CSCI 5105 Project 3 Writeup

Thomas Reinitz (reini050) and Lucas Olsen (olse0280)

System Design:

The servers of our xFS system work through two different versions of RPC servers: one for the peers, and one for the coordinator. The peer server is intended to be run by peers in the network and only has direct communication to the coordinator server for a majority of its operations. Once online, it will wait for communication from the coordinator before proceeding with any operations. The coordinator will call update_list on the peer to update the peer's locally cached file list (and also the coordinator's).

If a new update is detected via update_list, the coordinator will make another call to get_update on this peer to cache new changes. In a child process of the coordinator of the coordinator, a pinger() function repeatedly calls get_update during the coordinator's lifespan. If a peer disconnects, the server notices and untracks all files from that peer, removing files that were only tracked by that peer completely. If the server coordinator crashes, when it comes back online the main loop (pinger()) of the coordinator will re-track all of the files, repopulating all the tracked files for each client with the check_update function in pinger().

When a peer calls download through ./app, the peer contacts the coordinator server using get_host. This prompts the server to use the peer selection algorithm to find the best peer to download from, returning the best peer's host information. The app then downloads data directly from the peer information returned from the coordinator.

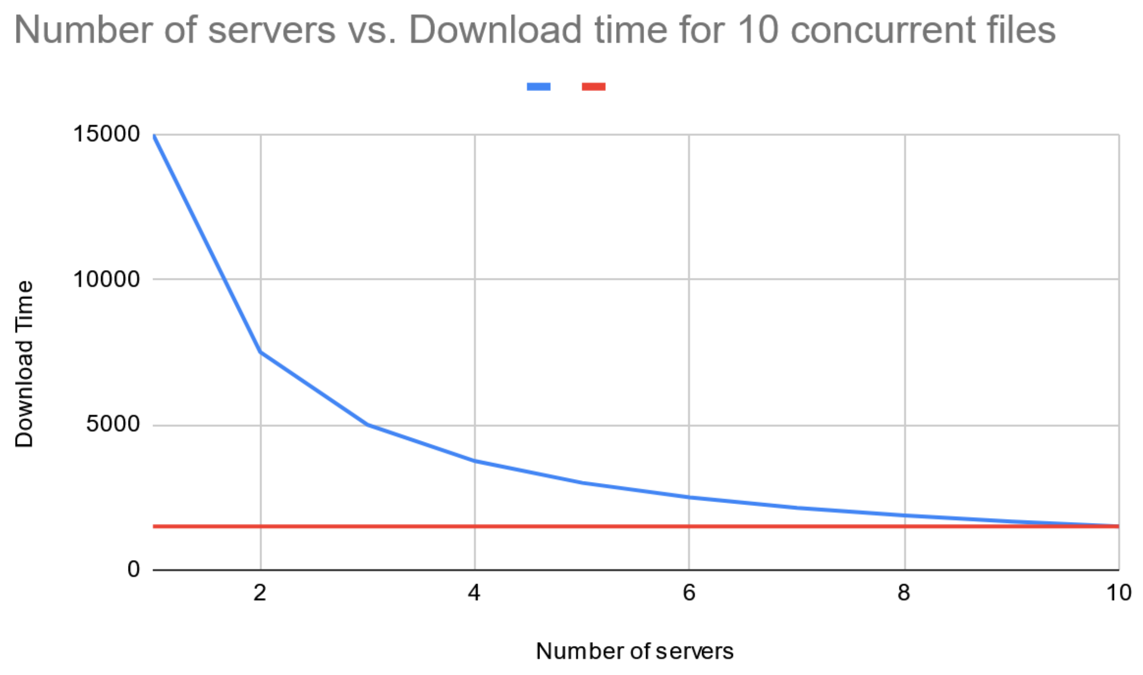
Our peer selection algorithm is based on the load index and latency of the nodes. It takes the minimum of (x +1) * y, where x is the load index of the peer server (the number of concurrent downloads currently happening) and y is the latency in milliseconds of the peer server. This is precalculated in the getLoad function, and sent to the coordinator as the load score. In this way, low latency servers are prioritized over high latency servers, and high load servers are deprioritized. As the number of hosting servers for a file increases, the average download time ends up closer and closer to the average latency of all the hosting servers. If there are more servers than file downloads happening, the download latency can be even better than the average server latency, as the scheme will pick the servers with best latency first. This is shown on the graph on the next page.

After receiving the return from get_host, the client starts the download process from that peer, repeatedly calling download until the entire file has been downloaded.

On each download call, a XOR checksum is calculated and added as the last char in the data string by the peer server. Upon receiving the download return, the client runs the checksum function over all of the returned content from that download section, and if the checksum function returns a non-zero answer, then the checksum has failed and the file contents are deleted, and the client is notified that the download failed, in which case the client can query the coordinator server again for the file. A client can find a file by querying the coordinator through the ./app executable with the list command or the list_all, in which case the coordinator either lists all hosts with the file, or lists all hosts with all their files respectively.

Graph for peer selection performance:

For a given average latency of 1500, the graph shown below is the average download time for the file for x amount of hosting servers, given some arbitrary amount of file downloads (10 concurrent downloads in this case). The red line represents the minimum possible average time to download the file, and the blue line represents the rough average time to download 10 files concurrently.


Number of servers vs. Download time for 10 concurrent files
Download Time
Number of servers