**Compsci 711 Assignment 1**

Megan Ganley 5381586

The programs have been tested on lab machines.

*Instructions*

Included in this assignment are three solutions – one representing each, client, cache and server. Compile and run the solutions from the desktop, ensuring that each solution is set to the right start-up program (they should be Client.GUI, Cache.GUI and Server.ConsoleApp). The programs should create a folder on your desktop called ”711 Files – Megan Ganley”, and nested inside will be a folder to hold files for each of the three programs respectively. Theoretically, we could store these 3 folders, and run these 3 programs anywhere, but for the purposes of this assignment, we conveniently keep everything in one place.

The ServerFiles folder should be populated with your desired files. The client will be able to make a request through the cache to see these files available on the server. The client can then request a download of any of these files by selecting the appropriate line in the listbox, then pressing the download button. The server will send the entire file back through the cache, which will store a cached version of the file. The cache will send the file back to the client. So, eventually this file will be stored in each of the 3 folders.

Once a file has been downloaded, the client will indicate this with a “downloaded” label. This may not necessarily be up-to-date, because files could be deleted from the ClientFiles folder manually, but it is updated to our best knowledge. We can then view the contents of any downloaded file using the default Windows program for the particular file type, when the display button is pressed.

On the cache, we maintain a log of requests and responses. This is stored in the ”711 Files – Megan Ganley” folder, and can be viewed by the pressing the view log button, which will open the log in the default text editor. We can clear the cache of all cached files by pressing the clear cache button.

PLEASE NOTE: The cache UI also displays a list of cached files, once the display button has been pressed. However, this will not always display the most current information. Please press the refresh button when an up-to-date list of cached files is desired.

*Part 1*

We implement basic caching functionality in Part 1, such that file download requests to the server will be fulfilled by the cache if a cached file exists, and is up-to-date (as determined by a comparison of SHA-256 hashes of the full file on the cache and server).

*Part 2*

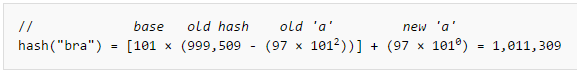
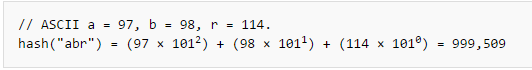
An obvious problem with our implementation in Part 1 arises with the fact that any size update of a file on the server will prompt the cache to redownload the file. This means that even if a file has changes amounting to just 0.1%, 100% of the data will be sent from the server to the cache. By breaking the file into chunks, we can hopefully eliminate some of this redundancy.

In Part 2, the Rabin-Karp function is implemented to slice files into chunks. A hash value is generated for every chunk on the server and the cache, and where these hashes match, we assume the content is identical, and does not need to be resent. Here, we use an SHA-256 hash, which should yield unique and fast results. While we could break files into chunks of a fixed size, this may not maximize the reusable data on the cache. For example, if a few bytes were inserted near the beginning of the file, the fixed size of the chunks would mean that the starting and ending bytes for all chunks would be shifted accordingly, and as such, every chunk would register as modified.

In using the Rabin-Karp function, we can deterministically break a file into chunks that will, on average, gravitate towards some fixed size. We look at a window of some number of bytes (here, 48), calculate a hash value for the window, then make some arbitrary decision as to whether the ending byte is an appropriate ending for a chunk. To do this we specify a bit mask, where we can examine some number of bits at the end of the hash. If just one of those bits is not a one, we decide this is an appropriate boundary.

A bit mask of 13 bits should provide chunks of size around 8192 bytes. If we use a smaller number of bits, the average chunk size will be larger, and conversely, if we use a larger number of bits, the average chunk size will be smaller. There is a fine balance between having a small number of large chunks, in which we may have to resend more data than necessary, and having a large number of small chunks, in which we have to calculate and compare a large number of hashes, which could be a costly computation. 13 bits seems to suit well in this case.

The Rabin-Karp function implemented also makes use of a rolling hash, which minimizes the computations necessary on the cache and server by maintaining a hash value over every iteration, and only adding and subtracting the relevant elements in the formula.



*Example of a similar rolling hash, Wikipedia*

When the client requests a file download from the cache, first we check if a cached version of the file exists. If not, there is nothing to be done, and the file is simply requested from the server.

If the file exists on the cache, we can then check if the cached file is missing updates from the server. We can send a hash of the full cached file to the server for comparison. Then if the file exists, and is up-to-date, we can simply send the cached file back to the client.

If the server detects that the full hash does not match on the cache and server, we slice the cache and server files, and compare hashes of every chunk. The server will determine which content the cache is missing, as well as sending through an index of how the final file should be assembled. This means that chunks can be reused on the cache, even if their arrangement has been changed.

Finally, the cache receives the information, assembles the file, then caches it and sends it in full to the client.

Theory of bandwidth, response time, work done by cache/server

Set of files to test theory on: benchmarks



Small amount of change: insertions/deletions/modifications

50%

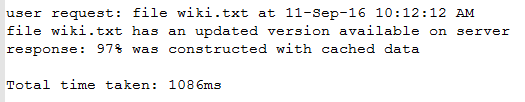
100% changed

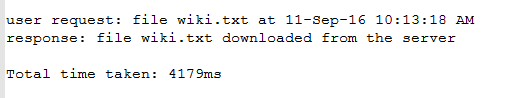
Large txt file

Small txt file

Image file

With a small modification to a large text file, we find that the response takes around four times as long using the programs from Part 1 than with using the programs from Part 2.





How long to generate hash

How long to compare hash

How long to slice file

Bandwidth as measured by size bytes over time