Lab 4 Design Problem - Downloading From Multiple Peers

Interface

The interface of our design problem is to allow the client in the lab 4 implementation be able to download from multiple peers at once, instead of one peer at a time. By being able to do this, it can speed up the time it takes to download a file if smaller parts are downloaded from multiple peers instead of the entire file downloaded from a single peer. In order to implement/support this feature, many changes will be needed to be made to the source code of "osppeer.c".

Implementation

This list is a brief overview of what the client currently does and the steps it takes when it wants to download a file:

1. Get request from standard input of file to download
2. Check the tracker for peers that have the filename that was specified in step 1
3. Create a download task that has a list of peers we can download from
4. Go through the list of peers and connect to one by opening a socket and request for file
5. Open disk file for result
6. Read file into task buffer from the peer
7. Write from task buffer into disk

A couple of steps need to be changed/added in order to allow downloading from multiple peers instead of one peer at a time. This is a general overview of how the new implementation would be to support multiple peers.

1. Get request from standard input of file to download
2. Check the tracker for peers that have the filename that was specified in step 1
3. Create a download task that has a list of peers we can download from
4. Find size of file and divide the task based on number of peers
5. Go through the list of peers and connect to all of them by opening sockets to each peer and request for certain segment of file
6. Open disk file for result
7. Read file into task buffer from the peers (each peer responsible for a certain section)
8. Write from the task buffer into disk

The steps in red are either new or have been modified from the original one peer download model. In order to implement the steps in red, modifications to the code will need to be implemented.

4. Find size of file and divide the task based on number of peers:

To find the size of the file, this code will needed to be added:

struct stat st;  
stat(filename, &st);  
size\_t size;

size = st.st\_size;

Then we need to find the number of peers in the peer list. We can do this by adding code:

int num\_peers = 0;  
peer\_t\* iter = t->peer\_list;

for(; iter != NULL; iter = iter -> next)

num\_peers++;

Once we have the size of the file and the number of peers, we need to divide the work for each peer. We need to do this by assigning a starting and ending point to read from. This will probably be stored in a simple struct such as:

typedef struct readindex {

int start;

int end;

} readindex\_t;

We will assign a different start and end to each peer connection. For example, if the file size is 512, and there are 8 peers, then each peer will contribute 512/8 = 64 bytes. So for example, peer 1 start will be 0 and end will be 63. Peer 2 start will be 64 and end will be 127 and so on for the remaining peers.

We are also going to have to create more threads using pthreads so that we can do the multiple peer download in parallel, instead of sequentially. The procedure will be similar to doing multiple downloads for different files in parallel. So we will create a thread for each peer, and we will need to create an additional struct for info to pass to the thread. The new struct will need the filename as well as the readindex struct created above so the peer knows what file to read and from what indexes.

5. Go through the list of peers and connect to all of them by opening sockets to each peer and request for certain segment of file:

Originally the code only opened one socket:

t->peer\_fd = open\_socket(t->peer\_list->addr, t->peer\_list->port);

For opening multiple sockets, we will need to iterate through the peer list and open the sockets, so t->peer\_fd will need to be a structure that can grow in size, such as a linked list to store each descriptor for the socket. Here is a rough implementation of what the code will look like:

peer\_t\* iter = t->peer\_list;

for(; iter != NULL; iter = iter -> next)

{

int peer\_fd = open\_socket(t->peer\_list->addr, t->peer\_list->port);

peer\_fd\_struct.add(peer\_fd); // this is just an example

}

This will open all of the sockets for each peer so we can download from multiple peers instead of just a single peer when only one socket is open.

7. Read file into task buffer from the peers (each peer responsible for a certain section):

This task will be done in a separate thread for each peer, so that it can be done concurrently. It will require the struct readindex described above, so that each peer knows what to contribute to the desired file to download.

Another part posed in the design problem was performance issues based on download strategies that result in faster download times. Some peers should be favored over others if they are overall faster. An algorithm that could be implemented to do this is have the tracker keep track of the peers in a ranking based on how fast they finish tasks. For example, for the first download, have peers all send equal byte sizes. And based on when they come in, have the tracker rank them accordingly. So the next time when a request is made, a fast peer with a higher ranking will send more data bits than a slow, low ranking peer. This will help us to reduce bottleneck peers that are always slower than the others by having them transmit less data to the client.

Summary of Results / Design Tradeoffs and Possible Implementation Issues

We only began the beginning of designing the implementation but did not completely implement a full working solution. In designing, there were many parts that were hard to account for. For example, the data integrity seems like it would be challenge because as stated in the specifications, the checksum is for the entire file and not a part. However, even if we used the original checksum after the file was transferred by all peers, the overall runtime would still be less than before. So even though this provides an additional overhead that could be averted if there was way to check integrity within each peer, the overall runtime is still decreased.

We learned from this design lab that running things in parallel is usually better, even if that means taking more time to implement things. The normal specification of lab 4 already showed the importance of parallelization by making us do multiple downloads in parallel. But the design problem further emphasized the point by from the multiple downloads in parallel, have each download receive from multiple peers in order to speed up requests even more.

There are many different possible designs that could have been implanted. The break-up of the file could be distributed differently than an equal distribution. In addition, there could be an alternative to using pthreads to run the reads in parallel. Another challenge in the implementation would be if the file size was not a direct multiple of the number of peers. A way around this is that one peer can have less data, or there can be a slight overlap in reads (eg. peer 1 bytes 0-65 and peer 2 bytes 65-129). We also would probably need a lock in that case so we don't corrupt the file on the disk when we write to it.

Who did what

Corey was in charge of the design document and Jonathan was in charge of the presentation slides. We both worked on the implementation of the design.

I jus