodically send some information about themselves to the master. The status page will also have an input form that allows the administrator to specify a MapReduce job to run, as well as some parameters (such as a subdirectory of the storage directory to read data from, a subdirectory to write the output to, etc.). When the administrator submits this form, the master forwards this information to each of the workers, which then begin processing the data.

The master and the workers will all be implemented as servlets – in other words, the master node will run your **servlet container from HW1 with a MasterServlet**, and the workers will run the same servlet container with a **WorkerServlet** (possibly on a different machine). The actual MapReduce jobs will simply be classes that implement a special interface (which contains a `map` and a `reduce` function). In a MapReduce framework like Hadoop, these classes would be sent from the master to the workers; to simplify the assignment, we will assume that these classes are already in the classpath on each worker.

3 The master

Your first step will be to implement the MasterServlet.

3.1 Status updates

The MasterServlet should provide a way for the clients to report their status. Specifically, it should accept GET requests for the URL `/workerstatus`, with the following parameters (in the query string):

- `port`: the port number on which the worker is listening for HTTP requests (e.g., `port=4711`)
- `status`: `mapping`, `waiting`, `reducing` or `idle`, depending on what the worker is doing (e.g., `status=idle`)
- `job`: the name of the class that is currently being run (for instance, `job=edu.upenn.cis455.mapreduce.job.MyJob`)
- `keysRead`: the number of keys that have been read so far (if the status is `mapping` or `reducing`), the number of keys that were read by the last `map` (if the status is `waiting`) or zero if the status is `idle`
- `keysWritten`: the number of keys that have been written so far (if the status is `mapping` or `reducing`), the number of keys that were written by the last `map` (if the status is `waiting`) or the number of keys that were written by the last `reduce` (if the status is `idle`). If the node has never run any jobs, return 0.

Note that the worker is *not* reporting its IP address because the master can get it from the request. (The port number is needed because the worker will usually use different ports for listening and for sending requests.) The master should keep this information, along with the time it has last received a `/workerstatus` request with a given `IP:port` combination.

3.2 The status page

When a client requests the URL `/status` from the master servlet, the servlet should return a web page that contains a) a table with status information about the workers, and b) a web form for submitting jobs. The table should contain one row for each active worker (a worker is considered active if it has posted a `/workerstatus` within the last 30 seconds) and columns for 1) IP:port, 2) the status; 3) the job; 4) the keys read, and 5) the keys written.

The web form for submitting jobs should contain fields for:

- The class name of the job (e.g., `edu.upenn.cis.cis455.mapreduce.job.MyJob`)
- The input directory, relative to the storage directory (e.g., if this is set to `bar` and the storage directory is set to `~/foo`, the input should be read from `~/foo/bar`)
- The output directory, relative to the storage directory
- The number of map threads to run on each worker
- The number of reduce threads to run on each worker

3.3 Processing jobs

When the administrator submits the web form on the `/status` page, the master node should itself POST a message to the URL `/runmap` on each active worker. The POST should contain the following parameters (in the body of the HTTP request):

- `job`: the name of the class whose map function is to be run
- `input`: the name of the input directory (see above)
- `numThreads`: the number of map threads to be instantiated on this worker

- `numWorkers`: the number of workers that were active when the job was submitted
- `worker1`, `worker2`, `worker3`...: the IP and port of the first, second, third, ..., worker (Example: `worker7=158.130.53.72:8080`).

The master should remember the parameters of the job (e.g., the class name). From now on, whenever a `/workerstatus` is submitted by a worker, the server should check whether the statuses of all the active clients are `waiting` - i.e., whether the map phase has completed on all the workers. (You may want to perform this check in the handler for /workerstatus.) Once this is the case, the master should POST a message to the URL `/runreduce` on each active worker. This should contain the following parameters:

- `job`: the name of the class whose reduce function is to be run
- `output`: the name of the output directory (see above)
- `numThreads`: the number of reduce threads to be instantiated on this worker

4 The Worker

4.1 Status updates

When initialized, the worker servlet should look for an init parameter called `master`, which should be in the form `IP:port` (Example: `158.138.53.72:3000`). It should then create a thread that issues a GET `/workerstatus` to this IP and port once every 10 seconds; see 3.1 for the required parameters. The thread should be set up in such a way that it does not abort or crash if the connection fails for some reason (e.g., when the master is down, or has not started yet).

The servlet should also look for an init parameter called `storagedir`, which should specify its local storage directory. All the input and output directories should be relative to this directory.

4.2 Handling /runmap

When the worker receives a `/runmap` POST from the master, it should first load the class that was specified in the `job` parameter (using the classloader, just like you did with the servlet). It should then instantiate the requested number of threads (`numThreads` parameter); each thread should read key-value pairs from the files in the specified input directory (`input` parameter) and invoke the `map` function for each pair.

You may assume that the input files contain one line for each key-value pair, and that the value is separated from the key by a tab character. You must ensure that no key-value pair is read twice; for instance, you can do the reading in a synchronized method of the servlet itself, or you can somehow divide up the files in the input directory among the workers.

The worker should also create two local directories called `spool-out` and `spool-in` (delete first if they already exist). When the `map` function emits a key-value pair, the worker should hash the key to one of the workers in the list from the `/runmap` POST, and then append it to a file in the `spool-out` directory; there should be exactly one file for each of the workers, and the format should be exactly like in the input files (one line per key-value pair, and keys separated from values by a tab character). To ensure testability, we *require* that you use SHA-1 as your hash function, and that you split the hash range equally among the workers. For example, if you have four workers, the first worker should handle all the keys that hash to the range `0x0000...0x3FFF`, the second worker should handle the keys that hash to the

range `0x4000...0x7FFF`, the third should handle `0x8000...0xBFFF`, and the fourth should handle `0xC000...0xFFFF`.

Once all the keys have been read on this worker (requires some coordination among the threads!!), the worker should go through the list of files in the `spool-out` directory and issue a POST to the URL `/pushdata` on the corresponding worker. The body of the post should consist of the contents of the corresponding file, exactly as-is (no need to encode as parameters!). Once this is done, the worker should change its status to 'waiting'; it is a good idea to send another `/workerstatus` to the master at this point, even if it isn't 'time' yet.

If the worker receives a POST to `/pushdata`, it should store the data from the body of the request in a file in the `spool-in` directory (name doesn't matter).

4.3 Handling /runreduce

When the worker receives a `/runreduce` POST from the master, it should first load the class as above. It should then sort the key-value pairs in the files in `spool-in` by key. (You may implement this yourself in Java, if you like, but perhaps the easiest way is to use `Runtime.exec` to run the UNIX `sort` command, with appropriate parameters, and to wait for its completion. `sort` has the ability to sort files that are larger than the available memory, which is quite tricky to get right when you implement it yourself.) Once the sorting is completed, the worker should proceed essentially as with `/runmap` (instantiate the requested number of threads, read the key-value pairs, etc.); however, it should now a) run the `reduce` function instead of the `map` function; b) read all the values with the same key at once, so it can give a set to the `reduce` function; and c) write any output key-value pairs to the specified output directory within the storage directory, rather than to `spool-out`. (If the output directory already exists, it should be deleted first.) Once all the data has been read from `spool-in`, the worker should change its status to 'idle' and, again, issue a `/workerstatus` to the master, even if it isn't 'time' yet.

5 WordCount

For testing, you should write a simple MapReduce job that implements WordCount. The name of the class should be `edu.upenn.cis455.mapreduce.job.WordCount`. Please submit the source code for this class along with your two servlets.

6 Testing and debugging

Recall that, for HW1, you assumed that the host name was `localhost`. This won't be a good assumption for this assignment anymore, particularly if you choose to deploy your workers on Amazon EC2, so you should change your servlet container to accept any other hostname.

For testing, you can run a master and a few workers locally in your VM (but on different port numbers, and with different storage directories).

A good way to start is to implement the master's status page first (you can test this by issuing a few `/workerstatus` GETs manually). Next, you could write a simple worker servlet that issues `/workerstatus` to the master, and test whether this works (by starting a few workers and checking that they show up on the master's status page). After this, you could implement the server's web form, and test it with dummy implementations on the workers, etc.

Caution: If you are not completely confident that your servlet container from HW1 works correctly, you may want to initially develop and test your servlets with Jetty. Once they work with Jetty, you can then run them in your own servlet container and fix any additional bugs that may be triggered there.

7 Requirements

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

1. Your master servlet must be called `edu.upenn.cis455.mapreduce.MasterServlet`, and the worker servlet must be called `edu.upenn.cis455.mapreduce.WorkerServlet`.
2. The format of the various GET and POST messages must be exactly as specified in the handout.
3. MapReduce jobs must implement the `Job` interface that came with your framework code in `svn`.
4. IP and port number of the master, and the location of the storage directory, must be read from init parameters as described above, and may *not* be hard-coded.
5. Your submission must contain a) the entire source code for the two servlets, b) the source code for WordCount, c) an `ant` build script called `build.xml`, and d) a `README` file. The `README` file must contain 1) your full name and SEAS login name, 2) a description of features implemented, 3) any extra credit claimed, 4) a list of source files included, and 5) brief instructions on how to run the application on your application server. You must also complete all the yes/no questions.
6. When your submission is unpacked in the original VM image and the `ant` build script is run, your solution must compile correctly. Please test this before submitting!
7. Your servlet must display your full name and SEAS login name on the master's status page. We use this as a sanity check during grading.
8. Your solution must be submitted via `turnin` before 10:00pm EDT on April 8, 2014. The project name for `turnin` should be `hw3`.
9. Your code must contain a reasonable amount of useful documentation.

Reminder: All the code you submit (other than any code we have provided) must have been written by you personally, and you may not collaborate with anyone else on this assignment. Copying code from the web is considered plagiarism.

8 Extra Credit

8.1 Dynamic deployment (+5%)

Extend your master and worker servlets such that the administrator can upload the `.class` file for a new job via the web interface (i.e., it doesn't have to previously exist in the workers' class paths).

8.2 Load balancing (+15%)

Add a function to your master that monitors the progress of the workers during the 'map' and 'reduce' phases and that reassigns some of the work when some workers have completed their phase while others still have a lot left to do. For instance, the master could send a POST to some of the workers that have already finished and tell each of them about one of the workers that is still busy; the idle workers could then issue a POST to the busy workers (maybe to an URL called `/stealwork`) and download some of the key-value pairs that have not yet been processed (be sure to remove the downloaded key-value pairs from the input files of the busy workers, so that they are not processed twice!). There are several important design decisions you'll need to make - e.g., when to reassign work, who to steal work from, who to assign the work to, and how much of it - and you'll probably need to experiment a bit to get to a point where all the workers finish approximately at the same time.