DartSync Final Project Report

Team HelloWorld
Tristan Chu, Wei Huang, Matt Krantz, Jinzheng Sha

June 2, 2015

1 Introduction

Dartsync is an application that enables data and file sharing between users. In contrast to popular services that rely on cloud-based storage, Dartsync employs local file exchange and synchronization across device in order to create data and file sharing capabilities. This method of data sharing has two benefits. First, the user's data privacy is virtually assured, as all files and data are stored locally and managed by users. No extra copies are created on a centralized cloud, which severely reduces the possibility of information theft. Second, traffic is constrained within a local network, which minimized traffic burden and allows for faster file synchronization.

Dartsync operates on a local, perpetually-on node referred to as a "tracker," and a series of personal devices controlled by individual users referred to as "peers." Users specify the files that they wish to synchronize across peers, and the tracker will enable this synchronization upon file update, addition, or removal. In our implementation of Dartsync, we implementing the tracker node using a miniature Linux machine called Raspberry Pi (more information can be found at the Pi site, http://www.raspberrypi.org).

This document describes the input, data flow, and output specifications for the main modules of DartSync (components relating to the tracker/server and peer/client). The pseudo code for these modules is also given. After we detail our design specifications, we discuss lessons learned, future applications, and goals going forward.

2 Design Overview

2.1 Tracker

The tracker node can be considered the central hub of activity that manages the information sent and received by all peer nodes, maintaining file records and notifying peers upon any relevant file updates. An important distinction is that the tracker does not store files, but instead only keeps file information on all peers' nodes. This file information consists of `FILE INFORMATION`, the `PEER IP ADDRESS`, and the `TIMESTAMP`. The tracker will periodically handshake with peer nodes to receive their file information; the tracker then compares tracker-side and peer-side file information to determine if it needs to broadcast updates to all peers. The tracker will also keep track of which peer has the newest file. The handshake will also tell the tracker which peers are online/alive. If a peer is online/alive, the tracker will receive an update via a "heartbeat" message at a set time interval. The tracker will maintain a peer table for the list of active peers and updates it regularly. Failure to receive a "heartbeat" message from a peer will result in the peer being deleted from the peer table.

2.2 Peer

Peer nodes will each individually monitor a local file directory, communicating with the tracker node and updating files as necessary. Each peer node will send out a handshake message to the tracker node if updates are made within the local file directory. The handshake message from the tracker will contain the timestamps of the latest files and a list of IP addresses that own these files. Each peer node will determine via this handshake whether or not it needs to download files from other peers. Peers utilize Peer-to-Peer (P2P) connections to upload and download files from each other. Each peer has a thread that listens for messages from other peers, as well as a thread to create P2P connections. Peers will maintain a peer-side peer table to track all existing P2P download threads. If multiple peers have a file, peers can request different pieces of said file from these peers concurrently.

3 Features

Our program currently features the following baseline features:

- Compatibility with Linux systems
- TCP connection and data transfer
- Local file monitoring
- Synchronization of multiple files by comparing timestamps
- File replacement when updating files
- Data retrieval from multiple peers
- Tracker and peer implementation as described above

In addition, our program supports the following extra credit features:

- Dedicated tracker node implemented on Raspberry Pi
- Resume from partial download
- Compatability with OSX, Linux, and Raspberry Pi
- Compression and decompression of files
- Authentication and password protection
- Dynamic implementation of program (no configuration file necessary)
- Synchronization of multiple folders

- Basic conflict resolution
- Delta updates (not included in final submission)
- GTK GUI (not included in final submission)

4 Module Specifications

4.1 Heartbeat Module

(1) Input: Any inputs to the module

Peer Side Functions
`void* heartbeat(void* arg)` — thread function used to send heartbeat

Peer Side Parameters
`SIGNAL_HEARTBEAT` — signal type of heatbeat
`HEARTBEAT_INTERVAL_SEC` — the frequency of heartbeat

Tracker Side Functions
`void* handshake_handler(void* arg)` — thread used to receive all incoming data from a peer
`void* peer_live_handler(void* arg)` — thread used to check the peer table
`void peer_table_add(unsigned long ip, int socket)` — function used to add a peer to peer table
`void peer_table_update_timestamp(unsigned long ip)` — function used to update the timestamp of a given peer

Tracker Side Parameters
`HEARTBEAT_INTERVAL_SEC` — the frequency of heartbeat
`client_handshake_socket` — socket used by the `handshake_handler` to get data from peer
`peer_table` — peer table maintained by tracker including all alive peer info

(2) Output: Any outputs of the module

Peer Side
The heartbeat() function will send a heartbeat signal `SIGNAL_HEARTBEAT` to the tracker every set number of seconds. We define the value of the these parameters in `network.h`

Tracker Side
Each time a peer registers with the tracker, tracker will call the `peer_table_add()` function to add the new peer to the peer table. In the `handshake_handler()`, if the tracker receives a signal with type `SIGNAL_HEARTBEAT`, it will call the `peer_table_update_timestamp()` function to update the timestamp of a peer. There is a `peer_live_handler` that checks the peer table periodically, deleting out-of-date peers from the peer table.

(3) Data Flow: Any data flow through the module

See image on next page!

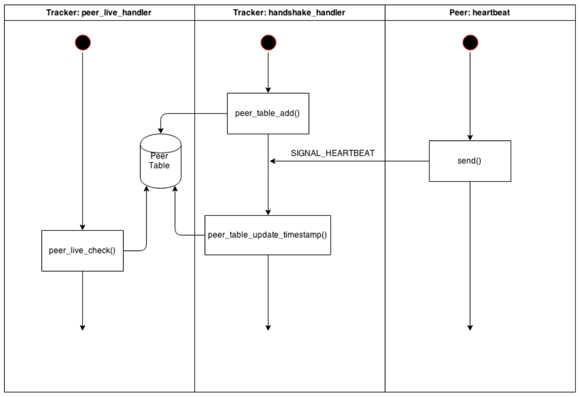
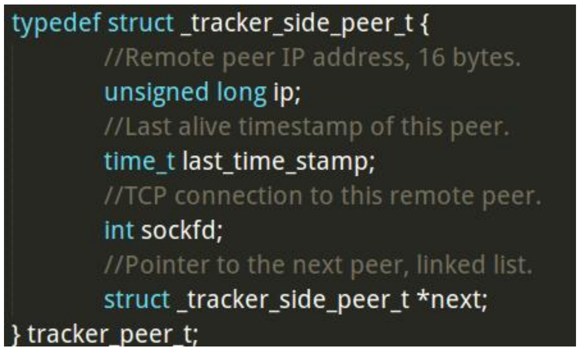

Tracker: peer_live_handler
Tracker: handshake_handler
Peer: heartbeat
peer_table_add()
Peer Table
SIGNAL_HEARTBEAT
send()
peer_live_check()
peer_table_update_timestamp()

Figure 1: Data flow through the heartbeat module

(4) Data Structures: Major data structures used by the module



```
typedef struct _tracker_side_peer_t {
        //Remote peer IP address, 16 bytes.
        unsigned long ip;
        //Last alive timestamp of this peer.
        time_t last_time_stamp;
        //TCP connection to this remote peer.
        int sockfd;
        //Pointer to the next peer, linked list.
        struct _tracker_side_peer_t *next;
} tracker_peer_t;
```

Figure 2: Peer table data structure for heartbeat module

(5) Pseudo Code: Pseudo code description of the module

Peer Side

```
void* heartbeat(void* arg) {
    while (main_thread_alive) {
        send SIGNAL_HEARTBEAT to tracker
        sleep(HEARTBEAT_INTERVAL_SEC)
    }
}
```

Tracker Side

```
void* handshake_handler(void* arg) {
   while (recv(client_handshake_socket, SIGNAL)) {
      switch (SIGNAL) {
         case SIGNAL_HEARTBEAT:
            call peer_table_update_timestamp
            ...
      }
   }
}

void* peer_live_handler(void* arg) {
   while (main_thread_alive) {
      delete out-of-date peer from peer table
      sleep
   }
}
```

4.2 File Monitor Module

(1) Input: Any inputs to the module

Functions

`void* file_checker(void* arg)` — thread in peer that periodically checks the local directory and sends file table to tracker if any change detected
`int file_table_update()` — function called by `file_checker()` that will update the file table and tell whether there is change
`int file_table_update_helper(char* directory, file_node** last)` — helper for above function, called recursively to get updated file node and detect changes

Parameters

`file_table` — the head of linked list of file node
`root_directory` — top level directory that peer would monitor
`last_table_update_time` — variable used to record the time of last scan of the root directory
`SIGNAL_FILE_UPDATE` — signal type of file update
`FILE_UPDATE_INTERVAL_SEC` — frequency of file monitor

(2) Output: Any outputs of the module

The thread function `file_checker()` sets the root directory to the directory to monitor. Then it calls `file_table_update()` every set number of seconds. If any change is detected, it will send the file table to the tracker

(3) Data Flow: Any data flow through the module

See image below!

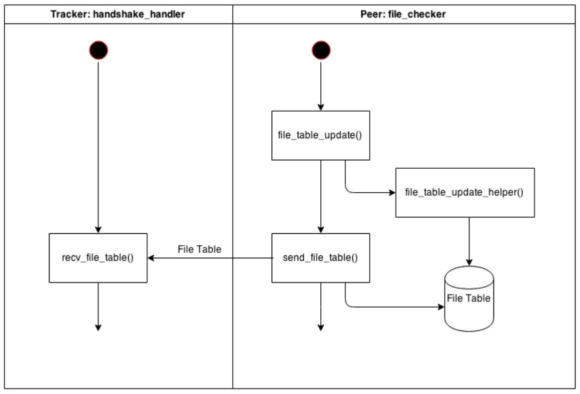
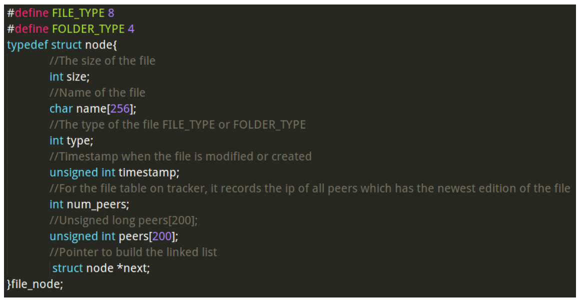


Figure 3: Data flow through the file monitor module

(4) Data Structures: Major data structures used by the module



```
#define FILE_TYPE 8
#define FOLDER_TYPE 4
typedef struct node{
        //The size of the file
        int size;
        //Name of the file
        char name[256];
        //The type of the file FILE_TYPE or FOLDER_TYPE
        int type;
        //Timestamp when the file is modified or created
        unsigned int timestamp;
        //For the file table on tracker, it records the ip of all peers which has the newest edition of the file
        int num_peers;
        //Unsigned long peers[200];
        unsigned int peers[200];
        //Pointer to build the linked list
         struct node *next;
}file_node;
```

Figure 4: File table data structure as used by the file monitor module

(5) Pseudo Code: Pseudo code description of the module

```
void* file_checker(void* arg) {
    set root directory for monitoring
    while (main_thread_alive) {
        if (file_table_update()) {
            send SIGNAL_FILE_UPDATE to tracker
            send_file_table updated file table to tracker
         }
         sleep(FILE_UPDATE_INTERVAL_SEC);
    }
}

int file_table_update() {
    if (update_enable) {
        delete the previous file table
        save the current time to tmpt_time
        // Call the file_table_update_helper()
        // If any change detected, the functions return 1, else return 0
        int is_updated = file_table_update_helper(root_directory)
        last_table_update_time = tmpt_time;
        return is_updated;
    } else {
        return 0;
    }
}

int file_table_update_helper(char* directory, file_node** last) {
    int is_update = 0
    if the directory's last modified time is later than last_table_update_time {
        set  is_update to 1
    }
    open the directory {
        for each file in the directory {
            create a file node about the file and append it to the file table
            if the file's last modified time is later than last_table_update_time {
                set is_update to 1
            }
        }
        for each folder in the directory {
            create a file node about the folder and append it to the file table
            recursively call the file_table_update_helper() to scan the subdirectory
            if the inner file_table_update_helper() function returns 1 {
                set is_update to 1
            }
        }
    }
    return is_update
}
```

4.3 Tracker Sync Module

(1) Input: Any inputs to the module

Functions

`void* handshake_handler(void* arg)` — thread used to receive all incoming data from a peer
`void sync_from_client(file_node* client_table)` — function called by tracker to update the global file table
`void broadcast_file_table()` — called by tracker to broadcast the updated file table to all the peers

Parameters

`file_table` — the head of linked list of global file table

`fclient_table` — the head of the linked list of peer file table
`SIGNAL_FILE_UPDATE` — signal type of file update

(2) Output: Any outputs of the module

Once a peer detects any change in the monitoring directory, it will send a file table including the information of all the files in the monitoring directory to the tracker. On the tracker side, the `handshake_handler` thread will receive the file table and try to synchronize the tracker side file table with the incoming peer side file table after comparing the global file table with the local file table from the peer. The tracker will decide which node(s) to add, delete or update. Below are the rules used by tracker to manage the global file table.

Tracker Side Synchronizing Process

There are three types of files:

Type 1: File exists in both the peer side file table and the tracker side file table. Compare timestamp. if the tracker side file table has a newer version of this file, the tracker simply ignores it. If the peer side file table has a newer version of this file , the tracker will update the time stamp of this file in the tracker side file table and update the peer list of the file to only contain this peer. If both the tracker and peer side file table have the same version of the file, the tracker will add the peer to the peer list of the files (if this peer does not exist in the peer list of this file).

Type 2: File exists in the peer side file table, but does not exist in the tracker side file table.Tracker will add this file to the tracker side file table.

Type 3: File exists in the tracker side file table, but does not exist in the peer side file table. Tracker will look in the the peer list of this file in the tracker side file table. If the peer exists in the peer list of this file, tracker will delete the file node from tracker side file table. If the peer is not in the peer list of this file, tracker will ignore it.

There are three types of folders:

Type 1: Folder exists in both the peer side file table and the tracker side file table. The tracker will add the peer to the peer list of the folder, if this peer does not exist in the peer list of this folder.

Type 2: Folder that exists in the peer side file table, but does not exist in the tracker side file table. Tracker will add this folder to the tracker side file table.

Type 3: Folder that exists in tracker side file table, but does not exist in the peer side file table. Tracker will look in the the peer list of this folder in the tracker side file table. If the peer exists in the peer list of this folder, tracker will delete the file node from tracker side file table. If the peer does not exist in the peer list of this folder, the tracker will ignore it.

Corner Case: If a file in the tracker side has the same name as a folder in the peer side, or vice versa. In this case we will update the type and timestamp of the file node in the tracker side file table. After the tracker finishes synchronizing, it will broadcast the newer version of tracker side file table to all the peers.

(3) Data Flow: Any data flow through the module

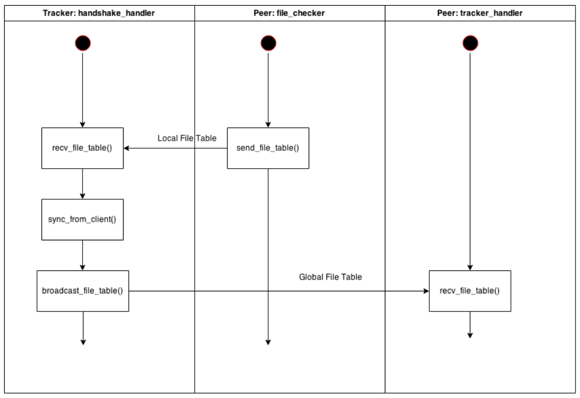


Figure 5: Data flow through the tracker sync module

(4) Data Structures: Major data structures used by the module

Peer table and file table from above!

(5) Pseudo Code: Pseudo code description of the module

```
void* handshake_handler(void* arg) {
   while (recv(client_handshake_socket, SIGNAL)) {
       switch(SIGNAL) {
           case SIGNAL_FILE_UPDATE:
               receive file table from peer
               call sync_from_client() to update the global file table
               call broadcast_file_table() to broadcast lastest global file table
               ...
        }
    }
}

void sync_from_client(file_node* client_table) {
    for each file node {
        if (file node exist in global file table, but not in peer file table) {
            check whether ip of that peer exists in peer ip list of that file node
                if (ip exist) {
                    delete the file node from global file node
                } else {
```

```
                peer should add the file node, so tracker ignore it
            }
        } else if (file node exist in peer file table, but not in global file table) {
            tracker simply add this file node to the global file node
        } else if (both file table have this node) {
            compare the time stamp of the file
            if (tracker has a newer time stamp) {
                peer should update the file node, so tracker ignore it
            } else if (peer has a newer time stamp) {
                tracker update the file node
            } else {
                tracker add the ip address to the ip list of that file node
            }
        }
    }
}
```

4.4 Peer Sync Module

(1) Input: Any inputs to the module

Functions
`void* tracker_handler(void* arg)` — thread function that receives file table broadcast by tracker
`void sync_with_server()` — called by the peer to update the local directory using the file table from tracker

Parameters
`file_table` — the head of the linked list of peer file table
`server_table` — the head of linked list of global file table

(2) Output: Any outputs of the module

Once the tracker finishes synchronizing the global file table, it will broadcast the latest global file table to all the peers. On the peer side, the tracker handler thread will receive the file table and try to synchronize the local file table with the incoming tracker side file table. After comparing the global file table with the local file table, the peer will decide which file(s) to add, delete or update. Below are the rules used by peer to manage the local directory.

Peer Side Synchronizing Process

There are three types of files:

Type 1: File exists in both the peer side file table and the tracker side file table. Compare the time stamp. If the tracker side file table has a newer version of this file, the peer will replace the local file will the newer version by downloading it from other peers. If the peer side file table has a newer version of this file or both tracker and peer side file tables have the same version of file, the peer just ignores it.

Type 2: File that exists in the peer side file table, but does not exist in the tracker side file table. Peer will delete the file.

Type 3: File that exists in tracker side file table, but does not exist in the peer side file table. Peer will download the file from other peers.

For folders, in the case of type one, the peer will ignore it. And in the cases of type 2 and type 3, the peer will deal with it as if it is a file.

Corner Case: If the file in the tracker side has the same name with a folder in the peer side, or vice versa. In this case we will update the type and time stamp of the file node in the tracker side file table. Note that during the peer side synchronizing process the file monitor process will be blocked.

(3) Data Flow: Any data flow through the module

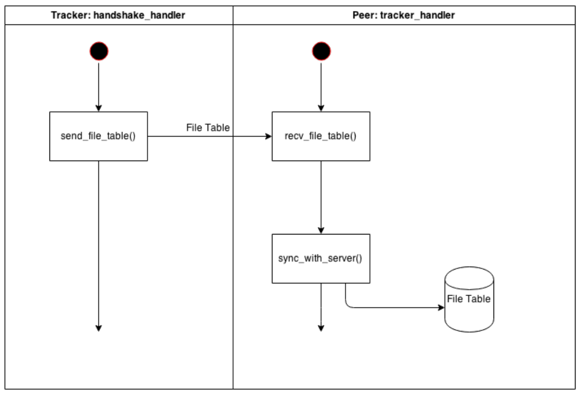


Figure 6: Data flow through the peer sync module

(4) Data Structures: Major data structures used by the module

File table from above!

(5) Pseudo Code: Pseudo code description of the module

```
void* tracker_handler(void* arg) {
    while (main_thread_alive) {
        receive file table from tracker
        compare the local file table and the global file table received
        decide which file to add, delete or update
    }
}
```

```
void sync_with_server(file_node* server_table) {
    for each file node {
        if (file node exist in global file table, but not in peer file table) {
            peer would download this file from other peer.
        } else if (file node exist in peer file table, but not in global file table) {
            peer would delete the file
        } else if (both file tables have this node) {
            compare the time stamp of the file
            if (tracker has a newer time stamp) {
                peer should update the file
            } else {
                ignore it
            }
        }
    }
    if (peer make any change to the local directory) {
        call file_table_update_helper() to update the local file table
    }
}
```

4.5 Download Module

(1) Input: Any inputs to the module

Functions
`void download_file_multi_thread(file_node* f_node)` — download files from one or more peers
`void* download_handler(void* arg)` — handle individual downloads

Parameters
`file_node* f_node` — contains the information of the file to be downloaded, including file name, file size, file timestamp, and a list of peers who currently have the file

(2) Output: Any outputs of the module

If the module exits normally, it will finish downloading the file. Otherwise, it will create several temporary files, each containing a trunk of the file. The temporary file name is of format file name + start offset of the trunk + expected length of the trunk + timestamp . Using the start offset and expected length information stored in the temporary file name, we can resume downloading when the peer is back online.

(3) Data Flow: Any data flow through the module

After the file node parameter is passed into the function, the function will connect to uploading peers, and send to them the information of the file it is requesting. The downloading peer will also tell each uploading peer which part of the file it wants to download. Then the uploading peers begin to send the requested data 1 KB at a time. On the downloading side, we use a buffer to keep receiving the stream data. Once the buffer is full, we will write the buffer content to local temporary files.

(4) Data Structures: Major data structures used by the module

In this module, the main data structure is used to transmit messages between downloading peer and uploading peer. We create a structure called `pr_msg` containing the following information:
`filename` — name of the file downloading peer requests
`piece_start_idx` — start index of the trunk downloading peer requests from the specific uploading peer
`piece_len` — length of the trunk

(5) Pseudo Code: Pseudo code description of the module

```
void download_file_multi_thread(file_node* f_node) {
    parse file name
    if (some parent folders the file not exist) {
        create those folders
    }
    num = 0
    if (there are not temporary files) {
        num = number of active peers
        for i from 1 to num {
            calculate the start index and length of the trunk
            create a thread and start the new downloading
        }
    } else {
        num = number of temporary files
    }
    // Resume partial downloading
    while (not all temporary file lengths equal to expected lengths) {
        for each temporary file not being downloaded {
            if the actual size does not equal the expected size {
                calculate the start index and length of the reset part
                wait to find an available peer
                create a thread and pass download_handler() into the thread
            }
        }

        while (not all downloading sub-threads have returned) {
            sleep
        }
    }
    merge all the temporary files
    return
}

void* download_handler(void* arg) {
    create a TCP socket sockfd and connect to uploading peer
    create a pr_msg pr_msg struct, assign the start index and trunk length to it
    send the struct to the specific uploading peer
    recv_len = 0
    while (recv_len less than expected trunk length) {
        recv from socket and put in buffer
        decompress the received data
        write the decompressed data to local temporary file
        increment by 1 KB
    }
}
```

4.6 Upload Module

(1) Input: Any inputs to the module

Functions
`void* peer_handler_multi_thread(void* arg)` — listen for incoming download request
`void* upload_handler(void* arg)` — handle specific download request

Parameters
The first function does not need any parameters. For the second function, the TCP connection descriptor is passed in as a parameter.

(2) Output: Any outputs of the module

There is no output since this module is responsible for uploading data.

(3) Data Flow: Any data flow through the module

After creating the TCP connection descriptor and passing it into the `upload_handler()` function, this function will receive a `pr_msg` struct from the downloading peer. Based on the information in the struct, the uploading peer will send the corresponding trunk of file to the other side by transferring 1 KB at a time.

(4) Data Structures: Major data structures used by the module

The data structure used in this module is the `pr_msg` struct, which contains the name of the file, start index of the trunk, and the size of the trunk.

(5) Pseudo Code: Pseudo code description of the module

```
void* peer_handler_multi_thread(void* arg) {
    create a server socket
    while (1) {
        listen on the server socket
        once accepted, create a new thread, and pass upload_handler() into it
    }
}

void* upload_handler(void* arg) {
    receive the pr_msg structure
    get the file name, start index, and trunk length
    send_len = 0
    while (send_len less than trunk length) {
        read the trunk into buffer
        compress the data in the buffer
        send and increment sen_len appropriately
    }
}
```

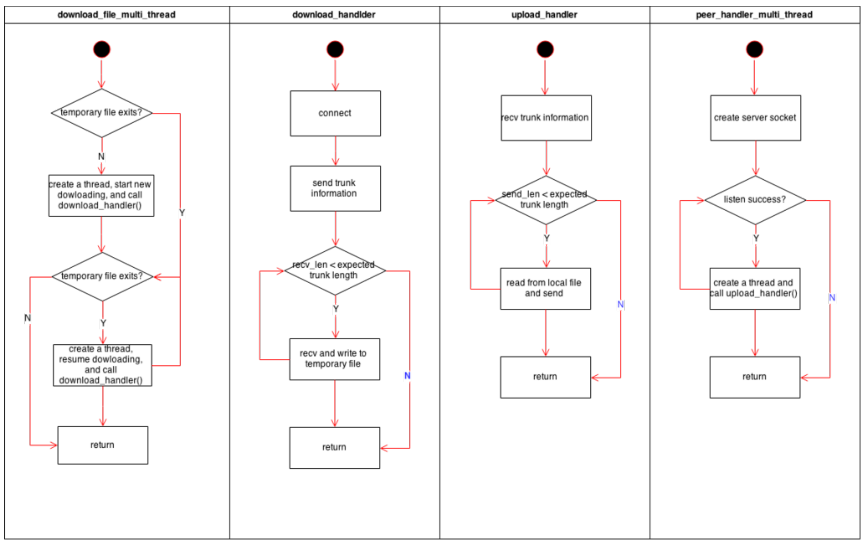

download_file_multi_thread
temporary file exits?
N
create a thread, start new dowloading, and call download_handler()
Y
temporary file exits?
N
Y
create a thread, resume dowloading, and call download_handler()
return
download_handlder
connect
send trunk information
recv_len < expected trunk length
Y
recv and write to temporary file
N
return
upload_handler
recv trunk information
send_len < expected trunk length
Y
read from local file and send
N
return
peer_handler_multi_thread
create server socket
listen success?
Y
create a thread and call upload_handler()
N
return

Figure 7: Data flow through the upload and download modules

5 Extra Credit Information

5.1 Partial Download

Assume there are three peers: A, B, and C. If Peer A is downloading a file from Peer B and Peer C, and both B and C suddenly go offline before A completes the download, Peer A will store a temporary file which stores the progress Peer A has made in downloading the file. When any peer with the file comes back online, Peer A will resume downloading the file from wherever it left off.

This is especially useful for large files; if connection is unstable and peers with the file accidentally go offline, Peer A does not need to restart downloading from the beginning, improving efficiency and quality of life for the user. The design and implementation of this feature is described in more detail above.

5.2 Compression and Decompression

When uploading and downloading files, peers will take the total size of the file and compress/decompress it using the data compression algorithm supplied by the `zlib` software library. After compressing/decompressing the file, peers will proceed to upload/download in chunks like before. This is also useful for sending large files, and allows us to send files we otherwise would not feasibly be able to send.

5.3 Cross Platform Compatibility

Our implementation of DartSync runs on OS X, Linux, and the Raspberry Pi. The main challenge involved in this was managing portability in bytes, such as ensuring our code worked with 64 bit and 32 bit platforms. Obviously, the more platforms on which our code runs the better, as this expands the range of products that could run our application and provides easier accessibility to the consumer.

5.4 Multiple Folders

Our implementation of DartSync allows for not only the synchronization of multiple files, but also for the synchronization of multiple file folders. This allows us to recursively monitor everything below our root directory (again, see above for more information about how we implemented this feature).

5.5 Conflict Resolution

Our implementation of DartSync accounts for conflict resolution by always updating based on the file with the latest timestamp. This way, if there are two different version of a file, the one that other peers update based on will always be the newest file.

5.6 Dynamic Implementation

When implementing DartSync, we avoiding using a config file by prompting the user for inputs whenever possible. This way, the user has more flexibility when running DartSync, which improves the user experience and quality of life.

5.7 Password Authentication

When implementing DartSync, we add some measure of security by prompting the user for a password when attempting to connect to the tracker. Additionally, the password is not displayed when the user enters it, which also increases the level of security provided.

6 Lessons Learned

While working on our project, we utilized two important tools in order to improve the experience of coding as a group. The first tool we used was Github, which we used to maintain stable versions of our code before we made any major changes. Our core philosophy throughout the project was to maintain an incremental build; we would ensure that we always had a stable, thoroughly tested version of our code before implementing any new or major changes. This way, if our new code broke anything or introduced any major bugs, we would always have a stable version to fall back on. The second major tool we used was Slack, which is a group-chatting communications tool that has several useful features such as multiple channels within the same group chat for discussing different subjects, the ability to share snippets of formatted code, and the ability to share images. By using Slack, we were always able to stay up to date on what our respective jobs were, when we were meeting, and generally what needed to be done. We were also able to immediately share any bugs or interesting edge cases that we discovered, so the group as a whole could look them over and discuss.

Our use of these two tools taught us two important lessons while doing the DartSync project: the importance of maintaining stable working versions of our code, and the importance of communication. After talking to other groups and discussing the challenges they faced, our commitment to slow, incremental improvements to our code saved us a lot of trouble in the long run, as we always made sure we had a strong foundation for our code. Additionally, by using Slack to communicate, we were able to work efficiently as a group, enabling us to follow the same design structure and avoid committing code segments that would not work together.

Another important lesson we learned while working on this project was the practicality of debugging as a group. After introducing compression and decompression to our implementation of DartSync, we ran into several bugs, some of which were incredibly arcane and had no immediate discernible cause. For example, when sending a compressed video, we ran into a strange bug where the age of the video (it was from 2004) actually proved to be the reason behind our code failing. By working as a group and bouncing ideas off one another, we were able to cover more possibilities faster, while also checking everyone's work to minimize the amount of bugs we encountered. If any one of us were to individually try and discover that, the time and stress involved would undoubtedly have been much higher.

7 Improvements and Future Applications

If given more time, there are several features that we could implement that would improve our version of DartSync. The first thing that we would implement would be delta updates; while we were partially successful in implementing delta updates during our project, we could not ensure a level of stability that we were satisfied with enough to commit to our final version. A stable version of delta updates would be a noticeable improvement to our project, as by only downloading portions of a file that changed rather than re-downloading the entire file, the entire application would be both faster and more efficient.

Secondly, we would implement a GUI for the user. During the course of the project, we attempted to implement a GUI using GTK+. However, due to some unexpected difficulties we faced in learning how to use GTK+ and compatibility with our code, we were also unable to create a stable final version that would have met our standards for the project. Given more time, we could either improve our implementation using GTK+ or explore an alternative way to create a GUI, such as PyGtk or Qt. Creating a GUI would greatly enhance the user experience, as it would be difficult to market an application without providing a reasonable amount of front-end for the consumer.

Thirdly, we would implement version control so users would have the option of revisiting previous version of files that they've downloaded. The premise for this would be similar to how we approached the design of our project. By maintaining stable versions of our code, we always had the option to revert to an older version if the newest version turned out to be buggy or inadequate in some way. Users would be able to access older versions of files in case there are problems with the newest available version.

Finally, we would improve our application's security by adding data encryption to the files and the password used for authentication, possibly by creating a public and private key hash. Doing this would better protect the user's data, which would incentivize use of our application and allow the user to send more sensitive information.

8 Acknowledgements

We would like to take a moment to acknowledge those who helped us during our time working on DartSync. Thanks to Professor Zhou and the CS 60 course staff for always being there and responding so diligently to questions over the past two weeks. Special thanks to Ethan Yu, who was an amazingly helpful and accessible shepherd (even during our late nights in Sudikoff). We couldn't have done it without you guys!