XIANKAI YANG, LUYANG LI, SHANZAE NADEEM KHAN, KHIZAR HUSSAIN

DARTSYNC

May 30, 2016

Team C-Port

Final Project

Culminating Project for Networks Class offered by Professor Xia Zhou

Abstract

In this project, a completely different paradigm for synchronization of personal data across different devices is explored. Rather than relying on the centralized cloud, we enable local file exchange and synchronization across devices. Devices can directly talk to each other to obtain the latest files, and all files are stored locally. The resulting benefits are twofold. First, it naturally protects user's data privacy, since all data are stored locally and controlled by users, and no copies are made on centralized cloud. Second, it constraints traffic within a local network, which not only reduces the network traffic burden to remote cloud servers, but also leads to faster file synchronization by accessing local networks. This design paradigm is towards our vision of a "decentralized cloud"

DartSync

DartSync runs on a tracker (server) and multiple peers (clients). These can be run with arguments: ./DartSync -tracker or ./DartSync âĂŞpeer

Design Spec

The Tracker:

The tracker collects file information from all peers, maintains records of all files, and notifies peers upon any file updates. Note that the tracker does not store files, but rather only keeps file information of all peers. Specifically, the tracker has the following functionalities:

- *Maintaining file records.* The tracker collects file information from all peers and keeps the information in a file table. Each entry in the table consists of file information, the peer IP address, and the timestamp. The tracker periodically handshakes with peers to receive their file information. The tracker then compares the tracker-side and peer-side file information and see whether there are updates to broadcast to all peers. The tracker always knows which device has the newest file.
- *Monitoring online/alive peers*: The tracker should always know whether a peer is online or alive, by receiving a heartbeat (alive) message every ten minutes (or any time interval you define) from every peer. The tracker maintains a peer table for the list of alive peers, and updates the table based on the heartbeat messages it receives. If the tracker does not receive any heartbeat message from a peer for ten minutes, this peer will be deleted in the peer table

Peers:

A peer node monitors a local file directory, communicates with the tracker, and updates files if necessary

- *Monitoring a local file directory.* Users define a local file directory that contains all the files users want to synchronize. This file directory is similar to the Dropbox root directory. The peer node monitors this folder and sends out handshake messages to the tracker of any updates

- *Communicating with the tracker:* A peer node communicates with the tracker by handshake messages. The handshake message from the tracker contains the timestamps of the latest files and the list of IP addresses that own these files. A peer parses the message to know whether it needs to download any files from other peers
- *Downloading and uploading files:* Peers upload and download the newest files from other peers using Peer-to-Peer (P2P) connections. Each peer has a thread that keeps listening to messages from other peers, and another thread that creates P2P connections to upload or download files. To avoid duplicated downloads, the peer maintains a peer-side peer table that tracks all its existing P2P download threads. When multiple other peers have the latest file, the peer can request different file pieces from these peers concurrently

Implementation Specifications:

1. **TCP Connection:**
   - **Tracker To Peer:** Tracker acts as a server waiting for peers to connect using the following bit of code:

```
int tracker_init (int port){
    int track_socket_desc;

    struct sockaddr_in tracker;
    bzero(&tracker,sizeof(tracker));

    //1. initialize tracker and peer.
    tracker.sin_family = AF_INET;
    tracker.sin_port = htons(port);
    tracker.sin_addr.s_addr = htonl(INADDR_ANY);

    //2. Create TCP socket desc.
    track_socket_desc = socket(AF_INET,SOCK_STREAM, 0);
    if(track_socket_desc< 0){
        printf("tracker_init(): Cannot create tracker_socket_desc\n");
        return -1 ;
    }

    printf("%s Waiting for incoming connections...\n", tracker_trace);
```

```
    //3. bind tracker_socket.
    if(bind(track_socket_desc,(struct sockaddr *)&tracker,sizeof(tracker))
    <0){
        printf("tracker_init(): Cannot bind tracker.\n");
        return -1;
    }

    //4. listen to incoming connections.
    if(listen(track_socket_desc , MAX_PEER)<0){
        printf("tracker_init(): Cannot listen.\n");
        return -1;
    }
    //5. return tracker_socket
    return track_socket_desc;
}
```

Server Socket

- **Peer to Tracker:** Peers work as clients that connect to the tracker the centralizing system that helps them sync files with other peers:

```
int connectToTracker()
{
  struct sockaddr_in server_addr;

  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = inet_addr(TRACKER_IP);
  server_addr.sin_port = htons(TRACKER_PORT);

  int conn;
  assert((conn = socket(AF_INET, SOCK_STREAM, 0)) >= 0);
  assert(connect(conn, (struct sockaddr*) &server_addr, sizeof(server_addr)
    ) == 0);

  return conn;
}
```

Connecting to Tracker

2. **Threads:**

- **Tracker Threads:**

  (a) **Main Thread:** Listen on the handshake port and create a Handshake thread when a new peer joins:

```
void tracker_main() {

    //1.1 Initialize the peerTable of the tracker side.
    tracker_Ptable_Initialize();
    //1.2 Initialize the trackerFile table for the tracker side.
    fileTable_Initialize();

    //1.3 Initialize the tracker.

    if((tracker_socket = tracker_init(TRACKER_PORT) )<0){
        printf("%s cannot create tracker socket.\n", tracker_trace);
        exit(1);
    }
    //2. register a signal handler which is used to terminate the
    process

    signal(SIGINT,tracker_stop);

    //3.1: Create a thread to monitor all peers whether they are alive.
    pthread_t monitor_alive_thread;
    pthread_create(&monitor_alive_thread,NULL,monitor_alive,(void*)0);

    //4. Always recived the pkts sent from peer.
    while(1){

        int peer_socket;
        struct sockaddr_in peer;
  int peerlen = sizeof(peer);
        ptp_peer_t firstpkt;
        peer_socket = accept(tracker_socket,(struct sockaddr*)&peer,(
    socklen_t*)&peerlen);

        if(peer_socket>0){

            // 4.1: First pkt will be the REGISTER pkt from peer.
            TfromP_recvpkt(&firstpkt, peer_socket);

            if(firstpkt.type == REGISTER){

                time_t currTimestamp;
                time(&currTimestamp);
    printf("%s received a REGISTER packet from %s\n", tracker_trace,
```

```
        inet_ntoa(peer.sin_addr));
                    //4.2 add this peer to tracker's global peertable.
                    tracker_peer_t *tmp = tracker_Ptable_Add(firstpkt.
        peer_ip, peer_socket, currTimestamp);

                    //4.3 Send the ptp_tracker_t ack pkt to peer which will
         set the interval and piece_len of that peer.
                    //The ack pkt need has 3 fileds, 1 interval and 2
        piece_len are constants, so we only send the 3 file_table to the
        peer.
                    //Using T2P_pkt_init() to initilize this ack pkt.

                    //4.3.1: Initilize the ack pkt, set interval, piece_len
        , size, and tracFileTable
                    //                ptp_tracker_t* ackpkt;
                    //                T2P_pkt_init(ackpkt,
                    //                             ALIVE_INTERVAL,
        PIECE_LENGTH, tfsize,
                    //                             tracFileTable);

                    // T2P_send(ackpkt,peer_socket);

                    fileTable_Send(peer_socket);


                    // 5.1: using peer_socket as an parameter for handshake
         communicating with the peer.
                    pthread_t handshake_thread;
                    pthread_create(&handshake_thread, NULL, handshake, (
        void *) tmp);
                }

            }else{
        perror("tracker accept");
                printf("track_main(): Cannot accept in while loop.\n");
            }
        }
        return;
    }
```

Tracker Main

(b) **Handshake Thread:** receive messages from a specific peer and respond if

needed, by using peer-tracker handshake protocol:

```
void* handshake(void *arg){
//Peer sends a pkt to tracker for the first time, thus tracker has
to register this peer.
ptp_peer_t handshk_pkt;
memset(&handshk_pkt,0,sizeof(ptp_peer_t));
int P2Tconn = *(int*) arg;
int broadflag;
//recv>0
while(TfromP_recvpkt(&handshk_pkt, P2Tconn)>0){
    unsigned long currTime;
    switch (handshk_pkt.type) {
        case REGISTER:
            printf("handshake(): During handshake stage, it shall
not receive any REGISTER pkt.\n");
            break;
        case KEEP_ALIVE:
            //update the timestamp of peernode.
            time(&currTime);
            tracker_Ptable_UpdateTimeStamp(tracPeerTable,
handshk_pkt.peer_ip, currTime);
            break;
        case FILE_UPDATE:
            //update the file table.
            broadflag = -1;
            filenode* peerfileThead =  handshk_pkt.file_table;
            //1.: Check the peer File tabel first.
            while (peerfileThead!=NULL) {
                //1.1: if this file doesn't exist in global file
table, then we add this file to the GLOBAL file table.
                //1.2 : if this file exists in global file table,
check whether ip belongs to the same peer.
                // Both things above can be done by
fileTable_exists();
                filenode* checkglobal = fileTable_exists(
tracFileTable, peerfileThead->filename, peerfileThead->peerIP);
                if (checkglobal == NULL) {
                    fileTable_Add(tracFileTable, peerfileThead);
                    broadflag = 1;
                }
                peerfileThead = peerfileThead->next;
            }
```

```
                // 2.: Check GLOBAL file table.
                filenode* globalfileThead = tracFileTable;
                while(globalfileThead!=NULL){
                    filenode* checkpeer = fileTable_exists(handshk_pkt.
    file_table, globalfileThead->filename, globalfileThead->peerIP);
                    // If it is the peer who wants to delete this file,
     GLOBAL tracker file table will delete this node and broadcast this
     deletion.
                    if(checkpeer!=NULL){
                        fileTable_Delete(tracFileTable, checkpeer->
    filename);
                        broadflag = 1;
                    }
                    globalfileThead = globalfileThead->next;
                }
                if(broadflag == 1){
                    broadcast_fileChange(tracFileTable);
                }
                //BroadCast changes to all alive peers.
                break;
            default:
                break;
        }
    }
    return NULL;
}
```

Handshake Thread

(c) **Monitor Alive Thread:** monitor and accept alive message from online peers periodically, and remove dead peers if timeout occurs:

```
void* monitor_alive(void *arg){
    while(1){
        //sleep for MONITOR_ALIVE_INTERVAL to check periodly.
        sleep(MONITOR_ALIVE_INTERVAL);
        //Set a pointer tracker_peer_tp to traverse the whole
    trackerpeertable.
        tracker_peer_t *tracker_peer_tp;
        tracker_peer_tp = tracPeerTable;

        unsigned long currTime;
        time(&currTime);
```

```
            //check all peer's timestamp.
            while(tracker_peer_tp!=NULL){
                /*The peer is dead.*/
                if(currTime - tracker_peer_tp->last_time_stamp >
        DEAD_TIMEOUT)//remove dead Peer and delete all files.
                {
                    printf("monitor_alive(): peer has dead.\n");
                    // remove peer from tracker peer table.
                    removePeer(tracker_peer_tp->IP);
                }
                tracker_peer_tp = tracker_peer_tp->next;
            }
        }
        //pthread_exit(NULL);
    }
```

Monitor Alive Thread

- **Peer Threads:**

  (a) **Main Thread:** after connecting to the tracker, receive messages from tracker, and then create P2PDownload Threads if needed.

```
void peer_main()
{
  tracker_conn = -1;
  fileTable_Initialize();
  peerPtable_Initialize();

  file_table_mutex  = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t
    ));
  peer_peer_table_mutex = (pthread_mutex_t *) malloc(sizeof(
    pthread_mutex_t));
  pthread_mutex_init(file_table_mutex, NULL);
  pthread_mutex_init(peer_peer_table_mutex, NULL);

  tracker_conn = connectToTracker();
  if (-1 == tracker_conn) {
    printf("%s Can't connect to tracker", peer_trace);
    return;
  }
  printf("%s Connection to the tracker established.\n", peer_trace);

  myIP = my_ip();
```

```
ptp_peer_t sendpkt;
p2T_pkt_set(&sendpkt, 0, NULL, REGISTER, NULL, myIP, PEER_PORT, 0,
  NULL);

assert(pToT_sendpkt(&sendpkt, tracker_conn) == 1);

// create the alive thread
pthread_t alive_thread;
pthread_create(&alive_thread, NULL, Alive, NULL);
printf("%s Alive thread created.\n", peer_trace);

// create a P2P Listen thread
pthread_t p2p_listen_thread;
pthread_create(&p2p_listen_thread, NULL, P2PListen, NULL);
printf("%s p2p listen thread created.\n", peer_trace);


filenode *global_table;
for (;;) {
  global_table = fileTable_Receive(tracker_conn);
  if (global_table == (filenode *) 0xffffffff) {
    printf("%s Connection to the tracker is lost. Exiting...\n",
  peer_trace);
    break;
  }
  printf("%s Received a file table from the tracker!\n", peer_trace);
  pthread_mutex_lock(file_table_mutex);

  // based on the global file table, update the local file table
  filenode *cur = global_table;
  while (cur != NULL) {
    if (cur != fileTable_exists(global_table, cur->filename, NULL)) {
      // this file has been processed before
      cur = cur->next;
      continue;
    }

    filenode *tmp = fileTable_exists(file_table, cur->filename, NULL)
  ;

    if (tmp == NULL) {
      // if our local file table doesn't have this file, should we
```

```
    add it?
//        if (fileTable_exists(global_table, cur->filename, myIP) !=
    NULL)
//          // if this ip used to have the file in the global table,
    that means we just deleted locally, but the tracker hasn't updated
    it yet
//          // thus we should not add this file, cause the delete
    action is the latest one
//          continue;
//        }
//        else {
//          // if this ip never has this file, we should add it
          fileTable_Add(cur);
          Download(cur->filename, cur->filesize, global_table);
//        }
      }

      // if out local file table has this file, should we update it?
      else {
        // if this file is newly created but hasn't been marked with a
    timestamp from the tracker
        if (tmp->timestamp == 0) {
//          filenode *p = fileTable_exists(global_table, cur->filename,
     myIP);
//          if (p != NULL && p->timestamp == fileTable_latest(
    global_table, cur->filename)) {
//            // update the timestamp given from the tracker
            tmp->timestamp = cur->timestamp;
//          }
//          else {
//            // some peer created this new file with the same name
    first or updated it first
//            // then we shuold change our name
//            strcat(strcat(tmp->name, " from "), myIP);
//          }
        }
        else {
          if (difftime(tmp->timestamp, fileTable_Latest(global_table,
    cur->filename)) == 0) {
            // this file is the latest one, do nothing
            continue;
          }
```

```
            else {
              // not the latest one
              // need to update
              fileTable_Update(cur);
              Download(cur->filename, cur->filesize, global_table);
            }
          }
        }
        cur = cur->next;
      }

      //traverse the local file table to see if there's anything need to
      be deleted
      cur = file_table;
      while (cur != NULL) {
        if (fileTable_exists(global_table, cur->filename, NULL) == NULL)
      {
          // the global table doesn't have this file and the timestamp
      used to be assigned by the tracker
          fileTable_Delete(cur->filename, myIP);
        }
        cur = cur->next;
      }

      pthread_mutex_unlock(file_table_mutex);
    }

    close(tracker_conn);
    pthread_mutex_destroy(file_table_mutex);
    pthread_mutex_destroy(peer_peer_table_mutex);
    free(file_table_mutex);
    free(peer_peer_table_mutex);
    // there might be more to free
}
```

Main Thread

(b) **P2P listening thread:** listen on the P2P port; when receiving a data request from another peer, create a P2PUpload Thread:

```
void *P2PListen(void *arg)
{
  int serv_sd = socket(AF_INET, SOCK_STREAM, 0);
  struct sockaddr_in server_addr;
```

```
  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family      = AF_INET;
  server_addr.sin_addr.s_addr   = htonl(INADDR_ANY);
  server_addr.sin_port      = htons(P2P_PORT);

  bind(serv_sd, (struct sockaddr *) &server_addr, sizeof(server_addr));
  listen(serv_sd, 1);

  while (1) {
    struct sockaddr_in peer_addr;
    socklen_t peer_addr_len = sizeof(peer_addr);
    int *conn = (int *) malloc(sizeof(int));
    *conn = accept(serv_sd, (struct sockaddr *) &peer_addr, &
    peer_addr_len);
    if (*conn < 0) {
      free(conn);
      perror("P2PListen: accept");
      return NULL;
    }
    //analyse the data request from another peer
    //create a P2PUpload thread if needed
    pthread_t p2p_upload_thread;
    pthread_create(&p2p_upload_thread, NULL, P2PUpload, (void *) conn);
  }
}
```

Listening Thread

(c) **P2PDownload thread:** download data from the remote peer

```
// download data from the remote peer
void *P2PDownload(void *arg)
{
  peer_peer_t *p = (peer_peer_t *) arg;

  // send the required name and range
  assert(send(p->sockfd, p->filename   , sizeof(p->filename) , 0) > 0);
  assert(send(p->sockfd, &p->start   , sizeof(p->start)   , 0) > 0);
  assert(send(p->sockfd, &p->end     , sizeof(p->end)   , 0) > 0);

  // read file from peer
  FILE *f;
  f = fopen(p->filename, "r+");
```

```
  assert(f != NULL);
  fseek(f, p->start, SEEK_SET);

  char buf[MAX_DATA_LEN];

  // recv and gather
  while (p->start != p->end) {
    int l = MIN(p->end - p->start, MAX_DATA_LEN);
    assert(recv(p->sockfd, &buf, l, 0) > 0);
    fwrite(buf, l, 1, f);
    p->start += l;
  }

  fclose(f);
  pthread_mutex_lock(peer_peer_table_mutex);
  peerPtable_Delete(p);
  pthread_mutex_unlock(peer_peer_table_mutex);
  return NULL;
}
```

P2PDownload Thread

(d) **P2PUpload Thread:** upload data to the remote peer:

```
void *P2PUpload(void *arg)
{
  int conn = *(int *) arg;

  char filename[FILE_NAME_LEN];
  int start;
  int end;

  // recv the required name and range
  assert(recv(conn, filename  , sizeof(filename)  , 0) > 0);
  assert(recv(conn, start   , sizeof(start)   , 0) > 0);
  assert(recv(conn, end   , sizeof(end)   , 0) > 0);

  File *f;
  f = fopen(filename, "r");
  assert(f != NULL);
  fseek(f, start, SEEK_SET);

  char buf[MAX_DATA_LEN];
```

```
  // divide and send
  while (start != end) {
    int l = MIN(end - start, MAX_DATA_LEN);
    fread(buf, l, 1, f);
    assert(send(conn, &buf, MAX_DATA_LEN, 0) > 0);
    start += l;
  }

  fclose(f);
  return NULL;
}
```

P2PUpload Thread

(e) **File Monitor Thread:** monitor a local file directory; send out updated file table to the tracker if any file changes in the local file directory:

```
//monitor a local file directory; send out updated file table to the
    tracker if any file changes in the local file directory
void *FileMonitor(void *arg)
{
  ptp_peer_t sendpkt;
  p2T_pkt_set(&sendpkt, 0, NULL, FILE_UPDATE, NULL, myIP, PEER_PORT, 0,
     NULL);

  // create a loval file directory to monitor
  int status = mkdir("./Dart_Sync", S_IRWXU | S_IRWXG | S_IROTH |
    S_IXOTH);
  if (status != 0) {
    perror("mkdir");
    return NULL;
  }
  //monitor a local file directory
  int length, i = 0;
  int fd;
  int wd;
  char buffer[BUF_LEN];

  fd = inotify_init();

  if (fd < 0) {
    perror("inotify_init");
  }
```

```
wd = inotify_add_watch(fd, "./Dart_Sync", IN_MODIFY | IN_CREATE |
  IN_DELETE);

while (1) {
  length = read( fd, buffer, BUF_LEN );
  if (length < 0) {
      perror("read");
  }
    while (i < length) {
    pthread_mutex_lock(file_table_mutex);
    struct inotify_event *event = (struct inotify_event *) &buffer[ i
   ];
    if (event->len) {
      if (event->mask & IN_CREATE) {
        if (event->mask & IN_ISDIR) {
          printf("The directory %s was created.\n", event->name);
        }
        else {
          printf("The file %s was created.\n", event->name);
        }

        filenode tmp;
        strcpy(tmp.filename, event->name);
        tmp.filesize = getFileSize(tmp.filename);
        tmp.timestamp = 0;
        strcpy(tmp.peerIP, myIP);

        // try to add this node into the local file table
        // if there's already such a node in the table, that means
  this file is downloaded
        // otherwise it's created
        fileTable_Add(&tmp);

        // either way we should send the updated table to the tracker
        assert(pToT_sendpkt(&sendpkt, tracker_conn) == 1);
      }
      else if (event->mask & IN_DELETE) {
        if (event->mask & IN_ISDIR) {
          printf("The directory %s was deleted.\n", event->name);
        }
        else {
          printf("The file %s was deleted.\n", event->name);
```

```
            }

            filenode *p = fileTable_exists(file_table, event->name, myIP)
        ;
            if (p != NULL) {
              // if the file is still in the table, that means the peer
        itself delete it instead of the tracker told it to
              fileTable_Delete(event->name, myIP);
              // then we need to send the updated table to the tracker
              assert(pToT_sendpkt(&sendpkt, tracker_conn) == 1);
            }
          }
          else if (event->mask & IN_MODIFY) {
            if (event->mask & IN_ISDIR) {
              printf("The directory %s was modified.\n", event->name);
            }
            else {
              printf("The file %s was modified.\n", event->name);
            }

            filenode *p = fileTable_exists(file_table, event->name, myIP)
        ;
            if (p != NULL) {
              // the file is modified locally, reset the timestamp and
        send to the tracker
              p->timestamp = 0;
              assert(pToT_sendpkt(&sendpkt, tracker_conn) == 1);
            }
            else {
              printf("I should not be here!\n");
            }
          }
        }
        pthread_mutex_unlock(file_table_mutex);
      }
    }

    (void) inotify_rm_watch(fd, wd);
    (void) close(fd);
    exit(0);
}
```

File Monitor Thread

(f) **Alive Thread:** send out heartbeat (alive) messages to the tracker to keep its online status:

```
//send out heartbeat (alive) messages to the tracker to keep its online
    status
void *Alive(void *arg)
{
  ptp_peer_t sendpkt;
  p2T_pkt_set(&sendpkt, 0, NULL, KEEP_ALIVE, NULL, myIP, PEER_PORT, 0,
    NULL);
  while (1) {
    sleep(ALIVE_INTERVAL);
    assert(pToT_sendpkt(&sendpkt, tracker_conn) == 1);
  }
}
```

Alive Thread

3. **Data Structures:**

- Main File Table:

```
typedef struct node{
  int filesize;
  char filename[FNAME_LEN];
  unsigned long timestamp;
  struct node *next;
  char peerIP[IP_LEN];
}filenode;
```

Main File Table Node

- Tracker side Peer Table:

```
typedef struct _tracker_side_peer_t{
  char IP[IP_LEN];
  unsigned long last_time_stamp;
  int sockfd;
  struct _tracker_side_peer_t *next;
}tracker_peer_t;
```

Tracker Side Node

- Peer File Table:

```
typedef struct peer_fnode{
  int filesize;
  char filename[300];
  unsigned long timestamp;
  struct node *next;
  char peerIP[16];
  int peerNum;
  int status;
}peer_filenode;
```

Peer Side Node

- Peer To Tracker Segment:

```
//The packet data structure sending from peer to tracker
typedef struct segment_peer {
  //Protocol Length
  int protocol_len;
  //protocol name
  char protocol_name[PROTOCOL_LEN + 1];
  //packet type : register, keep alive, update file table
  int type;
  //reserved space, you could use this space for your convenient, 8
bytes by default
  char reserved[RESERVED_LEN];
  //the peer ip_address sending this packet
  char peer_ip[IP_LEN];
  //listening port number in p2p
  int port;
  //the number of files in local file table -- optional
  int file_table_size;
  //file table of the client -- your own design
  filenode *file_table;
} ptp_peer_t;
```

Peer To Tracker Segment

- Tracker To Peer Segment:

```
typedef struct segment_tracker {
  //time interval that the peer should sending alive messages
periodically
```

```
    int interval;
    //piece length:
    int piece_len;
    //packet type : register, keep alive, update file table
    int file_table_size;
    //file table of the tracker -- your own design
    filenode file_table;
  } ptp_tracker_t;
```

Tracker To Peer Segment

4. **Additional Features:**

- **Downloading from multiple Peers:**

```
// partition the file and download from different peers
void Download(char *name, int size, filenode *global)
{
  FILE *f;
  f = fopen(name, "r");
  if (f == NULL) {
    f = fopen(name, "w");
    char *buf = (char *) malloc(sizeof(char) * size);
    fwrite(buf, size, 1, f);
    free(buf);
  }
  fclose(f);

  filenode *p = global;
  int count;
  time_t latest = fileTable_Latest(global, name);

  while ((p = fileTable_exists(p, name, NULL)) != NULL && difftime(p->
    timestamp, latest) == 0) {
    count++;
  }

  int seg_len = size / count;
  int start = 0;
  p = global;
  while ((p = fileTable_exists(p, name, NULL)) != NULL && difftime(p->
    timestamp, latest) == 0) {
```

```
            pthread_mutex_lock(peer_peer_table_mutex);
            peer_peer_t *peer_t = peerPtable_Add(p, start, MIN(size, start +
            seg_len));
            pthread_mutex_unlock(peer_peer_table_mutex);
            pthread_t p2p_download_thread;
            pthread_create(&p2p_download_thread, NULL, P2PDownload, (void *) peer_t
            );
        }
    }
```

Multi-Peer Download Implementation

- **Using Raspberry Pi:** We were able to integrate our program into the Raspberry Pi and use the Pi as the tracker device as follows:

Difficulties Faced During Project Work: