Project Description - DartSync

CS60 - Spring 2015

May 14, 2015

We all enjoyed the use of Dropbox – we can have all our documents, photos, and videos available anywhere and anytime. All the files we store at Dropbox are automatically synchronized across all our personal devices. The advantages of using such cloud-based storage services are many. However, these advantages come with prices. By uploading our data to the centralized, third-party cloud, we are not only imposing a huge traffic burden to the network that connects mobile to the cloud, but also running the risk that our data, sometimes private data, can fall into the wrong hands. These unwanted data leaks can be caused by secretive trading of our data for profits, or accidental mis-management inside the cloud, or malicious outsider attacks.

In this project, we explore a completely different paradigm for synchronizing our personal data across different devices. 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".

Specifically, we are implementing a system called DartSync, which is an application that runs on a local always-on node as a tracker, and a set of our personal devices as peers. Users can specify files that they want to synch across devices, and DartSync will synchronize these files across devices upon file update, addition, or removal. Ideally we would like to have you implement the tracker on a Raspberry Pi[1] node, a credit-card size Linux machine supporting Python and C. We choose Pi to implement the tracker because it entails low power to stay always on. We do recognize, however, the possible stability issues of Pi, so we do not put implementation on Pi as a requirement. If you can implement your tracker on Pi, you will gain extra credits. The peer nodes can be our laptops, or the Linux machines in Lab 001, or even smartphones – it really up to you in terms of how you want to extend your code to different devices. We do not have a strict requirement here, except that you should be able to demo the synchronization across no less than 4 devices.

As the following, we will describe the project structure and requirements, and provide some design details for your reference.

1 Project Structure

DartSync runs on a tracker (server) and multiple peers (clients). You may run it with different argument for different node type: ./DartSync -tracker or ./DartSync -peer.

1.1 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:

[1] `http://www.raspberrypi.org/`.

- *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.

1.2 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 if 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.

2 Requirements

Unlike our previous assignments where we give you well-defined code structure and you mainly fill in the implementation of specific functions, the class project is more open, where you have the freedom to design the code structure and implement functions as you wish. That said, your code needs to realize the following features:

1. C implementation that can run in any Linux system.
2. TCP connection and data transfer.
3. Local file monitoring.
4. Synchronization of multiple files by comparing file timestamps. Synchronization of multiple folders is not required.
5. File replacement when updating files.
6. Fetching data from multiple peers.

Here is the list of features that you can implement to gain extra edits:

1. Implementing the tracker on Raspberry Pi.

2. Synchronization of multiple file folders.
3. Incremental (delta) file update where only the differences rather than the whole file are transmitted.
4. Resuming from partial download.
5. Resolving conflict.
6. Supporting other OS platforms such as Mac and Windows, and other devices such as Android smartphones or iPhones.
7. Any other cool useful features that you can think of...

3 Reference Design

In this section, we provide the design detail for you to get started. But do not feel obligated to strictly follow the design. Any design that meets the requirements will be accepted. We will go over the threads, transport protocol, important data structures, and design components.

3.1 Threads

Figure 1 shows the major threads running on the tracker and peers and their interactions. In particular, three threads are running on the tracker node:

- **Main thread**: listen on the handshake port, and create a Handshake thread when a new peer joins.
- **Handshake thread**: receive messages from a specific peer and respond if needed, by using peer-tracker handshake protocol that we will describe in Section 3.2.
- **Monitor alive thread**: monitor and accept alive message from online peers periodically, and remove dead peers if timeout occurs.

Peers run the following threads:

- **Main thread**: after connecting to the tracker, receive messages from tracker, and then create P2PDownload Threads if needed.
- **P2P listening thread**: listen on the P2P port; when receiving a data request from another peer, create a P2PUpload Thread.
- **P2PDownload thread**: download data from the remote peer.
- **P2PUpload thread**: upload data to the remote peer.
- **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.
- **Alive thread**: send out heartbeat (alive) messages to the tracker to keep its online status.

3.2 Peer-Tracker Protocol

Peer-Tracker Protocol (PTP) defines the structure of packets sent between peers and the tracker. A peer needs to inform the tracker when it logs in the system for the first time. It also needs to send keep-alive messages periodically, and send local file table when necessary. On the other hand, the tracker provides information for peers to set up, and broadcasts latest file table. The most important part here is how to define and send the file table, whose structure is left for you to design. In this section, we will cover the protocol in details.

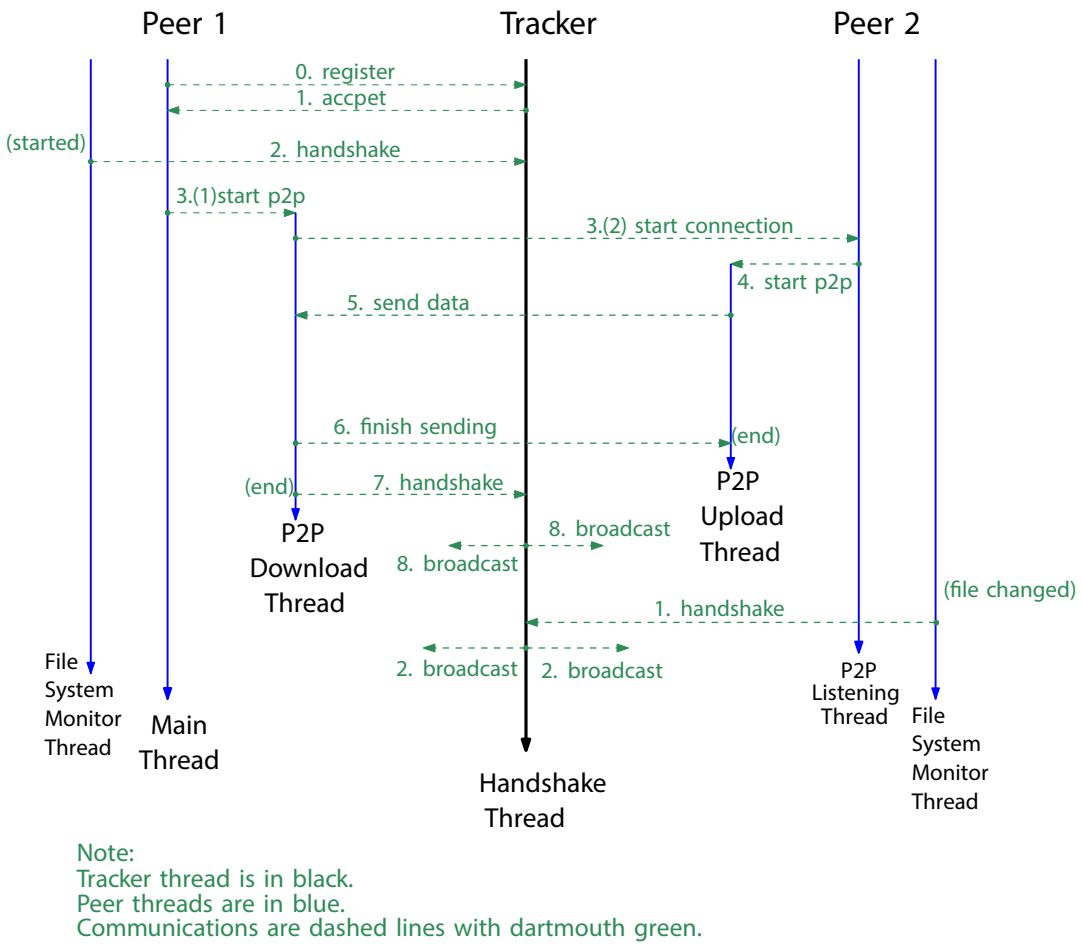

Figure 1: Threads and their interactions in DartSync.

3.2.1 Protocol

- **Peer to Tracker**

Protocol Length	Protocol Name	Request Type	Reserved	Peer IP	Listening Port	Files

Protocol Length: protocol length.
Protocol Name: protocol name.
Reserved: reserved space for the further extensions.
Request Type: type of the packet: 0 –REGISTER 1 –KEEP ALIVE 2 –FILE UPDATE.
Peer IP: IP address of the peer.
Listening Port : p2p listening port number.
Files: file table of the peer.

- **Tracker to Peer**

Interval	Piece Length	Files

Interval: interval between two consecutive keep-alive messages.
Piece Length: length of a piece of the file
Files: file table of the server.

When a peer first logs in the system, it should send a REGISTER packet to the tracker to notify its existence. `Files` field in a REGISTER packet should be null. Upon receiving a REGISTER packet, the tracker sends back a packet informing the `Interval` and `Piece Length` for the peer to set up.

A peer needs to send a KEEP ALIVE packet periodically according to the interval returned by the tracker. Moreover, a FILE UPDATE packet must be sent upon local file changes. Once there is a file change/update in the local directory, the peer constructs a packet containing the updated local file table and sends it to the tracker. Once the tracker receives a FILE UPDATE packet from a peer, it compares the received file table with the one it already has. If an update is necessary, the tracker will broadcast the updated file table to all peers currently alive.

3.2.2 Data Structures

We give out a template of the data structures in this protocol. The ptp_peer_t and ptp_tracker_t data structures illustrated in the following correspond to the protocol messages described above.

```
/* 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 the local file table -- optional
  int file_table_size;
  // file table of the client -- your own design
        file_t file_table;
}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
  file_t  file_table;
} ptp_tracker_t;
```

For the file table, consider maintaining the following information: **File Name, Timestamp, File Size,** and **Peers**. The **Peers** field is a set of peers that have a certain file.

3.2.3 Interfaces

Two groups of functions you need to pay attention to:

- **Constructing packets:** It is better to have some init functions to initialize a packet, and setting functions to fill up the packet.
- **Sending and receiving:** To design the sending and receiving functions, you need to think about the data structure of the file table. If you use pointers, then you need to be careful about the packet length. If you use arrays, you would need some packing and unpacking functions to construct the array and extract the file information from an array.

3.3 Peer-side and Tracker-side File Table

On the peer side, there is a file table to record the information of all the files in the monitored directory. You can implement the file table as a linked list to handle the dynamics of file changes.

```
//each file can be represented as a node in file table
    typedef struct node{
      //the size of the file
      int size;
      //the name of the file
      char *name;
      //the timestamp when the file is modified or created
      unsigned long int timestamp;
      //pointer to build the linked list
      struct node *pNext;
      //for the file table on peers, it is the ip address of the peer
      //for the file table on tracker, it records the ip of all peers which has the
      //newest edition of the file
      char *newpeerip;
   }Node,*pNode;
```

When the File Monitor (described in Section 3.6) on a peer detects any file change, the file table should be updated. There are at least three functions to maintain the linked list: add, delete, modify.

Add: A new file is added to the folder, so a node is added to the linked list.

Delete: A file is deleted from the folder, so the corresponding node is deleted from the linked list.

Modify: The context of a file is changed, then find the node by its file name in the linked list and update the attributes.

Based on the monitoring results from the File Monitor, the peer calls the corresponding function.

On the tracker side, the file table is global. This means that the File Table stores the union of all the file tables in all peers. The size, name should be the same with the latest version and the newpeerip includes the IPs of all peers that store the latest files. When a peer sends its local file table to the tracker, the global file table on the tracker should be updated if necessary. The tracker uses the time stamp of a file to decide whether it needs to update this file's entry in the file table.

When the tracker sends back the global File Table to a peer, the local files on the peer will be synchronized. The peer should delete the files that are no longer in the global file table.

3.4 Peer-side and Tracker-side Peer Table

On the peer side, there should be a peer table that records the ongoing downloading tasks. This helps avoid the potential duplicated downloads. While on the tracker side, the peer table stores the current online peers

for the purpose of broadcasting. Both tables are implemented using linked list.

In the peer-side peer table, each entry represents an ongoing downloading task. A peer can download different files from the same remote peer at same time in different P2P threads. A peer can download single file form multiple remote peers at same time in different P2P threads. A peer CANNOT download a single file from a single remote peer in multiple P2P threads simultaneously. Here is a template for the peer-side peer table.

```
//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;
```

In the tracker-side peer table, the tracker will update the related entry (according to the peer's IP address) after receiving a keep-alive message from this peer (OPTIONAL: also update this table after receiving handshake messages from the peer). If a peer does not send keep-alive messages during a certain time interval (*e.g.* 10 minutes), the tracker removes this peer from its peer table. Here is a template for 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;
```

3.5 P2P File Transfer

While peers communicates with the tracker to obtain information about the latest versions of files, the actual file transfer happens among peers themselves. During a P2P file transfer, a peer is both an uploader and a downloader. We cover the general design of the P2P file transfer in this section.

3.5.1 Uploading

Each peer could be a potential uploader. Therefore, when a peer starts to execute, it will start a listening thread for other peers to connect to. Whenever another peer connects to it, the listening thread will start an uploading thread, dealing exclusively with this other peer.

3.5.2 Downloading

Each peer could be a downloader. After parsing responses from the tracker, a peer will connect to other peers according the tracker's response in order to request the latest versions of files. For each peer in the

tracker's response, the downloading peer will connect to it and create a downloading thread, requesting file data exclusively from this peer.

3.5.3 File Transfer

Requesting file data from multiple peers can boost download speed. In order to achieve this, we need to partition a file into pieces, and allow a peer to download different pieces from multiple peers and finally combine these pieces into a whole file. Specifically a piece represents a range of data. The downloader will send a request to the uploader with a piece number. Then, the uploader will choose the corresponding range of data from its memory and send that data back. The downloader will order the received pieces based on their piece numbers and store them in the buffer. When all pieces are received, the data in the buffer is exactly the same as the requested file. Recall that the piece length is retrieved from the server. The last piece might not be a full piece.

3.6 Local File Monitor

The File Monitoring module monitors file system activities. For simplicity, you can implement monitoring only one directory without recursion on subdirectories. You can gain extra credits by implementing recursive sub-directory notice. This module can detect file changes including additions, removals, and updates. Other modules will be notified upon file changes.

3.6.1 Implementation

Here are two options to implement this module:

- Using provided third-party libraries/APIs. Windows and Linux both have libraries that can notify applications about the events of your specified directory and sub-directories. For example, on Linux, Inotify (inode notify) is a Linux kernel subsystem that extends the file system to notify any file changes, and reports those changes to applications in real time. To use the corresponding functions, you will need to get familiar with the library first.
- Recording file information in your program. You periodically check the directory to detect changes of file information (file names and time stamps). The advantage of this approach is that you have complete control. But you might have to code more and not be able to know the changes in real time.

3.6.2 Data Structure

Here is an example data structure to define the file information:

```
typedef struct {
  filepath          // Path of the file
  size              // Size of the file
  lastModifyTime    // time stamp of last modification
} FileInfo;
```

3.6.3 Interfaces:

The following functions can be viewed as interfaces provided for other modules. By calling these functions, other modules can interact with the File Monitor.

1. watchDirectory(char* directory) //Begin to watch a directory.
2. readConfigFile(char* filename) //Read config file from disk.

3. fileAdded(char* filename) //Called when a file is added to the directory.
4. fileModified(char* filename) //Called when a file is modified.
5. fileDeleted(char* filename) //Called when a file is deleted.
6. getAllFilesInfo() //Get all filenames/sizes/timestamps in the directory
7. getFileInfo(char* filename) //Get information of a specific file
8. freeAll() //Free all memory.
9. blockFileAddListenning() //Prevent unnecessary alert when downloading new files
10. unblockFuleAddListenning() //Unblock it.
11. blockFileWriteListenning() //Prevent unnecessary alert when modifying a file
12. unblockFileWriteListenning() //Unblock it.
13. blockFileDeleteListenning() //Prevent unnecessary alert when deleting file
14. unblockFileDeleteListenning() //Unblock it.

For function 2, your configuration file should have at least the information about which directory you want to monitor, thus changing the directory does not mean changing the code and recompiling everything.

Function 3, 4 and 5 are three functions that will be called when the corresponding events happen. We recommend implementing them as function pointers, so that other modules can define their own functions to handle these events by binding their routines to these functions.

From 9 to 14, these functions will be called when other modules are not supposed to be notified about the file change events. You might run into some cases when you find these operations are desirable.

You can also define an interface for file name change event. This will be tricky. So if you are not sure about implementing this interface, you can view file name change event as file deletion event followed by a file creation.