Lab 3 – Distributed Word Count with RMI

CC5212-1

March 23, 2016

This week, instead of sending messages, we are going to use a similar RMI infrastructure to perform a word count in a distributed way. More specifically, we are going to try again to count trigrams (3-grams) using the same file as before. We saw previously that we could not fit the trigrams in memory to count them, but that we could count them using the hard-disk. This time we are going to try to use the collective main memory of the lab to store the count, so we have 30 times the amount of RAM. We will use RMI to communicate between machines. As we will see though, designing such an infrastructure to work reliably in the lab is not so trivial.

How would we design such an infrastructure? Let us say we are only interesting in the top-10 trigrams from the file. One idea would be to have each machine in the lab take a different part of the input file (such that the lab covers the entire file exactly once). Then each machine sends batches of consecutive trigrams to another machine in lab to get them to count the batch (either returning the counts or storing them remotely). But actually there would not be much benefit to doing this because collectively speaking, it would be the same as every machine counting the same batch locally, just that the batches are shuffled amongst machines. Thus a better option is perhaps that each machine counts the trigrams in its local part of the file and later the machines share the top-10 trigrams of their local part with all the other machines, which can be added together to form the final top-10, but this has two problems: (1) in a local part of the file, a trigram that would be in the global top 10 might not be in the local top 10 and actually there's no obvious way[1] to be sure of computing the correct global top-10 without each machine sharing *all* counts with each other, requiring all machines to store all counts which we know won't fit in memory, (2) the counts of many trigrams will end up on multiple machines, meaning that it may be inefficient in terms of memory.

Instead, the option we will go for is to send specific trigrams to specific machines in a deterministic manner (meaning that everyone in the lab will send the same trigram from different parts of the file to the same machine), where that machine keeps track of the counts for its trigrams (rather than returning counts to the client machine). That way, each machine will have the final global count for each trigram it is in charge of, so we need only share the top-10 from each machine to find the global top-10. Also, with n machines and a good function for mapping trigrams to machines, each machine will end up roughly $\frac{1}{n}$ of all unique trigrams (some machines may end up with more common trigrams, but unique trigrams should be about the same on each machine). So this sounds like a good infrastructure to start with!

That's a lot of text. But let's go step by step.

- Download `http://aidanhogan.com/teaching/cc5212-1-2016/lab/03/mdp-lab03.zip`.
- I will set up a central directory somewhere in the class that you can connect to. I will give the details of the hostname and port in the class. Like last week, we will use this central directory so machines can share their IP, port, etc. Everything will be the same as last time.

  If you are doing this assignment outside the lab, it will be a little boring in that you will have to send word count requests to yourself, but you could try to set up multiple word-count servers later on. As a first step, you will need to set up the directory yourself, for example on your local machine. In the code, there is `rmi.jar` in the `dist/` folder.

[1] There are some tricks we could use to avoid this, like trying to compute a lower bound on the count that a top-10 trigram should have, etc., but it's not so trivial in a distributed setting.

- `java -jar rmi.jar StartRegistryAndServer -r -s 1`

  This starts a central directory. You need to keep this running, e.g., in a separate window (hence it is easier to run as a jar then in Eclipse). You can connect to this directory through localhost. The default port is 1099.

- Next we are going to have a look at the server for counting words. This is the code that implements the count on each machine. It's already done for you.

  - Have a look at the code for `org.mdp.wc.naive.server.NaiveStringCountStub` and `org.mdp.wc.naive.server.NaiveStringCountServer`. There's nothing so fancy in there. Note that the server stores the counts of the strings locally, it does not return them. Later a client can request the counts or the top-k.

- Next head to `org.mdp.cli.DistributedNgramCountApp`. A bunch of input/output stuff is done for you in the `main` method. The interesting part is the `distributedNgramCount` method. A lot of steps are already done for you (and are similar to last week's lab) but some parts you have to code, marking below with `TODO`.

  - First it prints a logo. This is for marketing purposes.
  - It opens the file of abstracts you will be counting n-grams from. You will have a unique part of the overall Wikipedia file.
  - It takes user details from you on the command line, like last week.
  - It starts a registry and a remotely accessible `NaiveStringCountServer` instance for you. This will let other people ask you to count n-grams for them.
  - It connects to the directory and announces your details where others can find them.
  - Next it sets a coordination point. Before continuing we need to ensure all machines have started their server and have connected to the directory. Once this is the case, the user can type `'next'` to continue.
  - Now we open a connection to each of the operational machines in the lab and test the connection. (If something doesn't work while running this, we may have to start again. Otherwise if things are not working, we can try catch the exception and remove the slave.)
  - The next part is what you need to implement (look for `TODO` in `DistributedNgramCountApp`):
    - First you need to decide where to send each n-gram. Use the *positive* hashcode of the n-gram, modulo the number of machines.
    - Add the n-gram to the batch for that machine. If the batch is full, send it to the server, clear that batch, and print a message.
    - After the iteration ends, you might still have data cached into the batches. You need to check every batch and if it's non-empty, send it to the corresponding machine.
    - Finally you want to call each machine to tell them you have finished sending all data and are ready to continue to the next part, using the `finalise` method sending your user details.
  - After this there is some code done for you. You will check from your `NaiveStringCountServer` if all machines have finalised or not. Everyone needs to wait until all machines have sent all data. If we run a count before that, we may miss data. This is handled automatically (though you need to hit enter to refresh).
  - Now that all machines have sent all data, we can ask each machine for its top-k. We can add this to a local count object, and then take the final top-k from that.
  - Last we just keep the program alive in case other machines have yet to get the counts you have locally.

- Now we try run the task. There are many things that can go wrong here, in that if one machine fails, the whole task fails, but hopefully we'll have enough time to fix what needs to be fixed and get a working system. ☺
  - First of all, you need to get your own part of the file to download. I will split the file according to how many people we have and let you know your split to download. (If you are doing this from home, try test with `http://aidanhogan.com/teaching/data/wiki/es/es-wiki-abstracts-1k.txt`.)
  - Second, you need to run `DistributedNgramCountApp`. You need to pass as command line arguments the location of the file on disk `-i`, (also pass `-igz` iff it is GZipped), the top-k sought (`-k 10`), the length of n-gram (`-n 3`), as well as the hostname (`-dh` DIRECTORY HOSTNAME) and port (`-dp` DIRECTORY PORT) of the directory. **Make sure to set the heap-space as high as you can with `-Xmx`amount as a virtual argument.**
  - Follow the command-line prompts. At the end, take a note of the top-10 trigrams. Also take note of how many strings your machine counted. We can check in the class how even load was.
  - (If you are doing this from home, you can try run the application multiple times on localhost but with different ports to act as different slaves. This might be useful if you want to do that in Eclipse: `http://stackoverflow.com/questions/5453894/run-two-java-programs-from-eclipse-at-once`.)
- QUESTIONS: How was the load on different machines? What sort of system did we implement today in terms of CAP? How could we make the system more tolerant to faults? What would be the trade-offs?
- OPTIONAL: In the `org.mdp.wc.ft.server` package, I started to implement some ideas for a more fault-tolerant system with replication. I decided it was a bit too complex so I abandoned it without really testings the code but left it in the code project in case you're curious.
- Submit the `DistributedNgramCountApp` classes to ucursos, deadline on Monday.