Computer Networks
DartSync Project Report
The C-Men Group
Justin Lee, Tianqi Li, Charles Li & Josh Kerber
2 June 2016

Contents:

Note: We followed the provided design document found at http://www.cs.dartmouth.edu/~xia/cs60/project/dartsync.pdf

I. Design Specifications

A. Tree Structure

Note: Test files and executables are omitted from this tree structure display.

```
.C-Men
└── dartsync
    ├── browser                 // web interface advanced feature
    │   ├── browse              // executable
    │   ├── file_browser.c
    │   └── Makefile            // compiling file browser
    ├── common
    │   ├── constants.h
    │   ├── fileMonitor.c       // file monitoring
    │   ├── fileMonitor.h
    │   ├── filetable.c         // functions to manipulate a file table
    │   ├── fileTableDemo.c
    │   ├── filetable.h
    │   ├── seg.c               // sending and receiving data
    │   └── seg.h
    ├── dartsync_folder         // all peers but the first to connect store files here
    ├── Makefile                // compiling DartSync
    ├── peer
    │   ├── dartsync_folder     // the first peer to connect holds its files here
    │   ├── DartSync_peer       // peer executable
    │   ├── p2p_client.c
    │   ├── p2p_server.c
    │   ├── peer.c              // peer functionality
    │   └── peer.h
    ├── README.md               // README
    └── tracker
        ├── DartSync_tracker    // tracker executable
        ├── tracker.c           // tracker functionality
        └── tracker.h
```

B. Data structures

- Used in `fileMonitor.c`

```
//File monitor struct
typedef struct monitor{
      int               isBlocked;        //block mark of the monitor
      int               isAlive;          //alive mark of the monitor
      time_t            refreshTime;      //last fresh time of the monitor
      file_table_t *    filetable;        //file monitor maintains a file
table
}Monitor;
```

1. File table design

The file table consisted of file entries and was stored as a linked list. The file table also contains the target directory name, the total number of file entry in the table, and a mutex. Each file entry has the name, size, timestamp, and the type (regular file or a directory) and owner of a certain file. We implemented function for manipulating the file table, described in section I.E. The basic functions including initializing, scanning the target directory and filling the file table, destroying, adding/deleting file entries, printing the file table etc. Since the file table will be transferred between peer and tracker, we implemented the function that will convert file table into array before sending out, as well as a function converting array back to file table after receiving.

- Used in `filetable.c`

```
//Determines file type
enum fileType{
    REGULAR,      //regular file
    DIRECTORY,    //directory
};

//Owners of a file in the file table
typedef struct peerListEntry{
    char                ip[IP_LEN];      //ip address of owner
    uint16_t            port;            //port number of owner
}peerEntry_t;
```

```
//File table struct
typedef struct fileTable{
    char                dirPath[MAX_NAME_LEN]; //name of target directory
    int                 nfiles; //number of files in the table
    pthread_mutex_t     fileTableMutex;  // mutex of the table
    struct fileEntry*   head; // head of the file list
    struct fileEntry*   tail; // tail of the file list
}file_table_t;


//Entry in a file table
typedef struct fileEntry{
    char                filename[MAX_NAME_LEN];      //name of file
    time_t              timestamp;       //last updated timestamp
    unsigned int        size;            //size of the file
    enum fileType       type;            //type of file
    int                 peerArraySize;   //how many peers have this file;
    struct peerListEntry owners[MAX_PEER_ENTRIES];    //owners of file
    struct fileEntry*   next;      //pointer to next entry in file table
}file_entry_t;

//File table converted to array of fileEntrySend structs before sending
typedef struct fileEntrySend{
    char                filename[MAX_NAME_LEN];      //name of file
    time_t              timestamp;       //last updated timestamp
    unsigned int        size;      //size of file
    enum fileType       type;      //directory of file
    int                 peerArraySize;   //amount of peers owning file
    struct peerListEntry owners[MAX_PEER_ENTRIES];    //owners of file
}file_entry_send_t;
```

- Used in `seg.c`

```
//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 : 0 register, 1 keep alive, 2 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 the local file table -- optional
    int file_table_size;
    // file table of the client -- your own design
    char file_table_array[MAX_TBLARRAY_LEN];
} ptp_peer_t;
```

```
//The packet data structure sending from tracker to peer.
typedef struct segment_tracker {
    // time interval that the peer should sending alive message periodically
    int interval;
    // piece length
    int piece_len;
    // file number in the file table -- optional
    int file_table_size;
    // file table of the tracker -- your own design
    char file_table_array[MAX_TBLARRAY_LEN];
      // for advanced feature
    char password[RESERVED_LEN];
} ptp_tracker_t;
```

- Used in `peer.c`

```
//Peer-side peer table.
typedef struct _peer_side_peer_t {
    //Remote peer IP address, 16 bytes.
    char ip[IP_LEN];
    //Current downloading file name.
    char file_name[FILE_NAME_LEN];
    //Timestamp of current downloading file.
    unsigned long file_time_stamp;
    //TCP connection to this remote peer.
    int sockfd;
    //Pointer to the next peer, linked list.
    struct _peer_side_peer_t *next;
} peer_peer_t;

//Structure to define the file information for the file monitoring system.
typedef struct {
    // Path of the file
    char filepath[200];
    // Size of the file
    int size;
    // time stamp of last modification
    unsigned long last_modify_time;
} FileInfo;

// struct to provide to individual P2PDownload threads
typedef struct p2p_info_structure {
    //TCP connection to LISTENING thread
    int sockfd;
    // for partition-file case
    int piece_id;
    // for partition-file case
    int range_start;
    int range_end;
    // file information
    char filepath[FULL_PATH_MAX_LENGTH];
    // size of the file
    int size;
    //last timestamp to maintain correct modification times
    int timestamp;
    // file name
    char filename[FILE_NAME_LEN];
} P2PInfo;
```

- Used in `tracker.c`

```
//Tracker-side peer table.
typedef struct _tracker_side_peer_t {
    //Remote peer IP address, 16 bytes.
    char ip[IP_LEN];
    //Last alive timestamp of this peer.
    unsigned long 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;
```

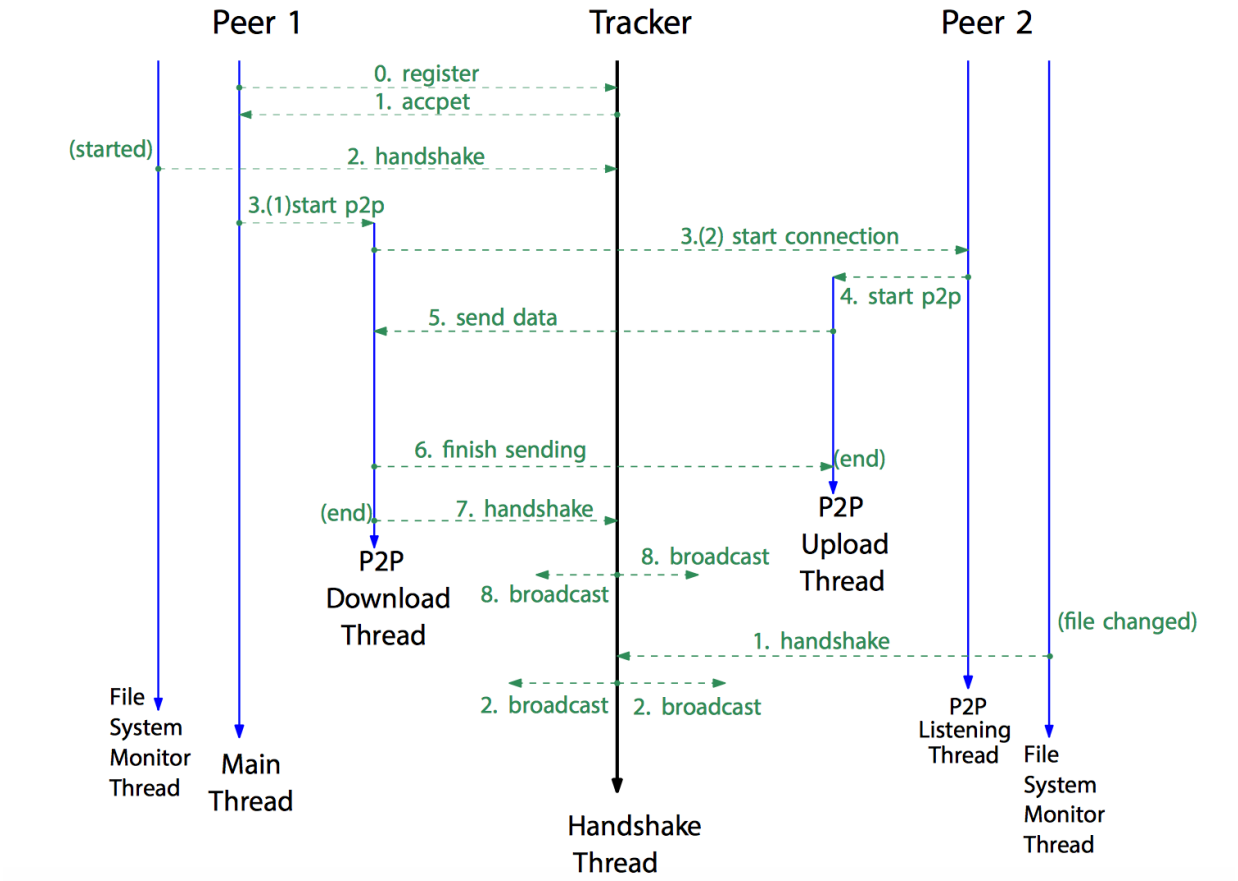
C. Data flow

Our DartSync data flow begins with the tracker being initialized. The first peer is initialized using files in the `./dartsync/peer/dartsync_folder` directory. The tracker will initialize its global file to match that of first peer. All succeeding peers that are initialized will store their files in the `./dartsync/dartsync_folder`, and will receive a file table from the tracker upon startup. As a file is added, modified, or deleted in a `/dartsync_folder` directory, the file monitoring system will detect that change and update the peer's file table who made the change accordingly. Then, the peer's table is sent to the tracker. Upon receiving a file table from the peer, the tracker will update it's global file to that of the one it received, then broadcast this table to each and every peer it is tracking. Upon receiving a new file table from the tracker, the peer will transfer (upload/download) files if need be using the peer-to-peer uploading and downloading threads described in section I.D.1. below. Constantly running in the background, the tracker is receiving keep-alive heartbeats from peers to ensure dead peer communications are closed.

D. Threads

As a group, we took advantage of all threads shown below. The details are as follows.

Note: The image below was taken directly from the provided design document found at http://www.cs.dartmouth.edu/~xia/cs60/project/dartsync.pdf



1. *Tracker Handshake*

After a tracker receives a connection from a peer and adds it to its current peer table, a handshake thread is started on the tracker side that acts as the packet handler for that specific peer. Upon receiving a packet from a peer, the tracker checks the type of the packet by checking `struct segment_peer -> int type;`.

If a packet is of type 0, this marks the initial register of a peer to the tracker. If this is the first time the tracker receives a register from any peer, the tracker will initialize it's file table to match that of the peers. If this isn't the first time the tracker has shaken hands with a peer (and thus the tracker-side file table is already initialized) then the file table received isn't acknowledge. In both cases, the tracker sends a packet back to the peer containing the heartbeat interval, piece length, the current tracker file table size, and the current tracker file table.

If the packet is of type 1, this marks a keep-alive segment or “heartbeat.” The tracker handles this by simply updating the peer’s timestamp in the current tracker peer table `struct _tracker_side_peer_t -> unsigned long last_time_stamp.`

If the packet is of type 2, this marks a file table update from the peer. The tracker then updates its file table to that of the one it received and broadcasts this new table out to all peers. This ensures a change on a file table in one peer will be acknowledged by all peers.

2. P2P Download/Upload

With regards to peer-to-peer uploading and downloading, our group adhered closely with the provided design spec. Specifically, we abided by the diagram depicted in the official design spec (Figure 1 in the document).

The step-by-step process is as follows:

1. Peer 1 starts a p2p transfer by initializing a P2P Download Thread. Peer 1, when initializing the P2P Download Thread, passes it a struct containing the file information. The P2P Download then forwards this requested downloading information to the P2P Listening Thread on Peer 2's side. This struct contains information such as the file name, the file size, the file modification time, etc.

2. Peer 2's P2P Listening Thread receives the above message from Peer 1's P2P Download Thread containing the struct of information. The Listening Thread then starts a P2P Upload Thread, passing in the same struct (except it modifies the socket file descriptor, because the new P2P Upload Thread will be communicating directly with the P2P Download Thread).

3. Once Peer 2's P2P Upload Thread is initialized and has the data about the file to send, it uploads the file, sending it over a socket connection to Peer 1's P2P Download Thread.

To enable smooth data transfer, we used the same idea as suggested in the earlier lab assignments during the term. That is, we had delimiters indicating the start of a data transfer and delimiters indicating the end.

4. The P2P Download Thread receives the file over the socket connection, downloading it and storing it locally in its own directory. Similarly, it receives data using the delimeter method described above, to help enable smooth data transfer.

Note that by starting threads for each file, our file transfer process is efficient. In other words, multiple files can be uploaded or downloaded concurrently.

We added logic to protect the code that assumed the upload/download was finished (ie. every file was uploaded or downloaded). To do this, we made sure threads had completed through control flow before proceeding to further code. This prevented typical issues that arise with threading such as race conditions.

3. Peer-Side Main

The main thread on the peer side first shakes hands with the tracker. The peer will receive a file table upon shaking hands, and will store it (unless it is the first incoming peer for the tracker in which the peer would've already allocated memory for a file table in the main method in `peer.c`). After the handshake, the peer-side main threads handles incoming packets from the tracker — all will be file table updates. The details of this update are in section I.E.

4. Heartbeat threads

The tracker is constantly receiving keep-alive heartbeats from peers' heartbeat threads upon which it will update the peer's latest timestamp in the tracker-side peer table. The tracker-side monitor alive thread checks timestamps every `HEARTBEAT_INTERVAL`, closing peer connections and removing peers from the peer table if the time since the latest time stamp exceeds `HEARTBEAT_INTERVAL`.

5. File System Monitor

The file monitor will periodically scan the target directory. It is implemented in a recursive way and can handle the subdirectory. Each time the monitor scans the directory, a new file table will be generated. The attributes of each file will be used to create a file entry and added to the file table. After a file entry is added to file table, the type of current file will be checked – if it is a directory, the function

will call itself recursively to check the subfolders; otherwise, it will continue to create file entry for the next file, until all files have been added to the file table.

The new file table will be compared with the old file table. We detect change by checking the timestamps of target directory in both versions. If they are different, it means there is some change in the directory. So we update the monitor's file table by destroying the old file table, and replacing with the new table. Then the new file table will be sent to the tracker.

E. Advanced features

1. *Raspberry Pi*

   We implemented our tracker functionality on a Raspberry Pi. The main problem with using a Pi as a tracker was the possibility of a tracker-side internet disconnect. This unreliability was handled by restructuring the main method of the peer. If a connection is lost with the tracker, instead of exiting the program, the peer will reset and offer the option of reconnecting to another tracker. At this point the tracker would be rebooted and would be able to reconnect with a peer, and the peer would reinitialize all of it's threads (besides its peer to peer threads that do not rely on tracker communication).

2. *Recursive subdirectory notice*

   **This advanced feature is described in section* I.D.5 *File System Monitor thread*

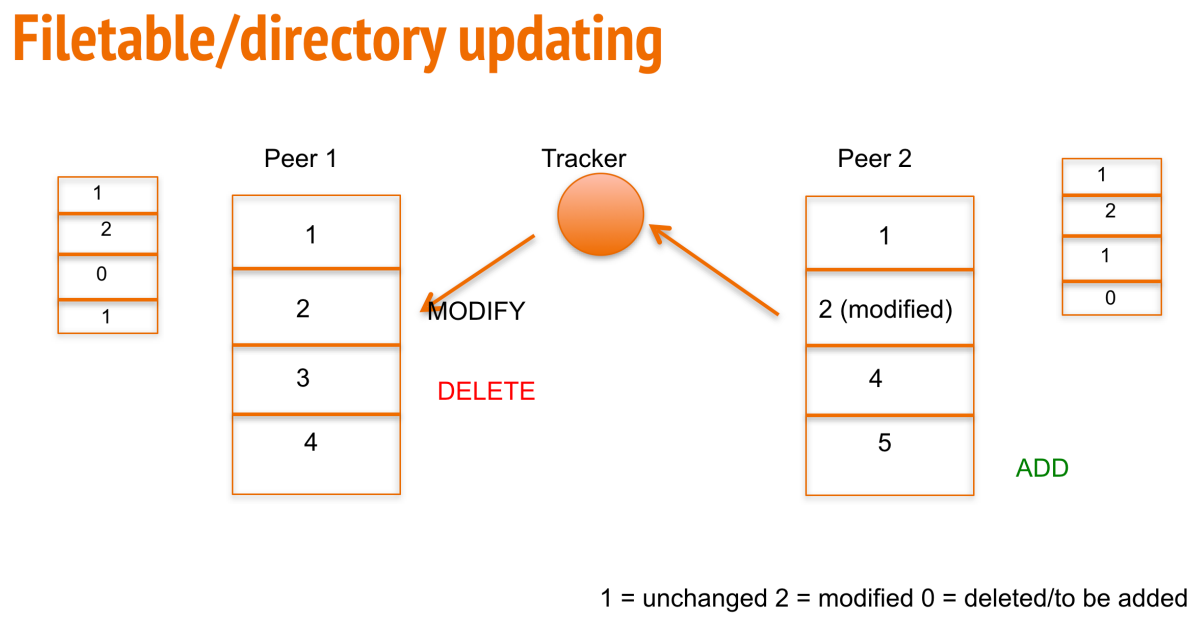
3. *Web interface*

   For this feature, we implemented a simple, iterative HTTP/1.1 web server to display the current directory of a selected peer. We used code similar to Lab 2 of this course, however we had to make modifications to handle sub-directories and omit logging browser history. This web interface displays directories and files and the user can explore the current files synced with DartSync. If there is an update, the user will see that as the files will change online as well upon a page refresh.

4. *Password protected*

We added password protection to DartSync by modifying our tracker segment. `struct segment_tracker -> char password[RESERVED_LEN];` was added. Upon starting the tracker, the user can enter a password for DartSync. Each and every peer that is started will be prompted for a password. Once connected, this password is verified by the peer by comparing it with the password received in the tracker handshake segment. If a false password is entered, the peer is terminated and the user can retry upon another executable run.

F. File table update algorithm



C-Men designed its own algorithm for a peer to update its file table upon receiving a file able from another peer. Pictured above, peer 1 is has just received a file table from peer 2. Our algorithm begins by initializing two integer arrays, one to mirror the file table the peer has just received, and one to mirror the peer's own current file table. Specifically, the integer array will have the same amount of slots as the file table it is mirroring, and the integer in each slot will determine the status of the file table entry for that index. Details to follow. Peer 1 will iterate through the table it received from peer 2, checking to see if the files in peer 2's file table entries exist it's own table. If peer 1 finds a matching file in peer 2's

table, peer 1's table array is marked with a 1 at that slot in which it is found. If the the file is found with a different timestamp, the array slot is marked with a 2, signaling a modification. If the file is not, it is marked with a 0. At the end of these iterations, a 0 in peer 1's mirroring array signals for the file to be deleted because it was not received in peer 2's table. A 0 in peer 2's mirroring array signals for the file to be downloaded because it is not currently in peer 1's table. Additionally, a 1 in both tables means an unchanged file, and a 2 in both tables means an existing file with modifications. The final step for the peer (in the above case peer 1) to iterate through the integer array mirroring its file table, deleting or updating any files if necessary. Lastly it iterates through the the integer array mirroring the received file table, adding or updating any files if necessary. The file updates are done using the peer upload and download threads — details in section I.D.2. above.

II. What we learned

- Comment as you code.

  Adding comments to our code as we went prevented any potential confusion.

- Handle error cases as they arise.

  It is easy to neglect defensive coding especially in the early phases. Our group agreed to code defensively as we went and not leave any error case behind. Specifically, we made sure to handle thread interferences in file monitoring portion of our code. As a file was being downloaded, the file monitoring system would detect several changes however this is obviously unnecessary until the downloading is complete. This was an easy fix by adding a simple integer value the file monitor struct to determine weather or not the file monitoring system is allowed to be detecting changes in the files.

- Communication is key.

  The potentially biggest key to C-Men's success was our ability to effectively communicate. The first thing we discussed as a group was how we were going to communicated

and how we were going to handle merge conflicts with our code. We landed on using Slack and GitHub. Our Slack chat was constantly being used to let other members know the status of our respective parts. If a user had a bug, a simple "Is anyone available to help debug?" went a long way. This ensured we finished this project in a timely manner.

- Be open to new ideas.

  Sometimes conversations amongst our group involved new ideas that pushed us out of our comfort zone. Fortunately, all C-Men enjoy challenges and are eager to attack new ways to solve problems. At times, the in-depth procedural details got confusing, however we were not discouraged. We took time to develop our algorithms and never backed down and took the easy way out, in this case being a simple, inefficient, elementary way to code DartSync. Without this drive, this project would've been less enjoyable, exciting, and challenging.

- Design first, code later.

  C-Men were eager to begin coding and jumped right into implementation. However, this proved to be ineffective as our peer-to-tracker communications had a few early bugs. We went back to the drawing board to construct our program, and this led to a jump in progress in the implementation phase. On a larger scale, coding is not just about the lines you right down. It is arguably more about how you are making lives easier, and what you are accomplishing with computer science. A good attack plan leads to getting the job done, in this case a real-time file sharing system.