Dennis Shen
004002509

Computer Science 111: Design Problem for Lab 4

Introduction
This design document aims to describe the process of which to create a true Peer-to-Peer network by which a client downloads the same file from multiple peers in parallel. Currently the client downloads from one peer at a time. It is possible to speed up the process by downloading it in parts from multiple peers at the same time. This is the fundamental part of a peer-to-peer network such as Bit-torrent. This design document aims to present to the reader a complete description of this user program and to introduce a sensible implementation strategy along with a summary of results which describes the successes and failures of the design implementation.

Goals:
The ultimate goal of this added feature is to allow the user to download from multiple peers.
```
./osppeer -dtest -p cat1.jpg
```
Ideally this modification will allow the user to add a new flag to allow to download the file from others via a new peer-to-peer network. This new -p flag will indicate to the peer that it wants to download the file in pieces from the other peers on the network.

Requirements and don't care behavior:
- Break a file into pieces that can be sent over a network
- Add file integrity tools to these new pieces (checksums on each piece)

Since the focus of this design document is on the creation of downloading from multiple peers then the issues of uploading to multiple peers will not be covered in full, but certain assumptions will have to be made.
Dont Care Behavior:
Some behavior that we would be to create a means of uploading something parallel with other peers. Also in modern BitTorrent networks, peers that are downloading files can also be downloading from each other even if their respective files have not been completely downloaded yet. This behavior is also don't care behavior.

Implementation:
To create the blueprint to create download from multiple peers, I first have to go over uploading. While I mentioned that it was not in the scope of this design document to go over uploading to multiple peers through breaking it apart, it is important to mention how peer-to-peer networks function. The user who wants to upload the first file acts as the original seed file who gives away this file. This file is broken into pieces where each piece can be 1kb to even 1MB of size. Each piece is protected with a hash or checksum that ensures modification is detected. In BitTorrent protocol – the original seed creates a *torrent*, a descriptor file that allow for peers to download and validate their file.

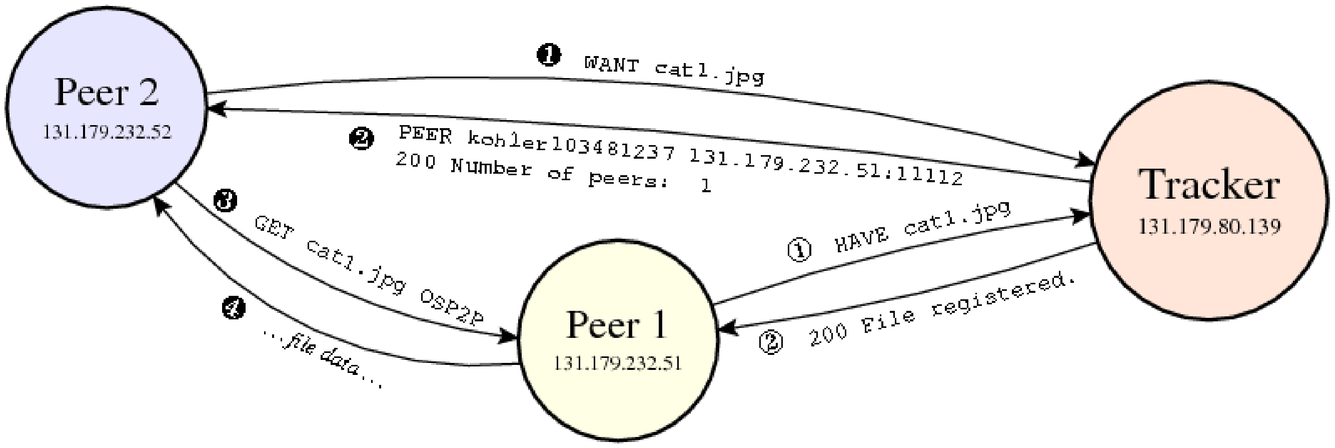

Figure 1: Normal OSP2P File operations

This is the sort of behavior we want – a fully functional peer-to-peer network where a single file can be share and downloaded from multiple peers at once.

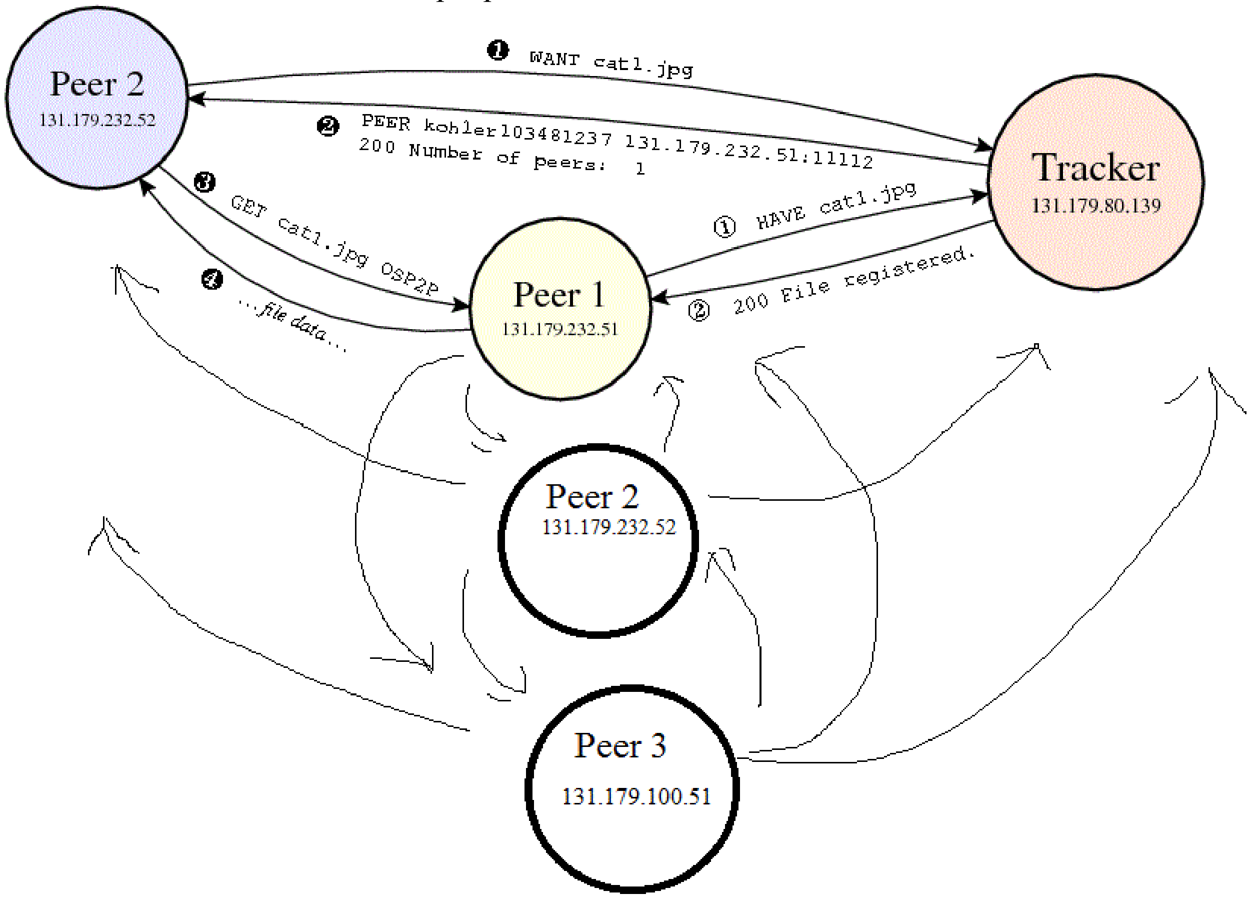

Figure 2: Ideal OSP2P File operations

One of the questions raised in this design problem is file integrity. File integrity has to be validated and enabled at the upload level. The original seed will have to ensure that there is a checksum for each piece that he is uploading along with the original file. The peer's responsibility is to validate the checksum to the original pieces along with reassembling the file. There is a problem with this implementation still in that simply distributing the checksums blindly into the network is not enough. There needs to be a central piece similar to the BitTorrent protocol where there is a descriptor file which holds the checksums. This ensures that a malicious peer cannot replace a checksum and a piece of file altogether.

This solves our file integrity problem and ensures that we can also detect who has sent the bad piece because we treat each piece like its own separate file on the network. The only modification to the peer would be to download the central checksums and ensure that the file pieces match the checksum. This would be equivalent of downloading the metainfo file or a .torrent file.

The implementation would have to thereby download the pieces and ensure their checksums and reassemble them into the original file.

Code Implementation:

While in BitTorrent clients they often use auto-detection to determine the block size for each individual piece that is generated as they often can handle anything from a couple of MB to dozens of GB in size.

Some of the code will be in pseudocode as some functional implementations will be left out due to complexity and it not being within the scope of this design document.

Add a field called parallel mode and

```
	else if (argc >= 3 && strcmp(argv[1], "-p") == 0
		&& osp2p_sscanf(argv[2], "%d", &parallel_mode) >= 0) {
		argc -= 2, argv += 2;
		goto argprocess;
	}
	else if (argc >= 2 && argv[1][0] == '-' && argv[1][1] == 'p'
		&& osp2p_sscanf(argv[1], "-p%d", &parallel_mode) >= 0)
	{
		--argc, ++argv;
		goto argprocess;
	}
	else if (argc >= 2 && strcmp(argv[1], "-p")==0){
		parallel_mode=1;
		--argc, ++argv;
	}
```

This code will handle the new parameter for the parallel mode

Next I created this function to handle the parallel download.

```
void parallel_download(task_t *t, task_t *tracker_task)
{
	int i;
	if(t == NULL)
	{
			error("Unable to get a list of peers to download 
from\n");
			return;
	}

			task_download(t,tracker_task);
			//this will grab the torrent file or whatever metadata 
file.
			//this metadata can be something simple - a number of
```

```
pieces
               //where each file can be seperated into something like
file.part1...file.part10 etc.

               int num_pieces = SOME_PARSING_FUNCTION(t->filename);
               int child_count=0;
               //a set of functions will parse out the number of
pieces to download
               //as well as the checksums associated for each piece.
               in_addr* usage = (in_addr*) malloc (in_addr *
num_pieces);

               //create an array that represents usage where each
element of the array represents the IP
               //responsible for each piece and the piece number.
```

I am creating an assumption that we are downloading a metainfo file. Instead of calling

```
./osppeer -dtest -p cat1.jpg
```

I am calling something like this

```
./osppeer -dtest -p cat1.torrent
```

The task download will download the metadata into a file and use some parsing functions to examine the number of pieces the peer has to download and the checksums associated for each file. I am assuming this file is uncorrupted in the same way an individual wouldn't just download a file from a shady user.

```
for(i=0;i<num_pieces;i++)
{
     if(usage[i]==0)
     {
          usage[i]=t->peer_list->addr;
          pid_t child= fork();
          if(t->peer_list->addr.s_addr == listen_addr.s_addr && t-
>peer_list->port == listen_port)
               continue;
          if( child < 0)
          {
               error("Unable to fork anymore downloads\n");
               task_free(t);
               return;
          }
          // Parent
          if(child > 0)
          {
               task_pop_peer(t);
               continue;
          }
          //go download your part.
          else{
               child_count++;
               t->peer_fd = open_socket(t->peer_list->addr, t-
>peer_list->port);
```

```
               if (t->peer_fd == -1) {
                    error("* Unable to open socket to %s:%d\n", 
inet_ntoa(t->peer_list->addr), t->peer_list->port);
                    task_free(t);
                    exit(1);
               }
               //download the part if we treat each part like a 
separate file - all checksum handling will be done in the download
               task_download( t->filename+".part"+i);

          }
     }
}
```
The iterates through the number of pieces, forking for each piece similar to the original lab4 implementation to download in parallel. I make another assumption that the task_download function can handle the separate checksum check for the individual piece file. In essence I am treating each piece like its own file. This solves our file integrity problem by comparing each file's checksum with each the metainfo file and allows us to handle the problem of bad peers sending us bad data. We have their IP associate with the block they tried to send so we can black list them later which is out of the scope of this design document.
```
//wait for all the childs to end
while(child_count-- > 0)
     waitpid(-1, NULL, 0);

reassemble();
//some function to reassemble the blocks into one contigious file
task_free(t);
}
```
Next we have a reassemble function which will handle the process of reassembling these individual part files into something contiguous and usable.
From a performance perspective this is not the best implementation. Most BitTorrent implementations use random blocks to maximize file movement across the network while I used sequential block downloads (in the sense I fork first from the lowest block). Obviously in real BitTorrent applications there are priority downloads where a peer can be given higher priority – controlling bandwidth is out of the scope of this design, but something to consider. Also in real BitTorrent applications concurrent downloading/uploading is common.

Summary:
The goal of this design project was to enable the user to download a file in parallel from other peers. While I did not specifically write out the means of which to do this, I was able to identify how one would go about doing it and how to solve the problems that arise from introducing this new behavior. Some failures is a lack of real implementation that actually worked although to test this on the OSP2P network where there are no files that have been broken up into pieces will be difficult seeing as the other peers don't use this protocol or code. Also I have determined that in order to ensure that validation is successful some metadata information must be used that is independent of the peers one is downloading from. I hope this design document has helped illustrate the means of which to implement this new feature.