Lab Handout 7: `proxy` Redux and MapReduce

Students are encouraged to share their ideas in the #lab7 Slack channel. SCPD students are welcome to reach out to me or Hemanth directly if they have questions that can't be properly addressed without being physically present for a discussion section. The lab checkoff sheet for all students–both on-campus and off–can be found right here.

Before starting, go ahead and clone the `lab7` folder, which you'll descend into to create custom mapper and reducer executables that, when used by my own solution to Assignment 8, will process several MBs of data to produce a custom result that might be of interest to the wine industry.

```
poohbear@myth15:~$ hg clone /usr/class/cs110/repos/lab7/shared lab7
poohbear@myth15:~$ cd lab7
poohbear@myth15:~$ make
```

Problem 1: `proxy` Thought Questions

- Some students suggested that your Assignment 7 `proxy` implementation open one persistent connection to the secondary proxy when a secondary proxy is used, and that all requests be forwarded over that one connection. Explain why this would interfere with the second proxy's ability to handle the primary proxy's forwarded requests.
- **Long polling** is a technique where new data may be pushed from server to client by relying on a long-standing HTTP request with a protracted, never-quite-finished HTTP response. Long polling can be used to provide live updates as they happen (e.g. push notifications to your smartphone) or stream large files (e.g. videos via Netflix, youtube, vimeo). Unfortunately, the long polling concept doesn't play well with your `proxy` implementation. Why is that?
- HTTP pipelining is a technique where a client issues multiple HTTP requests over a single persistent connection instead of just one. The server can process the requests in parallel and issue its responses in the same order the requests were received. Relying on your understanding of HTTP, briefly explain why `GET` and `HEAD` requests can be safely pipelined but `POST` requests can't be.
- When implementing `proxy`, we could have relied on multiprocessing instead of multithreading to support concurrent transactions. Very briefly describe one advantage of the multiprocessing approach over the multithreading approach, and briefly describe one disadvantage.
- We interact with socket descriptors more or less the same way we interact with traditional file descriptors. Identify one thing you can't do with socket descriptors that you can do with traditional file descriptors, and briefly explain why not.

- When I updated `createClientSocket` for the `proxy` assignment, I replaced the call to `gethostbyname` with a call to `gethostbyname_r`, which has the prototype listed below. This second, reentrant version is thread-safe, because the client shares the location of a *locally* allocated `struct hostent` via argument 2 where the return value can be placed, thereby circumventing the caller's dependence on shared, statically allocated, global data. Note, however, that the client is expected to pass in a large character buffer (as with a locally declared `char buffer[1 << 16]`) and its size via arguments 3 and 4 (e.g. `buffer` and `sizeof(buffer)`). What purpose does this buffer serve?

```
struct hostent {
  char *h_name;       // real canonical host name
  char **h_aliases;   // NULL-terminated list of host name aliases
  int h_addrtype;     // result's address type, typically AF_INET
  int length;         // length of the addresses in bytes (typically 4, for
IPv4)
  char **h_addr_list  // NULL-terminated list of host's IP addresses
};


int gethostbyname_r(const char *name, struct hostent *ret,
                    char *buf, size_t buflen,
                    struct hostent **result, int *h_errnop);
```

- In lecture, we presented three different siblings of the `sockaddr` record family, as shown below. The first one is a generic socket address structure, the second is specific to traditional IPv4 addresses (e.g. `171.64.64.131`), and the third is specific to IPv6 addresses (e.g. `4371:f0dd:1023:5::259`), which aren't in widespread use just yet. The addresses of socket address structures like those above are cast to (`struct sockaddr *`) when passed to all of the various socket-oriented system calls (e.g. `accept`, `connect`, and so forth). How can these system calls tell what the true socket address record type really is–after all, it needs to know how to populate it with data–if everything is expressed as a generic `struct sockaddr *`?

```
struct sockaddr {         struct sockaddr_in {            struct sockaddr_in6 {

  short sa_family;          short sin_family;               short sin6_family;
  char sa_data[14];         short sin_port;                 short sin6_port;
};                          struct in_addr sin_addr;        // other fields;
                            char sin_zero[8];             };
```

```
                                };
```

- Briefly explain why a nonblocking HTTP server can manage many more open requests than a blocking one.

Problem 2: Using MapReduce

For this problem, you're going to flesh out the implementation of mapper and reducer executables (analogues to the `word-count-mapper.cc` and `word-count-reducer.cc` files that ship with your `assign8` repos) that process CSV file chunks housing, um, wine ratings. You'll rely on these mapper and reducer executables to generate output files that map country names to the average rating of all of the wines produced within the country. (By the way, I downloaded the data files for this problem from www.kaggle.com, which is chockful of all kinds of awesome data sets that you can download yourself and play with.)

Here's the series of steps I'd like you to work through for this problem:

- Spend the time needed to read through the Assignment 8 handout. In particular, I'd like you to clone your `assign8` repo, read enough of the assignment handout to know what `make`, `make directories`, `make filefree`, `make clean`, and `make spartan` all do, and follow through all of the steps necessary to run partial and complete MapReduce jobs that tokenize the text of ***The Odyssey***. You'll want to learn how to invoke `mr_soln` (with `mrm_soln` and `mrr_soln` as worker executables), letting a supplied configuration file inform mr where the data files live, how many mappers should be spawned, how many reducers should be spawned, and so forth. Read and do everything up through Task 1 of the Assignment 8 handout.
- Once you have a sense of how to run a MapReduce job, you'll want to focus on a different MapReduce job that's managed by within lab7 repo. In particular, you're going to flesh out the implementation of `wine-ratings-mapper.cc` and `wine-ratings-reducer.cc` to process a collection of CSV files to aggregate wine review averages on a per-country basis. For example, one of the input chunks looks like this:

```
poohbear@myth12:~/lab7$ head -10 /usr/class/cs110/samples/lab7/
wine-ratings-partial/00003.input

1500,Italy,92,43,Sicily & Sardinia,Sicilia,,,,Planeta 2011 Cometa Fiano
(Sicilia),Fiano,Planeta

1501,US,92,32,California,Santa Lucia Highlands,Central Coast,,,Sequana 2010
Pinot Noir (Santa Lucia Highlands),Pinot Noir,Sequana

1502,US,92,71,California,Russian River Valley,Sonoma,,,Small Vines 2010 Baranoff
Vineyard Pinot Noir (Russian River Valley),Pinot Noir,Small Vines

1503,Italy,92,65,Sicily & Sardinia,Sicilia,,,,Tasca d'Almerita 2009 Cabernet
Sauvignon (Sicilia),Cabernet Sauvignon,Tasca d'Almerita
```

```
1504,US,92,60,Oregon,Eola-Amity Hills,Willamette Valley,Paul
Gregutt,@paulgwineξ,Bergstr?_m 2010 Temperance Hill Pinot Noir (Eola-Amity
Hills),Pinot Noir,Bergstr?_m

1505,Austria,92,,Kamptal,,,Roger Voss,@vossroger,Br?_ndlmayer 2011 Steinmassel
Erste Lage Riesling (Kamptal),Riesling,Br?_ndlmayer

1506,France,92,15,Southwest France,Madiran,,Roger Voss,@vossroger,Ch??teau
d'Aydie 2010 Madiran Laplace Tannat (Madiran),Tannat,Ch??teau d'Aydie

1507,France,92,35,Southwest France,Cahors,,Roger Voss,@vossroger,Ch??teau de
Gaudou 2009 R??serve Caillou Malbec (Cahors),Malbec,Ch??teau de Gaudou

1508,France,92,,Beaujolais,Moulin-??-Vent,,Roger Voss,@vossroger,Ch??teau des
Jacques 2011 Clos de Rochegr??s  (Moulin-??-Vent),Gamay,Ch??teau des Jacques

1509,Chile,92,92,Colchagua Valley,,,Michael Schachner,@wineschach,Emiliana 2009
G??  Red (Colchagua Valley),Red Blend,Emiliana

poohbear@myth12:~/lab7$
```

- Each line of each input chunk is a single CSV record for some wine review. You'll want to update the implementation of `wine-ratings-mapper.cc` to process an entire chunk like that above, marshaling each line (like `1500,Italy,92,43,Sicily & Sardinia,Sicilia <etc>`) to a single key value pair (like `Italy 92`). in particular, each line in an input chunk is guaranteed to have a country name and a wine review score (between 0 and 100, inclusive) as the second and third CSV values. (All other values can be discarded.) . Some of the country names themselves have spaces in them (e.g. New Zealand and South Africa), and rather than let those countries confuse the structure of the intermediate output files, country names like South Africa should be written as South-Africa instead–that is, all spaces in between the tokens of multitoken country names should be replaced with hyphens.
- You can test the above by invoking `./slink/mr_soln` with the `--map-only` flag to confirm that it properly generates what appear to be correctly structured intermediate files.
- Once you're confident that looks good, you can tackle `wine-ratings-reducer.cc`, which processes intermediate files that have already been sorted and grouped by key (e.g. each line of the file might look like France `92 92 92 91 89 94 88` or `South-Africa 93 92 86 90`). The final output files should be such that each line is a key value pair associating a country name with the average of all of the wine reviews for those wines bottles in that country–something like `US 93.2` or `New-Zealand 91.1`).